Requirements Specifications and Recovered Architectures as Grounded Theories

Daniel M. Berry, University of Waterloo, Michael W. Godfrey, University of Waterloo, Ric Holt, University of Waterloo, Cory J. Kapser, Mobile Data Technologies, Isabel Ramos, University of Minho

Abstract

This paper describes the classic grounded theory (GT) process as a method to discover GTs to be subjected to later empirical validation. The paper shows that a well conducted instance of requirements engineering or of architecture recovery resembles an instance of the GT process for the purpose of discovering the requirements specification or recovered architecture artifact that the requirements engineering or architecture recovery produces. Therefore, this artifact resembles a GT.

Introduction

The purpose of this paper is to show that well conducted instances of two different activities in Software Engineering, *requirements engineering* (RE) and *architecture recovery* (AR) resemble grounded theory (GT) processes. Each verifies the power of the classic GT process, as discovered by Glaser and Strauss (1967), to identify what is happening in a practical situation, producing a working GT of the requirements or architecture of a system. The aim is to point out some striking similarities between the classic GT process and software engineers' approaches to requirements engineering and architecture recovery, thus demonstrating how requirements engineering and architecture recovery practitioners might be producing working GTs.

The purpose of requirements engineering is to use whatever data are available, from documents to spoken words, to construct a *requirements specification* for a software system. The purpose of architecture recovery is to use whatever data are available, from existing code and documentation to spoken words, to construct a *recovered architecture* for an existing software system. This paper is *not* trying to invent a new form of the GT process, but is simply showing, by appeal to a description of the classic GT process, that what software engineers are doing in either of these two specific cases amounts to a GT process and that the artifact produced, a requirements specification or a recovered architecture, resemble a GT.

Section 2 describes the classic GT process and its resulting working GTs. Section 3 argues that two activities in Software Engineering, Requirements Engineering and Architecture Recovery, are GT processes. Section 4 describes related work, and Section 5 concludes the paper.

In what follows, an arbitrary GT process practitioner is without loss of generality assigned the male gender and an arbitrary requirements or architectural analyst is without loss of generality assigned the female gender. Note also that architecture recovery is a major and essential component of reverse engineering, whose common acronym, "RE" is identical with the acronym used for "requirements engineering". However, reverse engineering includes steps that are not considered in this paper and is thus regarded as outside the scope of this paper.

2 Grounded Theory

The classic GT process is a method for developing grounded theories (Glaser & Strauss, 1967; Glaser, 1992), each of which is a theory about a named pattern of human behavior. In the 1960s, discomfort was growing with the application of traditional statistical methods to understanding and explaining social phenomena. The GT process was developed in response to this discomfort, and its purpose is to provide a means to gather detailed empirical evidence for theory that could be later subjected to traditional statistical empirical validation using controlled experiments or other means. The GT process is an adaptive research process for finding emergent theory that could not be anticipated in advance of the research. The researcher adapts the research process based on what he has learned from the data he has seen so far in order to pursue data that support the emergent theory. Therefore, not only is the theory emergent, but also the process and the set of data that are sought are emergent, as the researcher learns more and more about the phenomena involved and, thus, what data should be sought. Glaser (1992) says that *everything* is potentially data to the GT process practitioner.

The steps of an instance of the GT process are:

1. **Data collection**: collecting data about the phenomena to be modeled from a representative population,
2. **Coding**: coding the data in order to understand and categorize them,
3. **Sampling**: sampling the data by focusing on some categories,
4. **Memoing**: recording the data about categories found to be important into memoranda,
5. **Sorting**: sorting the memoranda by categories, and
6. **Writing up**: writing up the hypotheses that have been developed.

In remodeled versions of the classic GT process, Brower and Jeong (2008) provide more detailed kinds of coding, and Dick (2005) adds a note-taking step between Items 1 and 2.

Steps 2 through 4 repeat until a core category and a set of interrelated hypotheses deemed worthy of testing empirically are formed. While the steps are numbered in a particular order—the order even makes sense, because nothing can be written up until there is something to be written up—the reality is that dynamism reigns. In the middle of doing one step, one might see the opportunity for information requiring initiation of a different step. Hence, the steps can and do happen simultaneously.

A GT process practitioner immerses himself in an instance of the method, observing, with as little prejudice as is possible, what is happening, and drawing conclusions supported by his ongoing observations. Ideally, the GT process practitioner should begin the GT process with no hypotheses that he hopes to prove, in order to avoid being swayed (1) into seeing things that are not there and (2) into missing things that are there. In reality, totally avoiding opinions is impossible, but he should be aware of the opinions he does form, in order to keep himself honest. Moreover, he must clearly state his opinions in any write-up so that others can understand from where his decisions came (Walsham, 1995).

3 Requirements Engineering and Architecture Recovery as GT Processes

Software Engineering concerns itself with methods and processes for the development of software-intensive computer-based systems (Sommerville, 2007), hereinafter called "programs".

Requirements engineering is the discovery and construction of requirements for a program that a client needs and wants from the very incomplete and inconsistent information provided by the client and the client's employees and associates, who will probably be the program's users (Robertson & Robertson, 2006; Gause & Weinberg, 1990). Indeed, this information may be so incomplete and inconsistent that the requirements engineering effort may include determining *what the problem is* that the program is supposed to solve, particularly if the problem is wicked (Rittel & Webber, 1973). It is generally not clear up front what information in addition to that supplied by the client will be needed. Thus, requirements engineering may include significant unstructured information gathering from the client's organization, including research into the problem itself. The sources of information can be any of the following:

- written documents,
- questionnaires,
- conversations with clients and users,
- interviews of clients and users,
- brainstorming sessions with clients and users,
- focus groups with clients and users,
- developing scenarios (storyboards) with clients and users, and
- walking through prototypes with clients and users, and
- even inventive inspiration,

about the way the problem is solved now, about the future program, or both. It is understood in requirements engineering that requirements are both discovered, by elicitation, and constructed, by invention (Robertson & Robertson, 2006; Gause & Weinberg, 1990). In other words, as with the classic GT process, *everything* about the problem or program is potentially useful information.

Architecture recovery occurs much later in a program's lifecycle, after it has been deployed for long enough that many of the original developers are no longer around or have forgotten many details that drove the original development, including the program's underlying architecture (Chikofsky &, Cross, 1990) and the rationale for it. If the program must now be changed in some way, the changes must respect the forgotten architecture. Therefore, it is necessary to recover the program's architecture and the rationale for the architecture from a detailed and thorough examination of the program's code and any other available related artifacts. This recovery is very much detective work, relying on intuition and experience about how code, in general, works and some lucky discoveries. The sources of information can be any of the following:

- the current and past versions of the code,
- comments in the current and past versions of the code,
- documentation about the current and past versions of the code,
- interviews and conversations with current and past designers and developers, and
- e-mail messages sent during current and past work on the program,

whether correct or not. Here again, as with the classic GT process, *everything* about the current and past versions of the program is potentially useful information.

It has occurred to us that:

- requirements engineering can be done in a way that resembles using a classic GT process to discover and construct requirements of the program that its client needs and wants, and
- architecture recovery can be done in a way that resembles using a classic GT process to discover and reconstruct the architecture of the program being examined.

A consequence of this observation is that

- the requirements specification that results from a requirements engineering effort resembles a GT, and
- the recovered architecture that results from an architecture recovery effort resembles a GT.

The classic GT process steps can be applied directly to requirements engineering and to architecture recovery. All that are changed are the subjects examined and the artifacts produced. As with any other GT construction effort, it is best that the requirements analyst or architectural analyst avoid having preconceived ideas of the outcome.

3.1 Requirements engineering as a GT process

Requirements engineering has as its purpose to discover requirements for a program to be built by developers at the behest of a client for the benefit of users (Robertson & Robertson, 2006). In requirements engineering for a program, the requirements analyst initially has a vague notion of the program's requirements, i.e., what the program is supposed to do. By reading requests for proposals, vision documents, and other written materials supplied by the client of the program, by talking with the client, users, or both, of the program, the requirements analyst begins to build a mental model of the program to be built. Each mental model must be both validated and refined by asking questions of the client and users. The questions that are asked at any time are derived from the mental model that has emerged so far. That is, the requirements analyst asks follow-up questions to clarify what he has learned already and to test emerging hypotheses.

While the typical requirements analyst may not specifically follow the six steps of the GT process, she normally does every step in some form, possibly in a different order and possibly in parallel, *as is allowed in the classic GT process*.

The requirements engineering variants of the steps of the GT process are:

1. **Data collection**: collecting requirement ideas from (1) a request for proposals; (2) vision documents; (3) interviews of clients and users; (4) client and user reactions to draft scenario descriptions, draft requirements specification sections, models, prototypes, etc.; (5) etc.,
2. **Coding**: (1) classifying requirements as functional or nonfunctional; (2) ranking requirements by necessity, desirability, feasibility, costs, etc.; (3) determining stakeholders affected by and affecting each requirement; (4) clustering requirements into feature groups; (5) etc.,
3. **Sampling**: asking customers and users follow up questions about the various codings of requirements ideas,
4. **Memoing**: writing stories, scenarios, requirements specification sections, etc.,
5. **Sorting**: sorting the memoranda by categories, and
6. **Writing up**: writing up the final requirements specification.

Thus, the resulting requirements specification, which is a reflection of human-made decisions about the expected behavior of a program that meets human needs, *is* the working GT. This requirements specification may take any of several possible forms, including those of a formal specification written in some mathematical notation (Bowen, 1996), an IEEE-standard Software Requirements Specification (SRS) written in mostly natural language (IEEE, 1998), and a preliminary user's manual written in mostly natural language (Berry, Daudjee, Dong, Fainchtein, Nelson, Nelson, & Ou, 2004).

Recall that the GT process provides a way to gather detailed empirical evidence for theory that could be subjected later to traditional statistical validation using, e.g., controlled experiments. There is a correspondence to even this follow-up experimentation in requirements engineering! Very often, a prototype or early version of the program under development in a requirements engineering effort is subjected to usability studies. Some of these studies are conducted as controlled experiments. Even if there are no usability studies, no matter what, the final program is subjected to the most externally valid experiment possible, albeit possibly not controlled, of its acceptability to users: deployment among users, for bespoke software, or release to the market, for mass-market software. The lack of controls in deployment or release experiment is irrelevant, because the purpose of controls is to ensure that the small sampling of a normal experiment reflects the real world. A deployment or release *is* the real world.

With the application of the GT process, requirements engineering for a program becomes an interpretive and collaborative effort to develop a contextual and in-depth working GT about the program that a client needs and wants. The program's requirements should be constructed jointly by the developers and the client and users so that the clients and users will be motivated to support and use the program when it is finally built (Ramos, 2000). As with any other GT, this working GT must be validated. This validation consists in having the client and the users accept the requirements specification as specifying their collective requirements. Generally the client and users participate in a walkthrough of the requirements specification during which users' scenarios are exercised according to the specifications to see if what is specified is what the client and users want.

3.2 Architecture recovery as a GT process

Architecture recovery has as its purpose to determine a useful and reasonable model of the software architecture of an existing program (Chikofsky & Cross, 1990). Although architecture recovery is sometimes called "architecture extraction," that term is misleading, in that an explicit architectural model of a program commonly exists neither in the actual program nor in its documentation. Moreover, the architecture often does not exist even in the minds of the developers. Architecture recovery typically begins by searching for hints or descriptions of the architecture, such as might exist in any documentation of the program. Often, no such or poor documentation exists. The search may include interviewing any of the program's software architects and developers that are still available and other key stakeholders. The source code of the program may be analyzed manually, using fact extractors that automatically create a graphical representation of the code, or both.

The architectural analyst carrying out this analysis generally begins understanding neither the target architecture nor the best way to discover this architecture. Rather, she follows what is essentially a classic GT process. She gathers more and more data about the program and develops, in an emergent fashion, what is hoped to be an increasingly useful and detailed model of the architecture of the program (Holt, 2002). Involving developers in the recovery helps in two ways: The developers can provide intimate knowledge of the implemented program and at the same time, can direct the creation of a model that is more likely to be useful to the developers. As the architecture recovery proceeds, the analyst makes decisions on the fly, (1) that modify what she is doing to deal better with the data gained so far and (2) that refine the emergent model of the program's architecture.

The architecture recovery variants of steps of the GT process are:

1. **Data collection**: (1) collecting any reports that may document the program's architecture or aspects of it; (2) interviewing key stakeholders about the architecture; (3) inspecting the source code, manually or with tool support; (4) interacting with the running program, often using an interactive debugger or other instrument; (5) etc.
2. **Coding**: classifying collected information as essential or coincidental to the architecture, determining aspects of the program which have importance to the stakeholders and to the architecture, preliminary division of the program into upper level subsystems, etc.
3. **Sampling**: (1) probing the source code, or any preliminary graph model, to see if any proposed decompositions are reflected in the actual implementation; (2) asking stakeholders if a proposed decomposition is useful and intuitive; (3) etc.
4. **Memoing**: (1) writing up preliminary descriptions of modules or components; (2) preparing preliminary diagrams of module or component interactions, as determined thus far; (3) etc.
5. **Sorting**: (1) collecting and sorting the various data, descriptions and diagrams, along with collected motivations, toward formulating an model of the overall architecture; (2) etc.
6. **Writing up**: writing up a description of a determined model of the architecture, including motivating rationale, top-level decomposition into subsystems, description and documentation of those subsystems, and further descriptions and decomposition as appropriate to the program.

Thus, the recovered architecture, which is a reflection of the human-made architectural decisions made during the initial construction and at each modification thereafter, *is* the working GT. The recovered architecture may take any of several possible forms including that of a collection of diagrams and code fragments, accompanied by a natural language description (Bachmann, Bass, Carriere, Clements, Garlan, Ivers, Nord, & Little, 2000), with the diagrams in the form of UML class or object diagrams (Booch, Jacobson, & Rumbaugh, 1998).

Architecture recovery is, therefore, a collaborative effort for developing a working GT about the architecture of a program. Some elements of this working GT, e.g., the code facts, are discovered by examining the program, and some other elements, e.g., the architecture, are constructed by thinking about the discovered facts. This working GT must be validated by showing the recovered architecture to all of the code's developers that are available for consultation.

As mentioned, the main purpose of recovering an architecture for a program is to be able to make needed modifications to the program. The recovered architecture tells the modifying developer where, in the program's code, the changes need to be made. A very effective validation of the correctness of the recovered architecture is that the modifications proceed straightforwardly.

4 Other Work

The GT process has been used extensively to develop theories explaining social behaviors of all kinds (e.g., Glaser & Strauss, 1967; Glaser, 1992; Jeong, 2006; Pershin, 2006), even in technical disciplines such as software engineering (e.g., Walsham, 1995; Carver, 2007; Coleman & O'Connor, 2006 & 2007; Hoda, Noble, & Marshall, 2010; Adolph, Hall, & Kruchten, 2011), requirements engineering (e.g., Calloway & Knapp, 1995; Johansson &

Timpka, 1996; Galal & Paul, 1999; Ramos, 2000; Galal, 2001; Power, 2002; Lang & Fitzgerald, 2007; Breaux & Antón, 2008), and architecture recovery (e.g., Sillito, Volder, Fisher, & Murphy, 2005; Briand, 2006; Kapser & Godfrey, 2006; Sillito & Wynn, 2007). We call these uses of the GT process *methodological uses* because they study methods.

While there is much empirical work, including using the GT process, about requirements engineering and architecture recovery methods, in order to understand requirements engineering and architecture recovery, to the authors' knowledge, there is very little other work that specifically describes either requirements engineering or architecture recovery as an empirical method itself. For example, Galal and Paul (1999) describe one part of requirements engineering as a GT process when they presented GSEM (Grounded System Engineering Methodology), a grounded analysis method for "developing qualitative scenarios against which statements of requirements can be evaluated". Gold and Bennett (2002) offer Hypothesis-Based Concept Assignment as a way of assigning meaning to code fragments by pairing concepts, i.e., meanings, with indicators, i.e., evidence in source code. The discovery of an indicator serving as evidence for a concept is called a hypothesis. It is not unreasonable to view this hypothesis generation as another instance of the GT process in architecture recovery. Weber (2010) used the GT process in order to determine the set of typical users for the privacy-and-security relevant portions of arbitrary CBSs from quotations gathered during interviews of 32 such users. She identified five different types of users and describes each as a persona. The set of personas are intended to inform requirements engineering for the privacy-and-security relevant portion of any program to be developed. That is, requirements analysts internalize the specifications of the personas in order to be able to answer questions that arise during requirements analysis without having to keep a set of users continuously available for questions during the analysis. Teixeira, Ferreira, and Santos (2010) describe as a GT process the data collection part of the user-centered requirements engineering that they did for a Web-based information system for managing the clinical information in hemophilia care.

5 Conclusions

Requirements engineering for a program can be viewed as a GT process for the purpose of discovering the program's requirements, and architecture recovery for a program can be viewed as a GT process for the purpose of discovering the program's architecture. In brief, the GT process provides a systematic description of the activities of requirements engineering and architecture recovery, which might otherwise seem to be random searches. Consequently, the requirements specification emerging from a requirements engineering effort or the recovered architecture emerging from an architecture recovery effort resembles a GT and must be subjected to validation in a manner appropriate for the artifact.

The emergence of the information that requirements engineering or architecture recovery normally finds is consistent with considering requirements engineering and architecture recovery as GT processes. In each of requirements engineering and architecture recovery, not only is the final product of the activity emergent, but also the way in which the final product emerges is emergent. This observation says that any attempt to standardize requirements engineering or architecture recovery methods is unlikely to succeed.

That being said, it should be emphasized that the artifacts produced by requirements engineering and architecture recovery efforts are not GTs as defined by GT academics. Neither requirements engineering nor architecture recovery practitioners work under the banner of a classic GT process. Important aspects of classic GT generation, such as the constant comparative method, conceptualization, and the interchangeability of indicators have not been discussed in this paper. However, we found it a valuable exercise to compare similarities between the classic GT process and the problem solving that occurs in software engineering's requirements engineering and architecture recovery.

Author Berry has often said in his requirements engineering courses that each problem seems to beget its own requirements engineering method. Certainly, he never predetermines how he will discover any particular client's requirements. He listens and adapts his methods to the emerging situation. Our reading of the requirements engineering textbooks by Gause and Weinberg (1990) and by Robertson and Robertson (2006) suggests that each of these authors operates in the same way. Cockburn (2000) agrees for the entire lifecycle, not just requirements engineering.

Acknowledgements

The authors thank Ralph Brower for correcting our misconceptions about what is a process and what is a theory.

References

Adolph, S., Hall, W., & Kruchten, Ph. (2011). Using grounded theory to study the experience of software development. *Empirical Software Engineerin*g 16, 487–513

Bachmann, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R., & Little, R. (2000). Software Architecture Documentation in Practice: Documenting Architectural Layers. CMU/SEI-2000-SR-004. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, http://www.sei.cmu.edu/reports/00sr004.pdf

Berry, D.M., Daudjee, K., Dong, J., Fainchtein, I., Nelson, M.A., Nelson, T., & Ou, L. (2004). User's Manual as a Requirements Specification: Case Studies. *Requirements Engineering Journal* 9, 67–82

Booch, G., Jacobson, I., & Rumbaugh, J. (1998). *The Unified Modeling Language User Guide.* Addison-Wesley Longman, Reading, MA, USA

Bowen, J. (1996). *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press, New York, NY, USA, also at http://www.zuser.org/zbook

Breaux, T.D., & Antón, A.I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering* 34, 5–20

Briand, L.C. (2006). The experimental paradigm in reverse engineering: Role, challenges, and limitations. In: *Thirteenth Working Conference on Reverse Engineering (WCRE),* 3–8

Brower, R.S., & Jeong, H.S. (2008). Grounded analysis: Beyond description to derive theory from qualitative data. In Yang, K., Miller, G.J., eds.: *Handbook of Research Methods in Public Administration.* Boca Raton, FL, USA, Auerbach, Taylor & Francis, 823 – 839

Calloway, L.J., & Knapp, C.A. (1995). Using grounded theory to interpret interviews. Technical report, School of Computer Science and Information Systems, Pace University, New York, NY, USA, http://csis.pace.edu/~knapp/AIS95.htm

Carver, J. (2007). The use of grounded theory in empirical software engineering. In: *Empirical Software Engineering Issues. Critical Assessment and Future Directions.* LNCS 4336, Berlin/Heidelberg, Germany, Springer, 42–42

Chikofsky, E.J., & Cross, J.H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7, 13–17

Cockburn, A. (2000). Selecting a project's methodology. *IEEE Software* 17, 64–71

Coleman, G., & O'Connor, R. (2006). Software process in practice: A grounded theory of the irish software industry. In: *Software Process Improvement.* LNCS 4257, Berlin/Heidelberg, Germany, Springer, 28–39

Coleman, G., & O'Connor, R. (2007). Using grounded theory to understand software process improvement: A study of Irish software product companies. *Information and Software Technology* 49, 654–667

Dick, B. (2005). Grounded theory: a thumbnail sketch. Technical report, Graduate College of Management, Southern Cross University, Lismore, NSW, Australia, http://www.scu.edu.au/schools/gcm/ar/arp/grounded.htmlGalal, G.H. (2001). From contexts to constructs: the use of grounded theory in operationalising contingent process models. European Journal of Information Systems 10, 2–14

Galal, G.H., & Paul, R.J. (1999). A qualitative scenario approach to managing evolving requirements. *Requirements Engineering Journal* 4, 92–102

Gause, D., & Weinberg, G. (1990). *Are Your Lights On? How to Figure Out What the Problem REALLY Is.* Dorset House, New York, NY, USA

Glaser, B.G. (1992). *Basics of Grounded Theory Analysis: Emergence vs. Forcing.* Sociology Press, Mill Valley, CA, USA

Glaser, B.G., & Strauss, A.L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine, Chicago, IL, USA

Gold, N.E., & Bennett, K.H. (2002). Hypothesis-based concept assignment in software maintenance. *IEE Proceedings — Software* 149, 103–110

Hoda, R., Noble, J., & Marshall, S. (2010). Using grounded theory to study the human aspects of software engineering. In: *Human Aspects of Software Engineering,* Reno, NV, USA, 5:1–2

Holt, R. (2002). Software architecture as a shared mental model. In: *Proceedings of the Tenth International Workshop on Program Comprehension (IWPC).* Paris, France

IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications.* ANSI/IEEE Standard 830-1998. IEEE Computer Society, Los Alamitos, CA, USA

Jeong, H.S. (2006). A Grounded Analysis of the Sensemaking Process of Korean Street-Level Fire Service Officials. PhD thesis, Public Administration, Florida State University, Tallahassee, FL, USA

Johansson, M., & Timpka, T. (1996). Quality functions for requirements engineering in system development methods. *Informatics for Health and Social Care* 21, 133–145

Kapser, C.J., & Godfrey, M.W. (2006). “Cloning considered harmful” considered harmful. In: *Thirteenth Working Conference on Reverse Engineering (WCRE)*, 19–28

Lang, M., & Fitzgerald, B. (2007). Web-based systems design: a study of contemporary practices and an explanatory framework based on “method-in-action”. *Requirements Engineering Journal* 12, 203–220

Pershin, G. (2006). Adoption of Policies that Permit Community Colleges to Grant Bachelor Degrees in Florida. PhD thesis, Public Administration, Florida State University, Tallahassee, FL, USA

Power, N. (2002). A Grounded Theory of Requirements Documentation in the Practice of Software Development. PhD thesis, Dublin City University, Dublin, Ireland

Ramos, I.M. (2000). Aplicações das Tecnologias de Informação que Suportam as Dimensões Estrutural, Social, Política e Simbólica do Trabalho. PhD thesis, Departamento de Informática, Universidade do Minho, Guimarães, Portugal

Rittel, H., & Webber, M. (1973). Dilemmas in a General Theory of Planning. Policy Sciences 4, 155–169

Robertson, S., Robertson, J. (2006). *Mastering the Requirements Process.* Second Edition. Addison-Wesley, Harlow, UK

Sillito, J., Volder, K.D., Fisher, B., & Murphy, G. (2005). Managing software change tasks: An exploratory study. In: *Proceedings of the International Symposium on Empirical Software Engineering (ISESE),* 23–32

Sillito, J., & Wynn, E. (2007). The social context of software maintenance. In: *Proceedings of the Twenty-Third IEEE International Conference on Software Maintenance (ICSM),* 325–334

Sommerville, I. (2007). *Software Engineering.* Pearson Education, Harlow, UK

Teixeira, L., Ferreira, C., & Santos, B.S. (2010). User-centered requirements engineering in health information systems: A study in the hemophilia field. *Computer Methods and Programs in Biomedicine* 106, 160–174

Walsham, G. (1995). Interpretive case studies in IS research: Nature and method. *European Journal of Information Systems* 4, 74–83

Weber, J.L. (2010). Privacy and security attitudes, beliefs and behaviours: Informing future tool design. Master’s thesis, University of Waterloo, Waterloo, ON, Canada, http://se.uwaterloo.ca/~dberry/FTP_SITE/students.theses/janna.weber/thesis.pdf