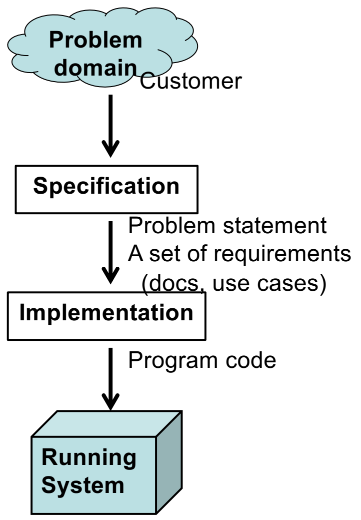
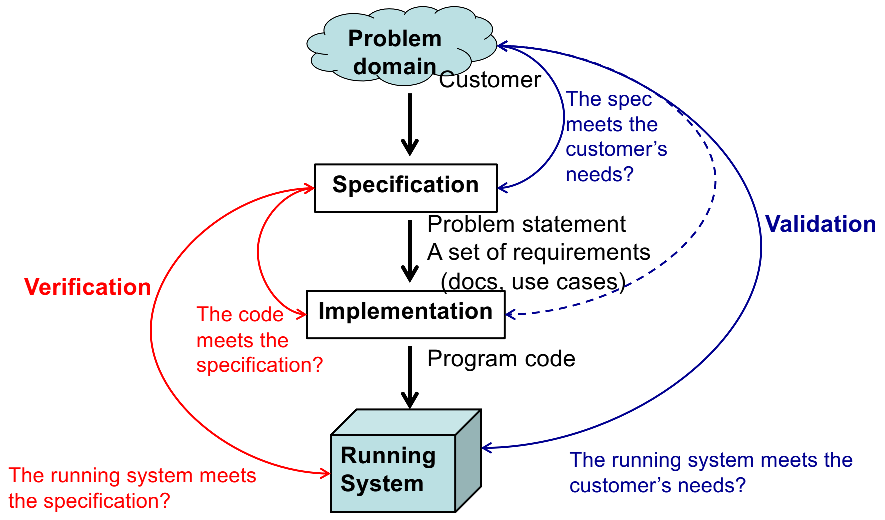
Software Testing: Verification and Validation

Software Development



Verification and Validation (V&V)

- Verification
  - Testing whether a system is developed in accordance with its specification (i.e., a set of gathered requirements).
    - Ensures you built it right.
- Validation
  - Testing whether a system meets your customer's needs.
    - Ensures you built the right thing.


Problem domain
Customer
The spec meets the customer's needs?
Specification
Problem statement
A set of requirements
(docs, use cases)
Validation
Verification
The code meets the specification?
Implementation
Program code
Running System
The running system meets the specification?
The running system meets the customer's needs?

Defects in V&V

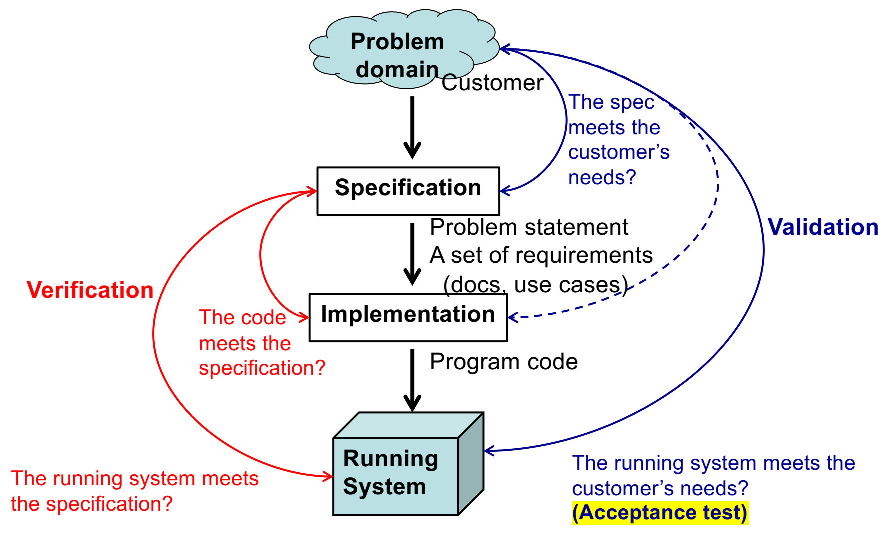
- Defects found in verification
  - Found when the implementation and/or running system fail to meet the specification.
    - e.g., The spec. of a printer's firmware states that the printer stops printing when its paper tray is empty.
      - However, the firmware doesn't stop the printer when a tray is empty.

- Defects found in validation
  - Found when the specification is wrong or misses the customer's needs.
    - e.g., The firmware's spec. states nothing about how the printer should behave when its tray is empty.
      - Thus, the firmware does not stop printing when a tray gets empty.
      - However, the customer wants the printer to stop.

Importance of Validation

- Need to correctly define requirements in the specification, so…
  - Developers can clearly tell what needs/features to implement and how to implement them.

- However, it is not always easy to make the specification sufficiently comprehensive, so..
  - Developers do not miss the customer's needs.
  - Requires numerous "what-if" discussion.
    - What if a tray gets empty?
      - The on-going print job should stop immediately?
      - What if another tray has papers?
      - Can the printer still accept extra print jobs from computers?
  - Requires "acceptance test" by the customer



An Example Defect in Validation

- Firmware for Boeing 787's generator control unit (GCU)
  - Does periodic "status check" every 10 milliseconds.
  - Had a counter (timestamp) with signed (!) 32-bit integer.
    - 2^31=2,147,483,648 (> 2B)
    - 10 msec * 2,147,483,648 = 248.551 days
    - An integer overflow occurs once GCU has continuously operated for 248.551 days.

Counter	Status
0	G
1	G
2	G
⋮	⋮

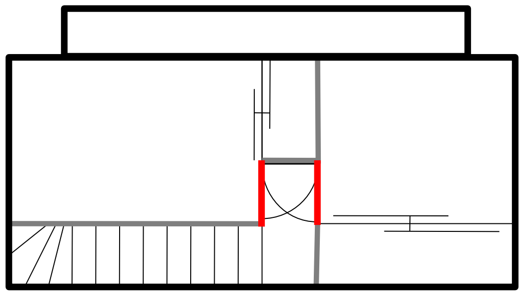
- The power generator (GCU) fails if it is powered on for about 248 days.
- A 787 aircraft has 4 generators.
  - If all of them are powered on at the same time, the aircraft can lose its control in about 248 days.

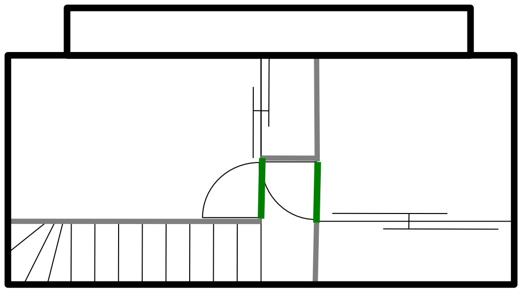
- GCU customer
  - Didn't consider and wasn't asked how long a GCU should be able to keep running if it is not turned off.
    - Status check might look like a minor feature in GCU development.
  - Did consider or was asked about it, but it was not stated in the firmware specification.
- GCU manufacturer
  - Didn't consider and wasn't instructed (by the specification) about up to how long a GCU should run if it is not turned off.
  - Decided to use one of the simplest data types for the counter and didn't have a chance to re-visit the decision.
    - Status check might look like a minor feature in firmware development.

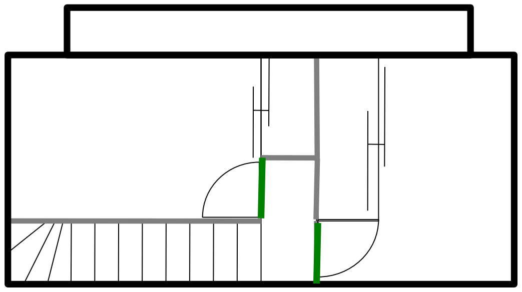
X-day Problems

- 248-day problem
- 494-day problem
  - Occurs if a counter/timer relies on an unsigned 32-bit integer
    - Server OSes, WiFi routers, network switches, etc. etc.
- 24-day and 49-day problems
  - Occur if a counter/timer relies on a signed or unsigned 32-bit integer and its counting/timing resolution is 1 millisecond.
- 830-day problem
  - Occurs if a counter/timer relies on an unsigned 32-bit integer and its counting/timing resolution is 60 Hz (1/60 second; 16.67 msec)
- Year 2038 problem (Unix millennium problem)
  - Many OSes have a timer that counts time in second from 1970/1/1 0:00:00, using a signed integer. The timer will overflow at January 19 in 2038.

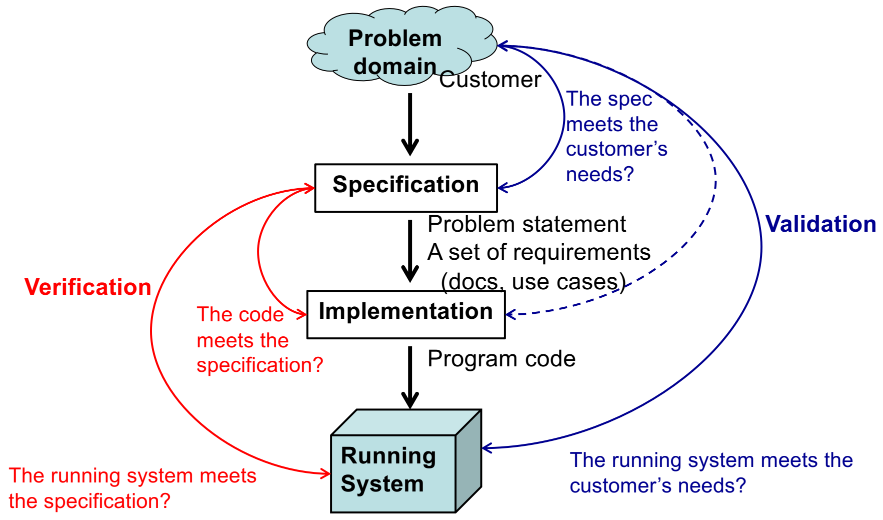
When I was a kid...





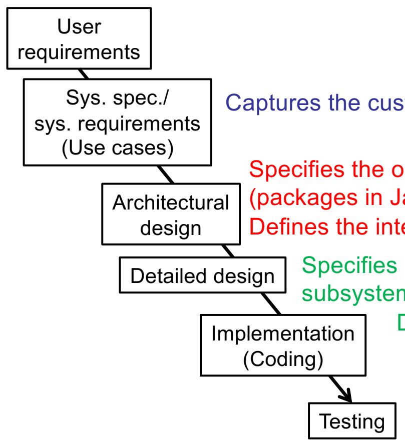


Software Development


Problem domain
Customer
The spec meets the customer's needs?
Specification
Problem statement
A set of requirements
(docs, use cases)
Validation
Verification
The code meets the specification?
Implementation
Program code
Running System
The running system meets the specification?
The running system meets the customer's needs?

Waterfall Process Model

- One of the earliest models to describe development processes.



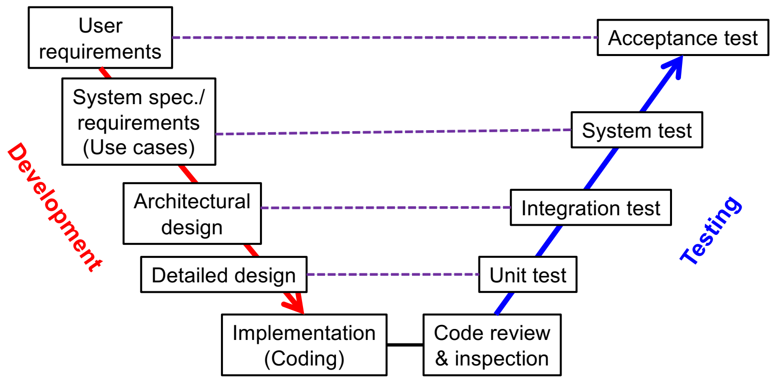
Captures the customer's requirements/needs.

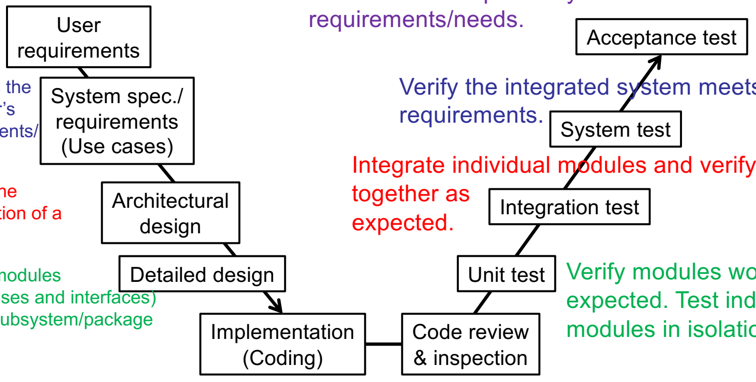
Specifies the organization of a system. Defines subsystems (packages in Java) and their structures and behaviors. Defines the interfaces between them.

Specifies modules (e.g., classes and interfaces) in each subsystem. Defines their structures and behaviors. Defines the interfaces between them.

V-Model

- Extends the waterfall model.
  - Testing phase is expanded
- Explicitly states which testing phase corresponds to which development phase.





Captures the customer's requirements/ needs.

Specify the organization of a system.

Specify modules (e.g. classes and interfaces) in each subsystem/package

Validate the completed system meets the customer's requirements/needs.

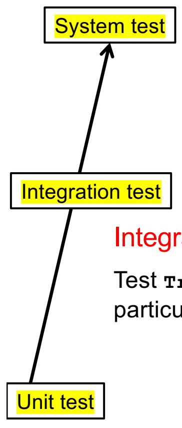
Verify the integrated system meets system requirements.

Integrate individual modules and verify they work together as expected.

Verify modules work as expected. Test individual modules in isolation.

Verify the impl of modules. Tools to check coding style and detect error-prone code. Peer review among proj. members for error detection, quality/performance improvements, etc.

Example Tests



Verify the integrated system meets system requirements.

Verify that…

- Paper feeding stops when a particular tray becomes empty.
- An error notification is sent to OS's printing facility (e.g. a print queue)
  OR
- Paper feeding re-starts with another tray if it is not empty.

Integrate individual modules and verify they work together as expected.

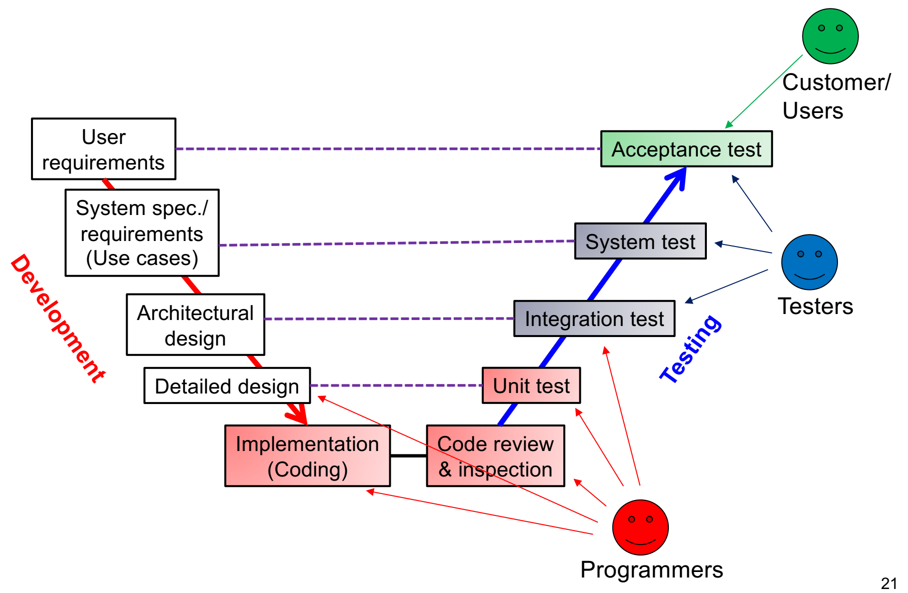
Test **`Tray`** and **`PaperFeeder`** together to verify that paper feeding stops when a particular tray becomes empty:

```
        if(aTray.isEmpty()){ aPaperFeeder.stop(...); }
```

Verify modules work as expected. Test individual modules in isolation.

Test **`Tray`** to verify its **`isEmpty()`** returns true when a particular tray is empty.
Test **`PaperFeeder`** to verify its **`stop()`** actually stops paper feeding.

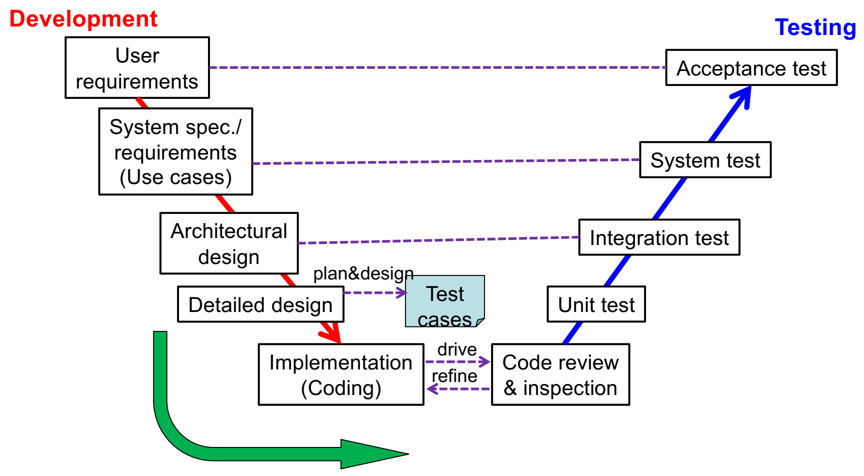
Division of Responsibilities

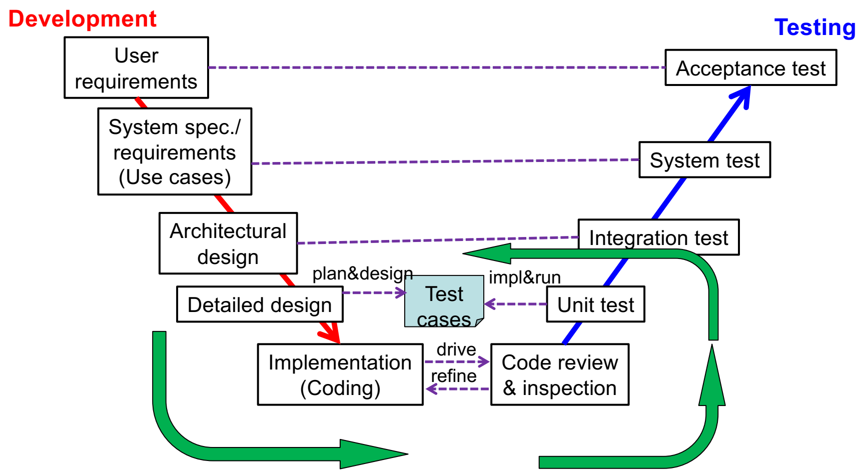


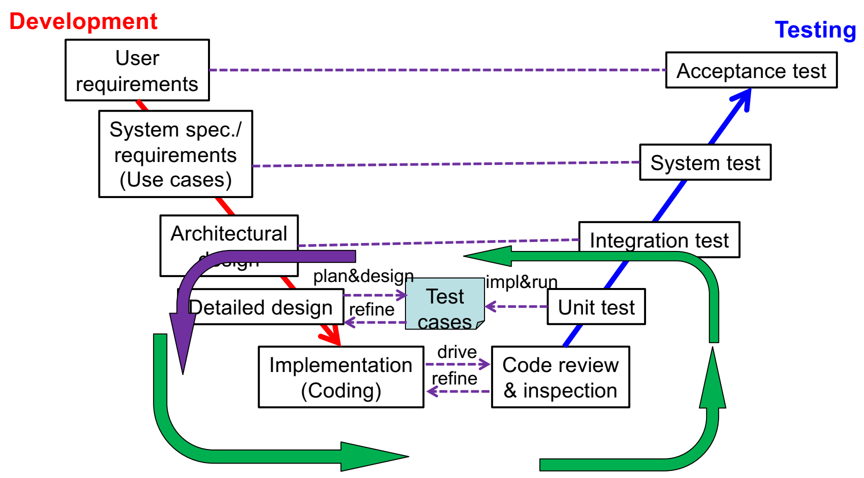
Problems in Waterfall Process

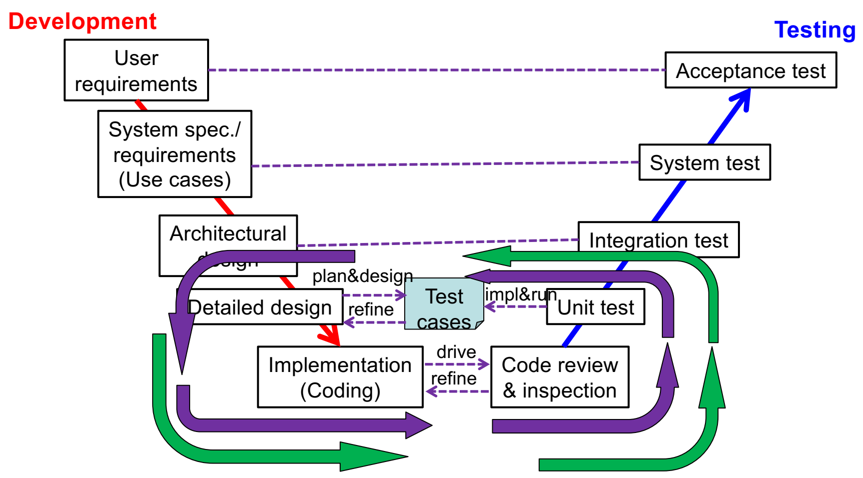
- Defects are found at the end of the project.
  - Testing does not take place until the end of the project.
- It is often too late and too expensive to push feedback up the waterfall.
- One-way waterfall is not often encouraged.
- Iterative processes are encouraged.

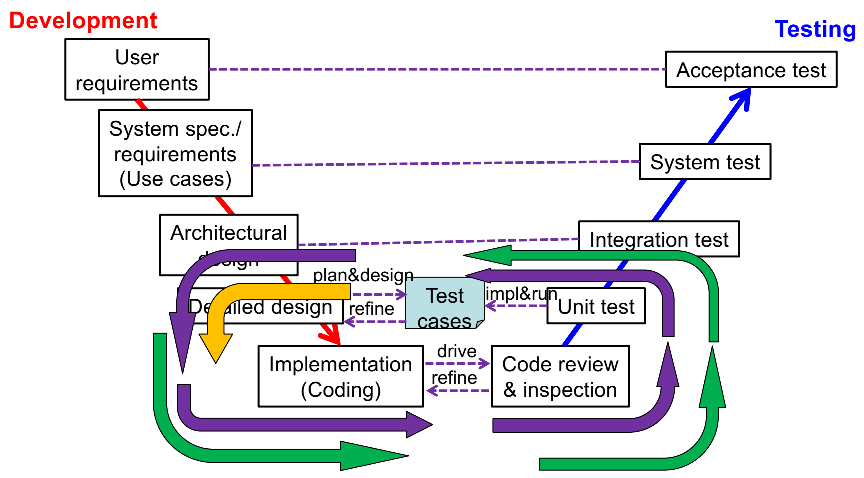
V-Model and Iterative Development Process

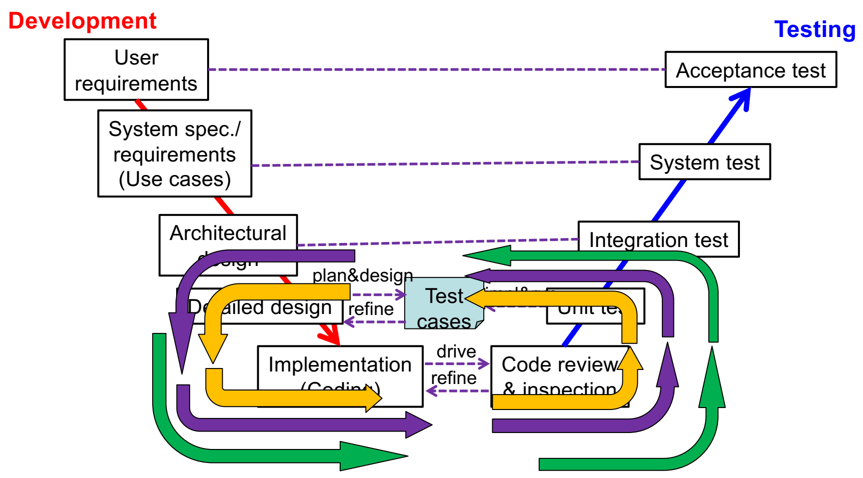





Development
Testing
User requirements
Acceptance test
System spec./ requirements (Use cases)
System test
Architectural design
Integration test
plan&design
Detailed design
refine
Test cases
impl&run
Unit test
Implementation (Coding)
drive
refine
Code review & inspection


Development
Testing
User requirements
Acceptance test
System spec./ requirements (Use cases)
System test
Architectural design
Integration test
plan&design
Detailed design
refine
Test cases
impl&run
Unit test
Implementation (Coding)
drive
refine
Code review & inspection


Development
Testing
User requirements
Acceptance test
System spec./ requirements (Use cases)
System test
Architectural design
Integration test
plan&design
Detailed design
refine
Test cases
impl&run
Unit test
Implementation (Coding)
drive
refine
Code review & inspection


Development
Testing
User requirements
Acceptance test
System spec./ requirements (Use cases)
System test
Architectural design
Integration test
plan&design
Detailed design
refine
Test cases
Unit test
Implementation (Coding)
drive
refine
Code review & inspection

Test Levels and Test Types

- Test level
  - Corresponds to a "development level."
    - e.g., unit test, integration test, system test and acceptance test.
  - A group of test activities that are organized and managed together.
- Test type
  - Focuses on a particular test objective.
    - e.g. functional test, non-functional test, structural test, confirmation test, etc.
  - Takes place at one test level or at multiple levels.

- Test levels and test types are orthogonal.

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test				
System test				
Integration test				
Unit test				
Code rev&insp.				

- Different projects have difference policies on which test types involve in which levels.
- For example…

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

Test Types: Functional Test

- Focuses on the functional (external) behaviors of tested code
  - Driven by the descriptions and use cases specified in the specification.
  - Black-box testing
    - Treat the tested code as a black-box
      - Testing *without* knowing the internals of tested code
    - Give an input to tested code and compare its output with the expected result.
      - Coarse-grained testing: Testing the external behaviors of tested code

Client code

```
Robot r = new Robot();
r.control(Robot.CMD_MOVE_FORWARD);
```

Robot

+ CMD_MOVE_FORWARD: int=0 {final}
+ CMD_STOP: int=1 {final}
+ CMD_MOVE_BACKWARD: int=2 {final}

+ control(command: int): void

Client code

```
Student s = new Student(...);
s.getTuition();
```

Student

- name: String

Student(status: StudentStatus, Name: String)
+getTuition(): float
+getName():String
…

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

Test Types: Non-Functional Test

- Focuses on the non-functional quality characteristics of tested code.
  - Driven by the descriptions specified in the specification.
  - Security test
    - Check if security vulnerability exists in tested code.
  - Usability test
    - Ease of use/browse/comprehension, intuitive page/screen transition
  - Efficiency test
    - Performance (e.g. response time, throughput), resource utilization (e.g. memory, disk, bandwidth, energy/battery)

- Reliability test
  - Stress test (load test)
    - How does tested code behave under an excessive load?
      - » Example loads: huge data inputs, numerous network connections
  - Long-run test
    - Does performance degrade when tested code runs for a long time?
  - High frequency test
    - How does tested code behave when it repeats a certain task at excessively high frequency?
  - Fault-tolerance test
    - Can a tested code continue its operation under a fault?
  - Recoverability test
    - How can a tested code recover its operation and data after a disaster (e.g. physical damages of hardware, blackout)?
  - Compliance test
    - Data retention, access control, logging, etc.

– Environmental test
- Configuration/compatibility test
  - – Can the tested code be installed on certain OS(es) and HW(s)?
  - – How does the tested code behave on certain OS(es) and HW(s)?
  - – How does the tested code interact with an external required service(s)?
    - » Does it work with Version X of the service? How about Version Y?
- Co-existence test
  - – Can the tested code run correctly when other software/services run on the same machine?

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

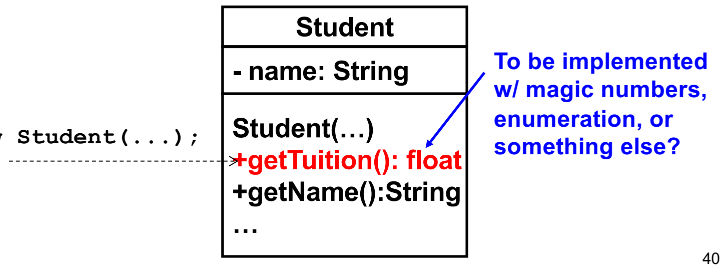
Test Types: Structural Test

- Testing the internal structure of an individual module or a set of (integrated) modules.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  - – Refactoring
    - » e.g. Replacing conditionals with polymorphism, replacing a magic number with a symbolic constant.
    - » Revising the interfaces of modules if an integration test fails.
      - » Interface: Defines how modules interact with each other
  - – Use of design patterns
    - » e.g., Replacing conditionals with the *State* design pattern

- White-box testing
  - – Treat tested code as a white-box
    - Testing *with* the knowledge about the internals of the tested code
  - – Fine-grained testing: Taking care of internal behaviors of tested code

Client code

```
Student s = new Student(...);
s.getTuition();
```



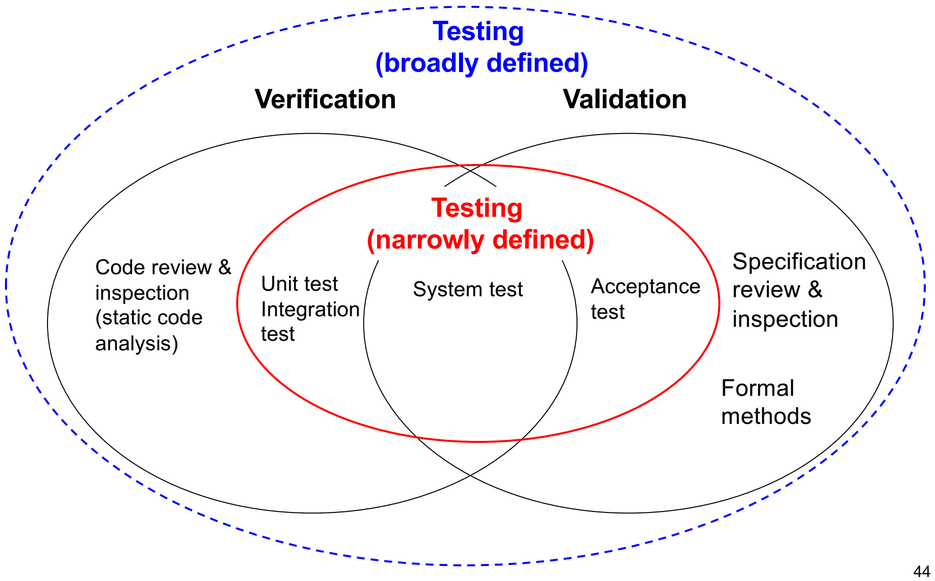
	Functional test	Non-functional test	**Structural test**	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	**X**	X
Unit test	X	?	**X**	X
Code rev&insp.	X	?	**X**	X

Test Types: Confirmation Test

- Re-testing
  - When a test fails, detect a defect and fix it. Then, execute the test again
    - To confirm that the defect has been fixed.
- Regression testing
  - In addition to re-testing, execute ALL tests to confirm that the tested code has not regressed.
    - That is, it does not have extra defects as a result of fixing a bug.
    - Verifying that a change in the code has not caused unintended negative side-effects and it still meets the specification.

	Functional test	Non-functional test	Structural test	**Confirmation test**
Acceptance test	X	X		
System test	X	X		**X**
Integration test	X	?	X	**X**
Unit test	X	?	X	**X**
Code rev&insp.	X	?	X	**X**

V/V Methods


Testing
(broadly defined)
Verification
Validation
Testing
(narrowly defined)
Code review & inspection (static code analysis)
Unit test
Integration test
System test
Acceptance test
Specification review & inspection
Formal methods