CISC327 - Software Quality Assurance

Lecture 12
Black Box Testing
Black Box Testing

• Outline
  – Last time we continued with black box testing and looked at input coverage testing
  – Today we look at the third kind of black box method, output coverage testing, and consider the role of black box methods in unit and integration testing
  – Output coverage methods (exhaustive output testing, output partitioning)
  – Testing multiple input or output streams
  – Black box testing at the unit and integration levels ("gray box" testing)
  – Model-based testing
Output Coverage Testing

• The third kind of black box testing
• **Idea**: Analyze all the possible outputs specified in the functional specification (requirements), create tests to cause each one

• More **difficult** than input coverage
  – Must analyze requirements to figure out what **input** is required to produce each **output**
  – This can be a complex and time consuming process

• **But** can be very effective in finding problems, because it requires a **deep understanding** of the requirements
Output Coverage Testing

- Output coverage testing is definitely different from input coverage
- For example, suppose the requirements say:
  - "Output 1 if two input integers are equal, 0 otherwise"
- This specification allows two integer inputs, so if we do input partitioning, we have the test cases:
  - Numbers equal, numbers not equal, first number zero / positive / negative, second number zero / positive / negative
- Whereas we can do exhaustive output testing with only two test cases:
  - Output 1, output 0
Exhaustive Output Testing

• More practical than Input Testing?
  – Exhaustive output testing makes one test for every possible output
  – Practical more often than exhaustive input testing
    • Many programs reduce or summarize input data (like the previous example)
  – But still impractical in general
    • Most programs have an infinite number of different possible outputs
Output Partitioning

- Output **partitioning** is like input partitioning, only we analyze the possible **outputs**
- In a fashion similar to input partitioning, we partition all the possible **outputs** into a set of **equivalence classes** with something in common
  - "Given as input two integers $x$ and $y$, output all the numbers smaller than or equal to $x$ that are evenly divisible by $y$. If either $x$ or $y$ is zero, then output zero."
Output Partition Testing

- "Given as input two integers \( x \) and \( y \), output all the numbers smaller than or equal to \( x \) that are evenly divisible by \( y \). If either \( x \) or \( y \) is zero, then output zero."

- The output is a list of integers, so we might partition into the following cases:

<table>
<thead>
<tr>
<th>output values</th>
<th>zero</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>all zero</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>some zero</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>all positive</td>
<td>P7</td>
<td>P8</td>
<td>P9</td>
</tr>
<tr>
<td>all negative</td>
<td>P10</td>
<td>P11</td>
<td>P12</td>
</tr>
<tr>
<td>mixed</td>
<td>P13</td>
<td>P14</td>
<td>P15</td>
</tr>
</tbody>
</table>
Output Partition Testing: Designing Inputs

• Once we have the output partitions, we must design inputs to cause outputs in each class

• This is difficult and time-consuming
  – The biggest drawback to output coverage testing!

• Sometimes, we discover that we cannot find such an input
  – This implies an error or oversight in either the requirements or in the partition analysis
Multiple Input or Output Streams

• A Separation of Concerns
  – For both input and output coverage methods, if there is more than one input or output stream or file, we must create separate coverage tests for each one
  – Effectively, what we do is treat each separate file or stream as a pre-made input or output partition, within which we make a separate set of smaller partitions
Multiple Input or Output Streams

• A Separation of Concerns
  – For example, if we were output-coverage testing our Back Office, then there are two separate outputs to test: the New Master Accounts File, and the New Valid Account List
  • In such a case, we should create separate output partition test sets for each
  – This is consistent with the partitioning system in general—we assume that each class of input or output is independent of the others
Black Box Testing at Different Levels

• Recall that there can be many levels of testing, corresponding to the stages of software development

• In particular, black box testing of all kinds can be used at every level of software development
Black Box Testing at Different Levels

• At the **system testing** level, we have seen how to create functional, input, and output coverage tests for the entire program's functional specification (the **requirements** for the software)
  
  – This is **pure** black box testing, because it does not require us to have done any development at all
“I do not understand why everything in this script course must inevitably explode be a box.”
"Gray" Box Testing

• If we already have a design...
  – If we allow ourselves to wait until we have an architectural (class level) design, or even a detailed (method level) design, then we can use black box testing at each of those levels as well
  – Since we can see the design of the software, black box testing at these levels is not really "pure" black box
  – For that reason, it is sometimes called "gray box" testing
"Gray" Box Testing

• If we already have a design...
  – At the architectural (class) design level, we can apply the same black box coverage analyses to the interface of each class to create class level black box tests (a.k.a. interface tests)
  – If we know how a software code fragment is written, we can design tests with that in mind
"Gray" Box Testing

• Imagine that our **divisors** example program was used in an interface
  – Possible case:
    • A user clicks on a button that brings up a dialog box with two inputs
    • Those two inputs are passed to the divisors interface, which then returns the result
  – If we know immediately that the interface code disregards **negative numbers**, we can design our tests differently
Model-Based Testing

- **Model-Driven Engineering (MDE)**
  - A modern new black-box method is *model-based testing*, part of MDE
  - Model-based testing does not use a specification, but rather a formal *state model* of the process implemented by the program
  - State models are *high-level abstractions* (simplifications) of the program’s intent, usually expressed at the level of the problem domain rather than the computer
  - State models ignore implementation details, but retain *essential states* of the process
Model-Based Testing

• Model-Driven Engineering
  – For example, the following might be a state model of the login aspect of the QBASIC Front End

![Diagram of the state model](image-url)
Model-Driven Engineering

• Essentially, models form formal (mathematical) specifications of the process to be implemented

• Formal models can be used in several ways
  – To verify that the model (formal specification) is itself correct, using model checking (NASA, Airbus) (CISC 422)
  – To generate some or all of the implementation from the formal model, if it is detailed enough (General Motors)
  – To test that the implementation is consistent with the formal model (model-based testing)
Model-Based Testing

• The basic idea of model-based testing is that the model is smaller and simpler than the code, so we can generate far fewer tests to cover it than we would for the implementation
  – For example, white-box testing

• The model also encodes the entire specification, so we know that if we make a set of tests to cover the model, every essential requirement is tested
Model-Based Testing

• Because the model is formal, we can automatically generate the tests, then run them against the implementation to verify that it correctly implements the model (which was itself verified using model-checking)
  – Of course, this is the ideal situation
  – In practice, models may be partial or may address only some aspect of the requirements
Model-Based Testing

• We can generate tests to cover every state in the model, every state transition in the model, every path in the model, or so on

• In essence, this uses white-box coverage methods, but for the model rather than the code, automatically yielding complete, high-quality functionality tests
Model-Based Testing

• **Example:** We can cover all the states of our example model for login using only two tests:

T1: login machine logout

T2: login agent logout

The diagram illustrates the model with states 0, 1, 2, and 3, and transitions for login, logout, machine, agent, and actions deposit, withdraw, transfer, createacct, deleteacct, deposit, and withdraw.
Model-Based Testing

• Advantages:
  – Automatic test generation
  – Tests against a formal specification
    (the verified model)
  – Covers all essential behaviour
  – Still a black box method, with all its advantages
    • Requires only the model, not the code
  – Yields high confidence in the correctness of the final code
Model-Based Testing

• Disadvantages:
  – Heavyweight test method, probably only practical for safety-critical and security-critical applications (aerospace, automotive, etc.)
Summary...

- **Black Box Testing**
  - Output coverage methods analyze the set of possible **outputs** specified and create tests to cover them
  - **Exhaustive output** testing and **output partitioning** are similar but distinct from input coverage methods
  - **Multiple** input or output streams / files are handled by treating them as a predefined partitioning boundary
Summary.

- **Black Box Testing**
  - We can also apply black box methods at lower levels of testing, if we have the architecture or detailed design
  - Model-driven engineering (MDE) can assist to automatically generate high quality tests using model-based testing