Class 14

- Questions/comments
- Testing continued
- Assign (see Schedule for links)
  - Readings on regression testing, prioritization
  - Problem Set 6

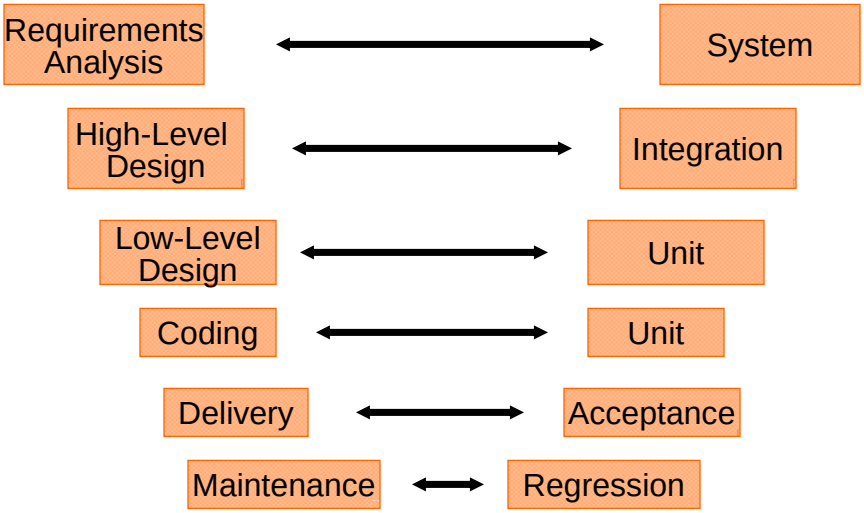
Terms

- V&V
- Failure, error, fault
- Coincidental correctness
- Oracle
- Coverage criteria
- Black box, white box testing
- Test requirements, test specifications, test case

Software Development Phases and Testing

Requirements Analysis Phase:	Develop test plan and system tests; perform technical review
Design Phase:	Develop integration tests; perform technical review
Implementation Phase:	Develop and run unit tests; perform technical review
Integration Phase:	Run integration tests
Maintenance Phase:	Run system tests
	Run regression tests

Software Development Phases and Testing (Graphical View)



White Box vs. Black Box

Black box

- Is based on a functional specification of the software
- Depends on the specific notation used
- Scales because we can use different techniques at different granularity levels (unit, integration, system)
- Cannot reveal errors depending on the specific coding of a given functionality

White box

- Is based on the code; more precisely on coverage of the control or data flow
- Does not scale (mostly used at the unit or small-subsystem level)
- Cannot reveal errors due to missing paths (i.e., unimplemented parts of the specification)

Black Box Testing

Black-Box Testing

- Black-box criteria do not consider the control structure and focus on the domain of the input data
- In the presence of formal specifications, it can be automated (rare exceptions)
- In general, it is a human-intensive activity
- Different black-box criteria
  - Category partition method (read paper)
  - State-based techniques
  - Combinatorial approach
  - Catalog based techniques
  - ...

Black-Box Testing: Exercises

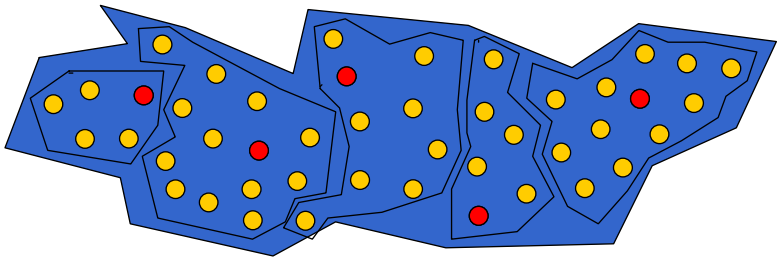
- Identify five test cases for a program that inputs an integer and prints its value
- Identify five test cases for a program that inputs a line of text and breaks it into chunks of up to 80 characters
- Identify five test cases to test a stack implementation

Black-Box Testing: Principles

- Equivalence partitioning of the input space
  - Divide the infinite set of possible inputs into a finite set of classes, with the purpose of picking one or more test cases from each class
- Identification of boundary values
  - Identify specific values (in the partitions) that may be handled incorrectly
- Use of a systematic approach
  - Divide the test generation process into elementary steps

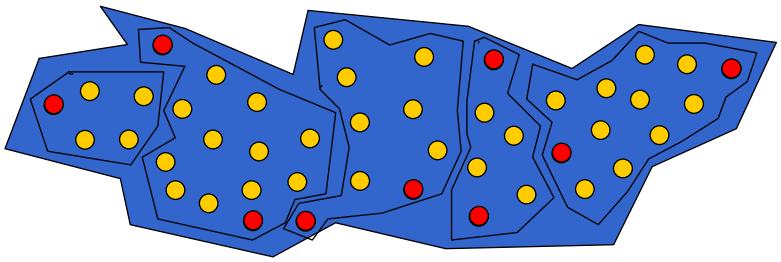
Equivalence Partitioning

- Basic idea: to identify test cases that may reveal classes of errors (e.g., erroneous handling of all inputs > 100)
- Partitioning the input domain in classes from which to derive test cases
- A class is a set of data whose components are likely to be treated homogeneously by the program
- Ideal case: all test cases in a class have the same outcome



Identification of Boundary Values

- Basic idea: errors tend to occur at the boundaries of the data domain ⇒ select test cases that exercise such data boundaries
- Complementary to equivalence partitioning: after identifying the equivalence classes, select for each class one or more boundary values
- Example: if one equivalence class consists of the integer values between 0 and 100, then we may test with inputs –1, 0, 1, 99, 100, and 101



A Systematic Approach

- Deriving test cases from a functional specification is a complex analytical process
- Brute force generation of test cases is generally an inefficient and ineffective approach
- A systematic approach simplifies the overall problem by dividing the process in elementary steps
  - Decoupling of different activities
  - Dividing brain-intensive steps from steps that can be automated
  - Monitoring of the testing process

A Generic Black-Box Technique

Identify independently-testable features

Defining all the inputs to the features

Identify representative classes of values

Which values of each input can be used to form test cases (categories, boundary or exceptional values)

A (partial) model may help (e.g., a graph model)

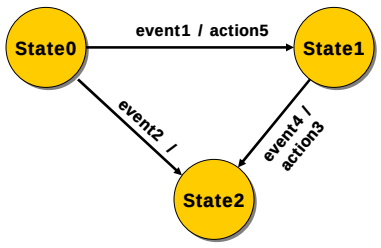
Generate test case specifications

Suitably combining values for all inputs of the feature under test (subset of the Cartesian product—cost, constraints)

Generate and instantiate test cases

Finite State Machines

- Nodes: states of the system
- Edges: transitions between states
- Edge attributes: events and actions



Finite State Machines

- Nodes: states of the system
- Edges: transitions between states
- Edge attributes: events and actions

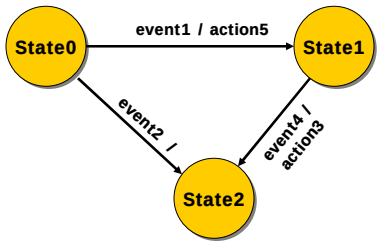


Table of the states

	Event1	Event2	Event4
State0	State1	State2	
State1			State2
State2			

Table of the output

	Event1	Event2	Event4
State0	action5		
State1			action3
State2			

Finite State Machines: Approach

- Identify boundaries of the system
- Identify inputs to the system
- Identify states of the system (trade-off abstraction level/number of states)
- Identify outputs of the system
- Build table of the states (state + event -> state)
- Build table of the outputs (state + event -> output)
- Design tests
- Run tests

Finite State Machines: Some Considerations

- Applicability
  - Menu-driven Software
  - Object-oriented software
  - Device driver
  - Installation software
  - Device-control software
- Limitations
  - Number of states
  - Problems in identifying states, mapping
  - Problem in constructing oracles (What is the state of the system? How do you check events/actions?)

Black-box Testing: Summarizing

- Two main approaches
  - Identification of representative values
  - Derivation of a model
- Most widely used (industry and research)
- No general and satisfactory methodologies
  - Intrinsically difficult
  - Informal specifications

White-Box Testing

White-Box Testing

- Selection of test suite is based on some elements in the code
- Assumption: Executing the faulty element is a necessary condition for revealing a fault
- We'll consider several examples
  - Control flow (statement, branch, basis path, path)
  - Condition (simple, multiple)
  - Loop
  - Dataflow (all-uses, all-du-paths)
  - Fault based (mutation)

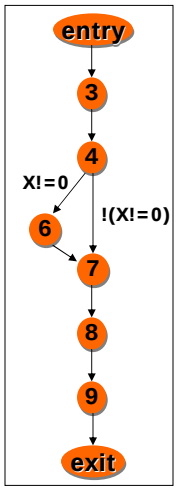
Statement Coverage

Test requirements: Statements in program

$$C_{stmts} = \frac{\text{(number of executed statements)}}{\text{(number of statements)}}$$

Statement Coverage: Example

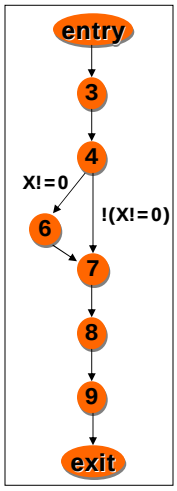
```
1. void main() {
2.    float x, y;
3.    read(x);
4.    read(y);
5.    if (x!=0)
6.       x = x+10;
7.    y = y/x;
8.    write(x);
9.    write(y);
10. }
```



- Identify test cases for statement coverage

Statement Coverage: Example

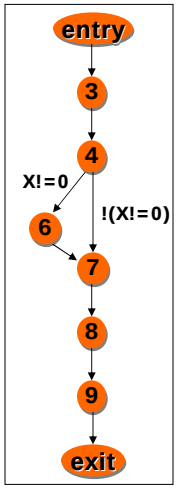
```
1. void main() {
2.    float x, y;
3.    read(x);
4.    read(y);
5.    if (x!=0)
6.       x = x+10;
7.    y = y/x;
8.    write(x);
9.    write(y);
10. }
```



- Test requirements
  - Nodes 3, …, 9
- Test specification
  - (x!=0, any y)
- Test cases
  - (x=20, y=30)
- Such test does not reveal the fault at statement 7
- To reveal it, we need to traverse edge 4-7
- ⇒ *branch coverage*

Branch Coverage: Example

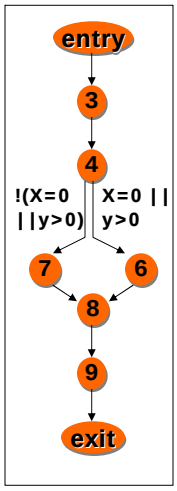
```
1. void main() {
2.    float x, y;
3.    read(x);
4.    read(y);
5.    if (x!=0)
6.      x = x+10;
7.    y = y/x;
8.    write(x);
9.    write(y);
10. }
```



- Test requirements
  - Edges 4-6 and 4-7
- Test specification
  - (x!=0, any y)
  - (x=0, any y)
- Test cases
  - (x=20, y=30)
  - (x=0, y=30)}

Branch Coverage: Example

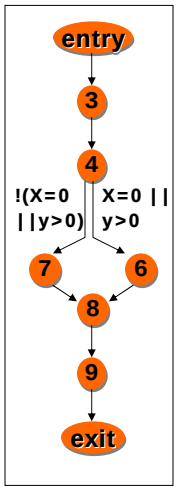
```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=5,y=5), (x=5, y=-5)}

Branch Coverage: Example

```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0||(y>0)
6.     y = y/x;
7.   else x = y+2/x;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=5,y=5), (x=5, y=-5)}
- The test suite is adequate for branch coverage, but does not reveal the fault at statement 6
- Predicate 4 can be true or false operating on only one condition

⇒ Basic c*ondition coverage*

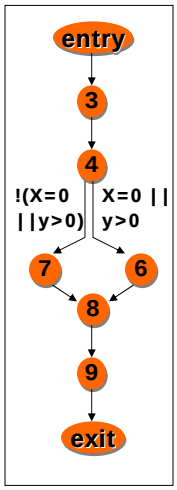
Basic Condition Coverage

Test requirements: Truth values assumed by basic conditions

C_{bc} = (number of boolean values assumed by all basic conditions) / (number of boolean values of all basic conditions)

Basic Condition Coverage: Example

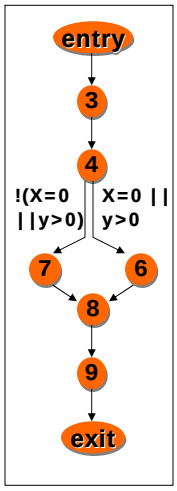
```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=0,y=-5), (x=5, y=5)}

Basic Condition Coverage: Example

```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=0,y=-5), (x=5, y=5)}
- The test suite is adequate for basic condition coverage, but it does not reveal the fault at statement 6
- The test suite is not adequate for branch coverage.

⇒*Branch and condition coverage*

Branch and Condition Coverage

Test requirements: Branches and truth values assumed by basic conditions

if ((a || b) && c) { … }

a	b	c	Outcome
T	T	T	T
F	F	F	F

Compound Condition Coverage

Test requirements: All possible combinations of basic conditions

Very thorough, but also very expensive for non-trivial programs.

Compound Condition Coverage: Example

((((a || b) && c) || d) && e)

- How many test requirements?

Compound Condition Coverage: Example

((((a || b) && c) || d) && e)

Test case	a	b	c	d	e
1	True	-	True	-	True
2	False	True	True	-	True
3	True	-	False	True	True
4	False	True	False	True	True
5	False	False	-	True	True
6	True	-	True	-	False
7	False	True	True	-	False
8	True	-	False	True	False
9	False	True	False	True	False
10	False	False	-	True	False
11	True	-	False	False	-
12	False	True	False	False	-
13	False	False	-	False	-

Compound Condition Coverage

- Advantage for short-circuit operator is that it requires very thorough testing without considering all the combinations
- Disadvantage is to determine the minimum number of test cases required

- The number of test cases required for complex conditions can be substantial (2^n in the worst case!)

Modified Condition/Decision Coverage (MC/DC)

- MC/DC criterion requires that each basic condition be shown to independently affect the outcome of each decision.
- For each basic condition C, there are two test cases in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to True for one of those test cases and False for the other