วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

Quality Management

Software Testing Strategies

Software Testing Strategies

- A strategy for software testing provides a road map that describes ...
  - the steps to be conducted as part of testing,
  - when these steps are planned and then undertaken, and
  - how much effort, time, and resources will be required.

- Therefore, any testing strategy must incorporate test planning, test-case design, test execution, and resultant data collection and evaluation.
  - A software testing strategy should be flexible enough to promote a customized testing approach.
  - At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses.

What Is Software Testing Strategies?

- How do you conduct the tests to uncover errors?
- Should you develop a formal plan for your tests?
- Should you test the entire program as a whole or run tests only on a small part of it?

- Should you rerun tests you've already conducted as you add new components to a large system?

- When should you involve the customer?

- These and many other questions are answered when you develop a software testing strategy.

Who Develops Software Testing Strategies

- A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important?

- Testing often accounts for more project effort than any other software engineering action.
- If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected.
- It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps?

- Testing begins "in the small" and progresses "to the large."
  - Early testing focuses on a single component or on a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s).
  - After components are tested, they must be integrated until the complete system is constructed.
  - At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements.
- As errors are **uncovered**, ...
  they must be **diagnosed** and **corrected** using a process that is called debugging.

What is the work product?

- **Test Specification** documents the software team's approach to testing by defining ...
  - a **plan** that describes an overall strategy and

  - a **procedure** that defines specific testing steps and the types of tests that will be conducted.

How do I ensure that I've done it right?

- By reviewing the Test Specification prior to testing, you can assess the completeness of test cases and testing tasks.
- An effective test plan and procedure will lead to ...
  - the orderly construction of the software and
  - the discovery of errors at each stage in the construction process.

A Strategic Approach To Software Testing

- All software testing strategies proposed in the literature have the following generic characteristics:
  - To perform effective testing, you should conduct effective **technical reviews**. It helps eliminate many errors before testing commences.
  - Testing begins at the **component level** and works "outward" toward the **integration** of the entire computer-based system.
  - Different testing techniques are appropriate for different software engineering approaches and at different points in time.
  - Testing is conducted by the developer of the software and (*for large projects*) an independent *test group*.
  - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- **Boehm** states this another way:
  - Verification: "Are we building the product **right**?"
  - Validation: "Are we building the **right** product?"

Many definitions !!!

- V&V includes various testing and many other activities

You can't test in quality.
If it's **not there before** you begin testing, it won't **be there when** you're **finished** testing."

Organizing for Software Testing

- Software analysis and design (along with coding) are **constructive** tasks.

- Testing can be considered to be (psychologically) **destructive**.

- Developers understand the system but, will test "gently" and, is driven by "delivery".
- *Independent testers* must learn about the system, but, will attempt to "break" it and, is driven by "quality"

Organizing for Software Testing

- There are often a number of misconceptions :

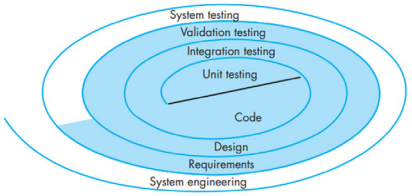
  (1) that the developer of software should do no testing at all,
  (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly,
  (3) that testers get involved with the project only when the testing steps are about to begin.

Organizing for Software Testing

- An ***independent test group*** (ITG) is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project.

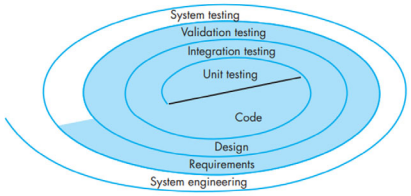
To whom should the ITG report?
(in a large organization)

Software Testing Strategy—The Big Picture



- The **software process** may be viewed as the spiral.
- Initially, **system engineering** defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.
- Moving inward along the spiral, you come to design and finally to coding.
- To develop computer software, you **spiral inward** along streamlines that decrease the level of abstraction on each turn.

Software Testing Strategy—The Big Picture

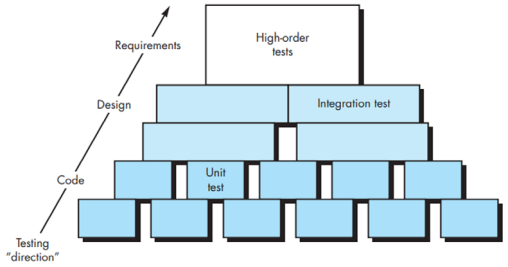


- A strategy for **software testing** may **also** be viewed in the context of the spiral.
- **Unit testing** begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source *code*.
- Testing progresses by moving outward along the spiral to **integration testing**, where the focus is on *design* and the construction of the SW architecture.
- Taking another turn outward on the spiral, you encounter **validation testing**, where *requirements* established as part of requirements modeling are validated against the software that has been constructed.
- Finally, you arrive at **system testing**, where the software and other system elements are tested as *a whole*. (**spiral out** & broaden the scope of testing)

Who Develops Software Testing Strategies

Software Testing Steps

- Testing within the context of software engineering is a series of **four steps** that are implemented sequentially.
- Unit testing should ensure complete coverage and maximum error detection in **each component**.

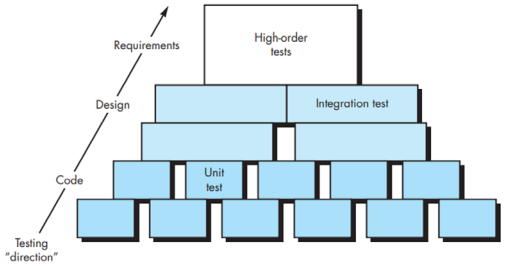


- Components are assembled or to form the complete software **package**. Integration testing addresses the issues associated with the dual problems of verification and program construction.

Who Develops Software Testing Strategies

Software Testing Steps

- After the software has been integrated (constructed), a set of high-order tests is conducted.
- Validation testing provides final assurance that software meets all functional, behavioral, and performance *requirements*.



- Software, once validated, must be combined with other system elements (e.g., hardware, people, databases).
  - *System testing* (the last high-order testing step) verifies that all elements mesh properly and that ***overall*** system function/performance is achieved. (falls outside the boundary of software engineering and into the broader context of computer system engineering.)

Criteria for Completion of Testing

- "When are we done testing—how do we know that we've **tested enough?**"
  - There is no definitive answer to this question.
  - Someone said that "You're never done testing; the burden simply shifts from you (the software engineer) to the end user."
  - Another replied that "You're done testing when you run out of time or you run out of money."
- Statistical technique:
  - Collect metrics during software testing and making use of existing statistical models to develop meaningful guidelines for answering the question: "When are we done testing?"

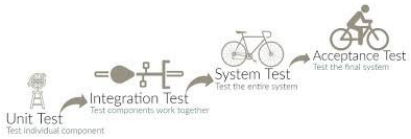
Strategic Issues

- Tom Gilb argues that a **software testing strategy** will succeed only when software testers:

(1) **specify** product requirements in a **quantifiable manner** long before testing commences,
(2) **state** testing objectives **explicitly**,
(3) **understand** the users of the software and **develop** a profile for each user category,
(4) **develop** a testing plan that emphasizes "rapid cycle testing,"
(5) **build** "robust" software that is designed to test itself (antibugging),
(6) use effective technical reviews as a filter prior to testing,
(7) conduct technical reviews to assess the test strategy and test cases themselves, and
(8) develop a continuous improvement approach for the testing process

Test Strategies For Conventional Software

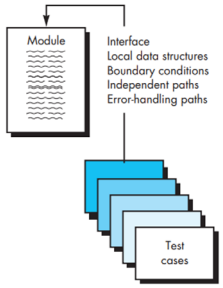
- Traditional software architectures are not object oriented and do not encompass WebApps or MobileApps.
- Conduct **tests** on a **daily basis**, whenever any part of the system is constructed.
- A common testing strategy takes an **incremental** view of testing, ...
  - beginning with the testing of individual program units,
  - moving to tests designed to facilitate the integration of the units (sometimes on a daily basis), and
  - culminating with tests that exercise the constructed system.



Unit Testing

- Unit testing focuses verification effort on the smallest unit of software design—the software **component** or **module**.
  - Important control **paths** are tested to uncover errors within the boundary of the module.
  - The unit test focuses on the *internal* **processing logic** and **data structures** within the boundaries of a component.
  - This type of testing can be conducted in parallel for multiple components.

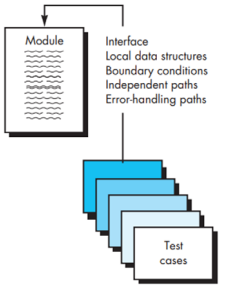
Unit Testing



- The module **interface** is tested to ensure that **information** properly flows **into** and **out** of the program unit under test.
- **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent **paths** through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.

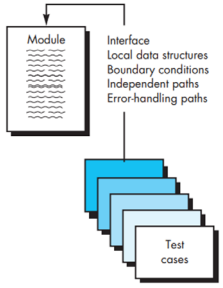
- And finally, all **error-handling paths** are tested.

Unit Testing



- **Data flow** across a component interface is **tested before any other** testing is initiated.
  - Otherwise, all other tests are moot.
- Test cases should be designed to uncover errors due to ...
  - erroneous computations,
  - incorrect comparisons,
  - improper control flow, etc..

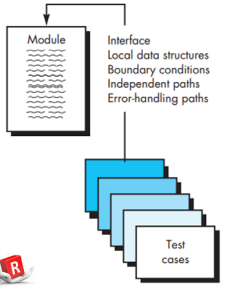
Unit Testing



- Boundary testing is one of the most important unit testing tasks.
  - Software often fails at its **boundaries**.

  - That is, errors often occur ...
    - when the *n*th element of an *n*-dimensional array is processed,
    - when the *i*th repetition of a loop with *i* passes is invoked,
    - when the maximum or minimum allowable value is encountered.
  - Test cases that exercise *data structure*, *control flow*, and *data values* **just below**, **at**, and **just above** *maxima* and *minima* are very likely to uncover errors.

Unit Testing



- A good design anticipates error conditions and establishes **error-handling paths** to *reroute* or *cleanly terminate* processing when an error does occur.

  - Yourdon calls this approach **antibugging**.
  - If error-handling paths are implemented, they must **be tested**.
- The **potential errors** that should be tested when error handling is evaluate are ...
  (1) error description is unintelligible,

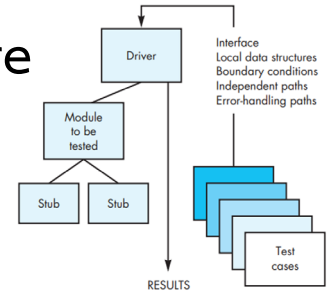
  (2) error noted does not correspond to error encountered,

  (3) error condition causes system intervention *prior to error handling*,

  (4) exception-condition processing is incorrect, or

  (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit Test Procedure



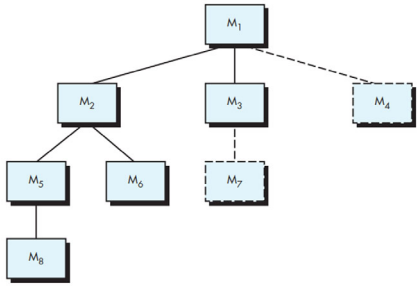
- Because a component is not a stand-alone program, **driver** and/or **stub** software must often be developed for each unit test (the right figure).
- In most applications a driver is nothing more than a "main program" that accepts test-case data, passes such data to the component (to be tested), and prints relevant results.
- Stubs serve to replace modules that are *subordinate* (invoked by) the component to be tested.
  - A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs must be coded → "overhead"

Integration Testing

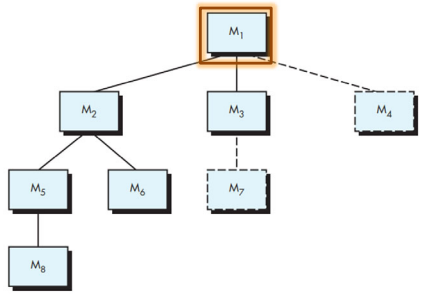
- Integration testing uncovers **errors** associated with **interfacing**.
- **Incremental integration**.
  - The program is constructed and tested in small increments, where errors are easier to isolate and correct;
  - Interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-down integration testing



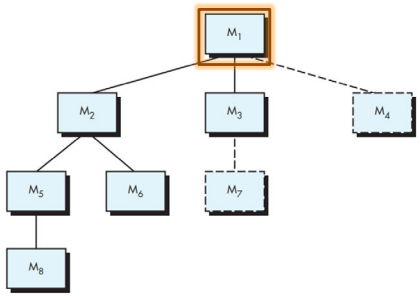
- Modules are integrated by moving downward through the control hierarchy, beginning with the **main** control module (main program).
- Modules **subordinate** (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Top-down integration testing



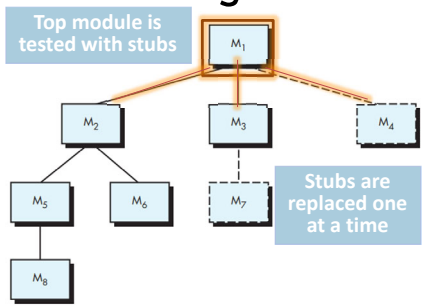
- Depth-first integration integrates all components on a **major control path** of the program structure.
  - Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.
- For example, selecting the left-hand path, components **M1**, **M2** , **M5** would be integrated first.
- Next, **M8** or (if necessary for proper functioning of M2) **M6** would be integrated.
- Then, the central and right-hand control paths are built.

Top-down integration testing



- Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure **horizontally**.
- From the figure, components **M2**, **M3**, and **M4** would be integrated first.
- The next control level, **M5**, **M6**, and **so on**, follows.

Top-down integration testing



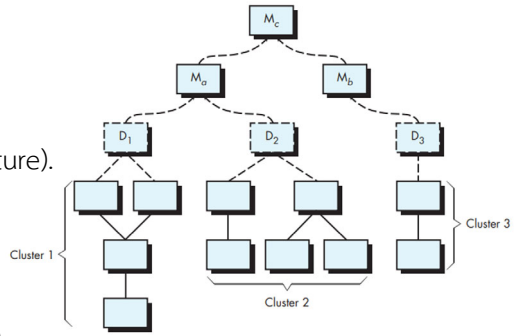
- The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and ...

   stubs are **substituted for all** components directly **subordinate** to the main module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are **replaced *one at a time* with actual components**.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the **real component**.
5. **Regression testing** may be conducted to ensure that new errors have not been introduced.

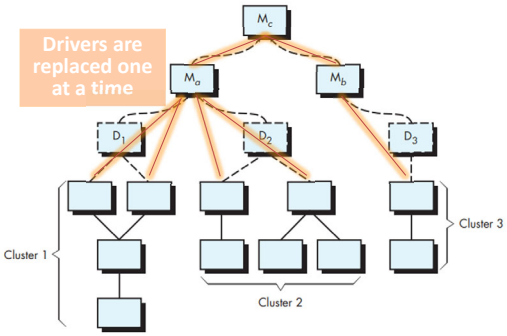
The **process continues** from step 2 **until the entire program** structure is **built**.

Bottom-up integration testing



- Begins construction and testing with atomic modules (i.e., components at **the lowest levels** in the program structure).
  - Because components are integrated from the bottom up, the functionality provided by components **subordinate** to a given level is always **available** and ***the need for stubs is eliminated***.

Bottom-up integration testing



- A bottom-up integration strategy may be implemented with the following **steps**:

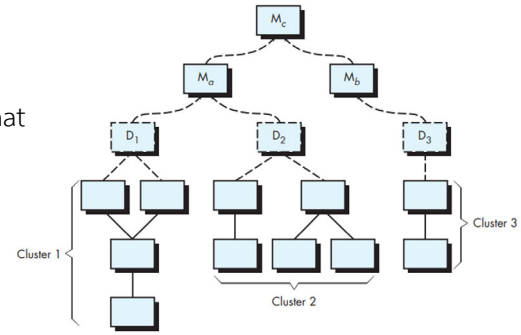
1. Low-level components are **combined** into **clusters** (sometimes called builds) that perform a specific SW subfunction.
2. A driver (a control program for testing) is **written** to coordinate test-case input and output.
3. The cluster is **tested**.
4. **Drivers** are **removed** and clusters are combined moving upward in the program structure.

As integration **moves upward**, the **need for separate test drivers lessens.**

Integration Testing

Sandwich testing

- In general, a combined approach (sometimes called **sandwich testing**) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.
  - In the previous example, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.



Integration Testing

Regression Testing

- Regression testing is the **re-execution** of some subset of **tests** that have **already been conducted** to *ensure* that *changes* have *not propagated* unintended *side effects*
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is **changed**.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce **unintended behavior** or **additional errors**.
- Regression testing may be conducted **manually**, by re-executing a subset of all test cases or using **automated** capture/playback **tools**

Integration Testing

Smoke Testing

- It is designed **for** creating "**daily builds**" especially for time-critical SW projects, allowing the software team to ***assess*** the project ***on a frequent basis***.
- Smoke testing steps:
  - **Software components** that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - **A series of tests** is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The **build** is **integrated with other builds** and the *entire product* (in its current form) *is smoke tested daily*.
    - The integration approach may be top down or bottom up.

Build Verification Testing (BVT) or Confidence Testing

Find major errors

OO Software

Test Strategies For Object-Oriented Software