Continuously Delivered? Periodically Updated? Never Changed? Studying an Open Source Project's Releases of Code, Requirements, and Trace Matrix

Wentao Wang*, Arushi Gupta*, and Yingbo Wu†
* Department of Electrical Engineering and Computing Systems, University of Cincinnati, USA
† School of Software Engineering, Chongqing University, China
{wang2wt, gupta2ai}@mail.uc.edu, wyb@cqu.edu.cn

***Abstract*—Many open source software projects deliver code continuously. How are the project's requirements updated? What about the traceability information of those requirements? To answer these questions, this paper reports our initial analyses of the iTrust medical care project's all publicly accessible releases. The results show that, as iTrust releases two versions per year, the code growth is smooth but the requirements growth experiences periodic mass updates. The asynchronous evolving paces cause the RTM stagnant, outdated, and inaccurate. Our work provides concrete insights into what updates should be applied to the requirements and the RTM in the face of the code changes, and illustrates the need for new ways to automatically keep requirements in sync over continuous release cycles.**

***Index Terms*—Traceability, trace matrix, just-in-time requirements, continuous delivery, software evolution, iTrust.**

I. Introduction

Requirements traceability matrix (RTM) establishes the mapping between a project's requirements and other types of artifacts such as source code and test cases. RTM can support many software engineering tasks [1]: verification and validation, risk assessment, system-level test coverage analysis, just to name a few.

One area that RTM can be specially helpful is *continuous delivery* where the development team keeps producing valuable software in short cycles and ensures that the software can be reliably released at any time [2]. In each release cycle, RTM can assist developers in managing the backlog of implementation issues, reasoning about the dependencies of upcoming new features with existing ones, and assessing the impact of requirements change on code and architecture.

Creating the RTM for a single release is no simple matter. Despite automated support (e.g., candidate traceability link generation via information retrieval [1]), much manual effort is still required to make sure that the traceability information is accurate and complete [3]. Having an explicit RTM as part of a release, therefore, is rare, and having RTMs in continuous releases is much rarer.

One project falling into this rare category is iTrust (http://agile.csc.ncsu.edu/iTrust), an open-source Java application aimed at providing patients with a means to keep up with their medical records, as well as to communicate with their doctors. The project is developed by students from the North Carolina State University, and follows an agile way of releasing a new version of the software (including source code, requirements-level use cases, RTM documenting the implementation relationship between use cases and code, etc.) in every academic semester (i.e., twice a year in Fall and in Spring) [4]. Because the RTM released by the project team is so valuable, it has been used as the "answer set" (also known as "gold standard" or "ground truth") extensively by researchers to evaluate the effectiveness of automated methods (e.g., [5, 6]) and to understand human analyst's behavior in requirements tracing (e.g., [7, 8]).

How accurate is the RTM in the face of a software project's continuous delivery? We set out to answer the research question by analyzing all publicly available iTrust's releases. This paper reports our initial study showing that iTrust's source code and requirements follow different increase patterns, and that some requirements changes appear later than the corresponding code changes. Such an asynchronous evolving pace leads to a stagnant, outdated, and inaccurate RTM as part of the project release. The results not only challenge the "ground truth" of manually maintained RTMs, but also suggest new ways to automatically keep requirements in sync over continuous release cycles.

II. iTrust and its Releases

The iTrust electronic health care system is an active team project for undergraduate students in North Carolina State University's Software Engineering course. Dr. Laurie Williams created iTrust in the Fall of 2005 as a patient-centric application for maintaining electronic health records. The main rationale is to have a software-intensive system that combines medical information from multiple sources to provide a summary or detailed view of a particular patient's history in a way that is useful to the health care practitioners [4].

The project employs Java Server Pages (JSPs) to handle user interfaces and HTTP requests. The business logics and data accesses are coded in Java production classes. The project portal uses a wiki style to organize the release information including source code, testing documents (acceptance test plan and test data), requirements (glossary, functional requirements defined by use cases, non-functional requirements, constraints, and data field formats), and RTM stored in a spreadsheet.

...	...
UC32S1	No links
UC32S2	/auth/hcp/viewPrescriptionRenewalNeeds.jsp
UC32S2	ViewPrescriptionRenewalNeedsAction.get RenewalNeedsPatients()
UC32S2	PersonnelDAO.getPersonnel()
UC32S2	PatientDAO.getRenewalNeedsPatients()
UC32E1	/auth/hcp/viewPrescriptionRenewalNeeds.jsp
...	...

Fig. 1. Excerpt of iTrust's RTM.

TABLE I
PUBLICLY AVAILABLE ITRUST'S RELEASES TILL JULY 13, 2015

Version	Date (mm/dd/yy)	# of Java Methods		# of Req.s Units	
		total	new	total	new
v4	12/12/07	1106	–	92	–
v6	08/23/08	1496	522	128	37
v7	01/15/09	1548	180	140	19
v8	08/17/09	1500	86	153	14
v9	01/11/10	1636	153	181	20
v10	08/17/10	1737	103	198	23
v11	01/17/11	1856	123	287	140
v12	08/14/11	2135	344	194	48
v13	01/17/12	2342	202	199	6
v14	08/16/12	2336	133	203	7
v15	01/06/13	2378	73	208	5
v16	08/19/13	2421	72	205	11
v17	01/09/14	2665	256	211	10
v18	08/20/14	2849	181	227	16
v19	01/08/15	2946	97	242	15

In iTrust, the RTM specifies how the sub-flows and alternative flows of a use case are implemented by JSPs and Java methods. An excerpt is shown in Fig. 1 where 3 requirements-level elements are traced: 2 sub-flows (UC32S1 and UC32S2) and 1 alternative flow (UC32E1). We therefore use "requirements unit" to refer to a sub-flow or an alternative flow, and select Java method as the implementation-level unit of analysis in our study. "No links" in Fig. 1 indicates that no traceability information for UC32S1 is recorded in this RTM.

The release of iTrust became open source since version 4 in December 2007. Starting from version 6, the source code was released twice a year (Fall and Spring) over SourceForge (http://sourceforge.net/projects/itrust/). The two leftmost columns of Table I display the basic information of all publicly accessible iTrust versions. The latest version up to this writing is v19 released in January 2015. It is clear from iTrust's portal that source code changes drive project release, though requirements are maintained separately in the project wiki. It is important to note that the spreadsheet defining the RTM can only be downloaded as part of the code release.

The code-driven releases of iTrust present challenges for keeping other artifacts on track. Requirements, for example, may not be kept up-to-date. In fact, Charrada *et al.* [9] studied iTrust in their work and identified 14 outdated use cases when the software evolved from v10 to v11. While their work helped automatically identify which requirements might be updated between two consecutive versions, our analyses in the next section investigate the actual changes that should be applied to the requirements and trace those changes throughout the entire project history.

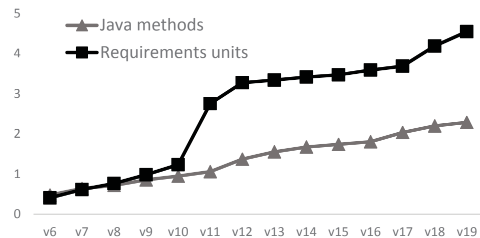

Fig. 2. Cumulative increase rate (y-axis) over continuous releases (x-axis).

III. CHANGES OF CODE, REQUIREMENTS, AND RTM

Our analyses are based on iTrust's release versions. For each version, we used an automated tool [7] to extract all the Java methods in the code base. We consider a method of the latter version to be *new* if it does not exist in the previous version. The diff is performed on the basis of a method's name and its full path. Because iTrust's requirements are maintained in the project wiki, we take a requirements snapshot on the release day or on the prior day closest to the release to be the one associated with a particular code release. Each requirements unit (i.e., sub-flow or alternative flow) is then identified manually following the unique ID (e.g., UC32S1, UC32E1, etc). Similar to Java method, a requirements unit is *new* if it does not appear in the previous version. Table I provides the quantities of total and new methods, as well as the requirements counterparts.

To analyze the change trends, we plot in Fig. 2 the cumulative increase rate of code and requirements. By cumulative, we mean the number of new methods (or requirements units) accumulated up to the current version. The cumulative increase rate then equals to the method-cumulative (or requirements-unit-cumulative) divided by the baseline number (i.e., v4 in our case). Fig. 2 shows that the growth of Java methods is smooth, complying particularly well with Lehman's law of "continuing growth" [10]. The requirements growth is smooth in the beginning, experiences an abrupt jump from v10 to v11, and flattens down after that. The main reason of the jump is due to the "mass update of all requirements to reflect actual implementation of iTrust" logged by the actual developers between v10 and v11 on 12/31/2010. As a result, 140 new requirements units were added (shown in Fig. 2), and meanwhile 52 outdated units were removed (not shown in Fig. 2). This indicates the periodic, catching-up nature of requirements update in a project whose releases are driven by code changes.

Compared to Java method's continuously smooth growth and requirements unit's periodically jumpy increase, iTrust's RTM never changed. The RTM spreadsheet first appeared in v10's release and stayed the same since then in every release till v19. Interestingly, the manually constructed RTM was released together with v10's code. This was before the mass update mentioned above, which resulted in better synchronization between code and requirements. We speculate that, by creating the RTM, the project team realized many requirements

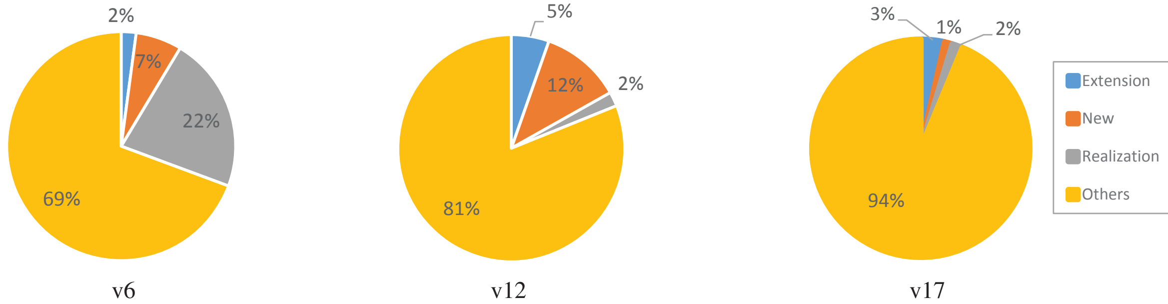

Fig. 3. Classifying new Java methods as they relate to requirements.

were outdated and hence the need for mass update. Testing the speculation will require future work. In any event, the stagnant status of the RTMs motivates our analyses of what changes that the requirements and the RTM *should* have in order to keep up with the continuous code change.

We observe that each newly added Java method can be linked to the requirements in three different ways.

- **Extending an existing requirement.** A pair of additions in v6 corresponded to each other. On the requirements side, UC11S2 (document office visit) was expanded to have an optional ordering by the lab procedures. Accordingly, the new method OfficeVisitDAO.addLabProcedureToOfficeVisit() was implemented to fulfill the requirements extension.
- **Implementing a new requirement.** In v17, AuthDAO.setDependent() was created to implement UC58S2 (manage dependency). UC58S2 appeared as a new requirements unit in v18 — an instance of delayed requirements update compared to the corresponding code change.
- **Realizing a previously unfulfilled requirement.** Although UC3S4 (authenticate users) appeared in v4, it had not been fulfilled. Had the RTM existed in v4, UC3S4 would have "no links". The method LabProcedureDAO.getLabProceduresForLHCPForNextMonth() was added to the code base in v6 to implement UC3S4. In this way, the new code helped clean up the backlog of existing but unfulfilled requirements.

We choose to analyze 3 iTrust versions (v6, v12, and v17) as they exhibit the greatest number of new Java methods (522, 344, and 256) shown in Table I. For each new method, we manually classify it based on the above categories. If none of the 3 categories applies, we place the method into "others". The analysis results are presented in Fig. 3. In an early version (v6), 22% of the methods were created to fulfill previously unfulfilled requirements. Such cleaning ups, whose proportions decreased in proportion in v12 and v17, had direct impact on RTM changes, namely, the "no links" should be replaced by the methods to accurately capture the updated requirements traceability information. Requirements extensions remained relatively stable in Fig. 3. This may suggest the addition of new requirements-code tuples is expected to be normal during evolution, confirming Lehman's law of "self regulation" [10].

TABLE II
DETAILING THE CATEGORIES OF NEW METHODS

Method / Req.s	v6				v12				v17			
	Ext.	New	Realiz.	Total	Ext.	New	Realiz.	Total	Ext.	New	Realiz.	Total
v4	11		103	114	17.5			17.5	6		4	10
v6		30.5		30.5	1			1	2			2
v7		2		2								
v8		1		1								
v9		0.5		0.5			7	7	2			2
v10												
v11												
v12						12.5		12.5				
v13												
v14		2		2								
v15												
v16												
v17						27		27				
v18										3		3
v19												
Total	11	36	103	150	18.5	39.5	7	64	10	3	4	17

The category of new requirements shows fluctuated distributions in Fig. 3. To better understand the extent to which requirements addition synchronizes with code addition, we present a detailed view in Table II. Take the set of Java methods created in v6 as an example, 36 appeared to implement the requirements that did not exist in v4. While a majority of the new requirements were synchronized in v6, some were delayed (e.g., 2 showed up in v14). The reason of non-integers in Table II is due to the many-to-many relation between code and requirements [11]. The newly created method PatientDAO.getRepresenting(), for instance, was responsible for 2 requirements' implementation: UC23S3 (view comprehensive patient report) appeared in v6 and UC34S4 (report telemedicine monitoring details) appeared in v9. Similarly, PatientInstructionsDAO.add() of v12 both extended UC11S2 (document office visit) of v4 and implemented a new requirement UC44S1 (patient specific instructions) of v12.

Recognizing the delayed update of new requirements also helps explain the large proportions of "others" in Fig. 3. Although some methods were added to the code base in a particular release, the requirements that these methods were implementing might appear much later in the project repository. One explanation is that the developers perceive future requirements and start experimenting them with the current code base. For example, several new methods containing "CDC" in their names were created in v17, possibly foreseeing

features related to the U.S. Centers for Disease Control (CDC). However, "CDC" was not mentioned in the requirements at all (v17, v18, and v19). The use of such foreseeably targeted requirements was reported as part of just-in-time requirements engineering [12]. Developers can take advantage of the traceability information of the targeted requirements to refactor the code to facilitate the fulfillment of the upcoming requirements.

Another possible reason why requirements are documented later is that complete implementation of the requirements has not been achieved yet, even though partial implementation may be already on the way. This incremental delivery fits well with agile development. However, it presents a specific challenge on keeping the requirements and the RTM in sync with the continuous code changes.

Limitations. A major limitation of our study is to rely only on the name and full path to distinguish new method. Original analysis [13] and heuristics derived from observations [9] can help improve the accuracy of new methods' identification. Another limitation is that we examine only additions as the software evolves but ignore other changes such as removals and refactorings. Clearly, when the mass requirements update was conducted from v10 to v11, not only a great number of new requirements were added (140), but many outdated requirements were removed as well (52). Analyzing other types of changes will likely offer new insights and implications. For the traceability-related analyses, we select only 3 release versions with highest number of new methods. Looking into other versions will provide more data points to uncover patterns; however, as our current analyses show, considering only two consecutive releases can miss important change correlations across multiple versions. Finally, our analyses are driven by the source code release dates. Although we feel the choice is reasonable for iTrust, sticking strictly to the release date may pose some threats. As mentioned before, iTrust uses the project wiki to maintain and evolve requirements artifacts. Oftentimes, after a code version was released, requirements would experience some changes ranging from a few days to a few weeks [9]. The practice may be unique to student projects like iTrust, but taking project-specific release practices into account is important.

IV. Summary

Researchers working on requirements tracing need datasets to evaluate their approaches. Although many open-source software projects exist, few maintain requirements evolutions let alone the traceability information about the requirements. Having these valuable pieces of information has made iTrust a well-adopted traceability benchmark used in numerous studies. Our work is motivated by the question concerning the accuracy of manually maintained RTMs. The initial analyses reported in this paper show that the RTM released together with iTrust's source code was never updated, despite considerable changes at the requirements and code levels happened in software evolution. One possible cause of RTM's stagnancy, according to our analyses, can be the asynchronous evolution paces of code and requirements. In another word, because iTrust's RTM records the implementation relationship between requirements and code, the update of RTM relies heavily on the update of requirements and the update of code. When the code and requirements updates are not very well coordinated, keeping the RTM up-to-date faces significant challenges.

Approaches supporting automated identification of outdated requirements began to emerge [9]. Building on this research thread, our work further contributes 3 types of requirements changes (extension of an existing requirement, implementation of a new requirement, and realization of a previously unfulfilled requirement) that could be applied to an evolving software project. Our future work includes conducting more in-depth studies on iTrust and other software projects and developing automated ways to updating requirements and their traceability information.

Acknowledgement

Funding for this work was partially provided by the U.S. National Science Foundation (Grant CCF-1350487) and the National Natural Science Foundation of China (Grant No. 61375053).

References

[1] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, January 2006.

[2] L. Chen, "Continuous delivery: huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, March/April 2015.

[3] D. Cuddeback, A. Dekhtyar, and J. H. Hayes, "Automated requirements traceability: the study of human analysts," in *International Requirements Engineering Conference (RE)*, Sydney, Australia, September-October 2010, pp. 231–240.

[4] A. Meneely, B. Smith, and L. Williams, "iTrust electronic health care system case study," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012, pp. 425–438.

[5] N. Niu and A. Mahmoud, "Enhancing candidate link generation for requirements tracing: the cluster hypothesis revisited," in *International Requirements Engineering Conference (RE)*, Chicago, IL, USA, September 2012, pp. 81–90.

[6] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *International Conference on Program Comprehension (ICPC)*, Passau, Germany, June 2012, pp. 183–192.

[7] W. Wang, N. Niu, H. Liu, and Y. Wu, "Tagging in assisted tracing," in *International Symposium on Software and Systems Traceability (SST)*, Florence, Italy, May 2015.

[8] N. Niu, A. Mahmoud, Z. Chen, and G. Bradshaw, "Departures from optimality: understanding human analyst's information foraging in assisted requirements tracing," in *International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, May 2013, pp. 572–581.

[9] E. B. Charrada, A. Koziolek, and M. Glinz, "Identifying outdated requirements based on source code changes," in *International Requirements Engineering Conference (RE)*, Chicago, IL, USA, September 2012, pp. 61–70.

[10] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. of IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[11] N. Niu and S. Easterbrook, "Analysis of early aspects in requirements goal models: a concept-driven approach," *Transactions on Aspect-Oriented Software Development*, vol. III, pp. 40–72, 2007.

[12] N. Niu, T. Bhowmik, H. Liu, and Z. Niu, "Traceability-enabled refactoring for managing just-in-time requirements," in *International Requirements Engineering Conference (RE)*, Karlskrona, Sweden, August 2014, pp. 133–142.

[13] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, February 2005.