

Agile Testing

Janakirama Raju Penmetsa

Abstract In recent times, software development has to be flexible and dynamic due to ever-changing customer needs and high competitive pressure. This competitive pressure increases the importance of Agile methods in software development and testing practices. Traditional testing methods treat development and testing as a two-team two-step process. The process discovers bugs in software at later stage of development. Further, the process frequently leads to an internal division between teams. Agile testing combines test and development teams around the principles of collaboration, transparency, flexibility, and retrospection. This testing enables the organization to be nimble about uncertain priorities and requirements. It helps to achieve higher quality in software products. This chapter with a brief introduction on Agile-based software engineering deals with Agile-based testing. Agile testing focuses on test-first approaches, continuous integration (CI), and build–test–release engineering practices. The chapter also explains advantages and disadvantages of Agile testing practices. The process is explained with an example.

Keywords Agile testing • Testing in parallel • Team collaboration • Continuous improvement • Test engineering

1 Introduction

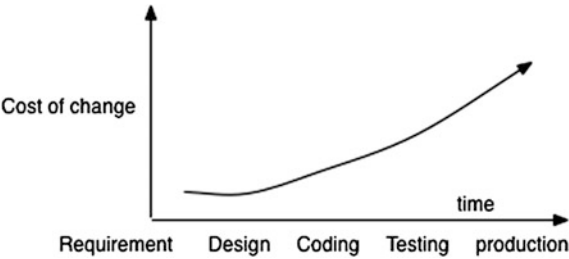
In traditional testing, a two-team, two-step process is followed where development team builds software to the state of perfection. Thereafter, testing team strives hard to find bugs in software and sends test report back to the development team. Only minimal collaboration happens between developers and testers through reviewing test case documents, design documents, requirement documents, etc. Sometimes, the reviews are ignored to accommodate change and critical timelines. This traditional process requires more time and money and often leads to an illusory divide

J.R. Penmetsa (✉)

International Game Technology (IGT), Seattle, WA, USA

e-mail: Janaki.Penmetsa@igt.com

Fig. 1 Cost of change curve



among developers and testers. The cost of change to fix a bug increases exponentially based on the time delay between the introduction of a bug and its finding. This phenomenon is described in Fig. 1.

Testing as close to development as possible is the key to Agile testing. Critics often call traditional methods as heavily regulated, regimented, and micro-managed. New Agile and light-weight methods evolved in mid-1990s to avoid shortcomings in traditional methods. Once Agile manifesto defined in 2001, its practice has come to prominence. The common aspect of all Agile methodologies is delivering software in iterations, keeping user priority in mind. Changes in requirements may change iteration priorities, but it is easy to reschedule the iterations as situation demands. The majority of successful Agile teams used the best development practices to come up with their own flavor of agility. Scrum and XP are two such popular Agile methods.

Agile testing means testing within the context of an Agile workflow. An Agile team is usually a cross-functional team, and Agile testing involves all of them in the process. Tester’s particular expertise is used to embed quality into the deliverables at every iteration (Fig. 2).

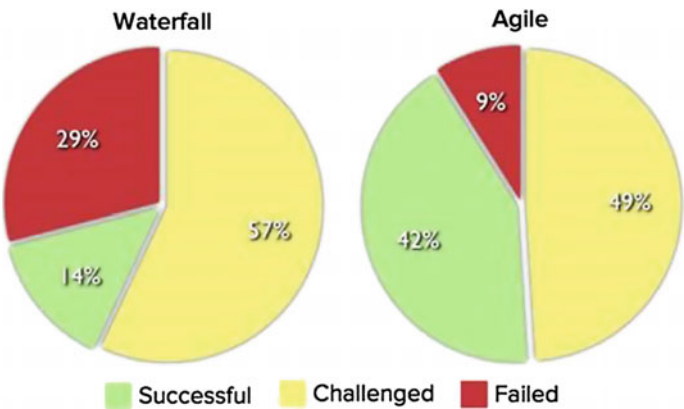


Fig. 2 As per CHAOS Manifesto 2012, a project is “Successful” means “Project is on budget and on time as initially specified”. A project is “Challenged” means “Completed but over budget, over time”. A project is “Failed” means “Project canceled at some point”. *Source* The CHAOS manifesto, The Standish group, 2012

According to CHAOS Manifesto, Agile methods are about three times more successful than traditional methods as shown in Fig. 2 [1]. Keeping in this view, Agile testing has gained acceptance among practitioners.

In this chapter, Agile testing process is described in detail using a simple illustration. In Sect. 2, a short description on traditional testing process is described. Sections 3 and 4, respectively, present discussions on Agile-based software engineering and Agile-based testing. Section 5 presents an example explaining Agile testing process. Next section briefs on engineering process in Agile testing. Section 7 presents a brief analysis highlighting advantages and disadvantages in Agile testing. The chapter ends with a concluding remark in the next section.

2 Traditional Testing Practices

Traditional testing practices revolve around the testing center of excellence model [1]. A group of seasoned testers form as a separate testing team, who has the best customer focus, motivation, and intuition. Development team handles code delivery and fixing bugs.

Typically, after developers deliver code, testing team tries to find as many bugs as possible. Developers keep poor attention to quality while developing code. Consequences are harder and costly to fix.

Traditional testing teams keep costs low by using tools to document test cases and bugs and sometimes through labor outsourcing. However, this does not reduce systemic issues and shift costs back upstream into the development cycle. It results in higher levels of scrap and rework.

Preparing detailed test cases appear to help and optimize testing, but exacerbate the problem whenever requirements change. Change in requirements is almost unavoidable. Extensive efforts of testing activities slow down delivery. Even throwing a phalanx of testers is not very efficient. In reality, daily builds, functional and nonfunctional testing cadence overhead nullify any gains achieved due to specialized testing teams and practices.

Usually, testing is pushed to end of projects and gets squeezed. Unfortunately, when projects fall behind schedule, teams compress and sacrifice testing time to make up for delays in other processes. Thus, quality is always compromised.

Last-minute discovery of defects results in waste and high rates of rework. Longer it takes to provide feedback to developers, longer it takes to fix them. Developers take additional time to re-engage with the context of the code as they move on to the new project and new problems. The problem is worse if that last-minute bug is an architecture or design issue. A simple misunderstood fundamental requirement can cause havoc to timelines if discovered last minute.

Teams build up too much technical debt (also known as code debt or design debt). Technical debt is a metaphor referring to the possible future work of any system architecture, design, and development within a codebase [2]. Solution to these problems is to push testing to earlier phases of the development cycle.

In the mid-1990s, to overcome some of the above deficiencies, a collection of lightweight software development methods evolved. After Agile Manifesto published in 2001, these methods are referred to as Agile methods. Often loosely abbreviated as Agile, with a capital “A,” next section describes Agile software practices in more detail.

3 Agile-based Software Engineering

The Agile manifesto, principles, and values of Agile software engineering are formed revealing better ways of producing software.

3.1 *Agile Manifesto [3]*

Agile Manifesto uncovers better ways of developing software by giving value to:

- **Individuals and interactions** over processes and tools,
- **Working software** over comprehensive documentation,
- **Customer collaboration** over contract negotiation,
- **Responding to change** over following a plan.

3.2 *Agile Processes*

Agile-based software engineering incorporates the principles mentioned just before. Several Agile processes such as Scrum, Kanban, Scrumban, Extreme Programming (XP), and Lean methods are in practice. Each one follows the Agile principles. The majority of successful Agile teams tune the processes with their own particular flavor of agility. Among those, this chapter describes two common processes Extreme Programming and Scrum.

3.3 *Extreme Programming (XP)*

XP is a software development methodology emphasizes teamwork and advocates frequent “releases” in short development cycles.

In short, take every observed effective team practice and push it to the extreme level. Good code review process pushed to the extreme of pair programming for instant feedback. Good software design is pushed to the extreme of relentless refactoring. Simplicity is pushed to the extreme of the simplest piece of software

that could possibly work. Testing is pushed to the extreme of test-driven development and continuous integration (CI).

Managers, customers, developers, and testers are all equal participants in a collaborative team. XP enables teams to be highly productive by implementing a simple, yet effective self-organizing team environment [4]. XP improves a software deliverable in five essential ways; simplicity, communication, respect, courage, and feedback. Extreme Programmers regularly communicate with their fellow developers and customers. They keep their design clean and simple.

Testers start testing on day one and developers get feedback immediately. Team delivers the system as soon as possible to the customers and implements suggested changes. Every small success increases their respect for the individual contributions of team members. With this foundation, Extreme Programmers can courageously respond to changing technology and requirements [4].

3.4 Scrum

Scrum is a management framework for incremental or iterative product development utilizing one or more self-organizing, cross-functional teams of about seven people each. Within the Scrum framework, teams create their own process and adapt to it [5] (Fig. 3).

A product owner produces a prioritized wish list called as the product backlog. At the beginning of the iteration, the team takes a small chunk from the top of that backlog, and it is called as sprint backlog. The team then plans on how to implement those pieces in detail. The team has about two to three weeks time to complete its work. However, the team meets every day to assess its progress called as daily Scrum. Along the way, the Scrum master responsibility is to keep the team focused on its goal. The work is potentially shippable and demonstrated to the stakeholder.



Fig. 3 Scrum framework [5]

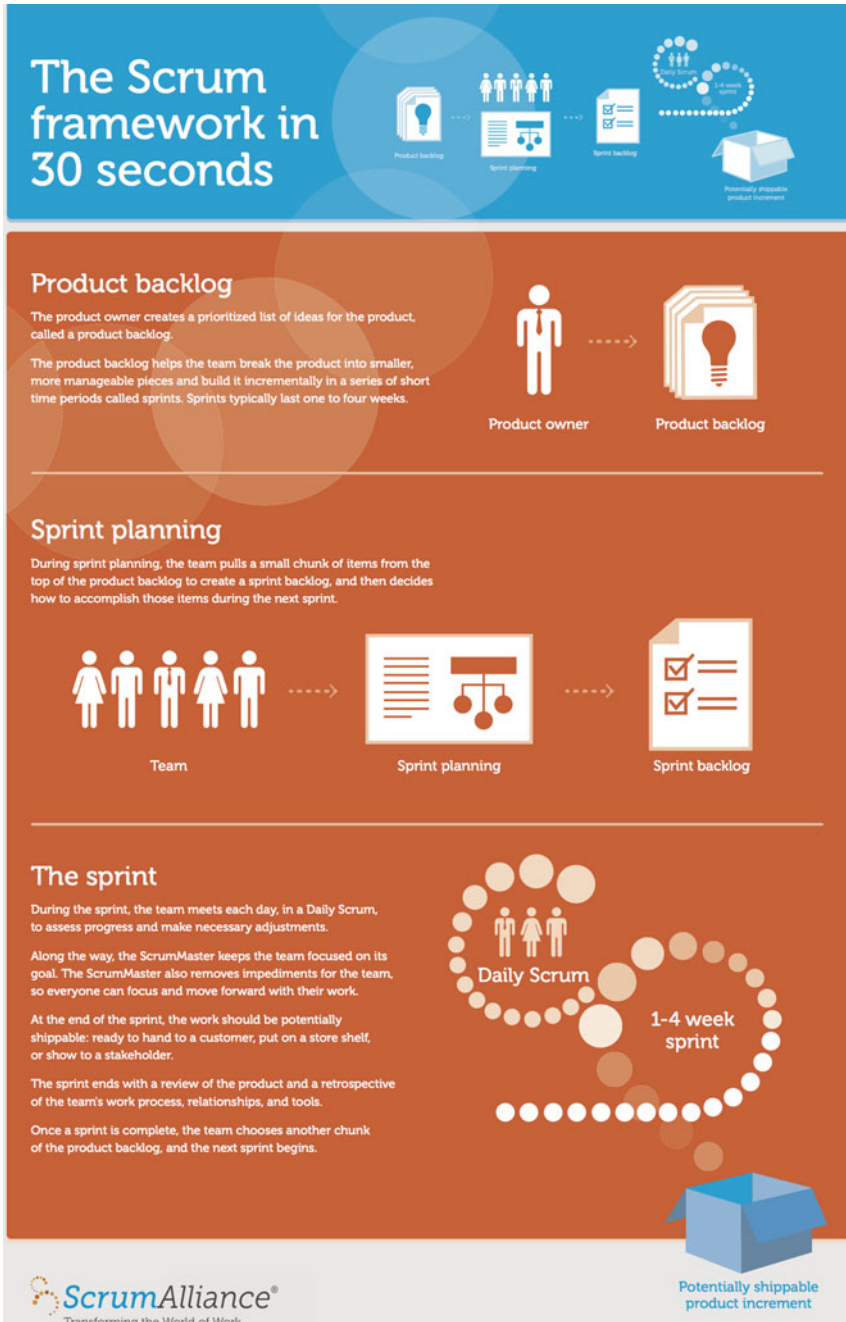


Fig. 4 Scrum process [6]

The sprint ends with a sprint retrospective and review. And the whole process repeats, with next chunk of work from the product backlog [5] (Fig. 4).

After understanding Agile processes in detail, it is time to explore Agile-based testing (at times referred as “Agile testing”) and how it helps to achieve higher quality.

4 Agile-based Testing

Agile-based testing means testing within the context of an Agile process such as Scrum or XP. Agile suggests development and testing, two essential functions of software development, proceed concurrently. *Agile testing emphasizes on positive desire to implement a solution that passes the test, confirmation of user stories, whereas traditional testing methods focus on negative desire to break the solution, falsification of given solution.*

Development and testing teams are united based on Agile principles to perform Agile testing. Illusory divide between code breakers (testers) and code creators (developers) is reduced. The necessity of both development and testing roles is respected. The testing team works to provide feedback to developers as soon as possible. This integration implies both that developers cultivate skills of testers and testers understand the logic in development.

Applying Agile concepts to the testing and QA process results in a more efficient testing process. Pair programming is encouraged to provide instant feedback, and also test-driven methodologies come handy, where tests can be written before the actual development. Lack of automation is the principle barrier to provide instant feedback to developers. End-to-end automated testing process integrated into development process achieves continuous incremental delivery of features.

In short, Agile requires teamwork and constant collaboration among software development teams and stakeholders. Quality is “baked into” the product at every phase of its development through continuous feedback. Everyone in the team holds a vision of the final product.

Working software is preferred over documentation and produced in small iterations frequently. However, every iteration delivered is in line with demands of the final system. Every iteration has to be fully integrated and carefully tested as a final production release.

In Agile testing, there is no strict adherence to requirement documents and checklists. The goal is always to do what is necessary to complete customer’s requests. Documentation is often replaced with constant collaboration through in-person meetings and automated tests. Only essential and minimal documentation is maintained. Unified self-organizing project teams work closely together to achieve a high-quality software product. The notion of separate testing team disappears entirely. In some cases, it might be necessary to do separate release candidate testing for not to find problems or bugs. It is just performed for verification, trail audit completion, and regulatory compliance.

The continuous internal dialogue between team provides better-working relationships between testers and developers. Constant collaboration with end users improves responsiveness to ever-changing project requirements.

The retrospective meeting is an integral part of Agile process and is done after every iteration for continuous improvement. This meeting is an integral part of Agile process. These meetings reflect self-organization and regular adaptation of the team.

5 Illustration

Let us consider a use case to design a small messaging service to understand the process better. This new service sends a message to the user if the user is online. Otherwise, it would persist until the user comes online. Assume Scrum process is being followed to implement this use case.

Developers and testers discuss customer requirements and identify the tests that need to be designed for acceptance criteria. A lot of questions are raised by the testing team during this discussion to drive clarity for the requirements (e.g., message persistence duration, the number of messages limits, and the priority of messages). Development teams focus on identifying challenges in solution design. These discussions usually happen in pre-iteration planning meetings (also referred as backlog grooming meetings). Also, the work is divided into several small user stories. User stories are one of the primary development artifacts for Scrum project teams. A user story is a simplified representation of a software feature from a customer point of view. It describes what customer wants in terms of acceptance criteria. Also, it explains why the customer needs that feature that helps in the understanding of business value.

For example, following user stories are identified for the “small messaging service”:

The division of the user stories is completely at the discretion of the team and business owner and what they think appropriate to divide the work. Prioritization of the stories is also done during the meeting based on the effort and priority from the business owner.

After prioritization of stories is completed, and a backlog is prepared, a sprint/iteration can be started by taking a small chunk from the top of the backlog. In the sprint, developers and testers work in parallel. Every day, the team meets and discusses what has to be done that day and what testing has to be done to verify the work. Quality is thus embedded into working software delivered by the team with constant collaboration, negotiating better designs for easy testability.

Testers participate from day one and insist on writing code that is testable. This emphasis often leads to use design patterns extensively and achieves better code layout, architecture, and quality. Design pattern means a reusable solution to a commonly occurring problem within a given context in software design.

In the present illustration, as testers and developers approach first user story in Table 1, they have to agree on a standard API or interface to write their code or test, to start their work in parallel as in interface-based programming pattern. Teams agree upon interface “FetchUndeliveredUserMessageResource” that contains “fetchUndeliveredUserMessages” method, to accept “userId” as the parameter as described in Table 2. Testers start writing their integration tests, and developers start their implementation in parallel.

Interface-based programming is claimed to increase the modularity of the application and hence its maintainability. However, to guarantee low coupling or high cohesion, we need to follow a couple of guiding principles. First one is single responsibility principle. It suggests that every class should have the responsibility for a single functionality or behavior, and the class should encapsulate that implementation of that responsibility. This principle keeps tedious bug prone code contained. The second one is interface segregation principle. It suggests that no client should be forced to depend on methods that do not use. In that cases, interfaces are from the third party, using adapter pattern that makes the code well organized. The basic idea of adapter pattern is changing one interface to the desired interface.

Table 1 User stories

<div>1. As a user, I want to receive all persisted messages at login so that I can know all the pending messages.</div> <div>Acceptance Criteria: - all messages received when the user offline has to be received by the user when he comes online</div>
<div>2. As a user, I want to persist messages when I am not online so that those messages are not lost.</div> <div>Acceptance Criteria: - store user message, if not delivered.</div>
<div>3. As a user, I want to receive messages when I am online so that I can be aware of events happening around me.</div> <div>Acceptance Criteria: - when online, receive notifications immediately as they happen.</div>
<div>4. As a user, I want to receive messages in a priority order, so that important messages are received first.</div> <div>Acceptance Criteria: - sort messages based on the priority and deliver the important one first. - Higher the priority.. higher the importance.</div>
<div>etc.,</div>

Table 2 *FetchUndeliveredUserMessageResource* interface

@Path("/api")	// HTTP base URI Java Annotation
public interface FetchUndeliveredUserMessageResource {	
@POST	// Supports both POST and GET
@GET	
@Path("/{userId}/getundeliveredmessages")	// URI for this resource
@Consumes("application/json")	// Content-Type is application/json
void fetchUndeliveredUserMessages(@PathParam("userId") String userId);	// userId is the parameter
}	

In the present illustration, as testers and developers approach second user story in the Table 1, the team identifies the need to separate database retrieval so that it can be tested independently and defines “PersistMessage” interface to access data from the database as described in Table 3. This repository design pattern comes handy to separate business logic, and data model, and retrieval. This pattern centralizes the data logic or Web service access logic and provides a substitution point for the unit tests and makes it flexible.

In the present illustration, as testers and developers approach third user story in Table 1, “SendMessageResource” interface has been identified as described in Table 4. As the testing team writes tests for these user stories, they would design them to run in isolation and test each story independently and in isolation. It reduces test complexity and forces development to reduce complex dependencies between components. As much as possible, no test is made dependent on another test to run.

Also, to reduce interdependencies among different components and classes, dependency injection pattern is used.

Dependency injection is a software design pattern that implements inversion of control for software libraries. That means component delegates to external code (the injector) the responsibility of providing its dependencies. The component is not allowed to call the injector system. This separation makes clients independent and easier to write a unit test using stubs or mock objects. Stubs and mock objects are used to simulate other objects that are not in the test. This ease of testing is the first benefit noticed when using dependency injection. Better testable code often produces simple and efficient architected system.

Table 3 *PersistMessage* interface

public interface PersistMessage {
public boolean persist(Message msg);
public boolean delete(Message msg);
}

Table 4 *SendMessageResource* interface

```

@Path("/api")
public interface SendMessageResource {

    @POST
    @Path("/api/type/{userId}/sendmsg")
    @Consumes("application/json")
    void sendMessageToUser(Map<String, Object> data);
}

```

When the sprint is in progress, if new work is identified, or priorities changed by business, product backlog is groomed as appropriate.

A retrospective meeting is required to talk freely about what good practices has to be continued and what practices need to be stopped or improved. This meeting happens at the end of the sprint. As the team progress during the sprint and integrate their work and do build and test, fixed as often they can, it necessitates better test engineering practices.

6 Engineering of Agile Testing

Agile methods do not provide actual solutions but do a pretty good job of surfacing problems early. The motto is to “catch issues fast and nip them in bud.”

Agile testing team has to test every iteration carefully as if it is a production release and integrate it into the final product. As a result of repeated integration tests, testing team needs to design and implement test infrastructure and rapid development and release tools. The success of Agile testing depends on the team’s ability to automate test environment creation and automatic release candidate validation and report test metrics.

6.1 Continuous Integration

As teams deliver fully functional features every iteration in Agile practice, they need to work ways to reduce build, deployment, and running test overhead. CI is a practice that requires developers to integrate code several times a day. Each time, a

developer checks in code, which is verified by an automated build, allowing teams to detect problems early. CI allows automated tests to grow, live, and thrive. By integrating regularly, errors can be detected quickly and easily.

6.2 *Automated Build System*

Automated build system enables the team to do CI. It has become a cornerstone of Agile development.

Automated build system reduces the cost associated with bug fixing exponentially and improves quality and architecture. There have been several sophisticated continuous build systems available such as Jenkins [7], Hudson [8], Team Foundation Server [9], and Apache Continuum [10].

Build systems need to be optimized for performance, ease of use by developers, incremental execution of tests, and software language agnostic.

In the above illustration, the team used Jenkins as their automated build system. Every time, a developer checks in code, or tester checks in a test in which a new build is made, and all the unit and integration tests are executed. If any issues are found, a build failure email is sent to the team. Jenkins makes the deployment artifact as an Redhat package manager (RPM). These RPMs are deployed to the team environment by deployment scripts automatically so that testers can perform post-deployment verification as needed. Next section does a comparative study of Agile testing and its advantages and disadvantages.

7 *Agile Testing: An Analysis*

Let us briefly review the distinction between Agile and Spiral models, as there is apparent similarity between the two. Then, an analysis is made showing both advantage and disadvantage in Agile testing [11].

7.1 *Comparison of Agile and Spiral Model*

The common aspect of Agile process and Spiral model is iterative development and differ in most of the other aspects. In case of Spiral model, length of iteration is not specified, and an iteration can run for months or years. But in Agile model, length must be small and usually, it is about 2–3 weeks. In Spiral model, iterations are often well planned out in advance and executed in order of risk. In Agile process, focus is on one iteration at a time, and priority of user stories drives the execution order instead of risk. Within each iteration, Spiral model follows traditional

waterfall methodology. Agile recognizes the fact that people build technology and focuses “test-first” approaches, its collaboration, and interactions within the team.

7.2 Advantages of Agile Testing

Knowledge transfer happens naturally between developers and testers as they work together with constant feedback on each iteration. Junior team members benefit from active feedback, and testers perceive the job environment as comfortable. Resilience is built into the teams and nourishes team atmosphere.

Small teams with excellent communication require small amounts of documentation. This collaboration facilitates learning and understanding for each other. There is no practical way to comprehend or measure this particular advantage but produces an awesome environment and team.

Requirements volatility in projects is reduced because of small projects and timelines. The wastage of unused work (requirements documented, unused components implemented, etc.,) is reduced as the customer provides feedback on developed features and helps prioritize and regularly groom the product backlog. Customers appreciate active participation in projects.

Small manageable tasks and CI helps process control, transparency, and increase in quality. Very few stories are in progress at any point in time and handled in priority order, and this reduces stress on teams. The risk associated with unknown challenges is well managed. Frequent feedback makes problems and successes transparent and provides high incentives for developers to deliver high quality.

As bugs are found close to the development, they can be fixed with very limited extra overhead and thus increase actual time spent on designing and developing new features (Fig. 5).

The quality of work life improves as the social job environment is trustful, peaceful, and responsible.

7.3 Disadvantages of Agile Testing

As the focus on intra-team communication increases too much, it sometimes leads to inter-team communication issues, and “us” versus “them” attitude between the teams. Teamwork is critical to the success of Agile testing, and it is essential that team members are willing to work together. Personalities of the team members play a big role in Agile practices.

Agile methods have lots of challenges with scaling them to large groups and organizations. Coming up the division of teams and projects is often not trivial needs lot of commitment and patience. Generating the priority list of all the projects, broken up into small stories, is a herculean task and so also to maintain.

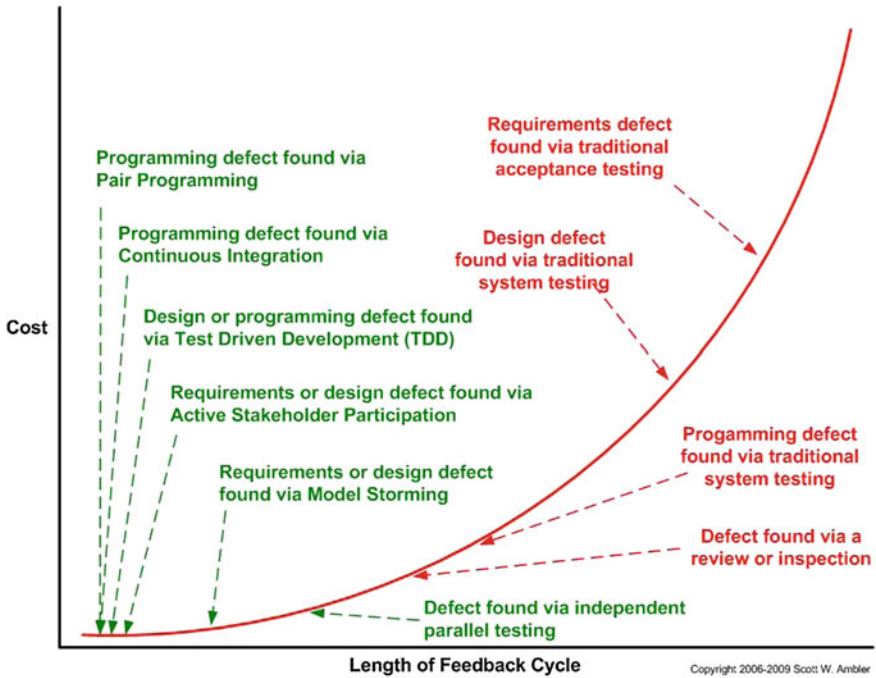


Fig. 5 Cost of feedback cycle [12]

Implementation starts very early in the process, and sometimes, enough time is not spent on design and architecture. Poor design choices might lead to rework. Sometimes, dependencies buried inside implementation details are hard to identify, often causes impediments to the team. Many people would be required to manage increased number of builds, releases, and environments.

8 Conclusion

Agile testing combines test and development teams around the principles of collaboration, flexibility, simplicity, transparency, and retrospection. Its focus is on positive desire to implement a solution that passes the test, rather than negative desire to break a solution. Agile practices do not provide actual solutions but do a pretty good job of surfacing them early. Developers and testers are empowered to work toward solving problems.

Agile has already won the battle for mainstream acceptance, and its general precepts are going to remain viable for some time. As empowerment of people is key to good governance so is for software development. Agile approaches empower

a team possibly replacing command control structures in organizations with more democratic practices. So, the success of this process is well predicted. Hence, both academia and industry are currently showing increasing interest in Agile testing. This chapter with an intention to highlight this upcoming approach from industry perspective has briefly described both Agile software development process and Agile testing approach. The approach has been illustrated with a case study. We aim the chapter will be useful to the beginners interested in this nascent area of software testing.

References

1. <http://www.computerweekly.com/feature/Why-agile-development-races-ahead-of-traditional-testing>
2. https://en.wikipedia.org/wiki/Technical_debt
3. <http://www.agilemanifesto.org/>
4. <http://www.extremeprogramming.org/>
5. <https://www.scrumalliance.org/why-scrum>
6. <https://www.scrumalliance.org/scrum/media/ScrumAllianceMedia/Files%20and%20PDFs/Why%20Scrum/ScrumAlliance-30SecondsFramework-HighRes.pdf>
7. <https://jenkins-ci.org/>
8. <http://hudson-ci.org/>
9. <https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx>
10. <https://continuum.apache.org/>
11. <http://www.sciencedirect.com/science/article/pii/S0164121209000855>
12. <http://www.ambysoft.com/essays/agileTesting.html>
13. <http://www.agilemanifesto.org/principles.html>
14. <https://www.mountangoatsoftware.com/blog/agile-succeeds-three-times-more-often-than-waterfall>

Trends in Software Testing

Mohanty, H.; Mohanty, J.R.; Balakrishnan, A. (Eds.)

2017, XVII, 176 p. 65 illus., 60 illus. in color., Hardcover

ISBN: 978-981-10-1414-7