วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

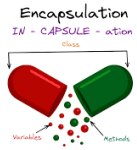
Quality Management

Software Testing Strategies

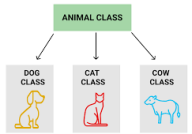
Test Strategies For Object-Oriented Software

- **Begins** by evaluating the correctness and consistency of the ***analysis and design models***

Unit Testing in the OO Context



- Testing strategy changes
  - An **encapsulated class** is usually the focus of unit testing.
  - Operations (**methods**) within the class are *the smallest testable units*.
  - **Some *operations*** may exist as ***part of a number of different classes***, the tactics applied to unit testing must change.
  - You can **no longer** test a single operation **in isolation** but rather as part of a class.



Test Strategies For Object-Oriented Software

Unit Testing in the OO Context

- To illustrate, consider a class hierarchy in which an operation X is defined for the superclass and is inherited by a number of subclasses.
- **Each subclass** uses **operation X**, but it is applied within the context of the **private attributes** and **operations** that have been defined for the subclass.
- Because the context in which operation X is used varies in subtle ways, it is necessary to test operation X in the context of each of the subclasses.
  - This means that testing operation X in a stand-alone fashion (the conventional unit-testing approach) is usually **ineffective** in the object-oriented context.

Test Strategies For Object-Oriented Software

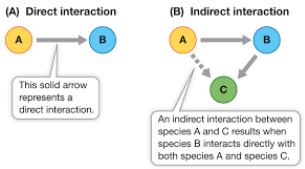
Unit Testing in the OO Context

- **Class testing** for OO software == **Unit testing** for conventional software.
  - Unit testing of conventional software tends to focus on the ***algorithmic detail*** of a ***module*** and the ***data*** that flow across the module interface.
  - Class testing for OO software is driven by the **operations** encapsulated by the class and the **state behavior** of the class.

Test Strategies For Object-Oriented Software

Integration Testing in the OO Context

- Because OO software does **not** have an **obvious hierarchical** control structure, ...
  - traditional top-down and bottom-up integration strategies have **little meaning**.
- Integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible ...
  - because of the "**direct** and **indirect** interactions of the components that make up the class"



Test Strategies For Object-Oriented Software

Integration Testing in the OO Context

- There are **two different strategies** for integration testing of OO systems.

❶ The first, **thread-based testing**, integrates the set of classes required to respond to one input or event for the system.

Threads are sets of classes that respond to an input or event

  - Each thread is integrated and tested individually.
  - Regression testing is applied to ensure that no side effects occur.

❷ The second integration approach, **use-based testing**, ...

Use-based tests focus on **classes** that do **not collaborate heavily with other classes.**

  - **begins** the construction of the system by testing those classes (called **independent classes**) that use very few (if any) server classes.
  - After the independent classes are tested, the **next** layer of classes, called **dependent classes**, that use the independent classes are tested.
  - This sequence of testing layers of dependent classes continues until the entire system is constructed.

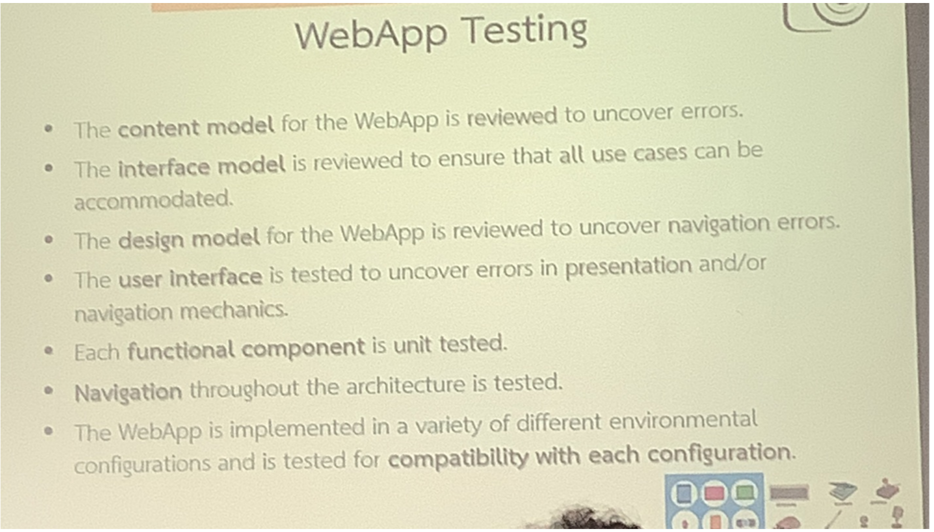
Test Strategies For Object-Oriented Software

Integration Testing in the OO Context

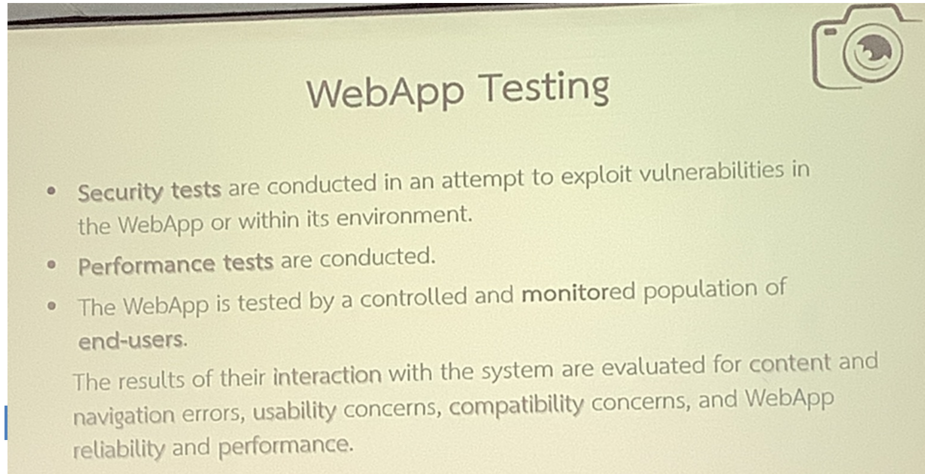
- The ***use of*** **drivers** and **stubs** ***also*** ***changes*** when integration testing of OO systems is conducted.
  - **Drivers** can be used to test **operations at the lowest level** and for the testing of **whole groups of classes**.
  - A driver can also be used to **replace** the **UI** so that tests of system functionality can be conducted *prior to implementation* of the interface.
  - **Stubs** can be used in situations in which collaboration between classes is required but **one or more of the collaborating classes** has **not yet** been **fully implemented**.
- **Cluster testing** is one step in the integration testing of OO software.
  - A **cluster of collaborating classes** is exercised by designing test cases that attempt to uncover **errors** in the **collaborations**.

WebApp Testing

มีการทดสอบในที่เฉพาะเจาะจง
สำหรับทำเว็บแอพ



WebApp Testing



MobileApp Testing

- *User-experience testing*. Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices.

ความเข้ากันได้กับ device

- *Device compatibility testing*. Testers verify that the MobileApp works correctly on all required hardware and software combinations.
- *Performance testing*. Testers check nonfunctional requirements unique to mobile devices (e.g., *download times*, *processor speed*, *storage capacity*, *power availability*).

- *Connectivity testing*. Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access.

เน็ตเร็ว, ช้าเกิน

MobileApp Testing

- *Security testing*. Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users.

- *Testing-in-the-wild*. The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe.
- *Certification testing*. Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it.

No more details on methods for MobileApp testing will be **provided** in this course.

Validation Testing

- Validation tries to uncover **errors at the requirements level**—on things that will be *immediately apparent to the end user*.
- A series of tests that demonstrate **conformity with requirements**.
  - **Configuration Review**.
    It ensures that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. (*sometimes called* an ***audit***)
  - **Alpha and Beta Testing**.
    Focus is on customer usage.
    Impossible for a software developer to *foresee how the customer* will really *use a program*. (Instructions misinterpreted, use of strange data, output seemed clear to the tester but unintelligible to a user).

Validation Testing

- The **alpha test** is conducted **at the developer's site** by a representative group of end users.
  - The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
  - Alpha tests are conducted in a controlled environment.
- The **beta test** is conducted **at** one or more **end-user sites**.
  - Unlike alpha testing, the developer generally is not present.
  - A "live" application of the SW in an environment not controlled.
  - The customer records all problems (real or imagined) that are encountered and reports these to the developer at regular intervals.
  - Problems reported, modifications made.

Validation Testing

- A variation on beta testing, called **customer acceptance testing**, is sometimes performed when custom *software is delivered to a customer under contract*.
- The customer performs a series of specific tests in an attempt to **uncover errors before accepting the software from the developer**.
- In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

System Testing

System Testing

- Focus is on **system integration**.
- Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted.
- These tests fall outside the scope of the software process and are **not conducted solely by software engineers**.
- However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

System Testing

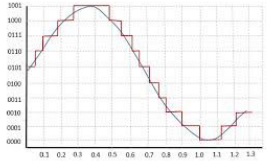
- **Recovery Testing.**
  Forces the software to fail in a variety of ways and verifies that **recovery** is properly performed and resume processing with little or no downtime.
  - In **some cases**, a system must be **fault tolerant** (must not cause overall system function to cease).
  - In **other cases**, a system failure must be **corrected** within a specified period of time or severe economic damage will occur.
- If **recovery** is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are **evaluated for correctness**.
- If recovery **requires human intervention**, the mean-time-to-repair (**MTTR**) is **evaluated** to determine **whether** it is **within acceptable limits**.

System Testing

- **Security Testing.**
  Verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration.
  - Given enough time and resources, good security testing will ultimately penetrate a system.
  - The role of the system designer is to make penetration cost more than the value of the information that will be obtained.
- **Stress Testing.**
  Executes a system in a manner that **demands resources** in abnormal *quantity*, *frequency*, or *volume*.
  Essentially, the tester ***attempts to break the program***.
  - For example, special tests may be designed that generate 10 interrupts per second, when one or two is the average rate.

System Testing

- A **variation** of **stress testing** is a technique called ***sensitivity testing***.
- In some situations (the most common occur in mathematical algorithms), a very small range of data contained **within the bounds of valid data** for a program may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover **data combinations** within valid input classes that may **cause** instability or improper processing.



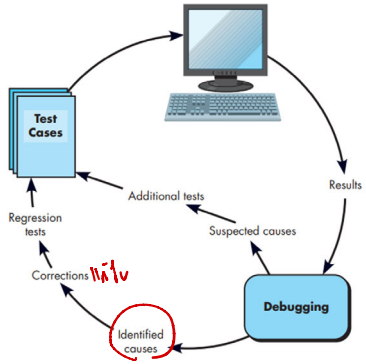
System Testing

- **Performance Testing.**
  Test the run-time performance of software within the context of an integrated system, especially for real-time and embedded systems.
  - Performance testing occurs throughout all steps in the testing process (even at the unit level).
  - Performance tests are often coupled with *stress testing*.
- **Deployment Testing.**
  Sometimes called ***configuration testing***, exercises the software in each environment in which it is to operate.
  - **In addition**, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Debugging

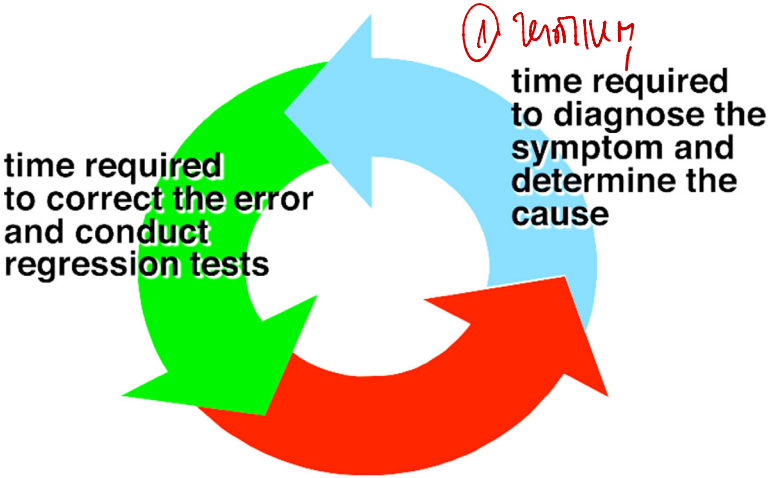
- Debugging occurs as a **consequence** of **successful testing**.
- When a test case uncovers an error, debugging is the process that results in the removal of the error.
- As a software engineer, you are often confronted with a "***symptomatic***" ***indication*** of a ***software problem*** as you evaluate the results of a test.
  - That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another.
- The poorly understood mental process that connects a symptom to a cause is **debugging**.

The Debugging Process



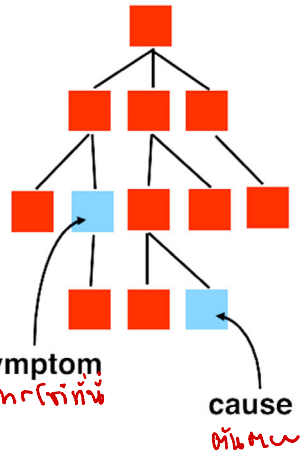
- The debugging process begins with the **execution** of a **test case**.
- **Results** are **assessed** and a lack of correspondence between expected and actual performance is encountered.
- The **debugging process** attempts to **match** ***symptom*** with ***cause***, thereby leading to error correction. Possible **two** outcomes are:
  - (1) the cause will be found and corrected **OR**
  - (2) the cause will not be found.
- In the latter case, the person performing debugging may **suspect a cause**, **design a test case** to help validate that suspicion, and work toward error correction in an iterative fashion.

Debugging Effort



Why is debugging so difficult

- ❑ symptom and cause may be geographically separated
- ❑ symptom may disappear when another problem is fixed — temporarily
- ❑ cause may be due to a combination of non-errors — Round-off inaccuracies
- ❑ cause may be due to a system or compiler error
- ❑ cause may be due to assumptions that everyone believes
- ❑ symptom may be intermittent



Debugging Strategies

- Some people are good at debugging and others aren't.
- Although it may be **difficult** to "**learn**" debugging, a number of approaches to the problem can be proposed.
- **Three debugging strategies** have been proposed:
  brute force,
  backtracking, and
  cause elimination (*induction* or *deduction*).
  - Each of these strategies can be conducted **manually** *OR* using **debugging tools**.

Debugging Tactics

- The **brute force** category of debugging is probably **the most common** and **least efficient** method for isolating the cause of a software error.
  - You apply brute force debugging methods when all else fails.
  - Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements.
  - You hope that somewhere in the morass of information that is produced you'll find a clue that can lead us to the cause of an error.
  - Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time.
  - Thought must be expended first!

Debugging Tactics

- Backtracking is a fairly **common** debugging approach that can be used successfully **in small programs**.
  - Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found.
  - Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

- The third approach to debugging—cause elimination—is manifested by *induction* or *deduction* and introduces the concept of **binary partitioning**.
  - Data related to the error occurrence are organized to isolate potential causes.
    - A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis.
    - **Alternatively**, a list of all possible causes is developed, and tests are conducted to eliminate each.
  - If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

A final maxim for debugging might be: "When all else fails, **get help**!"

Correcting the Error

The **correction** of a bug can introduce other errors and therefore do more harm than good. Three simple questions that you should ask *before* making the "*correction*" :

1. Is the **cause** of the bug **reproduced in another part** of the program?
   - In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
2. **What** "**next bug**" might be introduced **by** the **fix** I'm about to make?
   - **Before** the **correction** is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
3. **What** could we have done **to prevent** this bug in the first place?
   - This question is the first step toward establishing a statistical software QA approach.
   - If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

Final Thoughts

- **Think** -- before you act to correct

- Use **tools** to gain additional insight
- If you're at an impasse, **get help** from someone else


It's OK to get help

- Once you correct the bug, use **regression testing** to uncover any side effects