Dealing with Technical Debt in Agile Development Projects

Harry M. Sneed

ANECON GmbH, Vienna, Austria
Fachhochschule Hagenberg, Upper Austria
Harry.Sneed@T-Online.de

Abstract. Technical Debt is a term coined by Ward Cunningham to denote the amount of rework required to put a piece of software into that state which it should have had from the beginning. The term outdates the agile revolution. Technical debt can occur using any development approach, but since agile development has become wide spread the notion of "technical debt" has gained much more attention. That is because if agile development is not done properly, technical debt accrues very fast and threatens to strangle the project. In this paper the author describes how technical debt comes to being, how it can be measured and what can be done to prevent it in agile development. The emphasis of the paper is on how to prevent it. What measures are necessary in an agile development project to keep technical debt from accruing over time? The paper lists out a number of measures which can be taken – organizational, procedural and technical. Automated tools play an important role in the struggle against technical debt. Samples are given how a tool can be helpful in identifying and removing problems before they get out of hand. Also important is an external auditing agency which monitors projects in progress with the aide of automated tools. In the end a case study is presented which illustrates the monitoring of technical debt within an agile development and what counter measures are required to stop it.

Keywords: Technical debt, code quality, product documentation, software defects, refactoring, software metrics, agile team organization, agile testing.

1 Technical Debt

"*Technical Debt*", is a term for the work required to put a piece of software in the state that it should be in. It may be that this is never done and the software remains in the substandard state forever, but still the debt remains. In other words, technical debt is a term for substandard software. It was coined to describe poor quality software in terms that business managers can relate to, namely in terms of money, money required to fix a problem [1]. Managers should become aware of the fact the neglect of software quality costs them money and that these costs can be calculated. The notion of "debt" should remind them that someday they have to pay it back or at least try to reduce it, just like a country should reduce the national debt. The size of the national debt is an indicator that a national economy is spending more than what it is

D. Winkler, S. Biffl, and J. Bergsmann (Eds.): SWQD 2014, LNBIP 166, pp. 48–62, 2014.

producing. It has a negative balance. The size of the technical debt is an indicator that an organization is producing more software than it can make correctly as it should be. The amount of the national debt can be measured absolutely in terms of Dollars or Euros and relatively in relation to the gross national product. The same applies to the technical debt. It too can be measured absolutely in terms of money required to renovate the software and relatively in terms of the costs of renovation relative to the development costs. Just as a country whose national debt exceeds its annual gross national product is in danger of bankruptcy, a software producer whose technical debt exceeds its annual development budget is in danger of collapsing. Something must be done to eliminate or at least to reduce the size of the debt.

The notion of "*technical debt*" was coined by Ward Cunningham at the OOPSLA conference in 1992. The original meaning as used by Cunningham was "all the not quite right code which we postpone making it right." [2]. With this statement he was referring to the inner quality of the code. Later the term was extended to imply all that should belong to a properly developed software system, but which was purposely left out to remain in time or in budget, system features such as error handling routines, exception conditions, security checks and backup and recovery procedures and essential documents such as the architecture design, the user guide, the data model and the updated requirement specification. All of the many security and emergency features can be left out by the developer and the users will never notice it until a problem comes up. Should someone inquire about them, the developers can always say that these features were postponed to a later release. In an agile development they would say that they were put in the backlog. In his book on "Managing Software Debt - Building for Inevitable Change", Sterling describes how this debt accumulates – one postponement at a time, one compromise after another [3].

There are many compromises made in the course of a software development. Each one is in itself not such a problem, but in sum the compromises add up to a tremendous burden on the product. The longer the missing or poorly implemented features are pushed off, the less likely it is that they will ever be added or corrected. Someone has to be responsible for the quality of the product under construction. It would be the job of the testers in the team to insist that these problems be tended to before they get out of hand. That means that the testers not only test but also inspect the code and review the documents. Missing security checks and exception handling are just as important as incorrect results. For that the testers must be in a position to recognize what is missing in the code and what has been coded poorly. This will not come out of the test. The test will only show what has been implemented but not what should have been implemented. What is not there cannot be tested. Nor will it show how the code has been implemented. Testers must have the possibility and also the ability to look into the code and see what is missing. Otherwise they would have to test each and every exception condition, security threat and incorrect data state. That would require much too much testing effort. For this reason testers should also have good knowledge of the programming language and be able to recognize what is missing in the code. Missing technical features make up a good part of the technical debt [4].

The other part of the technical debt is the poor quality of the code. This is what Ward Cunningham meant when he coined the term. In the heat of development developers make compromises in regard to the architecture and the implementation of

the code in order to get on with the job or simply because they don't know better. Instead of adding new classes they embed additional code into existing classes because that is simpler. Instead of calling new methods they nest the code deeper with more if statements, instead of using polymorphic methods, they use switch statements to distinguish between variations of the same functions, instead of thinking though the class hierarchy, they tend to clone classes thus creating redundant code. There is no end to the possibilities that developers have to ruin the code and they all too often leave none out, when they are under pressure as is the case with sprints in Scrum. In their book on "Improving the Design of existing Code", Fowler and Beck identify 22 code deficiencies, which they term as bad smells, or candidates for refactoring [5]. These smells such bad practices as duplicated code, long methods, large classes, long parameter lists, divergent change, shotgun surgery and feature envy. In a contribution to the journal of software maintenance and evolution with the title "Code Bad Smells – a Review of Current Knowledge", Zhang, Hall and Baddoo review the literature on bad smells and their effects on maintenance costs [6]. The effects of bad smells range from decreasing readability to causing an entire system to be rewritten. In any case they cause additional costs and are often the source of runtime errors which cost even more. The fact is undisputed that certain bad coding habits drive up maintenance costs. Unfortunately these costs are not obvious to the user. They do not immediately affect runtime behavior. The naïve product owner in an agile development project will not recognize what is taking place in the code. However, if nothing is done to clean up the code, the technical debt is growing from release to release (see Figure 1: accumulating technical debt)

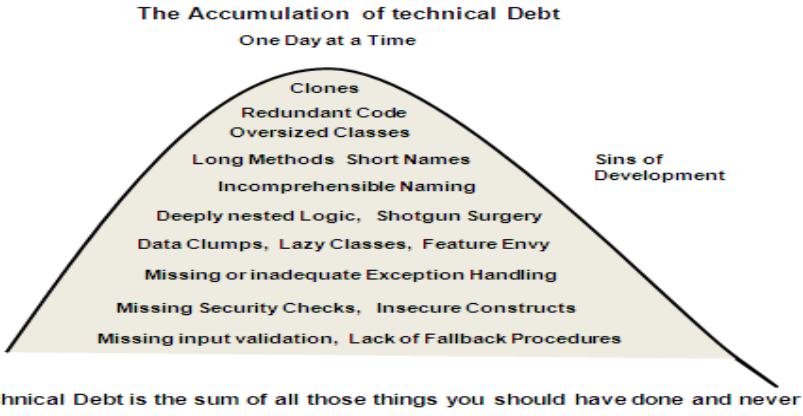

Fig. 1.

It has always been known that poor coding practices and missing features cause higher maintenance costs but no one has ever calculated the exact costs. This has now been done by Bill Curtis and his colleagues from CAST Software Limited in Texas. Their suggestion or computing the costs of bad quality joins each deficiency type with the effort required to remove that deficiency. The basis for their calculation is an experience database from

which the average cost of removing each deficiency type is calculated. The data from scores of refactoring projects has been accumulated in this database [7]. The refactoring and retesting of a single overly complex method can cost a half day. Considering an hourly rate of US $70, that leads to a total cost of $280. The cost of a missing exception handling may cost only an hour but if five hundred such handlings are missing the cost will amount to some $35,000. The cost of adding a missing security can cost up to three person days or $1600. The main contribution of Curtis and his colleagues is that they have identified and classified more than a hundred software problem types and put a price tag on them. Samples of their problem types are:

- Builtin SQL queries (subject to hacking)
- Empty Catch Blocks (no exception handling)
- Overly complex conditions (error prone)
- Missing comments (detracts from comprehension)
- Redundant Code (swells the code amount and leads to maintenance errors)
- Non-uniform variable naming (making the code hard to understand)
- Loops without an emergency brake (can cause system crash)
- Too deeply nested statements (are difficult to change).

The costs of such deficiencies are the sum of the costs for each deficiency type. The cost of each deficiency type is the number of occurrences of that deficiency times the average effort required to remove and retest that deficiency as taken from the experience database. (see Figure 2: Code Deficiencies)

Technical Debt in Terms of Code Deficiencies

Major deficiencies sorted by number of occurrences	Correction Effort (Person Hours)	
(10) Return Value is not controlled after Method Call 2435	x 0,5	= 1217
(22) Public Variables should be avoided in Classes 2280	x 1	= 2280
(01) IO-Operations are not in a try block 913	x 1,5	= 1370
(25) Class is not derived from a Superordinate Class 219	x ?	= ?
(14) Control logic exceeds maximum allowed nesting level 962	x 2	= 1924
(15) Missing Final clause in method declaration 856	x 0,5	= 428
(04) Data Casting should be avoided 692	x 2	= 1384
(18) Check on incoming public request is missing 535	x 4	= 2140
(26) Nested Classes are not allowed 376	x 4	= 1504
(27) Returning a Function my cause an endless loop 239	x 6	= 1434
(11) Conditions should not contain an Assignment 150	x 1	= 150
(17) Try and Catch clauses do not match 112	x 1	= 112
(13) Default is missing in Switch Statement 85	x 1	= 85
(12) Case block should contain a Break statement 77	x 2	= 154
(08) Method Invocation with Array is not in a try Block 31	x 3	= 93
(07) External Variables are not allowed 29	x 2	= 58
(06) Standard IO Functions are prohibited 21	x 4	= 84
(09) There should be no global Data Definitions in C# 5	x 2	= 10
(02) Two Dimensional Arrays violate 1. Normal Form 2	x 8	= 16
Total Correction Effort		= 14.443

Fig. 2.

To the costs of these statically recognizable deficiencies must be added the costs of removing errors which come up during operations. Due to time constraints not all errors will be fixed before release. These errors may lead later to interruptions in production if they are not corrected. Other errors may lead to false results which must be manually corrected. Bad code can lead to bad performance and that too will

become annoying to the users. The users may accept living with these inconveniences for a while but eventually they will insist that they be fixed. The cost of fixing them is not trivial. Together with the code smells and the missing features this adds up to a significant debt. Bill Curtis estimates that the median debt of an agile project is $ 3.61 per statement [8].

That is the absolute measure for debt. It is also possible to compute the relative debt. That is the cost of renovating the software relative to the cost of development.

Relative Debt = Renovation Cost / Development Cost

It could be that the cost of renovation is almost as high as the development cost. In that case the software should be abandoned and rewritten.

2 Accumulating Technical Debt

In their endeavor to give the user a functioning software product in the shortest possible time, developers are inclined to take short cuts. They leave the one or the other feature out with the good intention of adding it later. A typical example is error handling. If a foreign function is called on another server, the developer should know that he will not always get an answer. It can be that the remote server is overloaded or that it is out of operation. This applies particularly to web services. For this reason, he should always include an exception handler to check the time and hen a given time is exceeded to invoke an error handling routine. The same applies to accessing files and databases as well as to all external devices like printers and displays. In addition, a cautious developer will always check the returned results to make sure that they are at least plausible. The result of this defensive programming is a mass of additional code. Experts estimate that at least 50% of the code should be for handling exception conditions and erroneous states if the developer is really concerned about reliability [9].

By disregarding reliability problems the developer can save half of his effort for coding and testing, since the error handling he does not code will also not have to be tested. The temptation is very great to do this. The agile developer knows that the user representative will not notice what goes on behind the user interface. The code does not interest him. Often it will have to come to a tremendous load or to extreme situations before an exception condition is triggered. Other exceptions come up only in certain cases. Even those exceptions which do come up in test, leading to short interruptions, will not be immediately registered by the user. He is too busy dealing with other things. Therefore, if a punctual release is the overriding goal, it is all too tempting to leave the exception handling out. As John Shore, the manager of software development at the Naval Research Lab once pointed out, programmers are only poor sinners, who, under pressure, cannot withstand the temptation to cheat, especially if they know they will probably not be caught [10]. The developer has the good intention of building in the missing code later when he has more time. Unfortunately, this “later” never comes. In the end it is forgotten until one day a big crash takes place in production. The system goes down because an exception occurs and is not handled properly. The path to hell is paved with good intentions.

Tilo Linz and his coauthors also comment on this phenomena in their book on testing in Scrum projects [11]. There they note that the goal of finishing a release in a fully transparent environment with an unmovable deadline in a prescribed time box can lead to tremendous mental burden on the developers. Theoretically, according to the Scrum philosophy they should be able to adjust the scope of the functionality to their time and their abilities. In practice, this is most often not possible. Instead of cutting back on functionality, the team cuts back on quality. Discipline is sacrificed to the benefit of velocity. The functionality is not properly specified, test cases that should be automated are executed manually, necessary refactoring measures are pushed off. The software degrades to a point where it can no longer be evolved. The team is sucked under by a downwards spiral and in the end the project is discarded.

The situation with security is similar. Secure code is code that protects itself against all potential intrusions. But to achieve that means more code. A secure method or function controls all incoming calls before it accepts them. Every parameter value should be checked to make sure it is not corrupt. The developer should also confirm that the numbers and types of arguments matches that what is expected, so that no additional parameters are added to the list. These could be hooks to get access to internal data. The values of the incoming parameters should be checked against predefined value ranges. This is to prevent corrupted data from getting into the software component. To make sure that only authorized clients are calling, the function called can require an identification key to each call. In Java the incoming objects should be cloned, otherwise it can be mutated by the caller to exploit race conditions in the method. Methods which are declared public for testing purposes should later be changed to private or protected to restrict access rights, and classes should be finalized when they are finished to protect their byte code from being overwritten [12]. Embedded SQL statements are particularly vulnerable to attack since they can be easily manipulated at runtime. They should be outsourced into a separate access shell, but that entails having extra classes which also have to be tested [13]. Thus, there are many things a developer can do to make his code more secure, if he takes the time to do it. But here too, the developer can rest assure that nobody will notice the missing security checks unless they make an explicit security test. So security is something that the developer can easily postpone. The users will only become aware of the problem when one day their customer data is stolen. Then it will be too late.

That is the problem with software. Users cannot easily distinguish between prototypes and products. A piece of software appears to be complete and functioning, even when it is not. As long as the user avoids the pitfalls everything seems to be ok. They have no way of perceiving what is missing. For that they would either have to test every possible usage or they have to dig into the code. The question of whether a piece of software is done or not, cannot be answered by the user, since he has no clue of what “done” really means. The many potential dangers lurking in the code cannot be recognized by a naive user, therefore it is absurd to expect that from him. He needs an agent to act in his behalf who can tell him if the software is really done or not. This agent is the tester, or testers, in the development team [14].

3 Role of Automated Static Analysis in Detecting Technical Debt

Static analysis is one of many approaches aimed at detecting defects in software. It is a process of evaluating a system or component based on its form, structure, content or documentation [15] which does not require program execution. Code inspections are one example of a classic static analysis technique relying on manual examination of the code to detect errors, violations of coding conventions and other potential problems. Another type of static analysis is the use of automated tools to identify anomalies in the code such as non-compliance with coding standards, uncaught runtime exceptions, redundant code, incorrect use of variables, division by zero and potential memory leaks. This is referred to as automated static analysis.

There are now many tools available to support automated static analysis – tools like Software Sonar, Conformity and SoftAudit. In selecting a proper tool, the main question is what types of deficiencies are reported by the tool. Another question has to do with the number of false positives the tool produces that is how many defects or deficiencies are by closer examination not really true. When tools produce a great number of false positives, this only distracts from locating the real problems. As to the types of true problems reported, it is important to have a problem classification schema. From the viewpoint of technical debt there are three major classes of code problems:

- missing code = statements that should be there and or not
- redundant code = statements that are there and should not be
- erroneous code = statements that may cause an error when the code is executed
- non-conform code = statements which violate a local standard or coding convention
- deficient code = statements that reduce the overall quality of the code, maintainability, portability, testability, etc.

Samples of missing code are:

- a missing break in a switch statement
- a missing try clause in the call of a foreign method
- a missing check of return value after a function call

Samples of redundant code are:

- cloned codes, i.e. code sections which are almost the same except for minor variances
- variables declared but not used
- methods implemented but never called
- dead code or statements which can never be reached

Samples of erroneous code are:

- missing loop termination conditions
- pointers or object references that are used before they are set, i.e. null-pointers
- variables that are used before they are initialized
- casting of variables, i.e. changing the data type at run time.

Samples of non-conform code are:

- missing prescribed comments
- data names which do not adhere to the naming convention
- procedures which are too long or too complex, i.e. they exceed the maximum number of statements or the maximum nesting level
- statements which exceed the maximum line length

Samples of deficient code:

- nested statements which are not intended
- conditions which are too complex
- non portable statements, i.e. statements which may work in one environment but not in another
- statements which deduct from the readability such as expressions in conditions

Also many of the security threats in the code can be detected through automated static analysis. *Jslint* from Citital is representative of tools which perform such a security analysis. It enforces 12 rules for secure Java code. These rules are:

1) Guard against the allocation of uninitialized objects
2) Limit access to classes and their members
3) Declare all classes and methods as final
4) Prevent new classes from being added to finished packages
5) Forbid the nesting of classes
6) Avoid the signing of code
7) Put all of the code in one jar or archive file
8) Define the classes as uncloneable
9) Make the classes unserializeable
10) Make the classes undeserializeable
11) Forbid company classes by name
12) Avoid the hardwiring of sensitive data in the code

Jslint scans the code for infringement of these rules. All of the rule violations can be recognized except for the last one. Here it is not possible to distinguish between sensitive and non-sensitive data. Therefore it is better not to use hard-wired text in the code at all. If it is, then it can be easily recognized. In some cases *Jslint* can even correct the code, for instance by overwriting the access class, by adding the final

clause and removing code signatures. The automated correction of code can however be dangerous and cause undesired side effects. Therefore, it is better to use automated tools to detect the potential problems and to manually eliminate them [16].

A study was performed at the North Carolina State University on over 3 million lines of C++ Code from the Nortel Networks Corporation to demonstrate how effective automated static analysis can be. Using the tool "FlexLint" the researchers there were able to detect more than 136.000 problems or one problem for every 22 code statements. Even though this code had been in operation for over three years there remained a significant technical debt. The problems uncovered by the automated static analysis also correlated well to the defects discovered during the prerelease testing with an accuracy of 83%. That implies that 83% of the erroneous modules could have been identified already through static analysis.

Of the defects uncovered in the static analysis, 20% were critical, 39% were major and 41% were minor. The coding standard violations were not included in this count. Based on their findings, the authors of this study came to the following conclusions:

- the defect detection rate of automated static analysis is not significantly different than that of manual inspections
- the defect density recorded by automatic static analysis has a high correlation to the defect density recorded in prerelease testing
- the costs of automated static analysis is a magnitude less – under 10% - of the costs occurred by manual inspection and prerelease testing

Thus, automated static analysis can be considered an effective and economic means of detecting problematic code modules [17].

One of the areas addressed by the static analysis study at North Carolina State was that of security vulnerabilities. The authors of the study paid particular attention to code constructs that have the potential to cause security vulnerabilities if proper protection mechanisms are not in place. Altogether 10 security vulnerabilities were identified by the automated analysis tool:

- Use of Null pointers
- Access out of bound references, i.e. potential buffer overflow
- Suspicious use of malformed data
- Type mismatch in which statements
- Passing a null pointer to another function
- Failure to check return results
- Division by zero
- Null pointer dereference
- Unrecognizable formats due to malformed data
- Wrong output messages

The number of false positives in the check of these potential security violations was higher than average, however the results indicate that automated static analysis can also be used to find coding deficiencies that have the potential to cause security vulnerabilities. The conclusion of this research is that static analysis is the best instrument for detecting and measuring technical debt [18].

4 Early Recognition of Mounting Technical Debt

The main contribution of agile testing is to establish a rapid feedback test to development. This is the rational for having the testers in the team. When it comes to preventing quality degradation and increasing debt, the testers in an agile project have more to do than just test. They are there to assure the quality of the product at all levels during construction. This should be accomplished through a number of control measures, measures such as reviewing the stories, transforming the stories into testable specifications, reviewing the architectural design, inspecting the code, validating the unit tests, performing continuous integration testing and running an acceptance test with the user. On top of that, there still have to be special performance, load and security tests, as well as the usability and reliability tests. If the product under construction is not just a trivial temporary solution, then those special tests should be made by a professional testing team, which works independently of the agile developing teams. This solution is referred to by Linz and his coauthors as "System Test Team" whose job it is to shake down the systems delivered by the Scrum teams independently of the schedule imposed on the project. That means that the releases of the agile development team are really only prototypes and should be treated as such. The final product comes only after the prototypes have gone through all of the special tests. To these special tests belongs also the final auditing of the code [19]. (see Figure 3:The role of the Tester)

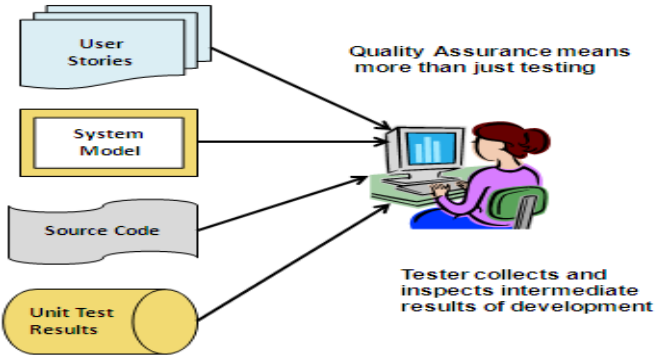

Fig. 3.

The review of the stories is concerned with discussing and analyzing the stories with the view of enhancing and improving them. They should also be checked for testability. The user representative might overlook something or fail to cover all describe a function adequately. It is up to the testers in the team to point this out and to clear the missing and unclear issues with him. Special attention should be paid by the testers to the non-functional aspects of the stories like security and usability. In any case the stories need to be purified before the developers start to implement them. The testers need stories they can interpret and test.

In order to establish a testing baseline, the testers should convert the informally defined user stories into an at least semi-formal requirement specification. As noted

above a significant portion of the technical debt is caused by missing functions, but who is to know that they are missing when they are not specified anywhere. All of the exception conditions, security checks and other quality assuring functions of a system should be noted down as non-functional requirements or as business rules. They should be specified in such a way as to be recognizable, for instance with key words and to be convertible into test cases. For every business rule and non-functional requirement at least one test case should be generated to ensure that the possible exceptions and security threats are tested. In this way the requirements can be used as an oracle for measuring test coverage [20].

In checking the code for conformity with the coding rules testers should also check the format and the readability. Most all of the missing exception handling, security measures and data validation checks can be recognized in the code. This is the purpose of an automated code audit. It only takes a few minutes of the tester's time to set up and run such an audit. More time is needed to explain the reported deficiencies to the developers and to convince them that they should clean them up. Of course the tester cannot force the developers to do anything, but he can post the deficiencies on the project white board and try to convince the team members that it is to their benefit to have these coding issues settled, if not in the next release then sometime in a future release. In no case should he let the team forget them. A good way of reminding the team is to post a chart on the amount of technical debt accrued in the project room for everyone to see, similar to the clock in Times Square which reminds Americans constantly of the current state of their national debt. Just as in New York where many American citizens simply shrug their shoulders and say "so what", project members who could care less about the final quality of their product will ignore the issue of technical debt, but at least it will remind them of their sins. .

By controlling the results of the unit tests, the testers can point out to the developers what they have missed and what they can do to improve their test. In checking through the trace and coverage reports, the tester can identify what functions have not been adequately tested and can suggest to the developer additional test cases required to traverse those hidden corners of the code. Here too, he must convince the developer that it is to his benefit to filter out the defects as early as possible. Pushing them off to the acceptance test only increases the cost of correcting them. Every defect removed in unit testing is one defect less to deal with in integration and acceptance testing [21].

The testers in an agile development team must strive to be only one or maximal two days behind the developers [22]. No later than two days after turning over a component or class, the developer should know what he has done wrong. "Continuous Integration" makes this possible [23]. The tester should maintain an automated integration test frame into which he can insert the new components. The existing components will already be there. Every day new or corrected components are taken over into the build. In the case of new functions, the test scripts and test data have to be extended. Not only will the new functions be tested, but the test of the old functions will be repeated. The setting up of such an automated regression test may take days, but after that the execution of each additional test should take only few hours. The goal is to keep a steady flow of testing. The actual data results should be

constantly compared with the specified results by means of an automated data comparator. Should new input data be required, it should be generated by an automated test data generator. The two tools should be joined by a common test language which ensures that the test outputs match with the test inputs. It is imperative that all deviations from the expected behavior of the software be automatically registered and documented. It should go so far that the defect reports are also automatically generated. The tester should only have to enhance them. In agile development time is of essence and anything which can save time should be introduced (see Figure 4: Testing in an agile Development Project).

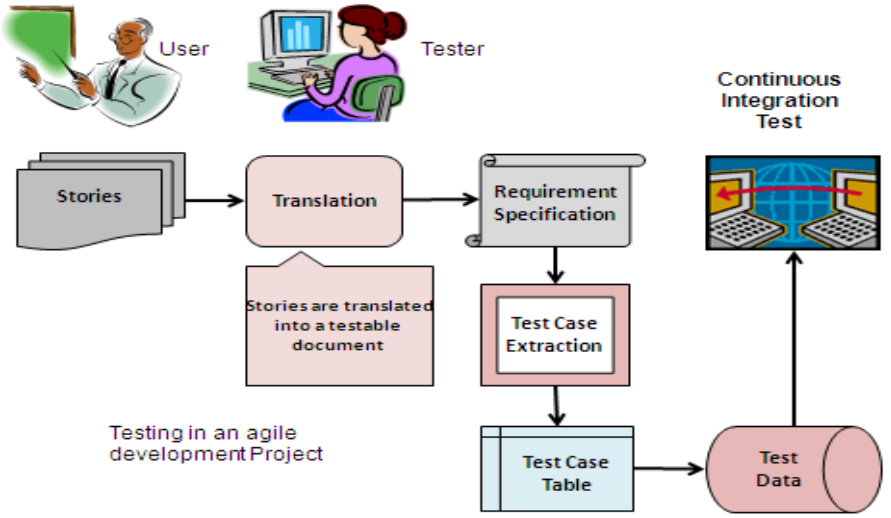

Fig. 4.

The main goal of agile development is rapid feedback. Problems should be reported back immediately to the developer to be resolved before they become acute. This is also the great advantage of agile development visa via classical phase-oriented software development where it often took weeks if not months before the problems were detected. In the meantime, the developers assumed that all was honky dory and went on developing new erroneous components or, in the worst case, they did nothing and just waited on the feedback from the testers. In this way many valuable hours of tester time were lost. Thus, it is truly beneficial to have component integration testing running parallel to component development [24].

The early recognition of defects and the quick non bureaucratic handling of defect reports is one the main advantages of the agile development process over the classic waterfall process [25]. To function well the testers should be in constant contact with the developers. Whether this contact must be physical is disputed. The authors of the agile manifesto put great emphasis on “face to face” communication. Many proponents of agile development agree with that and insist that the testers be physically together with the developers. Others argue that it is enough for the testers

to be in virtual contact with the developers. Physically they might be in another continent. The proponents of the virtual team claim that it suffices to communicate daily with a social networking system and video conferences via the internet. Whether this really is feasible or not remains to be seen. For the moment there remains two schools of thought [26].

5 What Is "Done"?

At the Belgian "Testing Days" conference in 2012 Johanna Rothman und Lisa Crispin, two recognized experts on agile testing, discussed what is meant by "done" [27]. In the opinion of Rothman this is something that must be decided upon by the team as a whole. It is, however, up to the testers to trigger the discussion and to lead it a conclusion. The testers should feed the discussion with their experience from previous projects and to steer it in a positive direction. Rothman proclaims "you have to get the team thinking about what is done. Does it mean partially done, as in it is ready for testing or fully done, as in it is ready for release?" A minimum quality line is required in order to make an intermediate release. If the users know they are working with a prototype they are more likely to be error tolerant. If they know, this release will be the final product they will be stricter. In the end the users must decide what is done. But, since users only have a restricted view of the system, they need others to help them in passing final judgment on the state of the software. Those others are the testers who are prepared to assess the system on behalf of the potential users. It is all too easy, just to declare a product as being "done". The lead tester should, in any case, be involved I the discussion. The decision as to whether something is "done" or not is, in the end, a political decision, which must be considered from several points of view.

Developers tend to become impatient and to push the product thru in any case, even if it is falling apart. They will always argue that the quality is sufficient. Testers will, on the other hand, claim that the quality level is not high enough. They will argue for more time to test. Otherwise the quality problems will only be pushed off on the maintenance team. Rothman suggests using the KANBAN progress charts to show the current quality state of each and every component as well as the product under development. This way everyone in the project can see where they are, relative to the goals they have set for themselves. In effect, the project should produce two quality reports, one on the functionality and one on the quality. (see Figure 5: Measuring Technical Debt).

The functional state of a project is easier to assess than the qualitative state. It is visible whether a function is performing or not. The quality state is not so visible. To see what security and data checks are missing, the tester has to dig into the code. To see how many rules are violated he has to have the code audited. There is no easy way to determine how many defects remain. This can only be known after every last function has been tested for all variations. That is why, one must often guess when deciding on the state of product quality. Perhaps the best indicators of product quality are the number of code deficiencies relative to the number of code statements and the number of defects detected so far relative to the test coverage. For both metrics there are benchmark levels which the testers can discuss with the user and the other team members [28].

Measuring Technical Debt relative to Development Costs

Accumulated debt should be reduced from Release to Release

Component	1. Release	2. Release	3. Release	4. Release	5. Release
Client-GUI	0,30	0,20	0,15	0,10	0,05
Input-Validation	0,25	0,15	0,10	0,05	0,03
Output-marshall	0,20	0,20	0,15	0,10	0,05
Database-Access	0,15	0,15	0,10	0,05	0,01
Total Debt	0,225	0,175	0,125	0,075	0,03

Debt = (Renovation Costs / Construction Costs)

Fig. 5.

Johanna Rothman argues that testers must be involved in the discussion as to what is "done" from the beginning of the project on. "To be done also means that the quality criteria set by the team are met". For that this criteria has to be accepted and practiced by all team members. Everyone in the team must be fully aware of his or her responsibility for quality and must be dedicated to improving it. Rothman summarizes "Everybody in the team needs to take responsibility for quality and for keeping technical debt at a manageable level. The whole team has to make a meaningful commitment to quality". It is true that quality is a team responsibility but the testers have a particular role to play. They should keep track of the technical debt and keep it visible to the other team members [29].

Lisa Crispin points out that software quality is the ultimate measure of the success of agile development [30]. Functional progress should not be made by sacrificing quality. After each release, i.e. every 2 to 4 weeks, the quality should be assessed. If it falls below the quality line, then there has to be a special release devoted strictly to the improvement of quality, even if it means putting important functions on ice. In this release the code is refactored and the defects and deficiencies removed. Crispin even suggests having a separate quality assurance team working parallel to the development team to monitor the quality of the software and to report to the developers. This would mean going back to the old division of labor between development and test, and to the problems associated with that. The fact is that there is no one way of ensuring quality and preventing the accumulation of technical debt. Those responsible for software development projects must decide on a case by case basis which way to go. The right solution is as always context dependent.

References

1. Kruchten, P., Nord, R.: Technical Debt – from Metaphor to Theory and Practice. IEEE Software, S.18 (December 2012)
2. Cunningham, W.: The Wgcash Portfolio Management System. In: Proc. of ACM Object-Oriented Programming Systems, Languages and Applications, OOPSLA, New Orleans, p. S.29 (1992)

3. Sterling, C.: Managing Software Dept – Building for inevitable Change. Addison-Wesley (2011)
4. Lim, E., Taksande, N., Seaman, C.: A Balancing Act – What Practioneers say about Technical Debt. IEEE Software, S.22 (December 2012)
5. Fowler, M., Beck, K.: Improving the Design of existing Code. Addison-Wesley, Boston (2011)
6. Zhang, M., Hall, T., Baddoo, M.: Code Smells – A Review of Current Knowledge. Journal of Software Maintenance and Evolution 23(3), 179 (2011)
7. Curtis, B., Sappidi, J., Szynkarski, A.: Estimating the Principle of an Application's Technical Debt. IEEE Software, S. 34 (December 2012)
8. Wendehost, T.: Der Source Code birgt eine Kostenfalle. Computerwoche 10, S.34 (2013)
9. Nakajo, T.: A Case History Analysis of Software Error Cause and Effect Relationships. IEEE Trans. on S.E. 17(8), S.830 (1991)
10. Shore, J.: Why I never met a Programmer that I could trust. ACM Software Engineering Notes 5(1) (January 1979)
11. Linz, T.A.O.: Testing in Scrum Projects, p. 177. Dpunkt Verlag, Heidelberg
12. Lai, C.: Java Insecurity – accounting for Sublities that can compromise Code. IEEE Software Magazine 13 (January 2008)
13. Merlo, E., Letrte, D., Antoniol, G.: Automated Protection of PHP Applications against SQL Injection Attacks. In: IEEE Proc. 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), Amsterdam, p. 191 (March 2007)
14. Crispin, L., Gregory, J.: Agile Testing – A practical Guide for Testers and agile Teams. Addison-Wesley-Longman, Amsterdam (2009)
15. Ayewah, N., Pugh, W.: Using Static Analysis to find Bugs. IEEE Software Magazine, 22 (September 2008)
16. Vieg, J., McGraw, G., Felten, E.: Statically Scanning Java Code – Finding Security Vulnerabilities. IEEE Software Magazine, 68 (September 2000)
17. Zheng, J., Williams, L., Nagappan, N., Snipes, W.: On the Value of Static Analysis for Fault Detection in Software. IEEE Trans. on S.E. 32(4), 240 (2006)
18. Gueheneu, Y.-G., Moha, N., Duchien, L.: DÉCOR – A Method for the Detection of Code and Design Smells. IEEE Trans. on S.E 36(1), 20 (2010)
19. Linz, T.A.O.: Testing in Scrum Projects, p. 179. Dpunkt Verlag, Heidelberg
20. Sneed, H., Baumgartner, M.: Value-driven Testing – The economics of software testing. In: Proc. of 7th Conquest Conference, Nürnberg, p. 17 (September 2009)
21. Marre, M., Bertolini, A.: Using Spanning Sets for Coverage Measurement. IEEE Trns. on S.E. 29(11), 974 (2003)
22. Bloch, U.: Wenn Integration mit Agilität nicht Schritt hält. Computerwoche 24, S. 22 (2011)
23. Duvall, P., Matyas, S., Glover, A.: Continuous Integration – Improving Software Quality and reducing Risk. Addison-Wesley, Reading (2007)
24. Humble, J., Fairley, D.: Continuous Delivery. Addison-Wesley, Boston (2011)
25. Cockburn, A.: Agile Software Development. Addison-Wesley, Reading (2002)
26. Bavani, R.: Distributed Agile Testing and Technical Debt. IEEE Software, S.28 (December 2012)
27. Rothman, J.: Do you know what fracture-it is? Blog (June 2, 2011), http://www.jrothman.com
28. Letouzey, J.L., Ilkiewicz, M.: Managing Technical Debt. IEEE Software Magazine, 44 (December 2012)
29. Rothman, J.: Buy Now, Pay Later. In: Proc. of Belgium Testing Days, Brussels (March 2012)
30. Crispin, L., House, T.: Testing Extreme Programming. Addison-Wesley-Longmann, Reading (2002)