**Multidisciplinary Projects**

**Class 5: Testing**

Contents

[**Bugs, Faults, and Failures** 3](#_Toc140228471)

[**Failure of Requirements** 3](#_Toc140228472)

[**Bugs and Features** 3](#_Toc140228473)

[**Terminology** 3](#_Toc140228474)

[**Failures: A Case Study** 4](#_Toc140228475)

[**Case Study: Analysis** 4](#_Toc140228476)

[**Case Study: Software Lessons** 4](#_Toc140228477)

[**Building Reliable Software: Quality Management Processes** 4](#_Toc140228478)

[**Building Reliable Software: Communication with the Client** 5](#_Toc140228479)

[**Building Reliable Software: Complexity** 5](#_Toc140228480)

[**Building Reliable Software: Change** 5](#_Toc140228481)

[**Building Reliable Software: Fault Tolerance** 6](#_Toc140228482)

[**Fault Tolerance: Recovery** 6](#_Toc140228483)

[**Building Reliable Software: Small Teams and Small Projects** 6](#_Toc140228484)

[**Reliability Metrics** 7](#_Toc140228485)

[**Reliability Metrics for Distributed Systems** 7](#_Toc140228486)

[**Metrics: User Perception of Reliability** 7](#_Toc140228487)

[**Reliability Metrics for Requirements** 8](#_Toc140228488)

[**Metrics: Cost of Improved Reliability** 8](#_Toc140228489)

[**Example: Central Computing System** 8](#_Toc140228490)

[**Building Reliable Systems: Two Principles** 9](#_Toc140228491)

[**Static and Dynamic Verification** 9](#_Toc140228492)

[**Static Verification: Reviews** 9](#_Toc140228493)

[**Reviews** 9](#_Toc140228494)

[**The Review Meeting** 10](#_Toc140228495)

[**Benefits of Reviews** 10](#_Toc140228496)

[**Successful Reviews** 10](#_Toc140228497)

[**Static Verification: Pair Design and Pair Programming** 10](#_Toc140228498)

[**Static Verification: Program Inspections** 11](#_Toc140228499)

[**Static Verification: Analysis Tools** 11](#_Toc140228500)

[**Dynamic Verification: Stages of Testing** 11](#_Toc140228501)

[**Testing Strategies** 12](#_Toc140228502)

[**Methods of Testing** 12](#_Toc140228503)

[**Testing: Unit Testing** 12](#_Toc140228504)

[**Testing: System and Sub-System Testing** 13](#_Toc140228505)

[**Dynamic Verification: Test Design** 13](#_Toc140228506)

[**Test Cases** 13](#_Toc140228507)

[**Variations in Test Sets** 13](#_Toc140228508)

[**Incremental Testing (e.g., Daily Testing)** 14](#_Toc140228509)

[**Dynamic Verification: Regression Testing** 14](#_Toc140228510)

[**Regression Testing: Program Testing** 14](#_Toc140228511)

[**Incremental Testing: an Example** 14](#_Toc140228512)

[**Documentation of Testing** 15](#_Toc140228513)

[**Fixing Bugs** 15](#_Toc140228514)

[**Moving the Bugs Around** 15](#_Toc140228515)

[**Difficult Bugs** 16](#_Toc140228516)

[**Bugs in Hardware** 16](#_Toc140228517)

[**When Fixing a Bug Creates a Problem for Customers** 16](#_Toc140228518)

[**Acceptance Testing** 17](#_Toc140228519)

[**Acceptance Testing in the Modified Waterfall Model** 17](#_Toc140228520)

[**Acceptance Testing with Iterative Development** 17](#_Toc140228521)

[**Acceptance Testing with Agile Development** 17](#_Toc140228522)

[**Resources for Acceptance Testing** 18](#_Toc140228523)

[**Acceptance Tests** 18](#_Toc140228524)

[**Techniques for Release** 18](#_Toc140228525)

[**Release: Parallel Testing** 18](#_Toc140228526)

[**Release: Alpha and Beta Testing** 19](#_Toc140228527)

[**Delivery: Summary** 19](#_Toc140228528)

[**Training** 19](#_Toc140228529)

[**Training and Usability** 20](#_Toc140228530)

[**Help Systems** 20](#_Toc140228531)

[**Categories of Documentation** 20](#_Toc140228532)

# **Bugs, Faults, and Failures**

* Bug (fault):
  + Programming or design error whereby the delivered system does not conform to speciﬁcation (e.g., coding error, protocol error)
* Failure:
  + Software does not deliver the service expected by the user (e.g., mistake in requirements, confusing user interface)

# **Failure of Requirements**

* An actual example
  + The head of an organization is not paid his salary because it is greater than the maximum allowed by the program. (Requirements problem)

# **Bugs and Features**

* That's not a bug. That's a feature!
  + Users will often report that a program behaves in a manner that they consider wrong, even though it is behaving as intended. That’s not a bug.
* That’s a failure!
  + The decision whether this needs to be changed should be made by the client not by the developers.

# **Terminology**

* Fault avoidance
  + Build systems with the objective of creating fault-free (bug - free) software.
* Fault detection (testing and verification)
  + Detect faults (bugs) before the system is put into operation or when discovered after release.
* Fault tolerance
  + Build systems that continue to operate when problems (bugs, overloads, bad data, etc.) occur.

# **Failures: A Case Study**

* A passenger ship with 1,509 persons on board grounded on a shoal near Nantucket Island, Massachusetts. At the time the vessel was about 17 miles from where the officer thought they were. The vessel was enroute from Bermuda to Boston.

# **Case Study: Analysis**

* From the report of the National Transportation Safety Board
  + The ship was steered by an autopilot that relied on position information from the Global Positioning System (GPS).
  + If the GPS could not obtain a position from satellites, it provided an estimated position based on Dead Reckoning (distance and direction traveled from a known point).
  + The GPS failed one hour after leaving Bermuda.
  + The crew failed to see the warning message on the display (or to check the instruments).
  + 34 hours and 600 miles later, the Dead Reckoning error was 17 miles.

# **Case Study: Software Lessons**

* All the software worked as specified(no bugs), but ...
  + After the GPS software was specified, the requirements changed (standalone system now part of integrated system).
  + The manufacturers of the autopilot and GPS adopted different design philosophies about the communication of mode changes.
  + The autopilot was not programmed to recognize valid/invalid status bits in messages from the GPS.
  + The warnings provided by the user interface were not sufficiently conspicuous to alert the crew.
  + The officers had not been properly trained on this equipment. Reliable software needs all parts of the software development process to be carried out well.

# **Building Reliable Software: Quality Management Processes**

* Assumption:
  + Good software is impossible without good processes
* The importance of routine:
  + Standard terminology (requirements, design, acceptance, etc.)
  + Software standards (coding standards, naming conventions, etc.)
  + Regular builds of complete system (often daily)
  + Internal and external documentation
  + Reporting procedures
* This routine is important for both heavyweight and lightweight development processes.
* When time is short...
  + Pay extra attention to the early stages of the process: feasibility, requirements, design.
  + If mistakes are made in the requirements process, there will be little time to fix them later.
  + Experience shows that taking extra time on the early stages will usually reduce the total time to release.

# **Building Reliable Software: Communication with the Client**

* A system is no use if it does not meet the client's needs
  + The client must understand and review the agreed requirements in detail. It is not sufficient to present the client with a specification document and ask him/her to sign off.
  + Appropriate members of the client's staff must review relevant areas of the design (including operations, training materials, system administration).
  + The acceptance tests must belong to the client.

# **Building Reliable Software: Complexity**

* The human mind can encompass only limited complexity
  + Comprehensibility
  + Simplicity
  + Partitioning of complexity
* A simple component is easier to get right than a complex one

# **Building Reliable Software: Change**

* Changes can easily introduce problems
  + Change management
  + Source code management and version control
  + Tracking of change requests and bug reports
  + Procedures for changing requirements specifications, designs and other documentation
  + Regression testing
  + Release control
* When adding new functions or fixing bugs it is easy to write patches that violate the systems architecture or overall program design. This should be avoided as much as possible.
* Be prepared to modify the architecture to keep a high quality system.

# **Building Reliable Software: Fault Tolerance**

* Aim:
  + A system that continues to operate when problems occur.
* Examples:
  + Invalid input data (e.g., in a data processing application)
  + Overload (e.g., in a networked system)
  + Hardware failure (e.g., in a control system)
* General Approach:
  + Failure detection
  + Damage assessment
  + Fault recovery
  + Fault repair

# **Fault Tolerance: Recovery**

* Backward recovery
  + Record system state at specific events (checkpoints). After failure, recreate state at last checkpoint.
  + Combine checkpoints with system log (audit trail of transactions) that allows transactions from last checkpoint to be repeated automatically.
* Recovery software is difficult to test
  + Example: After an entire network is hit by lightning, the restart crashes because of overload. (Problem of incremental growth.)

# **Building Reliable Software: Small Teams and Small Projects**

* Small teams and small projects have advantages for reliability
  + Small group communication cuts need for intermediate documentation, yet reduces misunderstanding.
  + Small projects are easier to test and make reliable.
  + Small projects have shorter development cycles. Mistakes in requirements are less likely and less expensive to fix.
  + When one project is completed it is easier to plan for the next.
* Improved reliability is one of the reasons that agile development has become popular over the past few years.

# **Reliability Metrics**

* Reliability
  + Probability of a failure occurring in operational use.
* Traditional measures for online systems
  + Mean time between failures
  + Availability (up time)
  + Mean time to repair
* Market measures
  + Complaints
  + Customer retention

# **Reliability Metrics for Distributed Systems**

* Traditional metrics are hard to apply in multi- component systems
  + A system that has excellent average reliability might give terrible service to certain users.
  + In a big network, at any given moment something will be giving trouble, but very few users will see it.
  + When there are many components, system administrators rely on automatic reporting systems to identify problem areas.

# **Metrics: User Perception of Reliability**

* Perceived reliability depends upon:
  + user behavior
  + set of inputs
  + pain of failure
* User perception is influenced by the distribution of failures
  + A personal computer that crashes frequently, or a machine that is out of service for two days every few years.
  + A database system that crashes frequently but comes back quickly with no loss of data, or a system that fails once in three years but data has to be restored from backup.
  + A system that does not fail but has unpredictable periods when it runs very slowly.

# **Reliability Metrics for Requirements**

* Example: ATM card reader

**A screenshot of a computer

Description automatically generated**

# **Metrics: Cost of Improved Reliability**

* Example. Many supercomputers average 10 hours productive work per day. How do you spend your money to improve reliability?

**A graph of a function

Description automatically generated**

# **Example: Central Computing System**

* A central computer system (e.g., a server farm) is vital to an entire organization (e.g., an Internet shopping site). Any failure is serious.
* Step 1: Gather data on every failure
  + Create a database that records every failure
  + Analyze every failure:
    - hardware software (default)
    - environment (e.g., power, air conditioning)
    - human (e.g., operator error)
* Step 3: Invest resources where benefit will be maximum, e.g.,
  + Priority order for software improvements
  + Changed procedures for operators
  + Replacement hardware
  + Orderly restart after power failure

# **Building Reliable Systems: Two Principles**

* For a software system to be reliable:
  + Each stage of development must be done well, with incremental verification and testing.
  + Testing and correction do not ensure quality, but reliable systems are not possible without thorough testing.

# **Static and Dynamic Verification**

* Static verification:
  + Techniques of verification that do not include execution of the software.
  + May be manual or use computer tools.
* Dynamic verification :
  + Testing the software with trial data.
  + Debugging to remove errors.

# **Static Verification: Reviews**

* Reviews are a form of static verification that is carried out throughout the software development process.

A diagram of a company

Description automatically generated

# **Reviews**

* Reviews are a fundamental part of good software development
* Concept
  + Colleagues review each other's work:
    - can be applied to any stage of software development, but particularly valuable to review program design or code
    - can be formal or informal
* Preparation
  + The developer(s) provides colleagues with documentation (e.g., models, specifications, or design), or code listing.
  + Participants study the materials in advance.
* Meeting
  + The developer leads the reviewers through the materials, describing what each section does and encouraging questions.

# **The Review Meeting**

* A review is a structured meeting
* Participants and their roles:
  + Developer(s): person(s) whose work is being reviewed
  + Moderator: ensures that the meeting moves ahead steadily
  + Scribe: records discussion in a constructive manner Interested parties: other developers on the same project
  + Outside experts: knowledgeable people who are not working on this project
  + Client: representatives of the client who are knowledgeable about this part of the process

# **Benefits of Reviews**

* Extra eyes spot mistakes, suggest improvements
* Colleagues share expertise; helps with training
* Incompatibilities between components can be identified
* Gives developers an incentive to tidy loose ends
* Helps scheduling and management control

# **Successful Reviews**

* To make a review a success:
  + Senior team members must show leadership
  + Good reviews require good preparation by everybody
  + Everybody must be helpful, not threatening Allow plenty of time and be prepared to continue on another day

# **Static Verification: Pair Design and Pair Programming**

* Concept: achieve benefits of review by shared development
* Two people work together as a team:
  + design and/or coding testing and system integration documentation and hand-over
* Benefits include:
  + two people create better software with fewer mistakes
  + cross training Many software houses report excellent productivity

# **Static Verification: Program Inspections**

* Formal program reviews whose objective is to detect faults
  + Code is read or reviewed line by line
  + 150 to 250 lines of code in 2 hour meeting
  + Use checklist of common errors
  + Requires team commitment, e.g., trained leaders
* So effective that it is claimed that it can replace unit testing

# **Static Verification: Analysis Tools**

* Program analyzers scan the source of a program for possible errors and anomalies.
  + Control flow: Loops with multiple exit or entry points
  + Data use: Undeclared or uninitialized variables, unused variables, multiple assignments, array bounds Interface faults:
  + Parameter mismatches, non-use of functions results, uncalled procedures
  + Storage management: Unassigned pointers, pointer arithmetic
* Static analysis tools
  + Cross-reference table: Shows every use of a variable, procedure, object, etc.
  + Information flow analysis: Identifies input variables on which an output depends.
  + Path analysis: Identifies all possible paths through the program.

# **Dynamic Verification: Stages of Testing**

* Testing is most effective if divided into stages
  + User interface testing
  + Unit testing
    - unit test
  + System testing
    - integration test
    - function test
    - performance test
    - installation test
  + Acceptance testing (carried out separately)

# **Testing Strategies**

* Bottom-up testing
  + Each unit is tested with its own test environment. Used for all systems.
* Top-down testing
  + Large components are tested with dummy stubs. Particularly useful for: user interfaces work-flow client and management demonstrations
* Stress testing
  + Tests the system at and beyond its limits. Particularly useful for:
    - real-time systems
    - transaction processing
  + Most systems require a combination of all three strategies.

# **Methods of Testing**

* Open box testing
  + Testing is carried out by people who know the internals of what they are testing.
  + Example: Tick marks in a graphing package
* Closed box testing
  + Testing is carried out by people who do not know the internals of what they are testing.
  + Example: Educational demonstration that was not foolproof

# **Testing: Unit Testing**

* Tests on small sections of a system, e.g., a single class
* Emphasis is on accuracy of actual code against specification
* Test data is usually chosen by developer(s) based on their understanding of specification and knowledge of the unit
* Can be at various levels of granularity
* Can be open box or closed box: by the developer(s) of the unit or by special testers
* If unit testing is not thorough, system testing becomes almost impossible. If you are working on a project that is behind schedule, do not rush the unit testing.

# **Testing: System and Sub-System Testing**

* Tests on components or complete system, combining units that have already been thoroughly tested
* Emphasis on integration and interfaces
* Trial data that is typical of the actual data, and/or stresses the boundaries of the system, e.g., failures, restart
* Carried out systematically, adding components until the entire system is assembled
* Can be open or closed box: by development team or by special testers
* System testing is finished fastest if each component is completely debugged before assembling the next

# **Dynamic Verification: Test Design**

* Testing can never prove that a system is correct. It can only show that either (a) a system is correct in a single case, or (b) that it has an error.
  + The objective of testing is to find errors or demonstrate that program is correct in specific instances.
  + Testing is never comprehensive.
  + Testing is expensive.

# **Test Cases**

* Test cases are specific tests that are chosen because they are likely to find specific problems. Test cases are chosen to balance expense against chance of finding serious errors.
  + Cases chosen by the development team are effective in testing known vulnerable areas.
  + Cases chosen by experienced outsiders and clients will be effective in finding gaps left by the developers.
  + Cases chosen by inexperienced users will find other errors.

# **Variations in Test Sets**

* A test suite is the set of all test cases that apply to a system or component of a system.
* When running tests, there are some errors that occur only under certain circumstances, e.g., when certain other software is running on the same machine, when tasks are scheduled in specific sequences, or with unusual data sets, etc.
* Therefore it is customary for each test run to vary some of the test cases systematically, and to change the order in which the tests are made, etc.

# **Incremental Testing (e.g., Daily Testing)**

* Spiral development and incremental testing
  + Create a first iteration that has the structure of the final system and some basic functionality.
  + Create an initial set of test cases.
  + Check-in changes to the system on a daily basis, rebuild entire system daily.
  + Run a comprehensive set of test cases daily, identify and deal with any new errors.
  + Add new test cases continually.
* Many large software houses, e.g., Microsoft, follow this procedure with a daily build of the entire system and comprehensive sets of test cases. For a really big system this may require hundreds or even thousands of test computers and a very large staff of testers.

# **Dynamic Verification: Regression Testing**

* When software is modified, regression testing is used to check that modifications behave as intended and do not adversely affect the behavior of unmodified code.
* After every change, however small, rerun the entire testing suite.

# **Regression Testing: Program Testing**

* Collect a suite of test cases, each with its expected behavior.
* Create scripts to run all test cases and compare with expected behavior. (Scripts may be automatic or have human interaction)
* When a change is made to the system, however small (e.g., a bug is fixed), add a new test case that illustrates the change (e.g., a test case that revealed the bug).
* Before releasing the changed code, rerun the entire test suite.

# **Incremental Testing: an Example**

* Example
  + A graphics package consisting of a pre-processor, a runtime package (set of classes), and several device drivers.
* Starting point
  + A prototype with a good overall structure, and most of the functionality, but hastily coded and not robust.
* Approach
  + On a daily cycle:
    - Design and code one small part of the package (e.g., an interface, a class, a dummy stub, an algorithm within a class)
    - Integrate into prototype.
    - Create additional test cases if needed.
    - Regression test.

# **Documentation of Testing**

* Every project needs a test plan that documents the testing procedures for thoroughness, visibility, and for future maintenance.
* The test plan should include:
  + Description of testing approach.
  + List of test cases and related bugs.
  + Procedures for running the tests.
  + Test analysis report.

# **Fixing Bugs**

* Isolate the bug
  + Intermittent 🡪 repeatable
  + Complex example 🡪 simple example
* Understand the bug and its context
  + Root cause
  + Dependencies
  + Structural interactions
* Fix the bug
  + Design changes
  + Documentation changes
  + Code changes
  + Create new test case

# **Moving the Bugs Around**

* Fixing bugs is an error-prone process
  + When you fix a bug, fix its environment. Bug fix need static and dynamic testing.
  + Repeat all tests that have the slightest relevance (regression testing).
* Bugs have a habit of returning
  + When a bug is fixed, add the failure case to the test suite for future regression testing.
* Persistence
  + Most people work around a problem. The best people track down the root cause and ?x it forever!

# **Difficult Bugs**

* Some bugs may be extremely difficult to track down and isolate. This is particularly true of intermittent failures.
  + A large central computer stops a few times every month with no dump or other diagnostic.
  + A database load dies after running for several days with no diagnostics.
  + An image processing system runs correctly, but uses huge amounts of memory.
* Such problems may require months of effort to track down.

# **Bugs in Hardware**

* Three times in my career I have encountered hardware bugs:
  + The film plotter with the missing byte (1:1023)
  + Microcode for virtual memory management
  + The Sun page fault Each problem was actually a bug in embedded software/firmware

# **When Fixing a Bug Creates a Problem for Customers**

* Sometimes customers will build applications that rely upon a bug. Fixing the bug will break the applications.
  + The graphics package with rotation about the Z-axis in the wrong direction.
  + An application crashes with an emulator, even though the emulator is bug free. (Compensating bug problem)
  + The 3-pixel rendering problem with Internet Explorer.
* With each of these bugs the code was easy to fix, but releasing it would have caused problems for existing programs.

# **Acceptance Testing**

* The complete system, including documentation, training materials, installation scripts, etc. is tested against the requirements by the client, assisted by the developers.
  + Each requirement is tested separately.
  + Scenarios are used to compare the expected outcomes with what the system does.
  + Emphasis is placed on how the system handles problems, errors, restarts, and other difficulties.
  + Is the system we have built, the system that you wanted? Does it meet your requirements?

# **Acceptance Testing in the Modified Waterfall Model**

A diagram of a program

Description automatically generated

# **Acceptance Testing with Iterative Development**

* Iterative development If the client is properly involved in the development cycle:
  + The client will have tested many parts of the system, e.g., 🡪 At the end of each iteration 🡪 During user testing
  + Problems and suggestions for improvement will have been incorporated into the system.
* BUT: There must still be an acceptance test of the final system before it is released.

# **Acceptance Testing with Agile Development**

* Acceptance testing is particularly important with agile development, since each sprint should end with fully tested code.
* Each sprint should be a complete development process, ending with acceptance testing by the client.
* If several sprints build on each other, each sprint may need to repeat the acceptance tests for earlier sprints to check that they are still met.

# **Resources for Acceptance Testing**

* Acceptance Testing is a important part of a software project
  + It requires time on the schedule
  + It may require substantial investment in test data, equipment, and test software.
  + Good testing requires good people.
  + Help systems and training materials are important parts of acceptance testing.

# **Acceptance Tests**

* Closed box by the client without knowledge of the internals
* The entire system is tested as a whole
* The emphasis is on whether the system meets the requirements
* The tests should use real data in realistic situations, with actual users, administrators, and operators
* The acceptance tests must be successfully completed before the new system can go live or replace a legacy system. Completion of the acceptance tests may be a contractual requirement before the system is paid for.

# **Techniques for Release**

* The transition from the previous version of a production system to a new release is challenging.
* Parallel Testing: Clients operate the new system alongside the old production system with same data and compare results
* Alpha Testing: Clients operate the system in a realistic but non- production environment
* Beta Testing: Clients operate the system in a carefully monitored production environment

# **Release: Parallel Testing**

* For data processing systems, such as financial systems, payroll, etc., the old and the new systems are run together for several productions cycles to check that the new system replicates the functionality of the old.
  + Requires two sets of everything (staff, equipment, etc.).
  + Requires software to control changeover (e.g., do not mail two sets of payments).
  + Requires automated scripts to compare results. Parallel testing may take several months. Often, the new system will be brought into production in phases.

# **Release: Alpha and Beta Testing**

* Alpha testing can be done with software that lacks some functionality.
* Beta testing requires fully functional system.
* Alpha and Beta testing must be managed
* If you simply make versions of the software available to clients:
  + Many clients will never use it.
  + Other clients will use a few features.
  + Only a few clients will test it systematically.
  + Only a few clients will report problems systematically. What incentives can you give clients to test your software for you?
  + Financial (e.g., discounts on products)
  + Prestige (e.g., recognition, publicity)

# **Delivery: Summary**

* A good delivery package results in
  + happy client
  + happy users
  + less expense in support and maintenance
* But many projects rush the packaging, help systems, and training materials, give them to the least experienced members of the team, do not test them properly, and generally neglect this part of the software process.

# **Training**

* Time and money spent on training is usually well spent:
  + one-on-one
  + in-house training
  + training courses
  + distance education
  + online tutorials
* Development team provides information for training materials:
  + users (perhaps several categories)
  + system administrators
  + system maintainers
  + trainers

# **Training and Usability**

* A well-designed system needs less training
  + good conceptual model
  + intuitive interfaces
* Different skill levels need different types of training
  + skilled users work from the conceptual model
  + less-skilled users prefer cookbook sets of instructions
  + occasional users will forget complex details, but remember general structure

# **Help Systems**

* Resources
  + A good help system is a major sub-project (time- consuming, expensive)
  + A good help system saves user time and support sta? (time-saving, cost-saving)
* Help system design
  + Users need many routes to find information (index by many terms, examples, mini-tutorials, etc.)
  + Help systems need to be tested with real users

# **Categories of Documentation**

* Software development
  + Requirements, design
  + Source code, test plans and results
* User
  + Introductory (various audiences)
  + User manual
  + Web site of known problems, FAQ, etc.
* System administrator and operator
  + System manuals Business
  + License, contract, etc.