UNIVERSITY OF MISKOLC
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATICS

Software Technology Lab

GEIAL316-B2

Software testing levels

Tamas Tompa, PhD
assistant professor
Department of Information Technology

Miskolc, 2026

Software testing levels

- Acceptance test
- System test
- Integration test
- Component /unit test

Reasons

- Testing **after the programming task, in the past**
  - **only testing of single functions**
    - troubleshooting **during** system operation
      - serious **overtime**
  - → creation of lower-level testing techniques

- Reasons:
  - the highest level user interface of the system and the **processes there** resulting complex operations, thus **the testing of them** is also **a complex task**

Reasons

- Often, a **reused component can cause incorrect operation** in any functions and sub-functions
  - **after modifying a unit, high-level test cases fail** for seemingly incomprehensible reasons
    - a long analysis may be necessary to find the reasons

- **The incorrect operation is discovered in** the late development phase
  - each unit or component must be developed before it can be tested

Reasons

- User **interface tests** often **only indicate during runtime that** the interface or structure of the given **functionality has changed**
  - on the other hand, at a lower level, changes to the interface of a given unit may already be detected during compile
  - an IDE should be able to immediately indicate this kind of change

- Using **lower-level testing techniques**
  - → can be decrease the probability of tests failures of user processes

Component test

- **test of the smallest, independently operating units of the system**

- **a component test by itself**
  - usually the source code know (white box)
  - **in the O.O. environment**
    - classes
    - methods
  - **in the procedural environment**
    - procedures
    - functions

Component test

- **Types**
  - **Unit test**
    - testing the methods (small units)
    - the method returns the value of the given parameters and then checks whether the actual return value is the same as the expected one
  - **Module test**
    - testing non-functional properties
      - speed
      - is there a memory leak (memory lake)
      - is there a bottleneck

Unit testing rules

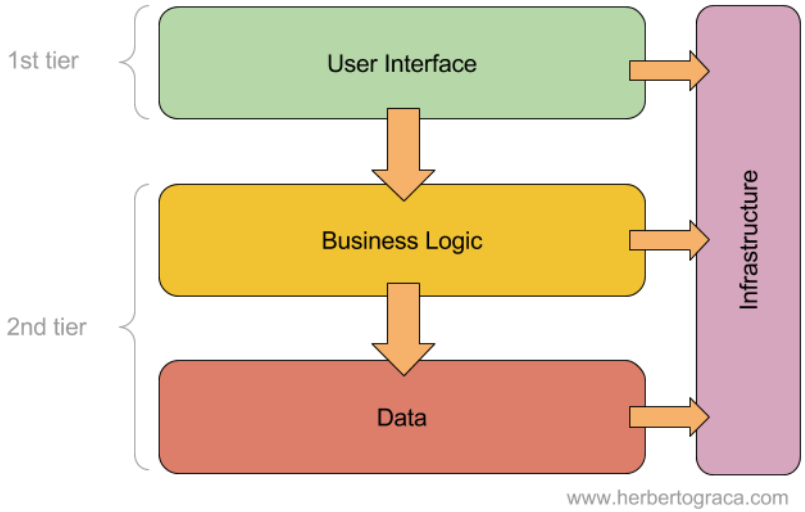
- **The following samples, suggestions correctly usage is mandatory!**
  - **An unit test tests exactly a program unit**
    - What cases does a class work by itself?
      - almost never...
        - why not?
          - objects, and their properties
      - additional problem: if a program unit does not implement a single task/function
        - Why is it problem?
        - see later: SOLID principles

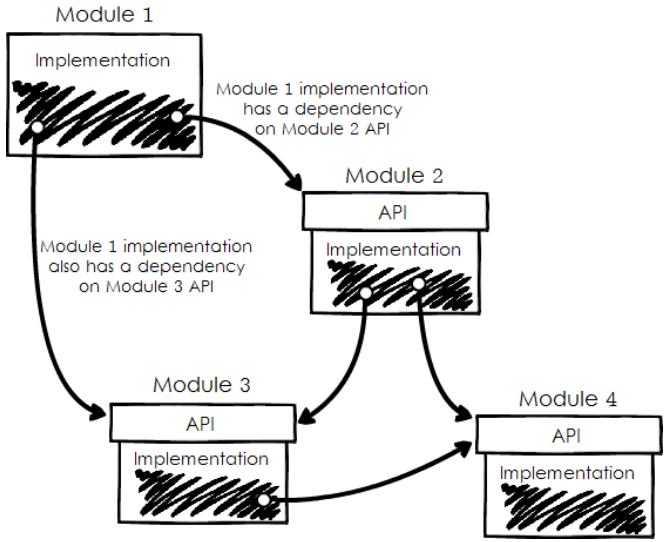
Source: https://herbertograca.files.wordpress.com/2017/07/1980s-90s-layered-architecture.png

https://devopedia.org/images/article/23/8937.1530525110.png

Unit testing rules

- The following samples and suggestions correctly usage is mandatory!
  - **it is difficult to imagine an architecture that does not build on other layers or other tools**
    - layered architecture, anti-pattern: lasagna architec.
  - **→ can be dependencies and it will occur**
    - their behavior must also be controlled





Unit testing rules

- **Unit tests do not cross the boundaries of their own modules**
  - if occur a program unit that **uses a service from another module**, then need to be emulate the behaviour of service in the other module

- **Unit tests run independently of each other**
  - each test has its own prerequisites and end conditions
    - **prerequisite**
      - must be available before running
      - defined by a programmer
    - **end condition**
      - after the running

Unit testing rules

- **Unit tests run independently of each other**
  - no test case can expect another test case to produce the desired preconditions for it
- **Unit tests work independently of their runtime environment**
  - if want to connect to external database or external system, then permanent configuration and/or emulation would be required regarding external connections

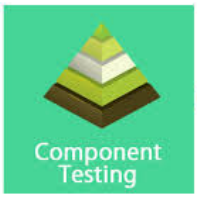
Unit testing rules

- **Unit tests have no side effects**
  - after running, there is no changes in the runtime environment, especially to global variables need to pay attention

- **Have to implement unit tests that way they can be run at compile**
  - tests which are environment-independent can run efficiently, both in the compiler and the test framework can run anywhere, stopping compilation if any test fails

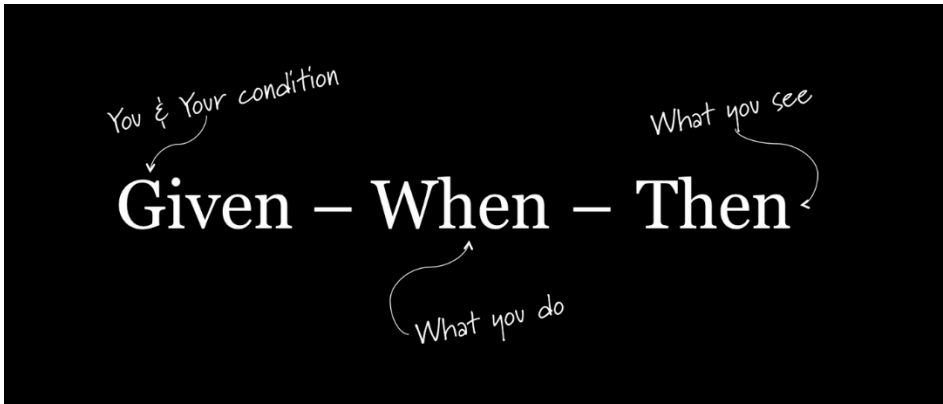
Unit testing rules

- → unit tests **must follow strict rules**
  - **stubbing** and **mocking** techniques help to follow the rules
  - in practice, the type or level of the test will often be defined by which of these rules are followed and to what extent



Unit Test – GivenWhenThen Pattern

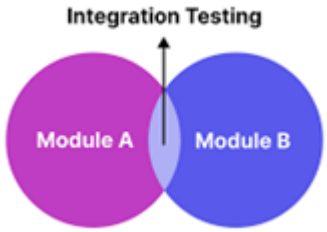
- **G**iven, **W**hen, **T**hen (**A**rrange, **A**ct, **A**ssert)
  - general structure of unit tests
- Given
  - initialization of values and data which needed for the given test case
- When
  - invoke the tested method
- Then
  - check the behavior



Source:
https://miro.medium.com/v2/resize:fit:1100/format:webp/1*o-_lILSt-z5Uwi_Antc3VQ.png

Unit test – GWT sample

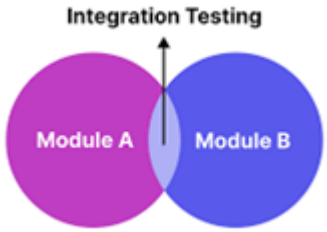

```
1. // given / arrange
2. double r1 = 1.0;
3. double area1 = r1*r1*Math.PI;Circle
4. c1 = newCircle(newPoint(0.0,0.0),r1);
5.
6. // when / act
7. double resultArea = c1.Area();
8.
9. // then / assert
10. Assert.AreEqual(area1,resultArea,0.0001);
```



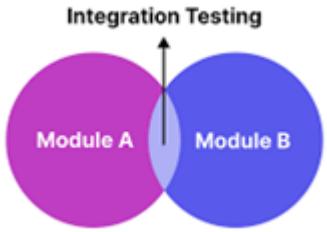
Integration test

- Testing program units, **modules, and their interaction with each other and the environment**
  - testing real operation of classes, **interfaces between comopnents**, **services** and their dependencies, including data access
  - **searches for errors that occur during assembly the components**
    - since the different parts of software were developed by different programmers and teams, errors can occur due to insufficient communication
      - For example, one programmer assumes something (e.g. the method only accepts positive numbers in its parameters) that the other does not know about it
        - avoidance → with contract-based design (design by contract)

Integration test



- worth as soon as possible to execute
  - the greater the degree of integration, the more difficult it is to determine where the detected error (usually a runtime error) comes from
  - otherwise, when all components are developed and test only after it ("big bang test"), it is extremely risky

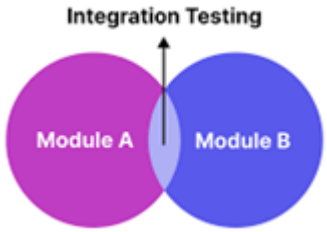


Integration test

- **examining the operation of services, logics, and modules together**
  - may be used to write tests
    - in which each program unit performs the real operation
    - in which the operation must be emulated

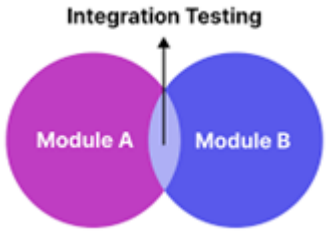
- **Unit test**
  - **→ well-functioning logic**
- **Integration test**
  - **persistent operation of logic**

Integration test - phases



- After unit tests
  - it is ensured that the components themselves are already working correctly

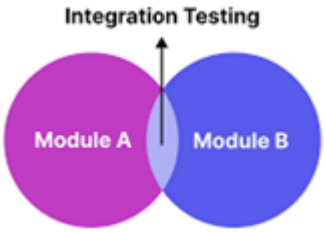
- **Phases**
  - **technical integration test (Integration Level Testing, ILT)**
    - examination of cooperating units
    - assembling a subsystem from already tested components
    - primarily aim to verification → finding errors



Integration test - phases

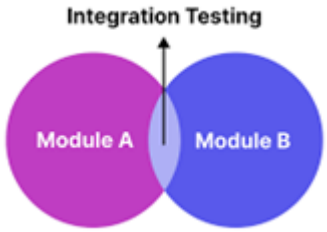
- **System Level Testing (SLT)**
  - comprehensive testing of all components of the system (functional, non-functional)
  - Can the system be released to the customer?
  - twofold goal:
    - Verification: finding errors in the system that have not been revealed during previous testing process
    - Validation: mainly by testing non-functional requirements to ensure that the system functions according to the user's goals
  - Several aspects
    - Service testing
    - Quantitative testing
    - Stress testing
    - Usability testing

Integration test - considerations

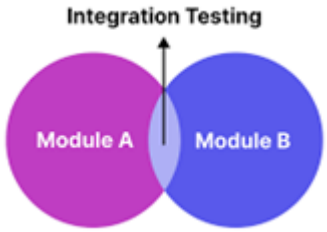


- **Quantitative test**
  - testing with huge amounts of data
  - Does the huge of data cause malfunctions?
- **Stress test**
  - exposing the system to a heavy load
  - checking time of responses
- **Usability test**
  - used by a specific user, within a specific scope of use
  - to effectively achieve specific goals, how satisfactory it is and how much it leads to satisfaction

Integration test - considerations

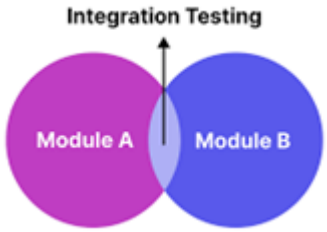


- **Safety test**
  - errors examination related to data security and data protection
- **Configuration test**
  - different environments (hardware, operating system) system, other software installations)
- **Documentation test**
  - user and development documents
  - checking of completeness
  - examples mapping to test cases then execution



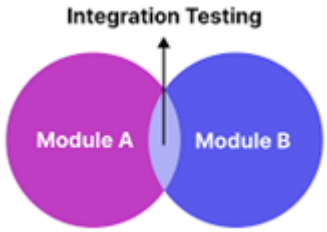
Integration test - phases

- **User Acceptance Testing (UAT)**
  - Can the system be put into production?
  - examining the satisfaction of the user and all stakeholders
  - must be execute at the final environment and the final operating conditions



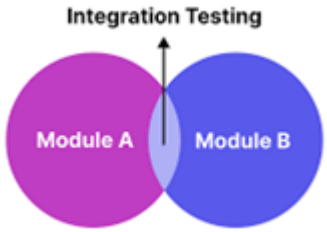
Integration test - types

- **Types**
  - **component – component integration test**
    - after component test, test of interactions between components
  - **system – system integration test**
    - after the system test, test of interactions between the system and other systems



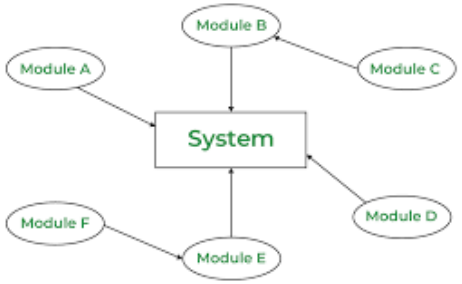
Integration strategies

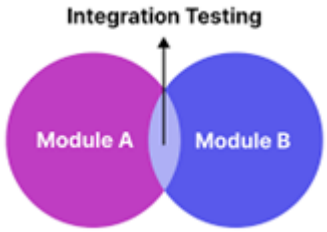
- **Strategies for building subsystems** and designing and running test cases
  - **"Bing Bang"**
  - **Incremental**
  - **Top-down**
  - **Bottom-up**



Integration strategies

- **"Bing-bang" integration**
  - **each unit of the system are available** and **building the entire system from these units**
  - Advantage
    - unit test can be more easily derived from the analysis requirements
    - a few code need to write
  - Disadvantage
    - it is very difficult to find the cause of errors
      - a single error can be caused by a combination of several errors





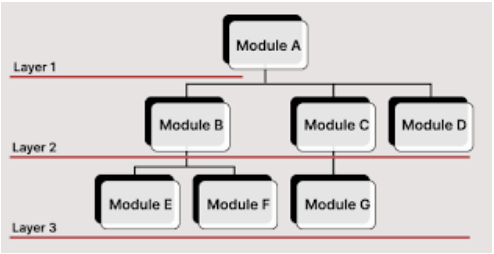
Integration strategies

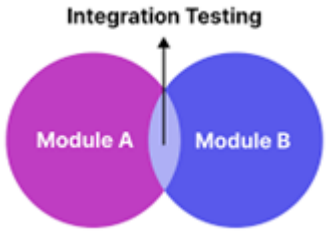
- **Incremental integration**
  - **gradually integrate the elements of the system**
  - perform test every level of integration
  - Steps:
    - 1. **combines some elements** (modules or subsystems)
    - 2. runs tests that **only require the combined elements**

3. if all **tests are successful, adds new elements** to the system

4. designs additional **test cases** that also require the existence of the new elements

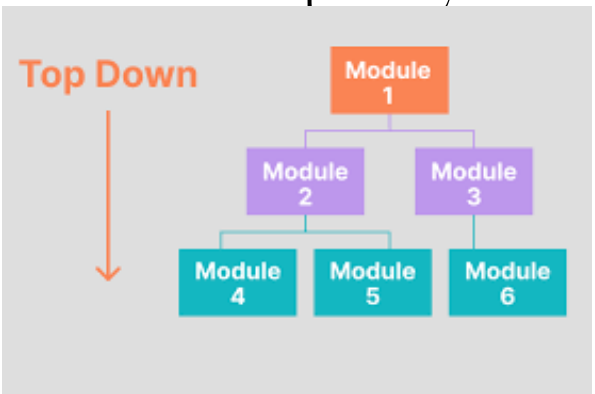
5. reruns **all previous**

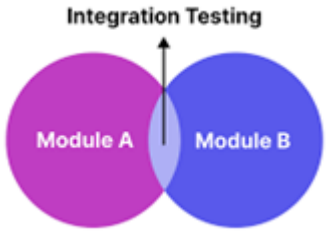




Integration strategies

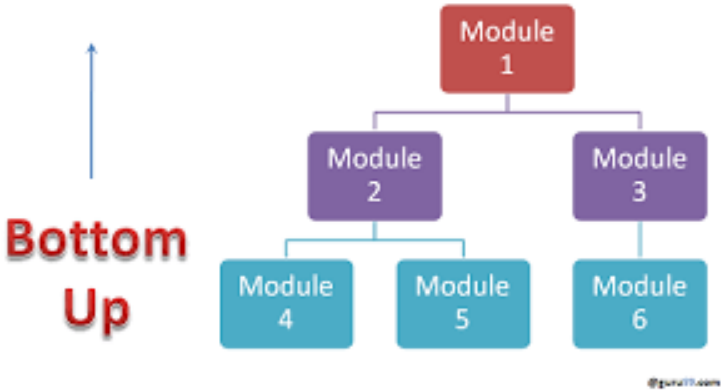
- **Top-down integration**
  - 1. **starting the test with the element at the top** of the hierarchy
  - 2. temporary elements (stubs) **simulating the behavior and interface of the elements** one level lower are **needed**
    3. if the test is successful, **replace the temporary elements with the real ones**, and simulates the ones they use with new temporary elements

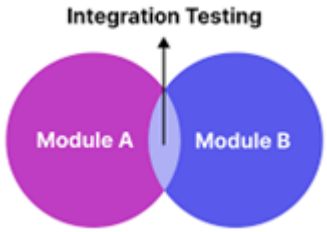




Integration strategies

- **Bottom-up integration**
  - 1. first **testing the modules at the lowest level, then move up** the hierarchy
    2. for this, need to write a testing environment (test driver) that simulates the upper levels
  - 3. if the **test is successful, replace the test drivers with the real implemented elements** and replaces the next level with test drivers

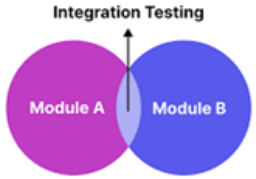




System integration test

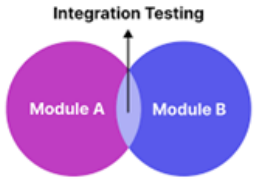
- Corporate environment
  - → **several physically separate subsystems are required to connect**
- Testing these systems involves a special type of integration testing , **with system integration test**
- treating the interfaces of **external systems as 3rd party module dependencies** and hide them behind a own interface
  - stubbing and mocking techniques help to emulating external systems

Integration test - emulation of dependencies

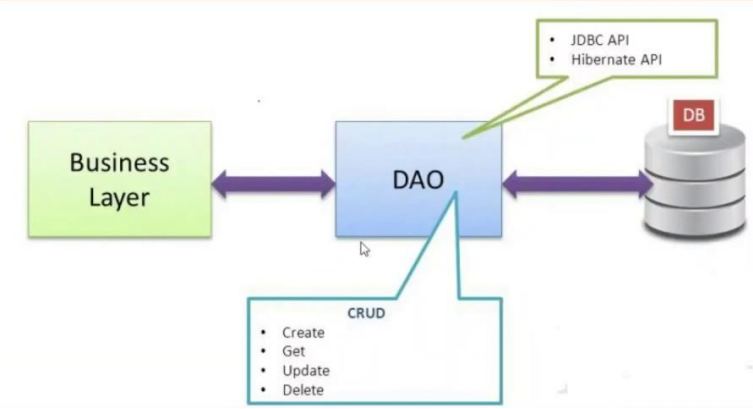


- **replacing the real instance of dependencies with another instance** that we implement or configure with the same interface
  - **replacing external dependencies for testing individual components in isolation**
  - a. eliminate possible errors of existing dependencies
  - b. eliminate incomplete or non-existent implementations
    - e.g.: testing a function that requires login, but there is no login (and no user) yet
- In case of unit tests
  - all dependencies must be emulated
- In case of integration tests
  - the readiness of the test case and dependencies determines whether we use some emulation technique or not not

Integration test - emulation of dependencies

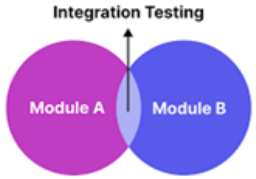


- Can only be emulate own type!
  - every 3rd **party tool** , API **must be hidden behind own interface**, then use an implementation of it as a dependency when implementing your own logicwn logic



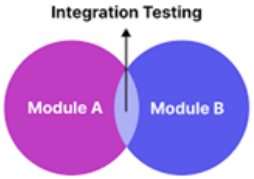
- E.g. Data access layer emulation
  - **Creation of DAO (Data Access Object) layer →** hiding database operations (CRUD)

Integration test - emulation of dependencies



- **Stubbing**
  - **provides a predefined response to a given call without actually executing the underlying logic**
  - **we create our own type implementing the interface of the given dependency, and replace the original implementation with the stub instance we created**
    - stub behavior is displayed transparently for the tested program unit
  - can be implement any test, result, or behavior

Integration test - emulation of dependencies



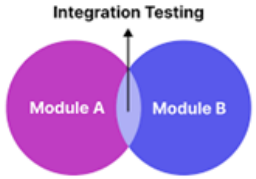
- **Stubbing**

```
public String getUserName (int userId) {
return database .getUserById (userId) .getName ();
}
```

```
UserRepository repoStub = mock(UserRepository. class );
when (repoStub.getUserById( 1 )).thenReturn( new User( 1 , "Test Elek"
assertEquals( "Test Elek" , repoStub.getUserById( 1 ).getName());
```

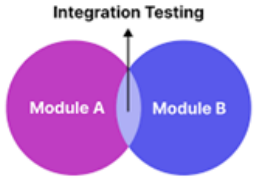
- query a user from the database:
  - but the database is slow or not ready yet, so we replace it with a stub
- it doesn't matter whether you actually queried from the database, it returns the expected response to the given request

Integration test - emulation of dependencies



- **Mocking**
  - evolved from stubbing
  - not only can you give responses, but **can be also monitor how to be used**
  - **it sets special requirements for specific method calls and defines what results are returned by the given method for certain parameters**
    - the focus is on the parameter data and the return result

Integration test - emulation of dependencies



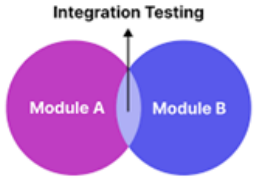
- **Mocking**

```
public class NotificationService {
    private EmailService emailService;

public NotificationService (EmailService emailService) {
        this .emailService = emailService;
}

public void sendWelcomeEmail (String email) {
( email ," Welcome ! " , " Thank you for registering ! " ) ;
}
}
```

```
EmailService mockEmailService = mock ( EmailService.class ) ;
NotificationService service = new NotificationService ( mockEmailService ) ;
service.sendWelcomeEmail ( "test@example.com" ) ;
verify ( mockEmailService ) . sendEmail ( " test@example.com " , " Welcome ! " , " Thank you for registering
```

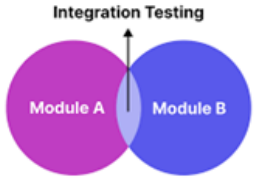
- class that sends notifications via an email service
- whether emailService.sendEmail was actually invoked with the correct parameters
- here it is not only the result is important, but also whether **the method actually ran and was called correctly**

Integration test - emulation of dependencies



- **Mocking**
  - frameworks
    - e.g. **Mockito**
  - mocking frameworks also provide additional tools
    - can be also set behavioral expectations using the framework's toolset
      - e.g. whether the method was invoked with certain parameters or not
      - how many times was it invoked (verification)
      - etc.

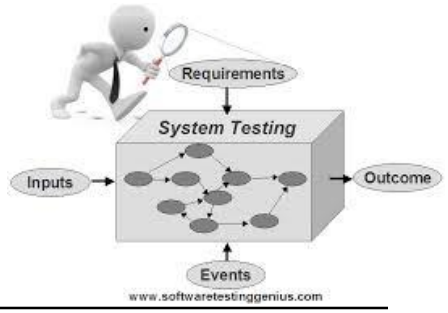
Integration test - emulation of dependencies



- **Stubbing vs. Mocking**

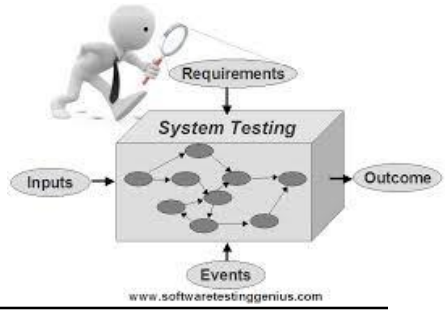
	Stubbing	Mocking
Purpose	predefined response	also observes how invoked
Checking the method invokes?	not	yes
Use case	replacing a response of an external system	how many times and how a function was invoked

System test



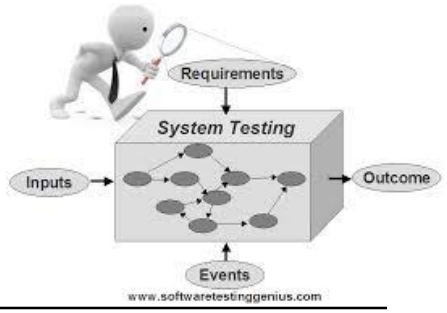
- One level up
- **Testing a complete integrated system**
  - already **ready software product tests** that is it suitable:
    - for the requirement specification
    - for the functional specification
    - for the system plan
- There is no emulation technique
- Testing the entire system in an environment where it will work
- Often black box testing
  - performed by an external company
    - Developers - Testers communication with bug tracking system

System test



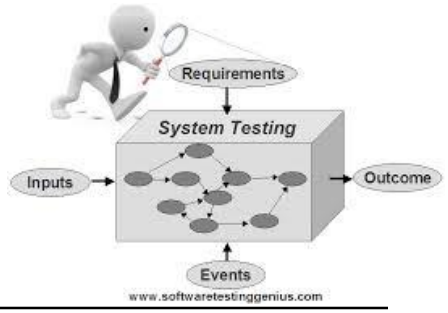
- **Features**
  - Through a user interface (GUI)
    - Controls
    - Screen states
    - Messages, error messages
    - Consistency
    - Processes and their states
  - Input and output files
    - CSV
    - XML
    - PDF
    - Reports

System test



- Based on test reports
  - is the system in a state that it can **be handed over to the customer or not?**
  - all functional requirements been tested?
  - discusses all functional needs and contains known bugs
  - **decisions are made** based on this document

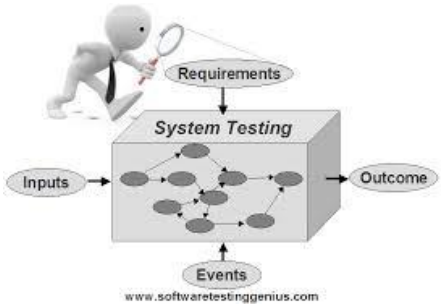
System testing - automation



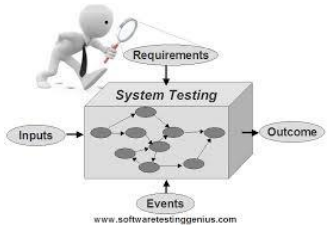
- Test cases run automatically
- Depending on the type of the system
  - running tests manually
  - and automated execution in certain parts of the system

- Advantages
  - execution of tests is fast and efficient
  - cost-effective in long term
  - more interesting than manually filling forms over and over again
  - the results are immediate and easily delivered to interested partners

System testing - automation



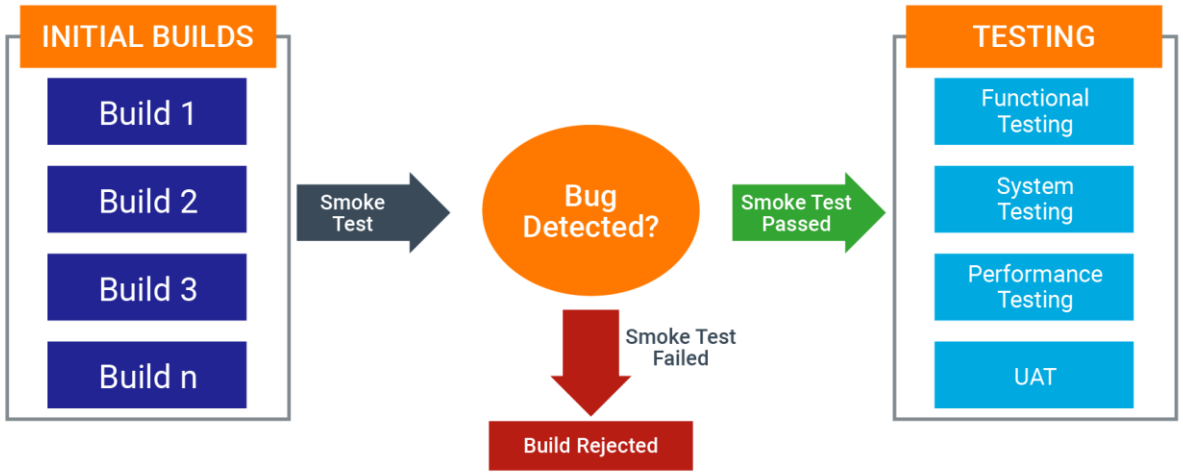
- Disadvantages
  - expensive tools
  - testing is initially less efficient due to the development of test scripts
  - maintaining tests may can require long time
  - test tools have limitations



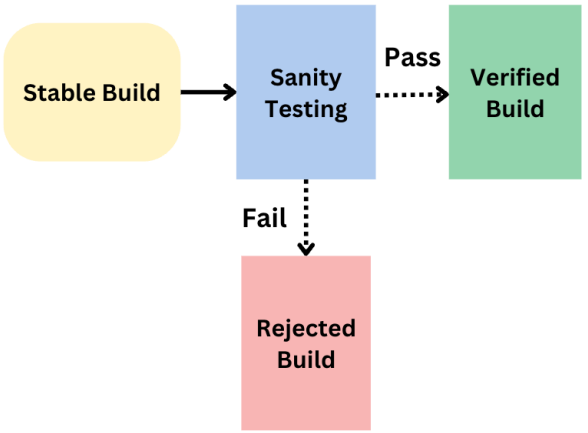
System test – special types

- Special system tests
  - rapid testing methods
  - is the software is working in a given state or not

Smoke -body



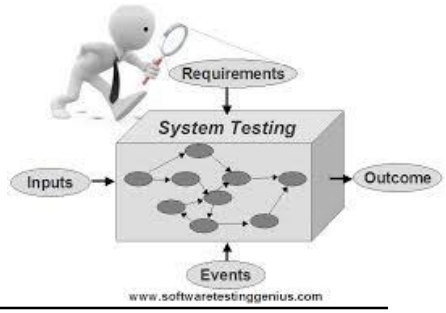
Sanity test



Source: https://www.3pillarglobal.com/insights/blog/what-is-smoke-testing-in-software-qa-testing/

https://www.browserstack.com/guide/sanity-testing

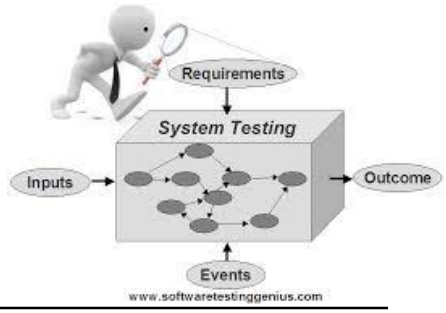
System test – Smoke test



- **Originating from the hardware world**
- **Does it smoke or not after turning it on?**
  - **does it work at all?**
  - if even the basic functions don't work, there's no reason to testing further

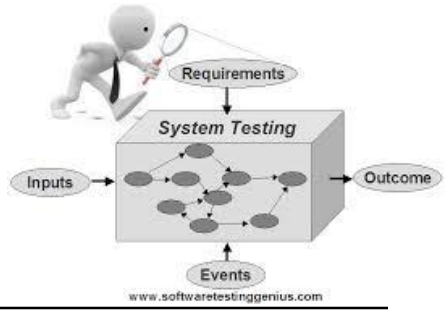
- In software testing, it means a collection of wide and shallow tests
  - Wide: affecting all functionality of the system
  - Shallow: testing of some functionalities with only minimal effort

System test – Smoke test



- Purpose
  - e.g. searching database schema inaccuracies
  - revealing the inability of integration relationships and services
- If this test fails
  - → application is architecturally unsuitable for operation
  - → reject the current release
  - → even the most important functions do not work
- E.g.: in the case of a webshop
  - does registration work?
  - is the page loading?
  - does the cart work, etc.

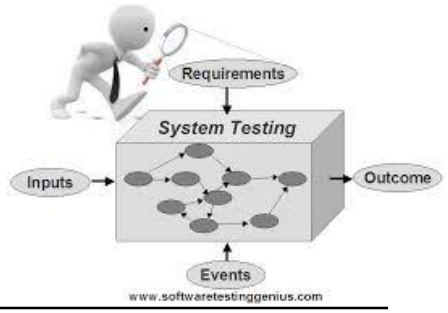
System Test – Sanity Test



- **Is it even worth continuing to test the system after certain modifications?**
  - if a specific feature has been changed or improved, we check whether that part actually works well and has not caused other problems
  - **working logically?**
  - no complete regression test, only one fast, targeted check

- Narrow and deep test
  - examines the functionality affected by the modifications in detail

System Test – Sanity Test



- Usually manually
  - verifying a new or modified functionality enough to show that further testing makes sense
- E.g. in the case of a webshop:
  - adding coupons, but other features are broken
  - if a payment error has been corrected, can the purchase now be completed successfully?

Acceptance test

- It also **tests the entire system**, but this **is done by the end users**
  - performed by the client or his agent
    - tester independent from the development team
- Final testing level
  - **separates the system from its production environment**
- The most well-known varieties
  - alpha test
  - beta test
  - user acceptance test
  - operator acceptance test

Acceptance test

- **Alpha test**
  - testing of the finished product at the development company, but not by the development team
  - e.g. can the program be laid out by clicking back and forth (monkey test)

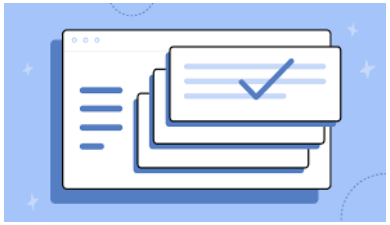
- **Beta test**
  - performed by a certain group of end users
  - e.g. in games: before release, the game is sent to a few fanatic players, who play it a lot in a short time. But they have to discover any bugs

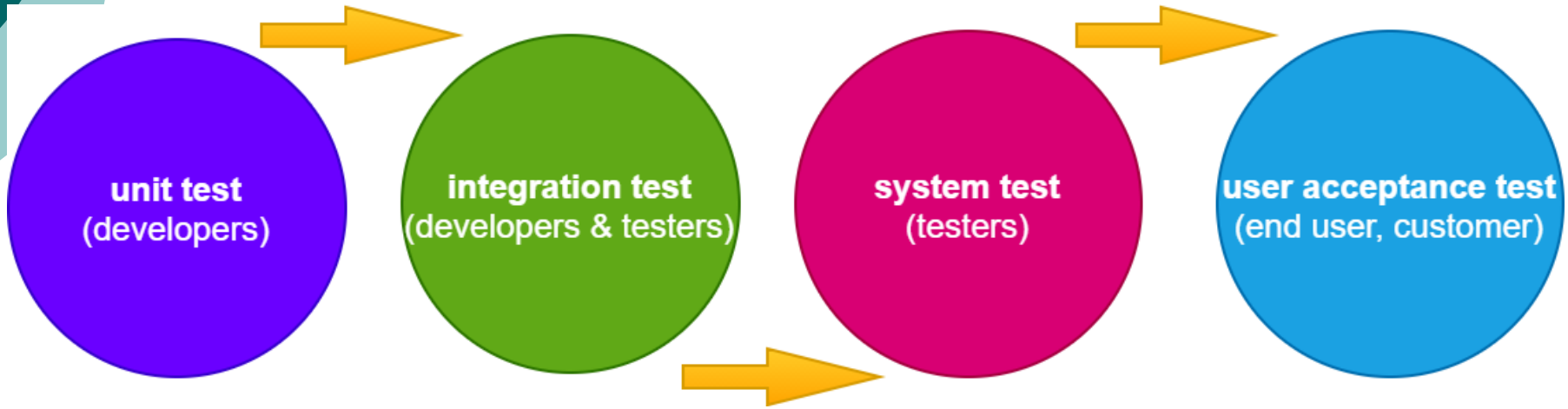
Acceptance test

- **User acceptance test**
  - almost all users receive the software and use it in the production environment
  - installed and used, but not yet in production
  - goal: users should be convinced that the product is safe to use

- **Operator acceptance test**
  - administrators check
    - security features, e.g. are the backup and recovery working correctly

Summary




unit test
(developers)
integration test
(developers & testers)
system test
(testers)
user acceptance test
(end user, customer)

Thank you you for your attention !