Time for Testing

at an intermediate Dutch SME

For confidentiality reasons, various names have been altered and some sections have been removed

Committee *University of Twente*
Dr. Roland Müller
Dr. Chintan Amrit

Topicus
Martin Krans
Wouter de Jong

Author Yoni Meijberg, Bsc.
Graduated 13-01-2009
Bachelor program
Industrial Engineering and Management,
University of Twente

Thesis Deventer, 02-01-2008, *private* version 1.00

MANAGEMENT SUMMARY

Recent development projects as well as their worried project managers show an urge for improving software quality by proper testing. High post-release defect levels, uncertain software quality and lack of testing methodology underlie this urge.

A case study was performed on two of the six SME business units to first assess current testing workings and thereafter to identify a desired future testing state. The various applied research methods – literature studies, semi-open interviews, and a survey – together provide a multi-perspective overview of SME' testing methodology.

The research shows testing at SME to reside in a troubling state, probably in a worst-case scenario. This is because various aspects of testing are considered weak: responsibilities are too informal, there is a lack of testing knowledge amongst employees, testing holds a low priority, there's a lack of testing resources, quality management lacks objectivism and finally customers are offered little guidance in proper testing. Furthermore, all types of testing – except for acceptance testing – are hardly applied and testing is performed without a methodology.

To improve these weaknesses, three best-practices have been identified out of structured literature studies on agile testing methods. These are: *Test-Driven Development*, which holds a new paradigm on testing that shortens and increases amounts of test cycles; *Continuous Integrated Testing*, forcing regression and integration testing via automated test runs; and finally the use of *Metrics* to provide real-time insights in software quality and performed tests, enabling planning, steering and control.

Together these practices will solve current testing methodology issues, and improve software quality significantly by reducing defect injection rates and by finding and fixing defects closer to their origin.

PREFACE

After a year this report is finally done. A lot has happened during the last months. Discovering the researcher in me next to the always oppressing consultant, speaking with and to enthusiastic minds, endlessly reframing the research, rewriting the report again from the bottom up, taking on several challenging – but time costly – side jobs at SME and a tragically ended personal relationship. All this has come to an exceeded normally required time for an assignment like this, but the result is there: a report that is meant to shock at first while disclosing valuable aid in building a brighter (testing) future later on.

I want to pay my thanks to all interviewees and survey respondents for supplying me with their opinions and lending me a helping hand from time to time.

Major thanks comes to Martin Krans and Wouter de Jong at SME, and Ronald Müller and Chintan Amrit at the University of Twente for making this research to what it is now and by helping me grow on both a personal and professional level.

TABLE OF CONTENTS

CONTENTS

This report is divided into six chapters. Chapter 1 introduces SME and the business case underlying this research. Chapter 2 enlists the research phasing and three applied research methods. Chapter 3 is the first chapter to discuss results; here insights into the troubled current situation of the test process are provided. Chapter 4 elaborates on current test performance by showing limited use of various test types and underlying agile principles. Chapter 5 shows a glimpse of the future, by listing three identified agile best-practices. Chapter 6 thereafter wraps up research results, by showing an urgent need to improve. This is supplemented by reflecting on what the best-practices are likely to improve when implemented. It concludes with some exit arguments as well as future research possibilities.

1 INTRODUCTION

SME, a rapid growing software development business, is continuously seeking to improve its application development methodology. In the recent past a few projects have been coping with defects surfacing post release time, causing developers to rush bug-fixes in order to keep customers satisfied. But there should be a way to prevent this late defect surfacing. This is done via testing, a development activity known to be lagging at SME. This report answers questions about the current situation of SME' testing and delivers the newest agile testing best-practices to improve the detection and removal of defects. After all: "Your Software quality is only as good as the quality of your testing efforts" (Scott Ambler)

1.1 SME, an intermediate-sized SME

<<REMOVED FOR CONFIDENTIALITY REASONS>>

1.2 Case description

This section holds three underlying sections that together describe the business case. First the motive for this research is described in section 1.2.1: testing activities do not match up in quantity or quality when compared to other development activities. Secondly in section 1.2.2 the case sample is discussed: the subset of two SME businesses Energy and Hub was used. The third and final section (1.2.3) discusses the research focus.

1.2.1 Motive: get testing up to speed

One of the SME businesses (SME = Powered By SME, the holding company), the 'hub' SME –from which the remaining other businesses are subsidiaries – wants to improve the methods of testing as a part of their application development process. This is due to various reasons.

SME uses an agile approach for application development, by using small independent developer teams that work with short 2-to-4-weekly release cycles (except for the first cycle which usually takes months to build). While this general agile approach works well for SME, testing procedures aren't integrated into this approach yet. A lack of a structure in deciding which tests to perform and when is missed by developers and project management. They also want to be able to test earlier in the development process, so that errors are easier to fix because of earlier discovery. A final remark about the role of testing has to do with the traditional underexposure of testing when looking at the entire application development process.

While some expertise in testing is available – developers of course test their applications in various ways (often with help of end users) before release to customer – this knowledge isn't indexed and/or bundled into a commonly known and applied testing approach. One exception hereto forms the wide application of Use Cases as testing aid, where analysts and designers structurally create functional tests for acceptance testing.

During a short office tour developers expressed a legitimate interest in improving availability of testing methods and tools and have ideas and clear opinions on this subject. Especially the possibility of testing earlier received a lot of attention.

All of this has lead to the demand for the development of an agile testing framework which needs to fit the SME' general agile software development method.

1.2.2 Sample subset Energy & Hub

This research targets two SME businesses, 'Hub' and 'Energy'. These two businesses are treated as one case in this research. This is deemed possible because (1) both businesses currently operate in the same – financial – industry, (2) 'Energy' just recently (January 2007) split off from 'Hub', and (3) work processes are highly alike.

The other two SME businesses 'Transport' and 'Government' operate in different industries, develop other types of software, split off earlier and have more differentiated work processes (probably due to the same former mentioned factors). The less commonalities result in the conclusion that these businesses won't be targeted in this research. However, they will be cross-examined for best-practices to prevent reinventing the wheel.

Next to the above described inter-SME similarities, intra-Hub and intra-Energy development projects are highly alike, for they apply a common development approach. This proves that one fitting testing framework can be developed, fit for use at both businesses.

1.2.3 Research focus

During the assignment formulation some limiting conditions for this research were agreed on:

- Test processes will/can only follow lightweight approaches, matching current SME software development process agility.
- Only the testing phase(s) of the software development process will be included in this research.
- The application development process will be approached from a development angle geared towards testing, thus not from a total software quality approach.
- The level of research lies at an application level.
- The focus of this research isn't aimed at delivering an exhaustive list of tools and methods to improve application development, but more at improving and aiding the development processes.
- Test case development and test planning is excluded from this research, for another SME employee is highly involved herein and is making good progress.

1.3 Problem identification

This section holds three underlying sections which problems and questions shaped this research. Section 1.3.1 elaborates upon the variety of issues of software quality and testing activities at SME. Section 1.3.2 covers the main question of this research, how to raise software quality by proper testing. Section 1.3.3 divides this question into manageable smaller parts.

1.3.1 Preliminary Issues

From the preliminary sessions with project managers and research coaches Wouter de Jong and Martin Krans current troubles of testing were distilled:

- *Uncertainty about software quality*. Roll-out of applications at customers net few bugs (exceptions left amid) and customers are happy with the functionality of the software, but at the same time developers fear possible latent high-impact bugs.
- *Software quality can't be proven.* Customers often require proven software quality. Currently there are no objective metrics in place, so true statements on software quality cannot be provided. Another effect hereof is that project managers remain oblivious about software quality and thus cannot adequately steer developer effort upon maintaining/raising it.
- *Software defects (or: bugs) are uncovered too late in the development process.* Due to testing being performed as an (almost) separate phase at the end of the development process, bugs aren't uncovered timely and thus require disproportionate amounts of effort to fix compared to timely uncovering and solving. (Kan 2003)
- *Testing activities cripple under new-feature pressure* at the end of development / when nearing a release. Customers and project managers structurally choose for new features over tested earlier features when the deadline of a release approaches. This occurs practically every release, thus seriously suppressing testing effort.
- *Unknown testing effort.* There is no clear overview on effort awarded to testing. Project managers have no other option than to turn to their gut feeling on what is/isn't tested and to what amounts. That's because project managers regard test efforts as developer's own responsibility and thus they have no obligation to report what was tested and with what results. Without objective knowledge on testing efforts it is also practically impossible to steer developers on test effort.
- *Lack of vision on testing.* There is limited organizational knowledge on how to perform testing; some knowledge is available but remains still at a personal level. This results in the lack of a universally applicable way (or vision) to perform testing.
- *Lack of testing responsibility,* while developers are individually responsible for their work, there are no incentives in place to steer them to deliver tested software. Time has proven that this open responsibility approach nets inadequate testing attention.

1.3.2 Main question: improve software quality by proper testing

All the issues above lead towards the formulation of the main question:

*'How can the **testing process** be improved to raise the **software quality**?'*

Testing process The part of the development process concerned with testing (not per se as a separate phase). This is typically comprised of requirements analysis, planning, execution, reporting, and retesting.

Software quality Delivering software with fewer defects.[1]

1.3.3 Sub questions: current and target situation

The form of the main question targeting improvement over current practices asks for a division in an *As Is* and *To Be* situation (current and target). For these two situations separate sub questions have been defined. These are listed in the next paragraph. For easy reference each question refers to the section where it's answered. The colored columns correspond with Figure 2-1 of section 2.1 where the phasing and methodology behind this research is depicted. The three research phases have all been appointed a different color in the figure. This coloring helps the reader to comprehend what research methods where applied at what research phase.

[1] This narrowest sense of product quality is commonly recognized as lack of 'bugs' in the product. It is also the most basic meaning of conformance to requirements, because if the software contains too many functional defects the basic requirement of providing the desired function isn't met.
This product quality definition is often expressed in two ways: defect rate (e.g. number of defects per million lines of source code) and reliability (e.g. number of failures per n hours of operation). (Kan 2003)
The third common component of quality is customer satisfaction, in this research however this is considered beyond material, for this type of quality depends on much more than what can be touched by improving testing.

This limited definition of software quality is necessary for this research limits itself to testing. To reach higher software quality in a broader sense would require attention to all phases of software development and include a customer focus, which isn't possible in this study's timeframe.

The general methodology applied to attain answers to these questions is described in Chapter 2. Detailed methodology descriptions are included as separate sections.

Questioning	Corresponding Section(s)	Color Coordi-nation
Current Situation (As Is)		
Test process descriptive		
• What comprises the current test process?	3	
○ What are the general characteristics?	3.1	
○ What types of testing are applied?	3.2	
○ What test methodology is currently used?	3.3	
○ How was the current use of test methods realized?	3.4	
Test process performance		
• What is the level of performance of the current test process?	4	
○ Which indicators are used to measure performance?	4.1[2]	
○ To what level are tests executed?	4.2	
○ What is the current performance on the indicators?	4.1	
Target Situation (To Be)		
• What best practices should be adopted to counter weaknesses and/or improve strengths of the test process(es)?	5	
○ What best practices are available?	5.1 - 5.4	
○ Which one(s) should be adopted?	6.2.1	

Table 1-1 Research questions and their references

[2] There weren't any indicators in use and so an ordinal scale was used to express performance. The research question thus couldn't be formally answered, but an attempt was made to provide some sort of measurement.

2 METHODOLOGY

This chapter covers the applied research methodology. Within this research several research methods have been applied. Their relations to research phasing and each other are described in section 2.1. After having clarified the application of a variety of research methods, their application is described in detail in sections 2.2 – 2.5. Results have been separated from methodology descriptions except at the preliminary literature review, for it isn't directly linked to research questioning and thus otherwise its results wouldn't be explicitly shown. For completeness purposes its results are thus listed along with the methodology.

2.1 Research phasing

This research holds three research phases, corresponding with question categories as defined in section 1.3.3. Figure 2-1 shows these phases as well as their relations with the variety of applied research methods.

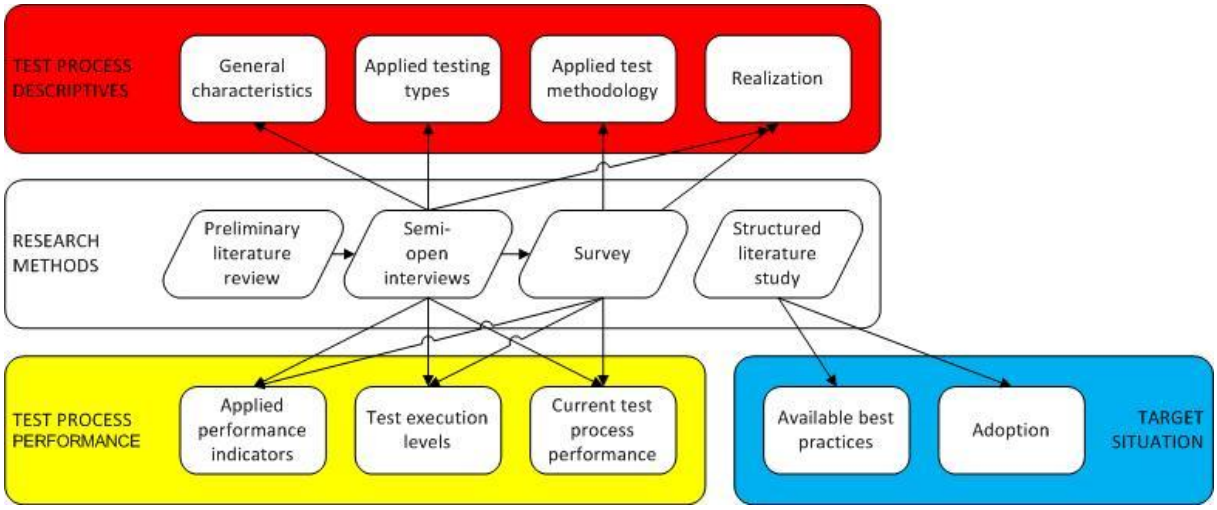


Figure 2-1 Research Methodology in phasing, activities and research methods

Each research phase is marked by a different color. *Test process descriptives* in **red**, *test process performance* in yellow and *target situation* in **blue**. Within every category their respective underlying research questions are depicted by white squares. In the centre of the figure the four applied research methods of this research are visible, depicted by diamond shapes. From these research methods lines are drawn that mark the usage of the four methods to solve individual questions. For instance the 'Structured literature study' research method answers questions 'Available Best Practices' and 'Adoption'.

N.B. During the entire research *observations* were applied when needed, as well as *quick office ask-arounds* to fill in small details. These aren't discussed as research methods, for they only served to supplement in providing small details.

2.2 Preliminary literature review

This research method was applied to get a feel in the fields of software testing and agile development. With the insights originating from retrieved journal papers, a first step towards the shaping of this entire research was set. The insights served as background information, allowing the researcher to shape a framework for asking the right questions in the following semi-open interviews.

The literature review was performed in a structured way. First a set of search key words was formed to guide the search. Applied key words were: “agile development”, “agile testing”,

The search was restricted to online accessible papers within access rights of University of Twente library. This omits books and practitioners reports. This restriction was applied due to the exploratory character central in this preliminary literature study.

For speed and scale manageability reasoning, only one journal search engine was applied, namely Web Of Science[3] (WoS). This search engine holds most of the Computer Science (CS) top journals. (Schwartz and Russo 2004) This argument combined with advanced search options and ease of use, this engine was selected for sole usage. Other candidates where Scopus[4] and IngentaConnect[5] that also reach a large portion of top CS journals. (Schwartz and Russo 2004)

The search was limited to papers no older than 10 years of age and originating from resources categorized by WoS as ‘computer science’ and as being a ‘review paper’. Herein lay two assumptions: (1) no highly valued papers originate from outside of CS field, which is reasonable because software testing is a specialist subject within software development. (2) Review papers provide a fast yet well grounded starting point to streams of research on the subject.

This resulted in a few papers that were too abstract to be of proper use. So a decision was made to drop the limitation of result type and thus accept regular papers as results as well. The new search resulted just under 300 papers. A selection mechanism was then needed, for analyzing 300 papers is too much for a preliminary study. For selection, the results were sorted on amount of references to guarantee stateliness by the academic field, where after a manual check for subject compatibility was performed by scanning paper abstracts. This resulted in a manageable set of papers that provided a quick insight into the aforementioned subjects software testing and agile development.

[3] Accessible at www.isiknowledge.com
[4] Accessible at www.scopus.org
[5] Accessible at www.ingentaconnect.com

A selection of insights gained:

- Agile development is a new software engineering paradigm with a new set of principles aiming for customer collaboration, responding to continuous change, and valuing individuals. These principles underlie every method of agile development and is agreed upon and opened to the world in a Agile Manifesto by respective method authors. (Fowler and Highsmith 2001)
- Agile testing focuses on early and frequent testing, with high amounts of tests being developed upfront or in parallel with coding efforts. This contradicts traditional development methodologies that perform tests as a separate phase at the end of development, often leading to integration issues.
- Agile testing requires high levels of automation and leans on regression testing.
- Agile testing is a relatively new practice[6]. Empirical evidence is limited, although they are to the least mildly positive[7]. (Janzen and Saiedian 2005)

2.3 Semi-open interviews

The second used research method semi-open interviewing. Goal was to get qualitative information about the current testing process arrangements and its performance. Another goal was to retrieve input on the format for the following survey. The preceding preliminary literature study provided the researcher with a means to talk in common testing concepts and jargon.

The interviews were held over all four (at the current time) business units, interviewing four people per business unit consisting of two project managers and two developers. This totaled 16 interviewees. The interviewees were handpicked by the two SME counselors supervising this research. The targets were selected as being eminent (highly valued by colleagues and well-formed opinions) and knowledgeable about SME's development (and testing) methodology. This aids in the shaping of a realistic reflection of current test process workings, required in this research.

Interviews were spread evenly over business units and personnel functions to have a diverse sample of BPT personnel, which improves research validity. The business unit division also served to see whether or not results can be generalized over the entire SME, while the split in personnel functions served to have input from both sides of development.

To improve answer value of interviewees a semi-open question format was crafted. The semi-open nature inspires interviewees to speak freely and explain by example. Exactly what is needed as qualitative information about the current testing process. The used format is listed as Appendix A.

Individual interviews won't be published in this report, for results were used primarily as anecdotal input for creating survey statement questions and to formulate expectancies towards survey question results. They also served to answer some *how* and *why* questions, but this doesn't provide a reason to publish individually either. Where needed in this report references are made towards interview results. Second reason for not publishing separate interviews is because of agreed confidentiality. A way to mitigate this confidentiality – publishing restriction is to anonymize data. This however isn't an option, for even anonymized interviews are easily traced back to individuals within the small amount of handpicked interviewees.

[6] Agile methodologies originate in 1995 or later. (For additional information see Figure 2-4)
[7] Janzen and Saiedian (2005) show results of empirical studies for both industry and academic settings on agile practices to have mildly positive outcomes.

What is published as direct interview results however is comparison Table 4-2 in section 4.2.2, which compares interview to survey results. More results are written in the form of statements and their respective expectancies (see: section 4.2.3.7) that mostly originate from interview responses.

Final comment on interview yield is the byproduct it delivered. During several interviews references were made to test reports. The researcher was told that these reports form a proper reflection of test process setup. This was deemed as enough reason to apprehend a set of these reports for analysis. Results of this short analysis are depicted in the following section.

2.3.1 Interviews side-product: sample test reports

During several interviews references were made towards test reports. Every development project at SME reports to their customer what was tested and in what matter in periodical test reports. Samples were asked and retrieved. A shortlist of four regarded as 'outstanding' documents were analyzed for test type application and general test process description. Conclusions aren't covered here; references to results are made at section 3.3. The sample test reports aren't included in this report, for they are available only in Dutch. Translation would be a time-consuming activity of little value.

2.4 Survey

The held survey served as quantitative backup of interview conclusions and premises. Its format also included some questions raised in the preliminary study. Questions target use of the variety in test types and agile principles as identified during the preliminary study.

Four underlying sections cover methodological attention points the researcher took into account to arrive at a proper survey and guided result analysis. Section 2.4.1 covers the survey format realization cycles, showing the transition from a first draft to a final full-scale online survey. Section 2.4.2 provides insights in applied questioning and scales, which are Likert scales were applicable and ordered-categorical or uniformly distributed elsewhere. Section 2.4.3 shows how survey results were analyzed for tendency. Section 2.4.4 concludes the description of the survey research methodology by listing the extensive steps that were taken to ensure high response rates and increase sample validity, as well as the defense for selecting a data subset.

2.4.1 Format realization cycles

Two review cycles passed to retain the final (online) survey format. These cycles consisted of the researcher drafting a survey format (first draft originating from previous research method results), where after a handpicked expert panel of developers and analysts commented upon its 'fit for use'. This fitness being: questions are clear and comprehensible, answer scales are easy to perceive and logical, there are comments and descriptions where needed, and the question order follows naturally.

After these review cycles passed, the final format was drafted. It's available in this report as Appendix B. Visible there is the enclosure of definitions where use of agile concepts where asked. This guarantees that respondents answer exactly to what is asked, instead of to their perception of the asked concepts. Another method of improving response quality was the enclosure of a comment box at every question, where respondents were free to respond anything they wished to add to their answering of the questions. Looking back upon comment box use, respondents used this mostly to enrich their answers with qualitative information. In a few cases however it was used to state that one couldn't answer a certain question, thus improving response quality as argued beforehand.

A notable miss in the applied format was the absence of multiple questions targeting the same construct. At this research single questions for constructs were deemed to hold enough validity, considering results were only to be analyzed as exploratory data. The expert reviews are thus regarded as sufficient to guarantee construct validity.

2.4.2 Questioning and scales

As visible in the final survey format included as Appendix B, questions are uniformly formatted (examples: 'To what extent…', 'How important…') and dictated in open form. This helps respondents to answer both easily and unbiased. Questions were asked in a categorical order that was fixed for every respondent.

Because questions are mostly described in a statement form and the respondent is asked to evaluate agreement level, most answer scales are five-point Likert scales. There were three question types that deviate from this scaling scheme: (1) principle or test type application interval questions, (2) automation level questions, and (3) numeric grading questions.

For the first type an increasing time-scale interval was used (see: questions 5 and 7) using common and fixed time indicators (like minutes) supplemented by one non-fixed category: release. This was included because during the semi-open interviews a lot of activities were mentioned to be performed 'per release', but with varying time intervals for a 'release' between respondents. To counter this variation a survey question was included to determine the exact duration of a 'release'. Below the results hereof are discussed:

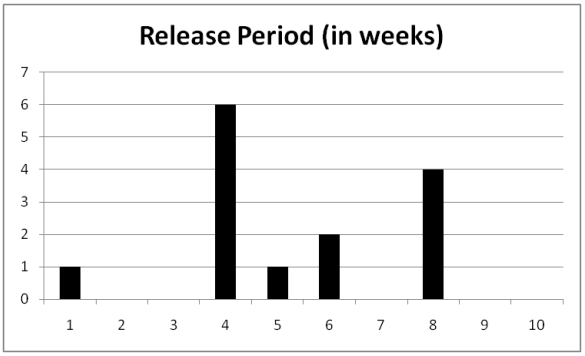


Figure 2-2 Release period

When observing the release periods mentioned by respondents, periods of 4 and 8 weeks dominate and thus are accepted as the two possible variations of release periods. Several observations prove that it's common to have release cycles of 4 (most often) or 8 weeks (less often), superseded by a primary release that takes a somewhat larger period to complete. This forms an explanation for the 5 and 6 months sometimes returned. Respondents were asked for the average release period, taking the longer primary period into account could come down to 5 or 6 weeks average. Thus 4 and 8 weeks will be used as release periods. This has implications for results on further questions where answers could be ranked (amongst other answers) as 'per month' or 'per release'. When using 4 weeks as a release period, month and release would be equal. For 8 weeks this would imply a doubled cycle. This split will be dealt with at the concerning survey questions in greater detail.

The second deviation is applied only at question 9. Here the automation level is asked, which is expressed in five fixed intervals of 20%. Using this fixed interval percentage scaling provides more insights than the other available option: applying an ordinal and increasing scale, ranging from 'no' to 'full' automation via 'little' and 'much'.

Third and final deviation is found at two places, questions 3, 4 and 16. Here numeric values are asked in the form of numbers. This scale holds the most value for it can provide detailed distribution figures for analysis.

2.4.3 Result analysis method and classification

To draw conclusions out of result data, a method of analysis is required. For this survey this was found in tendency analysis. This is possible for the question scales are in ordered-categorical form – a minimum demand for tendency analysis – or even better in the form of exact numbers resulting at some questions.

Deeper analysis through the use of statistics is regarded beyond consideration by the researcher, because of the limited number of respondents undermining usefulness and the required additional computations that isn't a necessity in this research of exploratory nature.

For the analysis of statements results, the tendency analysis wasn't satisfactory. A further assessment of data was required. This called for more advanced result classification, which steps are described in detail next:

All statement results follow a 5-point Likert scale, ranging from Completely Disagree to Completely Agree. First this scale was reduced to a three category scale to analyze result tendency:

- Low – Sum of response frequencies at Completely Disagree + Disagree
- Neutral – Response frequency at Neutral
- High – Sum of response frequencies at Completely Agree + Agree

A rule set for tendency conclusions was applied next. The threshold value of 4 (29% out of a total 14) respondents is used throughout as the lower limit for significant response frequency in a category. The final scheme resulting in five possible results classes is as follows:

Classification / **Frequency** / **Min-Max responses**	*Disagree*	*Disagree - Neutral*	*Neutral*	*Neutral - Agree*	*Agree*	*Spread*[8]
Low	9-14	4-8	0-3	0-3	0-3 *(0)*	4-7
Neutral	0-3	4-8	9-14	4-8	0-3 *(1)*	0-6
High	0-3	0-3	0-3	4-8	9-14 *(13)*	4-7

Table 2-1 Statement category classification scheme

[8] A classification where responses are either homogenously distributed over the responses range, or amass equally at the Disagree and Agree ends of the response range with little neutral responses.

As an example one of the statements' results is provided:

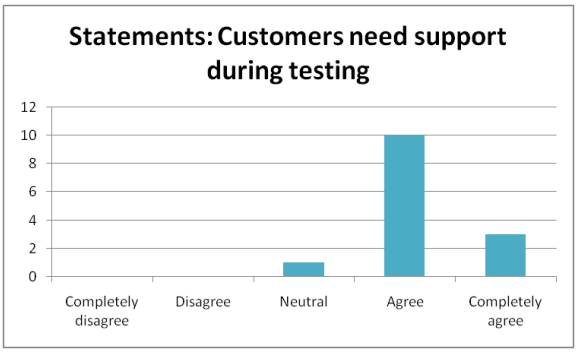


Figure 2-3 Example for classification: tendency analysis

This example scores 0 (0+0) on Low, for it has no responses on both the 'Disagree' options. It scores 1 on Neutral for 1 respondent answered 'neutral'. High scores 13 (10+3) as 'Agree' and 'Completely agree' are mentioned often. When looking at the classification scheme, this 0-1-13 score corresponds with the classification 'Agree' as it falls within the corresponding min-max responses corresponding with the 'Agree' classification. The example is also marked *Italic* in Table 2-1 within the 'Agree' column.

Results of applying the classification scheme on statements are listed as tables in section 4.2.3.7.

2.4.4 Sampling, response rate and sample validity

After the format – with underlying scales and result classifications – was agreed on, the actual survey could be distributed. Thus email invitations for the now online survey were sent. These invitations were sent to the entire population of SME employees, and concluded with a remark that the survey was only intended for developers and project managers. (Response reliability: only a handful of employees at SME have non-developing jobs so there is low risk of non-developing personnel corrupting survey responses; also at question 2 of the survey once more the survey targets were stressed reducing false data risk as well) Next to the primary invitations, two more reminders were sent by the researcher, and a final participation request was sent by a renowned SME employee.

Along with the full population sampling, easy online survey access and repeated reminders described in the previous paragraph, anonymity was guaranteed to improve the survey response rate, as well as session storage to enable respondents to complete the survey over multiple timeslots.

34 responses were received after the expiration date for survey completion. Of these 8 were incomplete, of which 7 had responses showing corrupt data such as all questions scoring neutral or min/max values throughout. These thus were omitted. The remaining single response held a comment that the respondent in question worked less than one month at SME. This response was also omitted for having a probable bias due to inexperience with SME development processes. Thus in total 26 responses were included for analysis. Total response rate is 37% (26 respondents / 70 developing personnel).

During result analysis, the variance in results was found to be too high to draw conclusions. This implied taking measures to reduce this variance. There were three respondent identifiers included in the survey: respondent's business unit, respondent's function, and respondent's number of years employed at SME. After analyzing results using these identifiers for result classification, business unit was found accounting for the high variance. To the researcher this didn't come as a surprise, as earlier held interviews also showed variance in results between business units. Impacts of the other two identifiers weren't further analyzed, for they are regarded out of scope for this research, because it aims to provide an holistic overview of test processes and not to dig into little varying results amongst team members that happen to have different functions or employment durations.

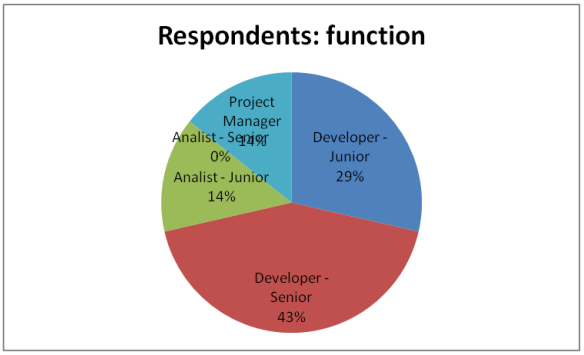
After the identifier causing the high variance was identified, it was time to decide upon results that should or shouldn't be taken into account. At that time four business units existed at SME: Hub, Energy, Automobile (Government) and Transport (Care). One way to pursue would be to separate results for every business unit. But this would cause the result analysis to increase to four times the original effort. So commonalities in results were sought. Energy and Hub (E&H) were found to have similar results and could thus be regarded as one result subset, while still servicing two business units with result analysis. The decision was made to pursue this subset and discard the remaining two business units from further analysis to prevent result analysis from taking three times the effort as forecasted. The following paragraph goes into detail on the validity of the E&H subset.

Results originating from Energy and Hub respondents are similar; this holds enough premises for analysis validity. But a second opinion on the premises was asked. A handful of analysts and developers of these two business units were consulted for their expert opinion. Their opinion was that the subset E&H is valid for use. That's because both business units have a high degree of likeliness in development methodic. This is due to several reasons:

- They serve similar customers: both service mortgage lenders and insurers and build similar applications.
- They apply alike market mechanisms: both engineer applications to order. At Automobile and Transport (A&T) on the other hand, focus shifts towards Assemble-to-order (adapting existing software to customer needs) or even towards Make-to-Stock (application is build for future customers).
- They have an indifferent development environment: they both work with Microsoft's .NET, while at A&T Java is used. Both development camps have their own distinctive tools to aid development. Along with that, Energy has only recently split off from Hub (January 2007), which gives rise to large knowledge and work process overlaps.
- Manpower is often exchanged: developers and analysts are frequently traded for use in each A&T's projects. This trades knowledge and development routines intensively.

After discarding results beyond the selected subset – which was defended in the previous two paragraphs – 14 respondents remained for inclusion in this report's analysis. These 14 account for 56% of total developing personnel within these businesses (total developing personnel at Energy = 13 and at Hub = 12), which is a high enough rate for a survey's data to be reliable.

<<REMOVED DUE TO CONFIDENTIALITY REASONS>>



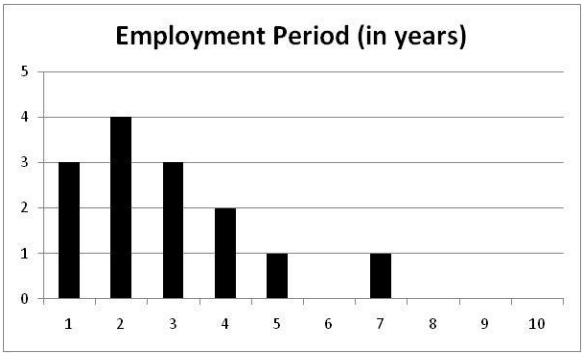


Table 2-2 Survey Results: Population

Table 2-2 shows an almost even spread in respondents originating from Hub as opposed to Energy, which proves an almost equal response rate for the two, that's because businesses hold almost equal amounts of development personnel and thus are likely to show equal respondent numbers. The next respondent identifier is job function. Almost $3/4^{th}$ part of respondents is developer while $1/4^{th}$ had a more shaping role as analyst or project manager. Senior Analyst scores 0%, which likely has to do with respondents regarding themselves Project Managers over Analysts. These findings correspond with enrollment data. This is also the fact with returned employment periods. These three factors let the research conclude that the combined respondents form a valid sample for analysis.

Analysis on other possible subsets of the sample – like showing results sorted on function or showing differences between older and younger personnel scores – weren't pursued. This would supersede this research' exploratory nature.

2.5 Structured literature study

Goal of the performed structured literature study (see: section 2.5.1) was to identify agile best-practices that would benefit SME's testing process. In theory this resolves to identify and review available agile development methodologies, and scan them for agile testing practices fit for use at SME. After this scan is complete a multi-criteria decision structure would decide upon the one(s) to use at SME testing. But in practice things turned out differently; out of the ten available agile development methods, just one agile testing practice could be identified that was fit for use. The selection of this practice is justified in section 2.5.2.3 and the research path leading thereto in sections 2.5.2.1 – 2.5.2.3.

But the single selected agile best practice on its own doesn't solve all issues of testing at SME, for it lacks in delivering project management aid. Because the aim of this research is to provide SME with a solid holistic approach to testing, supplement practices that can provide the needed guidance were needed. Thus a supplementary literature study was required. Two additional best-practices – Metrics and Continuous Integrated Testing – were selected to deliver renewed steering and control. Further details on research steps towards selection of these additional best-practices are described in section 2.5.3.

2.5.1 Agile methodologies study

For a literature study to be representative to a topic a systematic search needs to be performed at first. This section describes the search and evaluation methodology followed to identify articles for use in this study.

To arrive at a proper literature study a primary principle is to use quality sources. To satisfy PhD-level sourcing a search through the top 25 journals on Information Systems (IS) is required.

To search the academic field search engines *Web of Science*, *Scopus* and *Ingenta* can be used in conjunction to (almost) satisfy the PhD-demand. Except for *Communications of the AIS* the combined uses of these three indexers will result in a full coverage. (Schwartz and Russo 2004) But the same reference lists top 50 IS journal so why not try and cover them all? This should results in an even lower level of false-negatives. When cross-examining the coverage of the three search engines, the combination covered 44 journals. So six journals were lacking: *Communications of the AIS, Journal of the AIS, Journal of Information Systems, Electronic Markets, Journal of CIS, Australasian Journal of IS* and *Scandinavian Journal of IS.* These were successfully looked up and searched through on an individual basis.

In the 1st tier of the research combinations of the following keywords were used:

Software	Test	Information System	Application

These keywords were lead by added 'agile' and synonyms thereof to ascertain a genuine search in the direction of agile methodology: 'light' / 'short-cycle time' / 'internet time' / 'web time' / 'rapid' / 'test driven'

These keywords were followed by an added synonym of 'methodology', including 'method' / 'tool' / 'development' / 'framework' (For example the search phrasing 'agile software methodology' would return as one of the search terms.)

The effect of this combination was using 6*4*5 = 120 search terms.

This initial search provided 373 results. After combining the results of the various search engines 211 unique articles remained of which 49 articles were selected by referencing article title (rough) and abstract (thorough) to possible agile testing importance. The high percentage of false-positives was (also expected) due to the 'hot' and thus commonly used keywords 'internet', 'web' and 'rapid'. Using full text analysis 40 articles were awarded valid for use. Inclusion criteria that needed to be met were: (1) the focus of article lies on methodology and (2) the describes methodology must follow agile principles.

At the same time this resulted in a new set of keywords and synonyms to be used in the next search tier.

In the 2nd tier of the research the names of the individual agile methodologies were used as keywords.

Analog to the first tier there were 76 results, with 54 unique articles. After cross-examining these with earlier results 27 new papers remained of which 12 were selected as being fit for use.

Using forward and backward searching out of valid results 8 extra articles were added.

The 3rd tier concentrated on metrics by combining tier 1 keywords with keywords 'metric', 'measurement', and 'indicator'. And on CIT by searching on 'Continuous Integration', 'Continuous Integrated Testing' and their abbreviations CI and CIT.

2.5.2 Results agile methodologies study: the path to TDD

N.B. This section corresponds with the former agile research questioning, a literature study was conducted to identify agile development methodologies that are widely known and applied and could have some implications for testing. In the renewed research questioning XP's iterative testing that is identified below, is awarded a best-practice.

A small overview of the nine available agile methods is given. Their overlap was found to be limited to the approach of iterative testing. Of all nine methods only one method (XP) was found to have a detailed and concrete way to perform testing, where the others remain at a high abstraction level limiting application of their principles and at the same time lacking empirical backup for their axioms.

2.5.2.1 Nine agile methods

Over the last few years agile software development methods gained momentum in both business application as academic information science coverage. (Abrahamsson, Salo et al. 2002; Lindvall, Basili et al. 2002; Lindstrom and Jeffries 2004; Nerur, Mahapatra et al. 2005; Schwaber and Fichera 2005). But while their popularity is on the rise, only one article could be identified that includes a structured overview of currently available agile methods during the performed systemic literature study. The authors thereof provide a detailed evolutionary overview of (agile) software development methods (See: Figure 2-4) leading to what they define as the nine agile methods of today. (Abrahamsson, Warsta et al. 2003)

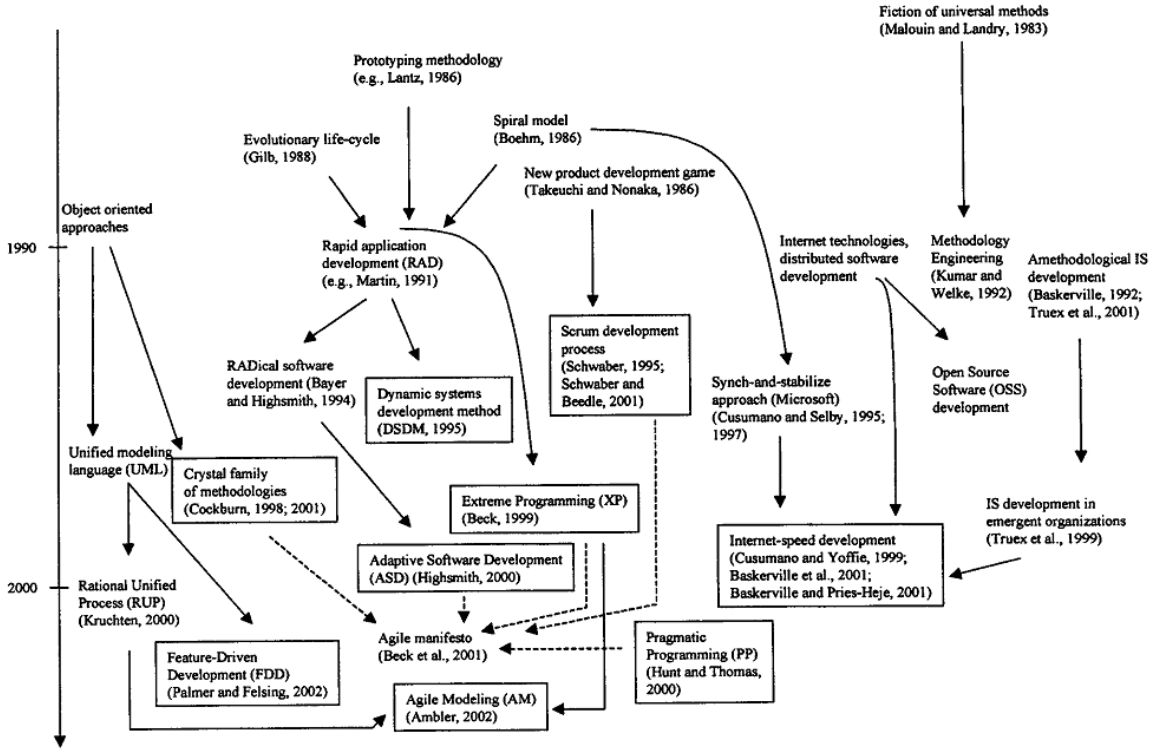


Figure 2-4 Evolutionary map of agile methods (Abrahamsson, Warsta et al. 2003)

These are[9]:

Agile development methodology	Author(s)
Adaptive software Development (ASD)	(Highsmith 2000)
Agile Modeling (AM)	(Ambler 2002)
Crystal family	(Cockburn 1998, 2000, 2002)
Dynamic systems development method	(DSDMConsortium 1997; Stapleton 1997)
Extreme Programming (XP)	(Beck 1999a, b, 2000)
Feature-Driven Development (FDD)	(Coad, J. et al. 1999; Palmer and Felsing 2002)
Internet-Speed Development	(Cusumano and Yoffie 1999; Baskerville, Levine et al. 2001; Baskerville and Pries-Heje 2001)
Pragmatic Programming (PP)	(Hunt and Thomas 2000)
Scrum	(Schwaber 1995; Schwaber and Beedle 2002)

Table 2-3 Agile Methods

[9] For detailed descriptions of the methods please refer to the original author's papers.

But what exactly makes them agile? All Agile methods aim for simplicity and speed. Also all of them follow the four values and 12 principles of the Agile Manifesto[10]. (Fowler and Highsmith 2001)

In more detail agile methods are characterized as having the following attributes: (Abrahamsson, Warsta et al. 2003)

- Incremental – using small software releases with rapid development cycles
- Cooperative – close customer and developer interaction
- Straightforward – method is easy to learn, modify and is sufficiently documented
- Adaptive – the ability to make and react to last moment changes.

Lindvall, Basili et al. (2002) devised a highly similar characteristics list, consisting of: Iterative, Incremental, Self-Organizing and Emergent. Emergent can be easily translated to the combination of former Adaptive and Straightforward, while Self-Organizing is a certain addition to the agile characteristics, for all methods use less dictated control and let teams organize themselves.

Other authors have made several other characterizations for agile methods, but these won't be used here. For a full discussion see the elaborate analysis of Abrahamsson, Salo et al. (2002).

2.5.2.2 Iterative testing

When taking a step deeper into characteristics of agile methods towards commonalities for testing only one common ground could be discovered: iterative testing. Because short release iterations are common in all methods, a direct implication for testing is that it will be performed more regularly than when compared to traditional formal methods for software development. While testing in formal methods is still done as one of the late and separated activities in software development (behind requirements analysis, design and coding) it will be done continuously and earlier than before, because of small programming cycles.

2.5.2.3 Empirically only XP and SCRUM are covered

When examining possibilities in testing approach more closely, the agile methods lack detail. This is due to these methods aiming to enable a collaborative mindset within the development team and thus not go into details about how to implement this. (Abrahamsson, Salo et al. 2002) The same authors performed an elaborate analysis of current agile methods using several analysis lenses. The first lens *Systems Development Life Cycle* (see: Boehm (1988)) answers the question "Which stages of the software development life-cycle does the method cover". For testing the following four phases are identified. These are (in chronological order) Unit test, Integration Test, System Test and Acceptance Test. The second lens *Abstract principles vs. concrete guidance* tries to answer whether or not the method has concrete guidance available. XP is the only method that (partially) satisfies this demand looking at test phases. Unit, Integration and Acceptance tests are covered. This means concrete guidance for system tests is missed in all agile methods. When applying a third lens *Project Management*, sadly XP is found lacking throughout. Thus supplement practices to cover project management activities should be sought.

XP's workings and accompanying promising benefits for testing are shown in sections 5.1.1 and 5.1.2.

[10] For details please refer to www.agilemanifesto.org

N.B. Please note once more that system testing isn't handled in any of the agile methods. When combining this with time constraints of this Bachelor assignment – where there isn't enough time to perform another literature research in other than agile methods – this implies this Bachelor thesis won't (fully) cover system testing.

2.5.3 Metrics and CIT to guide project management

With the agile methodologies study delivering just one best practice – XP's TDD – it can be concluded that not all issues will be solved properly after implementation, although TDD certainly does help (see: section 6.3.1). The previous section showed that TDD falls short on project management guidance. Because this guidance is key to a proper testing approach, and because SME currently severely lacks project management for testing, additional best-practices to counter this shortfall are required. This implied performing further literature study.

This literature study provided two additional best practices, Metrics and Continuous Integrated Testing (CIT). Metrics was selected for its enforcement of objective steering and control through numbers. CIT was selected because it severely shortens the feedback loop on software quality through the use of an automated regression test suite immediately uncovering defects upon injection.

The additional literature study was performed analog to the agile methodologies study. The same search engines and journals were consulted. Keywords did differ, used search keywords were combinations of prefixes 'project/defect/software/quality' joined by suffixes 'measurement/ management/metrics/evaluation/steering/control'. This search was further supplemented by rewarding extra attention to Software Process Improvement (or: SPI) frameworks[11]. These were searched for at the search engines and in a supplementary web search. After two search iterations 63 books, articles and/or reports remained fit for use.

The applied selection criteria were: the best practice must (1) ensure testing project management, (2) should contribute as much as possible in solving SME's testing issues, and (3) should fit within the agile development setup at SME.

During analysis of search results, it quickly became evident that every SPI framework relies heavily on the use of metrics. They list metrics as a key ingredient for project management's steering and control. (Fowler and Rifkin 1990; McFeeley 1996; Rainer and Hall 2002; van Solingen 2004) Metrics help to ensure the development process is under control. (Kan, Basili et al. 1994) At the same time steers people to change their behavior — they agree on a target and work toward it. As progress becomes reflected in the measured results, people make modifications to improve the outcome. (Fowler and Rifkin 1990) This holds enough value for SME to gratify selection. The actual metric instantiations and selection methodology thereof is listed in section 5.3.

[11] Reports on how to setup process improvement projects to improve development. These frameworks are mostly published by renowned Universities and provide valuable insights on available best-practices. These make excellent reference points on best-practices up for selection.

Second selected best-practice is CIT. CIT was selected because it minimizes the project management steer and control loop by continuously verifying whether the extensive automated test suite still passes. The researcher expects CIT to perform as a catalyst to guarantee (needed) continuous attention for testing.

CIT and its benefits are explained in section 5.2.

N.B. Stringent selection criteria to arrive at the two selected practices weren't applied because of the already boomingly rising size of this bachelor's thesis research due to this additional required literature study. The researcher chose to spend more time in indexation and analysis of best-practices than to spend it in devising a multi criteria decision analysis tree, normally required for this type of selection. A pragmatic selection was used on basis of researcher's good judgment.

3 CURRENT SITUATION

The current situation at SME doesn't suit testing needs, nor does it stimulate proper testing. This has various reasons:

- Responsibilities are too informal and lacking enforcement for test execution
- Developers hold inadequate knowledge about testing possibilities and methods
- The development priority for testing versus other development activities is too low
- Some test resources aren't available, impeding proper testing
- Quality isn't objectively measured which causes unwanted subjectivism in defect management
- Customers aren't adequately steered/helped towards useful functional and acceptance testing

Sections 3.1 and 3.1.1 – 3.1.6 will argue for the above. Section 3.2 forwards to types of testing currently in use; these will be dealt with further on at section 4.2.2. Section 3.3 states the absence of a common test methodology, despite from limited customer-appointed *TMAP Next*[12] usage. In section 3.4 probable reasons behind this testing absence are mentioned: the evolutionary or chaotic way of work process development combined with deadline and monetary pressure limit software process improvement projects (of which testing is one needed project).

[12] A structured software testing methodology from Sogeti, commonly applied in the Dutch financial industry. For details please refer to http://www.tmap.net.

N.B. From this point forward, cross-references to survey statement results will be depicted as (<question number> - <result>). For instance (8 – Agree) corresponds to statement #8 – testing occurs ad-hoc – scoring mostly Agree. The numbered statements are listed in Table 3-1 below. Please refer to section 4.2.3.7 for the elaborate statements list including expectancy and deviations as well as the result classification scheme.

#	Statement	Result
1	*Customers should write own test plans*	Neutral - Agree
2	*Customers test properly*	Disagree - Neutral
3	*Customers need support during testing*	Agree
4	*Customers should test at SME*	Disagree - Neutral
5	*Customers are available for context questions*	Neutral - Agree
6	*Customers are involved in the testing process*	Agree
7	*The current development planning guarantees enough testing*	Disagree
8	*Testing occurs ad-hoc*	Agree
9	*Testing is skipped/ severely shortened upon endangered development deadlines*	Agree
10	*Test execution is stimulated by project managers*	Neutral - Agree
11	*Feedback on own code via bugs arrives soon enough*	Spread
12	*Testing receives enough attention*	Disagree
13	*I am certain of effects of checked-in code on application*	Neutral - Agree
14	*Amount and quality of testing is highly dependent on developer personality*	Agree
15	*Attention for testing degrades as development progresses*	Spread
16	*Influence of code changes on total system behavior is underestimated*	Spread
17	*Testing is performed without a clear strategy*	Spread
18	*Code is written to be testable*	Disagree
19	*Effects on the rest of the application are unknown during refactoring*	Disagree - Neutral
20	*Current unit tests test to large chunks of code at once*	Disagree - Neutral
21	*Less time required for bug fixes by testing more nets less total development time*	Agree
22	*I feel confident about bug freeness of current delivered applications*	Disagree - Neutral
23	*When working with live code, tests are executed more often*	Spread
24	*Acceptance tests provide a proper point of departure to see how 'done' the application is*	Agree

Table 3-1 Statements and collapsed survey results

3.1 Overall picture: testing in trouble

Both the held interviews as well as the survey showed testing processes aren't structurally anchored in the development process. (8 – Agree on 'testing occurs ad-hoc', 7 – Disagree on 'Development planning guarantees enough testing' and 12 – Disagree on 'Testing receives enough attention). Testing is awarded a fixed percentage of time of the development budget which is theoretically executed near the end of the development process, like in the well known waterfall model. In practice however this testing budget and time is swapped for even more untested features, originating from new-feature request customer pressure near release (frequently called 'feature creep') or simply by not reaching development deadlines. Another factor severely preventing proper testing to take place is the developers lagging behind on writing testable code (18 – Disagree).

Testing is reactively steered by high levels of defects in part(s) of an application. The only observed exception to this latent attitude towards defects is the creation of test documents in conjunction with customers that list use cases that need at least to function upon release. This is used proactively for the creation of functional tests, however as development progresses these tests aren't updated and thus lack accordance to requirements change during development. These tests are manually handled just before a release. Other types of tests are hardly executed and automation levels are low. For a detailed description of test types and execution levels please refer to sections 4.2.1 - 4.2.3.

When discussing repetition of test cycles, or more advanced regression testing, this isn't applied at all. Probably this is due to lack of a central tests database and agreement on execution cycles.

Integration of separate developed application features to form a release is also a painful issue, because this isn't attempted earlier than applications being 80% complete, commonly resulting in integration defects. Likely cause to this delayed undertaking is the acceptance reasoning in which testing is entrenched. The very nature of (the current non-agile form of) acceptance testing from use cases requires applications to be near complete in order to create viable test cases.

Above test process observations, perceived problems and possible trouble causes are listed at a macro overview level. To further strengthen the analysis deeper cause and effect relations were sought in interviewing key developers and analysts. This was supplemented further by continuously observing a development team in action. After a couple interviews and moving further ahead in time a handful issues categories were visible. They are:

Issue category	Description
Responsibilities	In which way are responsibilities for testing divided through development teams?
Knowledge / Competence	Does the development team have enough expertise to properly perform testing tasks?
Priority	How does testing wage up against other development activities?
Resources	Are tangibles (hardware, software, methods, checklists, etc.) available?
Quality Management	What actions are taken by project managers to guarantee software quality?
Customer Guidance	Does the customer gain proper testing guidance if needed?

Table 3-2 Issue categories

The following sections 3.1.1 - 3.1.6 go into detail on the role of testing and quality maintenance during development. Each section features an individual issue category. Please note that some issues fit multiple categories, but for matter of oversight these are only appointed to the best fitting category, instead of mentioning them repeatedly.

3.1.1 Responsibilities: too informal

The responsibility for quality code rests with developers. It's up to themselves whether or not this requires testing, and to what amounts (14 – Agree). Project managers do not adequately steer on required testing levels or specific functionality that need to be thoroughly tested (10 – Agree).

Both these issues cause causes low levels of test execution and test script maintenance. The individualism (14 – Agree) originating from this free and uncontrolled test paradigm has lead to personal and private testing. It also leads to an apparent impossibility for project managers to address these problems like the unwanted high variation in test quality and impossibility to set minimum test quality and quantity demands. See also section 3.1.5.

3.1.2 Inadequate knowledge / competence

Developers' knowledge of various test types is inadequate. The survey frequently reports limited and absent levels, while minimum knowledge levels should naturally score at least adequate. In customers' test type knowledge the survey showed high spread, proving interview anecdotes of some customers being more test capable than others. But high spread sadly also covers a lot of limited and absent survey scores on test competence, which again shouldn't be observed. This is possibly due to lack of SME originating guidance on how to setup and execute proper test procedures. See section 4.2.3.5 for corresponding survey results.

3.1.3 Low priority

The most pressing concern on the role of testing at SME is a lack in priority. This is due to two reasons, firstly a great amount of researcher observations point to its existence and secondly the far-reaching consequences of shorting a core activity of the *Systems Development Life Cycle*[13], which are visible throughout this report.

The survey proved that testing in general doesn't receive enough attention (11 – Disagree). Furthermore there is neither test planning nor targets (7 – Disagree) and thus testing occurs ad-hoc (8 – Disagree). Next to this, over half of the survey respondents report (Completely) Agree when asked if testing is performed without clear strategy. (17 – Spread) Wrapping up, a small anomaly in the priority-lack presumption needs to be mentioned, for test execution is somewhat stimulated by project managers, (10 – Neutral – Agree) which does show testing isn't missed completely.

An example of testing receiving too little attention – at the cost of other activities – is a phenomenon called feature creep, where near release deadlines customers and project managers tend to include additional untested features over testing already implemented ones. According to the interviews, feature creep is a frequently observed phenomenon at SME development projects. Two survey results (partially) back this observation: testing tends to be skipped or severely shortened upon endangered deadlines (9 – Agree) and attention for testing degrades as development progresses (15 – Spread). While the customer gets some extra wanted features, this is a high stakes game to play for SME. This is because for now the customer is happy with the extras, but this takes a high toll on the future. Latent bugs in older features aren't discovered by testing and at the same time new bugs are likely to be implemented at a high rate, because of the high coding pressure and speed that's required in last-minute development work leaves great margin for error.

[13] SDLC adheres to important phases that are essential for application development. A traditional SDLC is composed of the phases Initiation, Requirements Analysis, Design, Build, Testing, Implementation and Operation and Maintenance. Every phase holds essential and unique development activities that no development can do without.

3.1.4 Partial lack of resources

In the survey the level of dedicated test resources was asked, to find that three out of five basic testing resources (mentioned in the interviews) are hardly available. There's a lack in test scenarios (Limited), templates (Absent – Limited) and common tools (mostly Limited). It's odd to see that relevance of these resources is awarded importance throughout, but missing availability. A reasoning as to why these resources aren't available probably rests within the aforementioned lack of priority. The fourth resource, a build server, is available (Adequate – Perfect), but during observations it became evident that real agile possibilities hereof aren't applied. These include the heavy use of automation and regression, which are hardly applied (though deemed important from survey outcomes; see: Table 4-6 and Table 4-9 of sections 4.2.3.2 and 4.2.3.4 with results showing tendency towards agreement) at SME. At the final resource – dedicated test hardware – great spread in survey response was observed, over half of the answers however were marked as Adequate.

For full results of test resource availability see section 4.2.3.6.

3.1.5 Quality management bears subjectivism

The interviews proved that software quality is currently steered subjectively by gut feel of project managers. This is supplemented only by reactive countermeasures taken when (parts of) the application doesn't function as supposed. The lack of objectivism (i.e.: quality indicators) adds to existing difficulties (see: section 3.1.1) in steering on quality. The lack of objectivism is likely to fuel the lack of development teams' confidence in bug-freeness of developed applications (22 – Disagree – Neutral).

3.1.6 Limited customers guidance

According to the survey, customers don't test well enough (2 – Disagree – Neutral), the interviews show that some customers know a lot about software development while others don't. This is furthermore backed by showing almost a normal distribution pattern on knowledge level of the customers at both functional and acceptance testing. Next to knowledge, customer involvement was also asked. This was – in contrary to developer interviewee anecdotes – found to be ok (5 – Agree and 6 – Neutral – Agree), during the interviews it became evident that knowledge and priority lack cause less than required amounts of testing. A way to counter (at least) the knowledge lack would be to assist customers in their testing. Sometimes this is done with customers via testing workshops and meetings, but still these activities do not amount to the regarded necessity for customer test assistance (3 – Agree). This limited assistance probably has to do with development teams' slight tendency to have customers write their own tests (1 – Agree).

3.2 Types of testing

For a complete overview of testing types see section 4.2.1.

3.3 No overall methodology, limited TMAP NEXT application

There is no common testing methodology in use at SME. Some recent projects do use Sogeti's *TMAP NEXT* methodology because of the financial customers' familiarity with this testing framework. But interviews state the 'following' of this methodology is limited to using the templates for functional testing only. This was confirmed by examining a sample of previously delivered TMAP NEXT formatted test reports (only in Dutch available thus unattached to this report), where only functional and user acceptation tests outcomes were mentioned and results thereof reported. Furthermore; observations supplement the finding that testing doesn't follow any methodology. Testing processes function in an ad hoc and ill-defined way.

3.4 Realization current test process

The interview questions aiming for reasoning behind the absence of testing didn't result in useful answers. Practically every interviewee has no answer as to why testing currently is a mere shell. An explanation few times mentioned is the natural reluctance developers hold against testing, preferring coding as building over testing as breaking code. One developer provided another answer deemed as a more likely explanation by other development team members as well as the author of this report: so far there hasn't been anyone within SME that has a clear vision with underlying expertise in testing and champions its application. Without such a champion pushing for change there won't be anybody that raises questions or undertakes actions on improving testing.

4 CURRENT PERFORMANCE

Current performance in software quality at SME was found impossible to measure, because it isn't measured during the development process. Unfortunately this couldn't be mitigated by setting up objective measurements, for they are too time-consuming to setup and collect within the context of this Bachelor's Thesis.

A more subjective and less detailed measurement scale was thus required. This was found in Kan's *Effort/Outcome Matrix*. (Kan 2003) This tool is applied in section 4.1 to find that SME resides in either 'Unsure' or 'Worst-Case' test functioning.

This matrix aids in assessing testing performance, for it supplements the usual – but here unavailable – *Outcome* (quality as defects found) by *Effort*. While *Effort* doesn't hold a perfect positive correlation with *Outcome*, it does hold some premises. That's because the use of tests correlates with finding defects more often as well as earlier. In other words: higher *Effort* means improved *Outcome*.

Effort was found to be measurable: it's quantified in this research by analyzing the use of test types and underlying principles. Section 4.2 shows that testing isn't applied in required intervals, and underlying test principles also mostly aren't in effect.

4.1 Testing performance: ordinal 'Unsure' at best

During the interviews it became evident that there are no structural testing performance indicators currently in use. The only indirect performance metric available is the amount of outstanding defects with accompanying defect backlog. This is used subjectively, when project management believes there are too many outstanding bugs the software quality is found to be lacking, while when there are few bugs discovered the quality is perceived as high enough. This is a risky way to attain software quality as Kan (2003) clarifies in his Effort / Outcome Matrix. To show some classification of testing performance, compensating for the lack of indicators, this matrix was used.

		Outcome (Defects Found)	
		High	Low
Effort (Testing Effectiveness)	High	Cell1 Good/Not Bad	Cell2 Best-Case
	Low	Cell3 Worst-Case	Cell4 Unsure

Figure 4-1 Effort/Outcome Matrix (Kan 2003) (adapted)

When observing the Matrix (see: Figure 4-1), four quadrants are identified. These quadrants are differentiated by two dichotomous indicators. The first is the amount of effort put into testing, the second the amount of discovered defects.

The *Effort* level is certain at 'Low' level. This is due to limited current attention to testing, expressed in lack of indicators (mentioned at the beginning of this section), limited testing over the six levels (see section 4.2.2) and unit as well as integration testing often coming to a halt because of high new feature pressure originating from the customer near the end of development pushing testing out of release planning. The held interviews and survey both clearly indicate this limited effort.

At SME the *Outcome* indicator is Low, because in general the amount of discovered defects in software releases is perceived as low by both project managers directly via low levels of bugs listed in the Mantis Bug Tracking Tool as by customers that state SME delivers great software (both notions originate from held interviews). Interviews often resulted in the statement: "we are lucky to have great developers", pointing out clearly that the few occurring defects have little to do with proper testing, but with few errors being made by well performing developers. Perhaps the learning curve of going through earlier likewise development projects accounts for this outcome. Extra care needs to be accounted to the internal perception of defects however, there is no baseline measurement to tell whether this ordinal scale should be valued High or Low. In the current situation, where limited attention is geared towards testing one could would probably only see a small part of total defects. (Indicating the likely high but unknown level of sub-merged defects for a limited part of the application is tested). The perception of customers judging defects as Low, can be awarded higher value however. It is highly likely that these customers have gone through similar software implementations with other application developers like SME before SME and so they do have baseline information to benchmark 'right' from 'wrong'.

The result from combining the two above indicators, places SME in either Cell4: 'Unsure' or in Cell3: 'Worst-Case', where the likeliness of the former is highest. That's because the opinions of customers (Low Outcome) are valued higher than project managers' (unknown Low or High Outcome) opinions. After all, the customer is the one who perceives defects upon usage. Either way, by putting more effort into Testing Effectiveness this can be raised to at least a certain Not Bad level, which should result in a raised perceived software quality (i.e. fewer defects). In chapter 5 possible improvements to achieve this upgrade are listed.

Please note that this section doesn't hold quantified results to be able to state with confidence how SME performs towards defect levels. This requires both an elaborate setup as strictly disciplined and lengthy data gathering which simply isn't possible to perform in the light of this research. This remains something for the future.

4.2 Low execution levels throughout test types

The following sections cover definitions of test types (section 4.1) as well as the troublesome test execution levels and underlying principle usage (see: section 4.2).

4.2.1 Test types

In order to properly discuss testing types (or their execution levels), first the seven main types of testing are discerned below. These definitions aren't agreed on[14] completely, and often they are confused with one another, so their exact meaning (at least in this research) is listed here. (Fewster and Graham 1999; Jones 2000; Sogeti 2008)

Test type	Definition
Unit	Tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class (containing attributes and methods) level, and the minimal unit tests include the constructors and destructors.
Integration	Exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.
System	Tests a completely integrated system to verify that it meets its requirements.
System integration	Verifies that a system is integrated to any external or third party systems defined in the system requirements.
Functional	Tests at any level (method, class, module, interface, or system) for proper functionality as defined in the specification.
Acceptance	Can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Table 4-1 Seven types of testing (Sogeti 2008) (Jones 2000) (Fewster and Graham 1999) (adapted)

[14] Definitions vary greatly as well as how many different test types should be discerned. It's a matter of focus the author believes. In agile frameworks the mentioned six test types are commonly discerned, although their definitions vary slighty amongst different practitioners and researchers. Definitions from the three mentioned papers were combined to provide the best fitting definitions.

4.2.2 Condensed test and principle usage

Below the usage, automation and regression quantity of the six types of testing at SME is shown, split in qualitative and quantitative results.

Principle	*Usage*		*Automation*		*Regression*	
Source / **Test type**	**Int+ Obs.**	**Survey**	**Int+Obs**	**Survey**	**Int+Obs**	**Survey**
Unit testing	Weekly – Monthly	Spread	None	Spread	None	Spread
Integration testing	Monthly	Per Release – (Almost) Never	None	0-20%	None	Per Release – (Almost) Never
System testing	Per Release / Only when threatened by SLA	Monthly - Per Release	None	0-40%	None	Per Release – (Almost) Never
System integration testing	Per Release	Per Release – (Almost) Never	None	0-20%	None	Per Release – (Almost) Never
Functional testing	Per Release	Per Release	Sporadic	0-40%	None	(Almost) Never
Acceptance testing	Per Release	Per Release	None	0-40%	None	Per Release – (Almost) Never

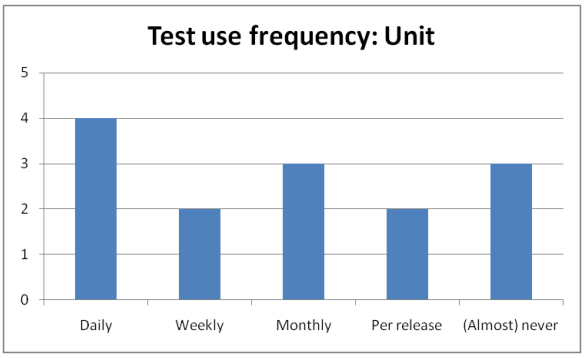
Table 4-2 Condensed Results Test Execution: Quantitative vs. Qualitative

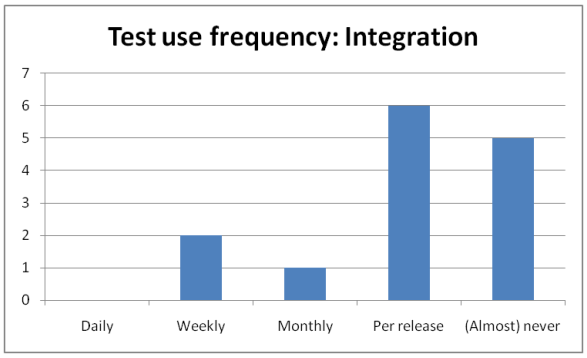
This section only handled condensed findings to provide a quick test use overview, and thus will not comment on results. For the full analysis please refer to all sections under 4.2.3.

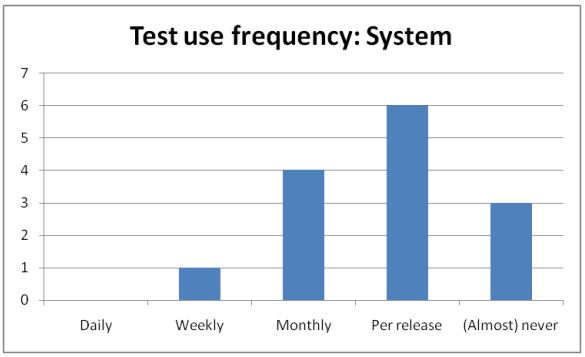
4.2.3 Complete test and principle usage

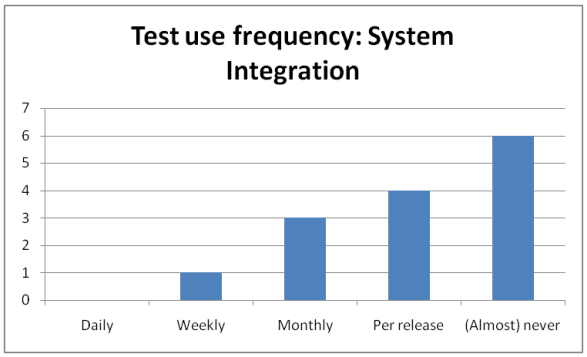
Sections 4.2.3.1 – 4.2.3.7 below cover results from the survey. Available statistics will be provided and supplemented by qualitative comments per survey category. Section 4.2.3.1 entails the limited use of tests at (almost) all test types, even when deemed highly relevant. Section 4.2.3.2 and 4.2.3.3 list absent regression and thus regression usage falling greatly behind on test frequencies. In section 4.2.3.4 automation is found to be hardly applied, possibly due to neutral relevance. The test knowledge level of developers is proven mostly limited in section 4.2.3.5, while customers' knowledge is spread. Various test resources are deemed important, but found unavailable at section 4.2.3.6. Lastly at section 4.2.3.7 various statements on development undertakings are commented.

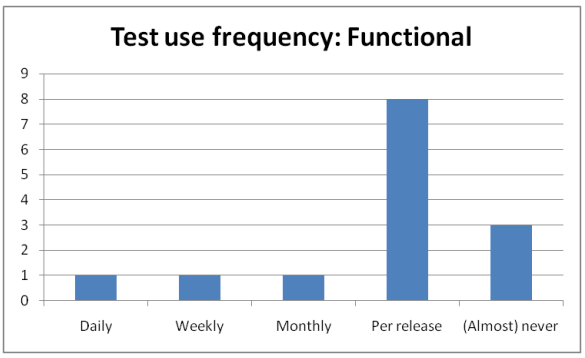
4.2.3.1 Test use far too limited for perceived relevance











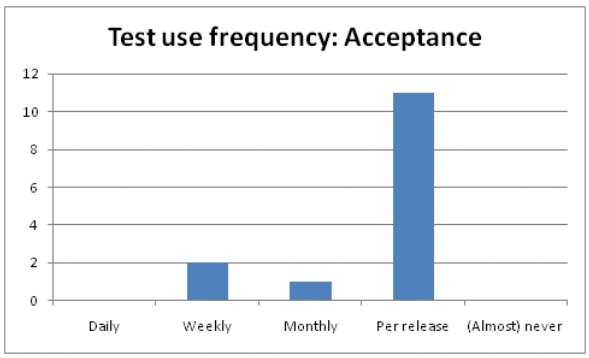


Table 4-3 Survey Results: Test Use Frequency

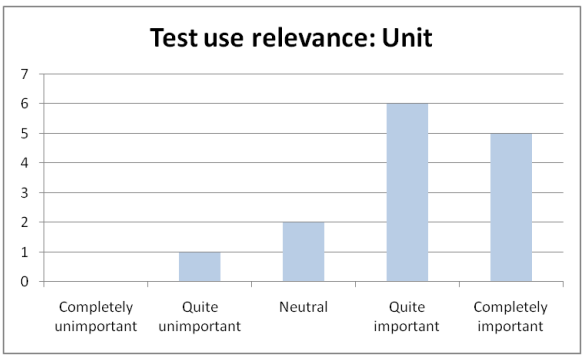
The table above lists survey results for test use frequencies. The figures show number of responses per category[15]. Clearly visible in all test types except for Unit Tests is that the respondents tend to choose for longer intervals of test occurrences, starting at monthly and ending at (Almost) never with per release being the modus. At Unit testing an unexpected distribution of answers is visible with all answers occurring almost evenly distributed. This wasn't expected for interview results show virtually no use of unit tests.

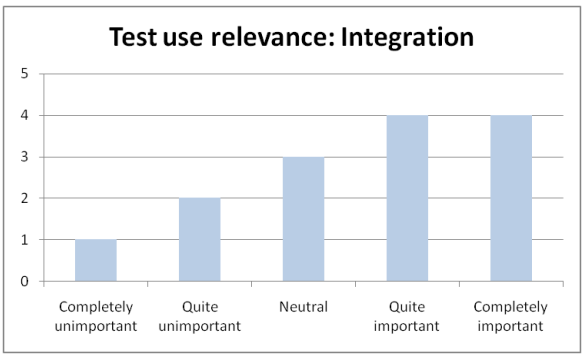
Conclusions:

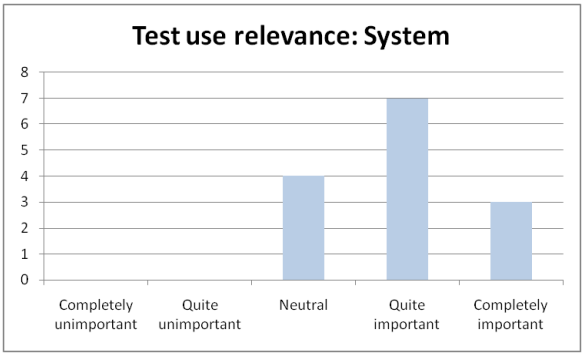
- Too often too large test intervals are observed.
- (Almost) never's shouldn't show in a viable testing setup, but they do occur.

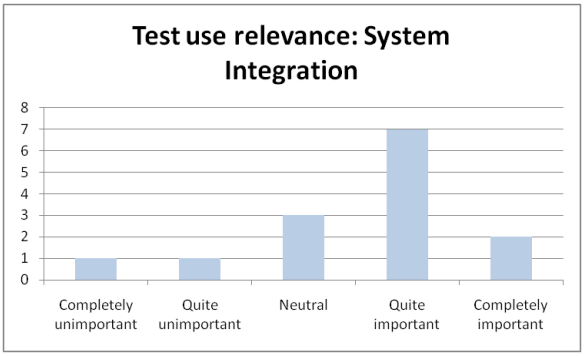
[15] All survey result figures report response frequency per answer category, but with varying scales. For an explanation on applied scales, please refer to Section 2.4.2

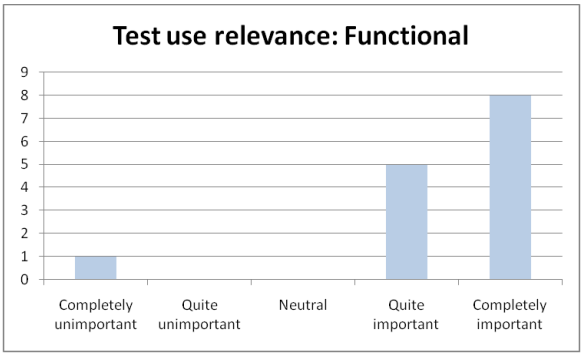
- Monthly Unit and Integration Testing frequency is below agile frequency range, testing these weekly might even be too few, for agile methodologies report sub-daily builds including automated test runs.
- System (+ Integration) Testing occurs monthly, which relates exactly to a 4 week per release system test or a per halve release system test when using 8 weeks as the release period.
- Functional Testing per release is too few, this implies once per release / per half release (depending on chosen release frequency 4/8 weeks) testing to see if required functions work. In agile development this is generally performed as soon as individual functions are completed, thus: well before the end of a release.
- Acceptance testing is performed per release, which is also few in agile testing/development, where customers are highly valued for continuous input.











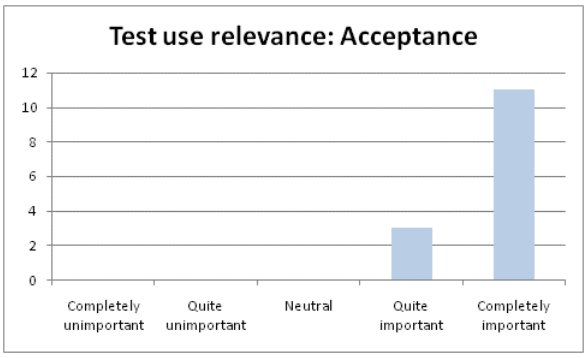


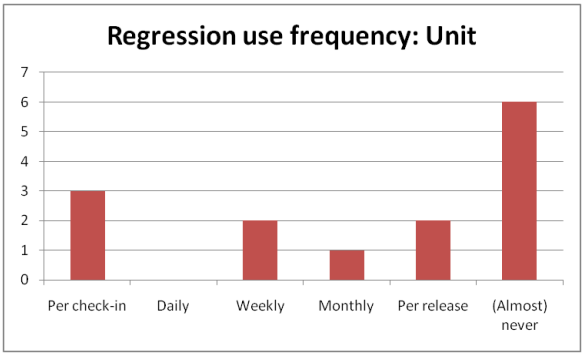
Table 4-4 Survey Results: Test Use Relevance

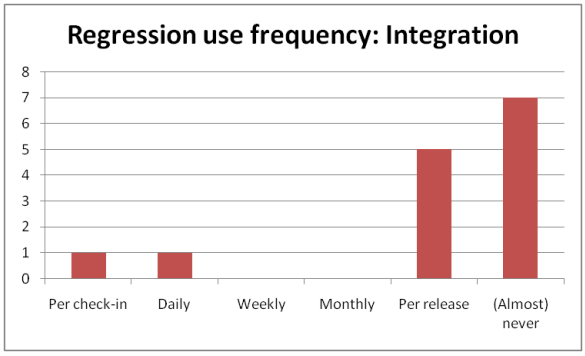
Table 4-4 shows to what importance the test types are valued. For Functional and Acceptation the results show (almost) everyone agrees they are of high importance. For Unit, System and System Integration tests there is more spread in the results, but a tendency can be observed towards Quite Important. Integration relevance isn't agreed on, but a tendency towards importance is present.

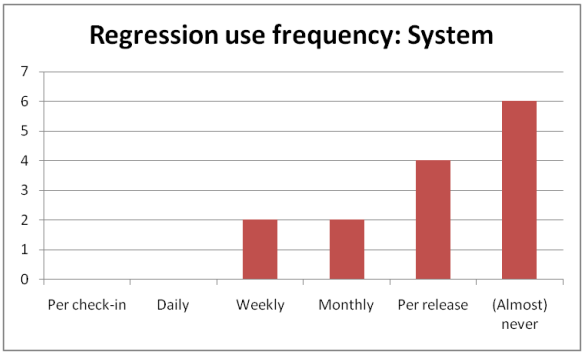
Conclusions:

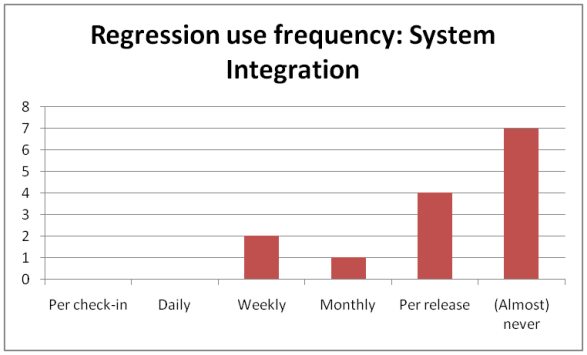
- All tests are regarded as important, but to varying levels.

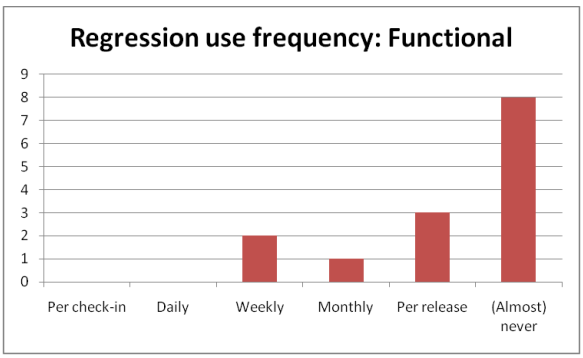
4.2.3.2 Regression? What regression?











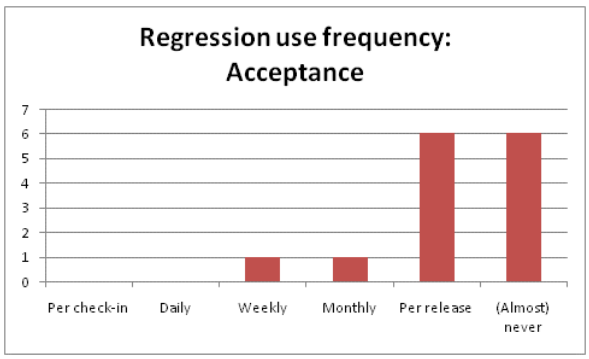
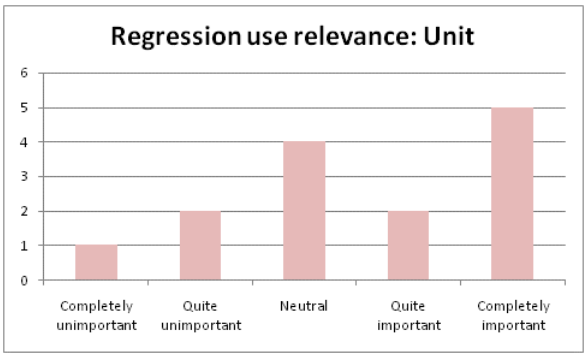


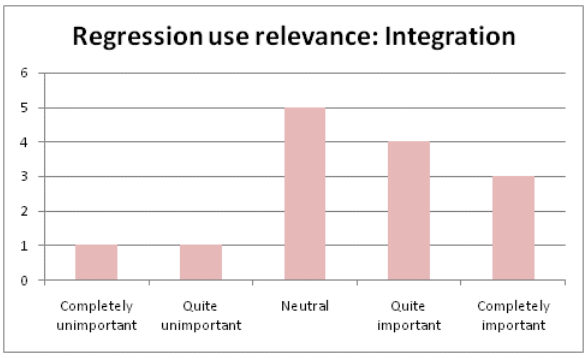
Table 4-5 Survey Results: Regression Use Frequency

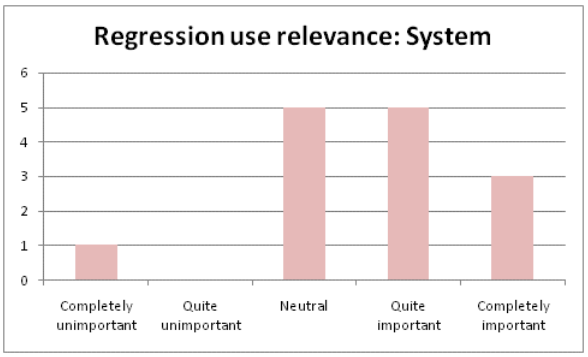
The above Table 4-5 shows the use of regression (performing tests in a repeated setting) at the various test types. A general negative tendency is clearly visible, with highest occurring response frequencies on (Almost) never. For Unit and Integration tests a wider spread can be observed.

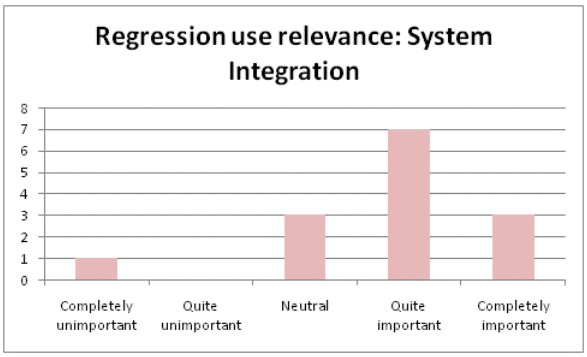
Conclusions:

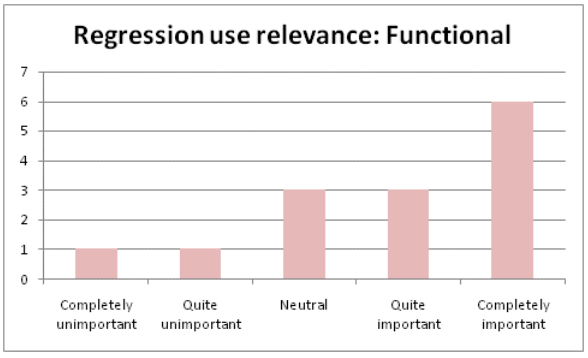
- (Almost) never shouldn't occur.
- Regression is at a maximal per release activity, this is far apart from proposed build integration cycles in various agile methodologies.











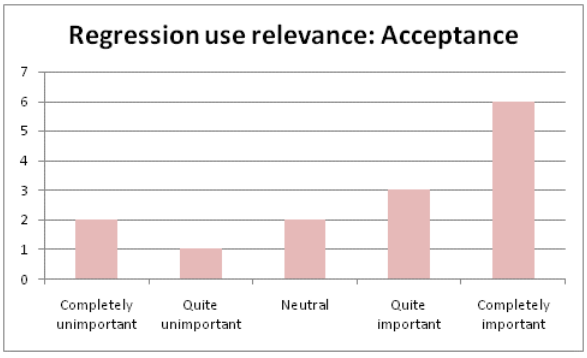


Table 4-6 Survey Results: Regression Use Relevance

Relevance levels for Integration, System, System Integration, Functional and Acceptance testing show a tendency towards importance of regression. At Unit testing however the tendency is visible, but less severe, for results include a lot of neutral responses and some unimportant rated ones.

Conclusions:

- Regression is deemed a relevant practice for all testing types.

Upon comparing actual regression use and its perceived relevance, a large gap between need and use is observed. Where actual use is (severely) limited to happening per release or (almost) never, perceived usefulness instead scores very high. This is a remarkable contrast, for the essence of regression is frequent occurrence, which misfits current testing practice.

4.2.3.3 Regression lacks behind test frequencies

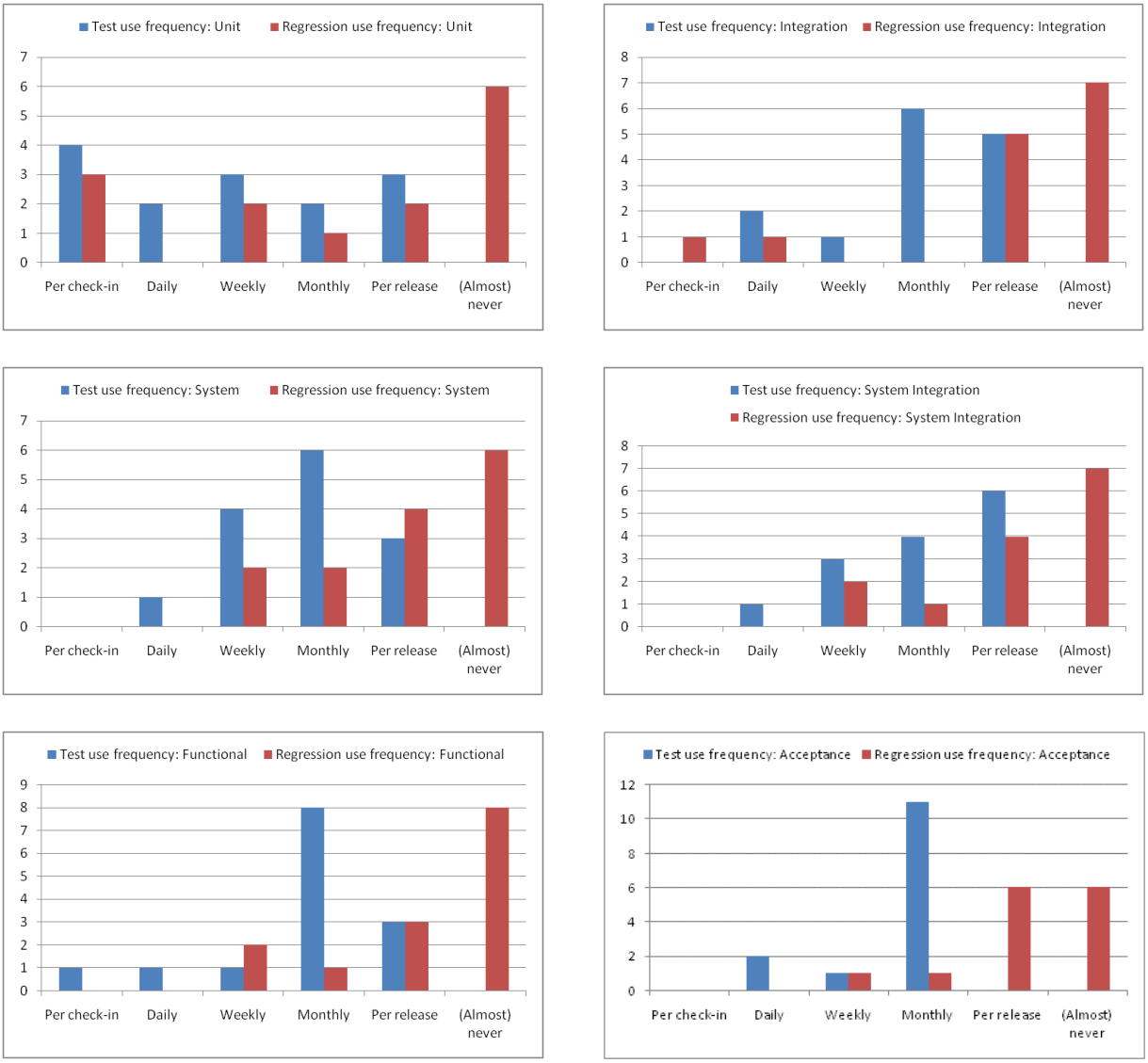


Table 4-7 Survey Results: Test Use Frequency vs. Regression Use Frequency

Table 4-7 shows usage next to regression of the different test types. It shows regression levels lagging behind on test execution levels (visible as red columns amassing further right than their blue counterparts). A counterargument for this could be that not all tests need to be included in regression suites, but the results of these two measurements lie too far apart to fully account for that variation.

4.2.3.4 Automation zero to none accompanied by relevance neutrality

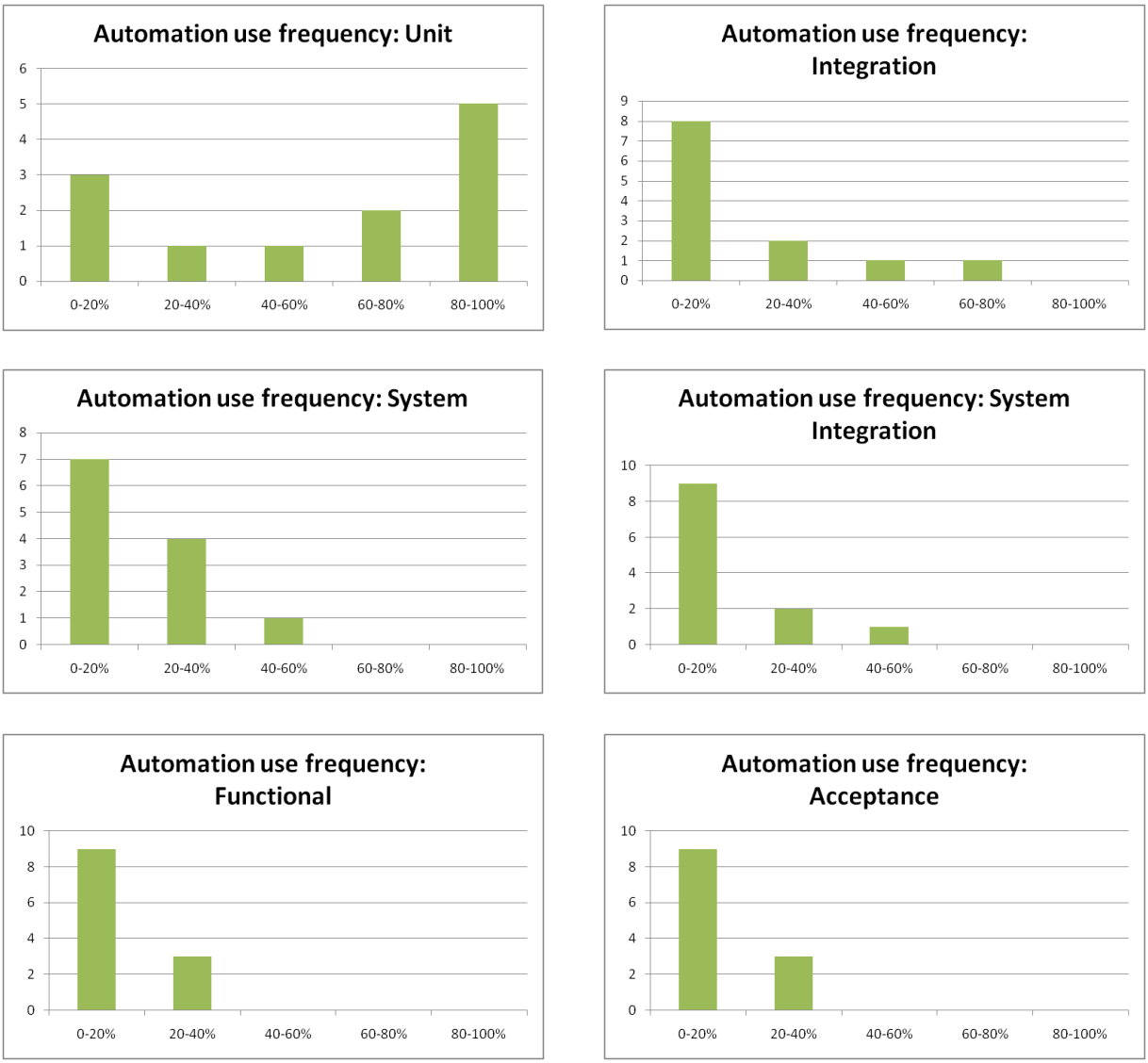
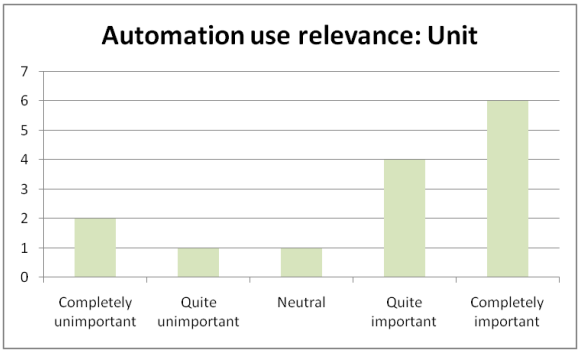
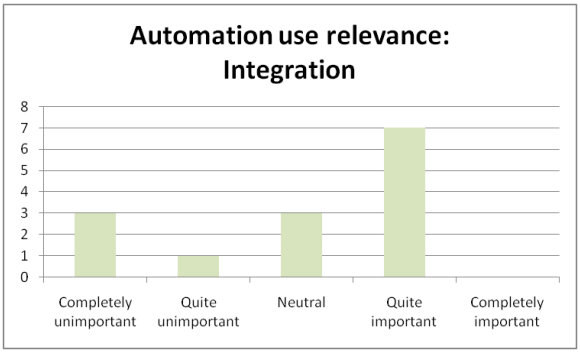
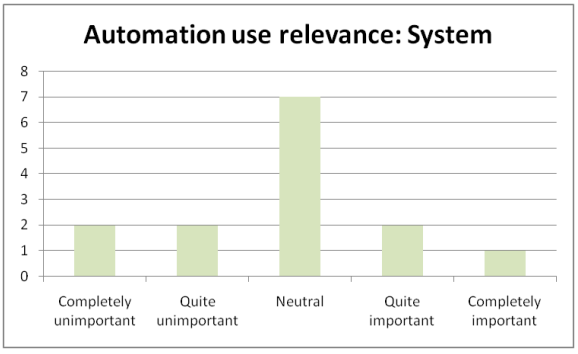


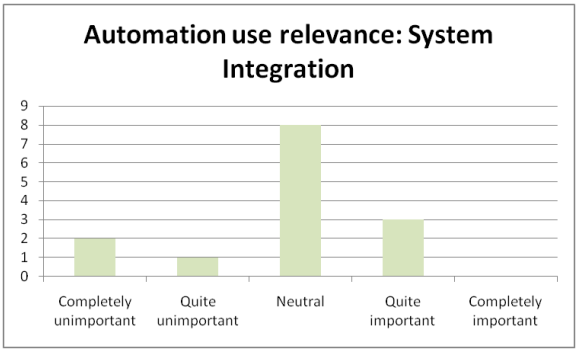
Table 4-8 Survey Results: Automation Use Frequency

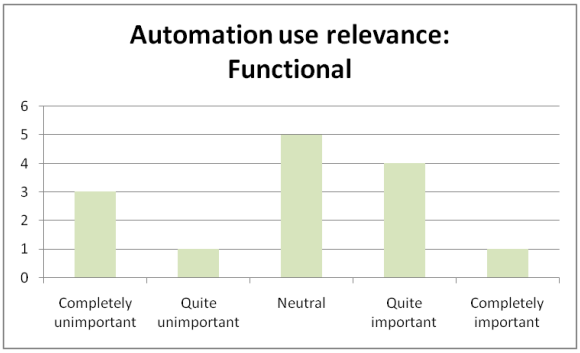
Table 4-8 above shows automation levels of all tests to be very low, mostly peaking around 0-20%. An exception forms the Unit tests where a distribution is observed ranging from 0 to 100%, indicating high variance in Unit test application, which is likely linked to the individual and private Unit test approach observed and uncovered during the semi-open interviews.











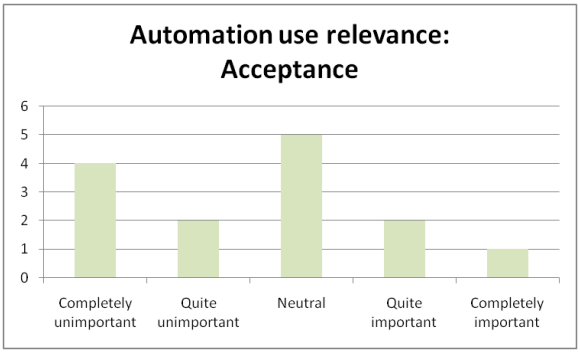
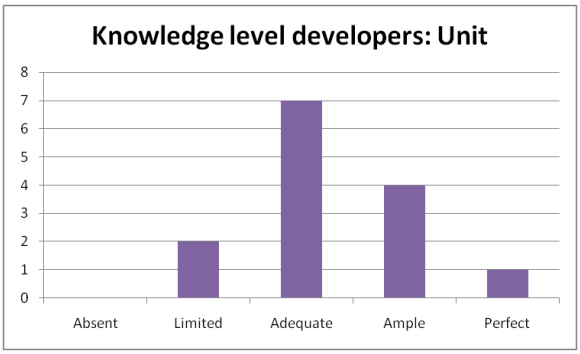


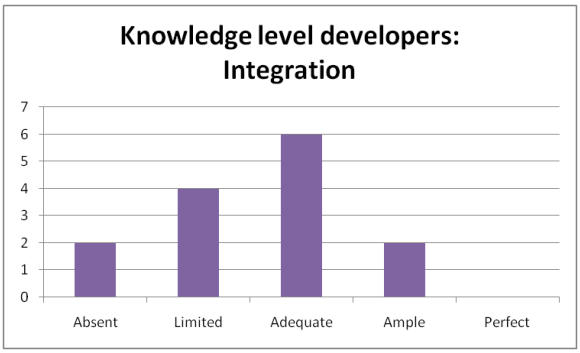
Table 4-9 Survey Results: Automation Use Relevance

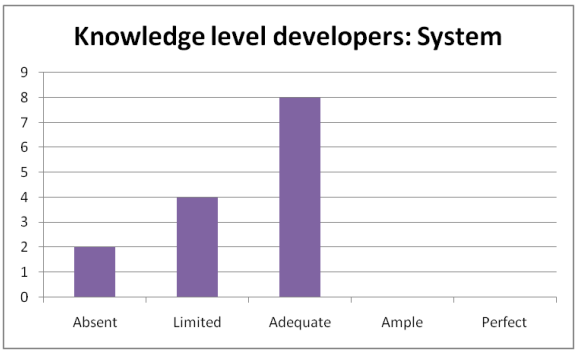
Table 4-9 above lists relevance levels for the test types. Unit and Integration tests are deemed important, where System, System Integration and Functional tests tend to result in neutrality. Finally acceptance test automation has a slight trend towards unimportance. Overall the matter of automation thus shows neutral opinions, however with Unit tests as highly important and Acceptance tests as slightly less important. Possible reasons for this neutrality weren't further investigated but could include: low awareness of automation potential or (especially for system testing) unexpected payback of automation over manually execution.

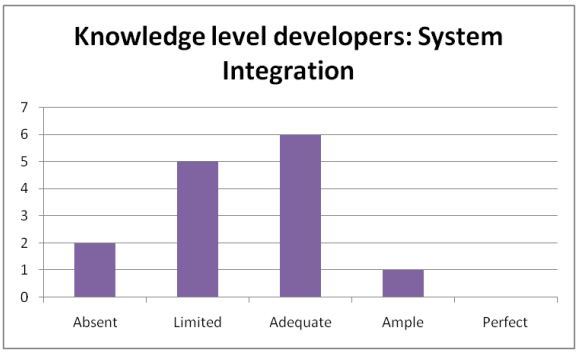
When comparing automation use frequency levels with automation use relevance levels of the two high relevance scoring tests Unit and Integration, automation levels have fallen behind on perceived importance demands (read: high automation).

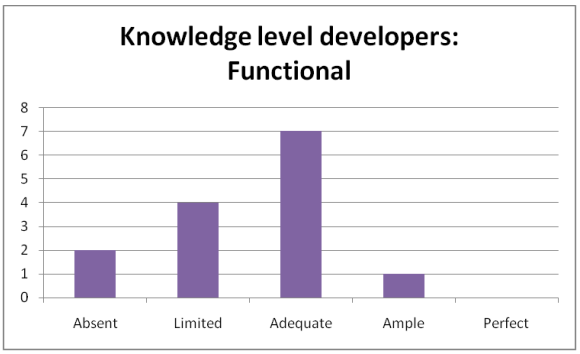
4.2.3.5 Developer knowledge level lacking, customers' level varies

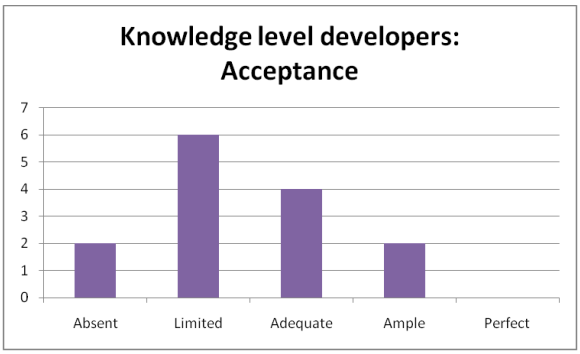


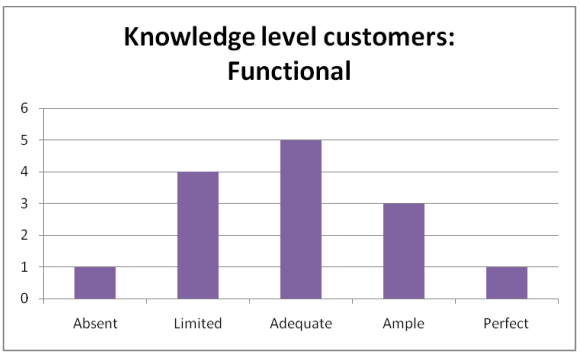












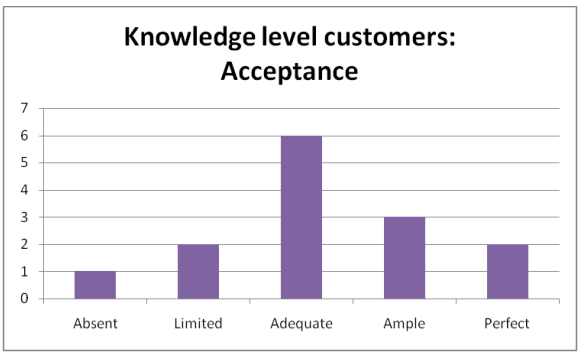


Table 4-10 Survey Results: Knowledge Level Developers / Customers

In Table 4-10 knowledge levels of developers as well as customers are listed. Developers are expected to have knowledge about all test types and thus were asked for all types. Customers however only see and perform Functional and Acceptance tests and thus their competence on only these two tests was asked. At least adequate scores on every test type should be attained for them to be performed properly. When observing the resulting data, this isn't the case. When discussing developers' knowledge level, all test types except for Unit tests show a distribution peaking towards lack of knowledge. Unit tests score adequate with a slight tendency towards ample knowledge. Possible reason for this 'anomaly' is that this test type requires hardly any testing skills, as their creation and execution lies close to 'normal' coding activities developers perform. When observing customer knowledge levels a somewhat normal distribution is visible. This shows what was observed in the interviews, that some customers have adequate knowledge of testing, while others haven't.

4.2.3.6 Test resources important yet unavailable

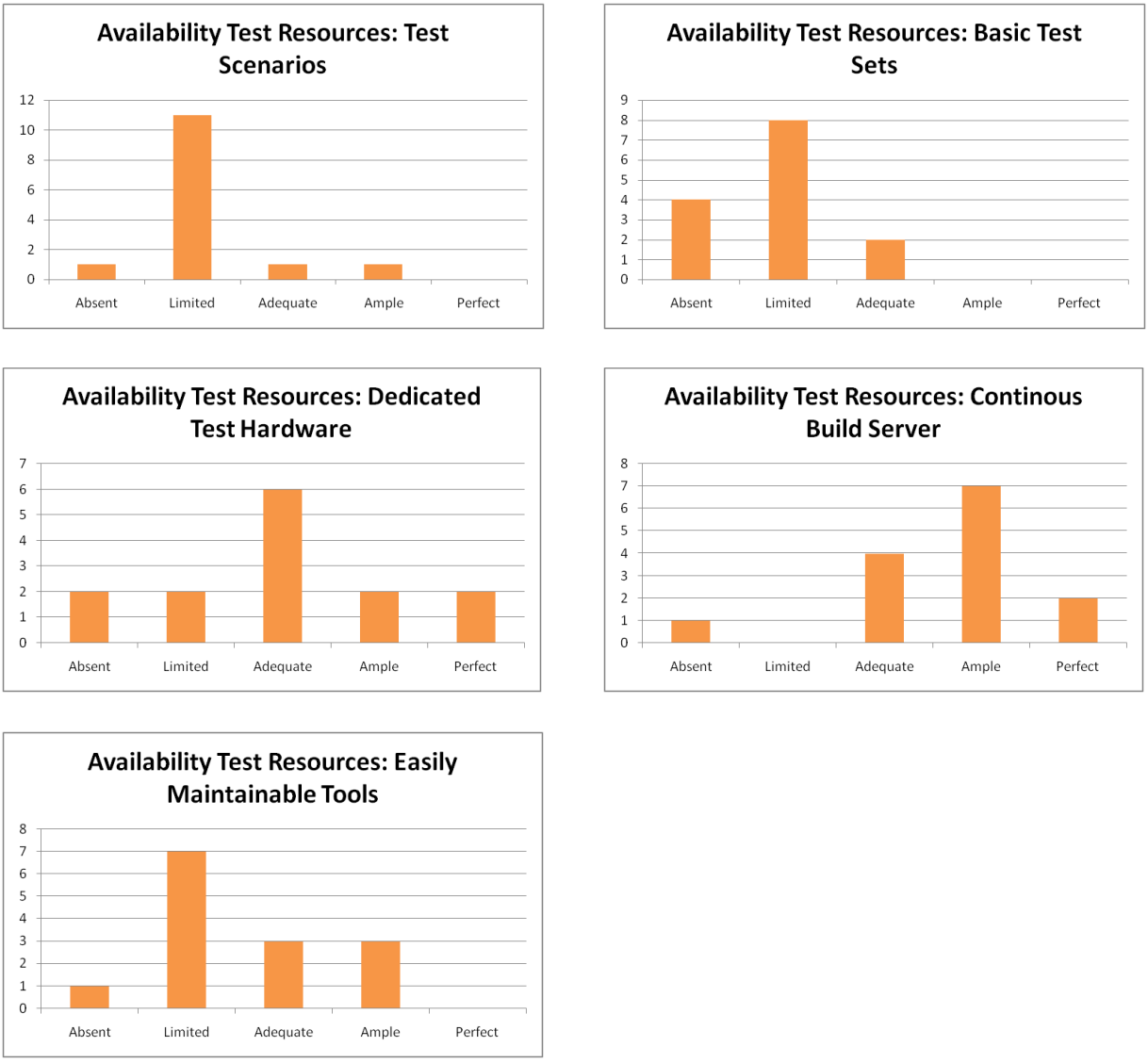
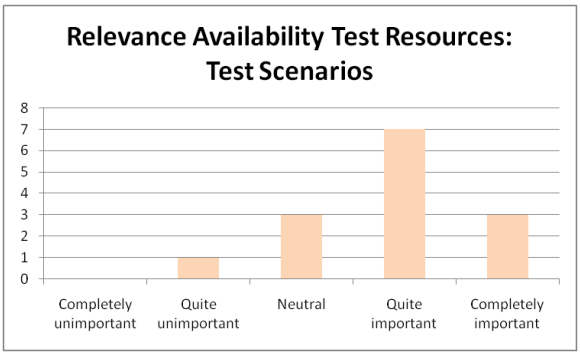
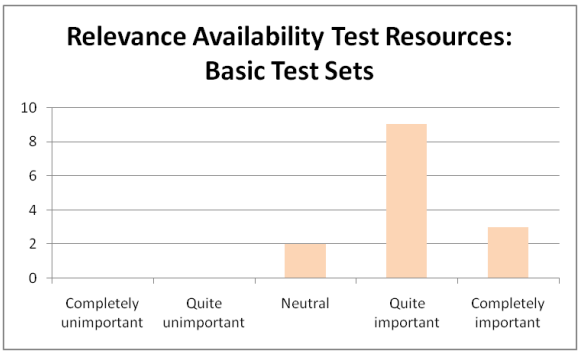
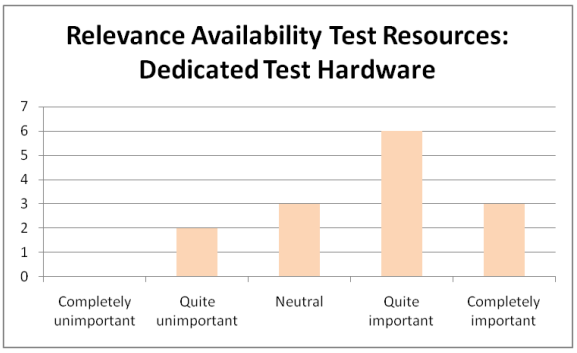


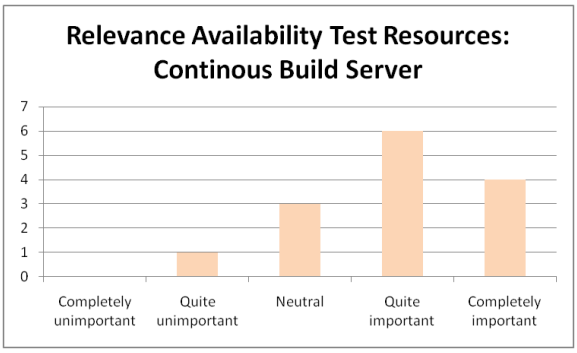
Table 4-11 Survey Results: Availability Test Resources

In Table 4-11 the availability of several test resources is visible. Test scenarios, basic test sets and easily maintainable tools are found to be limitedly available. dedicated test hardware centers around Adequate but covers all availability levels, so caution needs to be exercised when drawing conclusions from that data. Finally continuous build server(s) are available and score adequate to ample availability.









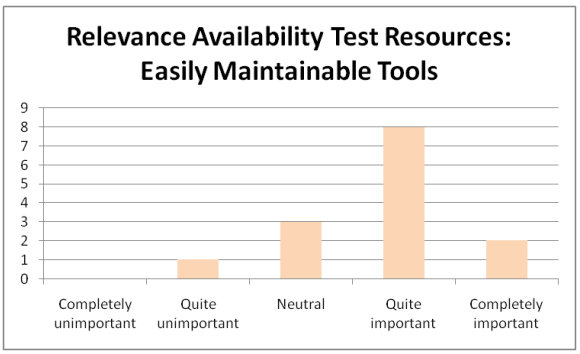


Table 4-12 Survey Results: Relevance Availability Test Resources

Table 4-12 shows the importance of test resources availability to developers and project managers. When examining the results of the survey, four out of five test resources look to have a similar relevance distribution, peaking towards quite important. The fifth test resource: basic test sets, shows a similar peak on quite important, but isn't mentioned as unimportant at all compared to the other four resources. So in general, all these resources are valued important.

When comparing these results to those of the test resources availability, they are found to be lacking, for at every resource availability relevance ranks high, while actual availability scores mostly limited.

4.2.3.7 Various statements

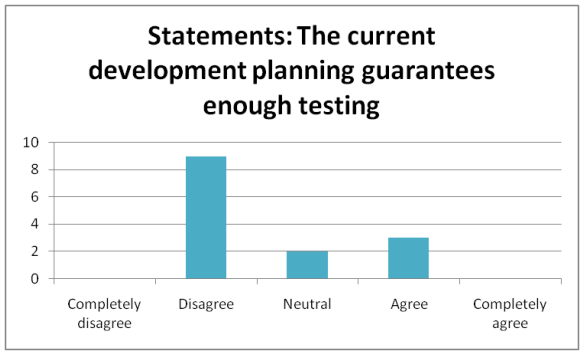
In Table 4-14, Table 4-16, Table 4-18 and Table 4-20 results of the survey's statements are covered.

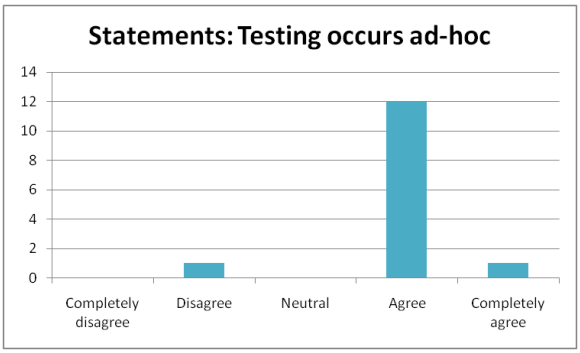
These statements were included to test anecdotes coming from interviewees for general validity. In short tables, actual results are compared to expectance and awarded a color code for amount of match. **CAPS GREEN** equals a perfect match, ***Italic Yellow*** a partial match and **Underline Red** no match. The results from individual statements will not be commented upon, for this hold few use to impact the total testing process. These statements serve more as a way to see if development is performed following agile principles and to see whether or not preliminary conclusions of the analysis are valid. Results from individual statements will be highlighted where applicable throughout this report.

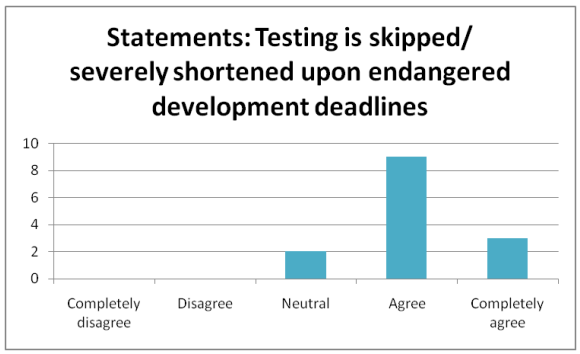
Table 4-13 Survey Results: Statements pt1

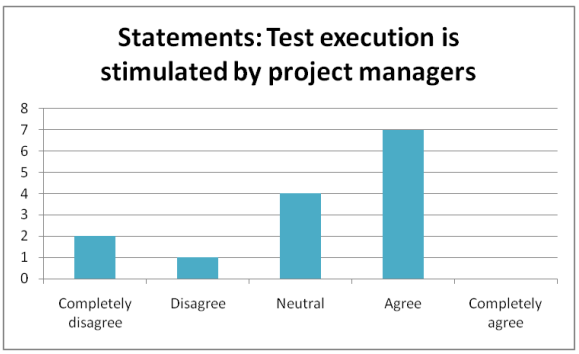
#	Statement	Expectance	Result
1	Customers should write own test plans	Agree	*Neutral - Agree*
2	Customers test properly	Disagree	*Disagree - Neutral*
3	Customers need support during testing	Agree	**AGREE**
4	Customers should test at SME	Neutral	*Disagree - Neutral*
5	Customers are available for context questions	Agree	*Neutral - Agree*
6	Customers are involved in the testing process	Neutral	**Agree**

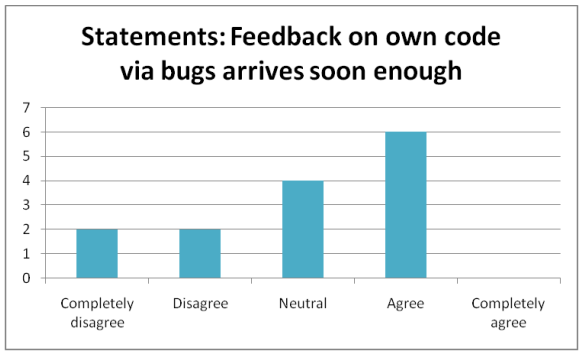
Table 4-14 Statements pt1 Expectance vs. Results











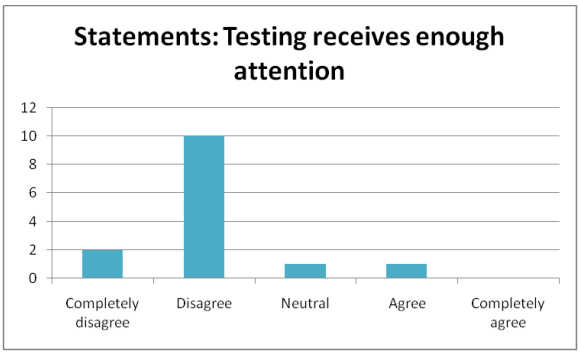
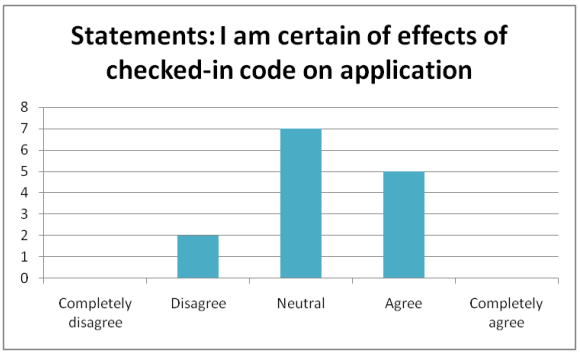
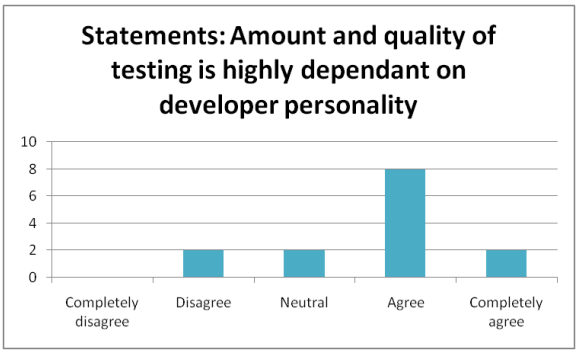


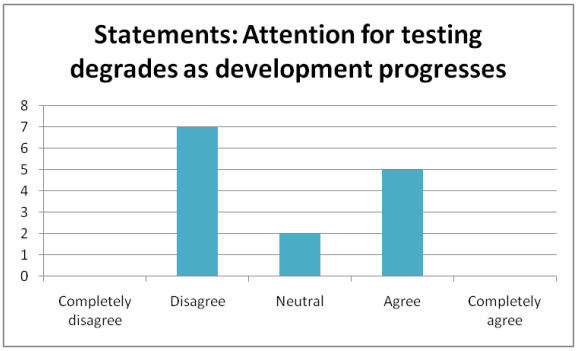
Table 4-15 Statements pt2

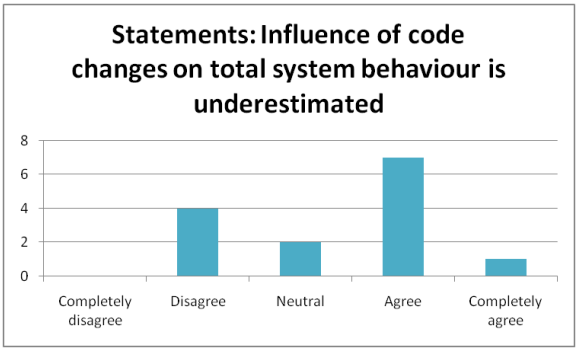
#	Statement	Expectance	Result
7	The current development planning guarantees enough testing	Disagree	DISAGREE
8	Testing occurs ad-hoc	Agree	AGREE
9	Testing is skipped/ severely shortened upon endangered development deadlines	Agree	AGREE
10	Test execution is stimulated by project managers	Agree	*Neutral - Agree*
11	Feedback on own code via bugs arrives soon enough	Disagree	Spread
12	Testing receives enough attention	Disagree	DISAGREE

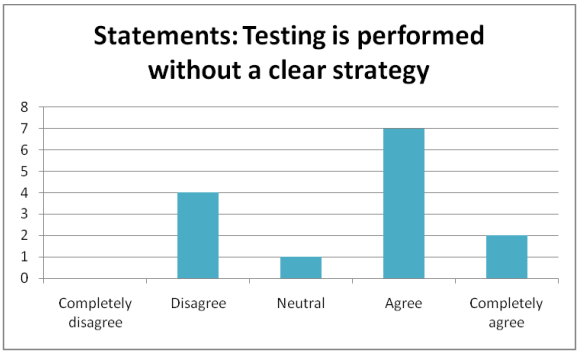
Table 4-16 Statements pt2 Expectance vs. Results











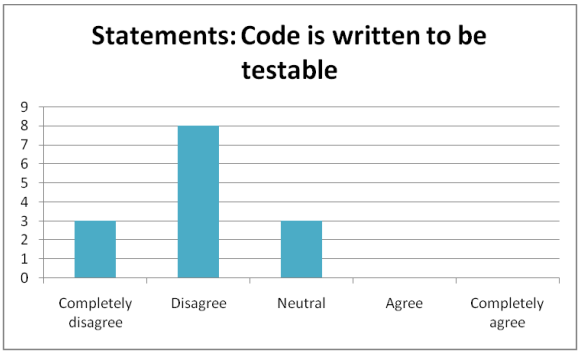
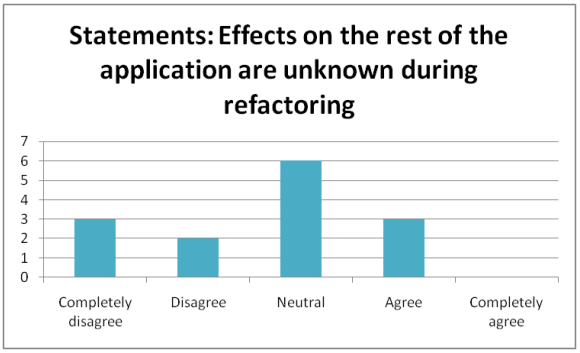
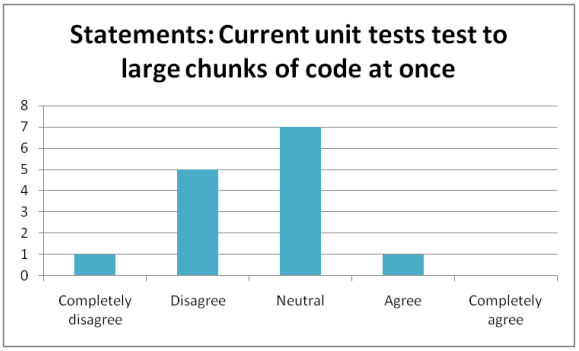


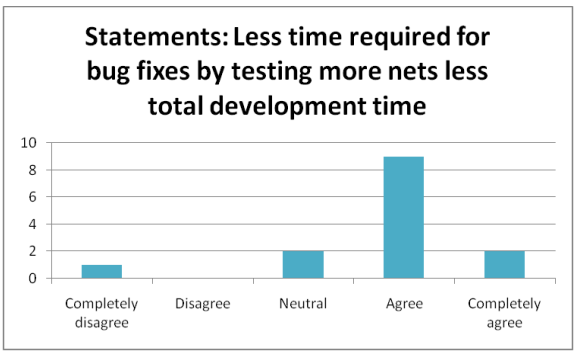
Table 4-17 Statements pt3

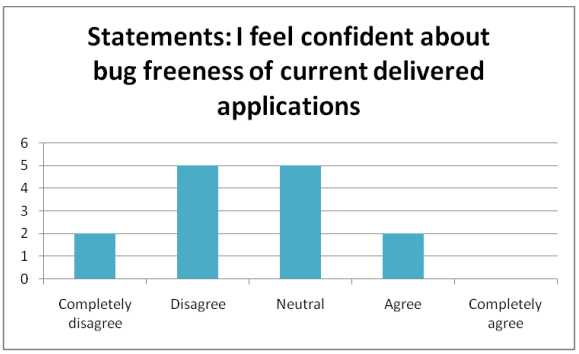
#	Statement	Expectance	Result
13	I am certain of effects of checked-in code on application	Disagree	Neutral - Agree
14	Amount and quality of testing is highly dependent on developer personality	Agree	AGREE
15	Attention for testing degrades as development progresses	Agree	Spread
16	Influence of code changes on total system behavior is underestimated	Agree	Spread
17	Testing is performed without a clear strategy	Agree	Spread
18	Code is written to be testable	Disagree	DISAGREE

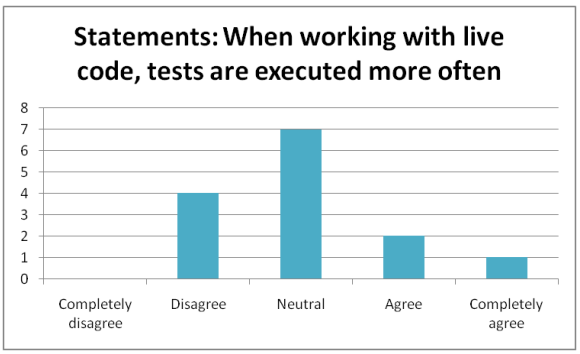
Table 4-18 Statements pt3 Expectance vs. Results











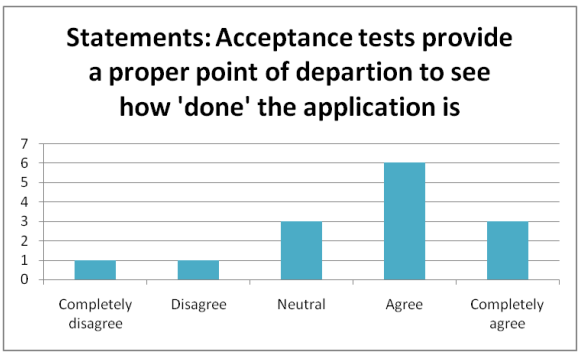


Table 4-19 Statements pt4

#	Statement	Expectance	Result
19	Effects on the rest of the application are unknown during refactoring	Disagree	*Disagree - Neutral*
20	Current unit tests test to large chunks of code at once	Agree	**Disagree - Neutral**
21	Less time required for bug fixes by testing more nets less total development time	Neutral	**Agree**
22	I feel confident about bug freeness of current delivered applications	Disagree	*Disagree - Neutral*
23	When working with live code, tests are executed more often	Agree	**Spread**
24	Acceptance tests provide a proper point of departure to see how 'done' the application is	Agree	**AGREE**

Table 4-20 Statements pt4 Expectance vs. Results

4.2.4 Qualitative additions of overall testing process

Originating from held interviews and comments placed at survey questions (all survey questions held comment options) a few general recurring qualitative remarks were made and are thus useful to list here:

- Unit and integration testing is currently considered by project managers as developer's own responsibility, which in practice results in these hardly ever being run. Previous individual attempts to begin these tests stranded due to difficulties with the many database couplings of web applications that were found hard to unit test. Also currently earliest integration practices take place when the developed software is at least at 80% complete.
- At the end of development there is a traditional separate testing phase, where time and resources are reserved for testing. This testing is comprised of system (+integration), functional and acceptance testing of the completed and integrated system, SME wrongfully calls this a system test. This testing is guided by customer approved test plans that include elaborate test-scenarios, however they are mostly of functional (use-case certification) nature and test too large chunks of functionality at once.
- In general functional and user acceptance testing is executed to a very limited extent by customers, mostly because customers to be handed over the ideal software package on time with as little as possible involvement. This then results in less freed time for customer employees to test the software. Next to this customers have very little knowledge of testing and thus do not work in a structured way or at regular intervals.
- System testing is in practice only performed when developers or customers perceive slow or malicious performance of the software at functional testing.

5 BEST PRACTICES

Several agile best practices have been identified via a structured literature study aimed at practices from agile development methods. The aim was to find practices that address the main question and underlying issues. The expected contributions to these goals per practice are attended at section 6.2.1 and thus won't be handled here. This section and underlying sections elaborate upon possible best practices, discusses their operation and propagates corresponding benefits for SME.

Section 5.1 handles Test-Driven Development as a key enabler for fewer defects in developed applications as well as reducing total development effort. In section 5.2 Continuous Integrated Testing – the support/enabler behind TDD – is shown. The rapid feedback loop and high project visibility it supports, should improve development significantly. Section 5.3 provides useful metrics for various development issues that need monitoring; this enables objective management steering and provides more quality securities. Section 5.4 is dedicated to a proper environment that fosters testing. The two notions the section hold originate from agile principles: (1) putting the customer into the centre of attention, whereas all agile methodologies require intense collaboration between developers and customers to create A-grade software. (2) Competence diffusion. For developers were found to have limited knowledge of testing types and procedures, a quick comeback is required. This can be achieved via stimulated spreading of knowledge.

5.1 Testing paradigm shift: XP's Test-Driven Development

The first best-practice identified is XP's Test-Driven Development (TDD), it draws from TDD practices, but takes its practices to extremes. Section 5.1.1 shows its essentials (and extremes), while section 5.1.2 is all about its possible benefits for SME.

5.1.1 XP TDD in short

XP incorporates a new paradigm on testing. Where most (older) development methodologies show a separate lengthy testing phase at the end of development, XP comes with a new volatile way of testing – a paradigm even – : *always* test before coding. This powerful statement originates from the concept of Test-Driven Development. (Beck 2002) Paraphrased to strengthen the statement: "Only ever write code to fix a failing test". (Koskela 2008) This test-first paradigm of TDD is best explained using a graphic. Figure 5-1 shows the test-first design process is shown.

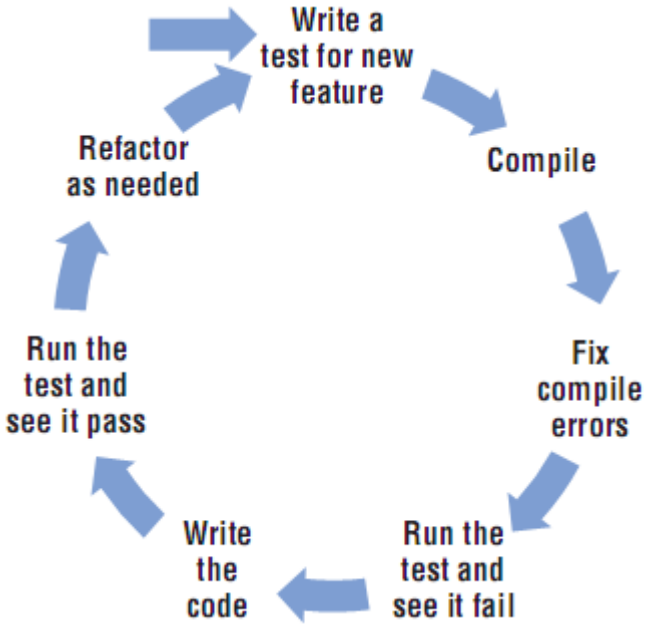


Figure 5-1 Test-first design process (Grenning 2001)

Clearly visible is the precondition of having tests before coding, but lacking in this picture is the 'naturally regarded' fact that when a developer thinks up a test while coding, this can and should be added as well in the XP setting when he/she thinks the need arises. Testing up front can also be regarded as an as analysis step, for one is required to clearly decide on what the code should do.

Next to demanding tests to be built first, XP states that all added code *must have* accompanying unit and function tests, and integration / regression tests are executed as soon as possible (for integration: when more than one components have functions that can be tested for integration, for regression: as of the first available test) . Next to taking testing as leading over coding, the idea is to automate as much tests as possible. When automated this provides possibilities to *extremely* demand that for every added code sequence all previous tests *must be* ran and test scores must be no less than *100%*. In this way developers' confidence will be greatly improved with every code addition, after all: everything coded so far works completely, and if tests fail they immediately know exactly where the fault lies!

Furthermore a unique feature of XP is that developers write their own tests, but soon this results in the obvious statement 'one can never test his own code as well as someone else can', because of losing some white-box[16] advantages that only occur when self-writing tests. (Cockburn and Highsmith 2001) This statement can be refuted by another XP practice: pair programming, being two developers coding on just one screen and keyboard. Along with many other benefits (not covered here) hereof the chance that code faults are missed is greatly reduced by two minds working as one. (Cockburn and Williams 2000)

[16] The developer that built a code piece knows the executed methods, dependencies and objects handled in the code piece best, for he/she developed it his/herself and is the (sole) mind behind all design considerations and decisions.

System and acceptance testing is performed by the customer under collaboration with analysts. At the *beginning* of a development iteration the customer thinks up what would convince him that the requirements for that iteration are fulfilled. These convincers are then converted into system wide tests.

Sometimes here the need for aid of a developer / tester to empower the customer with coding experience is needed. Because the customer has the most expertise and feeling of how delivered applications must work – after all he knows/supplies the business logic – this isn't performed in-house.

For clarification Figure 5-2 below shows the relations of tests, code and actors:

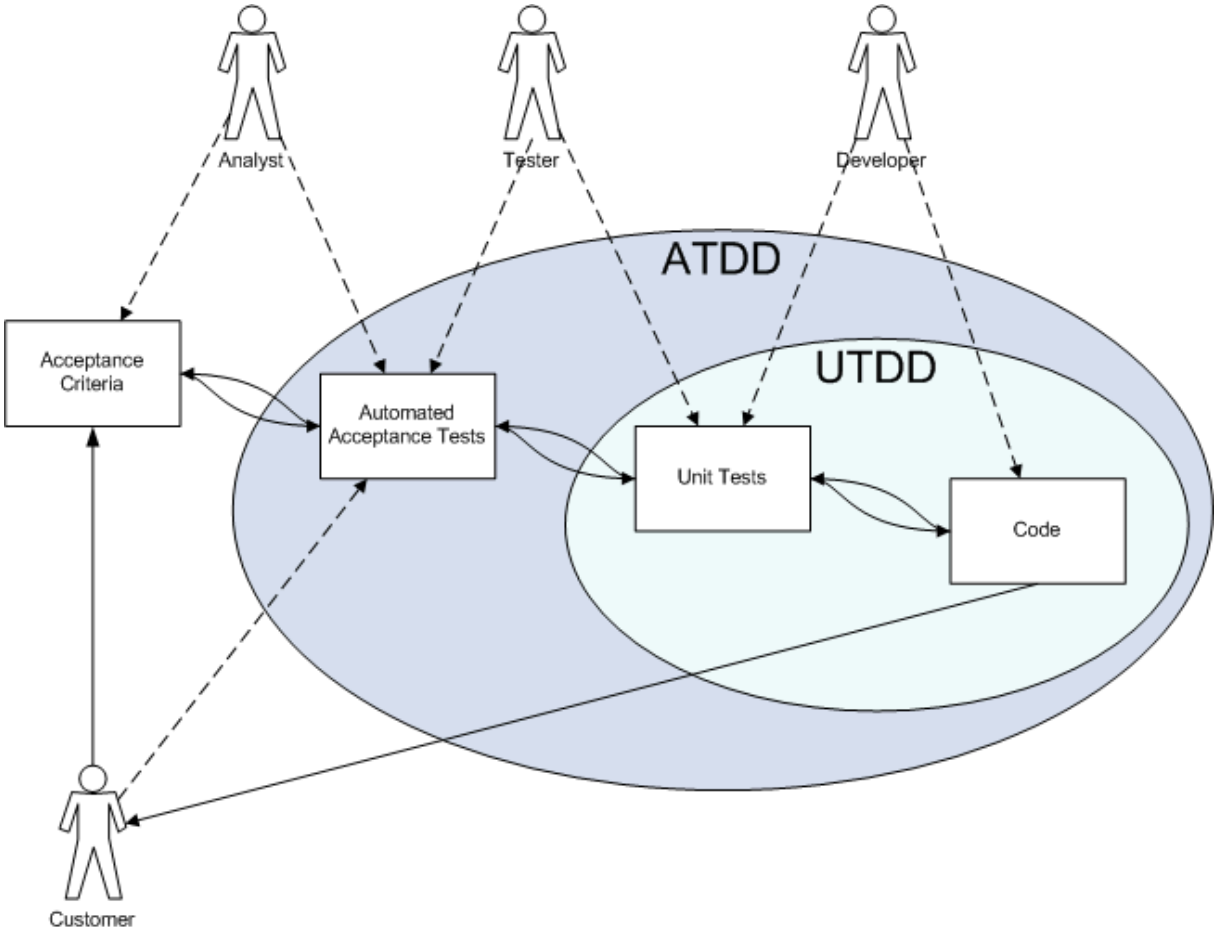


Figure 5-2 Relations in TDD explained

The figure shows a division in two types of TDD, Acceptance Test Driven Development (ATDD) and Unit Test Driven Development (UTDD). Where the former is concerned with building the right application, the latter tries to build the software right.

In a typical development cycle the customer draws up acceptance criteria together with an analyst. After agreeing upon these criteria, these are then converted to automated acceptance tests with help from testers providing coding and test cases experience. As soon as the first automated acceptance test – which guides further development effort – is in place the actual implementation of the criteria/features is started. This enters the domain of UTDD. Every automated acceptance test is covered by at least one unit test devised by developers themselves and supplemented by tester experience when necessary. These unit tests than serve as input for actual code implementation effort. When the implemented code passes all unit tests, the UTDD cycle is completed and the feedback towards the ATDD cycle commences. The final step in the larger cycle is to make the acceptance tests pass, when this is the case the customer criterion is considered met and the entire cycle starts over from the beginning by supplying a new automated acceptance test.

5.1.2 XP testing: various benefits

Several authors report a variety of benefits when applying XP testing, while some aren't directly linked to quality as defined at section 1.3.2 they will be enumerated on the next page.

Benefit category	Benefit	Author(s)
Higher software quality	Higher testing frequency combined with immediate test feedback increases software quality, because flaws are detected earlier and more often as well as programmers retaining a higher learning curve due to shorter feedback cycles.	(Beck 2000; Erdogmus, Morisio et al. 2005)
	Test automation improves software reliability, for automated tests are executed more often and achieve higher accuracy than manual ones.	(Maximilien and Williams 2003)
	Requiring written tests upfront of coding raises programmer's and customer's awareness of desired process outcomes, thus improves insights on how following code must rise. Also better insights in required functionality improve software quality.	(Erdogmus, Morisio et al. 2005)
	Test-driven methods as XP report up to 50% reduction in defects compared to ad-hoc testing, it also leads to less time needed in manual debugging and thus reduces fault injections –the probability of bugs in hacked debug code is over 40 times higher than with original code – due to hack-based debugging.	(Maximilien and Williams 2003; Williams, Maximilien et al. 2003)
Increased develop-ment speed	With automated high coverage testing, at every code edit a developer knows the new code works, for all test must run and pass at every code injection. As developers know exactly when they're done (tests run and pass), they can move on to the next task quicker.	(Jeffries 1999)
	Automated testing reduces total testing effort. (exact effect unmentioned)	(Dustin, Rashka et al. 1999)
	Less time spent fixing defects, for the reduction in defects injection and earlier discovery due to very small development cycles is bound to pay back.	(Koskela 2008)
Increased customer confidence	Increased involvement via continuous functional testing and seeing results thereof increases confidence in functionality for customers are convinced that their own tests pass and thus have desired functionality.	(Beck 2000)
Increased developer confidence	Developer confidence in functional and system reliability improves dramatically due to automated tests always require a 100% score and provide direct feedback to validity of checked-in code. In ad-hoc – and mostly too late – testing, results usually take days or weeks to get back at the original developer and even then they provide limited insights in to where to bug originates.	(McMahon 2003)

Table 5-1 Benefits of XP testing

N.B. XP TTD affects four out of six available test types. The two types unaffected are System and System Integration Testing. But this isn't problematic, for past SME development projects haven't dealt with many issues that could have been counteracted with these two testing types. Observations proved that the current reactive attitude towards system performance and integration problems suffices. A second reason for not being problematic lies in the fact that the current development process is expected to improve (in terms of defect levels) most by applying XP effected test types.

5.2 Beyond tools: Continuous Integrated Testing

The second identified best-practice is Continuous Integrated Testing, which extends Continuous Integration (CI) with proper testing. First the concept of CI will be clarified in short at section 5.2.1, after which full CIT will be shown at section 5.2.2. The CIT practice provides rapid feedback, which ultimately reduces required time and money to detect and fix defects. Section 5.2.3 lists previously observed benefits from various authors that should prove beneficial to SME. Section 5.2.4 ends the information on the CIT practice. For it leans heavily on automated testing, this final section is included to remind the reader once more of the possible pitfalls of automated testing. Also issues corresponding with pitfalls of manual testing are mentioned in contrast. These are visible at SME. This backs the notion of a needed change in testing.

5.2.1 Continuous Integration explained

Before discussing CIT an explanation on Continuous Integration is required. The primary explanation of a typical CI scenario is supplied below:

1. A developer commits code to the version control repository. Meanwhile, the CI server on the integration build machine is polling this repository for changes (e.g. every few minutes),
2. Soon after a commit occurs, the CI server detects that changes have occurred in the version control repository, so the CI server retrieves the latest copy from the code from the repository and then executes a build script, which integrates the software,
3. The CI server generates feedback by e-mailing build results to specified project members,
4. The CI server continues to poll for changes in the version control repository.

Figure 5-3 illustrates the parts in a CI system, as described by the foregoing CI scenario activities.

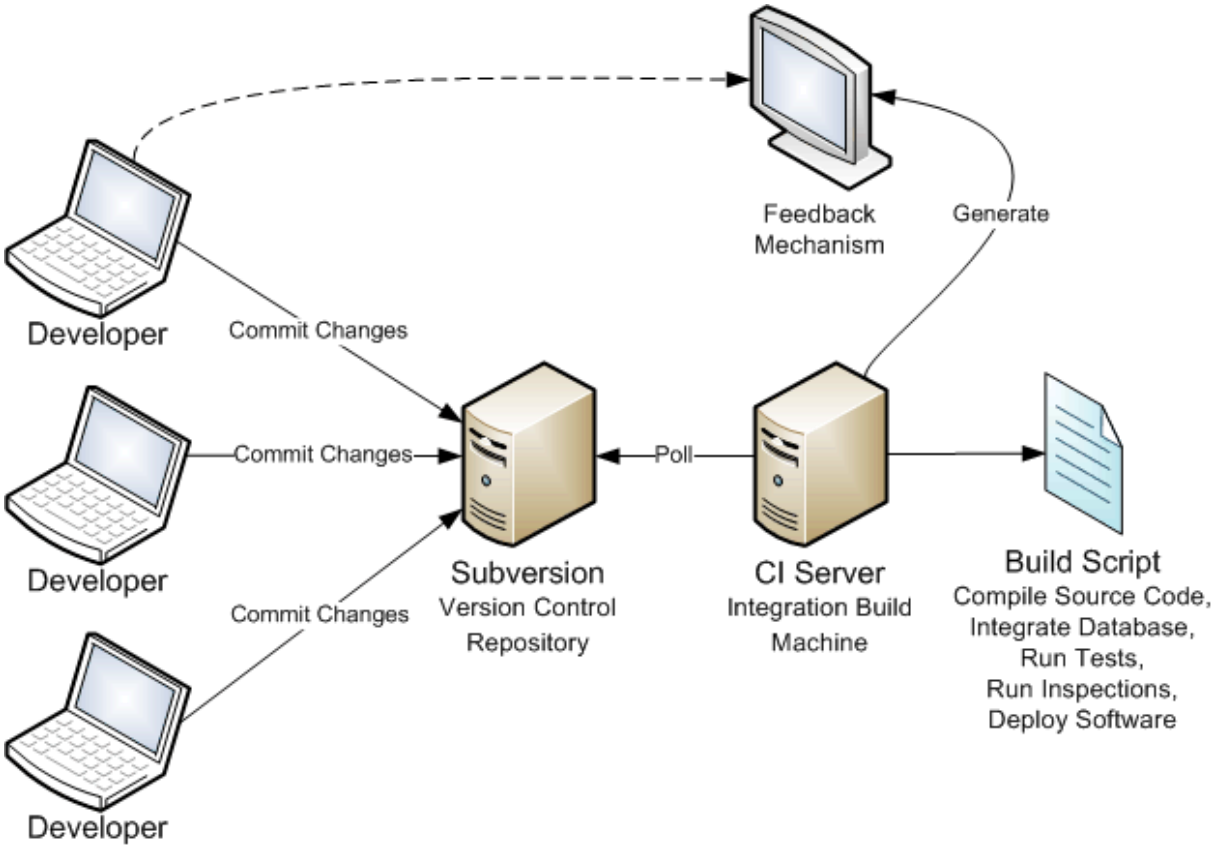


Figure 5-3 Components of a CI System (Duvall 2007) (adapted)

An expansive description of CI that won't be handled here is available from one of its founding fathers, Martin Fowler. Please refer to: Fowler (2006).

5.2.2 The CIT approach

"Continuous Integrated Testing is a combined development and testing methodology that enables organizations to maximize the use of testing throughout the development process to increase overall application quality. More specifically, CIT enables unit and functional tests to be run while profiling the application code. It provides developers, testers, and managers with daily/hourly/minutely updates of the performance and stability of the application in development. Rather than waiting for a scheduled code release or predetermined date and time for checking the overall status and health of an application, CIT can be used throughout the entire development phase to provide detailed, granular analysis that highlights issues such as performance, memory or poor coding." (Compuware 2006)

In short: CIT is a new approach for cost-effectively increasing both the number of testing cycles and the resulting application quality while decreasing the amount of time it takes to find problems and the cost to fix them (Compuware 2006; Fowler 2006; Haines 2008).

In traditional waterfall software development methodology, application testing is performed as a separate development phase that starts no earlier than at completed development. At CIT however, this testing is performed in parallel with coding development activities. The results are continuous test cycles, enabling developers to isolate defects when they're injected and take corrective action immediately. Thus instead of developing large chunks of code and simply 'pass it over the wall'[17] to testing, resulting in late and less effective tests to discover errors, now testing and coding forms a synergy. This practice is sure to improve development.

CIT aligns developer performance tools with testing. Testing tools are used to create and regressively execute tests while developing the application, enabling them with method and line-level details on performance, memory issues or test code coverage.

When this process is automated and fused with developer performance tools, a daily/hourly/minutely repeatable process can be forged that highlights issues introduced in that timeframe. Despite an initial reaction that this practice will cost more time, it reduces testing efforts due to eliminating defects at their roots, even before it touches other code under development. This shortens testing thus reducing solving cost per defect.

5.2.3 A variety of benefits from CIT

Benefit	Description	Author(s)
Less defects	Developers are empowered by continuous (testing) feedback loops on integrated code that net a significant reduction in defects.	(Compuware 2006; Fowler 2006)
Detect and fix defects earlier	Defects are easy to pinpoint, because of short build + test cycles and thus a limited area to look for them; when testing several times a day, defects are more likely discovered upon injection, instead of at late-cycle testing; amount of assumptions to test is reduced by limited search window; defects are always reproducible; find bugs that are (practically) impossible to find manually; error status monitoring, CIT allows for auto-entry into defect trackers and metrics, which aids early fixing of defects.	(Dustin, Rashka et al. 1999; Fewster and Graham 1999; Compuware 2006; Flowers 2006; Fowler 2006; Duvall 2007)
Less development time	Defects are found earlier when they are less likely to be compounded; automated execution saves a lot of costly testing team time and increases test numbers; test case generation and recording tools speed up test creation; decreased debugger time; testing doesn't require traditional code freeze.	(Dustin, Rashka et al. 1999; Compuware 2006; Flowers 2006)
Less costs	Bugs are discovered and corrected immediately, instead of weeks or months later, saving costly debugging costs.	(Compuware 2006)

[17] Forwarding code without proper comments or any other form of communication, let stand collaboration between development and testing actors.

Benefit	Description	Author(s)
Gap development ←→ QA bridged	The CIT process takes down the wall between development and testing. The new structure of development requires developers to become part tester and vice versa thus creating collaborations and mutual respect; also the use of common testing tools creates synergies between Development and Quality Assurance.	(Dustin, Rashka et al. 1999; Compuware 2006)
Measurable quality	CIT process monitors health attributes such as total build status, quality metrics, defect rates and completion rates; problematic areas are automatically highlighted at every change or compile and trends are visible early on; automated tests procedures can be measured and repeated, this allows for previously (manual) hard to reach analysis and optimization; automated tests provide easy and adequate reports.	(Dustin, Rashka et al. 1999; Compuware 2006; Duvall 2007)
Deploy at any time	CIT's short integration cycles allows for (almost) immediate deployment of working builds	(Flowers 2006; Fowler 2006; Duvall 2007)
More higher-ranked development time available	Reduces repetitive manual processes; Continuous testing reduces variance in defect arrivals, decreasing frequency of rush jobs in bug fixing; test playback can be unattended	(Dustin, Rashka et al. 1999; Fewster and Graham 1999; Compuware 2006; Duvall 2007)
Raised confidence in application	With every build an extensive set of tests should pass, immediately showing the current build status to developers	(Fewster and Graham 1999; Duvall 2007)

Table 5-2 Benefits of CIT

5.2.4 Remarks on automated testing

While the previous section has pointed out a variety of benefits from applying CIT, some limitations of the automated testing it holds do apply and deserve at least a mentioning here: (Fewster and Graham 1999)

- It *does not replace manual testing*: tests that shouldn't be automated are the ones being (1) rarely run, (2) where the software is highly volatile, (3) where the result is easily verified by a human instead of a test script and (4) ones that require physical interaction.
- *Manual tests uncover more defects* than automated tests: because tests most likely reveal defects in their first run. Automation only re-runs tests and thus are less likely to discover latent defects. Nevertheless, an extensive experience report by James Bach shows that automated tests find 15% of defects, not uncovered by manual testing. (Bach 1997)
- *Greater reliance on test quality*: Because a tool only compares actual to expected outcome, it becomes a greater burden to verify the correctness of expected outcomes. A tool will happily report pass on a test, while it has only verified a match in expected outcome.
- *It doesn't raise effectiveness*: automated tests aren't more effective than their manual counterparts. Automation only improves efficiency (costs and time).
- *It may limit development*: Automated tests are more fragile than manual tests, they can be broken by minimal changes to the tested application. When written properly this fragility can be countered. (Currently at SME a team is busy defining a template how to create proper tests which should counter this limitation)
- *Tools have no imagination*: A tool is just software, and thus only obediently follows instructions. It holds no creativity and cannot adapt to exceptions in execution or situational conditions as humans can.

Then again manual testing suffers from several issues as well: (Crispin and House 2003)

- *They're unreliable*: Under schedule pressure, quality of testing decreases. Developers/testers start to cut corners, omit tests, and miss problems. A quote by Lisa Crispin shows than when using automation with immediate result feedback keeps people from doing these undesired practices: "The warm comfy feeling the manual tests gave us by promising to keep defects from getting through to the customer is replaced by the burning flames of perdition". Next to manual testing under individual developers creates unwanted high variety in test quality level.
- *They undermine proper development*: Developers keep tests to themselves or even skip on them completely, manual tests are practically impossible to verify if they've been executed or even designed.
- *They're divisive*: Manual tests are very personal, when you discover a defect with a manual test, *you* found the defect. When you miss one, *you* missed it[18]. If someone misses something when stakes are high, it's the person doing the test that failed. He should have seen it, even if he was distracted, under pressure, etc.. This kind of atmosphere is unwanted in a project, and can be avoided by automating the test.

[18] Pardon my phrasing, the statement made here is only perceivable in a dictating personal phrasing.

The aforementioned manual testing issues are visible at SME as well (and have been shown at various places in this report), which strengthens the case for automation.

5.3 Metrics: A new set of performance indicators

At SME the only current internal software quality performance 'indicator' is the project managers' gut feel originating from the bug arrival rate. This calls for high uncertainty. To counter this uncertainty and at the same time move from subjectivism to objectivism a set of direct and indirect performance indicators is suggested further on as the third best-practice. The developed measures will provide project managers as well as developers with a simple means to evaluate current software reliability standings and to enable them to steer on test amounts and quality. (Jones 1997) Jones also states that there is a perfect correlation between being able to measure and to estimate, for planning and estimation are the mirror images of measurement. So not only current testing performance will be clarified, but also what the future holds on software quality.

Via a systematic literature search a diversity of testing performance indicators was uncovered. These indicators will be enumerated below, combined with a discussion about their usability. They have been selected to fit business goals (1) improve quality (reduce total defects), (2) find and fix defects closer to their origin, (3) predict find and fix rates pre en after release, and (4) allow for comparison to earlier development results.

Next to required alignment with business goals, the criteria for selection were:

- Easy to comprehend
- Macro view
- Relevant to the testing process
- Driving improvement
- (Technically) measurable

The number of selected metrics was also kept at a minimum yet adequate level to guarantee test process improvement, and to prevent micromanagement and bureaucracy, which suits the agile approach and corporate identity at BPT. The selected metrics set corresponds to business goals:

Metrics to be implemented are split into direct measurements of test activity levels, measurements of arriving/remaining defects and overall testing performance. The first and second categories both hold two metrics and the latter limits to one metric. (Variations on metrics aren't included in this count). This total package of five indicators should provide SME with the tools to steer internal testing activities and quality and at the same time deliver a means to show customers a reasonable indication of the quality level of their software. Sections 5.3.1 - 5.3.3 show the suggested metrics. Final section 5.3.4 holds some insights for SME to know when to stop testing, and shows why this research hasn't exhaustively researched all possible methods to decipher the point of enough testing.

N.B. Results of this chapter rely heavily on the work of Kan (2003), who provides a detailed overview with adoption recommendations on currently available software development metrics and makes recommendations for adoption on a variety of metrics. Where necessary their work is supplemented by opinions of other authors.

5.3.1 Measurements of test execution levels

The first category of indicators deal with the monitoring of current testing levels. Structurally dealing with testing levels improves software reliability. (Maximilien and Williams 2003; Williams, Maximilien et al. 2003; Erdogmus, Morisio et al. 2005) To that extent two metrics were uncovered fit for SME use. These are the measures 'code coverage', highlighting the fraction of code covered in tests , and 'Test Progress Curve' showing the buildup in performed tests with accompanying results thereof. When used in combination, this will result in a simple yet precise overview of what was tested and to what amounts. This raises reliability of developed software through steering on adequate amounts of testing and raises trust of developers as well as project managers in the code due to having access to an easy overview of current standings (metrics read good or bad in terms of quality or schedule) (Kan 2003).

5.3.1.1 Code Coverage

Code coverage is divided into two coverage types. *Edge coverage* reports which branches or code decision points were executed to complete the test. *Line coverage* reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. Both tests report a coverage metric in the form of a percentage.

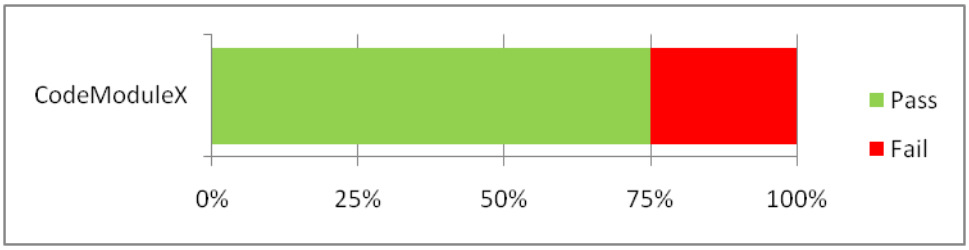


Figure 5-4 Sample Code Coverage

These two coverage metrics should provide a simple and thorough way to see if all parts of the code have been tested *at least* once.

Purpose of these metrics is to show what *isn't* tested, to focus upon writing new tests to check missed code parts. Also it is easy to set target levels of test coverage, to force developers to write more tests when achieved coverage levels are found lacking, or to show that test levels are on or even over target.

However, caution needs to be exercised upon usage however as Marick (1997) argues: Code coverage should be used for performance management, but only in retrospective. Thus this metric cannot be applied solely to lead test design, but is used better in evaluation.

This warning is due to two apparent notions: (Marick 1997)

First the notion that tests required for proper testing *always* amount to more than the ones required for coverage goals. Tests that don't necessarily improve code coverage are required as well. This is due to the way bugs behave. Bugs that occur independent of how the code is executed, will always surface (and thus will be found using coverage). Bugs that depend on interoperation or only surface in specific conditions will require redundant tests in terms of coverage metric demands. In short: tests only satisfying code coverage demands are *never* enough for proper testing. Second notion is that of cutting corners under pressure. People will – especially under pressure – supply exactly what you measure. This implies tests only being written when they raise code coverage towards the set target, even if they're of bad quality and are less likely to reveal defects they can fool the metric's purpose.

To counter these aforementioned possible risks of applying code coverage as a way to see 'how done' testing is, the following guidelines from the 'Certified for Windows Logo Procedure' should make the difference: (Bach 1998)

The following supplementing test coverage requirements should always apply:

- Test all the primary functions that can reasonably be tested in the time available
- Test a sample of interesting contributing functions
- Test selected areas of potential instability

Applying the supplementing demands, next to the original quite elementary code coverage metric of simply a percentage of coverage, should make a fit for use metric. The general metric will function as a blunt way of seeing if test case numbers are adequate, while the additional requirements function to keep test design quality high.

5.3.1.2 Test Progress Curve (Planned, Attempted, Actual)

This test cumulatively tracks the number of test cases over time. The resulting 'S' shape is a result of a period of intense testing activity causing a steep ramp up in planned tests. The graph shows:

- Planned number of test points to be performed successfully per week
- Number of test cases attempted per week
- Number of test cases completed successfully per week

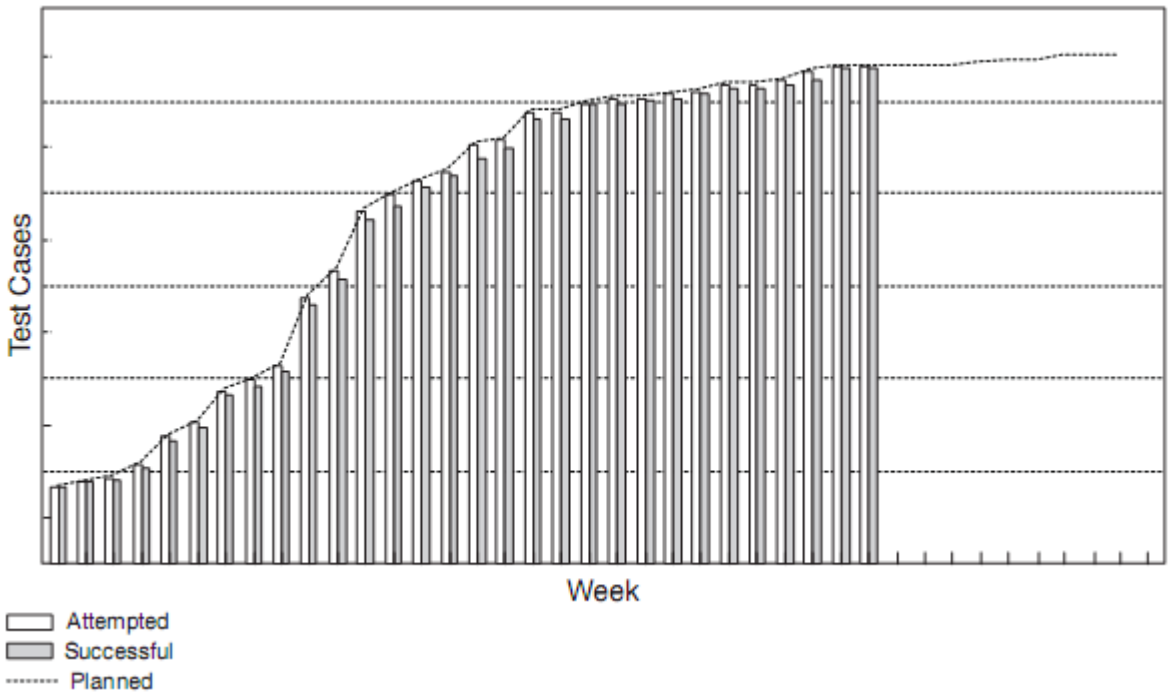


Figure 5-5 Sample Test Progress S Curve (Kan 2003)

Purpose of this metric is to track test progress and compare it to the plan, and therefore be able to take action upon early indications that testing activity is falling behind. It is well known that when the schedule (often!) is under pressure, testing is affected. With this metric in place, schedule slipping is much harder for the team to ignore. For instance a disparity target of 10% between attempted (or successful) and planned can be used to trigger additional actions. Another purpose is that it forces planning for numbers of test case upfront and demands testing to be performed continuously instead of at the end of the programming cycle. A final application of this metric could be that of release-vs-release or project-vs-project comparison to compare quality and schedule. This metric is best suited for unit / functional tests.

5.3.2 Indirect measurements of code quality through defect analysis

Next to test reach and effect measuring also defect arrival tracking is relevant to quality measuring. For instance even with the same overall defect rate discovered during testing, different patterns of defect arrivals may imply different scenarios of field quality. Two additional indirect metrics have been identified to improve testing even more, divided into defect arrival and defect backlog tracking.

5.3.2.1 Testing Defect Arrivals over Time

This metric tracks the number of defect arrivals over time. On the X-axis the weeks before product ship are listed, with accompanying Y-values as the number of defect arrivals for the week.

Like with earlier mentioned metrics, this metric can also be customized to show separate development phases or defects discovered by different test types. Differentiation is also possible to show differences in defect severity or instead of absolute values, customized to show relative arrivals. Another very useful variation shows defect origin, when all defects are awarded to their origin it becomes easy to see where most mistakes are made and undertake actions to prevent this from happening again. Figure 5-6, Figure 5-7, Figure 5-8 and Figure 5-9 show the main graph and three possible variations.

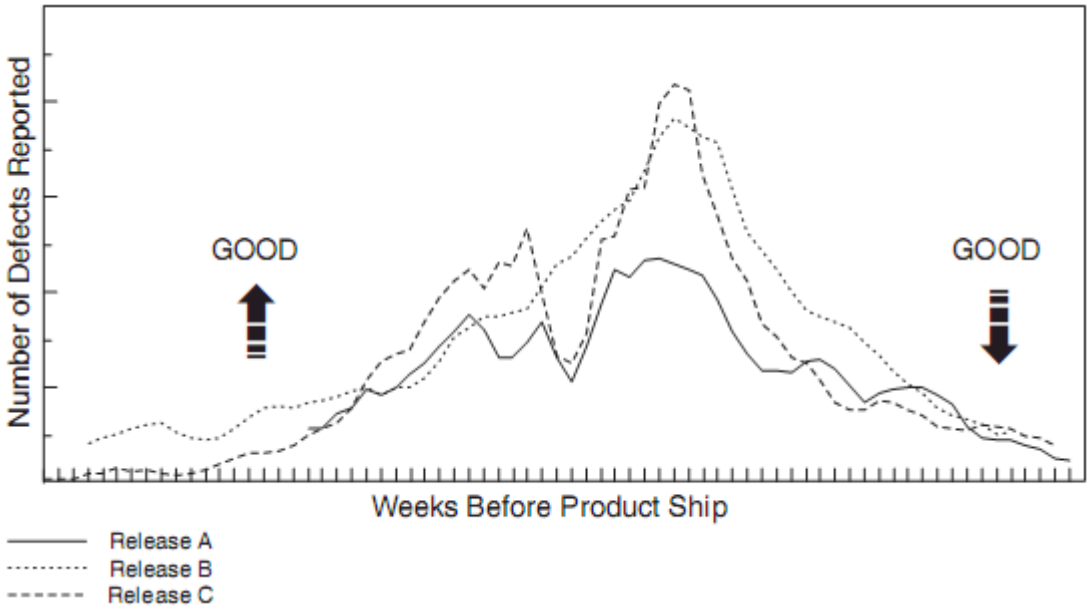


Figure 5-6 Sample Testing Defect Arrival Metric (Main) (Kan 2003)

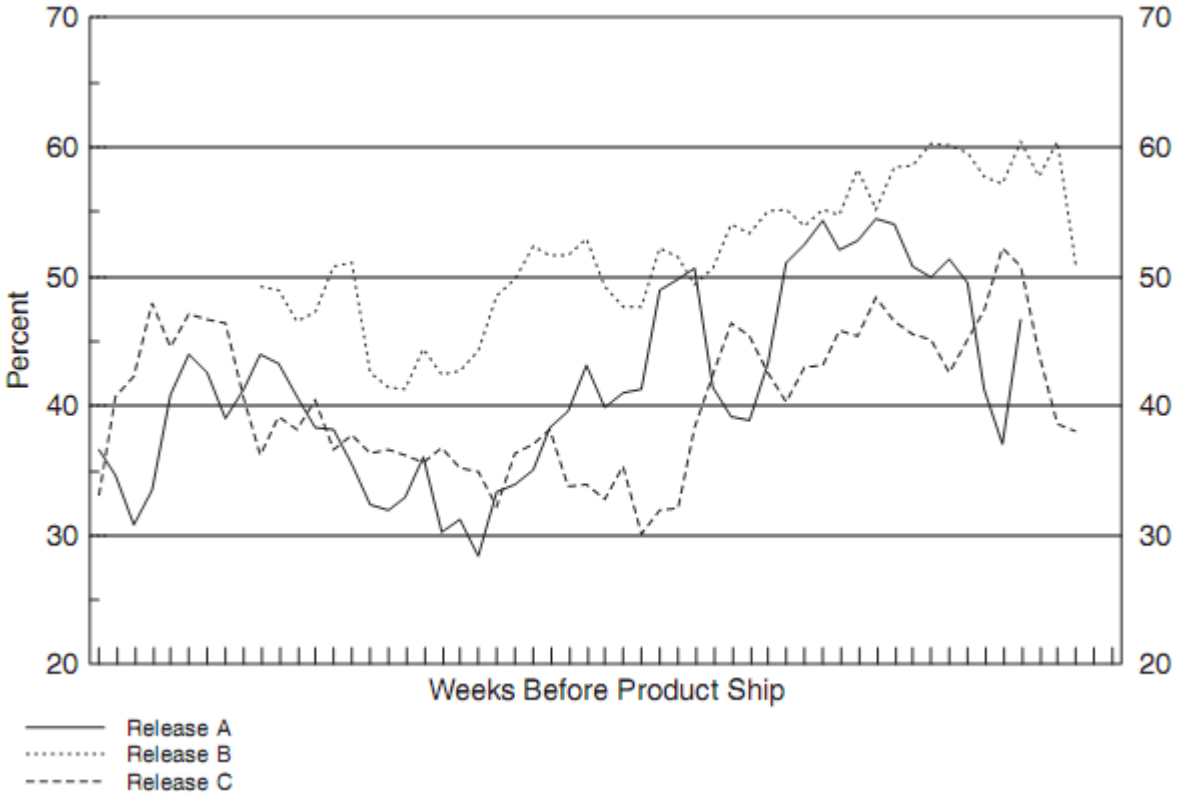


Figure 5-7 Sample Testing Defect Arrival Metric – Percentage of Severity 1 and 2 Defects (variation) (Kan 2003)

When transforming the weekly arrival curve (density form) to a cumulative form, the curve becomes a well-know form of the software reliability growth pattern. This graph can be used to estimate the latent defects between Product Ship Date and when the curve approaches its limit. See below for a visual representation.

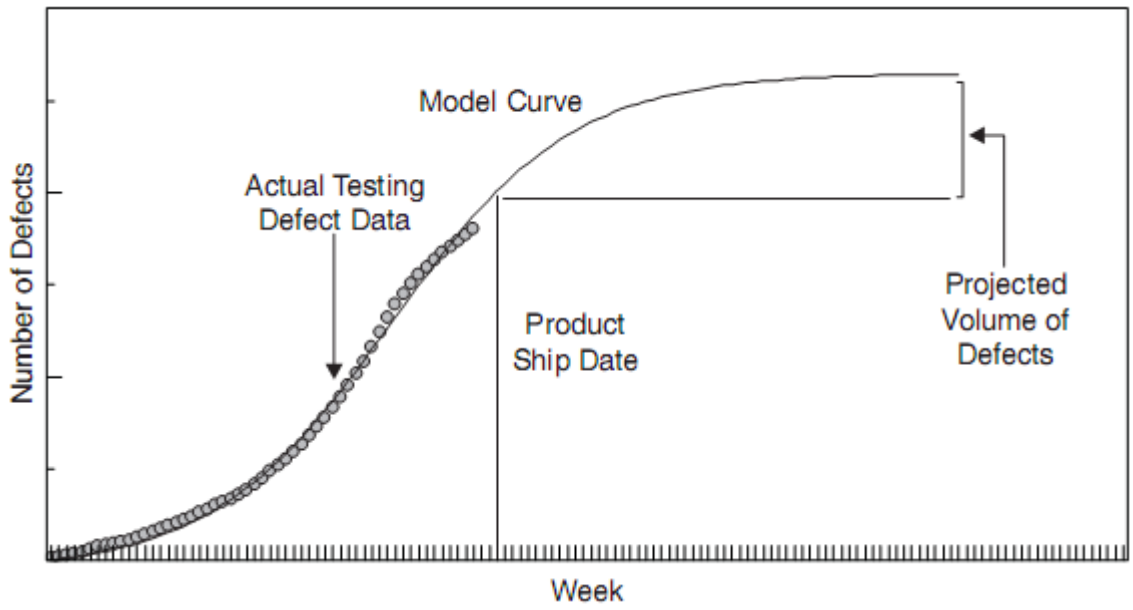


Figure 5-8 Sample 76

Testing Defect Arrival Curve, Software Reliability Growth Model, and Defect Projection (Kan 2003)

There are also authors that state this 'S' shape could very well be in another form, namely following a Weibull, Log-Logistic or Exponential distribution (example see: Figure 5-9 and for a comprehensive review of available functions see: Wood (1996)). So far the real shape is undecided in literature, or could very well be different at every development project. Gokhale and Trivedi (1999) state to use all possible distributions and determine which one best fits the development data (read: historical test coverage) of the software using goodness-of-fit, bias and bias trend indicators.

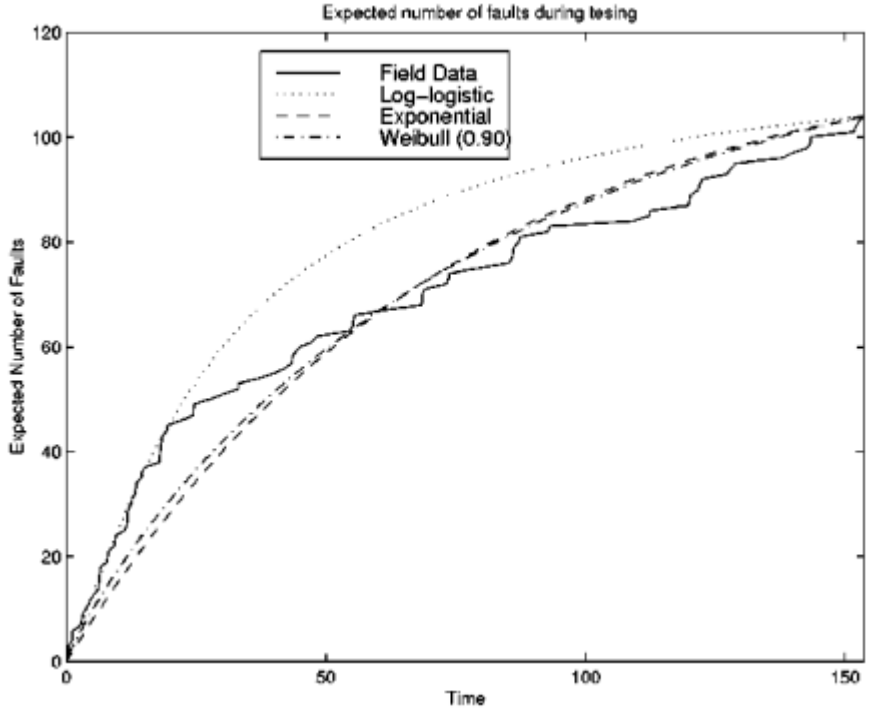


Figure 5-9 Sample Testing Defect Arrival Curves of other distributions (Gokhale and Trivedi 1999)

The use of incorporating historical coverage data in the prediction model is that to filter too optimistic predictions originating from inaccuracies in operational profile and the saturation effect of testing. Usage of an attribute of the code (here coverage) helps to counter effect this optimistic trend and thus improve prediction accuracy.

A critical note needs to be made however for the use of the bug-curve model. (Kaner and Bond 2004) Bug curve models are based upon several assumptions that are violated too often. On the next page the assumptions and their violations are listed.

- *Detection rate is proportional to current defect content*. Some defects are harder to find than others, also testers change testing techniques as the application in development becomes more stable, changing from easy tests to complex tests with many variables.
- *Defect detection rate remains constant*. When test techniques, staffing, or focus changes, the detection rate is likely altered.
- *Instant and always proper bug-fixes.* What would be the purpose of regression testing if this was the case?
- *All defects are equally likely to be encountered.* Of course fundamentally implausible. While some are almost impossible to miss, others occur only at borderline situations.
- *The application holds a fixed and finite number of defects at testing start.* This holds only in a utopian world, where no new bugs are introduced upon fixing others and when no code is added as soon as testing has started.
- *All defects are independent.* Defects often hide others.
- *The number of defects discovered in one interval is independent of detection numbers at others.* See: defect detection rate remains constant.

There are other questions to be placed at the validity of defect prediction models, comprehensive reviews of these prediction models including critique is found in Fenton and Neil (1999) and Wood (1996). For the coverage-based prediction model as shown above some promising results do have surfaced. See: Veevers and Marshall (1994) and Gokhale and Trivedi (1999) Please keep in mind that these are promising, yet inconclusive. The authors that report on its effectiveness stress this fact.

Overtaking the applicability discussion, another variation upon the previous metric could serve a different purpose. Uncovered defects can be represented by a histogram (here in pareto form) of total defects per origin or severity (pie chart is also a possibility). See Table 5-3 and Figure 5-10. This can provide useful insides in what area of testing results perform well and where they are slipping and thus need extra attention.

	Severity level				
Defect Origin	*1, %*	*2, %*	*3, %*	*4, %*	*Total, %*
Requirements	5	5	3	2	15
Design	3	22	10	5	40
Coding	2	10	10	8	30
Documentation	0	1	2	2	5
Bad fixes	0	2	5	3	10
Total Defects	10	40	30	20	100

Severity Legend	
Level	**Description**
1	System or program inoperable
2	Major functions disabled or incorrect
3	Minor functions disabled or incorrect
4	Superficial error

Table 5-3 Sample Defects per Origin and Severity (Jones 1997)

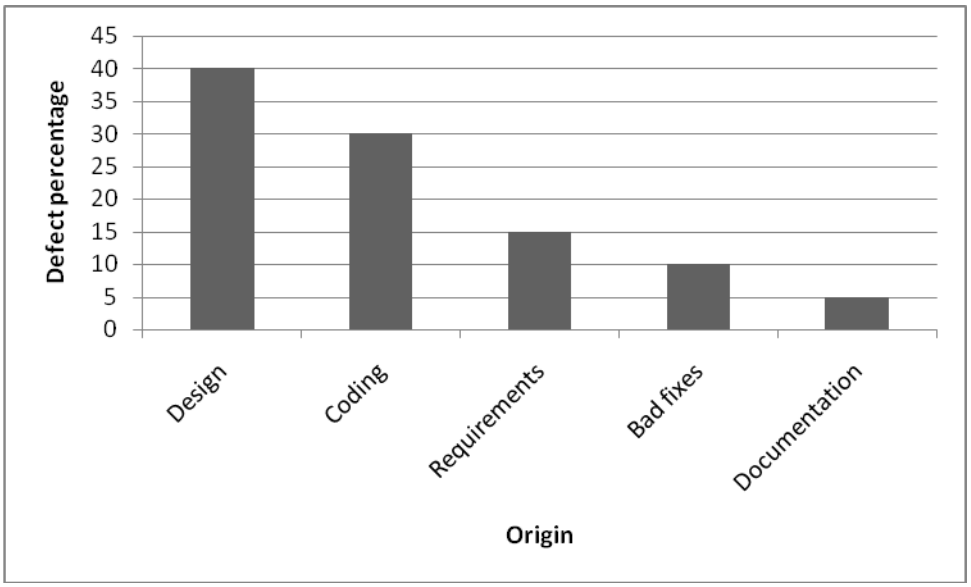


Figure 5-10 Sample Defects per origin Pareto (Jones 1997)

5.3.2.2 Testing Defect Backlog over Time

This metric tracks the defect backlog over time. It does this by crafting a graph out of the weekly number of open defects. (=arrivals – closed) Defect backlog tracking and management is important from the perspective of both test progress and customer rediscoveries. A large number of open defects during development will hinder test progress. Also when a release is shipped to a customer with a high defect backlog there increases the likelihood of rediscoveries of defects found in earlier development.

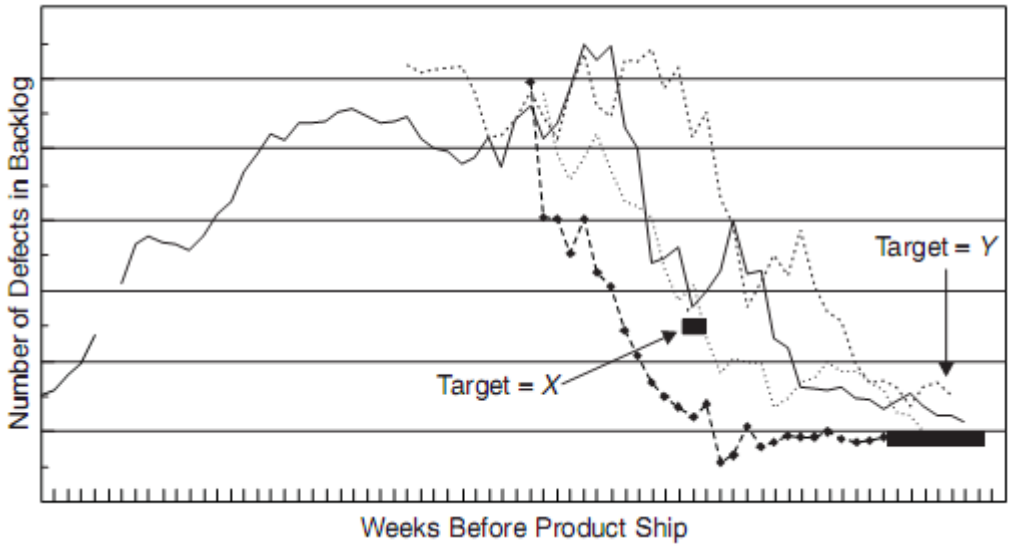


Figure 5-11 Testing Defect Backlog over Time (Kan 2003)

Purpose of this metric is to prevent high backlog especially when close to a release date. It is suited to set targets to help encourage developers to fix bugs and so reach low backlog.

N.B. A final note on defect arrival metrics is at order here. Three gaps in the measurement of these metric exist that need to be taken into account: (1) Private desk checking of defects by developers themselves, (2) defects found during private unit testing, (3) defects found during informal reviews by friends or colleagues. The result of these gaps is that the 'official' bug arrivals are always smaller than real arrivals. The exact effects of these gaps on the metric haven't been documented before and are unlikely to be uncovered in the future, due to (too) stringent measurement demands (endless defect reporting discipline) to fill these gaps.

5.3.3 Overall performance of testing

Previous metrics focused on measuring actual execution of tests and analyzing defect rates. The metric handled in this section shows whether or not current testing at SME is fit for the job of discovering defects.

5.3.3.1 Defect Detection Percentage

Wherever an application is developed, defects are inserted. The better the testing, the less bugs will remain latent till operation. Defect Detection Percentage (DDP) is a measure to show how testing has performed in a retrospective way. DDP is the percentage of the number of defects found in testing, divided by the number of total known defects. (Total known defects = defects found in testing + defects discovered afterwards) Of course a 100% score is best, where all defects were uncovered in testing, but this remains a utopian state.

Although this measure only provides valuable insights at the end of a development project (DDP will yield imbalanced results when applied when still very few defects are uncovered), it definitely has its use. It enables monitoring whether or not testing has performed over time and thus serves as a metric to see whether or not previously taken measures to improve testing have worked out or as a starting point to set new measures for the future, i.e. aiming for higher effectiveness.

This metric can also be customized to provide deeper insights in detection effectiveness, by measuring DDP per testing type (i.e. the seven testing types of section 4.2.1). However be wary of the implications this has for data collection, for every defect has to be properly appointed to a defect detection stage where it should have been caught, and often this is particularly hard to do due to defect complexity.

5.3.4 Special case: When to stop testing?

A special case of importance for SME concerning metrics is to know when to stop testing, for less testing leaves too much hidden bugs, and too much testing is costly and less effective. An optimal breakpoint thus is needed. There are several ways to decide when this breakpoint – the desired level of quality / testing – has occurred. Some are currently in use, but these should be supplemented to have an improved estimate on when to stop testing. Above two of these supplements have been mentioned, code coverage and test progress curve, but there are others. The complete list of stop metrics:

Stop metric	Description
Agreement-based	Development team and customer agree on what to test and on quantities
Effort	Fixed amount of time / tests
Coverage	Minimal percentage of functions covered in tests
Project-history based	Estimate future by looking at the past
Risk-based	Parts of the application are awarded a risk and so high risk is tested more intensively than low risk

Table 5-4 Stop metrics

N.B. These extensive stop criterions are excluded here due to parallel research and primary project usage by a new tester at SME. Also this would imply digging in too deep into metrics instead of improving software quality (this research's goal). Please refer to Kaner (1996) for an extensive list of metrics per category.

A final practitioners note for improvement of the 'when to stop' trade-off is to apply a mixture of stop metrics. The currently observed limited Agreement-based approach – agreement on who does what, but lacking in what quantities of testing need to be performed – that comes down a fixed effort test schedule should be supplemented. Supplement the trade-off decision with minimal test coverage requirements per component and increase coverage and test cases at high risk application parts. This should form a base for testing towards becoming a true development phase instead of the mere shell thereof currently in effect. (Kaner 1996) concurs on applying a mixture of stop-metrics to obtain best trade-off results.

5.4 Setting a testing atmosphere

This section is about setting the right atmosphere that nurtures testing. While the two principles described in the following underlying sections aren't testing best-practices, they do hold some valuable intelligence concerning change management that belongs in this report. They are (or should be) valued in any agile development environment. The two principles, customer involvement at section 5.4.1 is about seeing customers as team players instead of contract negotiators and at section 5.4.2 several ways to speed up testing skill distribution amongst developers are mentioned.

N.B. this section is held brief, for a literature research into change management would be required to cover all aspects of setting a changing process atmosphere. This research targeted the assessment of the current situation and sketches a desired future state, the transition is considered beyond content.[19]

5.4.1 Customers: high(er) involvement

All agile development methodologies take customers involvement to another level. They've changed customers' interaction. Instead of negotiating over contracts, the focus is shifted towards development collaboration. Direct communication is marked of highest priority. That's needed, for customers have more power in what features will be available next release and are required to perform more acceptance testing. The agile development practice SCRUM holds story/feature meetings and daily stand-ups where customers should be represented. Please refer to Schwaber (1995) for a more detailed overview of its practices and implications for development.

5.4.2 Developers: diffuse knowledge

In order to get both developers and customers up to speed on testing, extra attention is required. Learning from each other needs to be set as a key value. But that isn't enough; to get real results continuously reinforcing this principle during development activities is required.

A couple of measures to support this undertaking have been selected (but a lot more are available):

Have test templates available for developers less experienced in testing, to have a quick start kit from which they can expand to their own templates. These templates then again need to be shared and reviewed with others and so an evolving pattern of templates will start to emerge.

Hold workshops promoting and informing test types and supporting tools available. This is an easy way to let developer familiarize themselves with testing in a Greenfields setting. An open dialogue will also be able to take away any remaining reluctance to test usage or knowledge lacks.

Apply – or at least start with – as pair-wise test writing. A noticeable quote of agility guru Scott Ambler (2006) shows the core message clearly: "Pair testing, just like pair programming and modeling with others, is an exceptionally good idea. My general philosophy is that software development is a lot like swimming – it's very dangerous to do it alone."

[19] For interested readers on making this transition, search literature using keywords 'Software Process Improvement / SPI' and 'Change Management' or contact the author for a quick start list.

6 DISCUSSION

This chapter is split into the traditional sections Conclusion, Recommendations and Limitations/Future Work. At section 6.1 findings on the current testing situation are summarized. Section 6.2 holds the recommendations for a future SME in testing. In section 6.3 the best-practices are found to cover all required aspects, solve listed testing issues and improve overall software quality. Lastly section 6.4 covers exit clauses together with future research possibilities and interests.

6.1 Conclusion: testing severely underexposed

Testing proved to be severely under applied at SME. The outcome / effort model shows SME scoring a 'Unsure' or 'Worst-Case' score, which denotes inferior performance. The interviews and surveys back the lack of test types and underlying principles application, like automation and regression. The respondents of the held survey furthermore rank testing as inadequate for the task by grading testing activities a 5.2 on average and 71% scoring testing as falling short. The high variety of issues as gathered during this research' formulation were also proven to be present. After examining the testing process in more detail, it was uncovered that there is hardly a testing process in effect, except for some functional and acceptance testing. All this demarks a gap towards application of state-of-art testing procedures and correlating software quality. Luckily this was expected somewhat at SME, which makes these notions somewhat less harsh.

Findings show that there's much to be gained easily, a handful simple metrics can provide project managers with the power to monitor and steer upon proper testing. Combined with the available benefits of TDD and CIT, this will allow SME's software development process to improve. The transition towards enhanced testing will take time and effort, but fortunately all consulted SME developers and analysts agree that the current standing is inadequate and they are willing to change. This healthy open and change willing culture will provide a sound basis for implementation of the recommendations. Please refer to section 6.2.2 for a short overview of actions that can be taken to aim for better software trough better testing. But please keep in mind this research has devoted little attention to implementation trajectory, for the follow up master's thesis research will cover the actual implementation of best practices.

6.2 Recommendations

The underlying sections below cover two aspects of best-practice adoption. The first – section 6.2.1 – explains that the identified best-practices shouldn't be regarded as alternatives, but more as subsequent improvements to be implemented. The route to follow is (1) Metrics, (2) TDD and (3) CIT. The second – section 6.2.2 – argues for the best-practice implementation path best followed as regarded by the author of this research.

6.2.1 Best-practice adoption, not which but in what order

This section should – according to the research questioning – cover what best-practice(s) to adopt, but instead this question is rephrased to "In what order should the identified best-practice(s) be adopted?" This is because the original research question was found to be obsolete. This is due to two reasons: (1) All best-practices were shown to be useful / high on potential benefits, are relatively easy to apprehend, have countless empirical examples, and their agile background should fit SME's mentality fine. (2) Jones (2000) states there are no coincidences in the order of Software Process Improvement (SPI)[20] practices that are implemented. As his extensive benchmarks amongst 500 U.S. businesses prove, there's a fixed order in which SPI is performed:

Stage	Activities
0	Software process assessment, baseline, and benchmark
1	Focus on management technologies
2	Focus on software processes and methodologies
3	Focus on new tools and approaches
4	Focus on infrastructure and specialization
5	Focus on reusibility
6	Focus on industry leadership

Table 6-1 7 stages of software process improvement (Jones 2000)

Jones (2000) identifies several stages of SPI where each stage holds its own focus activities. The first stage is an analysis stage to determine current standings of development quality. The following stages are therapeutic and aimed at curing weaknesses found at stage 0. The stages that are of current matter to SME (0-3) will be elaborated upon. Stage 0 is primarily a diagnostic phase that identifies strengths and weaknesses. Stage 1 focuses on supporting management issues of justification in investment on SPI with tools and training, only when management is up-to-date in calculating ROI of future process improvements the following phases can/will be executed. Stage 2 involve introduction of specific process methodologies. An example here is the requiring of formal design and code inspections. Stage 3 calls for heavy investment in time and people for using new tool suites.

The current performance analysis in this research sticks to stage 0, which holds a diagnostic phase that identified baseline performance and made a process assessment (benchmark comparing SME to other businesses wasn't applied). Recommendations for best-practices of this research fall under stages 2 and 3. Stage 1 is missed somewhat due to being out-of-focus with this research' goals. However some metrics are introduced improve project managers' grip on the software quality process.

Stages 4 through 6 are still out of reach for current research as well as SME practices, for they require lower levels to be present and able that SME still lacks.

Following Jones' (2000) logic, the only legitimate order to rollout the aforementioned best-practices would be: (1) metrics as a management technology enabler (stage 1), (2) TDD as a development methodology (stage 2), and finally (3) CIT as methodology towards tooling (stage 3).

[20] Continuous and iterative improvement of software development practices

6.2.2 Stepwise improvement via three consecutive best-practices

To get SME to an acceptable level of testing, the following steps need to be taken to enable a test fostering development process.

Start by setting up measurements of current defect and test execution performance. Parallel to that set targets for test levels and appoint responsibility to reaching these. Switch from individual to team responsibility for the code base and adjoining tests. This should make testing efforts more visible, controllable and avoids personal pitfalls of working under high pressure. Next thereto developers will learn from each other by looking into each other's tests. (Currently a first-start template package for developers is under development by a test group)

When this is starting to make sense, switch towards UTDD, aiming for high levels of unit and functional tests. Use pairing as a fast way to diffuse test knowledge (full pairing is out of the question for now when discussing the subject with project managers), or arrange formal test reviews to the least.

When developers start to feel as testers and embrace testing everything they code, attach automated test suites to the build and version management servers (CIT principles). Let the developers only check in code that is accompanied by tests. These tests run along with every build, showing (integration) defects as soon as they're introduced. By the time this is becoming reality the amount of defects missed during development should have been greatly reduced. Not only will this save countless hours in debugging, but also the costs of delivering too late, and extra personnel hours to patch things up again.

Finally, after internal testing levels are found to be acceptable, shift testing focus towards the customer. Instead of the current strictly separated responsibility of acceptance testing, which lies completely at the customer, aim for collaboration instead of mitigating responsibility (and thus risk). This collaboration could very well take place by applying ATDD. This enables earlier and more accurate acceptance testing.

6.3 Issues and main question revisited: improvements throughout

This section serves as a reflection upon this research to see if testing issues and the research questions are solved to satisfactory levels. This is the case in both following reflections. First the issues will be revisited at section 6.3.1 where after a positive reflection upon the main research question will follow at section 6.3.2.

6.3.1 All issues covered

Table 6-2 shows the effects of the best-practices on the test issues of SME. As the table verifies, all issues are adequately covered by the recommendations of this report.

Issues	Best-Practices	Effects
Uncertainty about software quality	XP's TDD	Writing tests for every code piece will feel like assured quality at developers, project managers and customers
	CIT	Green lights of the test suite feedback at every build shows development is on track
	Metrics	By keeping track of various quality indicators, quality can be proven
Software quality can't be proven	XP's TDD	See previous issue commentary
	CIT	See previous issue commentary
	Metrics	See previous issue commentary
Software defects (or bugs) are uncovered too late in the development process	XP's TDD	More tests raise odds of defects being uncovered early, especially tests being written upfront speed detection
	CIT	Integration and regression tests combined with a central testing repository will show reintroduced bugs or malfunctioning other parts due to check-ins immediately
	Metrics	Metrics show defect discovery trends, which indirectly aid detection due to (corrective) measures being taken
Testing activities cripple under new-feature pressure	XP's TDD	Testing is demanded upfront of coding, so testing is guaranteed to persist
	CIT	At worst case scenario existing auto regression tests will run
	Metrics	Metrics lay a base for performance targets, which will include required testing levels. This can function as a safety net for testing to occur
Unknown testing effort	XP's TDD	Everyone writes tests for every code, so there is at least a major effort
	CIT	The central code repository holds tests as well, automated reports show exactly what tests run and how they score
	Metrics	Trends are visible throughout, including a metric for test effort
Lack of vision on testing	XP's TDD	TDD goes beyond vision; it is a new testing paradigm.
	CIT	Automation, centrality and regression are new values.
	Metrics	No direct effect
Lack of testing responsibility	XP's TDD	All code is forced to have an accompanying unit/functional test. Individuality is swapped for team effort, increasing cohesion to responsibilities.
	CIT	Central repository makes for team effort in testing, instead of individuality while at the same time showing exactly whether or not code has adjoin tests and thus serves as a way to reflect upon responsibility to write tests.
	Metrics	Metrics can be used for target test levels verification, thus reflecting upon test responsibilities.

Table 6-2 Effects of best-practices on current testing issues

6.3.2 Software quality sure to improve

Now it's time to revisit the main research question of this research, to see whether or not it's been solved:

*'How is the **testing process** to be improved to raise **software quality**?'*

It's safe to say the *how* has been answered thoroughly by showing gaps in current test performance and providing three best-practices to counter these shortfalls.

As for the reflection upon the foretold rise of software quality: Software quality was defined earlier (see: section 1.3.2) as delivering software with less defects. Kan (2003) has published a clear view upon the Rayleigh Model. This model shows the distribution of defects over time and what forces can reduce the defect rate. Two forces improve software quality: (1) reduce error injection and (2) early defect removal, which is a combination of discovery rate and removal efficiency improvements.

The typical Rayleigh Model Curve of software development is depicted below:

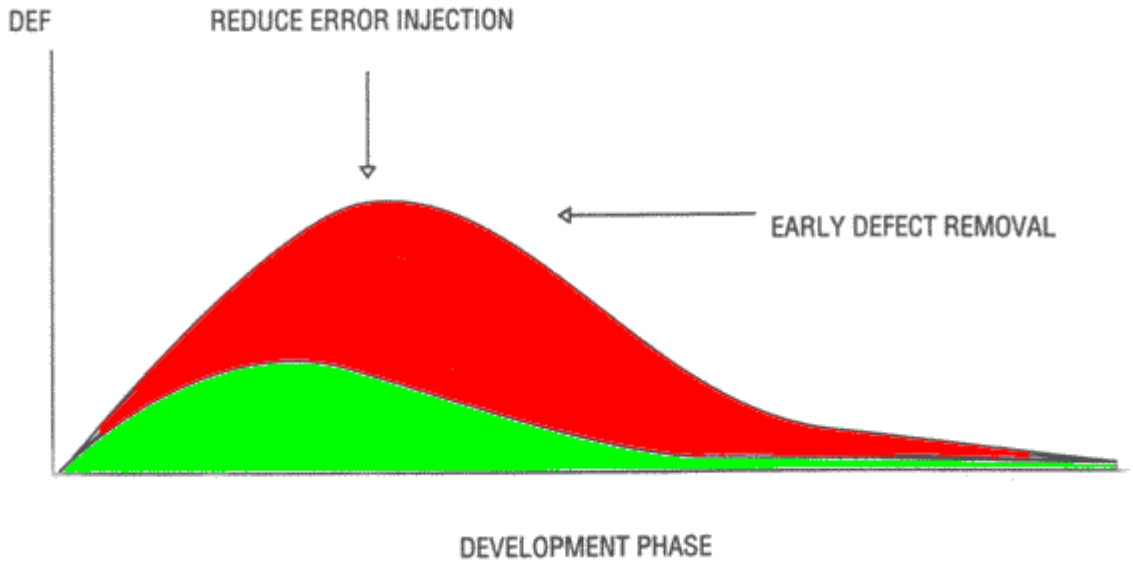


Figure 6-1 Sample Rayleigh Model (Kan 2003) (adapted)

The figure above shows a sample of outstanding defects over time in a typical development process. The **red** area depicts a **buggy** process, while the **green** and smaller area signifies an **improved** process with reduced total injection rates (top curve green << top curve red) and earlier defect removal (decline of outstanding defects occurs earlier in the development process at green over red). Together these forces amount to reduced total defects (area green << area red).

So, to reflect upon raising software quality these two forces need to be checked for occurrence in the best-practices. See Table 6-3 below. Both forces are adequately covered, so it is safe to say the main question's contents have been handled in this research.

Defect Rate Improvement Forces	Best-Practices	Effects
Reduce Error Injection	XP's TDD	By demanding tests upfront, developers are forced to think harder and earlier about the proper implementation of features; less error-prone debugging required
	CIT	Reduces barriers development ←→ QA[21], synergies in test and development knowledge occur
	Metrics	No direct effect
Early Defect Removal (Discovery Rate + Removal Efficiency)	XP's TDD	TDD shortens feedback cycles and greatly enhanced test quantities by demanding tests upfront for all code
	CIT	Integration and regression principles ensure that defects are uncovered immediately when defects are (re)introduced; shortened feedback-cycles; previously manually indiscoverable bugs are found with automation; auto-entry of defects in trackers improve removal reaction speed
	Metrics	While not so much helping directly, the trends originating from metrics do show how the defect discovery and removal rates behave and can form a reason for intervention

Table 6-3 Effects of best-practices on software quality (here: defect rate)

6.4 Limitations and future work

This section holds two underlying sections. The first entails the limitations to research conclusions while the second shows proposes and promising future research possibilities.

6.4.1 Limitations

Several limitations hold for (the conclusions of) this research, these are enumerated below:

- The research was targeting the development process and mostly at the testing phase(s) thereof, but not from a total software quality view. In this same context, pre- (and post somewhat) testing defect injection reduction and removal activities aren't included.
- Functioning of the proposed best-practices in reality remains partly unknown, although the identified best-practices should increase test performance.
- The research only covers lean / agile methodologies and supporting tools, traditional frameworks were excluded.
- This research only covers immediately applicable and needed actions, further steps in the Software Process Improvement trajectory haven't been analyzed, at maximum mentioned.
- While this research shows best practices that should improve software quality, effort and monetary implications for their adoption is deemed out of scope. Before adopting (one of these) best-practices a full business case needs to be performed, carefully weighing costs against benefits.

[21] Quality Assurance, or in this case: testing.

6.4.2 Future work

This research covered a wide range of testing aspects and solutions. For the immediate future several possible activities have been identified:

This research has identified several benefits that could (or should!) improve software quality. But before putting these to the test, an objective and thus quantified measurement of software quality is needed to see improvements arising from the package adoption. This implies performing a *comprehensive baseline measurement* with the identified new to implement metrics. When this baseline has been established, rollout of best-practices may commence. During that time measurements on these metrics need to remain intact.

Second recommended future practice is *incorporating more phases of software development* into the baseline measurement and following SPI , for software quality is not only dependent on testing quality but also on other development phases. In this light testing should be viewed as part of defect removal operations, proper requirements and design inspections will net high removal rates (higher than those in the tests covered in this research) All tests applied in together and in full effect max out at 70% defect discovery. (Jones 2000) The following table depicts possibly to be researched practices beyond testing that can improve software quality:

Technique	**Author(s)**
Defect Prevention	
Examination of constraints	(Dustin, Rashka et al. 1999)
Prototypes	(Jones 2000)
Early test involvement	(Dustin, Rashka et al. 1999)
Use of process standards	(Dustin, Rashka et al. 1999)
Inspections and walkthroughs	(Dustin, Rashka et al. 1999; Jones 2000)
Quality Gates	(Dustin, Rashka et al. 1999)
Defect Detection	
Inspections and walkthroughs	(Dustin, Rashka et al. 1999; Jones 2000)
Usability labs	(Jones 2000)
Quality Gates	(Dustin, Rashka et al. 1999)
Testing of product deliverables	(Dustin, Rashka et al. 1999)
Designing for testability	(Dustin, Rashka et al. 1999)
Use of automated test tools	(Dustin, Rashka et al. 1999)
Unit testing	(Dustin, Rashka et al. 1999; Jones 2000)
Integration testing	(Dustin, Rashka et al. 1999; Jones 2000)
System testing	(Dustin, Rashka et al. 1999; Jones 2000)
Functional testing	(Jones 2000)
Acceptance testing	(Dustin, Rashka et al. 1999; Jones 2000)
Following defined test process	(Dustin, Rashka et al. 1999)
Risk assessment	(Dustin, Rashka et al. 1999)
Strategic manual and automated test design	(Dustin, Rashka et al. 1999)
Execution and management of automated tests	(Dustin, Rashka et al. 1999)
Test verification method	(Dustin, Rashka et al. 1999)
User involvement	(Dustin, Rashka et al. 1999)

Table 6-4 Software defect prevention and detection techniques (Jones 2000) (Dustin, Rashka et al. 1999)

The fact that this research only (some overlap to prevention however) covers the second part 'Defect Detection' is evident from the table. To discover and solve even higher ratios of defects, the first column needs to be taken into account as well. A note about efficient defect prevention and removal methods per defect origin is in place here as well. See the table below:

	Requirements Defects	Design Defects	Code Defects	Document Defects	Performance Defects
Reviews/ Inspections	Fair	**EXCELLENT**	**EXCELLENT**	**GOOD**	Fair
Prototypes	**GOOD**	Fair	Fair	Not Applicable	Good
Testing (all forms)	Poor	Poor	Good	Fair	**EXCELLENT**
Correctness Proofs	Poor	Poor	Good	Fair	Poor

Table 6-5 Defect removal methods (Jones 1997)

Table 6-5 shows effectiveness of the different defect detection and removal methods. Best in class effectiveness for a type of defect is marked by using CAPS and **underline**. Testing is awarded color labels for its effectiveness. **Red** indicating poor to **green** indicating excellent performance. The table clearly shows Reviews/Inspections as a useful defect removal practice next to Testing. Prototypes are also quite useful for discovering requirement defects. These practices should thus be examined for use at SME in a later research. Correctness proofs is handled later, for it delivers effectiveness values equal or below Testing, but at the other hand they are easy to implement, for they incorporate static testing (which do not even execute code) and can be automated almost in full. This holds many possible fully automated test tools like FxCop and Findbugs and thus can easily extend the proposed CIT best-practice implementation to attain quickly realizable benefits.

Final recommendation for future work is to *apply a broader definition of software quality* to steer processes, not just to minimize defects. Also take into account the productivity ratio, for quality always balances with costs. During this research several out-of-scope development process issues were encountered that were out-of-bounds, but certainly worthy of further examining.

REFERENCES

Abrahamsson, P., O. Salo, et al. (2002). "Agile software development methods: Review and Analysis." Technical Research Centre of Finland, VTT Publications 478, from http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf.

Abrahamsson, P., J. Warsta, et al. (2003). New Directions on Agile Methods: A Comparative Analysis. Proceedings of the Twenty-Fifth International Conference on Software Engineering**:** 244-254.

Ambler, S. W. (2002). Agile Modeling: Best Practices for the Unified Process and Extreme Programming. New York, John Wiley & Sons.

Ambler, S. W. (2006). "Agile Testing Strategies." Architecture & Design, from http://www.ddj.com/architect/196603549.

Bach, J. (1997). Test automation snake oil. 14th International Conference on Testing Computer Software. Washington D.C., US Professional Development Institute.

Bach, J. (1998). "A framework for good enough testing." Computer **31**(10): 124-126.

Baskerville, R., L. Levine, et al. (2001). "How Internet companies negotiate quality." Computer **34**(5): 51-57.

Baskerville, R. and J. Pries-Heje (2001). Racing the E-bomb: How the Internet is redefining information systems development methodology. Realigning research and practice in IS development. B. Fitzgerald, N. Russo and J. DeGross. New York, Kluwer**:** 49-68.

Beck, K. (1999a). "Embracing change with extreme programming." Computer **32**(10): 70-77.

Beck, K. (1999b). Extreme Programming Explained. Reading, MA, Addison-Wesley.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. Reading, Addison-Wesley.

Beck, K. (2002). Test Driven Development: By Example. Reading, Addison-Wesley.

Boehm, B. W. (1988). "A Spiral Model of Software-Development and Enhancement." Computer **21**(5): 61-72.

Coad, P., D. L. J., et al. (1999). Java Modeling in Color. Englewood Cliffs, NJ, Prentice Hall.

Cockburn, A. (1998). Surviving Object-Oriented Projects: A Manager's Guide. Longman, Addison-Wesley.

Cockburn, A. (2000). Writing Effective Use Cases, The Crystal Collection for Software Professionals, Addison-Wesley Professional.

Cockburn, A. (2002). Agile Software Development. Boston, Addison-Wesley.

Cockburn, A. and J. Highsmith (2001). "Agile software development: The people factor." Computer **34**(11): 131-133.

Cockburn, A. and L. Williams (2000). The costs and benefits of pair programming. Proceedings of eXreme Programming and Flexible Processes in Software Engineering. Cagliary**:** 223-243.

Compuware. (2006). "Continuous Integrated Testing: Delivering Apps With Confidence." Java Retrieved 01-03-2008, 2008, from http://www.ddj.com/java/196601755.

Crispin, L. and T. House (2003). Testing Extreme Programming. Boston, MA, Addison-Wesley.

Cusumano, M. A. and D. B. Yoffie (1999). "Software development on Internet time." Computer **32**(10): 60-69.

DSDMConsortium (1997). Dynamic Systems Development Method, version 3. Ashford, DSDM Consortium.

Dustin, E., J. Rashka, et al. (1999). Automated Software Testing. Reading, Addison-Wesley.

Duvall, P. M. (2007). Continuous Integration: Improving Software Quality and Reducing Risk, Addison Wesley.

Erdogmus, H., M. Morisio, et al. (2005). "On the effectiveness of the test-first approach to programming." Ieee Transactions on Software Engineering **31**(3): 226-237.

Fenton, N. E. and M. Neil (1999). "A critique of software defect prediction models." Ieee Transactions on Software Engineering **25**(5): 675-689.

Fewster, M. and D. Graham (1999). Software test automation : effective use of test execution tools. Harlow, Addison-Wesley.

Flowers, J. (2006). "A recipe for build maintainability and reusability." from http://jayflowers.com/joomla/index.php?option=com_content&task=view&id=26.

Fowler, M. (2006). "Continuous Integration." from http://martinfowler.com/articles/continuousIntegration.html.

Fowler, M. and J. Highsmith (2001). "The Agile Manifesto." Software Development: 28-32.

Fowler, P. and S. Rifkin (1990). Software Engineering Process Group Guide. CMU/SEI-90-TR-24, Carnegie Mellon University.

Gokhale, S. S. and K. S. Trivedi (1999). "A time/structure based software reliability model." Annals of Software Engineering **8**(1-4): 85-121.

Grenning, J. (2001). "Launcing Extreme Programming at a Process-Intensive Company." Ieee Software **18**(3): 3-9.

Haines, S. (2008). "Continuous Integration and Performance Testing." Java.

Highsmith, J. (2000). Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York, Dorset House Publishing.

Hunt, A. and D. Thomas (2000). The Pragmatic Programmer, Addison-Wesley.

Janzen, D. and H. Saiedian (2005). "Test-driven development: Concepts, taxonomy, and future direction." Computer **38**(9): 43-50.

Jeffries, R. (1999). "eXtreme testing." Software Testing & Quality Engineering **1**(2): 23-26.

Jones, C. (1997). Applied software measurement : assuring productivity and quality. 2nd ed. New York, McGraw-Hill.

Jones, C. (2000). Software Assessments, Benchmarks, and Best Practices. Boston, Addison-Wesley.

Kan, S. H. (2003). Metrics and models in software quality engineering. 2nd ed. Boston, Addison-Wesley.

Kan, S. H., V. R. Basili, et al. (1994). "Software Quality - An Overview From The Perspective of Total Quality Management " IBM Systems Journal **33**(1): 4-19.

Kaner, C. (1996). Negotiating testing resources: A collaborative approach. Ninth International Software Quality Week Conference. San Fransisco.

Kaner, C. and W. P. Bond (2004). Software Engineering Metrics: What Do They Measure and How Do We Know? 10th EEE International Software Metrics Symposium.

Koskela, L. (2008). Test driven : practical TDD and acceptance TDD for Java developers. Greenwich, Manning.

Lindstrom, L. and R. Jeffries (2004). "Extreme programming and agile software development methodologies." Information Systems Management **21**(3): 41-52.

Lindvall, M., V. R. Basili, et al. (2002). "Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods." Lecture Notes in Computer Science **2418**: 197-207.

Marick, B. (1997). Classic Testing Mistakes. Sixth International Conference on Software Testing, Analysis, and Review.

Maximilien, E. M. and L. Williams (2003). Assessing test-driven development at IBM. Proceedings - International Conference on Software Engineering.

McFeeley, B. (1996). IDEAL SM: A User's Guide for Software Process Improvement. Handbook CMU/SEI- 96-HB-001. Pittsburgh, PE, Software Engineering Institute.

McMahon, J. (2003). "5 lessons from transitioning to eXtreme programming." Control Engineering **50**(3): 59-65.

Nerur, S., R. Mahapatra, et al. (2005). "Challenges of Migrating to Agile Methodologies." Communications of the ACM **48**(5): 73-78.

Palmer, S. R. and J. M. Felsing (2002). A Practical Guide to Feature-Driven Development. Upper Saddle River, Prentice-Hall.

Rainer, A. and T. Hall (2002). "Key success factors for implementing software process improvement: a maturity-based analysis." Journal of Systems and Software **62**(2): 71-84.

Schwaber, C. and R. Fichera. (2005). "Corporate IT leads the second wave of agile adoption." from http://www.forrester.com/Research/Document/Excerpt/0,7211,38334,00.html.

Schwaber, K. (1995). SCRUM Development Process. OOPSLA'95 Workshop on Business Object Design and Implementation, Springer-Verlag.

Schwaber, K. and M. Beedle (2002). Agile Software Development with Scrum. Upper Saddle River, NJ, Prentice-Hall.

Schwartz, R. B. and M. C. Russo (2004). "How to quickly find articles in the top IS journals." Communications of the Acm **47**(2): 98-101.

Sogeti (2008). TMAP NEXT: Overzicht Toegepaste Testvormen. Vianen, Netherlands, Sogeti Netherlands.

Stapleton, J. (1997). Dynamic systems development method - The method in practice, Addison-Wesley.

van Solingen, R. (2004). "Measuring the ROI of software process improvement." Ieee Software **21**(3): 32-+.

Veevers, A. and A. C. Marshall (1994). "A Relationship between Software Coverage Metrics and Reliability." Journal of Software Testing, Verification and Reliability **4**: 3-8.

Williams, L., E. M. Maximilien, et al. (2003). Test-Driven Development as a Defect-Reduction Practice. Ieee International Symposium on Software Reliability Engineering. Denver**:** 34-45.

Wood, A. (1996). "Predicting software reliability." Computer **29**(11): 69-77.

APPENDICES

A Semi-open interview format

This Appendix covers the format used at the semi-open Interviews. The original format is written in Dutch, but was translated to English to suit this report. Questions are dived into four categories: (1) interviewee characteristics/identifiers, (2) test process descriptives and (3) test process performance.

Employee personal details
Name
Function
Business unit
Years employed
Years in software development

Current testing process descriptives
Which (partial) activities are involved at testing?
How are activities divided into sub-activities i.e. over employees?
What sorts of tests are performed?
Which methods & tools are used to aid these tests?
What goes well in testing?
What can be improved?
What would be your ideal testing method?
What do you miss in testing?
What problems occur in testing?

Current testing process performance
What performance factors are being measured?
Do you miss any performance factors?
How does the collection of testing performance data take place?
What difficulties occur during measurement?
How was/where the current measurement process/factors realized?
And Why?
What is current performance on these (or to be developed) factors?

B Survey format

This Appendix lists the format for the used online survey 'Testing at SME'. The original is in Dutch, but was translated to English to suit this report.

N.B. questions 6 – 16 held a text field for open comments, but where removed here for typographic reasons.

Testing at SME

Introduction

Dear SME-ist,

Welcome to my survey on testing at SME.

This survey tries to determine how testing at SME performs and tries to determine where improvements can / should be made to make SME' software even better.
I expect the survey to take about 15 minutes of your time.
At every question there's a comment box, so if you need to state something specific about a particular question, please fill this in.
Finally: All questions must be filled out as being applicable to the SME company (eg: Hub) at which you work.

Questions / comments / etc. do not hesitate in emailing me! (yoni.meijberg@SME.nl)
With your help SME will improve for sure!

Success in completing the questionnaire and thanks in advance.

Yoni

Identification

1. At which 'SME' do you work?

   ○ Hub ◉ Energy ○ Automobile ○ Transport

2. What is your (main) function?
   (Note if you do not belong to any of these functions, then this survey is not suitable for you and I ask you to close this survey.)

   ○ Developer Junior - ○ Developer Senior - ○ Analyst Junior - ○ Analyst Senior - ○ Project Manager

3. How many years are you employed by SME? (rounded up to half years with 1 decimal; separated by one point. example: 2.5)

Year(s)

Test types, regression & automation

4. How many weeks does the average application release span?

   Weeks

5. How often are the following tests applied?
   (Please keep the release span of question 3 to mind)

	Daily	Weekly	Monthly	Per release	(Almost) Never
Unit Tests the smallest possible software component or module. Every unit of the software is tested to verify whether the detailed unit design has been implemented correctly.	◉	○	○	○	○
Integration Checks for errors in interfaces + interactions between integrated components. Progressively increasing components of the architectural design are integrated and tested until the software functions as a system.	○	○	○	○	○
System Tests a completely integrated system, to verify whether it meets demands.	○	○	○	○	○
System Integration Checks whether the system is properly integrated with external or third party systems.	○	○	○	○	○
Functional Tests at every level (class, module, interface or system) for functionality as stated in requirements specifications.	○	○	○	○	○
Acceptance Testing by end user / customer / analyst to verify whether or not a release is accepted for production use.	○	○	○	○	○

6. How important are previously mentioned tests for project success?

	Completely unimportant	Quite unimportant	Neutral	Quite important	Very important
Unit	○	○	○	○	○

Integration					
System					
System Integration					
Functional					
Acceptance					

7. To what extent is regression (after a code change rerun all earlier designed tests) applied at previously mentioned tests?

	Per check-in	Daily	Weekly	Monthly	Per release	(Almost) Never
Unit						
Integration						
System						
System Integration						
Functional						
Acceptance						

8. How important is regression to previously mentioned testing for project success?

	Completely unimportant	Quite unimportant	Neutral	Quite important	Very important
Unit					
Integration					
System					
System Integration					
Functional					
Acceptance					

9. To what extent are previously mentioned tests automated?

	0-20%	20-40%	40-60%	60-80%	80-100%
Unit					
Integration					
System					
System Integration					
Functional					
Acceptance					

10. How important is automation at previously mentioned tests for project success?

	Completely unimportant	Quite unimportant	Neutral	Quite important	Very important
Unit	○	○	○	○	○
Integration	○	○	○	○	○
System	○	○	○	○	○
System Integration	○	○	○	○	○
Functional	○	○	○	○	○
Acceptance	○	○	○	○	○

Knowledge & resources

11. What is the knowledge level of developers on to the following tests?

	Absent	Limited	Adequate	Ample	Perfect
Unit	○	○	○	○	○
Integration	○	○	○	○	○
System	○	○	○	○	○
System Integration	○	○	○	○	○
Functional	○	○	○	○	○
Acceptance	○	○	○	○	○

12. What is the knowledge level of customers on to the following tests?

	Absent	Limited	Adequate	Ample	Perfect
Functional	○	○	○	○	○
Acceptance	○	○	○	○	○

13. To what extent are the following test resources available?

	Absent	Limited	Adequate	Ample	Perfect
Test scenarios	○	○	○	○	○
Basis testsets	○	○	○	○	○
Dedicated test hardware	○	○	○	○	○
Continuous build server (with auto check-in)	○	○	○	○	○
Easily maintained test tools	○	○	○	○	○

14. How important is the availability of the following test resources to project success?

	Completely unimportant	Quite unimportant	Neutral	Quite important	Very important
Test scenarios	○	○	○	○	○
Basis testsets	○	○	○	○	○
Dedicated test hardware	○	○	○	○	○
Continuous build server (with auto check-in)	○	○	○	○	○
Easily maintained test tools	○	○	○	○	○

Statements

15. To what extent do you agree to the following statements?

	Completely disagree	Disagree	Neutral	Agree	Completely agree
Customers need to write test plans on their own	○	○	○	○	○
Customers test properly	○	○	○	○	○
Customers need support during testing	○	○	○	○	○
Customers need to test at SME internally	○	○	○	○	○
Customers are available for business context coding questions	○	○	○	○	○
Customers are involved in the testing process	○	○	○	○	○
The current development planning guarantees adequate testing (in practice)	○	○	○	○	○
Testen occurs ad-hoc	○	○	○	○	○
Testing is skipped / severely shortened when developing deadlines aren't met	○	○	○	○	○
Test execution is stimulated by project managers	○	○	○	○	○
Feedback in own code (for instance as bugs) arrives soon enough	○	○	○	○	○
Testing receives enough attention	○	○	○	○	○

I am certain of the effects of inserting new code into the application base					
The amount of testing and quality hereof is highly dependent on the developer personality					
Attention for testing degrades as development progresses					
Influence of code changes on system functioning is underestimated					
Testing is executed without a clear strategy					
Code is written to be testable					
Effects on the rest of the application are unknown when refactoring					
Current unit tests test too large chunks of code simultaneously					
Less time required in bug-fixing by spending more time at testing nets less development time					
I trust bug-freeness of currently running applications					
When working with live code, considerately more tests are executed					
Acceptance testing forms an ample viewing point for application completeness.					

Wrap-up

16. What grade would you like to award the current testing process? (integer; 1 = worst - 10 = best)

Please fill in the principal reason for the realization of this grade.

17. Open space: if you've got things you want to share or mention about the current testing at SME, things that please or bother you or even possible improvements, let them know here.

N.B. All information is handled confidentially and results will be published anonymously.

18. Would you like to be informed about the results of this survey?

- ○ Yes
- ○ No

19. Are you available for possible questions in response of your answers?

- ○ Ja
- ○ Nee