

Engineering Software Intensive Systems

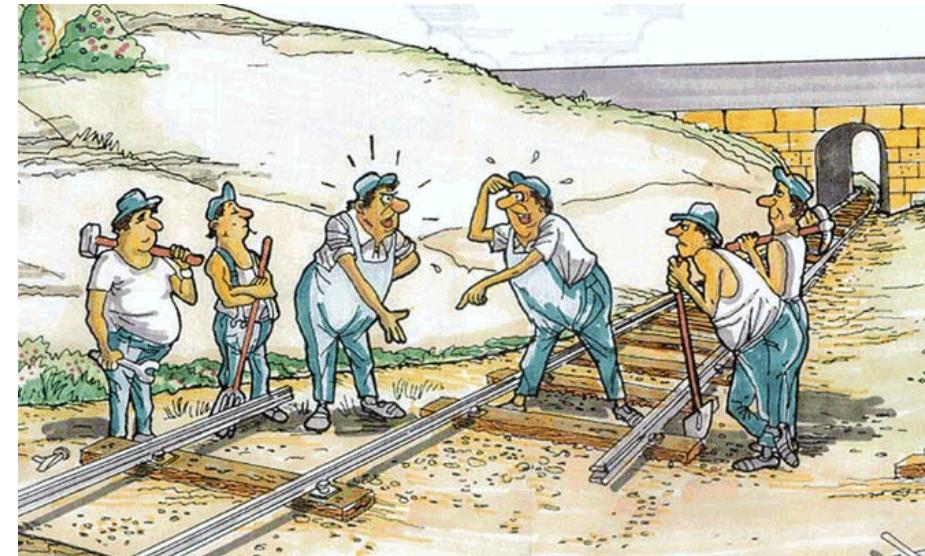
Testing

Lecture 13

26 June 2025

Slides created by
Prof Amir Tomer
tomera@cs.technion.ac.il

26 June 2025



Topics for Today

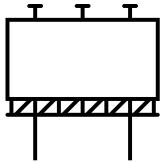
- Testing

Testing Levels for Software Products and Systems

- Three primary levels of testing during development

Unit Tests

- Performed on each unit during coding
- Repeat each time the unit changes



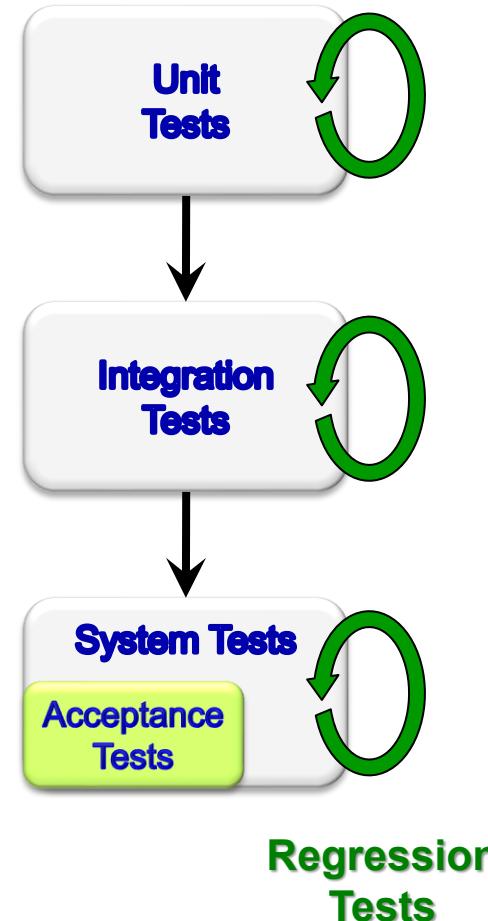
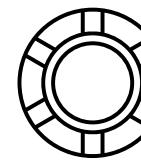
Integration Tests

- Performed on each sub-group of components after integration
- Repeat previous stage of integration tests to ensure next integration doesn't break stuff



System/Product Tests

- Different types of tests on the final product
- Repeat each time the product changes



Software Testing

“Software testing is a formal process that is performed by a trained testing staff in which individual software units, collections of software units, or complete systems are tested by running the software on computers. All tests are performed in accordance with approved test procedures on approved test cases.”

בדיקות תוכנה הוא תהליך פורמלי, המבוצע בידי צוות בדיקות מומחה, אשר במהלך הבדיקה, מספר יחידות תוכנה משולבות או מערכת תוכנה שלמה נבדקות באמצעות הרצת התוכנה על גבי מחשב. כל הבדיקות מבוצעות על פי נוהלי בדיקה (test procedures) מאושרים מעלה מקרי בדיקה (test cases) מאושרים.

Daniel Glin, Technion

- Testing goals
- Direct
 - Identify as many errors as possible in the tested software
 - Raise software quality to an acceptable level by fixing found faults and checking to ensure their absence
 - Perform required tests efficiently, effectively, on-time, and on-budget
- Indirect
 - Record observed software faults to use in future fault prevention efforts (corrective and preventive actions)

Ranking bugs by severity

Severity	Description
5 (Critical)	(1) Prevents performance of critical capabilities (2) Endangers safety, security, or other critical requirements
4	(1) Badly affects performance of critical capabilities. No known work-around. (2) Badly affects cost risks, project schedule, technical risks, or support for the project. No known work-around.
3	(1) Badly degrades performance of critical capabilities, but with known work-around (2) Badly affects cost risks, project schedule, technical risks, or support for the project, but with known work-around .
2	(1) Causes user/operator discomfort , but does not affect critical operational or mission capabilities (2) Causes development or support team discomfort , but does not prevent them fulfilling their tasks
1 (Minimal)	All other effects

<https://dilbert.com/strip/1995-11-13>



Seven Principles of Software Testing (Meyer, 2008)

1 Definition

To test a program is to try to make it fail

2 Tests versus specs

Tests are no substitute for specifications

3 Regression testing

Any failed execution must yield a test case, to remain a permanent part of the project's test suite.

4 Applying oracles

Determining success or failure of tests must be an automatic process.

5 Manual and automatic test cases

An effective testing process must include both manually and automatically produced test cases.

6 Empirical assessment of testing strategies

Evaluate any testing strategy, however attractive in principle, through objective assessment using explicit criteria in a reproducible testing process.

7 Assessment criteria

A testing strategy's most important property is the number of faults it uncovers as a function of time.

Testability

- Definition: The amount which a system or component allows the definition of testing criteria and the performance of tests that establish whether the criteria were met
- Implications
 1. What are the odds that a program will find a fault during testing (if one exists)?
 2. How easy is it to reach test coverage?
- Coverage criteria

Functional coverage

- Cover all program functions

Statement coverage

- Cover all program statements

Condition coverage

- Cover all decision points and branches in the code

Path coverage

- Cover all potential program paths

Entry/Exit coverage

- Cover all calls and returns in the program

Assuring testability: Control (J Bach, 2003)

“The better we can control the software, the more the testing can be automated and optimized”

- **Controlability:** Existence of an interface or mechanism to define test cases
 - Debuggers, Commercial testing tools
- Test engineers can directly control program state, hardware state, and system variables
- Objects, modules, and functional layers can be tested independently

Assuring testability: Observability (J Bach, 2003)

“What you see is what can be tested” or “You can’t test what you can’t see”

- **Observability:** Past states and historical values of system variables are obvious and queryable (e.g. logs)
- Different output created for each input
- System state and variables are visible or queryable at runtime
- All elements that affect output are visible
- Invalid output is easily identified
- Internal errors are automatically detected and reported via self-test

Assuring testability: Availability (J Bach, 2003)

“To test it, we have to get at it”

- Availability
- Software may have many faults (bugs add overhead and time to system analysis and testing reports)
- Bugs don't prevent the running of tests
- Product is developed in functional iterations
 - Allows development and testing in parallel
 - Already tested parts can help test others
- Access to source code

Assuring testability: Simplicity (J Bach, 2003)

“The simpler it is, the less there is to test”

- **Simplicity:** Design must preserve internal consistency
- Functional simplicity: System has the minimum number of features needed to meet all requirements
- Structural simplicity: Modules have tight cohesion and loose coupling
- Code simplicity: Code is not overly complex. An external reader can effectively read it

Assuring testability: Stability (J Bach, 2003)

“The fewer the changes, the fewer the disruptions to testing”

- Code doesn't change often
- Changes to code are controlled and public
- Changes to code don't prevent or invalidate automatic tests

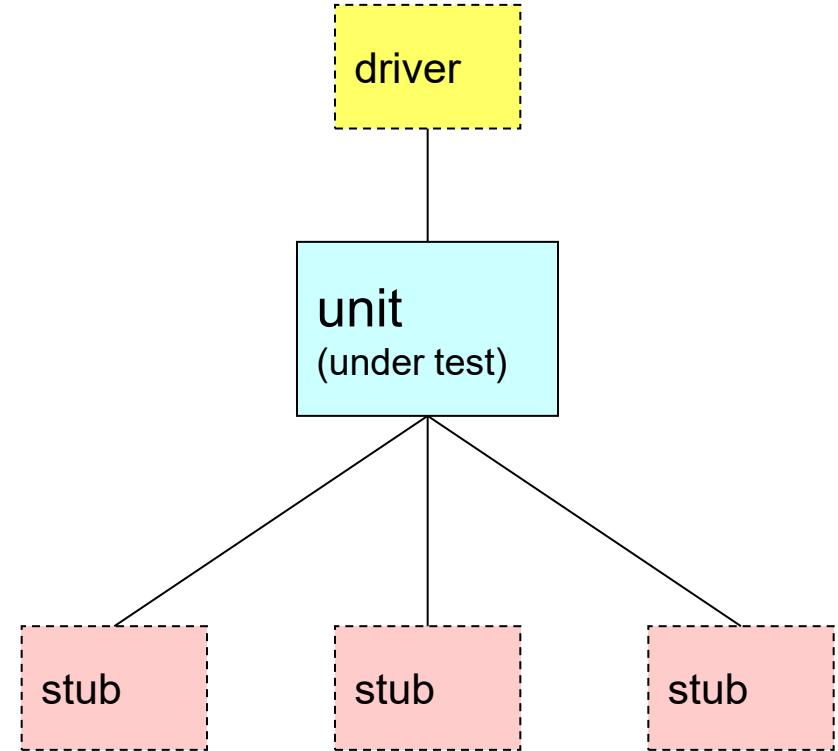
Assuring testability: Information (J Bach, 2003)

“The more information we have, the smarter we will test”

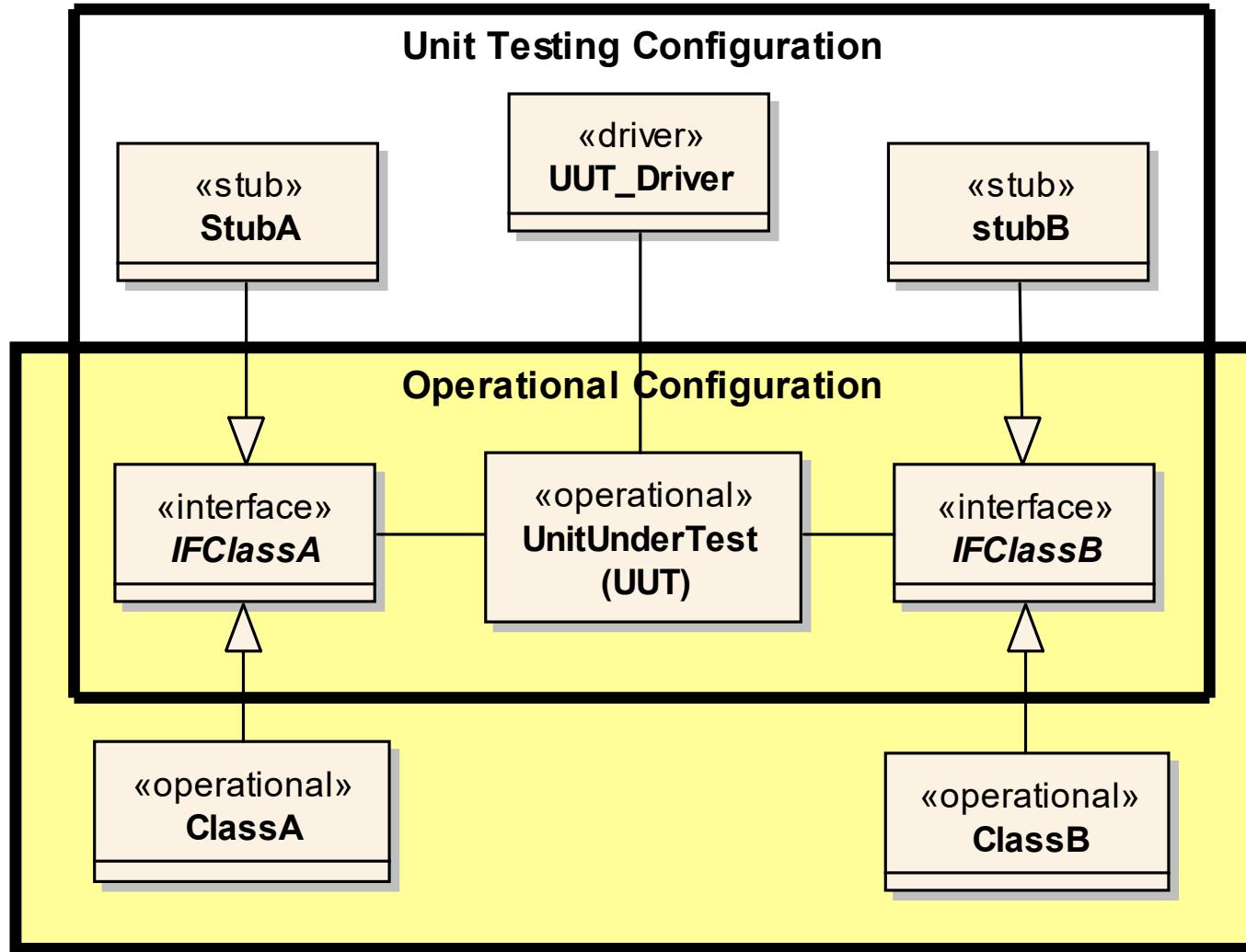
 Design is similar to existing products that are well understood	 Product is based on proven technology	 Dependencies on internal, external, and shared components are explicit and well understood	 Program's goals are well understood
 Intended users are well understood	 Program environment is well understood	 Documentation is available, accurate, organized, specific, and detailed	 Program requirements are well understood

Unit Tests

- **Unit**
 - A selection of code that can be compiled and tested independently
 - Often written by one programmer
- **Driver**
 - Simulated unit that operates the unit under test
- **Stub**
 - Simulated unit operated by another unit to test **the operator**



Unit testing environment for a class



Unit Testing Methods



Black box testing

- Check the operation of each unit in the system
- Ensure outputs and responses are OK
 - Legal inputs
 - Illegal inputs
- Response time

White box/glass box testing

- Check the internal structure and state of the unit
- Computational steps
- Computational correctness
- Correctness of logical decisions

Both methods require sets of test data to cover all test cases

Black box testing

- Covering a function with numerical inputs $f(X_1, \dots, X_n)$
- Test data must cover all equivalence classes of inputs for all arguments
- For each argument X
 - If the function must respond in a certain way in the interval $[L, U]$, check X with (at least) the following values
 - $X > U, X = U, L < X < U, X = L, X < L$
- Example: Function power(Range, Speed) computes power (High, Medium, Low) based on range and speed based on the following table

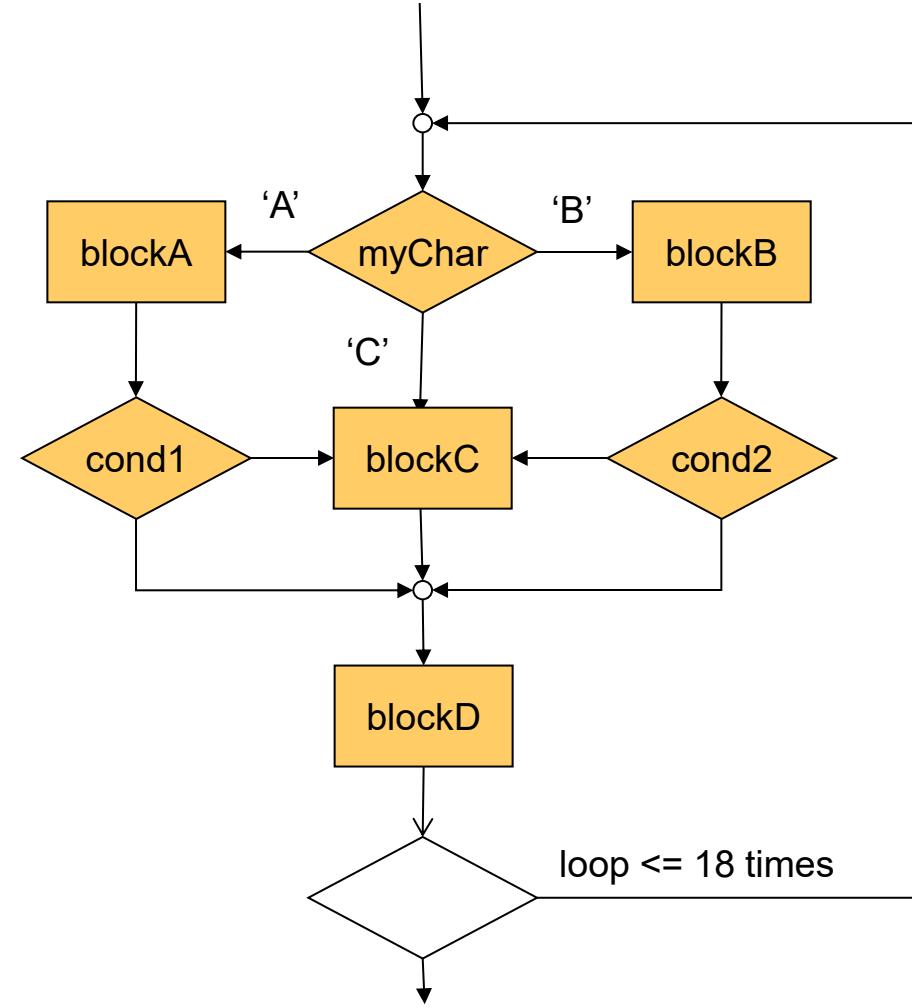
Speed \ Range	$R < 1$	$1 \geq R \geq 5$	$R > 5$
$S < 100$	L	L	M
$100 \leq S \leq 200$	L	M	H
$S > 200$	M	H	H

Unit Testing: Test Vector

R	S	Expected Result
0.5	50	L
0.5	100	L
0.5	150	L
0.5	200	L
0.5	250	M
1	50	L
1	100	M
1	150	M
1	200	M
1	250	H
...		

Unit Testing: White Box

```
read (kmax) //1 <= kmax <= 18
for (k=0; k < kmax; k++) do {
    read(myChar)
    switch (myChar){
        case 'A':
            blockA;
            if (cond1) blockC;
            break;
        case 'B':
            blockB;
            if (cond2) blockC;
            break;
        case 'C':
            blockC;
            break;
    }
    blockD
}
```



$$\text{Possible path count: } 5^1 + 5^2 + \dots + 5^{18} = 4.77 \times 10^{12}$$

Base path testing (McCabe, 1976)

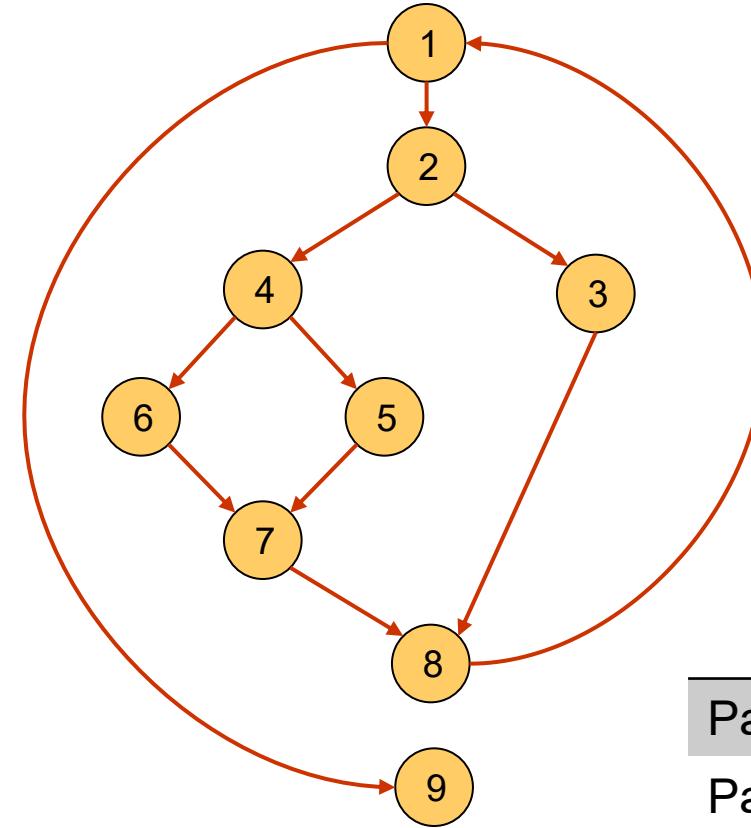
- Classic white box testing method: Cover all potential transitions in the unit
- Create a control flow graph of the unit
 - Node: A computational unit without branching
 - Edge: A computational branch
- Independent paths: Path from start to finish that includes at least one new node compared to other paths

Base path testing example

PDL procedure

```
1: do while records remain
   read record;
2:  if record field 1 = 0
      then process record;
         store in buffer;
         increment counter;
   else
4:   if record field 2 = 0
5:     then reset counter;
6:   else process record;
      store in file;
7:   endif
   endif
8: enddo
9: end
```

Flow graph



Base paths

Path 1	1-9
Path 2	1-2-3-8-1-9
Path 3	1-2-4-5-7-8-1-9
Path 4	1-2-4-6-7-8-1-9

Cyclomatic Complexity – $V(G)$ of graph G

- Number of base paths is small and finite
- Maximum number of independent paths is:

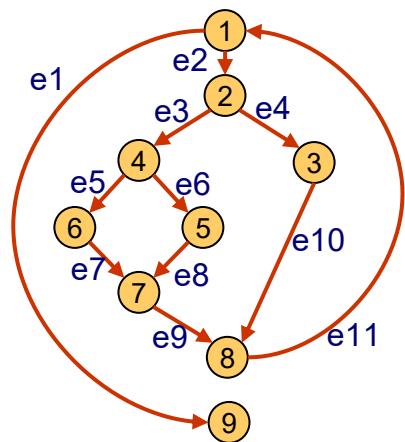
$$V(G) = E - N + 2$$

$$= P + 1$$

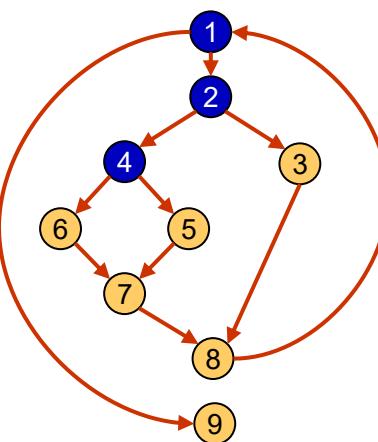
E = #edges, N = #nodes

P = number of nodes with more than one exiting edge - decision points(for graphs with max 2 exits per node)

$$= R$$

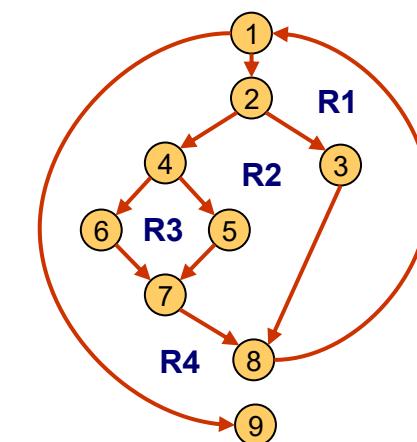


$$11 - 9 + 2$$



$$3+1$$

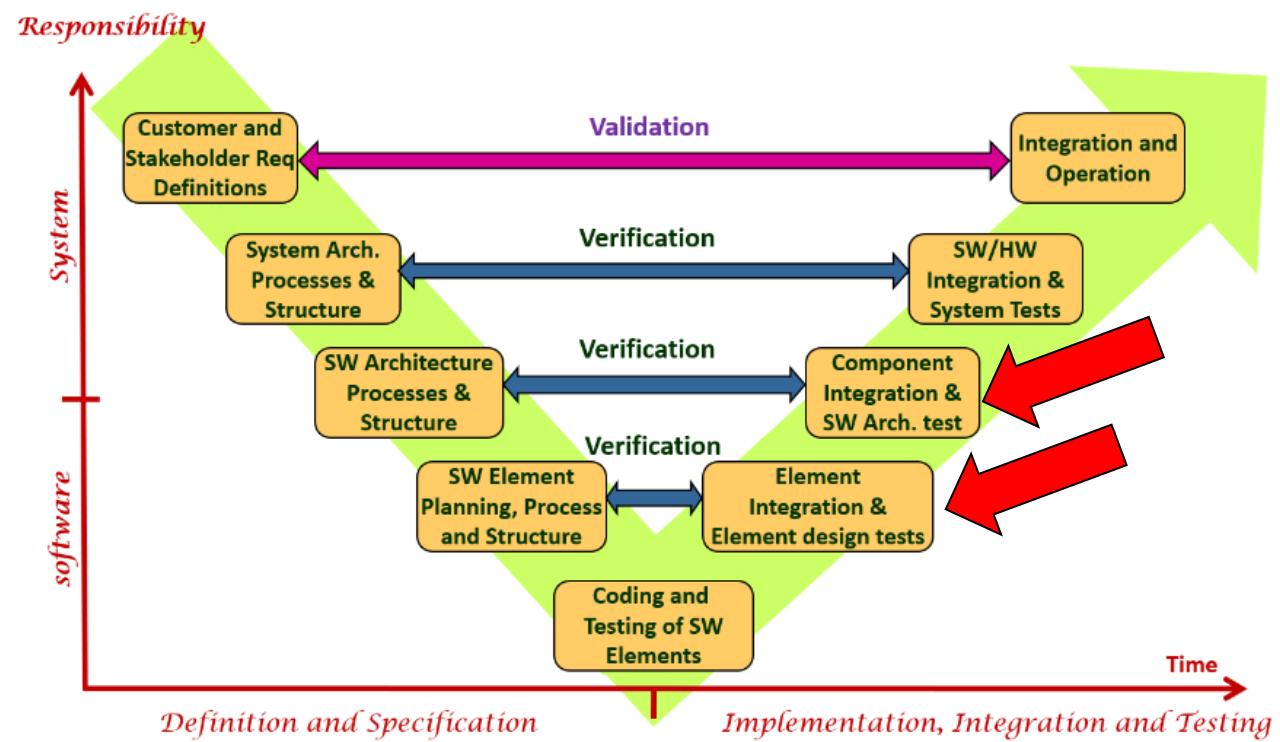
R = number of closed regions + the open region



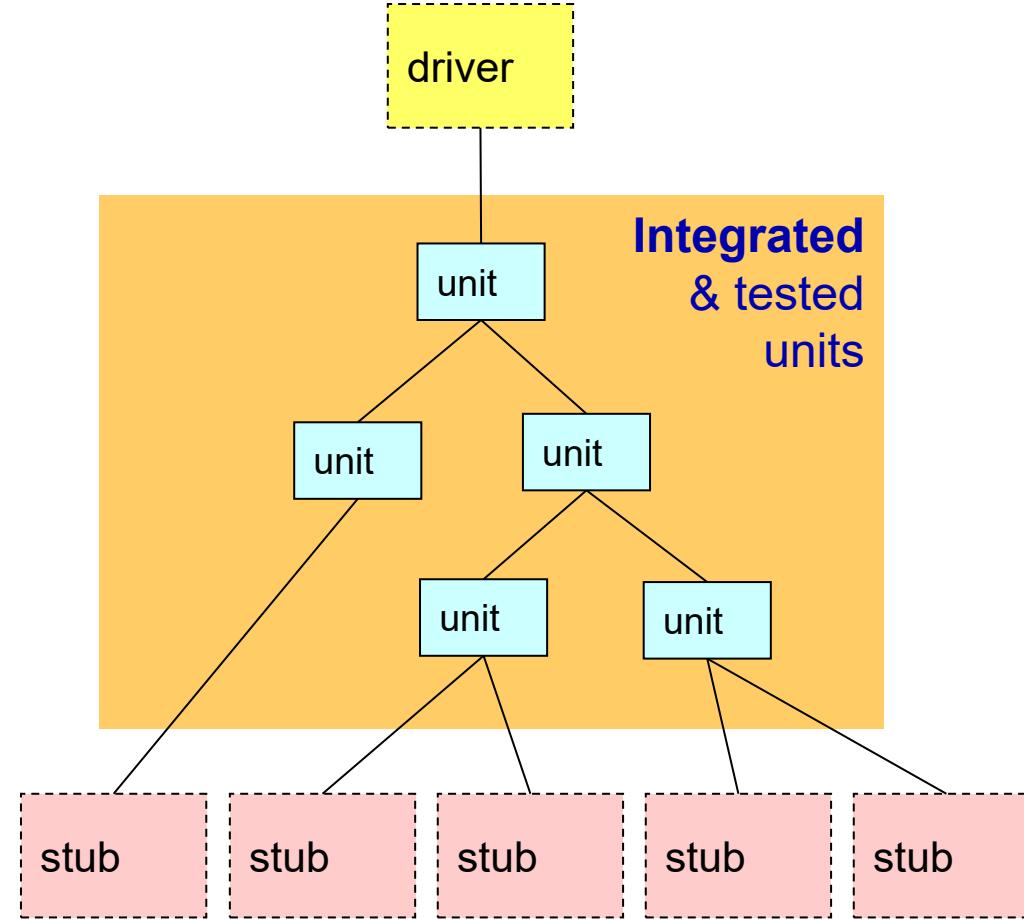
$$4$$

Integration Testing

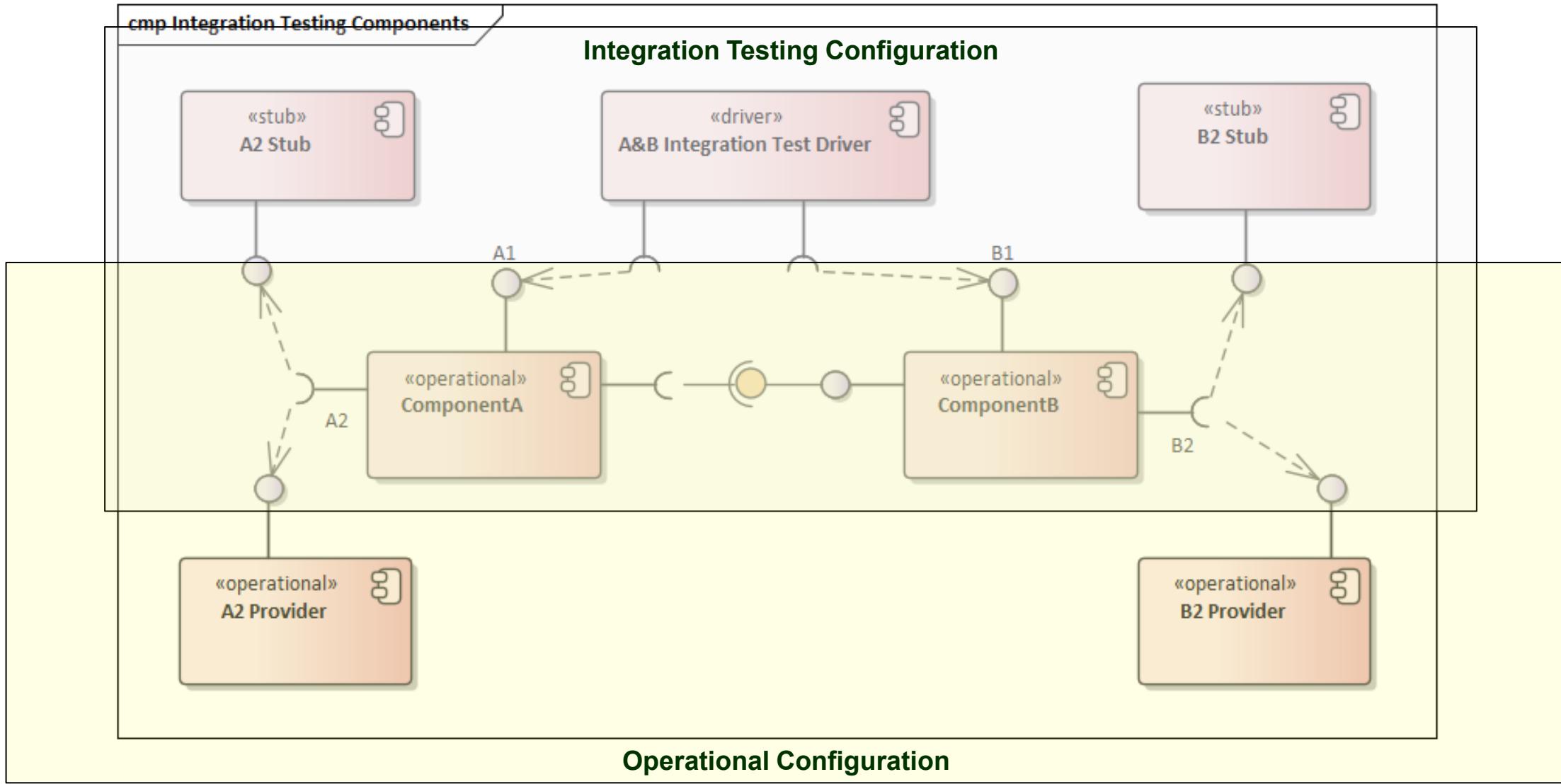
- Our goals:
 - Create components (applications) that work and are tested
- Inputs:
 - Tested software units
- Outputs:
 - Tested software components



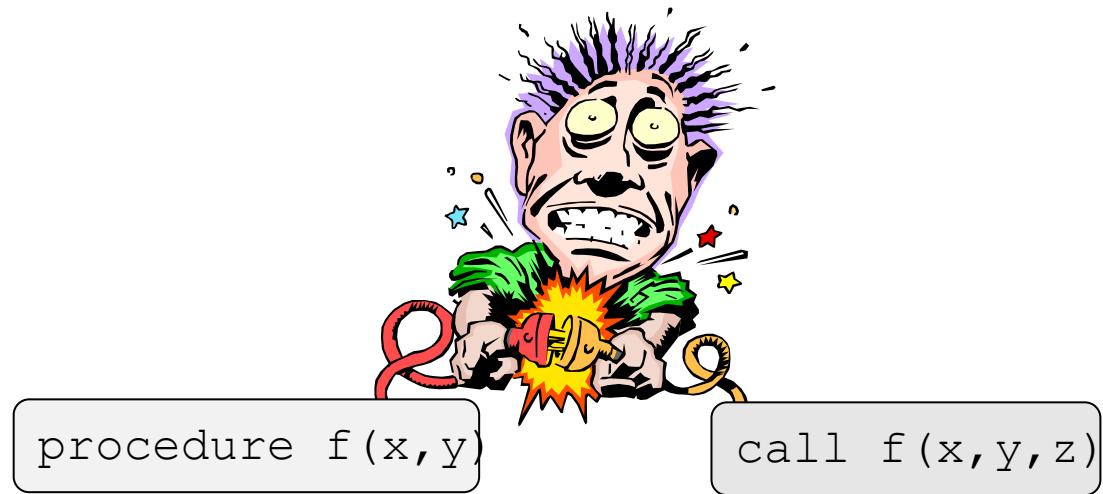
Integration Testing Environment



Component integration testing environment (UML)



Managing Integration



See you at integration!

- Making integration work
 - Proper planning of integration steps
 - Manage interfaces throughout the development process
 - Check units early on (before final integration)

Continuous Integration – Test Early, Test Often, Avoid surprises

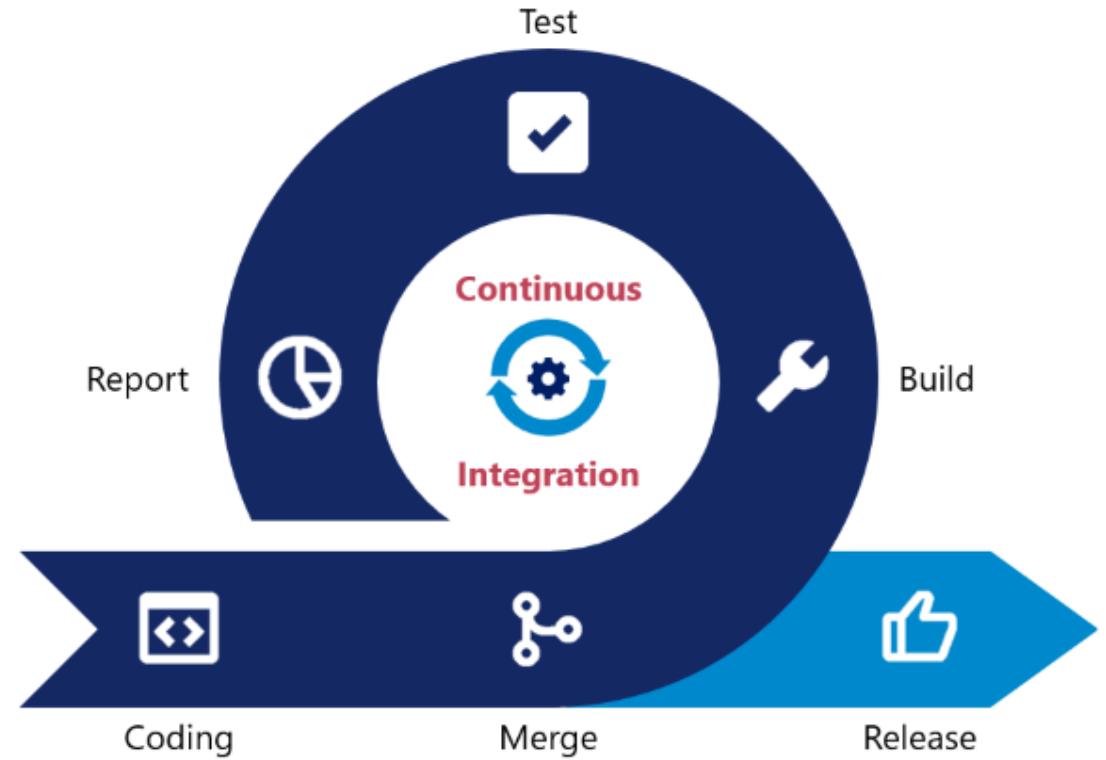
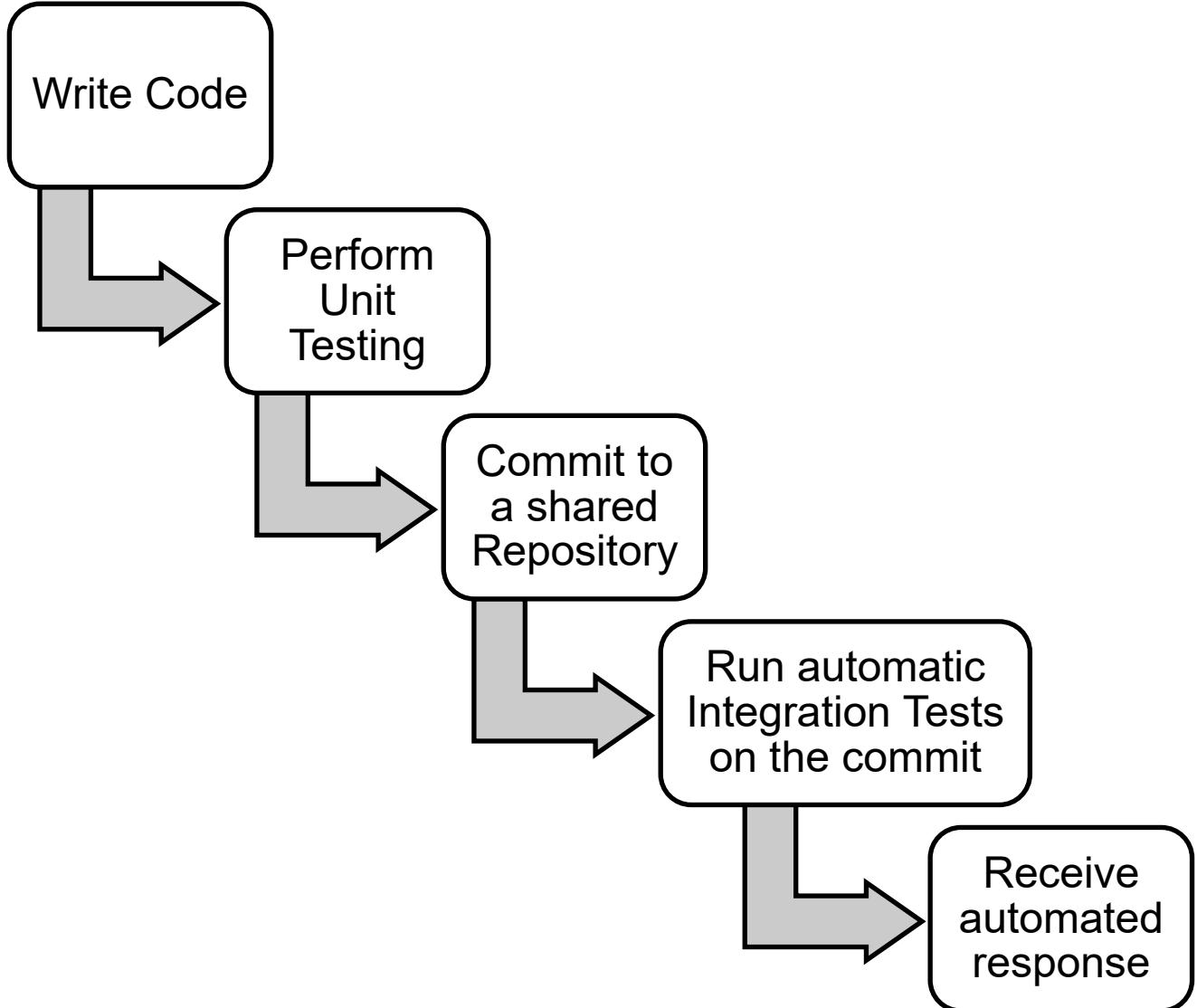


Image © DevopsWithDeepss
Source: <https://techwithdeepss.hashnode.dev/supercharge-your-software-development-with-effective-ci-cd-tools>

Sample Continuous Integration Tools (there are more)

BitBucket Pipeline

- Cloud solution
- On top of BitBucket source code management



Jenkins

- On premises
- Robust and veteran tool



AWS Code Pipeline

- Cloud solution
- Integrates with Amazon Web Services



CircleCI

- Pairs with GitHub
- Can be on-premises or cloud



Azure Pipelines

- Integrates with MS Azure Cloud
- Supports GitHub and Containers



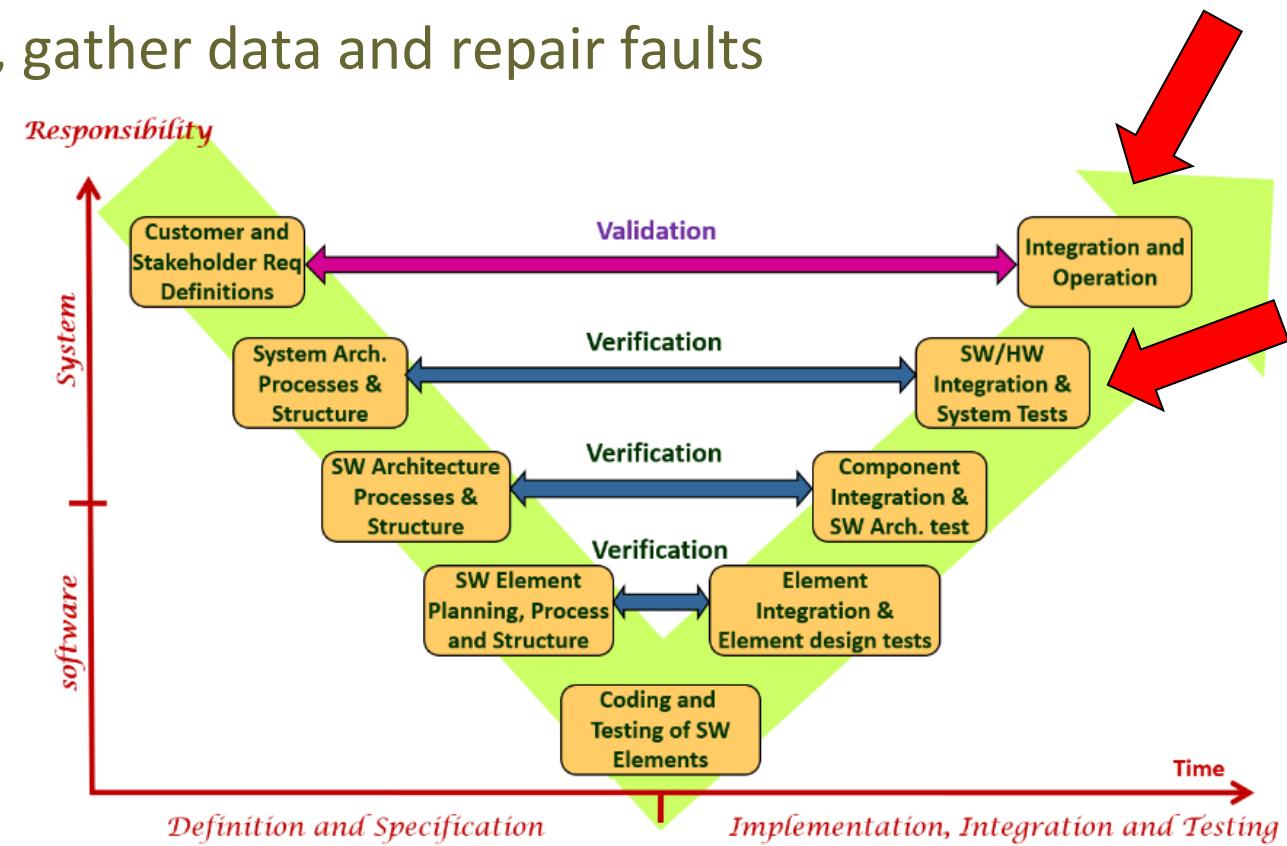
GitLab

- Integrates with GitLab source control
- Can be on-premises or cloud



Integration, Verification, Validation

- Our goals:
 - Create a system that meets acceptance tests that the customer agreed to
 - Operate a system in its environment, gather data and repair faults
- Inputs:
 - Tested software, tested hardware
- Outputs:
 - Tested and working system



Validation and Verification: Proving the product

Verification

Did we build the
system correctly?

Validation

Did we build the
correct system?

Validation and Verification: Proving the product

Prove the developed product

- Meets its requirements – verification
- Fulfills its goals in the production environment - validation

Example: ATM

- Requirements
 - The user can withdraw cash only if the account balance is greater than or equal to the withdrawal amount
 - If the user repeatedly enters the wrong PIN, the card will not be returned
- Goals
 - Offer online banking services to 200,000 customers a day, without violating usage rule, and while preserving data completeness and correctness

Proving a product

Tests, Field Tests

- Operate the product or part of the product to examine its actual performance
- E.g., Test the ATM in its defined use cases

Prototype, Demo

- Build part of the solution to test its expected behavior
- E.g., Build demo screens and transition logic for an ATM on a PC

Review

- Examine the specification of the offered solution
- E.g., Read requirements, use cases, and compare to operational requirements

Simulation

- Build a model of how the solution behaves
- E.g., Simulate space flight via the navigation and flight algorithms in the space craft code

Analysis

- Theoretical proof of correctness
- E.g., Build a state machine that shows undesirable states are not reachable

Conclusion

- Testing