Software Testing and Validation

A.A. 2024/2025

Corso di Laurea in Informatica

Testing within the Software Process

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

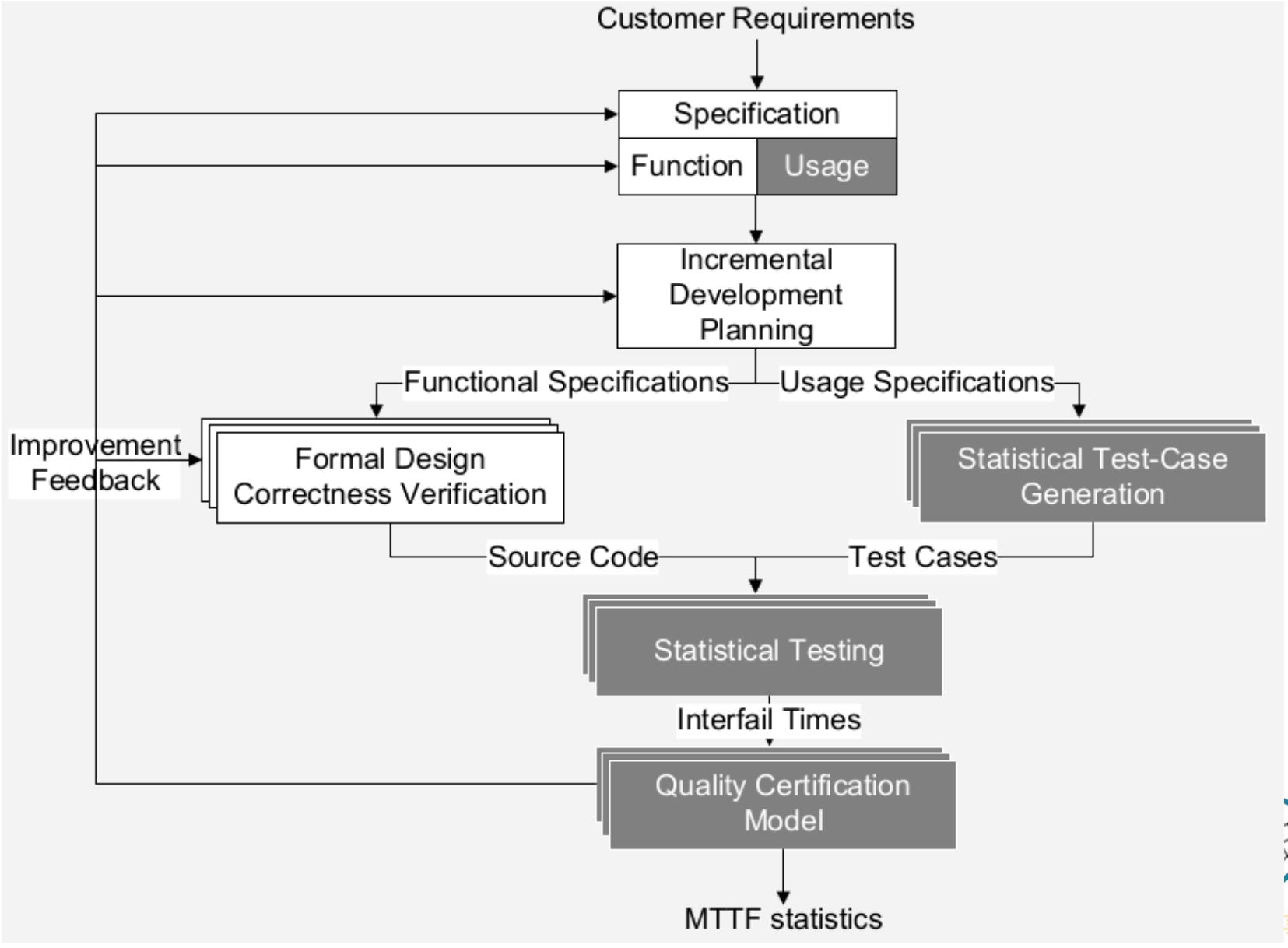
UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Quality Planning

- That is: embed testing in the software design process
  - "quality" in the sense of errors being discovered and corrected
  - both functional and non-functional errors
- Quality planning involves deciding in advance when and how to perform testing
  - of course, intertwined with the overall software design process
- *Incremental* development: a first draft is initially written and then continuously revised
  - again, the same usually happens for the software design process
- Better solution is to appoint someone for the software quality process

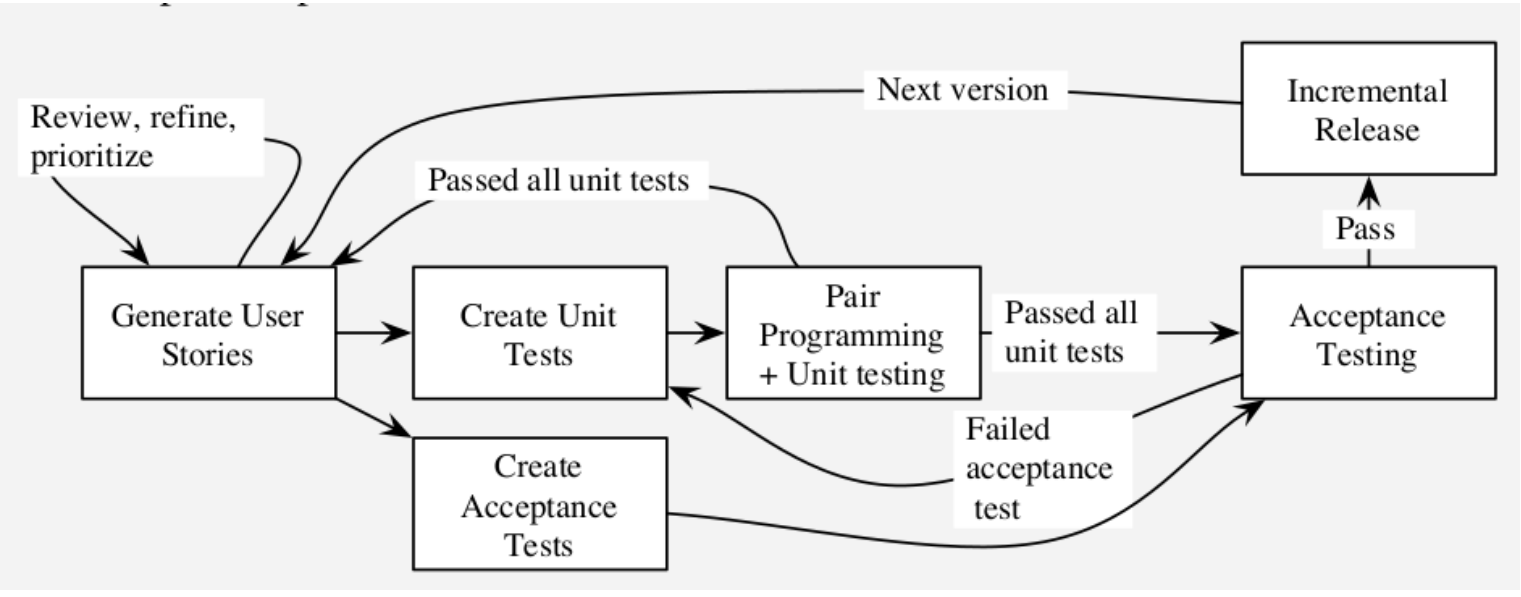
IBM CleanRoom Process Model



IBM CleanRoom Process Model

- Developed in the 1980s
- Involves two cooperating teams:
  - the development team is responsible for the software design and implementation
  - the quality team is responsible to find errors and certify that the system is being developed in the right way
- Five major activities, continuously revised:
  - specification: required behavior of the system (development team) and usage scenarios for test suites (quality team)
  - planning: design and update development and the quality plan
  - design and verification: system increments are developed and certified
  - quality certification: test the system w.r.t. the specification
  - feedback: both for errors (quality $\rightarrow$ development) and also for the whole process if the error rate increases too much

Extreme Programming Process Model



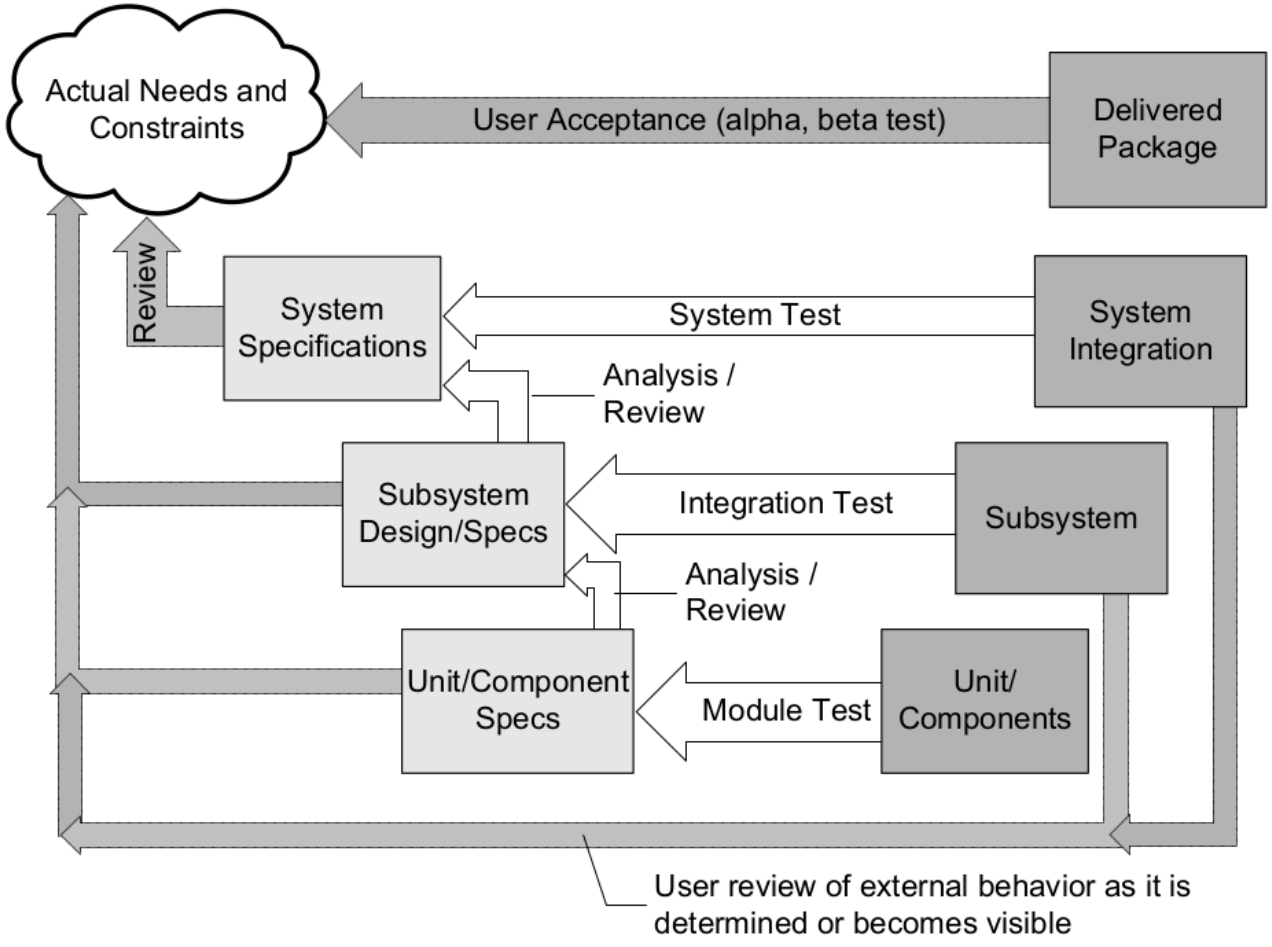
Extreme Programming Process Model

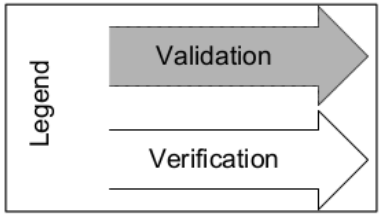
- The extreme programming methodology emphasizes:
  - global vision and communication over structured organization
  - frequent changes over big releases
  - continuous testing and analysis over separation of roles and responsibilities
  - continuous feedback over traditional planning
- *User stories*: requirements from customers
  - test cases corresponding to scenarios in user stories serve as partial specifications
- *Test-first*: test cases specifications are built *before* the actual code to test
- *Pair programming*: developers work in pairs, incrementally developing and testing a module
- For each release, run all the tests devised up to that point: kind of merging of unit testing with integration and system testing

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Traditional V Model





Traditional V Model

- Four degrees of granularity in testing:
  - module or unit: each part of program agains its specifications
  - integration: checks compatibility among selected modules
  - system: checks whole system against specifications
  - acceptance: checks whole system w.r.t user needs (validation)
- Integration faults: faulty specifications or implementations of interfaces, resource usage, or required properties
- Integration errors may reveal something flawn also at unit testing
- In other cases, side-effects of module faults may become apparent only in integration test
- "Incompatible" components: when they do not work together for some reason
- Integration tests focus on checking compatibility between module interfaces

Integration Faults

- Inconsistent interpretation of parameters or values
  - ok if taken separately, not ok when together
  - typical case: different units used in different methods, e.g., meters vs feet
- Violations of value domains or of capacity or size limits
  - violation of (implicit) assumptions on ranges of values or sizes
  - buffer overflow
- Side-effects on parameters or resources
  - two modules writing on the same file
- Missing or misunderstood functionality
  - a module expects another module to return something, but it is something else
  - e.g., Web hits counted per unique IP address or per request
- Nonfunctional problems
  - e.g., missed deadlines due to module call
- Dynamic mismatches
  - e.g., polymorphic calls bound to the wrong method

Integration Testing Strategies

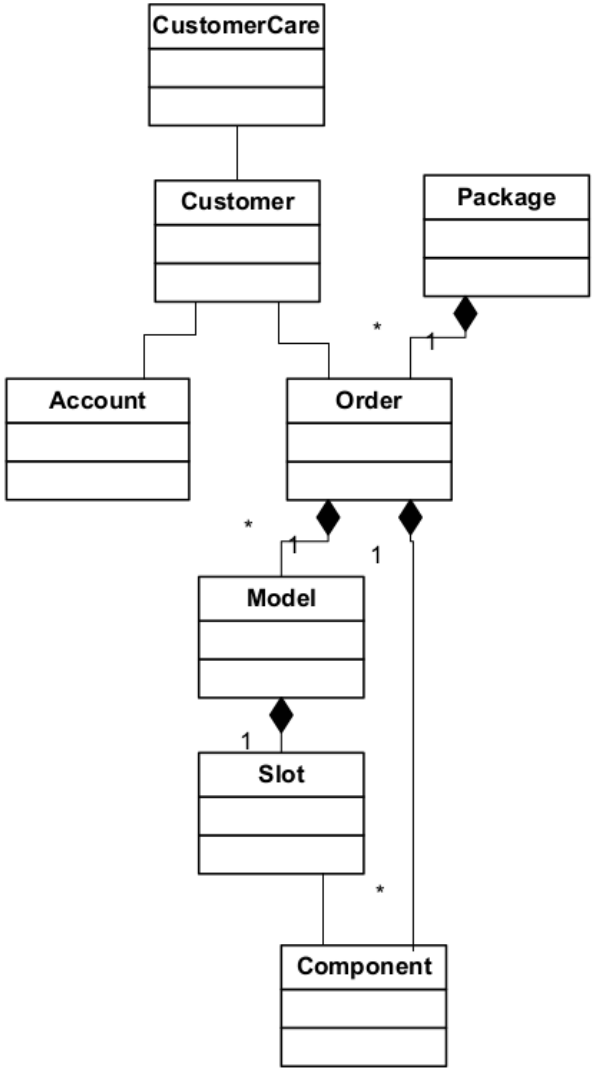
- First thing to be decided: the sequence of modules integration to be tested
  - A, B and then B, C, D? or C, E first?
- Better to follow the build plan: test modules integration as soon as they are ready
  - often, the viceversa also holds, i.e., integration testing drives build plan
- In any case, an incremental strategy is followed
  - when all modules are present, we have system testing
  - even with incremental strategies, some modules may not be available, thus scaffolding is needed for drivers and stubs
- Sometimes, the big bang strategy may be used: directly go with system testing
  - all modules are available
  - good for budget
  - not good for early errors discovery, which is however a problem for budget...
  - *desperate tester strategy*

Integration Testing Strategies

- So, let's go back to incremental integration testing; we have two possibilities:
  - structural oriented:
    - order of integration is based on hierarchical structure in the design
  - feature oriented: derive the order of integration from characteristics of the application
    - more "important" and "critical" modules are tested first
    - includes threads and critical modules testing strategies
- Within structural testing, both bottom-up and top-down strategies are possible
  - basing on the use/include hierarchy
  - especially ok if the build plan follows the same order
- Sandwitch or backbone strategy: both bottom-up and top-down
  - start from both ends and go towards the middle

Integration Testing Strategies: Example

Use/Include hierarchy from a class diagram



Integration Testing Strategies: Example

Top-down:

1. Integrate CustomerCare with Customer; use stubs for Account and Order
2. Add Account
3. Add Order and Package, stubbing Model and Component
4. Add Model, Slot, and Component in this order
   - drivers are built only in the first step, then incrementally updated

Bottom-up:

1. Integrate Slot with Component, using drivers for Model and Order
2. Add Model and Order, using a driver for Package
3. Add Package, Account, Customer and CustomerCare
   - no stub need in any point

Integration Testing Strategies: Example

Sandwitch: suppose we are reusing existing modules for Model, Slot and Component, and developing CustomerCare and Customer as part of an early prototype

1. Integrating CustomerCare and Customer, with stubs for Account and Order
2. Integrate Model, Slot and Component with Order, using drivers for Customer and Package
3. Integrate Account with Customer, and Package with Order, before finally integrating the whole prototype system.

Feature-Oriented Integration Testing Strategies

- Thread testing: choose one functionality and pick the interested modules
  - not necessarily "threads" as in parallel programming
- Critical module integration testing: first test modules that represent a risk for the project
  - modules may be sorted basing on the risk they pose in case of failure
  - both external risks (e.g., safety) and internal risks (e.g., missing project deadlines) must be considered
- Feature-oriented testing is more expensive than structural-oriented testing
  - used for bigger projects

Components and Frameworks

Component reusable unit of deployment and composition

- may be used many times by different teams
- may have an internal state
- may be composed by many objects
- may use persistent storage
- may require some communication layer (not simply method calls)

Component contract or interface describes component access points and parameters

- also specifies functional and non-functional component behaviour
- also specifies required (assumed) conditions
- sometimes also called API (Application Program Interface)

Components and Frameworks

Framework micro-architecture or skeleton of an application

- easy to add application-specific functionality or configuration-specific components
- may be seen as a circuit board with empty slots for components
- not to be confused with design patterns:
  - patterns are logical design fragments, frameworks are concrete elements of the application
  - frameworks often implement patterns

Component-based system system built by assembling software components

- connected by a framework or specific code

COTS Commercial Off-The-Shelf (component): built to be sold to other developers

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Testing for Components and Frameworks

- Component built for general use are typically more complex than components built ad-hoc for a given application
- Main problem: developers do not know the context in which their component will be used
  - may be used also if it does not perfectly fit
- Of course, better to start with applications of typical usage
- Possible uses may be classifed in scenarios

System, Acceptance and Regression Testing

- All of them look at the whole software to be delivered
- System testing: integration testing with all components available
  - also including properties about the whole system
- Acceptance testing: aka validation
  - instead of checking specifications, ask the final users
- Regression testing: check if, when going from an old release to a new one, some errors have been introduced
  - code modifications may produce failures not experienced in previous releases

System, Acceptance and Regression Testing

System, Acceptance, and Regression Testing		
System test	Acceptance test	Regression test
Checks against requirements specifications	Checks suitability for user needs	Rechecks test cases passed by previous production versions
Performed by development test group	Performed by test group with user involvement	Performed by development test group
Verifies correctness and completion of the product	Validates usefulness and satisfaction with the product	Guards against unintended changes

System Testing

- Ideally, no scaffolding: based on observable evidence of the whole system
  - design and implementation are not important, it must work
  - more: it must be independent on design and implementation
- However, some scaffolding may be needed to test controllers
  - a simulator is (initially) used instead of the system to be controlled
- ... or to keep track of the test results (e.g., in a DB)
- System test suites may contain some test suites used for integration or even unit test
  - especially true if testing was feature oriented
  - structural testing is not good for system testing, as it is not independent on the implementation

System Testing

- How to obtain test suites independent on the design/implementation:
  - give the task to a different team
  - design system tests very early, before any design choice has been done
- Agile software development: develop a new functionality as soon as it is specified
  - in between specification and implementation, derive test cases
- System testing only looks at system-wide properties and usage scenarios
  - each desired behaviour must be taken into account by at least one test case
- Additional test cases can be added during development if unforeseen observable failures happen
  - also considering final users annotations
  - intertwined with acceptance and regression testing

System Testing

- The type of properties we want in system testing are the harder to evaluate
  - often non-functional: low latency, system response, mean time between failures
  - also functional like security or safety
- For security or safety, better use model checking
  - for security, have another team try to breach the system...
- Performance testing: outside the scope of this course
- Note that the environment is important
  - impractible for a fast-response system to withstand too many request
- Stress test: repeat tests many times

System Testing: Fuzzing

- Especially useful when the system is a *server*
  - that is: in a never-ending loop, check if there is some request and, if it is the case, handle it
  - example: any Internet-based service such as Web, email etc.
  - also ok for non-server software which request inputs at different stages of execution
  - e.g., highly *interactive* software
- Fuzzing consist in feeding illegal inputs *only*
  - and to feed them with high frequency
- Often combined with instrumented software to measure coverage (*fuzzing coverage*) and with Property-based Testing
  - it would be difficult to generate correct outputs to be checked...
- This can be seen as a special type of stress test

All Testing

Unit, Integration, and System Testing

	Unit Test	Integration Test	System Test
Test cases derived from	module specifications	architecture and design specifications	requirements specification
Visibility required	all the details of the code	some details of the code, mainly interfaces	no details of the code
Scaffolding required	Potentially complex, to simulate the activation environment (drivers), the modules called by the module under test (stubs) and test oracles	Depends on architecture and integration order. Modules and subsystems can be incrementally integrated to reduce need for drivers and stubs.	Mostly limited to test oracles, since the whole system should not require additional drivers or stubs to be executed. Sometimes includes a simulated execution environment (e.g., for embedded systems).
Focus on	behavior of individual modules	module integration and interaction	system functionality

Acceptance Testing

- Tries to ask the question: "Should we release the product?"
- To this aim, we may:
  - still perform some dedicated testing, in addition to system testing
  - ask the users (validation)
    - easy for very specialized software commissioned by a small group of users
    - otherwise, a specific organization must be set up
- Dedicated testing for acceptance: must be separated from system testing
  - unit, integration and system testing: expose as many failures as possible
  - acceptance testing: understand if it is useful for the final user

Acceptance Testing: Validation

- Validation: directly ask users
- Two main workhorses: alpha and beta testing
  - as it may be guessed, alpha refers to early development releases
  - where very few testing has been carried out
  - beta is for more advanced development phases of a release
- Alpha testing may be performed by the software company
- Beta testing is usually performed by volunteering final users
  - note that beta testing is not organized
  - i.e., final users simply use the product, and report failures to developers
  - if different categories of final users are present, choose at least a representative in each category
  - in some sense, the users themselves are sampling their *operational profiles*
  - some scaffolding to send feedback is necessary

Acceptance Testing

- *Operational profiles*: statistical models of usage
  - available from previous similar projects
  - e.g., how a new DBMS will be used should not be different from how old ones were used
- Sensitivity testing: identify parameters of operational profiles and determine which are the important ones
  - repeat many times statistical testing, each time varying some parameters
  - e.g., vary the incoming load to see the effect in system throughput
- This kind of validation is somewhat "statistical": we want a "measure" of the product reliability

Early Acceptance Testing: Usability

- Alpha and beta testing are for the final product, but users may be involved earlier
  - especially for the usability of the software
- Exploratory testing: investigate the "mental model" of end users
  - especially for GUI: first present a very simplified version and see what users choose first
  - useful when designing a product for a new population
- Comparison testing: evaluate different options
  - observe users reactions to different proposals
  - again, early stages of software design
  - mainly to refine interaction patterns

Early Acceptance Testing: Usability

- Validation testing: assess overall usability
  - identify difficulties and obstacles for final users
  - time to perform tasks
  - error rate
- Overall, usability testing go through:
  - preparation:
    - define the objectives of the session
    - identify the items to be tested
    - select a representative population of end users
    - plan the required actions
  - execution:
    - execute planned actions in a controlled environment
  - review and analysis
    - plan changes, if required

Acceptance Testing: About Final Users

- Users time is very expensive
- Number of users must be chosen accordingly to project budget
  - representative of users classes, if any
  - questionnaires should be prepared, also to verify class belonging
  - opinions from different classes of users could be weighted differently
- Alpha and beta testing are at users premises
- Especially for usability, testing for users is instead in a controlled environment
  - users are given tasks to be completed
  - their interactions are recorded, sometimes in a light (mouse clicks) sometimes in a heavy (eye tracking and similar) way

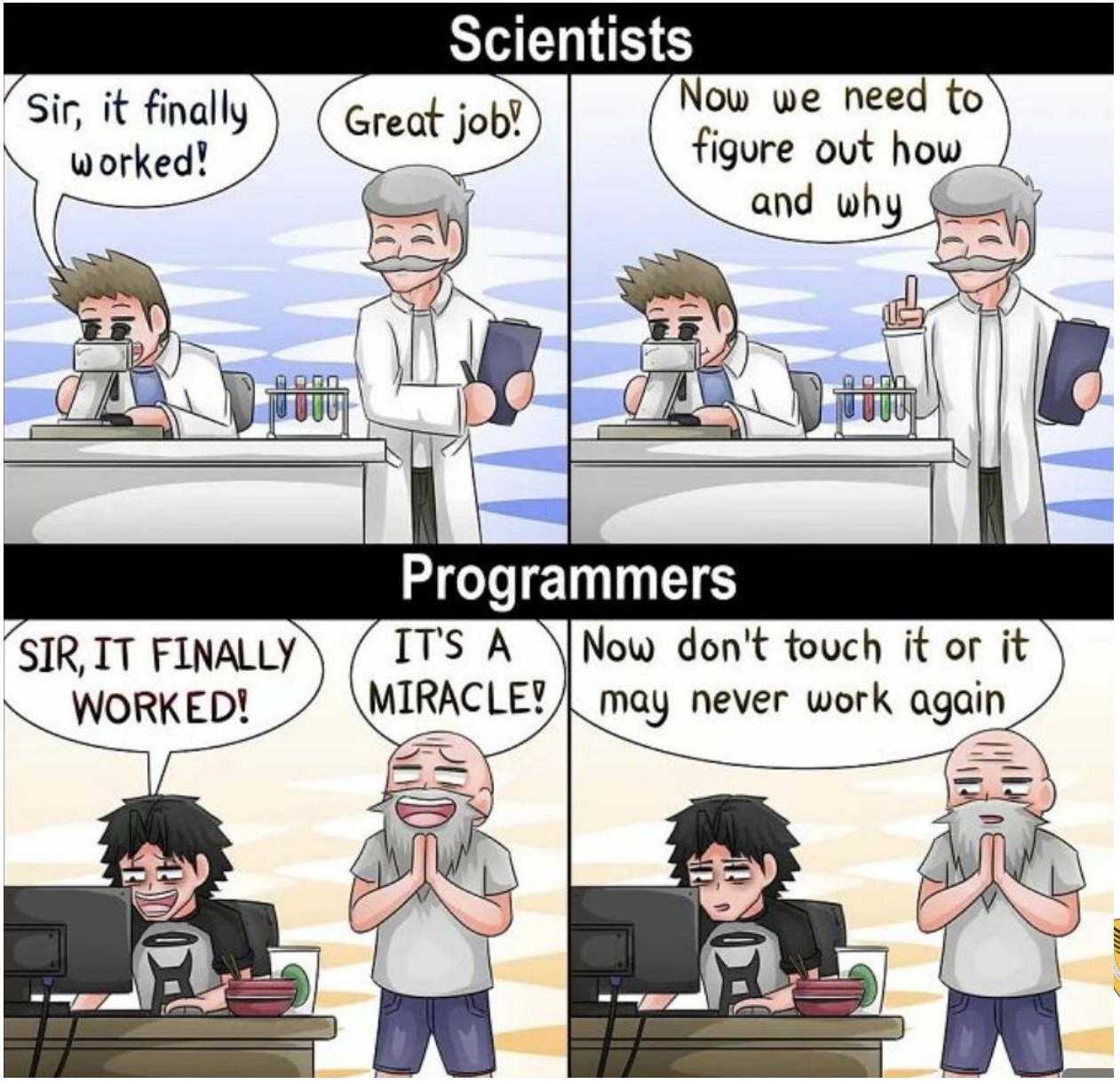
Acceptance Testing: About Final Users

- Accessibility: usability for users with disabilities
  - legally required in some application domains
  - e.g., Web sites of public institutions
  - we also have a standard: Web Content Accessibility Guidelines (WCAG)

Regression Testing

- Software applications are almost never built once and for all
- New releases may be required because of:
  - removing faults (or security errors)
  - changing some functionalities (including changes in the code only)
  - adding new functionalities
  - removing old functionalities
  - porting the system to a new platform
  - extending interoperability
- Where there are changes, there is trouble!
- May be needed to restart the whole testing phase, from unit to acceptance

Regression Testing



Regression Testing

- The smallest change may affect other software parts in unintended ways
  - e.g., a guard added to an array to fix an overflow problem may cause a failure when the array is used in other contexts
  - e.g., porting the software to a new platform may expose a latent fault
  - e.g., even compiling some C code with optimization options may cause previously undected errors
- *Regression*: when a new release of the system introduces new errors in previously working parts
  - thus we want *nonregression* to happen
- Of course, this should be achieved at design time, but wanting is not achieving
- Thus, we need *(non)regression testing*

Regression Testing Techniques

- Solution 0: for a new release, *retest all*
  - this is ok if we only changed the implementation of some methods
  - or if we made a porting
  - for any other modification, old test cases may not work any more
  - in that case, try to select a working test cases subset, if any
- Solution 1: if we have scaffolding able to interpret test case specifications, we simply modify the scaffolding
  - e.g.: if we modify the collapsing strings example by adding the further input k in a new release, we may still adapt the "old" test cases generator
  - still a problem for new functionalities
  - also for old ones, but removing is easier than designing new test cases

Regression Testing Techniques

- Note that some test cases may become redundant
  - especially for structural testing: e.g., two tests covering different paths now cover the same path
  - may become redundant also for changes in the testing itself
  - e.g., in the partition method, we change the partition, causing two previously different tests to be now on the same partition
- Redundant test cases do not reduce the overall effectiveness of tests, but impact on the cost-benefits trade-off
  - unlikely to reveal faults
  - augment the costs of test execution and maintenance
- However, redundant test cases are typically kept
  - may become helpful in successive versions of the software
- Documentation is important, must include testing info

Regression Testing Techniques

- Often the retest all, even if "corrected", is not viable for the excessive cost
  - large software may need to be tested in many different platforms
  - or however need scarce resources, e.g. users testing or time-to-market
- Is it possible to reduce the size of the tests to be performed?
  - e.g.: we changed the window management, no need to recheck file usage

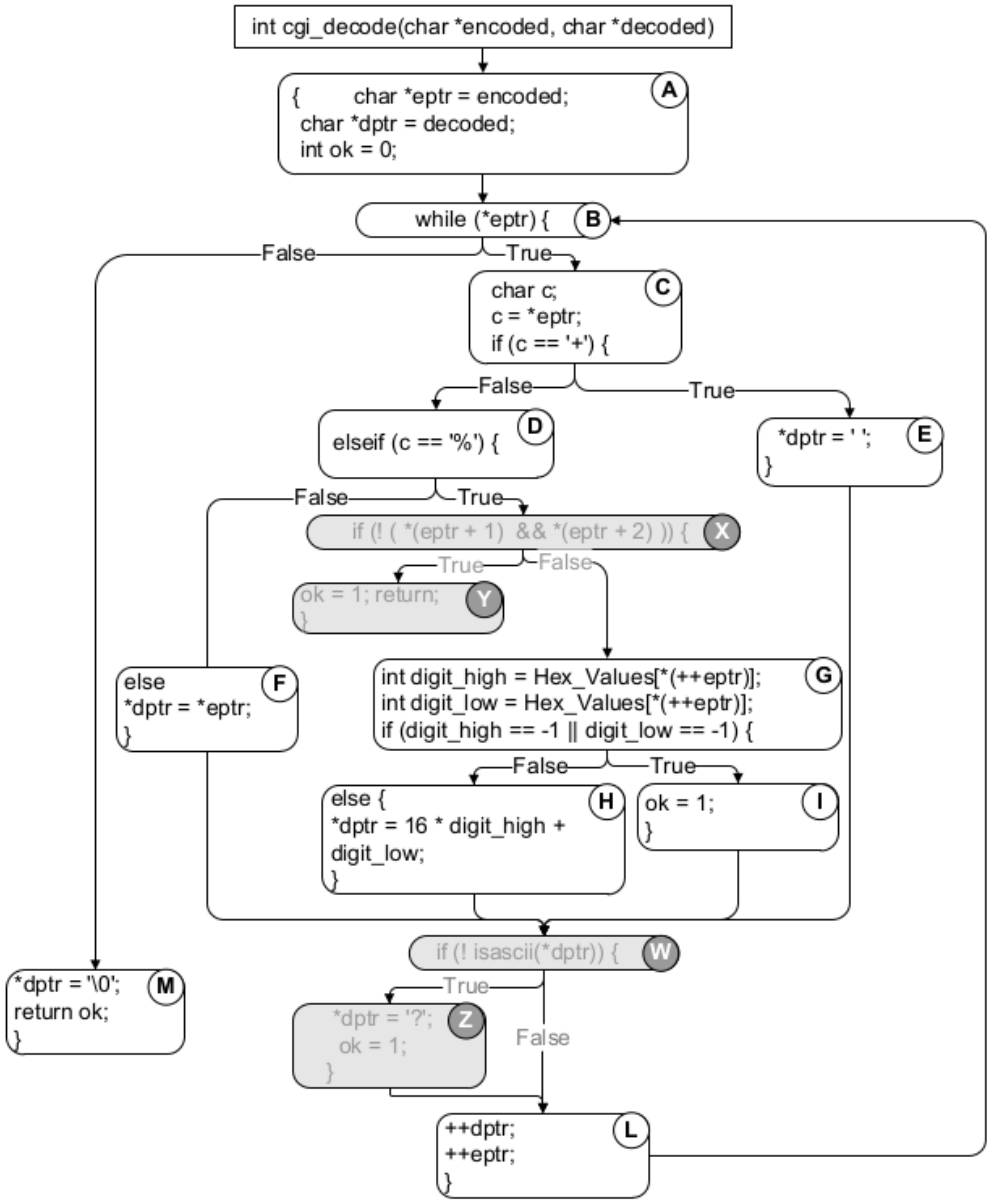
Regression Testing Techniques

- Regression test selection techniques are based on either:
  - *code*: select a test case if it exercises a portion of the code that has been modified
  - *specification*: select a test case if it is relevant to a portion of the specification that has been changed
- Code-based selection may be done automatically
  - especially ok for unit testing, not for integration or system testing
  - not ok if changes are huge
- Specification-based selection work well for all types of testing
  - provided that specification are well written
  - partly automatable if specifications are very well written and organized

Regression Testing Techniques

- CFG regression test selection: based on differences between the two CFGs
  - before and after the change
  - of course, we are in some unit for which the CFG can be produced
  - differences: missing nodes or edges, but also in single nodes annotations
  - for changes in single statements
  - added nodes: selection may be useless, as there were not previous test cases...
- Requires to record the path exercised by the tests
  - must be done automatically
- Selects (past) tests exercising modified CFG parts

CFG Regression Test Selection



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

CFG Regression Test Selection

Id	Test case	Path
TC1	" "	A B M
TC2	"test+case%1Dadequacy"	A B C D F L ... B M
TC3	"adequate+test%0Dexecution%7U"	A B C D F L ... B M
TC4	"%3D"	A B C D G H L B M
TC5	"%A"	A B C D G I L B M
TC6	"a+b"	A B C D F L B C E L B C D F L B M
TC7	"test"	A B C D F L B C D F L B C D F L B C D F L B M
TC8	"+%0D+%4J"	A B C E L B C D G I L ... B M
TC9	"first+test%9Ktest%K9"	A B C D F L ... B M

We may discard TC1

If we only look at X and Y, we may also discard TC6 and TC7

Regression Testing Techniques

- Data flow regression test selection: based on differences between the DU pairs
  - before and after the change, again in some given unit
  - differences: DU pairs may be deleted, added, or modified (definition and/or use were moved)
  - for added ones, selection only is useless...
- Specification-based test selection techniques do not require recording the control flow paths executed by tests
- Regression test cases can be identified from correspondence between test cases and specification items
  - if there was a model extracted from specification, simply update the model and extract tests again
  - code-based selection techniques may be used on such models

Regression Testing Techniques

- Instead of reducing, also giving priorities could be good
  - could be based on the code changes (see below)
  - or also on testing history for previous versions
  - e.g., give low priority to tests which never failed and are not affected by current modifications
- All tests will be eventually executed, but...
  - there are many releases, so typically the same tests are executed many times
  - in each release, execute only tests with priority above a given threshold
  - as a result, some tests will have higher *frequency* than other
  - however, it is guaranteed that all tests will be eventually executed
  - so high priority is also given to tests which have been "waiting" too much

Regression Testing Techniques

- *Execution history priority schema*: low priority to the recently executed tests
  - similar to round robin...
- *Fault revealing priority schema*: high priority to tests which reveled faults
  - faults are not evenly distributed...
  - exercise the parts which needs most testing
- *Structural priority schema*: high priority to tests which cover most "elements"
  - statements, branchs, conditions for unit testing
  - methods, features for integration/system testing