Integrating Testing and Implementation into Development

Maaret Pyhäjärvi, Conformiq Software Ltd.
Kristian Rautiainen, Helsinki University of Technology

Abstract: Cost of defects increases significantly the later the defects are found. Testing is the means to find defects, and we view testing in the broader perspective of maximizing customer satisfaction and providing feedback for process refinement, in addition to just detecting and getting defects corrected in the software. Testing is an integral activity in software development. Testing should be included early in the software development, but achieving this right-from-the-beginning integration of testing and implementation into efficient development has proven a challenge in practice. This article presents problems with use of the V-model as a reference model for this integration. We identify differences in testing in plan-driven and agile development approaches, and explain how to use a general framework for managing software product development, Cycles of Control, to structure the links between implementation and development and to use software development project dynamics for the benefit of the project.

Keywords: Software Testing, Test Planning, Agile Testing, Exploratory Testing

EMJ Focus Areas: Systems Engineering, Program & Project Management

Developing software is a challenging, intellectual task. The essence of software—complexity, conformity, changeability, and invisibility (Brooks, 1987)—makes introducing defects into the software in the construction process a fact. Software does not have a manufacturing stage in the sense of traditional products with statistical quality control, but stays in the design stage, also during software maintenance (Beizer, 1990). Testing in order to find the defects is an essential activity, but it can be difficult to integrate into software development. Effort estimates for testing vary from 30 to 90% (Beizer, 1990), with half of the development costs being the typical estimate. Test processes are considered excellent candidates for improvement (Rico, 2000), yielding results for Cycle Time Reduction, Productivity Increase, Quality Increase, and Return-on-Investment, such as 6-fold, 6-fold, 6-fold, and 9:1. Optimizing software development as a whole, consisting of implementation (including requirements) and testing, would be an opportunity for improvement.

The state-of-the-art model on how testing should be coordinated with development is the so-called V-model described in Craig and Jaskel (2002). The V-model forms the basis of what is taught on how to structure testing in relation to development. As such, it was also the basis for testing in our research on managing software product development in small companies. In applying the V-model in practice, it became evident that the model could be easily misused to form structures that are rigid and contain overlapping tasks.

The V-model depicts a plan-driven approach to developing software, as it is essentially based on the waterfall model, presented by Royce (1970) with more degrees of freedom than is depicted nowadays (Schach, 2002). In contrast to the traditional plan-driven software development processes, several agile process models have been proposed (e.g., Beck, 2000; Highsmith, 2000; Schwaber and Beedle, 2002). Empirical studies have shown that many companies in Internet software and PC software use flexible processes (Casumano and Selby, 1998; Casumano and Yoffie, 1999), and such flexible processes have been found to result in increased customer satisfaction (MacCormack et al., 2001).

According to Boehm and Turner (2003) we should tailor our development life cycles to balance between plan-driven and agile development. The agile processes do not give clear instructions on how to do testing, except for extreme programming (Beck, 2000), and basing testing on a plan-driven approach if our goal is realizing benefits of agility and flexibility is very difficult in our experience.

About the Authors

Maaret Pyhäjärvi works as senior testing consultant at Conformiq Software Ltd, a company specializing in software testing. She has worked as a tester and test manager in companies, as well as a researcher and teacher of software testing at Helsinki University of Technology (HUT). Her focus is on improving the testing process and integrating testing into development in different contexts and packaging lessons learned from various application areas.

Kristian Rautiainen received his MSc at HUT and is currently working on his DSc (Tech) degree. He has been teaching software processes at HUT since 1998 and his research interests include software processes and software engineering management. He is currently exploring how to manage software product development in SMEs.

Contact: Maaret Pyhäjärvi, Conformiq Software Oy Ltd., Lars Sonckin kaari 16, 02600 Espoo, Finland; phone +358 40 823 3777; maaret.pyhajarvi@conformiq.com

Referred management tool manuscript. Accepted by special issue editor Hans Thamhain. Previous version presented at HICSS-36 in 2003.
In this article we present our findings on applying the V-model in practice and suggestions on how to augment the V-model in efforts to integrate implementation and testing into development. With integration we mean that actual test execution, also for higher levels of tests, can take place early in development instead of just planning for testing to be done in the later phases of a project. We do this by using the V-model as a basis for defining the needed levels of testing and the Cycles of Control (CoC) framework (Rautiainen et al., 2002) to help understand the software development project dynamics and links between testing and implementation.

The rest of the article is structured as follows: research goals and methodology, the V-model and challenges in its use, combining the V-model and the CoC framework, and a short discussion and managerial implications.

Research Methodology
The research presented in this article has been performed as action research in close cooperation with several Finnish software companies, guaranteeing practical relevance. The article is based on research conducted in two phases. In the first phase the focus was on small software product companies, which were involved in the research in order to improve the management of their software development. The researchers' role was mostly consulting. We conducted semi-structured interviews and reviewed available documentation to understand the situation at hand as well as the situation after improvements.

In the second phase the second author continued the work with small software product companies, while the first author switched jobs to a company specializing in software testing. The first author was involved in a test process improvement (TPI) (Kooren and Pol, 1999) benchmarking of 15 companies, as well as improvement of the whole testing process of developers, testers, customers, and users. The focus was thus widened from small software product companies to include large software product companies and tailor-made software from the perspectives of customer and developer-made testing. In the benchmarking, as well as in the analysis of the situation with developer testing, we conducted semi-structured interviews and reviewed existing documentation. To understand the coordination between testing and implementation, we used the CoC framework as a basis, as the TPI model is focused on high-level testing by a separate testing group, with its basis in the V-model's built-in assumptions.

Summarizing experiences reported by others in literature helped us in explicating the reasons behind the difficulties we had observed in the companies. This helped us formulate our approach better. There seemed to be little information on the test process and its integration to the whole development lifecycle in the software engineering journals and conferences. Results of empirical and case research within the testing community is typically published in magazines such as Software Testing and Quality Engineering Magazine and in non-academic conferences, into which we widened our scope, having realized the mathematical nature of testing research in the academic conferences.

The V-Model of Testing
Overview. In the field of software testing, the V-model is the state-of-the-art taught on practically every course on testing. The V-model (see Exhibit 1) splits the testing process into levels on which testing is carried out incrementally in conjunction with system implementation.

The V-model is an extension of the simple waterfall model, where each process phase concerned with implementation has an associated verification and validation activity called test level. Testing on each level should be controlled to avoid overlapping. The V-model shows how the testing activity can—and should—be taken into account much before there is some source code to actually be tested. Traditionally the individual test plans for the test levels are set as the links between these activities (Pol et al., 2002; Craig and Jaskiel, 2002), coordinated with a master test plan and controlled with entry and exit criteria.

Unit testing and integration testing are typically seen as the low-level tests (Pol et al., 2002) and as such suggested to be
executed by developers themselves. Some interpretations see the integration level as a higher-level test and include new resources in testing this point of view. For someone working as a tester or test manager, as the first author has, it is typical to think of the separate group's testing as the system testing, if working within the organization developing software. If working at the customer or user organization, the same approach is applied to acceptance testing. For these high-level tests the lower levels are expected to have been performed, but these are typically planned by the testers to target as many of all possible defects as possible.

The V-model and its variations essentially bring forth three important points: testing a smaller part before putting it into a larger system is a good approach, there are several points of views to test for, and testing efforts can start with planning as soon as the higher level requirements have been identified.

Problems with the V-model. The V-model, as well as its variations, are just extensions of the waterfall model. The waterfall model includes the assumption that a complete set of requirements is defined in advance. There is no allowance for redefinition and redesign, but changes become more expensive as the process goes forward, because so much depends on what has already been done. Reluctance to fix bugs such as requirement bugs found in acceptance testing or specification bugs found in system testing, as these are seen as the V-model waterfall's last phases, places testers as change recommenders late in the project, when every change they recommend will carry a significant price (Kaner et al., 2002). Reviewing the requirements and specifications does not fully solve the problem, as people tend to know what they want only after it is tangible in their own perspective. Late in the V-model, the features have been designed and committed to and the available money has been spent on them. The remaining free variables at the time of high-level tests are time-to-completion and quality. To counter these risks, incremental development is an increasingly popular mode of development (Iansiti and MacCormack, 1997). Basing testing on the expectations set by the V-model in such a context is difficult, especially due to the increased role of regression testing invisible in the V-model.

One important context in which the V-model is considered to be appropriate is contract testing if a customer is likely to change his mind and unlikely to willingly take responsibility of the costs associated with the changes. The V-model's built-in waterfall creates a clear contract with clear rules for allocation of risk (Kaner et al., 2002). After signing off requirements, every change request is a scope change associated with a cost to the customer. This may not serve the customer best, and for in-house development, the cost structure is not favorable.

The V-model used as a basis for creating schedules results in defining a testing phase—a reserved time for the activity—for each of the levels. The levels should, however, be interpreted as continuous activity with a perspective. The form in which the V-model is presented communicates this idea poorly. These structures are, in our experience, common and ineffective, as well as difficult to change.

Having given up on the waterfall assumption for testing, the structure for coordinating testing with the V-model does not provide enough details to support the integration of testing and implementation. From such a context, the V-model as a basis for testing activities has been strongly criticized (Marick, 1999). Marick points out that any model for testing that is just an extension to a development model will not suffice in providing the structure testing needs. The essential problems with these models are that they ignore that software is developed as a series of handoffs and these handoffs are the structure that enables testers to provide early value for the project. Each handoff changes the behavior of the system and thus the need for intelligent ways of focusing regression testing is needed. Basing the testing process on assumptions about existence, accuracy, completeness, and timeliness of development documentation is not always feasible. There will always be changes and tests cannot be designed based on a single version of a document. Also, it is not necessary to execute all tests derived from a single document together, but the tests should be designed based on outputs of each activity, e.g., level of requirements documentation, and then execute all tests in the order that makes the most sense, not the order dictated by the V-model.

Marick's ideas (1999) were criticized by traditional testing V-model advocates (Goldsmith, 2002; Goldsmith and Graham, 2002a; Goldsmith and Graham, 2002b) as claiming that system development can proceed without adequate requirements and that it is predicated upon poor practices just because they are common. They note that, "if you leave test design until the last moment, you won't find the serious errors in architectural and business logic until the very end" (Goldsmith and Graham, 2002b), which assumes that, for example, system testing would be a phase, instead of a continuous activity. We see this as a typical example of conflict between advocates of agile and plan-driven approaches.

The V-model is essentially document-driven. Our experiences with product development have shown that documents are important to the level they are actually used, either internally or by the customer. This is especially true in a small company context where there is always a tradeoff to be made with every hour of effort used.

Testing is supposed to find bugs, and correcting these bugs changes the program. Even if the need for correction is explicited as in Spillner et al. (2002), there would still be several levels of regression testing to be performed. Even if the contents of the test levels would otherwise be specified, this has proven difficult in practice.

The V-model advocates early test design, definition of test cases as soon as the appropriate level of documentation has been finished. It has been suggested that the V-model's early test planning approach would help programmers avoid defects by using detailed test cases testers have written based on first versions of documentation, like reported in Kaner et al. (2002); however, reviews and inspections are likely to be more efficient in order to help correct defects early than relying on pre-writing tests that will never be run (Kaner et al., 2002). The idea should be that the views that testing can provide are used early to improve the specification (Beizer, 1990), not to define the details of test cases that might be thrown away and never used if plans change.

If several projects are coordinated together for implementation of one system, applying the V-model has been deemed difficult. One system's acceptance tests should take place before system testing the whole, and when changes are made the acceptance test may need to be re-executed for every installation of the subsystem in the shared environment. System testing such a large system should be a continuous activity.
Exhibit 2. Factors Discriminating Plan-Driven and Agility

<table>
<thead>
<tr>
<th>Factor</th>
<th>Plan-driven Discriminator</th>
<th>Agility Discriminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Methods evolved to handle large products and teams; hard to tailor down to small projects</td>
<td>Well matched to small products and teams; reliance on tacit knowledge limits scalability</td>
</tr>
<tr>
<td>Criticality</td>
<td>Methods evolved to handle highly critical products; hard to tailor down efficiently to low-criticality products</td>
<td>Untested on safety-critical products; potential difficulties with simple design and lack of documentation</td>
</tr>
<tr>
<td>Dynamism</td>
<td>Detailed plans and &quot;big design up front&quot; excellent for highly stable environment, but a source of expensive rework for highly dynamic environments</td>
<td>Simple design and continuous refactoring are excellent for highly dynamic environments, but present a source of potentially expensive rework for highly stable environments</td>
</tr>
<tr>
<td>Personnel</td>
<td>Need a critical mass of scarce experts able to tailor methods during project definition, but can work with fewer later in the project, unless the environment is highly dynamic. Can usually accommodate some people able to follow step-by-step instructions</td>
<td>Require continuous presence of critical mass of scarce experts able to tailor methods; risky to use non-agile people, who need step-by-step instructions</td>
</tr>
<tr>
<td>Culture</td>
<td>Thrive in a culture where people feel comfortable and empowered by having their roles defined by clear policies and procedures; thrive on order</td>
<td>Thrive in a culture where people feel comfortable and empowered by having many degrees of freedom; thrive on chaos</td>
</tr>
</tbody>
</table>

In small companies and projects with less than 10 developers, it has been difficult to separate one level from another in a practical fashion. Implementing all levels has resulted in confusion of the differences that are difficult to explain. Especially with continuous system testing by a separate group, integration testing is done from a technical perspective as part of unit testing the implemented interface, and incremental system testing checks the newly available features, leaving separate integration testing difficult to justify.

There are several variations of the V-model, each complementing some of the difficulties in communicating testing with the model. Examples of variations in number of levels are: British Computer Society (1999), IEEE (2001), and Beck (2000); variation in explication testing and implementation on both sides of the V-model in Spillner et al. (2002); and applying V-model with iterations in Brockman and Notenboom (2003).

Combining Plan-driven vs. Agile Approaches

Differences in Development Approach. Plan-driven development and agile development have different home grounds and critical factors (Boehm and Turner, 2003). The critical factors are summarized in Exhibit 2, which is a direct quote from Boehm and Turner (2003) on how the differences on home grounds and risks are seen in practice.

A risk based approach for selecting and balancing between a plan-driven and agile approach is suggested in Boehm and Turner (2003). If agile risks dominate, the development approach should take the plan-driven approach and vice versa.

If the approach is plan-driven, applying the V-model, which has its roots in the same home ground, is applicable. If the approach taken in development is agile, testing needs to be structured differently.

Differences in Testing. Whereas the plan-driven testing could be referred to as “traditional testing,” comparison to the agile home ground and discriminators (Boehm and Turner, 2003) and agile values (Beck et al., 2001) shows that the agile testing approach would be exploratory testing, described in Bach (2003). One should note that exploratory testing is not plan-driven, but it is still a planned and systematic approach in testing. In Exhibit 4 we have summarized some of the essential differences in how testing is described in these approaches. As a representative of the plan-driven approach, we use the ISEB certificate (British Computer Society, 1999) and as a representative of the agile approach we use the ideas presented in “Lessons Learned in Software Testing” by Kaner et al. (2002). As insightfully noted by Boehm and Turner (2003), projects may need tailoring of their lifecycle to include both of the approaches, and this is what the context-driven school of testing (Kaner et al., 2002) also suggests.

When coordinating testing and implementation as parts of development, we have essentially four ways of combining plan-driven and agile implementation efforts and plan-driven and agile testing. We have experiences in implementation and testing being plan-driven and both being agile-oriented and working together productively; however, mixing the approaches is an interesting question. Our experiences have shown that introduction of agile working methods in a small company context may be severely handicapped by having a plan-driven testing approach. We have little experience in combining plan-driven implementation and agile testing, but the cultural barrier in introducing that seems to be quite high.

Our Approach. To augment the V-model in efforts to integrate implementation and testing into development and bridge the two views, we suggest a two-phased approach. First, we suggest that the positive in the V-model should be emphasized, and the negative impact the typical interpretations based on the form of the model should be minimized. As suggested above, three important lessons in the V-model apply: early involvement of testing, testing smaller parts first to minimize time for locating the defect, and use of several points of view. This
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Plan-driven Testing</th>
<th>Exploratory Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order of test execution</td>
<td>Low-level tests are executed before high-level tests</td>
<td>Testing should take place on all levels in an order that makes most sense</td>
</tr>
<tr>
<td>Focus in controlling the testing</td>
<td>V-model and size of testable items on each level</td>
<td>Depth of testing for the system currently at hand based on its maturity</td>
</tr>
<tr>
<td>View of test strategy</td>
<td>Seen as something above test project plans, combining several projects</td>
<td>Seen as the project's tangible testing approach combining test techniques to quality criteria, project environment and product elements</td>
</tr>
<tr>
<td>Test design</td>
<td>Early test design emphasized</td>
<td>On-time test design emphasized</td>
</tr>
<tr>
<td>Test cases</td>
<td>Test case can be executed and interpreted by someone other than who designed it. The test case must define the expected output</td>
<td>Questions to the software that may also be open-ended, even though thinking of the hypothesis is important, documentation need varies</td>
</tr>
<tr>
<td>Test techniques</td>
<td>Preferably mathematical form, enables same results for different users</td>
<td>Preferably heuristic form, enables useful results for different users</td>
</tr>
<tr>
<td>Role in test design</td>
<td>Test design first by tester to find faults in high-level requirements</td>
<td>Test first design by developer</td>
</tr>
<tr>
<td>Culture</td>
<td>Control, in form of emphasizing entry and exit criteria in testing</td>
<td>Collaboration, in form of emphasizing testing as service providing value constantly</td>
</tr>
<tr>
<td>Types of processes</td>
<td>Step-by-step processes, requiring as little interpretation as possible</td>
<td>Backward-forward processes, leaving room for deciding when each process phase should be done</td>
</tr>
<tr>
<td>Role emphasis</td>
<td>Emphasizes test manager and the manager's skills</td>
<td>Emphasizes each tester and his skills</td>
</tr>
<tr>
<td>Focus of improvement</td>
<td>Focus on developing methods</td>
<td>Focus on developing skills</td>
</tr>
<tr>
<td>View on practices</td>
<td>Best practices exist and should be applied</td>
<td>All so-called best practices are just heuristics, rules of thumb</td>
</tr>
<tr>
<td>Requirements specification</td>
<td>If you don't have a specification, you can't test</td>
<td>Requirements are useful fiction at best</td>
</tr>
<tr>
<td>Validation and testing</td>
<td>Testing is verification and validation is done if needed</td>
<td>Validation is an essential part of testing</td>
</tr>
</tbody>
</table>

is done with selection of appropriate testing levels. Second, we suggest applying the CoC framework to find the different timeframes in which testing and implementation are integrated most effectively.

Selecting Testing Levels. When structuring testing and implementation in development, the first thing in structuring testing to fit the needs is to assess the goals and needs of each of the test levels. Each level carries a handoff and communication overhead, and thus the number of levels should be minimized. In the extreme programming model (Beck, 2000), the unit and integration testing levels are combined to form the unit testing perspective by developer. The system and acceptance testing levels are combined to form the acceptance testing perspective by the customer. As integration takes places all the time in very small increments and the system is built based on customer stories of features, this combination is rational.

Exhibit 4 describes the options to consider with test levels. Based on our experiences, we identify factors that influence the selection of test levels. As a general rule, we suggest minimizing the number of test levels applied.

Applying CoC Framework. In efforts to understand software product development and how to control it, a framework for managing software product development was introduced, called Cycles of Control (CoC) (Rautiainen et al., 2002). With the limitations in the V-model as described, our experiences suggest that CoC can be effectively used in communicating the interfaces in testing and development to facilitate both plan-driven as well as agile approaches.

The details of testing provided in software process models do not help testers understand their role in relation to the process. A tester's role is to find and report defects and verify that the reported defects have been resolved, either by a programmer fixing them or by management deciding that they will not be fixed for some reason. The CoC framework helps in understanding testing in relation to other software development activities. It sets four timeframes on which one needs to address certain issues in development. The time frames—depicted as cycles—are presented in Exhibit 5. The leftmost cycle, named Strategic Release Management, organizes and deals with all ongoing major activities requiring attention from product development. Release Project Management deals with issues on the level of individual
Exhibit 5. The CoC Framework

projects aiming for a product release. Increment Management deals with managing individual increments producing a part of a release project’s deliverables. Heartbeat deals with coordinating status within the project or team. From the perspective of testing, we can separate two types of heartbeats: management heartbeat could be a biweekly meeting to coordinate what has been done and what will be done next, whereas implementation/testing coordination heartbeat could take the form of daily-build/daily-test. This will help the testers stay in pace with the developers and both parties can use the latest information. At the end of an increment, the team should have completed a feature or set of features that are tested from some perspectives. For example, functional testing should have been done to verify and validate the added functionality. Using short increments, e.g., one month, makes completing all testing at the end of each increment impossible. For example, customer acceptance tests can be done only when the system including the new increment is delivered to the customer. Also, time-consuming testing, such as running different performance tests cannot be done in time for the end of the same increment in which the tested functionality has been developed. Therefore part of the testing is always done in parallel rather than in pace with implementation. This should be considered when planning the project, i.e., through planning a stabilizing increment at the end of the project, where no new functionality is added, but rather the testing is allowed to “catch up” and the bugs are fixed.

Discussion and Managerial Implications

In this article, we discussed the use of a general iterative and incremental framework defined for managing product development—CoC—to help with integrating testing into software development projects. We suggested that the number of test levels should be minimized and some of the typical interpretations of the V-model avoided. To provide the detail of integration missing from the V-model, we suggest use of CoC-framework to assess the different time frames of handoffs and communication between roles and teams. Our experiences suggest that added detail in the pacing in CoC is critical for success in the integration.

We conclude this article with implications to planning development that project managers and test managers should explicitly take into account in planning software development:

- In scheduling a project, test levels should not be addressed as phases, but as continuous activities.
- The test phases to be defined in schedules should proceed by depth of test from smoke test (basic functionality) to function test (testing one functionality at a time to capability test (testing several functionalities but limiting data) to reliability test (testing several functionalities with varying data).
- The actual differences in different test levels should be addressed and the number of levels minimized; however, at
least two levels are needed to provide the technical and the end-user perspectives. Combining several projects brings forth new levels of testing needed.

- Instead of basing a test level’s tests on a level of requirements, plan for designing and executing tests in the order that makes most sense for the project as early as possible
- Early involvement of test execution requires projects to focus on architectures that enable early testing; typically it should be possible to add and test features one by one.
- Instead of early test case definition to documented test cases, consider including the testing perspective in reviews.
- Enable on-time feedback for developers from testing to control the testing costs.

Our efforts to research this area continue. We aim at understanding more on dynamics of levels and mixes of levels and timescales for different types of development projects, as well as being able to provide more detail to contexts influencing different types of selection in development and testing as an integral part of it.

References
Goldsmith, Robin F., “This or That, V or X?” Software Development (2002).