

GOAL

- An overarching goal for software development is to satisfy your customer
 - This is one definition of *quality*
- This means having confidence that what you build is what the customer wanted
- There are various approaches for raising confidence; none that provide any guarantees

Definitions

- *Verification*: has the program been implemented correctly?
 - *Are we building the program right?*
 - Process question
- *Validation*: has the correct program been implemented?
 - *Are we building the right program?*
 - Product question
- Collectively, they are called V&V

Definitions - 2

- The most common way of raising confidence and improving quality is testing
- *Testing*: the process of executing a program with the intent of detecting problems
 - Can be used for either verification or validation

Definitions - 3

- *Failure*: a condition arising when executing software produces an unexpected result
 - Incorrect output, infinite loop, program crash
- *Fault/bug/defect*: the manifestation of an error in the software
 - May or may not lead to a failure
- *Error*: a (human) mistake made in the process of constructing software

Approaches to V&V

- Testing
 - Exercising software to try and generate failures
- Inspection / review / walkthrough
 - Systematic group review of program text to detect faults
- Formal proof
 - Mathematical proof that the program text implements the program specification

Activity Comparison

Technique	Purpose	Danger
<i>Testing</i>	Create failures	Inadequate test set
<i>Proof</i>	Prove <i>correctness</i>	Functionality only
<i>Review</i>	Raise issues	Informality
<i>Debugging</i>	Locate and correct defects	Local fixes; Defect introduction
<i>Reliability</i>	Estimate failure likelihood	Inadequate probability model

Kinds of Testing

- Many terms are used to describe different kinds of testing
- Some indicate when the testing is done and some indicate the technique that is applied

Testing and the Software Life Cycle

Requirements *Acceptance* Testing

Design *Integration* Testing

Coding *Unit* Testing

Maintenance *Regression* Testing

Development Cycle Testing

- *Acceptance Testing*: validation testing that developed software meets customer requirements
 - May be performed in-house (*system testing*) or by customer
- *Integration Testing*: verification testing the independently developed component behave correctly when combined
 - May be performed incrementally or all at once
- *Unit Testing*: verification testing on an individual component
 - Usually done by the developer of the component
- *Regression Testing*: validation testing that maintenance changes haven't broken any preexisting features

Delivery Testing

- *α (alpha) test*
 - Informal, in-house, early
- *β (beta) test:*
 - Formal, out-of-house, pre-release
- *Sanity check (smoke test):*
 - Quick-and-dirty, just before shipping

Test Planning

- Testing is typically the most expensive part of software development. Therefore, it is essential that testing be carefully planned to make effective use of tester and machine time
- This means developing a test plan
- For significant efforts, the plan itself should be reviewed

Test Plan Document

- Adequacy criterion
- For each test
 - What is the objective of the test?
 - How will you know if the objective has been met?
 - The input to use
 - Expected output
 - The actual output obtained
- Testing strategy

Test Adequacy

- Test sets, like programs, are designed, to accomplish a goal—to adequately exercise a program
- *Adequacy criterion*: precise, deterministic rule for deciding beforehand how you will know when to stop testing
 - Alternatively—how do you know when you have a good suite of tests?
- Examples
 - Every statement executed
 - Every requirement tested
 - Error-based tests
 - Reliability based criterion

Test Strategy

- The order in which tests are performed is closely tied to the development process used
- Two primary questions arise
 - Whether the testing is done while implementation is underway or after it is complete?
 - In what order are the tests applied?

Incremental vs. Big Bang Testing

- *Incremental*: try to test as each new component is added so that you know what to concentrate on if a failure occurs
 - Discovers problems earlier thereby reducing risk
- *Big bang*: testing phase independent from and started after implementation
 - Can focus resources and reduce thrashing
- These days, the former is used almost exclusively

Ordering Tests

- The order in which tests are performed can be tied to the development process used
- *Top down*: test the main routine plus stubs; then replace a stub by code (+ lower-level stubs) and integrate
 - Advantage: always have a complete system
 - Disadvantages: stub generation; meaningless output
- *Bottom up*: test modules separately, then combine
 - Advantage: meaningful values generated
 - Disadvantages: requires harness; hides integration problems until late

Ordering Tests - 2

- The order in which tests are performed can be tied to the system architecture
- *Horizontal*: if the system can be thought of as having a layered architecture, then each layer can be thought of as a virtual machine and tested independently
 - Advantage: modularity; loose coupling
 - Disadvantages: virtual machine specification; integration of machines
- *Vertical*: organize tests around features/user stories where each test takes input from the user and returns output to the user
 - Advantages: reliability computation; works well with feature-oriented development
 - Disadvantages: unanticipated uses; reduced attention to infrastructure

Testing Techniques

- There are two main categories of testing techniques that differ depending on whether the source code is used to control the testing process
 - *Black box (functional)*: look only at the behavior of the program
 - *White box (clear box)*: base the testing on the structure of the code
- In addition, error-based testing focuses on the kinds of mistakes actually made by programmers

Black Box Testing

- Generate tests to check desired program functionality without knowledge of how the code works
- Examples
 - *Equivalence partitioning*: divide possible test sets into equivalence classes, and run one test from each class
 - *Boundary value analysis*: look for test cases on the boundaries of the equivalence classes
 - *Exhaustive testing*: run every possible input (infeasible)
 - *Functional testing*: construct tests directly from the requirements document
 - *Random*: if the tests are randomly generated with a distribution that corresponds to the expected usage of the program, then you can get a reliability estimate to use as an adequacy criterion

Two Black-Box Error Estimation Strategies

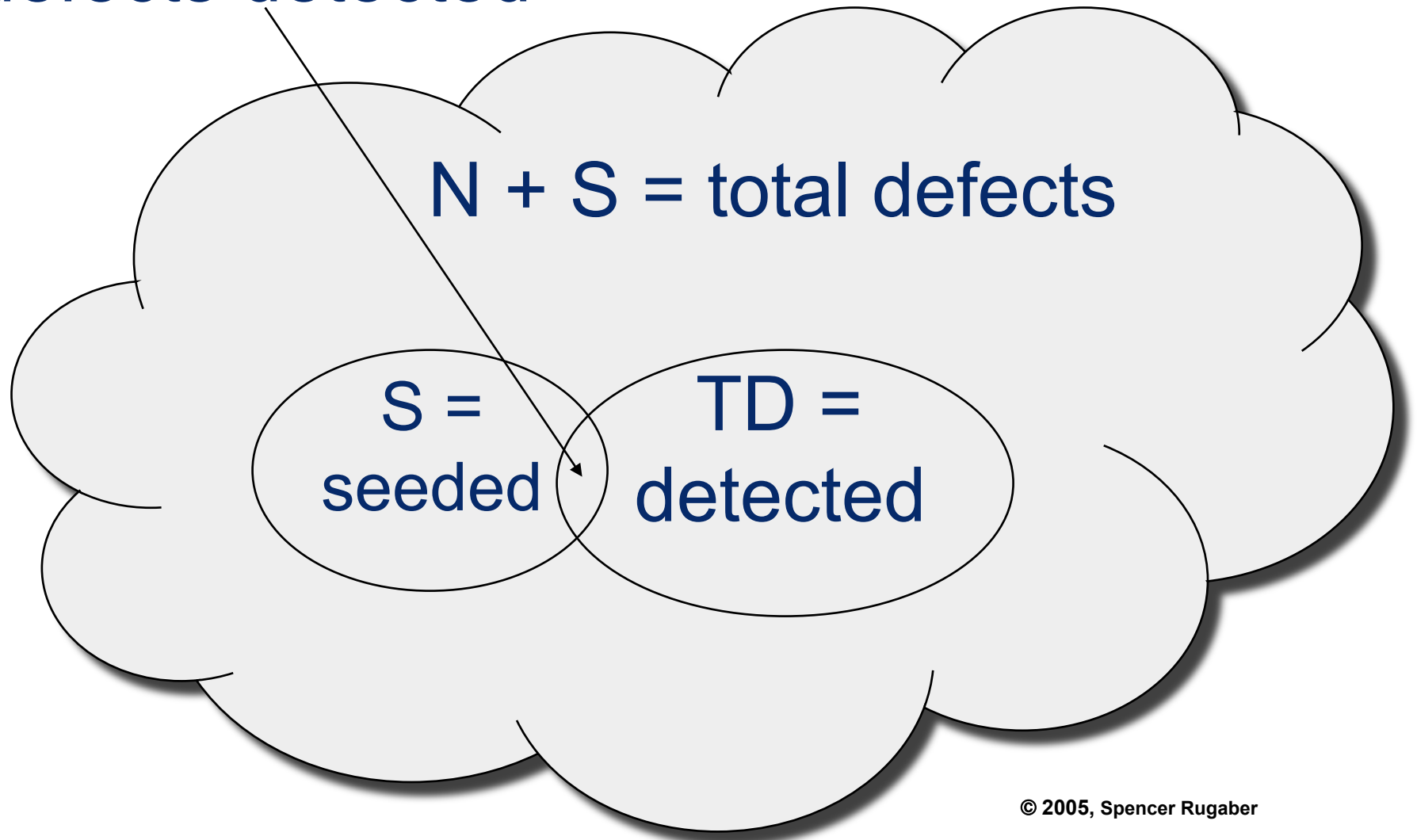
- *Bebugging*
 - Insert a known set of defects
 - Test
 - Measure the percent of inserted defects detected
 - Use to estimate the number of remaining defects
- Parallel test groups
 - Separate teams separately test same software
 - Use percentage of duplicate defects detected to estimate remaining defects

"Error" Seeding / Bebugging

- Intentionally seed “typical” faults
- Try to estimate remaining, undetected faults by using percentage of seeded ones that were detected
- That is, if you detect x % of the seeded bugs, then estimate that you have detected x % of all bugs
- Computation
 - S = seeded defects
 - SD = seeded defects detected
 - UD = unseeded defects detected
 - TD = total defects detected (SD + UD)
 - N = total unseeded defects
 - $(SD / S) \cong (UD / N) \Rightarrow N = (S * UD / SD)$
 - R = remaining undetected defects = N - UD
 - P = percent completion = $100 * UD / N$

Bebugging Venn Diagram

SD = seeded
defects detected



Bebugging - Example Computation

- $S = 100$ // seed one hundred defects
- $TD = 40$ // test and detect 40 defects
- $SD = 5$ // of which five are seeded
- $UD = TD - SD = 40 - 5 = 35$ // detected unseeded
- $N \cong (S * UD / SD) = (100 * 35) / 5 = 700$
- $R = N - UD = 700 - 35 = 665$ // remaining
- $P = 100 * (UD / N) = 100 * (35 / 700) = 5\%$

Parallel Testing Groups

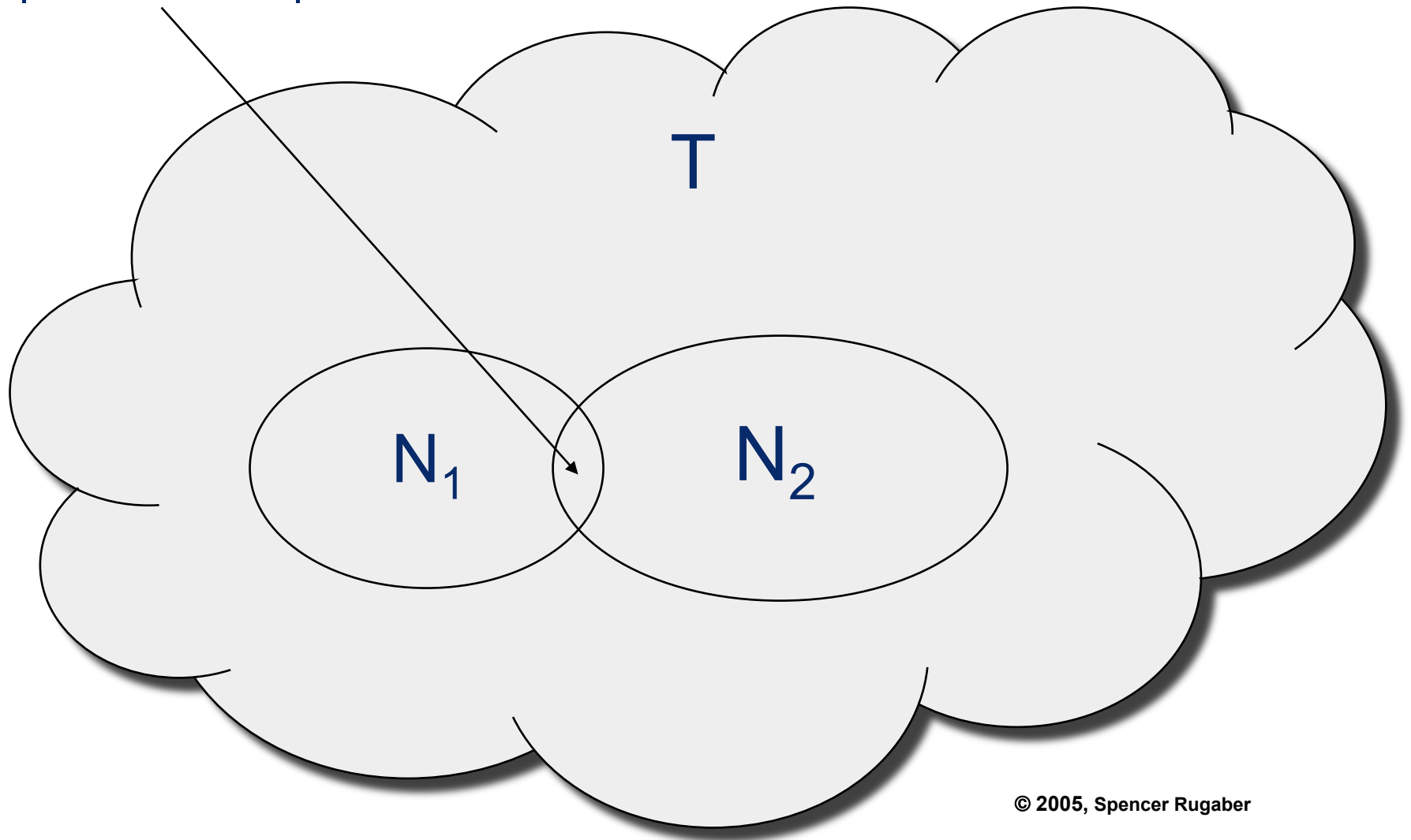
- Same program tested by two groups
- Use faults found by both as an indication of completeness
- Symbols
 - B = number found by both groups
 - N_1 = number found by first group
 - N_2 = number found by second group
 - T = estimated total defects

Rationale

- The first group has found some percentage of the total defects: (N_1 / T)
- Part of the ones that the second group found will be within the set found by the first group. That proportion is (B / N_2)
- This ratio should be a good estimate of the size of N_1 relative to T (i.e. N_1 / T)
- That is, $(B / N_2) \cong (N_1 / T)$
- Solving for $T = (N_1 * N_2) / B$

Parallel Testing Graphic

$$B = |N_1 \cap N_2|$$



Parallel Testing Example

- N_1 = number found by first group = 70
- N_2 = number found by second group = 100
- B = number found by both groups = 10
- T = estimated total defects =
$$(N_1 * N_2) / B = (70 * 100) / 10 = 700$$

White Box Testing - Coverage

- Adequacy based on control or data flow properties
- *Statement coverage*: % of all executable statements exercised during testing
 - Testing a statement once does not guarantee that the statement is correct
 - Some statements may be *dead code*
- *Branch coverage*: % of conditional branches exercised
 - Do tests exercise both the `then` and `else` branches of each `if` statement even if one of them is empty?

Coverage - 2

- *Condition coverage*: % of boolean expression constituents exercised
 - Was there a test that caused the `then` branch to be taken because `a` was true? Because `b` was true?

```
if (a || b) then ... else ...
```
- *Data flow coverage* measures
 - For a given use of a variable and for each assignment to that variable that could *reach* that use, is there a test that causes execution to proceed from the assignment to the use?
- *Path coverage*: % of program paths executed (infeasible or impossible)

Error-Based Testing

- Look for evidence of a specific class of errors, such as *off-by-one*, known to occur in programs
- Adequacy is the extent of possible error opportunities evaluated
- Mutation analysis
 - Generate a set of program variants (*mutants*) containing a comprehensive set of intentionally seeded bugs representative of typical errors
 - Compose tests sufficient to distinguish the generated programs from the original (*kill the mutant*)

Non-Functional Tests

- Performance
- Load
- Usability
- Security

Testing Tools

- Harnesses
- Stub generators
- Unit testing frameworks
- Comparators / regression testing
- Test data generators
- Coverage measurers

Static Analyses

- Anomaly detection
 - Unreachable code, uninitialized variables, uncalled functions, signature matching, unreferenced variables, function value not set before return, result of function call not used, variable set but not used
 - Conservative estimates
- Symbolic execution
- Coding standards
- Maintainability / complexity metrics
- Portability checks (**lint**)

Testing Guidelines

- *Program testing can best show the presence of errors but never their absence* - Dijkstra
- 50% of program development effort goes into testing, primarily integration testing
- *The act of designing tests is one of the most effective error prevention mechanisms known* - Beizer

More Guidelines

- Effective testing requires a different attitude (destructive) from effective programming (constructive) and should be performed by different kinds of people
- Test that a program doesn't do what it isn't supposed to do as well as doing what it is supposed to do
- Test how well the program deals with erroneous input and internally detected failures

Guidelines

(Pressman)

- Tests should be traceable to customer requirements
- Tests should be planned long before testing begins
 - *Test-first* strategy of Extreme Programming
- The Pareto principle applies to software testing
 - 80% of defects will be traced to 20% of modules
- To be most effective, testing should be conducted by an independent third party