

R.M.K

GROUP OF ENGINEERING INSTITUTIONS



R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS



R.M.K
GROUP OF
INSTITUTIONS



Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

24CS102

SOFTWARE DEVELOPMENT PRACTICES

Batch/Year/Semester: 2024-28 / I / I

Created by: Subject Handling Faculties - RMK Group of
Institutions

1. CONTENTS

S. No.	Contents	Page No
1	Contents	5
2	Course Objectives	6
3	Pre Requisites	7
4	Syllabus	8
5	Course outcomes	11
6	CO- PO/PSO Mapping	12
7	Lecture Plan	13
8	Activity based learning	15
9	Lecture Notes	16
10	Assignments	104
11	Part A Questions & Answers	105
12	Part B and Part C Questions	118
13	Supportive online Certification courses	120
14	Real time Applications	121
15	Contents beyond the Syllabus	125
16	Assessment Schedule	128
17	Prescribed Text Books & Reference Books	129
18	Mini Project Suggestions	130

2. Course Objectives

- ✿ To discuss the essence of agile development methods.
- ✿ To set up and create a GitHub repository.
- ✿ To create interactive websites using HTML.
- ✿ To design interactive websites using CSS.
- ✿ To develop dynamic web page using Java script.



3. Prerequisites

SUBJECT CODE: 24CS102

**SUBJECT NAME: SOFTWARE DEVELOPMENT PRACTICES
(Theory Course with Laboratory Component)**


BASIC COMPUTER KNOWLEDGE



**24CS102 – SOFTWARE DEVELOPMENT PRACTICES
(Theory Course with Laboratory Component)**

4. Syllabus

24CS102	SOFTWARE DEVELOPMENT PRACTICES (Common to All Branches)	L	T	P	C
		3	0	3	4.5
OBJECTIVES: The Course will enable learners to: <ul style="list-style-type: none">To discuss the essence of agile development methods.To set up and create a GitHub repository.To create interactive websites using HTMLTo design interactive websites using CSS.To develop dynamic web page using Java script.					
UNIT I	AGILE SOFTWARE DEVELOPMENT AND Git and GitHub	9+9			
<p>Software Engineering Practices – Waterfall Model - Agility – Agile Process – Extreme Programming - Agile Process Models – Adaptive Software Development – Scrum – Dynamic Systems Development Method – Crystal – Feature Driven Development – Lean Software Development – Agile Modeling – Agile Unified Process – Tool set for Agile Process.</p> <p>Introduction to Git –Setting up a Git Repository - Recording Changes to the Repository - Viewing the Commit History - Undoing Things - Working with Remotes -Tagging - Git Aliases - Git Branching - Branches in a Nutshell - Basic Branching and Merging - BranchManagement - Branching Workflows - Remote Branches - Rebasing.</p> <p>Introduction to GitHub – Set up and Configuration - Contribution to Projects, Maintaining a Project – Scripting GitHub.</p>					
List of Exercise/Experiments: <ol style="list-style-type: none">Form a Team, Decide on a project:<ol style="list-style-type: none">Create a repository in GitHub for the team.Choose and follow a Git workflow<ul style="list-style-type: none">Each team member can create a StudentName.txt file with contents about themselves and the team projectEach team member can create a branch, commit the file with a propercommit message and push the branch to remote GitHub repository.Team members can now create a Pull request to merge the branch to master branch or main development branch.The Pull request can have two reviewers, one peer team member and one faculty. Reviewers can give at least one comment for Pull Request updating.Once pull request is reviewed and merged, the master or main development branch will have files created by all team members.Create a web page with at least three links to different web pages. Each of the webpages is to be designed by a team member. Follow Git workflow, pull request and peer reviews.					



R.M.K.
GROUP OF
INSTITUTIONS

UNIT II	HTML	9+9
Introduction – Web Basics – Multitier Application Architecture – Cline-Side Scripting versus Server-side Scripting – HTML5 – Headings – Linking – Images – Special Characters and Horizontal Rules – Lists – Tables – Forms – Internal Linking – meta Elements – Form input Types – input and datalist Elements – Page-Structure Elements.		
List of Exercise/Experiments: 1. Create web pages using the following: <ul style="list-style-type: none"> • Tables and Lists • Image map • Forms and Form elements • Frames 		
UNIT III	CSS	9+9
Inline Styles – Embedded Style Sheets – Conflicting Styles – Linking External Style Sheets – Positioning Elements – Backgrounds – Element Dimensions – Box Model and Text Flow – Media Types and Media Queries – Drop-Down Menus – Text Shadows – Rounded Corners – Colour – Box Shadows – Linear Gradients – Radial Gradients – Multiple Background Images – Image Borders – Animations – Transitions and Transformations – Flexible Box Layout Module – Multicolumn Layout.		
List of Exercise/Experiments: Apply Cascading style sheets for the web pages created.		
UNIT IV	JAVASCRIPT BASICS	9+9
Introduction to Scripting – Obtaining user input – Memory Concepts – Arithmetic – Decision Making: Equality and Relational Operators – JavaScript Control Statements – Functions – Program Modules – Programmer-defined functions – Scope rules – functions – Recursion – Arrays – Declaring and Allocating Arrays – References and Reference Parameters – Passing Arrays to Functions – Multidimensional arrays.		
List of Exercise/Experiments: Form Validation (Date, Email, User name, Password and Number validation) using JavaScript.		
UNIT V	JAVASCRIPT OBJECTS	9+9
Objects – Math, String, and Date, Boolean and Number, document Object – Using JSONto Represent objects – DOM: Objects and Collections – Event Handling.		
List of Exercise/Experiments: Implement Event Handling in the web pages.		
Mini Projects-Develop any one of the following web applications (not limited to one) usingabove technologies. <ol style="list-style-type: none"> Online assessment system Ticket reservation system Online shopping Student management system 		

- e. Student result management system
- f. Library management
- g. Hospital management
- h. Attendance management system
- i. Examination automation system
- j. Web based chat application

TOTAL: 45(L)+45(P)=90 PERIODS

OUTCOMES:

Upon completion of the course, the students will be able to:

CO1: Understand basic software engineering practices effectively.

CO2: Apply version control using Git and GitHub, and manage code repositories proficiently.

CO3: Design web applications using HTML, CSS, and JavaScript.

CO4: Analyze problems and create solutions using CSS for better web page presentation and usability.

CO5: Develop interactive web pages using JavaScript with an event-handling mechanism.

CO6: Apply the technological changes and improve skills continuously.

TEXT BOOKS:

1. Roger S. Pressman, “Software Engineering: A Practitioner’s Approach”, McGrawHill International Edition, Ninth Edition, 2020.
2. Scott Chacon, Ben Straub, “Pro GIT”, Apress Publisher, 3rd Edition, 2014.
3. Deitel and Deitel and Nieto, “Internet and World Wide Web - How to Program”, Pearson, 5th Edition, 2018.

REFERENCES:

1. Roman Pichler, “Agile Product Management with Scrum Creating Products that Customers Love”, Pearson Education, 1 st Edition, 2010.
2. Jeffrey C and Jackson, “Web Technologies A Computer Science Perspective”, Pearson Education, 2011.
3. Stephen Wynnkoop and John Burke, “Running a Perfect Website”, QUE, 2nd Edition, 1999.
4. Chris Bates, “Web Programming – Building Intranet Applications”, 3rd Edition, Wiley Publications, 2009.
5. Gopalan N.P. and Akilandeswari J., “Web Technology”, Second Edition, PrenticeHall of India, 2014.
6. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013382690411003904735_shared/overview
7. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_0130944214274703362099_shared/overview

LIST OF EQUIPMENTS:

1. Systems with either Netbeans or Eclipse
2. Java/JSP/ISP Webserver/Apache
3. Tomcat / MySQL / Dreamweaver or
4. Equivalent/ Eclipse, WAMP/XAMP

5. Course Outcomes

COURSE OUTCOMES		HKL
CO1	Understand basic software engineering practices effectively.	K3
CO2	Apply version control using Git and GitHub, and manage code repositories proficiently.	K6
CO3	Design web applications using HTML, CSS, and JavaScript.	K3
CO4	Analyze problems and create solutions using CSS for better web page presentation and usability.	K6
CO5	Develop interactive web pages using JavaScript with an event-handling mechanism.	K6
CO6	Apply the technological changes and improve skills continuously	K6

Knowledge Level	Description
K6	Evaluation
K5	Synthesis
K4	Analysis
K3	Application
K2	Comprehension
K1	Knowledge

6. CO-PO/PSO Mapping

C O	CO ATTAI NMENT	PROGRAM OUTCOMES												PSO		
		PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PS O 1	PS O 2	PS O 3
CO1	1	3	3	3	1	1	1	-	-	2	1	2	1	2	2	1
CO2	2	3	2	3	1	2	1	-	-	2	1	2	1	-	2	-
CO3	3	3	3	3	1	2	1	-	1	2	1	2	1	2	-	2
CO4	4	3	3	3	1	2	1	-	-	2	1	2	1	-	2	2
CO5	5	3	3	3	1	2	1	-	1	2	1	2	1	2	-	2
CO6	6	3	2	3	1	2	1	-	1	2	1	2	1	-	2	-
CO		3	3	3	1	2	1	-	1	2	1	2	1	2	2	2

7. LECTURE PLAN : UNIT – I

AGILE SOFTWARE DEVELOPMENT AND GIT AND GITHUB

UNIT – I AGILE SOFTWARE DEVELOPMENT AND Git and GitHub9									
S. No	Proposed Lecture Date	Topic	Actual Lecture Date	CO	Highest Cognitive Level	Mode of Delivery	Delivery Resources	LU Outcomes	Remark
1	05.09.2024	Software Engineering Practices, Waterfall Model, Agility, Agility Process	05.09.2024	CO 1	K3	MD1, MD4, & MD5	T1	Software Engineering Practices, Waterfall Model, Agility, Agility Process	
2	09.09.2024	Extreme Programming, Agile Process Models, Adaptive Software Development, Scrum,	09.09.2024		K3	MD1, MD4, & MD5	T1	Extreme Programming, Agile Process Models, Adaptive Software Development, Scrum,	
3	10.09.2024	Dynamic Systems Development Method, Crystal, Feature Driven Development,	10.09.2024		K3	MD1, MD4, & MD5	T1	Dynamic Systems Development Method, Crystal, Feature Driven Development,	
4	11.09.2024	Lean Software Development, Agile Modeling,	11.09.2024		K3	MD1, MD4, & MD5	T1	Lean Software Development, Agile Modeling,	
5	12.09.2024	Agile Unified Process, Tool set for Agile Process,	12.09.2024		K3	MD1, MD4, & MD5	T1	Agile Unified Process, Tool set for Agile Process,	
6	13.09.2024	Introduction to Git, Setting up a Git Repository,	13.09.2024		K6	MD1, MD4, & MD5	T1	Introduction to Git, Setting up a Git Repository,	
7	14.09.2024	Recording Changes to the Repository	14.09.2024		K6	MD1, MD4, & MD5	T1	Recording Changes to the Repository	
8	17.09.2024	Viewing the Commit History,	17.09.2024		K6	MD1, MD4, & MD5	T1	Viewing the Commit History,	
9	18.09.2024	Undoing Things, Working with Remotes, Tagging,	18.09.2024		K6	MD1, MD4, & MD5		Undoing Things, Working with Remotes, Tagging,	
10	19.09.2024	Git Aliases, Git Branching, Branches in a Nutshell,,	19.09.2024		K6	MD1, MD4, & MD5		Git Aliases, Git Branching, Branches in a Nutshell,	
11	20.09.2024	Basic Branching and Merging	20.09.2024		K6	MD1, MD4, & MD5		Basic Branching and Merging	
12	21.09.2024	Branch Management, Branching Workflows	21.09.2024		K6	MD1, MD4, & MD5		Branch Management, Branching Workflows	
13	23.09.2024	Remote Branches, Rebasing, Introduction to GitHub,	23.09.2024		K6	MD1, MD4, & MD5		Remote Branches, Rebasing, Introduction to GitHub,	
14	24.09.2024	Set up and Configuration, Contribution to Projects,	24.09.2024		K6	MD1, MD4, & MD5		Set up and Configuration, Contribution to Projects,	
15	25.09.2024	Maintaining a Project, Scripting GitHub	25.09.2024		K6	MD1, MD4, & MD5	T1	Maintaining a Project, Scripting GitHub	

- **ASSESSMENT COMPONENTS**

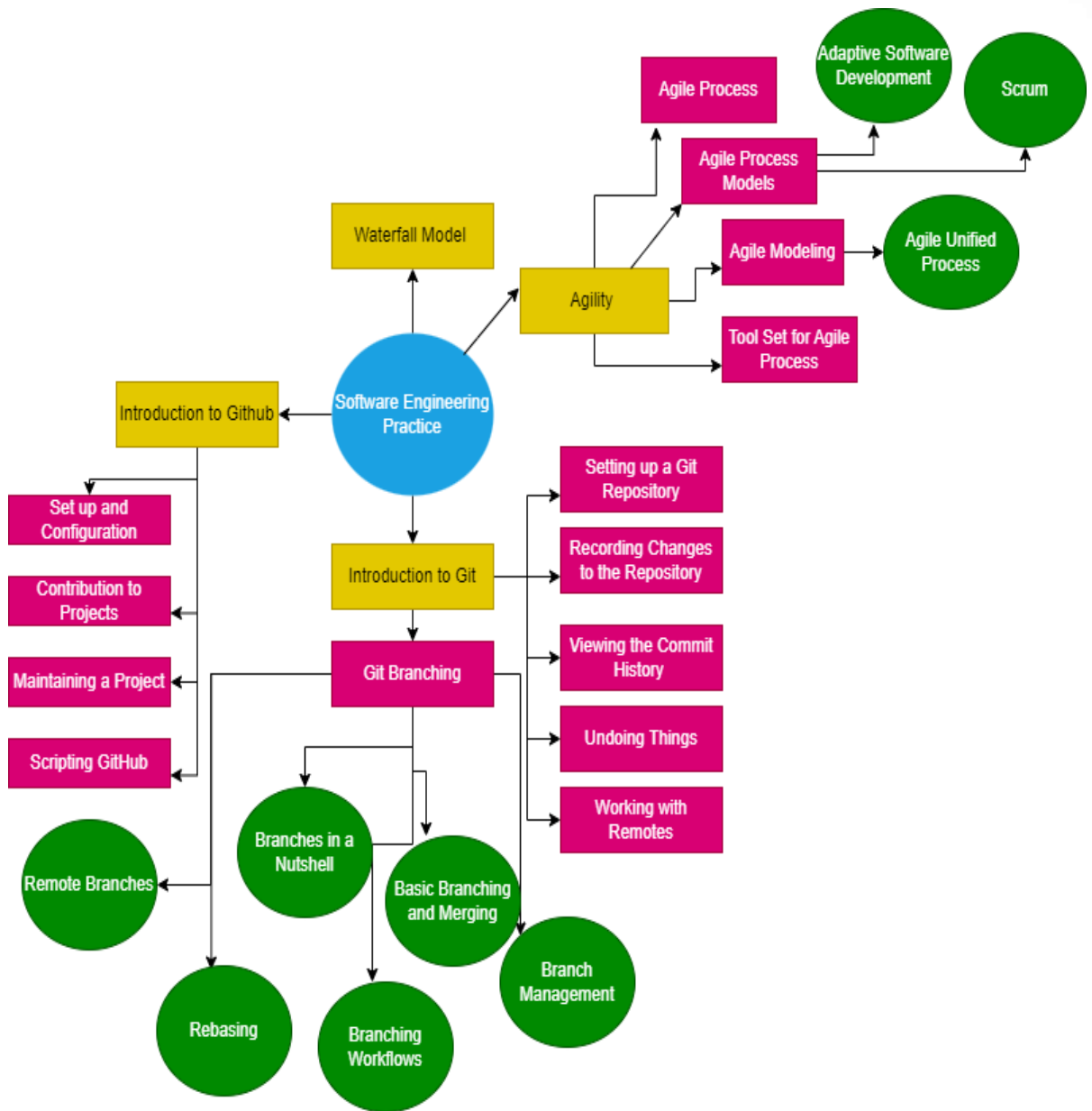
- AC 1. Unit Test
- AC 2. Assignment
- AC 3. Course Seminar
- AC 4. Course Quiz
- AC 5. Case Study
- AC 6. Record Work
- AC 7. Lab / Mini Project
- AC 8. Lab Model Exam
- AC 9. Project Review

- **MODE OF DELEIVERY**

- MD 1. Oral presentation
- MD 2. Tutorial
- MD 3. Seminar
- MD 4. Hands On
- MD 5. Videos
- MD 6. Field Visit

8. ACTIVITY BASED LEARNING : UNIT – I

MIND MAP FOR UNDERSTANDING SOFTWARE ENGINEERING PRACTICES AND GIT AND GITHUB



Lecture Notes-Unit I

Agile Software Development, Git and GitHub

Syllabus

UNIT I	AGILE SOFTWARE DEVELOPMENT, Git AND GitHub	9+9
<p>Software Engineering Practices – Waterfall Model - Agility – Agile Process – Extreme Programming - Agile Process Models – Adaptive Software Development – Scrum – Dynamic Systems Development Method – Crystal – Feature Driven Development – Lean Software Development – Agile Modeling – Agile Unified Process – Tool set for Agile Process.</p> <p>Introduction to Git – Setting up a Git Repository - Recording Changes to the Repository - Viewing the Commit History - Undoing Things - Working with Remotes -Tagging - Git Aliases</p> <p>- Git Branching - Branches in a Nutshell - Basic Branching and Merging - Branch Management - Branching Workflows - Remote Branches - Rebasing.</p> <p>Introduction to GitHub – Set up and Configuration - Contribution to Projects, Maintaining a Project – Scripting GitHub.</p>		

1. SOFTWARE ENGINEERING PRACTICES

A Generic software process model will be composed of a set of activities that establish a framework for software engineering practice.

1. Generic framework activities

- ✿ Communication
- ✿ Planning
- ✿ Modeling
- ✿ Construction
- ✿ deployment

2. ESSENCE OF SOFTWARE ENGINEERING PRACTICE

George Polya outlined the essence of problem solving and the essence of software engineering practice.

- ✿ Understand the problem (communication and analysis).
- ✿ Plan a solution (modeling and software design).
- ✿ Carry out the plan (code generation).
- ✿ Examine the result for accuracy (testing and quality assurance).

Understand the problem

- ✿ Who has a stake in the solution to the problem? That is, who are the stake- holders?
- ✿ What are the unknowns? What data, functions, and features are required to properly solve the problem?
- ✿ Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- ✿ Can the problem be represented graphically? Can an analysis model be created?

Plan the solution

- ✿ Have you seen similar problems before?
- ✿ Has a similar problem been solved? If so, are elements of the solution reusable?
- ✿ Can sub problems be defined? If so, are solutions readily apparent for the sub problems?
- ✿ Can you represent a solution in a manner that leads to effective implementation?
- ✿ Can a design model be created?

Carry out the plan

- ✿ Does the solution conform to the plan? Is source code traceable to the design model?
- ✿ Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result

- ✿ Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- ✿ Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

1.1.3 GENERAL PRINCIPLES OF SOFTWARE ENGINEERING PRACTICES

✿ The First Principle: The Reason It All Exists

“Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

✿ The Second Principle: KISS (Keep It Simple, Stupid!)

All design should be as simple as possible but no simpler.

✿ The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project.

✿ The Fourth Principle: What You Produce, Others Will Consume

Always specify, design, and implement knowing someone else will have to understand what you are doing.

✿ The Fifth Principle: Be Open to the Future

Never design yourself into a corner.

✿ The Sixth Principle: Plan Ahead for Reuse

Reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

✿ The Seventh principle: Think!

Placing clear, complete thought before action almost always produces better results.

1.2 Software Development Life Cycle (SDLC) MODELS

There are various software development life cycle models defined and designed which are followed during the software development process.

These models are also referred as "Software Development Process Models".

Each process model follows a Series of steps unique to its type to ensure success in the process of software development.

Example: Waterfall Model, Iterative Model, Spiral Model, V-Model, Big Bang Model.

WATERFALL MODEL

- It is a SDLC Model. It is also called Linear-sequential life cycle model.
- This classic model divides the life cycle into a set of phases. The output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other. It is also called Linear-sequential life cycle model.
- It is used when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made.

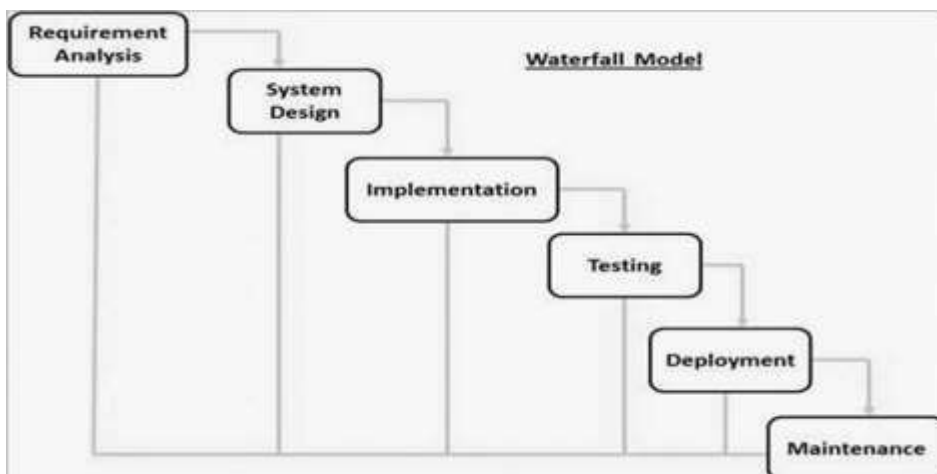


Fig 1.1 Waterfall model

LIFE CYCLE PHASES OF WATERFALL MODEL:

❁ REQUIREMENT GATHERING AND ANALYSIS

All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

❁ SYSTEM DESIGN

The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

❁ IMPLEMENTATION

With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

❁ INTEGRATION AND TESTING

All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

❁ DEPLOYMENT OF SYSTEM

Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.

❁ MAINTENANCE

There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

APPLICATIONS OF WATERFALL MODEL

- ❁ Requirements are very well documented, clear and fixed.
- ❁ Product definition is stable.
- ❁ Technology is understood and is not dynamic.

- ✿ Ample resources with required expertise are available to support the product.
- ✿ The project is short.

ADVANTAGES:

- ✿ Simple and easy to understand and use.
- ✿ Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- ✿ Phases are processed and completed one at a time.
- ✿ Works well for smaller projects where requirements are very well understood.
- ✿ Clearly defined stages.
- ✿ Well understood milestones.
- ✿ Easy to arrange tasks.
- ✿ Process and results are well documented.

DISADVANTAGES:

- ✿ No working software is produced until late during the life cycle.
- ✿ High amounts of risk and uncertainty.
- ✿ Not a good model for complex and object-oriented projects.
- ✿ Poor model for long and ongoing projects.
- ✿ Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- ✿ It is difficult to measure progress within stages.
- ✿ Cannot accommodate changing requirements.
- ✿ Adjusting scope during the life cycle can end a project.
- ✿ Integration is done as a "big-bang" at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

1.3 AGILITY

- ✿ The meaning of the word “agility” is ability to move your body quickly and easily or nimbleness.
- ✿ An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software.
- ✿ An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.
- ✿ Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

AGILITY AND COST OF CHANGE

- ✿ the cost of change increases nonlinearly as a project progresses (solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements.
- ✿ a well-designed agile process “flattens” the cost of change curve (shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact.

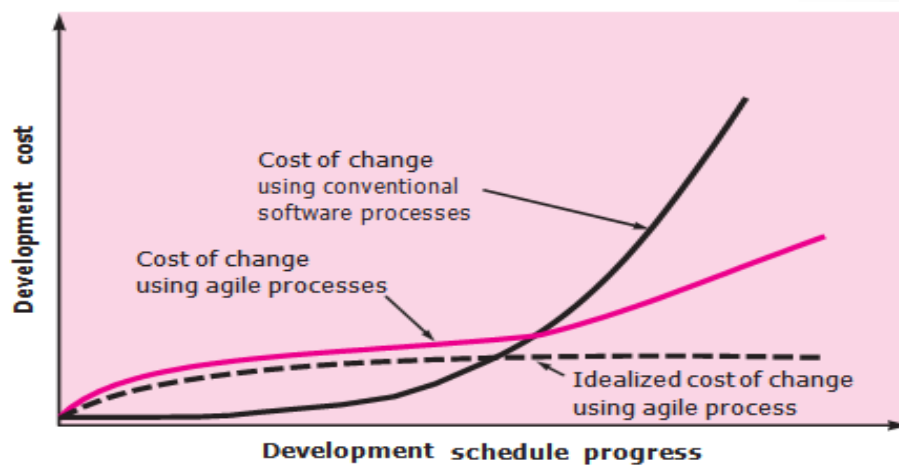


Fig 1.2 Agility and cost of change

1.4 AGILE PROCESS

Any agile process will be characterized by the following number of key assumptions:

- ✿ It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
- ✿ For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- ✿ Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.
- ✿ An incremental development strategy should be instituted. Software increments (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability).
- ✿ This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

AGILITY PRINCIPLES

- ✿ Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- ✿ Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- ✿ Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- ✿ Business people and developers must work together daily throughout the project.
- ✿ Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- ✿ The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- ✿ Working software is the primary measure of progress.
- ✿ Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- ✿ Continuous attention to technical excellence and good design enhances agility.
- ✿ Simplicity—the art of maximizing the amount of work not done—is essential.
- ✿ The best architectures, requirements, and designs emerge from self-organizing teams.
- ✿ At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

AGILE MANIFESTO

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

HUMAN FACTORS

- ✿ If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:
- ✿ **Competence.** In an agile development (as well as software engineering) context, “competence” encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.
- ✿ **Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.
- ✿ **Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate with one another and all other stakeholders.
- ✿ **Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

❁ **Fuzzy problem-solving ability.** Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

❁ **Mutual trust and respect.**

A jelled team exhibits the trust and respect that are necessary to make them “so strongly knit that the whole is greater than the sum of the parts.”

❁ **Self-organization.** In the context of agile development, self-organization implies three things.

❁ the agile team organizes itself for the work to be done.

❁ the team organizes the process to best accommodate its local environment.

❁ the team organizes the work schedule to best achieve delivery of the software increment.

❁ Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

❁ In essence, the team serves as its own management. The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself

1.5 EXTREME PROGRAMMING

- Extreme Programming (XP), the most widely used approach to agile software development.
- a variant of XP, called Industrial XP (IXP) has been proposed. IXP refines XP and targets the agile process specifically for use within large organizations.

XP VALUES

- A set of five values that establish a foundation for all work performed as part of XP
- Communication
- Simplicity
- Feedback courage
- Respect
- **COMMUNICATION:** XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors for communication important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.
- **SIMPLICITY:** XP restricts developers to design only for immediate needs, rather than consider future needs. If the design must be improved, it can be refactored at a later time.
- **FEEDBACK** is derived from three sources the implemented software itself, the customer, and other software team members. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the user stories or use cases that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback.

- Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.
- COURAGE:** An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.
- RESPECT:** As they achieve successful delivery of software increments, the team develops growing respect for the XP process

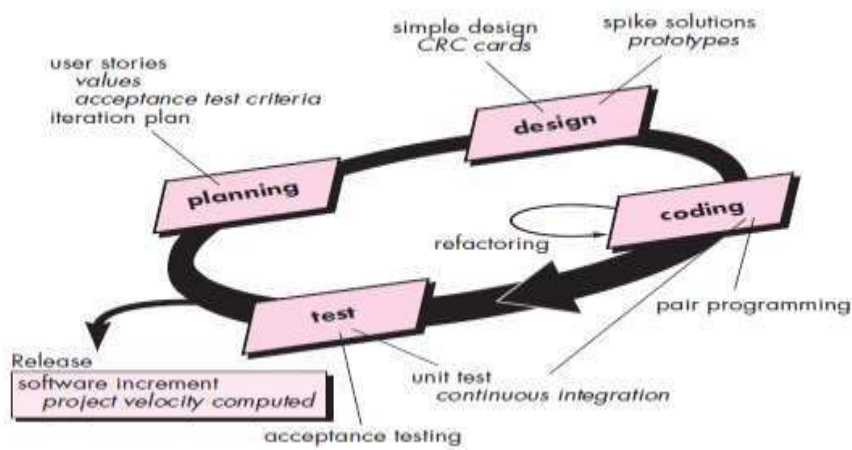


Fig 1.3 Extreme Programming

XP PROCESS

Extreme Programming uses an object-oriented approach.

- Planning.** The planning activity begins with listening a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad. feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" (user stories) that describe required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function.

- ❁ Members of the XP team then assess each story and assign a cost measured in development weeks to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.
- ❁ Once a basic commitment (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways:
 - ❁ all stories will be implemented immediately (within a few weeks).
 - ❁ the stories with highest value will be moved up in the schedule and implemented first.
 - ❁ the riskiest stories will be moved up in the schedule and implemented first.
- ❁ After the first project release (also called a software increment) has been delivered.
- ❁ **project velocity** is the number of customer stories implemented during the first release.
- ❁ Project velocity can then be used to help estimate delivery dates and schedule for subsequent releases.
- ❁ determine whether an over commitment has been made for all stories across the entire development project. If an over commitment occurs, the content of releases is modified or end delivery dates are changed.

DESIGN

- ✿ XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment.
- ✿ The CRC cards are the only design work product produced as part of the XP process.
- ✿ If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a spike solution, the design prototype is implemented and evaluated.
- ✿ The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

CODING

- ✿ After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).
- ✿ Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added.
- ✿ Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.
- ✿ As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team.

TESTING

- As the individual unit tests are organized into a “universal testing suite” Integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry.
- XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

INDUSTRIAL XP

- IXP is an organic evolution of XP. IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.
- Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a readiness assessment. The assessment ascertains
 - (1) an appropriate development environment exists to support IXP
 - (2) the team will be populated by the proper set of stakeholders
 - (3) the organization has a distinct quality program and supports continuous improvement
 - (4) the organizational culture will support the new values of an agile team
 - (5) the broader project community will be populated appropriately.
- Project community.** A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types).

In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization.

Test-driven management. Test-driven management establishes a series of measurable “destinations” and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. The review examines “issues, events, and lessons-learned” across a software increment and/or the entire software release. The intent is to improve the IXP process.

DISADVANTAGES OF XP

- ❖ Requirement volatility
- ❖ Conflicting customer needs
- ❖ Requirements are expressed informally
- ❖ Lack of formal design.

1.6 AGILE PROCESS MODELS

- ❖ Adaptive Software Development (ASD)
- ❖ Scrum
- ❖ Dynamic Systems Development Method (DSDM)
- ❖ Crystal
- ❖ Feature Drive Development (FDD)

- ✿ Lean Software Development (LSD)
- ✿ Agile Modeling (AM)
- ✿ Agile Unified Process (AUP)

1.6.1 ADAPTIVE SOFTWARE DEVELOPMENT (ASD)

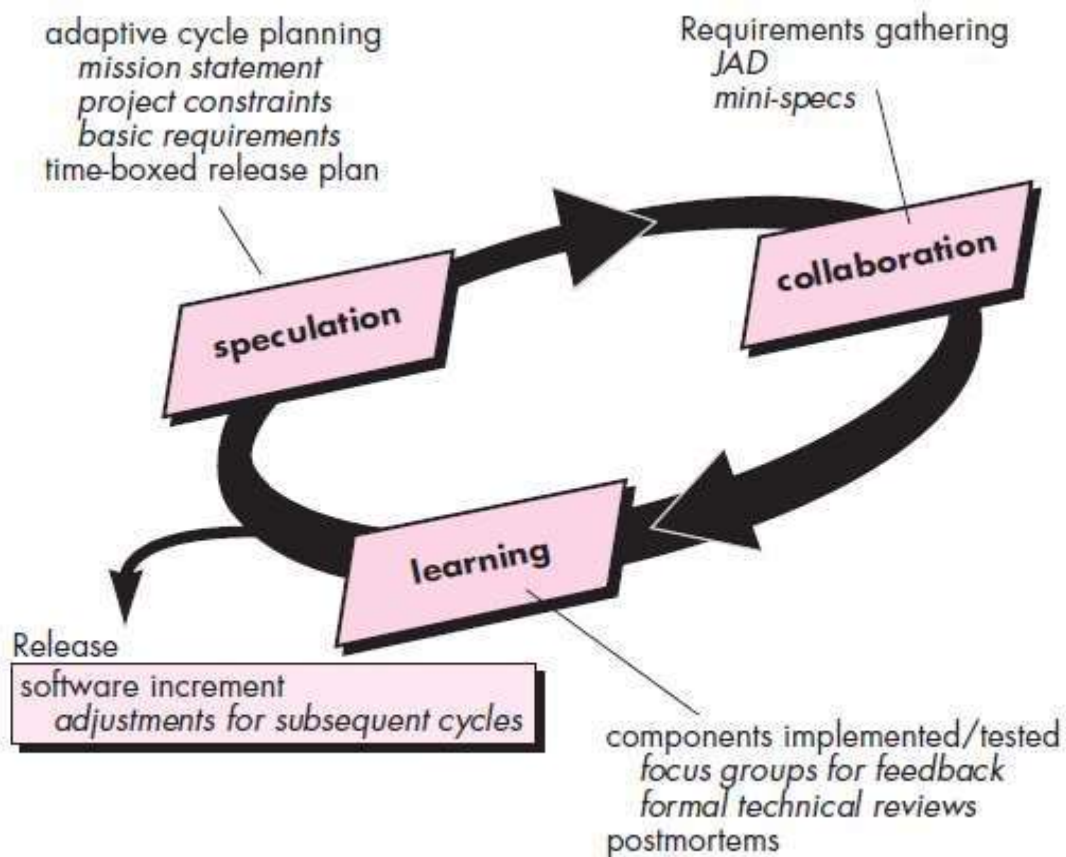


Fig 1.4 Adaptive Software Development

ASD life cycle incorporates three phases -speculation, collaboration, and learning.

SPECULATION

- ✿ During speculation, the project is initiated and adaptive cycle planning is conducted.

- ✿ Adaptive cycle planning uses project initiation information—the customer’s mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements to define the set of release cycles (software increments) that will be required for the project.
- ✿ No matter how complete and farsighted the cycle plan, it will invariably change.
- ✿ Based on information obtained at the completion of the first cycle, the plan is reviewed.
- ✿ and adjusted so that planned work better fits the reality in which an ASD team is working.

COLLABORATION

- ✿ People working together must trust one another to
 - (1) criticize without animosity
 - (2) assist without resentment
 - (3) work as hard as or harder than they do
 - (4) have the skill set to contribute to the work at hand
 - (5) communicate problems or concerns in a way that leads to effective action.

LEARNING

- ✿ learning will help them to improve their level of real understanding. SD teams learn in three ways: focus groups, technical reviews and project postmortems.

MERIT

- ✿ ASD’s overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

1.6.2 SCRUM

- Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.
- Within each framework activity, work tasks occur within a process pattern called a sprint. The work conducted within a sprint is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.

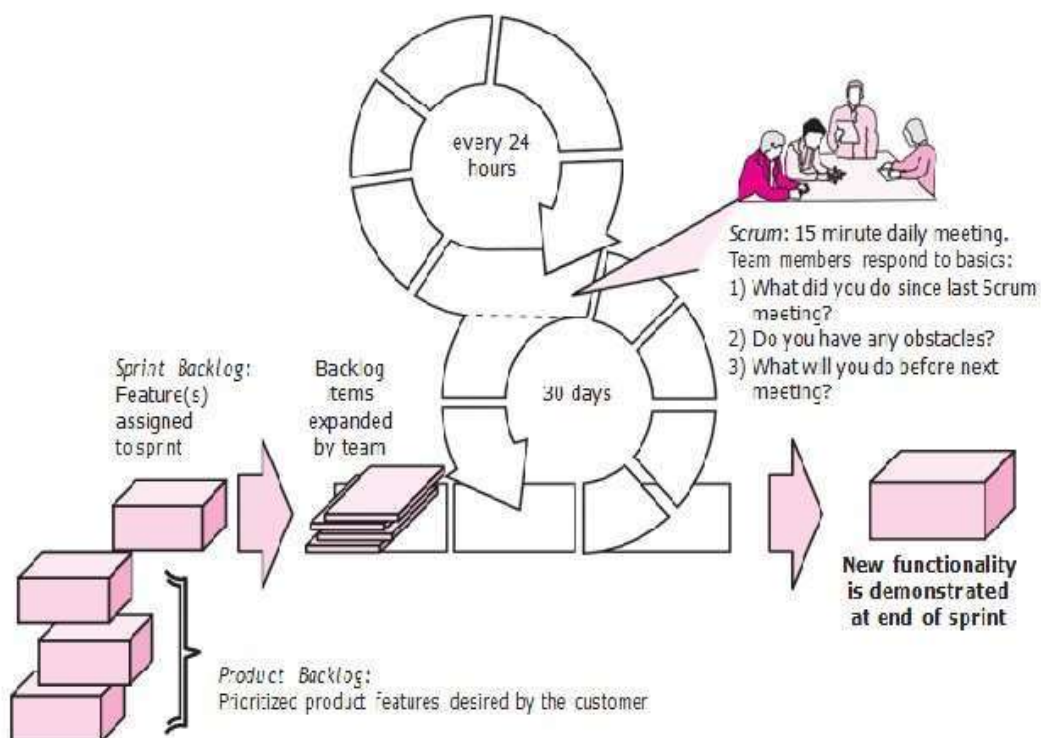


Fig 1.5 Scrum

- Scrum emphasizes the use of a set of software process patterns.
- Each of these process patterns defines a set of development actions:
- Backlog a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

- ✿ Sprints consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days)

Changes (e.g., backlog work items) are not introduced during the sprint. Hence,

- ✿ the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings are short (typically 15 minutes) meetings held daily by the

- ✿ Scrum team.

Three key questions are asked and answered by all team members

- ✿ What did you do since the last team meeting?

- ✿ What obstacles are you encountering?

- ✿ What do you plan to accomplish by the next team meeting?

- ✿ A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible

Demos—deliver the software increment to the customer so that functionality that

- ✿ has been implemented can be demonstrated and evaluated by the customer.

1.6.3 DYNAMIC SYSTEMS DEVELOPMENT METHOD (DSDM)

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

DSDM life cycle defines three different iterative cycles, preceded by two additional life cycle activities:

- ✿ Feasibility study

- ✿ Business study
- ✿ Functional model iteration
- ✿ Design and build iteration
- ✿ Implementation

Feasibility study establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration produces a set of incremental prototypes that demonstrate functionality for the customer. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently

Implementation places the latest software increment (an “operationalized” prototype) into the operational environment.

(1) the increment may not be 100 percent complete or

(2) changes may be requested as the increment is put into place.

In either case, DSDM development work continues by returning to the functional model iteration activity.

1.6.4 CRYSTAL

To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each.

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

1.6.5 FEATURE DRIVEN DEVELOPMENT (FDD)

FDD adopts a philosophy that

- ✿ emphasizes collaboration among people on an FDD team;
- ✿ manages problem and project complexity using feature-based decomposition followed by the integration of software increments
- ✿ communication of technical detail using verbal, graphical, and text-based means.

The emphasis on the definition of features provides the following benefits:

Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.

Features can be organized into a hierarchical business-related grouping.

- ✿ Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- ✿ Because features are small, their design and code representations are easier to inspect effectively.
- ✿ Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

❁ Template for defining a feature:

❁ <action> the <result> <by for of to> a(n) <object>

❁ where an <object> is “a person, place, or thing (including roles, moments in time or

❁ intervals of time, or catalog-entry-like descriptions).”

❁ Examples of features for an e-commerce application might be:

❁ Add the product to shopping cart

❁ Display the technical-specifications of the product

❁ Store the shipping-information for the customer

❁ A feature set groups related features into business-related categories and is defined <action><-ing> a(n) <object>

❁ For example: Making a product sale

❁ The FDD approach defines five “collaborating” framework activities (in FDD these are called “processes”) as shown in Figure

❁ FDD defines six milestones during the design and implementation of a feature:

❁ “design walkthrough, design, design inspection, code,code inspection, promote to build”

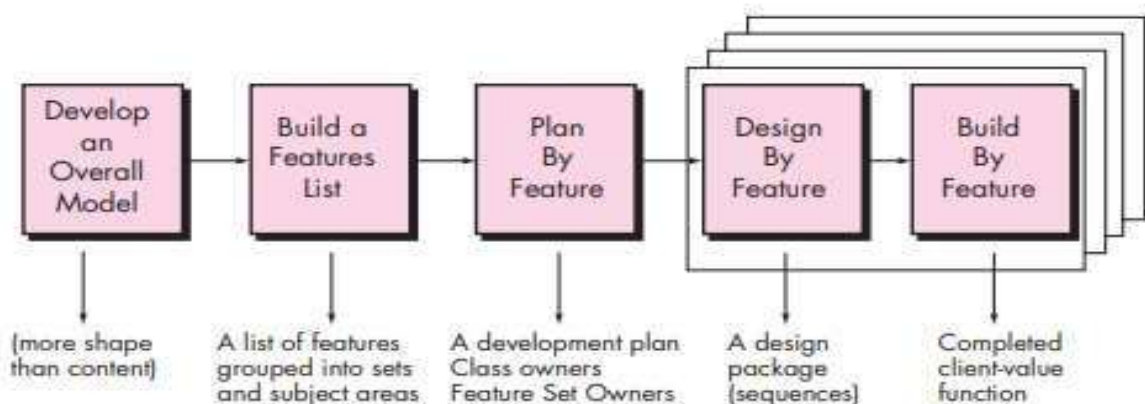


Fig 1.6 Feature Driven Development

1.6.6 LEAN SOFTWARE DEVELOPMENT (LSD)

The lean principles that inspire the LSD process can be summarized as eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole.

For example, eliminate waste within the context of an agile software project can be interpreted

- (1) adding no extraneous features or functions
- (2) assessing the cost and schedule impact of any newly requested requirement
- (3) removing any superfluous process steps
- (4) establishing mechanisms to improve the way team members find information
- (5) ensuring the testing finds as many errors as possible
- (6) reducing the time required to request and get a decision that affects the software or the process that is applied to create it
- (7) streamlining the manner in which information is transmitted to all stakeholders involved in the process.

1.7 AGILE MODELING (AM)

Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner.

Although AM suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are Model with a purpose.

A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model.

Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

Use multiple models. There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

- ✿ As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur This represents work that slows the team down.
- ✿ Content is more important than representation. Modeling should impart information to its intended audience.
- ✿ A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.
- ✿ Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.
- ✿ Adapt locally. The modeling approach should be adapted to the needs of the agile team.

1.8 AGILE UNIFIED PROCESS (AUP)

The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building computer-based systems.

Each AUP iteration addresses the following activities:

- ✿ **Modeling.** UML representations of the business and problem domains are created. However, to stay agile, these models should be “just barely good enough” to allow the team to proceed.
- ✿ **Implementation** Models are translated into source code.
- ✿ **Testing** Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- ✿ **Deployment.** Like the generic process activity deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- ✿ **Configuration and project management.** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

- ❁ **Environment management.** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

1.9 A TOOL SET FOR THE AGILE PROCESS

- ❁ Agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage.
- ❁ Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops.”
- ❁ Cockburn argues that “tools” that address these issues are critical success factors for agility. For example, a hiring “tool” might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The “fit” can be assessed immediately.
- ❁ Collaborative and communication “tools” are generally low tech and incorporate any mechanism (“physical proximity, whiteboards, poster sheets, index cards, and sticky notes”)
- ❁ Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by “information radiators” (e.g., a flat panel display that presents the overall status of different components of an increment).
- ❁ Project management tools deemphasize the Gantt chart and replace it with earned value charts or “graphs of tests created versus passed .
- ❁ Other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions(e.g., colocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)”

- ❁ Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by “information radiators” (e.g., a flat panel display that presents the overall status of different components of an increment).
- ❁ Project management tools deemphasize the Gantt chart and replace it with earned value charts or “graphs of tests created versus passed.
- ❁ other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)”.

1.10 INTRODUCTION TO GIT

VERSION CONTROL:

- ❁ Version control is a system that records changes to a file or set of files over time so that can recall specific versions later. It allows to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see which is 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, get all this for very little overhead.

LOCAL VERSION CONTROL SYSTEMS



Fig 1.7 Local Version Control Systems

CENTRALIZED VERSION CONTROL SYSTEMS

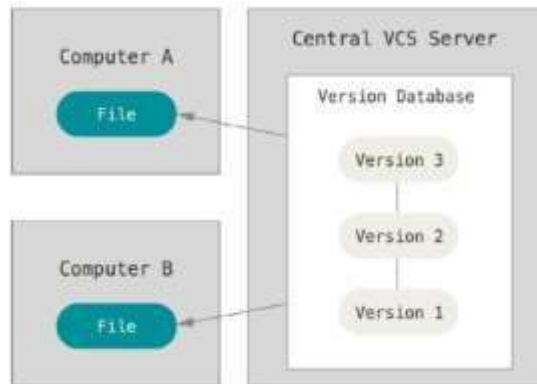


Fig 1.8 Centralized Version Control Systems

DISTRIBUTED VERSION CONTROL SYSTEMS

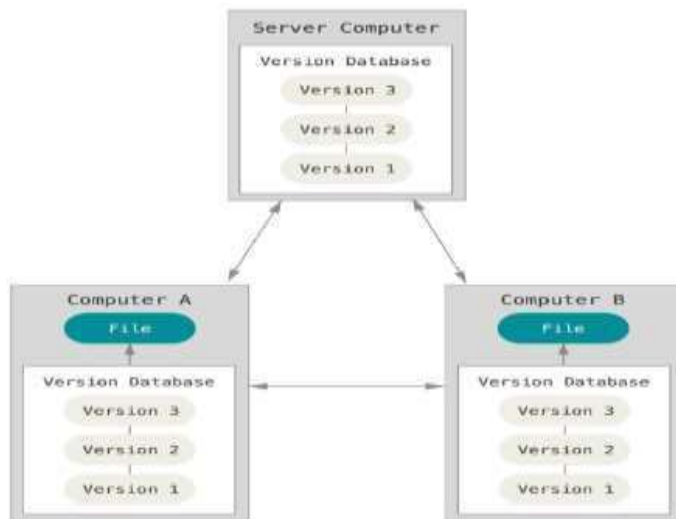


Fig 1.9 Distributed Version Control Systems

THE THREE STATES

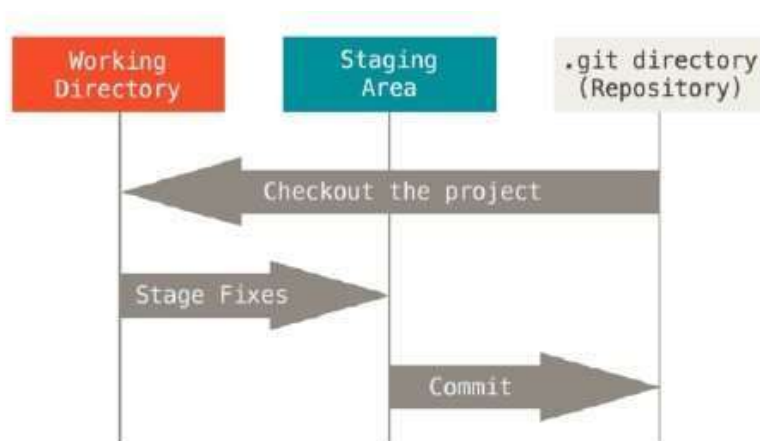


Fig 1.10 The Three States

1.11 INSTALLING GIT

Installing on Linux

```
$ sudo yum install git-all  
(or)
```

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

Installing on Mac

An OSX Git installer is maintained and available for download at the Git website, at <http://git-scm.com/download/mac>.

Installing on Windows

☛ Download from the GitHub for Windows website, at <http://windows.github.com>.

GETTING A GIT REPOSITORY

It first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

INITIALIZING A REPOSITORY IN AN EXISTING DIRECTORY

For Starting the existing project, go to the project directory and type:

```
$ git init
```

This creates a new subdirectory named .git that contains all of your necessary repository files. Git add commands that specify the files for track, followed by a git commit:

```
$ git add *.c
```

```
$ git add LICENSE
```

```
$ git commit -m 'initial project version'
```

Now, have a Git repository with tracked files and an initial commit.

1.11 INSTALLING GIT

Installing on Linux

```
$ sudo yum install git-all  
(or)
```

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

Installing on Mac

An OSX Git installer is maintained and available for download at the Git website, at <http://git-scm.com/download/mac>.

Installing on Windows

☛ Download from the GitHub for Windows website, at <http://windows.github.com>.

GETTING A GIT REPOSITORY

It first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

INITIALIZING A REPOSITORY IN AN EXISTING DIRECTORY

For Starting the existing project, go to the project directory and type:

\$ git init

This creates a new subdirectory named `.git` that contains all of your necessary repository files. Git add commands that specify the files for track, followed by a git commit:

\$ git add *.c

\$ git add LICENSE

\$ git commit -m 'initial project version'

Now, have a Git repository with tracked files and an initial commit.

CLONING AN EXISTING REPOSITORY

To get a copy of an existing Git repository

For example, To clone the Git linkable library called `libgit2`, the format is given below:

\$ git clone https://github.com/libgit2/libgit2

The above line creates a directory named `"libgit2"`, initializes a `.git` directory inside it, To clone the repository into a directory named something other than `"libgit2"`, can specify that as the next command-line option:

\$ git clone https://github.com/libgit2/libgit2 mylibgit

That command does the same thing as the previous one, but the target directory is called `mylibgit`.

1.12 RECORDING CHANGES TO THE REPOSITORY

- ✿ Each file in the working directory can be in one of two states: tracked or untracked.
- ✿ Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.
- ✿ Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area.

- Initially, clone a repository, all of the files will be tracked and unmodified because Git just checked them out and haven't edited anything. On editing the files, Git sees them as modified, because it has changed them since the last commit. At stage, these modified files and then commit all your staged changes, and the cycle repeats.

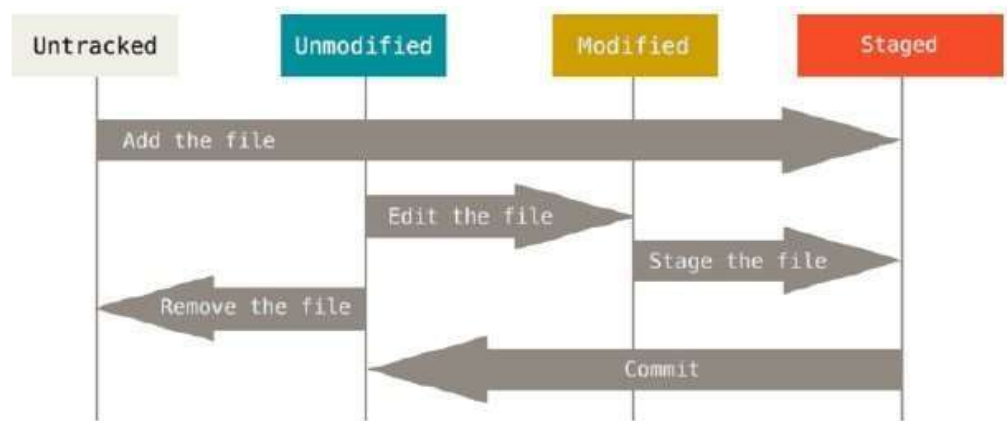


Fig 1.11 The lifecycle of the status of files

CHECKING THE STATUS OF YOUR FILES

The main tool used to determine which files are in which state is the git status command.

\$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

This means it has a clean working directory, in other words, there are no tracked and modified files. On adding a new file to the project, a simple README file. If the file didn't exist before, and run git status, the untracked file can be viewed as given below:

```
$ echo 'My Project' > README
```

\$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

TRACKING NEW FILES

In order to begin for tracking a new file, use the command git add. To begin tracking the first README file, by running this:

\$ git add README

On running the status command again, can find that README file is now tracked and staged to be committed:

\$ git status

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
```

STAGING MODIFIED FILES

To stage a file, for example, **CONTRIBUTING.md**, run the **git add** command. **git add** is a multipurpose command – use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved.

\$ git add CONTRIBUTING.md

\$ git status

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   CONTRIBUTING.md
```

SHORT STATUS

\$ git status -s

```
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

IGNORING FILES

\$ cat .gitignore

```
*.[oa]
```

The first line tells Git to ignore any files ending in “.o” or “.a” – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work.
- Can start patterns with a forward slash (/) to avoid recursivity.
- Can end patterns with a forward slash (/) to specify a directory.
- Can negate a pattern by starting it with an exclamation point (!).

VIEWING YOUR STAGED AND UNSTAGED CHANGES

To see what have changed but not yet staged, type git diff with no other arguments:

\$ git diff

```
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
```

This command compares the staged changes to last commit:

\$ git diff --staged

```
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

\$ git diff --cached

```
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
```

COMMITTING THE CHANGES

The simplest way to commit is to type git commit:

```
$ git commit
```

REMOVING FILES

To remove a file from Git, remove it from tracked files (more accurately, remove it from staging area) and then commit.

output:

```
$ rm PROJECTS.md
```

```
$ git status
```

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

For File removal, run **git rm**.

Example:

```
$ git rm PROJECTS.md
```

```
rm 'PROJECTS.md'
```

```
$ git rm --cached README
```

To pass files, directories, and file-glob patterns to the git rm command,

```
$ git rm log/\*.log
```

Note the backslash (\) in front of the *. This is necessary because Git does its own filename expansion in addition to the shell's filename expansion. This command removes all files that have the .log extension in the log/ directory. Otherwise, do something like this:

```
$ git rm \*~
```

This command removes all files whose names end with a ~.

Moving Files

For renaming the file name use the command given below:

```
$ git mv file_from file_to
```

Example:

```
$ git mv README.md README
```

```
$ git status
```

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Another Method for replacing the file name:

```
$ mv README.md README
```

```
$ git rm README.md
```

```
$ git add README
```

1.13 VIEWING THE COMMIT HISTORY

After the creation of few commits or the repository is cloned with the existing history and to see what happened, the git log command can be used.

Example: Very simple project called "simplegit". To get the project, run

```
$ git clone https://github.com/schacon/simplegit-progit
```

LIMITING LOG OUTPUT

The time-limiting options such as --since and --until are very useful. For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

```
$ git log -Sfunction_name
```

1.14. UNDOING THINGS

Undo is possible at any stage. **\$ git commit –amend**

This command takes the staging area and uses it for the commit. If there is no changes since the last commit, as an example, do commit and then realize that forgot to stage the changes in a file is wanted to add to

this commit, for that do something like this:

\$ git commit -m 'initial commit'

\$ git add forgotten_file

\$ git commit –amend

To end up with a single commit, the second commit replaces the results of the first.

1.UNSTAGING A STAGED FILE

Suppose it has changed two files and want to commit them as two separate changes, but it accidentally type git add * and stage them both. How can unstage one of the two?

Use the following command:

\$ git add *

\$ git status

On branch master Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README modified:

CONTRIBUTING.md

From the above output, right below the "Changes to be committed" text, it says use git reset HEAD <file>... to unstage. So, use that advice to unstage the CONTRIBUTING.md file:

\$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

\$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

1.14.2 UNMODIFYING A MODIFIED FILE

Use git checkout option, for unmodifying the changes made.

\$ git checkout -- CONTRIBUTING.md

\$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

1.15 WORKING WITH REMOTES

Remote repositories are versions of the project that are hosted on the Internet or network somewhere. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when it need to share work.

SHOWING YOUR REMOTES

```
$ git clone https://github.com/schacon/ticgit
```

```
Cloning into 'ticgit'...
```

```
remote: Reusing existing pack: 1857, done.
```

```
remote: Total 1857 (delta 0), reused 0 (delta 0)
```

```
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
```

```
Resolving deltas: 100% (772/772), done.
```

```
Checking connectivity... done.
```

```
$ cd ticgit
```

```
$ git remote
```

```
Origin
```

Specify -v, which shows the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
```

```
origin https://github.com/schacon/ticgit (fetch)
```

```
origin https://github.com/schacon/ticgit (push)
```

```
$ git remote add pb https://github.com/paulboone/ticgit
```

```
$ git remote -v
```

To fetch all the information run the command **git fetch pb**.

```
$ git fetch pb
```

```
remote: Counting objects: 43, done.
```

Adding Remote Repositories

To add a new remote Git repository as a shortname it can reference easily, run git remote add <shortname> <url>:

```
$ git remote
```

```
origin
```

FETCHING AND PULLING FROM YOUR REMOTES

\$ **git fetch [remote-name]**

The command goes out to that remote project and pulls down all the data from that remote project that don't have yet.

PUSHING TO YOUR REMOTES

When the project is at a point that want to share, then push it to upstream. The command for this is simple: **git push [remote-name] [branch-name]**.

\$ **git push origin master**

INSPECTING A REMOTE

To see more information about a particular remote, use the **git remote show [remote-name]** command. Run this command with a particular shortname, such as origin, get something like this:

\$ **git remote show origin**

The command tells that if it is on the master branch and run **git pull**, it will automatically merge in the master branch on the remote after it fetches all the remote references. It also lists all the remote references it has pulled down.

\$ **git remote show origin**

This command shows which branch is automatically pushed to when it run **git push** while on certain branches. It also shows which remote branches on the server don't yet have, which remote branches has and that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when run **git pull**.

REMOVING AND RENAMING REMOTES

Run **git remote rename** to change a remote's shortname. For instance, to rename pb to paul, can do so with **git remote rename**:

\$ **git remote rename pb paul**

\$ **git remote**

To remove a remote, use **git remote rm**:

```
$ git remote rm paul
```

```
$ git remote
```

Origin

1.16 TAGGING

Tagging is used to know how to list the available tags, how to create new tags, and what the different types of tags are

LISTING YOUR TAGS

Listing the available tags in Git is straightforward. Run **git tag**:

```
$ git tag
```

```
v0.1
```

```
v1.3
```

This command lists the tags in alphabetical order; the order in which they appear has no real importance.

Can also search for tags with a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you're only interested in looking at the 1.8.5 series, run this:

```
$ git tag -l "v1.8.5*"
```

```
v1.8.5
```

```
v1.8.5-rc0
```

CREATING TAGS

Git uses two main types of tags:

- **Lightweight**
- **Annotated**

A **lightweight tag** is very much like a branch that doesn't change, it is just a pointer to a specific commit.

Annotated tags are stored as full objects in the Git database.

ANNOTATED TAGS

Creating an annotated tag in Git is simple. The easiest way is to specify **-a** when run the tag command:

```
$ git tag -a v1.4 -m "my version 1.4"
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```

The **-m** specifies a tagging message, which is stored with the tag.

To see the tag data along with the commit that was tagged by using the **git show** command:

```
$ git show v1.4
```

```
tag v1.4
```

```
Tagger: Ben Straub <ben@straub.cc>
```

```
Date: Sat May 3 20:19:12 2014 -0700
```

```
my version 1.4
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

LIGHTWEIGHT TAGS

Other way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file, no other information is kept. To create a lightweight tag, don't supply the **-a**, **-s**, or **-m** option:

```
$ git tag v1.4-lw
```

```
$ git tag
```

```
v0.1
```

Run **git show** on the tag, don't see the extra tag information. The command just shows the commit:

```
$ git show v1.4-lw
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

TAGGING LATER

To tag commits after that moved past them. Suppose the commit history looks like this:

```
$ git log --pretty=oneline
```

suppose forgot to tag the project at v1.2, which was at the “updated rakefile” commit. Now, can add it after the fact. To tag that commit, specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```

```
$ git tag
```

```
v0.1
```

```
v1.2
```

```
$ git show v1.2
```

```
tag v1.2
```

```
Tagger: Scott Chacon <schacon@gee-mail.com>
```

```
updated rakefile
```

The **git push** command doesn’t transfer tags to remote servers. Now it has to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches you can run **git push origin [tagname]**.

```
$ git push origin v1.5
```

```
Counting objects: 14, done.
```

If there is lot of tags that want to push up at once, can also use the **--tags** option to the **git push** command. This will transfer all of the tags to the remote server that are not already there.

```
$ git push origin --tags
```

```
Counting objects: 1, done.
```

Now, when someone else clones or pulls from the repository, they will get all the tags as well.

CHECKING OUT TAGS

It can’t check out a tag in Git, since they can’t be moved around. To put a version of the repository in the working directory that looks like a specific tag, create a new branch at a specific tag with **git checkout -b [branchname] [tagname]**:

\$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

Of course to do this and do a commit, your version2 branch will be slightly different than v2.0.0 tag since it will move forward with new changes.

1.17 GIT ALIASES

Git doesn't automatically infer the command if the type it in partially. If it don't want to type the entire text of each of the Git commands, it can easily set up an alias for each command using git config. Here are a couple of examples may want to set up:

```
$ git config --global alias.co checkout
```

```
$ git config --global alias.br branch
```

```
$ git config --global alias.ci commit
```

```
$ git config --global alias.st status
```

Instead of typing git commit, just need to type git ci. On using Git, it will probably use other commands frequently as well; don't hesitate to create new aliases.

This technique can also be very useful in creating commands that the think should exist. For example, to correct the usability problem that encountered with unstaging a file, then can add the own unstage alias to Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

This makes the following two commands equivalent:

```
$ git unstage fileA
```

```
$ git reset HEAD -- fileA
```

```
$ git config --global alias.last 'log -1 HEAD'
```

```
$ git last
```

```
$ git config --global alias.visual '!gitk'
```

1.18 GIT BRANCHING

- Branching means diverging from the main line of development and continue to do work without messing with that line.

1.19 BRANCHES IN A NUTSHELL

- Git stores data as a series of snapshots.

- When we make a commit, Git stores a commit object that contains a pointer to the snapshot of the content staged. Assume a directory containing 3 files, and stage them all and commit. Staging the files computes a checksum for each one, stores that version of the file in the Git repository, and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'Initial commit'
```

While creating commit by running `git commit`, Git checksums each subdirectory and stores them as a tree object in the Git repository. Git creates a commit object that has the metadata and a pointer to the root project tree to re-create that snapshot when needed.

The Git repository will now contains 5 objects: 3 blobs (each representing the contents of one of the 3 files), 1 tree that lists the contents of the directory and specifies which file names are stored as which blobs, and 1 commit with the pointer to that root tree and all the metadata.

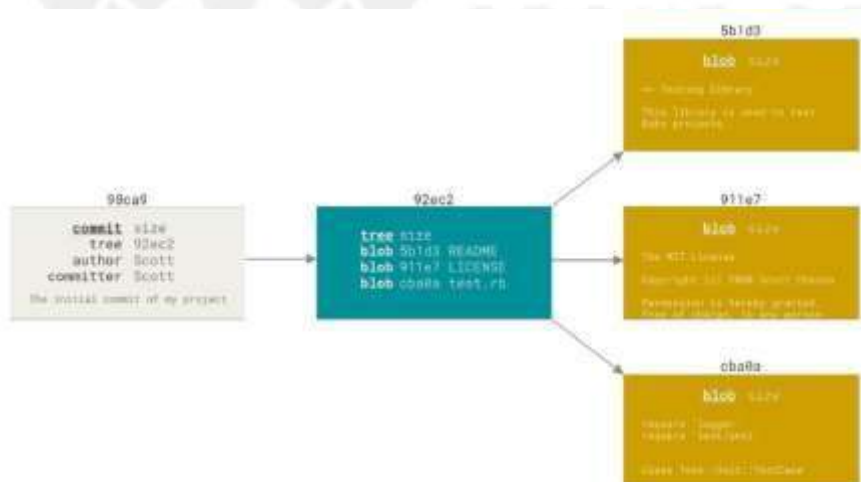


Fig 1.12 A commit and its tree

When making changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is master. When making commits, a master branch will be created that points to the last commit made.

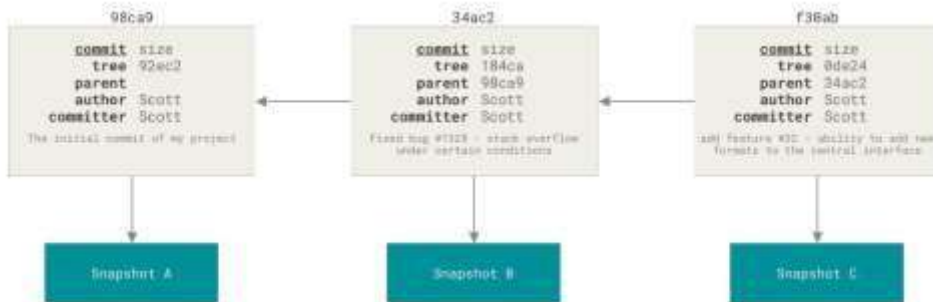


Fig 1.13 Commits and their parents

- When commit is done, the master branch pointer moves forward automatically.
- The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is the git init command creates it by default and most people don't bother to change it.

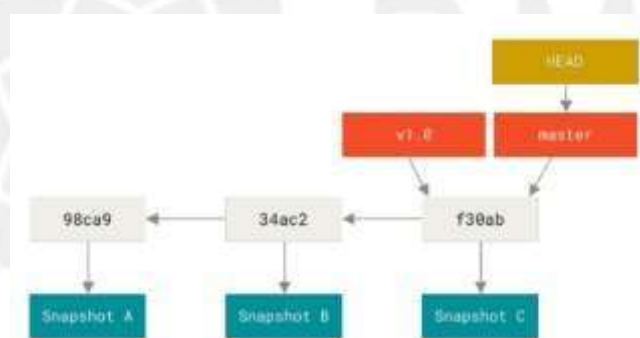


Fig 1.14 A branch and its commit history

CREATING A NEW BRANCH

Creating a new branch creates a new pointer to move around. To create a new branch called testing, git branch command:

\$ git branch testing



Fig 1.15 Two branches pointing into the same series of commit

In Git, HEAD is a pointer to the local branch master we are currently on. The git branch command only created a new branch it did not switch to that branch.



Fig 1.16 HEAD pointing to a branch

It can be seen by running a simple git log command that shows where the branch pointers are pointing. This option is called decorate.

\$ git log --oneline--decorate f30ab (HEAD -> master, testing)

SWITCHING BRANCHES

To switch to an existing branch, run the git checkout command.

\$ git checkout testing

This moves HEAD to point to the testing branch.



Fig 1.17 HEAD points to the current branch

Do another commit:

\$ vim test.rb

\$ git commit -a -m 'made a change'

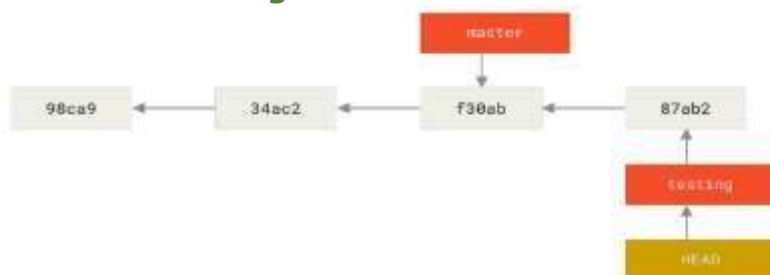


Fig 1.18 The HEAD branch moves forward when a commit is made

The testing branch has moved forward, but master branch still points to the previous commit. To switch back to the master branch:

\$ git checkout master

git log will only show commit history below the branch we have checked out.

To show commit history for the desired branch, explicitly specify it: git log testing.

To show all of the branches, add --all to git log command.

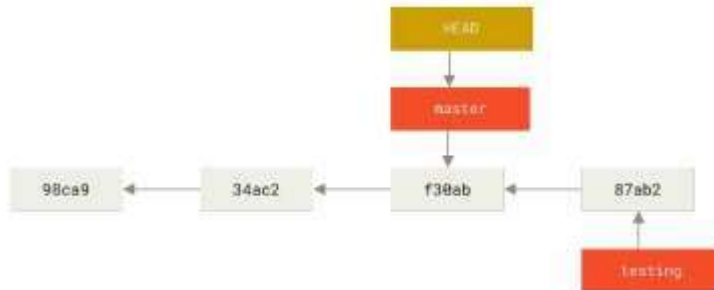


Fig 1.19 HEAD moves when we checkout

- Above command moved the HEAD pointer back to point to the master branch, and it reverted the files in working directory back to the snapshot that master points to.

\$ vim test.rb

\$ git commit -a -m 'made other changes'

Now project history has diverged (Fig Divergent history). Switching back and forth between the branches and merge them together are done with simple branch, checkout, and commit commands.

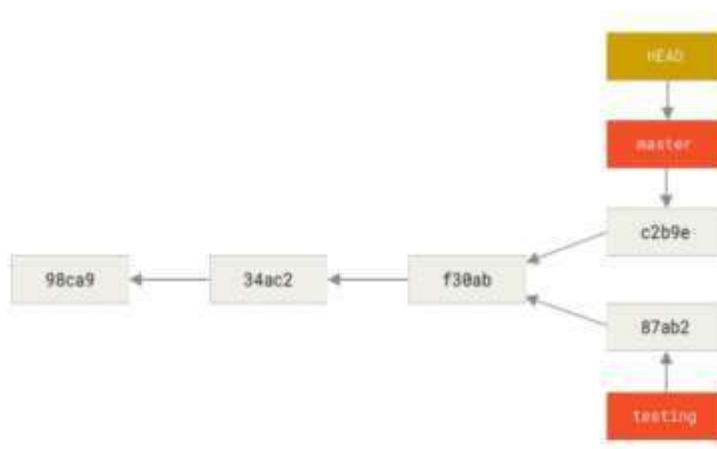


Fig 1.20 Divergent history


```
$ git log --oneline --decorate --graph --all
```

```
* c2b9e (HEAD, master) Made other changes
| * 87ab2 (testing) Made a change
|/
```

CREATING A NEW BRANCH AND SWITCHING TO IT AT THE SAME TIME

- ✿ To create a new branch and want to switch to that new branch at the same time — this can be done with **git checkout -b <newbranchname>**.
- ✿ Create a new branch and switch to it: **git switch -c new-branch**. The -c flag stands for create, we can also use the full flag: --create.
- ✿ Return to previously checked out branch: git switch.

1.20 BASIC BRANCHING AND MERGING

Steps for branching and merging with a workflow that might be used in the real world:

1. Do some work on a website.
2. Create a branch for working on a new user story.
3. Do some work in that branch.

To solve a critical issue by hotfix, following steps will be followed:

1. Switch to production branch.
2. Create a branch to add the hotfix.
3. After it is tested, merge the hotfix branch, and push to production.
4. Switch back to your original user story and continue working.

BASIC BRANCHING

Consider in a project couple of commits are already done on the master branch.

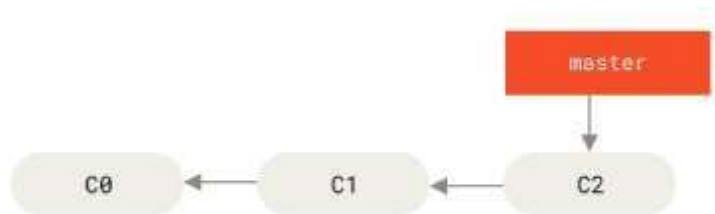


Fig 1.21 A simple commit history

To work on issue #53 in whatever issue-tracking system company uses, create a new branch and switch to it at the same time using git checkout command with the -b switch:

```
$ git checkout -b iss53
```

Switched to a new branch "iss53"

This is shorthand for:

```
$ git branch iss53
```

```
$ git checkout iss53
```

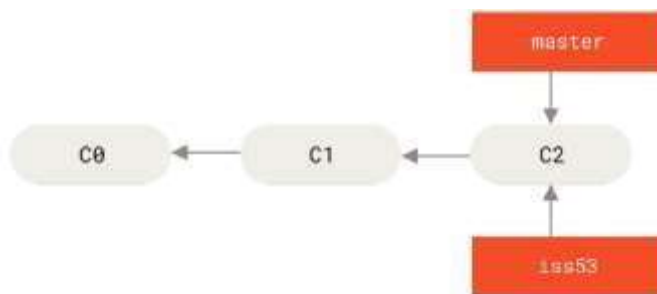


Fig 1.22 Creating a new branch pointer

```
$ vim index.html
```

```
$ git commit -a -m 'Create new footer [issue 53]'
```

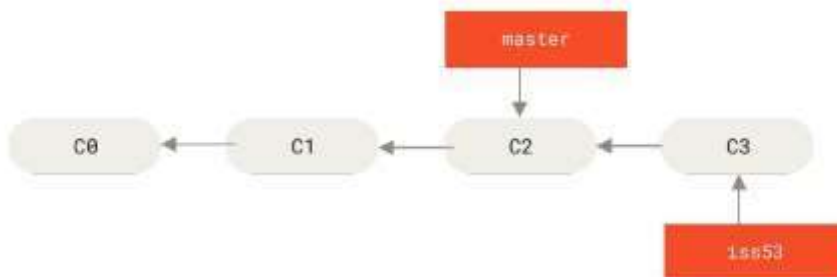


Fig 1.23 The iss53 branch has moved forward with your work

Command to switch back to master branch:

```
$ git checkout master
```

Git resets the working directory to look like it did the last time we commit on that branch. It adds, removes, and modifies files automatically to make sure the working copy is what the branch looked like on the last commit to it.

Create a hotfix branch on which to work until it's completed:

```
$ git checkout -b hotfix
```

Switched to a new branch 'hotfix'

```
$ vim index.html
```

```
$ git commit -a -m 'Fix broken email address'
```

```
[hotfix 1fb7853] Fix broken email address
```

```
1 file changed, 2 insertions(+)
```

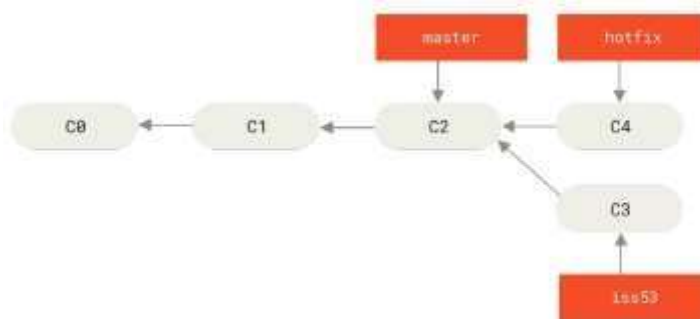


Fig 1.24 Hotfix branch based on master

To merge the hotfix branch back into master branch to deploy to production, use the git merge command:

```
$ git checkout master
```

```
$ git merge hotfix
```

Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2, Git simply moves the pointer forward. To merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a “fastforward.”

The change will be now in the snapshot of the commit pointed to by the master branch, and we can deploy the fix.

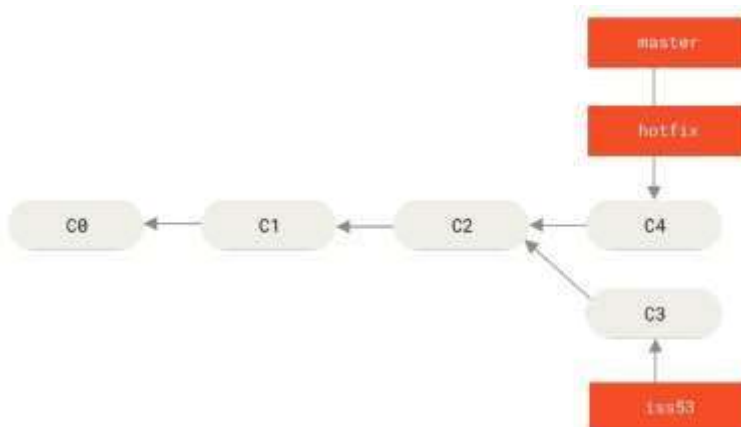


Fig 1.25 master is fast-forwarded to hotfix

To delete the hotfix branch, command used is:

```
$ git branch -d hotfix
```

Deleted branch hotfix (3a0874c).

To switch back to the branch on issue #53 and continue working on it.

```
$ git checkout iss53
```

Switched to branch "iss53"

```
$ vim index.html
```

```
$ git commit -a -m 'Finish the new footer [issue 53]'
```

[iss53 ad82d7a] Finish the new footer [issue 53]

1 file changed, 1 insertion(+)

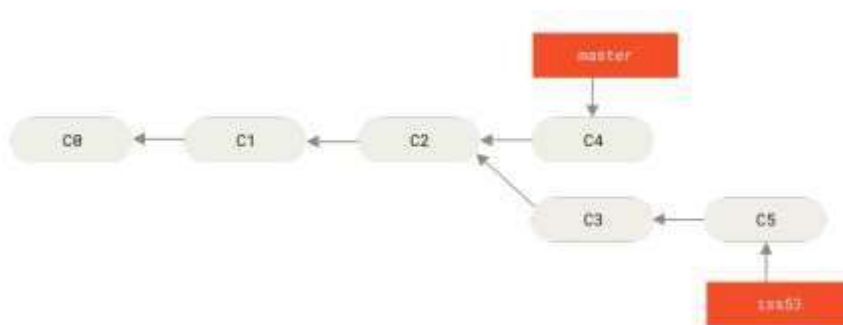


Fig 1.26 Work continues on iss53

To pull the work in hotfix branch into iss53 branch, merge master branch into iss53 branch by running `git merge master`, or can wait to integrate those changes until pulling the iss53 branch back into master later.

BASIC MERGING

To merge iss53 branch into master branch, check out the branch and then run the `git merge` command:

```
$ git checkout master
```

Switched to branch 'master'

```
$ git merge iss53
```

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

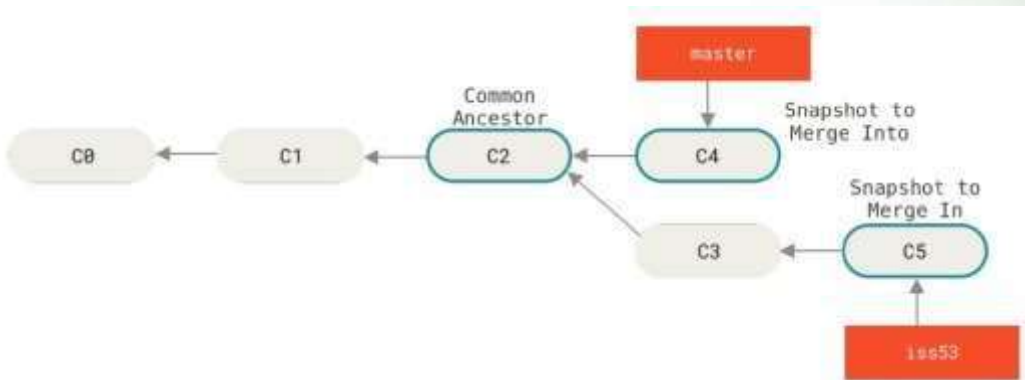


Fig 1.27 Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

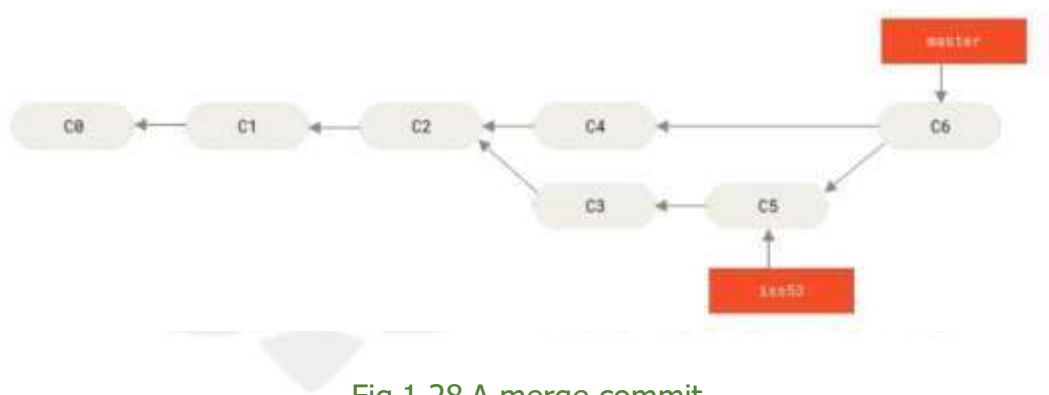


Fig 1.28 A merge commit

As the work is merged in, and there is no further need for the iss53 branch, it can be deleted with the following command:

\$ git branch -d iss53

BASIC MERGE CONFLICTS

If the same part of the same file is changed differently in the two merging branches, Git won't be able to merge them cleanly. If fix for issue #53 modified the same part of a file as the hotfix branch, a merge conflict message will be displayed as follows:

\$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git pauses the merge commit process to resolve the conflict.

\$ git mergetool

\$ git status

Type git commit to finalize the merge commit.

1.21 BRANCH MANAGEMENT

To see the last commit on each branch,

\$ git branch -v

The useful **--merged** and **--no-merged** options can filter this list to branches merged into the current branch. To see which branches are already merged into the Branch run the following command:

\$ git branch --merged

iss53

- Master

To see all the branches that contain work not yet merged, run the following command:

\$ git branch --no-merged

testing

This shows the other branch. Because it contains work that is not merged in yet, trying to delete it with git branch -d will fail:

\$ git branch -d testing

error: The branch 'testing' is not fully merged.

To forcefully delete it, run the following:

\$ git branch -D testing.

To chat what is not merged into the master branch?

\$ git checkout testing

\$ git branch --no-merged master

topicA

featureB

CHANGING A BRANCH NAME

Rename the branch locally with the git branch --move command:

```
$ git branch --move bad-branch-name corrected-branch-name
```

This replaces your bad-branch-name with corrected-branch-name, but this change is only locally.

To let others see the corrected branch on the remote, push it:

```
$ git push --set-upstream origin corrected-branch-name
```

Now we'll take a brief look at where we are now:

```
$ git branch --all
```

```
*corrected-branch-name
```

```
Main
```

The branch with the bad name is also still present there but delete it by executing the following command:

```
$ git push origin --delete bad-branch-name
```

CHANGING THE MASTER BRANCH NAME

Changing the name of a branch like master/main/mainline/default will break the integrations, services, helper utilities and build/release scripts that the repository uses.

Rename local master branch into main with the following command:

```
$ git branch --move master main
```

To let others see the new main branch, push it to the remote. This makes the renamed branch available on the remote.

```
$ git push --set-upstream origin main
```

Now we end up with the following state:

```
$ git branch --all
```

```
* main
```

```
remotes/origin/HEAD -> origin/master
```

```
remotes/origin/main
```

```
remotes/origin/master
```

After confirming that the main branch performs just as the master branch, delete the master branch by the following:

```
$ git push origin --delete master
```

1.22 BRANCHING WORKFLOWS

Lightweight branching makes common workflow possible, and it can be to incorporated into development cycle.

LONG-RUNNING BRANCHES

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch possibly only code that has been or will be released. They have another parallel branch named develop or next that they work from or use to test stability. it is not necessarily always stable, but whenever it gets to a stable state, it can be merged into master.

It's used to pull in topic branches (short-lived branches, like is s53 branch) when They are ready, to make sure they pass all the tests and don't introduce bugs.

The stable branches are farther down the line in commit history, and the bleeding-edge branches are farther up the history.

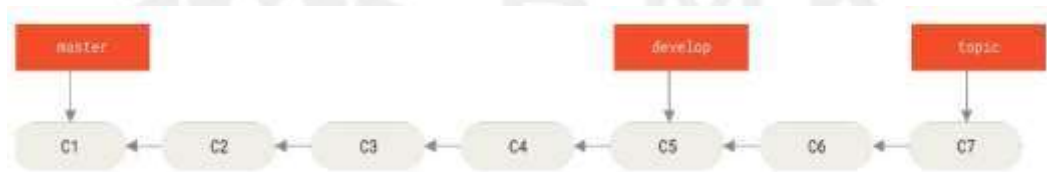


Fig 1.29 A linear view of progressive-stability branching

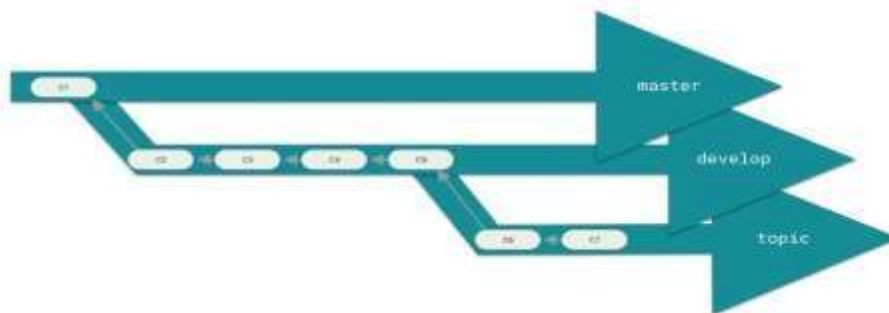


Fig 1.30 A "silo" view of progressive-stability branching

Having multiple long-running branches isn't necessary, but it is often helpful, especially while dealing with very large or complex projects.

Topic Branches

- Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch used for a single particular feature or related work.
- Consider an example of doing some work (on master), branching off for an issue (iss91), working on it for a bit, branching off the second branch to try another way of handling the same thing (iss91v2), going back to master branch and working there for a while, and then branching off there to do some work that are not sure is a good idea (dumbidea branch). The commit history will look something like this:

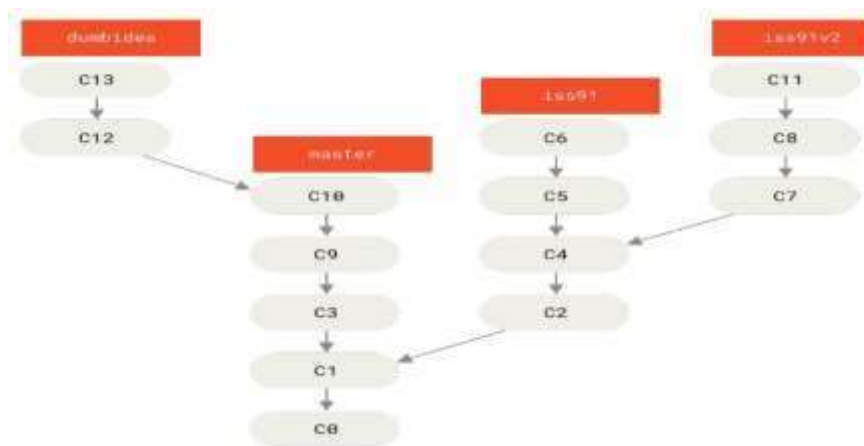


Fig 1.31 Multiple topic branches

If suppose the second solution to the issue is best (iss91v2); and if the dumbidea branch is selected, and need to throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two, it can be done. The history then looks like this:

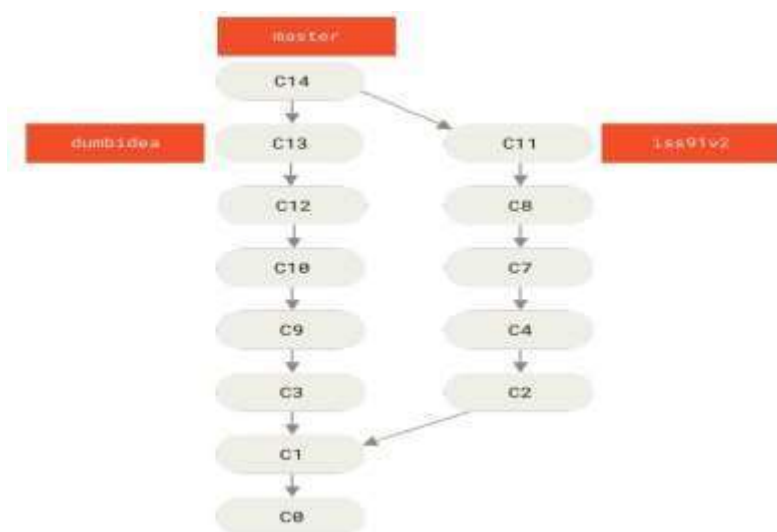


Fig 1.32 History after merging dumbidea and iss91v2

1.23 REMOTE BRANCHES

Remote references are references (pointers) in the remote repositories, including branches, tags, and so on. We can get a full list of remote references explicitly with `git ls-remote <remote>`, or `git remote show <remote>` for remote branches as well as more information.

Remote-tracking branches are references to the state of remote branches.

"origin" is not special. Just like the branch name "master" does not have any special meaning in Git, neither does "origin". While "master" is the default name for a starting branch to run `git init`, "origin" is the default name for a remote while running `git clone`.

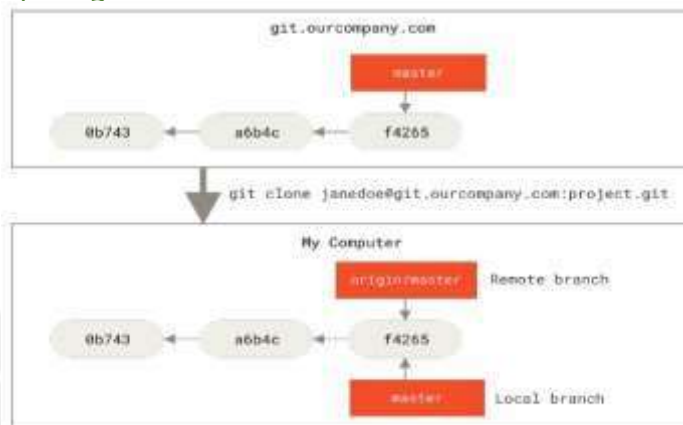


Fig 1.33 Server and local repositories after cloning

If we work on local master branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its master branch, then histories move forward differently. If we stay out of contact with origin server, origin/master pointer does not move.

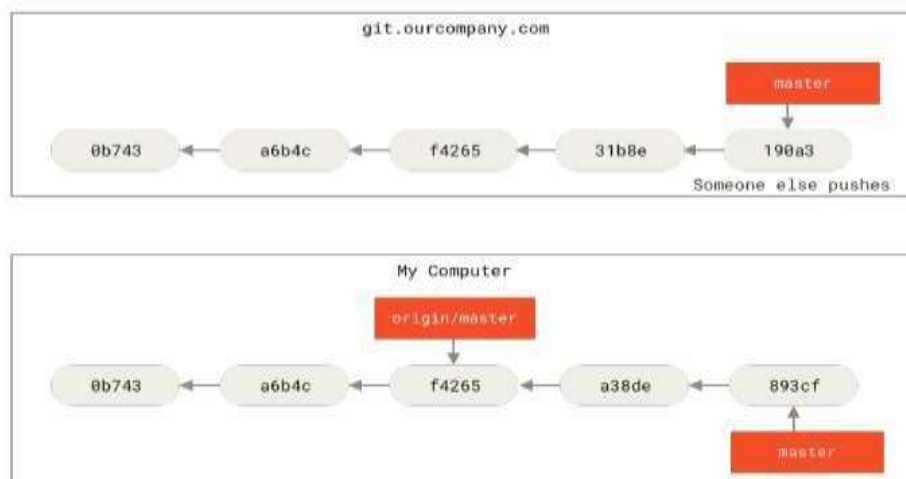


Fig 1.34 Local and remote work can diverge

To synchronize work with a given remote, run a `git fetch <remote>` command (here, `git fetch origin`). This command looks up which server “origin” is (in this case, it’s `git.ourcompany.com`), fetches any data from it and updates it to local database, moving origin/master pointer to its new, more up-to-date position.

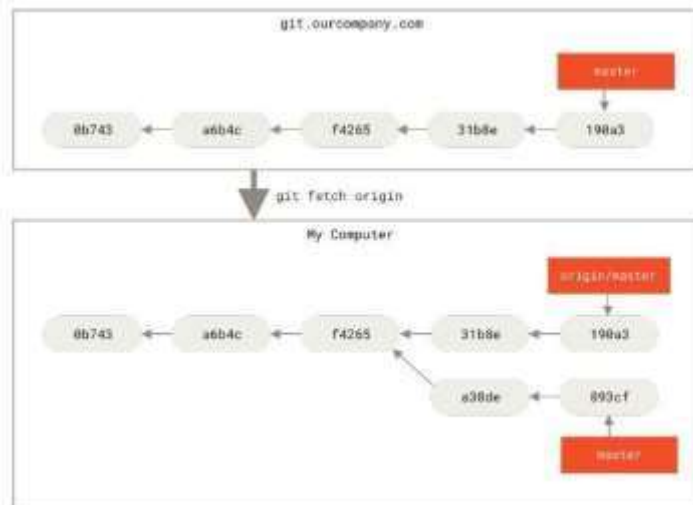


Fig 1.35 git fetch updates remote-tracking branches

If we have another internal Git server that is used only for development by one of a sprint teams and this server is at `git.team1.ourcompany.com`, add it as a new remote reference to the current project by running the `git remote add` command. Name this remote `teamone`, which will be short name for that whole URL.

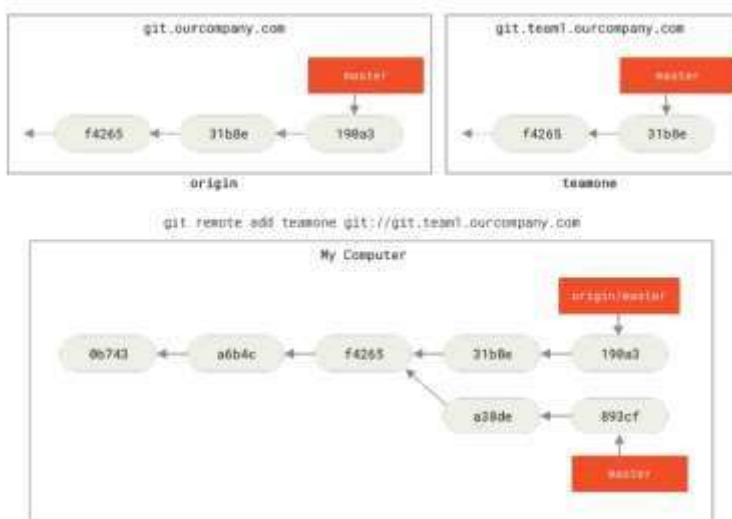


Fig 1.36 Adding another server as a remote

Run `git fetch teamone` to fetch everything the remote teamone server has. Because that server has a subset of the data origin server has right now, Git fetches no data but sets a remote-tracking branch called `teamone/master` to point to the commit that teamone has as its master branch.

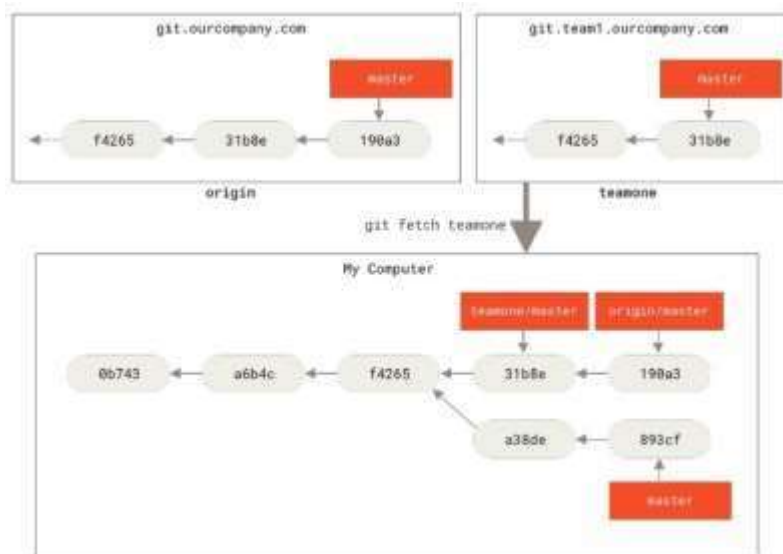


Fig 1.37 Remote-tracking branch for teamone/master

PUSHING

If there is a branch named `serverfix` to work on with others, push it up the same way we pushed first branch. Run `git push <remote> <branch>`:

\$ git push origin serverfix

Counting objects: 24, done.

To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own `serverfix` branch that you can work on, you can base it off your remote tracking branch:

\$ git checkout -b serverfix origin/serverfix

Branch `serverfix` set up to track remote branch `serverfix` from `origin`.

Switched to a new branch '`serverfix`'

This gives you a local branch that you can work on that starts where `origin/serverfix` is.

TRACKING BRANCHES

This is a common enough operation that Git provides the `--track` shorthand:

\$ git checkout --track origin/serverfix

Branch `serverfix` set up to track remote branch `serverfix` from `origin`.

Switched to a new branch '`serverfix`'

\$ git fetch --all; git branch -vv

PULLING

There is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases.

DELETING REMOTE BRANCHES

We can delete a remote branch using the `--delete` option to `git push`. To delete `serverfix` branch from the server, run the following:

\$ git push origin --delete serverfix

To <https://github.com/schacon/simplegit>
- [deleted] `serverfix`

Basically all this does is to remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it's often easy to recover.

1.24 REBASING

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

THE BASIC REBASE

In Basic Merging, we diverged the work and made commits on two different branches.

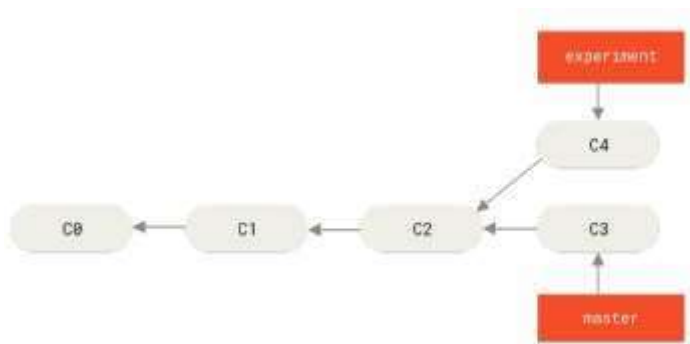


Fig 1.38 Simple divergent history

The easiest way to integrate the branches, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).

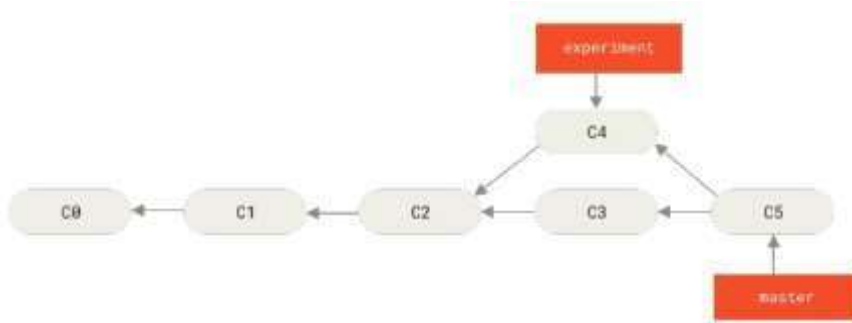


Fig 1.39 Merging to integrate diverged work history

However, there is another way: we can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called rebasing. With the rebase command, take all the changes that were committed on one branch and replay them on a different branch.

For this example, check out the experiment branch, and then rebase it onto the master branch as follows:

\$ git checkout experiment

\$ git rebase master

First, rewinding head to replay work on top of it.

Applying: added staged command

This operation works by going to the common ancestor of the two branches (the one we're on and the one we're rebasing onto), getting the diff introduced by each commit of the branch we're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch we are rebasing onto, and finally applying each change in turn.



Fig 1.40 Rebasing the change introduced in C4 onto C3

At this point, we can go back to the master branch and do a fast-forward merge.

\$ git checkout master

\$ git merge experiment

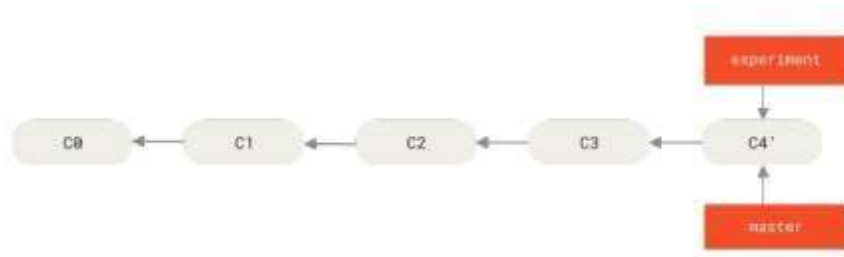


Fig 1.41 Fast-forwarding the master branch

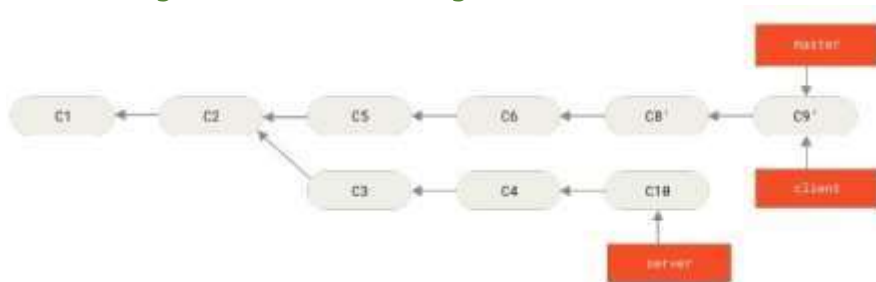


Fig 1.44 Fast-forwarding your master branch to include the client branch changes
Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running `git rebase <basebranch> <topicbranch>` which checks out the topic branch (in this case, server) for you and replays it on to the base branch (master):

\$ git rebase master server

This replays your server work on top of your master work, as shown in Rebasing your server branch on top of your master branch.

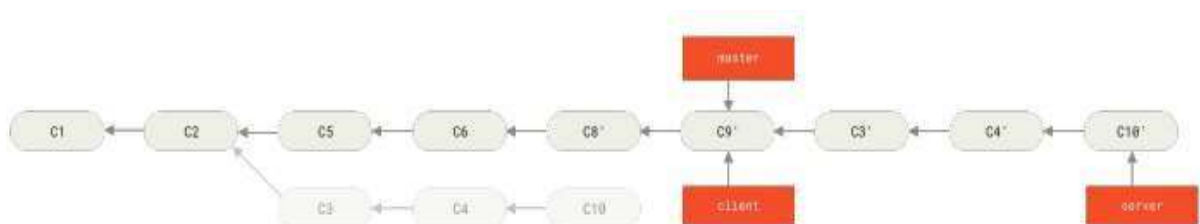


Fig 1.45 Rebasing your server branch on top of your master branch

Then, we can fast-forward the base branch (master):

```
$ git checkout master
```

```
$ git merge server
```

We can remove the client and server branches because all the work is integrated and we don't need them anymore, leaving history for this entire process looking like Final commit history:

```
$ git branch -d client
```

```
$ git branch -d server
```

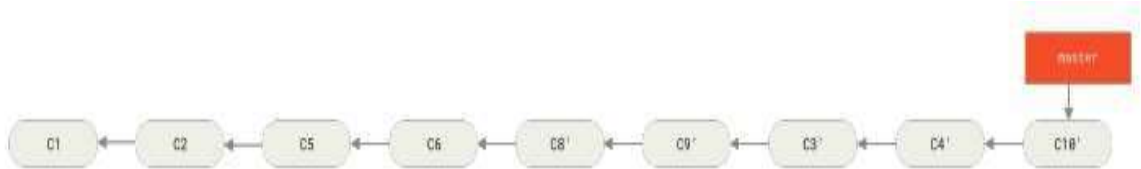


Fig 1.46 Final commit history

THE PERILS OF REBASING

Do not rebase commits that exist outside your repository and that people may have based work on.

When we rebase stuff, we're abandoning existing commits and creating new ones that are similar but different. If we push commits somewhere and others pull them down and base work on them, and then we rewrite those commits with git rebase and push them up again, the collaborators will have to re-merge their work and things will get messy when we try to pull their work back.

Example:

Suppose we clone from a central server and then do some work off that. The commit history looks like this:

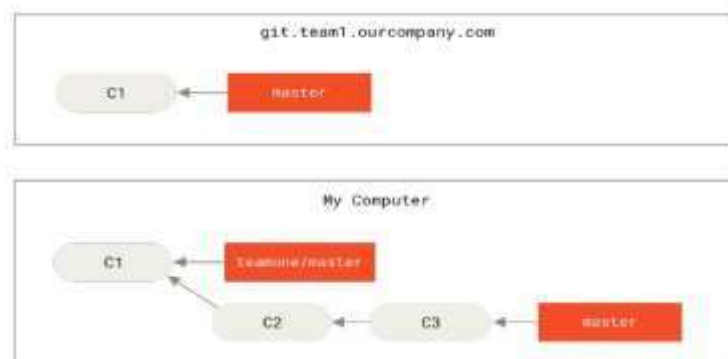


Fig 1.47 Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. We fetch it and merge the new remote branch into the work, making the history look like this:

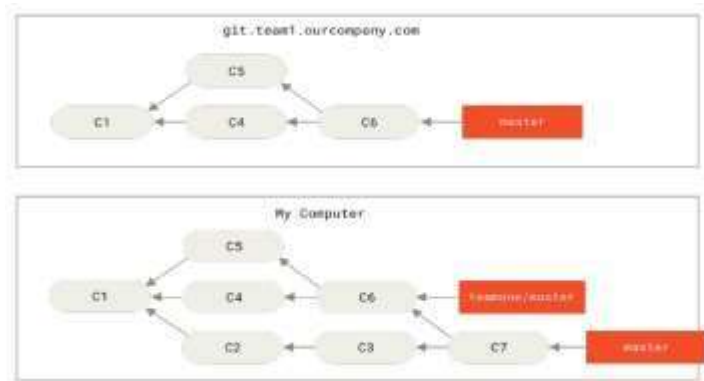


Fig 1.48 Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a git push force to overwrite the history on the server. We then fetch from that server, bringing down the new commits.

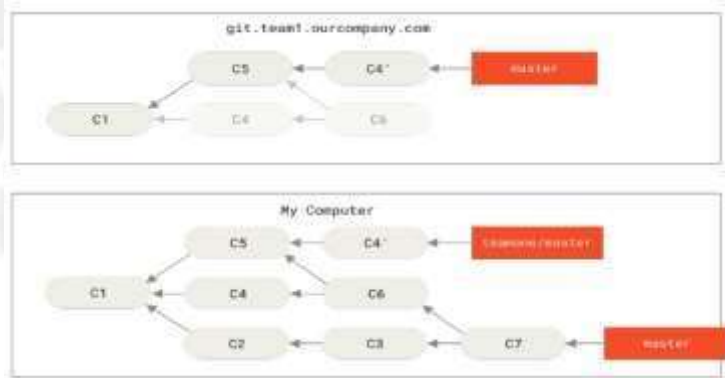


Fig 1.49 Someone pushes rebased commits, abandoning commits you have based your work on

Now if we do a git pull, we will create a merge commit which includes both lines of history, and the repository will look like this:

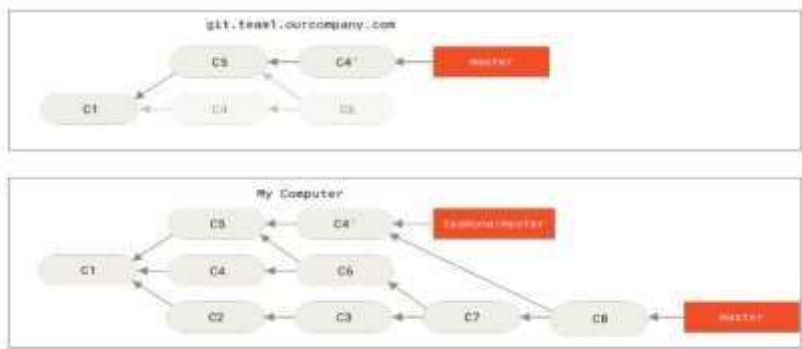


Fig 1.50 You merge in the same work again into a new merge commit

If we run a git log when the history looks like this, there will be two commits that have the same author, date, and message, which will be confusing. If we push this history back up to the server, it will reintroduce all those rebased commits to the central server, which can further confuse people. It is pretty safe to assume that the other developer does not want C4 and C6 to be in the history; that is why they rebased in the first place.

REBASE WHEN YOU REBASE

If you do find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you have based work on, your challenge is to figure out what is yours and what they have rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a “patch-id”.

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we're at Someone pushes rebased commits, abandoning commits you have based your work on we run `git rebase teamone/master`, Git will:

Determine what work is unique to our branch (C2, C3, C4, C6, C7)

- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')

Apply those commits to the top of teamone/master

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work.

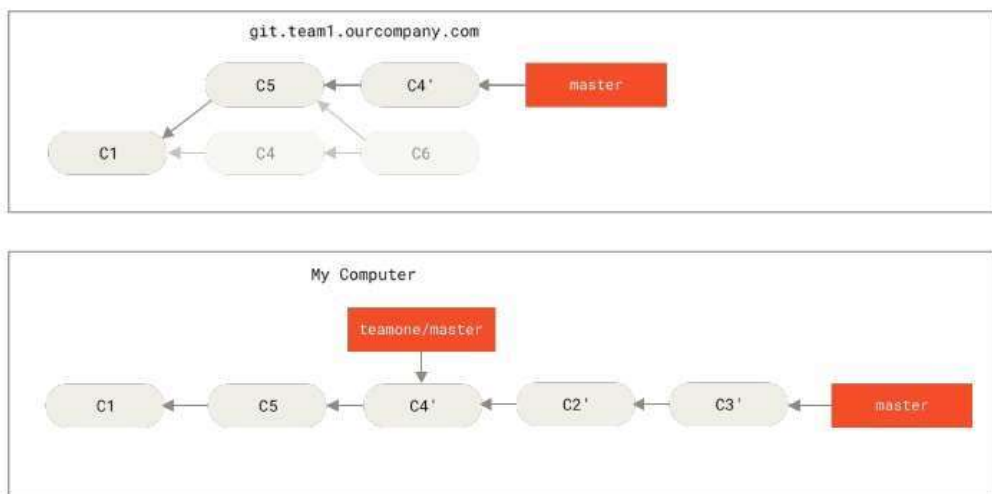


Fig 1.51 Rebase on top of force-pushed rebase work

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it is a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a `git pull rebase` instead of a normal `git pull`. Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If you are using `git pull` and want to make rebase the default, you can set the `pull.rebase` config value with something like `git config global pull.rebase true`.

If you only ever rebase commits that have never left your own computer, you will be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you'll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull rebase` to try to make the pain after it happens a little bit simpler.

REBASE VS. MERGE

Now that you have seen rebasing and merging in action, you may be wondering which one is better.

Before we can answer this, let us step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a record of what actually happened. It is a historical document, valuable in its own right, and should not be tampered with.

From this angle, changing the commit history is almost blasphemous; you are lying about what actually transpired. So what if there was a messy series of merge commits? That is how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the story of how your project was made.

You would not publish the first draft of a book, so why show your messy work? When you are working on a project, you may need a record of all your missteps and dead-end paths, but when it is time to show your work to the world, you may want to tell a more coherent story of how to get from A to B. People in this camp use tools like rebase and filter-branch to rewrite their commits before they are merged into the mainline branch. They use tools like rebase and filter-branch, to tell the story in the way that is best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it is not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it is up to you to decide which one is best for your particular situation.

You can get the best of both worlds: rebase local changes before pushing to clean up your work, but never rebase anything that you have pushed somewhere.

1.25 GITHUB

GitHub is the single largest host for Git repositories and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things.

1.26 ACCOUNT SET UP AND CONFIGURATION

1. Set up a free user account by visiting <https://github.com>
2. Fill sign up form for github
3. Clicking the Octocat logo at the top-left of the screen will navigate to dashboard page.
4. Public key can be configured if one wants to have SSH access
5. click the "Add an SSH key" button, give key a name, paste the contents of `~/.ssh/id_rsa.pub` public-key file into the text area, and click "Add key"
6. Add Profile details like email address and profile picture and two factor authentication can be used to enhance security.

1.27 CONTRIBUTION TO PROJECTS

1.27.1 FORKING PROJECTS

- When one have to use the existing project in git without push access fork command can be used. Fork command will clone or copy the existing project and further update can be done on cloned project.
- To fork a project, visit the project page and click the "Fork" button at the top-right of the page. After a few seconds, you'll be taken to your new project page, with your own writeable copy of the code.

1.27.2 GITHUB FLOW

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you're collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the Topic Branches workflow covered in Git Branching.

Here's how it generally works:

1. Fork the project.
2. Create a topic branch from master.
3. Make some commits to improve the project.
4. Push this branch to your GitHub project.
5. Open a Pull Request on GitHub.
6. Discuss, and optionally continue committing.
7. The project owner merges or closes the Pull Request.
8. Sync the updated master back to your fork.

1.27.3 CREATING A PULL REQUEST

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at <https://github.com/schacon/blink>.

First, we click the 'Fork' button as mentioned earlier to get our own copy of the project. Our user name here is "tonychacon" so our copy of this project is at <https://github.com/tonychacon/blink> and that's where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

```
$ git clone https://github.com/tonychacon/blink ①
```

Cloning into 'blink'...

```
$ cd blink
```

```
$ git checkout -b slow-blink ②
```

Switched to a new branch 'slow-blink'

```
$ git diff --word-diff ④
```

```
diff --git a/blink.ino b/blink.ino
```

```
index 15b9911..a6cc5a5 100644
```

```
--- a/blink.ino
```

```
+++ b/blink.ino
```

\$ git commit -a -m 'Change delay to 3 seconds' ⑤

\$ git push origin slow-blink ⑥

•[new branch] slow-blink -> slow-blink

- ① Clone our fork of the project locally.
- ② Create a descriptive topic branch.
- ③ Make our change to the code.
- ④ Check that the change is good.
- ⑤ Commit our change to the topic branch.
- ⑥ Push our new topic branch back up to our GitHub fork.

we can alternatively go to the “Branches” page at <https://github.com/<user>/<project>/branches> to locate your branch and open a new Pull Request from there.

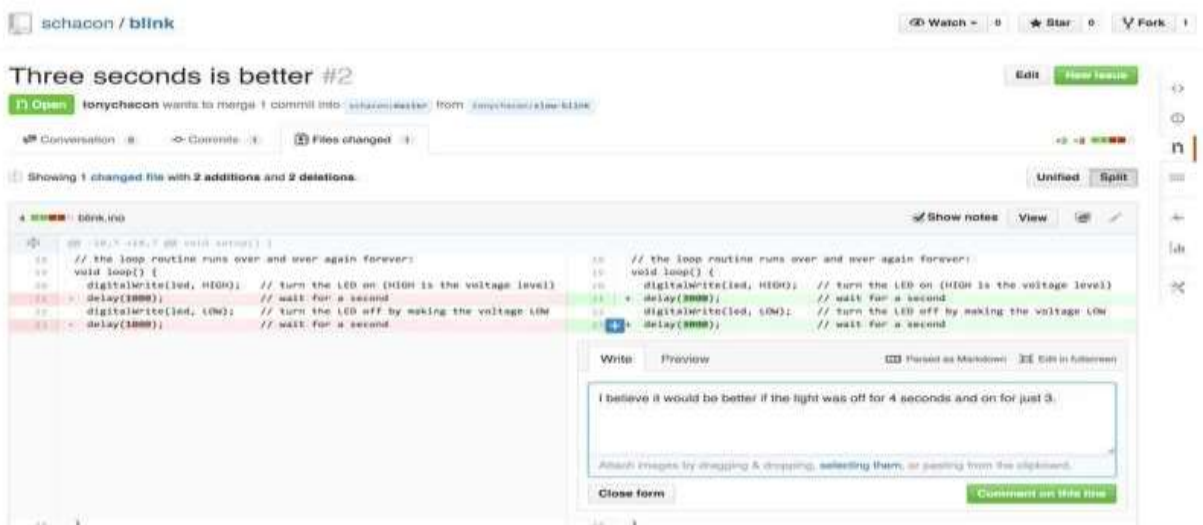
1.27.4 Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it,

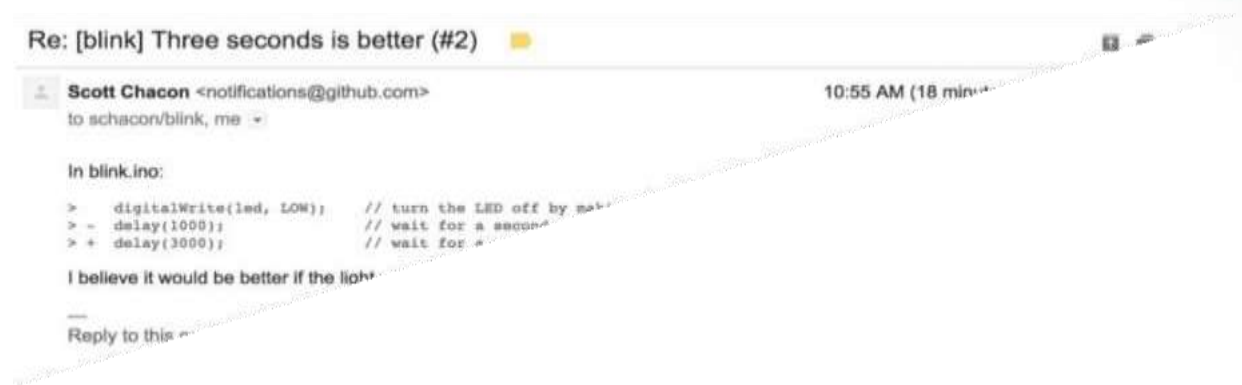
reject it or comment on it. Let’s say that he likes the idea, but would prefer a

slightly longer time for the light to be off than on.

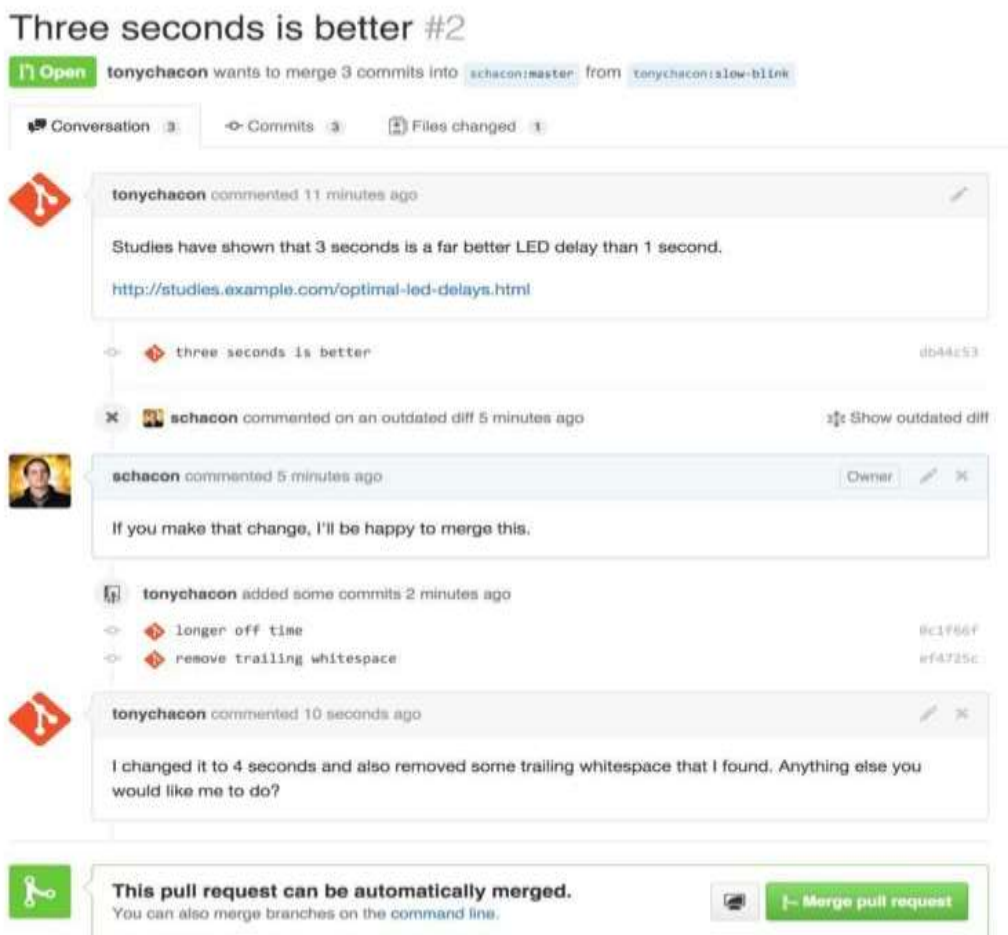
Where this conversation may take place over email in the workflows presented in Distributed Git, on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.



Once the maintainer makes this comment, the person who opened the Pull Request will get a notification. We'll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:



Anyone can also leave general comments on the Pull Request. In Pull Request discussion page we can see an example of the project owner both commenting on a line of code and then leaving a general comment in the discussion section. You can see that the code comments are brought into the conversation as well.



The other thing you'll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a "non-fast-forward" merge, meaning that even if the merge **could be a fast-forward, it will still** create a merge commit.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

1.27.5 ADVANCED PULL REQUESTS

Now that we've covered the basics of contributing to a project on GitHub, let's cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

1.27.6 PULL REQUESTS AS PATCHES

It's important to understand that many projects don't really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by Merging.

Pushing the "Merge" button on the site purposefully creates a merge commit that references the Pull Request so that it's easy to go back and research the original conversation if necessary.

1.27.7 KEEPING UP WITH UPSTREAM

If your Pull Request becomes out of date or otherwise doesn't merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.



This pull request contains merge conflicts that must be resolved.
Only those with [write access](#) to this repository can merge pull requests.



Fig 1.52 Pull request does not merge cleanly

\$ git remote add upstream https://github.com/schacon/blink ①

\$ git fetch upstream ②

\$ git merge upstream/master ③

\$ vim blink.ino ④

\$ git push origin slow-blink ⑤

Counting objects: 6, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.

Total 6 (delta 2), reused 0 (delta 0)



Fig 1.54 An example of Github Flavored Markdown as written and as rendered

Task Lists The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

- Write the code
- Write all the tests
- Document the code

If we include this in the description of our Pull Request or Issue, we'll see it rendered like Task lists rendered in a Markdown comment.



Fig 1.55 Task lists rendered in a Markdown comment

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment — you don't have to edit the Markdown directly to check tasks off.

GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch.



Fig 1.56 Task list summary in the Pull Request list

This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

If you add a language name like we did there with 'java', GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like Rendered fenced code example.



Fig 1.57 Rendered fenced code example

1.27.8 QUOTING

If you are responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the > character.

In fact, this is so common and so useful that there is a keyboard shortcut for it.

If you highlight text in a comment that you want to directly reply to and hit the r key, it will quote that text in the comment box for you.

The quotes look something like this:

> Whether 'tis Nobler in the mind to suffer

> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

Once rendered, the comment will look like Rendered quoting example.

1.27.9 Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a `:` character, an autocompleter will help you find what you're looking for.

Emojis take the form of `:<name>`: anywhere in the comment. For instance, you could write something like this:

I `:eyes:` that `:bug:` and I `:cold_sweat:`.

`:trophy:` for `:microscope:` it.

`:+1:` and `:sparkles:` on this `:ship:`, it's `:fire::poop:!`

`:clap::tada::panda_face:`

1.27.10 IMAGES

This isn't technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.

If you look at Drag and drop images to upload them and auto-embed them, you can see a small "Parsed as Markdown" hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

\$ git checkout master ①

\$ git pull https://github.com/progit/progit2.git ②

\$ git push origin master ③ ① If you were on another branch, return to master. ② Fetch changes from <https://github.com/progit/progit2.git> and merge them into master. ③ Push your master branch to origin.

This works, but it is a little tedious having to spell out the fetch URL every time. You can automate this work with a bit of configuration:

```
$ git remote add progit https://github.com/progit/progit2.git ①
```

```
$ git fetch progit ②
```

```
$ git branch --set-upstream-to=progit/master master ③
```

\$ git config --local remote.pushDefault origin ④ ① Add the source repository and give it a name. Here, I have chosen to call it progit. ② Get a reference on progit's branches, in particular master. ③ Set your master branch to fetch from the progit remote.

④ Define the default push repository to origin.

Once this is done, the workflow becomes much simpler:

```
$ git checkout master ①
```

```
$ git pull ②
```

```
$ git push ③
```

① If you were on another branch, return to master.

② Fetch changes from progit and merge changes into master.

③ Push your master branch to origin.

1.28 MAINTAINING A PROJECT

Now that we're comfortable contributing to a project, let's look at the other side: creating, maintaining and administering your own project.

1.28.1 CREATING A NEW REPOSITORY

Let's create a new repository to share our project code with. Start by clicking the

"New repository" button on the right-hand side of the dashboard, or from the + button in the top toolbar next to username as seen in The "New repository" dropdown.



This takes you to the “new repository” form:

A screenshot of the GitHub 'New repository' form. The 'Owner' field is set to 'ben' and the 'Repository name' field is set to 'iOSApp', with a green checkmark indicating it's valid. Below these fields, there's a tip: 'Great repository names are short and memorable. Need inspiration? How about drunken-dubstep.' The 'Description (optional)' field contains the text 'iOS project for our mobile group'. The 'Visibility' section has two radio buttons: 'Public' (selected) and 'Private'. The 'Initialize this repository with a README' checkbox is checked. Below this, there are two dropdown menus for 'Add a gitignore' and 'Add a license', both set to 'None'. At the bottom, there is a green 'Create repository' button.

All you really have to do here is provide a project name; the rest of the fields are completely optional. For now, just click the “Create Repository” button, and boom – you have a new repository on GitHub, named <user>/<project_name>.

Now that your project is hosted on GitHub, you can give the URL to anyone you want to share your project with. Every project on GitHub is accessible over HTTPS as https://github.com/<user>/<project_name>, and over SSH as ssh://github.com:<user>/<project_name>.

1.28.2 ADDING COLLABORATORS

If you're working with other people who you want to give commit access to, you need to add them as "collaborators". If Ben, Jeff, and Louise all sign up for accounts on GitHub, and you want to give them push access to your repository, you can add them to your project. Doing so will give them "push" access, which means they have both read and write access to the project and Git repository.

Click the "Settings" link at the bottom of the right-hand sidebar.

Then select "Collaborators" from the menu on the left-hand side. Then, just type a username into the box, and click "Add collaborator." You can repeat this as many times as you like to grant access to everyone you like. If you need to revoke access, just click the "X" on the right-hand side of their row.

1.28.3 MANAGING PULL REQUESTS

Now that you have a project with some code in it and maybe even a few collaborators who also have push access, let's go over what to do when you get a Pull Request yourself.

Pull Requests can either come from a branch in a fork of your repository or they can come from another branch in the same repository. The only difference is that the ones in a fork are often from people where you can't push to their branch and they can't push to yours, whereas with internal Pull Requests generally both parties can access the branch.

1.28.4 EMAIL NOTIFICATIONS

There are a few things to notice about this email. It will give you a small diffstat — a list of files that have changed in the Pull Request and by how much. It gives you a link to the Pull Request on GitHub. It also gives you a few URLs that you can use from the command line.

If you notice the line that says `git pull <url> patch-1`, this is a simple way to merge in a remote branch without having to add a remote. We went over this quickly in Checking Out Remote Branches.

\$ curl https://github.com/tonychacon/fade/pull/1.patch | git am

1.28.5 COLLABORATING ON THE PULL REQUEST

As we covered in The GitHub Flow, you can now have a conversation with the person who opened the Pull Request. You can comment on specific lines of code, comment on whole commits or comment on the entire Pull Request itself, using GitHub Flavored Markdown everywhere. Every time someone else comments on the Pull Request you will continue to get email notifications so you know there is activity happening.

If you decide you don't want to merge it, you can also just close the Pull Request and the person who opened it will be notified.

1.28.6 PULL REQUEST REFS

If you're dealing with a lot of Pull Requests and don't want to add a bunch of remotes or do onetime pulls every time, there is a neat trick that GitHub allows you to do. This is a bit of an advanced trick and we'll go over the details of this a bit more in The Refspec, but it can be pretty useful.

1.28.7 PULL REQUESTS ON PULL REQUESTS

Not only can you open Pull Requests that target the main or master branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request.

If you see a Pull Request that is moving in the right direction and you have an idea for a change that depends on it or you're not sure is a good idea, or you just don't have push access to the target branch, you can open a Pull Request directly to it.

1.28.8 MENTIONS AND NOTIFICATIONS

GitHub also has a pretty nice notifications system built in that can come in handy when you have questions or need feedback from specific individuals or teams.

In any comment you can start typing a @ character and it will begin to autocomplete with the names and usernames of people who are collaborators or contributors in the project.

1.28.9 Email Notifications

Email notifications are the other way you can handle notifications through GitHub. If you have this turned on you will get emails for each notification. We saw examples of this in Comments sent as email notifications and Email notification of a new Pull Request. The emails will also be threaded properly, which is nice if you're using a threading email client.

There is also a fair amount of metadata embedded in the headers of the emails that GitHub sends you, which can be really helpful for setting up custom filters and rules.

1.28.10 SPECIAL FILES

There are a couple of special files that GitHub will notice if they are present in your repository.

README

The first is the README file, which can be of nearly any format that GitHub recognizes as prose. For example, it could be README, README.md, README.asciidoc, etc. If GitHub sees a README file in your source, it will render it on the landing page of the project.

Many teams use this file to hold all the relevant project information for someone who might be new to the repository or project. This generally includes things like:

- What the project is for
- How to configure and install it
- An example of how to use it or get it running
- The license that the project is offered under
- How to contribute to it

Since GitHub will render this file, you can embed images or links in it for added ease of understanding.

1.28.11 CONTRIBUTING

The other special file that GitHub recognizes is the CONTRIBUTING file. If you have a file named CONTRIBUTING with any file extension, GitHub will show Opening a Pull Request when a CONTRIBUTING file exists when anyone starts opening a Pull Request.

The idea here is that you can specify specific things you want or don't want in a Pull Request sent to your project. This way people may actually read the guidelines before opening the Pull Request.

1.28.12 Project Administration

Generally there are not a lot of administrative things you can do with a single project, but there are a couple of items that might be of interest.

CHANGING THE DEFAULT BRANCH

If you are using a branch other than "master" as your default branch that you want people to open Pull Requests on or see by default, you can change that in your repository's settings page under the "Options" tab.

Simply change the default branch in the dropdown and that will be the default for all major operations from then on, including which branch is checked out by default when someone clones the repository.

TRANSFERRING A PROJECT

If you would like to transfer a project to another user or an organization in GitHub, there is a "Transfer ownership" option at the bottom of the same "Options" tab of your repository settings page that allows you to do this.

This is helpful if you are abandoning a project and someone wants to take it over, or if your project is getting bigger and want to move it into an organization.

1.29 SCRIPTING GITHUB

So now we've covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

1.29.1 SERVICES AND HOOKS

The Hooks and Services section of GitHub repository administration is the easiest way to have GitHub interact with external systems.

1.29.2 SERVICES

First we'll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the "Webhooks and Services" tab you will see something like Services and Hooks configuration section.

In this case, if we hit the "Add service" button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

1.29.3 HOOKS

If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the "Add webhook" button in Services and Hooks configuration section. This will bring you to a page that looks like Web hook configuration.

The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret key and hit "Add webhook". There are a few options for which events you want GitHub to send you a payload for — the default is to only get a payload for the push event, when someone pushes new code to any branch of your repository.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at <https://developer.github.com/webhooks/>.

1.29.4 THE GITHUB API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly anything you can do on the website in an automated fashion. In this section we'll learn how to authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

BASIC USAGE

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
```

1.29.5 COMMENTING ON AN ISSUE

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate the from the "Applications" tab of your settings page.

1.29.6 CHANGING THE STATUS OF A PULL REQUEST

There is one final example we'll look at since it's really useful if you're working with Pull Requests.

Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed any number of things.

1.29.7 OCTOKIT

Though we've been doing nearly everything through curl and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way.

The supported languages include Go, Objective-C, Ruby, and .NET. Check out <https://github.com/octokit> for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out <https://developer.github.com>.



GIT and GITHUB Commands - Working in Local Repository

1. `cd`
2. `cd /c`
3. `cd git`
4. `cd RMD`
5. `git config --global user.name "suhasiniselvam"`
6. `git config --global user.email suhasiniselvam13@gmail.com`
7. For initializing: **git init**
8. To create a file in a folder: **touch file.txt**
9. To edit the newly created text file: **vim file.txt**

After this command, press insert button to add the text in the file. Once the adding of text is done press **esc** and type **:wq**

1. Now you can check the status of the git by using the command: **git status**
2. For adding the new file which created use the command: **git add file.txt**
3. Check whether it is added in your local git repository: **git status**
4. Now, to commit the newly add file use the command: **git commit -m "First Commit" file.txt**
5. Again check the status, **git status**
6. To check the changes done use the command: **git log**

Working Remote Repository

1. Go to github account, create a new repository in your own account.
2. Now cloning has to be done between local and remote repository: **git clone URL**
3. After creating the repository, now we want to push the file which is created in local system using the command: **git push -u origin.**
4. Now u can check the remote repository the new file we get added.
5. If you want to make changes in file which is present in remote you can do by click edit option and for storing the changes in your local repository you have the pull the files from remote. **git pull URL.** Likewise, you can make changes in your local file also and push the changes to remote by **add, commit and push** commands.

Git Branching

1. Initially, check any branch is available using: **git branch** it will display the branch list
2. To create a new branch: **git branch branchname (eg: git branch develop)**
3. To switch from one branch to another use: **git checkout develop**
4. To do creation of new and switching of branch use the command: **git checkout -b features**
5. Now, you can add files, edit files using the command **touch and vim.**
6. For merging the two branch use: **git merge develop features**
7. For deleting the branch: **git branch -d develop**

Git Rebase: (Alternate for Merging)

1. Git rebase is used to merge the two branch only after few commits, to do that use: **git rebase main features**

10. ASSIGNMENT : UNIT – I

(CO1 CO2, K3 K6)

Infosys Springboard Course :-

Software Engineering and Agile Software Development

Link to access the course:

https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013382690411003904735_shared/overview



PART A Q&A UNIT - I

| S
N
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|-------------|--|-------------|-----------|
| 1 | List the goals of software engineering?
Satisfy user requirements , High reliability , Low maintenance cost , Delivery on time , Low production cost , High performance , Ease of reuse. | CO1 | K3 |
| 2 | What is the difference between verification and validation?
Verification refers to the set of activities that ensure that software correctly implements a specific function.
Verification: "Are we building the product right?"
Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Validation: "Are we building the right product?" | CO1 | K3 |
| 3 | What are the two types of software products?
1. Generic products: these are stand-alone systems that are produced by a development Organization and sold in the open market to any customer who wants to buy it.
2. Customized products: these are systems that are commissioned by a specific customer and developed specially by some contractor to meet a special need. | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO3 | K3,
K6 |
|---------|--|-------------|-----------|
| 4 | What is the advantage of adhering to life cycle models for software?
It helps to produce good quality software products without time and cost over runs. It encourages the development of software in a systematic & disciplined manner. | CO1 | K3 |
| 5 | What is software process? List its activities.
Software process is defined as the structured set of activities that are required to develop the software system.
Activities – Specification, design & implementation, validation & evolution. | CO1 | K3 |
| 6 | What does Verification represent?
Verification represents the set of activities that are carried out to confirm that the software correctly implements the specific functionality. | CO1 | K3 |
| 7 | What does Validation represent?
Validation represents the set of activities that ensure that the software that has been built is satisfying the customer requirements. | CO1 | K3 |
| 8. | What is meant by Software engineering paradigm?
The development strategy that encompasses the process, methods and tools and generic phases is often referred to as a process model or software engineering paradigm. | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO3 | K3,
K6 |
|---------|--|-------------|-----------|
| 9 | <p>Define agility and agile team.</p> <p>Agility-Effective (rapid and adaptive) response to change (team members, new technology, requirements)</p> <p>Effective communication in structure and attitudes among all team members, technological and business people, software engineers and managers.</p> <p>Drawing the customer into the team. Eliminate "us and them" attitude.</p> <p>Planning in an uncertain world has its limits and plan must be flexible.</p> <p>Organizing a team so that it is in control of the work performed</p> <p>The development guidelines stress delivery over analysis and design although these activates are not discouraged, and active and continuous Communication between developers and customers</p> <p>Eliminate all but the most essential work products and keep them lean.</p> <p>Emphasize an incremental delivery strategy as opposed to intermediate products that gets working software to the customer as rapidly as feasible.</p> | CO1 | K3 |
| 10 | <p>Write any two characteristics of software as a product.</p> <p>1. Software is developed or engineered, it is not manufactured in the classical sense</p> <p>2. Software doesn't "wear out."</p> <p>3. Although the industry is moving toward component-based assembly, most software continues to be custom built.</p> | CO1 | K3 |
| 11 | <p>Write the IEEE definition of software engineering.</p> <p>According to IEEE's definition software engineering can be defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.</p> | CO1 | K3 |
| 12 | <p>List two deficiencies in waterfall model . Which process model do you suggest to overcome each deficiency.</p> <p>Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.</p> <p>No working software is produced until late during the life cycle.</p> | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO3 | K3,
K6 |
|---------|---|-------------|-----------|
| 13 | <p>What is Agile?</p> <p>The word 'agile' means –</p> <p>Able to move your body quickly and easily.</p> <p>Able to think quickly and clearly.</p> <p>In business, 'agile' is used for describing ways of planning and doing work wherein it is understood that making changes as needed is an important part of the job. Business 'agility' means that a company is always in a position to take account of the market changes.</p> <p>In software development, the term 'agile' is adapted to mean 'the ability to respond to changes – changes from Requirements, Technology and People.'</p> | CO1 | K3 |
| 14 | <p>What is Agile Manifesto?</p> <p>The Agile Manifesto states that –We are uncovering better ways of developing software by doing it and helping others do it.</p> <p>Through this work, we have come to value –</p> <p>Individuals and interactions over processes and tools.</p> <p>Working software over comprehensive documentation.</p> <p>Customer collaboration over contract negotiation.</p> <p>Responding to change over following a plan.</p> <p>That is, while there is value in the items on the right, we value the items on the left more</p> | CO1 | K3 |
| 15 | <p>What are the Characteristics of Agility?</p> <p>Agility in Agile Software Development focuses on the culture of the whole team with multi-discipline,cross-functional teams that are empowered and self organizing.</p> <p>It fosters shared responsibility and accountability.</p> <p>Facilitates effective communication and continuous collaboration.</p> <p>The whole-team approach avoids delays and wait times.</p> <p>Frequent and continuous deliveries ensure quick feedback that in in turn enable the team align to the requirements.</p> <p>Collaboration facilitates combining different perspectives timely in implementation, defect fixes and accommodating changes</p> | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|--|-------------|-----------|
| 16 | <p>What are the principles of agile methods?</p> <p>Customer involvement Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the System.</p> <p>Incremental delivery
The software is developed in increments with the customer specifying the requirements to be included in each increment.</p> <p>People not process
The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.</p> <p>Embrace change
Expect the system requirements to change and so design the system to accommodate these changes.</p> <p>Maintain simplicity
Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.</p> | CO1 | K3 |
| 17 | <p>What are the Problems with agile methods?</p> <p>It can be difficult to keep the interest of customers who are involved in the process.</p> <p>Team members may be unsuited to the intense involvement that characterizes agile methods.</p> <p>Prioritizing changes can be difficult where there are multiple stakeholders.</p> <p>Maintaining simplicity requires extra work.</p> <p>Contracts may be a problem as with other approaches to iterative development.</p> | CO1 | K3 |

| SNO | QUESTION AND ANSWERS | CO | K |
|-----|--|-----|----|
| 18 | <p>What is Extreme Programming?</p> <p>XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software. Extreme Programming (XP) was conceived and developed to address the specific needs of software development by small teams in the face of vague and changing requirements.</p> <p>Extreme Programming is one of the Agile software development methodologies. It provides values and principles to guide the team behavior. The team is expected to self-organize.</p> <p>Extreme Programming provides specific core practices where –</p> <p>Each practice is simple and self-complete.</p> <p>Combination of practices produces more complex and emergent behavior.</p> | CO1 | K3 |
| 19 | <p>HOW Embrace Change happens in Extreme programming?</p> <p>A key assumption of Extreme Programming is that the cost of changing a program can be held mostly constant over time.</p> <p>This can be achieved with –</p> <p>Emphasis on continuous feedback from the customer</p> <p>Short iterations</p> <p>Design and redesign</p> <p>Coding and testing frequently</p> <p>Eliminating defects early, thus reducing costs</p> <p>Keeping the customer involved throughout the development</p> <p>Delivering working product to the customer.</p> | CO1 | K3 |
| 20 | <p>How Extreme Programming used in a Nutshell?</p> <p>Extreme Programming involves –</p> <p>Writing unit tests before programming and keeping all of the tests running at all times. The unit tests are automated and eliminate defects early, thus reducing the costs.</p> <p>Starting with a simple design just enough to code the features at hand and redesigning when required.</p> <p>Programming in pairs (called pair programming), with two programmers at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.</p> <p>Integrating and testing the whole system several times a day.</p> | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|---|-------------|-----------|
| 21 | <p>Why is it called "Extreme?"</p> <p>Extreme Programming takes the effective principles and practices to extreme levels.</p> <p>Code reviews are effective as the code is reviewed all the time.</p> <p>Testing is effective as there is continuous regression and testing.</p> <p>Design is effective as everybody needs to do refactoring daily.</p> <p>Integration testing is important as integrate and test several times a day.</p> <p>Short iterations are effective as the planning game for release planning and iteration planning.</p> | CO1 | K3 |
| 22 | <p>What are the Extreme Programming Advantages?</p> <p>Extreme Programming solves the following problems often faced in the software development projects –</p> <p>Slipped schedules – and achievable development cycles ensure timely deliveries.</p> <p>Cancelled projects – Focus on continuous customer involvement ensures transparency with the customer and immediate resolution of any issues.</p> <p>Costs incurred in changes – Extensive and ongoing testing makes sure the changes do not break the existing functionality. A running working system always ensures sufficient time for accommodating changes such that the current operations are not affected.</p> <p>Production and post-delivery defects: Emphasis is on – the unit.</p> | CO1 | K3 |
| 23 | <p>What is Scrum?</p> <p>The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific agile practices. There are three phases in Scrum:</p> <p>1.The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.</p> <p>2.This is followed by a series of sprint cycles, where each cycle develops an increment of the system.</p> | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|---|-------------|-----------|
| | 3. The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project. | | |
| 24 | <p>What are the Advantages of scrum?</p> <p>The product is broken down into a set of manageable and understandable chunks.</p> <p>Unstable requirements do not hold up progress.</p> <p>The whole team has visibility of everything and consequently team communication is improved.</p> <p>Customers see on-time delivery of increments and gain feedback on how the product works.</p> <p>Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.</p> | CO1 | K3 |
| 25 | <p>Mention the Two perspectives on scaling of agile methods?</p> <p>1. Scaling up</p> <p>2. Scaling out</p> | CO1 | K3 |
| 26 | <p>What is Scaling up?</p> <p>Using agile methods for developing large software systems that cannot be developed by a small team. For large systems development, it is not possible to focus only on the code of the system; you need to do more up- front design and system documentation. Cross-team communication mechanisms have to be designed and used, which should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible; however, it is essential to maintain frequent system builds and regular releases of the system.</p> | CO1 | K3 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|--|-------------|-----------|
| 27 | What is Scaling out?
How agile methods can be introduced across a large organization with many years of software development experience. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities. | CO1 | K3 |
| 28 | What is agile development?
Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process. Projects include elements of plan-driven and agile processes. Deciding on the balance depends on many technical, human, and organizational issues. | CO1 | K3 |
| 29 | What is Scrum master?
The role of the Scrum Master is to protect the development team from external distractions. At the end of the sprint the work done is reviewed and presented to stakeholders (including the product owner). | CO1 | K3 |
| 30 | List the types of Version Control System.
Local Version Control System
Centralized Version Control System
Distributed Version Control System | CO2 | K6 |
| 31 | Write a command that initialize the repository in the existing directory?
For Starting the existing project, run the command from the project directory
\$ git init | CO2 | K6 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|---|-------------|-----------|
| 32 | How to get a copy of an existing Git repository?
The command need is git clone. clone a repository with git clone [url].
For example, to clone the Git linkable library called libgit2, use the format given below:
\$ git clone https://github.com/libgit2/libgit2 | CO2 | K6 |
| 33 | Define the two states of the file in the working directory.
Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.
Untracked files are everything else – any files in the working directory that were not in last snapshot and are not in staging area. | CO2 | K6 |
| 34 | Define and Mention the two types of tags in tagging.
Git uses two main types of tags: - Lightweight, Annotated
A lightweight tag is very much like a branch that doesn't change, it is just a pointer to a specific commit.
Annotated tags are stored as full objects in the Git database. | CO2 | K6 |
| 35 | Write the command for adding the remote repositories.
To add a new remote Git repository as a shortname that can reference easily, run git remote add <shortname> <url>:
\$ git remote
origin
\$ git remote add pb https://github.com/paulboone/ticgit
\$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push) | CO2 | K6 |
| 36 | What is the command used to unmodifying the modified file?
Use git checkout option, for unmodifying the changes made.
\$ git checkout -- CONTRIBUTING.md | CO2 | K6 |
| 37 | How to install github in linux?
By using the command, the github can be installed.
\$ sudo yum install git-all (or) \$ sudo apt-get install git-all | CO2 | K6 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|---|-------------|-----------|
| 38 | What is a branch in Git? <ul style="list-style-type: none"> A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. When making commits, a master branch will be created that points to the last commit made. When commit is done, the master branch pointer moves forward automatically. The "master" branch in Git is not a special branch. It is exactly like any other branch. | CO2 | K6 |
| 39 | How to resolve a conflict in Git?
The following steps will resolve conflict in Git-
Identify the files that have caused the conflict.
Make the necessary changes in the files so that conflict does not arise again.
Add these files by the command git add.
Finally to commit the changed file using the command git commit. | CO2 | K6 |
| 40 | What is the difference between git pull and git fetch?
Git pull command pulls new changes or commits from a particular branch from your central repository and updates your target branch in your local repository.
Git fetch is also used for the same purpose but it works in a slightly different way. When you perform a git fetch, it pulls all new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, git fetch must be followed with a git merge. Your target branch will only be updated after merging the target branch and fetched branch. Just to make it easy for you, remember the equation below:
Git pull = git fetch + git merge | CO2 | K6 |
| 41 | What work is restored when the deleted branch is recovered?
The files which were stashed and saved in the stash index list will be recovered back. Any untracked files will be lost. Also, it is a good idea to always stage and commit your work or stash them. | CO2 | K6 |
| 42 | What is the function of 'git stash apply'?
If you want to continue working where you had left your work then 'git stash apply' command is used to bring back the saved changes onto your current working directory. | CO2 | K6 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|--|-------------|-----------|
| 43 | What is the difference between the 'git diff 'and 'git status'?
'git diff ' depicts the changes between commits, commit and working tree, etc. whereas 'git status' shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively. 'git diff' is similar to 'git status', the only difference is that it shows the differences between various commits and also between the working directory and index. | CO2 | K6 |
| 44 | What is the difference between 'git remote' and 'git clone'?
'git remote add' creates an entry in your git config that specifies a name for a particular URL whereas 'git clone' creates a new git repository by copying an existing one located at the URL. | CO2 | K6 |
| 45 | How will you know in Git if a branch has already been merged into master?
The answer is pretty direct.
To know if a branch has been merged into master or not you can use the below commands:
git branch --merged – It lists the branches that have been merged into the current branch.
git branch --no-merged – It lists the branches that have not been merged. | CO2 | K6 |
| 46 | Define github.
GitHub is the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things. | CO2 | K6 |
| 47 | Elaborate Two Factor Authentication.
Two-factor Authentication or "2FA". Two_factor Authentication is an authentication mechanism that is becoming more and more popular recently to mitigate the risk of your account being compromised if your password is stolen somehow. Turning it on will make GitHub ask you for two different methods of authentication, so that if one of them is compromised, an attacker will not be able to access your account. | CO2 | K6 |

| SN
O | QUESTION AND ANSWERS | CO1,
CO2 | K3,
K6 |
|---------|---|-------------|-----------|
| 48 | Define Scripting GitHub.
covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.
Luckily for us, GitHub is really quite hackable in many ways. | CO2 | K6 |
| 49 | Define Pull Requests on Pull Requests.
Not only can you open Pull Requests that target the main or master branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request. | CO2 | K6 |
| 50 | Define Creating a New Repository.
create a new repository to share our project code with. Start by clicking the "New repository" button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your
username as seen in The "New repository" dropdown. | CO2 | K6 |



12. PART B QUESTIONS : UNIT – I

(CO1,CO2 – K3,K6)

1. Discuss the various life cycle models in software development?
2. List the principles of Agile software development?
3. Define XP values and write in detail about Extreme Programming?
4. What is the difference between XP and industrial XP? Discuss Industrial Extreme Programming.
5. Explain adaptive software development and Scrum.
6. Explain Dynamic systems development method and Crystal.
7. Explain feature drive development and lean software development.
8. Explain agile modeling and agile unified process.
9. Describe in detail about the version control system and its types with diagram.
10. Write the procedure for installing github in various platform.
11. Explain about the need of recording changes in the repository.
12. Elaborate the procedure for viewing the commit history.
13. Discuss the different ways of undoing things.
14. Explain in detail about the Git Aliases.
15. Produce what is tagging and its types in detail.
16. Write the procedure for accessing and working remote repositories.
17. Explain how to contribute to a project through forking.
18. Briefly explain how to maintaining a project using GitHub .
19. Explain in detail about Git Branching and Management.
20. Discuss the branching workflows in detail.
21. Write in detail about Remote Branches.
22. What is Rebasing? Discuss in detail.

PART C (CO1,CO2 – K3,K6)

1. Case Study for Uberization Project – In-plant Vehicle movement
Business Requirement:- Need to minimize the use of vehicle in the plant area. Usage of the vehicle is for movement of logistics and to give a critical support in un-planned shut down.Need online booking of vehicles as mobile app
2. Case Study for Rack monitoring
Business Requirement:- Need to create a GPS tracker which can be plotted on the railways racks to maximize the utilization and monitor the same
3. Case Study to print a online visitor diary after retirement
Business Requirement:- Need to create visitor diary for retired person and on submission of I-card and other belongings , system will auto issue a printed visitor diary to enter in the office premises
4. Case Study for Online display of OEE for all Plants
Business Requirement:- Need to create dashboard which will display the OEE for all plants on monthly basis and trends for entire year and provide a facility to display the loop holes for reduced OEE(Overall equipment effectiveness)
5. Explain about the following:
 - a. Set up and Configure the Project in GitHub.
 - b. Scripting in GitHub.

13. SUPPORTIVE ONLINE CERTIFICATION COURSES

UNIT I

COURSERA

<https://www.coursera.org/professional-certificates/devops-and-software-engineering>
<https://www.coursera.org/learn/introduction-to-software-engineering>
<https://www.coursera.org/learn/introduction-git-github>
<https://www.coursera.org/learn/getting-started-with-git-and-github>
https://www.coursera.org/learn/agile-project-management?action=enroll&adgroupid=137976342650&adpostion=&campaignid=18216928758&creativeid=630416489407&device=c&device_model=&hide_mobile_promo=&keyword=&matchtype=&network=g&utm_campaign=B2C_INDIA_branded_FTCOF_courseraplus_arte&utm_content=B2C&utm_medium=sem&utm_source=gg

UDEMY

<https://www.udemy.com/course/complete-professional-scrum-master-training-exam-simulator/?src=sac&kw=AGILE+S>

UDEMY

<https://www.udemy.com/topic/Software-Engineering>

<https://www.udemy.com/course/github-ultimate/>

NPTEL

<https://nptel.ac.in/courses/106105182>

14. REAL TIME APPLICATIONS : UNIT – I

SRS Document for Purchasing Gifts Online

Objective:

Amaze Pack is an online store for purchasing gift items.

Users of the System:

1. Admin
2. Customer

Functional Requirements:

- Build an application that customers can access and purchase gifts online.
- The application should have signup, login, profile, dashboard page, and product page.
- This application should have a provision to maintain a database for customer information, order information and product portfolio.
- Also, an integrated platform required for admin and customer.
- Administration module to include options for adding / modifying / removing the existing product(s) and customer management.
- There should be a maximum of 5 products per cart.

While the above ones are the basic functional features expected, the below ones can be nice to have add-on features:

- Filters for products like Low to High or showcasing products based on the customer's price range, specific brands etc.
- Email integration for intimating new personalized offers to customers.
- Multi-factor authentication for the sign-in process

•Payment Gateway

Output/ Post Condition:

- Records Persisted in Success & Failure Collections
- Standalone application / Deployed in an app Container

Non-Functional Requirements:

Security

- App Platform –User Name/Password-Based Credentials
- Sensitive data has to be categorized and stored in a secure manner
- Secure connection for transmission of any data

Performance

- Peak Load Performance (during Festival days, National holidays etc)
- eCommerce -< 3 Sec
- Admin application < 2 Sec
- Non-Peak Load Performance
- eCommerce < 2 Sec
- Admin Application < 2 Sec

Availability

- 99.99 % Availability
- Scalability
- Maintainability
- Usability
- Availability
- Failover

Logging & Auditing

- The system should support logging(app/web/DB) & auditing at all levels

Monitoring

- Should be able to monitor via as-is enterprise monitoring tools

Cloud

- The Solution should be made Cloud-ready and should have a minimum impact when moving away to Cloud infrastructure.

Browser Compatible

All latest browsers

Technology Stack

Front End Angular 10+/ React 16+

Material Design Bootstrap / Bulma

Server Side Spring Boot / .Net WebAPI/ Node Js

Database MySQL or Oracle or MSSQL

Frontend:

Customer:

1.Auth: Design an auth component (Name the component as auth for angular app whereas Auth for react app. Once the component is created in react app, name the jsx file as same as component name i.e Auth.jsx file) where the customer can authenticate login and signup credentials

2. Signup: Design a signup page component (Name the component as signup for angular app whereas Signup for react app. Once the component is created in react app, name the jsx file as same as component name i.e Signup.jsx file) where the new customer has options to sign up by providing their basic details.

a. Ids:

- email
- username
- mobilenumber
- password
- confirmpassword
- submitButton
- signinLink
- signupBox

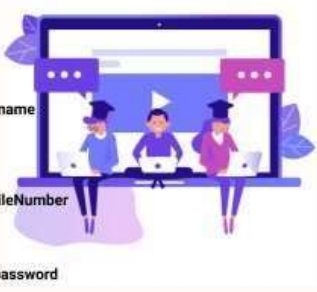
b. API endpoint Url: <http://localhost:8000/signup>

c. Output screenshot:

Register

Sign Up


Already a user? [Login](#)



Output screenshot:

Login

New User? [Sign Up](#)



loginBox

15. CONTENT BEYOND SYLLABUS : UNIT – I

1. Incremental Model is a process of software development where requirements divided into multiple standalone modules of the software development cycle. In this model, each module goes through the requirements, design, implementation and testing phases. Every subsequent release of the module adds function to the previous release. The process continues until the complete system achieved.

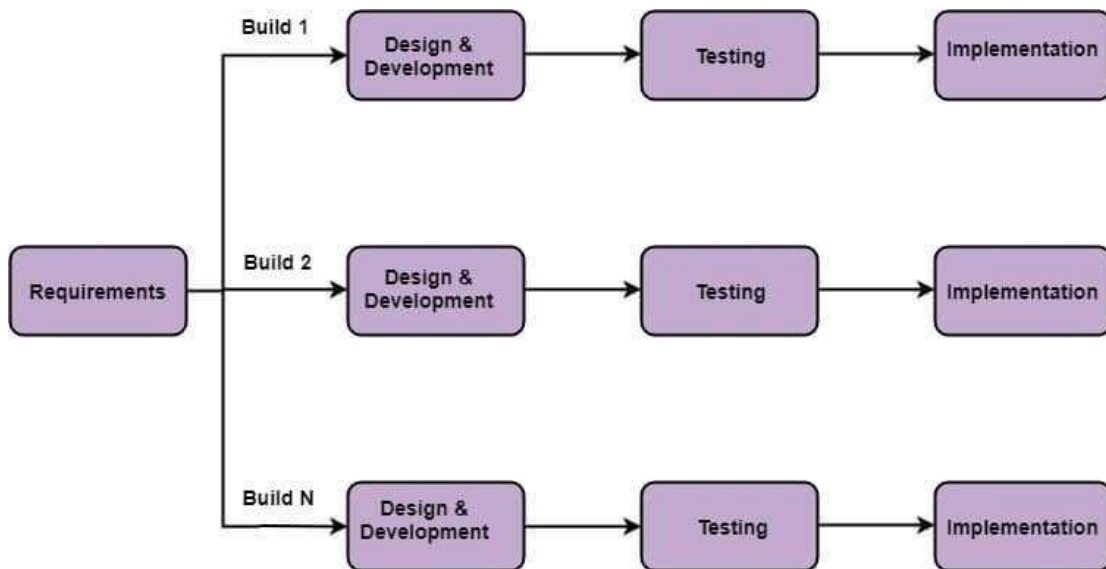


Fig: Incremental Model

The various phases of incremental model are as follows:

1. **Requirement analysis:** In the first phase of the incremental model, the product analysis expertise identifies the requirements. And the system functional requirements are understood by the requirement analysis team. To develop the software under the incremental model, this phase performs a crucial role.
2. **Design & Development:** In this phase of the Incremental model of SDLC, the design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.
3. **Testing:** In the incremental model, the testing phase checks the performance of each existing function as well as additional functionality. In the testing phase, the various methods are used to test the behavior of each task.
4. **Implementation:** Implementation phase enables the coding phase of the development system. It involves the final coding that design in the designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up to the final system product.

When we use the Incremental Model?

- When the requirements are superior.
- A project has a lengthy development schedule.
- When Software team are not very well skilled or trained.
- When the customer demands a quick release of the product.
- You can develop prioritized requirements first.

Advantage of Incremental Model

- Errors are easy to be recognized.
- Easier to test and debug
- More flexible.
- Simple to manage risk because it handled during its iteration.
- The Client gets important functionality early.

Disadvantage of Incremental Model

- Need for good planning
- Total Cost is high.
- Well defined module interfaces are needed.

2. Getting Git on a Server

In order to initially set up any Git server:

Export an existing repository into a new bare repository, a repository that does not contain a working directory. This is generally straightforward to do. In order to clone repository, create a new bare repository and run the clone command with the **--bare** option. By convention, bare repository directories end in **.git**, like so: **\$ git clone --bare my_project my_project.git**

Cloning into bare repository 'my_project.git' done.

Now, have a copy of the Git directory data in my_project.git directory.

\$ cp -Rf my_project/.git my_project.git

There are a couple of minor differences in the configuration file; but for some purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

Putting the Bare Repository on a Server

Set up a protocol on a server with a bare copy of the repository. Now, set up a server called `git.example.com` that have SSH access to, and to store all the Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, now set up the new repository by copying the bare repository over:

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

At this point, other users who have SSH access to the same server which has read-access to the `/srv/git` directory can clone the repository by running

```
$ git clone user@git.example.com:/srv/git/my\_project.git
```

If a user SSHs into a server and has write access to the `/srv/git/ my_project.git` directory, they will also automatically have push access. Git will automatically add group write permissions to a repository properly on running the **git init** command with the **--shared** option.

```
$ ssh user@git.example.com
```

```
$ cd /srv/git/my_project.git
```

```
$ git init --bare --shared
```

See how easy it is to take a Git repository, create a bare version, and place it on a server and have SSH access for collaborators. Now it is ready to collaborate on the same project.

16. ASSESSMENT SCHEDULE

Tentative schedule for the Assessment During 2024-2028 Odd semester

| S.NO | Name of the Assessment | Start Date | End Date | Portion |
|------|------------------------|------------|------------|----------------|
| 1 | Unit Test 1 | - | - | UNIT 1 |
| 2 | IAT 1 | 17.10.2024 | 22.10.2024 | UNIT 1 & 2 |
| 3 | Unit Test 2 | - | - | UNIT 3 |
| 4 | IAT 2 | 23.11.2024 | 28.11.2024 | UNIT 3 & 4 |
| 5 | Revision 1 | - | - | UNIT 5 , 1 & 2 |
| 6 | Revision 2 | - | - | UNIT 3 & 4 |
| 7 | Model | 16.12.2024 | 23.12.2024 | ALL 5 UNITS |

17. PRESCRIBED TEXT BOOKS & REFERENCE BOOKS

TEXT BOOKS

1. Roger S. Pressman, "Software Engineering: A Practitioner's Approach", McGraw Hill International Edition, Ninth Edition, 2020.
2. Scott Chacon, Ben Straub, "Pro GIT", Apress Publisher, 3rd Edition, 2014.
3. Deitel and Deitel and Nieto, "Internet and World Wide Web - How to Program", Pearson, 5th Edition, 2018.

REFERENCE BOOKS

- 1 Roman Pichler, "Agile Product Management with Scrum Creating Products that Customers Love", Pearson Education, 1 st Edition, 2010.
2. Jeffrey C and Jackson, "Web Technologies A Computer Science Perspective", Pearson Education, 2011.
3. Stephen Wynkoop and John Burke, "Running a Perfect Website", QUE, 2nd Edition, 1999.
4. Chris Bates, "Web Programming – Building Intranet Applications", 3rd Edition, Wiley Publications, 2009.
5. Gopalan N.P. and Akilandeswari J., "Web Technology", Second Edition, Prentice Hall of India, 2014.
6. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013382690411003904735_shared/overview
7. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_0130944214274703362099_shared/overview

18. MINI PROJECT SUGGESTION

- SRS Document Preparation and implementation.

Topics suggested:

1. Scholarship Management System for students.
2. Gram Panchayat Website.
3. Blood Donation Website.
4. A Food Wastage Management Website.
5. Organ Donator and Finder Website.
6. Subject Allocation for college Faculty.
7. Seminar Hall Booking Website.
8. College Admission Management.
9. Agri shop: Farmers Online Selling.
10. Online Complaint Registration and Management System: Street Light, Water Pipe Leakage, Rain Water Drainage, Road Problem , etc.



Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.