

Kevin C. Martin

The Agile Tester

Table of Contents

<u>Preface</u>	2
<u>Chapter 1. About the Author</u>	7
<u>Chapter 2. Introduction</u>	10
<u>Chapter 3. Foundations of Testing</u>	14
<u>Chapter 4. The Testers Mindset</u>	32
<u>Chapter 5. Types of Testing</u>	39
• <u>Black Box Testing</u>	41
• <u>White Box Testing</u>	71
• <u>Static Testing</u>	77
<u>Chapter 6. The Path to Agile</u>	83
<u>Chapter 7. The Agile Process Framework</u>	95
<u>Chapter 8. The five agile meetings</u>	127
<u>Chapter 9. The Agile Tester</u>	145
<u>Chapter 10. The Agile Organisation</u>	163
<u>Chapter 11. Moving to agile and what to avoid</u>	178
<u>Chapter 12. User Stories</u>	189
<u>Chapter 13. Burndown Charts</u>	195
<u>Chapter 14. Automated Testing with Selenium</u>	199
<u>Chapter 15. Selenium and Java</u>	207
<u>Chapter 16. Selenium and ASP.Net</u>	217
<u>Chapter 17. The first Selenium test</u>	223
<u>Chapter 18. The Selenium Command Set</u>	228
<u>Chapter 19. Continuous Integration and Deployment</u>	247
<u>Chapter 20. Team Foundation Server</u>	253
<u>Chapter 21. Conclusion</u>	261
<u>Chapter 22. Agile Myths</u>	263
<u>Glossary</u>	274
<u>Appendix 1 – Further Reading</u>	298
<u>Appendix 2 – Useful download URL's</u>	300

Preface

While many organisations have adopted the agile framework fully with a carefully planned strategy and 100% company commitment which means they are now reaping the benefits gained there are still plenty of software companies out there who have, for one reason or another, not. These companies still ignore the agile framework methodology or they have simply placed a taskboard in the centre of the office and stated 'there, now we are agile'.

While it is true that the agile methodology is not for everyone and not every software development project is suited to the framework it is however the way forward for the majority of companies who are involved in software development.

As agile has grown in popularity and usage over the decades the amount of literature about the subject has also grown. However most of the books currently available on the market focus on the project management or software development areas of the software development life cycle, there is still very little for the agile software tester to read. In the agile world; testing and the software tester are just as important as any other process or person and that is why I have written this book. Hopefully experienced and new testers alike will find some useful pointers within these humble pages which will help them enhance their career and enjoyment of testing software.

Test professionals involvement in agile projects remains challenged because of the very different nature of the agile methodology compared to older methodologies such as waterfall and the V modal. This is also not helped by a level of mis-understanding about the true nature of agile that persists in many companies and deep rooted prejudices aimed at testers by a very small percentage of programmers and project managers (*they are nothing more than failed programmers being a common feeling*).

Although there are many test professionals succeeding in agile projects, many others continue to struggle to succeed and achieve their true potential that their skills and dedication deserve. Testers who have spent many years testing outside of agile can also often struggle to make the jump across from the waterfall methodology. However with quality training, good management and self belief this jump can be completed, this is where this book comes in. This is edition three of The Agile Tester, in this edition I have updated most chapters and corrected some typing mistakes which you the readers have kindly made me aware of, if you find any more in this version please let me know.

So do you want to be an agile software tester?

Have you got what it takes?

Time to find out, read on and see.

Enjoy.

What you will need for this book

- | Web Browsers: Microsoft Internet Explorer, Google Chrome or Mozilla Firefox
- | Selenium browser drivers: Chrome Driver, Internet Explorer Driver
- | Visual Studio Community Edition
- | Eclipse and Java
- | A basic understanding of software testing

Who this book is for

This book is for existing software testers who wish to extend their skill set into the world of agile and automated software testing. The book is also for anyone who is considering entering this exciting, rewarding and challenging line of work.

Reader feedback

Feedback from my growing collection of readers is always welcome. Please let me know what you think about this book—what you liked or may have disliked. Constructive reader feedback is important for me to develop this publication further in order that you really do get the most out of it.

To send me general feedback, simply send an e-mail to feedback@Kevsbox.com, and mention the book title inside the subject of your message.

Errata

Although I have taken every care to ensure the accuracy of the content, mistakes do happen; I am after all only human. If you find a mistake in this publication—maybe a mistake in the text or the code—I would be grateful if you would report this to me. By doing so, you can save many other readers from frustration and help me improve subsequent versions of this book. If you find any errata, please report them by sending an e-mail to feedback@Kevsbox.com and include the details of the errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website (www.kevsbox.com), or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Kevsbox.com, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of my works, in any form, on the Internet, please provide me with the location address or website name immediately so that I can pursue a remedy.

Please contact us at copyright@Kevsbox.com with a link to the suspected pirated material. I appreciate your help in protecting my work, and my ability to bring you valuable content.

Questions

You can contact me at questions@Kevsbox.com if you are having a problem with any aspect of the book, and I will do our best to address it.

Kevin Charles Martin is a professionally qualified Software Developer and Software Tester with over twenty five years experience of agile, selenium, scrum, software testing, software design, support and

installation. During his career in IT he has developed, maintained and tested applications using COBOL, C, C++, Visual Basic, C# and Java.

Kevin graduated from The University of Portsmouth in September 2009 with Master of Science in Software Engineering with Merit. More recently he has passed the Professional Scrum Master 1 examination and in October 2013 he also sat and passed the ISTQB Advanced Test Manager examination while in April 2014 he sat and passed the ISTQB Certified Agile Tester examinations.

Kevin started as a software programmer and has since moved into Quality Assurance and Software Testing. He has experienced the software development cycle from all angles and has used that experience to create this book which he hopes will help both new and experience software testers equally. Kevin's other interests also include keeping fit, rock music, radio controlled ships and cooking.

Software development has changed dramatically over the past sixty years. In the early days of software development system memory was at a premium, coding was achieved in machine code and most programs were small and simple when compared to their modern counterparts. As a result testing was considered a minor task for lesser mortals and a hindrance by most programmers.

Gradually as software development tools evolved we entered the era of the DOS programmer. These programmers used development tools such as C, Ada and Pascal to write their beloved programs. They still considered testing a hindrance and tried to avoid it at any cost, it was of course a job for lesser mortals. Most software companies spent as little investment as possible on testing and most bugs tended to be reported by end users as systems crashed and burned around the globe.

With the advent of Windows a new level of complexity and confusion was added to the software mix. Suddenly not only did software development companies have to worry about their own bugs but there was also the added concerns of just how stable was the Windows operating systems their clients were now using.

Windows and other GUI's have evolved tremendously over the past two decades, some versions have been much better than others, a prime example of a quickly dropped but never to be forgotten version will always be the infamous Windows ME. All of these concerns have awoken most software houses to the real importance of high quality, technically adept, well paid software testers.

A competent, well trained software tester will also have a basic understanding of programming, SQL and computer architecture. They will be very methodical in their approach to work and the need for patience, concentration and tolerance is also very high. Today good, high quality programmers are everywhere, however good talented software testers are still few and far between.

So important is software testing today that I will even go as far to say that good software testers are a much more valuable asset than good software developers.

Over the past twenty five years there has been a slow but sure change in software development strategies as users have come to accept web based applications and now regard them as a normal part of their computing experience. No matter if it is leisure based software such as Facebook or business related most

modern applications are now either web based or have a web alternative to their more traditional windows based versions.

The richness and complexity of web software is also increasing, however the more complex a program is the more likely it has bugs and the more important the testing role becomes. Some software developers get round the testing issue by keeping their applications in a permanent Beta mode. This passes the testing mantle on to the actual user who, it would seem, is happy to test the product as they use it, usually because the software is very cheap or free.

For serious, business applications this is not an option. Applications that offer business related solutions such as job management and online ordering need to offer a high level of reliability and integrity in order to maintain customer support and confidence and avoid litigation when the system falls apart at the seams. This is where the software tester steps in and saves the day. Today customers expect more for their money and are more likely to demand compensation when things go horribly wrong. Software houses now need to be more aware of the testing role.

As I have already said today the world is awash with good software programmers; let's face it they are everywhere, look under any mouse mat and a dozen highly skilled programmers will appear demanding extra strong coffee and a connection to World of Warcraft. The same cannot yet be said for good software testers.

Good software testers are now a very valuable and sought after asset. They are far less common than a good software programmer, their skills and mentality are different and what they set out to achieve is also very different.

A programmer's aim in life is to construct what they will consider to be a flawless work of art that they see as the best code on the planet, and why not, that is where their skills are. Meanwhile a tester will do his or her best to detect and highlight any errors in the code and the logical way it works, and also why not, it is after all what they are paid to do. Despite what some people think good testers do not do this in order to simply annoy and anger the programmers; it is a serious attempt to capture as many bugs as possible before the product falls into the hands of the customer.

Therefore their role is as important as any other member of the development team. So let us look at the foundations of software testing and get an idea of where it all started back in the early days of software development.

Today software is everywhere and in everything; in fact it is now an integral part of human life. Software control's many aspects of day to day life from Traffic lights to the humble washing machine and down to the little watch strapped around your wrist. No matter who you are or what you do and where you are today you will never be far from the humble little microchip. Like it or not they do control modern life.

All of these systems have to be conceived, designed, coded and (hopefully) tested to a high standard; that is after all the software lifecycle in its most basic form. Software is designed and coded by human beings who will by their very nature make mistakes and get things wrong, after all no one is perfect, not even programmers (however only say that in front of them if you are feeling very brave).

Over the years there have been some very spectacular software failures, some of which you may have heard of already, others may be new to you. Below are just a few classic well known examples from what is now a very long and tragic list.

The Toyota Recall (2009) - There have been in fact three totally separate recalls by Toyota since 2009, and amazingly all the recalls have been related to the same thing, that is the accelerator sticking. First in August 2009 a Lexus ES350 suddenly accelerated out of control at speeds estimated to exceed 100 mph. One of the passengers called 911 before they crashed and reported that the car had "no brakes." Sadly all four passengers were killed when the car crashed. Next in November 2009, Toyota dealers were instructed to remove and shorten the gas pedals and also update the onboard computers with a new program that would override the electronic gas pedal when the brake pedal was pressed. Eventually Toyota ended up recalling more than 9 million cars worldwide in 2010, but it wasn't because of a mechanical issue. The cars all suffered from a software bug that caused a lag in the anti-lock-brake system. Overall these recalls, legal costs and other consequences are thought to have cost Toyota over \$3 billion, a very costly bug indeed.

Ariane 5 Flight 501 (1996) – In 1996 Europe's newest unmanned satellite-launching rocket, the Ariane 5, reused previously working software from its predecessor, the Ariane 4 rocket. Unfortunately, the Ariane 5's faster; more powerful engines exploited a bug that was not realized or detected in previous models. The rocket repaid this oversight and self destructed 36.7 seconds into its maiden launch.

Mariner 1 (1962) - On July 22, 1962, the first spacecraft of NASA's Mariner program blasted off on a mission to fly by Venus. Initially all looked good as the space craft happily headed towards outer space, but after a few minutes the space ship began to yaw off course. The guidance system failed to correct

the trajectory, and guidance commands failed to correct it manually. As the rocket veered off toward the busy North Atlantic shipping lanes, the range safety officer did the only thing he could do, destroy the spacecraft. Eventually the cause was nailed down to a mis-transcription of a single punctuation mark by an engineer.

The Mars Climate Orbiter Crash (1998) – This crash was eventually root caused back to a sub contractor who had designed the navigation system on the orbiter using imperial units of measurement instead of the metric system that was clearly specified by NASA. As a result the space craft attempted to stabilize its orbit too low within the Martian atmosphere and of course crashed into the ground.

Soviet Gas Pipeline Explosion (1982) - When the CIA (allegedly) discovered that the Soviet Union was (allegedly) trying to steal sensitive U.S. technology for its operation of their trans-Siberian pipeline, CIA operatives (allegedly) introduced a bug into the Canadian built system that would pass Soviet inspection but fail when in operation. This caused the largest man made non-nuclear explosion in the planets history.

The Plague in World of Warcraft (2005) - The hugely successful World of Warcraft (WoW), an online computer game created by Blizzard Entertainment, suffered an embarrassing glitch following an update to their game on September 13, 2005 – causing mass (fictional) death. Following an update to the game content, a new enemy character, Hakkar, was introduced. This character had the ability to inflict a disease called Corrupted Blood upon the playing characters that would drain their health over a period of time. This disease could be passed from player to player, just as in the real world, and had the potential to kill any character contracting it. This effect was meant to be strictly localised to the area of the game that Hakkar inhabited.

However, one thing was overlooked: players were able to teleport to other areas of the game while still infected and pass the disease onto others – which is exactly what happened. I can't find any figures on the body count, but entire cities within the game world were no-go areas, with dead player's corpses littering the streets. Fortunately, player death is not permanent in WoW and the event was soon over when the administrators of the game reset the servers and applied further software updates. Particularly interesting is the way players reactions in the game could closely reflect their reactions to a similar real-life incident. While the deaths in world of War craft were fictional this has not always been the case as the next example will show.

Therac-25 (1985-1987) - The Therac-25 was a machine for administering radiation therapy, generally for treating cancer patients. It had two modes of operation. The first consisted of an electron beam targeted directly at the patient in small doses for a short amount of time. The second aimed the electron beam at high energy levels at a metal 'target' first, which would essentially convert the beam into X-rays that were then passed into the patient.

In previous models of the Therac machine, for this second mode of operation, there were physical fail-safes to ensure that this target was in place as, without it, very high energy beams could be mistakenly fired directly into the patient. In the new model, these physical fail-safes were replaced by software ones.

Unfortunately, there was a bug in the software: an 'arithmetic overflow' sometimes occurred during automatic safety checks. This basically means that the system was using a number inside its internal calculations that was too big for it to handle. If, at this precise moment, the operator was configuring the machine, the safety checks would fail and the metal target would not be moved into place. The result was that beams 100 times higher than the intended dose would be fired into a patient, giving them radiation poisoning. This happened on 6 known occasions, causing the later death of 4 unfortunate patients.

Knight Capital Group's trading violations (2012) - In August 2012 Knight Capital Group Inc., which is one of America's largest trading firms, mistakenly sent out more than four million stock orders in less than one hour. These orders should have been spread out over a period of days—and reversing the trades cost almost half a billion dollars. Knight Capital would have been sent into bankruptcy had it not been for a group of investors that saved the day and came up with \$400 million. The problem was that when a code change was released it was not deployed to all the servers, one server was missed, and this caused the server to use old code to create millions of orders. As a result of this error the firm's shares lost 75% in just two days after the faulty software flooded the market with unintended trades, sending dozens of stocks into spasms. The software bug caused over \$440 million in losses, which is almost four times what the company had made in 2011.

The Cold War Missile Crisis (September 26, 1983) - Stanislav Petrov was the duty officer of a secret bunker near Moscow responsible for monitoring the Soviet early warning satellite system. Just after midnight, they received an alert that the US had launched five Minuteman intercontinental ballistic missiles. As part of the mutually assured destruction doctrine that came into prevalence during the Cold War, the response to an attack by one power would be a revenge attack by the other.

This meant that if the attack was genuine, they needed to respond quickly. However, it seemed strange that the US would attack with just a handful of warheads: although they would cause massive damage and loss of life, it wouldn't be even nearly sufficient to wipe out the Soviet opposition. Also, the radar stations on the ground weren't picking up any contacts, although these couldn't detect beyond the horizon because of the curvature of Earth, which could have explained the delay.

Another consideration was the early warning system itself, which was known to have flaws and had been rushed into service in the first place. Petrov weighed all these factors and decided to rule the alert as

a false alarm. Although Petrov didn't have his finger on the nuke button as such, had he passed on a recommendation to his superiors that they take the attack as real, it could have led to all-out nuclear war. Whether based on experience, intuition, or just luck, Petrov's decision was the right one.

It was later determined that the early detection software had picked up the sun's reflection from the top of clouds and misinterpreted it as missile launches.

So while most software bugs are annoying and short lived others can have very large and serious repercussions on human life. Not only have software bugs caused death and severe injury in the most extreme cases they have also allowed security breaches to banking systems and government systems. No one knows how much money has been stolen by exploiting these glitches but the total probably runs into trillions.

Therefore software testing is very important; in fact it is very, very important. As a result good, motivated and highly trained software testers are also a very important part of the software development team. In many software development companies the tester used to be someone from administration or the post room who was not very busy that week, thankfully most of the software companies who used this strategy have either gone out of business or have changed their methods. In these more advanced and enlightened times testers are usually held in more esteem.

So what is a bug? It is a defect, a flaw in code, software or documentation that can cause the said artifact to fail to perform its required function. When executed such a defect could cause the software to fail with unexpected and potentially dangerous results. A defect can nearly always be traced back to human error.

This could be an error in design or in code or in testing or even in hardware specification, for example if the server does not have enough resources to handle the work put through it then it was under specified, a human error. No matter what the root cause was and what type of human error caused it the outcome will always be called computer error by the press and the company who released the software.

So what is the role of testing in software development, maintenance and operations? Basically the role of the tester is to improve the quality of the software under test by finding defects and logic issues that can then be corrected. These corrections are then tested to confirm the risk of operational problems has been reduced and the quality of the software has therefore increased. The location and correction of bugs will increase confidence in the software and helps the software company meet contractual, legal requirements and any industry-specific standards.

Testing is part of quality assurance; it can help measure the quality of the software and determine if it is fit for purpose. Metrics are available for reporting if all bugs are recorded by their type, severity and priority. These metrics can help identify useful trends and highlight any potential weak points within the team.

The life cycle of each bug is also captured from creation through to resolution, this builds confidence in the process and lessons learnt can help improve the quality of the processes used in design, development and testing.

The quality of testing is also important, a common mistake is to assume that if no bugs are found then no bugs exist within the software under test. It is also very possible that the bugs exist and they have simply not been found yet because the scope of testing is too narrow and not all available logic paths have been checked.

One thing is certain though; if you do not find them during testing the end users or customers will find them when they are using the system on live data and the implications can be very serious indeed. Therefore experienced and competent testers are a vital part of the team.

They know what questions to ask, where to look, what boundaries to probe and how to test. Because of their knowledge they are also able to teach the less experienced and new testers and they should always be present at important development meetings. Development meetings are a vital part of the development life cycle. All interested parties should be present including the complete development team, which includes all of the testers.

An important topic during these meetings is how much testing is required to mark the product as done, fit for purpose and safe. There is no simple answer to this question but a vital consideration is what risks are involved. As already seen some systems have risks involving human life, others have financial and security risks, others have much lower risks, however a risk is a risk and they should always be considered.

So, how much testing should be considered enough?

- It depends on the risks that have been identified for the system
 - This includes technical and business risks
 - Human and other animal health risks
 - Financial risks
 - Customer confidence risks.
- It also depends on project constraints
 - Time
 - Budget
 - Manpower

Therefore testing is necessary because software is likely to have defects during the early stages of the development cycle. Testing the software not only helps locate these bugs but it also helps build a good level of confidence in the reliability of the system. Reducing the number of bugs also reduces the risk and helps to avoid the development company from facing litigation, financial loss and ultimately going out of business.

However resources are always finite and time is always at a premium, therefore risk must always be identified and prioritised as early as possible. The only way to do this is through teamwork, regular meetings and feedback from all parties including the stakeholders. These meetings are discussed in more detail later in the book. This enables the team to determine what to test first what to test most how thoroughly to test each item i.e. where to place emphasis and what not to test (this time).

Eight important principles of software testing to be considered

- Software testing will show the presence of defects.
- Exhaustive testing is impossible, there is not enough time left in the universe.
- Testing reduces the probability of undiscovered defects remaining in the software but finding no defects is not a proof of overall correctness.
- Testing everything (all combinations of inputs and preconditions and all logic paths) is simply not feasible. Instead, risk analysis and priorities should be used to focus testing efforts in specific areas,
- To find defects early, testing activities should be started as early as possible in the software or system development life cycle, and should be focused on defined objectives.
- If the same tests are repeated over and over again, they will no longer find any new defects. To overcome this, the existing test cases will need to be reviewed and revised, to exercise different parts of the software.
- Testing effort should be focused proportionally to the expected and later observed defect density of modules. Finding and fixing defects does not help if the system built is unusable and does not fulfill the user's needs and expectations.

- Testing is done in different contexts. For example, safety-critical software is tested differently from an e-commerce site and a medical system will be tested differently to the latest game release. A good understanding of the system under test is vital.

The most visible part of testing is the actual test execution. This is what most people will associate with testing. But to be truly effective and efficient, test plans should also include time to be spent on planning the tests, designing the test cases to be used, preparing the system for the test execution and finally evaluating the end results. Therefore for each development life cycle a full test plan is required. A test plan is a guide to how the test strategy and project test plan apply to the software under test. It is important to document any exceptions to the test strategy, e.g. only one test case design technique needed for this functional area because it is less critical than the overall project. Both dynamic testing and static testing can be used as a means for achieving similar objectives, and will provide information that can be used to improve both the system being tested and the development and testing processes, the actual methods available will be discussed further on in this book.

An important consideration about testing is that different viewpoints of testing will take different objectives into account. By example, in development testing (e.g., component, integration and system testing) the main objective may be to cause as many failures as possible so that defects in the software code are identified and can be fixed before final release.

By contrast in acceptance testing, the main objective may be to confirm that the system works as expected, has a good, well designed user interface and will allow interested parties to gain confidence that the system has met all of the requirements and is in a state of done. In some cases the main objective of testing may be to assess the quality of the software (with no intention of fixing defects), to give information to stakeholders of the risk of releasing the system at a given time.

Maintenance testing often includes testing that is designed to ensure no new defects have been introduced during the latest development cycle, merging of changes and bug fixes. During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

The test team also needs to be aware that debugging and testing are in fact different entities. Dynamic testing can show failures that are caused by defects if implemented correctly.

In contrast debugging is the development activity that finds, analyzes and removes the cause of the failure. Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure. The responsibility for these activities is usually testers test and programmers debug. You should always remember that both are developers in the agile world.

Test control is the ongoing activity of comparing actual real time progress against the original test plan, and reporting the status, including deviations from the plan at regular meetings. Test control involves taking actions necessary to meet the mission and objectives of the project. In order to control testing, the testing activities should be monitored and discussed throughout the project. Test planning takes into account the feedback from monitoring and control activities.

At the end of every cycle or sprint test closure activities should be employed to collect data from the completed test activities to consolidate experience, facts, mistakes and numbers. Test closure activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed or in the agile world at the end of every sprint. Typical closure activities include the following tasks:

- Checking which planned deliverables have been completed and delivered.
- Closing incident reports or raising change records for any that remain open.
- Documenting the acceptance of the system.
- Finalizing and archiving test software, the test environment and the test infrastructure for reuse in later sprints.
- Analyzing lessons learned to determine changes needed for future releases and projects.
- Using the information gathered to improve test maturity of the team.

So that is a brief introduction into the foundations of software testing. You are still here and still reading so hopefully I have not frightened you off yet. The question is though, are you good enough to be a software tester? Do you still think this is the correct career path for you? Hopefully the next chapter will answer these questions and more for you, please read on and enjoy as we look into the testers mindset.

Most people can test software at a very basic level. After all anyone should be able to click buttons, look at web pages, enter text and press [Save]. However being able to test software proficiently over many years at an expert level requires a very special type of mindset. Not everyone has this ability, and there is no shame in this, not everyone can race cars or fly an airplane either. A software tester does and must have a very different mindset to that of a software programmer and while both do not always mix well their interaction and cooperation within a development team is crucial in the agile world. Some programmers are able to test their own code and indeed all programmers should test at a unit level before sending updates out for test. However separation of this responsibility to a professional, well trained tester is typically done to help focus effort and provide additional benefits, such as an independent view by trained and professional testing resources. Independent testing can and should be carried out at every level of testing.

A certain degree of independence (also known as avoiding the author bias or programmer arrogance) often makes the tester more effective at finding defects and failures. Independence is not, however, a replacement for familiarity and the best testers are often those who are familiar with the product under test rather than testers who have simply been bought in for a single project.

Several levels of independence can be defined as shown here from low to high:

- Tests designed by the programmer(s) who wrote the actual software under test (low level of independence).
- Tests designed by another programmer(s) within the same programming team.
- Tests designed by a person(s) from a different organizational group (e.g., the test team) or test specialists.
- Tests designed by a person(s) from a different organization bought in just to test the product.

Testing can sometimes be a tedious and repetitive undertaking and not everyone is able to handle this task for very long, when you see a tester banging the head on the their desk or throwing darts at photographs of the programmers it may well be time to move them elsewhere within your organisation. A good tester will be methodical in their approach and will have good written and verbal communication skills. To them each day is a fresh, interesting new challenge. They will have a lot of patience and be able to maintain a high level of concentration throughout the day. Also when writing up their conclusions of a recent test they need to be able to explain their findings in a complete, literate but sensitive manner, any hint of trying to indicate a programmer has done a poor job should always be avoided even if it is true.

Such comments only lead to divisions within a company and these are ultimately destructive. Also making a programmer feel as though you enjoy finding fault with their work is a fast way of isolating the programming and testing teams.

If the worst does happen and such divisions do become established they can be impossible to remove until certain people are removed from the firing line or company structure.

Programmers are a highly skilled but sometimes temperamental group of people, many of whom consider themselves the intellectual tip of the company iceberg that they work for and as such an untouchable elite that should not be disrespected or upset in any way. However the world is changing and as I have already stated the world is awash with highly skilled and talented programmers.

The rise of the software tester is now unstoppable and their importance within the software development world is now indisputable. So how do you incorporate these two groups of very different and potentially at war teams with one software development company?

Probably the best solution is to reinforce the team ethos and drive home the fact that everyone is working to the same end game. Software development is a team game. The development team is made up of programmers, testers, product owners and stakeholders. Testers are as important as any other member of the team but not more important than anyone else. Agile is a great tool for bringing these ideas forward and that is why this book was written.

Regular meetings are an important part of building the team spirit it is also essential to ensure a well designed, well programmed and well tested software package. Stakeholder involvement is also essential at every stage of the development cycle and I have always involved the end user throughout the complete life cycle. Thankfully the agile model enforces this idea home fully. It is important to remember that people and projects are usually driven by known objectives and deliverables. People tend to align their plans with the objectives set by management and other stakeholders, for example, to find defects or to confirm that software meets its objectives. Therefore, it is important to clearly state the objectives of testing at an early stage.

Identifying failures during testing may be and often is perceived as criticism against the product and against the programmer who wrote it. As a result, testing is often seen as a destructive activity in some eyes, even though it is very constructive in the management of product risks. Looking for failures in a system requires curiosity, professional pessimism, a critical eye, attention to detail, good communication with development peers, and experience on which to base error guessing. These are a very special set of skills that make experienced software testers a very valuable commodity.

If errors, defects or failures are communicated in a constructive way, bad feelings between the testers and the analysts, designers and programmers can be avoided. This applies to defects found during reviews as well as in testing. The hardest group to keep happy in these circumstances is the programmers; as previously mentioned they tend to be a very sensitive and temperamental breed.

The testers and test leader will need very good interpersonal skills to communicate factual information about defects, progress and risks in a constructive and non-destructive way. For the programmer of the software good information can help them improve their own skills. Defects found and fixed during testing will save time and money later, and reduce risks.

Unfortunately communication problems can occur, particularly if testers are seen only as harbingers of doom and unwanted news about defects. However, there are several ways to improve communication and relationships between testers and others:

- Collaborate: Start with team collaboration rather than battles – remind everyone in the team that the common goal is better quality software systems and also that they are all of equal importance within the team.
- Communication: Communicate findings on the product in a neutral, fact-focused way without criticizing the programmer(s) who created it, for example, write objective and factual incident reports and review findings.
- Understanding: Try to understand how the other person feels and why they react as they do. Get to know the people in your team.
- Confirmation: Confirm that the other person has understood what you have said and vice versa.

So that's what it takes to be a good tester.

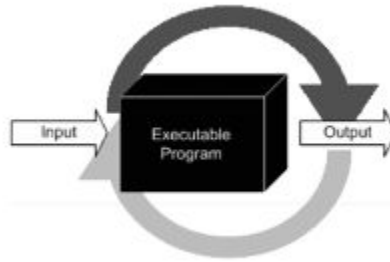
So do you still think you can be a software tester?

If yes then well good for you, you may well have a long, fulfilling future in the world of software development. Now let's look at the common types of testing next.

So before we move on to the world of agile software development and in particular agile software testing let us discuss the major types of testing currently in use today. This is not a complete, exhaustive list and I am sure some of you will all know of a few other types and want to say 'what about ...' but please remember this book has already been written and published. Also I cannot hear you but please feel free to email any suggestions for the next edition. However most of the common styles are here, you will not use all of them but you will come across most during your careers in IT.

These days software testers have a good array of testing methods and tools at their disposal. These can be generally classified within the Dynamic and Static areas. Static testing is based around the methods used for Code Reviews, Walkthroughs and Inspections. Dynamic testing methods involve a developer or tester actually using the computer program or parts of it. Types can also be split between Black Box and White Box as detailed below.

Black box testing



Black box testing is a method of software testing where the test team are not required to know or understand the code and internal structure of the software on test. For example, in a black box test on software design the tester only knows the required inputs and what the expected outcomes should be and not how the program code arrives at those outputs. The test team will never examine the actual programming code and they do not need any further in depth knowledge of the program other than its specifications.

The advantages of this type of testing include:

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need knowledge of any specific programming languages.
- The test is done from the point of view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.

Discussed next are the most common types of Black Box Testing

Acceptance testing is a formal type of software testing that is performed by end user when the features have been delivered by developers. The aim of this type of testing is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery. Finding defects is not the main focus in acceptance testing. In some organisations this is also known as Beta Testing. Acceptance tests are normally documented at the beginning of the sprint (in agile) and is a means for testers and developers to work towards a common understanding and shared business domain knowledge.

Acceptance testing may occur at various times in the life cycle, for example:

- A COTS (Commercial off the shelf) software product may be acceptance tested when it is installed or integrated.
- Acceptance testing of the usability of a component may be done during component testing.
- Acceptance testing of a new functional enhancement may come before system testing.

Accessibility Testing is designed to ensure the contents of the website can be easily accessed by disabled people. Accessibility testing is very similar to usability testing, in that it is about making sure that the website or application under test is easy for its intended audience to use. That audience includes users who access the service via a range of assistive technologies like:

- screen readers
- voice recognition software
- trackball devices

It's important to consider a range of disabilities when you are testing any website or application service, including those with:

- cognitive and learning disabilities, e.g. dyslexia or attention deficit disorders
- visual impairments, e.g. total and partial blindness, colour blindness, poor vision
- auditory disabilities, which can also affect language
- motor skills impairments, e.g. those affected by arthritis, strokes, RSI

Ad-hoc testing, this form of software testing is usually very informal and unstructured and can be performed by any stakeholder without any reference to any test case or test design documents. The person performing Ad-hoc testing should however have a good understanding of the domain and workflows of the application in order to find defects and attempt to break the software. As a result Ad-hoc testing is intended to find defects that were not found by existing test cases.

Agile Testing is a type of software testing that accommodates agile software development approach and practices. In an Agile development environment, testing is an integral part of software development and is done along with coding in monthly sprints. Agile testing allows incremental and iterative coding and testing and relies on a well organised and development cell.

All Pairs Testing is also known as Pair wise testing. This is a black box testing approach that can be done by software testers, developers, business analysts or any other interested stakeholder. It is a combinatorial testing method that, for *each pair* of input parameters to a system (typically, a [software algorithm](#)), tests all possible discrete combinations of those parameters. By the careful selection of [test vectors](#), this can be done much faster than an exhaustive search of [all combinations](#) of all parameters, by "parallelizing" the tests of parameter pairs.

Automated testing is a testing approach that makes use of testing tools and/or programming to run the test cases using software or custom developed test utilities. Most of the automated tools provided capture and playback facility, however there are tools that require writing extensive scripting or programming to

automate test cases. One of the best tools currently available is [Selenium](#) which can be configured to work with Java and .Net. [Selenium](#) is a fascinating and very useful tool that will be discussed later in this book.

Backward Compatibility Testing is a type of software testing performed to check that a newer version of the software can work after being successfully installed over a previous version of the same software. Also this test will confirm that the newer version of the software works as fine with table structures, data structures and files that were created by previous version of the software.

Boundary Value Testing (BVT) is a testing technique that is based on the proven concept that “error aggregates at boundaries”. In this testing technique, testing is done extensively to check for defects at known boundary conditions. A classic example of this is if a field accepts values from 1 to 99 then testing is done for values 0, 1, 2, 98, 99 and 100 because these are the boundaries. In this example 1, 2, 98, 99 are valid while 0 and 100 would be invalid. Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is chosen.

Boundary value analysis can be applied at all test levels. A good feature of this test is that it is relatively easy to apply and its defect-finding capability is high. Detailed specifications are helpful in determining the interesting boundaries.

Bottom up Integration testing is an integration testing approach where the testing cycle will start with smaller pieces or sub systems of the whole software and gradually build its way up until testing is covering the entire software system. The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These units are exercised through their interfaces using black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test whether all the components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a building block approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Browser compatibility Testing is one of the sub types of testing of compatibility testing performed by the test team. Browser compatibility testing is performed for web applications with a combination of different browsers and operating systems. This is also referred to as User Experience Testing and is designed to ensure the following.

- Users have the same visual experience irrespective of the browsers through which they view the web application.

- In terms of functionality, the application must behave and respond the same way across different browsers and different operating systems.

Decision Table Testing methods are a very effective way to capture system requirements that contain logical conditions (e.g., True or False), and to document internal system design. They can also be used to record complex business rules that a system has to implement. When creating decision tables, the specification is analysed, and conditions and actions that the system must meet are identified. The input conditions and actions are most often stated in such a way that they must be true or false. The decision table contains the triggering conditions, often combinations of true and false for all input conditions, and the resulting actions for each combination of conditions. Each column of the table corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule. The coverage standard commonly used with decision table testing is to have at least one test per column in the table, which typically involves covering all combinations of triggering conditions.

End-to-end testing is performed by the testing team. The focus of end to end testing is to test end to end flows e.g. right from new order creation until order completion and reporting of created orders which are now complete. End to end testing is usually focused on mimicking real life scenarios and usage. End to end testing involves testing information flow across applications.

Equivalence Partitioning is also known as Equivalence Class Partitioning. This is specialised software testing technique and not a complete type of testing by itself. Equivalence partitioning technique is used in black box testing types. Equivalence partitioning classifies test data into Equivalence classes as positive Equivalence classes and negative Equivalence classes, such classification ensures both positive and negative conditions are tested. Partitions can also be identified for outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing). Tests can be designed to cover all valid and invalid partitions. Equivalence partitioning is applicable at all levels of testing. Equivalence partitioning can also be used to achieve input and output coverage goals. It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing.

Exploratory Testing is another informal type of testing which is conducted by testers to learn and understand the software while at the same time looking for errors or any application behaviour that seems non-obvious or incorrect. Exploratory testing is usually done by testers but it can also be done by other stake holders as well like business analysts, developers, end users etc. If fact anyone who is interested in learning functions of the software and at the same time looking for errors or behaviour is seems non-obvious is a potential tester for this method.

Functional testing will nearly always be required during a software test phase. There are essentially two types of functional testing; these are Full program testing and Change testing.

Functional tests are based on the functions and features (described in documents or understood by the testers) within the system and their interoperability with specific systems, and may be performed at all test levels (e.g., tests for components may be based on a component specification).

Full program testing is most commonly used when a module is being tested prior to its first release into production. The test document created will be a complete step by step test of the module. Every process should be carefully tested and the results analyzed and fully documented.

In contrast change testing is used when a previously published module has undergone changes, bug fixes or enhancements. These changes may be very small or very significant. No matter what the scope of change the same processes and attention to detail should always apply. To successfully complete this type of testing the test team will require a copy of the change test document and at least one test case document.

When testing a web based application all functional testing (both types) should be carried out in all of the most common browsers, such as Firefox, Chrome, Safari and Internet Explorer. The browsers and versions used should also be documented in the test document. This will ensure that anyone reading the document in the future will be fully aware of which browsers and versions were used during the test. The tester should fully detail the functional test by outlining each process with as much detail as is required to confirm the thoroughness of the testing. Screenshots should also be included, these provide a very informative graphic view which backs up the textual description of the test. The order in which the test is completed is not important as long as every aspect is covered.

The aim of functional testing is to attempt to detect as many defects as possible before the package reaches the Beta stage and is given to the customers testing team. While it should be accepted that it is impossible to capture every bug within a program providing the customer with a Beta version that is very functional and reliable will enhance their view of the product and this in turn will help them eventually accept the final product with a good degree of confidence.

Fuzz Testing or fuzzing is a software testing technique that involves testing with unexpected or random inputs. Software is monitored for failures or error messages that are presented due to the input errors.

Integration Testing is also known as I&T in short, this is one of the more important types of software testing. Once the individual units or components have been unit tested by developers and confirmed as working then the testing team will run tests that will test the connectivity among these units/component or multiple units/components. There are different approaches for Integration testing as shown below:

- **Big Bang Integration testing** is one of the integration testing approaches, in Big Bang integration testing all or all most all of the modules are developed and then coupled together.
- **Incremental Integration Testing** is a bottom up approach for continuous testing of an application as new functionality is added; Application functionality and modules should be independent enough to test separately. This type of testing is done by programmers and/or by testers.
- **Component integration** testing tests the interactions between software components and is done after component testing
- **System integration** testing tests the interactions between different systems or between hardware and software and may be done after system testing.

An important consideration here is that the greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

Systematic integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to facilitate fault isolation and detect the defects early, integration should ideally be incremental rather than “big bang”. Testing of specific non-functional characteristics (e.g., overall performance) may be included in integration testing as well as functional testing.

At each stage of the integration testing phase, testers concentrate fully on the integration itself. For example, if the team are integrating module X with module Y then they will be interested in testing the communications between the two modules, not the functionality of the individual module as that was done during component testing. Both functional and structural approaches may be used.

To help test efficiently the testers would ideally understand the architecture of the design and they should be allowed to influence the integration planning. If integration tests are planned before components or systems are built, those components can be built in the order required for most efficient testing.

Logical Access Testing is required when testing web based applications and in these circumstances logical access is a vitally important part of the test regime. Testing should be broken into two distinct sections; the first section is Zero Access.

Zero access is the operation of testing that a URL can only be reached after a user has successfully logged into the application using a valid User ID and a valid password, otherwise they should be returned to the login page or a pre-defined warning page. An effective method for preparing for this test is to record all URL's that the tester encounters during the Functional test stage. It is vital that every possible URL

is gathered during the functional test and then tested during this stage and this method helps reduce the number of missed URL's.

The method most often used is to log out of the application and then to paste each URL into the address bar and check the response. Your desired response will probably be for the user to be sent to a log in screen or a pre-defined error page. What you do not want is for the user to be allowed system access after they have logged out of the system.

The second section of logical access testing is Profile Access. This section is more complex than Zero Access and requires more thought while testing. In this section you can assume the user has logged in correctly and you should use the same URL's noted in functional testing as you used in the Zero Access section however this time do not paste them into the address box but actually navigate to them. In this section the questions to be answered are:

1. Does the system allow the user access to a module that their profile states they should have access to? This should be tested by logging into the system and navigating to the required module or section.
2. Next this test should be undertaken on a user profile that does not have access to the module. The test should be undertaken in the same manner as 1 and both tests with results should be fully documented in the test document.

How far you take profile testing will of course depend on the complexity of your system. If multiple modules are dependant of profile access then each module will need testing. If different users have different access rights to certain actions such as create records, deleting records and running reports then these will all have to be tested. Also if user profile's also defines a user's geographical access then this must also be tested, as demonstrated next.

Profile testing access for a particular Country/Region. Take for example the URL below.

<http://www.yourapplication.com/customeraccounts/welcome.do?cty=de>

This address is for a customer accounts module on a test server and the Country is Germany (de). Testing should be undertaken on this address with user profiles that allow and do not allow access to the given Country. To confirm this procedure another Country should also be tested in the same way and all of the results should be fully documented on the test document.

These steps are very important. They confirm to the project leader and the customer that profile access is secure at a Module and a Geographical level. This type of testing is ideally suited to automation and we will discuss this later in the Selenium section.

Locale Testing Another important type of test for web based applications is locale testing. This is another type of test that is becoming more common as web based applications grow in number and the potential user base becomes global. Locale testing will check the quality of your applications localization for a particular target culture/locale. While it is impossible to test every potential global locale you should certainly test a good sample to confirm the process appears to be reliable.

Maintenance Testing After the initial deployment, a good software system is often in service for years or even decades. During this time the system, its configuration data, or its environment are often corrected, changed or extended. The careful planning of releases in advance is crucial for successful maintenance testing. A distinction has to be made between planned releases and bug fixes. Maintenance testing (retesting) is done on an existing operational system, and is triggered by modifications, migration, or retirement of the software or system.

Modifications include planned enhancement changes (e.g., release-based), corrective and emergency changes, and changes of environment, such as planned operating system, web browser or database upgrades, planned upgrade of linked Commercial-Off-The-Shelf software, or patches to correct newly exposed or discovered vulnerabilities of the operating system or web browsers.

Maintenance testing for migration (e.g., from one platform to another or from one server to another) should include operational tests of the new environment as well as of the changed software. Migration testing is also needed when data from another application will be migrated into the system being maintained.

In addition to testing what has been changed, maintenance testing includes regression testing to parts of the system that have not been changed. The scope of maintenance testing is related to the identified risk of the change, the size of the existing system affected and to the size of the change.

Depending on the changes to be implemented, maintenance testing may be done at any or all test levels and for any or all test types.

Determining how the existing system may be affected by changes is called impact analysis, and is used to help decide how much regression testing to do. This is a critical exercise that should always be undertaken. The impact analysis may be used to determine the regression test suite.

Negative Testing is a type of software testing which calls out the “attitude to break”; these are functional and non-functional tests that are designed to break the software by entering incorrect data like incorrect date, time or string or attempt to upload a binary file when a text file is supposed to be uploaded. Another method is to enter huge text strings for input fields etc. The core difference between positive testing and negative testing is that throwing an exception is not an unexpected event in the latter. When you perform negative testing, exceptions are expected – they indicate that the application handles improper user behaviour correctly. It is generally considered a good practice to combine both the positive and the negative testing approaches

Non-Functional testing refers to aspects of the software that may not be related to a specific function or user action, such as usability or performance. Non-functional testing tends to answer such questions as ‘how well does the system perform when I save a new record?’

Non-functional testing can include (but is not limited to): Usability, Robustness, Compatibility, Performance, Load, Stress, Endurance, Stability, Accessibility, Extensibility, Scalability, functionality and Portability. Generally speaking, it is the testing of 'how well' the system works under normal usage.

The aim of non-functional testing is to verify that the software functions correctly even if the input is invalid or unexpected. The software should handle such issues in a controlled manner and respond with meaningful error messages. The program should not crash. Non-functional testing is also concerned with how well the software works when under load. Does the screen hang for a long while when connecting to a database? Are there delays when switching screens? Such tests are also known as Stress Testing and Load Testing. Whatever the title is, if the software slows down under load then this affects users working speed and they will soon become unhappy with the product. Companies will also be concerned that productivity will be compromised and they may well eventually reject the package as not fit for purpose.

Non functional testing and functional testing are usually completed at the same time. Some development teams will document them separately while others will merge both types into the same part of the document. Neither method is actually wrong as long as the testing is complete and correctly recorded.

Regression Testing is any type of software testing that seeks to uncover new errors, or *regressions*, in existing functionality after changes have been made to the software, such as functional enhancements, bug fixes or configuration changes. This is sometimes confused with maintenance testing and you may wonder how they differ? In regression testing you retest the test case, therefore you could consider regression testing to be a subset of maintenance testing/retesting. However in the real world this is mostly a semantics issue. More commonly different teams will use "maintenance testing " and "regression testing" interchangeably.

Changes made to existing software packages is a common way of introducing software bugs. These bugs will quite often appear in areas of the program that have not been altered by recent changes but often due to shared classes, once stable sections of code are suddenly unstable. This is why regression testing should always be performed alongside Confirmation Testing. This form of testing can be tedious if a tester is expected to complete this function after every upgrade. It is however essential if the software is to be released in a stable fashion.

The intent of regression testing is to assure that a change, such as a bug fix, did not introduce new bugs and that the base functionality of the application has not been broken. This form of testing is not required if the module under test is a new application that has not been previously released, however existing modules will need regression testing.

The most common method of regression testing is rerunning previously run tests. The tester should locate the most recent complete test document for the module as well as the last three change test documents (less if only one or two previous documents exist). This is one reason why every test should be recorded and stored at a secure location.

Regression testing involves retesting the unchanged parts of the module and to achieve this goal the tester should step carefully through the previous test documents checking that the results are still the same. The steps need to be carefully mirrored and should be completed in Internet Explorer (IE), Firefox and other popular browsers. Which browsers were used should also be documented.

Any crashes, unexpected results or strange responses should be reported to the programmer via your company's usual bug reporting facility, my personal preference is Bugzilla but this is simply one of many options. When a fix is released for a reported bug in regression testing a new full regression test should then be started to ensure that the new fix has not introduced yet more new errors. This process should continue until a complete successful regression test is recorded.

Penetration Testing is a type of security testing. This is also known as pentest in its shortened name. Penetration testing is undertaken to test how secure software and its environments (Hardware, Operating system and network) really are when subject to attack by an external or internal intruder. The intruder can be a human/hacker or a malicious program. Pentest uses methods to forcibly intrude (by brute force attack) or by using a weakness (vulnerability) to gain access to a piece of software or its data or hardware with the intent to expose ways to steal, manipulate or corrupt the data, software files or configuration. Penetration Testing is a way of ethical hacking, an experienced Penetration tester will use the same methods and tools that a hacker would use but the intention of Penetration tester is to identify vulnerability and get them fixed before a real hacker or malicious program exploits it.

Risk based Testing is a type of software testing and a different approach towards testing developing software. In Risk based testing the requirements and functionality of the software to be tested are prioritized as Critical, High, Medium and low. In this approach, all critical and high priority tests are tested first and then this is followed by the medium tests. The low risk functionality is then tested at the end. However these may not be tested at all if there is no further time available for testing.

Sanity Testing is a type of testing that is carried out mostly by testers and in some projects by developers as well. Sanity testing is a quick evaluation of the software, environment, network, external systems are up & running, software environment as a whole is stable enough to proceed with extensive regression testing. Sanity tests are narrow and most of the time sanity tests are not documented however they do help to avoid wasting time and cost that is incurred in testing if the build has failed.

Smoke testing is a type of testing that is carried out by software testers to check if the new build provided by the development team is stable enough i.e., major functionality is working as expected in order to carry out further or detailed testing. Smoke testing is intended to find “show stopper” defects that can prevent testers from testing the application in detail. Smoke testing carried out for a build is also known as ‘build verification test’. This type of testing is very similar to Sanity Testing.

Soak Testing is a special type of performance testing, where the software under test is subjected to load over a significant duration of time, soak testing may go on for a few days or even for a few weeks. Soak testing is a type of testing that is conducted to find errors that result in degeneration of software performance with continued usage. Soak testing is extensively done for electronic devices, which are expected to run continuously for days or months or years without restarting or rebooting. With growing web applications soak testing has gained significant importance as web application availability is critical for gaining and then sustaining customer confidence which will help ensure the success of the business in question.

System Integration Testing is also known as “SIT” and is a type of testing conducted by software testing teams. As the name suggests, focus of System integration testing is to test for errors related to integration among different applications, services, third party vendor applications etc. As part of SIT, end-to-end scenarios are tested that would require software to interact (send or receive data) with other upstream or downstream applications, services, third party application calls etc.

System Testing is a test exercise that is concerned with the behaviour of the complete and whole system or product under development. The testing scope shall be clearly addressed and defined in the Master and/or Level Test Plan for that test level. In system testing, the test environment should correspond as closely as possible to the final target or production environment platform in order to minimize the risk of environment-specific failures not being found in the testing phase.

System testing can include tests based on risks and/or on requirements specifications, business processes, user stories and other high level text descriptions or models of system behaviour, interactions with the operating system, and system resources.

System testing is comprehensive and should investigate functional and non-functional requirements of the system, and data quality characteristics. The Testers also need to deal with incomplete or undocumented requirements as they find them. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. Structure-based techniques (white-box) can then be used to assess the thoroughness of the testing with respect to a structural element, such as menu structure or web page navigation.

Volume testing is a non-functional type of testing carried out by performance engineering teams. Volume testing is one of the types of performance testing. Volume testing is carried out to find the response of the software with different sizes of the data being received or to be processed by the software. For e.g. If you were to be testing Microsoft word, volume testing would be to see if word can open, save and work on files of different sizes (10 to 100 MB).

Vulnerability Testing involves identifying and exposing the software, hardware or network vulnerabilities that can be exploited by hackers and other malicious programs such as viruses or worms. Vulnerability Testing is a key test for software security and availability. With a huge increase in the number of hackers and malicious programs worldwide, Vulnerability Testing is now critical for the success of a Business. This test is often used on conjunction with Penetration testing.

White box Testing

White box testing can also be known as clear box testing, transparent box testing and glass box testing. White box testing is a software testing approach, which intends to test software with knowledge of the internal code and intended working of the software.

Typically the white box testing approach is used in Unit testing which is usually performed by software developers. White box testing intends to execute code and test statements, branches, path, decisions and data flow within the program being tested. White box testing and Black box testing complement each other as each of the testing approaches have potential to un-cover specific category of errors. Common types of white box testing are discussed next.

API Testing is a type of testing that is similar to unit testing. Each of the Software APIs is tested as per the API specification. API testing is mostly done by the test team unless APIs to be tested is very complex and needs extensive coding. API testing requires understanding both the API functionality and possessing good coding skills.

So you may ask, what is an API? Well API is short for application programming interface, which is a bit of a mouthful so let us stick with API. So an API is a set of programming instructions and defined standards for accessing a Web-based software application or a Web tool. Typically a software company will release its API to the public so that software developers can design new products that are powered by or make use of its service.

A very good example of this is when Amazon.com released its API so that Web site developers could more easily access Amazon's product information databases. Using the Amazon API, a third party Web site can easily post direct links to Amazon products with updated prices and specific options such as "buy now".

Branch Testing is a white box testing method for designing test cases to test code for every branching condition. The branch testing method is typically applied during unit testing.

Component Testing is a type of software test that is performed by developers. Component testing is carried out after completing unit testing. Component testing involves testing a group of units as code together as a whole rather than testing individual functions, methods.

Condition Coverage Testing is a testing technique used during unit testing, where a developer tests for all the condition statements like if, if else, case etc, in the code being unit tested.

Decision Coverage Testing Is a testing technique that is used in Unit testing. The objective of decision coverage testing is to expertise and validate each and every decision made in the code e.g. if, if else, case statements. The decision testing technique derives test cases to execute specific decision outcomes. Branches originate from decision points in the code and show the transfer of control to different locations

in the code. Decision testing is a form of control flow testing as it follows a specific flow of control through the decision points. Decision coverage is stronger than statement coverage; 100% decision coverage guarantees 100% statement coverage, but not vice versa.

Structural Testing is the testing of the structure of the system or component. In structural testing the testers are required to have the knowledge of the internal implementations of the code. Here the testers require knowledge of how the software is implemented, how it works. During structural testing the tester is concentrating on how the software does it. For example, a structural technique wants to know how loops in the software are working. Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.

Structural testing can be used at all levels of testing. Developers use structural testing in component testing and component integration testing, especially where there is good tool support for code coverage. Structural testing is also used in system and acceptance testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

Unit testing (also known as component testing) is a type of white box testing that is performed by software developers whenever they update their code. Using white box testing techniques, testers (usually the developers creating the code implementation) verify that the code does what it is intended to do at a very low structural level. Unit testing usually involves developing stubs and drivers. Unit tests are often ideal candidates for automation. Automated tests can run as Unit regression tests on new builds or new versions of the software.

There are many useful unit testing frameworks like Junit, Nunit etc, available that can make unit testing more effective. When available, the tester will examine the low-level design of the code; otherwise, the tester will examine the structure of the code by looking at the code itself. Unit testing is generally done within a class or a component. Programmers will nearly always press the correct buttons when testing and they always test the sequence in the correct order. Rarely do they carry out destructive testing or integration testing. Unfortunately it is difficult for a tester to carry out unit testing unless they are proficient in programming and have access to the developer's code, currently this is rare.

Component testing may include testing of functionality and specific non-functional characteristics, such as resource-behavior (e.g., searching for memory leaks) or robustness testing, as well as structural testing (e.g., decision coverage). Test cases should be derived from work products such as a specification of the component, the software design or the data model.

One possible approach to component testing is to prepare and automate test cases before the actual coding process. This is called a test-first approach or test-driven development (TDD). This approach is highly iterative and is based on cycles of developing test cases, before building and integrating small pieces of code, and executing the component tests correcting any issues and iterating until they pass.

This may seem a strange way of doing things at first, but with practice it's much more efficient than writing a bucket load of code, running it, and going back later to figure out everywhere it's broken (a process lovingly known as debugging). This process puts the programmer in a testing mindset while writing code, which leads to higher-quality code which it turn makes life easier for the tester.

Static Testing

This is a form of testing that is commonly used in reviews and walkthroughs which are employed to evaluate the correctness of the deliverable. Unlike dynamic testing, which requires the execution of software; static testing techniques rely on the manual examination (reviews) and automated analysis (static analysis) of the code or other project documentation without the execution of the code. The code is reviewed for syntax, commenting, naming convention, size of the functions and methods etc. Static testing usually has check lists against which deliverables are evaluated.

Manual reviews are a good way of testing software work products (including code) and can be performed well ahead of dynamic test execution. Defects that are detected during reviews early in the life cycle (e.g., defects found in requirements) are often much cheaper to remove than those detected by running tests on the executing code.

In most cases a review could be done entirely as a manual activity, but there is also tool support available if required or desired. The main manual activity is to examine a work product and make constructive comments about it. Any type of software work product can be reviewed, including requirements specifications, design specifications, raw code, test plans, test specifications, test cases, test scripts, user guides or web pages.

The benefits of reviews include early (and cheaper) defect detection and correction, improved development productivity improvement, reduced development timescales, reduced testing cost and time, fewer defects in code under test and improved communication between all affected parties. Reviews can also find critical omissions in requirements which are less likely to be found in dynamic testing.

All reviews, static analysis and dynamic testing should have the same objective, which is to find defects. These methods are complementary and the different techniques can find different types of defects effectively and efficiently. In comparison to dynamic testing, static techniques find the causes of failures and defects rather than the actual failures themselves.

The style of the different types of reviews vary from informal, characterized by no written instructions for reviewers, to systematic, characterized by team participation, documented results of the review, and fully documented procedures for conducting the review. The formality of a review process is related to factors such as the maturity of the development process, customer requirements, any legal or regulatory requirements or the need for an audit trail.

A single software product may be the subject of more than one review during its life cycle. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with stakeholders. The main characteristics, options and purposes of common review types are:

Informal Review

- No formal process used.
- These could take the form of pair programming or a technical lead reviewing designs and code.
- The results may (or may not) be documented.
- These can vary in usefulness depending on the reviewers.
- Main purpose: an inexpensive way to get some benefit.

Walkthrough

- The meeting is always led by the author.
- These may take the form of scenarios, dry runs and peer group participation.
- They are open-ended sessions.
- They can include an optional scribe (but this should not be the author).
- They may vary in practice from quite informal to very formal.
- Main purposes: learning, gaining understanding, finding defects.

Technical Review

- A formally documented, defined defect-detection process that includes peers and technical experts with optional management participation.
- These may also be performed as a peer review without management participation.
- Ideally the review will be led by trained moderator (who should not be the author).
- There should be pre-meeting preparation by reviewers.
- There is scope for optional use of checklists.
- These will lead to the preparation of a review report which includes the list of findings, the verdict whether the software product meets its requirements and, where appropriate, recommendations related to findings.
- They may vary in practice from quite informal to very formal.
- Main purposes: discussing, making decisions, evaluating alternatives, finding defects, solving technical problems and checking conformance to specifications, plans, regulations, and standards.

Inspection

- These are led by trained moderator (should not be the author).
- They are usually conducted as a peer examination.
- They must have defined roles.
- They will usually include metrics gathering.
- This is a formal process that is based on rules and checklists.
- There will be specified entry and exit criteria for acceptance of the software product.
- They should always include pre-meeting preparation.
- At the end there should be an inspection report including list of findings.
- Main purpose: finding defects.

Remember that the objective of static analysis is to find defects in software source code and software models. Static analysis of the code is performed without actually executing the software being examined by the tool. Also static analysis can locate defects that are hard to find while dynamically testing the compiled software. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as generated output such as HTML and XML.

Static analysis tools are typically used by programmers before and during component and integration testing. They are also used when checking-in code to configuration management tools and by designers during software modeling. Static analysis tools may produce a large number of warning messages, which need to be very well-managed to allow the most effective use of the tool.



In the beginning the universe was created, and if you are a fan of the late, great Douglas Adams you will know this has made a lot of people very angry and was widely regarded as a bad move. Many years later software development was created (some people also regarded this as a bad move) and as already discussed in the early days very little testing was undertaken.

If you are an experienced, intelligent and sensible person you will know that software development is hard to do well. The question is why? Professionals in the software development business tend to be very clever, bright and hardworking people. As a rule they do not plan to deliver software that is over budget, past its deadline, incomplete or with defects. Despite this these issues still arise time after time. Computers are complex, so are networks. There are many things that can go wrong, software issues in web browsers and operating systems can also affect the stability of your code. Users can get it wrong and hit the wrong combination of keys on the keyboard. These are all risks that need consideration.

Agile was designed as an attempt to make the software development process better, faster and more effective. Since its initial conception the methodology has steadily grown in popularity, usage and success. Before talking about agile let us first consider the older, traditional software methods of Waterfall and V-Model. Both of these methodologies were based on the following assumptions

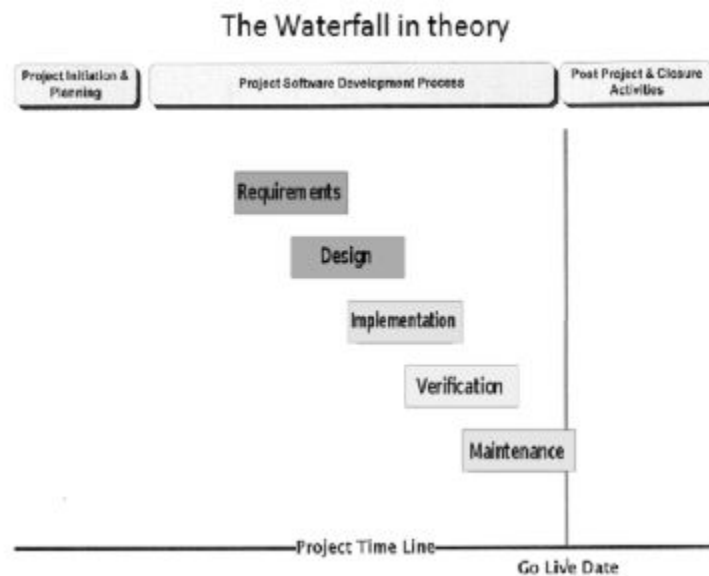
1. The customer knew exactly what they wanted and could clearly articulate upfront what it is they wanted in such a way that everyone involved could clearly understand the requirements.
2. The project would have a set of well defined, clear and unambiguous requirements which had been discussed, considered and agreed by all concerned.
3. Everything that would be needed for the new system was known and available upfront.
4. Once the system was defined it was a set of static requirements and nothing would ever change.

5. Requirements were clearly and professionally detailed up front. The system was then designed, coded and tested in a logical order. Such well defined systems were always completed on time and with no major defects.



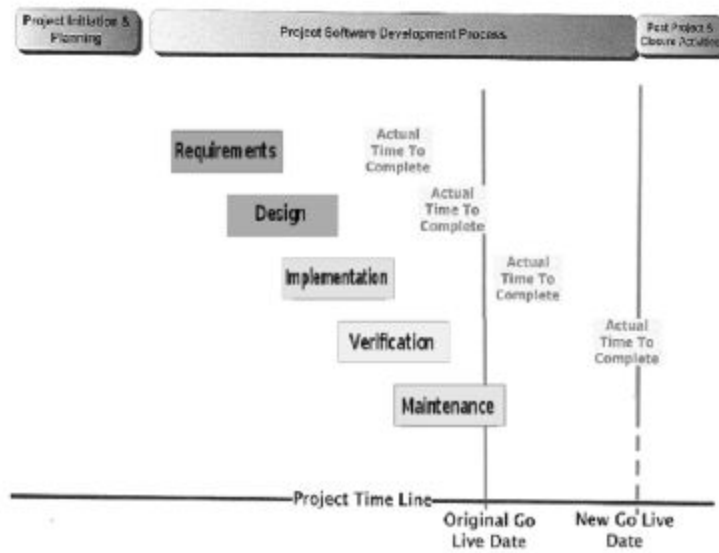
6. And of course.....

The Waterfall in theory was a good methodology and in a perfect world it would have solved all software development issues. However as we know the world is not perfect, the human race has made sure of that.



Waterfall in reality is a truer picture of how many large scale projects have fallen behind their release schedules. Waterfall is still a great methodology for small projects (and in many cases is still the best option) however with large projects its limitations are soon noticed. As a result of slipping time deadlines either testing is sacrificed or parts of the application are held back for future releases.

To help rectify this issue the V-Model was created. In this model testing activities were aligned with the different levels of documentation produced. The model included feedback loops so that improvements could be made before the application was released.



The V-Model

Although many variants of the V-model exist, a common type of V-model uses four test levels. These are:

- Component (unit) testing
- Integration testing
- Systems testing
- Acceptance testing

In the real world a V-model project may have more, fewer or different levels of development and testing. This will depend on the project and the software product. For example, there may be component integration testing after component testing, and system integration testing after system testing. While this was an improvement over the waterfall method it was still very restrictive in many areas and did not easily allow change or adaption. This opened the door for a new method and the ideas that became Agile began to form.

In the early days these Iterative incremental development models slowly began to gain popularity as their worth in large complex projects began to be realised. Iterative-incremental development is the process of establishing requirements, designing, building and testing a system in a series of short development cycles (sprints).

Examples are:

- Agile development models.
- Prototyping,
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)

A system that is produced using these models may be tested at several test levels during each and every iteration. An increment, added to others developed previously, forms a growing partial system, which should also be tested. Regression testing is increasingly important on all iterations after the first one. Verification and validation can be carried out on each increment.

In the 1990s, these lightweight approaches gained popularity in an effort to come up with an effective alternative to Waterfall. RAD, or Rapid Application Development, relied on building prototypes to allow requirements to emerge and elicit frequent feedback.

Despite the improved methods RAD still managed to get a bad name; however this was down to bad planning and implementation on some major, high profile projects rather than the agile framework itself.

For this reason the term RAD has now been buried and replaced with Agile to allow these early mistakes to be forgotten in the depths of time.

The Scrum and XP (Extreme Programming) methodologies began to take root, both placing a heavy focus on short iterations to allow frequent delivery of software. In general, to serve the actual business needs and improve software project success rates. The early origins of agile go back before software was used however the ideals of agile are easily transferable to most industries.

A brief history in time of the agile framework

1948: The Toyota Way is conceived by Taiichi Ohno, Shigeo Shingo and Eji Toyoda.

1990: Rapid Application Development (RAD) is documented by James Martin.

1995: Ken Schwaber and Jeff Sutherland present a paper of scrum at the OOPSLA (Object-Oriented Programming Systems, Languages and Applications conference).

1999: Kent Beck publishes Extreme Programming (XP) Explained, a very good book that everyone should read.

2001: The Agile Manifesto is created, see below.

February of 2001 is now a pivotal date in the history of agile. On this date a group of highly ambitious and talented developers who shared a joint interest in advancing lightweight development methodologies got together to discuss their views and try to forge some common ground.

From this get together the concept of agile was born. The developers who created agile all understood the importance of creating a software development model in which each iteration in the development cycle would learn from the previous iteration. The result was a methodology that was more flexible, efficient, and team-oriented than any of the previous models.

No matter what agile methods you use they all adhere to 12 core principles for guidance. It is the adherence to the guidance provided by the manifesto and principles is what makes a software development team agile, not a specific process, tool or label.

So what is Agile and what are its foundations?

- Agile is not a development tool, rather it is a collection of evolving delivery and management frameworks for dynamic and innovative delivery environments, such as software development.
- Individually and collectively the frameworks are focused on ensuring that the highest priority is to satisfy the customer(s) through early and continuous delivery of valuable product, listening to the customer and by adapting to changing requirements.

The Manifesto for Agile Software Development

Individuals and interactions	over	processes and tools
Working software	over	documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

The 12 principles behind the Agile Manifesto

Your highest priority is to satisfy the customer through early and continuous delivery of valuable, usable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

So is the company you work for really agile? The answer is probably not. While it may be true that you have a task board in the middle of the main meeting room and some of the teams undertake daily stand-ups which may, or may not relate to work, this does not mean your company is truly agile. The agile methodology is a framework that your organizations either adopts fully and does not simply cherry pick the parts they like or the parts that seem easy to implement. That approach does not make you agile, it is like going out on a Friday night and drinking ten pints of lager or two bottles of wine before eating a fat laden kebab or burger on the way home and saying you are healthy because you also drank a diet coke at the same time. For agile to work fully for your company you have to adopt it fully.

So think about the question again, is your company agile? Now answer honestly before reading further.

Agile versus traditional waterfall

Metric	Waterfall	Agile
Planning scale	Long-term	Short-term
Distance between customer and developer	Long	Short
Time between specification and implementation	Long	Short
Time to discover problems	Long	Short
Project schedule risk	High	Low
Ability to respond	Low	High

Agile Software Delivery

The level of risk associated with software development is increasing as the level of complexity in software and associated hardware also increases. As the level of risk increases so does the chance of failure and litigation if things go drastically wrong. To summarise this

- The ever increasing complexity of systems being built or modified.
- The need to integrate new systems and technologies with legacy systems.
- Reduced timescales for delivery resulting in a reduction of testing.
- Internet based solutions now have to cater to a market that is global.

These pressures have given rise to the need to streamline the delivery cycle to provide greater flexibility and the ability to embrace change. These methods are now collectively called Agile Methods. The level of agile usage has increased greatly over the past decade. This is driven primarily by:

- The desire for faster (and therefore cheaper) time to market.
- A view that traditional methods are too process heavy, expensive and wasteful.
- The need to respond better to business and market change.
- Improved quality and usability.
- The realisation that new information and changing news during a development life cycle have to be easily integrated.

Agile processes always value communication over documentation but at no point does any agile method say 'No Documentation' this is a myth that some companies use as a barrier to agile. This no documentation idea was a myth brought about by poor understanding of agile by those who did not want to change.

To put it correctly the concept of just enough documentation is the correct agile way. Exactly how much documentation is required will vary from project to project and affected by these requirements and others:

- Knowledge of implied requirements
- Audit Requirements
- Product Lifetime
- Fluctuation of team members
- Regulatory requirements

In other words just enough documentation at just the right time for just the right audience will provide the correct information for the correct people.

Spotting when too little documentation is being produced is a critical exercise, below are some pointers that will help achieve this goal.

- Too many defects and bugs are being returned as not fixed or not fully fixed.
- At the review meeting completed products are not getting accepted as done by the product owner.
- Requirements are being misunderstood or missed completely.
- Too much time being spent gathering information at the beginning of each individual sprint.
- In long term projects it is becoming harder and harder to understand the status quo of the system due to a lack of product documentation.

If one or more of these issues are identified then an urgent review of the documentation process should be undertaken. When the documentation level has been optimised then the efficiency of the entire team will be improved. This opens the door to a world of being able to continuously deliver working software while allowing for and supporting changing requirements.

A reminder of the Agile Manifesto

Individuals and interactions	over	processes and tools
Working software	over	documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

Also a reminder of the 12 Principles of Agile Software Development

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

So now you are really getting into this agile development world let us look at some of the key principles:

Iterative: Agile software processes correctly acknowledge that we often get things wrong before we get them right. Therefore, agile processes focus on short cycles (usually 1 month sprints). Within each cycle, a certain set of activities is agreed and completed (if all goes well). These cycles will be started and completed in a matter of weeks. However, a single cycle (also called iteration) will probably not be enough to get the more complex element's 100% correct. Therefore, the short cycle is repeated many times to refine the deliverables and they are considered done, complete and fit for purpose.

Incremental: An agile process does not try to build the entire system at once (that would be the good old waterfall method). Instead, it partitions the nontrivial system into increments which may be developed in parallel, at different times, and at different rates. The programmers will then unit test each increment

independently. When an increment is completed and tested, it is integrated into the system for further testing.

Collaborative: One cool feature of agile processes is that they help foster communication among team members. This communication is a vital part of any software development project. When a project is developed in pieces, understanding how the pieces fit together is vital to creating the finished product. However there is more to integration than simple communication. Quickly integrating a large project while increments are being developed in parallel requires collaboration, teamwork and a good general understanding of the product under development and the desired end goal.

Adaptive: During any monthly iteration, new risks may be exposed which require some activities that were not planned previously. The agile process adapts the process to attack these new found risks. If the goal cannot be achieved using the activities planned during the iteration, new activities can be added to allow the goal to be reached. Similarly, activities may be discarded if the risks turn out to be ungrounded.

So that is the key principles of agile but now what about the agile team at the centre of all this agile development, just who are these players? First, to help understand how the agile team works best let us first look at a Traditional Organisational Structure as shown below.

Project Leader	Development	Testing
Project Leader 1	Programmer 1	Tester 1
Project Leader 2	Programmer 2	Tester 2
Project Leader 3	Programmer 3	Tester 3

In this type of organisation everyone is separate. Typically the Business people (project managers etc will reside on the top floor when they talk to the customers via email and telephone with frequent meetings at expensive restaurants and tapas bars.)

The testers will be in the middle ground where they do other administration duties when not on a testing task. Finally the programmers will be down in the basement writing highly complex code.

On a business level there is little cross team communication and usually the testers are given very little knowledge of future release plans, testers even less so.

So now let us look at a modern agile Structure.

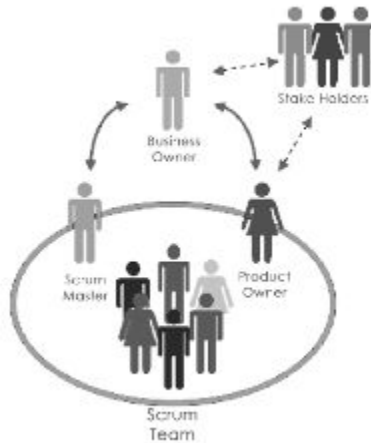
Project Leader	Development	Testing	
Product Owner 1	Programmer 1	Tester 1	Team 1
Product Owner 2	Programmer 2	Tester 2	Team 2
Product Owner 3	Programmer 3	Tester 3	Team 3

Here we see a more team orientated structure. Such teams (cells) may change from project to project but while a team exists they work together and communicate within the cell. All members of the team are present at regular development meetings and the daily scrums help improve the team group knowledge and also help gel the cell into one tight cohesive unit. Below is an image that shows how the scrum team fits into the bigger picture. This also shows how and when the scrum master is involved.

So now let's have a look at these individual people and groups.

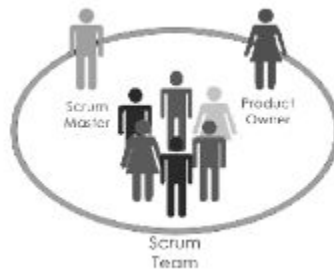
The Stakeholder(s)

A *stakeholder* is an organization who will be financially impacted by the outcome of the solution and is clearly much more than just a standard end-user. A stakeholder may be one of the following:



- A direct or indirect user of the system.
- A manager of a group of users.
- A senior manager or company director.
- Developer(s) who are working on other systems that integrate or interact with the one under development.
- An operations or IT staff member.
- The owner who funds the project.
- Auditor.
- Your program/portfolio manager.

The Product Owner



The product owner is the one team member who speaks on behalf of the customer. This person represents the needs and desires of the stakeholder community to the agile delivery team; as a result they should be considered an important part of the team and a vital gateway. He or she is accountable for ensuring that the development team delivers the required value to the business. The Product Owner is also responsible for maximizing the value of the product and the work of the Development Team

The Product Owner writes (or has the team write) customer-centric items (also known as user stories). They clarify any details regarding the solution and are also responsible for maintaining a prioritized list of work items that the team will implement to deliver the solution. They will then rank and prioritize them, and then adds them to the product backlog; this task is the sole responsibility of the product owner. While the product owner may not be able to answer all questions, it is their responsibility to track down the answer in a timely manner so the team can stay focused on its current tasks. Each agile team, or sub-team in the case of larger projects organized into a team of teams, has a single product owner.

The Product Owner is always just one person, they are never a committee. The Product Owner may represent the desires of a committee in the Product Backlog, but those wanting to change a Product Backlog item's priority must address the Product Owner directly.

Therefore every scrum team should have only one Product Owner and this role should never be combined with that of the Scrum Master. In an enterprise environment, though, the Product Owner is often combined with the role of Project Manager as they have the best visibility regarding the scope of work (products).

Additionally the product owner owns the following roles:

- Communicates the project status and represents the work of the agile team to key stakeholders
- Develops strategy and direction for the project and sets long- and short-term goals

- Understands and conveys the customers' and other business stakeholders' needs to the development team
- Gathers, prioritizes, and manages product requirements
- Directs the product's budget and profitability
- Chooses the release date for completed functionality
- Answers questions and makes decisions with the development team
- Accepts or rejects completed work during the sprint
- Presents the team's accomplishments at the end of each sprint

The Scrum Master

The daily scrum is overseen by a scrum master, although they do not have to actually attend each meeting. They are responsible for ensuring Scrum is fully understood and enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum theory, practices, and rules.

The scrum master is also accountable for removing impediments to the ability of the team to deliver the product goals and deliverables. The scrum master is not a traditional team leader or project manager, but acts as a buffer between the team and any distracting influences.

During the daily scrum the scrum master ensures that the Scrum process is used as intended, for example a short, daily meeting held every day. He or She is the enforcer of the rules of Scrum, often chairs key meetings, and challenges the team to improve. The role has also been referred to as a *servant-leader* to reinforce these dual perspectives.

Servant leadership is both a leadership philosophy and set of leadership practices. Traditional leadership generally involves the accumulation and exercise of power by one at the “top of the pyramid.” By comparison, the servant leader share power and puts the needs of others first and helps people develop and perform as highly as possible.



The Development Team

The Development Team is responsible for delivering potentially shippable increments of product (working software) at the end of each Sprint. A development team is typically made up of 3–9 individuals with cross-functional skills who do the actual development work. These skills include the following

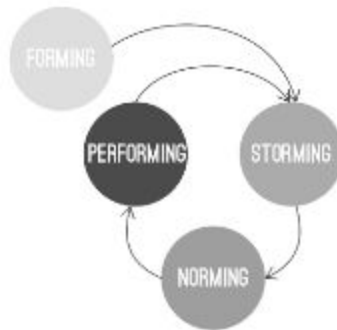
- Analyze the needs
- Designing the system
- Developing Code
- Testing Code
- Create/Update the documentation

No matter what your job title or your role within the team you are as important as the other members. The role of each team member focuses on producing the actual solution for stakeholders. Team members perform testing, analysis, architecture, design, programming, planning, estimation, and many more activities as appropriate throughout the project.

The optimal Development Team size is always small enough to remain nimble and large enough to complete significant work within a Sprint. A team that is fewer than three Development Team members will suffer from decreased interaction and smaller productivity gains. Smaller Development Teams may also encounter skill constraints during any given Sprint, causing the Development Team to be unable to deliver a potentially releasable Increment. At the other end of the scale having more than nine members requires too much coordination. Large Development Teams generate too much complexity for an empirical process to manage. The Product Owner and Scrum Master roles are not included in this count unless they are also executing the work of the Sprint Backlog.

It is unlikely that every team member has every single skill (at least not yet), but they have a subset of them and good team members will strive to gain more skills over time. Team members should identify, estimate, sign-up for, and perform tasks and track their completion status.

Within the scrum environment the development team is a self-organizing entity even though there may be some level of interface with project management offices. Mature teams are built around trust and respect for each other's key skills and knowledge. New teams will need to go through a forming to performing cycle as shown below.



Forming:

- A high degree of guidance is required from management
- Individual roles are still unclear
- Processes usually not well established

Storming:

- Beginning to understand how team decisions will be made
- The team purpose is clear but relationships are still forming

Norming

- Relationships are now well understood within the team
- Commitment to team goals now realised
- The team start to optimise team processes

Performing

- The team is now committed to performing well
- The team can now focus on being strategic
- Team runs well as a cell

When a team reaches level four (performing) they should be able to deliver high quality increments on a regular basis. The velocity of the team will also improve as they grow to understand each other's strengths and weaknesses. So with the team in place it is time to go agile, I hope you are now feeling excited and eager to move forward.

For an agile team to strive onwards and start delivering deliverables that can be considered done and fit for purpose they need to work within the agile requirements specification, next we will consider some of the key points of this.

- **Active stakeholder participation:** Stakeholders should be available to provide key information in a timely manner. They should also be able to make important decisions when required and be very actively involved in the development process through the use of inclusive tools and techniques.
- **Prioritised requirements:** Agile teams implement requirements in priority order, as defined by their stakeholders, so as to provide the greatest return on investment possible. These priorities are agreed and ordered at the beginning of each sprint.
- **Requirements envisioning:** At the beginning of each agile project the team will need to invest some time to identify the complete scope of the project and to create the initial prioritised list of requirements. This effort should take a from a few days and up to two weeks, assuming you can overcome the logistical challenges associated with getting the right people involved.
- **Test-driven development:** The basis of this is to write a single test, either at the requirements or design level. Then write just enough code to fulfill that test. TDD is a JIT approach to detailed requirements specification and a confirmatory approach to testing.
- **Daily Scrum:** In the daily scrum the team members will each answer three questions, these are: What I did yesterday; what I plan to do today; what impediments are stopping me from completing these tasks. The scrum takes place every morning at the same time and at the same place. The job of the scrum master is to manage the scrum and deal with any impediments.
- **XP and Requirements:** The XP approach is to capture requirements through User Stories. This has become a very common method in Agile and has proven a very popular and affective method.

The focus of XP is customer satisfaction. XP teams achieve high customer satisfaction by developing features when the customer needs them. New requests are part of the development team's daily routine, and the team must deal with requests whenever they become known. The team automatically organizes itself around any problem that arises and solves it as efficiently as possible.

XP is based around a set of 29 well designed and intuitive rules, these are as follows

- Planning
 - User stories are written.
 - Release planning creates the release schedule.
 - Release often Make frequent small releases.

- The project is divided into iterations.
- Iteration planning will start each iteration.
- Managing
 - Optimize last Give the team a dedicated open work space.
 - Always work at a sustainable pace.
 - A stand up meeting starts each day.
 - The Project Velocity is measured.
 - Move people around.
 - Fix XP when it breaks.
- Designing
 - Simplicity.
 - Choose a system metaphor.
 - Use CRC cards for design sessions.
 - Create spike solutions to reduce risk.
 - No functionality is added early.
 - Refactor whenever and wherever possible.
- Coding
 - The customer is always available.
 - Code must be written to agreed standards.
 - Test Driven Development, always code the unit test first.
 - All production code is pair programmed.
 - Only one pair integrates code at a time.
 - Continuous integration.
 - Set up a dedicated integration computer.
 - Use collective ownership.
- Testing
 - All code must have unit tests.
 - All code must pass all unit tests before it can be released.
 - When a bug is found tests are created.
 - Acceptance tests are run often and the score is published.

So those are the rules of XP, now let us discuss some of the major key features in more detail.

Pair programming

This is sometimes referred to as Peer programming. All code to be sent into production is created by two programmers working together at a single computer terminal. Pair programming has been shown to increase software quality without impacting time to deliver. At first it may appear counter intuitive, but 2 programmers working at a single computer will add as much functionality as two working separately

except that it will usually be much higher in quality as each member of the team strive to outdo the other with well designed and clever code. With this increased quality comes big savings later in the project as fewer bugs will require fixing.

One important consideration that management will need to consider is that pair programming is a social skill that for some programmers will take time to learn. However you are striving for a cooperative way to work that includes give and take from both partners regardless of experience, age and company status; therefore programmers will learn to cooperate if they are to fit into modern agile teams.

This is one area of agile that is most likely to experience resistance from programmers. So of them will take on the methodology happily, others will kick up a fuss over the idea and a few may even leave rather than adjust. These are issues to be considered when planning a move to agile and whatever is decided during the planning phase should remain policy no matter what resistance is raised.

No functionality is added early

Keep the system uncluttered with extra stuff that you guess will be used later. History has demonstrated that only 10% of unrequested extra functionality that has been added will ever get used, so in affect programmers are wasting 90% of their time by putting it in there at the early stages.

Not only is this a waste of time for the programmer it also increased the testing workload and increases risk. We are all tempted to add functionality and show how clever we are now rather than later because we see exactly how to add it or because we believe it would make the system so much better. It seems like it would be faster to add it now.

When a bug is found

When a bug is found tests should always be created to guard against it coming back. Also a bug that is discovered in production always requires an acceptance test be written to guard against it. As a result all bugs should be recorded even if the programmer insists it is not necessary.

Move people around

Move people around to avoid serious knowledge loss and coding bottle necks. If only one person on your team can work in a given area and that person leaves or just has too much to do you will find your project's progress reduced to a crawl. This also stops little gangs forming who will work together and against anyone they collectively do not like.

Integrate often

Developers should be integrating and committing code into the code repository every few hours, whenever possible. In any case they should never hold onto changes for more than a day on their local

computer. Continuous integration often avoids diverging or fragmented development efforts caused where developers are not communicating with each other about what can be re-used, or what could be shared. Everyone needs to work with the latest version. Changes should not be made to obsolete code causing integration headaches.

Collective Ownership

Collective Ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, improve designs or refactor. No one person becomes a bottle neck for changes. This is hard to understand at first. It's almost inconceivable that an entire team can be responsible for the system's design. Not having a single chief architect that keeps some visionary flame alive seems like it couldn't possibly work.

User Stories

These are light weight, very brief descriptions that are written by the user/stake holder. They should say what they want the system to do. A user story could be 'Create an Add customer page'. A user story briefly explains:

- the person using the service (actor)
- what the user needs the service for (narrative)
- why the user needs it (goal)

7 10

As a
I would like
So that ...

Acceptance Criteria:

- Criteria one
- Criteria two
- Criteria three

User Stories are discussed further later in the book

So those are the major key features of XP. Another interesting agile tool that can be used by itself or alongside other agile methods such as XP is Kanban. Kanban comes from the Japanese terms Kan (visual) and Ban (board) and is quite simply that, a visual board or taskboard. It is a system to control the logistical chain from a production point of view, and is an inventory control system. Kanban was developed by Taiichi Ohno, an industrial engineer at Toyota, as a system to improve and maintain a high level of production. Today it is a very popular tool for use in software development.



The Kanban Method is used by organizations to manage the creation of products with an emphasis on continual delivery while not overburdening the development team. Like Scrum, Kanban is a process designed to help teams work together more effectively.

The Kanban method is rooted in four basic principles:

Start with what you do now:

The Kanban method does not prescribe a specific set of roles or process steps. The Kanban method starts with the roles and processes you have and stimulates continuous, incremental and evolutionary changes to your system.

Agree to pursue incremental, evolutionary change:

The company (or team) should agree totally and completely that continuous, incremental and evolutionary change is the way to make system improvements and make them stick.

Sweeping changes may seem more effective within an organisation but also have a higher failure rate due to resistance and fear in people within the organization. The Kanban method encourages continuous small incremental and evolutionary changes to your current system which people tend to be more comfortable with.

Respect the current process, roles, responsibilities and titles:

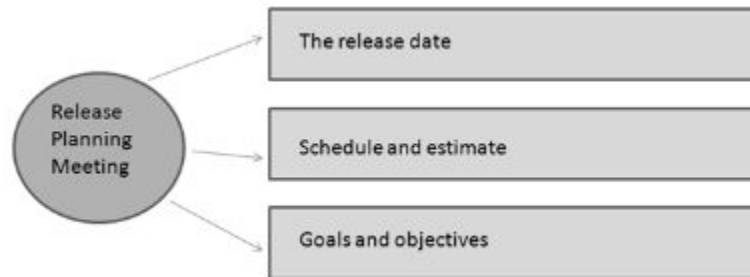
It is likely that the organization currently has some elements that work acceptably and are worth preserving, change for changes sake is not always the best way forward. By agreeing to respect current roles, responsibilities and job titles you can eliminate some of the initial change fears.

Leadership at all levels:

Acts of leadership at all levels in the organization from individual contributors to senior management should be encouraged. Meetings are an important part of agile software development. Meetings keep everyone involved in the development life cycle updated, informed and in contact with progress. Regular meetings also help identify forthcoming problems and road blocks at an early stage. Next we will discuss the five types of agile meetings.

1: The Release Planning Meeting

A vital and important part of agile is regular meetings. There are five meetings of particular importance with agile, so let us now discuss the first of these.



This is the release planning meeting. This meeting takes place at the start of every release cycle. The purpose of the release planning meeting is also to get an idea of what stories the team will try to finish by the release end date.

The product owner will select the stories in a logical order of importance. There may be dependencies which will affect this order as well but in general the highest priority will be started first.

It is also important that the team understands the possibility that not all of the stories will get completed by the release end date. One of the basic premises of agile is to deliver working software, so it is important to have the highest-value stories completed first (barring dependencies) so that the software you do deliver meets the customer's needs. The final acceptance criteria should be agreed at this meeting and the result of this meeting is the product backlog.

The product backlog is a dynamic list of stories. These stories may be edited, deleted or added at any time during the release cycle. As previously discussed the list is prioritized and items with the highest priority should always be completed first. The backlog is progressively refined, adjusted and improved during the life cycle.

When the product owner sets the scope of the next iteration, he or she needs to know that the scope is the right size for the team, in other words there isn't too much work to get done in the iteration. Like any other developers, agile team members estimate their own work. Unlike other developers, agile team members usually estimate in points. Points represent a size-based, complexity-based approach to estimation. Points

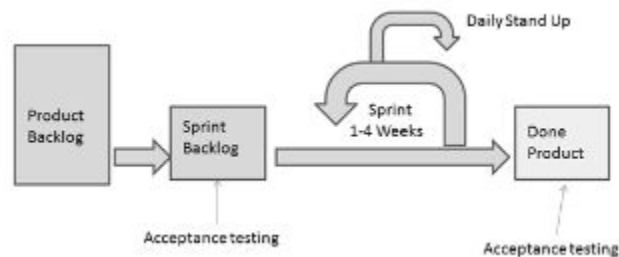
are assigned in whole numbers (1, 2, 3, 4 and so on with no fractions or decimals) and represent relative sizes and complexity of work items. Small and simple tasks are one point tasks, slightly larger/more complex tasks are two point tasks, and so on with 10 usually being the highest.

Points can be thought of as shirt sizes. There can be small, medium, large, extra large, and potentially other sizes (extra small, extra extra-large and these days even xxxx large). These sizes are relative to the team using them; no formal regulation dictates how much larger medium is compared to small.

Planning Poker
<p>As you refine your requirements, you will also need to refine your estimates also. Planning Poker is a technique to determine user story size and build consensus with the development team members. Planning poker is a very popular and straightforward approach to estimating story size.</p>
<p>To play planning poker you need a deck of cards with point values on them. There are plenty of poker tools and mobile apps that are free to use, or you can do it yourself with index cards. The numbers on the cards are usually from the Fibonacci sequence.</p>
<p>Only the development team plays estimation poker. The team leader and product owner do not get a deck and also do not provide estimates. However the team leader can act as a facilitator and the product owner usually reads the user stories as well as providing detailed information on the stories.</p>

In practice, one team's three-point size estimate for a work item may correlate to another team's two-point estimate for an identical work item. Teams need only agree on what size and complexity corresponds to what point count and remain internally consistent in their use. These points are used to help estimate the team's velocity, or work output.

2: The Iteration/Sprint Meeting



The next meeting is the iteration/sprint planning meeting. In Scrum, every iteration or sprint (typically 2-4 weeks) begins with the sprint planning meeting. At this meeting, the Product Owner and the development team will negotiate which stories the team will tackle during the current sprint.

These meetings are typically time-boxed to eight hours, this helps to focus the group and ensure no time is wasted on idle chatter such as the weekend's football results or what car each team member would like to purchase.

The final choice of what stories are to be included will always rest with the Product Owner. Typically they will select those with the highest business value, however the team has the power to push back and voice concerns about impediments and dependencies.

When the story list is decided and agreed this becomes the required work for the current sprint. Team members choose what they will each do. Nothing is assigned or pushed onto anyone. The result of this meeting is the sprint backlog.

So as you can see the sprint backlog is a negotiated set of items/stories from the product backlog that a team commits to complete during the time box of the current sprint. Items in the sprint backlog are broken into detailed tasks for the team members to complete. The team works collaboratively to complete the items in the sprint backlog, meeting each day (during a daily scrum) to share struggles and progress and update the sprint backlog and burn down chart accordingly.

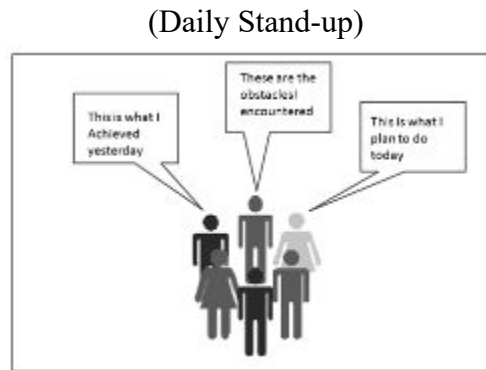
A sprint will typically last for 4 weeks, this can vary slightly but 4 weeks is by far the most common time frame. This is also known as time-boxed development. In time-boxed development the end calendar date for the Sprint is determined at the outset. The Sprint is then complete once that day is met and all work that's going to be launched needs to be done.

This work will have passed through QA and testing and is deemed and ready for launch.

Time-boxing the Sprint requires a much sharper focus on prioritizing work, since you can't push the end date if work isn't done.

Features that aren't complete get pushed to the next Sprint. This forces decision making much earlier on what will and what won't make it into the Sprint. In fact, hard decisions on prioritizing work should really be done right at the outset. Decisions on what features get pushed to the next Sprint have to happen very early, this helps ensure that no testing cannot be completed before the end of the Sprint.

3: The Daily Scrum Meeting



The heart of the Scrum process is the daily stand-up meeting, also known as the daily Scrum. No other meeting captures Scrum's emphasis on communication and transparency quite like the stand-up. This meeting helps ensure that the entire development team is always on the same page within the current sprint. Every day, the scrum team will gather together, usually in a team room or private office in order to report on the progress made since the last meeting, goals for the next one, and any impediments blocking their path. These reports are often phrased as responses to the following three questions:

- What have I done since the last Scrum meeting (yesterday)?
- What will I do before the next Scrum meeting (tomorrow)?
- What prevents me from performing my work as efficiently as possible (impediments)?

All development team members must participate on the entire daily scrum meeting and they must be prepared to help each other. The meeting is just 15 minutes (no longer), so everybody must participate and pay attention in what tasks are done and what needs to be developed, and make suggestions on other member's items when needed.

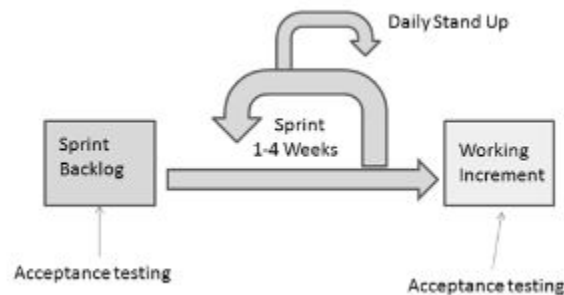
It is the scrum masters job to deal with any reported impediments and also to ensure the scrum meeting happens, runs correctly and is concluded within the agreed time frame. This is the meeting that those team members who are not fully committed to agile will struggle hardest with. They will consider these fifteen minutes every morning a total waste of their time even though they would probably spend that same time slot talking about other non work related items around the water or vending machine. Such people will be easy to spot; they are the ones sitting quietly with folded arms and a bored look on their face. They will say very little and will probably need prompting to divulge their three answers.

This behavior needs to be tackled early. At first friendly conversations to try and find out why they are resisting the change and finally as a last resort moving the non-committed to either another department or another company.

I have always found that it is best to do the daily scrum meeting in front of Kanban or the Sprint Backlog. By using this method the team can see clearly what are the remaining items to be developed, and also the team members can discuss the chosen items to be developed on that day in order to achieve the maximum speed configuration to achieve the sprint goal.

It is also good practice ask individually each development team member about the feeling concerning achieving the sprint goal. If the majority answers consider difficult to achieve the goal, the team must find, in group, different ways to change this scenario. Also you should also the focus solely on the daily scrum meeting; you will not have time for extra subjects besides the sprint backlog.

4: The Iteration (Sprint) Review

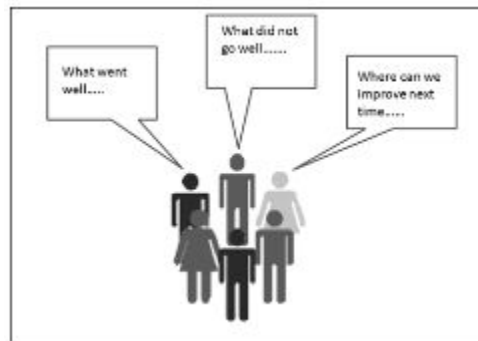


In Scrum, when the sprint ends, it's time for the team to present its work to the product owner for acceptance. This is known as the sprint review meeting. At this time, the product owner will go through the previously agreed sprint backlog and they will ask the team to present its work. The product owner checks the work against the acceptance criteria to determine if the work has been completed satisfactory or not. Even if only one percent of a story remains to be completed by a team, the product owner must reject the story as unfinished, incomplete work carries unacceptable risks and should never be passed as done. Teams commonly discover that a story's final touches often excise the most effort and time, so awarding partial credit for an incomplete story can contribute to a slanted velocity. The sprint review is a time boxed event of 4 hours for a monthly sprint, however if the sprint was shorter than the review can also be shorter.

Preparation for the sprint review meeting should not take more than a few minutes. Even though the sprint review might sound formal, the essence of showcasing in agile is informality. The meeting needs to be prepared and organized, but that doesn't require a lot of flashy presentation material. Instead, the sprint review focuses on demonstrating what the development team has achieved and what is considered done.

As a result you should gather sprint review feedback informally. The product owner or team lead can simply take notes on behalf of the development team, as team members often are engaged in the actual presentation and resulting conversation. New user stories may also come out of the sprint review. These new user stories can be new features altogether or changes to the existing code.

5: The Sprint Retrospective Review



When the sprint review meeting has been concluded the scrum master and the development team get together for a sprint retrospective review. This is a team led meeting however they can invite the product owner and other stakeholders if they feel their input will be valuable. This is a time boxed event lasting 3 hours.

During this meeting the team will consider three important issues, these are:

- What went well?
- What did not go well?
- What improvements could be made for the next sprint?

Because the product owner does not sit in on these meetings this is a good opportunity for the team members to talk frankly about the sprints successes and failures. This is an important meeting for the team because they have an opportunity to focus on its overall performance and they can identify potential strategies to improve its processes. What should be avoided always is these meetings being reduced to a blame and shame argument about who broke what and who did not do what correctly. If the meeting is poorly managed and some of the programmers have large ego's this can easily happen, it is at these points that the scrum master has to be at their strongest. The scrum master is able to observe common impediments that impact the team and how they work together. From this the scrum master can work out plans to resolve these impediments.

During every sprint the team has a need to current progress information; one of the most popular methods of keeping the team updated is the use of task boards. As previously mentioned one of the popular styles is Kanban and most agile teams will deploy a task board of some form. Task boards are a good guide to progress for all the team members and stake holders. They are an easy to understand visual representation of the current sprint state.

So the overall goal of the iteration retrospective is to continuously improve your processes. Improving and customizing processes according to the needs of each individual team increases team morale, improves efficiency, and increases velocity and work output.

We have already discussed task boards but to reiterate these are usually situated in a central location that is easily observed by most of the team that are based in the office. However electronic versions are becoming increasingly popular because they allow team members to access the data anywhere on the planet, this could be remote offices or even home workers. This is particularly useful in organisation which has a global presence and team members can be on different continents. One example is available in Team Foundation Server which we will cover later in the book.

Another important tool for the agile team is the iteration **burn down chart**. This is a graphical representation of work left against remaining time within the current sprint. This is useful for predicting when all of the work will be done and is also a good method of detecting potential deadline problems. Below is a typical example of a burn down chart.



These tools can help measure the team's performance which should improve over time as the team becomes more experienced and as they learn to work together as one efficient unit. This is known as the team velocity.

The main idea behind velocity is to help agile teams estimate just how much work they can complete in a given time period based on how quickly similar work was completed in previous sprints.

The following terminology is used in velocity tracking. To calculate velocity, a team first has to determine how many units of work each task is worth (10, 20 etc) and the length of each interval. During development, the team will keep track of all completed tasks and at the end of the interval count the number of units of work completed during the interval.

The team then writes down the calculated velocity in a chart or on a graph (paper or electronic). The first week will always provide little value, but is essential to provide a basis for comparison and a datum. Each week after that, the velocity tracking will provide better information as the team provides more accurate estimates and becomes more used to the methodology.

The unit of work can either be a real unit such as hours or days or an abstract unit like story points or ideal days. Each task in the software development process should then be valued in terms of the chosen unit. By contrast the interval is the duration of each iteration in the software development process for which the velocity is measured. The length of an interval is determined by the team. Most often, the interval is a week, but it can be as long as a month.

So that is the agile team, the methods they use and the tools they employ to get the job done. So you are still here, this means that either you still want to be an agile tester and/or you are finding this book a good read. Whatever the reason let us move forward to the next chapter, the Agile Tester.

So you are still here are you? Does that mean you still want to be an Agile Software Tester? If yes then good for you, so let us now look at what it takes to be a good agile tester. In previous chapters I have hinted at what makes a good tester, virtues such as patience, good people skills and a good eye for detail are essential. Other important considerations are a basic understanding of SQL (structured query language) and a basic knowledge of Java or C# will also help. You will know when you have become a competent tester when you are able to indulge in what is known as experience based testing.

So; what is experience based testing?

Experience-based testing is where tests are derived from the tester's (you or your team) skill's and intuition as well as their experience with previously tested similar applications and technologies. When this style of testing is used to augment systematic techniques, these techniques can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches. However, this technique may yield widely varying degrees of effectiveness, depending on the testers' experience and capability. Essentially the more experienced and better trained the tester is the more valid this type of testing will become.

A commonly used experience-based technique is error guessing. In general, testers anticipate defects based on their previous experience and knowledge of the programmers style of coding. A good structured approach to the error guessing technique is to enumerate a list of possible defects and to design a set of tests that attack these defects head on. This systematic approach is called fault attack. These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails. Another technique used by experienced testers is called exploratory testing.

This type of testing is concurrent test design, test execution, test logging and learning, based on an agreed test charter containing test objectives, and carried out within sprints. This is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check on the test process, to help ensure that the most serious defects are located and logged.

So moving on, what is agile testing anyway? First let's remind ourselves what the Agile Manifesto is.

Individuals and interactions	over	processes and tools
Working software	over	documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

So using the values from the Manifesto to guide the team, we strive to deliver small chunks of business value in extremely short release cycles (usually 4 weeks). It is important to understand that a tester on an agile project will work differently than one working on a traditional project.

The Testers need to understand the values and principles that underpin agile projects, and how testers are an integral part of a whole-team approach together with developers and business representatives. As you have already seen the members in an agile project communicate with each other early and frequently, which helps with removing defects early and developing a quality product.

The whole team is responsible for quality as a group in agile projects, remember there is no I in team, no one is alone in agile. The essence of the whole-team approach lies in the testers, programmers and the business representatives working together in every step of the development process.

Testers will need to work closely with both programmers and business representatives to ensure that the desired quality levels are achieved.

This includes supporting and collaborating with business representatives to help them create suitable acceptance tests, working with developers to agree on the testing strategy, and deciding on test automation approaches. Testers can thus transfer and extend testing knowledge to other team members and influence the development of the product.

Everyone on an agile team is a tester at some level. Anyone should be able to pick up testing tasks. If that's true, then what is special about a dedicated agile tester? If I define myself as a tester on an agile team, what does that really mean? Do agile testers need different skill sets than testers on traditional teams? What guides them in their daily activities? Read on to find out these answers.

You should define an agile tester this way: they are a professional tester who embraces change, collaborates well with both technical and business people, and understands the concept of using tests to document requirements and drive development. As already mentioned agile testers tend to have good technical skills, know how to collaborate with others to automate tests, and they should also be

experienced exploratory testers. They're willing to learn what customers do so that they can better understand the customers' software requirements.

Who's an agile tester? They are a team member who drives agile testing. There are many agile testers who started out in some other specialization. A developer can become test-infected and branches out beyond unit testing. An exploratory tester, accustomed to working in an agile manner, is attracted to the idea of an agile team. Finally professionals in other roles, such as business or functional analysts, might share the same traits and do much of the same work.

Successful projects are a result of a good team that is allowed to do good work. The characteristics that make someone succeed as a tester on an agile team are probably the same characteristics that make a highly valued tester on any team.

A good, experienced agile tester doesn't see themselves as a quality police officer, protecting their customers from inadequate code. They should be ready to gather and share information, to work with the customer or product owner in order to help them express their requirements adequately so that they can get the features they need, and to provide feedback on project progress to everyone.

Poor specifications are very often a major reason for project failure. Specification problems can result from the customer's lack of insight into their true needs, absence of a global vision for the system, redundant or contradictory features, and other forms of miscommunication. In agile development, user stories are written to capture requirements from the perspectives of developers, testers, and business representatives. In sequential development, this shared vision of a feature is accomplished through formal reviews after requirements are written. In agile development, this shared vision is accomplished through frequent informal reviews while the requirements are being written.

The user stories must address both functional and non-functional characteristics. Each story includes acceptance criteria for these characteristics. These criteria should be defined in collaboration between business representatives, developers, and testers. They provide developers and testers with an extended vision of the feature that business representatives will validate. An Agile team considers a task finished when a set of acceptance criteria have been satisfied.

This is one of the valuable roles of the agile tester; typically they will improve the user story by identifying missing details or non-functional requirements. An agile tester can contribute by asking business representatives open-ended questions about the user story, proposing ways to test the user story, and confirming the acceptance criteria. Different agile teams vary in terms of how they document user stories. Regardless of the approach taken to document user stories, documentation should be concise, sufficient, and necessary and you the tester should be taking an active role in their creation.

Testers should also always play an important role in the sprint retrospectives. Agile testers are part of the team and are able to bring their unique perspective to the table. Testing occurs in each sprint and vitally contributes to success. All team members, testers and non-testers, can and should provide input on both testing and non-testing activities. That is after all the agile way.

Another feature of the agile way is continuous integration. Delivery of each product increment requires working, reliable, and integrated software at the end of every sprint. Continuous integration can address this challenge by merging all changes made to the software and integrating all changed components to a central repository regularly, at least once a day.

In the agile world configuration management, compilation, software build, deployment, and testing are wrapped into a single, automated, repeatable process. As a result since developers integrate their work constantly, build constantly and test constantly, then any defects in code are detected much more quickly.

The good news for agile testers is that continuous integration allows them to run automated tests regularly, in some cases as part of the continuous integration process itself, and send quick feedback to the team on the quality of the code. These test results are visible to all team members, especially when automated reports are integrated into the process. Automated regression testing can also be continuous throughout the iteration.

Good automated regression tests cover as much functionality as possible, including user stories delivered in the previous iterations, they can never cover everything though and manual testing will always be required at some level.

Good coverage in the automated regression tests helps support building (and testing) large, complex, integrated systems. When much of the regression testing is automated, the agile testers are freed to concentrate their manual testing on new features, implemented changes, and confirmation testing of defect fixes.

In addition to automated tests, organizations that use continuous integration typically use associated build tools to implement continuous quality control. In addition to running unit and integration tests, such tools can also run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code, and facilitate manual quality assurance processes. This continuous application of quality control aims to improve the quality of the product as well as reduce the time taken to deliver it by replacing the traditional practice of applying quality control after completing all development.

Therefore continuous integration can provide the following benefits to the development team:

- Allows earlier detection and easier root cause analysis of integration problems and conflicting changes.
- Gives the development team regular feedback on whether the code is working, or not.
- Allows the version of the software being tested to stay within one day of the version being developed.
- Reduces regression risk associated with developer code refactoring due to rapid re-testing of the code base after each small set of changes.
- Provides confidence that each day's development work is based on a solid foundation.
- Makes progress toward the completion of the product increment visible, encouraging developers and testers.
- Eliminates the schedule risks associated with big-bang integration.
- Provides constant availability of executable software throughout the sprint for testing, demonstration, or education purposes.
- Reduces repetitive manual testing activities.
- Provides quick feedback on decisions made to improve quality and tests.

Another important element of agile is planning. For Agile lifecycles, two kinds of planning occur, release planning and iteration planning.

Release planning looks ahead to the release of a product, often a few months ahead of the start of a project. Release planning defines and re-defines the product backlog, and may involve refining larger user stories into a collection of smaller stories.

Release planning provides the basis for the agile test approach and agile test plan spanning all iterations. In release planning, business representatives establish and prioritize the user stories for the overall release, in collaboration with the team.

Based on these user stories, project and quality risks are identified and high-level effort estimation is performed.

Agile testers should always be involved in release planning and especially add value in the following activities:

- Defining testable user stories, including acceptance criteria
- Participating in project and quality risk analyses
- Estimating testing effort associated with the user stories
- Defining the necessary test levels

- Planning the testing for the release

When release planning is complete then the iteration planning for the first iteration begins. Iteration planning looks ahead to the end of a single iteration only and is concerned with the iteration backlog.

In iteration planning, the team selects user stories from the prioritized release backlog. They then elaborate the user stories, performs a risk analysis for the user stories and estimates the work needed for each user story. If a user story is found to be too vague and attempts to clarify it have failed, the team can refuse to accept it and use the next user story based on priority.

The business representatives must answer the team's questions about each story so the team can understand what they should implement and how to test each story. The number of stories selected is based on established team velocity and the estimated size of the selected user stories. After the contents of the iteration are finalized, the user stories are broken into tasks, which will be carried out by the appropriate team members.

Agile testers should always be involved in iteration planning and especially add value in the following activities:

- Participating in the detailed risk analysis of user stories
- Determining the testability of the user stories
- Creating acceptance tests for the user stories
- Breaking down user stories into tasks (particularly testing tasks)
- Estimating testing effort for all testing tasks
- Identifying functional and non-functional aspects of the system to be tested
- Supporting and participating in test automation at multiple levels of testing

Release plans will often change as the project proceeds, including changes to individual user stories in the product backlog. These changes may be triggered by internal or external factors. Internal factors include delivery capabilities, velocity, and technical issues. External factors include the discovery of new markets and opportunities, new competitors, or business threats that may change release objectives and/or target dates. In addition, iteration plans may change during iteration. For example, a particular user story that was considered relatively simple during estimation might prove far more complex and time consuming than originally expected.

These changes can be very challenging for the testers. Agile testers must always understand the big picture of the release for test planning purposes, and they must have an adequate test basis and test knowledge in each iteration for test development purposes. The required information must always be available to the tester early, and yet change must always be embraced according to agile principles. This

dilemma requires careful decisions about test strategies and test documentation which are constantly being reviewed.

Release and iteration planning should address test planning as well as planning for development activities and the agile tester should always ensure the point of view is heard. Particular test-related issues to address include:

- The scope of testing to be done and the extent of testing for those areas in scope, the test goals, and the reasons for these decisions.
- The team members who will actually carry out the test activities.
 - The test environment and test data required for correct testing, when they are needed, and whether any additions or changes to the test environment and/or data will occur prior to or during the project.
 - The timing, sequencing, dependencies, and prerequisites for the functional and non-functional test activities.
- The project and quality risks to be addressed.

One of the main differences between traditional lifecycles and agile lifecycles is the idea of very short iterations (2-4 weeks typically). Each of these iterations should result in working software that delivers features of added value to business stakeholders.

At the very beginning of the project, there is a release planning meeting. This will be followed by a sequence of sprints (iterations). At the beginning of each sprint, there is an iteration planning meeting. Once the sprint scope is established and agreed, the selected user stories are developed, integrated with the system, and then tested. These sprints are highly dynamic, with development, integration, and testing activities taking place all the way through each sprint, and with considerable parallelism and overlap. In agile the testing activities occur throughout the iteration, not as a final activity.

Within the agile team the testers, programmers and business stakeholders all have an important role in software testing, as with traditional lifecycles. Programmers should always perform unit tests as they develop features from the user stories. Testers then test those features as they become available. Business stakeholders also test the stories during implementation. Business stakeholders might use written test cases, but they also might simply experiment with and use the feature in order to provide fast feedback to the development team.

In some of the agile practices (e.g., Extreme Programming), team pairing is used. Pairing can involve testers working together in two's to test features. Pairing can also involve a tester working collaboratively with a programmer (yes this has been known to happen without one trying to kill the other) to develop

and test a feature. Pairing can be difficult when the test team is distributed across different geographical areas, but well planned processes and tools can help enable distributed pairing on a global basis.

Agile teams should always strive to progress forward by having working software at the end of each and every iteration. To determine when the team will have done, working software they need to monitor the progress of all work items in the iteration and release. Testers in agile teams will need to utilize various methods to record test progress and status. These include test automation results, the recording of progress of running test tasks and stories on the task board, and Burndown charts showing the team's overall progress. These can then be communicated to the rest of the team using media such as online wiki dashboards and board emails, as well as verbally during stand-up meetings.

Agile teams can also use tools that automatically generate status reports based on completed test results and task progress.

These in turn can update information dashboards and emails. This is a very important method of communication because it also gathers valuable metrics from the testing process, which can be used in process future improvement. [User Stories](#) and [Burndown](#) charts are discussed in the next chapters.

What approach is best for you

So what parts of agile does your company use (if any). Are these the best options for your company? Is the approach incomplete? Can other options offer a better solution for your company? Let's have a look at what is on offer....

Scrum



At this moment in time scrum is the most popular approach to agile software development in general use. In the scrum ethos any adjustments to the current project are based on experience and not on theory. Because Scrum is currently the most popular agile approach it is the one I will discuss in most depth, also it is the approach I personally favour.

The scrum methodology offers four deliverables, these are:

- Product backlog: This is the full list of requirements that define the whole product.
- Sprint backlog: This is the list of requirements and associated tasks in a given sprint (remember scrum calls iterations sprints).
- Burndown charts: These are the visual representations of the progress within a sprint and within the project as a whole.
- Shippable functionality: The final usable product that meets the customer's business goals and is considered done.

The five main practices that are key to Scrum are covered elsewhere in this book. However as a refresher they are:

Sprint Planning – 8 hours for monthly sprint

Daily scrum – 15 minutes

Sprint Review – 4 hours for monthly sprint

Sprint Retrospective– 3 hours for monthly sprint

The Definition of Scrum

Scrum is a process framework within which teams and organizations can address difficult and complex adaptive problems, while productively and creatively delivering products that are of the highest possible value.

Scrum is:

- Lightweight
- Simple to understand

- Difficult to master

So scrum has been around since the early 1990s. It is not a process or a technique for building products. Indeed it is a framework within which your company can employ various processes and techniques. Scrum makes clear the relative efficacy of your product management and development practices so that you can improve.

The Scrum Theory

Scrum is founded on empirical process control theory, otherwise known as empiricism. Empiricism asserts that knowledge comes from experience and making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk. Three pillars uphold every implementation of empirical process control: transparency, inspection, and adaptation. So let's have a closer look at these three pillars.

Transparency

Significant aspects of the process must always be visible to those responsible for the outcome. Therefore transparency requires those aspects be defined by a common standard so all observers share a common understanding of what is being seen.

Inspection

Scrum users should frequently inspect Scrum artifacts and the progress toward a Sprint Goal to detect undesirable variances. These inspections should not be so frequent however that inspection gets in the way of the work. The inspections are most beneficial when diligently performed by skilled inspectors at the point of work.

Adaptation

If the person doing the inspection determines that one or more aspects of a process deviate outside acceptable limits, and that the resulting product will be unacceptable, the process or the material being processed must be altered and adjusted. An adjustment should be made as soon as is possible to minimize further deviation.

Scrum prescribes four formal events for inspection and adaptation, these have already been mentioned but as a reminder they are:

- Sprint Planning
- Daily Scrum
- Sprint Review
- Sprint Retrospective

XP: Where the customer is put first



XP, or Extreme Programming is another popular approach to software development. The main focus of XP is complete customer satisfaction, in XP the customers needs always come first. XP teams achieve high customer satisfaction by developing required features when the customer needs them. New requests are part of the development team's normal daily routine, and the team must deal with requests whenever they pop up. The team organizes itself around any problem that arises and solves it as efficiently as they possibly can. The main XP practices are else in this [book](#).

Lean Programming: Producing JIT



Lean is not as popular as Scrum or XP and has not been discussed so far in this book. It is however a valid and productive framework for agile that should always be considered when a company is contemplating a move to agile. Lean has its origins in manufacturing. Way back in the depths of time, well ok the 1940's in Japan, a small little company called Toyota wanted to produce cars for the Japanese market but couldn't afford the massive investment that tooling up for mass production requires. Therefore they studied the way supermarkets worked, noting how consumers only buy what they need, secure with the understanding there will always be a supply of the goods they purchase. They also noted and how the stores restock shelves only as they empty. From this observation, Toyota created a JIT (just in time) process that it could translate to the factory floor.

The result was a significant reduction in inventory of parts and finished goods and a lower investment in the machines, people, and space required. The JIT process gives workers the ability to make decisions about what is most important to do next.

The workers take responsibility for the results. Toyota's success with JIT processes has helped change mass manufacturing approaches globally and as a result they are now one of the largest car producers in the world.

The seven tried and tested principles of lean manufacturing can be applied to software development. These can be used to optimize the whole IT value stream. The lean software development principles are eliminate waste, build in quality, create knowledge, defer commitment, deliver quickly, respect people, and optimize the whole.

KanBan



The Kanban method is a lean methodology that we have discussed elsewhere in this book. However as a refresher the two Kanban principles critical to success are:

- Visualizing the workflow: Teams use a Kanban board (whiteboard, corkboard or electronic board) that displays kanbans (indications of where in the process a piece of work or task currently is). The board is organized into columns, each one representing a stage in the process, a work buffer, or queue; and optional rows, indicating the allocation of capacity to classes of service. The board is updated by team members as work proceeds, and blocking issues are identified during daily meetings.
- Limit work in progress (WIP): Limiting WIP reduces average lead time, improving the quality of the work produced and increasing overall productivity of your team. Reducing lead time also increases your ability to deliver frequent functionality, which helps build trust with your stakeholders. To limit WIP, understand where your blocking issues are, address them quickly, and reduce queue and buffer sizes wherever you can.

Agile Modeling



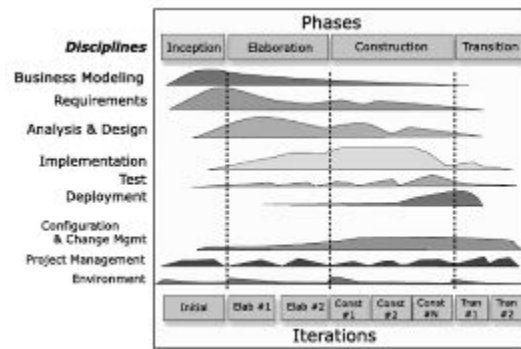
Agile Modeling (AM) is a collection of practices, principles and values that are used for modeling software that can be applied on a software development project in a lightweight and effective manner. AM was purposely designed to be a source of strategies that can be tailored into other base processes. With an Agile Model Driven Development (AMDD) approach, you typically do just enough high-level modeling at the beginning of a new project to understand the potential architecture and scope of the system under design. During construction iterations you do modeling as part of your iteration planning activities and then take a JIT model storming approach where you model for several minutes as a precursor to several hours of coding. AMDD recommends that teams take a test-driven approach to development although doesn't insist on it.

The Agile Modeling practices include the following:

- Active stakeholder participation: Stakeholders provide information, make decisions, and are actively involved in the development process.
- Architecture envisioning: This practice involves high-level architectural modeling to identify a viable technical strategy for your solution.
- Document continuously: Write documentation for your deliverables throughout the life cycle in parallel to the creation of the rest of the solution. Some teams choose to write the documentation one iteration behind to focus on capturing stable information.
- Document late: Write deliverable documentation as late as possible to avoid speculative ideas likely to change in favour of stable information.
- Executable specifications: Specify detailed requirements in the form of executable customer tests and your detailed design as executable developer tests.
- Iteration modeling: Iteration modeling helps identify what needs to be built and how.

- Just barely good enough artifacts: A model needs to be sufficient for the situation at hand and no more.
- Look-ahead modeling: Invest time modeling requirements you intend to implement in upcoming iterations. Requirements near the top of your work item list are fairly complex so explore them before they're popped off the top to reduce overall project risk.
- Model storming: Do JIT modeling to explore the details behind a requirement or to think through a design issue.
- Multiple models: An effective developer has a range of models in his toolkit, enabling him to apply the right model for the situation at hand.
- Prioritized requirements: Implement requirements in priority order, as defined by your stakeholders.
- Requirements envisioning: Invest your time at the start of an agile project to identify the scope of the project and create the initial prioritized stack of requirements.
- Single-source information: Capture info in one place only.
- TDD: Quickly code a new test and update your functional code to make it pass the new test.

Unified Process (UP)



The Unified Process (UP) is probably the least used methodology at this moment in time. However it is a valid framework that can provide a valuable solution should it suit your company needs. UP uses an iterative and incremental approach within a set life cycle and focuses on the collaborative nature of software development. It can be extended to address a broad variety of project types, including OpenUP, Agile Unified Process (AUP), and Rational Unified Process (RUP).

UP divides the project into iterations focused on delivering incremental value to stakeholders in a predictable manner. The iteration plan defines what should be delivered within the iteration, and the result is ready for iteration review or shipping. UP teams like to self-organize around how to accomplish iteration objectives and commit to delivering the results. They do that by defining and “pulling” fine-grained tasks from a work items list. UP applies an iteration life cycle that structures how micro-increments are applied to deliver stable, cohesive builds of the system that incrementally progress toward the iteration objectives.

UP structures the project life cycle into four phases, these are

- Inception
- Elaboration
- Construction
- Transition

The project life cycle provides stakeholders and team members with visibility and decision points throughout the project. This enables effective oversight and allows you to make “go or no-go” decisions at appropriate times. A project plan defines the life cycle, and the end result is a released application.

Not everything in agile is easy and straight forward. There are dangers involved with adopting agile blindly and pitfalls that need to be avoided at all costs. Below is a selection of the most common pitfalls that you and your organisation should be aware of, this is of course not a complete list and any move to agile should be carefully planned, discussed and time framed. Nothing should be committed to until everyone within your organisation is on board and understands what is involved (At least everyone who matters in the agile process, you can probably exclude the cleaning staff).

Do not become agile zombies. Companies can easily fall into the trap that if they attend an agile workshop and mandate to a certain easy to understand and out-of-the-box (OOTB) process, that they are now an ultra efficient, modern agile organisation, this is far from true agile. These companies train their teams to blindly follow and enforce the anointed process while not considering which practices may need to change to meet their organizations' unique needs and requirements. Remember agile isn't a prescribed process or set of practices; it's a philosophy and framework that can be supported by a practice and no two agile approaches are the same. One single OOTB methodology that fulfils all needs doesn't exist.

For any organisation an effective and workable agile adoption requires complete executive sponsorship and full support at the highest level. This involvement means more than simply showing up with a big smile at the kickoff meeting to say a few key words of encouragement and then disappearing back to the top floor and the golf course.

Without the executive's continued and proactive support of the overall initiative, the agile adoption is often doomed because agile initiatives require an upfront investment of resources and funding and continued backing during the initial stages. These are areas that executives typically control and if they are not fully behind the change or they do not completely understand what is required then the required funding is likely to fall short of what is essential.

Different agile companies find that they can fulfill the true spirit of the Agile Manifesto through different approaches. Ironically, most of these approaches focus on one phase or discipline within the delivery life cycle, oddly enough this goes against the underlying spirit that is lean, which has always advised us to consider the whole. You will find that most approaches focus on the construction phase.

Construction is typically a straightforward area to focus on when your company is taking on its agile transformation, but if companies only change the way they construct software, they should not really be

calling themselves agile, because they are not. The development teams could be evolving nicely along the agile path, happily delivering new working software every two or four weeks, but if the processes in other areas on the company only allow for deployment every four months or customer stakeholders aren't prepared to meet regularly, then the company is not realizing all the benefits agile can provide and as a result is not agile.

That well known but often ignored old adage, "If you fail to plan, plan to fail," is really true in the agile world. This is why the early forms of agile, typically RAD, had started to get a bad name.

Corners were often cut and the agile principles were not always fully adopted, as a result some early agile projects failed just as badly as earlier projects using methodologies such as waterfall. However for some companies the lesson was quickly learned, RAD became Agile and a reboot of the framework started.

Good planning is core to the success of any agile adoption no matter how large or small the company is. Companies should always answer these important questions and also understand the answers:

- Why do we want to be agile, and what benefits will agile provide?
- How will we achieve and measure agility?
- What cultural, technological or governance barriers exist, and how do we overcome them?

Without a plan that clearly shapes the agile initiative and includes addressing and resolving any known constraints to moving to agile, it is more difficult to control the initiative, staff it, fund it, manage blockers and maintain vital continued executive sponsorship.

You can quickly short circuit a planned agile adoption by focusing solely on a single software or system delivery team. While a single team can gain some benefit from agile, this will be limited. To be a truly successful agile company you need to look at the whole company adopting the agile processes. Agile should always be a change in culture for the entire organization. A good method to help this is to find champions in Operations, lines of business, product management, Marketing, and other functional areas to increase your success chances.

Moving to agile is very exciting and some people within your teams will want to rush in and start before all the planning and preparation is complete. This temptation is understandable but dangerous. If a proper roadmap for training, processes and tooling isn't outlined early in the adoption your company can soon run into issues that will affect confidence in the agile process.

Because an agile adoption isn't just a matter of a new delivery process, but is also major cultural shift, good training for the whole team is imperative and the process will fail without it. Programmers don't

like change and many people like working in their own little comfort world. As a result, the concept of not only changing the way they develop, but adding the concept that now they have to work closely with five, six, or ten other people all the time can be very frightening. An agile coach can work with these team members and help them through the initial phases of agile adoption.

Unfortunately some companies often see agile practice training, like coaching, as an area where they can save money, sending only a few key managers to learn the new process in hope that they will then train the rest of the teams while trying to implement the new approach.

How many times have you seen managers book themselves onto courses which involve four nights in a plush hotel. The course itself will always be better suited to the actual testers, programmers or project leaders but the managers see it as a rather nice few days out of the office and a bar bill the company will pay for.

Their intentions will be to return with new skills that they will then pass down but in reality time pressures and lack of interest with result in very little being fed back to the teams apart from jovial stories about what happened when they were all drunk.

If challenged about the lack of feedback by higher management then the response is often 'The course was a load of rubbish and no one else should be sent on it'. This is an age old problem that still exists to this day.

Therefore always remember agile involves a change in behavior and process. It is critical to send all team members to the appropriate training and provide them with ongoing training to reinforce agile values and update team members on processes that may have changed. If the right people are sent on the right courses the valuable skills will be picked up and this will increase the overall skill set of the company.

There are many classic mistakes that can take an agile project off the rails. Below is just a small sample of what can go wrong.

All you need is scrum

(Scrum is all you need)

Some companies believe it is possible to adopt agile without technical practices at all? Hopefully by now you will have realised this is not the case. However starting with scrum is never a bad idea. If you apply Scrum correctly and get everyone behind it you will inevitably decide to try some technical practices at a retrospective meeting.

What you should never do though is rely on Scrum solely as a process that will solve all the problems you encounter. It can do that, but only if you are open-minded and willing to try other various features such as pair programming.

The Scrum Master knows everything

The Scrum Master is not an all knowing, all seeing deity. While it is true they have some basic knowledge about scrum and some agile practices, but in many cases that's just it. They may have no hands-on experience with software development, project management or testing. As a result the scrum master should not act as a project manager and prescribe what to do and what not to do. These decisions should be made at a team level. The only real things the scrum master should care about are teams impediments and meta-process. The meta-process is the set of rules and procedures that allow the team to reflect and improve their existing development process.

We Can Live Without Customer Feedback

One of the strengths of agile is regular feedback. Feedback from the actual customer is the most important. If your team build something that is incorrect or does not fit the customer need then you need to know this sooner rather than later. Regular feedback from the customer helps keep the project on course and keeps you off the rocks. With feedback you are able to regularly correct the course should it be require. Remember the customer is a valuable team member and should be treated accordingly at all times.

Extreme programming recommends having the customer's spokes person on-site. While this is a great idea, it is rarely practical. However you can have the customer available remotely all the time to answer questions and communicate about a project. If you can't have feedback in a reasonable amount of time, agile methods simply will not work. You may build a technically perfect product, but yield zero customers' satisfaction in the end.

Self-organization is Easy

Scrum heavily advertises self-organization. Complexity theory has something to say about it as well. Self-organization is based on a set of simple rules, non-linearity and interactions between agents (in the case of scrum between people). You will never see self-organization working with just a set of rules. Self-organization in a software development team needs more, it needs leadership and cooperation.

Pure self-organization assumes that a leader will emerge. That does not happen frequently and in many cases the team will stagnate and fluctuate around mediocrity without a real leader. The leader sets a vision, motivates and pushes team to the right direction. The leader empowers confidence, passion and self-reflection. This leads to true self-organization eventually.

False Goal (E.g. The customer asked us to be agile)

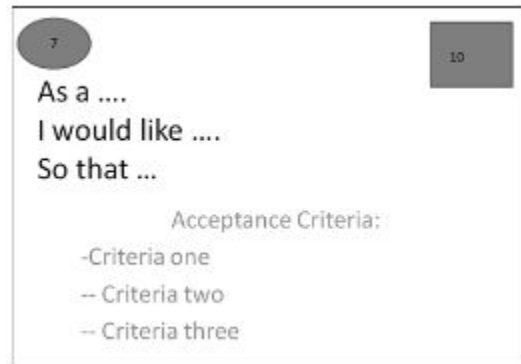
If you have a customer who insists you use agile process for their project you should smile and be happy with this gift. Use this chance as a turning point for agile adoption and a valid reason for moving forward.

Unfortunately, many companies still just try to “emulate” agile adoption with a desire to get this contract. They will reluctantly send some people to cheap agile courses. They will also purchase an agile project management tool and apply Scrum superficially.

They do all that without deep goals and culture change within the company. As a result there will be no real passion or desire to change fully. They do all that with a false goal, money. Almost inevitably there will be the following symptoms:

- The sprints will fail.
- No commitment will mean poor code.
- The scrum master will be ineffective and un-motivated.
- There will be little or no testing phase in each sprint resulting in bug ridden code.

The result of “false goal” agile adoption is failure and a long-term disappointment with agile software development.



7

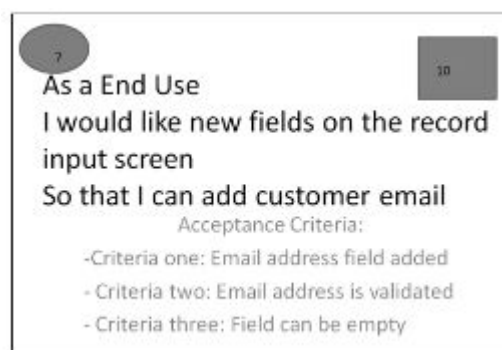
10

As a
I would like
So that ...

Acceptance Criteria:

- Criteria one
- Criteria two
- Criteria three

When stakeholders realize the need for a new software system, feature set, or application, the agile process will begin with the appointed product owner defining what the new or updated software will do and what services it will provide to the end users. Instead of following the more traditional process of product managers and business analysts writing lengthy requirements or specifications, agile takes a more efficient lightweight approach of writing down brief descriptions of the pieces and parts that are needed. These become work items and are captured in the form of user stories. A user story is a simple description of a product requirement in terms of what that requirement must accomplish for whom.



7

10

As a End Use
I would like new fields on the record
input screen
So that I can add customer email

Acceptance Criteria:

- Criteria one: Email address field added
- Criteria two: Email address is validated
- Criteria three: Field can be empty

Therefore user stories are the agile form of requirements specifications, and should explain how the system should behave with respect to a single, coherent feature or function. User stories are a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. Each story should be small enough to be

completed in a single sprint. Larger collections of related features, or a collection of sub-features that make up a single complex feature, may be referred to as "epics".

User stories are often written on index cards or sticky notes. These will then be arranged on boards, walls or tables to help facilitate the initial planning and discussion.

As a result they strongly shift the focus from writing about features to discussing them. In reality the following discussions are more important than whatever text is written.

Epics may include user stories for different development teams. These collections may be developed over a series of sprints. Each epic and its user stories should have associated acceptance criteria. The typical format for a User Story is thus:

Title	a name for the user story
As a	user or persona
I want to	take this action
So that	I get this benefit

The story should also include at least one validation step, these are steps to take to know that the working requirement for the user story is correct. That step is usually worded as follows:

- When I <take this action>, this happens <description of action>

Some typical examples are shown below.

- **Searching for customers**
 - As a user, I want to search for my customers by their status, first and last names so I can see who is still active.
- **Modify my diary**
 - As a user, I want to modify my own diary but not the dairy of other users.

User Stories are a central part of many agile development methodologies, such as in the XP's planning game. User stories define what has to be built in the software project development lifecycle. User stories can be written by any member of the team but they are always prioritized by the product owner to indicate which have the most value for the system. They will then be broken down into tasks and estimated by the developers.

When user stories are about to be implemented in a pending sprint the developers should have the option to talk to the product owner about it. The short stories may be difficult to interpret, may require some background knowledge or the requirements may have changed since the story was written. Clarity is very important in agile and communication is the key tool.

Every user story must at some point have one or more acceptance tests attached, allowing the developer to test when the user story is done and also allowing the customer to validate it.

Without a precise formulation of the requirements, prolonged destructive and costly arguments may arise when the product is to be delivered.

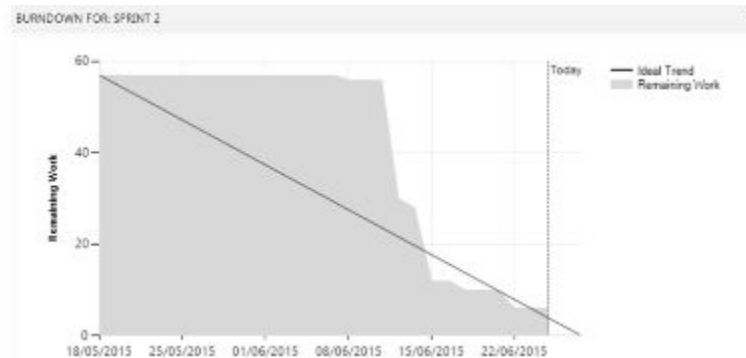
User stories are a great benefit to the agile team because they offer a quick way of handling the product owner requirements without having to create large, cumbersome formalized requirement documents and without performing administrative tasks related to maintaining them.

A good agile project will quickly gather user stories in order to respond faster and with less overhead to rapidly changing real-world requirements.

XP and other agile methodologies have always favoured face-to-face communication over comprehensive documentation and quick adaptation to change instead of fixation on the problem.

User stories can achieve this by:

- They allow the breaking of projects into small increments.
- Being small and simple they need very little maintenance.
- They make it easier to estimate development effort.
- They help maintain a close customer contact.
- They are suitable for projects which have badly written or poorly understood requirements. Iterations of discovery soon drive the refinement process and improve understanding.



Burndown charts are graphical artefacts that provide very useful information for the scrum team. There are two types of Burndown charts. The most commonly used type is the Iteration or sprint chart. There is also another type known as the Release Chart which is not as common.

These charts are a graphical representation of the work left to do versus the time remaining. The outstanding work (also known as the backlog) is usually on the vertical axis while the time along on the horizontal. The iteration Burndown chart is often used in agile as a metric for measuring progress over time and as a result they are a good tool for working out team velocity.

The main concept behind team velocity is to help agile teams estimate how much work they can reasonably complete within a given period of time. The Burndown charts help achieve this by giving a bench mark of how quickly similar work was previously completed. The following terminology is commonly used in velocity tracking.

- **Unit of work:** The unit is chosen by the team to measure velocity. This can either be a real unit like hours or days or an abstract unit like story points or ideal days, the choice will vary from team to team. Each task in the software development process should then be valued in terms of the chosen unit.
- **Interval:** The interval is the duration of each individual iteration in the software development process for which the velocity is measured. The length of an interval is determined by the team. Most often, the interval is a week, but it can be as long as a month, although no longer.

Below is a breakdown of the elements of a Burndown chart

X-Axis	The iteration/project timeline
--------	--------------------------------

Y-Axis	The work that needs to be completed for the iteration/project. The time or story point estimates for the work remaining will be represented by this axis.
Project Start Point	This is the farthest point to the left of the chart and occurs at day 0 of the iteration/project.
Project End Point	This is the point that is farthest to the right of the chart and occurs on the predicted last day of the iteration/project.
Ideal Work Remaining Line	This is a straight line that connects the start point to the end point. At the start point, the ideal line shows the sum of the estimates for all the tasks (work) that needs to be completed. At the end point, the ideal line intercepts the x-axis showing that there is no work left to be completed. This line is a mathematical calculation based on estimates, and the estimates are more likely to be in error than the work. The goal of a burn down chart is to display the progress toward completion and give an estimate on the likelihood of timely completion.
Actual Work Remaining Line	This shows the actual work remaining. At the start point, the actual work remaining is the same as the ideal work remaining but as time progresses; the actual work line fluctuates above and below the ideal line depending on how effective the team is. In general, a new point is added to this line each day of the project. Each day, the sum of the time or story point estimates for work that was recently completed is subtracted from the last point in the line to determine the next point.

To calculate the team velocity, the team first has to determine how many units of work each task is worth and the length of each individual iteration. During development, the team will then keep track of all completed tasks and, at the end of the interval, count the number of units of work completed during the interval. The team then writes down the calculated velocity in a chart or on a graph which is then made available for all interested parties.

It is normal for the first few weeks to provide little data of value, but is essential to provide a basis for comparison. Each week after that, the velocity tracking will provide better information as the team provides better estimates and becomes more used to the methodology. Also velocity should also improve as the team becomes more experienced and more confident in their abilities to achieve within agile.



Test automation at all levels of testing occurs in many agile teams, and this can mean that testers spend time creating, executing, monitoring, and maintaining automated test cases and the end results. Because of the heavy use of test automation, a higher percentage of the manual testing on agile projects tends to be done using experience-based and defect-based techniques such as software attacks, exploratory testing, and error guessing. While developers should focus on creating unit tests, testers should focus on creating automated integration, system, and system integration tests which can be reused with little or no modification in a later development cycle. This leads to a tendency for agile teams to favour testers with a strong technical and test automation background. Such highly skilled testers are in great demand and can expect better employment terms than lesser skilled, manual testers.

One goal of the automated tests is to confirm that the build is indeed functioning reliably and is installable. If any automated test fails, the agile team should fix the underlying defect in time for the next code check-in whenever possible.

While this requires an investment in real-time test reporting to provide good visibility into test results, the payback in results makes this effort very worthwhile. In particular this approach helps reduce expensive and inefficient cycles of "build-install-fail-rebuild-reinstall" that can occur in many traditional projects, since changes that break the build or cause software to fail to install are detected quickly.

One of the more popular automation tools on the market today is Selenium. You may well have heard of selenium and possibly even used it. If not then you are probably thinking; what on earth is this Selenium thing?


Well selenium is a portable software testing framework for web applications. Selenium IDE provides a record and playback tool for authoring tests without learning a test scripting language.

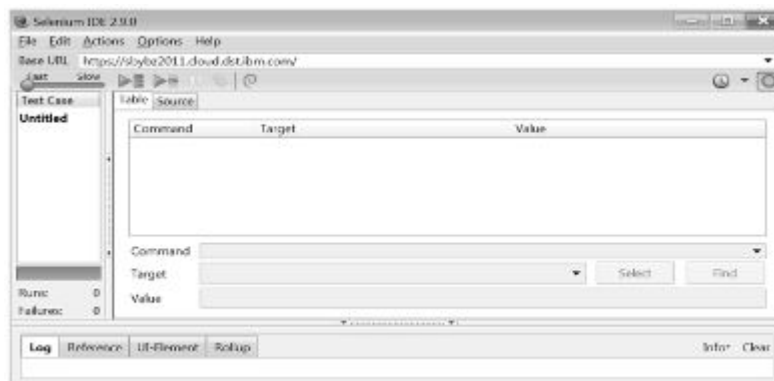
Selenium also provides a test domain-specific language called Selenese to write tests in a number of popular programming languages.

These languages include Java, C#, Perl, PHP, Python and Ruby. These tests can then be modified, saved and run against most modern web browsers. Another advantage of selenium is that it is cross platform and it will deploy on the Windows, Linux, and Macintosh platforms. It is open-source software, released under the Apache 2.0 license, and can be downloaded and used without charge, another good bonus.

To get going with selenium you first need to download and install Selenium IDE, this is a add on for Firefox which allows you to record your actions as you test a web application, The IDE at the time of writing was version 2.9.0 and can be downloaded here: <http://www.seleniumhq.org/download/>. Please be aware this version is subject to change and by the time you read this book it will almost certainly be higher.

Once you have downloaded and installed the plugin you are ready to record your first test, give it a go and follow these steps.

1. Open the web site you want to record actions for in the Firefox web browser
2. Now open the Selenium IDE by clicking the icon  on the Firefox toolbar, you should now see a window similar to this example
- 3.



4. Create a new test case in Selenium IDE using the menu: File → New Test Case
5. Enable recording by clicking the red circle from the top-right side of the Selenium IDE
6. Click on the browser
7. Create the test case by recording the required actions for the given site.
8. Click the Selenium IDE button on the Firefox toolbar.
9. Disable recording by clicking the red circle again.
10. Save the new test case using the menu: File → Save Test Case As; save it with a name that is logical for the Test case.
11. Run the new test case by clicking on the run button from the toolbar; watch the test case being executed in the Firefox browser
12. Save the code of the test case in the Java / Junit 4 / Remote Control format using the menu: File → Export Test Case As → Java / Junit 4 / Web Driver; save the code with selected file name it give it the extension .java
13. Open the folder where the .java file was saved
14. Open the file using Notepad++

15. The file shows the java code for the recorded test case: Below is a typical example of the generated code.

```
import java.util.regex.Pattern;
import java.util.concurrent.TimeUnit;
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.Select;

public class AcmeXYZUsers2 {
    private WebDriver driver;
    private String baseUrl;
    private boolean acceptNextAlert = true;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        driver = new FirefoxDriver();
        baseUrl = "http://xxxxxxxxx.com/";
        driver.manage().timeouts().implicitlyWait(30,
TimeUnit.SECONDS);
    }

    @Test
    public void testAcmeXYZUsers2() throws Exception {
        driver.get(baseUrl + "/");
        driver.findElement(By.id("userTxt")).clear();
        driver.findElement(By.id("userTxt")).sendKeys("xxxxxx");
        driver.findElement(By.id("loginBtn")).click();
        driver.findElement(By.linkText("Users")).click();
        driver.findElement(By.linkText("New User")).click();
        driver.findElement(By.id("passTxt")).clear();
        driver.findElement(By.id("passTxt")).sendKeys("xxxxxx");
        driver.findElement(By.id("useridTxt")).clear();
        driver.findElement(By.id("useridTxt")).sendKeys("testID");
        driver.findElement(By.id("nameTxt")).clear();

        driver.findElement(By.id("nameTxt")).sendKeys("testName");
        driver.findElement(By.id("passTxt")).clear();

        driver.findElement(By.id("passTxt")).sendKeys("pa22word");
        driver.findElement(By.id("saveBtn")).click();
        driver.findElement(By.id("GridView1_lnkEdit_2")).click();
    }
}
```

```

driver.findElement(By.id("nameTxt")).clear();

driver.findElement(By.id("nameTxt")).sendKeys("testNames");
driver.findElement(By.id("saveBtn")).click();

driver.findElement(By.id("GridView1_lnkDelete_2")).click();
driver.findElement(By.id("GridView1_btnNo_2")).click();

driver.findElement(By.id("GridView1_lnkDelete_2")).click();
driver.findElement(By.id("GridView1_btnYes_2")).click();
}

@After
public void tearDown() throws Exception {
    driver.quit();
    String verificationErrorString =
verificationErrors.toString();
    if (!"".equals(verificationErrorString)) {
        fail(verificationErrorString);
    }
}

private boolean isElementPresent(By by) {
    try {
        driver.findElement(by);
        return true;
    } catch (NoSuchElementException e) {
        return false;
    }
}

private boolean isAlertPresent() {
    try {
        driver.switchTo().alert();
        return true;
    } catch (NoAlertPresentException e) {
        return false;
    }
}

private String closeAlertAndGetItsText() {
    try {
        Alert alert = driver.switchTo().alert();
        String alertText = alert.getText();
        if (acceptNextAlert) {
            alert.accept();

```



```
    } else {  
        alert.dismiss();  
    }  
    return alertText;  
} finally {  
    acceptNextAlert = true;  
}  
}
```

So that is your first automated test, cool or what? Hopefully you are now infused and raring to move forward with automated testing. The next stage is to create a test case in Eclipse (Java) or Visual Studio (C#). So read on intrepid testers.

This chapter is designed to assist you in setting up Eclipse and Selenium on your system. Once setup we will then go through the creation, modification and running of a typical automated test plan. This is not designed to turn you into a Selenium guru. There are plenty of dedicated books available for that, rather I will be showing you how to get started with Selenium with some of the basics.

To get going you will need the following tools installed.

- [Java JDK](#)
- [Eclipse](#)
- Selenium IDE (Covered in previous chapter)
- [Firebug and Firepath](#)

If you have already installed these then please feel free to skip this section.

The Java JDK

First you will need to install the Java JDK (Java Development Kit). At the time of going to press the download link for this could be found at this URL:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

However please be aware this could change and you might prefer to search for Java JDK instead. Once you locate the download option that matches your environment you should accept the license agreement and start the download.

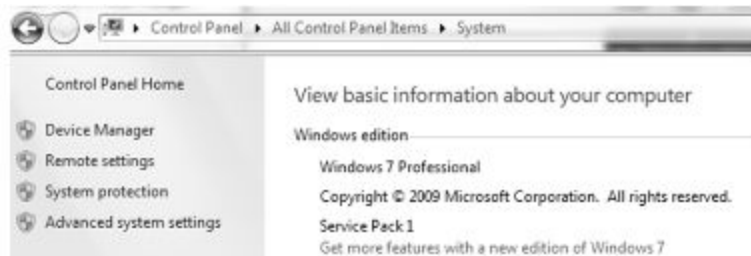
Java SE Development Kit 8u73		
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.		
<input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	77.73 MB	jdk-8u73-linux-arm32-vfp-hflt.tar.gz
Linux ARM v6/v7 Hard Float ABI	74.68 MB	jdk-8u73-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.75 MB	jdk-8u73-linux-i586.rpm
Linux x86	174.91 MB	jdk-8u73-linux-i586.tar.gz
Linux x64	152.73 MB	jdk-8u73-linux-x64.rpm
Linux x64	172.91 MB	jdk-8u73-linux-x64.tar.gz
Mac OS X x64	227.25 MB	jdk-8u73-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.7 MB	jdk-8u73-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.08 MB	jdk-8u73-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.36 MB	jdk-8u73-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u73-solaris-x64.tar.gz
Windows x86	181.5 MB	jdk-8u73-windows-i586.exe
Windows x64	186.84 MB	jdk-8u73-windows-x64.exe

When the download is complete you are then able to install the JDK using the default options as they appear. If you are wondering what the Java SDK actually is then read on. The Java JDK is a software development kit that is supplied by Oracle. It is a development environment for developing and testing applications using the Java programming language. The JDK also includes useful tools such as Application Servers and Debuggers.

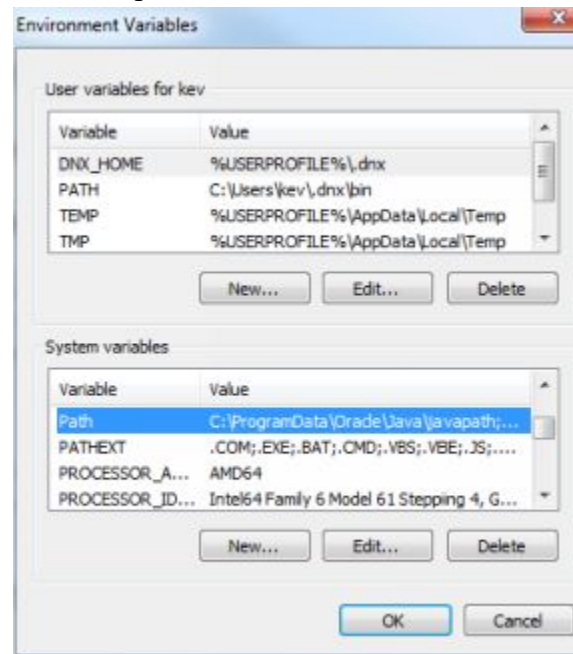
When complete there will be one final task before moving onto Eclipse. The next step involves adding the path to the JDK into the system windows. This ensures any application, such as eclipse, is able to find it.

To complete this task follow the steps below.

- Open Control Panel
- Select System -> Advanced System Settings



- Then select Advanced -> Environment Variables
- In System variables find the path value, select and click Edit



- The next step is important append, do not replace, the install path of the Java SDK to the end of the path string.
- Example path: C:\Program Files\Java\jdk1.8.0_60.
- Click OK to save the change
- To test this change open a command prompt and type in "java -version". You should then see confirmation of the Java version you installed.

```
C:\Users\kev>java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

Eclipse

Next if it is not already installed, eclipse should be downloaded and installed. Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment to your particular needs.

Although eclipse is written primarily in Java and its primary use is for developing Java applications it can also be used to develop applications in other languages. With the use of plugins eclipse can be used to create programs in C, C++, COBOL and PHP to name just a few. The true list is much more extensive but it is not required for the scope of this book.

To install eclipse you first need to download the latest version from the eclipse web site. At the time of publication this URL was:

<http://www.eclipse.org/downloads/>

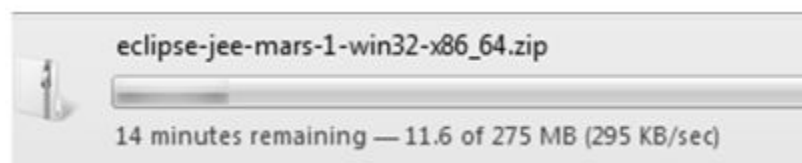
So navigate to this URL and select your correct flavor for your system. There are currently options for:

- Mac OS X 64 bit
- Windows 32 bit
- Windows 64 bit
- Linux 32 bit
- Linux 64 bit














An example Windows option is shown below



When you select the correct flavor you will be asked to select a download mirror. Whichever location you choose is relevant as long as you have selected the correct version. When selected a compressed (zip) file will be downloaded to your default download folder.



Once complete you will be able to uncompress the folder and place it into your preferred location (for example c:\program files\eclipse). When complete you should see a folder with similar contents to those shown below.

	configuration	5/23/2015 8:07 AM	File folder	
	dropins	2/19/2015 2:26 AM	File folder	
	features	5/23/2015 8:07 AM	File folder	
	p2	5/23/2015 8:08 AM	File folder	
	plugins	5/23/2015 8:11 AM	File folder	
	readme	5/23/2015 8:11 AM	File folder	
	.eclipseproduct	5/23/2015 8:11 AM	ECLIPSEPRODUCT...	1 KB
	artifacts	5/23/2015 8:11 AM	XML Document	121 KB
	eclipse	5/23/2015 8:11 AM	Application	314 KB
	eclipse	5/23/2015 8:11 AM	Configuration sett...	1 KB
	eclipsec	5/23/2015 8:07 AM	Application	26 KB
	epl-v10	5/23/2015 8:07 AM	Firefox HTML Doc...	13 KB
	notice	5/23/2015 8:07 AM	Firefox HTML Doc...	9 KB

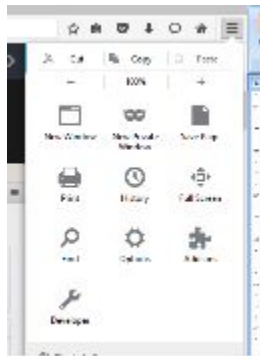
At this point you should be able to launch eclipse and load a workspace, for the purposes of this publication it is assumed you have the basic knowledge for this, if not please refer to appendix 1 for a list of good Eclipse books.

Firebug and Firepath



The next tool to install is the Firebug plug-in for Firefox. This useful web development tool integrates seamlessly with Firefox to put a useful collection of web development tools at your disposal while you browse websites that require automated testing. With Firebug you can edit, debug, and monitor CSS, HTML, and JavaScript live in any web page in real time. Another good feature of Firebug is, just like all the other tools discussed here, it is free.

To install the Firebug plug-in you first need to launch Firefox. When Firefox has fully loaded you should then select Add-ons from the menu options, see below.



Next make sure 'Get Add-ons' is highlighted. In the Search all add-ons search box type in Firebug and click the search icon.



If all goes well you will see something like this.



So, click on the Install button and wait for the plug-in to download and install. When done you should see something like this.



So that is Firebug installed and ready for use. Next we need to install Firepath. So now type Firepath in the search box and click the icon.



Hopefully you will see something similar to this.



Click install and wait, when done you will see a similar page to this.



Please note that you will need to restart Firefox for this plug-in to become active. You can do this when ready. Some of these tools will be used in later chapters of this fine book, you will also find links to some dedicated Eclipse/Selenium books in Append 1.

In the next chapter we will investigate setting up Microsoft Visual Studio for testing in Selenium for those who prefer ASP.Net and C#.

Selenium and ASP.Net

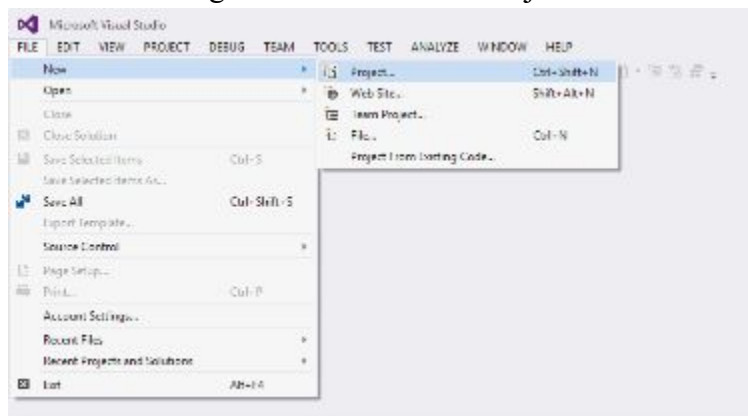
Selenium can be integrated into Visual Studio just as easily as it can be used with Eclipse. So if you are a C# type of person rather than a Java fan then fear not, you can still enjoy selenium with your preferred IDE. If you do not already have it then at the time of publication there is a free version of Visual Studio available. This is called Visual Studio Community 2015 (VSC2015) and it is available at this URL:

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>

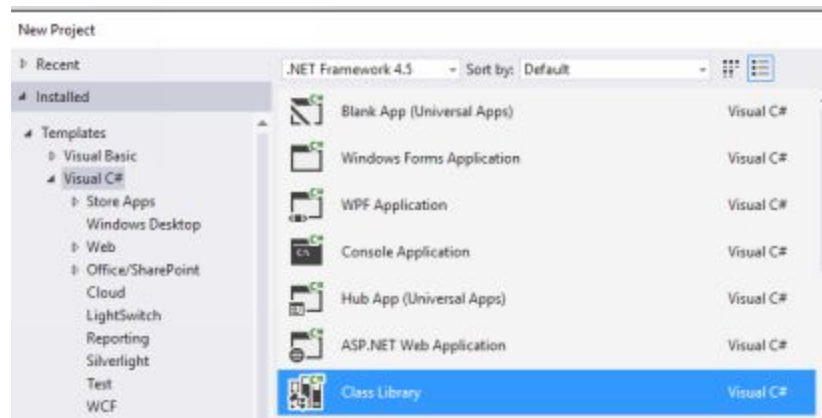


This is a fully functional version so if you have not yet installed VSC2015 then get your copy now. The initial download is quite fast but the actual install can take a while, however it only has to be done once. Once VCS2015 has been installed the next step is to create a new project and then download and add the plug-in.

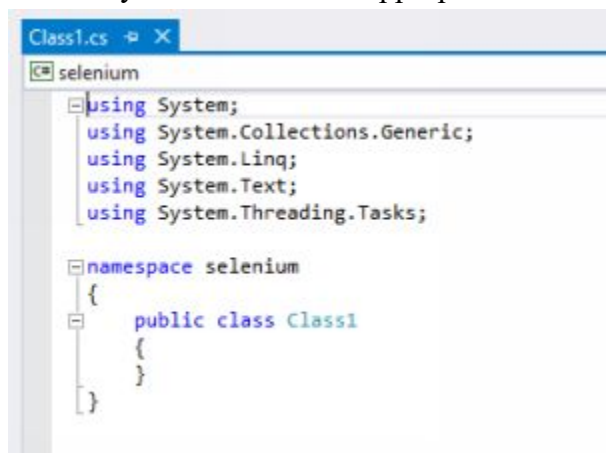
So first Launch Visual Studio and navigate to File > New > Project.



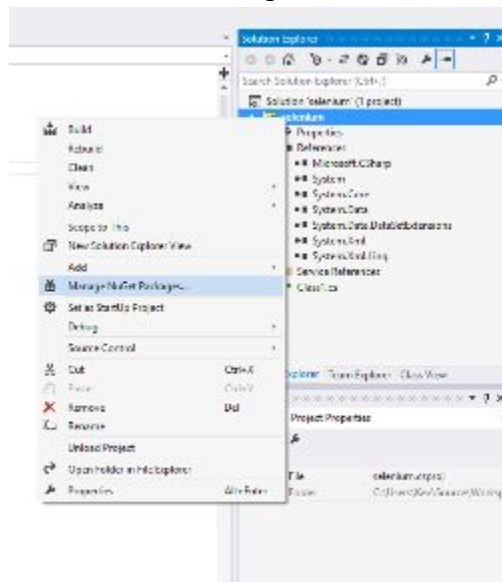
Then Select Visual C# > Class Library > Your project name > Click the OK button.



A Class1.cs file will be created. This you can rename as appropriate.



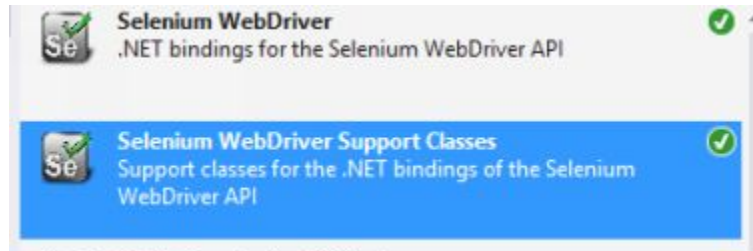
Next right click on the Project file in the Solution Explorer Window and Select Manage NuGet Packages.



Now conduct a search using Online and the search term Selenium the options for both the Selenium WebDriver and Selenium WebDriver Support Classes should appear. Ensure you install both of these packages to your project, they are essential.



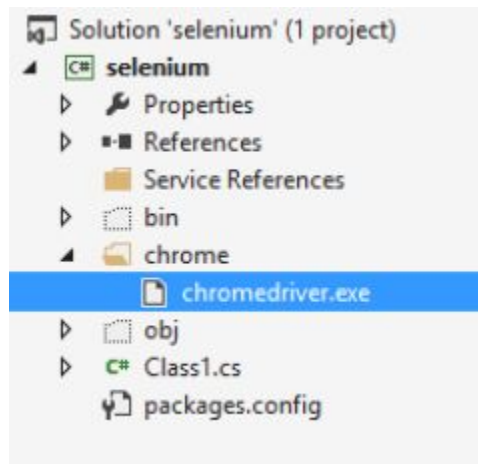
Install is a simple one button click for each package; the final result will be similar to below.



While both the Internet Explorer and Firefox drivers are included in the Selenium package, you will need to download an additional web driver for Chrome should you wish to test with that web browser as well? At the time of publication the download URL was:

<https://sites.google.com/a/chromium.org/chromedriver/downloads>, once again please be aware this may change over time and a search may be required.

When and if you download this driver you need to copy it to your project, one option is as per the example below, here the driver is copied to a folder called chrome.



Now right click on the chromedriver.exe and select Properties. Ensure the Build Action is set to the value of [Content]. Also check that Copy to Output Directory is set to the [Copy Always] value. This will ensure that chromedriver.exe is always in the folder of the running assembly so it can be used.

So that is it. You now have an empty test class which is configured for basic Selenium testing. Next we will start to look at some of the more important Selenium commands and see how they can help you create reliable, re-usable automated test cases.

To get this section going let us start with a very simple example which will do one thing, that is load a web page in maximized mode. That is all it will do but it is a good example of how simple things can be.

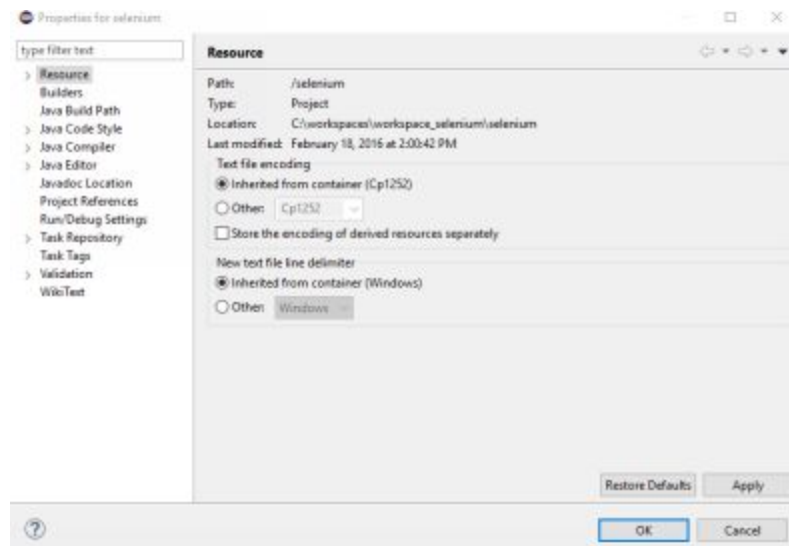
Below is the code for Eclipse. So if your flavor is Java then load eclipse and create a new workspace. Follow this up with a new java project; the names are at your discretion.

Next create a new class and call it kevsTest, when done add the code below.

```
Import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

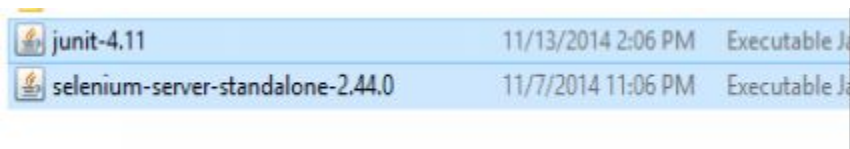
public class kevsTest {
    WebDriver driver = new FirefoxDriver();
    @Test
    public void test() {
        driver.manage().window().maximize();
        System.out.print("Window maximise");
        driver.get("http://www.kevsbox.com");
        System.out.print("Site is up");
        driver.quit();
        System.out.print("End of Testing");
    }
}
```

At this point you will probably see errors, this will due to the fact the required WAR files are not in the project build path. To rectify this right click on the project name and select Properties.

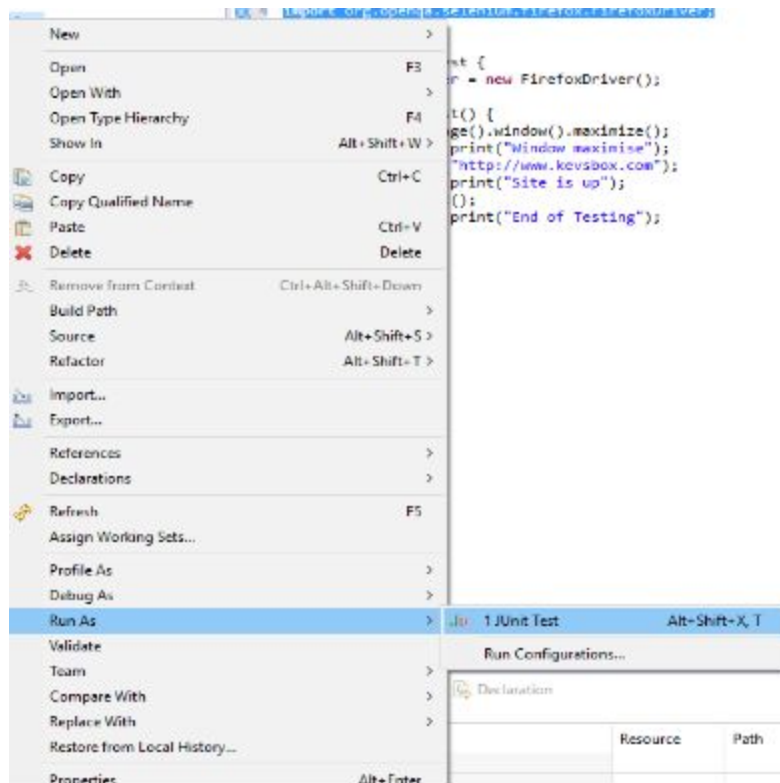


Now select Java Build Path -> Libraries -> Add External Jars...

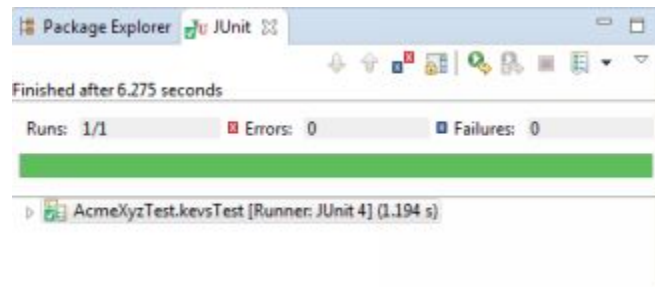
Now browse to the folder where you are storing the selenium drivers and highlight the required files.



At this point the project should rebuild and the errors should disappear. When ready save the test case. So now you have created this test case you must be itching to run it. So right click on the new test case and select Run As -> J Unit test as show below.



Eclipse will try to run your test case using Junit and on completion of execution, you should see your result in Junit pane as show below.



OK, so that is your first eclipse test run, now let us do the same thing in Visual Studio.

The code is very similar, see below. So give it a go if you have Visual Studio installed, the result will be the same.

```

using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium;

namespace selenium
{
    [TestClass]
    public class Class1
    {

```

```
private IWebDriver driver;

[TestMethod]
public void TheTest()
{
    driver = new FirefoxDriver();
    driver.Manage().Window.Maximize();
    driver.Navigate().GoToUrl("http://www.kevsbox.com");
    driver.Quit();
}
}
```

So there you have it, your first automated tests have been successfully run. Next we will look at some of the more complex commands available for your test cases.

Finding elements

No matter if you are using Java or C#, one of the critical requirements of automated testing will always be the ability to identify and locate elements from the web page under test and then being able to perform tests on these elements that will confirm the data returned is both valid and correct. This means that the test tool in use must be able to recognize the web elements quickly, correctly and effectively.

Selenium WebDriver is a tool that provides one of the most advanced techniques for locating elements on web pages in various popular web browsers. Selenium's very feature-rich API provides reliable multiple locator strategies such as ID, Name, CSS selectors, XPath etc. With Selenium you are also able to implement custom locator strategies for locating elements. Essentially these locators are the mainstay of your tests. Using the right locator for the situation ensures the tests are faster, more reliable and have a lower maintenance overhead in future releases. In any web development project regardless of what development language is being used it is always good practice to assign meaningful attributes such as Name, IDs or Class to all of the elements that exist on the web page. This makes the application more readable, testable and conforms to existing accessibility standards. There are rare occasions however when following these practices is simply not possible. For such scenarios you will have to use advanced locator strategies such as CSS selector and the XPath function.

While both CSS selector and XPath are popular among most Selenium users, the CSS selector option is always recommended over XPath due to its simplicity, speed, and performance advantages.

How to do it...

Locating elements in Selenium WebDriver is done by using the `findElement()` and `findElements()` methods provided by `WebDriver` and `WebElement` class.

The `findElement()` method returns a `WebElement` object based on a specified search criteria or throws up an exception if it does not find any element matching the search criteria.

The `findElements()` method returns a list of `WebElements` matching the search criteria. If no elements are found, it returns an empty list.

Find methods take a locator or query object as an instance of By class as an argument. Selenium WebDriver provides By class to support various locator strategies.

The following table lists various locator strategies currently supported by the Selenium WebDriver:

Strategy	Syntax
By ID	Java: driver.findElement(By.id(<element ID>)) C#: driver.FindElement(By.Id(<elementID>))
By name	Java: driver.findElement(By.name(<element name>)) C#: driver.FindElement(By.Name(<element name>))
By class name	Java: driver.findElement(By.className(<element class>)) C#: driver.FindElement(By.ClassName(<element class>))
By tag name	Java: driver.findElement(By.tagName(<htmltagname>)) C#: driver.FindElement(By.TagName(<htmltagname>))
By link text	Java: driver.findElement(By.linkText(<linktext>)) C#: driver.FindElement(By.LinkText(<linktext >))
By partial link	Java: driver.findElement(By.partialLinkText(<linktext>)) C#: driver.FindElement(By.PartialLinkText(<linktext >))
By CSS	Java: driver.findElement(By.cssSelector(<css selector>)) C#: driver.FindElement(By.CssSelector(<css selector >))
By XPath	Java: driver.findElement(By.xpath (<xpath query expression>)) C#: driver.FindElement(By. XPath(<xpath query expression>))

How to do it...

Locating elements using id, name, or class attributes is the preferred way to find elements in Selenium. So let's try using these methods to locate elements as described in the following sections.

Locate by ID

By far Ids are the preferred method to locate elements on a web page. This is because The W3C standard recommends that developers provide an id attribute for elements that are unique to each element. Having a unique id attribute provides a very explicit and reliable way to locate elements on the page. If for any reason the Ids are not unique or they are auto-generated this method should not be used.

With this method, the first element with the id attribute value matching the location will be returned. If no element has a matching id attribute, a *NoSuchElementException* will be raised.

Below is an example of how to use this method

```
<form name="userId">Login  
User ID: <input id="userid" type="text" name="login" />  
Password: <input id="password" type="password" name="password" />  
<input type="submit" name="signin" value="SignIn" />  
</form>
```

As you can see both input box have an id value, these can be used to locate the element. The example code for this is shown below.

```
WebElement elementUsername = driver.findElement (By.id ("userid"));
```

Locate by Name

The name attribute is another fast way to locate an element. However you must also be aware that the name may not be unique. With this method the first matching element will be returned, if no element has a matching name attribute, a `NoSuchElementException` will be raised.

Below is an example of how to use this method

```
<form name="userId">Login  
User ID: <input id="userid" type="text" name="login" />  
Password: <input id="password" type="password" name="password" />  
<input type="submit" name="signin" value="SignIn" />  
</form>
```

As you can see both input box have an id value, these can be used to locate the element. The example code for this is shown below.

```
WebElement elementUsername = driver.findElement (By.name ("login"));
```

Locate by XPath

XPath is the language used for locating nodes in any XML document. As HTML can also be an implementation of XML (XHTML), you lucky Selenium users can utilize this powerful language to locate elements in the web applications under test. XPath extends way beyond the simpler methods of locating by id and name attributes, it opens up new possibilities such as locating the fifth checkbox on the web page under test.

One of the main reasons for using XPath is when you don't have a suitable id or name attribute for the element you wish to locate (for example they are not unique). You can use the XPath feature to either locate the element in absolute terms, or relative to an element that does have an id or name attribute. This is not as preferred as ID or Name and can make your test cases less robust but there are times when no other option is available.

Below is an example of how to use this method

```
<html>
<html> <body>
  <form id="loginForm">
    <input name="userid" type="text" />
    <input name="password" type="password" />
    <input name="cButton" type="submit" value="Login" />
    <input name="cButton" type="button" value="Clear" />
  </form>
</body></html>
```

In this example the form elements can be located like this:

- login_form = driver.find_element_by_xpath("/html/body/form[1]")
- login_form = driver.find_element_by_xpath("//form[1]")
- login_form = driver.find_element_by_xpath("//form[@id='loginForm']")

The username element can be located like this:

- username = driver.find_element_by_xpath("//form[input/@name=userid]")
- username = driver.find_element_by_xpath("//form[@id='loginForm']/input[1]")
- username = driver.find_element_by_xpath("//input[@name=userid]")

Locate by LinkText and PartialLinkText

This is a very useful method to use when you know what the link text will be within an anchor tag. With this method, the first element with the link text value matching the location will be returned. If no element has a matching link text attribute, a *NoSuchElementException* will be raised instead.

Below is an example of how to use this method

```
<html> <body>
  <p>Are you sure you want to delete this record?</p>
  <a href="delete.html">Confirm</a>
  <a href="cancel.html">Cancel</a>
</body></html>
```

The find by link command will be

- `continue_link = driver.find_element_by_link_text('Confirm')`

Also the find by partial link will be

- `confirm_link = driver.find_element_by_partial_link_text('Confi')`

Locate by Tag Name

Use this when you want to locate an element by tag name. This is a limited method which is used less frequently than the methods already discussed. With this method, the first element with the given tag name will be returned. If no element has a matching tag name, a *NoSuchElementException* will be raised instead.

Below is an example of how to use this method

```
<html> <body>
  <h1>Welcome to Kevsbox.com</h1>
  <p>This is one cool site</p>
</body></html>
```

The find by link command will be

- `headingLink = driver.find_element_by_tag_name('h1')`

Locate by CSS

(Cascading Style Sheets)

Use this method when you want to locate an element by CSS selector syntax. With this method, the first element with the matching CSS selector will be returned. If no element has a matching CSS selector then as per usual a *NoSuchElementException* will be raised.

Below is an example of how to use this method

```
<html> <body>  
  <p class="content">This is kevsbox.com</p>  
</body></html>
```

In this example the “p” element can be located like this:

```
content = driver.find_element_by_css_selector('p. Content')
```

Browser Commands

In this section we will discuss some of the browser commands which are available to you for use in your automated scripts. This is by no means a complete list but I am sure you will find these commands very useful in the future.

Get

The Get command is used to load a new web page in the current browser window, below are two examples of how this works.

- `driver.get("http://www.kevsbox.com");`

or

- `String URL = "http://www.kevsbox.com";`
- `driver.get(URL);`

GetTitle

The getTitle method fetches the Title of the current page. This method accepts nothing as a parameter and returns a String value, for example.

- `String Title = driver.getTitle();`

GetCurrentURL

The getCurrentURL method returns a string representing the Current URL which is opened in the browser, for example.

- `String CurUrl = driver.getCurrentUrl();`

GetPageSource

The getPageSource method returns the Source Code of the page, for example.

- `String pSource = driver.getPageSource();`

Close

This method closes only the current window the WebDriver is currently controlling, for example.

- `driver.close();`

Quit

This method closes all windows opened by the WebDriver, for example.

- `driver.quit();`

Switching windows

The GetWindowHandle Command is used to get the window handle of the current window. This command returns a string value, as per this example:

- `String handle= driver.getWindowHandle();`

The GetWindowHandles Command is used to return the window handle of all the current windows. This command returns a set of strings, as per this example:

- `Set<String> handle= driver.getWindowHandles();`

The SwitchTo Window Command allows your test cases to move between named windows as per the example below:

- `driver.switchTo().window("windowName");`

Radio buttons and checkboxes

In Selenium, Radio Button and Check-Box Operations are easy to perform and when possible the powerful ID attribute should be used. However selection and de-selection are not the only things you want with these controls. You might want to check that if the Check Box is already checked or if the Radio Button is selected by default or anything. Check Boxes and Radio Button operate in exactly the same way and you can perform below mentioned operations on either of them.

Below is an example for locating and clicking a radio button

- `WebElement radBtn = driver.findElement(By.id("kevsCV"));`
- `radBtn.click();`

Dropdowns and Multi-Select

Just like the Check-Box & Radio Buttons already discussed, DropDown & Multiple Select Operations also work together and in almost the same way.

These are just a few of the commands available in the Selenium tool set, there are a lot more and there are plenty of books available which are dedicated to this subject. Some of these are listed in appendix 1. For the purpose of this book we now move on to the next subject, Continuous integration and deployment.

Continuous Integration and Deployment

Automation is a good aid when trying to achieve Continuous Integration and Deployment and as part of the overall scheme this practice can vastly improve the quality of code and the speed of development. Continuous integration (CI) is the practice of regularly integrating and testing your solution to incorporate changes made to its definition. These changes can include updating the source code, changing database schema or simply updating a text based configuration file.

For the best results when one or more changes are checked into your configuration management system, the solution should then be rebuilt and recompiled, retested and finally any code or schema analysis performed on it. If it is impractical to do this every time a change is saved you should strive to do so at least once if not several times a day, that is the essence of continuous integration.

Moving this process one step forward continuous deployment (CD) enhances CI by automatically deploying successful builds. For example, when the build is successful on a programmer's workstation, then the team may automatically deploy their changes to the project integration environment, which would invoke the CI system there. A successful integration in that environment could trigger an automatic deployment into another environment and so on.

On a developer's workstation, the integration job could run by example at specific times, perhaps once an hour, or better every time that they check in something that is part of the build. The whole process of continuously integrating a developer's code with the rest of a team's code and then running automated test regressions in an integration environment is the essence of CI.



This is a critical part of agile to ensure integration is done right. CI ensures high-quality working software at all times, and CD ensures that the software is running in the right place. When used with Selenium or another good automation tool good testing results can also soon be achieved.

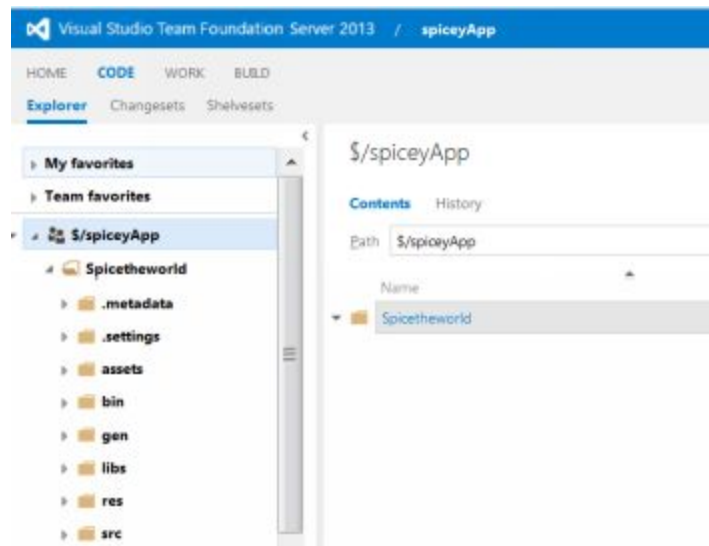
The longer a branch of code remains checked out on a programmer's computer, the greater the risk of multiple integration conflicts and failures when the branch is finally reintegrated into the main trunk. When programmers submit code to the repository they should firstly update their code to reflect the changes in the repository since they took their local copy. The more changes the repository contains, the more work programmer's must do before submitting their own changes.

There is a risk that eventually the repository may become so different from the programmer's local copy that the team can enter what is sometimes referred to as "merge hell", or "integration hell". This is where the time it takes to integrate exceeds the time it took to make their original changes and conflicts are everywhere. This is where continuous integration becomes a critical practice. It involves integrating early and often, so as to avoid the pitfalls of "integration hell".

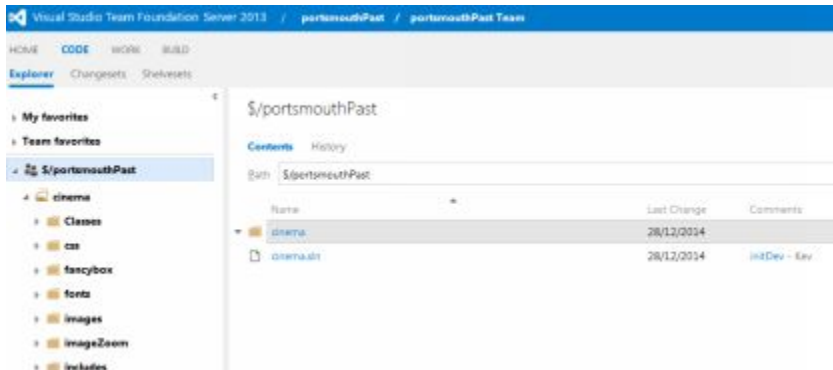
Regular integration requires a reliable repository where the main trunk of code and any development branches are stored. Off site backups of this code are also essential, should the unthinkable ever happen and the source server should go down or go missing at least code would still exist and a few hours work may be lost rather than a few years.

There are many options for repository management on the market today.

Personally I prefer the built in repository options of Team Foundation Server. This multi functioned and well designed application allows for safe storage of both Visual Studio code and also Eclipse code. Shown below is the repository for an Android application which is designed in Eclipse and all changes are stored within Team Foundation Server. Below is an example screenshot of this package. This example shows the source code structure for an android application written in Eclipse.



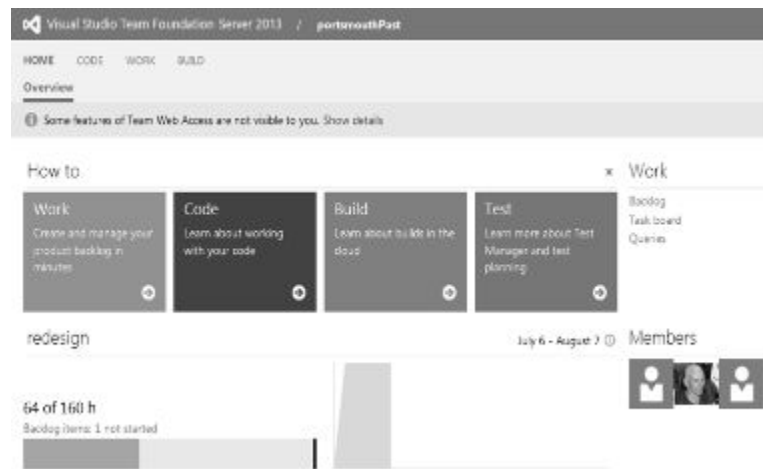
By contrast the next screenshot is from the same Team Foundation Server but this time is for a visual studio ASPX/C# application.



In the next chapter we will look further at Team Foundation Server and how it can help the agile tester in their quest to become an expert tester.

Team Foundation Server

Team Foundation Server (which is commonly abbreviated to TFS) is a Microsoft product which provides integrated project and source code management. TFS also provides reporting, requirements management, project management (for both agile software development and waterfall teams), automated builds, testing and release management capabilities. In fact TFS has the ability to cover the entire application lifecycle.



TFS is a complex and complete yet intuitive suite which can be used as a back end to numerous integrated development environments but is tailored for the Microsoft Visual Studio and Eclipse development platforms.

To help with agile teams of various sizes TFS supports multi teams and multi projects with ease and is very well suited to the agile development framework. TFS and other tools with similar features allow the entire team to keep up to speed on the current sprint progress and everyone including the testers can access the progress screens, burn down charts and any attached artifact such as test cases.

As with any agile project the development phase is conceived, agreed and recorded within TFS, an example being shown below. In this example phase the software company acmexyz and getting ready to design stage two of their flagship product acmexyzsoft. This data comprises a chosen name for the project and development date span and the application.

EDIT ITERATION

Version 2 Upgrades

Iteration name: Version 2 upgrades

Start date: 8/17/2015

End date: 12/31/2015

Location: FMManager

Save and close Cancel

Next this agreed development phase is broken down into monthly sprints as shown below. In this example each sprint is given a descriptive name which gives an idea on what it is hoped will be achieved during these sprints. Typically these sprints are one month in length and each will focus on a particular stage of development.

<input type="checkbox"/>	Version 2 upgrades	8/17/2015	12/31/2015
<input checked="" type="checkbox"/>	data integration sprint	7/31/2015	8/31/2015
<input checked="" type="checkbox"/>	Reporting Sprint	9/1/2015	9/30/2015
<input checked="" type="checkbox"/>	Exporting Sprint	10/1/2015	10/30/2015
<input checked="" type="checkbox"/>	db2 plugin sprint	11/2/2015	12/1/2015
<input checked="" type="checkbox"/>	Review and bug fix sprint	12/2/2015	12/31/2015

Initially each of these sprints will be empty and no backlog items will be visible.

Current

data integration sprint

Future

Reporting Sprint

Exporting Sprint

db2 plugin sprint

Review and bug fix sprint

Create query Column options

Title	State	Assigned To	Remaining Work
-------	-------	-------------	----------------

As with any agile project one of the first tasks is to decide who will be working on the project. The size and complexity of the team will vary depending on the size and complexity of the actual project, shown below is a small example.

In this example there are 3 people, each of whom work eight hours a day. In this small example we have the product owner, a programmer and a tester. This important data allows the team to manage resources and ensure that enough capacity exists to complete the agreed tasks within a sprint.

Backlog Board **Capacity**

Team Member	Capacity Per Day	Activity	Days Off
Administrator	8	Design	0 days +
Kev	8	Development	0 days +
Kevin Jr	8	Testing	0 days +
Team Days Off			0 days +

When the team makeup is decided it is then possible to add backlog items to the sprint and assign these to team members, below is a typical example of how this will look in TFS.

Product Backlog Item 383: Data integration change

Steps: Add...

Data integration change

Insights: PMManager/Version 2 upgrade/data integration update

STATUS	DETAILS
Assigned To: Administrator	State: Off
Code: Approved	Business Value: 18
Reason: Approved by the Product Owner	Area: PMManager

DESCRIPTION	STORY/REQUIREMENTS	TEST CASES	TASKS	ACCEPTANCE CRITERIA	HISTORY	LINKS	ATTACHMENTS

Save Save and Close Cancel

Once all the agreed backlog items have been added to the correct sprint and assigned they will then become visible in the sprint backlog.

Backlog Board Capacity

Create query Column options

Title	State	Assigned To	Remaining Work
<ul style="list-style-type: none"> Data integration change <ul style="list-style-type: none"> Design data integration changes Code data integration changes Test data integration changes 	Approved	Administrator	64
	To Do	Administrator	16
	To Do	Kev	32
	To Do	Kevin Jr	16

Each of the backlog items can be edited and external artefacts such as test cases can also be attached. This means the entire team can have access to any document that is relevant to the sprint. Another important piece of data on these cards is the Remaining Work value. The actual value that is used here can vary from company to company but in many cases this will relate to estimated hours remaining. These are usually updated daily just after the daily scrum. This is high value data which will help provide up to date burn down charts which in turn keep all interested parties up to speed on sprint progress.

Task 387: Test data integration changes

Assignee: Kevin Jr
 Priority: To Do
 Status: To Do
 Remaining Work: 16

DESCRIPTION
 Full description of testing requirements go here

HISTORY
 History notes here
 Images and files can also be inserted and files can be attached

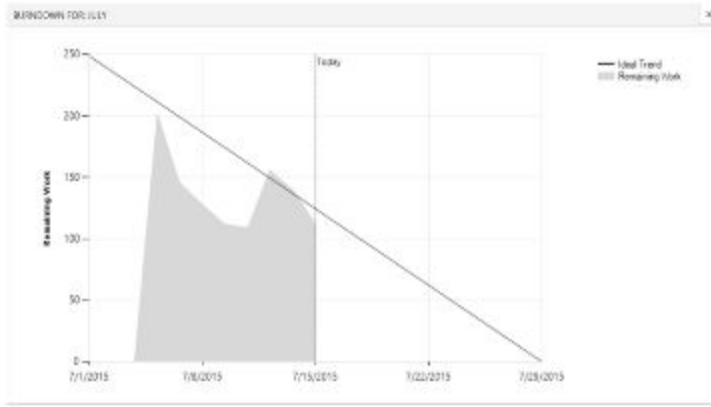
Below is a good example of a backlog with everyone assigned their workload and up to date Remaining Work on each backlog item.

Backlog			
Title	State	Assigned To	Remaining Work
Data integration change	Committed	Administrator	64
Design data integration changes	To Do	Administrator	16
Code data integration changes	To Do	Kev	32
Test data integration changes	To Do	Kevin Jr	16

The same data is also available in the task board view as shown below. In this view backlog items can be dragged from one column to another. This will have the effect of changing the status of the backlog item which will be automatically updated. If the item is dragged to the DONE column then the remaining work value will also be set to zero.



Finally all of this updated and valid data is visible in the monthly burn down chart, an example of which is visible below. This chart and any other data is only as good as the information that is input into the system. For agile to work correctly everyone has to be 100% behind the processes and everyone has to ensure the data is kept up to date and is as accurate as it can possibly be.



Conclusion

So here we are then, over two hundred pages into this nice little book and you are still here, so hopefully you have enjoyed this publication and found the contents interesting, useful and valid to your needs.

If you become a full time agile tester in the future or you already there then I hope you will enjoy your career. Testing can at times become very monotonous and as deadlines loom it can also become very stressful. As I have already pointed out not everyone is suited for software testing. Others can do it but only for a short period of time. People with the right mindset and temperament for software testing as a career are a scarce but valuable commodity. If you are one of these people then good for you, grow your skills and enhance your career, you will always be in demand.

Being a software tester in an agile team can be a more rewarding experience than working in a traditional organisational setup. Being part of a closely knit team who are together in regular meetings helps build knowledge and confidence in the product. This in turn helps make the testing process a more rewarding and less stressful experience. As your knowledge and skill set grow so will your confidence in becoming a more pro-active team member. This may well annoy some old school programmers who may still think you should be seen and not heard but this is, to put it bluntly, their problem.

So if you stay in software testing for a long period of time or you eventually move on to other things such as project management I hope you enjoy the agile experience and I hope you have enjoyed reading this book. So to conclude in the next chapter we will look at some of the more common agile myths.

The Agile Myths

Agile has been around for a few years now and is now considered a tried, trusted and established development approach. However for companies that has not yet adopted agile it still represents a new way of life and as a result the way agile works can still cause unrest among those who have never used it. The prospect of working in iterations instead of with a linear approach can be unsettling to managers and programmers who are deciding whether to make the leap to agile. They fear that focused efforts will be compromised and that control over projects and development teams will be sacrificed. In reality of course nothing is further from the truth, so let's get rid of some of the myths that surround agile.

Agile Is a Fad

No, agile is not a new fad. The Agile Manifesto was published in 2001; the Scrum Pattern language was presented in 1995 during the Object Oriented Programming, Systems, and Languages (OOPSLA) conference and there are some who trace agile's roots back further still.

So the agile approach to project management is far from a fad. Agile has been in use for many decades even though it was only recently formalized with the Agile Manifesto and its associated principles.

Agile exists because it works. Compared with traditional project management approaches, agile is better at producing successful projects.

Agile Means No Documentation

While some people believe that being agile means one doesn't need any documentation, that's hardly the truth; you can have as much documentation as you like in agile. Documentation is just another deliverable; if it brings you value then schedule it and produce it like anything else. Agile teams keep documentation as lightweight as possible, but they do document their solutions as they go. They follow strategies, such as documenting continuously and writing executable specifications.

Agile Is a Silver Bullet

Proponents of Agile will sometimes claim that moving to Agile will "fix all your problems." That isn't the case. Agile isn't needed for every team in every situation. It isn't a cure-all. Agile is a superb solution for projects that are in development or undergoing radical changes.

It's important to stress that the Agile Manifesto is a set of values and principles that define a core attitude for software development.

These values point to collaboration, rapid feedback loops and quality. If your project has a stable customer base and isn't undergoing a lot of change in the code, you may not need to use agile for that particular project. But for projects that are creating a new product or major updates then agile really is the best way to go.

It's enough for our Development

Team to Be Agile

This is so far from the truth that travelling to Pluto would be closer. For agile to work properly within your company, all of the teams have to buy in fully. So if your development team is agile, but your testing team is still a group of admin staff pulled in when needed, you will not get your best results, in fact things could get worse. Your agile delivery process is only going to be as effective as your slowest group. To make agile succeed at its greatest potential, make each piece of the chain as efficient as possible.

Agile Won't Work at My Company

When I hear this statement I always think why? What is your problem? I never think why can the company not adopt agile I always think why does this person say it will not work.

For many companies, the biggest challenge they face when considering changing to agile is the cultural change involved when implementing the changes. For some people this is a major challenge and they would rather it did not happen. Agile has explicit methods of frequent feedback and loops, which means that programmers, testers and managers may feel more exposed to scrutiny. But that doesn't mean that agile won't work at your company, it may mean some people should no longer be working at your company though. Remember folk's agile is a team approach; roles are cross-functional and shared. Programmers become testers and more frequent delivery creates more exposure and personal accountability.

There's Only One Way to Do Agile

This is so very wrong. The Agile Manifesto consists of four values and 12 principles; it doesn't document implementation details. There are many interpretations of Agile, including Scrum, XP, Kanban and Feature-Driven Development, to name a few. Each style has benefits, as well as weaknesses, and you must evaluate your own specific situation to determine which interpretation is the best match. As long as you're adhering to the Agile Manifesto's values and principles, you should be considered Agile.

Agile Isn't Disciplined

Sometimes agile can seem chaotic because it's a very collaborative process. Agile is a departure from the rigid assembly-line process. The iterative approach requires rapid response times and flexibility from the team. In fact, agile demands greater discipline than what's typical of traditional teams. Agile requires teams to reduce the feedback cycle on many activities, incrementally deliver a consumable solution, work closely with stakeholders throughout the life cycle, and adopt individual practices which require discipline in their own right.

Agile Is Only Effective for Collocated Teams

In an ideal world the whole team would be located within proximity of one another. However in this day and age, most development teams are distributed around the globe. While this can be a challenge with the right tools and planning it can still work. Just remember, to succeed, you need to adopt practices and tooling that build team cohesion. Careful planning is the key here.

Agile Means “We Don’t Plan”

With agile’s reliance on collaboration instead of big documents, it may well seem like no real planning occurs. But in reality, the planning is incremental, well understood and evolutionary.

Agile Is Unsuitable for

Regulated Environments

Regulated environments are those that are subject to some regulatory mandates, such as defense organizations, medical suppliers, financial companies and governmental departments to name but a few. These organizations are audited from time to time for compliance with regulations. With agile, these organizations can feel confident when they endure these audits. They benefit from faster delivery of data and higher quality of their output.

Glossary

A

Acceptance Test

Acceptance tests are tests that define the business value each story must deliver. They may verify functional requirements or non-functional requirements such as performance or reliability. Although they are used to help guide development, it is at a higher level than the unit-level tests used for code design in test-driven development. Acceptance test is a broad term that may include both business-facing and technology-facing tests.

Agile Operating Model

The holistic and simple definition of what an organisation, program, project or team mean when they use the term “Agile”. This could range from a single agile framework or an integrated implementation of many frameworks, the latter being much more likely. Agile operating models align to the “Agile Manifesto”.

Agile Persona

Someone (this could be a single person or a group) who will interact with the system being built, also known as a “user”.

Agile Project Management

The style of project management used to support agile software development. Scrum is the most widely used agile project management practice. XP practices also include practices that support agile project management.

Application Programming Interface (API)

APIs enable other software to invoke some piece of functionality. The API may consist of functions, procedures, or classes that support requests made by other programs.

B

Backlog

An ordered list of requirements/stories that the customer wants.

Baseline Plan

The plan that defines the start point from which an evolving product starts, normally high level.

Behavior Driven Development

Behavior driven development (or BDD) is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders.

Best Practice

The learned best approach for something at a particular point in time, best practice evolves over time.

Bugs

A software bug is a problem causing a program to crash or produce invalid output. It is caused by insufficient or erroneous logic and can be an error, mistake, defect or fault.

Build

A build is the process of converting source code into a deployable artifact that can be installed to run the application. The term “build” also refers to the deployable artifact.

Burn-down Chart

A chart showing the evolution of remaining effort against time. Burn-down charts are an optional implementation within Scrum to make progress transparent.

Burn-up Chart

A chart showing the evolution of an increase in a measure against time. Burn-up charts are an optional implementation within Scrum to make progress transparent.

Business

The customers, stakeholders and users involved with the product.

C

Command and Control

This is a style of management where the manager commands the team to do something and then controls them to do it. This style of management is the opposite of Agile self-organising teams and is counter to everything agile.

Commitment Plan

Typically a detailed forecast for a short period of time they are also known as iteration/ sprint (or time-box) plans.

Component

A component is a larger part of the overall system that may be separately deployable. For example, on the Windows platform, dynamic linked libraries (DLLs) are used as components, Java Archives (JAR files) are components on the Java platform, and a service-oriented architecture (SOA) uses Web Services as components.

Component Test

A component test verifies a component's behavior. Component tests help with component design by testing interactions between objects.

Conditions of satisfaction

Conditions of satisfaction, also called satisfaction conditions or conditions of business satisfaction, are key assumptions and decisions made by the customer team to define the desired behaviour of the code delivered for a given story.

Conditions of satisfaction are criteria by which the outcome of a story can be measured. They evolve during conversations with the customer about high-level acceptance criteria for each story. Discussing conditions of satisfaction helps identify risky assumptions and increases the team's confidence in writing and correctly estimating all the tasks to complete the story.

Context-driven testing

Context-driven testing follows seven principles, the first being that the value of any practice depends on its context. Every new project and every new application may require different ways of approaching a project.

Cost of Delay

The cost of delaying an investment decision, tend to grow over time.

Customer

The person/ people who own the product (e.g. usually known as a 'Product Owners' or 'Business Ambassadors' in certain frameworks).

Customer Team

The customer team identifies and prioritizes the features needed by the business. In Scrum, these features become epics or themes, which are further broken into stories and comprise the product backlog.

Customer teams include all stakeholders outside of the development team, such as business experts, subject-matter experts, and end users.

Testers and developers work closely with the customer team to specify examples of desired behaviour for each story and turn those examples into tests to guide development.

Customer Test

A customer test verifies the behaviour of a slice or piece of functionality that is visible to the customer and related directly back to a story or feature. The terms “business-facing test” and “customer-facing test” refer to the same type of test as customer test.

D

Daily Scrum/Standup

The daily time-boxed event of a maximum of 15 minutes sometimes less. The scrum is for the Development Team to re-plan the next day of development work during a Sprint. Updates are reflected in the Sprint Backlog.

Decomposition

The process of breaking user stories down into a) smaller, more executable user stories or b) tasks. Likewise, epics may be decomposed into user stories, and tasks may be decomposed into more fine-grained tasks. Decomposition is usually performed during backlog grooming and iteration planning, and is an important precursor to story sizing (estimation). Decomposition may also occur throughout the development process. In the typical product backlog, user stories will become finer grained as they near implementation, and are larger and less detailed the further down the queue they reside.

Definition of Done

Normally a list of working features that defines the complete product that must be delivered; must be standard across the team.

Definition of Ready

Normally a list that defines when artefacts within the delivery process are ready (e.g. story ready to go into iteration/ sprint). This can vary from organisation to organisation.

Development Team

Develop and test the product. The team is self-organized: There is no team lead, so the team makes the decisions. The team is also cross-functional.

DevOps

Viewing the development and operation of a software system as one continuous delivery value process.

E

Emergence

The process of the coming into existence or prominence of new facts or new knowledge of a fact, or knowledge of a fact becoming visible unexpectedly.

Empiricism

Process control type in which only the past is accepted as certain and in which decisions are based on observation, experience and experimentation. Empiricism has three pillars: transparency, inspection and adaptation.

Environment

This is the combination of all factors within an organisation, project, team etc. that drives suitability of a delivery or governance framework. In a dynamic environment, where things change all the time, an agile framework would be suitable.

Epic

An epic is a piece of functionality, or feature, described by the customer and is an item on the product backlog. An epic is broken up into related stories that are then sized and estimated. Some teams use the term “theme” instead of epic.

Estimation

The process of agreeing on a size measurement for the stories, as well as the tasks required to implement those stories, in a product backlog.

Exploratory testing

Exploratory testing is interactive testing that combines test design with test execution and focuses on learning about the application.

Extreme Programming (XP)

An Agile software development methodology that emphasizes customer involvement, transparency, testing and frequent delivery of working software.

The Extreme Programming cannon includes a Customer Bill of Rights and a Developer Bill of Rights, and lists its core values as communication, simplicity, feedback, courage and respect. XP is a developer-centric methodology, and unlike Scrum, it prescribes specific coding practices like Pair Programming, in which two developers work side by side at a single machine, automated unit testing, and frequent integration. Another key practice in XP is refactoring or the continual improvement of design over many iterations.

F

Facilitated Workshops

Where groups of people come together in a forum to achieve a stated objective, the achievement of which is facilitated by a workshop facilitator. Many activities (such as planning) within Agile are delivered within facilitated workshops.

Feature

A feature of the system that the customer wants. These are normally described as a story and ordered within a backlog.

Feature creep

Feature creep occurs when software becomes complicated and difficult to use as a result of too many features.

Functional tests

Functional tests verify the systems expected behaviour given a set of inputs and/or actions.

Forecast

The selection of items from the Product Backlog that a Development Team considers feasible for implementation in a Sprint.

Impediment

In Scrum, any obstacle preventing a developer or team from completing work. One of the three focusing questions each member of a Scrum team answers during the daily Stand Up Meeting is: What impediments stand in your way?

Increment

A piece of working software that adds to previously created Increments, where the sum of all Increments -as a whole - form a product.

Iteration/ Sprint

An iteration is a short development cycle, generally from one to four weeks, at the end of which production-ready code can potentially be delivered. Several iterations, each one the same length, may be needed to deliver an entire theme or epic. Some teams actually release the code to production each iteration, but even if the code isn't released, it is ready for release.

Iteration/ Sprint Goal

The goal that the entire team commit to in relation to an iteration/ sprint plan.

Iteration/ Sprint Plan

The forecast of what will be delivered within a short focused 'sprint' by the team.

K**Kanban**

Kanban is a management approach that is sometimes used in agile projects. The general objective is to visualize and optimize the flow of work within a value-added chain.

Knowledge Based Work

Work where the main capital is knowledge, such as doctors, engineers and information technology workers.

L

Lean

Lean software development is a translation of Lean manufacturing and Lean IT principles and practices to the software development domain. Adapted from the Toyota Production System and is a set of techniques and principles for delivering more values with the same or less resources by eliminating waste across organizations and business processes

N

Noise

Anything that interrupts the team within an iteration/ sprint, noise causes significant disturbance within a team and causes lack of focus on delivery.

P

Pair Programming

Pair programming is an agile software development technique in which two programmers work together at one workstation.

One types in code while the other reviews each line of code as it is typed in. The person typing is called the driver. The person reviewing the code is called the observer (or navigator). The two programmers switch roles frequently.

Planning Poker

Also called Scrum poker, is a consensus-based technique for estimating, mostly used to estimate effort or relative size of tasks in software development.

Product Backlog

The product owner manages a prioritized list of planned product items (called the product backlog). The product backlog evolves from sprint to sprint (called backlog refinement).

Product Backlog Refinement

This is the activity within a Sprint through which the Product Owner and the Development team add granularity to the Product Backlog.

Product Increment

Each sprint results in a potentially releasable/shippable product (called an increment).

Product Owner

Represents the customer, and generates, maintains, and prioritizes the product backlog. This person is not the team lead.

R

Ready

A shared understanding by the Product Owner and the Development Team regarding the preferred level of description of Product Backlog items introduced at Sprint Planning.

Requirements

These are more correctly described as ‘stories’ within most Agile frameworks.

Retrospective

This is a team meeting that should happen at the end of every complete development iteration. The purpose of this meeting is to review lessons learned and to discuss how the team can be more efficient in the future. It is based on the principles of applying the learning from the previous sprint to the upcoming sprint.

S

Scrum

The framework to support teams in complex product development. Scrum consists of Scrum Teams and their associated roles, events, artifacts, and rules, as defined in the Scrum Guide.

Scrum Board

A physical board to visualize information for and by the Scrum Team; often used to manage Sprint Backlog. Scrum boards are an optional implementation within Scrum to make information visible.

Scrum Guide

The definition of Scrum, written and provided by Ken Schwaber and Jeff Sutherland, co-creators of Scrum. This definition consists of Scrum’s roles, events, artifacts, and the rules that bind them together.

Scrum Master

Ensures that Scrum practices and rules are implemented and followed, and resolves any violations, resource issues, or other impediments that could prevent the team from following the practices and rules. This person is not the team leader, but a coach.

Scrum Team

A self-organizing team consisting of a Product Owner, Development Team and Scrum Master.

Scrum Values

A set of fundamental values and qualities underpinning the Scrum framework; commitment, focus, openness, respect and courage.

Self-organization

The management principle that teams autonomously organize their work. Self-organization happens within boundaries and against given goals. Teams choose how best to accomplish their work, rather than being directed by others outside the team.

Source Control System

Part of software configuration management, manages the central repository of code versions, etc.

Sprint

Scrum divides a project into iterations (called sprints) of fixed length (usually two to four weeks).

Sprint Backlog

At the start of each sprint, the Scrum team selects a set of highest priority items (called the sprint backlog) from the product backlog. Since the Scrum team, not the product owner, selects the items to be realized within the sprint, the selection is referred to as being on the pull principle rather than the push principle.

Sprint Goal

A short expression of the purpose of a Sprint, often a business problem that is addressed. Functionality might be adjusted during the Sprint in order to achieve the Sprint Goal.

Sprint Planning

Time-boxed event of 1 day, or less, to start a Sprint. It serves for the Scrum Team to inspect the work from the Product Backlog that's most valuable to be done next and design that work into Sprint backlog.

Sprint Retrospective

Time-boxed event of 3 hours, or less, to end a Sprint. It serves for the Scrum Team to inspect the past Sprint and plan for improvements to be enacted during the next Sprint.

Sprint Review

A time-boxed event of 4 hours for a monthly sprint, less if the sprint was shorter. The review is designed to conclude the development work of a Sprint. This meeting the Scrum Team and the stakeholders to

inspect the Increment of product resulting from the Sprint, and assess the impact of the work performed on overall progress and update the Product backlog in order to maximize the value of the next period.

Stakeholder

Any person or group who can help or hinder the team. They are external to the Scrum Team and they have a specific interest in and knowledge of a product that is required for incremental discovery. The Stakeholder(s) are represented by the Product Owner and they should actively engage with the Scrum Team at Sprint Review.

Story

A requirement or feature that may be delivered at some point; a story is a token to remind everyone that something may need to be delivered. Stories reside on the backlog.

T

Time-box

A fixed period of time within which delivery is made and stories are prioritised within a time-box. With Agile projects, releases and iterations/ sprints are all time-boxes.

Transparency

The development team reports and updates sprint status on a daily basis at a meeting called the daily scrum. This makes the content and progress of the current sprint, including test results, visible to the team, management, and all interested parties. For example, the development team can show sprint status on a whiteboard.

U

User

People who will use the product, known as ‘Agile personas’ within Agile.

V

Velocity

An optional, but often used, indication of the average amount of Product Backlog turned into an Increment of product during a Sprint by a Scrum Team, tracked by the Development Team for use within the Scrum Team.

W

Working Software

Software that works, it has all the elements associated with the ‘Definition of Done’ and is ready to deploy into an environment which should be the live production environment.

Appendix 1

Further Reading

Eclipse

- Eclipse: A Java Developer's Guide Paperback by Steve Holzner
- Eclipse IDE: Eclipse IDE based on Eclipse 4.2 and 4.3

Selenium

- Selenium Testing Tools Cookbook by Unmesh Gundecha
- Mastering Selenium WebDriver by Mark Collin
- Selenium 2 Testing Tools: Beginners Guide by David Burns

Team Foundation Server

- Professional Team Foundation Server 2013 by Steven St. Jean and Damian Brady
- Microsoft Team Foundation Server 2015 Cookbook by Tarun Arora

Testing

- Foundations of Software Testing ISTQB Certification by Dorothy Graham and Erik Van Veenendaal
- Testing in Scrum: A Guide for Software Quality Assurance in the Agile World by Tilo Linz

Appendix 2

Download URL's

Visual Studio Community Edition

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>

Selenium Chrome Driver

<https://sites.google.com/a/chromium.org/chromedriver/downloads>

Selenium

<http://www.seleniumhq.org/download/>

Eclipse

<http://www.eclipse.org/downloads/>

Java JDK

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Junit

<http://junit.org/>