DETAILED DESIGN, IMPLEMENTATION AND TESTING

Instructor: Dr. Hany H. Ammar
Dept. of Computer Science and Electrical Engineering, WVU
OUTLINE

- Detailed design, Implementation, and testing phase in the software development standards
- Software Testing
  - Black Box Testing
  - White Box Testing
Detailed design, Implementation, and Testing

- In the European Space Agency standard, the major activities in this phase include:
  - decomposition and the specification of the low-level software modules.
    
    the low-level components specified at the architectural design are decomposed into software modules.

  - This is followed by software production and documentation: implementing the modules specifications into code, integrating the software components, and the testing activities.
Detailed design, Implementation, and Testing

Decomposition of low-level components
- Using stepwise refinement, the low-level components specified at the architectural design are decomposed into software modules, and Library modules (language dependent Application Programming Interfaces APIs, e.g., Microsoft Foundation Classes, Java API packages for Graphical User Interfaces or GUIs, network programming, etc.)
Decomposition of low-level components

*Example: Process Pilot Request subsystem*

- Specify the detailed design of subordinate modules of Process_Pilot_Request
- Specify how buffers are structured (the data structures of the buffers used for inputs and outputs)
Detailed design, Implementation, and Testing

Software production

Software production consists of

- Implementing the modules specifications into code (structured Programming or object-oriented programming),
- Integrating the software components and the testing activities in testing procedures of unit testing, integration testing, and system testing
Detailed design, Implementation, and Testing

The activities required in the MIL-STD-498 standard for the Software Implementation and Unit Testing (SWIUT) are described as follows:

- Develop and document software corresponding to each software unit (CSU) of each software component (CSC) in the CSCI design
- Establish test cases (in terms of inputs, expected results, and evaluation criteria), test procedures, and test data for testing the software corresponding to each software unit
Detailed design, Implementation, and Testing

- The test cases shall cover all aspects of the unit's detailed design. The developer shall record this information in the appropriate software development files (SDFs).
- Test the software corresponding to each software unit. The testing shall be in accordance with the unit test cases and procedures.
Detailed design, Implementation, and Testing

- Make all necessary revisions to the software, perform all necessary retesting, and update the software development files (SDFs) and other software products as needed, based on the results of unit testing.
- Analyze the results of unit testing and record the test and analysis results in appropriate software development files (SDFs).
What is Software Testing?
- Is the process of executing a program with an established set of test cases
- Test cases are generated according to a well defined procedures or techniques
- Testing is a bottom-up process, starts with unit testing, components testing, CSCI testing
Levels of Real Time System Testing

- Unit Testing
- Software Integration Testing
- Software Validation and Verification Testing
- Software / Hardware Integration Testing
- System Testing
Software Testing (Unit Testing)

Set of test cases

Test Driver

Module under test

Stub

Stub

Stub

Data Module
Software Testing

- The test driver is developed based on a set of test cases.
- The driver invokes the module under test for each test case and establishes the test case data/control requirements.
- Stubs are sub-ordinate dummy modules that represent the modules (including global data structures) invoked (or accessed) by the module under test.
- They contain print and return statements.
Software Testing

- Testing techniques consist of
  - Black Box testing, where we focus on testing the software functional requirements, and testing the input/output interfaces

Only inputs and outputs of functions are considered
How outputs are generated based on a set of inputs is ignored
Run a suite of test cases
  - Exhaustive combination of all inputs
  - Corner cases (min, max, avg)
  - Pathological cases (inputs likely to result in error)
**Software Testing**

- **White Box Testing**, where we focus on developing test cases to cover the logical paths through the code (e.g., conditional statements, loops, etc.)
  - Based on developing a control flow graph of the code under test to identify the set of independent paths, and produce a set of test cases to cover these paths
  - Exercises all paths in a module Driven by logic
  - Static Example: Code Inspections, group walkthrough of software logic, inspect code line-by-line (Static Analysis tools)
  - Dynamic Example: Test all the links and buttons on a web page
  - For real-time systems dynamic testing is important to test time constraints, and reactive logical paths
Software Testing

Black Box Testing Techniques

1. Equivalence Partitioning
   Partition the input space into equivalence classes and develop a test case for each class of inputs

2. Boundary value analysis
   develop test cases at the boundaries of the possible input ranges (minimum and maximum values)

The above techniques are important for data processing intensive applications
Sample Equivalence Classes

**Valid data**
- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

**Invalid data**
- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place
Software Testing

- Black Box Testing Techniques (cont.)
  3. Cause-effect graphing
    - Used for control intensive applications,
      - Develops test cases to represent input events (or causes) and the corresponding actions (or effects)
- This technique is used intensively in real-time systems
Software Testing

- Cause-effect graphing consists of the following 4 steps
  1. List and label causes (input-events) and effects (output actions) for a module
  2. Draw a cause-effect graph describing the logical combinations of causes, intermediate causes and resulting effects,
  3. Develop a decision table (causes vs effects) from the graph
  4. Convert each row into a test case, or a set of rows into a testing scenario
Symbols used in the cause-effect graphs

\( c_i = \text{ith cause, } e_i = \text{ith effect} \)

- \( c_i \) causes \( e_i \)
- \( c_i \) does not cause \( e_i \)
- \( c_i \) OR \( c_j \)
- \( c_i \) AND \( c_j \)
Software Testing

- Example: Control motion module in our project (assume the module inputs are a state variable and a control input)

1. List and label all Causes and all effects

<table>
<thead>
<tr>
<th>Causes</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: Start_train</td>
<td>e100: Release all Brakes</td>
</tr>
<tr>
<td>C2: Idle</td>
<td>e200: Engage_Engine</td>
</tr>
<tr>
<td>C3: speed = Max_speed</td>
<td>e300: Disengage_Engine</td>
</tr>
<tr>
<td>C4: Accelerating</td>
<td>e400: Apply Ph. 1 Brakes</td>
</tr>
<tr>
<td>C5: Speed &gt; Max_speed</td>
<td>e500: Release Ph. 1 brakes</td>
</tr>
<tr>
<td>C6: Coasting</td>
<td>e600: Apply Ph. 2 brakes</td>
</tr>
<tr>
<td>C7: Speed &lt;= Min_speed</td>
<td>e700: Apply Safety brakes</td>
</tr>
</tbody>
</table>
Software Testing

This List is obtained form the following C-spec STD

- **Coasting**
  - Speed <= Min_speed / Engage Engine
  - Speed = Max_speed / Disengage Engine

- **Accelerating**
  - Start / RL all Br Engage Engine

- **Decelerating**
  - Speed > Max_speed / Apply Ph. 1 brakes

- **Idle**
  - Stop / Applying ph II

- **stopping**
  - Stop / Applying ph II

- **Applying ph II**
  - Stop / Applying ph II
Develop a Cause effect graph

- c1
- c2
- c3
- c4
- c7
- c10
- c20
- e40
- c100
- c200
- e300
- c30
- e400
- e30
- c5
- c6
**Software Testing**

- Develop a Decision Table

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>e100</th>
<th>e200</th>
<th>e300</th>
<th>e400</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Convert each row to a test case
Scenario testing: A testing scenario combines a set of test cases for testing the system behavior over a period of time.

For example, the following sample run can be used for scenario testing:

```
Start, idle; speed = max_speed, Accelerating;
Speed <= Min_speed, Coasting; Stop, Accelerating;
speed = stop speed; start, Idle; speed = max_speed, Accelerating;
speed > Max_speed, Coasting;
```