all these feature they will use high level programming languages such as C, C++, Pascal, Java and PHP and Different IDE like Eclipse ,Notepad++ and Idea IntelliJ .
Developers also write unit tests for each component to test the new code that they have written, review each other's code.

Testing

After the coding/developing the software or part of software it goes to testing phase where Quality Analysts (Software testers) test the software by forming various test cases. Experienced testers start to test the system against the requirements. Software testing comprises of Validation and Verification.
Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.
Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.
Testing could be manual or automated, in any case testers prepares the test cases for different sections and level of the code. This cycle is repeated until all requirements have been tested and all the defects have been fixed and the software is ready to be delivered.

Deployment & Maintenance

Once the software has been fully tested and no high priority issues encountered in software, Now it is time to deploy the software to production where customers can use the product. Once a version of the software is released to production, there is usually a maintenance team that look after any post-production issues.
Software maintenance is widely accepted part of SDLC now a days. It comprises for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required it could be client requirement, market need, organization changes, bug fixes, software upgrades etc.

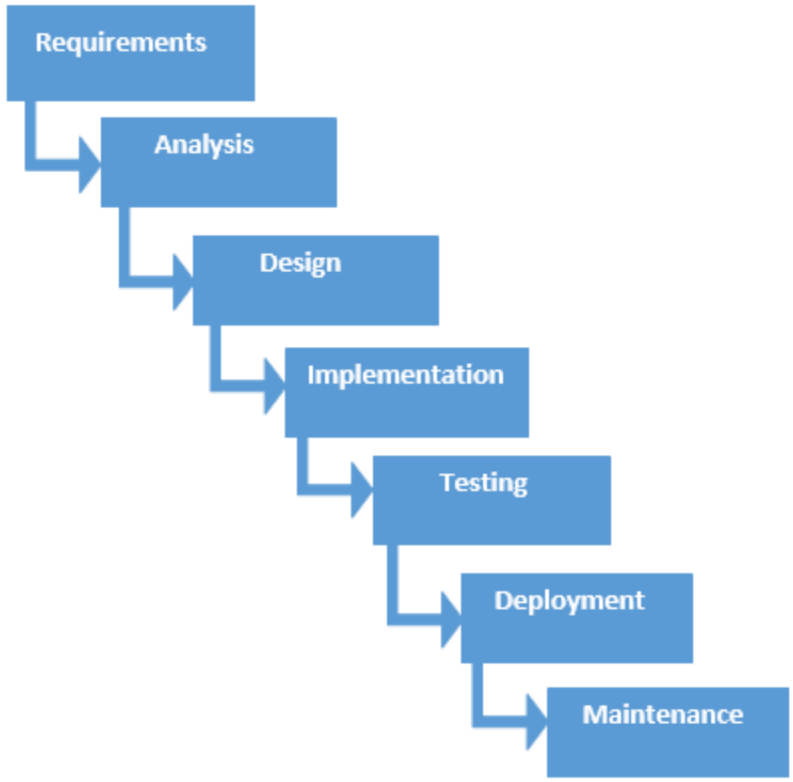
2. Software Development Life Cycle (SDLC)

A software development lifecycle is essentially a series of steps, or phases, that provide a model for the development and lifecycle management of an application or piece of software. Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.
There are various SDLC models like Waterfall, Spiral, iterative Agile and few other.

SDLC- Waterfall model

The Waterfall Model was the first Process Model to be introduced. It is also referred to as a linear-sequential life cycle model. Waterfall Model considered the classic approach to the Software development life cycle, the waterfall model describes a development method that is linear and sequential. It is very simple to understand and use. Waterfall development has distinct goals and target for each phase of development. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

The Waterfall model is the earliest SDLC approach that was used for software development.
The linear sequential flow waterfall model is too much time consuming because each phase should after completion another phase. In this waterfall model, the phases do not overlap.



Waterfall Model - Disadvantages

The main disadvantage of waterfall model is This type development of is too much time consuming and clumsy in test phase. Once an application is in the testing stage and we found something is not according to client requirement or any new requirement .it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.
Few major disadvantages of the Waterfall Model are as follows:

- No working software is produced until end of SDLC.
- Highly vulnerable for the projects where requirements are at a risk of changing. So, risk and uncertainty is high with this process model.
- Clumsy and Time-consuming model for Big/heavy weight Application.
- It is difficult to measure progress within stages.
- Less adaptive model for changing requirements.
- Bigger chances of losing clients and users with until the project developed and release to production.

Agile SDLC Model

Agile development model is a type of **Iterative Incremental model**. This Methods break the product into small incremental releases, each release is built on previous functionality. These builds are form iterations. Each release is thoroughly **tested** to ensure **software quality** is maintained. Each iteration typically lasts from about one to three weeks. Every iteration involves cross functional teams working simultaneously on various areas like:

- Planning
- Requirements Analysis
- Design
- Coding
- Unit Testing and
- Acceptance Testing.

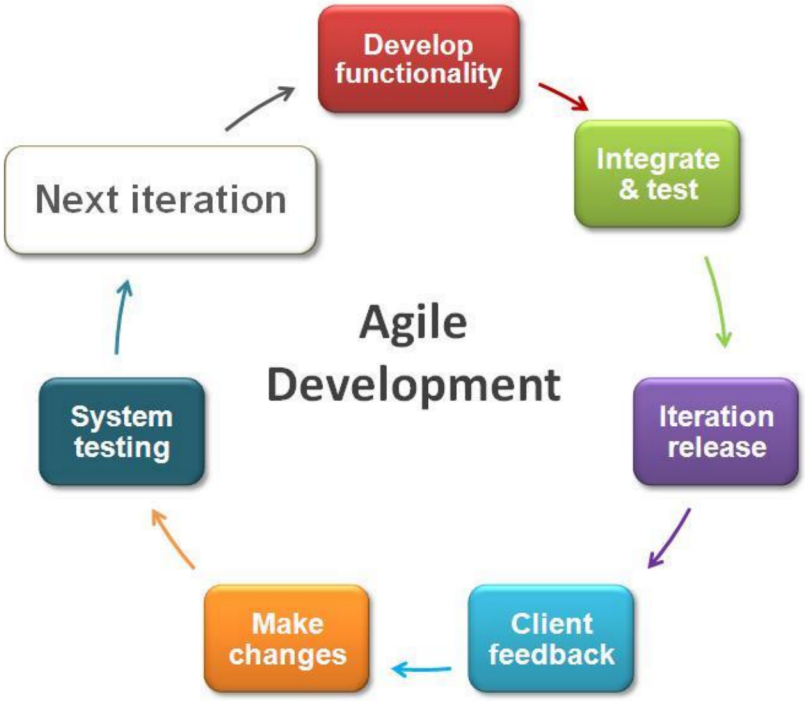
At the end of the iteration, a working product is displayed to the customer. Every software product is actually list of features and if it's a list it can be further divided into small chunks of list. For example, if there are 100 features in a product it can be divided into 5 parts, 20 features in every iteration. Developers and testers will work on 20 features at a time and then move to next iteration once it's developed and tested.

The agile software development emphasizes majorly on following four core values.

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

Here is a graphical illustration of the Agile Model:

Every Iteration is having time duration of 1-2 months based on features



The Agile thought process had started early in the software development and started becoming popular with time due to its flexibility and adaptability.
There are various methods present in agile model one of them is scrum technique given below:

Scrum

SCRUM is an agile development method which concentrates specifically on how to manage tasks within a team-based development environment. Basically, Scrum process is distinguished from other agile processes by specific concepts and practices, divided into the three categories of Roles, Artefacts, and Time Boxes.
Scrum is most often used to manage complex software and product development, using iterative and incremental practices. Scrum significantly increases productivity and reduces time to benefits relative to classic "waterfall" processes. Scrum believes in empowering the development team and advocates working in small teams (say- 7 to 9 members). It consists of three roles, and their responsibilities are explained as follows:

Scrum Master
Master is responsible for setting up the team, sprint meeting and removes obstacles to progress

Product owner
The Product Owner creates product backlog, prioritizes the backlog and is responsible for the delivery of the functionality at each iteration

Scrum Team
Team manages its own work and organizes the work to complete the sprint or cycle

Advantages of Agile model:

- Fast delivery of software (weeks rather than months).
- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Better relations and product management to the client.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are adapted according to requirement

IX. Version Control Systems

Whatever model developers follow to create softwares, there is going to be lot of code that developers write or integrate in their softwares. All this code from different developers in the team has to be merged at a centralised place, which can keep track of all the versions of their code, maintain the code and even revert back in time if anything breaks.

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

1. When to use VCS

Have you ever:

1. Made a change to code, realised it was a mistake and wanted to revert back?
2. Lost code or had a backup that was too old?
3. Had to maintain multiple versions of a product?
4. Wanted to see the difference between two (or more) versions of your code?
5. Wanted to prove that a particular change broke or fixed a piece of code?
6. Wanted to review the history of some code?
7. Wanted to submit a change to someone else's code?
8. Wanted to share your code, or let other people work on your code?
9. Wanted to see how much work is being done, and where, when and by whom?
10. Wanted to experiment with a new feature without interfering with working code?

In these cases, and no doubt others, a version control system should make your life easier.

2. VCS Terminologies

Basic Setup

- **Repository (repo)**: The database storing the files.
- **Server**: The computer storing the repo.
- **Client**: The computer connecting to the repo.
- **Working Set/Working Copy**: Your local directory of files, where you make changes.
- **Trunk/Main**: The primary location for code in the repo. Think of code as a family tree — the trunk is the main line.

Basic Actions

a) **Add/Push**: Put a file into the repo for the first time, i.e. begin tracking it with Version Control.

b) **Revision**: What version a file is on (v1, v2, v3, etc.).

c) **Head**: The latest revision in the repo.

d) **Check out/Pull/Fetch**: Download a file from the repo.

e) **Check in/Push**: Upload a file to the repository (if it has changed). The file gets a new revision number, and people can "check out" the latest one.

f) **Check In Message**: A short message describing what was changed.

g) **Changelog/History**: A list of changes made to a file since it was created.

h) **Update/Sync**: Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.

i) **Revert**: Throw away your local changes and reload the latest version from the repository.

Advanced Actions

- **Branch**: Create a separate copy of a file/folder for private use (bug fixing, testing, etc). Branch is both a verb ("branch the code") and a noun ("Which branch is it in?").
- **Diff/Change/Delta**: Finding the differences between two files. Useful for seeing what changed between revisions.
- **Merge (or patch)**: Apply the changes from one file to another, to bring it up-to-date. For example, you can merge features from one branch into another. (At Microsoft, this was called Reverse Integrate and Forward Integrate)
- **Conflict**: When pending changes to a file contradict each other (both changes cannot be applied).
- **Resolve**: Fixing the changes that contradict each other and checking in the correct version.
- **Locking**: Taking control of a file so nobody else can edit it until you unlock it. Some version control systems use this to avoid conflicts.

3. Famous Version Control Systems.

1. CVS

CVS may very well be where version control systems started. Released initially in 1986, Google still hosts the original Usenet post that announced CVS. CVS is basically the standard here, and is used just about everywhere – however the base for codes is not as feature rich as other solutions such as SVN.

One good thing about CVS is that it is not too difficult to learn. It comes with a simple system that ensures revisions and files are kept updated. Given the other options, CVS may be regarded as an older form of technology, as it has been around for some time, it is still incredibly useful for system admins who want to backup and share files.

2. SVN

SVN, or Subversion as it is sometimes called, is generally the version control system that has the widest adoption. Most forms of open-source projects will use Subversion because many other large products such as Ruby, Python Apache, and more use it too. Google Code even uses SVN as a way of exclusively distributing code.

Because it is so popular, many different clients for Subversion are available. If you use Windows, then Tortoisesvn may be a great browser for editing, viewing and modifying Subversion code bases. If you're using a MAC, however, then Versions could be your ideal client.

3. GIT

Git is considered to be a newer, and faster emerging star when it comes to version control systems. First developed by the creator of Linux kernel, Linus Torvalds, Git has begun to take the community for web development and system administration by storm, offering a largely different form of control. Here, there is no singular centralized code base that the code can be pulled from, and different branches are responsible for hosting different areas of the code. Other version control systems, such as CVS and SVN, use a centralized control, so that only one master copy of software is used.

As a fast and efficient system, many system administrators and open-source projects use Git to power their repositories. However, it is worth noting that Git is not as easy to learn as SVN or CVS is, which means that beginners may need to steer clear if they're not willing to invest time to learn the tool.

4. Mercurial

This is yet another form of version control system, similar to Git. It was designed initially as a source for larger development programs, often outside of the scope of most system admins, independent web developers and designers. However, this doesn't mean that smaller teams and individuals can't use it. Mercurial is a very fast and efficient application. The creators designed the software with performance as the core feature.

Aside from being very scalable, and incredibly fast, Mercurial is a far simpler system to use than things such as Git, which one of the reasons why certain system admins and developers use it.

There aren’t quite many things to learn, and the functions are less complicated, and more comparable to other CVS systems. Mercurial also comes alongside a web-interface and various extensive documentation that can help you to understand it better.

5. Bazaar

Similar to Git and Mercurial, Bazaar is distributed version control system, which also provides a great, friendly user experience. Bazaar is unique that it can be deployed either with a central code base or as a distributed code base. It is the most versatile version control system that supports various different forms of workflow, from centralized to decentralized, and with a number of different variations acknowledged throughout. . One of the greatest features of Bazaar is that you can access a very detailed level of control in its setup. Bazaar can be used to fit in with almost any scenario and this is incredibly useful for most projects and admins because it is so easy to adapt and deal with. It can also be easily embedded into projects that already exist. At the same time, Bazaar boasts a large community that helps with the maintenance of third-party tools and plugins.

4. What is Git

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. A staggering number of software projects rely on Git for version control, including commercial projects as well as open source. Developers who have worked with Git are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

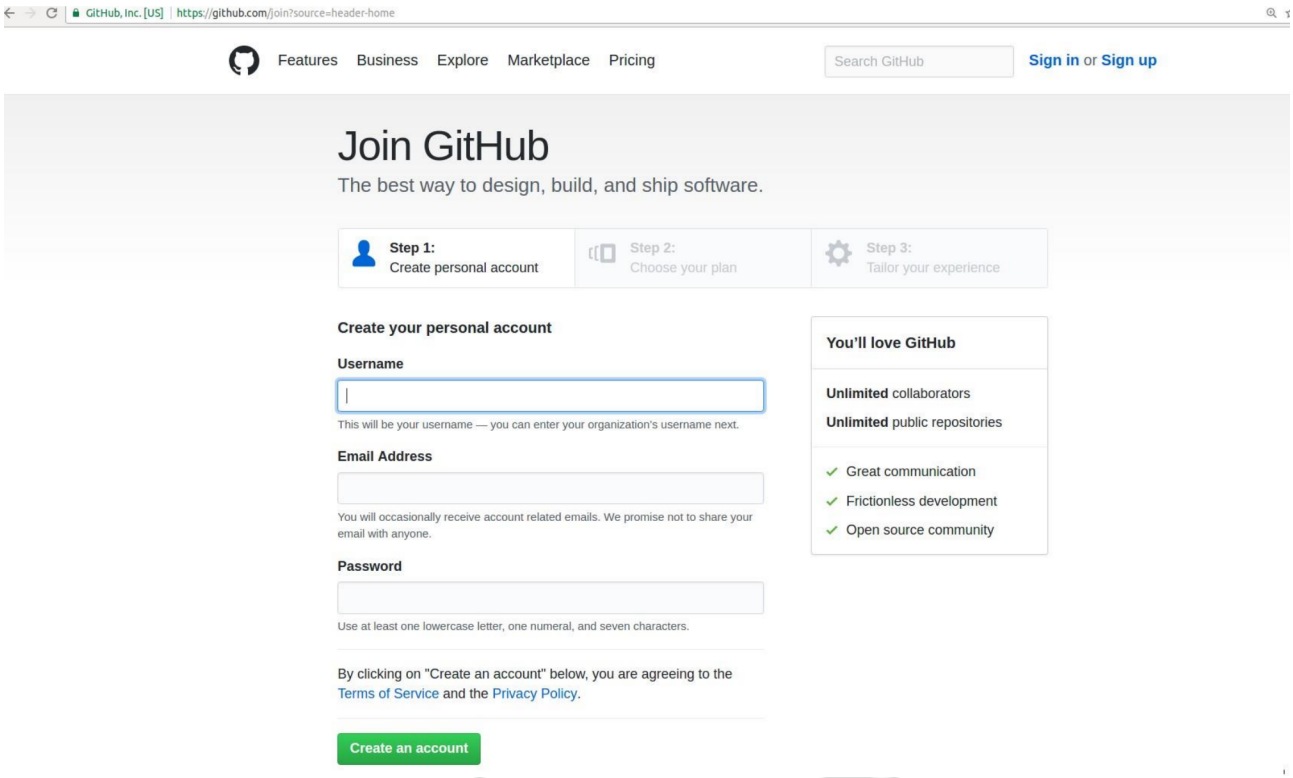
In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

5. Why use git?

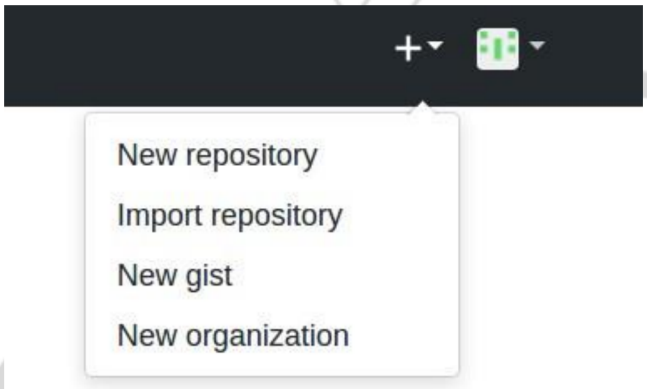
- Its fast
- You don't need access to a server
- Amazingly good at merging simultaneous changes
- Everyone’s using it

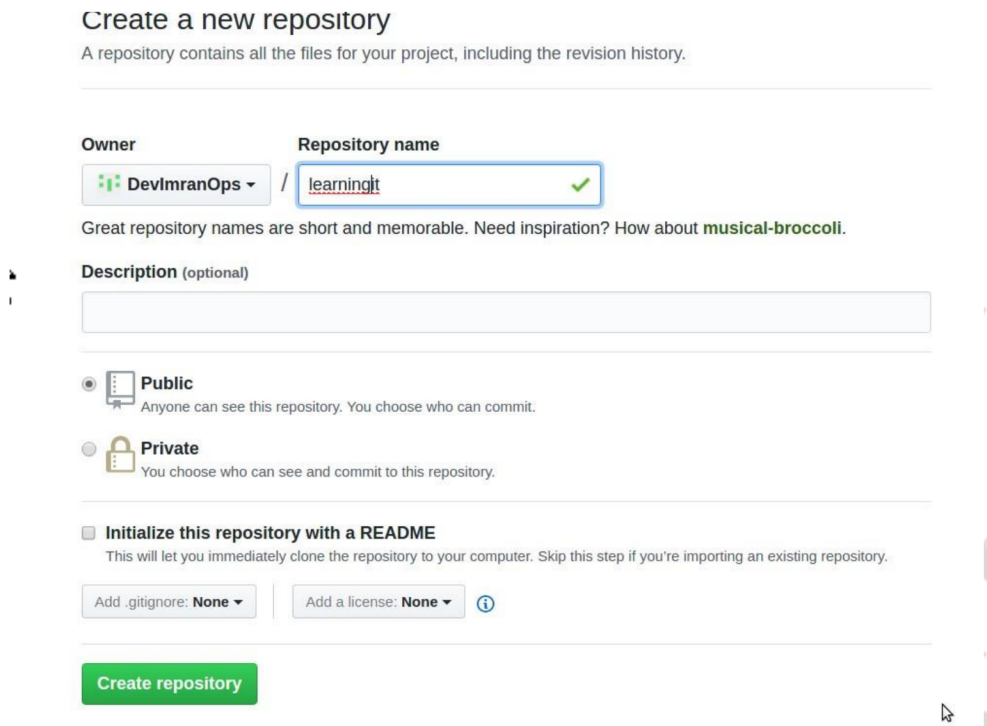
6. Git Quick Setup

1. Sign up for the github account.



2. Create public repository.





3. Copy the URL of the public repo.

4. Install git tool on your system.

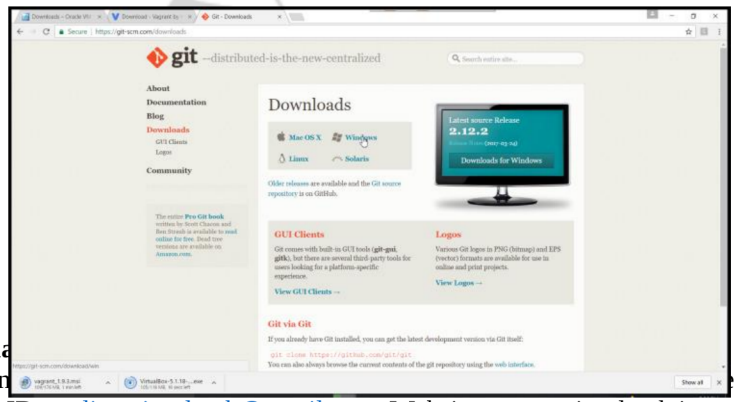
Windows:

-

Install git software

https://git-scm.com/download/win

✓ Go to git scm download page, Select windows.



- Open git installable and follow the Installation wizard, take all the default settings in the wizard.

Linux: -

```
# sudo apt-get install git (Ubuntu) or
# yum install git (Centos) or
```

5. Clone git repository on your local system

```
# git clone <Git Repo URL>
```

6. Create few directories and files in the Repo directory. (Directories should not be empty)

7. Go inside the repo directory

```
# cd <Repo name>
```

8. Update the index using the current content found in the working tree.

```
# git add .
```

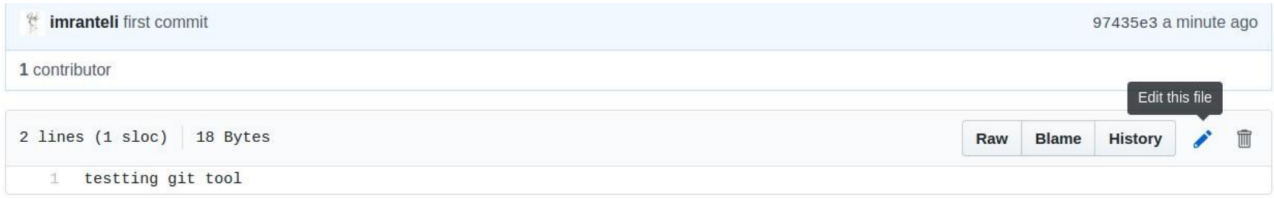
9. Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

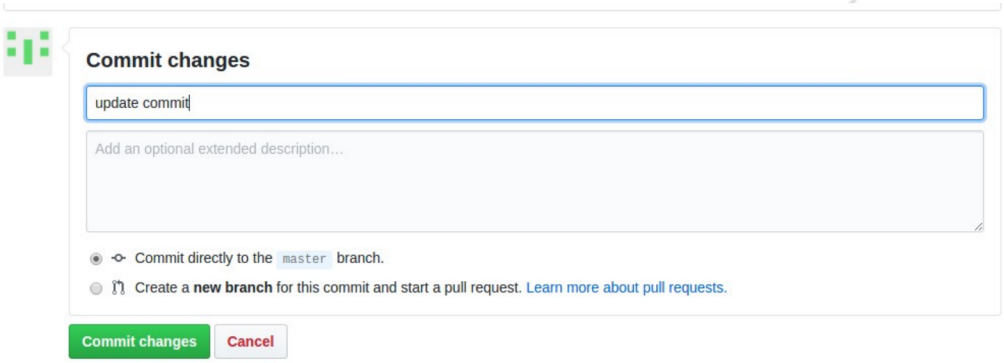
```
# git commit -m "<Put some message>"
```

10. Push the changes to the github.

```
# git push origin master
```

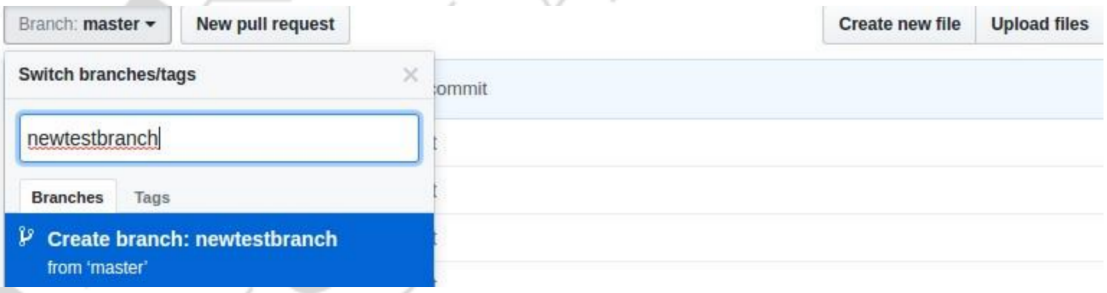
11. Make changes to some files from github UI and pull the changes back to the local repo





git pull

12. Create new branch from master branch and pull the changes again to local repo.



git pull

Switch to new branch

```
# git status (Shows the current branch)
# git checkout <branch name>
# git status
```

13. Make some changes to files and push it back to the latest branch

```
 # git add .
# git commit -m "<Some message>"
# git push origin <branchname>
```

14.Remove/Rename the content and push

```
# git rm <filename>
# git mv <filename>  <newnameoffile>

# git commit -m "<message>"
# git push
```

Explore the UI as much as you can, check histories, commits, changes, content of files, edit file content, create multiple repositories.

Go through the below mentioned git repo and explore it, once you're through with above exercise

https://github.com/wakaleo/gamgit

7. Git in detail

8. Installing Git

Before you start using Git, you must make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

> NOTE
> This book was written using Git version **2.0.0**. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently if you're using an older version. Since Git is quite excellent at preserving backwards compatibility, any version after 2.0 should work just fine.

Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora for example, you can use yum:

```
$ sudo yum install git-all
```

If you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ sudo apt-get install git-all
```

For more options, there are instructions for installing on several different Unix flavors on the Git website, at *http://git-scm.com/download/linux*.

Installing on Mac

There are several ways to install Git on a Mac. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run *git* from the Terminal the very first time. If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. An OSX Git installer is maintained and available for download at the Git website, at *http://git-scm.com/download/mac*.

Figure 1-7. Git OS X Installer.

You can also install it as part of the GitHub for Mac install. Their GUI Git tool has an option to install command line tools as well. You can download that tool from the GitHub for Mac website, at*http://mac.github.com*.

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to *http://git-scm.com/download/win* and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to *https://git-for-windows.github.io/*.

Another easy way to get Git installed is by installing GitHub for Windows. The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub for Windows website, at*http://windows.github.com*.

9. Setting up a repository

This tutorial provides a succinct overview of the most important Git commands. First, the Setting Up a Repository section explains all of the tools you need to start a new version-controlled project. Then, the remaining sections introduce your everyday Git commands.

By the end of this module, you should be able to create a Git repository, record snapshots of your project for safekeeping, and view your project's history.

git init

In git quick start guide we have seen to create git repository on github and then clone/pull that repo to local computer but git clone command. We will see now how to create repo locally and then later how to push it to github.

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the .git directory, an existing project remains unaltered (unlike SVN, Git doesn't require a .git folder in every subdirectory).

Usage

```
$ git init
```

Transform the current directory into a Git repository. This adds a .gitfolder to the current directory and makes it possible to start recording revisions of the project.

```
$ git init <directory>
```

Create an empty Git repository in the specified directory. Running this command will create a new folder called <directory containing nothing but the .git subdirectory.

```
$ git init --bare <directory>
```

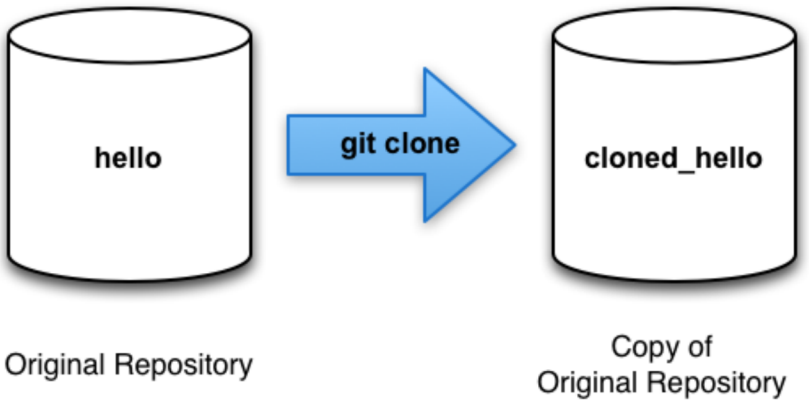
Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag (see discussion below). Conventionally, repositories initialized with

the --bare flag end in .git. For example, the bare version of a repository called my-project should be stored in a directory called my-project.git.

git clone

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

- 

Usage

```
git clone <repo>
```

Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

```
git clone <repo> <directory>
```

Clone the repository located at <repo> into the folder called <directory> on the local machine.

Example

```
$ git clone https://github.com/wakaleo/game-of-life.git
```

git config

The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behaviour of a repository. Several common configuration options are listed below.

Usage

```
$ git config user.name <name>
```

Define the author name to be used for all commits in the current repository. Typically, you'll want to use the --global flag to set configuration options for the current user.

```
$ git config --global user.name <name>
```

Define the author name to be used for all commits by the current user.

```
$ git config --global user.email <email>
```

Define the author email to be used for all commits by the current user.

```
$ git config --global alias.<alias-name> <git-command>
```

Create a shortcut for a Git command.

```
$ git config --system core.editor <editor>
```

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor (e.g., vi).

```
$ git config --global --edit
```

Open the global configuration file in a text editor for manual editing.

Discussion

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

<repo>/.git/config – Repository-specific settings.

~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.

$(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

```
[user]
name = John Smith
email = john@example.com
[alias]
st = status
co = checkout
br = branch
up = rebase
ci = commit
[core]
editor = vim
```

You can manually edit these values to the exact same effect as git config.

Example

The first thing you'll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

```
# Tell Git who you are
$ git config --global user.name "John Smith"
$git config --global user.email john@example.com
# Select your favorite text editor
$ git config --global core.editor vim
# Add some SVN-like aliases
$ git config --global alias.st status
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.up rebase
$ git config --global alias.ci commit
```