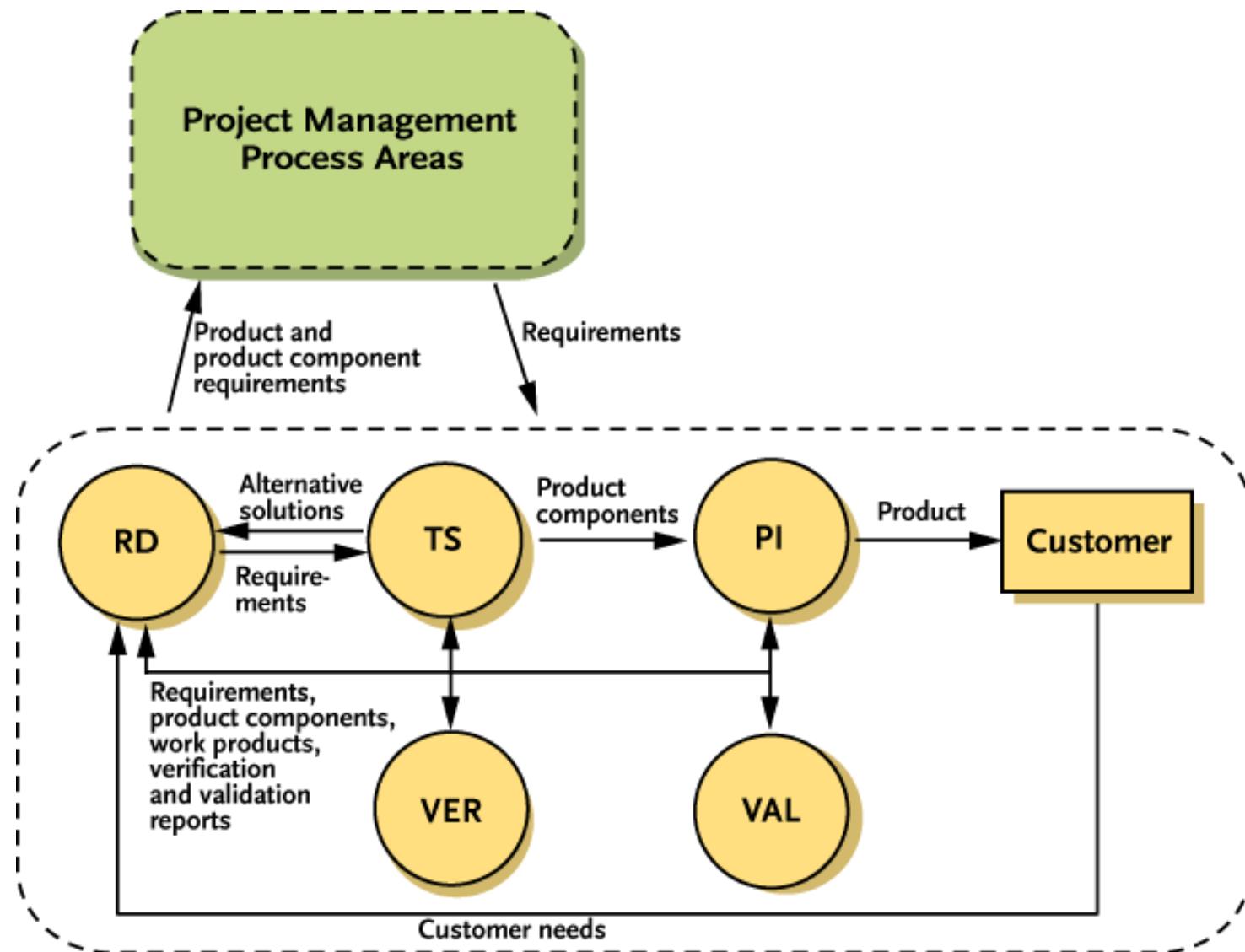


# Užduočių programavimas komandoje: versijų kontrolės sistema

Dr. Asta Slotkienė

# The Engineering (CMMI-DEV) process areas



# The Engineering (CMMI-DEV) process areas: **Technical Solutions**

## **TS.SG 1 Select Product Component Solutions**

Product or product component solutions are selected from alternative solutions.

## **TS.SG 2 Develop the Design**

Product or product component designs are developed.

## **TS.SG 3 Implement the Product Design**

Product components, and associated support documentation, are implemented from their designs.

- **TS.SP 3.1 Implement the Design:** implement the designs of the product components.
- **TS.SP 3.2 Develop Product Support Documentation:** develop and maintain the end-use documentation.

# How to effectively manage a team (tasks/issues programing)



# How to effectively manage a team (tasks programing)

1. Ideally, different people work on different features
2. Need to separate your work from that of others
3. Need to be able to only add your work when it's done

*How to effectively manage a Software Development Team?*

# How to effectively manage a team



# Reason to Use Version Control

- You find a set of files like this in a directory:

Report\_draft.txt

Report.doc

Report2.doc

Report2a.doc

Report\_2019\_03\_02.doc

...

NewReport\_2019\_05.doc

NewReport\_submitted.doc

NewReport\_submittedv2.doc

- Can I just use the newest?

# Folder system/File system

- Progressively harder to manage
- What if you have collaborators?
- What if you want to go back to a certain state?
- How do you track changes?

# Problems Working Alone

- Ever done one of the following:
  1. dead code that worked
  2. made changes and saved it, which broke the code,
  3. and now you just want the working version back...
  - *Accidentally deleted a critical file, and hundreds of lines of code are gone...*
  - *Somehow messed up the structure/contents of your code base, and want to just “undo” the crazy action you just did*
  - *Hard drive crash!!!! Everything’s gone the day before the deadline.*
- Possible options:
  - Save as (MyClass-v1.java)
  - UPS. And now a single line change results in duplicating the entire file...

# Problems Working in the Teams

- Whose computer stores the "official" copy of the project?
  - Can we store the project files in a neutral "official" location?
- Will we be able to read/write each other's changes?
  - Do we have the right file permissions?
  - Lets just email changed files back and forth!
- What happens if we both try to edit the same file?
  - X member just overwrote a file I worked on for 6 hours!
- What happens if we make a mistake and corrupt an important file?
  - Is there a way to keep backups of our project files?

**How do I know what code each teammate is working on?**

# Version Control System

- A software tool that helps you **keep track of changes** in your data (code) over time
- Version Control System (VCS) is a tool or set of tools that provides management of changes to files over time:
  - Uniquely identified changes (**what**)
  - Time stamps of the changes (**when**)
  - Author of the changes (**who**)

# What is version control?

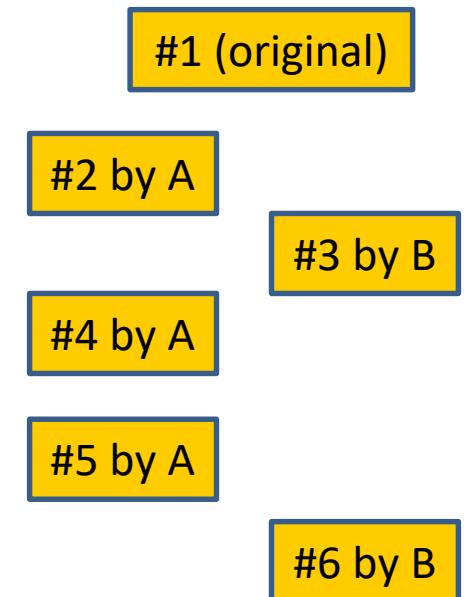
- **Version control** (or *revision control*) is the term for the management of source files, and all of the intermediate stages as development proceeds.
- A **version control system** is a repository of files.
  - Every change made to the source is tracked, along with who made the change, etc.
- Other items can be kept in a **version control system** in addition to source files -- Project Charter, Product Backlog, Design Document, Sprint Planning Document, Sprint Retrospective....

# History

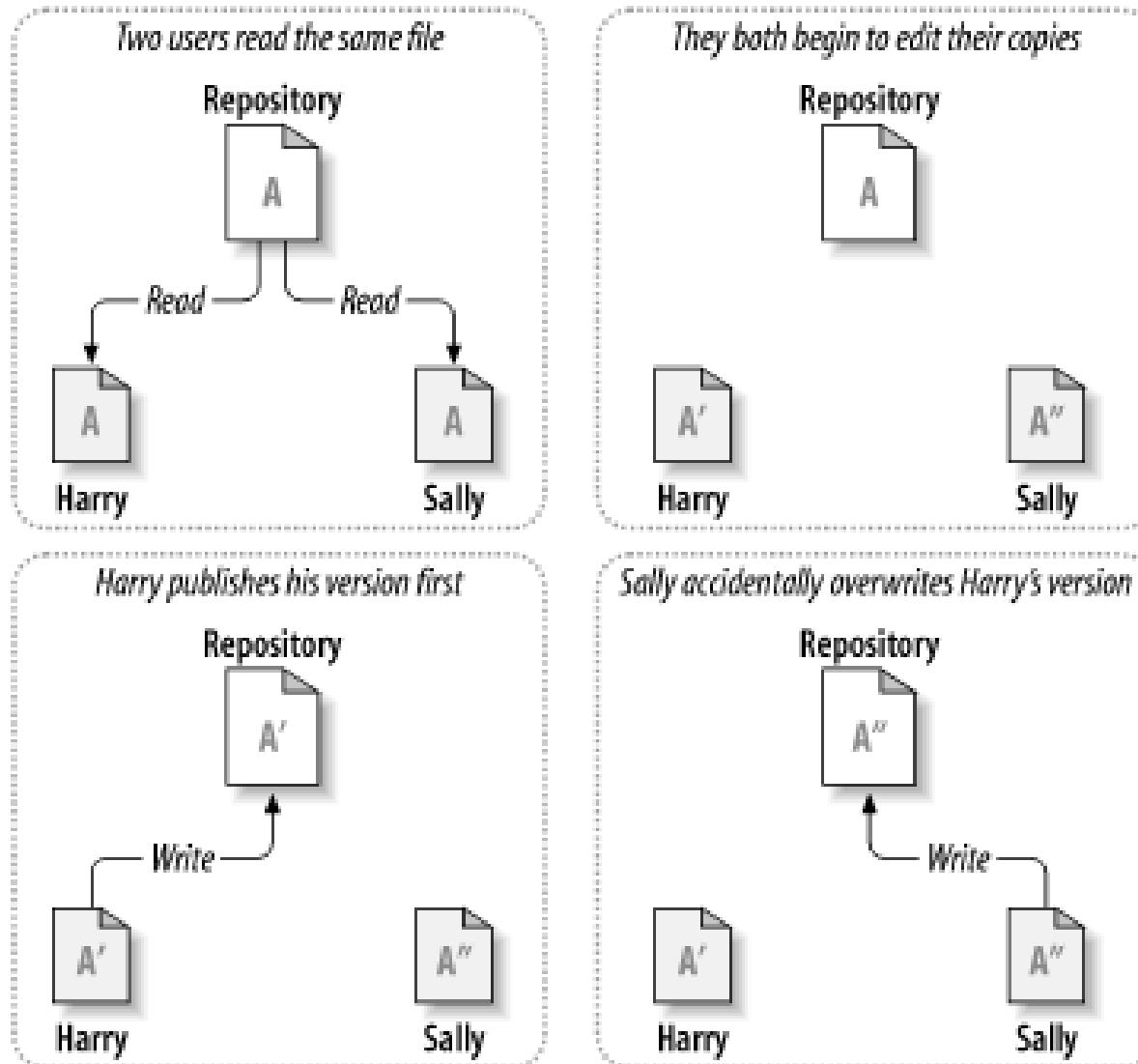
- 1972 – Source Code Control System (SCCS)
  - Store changes using deltas
  - Keeps multiple versions of a complete directory
  - Keeps original documents and changes from one version to the next
- 1982 – Revision Control System (RCS)
  - Keeps the current version and applies changes to go back to older versions
  - Single file at a time
- 1986 – Concurrent Versions Systems (CVS)
  - Start as scripts on top of the RCS
  - Handle multiple files at a time
  - Client-Server architecture

# Three Generations of Version Control

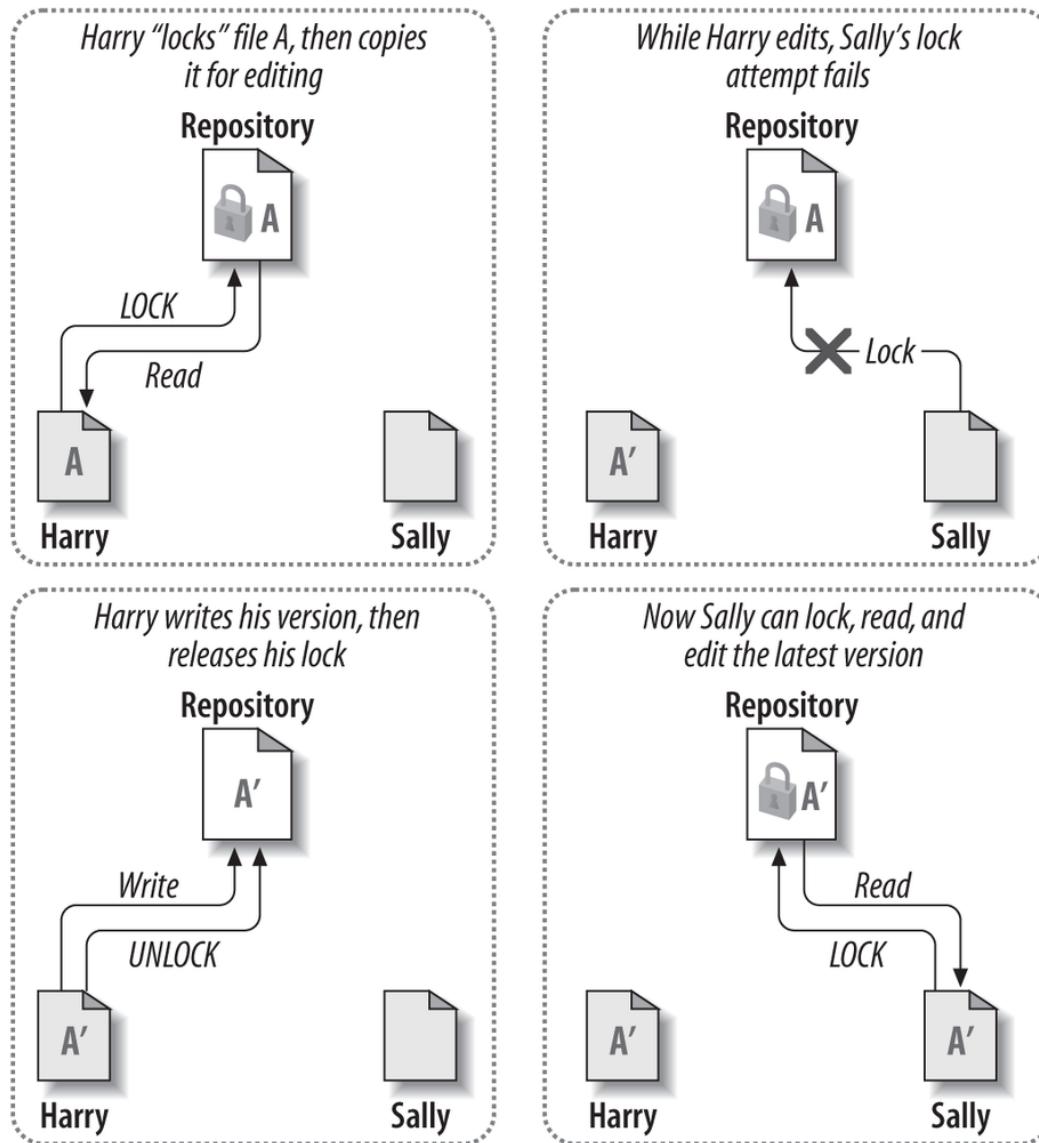
- **First Generation:**
  - One file at a time
  - Lock: only one person could be working on a file at a time.
  - Example: RCS, SCCS



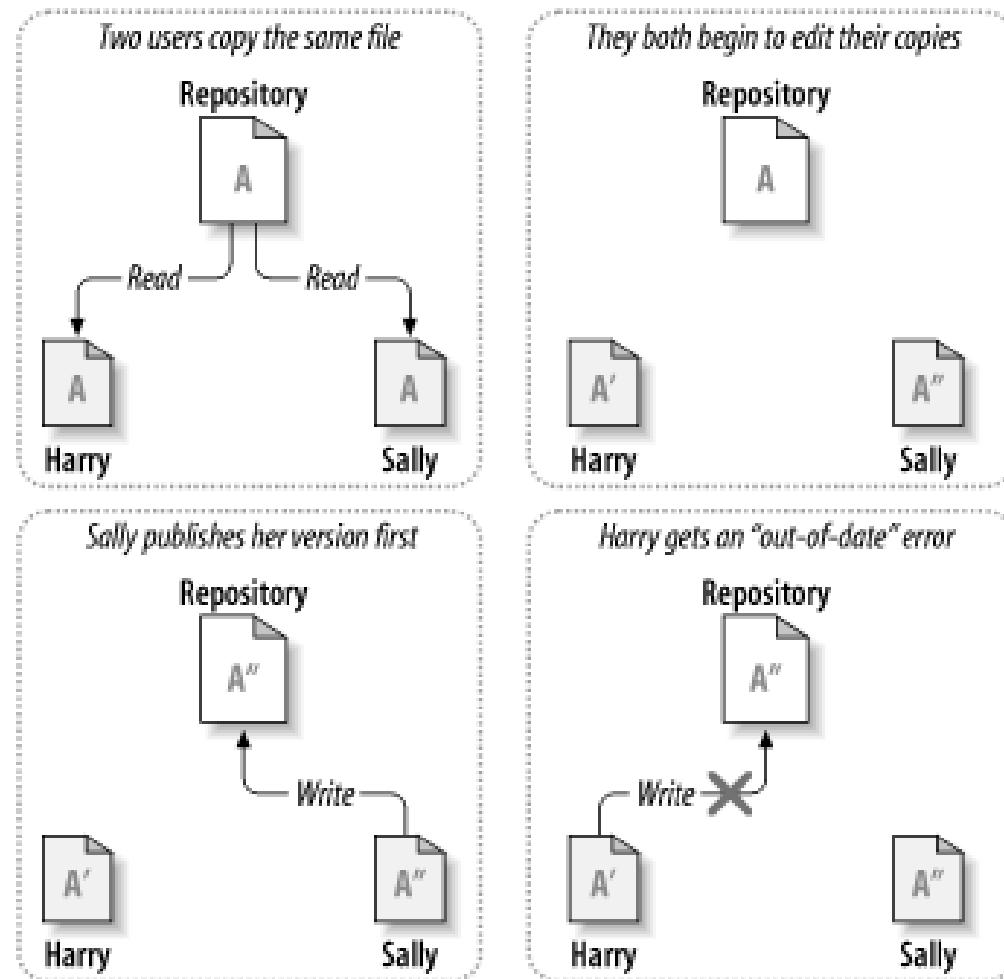
# First Generation: The Problem to Avoid



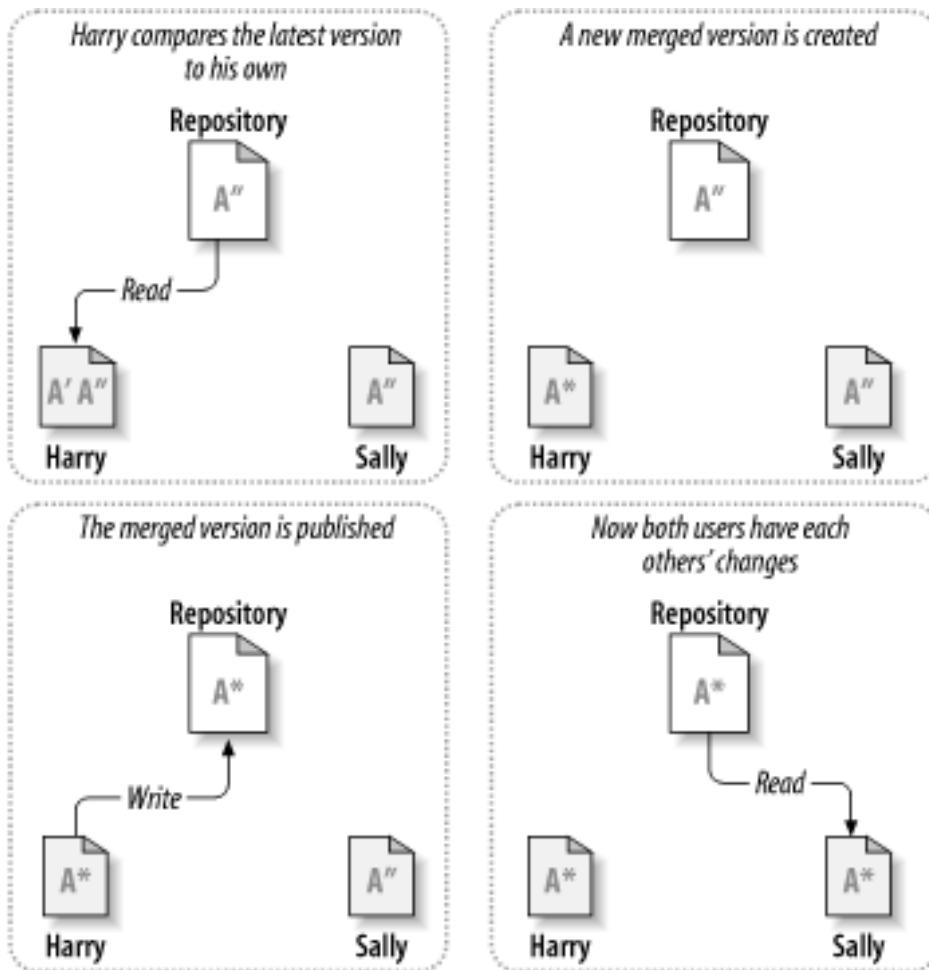
# First Generation: *lock-modify-unlock*



# First Generation: The Copy-Modify-Merge Solution 1

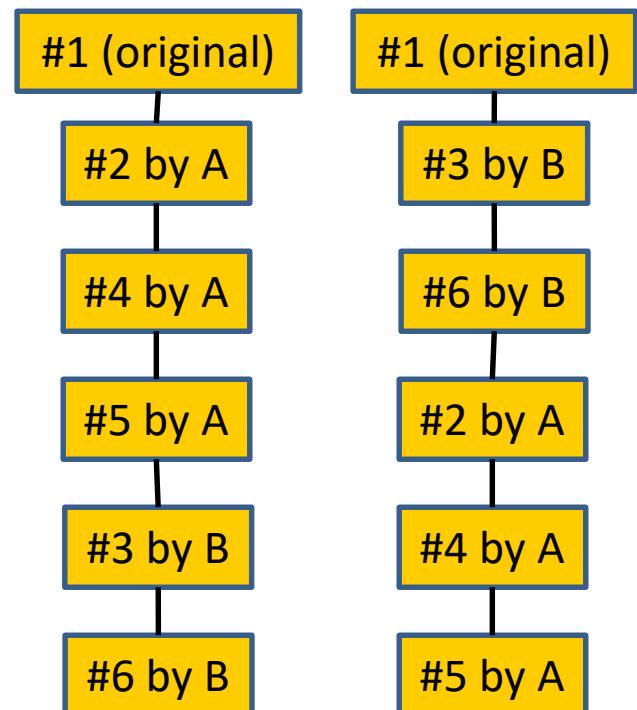


# First Generation: The Copy-Modify-Merge Solution 2



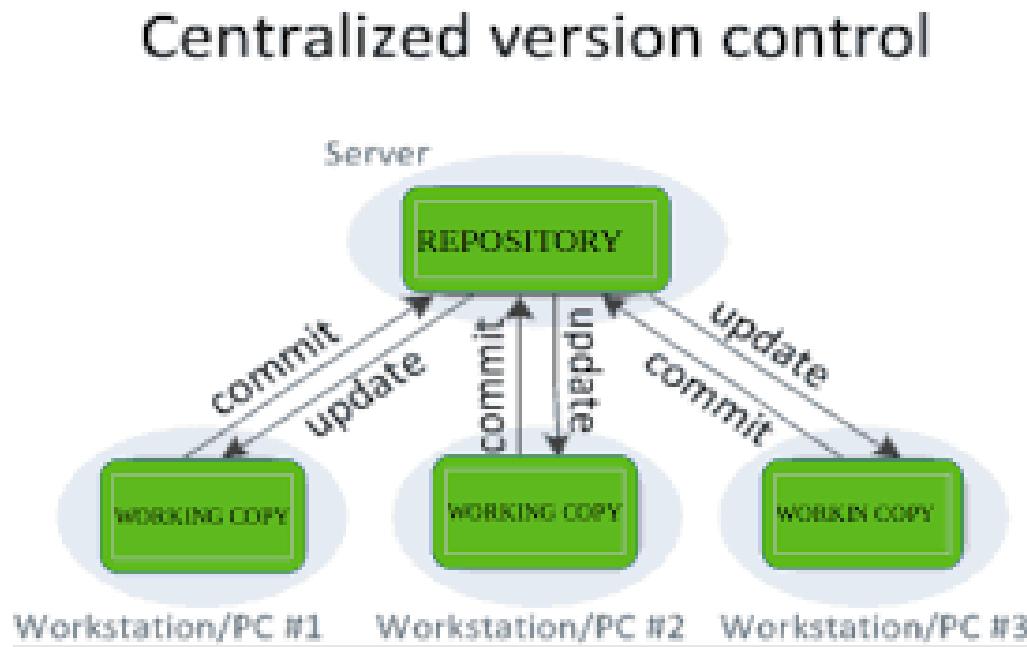
# Three Generations of Version Control

- **Second Generation:**
  - Centralization: one central shared repository
  - MultiFile
  - Merge before commit
  - Rewrites history
  - Example:
    - CVS,
    - SourceSafe,
    - Subversion,
    - Team Foundation Server



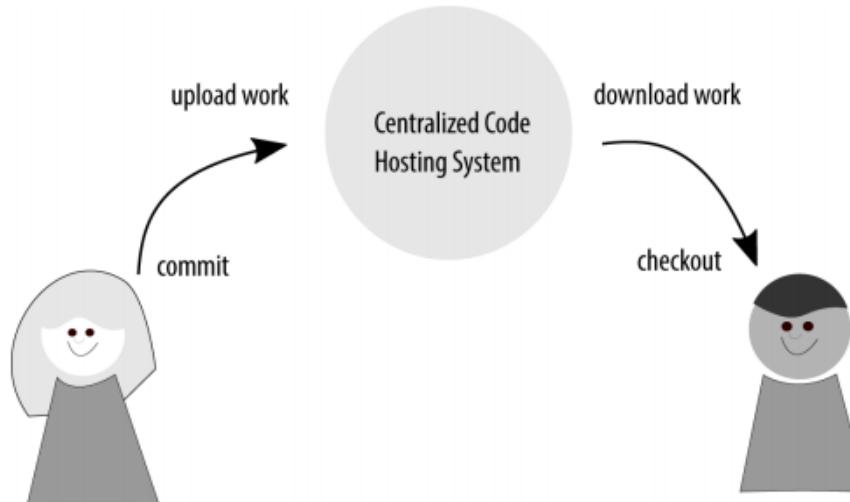
# Centralized version control

- **Centralized version control systems** are based on the idea that there is a **single “central” copy** of your project on a single server and programmers “commit” their changes to this central copy.



# Centralized version control

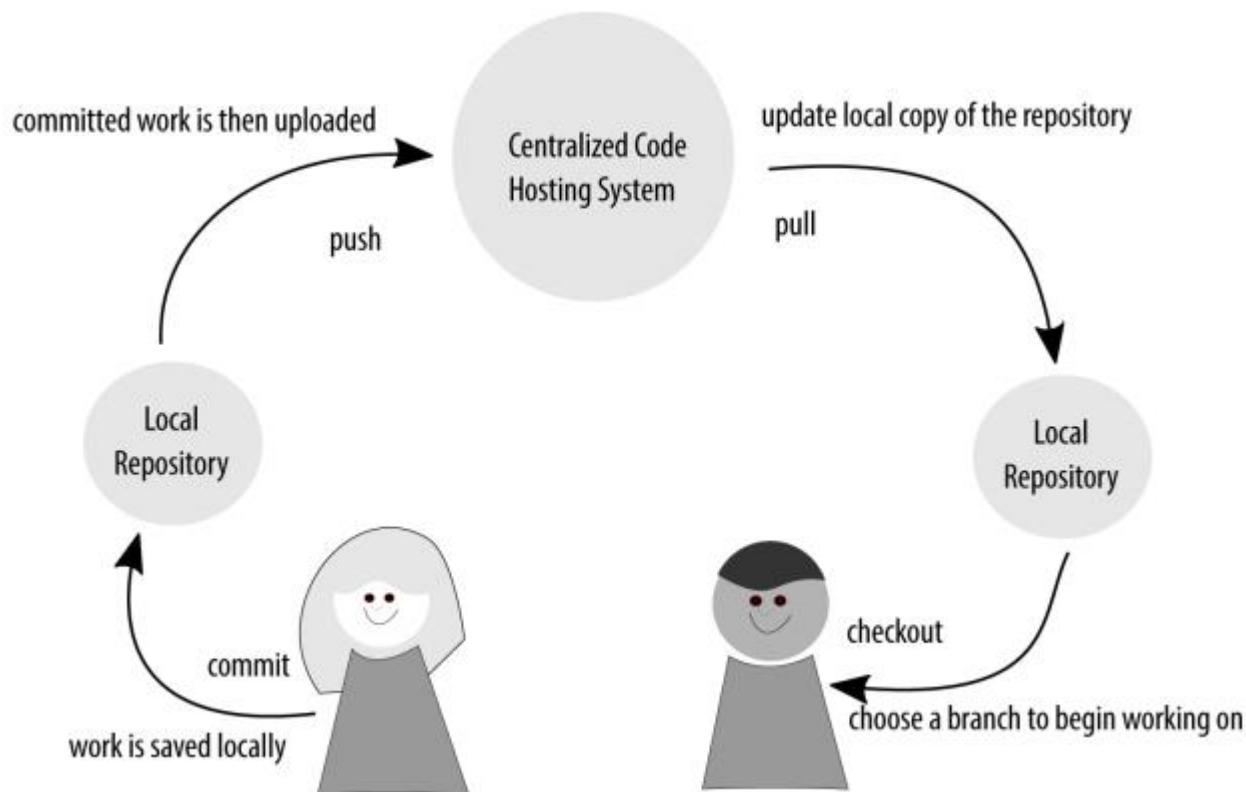
- Just when you thought you were ready to share your work, or request a code review, **you would sometimes be prevented from doing so if someone else had recently updated the same branch with their own work.**



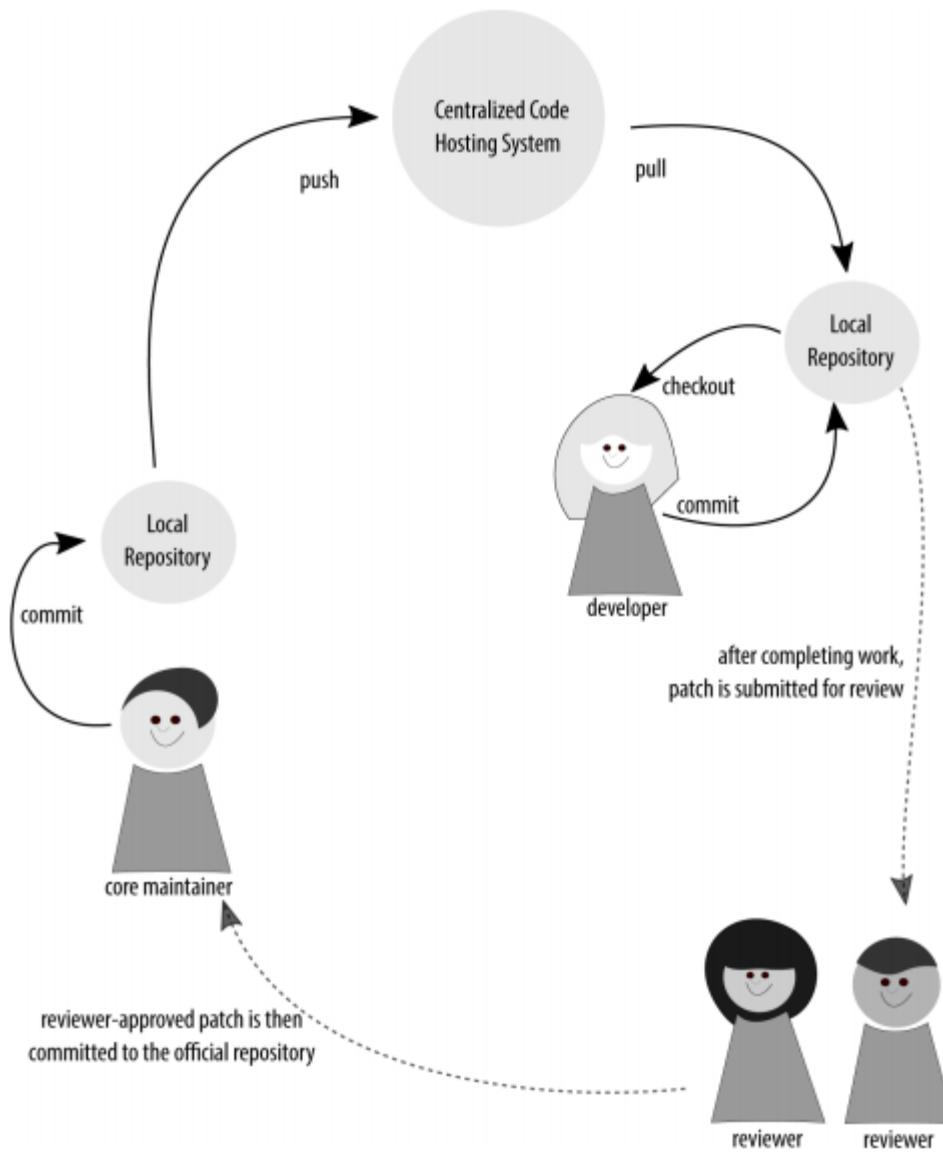
- Working with files in Subversion

# Centralized version control

Centralized is simple, and what you'd first invent:  
a single place everyone can check in and check out.



# The community review process for patches



# Centralized version control

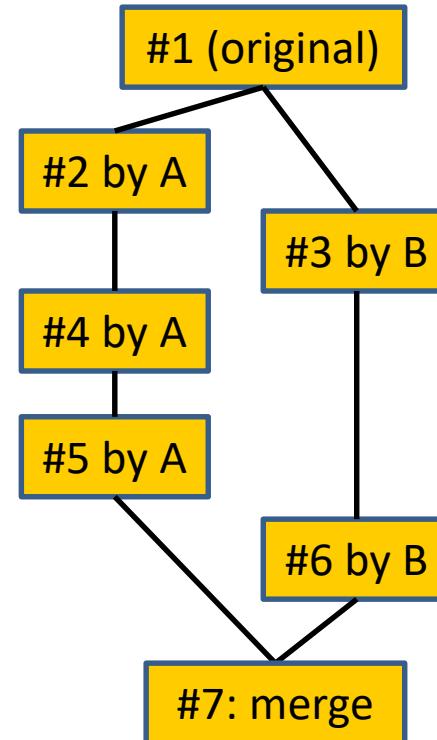
- This model works **for backup, undo and synchronization** but **isn't great for merging and branching changes people make.**
- As projects grow, you want to split features into chunks, developing and testing in isolation and **slowly merging changes into the main line.**
- **Recording/Downloading and applying** a change are separate steps
- Centralized version control focuses on **synchronizing, tracking, and backing up files.**

# Version Control Examples



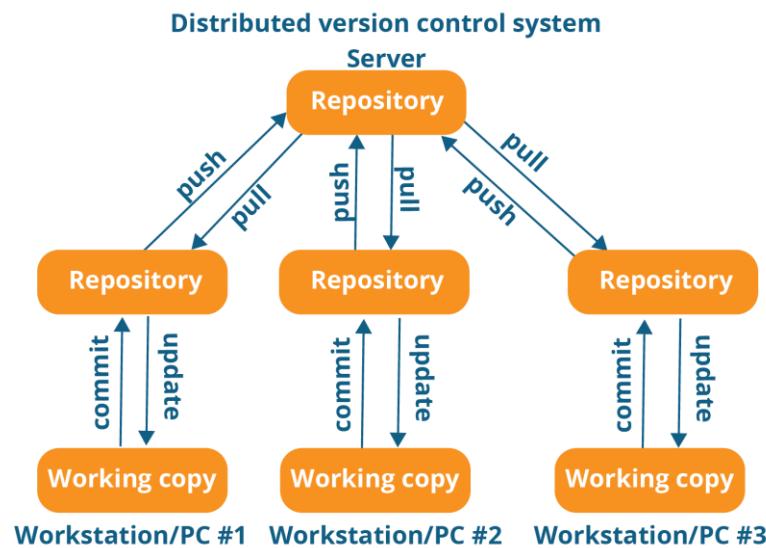
# Three Generations of Version Control

- **Third Generation:**
  - Changesets
  - Commit before merge
  - Example: Bazaar, Git, Mercurial



# Three Generations of Version Control

- Repository - including the **whole history** - is held on each developer's own machine
- **New revisions are checked in to local repository**
- Local repository can be synchronised with other remote repositories to send your changes to other people

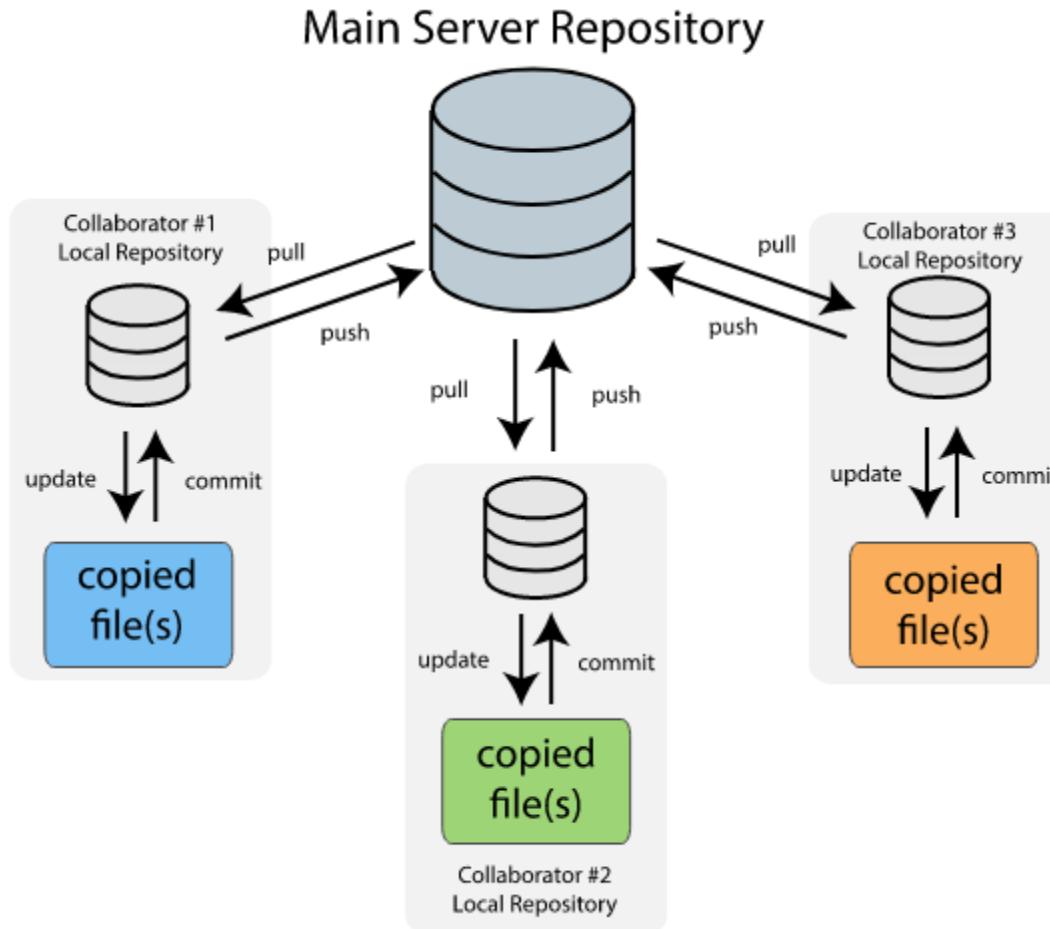


# Advantages of DVCS

- Most operations done on the local repository
  - Much faster
  - Possible offline
- Each developer has the full history of the project
  - If anyone loses theirs - they just need to clone someone else's copy
  - Changes can easily be sent to different remote repositories

E.g. always send to “development” repository, only send to “production” repository when code is tested

# Distributed Version Control Systems



Distributed Version Control Systems use a peer-to-peer approach to version control, which allows users to work productively when not connected to a network

# Additional

- Understanding Version-Control Systems Eric Raymond
  - <http://www.catb.org/~esr/writings/version-control/version-control.html>
- A brief history of version control
  - <https://www.red-gate.com/blog/database-devops/history-of-version-control>

# Why use version control

- A version control system (VCS) is a tool for **managing a collection of program code** that provides you with three important capabilities:
  - *Reversibility (grjžtamumas/atgaminimas)*,
  - *Concurrency (išlygiagretinimas)*
  - *annotation (anotavimas)*

# Why use version control: reverssibilty

- The ability to **back up to a saved**, known-good state when you discover that some modification you did was a mistake or a bad idea.

# Why use version control: **concurrency**

- The ability to have many people modifying the same collection of code or documents knowing that **conflicting modifications can be detected and resolved.**

# Why use version control: annotate

- The capability to *annotate* your data, **attaching explanatory comments** about the intention behind each change to it and a record of who was responsible for each change.
  - Even for a **programmer working solo**, change histories are an important **aid to memory**;
  - for a **multi-person project** they become a vitally important **form of communication among developers**.

# Why is version control important?

- **Version control** allows us to:
  - Keep everything of importance in one place
  - Manage changes made by the team
  - Track changes in the code and other items
  - Avoid conflicting changes
  - Revert to a previous state of data

# Benefits of Version Control System

- Individual benefits
  - Backups with tracking changes
  - Tagging – marking the particular version in time
  - Branching – multiple versions
  - Tracking changes
  - Revert (undo) changes

# Benefits of Version Control System

- Team benefits
  - Working on the same code sources in a team of several developers
  - Merging concurrent changes
  - Support for conflicts resolution when the same file (the same part of the file) has been simultaneously changed by several developers
  - Determine the author and time of the changes

# Version Control Systems



Gitea - Git with a cup of tea

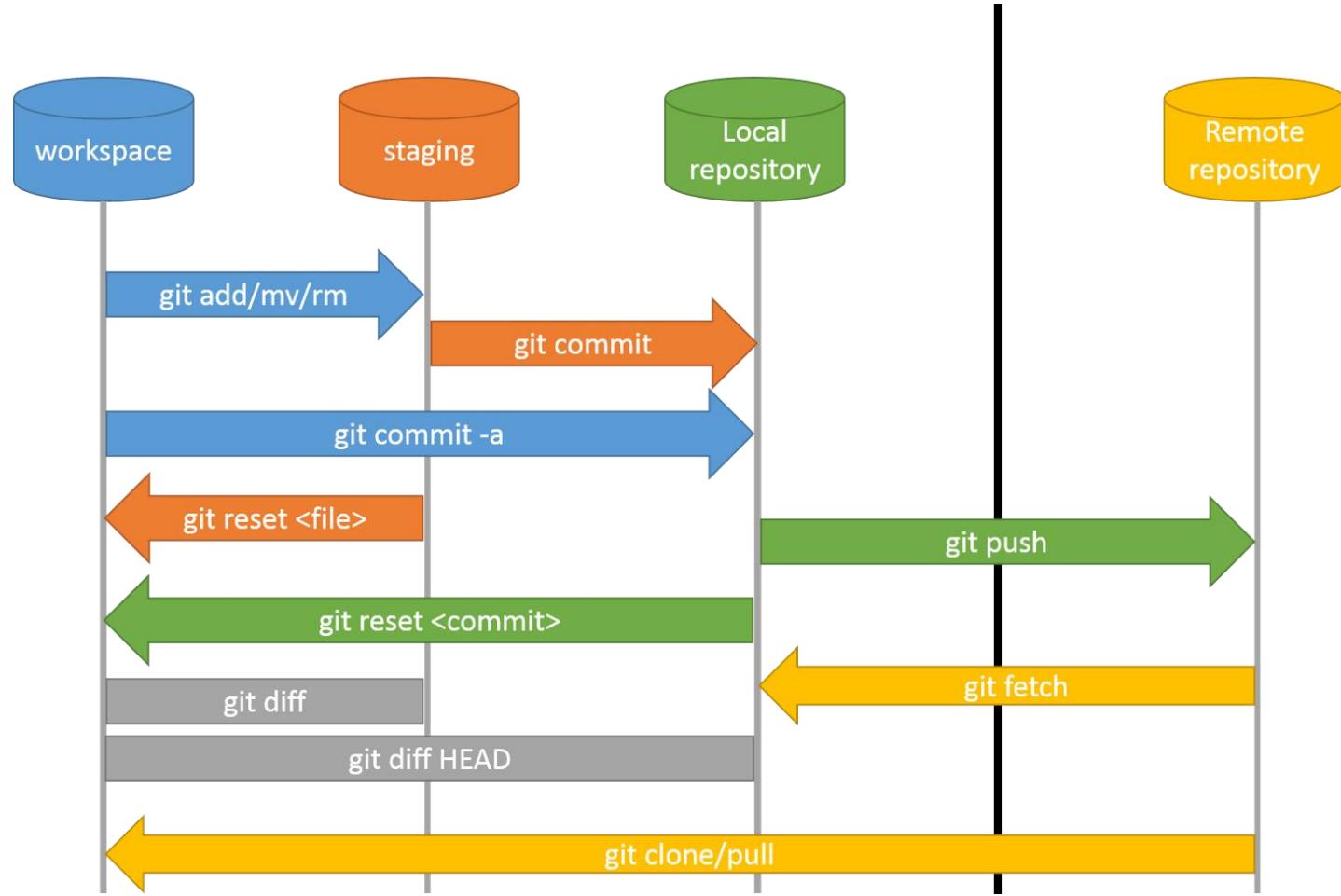


Azure DevOps



<https://www.tekkiwebsolutions.com/blog/github-vs-bitbucket-vs-gitlab/>

# GIT processes

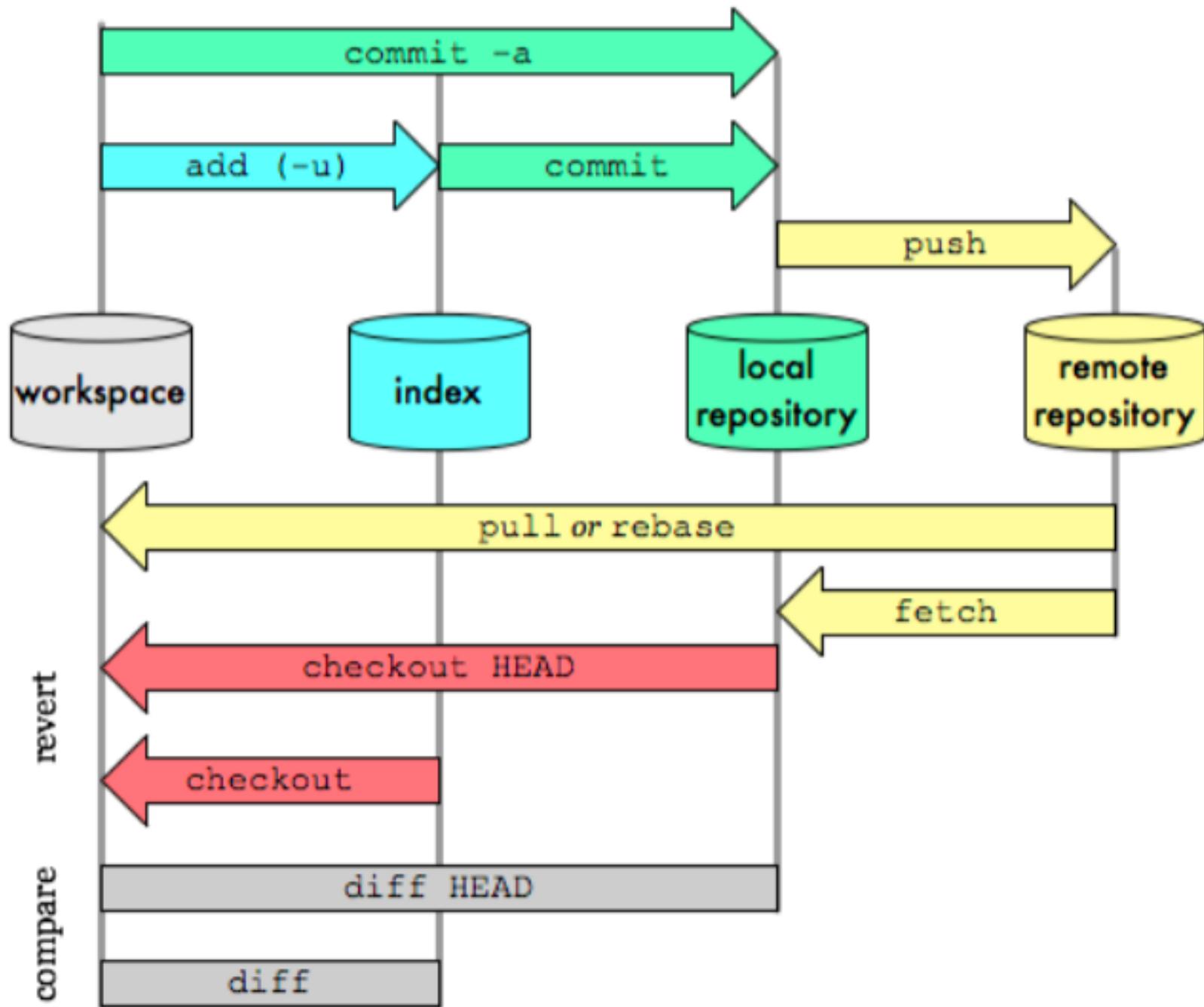


# Main terminology I

- **Repository (repo)**: a database containing the history
- **Server**: The computer storing the repo.
- **Client**: The computer connecting to the repo.
- **Working Set/Working Copy**: your local directory of files/ current state of the data, where you are
- **Revision / commit** (noun)- the state of data at a given time
- **Check out** - to retrieve a revision from the repository
- **Check in / commit** (verb) - to send a revision to the repository changes.

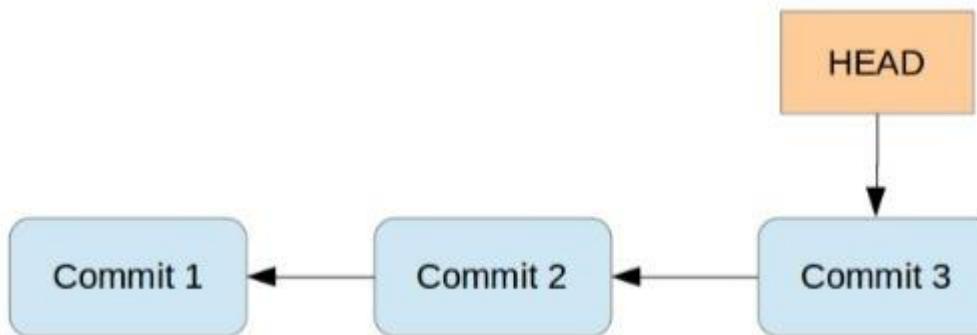
# Main terminology II

- **Staging area** is a place to record things before committing.
  - Physically it's the `.git/index` file that makes up the content of the Staging Area
- **Master**: Default name for initial branch. When a new repository is initialized with `$ git init`, a branch named `master` is automatically created.
- **Branch**: Lightweight moveable pointer referencing one particular *commit*.
- **Hash (SHA-ID, "commit hash")**: Unique commit identifier, derived from the content and metadata provided at time of commit, using the cryptographic SHA
- **Working Tree (Work Space)**: The actual files and folders on disk currently checked out for editing, e.g. what you see in your editor or file explorer. It also contains metadata about any changes made, that are not yet staged or ignored, and can be shown using `$ git status`.



# Commit/Revision

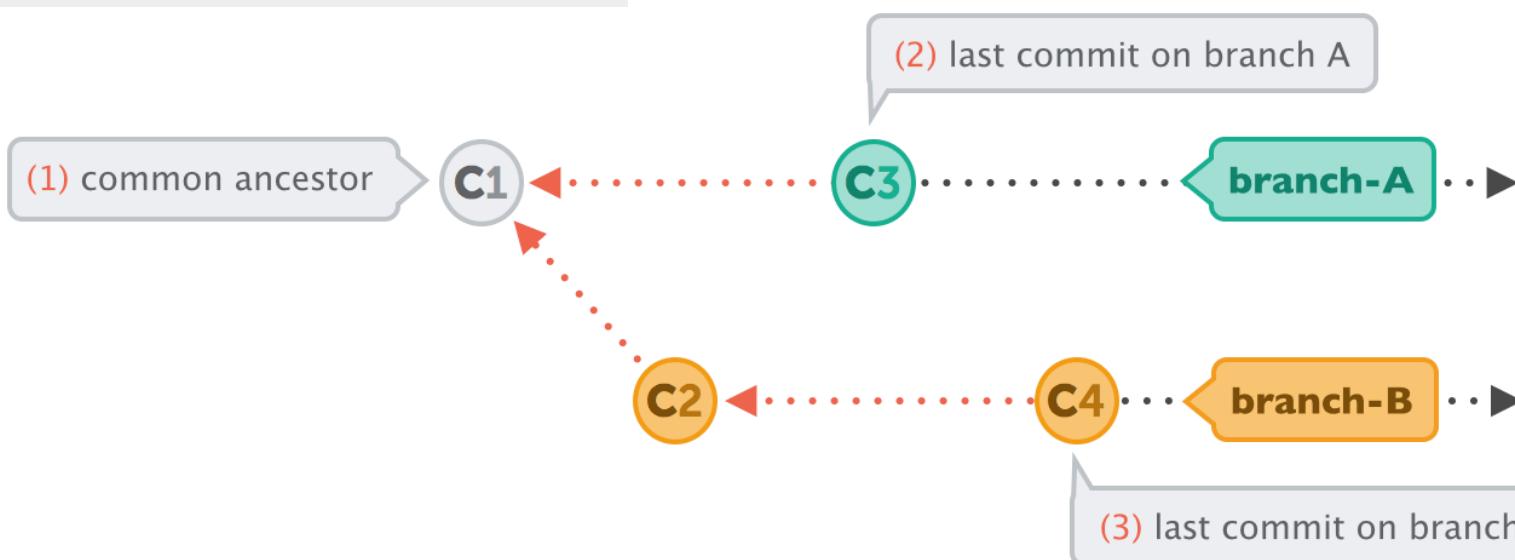
- When you commit your changes into a repository this creates a new *commit object* in the Git repository.
  - This *commit object* uniquely identifies a new revision of the content of the repository.
- Every revision has: a revision ID/number; message; author; time
- Commits in Git are **immutable snapshots**
  - *Immutable object:* In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created. This is in contrast to a mutable object, which can be modified after it is created.
- Every revision (except the first) has at **least one parent** revision that it builds upon.



# Commit/Revision

- Commit structure, where child commits reference parent commits, is technically a directed acyclic graph (DAG) and is what makes up the repository history.
- The DAG is referred to as a history graph or commit-graph.
- SHA algorithm produces a 40 digit long (20 byte) hash value, also known as message digest. In Git, a commit can be referenced using its entire hash

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
```



# Add/Restore

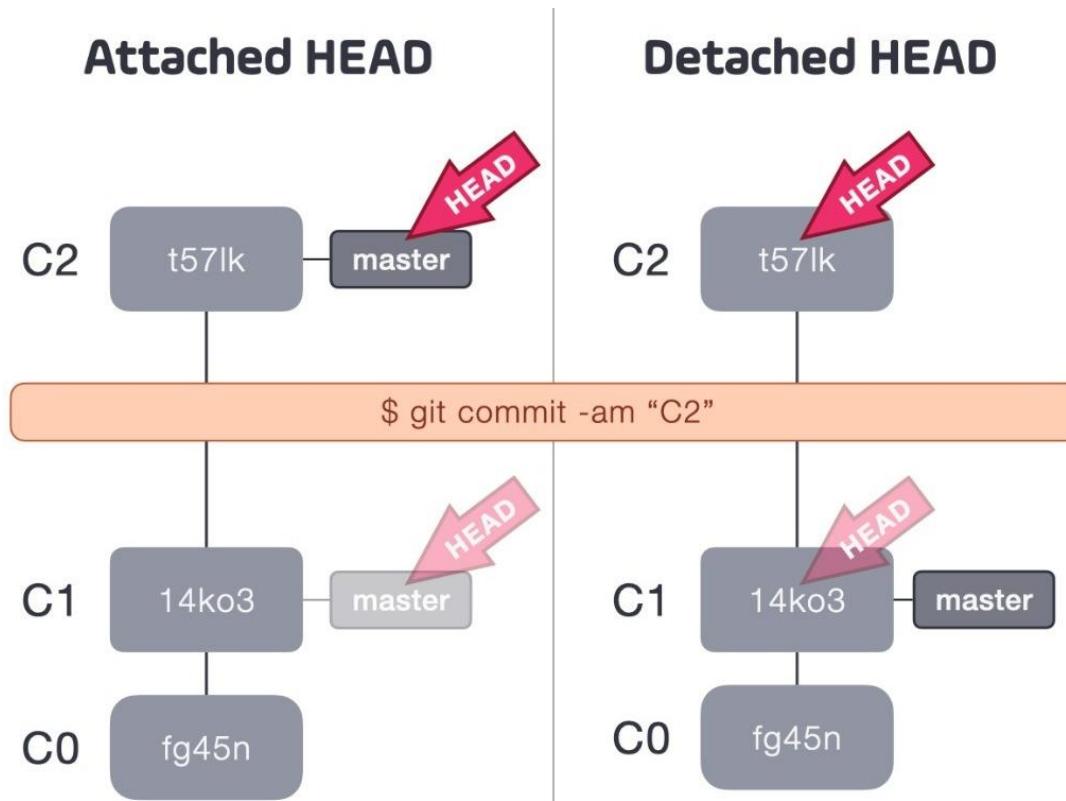
- **Add:** Moves files from *Working Tree* to *Staging Area*.
  - Updates **the index** using the current content found in the working tree, to prepare the content staged for the next commit.
  - `git add`
- **Restores** changes made to files in *Staging Area* or *Working Tree*.
  - When used for files in *Working Tree* it resets them to their initial state based on *HEAD*, discarding any changes made.
  - <https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified>

# HEAD

- *HEAD* is a symbolic reference most often pointing to the currently checked-out branch or commit
  - **Where am I right now in the repository?**
  - it's Git's way of knowing on which commit to mirror your local Working Tree on, and whether you're currently working on a branch (attached) or not (detached).
- Default state is attached, where any manipulation to the history is **automatically recorded to the branch HEAD is currently referencing.**
- Technically HEAD is a plain text reference, but unlike branches and lightweight tags that are stored in **.git/refs**, it is instead stored inside: **.git/HEAD**

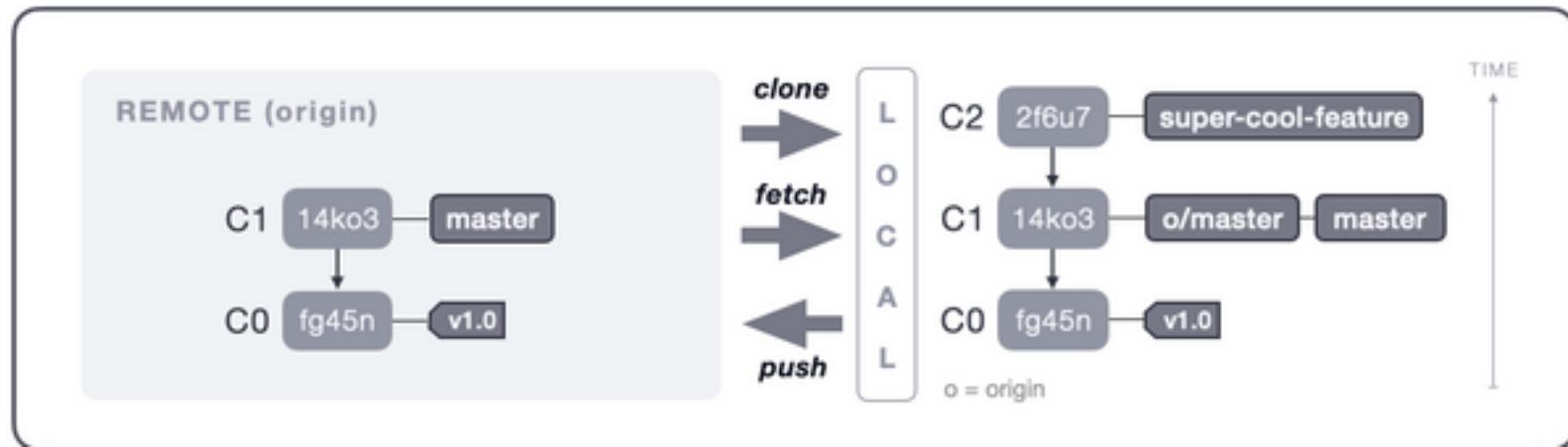
# HEAD

- The main difference is that **in the attached state the change is automatically recorded in the master branch.**
- In the detached state, **the change did not impact any existing branch and master remained to reference C1.**



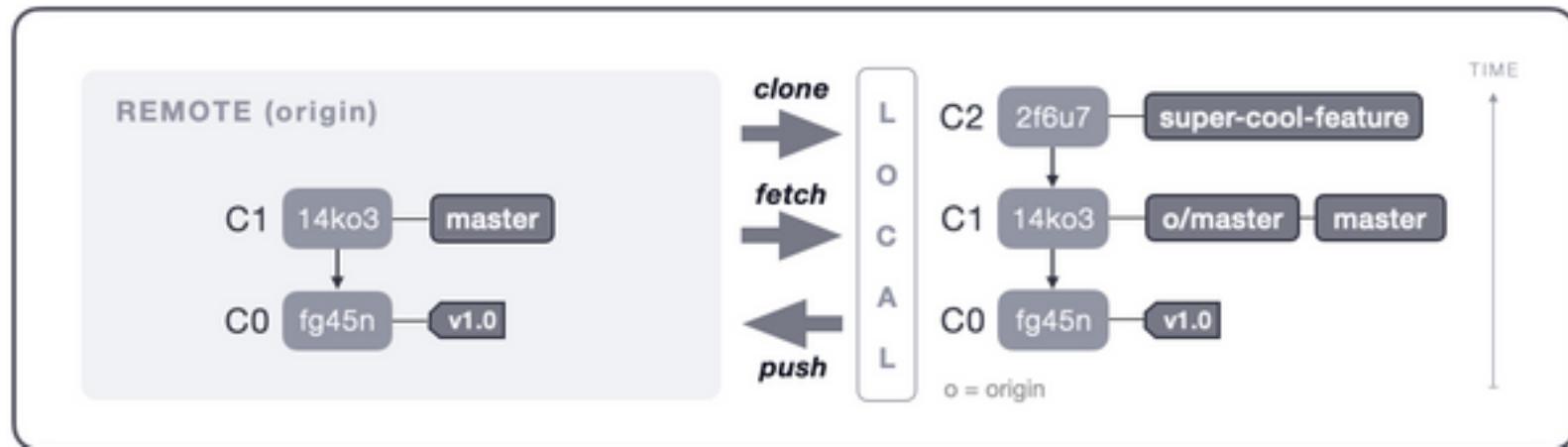
# Clone/Fetch

- **Clone:** Copies an entire remote repository down to your local machine, setting up a cloned version and checks out the default branch (generally master); this action is done only once.
- **Fetch:** Updates remote references in your *cloned* local repository.
  - if a developer has pushed changes to a remote branch, those changes will be pulled down to your repository whenever *fetch* is performed. Note: *fetch* won't automatically merge any changes, only update references.



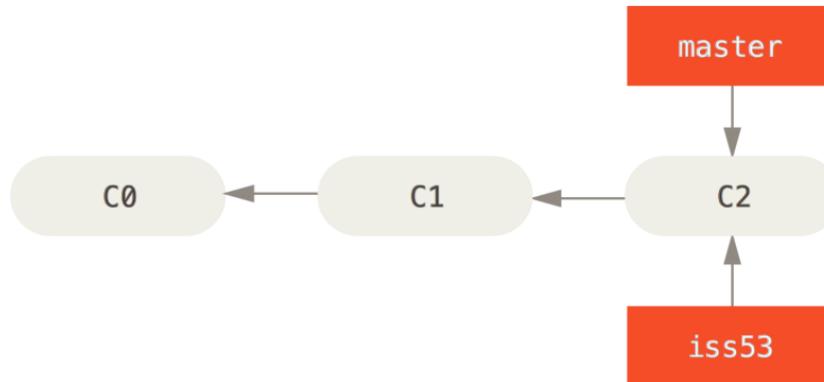
# Push

- **Push:** Makes your local changes publicly available in a remote repository
  - Updates remote refs using local refs, while sending objects necessary to complete the given refs.
  - `git push <repo name> <branch name>`



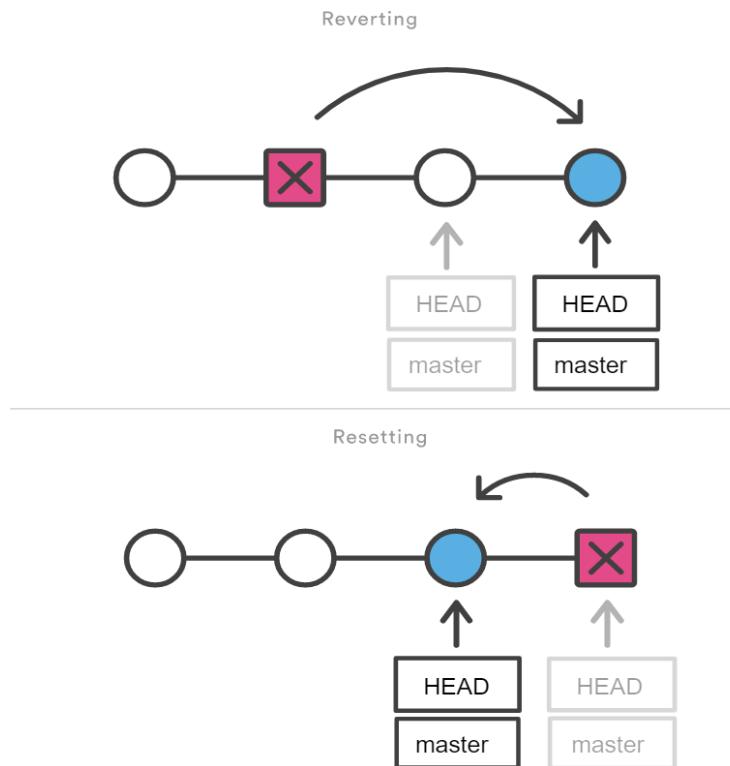
# Checkout

- **Check out-** download a file from the repo.
- checks out a particular version of one or multiple files from history into your *Working Tree*
- git checkout iss53



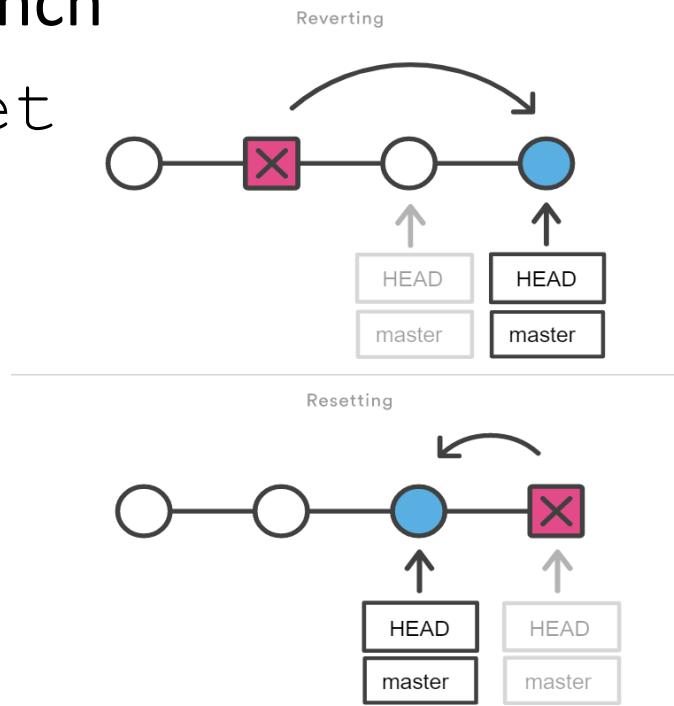
# Checkout/revert

- If you don't like your changes and want to start over, you can **revert** to the previous version and start again (or stop).
- Reverting undoes a commit by creating a new commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history.



# Checkout/reset

- Resetting is a way to move the tip of a branch to a different commit.
  - This can be used to remove commits from the current branch
  - `git reset`



# Git commands

```
git init      # initialize new repository  
git add       # add files or stage file(s)  
git commit    # commit staged file(s)  
git commit -m <title> -m <description>  
git status    # see what is going on  
git log       # see history  
git diff      # show unstaged/uncommitted  
               modifications  
git show      # show the change for a specific  
               commit  
git mv        # move tracked files  
git rm        # remove tracked files
```

# Conventional Commits

- Commitizen is a tool designed for teams
- Template:

<type>[optional scope]: <description>

[optional body]

[optional footer(s)]

# Conventional Commits

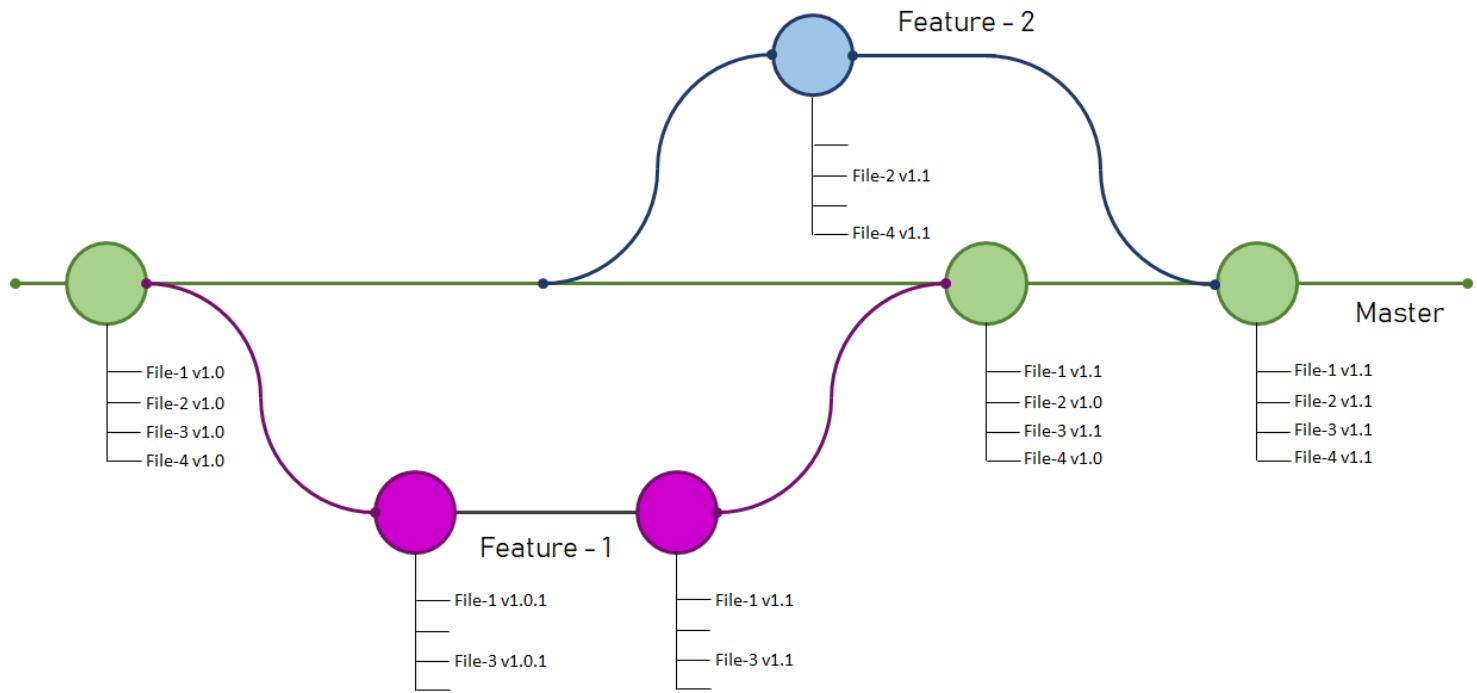
- feat – a new feature is introduced with the changes
- fix – a bug fix has occurred
- chore – changes that do not relate to a fix or feature and don't modify src or test files (for example updating dependencies)
- refactor – refactored code that neither fixes a bug nor adds a feature
- docs – updates to documentation such as a the README or other markdown files
- style – changes that do not affect the meaning of the code, likely related to code formatting such as white-space, missing semi-colons, and so on.
- test – including new or correcting previous tests
- perf – performance improvements
- ci – continuous integration related
- build – changes that affect the build system or external dependencies
- revert – reverts a previous commit

# Conventional Commits

- Good examples of commit
  - feat: improve performance with lazy load implementation for images
  - Fix bug preventing users from submitting the subscribe form
  - Update incorrect client phone number within footer body per client request
- Bad examples of commit
  - fixed bug on landing page
  - Changed style
  - I think I fixed it this time?
  - empty commit messages

# Branch

- The branch is a lightweight movable **pointer referencing a specific commit** in the repository's commit history.
- They **allow multiple developers to work on the same codebase** simultaneously without interfering with each other's work



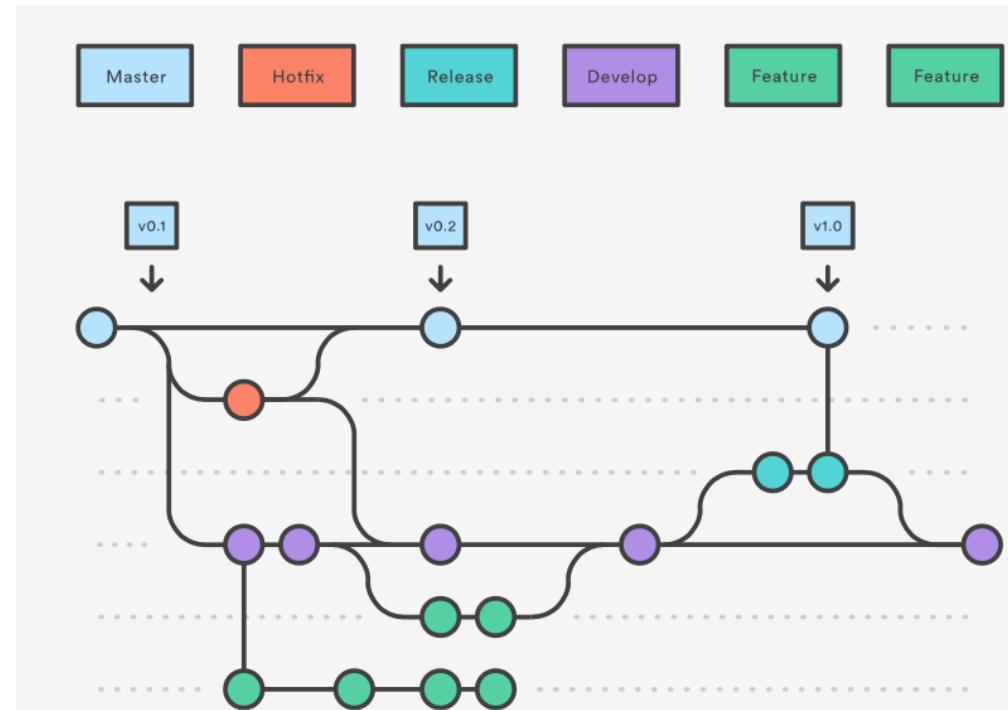
# Branch

- Branches enable you to have multiple parallel versions of data
- Separate development of unrelated features
- Generally the main branch (“master”) should have the latest tested & stable version of the software
- Operations with branch:
  - Create
  - Delete
  - Shows a list
  - Rename
  - Checkout
  - Visualise
  - ...
- Git mantra: “**branch early; branch often**”

# Types of Branches

Create GitFlow:

- Production branch: master
- Develop branch: develop
- Feature prefix: feature/
- Release prefix: release/
- Hotfix prefix: hotfix/
- Depending on the project



# Types of Branches

Smaller or solo projects:

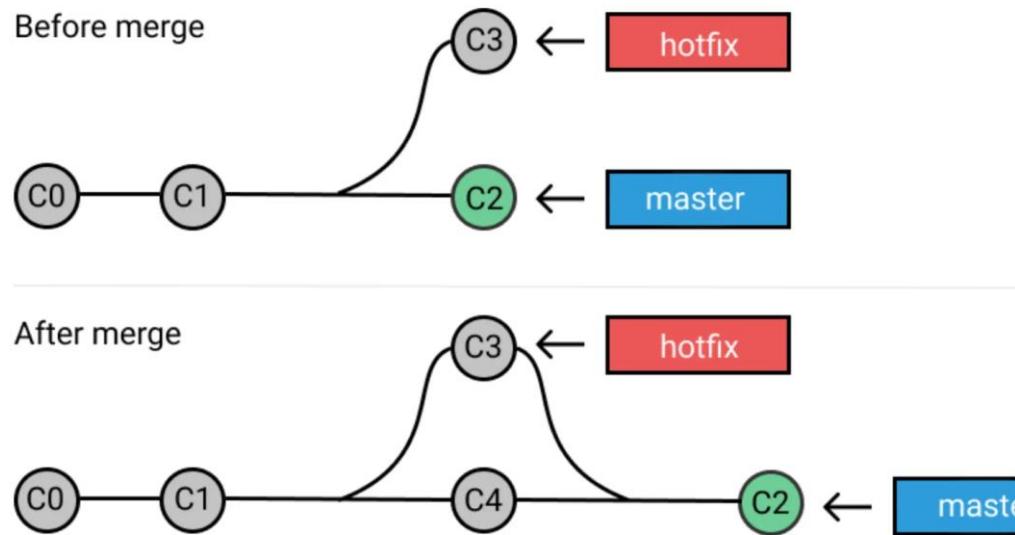
- Doing most of work in master
- Creating branches for bigger tasks
- Naming branches as seems appropriate

Companies or bigger open-source projects usually have special guidelines:

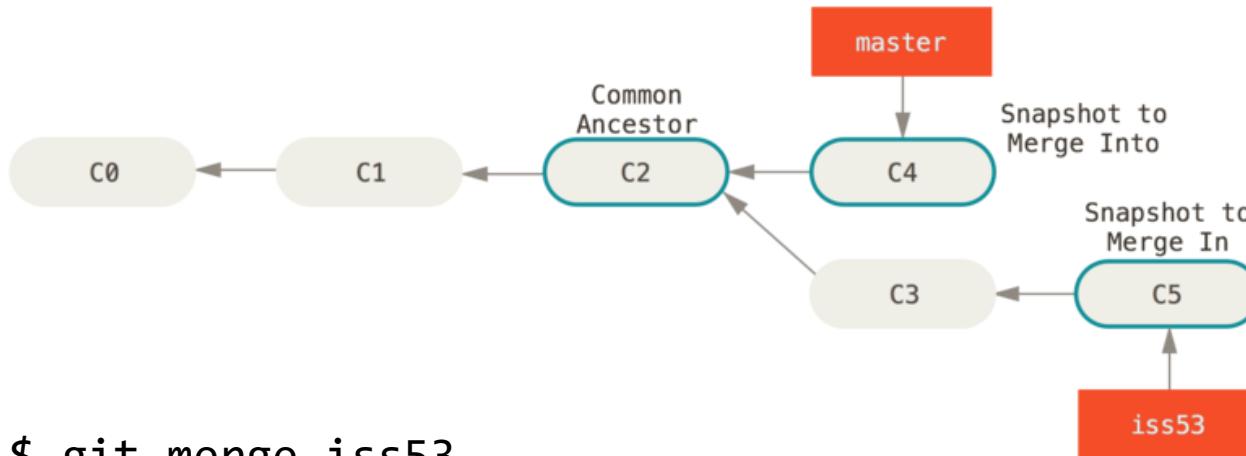
- How to name branches
- How often to create branches
- How to merge branches into master

# Merge

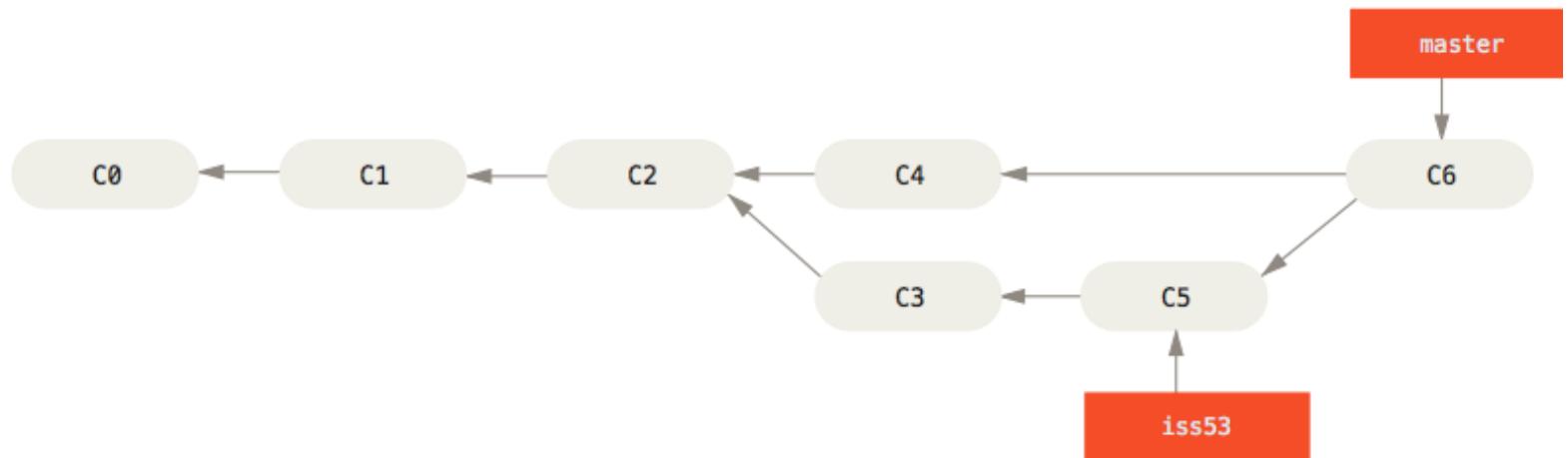
- Merge - where one branch connects with others, one revision has 2+ parents
- Git merges most changes automatically



# Merging Code



```
$ git merge iss53
Merge made by the 'recursive'
strategy.
```



# Branches merge

Types of merge strategies :

- Recursive (Fast Forward)
- Ours
- Octopus
- Resolve
- Subtree

# Branches merge strategies: Recursive

- Recursive merges **are the default** for any merges that aren't fast-forward merges
- These types of merges operate **on two different heads** using a three-way merge algorithm.
- The merge commit ends up having two parents once the merge is complete.
- The recursive strategy can detect and handle **merges that involve renaming**, but **it can't make use of detected copies**.

# Branches merge strategies: Recursive

- Recursive merges **are the default** for any merges that aren't fast-forward merges
- These types of merges operate **on two different heads** using a three-way merge algorithm.
- The merge commit ends up having two parents once the merge is complete.
- The recursive strategy can detect and handle **merges that involve renaming**, but **it can't make use of detected copies**.

# Merge Conflicts

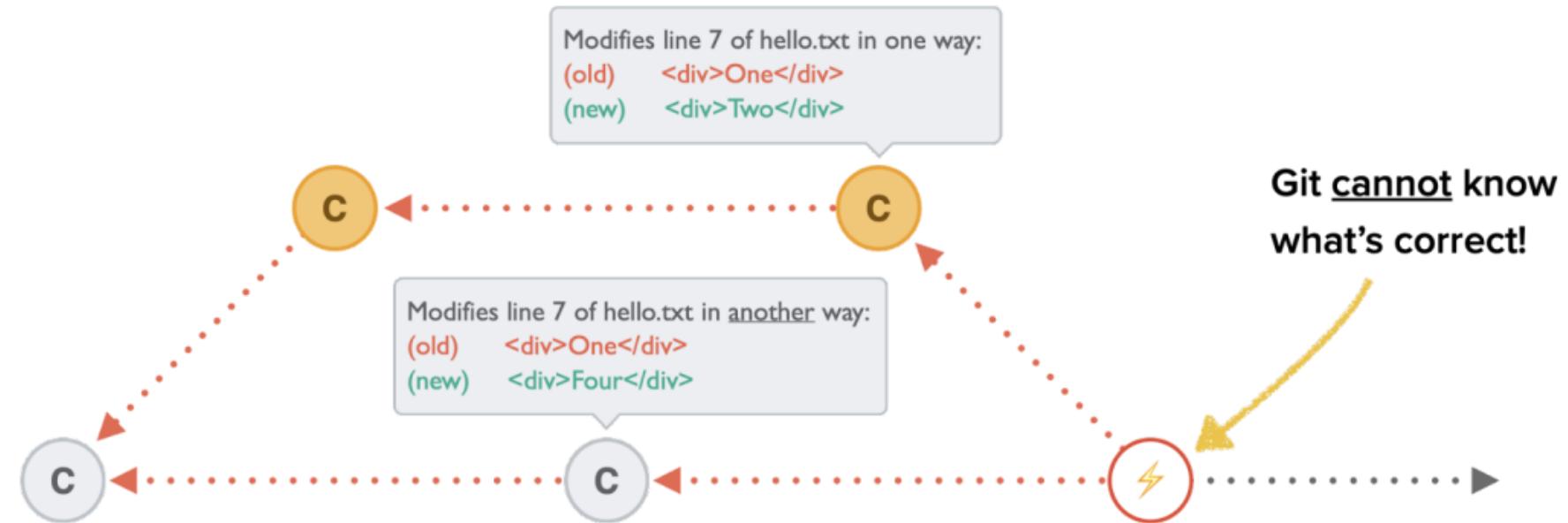
- “Merge conflict” - when Git is not sure how to combine changes in a file
- A **conflict** occurs when two **developers will change the same line of code in two different ways**, and the system is **unable to decide the changes**.

```
$ git merge new-branch
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

# Merge Conflicts: when and how

- When more than one person **changes the same line in a file** and tries to merge the change to the same branch
- When a developer **deletes a file, but another developer edits it**, and they both try to merge their changes to the same branch.
- When a **developer deletes a line, but another developer edits it**, and they both try to merge their changes to the same branch
- When a developer choose and take only the most beneficial a commit, which is the act of picking a **commit from a branch and applying it to another**
- When a developer **is rebasing a branch**, which is the process of moving a sequence of commits to a base commit

# Merge Conflicts



```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: test.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

# Merge Conflicts: sequence

1. Resolving a merge conflict
2. Find conflicts in file.
3. Decide what is the desired final state.
4. Delete the conflict markers, leaving only the desired state.
5. Add the resolved files and commit.

# Why is version control important?

- **Version control** allows us to:
  - Keep everything of importance in one place
    - Manage changes made by the team
    - Track changes in the code and other items
    - Avoid conflicting changes



# VCS software

- Automates the heavy lifting behind version management
- Enables you to:
  - Track changes in data
  - View history of data
  - Revert to a previous state of data
- Not just for code

# History of Git

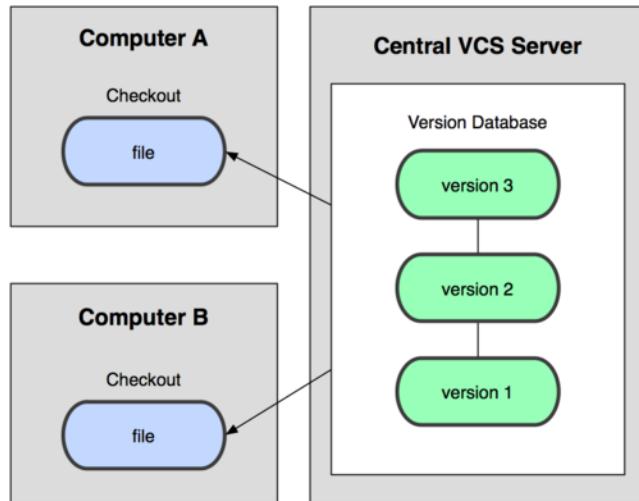
- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently

# Git – Properties and Features

- Local repository allows versioning without network connection
- Commit only adds the changes to the local repository therefore it is necessary to propagate the changes to the upstream using git push
- Files are stored as objects in a database (INDEX)
  - SHA1 fingerprints as file identifiers
- Support development a high usage of branches
- Support for applying path sets , e.g., delivered by e-mails
- Tags and Branches are marked points/states of the repository
- Suitability of the Git deployment depends on the project and model of the development

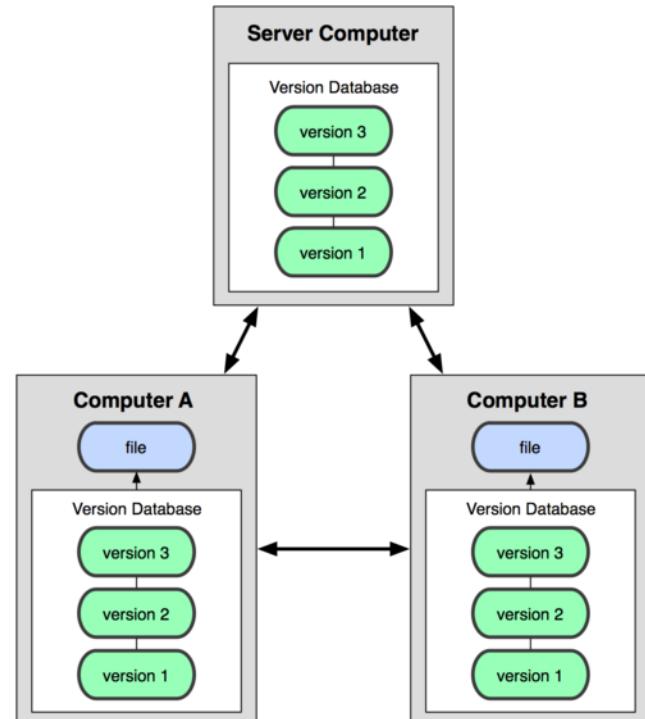
# Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model



(Git, Mercurial)

Result: Many operations are local

# VCS software: GitHub

- GitHub Documentation  
<https://help.github.com> and  
<https://www.atlassian.com/git/tutorials/>
- GitHub is the most popular open source code repository site

# VCS software: Bitbucket

- Bitbucket is a web-based hosting service for projects that use either the Git or Mercurial revision control systems
- Bitbucket <https://bitbucket.org>
- Bitbucket Documentation
  - <https://confluence.atlassian.com/display/BITBUCKET/Bitbucket+Documentation+Home>

# Additional

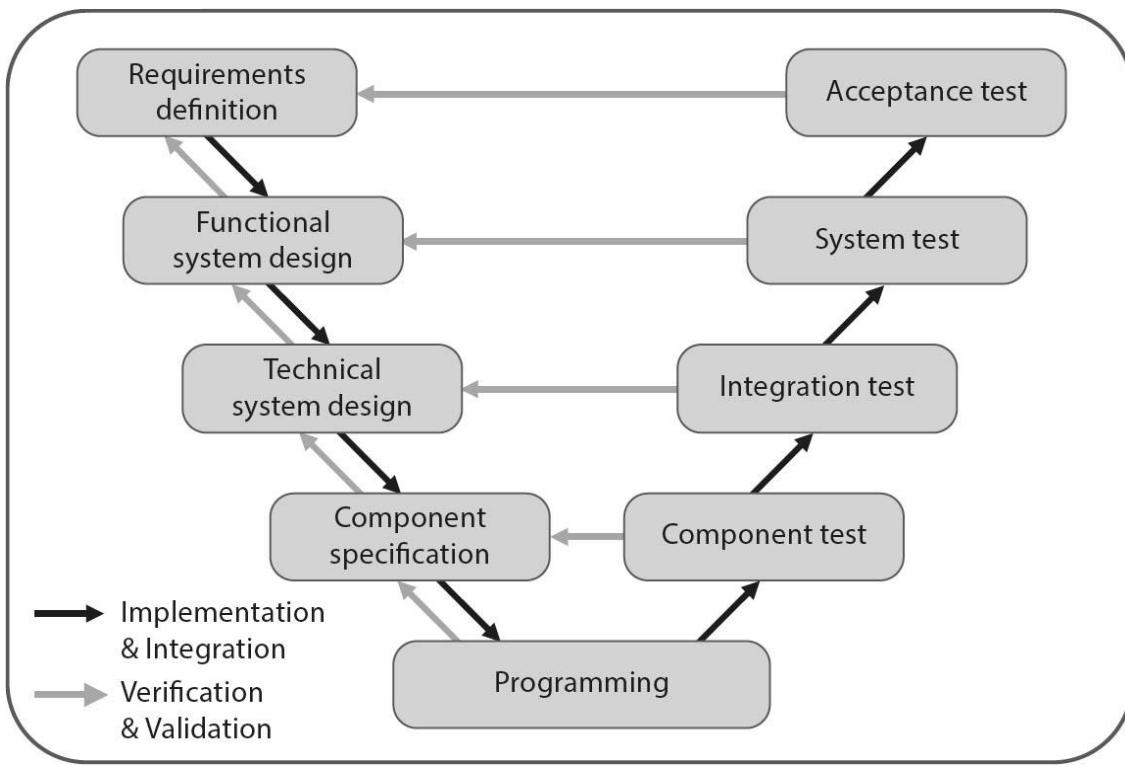
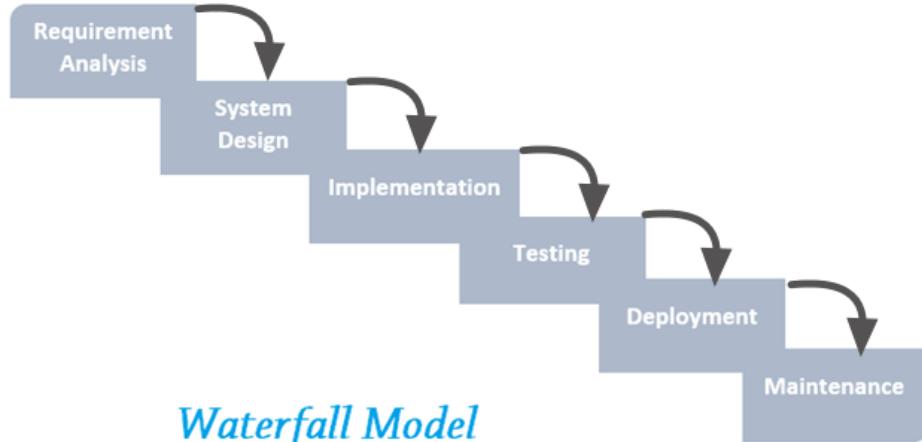
- Subversion (SVN) - <http://subversion.apache.org>
- TortoiseSVN (Windows) - <http://tortoisessvn.tigris.org>
- Concurrent Version Systems (CVS) -  
<http://savannah.nongnu.org/projects/cvs>
- Git - <http://git-scm.com>
- TortoiseGit (Windows) -  
<http://code.google.com/p/tortoisegit>
- Mercurial - <http://mercurial.selenic.com>
- RabbitVCS (Linux) - <http://www.rabbitvcs.org>

# Software System Testing

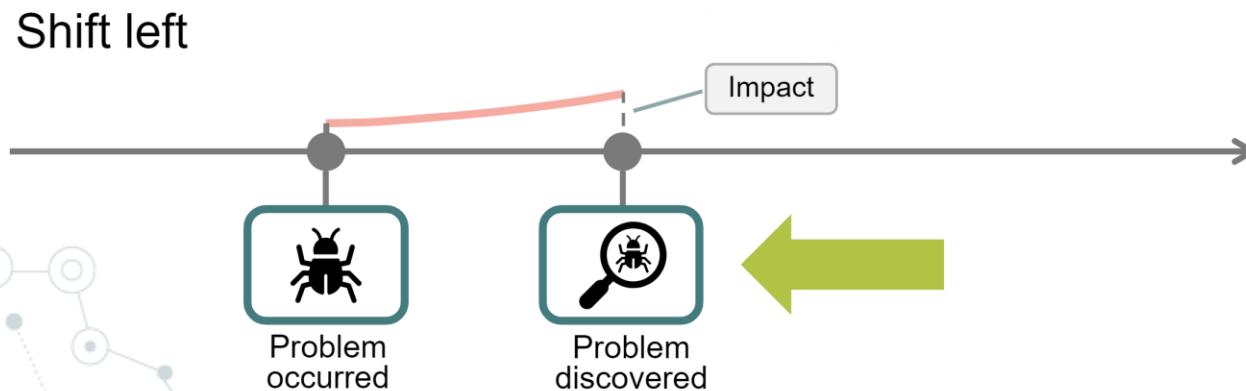
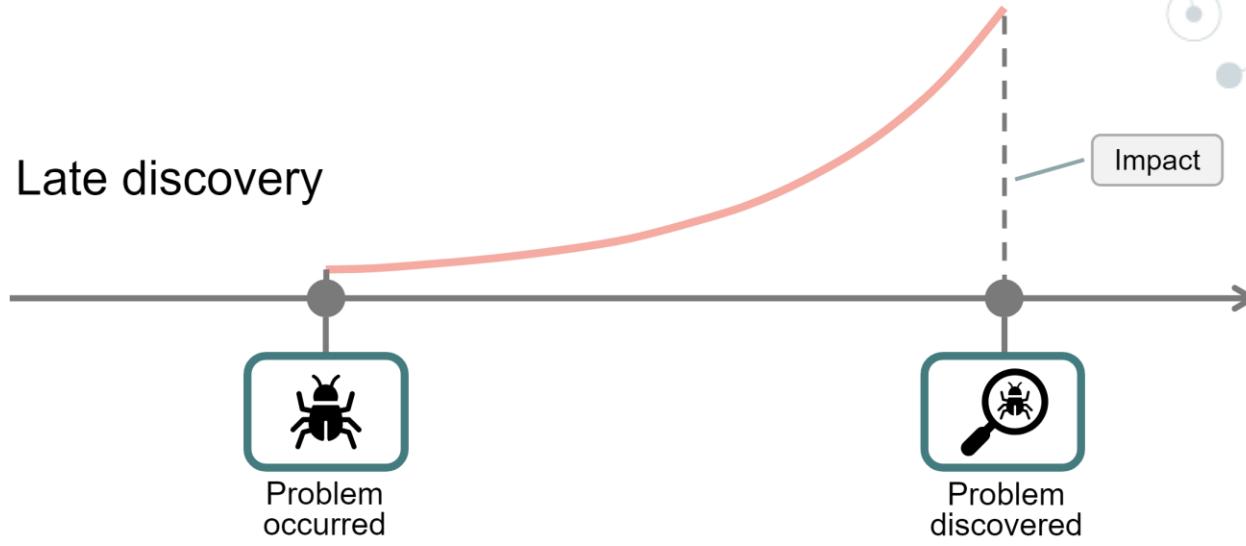
5th lecture

Assoc. Prof. Dr. Asta Slotkiene

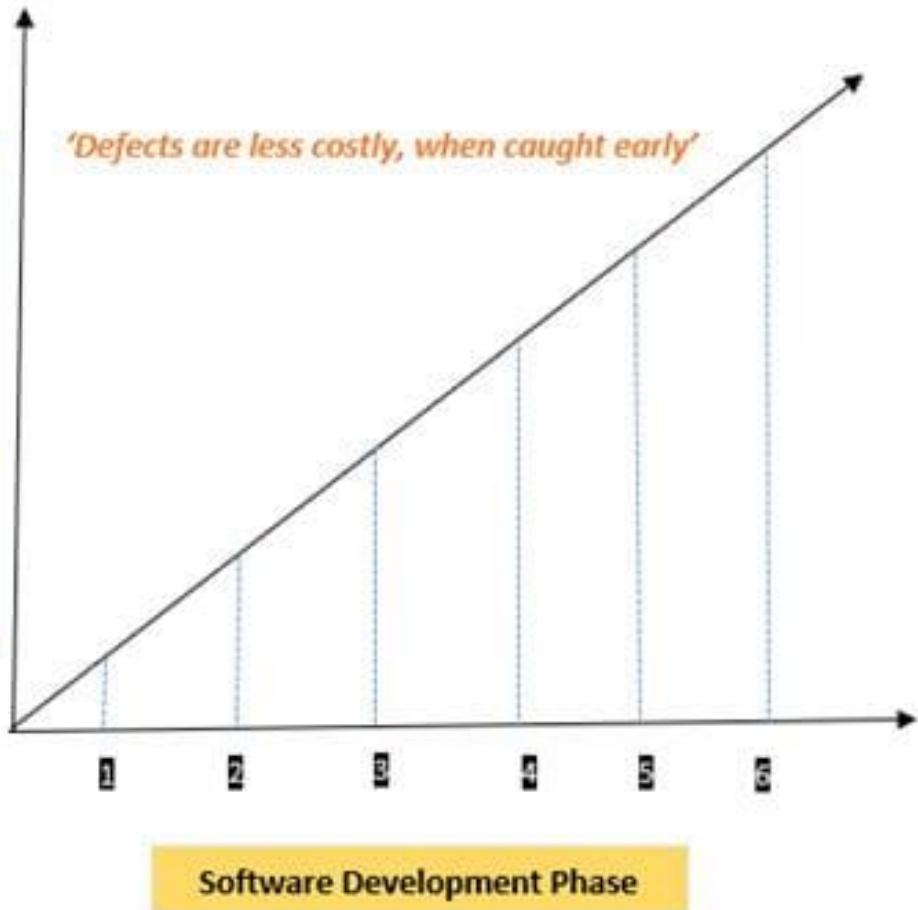
# Software Development Models



# Tradicional vs Agile Quality Practices



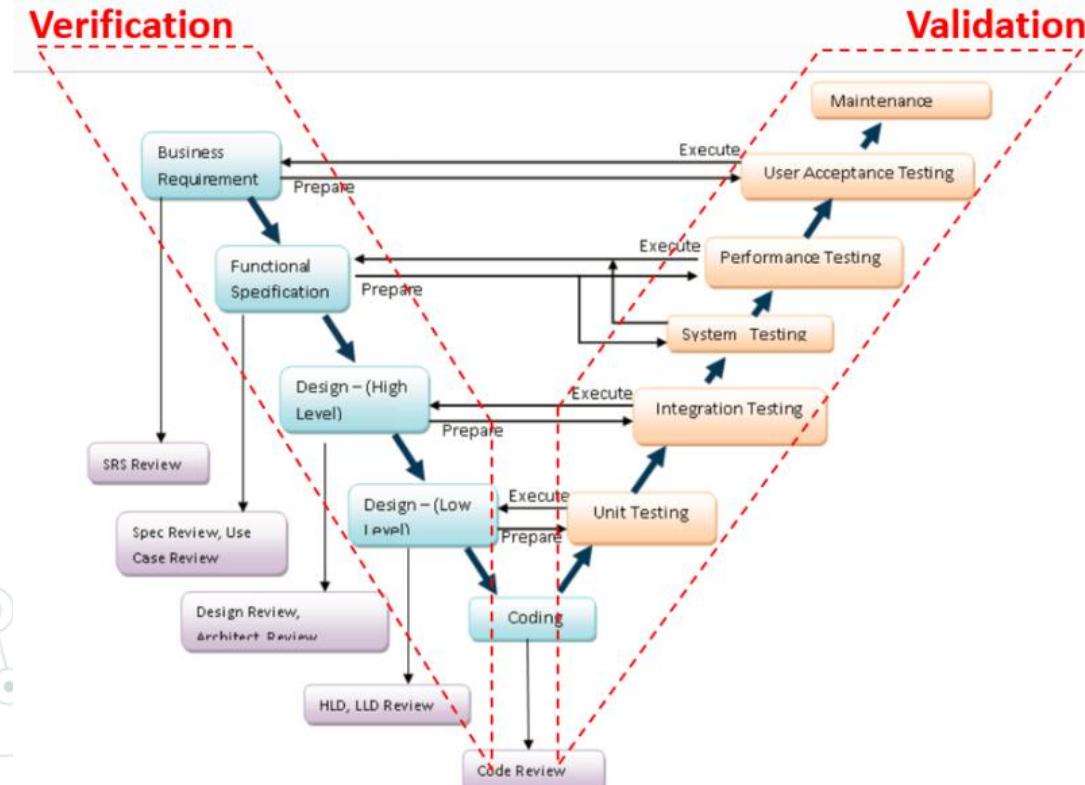
# Why Shift Left Testing?



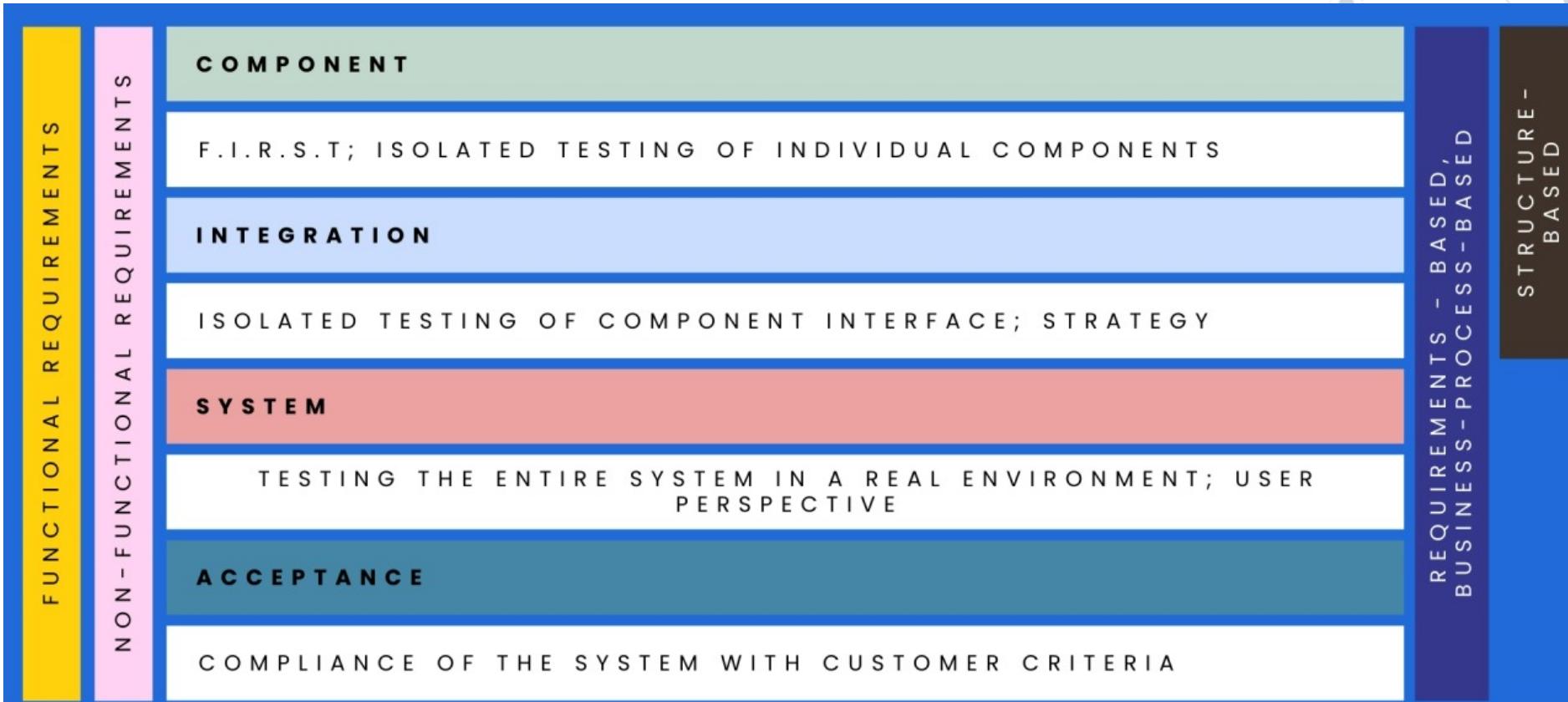
1. Requirement Stage
2. Design Stage
3. Coding Stage
4. Testing Stage
5. Production Deployment Stage
6. Post Go-Live Stage

# Software development models

- Testing needs to begin as early as possible in the life cycle.
- Testing can be **integrated** into each phase of the life cycle.
- Within the V-model, **validation** testing takes place especially during
  - the early stages, i.e. **reviewing the user requirements**
  - and late in the life cycle, i.e. **during user acceptance testing**



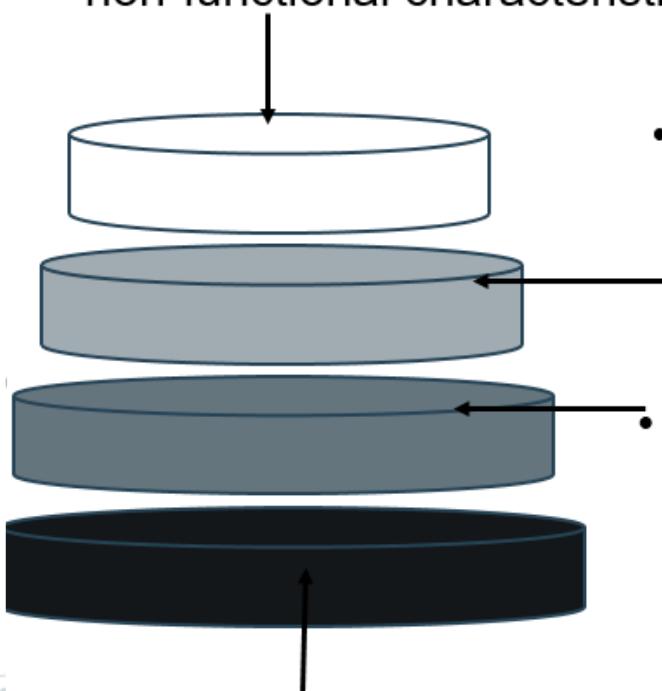
# Testing Levels



# Testing Levels

- **Acceptance**

Is the responsibility of the customer – in general. The goal is to gain confidence in the system; especially in its non-functional characteristics



- **System**

The behavior of the whole product(system) as defined by the scope of the project

- **Integration**

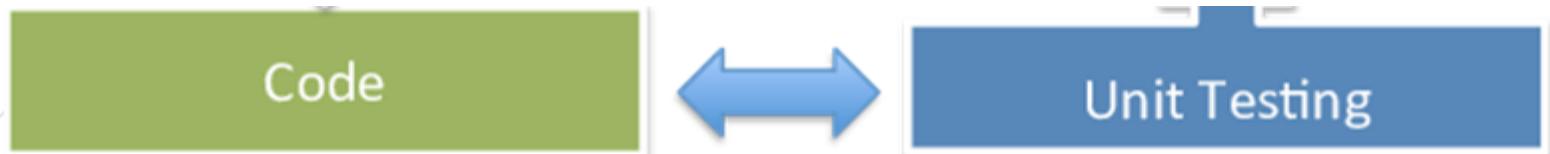
Interface between components; interactions with other systems (OS, HW, ...)

- **Unit**

Any module, program, object separately testable

# Testing Level: Unit/Component Testing

- ◎ **Unit/Component tests** ensure that each individual component fulfills its specified requirements.
- ◎ A unit is the smallest testable part of an application.
- ◎ Who: Developers
- ◎ How:
  - White-Box Testing Method
  - UT frameworks (e.g., jUnit), drivers, stubs, and mock/fake objects are used



# Testing Level: Integration Testing

- ◎ Who: Either Developers themselves or independent Testers
- ◎ Goal: ensure that groups of components interact as specified by the technical design
- ◎ More than one (tested) unit
  - Detecting defects: On the interfaces of units or Communication between units
- ◎ How:
  - Any of Black Box, White Box, and Gray Box Testing methods can be used
  - Test drivers and test stubs are used to assist in Integration Testing.

Design



Integration Testing

# Testing Level: System Testing

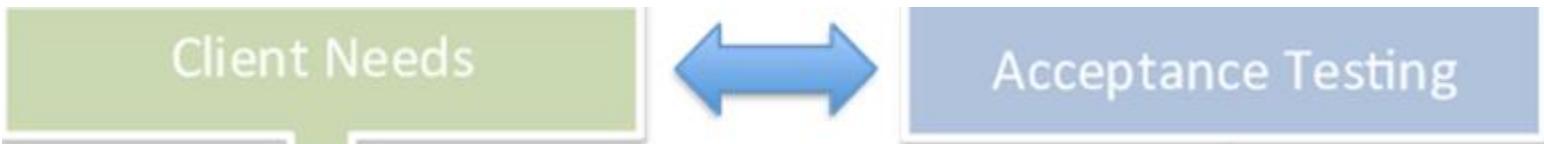
- ◎ Who: Normally, independent Testers perform System Testing
- ◎ Goal: ensure that the system as a whole functions according to its specified requirements
- ◎ Types: Smoke Testing, Functional Testing, Usability Testing, Security Testing , Performance Testing, Regression Testing ,Compliance Testing and etc.
- ◎ How:
  - Usually, Black Box Testing method is used.

Requirements

System Testing

# Testing Level: Acceptance Testing

- ◎ Who: Product Management, Sales, Customer Support, Customers
- ◎ Goal: checks that the system as a whole adheres to the contractually agreed customer and end-user criteria.
- ◎ How:
  - Usually, Black Box Testing method is used;
  - often the testing is done ad-hoc and non-scripted

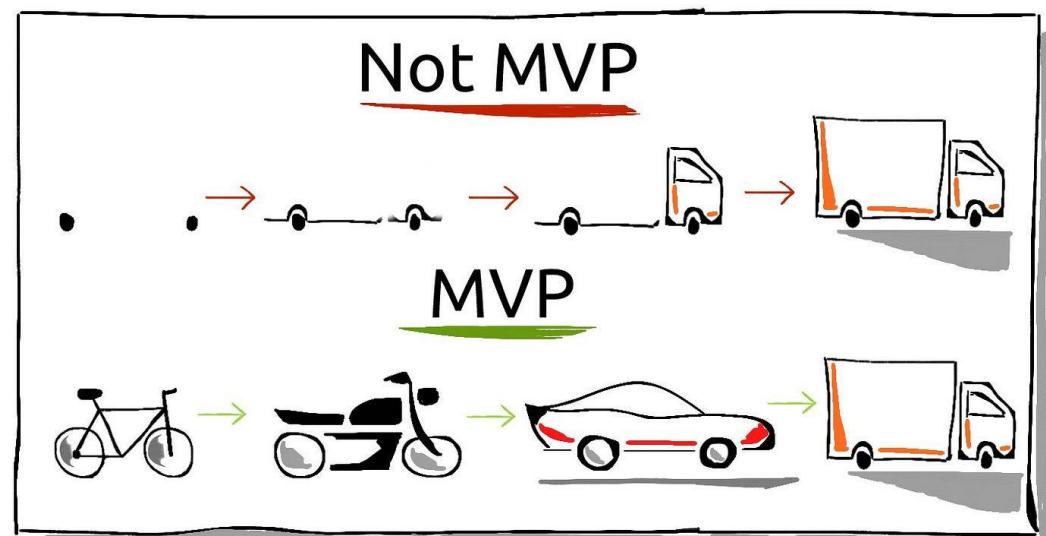
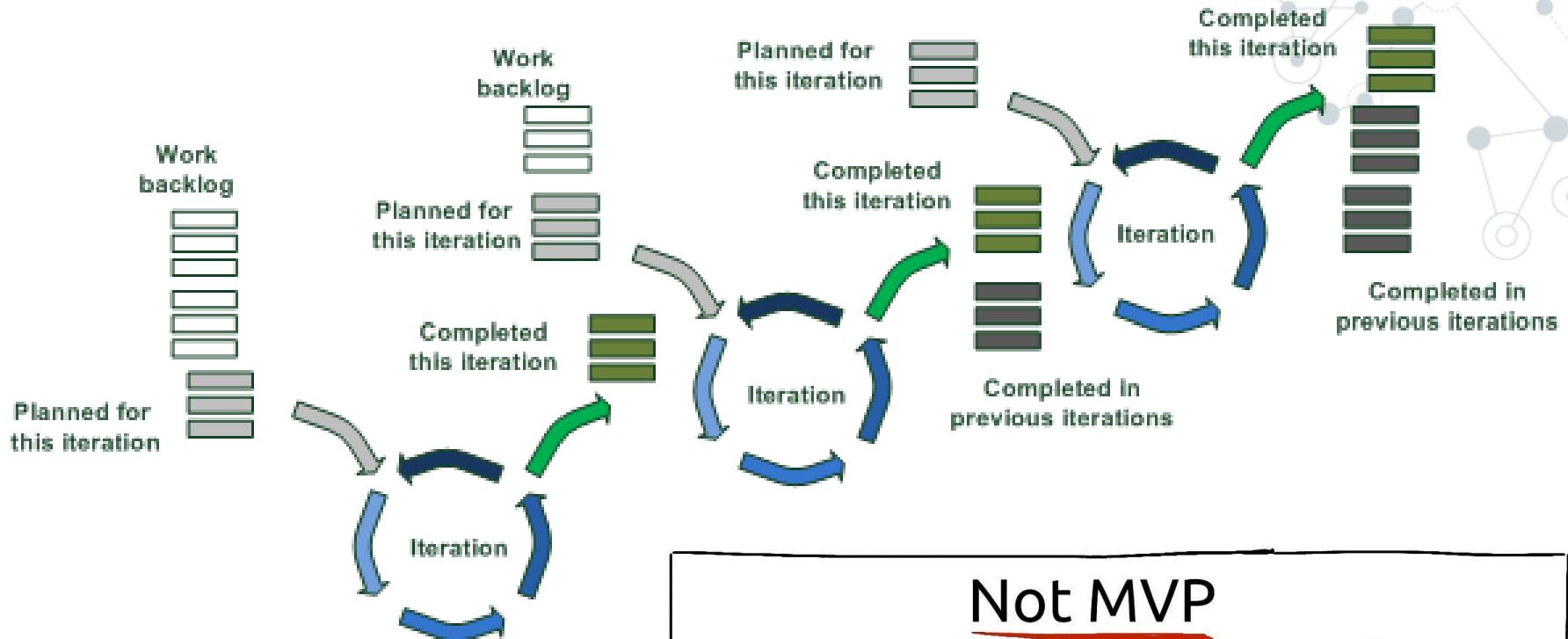


**also called: Behavior-driven testing (BDD)**

# Testing Levels

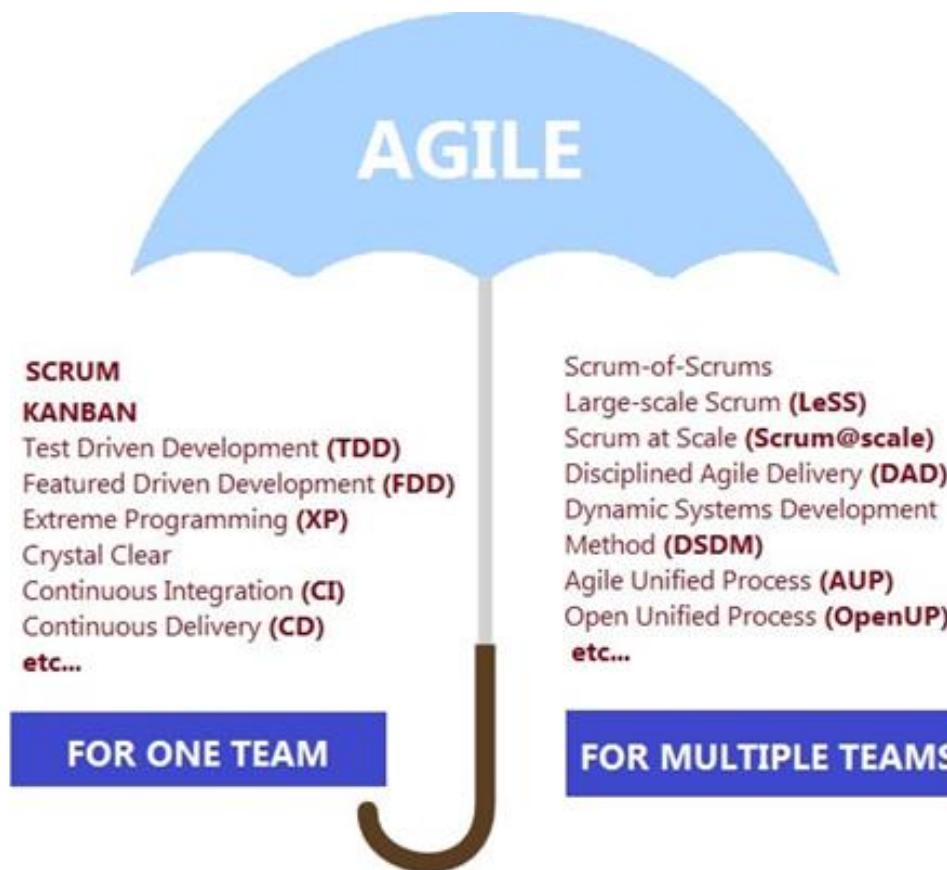
- ◎ For **each test level**, please note:
  - The **generic objectives**
  - The **test basis** (docs/products used to derive test cases)
  - The **test objects** (what is being tested)
  - Typical **defects** and **failures** to be found
  - Specific **approaches**

# Iterative and Incremental Development Models



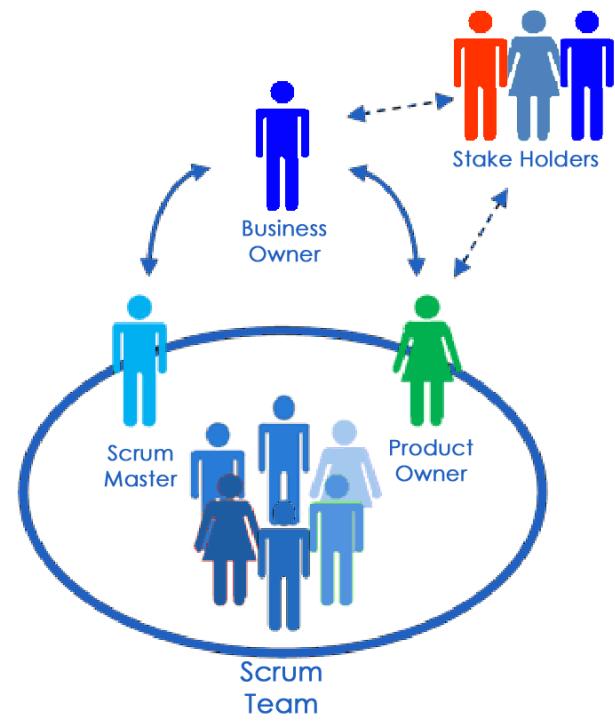
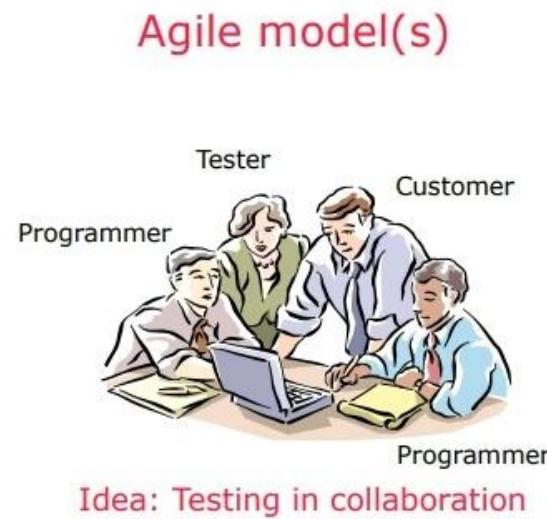
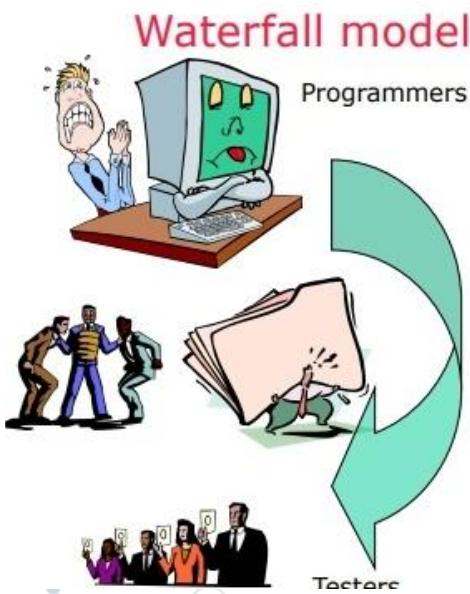
# Iterative and Incremental Development Models

- All forms of agile software development are iterative-incremental development models.
- The best-known agile models are:



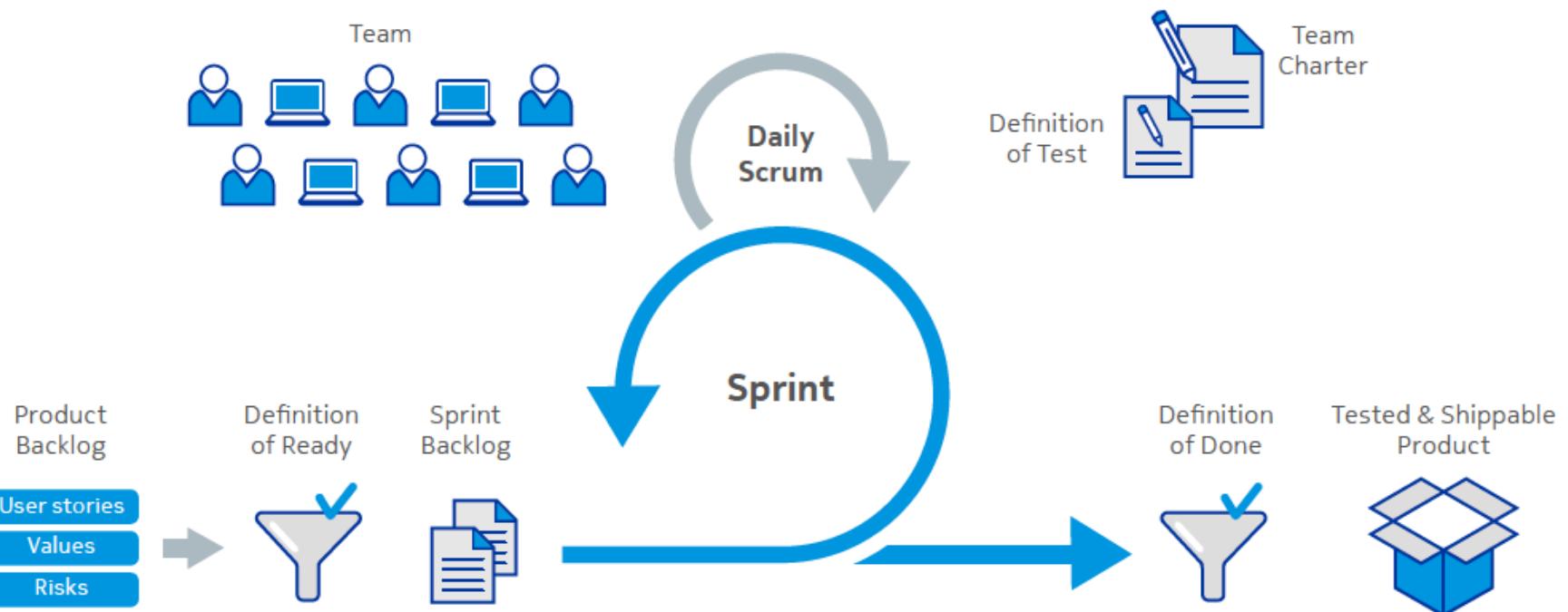
# Test Planning and Test Management: Scrum

- Scrum Team has no dedicated test manager and distributes the responsibilities associated with this role within the team



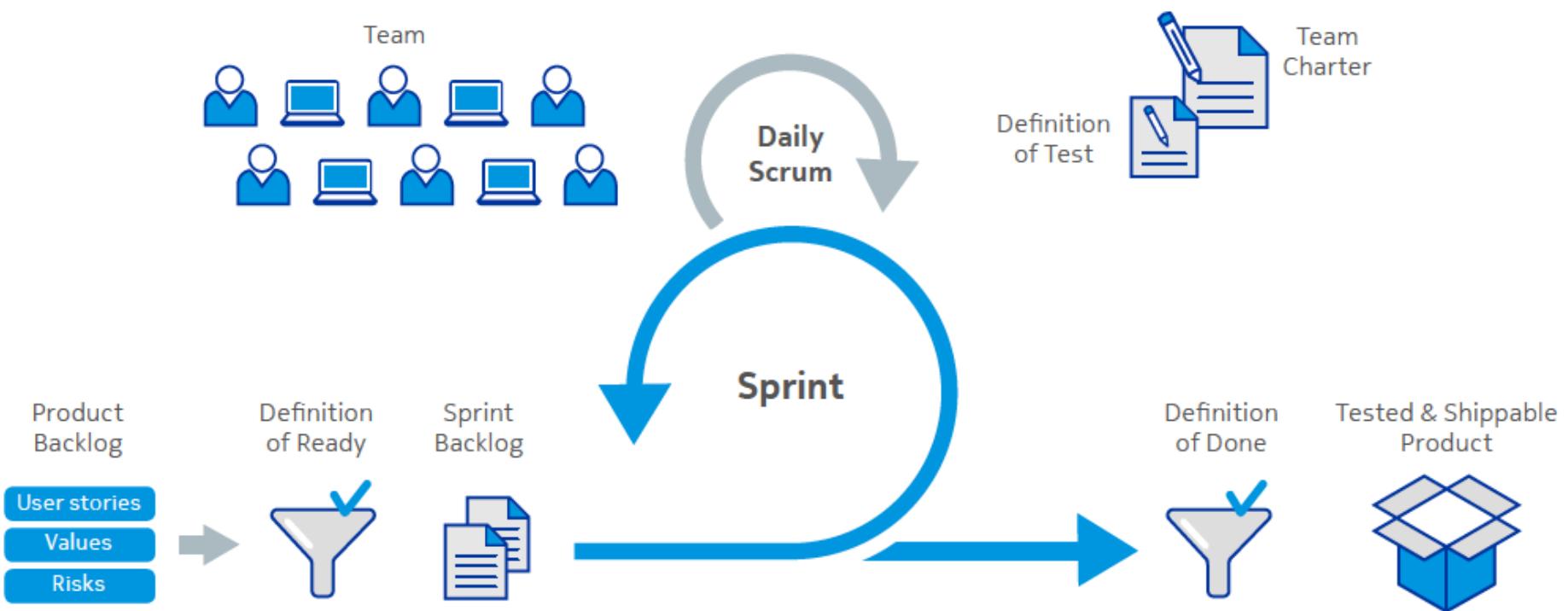
# Test Planning and Test Management: Scrum

- Testing tasks are scheduled explicitly as tasks within a Sprint or implicitly as part of the DoR and DoD criteria of other tasks.



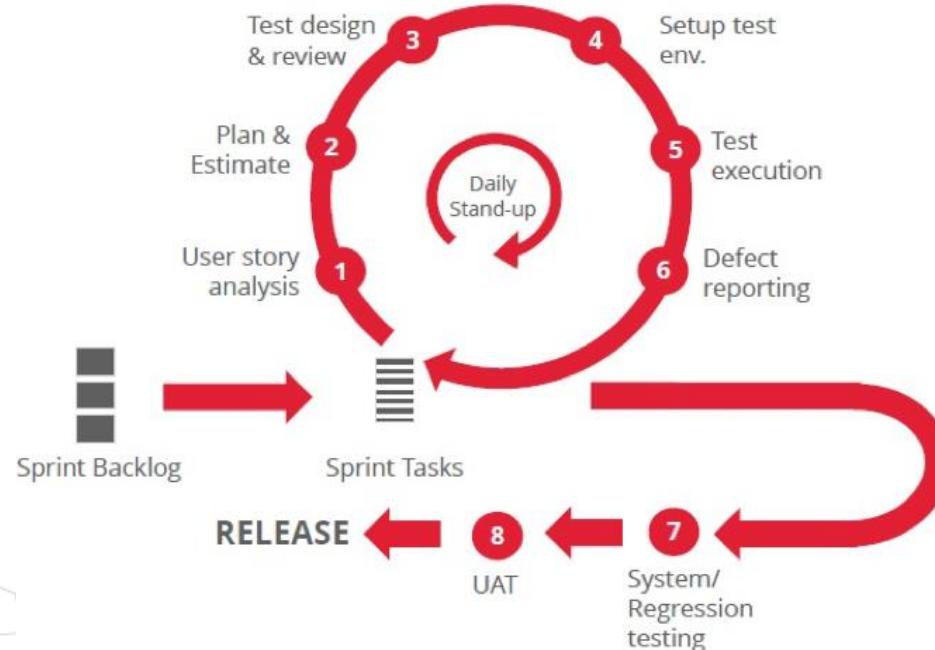
# Test Planning and Test Management: Scrum

- ◎ Team requires a member with dedicated testing expertise.
- ◎ **Responsible for designing appropriate risk-oriented tests and implementing those in every Sprint.**

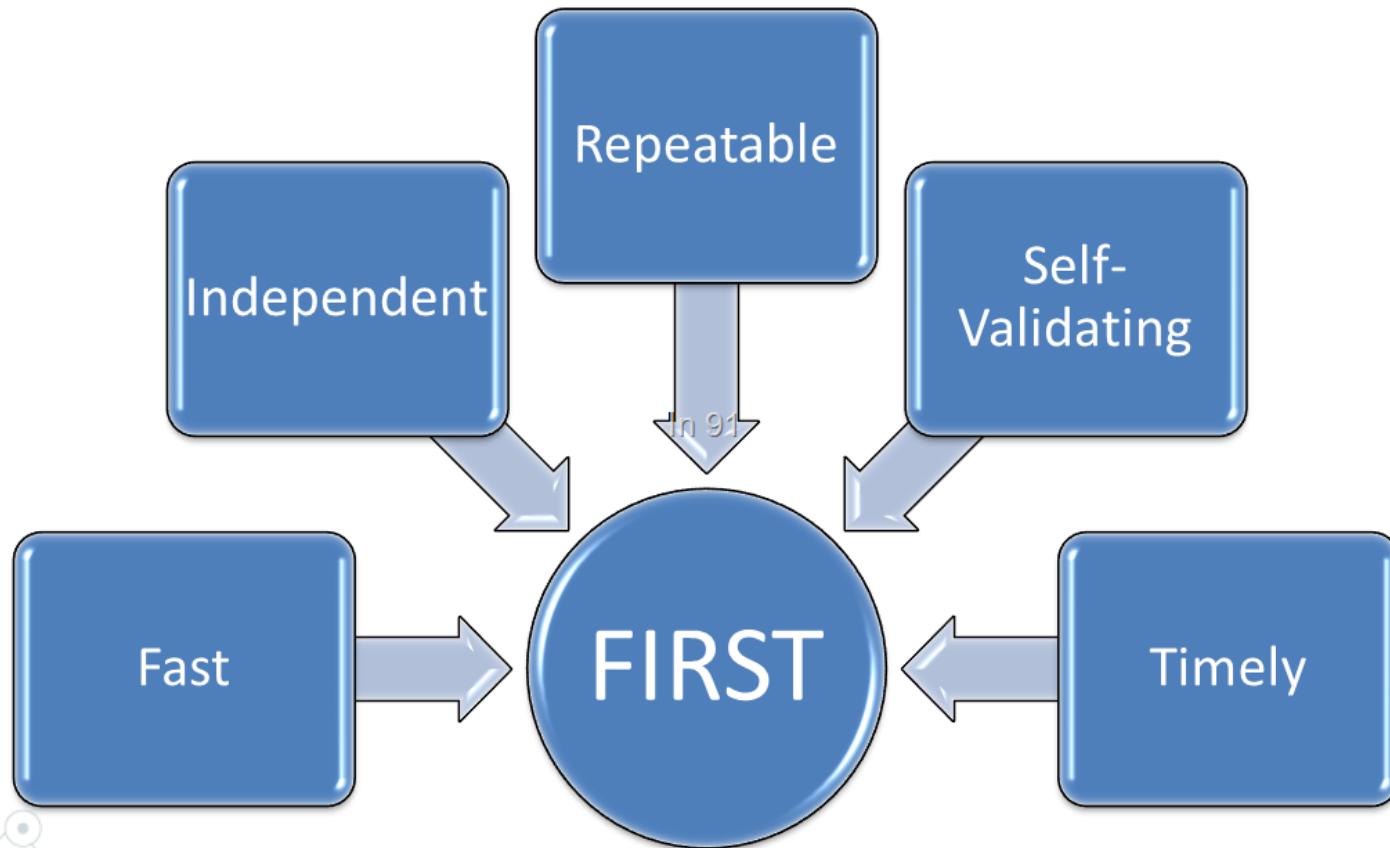


# Challenges of Testing in Agile Development

- Requirements change all the time
- Specification documents are never final
- Code is never ‘finished’, never ‘ready for testing’
- Limited time to test
- Need for **regression testing** in each increment

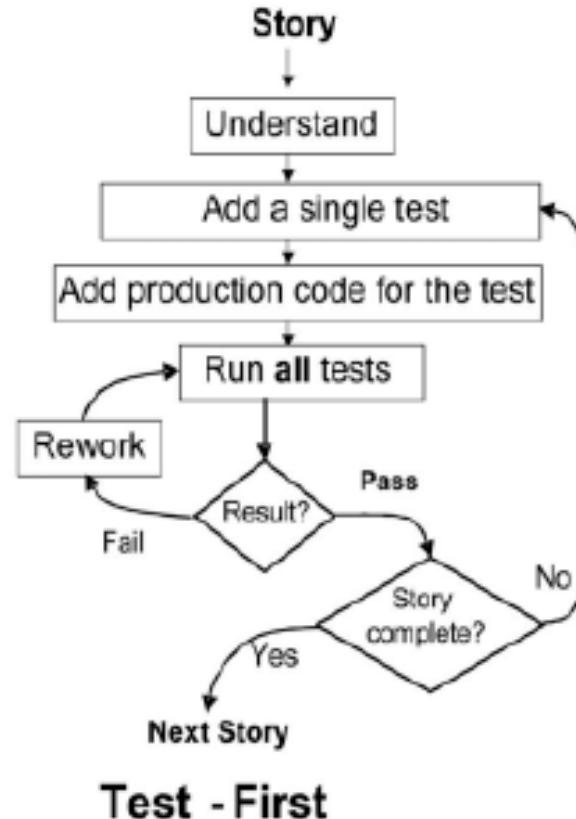
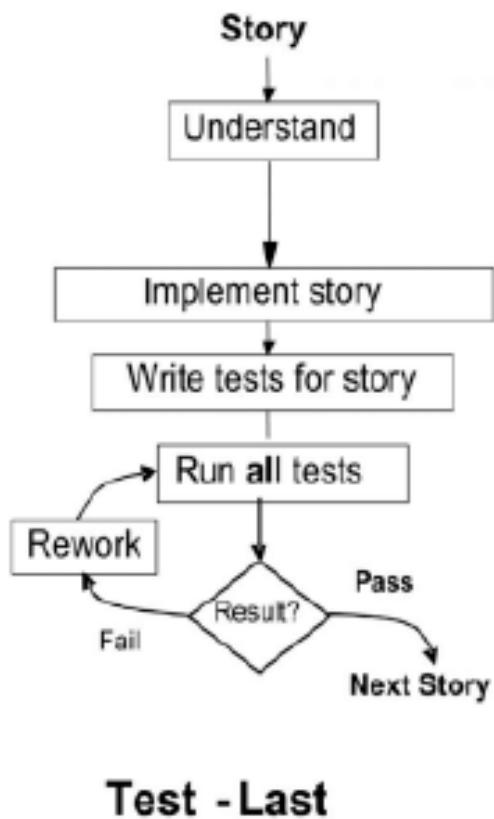


# F.I.R.S.T. test principles



# Test-first versus test-last development

- Test First means considering, which tests will be necessary to show that the software actually fulfills any new specifications before any changes in the code itself are performed.
  - Before you alter your code, write an automated test that fails. (Test Driven Development).

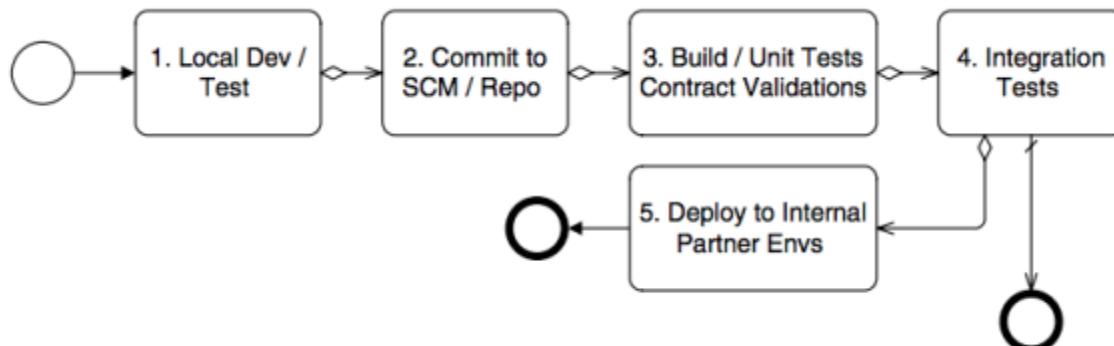


# Test-driven development (TDD)

- ◎ Test-driven development - a software development technique in which the test cases are developed, and often automated, and then **the software is developed incrementally to pass those test cases**.
  
- ◎ ISTQB Glossary
  - <https://glossary.istqb.org/search/>

# Unit testing and Scrum

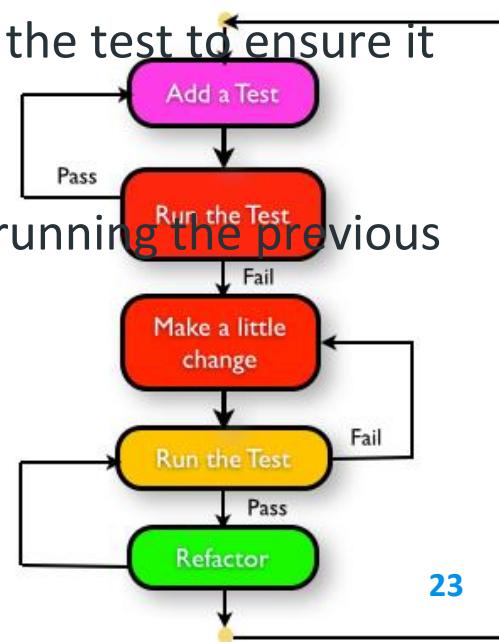
- ◎ Scrum itself doesn't demand the use of **Test First** and works just as well with traditional unit tests.
- ◎ However, the team can benefit significantly on a technical level and through the acceleration of the feedback loop it provides:
  - **For every change in the code, the programmer receives immediate feedback about its success or failure.**



# Unit testing and Scrum:TDD

◎ The process for test-driven development is:

1. Add a test that captures the programmer's concept of the desired functioning of a small piece of code
2. Run the test, which should fail since the code doesn't exist
3. Write the code and run the test in a tight loop until the test passes
4. Refactor the code after the test is passed, re-running the test to ensure it continues to pass against the refactored code
5. Repeat this process for the next small piece of code, running the previous tests as well as the added tests



TDD cycle: 1 step-> *Add a test that captures the programmer's concept of the desired functioning of a small piece of code*

**The user entered "I", the program must return 1.**

```
public class RomanNumberConverterTest {  
  
    @Test  
  
    void shouldUnderstandSymbolI() {  
  
        RomanNumeralConverter roman = new RomanNumeralConverter();  
  
        int number = roman.convert("I");  
  
        assertThat(number).isEqualTo(1);  
  
    }  
}
```

TDD cycle: 2 step->*Run the test, which should fail since the code doesn't exist*

**The user entered "I", the program must return 1.**

```
public class RomanNumberConverterTest {  
  
    @Test  
  
    void shouldUnderstandSymbolI() {  
  
        RomanNumeralConverter roman = new RomanNumeralConverter();  
  
        //fail  
  
        int number = roman.convert("I");  
  
        assertThat(number).isEqualTo(1);  
    }  
}
```

TDD cycle: 3 step-> *write the code and run the test in a tight loop until the test passes*

**The user entered "I", the program must return 1.**

```
public class RomanNumeralConverter {  
    public int convert(String numberInRoman)  
{  
    return 0; //fail, expected 1  
} }
```

TDD cycle: 4 steps->refactor the code after the test is passed, re-running the test to ensure it continues to pass against the refactored code

The user entered "I", the program must return 1.

```
public class RomanNumeralConverter {  
  
    public int convert(String numberInRoman) {  
  
        if (numberInRoman.equals("I"))  
  
            return 1;  
        return 0;  
    }  
}
```

TDD cycle: 5 step ->*repeat this process for the next small piece of code, running the previous tests as well as the added tests*

**The user entered „V”, the program must return 5.**

```
@Test
```

```
void shouldUnderstandSymbolV() {  
    RomanNumeralConverter roman = new RomanNumeralConverter();  
    int number = roman.convert("V");  
    assertThat(number).isEqualTo(5);  
}
```

## TDD cycle: step: from fail to pass

**The input „V”, the program must return 5.**

```
public class RomanNumeralConverter {  
  
    public int convert(String numberInRoman) {  
  
        if (numberInRoman.equals("I"))    return 1;  
        if (numberInRoman.equals("V"))    return 5;  
  
        return 0;  
    } }
```

## Why need fail test?

1. It verifies the test works, including any testing harnesses
2. Demonstrates how the **system will behave if the code is incorrect.**

# TDD process activities

- ◎ Start by **identifying the increment of functionality** that is required.
  - This should normally be small and implementable in a few lines of code.
- ◎ Write a test for this functionality and **implement this as an automated test**.
- ◎ Run the test, **along with all other tests** that have been implemented.
  - Initially, you have not implemented the functionality so the new test will fail.
- ◎ Implement the functionality and **re-run the test**.
- ◎ Once all tests run successfully, you move on to implementing the **next chunk of functionality**.

# TDD Limitations

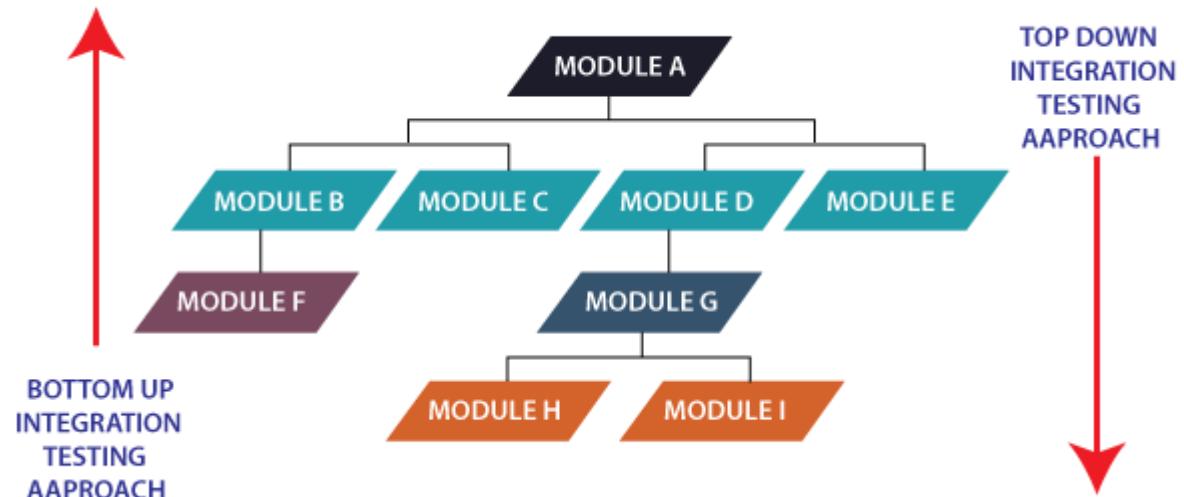
- ◎ Difficult in Some Situations
  - **GUIs, Relational Databases, Web Service**
  - Requires Mock objects
- ◎ TDD does not often include an upfront design
  - Focus is on implementation and **less on the logical structure**
- ◎ Difficult to write test cases for hard-to-test code
  - **Requires a higher level of experience from programmers**
- ◎ **TDD merge distinct phases of software development**  
design, code and test

# What need to do Scrum Master in the testing process?

- Make sure that all team members write similarly structured tests.
  - Tests have to be stored centrally on the team's development network in a file system with an agreed structure.
- Ensure that **test coverage** is measured reliably and its Sprint-by-Sprint development analyzed regularly.
  - The necessary coverage limits should be established in accordance with the risk analysis performed for each unit.
- Ensure **static code analysis**
- Ensure that regular program code reviews take place.
- Ensure that the test code is **reviewed** regularly.
- Define Unit testing **guidelines**

# Integration Testing

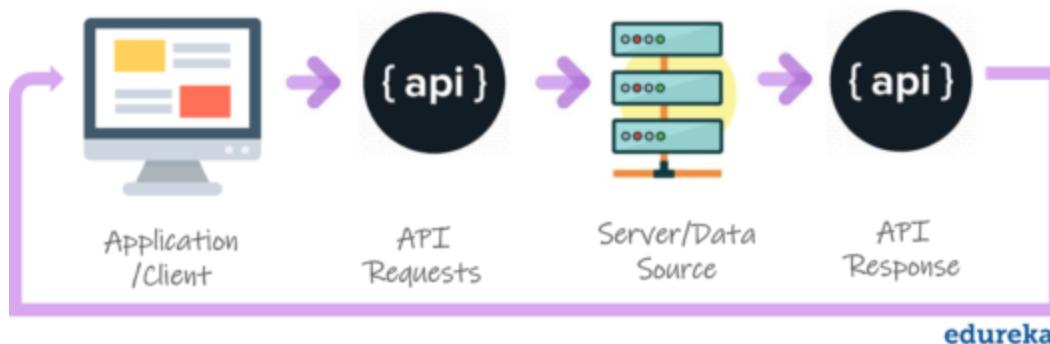
- Integration tests are designed to discover **potential defects in the interaction** of the individual components and their interfaces.
- Even if just a single component is altered, **always should re-run not only its unit test, but also all the integration tests** that cover components that it interacts with.



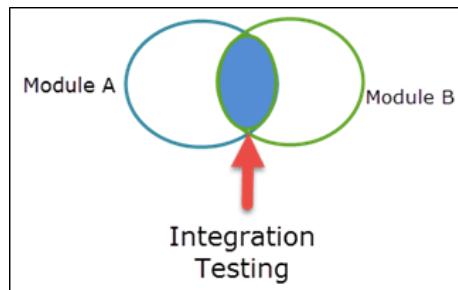
# Integration Testing includes:

## ◎ Dependencies:

- **Explicit.** If a component A calls a component B directly via its API.



- **Implicit.** If multiple components share a single resource (global variable, same database, etc.)



# Integration Testing in the Scrum: CI/CD

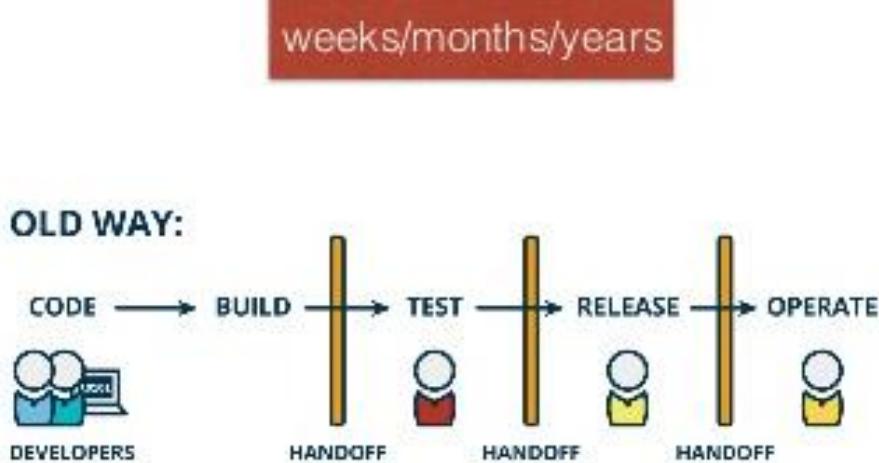
## ◎ The CI Process:

- Code repository (check in as soon as possible)
- Automated integration runs
- Compilation
- Static code analysis
- Deployment to the test environment
- Initialization Unit Testing
- Integration testing
- System testing
- Feedback and dashboard



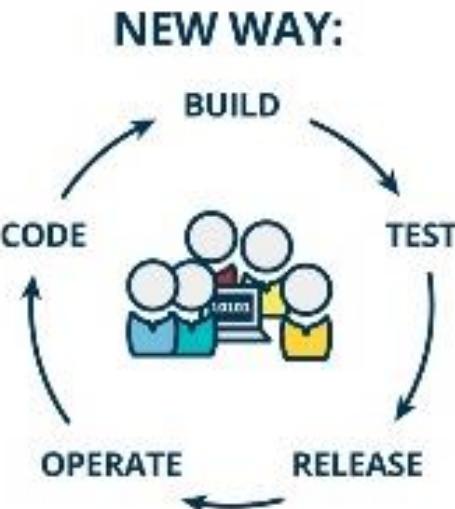
# Continuous Integration

- Each check-in is then verified by an automated build,
  - allowing teams to **detect problems early**.



Cyclic delivery  
of earlier times

hours/days!



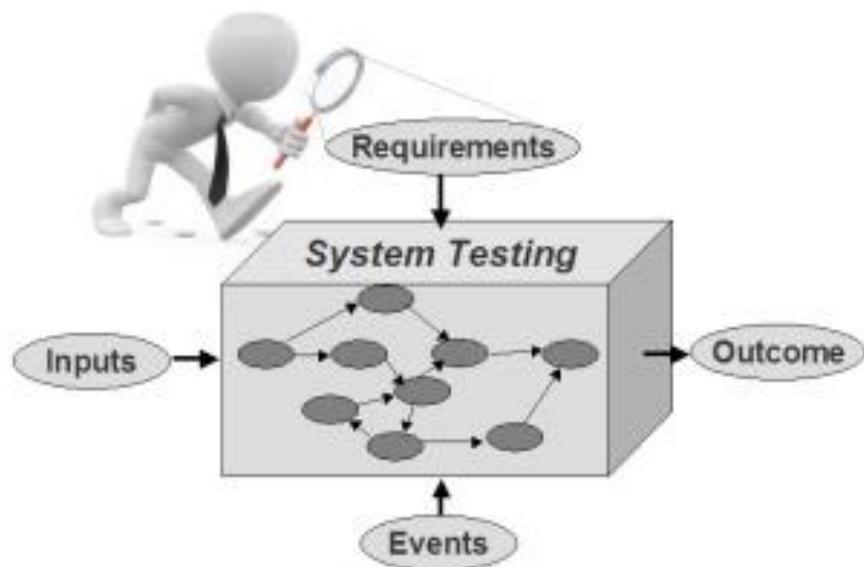
continuous delivery  
of modern times

# Integration Test Management in Scrum

- ◎ Ensure that a sufficient number of appropriate test cases are written and run
- ◎ Can and should be designed using **Test First principles**
- ◎ Take integration testing effort into account during Sprint Planning
- ◎ Check whether **additional integration-related code analysis is possible**
- ◎ Attention to sorting automated integration tests into batches and the continuous optimization of the speed of the CI process.

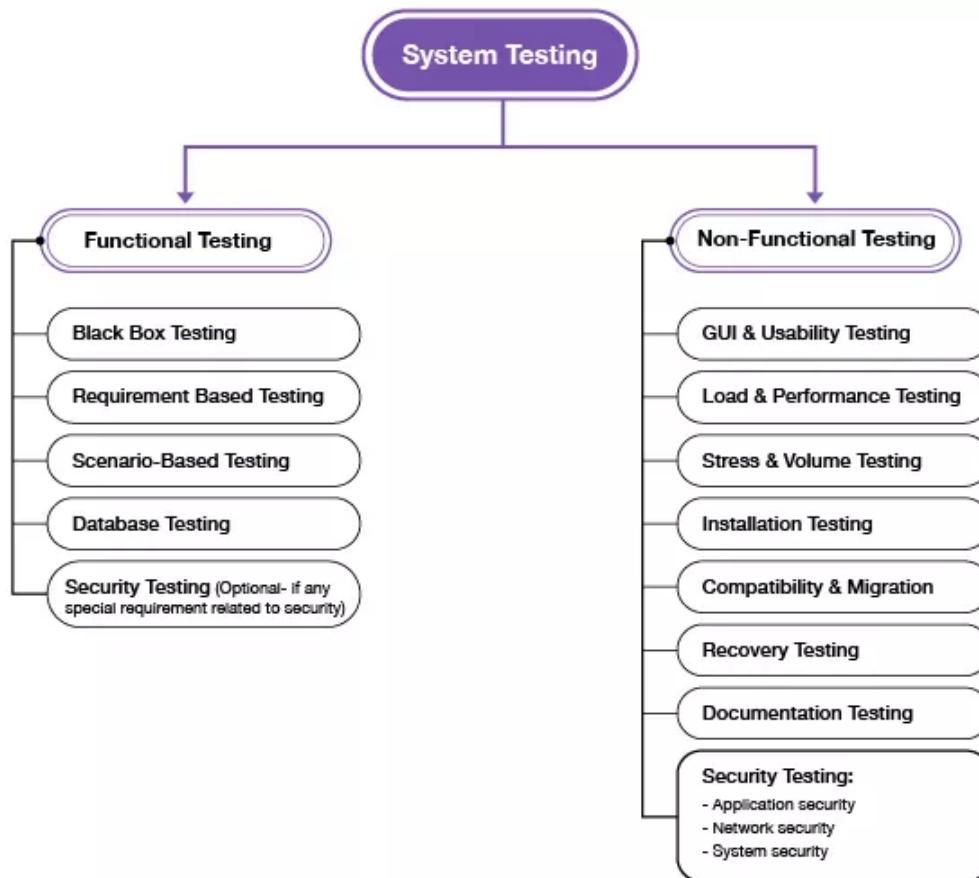
# System Testing in the Scrum

- ◎ System tests check that the system works from the user's point of view using the customer's interfaces.
- ◎ It can be derived directly from the requirements and acceptance criteria listed in the Product Backlog or from the corresponding use case description.
- ◎ The dialog with the Product Owner should be used to clarify when and whether requirements listed in the Backlog need to be more precisely defined and whether additional requirements need to be drafted and added to the Backlog.



# System testing in the Scrum

- The effort involved in system testing **does not scale according to the number of system test cases**, but rather with the number of **different test environments and quality criteria** that need to be checked for functionality.



# Behavior driven development (BDD)

- ◎ Behavior driven development - a **collaborative** approach to development in **which the team focuses on delivering the expected behavior of a component or system for the customer**, which forms the basis for testing.
  
- ◎ ISTQB Glossary
  - <https://glossary.istqb.org/search/>

# Why is a focus on BEHAVIOUR so important?

**Users** usually don't care about **technical implementation**

they care about **BEHAVIOUR** of the software

*“...our clients **don't value the code as such**;  
**they value the things that the code does for them.**”*

*Michael Bolton*

# Who should write acceptance tests?

## Answer: client???

- Probably manual tests: OHHH

Step	Req	Pass Conditions	Pass?
1. Select the Biology program.	UIR-2	System displays biology classes w/ first class BIOLOGY 1150, Section 01, Title GENERAL BIOLOGY, Instructor Block, Anna, Filled/Seats 52/53, Class# 1311, Credits 5, Meets BOE 0221 MWF 8:00-8:52	P / F
2. Double-click on Class# 1330	UIR-1	System includes Class# 1330 in schedule at bottom	P / F
3. Scroll down to Class# 1331 (BIOLOGY 1650, Section 01)	UIR-9	System displays Class# 1331 with a pink background	P / F
4. Show the list	UIR-9	All sections listed between #1311 and #1331 have a white background	P / F
5. Select the GENENG program.	UIR-2	System displays general engineering courses	P / F

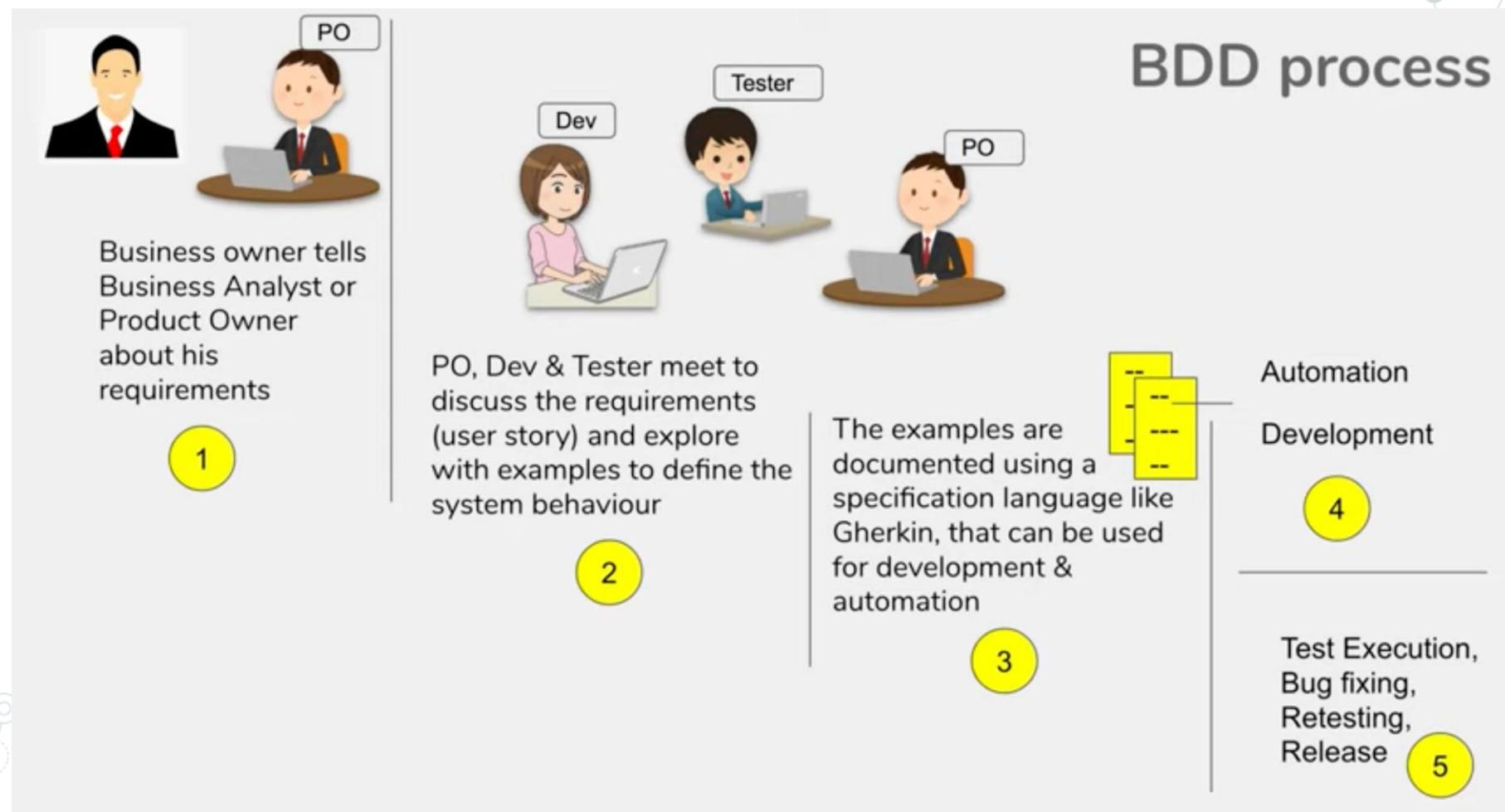
# Who should write acceptance tests?

I client write test cases,  
Not much help to development team

Step	Req	Pass Conditions	Pass?
1. Select the Biology program.	UIR-2	System displays biology classes w/ first class BIOLOGY 1150, Section 01, Title GENERAL BIOLOGY, Instructor Block, Anna, Filled/Seats 52/53, Class# 1311, Credits 5, Meets BOE 0221 MWF 8:00-8:52	P / F
2. Double-click on Class# 1330	UIR-1	System includes Class# 1330 in schedule at beginning of term	P / F
3. Scroll down to Class# 1331 (BIOLOGY 1650, Section 01)	UIR-9	System displays Class# 1331 with a pink background	P / F
4. Show the list	UIR-9	All sections listed between #1311 and #1331 have a white background	P / F
5. Select the GENENG program.	UIR-2	System displays general engineering courses	P / F

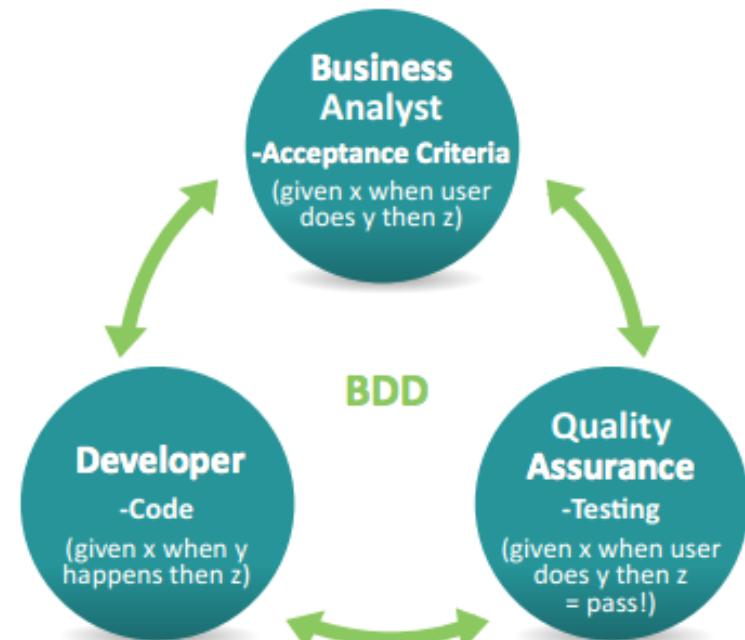
Lots of detail  
Must document passes manually  
Maintenance issue when steps depend  
on each other

# Behavior based development (BDD)



# Behavior based development (BDD)

- ◎ Simplify writing test cases is to use **behavior-driven development (BDD)**, which is an **extension of test-driven development** that encourages collaboration between:
  - **developers,**
  - **QA testers**
  - **non-technical**
  - **business participants**



# Behavior Driven Development: Big Idea

- ◎ **Tests from customer-friendly user stories**
  - Acceptance: **ensure satisfied customer**
  - Integration: ensure interfaces between modules have consistent assumptions, and communicate correctly
- ◎ **Meet halfway between the customer and developer**
  - User stories are not code, so clear to the customer and can be used to reach an agreement
  - Also not completely freeform, so can connect to real tests

# Behavior based development (BDD)

1. Business analyst **writes a user story**
2. (Acceptance) tester writes **scenarios based on user story**
3. Business team **reviews scenarios**
4. Test engineer writes the step definitions for the scenario steps
5. QA team writes test scripts (to automate the scenarios)
6. The test scripts are run, issues analysed and bugs fixed
7. The test scripts are run as regression tests
8. End user accepts the software if tests pass (acceptance criteria met)

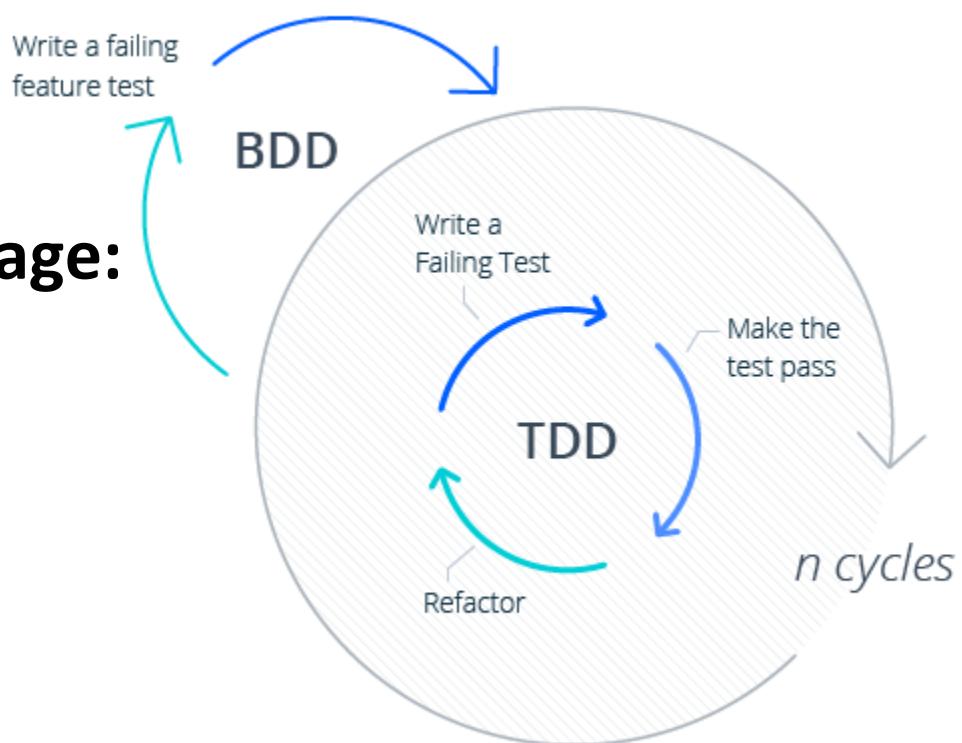
**Focus on the requirements**

**Starting by the test means starting by the requirements!!!**

# Behavior based development (BDD)

- ◎ Behavior-driven development should be focused on the business behaviors your code is implementing: **the “why” behind the code**

**Scenario definition language:  
Gherkin (DSL)**



<https://cucumber.io/docs/gherkin/>

# Requirements based testing: Gherkin

- ◎ A Domain Specific Language (DSL) that **helps non-programmers express requirements (features)** in a structured manner
- ◎ **Requirements-based testing involves examining each requirement and developing a test or tests for it.**

**Scenario Outline:** Newline before Examples

*Squashed together steps with no newlines to separate them makes it more difficult to discern which information belongs together especially if tables are involved*

Given I add a new person

And this person has the birthdate '<birthdate>'

When I try to save this person

Then I receive the error message for 'invalid birthdate'

**Examples:**

birthdate
01.01.1800

# Requirements based testing: Gherkin

1. The first line of this file starts with the keyword **Feature:** followed by a name
  - Features will be saved in **\*.feature files** in Cucumber.
2. Below – **Scenario/s**
3. The last three lines starting with **Given, When, and Then** are the steps of our scenario.

**Feature:**

*Cucumber Feature = Test Scenario*

**Scenario:**

*Cucumber Scenario = Test Case*

**Given**

**When**

**Then**

# Requirements based testing: Gherkin



What  
software will  
look like to  
user

Things that  
the user will  
do

What the  
user should  
expect

**Feature:** login to the system.

**As** a user,

I want to login into the system when I provide username and password.

**Scenario:** login successfully

**Given** the login page is opening

**When** I input username into the username textbox

And I input valid password into the password textbox

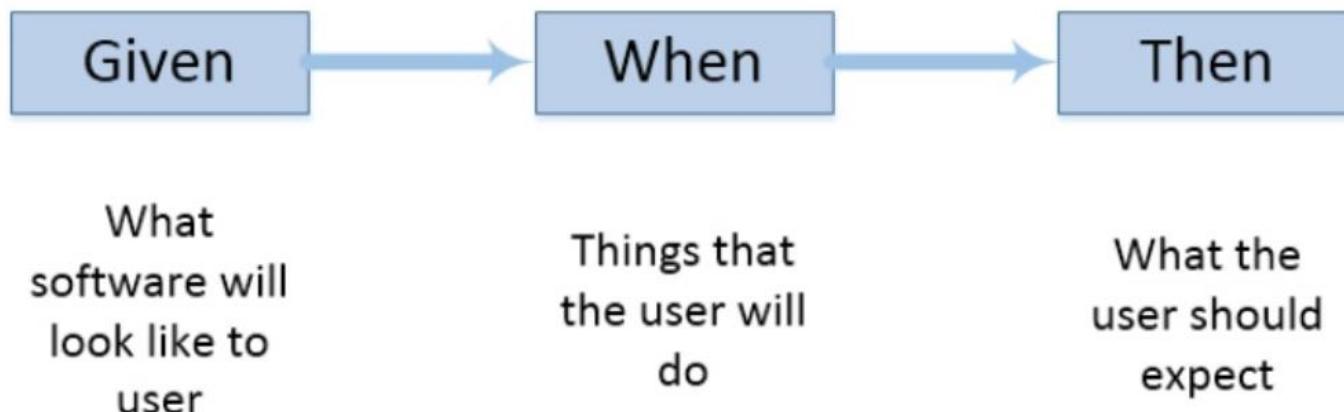
And I click Login button

**Then** I am on the Home page

# Requirements based testing: Gherkin

## Given:

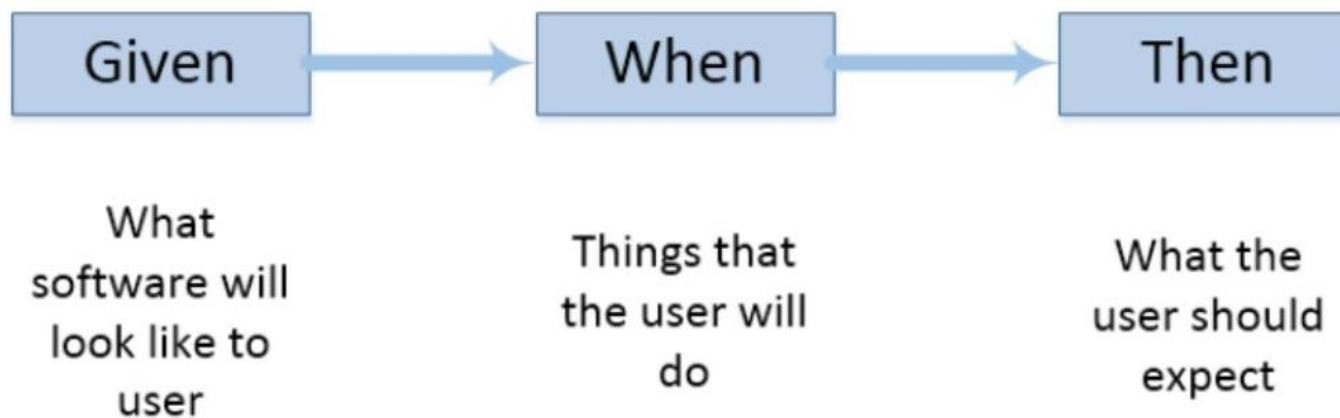
- The purpose of **Given** steps is to put the system in a known **state before the user** (or external system) starts interacting with the system (in the When steps).
- If you have worked with use cases, **givens are your preconditions**.



# Requirements based testing: Gherkin

## ◎ When:

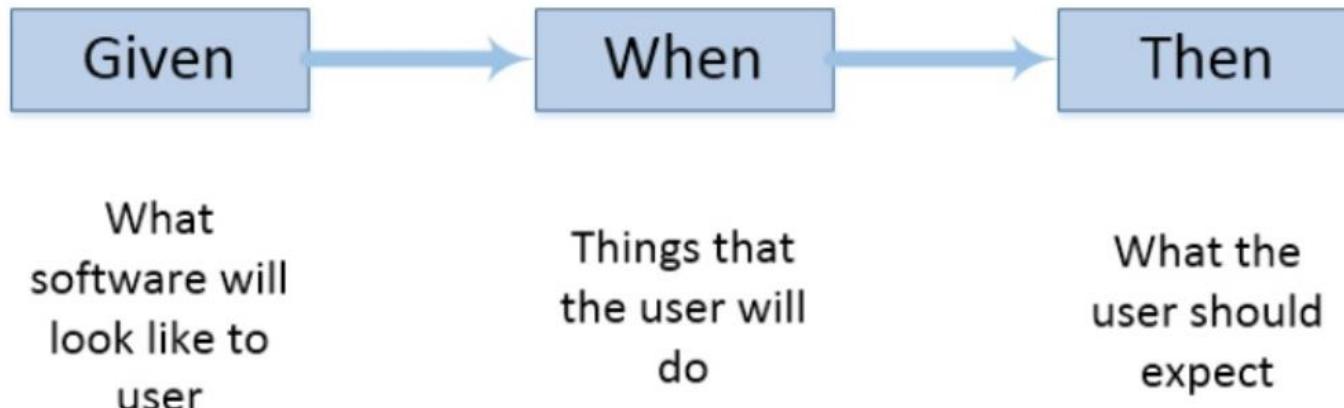
- The purpose of **When** steps is to describe the key action the user performs.



# Requirements based testing: Gherkin

## Then:

- The purpose of **Then** steps is to observe outcomes.
- The observations should be related to **the business value/benefit** in your feature description.
- Thus, it should be related to **something visible from the outside (behavior)**.



# Gherkin Format and Syntax

The keywords are:

- Feature
- Rule (as of Gherkin 6)
- Example (or Scenario)
- Given, When, Then, And, But for steps (or \*)
- Background
- Scenario Outline (or Scenario Template)
- Examples (or Scenarios)
- """ (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)

# Gherkin: example

## Feature: Login to the System

**As a user, I want to be able to log in to the system, So that I can access my account and perform various actions**

Scenario Outline: Successful Login

**Given** the user is on the login page

**When** the user enters their valid username "<username>" and password "<password>"

**And** the user clicks the "Login" button

**Then** the user should be redirected to the dashboard page

**And** the user should see a welcome message "Welcome, <full\_name>!"

Examples:

username	password	full_name
john_doe	Password123	John Doe
jane_smith	Passw0rd456	Jane Smith

# Gherkin: example

## Feature: Login to the System

**As a user I want to be able to log in to the system So that I can access my account and perform various actions**

### Scenario Outline: Unsuccessful Login (Invalid Username)

**Given** the user is on the login page

**When** the user enters an invalid username "<username>" and valid password "<password>"

**And** the user clicks the "Login" button

**Then** the user should see an error message "Username not found, please check your username"

**And** the user should remain on the login page

Examples:

username	password
invalid_user	Password123
unknown_user	Passw0rd456

# Gherkin: example

## Feature: Login to the System

**As a user I want to be able to log in to the system So that I can access my account and perform various actions**

Scenario Outline: Unsuccessful Login (Invalid Password)

**Given** the user is on the login page

**When** the user enters their valid username "<username>" and invalid password "<password>"

**And** the user clicks the "Login" button

**Then** the user should see an error message "Invalid password, please try again"

**And** the user should remain on the login page

Examples:

username	password	
john_doe	InvalidPass	
jane_smith	WrongPass	

## Advantages of BBD

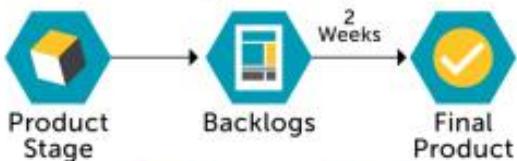
- ◎ **Better communication** between **developers, testers and product owners**.
- ◎ Being **non-technical** in nature, it can reach a wider audience
- ◎ The behavioral approach **defines acceptance criteria prior to development**.
- ◎ No defining ‘test’, but are defining ‘behavior’.

## Disadvantages of BDD

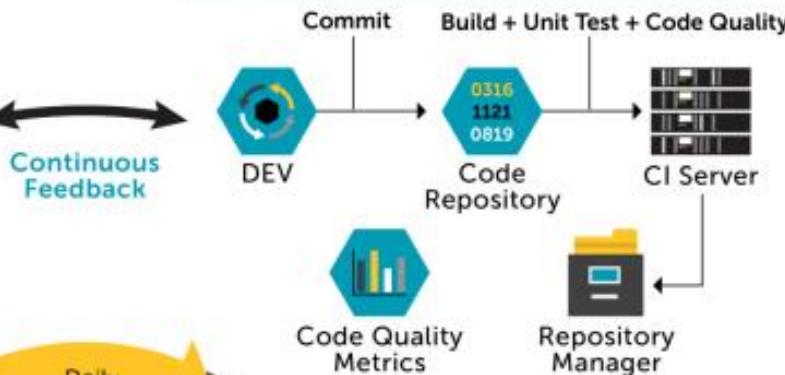
- ◎ To work in BDD, prior experience of TDD is required.
- ◎ If the requirements are not properly specified, BDD may not be effective.
- ◎ **Testers using BDD need to have sufficient technical skills.**

## AGILE DEVELOPMENT

### Daily Standup



## CONTINUOUS INTEGRATION



Process Flow Chart

User Inputs

Daily Scrum

Sprint Phases

Sprint Review  
Retrospection

Continuous Integration

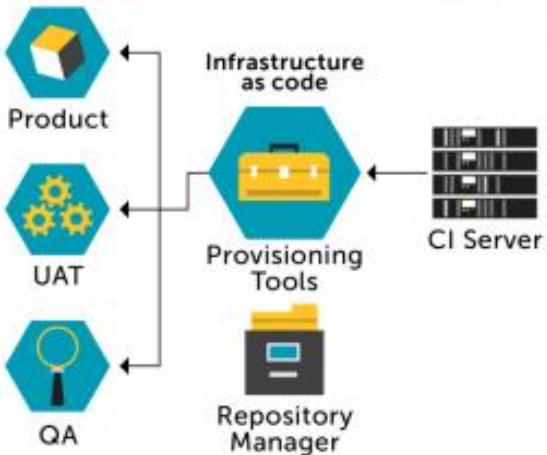
Continuous Testing

Continuous Delivery

Code Quality Metrics

Repository Manager

## CONTINUOUS DELIVERY



Infrastructure as code

Product

UAT

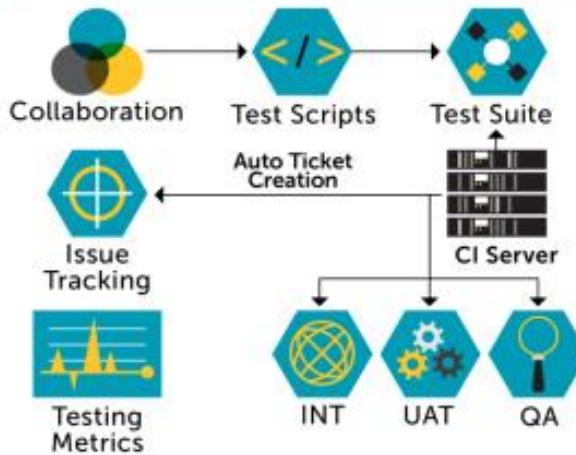
QA

Provisioning Tools

CI Server

Repository Manager

## CONTINUOUS TESTING



Collaboration

Issue Tracking

Testing Metrics

Auto Ticket Creation

CI Server

INT

UAT

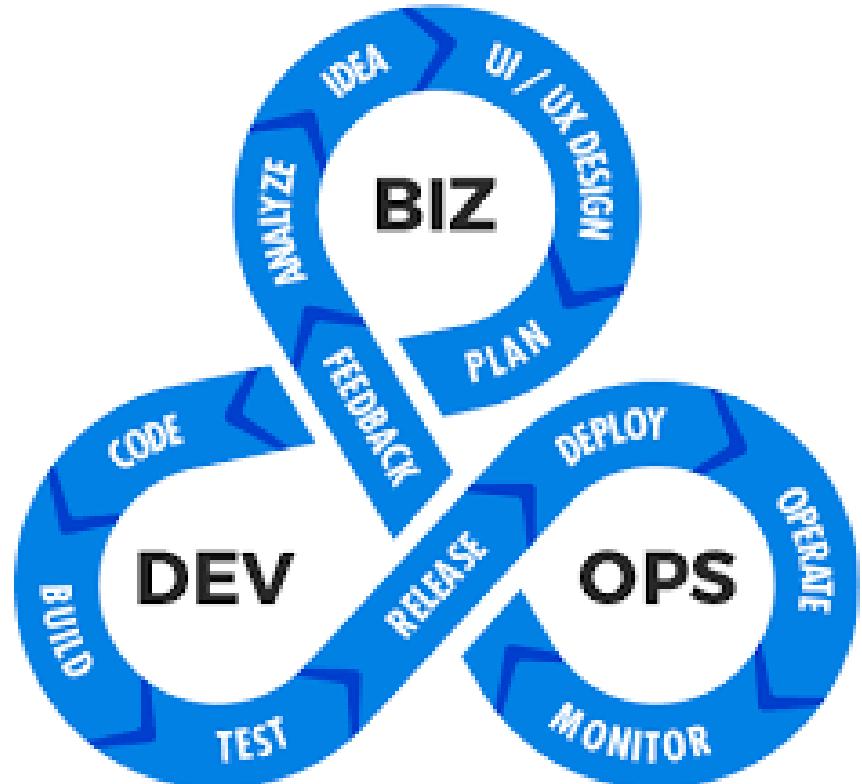
QA

Test Scripts

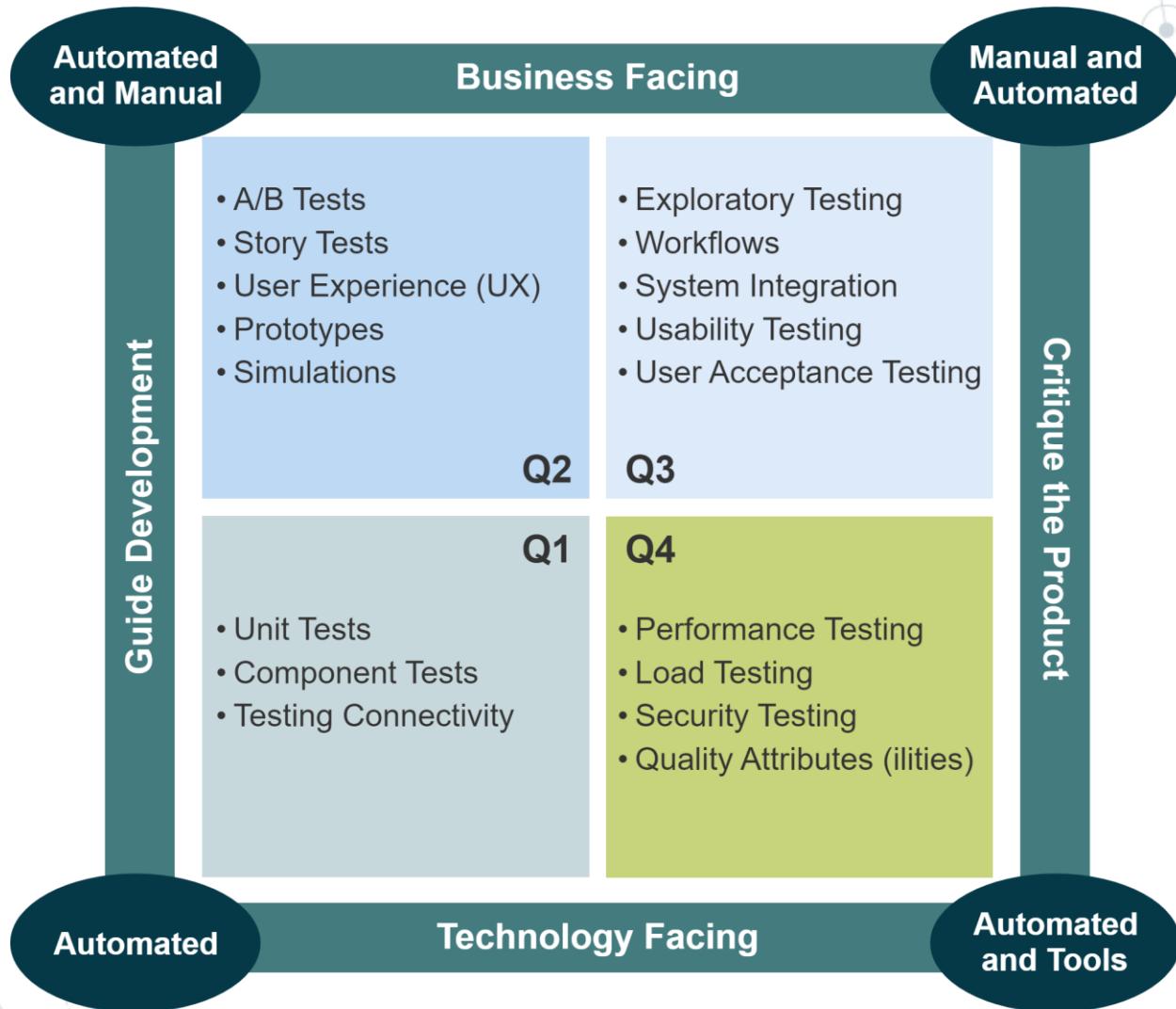
Test Suite

# Continues testing

- ◎ **Continuous Testing** is a software testing type that involves testing the software at every stage of the software development life cycle



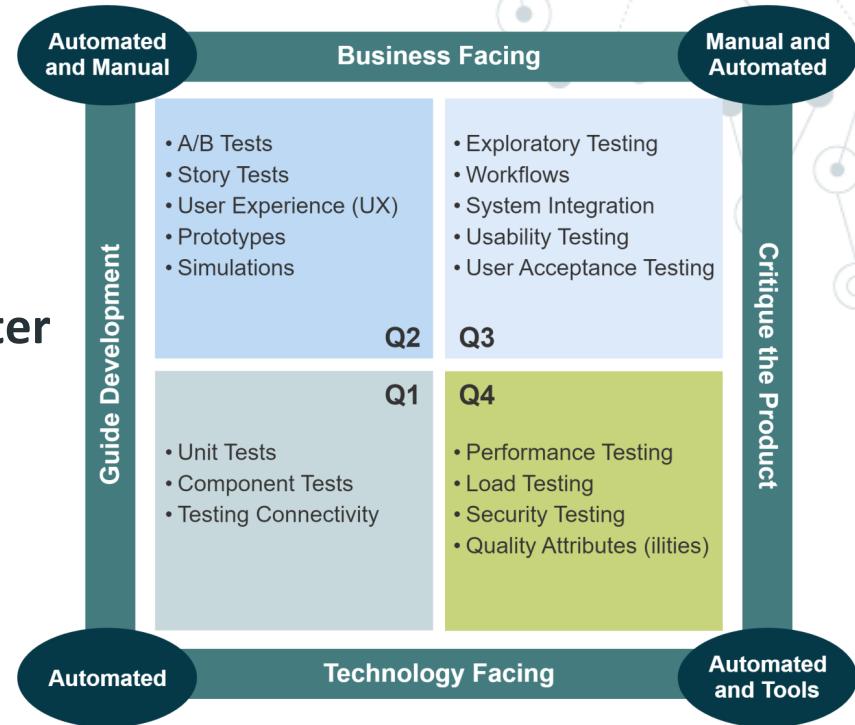
# Agile Continuous Testing



# Agile Continuous Testing

Q1 – contains **unit and component tests**.

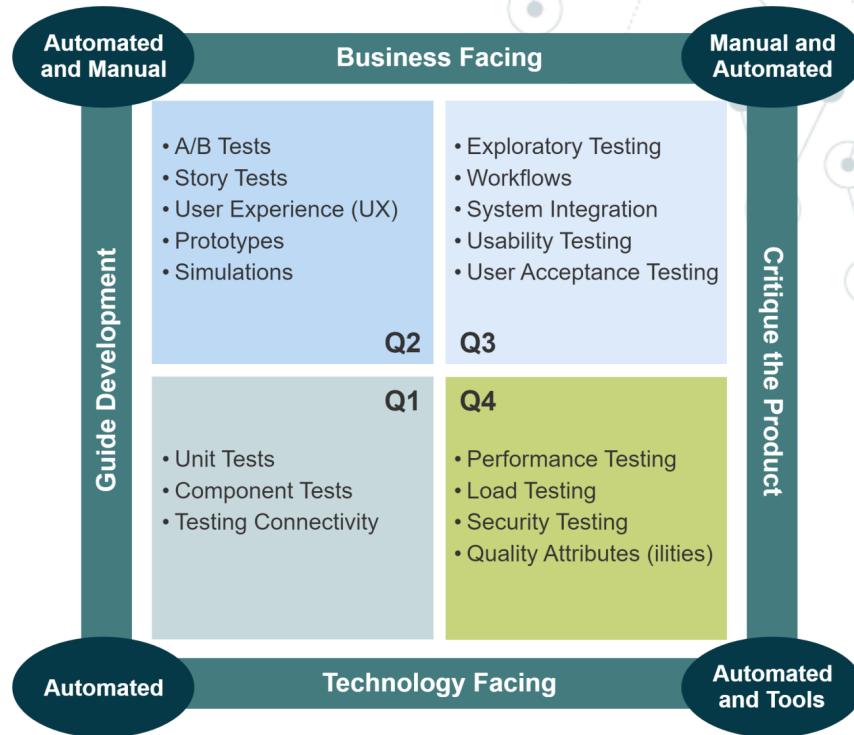
- **To confirm that the system works as agreed**
- **Tests are written to run before and after code changes.**
- In software, this is largely the home of TestDriven Development (TDD).



# Agile Continuous Testing

Q2 – contains **functional tests**:

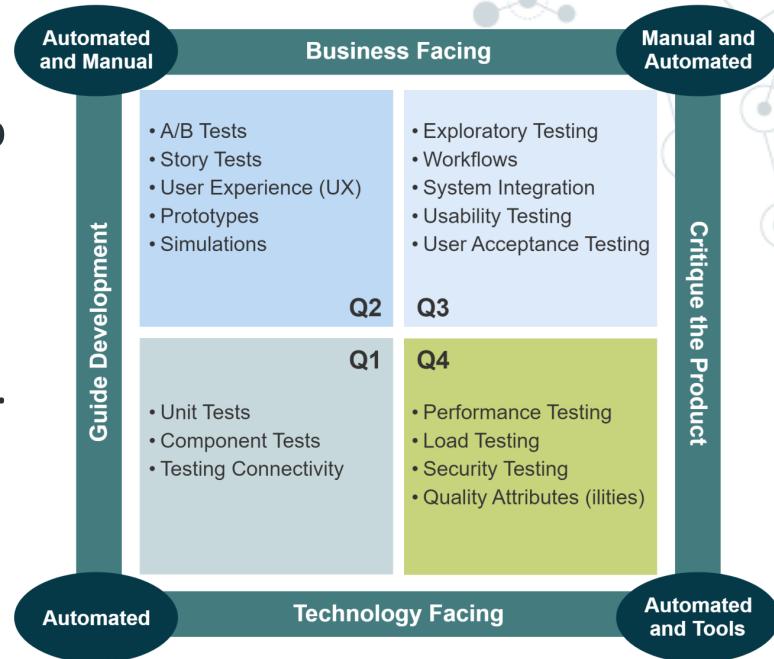
- **user acceptance tests** for stories, features, and capabilities, to validate that they work the way the Product Owner (or Customer/user) intended.
- **Feature-level and capability-level acceptance tests** confirm the aggregate behavior of many user stories.



# Agile Continuous Testing

Q3 – contains **System-level acceptance tests**

- ◎ Contains system-level acceptance tests **to validate that the behavior of the whole system meets usability and functionality requirements**, including scenarios that are often encountered during system use.
  - They involve users and testers engaged in real or simulated deployment scenarios, these tests are often manual.
- ◎ They're frequently **the final system validation** before delivery of the system to the end-user.

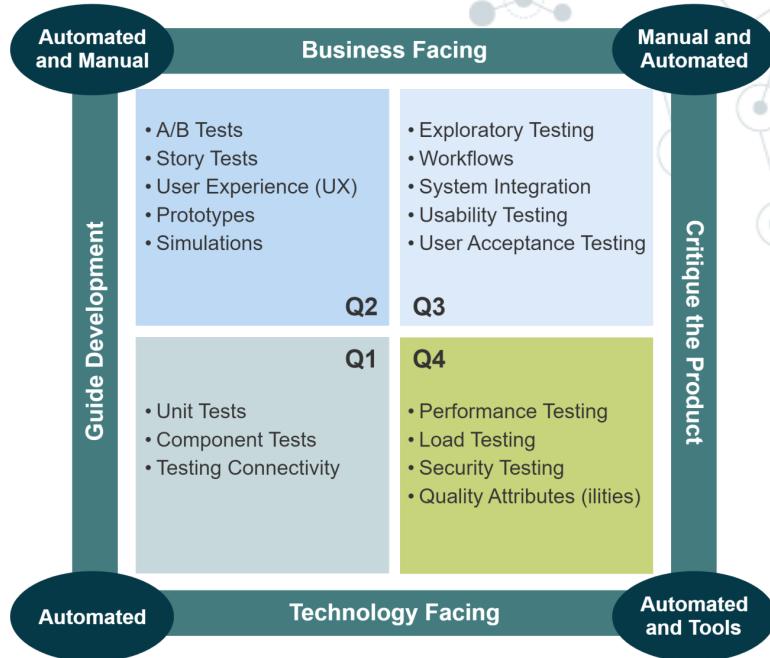


© Scaled Agile, Inc.

# Agile Continuous Testing

Q4 contains **systems qualities test**

- ◎ Contains system qualities testing to verify **the system meets its Nonfunctional Requirements (NFRs)**,
  - Typically, they're supported by a suite of **automated testing tools** (such as load and performance) designed specifically for this purpose.
- ◎ Since any system changes can violate conformance with NFRs, they must be run continuously, or at least whenever it's practical.



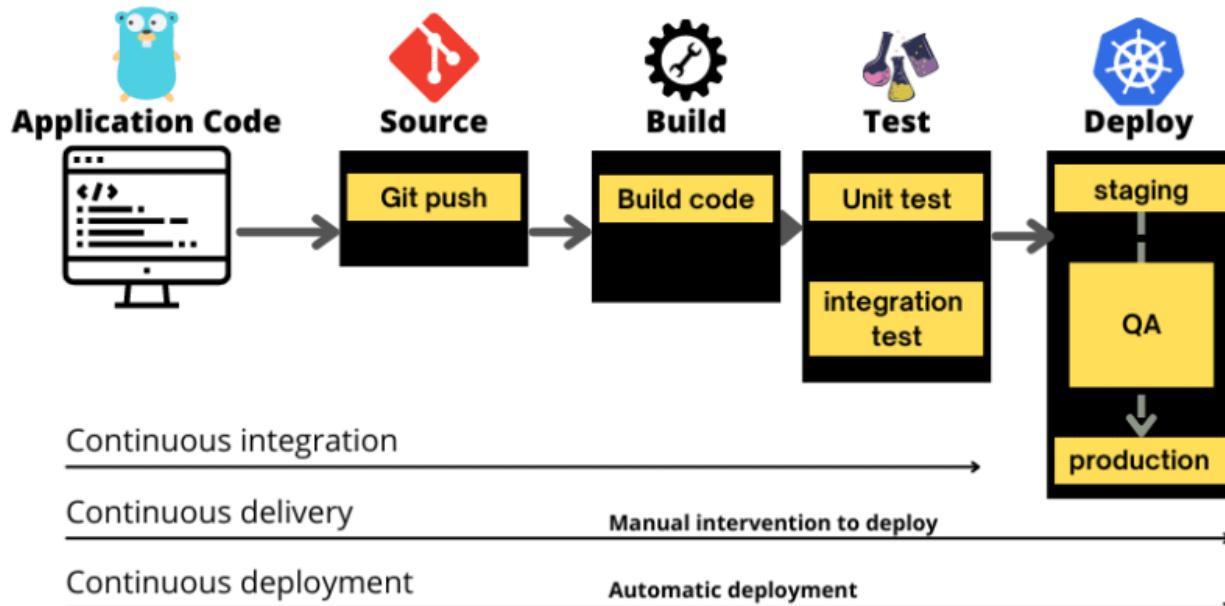
© Scaled Agile, Inc.

# Agile testing: Manual vs Automated

- ① **Exploratory Testing Manual System Testing** is a popular agile approach to testing and combines the investigation of the system that is to be tested with the design and performance of manual tests
  - **The tester does not require a detailed specification for the test object or the test.** The test is created while it is being performed and concentrates on suspect or defective components.
  - At the start of a session, the tester defines only the objectives of the test (which feature or User Story is to be tested).
  - **The tester tries to execute the user story or feature concerned and observes the system's behavior.**
  - This approach is perfect for taking a quick look at new or unknown features.
  - **The quality of an exploratory test depends heavily on the tester's degree of discipline, level of experience, and feel for the software.** Tests are difficult or impossible to reproduce and have to be performed manually.  
**Can not be managed or measured**

# Agile testing: Manual vs Automated

- ➊ In Scrum Projects, **unit and integration tests** are usually performed automatically.
  - If these test scripts are embedded in the CI environment, they are automatically run every time the code is altered.



# Agile testing: Manual vs Automated: System Testing

- ➊ To automate system tests is harder, because:
  - *The most important system test interface is the product's GUI,* which requires the use of **dedicated GUI test tools** or external systems.
  - **Test results often have to be analyzed manually** because it is too difficult to automatically compare expected behavior with actual behavior.
  - Some system test cases might require manual intervention.

# Agile testing: Manual vs Automated

Record/playback tools record sequences of commands

◎ A record/playback tool records:

1. all manually entered keyboard and mouse-driven commands
2. the tester performs during a test session
3. saves them in the form of a script.

◎ Running the script reproduces the recorded test sequence—an action that can be repeated as often as necessary.

Item	Operation	Value	Description
Select Device		"John Smith's iPad", ...	Specifies the mobile device as ...
Run TestedApp	Orders		Runs the "Orders" tested appli...
Device			
processOrders			
navigationBar1	TouchItem	"Edit"	Performs a touch on the 'Edit' i...
tableview1	TouchItemXY	0, "Samuel Clemens", 102, 12	Simulates a touch at point (10...
textfield0	SetText	"Mark Twain"	Enters the text 'Mark Twain' in...
navigationBar0	TouchItem	"Save"	Performs a touch on the 'Save' ...
alertView0	TouchButton	"Yes"	Performs a touch on the 'Yes' ...
navigationBar1	TouchItem	"Done"	Performs a touch on the 'Done' ...
Property Checkpoint		Aliases.Device.processOrde...	Checks whether the 'wItem("...")' ...

# Agile testing: Manual vs Automated

Record/playback tools record sequences of commands

- However, **the GUI is an element of the product that often changes** significantly during the course of the project's Sprints, so the corresponding tests have to be constantly altered to fit the changes.
- Generally, it is preferable to spend time during a Sprint writing new tests for new product functions rather than constantly maintaining and updating existing tests.



# Data-driven scripting vs Keyword –driven scripting

## Example: Simple Login Form

Test with **different** combinations of **username** and **password**

**Simple Login Form**

Username :  Password :

**Problem:** Necessary to write three scripts for three different combinations?

- 1. Go to login page
- 2. Type username “Hansen”
- 3. Type password“oslo123”
- 4. Click “Login” button

- 1. Go to login page
- 2. Type username “Olsen”
- 3. Type password“bergen456”
- 4. Click “Login” button

- 1. Go to login page
- 2. Type username “Jensen”
- 3. Type password“harstad789”
- 4. Click “Login” button

# Data-driven scripting vs Keyword – driven scripting

This **test** approach is **time-consuming**

**Solution:** Separate test script from data (username, password) → No hard-coding

One script retrieves different combinations of username and password

1. Go to page
2. Type username “`file.nextUsername()`”
3. Type password “`file.nextPassword()`”
4. Click “Login” button

Username	Password
Hansen	oslo123
Olsen	bergen456
Jensen	harstad789

# Data-driven scripting vs **Keyword** – driven scripting

## Keyword-driven scripting

Keywords symbolising actions (functionality)

“One level up” from data-driven scripting

Can write tests using keywords

*“What to test, rather than how to test it”*

Keyword	Script
Login	script1
CH_password	script2
Logout	script3

[script1]

1. Go to page
2. Type username “file.nextUsername()”
3. Type password “file.nextPassword()”
4. Click “Login” button

[script2]

1. Click on user avatar
2. Click “Change password”
3. Type current password
4. Type new password
5. Click “Confirm” button

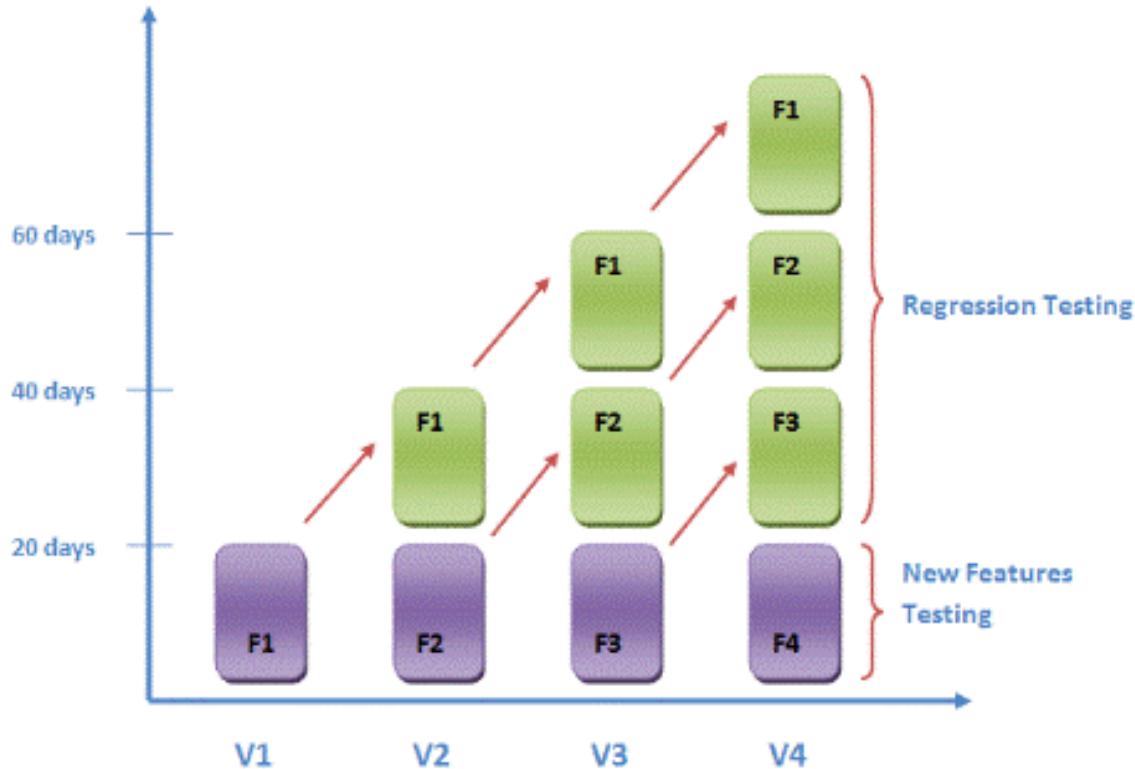
[script3]

1. Click on user avatar
2. Click “Logout” button

# Agile testing: Manual vs Automated

**Regression Testing** is testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made.

It is performed when the software or its environment is changed. [ISTQB]

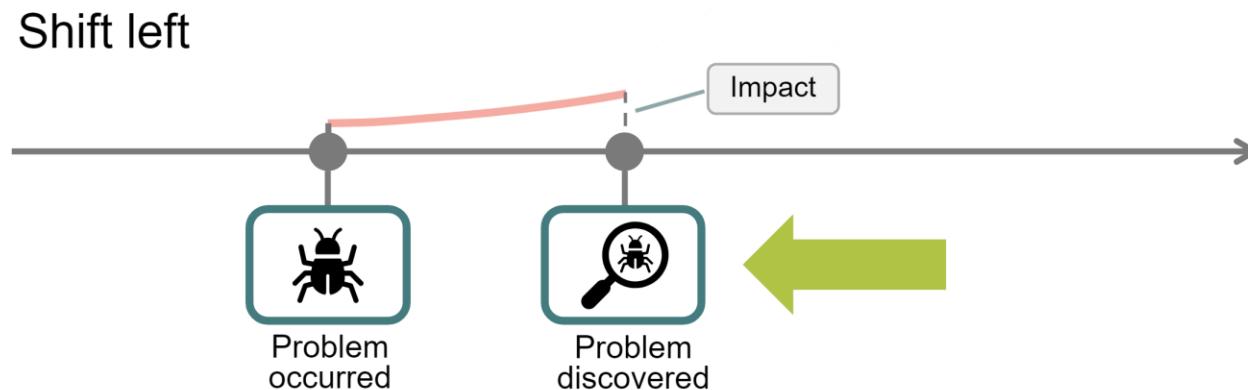
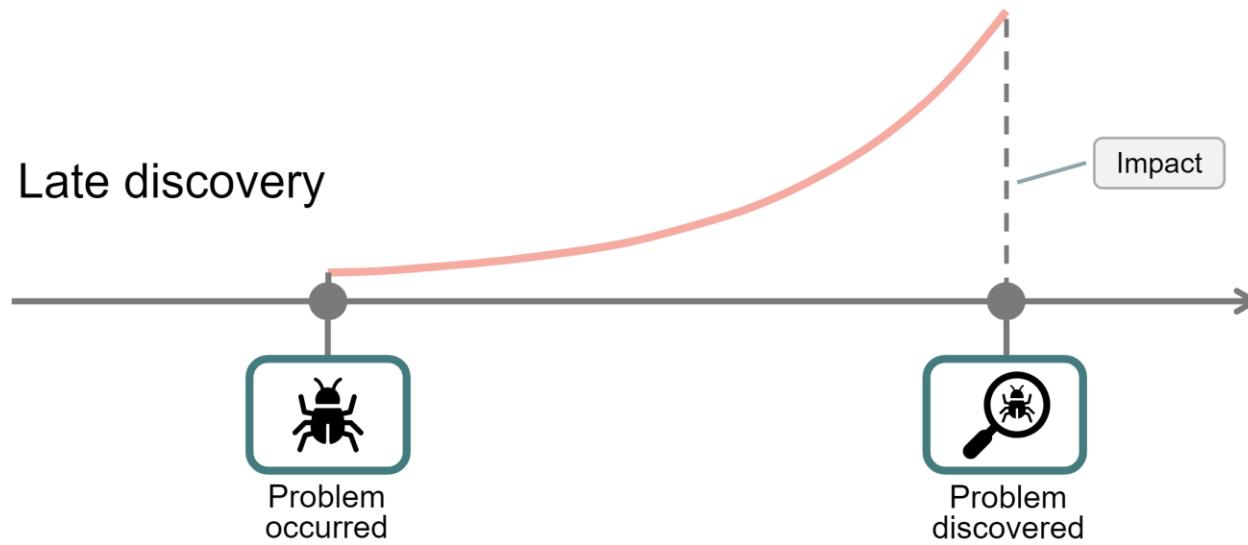


# Programų kūrimo procesas

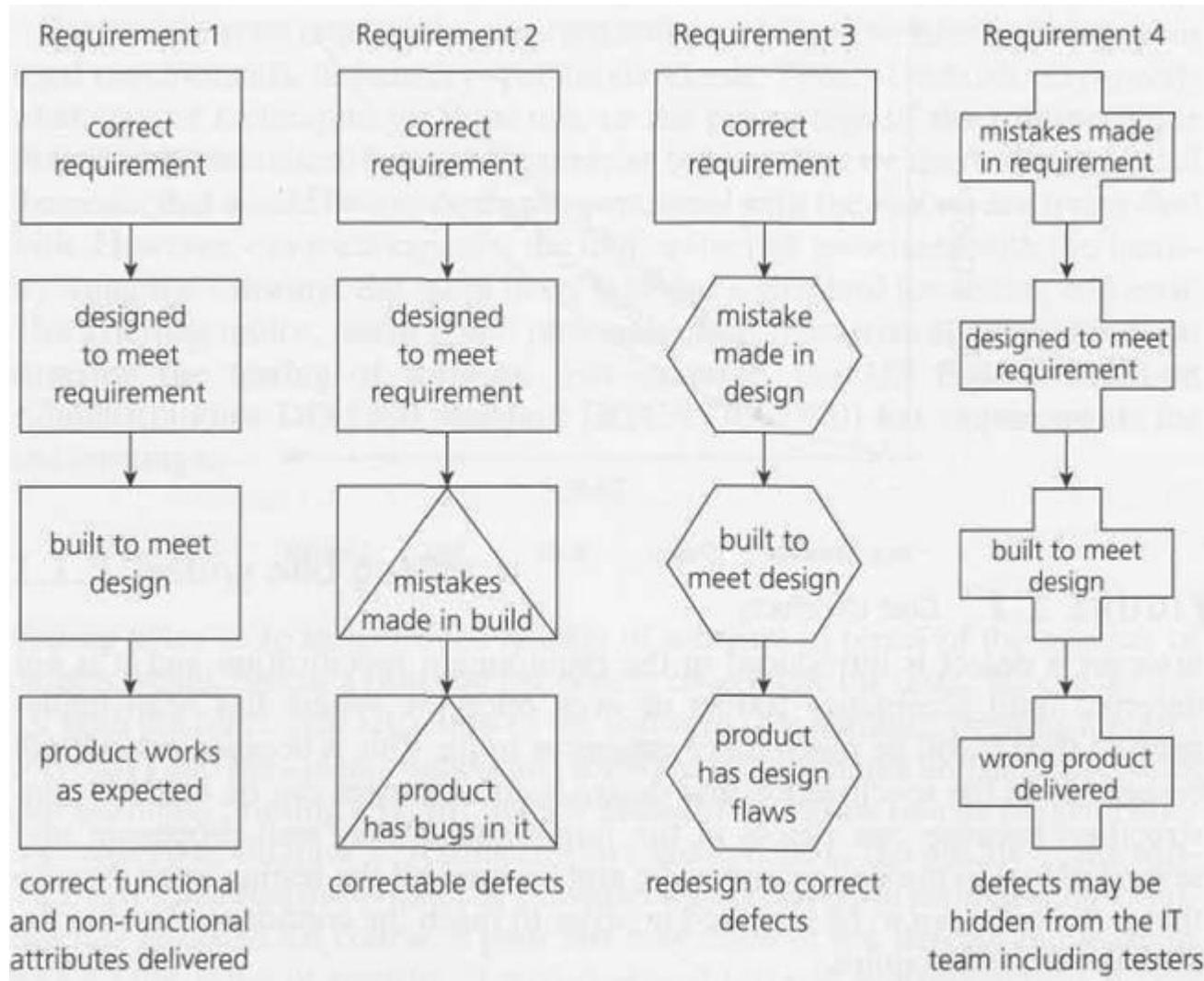
From Requirements to Test Cases

Dr. Asta Slotkiene

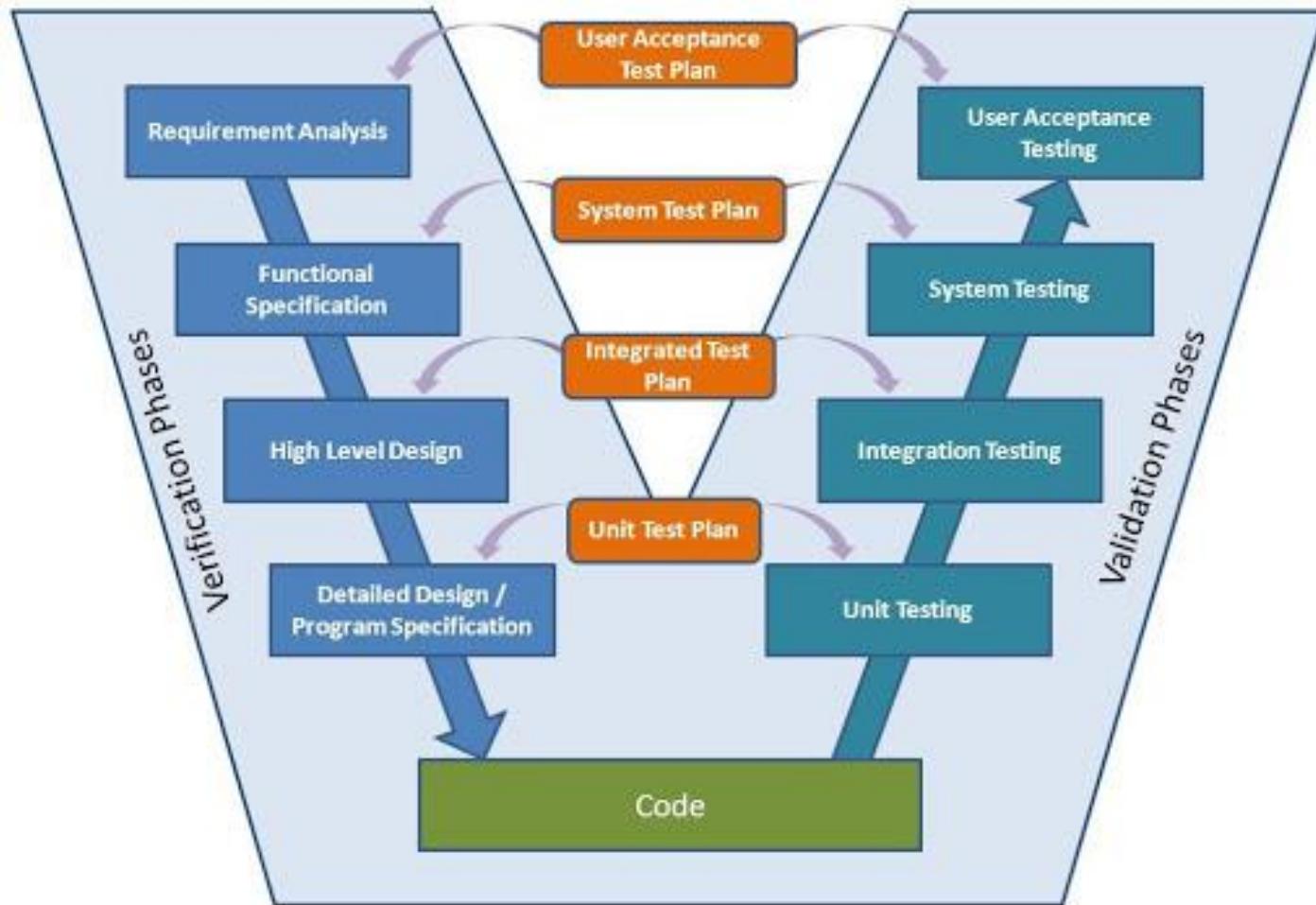
# Agile Quality Practices



# Types of error and defect

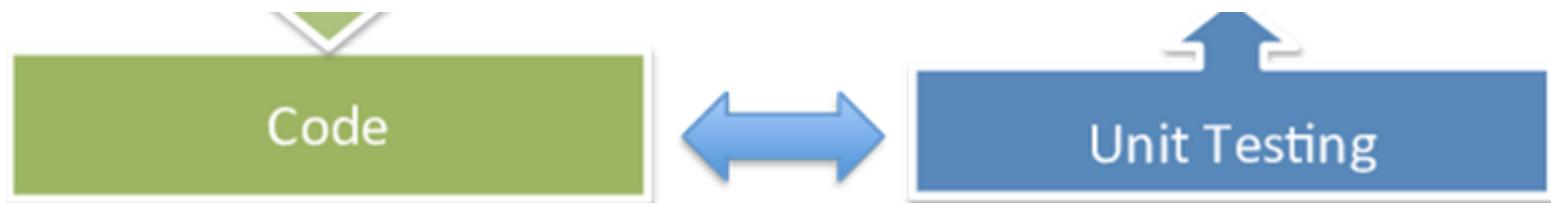


# V model



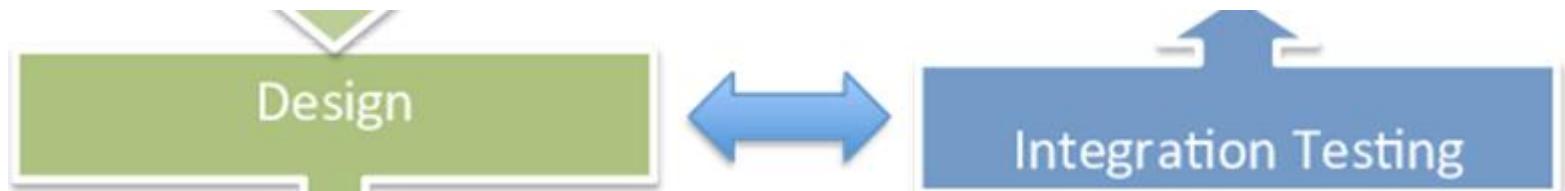
# Testing Level: Unit Testing

- Who: Developers
- How:
  - White-Box Testing Method
  - UT frameworks (e.g., jUnit), drivers, stubs, and mock/fake objects are used



# Testing Level: Integration Testing

- Who: Either Developers themselves or independent Testers
- How:
  - Any of Black Box, White Box, and Gray Box Testing methods can be used
  - Test drivers and test stubs are used to assist in Integration Testing.



# Testing Level: System Testing

- Who:
  - Normally, independent Testers perform System Testing
- How:
  - Usually, Black Box Testing method is used.



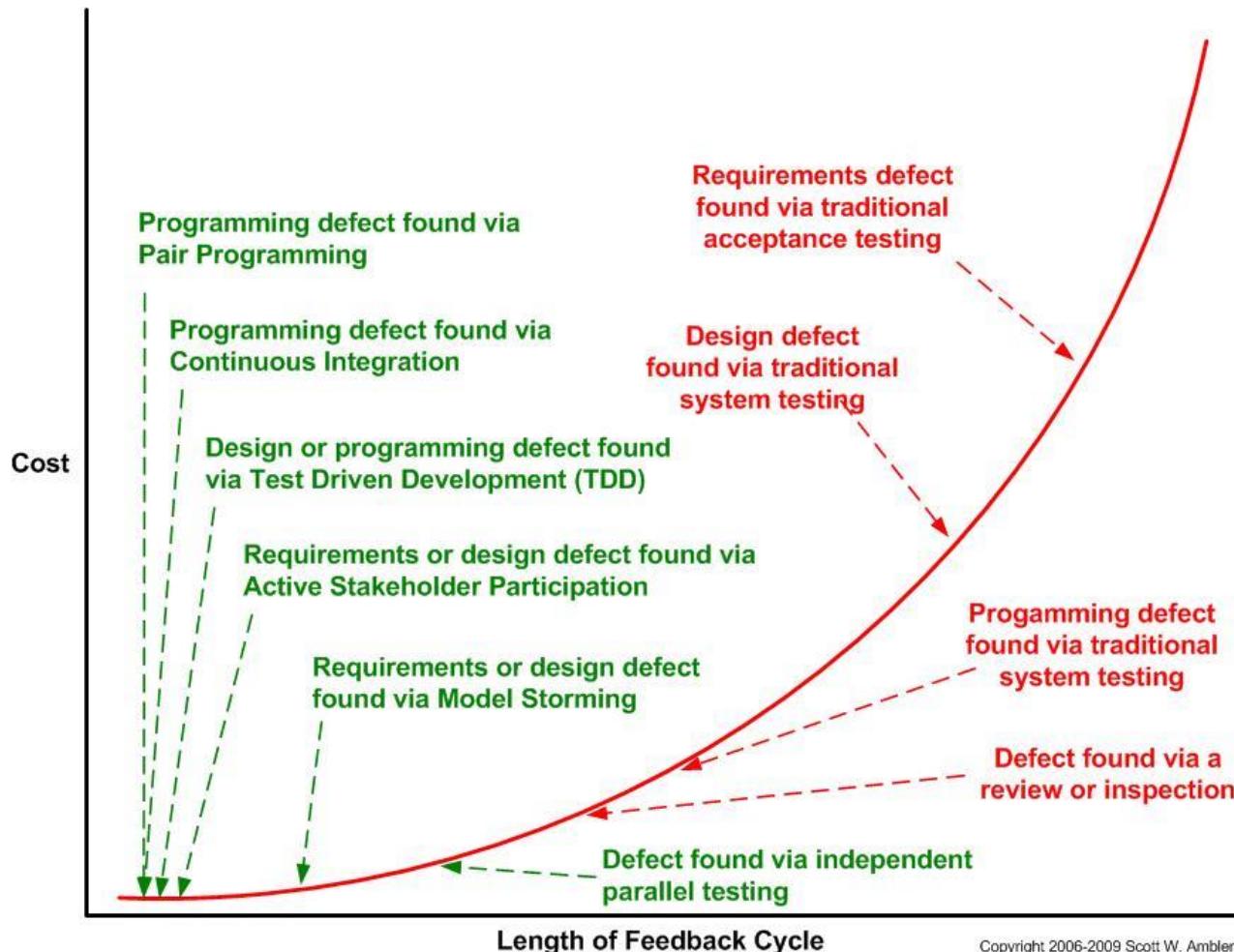
# Testing Level: Acceptance Testing

- Who:
  - Product Management, Sales, Customer Support, Customers
- How:
  - Usually, Black Box Testing method is used; often the testing is done ad-hoc and non-scripted



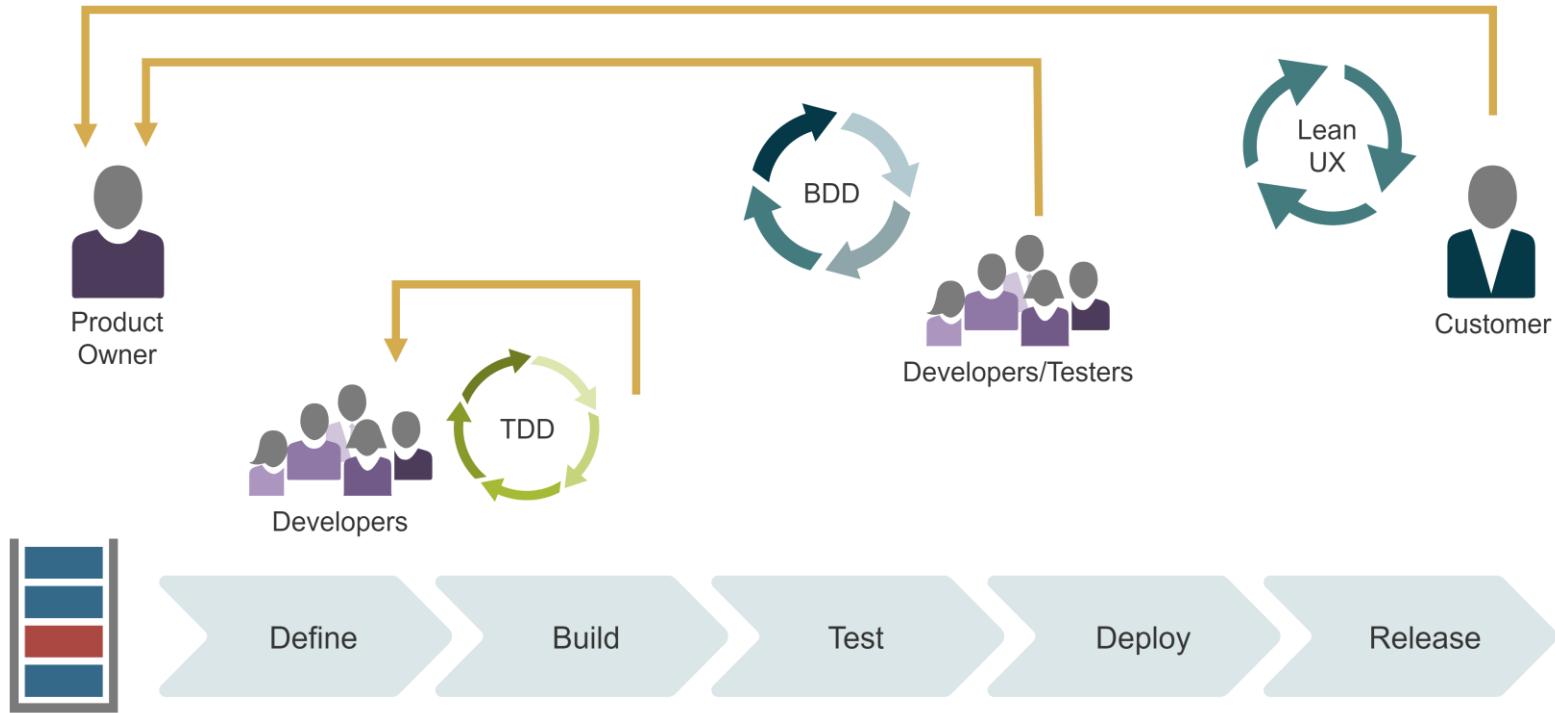
**also called: Behavior-driven testing (BDD)**

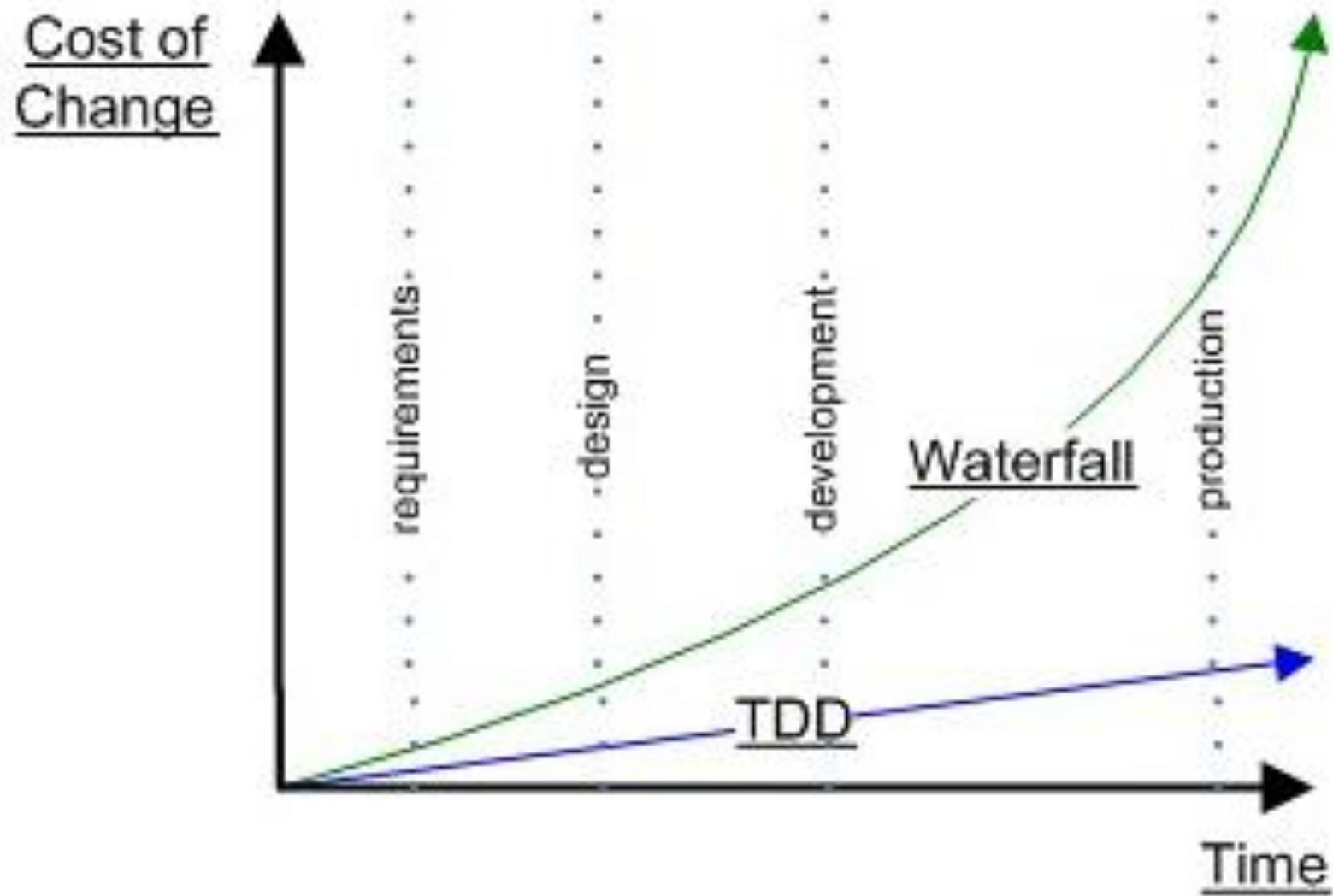
# Comparing the feedback cycle of various development techniques



Copyright 2006-2009 Scott W. Ambler

# Agile Quality Practices

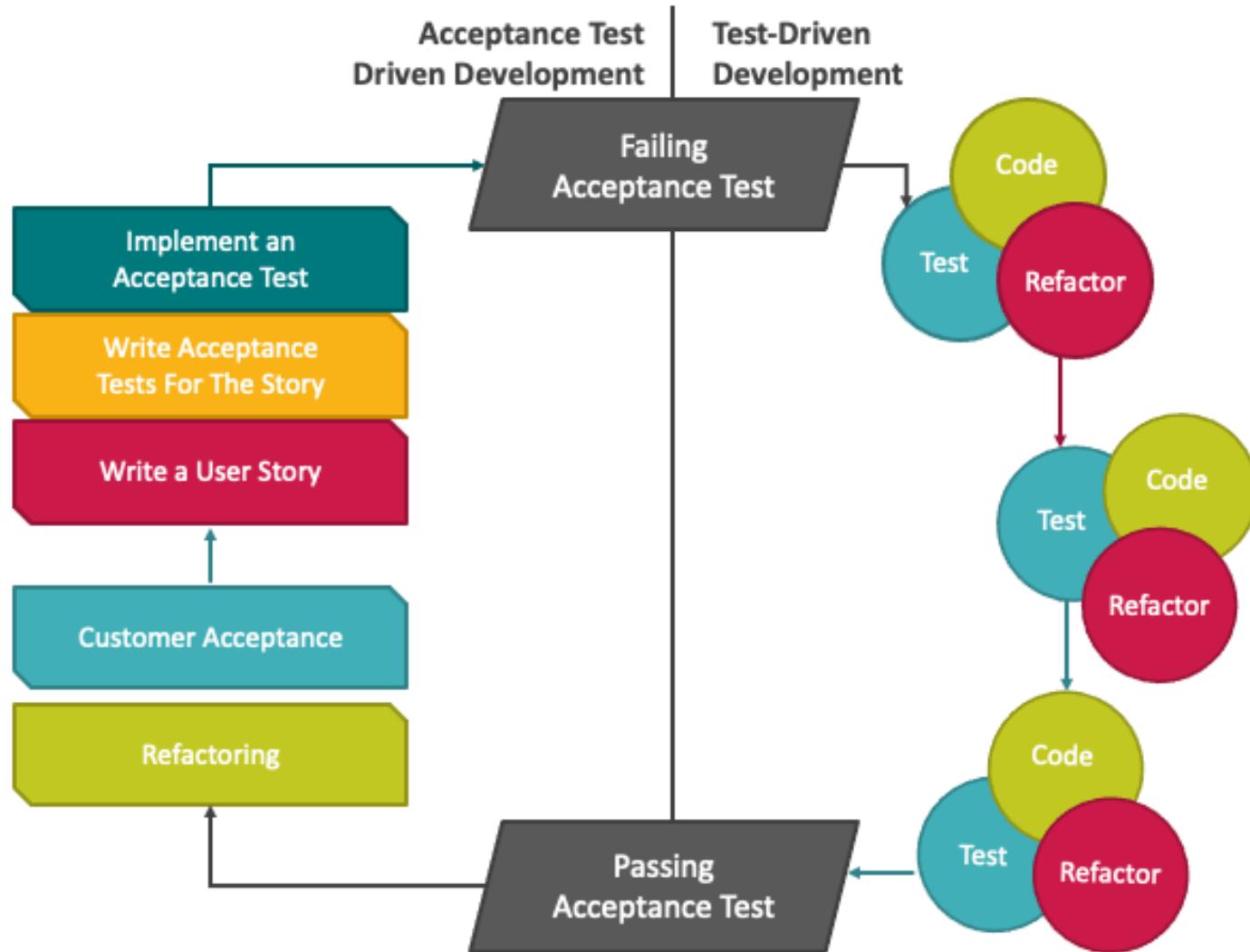




# Agile Testing Methodology

- Test Driven Development (TDD)
- Acceptance Test Driven Development
- Behavior Driven Development (BDD)

# ATDD



# Test-driven development (TDD)

- A software development technique in which the test cases are developed, and often automated, and then **the software is developed incrementally to pass those test cases.**
- ISTQB Glossary
  - <https://glossary.istqb.org/search/>

# Acceptance Test-driven development (ATDD)

- A collaborative approach to development in which **the team and customers are using the customers own domain language to understand their requirements**, which forms the basis for testing a component or system.
- ISTQB Glossary
  - <https://glossary.istqb.org/search/>

# Behavior driven development (BDD)

- A collaborative approach to development in **which the team is focusing on delivering expected behavior of a component or system for the customer**, which forms the basis for testing.
- ISTQB Glossary
  - <https://glossary.istqb.org/search/>

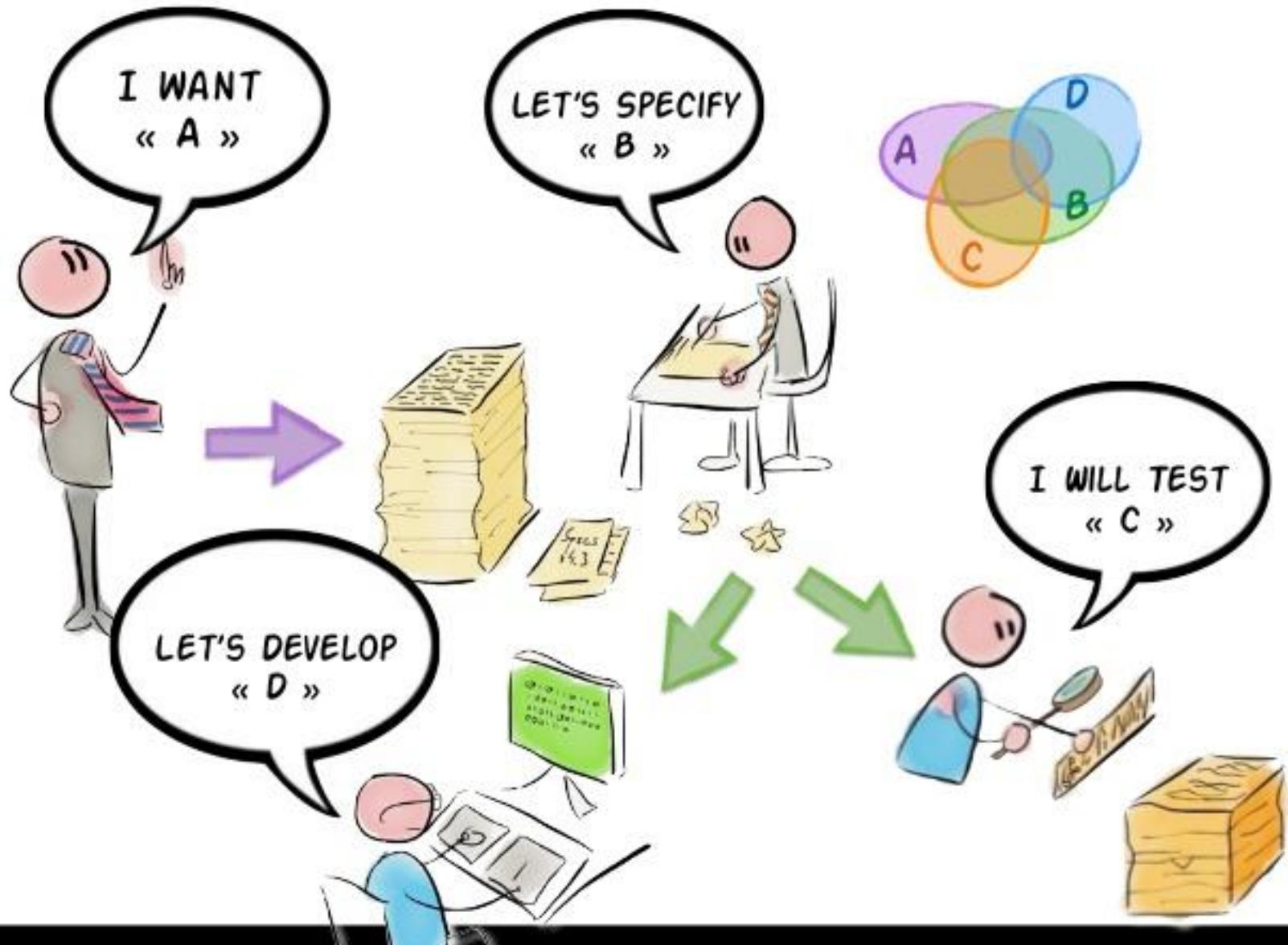
# Why is a focus on BEHAVIOUR so important?

**Users** usually don't care about **technical implementation**

they care about **BEHAVIOUR** of the software

*“...our clients **don't value the code as such**;  
**they value the things that the code does for them.**”*

*Michael Bolton*



# Who should write acceptance tests?

- Answer: client???
  - Probably manual tests: OHHH

Step	Req	Pass Conditions	Pass?
1. Select the Biology program.	UIR-2	System displays biology classes w/ first class BIOLOGY 1150, Section 01, Title GENERAL BIOLOGY, Instructor Block, Anna, Filled/Seats 52/53, Class# 1311, Credits 5, Meets BOE 0221 MWF 8:00-8:52	P / F
2. Double-click on Class# 1330	UIR-1	System includes Class# 1330 in schedule at bottom	P / F
3. Scroll down to Class# 1331 (BIOLOGY 1650, Section 01)	UIR-9	System displays Class# 1331 with a pink background	P / F
4.	UIR-9	All sections listed between #1311 and #1331 have a white background	P / F
5. Select the GENENG program.	UIR-2	System displays general engineering courses	P / F

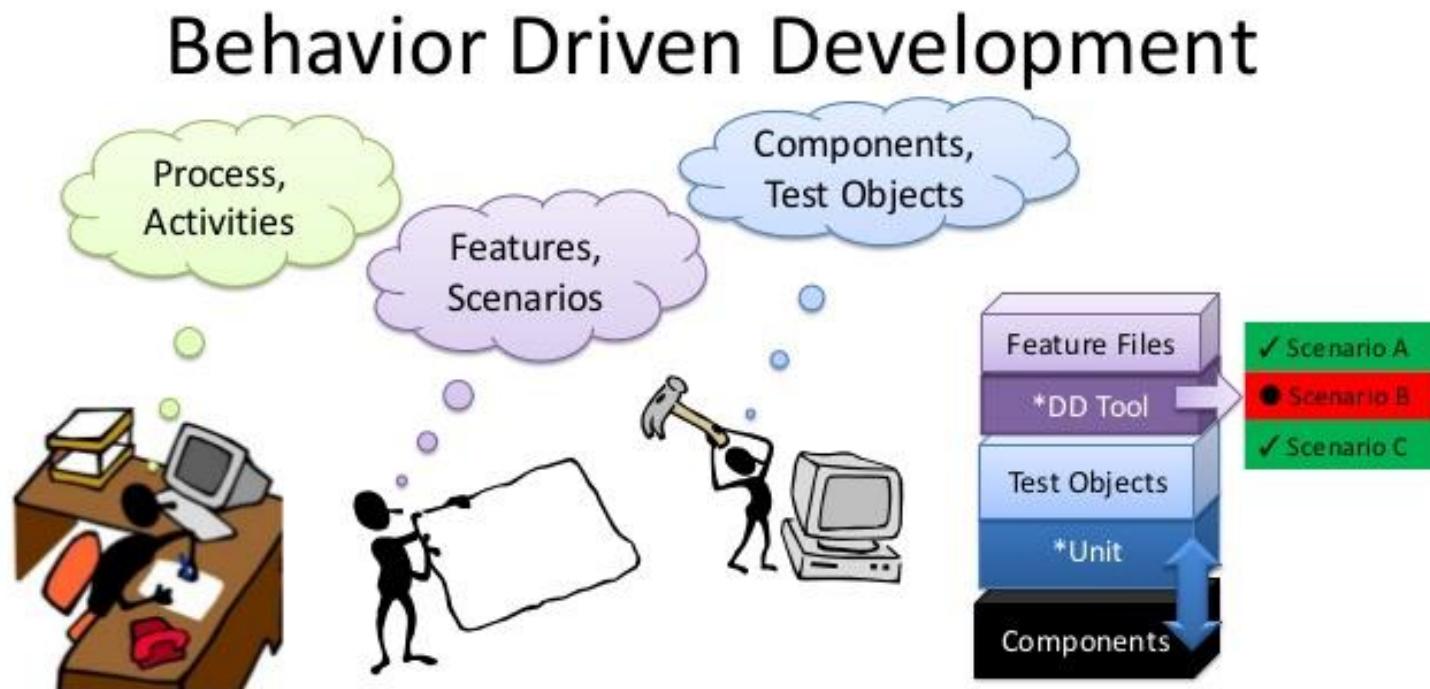
# Who should write acceptance tests?

Client write test cases, Not much help to development team

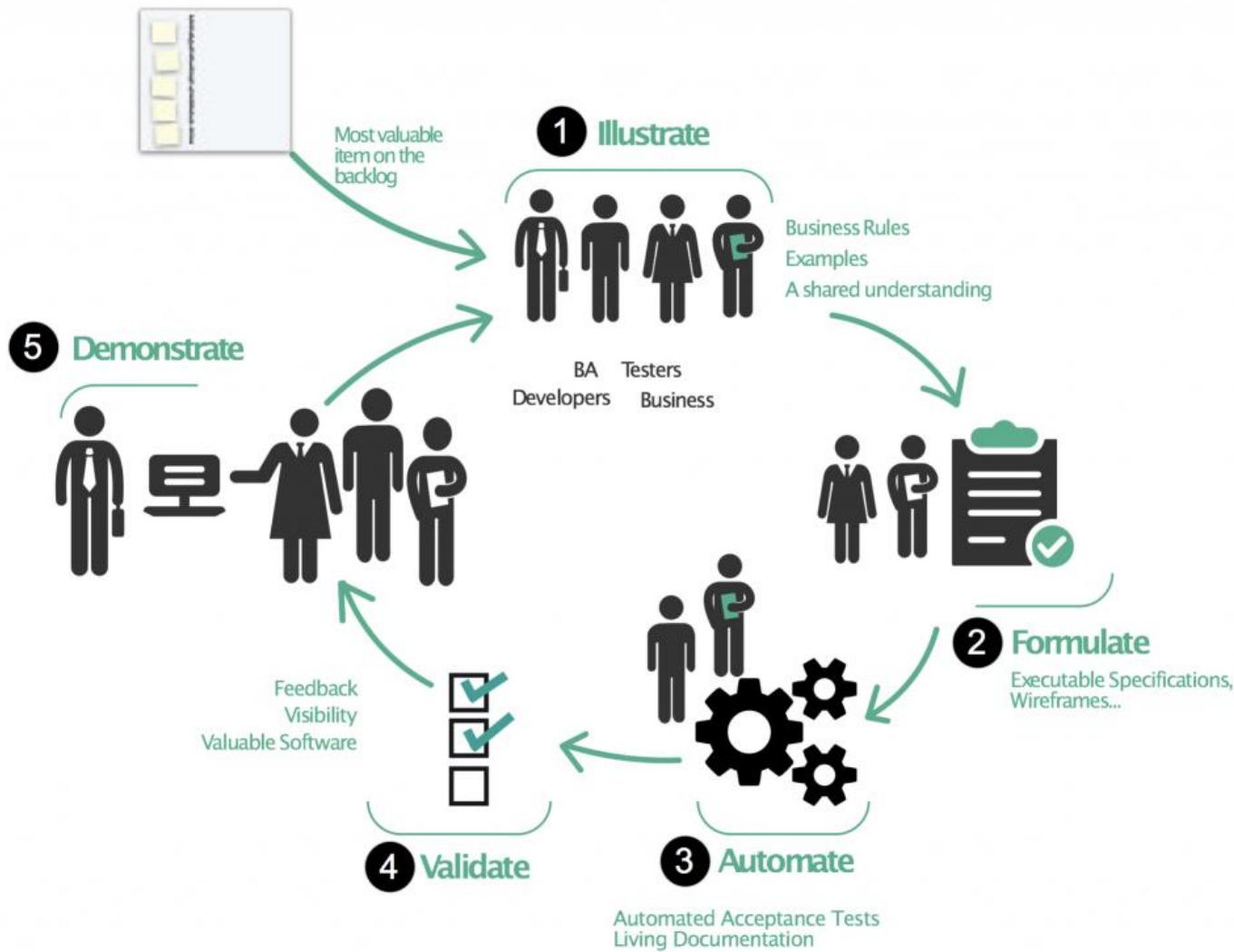
Step	Req	Pass Conditions	Pass?
1. Select the Biology program.	UIR-2	System displays biology classes w/ first class BIOLOGY 1150, Section 01, Title GENERAL BIOLOGY, Instructor Block, Anna, Filled/Seats 52/53, Class# 1311, Credits 5, Meets BOE 0221 MWF 8:00-8:52	P / F
2. Double-click on Class# 1330	UIR-1	Lots of detail Must document passes manually Maintenance issue when steps depend on each other	P / F
3. Scroll down to Class# 1331 (BIOLOGY 1650, Section 01)	UIR-9	All sections listed between #1311 and #1331 have a white background	P / F
4.	UIR-9		P / F
5. Select the GENENG program.	UIR-2	System displays general engineering courses	P / F

# Behavior based development (BDD)

- **Software development methodology based on TDD**

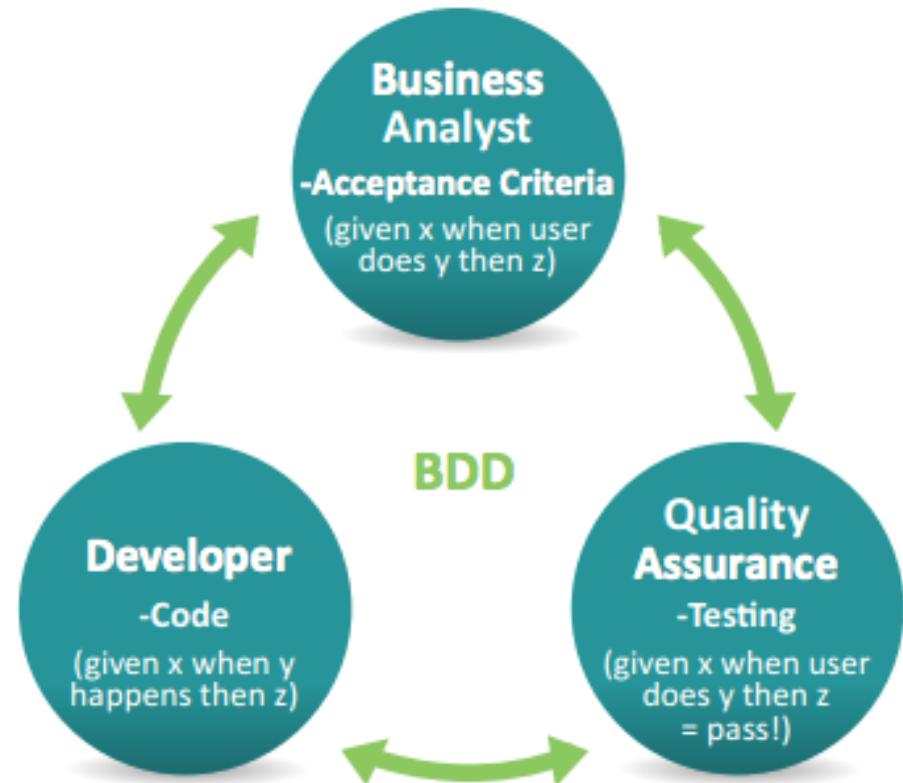


# Behaviour Driven Development



# Behavior based development (BDD)

- Simplify writing test cases is to use **behavior-driven development (BDD)**, which is an **extension of test-driven development** that encourages collaboration between:
  - developers,
  - QA testers
  - non-technical
  - business participants



# Behavior Driven Development

- Basic model: test-driven development
- Developers code to tests written by POs, clients
  - Alternatively, clients review tests written in Gherkin by developers
- If only programmers reviewing or writing tests, Gherkin is probably not useful

# Behavior Driven Development: Big Idea

- **Tests from customer-friendly user stories**
  - Acceptance: **ensure satisfied customer**
  - Integration: ensure interfaces between modules consistent assumptions, communicate correctly
- **Meet halfway between customer and developer**
  - User stories are not code, so clear to customer and can be used to reach agreement
  - Also not completely freeform, so can connect to real tests

# Behavior based development (BDD)

1. Business analyst **writes a user story**
2. (Acceptance) tester writes **scenarios based on user story**
3. Business team **reviews scenarios**
4. Test engineer writes the step definitions for the scenario steps
5. QA team writes test scripts (to automate the scenarios)
6. The test scripts are run, issues analysed and bugs fixed
7. The test scripts are run as regression tests
8. End user accepts the software if tests pass (acceptance criteria met)

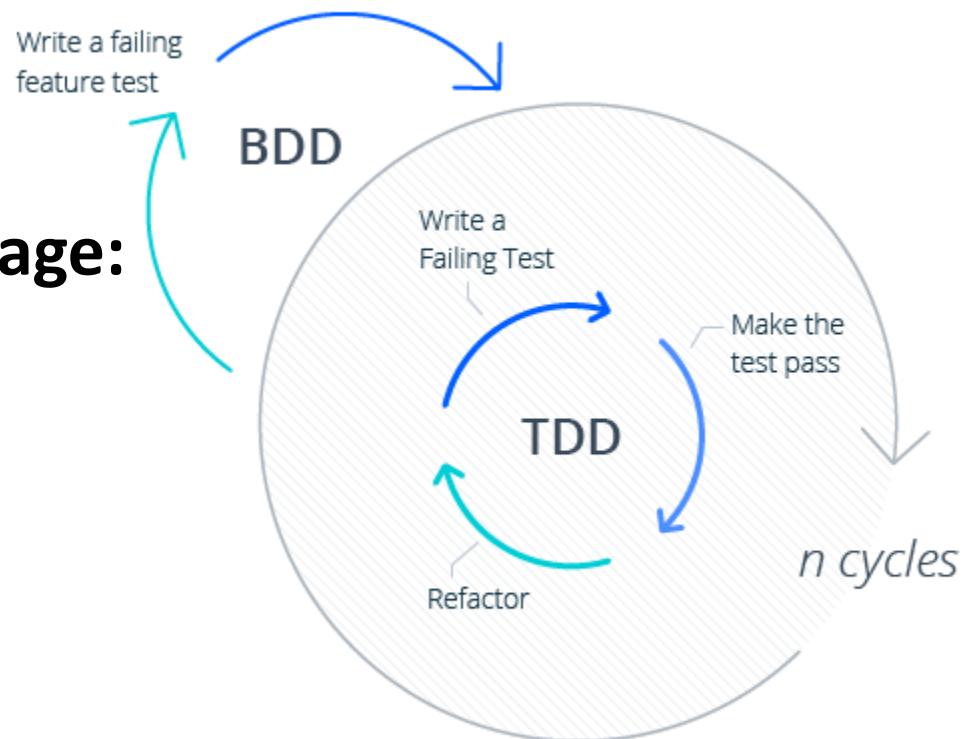
**Focus on the requirements**

**Starting by the test means starting by the requirements!!!**

# Behavior based development (BDD)

- Behavior-driven development should be focused on the business behaviors your code is implementing: **the “why” behind the code**

**Scenario definition language:  
Gherkin (DSL)**



<https://cucumber.io/docs/gherkin/>

# Requirements based testing: Gherkin

- A Domain Specific Language (DSL) that **helps non-programmers express requirements (features)** in a structured manner
- **Requirements-based testing involves examining each requirement and developing a test or tests for it.**

**Feature:** Is it Friday yet?

PMs want to know whether it's Friday

**Scenario:** Monday isn't Friday

**Given** today is Monday

**When** I ask whether it's Friday yet

**Then** I should be told "Nope"

# Requirements based testing: Gherkin

1. The first line of this file starts with the keyword **Feature**: followed by a name
  - Features will be saved in **\*.feature files** in Cucumber.
2. The fourth line - **Scenario**
3. The last three lines starting with **Given**, **When** and **Then** are the steps of our scenario. This is what Cucumber will execute.

**Feature:** *Cucumber Feature = Test Scenario*

*Cucumber Scenario = Test Case*

**Scenario:**

**Given**

**When**

**Then**

# Requirements based testing: Gherkin

**Feature:** login to the system.

As a user,

I want to login into the system when I provide username and password.

**Scenario:** login successfully

Given the login page is opening

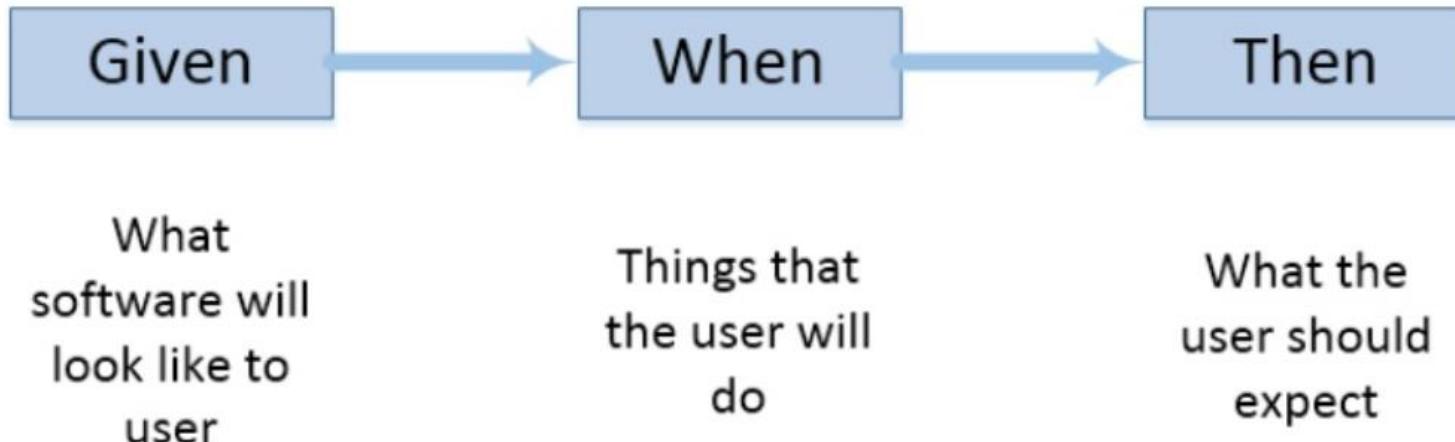
When I input username into the username textbox

And I input valid password into the password textbox

And I click Login button

Then I am on the Home page

# Requirements based testing: Gherkin



**Feature:** login to the system.

As a user,

I want to login into the system when I provide username and password.

**Scenario:** login successfully

Given the login page is opening

When I input username into the username textbox

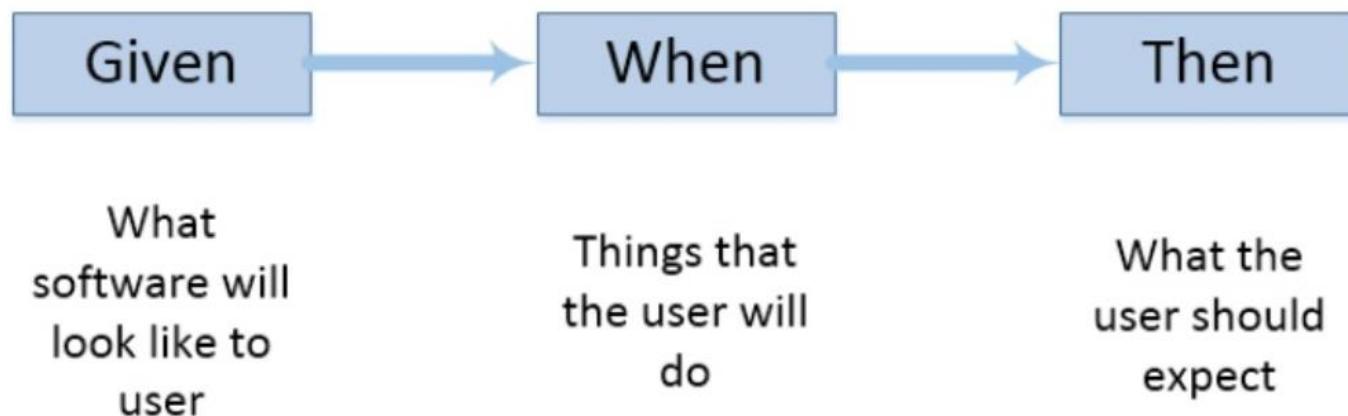
And I input valid password into the password textbox

And I click Login button

Then I am on the Home page

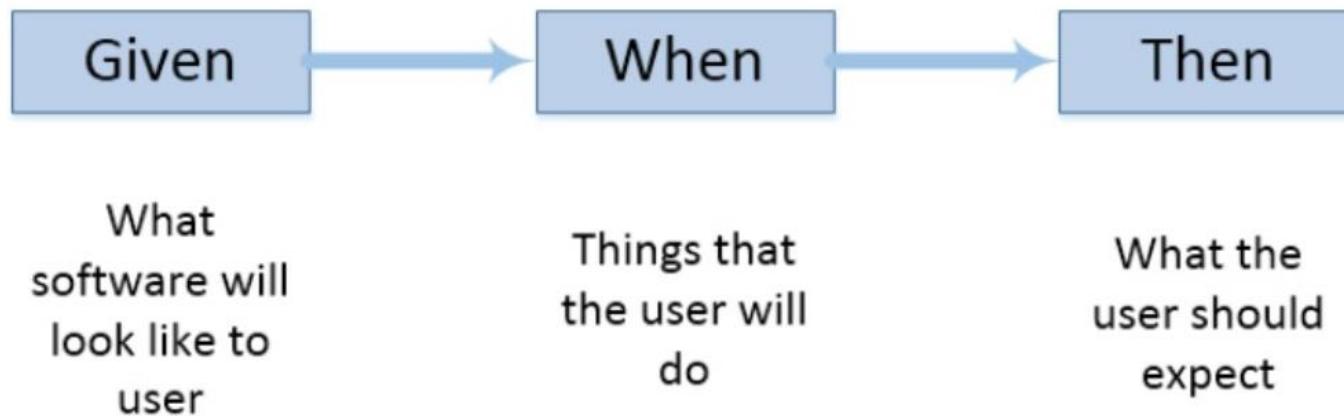
# Requirements based testing: Gherkin

- **Given:**
  - The purpose of **Given** steps is to put the system in a known **state before the user** (or external system) starts interacting with the system (in the When steps).
  - If you have worked with use cases, **givens are your preconditions.**



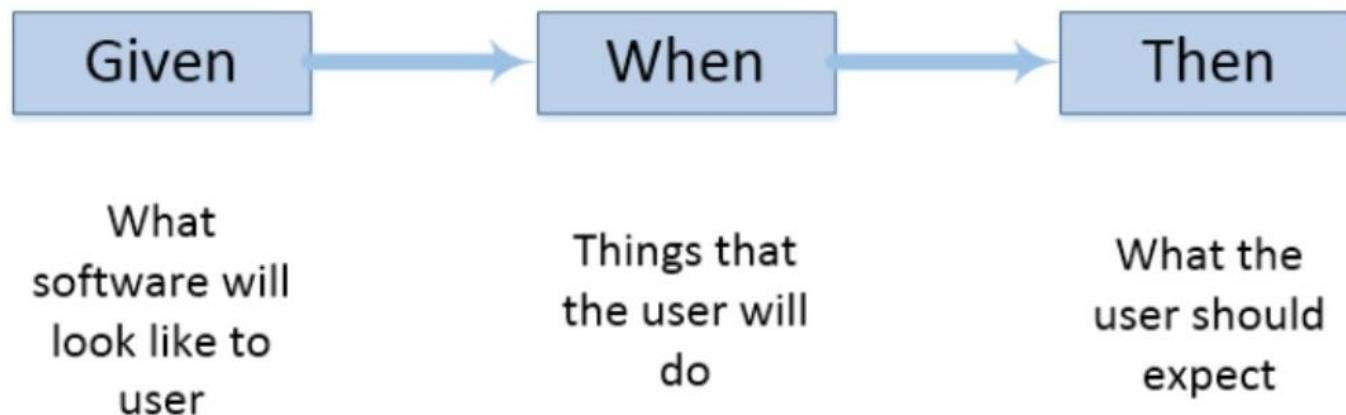
# Requirements based testing: Gherkin

- **When:**
  - The purpose of **When** steps is to describe the key action the user performs.



# Requirements based testing: Gherkin

- **Then:**
  - The purpose of **Then** steps is to observe outcomes.
  - The observations should be related to **the business value/benefit** in your feature description.
  - Thus, it should be related to **something visible from the outside (behavior)**.



# Gherkin: example

**Feature:** Multiple site support

Only blog owners can post to a blog, except administrators, who can post to all blogs.

**Background:**

Given a global **administrator named "Greg"**

And a blog named "**Greg's anti-tax rants**"

And a **customer named "Dr. Bill"**

And a **blog named "Expensive Therapy" owned by "Dr. Bill"**

# Gherkin: example

## **Scenario:**

Dr. Bill posts to his own blog

**Given** I am logged in as Dr. Bill

**When** I try to post to "Expensive Therapy"

**Then** I should see "Your article was published."

# Gherkin: example

## **Scenario:**

Dr. Bill tries to post to somebody else's blog,  
and fails

**Given** I am logged in as Dr. Bill

**When** I try to post to "Greg's anti-tax rants"

**Then** I should see "Hey! That's not your blog!"

# Gherkin: example

## **Scenario:**

Greg posts to a client's blog

**Given** I am logged in as Greg

**When** I try to post to "Expensive Therapy"

**Then** I should see "Your article was published."

# Advantages of BBD

- **Better communication** between developers, testers and product owners.
- Being **non-technical** in nature, it can reach a wider audience
- The behavioral approach **defines acceptance criteria prior to development**.
- No defining ‘test’, but are defining ‘behavior’.

# Advantages of BDD

- **Better communication** between developers, testers and product owners.
- Being **non-technical** in nature, it can reach a wider audience
- The behavioral approach **defines acceptance criteria prior to development**.
- No defining ‘test’, but are defining ‘behavior’.

# Disadvantages of BDD

- To work in BDD, prior experience of TDD is required.
- BDD is incompatible with the **waterfall approach**.
- If the requirements are not properly specified, BDD may not be effective.
- **Testers using BDD need to have sufficient technical skills.**

# Gherkin Format and Syntax

The keywords are:

- Feature
- Rule (as of Gherkin 6)
- Example (or Scenario)
- Given, When, Then, And, But for steps (or \*)
- Background
- Scenario Outline (or Scenario Template)
- Examples (or Scenarios)
- """ (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)

# BDD example: scenario outline

**Feature:** login to the system.

*As a user, I want to login into the system when I provide username and password.*

@tag\_login\_email

**Scenario Outline:** Verify that can login gmail

Given I launch "<https://accounts.google.com>" page

When I fill in "Email" with "<Email>"

And I fill in "Passwd" with "<Password>"

And I click on "signin" button

Then I am on the "Home" page

Scenarios:

Email	Password	
<a href="mailto:kms.admin@gmail.com">kms.admin@gmail.com</a>	kms@2013	
<a href="mailto:kms.user@gmail.com">kms.user@gmail.com</a>	kms@1234	

# BDD example: scenario outline

- **Tables as arguments** to steps are handy for specifying a larger data set - usually as input to a Given or as expected output from a Then.

**Feature:** login to the system.

*As a user, I want to login into the system when I provide username and password.*

@tag\_login\_email

**Scenario Outline:** Verify that can login gmail

Given I launch "<https://accounts.google.com>" page

When I fill in "Email" with "<Email>"

And I fill in "Passwd" with "<Password>"

And I click on "signin" button

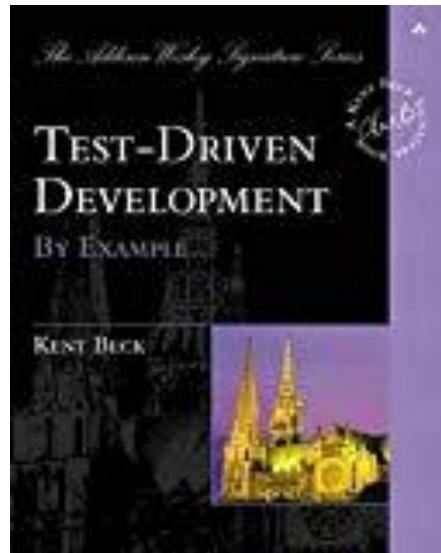
Then I am on the "Home" page

**Scenarios:**

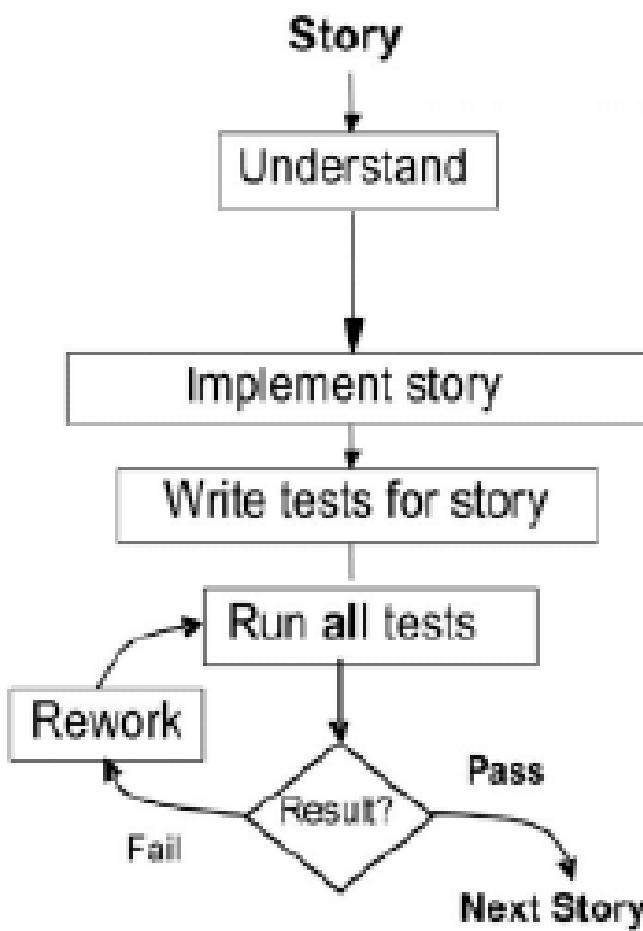
Email	Password	
<a href="mailto:kms.admin@gmail.com">kms.admin@gmail.com</a>	kms@2013	
<a href="mailto:kms.user@gmail.com">kms.user@gmail.com</a>	kms@1234	

# Test-Driven-Development

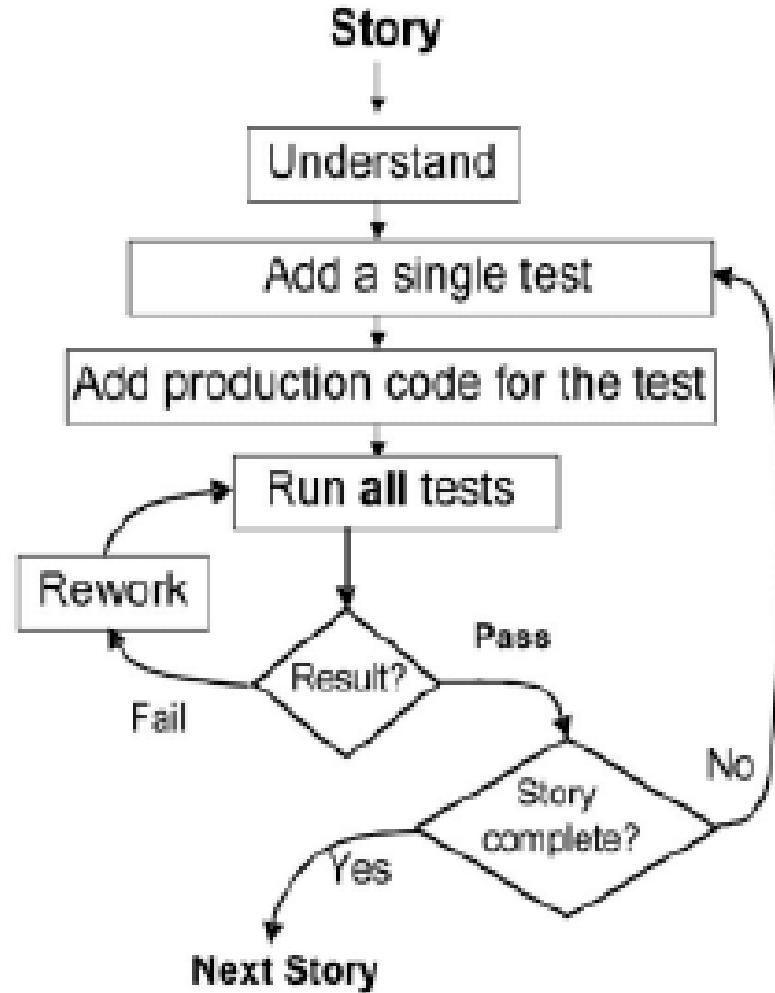
- **Popularized by Kent Beck (2003)**
- TDD completely turns traditional development around



# Test-first versus test-last development



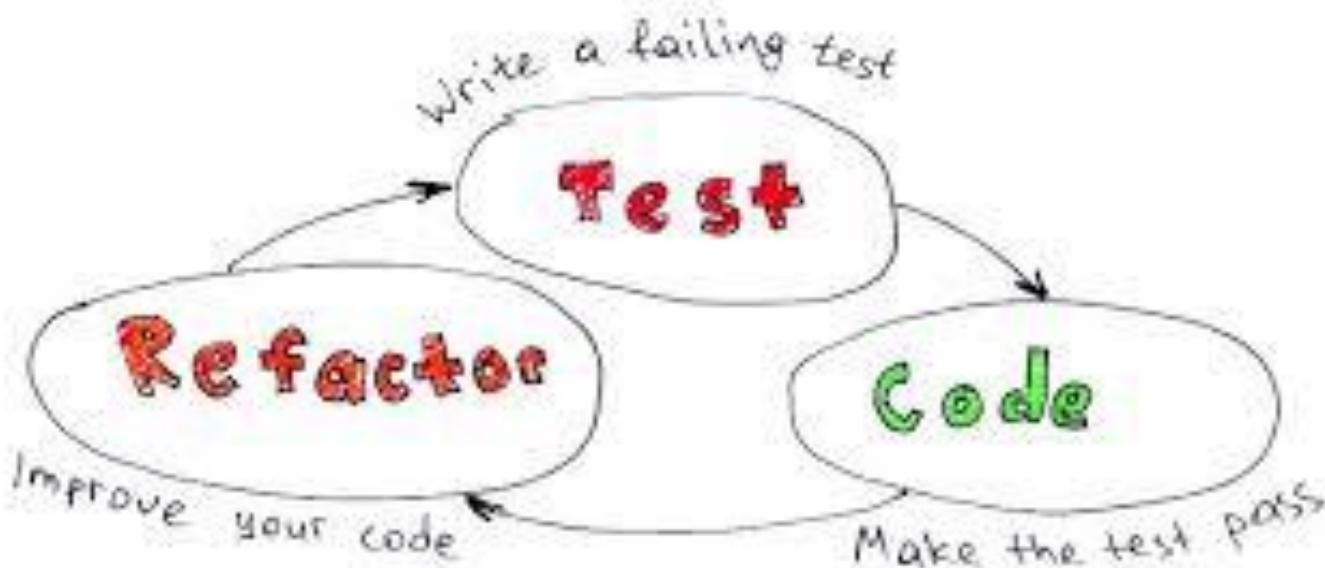
Test - Last



Test - First

# Test-Driven-Development

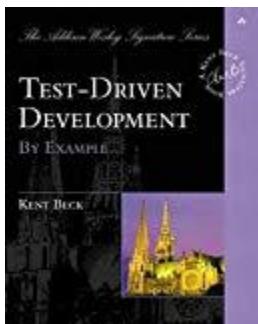
- To think through your requirements/design before you write your functional code



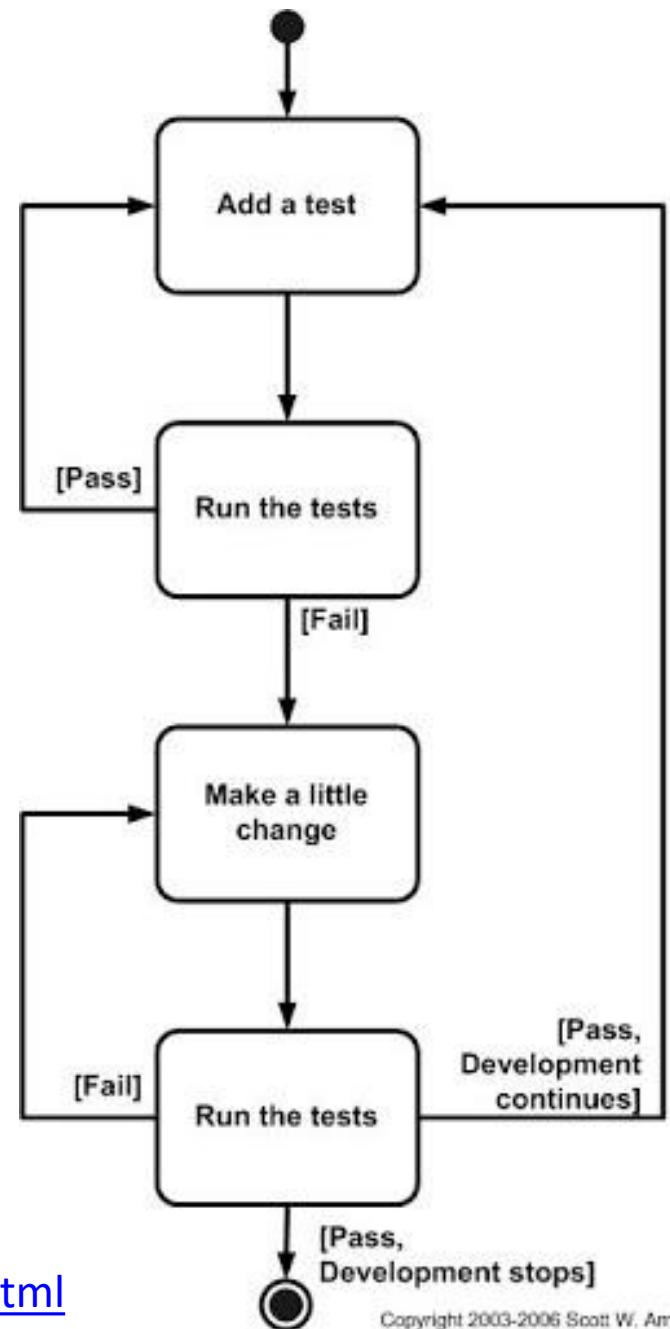
# TDD cycle

TDD works in small iterations

- 1.add a test
- 2.run all tests and watch the new one fail
- 3.make a small change
- 4.run all tests and see them all succeed
- 5.refactor (as needed)

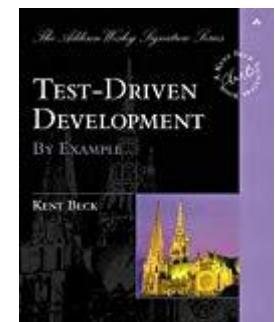
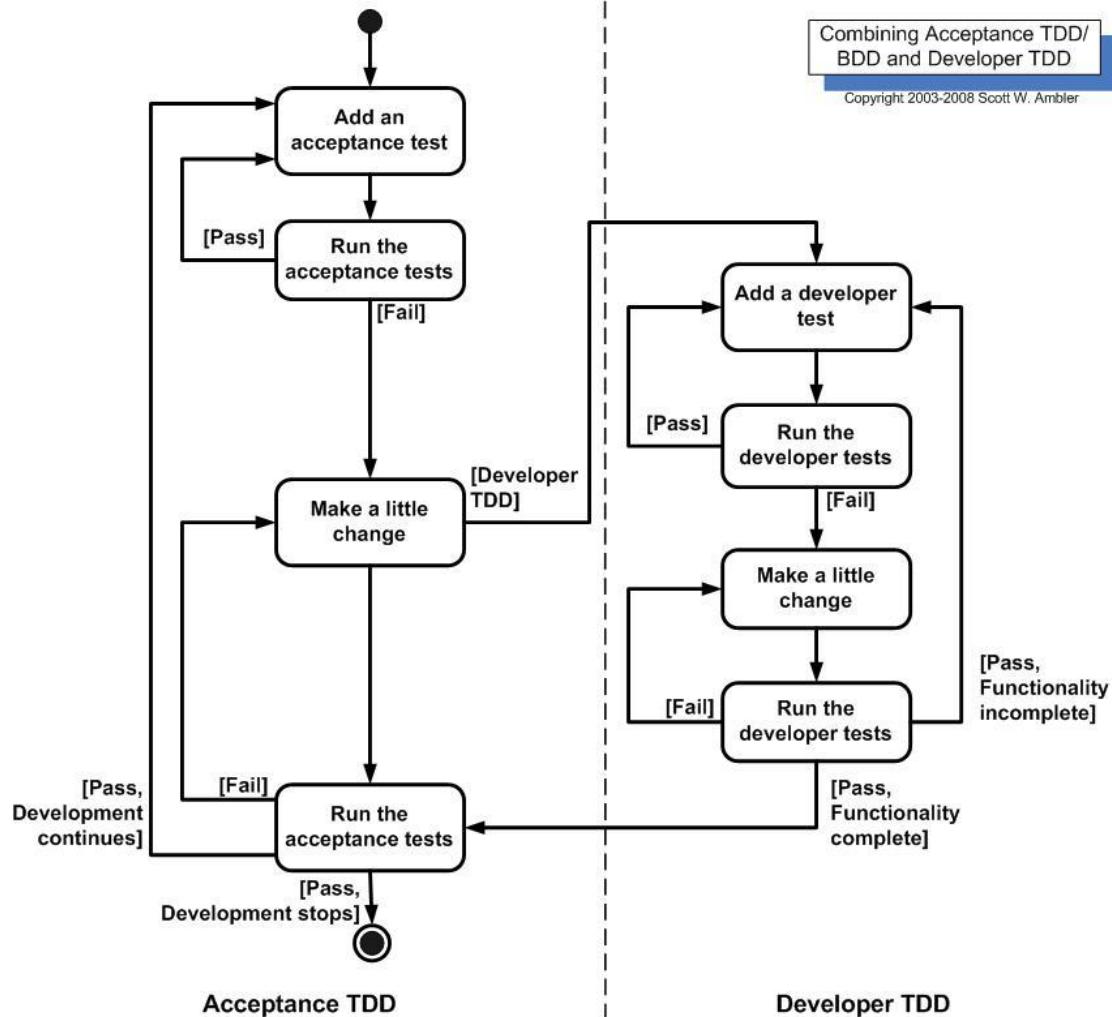


<http://www.agiledata.org/essays/tdd.html>



Copyright 2003-2006 Scott W. Ambler

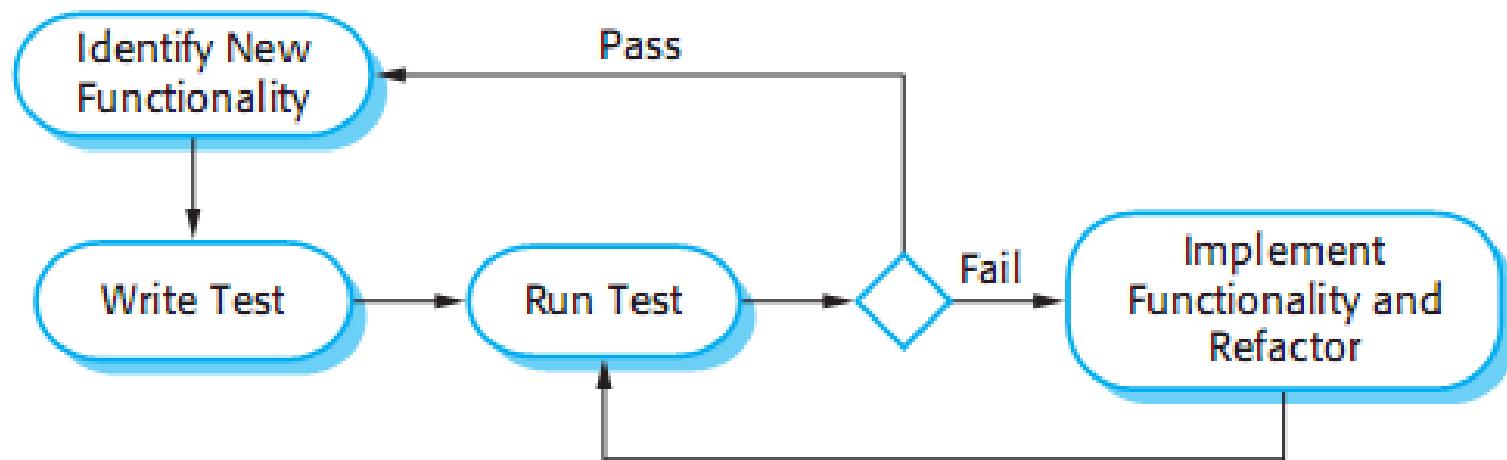
# ATDD cycle



<http://www.agiledata.org/essays/tdd.html>

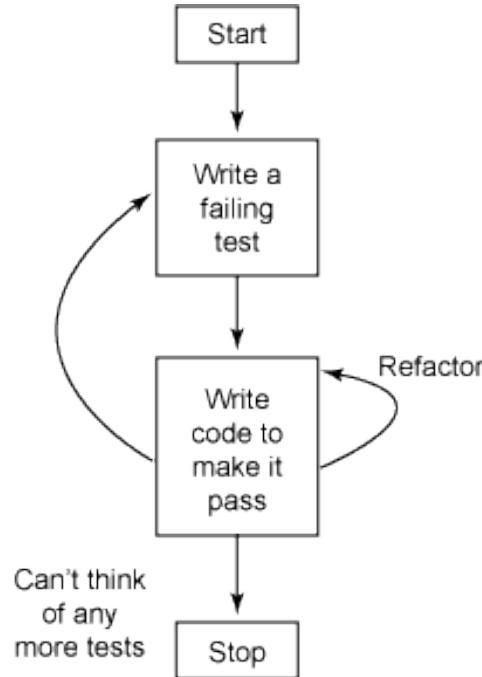
# Test-driven development

- Test-driven development (TDD) is an approach to program development in **which you inter-leave testing and code development.**
- You **develop code incrementally**, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test



# Test-driven development

- Tests are written before code and ‘passing’ the tests is the critical driver of development.

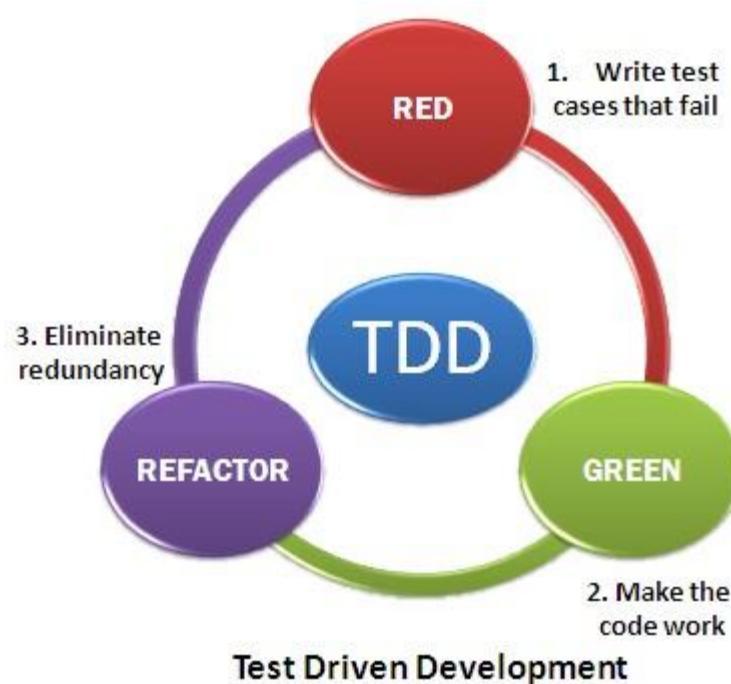


# TDD

**Red:** Create a test and make it fail.

**Green:** Make the test pass by any means necessary.

**Refactor:** Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.



# TDD example

US: *As a bank customer I want to check the strength of my password so that I don't get hacked easily*

AC: *A password should have between 5 and 10 characters*

# TDD example

US: As a bank customer I want to check the strength of my password so that I don't get hacked easily

AC: A password should have between 5 and 10 characters

```
package Prac;

public class PasswordValidator {
    public boolean isValid(String Password)
    {
        if (Password.length()>=5 && Password.length()<=10)
        {
            return true;
        }
        else
            return false;
    }
}
```

This is main condition checking length of password. If meets return true otherwise false.

```
package Prac;

import org.testng.Assert; ←
import org.testng.annotations.Test;

public class TestPassword { ← Needed for TestNG
    @Test ←
    public void TestPasswordLength() {
        PasswordValidator pv = new PasswordValidator();
        Assert.assertEquals(true, pv.isValid("Abc123"));
    }
}

package Prac;

public class PasswordValidator {
    public boolean isValid(String Password)
    {
        if (Password.length()>=5 && Password.length()<=10)
        {
            return true;
        }
        else
            return false; ← This is main condition checking length of password. If meets return true otherwise false.
    }
}
```

# TDD cycle: 1 step: add a test

**The input "I", the program must return 1.**

```
public class RomanNumberConverterTest {  
    @Test  
    void shouldUnderstandSymbolI() {  
        RomanNumeralConverter roman = new RomanNumeralConverter(); //fail  
        int number = roman.convert("I");  
        assertThat(number).isEqualTo(1);  
    } }
```

# TDD cycle: step: write a code

**The input "I", the program must return 1.**

```
public class RomanNumeralConverter {  
    public int convert(String numberInRoman)  
    {  
        return 0; //expected 1 //fail  
    } }
```

# TDD cycle: step: from fail to pass

**The input "I", the program must return 1.**

```
public class RomanNumeralConverter {  
  
    public int convert(String numberInRoman) {  
  
        if (numberInRoman.equals("I"))  
  
            return 1;  
        return 0;  
    } }
```

# TDD cycle: step: add a new test

**The input „V“, the program must return 5.**

```
@Test  
void shouldUnderstandSymbolV() {  
    RomanNumeralConverter roman = new  
RomanNumeralConverter();  
    int number = roman.convert("V");  
assertThat(number).isEqualTo(5);  
}
```

# TDD cycle: step: from fail to pass

**The input „V”, the program must return 5.**

```
public class RomanNumeralConverter {  
  
    public int convert(String numberInRoman) {  
  
        if(numberInRoman.equals("I"))    return 1;  
        if(numberInRoman.equals("V"))    return 5;  
  
        return 0;  
    } }
```

# Whye need fail test?

1. it verifies the test works, including any testing harnesses,
2. demonstrates how the **system will behave if the code is incorrect.**

# TDD process activities

- Start by **identifying the increment of functionality** that is required.
  - This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this **as an automated test**.
- Run the test, **along with all other tests** that have been implemented.
  - Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and **re-run the test**.
- Once all tests run successfully, you move on to implementing the **next chunk of functionality**.

# TDD Limitations

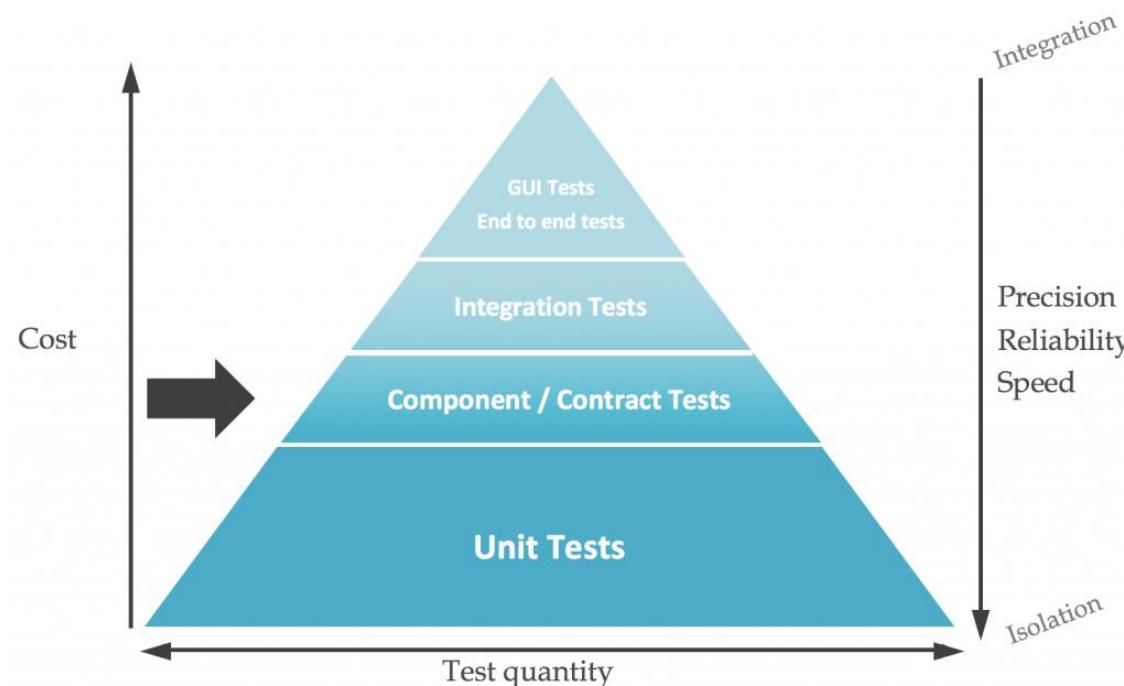
- Non-productive and **hard to learn**
- Difficult in Some Situations
  - **GUIs, Relational Databases, Web Service**
  - Requires mock objects
- TDD does not often include an upfront design
  - Focus is on implementation and **less on the logical structure**
- Difficult to write test cases for hard-to-test code
  - **Requires a higher level of experience from programmers**
- **TDD merge distinct phases of software development**
  - design, code and test

# Vienetų testavimas ir kodo padengimas testais

Dr. Asta Slotkienė

# Unit test

- Test individual pieces in isolation
- Focused, low-level; test individual methods or pieces of methods.
- Control: easier to ensure that each piece of code is tested



# Unit test

- A unit test is an **automated piece of code** that invokes a unit of work in the system. And a unit of work can span a **single method**, a whole **class** or **multiple classes** working together to achieve one single logical purpose that can be verified.

```
public class MyUnit {  
    public String concatenate(String one, String two) {  
        return one + two;  
    } }
```

```
import org.junit.Test;  
import static org.junit.Assert.*;  
public class MyUnitTest {  
  
    @Test  
    public void testConcatenate() {  
        MyUnit myUnit = new MyUnit();  
        String result = myUnit.concatenate("one", "two");  
        assertEquals("onetwo", result);  
    } }
```

# Python vieneto testo pavyzdys

```
import unittest

class Calculator:

    def __init__(self):
        pass

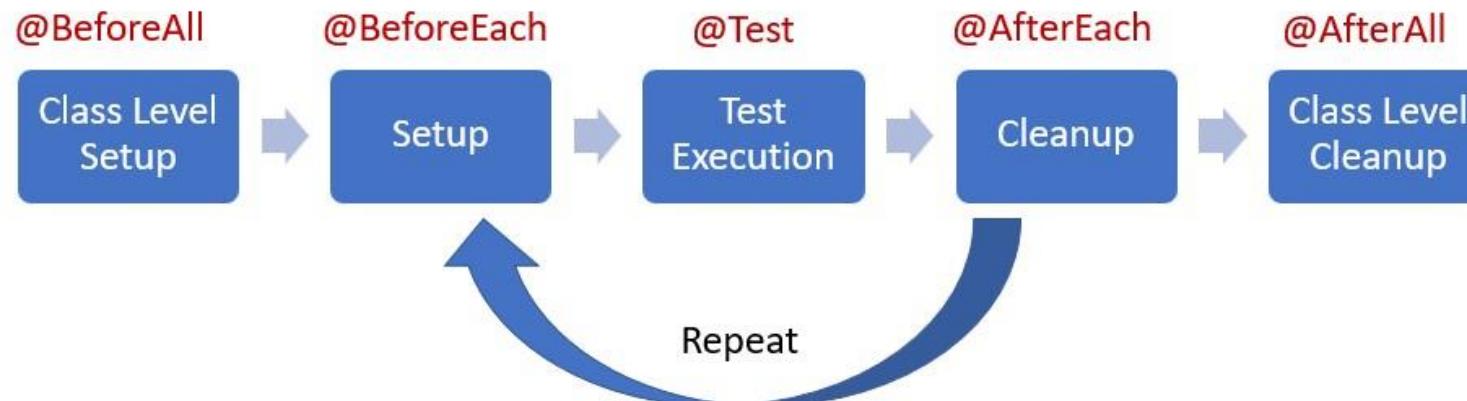
    def add(self, a, b):
        return a + b

class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.calc = Calculator()
        result = self.calc.add(4, 7)
        expected = 11
        self.assertEqual(result, expected)
```

# Unit test framework

- Python:
  - <https://docs.python.org/3.9/library/unittest.html#test-cases>
- Java
  - <https://junit.org/junit5/docs/current/user-guide/>



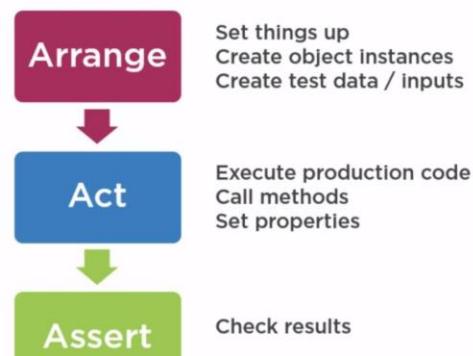
# Assert...

- `assertEqual(a,b)`       $a == b$
- `assertNotEqual(a,b)`     $a != b$
- `assertTrue(x)` - `bool(x)` is True
- `assertFalse(x)` - `bool(x)` is False
- `assertIs(a, b)`           $a \text{ is } b$
- `assertIsNot(a, b)`        $a \text{ is not } b$
- `assertIsNone(x)`         $x \text{ is } \text{None}$
- `assert IsNotNone(x)`     $x \text{ is not } \text{None}$
- `assertIn(a, b)`           $a \text{ in } b$
- `assertNotIn(a, b)`        $a \text{ not in } b$
- `assertGreater(a, b)`      $a > b$
- `assertLess(a, b)`         $a < b$

# Unit test pattern

- **The Arrange-Act-Assert pattern**

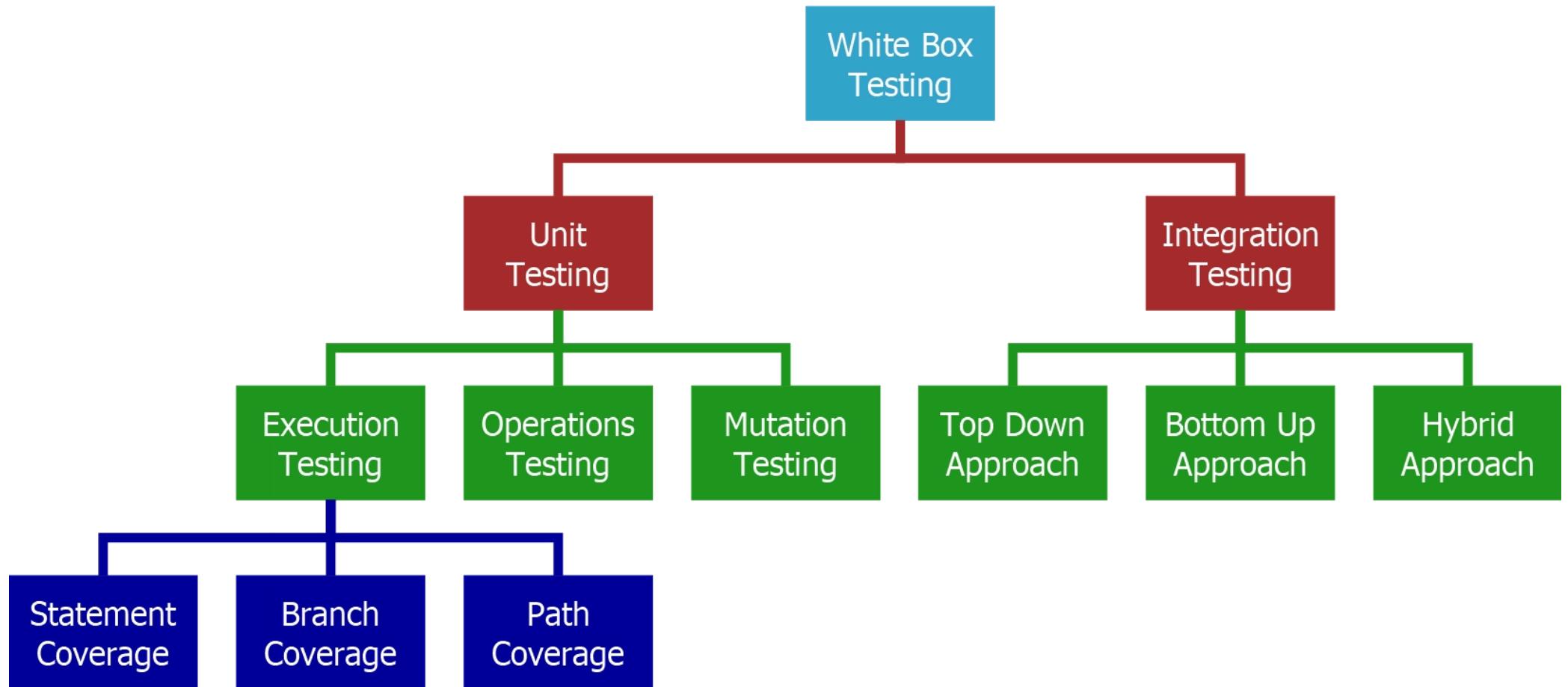
The Logical Phases of an Automated Test



```
@Test
void testPlus() {
    //Arrange
    var cash = new Cash(3);
    //Act
    cash.plus(4);
    //Assert
    assertEquals(7, cash.count());
}
```

# Baltos dėžės testavimas

## Types of White Box Testing



# Ivesties->Išvesties testavimas: kur problema?

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Ivestis:* x=[5, 10, 0]

*Tikėtinas rezultatas=?*

*Gautas rezultatas=?*

# Ivesties->Išvesties testavimas

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Ivestis:* x=[5, 10, 0]

*Tikėtinis rezultatas=1*

*Gautas rezultatas=1*

# Ivesties->Išvesties testavimas

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Input: x=[5, 10, 0]*

*Expected result=1*

*Actual result=1*

*Error=YES*

*Failure=NO*

# Ivesties->Išvesties testavimas

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Ivestis: x=[0, 5, 10]*

*Tikėtinis rezultatas=?*

*Gautas rezultatas=?*

# Ivesties->Išvesties testavimas

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Ivestis: x=[0, 5, 10]*

*Tikėtinis rezultatas=1*

*Gautas rezultatas=?*

# Ivesties->Išvesties testavimas

```
public static int numZero(int[] x) {  
    int count=0;  
    for (int i=1; i<x.length; i++) {  
        if (x[i]==0)  
            count++;  
    }  
    return count;  
}
```

*Ivestis:* x=[0, 5, 10]

*Tikėtinis rezultatas=1*

*Gautas rezultatas=0*

*Failure=YES*

# Vieneto testavimo algortimas

- 1) Pasirinkti kodo dalį
- 2) Pasirinkti testinj duomenj/is
- 3) Numatyti tiketiną rezultataj
- 4) Vykdyti testą
- 5) Palyginti gautus rezultatus su tiketiniais

# Kodo padengimas testais

- Kodo padengimas testais tai procentinis rodiklis, parodantis, kokia kodo dalis yra ištestuojama testais (unit tests)



- *Code coverage* is the extent to which a given test suite executes the source code of the software.

# Testiniai atvejai

- Perėjimai, galimi sprendimo keliai (Control flow)
  - Grafai
- Duomenų srautai (Data flow)
  - Įvesties duomenys
  - Konfigūraciniai duomenys
  - Generuojami duomenys

# Baltos dėžės testavimo technikos

- Sakinių (statement, Node) padengimas
- Išsišakojimų padengimas (branch=!desicions)
- Ciklo padengimai
- Sąlygų padengimai
- Sprendimo kelių padengimas (Path)
- Įvairios kombinacijos

Preference for types of code coverage



Line coverage



Statement coverage



Decision coverage



Branch coverage



Path coverage

# Padengimas sakiniais (statements)

- Skaičiuojama kiek kodo sakinių padengė įvykdymas testas
- Kodo padengiamumas=įvykdytų sakinių kiekis/visų sakinių kiekis

- Kodo padengiamumas=3/5 ->60%

```
metai = int(input().strip())
if metai<85:
    print(metai, "draudziamas")
else:
    print(metai, "nedraudziamas")
```

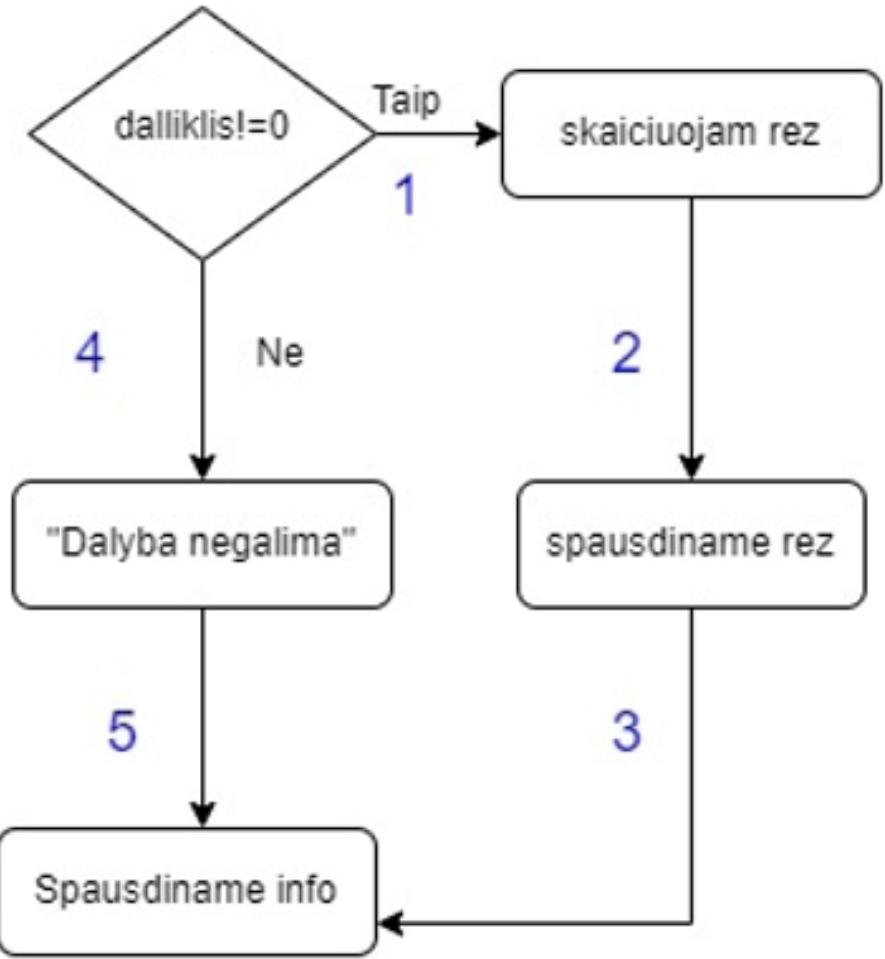
# Programos išsišakojimų (branch) padengimas

- Išsišakojimų padengimas=jvykdyti išsisakojimai/visų išsisakojimų kiekis

```
sk = int(input().strip())
daliklis= int(input().strip())
if daliklis!=0:
    rez=sk/daliklis
    print("Rezulatas", rez)
else:
    print("Dalyba negalima")
print ("Ivesti duomenys dalybai:", "skaicius ", sk, " daliklis ", daliklis)
```

# Programos išsišakojimų (branch) padengimas

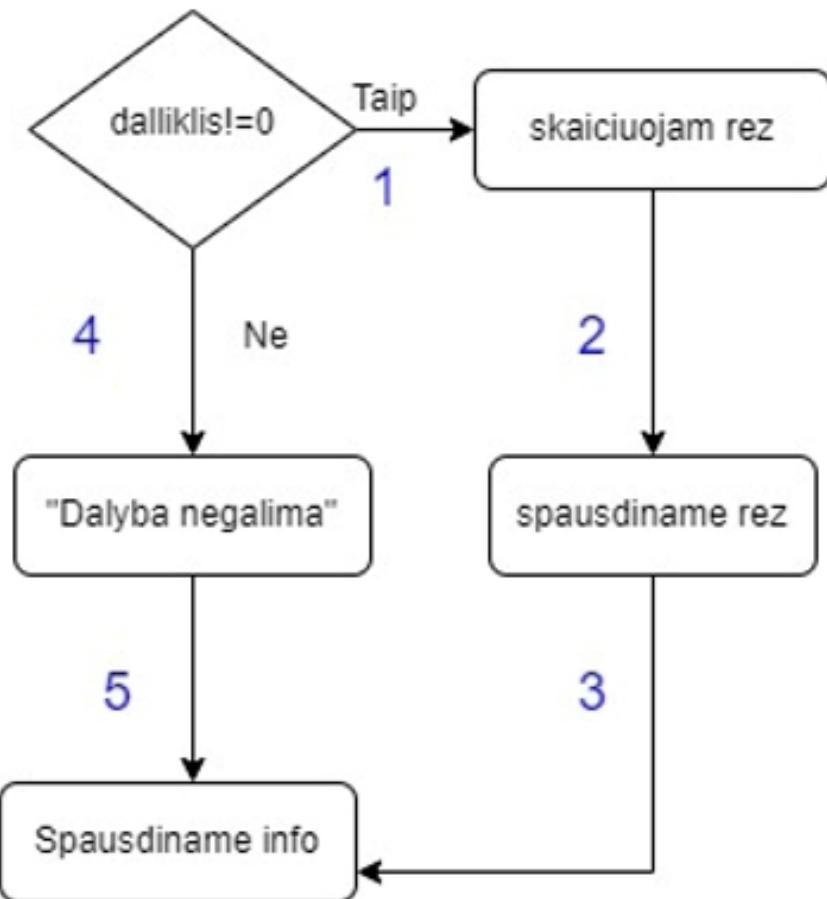
```
sk = int(input().strip())
daliklis= int(input().strip())
if daliklis!=0:
    rez=sk/daliklis
    print("Rezulatas", rez)
else:
    print("Dalyba negalima")
print ("Ivesti duomenys dalybai:", "skaicius ", sk, " daliklis ", daliklis)
```



# Programos išsišakojimų(branch) padengimas

```
result = (15, 5)
expected = 3
Padengiamumas>3/5 =60%
```

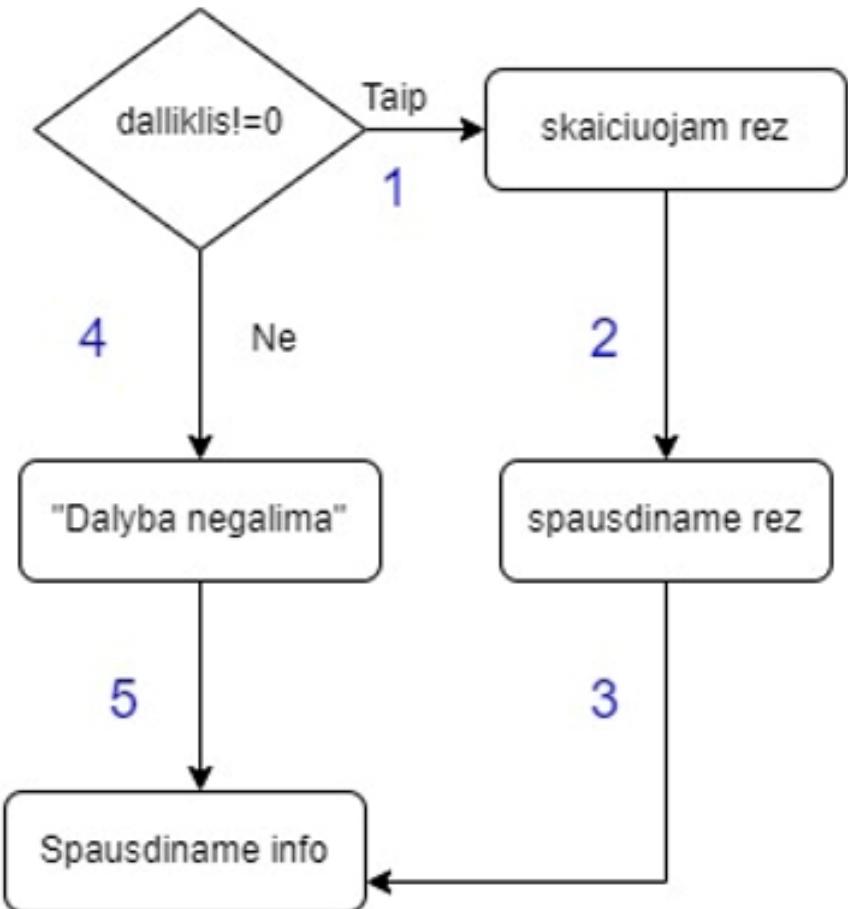
```
sk = int(input().strip())
daliklis= int(input().strip())
if daliklis!=0:
    rez=sk/daliklis
    print("Rezulatas", rez)
else:
    print("Dalyba negalima")
print ("Investi duomenys dalybai:", "skaicius ", sk, " daliklis ", daliklis)
```



# Programos išsišakojimų(branch) padengimas

```
result = (15, 0)
expected = "Dalyba negalima"
Padengiamumas>2/5 =40%
```

```
sk = int(input().strip())
daliklis= int(input().strip())
if daliklis!=0:
    rez=sk/daliklis
    print("Rezulatas", rez)
else:
    print("Dalyba negalima")
print ("Ivesti duomenys dalybai:", "skaicius ", sk, " daliklis ", daliklis)
```



# Programos išsišakojimų(branch!=decisions) padengimas

- 1 sprendimas: 1->2>3 → 50%
- 2 sprendimas: 4->5 → 50%

```
sk = int(input().strip())
daliklis= int(input().strip())
if sk>0:
    if daliklis!=0:
        rez=sk/daliklis
        print("Rezulatas", rez)
    else:
        print("Dalyba negalima")
print ("Investi duomenys dalybai:", "skaicius ", sk, " daliklis ", daliklis)
```



# Užduotis

- Įvestis-> 100, 100
- Rezultatas-> Klaida, 200
- Sakinių padengimas ?/7
- Išsišakojimų padengimas
- Sprendimų padengimas

```
x= int(input().strip())
y= int(input().strip())
if  x - 100 <= 0 :
    if  y - 100 <= 0 :
        if  x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

# Užduotis

- TA1: 5 (int), 100 (int)
- TA2: `nextInt`, 5 (int)
- TA3: 5 (int), `nextInt`
- TA4: `nextInt`, `nextInt`

```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

	TA1	TA2	TA3	TA4
X (sveikas skaičius)	T	K	T	K
X (sveikas skaičius)	T	T	K	K
Tikėtinas rezultatas	Apskaičiuojama	Error	Error	Error

# Ar pakanka juodos dėžės technikos?

TA1: 100, 100

TA2: nelnt, 5

TA3: 5, nelnt

TA4: nelnt, nelnt

**Sakinių padengimas 7/7=100%**

```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

# Ar pakanka juodos dėžės technikos?

TA1: 5, 100

TA2: nelnt, 5

TA3: 5, nelnt

TA4: nelnt, nelnt

- Sakinių padengimas 6/7=85%

Nepakanka!

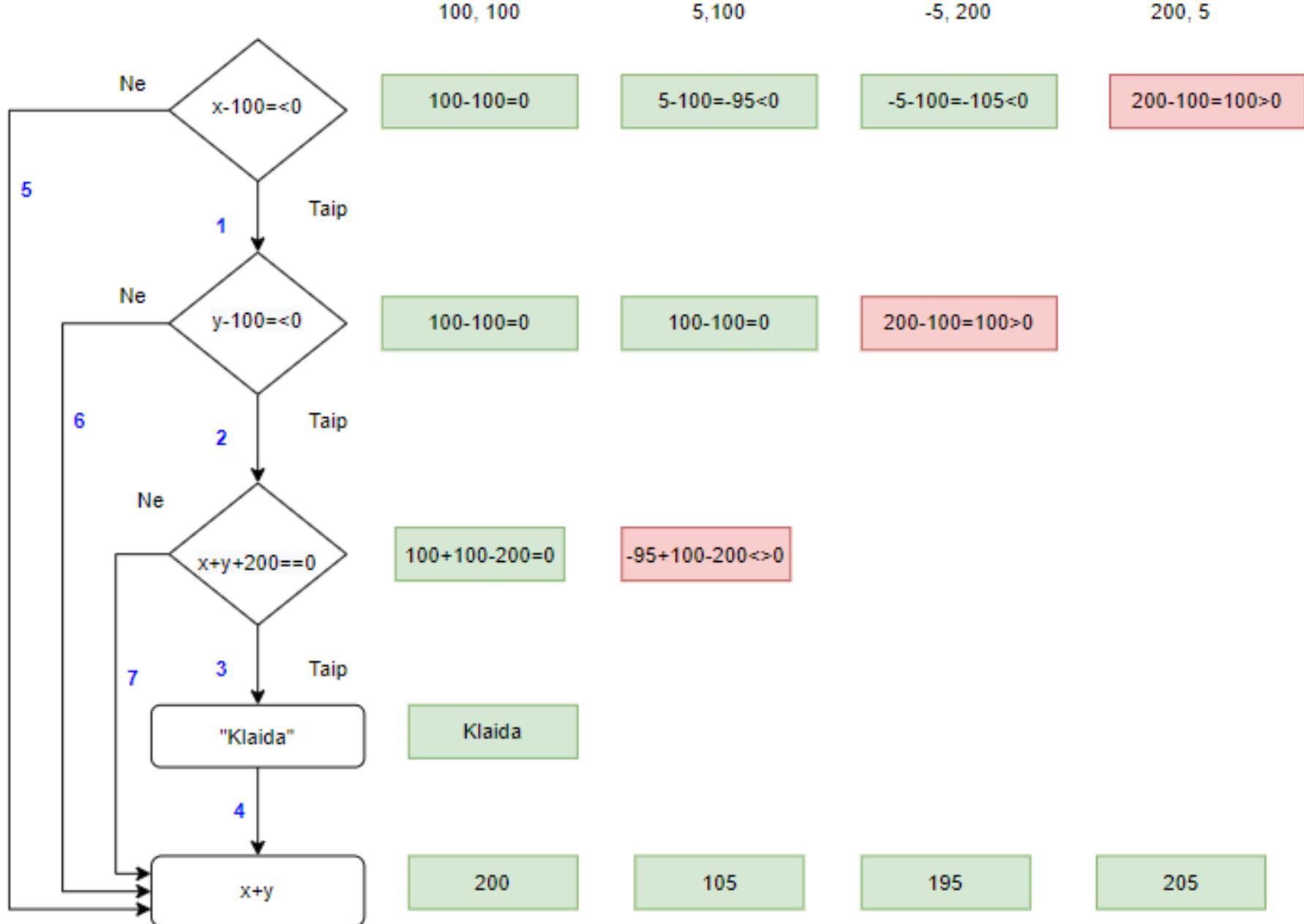
Sprendimas -> Taikyti Išsišakojimų ir sprendimų padengimą

```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

# Užduotis

- Įvestis-> 100, 100
- Sakinių padengimas: 7/7
- Išsišakojimų padengimas:
- Sprendimų padengimas:

```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```



# Užduotis

- Įvestis-> 100, 100
- Sakinių padengimas: 7/7
- Išsišakojimų padengimas: 4/7
- Sprendimų padengimas: 1/4

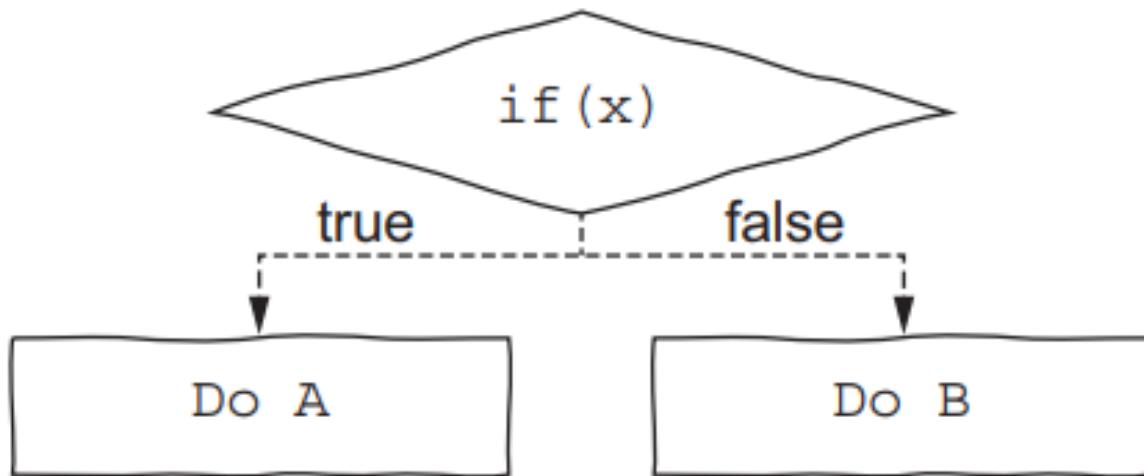
```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

# Užduotis

```
x= int(input().strip())
y= int(input().strip())
if x - 100 <= 0 :
    if y - 100 <= 0 :
        if x + y - 200 == 0:
            print("Klaida")
print(x + y);
```

Kodo padengiamumas	100, 100	5,100	-5, 100	200,5
Sakinių padengimas	100% (7/7)	86% (6/7)	71% (5/7)	28% (2/7)
Išsišakojimų padengimas	57% (4/7)	57% (3/7)	57% (4/7)	57% (1/7)
Sprendimų padengimas	25% (1/4)	25% (1/4)	25% (1/4)	25% (1/4)

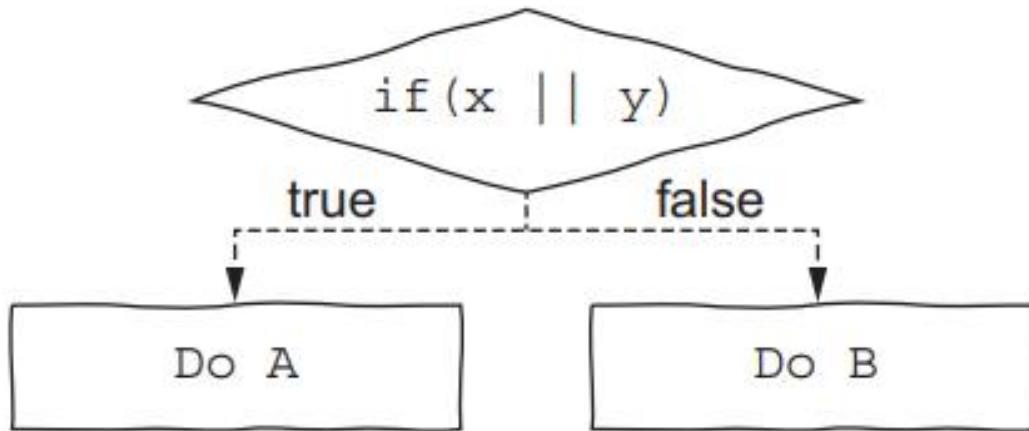
# Branch coverage



Suppose a test T1 makes the if true.

- Line coverage:  $2/3 = 66.6\%$
- Branch coverage:  $1/2 = 50\%$

# Condition + Branch coverage

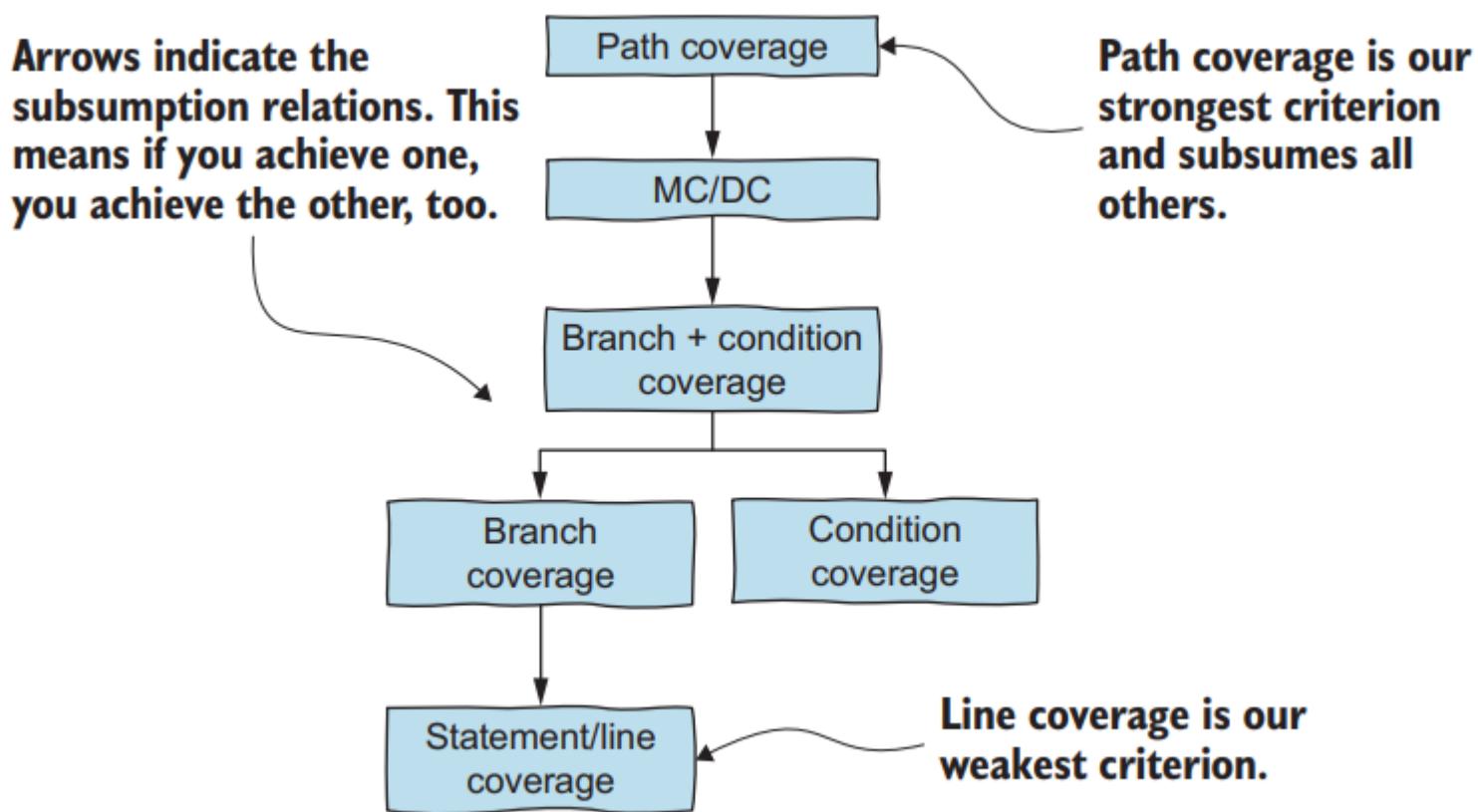


Imagine a test T1 where  $x = \text{true}$ .

- Line coverage:  $2/3 = 66.6\%$
- Branch coverage:  $1/2 = 50\%$
- Branch + condition coverage:  $(1 + 2)/(2 + 4) = 50\%$

$$\text{c+b coverage} = \frac{\text{branches covered} + \text{conditions covered}}{\text{number of branches} + \text{number of conditions}} \times 100\%$$

# The different coverage criteria



# Advanced Condition Coverage

- Condition/Decision Coverage (C/DC)
  - as DC plus: **every condition in each decision is tested in each possible outcome**
- Modified Condition/Decision coverage (MC/DC)
  - as above plus, **every condition shown to independently affect a decision outcome** (by varying that condition only)
  - Def: A condition independently affects a decision when, by flipping that condition's outcome and holding all the others fixed, the decision outcome changes
  - this criterion was created at Boeing and is required for aviation software according to RCTA/DO-178B
- Multiple-Condition Coverage (M-CC)
  - **all possible combinations of condition outcomes within each decision is checked**

# Modified condition/decision coverage

## **MC/DC**

*“Each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions”.*

# Modified condition/decision coverage

## *MC/DC*

Complete requirements for an MC/DC test suite:

- All program **entry and exit points covered**
- **Each decision exercised on both branches**
- Each condition takes **both values**
- Each condition is shown to affect its enclosing decision

# if (a and b) or c

Test Case	a	b	c	outcome	
1	True	True	True	True	
2	True	True	False	True	
3	True	False	True	True	
4	True	False	False	False	
5	False	True	True	True	
6	False	True	False	False	
7	False	False	True	True	
8	False	False	False	False	

# if (a and b) or c: from a value

Test Case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
5	False	True	True	True
6	False	True	False	False
7	False	False	True	True
8	False	False	False	False

Test Case	a	b	c	outcome
2	True	True	False	True
6	False	True	False	False

# if (a and b) or c: from b value

Test Case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
5	False	True	True	True
6	False	True	False	False
7	False	False	True	True
8	False	False	False	False

Test Case a	b	c	outcome
2	True	False	True
4	True	False	False

# if (a and b) or c: from c value

Test Case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
5	False	True	True	True
6	False	True	False	False
7	False	False	True	True
8	False	False	False	False

Test Case	a	b	c	outcome
3	True	False	True	True
4	True	False	False	False

if (a and b) or c

Required N+1 test cases: T2, T3, T4. T6

Test Case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
4	True	False	False	False
5	False	True	True	True
6	False	True	False	False
7	False	False	True	True
8	False	False	False	False

# Example

- Counts the number of words in a string that end with either “r” or “s”
- Given a sentence, the program should count the number of words that end with either “s” or “r”. A word ends when a non-letter appears. The program returns the number of words

```
public class CountWords {  
    public int count(String str) {  
        int words = 0;  
        char last = ' ';  
        for (int i = 0; i < str.length(); i++) {  
            if (!isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {  
                words++;  
            }  
            last = str.charAt(i);  
            if (last == 'r' || last == 's') {  
                words++;  
            }  
        }  
        return words;  
    } }
```

# Unit testai 1

```
@Test
void twoWordsEndingWithS() {
    int words = new CountLetters().count("dogs cats");
    assertThat(words).isEqualTo(2);
}

@Test
void noWordsAtAll() {
    int words = new CountLetters().count("dog cat");
    assertThat(words).isEqualTo(0);
}
```

# Code Coverage

```
public class CountWords {
    public int count(String str) {
        int words = 0;
        char last = ' ';
        for (int i = 0; i < str.length(); i++) {
            if (!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
                words++;
            }
            last = str.charAt(i);
        }
        if (last == 'r' || last == 's')
            words++;
        return words;
    }
}
```

# Unit testai 12

```
@Test  
void wordsThatEndInR() {  
    int words = new CountWords().count("car bar");  
    assertThat(words).isEqualTo(2);  
}
```

# Code Coverage

```
public class CountWords {
    public int count(String str) {
        int words = 0;
        char last = ' ';
        for (int i = 0; i < str.length(); i++) {
            if (!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
                words++;
            }
            last = str.charAt(i);
        }
        if (last == 'r' || last == 's')
            words++;
        return words;
    }
}
```

# Modified condition/decision coverage

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

<https://www.youtube.com/watch?v=DivaWCNohdw>

[https://en.wikipedia.org/wiki/Modified\\_condition/decision\\_coverage](https://en.wikipedia.org/wiki/Modified_condition/decision_coverage)

# isLetter: T1 opposite T5: (T1;T5)

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

$\text{last} == \text{s}$  or  $\text{last} == \text{r}$  : `isLetter`: opposite  
(T2;T6)

Test case	isLetter	<code>last == s</code>	<code>last == r</code>	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

$\text{last} == \text{s}$  or  $\text{last} == \text{r}$  : isLetter: opposite  
(T3;T7)

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

$\text{last} == \text{s}$  or  $\text{last} == \text{r}$  : `isLetter`: opposite  
~~(T4;T8)~~: both the same outcome

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

# MC/DC coverage: N + 1 tests

isLetter:

opposite (T1;T5), (T2;T6), (T3;T7)

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

# MC/DC coverage: N + 1 tests

isLetter:

opposite (T1;T5), (T2;T6), (T3;T7)->T6 or T7

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

# MC/DC coverage: N + 1 tests

T2, T3, T4, T6 or T7

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

# Papildomai

- Reinforced Condition/Decision Coverage (RC/DC)
- ISTQB Technical Test Analyst | 2.4 Modified Condition/Decision Coverage (MC/DC) Testing
  - <https://www.youtube.com/watch?v=9i9xpxn6pzM>

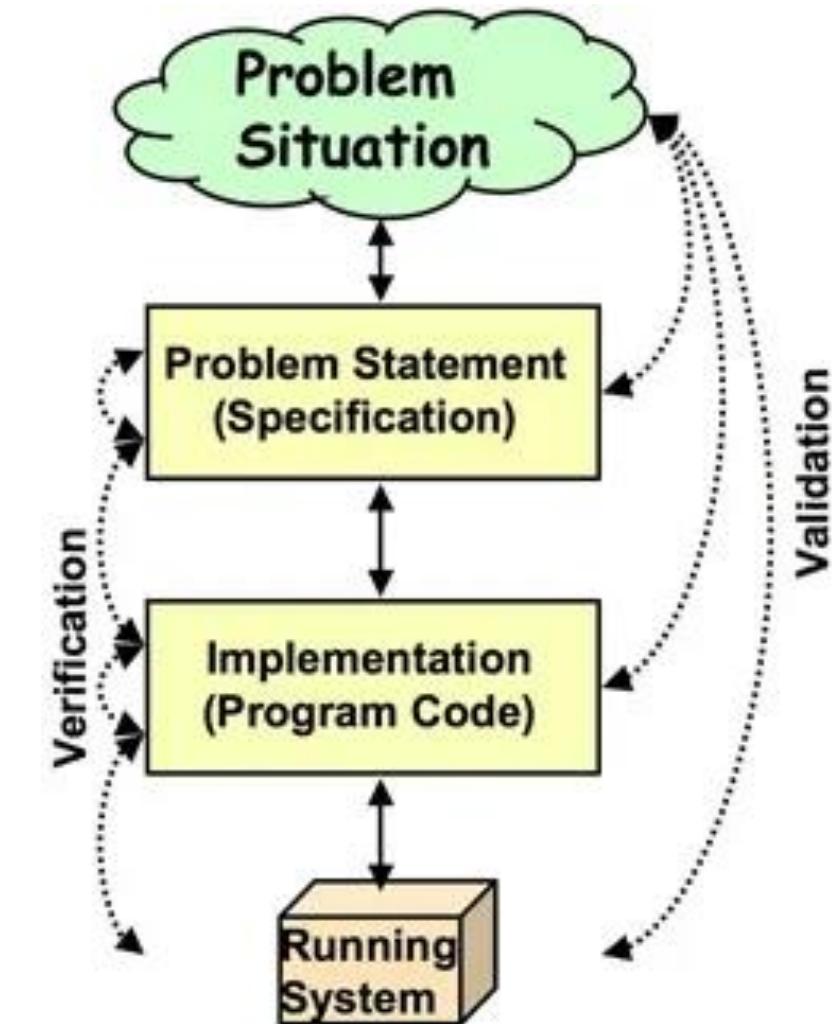
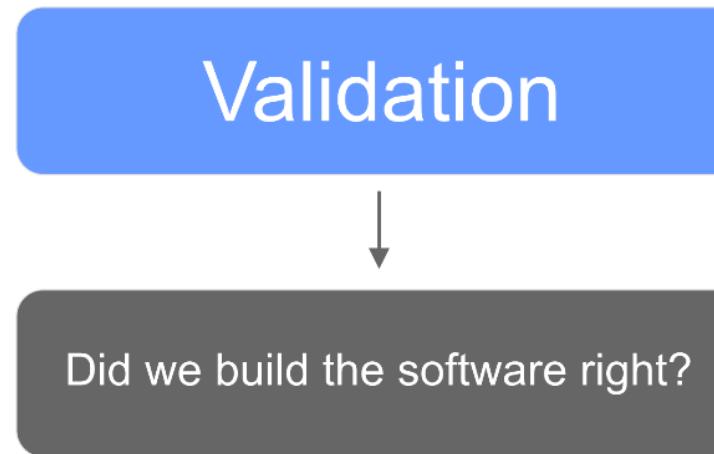
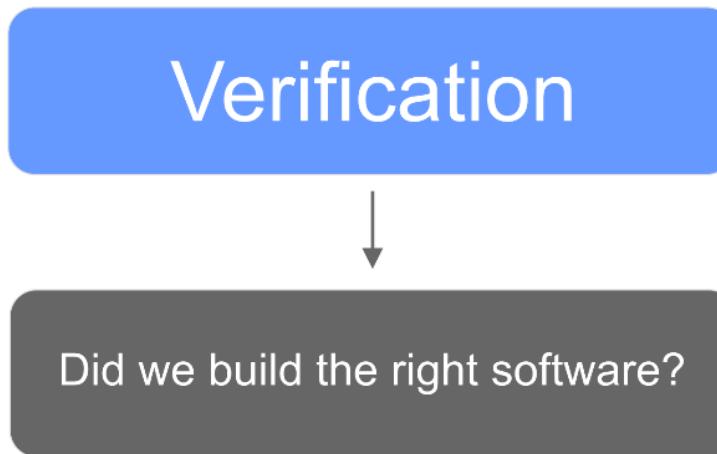
# PS kūrimo procesas

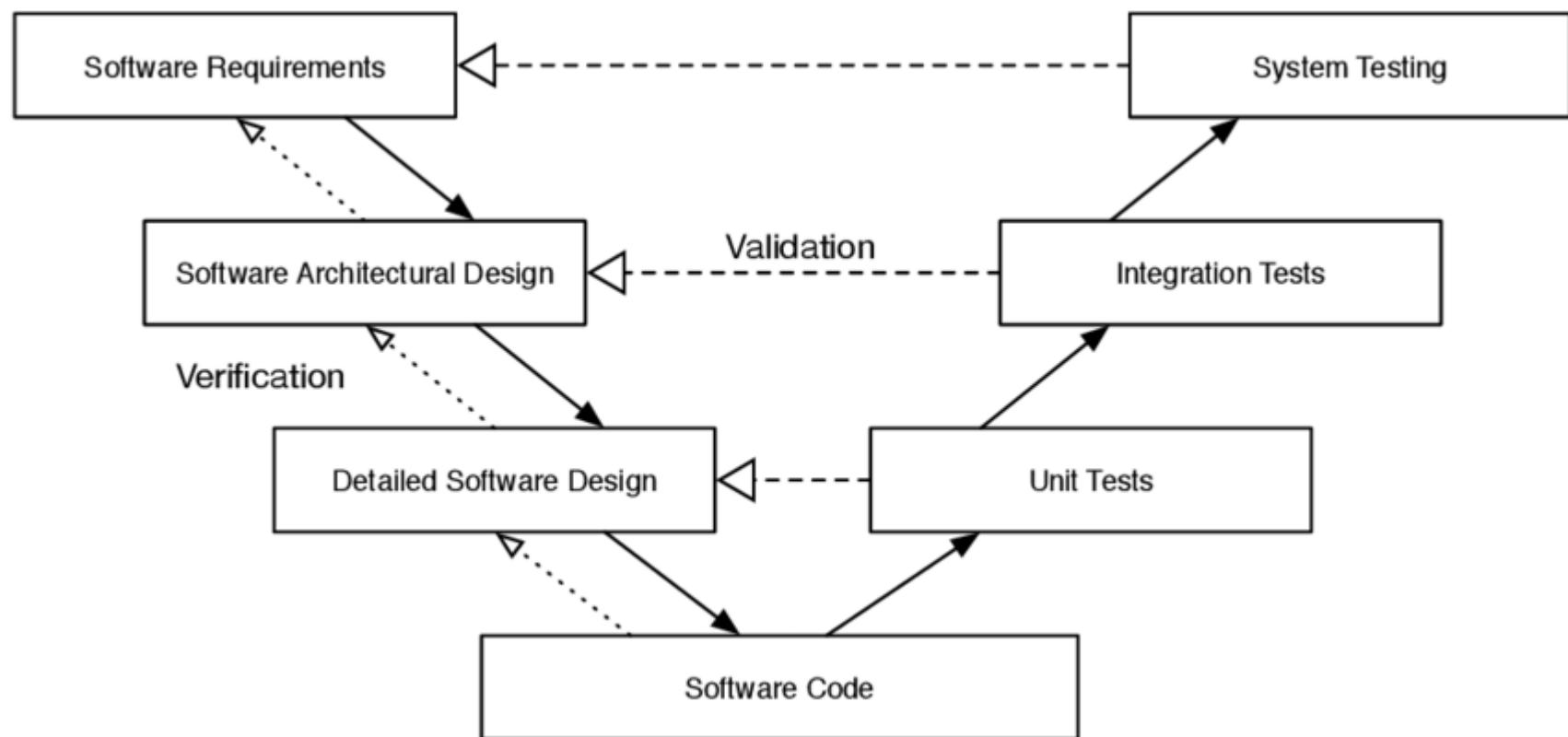
Dr. Asta Slotkienė

# CMMI-DEV Engineering PAs

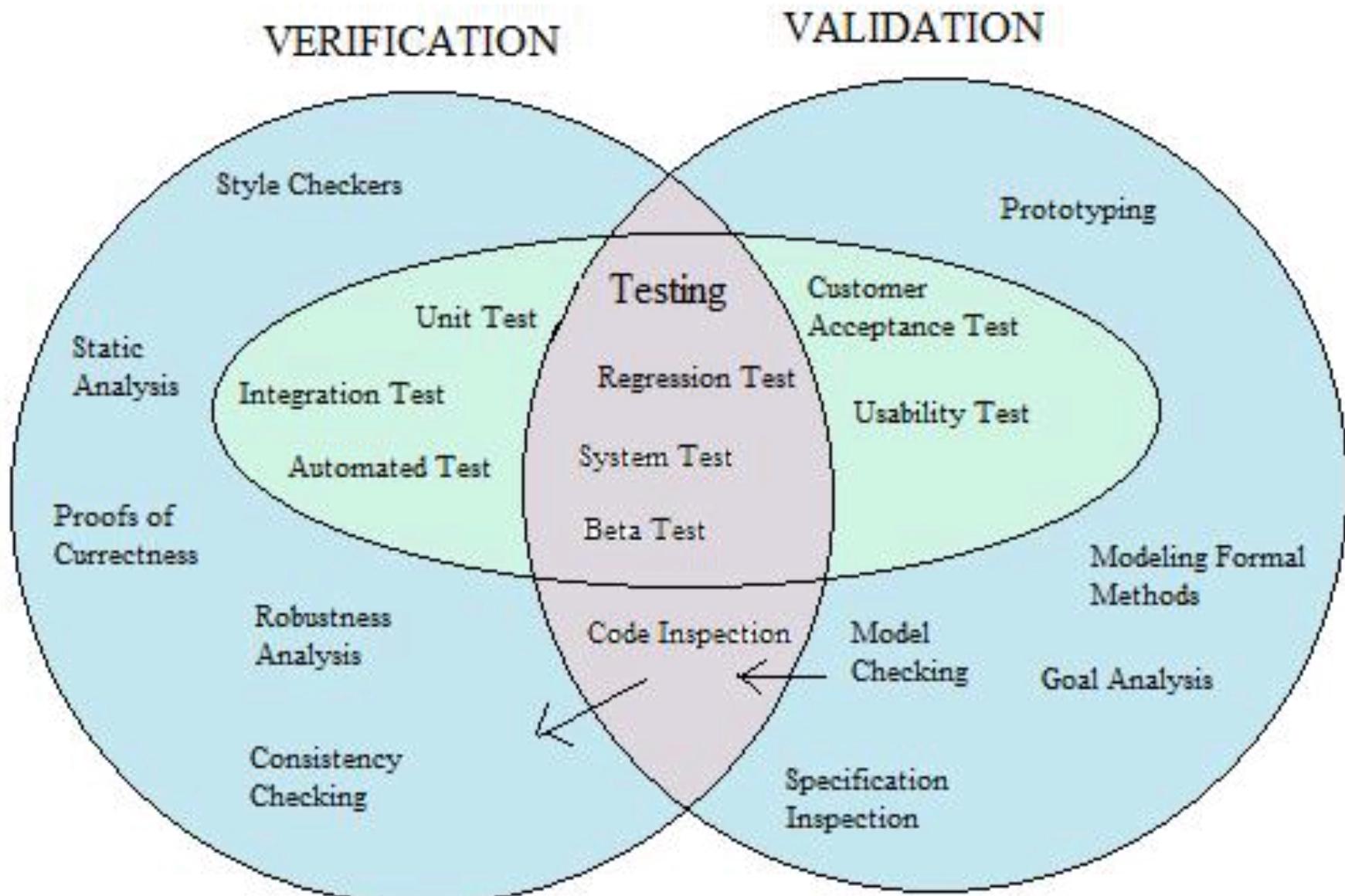
- Requirements Development (RD):
  - understanding what stakeholders think they need and **documenting that understanding for the people who will be designing solutions**
- Product Integration (PI):
  - **putting together all the product components** so that the overall product has expected behaviors and characteristics
- Technical Solution (TS):
  - using effective engineering **to build solutions that meet end user needs**
- Verification (VER):
  - **making sure that the solution you ended up with meets your agreement** about the needs
- Validation (VAL):
  - **making sure that the solution actually meets the needs of users** in the service environment

# Validation versus Verification





# Validation versus Verification



# CMMI-DEV ->Verification

- The purpose of Verification is to ensure that selected work products meet their specified requirements.
- The process area involves the following:
  - verification preparation,
  - verification performance
  - identification of corrective action.

# Validation definition

- The process of evaluating software during or **at the end of the development process** to determine whether it **satisfies specified (or implicit) business requirements**.
- To ensure that the **product actually meets the user's needs**,
- To ensure that the requirements were correct in the first place.

# Validation definition

- Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

<https://glossary.istqb.org/en/search/validation>

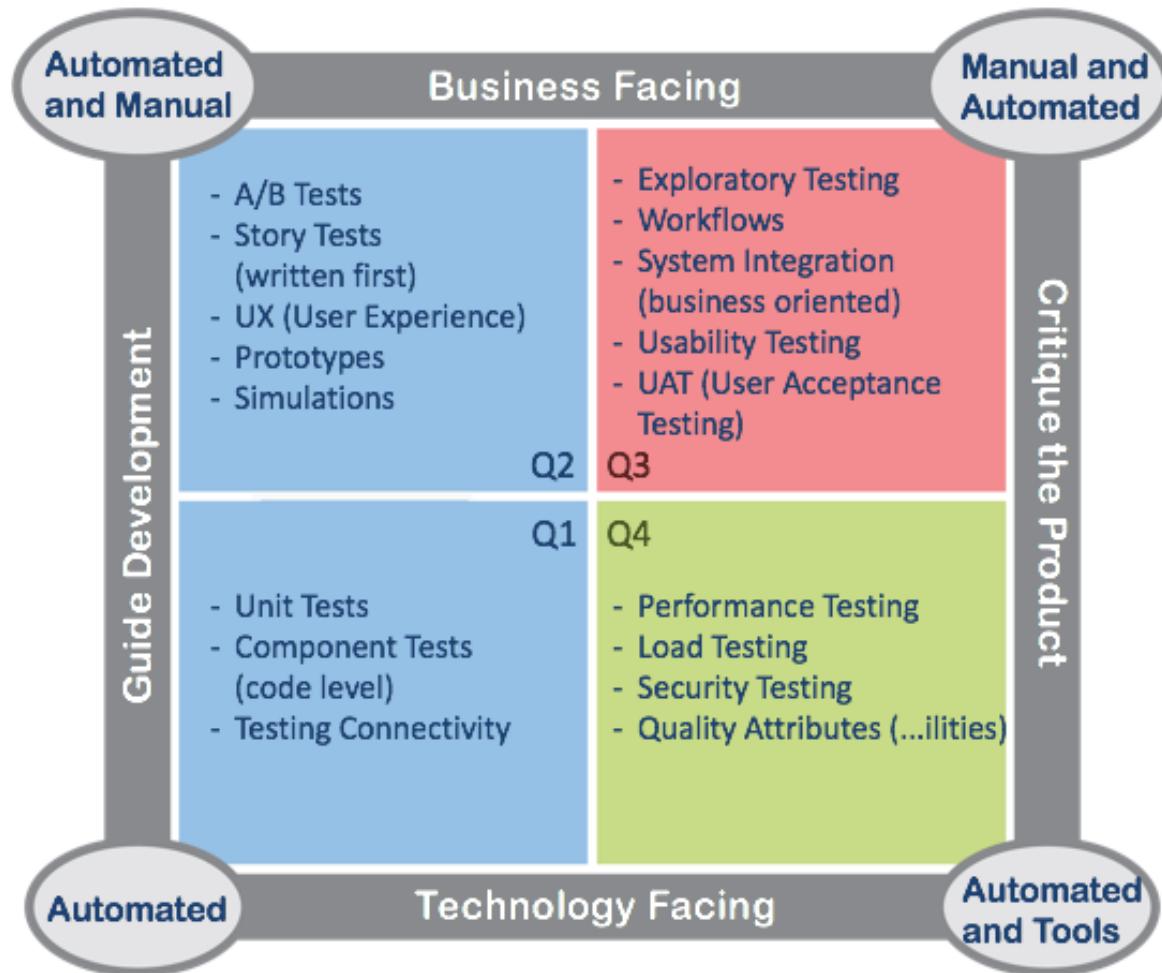
# CMMI-DEV -> Validation

- The purpose of Validation is to demonstrate that a product or product component fulfills its intended use when placed in its intended environment.

# CMMI-DEV -> Verification and Validation

In Engineering (DEV)	
<b>VER – Verification</b> <b>VAL – Validation</b>	<b>VER SP 1.1 Select Work Products for Verification</b> <b>VER SP 1.2 Establish the Verification Environment</b> <b>VER SP 1.3 Establish Verification Procedures and Criteria</b>
<b>VER SG 1</b> Preparation for verification is conducted. <b>VAL SG 1</b> Prepare for validation is conducted.	<b>VAL SP 1.1 Select Products for Validation</b> <b>VAL SP 1.2 Establish the Validation Environment</b> <b>VAL SP 1.3 Establish Validation Procedures and Criteria</b>
<b>VER SG 2</b> Peer reviews are performed on selected work products.	<b>VER SP 2.1 Prepare for Peer Reviews</b> <b>VER SP 2.2 Conduct Peer Reviews</b> <b>VER SP 2.3 Analyze Peer Review Data</b>
<b>VER SG 3</b> Selected work products are verified against their specified requirements.	<b>VER SP 3.1 Perform Verification</b> <b>VER SP 3.2 Analyze Verification Results</b>
<b>VAL SG 2</b> The product or product components are validated to ensure they are suitable for use in their intended operating environment.	<b>VAL SP 2.1 Perform Validation</b> <b>VAL SP 2.2 Analyze Validation Results</b>

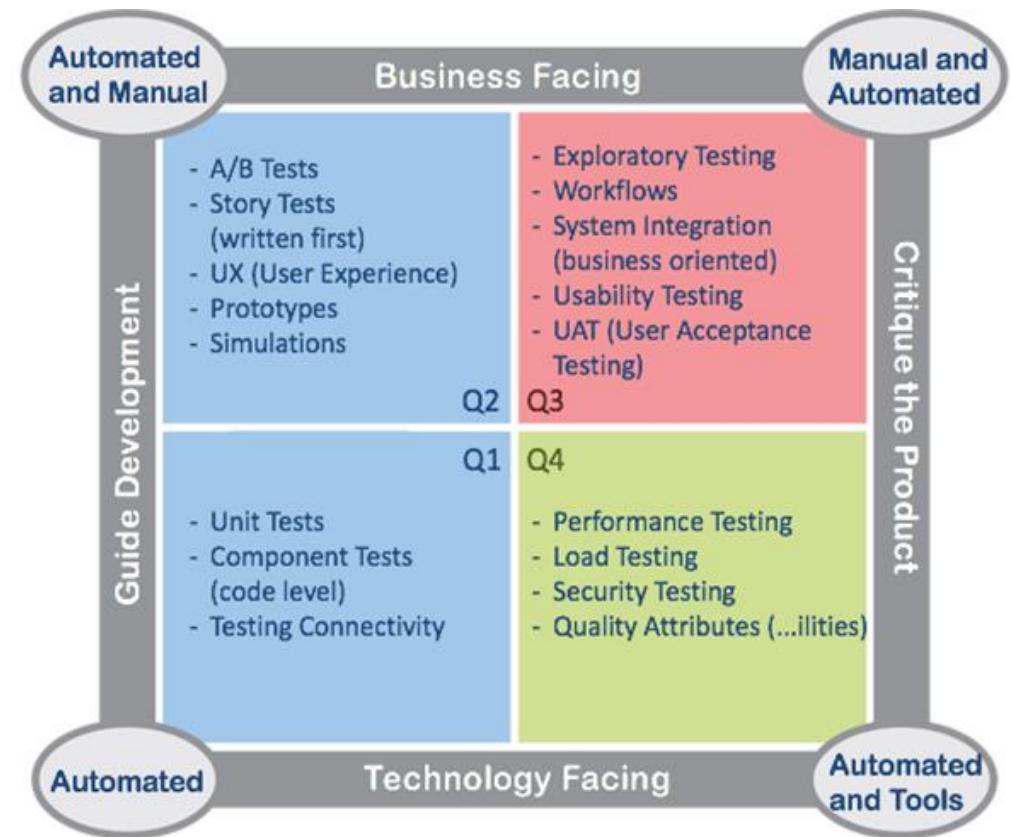
# Agile siūlomas testavimo procesas (workflow)



# Agile siūlomas testavimo procesas (workflow)

Q1 – Contains unit and component tests.

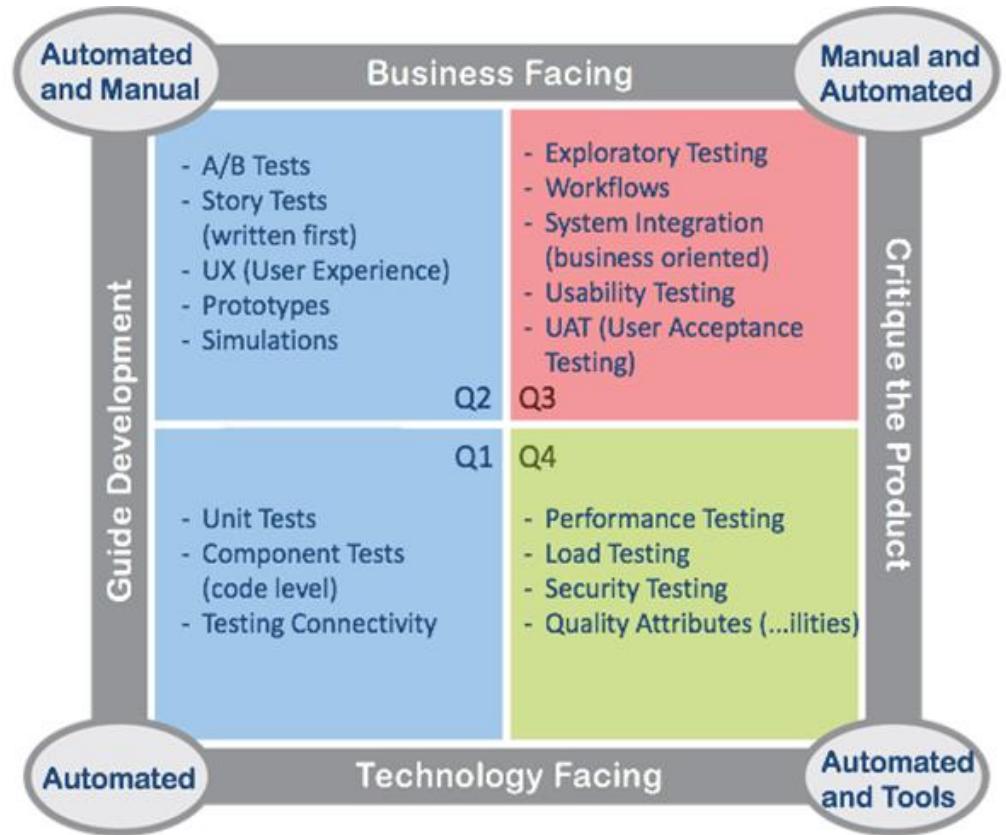
- **To confirm that the system works as agreed,**
- Tests are written to run before and after code changes.
- In software, this is largely the home of **Test-Driven Development (TDD)**.



# Agile siūlomas testavimo procesas (workflow)

Q2

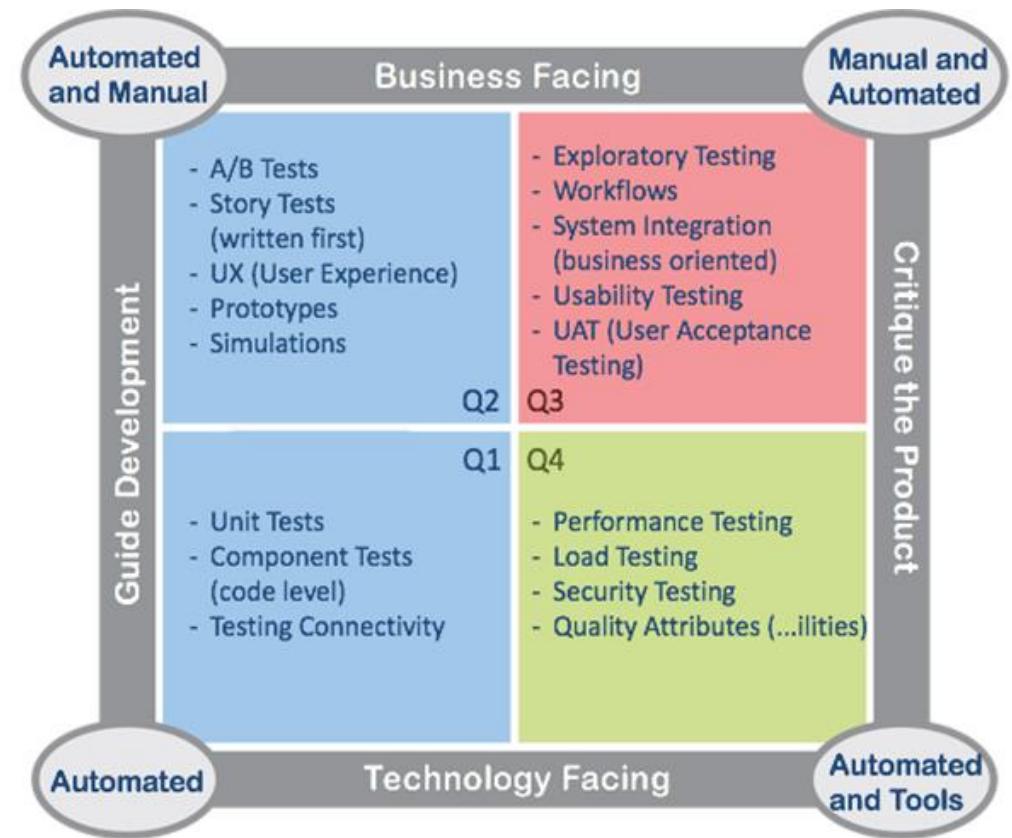
- Contains functional tests:
  - user acceptance tests)for stories, features, and capabilities, **to validate that they work the way the Product Owner (or Customer/user) intended.**
- Feature-level and capability-level acceptance **tests confirm the aggregate behavior of many user stories.**



# Agile siūlomas testavimo procesas (workflow)

Q3

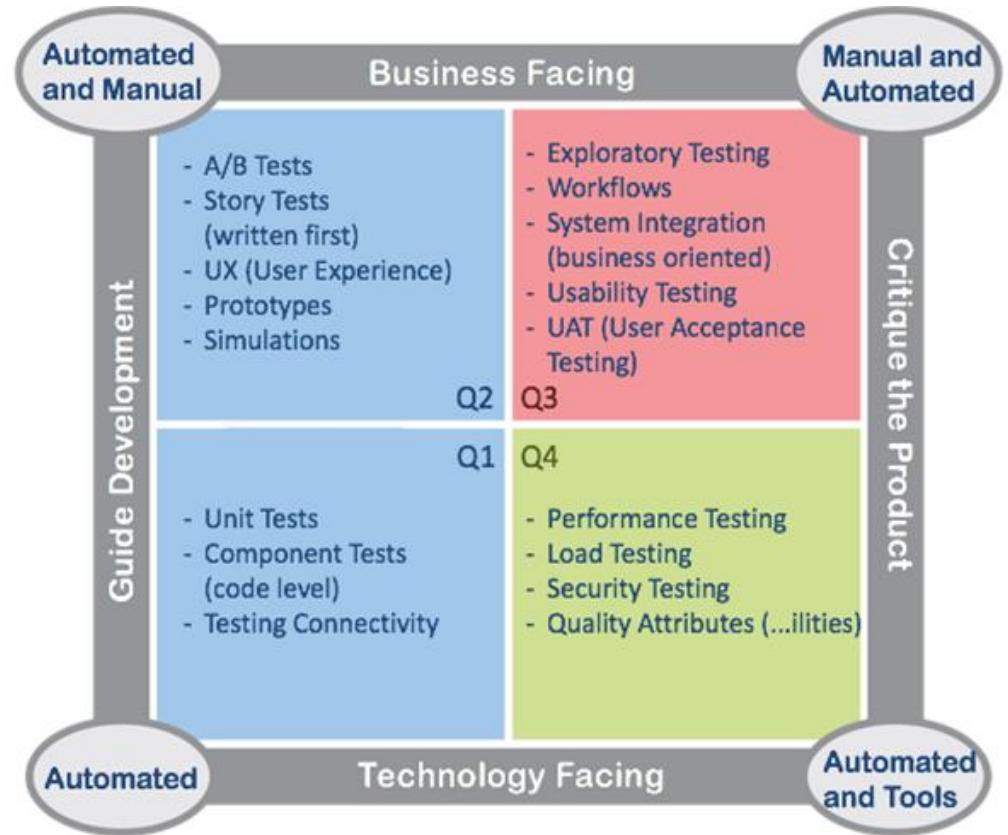
- Contains system-level acceptance tests **to validate that the behavior of the whole system meets usability and functionality requirements**, including scenarios that are often encountered during system use.
  - They involve users and testers engaged in real or simulated deployment scenarios, these tests are often manual.
- **They're frequently the final system validation** before delivery of the system to the end-user.



# Agile siūlomas testavimo procesas (workflow)

Q4

- Contains system qualities testing **to verify the system meets its Nonfunctional Requirements (NFRs)**,
  - Typically, they're supported by a suite of automated testing tools (such as load and performance) designed specifically for this purpose.
- Since any system changes can violate conformance with NFRs, **they must be run continuously**, or at least whenever it's practical.





# Programų kūrimo procesas

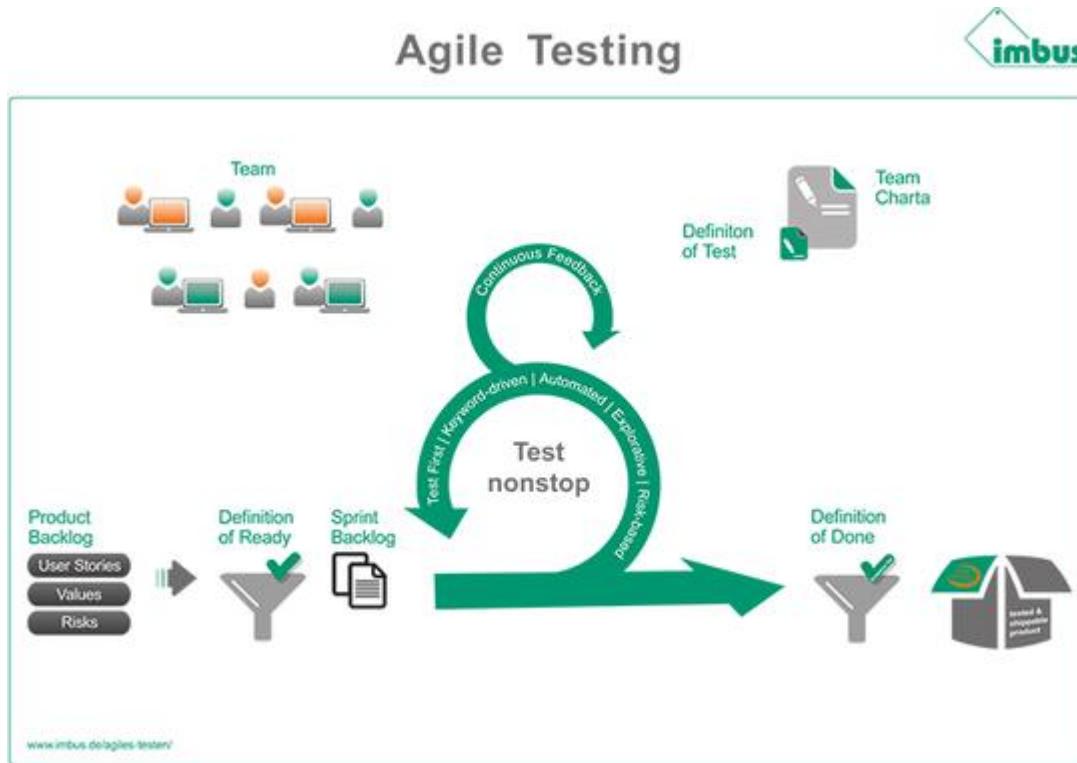
Dr. Asta Slotkienė

# Challenges of Testing in Agile Development

- Requirements change all the time
- Specification documents are never final Code is never ‘finished’, never ‘ready for testing’
- Limited time to test
- Need for regression testing in each increment
  - Developers always break things
  - How can we trust that the code is not broken?

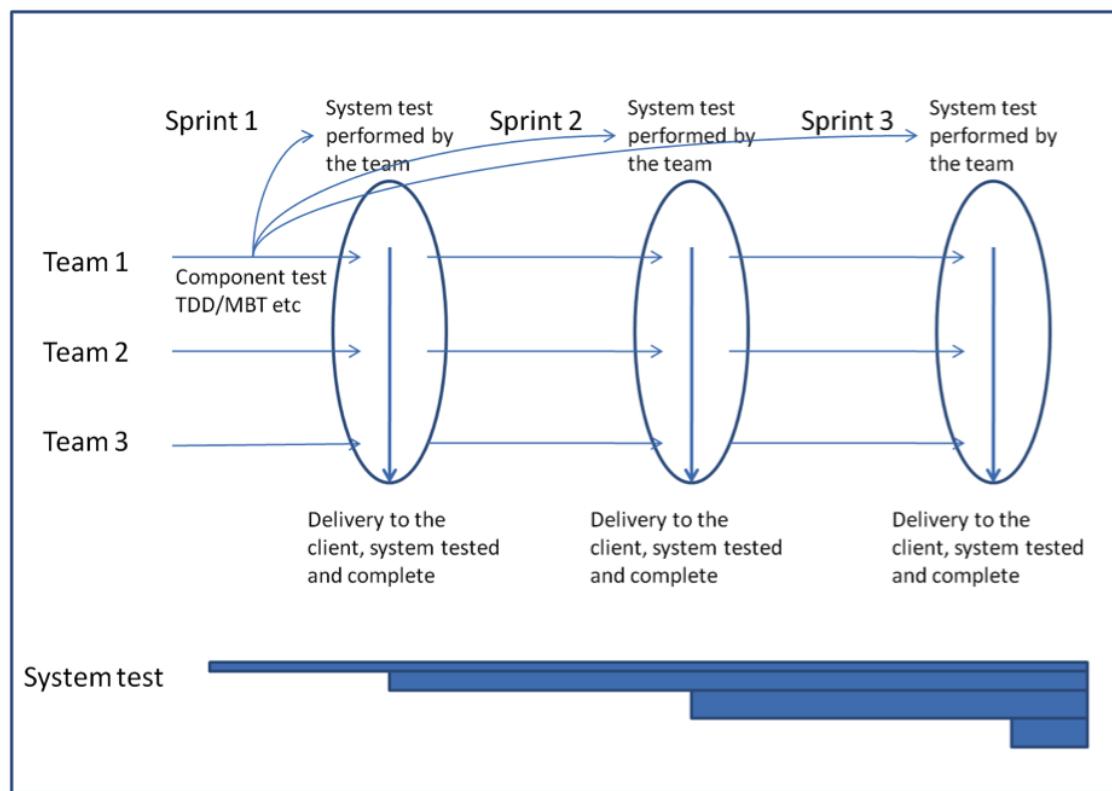
# Agile Testing

- When we modify a system, the modification may result in some unintended and undesirable effects on the system.
- **This effect is called a regression**



# Agile Testing

- Every iteration has its own testing phase.
- It allows implementing regression testing every time new functions or logic are released.



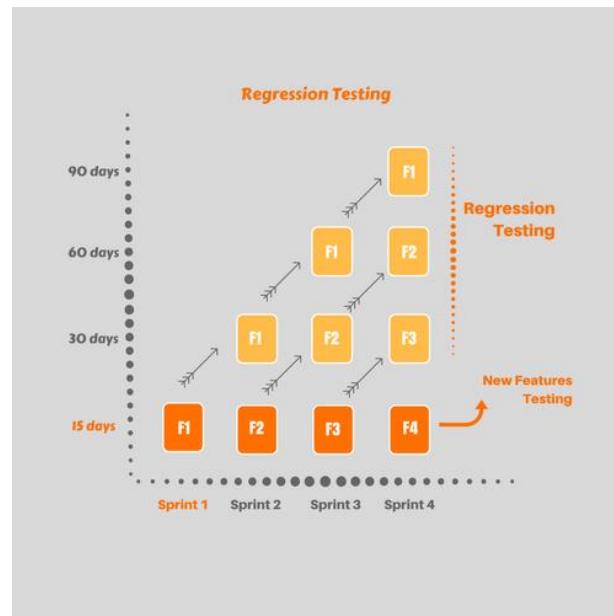
# Types of tools of regression

**Regression tests must be automated**

- Capture / Replay
  - Capture values entered into a GUI and replay those values on new versions –
- Version control
  - Keeps track of collections of tests, expected results, where the tests came from, the criterion used, and their past effectiveness –
- Scripting software
  - Manages the process of obtaining test inputs, executing the software, obtaining the outputs, comparing the results, and generating test reports

# Agile Testing: Regression Testing

- As the releases grow the functionality grows.
- The development time does not necessarily grow with releases, but the testing time does
- No company/its management will be ready to invest more time in testing and less for development



# Regression testing

- *Regression testing* is the re-testing of the software to detect regressions.
- That to detect regressions, **we need to retest all related components**, even if they had been tested before.
- Testing all the prior features and re-testing previously closed bugs

# Definition: Regression Testing

- Regression testing is a type of software testing **that intends to ensure that changes** (enhancements or defect fixes) **to the software have not adversely affected it.**
- **Regression Testing** is defined as a type of software testing to **confirm** that a recent program or **code** change has **not adversely affected existing features.**
- Regression Testing is nothing but a full or partial selection of **already executed test cases** which are **re-executed to ensure existing functionalities work fine.**

<https://www.guru99.com/regression-testing.html>

# Regression Testing

- Regression testing is more effective when it is **done frequently**, after **each small change**.
- However, doing so can be prohibitively **expensive if testing is done manually**.
- Hence, **regression testing is more practical when it is automated**.

# When apply the Regression Testing

Regression testing is required mainly in the these situations:

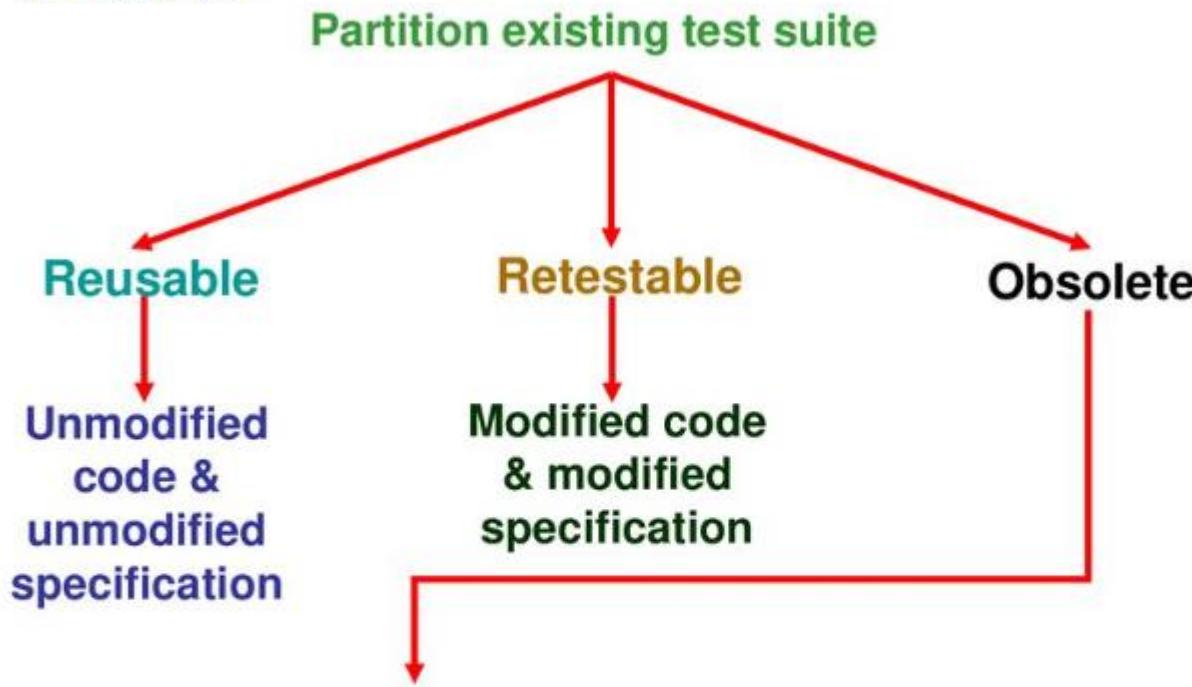
- Change in requirements and code is modified according to the requirement
- New feature is added to the software
- Configuration change
- Defect fixing
- Performance issue fix
- Patch fix

# How much times apply regression testing

- In an ideal case, a full regression test is desirable
  - but often times there are time/resource constraints.
- It is essential to do an **impact analysis of the changes to identify areas of the software that have the highest probability** of being affected by the change and that have the highest impact to users in case of malfunction and focus testing around those areas

# How much times apply regression testing

- ✓ Retest all



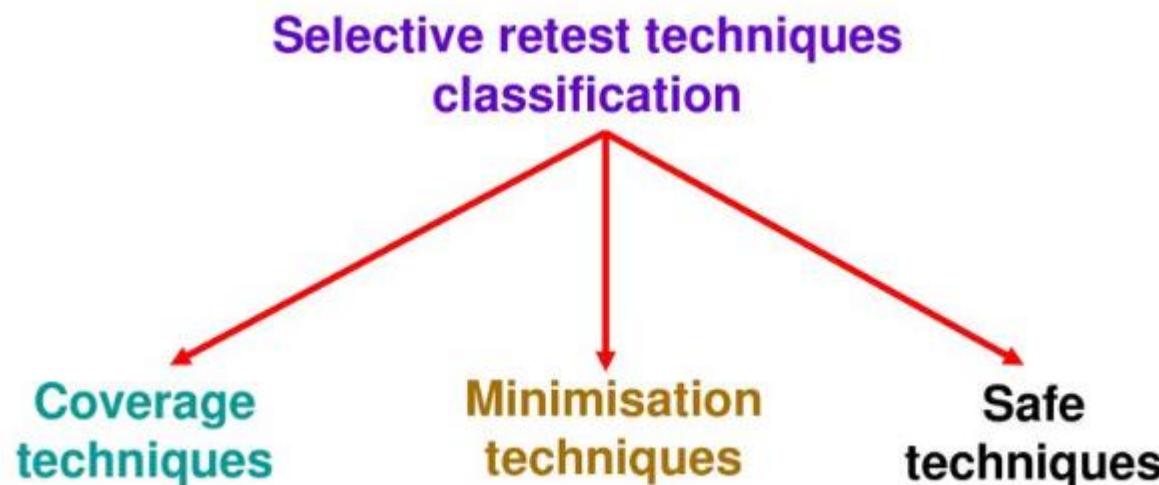
1. Specify incorrect Input Output relationships due to specification modification.
2. No longer exercise the components & specifications they were designed.

Regression Testing

<https://slideplayer.com/slide/16217911/>

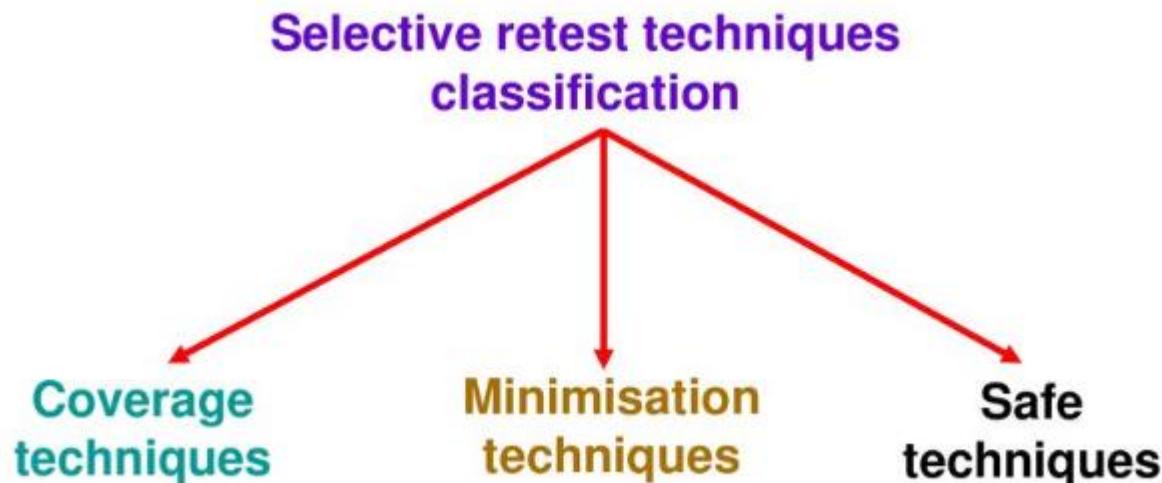
# How much times apply regression testing

- Coverable Techniques
  - coverable program components that **have been modified**
  - **select test cases that exercise these components.**



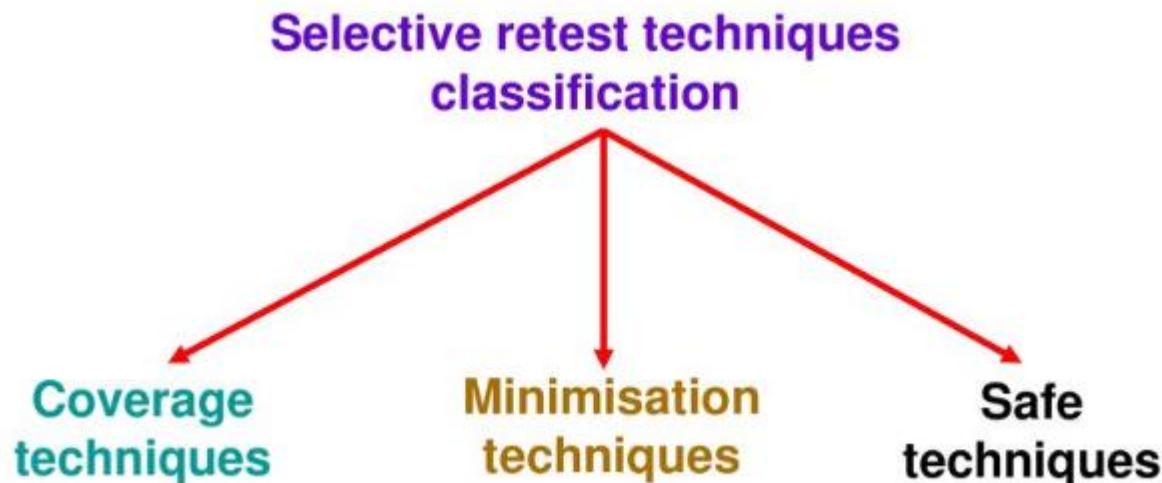
# How much times apply regression testing

- Minimization Techniques
  - except that they **select minimal sets of test cases.**



# How much times apply regression testing

- Safe Techniques
  - select every test case that causes a modified program **to produce different output** than its original version.



# How much times apply regression testing

1. Reuse existing test suite
2. Add new tests as needed
3. Remove obsolete tests
4. Only code impacted by change needs to be retested
5. Select tests that exercise such code

# Steps of Regression Testing

1. Write tests for new features added to the application
2. Execute tests on the new features added
3. Execute old test suite on the whole application
4. Execute old + new test suite on the whole application

Offer: using test automation, when apply Jenkins

# Retesting vs. Regression Testing

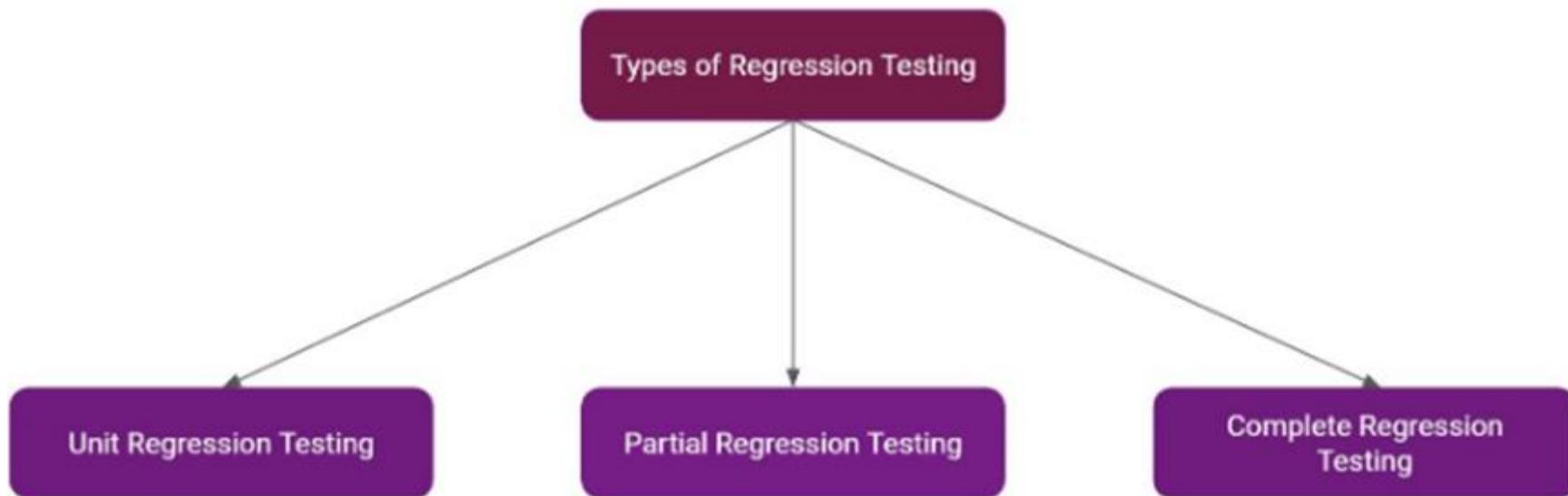
- **Retest means to test again**
  - Retesting is when you **repeat a test for any reason.**
  - retest current version functionality o
  - retest a bug fix,
  - retest previous version functionality,

<https://searchsoftwarequality.techtarget.com/definition/regression-testing>

# Retesting vs. Regression Testing

- **Regression tests** are going to be new versions over existing ones.
  - There is piling on of new features, extensions, etc.
  - Necessary have to test the software's base for **strength and stability**.

# Types of Regression



Tools: Selenium, AdventNet Qengine, Regression Tester, vTest, Watir, actiWate and etc.

<https://www.softwaretestinghelp.com/regression-testing-tools/>

# Types of Regression

## 1. Unit Regression

- Unit regression means you **retest the changed module/area of the application ONLY.**

## 2. Partial Regression

- Partial regression means you **retest the changed module. Plus include those that interact with it.**

## 3. Full Regression

- Full regression is you **test the entire application irrespective of the location of change.**

Tools: Selenium, AdventNet Qengine, Regression Tester, vTest, Watir, actiWate and etc.

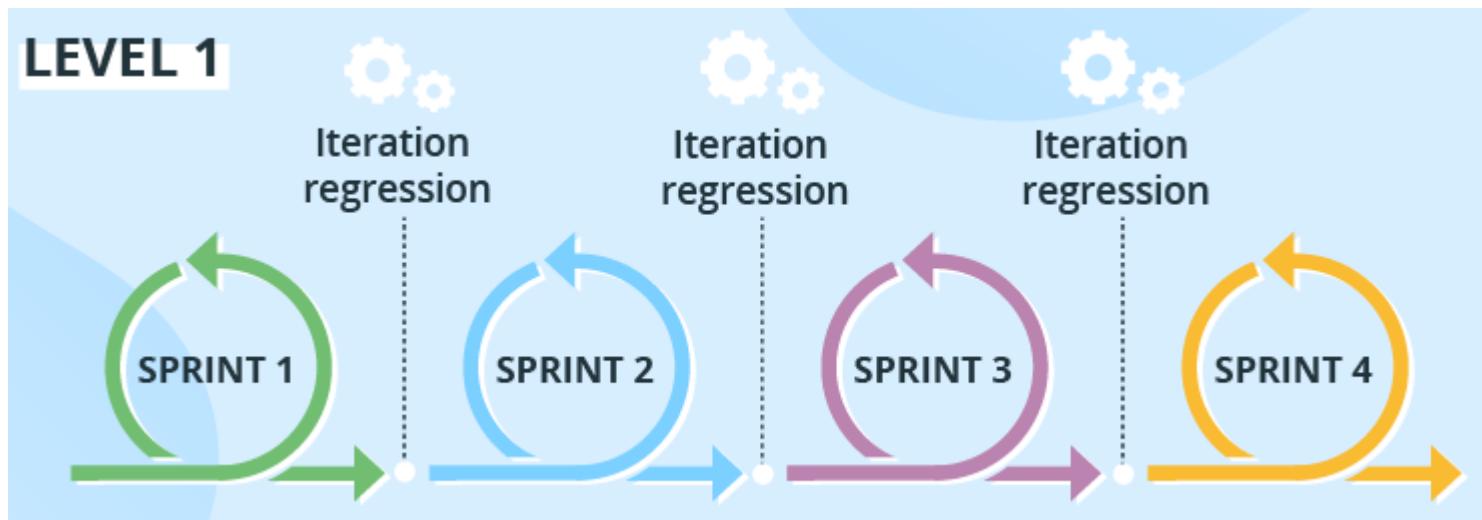
<https://www.softwaretestinghelp.com/regression-testing-tools/>

# When a Regression Test Fails

- Regression test failures represent three possibilities :
  - The software has a fault –>Must fix the fix
  - The test values are no longer valid on the new version –> Must delete or modify the test
  - The expected output is no longer valid –>Must update the test

# Regression In Agile

- Sprint Level Regression
  - Test cases from the test suite are selected as per **the newly added functionality or the enhancement that is done.**



# Regression In Agile

- End to End Regression
  - Includes all the test cases that are to be re-executed to test the complete product end to end by covering all the core functionalities of the Product.
  - It is difficult to perform a GUI Regression test when the GUI structure is modified

