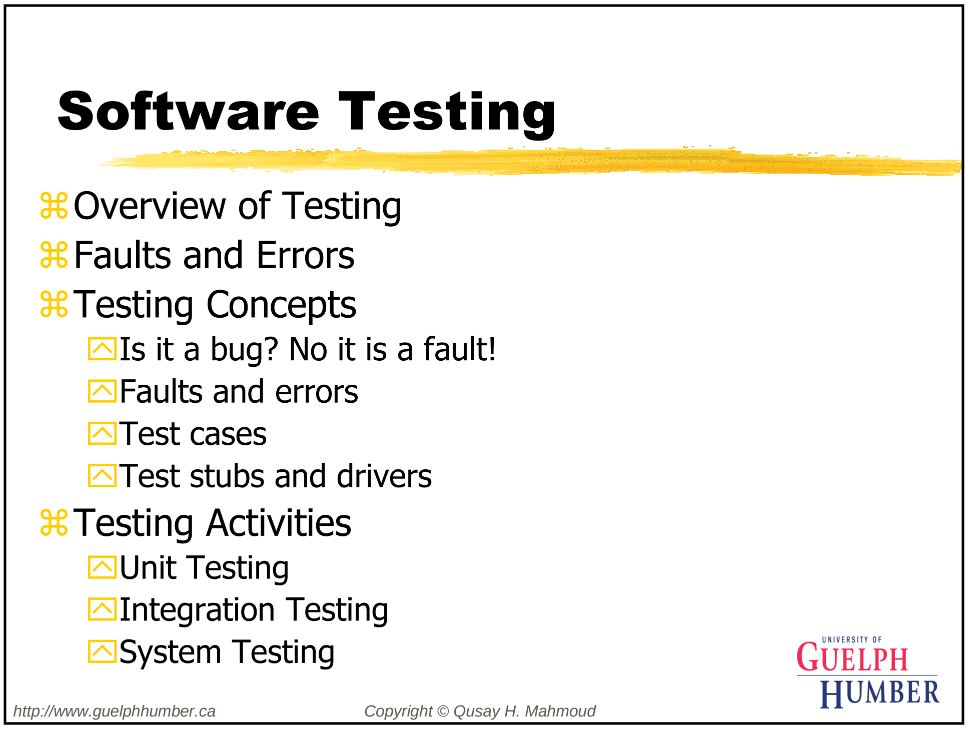
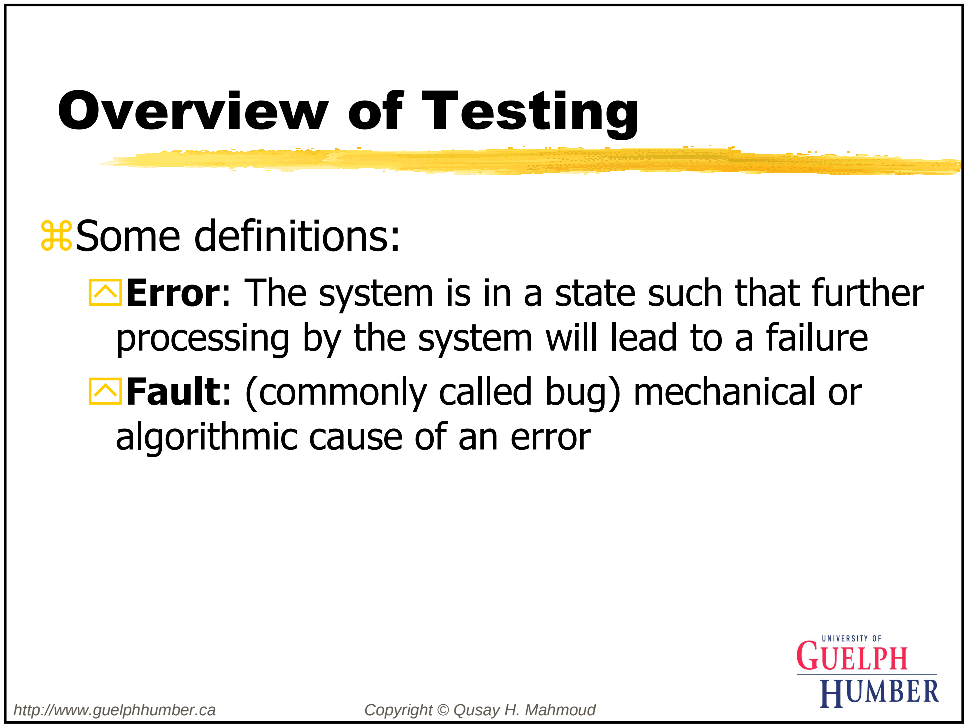


Software Testing

- Overview of Testing
- Faults and Errors
- Testing Concepts
  - Is it a bug? No it is a fault!
  - Faults and errors
  - Test cases
  - Test stubs and drivers
- Testing Activities
  - Unit Testing
  - Integration Testing
  - System Testing

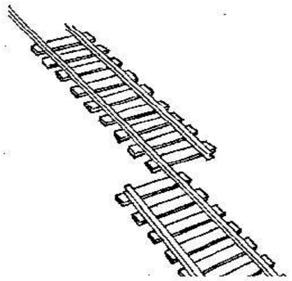


Overview of Testing

- Some definitions:
  - **Error**: The system is in a state such that further processing by the system will lead to a failure
  - **Fault**: (commonly called bug) mechanical or algorithmic cause of an error

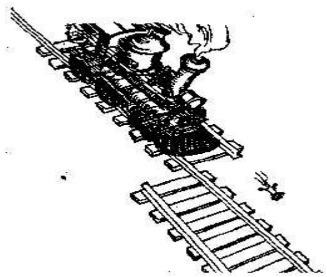
Faults and Errors (1)

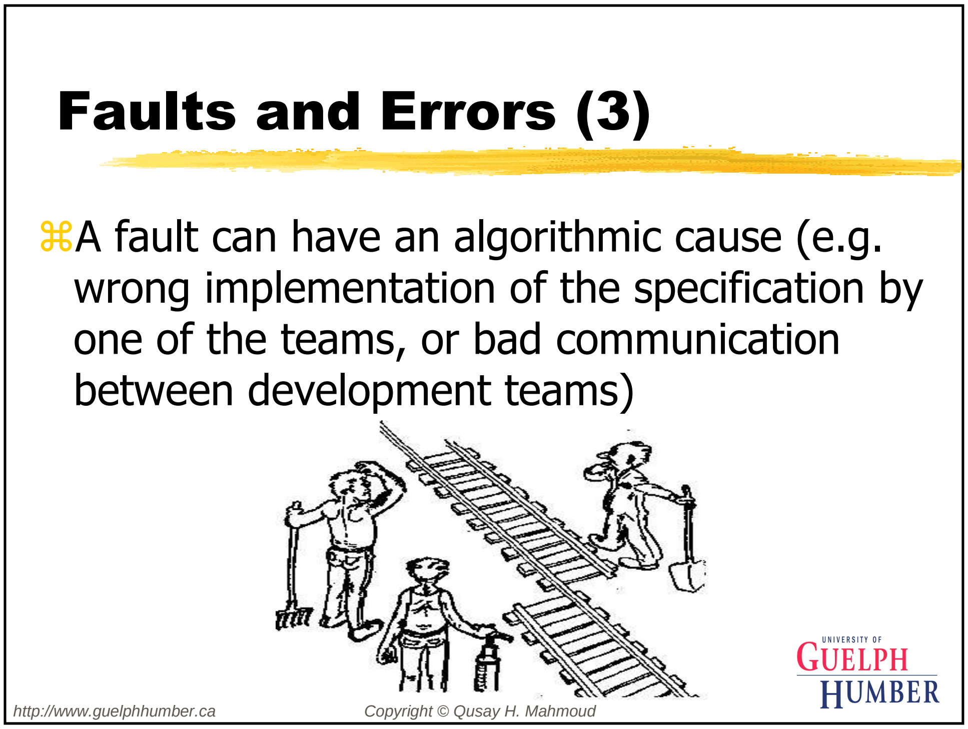
- Example of a fault (bug or defect) which is a design of coding mistake that may cause abnormal component behavior



Faults and Errors (2)

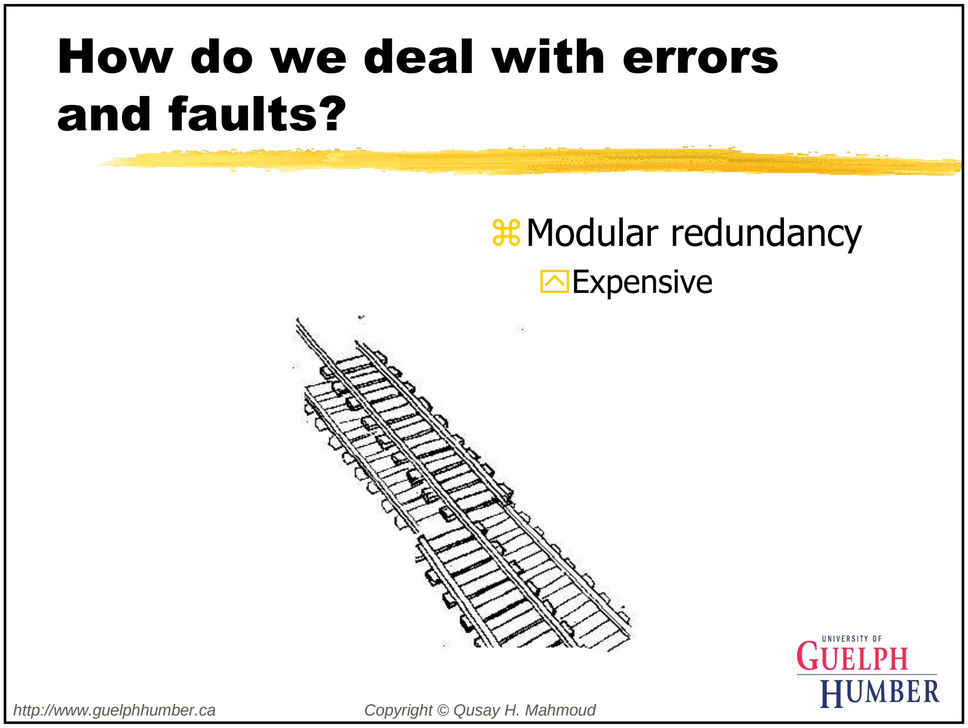
- Example of an error, which is a manifestation of a fault during the execution of a system





Faults and Errors (3)

- A fault can have an algorithmic cause (e.g. wrong implementation of the specification by one of the teams, or bad communication between development teams)

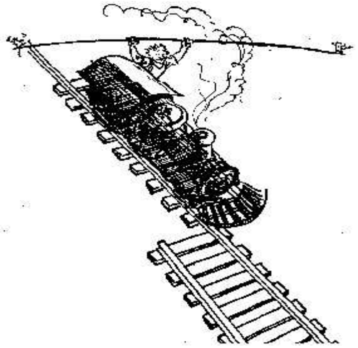


How do we deal with errors and faults?

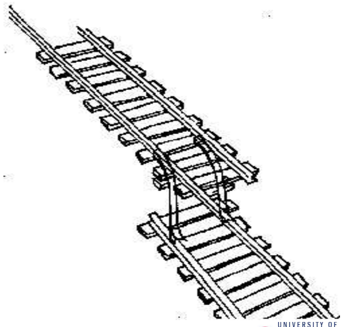
- Modular redundancy
  - Expensive

How do we deal with errors and faults?

- Declare a bug as a feature (MS)

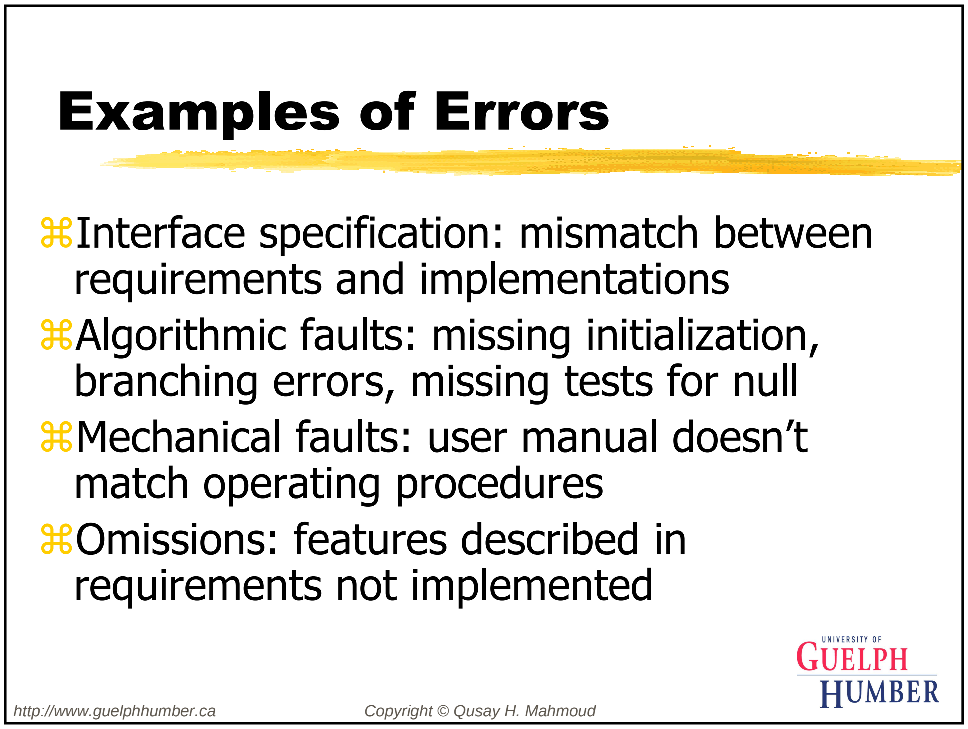


- Patching
  - Slows down performance



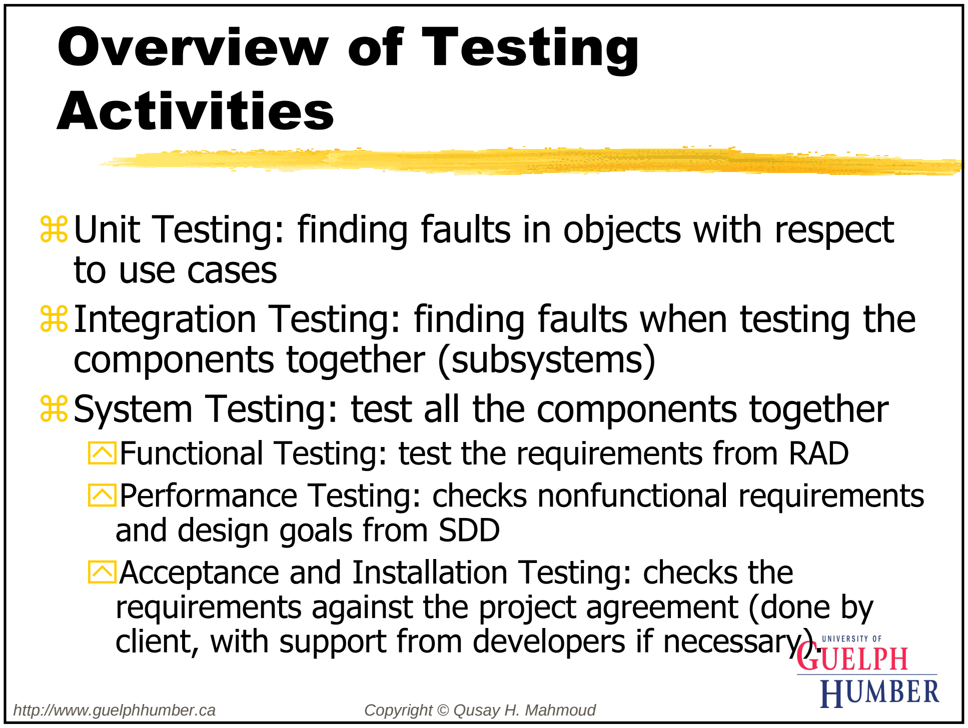
Software Testing

- What is it?
  - Systematic attempt to find errors in a planned way
    - Software Requirements Document vs. observed behavior
- Goals:
  - Maximize the number of discovered faults
  - Demonstrating that errors are not present
  - Dijkstra: show the presence of faults but not their absence
  - Demonstrating that the software can be depended upon.



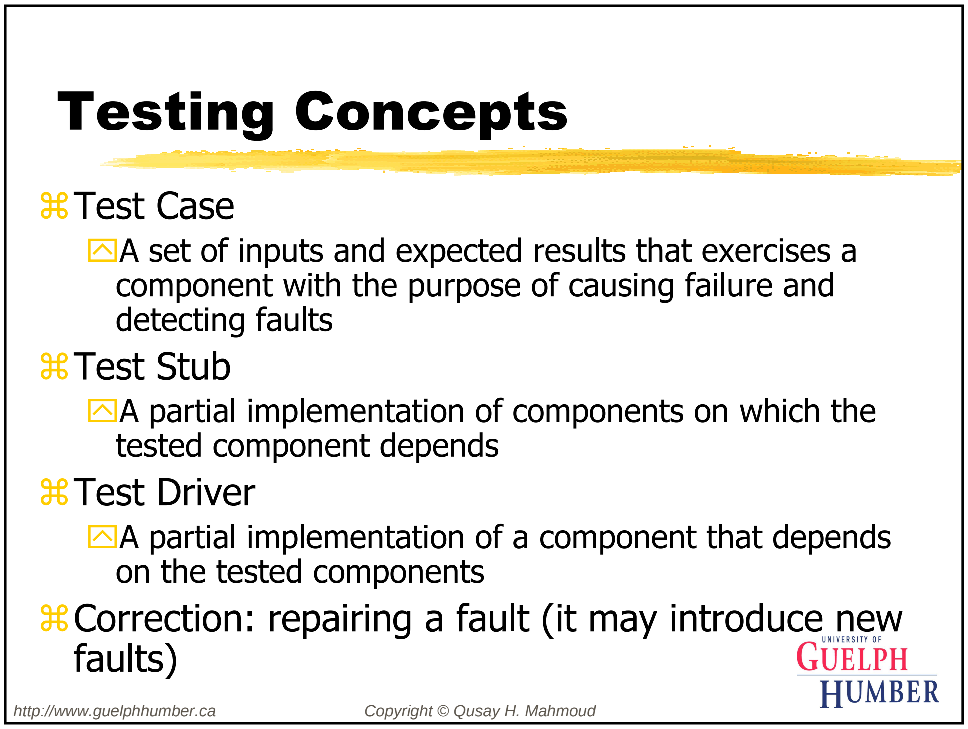
Examples of Errors

- Interface specification: mismatch between requirements and implementations
- Algorithmic faults: missing initialization, branching errors, missing tests for null
- Mechanical faults: user manual doesn’t match operating procedures
- Omissions: features described in requirements not implemented



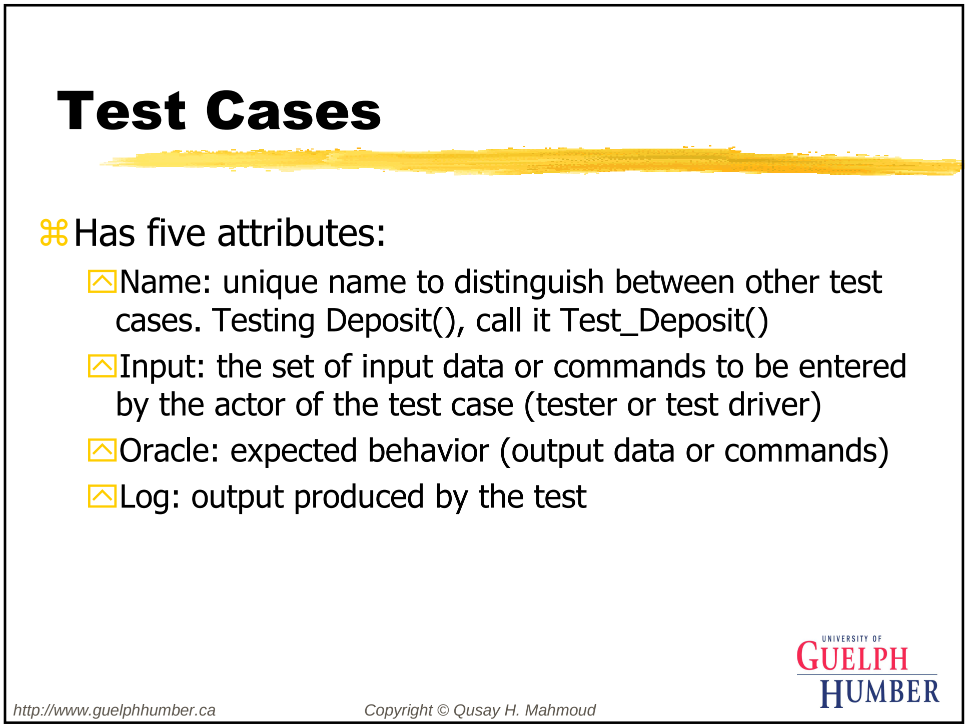
Overview of Testing Activities

- Unit Testing: finding faults in objects with respect to use cases
- Integration Testing: finding faults when testing the components together (subsystems)
- System Testing: test all the components together
  - Functional Testing: test the requirements from RAD
  - Performance Testing: checks nonfunctional requirements and design goals from SDD
  - Acceptance and Installation Testing: checks the requirements against the project agreement (done by client, with support from developers if necessary)



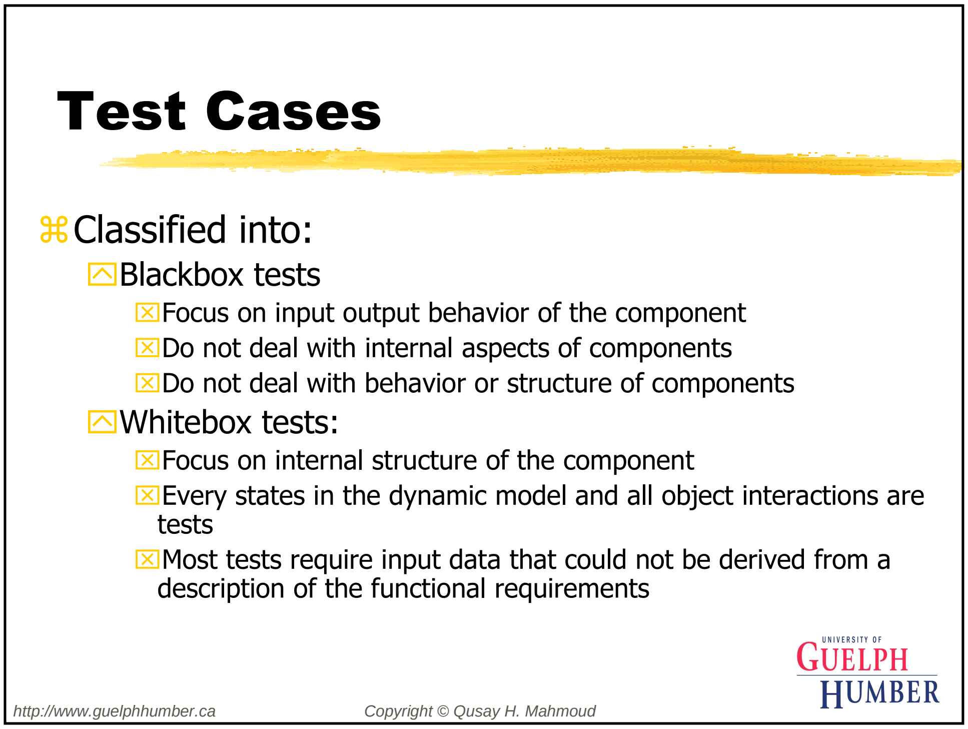
Testing Concepts

- Test Case
  - A set of inputs and expected results that exercises a component with the purpose of causing failure and detecting faults
- Test Stub
  - A partial implementation of components on which the tested component depends
- Test Driver
  - A partial implementation of a component that depends on the tested components
- Correction: repairing a fault (it may introduce new faults)



Test Cases

- Has five attributes:
  - Name: unique name to distinguish between other test cases. Testing Deposit(), call it Test_Deposit()
  - Input: the set of input data or commands to be entered by the actor of the test case (tester or test driver)
  - Oracle: expected behavior (output data or commands)
  - Log: output produced by the test

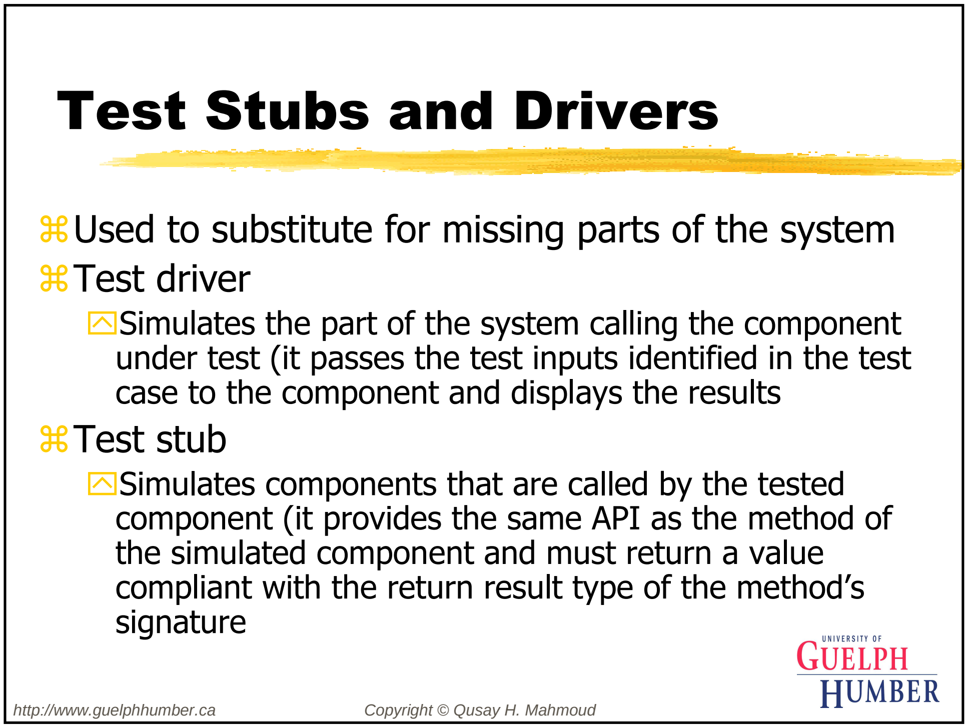


Test Cases

- Classified into:
  - Blackbox tests
    - Focus on input output behavior of the component
    - Do not deal with internal aspects of components
    - Do not deal with behavior or structure of components
  - Whitebox tests:
    - Focus on internal structure of the component
    - Every states in the dynamic model and all object interactions are tests
    - Most tests require input data that could not be derived from a description of the functional requirements

UNIVERSITY OF GUELPH HUMBER

http://www.guelphhumber.ca Copyright © Qusay H. Mahmoud

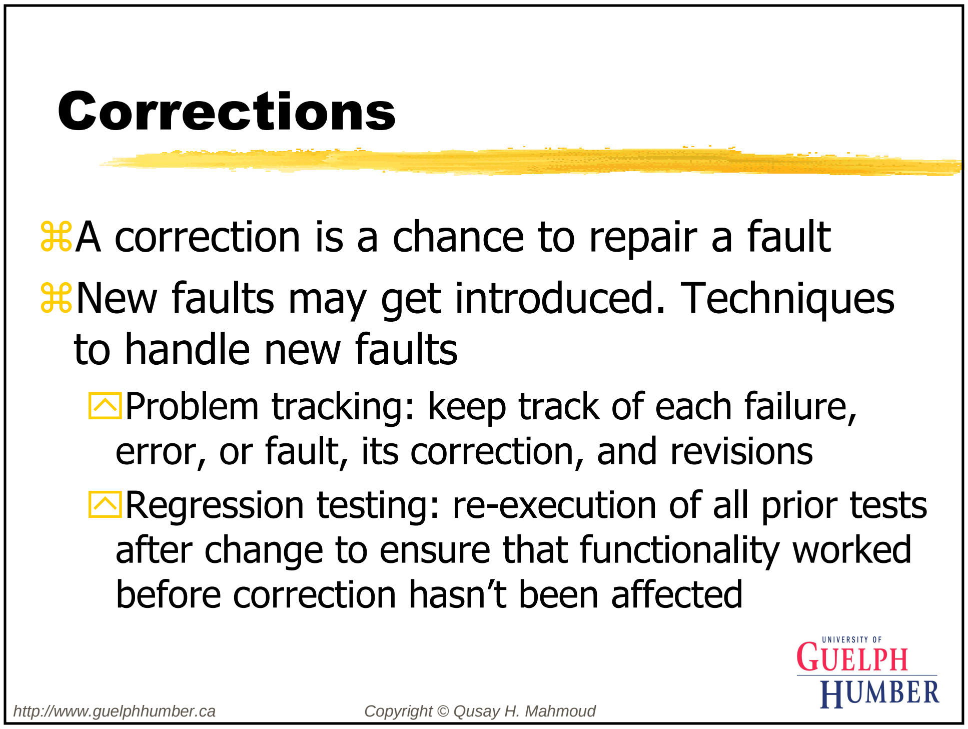


Test Stubs and Drivers

- Used to substitute for missing parts of the system
- Test driver
  - Simulates the part of the system calling the component under test (it passes the test inputs identified in the test case to the component and displays the results
- Test stub
  - Simulates components that are called by the tested component (it provides the same API as the method of the simulated component and must return a value compliant with the return result type of the method’s signature

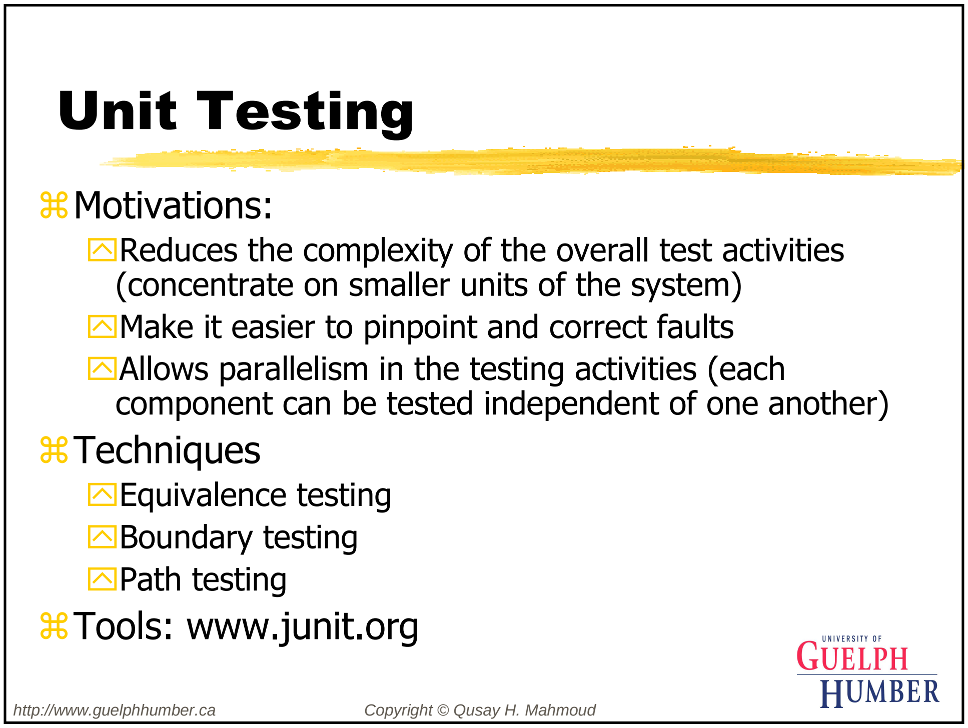
UNIVERSITY OF GUELPH HUMBER

http://www.guelphhumber.ca Copyright © Qusay H. Mahmoud



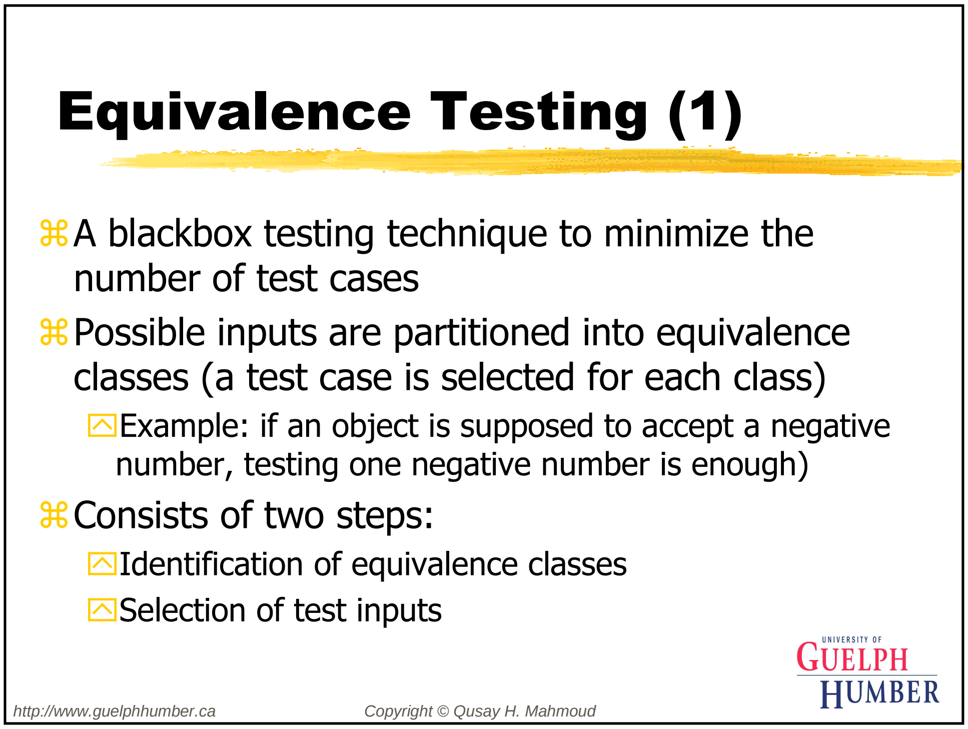
Corrections

- A correction is a chance to repair a fault
- New faults may get introduced. Techniques to handle new faults
  - Problem tracking: keep track of each failure, error, or fault, its correction, and revisions
  - Regression testing: re-execution of all prior tests after change to ensure that functionality worked before correction hasn't been affected



Unit Testing

- Motivations:
  - Reduces the complexity of the overall test activities (concentrate on smaller units of the system)
  - Make it easier to pinpoint and correct faults
  - Allows parallelism in the testing activities (each component can be tested independent of one another)
- Techniques
  - Equivalence testing
  - Boundary testing
  - Path testing
- Tools: www.junit.org

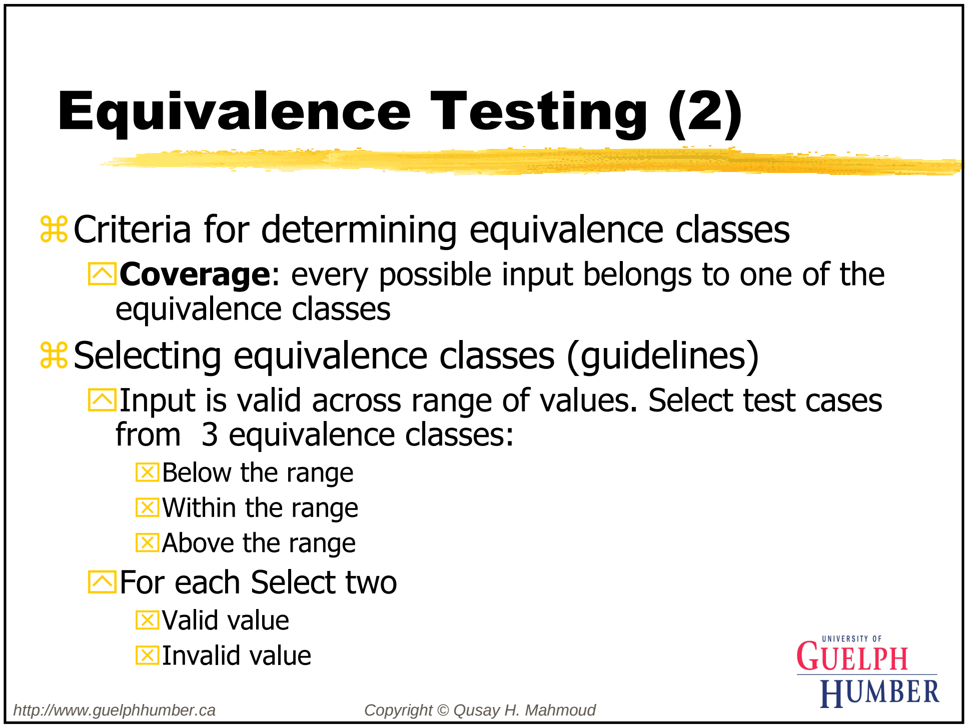


Equivalence Testing (1)

- A blackbox testing technique to minimize the number of test cases
- Possible inputs are partitioned into equivalence classes (a test case is selected for each class)
  - Example: if an object is supposed to accept a negative number, testing one negative number is enough)
- Consists of two steps:
  - Identification of equivalence classes
  - Selection of test inputs

http://www.guelphhumber.ca

UNIVERSITY OF GUELPH HUMBER

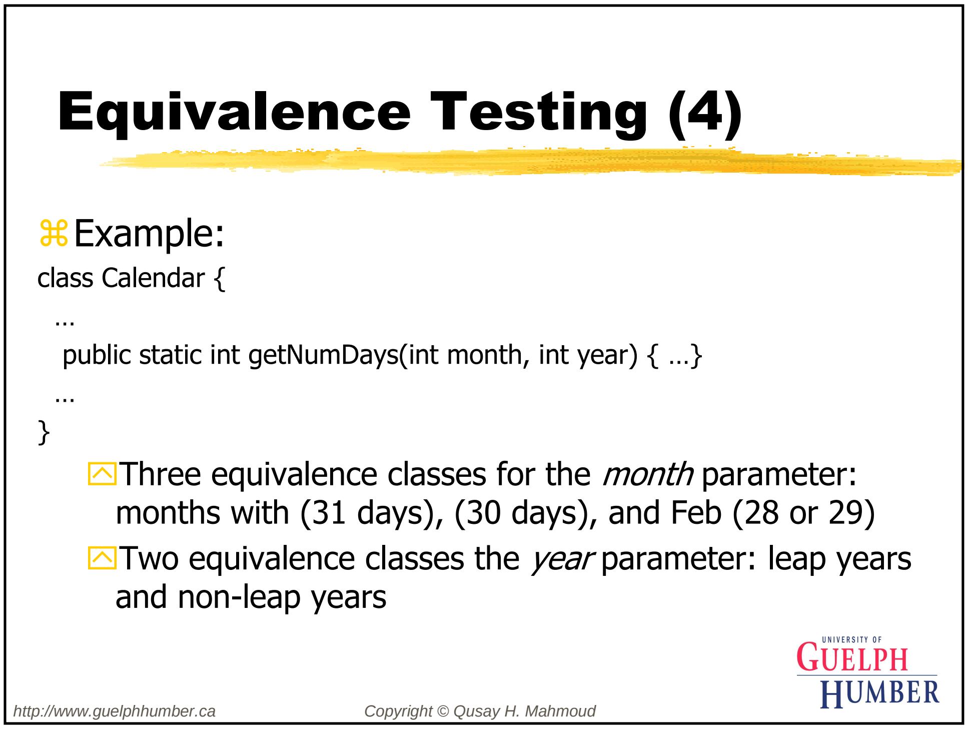


Equivalence Testing (2)

- Criteria for determining equivalence classes
  - **Coverage**: every possible input belongs to one of the equivalence classes
- Selecting equivalence classes (guidelines)
  - Input is valid across range of values. Select test cases from 3 equivalence classes:
    - Below the range
    - Within the range
    - Above the range
  - For each Select two
    - Valid value
    - Invalid value

http://www.guelphhumber.ca

UNIVERSITY OF GUELPH HUMBER

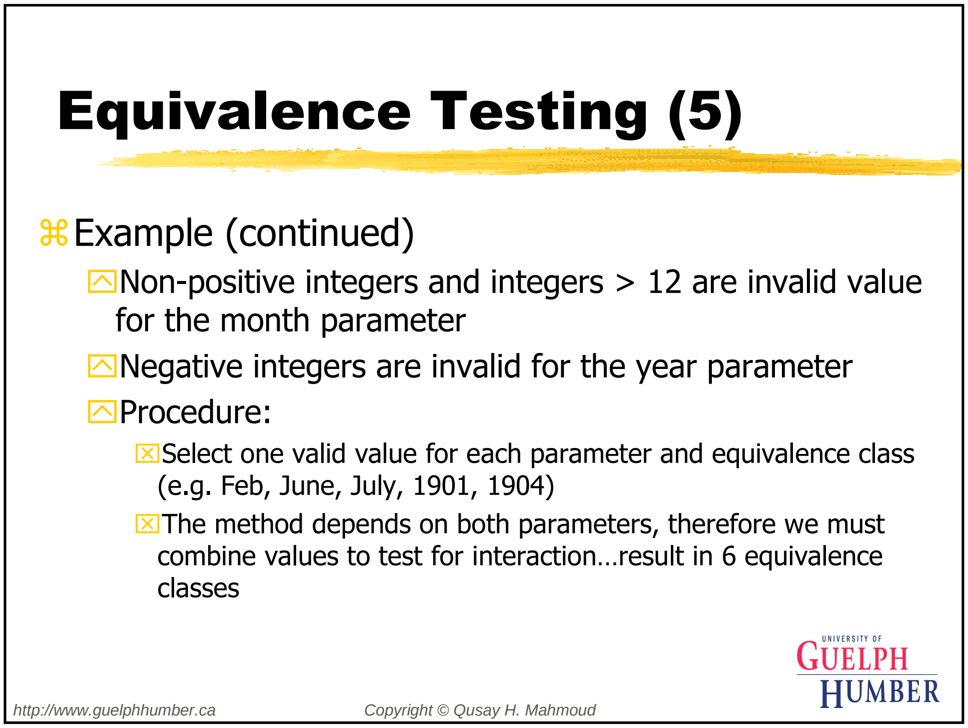


Equivalence Testing (4)

- Example:

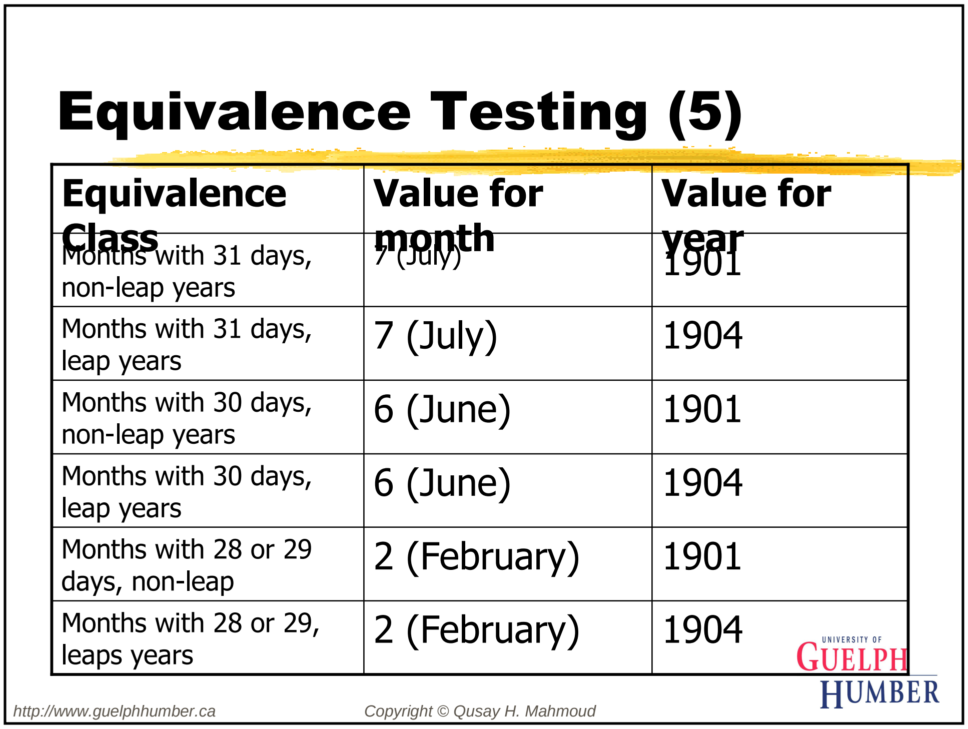
```
class Calendar {
  …
   public static int getNumDays(int month, int year) { …}
  …
}
```

  - Three equivalence classes for the *month* parameter: months with (31 days), (30 days), and Feb (28 or 29)
  - Two equivalence classes the *year* parameter: leap years and non-leap years



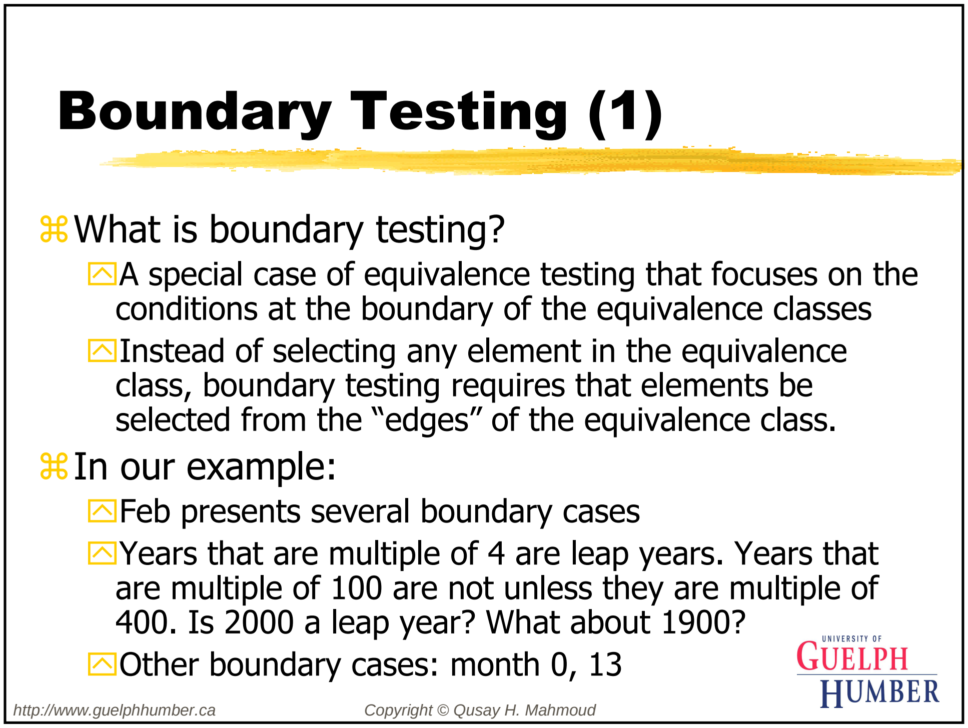
Equivalence Testing (5)

- Example (continued)
  - Non-positive integers and integers > 12 are invalid value for the month parameter
  - Negative integers are invalid for the year parameter
  - Procedure:
    - Select one valid value for each parameter and equivalence class (e.g. Feb, June, July, 1901, 1904)
    - The method depends on both parameters, therefore we must combine values to test for interaction…result in 6 equivalence classes



Equivalence Testing (5)

Equivalence Class	Value for month	Value for year
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap years	6 (June)	1904
Months with 28 or 29 days, non-leap	2 (February)	1901
Months with 28 or 29, leaps years	2 (February)	1904



Boundary Testing (1)

- What is boundary testing?
  - A special case of equivalence testing that focuses on the conditions at the boundary of the equivalence classes
  - Instead of selecting any element in the equivalence class, boundary testing requires that elements be selected from the "edges" of the equivalence class.
- In our example:
  - Feb presents several boundary cases
  - Years that are multiple of 4 are leap years. Years that are multiple of 100 are not unless they are multiple of 400. Is 2000 a leap year? What about 1900?
  - Other boundary cases: month 0, 13

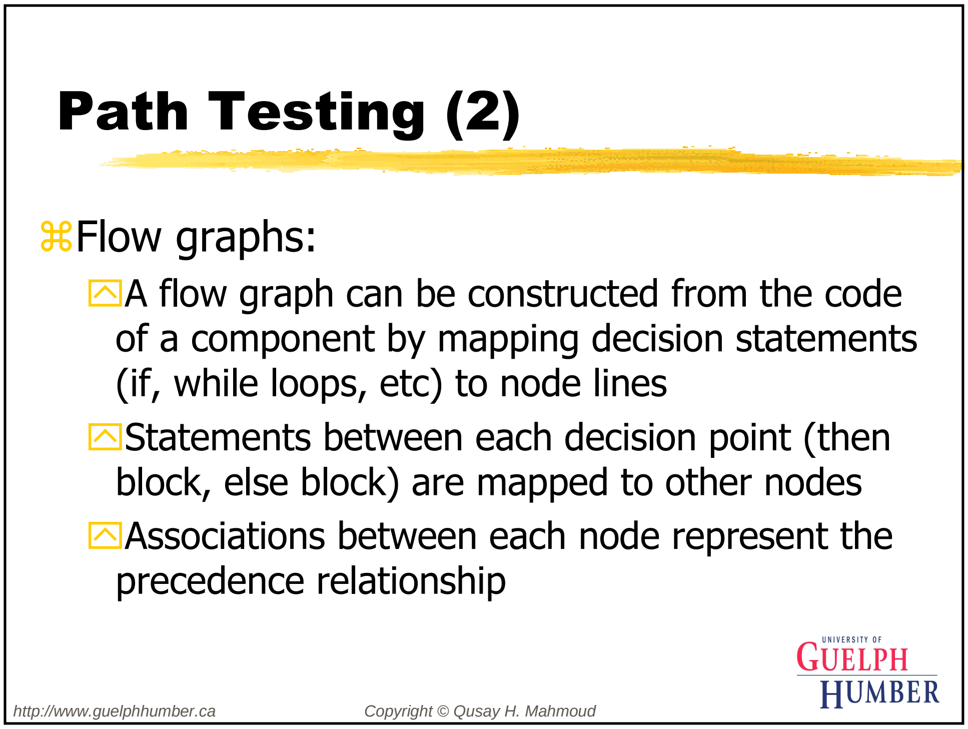
Boundary Testing (2)

- Additional boundary cases:

Equivalence class	Value of month	Value of year
Leap yrs divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

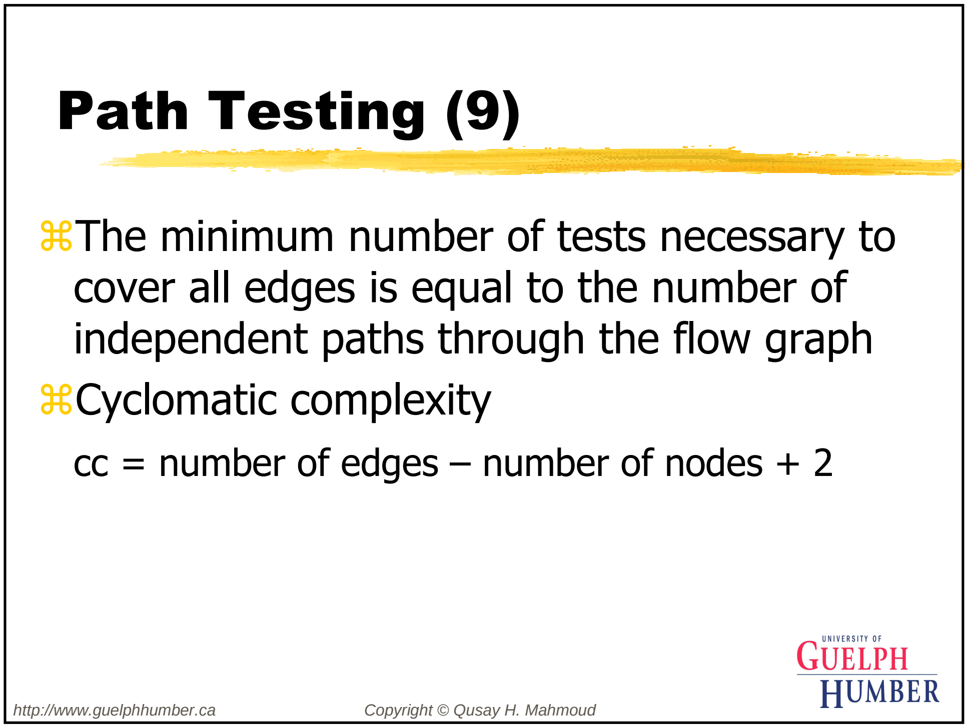
Path Testing (1)

- A whitebox testing technique that identifies faults in the implementation of a component
- Assumption: exercising all possible paths through the code at least once, most faults will trigger failures
  - Identification of paths require knowledge of source code and data structures
- Starting point: flow graph
  - Consists of nodes representing executable blocks and associations representing flow of control
  - A block is a number of statements between two decisions



Path Testing (2)

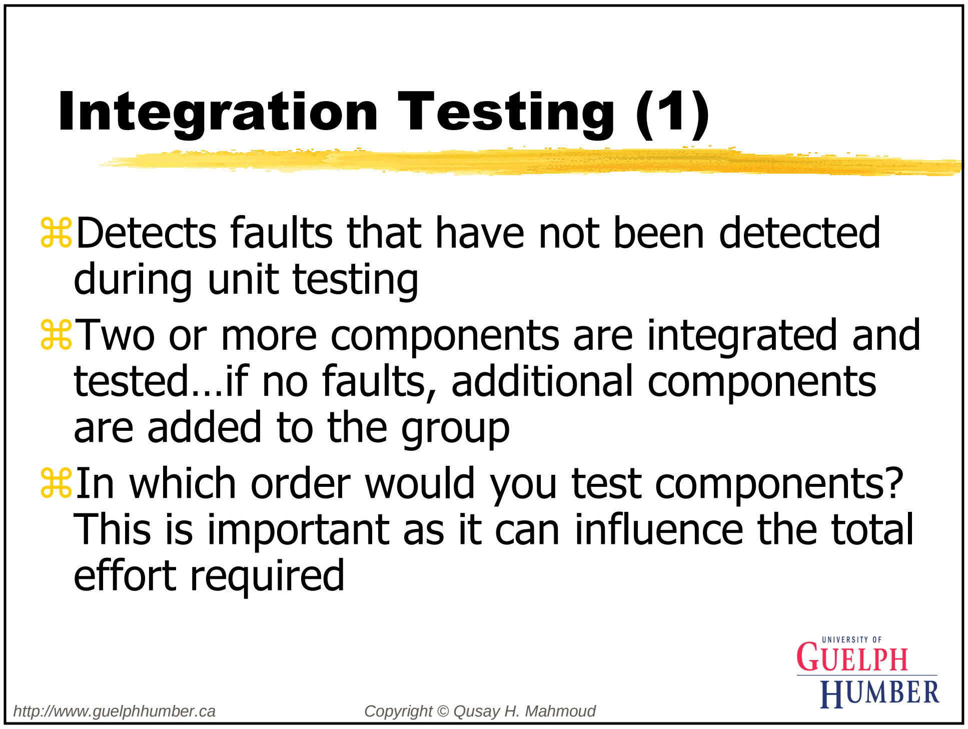
- Flow graphs:
  - A flow graph can be constructed from the code of a component by mapping decision statements (if, while loops, etc) to node lines
  - Statements between each decision point (then block, else block) are mapped to other nodes
  - Associations between each node represent the precedence relationship



Path Testing (9)

- The minimum number of tests necessary to cover all edges is equal to the number of independent paths through the flow graph
- Cyclomatic complexity

  cc = number of edges – number of nodes + 2



Integration Testing (1)

- Detects faults that have not been detected during unit testing
- Two or more components are integrated and tested…if no faults, additional components are added to the group
- In which order would you test components? This is important as it can influence the total effort required

http://www.guelphhumber.ca