CS 361
Software Engineering I

Testing

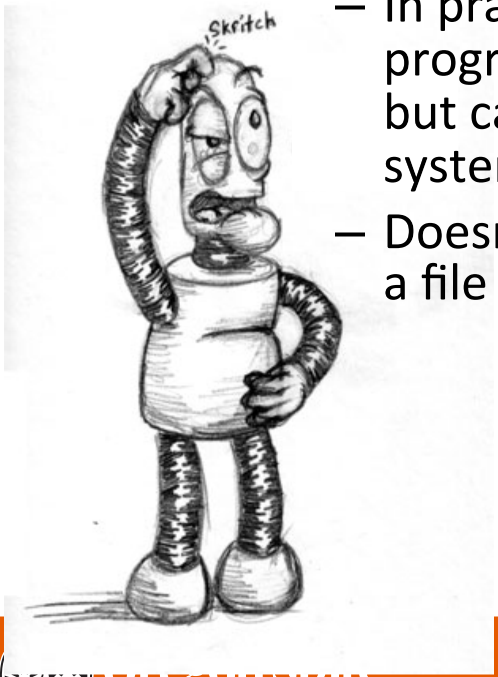
Oregon State University

Why Testing?

- Ideally: we' d *prove code* correct, using formal mathematical techniques (with a computer, not chalk)
  - Extremely difficult: for some trivial programs (100 lines) and many small (5K lines) programs
  - Simply not practical to prove correctness in most cases – often not even for safety or mission critical code

Why Testing?

- Nearly ideally: use symbolic or abstract *model checking* to *prove that a model* is correct
  - Automatically extract a mathematical abstraction from code
  - Prove properties with model over all possible executions

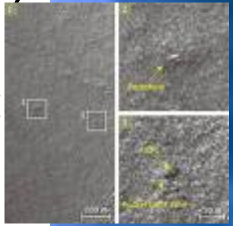


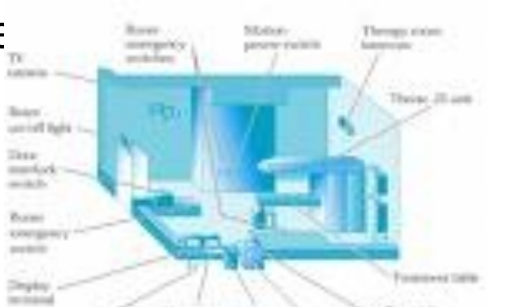
  - In practice, can work well for very simple properties ("this program never crashes in this particular way"), of some programs, but can't handle complex properties ("this is a working file system")
  - Doesn't work well for programs with complex data structures (like a file system)

Testing saves lives and money

- NIST report, "The Economic Impacts of Inadequate Infrastructure for Software Testing" (2002)
  - Inadequate software testing costs the US alone between $22 and $59 billion annually
  - Better approaches could cut this amount in half
- Major failures: Ariane 5 explosion, Mars Polar Lander, Intel's Pentium FDIV bug
- Insufficient testing of safety-critical software can cost *lives*: THERAC-25 radiation machine: 3 dead
- We want our programs to be reliable
  - Testing is how, in most cases, we find out if they are

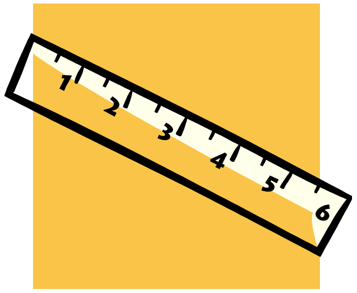
Ariane 5: exception-handling bug : forced self destruct on maiden flight (64-bit to 16-bit conversion: about 370 million $ lost)


Mars Polar Lander crash site?


THERAC-25 design

Terms: Coverage

- Coverage measures or metrics
  - Abstraction of “what a test suite tests” in a structural sense
  - Common measures:
    - Statement coverage
      - A.k.a line coverage or basic block coverage
      - Which statements execute in a test suite
    - Decision coverage
      - Which boolean expressions in control structures evaluated to both true and false during suite execution
    - Path coverage
      - Which paths through a program’s control flow graph are taken in the test suite
    - Mutation coverage
      - Ability to detect random variations to the code



Terms: Black Box Testing

- Black box testing
  - Treats a program or system as a
  - That is, testing that does *not* look at source code or internal structure of the system
  - Send a program a stream of inputs, observe the outputs, decide if the system passed or failed the test
  - Abstracts away the internals – a useful perspective for integration and system testing
  - Sometimes you don' t have access to source code, and can make little use of object code
    - True black box? Access only over a network

Terms: White Box Testing

- White box testing
  - Opens up the box!
    - (also known as glass box, clear box, or structural testing)
  - Use source code (or other structure beyond the input/output spec.) to design test cases

Stages of Testing

- Unit testing is the first phase, done by developers of modules
- Integration testing combines unit-tested modules and tests how they interact
- System testing tests a whole program to make sure it meets requirements
- Acceptance testing by users to see if system meets actual use requirements

Stages of Testing: Unit Testing

- Unit testing is the first phase, mostly done by developers of modules
  - Typically the earliest type of testing done
  - Unit could be as small as a single function or method
  - Often relies on stubs to represent other modules and incomplete code
  - Tools to support unit tests available for most popular languages,
    - Junit (http://junit.org)
    - Simpletest for PHP (http://simpletest.org)

Stages of Testing: Integration Testing

- Integration testing combines unit-tested modules and tests how they interact
  - Relies on having completed units
  - After unit testing, before system testing
  - Test cases focus on interfaces between components, and assemblies of multiple components
  - Often more formal (test plan presentations) than unit testing

Stages of Testing: System Testing

- System testing tests a whole program to make sure it meets requirements
  - After integration testing
  - Focuses on "breaking the system"
  - Defects in the completed product, not just in how components interact
  - Checks quality of requirements as well as the system
  - Often includes stress testing, goes beyond bounds of well-defined behavior

Stages of Testing: Acceptance Testing

- Acceptance testing by users to see if system meets actual use requirements
  - Black box testing
  - By *end-users* to determine if the system produced really meets their needs
  - May revise requirements/goals as much as find bugs in the code/system

Appropriate at all times: Regression Testing

- Regression testing
  - Changes can break code, reintroduce old bugs
    - Things that used to work may stop working (e.g., because of another "fix") – software regresses
  - Usually a set of cases that have failed (& then succeeded) in the past
  - Finding small regressions is an ongoing research area – analyze dependencies

". . . as a consequence of the introduction of new bugs, program maintenance requires far more system testing. . . . Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such regression testing must indeed approximate this theoretical idea, and it is very costly."
- Brooks, The Mythical Man-Month

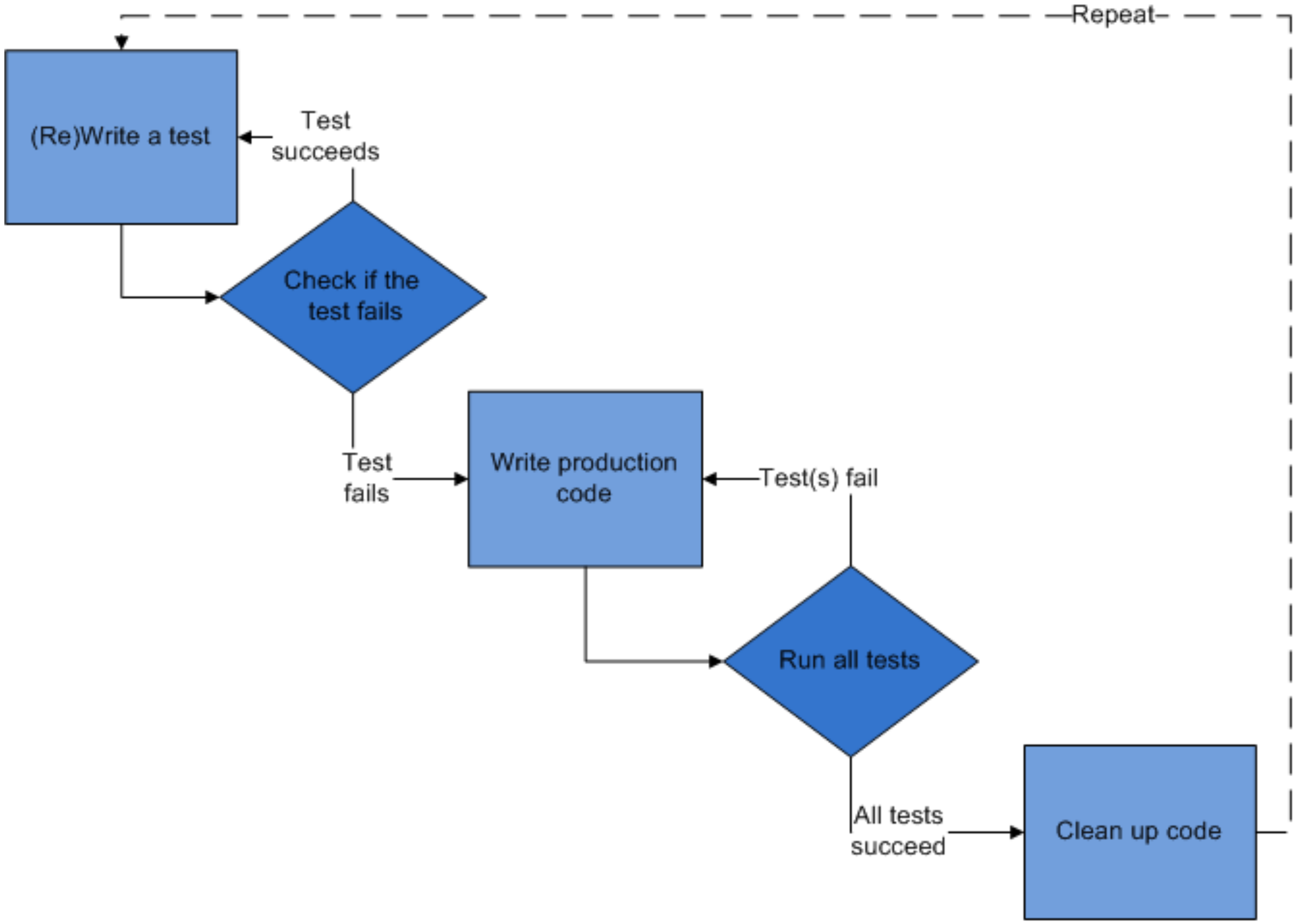
Test-Driven Development

- One way to make sure code is tested as early as possible is to *write test cases before the code*
  - Idea arising from Extreme Programming and often used in agile development
    - Write (automated) test cases first
    - Then write the code to satisfy tests

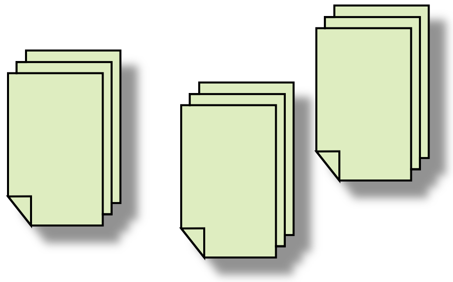
Test-Driven Development

- How to add a feature to a program, in test-driven development
  - Add a test case that *fails*, but would succeed with the new feature implemented
  - Run *all* tests, make sure only the new test fails
  - Write code to implement the new feature
  - Rerun *all* tests, making sure the new test succeeds (and no others break)

Test-Driven Development Cycle



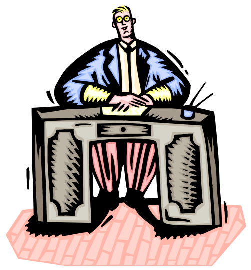
Test-Driven Development Benefits



- Results in lots of useful test cases
  - A very large *regression* set
- Forces attention to actual behavior of software: observable & controllable behavior
- Only write code as needed to pass tests
  - And may get good *coverage* of paths through the program, since they are written in order to pass the tests
  - Reduces temptation to tailor tests to idiosyncratic behaviors of implementation
- Testing is a first-class activity in this kind of development

Test-Driven Development Problems

- Need institutional support
  - Difficult to integrate with a waterfall development
  - Management may wonder why so much time is spent writing tests, not code



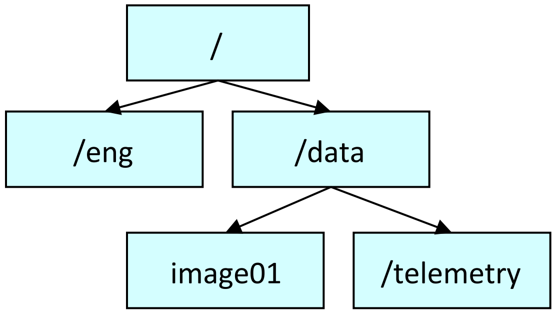
- Lots of test cases may create false confidence
  - If developers have written all tests, may be blind spots due to false assumptions made in coding and in testing, which are tightly coupled

Exhaustive vs. Representative Testing

- Can we test *everything*?
- File system is a library, called by other components of some flight software

Operation	Result
mkdir ("/eng", …)	SUCCESS
mkdir ("/data", …)	SUCCESS
creat ("/data/image01", …)	SUCCESS
creat ("/eng/fsw/code", …)	ENOENT
mkdir ("/data/telemetry", …)	SUCCESS
unlink ("/data/image01")	SUCCESS

File system



Example: File System Testing

- How hard would it be to just try "all" the possibilities?
- Consider only core 7 operations (`mkdir`, `rmdir`, `creat`, `open`, `close`, `read`, `write`)
  - Most of these take either a file name or a numeric argument, or both
  - Even for a "reasonable" (but not provably safe) limitation of the parameters, there are 266^{10} executions of length 10 to try
  - Not a realistic possibility (unless we have 10^{12} years to test)

Oregon State University

Not Testing: Code Reviews

- Not testing, exactly, but an important method for finding bugs and determining the quality of code
  - Code walkthrough: developer leads a review team through code
    - Informal, focus on code
  - Code inspection: review team checks against a list of concerns
    - Team members prepare offline in many cases
    - Team moderator usually leads

Not Testing: Code Reviews

- Code inspections have been found to be one of the most effective practices for finding faults
  - Some experiments show removal of 67-85% of defects via inspections
  - Some consider XP's *pair programming* as a kind of "code review" process, but it's not quite the same
    - Why?
  - Can review/walkthrough requirements and design documents, not just code!

Oregon State University