Visualizing Testing Activities to Support Continuous Integration: A Multiple Case Study

Agneta Nilsson, Jan Bosch, and Christian Berger

Software Engineering Division, Dpmt of Computer Science and Engineering, Chalmers University of Technology, University of Gothenburg, Gothenburg, Sweden

Abstract. While efficient testing arrangements are the key for software companies that are striving for continuous integration, most companies struggle with arranging these highly complex and interconnected testing activities. There is often a lack of an adequate overview of companies' end-to-end testing activities, which tend to lead to problems such as double work, slow feedback loops, too many issues found during post-development, disconnected organizations, and unpredictable release schedules. We report from a multiple-case study in which we explore current testing arrangements at five different software development sites. The outcome of the study is a visualization technique of the testing activities involved from unit and component level to product and release level that support the identification of improvement areas. This model for visualizing the end-to-end testing activities for a system has been used to visualize these five cases and has been validated empirically.

Keywords: continuous integration**,** software testing, and visualization.

1 Introduction

Software development companies are increasingly striving towards continuous integration in their efforts to deliver high quality software faster and faster. Continuous integration is an agile development practice, which has become increasingly popular with the growing agile movement [1], [2]. The agile movement advocates flexibility, efficiency, and speed to meet the ever-changing customer requirements and market needs. Continuous integration is about *integrating* software parts in order to assembling a complete system *continuously*, i.e. frequently, and throughout all phases in the development cycle [3].

The anticipated benefits of continuous integration are e.g. to improve release frequency and predictability, increase developer productivity, and improve communication [4]. However, there is currently no consensus on continuous integration as a single homogeneous practice, and there are great differences of experienced benefits in both literature and practice [4]. Many software development companies are struggling to achieve continuous integration and the benefits they were expecting.

One common and well-known bottleneck when introducing continuous integration is testing [1], particularly for large and complex software with many dependencies such as software intensive embedded systems. Efficient testing arrangements are the key for achieving continuous integration, and arranging these are highly complex. The complexity faced by developers and test engineers often leads to problems such as double work, slow feedback-loops, too many issues still present in the post-development, disconnected organizations, and unpredictable release schedules due to issues identified lately. In this multiple case study, we explore current testing arrangements at five different software development sites (four companies). We aim to improve our understanding of how testing activities are arranged, and how to support companies in their efforts towards continuous integration by providing right communications means about test efforts throughout the process.

The rest of the paper is structured as follows: In Section 2, related work is discussed; Section 3 describes the approach, which we have followed to realize our case study with five different development sites; Section 4 summarizes the challenges in testing, with which the companies that we have investigated are faced; Section 5 introduces our "CIViT" Continuous Integration Visualization Technique that helps companies to describe intuitively their testing efforts. Section 6 validates our CIViT model before our article closes with a conclusion and an outlook for future work.

2 Background and Related Work

Continuous integration is the consequent continuation of applying unit tests and automated regression testing as quality assurance during the software development. In its most agile variant, teams evolve towards test-driven (TDD) or even test-first development, where new code is only added to fulfill the previously added test cases, which formally describe the function to be realized [5]. Large companies even of the size of Google are nowadays not only able to implement the concept of test-driven development for a large variety of products ranging from "software only" up to "software/hardware" products"; they are also able to coach and improve their product development teams towards a better project hygiene and better and faster ways of testing [6].

The goal for this paper is to make the currently applied testing activities explicit to all involved stakeholders. Therefore, a suitable visualization technique is the basis for the communication between these different parties. To evaluate existing visualization techniques, we addressed the following research question: Which visualization techniques for testing activities are available that (criterion 1) focus on the entire product deployment/release process, (criterion 2) are proven to be successfully applicable in industrial contexts, and (criterion 3) considered to be the important decision support methodology to improve the overarching testing processes towards continuous integration?

We re-evaluated the results from the systematic literature review [4], performed by one of the co-authors together with another researcher, to address the aforementioned research question. Holck and Jørgensen [3] analyze the continuous integration process on the example of open source software (OSS) FreeBSD (operating system) and Mozilla (web browser suite). Due to the different nature of OSS, which – as they state – is “focusing on stability and performance” in their examples, a comparison to business-value driven commercial products is difficult. Therefore, the aforementioned three criteria are not applicable to their examples. Furthermore, their work does not suggest a taxonomy for describing and visualizing the dependencies and drawbacks of the currently applied software quality assurance process.

A number of studies report on various ways to communicate build status. In the work from Sturdevant [7], the adaptation of the freely available tool CruiseControl is outlined with the focus of the support for test process itself. Concerning the aforementioned criteria, the author focuses on end-to-end testing (criterion 1) applied at Jet Propulsion Labs (criterion 2) mainly considering cost-efficiency of testing (criterion 3). However, in contrast to our work, no specific visualization technique is suggested or applied with different dimensions to outline deficiencies of the surrounding test process.

Downs et al. [8] provide guidelines for build monitoring systems in their work. They focus on how problems from broken builds or defects committed to a centralized repository are utilized within a software development team. Concerning our research question, they investigate only communication among developers and testers (criterion 1) by conducting interviews with software developers and testers (criterion 2) to derive guidelines how communication channels between developers and testers should be utilized (criterion 3). In contrast to our work, they do not focus on visualization of inefficiencies of the overall test process.

Stolberg presents in his work a technical description of a continuous integration framework [9] by following Fowler’s checklist of 10 practices for continuous integration. Thus, he was able to visualize the status quo of the quality assurance process before and after implementing a continuous integration environment. Thus, his work focuses on the entire test and deployment process (criterion 1) using an experience report within an industrial setting (criterion 2). As results, continuous integration improved the quality assurance (criterion 3), however in contrast to our work, he did not propose a visualization scheme to unveil testing process bottlenecks.

Kim et al. present in their works [10], [11] a technical description of a test automation framework by applying the tool CruiseControl. They mainly focus on the test automation technology (criterion 1) applied to an industrial context (criterion 2) to improve the communication between involved stakeholders (criterion 3). However, they do not visualize the test process and its inefficiencies at large as we propose in our work.

Hoffman et al. [12] describe in their work how the tool chain cmake/ctest/cdash/cpack has been applied to a research lab of the Department of Defense. This work was mainly driven by employees of the supporting company of the aforementioned tools and thus, should be considered as an experience report. With respect to our research question, they focus on all aspects of the test and deployment process but from the tool support perspective (criterion 1) in the context of a research lab for high-performance computing (criterion 2). They conclude that the proposed tool chain is effective (criterion 3), however, they do not describe how to visualize the current status of the test process at large to derive improvement initiatives.

Ablett et al. [13] present “BuildBot” as a means to enforce the fixing of broken builds. With respect to our research question, they focus only on results from a continuous integration server (criterion 1) evaluated among a group of students (criterion 2). Thus, no evidence is given for industrial benefits (criterion 3). In comparison to our work, they do not focus on the overall test process and a subsequent visualization of its dependencies and deficiencies.

Yuksel et al. [14] present a technical description of a test automation framework. Regarding the aforementioned criteria, they are partly focusing on the entire process, which still includes some manual tasks (criterion 1), to ensure the quality of multi-platform control system (criterion 2); they achieve an improved quality of the code but they do not visualize interdependencies or deficiencies of the overall test process. However, they summarize the status of automation and periodicity in a tabular representation, which includes similar dimensions as we propose for our visualization technique. In contrast to our technique however, they do not describe the test process in a comparable granularity including dependencies to derive actions for the responsible management for improving the test process.

Lacoste describes in his work the introduction of continuous integration for the tool “LaunchPad”, which is used by several open source software projects [15]. Concerning our research question, his work focuses to run test-suites before integrating newly added features (criterion 1) by applying them to the widely used software “LaunchPad” (criterion 2). As a result, the testing process could improve its efficiency (criterion 3) but in contrast to our work, no visualization of interdependencies and bottlenecks is provided.

Goodman and Elbaz provide an experience report [16] focusing on the entire deployment/release process (criterion 1) for an industrial project (criterion 2). Their work confirms the need of an adequate visualization scheme for the test process to make inefficiencies in the infrastructure explicit to the management e.g. to take action on improvement initiatives (criterion 3). However, they do not propose a visualization methodology as we do in our work.

Downs et al. analyze the impact of ambient systems to notify about the build status [17]. They primarily focus on the influence of notification means like lighting devices on the quality of commits from developers, i.e. if the number of failed builds decreases while the total number of all commits is still similar. With respect to our research question, they focus only on build status notification (criterion 1) by evaluating their hypotheses among team members of an agile team in an industrial setting (criterion 2) to investigate the impact of these ambient devices within a software development process (criterion 3). However, a visualization scheme as proposed by our work is neither outlined nor addressed.

To summarize our findings on the aforementioned research question, continuous integration as the fundamental principle of the software development, testing, and deployment process in agile teams is implemented increasingly. However, we are not aware of any work, which proposed a structured and easily applicable test process description and classification scheme, which unveils interdependencies and bottlenecks of the overarching test process to derive test process improvement initiatives. Furthermore, no other work exists that evaluates such a taxonomy systematically in a multiple case study.

3 Research Approach

In this paper, we report from a multiple case study [18] involving five software development sites from four companies that are striving towards continuous deployment of software. The four companies are large, developing complex software intensive embedded systems. The companies range in size from around 10.000 to 115.000 employees of which the number or R&D staff ranges from a few thousand to close to 25.000. Depending on the company, from 30% to more than 80% of the R&D staff work with software. As most of the existing research and industrial practice related to continuous integration is concerned with, typically, smaller companies in the SaaS and Web 2.0 domain, we believe that studying continuous integration at embedded systems companies is particularly relevant.

In our study, we have focused on specific sites within these companies with demarcated products and projects. Two companies are within the automotive industry, one company is within the defense industry, and one company is within the telecom industry. The first three companies can be described as largely doing traditional development with various degrees of agile practices established and moving towards continuous integration. The fourth company can be described as a company with some degree of established practices for continuous integration. The companies' existing testing infrastructure, tools, and ways of working did not sufficiently support a transition to continuous integration.

This research is conducted within the Software Center[1], a research center for collaboration between Chalmers University of Technology, the University of Gothenburg, Malmö University and seven software-intensive companies in the Nordics with the aim to conduct research projects together.

The research question we focused on was: *How can we visualize end-to-end testing activities in order to support the transformation towards continuous integration?* With end-to-end testing, we refer to all code, from code written by individual engineers to product release. The aim of this research is to gain insights into how to support the transition towards continuous deployment in the software development industry. The transition from traditional software development to continuous deployment is dependent on and intertwined in complex organizational structures and processes, which makes it particularly suited for a case study approach [18], [19].

Data collection was conducted through group-interviews, workshops, and complementing email correspondence to ensure triangulation of data [18], [20]. We conducted group interviews [21] at each company site, using a semi-structured interview guide, each lasting approximately 2 hours, which has been recorded and transcribed afterwards. Each group comprised of 5-6 people, and we conducted the interviews with both questions answered in a round-robin-style to make sure that all participants were heard, as well as facilitating ample free discussions as needed. The members of each group were working together and knew each other well in order to be sufficiently comfortable to discussing freely together. We covered questions on what testing activities each site were conducting, the frequency of these and how long

[1] www.software-center.se

the feedback loop for each testing activity was, and their experiences of challenges and enablers during their processes.

We arranged two joint workshops with representatives from the various research sites to jointly share and discuss the tentative results from the ongoing research, as well as to further discuss their situations, reflections, and ideas of how to proceed. Each workshop lasted approximately three hours, and the involved researchers took careful notes of these discussions.

In the data analysis, we focused on synthesizing the data from the different sites by identifying common denominators in their descriptions of their current testing activities. The two dimensions *scope*, and *periodicity* emerged during the group interviews as a common way of discussing the sites' testing activities. Each site described their testing activities starting from their lowest level, continuing with the subsequent levels until the released product level. We also focused on understanding how frequently these testing activities were conducted and how long time their feedback loops took. In the analysis, we translated the local labels used at the different sites to more general levels of scope (referred to as component, subsystem, partial product, full product, release, customer), and similarly a more general periodicity (referred to as immediate/minutes, hour, day, week, month, once/release), to create the CIViT model as presented in this paper that captures and reflects the overview of each site. Having identified the two dimensions and the various labels for these dimensions, we iteratively tried out various ways to illustrate the current testing activities at a site. Eventually a box with four squares emerged to represent the different testing activities (referred to as functional, quality, legacy, edge). In order to illustrate coverage of these testing activities, we introduced a color scheme, and a similar color scheme was introduced to illustrate the level of automation.

As a first validation of the model, we interpreted the data from each company and created a CIViT model of their current testing activities. These models were then shared with each company site and confirmed as an accurate representation of their current testing activities. Some minor adjustments were made based on discussions with some of the sites about the interpretations of some of testing activities, mainly regarding interpretation of data such as the level of coverage or regarding the scope whether it would be regarded as subsystem or partial product in their context.

A second validation was conducted when each company used their CIViT model to identify a box to focus on for improvement. Each site found the model helpful as a basis for discussions about the current situation, and to decide what area to target and in what way, e.g. increase periodicity, increase scope, increase coverage in any of the testing areas, or increase the level of automation. The selected improvement initiative was followed-up and again the model was used to identify the intended initiative and the outcome of the initiative.

4 Problem Statement

Verification and validation of software systems through testing of software is a widespread activity that has been studied extensively over the last decades [22].

Traditionally, in the waterfall model of development, the testing activities were performed at the end of the development process, after the implementation of the software had been completed and the organization would move on to testing and fault fixing. With the increasing popularity of agile development methods, industrial practice, and consequently research, has moved towards more frequent testing during the development cycle as the ambition is to being able to deliver at the end of each agile sprint, i.e. every 2-4 weeks. Some companies have even adopted approaches where every check-in of code results in the release of software, resulting in dozens of releases of the software per day [23].

The companies studied in this paper, which predominantly operate in the software intensive embedded systems industry, have had similar developments towards agile practices over the last decade. Some testing efforts are performed more frequently and in a more automated fashion. However, due to the complexity and size of the systems developed by these companies, it became clear that several challenges around the verification and validation of the systems produced by these companies remained. These challenges can be summarized as follows:

No End-to-end Overview of Testing in Companies: During the interviews at the companies, it became abundantly clear that very few people, if anyone in the company, had a holistic, complete overview of all the testing activities going on in the development process, ranging from the individual developer checking in code to the release of a system to customers. Everyone involved in the verification process understood their own part really well and, by and large, knew who they received software assets from, whom to deliver these assets after completing their task and how to report issues. However, there was very limited understanding of the end-to-end process. This problem resulted in several additional problems.

Significant Duplicate Testing Efforts: Due to the lack of understanding of the type and quality of testing performed by others and the ambition to minimize the number of faults that slip through, every activity in the testing process repeated significant amounts of testing already performed by others. This caused both longer testing cycles and a reduction of focus on the areas of testing best performed in the current step in the end-to-end verification process.

Slow Feedback Loops: In all interviews, the challenge of long feedback loops was raised as a key challenge. Even though virtually all companies employ forms of unit testing by individual engineers that give feedback within minutes, receiving feedback on the quality of the code from all perspectives, i.e. correct functionality, not breaking any legacy functionality and achieving the desired quality requirements, would typically take several weeks if not months. As an illustrative anecdote, one engineer received testing results about a month after returning from a six-month paternity leave on code that he had written before his leave.

Late Testing of Quality Attributes: A common challenge at the interviewed case study companies is that testing of quality attributes of the system, e.g. performance, robustness, upgradability, etc., took place late in the development cycle and that identified issues, e.g. significantly degraded performance, caused unpredictable development efforts late in the process at a time when the organization can least afford it.

Ad-hoc, Tactical Improvement Efforts: In the case study companies, the verification activity was viewed as a challenge that required improvement efforts. However, the improvement efforts that were presented tended to be mostly tactical in nature, driven in a bottom-up fashion by a team responsible for one step in the process and based on limited understanding of the end-to-end nature of the verification process and the key issues.

When analyzing the data from the interviews at the case study companies for root causes, we realized that the problems that were identified share a common root cause that, if addressed, alleviates these problems significantly: the lack of a holistic, end-to-end understanding of the testing activities and their periodicity, i.e. the frequency of executing the testing activities. Once the organization has a solid understanding of these issues, changing when and how testing activities are performed becomes significantly easier as well as it allows for much easier understanding of the implications of changes.

Based on our analysis of root causes, in the next section we present our solution for addressing the lack of holistic, end-to-end understanding. The model has been validated with the case study companies and currently used to drive strategic improvement activities in testing.

5 Continuous Integration Visualization Technique (CIViT)

Customers expect quality from the products that they receive from the manufacturer or system provider. Verification has been part of the development of software intensive systems for as long as we have written code and interestingly the practices around testing have evolved only slowly. In the case study companies, we have identified that many testing activities take place in different organizations with different coverage of the requirements of the system. In response to this, we have developed a visualization technique called CIViT to show all testing activities performed around a product or product platform. The visualization technique is used by the companies to address the challenges that were discussed in the previous section.

Table 1 presents the different testing activities and their frequency at each participating site. The suggested dimensions “product granularity” and “periodicity” evolved during our workshops with the industrial partners. Based on the companies’ feedback and reports, we could cluster their technical and organizational approaches into these two dimensions. Afterwards, we could also validate these dimensions with the work from Yuksel et al. [14], who are using a similar classification scheme but only in tabular form.

Next, we first introduce the types of testing that are visualized. Subsequently, we describe the scopes of the testing activities. Then, the periodicity of the testing activities is discussed. The section is concluded with an illustrative example of a CIViT model from one of the case study companies followed by a summary.

Table 1. Research sites and their key testing activities

Research Site	Testing activities, frequency, and time for feedback loop
Site 1	V1 (SW, minutes) V2 (SW, minutes) V3 (SW + HW, 2 weeks, 8 weeks) V4 (SW + HW, 8 weeks) V5 (System, 10 weeks) V6 (Real product)
Site 2	Design (unit/component/system level, seconds, 30m minutes, 8 hours) Function (system level, 8 -12 hours, 20 hours) System (system level, 4 hours, 1 week, x weeks) Integration (network, weeks) Solution () Customer ()
Site 3	Individual (unit/component, seconds) Team (unit/component/function/load, 10 min) Logical product (unit/component/function/full legacy, minutes; 2 hours, 3-30 days) Real product (feature/integration, daily, weekly, 12-14 weeks) Release (acceptance, year, once, 10 days) Customer ()
Site 4	Unit (seconds, minutes, daily) Lab (function, Subsystem (integration/verification, weeks, months) System (integration, weeks, months) Release (acceptance, two times)
Site 5	Unit (seconds, minutes) Subsystem (months) System (six months) Release (six months)

5.1 Types of Testing

CIViT is concerned with four types of testing: new functionality, legacy functionality, quality attributes, and edge cases. These types of testing are described in more detail below.

New functionality testing is concerned with testing the functionality that is currently under development. As agile methods typically encourage or demand test-driven development, the test cases resulting from TDD fall into this category.

The second category of testing is concerned with legacy functionality, i.e. functionality that has already been built and operates according to its specification. The importance of testing legacy functionality is driven from the desire to have expansion in the functionality that the system provides without relapses. This requires frequent testing of legacy functionality to ensure that ongoing development efforts have not caused unwanted side-effects where new functionality works but legacy functionality now fails to function.

Quality attributes constitute the third category of testing. Quality attributes such as performance, reliability, safety, and security are affected by ongoing development and it is important to ensure that these quality attributes do not start to deteriorate below the minimal acceptable level. Similar to the case of legacy functionality, the intent of frequent testing of quality attributes is to guarantee that the system continues to satisfy the quality requirements and to avoid a situation where late in a development cycle significant effort has to be dedicated to improving deteriorated quality attribute levels.

Finally, there is a category that is not often mentioned in the literature, but that, based on the interviews at the case study companies, we have experienced as important: edge cases. Edge case testing is concerned with testing really unlikely or weird situations that, often, originate from faults that slipped through to customers and that were discovered after significant investigative effort. The company obviously wants to avoid similar situations in the future and consequently adds test cases to test for these specific exceptional situations.

In Fig. 1, we show the four squares forming a bigger square. The "F" stands for new functionality and the "L" for legacy functionality. The "Q" represents quality attributes and "E" edge cases. Each smaller square can have one of five colors, ranging from red to green. The color of the square refers to the level of coverage of all test cases in the specific square. In the figure, the mapping between coverage and color-coding is shown in the upper right.

Furthermore, the line around the four squares can have one of three colors, again ranging from red to green, which indicates the level of automation of the testing. The lower right part of the figure shows the mapping between colors and level of automation.

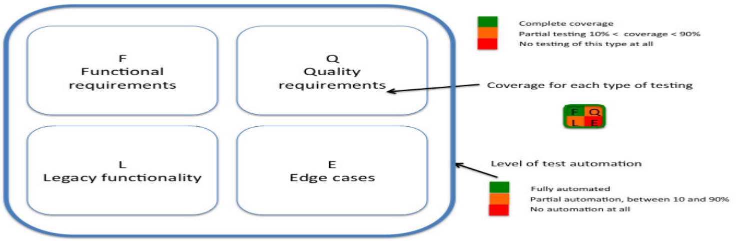

Fig. 1. The four types of testing in the CIViT model with explained color coding and mapping to coverage and automation. Red equals no coverage/no automation, orange equals partial coverage/automation, and green equals complete coverage/fully automated.

5.2 Scope of Testing

The second aspect of CIViT is the scope of testing. Scope, in this case, refers to the segment of the overall system that is tested as well as the level of trust that can be associated with the test results. CIViT is concerned with five main levels, ranging from a component, a subsystem, a partial product, the full product, on-site release testing and, finally, customer-site release testing. Below we describe each scope in more detail:

Component: A component or module is a part of the system that can be the scope of an individual engineers or a small team, in case of pair-wise programming. At this level, typically unit testing takes place.

Subsystem: In the case of component teams, a subsystem is often the scope of responsibility for a team or a small set of teams. At the subsystem level, the types of test cases are broader in the area of covered functionality and less white-box than the previous level.

Partial Product: Especially in the case of embedded systems, system level testing can only take place realistically in case some parts of the mechanics and hardware of the system are available and other parts are simulated. Frequently, companies build test rigs that combine the most important aspects in a structure that allows for testing the primary functional and quality requirements.

Product: No matter how accurate a test rig is in terms of providing a realistic testing environment, there still are significant needs to test the full product with all parts present, including mechanics, hardware and all software. The challenge with product-level testing is that the cost of providing the full product often is quite high and not all teams can have full and continuous access to the product. Also, in cases where the hardware and mechanics are developed in parallel with the software, the full product typically becomes available only late in the development process.

Release: Organizations are keen to minimize the number of issues that reach the customer and in response often create a separate release organization that tests the full product for all aspects that are of importance to the customer. The release organization is concerned with completeness of testing, including edge cases and quality attributes of secondary priority. Typically, the focus of release testing is on completeness and ensuring the expected functionality and quality at the customer site.

Customer: Finally, in the case of lower volumes, but highly priced embedded systems, the company often installs the system or product at the customer site and performs testing activities to ensure the correct operation of the system in the context of the customer.

Finally, it is important to recognize that the levels describing the scope of testing are not mutually exclusive, but rather the contrary. In practice, there are significant testing activities at each or at least at most levels.

5.3 Periodicity of Testing

Finally, CIViT is concerned with the periodicity of testing, which we define as a combination of the frequency of a testing activity and the time between the start of the testing activity and the availability of feedback from that testing activity. Again, we identify three levels of periodicity, i.e. “in the development workflow” (minutes and hours), “disrupting the development workflow” (days and weeks) and finally “outside the development workflow” (months and once per release).

Although the case study companies were quite pleased with giving feedback within days or one or a few weeks, the fact is that this is experienced as disruptive to the development workflow. In this case, the team working on a feature typically has moved on to other tasks and errors that are returned after days or weeks require the team to return to work that had already been completed. At this point, the engineers that originally implemented the feature need to stop working on what they were concerned with, make a context switch, make the change, submit the updated code and return to the task that they were working on now.

The even longer periodicity, i.e. months or once per release, often results in high-level, more complicated system errors that cannot be allocated to a team that did the original implementation. Hence, these tasks are assigned to dedicated teams or randomly to the development teams. Focusing on these errors is outside the normal development workflow in that the teams are not temporarily disrupted from development, but rather perform error fixing for a period of time without other tasks in the pipeline.

Finally, rather than providing exact feedback loop length, CIViT is concerned with indicating the order of magnitude of the feedback loop. For instance, the point “hours” indicates from one to a small number of hours, but clearly less than a day.

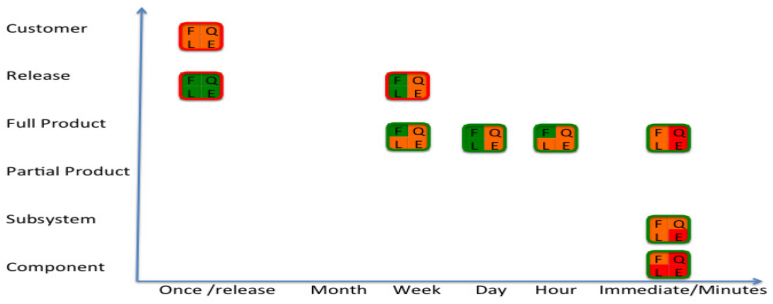

Fig. 2. An example instantiation of the CIViT model from one of the participating companies

5.4 Illustrative Example

To illustrate the CIViT model, below we show an example from one of the case study companies in Fig. 2. One can derive significant information from this chart, including the following:

- The organization uses automated unit testing for some parts of the functionality at the component level.
- At the subsystem level, automated testing of part of the functional, legacy and quality requirements takes place.

- The company does not employ partial product testing, but rather performs testing at the full product. Different tests take place every couple of hours, every day or days and every week or small number of weeks.
- Manual testing of all requirements and edge cases takes place once per release.
- Finally, at the customer site a subset of the requirements is tested to guarantee correct “in situ” operation of the system.

5.5 Summary

The CIViT model aims to visualize the testing activities that an organization deploys to achieve the desired quality levels during the development of a product or system. The model was developed in response to the observation that many researchers and, sometimes, even practitioners assume that the validation of a system or product occurs in a single location in the timeline of development as well as in the organization. In practice, this is obviously more complicated and the CIViT model visualizes this complexity while providing a simple overview that can be used for selection and prioritization of improvement activities.

6 Validation

In this study, we aim to improve our understanding of how testing activities are arranged, and how to support companies in their efforts towards continuous integration. We identified a number of challenges that remain around the verification and validation of the systems produced by the companies involved in this study. We developed the CIViT as a solution for addressing these challenges:

No End-to-end Overview of Testing in Companies: The CIViT model has been validated in a two-step process by all five cases in this multiple case study. As a first validation step, the data from each company were carefully analyzed and translated into the CIViT model for each company. Each company has reviewed and confirmed their model as a fair reflection of their end-to-end testing activities at the studied site. The feedback from the companies was positive that the model provides a useful overview of their end-to-end testing activities. The model helps to gain a clear overview and understanding of the end-to-end process of testing activities. As a second validation step we used the model for each company to identify what testing activities in their model that they would like to focus on to improve. Each company considered the model helpful as a basis for discussion about the current situation, and to decide what area to target on and in what way, e.g. increase periodicity, increase scope, increase coverage in any of the testing areas, or increase the level of automation. Each company selected a specific box in their model and explicated in what way they aimed to improve the selected testing activities, for example by increasing periodicity from e.g. month to week, or by increasing scope from e.g. subsystem to partial product. We followed-up the improvement initiative again using the model to discuss the intended initiative and the outcome of the initiative, and again the model proved useful as a basis for these discussions.

Significant Duplicate Testing Efforts: The overview provided by the CIViT model enables useful discussions that reveal what type and quality of testing that are performed within the settings. The study shows that this is helpful to identify unintended and undesired duplicate testing efforts, as well as to ensure that sufficient testing efforts are in place at the various levels of the end-to-end process.

Slow Feedback Loops: In a similar way, the CIViT model both visualizes directly the periodicity of the involved testing activities and consequently reveal their feedback loops in the settings, and enables useful discussions about what would be reasonable and desired times of feedback loops within the end-to-end process of testing activities.

Late Testing of Quality Attributes: The CIViT model also directly visualizes what different types of testing that are dealt with in the involved testing activities. For example, the study shows that this helps to reveal to what extent the testing of quality attributes, e.g. performance and robustness, takes place and when. As this is commonly dealt with late in the development cycle, the companies find the CIViT model useful to visualize the current end-to-end process of the various testing activities and that it serves as a useful basis for discussing reasonable and desired ambitions regarding the testing of quality attributes.

Ad-hoc, Tactical Improvement Efforts: Based on the overview that the CIViT model provides, it also enables useful discussions of the testing activities that are performed within the settings regarding what areas would be suitable to improve and how. This helps the companies to move away from the typical ad-hoc approach towards improvement efforts and have a better understanding of the end-to-end verification process and the key issues when they make decisions about what to do and how.

7 Conclusions and Future Work

In this work, we have collaborated with five software development sites from four companies affiliated with the Software Center. We have unveiled weaknesses and hurdles in the companies' evolution towards continuous integration to meet their customers and markets' needs.

Based on our findings, we have developed our holistic Continuous Integration Visualization Technique CIViT to provide a useful overview of end-to-end testing activities. Thus, engineers, testers, and managers have been enabled for the first time according to our studies on related work to see, understand, and act accordingly on an integrated and overarching test process.

The validation of the CIViT model carried out the involved companies confirms that the model serves as a solution to the lack of a holistic, end-to-end understanding of the testing activities and their periodicity. It also confirms that by enabling the organization a solid understanding of the end-to-end testing activities, it enables the organizations to identify how to best change their testing activities and to understanding the implications of changes. Companies that participated, as well as additional companies, have been using the model after the completion of the study and claim that the model is particularly useful as a basis for discussion, which help to identify problems and to reason about suitable measures.

While CIViT is our first step towards a simple and intuitive yet powerful visualization and test process improvement techniques, further aspects need to be addressed and investigated. As immediate next steps, we want to further analyze the motivations behind a selected test process improvement initiative. Furthermore, we need to understand commonalities and differences in the charts from the involved companies to derive guidelines where to focus on improvements and how to organize them.

Acknowledgments. This study would not have been possible without the help of the involved partners from the Software Center. Thus, we would like to thank all engineers, testers, and managers who participated in our study.

References

1. Fowler, M.: Continuous integration (2007), `http://martinfowler.com/articles/continuousIntegration.html`
2. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for the Agile Software Development (2001)
3. Holck, J., Jørgensen, N.: Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. Australian Journal of Information Systems 11(1), 40–53 (2004)
4. Ståhl, D., Bosch, J.: Modeling continuous integration practice differences in industry software development. Journal of Systems and Software 87, 48–59 (2014)
5. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
6. Whittaker, J.A., Arbon, C., Carollo, J.: How Google Tests Software. Addison-Wesley Professional (2012)
7. Sturdevant, K.: Cruisin' and Chillin': Testing the Java-Based Distributed Ground Data System 'Chill' with CruiseControl. In: Aerospace Conference 2007, pp. 1–8 (2007)
8. Downs, J., Hosking, J., Plimmer, B.: Status Communication in Agile Software Teams: A Case Study. In: Proceedings of the Fifth International Conference on Software Engineering Advances, pp. 82–87 (2010)
9. Stolberg, S.: Enabling Agile Testing through Continuous Integration. In: Proceedings of the Agile Conference, pp. 369–374 (2009)
10. Kim, E.H., Na, J.C., Ryoo, S.M.: Implementing an Effective Test Automation Framework. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, pp. 534–538 (2009)
11. Kim, E.H., Na, J.C., Ryoo, S.M.: Test Automation Framework for Implementing Continuous Integration. In: Proceedings of the Sixth International Conference on Information Technology: New Generations, pp. 784–789 (2009)

12. Hoffman, B., Cole, D., Vines, J.: Software Process for Rapid Development of HPC Software Using CMake. In: Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference, pp. 378–382 (2009)
13. Ablett, R., Sharlin, E., Maurer, F., Denzinger, J., Schock, C.: BuildBot: Robotic Monitoring of Agile Software Development Teams. In: Proceedings of the 16th IEEE International Symposium on Robot and Human Interactive Communication, pp. 931–936 (2007)
14. Yuksel, H.M., Tuzun, E., Gelirli, E., Biyikli, E., Baykal, B.: Using Continuous Integration and Automated Test Techniques for a Robust C4ISR System. In: Proceedings of the 24th International Symposium on Computer and Information Sciences, pp. 734–748 (2009)
15. Lacoste, F.: Killing the Gatekeeper: Introducing a Continuous Integration System. In: Proceedings of the Agile Conference, pp. 387–392 (2009)
16. Goodman, D., Elbaz, M.: 'It's not the pants, it's the people in the pants' Learnings from The Gap Agile Transformation. In: Proceedings of the Agile Conference, pp. 112–115 (2008)
17. Downs, J., Plimmer, B., Hosking, J.: Ambient Awareness of Build Status in Collocated Software Teams. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), pp. 507–517 (2012)
18. Yin, R.K.: Case study research: design and methods. Sage-Publications, Newbury Park (1994)
19. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131–164 (2009)
20. Patton, M.Q.: How to Use Qualitative Methods in Evaluation. Sage Publications, Newbury Park (1987)
21. Myers, M.D., Newman, M.: The qualitative interview in IS research: Examining the craft. Information and Organization 17, 2–26 (2007)
22. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Proceedings of Future of Software Engineering (2007)
23. Ries, E.: The Lean Startup: How Constant Innovation Creates Radically Successful Businesses. Portfolio Penguin (2011)