ASD -List of Experiments

1. **Understand the importance of Agile Approach in Software Development**

2. **Understand the business value of adopting Agile approaches.**

3. **Case Study of SCRUM.**

4. **Study of Test Driven Development in Agile Methodology.**

5. **Apply design principles and refactoring to achieve Agility.**

6. **To study automated build tool.**

7. **To study version control tool in Agile.**

8. **To study Continuous Integration tool.**

9. **Case Study of XP(Extreme Programming)**

10. **Study of Agile approach to Quality Assurance.**

Experiment No. 1

Objective-Understand the importance of Agile Approach in Software Development

Analysis:

1. Difference between agile software development model and waterfall model.
2. Why Agile is better?
3. Understanding the Agile Manifesto
4. Discussing Important Characteristics that make agile approach best suited for Software Development.

Theory

Agile software development is a group of software development methods in which requirements and solutions evolve through collaboration between self- organizing, cross-functional teams. It promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and encourages rapid and flexible response to change.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

Agile principles

The Agile Manifesto is based on 12 principles:

1. Customer satisfaction by rapid delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential

11. Self-organizing teams
12. Regular adaptation to changing circumstance

What's wrong With Traditional Approaches?

In 1970, Dr. Winston Royce presented a paper entitled "Managing the Development of Large Software Systems," which criticized sequential development. He asserted that software should not be developed like an automobile on an assembly line, in which each piece is added in sequential phases, each phase depending on the previous. Dr. Royce recommended against the phase based approach in which developers first gather all of a project's requirements, then complete all of its architecture and design, then write all of the code, and so on. Royce specifically objected to the lack of communication between the specialized groups that complete each phase of work.

It's easy to see the problems with the waterfall method. It assumes that every requirement can be identified before any design or coding occurs. Could you tell a team of developers everything that needed to be in a software product before any of it was up and running? Or would it be easier to describe your vision to the team if you could react to functional software? Many software developers have learned the answer to that question the hard way: At the end of a project, a team might have built the software it was asked to build, but, in the time it took to create, business realities have changed so dramatically that the product is irrelevant. Your company has spent time and money to create software that no one wants. Couldn't it have been possible to ensure the end product would still be relevant before it was actually finished?

Today very few organizations openly admit to doing waterfall or traditional command and control. But those habits persist.

Why Agile?

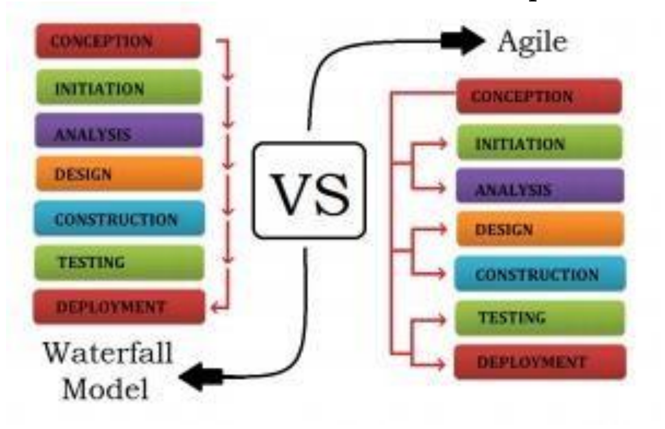
Agile development provides opportunities to assess the direction throughout the development lifecycle. This is achieved through regular cadences of work, known as Sprints or iterations, at the end of which teams must present a potentially shippable product increment. By focusing on the repetition of abbreviated work cycles as well as the functional product they yield, agile methodology is described as "iterative" and "incremental." In waterfall, development teams only have one chance to get each aspect of a project right. In an agile paradigm, every aspect of development — requirements, design, etc. — is continually revisited. When a team stops and re-evaluates the direction of a project every two weeks, there's time to steer it in another direction.

This "inspect-and-adapt" approach to development reduces development costs and time to market. Because teams can develop software at the same time they're gathering requirements, "analysis paralysis" is less likely to impede a team from making progress. And because a team's work cycle is limited to two weeks, stakeholders have recurring opportunities to calibrate releases for success in the real world. Agile development helps companies build the right product. Instead of committing to market a piece of software that hasn't been written yet, agile empowers teams to continuously replan their release to optimize its value throughout development, allowing them to be as competitive as possible in the marketplace. Agile development preserves a product's critical market relevance and ensures a team's work doesn't wind up on a shelf, never released.

Difference between agile software development model and waterfall model

It is worth mentioning here that the Waterfall model is the primitive model type and has been implemented in the development phase time after time. Hence in the due course if time developers found many drawbacks in this model which were later

rectified to form various other development models.



Advantages of the Agile Methodology

1. The Agile methodology allows for changes to be made after the initial planning. Re-writes to the program, as the client decides to make changes, are expected.

2. Because the Agile methodology allows you to make changes, it's easier to add features that will keep you up to date with the latest developments in your industry.

3. At the end of each sprint, project priorities are evaluated. This allows clients to add their feedback so that they ultimately get the product they desire.

4. The testing at the end of each sprint ensures that the bugs are caught and taken care of in the development cycle. They won't be found at the end.

5. Because the products are tested so thoroughly with Agile, the product could be launched at the end of any cycle. As a result, it's more likely to reach its launch date.

Conclusion

Both the Agile and waterfall methodologies have their strengths and weaknesses. The key to deciding which is right for you comes down to the context of the project. Is it going to be changing rapidly? If so, choose Agile. Do you know exactly what you need? Good. Then maybe waterfall is the better option. Or better yet? Consider taking aspects of both methodologies and combining them in order to make the best possible software development process for your project.

Experiment No. 2

Objective-Understand the business value of adopting Agile approaches.

Analysis:

1. Understanding the business values
2. Adopting Agile Software Development: Issues and Challenges
3. Overview of Scrum, Extreme Programming, Feature Driven development, Lean Software Development, Agile project management

Theory

When it comes to creating custom applications, too many of us live in denial. We want to believe that it's possible to predict accurately how long a group of developers will take to build software that meets our requirements. We also want to believe that we can know what those requirements are up front, before we've seen any part of the application, and that the requirements won't change during development. Sadly, none of these things are true for most projects. We can't predict how long development will take, largely because we can't get the requirements right up front and we can't stop them from changing. Because we deny these realities, many organizations still use software development processes that don't work well. Fortunately, this is changing. Agile development processes get more popular every day, primarily because they're rooted in reality: They're designed to accommodate change. Doing software development in this way can be scary at first, especially for the business leaders who are footing the bill. This needn't be the case, however. The truth is that agile processes are usually better both for development teams and for the business people who pay them. To understand why this is true, we need to start by understanding what an agile process really is. What Agile Development Means The challenge is always the same: We need to create software in the face of uncertainty. At the start of a development project, we don't know whether we've defined the project's requirements correctly. We also don't know how those requirements will change while we're building the software. A traditional development process does its best to pretend this uncertainty doesn't exist. In the classic waterfall approach, for example, an organization creates detailed plans and precise schedules before writing any code. But real development projects rarely comply with these plans and schedules—they're notoriously unruly. The core reason for this is that even though we use the term "software engineering", writing code isn't like other kinds of engineering. In traditional engineering projects—building a bridge, say, or constructing a factory—it's usually possible to define stable requirements up front. Once you've done this, creating plans and schedules based on previous experience is straightforward. Software development just isn't like this one . Creating stable requirements up front is usually impossible, in part because people don't know what they want until they see it. And since every development project involves some innovation—if it doesn't, you should be buying rather than building the software—uncertainty is unavoidable. Traditional development processes work against these realities. Agile processes, however, are designed for this situation. Because requirements change, an agile process provides a way to add, remove, and modify requirements during the development process. Rather than resisting change, an agile process embraces it. Just as important, the process recognizes that short-term plans are more stable than long-term plans. You might not know exactly what you want to be doing three months from now, but you probably do know what you want to do in the next three weeks. To accomplish this, an agile development process relies on iteration. Rather than the traditional approach of defining all of the requirements, writing all of the code, then testing that code, an agile process does these things over and over in smaller iterations. Each iteration creates more of the finished product, with the requirements updated at the start of each one.

Challenges in adopting agile Methodology

Challenge 1: Missing the Agile Master Role Agile master or Agile coach is an essential role during Agile adopting process in any organization. Agile coach is considered a consultant for the team in every step of a project using any Agile method, such as Scrum, that is responsible of providing guidance and helps to succeed in adopting Agile. Entity"S" management recognized the need to hire a contractor as an agile master. However the position was not filled due to financial constraints.

Challenge 2: The overzealous teams after attending a course on Agile methods, many of entity "S" teams wanted to adopt Agile methods as soon as possible hoping it will solve all their previous development challenges known for traditional methods. This overzealous team fast adoption of Agile resulted in a decrease in productivity because the development cycle took longer time due to many mistakes in implementation. This decrease in productivity led many team members to be less optimistic and started to lose interest in agile methods.

Challenge 3: The absent of a Pilot Project Another challenge is the absent of a pilot project in the transition from the previous traditional method to the scrum method. Conducting a pilot project was a recommended step in the adoption of agile development for the first time. As a part of the plan to adopt agile method, the pilot project is essential to evaluate how"S" environment will be able to move from the previous heavy-weight method to a new light method. Many organizations went through the same experience of running a pilot project especially those companies that have large projects such as Amazon, Yahoo, Microsoft and Intel. After investing the needed time and resources they have reached to a successful adoption of Agile.

Challenge 4: Scrum Implementation International Journal of Managing Value and Supply Chains (IJMVSC) Vol. 2, No. 3, September 2011 7 Although the employees in "S" were very experienced but yet none of them had any previous experience with agile development methods or Scrum implementation in particular. This is in addition of the absence of the agile master. For the team members, scrum implementation was not easy as it appeared to be during the training session. The team members find themselves, suddenly, in a completely new set up. The experience of traditional methods is completely different than committing to daily meeting, working with time boxes, finishing tasks in small period iteration and documenting the stories (or backlogs) in a different way.

Challenge 5: Current Work Pressure Although"S" software development team serves a very large organization of over 30 departments and developed numerous projects through the years, the development projects require continues maintenance and support. In addition, the team was working on a new project with firm deadlines. The work environment was very demanding and team worked under pressure to produce products according to the planned schedule. Scrum adoption process started while every member of the team was engaged in his/her everyday tasks. With such work pressure the daily Scrum meetings were considered waste of time and added extra pressure to the employees. They used to meet weekly and later twice a month and then only when required and usually after working hours. As teams started to skip daily meetings it also affected the learning process of scrum between the team members. That eventually leads to the failure of learning and implementing agile method correctly.

Challenge 6: Upper Management Concerns The upper management of"S" had many concerns about the effectiveness and success of the transition to a new method. They were not easily convinced to invest in a new method.

Challenge 7: Governmental bureaucratic System The traditional method currently in "S" was customized to comply with the governmental system of other department. The new Agile method being introduced, Scrum, is developed in such highly bureaucratic environment. The Agile team has to secure approvals and signatures before moving from one step to another. This was perceived by the team members as unnecessary and more time was taken into account to develop a new project. The scrum method requires much less correspondence, less time in communication between the customer and the team and requires significantly less paper work and approvals as the customer is supposed to be involved in every step.

Challenge 8: Documentation requirements after years and years of extensive documentation of every step in the traditional method, moving to a new method with minimum documentation requirements was one of the greatest challenges. Every project used to end up with dozens of document such as project charter, project plan, testing plan, SRS, STS, technical documents, user manual, etc. Each of these document contained large number of pages written by every member of the team and consumed hours of the valuable development time. The documentation requirements were driven basically from the previous challenge (the governmental system), upper management, ISO certificate requirements and the traditional development method that is currently used. Although agile development promises sufficient documentation of the projects, it didn't seem very convincing to the upper management when they end up receiving few documents in comparison with the previous model of documentation. Many attempts were made to try to balance between the upper management requirement regarding documentation and between adopting Scrum method. Agile teams started to increase the number of documents required for documentation and started to customize Scrum as much as possible to conform to all the upper management requirements of documentation norms. This did not work very well and it created extra burden on the agile teams

Agile Process Examples

1. **SCRUM**
2. **FDD**
3. **Lean software development**
4. **XP**

SCRUM
Scrum is an iterative and incremental agile software development methodology for managing product development. It defines "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal", challenges assumptions of the "traditional, sequential approach" to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines in the project.

FDD

Feature-driven development (**FDD**) is an iterative and incremental software development process. It is one of a number of lightweight or agile methods for developing software. FDD blends a number of industry-recognized best practices into a cohesive whole. These practices are all driven from a client-valued functionality (feature) perspective. Its main purpose is to deliver tangible, working software repeatedly in a timely manner.

Lean software development

Lean software development (**LSD**) is a translation of lean manufacturing and lean IT principles and practices to the software development domain. Adapted from the Toyota Production System,[1] a pro-lean subculture is emerging from within the Agile community. Lean is most popular with startups that want to penetrate the market, or test their idea and see if it would make a viable business.

XP

Extreme programming (**XP**) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed *continuously*, i.e. the practice of pair programming.

Critics have noted several potential drawbacks,[5] including problems with unstable requirements, no documented compromises of user conflicts, and a lack of an overall design specification or document.

Conclusion

Each agile methodology has a slightly different approach for implementing the core values from the Agile Manifesto, just as many computer languages manifest the core features of object-oriented programming in different ways. A recent survey shows that about 50 percent of agile practitioners say that their team is doing Scrum. Another 20 percent say that they are doing Scrum with XP components. An additional 12 percent say that they are doing XP alone. Because more than 80 percent of agile implementations worldwide are Scrum or XP, MSF for Agile Software Development v5.0 focuses on the core processes and practices of Scrum and XP

Experiment No. 3

Objective- Case Study of SCRUM.

Theory

Scrum is an iterative and incremental agile software development methodology for managing product development. It defines "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal", challenges assumptions of the "traditional, sequential approach" to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines in the project.

A key principle of scrum is its recognition that during a project the customers can change their minds about what they want and need (often called "requirements churn"), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.

History

Scrum was first defined as "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal" as opposed to a "traditional, sequential approach" in 1986 by Hirotaka Takeuchi and Ikujiro Nonaka in the *New New Product Development Game*. Takeuchi and Nonaka later argued in *The Knowledge Creating Company* that it is a form of "organizational knowledge creation, [...] especially good at bringing about innovation continuously, incrementally and spirally".

The authors described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, photocopier and printer industries. They called this the *holistic* or *rugby approach*, as the whole process is performed by one cross-functional team across multiple overlapping phases, where the team "tries to go the distance as a unit, passing the ball back and forth". (In rugby football, a scrum refers to a tight-packed formation of players with their heads down who attempt to gain possession of the ball.)

In the early 1990s, Ken Schwaber used what would become scrum at his company, Advanced Development Methods, and Jeff Sutherland, with John Scumniotales and Jeff McKenna, developed a similar approach at Easel Corporation, and were the first to refer to it using the single word *scrum*. In 1995, Sutherland and Schwaber jointly presented a paper describing the *scrum methodology* at the Business Object Design and Implementation Workshop held as part of Object-Oriented Programming, Systems, Languages & Applications '95 (OOPSLA '95) in Austin, Texas, its first public presentation. Schwaber and Sutherland collaborated during the following years to merge the above writings, their experiences, and industry best practices into what is now known as scrum.

In 2001, Schwaber worked with Mike Beedle to describe the method in the book *Agile Software Development with Scrum.*Its approach to planning and managing projects is to bring decision-making authority to the level of operation properties and certainties. Although the word is not an acronym, some companies implementing the process have been known to spell it with capital letters as SCRUM. This may be due to one of Ken Schwaber's early papers, which capitalized SCRUM in the title.

Later, Schwaber with others founded the Scrum Alliance and created the Certified Scrum Master programs and its derivatives. Schwaber left the Scrum Alliance in the fall of 2009, and founded Scrum.org to further improve the quality and effectiveness of scrum.

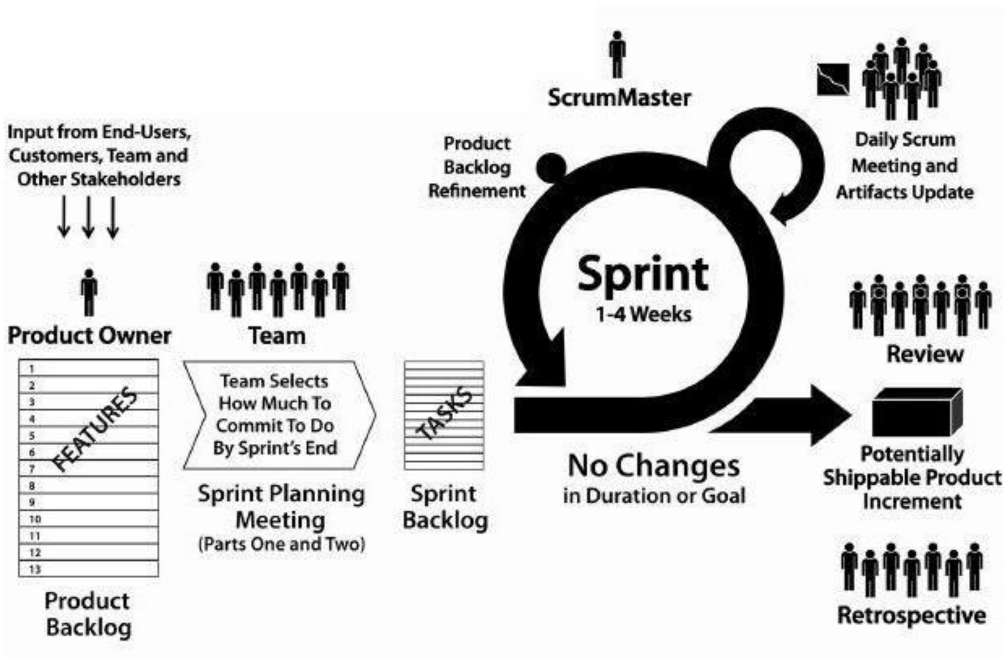

Input from End-Users, Customers, Team and Other Stakeholders
Product Owner
Team
ScrumMaster
Product Backlog Refinement
Daily Scrum Meeting and Artifacts Update
Sprint
1-4 Weeks
Review
FEATURES
Team Selects How Much To Commit To Do By Sprint's End
TASKS
No Changes
in Duration or Goal
Potentially Shippable Product Increment
Sprint Planning Meeting
(Parts One and Two)
Sprint Backlog
Product Backlog
Retrospective

Figure: **SCRUM Process**

How Does Scrum Fit With Agile?

The Agile Manifesto doesn't provide concrete steps. Organizations usually seek more specific methods within the Agile movement. These include Crystal Clear, Extreme Programming, Feature Driven Development, Dynamic Systems Development Method (DSDM), Scrum, and others. While I like all the Agile approaches, for my own team Scrum was the one that enabled our initial breakthroughs. Scrum's simple definitions gave our team the autonomy we needed to do our best work while helping our boss (who became our *Product Owner*) get the business results he wanted. Scrum opened our door to other useful Agile practices such as test-driven development (TDD). Since then we've helped businesses around the world use Scrum to become more agile. A truly agile enterprise would not have a "business side" and a "technical side." It would have teams working directly on delivering business value. We get the best results when we involve the whole business in this, so those are the types of engagements I'm personally the most interested in.

What's The Philosophy behind Scrum?

Scrum's early advocates were inspired by empirical *inspect and adapt* feedback loops to cope with complexity and risk. Scrum emphasizes decision making from real-world results rather than speculation. Time is divided into short work cadences, known as sprints, typically one week or two weeks long. The product is kept in a potentially shippable (properly integrated and tested) state at all times. At the end of each sprint, stakeholders and team members meet to see a demonstrated potentially shippable product increment and plan its next steps.

Scrum is a simple set of roles, responsibilities, and meetings that never change. By removing unnecessary unpredictability, we're better able to cope with the necessary unpredictability of continuous discovery and learning.

Scrum Roles

- **Product Owner:** The Product Owner should be a person with vision, authority, and availability. The Product Owner is responsible for continuously communicating the vision and priorities to the development team.

  It's sometimes hard for Product Owners to strike the right balance of involvement. Because Scrum values self-organization among teams, a Product Owner must fight the urge to micro-manage. At the same time, Product Owners must be available to answer questions from the team.

- **Scrum Master:** The Scrum Master acts as a facilitator for the Product Owner and the team. The Scrum Master does not manage the team. The Scrum Master works to remove any impediments that are obstructing the team from achieving its sprint goals. This helps the team remain creative and productive while making sure its successes are visible to the Product Owner. The Scrum Master also works to advise the Product Owner about how to maximize ROI for the team.
- **Team:** According to Scrum's founder, "the team is utterly self managing." The development team is responsible for self organizing to complete work. A Scrum development team contains about seven fully dedicated members (officially 3-9), ideally in one team room protected from outside distractions. For software projects, a typical team includes a mix of software engineers, architects, programmers, analysts, QA experts, testers, and UI designers. Each sprint, the team is responsible for determining how it will accomplish the work to be completed. The team has autonomy and responsibility to meet the goals of the sprint.

Scrum Meetings

There are number of regular meetings take place among Agile Team members. Let's go through different type of meetings.

1. Sprint Planning Meeting

During a sprint team works on selected features. Those features are planned and selected in sprint planning meeting which holds before every sprint. Product owner and scrum team participate this planning session which usually last for 4 hours.

Product owner come up with a set of features to discuss with intent to add into sprint backlog so team could work on it during the sprint. Usually those features have a specific theme attach to them, that theme is also called sprint goal.

Following are a few example of the sprint goal.

- Improve the performance of the current system.
- Build the UI of certain component.
- Create API of the product so other application could access some specific information.

Based on input from the team, features are finalized for the sprint. Team input helps to decide if proposed features can be completed in the sprint or there are any impediments involve in completing those features. Features story points should be fall within sprint velocity. Once features are decided scrum team breaks those features into tasks. Team may assign hours or story point to each task. The output of the meeting is a sprint backlog, sprint theme/goal and tasks related to each feature.

Timings: Before every Sprint

Duration: 4-8 hours

Participants: Scrum Owner, Scrum master, Scrum Team

Artifacts: Sprint Tag/Goal, sprint backlog, tasks

2. Release Planning Meeting

Purpose of the meeting

An Agile project may consist of number of iterations (sprints). A release consists of a sub set of such iterations. The length of release could be between 2 to 6 months. Release planning meeting is held at the beginning of each release. The purpose of the release planning meeting is to go through the backlog and give an estimate the number and set the priority of features that can be completed in a release. At the start of the meeting Product owner usually come up with the backlog with selected features he wants to get completed during the release time frame. Scrum team gives its opinion regarding the features and based on the discussion certain features are added or removed from the release, or their priority and time to complete may change. The technical knowledge of the team gets very helpful in this planning session. The scrum team may prefer to complete certain issues before than others because of architecture design or some other technical issues that product owner does not have much knowledge about. An important measure which plays a vital role in release planning and estimation meeting is Velocity. If it’s not the first release, velocity of previous releases can be helpful in finding out number of sprint may take to complete a release. If the velocity is not known then team have to estimate the work that can be completed in a release.

Initial estimate of the team can go wrong. In those situations team can hold another planning meeting to adjust the estimates based on real progress. With time velocity gets more predictable the release planning meeting start giving more accurate estimates based on velocity.

Timings: Beginning of each release or revised during the release

Duration: 4 – 16 hours

Participants: product owner, scrum master, scrum team

Best Practices:

- Project owner come with prioritized and estimated features, so that team could create a schedule for the release.
- Use the velocity to determine the number of sprints.
- Don't dictate technical team with your technical choices.
- Don't try to push features against the team recommendations.
- If teams are very big than one person representing each area should attend the meeting.

Artifacts: Release backlog.

3. Review Meetings

a. Sprint Review (Demo) Meeting

Purpose of the meeting

Once sprint is completed Sprint team present a demonstration of the features completed during the sprint. The Product owner and customer review the features and give their comments on the completed features. Product owner may accept or reject the feature or point out the deficiency or bug in the work completed during the sprint. Features that fulfill the definition of done, provided by the product owner, are accepted. The fate of those features that are not completed or partially completed decided during next sprint planning meeting.

Timings: After sprint completion.

Duration: 2 hours

Participants: Product owner, customer, scrum team

Artifacts: completed features.

b. Sprint Retrospective Meeting

Purpose of the meeting

The purpose of Retrospective meeting is to discuss with team what processes or practices went well during the sprint? and what process and practices need to be improved? In this meeting only scrum team and scrum team participate. This is the last meeting of the sprint.

Following are some of the examples.

- Smaller task work better than larger ones
- Definition of done need to be more clear
- Information radiators need to be placed where everyone could see them
- Daily meeting should not exceed 15 minutes.

Timings: After Sprint Review meeting

Duration: 2-4 hours

Participants: Scrum Team, Scrum Master

Best Practices: Following are some of the best practices we can follow.

- Use time line of the sprint to get feed back
- Make team comfortable and relax
- Try to let team to reach a consensus without scrum master interuptions
- Don't try to cover events outside of the sprint

Artifacts: better processes and practices.

4. Daily Stand-up Meeting

Purpose of the meeting

The Daily stand-up or daily scrum meeting is the most important scrum meeting. The purpose of the meeting is to understand what team members are working on and propose them a solution for any impediment they may be facing. Each scrum team member answer following three questions in each of the meeting.

What you have done since last meeting?

What, if anything, is preventing you to perform your task?

What you will do between now and next meeting?

Timings: Daily

Duration: 15 minutes

Participants: Scrum master, scrum team (other can participate as a silent participants)

Best Practices: Following are some of the best practices we can follow.

- Don't exceed more than 15 minutes.
- Only scrum team speaks, product owner can attend the meeting as silence participant
- Fix the time for the meeting.
- Don't discuss anything out of three questions mentioned above.

Artifacts

Product backlog

The *product backlog* comprises an ordered list of *requirements* that a scrum team maintains for a product. It consists of features, bug fixes, non-functional requirements, etc.—whatever needs doing in order to successfully deliver a viable product. The product owner orders the *product backlog items* (PBIs) based on considerations such as risk, business value, dependencies, and date needed.

Items added to a backlog are commonly written in story format. The product backlog is *what* will be delivered, ordered into the sequence in which it should be delivered. It is open and editable by anyone, but the product owner is ultimately responsible for ordering the items on the backlog for the development team to choose.

The product backlog contains the product owner's assessment of business value and the development team's assessment of development effort, which are often, but not always, stated in story points using a rounded Fibonacci sequence. These estimates help the product owner to gauge the timeline and may influence ordering of backlog items; for example, if the "add spellcheck" and "add table support" features have the same business value, the product owner may schedule earlier delivery of the one with the lower development effort (because the return on investment is higher) or the one with higher development effort (because it is more complex or riskier, and they want to retire that risk earlier).

The product backlog and the business value of each backlog item is the responsibility of the product owner. The size (i.e. estimated complexity or effort) of each backlog item is, however, determined by the development team, who contributes by sizing items, either in story points or in estimated hours.

There is a common misunderstanding that only user stories are allowed in a product backlog. By contrast, scrum is neutral on requirement techniques. As the Scrum Primer states,

Product Backlog items are articulated in any way that is clear and sustainable. Contrary to popular misunderstanding, the Product Backlog does not contain "user stories"; it simply contains items. Those items can be expressed as user stories, use cases, or any other requirements approach that the group finds useful. But whatever the approach, most items should focus on delivering value to customers.

Scrum advocates that the role of product owner be assigned. The product owner is responsible for maximizing the value of the product. The product owner gathers input and takes feedback from, and is lobbied by, many people, but ultimately makes the call on what gets built.

The product backlog is used to:

capture requests for modifying a product. This can include adding new features, replacing old features, removing features and fixing issues

ensure the development team is given work which maximizes the business benefit to the owner of the product

Typically, the product owner and the scrum team come together and write down everything that needs to be prioritized and this becomes content for the first sprint, which is a block of time meant for focused work on selected items that can be accommodated within a timeframe. The product backlog can evolve as new information surfaces about the product and about its customers, and so later sprints may address new work.

The following items typically comprise a scrum backlog: features, bugs, technical work, and knowledge acquisition. Web development can entail confusion as to the difference between a feature and a bug: technically a feature is "wanted", while a bug is a feature that is "unintended" or "unwanted" (but may not be necessarily a defective thing). An example of technical work would be: "run virus check on all developers' workstations". An example of knowledge acquisition could be a scrum backlog item about researching Wordpress plugin libraries and making a selection.

Managing the product backlog between product owner and scrum team

A backlog, in its simplest form, is merely a list of items to be worked on. Having well-established rules about how work is added, removed and ordered helps the whole team make better decisions about how to change the product.

The product owner prioritizes which of the items in the product backlog are most needed. The team then chooses which items they can complete in the coming sprint. On the scrum board, the team moves items from the product backlog to the sprint backlog, which is the list of items they will build. Conceptually, it is ideal for the team to only select what they think they can accomplish from the top of the list, but it is not unusual to see in practice that teams are able to take lower-priority items from the list along with the top ones selected. This normally happens because there is time left within the sprint to accommodate more work. Items at the top of the backlog, the items that are going to be worked on first, should be broken down into stories that are suitable for the development team to work on. The further down the backlog goes, the less refined the items should be. As Schwaber and Beedle put it "The lower the priority, the less detail, until you can barely make out the backlog item.

As the team works through the backlog, it needs to be assumed that "changes in the world can happen"—the team can learn about new market opportunities to take advantage of, competitor threats that arise, and feedback from customers that can change the way the product was meant to work. All of these new ideas tend to trigger the team to adapt the backlog to incorporate new knowledge. This is part of the fundamental mindset of an agile team. The world changes, the backlog is never finished.

Sprint backlog

The *sprint backlog* is the list of work the development team must address during the next sprint. The list is derived by selecting product backlog items from the top of the product backlog until the development team feels it has enough work to fill the sprint. This is done by the development team asking "Can we also do this?" and adding product backlog items to the sprint backlog. The development team should keep in mind its past performance assessing its capacity for the new sprint, and use this as a guide line of how much "effort" they can complete.

The product backlog items are broken down into tasks by the development team. Tasks on the sprint backlog are never assigned; rather, tasks are signed up for by the team members as needed according to the set priority and the development team member skills. This promotes self-organization of the development team, and developer buy-in.

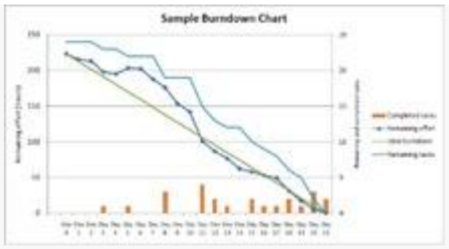
The sprint backlog is the property of the development team, and all included estimates are provided by the development team. Often an accompanying *task board* is used to see and change the state of the tasks of the current sprint, like "to do", "in progress" and "done".

Once a sprint backlog is committed, no additional functionality can be added to the sprint backlog except by the team. Once a sprint has been delivered, the product backlog is analyzed and reprioritized if necessary, and the next set of functionality is selected for the next sprint.

Product increment

The *increment* (or *potentially shippable increment*, PSI) is the sum of all the product backlog items completed during a sprint and all previous sprints. At the end of a sprint, the increment must be complete, according to the scrum team's Definition of Done (DoD), and in a usable condition regardless of whether the product owner decides to actually release it.

Sprint burn-down chart



The sprint burndown chart is a public displayed chart showing remaining work in the sprint backlog. Updated every day, it gives a simple view of the sprint progress. It also provides quick visualizations for reference. During sprint planning the ideal burndown chart is plotted. Then, during the sprint, each member picks up tasks from the sprint backlog and works on them. At the end of the day, they update the remaining hours for tasks to be completed. In such way the actual burndown chart is updated day by day.

The following terms are often used in a scrum process.

Sprint burn-down chart
Daily progress for a sprint over the sprint's length.
Release burn-down chart
Feature level progress of completed product backlog items in the product backlog.
Product backlog (PBL) list
A prioritized list of high-level requirements.
Sprint backlog (SBL) list
A prioritized list of tasks to be completed during the sprint.
Sprint
A time period (typically 1–4 weeks) in which development occurs on a set of backlog items that the team has committed to. Also commonly referred to as a time-box or iteration.
Spike
A time boxed period used to research a concept and/or create a simple prototype. Spikes can either be planned to take place in between sprints or, for larger teams, a spike might be accepted as one of many sprint delivery objectives. Spikes are often introduced before the delivery of large or complex product backlog items in order to secure budget, expand knowledge, and/or produce a proof of concept. The duration and objective(s) of a spike will be agreed between the product owner and development team before the start. Unlike sprint commitments, spikes may or may not deliver tangible, shippable, valuable functionality. For example, the objective of a spike might be to successfully reach a decision on a course of action. The spike is over when the time is up, not necessarily when the objective has been delivered.
Tasks
Work items added to the sprint backlog at the beginning of a sprint and broken down into hours. Each task should not exceed 12 hours (or two days), but it's common for teams to insist that a task take no more than a day to finish.

Definition of Done (DoD)

The exit-criteria to determine whether a product backlog item is complete. In many cases the DoD requires that all regression tests should be successful. The definition of "done" may vary from one scrum team to another, but must be consistent within one team.

Velocity

The total effort a team is capable of in a sprint. The number is derived by evaluating the work (typically in user story points) completed from the last sprint's backlog items. The collection of historical velocity data is a guideline for assisting the team in understanding how much work they can do in a future sprint.

Impediment

Anything that prevents a team member from performing work as efficiently as possible.

Conclusion

Scrum is an agile process most commonly used for product development, especially software development. Scrum is a project management framework that is applicable to any project with aggressive deadlines, complex requirements and a degree of uniqueness. In Scrum, projects move forward via a series of iterations called sprints. Each sprint is typically two to four weeks long.

Experiment No. 4

Objective- Study of Test Driven Development in Agile Methodology.

Theory:

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

Test Driven Development Cycle:

1. Add a test

In test-driven development, each new feature begins with writing a test. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

2. Run all tests and see if the new one fails

This validates that the test harness is working correctly, that the new test does not mistakenly pass without requiring any new code, and that the required feature does not already exist. This step also tests the test itself, in the negative: it rules out the possibility that the new test always passes, and therefore is worthless. The new test should also fail for the expected reason. This step increases the developer's confidence that the unit test is testing the correct constraint, and passes only in intended cases.

3. Write some code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test; no further (and therefore untested) functionality should be predicted nor 'allowed for' at any stage.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code. By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

Development Style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You aren't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods. In Test-Driven Development by Example, Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept such as a design pattern, tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Writing the tests first: The tests should be written before the functionality that is to be tested. This has been claimed to have many benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later. It also ensures that tests for every feature get written. Additionally, writing the tests first leads to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on software quality. When writing feature-first code, there is a tendency by developers and organisations to push the developer on to the next feature, even neglecting testing entirely. The first TDD test might not even compile at first, because the classes and methods it requires may not yet exist. Nevertheless, that first test functions as the beginning of an executable specification.

Each test case fails initially: This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has led to the "test-driven development mantra", which is "red/green/refactor," where red means *fail* and green means *pass*. Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the developer's mental model of the code, boosts confidence and increases productivity.

Keep the unit small

For TDD, a unit is most commonly defined as a class, or a group of related functions often called a module. Keeping units relatively small is claimed to provide critical benefits, including:

- Reduced debugging effort – When test failures are detected, having smaller units aids in tracking down errors.
- Self-documenting tests – Small test cases are easier to read and to understand.

Advanced practices of test-driven development can lead to Acceptance test-driven development (ATDD) and Specification by example where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process. This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy – the acceptance tests – which keeps them continuously focused on what the customer really wants from each user story.

Test structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT & UAT.

Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

Individual best practices

- Separate common set up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test oracle focused on only the results necessary to validate its test.
- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.

Practices to avoid, or "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases.
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A

failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.

- Testing precise execution behavior timing or performance.
- Building “all-knowing oracles.” An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.
- Testing implementation details.
- Slow running tests.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.

Programmers using pure TDD on new ("greenfield") projects reported they only rarely felt the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality is used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, the total code implementation time could be shorter based on a model by Müller and Padberg. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behavior. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Madeyski provided empirical evidence (via a series of laboratory experiments with over 200 developers) regarding the superiority of the TDD practice over the classic Test-Last approach, with respect to the lower coupling between objects (CBO). The mean effect size represents a medium (but close to large) effect on the basis of meta-analysis of the performed experiments which is a substantial finding. It suggests a better modularization (i.e. a more modular design), easier reuse and testing of the developed software products due to the TDD programming practice. Madeyski also measured the effect of the TDD practice on unit tests using branch coverage (BC) and mutation score indicator (MSI), which are indicators of the thoroughness and the fault detection effectiveness of unit tests, respectively. The effect size of TDD on branch coverage was medium in size and therefore is considered substantive effect.

Test-driven development has been adopted outside of software development, in both product and service teams, as test-driven work. Similar to TDD, non-software teams develop quality control checks (usually manual tests rather than automated tests) for each aspect of the work prior to commencing. These QC checks are then used to inform the design and validate the associated outcomes. The six steps of the TDD sequence are applied with minor semantic changes:

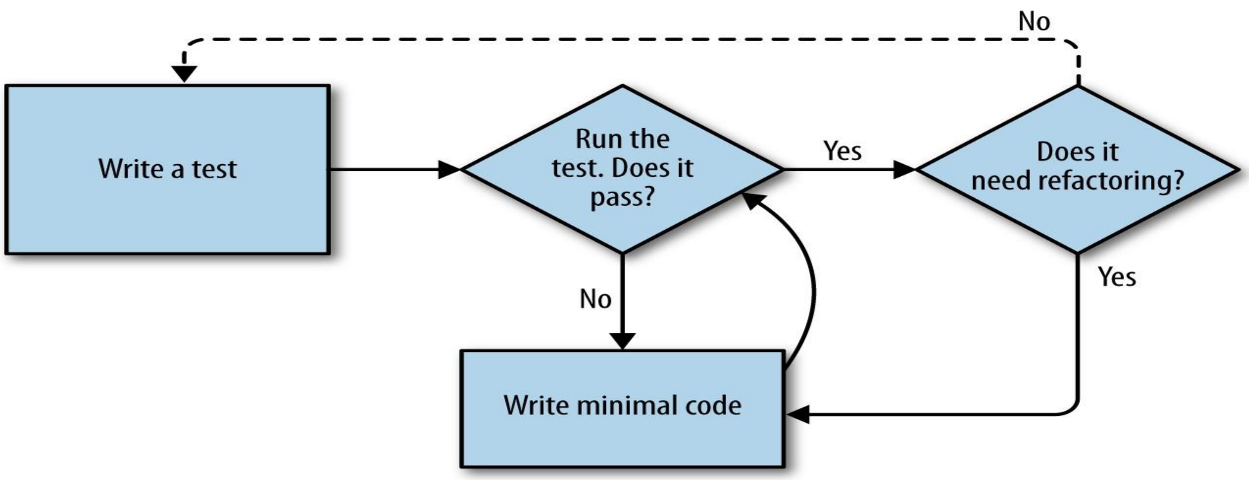


Figure: Test Driven Development

1. "Add a check" replaces "Add a test"
2. "Run all checks" replaces "Run all tests"
3. "Do the work" replaces "Write some code"
4. "Run all checks" replaces "Run tests"
5. "Clean up the work" replaces "Refactor code"
6. "Repeat"

TDD and ATDD

Test-Driven Development is related to, but different from Acceptance Test-Driven Development (ATDD). TDD is primarily a developer's tool to help create well-written unit of code (function, class, or module) that correctly performs a set of operations. ATDD is a communication tool between the customer, developer, and tester to ensure that the requirements are well-defined. TDD requires test automation. ATDD does not, although automation helps with regression testing. Tests used In TDD can often be derived from ATDD tests, since the code units implement some portion of a requirement. ATDD tests should be readable by the customer. TDD tests do not need to be.

Code visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand, normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to private data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access private fields and methods.[28] Alternatively, an inner class can be used to hold the unit tests so they have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This means the released code is not exactly the same as what was unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private methods and data anyway. Some argue that private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Thus it should be sufficient to test any class through its public interface or through its subclass interface, which some languages call the "protected" interface. Others say that crucial aspects of functionality may be implemented in private methods and testing them directly offers advantage of smaller and more direct unit tests.

Conclusion

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

Experiment No 5

Objective- Apply design principles and refactoring to achieve Agility.

Analysis:
1. Understanding different design principles
2. Understanding refactoring

Theory

What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns, packages, components, and classes. It is this level that we will concern ourselves with in this chapter. Our scope in this chapter is quite limited. There is much more to be said about the principles and patterns that are exposed here.

Architecture and Dependencies, what goes wrong with software? The design of many software applications begins as a vital image in the minds of its designers. At this stage it is clean, elegant, and compelling. It has a simple beauty that makes the designers and implementers itch to see it working. Some of these applications manage to maintain this purity of design through the initial development and into the first release. But then something begins to happen. The software starts to rot. At first it isn’t so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain. Eventually the sheer effort required to make even the simplest of changes to the application becomes so high that the engineers and front line managers cry for a redesign project.

Such redesigns rarely succeed. Though the designers start out with good intentions, they find that they are shooting at a moving target. The old system continues to evolve and change, and the new design must keep up. The warts and ulcers accumulate in the new design before it ever makes it to its first release. On that fateful day, usually much later than planned, the morass of problems in the new design may be so bad that the designers are already crying for another redesign.

Symptoms of Rotting Design:

There are four primary symptoms that tell us that our designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. they are: rigidity, fragility, immobility, and viscosity.

Rigidity:

Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multiweek marathon of change in module after module as the engineers chase the thread of the change through the application. When software behaves this way, managers fear to allow engineers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the engineers will be finished. If the managers turn the engineers loose on such problems, they may disappear for long periods of time. The software design begins to take on some characteristics of a roach motel -- engineers check in, but they don't check out. When the manager's fears become so acute that they refuse to allow changes to software, official rigidity sets in. Thus, what starts as a design deficiency, winds up being adverse management policy.

Fragility:

Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way. As the fragility becomes worse, the probability of breakage increases with time, asymptotically approaching 1. Such software is impossible to maintain. Every fix makes it worse, introducing more problems than are solved. Such software causes managers and customers to suspect that the developers have lost control of their software. Distrust reigns, and credibility is lost.

Immobility:

Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

Viscosity:

Viscosity comes in two forms: viscosity of the design, and viscosity of the environment. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing. Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view. If the source code control system requires hours to check in just a few files, then engineers will be tempted to make changes that require as few check-ins as possible, regardless of whether the design is preserved.

These four symptoms are the tell-tale signs of poor architecture. Any application that exhibits them is suffering from a design that is rotting from the inside out. But what causes that rot to take place?

Refactoring

Refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the non-functional attributes of the software. It is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. By continuously improving the design of code, we make it easier and easier to work with. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.

Why refactor?

The purpose of refactoring is to improve the quality, clarity and maintainability of your code. Simple really.

But also, refactoring can be a great lesson in understanding an unfamiliar code base.

Think about it, if you inherit a poorly designed code base that you've not seen before and you now need to either fix a bug or add a new feature, then implementing the code necessary would be a lot easier once you had refactored it to be in a more stable, maintainable and ultimately 'understandable' state.

Otherwise you would be forced to retro fit your new code on top of a poorly designed foundation and that would be the start of a very unhappy relationship.

When should you refactor?

You'll usually find the time you start refactoring the most is when you are fixing bugs or adding new features.

For example, you typically first need to understand the code that has already been written (regardless of whether it was you who wrote it originally or someone else).

The process of refactoring helps you better understand the code, in preparation for modifying it.

But don't fall into the trap of thinking that refactoring is something you set aside time for, or only consider at the start/end of a project. It's not. Refactoring should be done in small chunks throughout the entire life cycle of the project.

As the great Uncle Bob once said:

leave a module in a better state than you found it
...what this suggests is that refactoring is essential to your daily coding process.

Tests

Before we get started, it's important to mention that you should have tests in place when you're refactoring.

You *can* refactor without tests, but realize that without tests to back you up then you can have no confidence in the refactoring you are implementing.

Refactoring can result in substantial changes to the code and architecture but still leave the top layer API the same. So while you're refactoring remember the old adage...

program to an interface, not an implementation.

We want to avoid changing a public API where ever possible (as that's one of the tenets of refactoring).

If you don't have tests then I recommend you write some (now)… don't worry, I'll wait.

Remember, the process of writing tests (even for an application you don't know) will help solidify your understanding and expectations of the code you're about to work on.

Code should be tested regularly while refactoring to ensure you don't break anything. Keep the 'red, green, refactor' feedback loop tight. Tests help confirm if your refactoring has worked or not. Without them you're effectively flying blind.

So although I won't explicitly mention it below when discussing the different refactoring techniques, it is implied that on every change to your code you should really be running the relevant tests to ensure no broken code appears.

Refactoring Techniques

There are many documented refactoring techniques and I do not attempt to cover them all, as this post would end up becoming a book in itself. So I've picked what I feel are the most common and useful refactoring techniques and I try my best to explain them in a short and concise way.

I've put these techniques in order of how you might approach refactoring a piece of code, in a linear, top to bottom order. This is a personal preference and doesn't necessarily represent the best way to refactor.

Final note: with some of the techniques I have provided a basic code example, but to be honest some techniques are so simple they do not need any example. The Extract Method is one such technique that although really useful and important, providing a code example would be a waste of time and space.

Conclusion

The purpose of design principles and refactoring is to improve the quality, clarity and maintainability of your code.

Experiment No. 6

Objective-To study automated build tool.

Theory:

Build automation is the act of scripting or automating a wide variety of tasks that software developers do in their day-to-day activities including things like:

- compiling computer source code into binary code
- packaging binary code
- running automated tests
- deploying to production systems
- creating documentation and/or release notes

History

Historically, developers used build automation to call compilers and linkers from inside a build script versus attempting to make the compiler calls from the command line. It is simple to use the command line to pass a single source module to a compiler and then to a linker to create the final deployable object. However, when attempting to compile and link many source code modules, in a particular order, using the command line process is not a reasonable solution. The make scripting language offered a better alternative. It allowed a build script to be written to call, in a series, the needed compile and link steps to build a software application. GNU Make also offered additional features such as "make depend" which allowed some source code dependency management as well as incremental build processing. This was the beginning of Build Automation. Its primary focus was on automating the calls to the compilers and linkers. As the build process grew more complex, developers began adding pre and post actions around the calls to the compilers such as a check-out from version control to the copying of deployable objects to a test location. The term "build automation" now includes managing the pre and post compile and link activities as well as the compile and link activities.

New breed of tools

In recent years, build management tools have provided even more relief when it comes to automating the build process. Both commercial and open source tools are available to perform more automated build and workflow processing. Some tools focus on automating the pre and post steps around the calling of the build scripts, while others go beyond the pre and post build script processing and also streamline the actual compile and linker calls without much manual scripting. These tools are particularly useful for continuous integration builds where frequent calls to the compile process are required and incremental build processing is needed.

Advanced build automation

Advanced build automation offers remote agent processing for distributed builds and/or distributed processing. The term "distributed builds" means that the actual calls to the compiler and linkers can be served out to multiple locations for improving the speed of the build. This term is often confused with "distributed processing".

Distributed processing means that each step in a process or workflow can be sent to a different machine for execution. For example, a post step to the build may require the execution of multiple test scripts on multiple machines. Distributed processing can send the different test scripts to different machines. Distributed processing is not distributed builds. Distributed processing cannot take a make, ant or maven script, break it up and send it to different machines for compiling and linking.

The distributed build process must have the machine intelligence to understand the source code dependencies in order to send the different compile and link steps to different machines. A build automation tool must be able to manage these dependencies in order to perform distributed builds. Some build tools can discover these relationships programmatically (Rational ClearMake distributed,[1] Electric Cloud ElectricAccelerator), while others depend on user-configured dependencies (Platform LSF lsmake)

Build automation that can sort out source code dependency relationships can also be configured to run the compile and link activities in a parallelized mode. This means that the compiler and linkers can be called in multi-threaded mode using a machine that is configured with more than one core.

Not all build automation tools can perform distributed builds. Most only provide distributed processing support. In addition, most products that do support distributed builds can only handle C or C++. Build automation products that support distributed processing are often based on make and many do not support Maven or Ant.

The deployment task may require configuration of external systems, including middleware. In cloud computing environments the deployment step may even involve creation of virtual servers to deploy build artifacts into.

Advantages

The advantages of build automation to software development projects include

- Improve product quality
- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above.[5]

Types

- **On-Demand automation** such as a user running a script at the command line
- **Scheduled automation** such as a continuous integration server running a nightly build
- **Triggered automation** such as a continuous integration server running a build on every commit to a version control system.

Makefile

One specific form of build automation is the automatic generation of Makefiles. See List of build automation software.

Requirements of a build system

Basic requirements:

1. Frequent or overnight builds to catch problems early.
2. Support for Source Code Dependency Management
3. Incremental build processing
4. Reporting that traces source to binary matching
5. Build acceleration
6. Extraction and reporting on build compile and link usage

Optional requirements:

1. Generate release notes and other documentation such as help pages
2. Build status reporting
3. Test pass or fail reporting
4. Summary of the features added/modified/deleted with each new build

Conclusion

In recent years, build management tools have provided even more relief when it comes to automating the build process. Both commercial and open source tools are available to perform more automated build and workflow processing.

Experiment No. 7

Objective-To study version control tool in Agile.

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

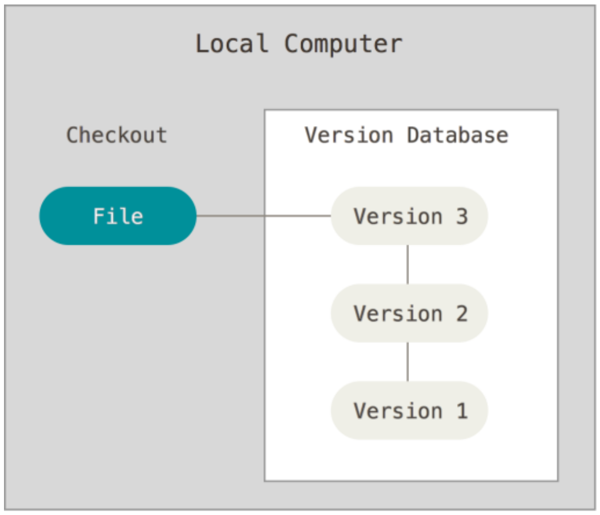
If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Theory:

Local Version Control Systems

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re cle000000000ver). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.



Local version control

One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. Even the popular Mac OS X operating system includes the `rcs` command when you install the Developer Tools. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Conclusion

Version control is a system that records changes to a file or set of files over time so that you can recall specific **versions** later. For the examples in this book you will use software **source** code as the files being **version controlled**, though in reality you can do this with nearly any type of file on a computer.

Experiment No. 8

Objective- To study Continuous Integration tool.

Theory:

Continuous integration (**CI**) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed by Grady Booch in his method,[1] but he did not advocate integrating several times per day. It was adopted as part of extreme programming (XP), which did advocate integrating more than once per day, perhaps as many as tens of times per day. The main aim of CI is to prevent integration problems, referred to as "integration hell" in early descriptions of XP. CI isn't universally accepted as an improvement over frequent integration, so it is important to distinguish between the two as there is disagreement about the virtues of each.

CI was originally intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running all unit tests in the developer's local environment and verifying they all passed before committing to the mainline. This helps avoid one developer's work-in-progress breaking another developer's copy. If necessary, partially complete features can be disabled before committing using feature toggles.

Later elaborations of the concept introduced build servers, which automatically run the unit tests periodically or even after every commit and report the results to the developers. The use of build servers (not necessarily running unit tests) had already been practised by some teams outside the XP community. Nowadays, many organisations have adopted CI without adopting all of XP.

In addition to automated unit tests, organizations using CI typically use a build server to implement continuous processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual QA processes. This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development. This is very similar to the original idea of integrating more frequently to make integration easier, only applied to QA processes.

In the same vein, the practice of continuous delivery further extends CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the actual deployment process very rapid.

Workflow

When embarking on a change, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the source code repository, this copy gradually ceases to reflect the repository code. Not only can the existing code base change, but new code can be added as well as new libraries, and other resources that create dependencies, and potential conflicts.

The longer a branch of code remains checked out, the greater the risk of multiple integration conflicts and failures when the developer branch is reintegrated into the main line. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes.

Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes referred to as "merge hell", or "integration hell",[2] where the time it takes to integrate exceeds the time it took to make their original changes. In a worst-case scenario, developers may have to discard their changes and completely redo the work.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.

A complementary practice to CI is that before submitting work, each programmer must do a complete build and run (and pass) all unit tests. Integration tests are usually run automatically on a CI server when it detects a new commit. All programmers should start the day by updating the project from the repository. That way, they will all stay up to date.

Continuous integration – the practice of frequently integrating one's new or changed code with the existing code repository – should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and correcting them immediately. Normal practice is to trigger these builds by every commit to a repository, rather than a periodically scheduled build. The practicalities of doing this in a multi-developer environment of rapid commits are such that it's usual to trigger a short time after each commit, then to start a build when either this timer expires, or after a rather longer interval since the last build. Many automated tools offer this scheduling automatically.

Another factor is the need for a version control system that supports atomic commits,

i.e. all of a developer's changes may be seen as a single commit operation. There is no point in trying to build from only half of the changed files.

To achieve these objectives, continuous integration relies on the following principles.

Maintain a code repository

This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimised.[5] Instead, it is preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

A single command should have the capability of building the system. Many build-tools, such as make, have existed for many years. Other more recent tools are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Debian DEB, Red Hat RPM or Windows MSI files).

Make the build self-testing

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.

Everyone commits to the baseline every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making. Committing all changes at least once a day (once per feature built) is generally considered part of the definition of Continuous Integration. In addition performing a nightly build is generally recommended. These are lower bounds; the typical frequency is expected to be much higher.

Every commit (to baseline) should be built

The system should build commits to the current working version to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the revision control system for changes, then automatically runs the build process.

Keep the build fast

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they deploy in the production environment, because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the pre-production environment should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances.

Make it easy to get the latest deliverables

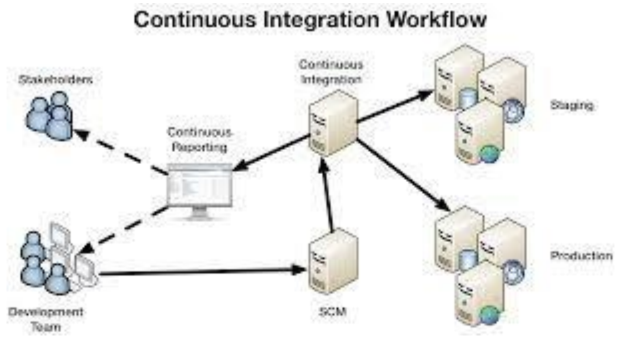
Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier also, in some cases, reduces the amount of work necessary to resolve them.

Everyone can see the results of the latest build

It should be easy to find out whether the build breaks and, if so, who made the relevant change.

Automate deployment

Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is Continuous deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions.



Conclusion

Continuous integration – the practice of frequently integrating one's new or changed code with the existing code repository – should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and correcting them immediately.

Experiment No. 9

Objective- Case Study of XP(Extreme Programming)

Theory:

Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

When Applicable

The general characteristics where XP is appropriate were described by Don Wells on www.extremeprogramming.org:

- Dynamically changing software requirements
- Risks caused by fixed time projects using new technology
- Small, co-located extended development team
- The technology you are using allows for automated unit and functional tests

Due to XP's specificity when it comes to it's full set of software engineering practices, there are several situations where you may not want to fully practice XP. The post When is XP Not Appropriate on the C2 Wiki is probably a good place to start to find examples where you may not want to use XP.
While you can't use the entire XP framework in many situations, that shouldn't stop you from using as many of the practices as possible given your context.

Values

The five values of XP are communication, simplicity, feedback, courage, and respect and are described in more detail below.

Communication

Software development is inherently a team sport that relies on communication to transfer knowledge from one team member to everyone else on the team. XP stresses the importance of the appropriate kind of communication – face to face discussion with the aid of a white board or other drawing mechanism.

Simplicity

Simplicity means "what is the simplest thing that will work?" The purpose of this is to avoid waste and do only absolutely necessary things such as keep the design of the system as simple as possible so that it is easier to maintain, support, and revise. Simplicity also means address only the requirements that you know about; don't try to predict the future.

Feedback

Through constant feedback about their previous efforts, teams can identify areas for improvement and revise their practices. Feedback also supports simple design. Your team builds something, gathers feedback on your design and implementation, and then adjust your product going forward.

Courage

Kent Beck defined courage as "effective action in the face of fear" (Extreme Programming Explained P. 20). This definition shows a preference for action based on other principles so that the results aren't harmful to the team. You need courage to raise organizational issues that reduce your team's effectiveness. You need courage to stop doing something that doesn't work and try something else. You need courage to accept and act on feedback, even when it's difficult to accept.

Respect

The members of your team need to respect each other in order to communicate with each other, provide and accept feedback that honors your relationship, and to work together to identify simple designs and solutions.

Practices

The core of XP is the interconnected set of software development practices listed below. While it is possible to do these practices in isolation, many teams have found some practices reinforce the others and should be done in conjunction to fully eliminate the risks you often face in software development.

The XP Practices have changed a bit since they were initially introduced.The original twelve practices are listed below. If you would like more information about how these practices were originally described, you can visit http://ronjeffries.com/xprog/what-is-extreme-programming/.

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-hour week
- On-site Customer
- Coding Standard

Roles

Although Extreme Programming specifies particular practices for your team to follow, it does not really establish specific roles for the people on your team.

Depending on which source you read, there is either no guidance, or there is a description of how roles typically found in more traditional projects behave on Extreme Programming projects. Here are four most common roles associated with Extreme Programming:

The Customer

The Customer role is responsible for making all of the business decisions regarding the project including:

- What should the system do (What features are included and what do they accomplish)?
- How do we know when the system is done (what are our acceptance criteria)?
- How much do we have to spend (what is the available funding, what is the business case)?
- What should we do next (in what order do we deliver these features)?

The XP Customer is expected to be actively engaged on the project and ideally becomes part of the team.

The XP Customer is assumed to be a single person, however experience has shown that one person cannot adequately provide all of the business related information about a project. Your team needs to make sure that you get a complete picture of the business perspective, but have some means of dealing with conflicts in that information so that you can get clear direction.

The Developer

Because XP does not have much need for role definition, everyone on the team (with the exception of the customer and a couple of secondary roles listed below) is labeled a developer. Developers are responsible for realizing the stories identified by the Customer. Because different projects require a different mix of skills, and because the XP method relies on a cross functional team providing the appropriate mix of skills, the creators of XP felt no need for further role definition.

The Tracker

Some teams may have a tracker as part of their team. This is often one of the developers who spends part of their time each week filling this extra role. The main purpose of this role is to keep track of relevant metrics that the team feels necessary to track their progress and to identify areas for improvement. Key metrics that your team may track include velocity, reasons for changes to velocity, amount of overtime worked, and passing and failing tests.

This is not a required role for your team, and is generally only established if your team determines a true need for keeping track of several metrics.

The Coach

If your team is just getting started applying XP, you may find it helpful to include a Coach on your team. This is usually an outside consultant or someone from elsewhere in your organization who has used XP before and is included in your team to help mentor the other team members on the XP Practices and to help your team maintain your self discipline.

The main value of the coach is that they have gone through it before and can help your team avoid mistakes that most new teams make.

Lifecycle

To describe XP in terms of a lifecycle it is probably most appropriate to revisit the concept of the Weekly Cycle and Quarterly Cycle. First, start off by describing the desired results of the project by having customers define a set of stories. As these stories are being created, the team estimates the size of each story. This size estimate, along with relative benefit as estimated by the customer can provide an indication of relative value which the customer can use to determine priority of the stories.
If the team identifies some stories that they are unable to estimate because they don't understand all of the technical considerations involved, they can introduce a spike to do some focused research on that particular story or a common aspect of multiple stories. Spikes are short, time-boxed time frames set aside for the purposes of doing research on a particular aspect of the project. Spikes can occur before regular iterations start or alongside ongoing iterations.

Next, the entire team gets together to create a release plan that everyone feels is reasonable. This release plan is a first pass at what stories will be delivered in a particular quarter, or release. The stories delivered should be based on what value they provide and considerations about how various stories support each other.Then the team launches into a series of weekly cycles. At the beginning of each weekly cycle, the team (including the customer) gets together to decide which stories will be realized during that week. The team then breaks those stories into tasks to be completed within that week.
At the end of the week, the team and customer review progress to date and the customer can decide whether the project should continue, or if sufficient value has been delivered.

Origins

XP was first used on the Chrysler Comprehensive Compensation (C3) program which was initiated in the mid 90's and switched to an XP project when Kent Beck was brought on to the project to improve the performance of the system. He wound up adding a couple of other folks, including Ron Jeffries to the team and changing the way the team approached development. This project helped to bring the XP methodology into focus and the several books written by people who were on the project helped spread knowledge about and adaptation of this approach.

Primary Contributions

XP’s primary contribution to the software development world is an interdependent collection of engineering practices that teams can use to be more effective and produce higher quality code. Many teams adopting agile start by using a different framework and when they identify the need for more disciplined engineering practices they adopt several if not all of the engineering practices espoused by XP.

Conclusion-contribution of XP is the focus on practice excellence. The method prescribes a small number of absolutely essential practices and encourages teams to perform those practices as good as they possibly can, almost to the extreme. This is where the name comes from. Not because the practices themselves are necessarily radical (although some consider some of them pretty far out) rather that teams continuously focus so intently on continuously improving their ability to perform those few practices.

Experiment No. 10

Objective- Study of Agile approach to Quality Assurance.

Theory- When it comes to software development, quality is everything. Quality Assurance (QA) is a systematic process that ensures product and service excellence. A robust QA team examines the requirements to design, develop, and manufacture reliable products whereby increasing client confidence, company credibility and the ability to thrive in a competitive environment. We want to give you some of our best practice tips for the agile QA process.

The agile QA process begins at the inception of the software development life cycle. From the initial design meeting, through the development phase, to final testing and hardening of the application. This process is repeated in two-week sprints until the project is released.

1. DEFINE AN AGILE QA PROCESS

Most companies have made the shift from the traditional waterfall development methodology to an agile process. Agile testing introduces QA into the project as early as possible to foresee issues, write and execute test cases, and uncover any gaps in requirements. With the project divided into iterative stages, QA engineers are able to add focus to the development process and provide rapid, continuous feedback.
Sprints benefit the client by delivering working software earlier rather than later, anticipating change, providing better estimates in less time, and allowing for course corrections instead of completely derailing the project. The QA team can incorporate lessons learned from previous projects to improve the process for future projects.

2. RISK ANALYSIS

An important aspect of any QA process is risk analysis. Risk analysis is defined as the process of identifying and assessing potential risks and their impact. The process helps organizations avoid and mitigate risks.

It’s highly unlikely for an application to be 100% bug free, but a dedicated QA team should attempt to remove or prevent the most problematic bugs. Understanding all the possible outcomes of a project allows your team to establish preventive measures that reduce the probability of occurrence.

3. TEST EARLY AND TEST OFTEN

The agile model aims to incorporate QA at each stage of the project’s lifecycle to identify issues as early as possible. Within each sprint, QA engineers test and retest the product with each new feature added. This allows them to validate that the new features were implemented as expected and to catch any problems that may have been introduced. Testing early and often leads to the conservation of time and budget.

4. WHITE-BOX VS. BLACK-BOX

Black-box testing assumes no knowledge of how a system does what it does. It only has an understanding of what it should do from the user’s perspective. White-box testing enables the QA engineer to develop a deeper understanding of the system’s internals. Armed with this knowledge, the QA engineer can begin testing much earlier. In an agile QA process, the test engineers need this extra level of system understanding to validate features as soon as they are developed.

White-box testing allows QA teams to anticipate potential error conditions and develop better test scenarios. Knowing how the system works ensures they have tested all possible input scenarios. It can also help identify potential security problems. Perhaps most important: white-box testing encourages close collaboration between development and QA.

5. AUTOMATE WHEN FEASIBLE

Automation can help maximize the effectiveness of your QA staff. Since regression testing can consume a large percentage of the QA team's time, automation provides a way to ensure previous deliverables continue to work while QA engineers focus on testing newly delivered features. Being able to reliably reproduce tests will free up resources for exploratory testing. Automation will give your development team the confidence to make changes to the system with the knowledge that any issues will be identified quickly, and can be fixed before delivery to the QA team.

All that being said, it is important to be cautious of over-automating. Your team should prioritize test cases and then determine which of them should be automated. Situations in which the data might change or where a scenario isn't consistently reproducible may not actually benefit from automation because the results can cause false failures.

Implementing automation costs more up front, but saves money in the long run by increasing efficiency between development and QA teams.

6. KNOW YOUR AUDIENCE

Understanding the target audience will improve the QA process. Tailoring the development and QA process around your users needs will enable your team to build value-driving applications. When you are familiar with who will be using the actual end-product, you can better prioritize the QA process to save time and money.

7. TEAMWORK MAKES THE DREAM WORK

Behind each high-quality product, there is a team of professionals that work incessantly to maintain the high standard of quality upheld by the organization. Although each team working on the project must take responsibility for ensuring quality, the primary responsibility for quality rests with the QA team. The QA team understands what the client needs the system to do and can prove the client's satisfaction with the system. Using the Agile QA process, engineers are the super-sleuths who root out problems and help the team to deliver high-quality products and ensure client confidence, company credibility and successful product delivery.

Conclusion- *"Quality Assurance"* is a set of activities often overlooked in SCRUM teams. The reasons/excuses vary, from tight schedule and the need for a fast delivery, the quality of the test activities or, worse, not understanding the concept at all! Yet, this is not something that should be so easily put on hold, or even postponed until the time is right, as it may never happen. And the consequences could be significant, ranging from time spent fixing (on your expense) after the delivery to losing the trust of your customer. Furthermore, Quality Assurance is not to be confused with *"Quality Control*" - the latter being the final step in the QA process, as it needs a functional, potentially shippable, product to be sure about its effectivene