PROJECT PROPOSAL

- The application that we are planning to develop is a website related to delivery of food. The idea is not related to the fast-food delivery as we normally observe, we want to innovate in this field and produce a brand that will be known as one of the healthiest foods with a high-quality level for our customers. In our website you will find one of the most balanced menus for your day-by-day routine. Here, you can find the most detailed nutrition information about the food that you are going to get delivered, showing you every percentage of the proteins, carbohydrates and food fat.

USE CASES

- Order Food

Main scenario

1. The user selects order food.
2. The system shows a list with the available food with a brief description that includes type of food, name and price.
3. The client user selects what is going to order.
4. The system shows a summary of the selected food and total price, showing the user data too (Name, Surname, Phone)
5. The user selects the option pay when it has ordered all the wanted food.
6. The system shows the payment screen.
7. The user fulfills all the requested payment data (address and payment method)
8. The system shows a summary of the order with all the data of the client and an approximation of the deliver time.

Alternative 1 - 2 step

If the system detects that there are not enough ingredients to make one of the meals, it shows it is unavailable so the user is not able to select it.

Alternative 2 - 3 step

2.1. The system gives the user the possibility of filter the different kinds of food by type, price, ingredients...

2.2. The user filters the food wanted.

2.3. The system shows the food filtered by the user.

Alternative 3 - 4 step

If the user hasn't chosen any food, the system would inform and take back to step 2.

Alternative 4 - 8 step

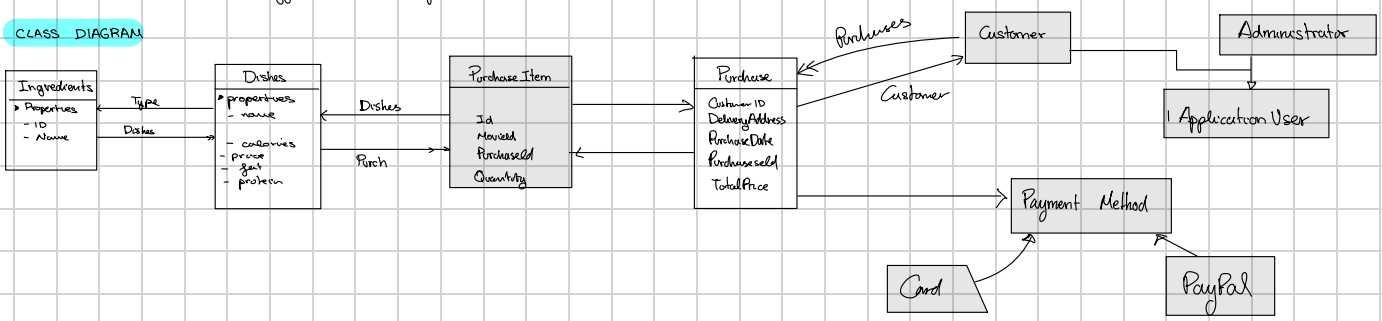
If the system detects that some data of the payment hasn't been fulfilled, it will show the message: "fulfill all the requested data" and it will come back to step 4.

Alternative 4 - 8 step

If there is a problem with the payment, it will show the message "couldn't process the payment, please try again. Thank you". And it would go back to step 6.

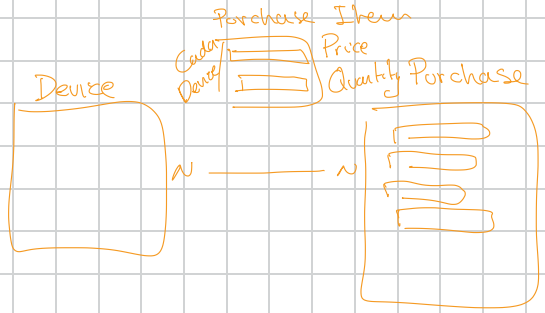
Precondition → Be logged in into the system.

CLASS DIAGRAM



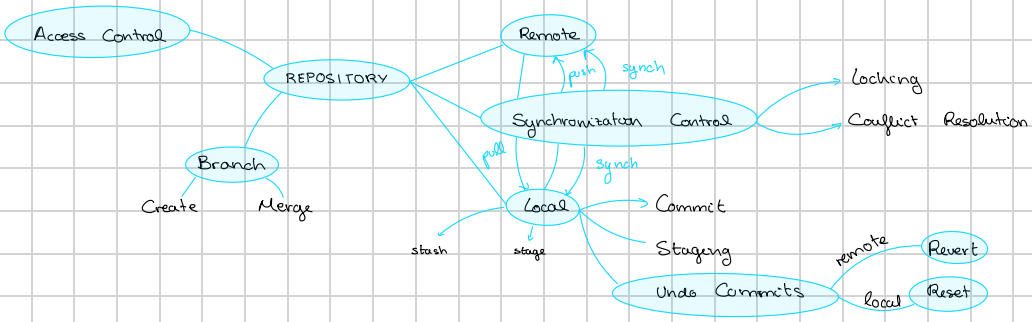
SPRINT 1

1. Requirements (2 points)
   - Flow definition ☐
   - Steps definition ☐
   - Data definition ☐ Revise
   - Precondition definition ☐
2. Class Diagram (3 points)
   - Model folder ☑
   - Properties defined in the Use-Case and the needed ones ☐ Revise.
3. Database Generation (2 points)
   - Generated my Use Case ☐
   - Generated general ☐ Done at the end of all on Development branch.
4. Configuration Management (3points)
   - Own branch ☐
   - Commit with meaningful comments and work items management ☑
   - Scrum master only accepting pull requests ☐

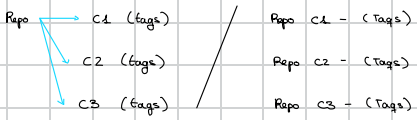


1	2	3	4
Model	Device	Purchase Item	Purchase
Scale	Repair	ReceiptItem	Receipt

compare
1 in 2
but not 2 in 1



How would you create the repo with repo, branches, tag, tag ... To support this (design 2d)? Exam question



CREATING DATABASE

→ FOLDER "MODELS" → Create all classes of the database → Primary key (Id unique int) [Key] to make something a PK.

- → Alternate key → ApplicationDbContext.cs → "builder.Entity<NameClass>().HasAlternateKey(g => g.Name);
- → Composed PK N-N Relation → ApplicationDbContext.cs → "builderEntity<NameClass>().HasKey(pi => new { pi.Pk1, pi.Pk2});
- → Attributes → Annotations
  - → For strong [StringLength(max, ErrorMessage = "", MinimumLength =)] // [RegularExpression(@"^[A-Z]+[a-zA-Z"-'\s...
  - → Special formats [EmailAddress], [Url], [CreditCard], [Phone].
  - → For Numbers. [Range(min, max, ErrorMessage = "")] // [Precision(precision, range)]

WHEN CREATED ↓

→ FOR EACH MODEL → ApplicationDbContext.cs → "public DbSet<YourClass> YourClass {get; set;}

→ API → Solution Explorer → "Set as Startup Project"

→ View > Other Windows > Package Manager Console. → Package origin → nuget.org

↳ Predetermined → API Project

→ EXECUTE: Add-Migration InitializingDatabase

Update-Database.

Update DataBase before pull request.

Inheritance → New attribute: Discriminator. → Determines which attributes we fill

↓

Alternative 1 better.

ApplicationUser (Slide 35) take a look.

Unit 1. Software Configuration Management

1. Introduction
2. Basic Concepts
3. Process of Configuration Management
4. CASE for SCM
5. Conclusions

Goals

- Understanding the importance of Software Configuration Management (**SCM**)
- Understanding which the key activities of Software Configuration Management are
- Understanding why Version Control and Change Control must be integrated
- Understanding the difference between Change Control and Version Control
- Being able to discuss which features a CASE should provide for SCM

1.1 Introduction

- Definition of Configuration (ISO/IEC/ IEEE 24765-2010)
  1. The arrangement of a computer system or component as defined by the number, nature, and interconnections of its constituent parts
  2. The arrangement of a system or network as defined by the nature, number, and chief characteristics of its functional units.
  3. The requirements, design, and implementation that define a particular version of a system or system component.
  4. The manner in which the hardware and software of an information processing system are organized and interconnected.

In Configuration Management, the **functional and physical characteristics** of **hardware** or **software** as set forth in technical documentation or achieved in a product

1.1 Introduction

Configuration Management

- Why CM is necessary: → Manage all versions of our software.
  - The basis of: ↳ Important for teleworh.
    - **Software products**: multiple version of different components that are running on different hardware and software platforms
    - **Software projects**: everything is prone to be changed
    - **Software development teams:** distributed development teams

control changes and modifications asked by the clients.

  - High complexity of software systems
  - High demand of software
    - Musa's Law: 900% rise in the demand per decade
    - Boehm's law 200% rise in the costs per decade
    - …. But we have just a 35% rise in productivity
  - The own nature of software: **laws of software evolution** formulated by **Lehman** and **Belady** → All software is changing through its lifecycle and CM allows us to make this.
    - An *E*-program is written to perform some real-world activity; how it should behave is strongly linked to the environment in which it runs, and such a program needs to adapt to varying requirements and circumstances in that environment
    - **Law of Continuing Change**: *an E-type system must be continually adapted or it becomes progressively less satisfactory*

1.1 Introduction

- How **Configuration Management** helps us:
  - To manage a system well, you have to know how it's built
  - In order to know what you've got after a change, you have to know what you had before the change
  - To find & fix a problem, you usually have to know in some detail what your "It works!" configuration was
  - It is finally an activity of **Quality Assurance** applied from **inception** to **maintenance**

1.2 GCS: Basic Concepts

- **Configuration Item (CI)** [IEEE24765]: an aggregation of hardware, software, or both, that is **designated for** Configuration Management (CM) and **treated** as a **single entity** in the Configuration Management process
  - Some of the CI to be tackled are:

Important

- System Specification
- Software Requirements Specification
- Design Specification
- Source code
- Testing Specification
- Operating System

all brings changes to the others we need to control them always.

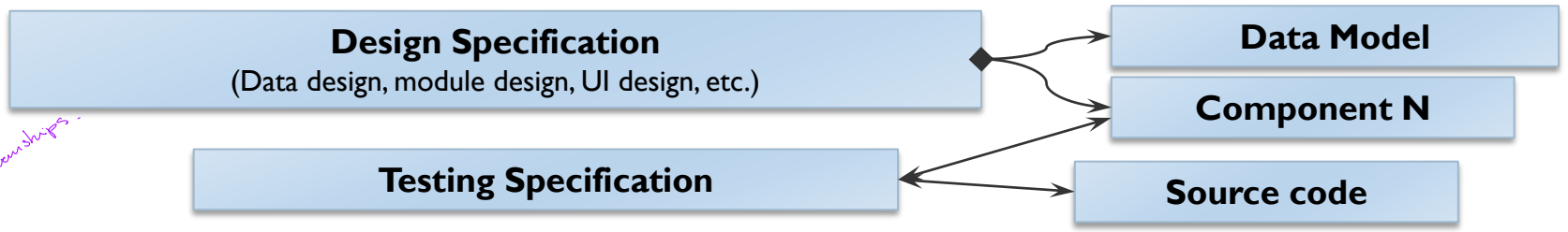
- Project Plan
- Documentation of Operation and Installation
- Runnable software
- DB specification
- User Documentation
- Documentation of maintenance
- Standards and procedures of Software Engineering

  - Tools and IDEs are also managed by the CM, **why?**

Because if the client changes it OS or something we are expected to work properly. We have to maintain every version of the OS, Tools and IDEs. SAVE LIBRARIES.

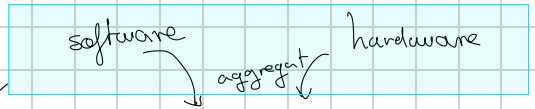
  - A CI has always **name, attributes** and **relations** with other CIs

Configuration Items.



Support relationships.

CONFIGURATION ITEMS



→ CI

- System Specification
- Software Requirements Specifications
- Design Specification
- Source Code
- Testing Specification
- Operating Systems.

CM

Manages → Tools and IDEs

Name, attributes, relations...

Design Specification → Data Model, Component N

Testing Specification → Source code

Producing software from a specification is like walking on water - it's easier if it's frozen.
Barry Boehm

1.2 GCS: Basic Concepts

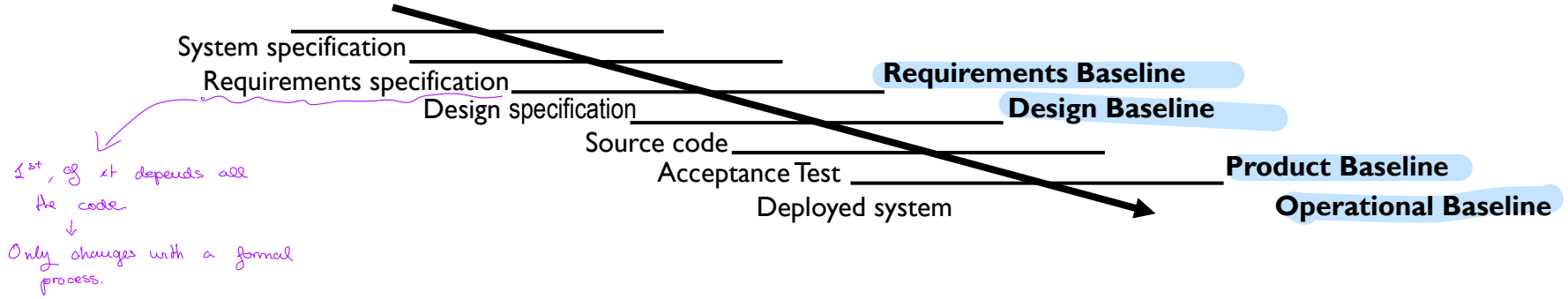
- **Configuration** [IEEE24765]:
  - the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product
- **Configuration Baseline** [IEEE24765]:
  - specification or product that has been formally **reviewed** and **agreed** upon, that thereafter serves as the basis for further development, and that can be **changed only** through **formal change control procedures**

  - **Objective** of a baseline:
    - to reduce a project's vulnerability against uncontrolled changes by formally **fixing** and **controlling** those **Configuration Items that are key** at critical points in the development life cycle.
    - to identify **the aggregate** of software and hardware components that make up **a specific release** of a system.

Producing software from a specification is like walking on water - it's easier if it's frozen.
Barry Boehm

1.2 GCS: Basic Concepts

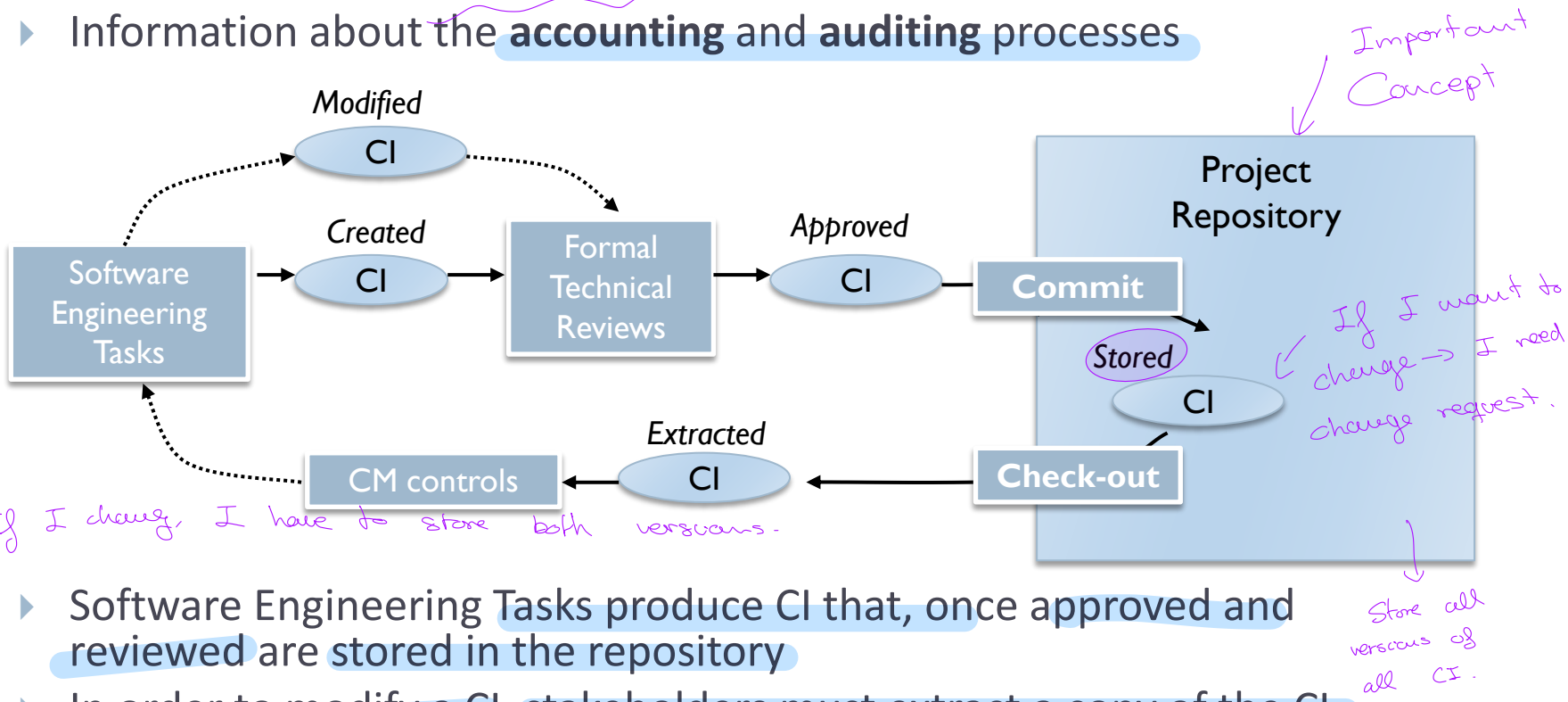
- **Configuration Baseline**:
  - When:
    - Usually established at the end of a stage of the Lifecycle, **why?**



  - "**to baseline**" refers to the act of placing an approved item under formal change control
  - **CI Baseline**: formally approved version of a CI, regardless of media, **formally designated and fixed** at a specific time during the Configuration Item's life cycle

1.2 GCS: Basic Concepts

- **Project Repository**: record all the relevant information related to the configuration :
  - Information of the CIs and their dependence relationships
  - Information of the **Change Requests** and its state
  - Information about the **accounting** and **auditing** processes



  - Software Engineering Tasks produce CI that, once approved and reviewed are stored in the repository
  - In order to modify a CI, stakeholders must extract a copy of the CI (following the dotted line).

1.2 GCS: Basic Concepts

- **Version**:
  - [IEEE24765]: an **operational software product** that differs from similar products in terms of capability, environmental requirements, and configuration
  - [IEEE24765]: an identifiable instance of **a specific file** or **release of a complete system**
  - Every change applied to a CI produce a new version of both that CI and its related product
  - Identification:
    - Number
    - Set of logic variables: language =C#, platform = W10, date = October 2015
    - Oriented to change: Set of changes sequentially applied

- **Release**:
  - [IEEE24765]: a delivered version of an application which may include all or part of an application
  - [IEEE24765]: a software version that is made formally available to a wider community
  - Number of releases < number of versions**??**

1.2 GCS: Basic Concepts

- Configuration Management ISO/IEC/IEEE 24765-2010:
  - a discipline applying technical and administrative direction and surveillance to:
    - **identify and document** the **functional and physical characteristics** of a **configuration item**,
    - **control changes** to those characteristics,
    - **record and report change processing** and **implementation status**, and
    - **verify compliance with specified requirements**
  - technical and organizational activities comprising configuration **identification**, **version and change control**, **status accounting**, and **auditing**

1.3 CM Process

- CM looks for answering the following questions:
  - How does an organization identify and manage the existing versions of a product?
  - How are changes controlled before and after a product is deployed?
  - Who is responsible for approving and assigning priorities to the changes?
  - How can the changes be checked?
  - How can be other stakeholders notified of our changes?

- Activities of the CM:
  - Identification
  - Version Control
  - Change Control
  - Configuration Status Accounting
  - Configuration Auditing

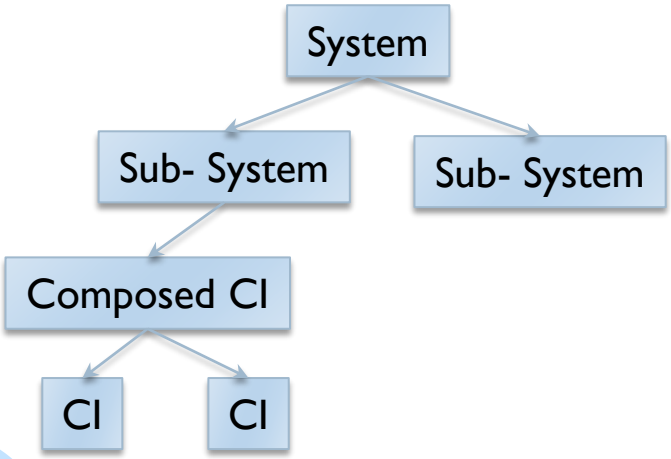
1.3 CM Process: **Identification**

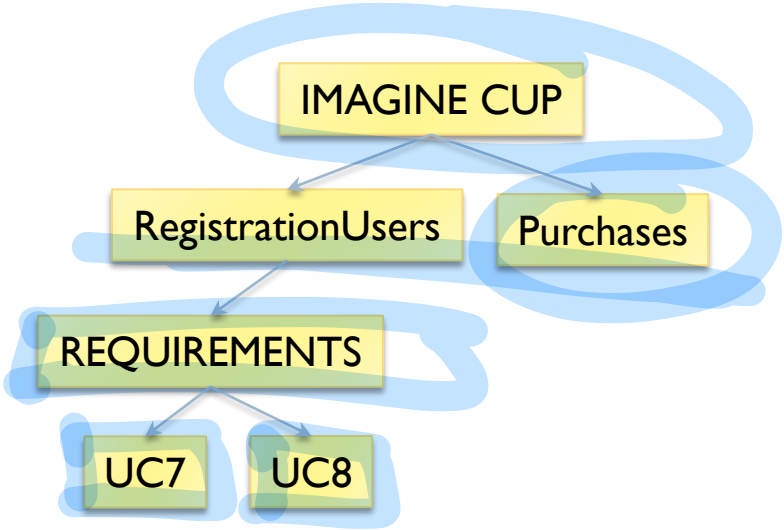
- Pre-requisite for the other activities of CM
- It implies three different tasks:
  - **Selection**: to determine which CI will be controlled
  - **Labelling Scheme**: to establish a labelling scheme and or numbering of the CIs to be managed by the CM
  - **Description:** to document the characteristics of each CI

- **Selection**:
  - Not every type of CI must be under CM:
    - Too many: managerial cost, higher time and cost of development
    - Too few: difficulty in controlling changes as well as reduced visibility
  - Determine Selection Criteria:
    - critical for product, reuse, relations with other CI, complexity of CI, used by different teams, etc.

1.3 CM Process: Identification

- **Selection**:
  - Select the CI and the relations among them
    - Atomic CI: ex. File of code
    - Composed CI: collection of basic or composed CIs
  - Establish hierarchical relations and dependencies among CIs:



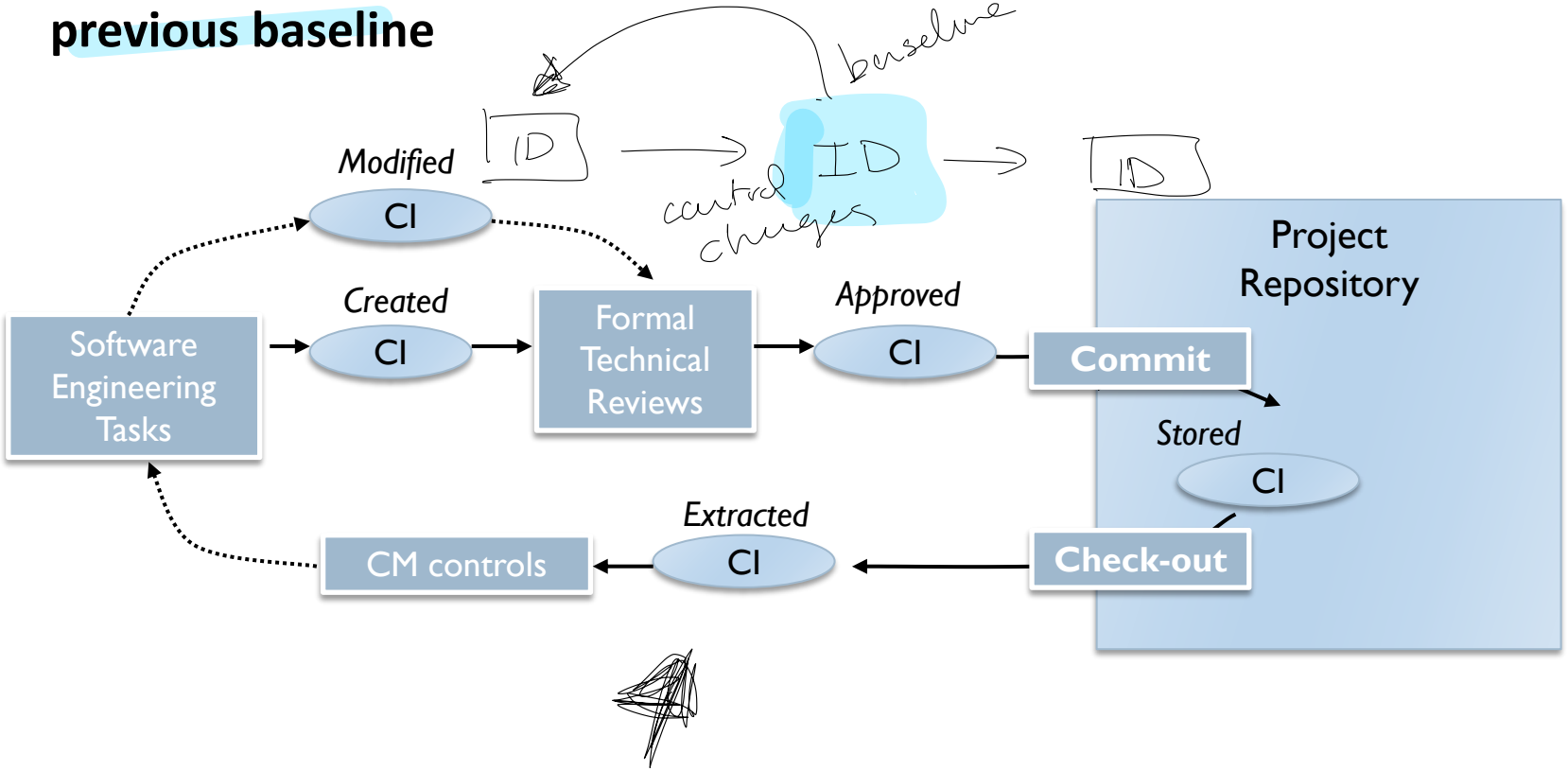


1.3 CM Process: Identification

- **Labelling Scheme:**
  - To establish a **scheme of labelling or numbering** to identify uniquely each CI
    - To establish the storage, recovering, monitoring and distribution
  - **Method of identification:** it can include conventions of naming, number and letter of version, as well as the name of the Project or system, its position in the hierarchy and the type of CI
    - Example: ICUP_RegistrationU_REQ_UC7_1.0
  - Other attributes
    - Name (string)
    - Description (type, id. of project, version and/or change)
    - Resources (entities required by the object)
    - Realization (reference of the object)

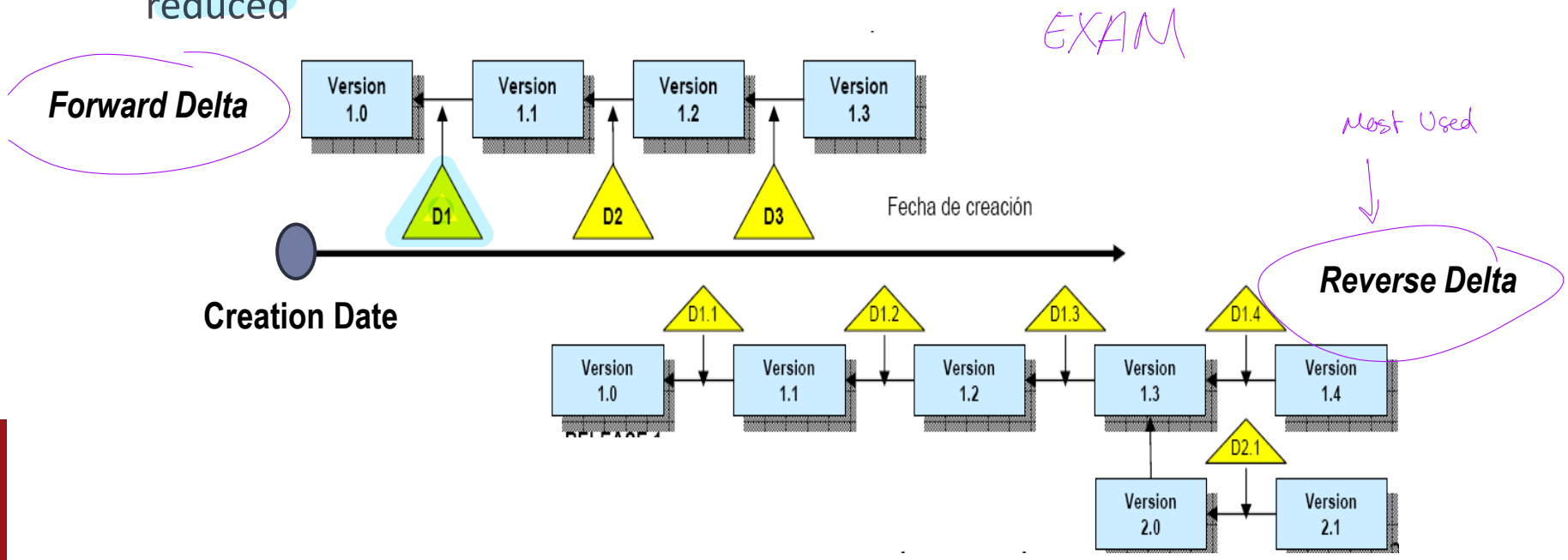
1.3 CM Process: **Version Control**

- Procedures and tools to manage the different versions of the CIs that are created during the software development process
- Establishment and maintenance of **baselines** and the **identification** and **control** of changes to **baselines** that make it possible to **return** to the **previous baseline**



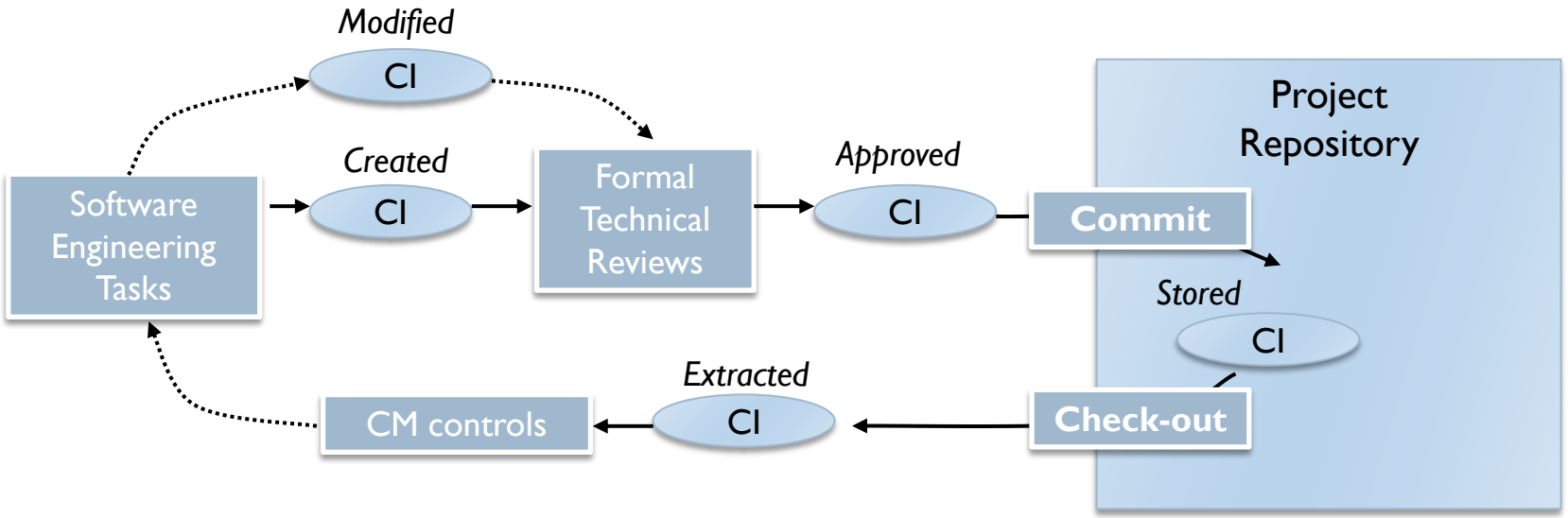
1.3 CM Process: Version Control

- Features to be supported by the Repository:
  - **Deltas**: when a new version is created, the difference between the new and the previous version is called delta
  - Instead of saving copies of all versions in the repository, we create deltas: the **amount of disk space** required for version management is greatly reduced



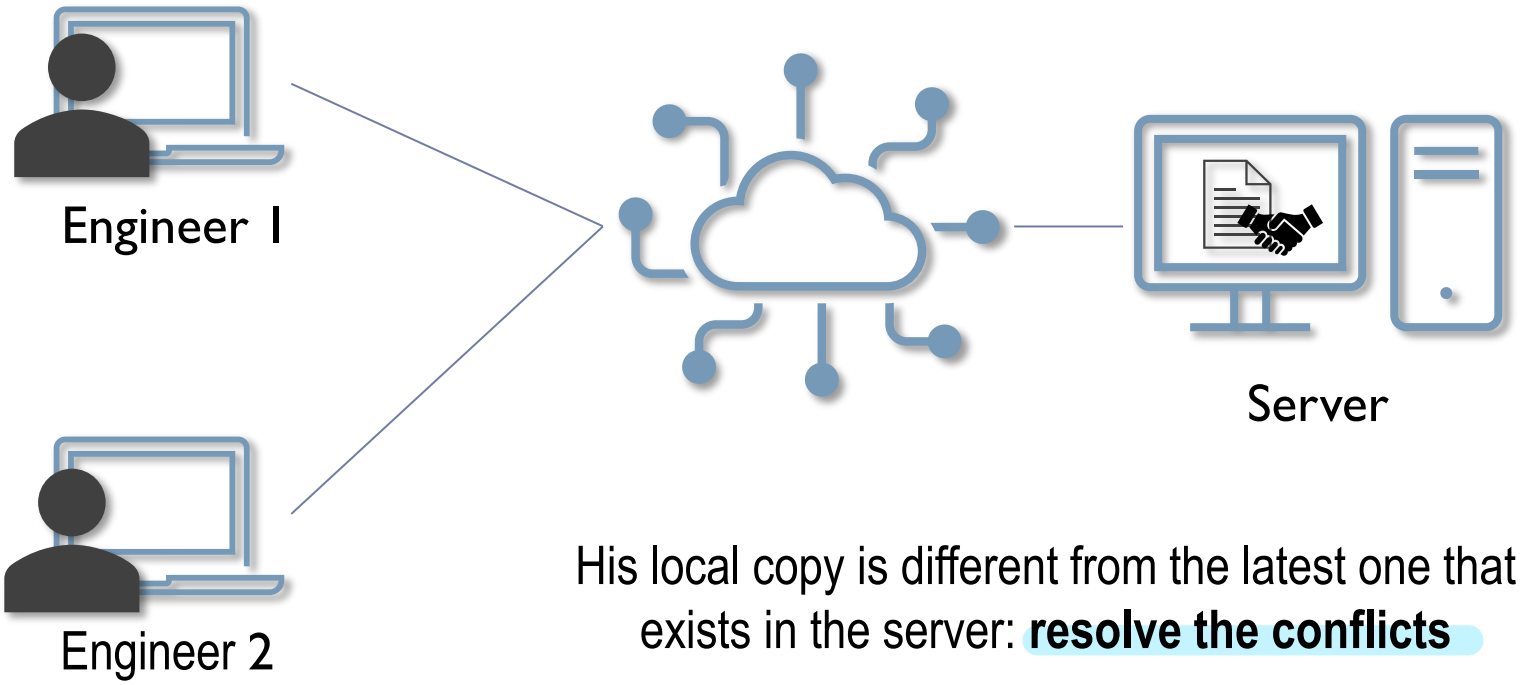
1.3 CM Process: Version Control

- Features to be implemented in the repository:
  - **Access control** → Only specified users will be able to write, read, specific CI from our Repository.
  - **Synchronization control** Repository should provide us facilities to maintain the same version of the CI in every replication of the Repository.



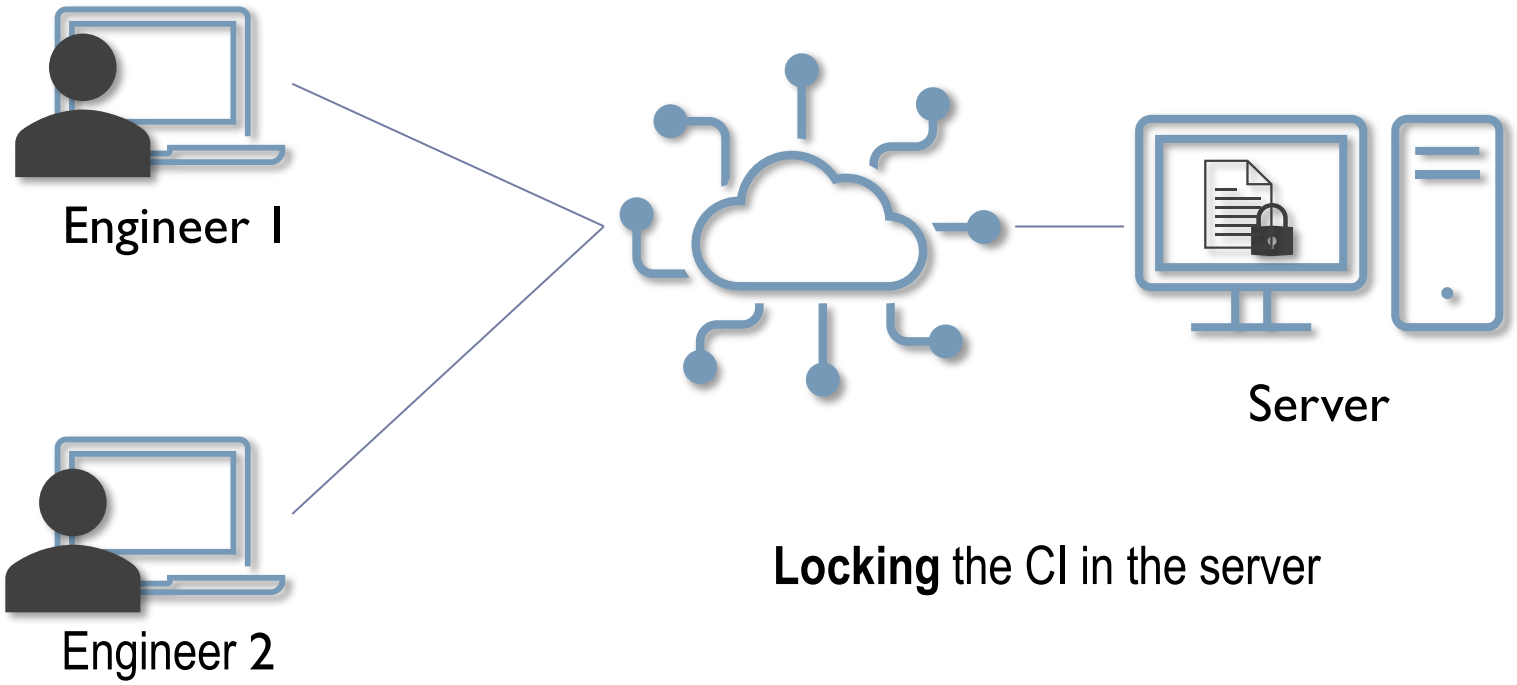
1.3 CM Process: Version Control

- What does it happen when there is a simultaneous update? Engineer 1 and Engineer 2 want to modify Class1.cs
  - Synchronization Control: **Option A) Resolving conflicts**



1.3 CM Process: Version Control

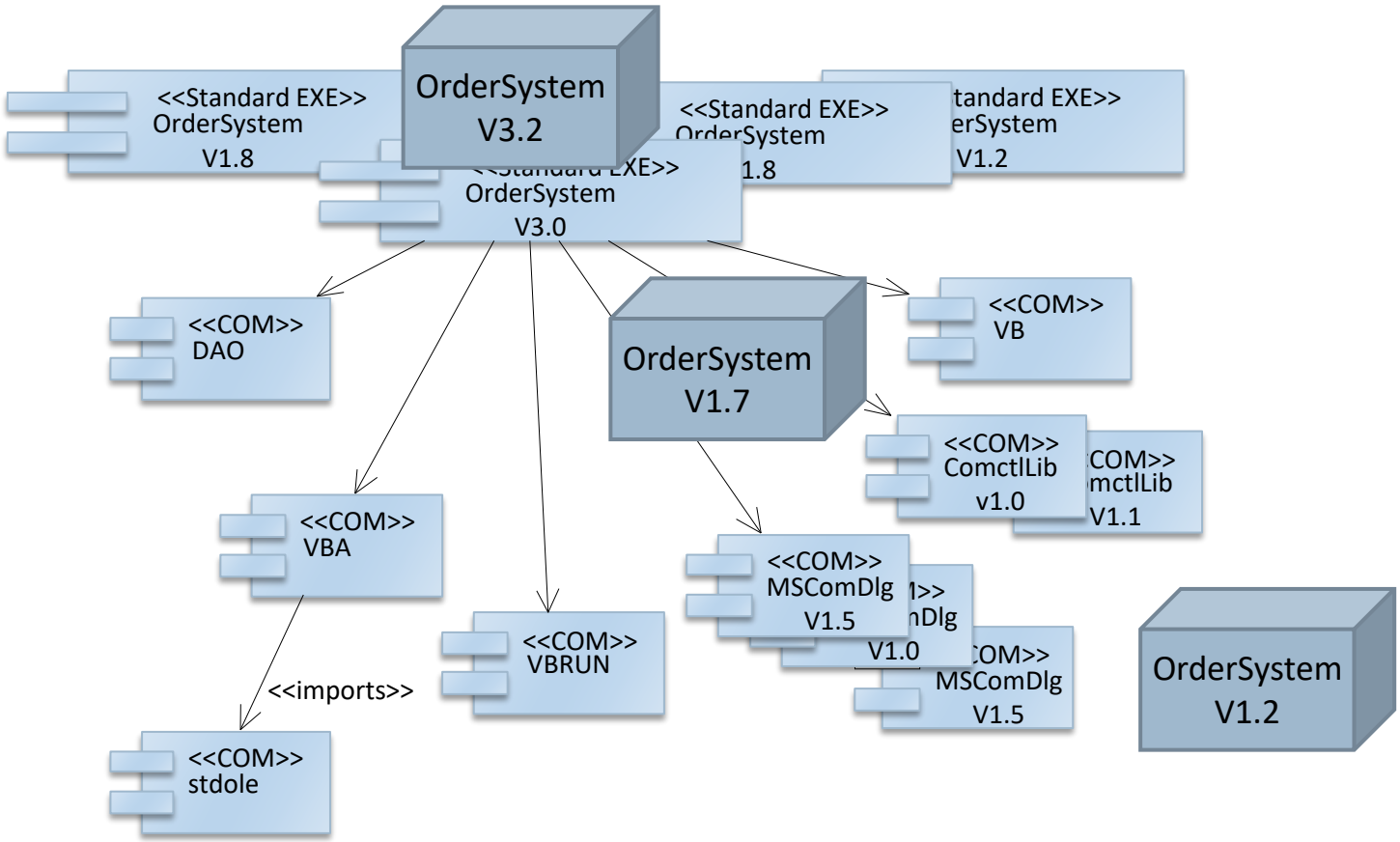
- What does it happen when there is a simultaneous update? Engineer 1 and Engineer 2 want to modify Class1.cs
  - Synchronization Control: **Option B) Locking/remove lock**



1.3 CM Process: Version Control

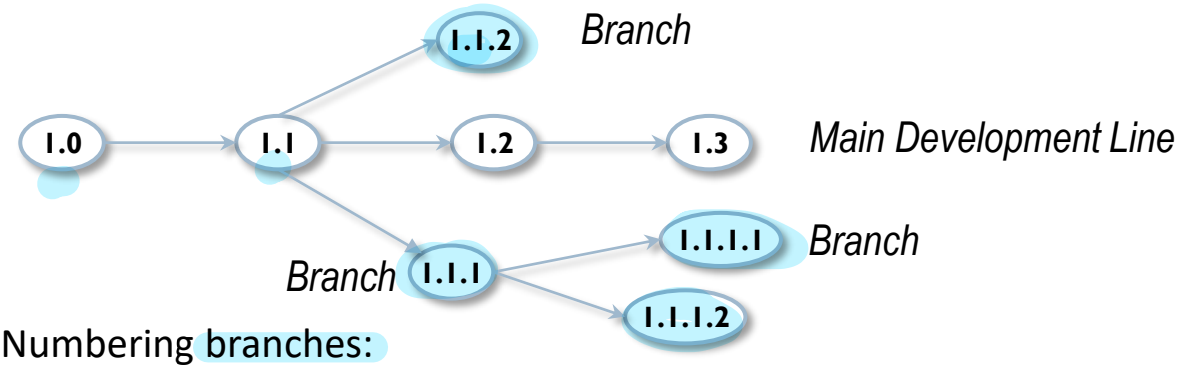
- Features to be supported by the Repository:
  - CIs Version:
  - **Product** Versions:

Problem: Different versions of the product are built using different combinations of versions of CIs



1.3 CM Process: Version Control

- Features to be supported by the Repository:
  - CIs Version:
  - **Product** Versions:
    - **Evolution Graph:**
      - Each **node** is a Product Version (collection of CI)
      - Each **Branch** is a deviation from the main development line for an item



      - Numbering branches:
        - Version numbers on the main development line have only two parts: major and minor number (i.e., 1.1, 1.2, etc)
        - Version numbers of branches have four parts: the first two parts represent the point at which the branch splits off the main line, the third which of the many possible branches it is (i.e., 1.1.1, 1.1.1.2, etc.)
      - **Branching and Merging**: mechanisms to be supported by VC system

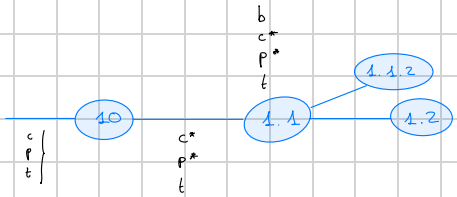
Facilitate product construction
Facilitate Parallel Development
Facilitate building alternative versions

Commit - c

Push - p

Branch - b

Tag - t



CHANGES CONTROL

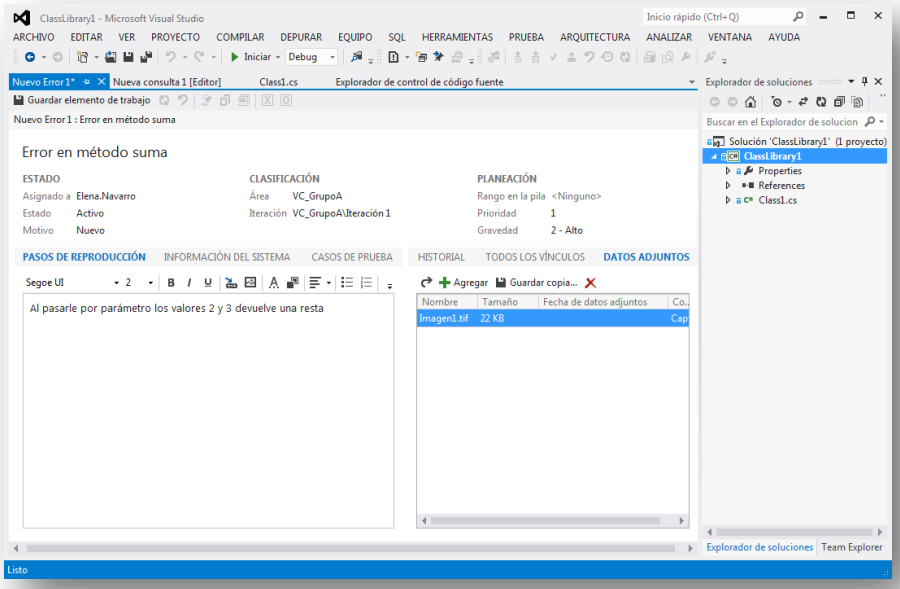
- The Change Request (Error in TFS) is emitted by an involved in the process of development faced to a need of modification.
  - It passes to close state when the person who emitted it verifies the change done.
  - PARTS
    - Name: Brief description of the Change Request.
    - Steps of reproduction: It allows to determinate how execute the application to see the error described.
    - State: Determinate on what point of the process of control Changes we find the Request.
  - PROTECT the modifications → Associate the new Changes with the Request.
  - TRAZABILITY → when we protect the modification and associate it to the Request.
    - Allows us to see the code versions before and after the modification needed to solve the Change Request.
  - The Change Request is associated to as many versions as times that it has been protected the solution and it has been established the relationship.
  - Change history of Change Request Nuevo → Activo → Motivar Todo → Corregido → Resuelto
  - No todas se resuelven modificando código.

1.3 CM Process: Version Control

- **System Building**:
  - **Build** (**Construcción**): operational version of a system that has sub-set of features that the final product will have
  - **System Building** :
    - **Combine the right versions** of the CIs, using the suitable configuration data, into components which execute on a particular target configuration (compiling and linking)
    - Activity carried out over and over throughout the lifecycle of a system to provide customers, developers, testers, etc, with that build they need
    - Required features:
      - **Replicable**
      - **Reproducible**
    - Automation:
      - Automated tools using scripts: components and their versions, their location, environmental parameters, etc. (for instance, a makefile)
      - Supporting tools and scripts should be saved in the Repository

1.3 CM Process: Change Control

- Identifying, documenting, approving or rejecting, and controlling changes to the project baselines
  - It has to be carried out whenever someone requests a change
- Concepts:
  - **Change Request** (Solicitud de Cambio, CR): Request submitted by a developer, member of the Quality Team, a reviewer, a user, a client that must be reported



Reasons:

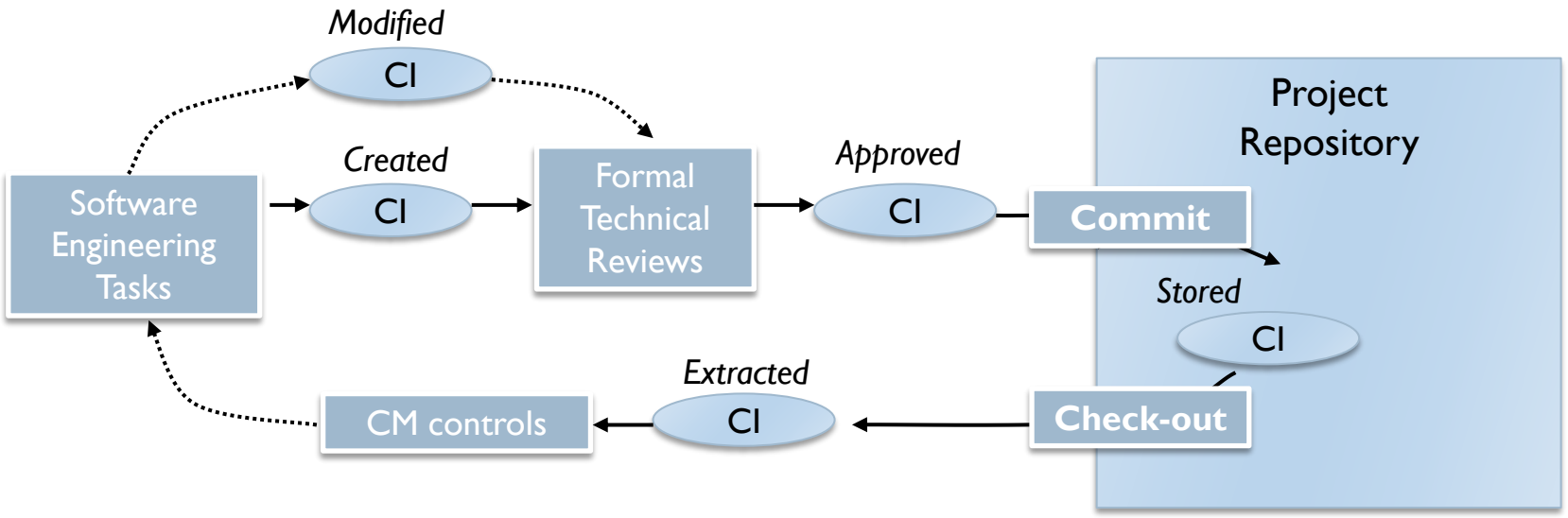
- Improve a design
- Bug found
- Functionality to be changed
- Etc.

Other names:

- Ticket
- Issue
- Work Item (Elemento de Trabajo)

1.3 CM Process: Change Control

- Change Control and Version Control **must be integrated**
  - When? When a change is being implemented
    - "Commit" "Check-out"
  - Why?

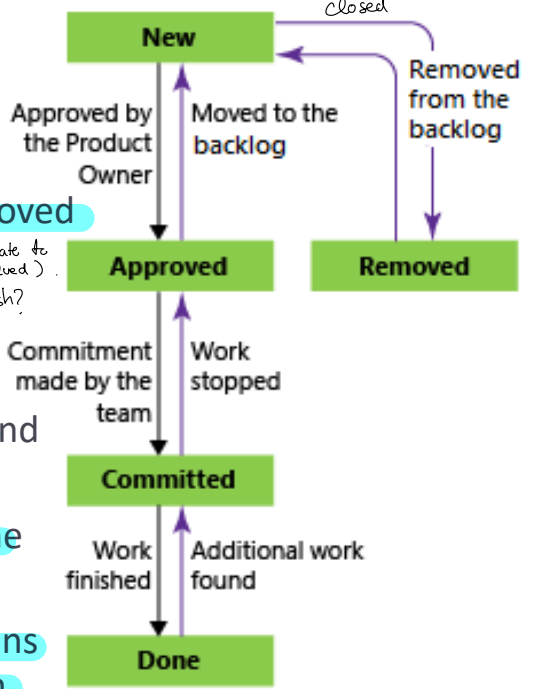


1.3 CM Process: Change Control

- Concepts:
  - **Change Control Process:** actions taken to identify, document, review, and authorize changes to a software or documentation product that is being developed
  - **Change Control Board** (Autoridad de Change Control, CCB): a formally constituted group of stakeholders responsible for reviewing, evaluating, approving, delaying, or rejecting changes to a project, with all decisions and recommendations being recorded.
  - **Levels of Change Control**: avoid excessive bureaucracy
    - **Informal Change Control**: the stakeholder can do whatever change till the CI is baselined.
    - **Project Change Control**: we have a baseline. To carry out a change it must be approved either by the Project Manager (local impact) or by CCB (global impact).
    - **Formal Change Control**: We have a release. All the process of Change Control must be carried out.

1.3 CM Process: Change Control

- **Change Control Process (example):**
  1. Change Initiation: Stakeholder submits a CR
  2. Change Classification: determine the category of the CR
     - Configuration Management Officer (CMO) classifies the CR according to its severity, importance, impact, cost, etc.
  3. Change Evaluation/Analysis (scope): CCB analyses the change for impact on product safety, releability, etc, identify the changes that will have to be made to implement the CR, Cis to be modified, etc.
  4. Change Disposition: CCB, generally, determines if a CR is either approved or denied:
     - Denied: a report is forwarded to the originator describing why
     - Approved: the CR is assigned to the developer team
  5. Change implementation: Developer team implements the change and test the product as needed
  6. Change Verification: The implemented change must be verified at the system level and reported to keep the change history
  7. Baseline Change Control: In order to minimize the number of versions and the frequency of delivery of products, the BCC will make a decision either:
     - Creating a new release to distribute the change
     - Or waiting for additional changes



1.3 CM Process: Status Accounting

- Consisting of the recording and reporting of information needed to effectively manage a software system and its characteristics
- Benefits:
  - Identifying problems, pinpoint the source of the problem and take corrective actions
  - Evaluate the progress of the Project
  - Determine why during maintenance

Info to use the software properly

- Record and communicate what information is needed to effectively manage CIs throughout the product life cycle:
  - Record of approved configuration documentation and identification numbers
  - Status of proposed changes
  - Implementation status of approved changes
  - Status of open Change Requests
  - Build state of all units of Cis

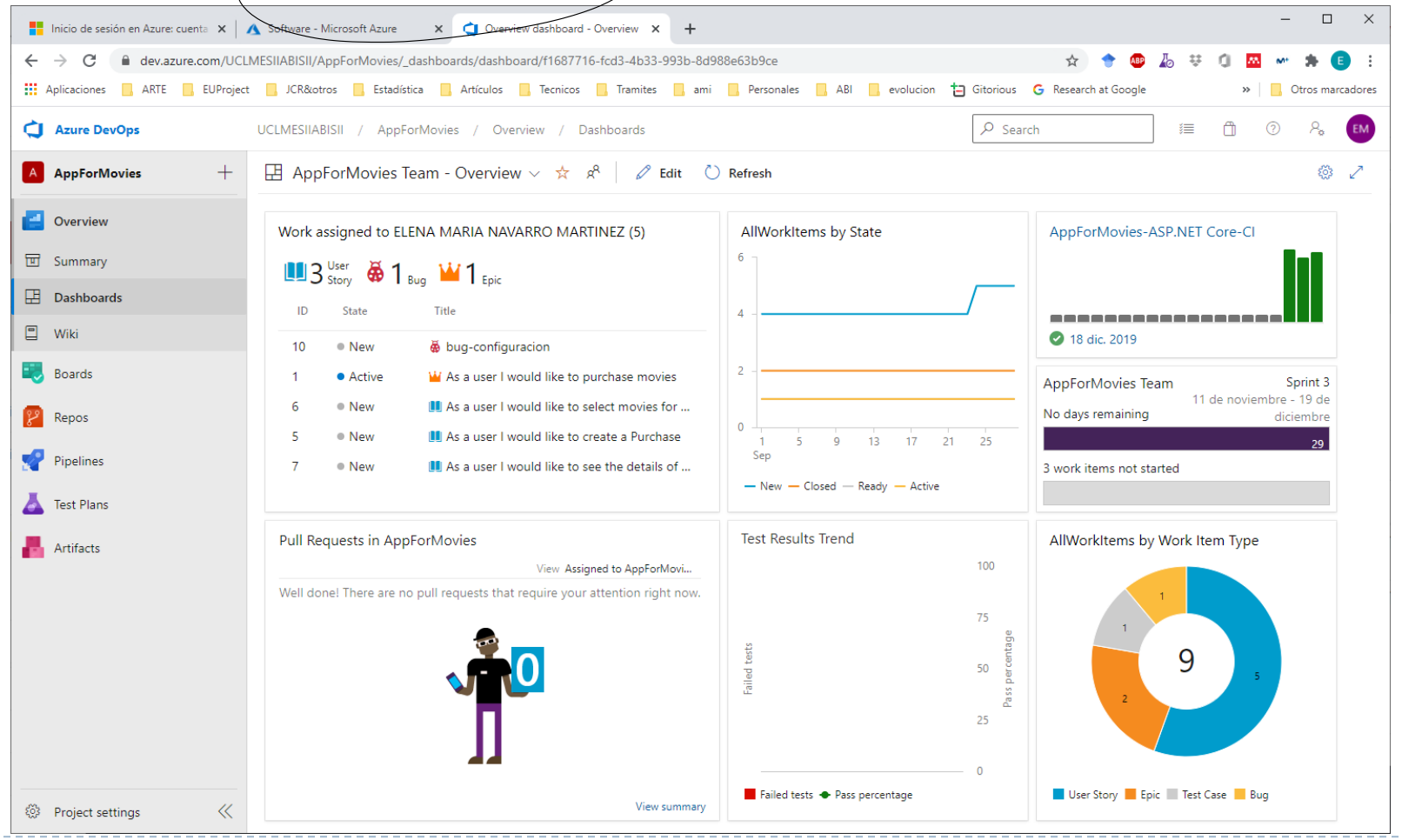
Information needed for managing CIs.

- Activities:
  - Defining types of Status Accounting Report (SAR) to be generated
  - Generating selected types of SAR by using the log of the Change Control process
  - Storing SAR in the repository of the Project
  - Distributing regularly SAR

Definir Sprints

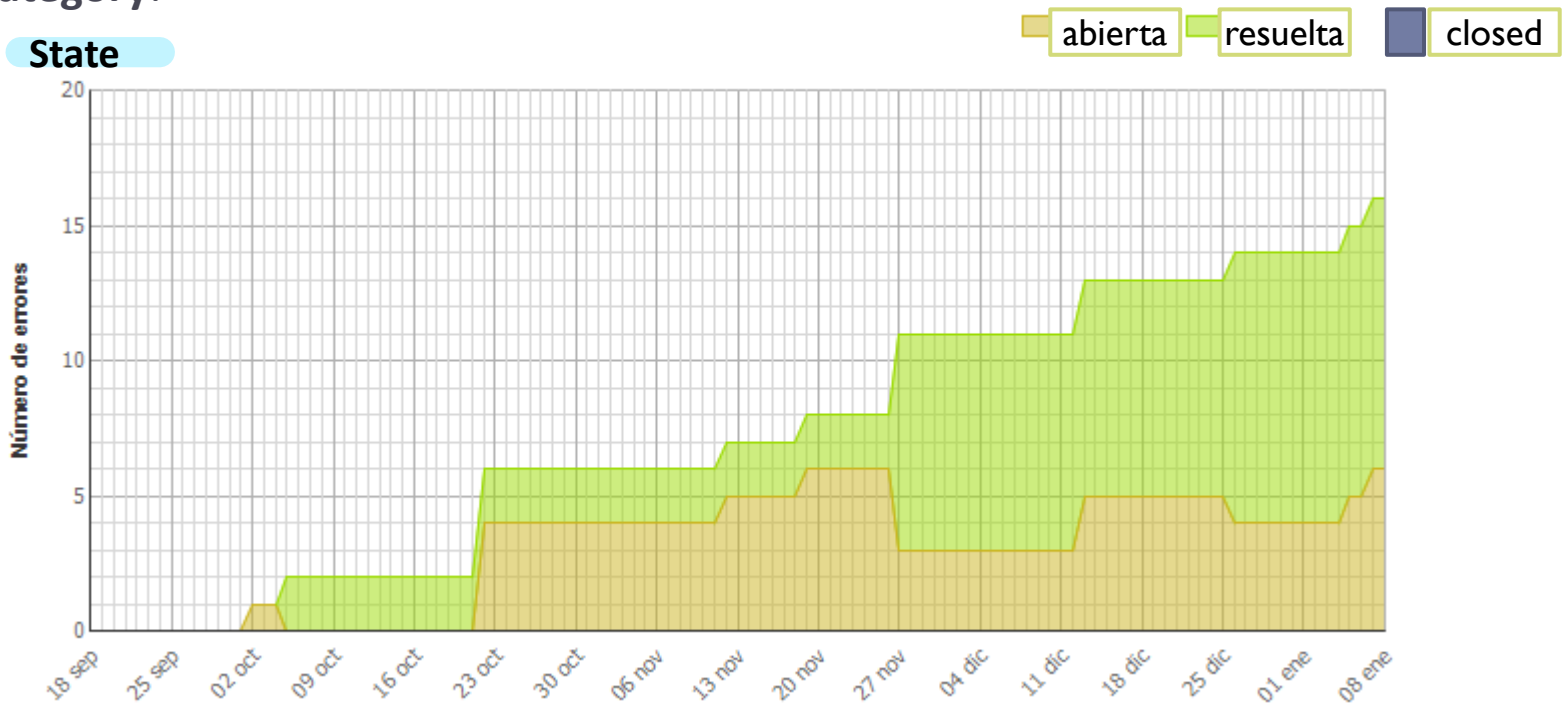
1.3 CM Process: Status Accounting

- Nowadays Status Accounting Reports are automatically generated: Dashboards



1.3 CM Process: Status Accounting

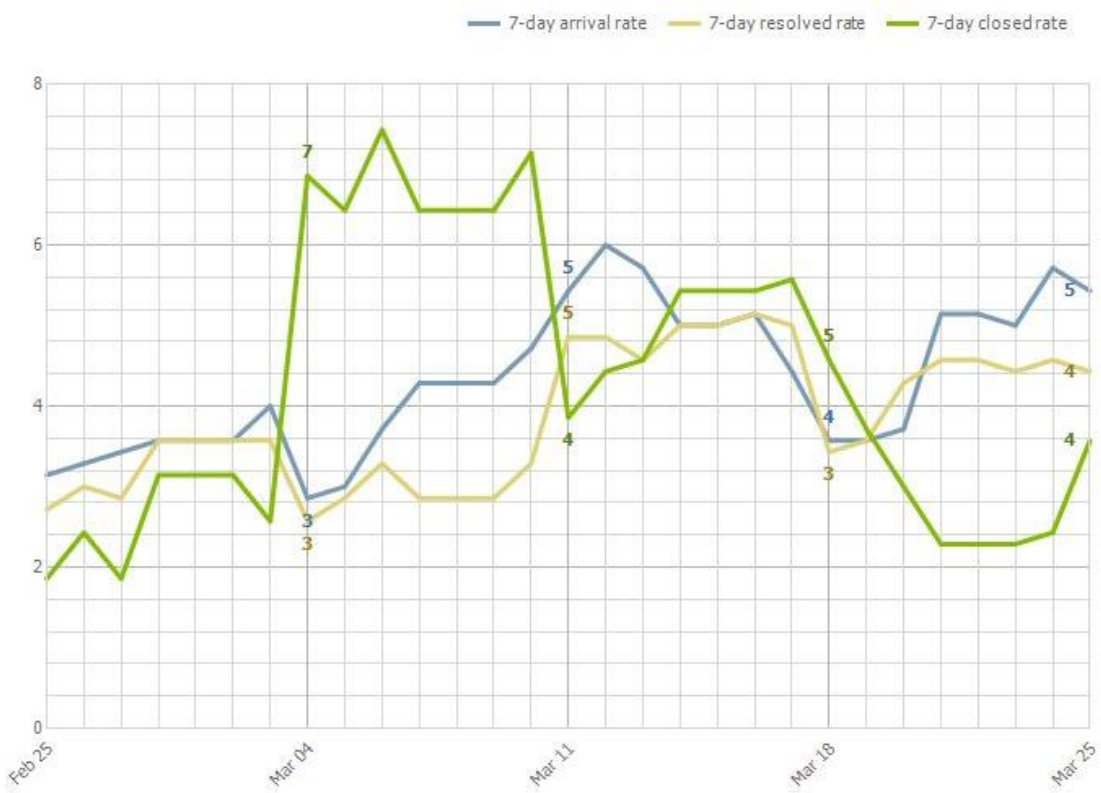
- Requirements of the Status Accounting Report:
  - **Frequency**: Daily, weekly, monthly, per iteration, per phase, at the end of the project
  - **Category**:
    - **State**



  - What questions can you answer?

1.3 CM Process: Status Accounting

- Requirements of the Status Accounting Report:
  - **Frequency**: Daily, weekly, monthly, per iteration, per phase, at the end of the project
  - **Category**:
    - **Trend**



  - What questions can you answer?

1.3 CM Process: Auditing

- Identification, Version and Change Control are used to manage the development process but they **cannot ensure that changes have been correctly implemented**.
  - Solution: **Audits:** [IEEE24765] an **independent examination** of a work product or set of work products **to assess compliance with** specifications, standards, contractual agreements, or other criteria
- It focuses on answering the following questions:
  - Have we implemented the planned change? Have we carried out additional modifications?
  - Have we reviewed the change implementation was correct?
  - Have we applied the SE standards?
  - Have we described the changes in the CIs? Did we specify date and author?
  - Have we applied the CM procedures to record, distribute and monitor the CRs?
  - Have we properly updated all the involved CIs?

Finish and merge → pull request (squash).

No conflicts – No migrations
- No delete.

Audits → Functional → Check whether every one of the requirements is solved.

EXAM → Functional: All tasks related to the requirements, checks state for change request, cfs configuration items and the result of the test. (Scan results and elements)
- Physical: Obtain versions of the configuration items that should be use to compile and generate the version of the product that is on audits. (Check I'm able to build the product.

BASELINE → Frose product if my change board approves it, but it cannot be modified anymore.

1.3 CM Process: Auditing

- Types of audits:
  - **Functional audits**: will determine whether the product complies with the goal requirements of the baseline
    - **How?**
  - **Physical Audits** determines whether the build baseline can be deployed using the repository
    - **How?**
- Inputs:
  - Requirements of the Baseline
  - Test results
  - CIs and hardware
  - Building instructions and tools
  - Configuration information, status and planning
  - Configuration Status
- When:
  - They are carried out just after integrating and testing the product and before baselining the product
- Who:
  - Management Officer, Customer, Independent agency
- Result: Report of Configuration Audit Findings

1.3 CM Process: Auditing

- If our CM process is formal, Audits are carried out by an independent team: **Quality Assurance Team**
- Alternatives to Configuration Audits:
  - **Alpha testing**: When the system has a lot of new, previously untested features, the development team look for evaluating the success or failure of the new features.
  - **Beta testing**: Development Team decides that some level of customer evaluation is needed before the final release of the product. Dev. Team is looking for (a high number of) beta testers to uncover bugs and faults in the system. (Chrome Releases: 2022 (googleblog.com))
  - **Test Readiness Reviews** (TRR): a review conducted to evaluate preliminary test results for one or more configuration items; to verify that the test procedures for each configuration item are complete, comply with test plans and descriptions, and satisfy test requirements; and to verify that a project is prepared to proceed to formal testing of the configuration items. Such review may be conducted for any hardware or software component [IEEE24765]
  - **Market Readiness Reviews** (MRR): distribution is ready

1.5 CASE for SCM

- Which features should a SCM tool offer?
  - Version management:
    - Product version
  - Change Management:
    - automate the change control procedures (workflows)
  - Problem tracking:
    - How and when a problem was fixed, how much time was taken, etc.
    - Notifying concerned personnel about arrival depending on criteria such as severity or impact
  - Promotion management:
    - Capture information and create trails to know what happened or to recreate an event or an item before or after a particular event
  - System building
  - Status accounting
  - Configuration audits
  - Access and security
  - Customization
  - Web enabling

Only thing not supported by azure devOps

- Which features does Azure DevOps support?

- Version management (Git – Tags)
- Workflows (Change requests – Epics – Tasks – Work Items)
- Problem Tracking
  - Task – branch – see implementation (Commits)
- We have to change all manually, we can but we have to do it.
- Promotion management → upload snapshots.
- System building → Yes
- Status Accounting → Yes, dashboards.
- Configuration auditives → Yes
  - , save all info
- Access and security → set security settings
- Customization
- Web enable

STATUS

Questions in exam.

1.6 Conclusions

- Benefits:
  - Best customer service
    - Customer has what he really wants Aditives
  - Improved productivity
    - Avoid duplicated efforts Change Control
  - Improved security
    - Avoid unauthorized changes Version Control → Synchronization and access control.
  - Defects reduction
    - Avoid defects are left unintentionally unresolved Change Control → no Change requests. And Status Accounting. → I can't conduct auditing and status accounting without change control.
  - Fastest defect finding and resolving
    - How did I do that? Version Control and Change Control → Describe problem in change control. change request.
  - Ensure that the proper system has been built
    - We use the right CIs versions according to the specifications Audits → Physical
  - Higher software reuse
- Which activity/activities and How?

Referencias

- [IEEE24765] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary
- ALEXIS LEON, Software Configuration Management Handbook, Second Edition, Artech House, 2005
- JEZ HUMBLE AND DAVID FARLEY, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Wiley, 2012

Unit 2. Software Testing: Techniques & Strategies

1. Introduction
2. Basic Concepts
3. Testing Types and Strategies

References

- BEIZIER, B., Testing and quality assurance, von Nostrand Reinhold, New York, 1984
- CARDWELL, K., Building Virtual Pentesting Labs for Advanced Penetration Testing, 2014
- COLLARD, J.F, BURNSTEIN, I, Practical Software Testing: A Process-Oriented Approach, Springer. 2003
- EVERETT, D., McLEOD, R. Software Testing. Testing Across the Entire Software Development Life Cycle, IEEE Press
- MOLYNEAUX, I., The Art of Application Performance Testing, 2009
- SOMMERVILLE, I., Software Engineering, 9th Edition. Addison Wesley, 2011.
- PFLEEGER, S. L., Ingeniería del Software: Teoría y Práctica. Prentice Hall, 2002.
- PRESSMAN, R. Ingeniería del software. Un enfoque práctico. 6ª Edición, McGraw-Hill, 2006.
- **WHITTAKER, J, ARBON, J., CAROLLO, J. How Google Tests Software, 2012**
- [IEEE24765] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary
- [IEEE829] IEEE 829:2008 Standard for Software and System Test Documentation

Goals

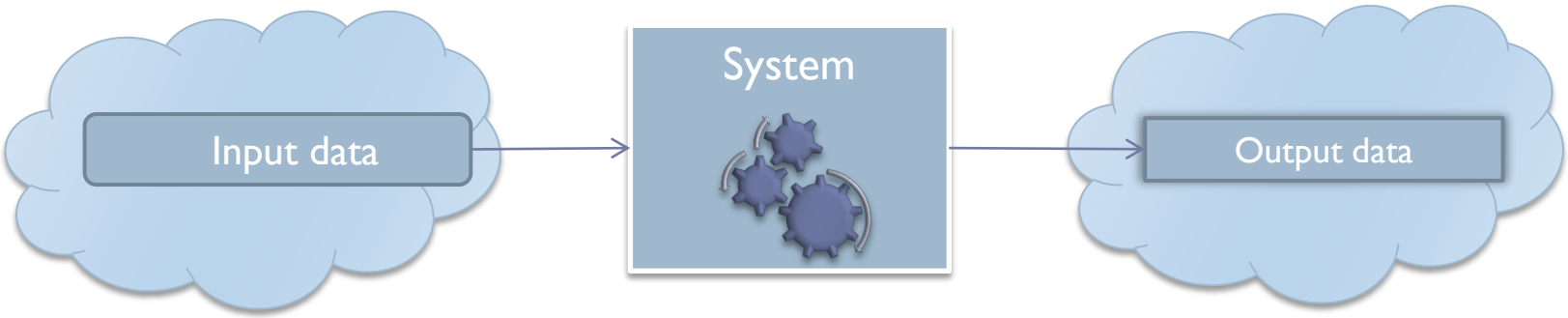
- To accept there is no way of carrying out exhaustive testing, that is, to test completely a product
- To understand why it is necessary to define testing boundaries
- To understand why we should use test case generation strategies

How Google thinks in testing

WHITTAKER, J, ARBON, J., CAROLLO, J. How Google Tests Software, 2012

1. Introduction

- Software testing:
  - the **dynamic verification** of the **behaviour** of a program on a **finite set of test cases**, suitably selected from the usually infinite executions domain, **against the expected behaviour** [SWEEBOK]



  - It is part of :
    - **Verification** & **Validation**: the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfil the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements.

2. Basic Concepts

- **Fault** (Defect, Defecto, bug): a problem which, if not corrected, could cause an application to either fail or to produce incorrect results. [IEEE24765]
- **Failure** (Fallo): an event in which a system or system component does not perform a required function within specified limits [IEEE24765]
  - It is essential to clearly distinguish between the *cause* of a malfunction (for which the term fault will be used here) and an undesired effect observed in the system's delivered service (which will be called a failure)
    - Fault=>Failure?
    - Does Testing reveal faults or failures?
- **Error**: it is used as synonym for other terms:
  - Failure: the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition
  - Failure: an incorrect result
  - Defect: an incorrect step, process, or data definition.
  - Error: a human action that produces an incorrect result, such as software containing a fault

People testing
+
People developing

2. Basic Concepts

- **Test Case (IEEE Std 829-2008):** A set of **test inputs**, **execution conditions**, and **expected results** developed **for** a **particular objective**, such as to exercise a particular program path or to verify compliance with a specific requirement.
  - Specification:
    - It describes the real input and output values
    - It defines specific constraints on the test procedures
    - It is separated of the test design in order to reuse this design for other test cases
  - Goal:
    - Analyse whether the system does not do what it should or it does what it should not do
    - Evaluate both valid and invalid inputs
  - Documentation:
    - They must be documented and stored
    - They must describe expected result
  - Execution:
    - We must inspect carefully the results of their execution
  - **Ideal test case?** → The one that shows a failure.

Real inputs
↓
Expected result
↓
Success

Exam question
↳ Is this a valid test case?
↓
If we don't have the expected output NO

I have to try to break my system to the limit → If it doesn't break → Good
Goal as tester → Break system

2. Basic Concepts

- **Test approach:** A particular method that will be employed to pick the particular test case values. This may vary in specificity from very general (e.g., black box or white box) to very specific (e.g., minimum and maximum boundary values).

- **Test design:** Documentation specifying the details of the test approach for a software feature or combination of software features and identifying the associated tests

- **Test:** (A) A set of one or more test cases. (B) A set of one or more test procedures. (C) A set of one or more test cases and procedures. (D) The activity of executing (A), (B), and/or (C).

- **Test bed**: an environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test

→ Every method we are going to create (software we are going to test)

- **System Under Test** (Software Bajo Pruebas, SUT): the parts of the computer-based software system (CBSS) to be tested.

Testing math operations -> always test 0.
Monkey test -> press all keyboard.

3. Testing Techniques

- Main aim of testing:
  - To **detect** as many **failures** as possible.
- Many techniques have been developed for this aim:
  - They attempt to **"break" a program** by being as **systematic** as possible in **identifying inputs** that will **produce representative** program **behaviours**
- Classification of testing techniques:
  - from the software engineer's intuition and experience, the specifications, the code structure, the real or imagined faults to be discovered, predicted usage, models, or the nature of the application.
  - ***white-box*** (also called *glass-box*), if the tests are based on information about how the software has been designed or coded, or as ***black-box*** if the test cases rely only on the input/output behaviour of the software

3. Testing Techniques

Design rule: *cover the maximum number of possibilities with the minimum number of test cases*

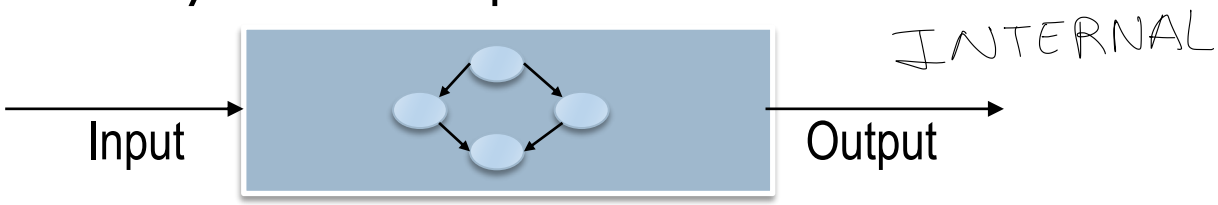
- **Test approach:** particular method that will be employed to pick the particular test case values
- Every product obtained as result of an engineering process can be tested following two alternatives:
  - **Black box testing:** testing a system or component whose inputs, outputs, and general function are known but whose contents or implementation are unknown or irrelevant

    - Testing just focuses on the interface of the SUT: inputs and outputs
    - Testing evaluates fundamental aspects of the SUT, revealing requirements and specification defects

3. Testing Techniques

Design rule: *cover the maximum number of possibilities with the minimum number of test cases*

- **Test approach:** particular method that will be employed to pick the particular test case values
- Every product obtained as result of an engineering process can be tested following two alternatives:
  - **White box testing**: testing that takes into account the internal mechanism of a system or component.



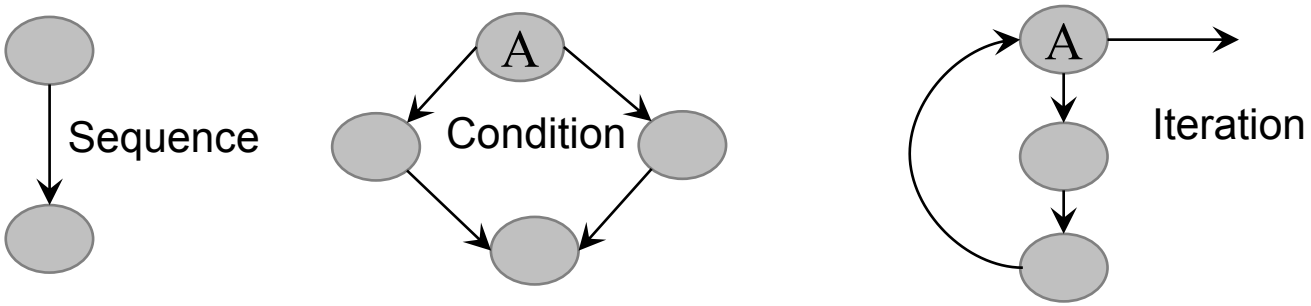
    - To design test cases, tester must have a knowledge about the inner structure of the SUT
    - White box testing is time consuming, as it is applied to smaller-sized pieces of software such as a module or member function
    - Useful for revealing design and code-based control defects, logic and sequence defects, initialization defects and data flow defects

3. Testing Techniques
White Box testing

- **White Box Testing: do they enable us to perform a exhaustive testing?**
  - Provided a program having 100 LOC written in C. After declaring some data structure, the program has two loops to be executed between 1 to 20 time every one, depending on some input conditions. The inner loop has four If-then-else structures.
  - There are 10^4 different paths, that is, different ways of running this program.
  - A *magic processor*, working 24 hoursX365 days per year, would need 3170 years to test this program.
  - **Conclusion**: it is impossible to exhaustively test all the paths of a program because the number of paths is too high

3. Testing Techniques
White Box testing

- **Goal:** to ensure internal components are working properly:
  - Usually it focuses on structural elements such as statements, decisions/branches, conditions.
    - Fact: all structured (prime) programs can be built from three basic primes-sequential (e.g. assignment), decision (if/then) and iterative (e.g. loop).



      - **Nodes** represent sequential statements combined in a block.
      - **Edges** represent transfer of control. The direction of the transfer depends on outcome of the condition in the predicate
  - Using the concept of a prime and the ability to use combinations of primes, a **flow diagram** for the SUT can be developed
  - Tester develops test cases that **exercise** (execute) these structural elements

3. Testing Techniques
White Box testing

- First, set the **test adequacy criterion** (also called **coverage criterion**): a stopping rule
  - A program is said to be adequately tested with respect to a given criterion if all the target structural elements have been exercised according to the selected criterion
    - For instance: *a test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed, respectively*
  - Usually, we decide between three adequacy criteria (or **coverage criteria**)
    - **Statement adequacy criterion:** all the statements in the SUT are executed at least once
    - **Decision adequacy criterion:** test cases must be designed so that each decision element in the code executed with all the possible outcomes at least once
    - **Condition adequacy criterion:** test cases must be designed so that each individual condition in a compound predicate takes on all possible values at least once
      - The stronger the coverage criterion, the _____ the number of test cases that must be developed to ensure complete coverage
  - **Coverage analysis**: to what extent the test cases satisfy the test adequacy criterion
    - For instance, *if only two branches were executed by the test cases and our SUT has four branches, which will the coverage be?*

3. Testing Techniques
White Box testing: **Path Coverage**

- **Goal**: to design test cases so that each independent path is executed at least once
  - **Path**: a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node
- Designing test cases according to the Path Coverage:
  1. Using the code (or its design), create the flow graph G.
  2. Calculate the McCabe's Cyclomatic Complexity V(G) of the flow graph
  3. Derive as many independent paths as V(G) determines
  4. Prepare the test cases so that the inputs cause the execution of these paths

EXAM EXERCISE

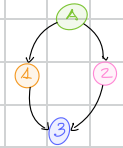
EXAMPLE: PATH COVERAGE

```
if (a > 5) {          A
    x = 7;            1
    y = 8;
} else {
    x = 2;            2
    y = 9;
}                     3
```

1 - Condition

A = True → Run 1

A = False → Run 2



→ Imagen we don't have else

Example 2 LOOP Condition

```
while (a > 100) {     A
    x = x + a;        1
    y = x + a;
}                     2
```

← This is a mistake → DON'T DO

↑ Not a condition → not 2 arrows leaving the node

Example 3 AND

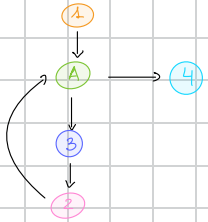
```
while (a > 100) && (b > 200) {     A  B
    x = a * b + x;                 4
    a--;
    b--;
}                                  2
```

Example 4 OR

```
while (a > 100) || (b > 200) {     A  B
    x = a * b + x;                 4
    a--;
    b--;
}                                  2
```

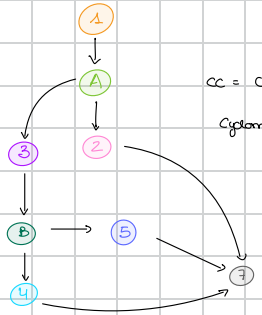
Example 5: FOR

```
for (int i = 0; i < 100; i++)      1  A  2
{   x = i * x;                     3
    y = x + i; }
                                   4
```



Example 6

```
int method (int a) {
    int x;                         1
    int y;
    if (a > 0)                     A
        return 0;                  2
    else {
        x = randomCa);             3
        y = x + 2
        if (y > 100)               B
            return x;              4
        else
            return y;              5
    }                              6
}                                  7
```



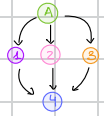
Small ones + Bigger one.

CC = Closed areas = 3

Cyclomatic complexity = Conditions + 1 = 3

Example 7

```
switch (a) {                       A
    case 1:                        1
        x = a + b;
        y = a + 2;
        break;
    case 2:                        2
        x = a + 3;
        y = a + b;
        break;
    default:                       3
        x = y - b;
}                                  4
```



I have to count as conditions the nº of cases (no default)

Conditions = 2 + 1 = 3

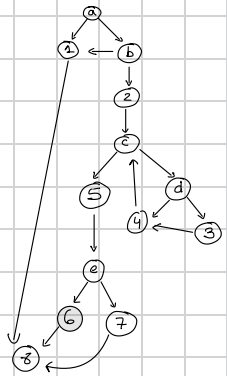
Regions = 3.

TEST CASES → Half mark.

Eglerocco path coverage

16/7/2026
25 a → b → 2 → c → 5 → e → 6 → 8

15/06/2026
65 a → b → 2 → c → d → 4 → c → 5 → e → 6 → 8



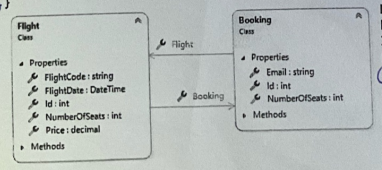
Flight 1
Flight 2
Booking 1

Path	Nodes	Input	FlightCode	FlightDate	IdFlight	N° Seats	Price	BId	User	NB	Return	Comment
a	a 1 8										Badrequest	Uncoverable → Bad request
b	ab 1 8										Bad Request	Uncoverable → Bad request
c	ab2c5e68	16/7/2026 25	None								Bad Request	1st cond 2nd not taken 3rd taken
d	ab2c5e78											None taken Uncoverable
e	ab2cd4c5e68	15/6/2026 65										Gone into the loop but not taken condition,
g	ab2cd4c5e78	Uncoverable										Gone into the loop but not taken condition
g	ab2cd34c5e68	Uncoverable										Loop, condition and 3rd condition tak
h	ab2cd34c5e78	Flight 2 Booking 1	YBG9076	15/06/2026	2	50	450€	1	Mario	50	OK (Flight 2)	Loop and condition taken 3rd not taken

Given the following code for the method *CreateRecommendedBooking* in the Booking controller related to the previous Use Case, apply the **Path Coverage technique** to create its unit tests.

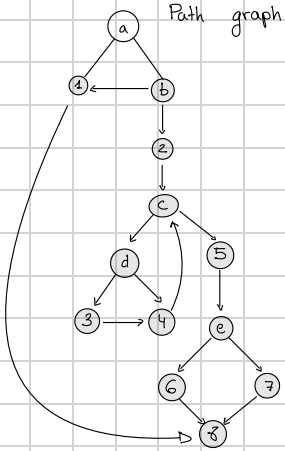
- The tables of the database **Flights** and **Bookings** are also considered as **input/output** of the test cases, depending on how they are used in the code.
- That indicated in the **return** instruction is considered as **outputs** for the test cases.

```
[HttpGet]
[ProducesResponseType(typeof(IList<Flight>), (int)HttpStatusCode.OK)]
[ProducesResponseType(typeof(string), (int)HttpStatusCode.BadRequest)]
public IActionResult GetFlightsAvailable(DateTime dateForBooking, int NumberOfSeats)
{
    if ((dateForBooking <= DateTime.Today) || (NumberOfSeats <= 0))
        return BadRequest("Flight date or the number of passengers are not correct");

    //it obtains all the flights that will fly on dateForBooking &&
    //have not booked yet (f.Booking == null)
    IList<Flight> flightsAvailable = _context.Flights
        .Where(f => f.FlightDate.Date.Equals(dateForBooking.Date) && f.Booking == null)
        .OrderBy(f => f.NumberOfSeats).ToList();

    int i = 0;
    IList<Flight> flights2return = new List<Flight>();

    while (i < flightsAvailable.Count)
    {
        if (flightsAvailable[i].NumberOfSeats >= NumberOfSeats)
            flights2return.Add(flightsAvailable[i]);
        i++;
    }

    if (flights2return.Count == 0)
        return BadRequest("There are no flights available for dateForBooking with such NumberOfSeats");

    else
        return Ok(flights2return);
}
```



Example of instances:
Flight1={XCT9675, 15/06/2026, 1, 20, 300.00€, null}
Flight2={YBG9076, 15/06/2026, 2, 50, 450.00€, **Booking1**}
Booking1={Marta.Martin@gmail.com, 1, 50, **Flight2**}

Example of tables:
Flights = {Flight1, Flight2}
Bookings = {Booking1}



Carmen Mª Noblejas Cameto
Héctor Ruiz López

Path	Nodes	Date For Booking	Number Of Seats	Flights Available	Return	Comment
a	a18	15/6/2021	x	None	BadRequest(Flight or nº of passengers incorrect)	Wrong date or seats
b	ab18	15/6/2026	0	None	BadRequest(Flight or nº of passengers incorrect)	Wrong seats
c	ab2c5e68	20/8/2030	3	None	BadRequest("No flights for dateBooking with such nº seats")	No flights for this date
d	ab2c5e78					Uncoverable.
e	ab2cd4c5e68	~~15/6/2026~~	~~70~~	None		Uncoverable (it would enter twice to while)
f	ab2cd4c5e78					Uncoverable (if it doesn't perform 3, can't perform 7).
g	ab2cd34c5e68					Uncoverable (if enters 3 don't enter 6)
h	ab2cd34c5e78	15/6/2026	20	Flight 2	Ok(F2)	1 flight available in date and seats required.
i	ab2cd34cd4c5e68					Uncoverable.
j	ab2cd34cd4c5e78					Uncoverable with the given examples because if F1 performs 3 F2 performs it 2.
k	ab2cd4cd34c5e68					Uncoverable
l	ab2cd4cd34c5e78	15/6/2026	35	Flight1, Flight2	Ok(F2)	Enters twice in the loop but just one has enough seats
m	ab2cd34cd34c5e68					Uncoverable
n	ab2cd34cd34c5e78	15/6/2026	10	Flight1, Flight2	Ok(F1,F2)	Both flights satisfy the conditions

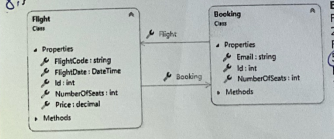
CORRECTION

→ Test independence paths

Given the following code for the method *CreateRecommendedBooking* in the Booking controller related to the previous Use Case, apply the **Path Coverage technique** to create its unit tests.

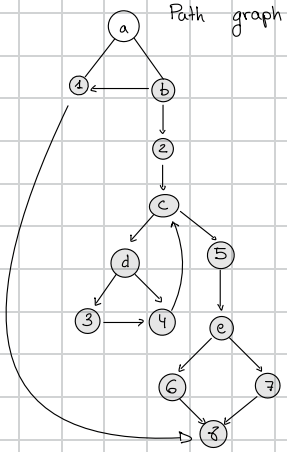
- The tables of the database **Flights** and **Bookings** are also considered as **input/output** of the test cases, depending on how they are used in the code.
- That indicated in the **return** instruction is considered as **outputs** for the test cases.

```
[HttpGet]
[ProducesResponseType(typeof(IList<Flight>), (int)HttpStatusCode.OK)]
[ProducesResponseType(typeof(string), (int)HttpStatusCode.BadRequest)]
public IActionResult GetFlightsAvailable(DateTime dateForBooking, int NumberOfSeats)
{
    if ((dateForBooking <= DateTime.Today) || (NumberOfSeats <= 0))
        return BadRequest("Flight date or the number of passengers are not correct");

    //it obtains all the flights that will fly on dateForBooking &&
    //have not booked yet (f.Booking == null)
    IList<Flight> flightsAvailable = _context.Flights
        .Where(f => f.FlightDate.Date.Equals(dateForBooking.Date) && f.Booking == null)
        .OrderBy(f => f.NumberOfSeats).ToList();

    int i = 0;
    IList<Flight> flights2return = new List<Flight>();

    while (i < flightsAvailable.Count)
    {
        if (flightsAvailable[i].NumberOfSeats >= NumberOfSeats)
            flights2return.Add(flightsAvailable[i]);
        i++;
    }

    if (flights2return.Count == 0)
        return BadRequest("There are no flights available for dateForBooking with such NumberOfSeats");
    else
        return Ok(flights2return);
}
```



Example of instances:
Flight1={XCT9675, 15/06/2026, 1, 20, 300.00€, null}
Flight2={YBG9076, 15/06/2026, 2, 50, 450.00€, **Booking1**}
Booking1={Marta.Martin@gmail.com, 1, 50, **Flight2**}

Example of tables:
Flights = {Flight1, Flight2}
Bookings = {Booking1}



Graph : 32 points

CC : 2 points

Each path : 3 points

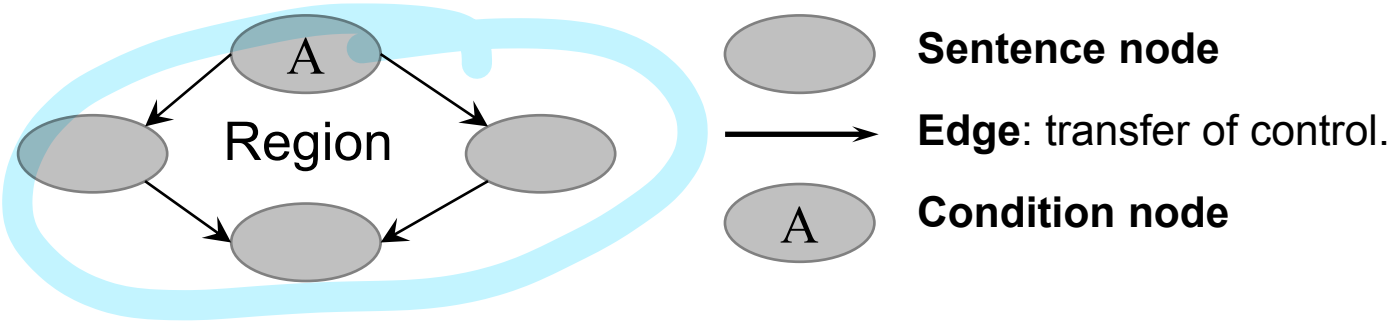
Each testcase : 8 points.

Cyclomatic complexity = Conditions + 1 = 6
↓ or
17 - 13 + 2 = 6

Path	Nodes	DateBooking	N° Seats	Flights	Bookings	Return	Comments.
a	a 1 8	1/10/2024	100	Fs	Bs	BadRequest("Flight date ....")	
b	a b 1 8	15/6/2026	0	Fs	Bs	BadRequest("Flight date ....")	
c	ab2c5e68	20/8/2030	100	FS	Bs	BadRequest("There are no ....")	
d	ab2c5e78						Uncoverable.
e	ab2cd4c5..	15/6/2026	100	FS	BS	BadRequest("There are no ....")	
f	ab2cd34c5e78	15/6/2026	10	Fs	Bs	Ok(Fl4)	Only first one because its → Covers D not booked.

3. Testing Techniques
White Box testing: **Path Coverage**

1. Using the code (or its design) we create the flow graph G.
   - Elements used while constructing the graph flow are:



If **While** **Switch/Case**

3. Testing Techniques
White Box testing: **Path Coverage**

1. Using the code (or its design) we create the flow graph G.
   - For this aim we should carry out the following steps:
     - First, mark **every condition in every predicate** of every control statement (IF, CASE, WHILE, UNTIL) and label it using a letter
       - **Hint: We don't mark decisions but conditions**
     - Second, group sequential sentences and label the group using a number
       - Hint: bear in mind that whenever we find ENDIF, ENDDO, END, etc. we will have a new node for that sentence
     - Third, using the created labels create the flow graph
       - Hint: remember to keep the transfer of control as it is in the code

3. Testing Techniques
White Box testing: **Path Coverage**

2. Calculate the **McCabe cyclomatic complexity** of the flow graph V(G)
   - It is a measure of the number of **independents paths** in a graph, and therefore, about the number of test cases to be defined in order to have **branch coverage:**
     - **Path**: a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node
       - It is designated by the sequence of nodes it encompasses (e.g. A-1-2-B-7-8)
     - **Independent path**: any new path through the graph that introduces a new edge that has not been traversed before the path is defined
   - It can be also used as a measure of testability of a piece of software:
     - The tester can use V(G) along with the past Project data to approximate the testing time and resources to test a software module

3. Testing Techniques
White Box testing: **Path Coverage**

2. Calculate the **McCabe cyclomatic complexity** of the flow graph V(G)
   - It can be calculated as:
     - The number of regions in the flow graph
       - Hint: the region surrounding the graph is also considered a region
     - V(G) = E – N + 2

       being E the number of edges and N the number of nodes
     - VG = P + 1

       being P the number of condition nodes in the flow graph

3. Testing Techniques
White Box testing: **Path Coverage**

3. Derive as many independent paths as V(G) determines:
   - Start out with one simple path in the graph, usually the shortest one
   - Iteratively add new paths to the set by adding new edges at each iteration until there are no new edges to add

3. Testing Techniques
White Box testing: **Path Coverage**

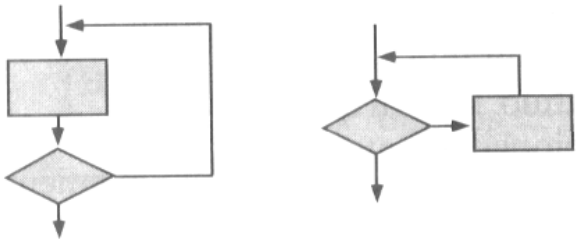
4. Prepare the test cases so that the inputs cause the execution of the defined paths
   - Select input data so that the conditions nodes are evaluated as each path needs
   - Identify likely outputs according to the path being run

3. Testing Techniques
White Box testing: **Additional approaches**

Adequacy criteria: When we are going to stop creating new test cases.

- **Data Flow testing:**
  - A variable is **defined** in a statement when its value is assigned or changed
  - A variable is **used** when its value is utilized in a statement without changing it
  - A **def-use path** is a path from a variable definition to a use
  - A predicate use (**p-use**) a variable that is used in a predicate
  - A computational use (**c-use**) the variable is used as part of a computation
  - Select the test adequacy criteria:
    - All defs
    - All p-uses
    - All c-uses/some p-uses
    - Al p-uses/some c-uses
    - All uses
    - All def-use paths

- **Loop testing**
  - Loops:
    - Among the most frequently used control structures
    - Many defects are associated with loops constructs
  - **Simple loops**: given a simple loop that can have a range of 0 to n iterations, tested cases should be developed so that there are:
    - Zero iterations of the loop,
    - 1 iteration of the loop,
    - 2 iterations of the loop
    - m iterations, where m<n,
    - n-1 iterations
    - n iterations
    - n+1 iterations (if possible)



3. Testing Techniques
White Box testing: **Additional approaches**

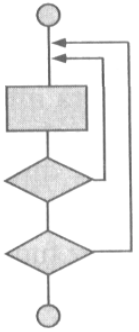
- **Loop testing**

```
public void exist(char[] value, int n_character, char character,
    out int pos, out bool found )
{
    pos = 0;
    if (n_character >= 1)
    {
        while ((value[pos]!=character) && (pos<n_character))
            pos++;
    }
    else
        pos = n_character;
    if (pos < n_character)
        found = true;
    else
        found = false;
}
```

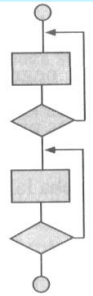
What would have happened if we had written the conditions in a different way?

White Box testing: **Additional approaches**

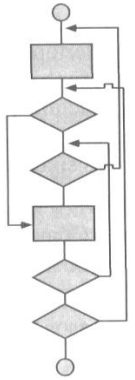
- **Loop testing:**
  - **Nested loops**:
    - A nested loop starts at the innermost loop.
    - For the innermost loop, conduct a simple loop test.
    - Work outward.
    - Continue until the outermost loop has been tested.



  - **Concatenated loops**: If the loops are independent then test them as simple loops, else test them as nested loops.



  - **Unstructured loops**:
    - Redesign



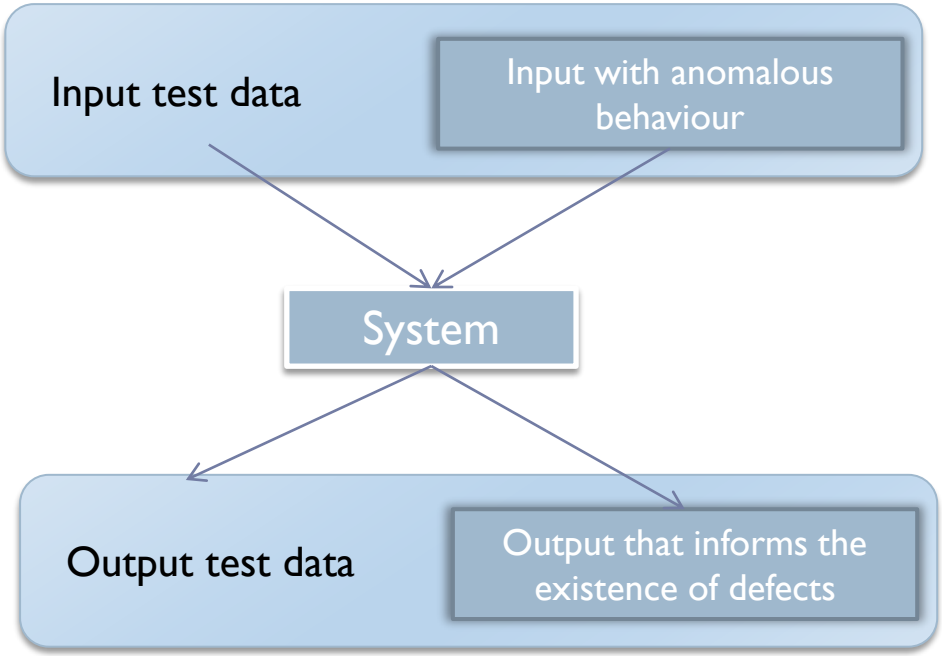
Avoid Spaguetti code ↓ Not worth testing.

3. Testing Techniques: **Black Box Testing**

- Tester considers SUT an opaque box:
  - Tester has **no knowledge** of its **inner structure**
  - Tester **only** has knowledge of **what SUT does**
- SUT can vary from a module, member function or a complex system
- Description of behaviour or functionality to be tested:
  - Formal specification
  - A set of pre- and post-conditions
  - A requirements specification
- How Black Box testing proceeds:
  - Tester provides inputs to the SUT, runs the test, and then determines if the outputs produced are equivalent to those in the specification
- Useful for revealing requirements and specification defects

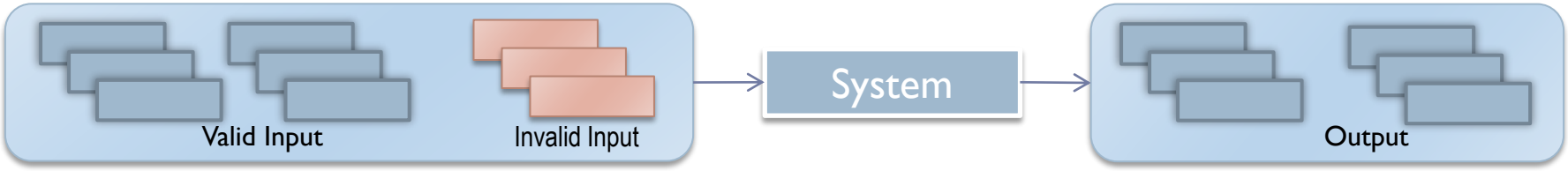
3. Testing Techniques
Black Box Testing

- Goal:
  - To effectively use the resources available by developing **a set of test cases** that gives the **maximum yield of defects** for the time and effort spent
- Techniques:
  - Equivalence Classes
  - Boundary Value Analysis
  - Random testing



3. Testing Techniques
Black Box Testing: **Equivalence Classes**

- **Equivalence Class Partitioning**:
  - Partitioning the input domain of the SUT



  - A test value in a particular class is equivalent to a test value of any other member of that class
    - If one test case in a particular equivalence class reveals a failure, then all the other test cases based on that class
  - Advantages:
    - Eliminate need for **exhaustive testing** (not feasible)
    - Select subset of test inputs with higher probability of detecting a defect
    - Cover a larger domain of inputs/outputs with a smaller subset

3. Testing Techniques
Black Box Testing: **Equivalence Classes**

- **Heuristics to identify** equivalence classes of an **input condition**:
  a) If an input condition for the SUT is specified as a **range of values**, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one for each end of the range
  b) If an input condition for the SUT is specified as a **finite number of values**, select one valid equivalence class that covers the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number
  c) If an input condition for the SUT is specified as a **set of valid input values**, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set
  d) If an input condition for the SUT is specified as a **"must be"** condition, select one valid equivalence class to represent the "must be" condition and one invalid equivalence class that does not include the "must be" condition
  e) If the input specification or any other information leads to the belief that an element in an equivalent class is not handled in an identical way by the SUT, then the class should be further divided into smaller equivalence classes (**Smaller classes**)

HEURISTIC

Example: $\frac{\leq 10}{> 20}$ Invalid inputs (2 classes) — CII

range of value – [10 – 20] — class valid inputs (CVI) — CII < 1 character

finicle number of values size(p) 1 – 5 characters CVI > 5 characters

set of valid input : AB, CU, TO, CR, GU (CVI) ——— Anything different (CII)

Must be number (CVI) ——— no number (CII)

TESTING TECHNIQUES FOR EQUIVALENCE CLASSES

- We don't need all combinations of valid inputs classes.
  - ↳ All classes but not all combinations
- Test invalid inputs one by one. → As many test cases as classes for invalid inputs.

EXERCISE OF EQUIVALENCE CLASS

FLIGTH CODE — CII ? < ? >

String of 7 characters (1)

First 3 characters Uppercase (must be) (2)

Last 4 characters Numbers [1000, 9999]

Parameter	Classes For Valid input	Classes For Invalid	Heuristic
Flight Code	7 characters 1	< 7 2	Number of values
		> 7 3	
	3 first characters	< 3 5	Number of values
	Uppercase 4	> 3 6	Smaller classes
		No UpperCase 7	Boolean
	[1000, 9999] 8	~~< 4~~	Range
		~~> 4~~	Boolean
		< 1000 9	Number values
		> 9999 10	
		No number 11	

Classes	Valid input	Invalid input
1 – 4 – 8	ABC1234	
2		ABC123
3		ABC12345
5		AB12345
6		ABCD123
7		abc1234
9		ABCD999
10		AB10000
11		ABCdefg

CORRECTION

Parameter	Classes for valid Inputs	Classes for Invalid Inputs	Heuristic
Flight code	3 Characters Uppercase (1)	< 3 characters (3) 2	Smaller Classes
		> 3 characters (4) 4	Finite number of elements
	[1000 - 9999] (2) 1000 9999	No uppercase (5)	Boolean
		< 1000 (6) 999	Range of values
		> 9999 (7) 10000	
		No number (8)	Boolean / must be

1st Heuristics
2nd Identify classes
3rd Test cases.

Classes	Valid Inputs	Invalid inputs	Outputs
1 - 2	AB2 1000		Oh + AB2 9999
3 - 2		AB 1000	Error
4 - 2		AB2X 1000	Error
5 - 2		aBc 1000	Error
1 - 6		AB2 999	Error
1 - 7		AB2 10000	Error
1 - 8		AB2 cosa	Error

BOUNDARY

Finite Number	CVI	
Range	upper	upper + 1
	lower	lower - 1

3. Testing Techniques
Black Box Testing: **Equivalence Classes**

- **Technique to develop the actual test cases**:
  1. For each input condition in the SUT, identify its equivalence classes using the heuristics
  2. Assign a unique identifier (a number) to each equivalence class
  3. Develop test cases for all **valid equivalence classes** until all have been **covered by** (included in) **a test case**. A given test case may cover more than one valid equivalence class
  4. Develop test cases for all **invalid equivalence classes** until all have been **covered individually**. This is to ensure that one invalid case does not mask the effect of another or prevent the execution of another

  This technique is also applicable to the output domain of the SUT

3. Testing Techniques
Black Box Testing: **Equivalence Classes**

- Using a file with the following format:
- being:

Employee-number	Employee-name	Months	Manager

  - Employee-number: a field that can have up to three digits (excluded 0).
  - Employee-name: an alphanumeric field whose length is 10.
  - Months: it indicates the number of months that an employee is working for a company. It is a field that can have up to three digits (including 0).
  - Manager: it is a field whose length is 1. It can be assigned to either «+» to indicate that the employee is a manager or «-» to indicate he/she is not a manager.

- A program computes a bonus for each employee, using the following rules:
  - P1 for managers with, at least, 12 months of service.
  - P2 for non-managers with, at least, 12 months of service.
  - P3 for managers with less than 12 months of service.
  - P4 for non-managers with less than 12 months of service.

- Assignment:
  - Create a table using the Equivalence Class technique where you will indicate in each row:
    - Input variable being analysed
    - Valid Classes
    - Invalid Classes
    - Applied Heuristic
  - Generate the test cases

3. Testing Techniques
Black Box Testing: **Equivalence Classes**

Input condition	Valid Classes	Invalid Classes	Heuristic
Employee-number	[1-999] 1	<=0 2 >999 3 No number 4	Range Boolean
Employee-name	10 characters 5	<10 6 >10 7	Number of values
Months	[0-11] 8 [12-999] 9	<0 10 >999 11 No number 12	Range Smaller classes Boolean
Manager	+ 13 - 14	Another charac 15	Set of values

Valid Classes	Input	Output
1 – 5 – 8 - 13	123, gumersindo, 9, +	P3
1 – 5 – 9 – 14	456, sebastiano, 13, -	P2

Invalid Cl.	Valid Cl.	Input	Output
2	- 5 - 9 – 13	0, gumersindo, 14, +	Error
3	- 5 - 9 – 14	1024, minotauros, 16, -	Error
4	- 5 - 8 – 13	abc, sebastiano, 8, +	Error
6	1 - - 8 – 13	123, cobos, 6, +	Error
7	1 - - 8 – 13	123, torreceballos, 3, +	Error
10	1 - 5 - - 13	123, margaritos, -1, +	Error
11	1 - 5 - - 14	123, margaritos, 1024, -	Error
12	1 - 5 - - 14	123, margaritos, abc, -	Error
15	1- 5 – 9 -	123, margaritos, 13, *	Error

3. Testing Types and Strategies
Black Box Testing: **Boundary Value Analysis**

- Whereas equivalence class directs to select test cases from any element of an equivalence class, **boundary value analysis** requires to select elements **close to the edges** so that both upper and lower edges of an equivalence class are covered by test cases. Heuristics:
  a) If an input condition for the SUT is specified as a **range of values**, develop valid test cases for the **ends of the range** and invalid test cases for possibilities **just above and below** the **ends of the range**
  b) If an input condition for the SUT is specified as a **number of values**, develop valid test cases for the **minimum and maximum** numbers and invalid test cases for one lesser and one greater than the maximum and minimum
  c) If the input or output of the SUT is an ordered set, such as a table or a linear list, develop test cases that focus on the first and last elements of the set
- Applying for testing both inputs and outputs of the SUT

3. Testing Types and Strategies
Black Box Testing: **Boundary Value Analysis**

Input condition	Valid Classes	Invalid Classes	Heuristic
Employee-number	1 (1) 999 (2)	0 (3) 1000 (4) No number (5)	Range Boolean
Employee-name	10 characters (6)	9 characters (7) 11 characters (8)	Number of values
Months	0 (9) 11 (10) 12 (11) 999 (12)	-1 (13) 1000 (14) No number (15)	Range Smaller classes Boolean
Manager	+ (16) - (17)	Another character (18)	Set of values

Valid Classes	Input	Output
1 – 6 – 9 - 16	1, sebastiano, 0, +	P3
2 – 6 – 10 – 17	999, sebastiano, 11, -	P4
1 – 6 – 11 - 16	1, sebastiano, 12, +	P1
1 – 6 – 12 - 17	1, sebastiano, 999, -	P2

Invalid Cl.	Valid Cl.	Input	Output
3	– 6 – 9- 16	0, sebastiano, 0, +	Error
4	– 6 – 9- 16	1000, sebastiano, 0, +	Error
5	– 6 – 9- 16	abc, sebastiano, 0, +	Error
7	1 - – 11 - 16	1, sebastian, 12, +	Error
8	1 -– 11 - 16	1, sebastianez, 12, +	Error
13	1 - 6 - - 16	1, margaritos, -1, +	Error
14	1 - 6 - – 16	1, margaritos, 1000, +	Error
15	1 - 6 - – 16	1, margaritos, abc, +	Error
18	1 - 6 - 9 –	1, margaritos, 0, *	Error

3. Testing Types and Strategies
Black Box Testing: **Error guessing**

- Zero is prone to cause failures:
  - Division by zero
- When we enter a variable number of values, tester should evaluate what happens when we do not enter anyone or just one value
- Check whether developer could have misunderstood the specification
- Evaluate what a user could enter while using the SUT
- List the most likely mistakes that developers can make, and other situations prone to errors
- ...

Unit 3. Unit Testing

1. Introduction
2. Test Levels
3. Unit Testing

References

- BEIZIER, B., Testing and quality assurance, von Nostrand Reinhold, New York, 1984
- CARDWELL, K., Building Virtual Pentesting Labs for Advanced Penetration Testing, 2014
- COLLARD, J.F, BURNSTEIN, I, Practical Software Testing: A Process-Oriented Approach, Springer. 2003
- EVERETT, D., McLEOD, R. Software Testing. Testing Across the Entire Software Development Life Cycle, IEEE Press
- MOLYNEAUX, I., The Art of Application Performance Testing, 2009
- SOMMERVILLE, I., Software Engineering, 9th Edition. Addison Wesley, 2011.
- PFLEEGER, S. L., Ingeniería del Software: Teoría y Práctica. Prentice Hall, 2002.
- PRESSMAN, R. Ingeniería del software. Un enfoque práctico. 6ª Edición, McGraw-Hill, 2006.
- **WHITTAKER, J, ARBON, J., CAROLLO, J. How Google Tests Software, 2012**
- [IEEE24765] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary
- [IEEE829] IEEE 829:2008 Standard for Software and System Test Documentation

Goals

- To know how we should plan our testing process
- To understand how the definition of a testing process is helpful
- To learn to what extent how Testing tools can help to test software

1. Introduction

- Software testing **no longer a post-coding** phase:
  - As Software Quality is one activity of prevention (to prevent is much better than to correct problems) and Software Quality entails Testing as one of its dynamic (execute the SUT) processes
  - Then, Testing can be seen, as a means for providing information about the functionality and quality attributes of the software

- Software testing is, or should be, **pervasive throughout the entire development and maintenance life cycle**
  - Indeed, planning for software testing should start with the early stages of the software requirements process

- Test plans and procedures should be systematically and continuously developed—and possibly refined—as software development procedes
  - Goal: to provide useful input for software designers and help to highlight potential weaknesses, such as design oversights/contradictions, or omissions/ambiguities in the documentation

2. Test Levels

Best test case → The one that shows a failure

- Which should the main goal of the testing process be?
  - Maximize use of time and resources
  - For this aim, testers will need to develop effective test cases:
    - Greater probability of detecting defects
    - A more efficient use of organizational resources
    - A higher probability for test reuse
    - Closer adherence to testing and project schedules as well as budgets
    - The possibility of delivery a higher-quality software product
- Software testing is usually performed at different levels throughout the development and maintenance processes. Levels can be distinguished based on:

  → Element that we are going to test

  - the object of testing, which is called the **target**, The target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or an entire system. Three test stages can be distinguished: **unit**, integration, and system. These three test stages do not imply any process model, nor is any one of them assumed to be more important than the other two.

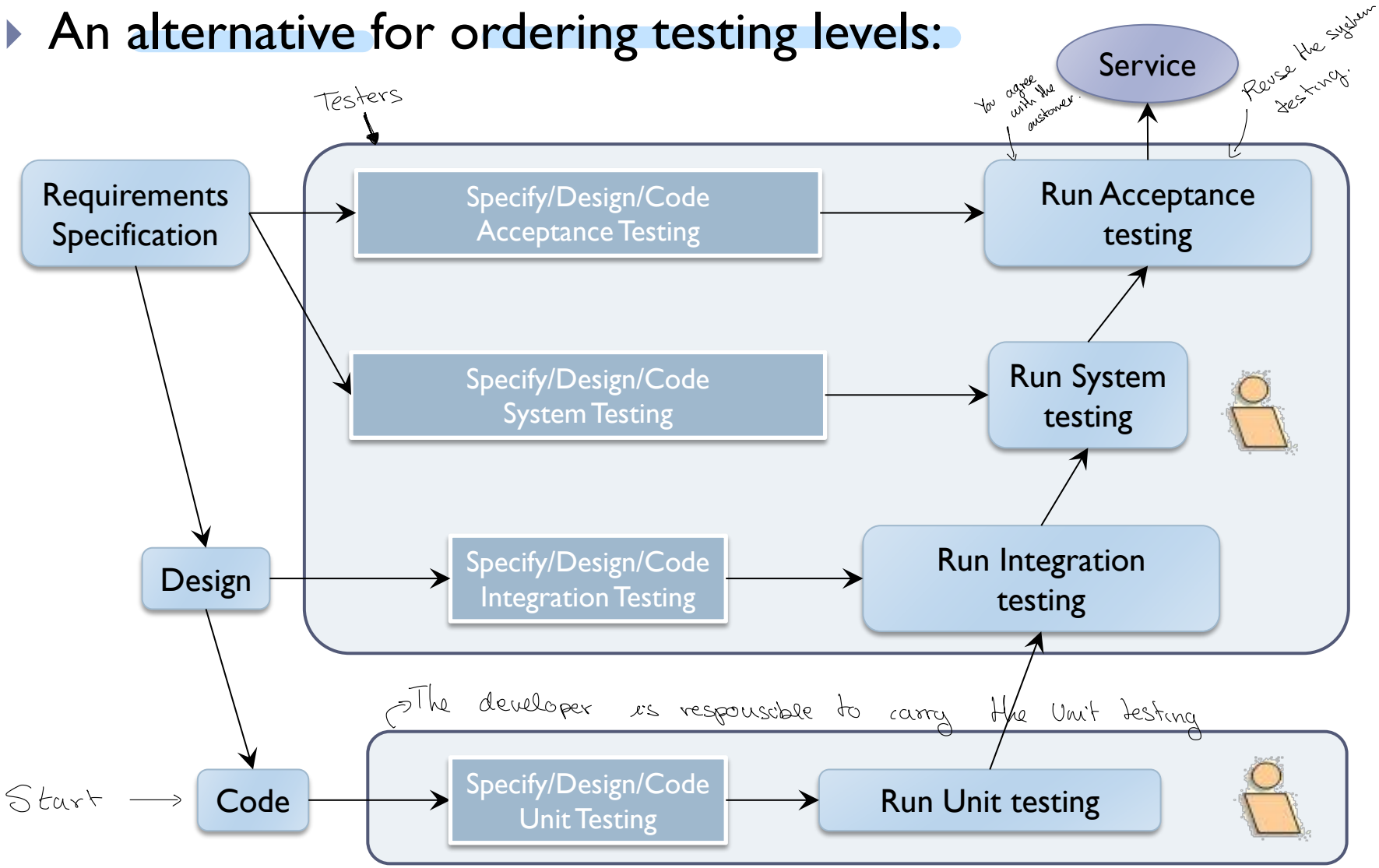
    My controllers

  - the purpose, which is called the **objective** (of the test level). Test cases can be designed to check that the functional specifications are correctly implemented, several other non-functional properties (including performance, reliability, and usability, among many others), etc

    → For functional requirements, performance... (Approach)

2. Test Levels: V Model

- An alternative for ordering testing levels:



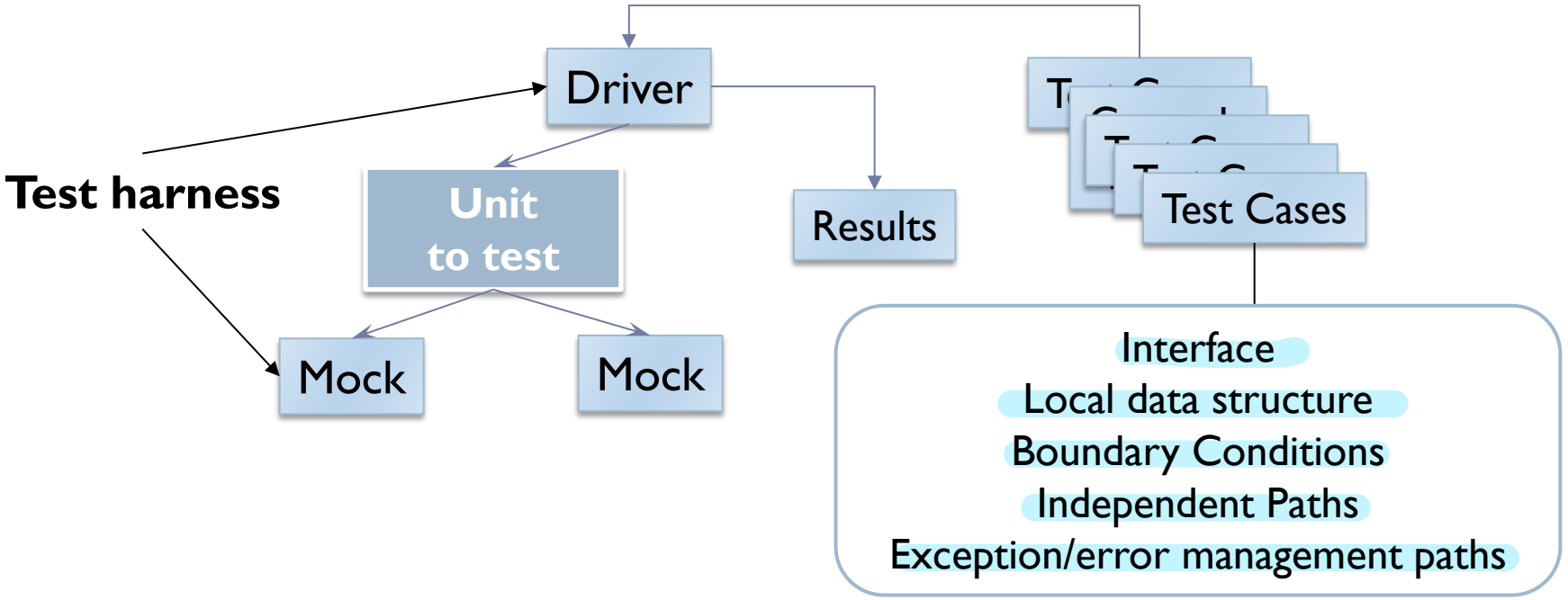
More understandable → Easier test

3. Unit testing

- Testing of individual routines and modules by the developer or an independent tester in order to ensure that there are no analysis or programming errors
- First, what a unit is:
  - **Unit**: the smallest testable software component:
    - Imperative language: a function
    - **Object-oriented language: method or class**
    - Component-Based Software Development: a component
- Constraint: unit with a high level of **cohesion** and low level of **coupling**
  - ↳ My unit is related to other units.
  - ↳ Unit is highly specialized → specific behavior
- Why:
  - Test is easier to design, run, record and analyse
  - Early detection of failures
- Automation: XUnit, NUnit, JUnit, SimpleTest

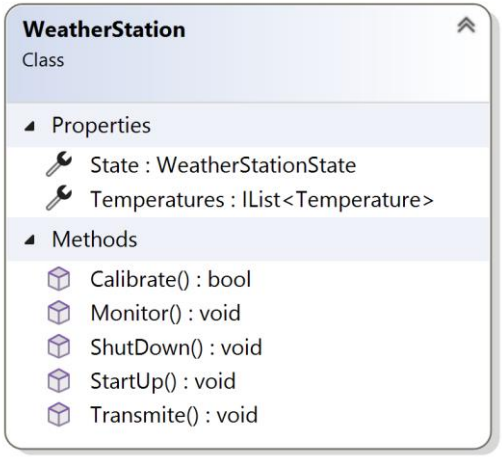
3. Unit testing

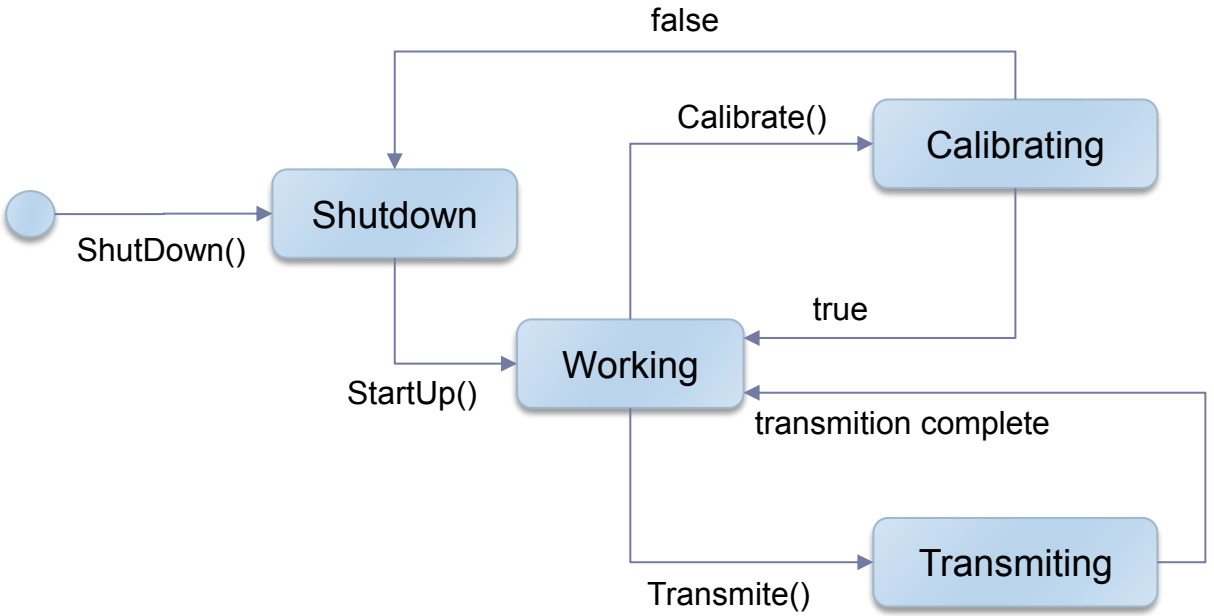
- Additional code to automate unit testing:
  - **Driver code** → Responsible of calling the unit under test passing the input of the test case and checking whether the expected and the actual results are the same to know if it pass the test or not.
  - **Mock code** → If our unit uses other units, we create mock code to replace the units that the unit under test is using
    - As it means additional workload, should we put off the development of driver and stub code till integration testing?



3. Unit testing

- Object-oriented systems: tester should focus on
  - Getter & setter
  - **Methods**
  - States:





  - Advantages:
    - As OO paradigm promotes encapsulation=> does it reduce the re-testing when a class is changed? No, because the external behavior doesn't change
    - Is it necessary to re-test the inherited subclasses? Modify parent → need to modify the children classes (Test subclasses).

Unit 4. DevOps

1. Introduction
2. Continuous Integration
3. Continuous Delivery
4. Continuous Deployment

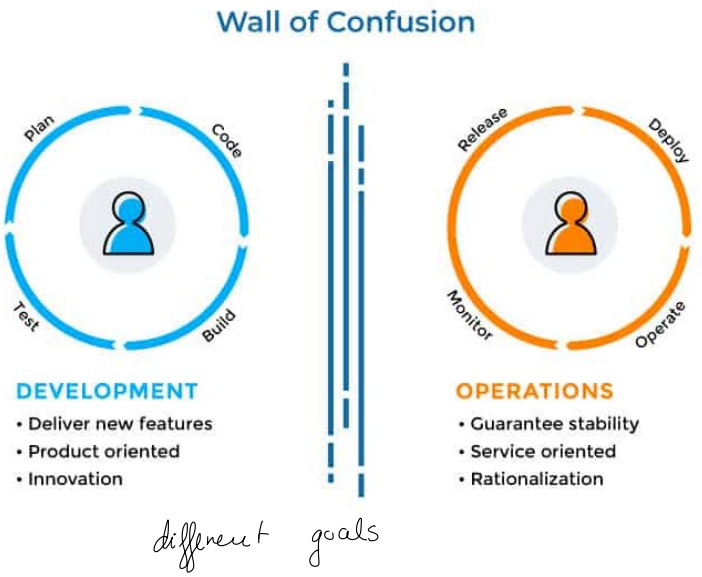
4.1 Introduction

- From 70s till present:
  - Jeffrey Immelt, CEO of General Electric, stated, "Every industry and company that is not bringing software to the core of their business will be disrupted."

	1970s–1980s	1990s	2000s–Present
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rails, PHP, etc.
Cycle time	1–5 years	3–12 months	2–12 weeks
Cost	$1M–$100M	$100k–$10M	$10k–$1M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

4.1 Introduction

- Dev & Ops:
  - Development teams will take responsibility for responding to changes in the market, deploying features and changes into production as quickly as possible
  - Operations teams will take responsibility for providing customers with IT service that is stable, reliable, and secure, making it difficult or even impossible for anyone to introduce production changes that could jeopardize production



"The core, chronic conflict":

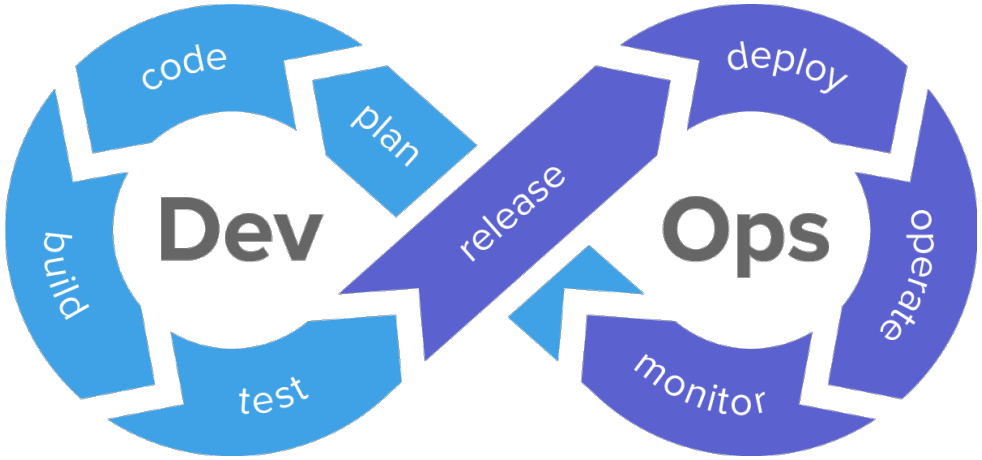
- Development and IT Operations have diametrically opposed goals and incentives
- It leads to poor software and service quality, and bad customer outcomes, as well as a daily need for workarounds, firefighting, and heroics

4.1 Introduction

- **DevOps**: *Puppet Labs' State Of DevOps Report* collected data from over **twenty-five thousand technology professionals**:
  - Code and change deployments: **thirty times more frequent**
  - Time required to go from "code committed" to "successfully running in production" (lead time): **two hundred times faster**
  - Production deployments: **sixty times higher change success rate**
  - Mean time to restore service: **168 times faster**
  - Productivity, market share, and profitability goals: two times more likely to exceed
  - Market capitalization growth: 50% higher over three years
  - Integrating security objectives into DevOps: 50% less time remediating security issues.
  - Reliability metrics
  - Throughput metrics
  - Organizational performance metrics
  - Higher employee job satisfaction, lower rates of **employee burnout**

4.1 Introduction

- DevOps: A set of software development tools, processes, and practices, combining software development (Dev) with information technology operations (Ops) to facilitate the software development lifecycle.
- DevOps needs an automated delivery cycle that includes planning, development, testing, release, deployment and monitoring with the active cooperation of the different team members.
  - Monitoring The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics.



Core activities:

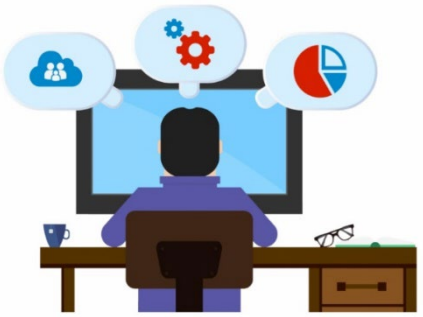
- Continuous Integration
- Continuous Deployment

4.2 Continuous Integration (CI)

- process of integrating new code written by developers with a mainline or "master" branch frequently throughout the day.
- to make sure that the integrations were successful, CI systems will usually run a series of tests automatically upon merging in new changes.
- when these changes are committed and merged, the tests automatically start running to avoid the overhead of people having to remember to run them
- Advantages:
  - Your software is proven to work (assuming a sufficiently comprehensive set of automated tests) with every new change—and you know the moment it breaks and can fix it immediately.
  - The teams that use CI effectively are able to deliver software much faster, and with fewer bugs, than teams that do not.
  - Bugs are caught much earlier in the delivery process when they are cheaper to fix, providing significant cost and time savings.

4.2 Continuous Integration (CI)

- Three things to start with CI
  - Version Control. Everything in your project must be checked into a single version control repository: code, tests, database scripts, build and deployment scripts, and anything else needed to create, install, run, and test your application.
  - Automated Build. You need to be able to run your build process in an automated way from your continuous integration environment so that it can be audited when things go wrong. Your build scripts should be treated like your codebase.
  - Agreement of the Team. You need everyone to check in small incremental changes frequently to mainline and agree that the highest priority task on the project is to fix any change that breaks the application.



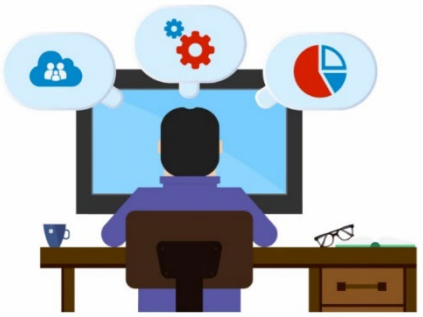
You

Server

Commit Stage
1. Compile
2. Build
3. Test

4.2 Continuous Integration (CI)

- Practices to be applied
  - Check In Regularly. At least a couple of times a day.
  - Create a Comprehensive Automated Test Suite. It's essential to have some level of automated testing to provide confidence that your application is actually working. Usually: unit, integration and acceptance testing
  - Keep the Build and Test Process Short. Otherwise your team will abandon the practice
  - Managing Your Development Workspace: use Configuration Management not just of source code, but also of test data, database scripts, build scripts, and deployment scripts. The whole team uses the same development environment



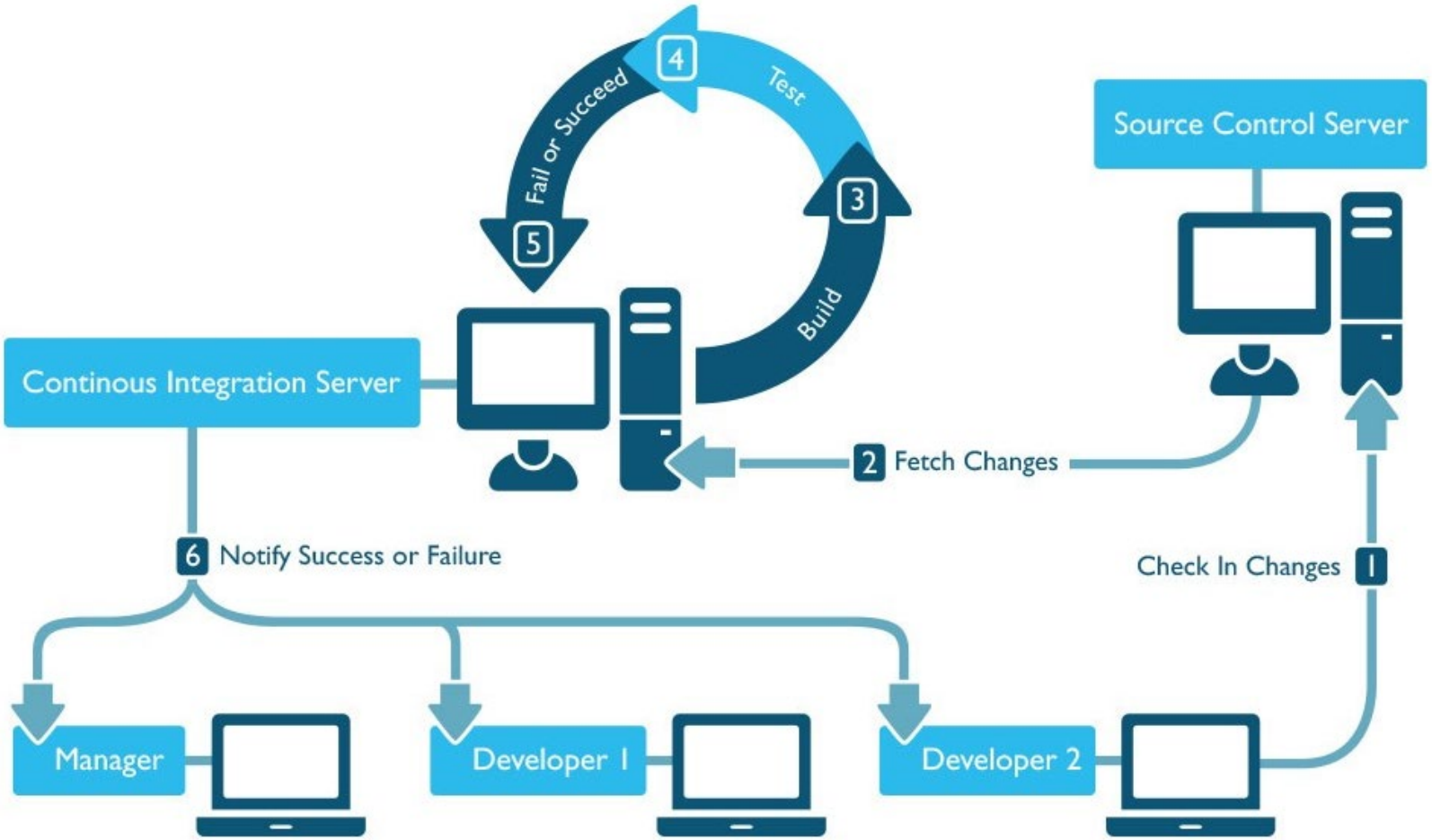
You

Server

Commit Stage
1. Compile
2. Build
3. Test

4.2 Continuous Integration (CI)

- Continuous Integration Architecture:

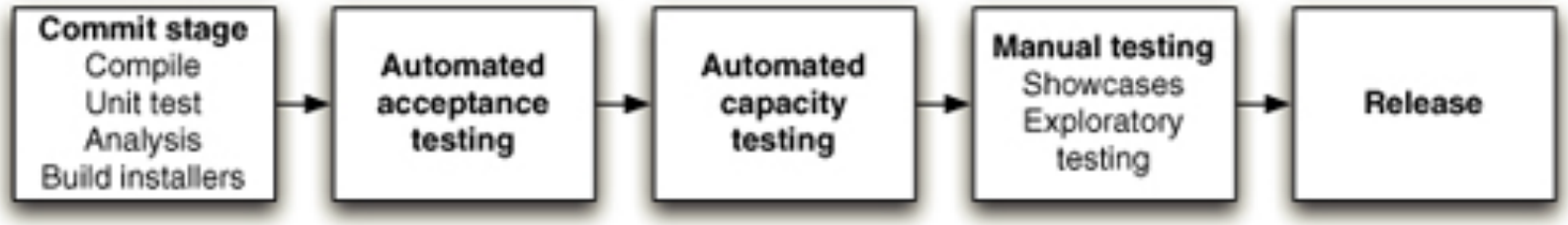


4.3 Continuous Delivery

- The Release Candidate:
  - A change to your code may or may not be releasable.
  - If you were to look at a change —whether it is new functionality, a bugfix, or a retuning of the system to achieve some change in performance— and ask, “Should we release this change?” Your answer is the build, deployment, and test process that we apply to that change that validates whether the change can be released.
  - Every change is, in effect, a release candidate. Every time a change is committed to version control, the expectation is that it will pass all of its tests, produce working code, and can be released into production.
- Continuous delivery will be in charge of deploying every release candidate into production (or production like environments):
  - Every change committed to version control is supposed to enhance the system that we are working on. How do we know if that is true? The only way in which we can tell is through exercising the software to see if it achieves the value that we had expected.

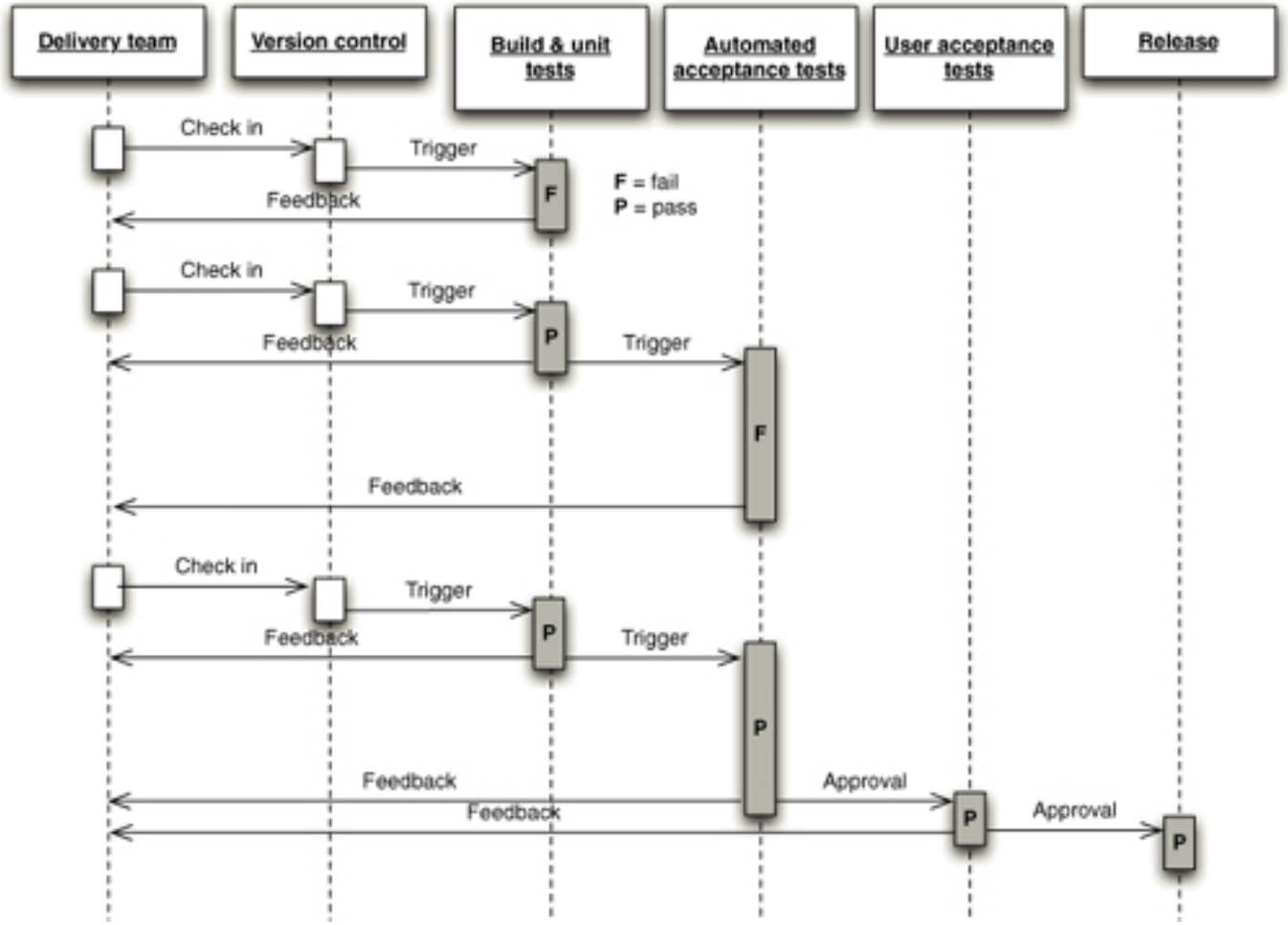
4.3 Continuous Delivery

- The process of deploying changes to production (or production-like environments) by defining tests and validations to minimize risk.
- Continuous delivery means that changes get deployed into production under request.
- The more quickly software changes make it into production, the sooner individuals see their work in effect.
- Continuous delivery also gets the product out to the customer faster, which can mean increased customer satisfaction.
- A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process.



4.3 Continuous Delivery

- Deployment pipeline how changes move through



4.3 Continuous Delivery

- A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process.

- The aim of the deployment pipeline is threefold.
  1. First, it makes every part of the process of building, deploying, testing, and releasing software visible to everybody involved, aiding collaboration.
  2. Second, it improves feedback so that problems are identified, and so resolved, as early in the process as possible.
  3. Finally, it enables teams to deploy and release any version of their software to any environment at will through a fully automated process.

4.3 Continuous Delivery

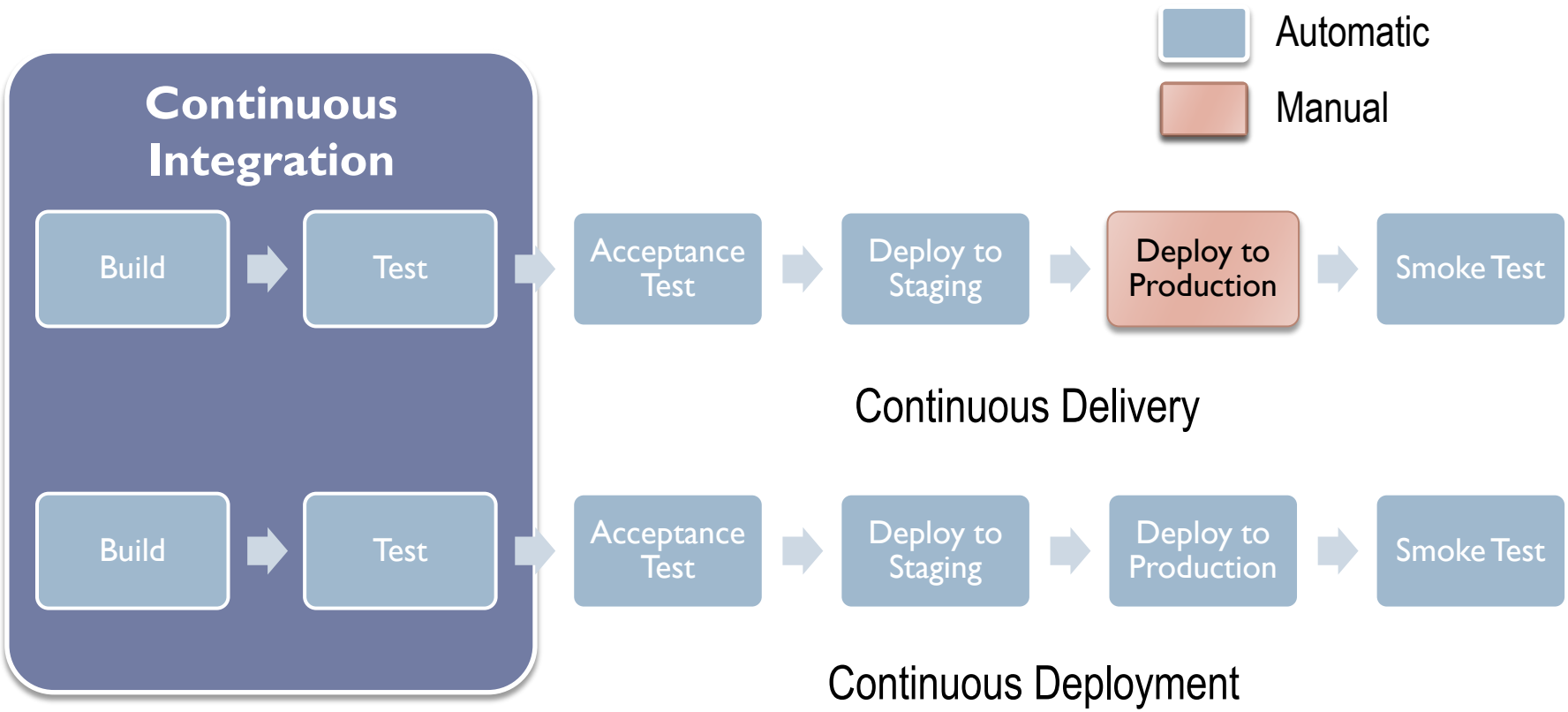
- Principles:
  1. Create a Repeatable, Reliable Process for Releasing Software
  2. Automate Almost Everything
  3. Keep Everything in Version Control
  4. If It Hurts, Do It More Frequently, and Bring the Pain Forward
  5. Build Quality In
  6. Done Means Released
  7. Everybody Is Responsible for the Delivery Process
  8. Continuous Improvement

4.4 Continuous Deployment (CD)

- Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.
- Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a Release Day anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.

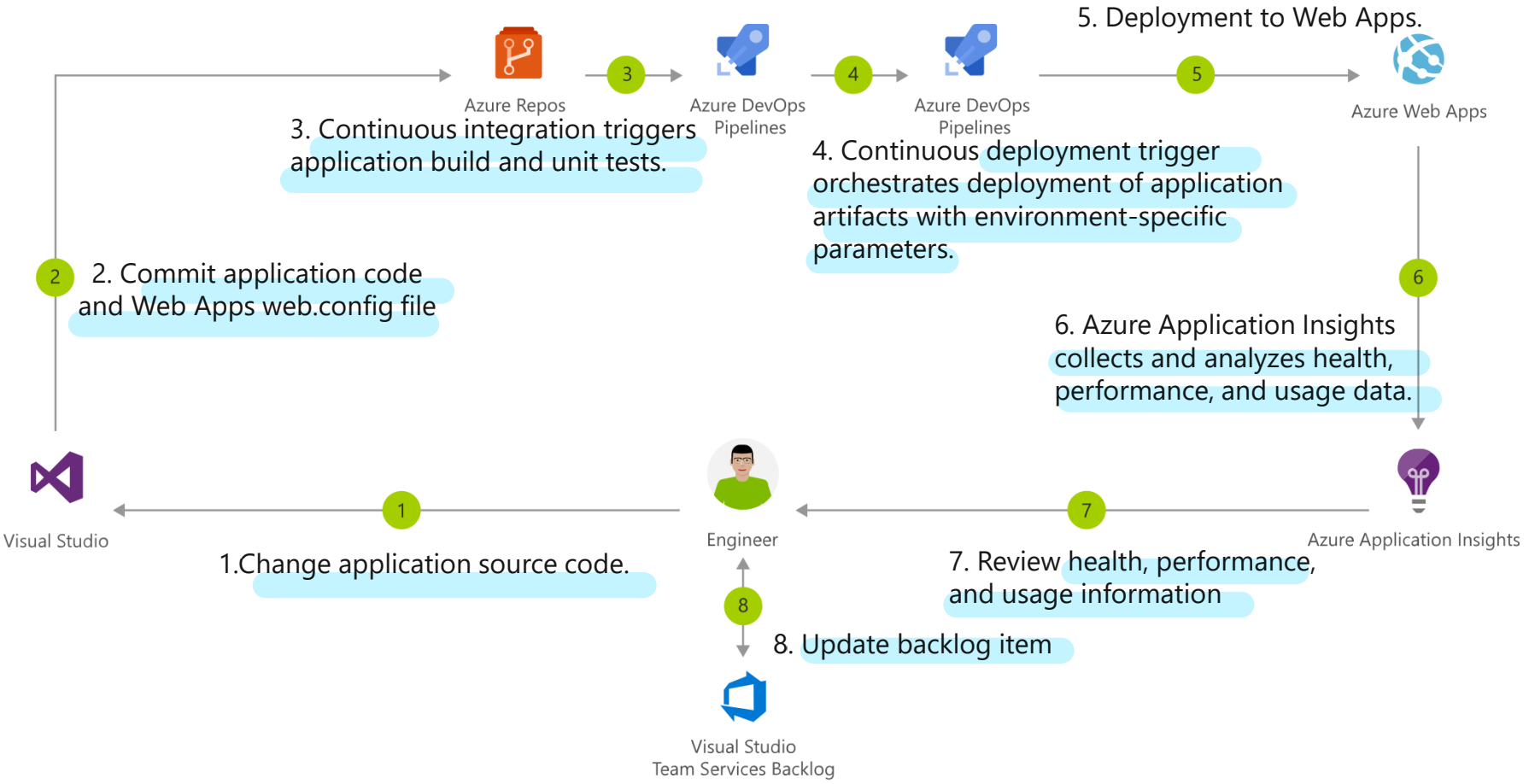
4.4 Continuous Deployment (CD)

- Continuous Integration vs Continuous Delivery vs Continuous Deployment



4.4 Continuous Deployment (CD)

- CI/CD for Azure Web Apps (AppForMovies)



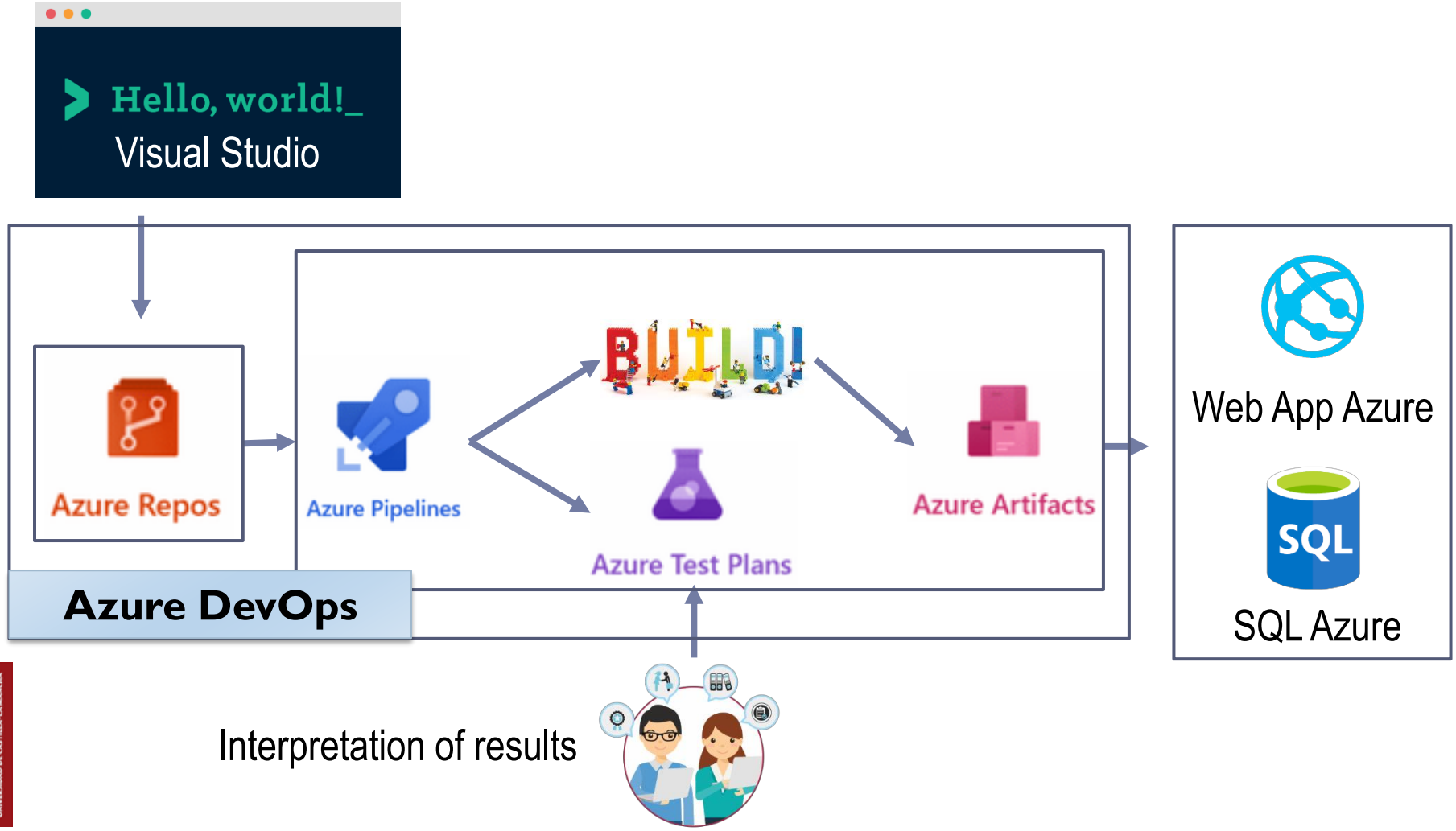
Referencias

- [IEEE24765] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary
- JEZ HUMBLE AND DAVID FARLEY, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Wiley, 2012

Visual Studio 2022 - Azure: Continuous Integration

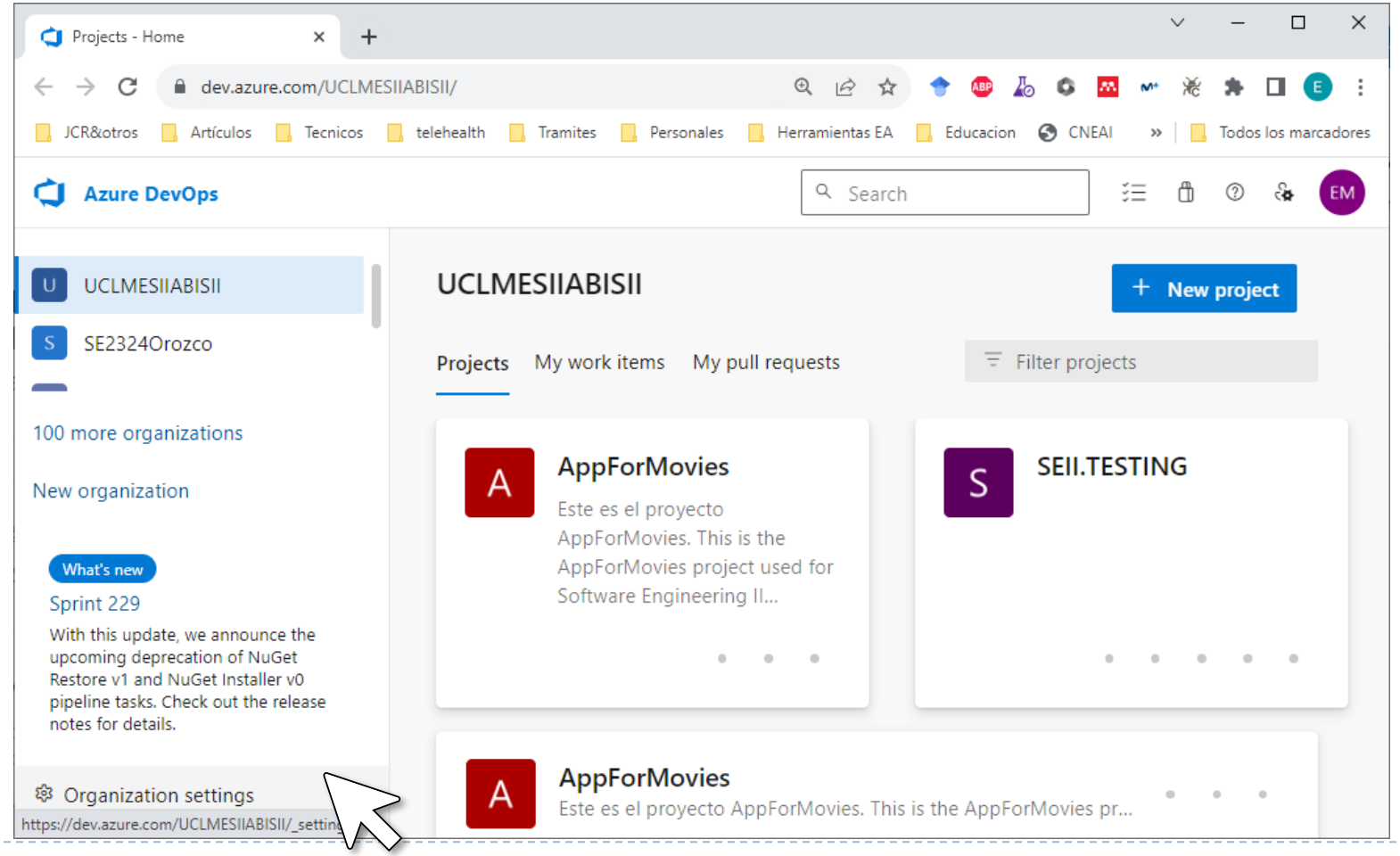
Continuous Integration with Azure Pipeline

CI and DC in Azure



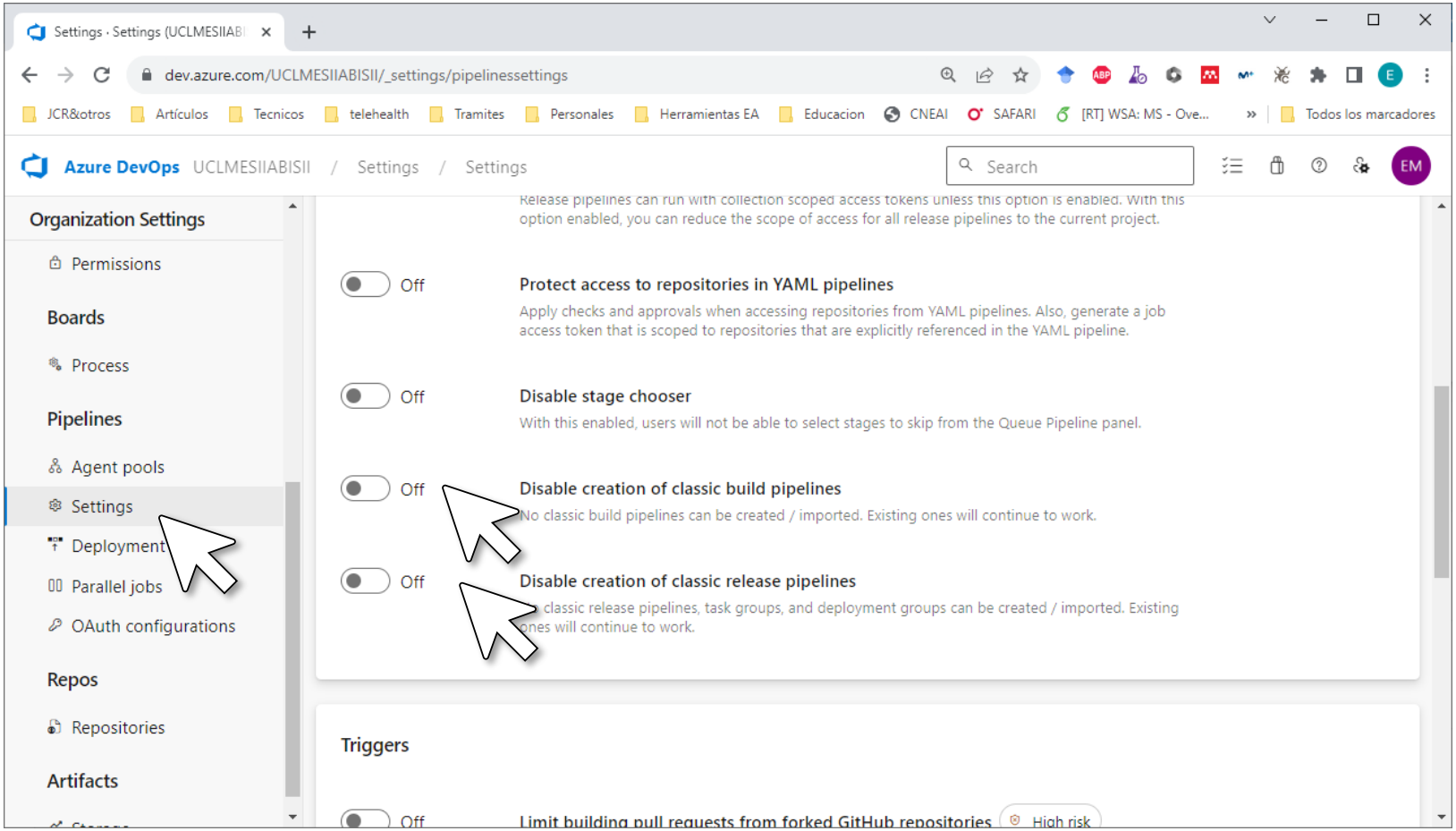
Continuous Integration: Azure Pipelines

- Open Azure DevOps:
  - Open your Organization Settings



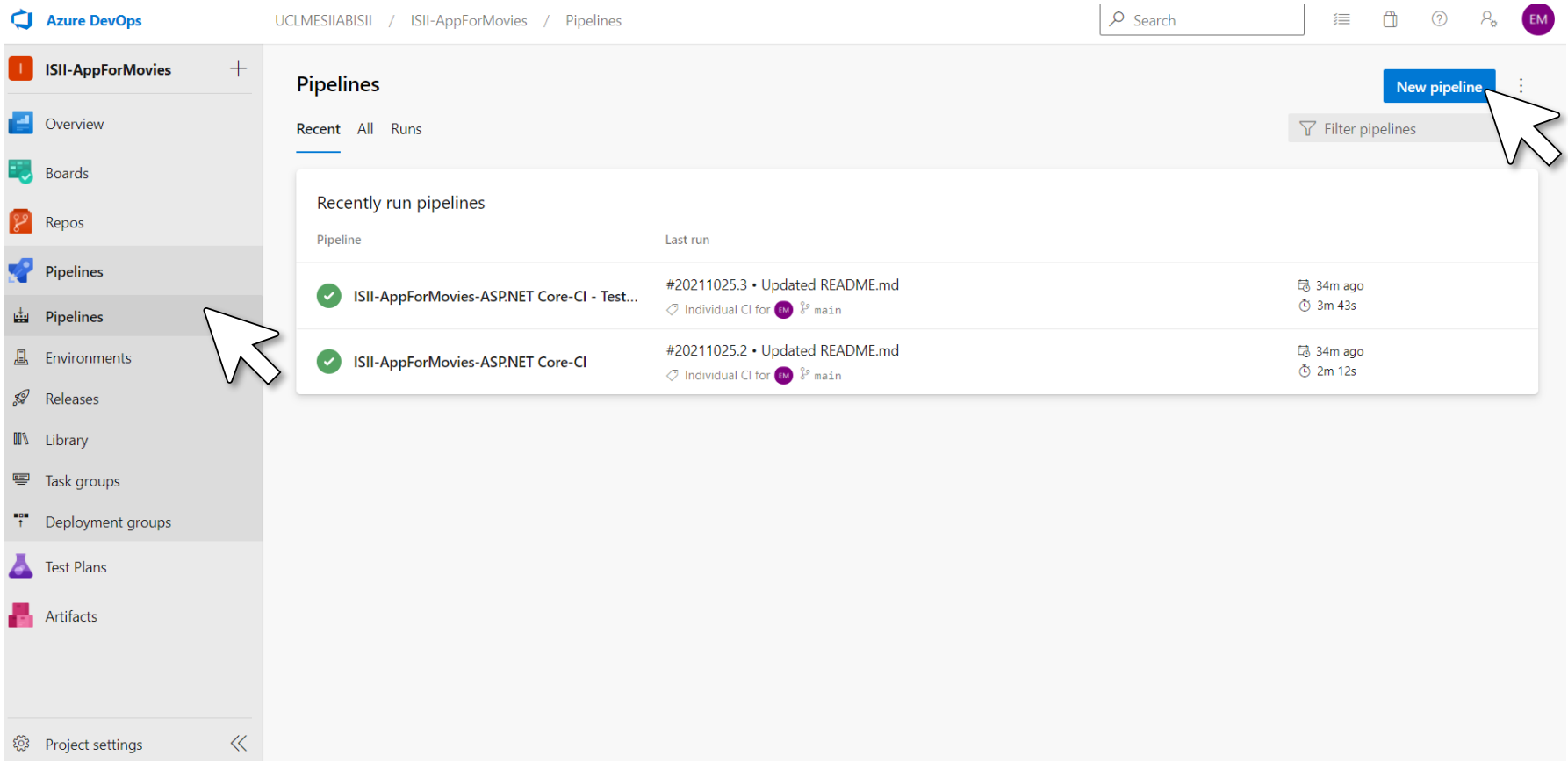
Continuous Integration: Azure Pipelines

- Open Azure DevOps:
  - **Enable** classic pipelines



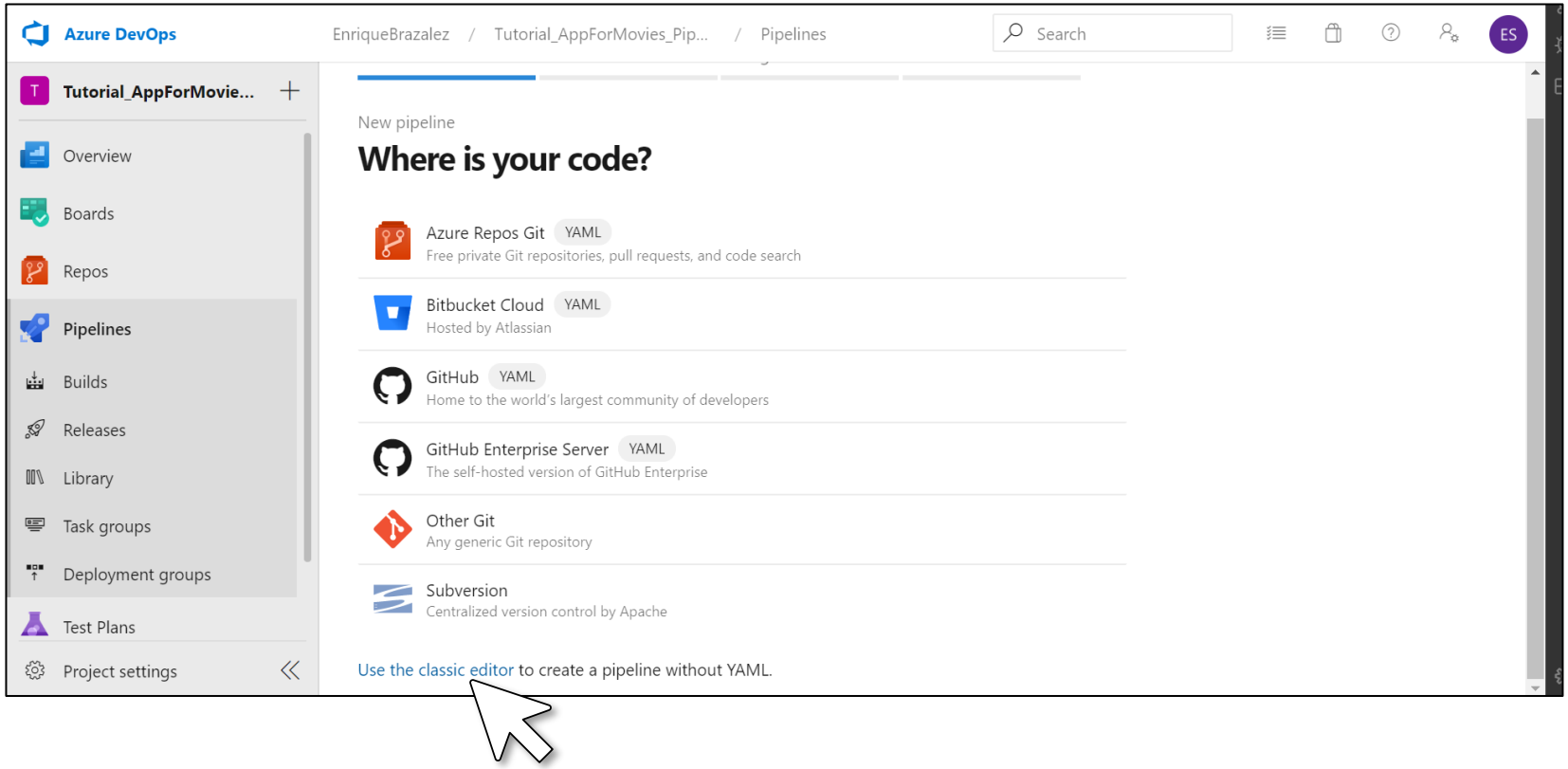
Continuous Integration: Azure Pipelines

- Open Azure DevOps:
  - Pipelines
    - Builds → New Pipeline

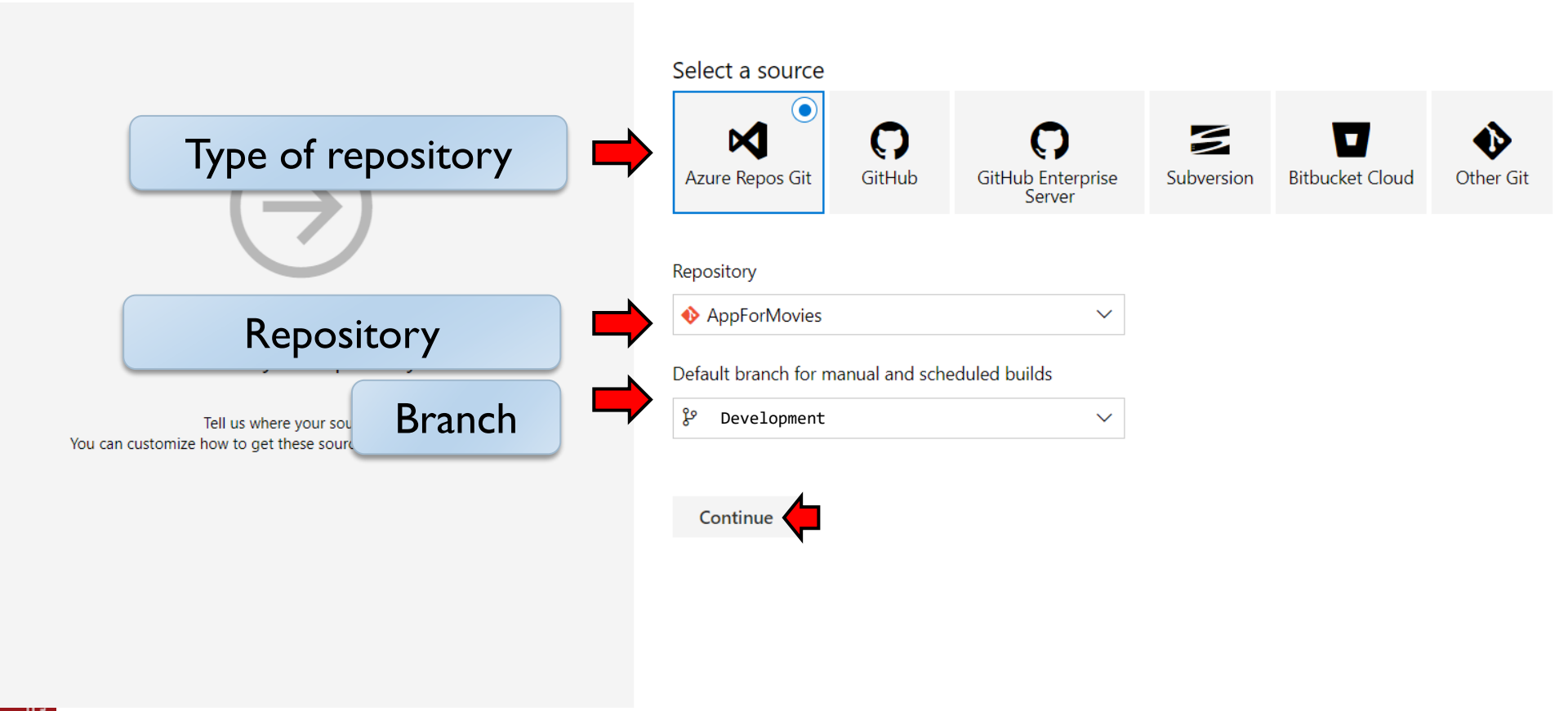


Continuous Integration: Azure Pipelines

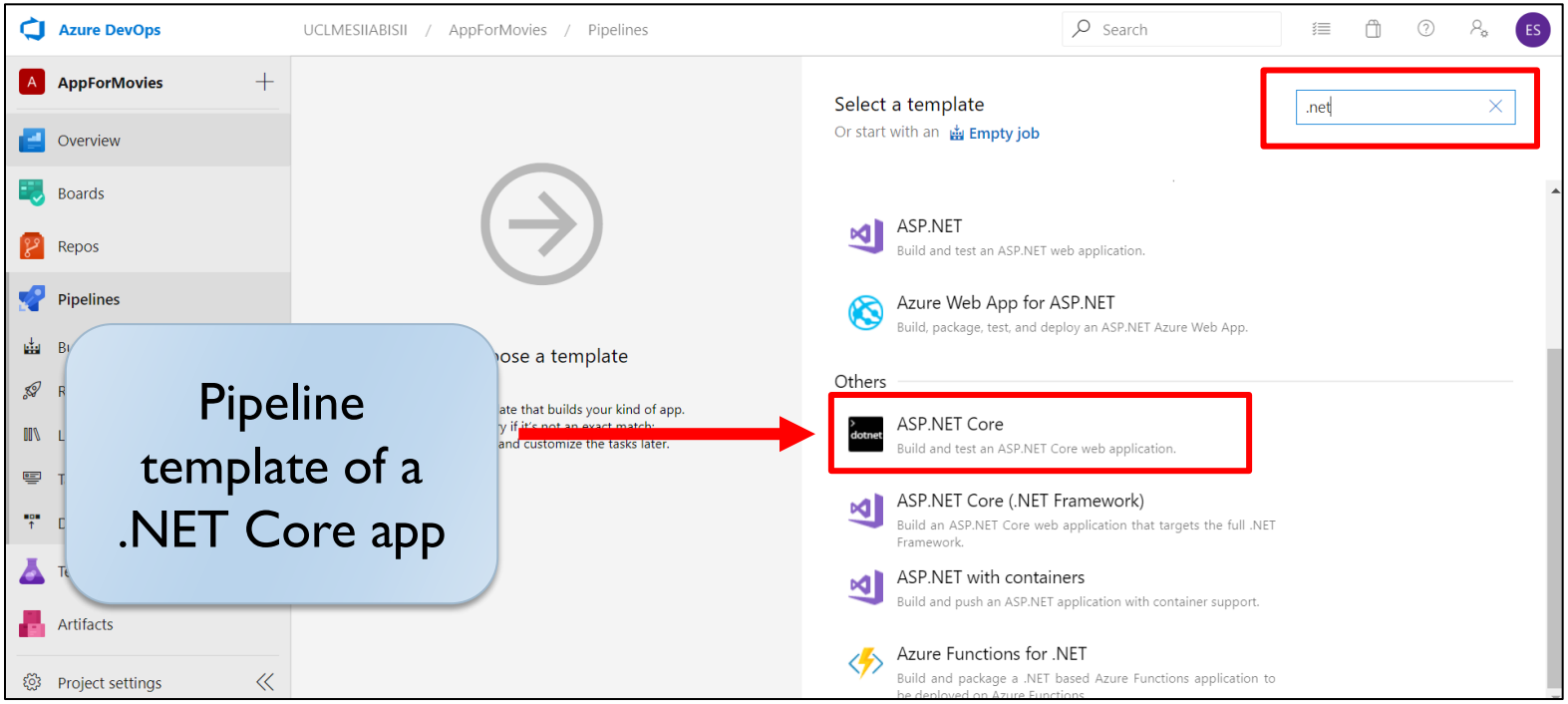
- Create the pipeline with the classic editor (easier for us)



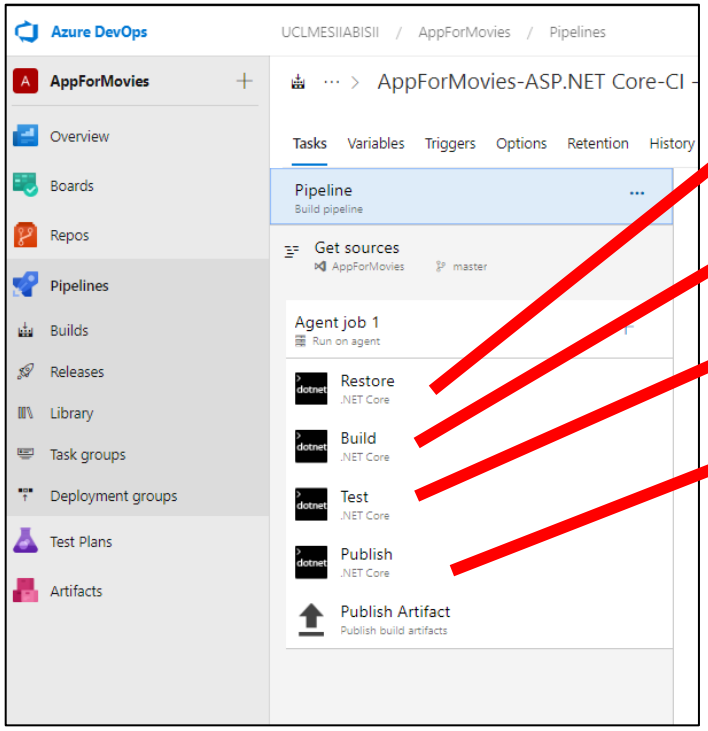
Continuous Integration: Azure Pipelines



Continuous Integration: Azure Pipelines



Continuous Integration: Azure Pipelines



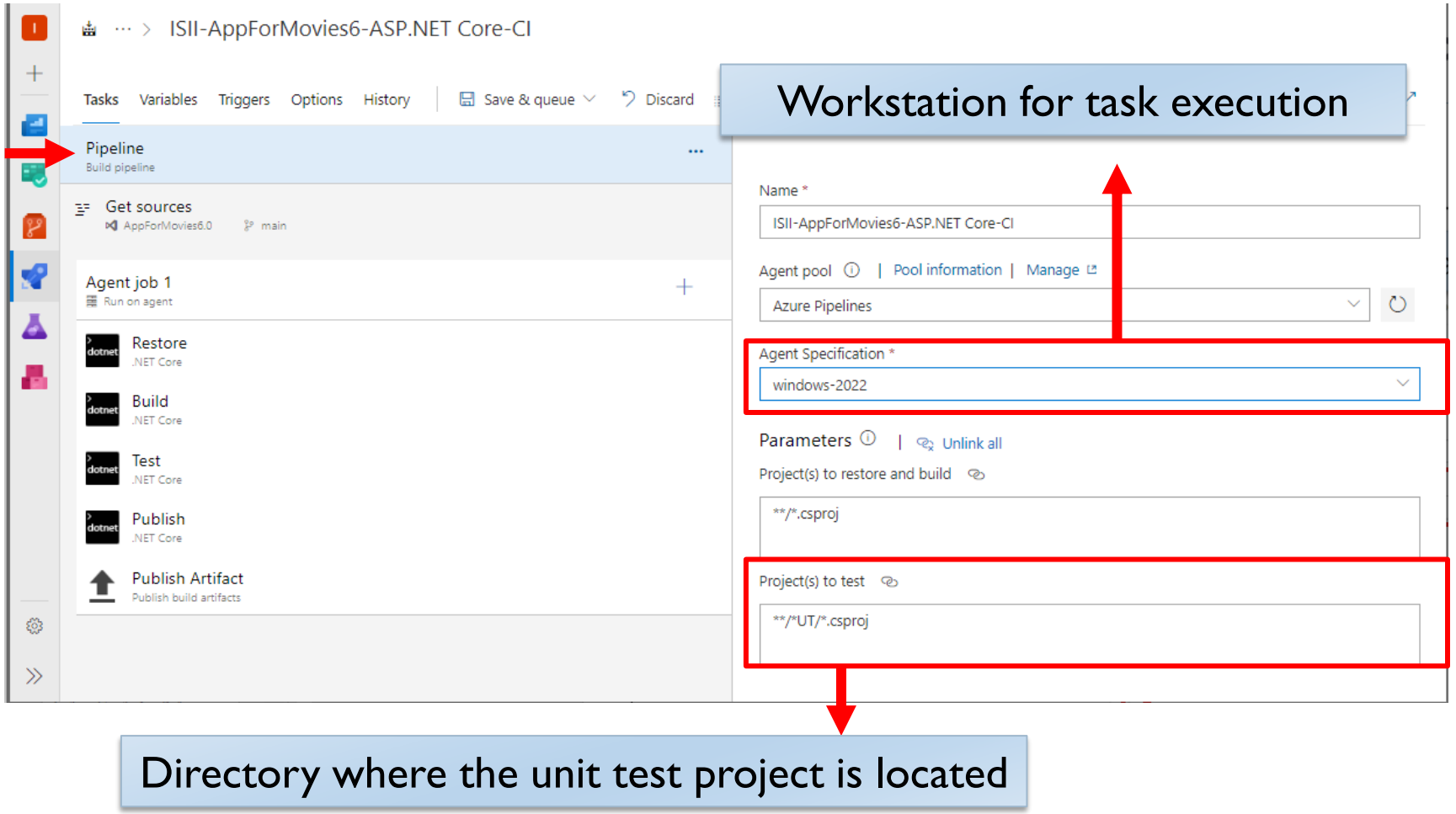
Task 1. Clean the project

Task 2. Compilation and generate build

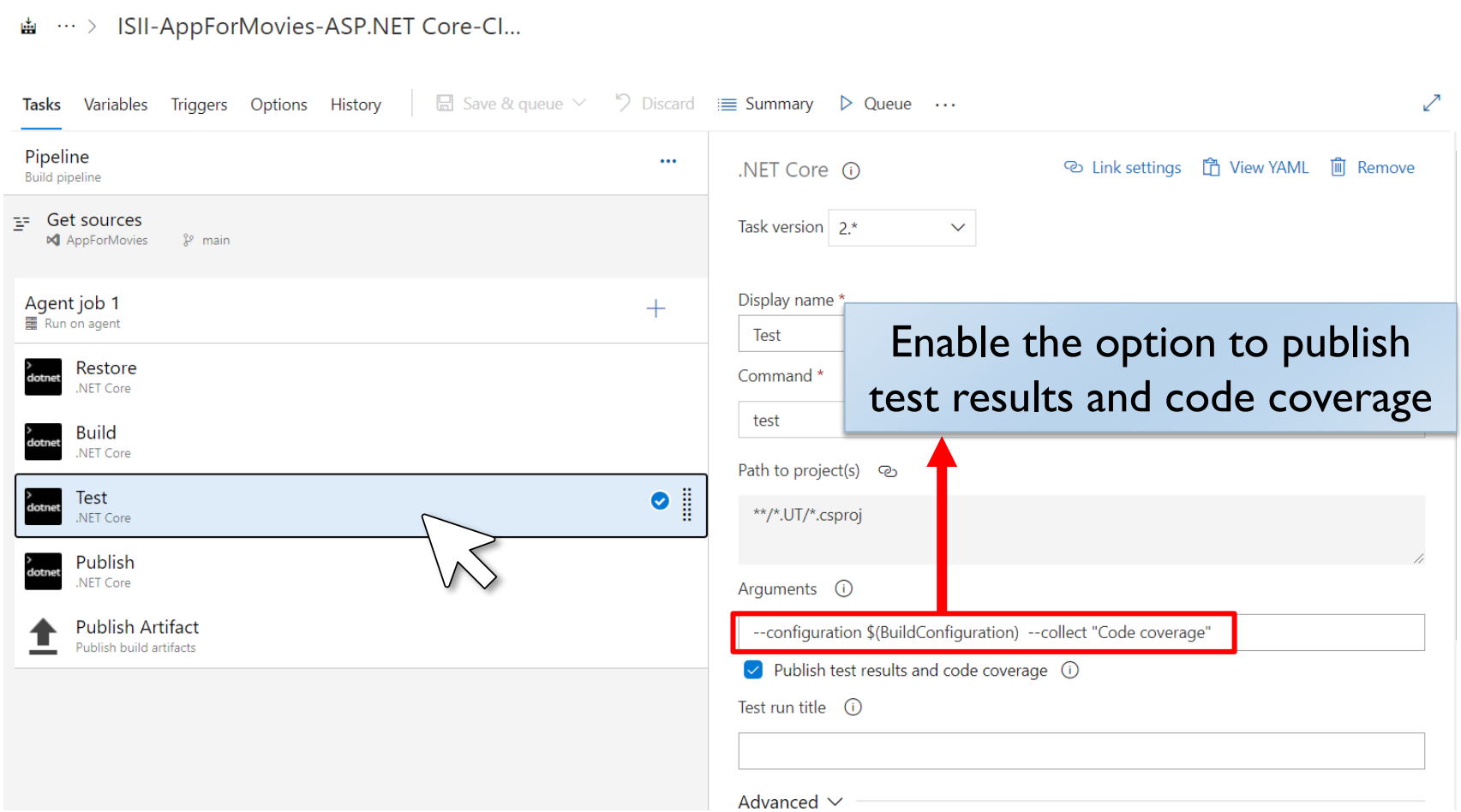
Task 3. Test

Task 4. Publish Artefact

Continuous Integration: Azure Pipelines

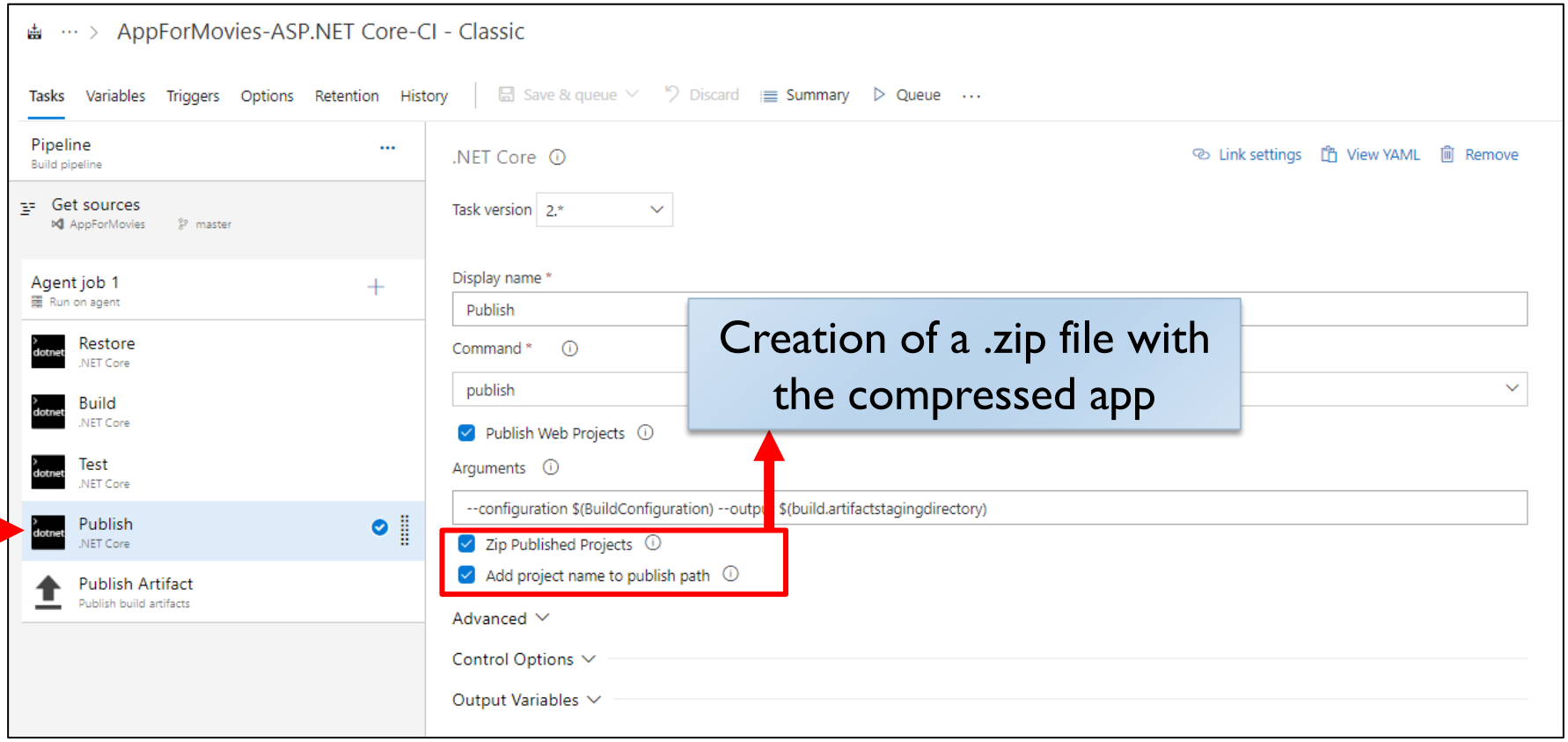


Continuous Integration: Azure Pipelines

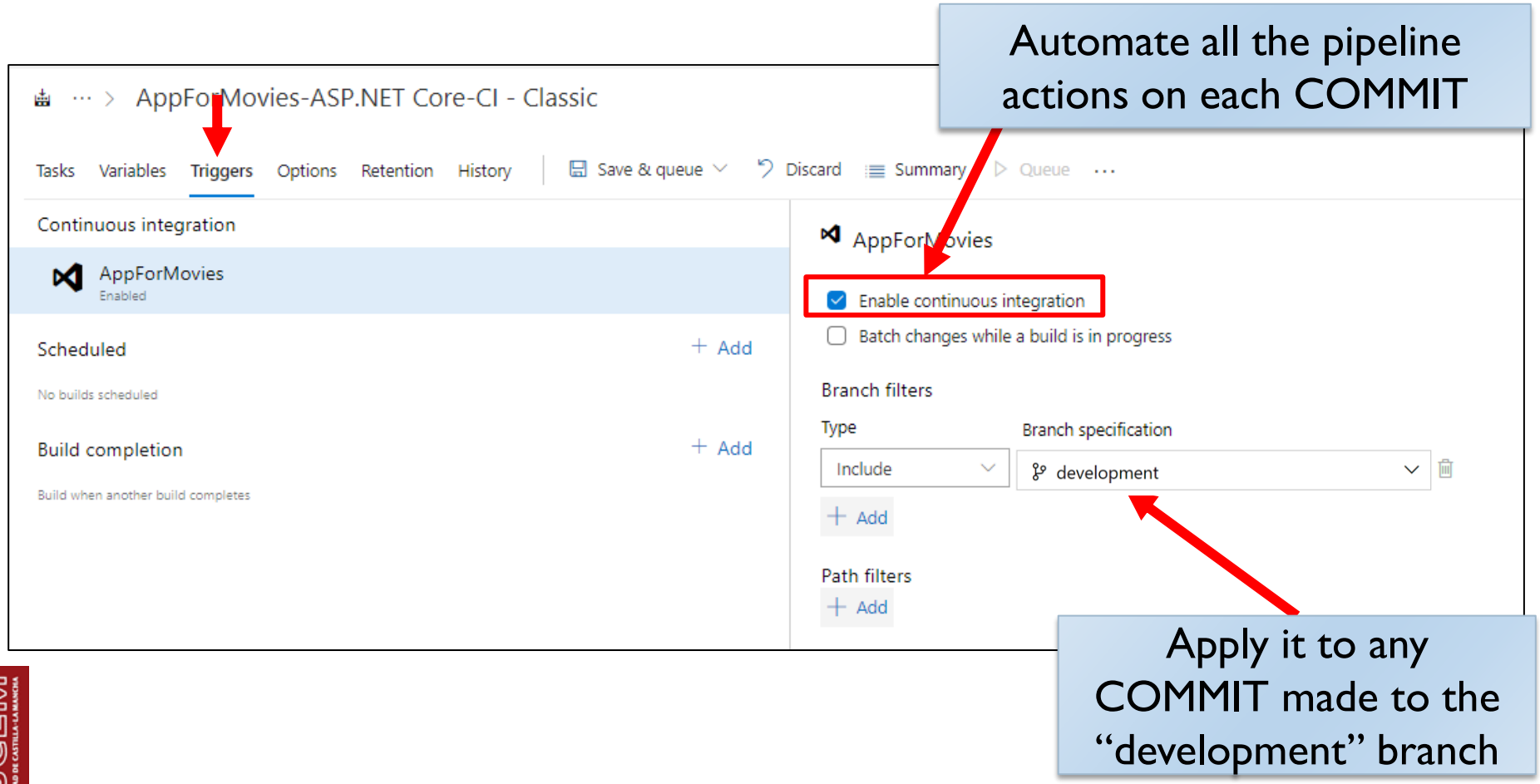


Continuous Integration: Azure Pipelines

- In the case of implementing Continuous Delivery (CD), an artefact must be published in the Releases section to automate the Delivery of our code in production.

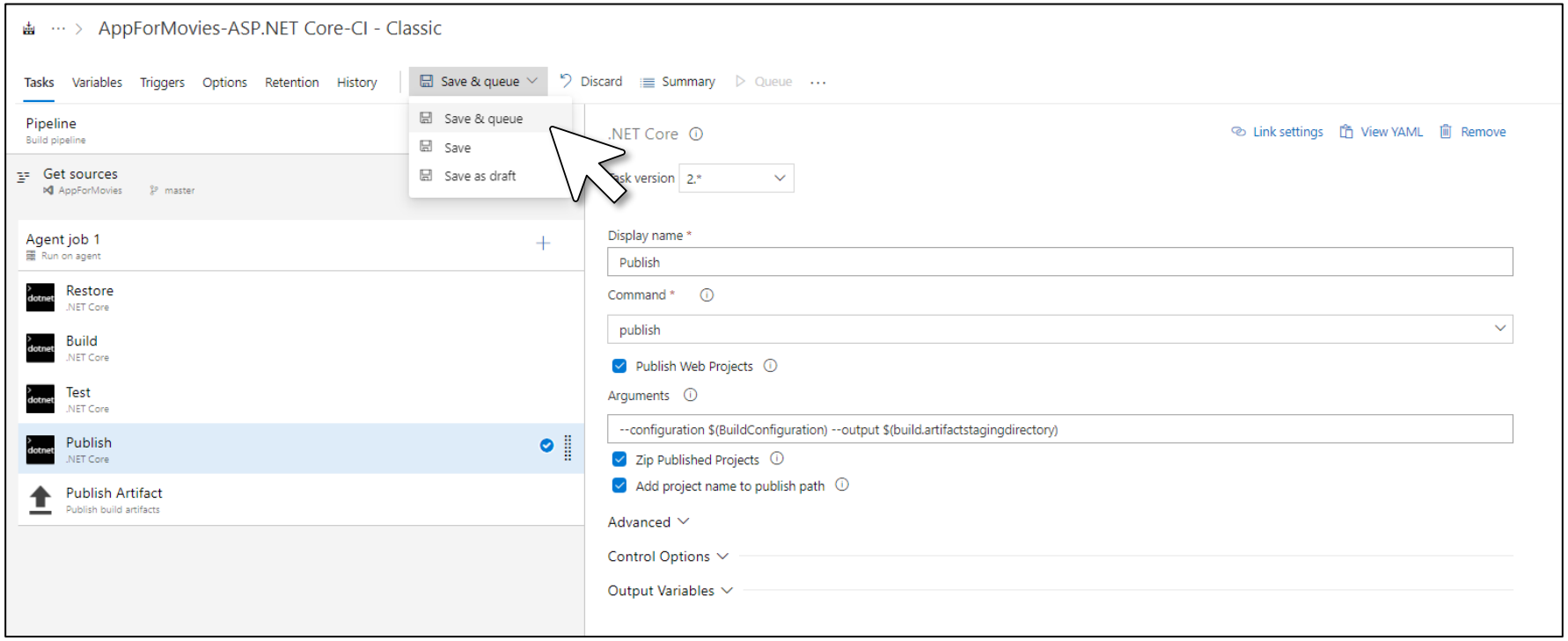


Continuous Integration: Azure Pipelines



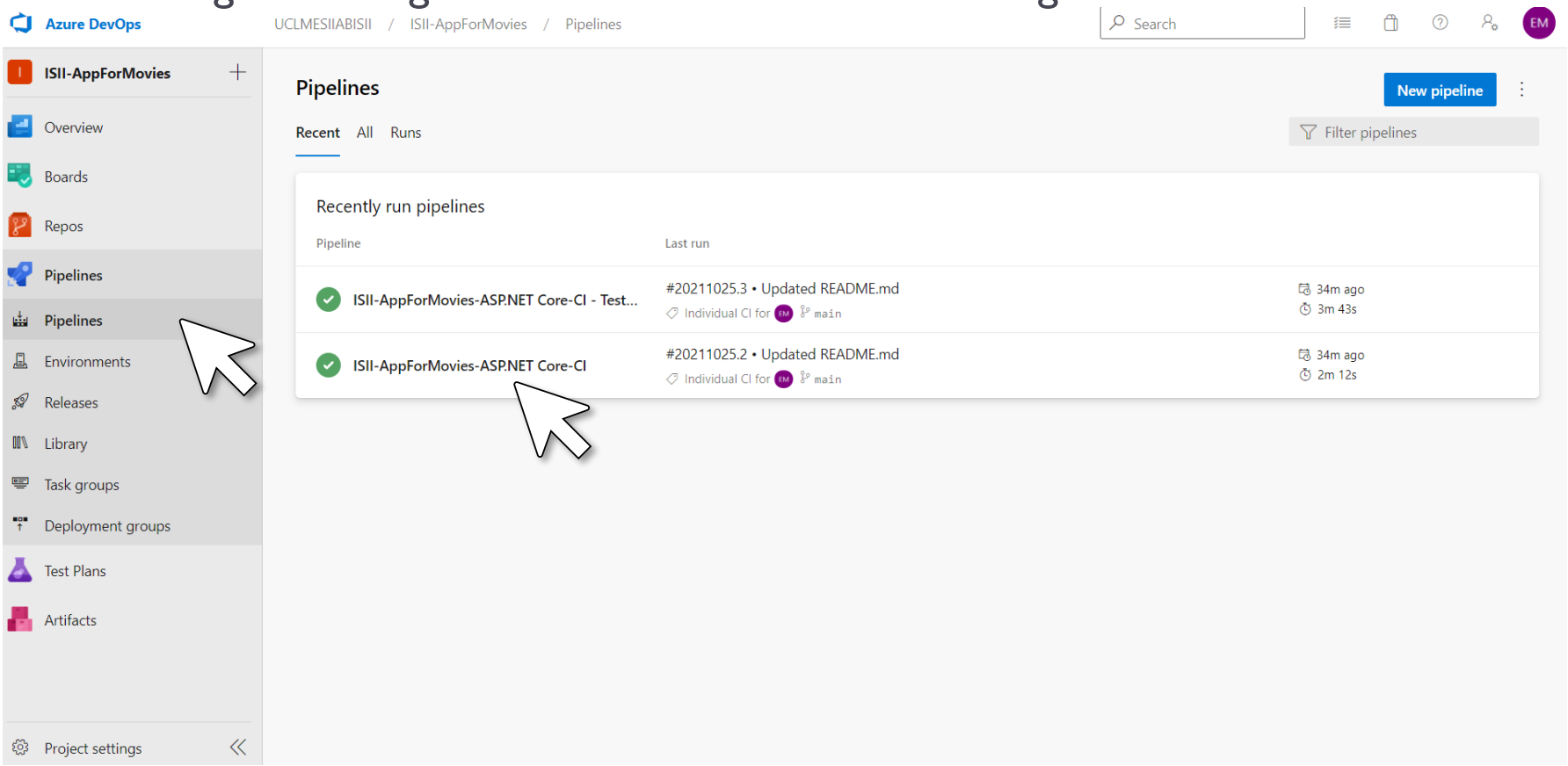
Continuous Integration: Azure Pipelines

- Now we have our CI pipeline already set up
  - Save and analyze your results



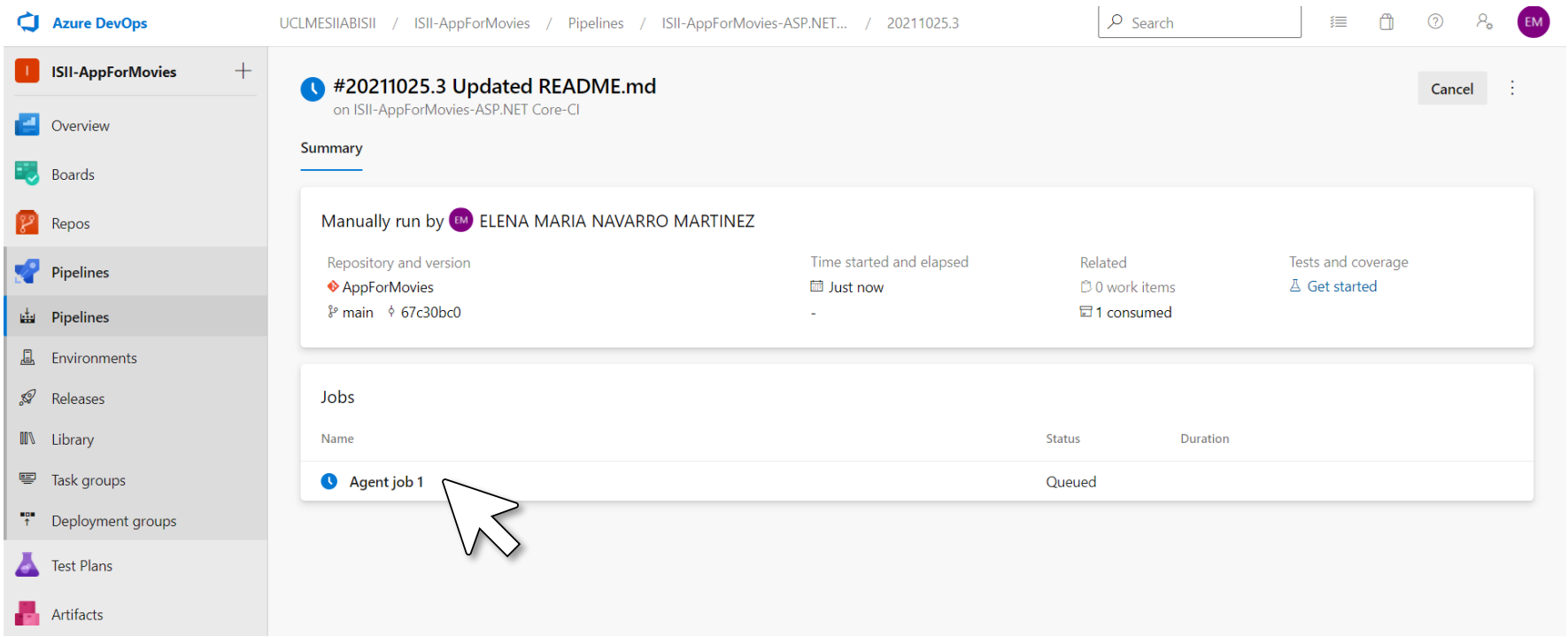
Continuous Integration: Azure Pipelines

- Pipeline execution
  - Once the Pipeline is created in the Azure DevOps portal, it will automatically launch an agent that will perform the tasks we have configured using the lastest COMMIT on the target branch.



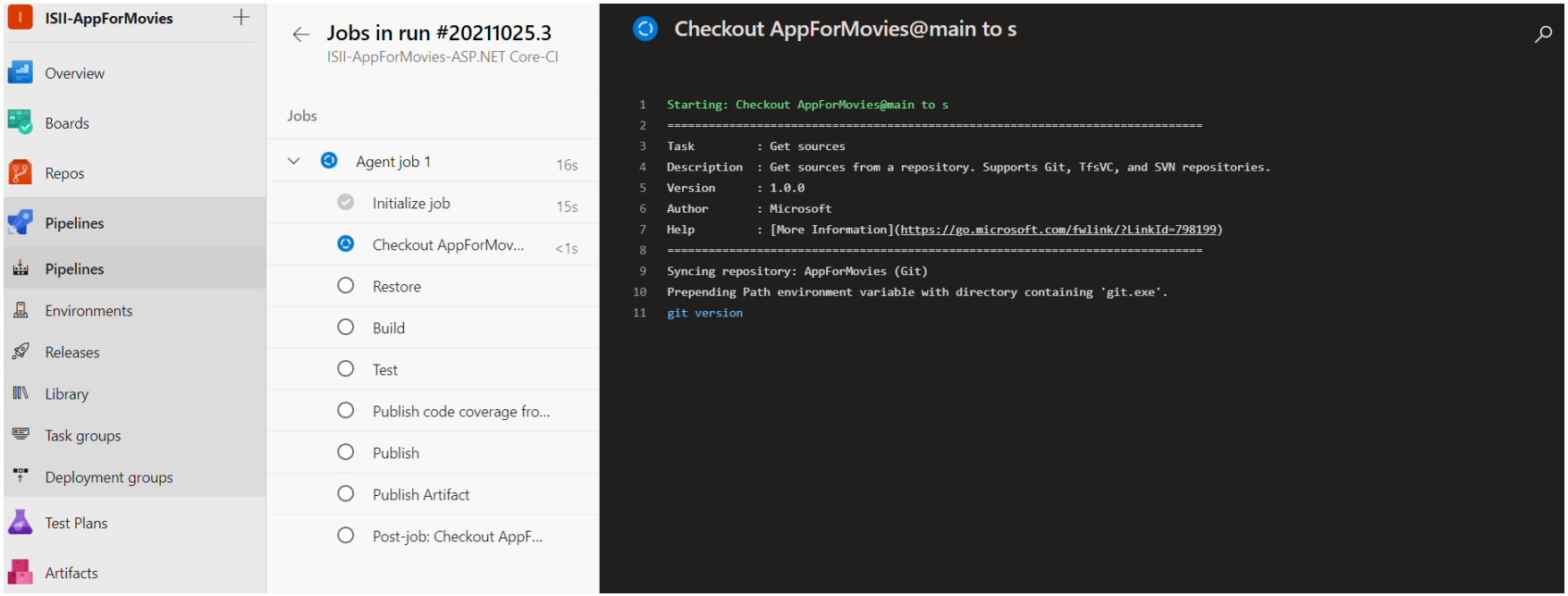
Continuous Integration: Azure Pipelines

- Log
  - See how each action executes

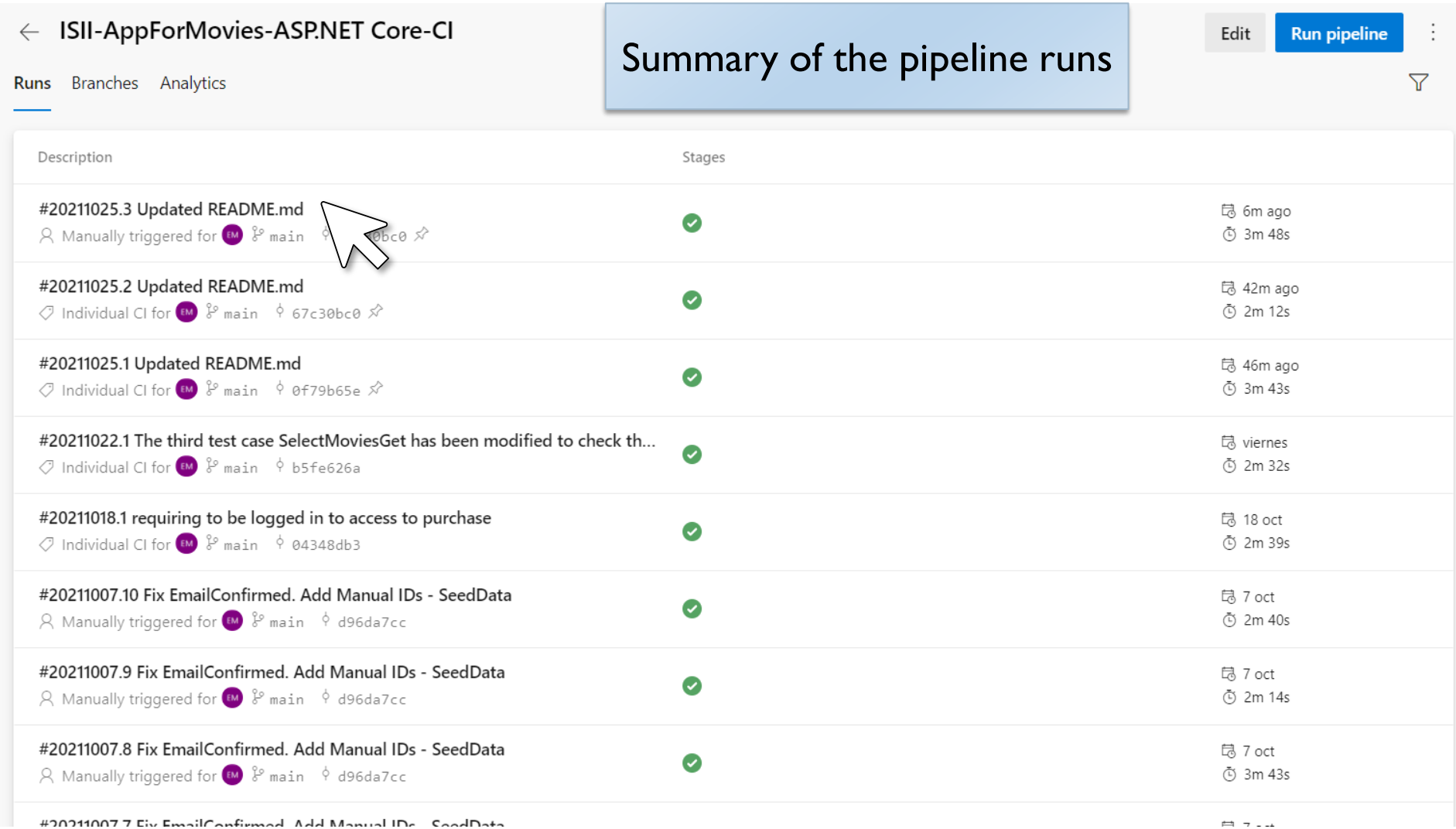


Continuous Integration: Azure Pipelines

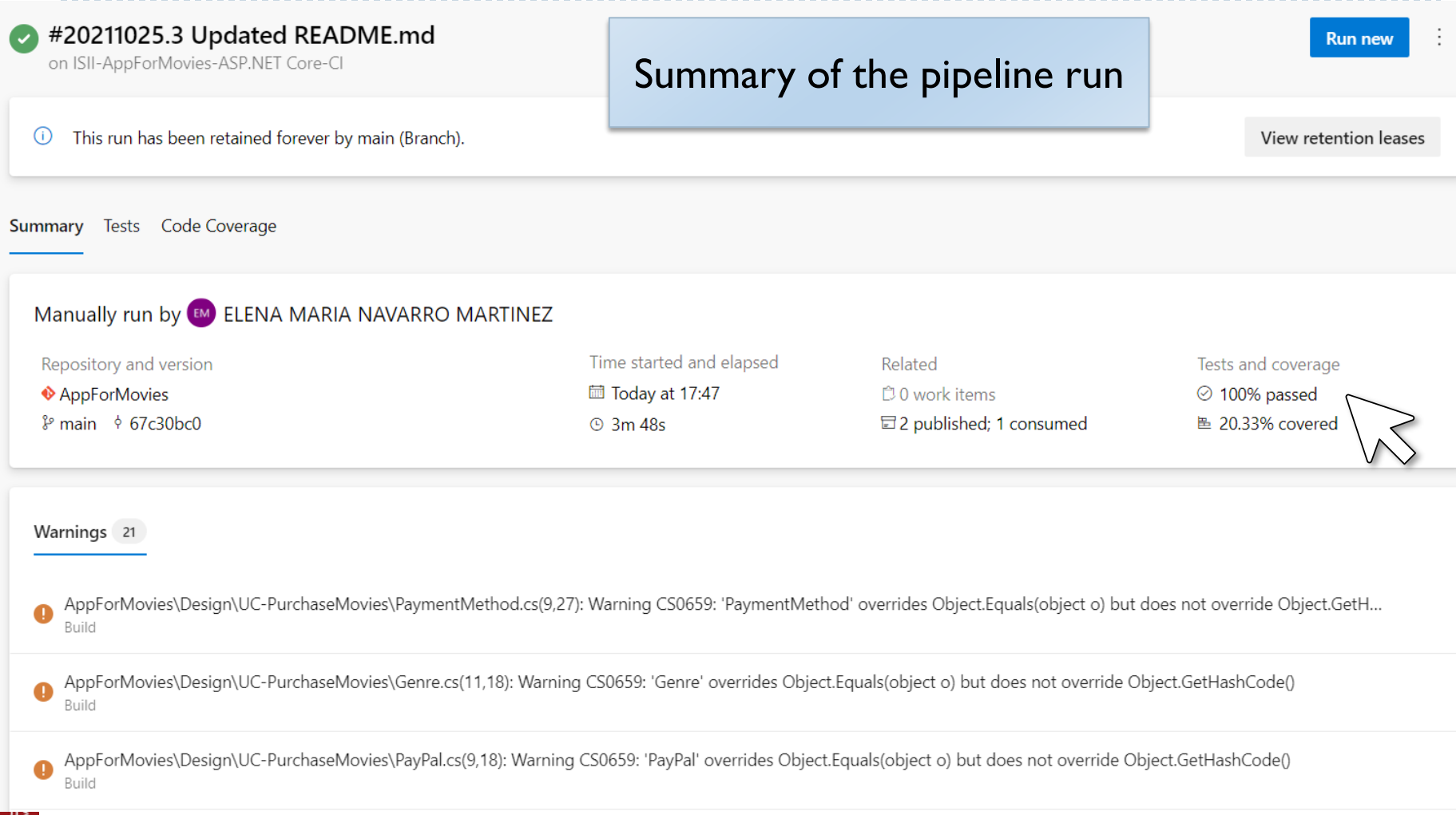
- Log
  - See how each action executes



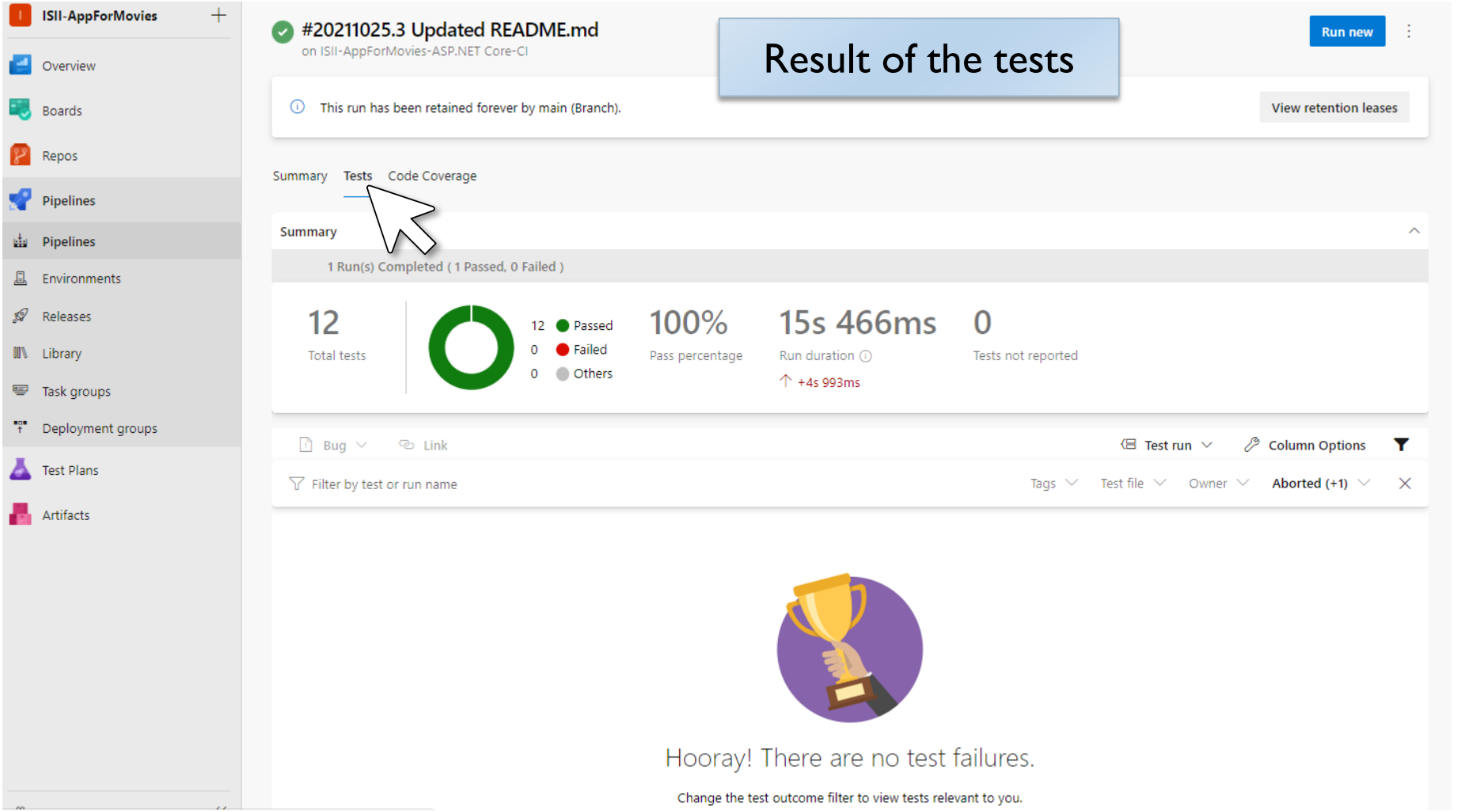
Continuous Integration: Azure Pipelines



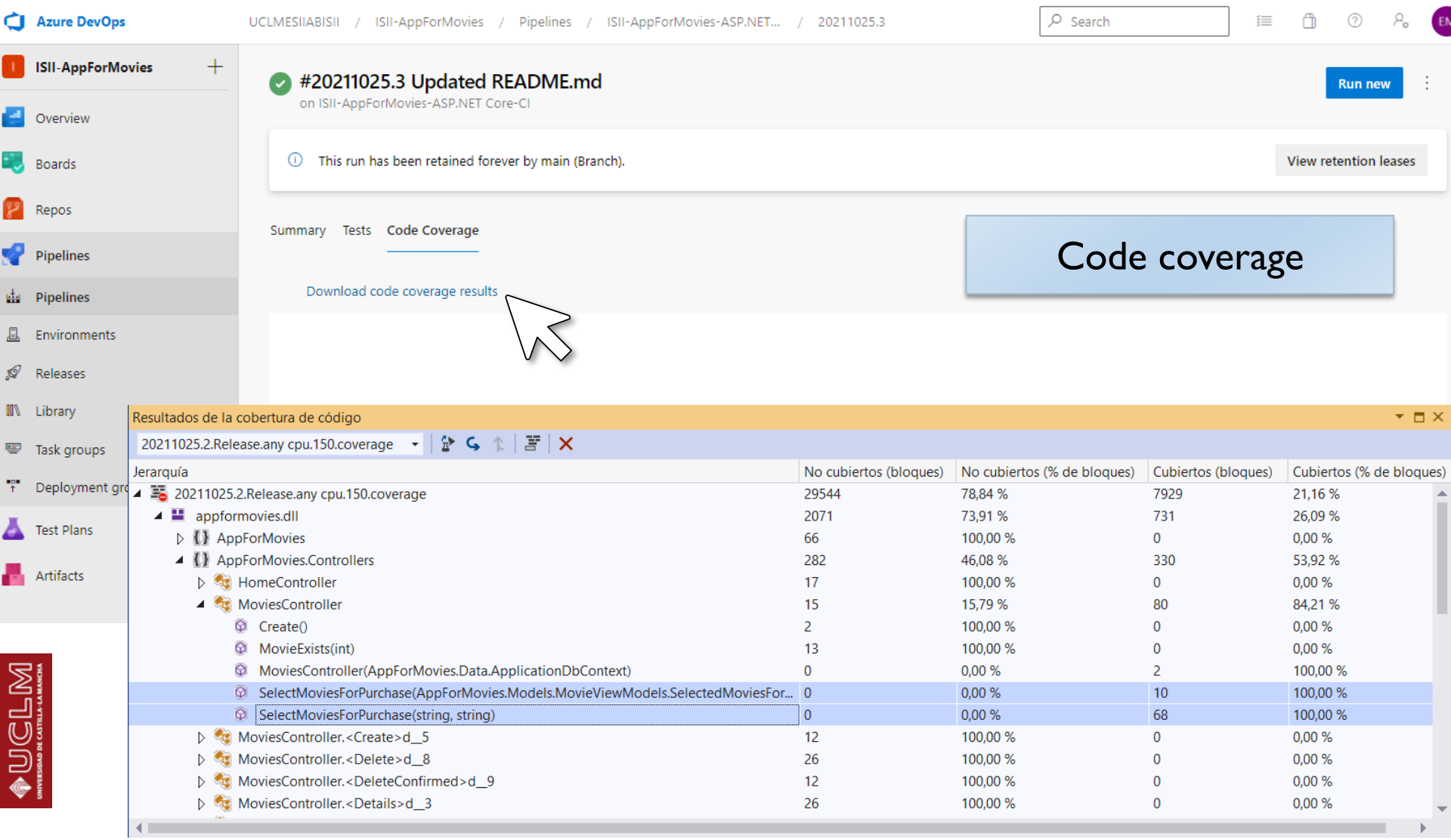
Continuous Integration: Azure Pipelines



Continuous Integration: Azure Pipelines

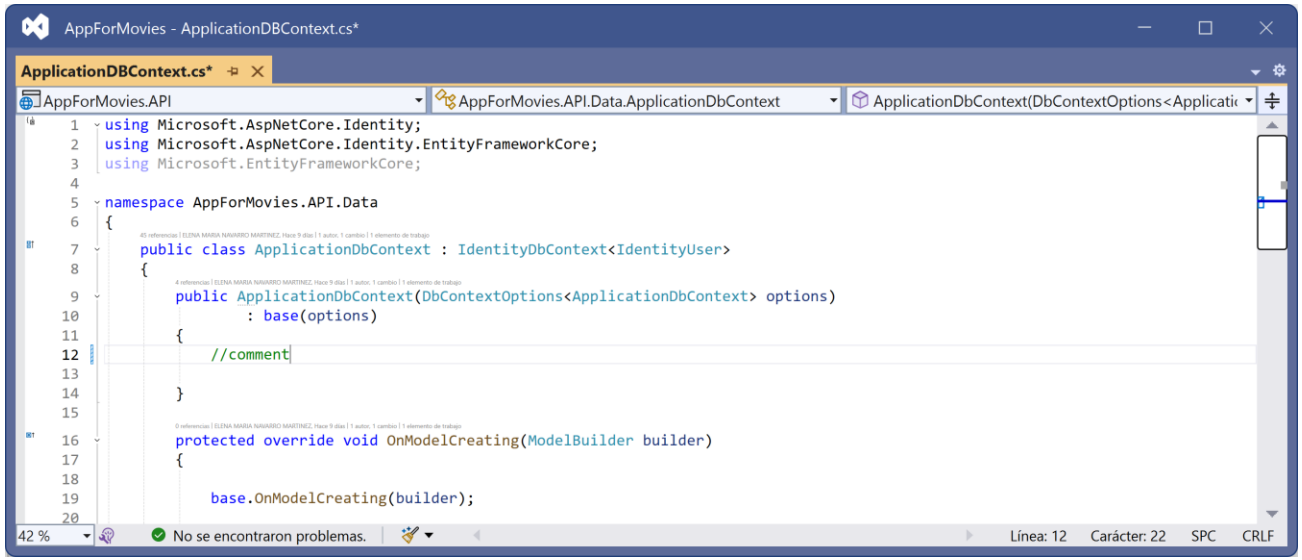


Continuous Integration: Azure Pipelines



Continuous Integration: Azure Pipelines

- How CI works
  1. Introduce a change in your Project, for instance a comment
  2. Commit and Sync the solution and pull request the changes in the “development" branch
  3. Observe the actions performed by the pipeline created



More information

- Links sobre Pipelines
  - Get started with Azure Pipelines
    - https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/?view=azure-devops
  - Create your First Pipeline
    - https://docs.microsoft.com/en-us/azure/devops/pipelines/create-first-pipeline?view=azure-devops&tabs=browser%2Ctfs-2018-2
  - Enabling Continuous Integration with Azure Pipelines
    - https://azuredevopslabs.com/labs/azuredevops/continuousintegration/

POST - PURCHASE - TEST

1- Crear constructor con los datos de la db temporal.

2- Crea los distintos test cases, un enumerable

-> No existe la tabla

-> No existe el user -> No registrado

-> Cantidad = 0

-> No existe el device en la tabla

3 - Test Task CreatePurchase - Test
- arrange -> mock
- act
- assert

Continuous delivery -> Human

Continuous deployment -> Automatic

Continuous integration -> Pipeline

Unit 5. Software Testing: Processes and Documentation

1. Introduction
2. Test Processes
3. Test Management Processes
4. Dynamic Test Processes
5. Integration Testing
6. System Testing
7. Acceptance Testing
8. Regression Testing

Goals

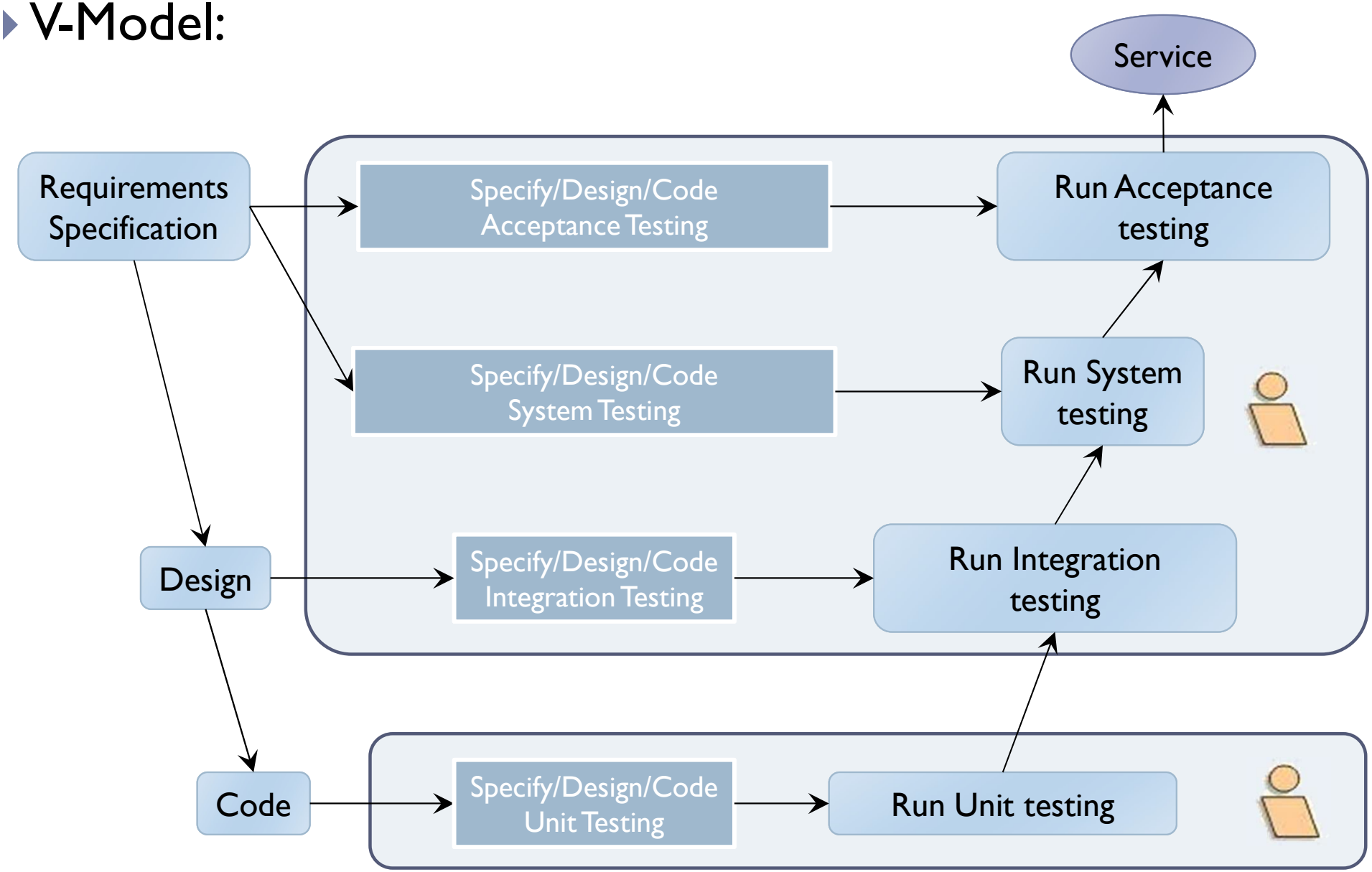
- To be able to identify which processes may be relevant for a testing process
- To understand which dependencies between processes are applicable
- To understand why we should document our testing process

1. Introduction

- Testing is a key approach to risk mitigation in software development.
- **Risk-based testing** is a best-practice approach to strategizing and managing testing, as it allows testing to be prioritized and focused on **the most important features and quality attributes.**
- **Test processes** that can be used to govern, manage and implement software testing for any organization, project or smaller testing activity.
- **Test documentation** is an output of the processes

1. Introduction

- V-Model:

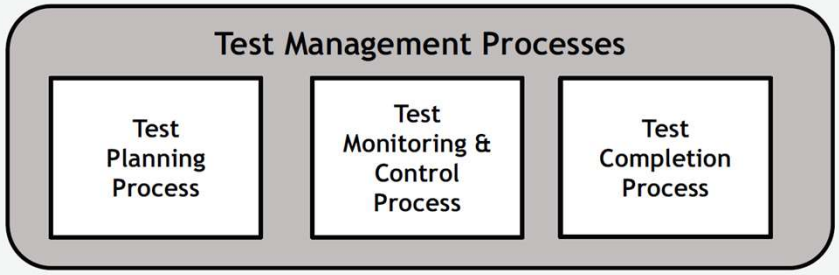


2. Test Processes

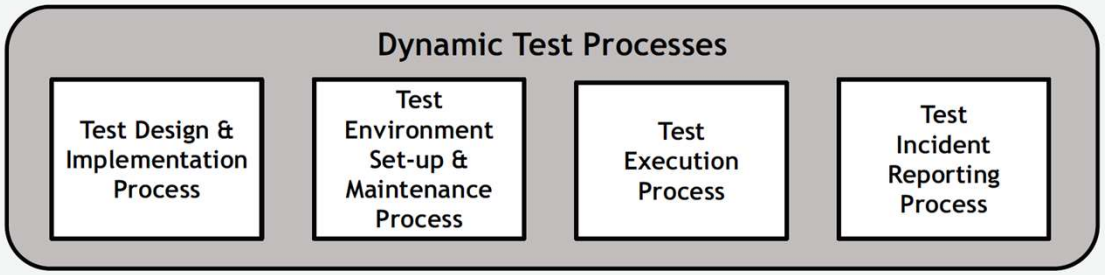
- Used to provide information on the quality of a software product comprising a number of activities, grouped into one or more test sub-processes (ISO/IEC/IEEE 29119-2:2021)

Document everything ↓ All result of testing

develops and manages organizational test specifications that typically apply to testing across the **whole organization** (i.e. they are not project-based).



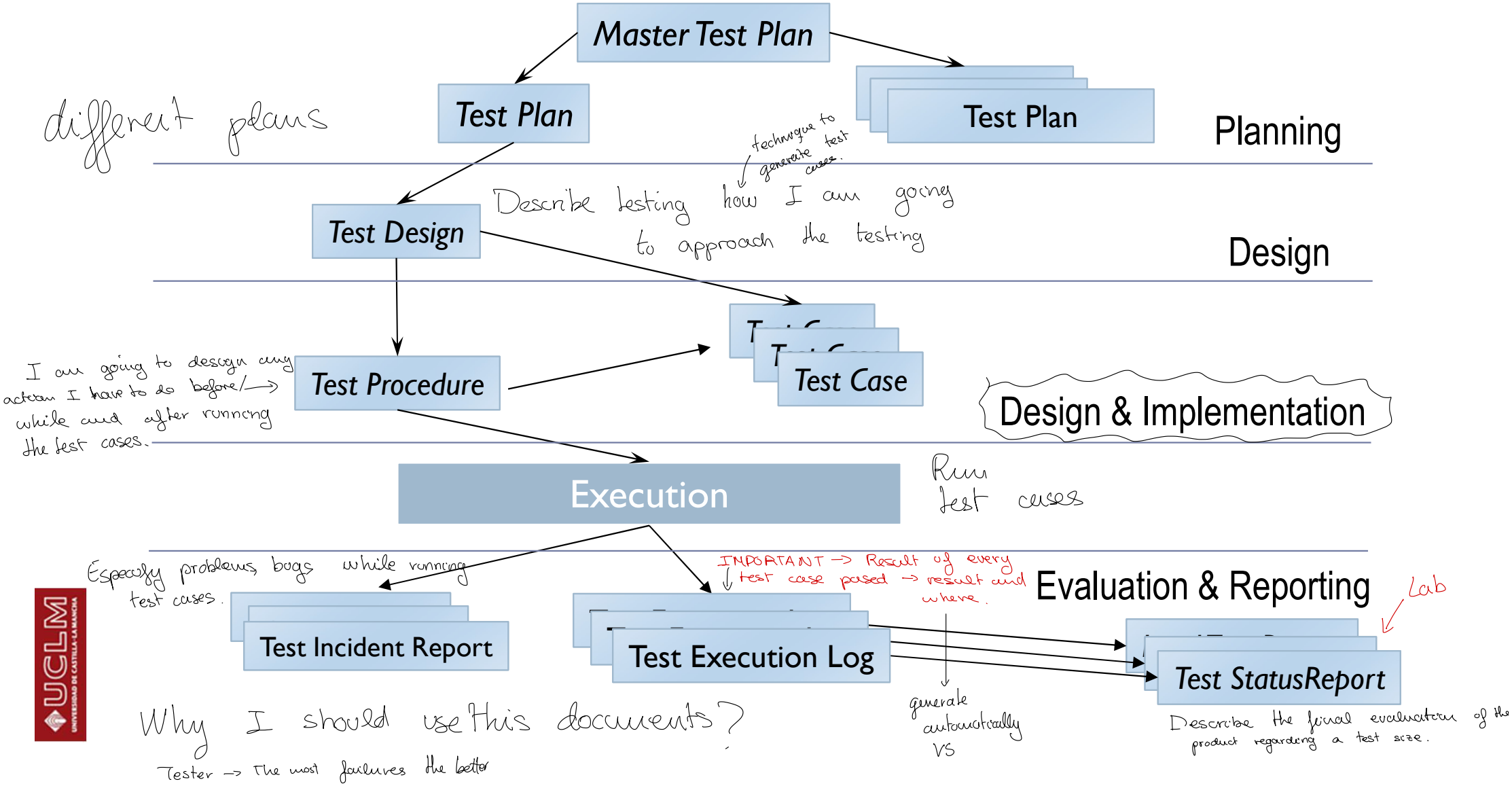
cover the management of testing for a whole test project or **any test phase** (e.g. system testing) **or test type** (e.g. performance testing) within a test project (e.g. project test management, system test management, performance test management).



for performing dynamic testing that may be performed at a **particular phase of testing** (e.g. unit, integration, system, and acceptance) or for a **particular type of testing** (e.g. performance testing, security testing, and functional testing) within a test project.

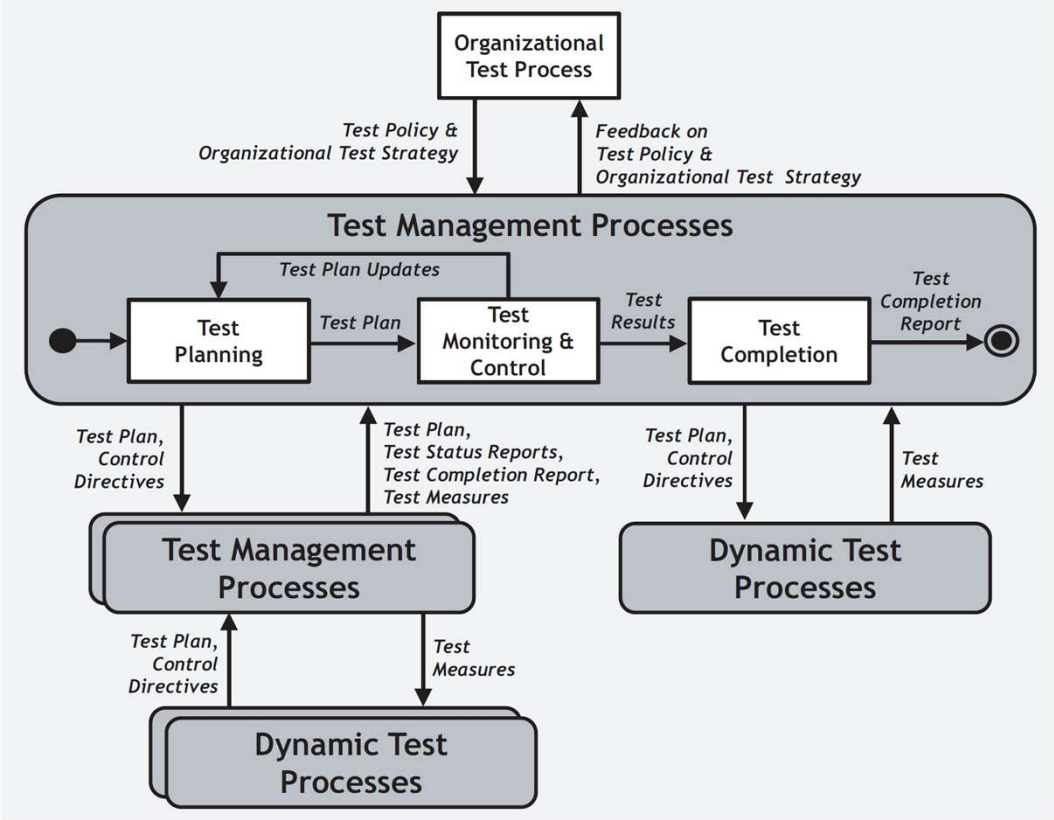
2. Test Processes: Types of documents

- Documents ISO/IEC/IEEE 29119-2:2021 for Software Test:



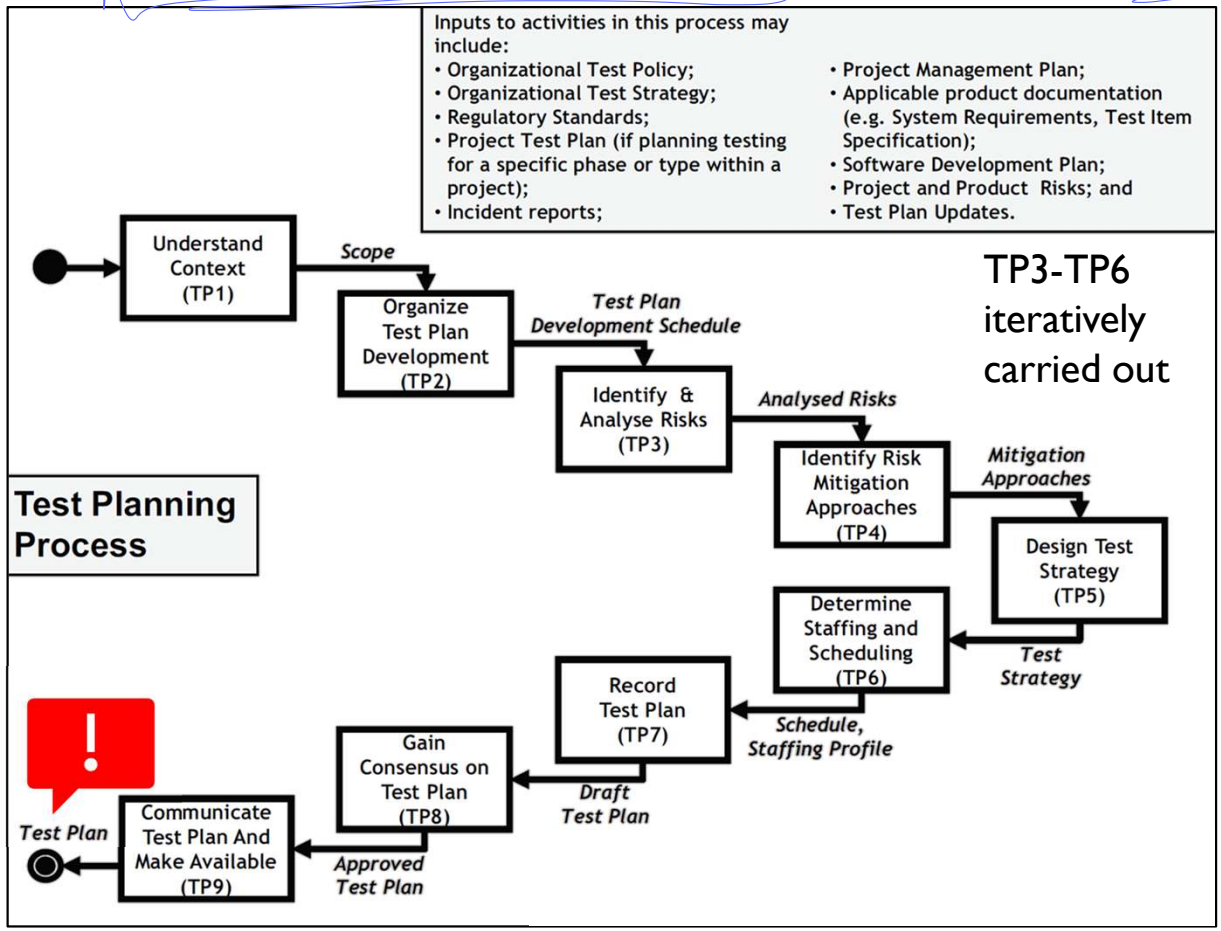
3. Test Management Processes

- These may be applied **at the project level** (project test management based on a project test plan), for test management **at different test phases** (e.g. system test management, acceptance test management based on separate test plans) and for managing **various test types** (e.g. performance test management, usability test management based on separate test plans).



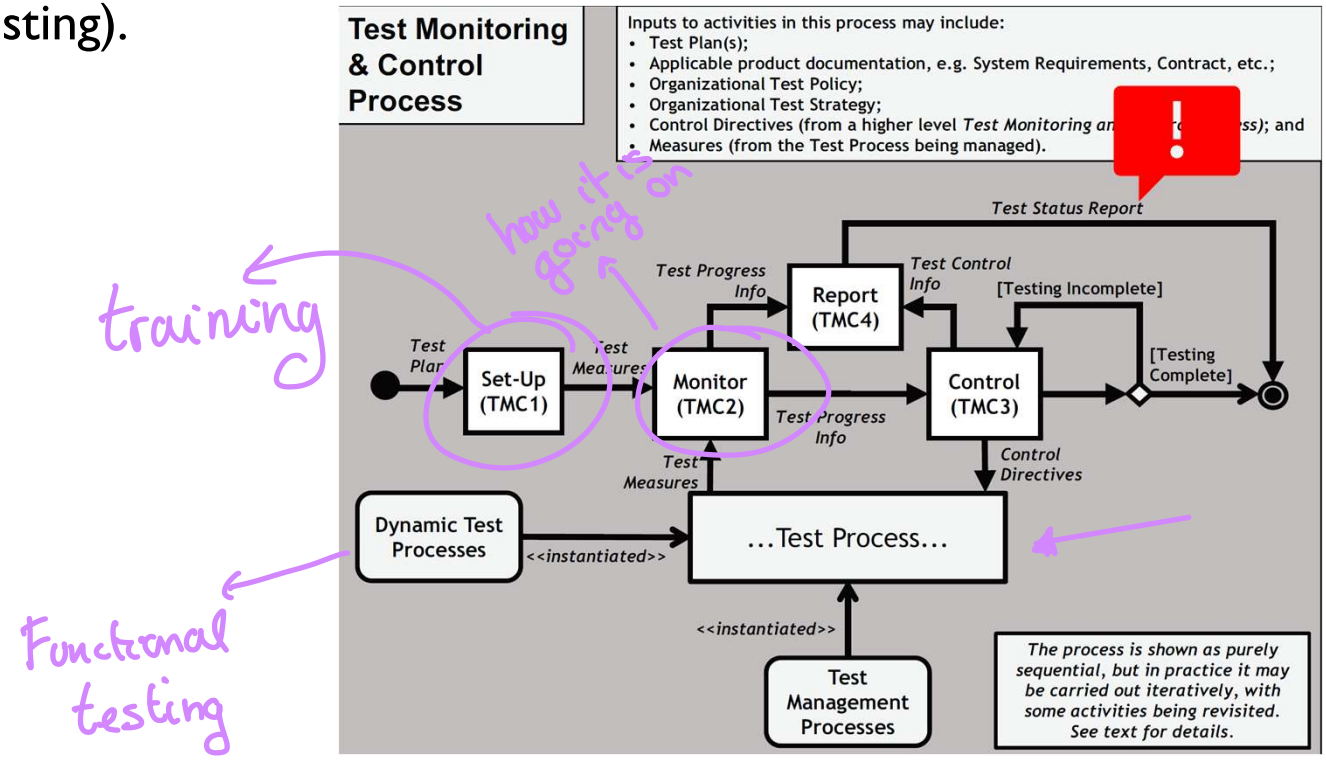
3. Test Management Processes

- The **Test Planning Process** is used to develop **the Test Plan**. Depending on where in the project this process is implemented this may be a **Project** Test Plan or a test plan for a **specific phase**, such as a System Test Plan, or a test plan for a **specific type of testing**, such as a Performance Test Plan.



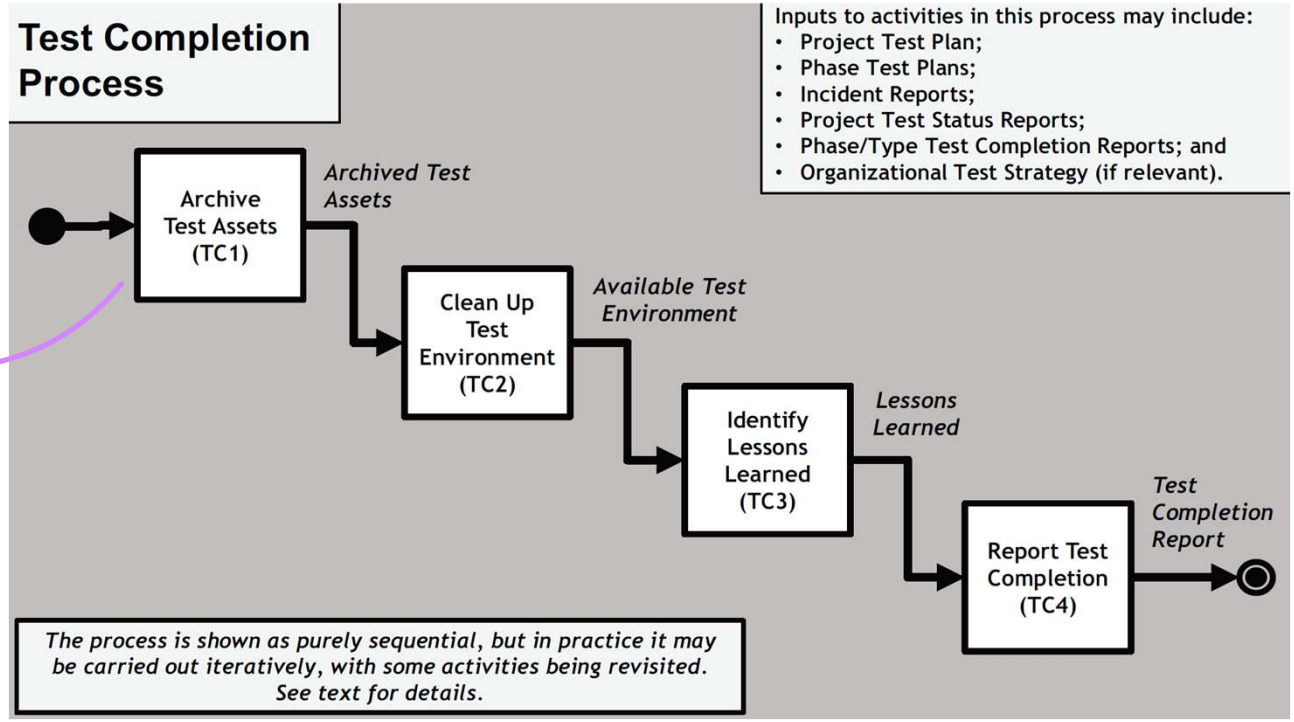
3. Test Management Processes

- **Test Monitoring & Control process** scrutinizes whether testing progresses in accordance with the Test Plan and the organizational test specifications. If there are significant departures from planned progress, activities, or other aspects of the test plan, activities will be initiated to correct or compensate for the resultant variances.
- This process can be applied to the management of a whole test project (normally made up of a number of test phases and test types) or to the management of the testing of a single test phase (e.g. system testing) or test type (e.g. performance testing).



3. Test Management Processes

- The **Test Completion Process** is performed when agreement has been obtained that the testing activities at a specific test phase (e.g. system testing) or test type (e.g. performance testing) are complete to make available useful test assets for later use, leave the test environment in a satisfactory condition, and record and communicate the results of the testing to relevant stakeholders. Test assets include Test Plans, Test Case Specifications, test scripts, test tools, test data and test environment infrastructure.



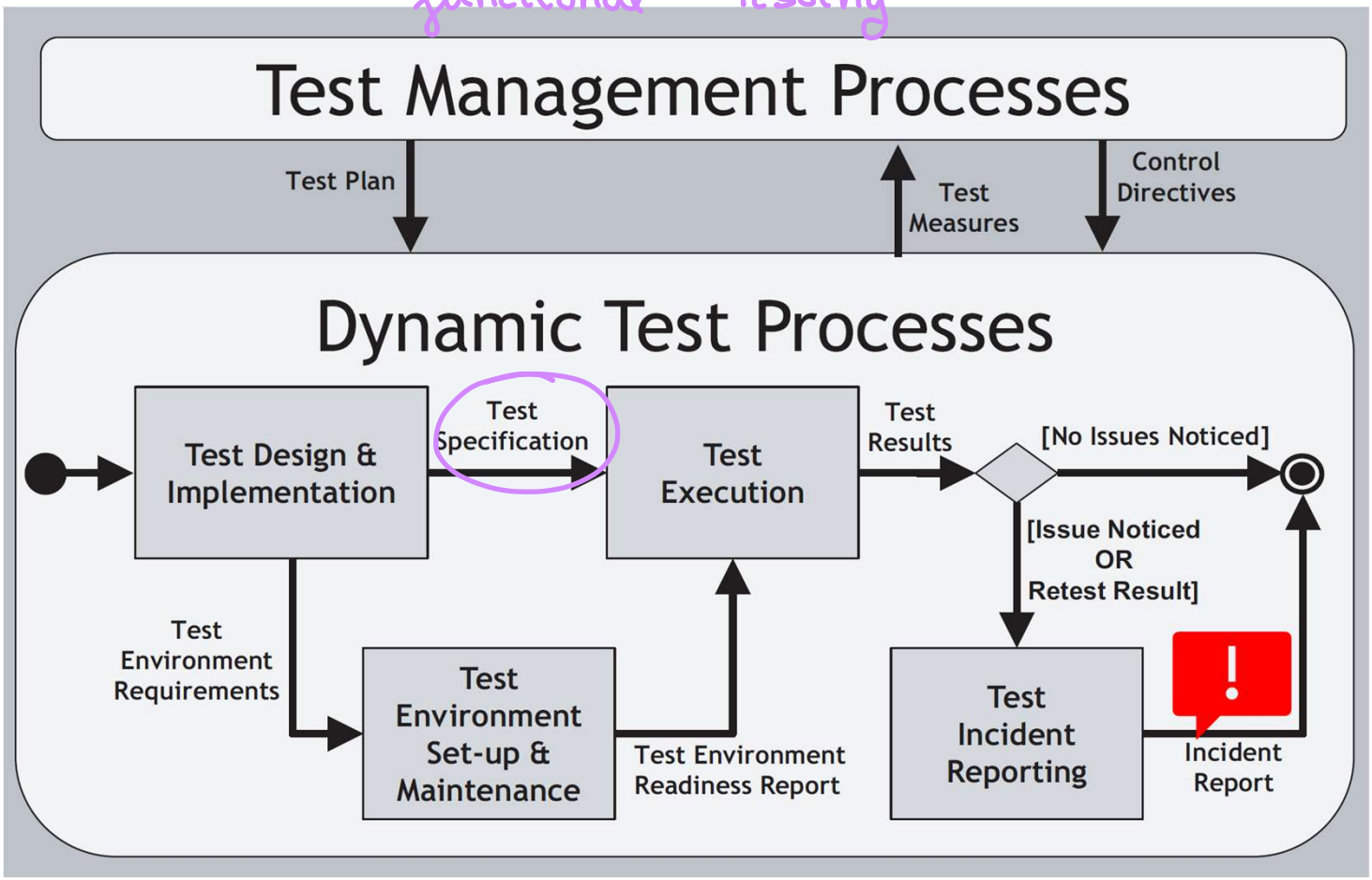
- Scripts
- db
- document

↓

Save in Repo

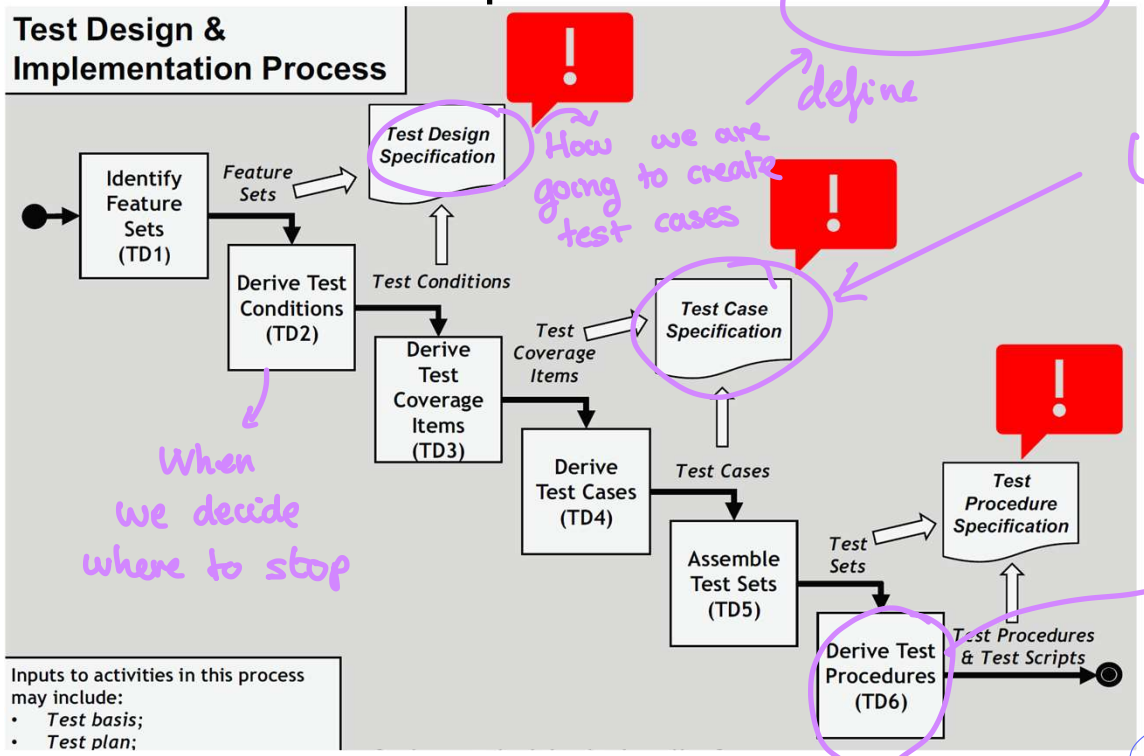
4. Dynamic Test Processes

- The **Dynamic Test Processes** are used to carry out dynamic testing within a particular phase of testing (e.g. unit, integration, system and acceptance) or type of testing (e.g. performance testing, security testing, usability testing).



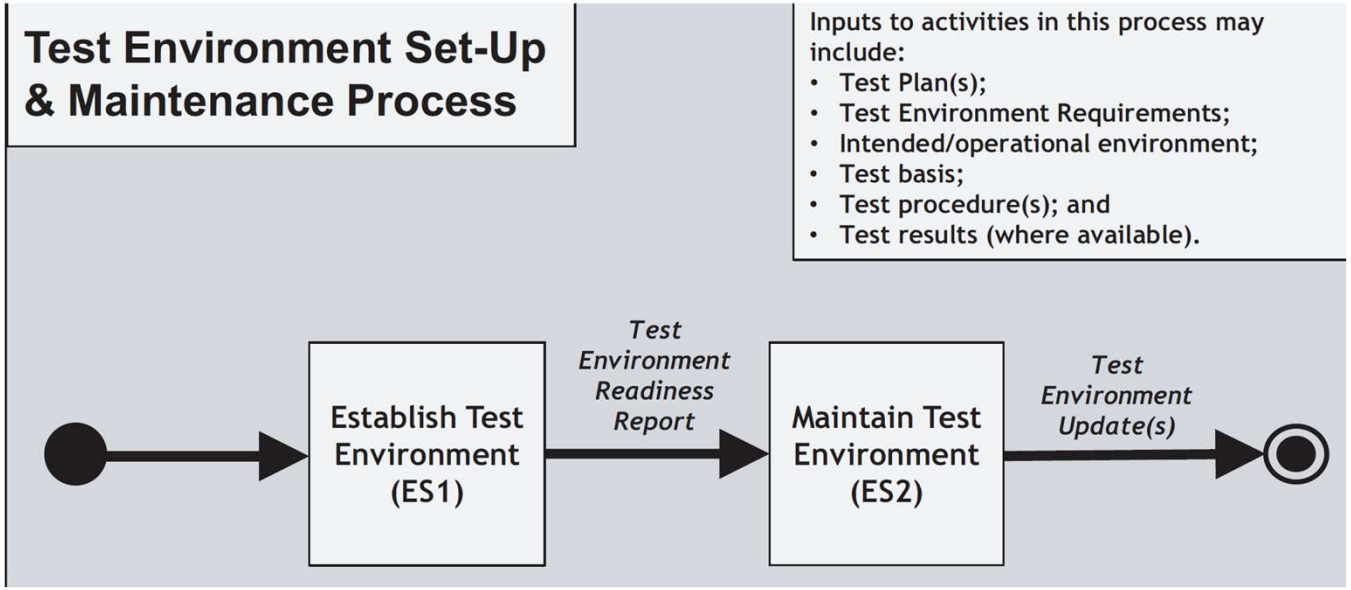
4. Dynamic Test Processes

- The **Test Design & Implementation Process** is used to derive test cases and test procedures; these are normally documented in a test specification. This process requires testers **to apply one or more test design techniques to derive test cases** and test procedures with the ultimate aim of achieving the test completion criteria, typically described in terms of test coverage measures. Those test design techniques and test completion criteria to use are specified in the Test Plan.



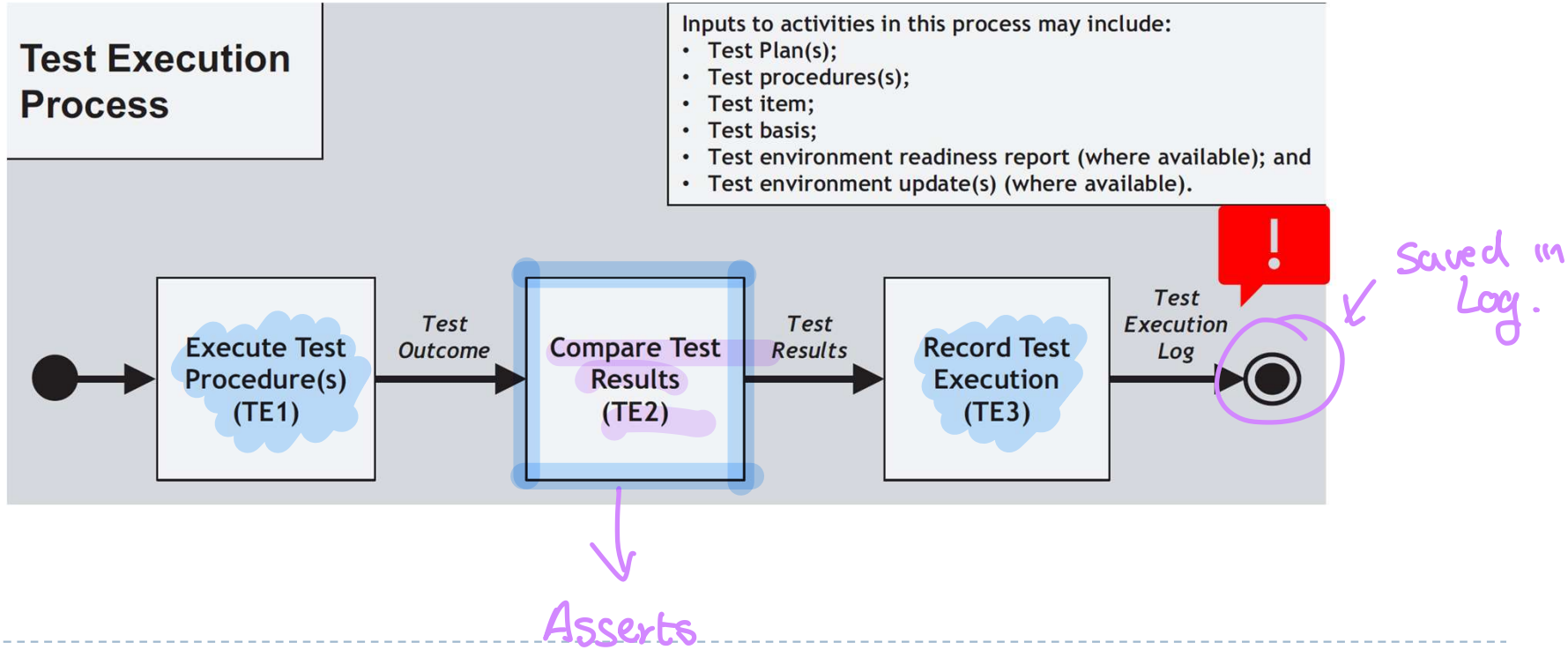
4. Dynamic Test Processes

- The **Test Environment Set-Up & Maintenance** used to establish and maintain the environment in which tests are executed. Maintenance of the test environment may involve changes based on the results of previous tests. **Where *change and configuration management processes exist*,** changes to the test environments may be managed using these processes. The requirements for a test environment will initially be described in the Test Plan, but the detailed composition of the test environment will normally only become clear once the Test Design & Implementation Process has started.



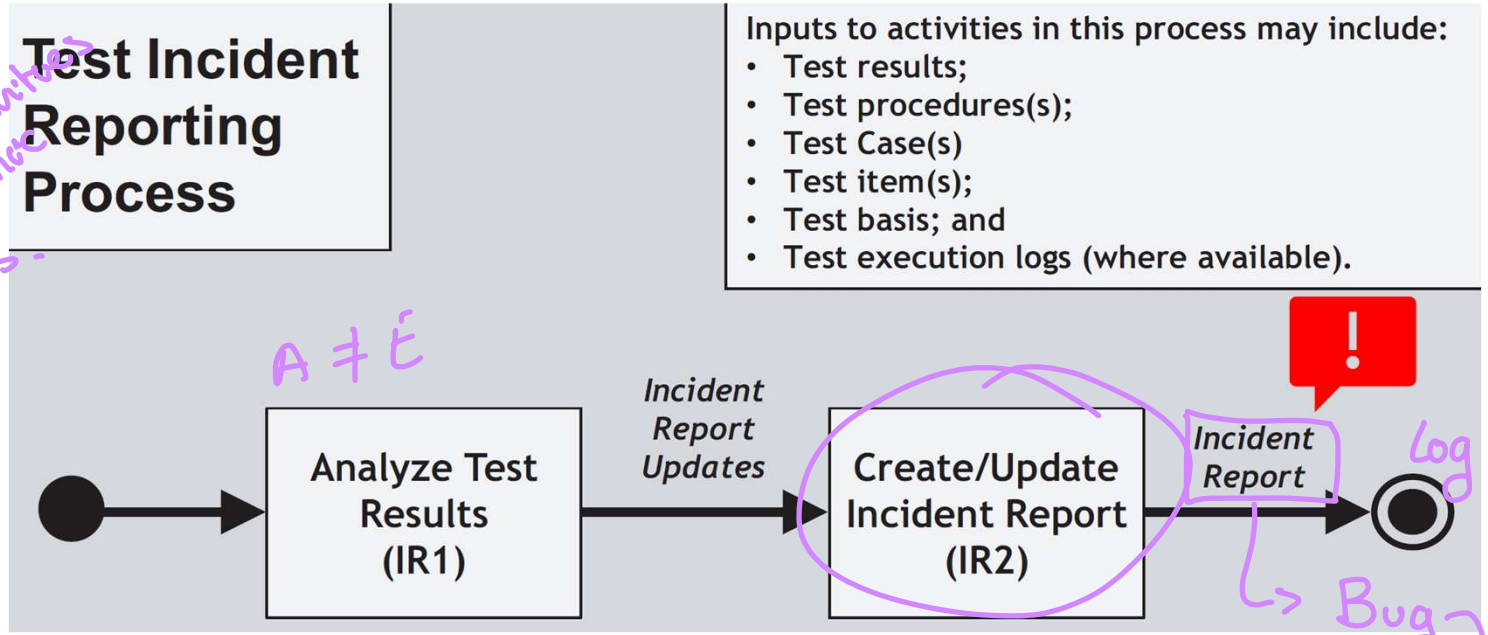
4. Dynamic Test Processes

- **The Test Execution Process** is used to run the test procedures generated as a result of the Test Design & Implementation Process on the test environment established by the Test Environment Set-Up & Maintenance Process. The Test Execution Process may need to be performed a number of times as all the available test procedures may not be executed in a single iteration. If an issue is fixed it should be retested by re-entering the Test Execution Process.



4. Dynamic Test Processes

- The **Test Incident Reporting Process** is used for the reporting of test incidents. This process will be entered as a result of the identification of test failures, instances where something unusual or unexpected occurred during test execution, or when a retest passes. In the case of a new test this will require an incident report to be created. In the case of a **retest**, this will require the status of a previously-raised incident report to be updated, but may also require a new incident report to be raised where further incidents are identified.



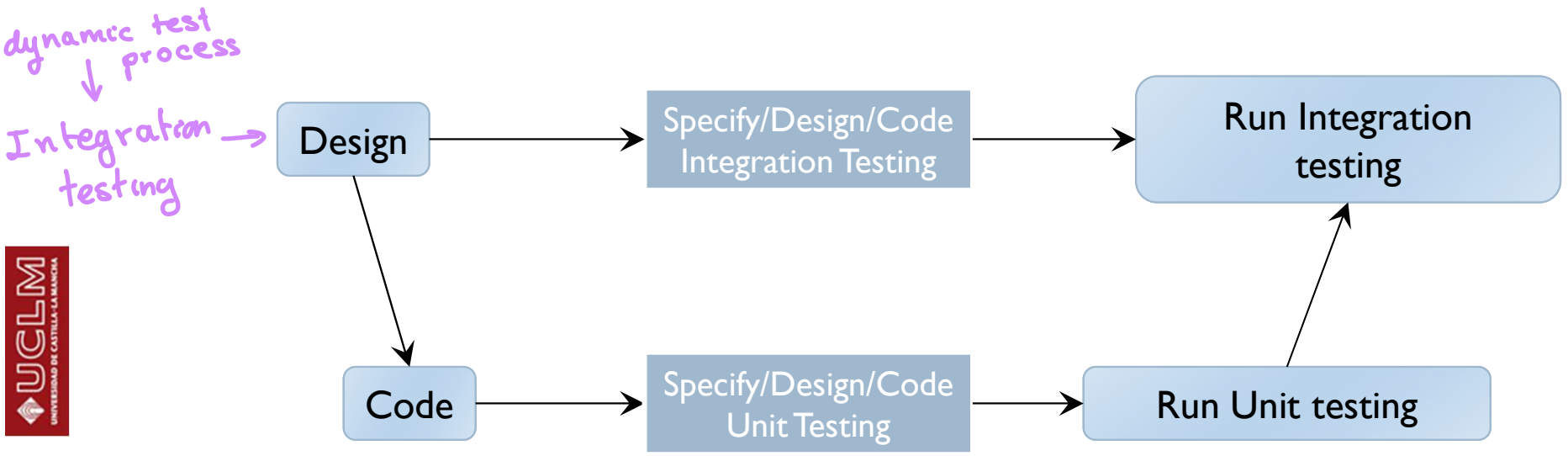
Exam: Describe the activities for the dynamic test process.

A ≠ É

Bug ← Update Azure Devops

5. Integration testing

- An alternative for ordering testing levels:

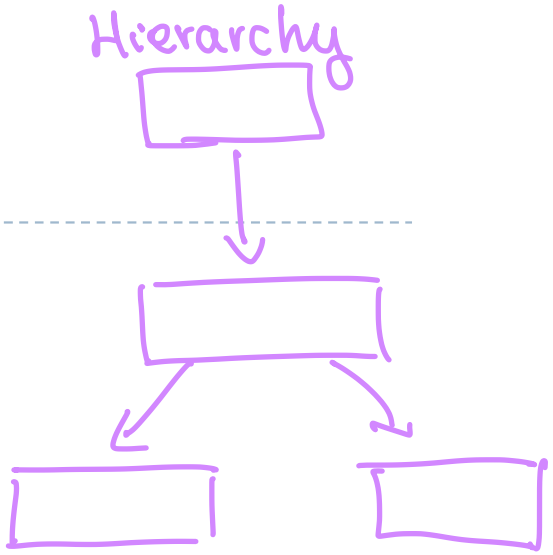


5. Integration testing

- **Integration test** the **progressive linking** and testing of programs or modules in order to ensure their proper functioning in the **complete system** [IEEE24765]
- **Integration testing:** testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them. [IEEE24765]
- **Goal**: testing conducted on **multiple complete, integrated systems** to evaluate their ability to **communicate successfully with each other** and to **meet** the overall integrated **systems' specified requirements**.

integrated system → comunicate → reach overall (goal) → requirements

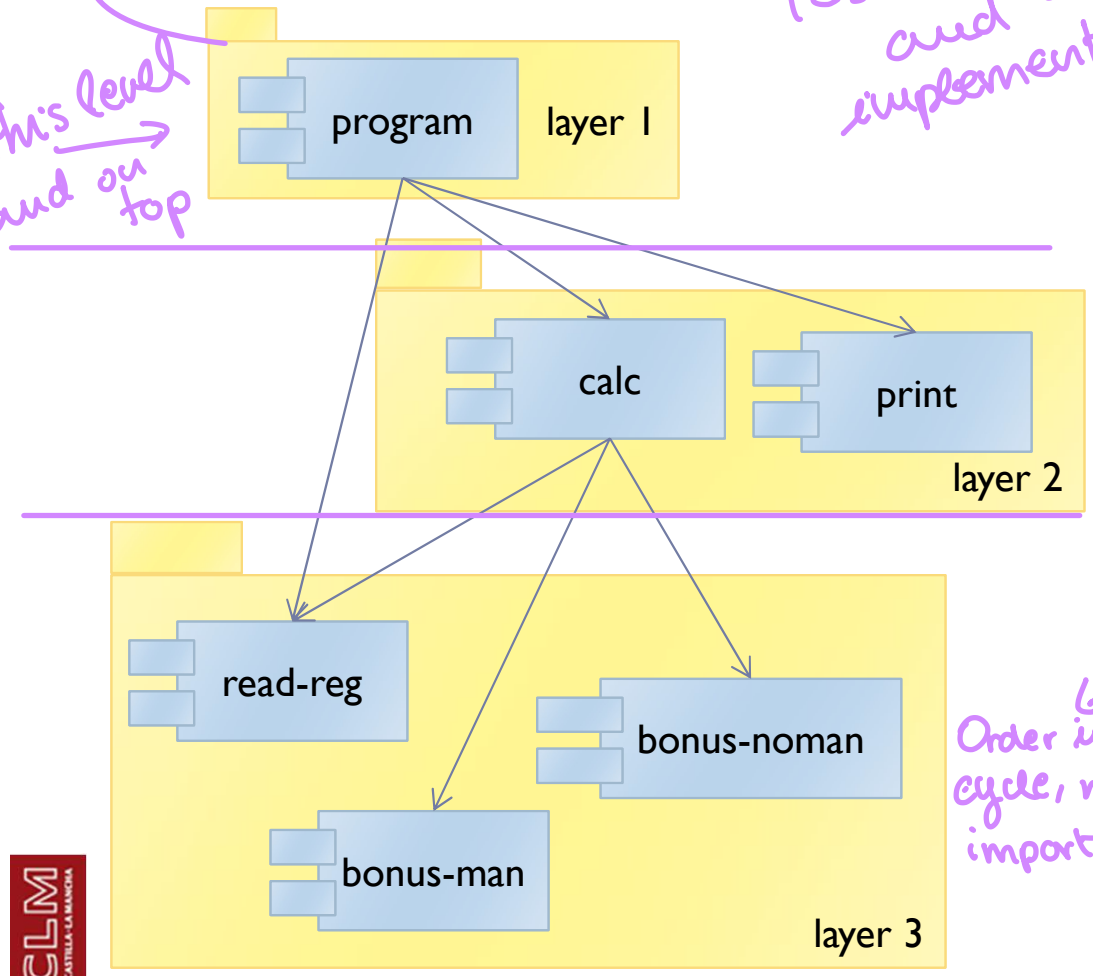
5. Integration testing



- Two approaches:
  - **Incremental testing**:
    - Constraint: The product must be hierarchically designed
    - Technique:
      - Each piece of software is tested individually (unit testing)
      - The different units are assembled using either **bottom-up** or **top-down** strategies, and applying white and/or black box testing to generate the test cases
    - **Advantages:** faults can be more easily located
    - **Disadvantages:** it is necessary driver and stub code
    - **Automation:** XUnit, JUnit, SimpleTest, MSTest…
  - **Big-bang testing:** software elements, hardware elements, or both are combined all at once into an overall system, rather than in stages
    - Technique: units are not tested until the whole system is assembled, then the system is tested as a whole
    - **Advantages**: driver and stub code is not necessary
    - **Disadvantages**: faults are hardly isolated

5. Integration testing

- Top-down testing:

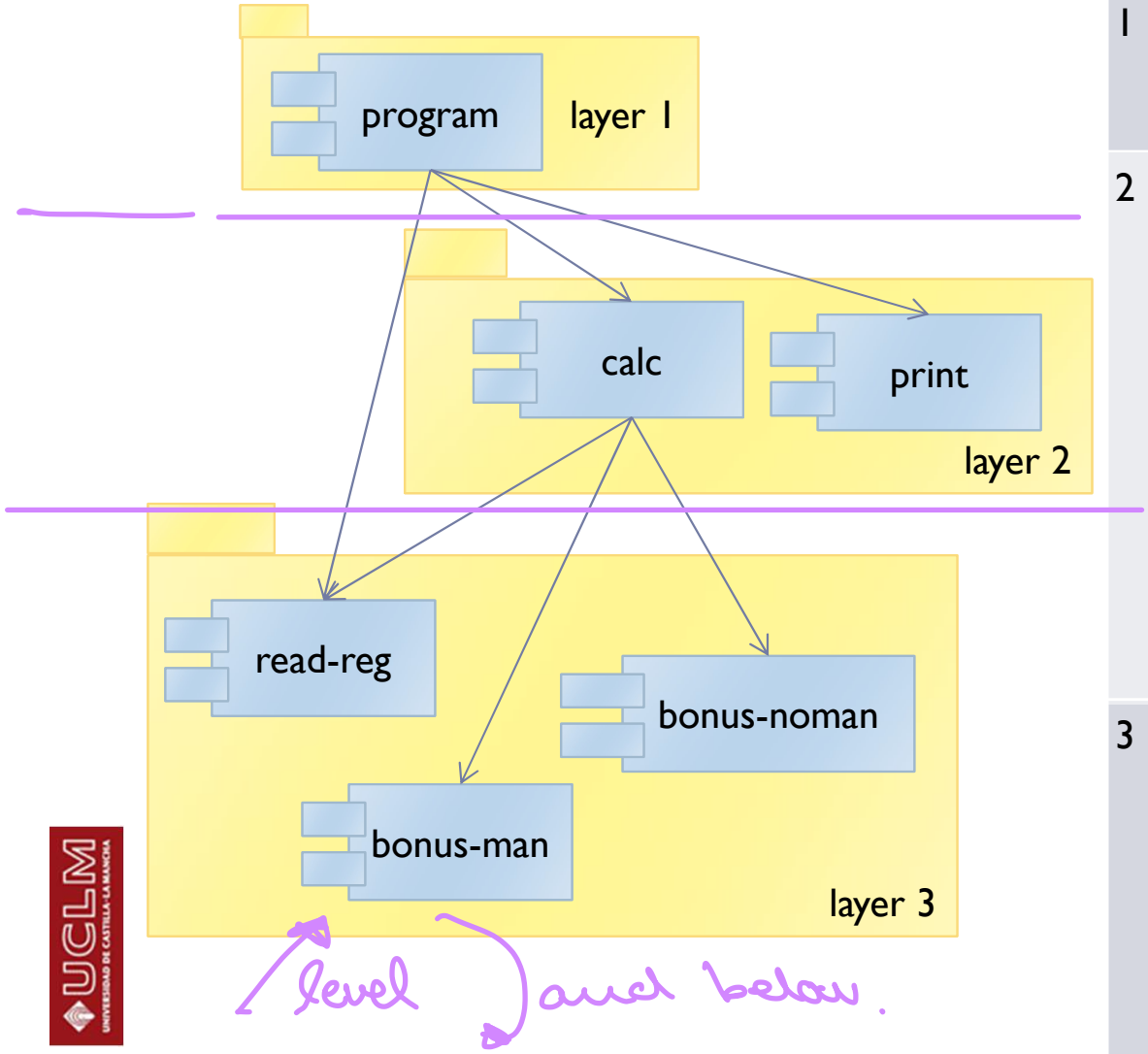


Nº	Unit	~~stub~~ mock	driver
1	program	calc, print, read-reg	program
2	program, **calc**	print, read-reg, bonus-man, bonus-noman	program
	program, calc, **print**	read-reg, bonus-man, bonus-noman	program
3	program, calc, print, **read-reg**	bonus-man, bonus-noman	program
	program, calc, print, read-reg, **bonus-man**	bonus-noman	program
	program, calc, print, read-reg, bonus-man, **bonus-noman**		program

5. Integration testing

Automatic testing of everyone of the unit.

- Bottom-up Testing:



Nº	Units	stub	driver
1	read-reg bonus-man bonus-noman		read-reg bonus-man bonus-noman
2	print calc, **read-reg** calc, read-reg, **bonus-man** calc, read-reg, bonus-man, **bonus-noman**	 bonus-man, bonus-noman bonus-noman	print calc calc calc
3	**program**, calc, read-reg, bonus-man, bonus-noman program, calc, **print**, read-reg, bonus-man, bonus-noman	print	program program

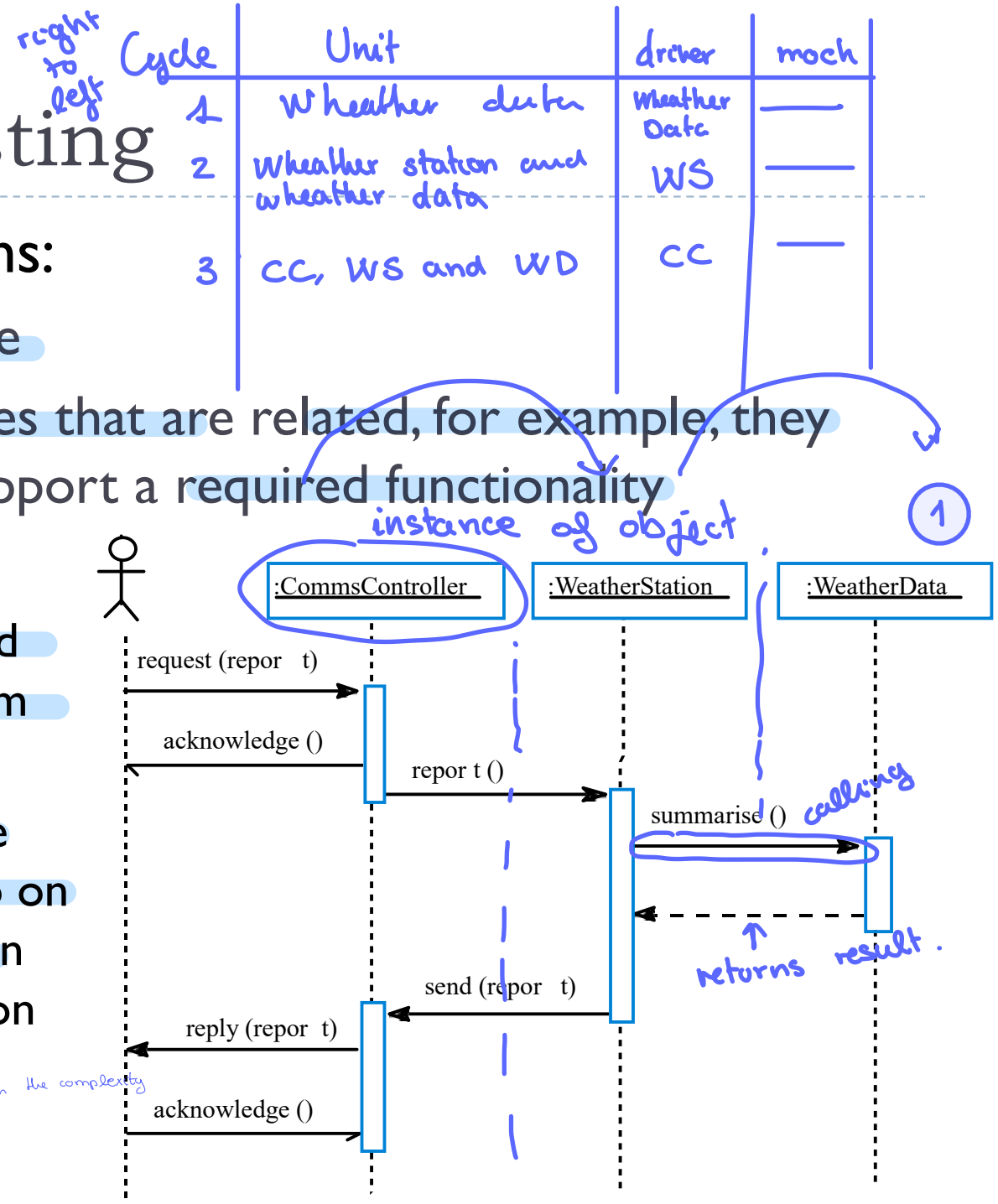
5. Integration testing

We can mix systems

- **Top-down** versus **bottom-up** testing:
  - Bottom-up:
    - Advantages:
      - lower level units are usually well tested early in the integration process
      - Driver code is more simple
    - Disadvantages:
      - Requires driver code
      - Upper level units are tested later and consequently may not be so well tested
      - The system as a whole does not exist until the last module is integrated
  - Top-down:
    - Advantages: upper-level units are tested early in integration (more time to re-design)
    - Disadvantages: Complex stub code
  - Which alternative should we select?
    - Take into account risk factors: mission/safety/business critical functions should be assembled and tested early
    - Availability of the units to be tested: project plan

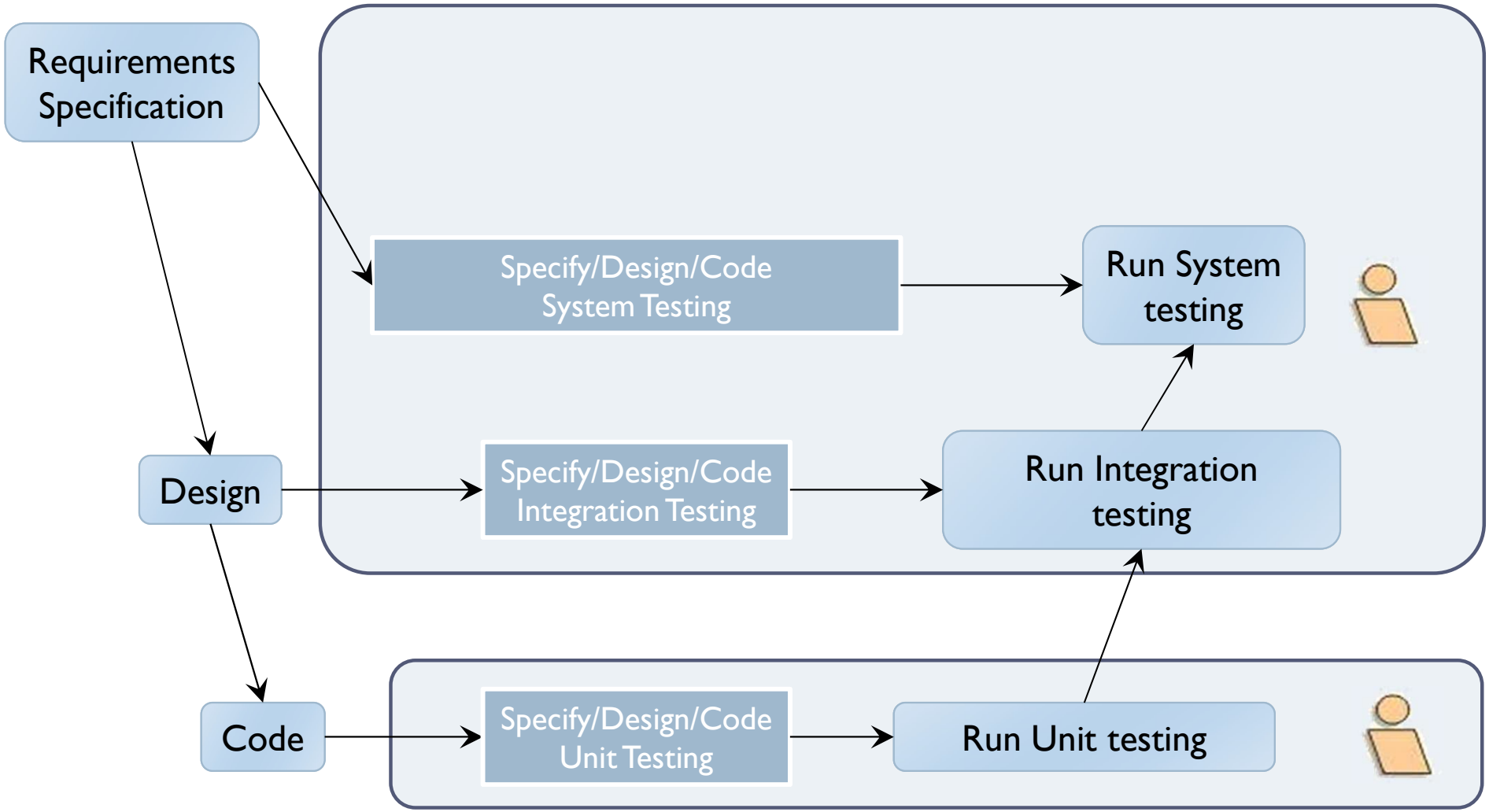
5. Integration testing

- Object-Oriented systems:
  - Hierarchy is not applicable
  - **Cluster** consists of classes that are related, for example, they may work together to support a required functionality
  - Testing technique:
    - First, select classes that send or request few services from other classes
    - Then, select classes that use them for integration, and so on until the successive selection leads to complete integration



6. System Testing

- An alternative for ordering testing levels:

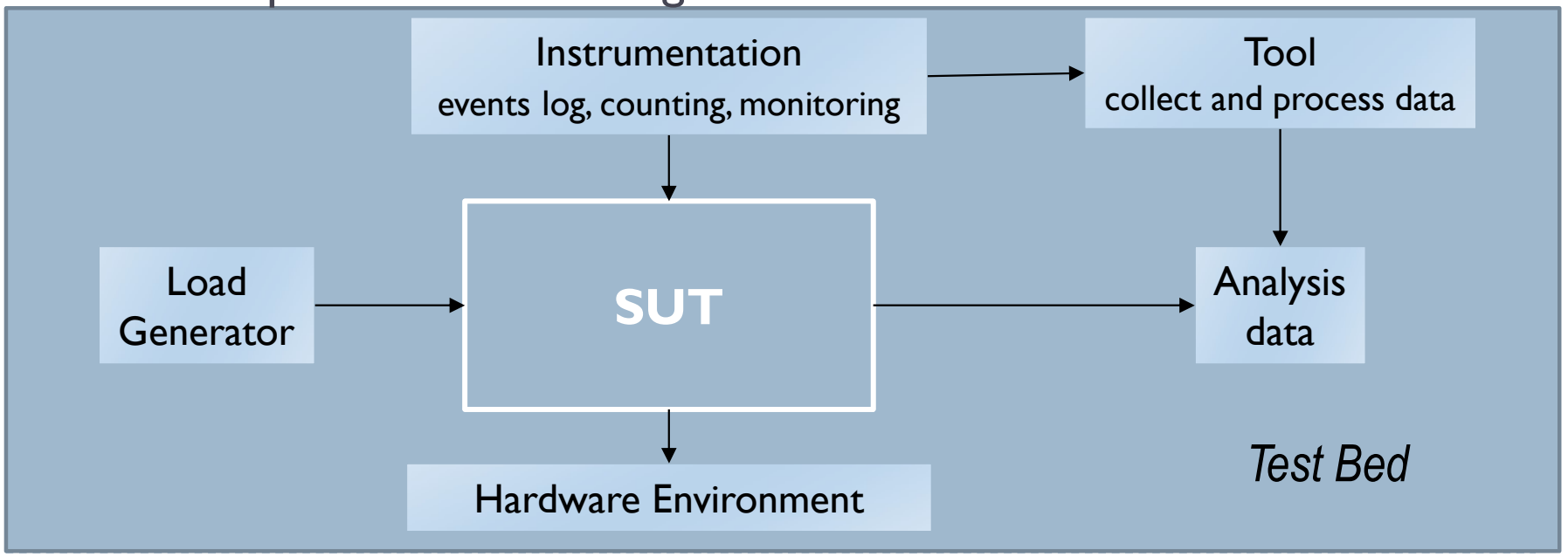


6. System Testing

- Testing conducted on a complete, **integrated system** to evaluate the system's **compliance** with its specified **requirements**
- Types of System testing:
  - Functional Testing
  - Performance Testing
  - Load & Stress Testing
  - Recovery Testing
  - Security Testing (subjects: Seguridad)
  - Usability testing (subject: IPO & IPOII)

6. System Testing: **Performance testing**

- Performance testing:
  - Goal: testing conducted to evaluate the compliance of a system or component with specified performance requirements
    - Ex. Re-assign pools of memory, modify priority levels, etc.
  - Requirements
    - Signed by the client
    - Quantified
  - Resources for performance testing:



6. System Testing: **Performance testing**

- Example of performance testing:
  - Response time of a transaction (medium, maximum)
  - Throughput: number of transactions per second
  - Workload: for instance number of simultaneous clients
  - Resources use: memory, storage, etc.

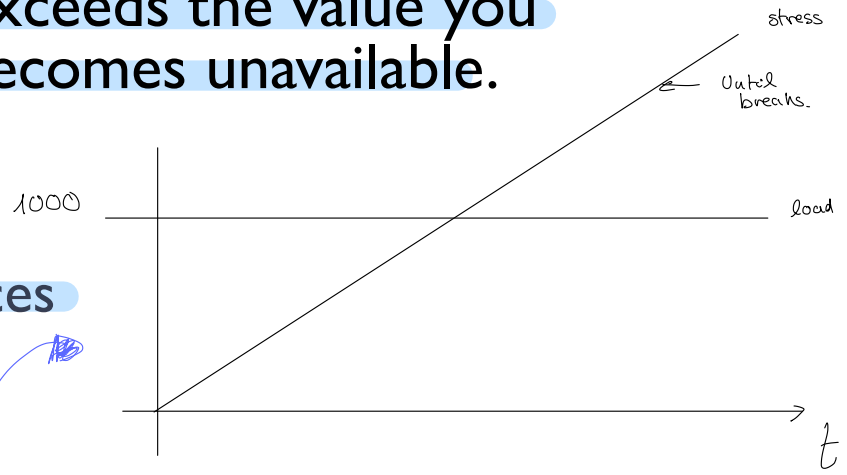
- Example of performance testing in RUP:
  - Information to design test cases:
    - Use Cases: special requirements section
    - Supplementary Specification

      At least one test case for each requirement specified
  - Specifying test cases:
    - Use Case: Search for Title of Movies

ID	Workload	Condition	Expected result
L01	1 customer	Search of a specified title	Response time <10 ms
L02	10 customer	Search of a specified title	Response time <10 ms
L03	100 customer	Search of a specified title	Response time <100 ms

6. System Testing: **Load & Stress Testing**

- **Load Testing**: [Molyneaux] testing conducted to evaluate a system or component at the limits of its specified requirements
- **Stress Testing**: [Molyneaux] to determine the upper limits or sizing of the infrastructure. A stress test continues until something breaks: no more users can log in, response time exceeds the value you defined as acceptable, or the application becomes unavailable.
- Advantages:
  - Reveal defects in real time systems
  - Poor design that cause unavailability of services
  - Build customers trust
- Resources: test bed
- Applicable to: unit, integration, system, sub-system
- Example: The AppForMovies can support till 1000 simultaneous users, how would the following tests be?
  - Load testing:
  - Stress testing:



You simply can't get by without it at Google as all our applications are heavily used and our data centers can be busy places.

6. System Testing

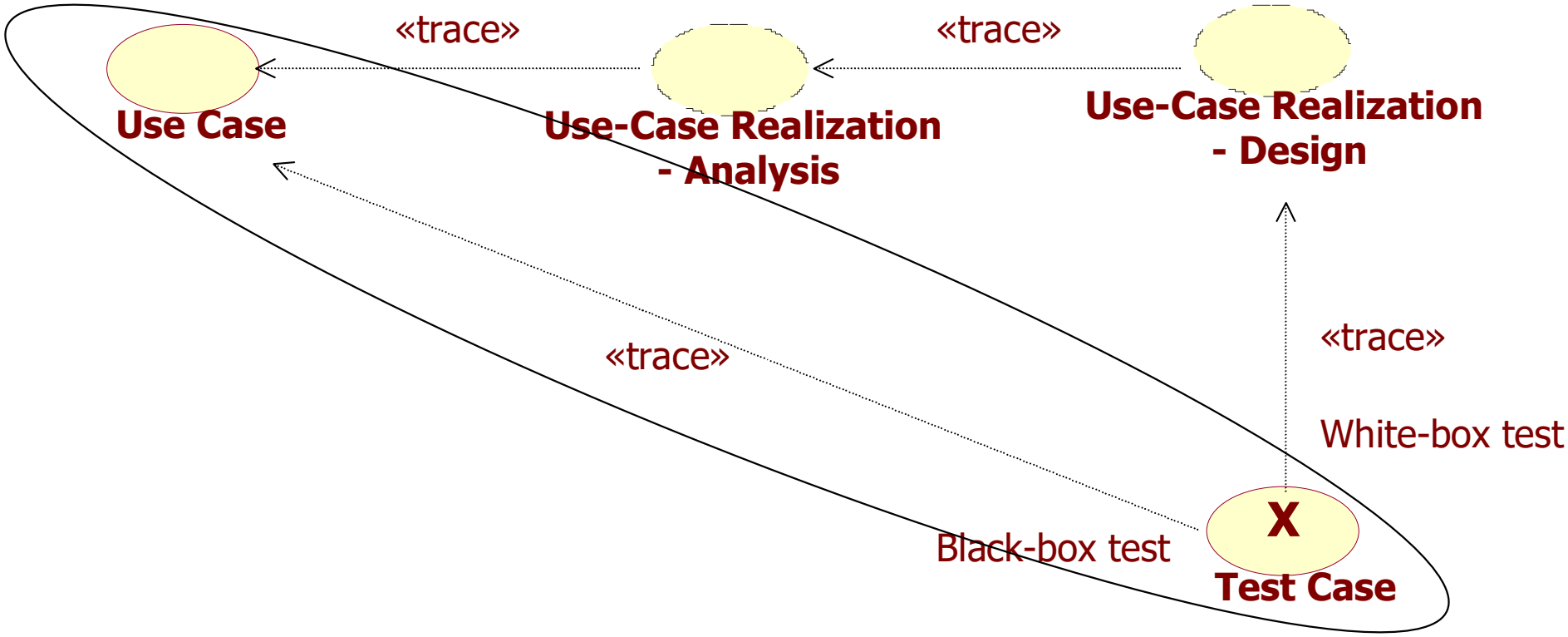
Black box system

You know nothing

- Functional Testing
  - Goal: to evaluate the compliance of a system or component with specified functional requirements.
  - Types of tests: usually black box testing (inputs and outputs of the evaluated functionality)
    - Equivalence class and Boundary Analysis
  - When: during the requirements stage
  - Principal points to be exercised:
    - Legal inputs
    - Illegal inputs
    - All likely outputs
    - All states and states transitions
    - All functions

6. System Testing
Functional Testing RUP – Test Cases

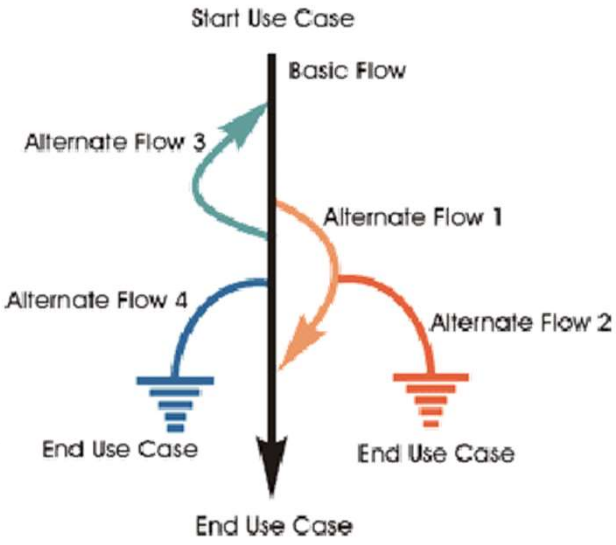
- RUP: Guided by Use Cases



6. System Testing
Functional Testing RUP – Test Cases

- RUP: Guided by Use Cases

Name	Of the use case
Brief description	Brief description of the use case
Basic Flow	How the system interacts with the actors
Special requirements	Non functional
Pre-conditions	Constraints to start the use case
Post-conditions	Constraints about the end of the use case
Alternative Flows	Alternatives to the interaction



UC scenario: a whole path through the UC

Scenario1	Basic Flow		
Scenario2	Basic Flow	Alternative Flow 1	
Scenario3	Basic Flow	Alternative Flow 1	Alternative Flow 2
Scenario4	Basic Flow	Alternative Flow 3	
…			

6. System Testing
Functional Testing RUP – Test Cases

- Process to generate Test Cases from Use Cases:
  1. For each UC generate a whole set of Scenarios:
     - **Scenario**: an instance of a use case, or a complete "path" through the use case

Scenario1. Customer successfully added	Basic Flow	
Scenario2. Customer does not exist	Basic Flow	2.2.1
….		

3 diff alternative → 4 scenarios normal +3

  2. Generate Test Cases: at least one Test Case per Scenario although it depends on the input conditions
     - Re-read the UC looking for conditions and inputs
     - Describe the Test Case

ID	Scenario/test condition	Input 1		Input N	Expected result
…	…	…	…	…	…

For every test case

↳ Notify bugs found

5 test cases for the mandatory data

6. System Testing
Functional Testing RUP – Test Cases

- Process to generate Test Cases from Use Cases:
  3. Identify data values for each Test Case:
     - Review and validate the Test Cases
     - Assign values for the input conditions
- **Advantages:**
  - Use Cases developed just at the beginning of the software development process
  - Test Cases generally related with the latest stages of the software development process

  - Testing team starts to define Test Cases early ⇒ early detection of defects
  - Coverage of testing improved

6. System Testing: Functional Testing RUP University Course Registration

1. Sign in

This use case starts when a Student accesses the Wylie University Web site. The system asks for, and the Student enters, the student ID and password.

2. Select 'Create a Schedule'

The system displays the functions available to the student. The student selects "Create a Schedule."

3. Obtain Course Information

The system retrieves a list of available course offerings from the Course Catalogue System and displays the list to the Student.

4. Select Courses

The Student selects four primary course offerings and two Alternative course offerings from the list of available course offerings.

5. Submit Schedule

The student indicates that the schedule is complete. For each selected course offering on the schedule, the system verifies that the Student has the necessary prerequisites.

6. Display Completed Schedule

The system displays the schedule containing the selected course offerings for the Student and the confirmation number for the schedule.

6. System Testing: Functional Testing RUP Alternative Flows

- FA1. Unidentified Student. Alternative to Step 1

If the system determines that the student ID and/or password is not valid, an error message is displayed.

- FA2. Quit. Alternative to any step

The Course Registration System allows the student to quit at any time during the use case. The Student may choose to save a partial schedule before quitting. All courses that are not marked as "enrolled in" are marked as "selected" in the schedule. The schedule is saved in the system. The use case ends.

- FA3. Course Catalogue System Unavailable. Alternative to Step 3

If the system is down, a message is displayed and the use case ends.

- FA4. Course Registration Closed. Alternative to any step

If, when the use case starts, it is determined that registration has been closed, a message is displayed, and the use case ends.

- FA5. Unfulfilled Prerequisites: Course Full, or Schedule Conflicts. Alternative to Step 5.

If the system determines that **prerequisites** for a selected course are **not satisfied** because the **course is full**, or that there are **schedule conflicts**, the system will not enrol the student in the course. A message is displayed that the student can select a different course. The use case continues at Step 4, Select Courses, in the basic flow.

6. System Testing
Functional Testing RUP – Test Cases

Id Scenario	Successful registration	Flows
Scen-1	Successful registration	FB
Scen-2	Unidentified student	FB+FA1
Scen-3	User quits	FB+FA2
Scen-4	Course catalog system unavailable	FB+FA3
Scen-5	Registration closed	FB+FA4
Scen-6	Cannot enroll	FB+FA5

6. System Testing
Functional Testing RUP – Test Cases

Id –TC	Scenario/ Condition	ID	Password	Courses selected	System Unavailable	Enroll. Open	Full Course	Conflicting Schedule	Expected results
UC1-1	Scen-1 successful registration	enavarro	123ab12	42324 42320 42322 42321 42327 42326	No	Yes	No	No	Schedule and confirmation number displayed
UC1-2	Scen-2 unidentified student	fmontero	1						Error message; back to login screen
UC1-3	Scen-3 valid user quits	fmontero	456654						A Login screen appears
UC1-4	Scen-3 valid user quits saving the schedule	fmontero	456654	42327 42326					A Login screen appears. Schedule saved
UC1-5	Scen-4 course registration system unavailable	fmontero	456654		Yes				Error message; back to step 2

6. System Testing
Functional Testing RUP – Test Cases

Id –TC	Scenario/ Condition	ID	Password	Courses selected	System Unavailable	Enrollment Open	Full Course	Conflicting Schedule	Expected results
UC1-6	Scen-5 registration closed	fmontero	456654			No			Error message; back to step 2
UC1-7	Scen-6 cannot enroll -- Full course	fmontero	456654	42324 42320 42322 42321 42327 42326			42326	No	Error message; back to step 4
UC1-8	Scen-6 cannot enroll -- Conflicting schedule	fmontero	456654	42324 42320 42322 42321 42327 42326			No	42324 42320	Error message; back to step 4

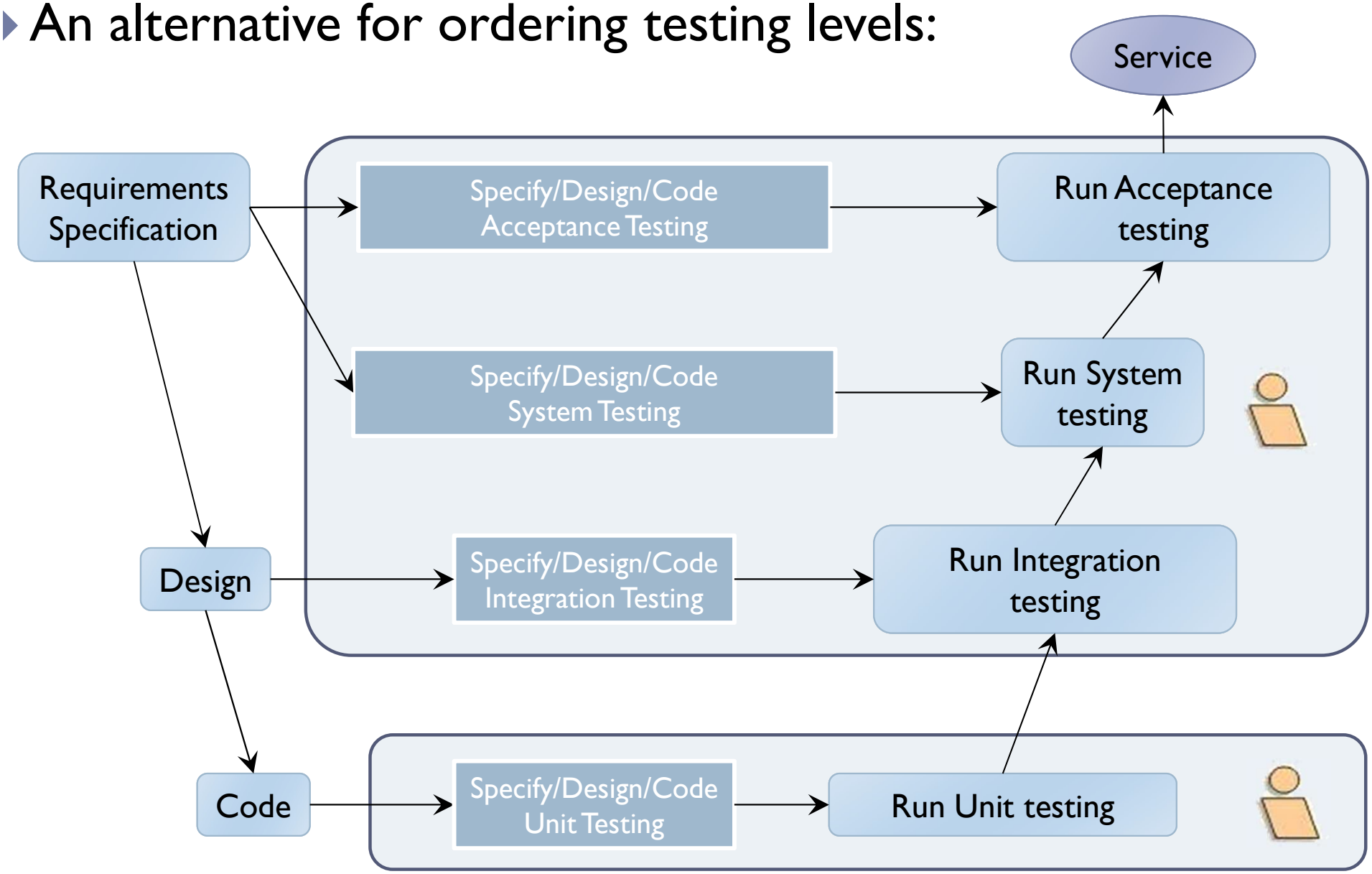
6. System Testing. **Tools**

- IBM:
  - Functional Tester
  - Rational PurifyPlus
  - Rational Performance Tester
- Testing for .NET:
  - Visual Studio: test projects
- Testing for Eclipse
  - Eclipse Test & Performance Tools Platform Project
  - JUnit
- Testing for Web:
  - Selenium: http://www.seleniumhq.org/
  - http://www.websiteoptimization.com/services/analyze/
- JMeter: Performance testing
  - https://jmeter.apache.org/
- Android:
  - https://developer.android.com/training/testing/espresso/basics

Automation is key to the long-term success and efficiency of the test team and to guard against regressions. ***Google***

7. Acceptance Testing: **V Model**

- An alternative for ordering testing levels:

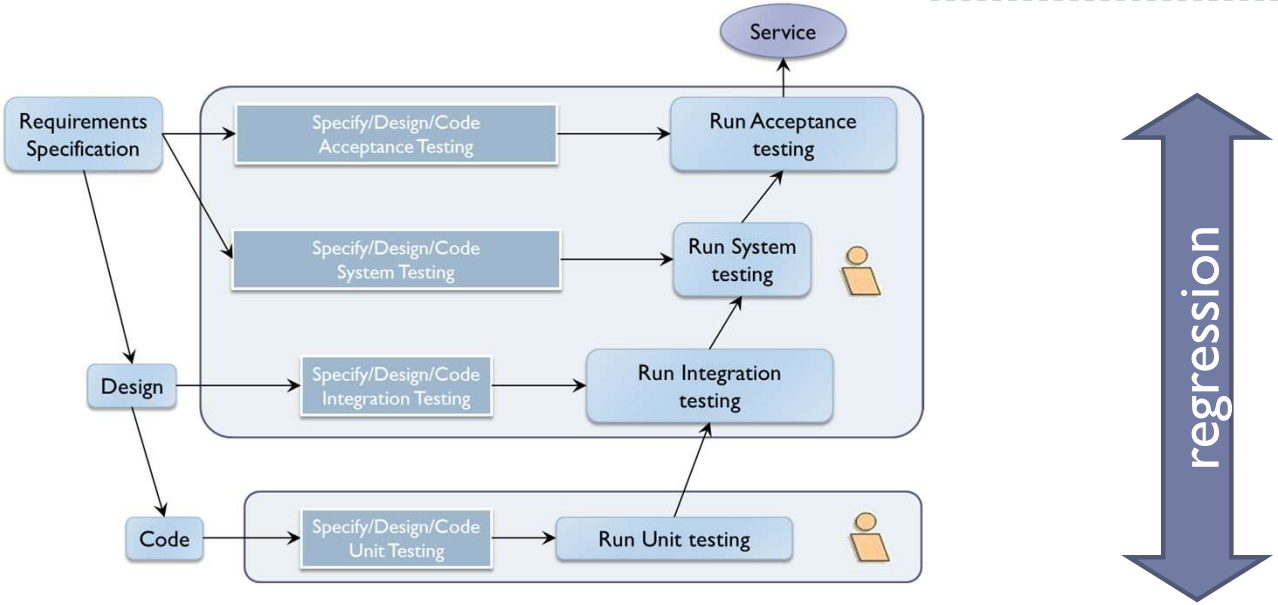


7. Acceptance Testing

- **Goal**: testing conducted to determine whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system.
- **When**: software developed **for a specific customer**, after system testing
- **Stakeholders**: users and testers
- **Pre-requirements**: system requirements and user manual, real environment (hardware & software)
- Points to be exercised:
  - Typical conditions of a working day
  - **Continuous systems: testing cycle of 25 hours**
  - Valid and invalid inputs of the main functionality
- Reuse System Testing

- When software is developed for **mass market**:
  - **Alpha testing**: at the developers' site, users are invited to test the system
  - **Beta testing:** System is sent to a cross-section of users who install it and use it under real world working conditions

8. Regression Testing



- Testing required to determine that a change to a system component **a)** has not adversely affected functionality, reliability or performance; **b)** and has not introduced additional defects.
- **Regression test. Retesting** to detect faults introduced by a modification
  - When must a test case belong to a regression test?
  - What happened after solving a Change Request?

References

- COLLARD, J.F, BURNSTEIN, I, Practical Software Testing: A Process-Oriented Approach, Springer. 2003
- WHITTAKER, J, ARBON, J., CAROLLO, J. How Google Tests Software, 2012
- [IEEE24765] ISO/IEC/IEEE 24765-2010 Systems and software engineering — Vocabulary
- [IEEE29119-2] ISO/IEC/IEEE 29119-2:2021 Software and systems engineering Software testing Part 2: Test processes
- [IEEE29119-3] ISO/IEC/IEEE 29119-3:2021 Software and systems engineering Software testing Part 3: Test documentation

Test document:

Carmen: 2, 8, 13, 16

Lucia: 1, 4, 6, 10

Javi: 3, 9, 11, 12

Agustín: 5, 7, 14, 15

Estructura -> Obtener purchase por fecha

```
[HttpGet]
[Route("[action]")]
[ProducesResponseType(typeof(DTO)), (int)HttpStatusCode.Ok)]   // Not found
public async Task<ActionResult> GetPurchase(DateTime date)
{
    if (_context.Purchases = null)
    { _logger.LogError("Error: Purchases is not available on the database");
      return NotFound(); }
    var purchasedetaildto = await _context.Purchases
    .Where(purchase => purchase.Date == date)
        .Include(purchase => purchase.PurchaseItems)
            .ThenInclude(purchaseItem => purchaseItem.Device)
                .ThenInclude(device => device.Model)
        .Select(purchase => new PurchaseDetailDto(todos,)
        .FirstOrDefaultAsync();
    if(purchasedetaildto == null)
    { _logger.LogError("    ");
      return NotFound();
    }
    return Ok(purchasedetaildto);
}
```

Filtros probados

Price For Purchase <=

Color

Fecha

Cantidad stock

Cantidad comprada

Brand

Pruebas

✓ Acordarte power ? en el test pues que funcione

✓

Igual que id -> pero con fecha

```
[TASK]
[Fact] Async Task <PerformAction>
```

Tipos De Respuestas | [ProducesResponseType(typeof(PurchaseDetailDTO)), (int) HttpStatusCode.Ok)] | [ProducesResponseType(typeof((int) HttpStatusCode.NotFound))]

[HttpGet]

[Route("[action]")]

GET -> public async Task<ActionResult> getSomething()

TEST

```
* [Fact]
public async Task GetSome()
{ //arrange
    Mock<ILogger<WhatController>> mockLogger = new Mock<ILogger<SomeController>>();
    SomeController sut = new SomeController(_context, MockLogger.Object());
  //act (recc DTO y variables
    var result = Sut.Method();

  //assert
  ///
}
```

Unit 6. Software Quality

1. Concept of Software Quality
2. Product Quality

Goals

- Understand the concept of Quality
- Understand the problematic related to the quality systems
- Understand the relation between quality product and quality process
- Under the existing metrics, when they should be employed and evaluate the results

SWEBOK definition

- The Software Quality KA deals with software quality considerations which transcend the software life cycle processes. The description of this KA covers three subareas.
  - The first subarea describes the **Software Quality Fundamentals** such as software engineering culture and ethics, the value and costs of quality, **models and quality characteristics**, and quality improvement.
  - The second subarea covers **Software Quality Management Processes**. The topics here are software quality assurance, verification and validation, and reviews and audits.
  - The third and final subarea describes **Practical Considerations** related to software quality. The topics are software quality requirements, defect characterization, software quality management techniques, and software quality measurement

1. Concept of Software Quality

- What is Software Quality? Two different views:
  - **Product Quality**:
    - Quality is **fitness for use** meaning that customers or users of a product should be able to count on it for what they needed it for. Quality is the absence of defects (Juran)
    - Quality is defined from the customer's point view, it is understood as **everything that increases its satisfaction** (Deming)
    - Quality is the whole set of characteristics of a product or service that has the ability to **satisfy implicit and stated needs** (ANSI)
    - To what extent a product **satisfy the requirements** (ISO 9000:2000)
    - **Set of properties and characteristics** of a product or service that provides it with the ability to satisfy implicit and stated needs (ISO 8402)
    - **Set of characteristics** inherent to a product, a component of product, or process, to wholly satisfy the **customer's requirements**

1. Concept of Software Quality

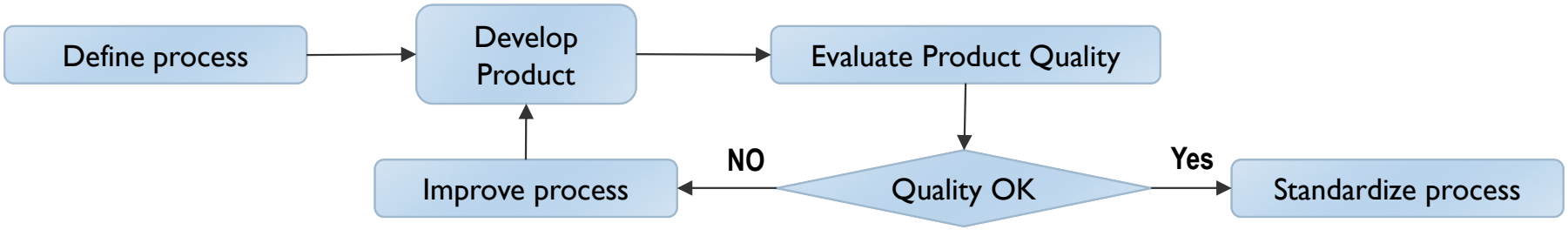
- What is Software Quality? Two different views:
  - **Process quality:**
    - Vision of development from the point of view of "Factory"
    - Establish a quality management system in a Company will allow it to produce a quality product
    - Different standards follow this approach: ISO9000, CMMI, ISO15504

CMM level	Minimum	Average	Maximum
1	150	750	4,500
2	120	624	3,600
3	75	473	2,250
4	23	228	1,200
5	2	105	500

Relation between the number of defects and the CMMI level of a organization [Jones, 2000]

1. Concept of Software Quality

- Product Quality VS Process Quality
  - Product quality is influenced by Process Quality
    - Important due to the difficulty to evaluate the quality attributes of a product
  - Difficulty to establish a relation between the SW product and process:
    - Application of individual abilities and experiences
    - External factors, as the novelty of the product



- Practices of process quality
  - Define process standards, such as reviews
  - Monitor the development process to ensure the product comply the standards
  - Inform of the process to the project management

2. Product Quality

- Quality is a subjective characteristic:
  - For the customer: a quality product is the one that can be labelled as "fitness for purpose"

- Quality is the satisfaction of the contractual requirements by both the software product developed, as well as the software development process

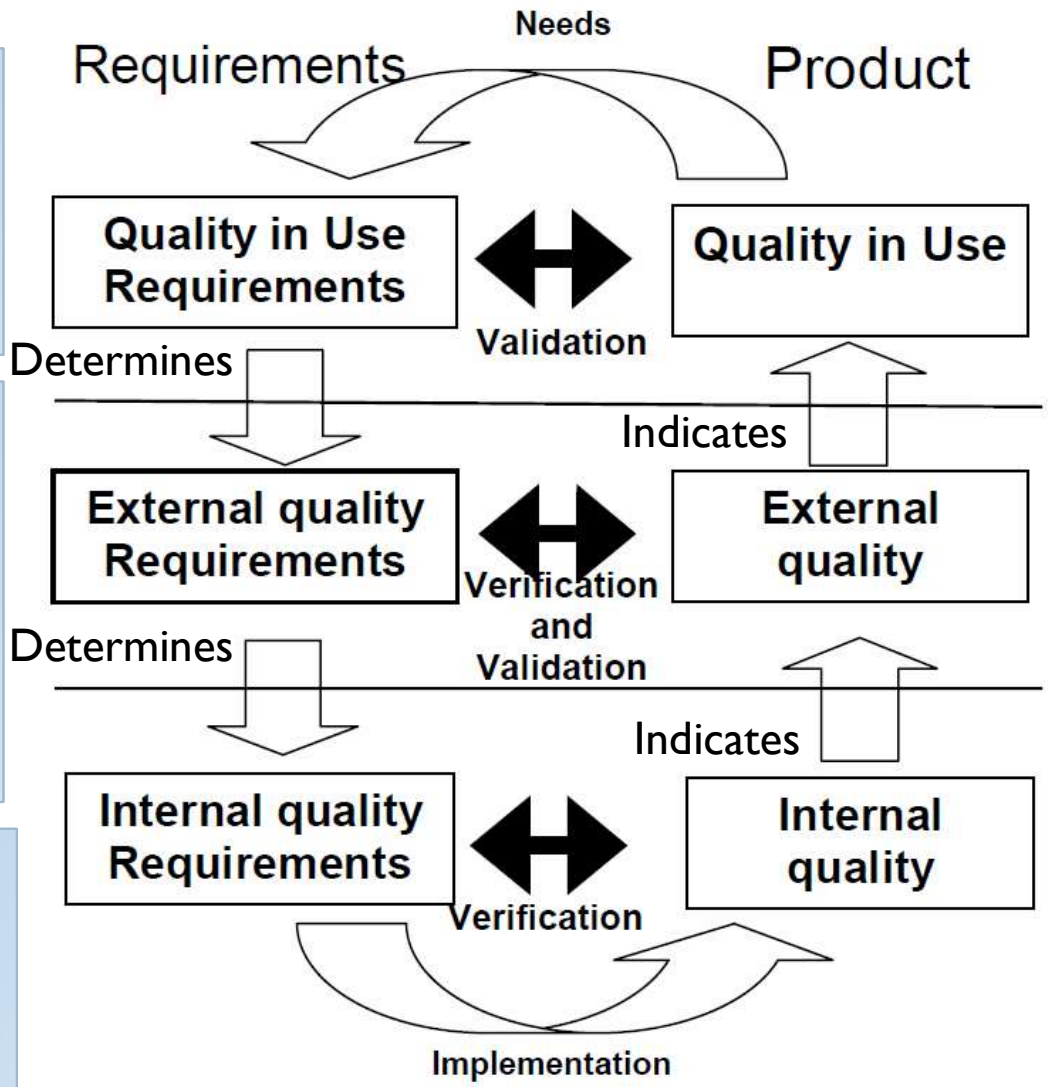
- **Quality Characteristics** (also called quality factors): characteristic whose presence or absence in a product determines its quality

2. Product Quality

Quality in use requirements. Used to **validate** the product by means of **metrics**.

External Quality Req.
They are the **criteria** (**goal**)used when a product is **evaluated.** They are detected by **users**.

Internal Quality Req.:
- only perceived by **developers**
- means for achieving External Quality requirements

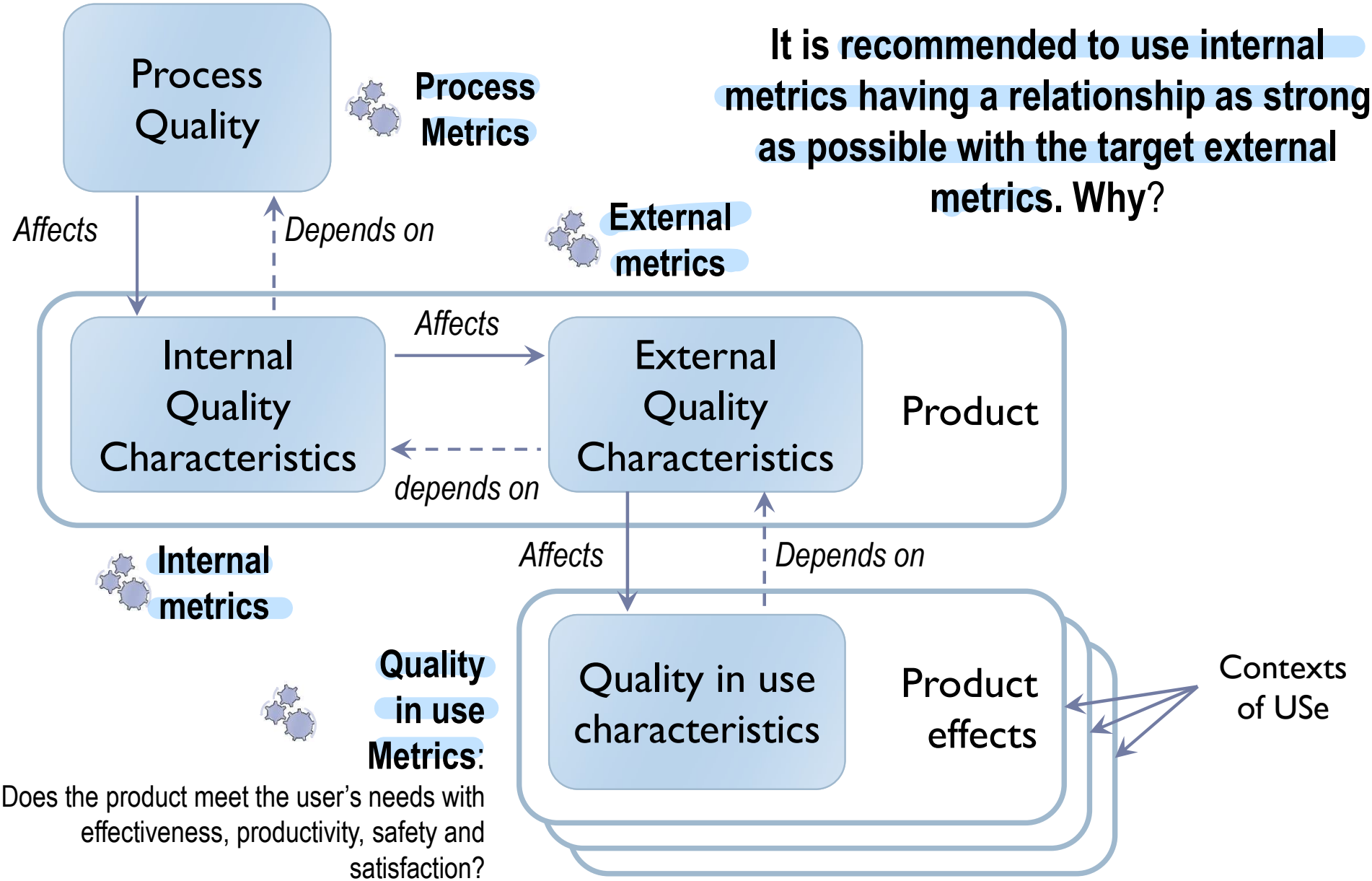


Quality in use:
It measures the extent to which users can achieve their goals in **a particular environment**, rather than measuring the properties of the software itself

External Quality:
is the quality **when the software is executed,** which is typically measured and evaluated while testing in a simulated environment with simulated data using external metrics

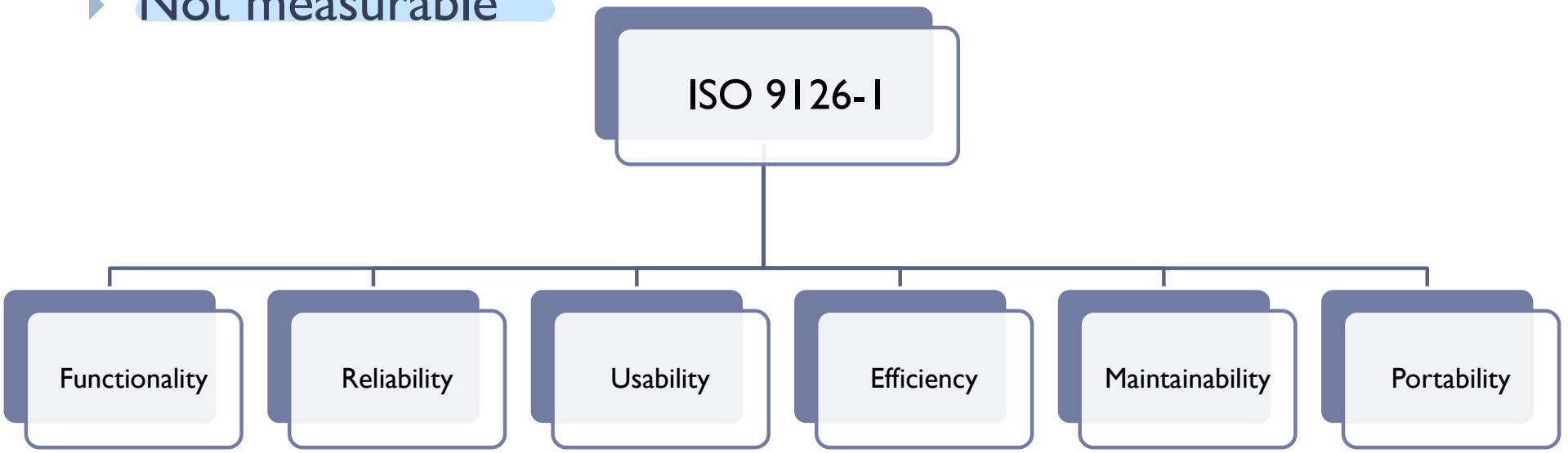
Internal Quality:
is measured and evaluated against the internal quality requirements (e.g. source code).

2. Product Quality

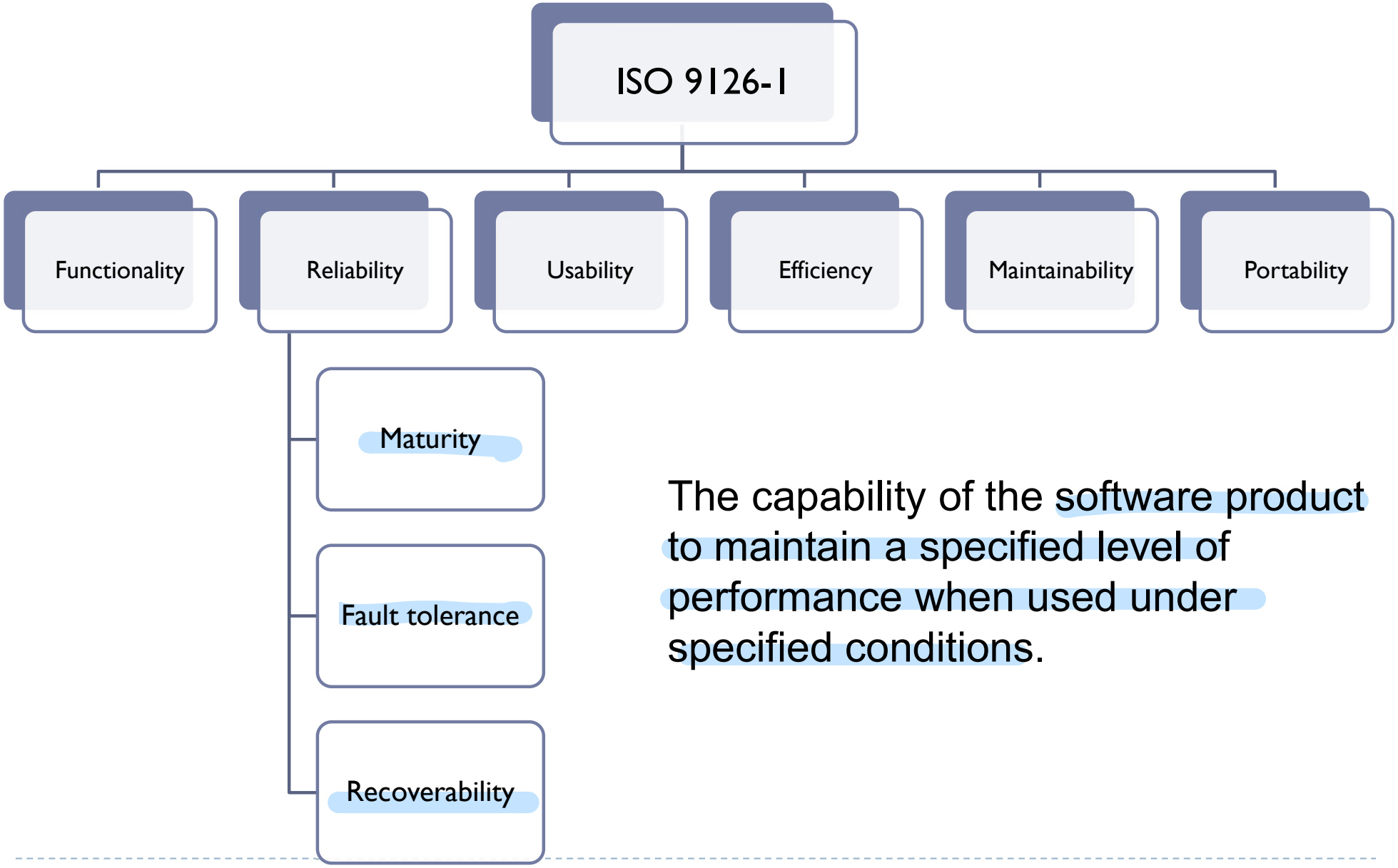


3. Product Quality Characteristics

- Quality Characteristics:
  - Attributes of quality of a product
  - Used to specify the Quality Requirements of a product
  - Not measurable

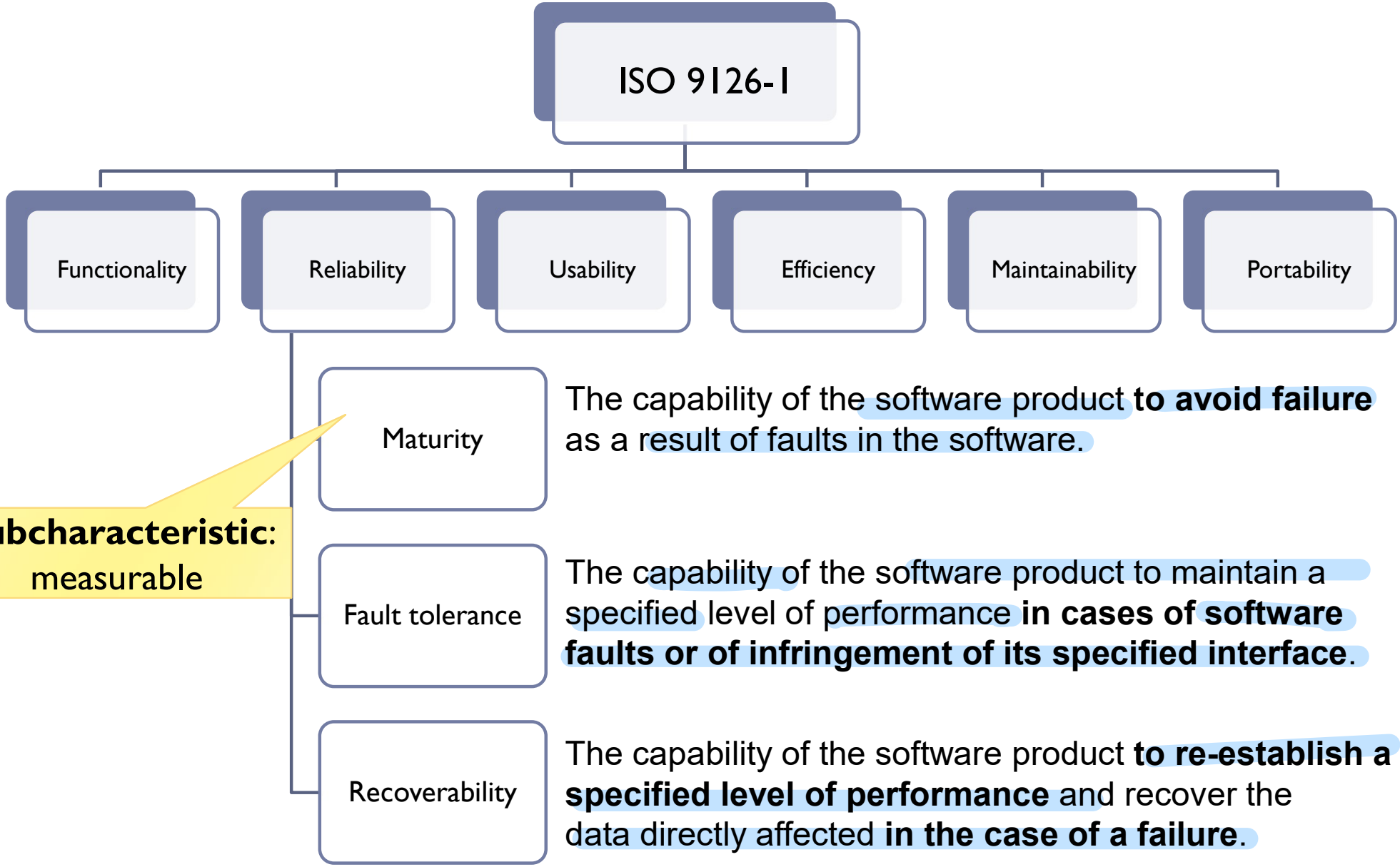


3. Product Quality Characteristics



The capability of the software product to maintain a specified level of performance when used under specified conditions.

3. Product Quality



3. Product Quality Characteristics

- Quality requirements are specified by using metrics that should be used as criteria when a product is evaluated:
  - ISO 9126-2: **external metrics:** for external quality requirements
    - Measure the product quality by measuring the behaviour of the system
    - It can be used only during the *testing stage* and during the operational stages
    - The measurement is carried out when the product is run in the system environment where it must work
  - ISO 9126-3: **internal metrics:** for internal quality requirements
    - It can be applied to a non-runnable software product, during the initial stages of the development process
    - They provide developers with the ability to measure the quality of intermedium deliverables so that they can estimate the quality of the final product
    - It allows the developers to identify the quality problems and initiate corrective actions as soon as possible during the lifecycle development

3. Product Quality Characteristics

- Quality Sub-characteristic: Maturity

It is recommended to use internal metrics having a relationship as strong as possible with the target external metrics. Why?

External maturity metrics

Metric name	Purpose of the metrics	Method of application	Measurement, formula and data element computations	Interpretation of measured value	Metric scale type	Measure type	Input to measure-ment
Mean time between failures (MTBF)	How frequently does the software fail in operation?	Count the number of failures occurred during a defined period of operation and compute the average interval between the failures.	a) X = T1 / A b) Y = T2 / A T1 = operation time T2 = sum of time intervals between consecutive failure occurrences A = total number of actually detected failures (Failures occurred during observed operation time)	0<X,Y The longer is the better.As longer time can be expected between failures.	a) Ratio b) Ratio	A = Count T1 = Time T2 = Time X =Time / Count Y =Time/ Count	Test report Operation (test) report

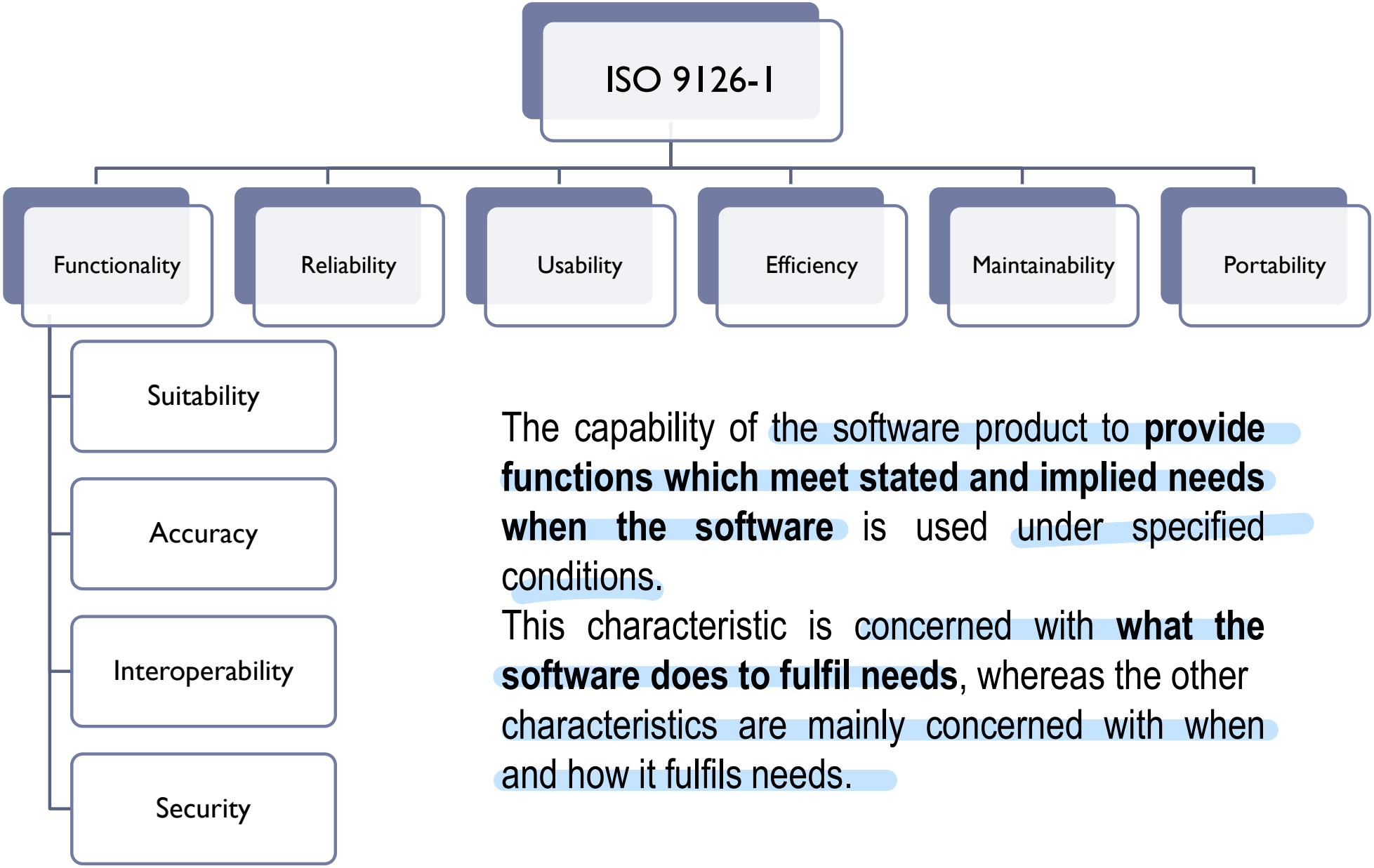
ISO 9126-2

Internal maturity metrics

Metric name	Purpose of the metrics	Method of application	Measurement, formula and data element computations	Interpretation of measured value	Metric scale type	Measure type	Input to measure-ment
Fault removal	How many faults have been corrected?		X=A A=Number of corrected faults in design/coding	0 <= X A high value of X implies, that less faults remain.	ratio	X=count A=count	Value A comes from fault removal report.
	What is the proportion of faults removed?	Count the number of faults removed during design/coding and compare it to the number of faults detected in review during design/coding.	Y=A/B A=Number of corrected faults design/coding B= Number of faults detected in review	0 <= Y <= 1 The closer to 1, the better. (more faults removed)	absolute	Y=count/count B=count	Value B comes from review report.

ISO 9126-3

2. Product Quality



The capability of the software product to **provide functions which meet stated and implied needs when the software** is used under specified conditions.

This characteristic is concerned with **what the software does to fulfil needs**, whereas the other characteristics are mainly concerned with when and how it fulfils needs.

2. Product Quality

- **Suitability**: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.
- **Accuracy**: The capability of the software product to provide the right or agreed results or effects with the needed **degree of precision**
- **Interoperability**: The capability of the software product **to interact with** one or more specified systems.
- **Security**: The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.

2. Product Quality

The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions

2. Product Quality

The capability of the software product to provide **appropriate response and processing times and throughput rates** when performing its function, under stated conditions.

The capability of the software product to use **appropriate amounts and types of resources** when the software performs its function under stated conditions.

Reference

- [IEEE, 1991] IEEE standard glossary of computer engineering terminology
- [KAN, 1995] Kan, S.H. Metrics and Models in Software Quality Engineering. Addison-Wesley. 1995
- [Galin, 2003] Galin, Software Quality Assurance: From theory to implementation, Addison-Wesley, 2003
- [Jones, 2000] Jones, C.: Software Assessments, Benchmarks, and Best Practices. Addison-Wesley Longman, Publishing Co., Boston (2000)

EXAM EXERCISES

2023/2024 Exam Ordinary:

1 UC: "Write a letter to the Three Wise Men" ~> Generation of test cases of Use Cases

a) Input and output specification

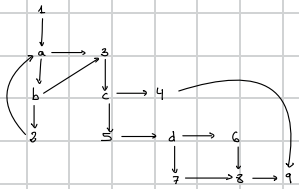
Id - Scen	Flows
Scen - 1	BF
Scen - 2	BF + AF1
Scen - 3	BF + AF2
Scen - 4	BF + AF3

We have to try the limits

Id	Scenario	Toys Selected	Address	Goodness Level	Letter	Expected Output
TC-1	Scen - 1 / goodness low	(Doll, Low, 5)	Calle Nueva	Low	I will try to be good.	letter, goodness, Address, toy
TC-2	Scen - 1 / goodness high	(Doll, low, 5) (Sauce, high, 5)	Albacete	High	I will be good	I will be good, High, Alb (Doll, 5, high)
TC-3	Scen - 2	"Only sugar..."				"Only sugar carrousel can be obtained" is shown.
TC-4	Scen - 3	"No toys available"				"No toys available"
TC-5	Scen - 4 / missing address	(Doll, low, 5)		High	I'll be good	Missing data, address mandatory
TC-6	Scen - 5 / missing goodness	(Doll, low, 5)	Albacete		I'll be good	Missing data, goodness level is mandatory

You have to do the test cases for all the missed data

2 GetToys ~> ToysController ~> PathCoverage technique



ID	PATH	Username	Age	Year	Mischefs	Toys	Pattern	Comment
a	1a3c49							uncoverable
b	1ab3c49							uncoverable
c	1 ab2a3c49	bart@uclm.es	10	2023	Ms	Ts	BadRequest	covers a.
d	1ab2a3c5d689	homer@uclm.es	30	2023	Ms	Ts	Ok(T1)	covers b.
e	1ab3c5d789	lisa@uclm.es	10	2023	MS	TS	Ok(Ts)	

4 Equivalence class table.

Property	Class valid inputs	Class invalid inputs	Heuristic
Id	>0 (1)	<= 0 (2) Nb number (3)	Range Boolean
Name	[5 - 255] char (4)	< 5 (5) >255 (6)	Fronche nº elements
Goodness Level	High (7) Low (8)	Anything else (9)	Set of valid inputs
MinAge	[0 - 80] (10)	<0 (11) >80 (12) No number (13).	Range Boolean.

ORDINARY 2022 - 2023

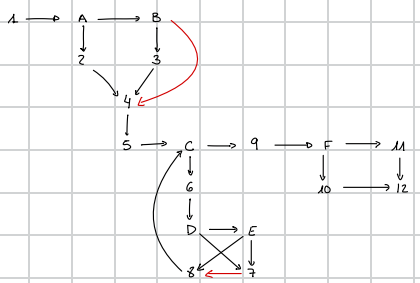
5

Description of scenarios

Scenario ID	Flow
Scen - 1	BF
Scen - 2	BF + AF1
Scen - 3	BF + AF2
Scen - 4	BF + AF3

TC - Id	Scenario	Clan	Nickname	Description	Hours	Output
TC1	Scen 1	Omatsaya	sully	hola	8	Sully, hola 8, Oma
TC2	Scen 2 / no clan		sully	hola	8	The clan is mandatory
TC3	Scen 2 / no name	Omatsully		hola	8	The name is ma
TC4	Scen 2 / no hours	Omatsaya	Sully	hola		The hours are
TC5	Scen 3	Omah	Sully	hola	8	
TC6	Scen 4 / Skill req	Oma	sully	hola	8	The skill ... not acquire.
	Scen 4 / ASO min score	11	21	41	11	The min...

6. Path Coverage



id	path
A	1AZ4 5C9F1012
B	1A245C9F1112
C	1AB45C6DE8C9F1012
D	1AB345C6DE78C9F...
E	1AB45C6D78C9F...
F	1AB45C6D78C9F...

PathID	AvatarSkills	ClanSkills	ClanMember	ClanMemberRequest	ClanMembe	ModelState	Return	Comment
A								
B								
C								
D								
E								
F								

FUNCTIONAL TESTING ~> Precondition: Funciona bien.

Select / Create / Detail.

- Filter { - Name, - Color } → Ambos funcionan
- RemovePurchaseItems → Funciona.
- NoDevicesAvailable. ~> Comprobar
- NoActivePurchaseButton → Funciona
- ErrorsInMandatoryData → Funciona ¿Falta algo?
- ModifySelectedDevices → Funciona
- MainFlow ~> Falta detail final.