COMP 3111
SOFTWARE ENGINEERING

LECTURE 15
TESTING

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ➡ **Implement Tests**
- Perform Tests
  - Unit
  - Integration
  - System
- Evaluate Tests

IMPLEMENT TESTS

Goal: To automate test procedures as much as possible.

- Running test cases can be very tedious and time consuming!
  - There are many possible input values and system states to test.

- A **test component** is a program that automates one or several test procedures or parts of them.

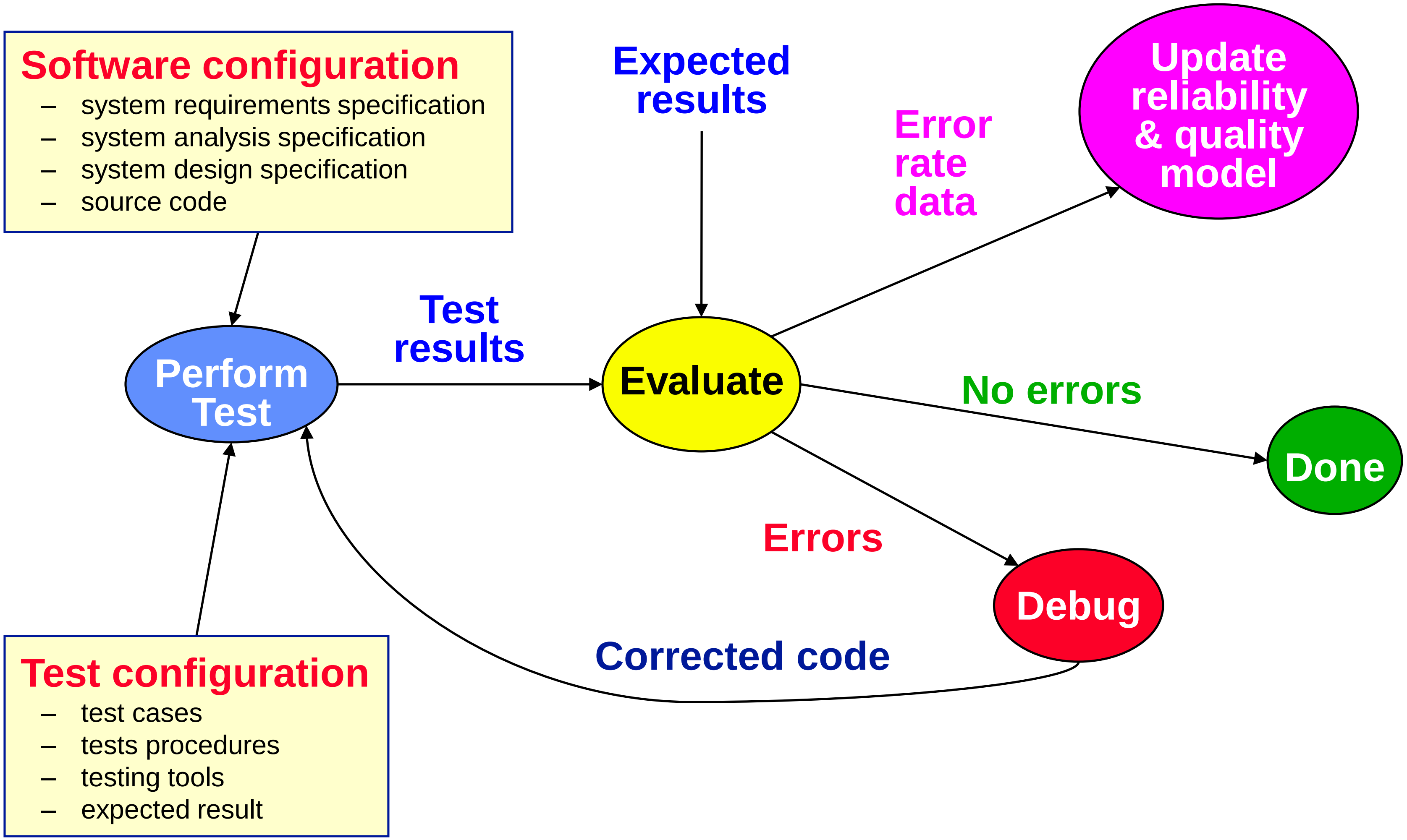
- There are tools available to help write test components that:
  - record the actions for a test case as the user performs the actions.
  - parameterize the recorded script to accept a variety of input values.

☞ **Spreadsheets and/or database applications can be used to store the required input data and the results of each test.**

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ✓ Implement Tests
- ➡ **Perform Tests**
  - – Unit
  - – Integration
  - – System
- Evaluate Tests

PERFORM TESTS



Software configuration

- system requirements specification
- system analysis specification
- system design specification
- source code

Expected results

Error rate data

Update reliability & quality model

Test results

Perform Test

Evaluate

No errors

Done

Errors

Debug

Corrected code

Test configuration

- test cases
- tests procedures
- testing tools
- expected result

PERFORM TESTS: TESTING STRATEGY

A *testing strategy* specifies which testing techniques (white box, black box, etc.) are appropriate at which point in time.

- A testing strategy integrates test cases into a well-planned series of steps that test a component/subsystem by specifying:
  - What are the steps that need to be conducted to test a component?
  - When are the steps planned and undertaken?
  - How much effort, time and resources will be required to do the steps?

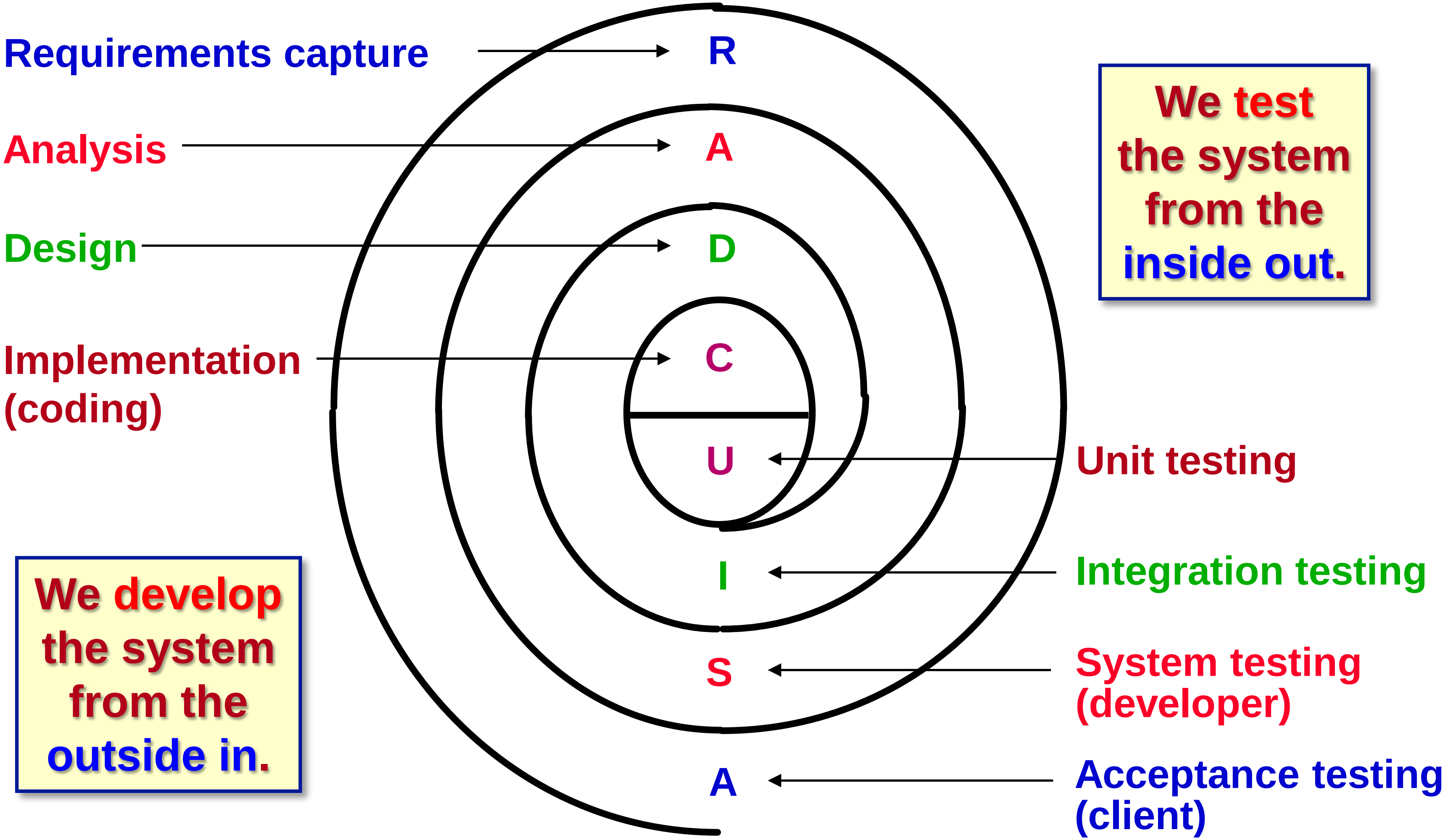
However, test planning is difficult due to the time uncertainty in the debug part!

☞ **Consequently, a testing strategy needs to balance:**

flexibility & creativity with planning & management

Testing often occurs when deadline pressures are most severe.
Progress must be measurable and problems identified as early as possible.

A TESTING STRATEGY



We test the system from the inside out.

We develop the system from the outside in.

A TESTING STRATEGY

Who should do the testing?

Unit Testing (using White Box and Black Box test cases)

- Verifies that each component/subsystem functions correctly.
  - Done by software engineer who develops the code.

Integration Testing (using White Box and Black Box test cases)

- Verifies that the components/subsystems interact correctly.
  - Done by software engineer and/or independent test group (integration/system tester).

System Testing (using Black Box test cases)

- Verifies that the system functions correctly as a whole.
  - Done by independent test group (integration/system tester).

DEVELOPERS

Acceptance Testing (using Black Box test cases)

- Validates the software against its requirements.
  - Done by client/user.

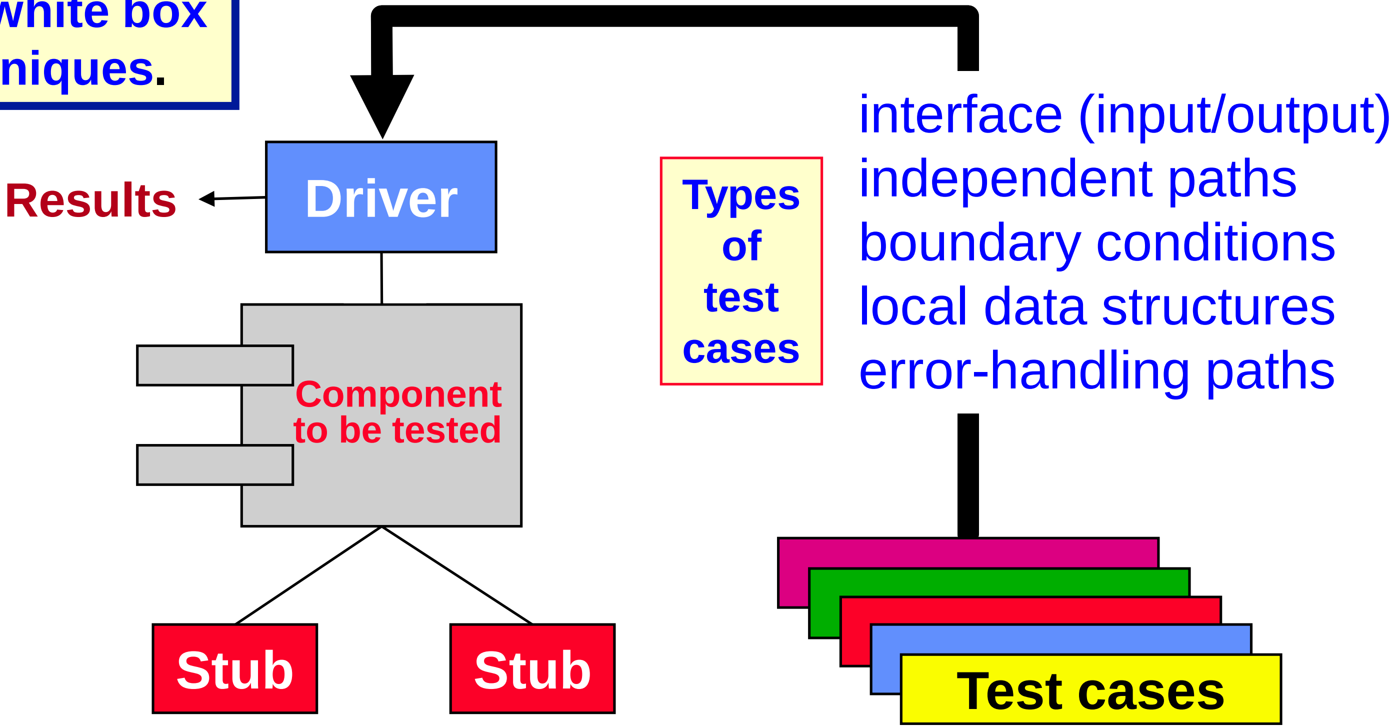
USERS

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ✓ Implement Tests
- ➡ **Perform Tests**
  - ☞ **Unit**
  - – Integration
  - – System
- Evaluate Tests

UNIT TESTING

Main emphasis is on white box techniques.



driver: a component that calls the component to be tested
stub: a component called by the component to be tested

A driver and/or stubs may need to be developed for each unit test.

UNIT TESTING: OBJECT-ORIENTED TESTING

What to unit test?

– A unit test has to be at least a class (i.e., we need to check an object's behaviour), but object state makes testing difficult.

☞ **A class must be tested in every state it can ever enter (i.e., use state-based testing).**

How to deal with inheritance and polymorphism?

– If a subclass overrides methods of an already tested superclass, what needs to be tested — only the overridden methods?

☞ **No! All of a subclass's methods need to be tested again due to dynamic binding and substitutability.**

UNIT TESTING: OBJECT-ORIENTED TESTING

How to deal with encapsulation?

– Encapsulation hides what is inside an object → hard to know its state.

☞ **We need to provide a method, *for testing only*, that reports all of an object's state.**

Example: Testing a stack (push, pop, *peek*)

Suppose popping an empty stack fails somehow.

– Peeking at an empty stack should show nothing.

– Push an element onto the stack, then peek at the stack expecting the same element you pushed.

– Push an element onto the stack, then pop it, expecting the same element you pushed and peek at the stack expecting the stack to be empty.

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ✓ Implement Tests
- ➡ **Perform Tests**
  - ✓ Unit
  - ☞ **Integration**
  - – System
- Evaluate Tests

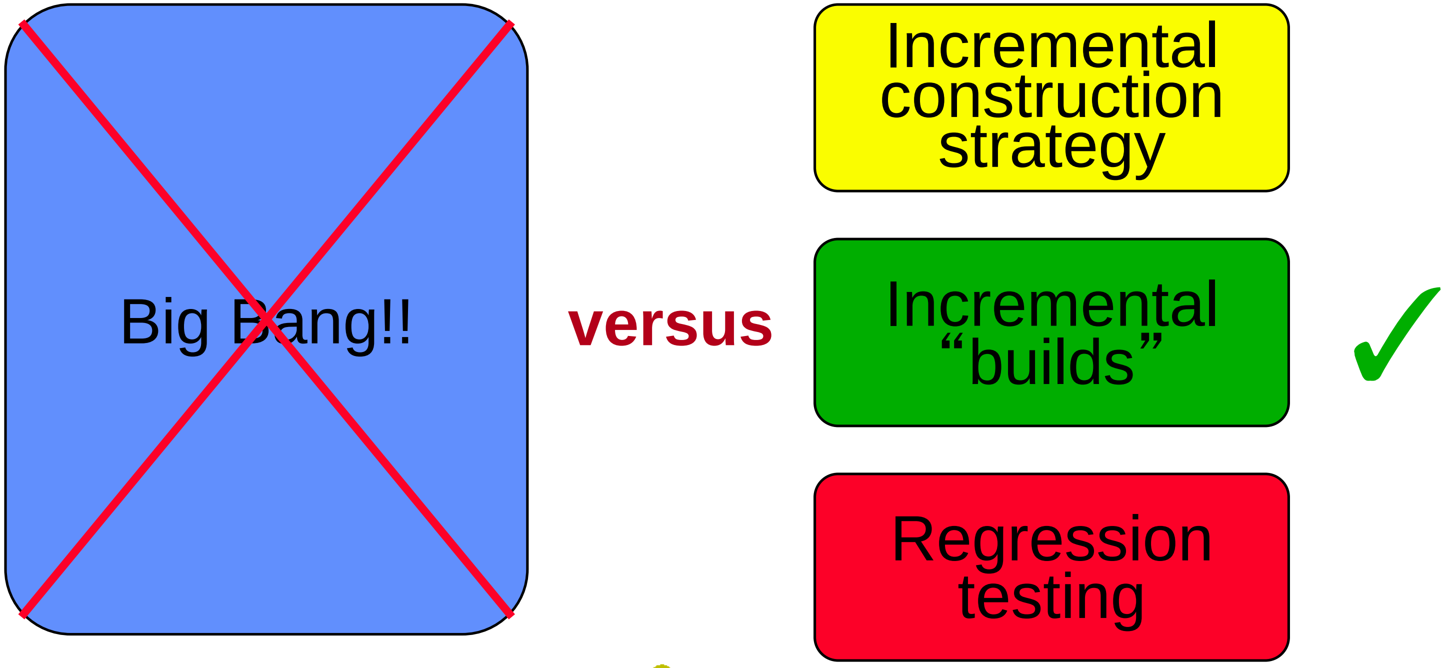
INTEGRATION TESTING

If components all work individually, why more testing?

Interaction errors cannot be uncovered by unit testing!

(e.g., interface misuse, interface misunderstanding, timing errors, etc.)

Integration approaches



~~Big Bang!!~~

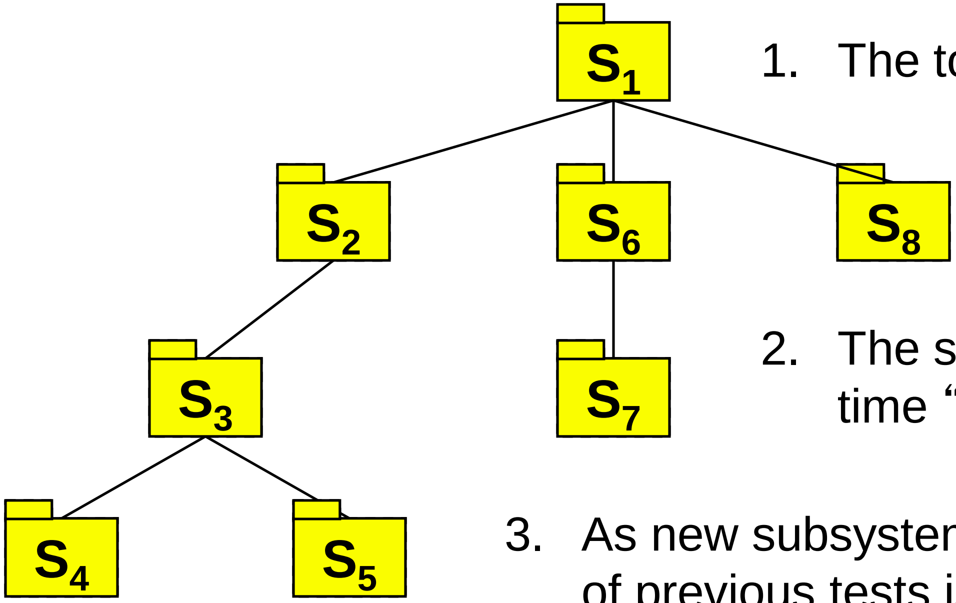
versus

Incremental construction strategy

Incremental "builds" ✓

Regression testing

INTEGRATION TESTING: TOP DOWN



1. The top subsystem is tested with stubs.

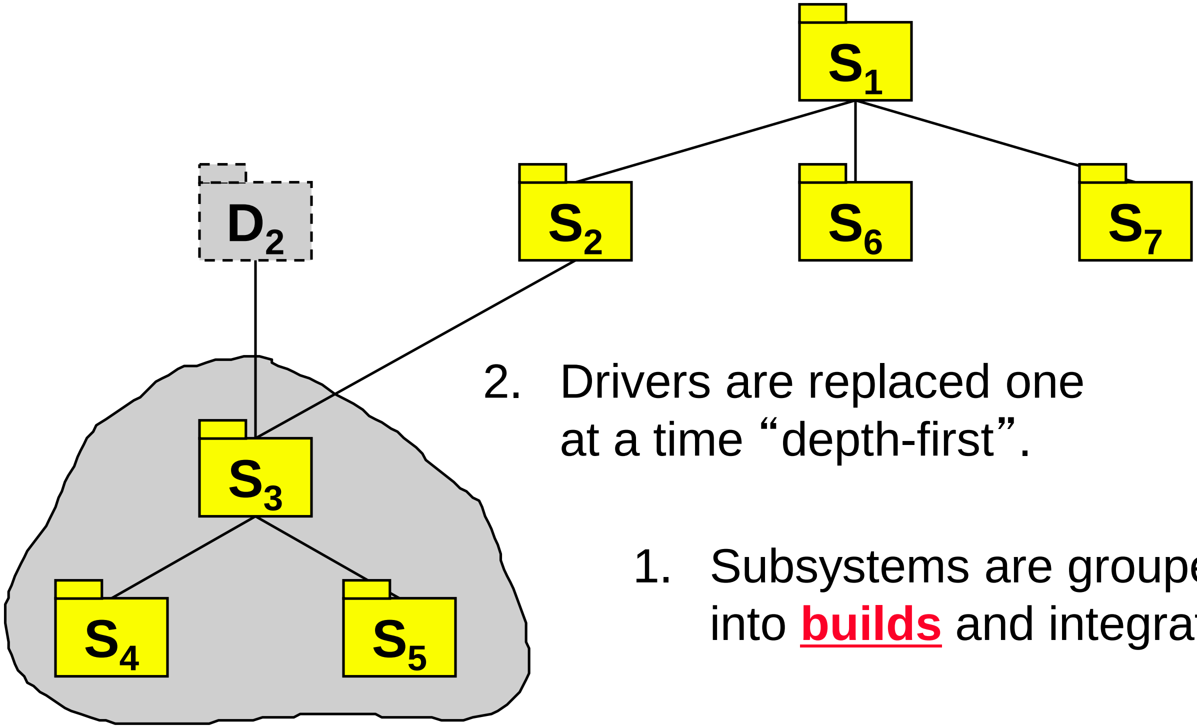
2. The stubs are replaced one at a time “depth-first” or “breadth-first”.

3. As new subsystems are integrated, some subset of previous tests is re-run (**regression testing**).

Pro: Early testing and error detection of user interface components; can demonstrate a complete function of the system early.

Con: Cannot do significant low-level processing until late in the testing; need to write and test stubs.

INTEGRATION TESTING: BOTTOM UP



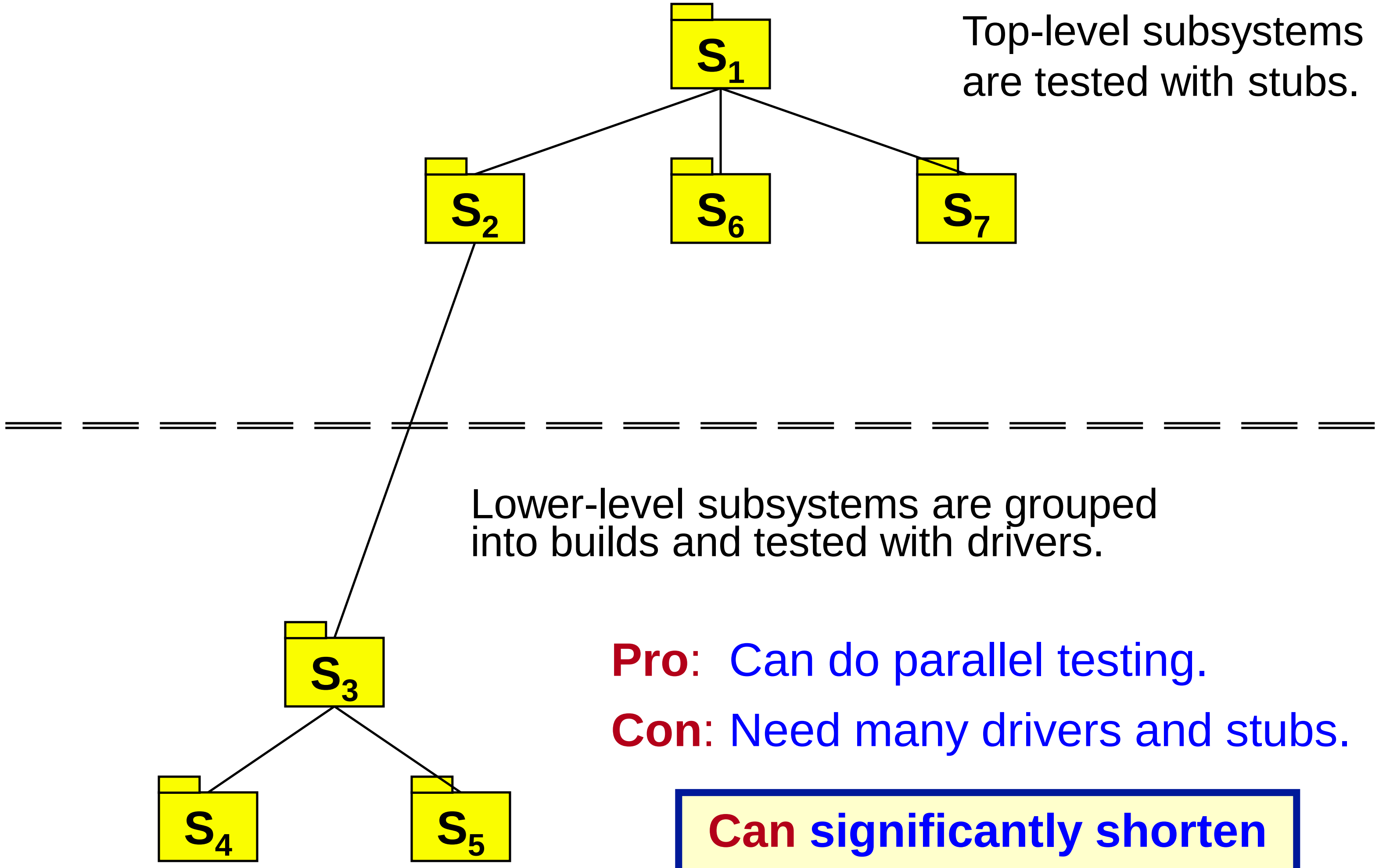
2. Drivers are replaced one at a time "depth-first".

1. Subsystems are grouped into **builds** and integrated.

Pro: Interaction faults are more easily found; easier test case design and no need for stubs.

Con: User interface components are tested last.

INTEGRATION TESTING: SANDWICH



Top-level subsystems are tested with stubs.

Lower-level subsystems are grouped into builds and tested with drivers.

Pro: Can do parallel testing.

Con: Need many drivers and stubs.

Can significantly shorten total testing time.

INTEGRATION TESTING: CRITICAL SUBSYSTEMS

***Critical subsystems* should be tested as early as possible!**

Critical subsystems are those that:

1. have high risk.
2. address several software requirements (i.e., they implement several use cases).
3. have a high level of control.
4. are complex or error prone → high cyclomatic complexity.
5. have specific performance requirements.

Regression testing is <u>required</u> for critical subsystems!

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ✓ Implement Tests
- ➡ **Perform Tests**
  - ✓ Unit
  - ✓ Integration
  - ☞ **System**
- Evaluate Tests

SYSTEM TESTING

System testing **is testing of the entire system to be sure the system functions properly when integrated.**

Some specific types of system tests:

- **Functional** → The developers verify that all user functions work as specified in the system requirements specification.
- **Performance** → The developers verify that the design goals (nonfunctional requirements) are met.
- **Pilot** → A selected group of end users verifies common functionality in the target environment.
- **Acceptance** → The client/user verifies usability and validates functional and nonfunctional requirements against the system requirements specification.
- **Installation** → The client/user verifies usability and validates functional and nonfunctional requirements in real use.

SYSTEM TESTING: PERFORMANCE TESTING

stress Verify that the system can continue functioning when confronted with many simultaneous requests.

☞ How high can we go? Do we fail-soft or collapse?

volume Verify that the system can handle large amounts of data, high complexity algorithms, or high disk fragmentation.

security Verify that access protection mechanisms work.

☞ Make penetration cost more than value of entry.

timing Verify that the system meets timing constraints.

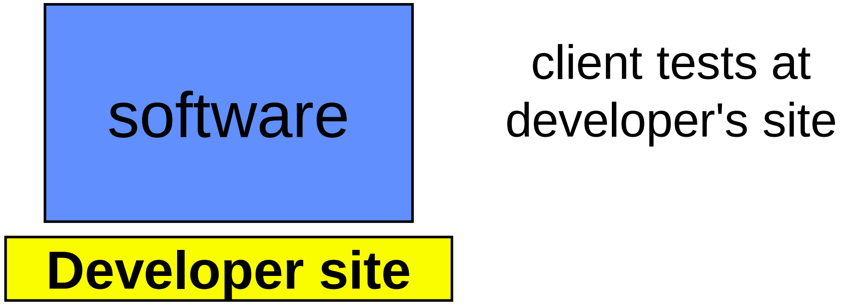
☞ Usually for real-time and embedded systems.

recovery Verify that the system can recover when forced to fail in various ways.

☞ Database recovery is particularly important.

SYSTEM TESTING: PILOT TESTING

Alpha test: A test in a controlled environment so that developers can observe users.

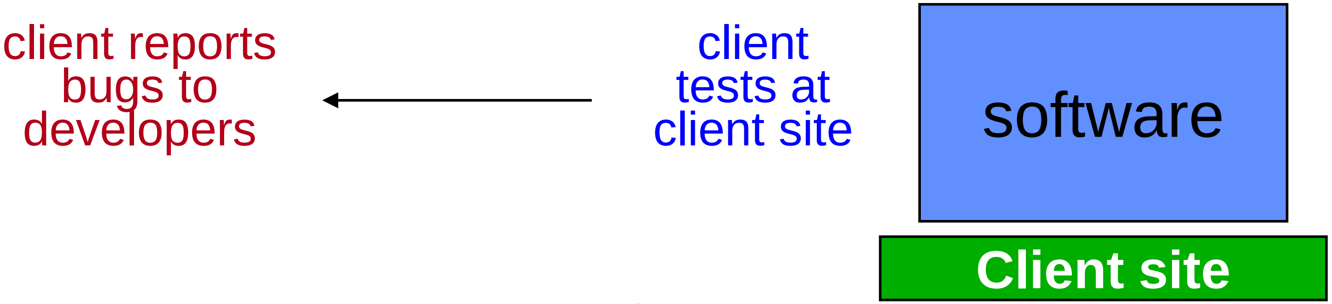


software

Developer site

client tests at developer's site

Beta test: A test in a real environment so that bugs are uncovered from regular usage patterns.



client reports bugs to developers

client tests at client site

software

Client site

SYSTEM TESTING: ACCEPTANCE TESTING

An *acceptance test* demonstrates to the client that a function or constraint of the system is fully operational.

Functional validity – Does the system provide the required functionality?

- ☞ **ASU:** Show that a professor can select a course offering to teach.

Interface validity – Do interfaces perform desired functions (accept desired input/provide desired output) and follow required design standards?

- ☞ **ASU:** Show that all data for course registration can be input.

Information content – Are the stored data correct (i.e., in the required format and obey the required constraints)?

- ☞ **ASU:** Show that all information of a student's course schedule is correct.
- ☞ **ASU:** Show that a student cannot register for more than four courses.

Performance – Does the system meet specified performance criteria?

- ☞ **ASU:** Show the response time to register for a course is less than 1 second.

SYSTEM TESTING:
DERIVING ACCEPTANCE TESTS

- **Restate** written requirements in a concise, precise and testable way by:
  - grouping related requirements.
  - removing any requirements that cannot be tested.

- **Add** any additional requirements gathered from users by:
  - looking at use cases for functional and interface requirements.
  - looking at domain model for information content requirements.
  - looking at nonfunctional requirements for performance requirements.

- **Construct**, for each requirement, an **evaluation scenario** that will demonstrate to the client that the requirement is met.

  (Since most evaluation scenarios depend on the user interface, they can not be completed until the user interface is designed.)

ASU COURSE REGISTRATION REQUIREMENTS

At the beginning of each term, students may request a course catalogue containing a list of course offerings needed for the term. Information about each course, such as instructor, department, and prerequisites are included to help students make informed decisions.

The new system will allow students to select four course offerings for the coming term. In addition, each student will indicate two alternative choices in case a course offering becomes filled or is canceled. No course offering will have more than forty students or fewer than ten students. A course offering with fewer than ten students will be canceled. Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the term.

Professors must be able to access the online system to indicate which courses they will be teaching, and to see which students signed up for their course offerings.

For each term, there is a period of time that students can change their schedule. Students must be able to access the system during this time to add or drop courses.

ASU: REQUIREMENTS PRECISELY STATED & GROUPED

1. Students can request course catalogues containing course offerings for the term.
2. Students decide which courses to take.
3. Course registration:
   3.1. Students can select up to four course offerings.
   3.2. Students can select up to two alternate course offerings.
   3.3. Students can add course offerings to their schedule.
   3.4. Students can drop course offerings from their schedule.
   3.5. A course offering cannot have more than 40 students enrolled.
   3.6. A course offering with less than 10 students enrolled will be canceled.
4. The Registration System sends information to the Billing System.
5. Courses to teach:
   5.1. Professors can choose courses to teach for the coming term.
   5.2. Professors can request class lists for the courses they are teaching.

ASU: REQUIREMENTS RESTATED IN A TESTABLE WAY

1. Students can request course catalogues containing course offerings for the term.

2. Students decide which courses to take.

3. Course registration:
   3.1. Students can select up to four course offerings.
   3.2. Students can select up to two alternate course offerings.
   3.3. Students can add course offerings to their schedule.
   3.4. Students can drop course offerings from their schedule.
   3.5. A course offering cannot have more than 40 students enrolled.
   3.6. A course offering with less than ten students enrolled will be canceled.

ASU: REQUIREMENTS RESTATED IN A TESTABLE WAY

4. The Registration System sends information to the Billing System.

5. Courses to teach:
   - 5.1. Professors can choose courses to teach for the coming term.
   - 5.2. Professors can request class lists for the courses they are teaching.

There will be other requirements that have been gathered from the users.

ASU: ACCEPTANCE TESTS

1. Demonstrate that the system can produce a course catalogue containing course offerings for a term.

2. Course registration:
   - 2.1. Demonstrate that course offerings can be selected up to a maximum of four.
   - 2.2. Demonstrate that alternate course offerings can be selected up to a maximum of two.
   - 2.3. Demonstrate that a course offering can be created and that a course offering can be added to an existing (i.e., non-empty) schedule.
   - 2.4. Demonstrate that students can drop course offerings from their schedule.
   - 2.5. Enroll students in course offerings and show that no more than 40 students can be enrolled in any one course offering.
   - 2.6. Enroll less than 10 students in a course offering and show that it is canceled at the end of the registration period.

ASU: ACCEPTANCE TESTS (cont'd)

3. Demonstrate that the Registration System can send the required information to the Billing System for each student.

4. Courses to teach:
   - 4.1. Demonstrate that professors can choose courses to teach for the coming term.
   - 4.2. Demonstrate that professors can display class lists for the courses they are teaching.

These tests must be made "operational" by devising test cases for use by the client.

TESTING OUTLINE

- ✓ Testing Overview
- ✓ Plan Tests
- ✓ Design Tests
  - ✓ White/Glass Box
  - ✓ Black Box
  - ✓ Regression
- ✓ Implement Tests
- ✓ Perform Tests
  - ✓ Unit
  - ✓ Integration
  - ✓ System
- ➡ **Evaluate Tests**

EVALUATE TESTS

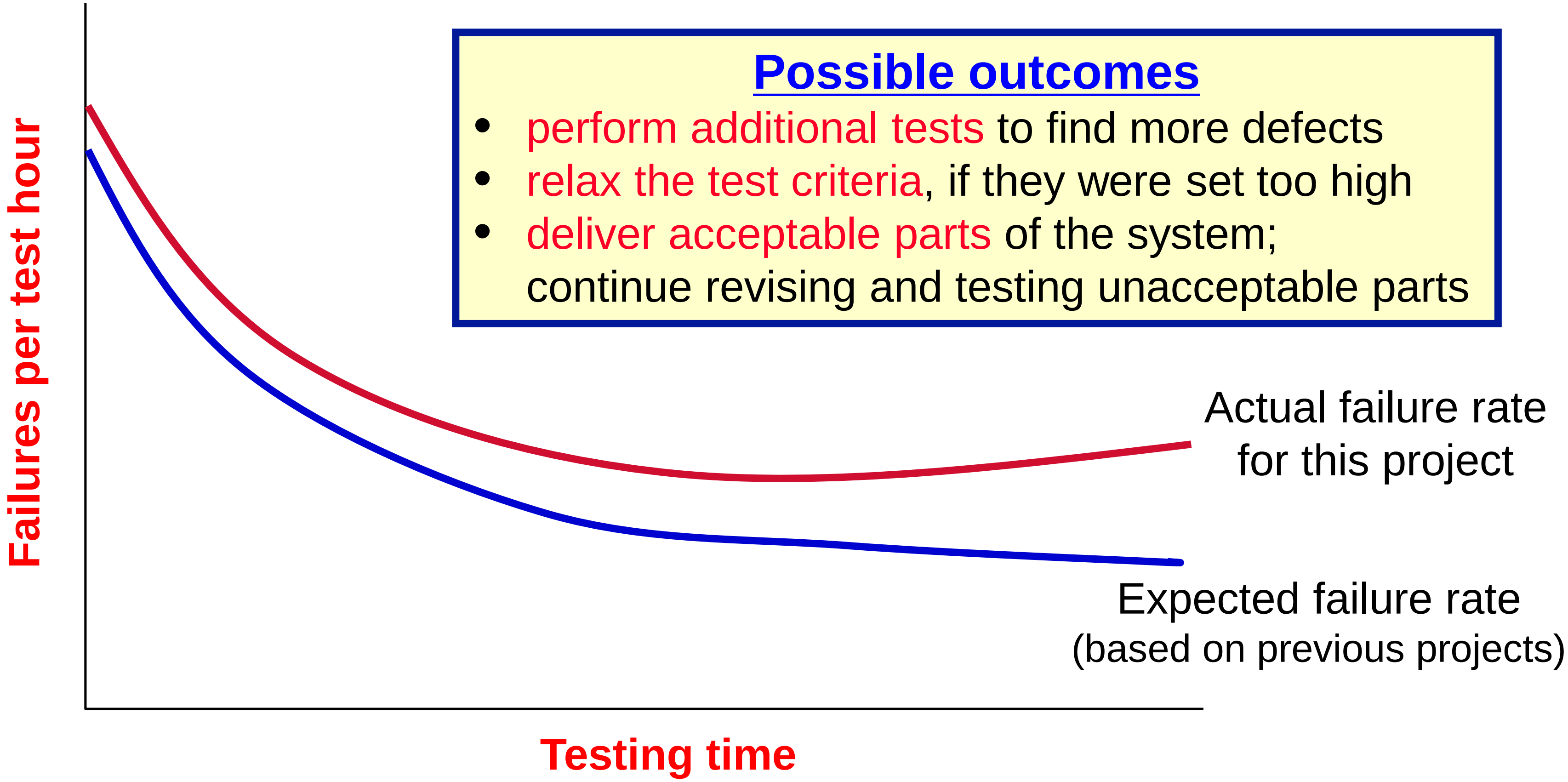
- The test engineer needs to evaluate the results of the testing by:
  - comparing the results with the goals outlined in the test plan.
  - preparing metrics to determine the current quality of the software.

☞ **How do we know when to stop testing?**

- We can consider the system's:

1. **testing completeness/coverage**: the % of test cases that have been run and the % of code that has been tested.
2. **reliability**: based on trends in testing error rate when compared to previous projects.

EVALUATE TESTS: TESTING ERROR RATE



Possible outcomes

- perform additional tests to find more defects
- relax the test criteria, if they were set too high
- deliver acceptable parts of the system;
  continue revising and testing unacceptable parts

☞ **Past testing history can be used to plot expected failure rate. We compare this with actual failure rate for this project.**

TESTING UNDER PRESSURE

All tests are important, but if you need to restrict testing …

- **Testing a system's capabilities (functionality) is more important than testing its components.**
  - It is important to identify things that will prevent users from using the system (e.g., doing their work).

- **Testing old capabilities is more important than testing new capabilities.**
  - It is important that previous functions keep working!

- **Testing typical situations is more important than testing boundary cases.**
  - It is important that the system work under normal usage patterns.

TESTING: SUMMARY

Effective testing requires:

- **Good planning.**
  - Know for what you are trying to test.
- **Use of the right test in the right situation.**
  - Choose the appropriate test type (white box, black box, regression).
- **Early and frequent testing**
  - Catch bugs early, before they have a chance to hide.
- **Systematic and thorough tests!**
  - Have a testing strategy (e.g., unit, integration, system, acceptance).
- **Clear criterion for stopping.**
  - Decide *beforehand* how much testing will be enough.

COMP 3111 SYLLABUS

✓ 1. Introduction

✓ 2. Modeling Software Systems Using UML

✓ 3. Software Development Process

✓ 4. System Requirements Capture

✓ 5. Implementation

✓ 6. Testing

➡ **7. System Analysis and Design**

8. Software Quality Assurance

9. Managing Software Development

TESTING
ACCEPTANCE TESTING EXERCISE