Slide 22: Debugging Integration and System Testing

Md. Mehedi Hasan Rafy

Debugging

Process of locating and fixing the exact program statements causing errors is called debugging.

Approaches

- ***Brute Force*** – Most common method of debugging. Insert print statements or use source code debuggers because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly. It is simple but inefficient.
- ***Backtracking*** – Trace the source code backwards from where error appears. As the number of potential backward paths increases, backtracking can get complex.
- ***Cause Elimination*** – Test the code with a list of possible causes, and eliminate errors. It is often used with fault tree analysis.
- ***Program Slicing*** – This is a similar technique like backtracking where search space is reduced to only the code lines that influence a variable's value.

Guidelines:

- Requires deep understanding of program design because partial understanding of the system design and implementation can lead to faulty debugging.
- Avoid fixing symptoms instead of real errors.
- Sometimes redesign is necessary.
- Error fixes can create new bugs → always do regression testing afterward.

Integration Testing

- Done after all modules are unit tested. It ensures that the unit or module as a whole working satisfactory.
- ***Goal:*** Detect errors at the module interfaces. No errors in parameter passing when modules interact.

- Uses an integration plan → defines order and steps for combining modules, guided by the module dependency graph.
- After each integration step, the partial system is tested.
- Module dependency graph or structure chart which specifies the order in different modules call each other guides the integration plan.

Integration Plan Development approaches

1. ***Big-Bang*** – integrate everything at once (simple, but hard to debug).
2. ***Top-Down*** – start from root module, add lower modules gradually (uses stubs).
3. ***Bottom-Up*** – start with low-level modules, build upward (uses drivers).
4. ***Mixed (Sandwiched)*** – combines top-down and bottom-up; most commonly used.

Big-Bang Integration Testing

- All modules are combined and tested in a ***single step***.
- Suitable only for very small systems.
- ***Main drawback:*** hard to localize the source of errors because it could be in any module.
- Hence, debugging becomes costly and complex.
- ***Rarely used*** for large programs.

Top-Down Integration Testing

- Starts with the ***root module*** and a few immediate sub-modules.
- Gradually adds lower-level modules and tests step by step.
- Uses ***stubs*** to mimic the effect of missing lower-level routines. This is called routine under test.
- ***Advantage:*** early testing of high-level design.
- ***Disadvantage:*** hard to test top-level routines properly without real lower-level modules (especially I/O operations).

Bottom-Up Integration Testing

- Starts with ***low-level modules***; subsystems are built and tested separately, then combined into higher-level subsystems.
- ***Advantages:***

    - Disjoint subsystems can be tested in parallel.
    - Low-level modules (I/O, critical functions) are thoroughly tested, improving reliability.
- ***Disadvantage:***
    - Becomes complex if there are too many small subsystems at the same level (can resemble Big-Bang).

Mixed Integration Testing

- It is also called sandwiched integration testing. Also most commonly used model in integration testing.
- Combines **top-down** and **bottom-up** strategies.
- In top-down approach, testing can start only after the top-level modules have been coded and unit tested. On the other hand, bottom-up testing can start only after the bottom-level modules are ready. But in mixed approach, testing can begin as soon as any module is ready (no need to wait for only top-level or bottom-level).
- Requires both ***stubs*** (for missing lower modules) and ***drivers*** (for missing higher modules).
- ***Advantage:*** flexible and widely used in practice.

System Testing

- Done after all modules are integrated and tested.
- Test cases are based on the ***SRS (requirements)***, not on implementation details.
- **Three types** of system testing:
    - ***α Testing*** → by the internal test team of the developer organization.
    - ***β Testing*** → by a friendly group of selected real users/customers.
    - ***Acceptance Testing*** → by the customer, to decide whether to accept the delivery of the system.

Smoke Testing

- A pre-check before system testing to ensure if at least the main functionalities of the system are working properly. If they fail, full system testing isn’t worthwhile.
- Uses only a few basic test cases because if integrated program can not pass even the basic tests, it is not ready for a vigorous testing.

- Example: In a library system, check book creation/deletion, member record handling, and book loan/return.

Performance Testing

- A key form of **system testing**, focused on **non-functional requirements** from the SRS.
- Treated as **black-box tests** (don't depend on internal code, just behavior).
- **Types:**
  1. ***Stress Testing (Endurance Testing):***
     - Push system beyond limits (input data volume, input data rate, utilization of memory, processing time) for short period of time.
     - This test is designed impose a range of abnormal and illegal input conditions so as to stress the capabilities of the software.
     - Useful for peak demand hours.
  2. ***Volume Testing:***
     - Test data structures (buffers, arrays, queues, stacks, etc) with very large inputs to check if they are designed to successfully handle extraordinary situations.
     - Example: compiler symbol table overflow check.
  3. ***Configuration Testing:***
     - Verify behavior across different hardware/software setups.
     - Ensures correct performance in minimal vs extended configurations.
     - The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.
  4. ***Compatibility Testing:***
     - Ensure if interfaces has proper interaction with external systems/databases.
     - Check speed and accuracy of data exchange.
  5. ***Regression Testing:***
     - After bug fixes or enhancements, ensure no old features such as functionalities, performance are are broken.

6. ***Recovery Testing:***
    - Test system's response to failures (power loss, data loss, device failure).
    - System is subjected to the loss of mentioned resources as discussed in the SRS document and it should recover gracefully.
7. ***Maintenance Testing:***
    - Verify diagnostic and maintenance tools/programs work properly.
8. **Documentation Testing:**
    - Ensure user manuals, technical manuals, and maintenance manuals are complete, correct, and audience-appropriate.
9. **Usability Testing:**
    - Check user interface for clarity, friendliness, and compliance with user requirements.
    - During usability testing, the display screens, messages, report formats, and other aspects related to user interface requirements are tested.
10. **Security Testing:**
    - Ensure system is safe against intrusions/attacks like password cracking, penetration testing, port attacks to safeguard confidential data.