Test classification

- By stages
  - Unit testing
  - Integration testing
  - Acceptance testing
  - Regression testing
- By domains
  - UI testing
  - Web testing
  - Mobile testing
  - CPS testing
  - AI application testing

Functional testing

- Usually black box
- Concerned with system functionality and features
- Not concerned with implementation details
- Takes the view point of the user

- Functional testing is hard in general
  - Human efforts are needed to determine the expected output

Enumerating attributes

- Test cases should test every aspect of the requirements
  - Each detail in the requirements is called an attribute
- Good first step: enumerate the attributes
  - Circle all of the important points in the requirements document (product backlog)
- Often many implicit attributes
- Example: A program that determines if three integers form a scalene, isosceles, or equilateral triangle

Black-box testing

- Testers provide the system with inputs and observe outputs
- Cannot see:
  - Source code
  - Internal data
  - Design documentation

Equivalence classes

- Brute force, every input permutation testing is usually impossible
  - It's also pointless
- Divide possible inputs into groups that are treated similarly by an algorithm
  - Called equivalence classes
  - Tester only needs to run one test per equivalence class

Examples

- Input a month (1-12)
  - Equivalence classes are:
  - [-∞..0], [1..12], [13..∞]
- Valid input is one of ten strings representing a type of fuel
  - Eleven classes – one for each string, one representing all other strings
- A grading system
  - A [90,100], B [80,90), …
- Compiler parser
  - An Id consists of letters and numbers, must start with letter
  - The length cannot exceed 30

Combinations

- Combinatorial explosion usually means you cannot test every possible input value
  - E.g., 4 inputs, 5 values ⟶ 5^4 = 625 combinations
- Try to at least run one test per equivalence class
- Test combinations where an input is likely to affect the interpretation of another
- Pair-wise combinations
  - Attribute A has values a1, a2, a3
  - Attribute B has b1, b2, b3
  - Attribute C has c1, c2
  - How many cases do I need for single value coverage
  - How many cases do I need for pair-wise coverage

Boundaries

- A lot of errors in software occur at the boundaries of equivalence classes
- Test at the extremes of each class
- Example: input a month (1-12)
  - Test 0, 1, 12, and 13

Test classification

- By properties under testing
  - ~~Functional testing~~
  - Higher order testing
    - Performance
    - Stress
    - Security
    - Usability
    - …
- By methods
  - ~~Black-box testing~~
    - ~~Equivalence partitioning and boundary value analysis, random testing~~
  - White-box testing
    - Coverage based testing: statement, edge coverage, and path coverage
  - Gray-box testing: fuzzing
  - Mutation testing
  - Meta-morphic testing
  - Differential testing

Performance testing

- To determine how well the system meets all of its stated and implied performance requirements
- Also determines the resource limitations and inefficiencies of the system
  - Include equipment, time, environment, data, algorithms, etc

Stress testing (volume testing)

- Put the system through a large volume of inputs
- Performance testing concerns the derivative of resources, whereas stress testing directly concerns the resources

Announcements (10/7)

- Homework 2 due 10/9
  - Search for keywords denoting design patterns such as "builder", "visit", "publish"
  - Don't list all the attributes for a class, just the critical ones (e.g., those denoting associations with another class)

Security Testing

- Fuzzing by AFL or Google Fuzzer
  - Randomly perturbs an initial set of seed inputs by adding, removing, mutating bits and bytes, guided by coverage changes.
  - The goal is to look for crashes
  - Fully automated
  - Runs in the scale of days and weeks
- Symbolic execution by KLEE or Java PathFinder
  - It automatically derives a legal input that causes exceptions
  - Starting from a seed input or no input at all, it derives new inputs but changing the execution path of the input
  - It is based on COTS reasoning engines such as Microsoft Z3
  - Runs in days and weeks
  - Less scalable than fuzzing but can find deeper bugs
- Penetration testing by POC

Usability testing (user study)

- Extremely important
- 3-5 people (recruited users or domain experts)
  - Hall-way testing – grab some users in the hallway to test your software
  - Amazon Mechanical Turk
  - …
- Interview or video taping
- Predetermined set of (acceptance) test cases
- Predefined set of questions

Test classification

- By properties under testing
  - ~~Functional testing~~
  - ~~Higher order testing~~
    - ~~Performance~~
    - ~~Stress~~
    - ~~Security~~
    - ~~Usability~~
    - …
- By methods
  - ~~Black-box testing~~
    - ~~Equivalence partitioning and boundary value analysis, random testing~~
  - White-box testing
    - Coverage based testing: statement, edge coverage, and path coverage
  - Gray-box testing: fuzzing
  - Mutation testing
  - Meta-morphic testing
  - Differential testing

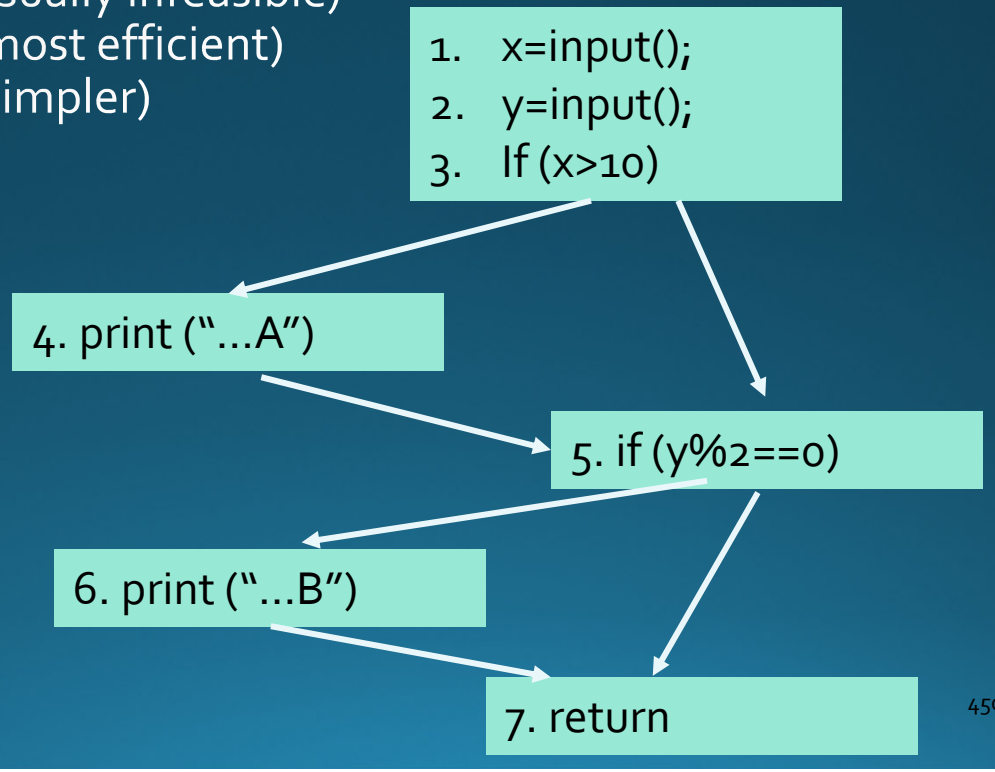
White-box testing

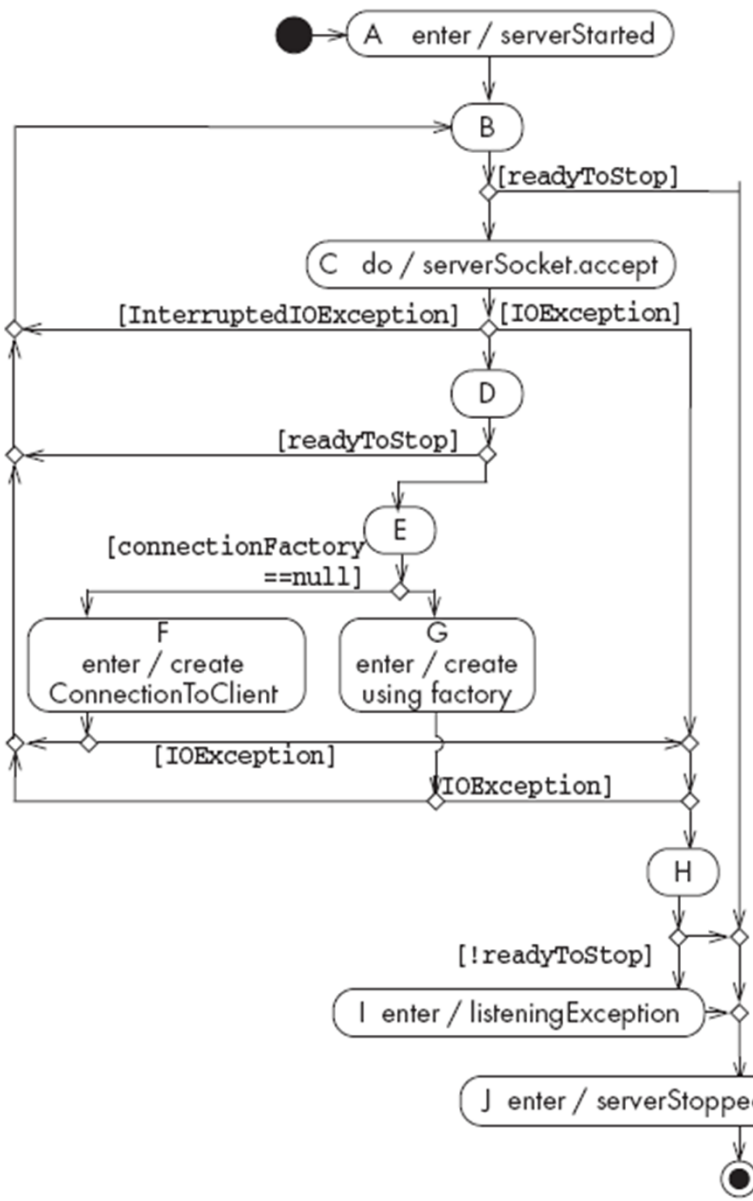
- Also called “glass-box” or “structural” testing
- Testers have access to system internals
  - Design documents
  - Code
  - Run time data and tracing
- Individual coders often informally employ white-box testing

Control flow graph

- Consecutive statements with single entry and single exit, that is, no branching (out or in) form a node
- Edges denote control flow between nodes
- Testing strategy has to reach a targeted coverage of statements and branches
  - Cover all possible paths (usually infeasible)
  - Cover all possible edges (most efficient)
  - Cover all possible nodes (simpler)

```
1. x= input();
2. y= input ();
3. if (x>10)
4.    print ("reach A");
5. If (y % 2=0)
6.    print ("reach B");
7.  return
```






```
readyToStop= false;
serverStarted();      // A
try
{
  while(!readyToStop)  // B
  {
    try
    {
      Socket clientSocket = serverSocket.accept();  // C
      synchronized(this)
      {
        if (!readyToStop) // D
        {
          if (connectionFactory == null) {  // E
            new ConnectionToClient(         // F
              this.clientThreadGroup, clientSocket, this);
          } else {
            connectionFactory.createConnection(    // G
              this.clientThreadGroup, clientSocket, this);
          }
        }
      }
    }
    catch (InterruptedIOException exception) { }
  }
}
catch (IOException exception)
{
  if (!readyToStop)   // H
  {
    listeningException(exception);  // I
  }
}
finally
{
  readyToStop = true;  // J
  connectionListener = null;
  serverStopped();
}
```

Statement coverage

- To be sure that all parts (hardware) and all statements (software) are used or executed at least once
- If you can't use it (hardware) or execute it (software), then why did you include it?
- May be a good reason – is it part of a reuse package and too expensive to remove?
- Gcov and Jcov

```
1. x= input();
2. y= input ();
3. if (x>10)
4.     print ("reach A");
5. If (y % 2=0)
6.     print ("reach B");
7.  return
```

Decision coverage

- Check to be sure all decision points in the product are exercised at least once for each possible outcome

```
1. x= input();
2. y= input ();
3. if (x>10)
4.     print (“reach A”);
5. If (y % 2=0)
6.     print (“reach B”);
7.  return
```

Path coverage

- Exercise the system in such a way that all possible paths through the decision points, both hardware and software, are used at least once

```
1. x= input();
2. y= input ();
3. if (x>10)
4.     print (“reach A”);
5. If (y % 2=0)
6.     print (“reach B”);
7.  return
```

Observations

- 100% coverage does not guarantee a program is bug free
- Statement coverage and decision coverage are widely used
- 60% statement coverage is good in practice for complex software
- Statement coverage is weakest and path coverage is strongest and also most expensive

Test classification

- By properties under testing
  - ~~Functional testing~~
  - ~~Higher order testing~~
    - ~~Performance~~
    - ~~Stress~~
    - ~~Security~~
    - ~~Usability~~
    - …
- By methods
  - ~~Black-box testing~~
    - ~~Equivalence partitioning and boundary value analysis, random testing~~
  - ~~White-box testing~~
    - ~~Coverage based testing: statement, edge coverage, and path coverage~~
  - ~~Gray-box testing: fuzzing~~
  - Mutation testing
  - Meta-morphic testing
  - Differential testing

Mutation testing

- Premise: test data sets are good if they can detect small random changes to the software
- Plant errors in the code and check to see if they are detected
- Good for build regression tests
  - Helps find "mutants" generated during maintenance
- Requires significant resources

```
1. x= input();
2. y= input ();
3. ~~if (x>10)~~ if (x>=10)
4.    print ("reach A");
5. If (y % 2=0)
6.    print ("reach B");
7.  return
```

Testcase: x=12, y=2, output =" reach A reach B"

Metamorphic Testing

- How to test a ML/AI application?

Metamorphic Testing

- How to test a ML/AI application?
  - A small delta in input only induce small delta in output

Differential testing

- Assume there are multiple implementations of the same functionalities (with different performance)
- When they are provided with the same input, they should produce the same output
- Very useful in practice

Test classification

- By stages
  - Unit testing
  - Integration testing
  - Acceptance testing
  - Regression testing
- By domains
  - UI testing
  - Web testing
  - Mobile testing
  - CPS testing
  - AI application testing

Integration testing strategies

- Big Bang – All at once
- Top Down
- Bottom Up
- Sandwich

Big Bang – All at once

- Each module is, hopefully, first tested in isolation
- The entire system is then assembled and tested as a unit
- These people dream a lot and also buy lots of lottery tickets

Big Bang disadvantages

- Very low probability of working
- Don't know if the interfaces are correct until late in the testing process
- Stubs and drivers are both needed to test the modules in isolation
  - If it even happens
- Very difficult to isolate defects

Top down testing

- Program modules are merged from the top to the bottom
- As each new module is integrated, the entire system is tested

Stubs

- Stubs are needed to simulate missing lower level modules
  - Pieces of code that have the same interface as the lower level module(s)
  - Do not perform real computations or manipulate real data
- Writing stubs can be expensive in itself

Top down improvement

- Test each module in isolation with both drivers and stubs before integrating them
- Prioritize the integration
- Build a working skeleton from top to bottom, then add the "flesh"
- Closer to integration testing

Top down advantages

- User interface is tested early
- Opportunity to involve the customer in early testing of the product
- If done right, an early working prototype can be made available for product validation
  - Is it the right product?

Top down disadvantages

- Bottom levels are rarely tested enough
- Testing time starts slowly and then grows at a rapid rate near the end
  - Usually occurs at about the same time as the project's money and time are running out

Bottom up testing

- Modules are merged and tested from the bottom to the top
- Higher level modules are added and tested in combination with previously tested lower level modules

Drivers

- Drivers are required to simulate the missing calls from the higher level modules
  - Simple programs designed specifically for testing that make calls to the lower layers
- Similar role to stubs in top down testing
  - Can also be time consuming to write

Bottom up improvement

- Test each module in isolation with drivers and stubs before integrating
- Prioritize integration of the modules
  - Build a working skeleton from bottom to top, then add flesh.
  - Stub out missing modules

Bottom up advantages

- Lower level modules are fairly well tested
- Testing can proceed early
  - And in parallel
  - And on separate parts of the system
- Good for real-time systems where many of the high priority, high risk modules are terminal

Bottom up disadvantages

- Top level UI is tested last
  - Cost of fixing it may kill the project
- Top level is rarely tested enough
- Product is shipped, then the user gets to complete the testing

Regression testing

- Set or subset of all tests that are re-run after any change or commit
- Cover as much of the system as possible
- Prevent regressions – system reverting to an earlier, incorrect state

Testing and product phases

- Alpha
  - Early, barely complete version of a product
  - Small number of trusted users
    - Understand that there are (potentially major) bugs
- Beta
  - Product is complete, but not thoroughly tested
  - Regular users are recruited to test in a normal work environment
    - Understand software is still "low-quality"

Test classification

- ~~By stages~~
  - ~~Unit testing~~
  - ~~Integration testing~~
  - ~~Acceptance testing~~
  - ~~Regression testing~~
- By domains
  - UI testing
  - Web testing
  - Mobile testing
  - CPS testing
  - AI application testing