



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps

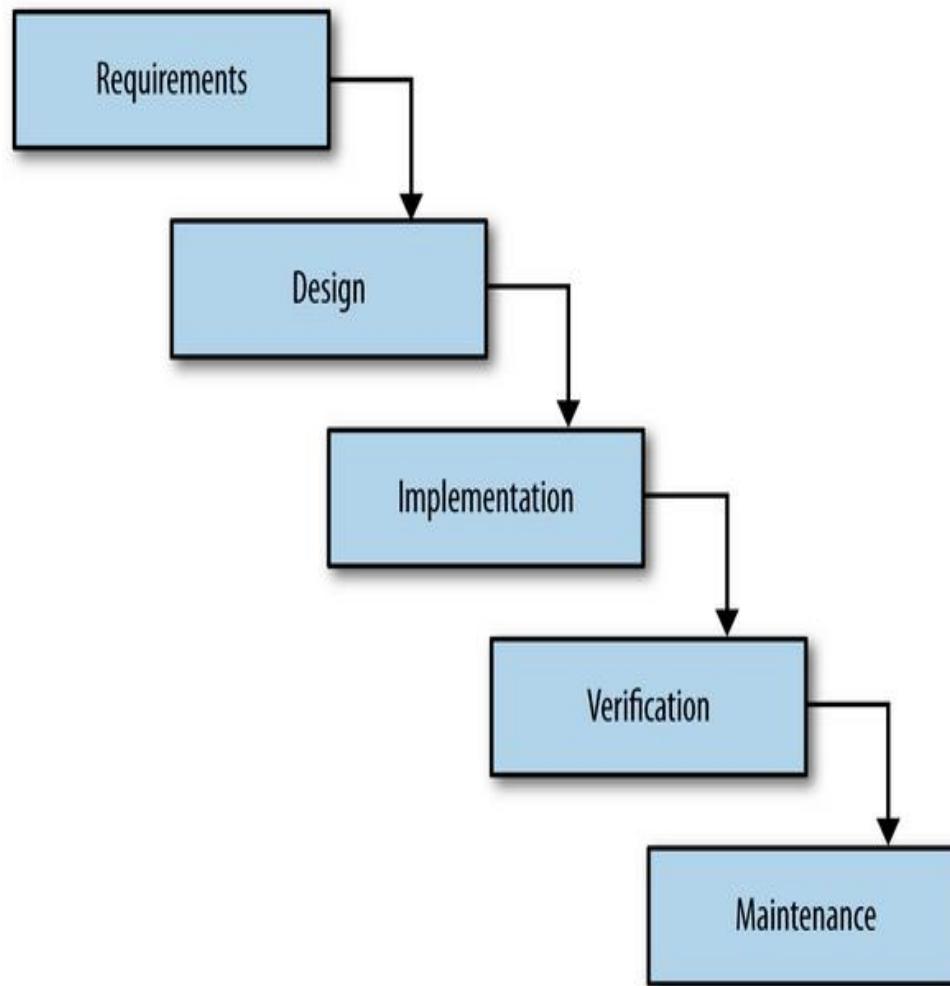




CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 1

Waterfall



Pros of Waterfall

Cons of Waterfall



Agile

- *individuals and interactions* over *processes and tools*
- *working software* over *comprehensive documentation*
- *customer collaboration* over *contract negotiation*
- *responding to change* over *following a plan*



Scrum

- **Scrum**
- In the mid-1990s, Ken Schwaber and Dr. Jeff Sutherland, two of the original creators of the Agile Manifesto, merged individual efforts to present a new software development process called Scrum. Scrum is a software development methodology that focuses on maximizing a development team's ability to quickly respond to changes in both project and customer requirements. It uses predefined development cycles called *sprints*, usually between one week and one month long, beginning with a sprint planning meeting to define goals and ending with a sprint review and sprint retrospective to discuss progress and any issues that arose during that sprint.

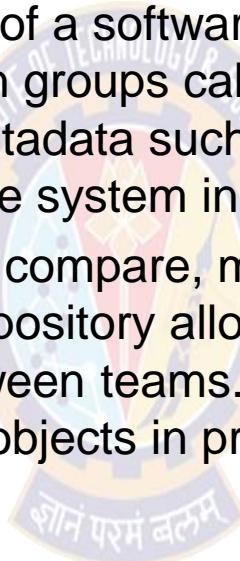
DSM

- What did I do yesterday that helped the team meet its sprint goals?
- What am I planning to do today to help the team meet those goals?
- What, if anything, do I see that is blocking either me or the team from reaching their goals?



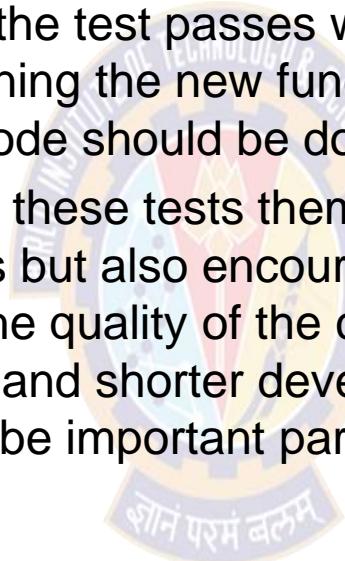
Version Control

- A version control system records changes to files or sets of files stored within the system. This can be source code, assets, and other documents that may be part of a software development project. Developers make changes in groups called *commits* or *revisions*. Each revision, along with metadata such as who made the change and when, is stored within the system in one way or another.
- Having the ability to commit, compare, merge, and restore past revisions to objects to the repository allows for richer cooperation and collaboration within and between teams. It minimizes risks by establishing a way to revert objects in production to previous versions.



Test Driven Development

- In test-driven development, the code developer starts by writing a failing test for the new code functionality, then writes the code itself, and finally ensures that the test passes when the code is complete. The test is a way of defining the new functionality clearly, making more explicit what the code should be doing.
- Having developers write these tests themselves not only greatly shortens feedback loops but also encourages developers to take more responsibility for the quality of the code they are writing. This sharing of responsibility and shorter development cycle time are themes that continue to be important parts of a devops culture.

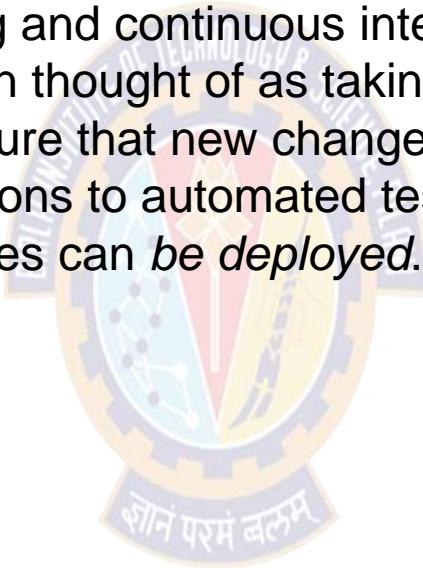


Continuous Integration

- Continuous integration (CI) is the process of integrating new code written by developers with a mainline or “master” branch frequently throughout the day. This is in contrast to having developers working on independent feature branches for weeks or months at a time, merging their code back to the master branch only when it is completely finished. Long periods of time in between merges means that much more has been changed, increasing the likelihood of some of those changes being breaking ones. With bigger changesets, it is much more difficult to isolate and identify what caused something to break. With small, frequently merged changesets, finding the specific change that caused a regression is much easier. The goal is to avoid the kinds of integration problems that come from large, infrequent merges.

Continuous Delivery

- Continuous delivery (CD) is a set of general software engineering principles that allow for frequent releases of new software through the use of automated testing and continuous integration. It is closely related to CI, and is often thought of as taking CI one step further, that beyond simply making sure that new changes can be integrated without causing regressions to automated tests, continuous delivery means that these changes can *be deployed*.

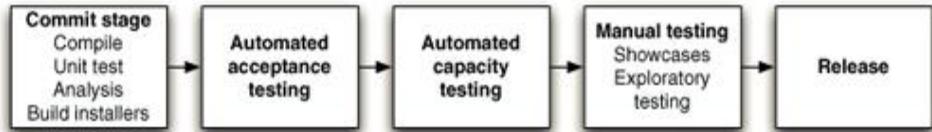


Continuous Deployment

- Continuous deployment (also referred to as CD) is the process of deploying changes to production by defining tests and validations to minimize risk. While continuous delivery makes sure that new changes can be deployed, continuous deployment means that they *get deployed* into production.



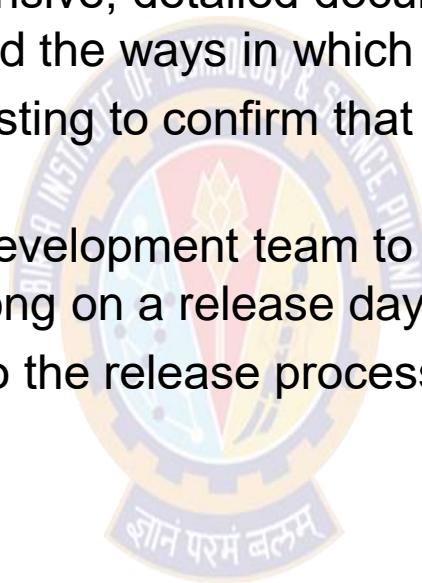
Deployment Pipeline



- The pattern that is central to this book is the *deployment pipeline*. A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process. Every organization will have differences in the implementation of their deployment pipelines, depending on their value stream for releasing software, but the principles that govern them do not vary. An example of a deployment pipeline is given in Figure

Antipattern: Deploying Software Manually

- The signs of this antipattern are:
 - The production of extensive, detailed documentation that describes the steps to be taken and the ways in which the steps may go wrong
 - Reliance on manual testing to confirm that the application is running correctly
 - Frequent calls to the development team to explain why a deployment is going wrong on a release day
 - Frequent corrections to the release process during the course of a release



Antipattern: Deploying to a Production-like Environment Only after Development Is Complete

- If testers have been involved in the process up to this point, they have tested the system on development machines.
- Releasing into staging is the first time that operations people interact with the new release. In some organizations, separate operations teams are used to deploy the software into staging and production. In this case, the first time an operations person sees the software is the day it is released into production.
- Either a production-like environment is expensive enough that access to it is strictly controlled, or it is not in place on time, or nobody bothered to create one.
- The development team assembles the correct installers, configuration files, database migrations, and deployment documentation to pass to the people who perform the actual deployment—all of it untested in an environment that looks like production or staging.

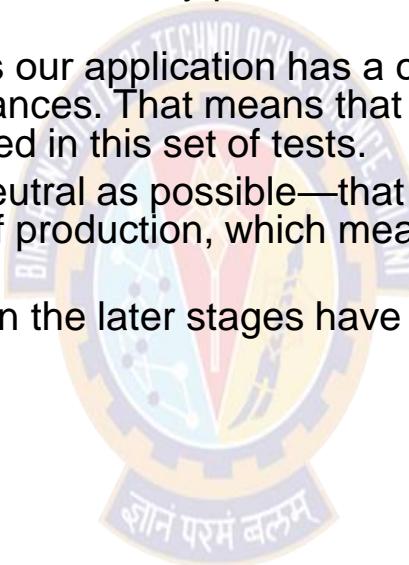
Every Change Should Trigger the Feedback Process

- Executable code changes when a change is made to the source code. Every time a change is made to the source code, the resulting binary must be built and tested. In order to gain control over this process, building and testing the binary should be automated. The practice of building and testing your application on every check-in is known as continuous integration



Possible

- They run fast.
- They are as comprehensive as possible—that is to say, they cover more than 75% or so of the codebase, so that when they pass, we have a good level of confidence that the application works.
- If any of them fails, it means our application has a critical fault and should not be released under any circumstances. That means that a test to check the color of a UI element should not be included in this set of tests.
- They are as environment-neutral as possible—that is, the environment does not have to be an exact replica of production, which means it can be simpler and cheaper.
- On the other hand, the tests in the later stages have the following general characteristics.





Thank You!

Acknowledgement:

- Significant portions of the information in the presentation is from IT Systems Management - Rich Schiesser and other books/Internet. Permission was requested from the publisher, sometime ago for use, but has not been received as yet. This is intended solely to teach BITS students enrolled for IM. I would like to acknowledge and reiterate that all the credit/rights remain with the original authors/publishers only. Mistakes if any are mine.



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps





CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 2 and 3

Agenda

DevOps Misconceptions and Dimensions

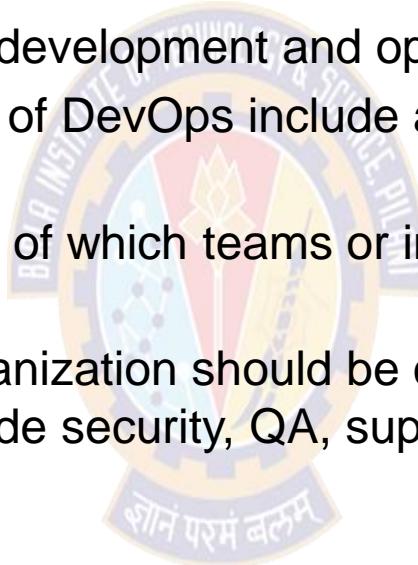
- DevOps Misconceptions
- DevOps Antipatterns
- Three Dimensions of DevOps



DevOps Misconceptions

DevOps Only Involves Developers and System Administrators

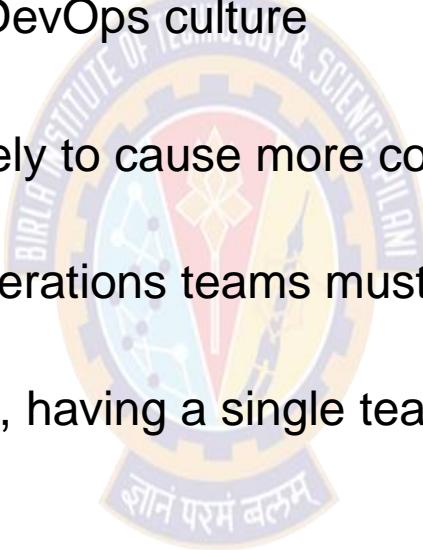
- As the name indicates Development and Operations
- **However the Truth is:**
- DevOps tagline is bring development and operations together
- The concepts and ideas of DevOps include all roles within an organization
- There is no definitive list of which teams or individuals should be involved
- Any team within the organization should be considered as a candidate for DevOps, it may include security, QA, support, and legal



DevOps Misconceptions Contd..

DevOps is a Team

- Creating a team called DevOps, OR
- Renaming an existing team to DevOps is neither necessary nor sufficient for creating a DevOps culture
- **However truth is:**
- An additional team is likely to cause more communication issues, not fewer
- the development and operations teams must connect frequently or it should work together
- In a startup environment, having a single team that encompasses both functions can work



DevOps Misconceptions Contd..

DevOps is a Job Title

- The “DevOps engineer” job title has started a controversial debate
- The job title has been described in various ways
- A system administrator who also knows how to write code
- A developer who knows the basics of system administration
- **However truth is:**
- In DevOps it makes sense to have people become more specialized in their job role
- DevOps is at its core a cultural movement, and its ideas and principles need to be used throughout entire organizations in order to be effective

DevOps Misconceptions Contd..

You Need a DevOps Certification

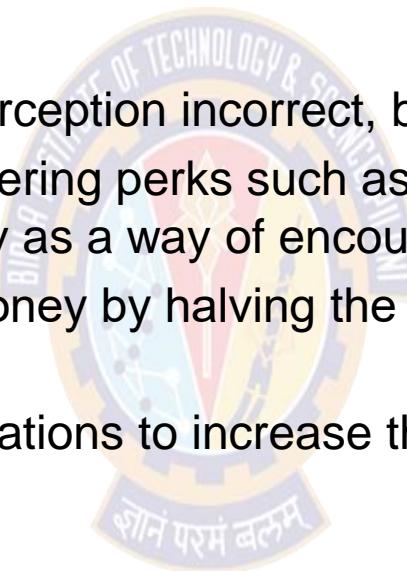
- Certification exams are testing knowledge where there are clear right or wrong answers, which DevOps generally does not have
- **However truth is:**
- A significant part of DevOps is about culture: how do you certify culture?
- DevOps doesn't have required technology or one-size-fits-all solutions



DevOps Misconceptions Contd..

DevOps Means Doing All the Work with Half the People

- Get both a software developer and a system administrator in one person and with one person's salary
- **However truth is:**
- Not only is this above perception incorrect, but it is often harmful
- At a time startups are offering perks such as three meals a day in the office and on-site laundry as a way of encouraging
- DevOps doesn't save money by halving the number of engineers your company needs
- Rather, it allows organizations to increase the quality and efficiency of their work



DevOps Misconceptions Contd..

DevOps Is About Automation

- Many innovations in DevOps-adjacent tools help to codify understanding, bridging the gaps between teams and increasing velocity through automation
- Practitioners have focused on tools that eliminate tasks that are boring and repetitive, such as with infrastructure automation and continuous integration
- **However truth is:**
- In both of these above cases, automation is a result of improved technology
- Automating repetitive tasks automation will help to human from having to do them, that automation helps that person work more efficiently
- Example: Automating server builds saves hours per server that a system administrator can then spend on more interesting or challenging work

DevOps Misconceptions Contd..

DevOps Is a FAD

- It's a Buzzword in Market
- Just a new word in place
- **However truth is:**
- As DevOps is not specific to a technology, tool, or process however, It is movement about improving organizational effectiveness and individual employee happiness
- One of the primary differences between DevOps and methodologies like ITIL and Agile is:
- It does not have strict definition
- DevOps is movement defined by stories and ideas of individuals, teams, and organizations
- DevOps is the evolution of processes and ideas that lead growth and change
- Harnessing and leveraging effective DevOps will lead to growth and evolution in tools, technology, and processes in your organization

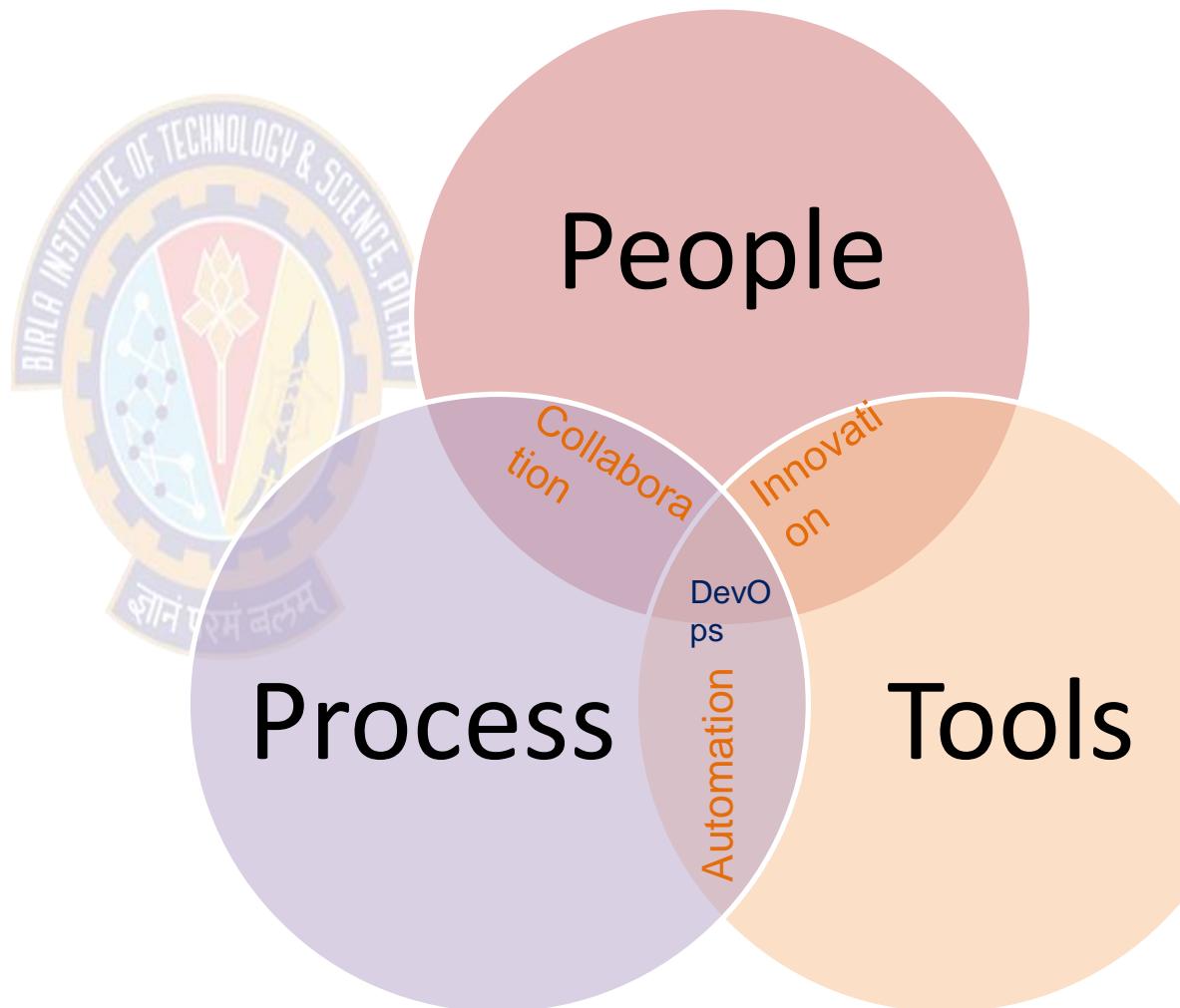
DevOps Anti-Patterns

- Blame Culture
 - A blame culture is one that tends toward blaming and punishing people when mistakes are made, either at an individual or an organizational level
- Silos
 - A departmental or organizational silo describes the mentality of teams that do not share their knowledge with other teams in the same company
- Root Cause Analysis
 - Root cause analysis (RCA) is a method to identify contributing and “root” causes of events or near-misses/close calls and the appropriate actions to prevent recurrence
- Human Errors
 - Human error, the idea that a human being made a mistake that directly caused a failure, is often cited as the root cause in a root cause analysis

Three dimensions of DevOps

3-D of DevOps

- People
- Process
- Tools / Technology



Agenda

Agile Methodology SCRUM

- Scrum



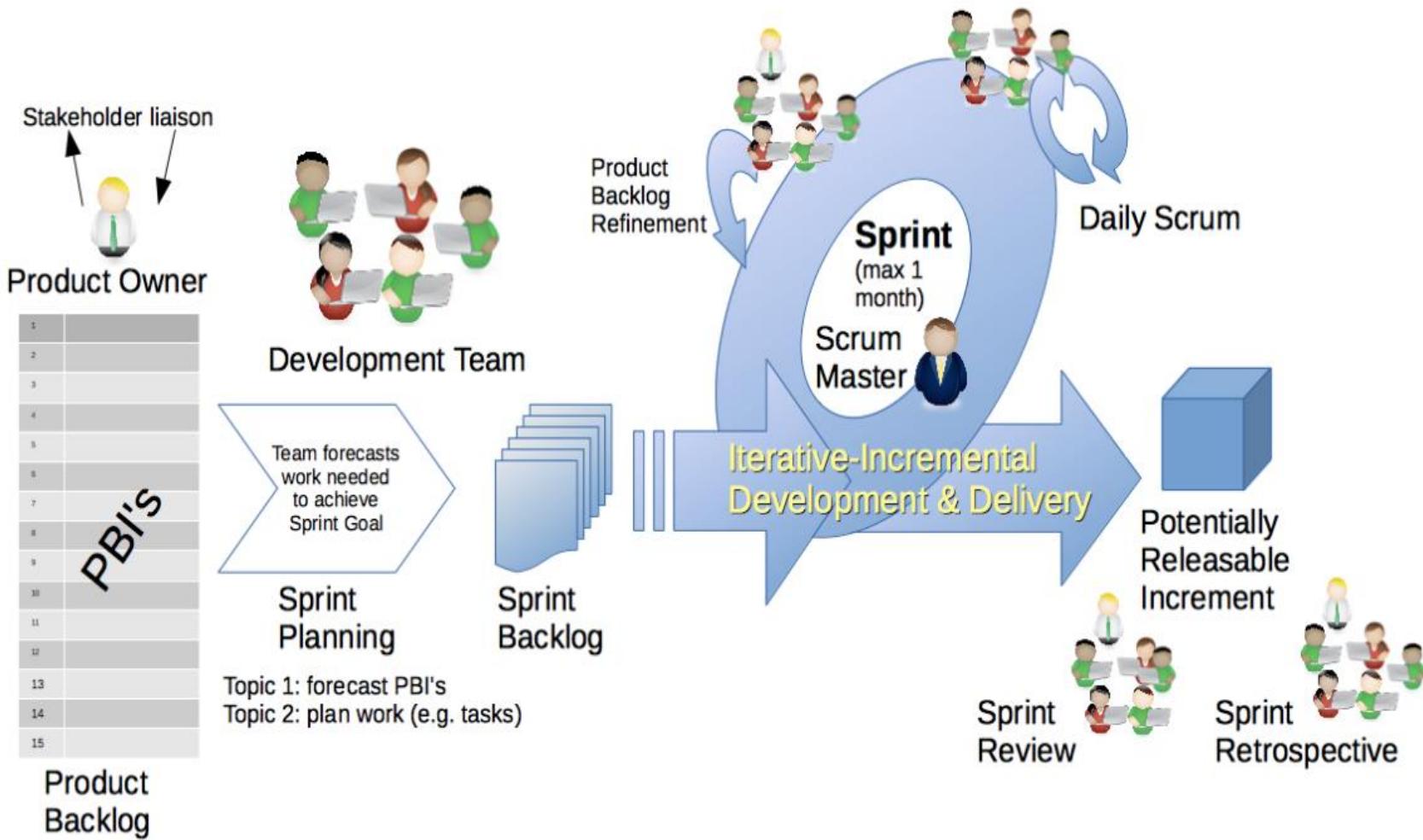
SCRUM Methodology

About Scrum

- Scrum is part of the Agile movement
- Scrum is an agile way to manage a project, usually software development
- Scrum's early advocates were inspired by empirical inspect and adapt feedback loops to cope with complexity and risk
- Scrum emphasizes decision making from real-world results rather than speculation
- By Mountain Goat Software – Pioneer of Agile
- “Agile software development with Scrum is often perceived as a methodology; but rather than viewing Scrum as methodology, think of it as a framework for managing a process”

SCRUM

Scrum Sprints



SCRUM

Benefits of Scrum

- Flexibility
- Quality
- Incremental Working Product
- Early to Market



Agenda

Agile Methodologies

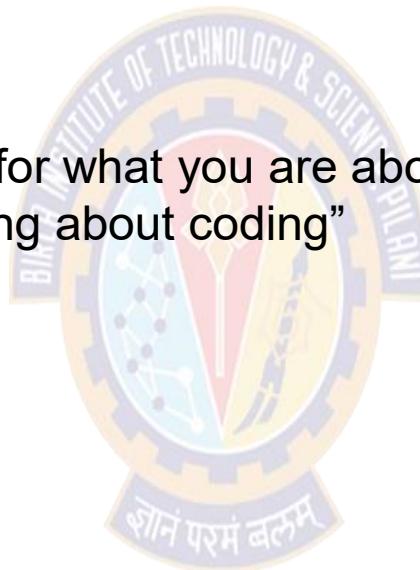
- Test Driven Development
- Feature Driven Development
- Behavior Driven Development



TDD

TDD Introduction

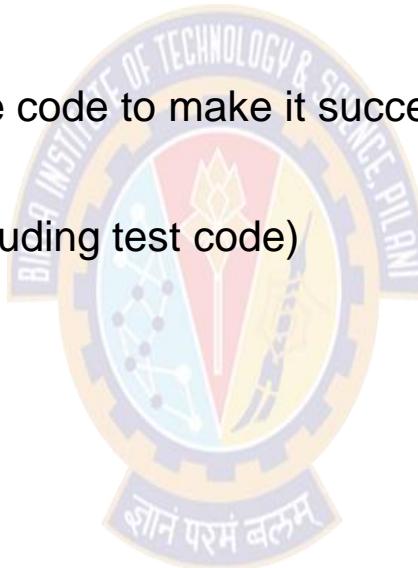
- Kent Beck said “Test-first code tends to be more cohesive and less coupled than code in which testing isn’t a part of the intimate coding cycle”
- “If you can’t write a test for what you are about to code, then you shouldn’t even be thinking about coding”



TDD

TDD Three steps

- **RED**
 - Write a new TEST which fails
- **GREEN**
 - Write simplest possible code to make it succeed
- **REFACTOR**
 - Refactor the code (including test code)



TDD

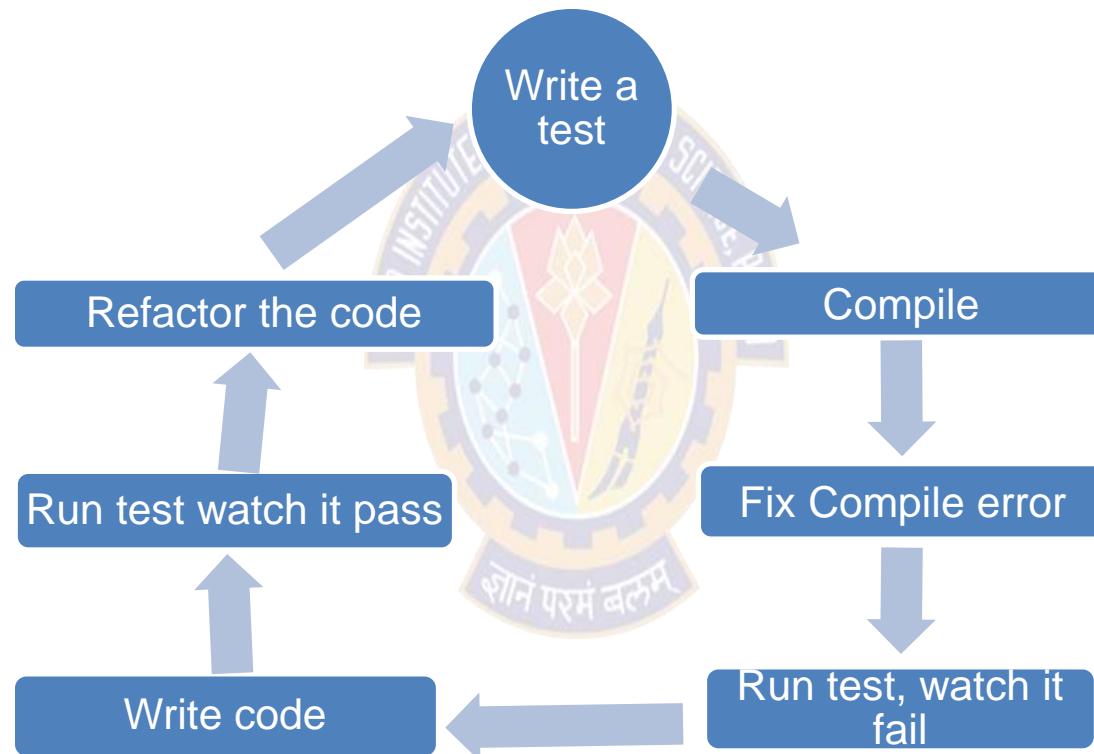
TDD Technique

- TDD is a technique whereby you write your test cases before you write any implementation code



TDD

TDD Overview



TDD

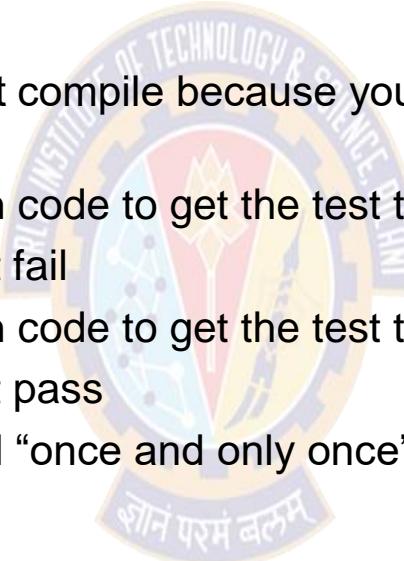
TDD Overview Contd..

- How It Works:
- Add a Test
 - Use Cases / User Stories are used to understand the requirement clearly
- Run all tests and see the new one fail
 - Ensures test harness is working correctly
 - Ensures that test does not mistakenly pass
- Write some code
 - Only code that is designed to pass the test
 - No additional functionality should be included because it will be untested
- Run the automated tests and see them succeed
 - If tests pass, programmer can be confident code meets all tested requirements
- Refactor code
 - Cleanup the code
 - Rerun tests to ensure cleanup did not break anything
- Repeat

TDD

TDD Overview Contd..

- In Extreme Programming Explored (The Green Book), Bill Wake describes the test / code cycle:
 1. Write a single test
 2. Compile it. It shouldn't compile because you've not written the implementation code
 3. Implement just enough code to get the test to compile
 4. Run the test and see it fail
 5. Implement just enough code to get the test to pass
 6. Run the test and see it pass
 7. Refactor for clarity and “once and only once”
 8. Repeat



TDD

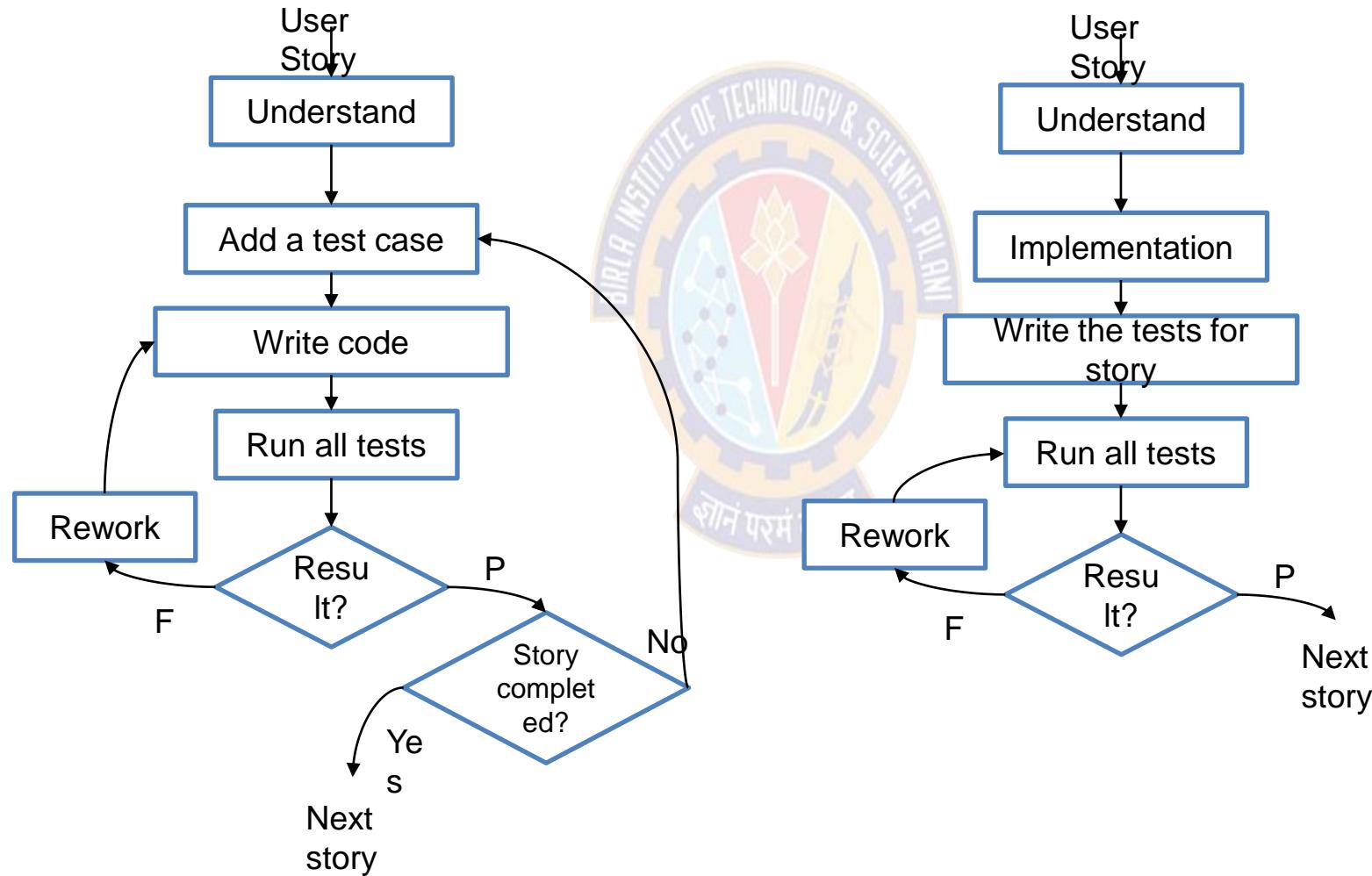
Why TDD

- TDD can lead to more modularized, flexible, and extensible code
- Clean code
- Leads to better design
- Better code documentation
- More productive



TDD

Test First vs. Test Last



Feature Driven Development

FDD History

- Original Creator: Jeff De Luca
 - Singapore in late 1997
- FDD evolved from an actual project
 - Bank Loan Automation
 - Luca was Project manager
 - 50 member developer team
 - Peter Coad : Chief Architect
 - 1990's object-oriented analysis and design expert



FDD

What is a Feature?

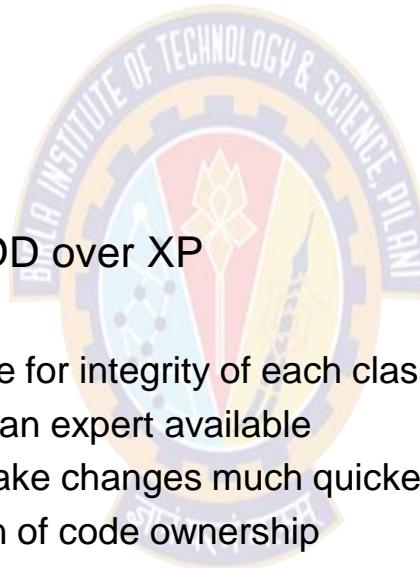
- Definition: small function expressed in client-valued terms
- FDD's form of a customer requirement



FDD

FDD Primary Roles

- Project Manager
- Chief Architect
- Development Manager
- Domain Experts
- Class Owners
 - This concept differs FDD over XP
 - Benefits
 - Someone responsible for integrity of each class
 - Each class will have an expert available
 - Class owners can make changes much quicker, if needed anytime
 - Easily lends to notion of code ownership
 - Assists in FDD scaling to larger teams, as we have one person available for complete ownership of feature.
- Chief Programmers



FDD

FDD Supporting Roles

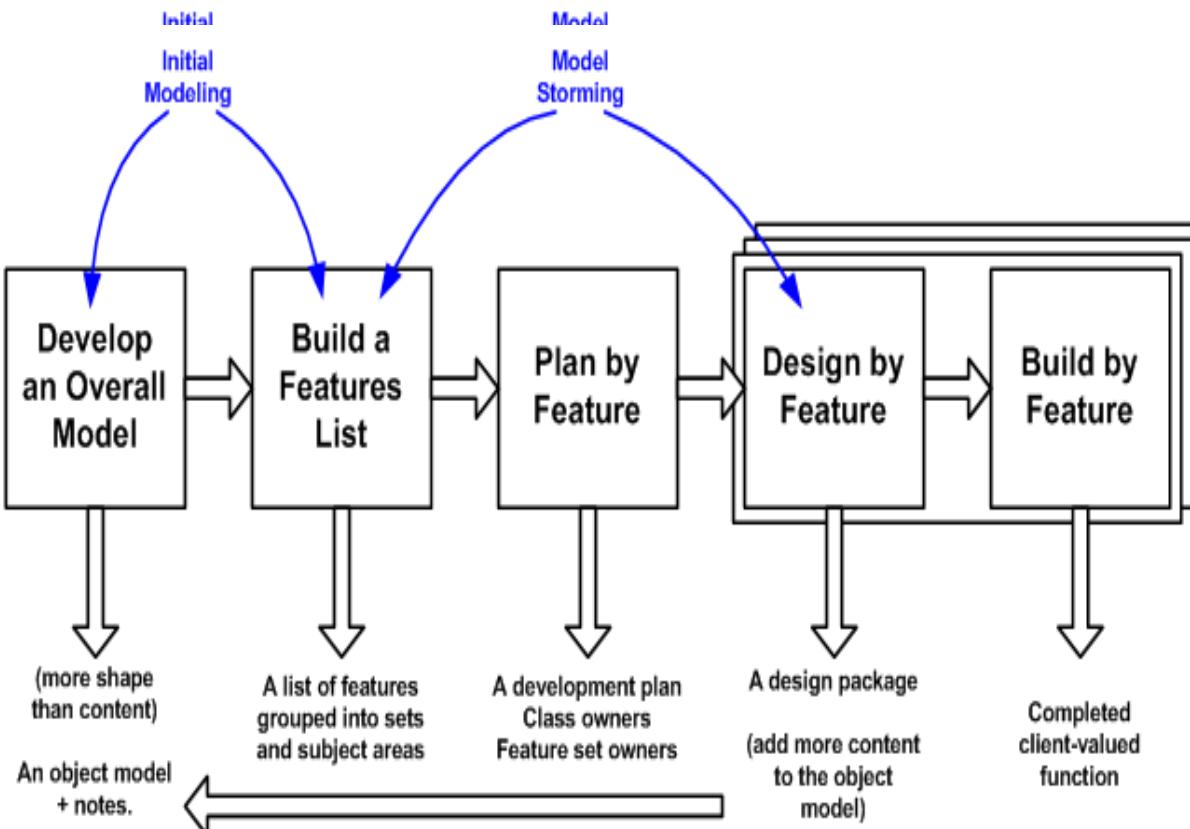
- Release Manager
- Language Guru
- Build Engineer
- Toolsmith
- System Administrator
- Tester
- Deployers
- Technical Writer



Feature Driven Development Process

- Process #1: Develop an Overall Model
- Process #2: Build a Features List
- Process #3: Plan By Feature
 - Constructing the initial schedule, Forming level of individual features, Prioritizing by business value , As we work on above factors we do consider dependencies, difficulty, and risks.
 - These factors will help us on Assigning responsibilities to team members, Determining Class Owners , Assigning feature sets to chief programmers
- Process #4: Design By Feature
 - Goal: not to design the system in its entirety but instead is to do just enough initial design that you are able to build on
 - This is more about Form Feature Teams: Where team members collaborate on the full low level analysis and design.
- Process #5: Build By Feature
 - Goal: Deliver real, completed, client-valued function as often as possible

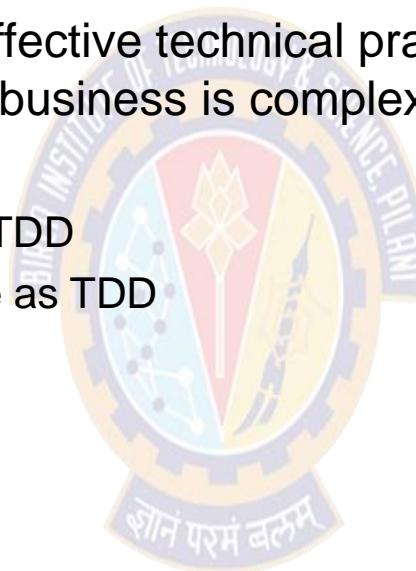
Feature Driven Development Process Contd..



Behavior Driven Development

BDD Introduction

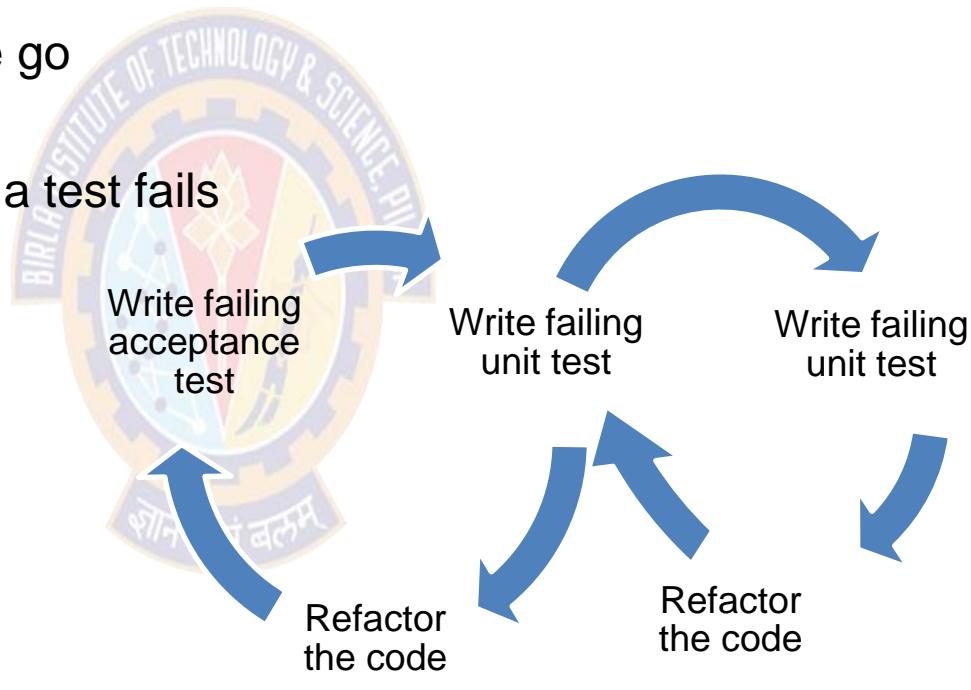
- Behavior-Driven Development (BDD) is a software development process
- BDD is considered an effective technical practice especially when the "problem space" for the business is complex
- What is BDD?:
 - General Technique of TDD
 - Follows same principle as TDD
 - Shared tools
 - Shared Process



BDD

BDD Focus

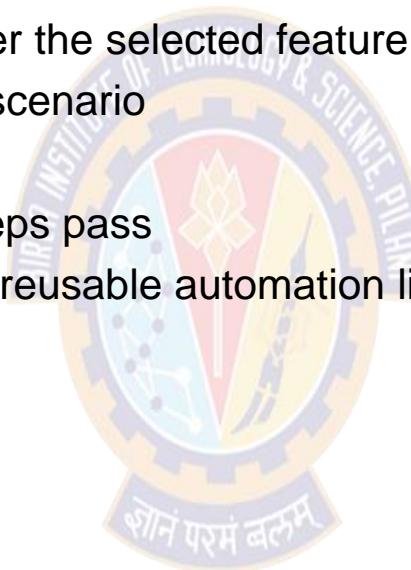
- Where to start in the process
- What to test and what not to test
- How much to test in one go
- What to call the tests
- How to understand why a test fails



BDD

BDD Basic structure

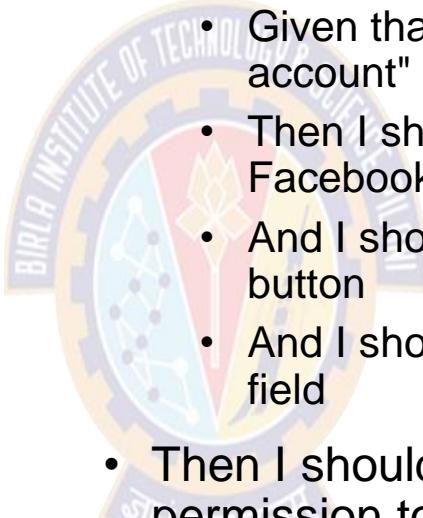
- BDD consists of cycles of a set of steps to follow.
 - Identify business feature
 - Identify scenarios under the selected feature
 - Define steps for each scenario
 - Run feature and fail
 - Write code to make steps pass
 - Refactor code, Create reusable automation library
 - Run feature and pass
 - Generate test reports



BDD

BDD Example

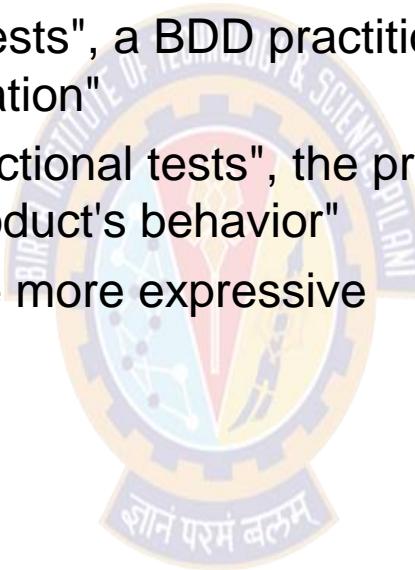
- User story:
- As someone interested in using the Mobile app, and want to sign up in app to enjoy app membership
- Mobile App Signup:
- Scenario 1:
 - Given that I am on the app's "Create new account" screen
 - Then I should see a "Sign up using Facebook" button
 - And I should see a "Sign up using Twitter" button
 - And I should see a "Sign up with email" form field
 - Then I should see a new screen that asks permission to use my Facebook account data to create my new Mobile app account



BDD

BDD Sign Use

- Functional Documentation in terms of User Stories & Executable Scenarios
- Instead of referring to "tests", a BDD practitioner will prefer the terms "scenario" and "specification"
- Rather than refer to "functional tests", the preferred term will be "specifications of the product's behavior"
- Specification Names are more expressive



References

- <http://agilemodeling.com/>



Agenda

DevOps – People

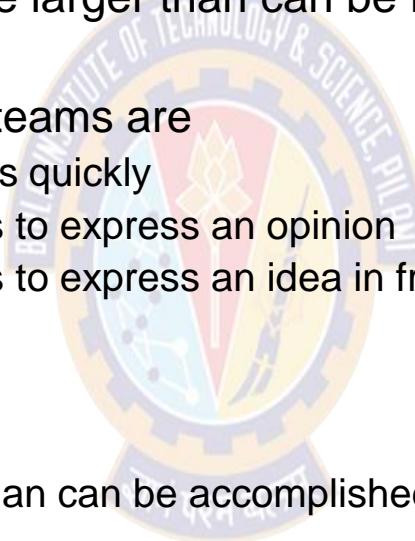
- DevOps Team Structure
- Team Coordination



Team Structure

Team Size

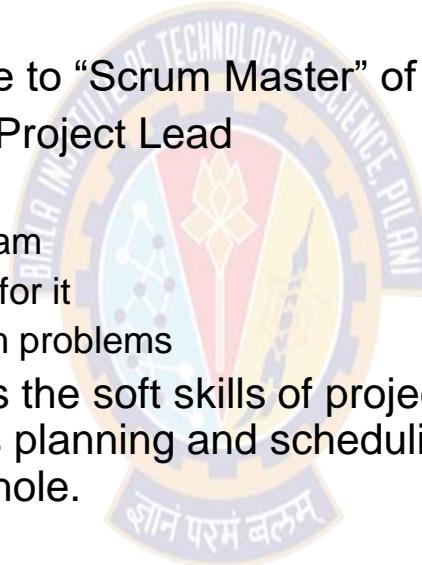
- The size of the team should be relatively small
- Amazon has a “two pizza rule”
- That is, no team should be larger than can be fed from two pizzas
- The advantages of small teams are
 - They can make decisions quickly
 - It is easier for individuals to express an opinion
 - It is easier for individuals to express an idea in front of a small group than in front of a large one
- The disadvantage
 - Some tasks are larger than can be accomplished by a small number of individuals
 - Task has to be broken up into smaller pieces
 - A small team, by necessity, works on a small amount of code



Team Structure

Team Roles

- We will study roles in the team from Scott Ambler's description of roles in an agile team
- Team lead :
 - This role you can relate to "Scrum Master" of SCRUM TEAM.
 - Also you can relate to Project Lead
 - Is responsible:
 - For facilitating the team
 - Obtaining resources for it
 - And protecting it from problems
 - This role encompasses the soft skills of project management but not the technical ones such as planning and scheduling, activities which are better left to the team as a whole.
- Team member
 - This role, sometimes referred to as developer or programmer
 - Is responsible for the creation and delivery of a system
 - This includes modeling, programming, testing, and release activities, as well as others



Team Structure

Additional roles

- Service owner
- Reliability engineer
- Gatekeeper
- And DevOps engineer



Team Structure

Service Owner

- This role on the team responsible for outside coordination
- The service owner participates:
 - System-wide requirements activities
 - Prioritizes work items for the team
 - And provides information both from the clients of the team's service and about services provided to the team.
 - The requirements gathering and release planning activities for the next iteration can occur in parallel with the conception phase of the current iteration
 - The service owner maintains and communicates the vision for the service
 - That is, the vision involves the architecture of the overall system and the team's role in that architecture

Team Structure

Reliability Engineer

- Reliability engineer monitors the service in the time period immediately subsequent to the deployment
- Second, the reliability engineer is the point of contact for problems with the service during its execution
- Google calls this role “Site Reliability Engineer.”
- Once a problem occurs, the reliability engineer performs short-term analysis to diagnose, mitigate, and repair the problem
- In any case, the reliability engineer has to be excellent at troubleshooting and diagnosis
- The reliability engineer should discover or work with the team to discover the root cause of a problem
- Increasingly, reliability engineers need to be competent developers, as they need to write high-quality programs to automate the repetitive part of the diagnosis, mitigation, and repair

Team Structure

Gatekeeper

Netflix uses the steps given in Figure 1.3 from local development to deployment.

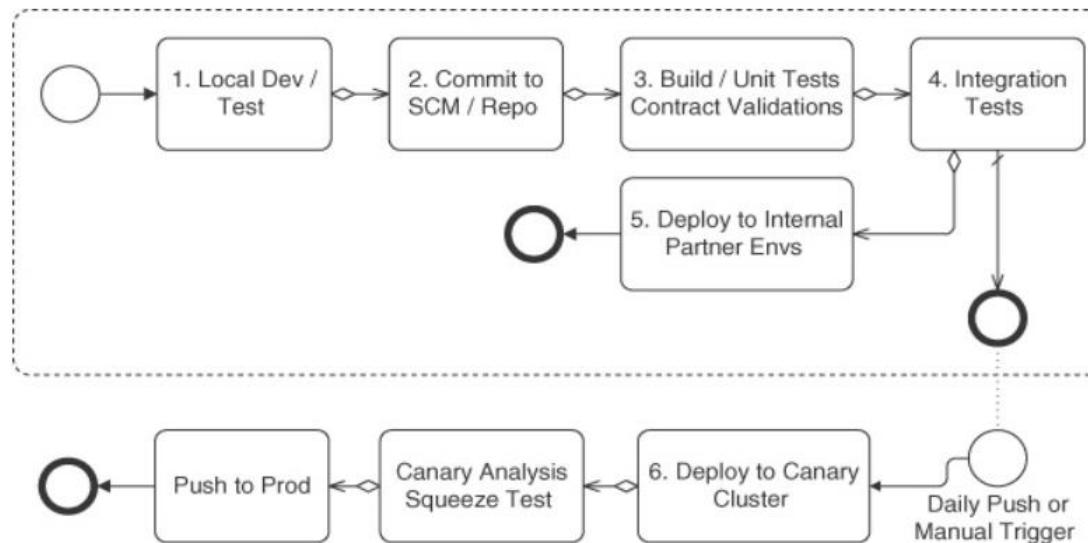


FIGURE 1.3 Netflix path to production. (Adapted from <http://techblog.netflix.com/2013/11/preparing-netflix-api-for-deployment.html>) [Notation: BPMN]

Team Structure

DevOps Engineer

- The DevOps engineer role is responsible for the care and feeding of the various tools used in the DevOps tool chain
- The DevOps engineer's role, as are the tailoring of the tool for the development team and monitoring
- The DevOps engineering role is inherent in automating the development and deployment pipeline
- DevOps engineer also deals with configuration managements tools which are like Puppet, Ansible and Chef etc.
- These configuration management solutions are applied to the source code for the service

Team Coordination

Forms of Coordination

- One goal of DevOps is to minimize coordination in order to reduce the time to market
- Two of the reasons to coordinate are, first, so that the pieces developed by the various teams will work together and, second, to avoid duplication of effort
- Forms of Coordination
 - Direct—the individuals coordinating know each other (e.g., team members)
 - Indirect—the coordination mechanism is aimed at an audience known only by its characterization (e.g. Operators)
 - Persistent—the coordination artifacts are available after the moment of the coordination (e.g., documents, e-mail, bulletin boards)
 - Ephemeral—the coordination, per se, produces no artifacts (e.g., face to face meetings, conversations, telephone/video conferencing)
 - Synchronous—individuals are coordinating in real time, (e.g., face to face)
 - Asynchronous—individuals are not coordinating in real time (e.g., documents, e-mail)

Team Coordination

Team coordination mechanisms

- Team coordination mechanisms are of two types
 - Human Processes and
 - Automated Processes
- The DevOps human processes are adopted from agile processes and are designed for high-bandwidth coordination with limited persistence
 - Examples: Stand-up meetings
- Automated team coordination mechanisms are designed to protect team members from interference of their and others' activities:
 - To automate repetitive tasks (continuous integration and deployment),
 - And to speed up error detection and reporting (automated unit, integration, acceptance, and live production tests)

Team Coordination

Cross-team Coordination

- Coordination must occur with customers, stakeholders, other development teams, and operations. Therefore, DevOps processes attempt to minimize this coordination as much as possible
- There are three types of cross-team coordination:
 - Upstream Coordination [with stakeholders and customers]
 - Downstream Coordination [with operations]
 - And Cross-Stream Coordination [with other development teams]
- There are two reasons for a development team to coordinate with other development teams
 - To ensure that the code developed by one team works well with the code developed by another
 - And to avoid duplication of effort

Agenda

DevOps People

- Transformation to Enterprise DevOps culture



Transformation to Enterprise DevOps culture

Constraints of Large Enterprise :: DevOps Transformation

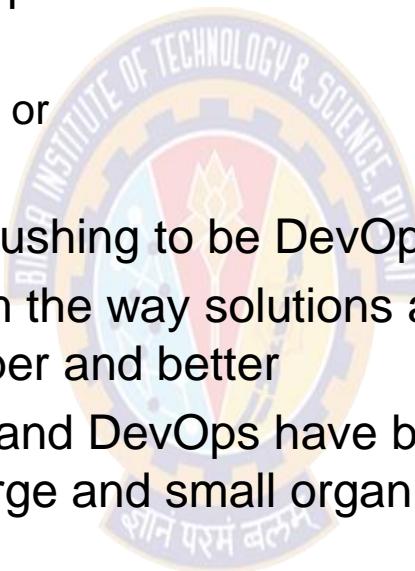
- Organization Structure
- Legacy Technology Stack
- Organization Culture



Transformation to Enterprise DevOps culture

Concepts

- Executives do not like terms such as “Transformation”
- Somehow it is often interpreted as:
 - Cost Cutting or
 - Resource Realignment or
 - Outsourcing
- Why organizations are pushing to be DevOps
- The intent is to transform the way solutions are being delivered today and to do it faster, cheaper and better
- The techniques of Agile and DevOps have been successfully implemented in many large and small organizations to enable those outcomes
- Look in to the approach that worked quite well for most enterprises



Transformation to Enterprise DevOps culture

Initial Planning for Enterprise Readiness

- Participants should absolutely include all the towers that make up the solution delivery
- Design Thinking : It is a great method; it leverages the expertise of all stakeholders, enables them to come to a common understanding
- High Level Output



Participant
& Inputs

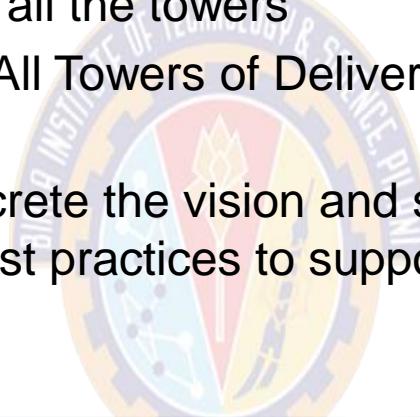
Design
Thinking

High Level
Outputs

Transformation to Enterprise DevOps culture

Establish a DevOps Center of Excellence

- At the right organizational level and enterprise authority
- Create a Face: It has to be led by an enterprise leader who has the support and buy-in from all the towers
- Active Participant from All Towers of Delivery & Participants must be chosen wisely
- These phases help concrete the vision and strategies of organization and help in setup the best practices to support cultural movement



Authority

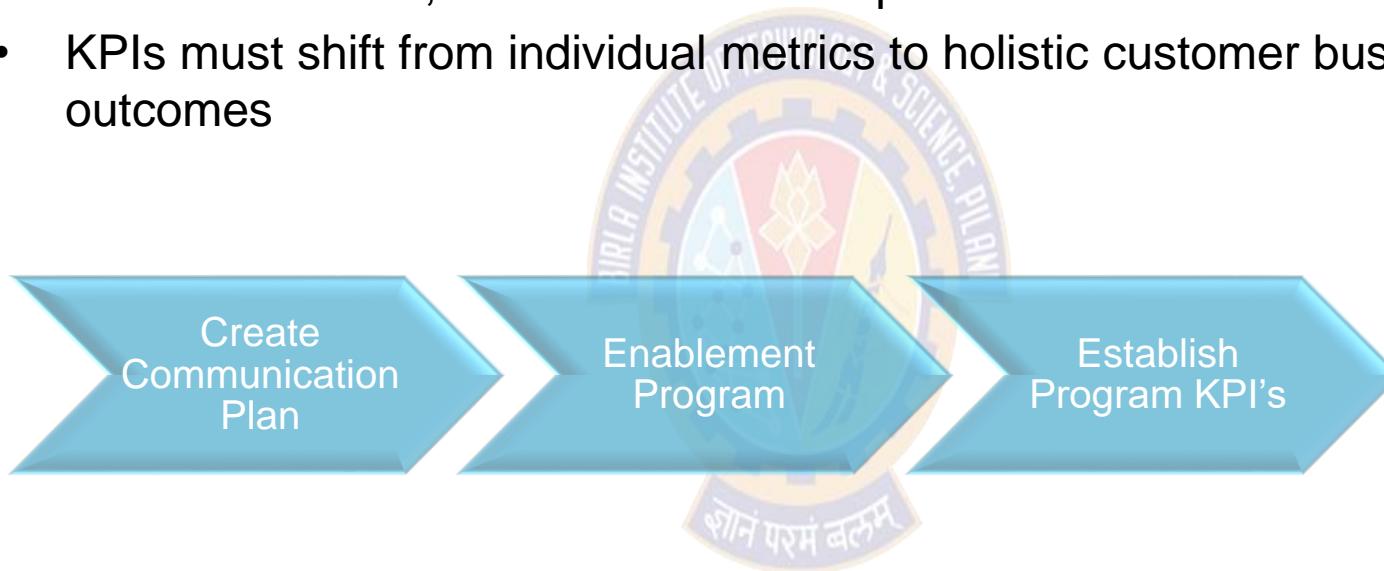
Create a Face

Involve All

Transformation to Enterprise DevOps culture

Establish Program Governance

- Agile and DevOps, practitioners' roles and responsibilities will change
- Need awareness, enablement and empowerment to succeed
- KPIs must shift from individual metrics to holistic customer business outcomes



Transformation to Enterprise DevOps culture

Establish Project In-take Process

- DevOps SME's conduct in-take workshops for Scrum Teams
- Automation scripts, Infrastructure assets
- Test Automation, Branching and Merging & Lessons Learned
- Fit for purpose tool selection and Application Criticality



Reusable Assets

Best Practices

Right Sizing

Transformation to Enterprise DevOps culture

Identify and Initiate Pilots

- Value stream-mapping exercise
- Level of detail necessary:
 - To identify end to end as-is process, tooling, manual and automated processes
 - And skills and people



Transformation to Enterprise DevOps culture

Scale Out DevOps Program

- Onboard Parallel Release Trains
- Apply Intake Process
- Support, Monitor and Manage



References

- I would like to thank you Mr. Sunil Joshi
- <https://devops.com/six-step-approach-enterprise-devops-transformation/>





Thank You!

In our next session:

Pros of Waterfall

Cons of Waterfall



Agile

- *individuals and interactions* over *processes and tools*
- *working software* over *comprehensive documentation*
- *customer collaboration* over *contract negotiation*
- *responding to change* over *following a plan*



Scrum

- **Scrum**
- In the mid-1990s, Ken Schwaber and Dr. Jeff Sutherland, two of the original creators of the Agile Manifesto, merged individual efforts to present a new software development process called Scrum. Scrum is a software development methodology that focuses on maximizing a development team's ability to quickly respond to changes in both project and customer requirements. It uses predefined development cycles called *sprints*, usually between one week and one month long, beginning with a sprint planning meeting to define goals and ending with a sprint review and sprint retrospective to discuss progress and any issues that arose during that sprint.

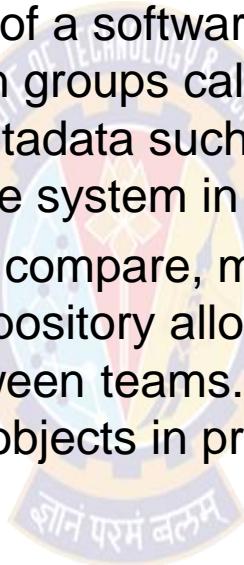
DSM

- What did I do yesterday that helped the team meet its sprint goals?
- What am I planning to do today to help the team meet those goals?
- What, if anything, do I see that is blocking either me or the team from reaching their goals?



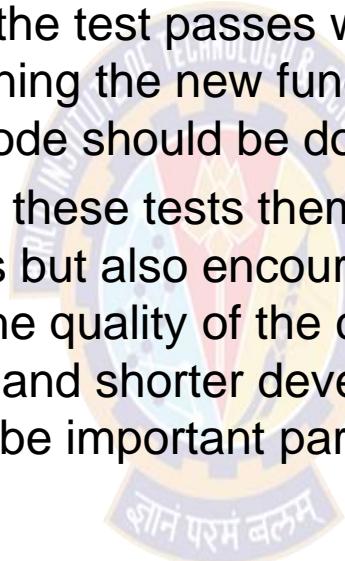
Version Control

- A version control system records changes to files or sets of files stored within the system. This can be source code, assets, and other documents that may be part of a software development project. Developers make changes in groups called *commits* or *revisions*. Each revision, along with metadata such as who made the change and when, is stored within the system in one way or another.
- Having the ability to commit, compare, merge, and restore past revisions to objects to the repository allows for richer cooperation and collaboration within and between teams. It minimizes risks by establishing a way to revert objects in production to previous versions.



Test Driven Development

- In test-driven development, the code developer starts by writing a failing test for the new code functionality, then writes the code itself, and finally ensures that the test passes when the code is complete. The test is a way of defining the new functionality clearly, making more explicit what the code should be doing.
- Having developers write these tests themselves not only greatly shortens feedback loops but also encourages developers to take more responsibility for the quality of the code they are writing. This sharing of responsibility and shorter development cycle time are themes that continue to be important parts of a devops culture.

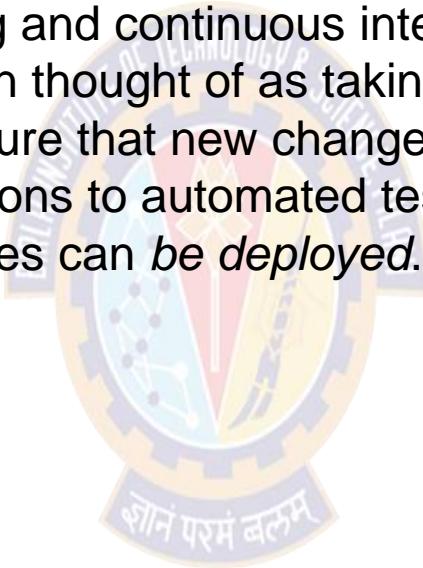


Continuous Integration

- Continuous integration (CI) is the process of integrating new code written by developers with a mainline or “master” branch frequently throughout the day. This is in contrast to having developers working on independent feature branches for weeks or months at a time, merging their code back to the master branch only when it is completely finished. Long periods of time in between merges means that much more has been changed, increasing the likelihood of some of those changes being breaking ones. With bigger changesets, it is much more difficult to isolate and identify what caused something to break. With small, frequently merged changesets, finding the specific change that caused a regression is much easier. The goal is to avoid the kinds of integration problems that come from large, infrequent merges.

Continuous Delivery

- Continuous delivery (CD) is a set of general software engineering principles that allow for frequent releases of new software through the use of automated testing and continuous integration. It is closely related to CI, and is often thought of as taking CI one step further, that beyond simply making sure that new changes can be integrated without causing regressions to automated tests, continuous delivery means that these changes can *be deployed*.

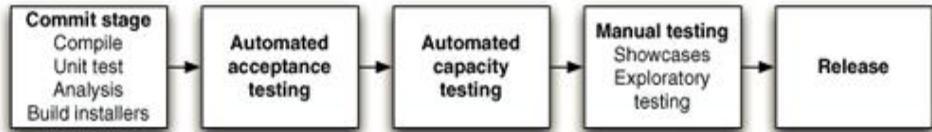


Continuous Deployment

- Continuous deployment (also referred to as CD) is the process of deploying changes to production by defining tests and validations to minimize risk. While continuous delivery makes sure that new changes can be deployed, continuous deployment means that they *get deployed* into production.



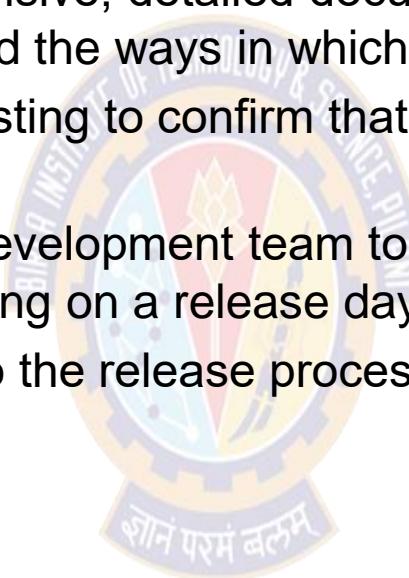
Deployment Pipeline



- The pattern that is central to this book is the *deployment pipeline*. A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process. Every organization will have differences in the implementation of their deployment pipelines, depending on their value stream for releasing software, but the principles that govern them do not vary. An example of a deployment pipeline is given in Figure

Antipattern: Deploying Software Manually

- The signs of this antipattern are:
 - The production of extensive, detailed documentation that describes the steps to be taken and the ways in which the steps may go wrong
 - Reliance on manual testing to confirm that the application is running correctly
 - Frequent calls to the development team to explain why a deployment is going wrong on a release day
 - Frequent corrections to the release process during the course of a release



Antipattern: Deploying to a Production-like Environment Only after Development Is Complete

- If testers have been involved in the process up to this point, they have tested the system on development machines.
- Releasing into staging is the first time that operations people interact with the new release. In some organizations, separate operations teams are used to deploy the software into staging and production. In this case, the first time an operations person sees the software is the day it is released into production.
- Either a production-like environment is expensive enough that access to it is strictly controlled, or it is not in place on time, or nobody bothered to create one.
- The development team assembles the correct installers, configuration files, database migrations, and deployment documentation to pass to the people who perform the actual deployment—all of it untested in an environment that looks like production or staging.

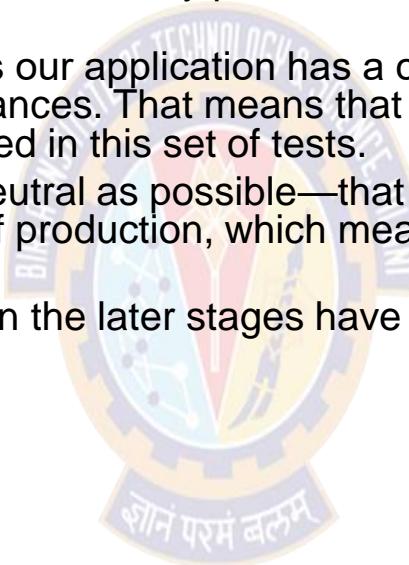
Every Change Should Trigger the Feedback Process

- Executable code changes when a change is made to the source code. Every time a change is made to the source code, the resulting binary must be built and tested. In order to gain control over this process, building and testing the binary should be automated. The practice of building and testing your application on every check-in is known as continuous integration



Possible

- They run fast.
- They are as comprehensive as possible—that is to say, they cover more than 75% or so of the codebase, so that when they pass, we have a good level of confidence that the application works.
- If any of them fails, it means our application has a critical fault and should not be released under any circumstances. That means that a test to check the color of a UI element should not be included in this set of tests.
- They are as environment-neutral as possible—that is, the environment does not have to be an exact replica of production, which means it can be simpler and cheaper.
- On the other hand, the tests in the later stages have the following general characteristics.





Thank You!

Acknowledgement:

- Significant portions of the information in the presentation is from IT Systems Management - Rich Schiesser and other books/Internet. Permission was requested from the publisher, sometime ago for use, but has not been received as yet. This is intended solely to teach BITS students enrolled for IM. I would like to acknowledge and reiterate that all the credit/rights remain with the original authors/publishers only. Mistakes if any are mine.



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps





CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 4

Agenda

Cloud as a catalyst for DevOps

- Introduction
- Characterization of the cloud
- Cloud services



Cloud as a Catalyst for DevOps

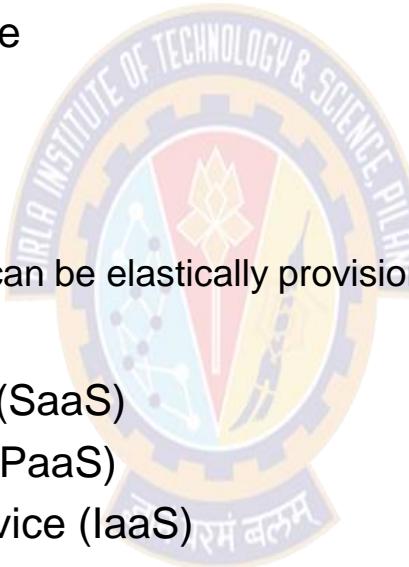
Introduction

- Unfortunately for many organizations, the cloud is still somewhat mysterious
- Not the concept of the cloud but why they are not seeing the benefits that were promised to them
- The focus is on adopting cloud-based infrastructure and managing it
- While we will discuss cost benefits primarily, one should not forget the other benefits and risks of the cloud
- Benefits:
 - Redundancy for resilience
 - The scalability of resources
 - The strong ecosystem of related
- Risks:
 - The dependency on a third-party provider
 - The challenges with data sovereignty
 - Risk of being attacked because you are on a popular platform

Cloud as a Catalyst for DevOps

Characterization of the cloud

- Characterization of the cloud by National Institute of Standards and Technology (NIST)
 - On-demand self-service
 - Broad network access
 - Resource pooling
 - Rapid elasticity
 - It is the Capabilities can be elastically provisioned and released
- Measured service
 - Software as a Service (SaaS)
 - Platform as a Service (PaaS)
 - Infrastructure as a Service (IaaS)



Cloud as a Catalyst for DevOps

Software as a Service (SaaS)

- In this the consumer is provided the capability to use the provider's applications running on a cloud infrastructure
- The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based e-mail) or an application interface.
- The consumer does not manage or control the underlying cloud infrastructure including networks, servers, operating systems, storage.
- For an example, you can relate google apps, Cisco WebEx, as a Service, Office 365 service, where Provider deals with the licensing of software's

Cloud as a Catalyst for DevOps

Platform as a Service (PaaS)

- The consumer is provided the capability to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider
- The consumer does not manage or control the underlying cloud infrastructure including networks, servers, operating systems, or storage, but consumer has control over the deployed applications and possibly configuration settings for the application-hosting environment
- For an Example: .NET Development platform is considered as a platform

Cloud as a Catalyst for DevOps

Infrastructure as a Service (IaaS),

- The consumer is provided the capability to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications
- The consumer does not manage or control the underlying cloud infrastructure but consumer has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls). For this you can consider any Server Provisioning is IaaS,

Cloud as a Catalyst for DevOps

Example

- Think of a shopping site that provides you with personalized recommendations based on your previous purchases
- If that service does not work for some reason, rather than not showing you the site or delaying the response for your request, ecommerce site might choose to show you some static recommendation
- For you as a user, the experience is slightly worse (although many people wouldn't notice), but it is still much better than getting a page error or time-out
- Leveraging this idea of graceful degradation works in many contexts but not all, and it will usually be a cheaper alternative to keeping the whole end-to-end availability at very high availability

Cloud as a Catalyst for DevOps

Summary

- The cloud has emerged as a major trend in IT during recent years
- Its characteristics include metered usage (pay-per-use) and rapid elasticity, allowing the scaling out of an application to virtually infinite numbers of VMs
- The cloud rests on a platform that is inherently distributed and exploits virtualization to allow rapid expansion and contraction of the resources available to a given user
- From an operational perspective, controlling the VMs, managing different database management systems, and ensuring the environments meet the needs of the development and operations tasks are new considerations associated with the cloud

Agenda

Evolution of Version Control

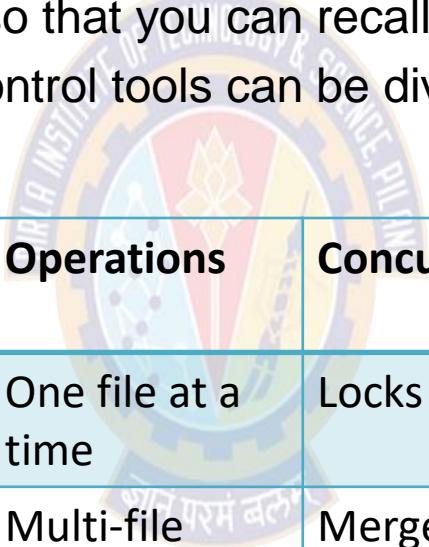
- What is Version Control
- Version Control System
- Benefits of Version Control System



Evolution of Version Control

What is Version Control

- What is “version control”, and why should you care?
- Definition: Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later
- The history of version control tools can be divided into three generations

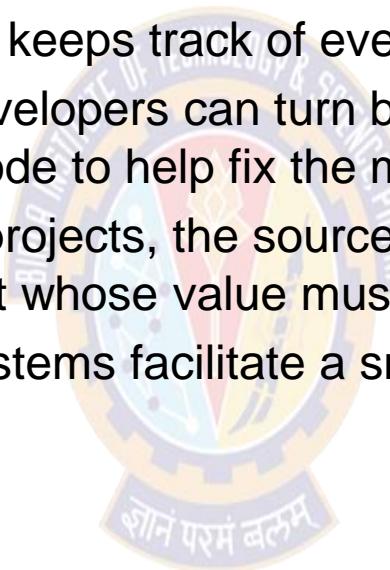


Generation	Networking	Operations	Concurrency	Example Tool
First Generation	None	One file at a time	Locks	RCS, SCCS
Second Generation	Centralized	Multi-file	Merge before commit	CVS, Subversion
Third Generation	Distributed	Changesets	Commit before merge	Bazaar, Git

Evolution of Version Control

Version Control System

- A category of software tools that help a software team manage changes to source code over time
- Version control software keeps track of every modification to the code
- If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake
- For almost all software projects, the source code is like the crown jewels - a precious asset whose value must be protected
- Great version control systems facilitate a smooth and continuous flow of changes to the code



Evolution of Version Control

Benefits of Version Control System (VCS)

- A complete long-term change history of every file
- Restoring previous versions
- Branching and merging; having team members work concurrently
- Traceability; being able to trace each change made to the software
- Backup



Evolution of Version Control

Summary

- The vast majority of professional programmers are using second generation tools
- However the third generation is growing very quickly in popularity
- The most popular Version Control System is Apache Subversion
- Commonly known as SVN:
 - Subversion is an open source version control system
 - Founded in 2000 by CollabNet, Inc.
 - Subversion is developed as a project of the Apache Software Foundation
 - It's a centralized version control system
 - Current Stable release is “Apache Subversion 1.11.1” as on date 11-Jan-2019

Agenda

Version control system and its types

- Centralized Version Control System
- Distributed Version Control System
- Version Control System Basics
- Version Control System Common Terminology
- Version Control System Basic Operations



Version Control System

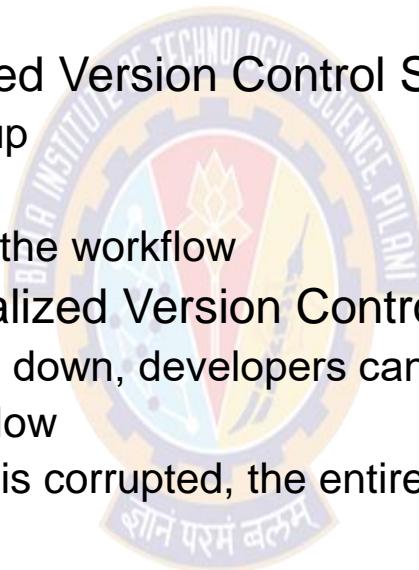
Version Control System Types

- Centralized Version Control System:[CVCS]
 - With centralized version control systems, you have a single “central” copy of your project on a server and commit your changes to this central copy
 - You pull the files that you need, but you never have a full copy of your project locally
 - Some of the most common version control systems are:
 - Subversion (SVN) by Apache
 - Perforce by Perforce Software
- Distributed Version Control System: [DVCS]
 - With distributed version control systems (DVCS), you don't rely on a central server to store all the versions of a project's files
 - Instead, you clone a copy of a repository locally so that you have the full history of the project
 - Some of most common distributed version control systems are:
 - Git
 - and Mercurial

Version Control System

Centralized Version Control System:[CVCS]

- Centralized Version Control Systems were developed to record changes in a central system and enable developers to collaborate on other systems
- Advantages of Centralized Version Control Systems:
 - Relatively easy to set up
 - Provides transparency
 - Enable admins control the workflow
- Disadvantages of Centralized Version Control Systems:
 - If the main server goes down, developers can't save versioned changes
 - Remote commits are slow
 - If the central database is corrupted, the entire history could be lost (security issues)



Version Control System

Distributed Version Control System:[DVCS]

- They allow developers to clone the repository and work on that version. Developers will have the entire history of the project on their own hard drives
- Advantages of Distributed Version Control Systems:
 - Performing actions other than pushing and pulling changesets is extremely fast because the tool only needs to access the hard drive, not a remote server
 - Everything but pushing and pulling can be done offline
 - Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback
- Disadvantages of Centralized Version Control Systems:
 - If your project contains many large files then the space will be more on local drives
 - If your project has a very long history then downloading the entire history can take an impractical amount of time

Version Control System

Version Control System Available:[Centralized]

- Open source:
 - Subversion (SVN) – versioning control system inspired by CVS
 - Concurrent Versions System (CVS) – originally built on RCS, licensed under the GPL
 - Vesta – build system with a versioning file system and support for distributed repositories
 - OpenCVS – CVS clone under the BSD license, with emphasis put on security and source code correctness
- Commercial:
 - AccuRev – source configuration management tool with integrated issue tracking based on "Streams" that efficiently manages parallel and global development. Now owned by Micro Focus
 - Helix Core, formerly Perforce Helix - for large scale development environments
 - IBM Rational ClearCase – SCC compliant configuration management system by IBM Rational Software
 - Team Foundation Server (TFS) - Development software by Microsoft which includes revision control

Version Control System

Version Control System Available:[Distributed]

- Open source:
 - Git – written in a collection of Perl, C, and various shell scripts, designed by Linus Torvalds based on the needs of the Linux kernel project; decentralized, and aims to be fast, flexible, and robust
 - Bazaar – written in Python, originally by Martin Pool and sponsored by Canonical; decentralized, and aims to be fast and easy to use; can losslessly import Arch archives
 - Mercurial – written in Python as an Open Source replacement to BitKeeper; decentralized and aims to be fast, lightweight, portable, and easy to use
- Commercial:
 - Visual Studio Team Services - Services for teams to share code, track work, and ship software for any language by Microsoft
 - Sun WorkShop TeamWare – designed by Larry McVoy, creator of BitKeeper
 - Plastic SCM – by Codice Software, Inc
 - Code Co-op – peer-to-peer version control system (can use e-mail for synchronization)

Version Control System Basics

What do you want from your version control system?

- Store all versions of your files ("version control")
- Associate versions of each file with appropriate versions of all other files ("configuration management")
- Allow many people to work on the same files, toward a common goal or release ("concurrency")
- Allow groups of people to work on substantially the same files, but each group towards its own goal or release ("branching")
- Recover, at any time, a coherent configuration of file versions that correspond to some goal or release, either for investigation or extension like bug fixing ("release management")

Version Control System Basics

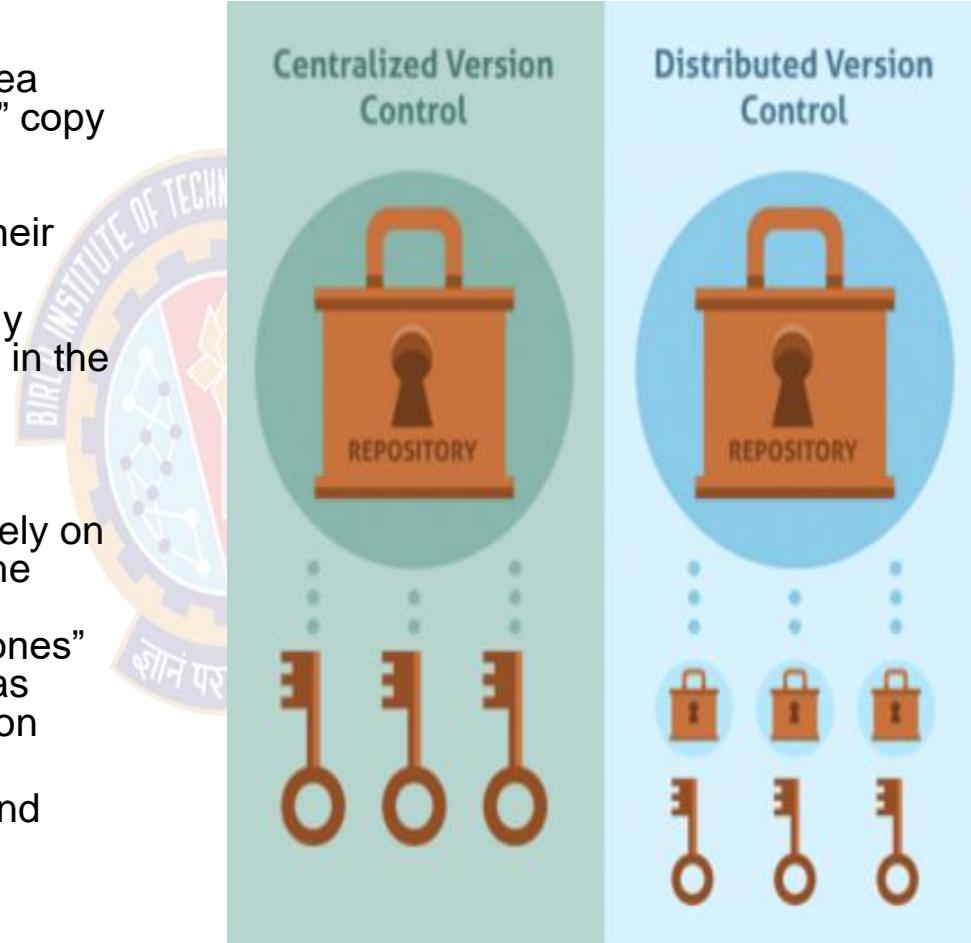
Version Control System Terminology

- Baseline : An approved revision of a document or source file from which subsequent changes can be made
- Branch : A set of files under version control may be branched or forked at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other
- Change : A change represents a specific modification to a document under version control
- Checkout : To check out is to create a local working copy from the repository. A user may specify a specific revision or obtain the latest
- Clone : Cloning means creating a repository containing the revisions from another repository
- Commit : To commit is to write or merge the changes made in the working copy back to the repository
- Conflict : A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes
- Fetch : Fetch is initiated by the receiving repository
- Initialize : To create a new, empty repository
- Merge : A merge or integration is an operation in which two sets of changes are applied to a file or set of files
- Repository : The repository is where files' current and historical data are stored, often on a server
- Tag : A tag or label refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number

Version Control System Basics

Summary to Understand:

- Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy
- “Committing” a change simply means recording the change in the central system
- Distributed Version control systems do not necessarily rely on a central server to store all the versions of a project’s files. Instead, every developer “clones” a copy of a repository and has the full history of the project on their own hard drive
- Then you collect all copies and prepare a final copy



Version Control System Basics

Centralized VCS over Distributed VCS

- Advantages of Centralized VCS over Distributed VCS :
 - If your project contains many large, binary files that cannot be easily compressed, the space needed to store all versions of these files can accumulate quickly
 - If your project has a very long history (50,000 changesets or more), downloading the entire history can take an impractical amount of time and disk space
- Advantages of Distributed VCS over Centralized VCS :
 - Performing actions other than pushing and pulling changesets is extremely fast because the tool only needs to access the hard drive, not a remote server
 - Committing new changesets can be done locally without anyone else seeing them. Once you have a group of changesets ready, you can push all of them at once
 - Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone
- Misconceptions: Distributed systems do not prevent you from having a single “central” repository, they just provide more options on top of that

Version Control System Basic Operations

Create

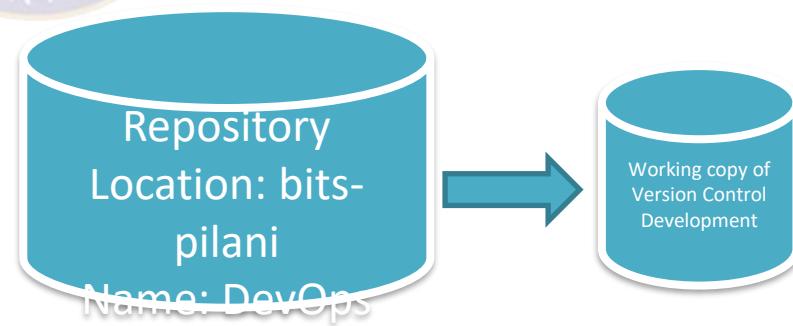
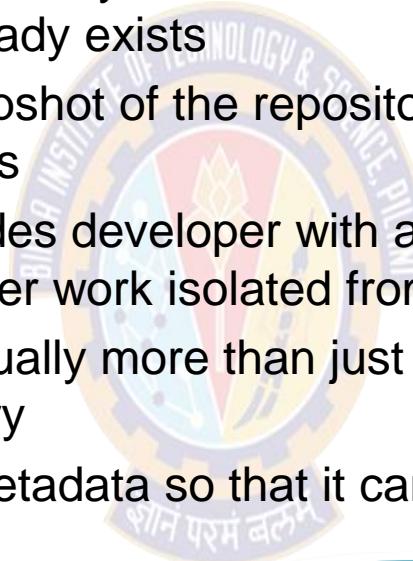
- Create a new, empty repository
- A repository is the official place where you store all your work.
- It keeps track of your tree i.e. all your files, as well as the layout of the directories in which they are stored
- A version control repository would be a ~~network~~ filesystem
- A repository is much more than that. A repository contains history
 - Repository = Filesystem * Time
- A filesystem is two-dimensional: Its space is defined by directories and files
- Whereas a repository is three-dimensional: It exists in a continuum defined by directories, files, and time
- What you need to create a Repository?
 - Location [Where you want to create it]
 - Name



Version Control System Basic Operations

Checkout

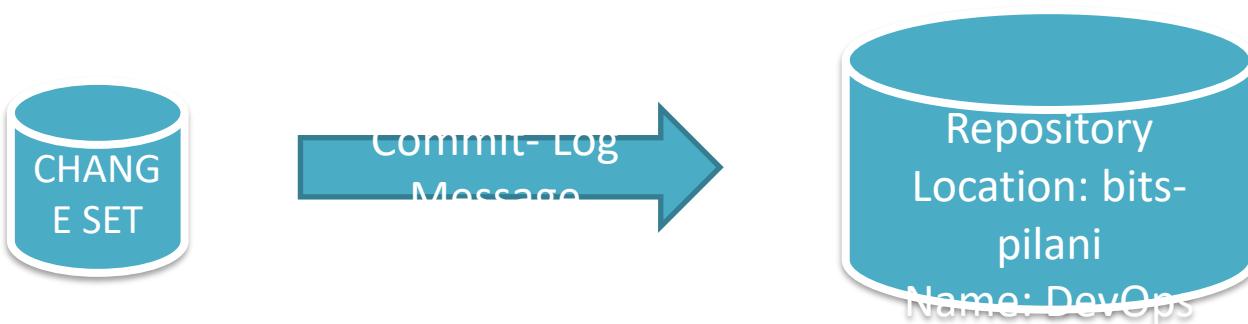
- Create a working copy
- This operation is used when you need to make a new working copy for a repository that already exists
- A working copy is a snapshot of the repository used by a developer as a place to make changes
- The working copy provides developer with a private workspace where developer can do his / her work isolated from the rest of the team
- The working copy is actually more than just a snapshot of the contents of the repository
- It also contains some metadata so that it can keep careful track of the state of things



Version Control System Basic Operations

Commit

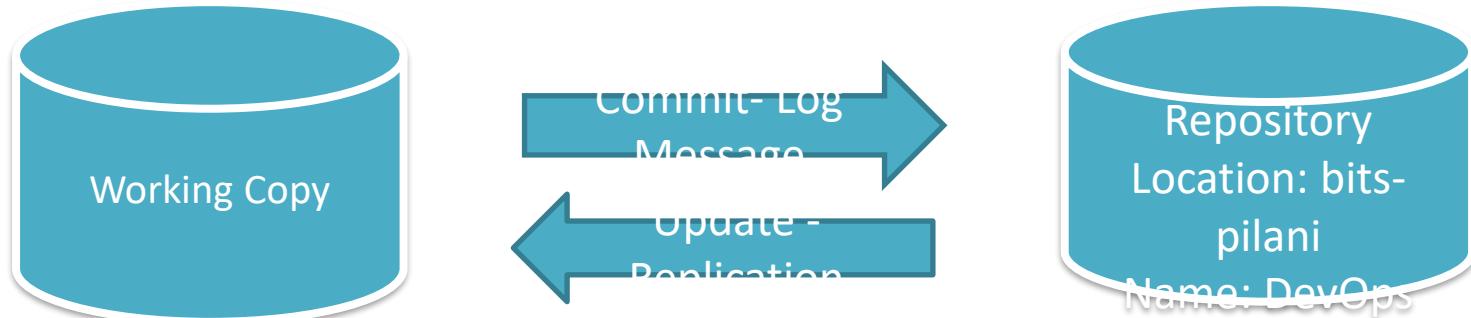
- Apply the modifications in the working copy to the repository as a new changeset
- This is the operation that actually modifies the repository
- Several others modify the working copy and add an operation to a list we call the pending changeset, a place where changes wait to be committed
- The commit operation takes the pending changeset and uses it to create a new version of the tree in the repository
- It is typical to provide a log message (or comment) when you commit, explaining the changes you have made. This log message becomes part of the history of the repository



Version Control System Basic Operations

Update

- Update the working copy with respect to the repository
- Update brings your working copy up-to-date by applying changes from the repository, merging them with any changes you have made to your working copy if necessary
- When the working copy was first created, its contents exactly reflected a specific revision of the repository
- Update is sort of like the mirror image of commit
- Both operations move changes between the working copy and the repository
- Commit goes from the working copy to the repository; Update goes in the other direction



Version Control System Basic Operations

Add

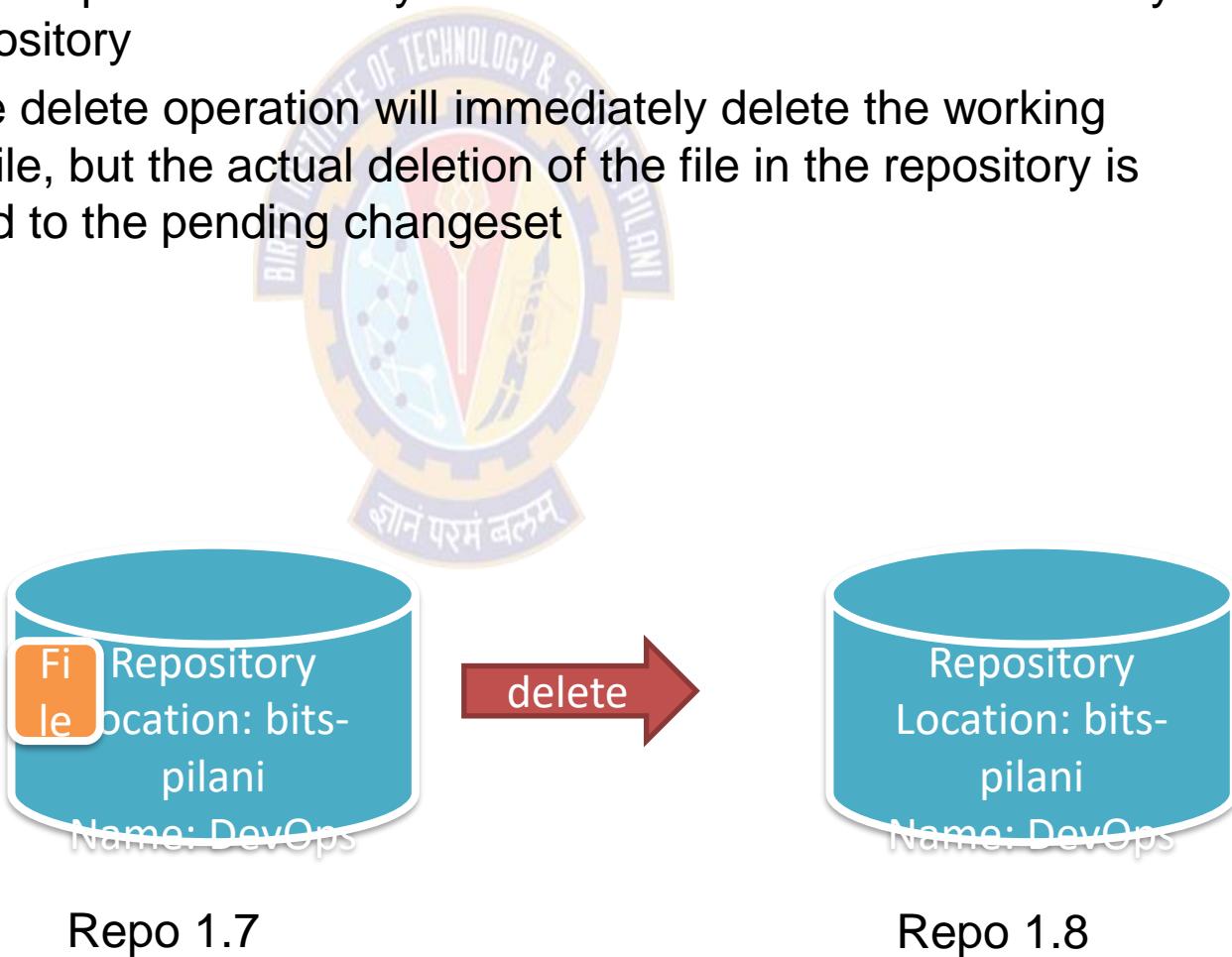
- Add a file or directory
- Use the add operation when you have a file or directory in your working copy that is not yet under version control and you want to add it to the repository



Version Control System Basic Operations

Delete

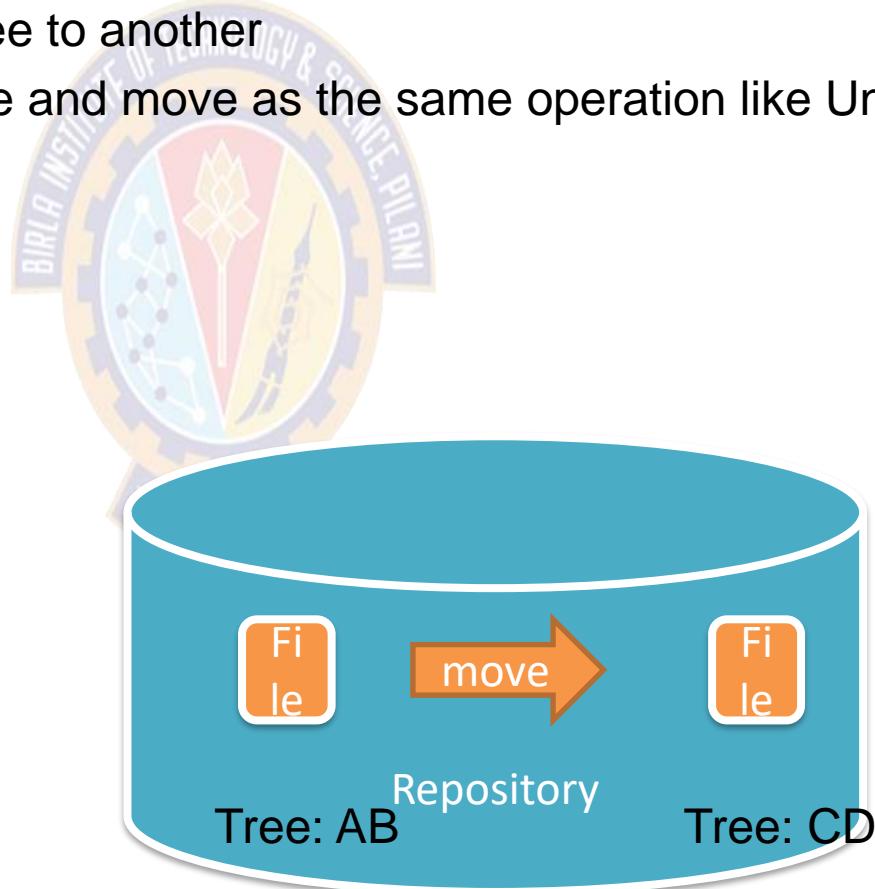
- Delete a file or directory
- Use the delete operation when you want to remove a file or directory from the repository
- Typically, the delete operation will immediately delete the working copy of the file, but the actual deletion of the file in the repository is simply added to the pending changeset



Version Control System Basic Operations

Move

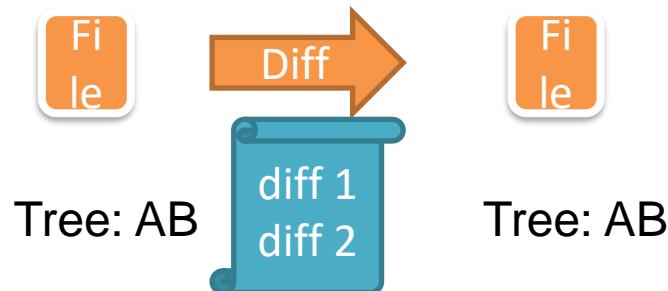
- Move a file or directory
- Use the move operation when you want to move a file or directory from one place in the tree to another
- Some tools treat rename and move as the same operation like Unix



Version Control System Basic Operations

Diff

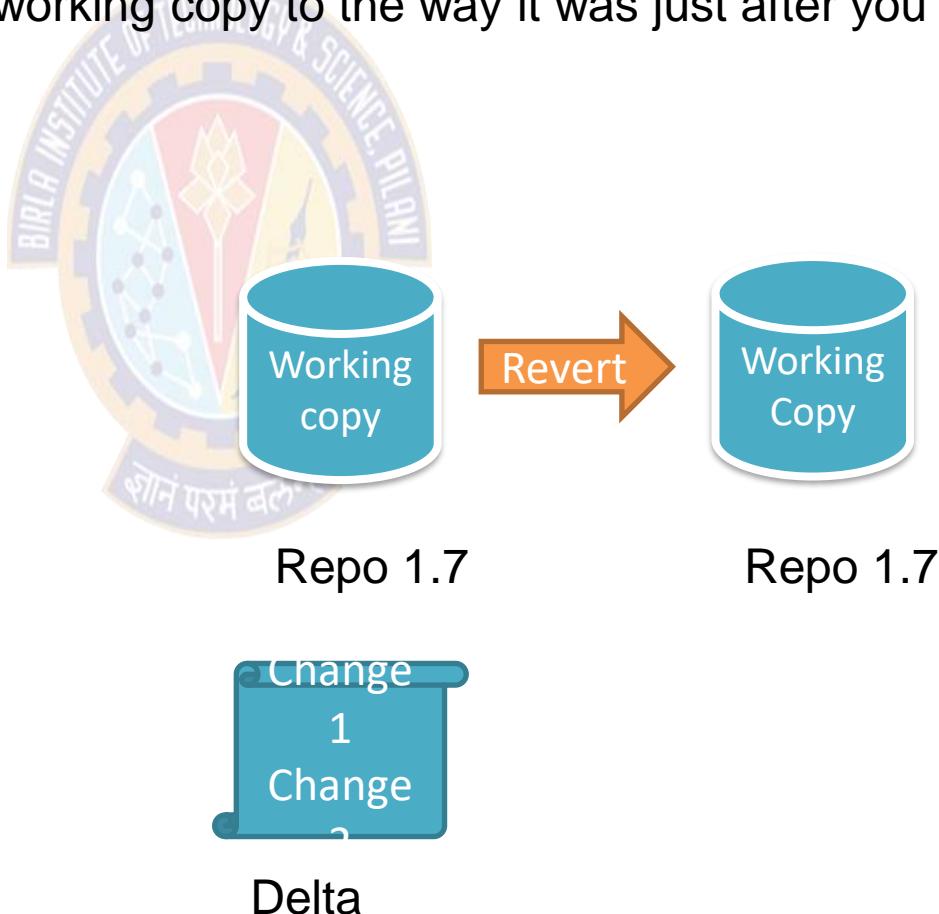
- Show the details of the modifications that have been made to the working copy
- In other hand Status provides a list of changes but no details about them
- To see exactly what changes have been made to the files, you need to use the diff operation



Version Control System Basic Operations

Revert

- Undo modifications that have been made to the working copy
- A complete revert of the working copy will throw away all your pending changes and return the working copy to the way it was just after you did the checkout



Version Control System Basic Operations

Log

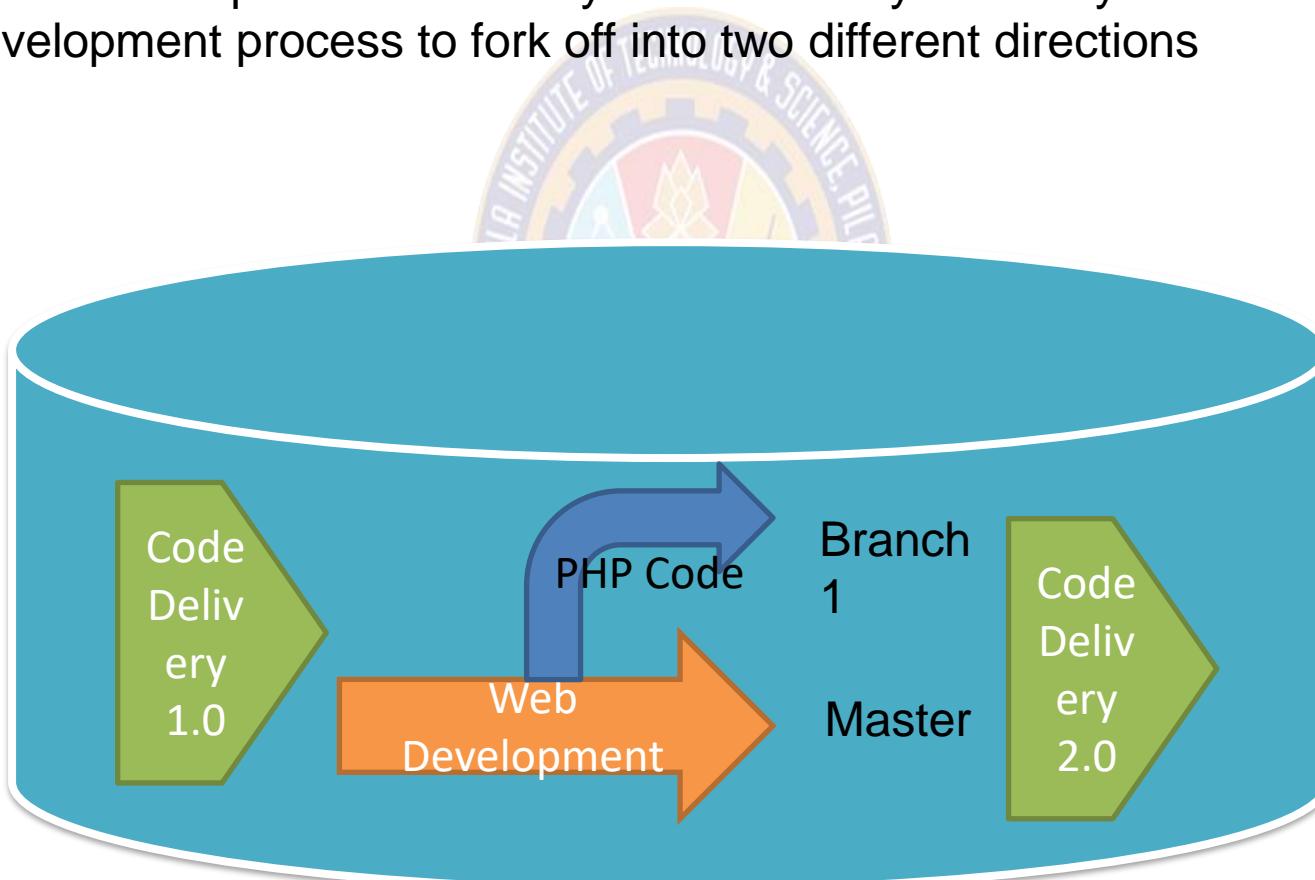
- Show the history of changes to the repository
- Your repository keeps track of every version that has ever existed
- The log operation is the way to see those records
- It displays each changeset along with additional data:
 - Who made the change?
 - When was the change made?
 - What was the log message?

- 
- Repo Created – 1 Jan 2019 – user Rathi – For Devops @ bits
 - Add File 1 – 10 Jan 2019 – user 1 – For Devops Source Code
 - Add File 2 – 12 Jan 2019 – user 1 – For Devops Source Code
 - Rename File 1 – 12 Jan 2019 – user A – For Java Ammend Code
 - Created Working Copy – 15 Jan 2019 – User B
 - Deleted File 2 – 5 Feb 2019 – User 1 – not need
 - Moved File 1 – 5 Feb 2019 – User Rathi - rename

Version Control System Basic Operations

Branch

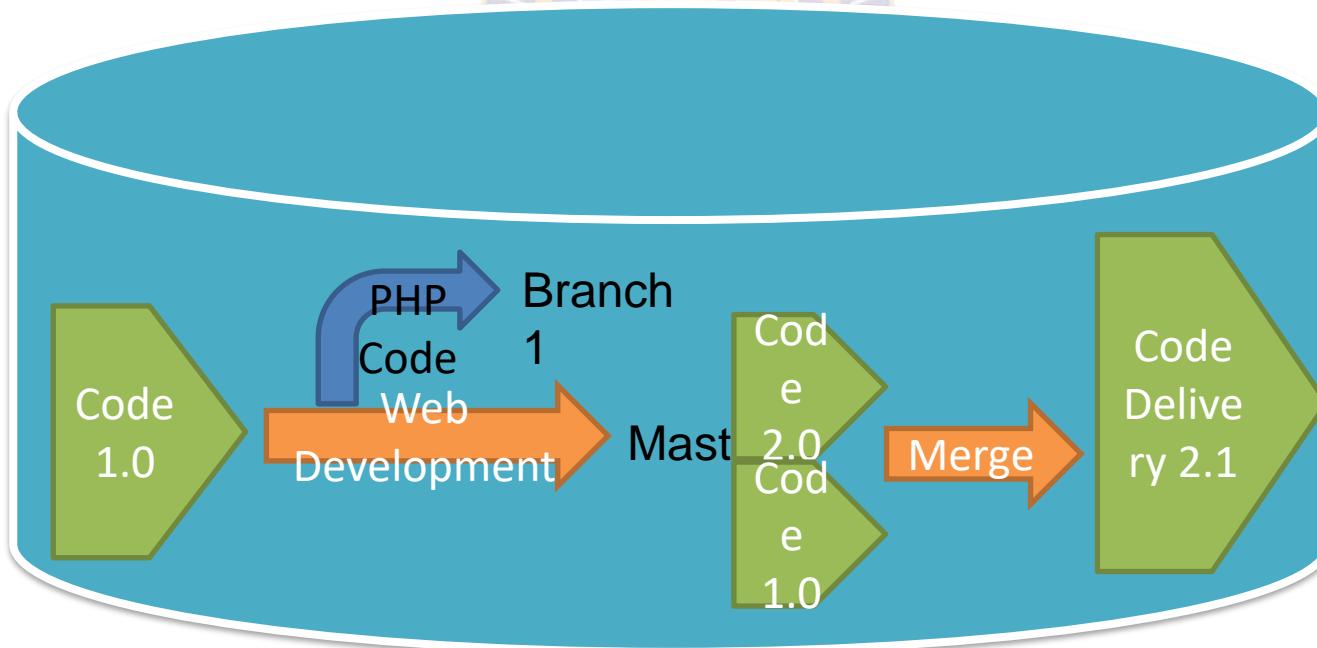
- Create another line of development
- The branch operation is what you use when you want your development process to fork off into two different directions



Version Control System Basic Operations

Merge

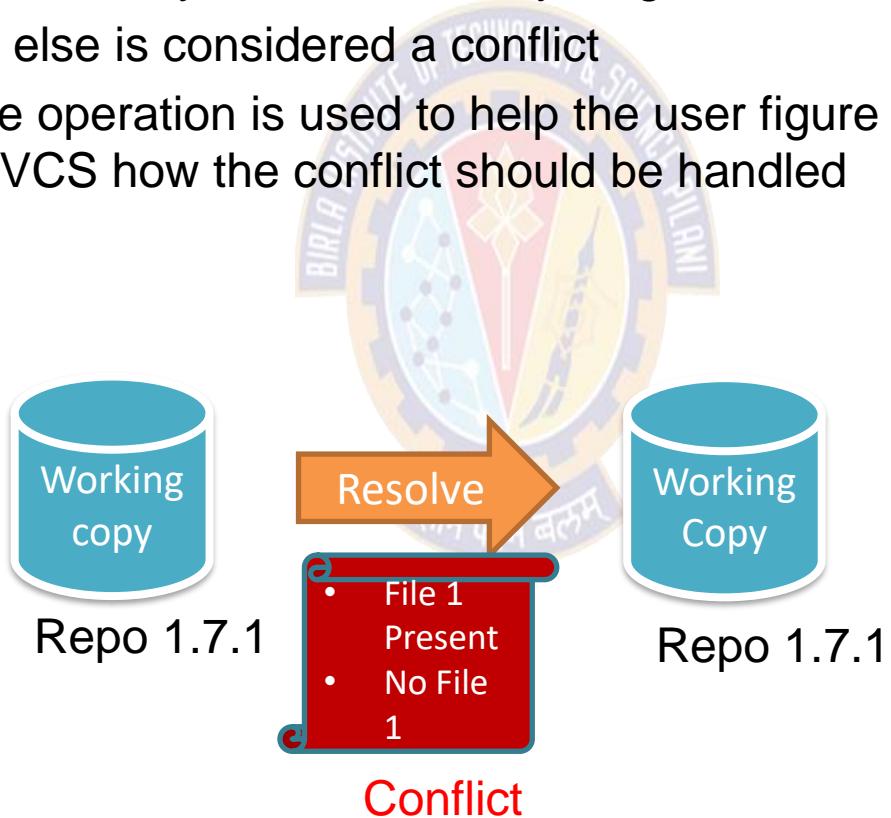
- Apply changes from one branch to another
- Typically when you have used branch to enable your development to diverge, you later want it to converge again, at least partially



Version Control System Basic Operations

Resolve

- Handle conflicts resulting from a merge
- Merge automatically deals with everything that can be done safely
- Everything else is considered a conflict
- The resolve operation is used to help the user figure things out and to inform the VCS how the conflict should be handled



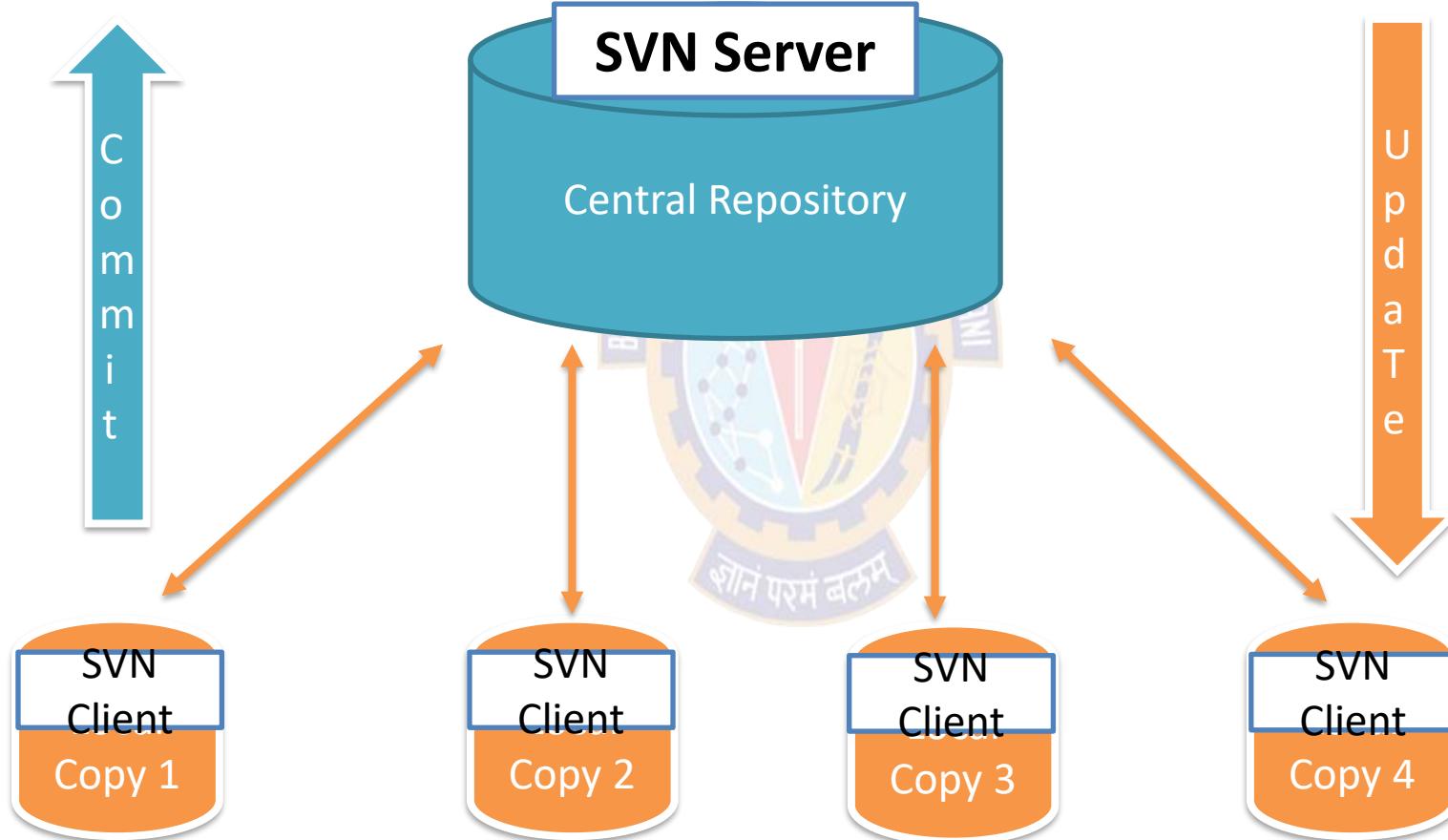
Examples of Centralized Version Control Systems

Apache Subversion (SVN)

- Subversion is a free/open source version control system
- Subversion manages files and directories, and the changes made to them, over time
- This allows you to recover older versions of your data, or examine the history of how your data changed
- Subversion is the most popular and its usage is growing larger
- Its centralized architecture make it easy to maintain a security hierarchy, access control, and backups, which is why it's preferred in many enterprise organizations over Perforce, VSS, and ClearCase
- Subversion is still widely popular amongst the open source community

Examples of Centralized Version Control Systems

Apache Subversion (SVN) Architecture



Examples of Centralized Version Control Systems

Features of Subversion

- CVS is a relatively basic version control system. For the most part, Subversion has matched or exceeded CVS's feature set where those features continue to apply in Subversion's particular design
- Subversion versions directories as first-class objects, just like files
- Copying and deleting are versioned operations. Renaming is also a versioned operation
- Subversion allows arbitrary metadata ("properties") to be attached to any file or directory
 - These properties are key/value pairs, and are versioned just like the objects they are attached to
 - These properties are not versioned, since they attach metadata to the version-space itself, but they can be changed at any time
- No part of a commit takes effect until the entire commit has succeeded
- Revision numbers are per-commit, not per-file, and commit's log message is attached to its revision
- Branches and tags are both implemented in terms of an underlying "copy" operation
- Unix users can place symbolic links under version control
- Subversion supports (but does not require) locking files so that users can be warned when multiple people try to edit the same file
- All output of the Subversion command-line client is carefully designed to be both human readable
- Subversion uses gettext() to display translated error, informational, and help messages, based on current locale settings
- The Subversion command-line client (svn) offers various ways to resolve conflicting changes, including interactive resolution prompting

Agenda

About GIT

- Examples of Distributed Version Control Systems
- Introduction to GIT
- States of GIT
- A GIT Project
- GIT workflow



About GIT

Examples of Distributed Version Control Systems

- Distributed Version Control Systems (DVCSS) don't rely on a central server
- They allow developers to clone the repository and work on that version
- Developers will have the entire history of the project on their own hard drives
- Revise:
 - Because of local commits, the full history is always available
 - No need to access a remote server (faster access)
 - Ability to push your changes continuously
 - Saves time, especially with SSH keys
 - Good for projects with off-shore developers
- Lets Discuss our Distributed Version Control System GIT



Introduction to Git

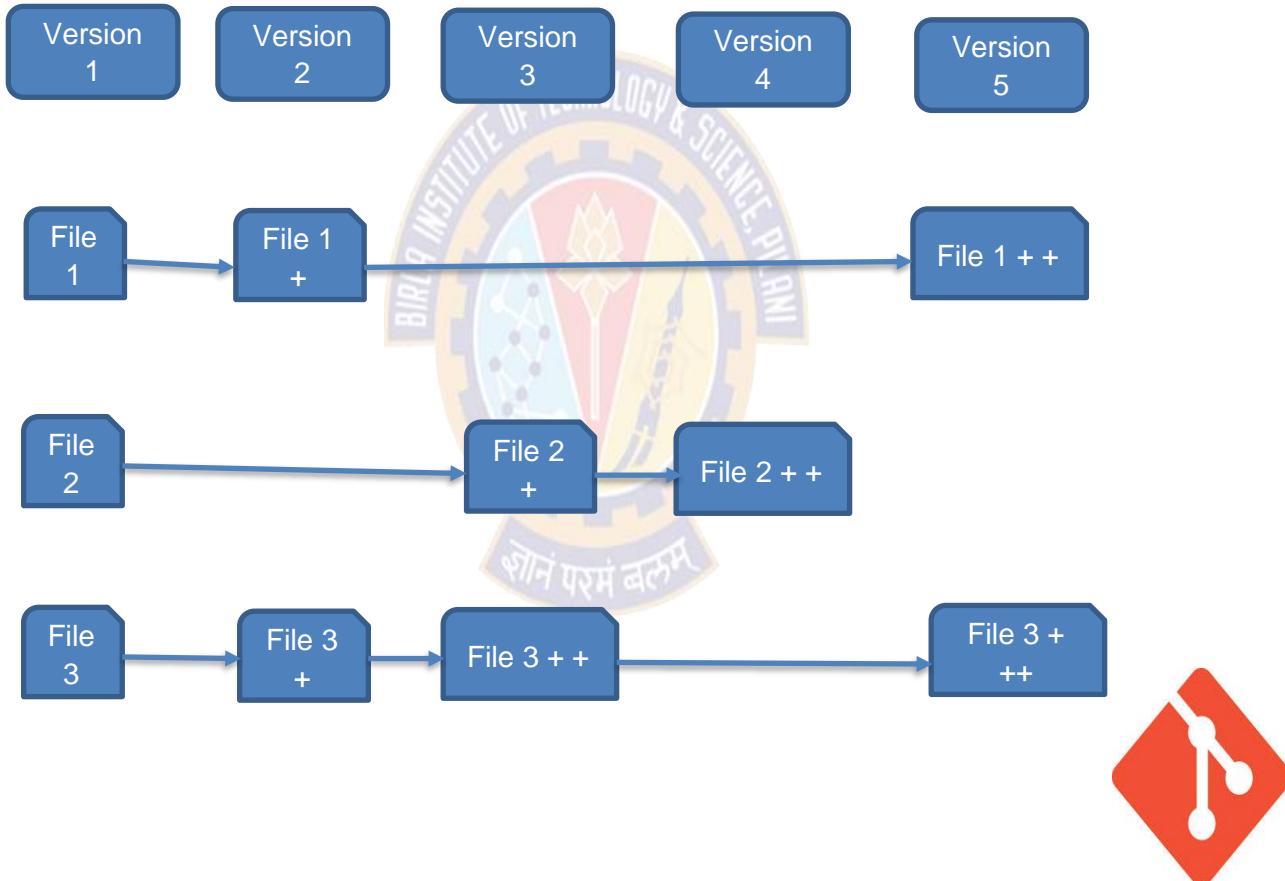
A Short History of Git

- The Linux kernel is an open source software project of fairly large scope
- For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files
- In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper
- In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down
- This prompted the Linus Torvalds the creator of Linux to develop their own tool based on some of the lessons they learned while using BitKeeper
- Since its birth in 2005, Git has evolved and matured to be easy to use
- The qualities considered for the development of Git was:
 - Speed
 - Simple design
 - Strong support for non-linear development (thousands of parallel branches)
 - Fully distributed
 - Able to handle large projects like the Linux kernel efficiently (speed and data size)



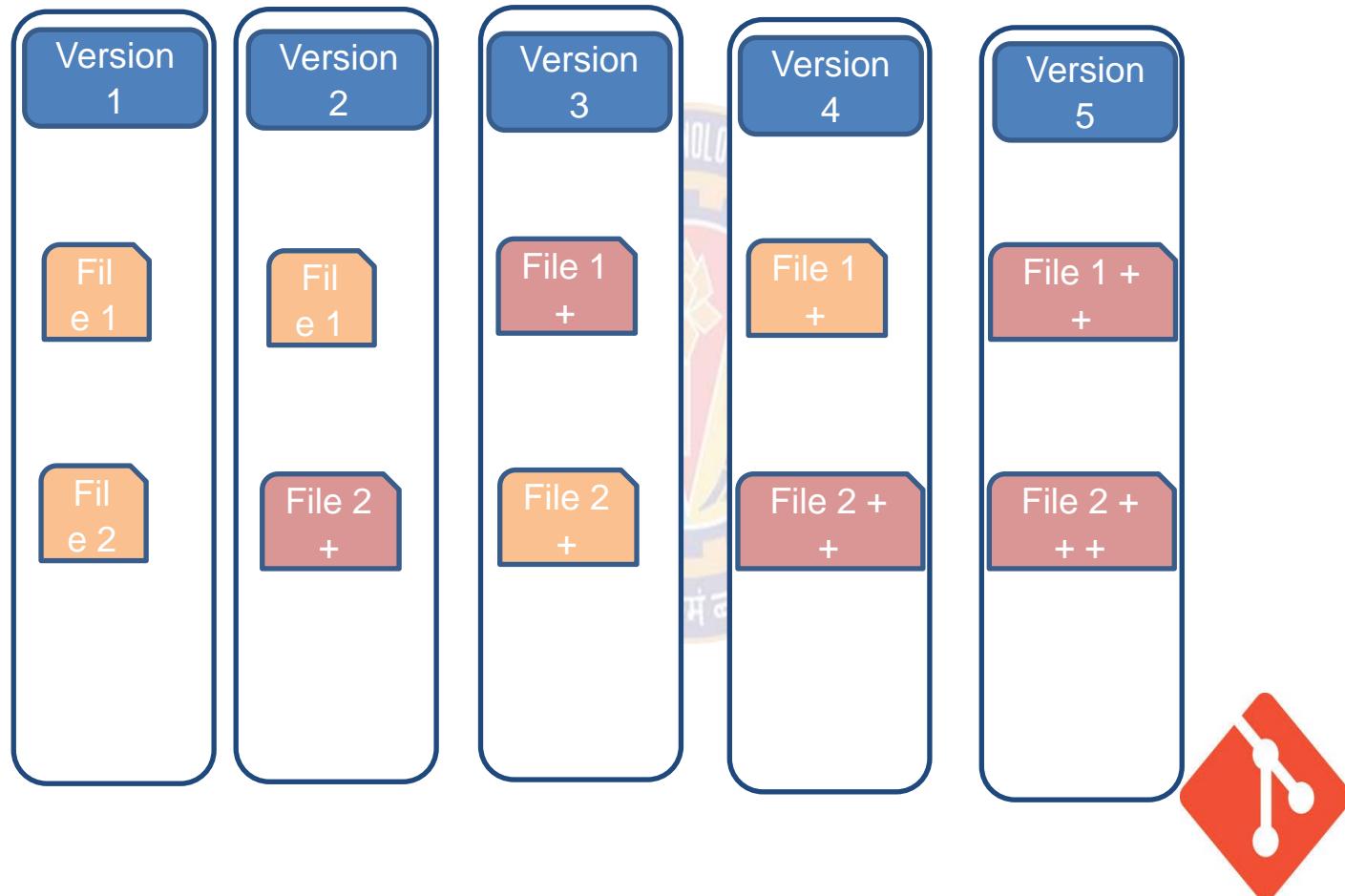
Introduction to Git

What is Different in Git?



Introduction to Git

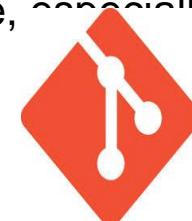
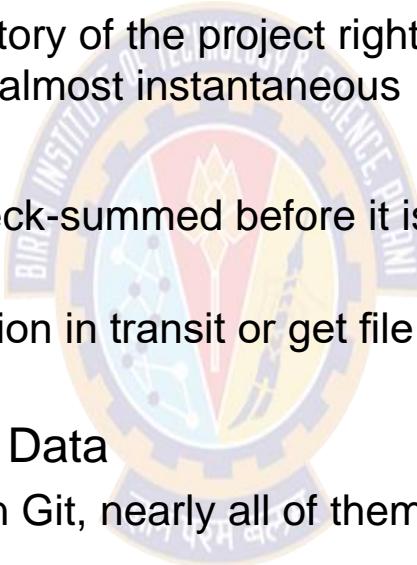
This is How Git treats:



Introduction to Git

How it works?

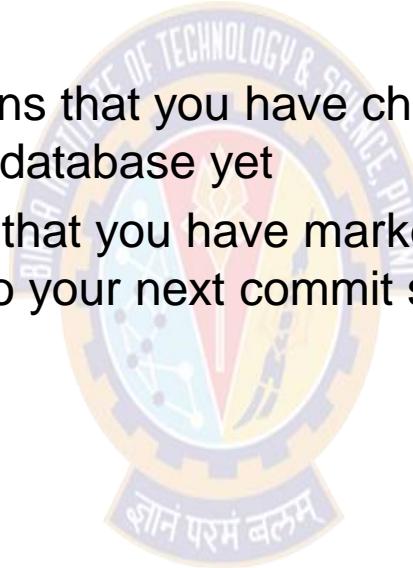
- Nearly Every Operation Is Local:
 - Most operations in Git need only local files and resources to operate
 - you have the entire history of the project right there on your local disk, most operations seem almost instantaneous
- Git Has Integrity
 - Everything in Git is check-summed before it is stored and is then referred to by that checksum
 - You can't lose information in transit or get file corruption without Git being able to detect it.
- Git Generally Only Adds Data
 - When you do actions in Git, nearly all of them only add data to the Git database
 - After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository



States of Git

The three states of Git

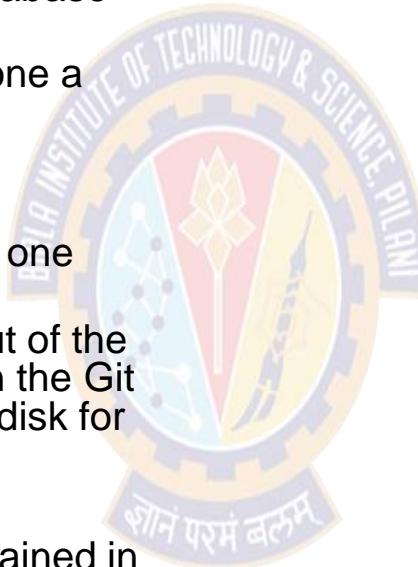
- Git has three main states that your files can reside in:
- Committed : Committed means that the data is safely stored in your local database
- Modified : Modified means that you have changed the file but have not committed it to your database yet
- Staged : Staged means that you have marked a modified file in its current version to go into your next commit snapshot



A Git Project

A Git Project will have

- The Git directory:
 - It is where Git stores the metadata and object database for your project
 - It is copied when you clone a repository from another computer
- The working tree:
 - It is a single checkout of one version of the project
 - These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify
- The staging area:
 - It is a file, generally contained in your Git directory
 - It stores information about what will go into your next commit
 - Its technical name in Git parlance



Git workflow

The basic Git workflow

- You modify files in your working tree
- You selectively stage just those changes you want to be part of your next commit; which adds only those changes to the staging area
- You do a commit; which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory



Agenda

Git workflow

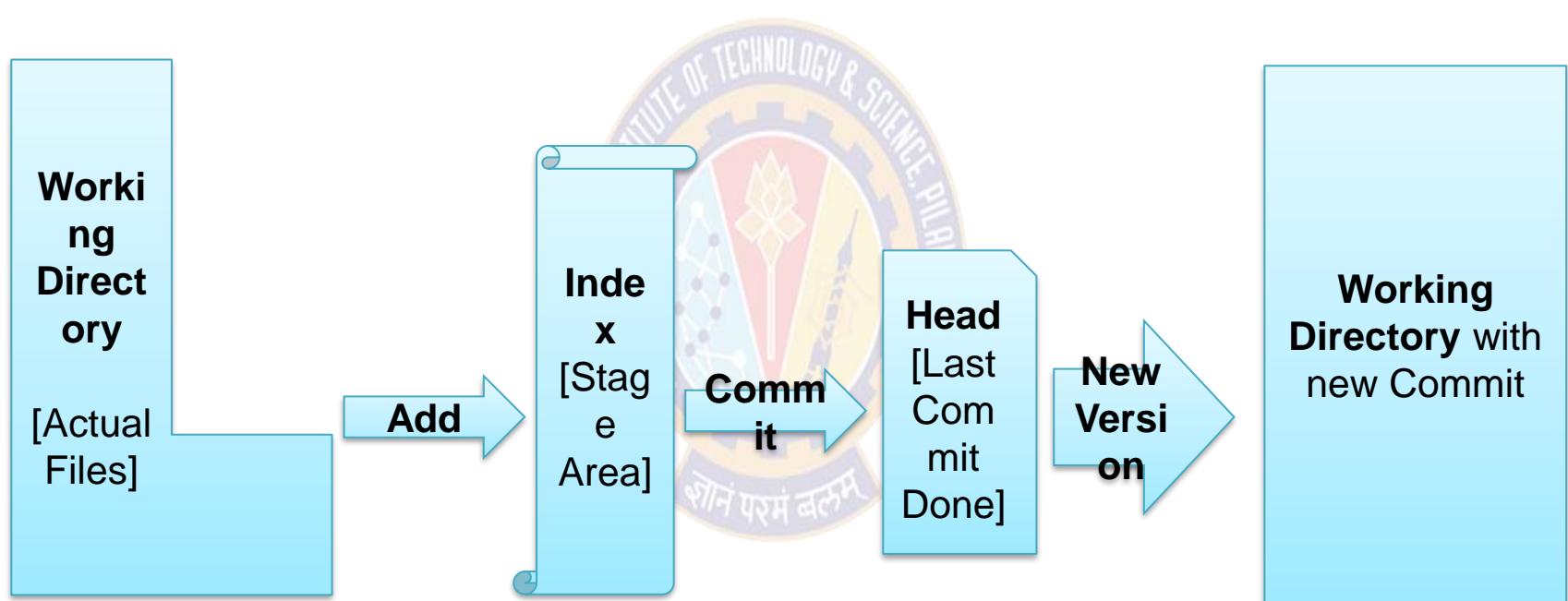
- Git tree Structure
- What is Git workflow?
- Git Centralized workflow



Git workflow

Git tree Structure

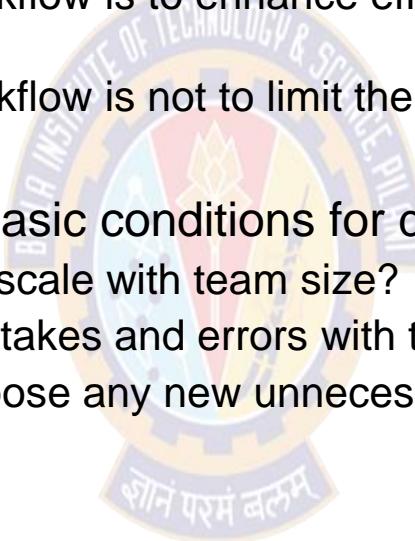
- Git local repository consists of three “trees”



Git workflow

Considerations

- To evaluate a Workflow of Git for a team you must consider:
 - Teams culture
 - Let team know the workflow is to enhance effectiveness of team and not to burden
 - Let team know the workflow is not to limit the productivity
- Selecting a Workflow? Basic conditions for decision
 - Will selected workflow scale with team size?
 - Is it easy to handle mistakes and errors with this workflow?
 - Does this workflow impose any new unnecessary cognitive overhead to the team?



Git workflow

What is Git workflow?

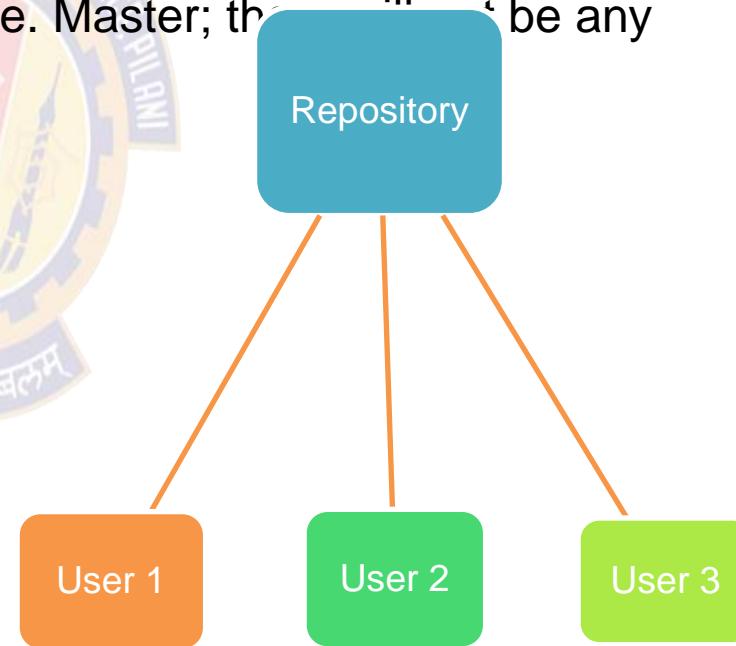
- A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner
- Git workflows encourage users to leverage Git effectively and consistently
- Git offers a lot of flexibility in how users manage changes
- There are several publicized Git workflows that may be a good fit for your team
- We will discuss:
 - Centralized Workflow
 - Feature Branching Workflow



Git Centralized workflow

Introduction

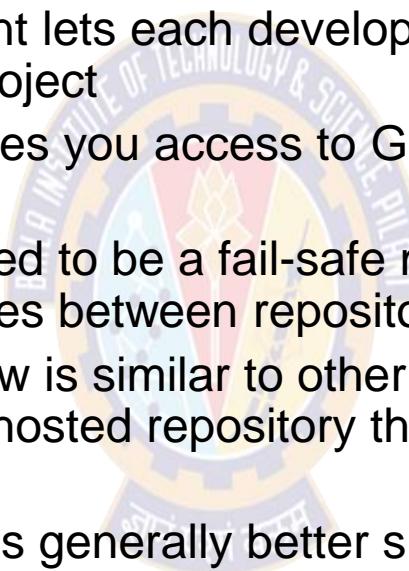
- It is a great Git workflow for teams transitioning from SVN (Subversion)
- It uses a central repository to serve as the single point-of-entry for all changes to the project
- Here there will be only one branch i.e. Master; there will not be any other branch



Git Centralized workflow

Benefits

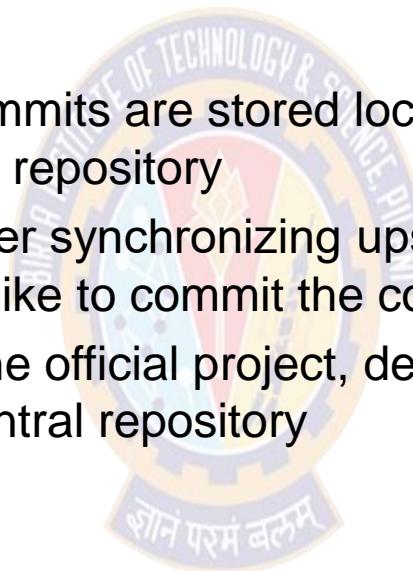
- Centralized workflow gives every developer their own local copy of the entire project
- This isolated environment lets each developer work independently of all other changes to a project
- Centralized workflow gives you access to Git's robust branching and merging model
- Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories
- The Centralized Workflow is similar to other workflows in its utilization of a remote server-side hosted repository that developers push and pull from
- A Centralized Workflow is generally better suited for teams migrating from SVN to Git and smaller size teams



Git Centralized workflow

Git Centralized workflow: How it works?

- Developer Starts Cloning the central repository
- In local copy they edit files and commit changes as they would with SVN
- However, these new commits are stored locally - they're completely isolated from the central repository
- This lets developers defer synchronizing upstream until they're at a state where they would like to commit the code to the Master
- To publish changes to the official project, developers "push" their local master branch to the central repository



Git Centralized workflow

Git Centralized workflow: Example

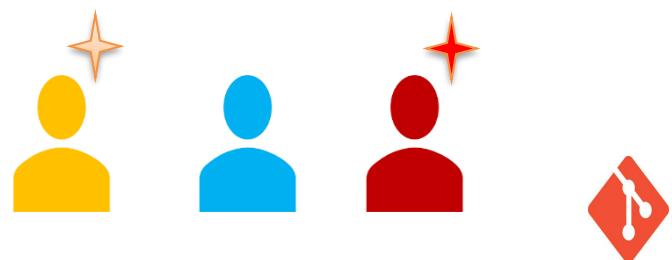
- Lets say the team of three people named Orange, Blue and Red are working on a Centralized Git Repository
- They will be collaborating with Centralized Workflow



Git Centralized workflow

Git Centralized workflow: Example Contd..

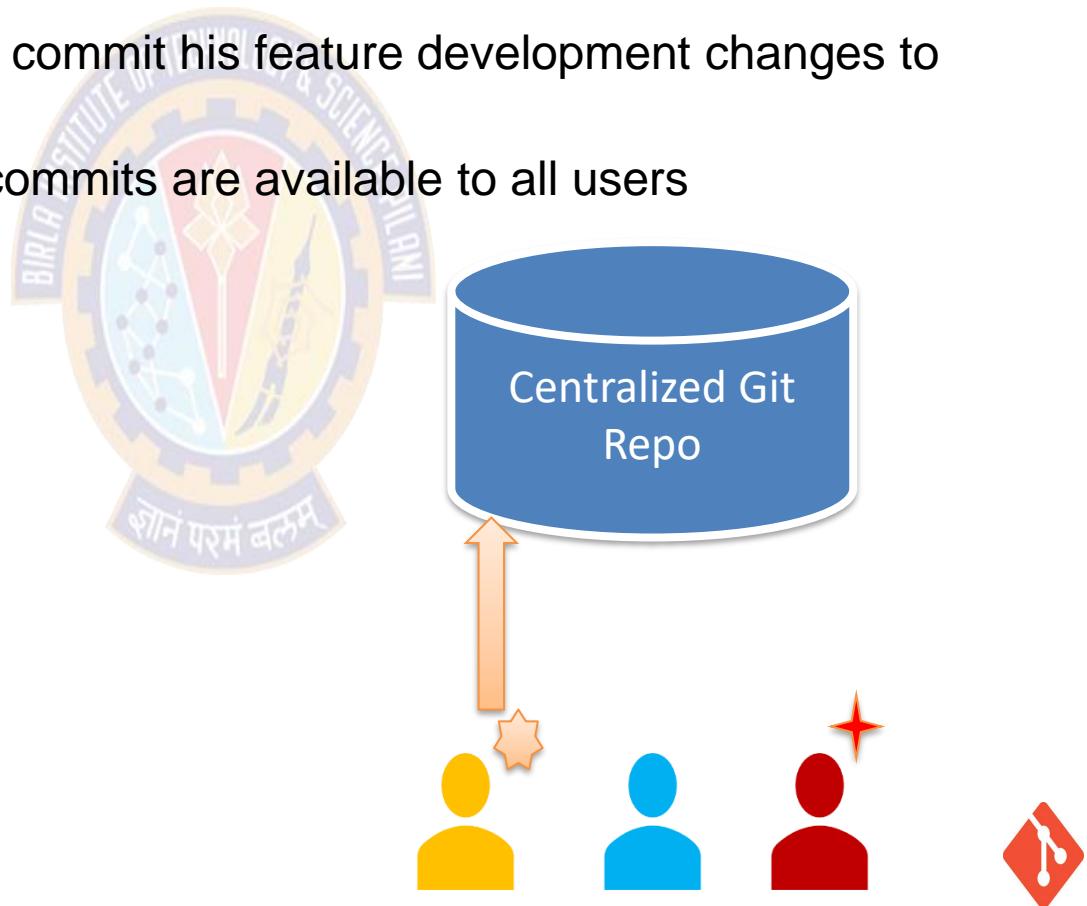
- User Orange is working on a feature development in his local repository
- At the Same time User Red is also working on a separate feature in his local repository



Git Centralized workflow

Git Centralized workflow: Example Contd..

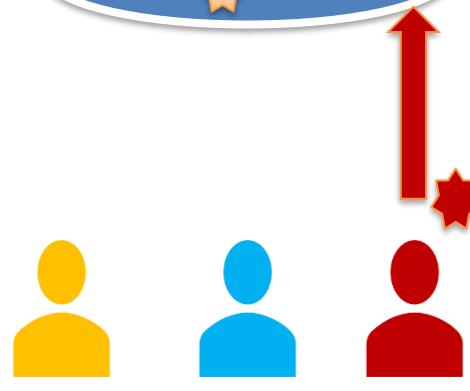
- User Orange has completed his feature development
- And user Orange publishes his feature
- I.e. User Orange should commit his feature development changes to Central repository
- So now these updated commits are available to all users



Git Centralized workflow

Git Centralized workflow: Example Contd..

- After the commit of Orange user
- User Red complete his feature development
- And now User Red would like to commit his changes to the Central Repository
- **And It Fails**



Git Centralized workflow

Git Centralized workflow: Example Contd..

- Here user Red need to pull the recent changes done to repository
- Incorporate his changes with recent changes done by User Orange
- I.e. Kind of rebase activity and resolve the conflicts if any



Git Centralized workflow

- Now user Red can commit his changes in local repository
- And Publish his feature to the central repository
- Its Successful



Git Centralized workflow

Summary

- The Centralized Workflow is great for small teams
- The conflict resolution process detailed above can form a bottleneck as your team scales in size
- This is great for transitioning teams off of SVN, but it doesn't leverage the distributed nature of Git



Agenda

Feature workflow

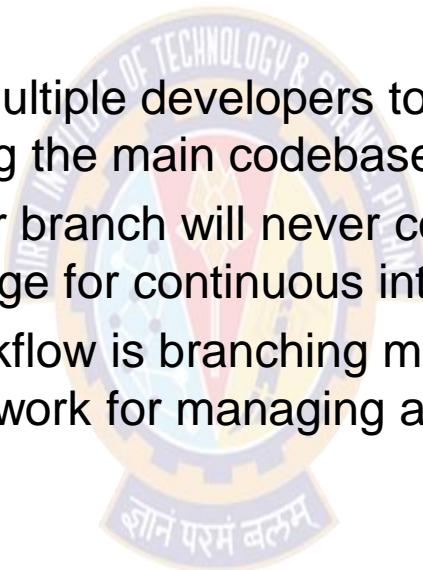
- Introduction
- How it works?



Git Feature Branch Workflow

Introduction

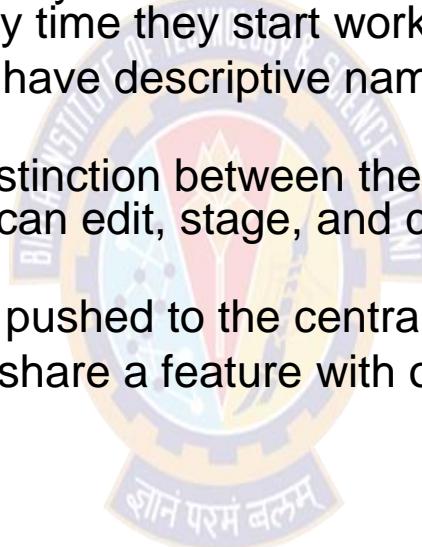
- The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch
- This makes it easy for multiple developers to work on a particular feature without disturbing the main codebase
- It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments
- Git Feature Branch Workflow is branching model focused, meaning that it is a guiding framework for managing and creating branches



Git Feature Branch Workflow

Git Feature Branch workflow: How it works?

- The Feature Branch Workflow assumes a central repository, and master represents the official project history
- Instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature
- Feature branches should have descriptive names like web page development #23
- Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch
- Feature branches can be pushed to the central repository
- This makes it possible to share a feature with other developers without touching any official code



Git Feature Branch Workflow

Git Feature Branch workflow: Example

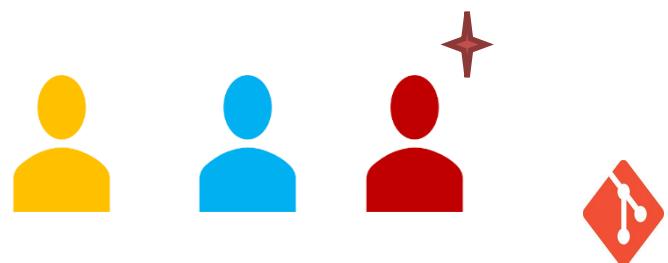
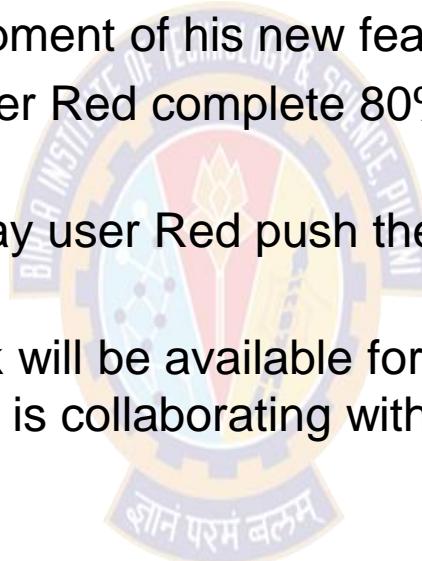
- Lets say the team of three people named Orange, Blue and Red are working on a Centralized Git Repository
- They will be collaborating with Feature Branch Workflow



Git Feature Branch Workflow

Git Feature Branch workflow: Example Contd..

- Before user Red starts working on a new feature, he request his own new branch
- User Red Starts development of his new feature
- At the end of the day user Red complete 80% of his feature development
- Before leaving for the day user Red push the commit to the Central Repository
- So the new feature work will be available for all to review and also to the member who is collaborating with user Red for feature development



Git Feature Branch Workflow

Git Feature Branch workflow: Example Contd..

- Next day User Red complete his development
- Now before merging the new Feature to master, User Red would like to let all other team member know he is done with his work by filing a pull request
- Before filing pull request User Red need to make sure the Branch is available on Central Repository



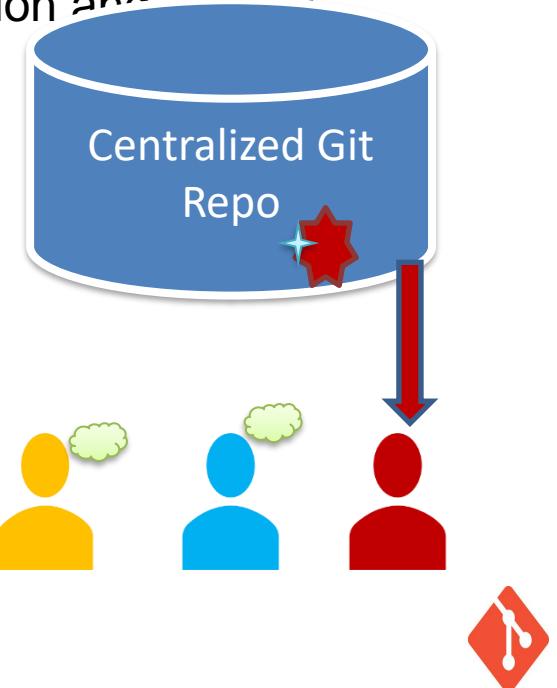
Only Push no merge
File a pull request by User Red



Git Feature Branch Workflow

Git Feature Branch workflow: Example Contd..

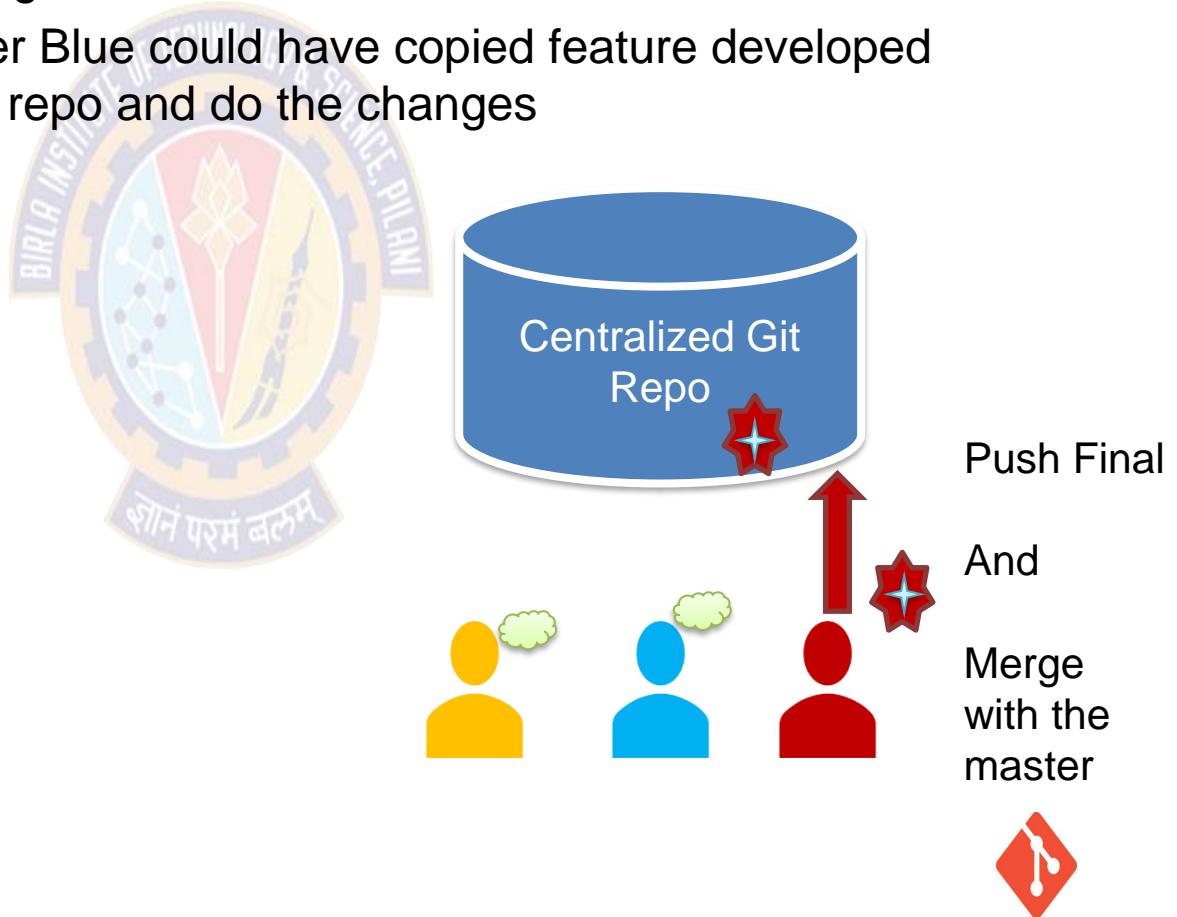
- Once User Red file a pull request Every user gets a notification that User Red is done with his feature development
- User Blue Checks the notification and visit the feature developed by User Red
- User Blue adds some more in to the feature pushed by user Red
- User Blue and User Red has discussed the addition and removed



Git Feature Branch Workflow

Git Feature Branch workflow: Example Contd..

- User Red combined the addition by user Blue and creates final Push
- And finally user Red merges feature with Master
- Even here if wanted User Blue could have copied feature developed by User Red to his local repo and do the changes



Git Feature Branch Workflow

Summary

- This workflow helps organize and track branches that are focused on business domain feature sets
- Feature Branch Workflow are focused on branching patterns
- Feature Branch Workflow promotes collaboration with team members through pull requests and merge reviews
- A feature branching model is a great tool to promote collaboration within a team environment





Thank You!

In our next session:



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps





CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 5

Agenda

Git

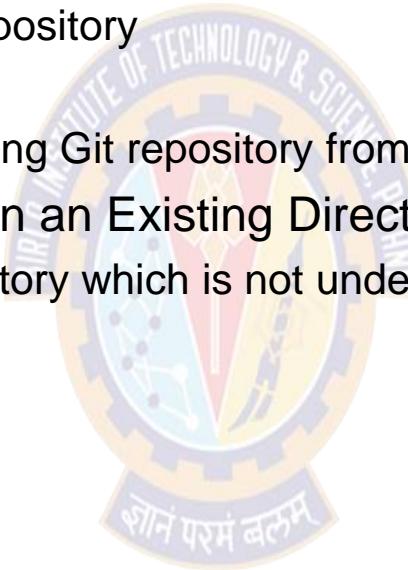
- Git basic commands



Git basic commands

Getting a Git Repository or Creating a Git Repository

- You typically obtain a Git repository in one of two ways
 - You can take a local directory that is currently not under version control, and turn it into a Git repository
 - OR
 - You can clone an existing Git repository from elsewhere
- Initializing a Repository in an Existing Directory
 - Go to the Project Directory which is not under version control
 - And type



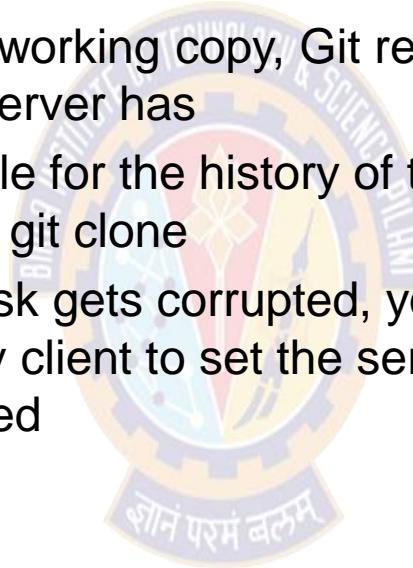
#git init



Git basic commands

Cloning an Existing Repository

- If you want to get a copy of an existing Git repository
- You should notice that the command is "clone" and not "checkout"
- Instead of getting just a working copy, Git receives a full copy of nearly all data that the server has
- Every version of every file for the history of the project is pulled down by default when you run `git clone`
- Benefit: if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned



```
#git clone <url>
```



Git basic commands

Create a working copy

- Create a working copy of a local repository

```
#git clone  
/path_to_repository
```

- When using a remote server:

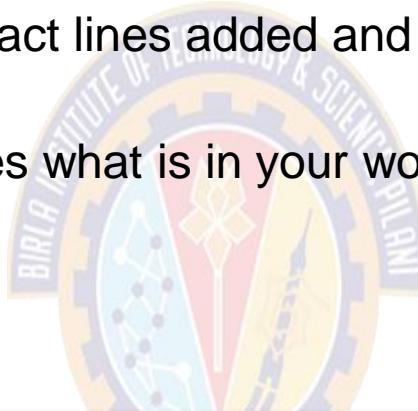
```
#git clone  
username@host:/path_to_repositor  
y
```



Git basic commands

Getting a Git Repository Status:

- The Git command to know exactly what you changed, not just which files were changed
- `git diff` shows you the exact lines added and removed — the patch, as it were
- That command compares what is in your working directory with what is in your staging area



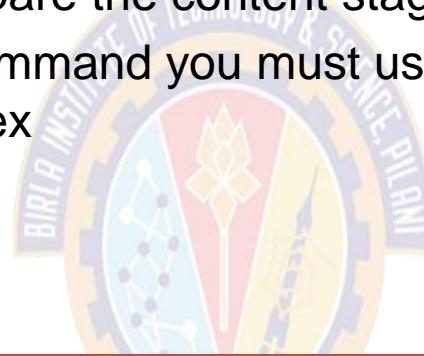
#git diff



Git basic commands

Add a file to Git Repository

- The Git command to add a file to the git repository is git add
- This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit
- Before using commit command you must use the add any new or modified files to the index



```
#git add <file  
name>
```



Git basic commands

Committing your changes

- Once the staging area is set up, you can commit your changes
- Remember that anything that is still unstaged i.e. any files you have created or modified that you haven't run git add on since you edited them, those won't go into this commit
- The default commit message contains the latest output of the git status

#git commit

**#git commit –m “Commit
Message”**



Git basic commands

Removing Files from a Git Repository

- To remove a file from Git, you have to remove it from your staging area and then commit
- The git rm command does that



```
#git rm <filename>
```

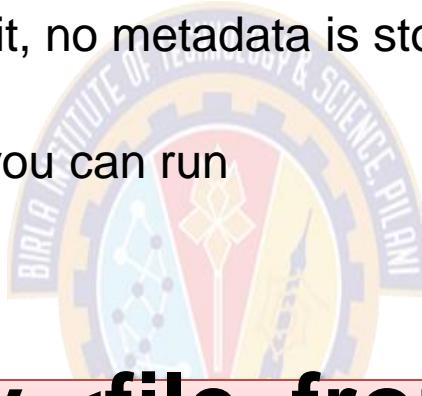


Git basic commands

Moving Files or Renaming Files in a Git Repository

- Unlike many other VCS systems, Git doesn't explicitly track file movement
- If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file
- To rename a file in Git, you can run

```
#git mv <file_from>  
      <file_to>
```



Git basic commands

Viewing the Commit History of Git Repository

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened
- The most basic and powerful tool to do this is the
- By default, with no arguments, git log lists the commits made in that repository in reverse chronological order i.e. the most recent commits show up first
- To see only the commits of a certain author

#git log

**#git log --
author=john**



Git basic commands

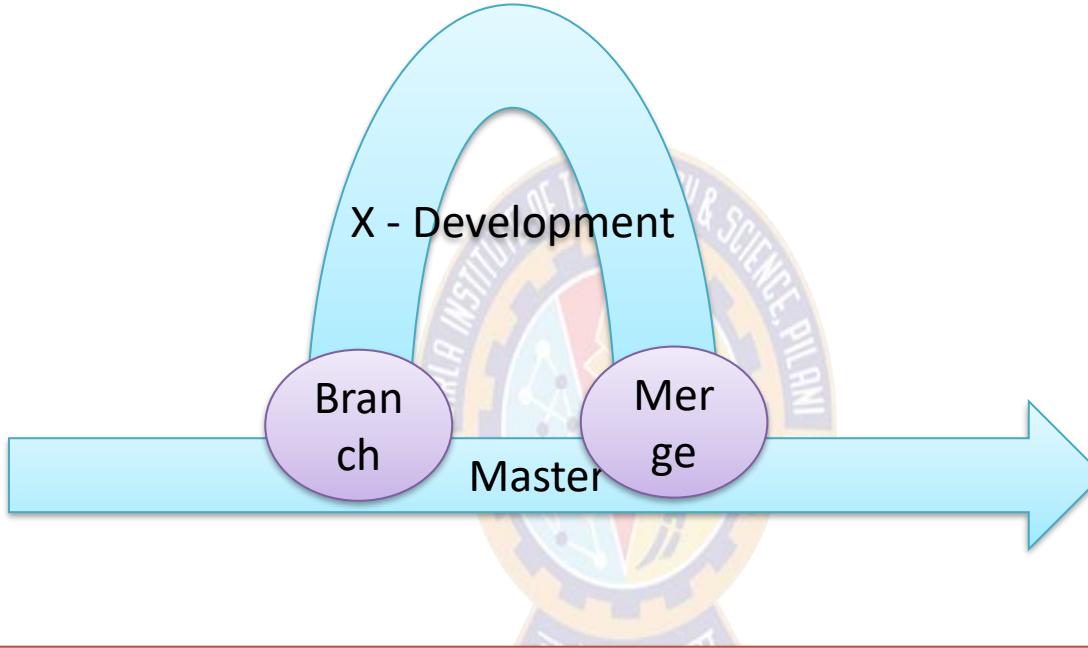
Git Branching

- As we saw nearly every Version Control System has Branching Support
- Branching means you diverge from the main line of development and continue to do work without messing with that main line
- Branches are used to develop features isolated from each other
- The way Git branches is incredibly lightweight, making branching operations nearly instantaneous
- Git doesn't store data as a series of changesets or differences
- However Git instead stores data as a series of snapshots
- Which means when you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged



Git basic commands

Git Branching Contd..



```
#git branch <branch name>
```



Git basic commands

Git Branching Contd..

- What happens if you create a new branch?
- Creation of Branch creates a new pointer for you to move around
- How does Git know what branch you're currently on?
- Git keeps a special pointer called HEAD; HEAD acts as a pointer to the local branch you're currently on
- The git branch command only created a new branch; it didn't switch to that branch, you will be still on master branch
- To switch to an existing branch, you run

```
#git checkout <branch name>
```



Git basic commands

Git Merging

- When to Merge?
- If you have completed the hotfix development or feature development or the Individual task branched, and up on completion of that now you need to add that component to the Master
- How does it work?
- First you need to checkout the branch you wish to merge in to, for an example here we will be merging in Master Branch
- Then Run

```
#git merge <name of branch>
```



Git basic commands

Getting a Git Repository Status & check conflicts

- The main tool you use to determine which files are in which state is the git status command
- The below output means you have a clean working directory — in other words, none of your tracked files are modified

#git
status

Example:\$ git status
On branch master

No commits yet

Changes to be committed:
(use "git rm --cached <file>..."
to unstage)



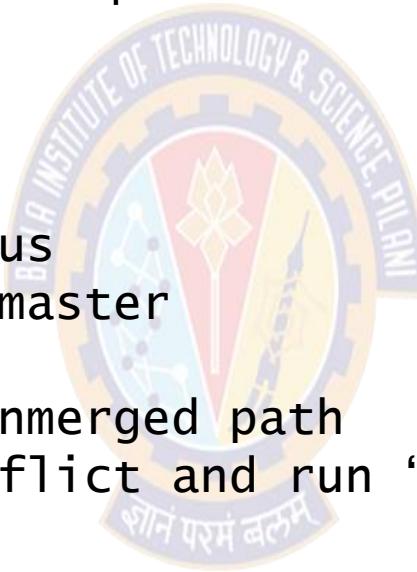
Git basic commands

Git check conflicts or status

- git status output is pretty comprehensive, it's quite self explanatory
- Lets see how the conflicts output looks like

```
$ git status  
On branch master
```

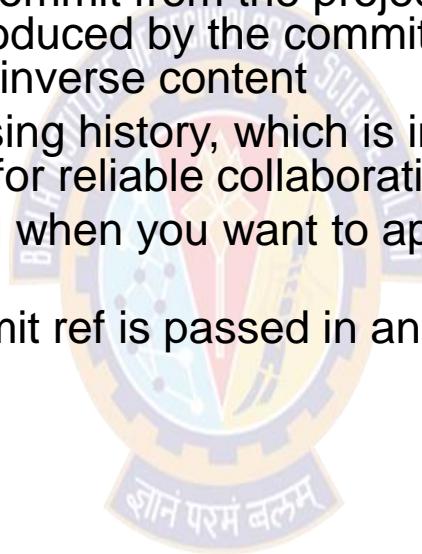
You have unmerged path
(Fix conflict and run “git-
commit”)



Git basic commands

Git Revert

- The git revert command can be considered an 'undo' type command
- however, it is not a traditional undo operation
- Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content
- This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration
- Reverting should be used when you want to apply the inverse of a commit from your project history
- Git revert expects a commit ref is passed in and will not execute without one



Git basic commands

Git Revert:

- Similar to a merge, a revert will create a new commit
- It's important to understand that git revert undoes a single commit
- It does not "revert" back to the previous state of a project by removing all subsequent commits
- The git revert command is a forward-moving undo operation that offers a safe method of undoing changes
- Instead of deleting or orphaning commits in the commit history, a revert will create a new commit that inverses the changes specified
- Git revert is a safer alternative to git reset in regards to losing work

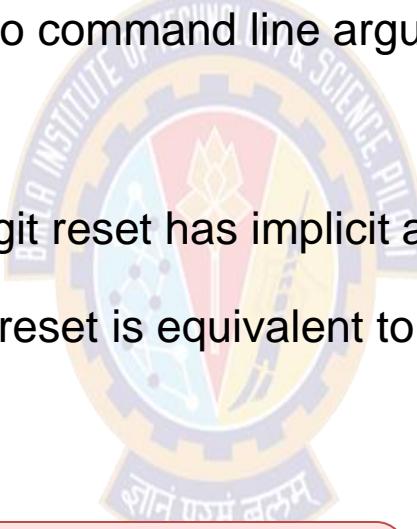
#git revert HEAD



Git basic commands

Git Reset

- The git reset command is a complex and versatile tool for undoing changes
- It has three primary forms of invocation
- These forms correspond to command line arguments
 - soft
 - mixed
 - hard
- The default invocation of git reset has implicit arguments of --mixed and HEAD
- This means executing git reset is equivalent to executing git reset --mixed HEAD



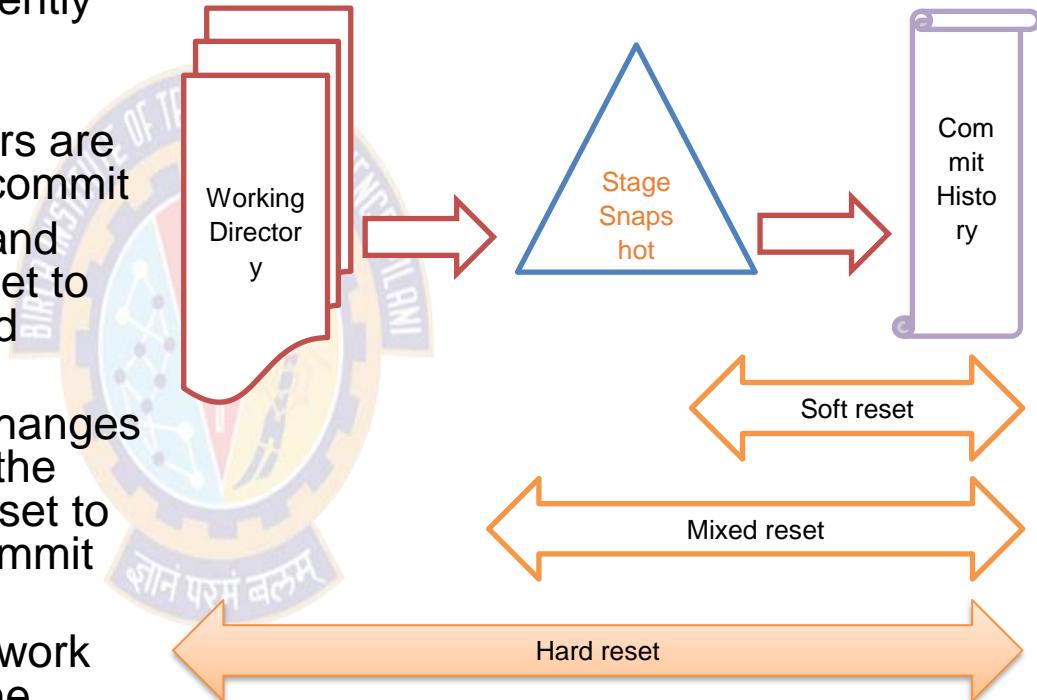
#git reset



Git basic commands

Git Reset: --hard

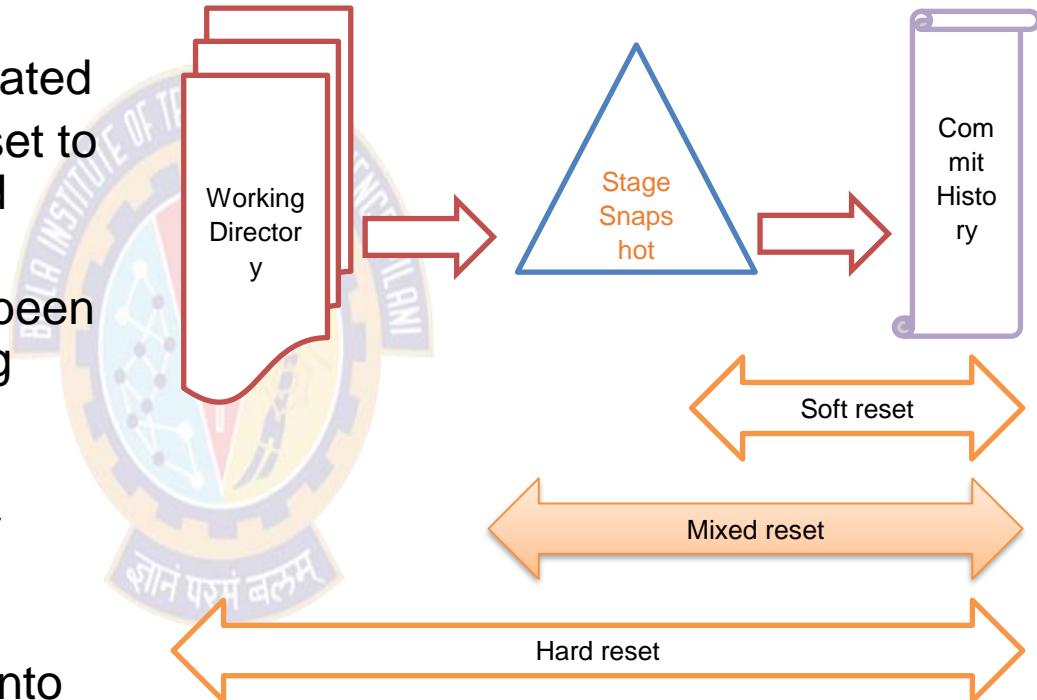
- This is the most direct, **DANGEROUS**, and frequently used option
- When passed `--hard` The Commit History ref pointers are updated to the specified commit
- Then, the Staging Index and Working Directory are reset to match that of the specified commit
- Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree
- This means any pending work that was hanging out in the Staging Index and Working Directory will be lost



Git basic commands

Git Reset: --mixed

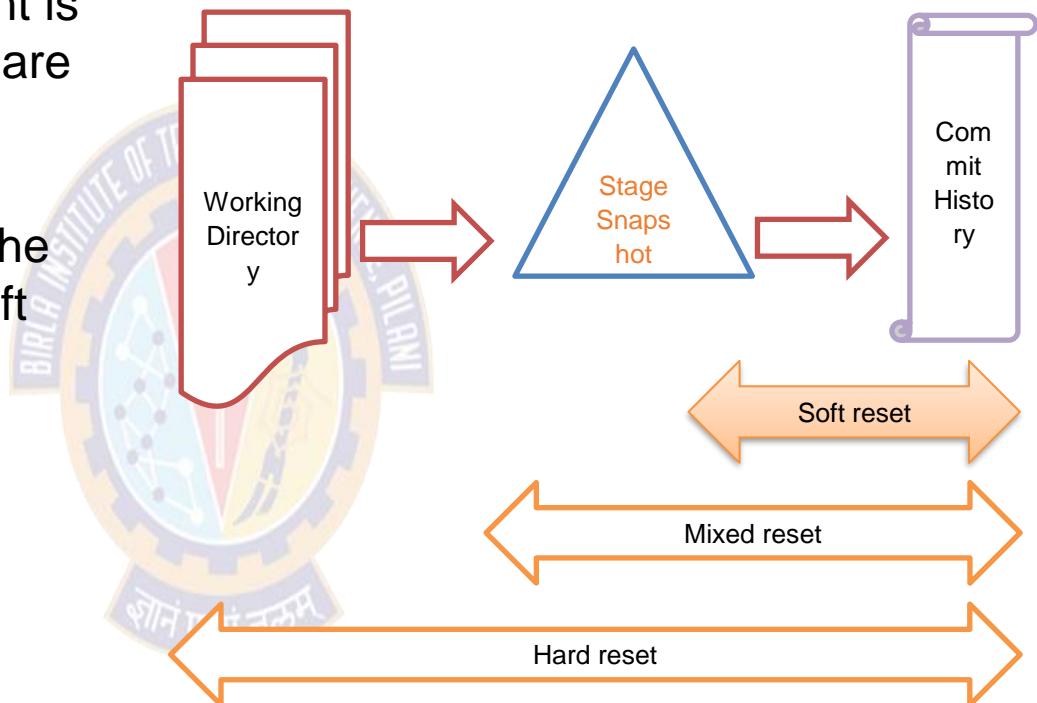
- This is the default operating mode
- The ref pointers are updated
- The Staging Index is reset to the state of the specified commit
- Any changes that have been undone from the Staging Index are moved to the Working Directory
- In other way we can say here Staging Index gets reset and the pending changes will be moved into the Working Directory



Git basic commands

Git Reset: --soft

- When the --soft argument is passed, the ref pointers are updated and the reset stops there
- The Staging Index and the Working Directory are left untouched



Git basic commands

Git Revert vs Git Reset

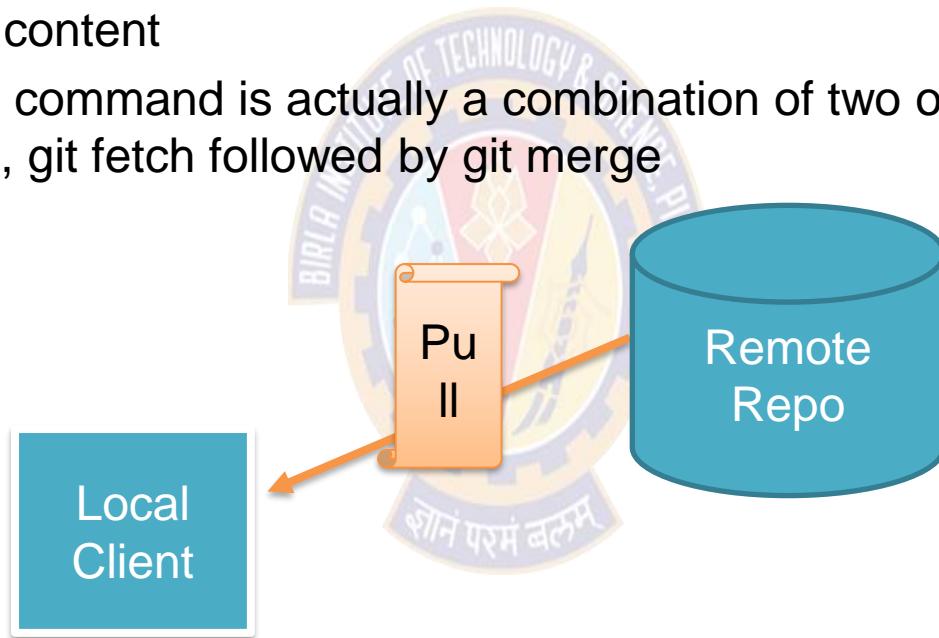
- Git Revert:
- It is a “safe” way to undo changes
- it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository
- git revert is able to target an individual commit at an arbitrary point in the history
- git revert is a safer alternative to git reset in regards to losing work
- Git Reset:
- It is the dangerous method
- There is a real risk of losing work with git reset
- git reset can only work backward from the current commit
- git reset will never delete a commit, however, commits can become 'orphaned' which means there is no direct path from a ref to access them
- For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits



Git basic commands

Git Pull

- The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content
- The git pull command is actually a combination of two other commands, git fetch followed by git merge



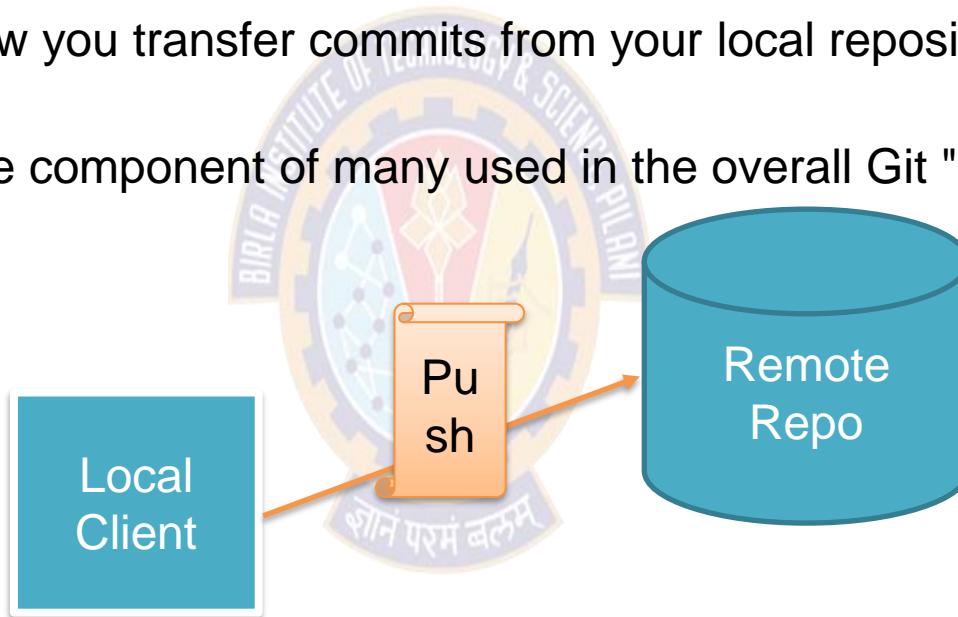
#git pull



Git basic commands

Git Push

- The git push command is used to upload local repository content to a remote repository
- Pushing is how you transfer commits from your local repository to a remote repo
- git push is one component of many used in the overall Git "syncing" process



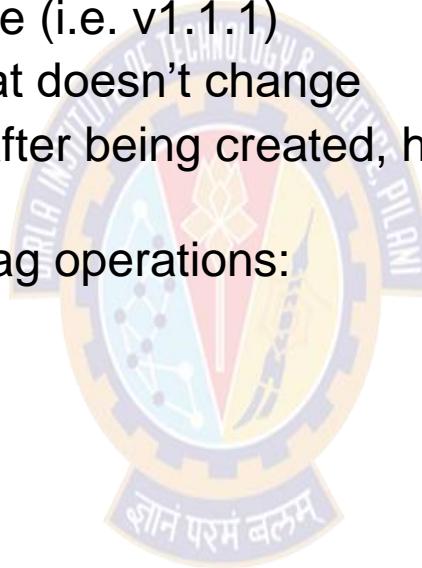
#git push



Git basic commands

Git tagging:

- Tags are ref's that point to specific points in Git history
- Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.1.1)
- A tag is like a branch that doesn't change
- Unlike branches, tags, after being created, have no further history of commits
- We will have look in to tag operations:
 - Create tag
 - List tags
 - Delete tag
 - Sharing tag



Git Tagging

Git create tag

- To create a new tag execute the following command

```
#git tag <tag name>
```

- A common pattern is to use version numbers like git tag v1.4
- Git supports two different types of tags, annotated and lightweight tags
- A best practice is to consider Annotated tags as public, and Lightweight tags as private
- Annotated tags store extra meta data such as: the tagger name, email, and date so it helps for public release
- Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, so useful for creating quick links to relevant commits



Git Tagging

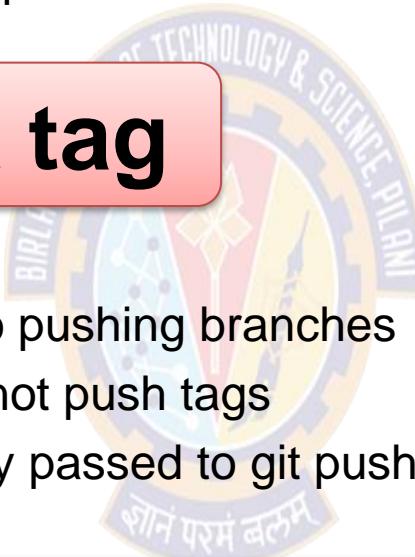
Git list tag and Git share tag

- Git list tag:
- To list stored tags in a repo execute the following command

#git tag

- Git share tag:
- Sharing tags is similar to pushing branches
- By default, git push will not push tags
- Tags have to be explicitly passed to git push

#git push origin <tag num>



Git Tagging

Git delete tag

- Deleting tags is a straightforward operation
- Passing the -d option and a tag identifier to git tag will delete the identified tag

```
#git tag -d v1
```

- Tagging is an additional mechanism used to create a snap shot of a Git repo
- Tagging is traditionally used to create semantic version number identifier tags that correspond to software release cycles



Git Tagging

Git gui

- Built in Git gui

#gitk

- Use colorful git output

**#git config
color.ui true**



Agenda

Best Practices of Clean Code

- Clean code
- General Rules



Clean code

Condition for a Clean Code

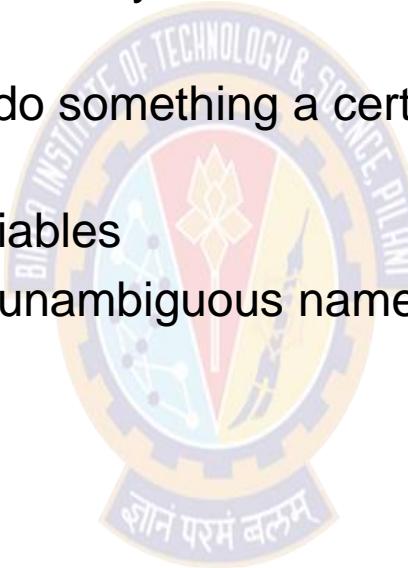
- If it can be understood easily
- It can be read and enhanced by a developer other than its original author
- If code follows readability, changeability, extensibility and maintainability



Best Practices of Clean Code

General Rules

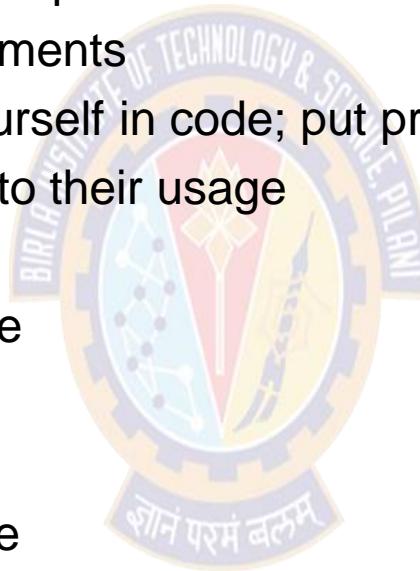
- Follow standard conventions
- Keep it simple, Simpler is always better, Reduce complexity as much as possible
- Be consistent i.e. If you do something a certain way, do all similar things in the same way
- Use self explanatory variables
- Choose descriptive and unambiguous names
- Keep functions small



Best Practices of Clean Code

General Rules Contd..

- Each Function should do one thing
- Use function names descriptive
- Prefer to have less arguments
- Always try to explain yourself in code; put proper comments
- Declare variables close to their usage
- Keep lines short
- Code should be readable
- Code should be fast
- Code should be generic
- Code should be reusable



Agenda

Dependencies

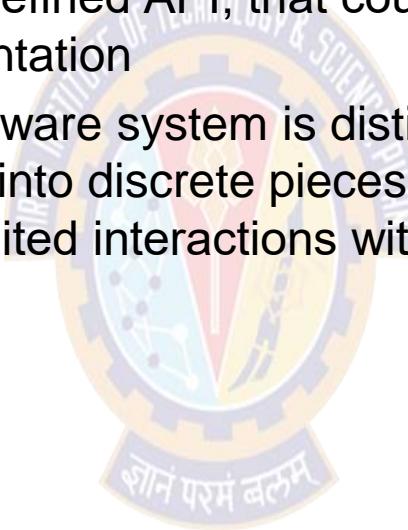
- What is component
- Best Practices for component based design
- Overview of Dependencies



Component

What is component?

- This is a overloaded term in software
- Components, mean a reasonably large-scale code structure within an application, with a well-defined API, that could potentially be swapped out for another implementation
- A component-based software system is distinguished by the fact that the codebase is divided into discrete pieces that provide behavior through well-defined, limited interactions with other components



Component

What is component? Contd..

- Components also refer as “modules”
- In Windows, a component is normally packaged as a DLL
- In UNIX, it may be packaged as an SO (Shared object) file
- In the Java world, it is probably a JAR file
- Benefits of component-based design
 - Encouraging reuse and good architectural properties such as loose coupling
 - It is one of the most efficient ways for large teams of developers to collaborate

Component

Challenges of component based design

- In large system, components form a series of dependencies, which in turn depend on external libraries
- Each component may have several release branches
- So, we have heard of projects where it takes months
 - finding good versions of each of these components
 - that can be assembled into a system which even compiles is an extremely difficult process
- Which results into ages to just release the system
- To overcome this we need to follow the best practices

Component

Keeping Your Application Releasable

- During development, teams continue an implementation of features, and sometimes need to make major architectural changes
 - In such cases the application can't be released, although it will still pass the commit stage of continuous integration
 - Usually, before release, teams will stop developing new functionality and focus only on bug fixing
 - When the application is released, a release branch is created in version control, and new development begins again
 - However, this process generally results in weeks or months between releases
- The aim of continuous delivery is for the application to always be in a releasable state
- How can we achieve this?
- One approach is to create branches in version control that are merged when work is complete, so that mainline is always releasable (feature workflow)

Component

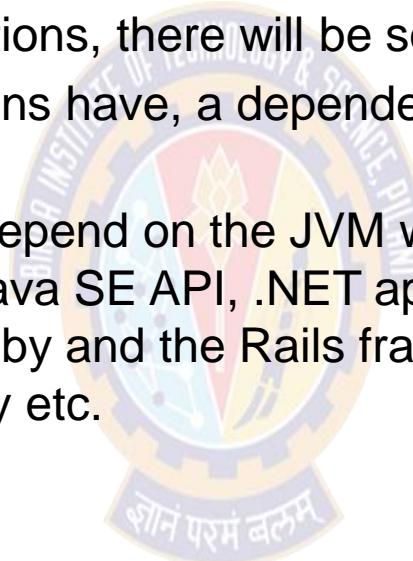
Strategies to keep your application releasable during change

- Hide new functionality until it is finished
 - problem with continuous deployment of applications is that a feature, or a set of features, can take a long time to develop which might prevent it being released
 - One solution is to put in the new features, but make them inaccessible to users
- Make all changes incrementally as a series of small changes, each of which is releasable
 - Here, analysis plays an important role to make large changes as a series of small changes
- Use branch by abstraction to make large-scale changes to the codebase
 - to make a large-scale change to an application
 - Instead of branching, an abstraction layer is created over the piece to be changed
 - A new implementation is then created in parallel with the existing implementation
 - Once it is complete, the original implementation and (optionally) the abstraction layer are removed

Dependencies

Overview of Dependencies

- A dependency occurs whenever one piece of software depends upon another in order to build or run
- In most trivial of applications, there will be some dependencies
- Most software applications have, a dependency on their host operating environment
- Like Java applications depend on the JVM which provides an implementation of the Java SE API, .NET applications on the CLR, Rails applications on Ruby and the Rails framework, C applications on the C standard library etc.



Dependencies

Types of Dependencies

- Build time dependencies and Run time dependencies
- Libraries and components

Libraries

- Software packages that your team does not control, other than choosing which to use
 - Libraries are Usually updated rarely
- This distinction is important because when designing a build process, there are more things to consider when dealing with components than libraries.
- For example, do you compile your entire application in a single step, or compile each component independently when it changes? How do you manage dependencies between components, avoiding circular dependencies?

Components

- your application depends upon, but which are also developed by your team, or other teams in your organization
- Components are usually updated frequently

Dependencies

Build time dependencies and Run time dependencies

Build-time dependencies must be present when your application is compiled and linked (if necessary)

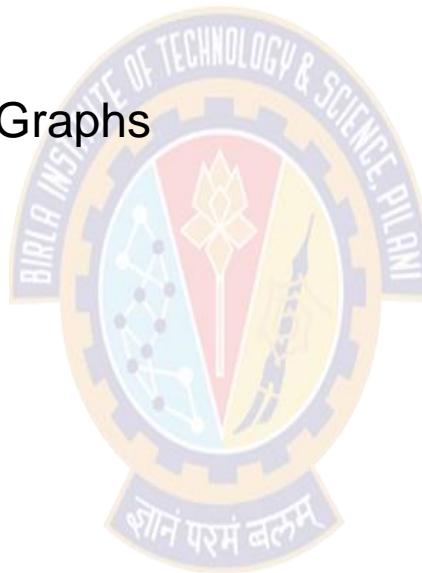
Runtime dependencies must be present when the application runs, performing its usual function

- Ex. In C and C++, your build-time dependencies are simply header files, while at run time you require a binary to be present in the form of a dynamic-link library (DLL) or shared library (SO)
- Managing dependency can be difficult

Agenda

Managing Dependency Problems

- Common dependency problem with libraries at run time
- Managing Libraries
- Managing Component
- Managing Dependency Graphs



Dependency Problems

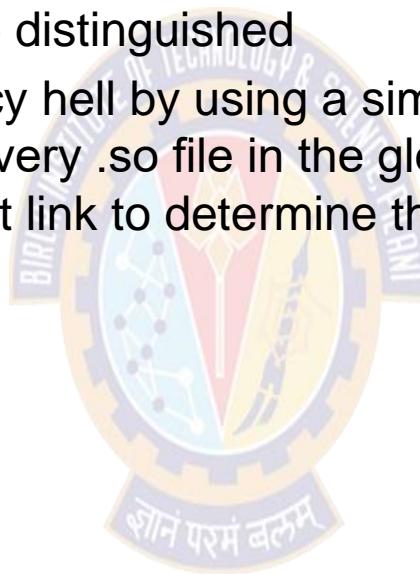
Common dependency problem with libraries at run time

- Dependency Hell also refer as DLL Hell
- Occurs when an application depends upon one particular version of something, but is deployed with a different version, or with nothing at all
- DLL hell was common problem in earlier versions of Windows,
 - as all shared libraries (DLLs), stored in a system directory (windows\system32) without any versioning; new versions would simply overwrite old ones
 - In versions of Windows prior to XP the COM class table was a singleton, so applications that required a particular COM object would be given whichever version had been loaded first

Dependency Problems

Common dependency problem with libraries at run time contd..

- The introduction of the .NET framework resolves the DLL hell problem by introducing the concept of assemblies that allow different versions of the same library to be distinguished
- Linux avoids dependency hell by using a simple naming convention: It appends an integer to every .so file in the global library directory (/usr/lib), and uses a soft link to determine the canonical system-wide version



Managing Libraries

Implementing Version Control

- One is to check them into version control
 - Is the simplest solution, and will work fine for small projects
 - a lib directory is created in your project's root to put libraries and by adding three further subdirectories: build, test, and run—for build-time, test-time, and runtime dependencies
 - using a naming convention for libraries that includes their version number; you know exactly which versions you're using
- Benefit:
 - Everything you need to build your application is in version control
 - Once you have a local check-out of the project repository, you know you can repeatably build the same packages that everybody else has
- Problems:
 - your checked-in library repository may become large and it may become hard to know which of these libraries are still being used by your application
 - Another problem crops up if your project must run with other projects on the same platform
 - Manually managing transitive dependencies across projects rapidly becomes painful

Managing Libraries

Automated

- Another solution is to declare libraries and use a tool like Maven or Ivy to download libraries from Internet repositories or (preferably) your organization's own artifact repository [Automated]



Managing Component

Overview

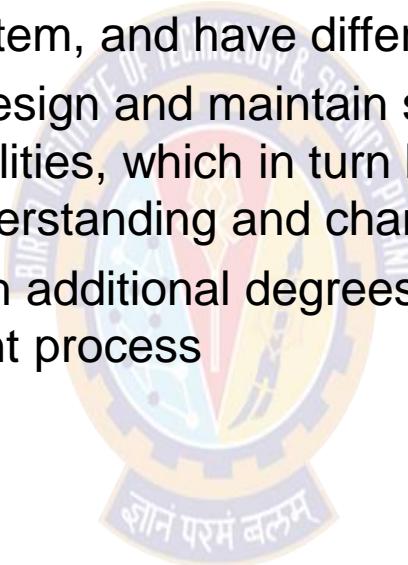
- Almost all modern software systems consist of a collection of components
- These components may be DLLs, JAR files, OSGi bundles, Perl modules, or something else
- Components have a relatively long history in the software industry



Managing Component

Components make development processes efficient

- They divide the problem into smaller and more expressive chunks
- Components often represent differences in the rates of change of different parts of the system, and have different lifecycles
- They encourage us to design and maintain software with clear description of responsibilities, which in turn limits the impact of change, and makes understanding and changing the codebase easier
- They can provide us with additional degrees of freedom in optimizing our build and deployment process



Managing Component

Components to be separated from Codebase

- Part of your codebase needs to be deployed independently (for example, a server or a rich client)
- You want to turn a monolithic codebase into a core and a set of plugins, perhaps to replace some part of your system with an alternative implementation, or to provide user extensibility
- The component provides an interface to another system (for example a framework or a service which provides an API)
- It takes too long to compile and link the code
- It takes too long to open the project in the development environment
- Your codebase is too large to be worked on by a single team

Managing Component

Pipelining Components

- Split your system into several different pipelines
- Parts of your application that have a different lifecycle (perhaps you build your own version of an OS kernel as part of your application, but you only need to do this once every few weeks)
- Functionally separate areas of your application that are worked on by different (perhaps distributed) teams may have components specific to those teams
- Components that use different technologies or build processes
- Shared components that are used by several other projects
- Components that are relatively stable and do not change frequently
- It takes too long to build your application, and creating builds for each component will be faster

Managing Component

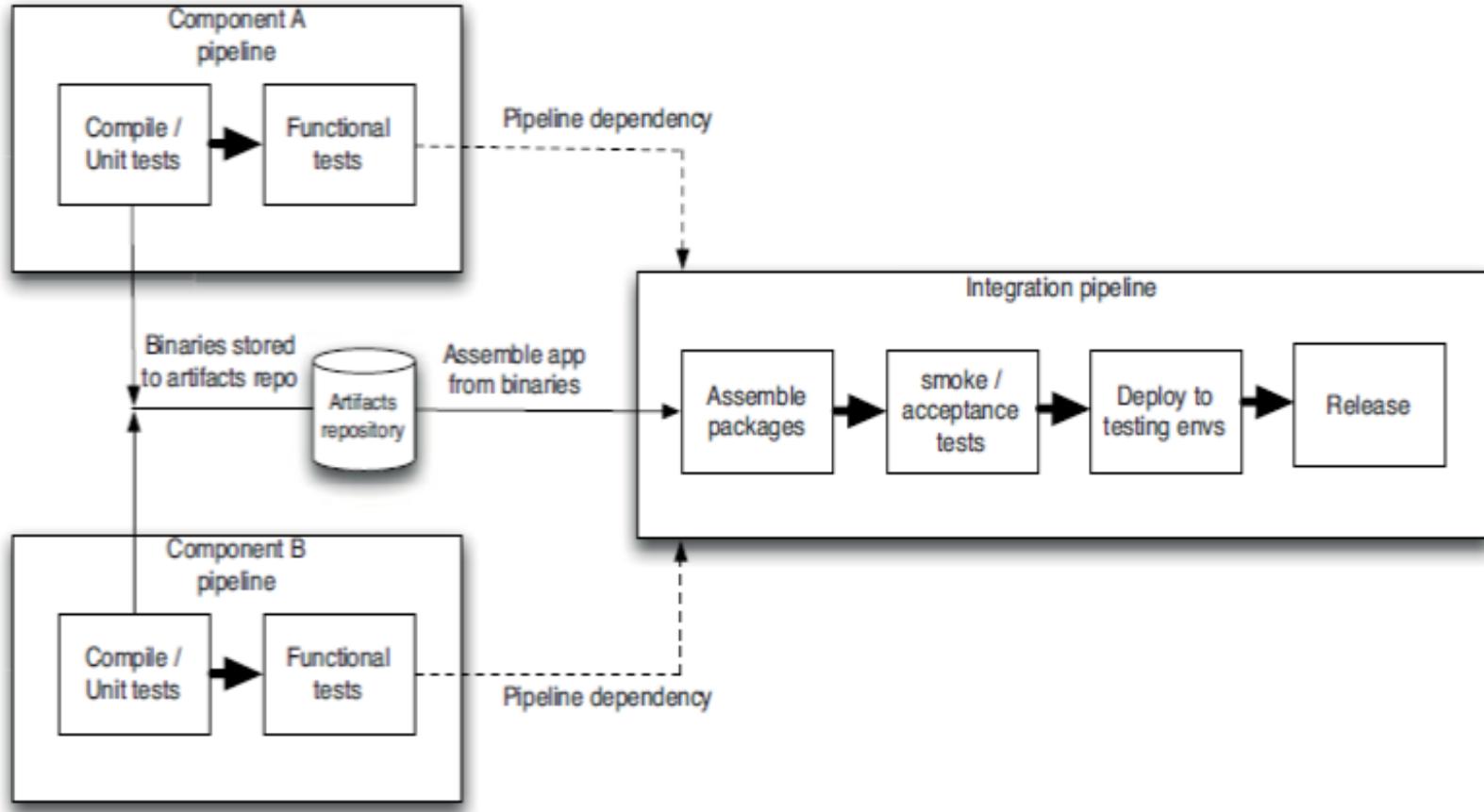
Pipelining Components

- The build for each component or set of components should have its own pipeline to prove that it is fit for release
- This pipeline will perform the following steps
 - Compile the code, if necessary
 - Assemble one or more binaries that are capable of deployment to any environment
 - Run unit tests
 - Run acceptance tests
 - Support manual testing, where appropriate



Managing Component

Integration Pipeline



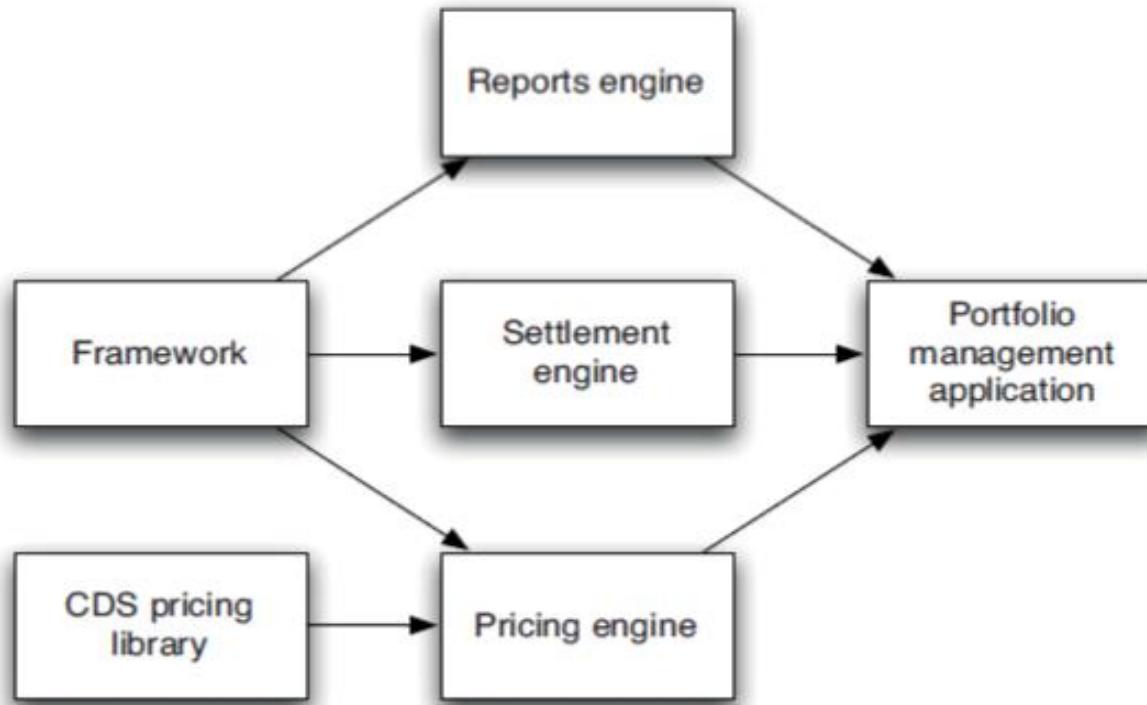
Managing Component

Managing Dependency Graphs

- Having set of components, each with their own pipeline, feeding into an integration pipeline which assembles the application and runs automated and manual tests on the final application
- However, things are often not quite this simple:
- Components can have dependencies on other components, including third-party libraries
- If you draw a diagram of the dependencies between components, it should be a directed acyclic graph (DAG)
- If this is not the case (and in particular, if your graph has cycles) you have a pathological dependency problem

Managing Component

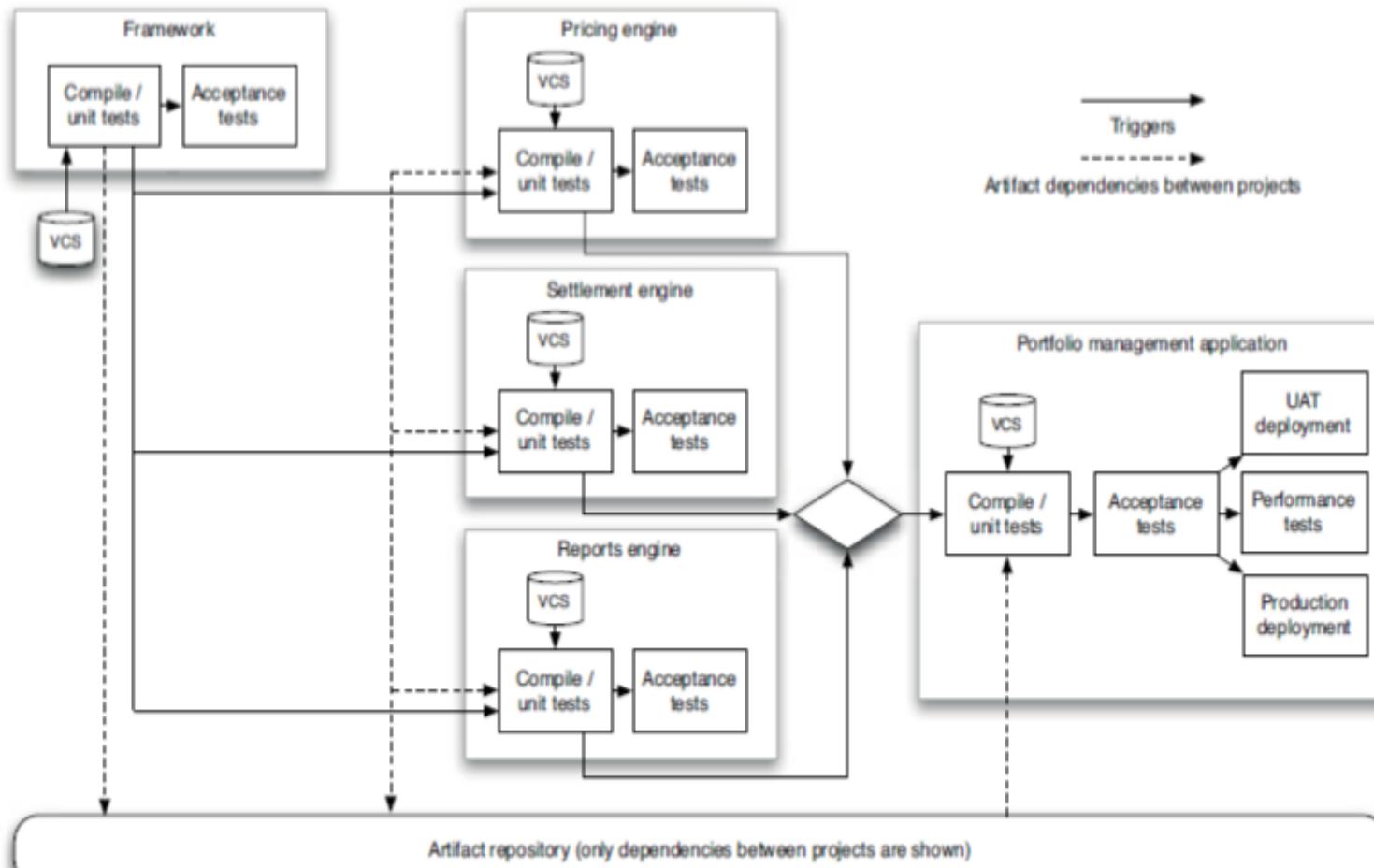
Building Dependency Graphs



credit default swap (CDS)
library that is provided by third
party

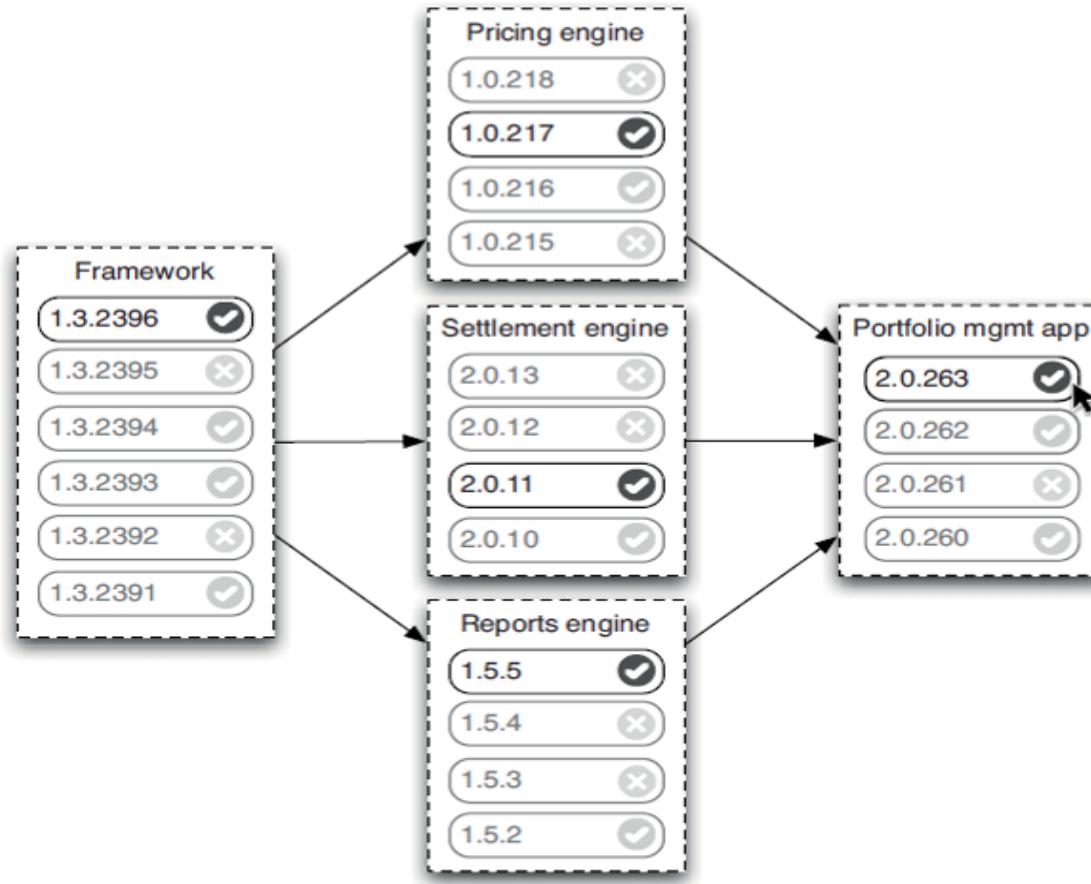
Managing Component

Pipelining Dependency Graphs



Managing Component

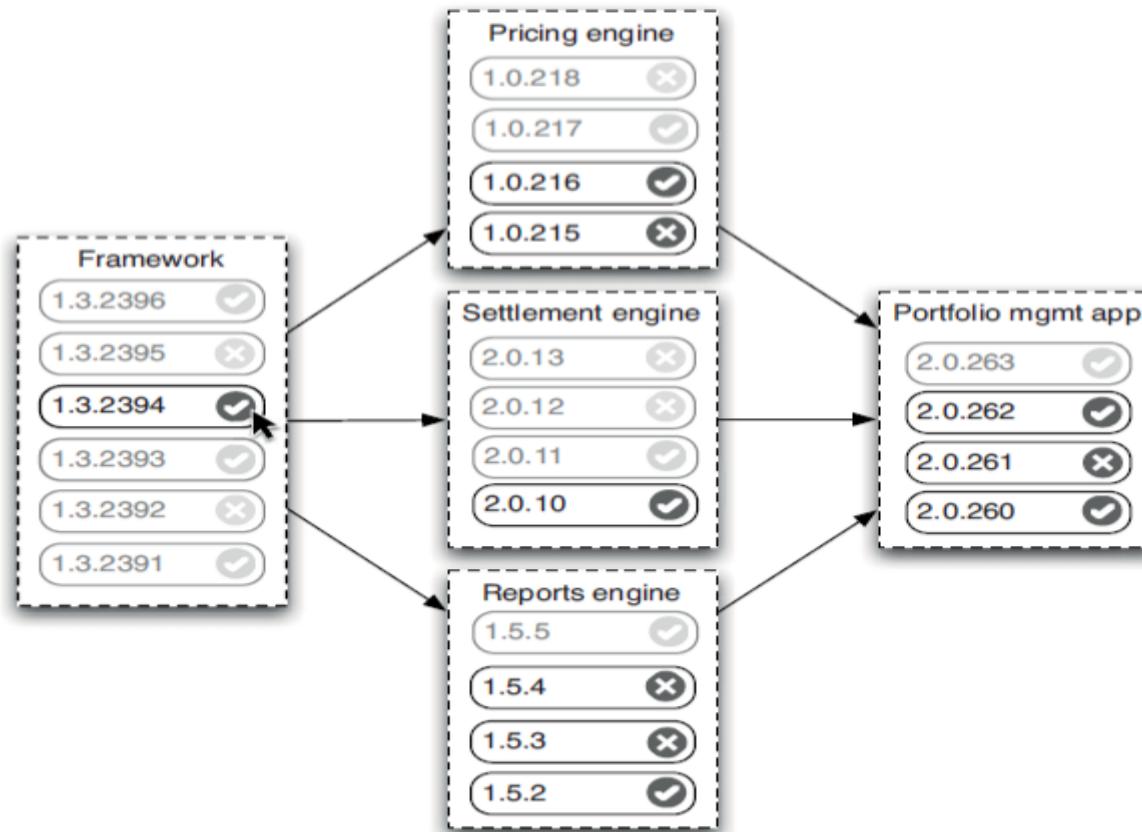
Pipelining Dependency Graphs Contd..



Visualizing upstream dependencies

Managing Component

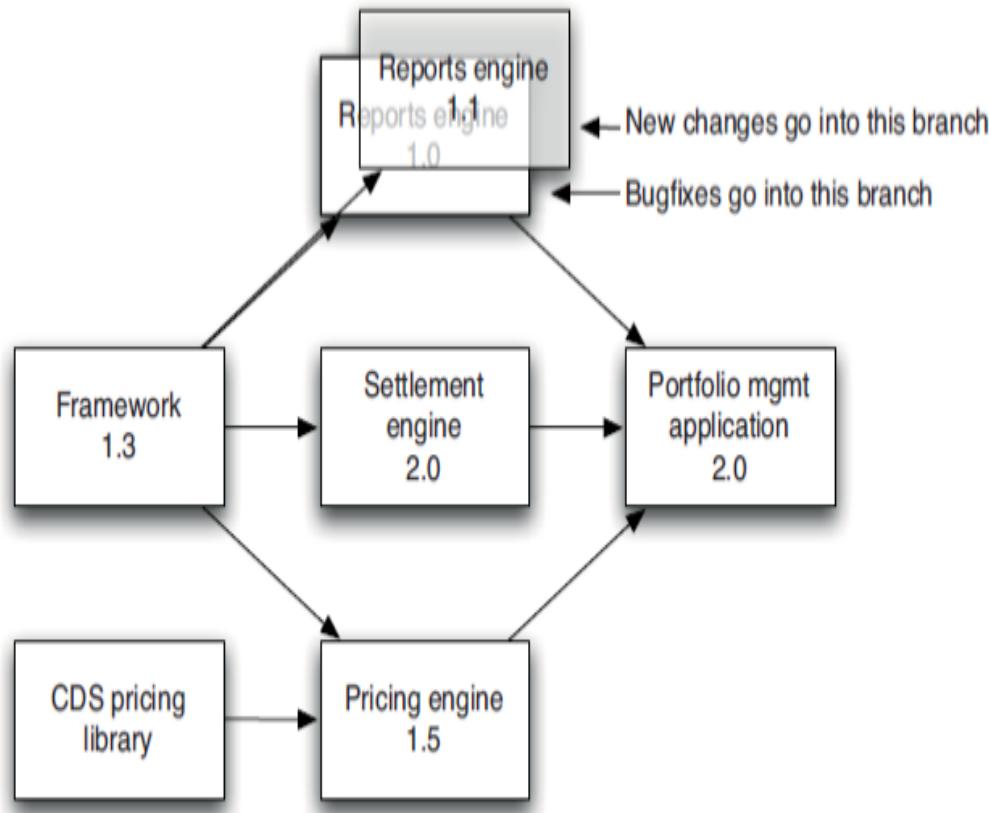
Pipelining Dependency Graphs Contd..



Visualizing downstream dependencies

Managing Component

Pipelining Dependency Graphs Contd..

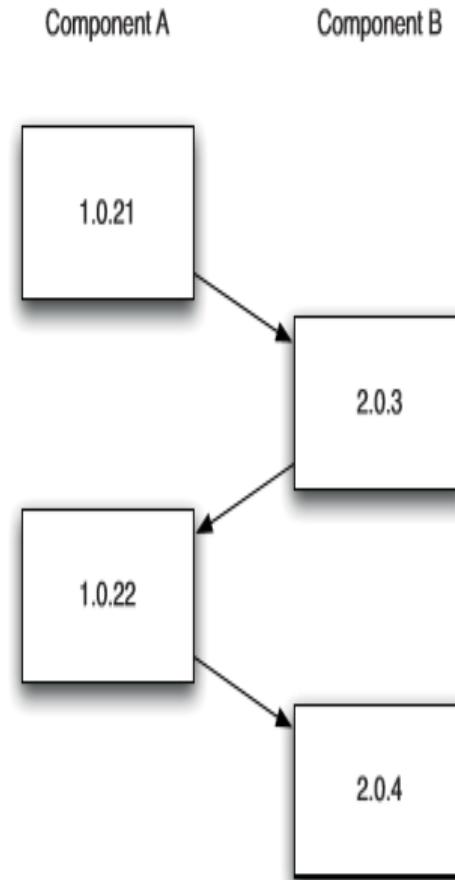
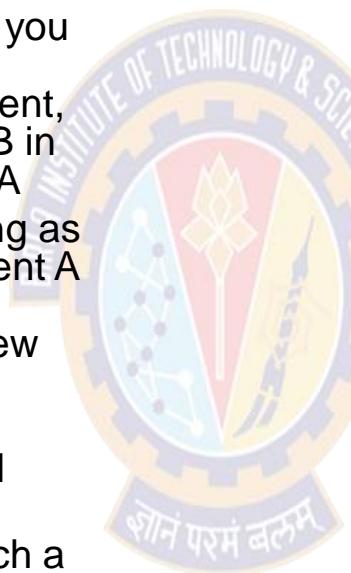


Branching Components

Managing Component

Circular Dependencies

- This occurs when the dependency graph contains cycles
- The simplest example is that you have a component, A, that depends on another component, B, unfortunately component B in turn depends on component A
- to survive this problem so long as there is a version of component A that you can use to build component B then use the new version of B to build the new version of A
- This results in a kind of “build ladder”
- No build system supports such a configuration out of the box, so you have to hack your toolchain to support it.
- also have to be cautious in how the parts of your build interact



**Circular dependency
build ladder**



Thank You!

In our next session:



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps





CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 6

Agenda

Introduction to build

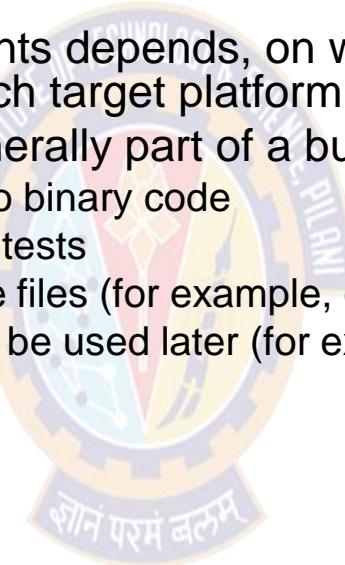
- Build automation and Build tools



Build automation and Build tools

Build tools

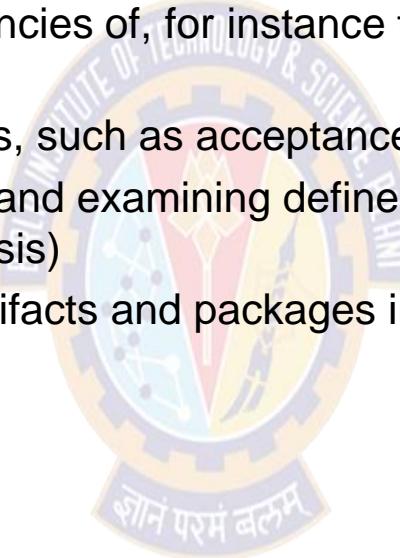
- Build tools automate software builds
- Usually, a software build consists of a number of steps that are dependent on each other
- In detail the sequence of events depends, on which programming language is used and for which target platform software is developed
- The following phases are generally part of a build:
 - Compiling the source code to binary code
 - Running and evaluating unit tests
 - Processing existing resource files (for example, configurations)
 - Generating artifacts that can be used later (for example, WAR or EAR files, Debian packages)



Build automation and Build tools

Build tools Contd..

- Additional steps that are often executed in the a build and that can likewise be automated with the employed build tools are:
 - Administering dependencies of, for instance the libraries that are used in the project
 - Running additional tests, such as acceptance and load tests
 - Analyzing code quality and examining defined conventions in the source code (static code analysis)
 - Archiving generated artifacts and packages in a central repository



Build automation and Build tools

Build tools Contd..

Technology	Tool
Rails	Rake
.Net	MsBuild
Java	Ant, Maven, Buildr, Gradle
C,C++	SCons
• Maver	
• Gradle	



Agenda

Build Tools – Maven and Gradle

- Maven
- Gradle



Build Tool - Maven

History

- Maven, a Yiddish word meaning accumulator of knowledge
- Originally started as an attempt to simplify the build processes in the Jakarta Turbine project
- Concept behind building Maven?
 - There were several projects each with their own Ant build files that were all slightly different and JARs were checked into CVS
 - Need of a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information and a way to share JARs across several projects
- The result is a tool that can now be used for building and managing any Java-based project

Build Tool - Maven

History

- With maven, day-to-day work of Java developers get easier and generally it help with the understanding of any Java-based project
- Maven addresses two aspects of building software
 - describes how software is built
 - describes its dependencies
- Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages
- The Maven project is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project
- And Initial Release of Maven was : 13 July 2004; 14 years ago

Build Tool - Maven

Software

- Maven is a software which collects all your dependency find it from all over internet compile your classes and provide you the final jar/war
- JAR: Java Archive Files
- WAR: Web Application Resource or Web Application Archive Files
- EAR: Enterprise Archive



Build Tool - Maven

Objective

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features



Build Tool - Maven

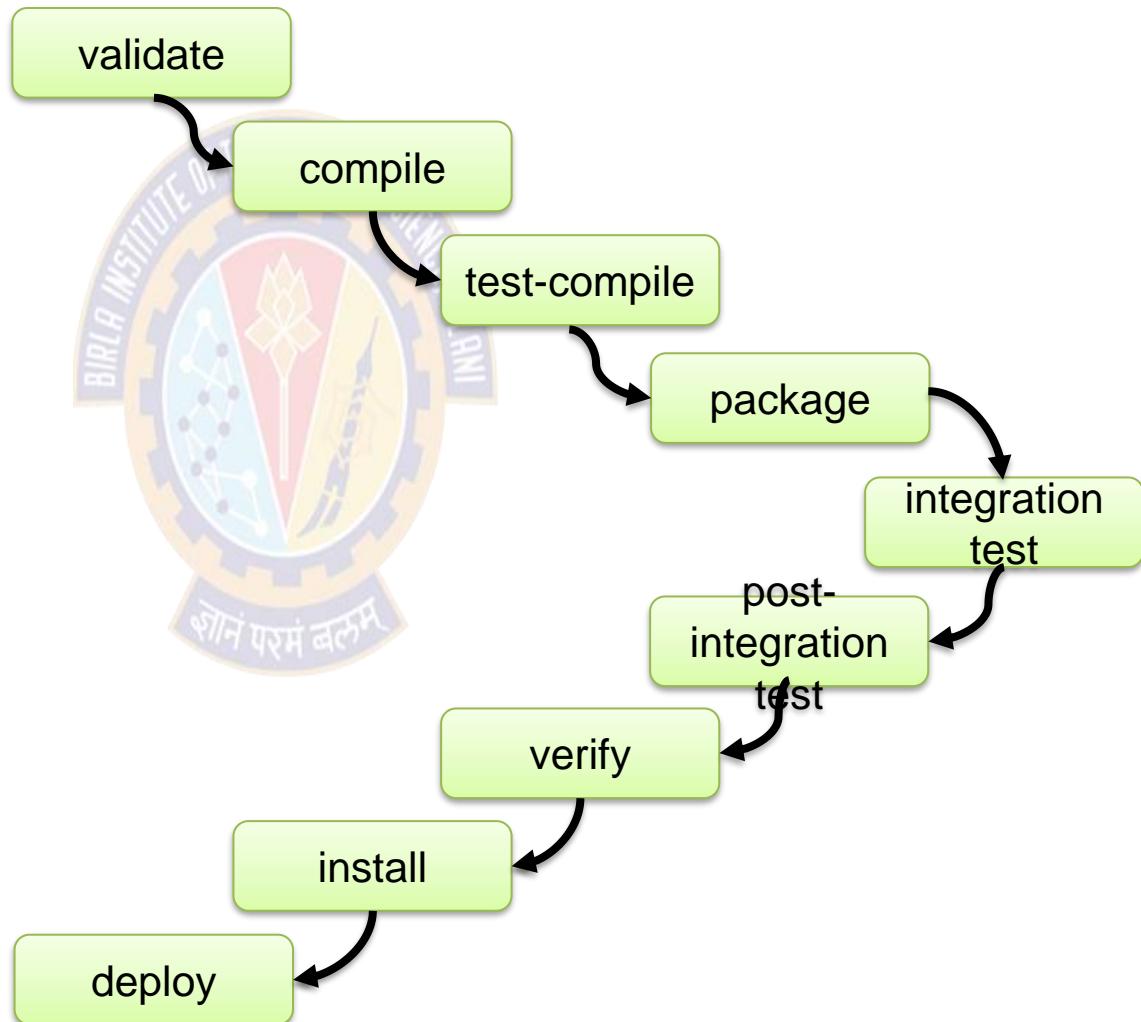
Phases of a standard Maven build

- Currently, Maven is likely the most widely used build tool for Java especially in large enterprises
- Maven default life cycle comprises a number of predefined and fixed phases
 - First, the source files are verified, and the build is initialized
 - Subsequently, the sources and resources are compiled and processed
 - The same happens with test sources and resources
 - Afterwards, the result is packaged, and optionally an integration test is run
 - Finally, the result can be verified, installed, and deployed in an artifact repository
- Not all actions are necessarily executed in each phase
- In this way Maven offers a skeleton along which your own build logic can be oriented

Build Tool - Maven

Phases of a standard Maven build

- Standardized phases, the commands for the compilation and testing of software are the same in each Maven project:
- *mvn package* executes all phases up to packaging and creates a build
- *mvn test* stops at the phase “test.”



Build Tool - Maven

Maven POM

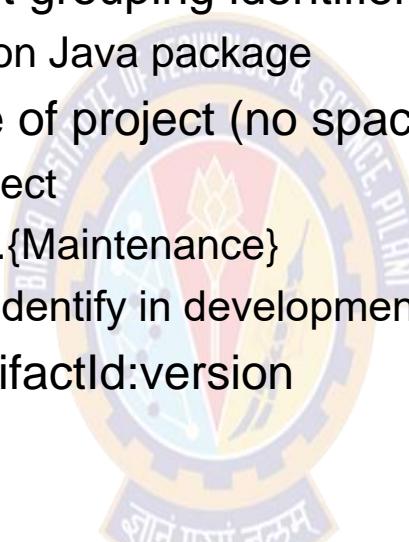
- Stands for Project Object Model
- Describes a project
 - Name and Version
 - Artifact Type
 - Source Code Locations
 - Dependencies
 - Plugins
 - Profiles (Alternate build configurations)
 - Uses XML by Default



Build Tool - Maven

Project Name (GAV)

- Maven uniquely identifies a project using
- groupId: Arbitrary project grouping identifier (no spaces or colons)
 - Usually loosely based on Java package
- artifactId: Arbitrary name of project (no spaces or colons)
 - version: Version of project
 - Format {Major}.{Minor}.{Maintenance}
 - Add '-SNAPSHOT' to identify in development
- GAV Syntax: groupId:artifactId:version



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>Maven_BITS_Training</groupId>
  <artifactId>Maven_BITS_Training</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

Build Tool - Maven

Packaging

- Build type identified using the “packaging” element
- Tells Maven how to build the project
- Example packaging types: – pom, jar, war, ear, etc.
- Default is jar



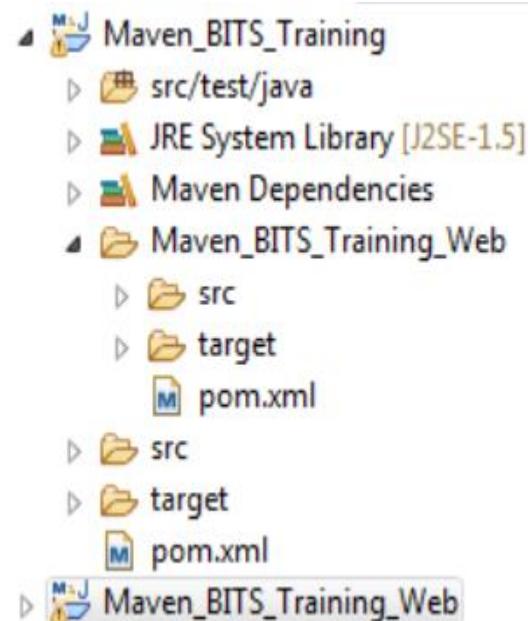
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>Maven_BITS_Training</groupId>
  <artifactId>Maven_BITS_Training</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
```

Build Tool - Maven

Multi Module Projects

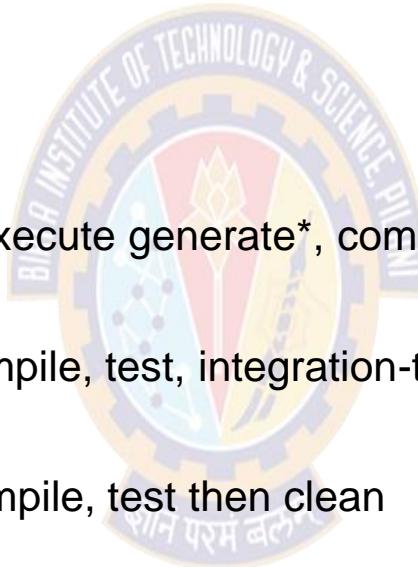
- Maven has 1st class multi-module support
- Each maven project creates 1 primary artifact
- A parent pom is used to group modules



Build Tool - Maven

Example Maven Goals

- *mvn install*
 - Invokes generate* and compile, test, package, integration-test, install
- *mvn clean*
 - Invokes just clean
- *mvn clean compile*
 - Clean old builds and execute generate*, compile
- *mvn compile install*
 - invokes generate*, compile, test, integration-test, package, install
- *mvn test clean*
 - Invokes generate*, compile, test then clean



Build Tool - Gradle

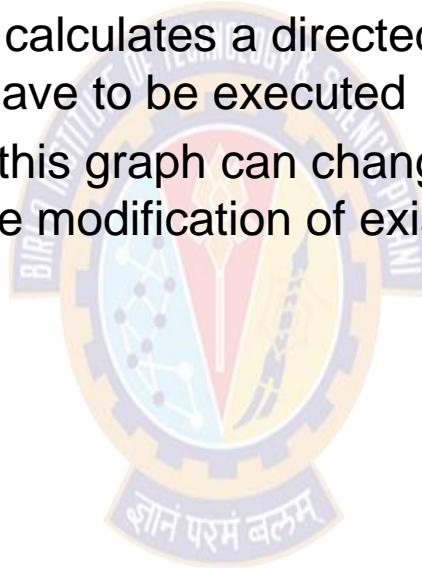
About Gradle

- Gradle is an open-source build automation tool focused on flexibility and performance
- Gradle build scripts are written in Domain Specific Language [DSL], i.e. Groovy
- Groovy resembles Java
- Every Java program is also a valid Groovy program, Groovy offers options for writing programs with substantially easier syntax
- Why Consider Gradle?
 - High performance
 - It is very specific on running only required tasks; which have been changed
 - Build cache helps to reuse tasks outputs from previous run
 - It has ability to have shared build cache within different machine
 - JVM foundation
 - It has JVM Foundation; where using Java Development Kit is Must
 - Benefits to use of standard Java API's in build logic
 - However it is not limited to Java

Build Tool - Gradle

Gradle concepts

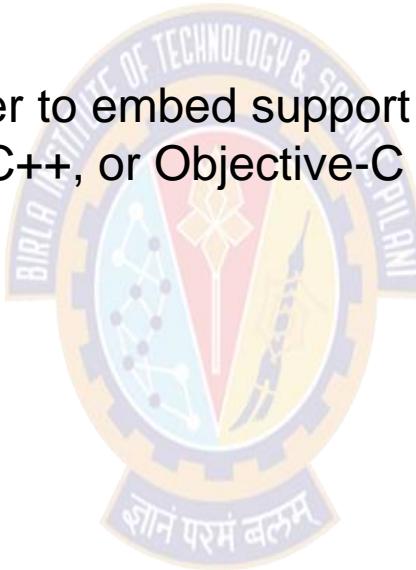
- The core concepts of Gradle are tasks and the dependencies between them
- Based on these, Gradle calculates a directed, acyclic graph to determine which tasks have to be executed in which order
- This is necessary since this graph can change through custom tasks, additional plug-ins, or the modification of existing dependencies



Build Tool - Gradle

Gradle offerings

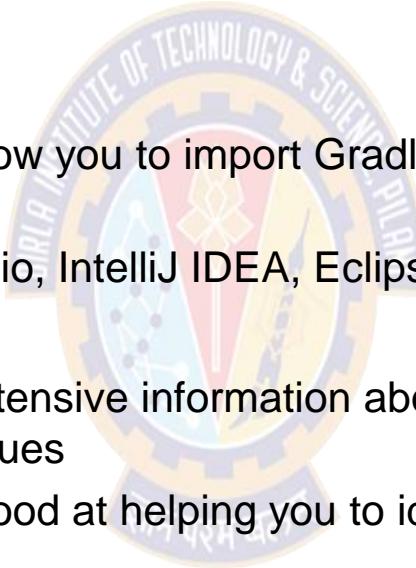
- Gradle also offers the option to extend its functionality by embedding plug-ins
- This allows the developer to embed support of other programming languages like Groovy, C++, or Objective-C in Gradle



Build Tool - Gradle

Gradle benefits

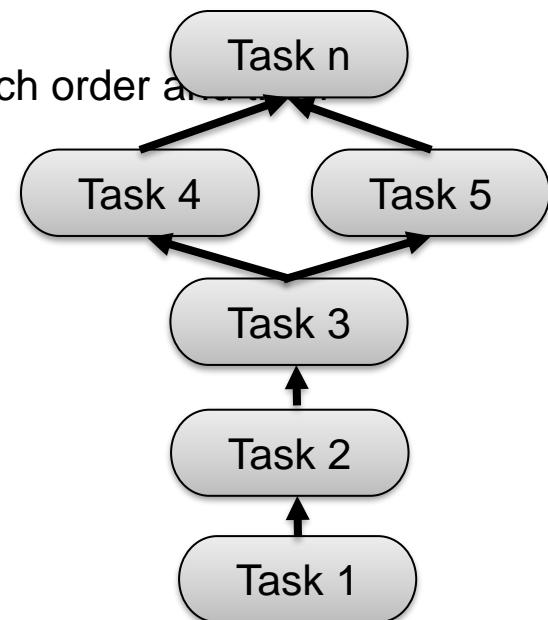
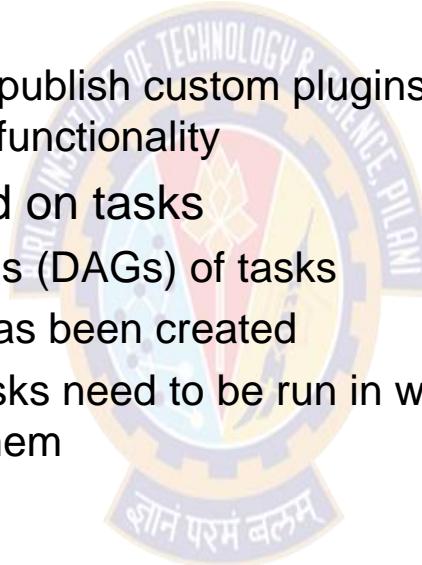
- Extensibility
 - You can readily extend Gradle to provide your own task types or even build model
- IDE Support
 - Several major IDEs allow you to import Gradle builds and interact with them
 - Example: Android Studio, IntelliJ IDEA, Eclipse, and NetBeans
- Insight
 - Build scans provide extensive information about a build run that you can use to identify build issues
 - They are particularly good at helping you to identify problems with a build's performance



Build Tool - Gradle

Gradle DAG

- Gradle is a general-purpose build tool
 - Currently it supports Maven- and Ivy-compatible repositories and the filesystem
 - It supports create and publish custom plugins to encapsulate own conventions and build functionality
- The core model is based on tasks
 - Directed Acyclic Graphs (DAGs) of tasks
 - Once the task graph has been created
 - It determines which tasks need to be run in which order and then proceeds to execute them
 - Example of a Graph
 - Tasks Consists of:
 - Action : To perform something
 - Input : values to action
 - Output : generated by



Build Tool - Gradle

Summary

- Gradle has several fixed build phases
 - It evaluates and executes build scripts in Three Phases
 - Initialization : Set-up environment & Identify which Projects will be part of it
 - Configuration: Construct Task graph and then Identify task to be executed in defined order
 - Execution: Run the task
- Gradle is extensible in more ways than one
 - It allows you to Custom Task Types – Define your own Task Types
 - Custom Task Actions
 - Extra Properties on Projects and Tasks
 - Custom Conventions – User friendly

Build Tool

Gradle vs Maven

Gradle	Maven
Flexibility: <ul style="list-style-type: none">Flexibility on ConventionsUser Friendly & Customized	Flexibility: <ul style="list-style-type: none">No flexibility on ConventionsIt is rigid
Performance: <ul style="list-style-type: none">It process only the files has been changedReusability by working with Build CacheShipping is faster	Performance: <ul style="list-style-type: none">It process the complete buildNo Build Cache conceptShipping is slow as compared to Gradle
User Experience: <ul style="list-style-type: none">IDE Support : Is in evolving StageCLI : Modern CLI	User Experience: <ul style="list-style-type: none">IDE Support : Is matureCLI solution is classic in comparison with Gradle

Agenda

Selenium - Test Automation

- Introduction
- Test login page – example
- Selenium Components
- Selenium History
- Selenium Supports
- Selenium Architecture
- Selenium Benefits



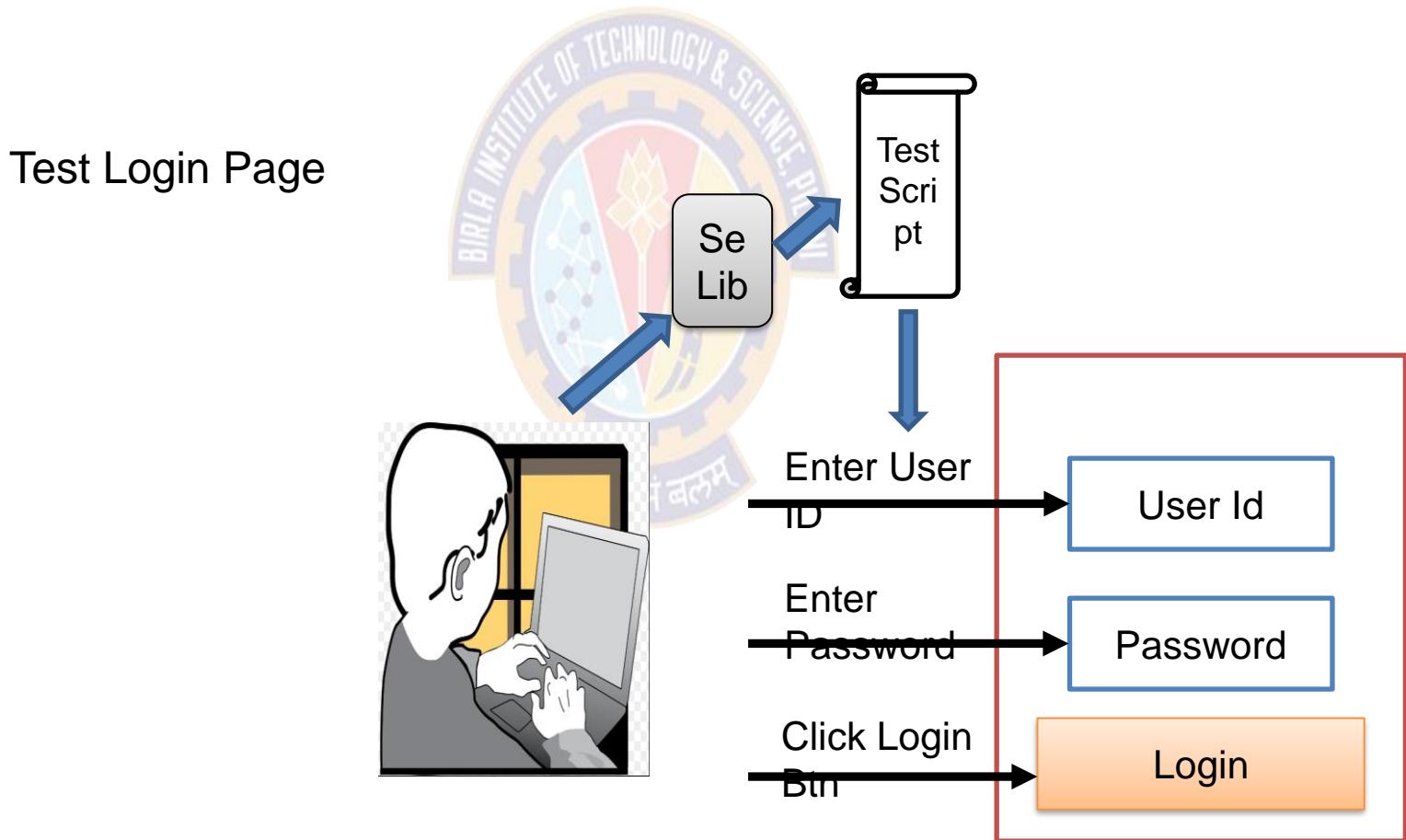
Selenium - Test Automation

Introduction

- Selenium helps to automate web browser interaction
- Scripts perform the same interactions that any user can perform manually
- Selenium can perform any sort of automated interaction; however it was originally designed for automated web application testing
- Common benefits of Test Automation:
 - Frequent regression testing
 - Rapid feedback to developers
 - Virtually unlimited iterations of test case execution
 - Support for Agile and extreme development methodologies
 - Disciplined documentation of test cases
 - Customized defect reporting
 - Finding defects missed by manual testing

Selenium - Test Automation

Example

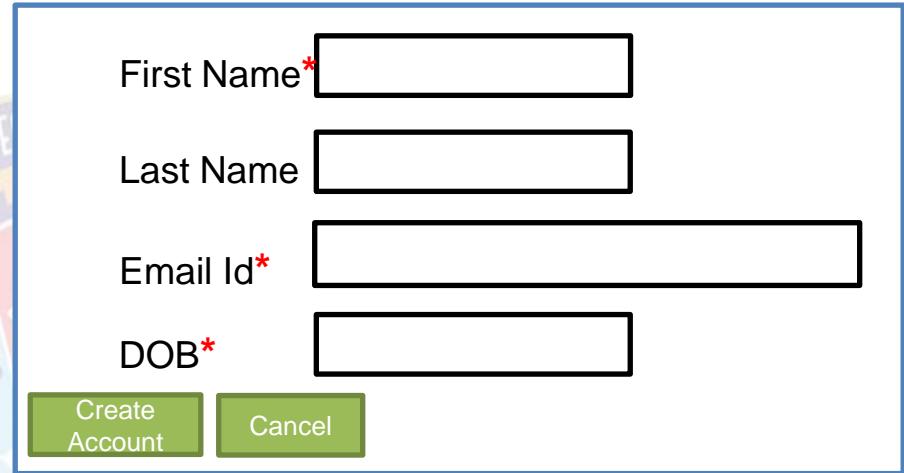
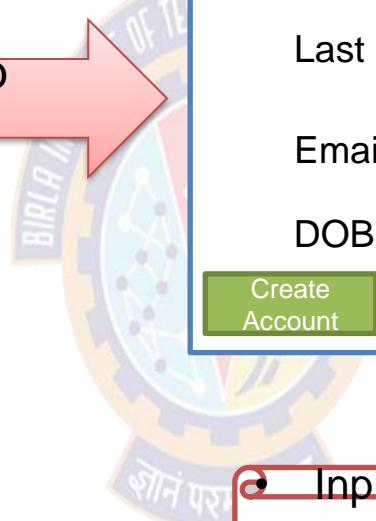


Selenium - Test Automation

Working with Selenium

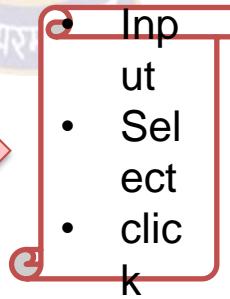
- At a high level you will be doing three things with Selenium

1. Recognize web elements



A screenshot of a web form for account creation. The form includes fields for First Name, Last Name, Email Id, and DOB, each with a red asterisk indicating it is required. Below the fields are two green buttons: "Create Account" and "Cancel".

2. Add Actions (Script using preferred lang)



Test Data

3. Run the Test

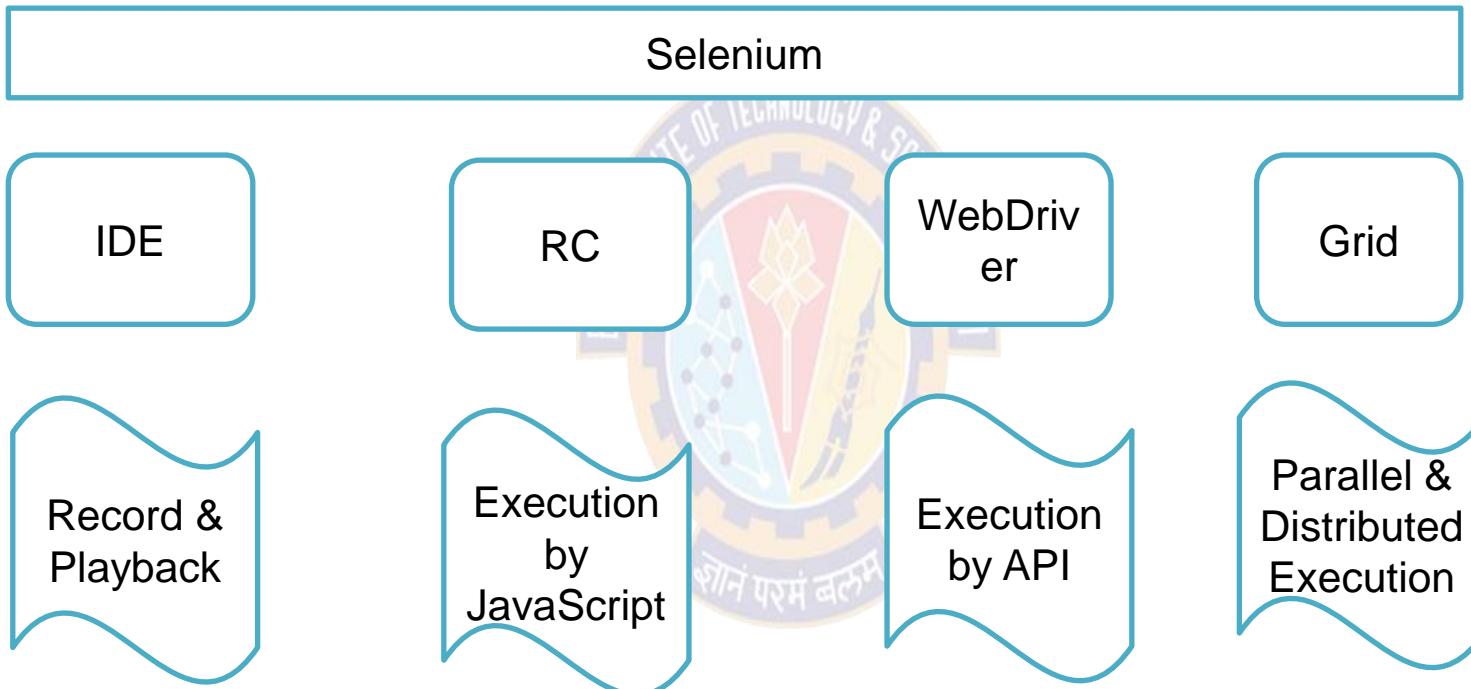
Selenium - Test Automation

Selenium Components

- Selenium IDE
 - A Record and playback plugin for Firefox add-on
 - Prototype testing
- Selenium RC (Remote Control)
 - Also known as selenium 1
 - Used to execute scripts (written in any language) using Javascript
 - Now Selenium 1 is deprecated and is not actively supported
- WebDriver
 - Most actively used component today
 - An API used to interact directly with the web browser
 - Is a successor to Selenium 1 / Selenium RC
 - Selenium RC and WebDriver are merged to form Selenium 2
- Selenium Grid
 - A tool to run tests in parallel across different machines and different browser simultaneously
 - Used to minimize the execution time

Selenium - Test Automation

Selenium Components Contd..



Selenium - Test Automation

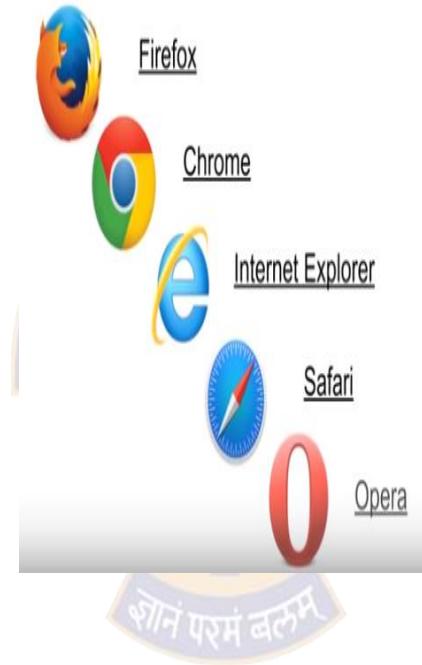
Selenium History

- Selenium is a set of different software tools each with a different approach to support test automation
- Selenium support executing one's tests on multiple browser platforms
- Selenium first came to life in 2004 when Jason Huggins was testing an internal application at ThoughtWorks
- It started with a Javascript library
- In 2006 at Google, Simon Stewart started work on a project WebDriver
- The merging of Selenium and WebDriver provided a common set of features for all users and brought some of the brightest minds in test automation under one roof in 2009

Selenium - Test Automation

Selenium - Supports

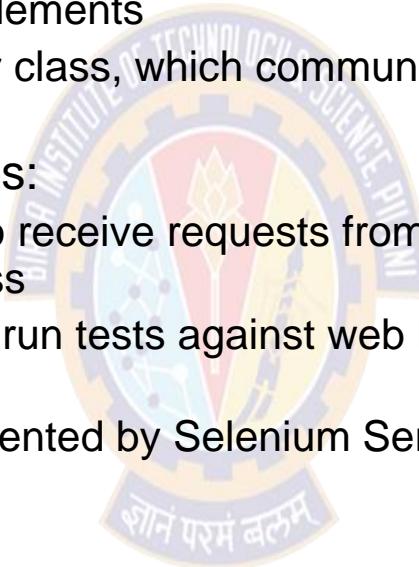
- Browsers
- OS
- Language



Selenium - Test Automation

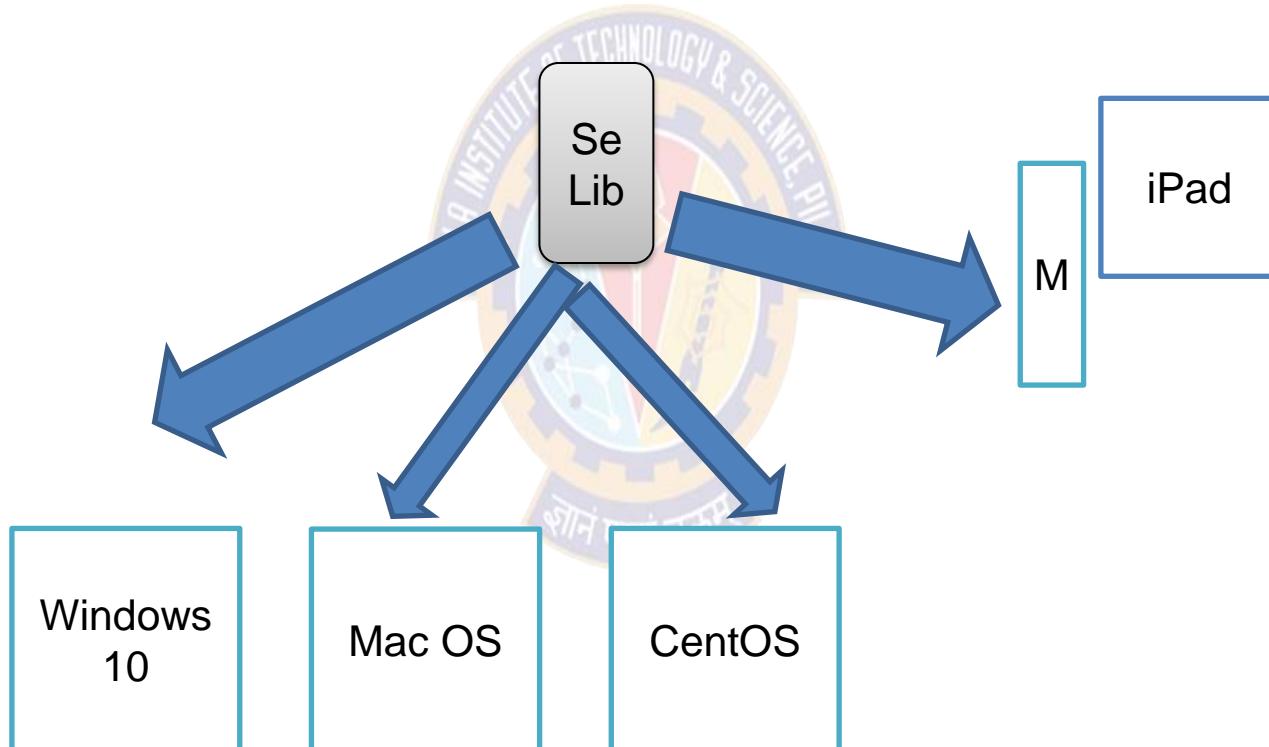
Selenium client-server architecture

- Selenium Client includes:
 - The WebDriver API, which is used to develop test scripts to interact with page and application elements
 - The RemoteWebDriver class, which communicates with a remote Selenium server
- Selenium Server includes:
 - A server component, to receive requests from Selenium Client 's RemoteWebDriver class
 - The WebDriver API, to run tests against web browsers on a server machine
 - Selenium Grid, implemented by Selenium Server in command-line options for grid features



Selenium - Test Automation

Selenium Grid



Selenium - Test Automation

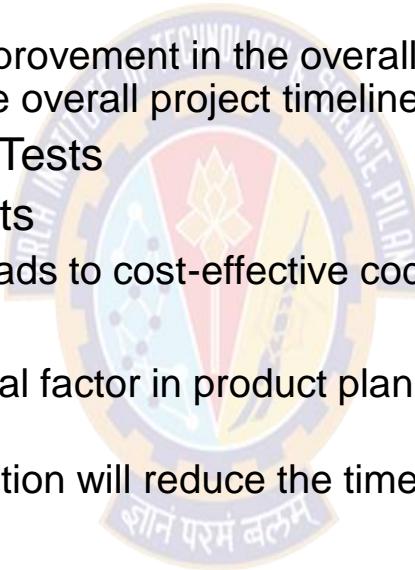
Benefits

- Faster Feedback
 - It enhances communication between product owners, developers and designers
 - Quick fix of malfunctions & bugs in real-time
- Accelerated Results
 - Testing can be executed on a repetitive basis, which in turn will deliver speedier results every time with minimized time and reduced effort
- Reduced Business Expenses
 - Time needed to execute tests will decrease dramatically
 - Larger quantity of work can be undertaken in the same amount of time with greatly increased levels of accuracy
 - This will directly reduce project costs

Selenium - Test Automation

Benefits Contd..

- Testing Efficiency Improvement
 - Testing occupies a considerable part of the whole Software Development Lifecycle
 - So even the smallest improvement in the overall efficiency can result in a massive difference to the overall project timeline
- Reusability of Automated Tests
- Earlier Detection of Defects
 - Early defect detection leads to cost-effective code rework
- Faster Time-to-Market
 - Time to market is a critical factor in product planning and success with regard to software applications
 - Efficiency of test automation will reduce the time of project; which results you to hit market early



References

- <https://www.seleniumhq.org/docs/index.jsp/>



Continuous Code Inspection

Code quality

- Continuous code inspection = Constantly scanning code
- Continuous code inspection is to Identify if any defects
- It is a process of code review; however the goal is to prevent defects
- Its been proved 90% of defects can be addressed using code inspections tools
- Code Inspection & Testing?
 - Testing verifies functionality and improves software quality
 - Testing is expensive if you have to go through it over and over again
 - By minimizing defects during code inspections, you can make your testing efforts more efficient
 - If your code is complex, it may not be fully testable
 - A code inspection, however, can find defects at the code level

Note: Even with automated testing, it takes time to verify functionality; by resolving defects at the code level, you'll be able to test functionality faster

Continuous Code Inspection

Code Inspection Measures

- Code inspections must be well-defined as per requirements:
 - Functional requirements : User Needs : Cosmetic
 - Structural requirements : System Needs : Re-engineering
- Run Time Defects:
 - Identify run time errors before program run
 - Examples: Initialization (using the value of unset data), Arithmetic Operations (operations on signed data resulting in overflow) & Array and pointers (array out of bounds, dereferencing NULL pointers), etc.,
- Preventative Practices:
 - This help you avoid error-prone or confusing code
 - Example: Declarations (function default arguments, access protection), Code Structure (analysis of switch statements) & Safe Typing (warnings on type casting, assignments, operations), etc.,
- Style:
 - In-house coding standards are often just style, layout, or naming rules and guidelines
 - Instead using a proven coding standard is better for improving quality

Continuous Code Inspection

How to Improve Your Code Inspection Process

- Involve Stakeholders
 - Developer, Management & Customer
- Collaborate
 - Collaboration — both in coding and in code inspections
- Recognize Exceptions
 - Sometimes there are exceptions to the rule
 - In an ideal world, code is 100% compliant to every rule in a coding standard
 - The reality is different
- Document Traceability
 - Traceability is important for audits
 - Capture the history of software quality
- What to Look For in Code Inspection Tools
- Automated inspection
- Collaboration system

Agenda

SonarQube - Continuous Code Inspection Tool

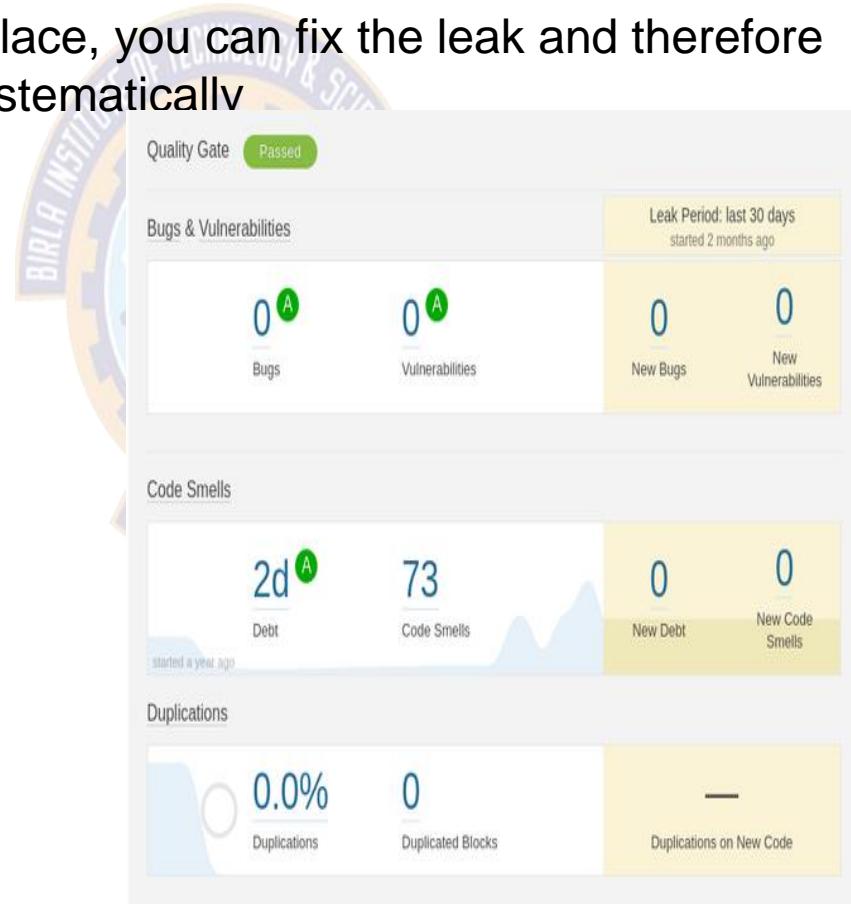
- SonarQube Quality Gate
- Continuous Inspection
- Focus on the Leak
- Enforce Quality Gate
- Benefits of SonarQube



SonarQube - Continuous Code Inspection Tool

SonarQube Quality Gate

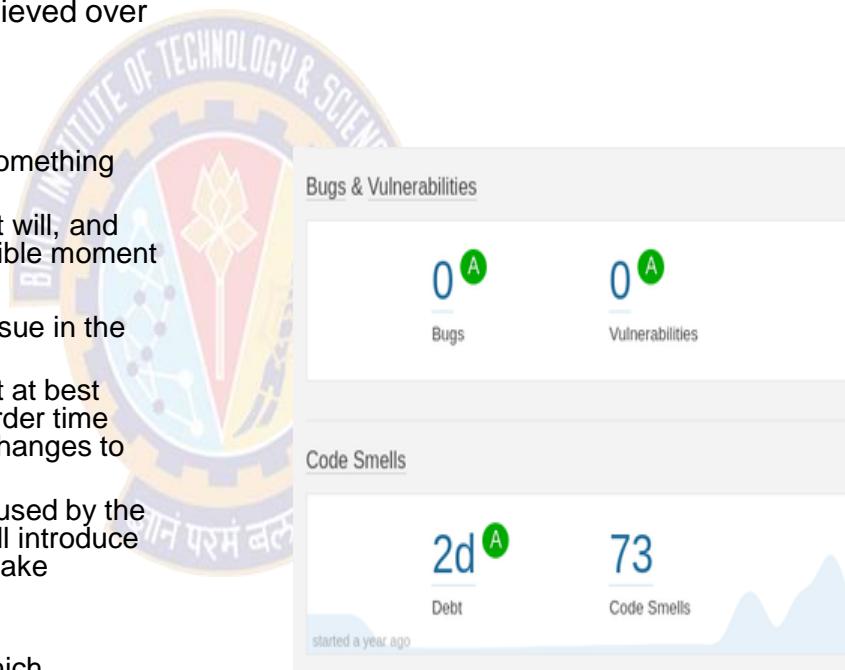
- SonarQube provides the capability to not only show health of an application but also to highlight issues newly introduced
- With a Quality Gate in place, you can fix the leak and therefore improve code quality systematically



SonarQube - Continuous Code Inspection Tool

Continuous Inspection

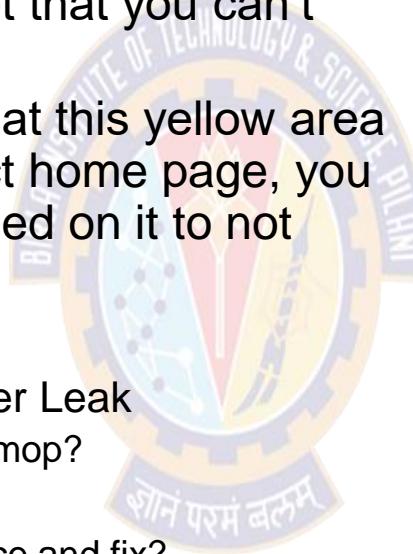
- Overall health
 - Tools shows overall health of the application
 - Also it shows good results achieved over time
- Terminology:
 - Bug:
 - An issue that represents something wrong in the code
 - If this has not broken yet, it will, and probably at the worst possible moment
 - Code Smell:
 - A maintainability-related issue in the code
 - Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code
 - At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes
 - Vulnerability:
 - A security-related issue which represents a backdoor for attackers



SonarQube - Continuous Code Inspection Tool

Focus on the Leak

- Once that Leak is under control, code quality will start improving systematically
- Leak is a built-in concept that you can't miss
- Once you've had a look at this yellow area on the left of your project home page, you will always remain focused on it to not miss any new issues
- Water Leak Paradigm:
 - Situation: Kitchen Water Leak
 - Do you reach out to mop?
 - OR
 - Do you find the source and fix?

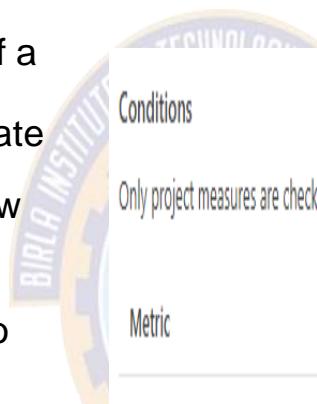


Note: Same logic should be used for Code

SonarQube - Continuous Code Inspection Tool

Enforce Quality Gate

- To fully enforce a code quality practice across all teams, you need to set up a Quality Gate
- A set of requirements that tells whether or not a new version of a project can go into production
- SonarQube's default Quality Gate checks what happened on the Leak period and fails if your new code got worse in this period
- A quality gate is the best way to enforce a quality policy in any organization
- Define a set of Boolean conditions based on measure thresholds against which projects are measured
- It supports multiple quality gate definitions



Conditions

Only project measures are checked against thresholds. Sub-projects, directories and files are ignored.

Metric	Over Leak Period	Operator	Warning	Error
Coverage on New Code	Always	is less than	80.0%	
Duplicated Lines on New Code (%)	Always	is greater than	3.0%	

SonarQube - Continuous Code Inspection Tool

Analyze pull requests

- To shorten the feedback loop, you can set up the analysis of your pull requests
- Analyses will be run on your feature branches without being pushed to SonarQube
- This gives you the opportunity to fix issues before they ever reach SonarQube



Benefits of SonarQube

Branch Analysis

- Track the quality of short-lived and long-lived code branches in SonarQube to ensure that only clean, approved code gets merged into master



References

- <https://docs.sonarqube.org>





Thank You!

In our next session:



BITS Pilani
Pilani Campus

BITS Pilani presentation

Yogesh B
Introduction to DevOps





CSIZG514/SEZG514, Introduction to DevOps

Lecture No. 7 and 8

Agenda

Using Continuous Integration Software

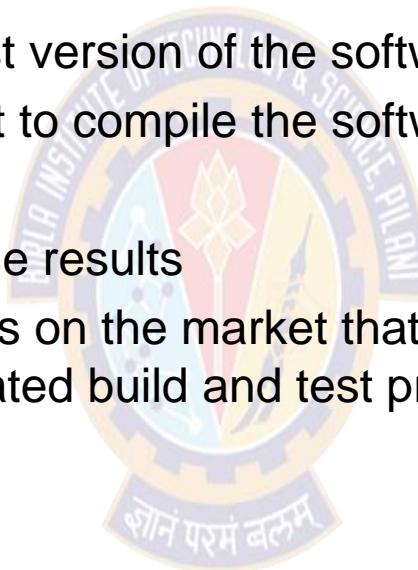
- Basic Functionality of Continuous Integration Software
- Basic Operations
- How it was before Continuous Integration
- Prevention to be adhere



Continuous Integration Software

Basic functionality of continuous integration software

- Poll version control system
- Check if any commits have occurred
- If so, check out the latest version of the software
- Trigger / Run build script to compile the software
- Run the tests
- And then notify you of the results
- There are many products on the market that can provide the infrastructure for automated build and test process



Continuous Integration Software

Basic Operation

- Continuous Integration server software has two components
 - The first is a long-running process which can execute a simple workflow at regular intervals
 - The usual CI workflow polls revision / version control system at regular intervals
 - If it detects any change, it will check out a copy of project to a directory on the server
 - It will then execute the commands (these commands build an application and run the relevant automated tests)
 - The second provides a view of the results of the processes that have been run, notifies for the success or failure of build and test runs, and provides access to test reports, installers, and so on
 - Most CI servers include a web server that shows you a list of builds that have run
 - Allows look at the reports that define the success or failure of each build

Continuous Integration Software

Bells and Whistles

- Apart from basic operations we can get the status of the most recent build sent to an external device
- Organizations use red and green lava lamps to show the status of the last build
- Even few organization go with all in one health board for all the tools used; where you can see health of each build, git commits and configuration management results
- Some Advanced experiments:
 - Flashing lights and sirens for build progress
 - Text-to-speech to read out the name of the person who broke the build
 - Display the status of the build, along with the avatars of the people who checked in

Note: These are just a great way to allow everyone to see the status of the build at a glance and Visibility is one of the most important benefits of using a CI server

How it was before Continuous Integration

Nightly Build

- It was a common practice for many years
- If I break it; I will monitor until next break by someone
- The idea is that a batch process will compile and integrate the codebase every night when everybody goes home
- If it is failed again during the night run, can do changes to fix it however need to wait for next night run
- So verification of System Integration will be on hold until next run

Note: this strategy will not be a good Idea when you have a geographically dispersed team working on a common codebase from different time zones

Prevention to be adhere

Don't Check In on a Broken Build

- Per process if a build breaks, developer team need to fix it ASAP
- If any check-in breaks the build then try to fix it before moving ahead
- At the Broken Stage; nobody should be allowed to check-in
- What if we do Check In on Broken Build:
 - If any new check-in or build trigger during broken state will take much longer time to fix
 - Frequent broken build will encourage team not to care much about working condition

Note: The CI System we are building is to make sure green build and working software

Prevention to be adhere

Commit locally than direct to Production

- Per process a commit triggers the creation of a release candidate
- Try to keep Check-in lightweight
- Generally regular check-in with 20 min interval is a achievable state
- To achieve this we need some commitment from Development Team
 - Commit locally
 - Always keep refreshed local copy
 - If feasible run local build and commit tests
 - If successful; then push to Production
- Why Check-in locally and Commit test run?
 - If there are check-in's before our last update; high chance of conflict :: So if we check out and run tests locally, we have chance to identify problem without breaking the build
 - If new artifacts inclusion has been missed in the Check In, it will result to common errors :: running locally will help to Identify missing configuration

Prevention to be adhere

Wait for Commit Tests to Pass before Moving On

- The CI System is a Shared resource
- At time of Check-in, we should monitor the build progress – Its Developers default duty
- Until the check-in is compiled and passed commit tests, Do not start any new task
- Its not good to say however “Developer should not even leave out for Lunch or Meeting”
- If the commit succeeds the developers are then and only then free to move
- If it fails “Either fix it or Revert it to previous working version”
- Never go home with broken build

Prevention to be adhere

Wait for Commit Tests to Pass before Moving On Contd..

**Geo – Distributed
Teams Working on
CI**



**Hardly it will be 3 hrs
offline time to fix
build**

Prevention to be adhere

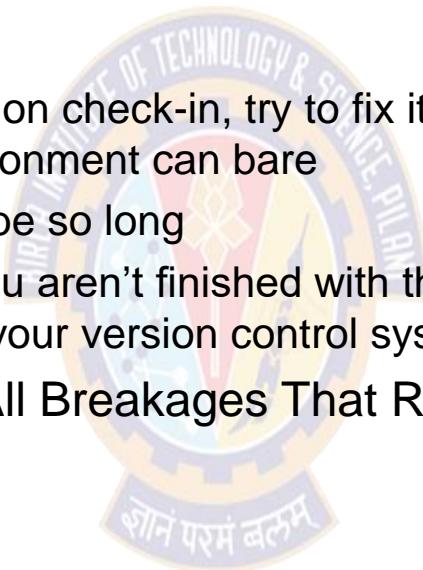
Always Be Prepared to Revert to the Previous Revision

- In a scenario of failed commit stage, it is important that you get everything working again quickly
- If you can't fix the problem quickly, for whatever reason, you should revert to the previous change-set held in revision control
- And remedy the problem in our local environment
- Example:
 - Airplane pilots are taught that every time they land, they should assume that something will go wrong
 - So they should be ready to abort the landing attempt and “go around” to make another try
 - Use the same mindset when checking in
 - Assume that it may break something that will take more than a few minutes, and know what to do to revert the changes and get back to the known-good revision in version control

Prevention to be adhere

Always Be Prepared to Revert to the Previous Revision Contd..

- If you try to revert every time then; how can you make progress?
- Time Boxing
 - Establish a team rule
 - When the build breaks on check-in, try to fix it for ten minutes or any approximate time environment can bare
 - However it should not be so long
 - If, after ten minutes, you aren't finished with the solution, revert to the previous version from your version control system
- Take Responsibility for All Breakages That Result from committed Changes



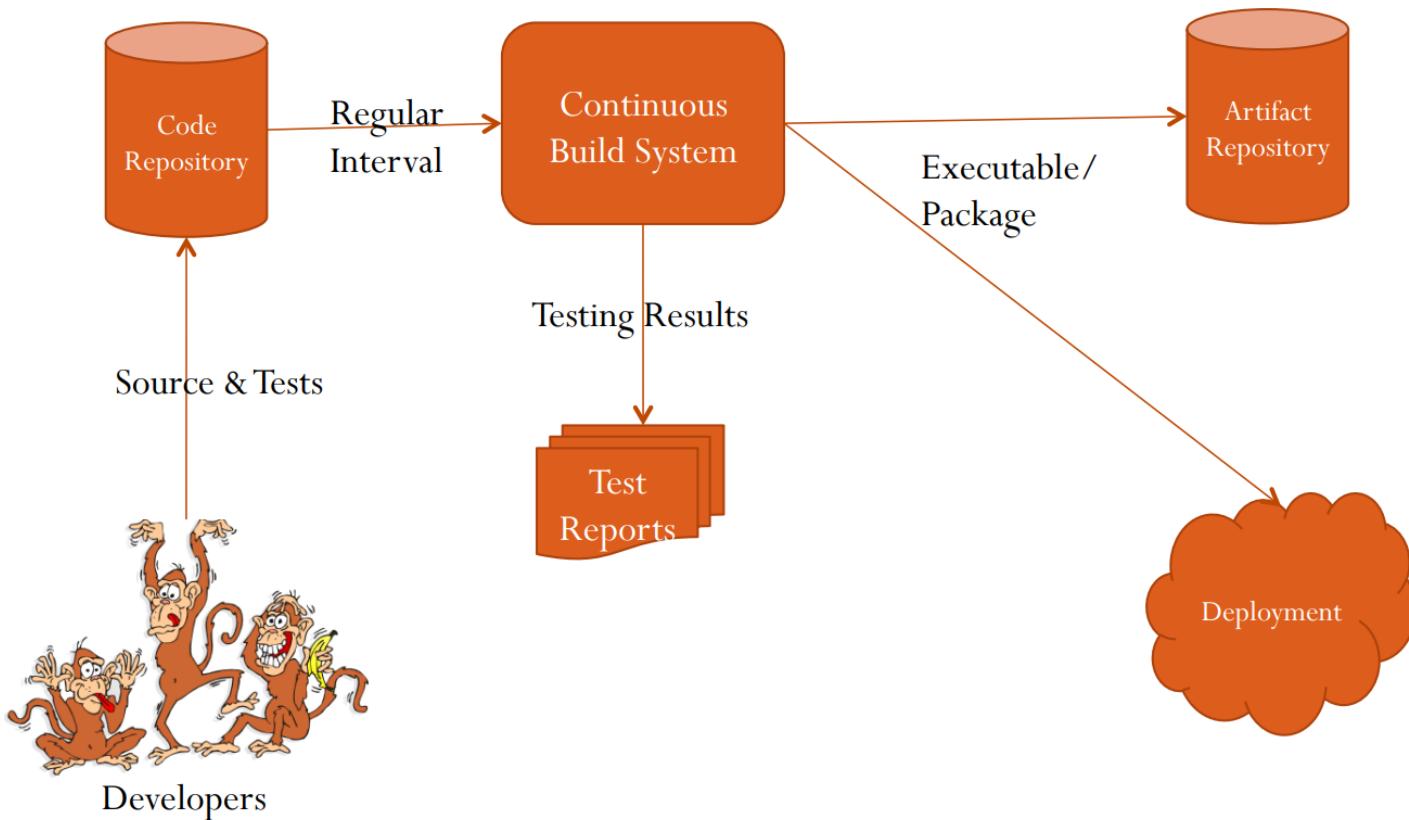
“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible”

~Martin Fowler

What does CI Mean

- At a regular frequency (ideally at every commit), the system is:
- **Integrated:** All changes up until that point are combined into the project
- **Built:** The code is compiled into an executable or package
- **Tested:** Automated test suites are run
- **Archived:** Versioned and stored so it can be distributed as is, if desired
- **Deployed:** Loaded onto a system where the developers can interact with it

CI workflow



Benefits of using CI and CD

- Reduces overhead across the development and deployment process
- Reduces the time and effort for integrations of different code changes
- Enables a quick feedback mechanism on every change
- Allows earlier detection and prevention of defects
- Helps collaboration between team members so recent code is always shared
- Reduces manual testing effort
- Building features more incrementally saves time on the debugging side so you can focus on adding features
- First step into fully automating the whole release process
- Prevents divergence in different branches as they are integrated regularly
- If you have a long running feature you're working on, you can continuously integrate but hold back the release with **feature flags**

Best practice workflow

1. Maintain a code repository
2. Automate your build
3. Make your build self-testing
4. Daily commits to the baseline by everyone on the team
5. Every commit (to the baseline) should be built
6. Keep your builds fast
7. Clone the production environment and test there
8. Make it easy to get the latest deliverables
9. Everyone on the team can see the results of your latest build
10. Automate build deployment

CD checklist

1. Before submitting changes, check to see if a build is currently in the "Successful" status. If not, you should assist in fixing a build before submitting new code.
2. If the status is currently "Successful", you should rebase your personal workspace to this configuration.
3. Build and test locally to ensure the update doesn't break functionality.
4. If Successful, check in new code.
5. Allow CI to complete with new changes.
6. If build **fails**, **stop** and **fix** on your machine. Return to step 3.
7. If build **passes**, **continue** to work on the next item.

Areas of CD Process

- Source Control
- Build Process
- Testing & Q&A
- Deployment
- Visibility

CD Maturity Matrix

	Novice	Beginner	Intermediary	Advanced	Expert
Build	Verification before commit run in developer's Workspace Common nightly build	CI server builds on commit Artifacts are managed	No build scripts -only configurations Dependencies are managed	Distributed builds Staged build sequence	Build from VM CI server orchestrate VMs
Test + QA	Unit Test Code Coverage	Metrics on technical debt & compliance Mock-up's & proxies	Peer-reviews Automated Functional Test	Test Data Test in target	Automated Acceptance Test
SCM	"Early Branching" Branches used for releases Merges are rare	"Late branching" Branches used for work isolation Merges are common	Pre-tested Commits Integration branch is pristine	All commits are tied to tasks Individual history rewrites in DVCS	Release notes & traceability analysis are generated automatically
Visibility	Build status is notified to committer	Latest build status is available to all stakeholders	Trend reports Build status can be subscribed to (pull vs push)	Monitors in work areas show real-time status	Build reports and statistics are shared with customer and public

Principle #1

Every build is a potential release



Principle #3

Automate wherever possible



Principle #2

Eliminate manual bottlenecks

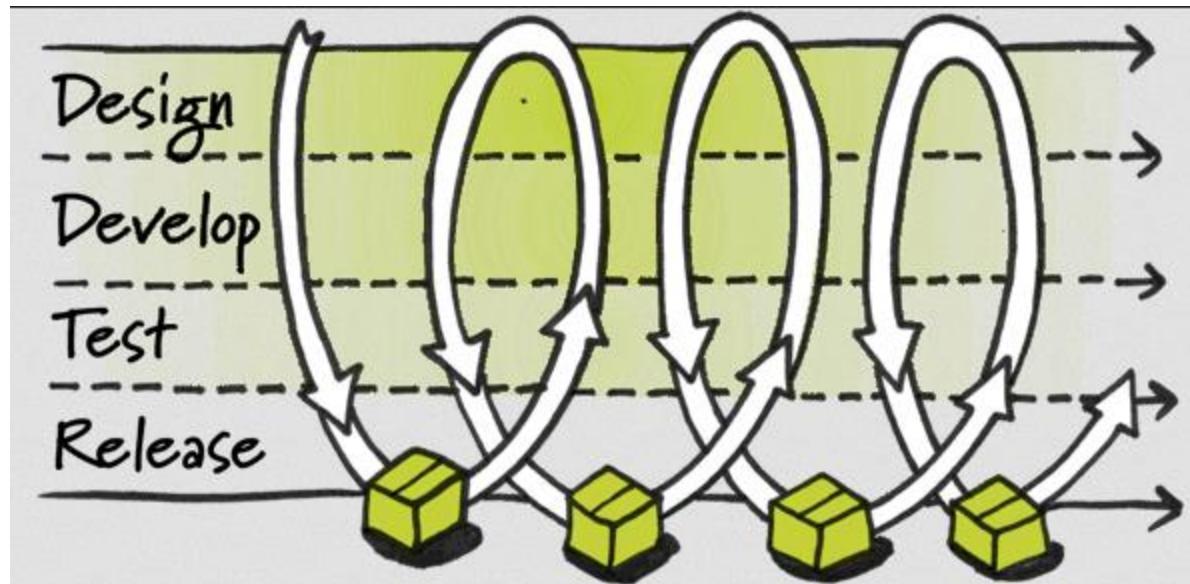
Principle #4

Have automated tests you can trust



Continuous Delivery

- Recall: Build software that is always ready to be deployed into production



Continuous Delivery cont.

- Is this a new idea?
- 1st principle of the Agile Manifesto:
 - “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”
- Goal: Release to production more often
 - Monthly
 - Weekly
 - Daily

Continuous Delivery cont.

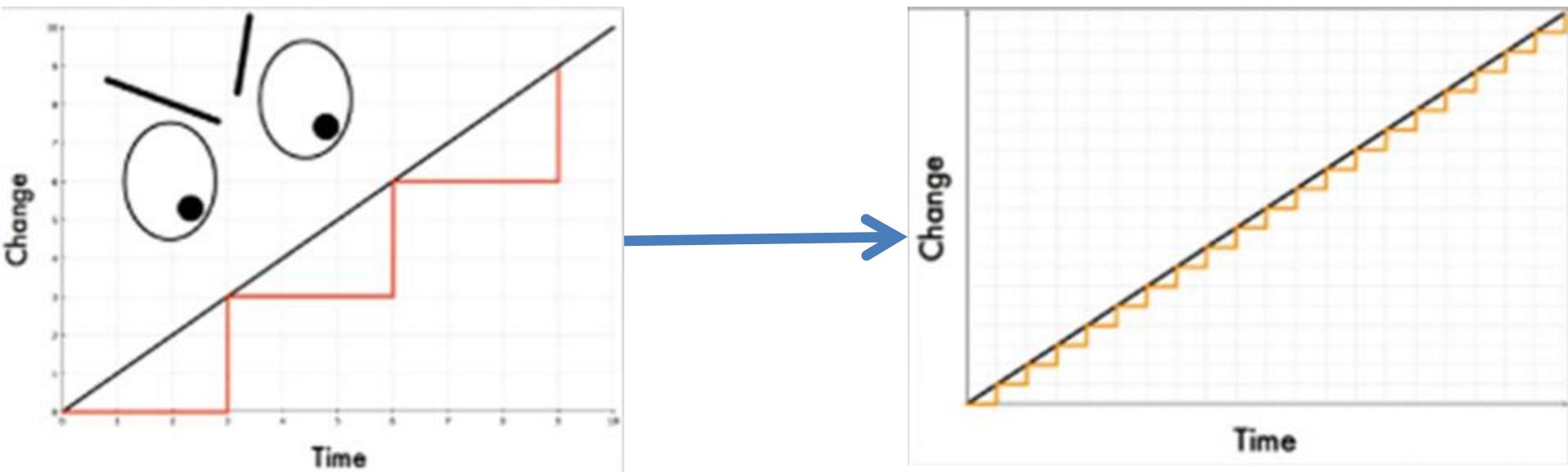
- Pros: Receive many benefits from the Agile process as well as others
 - Build the right product
 - Constant feedback from customers
 - Write thorough acceptance tests
 - Earlier benefits
 - Get product out before competitors
 - Find bugs earlier

Continuous Delivery cont.

- Ability to react quickly to change
 - Not a huge amount of time and money can be abandoned to adopt new requirements
- Save money and time
 - Save money and time if project goes bad
 - Save money and time with automation
 - As deploying to production becomes easier, more time can be spent developing valuable features

Continuous Delivery cont.

- Pros cont:
 - Reliability



Practice Continuous Delivery

- Basics to start practicing Continuous Delivery
 - Configuration Management
 - What needs to be kept under CM?
 - Everything!
 - Code
 - Tests
 - Configuration Files
 - Build Scripts
 - Environments
 - Documentation
 - Etc.

CM Branching

- Stay away from branching except in special cases
 - Branch on releases
 - Horror story example
- You must always check into the trunk!
 - Otherwise you're not continuously integrating

CM Branching

- What if your project team is releasing to production every week, but you're working on a feature that will take longer than the release cycle to implement?
- First option:
 - Gradually release feature into production
- Second option:
 - Feature toggle

Managing Environments

- Must have multiple environments when developing software
- Need to be able to duplicate environments with ease
- Environments configurations to take consideration of:
 - Operating systems including their framework and settings
 - Packages needed to be installed for the application to function properly
 - Network settings

Managing Environments

- Tools to help configure environments in an automated fashion:



Continuous Integration

- Vital step when practicing CD
- Recall: To practice CI, every developer on a project must integrate their work daily with every other developer
 - Everyone on the team needs to practice this for it to work
- Continuous integration is not a tool, but a technique
 - But there are many open source tools to help practice CI

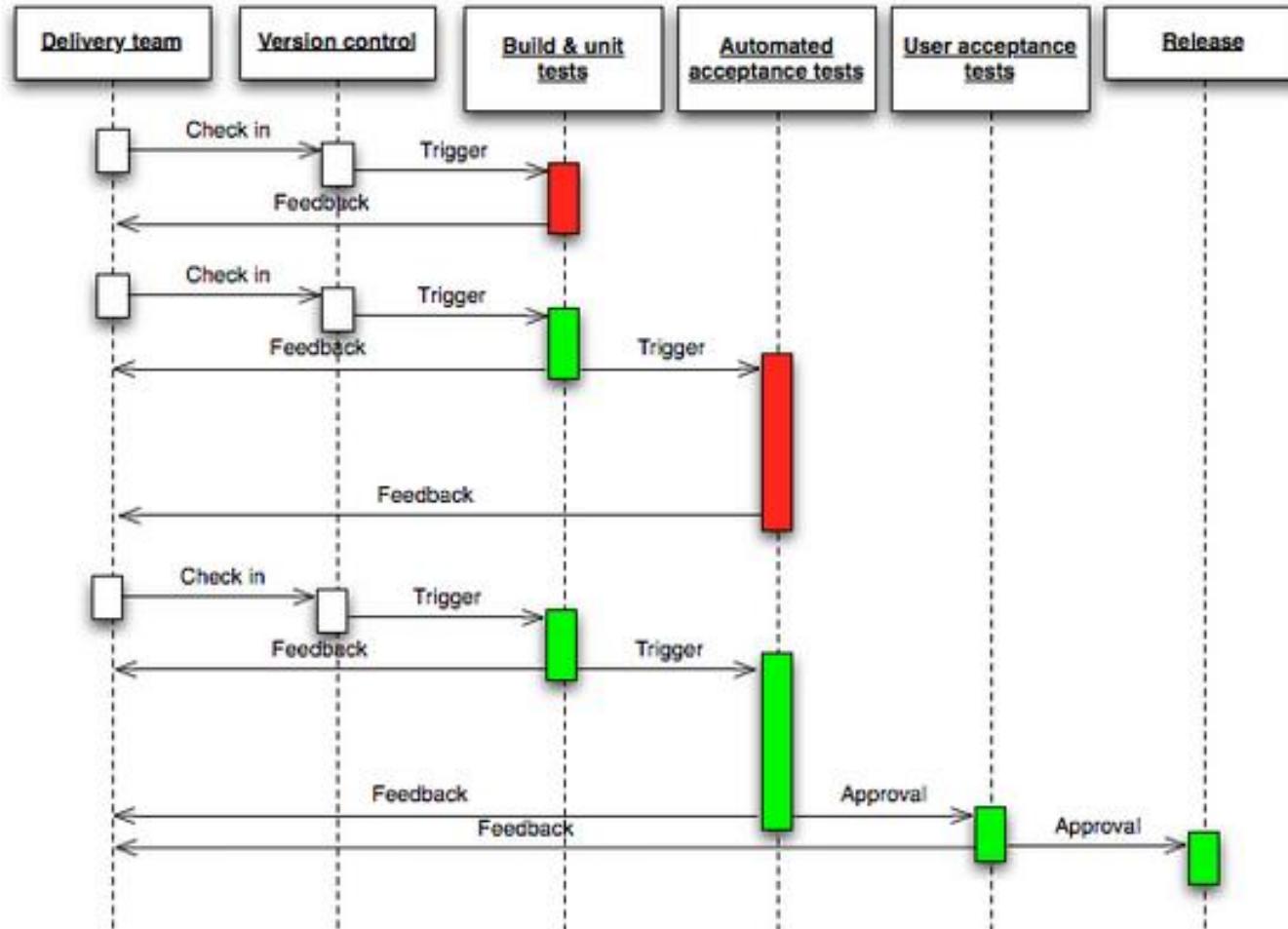
Continuous Integration

- Things a CI server can do:
 - After you check-in code it can:
 - Run build/test scripts
 - Notify developers of a failed build or failed tests
 - Can show all past check-ins and if they failed or passed all the tests
 - Can keep track of multiple projects
 - Plus much more!!

Deployment Pipeline

- Heart of Continuous Delivery
- Will give immediate feedback for how ready your software is for production
- Automate everything(unit tests, acceptance tests, performance tests,...) to the point of just being able to click a button to deploy to production

Deployment Pipeline cont.



Deployment Pipeline in Go

Pipeline Activity

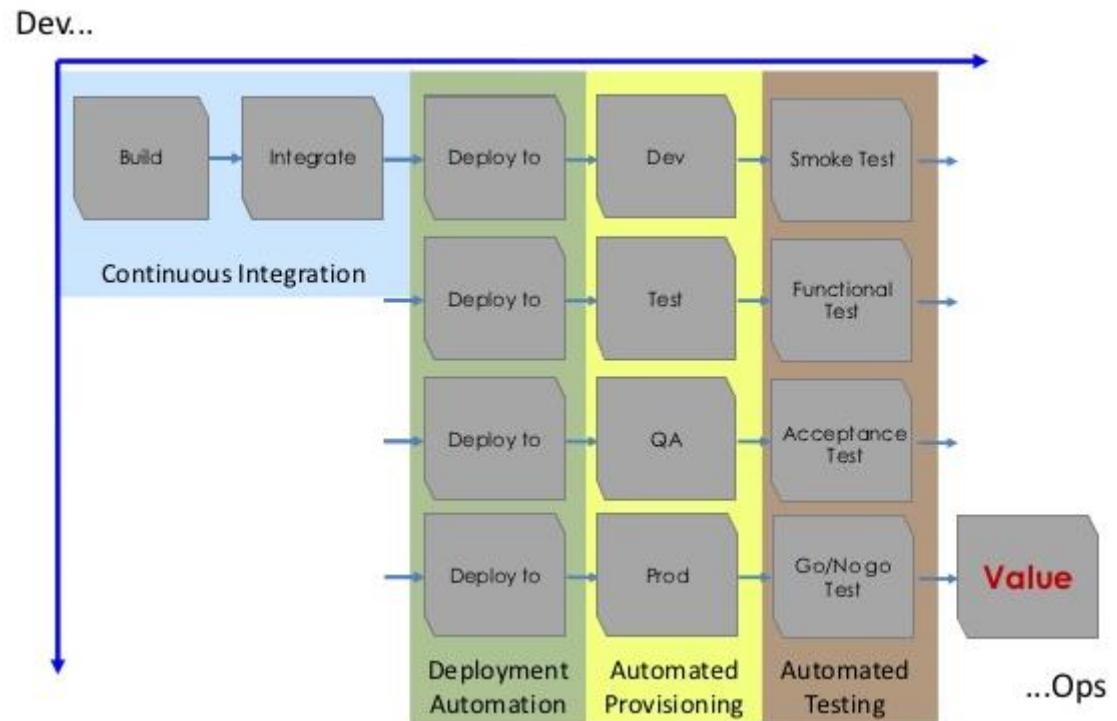
	Commit	Dist	Smoke	Acceptance	Dist-All	Performance
2.3.0.6689 revision: a3a5f8920c48... 1 day ago Triggered by ssudheen	 100	 100	 100	 100	 100	 100
2.3.0.6688 revision: 472ea60c48ac... 5 days ago Triggered by changes	 100	 100	 100	 100	 100	 100
2.3.0.6687 revision: 5780e01322cf... 5 days ago Triggered by changes	 100	 100	 100	 100	 100	 100
2.3.0.6686 revision: 733818836c8b... 5 days ago Triggered by changes	 100	 100	 100	 100	 100	 100
2.3.0.6685	<div style="border: 1px solid black; padding: 5px;"> Git - trunk - https://cruise_builder:*****@bitstdscm01.thoughtworks.com/git/cruise.git jmm <jmonahan@thoughtworks.com> Prototypes: v6 finished 733818836c8bfb6cdafb4b3406cf5558c548e2d Mercurial - twist - https://cruise_builder:*****@bitstdscm01.thoughtworks.com/hg/cruise_qa Rajeshvaran Appasamy <rappasam@thoughtworks.com> Committing the completed script for #5349 4f7813cbedcd8685d6ec3042b2a12aa1b5c95605 Triggered by changes </div>					

Automation – Automate all the things



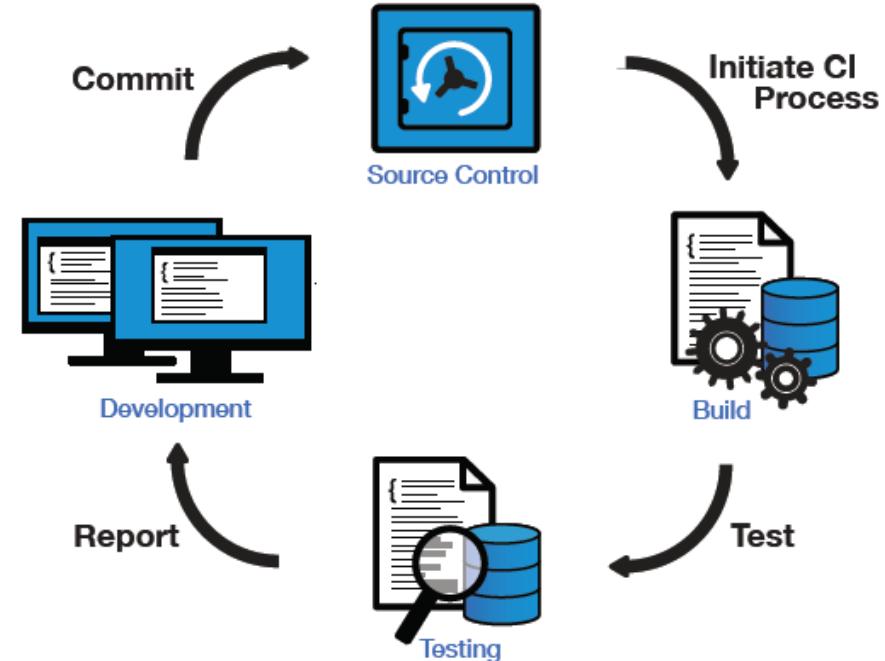
- Reproducible Build
 - Build on a Prod-Like environment
 - No more “Works on my machine”
- Test
 - Testing reduces risks
 - Make you more confident
- Deploy
 - Deploy to Dev, Deploy to QA
 - Deploy to Pre-Prod, Prod

Automation – Continuous Deployment



Automation – Continuous Integration

- Build each “Commit” – check the integration code
- Static Code Analysis
- Perform set of Tests – Unit Test ,Automation Tests , Regression, TDD and compatibility Test
- Nightly and Weekly
- Packaging
- Delivery
- Deploy



Automation – Continuous Deployment

- Deployment Automation/Continuous Delivery
 - Distributing the components of the releasable package into target environment (Dev, Test, Prod)
- Provisioning
 - Create and readying the target Env. And middleware required by your application.
- Automated Testing
 - Verify:
 - Functional
 - Non functional
- Feedback



TESTS = DEPLOYMENT HEALTH

Automation – IAC

Infrastructure As A Code

- Put Infra under **version control**
- History , Melodize your infra
- Deploy monitoring , backups and apps
- **IT IS NOT SCRIPTING !**
- Think Puppet, Chef, Ansible, Udeploy....

Automation – High Availability



Automation – Scalability



Measurements



Measurements - Metrics

- Easy to Create new Metrics
- Build Dashboards
- Learn from logs
- Learn from *

Measurements - Monitoring

- Monitor Every Platform – Dev, QA, Prod, see troubles early
- Run Time:
 - OS
 - Disk, CPU, I/O, Memory
 - Middleware
 - Queues
 - API calls
 - Connections
 - Application
 - Response time
 - Users
 - Objects
 - Usage

Measurements - Monitoring



Measurements – Measure Everything

- Deployments
- Commits
- Tickets
- Bugs

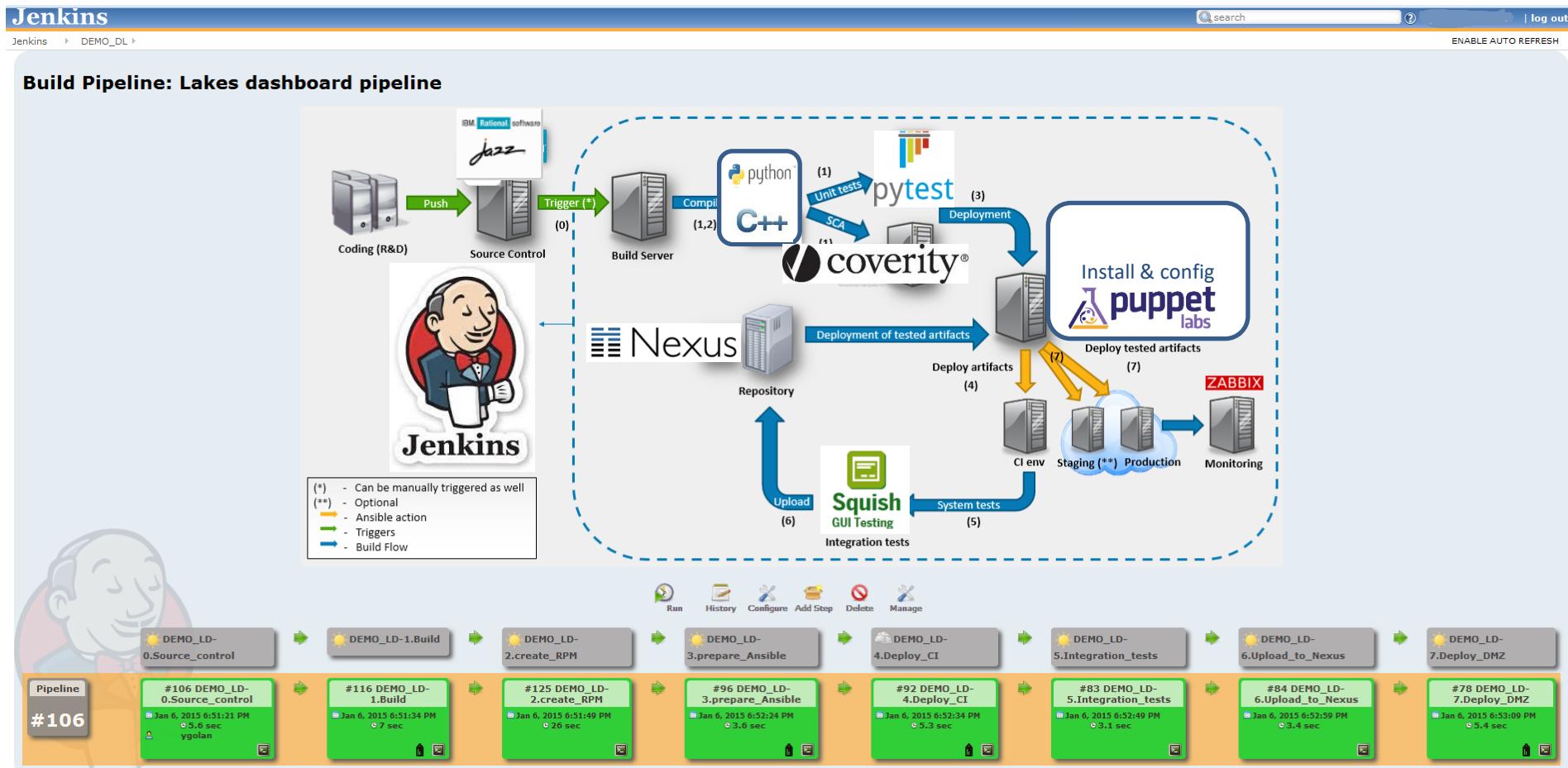
Measurements – Read Metrics

- Get Business Keys
- Share dashboard with dev ops mgmt.
- Find metrics that matter
- Learn from metrics

Sharing

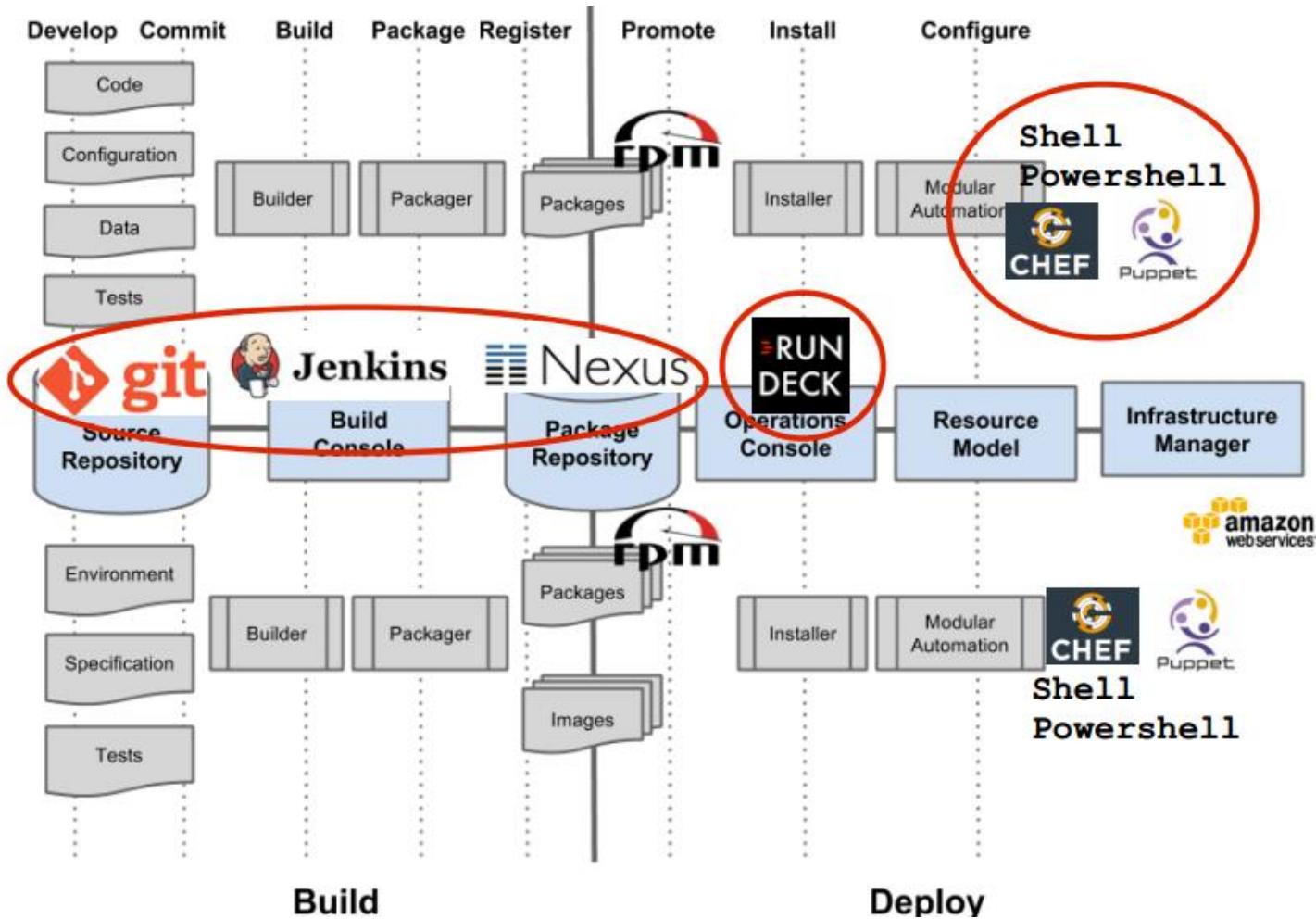


Implementation example



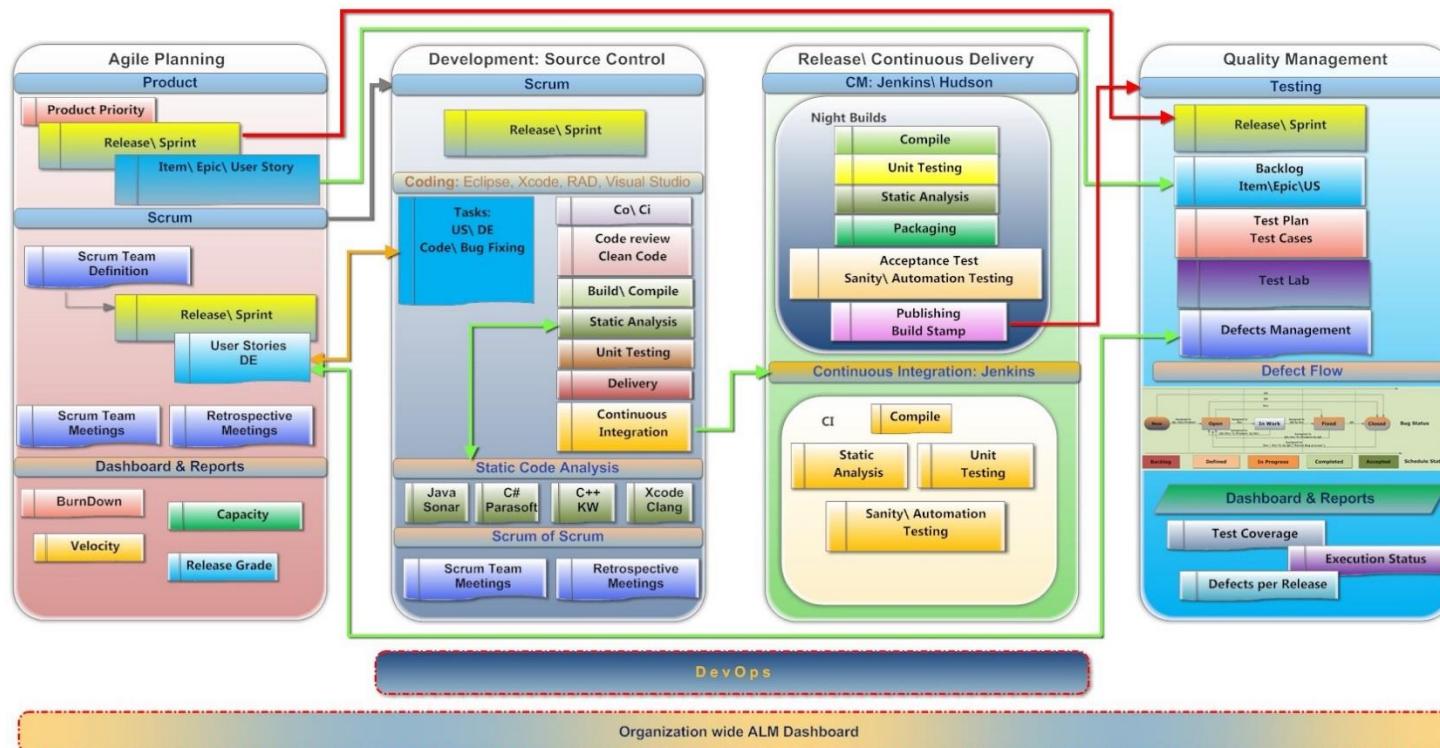
Application

Infrastructure



Full ALM Solution - CI

ALM and Agile



Jenkins

- Jenkins, a continuous build tool, enables teams to focus on their work by automating the build, artifact management, and deployment processes
- Jenkins' core functionality and flexibility allow it to fit in a variety of environments and can help streamline the development process for all stakeholders involved

CI Tools

- Code Repositories: SVN, Mercurial, Git
- Continuous Build Systems: Jenkins, Bamboo, Cruise Control
- Test Frameworks: JUnit, Cucumber, CppUnit
- Artifact Repositories: Nexus, Artifactory, Archiva

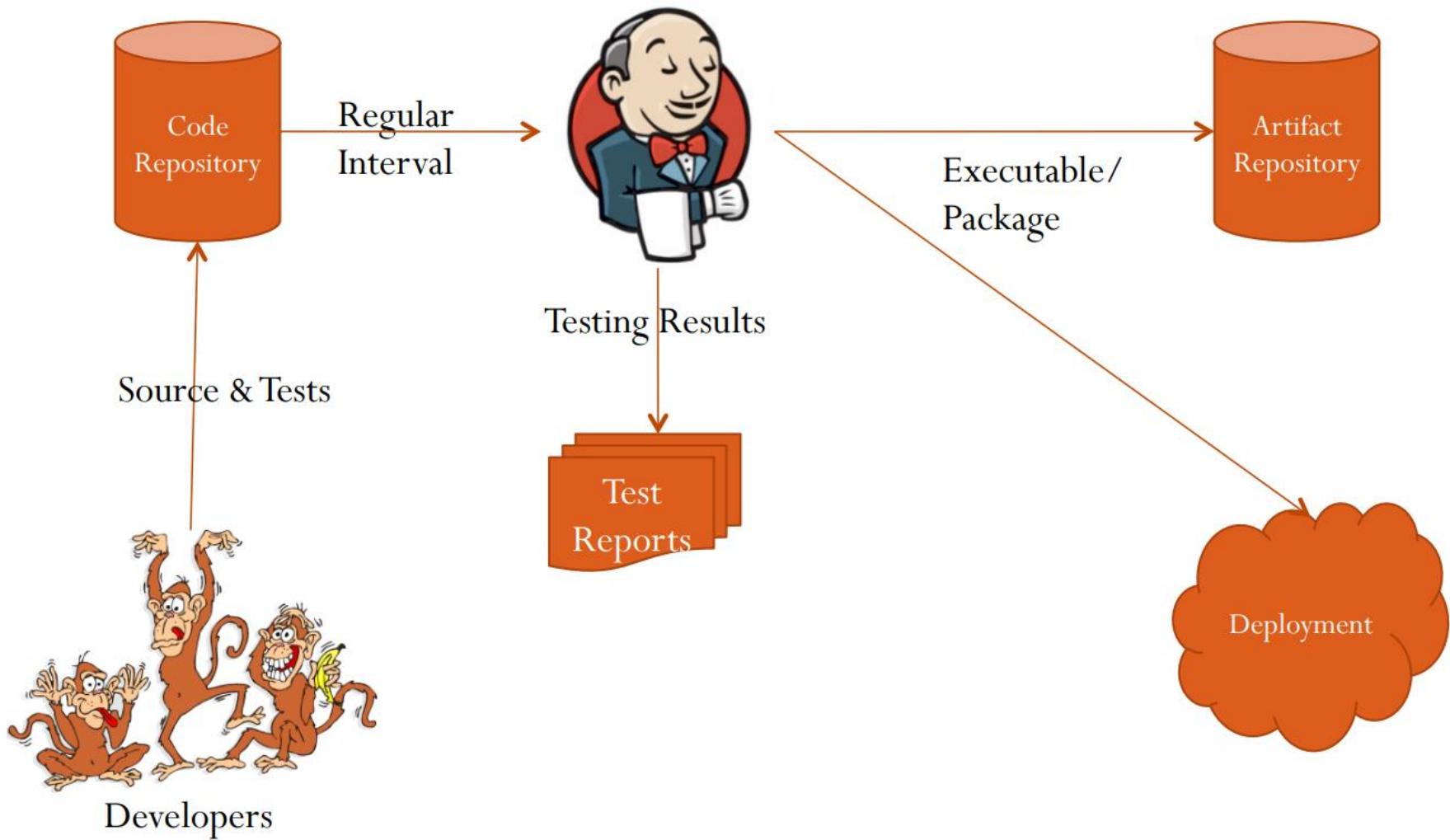
Jenkins

- Branched from Hudson
- Java based Continuous Build System
- Runs in servlet container
 - Glassfish, Tomcat
- Supported by over 400 plugins
 - SCM, Testing, Notifications, Reporting, Artifact Saving, Triggers, External Integration
- Under development since 2005
- <http://jenkins-ci.org/>

History

- 2005 - Hudson was first release by Kohsuke Kawaguchi of Sun Microsystems
- 2010 – Oracle bought Sun Microsystems Due to a naming dispute, Hudson was renamed to Jenkins Oracle continued development of Hudson (as a branch of the original)

Jenkins – Fitting in



Why Jenkins?

- Jenkins is a highly configurable system by itself
- The additional community developed plugins provide even more flexibility
- By combining Jenkins with Ant, Gradle, or other Build Automation tools, the possibilities are limitless

Why Jenkins? Free/OSS

- Jenkins is released under the MIT License
- There is a large support community and thorough documentation
- It's easy to write plugins
- Think something is wrong with it? You can fix it!

What can Jenkins Do?

- Generate test reports
- Integrate with many different Version Control Systems
- Push to various artifact repositories
- Deploys directly to production or test environments
- Notify stakeholders of build status
- ...and much more

How Jenkins works?

- When setting up a project in Jenkins, out of the box you have the following general options:
 - Associating with a version control server
 - Triggering builds
 - Polling, Periodic, Building based on other projects
 - Execution of shell scripts, bash scripts, Ant targets, and Maven targets
 - Artifact archival
 - Publish JUnit test results and Javadocs
 - Email notifications
- Plugins expand the functionality even further

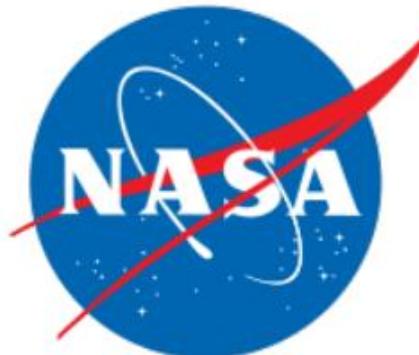
How Jenkins works?

- Jenkins comes with basic reporting features
 - Keeping track of build status
 - Last success and failure
 - “Weather” – Build trend
- These can be greatly enhanced with the use of pre-build plugins
 - Unit test coverage
 - Test result trending
 - Findbugs, Checkstyle, PMD

Enhancing Jenkins

- Jenkins plugin system can enable a wide range of features including (but certainly not limited to)
- SCM
 - Mercurial, Git, Subversion
- Testing
 - Selenium, Windmill, TestLink
- Notifications
 - IRC, Twitter, Jabber
- Reporting
 - Doxygen, PMD, Findbugs
- Artifact Saving
 - Artifactory, Amazon S3, SCP
- Triggers Jabber, Directory Watchers
 - External Integration GitHub, Bugzilla, JIRA
- And most importantly – The CI Game A points based game where developers compete against each other to develop the most stable, welltested code

Who uses Jenkins



Running Jenkins yourself

- Jenkins is packaged as a WAR, so you can drop it into whichever servlet container you prefer to use
- Jenkins comes pre-packaged with a servlet if you just want a lightweight implementation
- Native/Supported packages exist for
 - Windows
 - Ubuntu/Debian
 - Redhat/Fedora/CentOS
 - Mac OSX
 - openSUSE
 - FreeBSD
 - OpenBSD
 - Solaris/OpenIndiana
 - Gentoo

Running Jenkins updates

Jenkins has two release lines

- **Standard releases**
 - Weekly bug fixes and features
- **Long-Term Support releases**
 - Updates about every 3 months
 - Uses a “Stable but older” version from the standard release line
 - Changes are limited to backported, well-tested modifications

Tying it into Agile

- For an Agile team, Jenkins provides everything needed for a robust continuous build system
- Jenkins supports Agile principles by constantly providing access to working copies of software
- Jenkins' extensibility allows the system to adapt to many different pre-existing environments

Putting together

- While an integral part of a CI system, Jenkins is by no means the only component
- In order for a CI system to function, a common repository for the codebase needs to exist
- A database of artifacts needs to exist, so deliveries can be made at past iterations
- The last step in a CI process is the deployment of the components built
- ...and none of this matters if the developers don't use the system; procedures need to ensure the system is used as intended

Self Management

Self
configuring

Self
Optimizing

Self
Healing

Self
Adaptive

Self
Protective

Self
Organizing

Self * (Selfware)

- Self-configuring
- Self-healing
- Self-optimising
- Self-protecting
- Self-aware
- Self-monitor
- Self-adjust
- Self-adaptive
- Self-governing
- Self-managed
- Self-controlling
- Self-repairing
- Self-organizing
- Self-evolving
- Self-reconfiguration
- Self-maintenance

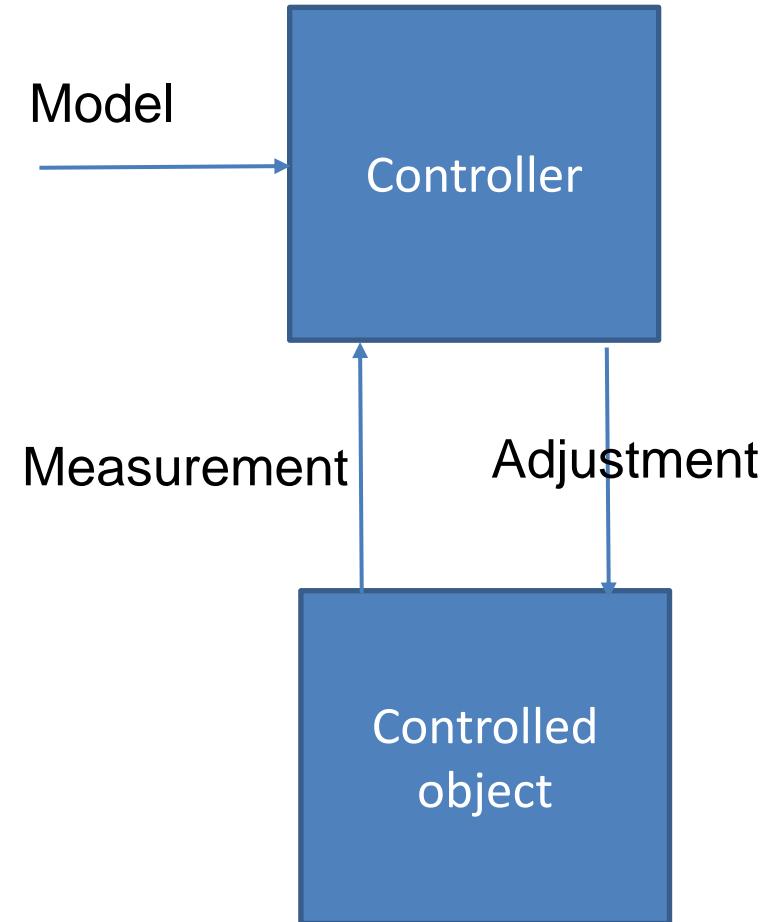
Eight goals of a system

1. System must know itself
2. System must be able to reconfigure itself within its operational environment
3. System must pre-emptively optimize itself
4. System must detect and respond to its own faults as they develop
5. System must detect and respond to intrusions and attacks
6. System must know its context of use
7. System must live in an open world
8. System must actively shrink the gap between user/business goals and IT solutions

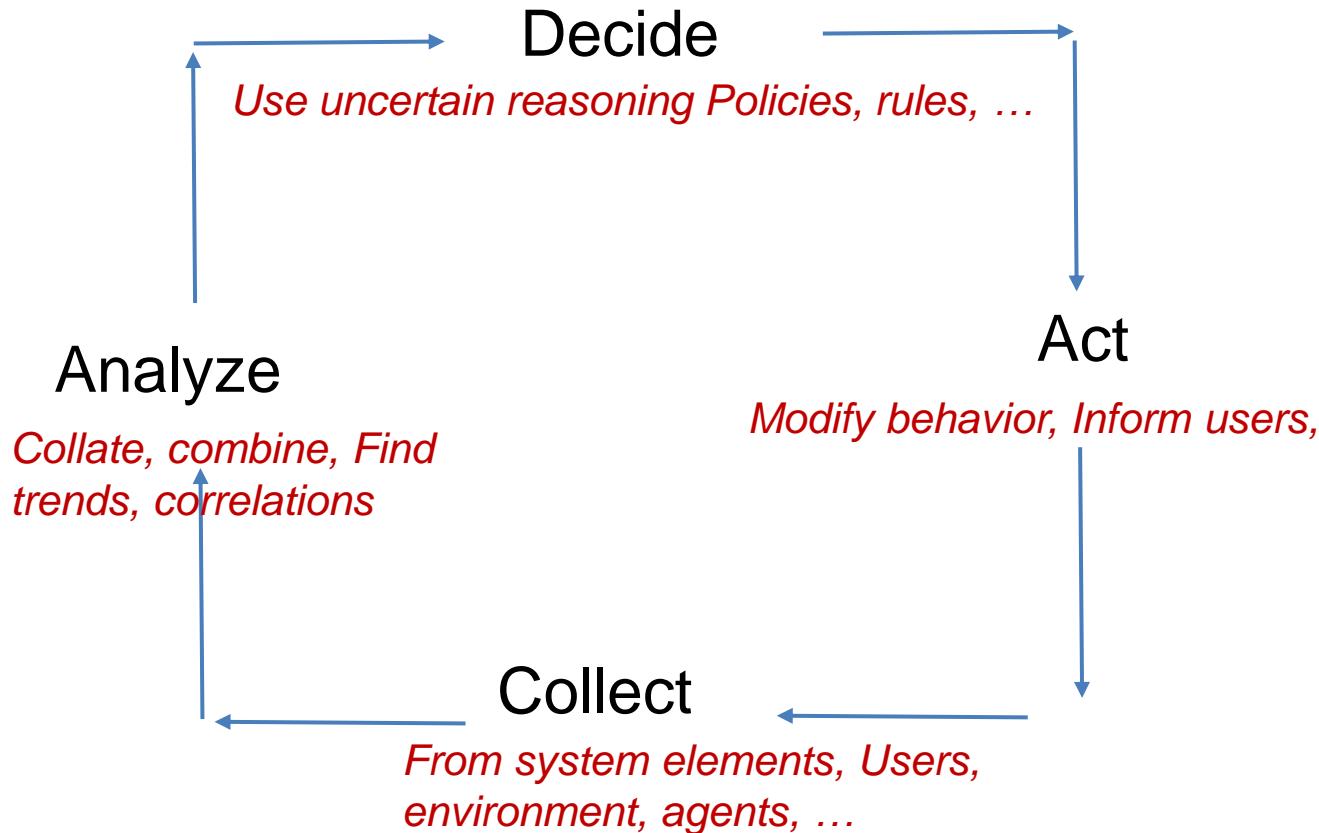
Autonomic Computing



- Basic model: closed control loops
 - Based on Process Control Theory
- Controller continuously compares the actual and expected behavior and makes needed adjustments



Autonomic Control Loop



Elements of Self healing

Fault Model	System Response	System Completeness	Design Context
<ul style="list-style-type: none">• Fault duration• Fault manifestation• Fault Source• Granularity• Fault profile expectations	<ul style="list-style-type: none">• Fault Detection• Degradation• Fault Response• Fault recovery• Time constants Assurance	<ul style="list-style-type: none">• Architectural completeness• Designer Knowledge• System Self-knowledge• System evolution	<ul style="list-style-type: none">• Abstraction level• Component homogeneity• Behavioral pre determination• User involvement in healing• System linearity• System Scope

Size of the self healing unit

Component

- Focus on connectors and component discovery
 - Service
 - Service interfaces, Service discovery, restart
 - Node
 - Network and interface failures, change to new connection

Architectural approach

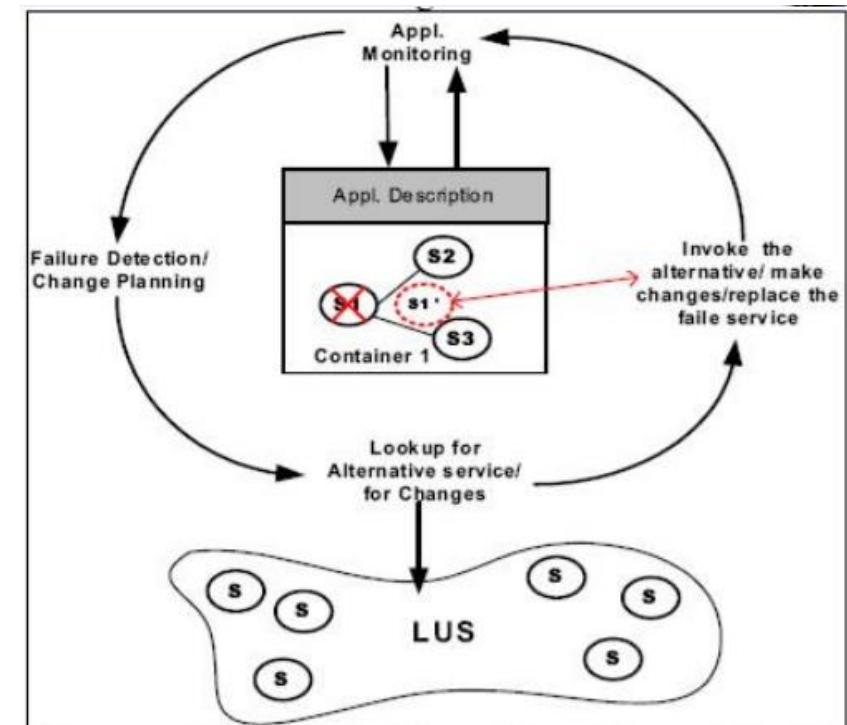
- The healing or recovery part often requires reconfiguration and adaptation
- They change the architecture
 - Locate and use alternative component
 - Restart (or rejuvenation or resurrection) the failed component
- Self-healing can be build on reflective middleware

Experiments

- OSAD – model (On-demand Service Assembly and Delivery)
- MARKS – Middleware Adaptability for Resource discovery, Knowledge usability and Self-healing
- PAC – Autonomic Computing in Personal Computing Environment
- Using self-healing components and connectors

Life-cycle of Self-Healing

- OSAD – On-demand Service Assembly and Delivery
- Prototype in JINI environment
- Looking for alternatives only by name





Thank You!

In our next session: