# **Software Process**

Activities to be done in specific order to **develop a software**

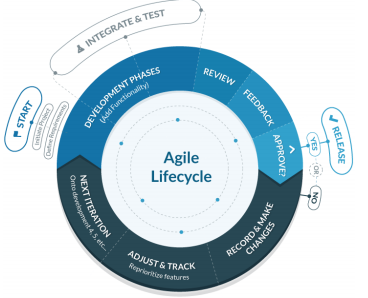
* **Specification requirements**
* **Design and implementation**
* **Validation**
* **Evolution**

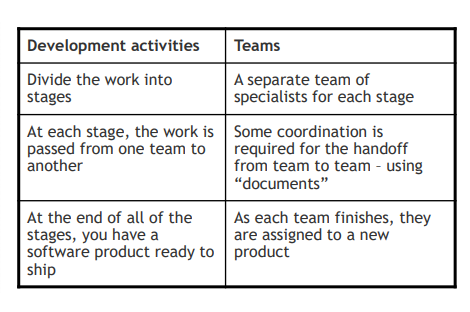
RIGID

* Condition in which incomplete and/or changing requirement is Expensive

# **Software Process Model**

|  |  |  |
| --- | --- | --- |
| **Waterfall Model** | **Spiral Model** | **Agile Model** |
| 1. Requirement analysis and definition 2. System and software design 3. Implementation and unit testing 4. Integration and system testing | * Incremental development based on risk and error | Iterative incremental process for **rapid software development** |
| Intensive planning and documentation nature  Milestones are planned ahead and progress is defined relative to the planned schedule |  | User satisfactory focused  Adaptive to changing requirement  Constant integration and deployment  Independent team work |
| Difficult to implement changes and errors are expensive |  | Difficult to facilitate throughout the phases. |



****

# **Artifacts**

* Items that can represent **work done**
  + Code
  + Requirements specifications

Preserve, communicate, maintain and protect **artifacts**

# Version Control system

Method for preserving versions of artifacts so that we can:

* Revert
* Compare changes
* Who made changes

|  |  |  |
| --- | --- | --- |
| Local VCS | Centralized VCS | Distributed VCS |
| Keep versions in **local computer** | **Single server** containing all the version files and number of client check out files from it | **Developers fully mirror the repository** including the full history. They themselves will be a repository source. |
| **No server settings** necessary | * Easy to collaborate * Easy to maintain admin works | * Easy collaboration * Admin controls |
| Difficult to distribute and Collaborate | Single point of Failure | Strict Server Management is required |

# **GIT**

* Web based **Central Version control**
* Stream of snapshots capturing **file system and changes**
  + Git **doesn’t store unchanged files** but points them to the previous files (pointer)
* Characteristics of GIT
  + Most of the operation is **local**
  + **Built in security** with Hash to store file contents
  + GIT only **adds data**

### **Commit -** Safely stored in **local database (repository)**

**Modified** - The file that has changed since the last commit

### Modified > Staged > Committed



## Working directory (TREE)

* Single checkout of one version of the project
* These files are pulled out off the compressed database and placed on disk for you to modifiy

## Staging Area (index)

* A file stores information about what will go into next commit

## Git directory (repository)

* Metadata and object database
* What is copied when you call **clone**

## Git repository

* Special directory containing project files
* .git stores the history of changes

## Metadata

* Each version should have
  + Unique name
  + Date
  + Author

## .git will keep a record of

* All of the information from past commits
* Repositories (local and remote)
* New files
* Adding and removing files to/from staging area
* Commit changes
* View commit history
* Undo changes

## Git Branching

* **Diverging** from the **main branch of development** and continue to do work **without messing with the main branch**

## Git add

* Will create a **blob** file containing the changes made to the git repo
* Git either creates new blob or points to old blob
  + Git will add only if the item has changed. If not it will point to the last committed blob item.
* It is in the staging area but not stored permanently, it will be deleted by the garbage collector if not committed.

## Git commit

* Create a **commit object** containing metadata and pointing to root project tree

There can be one of two types of files in git repositiory

* Tracked
  + Git knowing about the file
* Untracked
  + The file has yet been added as git object
* When you stage changes, you have **Clean directory**

## Remote repository

* Allows easy distribution, collaboration and communication between team members

## When git clone is called

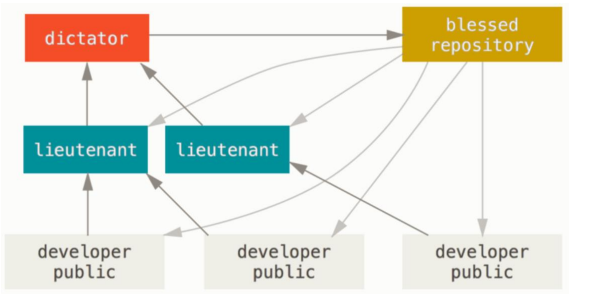
1. Names the repository origin
2. Pulls/downloads all it’s data
3. Creates a pointer to where it’s master branch is and calls it **origin/master** locally
4. Set you own **local master branch** be the same place where **origin master** branch is

## Integration manager workflow (PULL REQUEST)

1. **Project maintainer** pushes to their **public repo**
2. Contributor **clones** that repo and **make changes**
3. Contributor **sends** the **maintainer an email** asking them to **pull changes**
4. The maintainer **adds he contributor’s repository** as a **remote and merges locally**
5. The maintainer pushes **merged** **change** to the **main repository**

## Adv of Integration manager

* Development team can continually work on other features while maintainer assesses the work done
* Contributor does not need to wait for merge before starting another work



### Commit guidelines

* No White space error
* Commit logically separate changeset
* Use quality commit mssages

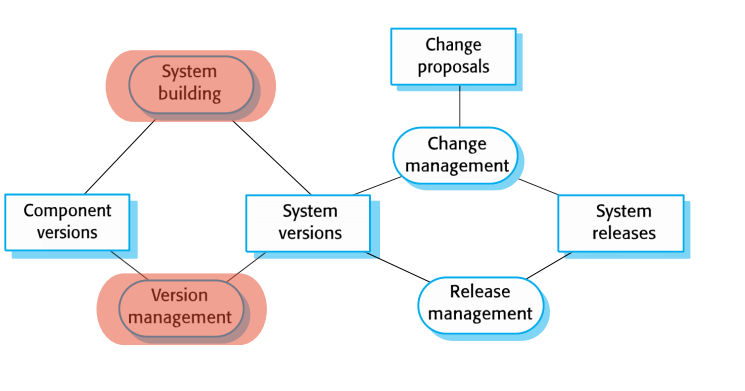
## Git organization

* Owner-member relationship where The owner has the admin authorization and can maintain accessibility to everything

# **Configuration management**

Management of

* Policies, processes and tools for managing changing software system
* Tracking what component versions incorporated into each system version
* Process of managing code-lines and base-lines



## System Building

* Assembling program component, data and libraries, then compiling these to create an executable system

## Version Management

* Keeping track of artifact, component and system versioning
* Ensure changes made to components by different developers do not interfere with eachother

## Change Management

* Keeping track of requests for changes to software from client
* Estimate the cost and impact of changed requirement and decide whether changes are plausible or worth it

## Release management

* Preparing software for external release and keeping track of system versions

## Base Line

* Ecosystem of the application and internal file versions

## Code Line

* Versions of **source code**

# Semantics of versioning

Major.Minor.Patch

### Major

* + Versions when you make **backward-incompatible** changes

### Minor

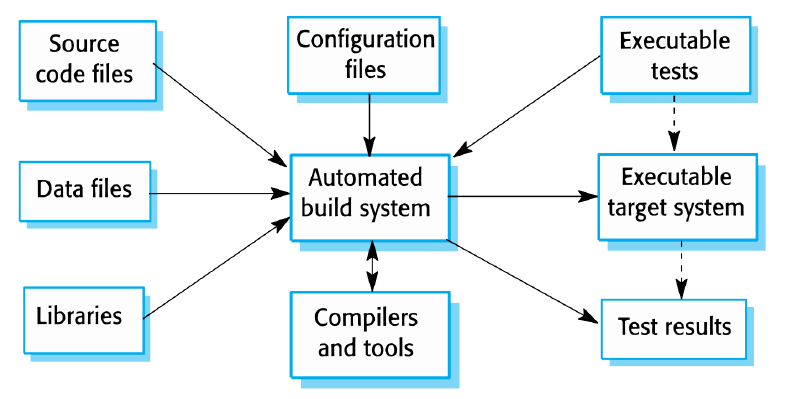
* + Versions when you **add functionalities** in a backward compatible manner

### Patch

* + Versions when you make a backwards compatible bug fix

## 

Firetruck uses Ladder 3.1.0

Ladder 3.1.9 is safe to use, Ladder 3.7.7 is safe to use. Ladder 4.1.1 is NOT because it is not backward compatible.

## **System building**

Process of creating **a complete executable system** by c**ompiling and linking**

# **GRADLE**

## TASKS

* Gradle uses objects called **tasks** to get instructions on how to make a **.jar** file
* Single atomic piece of work for a build
  + Compile a class

## Project

* Composition of several tasks
  + Creation of Jar file

## Build lifecycle

### Initialization

* + Projects are to participate in the build

### Configuration

* + **Task** **objects** are **assembled** into an internal object model called **Directed Acyclic Graph**

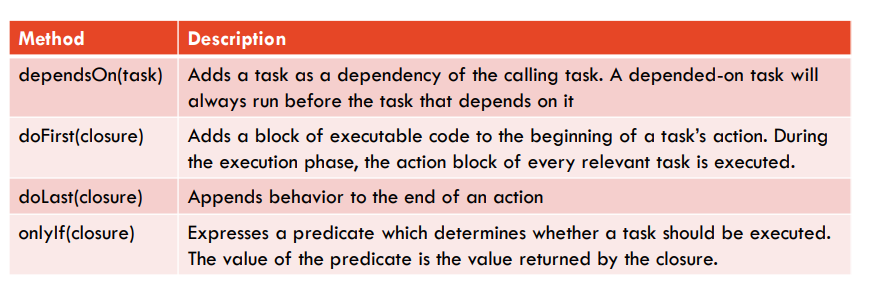
### Execution

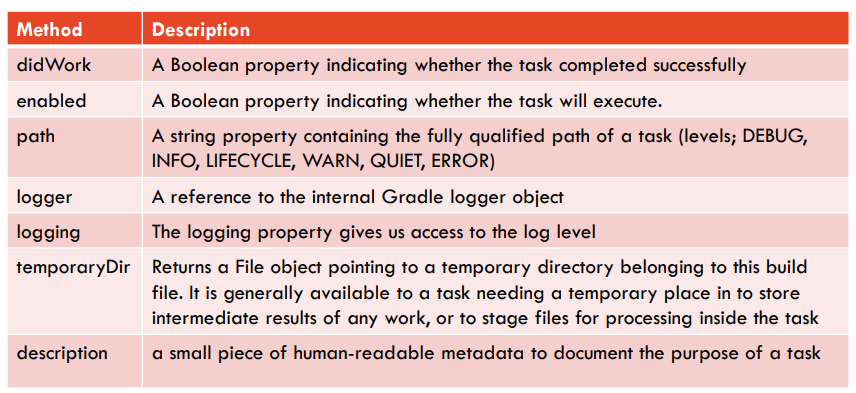
* + Build tasks are executed in the order required by their dependency relationship

## Configuration block

* To **setup variables** and **data structures** needed by the **task action** when it **runs in the build**
  + If the task requires information more than just build tasks, it can be called to configure the variables

## Closure

* Information about configuration and build actions
* Building code.

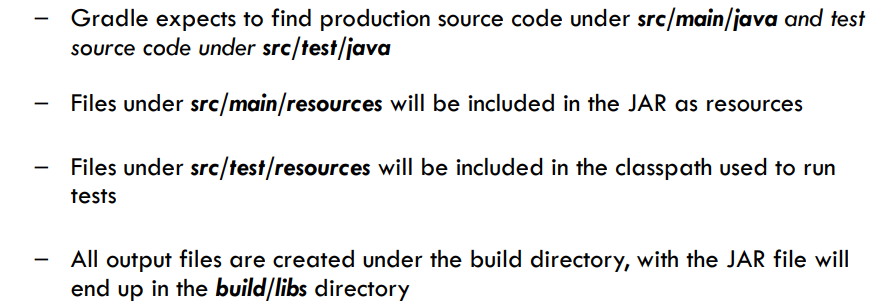


## Plug ins

* Allow adding new task definition
* Change the behavior of existing tasks
* Add new objects
* Create keywords to describe tasks that depart from the standard Gradle categories

## Java plug ins in Gradle

* Specific configurations that helps with Building projects.
* Default tasks area already defined for unit testing and creation of JAR file.



## Gradle clean

* Deletes the build directory, removing all built files

## Gradle assemble

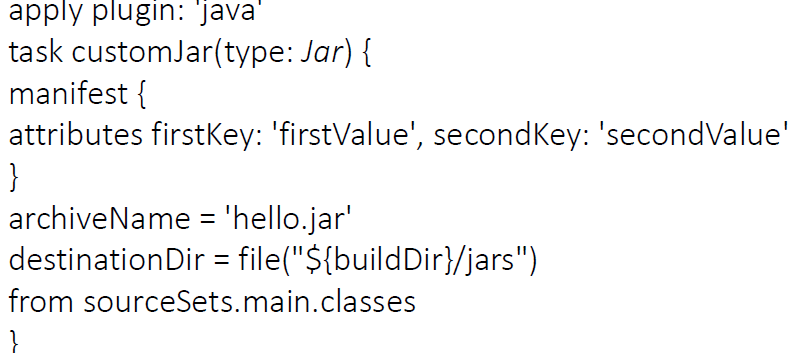
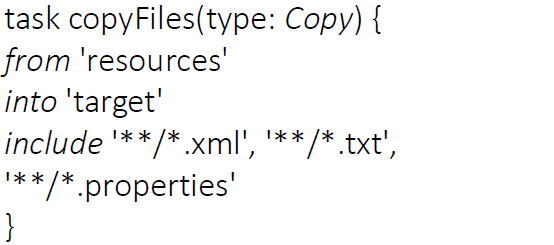
* Compiles and jar your code, but deos not run the tests

## Check

* Compiles and tests your code.

## When Gradle build is called

* Pre-built automatic tasks and projects to build java application. Specified by Java plug in/dependencies
* Class files generated and placed in a directory
* Test report files
* JAR Built using the Project Directory

**Copy Task Build Jar File**

# **Software testing**

## Software Quality

* + The satisfaction of client’s needs
  + Running in a correct behavior
  + Easy to use
  + No errors

## Software Quality Assurance

* Ensuring software under development meets a high quality and holds high integrity

## Software Testing

* Demonstrate that Software meets requirements
* The Tests should be **precise and quantitative** to measure and control the test process
* Decided by the Client
* Find incorrect or undesired behavior
  + System crash
  + Incorrect computation
  + Unnecessary interactions
  + data corruptions

## Who does tests?

* Development team
  + Their own code
  + Each other’s code
* Independent testers
  + To allow fresh perspectives
* End users

## Who would be adequate to test?

* People who has **detailed understanding of the system**
* Application and solution **domain knowledge**
* Knowledge of the **testing technique**

## Software Testing Process

* Select and prepare **suitable test cases**
* Selection of suitable **test technique**
* **Test execution**
* **Trade off analysis**

## Types of Software Errors

* Syntax error
* Run time error
* Timing error
* Logic error

### Functional Testing

* Performs all expected functions properly

### Non-functional testing

* Security, performance and usability

## Software Testing Levels

* Unit/ functional testing
  + Verify functionality of **Software Component** independently
* Integration Testing
  + Verify **interactions/communication** between software components
  + Incremental integration vs Big Bang testing
* System testing
  + Verify functionality and behavior of the **entire software system**
* Acceptance testing
  + Process of verifying **User satisfaction**
* Regression testing
  + Testing previous tests to make sure **new integrations have not introduced malicious interactions**

## Test driven development

* When you write tests before writing code
* Advantageous when you only want to **code what you must code.**
* Changes in requirement would require you to start from scratch

## Types of testing

* Black Box testing
  + No programming and software knowledge
  + Acceptance and system testing
  + Can be done by both **software testers** and **developers**
* White box testing
  + When you code relative to the code
  + Carried by software developer
  + Unit and integration testing
* Defective testing
  + When you purposely try to break an application
* Validation
  + When you are testing to meet the requirement ONLY

Choosing Test Cases

The number of **Tests** should be governed by the number of **edge cases** in each testing field

### Partition Testing / Equivalence partitioning

* Identify groups of inputs with **common characteristics** and **partition on**e from **each group**
  + Choose Edge cases from each partitions

### Guideline based testing

* Using previous experiences to guide what and where to test

## Code Coverage

* Extent to which a source code has been executed by a set of tests
  + Method
  + Statement
  + Branch
  + Condition
  + Loop

## Junit

* Unit testing Framework with Java

|  |  |
| --- | --- |
| **Junit annotations** | |
| Define and denote test methods | |
| **@Test** | Identifies a **method** as a **test** **method** |
| **@Before** | Executed **before** each tests to **prepare** **test** **environment** (variables, reading input data etc.) |
| **@After** | Executed **after** **each** **test** to **clean** up the **test** **environment** |
| **@BeforeClass** | Executed **once** **before** the start of all tests to perform time **intensive activities** |
| **@AfterClass** | Executed **once** **after** all the tests to perform **clean-up activities** |

|  |  |
| --- | --- |
| **Assertion types** | |
| Check expected result with actual result and construct meaningful messages per test | |
| **AssertTrue(Boolean, message)**  **AssertFalse(Boolean, message)** | Test if unit returns **true** or **false** |
| **AssertEquals(Expected, Actual, message)** | Tests whether **two values are the same**  For arrays and objects, the **reference** is checked **NOT** the **content of the arrays** |
| **AssertNull(object, message)**  **AssertNotNull(object, message)** | Tests whether the **object is null or not** |

|  |  |
| --- | --- |
| **Rules** | |
| A way to add or redefine the behavior of each test method in a test class | |
| **Temporary Folder** | Creates files and folders that are deleted when the test is finished |
| **Error Collector** | Let execution of test to continue after first problem is found |
| **Expected Exception** | Allows in test specification of expected exception type and messages |
| @Test(expected = IndexOutOfBoundsException.class) |
| **Time Out** | Applies the same timeout to **all test methods** in a class |
| @Test(timeout = 1000) <- milliseconds |
| **Rule Chain** | Allows ordering of test rules |

# **Continuous Integration, Continuous Delivery And Deployment**

## Continuous integration

* It is a **practice** where the **developer** **integrates** **code** into **shared repository frequently,** and the code is **built and tested**
* To **Ensure** that the **software** is always on a **working state**
  + The **application** is **built** and **tests** are done **automatically** in a **Regression** **testing** manner
* Minimize the duration and effort required by each integration
* Be able to deliver product version suitable for release in any moment

## Continuous integration is a collaboration of

|  |  |
| --- | --- |
| Version Control System | GitHub |
| Automated Build Process | Gradle  configuration management |
| Workflow Integrity and Consistency | Team members  managed by SCRUM Master |
| Configuration of the Build system and testing process | Junit  Gradle Configuration management |
| CI Server to automate the process | Jenkin  Web-Hook to GitHub |

## Maintain Effective Continuous Integration Workflow

## Regular Check-In

* + Small changes are less likely to break and easier to fix

## Create a comprehensive automated test suite

* + Unit testing
  + Component testing
  + Acceptance testing

## Keep the Build and Test process Short

* + So that it doesn’t bottleneck the process of integration

## Developer’s to manage their development workspace

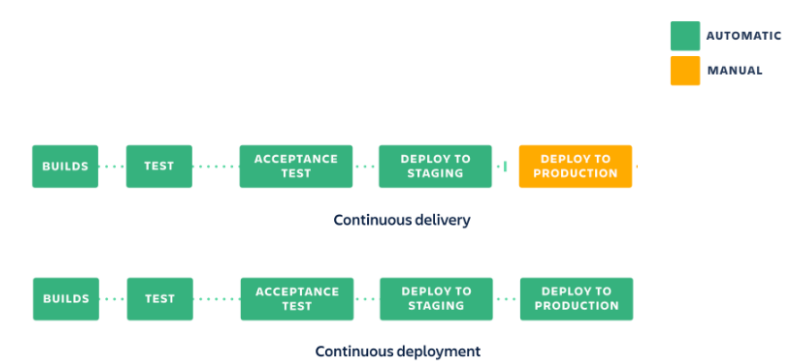
## Use CI Server

* + Report generation
  + Automatic Testing
  + Application Building

## Team Practices

1. **Don’t Check in on a Broken Build**
2. Always run all commit test locally or get your CI server to do it for you
3. **Wait for commit Tests to pass before moving on**
4. Never go home on a Broken Build
5. **Always be prepared to revert to the previous revision**
6. Time box fixing before reverting
7. **Do not comment out failing tests**
8. Take responsibility for all breakage that results from your changes

# Continuous Delivery vs Continuous Deployment

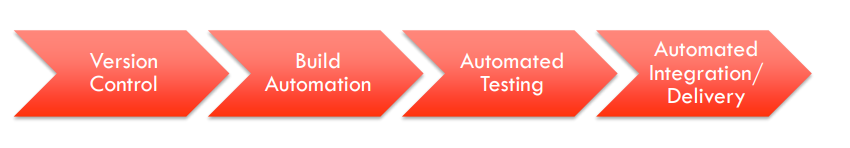


Essential changes is the fact that deployment of product is done **automatically on Continuous Deployment** while **Manually on Continuous Delivery**

Aim of CD is to

* Automate the release process so you can deploy your application at any time
* Continuous feedback

## Jenkins

****

* Automation server to automate tasks relating to
  + **Building**
  + **Testing**
  + **Delivering**
  + **Deployment**
* Running Multiple Steps
  + Retry
    - Retrying steps a number of times
  + Timeout
    - Exiting if a build takes too long
  + Clean up
    - run clean up step or perform post action based on the outcome of the pipeline

REPORT is generated using **Post** section.

If the test **fails** it will be marked as **unstable**

Notifications can be set up when things are failing, unstable or successful.

# Team dynamics

## Team dynamics in Waterfall method vs Agile

### Waterfall

* Teams are separated into clear authority lines
  + Specialist team
  + Project management
* Issue is that
  + There is a single point of failure
  + Inflexible
  + Lack of feedback from the client

### While in Agile

* Individuals and Interactions > Processes and tools
  + People can blindly follow processes and tunnel vision
  + Agile is a dynamic process, therefore all the team member need to be adaptive to any form of changes

## PEOPLE management

* Build project around **Motivated individuals,** give them the **environment and support** that they need and **trust** them to get their job done
* **Face to face** conversation
* Regularly reflects on method to **improve team effectiveness**
* **Businesspeople** and **Development teams** work together **throughout the project**
* **Self-organized team**
* Prioritize **Acceptance testing**

|  |  |  |
| --- | --- | --- |
| Origin of Team dynamics  * Personality * Work style * Role * Culture | Pros and Cons  * Overall Team Performance * Lead to unproductive Conflict   “over the wall”   * Team is divided and marginalized. | Solution Bad Team Dynamics  * Change the office layout * Team development workshop |

## Team Building Activities

### Goal Setting

* + Emphasize the importance of clear objective and individual/team goal

### Interpersonal relationship

* + Giving good supports to each other
  + Good communication and sharing information

### Stand up Meeting

### Roles and responsibility (Managed by Product Owner)

* + Definition of clear roles and responsibilities
  + Identify required role
  + Clarify own responsibility
  + Clarify each team role’s responsibility
  + Summarize roles and responsibilities

## Characteristics of High Performing Agile Team

### Cross functional skillset

### Mutual respect and mutual responsibility

### Sound Engineering practices

### Value and belief of agile practices and principles

### Apply agile practices effectively as individuals and as a team

### Receive continuous training and team monitoring

## Issue Tracking

* Process of listing backlogs and placing priorities, status, severity and complexity
* Allows Centralized overview of development request and their states

## AGILE PRINCIPLES

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

### **Welcome** **changing** **requirements** even late in development.

### Deliver **Working** **software** **frequently**

### **Business people** and **developers** must **work together daily** throughout the project

### **Build** projects around **Motivated** **individuals**, give them the **environment** and **support** they need and **trust** them to get the **job** **done**

### **Working** **Software** is the **primary** **measure** **of** **progress**

### Agile processes promote **sustainable** **development**.

* + The sponsors developers and users should be able to maintain a constant pace indefinitely

### **Continuous** **attention** to technical excellence and good design enhances agility

### The best architectures, requirements and design emerge from **self**-**organizing** **teams**

### At **regular** **intervals**, the **team** **reflects** on how to become more **effective** **then** **tunes** and **adjust** its **behaviours** **accordingly**

### **Individuals and interactions** > processes and tools

### **Working software** > comprehensive documentation

### **Customer collaboration** > contract negotiation

### **Responding to change** > following plans

## Extreme Programming (XP)

* Development and delivery of **very small increments** of functionality
* **Relies on constant code improvement**
* Test Driven Development

# **SCRUM**

Built on three pillars

* Transparency
* Inspection
* Adaptation

## Values

### Commitment

* + Personally commit to get the work done

### Courage

* + Members can go through tough problems

### Focus

* + Focus on getting the work done

### Openness

* + Open about work and challenges with performing work

### Respect

* + Team members respect each other

## Who is scrum for?

* Small team
* Cross functional
* Self-organizing
* Deliver product iteratively

## **TEAM ROLES**

### **Product Owner**

* This is the role responsible for **working with the customers and stakeholders** to determine the **requirements and communicate** the details of these to the team.
* *Note:*Although Scrum strongly emphasises the role of a **product owner** but the person performing this could also be a **business analyst**, **project manager** etc amongst other possible business titles.Manage the Product Backlog
* Manage the Product Backlog
  + Record product backlog items and order it
  + **Optimize the value** of the work the **development team performs**
  + Ensure **transparency** and **clarity** of the **product** **backlog**
  + Ensure the **development** **team** understand **product** **backlog**
  + Be there for team members when they require clearer explainations

### **Scrum Master**

* Keeps the team focused on using SCRUM rules and values properly
* Remove any impediments
* Help mange inter-team member collaboration and interactions
  + Service to **Product Owner** to ensure
    - Mutual **understanding** of **goals, scope and product domain**
    - Effective way for **managing** the **product backlog**
    - SCRUM team **understands** the **need for clear and concise product backlog items**
    - **Arranging** product backlog to **maximise value**
    - **Facilitate SCRUM events**
  + Service to **DEV team**
    - **Coaching** the **DEV team**
    - **Facilitate** SCRUM **event**
    - Remove **impediments** to DEV team’s progress
    - Create **high-value product**

### **Development Team**

* The development team consist of  professionals who do the work of delivering the product. This comprises of
  + Developers
  + Testers
  + Designers
  + Architects etc as applicable to the project or organizational needs.
* Create Sprint backlog
* People who do the work of delivering a potentially releasable product
  + Self-organising
  + Cross functional
  + No sub-teams
  + Whole team is accountable

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scrum Event** | | | | | |
| Sprint is a short, time-boxed period when a scrum team works to complete a set amount of work.  Form of formal inspection and adaptive feedbacks | | | | | |
|  | Sprint Planning | Daily Stand-up | Development Work | Sprint Review | Sprint Retrospective |
| Length | Two 4 hour meetings | 15 minutes | 2-4 Weeks | 4 hours | 1-3 hours |
| Who is included | * Product Owner * Dev Team * Scrum Master | * Product Owner * Scrum Master * Dev Team | * Dev Team | * Product Owner * Scrum Master * Dev Team * Stake Holder | * Scrum master * Dev team * Product Owner(optional) |
| What to do | * **Product Owner, Scrum Master** and **Dev team** select **items to be delivered** * **Estimate** **time** and work required | **Briefings** on   * What they did * What they will be doing * Any challenges   **No problem solving during the meeting**   * Could be done in the **follow up meeting** | Convert sprint backlog into actual application software  Communication with Product owner if over-committed  Must update Sprint backlog for transparency | **Dev team**   * Demonstrates working software   **Stake holder**   * Provides feedback   **Product Owner**   * Updates the Product backlog with change in requirements   **Scrum Master**   * Facilitates the event | **Reflection** on themselves   * What to **improve** * What went **well** * any changes in **velocity?**   **Scrum** **master**   * Adds **improvements** in the **Backlog** |
| Breaking down **defined user** story into **tasks**   * Dev team |
| Outcome | Sprint Backlog |  | Working Software | Revised Product Backlog  Probable item for next iteration | Potential improvements for next sprint |

## **SCRUM ARTIFACTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Scrum Artefacts** | | | |
| Items that can represent **work done**  Set of Features and Sub-features to build the product | | | |
|  | Product Backlog | Sprint Backlog | Burn Down chart |
| Who is responsible | Product Owner | Development Team | Product Owner |
| What is it for | * User Story * Enhancement and fixes from previous sprint * Requirements | Set of items to be delivered by the end of sprint and a plan on how to do it. | Measure the Progress |
| What is included | It is a simple spreadsheet of some items that are   * customer featured   + UX design   + Interaction Scenario   + Use Case * internal tasks   + internal functionality | Comprised of   * To do * In progress * Done | Total work remaining to reach the goal  Managed per sprint by Product Owner |

# **Requirements**

## Plan and document driven software development

### Aimed to have the project be predictable

* Requirement elicitation
* Requirement documentation
* Cost estimation
* Scheduling and monitoring schedule
* Change management for requirements, cost and schedule
* Ensuring implementation matches requirement feature
* Risk analysis and management

## Method to Collect Requirements

### Interview Stakeholders

* + Information discussions and formal questions

### Cooperatively create scenario

* + Initial state, happy/sad paths, concurrent and final state

### Create use Cases

|  |  |
| --- | --- |
| **Functional requirement** | **Non-functional requirement** |
| * What information goes in and out * Interactions between features * UX * functionality | * Performance * Security * usability |

* + User and system interactions to realise functions

**Stakeholders** should **manage requirement** checking’s as well when **creating requirements**

* Validity
* Consistency
* Completeness
* Feasibility

# **Behaviour Driven Development**

* **Conceptual** approach for **specifying application’s behaviour**
* Business value (features) >> Acceptance criteria >> code to deliver
* **Given – When- Then canvas**

## **User Story**

* Short and concise description of how application is expected to be used
* Simple lay language
* **As a- I want to – So that**
* Removes any unnecessary functionalities that users wouldn’t use

## SMART

##### Specific

##### Measurable

##### Achievable

##### Relevant

##### Time-Boxed

* 5 whys to uncover business values/needs

## Acceptance Criteria

* Condition of satisfaction to be written at the back of

## User Interface Sketch

|  |  |
| --- | --- |
| Low-Fidelity UI | High-Fidelity UI |
| * Rough sketch with low tech approach to show how UI looks like | * More defined and descriptive design approach |

|  |  |  |
| --- | --- | --- |
| Scenario | Story Board | User Story |
| Written description of the system interaction from **user’s perspective** | Similar to scenario but visual | Description of a single feature  As a-I want-So that |

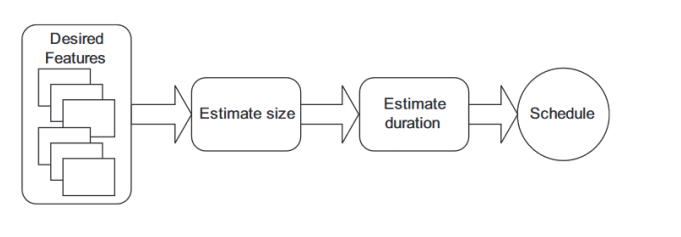
# Scenario Format

* Scenario title
* Given
* When
* Then
* (And Then)

|  |  |
| --- | --- |
| Implicit Scenario | Explicit Scenario |
| * Features that are not specified in the user story * Integration Testing | * Requirements that are explicitly specified in the user story * Acceptance testing |

|  |  |
| --- | --- |
| Imperative Scenario | Declarative Scenario |
| * Tend to have complicated **WHEN** statements and a lot of **AND** steps * Not a good Practice | * Focuses on the **features being described** by using the **step definition** based **domain language for the application** * **Domain Language**   + **Informal** and **specific** to your **application** rather than a generic term |

# **Estimation and Progress Tracking**



## Estimating Size of feature – Story point

* Metric value for the overall size of a user story, feature or piece of work
* # of story points ~ Overall Size of feature
* Scale could be either Fibonacci or sequential doubles
* This is aimed to
  + Measure the user story only
  + No emotional measure
  + Team velocity is considered

|  |  |
| --- | --- |
| Ideal Day | Elapsed Day |
| The amount of time user story will take to develop | The amount of time development will wake **with consideration of interruptions** |

## Estimation Technique

### Expert option

### Analogy: Comparing previous Experiences

### Disaggregation – Breaking stories into smaller tasks

### Planning Poker

# **PLANNING POKER**

* Process of selecting a Story point easier and interactively

1. Select a base feature to make it a 1 and make everyone agree to it.
2. Select each feature and allow the team to choose the item’s Story point
   * No one is able to know each other’s points as of yet
3. Reveal and see the values
   * If there is any outliers or disagreements? let them talk about why they would choose that number
4. Discuss until there is a mutual agreement amongst the team
5. Repeat until the backlog is completely done.

* This is aimed to **avoid individual influences/bias**

|  |  |
| --- | --- |
| Planning poker > Idea day | Ideal Day > Planning Poker |
| Story points are more Constructive and metric | Easier to explain |
| Estimation process is faster | Easier to roughly estimate |
| Avoid Individual influences/bias | Velocity predictions easier |
| Team involvement, team building exercise |  |
| Combination of expert opinion, analogy and disaggregation |  |

## **Velocity**

Measure of **teams rate of progress**

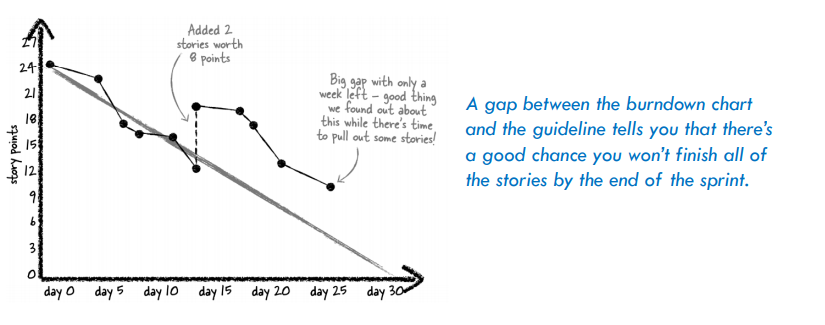
**Velocity = Number of SP done / Time it took to complete**

## Estimating Velocity

* Historical Value
  + Unreliable because each iteration is never the same
* Run an Iteration/sprint
  + Dip your feet in the water to see the rate
* Make a forecast
  + Estimate the available hours per person
  + Randomly select stories and expand them into basic tasks, repeat until number of hours are filled.
  + Convert the velocity determined in the prior step into a range

## Burn Down chart

* X-axis: first and end date of the sprint
* Y-axis: Story points that are Currently left
* Comparison between the number of story points left with ideal progress



# Ethics and Professional Frameworks

|  |  |
| --- | --- |
| Ethics vs Morals | |
| Ethics | Morals |
| Rules of conduct accepted within a social Context | Principles of right and wrong that guide personal behaviour |
| Externally defined | Internally defined |

## Professional Framework

* Primacy of Public interest
* Enhancement of Quality of Life
* Honesty
* Competence
* Professional Development
* Professionalism

|  |  |  |
| --- | --- | --- |
| **General Ethical Principles** | **Professional responsibility** | **Professional Leadership principles** |
| * Society and human well being * Avoid conflicts * Be honest and trustworthy * Be fair * Respect work, privacy * Honour confidentiality | * Quality of work and processes * Quality of review and judgement * Competence | * Quality of work * Training and competence * Public interest decisions to use or retire system |

# Privacy

* Access computing and communication resources only when authorized or when compelled by public good
  + Individuals and organisations have the right to **restrict access tot heir system and data** so long as the restrictions are consistent with other principles

## Intellectual property

* Category of property that includes intangible creations of human intellect
  + **Intangible**
    - Difficult to protect

## IP Types

|  |  |  |
| --- | --- | --- |
| **Copy Right** | **Trade Mark** | **Trade Secret** |
| * Grant a creator of an original work exclusive right * Granted automatically in Australia * Does not protect ideas or method of implementation | * Sign, design or expression which distinguishes products or services * Formal registration and require meeting certain criteria | Process, formula, design, pattern which is **publically unknown** and provide a **competitive advantage** |

## Patent

Form of IP protection which grant inventor the right to exclude others from making, using, selling or importing an invention for a limited time

* Inventor
  + Exclusive rights to the invention
* Public
  + Enrich body of knowledge and innovation
* Criteria
  + New
  + Non-obvious
  + Useful
* Local Protection vs international protection

## **Conditions for Patents**

* New
* Non-obvious
* Useful
* Meet certain formal and substantive standard
* Solution to a technical problem

## **Licensing**

## Open Source

* Software with **source** **code** that anyone can **inspect**, **modify** and **enhance**
* Something people can **modify** and **share** because it is **designed to be publicly accessible**
* **There are conditions** onto how they **use, study, modify and distribute software**
* **Consequent distribution** after modification should also be **open source** 
  + **Copy left**
* Focuses on distribution and collaboration
* Does not need to be free of charge

## Free Software

* Ability to run, copy, change, distribute and improve source code .

**Open source** focuses on **availability** of the **source code** and **ability to modify and share**

* Development methodology

**Free software** focuses on User’s **Freedom** to use program, modify it and share it.

* Social movement

# Copyleft

* Practice of providing freedom to use the software and distribute the application in condition that the licence is preserved

# Open Source Licenses

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Permissions | | | | |
|  | Commercial Use | Distribution | Modification | Private Use | Patent Use |
| MIT License | X | X | X | X |  |
| Apache License 2.0 | X | X | X | X | X |
| Mozila Public License 2.0 | X | X | X | X | X |
| GNU Affero General Public License 3.0 (AGPL) | X | X | X | X | X |
| GNU General Public License 3.0 (GPL) | X | X | X | X | X |
| GNU Lesser General Public License 3.0 (LGPL) | X | X | X | X | X |
| Unlicensed | X | X | X | X | X |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Conditions | | | | | |
|  | Disclose Source | License and copyright notice | Network Use is distribution | Same License | Same License (Library) | State changes |
| MIT License |  | X |  |  |  |  |
| Apache License 2.0 |  | X |  |  |  | X |
| Mozila Public License 2.0 | X | X |  | X |  |  |
| GNU Affero General Public License 3.0 (AGPL) | X | X | X | X |  | X |
| GNU General Public License 3.0 (GPL) | X | X |  | X |  | X |
| GNU Lesser General Public License 3.0 (LGPL) | X | X |  |  | X | X |
| Unlicensed |  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Limitations | | |
|  | Liability | Warrenty | Trademark Use |
| MIT License | X | X |  |
| Apache License 2.0 | X | X | X |
| Mozila Public License 2.0 | X | X | X |
| GNU Affero General Public License 3.0 (AGPL) | X | X |  |
| GNU General Public License 3.0 (GPL) | X | X |  |
| GNU Lesser General Public License 3.0 (LGPL) | X | X |  |
| Unlicensed | X | X |  |