

Traceability in Agile Projects

Jane Cleland-Huang

1 Introduction

Agile methodologies represent a set of development processes in which both the requirements and the delivered solution evolve incrementally through a series of short iterations. Such projects are characterized by an emphasis on human interactions and collaborations, lightweight development processes, frequent deliverables, and minimal documentation (Ambler, 2004; Beck and Andres, 2004; Cockburn, 2000; Schwaber, 2004; Warden and Shore, 2007). Not surprisingly, traceability is generally perceived by agile developers as a heavy-weight and burdensome activity which returns little value to the project (Appleton, B. ACME Blog, 2005; Cleland-Huang 2006). On the other hand, agile practices are increasingly adopted in larger, distributed, and sometimes safety-critical projects, and it is often in these environments that the benefits of traceability outweigh its costs. For example, if agile methods are used to build a healthcare device, then the approval process for that device will require a demonstration that the device is safe. The Federal Drug Administration (FDA), which is responsible for approving such devices in the USA, specifically requires traceability between requirements, design, code, and test cases (USA Food and Drug Administration, 2005). For such projects, the burden is therefore placed on developers to create a project environment that establishes the necessary traceability relationships (Gotel and Finkelstein, 1994; Pinheiro FAC, 2003; Ramesh and Jarke, 2001).

As explained in the chapter “[Traceability Fundamentals](#)” of this book, the traceability efforts for a project must be fit for purpose. This means that there is no “one-size-fits-all” option where traceability is concerned, and traceability decisions will vary widely across different kinds of agile projects. In this chapter we therefore explore the issues, challenges, and goals for tracing in an agile environment, and propose specific solutions that we believe balance the spirit of agility with the increasingly common challenges of scale, complexity, and compliance.

J. Cleland-Huang (✉)
DePaul University, School of Computing, 60604 Chicago, USA
e-mail: jhuang@cs.depaul.edu

2 A Quick Look at Agility

Although a thorough discussion of agile methods is outside the scope of this book, we briefly present the philosophies and practices of agile development in order to provide the context for the remainder of this chapter. In general, agile practices share a set of fundamental philosophies, documented in the Agile Manifesto.¹ These philosophies stress the importance of people and their interactions over a strict adherence to processes and tools and the delivery of executable code over masses of documentation. They also emphasize the need to work closely with customers to shape the product and to resolve issues as they emerge instead of relying upon rigidly defined contracts. Finally they stress the importance of embracing changing requirements instead of rigidly following a pre-set development plan (Beck and Andres, 2004; Warden and Shore, 2007).

One of the most popular and well adopted agile approaches is eXtreme Programming (Beck and Andres, 2004), introduced by Kent Beck in 2004. XP includes several different practices which Beck found to be a particularly effective mix in the projects he had worked on. These practices are summarized in Table 1 and discussed below.

Requirements in XP projects are first captured as user stories and then later transformed by the customer into user acceptance tests. We illustrate this using an example taken from the healthcare domain. CCHIT (Certification Commission for Health Information Technology) certification standards requires all healthcare related systems, such as iTrust, to use ISO-8601 compatible timestamps. In an XP project, this might be documented as the following user story:

User story
Export all time stamps using ISO-8601 format.

which could be transformed into an acceptance test as follows:

Acceptance Test
Given a system timestamp of “November 5, 1994, 8:15:30 am, US Eastern Standard Time”, when the system exports the timestamp as part of a patient’s visit log, then the timestamp shall appear as “1994-11-05T13:15:30-05:00”

In most XP projects, it is the test cases, rather than the user stories, which are considered to categorically document the user requirements. When the test cases are executed and the tests are passed, the system is considered to meet its requirements. A pure XP project will therefore have little in terms of documentation beyond some

¹ Agile Manifesto: <http://agilemanifesto.org>

Table 1 Practices of extreme programming (Beck and Andres, 2004)

XP practice	Brief description
1. Customer team member	Include a customer in the core team who can define & prioritize features, write test scripts, and work closely with developers
2. User stories	Capture initial requirements as user stories i.e. brief and informal descriptions of features, each of which typically takes 1-3 weeks of work
3. Short cycles	Build and deliver the product in a series of short iterations
4. Acceptance tests	Transform user stories into scripted acceptance tests
5. Pair programming	Developers work in pairs. One thinks and assesses while the other writes code
6. Test-driven development	Write test cases first before writing the code. Automate all test cases
7. Collective ownership	All developers are responsible for all of the code
8. Continuous integration	Integrate new code frequently, and rerun all automated tests following each integration. When tests fail, roll-back, fix code, resubmit, and rerun tests
9. Sustainable pace	Keep a maintainable pace throughout the project
10. Open workspace	Work in an environment which is conducive to collaboration and which emphasizes the use of informal shared work spaces for tracking progress, agile modeling etc
11. The planning game	Developers and customers work together to plan iterations and releases
12. Simple design	Choose the design option which is the simplest solution for meeting the current needs of the project. Don't engage in big upfront designs or design solutions which look too far into the future of "what if" scenarios
13. Refactoring	Keep the design clean. A clean design is more poised to accommodate change
14. Metaphor	Provide meaningful and well-understood names to elements of the project

possibly throw-away user stories, user acceptance tests, and code (including unit test cases).

Although, not part of eXtreme Programming per se, Agile Modeling (AM), as proposed by Scott Ambler (Ambler, 2004) uses basic models from the unified modeling language (UML) and other approaches to shape the overall design of the system. However, in AM, the architectural and design models are more likely to be sketched on white boards or flipcharts than in more formal modeling tools. Furthermore, many agile models are considered throw-away, i.e. they are used to help sketch important elements of the design, but are not retained as permanent documentation. As a result of these practices, agile projects tend to travel quite "light" and have few traceable artifacts.

As depicted in Table 1, eXtreme Programming also follows the three practices of simple design, refactoring, and continuous integration (Beck and Andres, 2004). Together, these three practices serve the primary purpose of keeping the system flexible and maintainable, so that it is poised to embrace change. The constantly

evolving nature of an agile project introduces additional challenges, and as a result, traditional approaches in which significant effort is needed to manually create and maintain traceability links are very unlikely to be adopted. Any viable traceability solution must therefore be light-weight, highly adaptable, and sufficiently robust to be effective in the continually changing environment of an agile project.

This discussion highlights an interesting facet of the traceability problem. While software engineers place significant emphasis on building maintainable systems in which classes or other kinds of modules are loosely coupled and highly cohesive, they have generally failed to extend these same practices to traceability. As a result, most traceability processes create a brittle and rigid layer over an otherwise relatively flexible software system. Given the environment and goals of an agile project, a brittle traceability solution is simply not a viable option. In the following sections we therefore explore some viable options for tracing in agile projects.

3 The Benefits of Tracing in Agile Projects

In many respects the traceability needs in agile projects are not that different from other project environments (Gotel and Finkelstein, 1994; Pinheiro FAC, 2003). Brad Appleton identified several reasons for tracing in an agile project (Appleton, B. ACME Blog, 2005), of which the most important ones are listed below:

- **Change impact analysis** – to assess how a proposed change will impact the existing system, in order to accommodate tasks such as communication, team coordination, and effort estimation.
- **Product conformance** – to ensure that the delivered product meets the customers' needs i.e. realizes their requirements. This is commonly referred to as requirements validation.
- **Process compliance** – to ensure that any procedural processes such as reviews and tests have been conducted.
- **Project accountability** – to provide assurance that the solution does not include gold-plating (i.e. excess functionality), and that all changes match a requested feature request.
- **Baseline reproducibility** – to support configuration of baselines, so that different versions can be reproduced.
- **Organizational learning** – to document rationales behind critical decisions in order to transfer knowledge to new team members.

Agile traceability goals are similar to those found in non-agile projects (Cleland-Huang et al., 2007; Gotel and Finkelstein, 1994), but differ in the way in which they are achieved. Appleton points out that if we look beyond the limitations of traditional traceability techniques, many tracing goals can be satisfied through trust and communication in a project. For example, if team members intrinsically hold the knowledge for how a modification will impact a system, then there is no need to capture that kind of information in the form of traceability links, or if developers

trust each other not to gold-plate the system, then there is no need to create traceability links to demonstrate that all sections of the code trace back to requirements. However, while many agile projects, successfully follow such approaches, the practices become increasingly inadequate in larger and more complex projects.

4 Tracing in Agile Projects

4.1 Basic Traceability

The most common tracing scenario for an agile project is depicted in the Traceability Information Model (TIM) shown in Fig. 1. It is interesting to note that this TIM emerged as a result of forum discussions with agile developers² and therefore reflects their perception of common practice. This TIM shows how traceability is first established between acceptance tests and user stories through inserting a cross-reference to one or more user story into each of the acceptance tests. This is very simple, and is currently supported by a number of agile management tools such as Rally SoftwareTM. When test cases are executed and passed, we are assured that the code implements the test case, and we can therefore implicitly establish an “implements” trace from the test case to the code. This means however, that the code is treated as a single high-level target artifact, and there is no visibility as to which classes are related to which test cases.

This raises the interesting question of whether traceability at this level of granularity supports the primary traceability goals of impact analysis and product

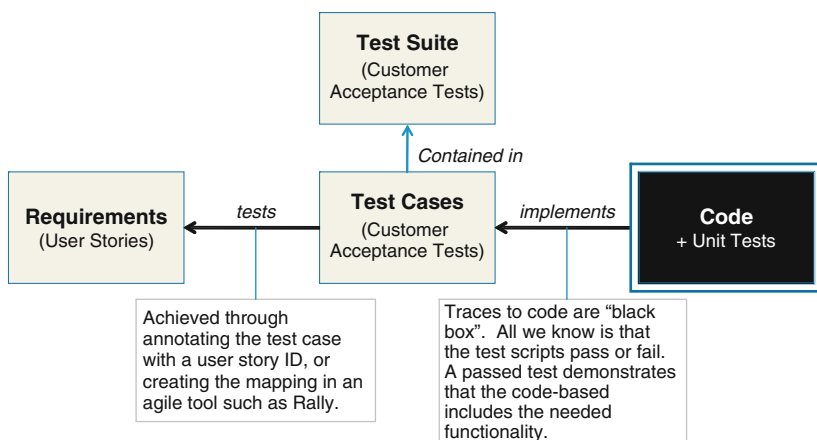


Fig. 1 A traceability information model for a basic agile project

² Agile Project Management Forum hosted on Yahoo – Discussion thread “Agile Traceability”

conformance. Clearly, it can demonstrate product conformance because for any given user story we can identify the related acceptance tests and determine if they are satisfied by the code (Richardson and Green, 2004). What it does not do, is to support traditional impact analysis in which traceability links are used to identify potentially impacted sections of the code in order to plan a proposed change, manage risks, or estimate effort. However, this level of support is not necessarily needed in many agile projects, especially if the projects are small or medium sized. In practice developers draw from their own knowledge of the system to plan and execute a change, and then run automated test cases to make sure that the change has been correctly implemented without unwanted side effects. A benefit of this approach is that it provides a highly resilient traceability mechanism anchored around acceptance and user tests, and which does not become brittle over time.

This approach is feasible unless the size of the project is too large, or the longevity of the project and staff turnover mean that communal knowledge is insufficient for the tracing task.

4.2 *Beyond the Basics*

In his article on “Lean Traceability” (Appleton et al., 2007), Brad Appleton describes several additional techniques for lightweight tracing. One such technique utilizes existing configuration management tools in order to capture traceability links. If changes are implemented at the level of granularity that would be expected in a test-driven environment, then software-level requirements, design, code, and test cases would all be part of the same task, and could easily be tracked to a single transaction ID in the change management system. Implementing simple configuration management, which is a natural part of an agile project, would therefore automatically produce traces into the code at the granularity of the check-in units (Guckenheimer, 2006). This is an improvement on the black-box approach depicted in Fig. 1.

Jacobsson conducted a series of interviews with developers, testers, configuration managers, product owners, scrum masters, and scrum coaches from several agile development projects (Jacobsson, 2009). He asked them “do you feel the need to trace in a project, and if so what kind of tracing?” According to Jacobsson, several of the interviewees were initially skeptical about the need for traceability, but then did identify several potentially useful traceability tasks, above and beyond the ones depicted in Fig. 1. We discuss the most important ones below:

- Stakeholder to requirements – to track who contributed the requirement (or user story). This makes it possible to identify the source of each requirement, and also to create a feedback loop to keep stakeholders notified of progress. The traceability link can be implemented simply by tagging each requirement with the contributing stakeholders’ names.
- Requirements (user stories) to versions – to track exactly which requirements have been implemented in a specific revision. This can be accomplished quite

trivially using a source control tool such as Perforce, SVN, or Clear Case by attaching user story IDs to all committed changes. This provides traceability between requirements and code as a natural byproduct of the configuration management process. Trace granularity is at the checkin level. In fact, this process is described in greater detail in the chapter “[Evolution-Driven Trace Acquisition in Eclipse-Based Product Line Workspaces](#)” of this book.

- Requirements to requirements – to track dependencies between user stories. Although this could potentially create significant overhead, it is relatively easy to insert identified dependencies into user stories. Furthermore, it is possible to document dependencies as they are observed, or as new stories are introduced. This can be useful during the planning process, and there is no need to retain such traces once stories have been implemented.

4.3 Trace Retrieval

One of the themes pervading the previous sections of this chapter is that in many cases traces can be generated as a byproduct of the normal agile process at only negligible cost. In this section we explore one additional low-cost option, which can be useful in larger projects when developers may not have sufficient intrinsic knowledge of the overall project, and may therefore need automated support to understand the impact of a new user story, or to determine the purpose behind a feature in the code.

Just-in-time Traceability (JITT) addresses this problem through using information retrieval techniques to automatically create candidate traceability links (Cleland-Huang et al., 2007; De Lucia et al., 2007; Hayes et al., 2003; Lin et al., 2006). Although JITT does not produce perfect results, it has been shown to significantly reduce the cost and effort of the tracing process. In this chapter we do not present the theory or algorithms behind JITT, as they have been described in significant detail in the chapter “[Automated Techniques for Capturing Custom Traceability Links Across Heterogeneous Artifacts](#)”; however we do discuss its application to agile projects.

Figure 2 shows an example of JITT in action. In this example, the developer wants to find out how a new user story “Display drug-related interactions for each patient at the time a new order is being placed” will impact the code. It may be that the developer is fairly new to the project, or that the system was deployed a while ago and not modified recently, or it could be that the developer has a general idea of which classes might be impacted but wants to augment her existing knowledge with a dynamic trace. Either way, the tool returns a list of candidate classes and shows the likelihood that the retrieved class is related to the user story. Classes exhibiting higher similarity to the user story are returned towards the top of the list as they are more likely to be correct links than those returned further down the list. The developer can explore the details of individual classes, or may interactively improve the trace query through filtering terms or adding additional search words. A JITT tool therefore provides an interactive environment for retrieving potentially relevant

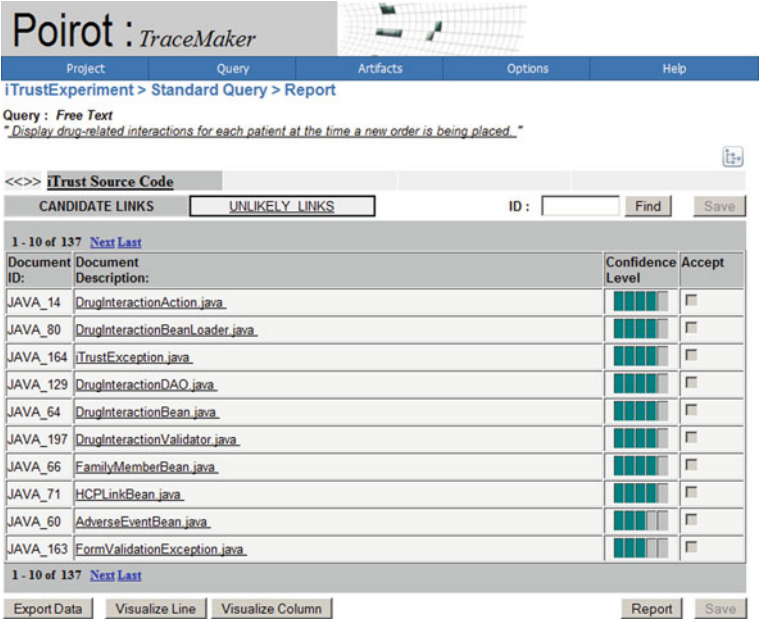


Fig. 2 Just-in-time Traceability in which traces are generated automatically

code. No permanent traces need to be stored or maintained, and the primary cost and effort of the trace is incurred at the time of need, thereby avoiding the problem found in more traditional approaches in which developers must invest significant time creating traceability links which they may or may not ever actually need.

On the other hand utilizing JITT on a project does require initial setup of the JITT tool, and this involves interfacing with the version control system so that queries can be launched against the code base. This is generally quite simple as there are several tools, such as src2ML (Collard et al., 2003) which can quickly convert code into a format parsable by JITT tools.

5 Traceability Across Different Types of Agile Projects

Agile projects come in many different shapes and sizes, and therefore their traceability needs differ quite significantly. This fact was stressed by Espinoza and Garbajosa, who proposed a model-based approach to planning traceability in agile projects (Espinoza and Grabajosa, 2011). This extends Ramesh’s previous work on metamodels (Ramesh and Jarke, 2001), and the general practice of using Traceability Information Models (TIMs) as outlined in the chapter “Traceability Fundamentals” of this book. Espinoza advocates making traceability fit for purpose through allowing user-defined traceability links, well defined roles, and linkage rules which specify what kinds of traceability links can and must be created.

In this chapter of the book, we do not attempt to propose specific TIMs for different types of projects, but rather provide some general guidelines for tracing in three different types of agile projects. These can be characterized according to size, longevity, complexity, and criticality.

5.1 Typical Small to Medium Sized Agile Projects

A typical project is characterized primarily by its small to medium size and by the volatility of its requirements. While it may be hard to justify the creation and maintenance of traditional traceability links in such projects, the basic traces from Requirements (user stories) to test cases, and from test cases to (black-box) code, provide sufficient support in many projects. As the project size increases, there is increasing value in extracting traceability information from the check-in transactions of the version control systems to establish more finely grained traceability links from requirements and test cases to code.

5.2 Large Scaled, Distributed, or Long-Lived Projects

The second kind of agile project is characterized by a multi-year development duration, large and potentially distributed teams, and size and complexity of the code-base. In such projects it becomes difficult, if not impossible, to rely on communal memory to understand where specific functions have been implemented in the code, or to recall parts of the code that are sensitive to change because of interrelated trade-offs. This is particularly important to consider, because studies of agile projects have shown that although the cost and effort of change in agile projects, increases far more slowly than in traditional projects, it still does increase over the life of the project.

Furthermore, agile projects often suffer from some of the same ailments as traditional projects in terms of quality degradation. While we are not advocating the need for a heavy-weight traceability solution we suggest a customized mix of three different techniques: (i) basic traces from user stories → test cases → black-box code, (ii) JITT, and (iii) more fine-grained user-stories → code traces achieved through tagging checkin units with associated user stories.

5.3 Safety Critical Project

Finally, we consider the traceability needs of agile projects that incorporate safety-critical components. Such projects are obligated to demonstrate that the devices, including the software that runs on them, are safe for use (Appleton et al., 2007; USA Food and Drug Administration, 2005). Under these circumstances, traceability becomes compulsory, and certification requirements typically require bi-directional

traces to be established from requirements to design, design to code, and code to test cases.

When the safety case is not adequately made, the company can be fined, or worse still, any devices running the uncertified software can be forcibly recalled. While a complete discussion of tracing in safety critical systems is outside the scope of this chapter, we do stress that such projects must follow a more rigorous traceability process in which risks are analyzed, hazards are identified, and mitigating requirements are specified (Cleland-Huang et al., 2012). It is then expected that a safety case will be developed which shows full life cycle traceability between hazards, requirements, design, code, and test cases. Such traces are likely to be stored in a more traditional traceability matrix which carries with it all the responsibilities and costs associated with manually creating and maintaining traceability links. These costs are justified due to the certification requirements of such a project.

6 Conclusions

This chapter has presented some basic ideas for effective tracing in agile projects. The guiding principles have been to eradicate the significant overheads of traditional traceability techniques and replace them with techniques that are significantly more cost effective. While many agile developers have a tendency to scoff at the idea of traceability, the increasing size and complexity of agile projects compels us to find ways to achieve the benefits of traceability without the cost and effort of traditional approaches. We are therefore forced to question the purpose of traceability, and look far beyond the confines and problems of traditional traceability techniques to discover new approaches for achieving the same goals.

References

- Ambler, S.: *The Object Primer: Agile Model-Driven Development with UML 2.0* (2004, 22 March)
- Appleton, B. ACME Blog: Traceability and TRUST-ability. <http://bradapp.blogspot.com/2005/03/traceability-and-trust-ability.html> (2005, Tuesday, 15 March). Accessed June 2011
- Appleton, B., Cowham, R., Berczuk, S.: *Lean traceability: A smattering of strategies and solutions*, CM Crossroads (Configuration Management) (2007, Tuesday, 18 September, 16:57)
- Beck, K., Andres, C.: *Extreme programming explained: embrace change*, 2nd edn. Addison-Wesley, Boston, MA (2004). ISBN:0321278658
- Cleland-Huang, J.: Just enough requirements traceability. *COMPSAC* **1**, 41–42 (2006)
- Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., Romanova, E.: Best practices for automated traceability. *IEEE Comp.* **40**(6), 27–35 (2007). ISSN:0018-9162
- Cleland-Huang, J., Heimdahl, M., Hayes, J.H., Lutz, R., Maeder, P.: Trace queries for safety requirements in high assurance systems. In: *Working Conference on Requirements Engineering for Quality*. Essen, Germany, March 2012
- Collard, M., Kagdi, H., Maletic, J.: An XML-based lightweight C++ Fact extractor. *IWPC* 134–143 (2003)
- De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.* **16**(4), 13 (2007). ISSN:1049-331

- Cockburn, A.: Selecting a project's methodology. *IEEE Softw.* **17**(4), 64–71 (2000). ISSN:0740-7459. doi:10.1109/52.854070
- Espinoza, A., Grabajosa, J.: A study to support agile methods more effectively through traceability. *Innov. Syst. Softw. Eng.* **7**, 53–69 (2011)
- Gotel, O., Finkelstein, A.: An analysis of the requirements traceability problem. In: *Proceedings of the International Conference on Requirements Engineering (RE)*, pp. 94–102. IEEE Computer Society, Springs, Colorado (1994)
- Guckenheimer, S.: *Software Engineering with Microsoft Visual Studio Team System*. Addison Wesley, Boston, MA (2006, May)
- Hayes, J.H., Dekhtyar, A., Osborne, J.: Improving requirements tracing via information retrieval. In: *Proceedings of the IEEE International Conference on Requirements Engineering (RE)*, p. 138. IEEE Computer Society, Washington, DC (2003). ISBN:0-7695-1980-6
- Jacobsson, M.: *Implementing traceability in agile software development*. Master's Thesis, Lund Institute of Technology (2009, January)
- Lin, J., Lin, C.C., Cleland-Huang, J., Settini, R., Amaya, J., Bedford, G., Berenbach, B., Ben Khadra, O., Duan, C., Zou, X.: Poirot: A distributed tool supporting enterprise-wide automated traceability. In: *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. IEEE Computer Society, Washington, DC (2006). ISBN:0-7695-2555-5
- Pinheiro, F.A.C.: Requirements traceability. In: Sampaio do Prado Leite, J.C., Doorn J.H. (eds.), *Perspectives on Software Requirements*, vol. 753, pp. 93–113. Springer, Berlin (2003)
- Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.* **27**(1), 58–93 (2001)
- Richardson, J., Green, J.: Automating traceability for generated software artifacts. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*, pp. 24–33. IEEE Computer Society, Washington, DC (2004). ISBN:0-7695-2131-2
- Schwaber, K.: *Agile Project Management with Scrum*. Microsoft Press, Redmond, WA (2004). ISBN:073561993X
- Warden, S., Shore, J.: *The Art of Agile Development: With Extreme Programming*. O'Reilly Media, Inc., Sebastopol, CA (2007). ISBN:0596527675
- USA Food and Drug Administration: *Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices*, May 11, 2005