Software Testing Part 1

Thareendhra Keerthi Wijayasiriwardhane
thareen@kln.ac.lk

DEPARTMENT OF INDUSTRIAL MANAGEMENT
UNIVERSITY OF KELANIYA

Introduction

- Goal of Software Engineering
  - To produce high quality software
  - But can not be achieved simply
    - Possibility for injection of human errors
    - Inability to perform and communicate perfectly
- Software development is accompanied by a Quality Assurance (QA) process

Software Validation & Verification

- Covers many of the activities come under Software QA
- A whole life-cycle process
  - Starts with requirement reviews
  - Continues through design reviews and code inspections
  - To software testing

Validation & Verification (Cont.,)

- Software Validation
  - Refers whether the right product is built [Boehm, 1979]
  - Ensures that the software meets its expectations
- Software Verification
  - Refers whether the product is built right [Boehm, 1979]
  - Checks that the software confirms to its specification
- Techniques used
  - Software Inspection
  - Software Testing

[Boehm, 1979] Boehm, B. W., Software Engineering: R & D Trends and Defence Needs, in *Research Directions in Software Technology*, Wegner, P., Eds. Cambridge, Massachusetts: MIT Press, 1979

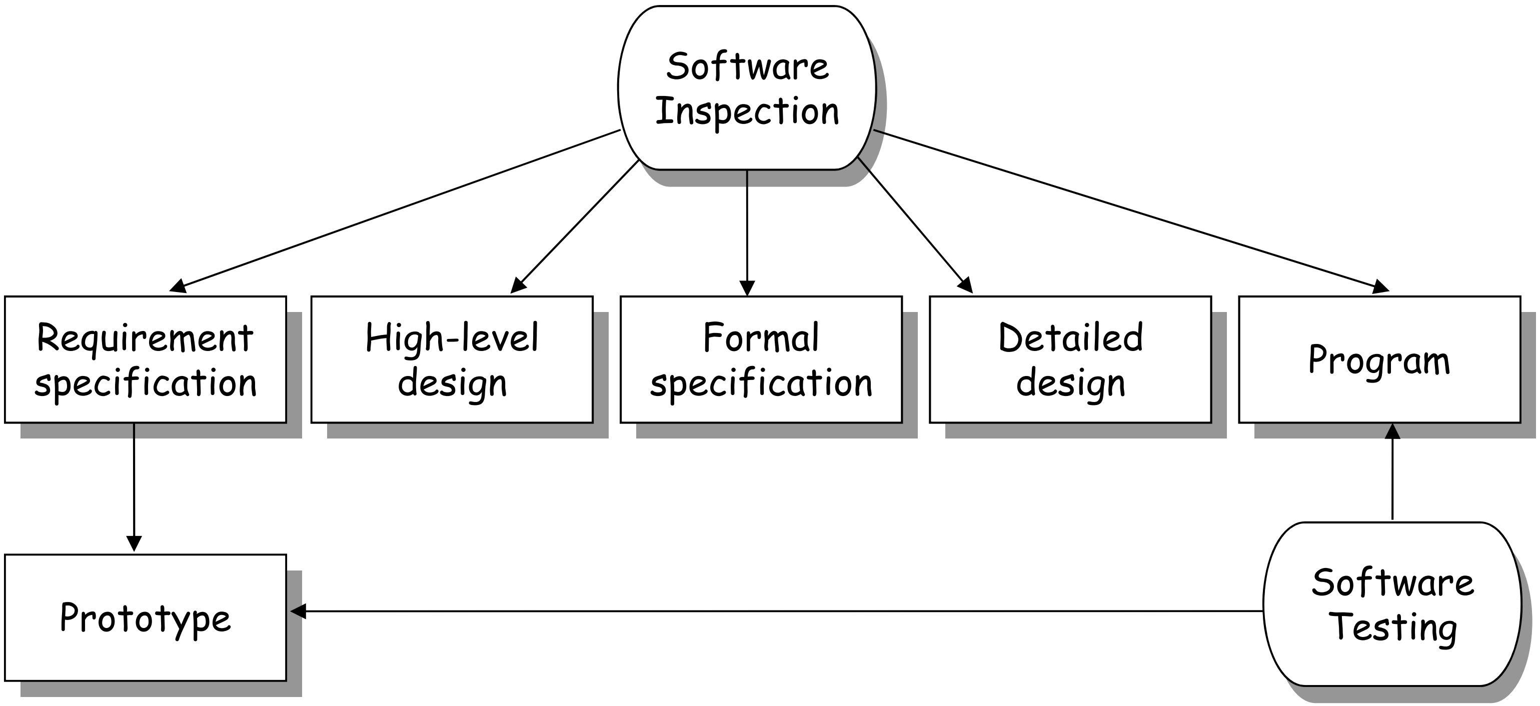
Software Inspection

- Applied to all stages
- Reviews the system's representations
- A static technique
  - Can only check the correspondence between a program and its specification
  - Can not demonstrate that the software
    - Operationally useful
    - Satisfies non-functional requirements

Software Testing

- The predominant V & V technique
- A dynamic technique
  - Involves exercising an executable representation of the software with a set of test data and
  - Examining its output and operational behavior to check whether the software performs as intended
- Usually carried out during the implementation and after the implementation is complete

Software Inspection & Testing



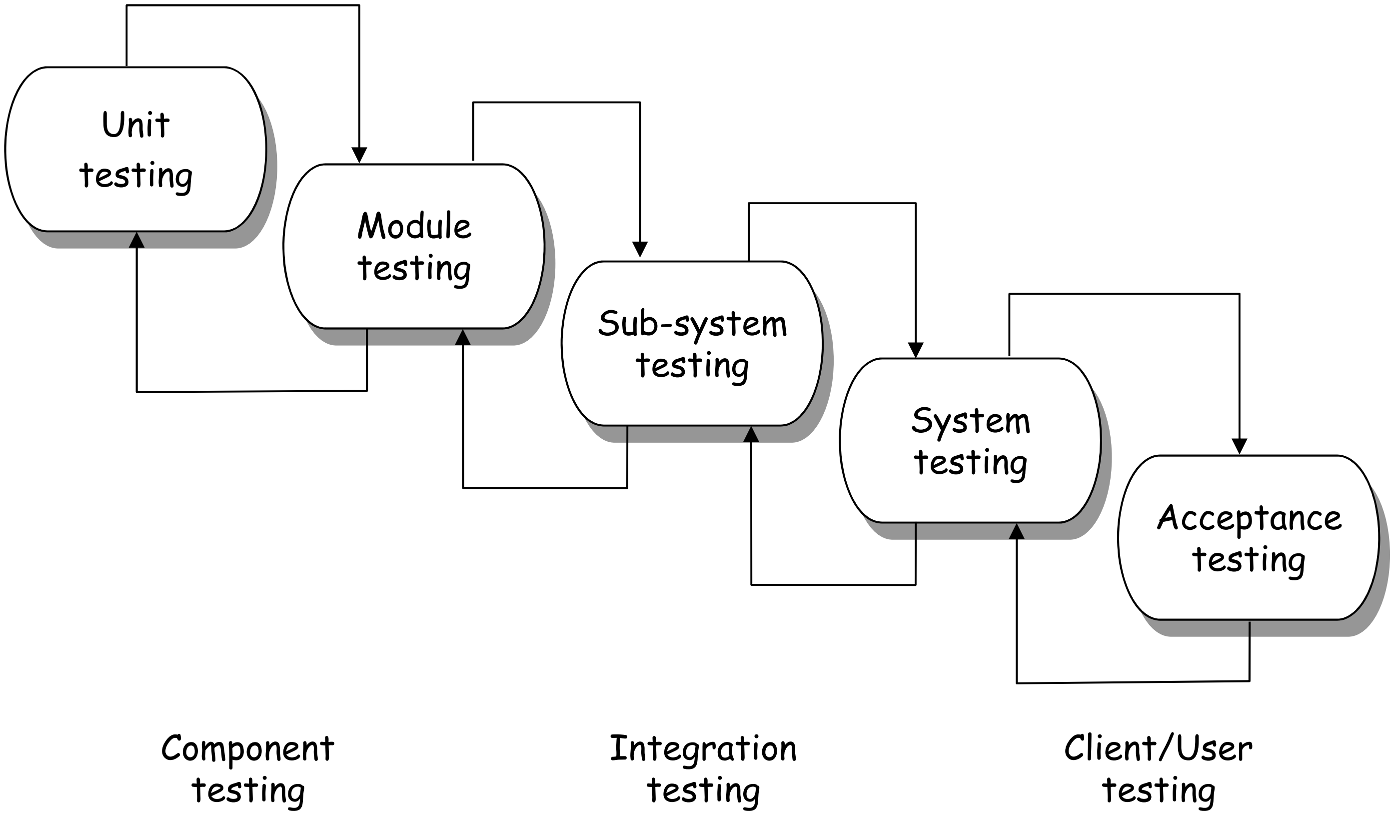
Software inspection & testing in software development process [Sommerville, 2001]

[Sommerville, 2001] Sommerville, I., *Software Engineering*, 6 ed., Addison-Wesley, 2001

Software Testing Process

- Software development
  - System is built out of sub-systems
  - Sub-systems are built with modules
  - Modules are assembled with individual program units
- Software testing
  - Starts with testing of individual program units
  - Continues with testing of the integration of these units
  - Ends with testing of the system's functionality as a whole

Stages in Software Testing



Stages in software testing process

Software Testing Cost

- Accounts not less than 50% of software development cost [Harrold, 2000] & 70-80% of development time [Ko, 2005]
- In extreme, testing of critical software can cost as much as all other software engineering costs combined
  - Software used in medical, aviation, nuclear and military applications
  - Failure can result in lose of human life

[Harrold, 2000] Harrold, M. J., Testing: A Roadmap, In *Proceedings of the 22nd International Conference on Software Engineering*, Ireland, 2000

[Ko, 2005] Ko, A. J. and Myers, B. A., A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *Journal of Visual Languages and Computing*, vol. 16, no. 1-2, 2005.

Software Testing Cost (Cont.,)

- Can be minimized
  - By controlling the causes that lead to errors [McCabe, 1982]
  - By increasing the testability of programs [Voas, 1995]
- Main cause that lead to errors and also for the decrease of testability
  - Software complexity

[McCabe, 1982] McCabe, T., Structured Testing: *A Software Testing Methodology using the Cyclomatic Complexity Measure*, Special Publication, U. S. Department of Commerce / National Bureau of Standards, December 1982

[Voas & Miller, 1995] Voas, J. and Miller, K., Software Testability: The New Verification, *IEEE Software*, vol. 12, no. 3, May 1995

Software Complexity

- Strongly correlates to
  - Error proneness of a program [Watson, 1996]
  - Testability of the software
  - Reliability of the system
- Beyond a certain level of complexity, the likelihood that a program contains errors, increases very exponentially [McCabe, 1982]
- Solution
  - Control the complexity of programs

[Watson, 1996] Watson, A. and McCabe, T., Structured Testing: *A Testing Methodology Using the Cyclomatic Complexity Metric*, Special Publication, National Institute of Standards and Technology, United State, August 1996

Controlling Software Complexity

- Using structured programming
  - Makes programs more simple
  - But does not solve the problem
    - Can write structured but complex programs
- Coding program units as simple as possible
  - Limit the complexity when coding
    - Measure complexity quantitatively

Software Complexity Measures

- Number of LOC
  - Increases with complexity
  - Depends on programming language
  - Depends on coding style and formatting used
- Cyclomatic Complexity [McCabe, 1982]
  - Measures the amount of decision logic in a program
  - Completely independent of coding style and formatting used
  - Nearly independent of programming language used
  - Entirely based on the control flow representation of a program [Pressman, 2005]

[Pressman, 2005] Pressman, R., *Software Engineering, A Practitioner's Approach*, 6th ed. McGraw Hill, 2005

Control Flow Representation

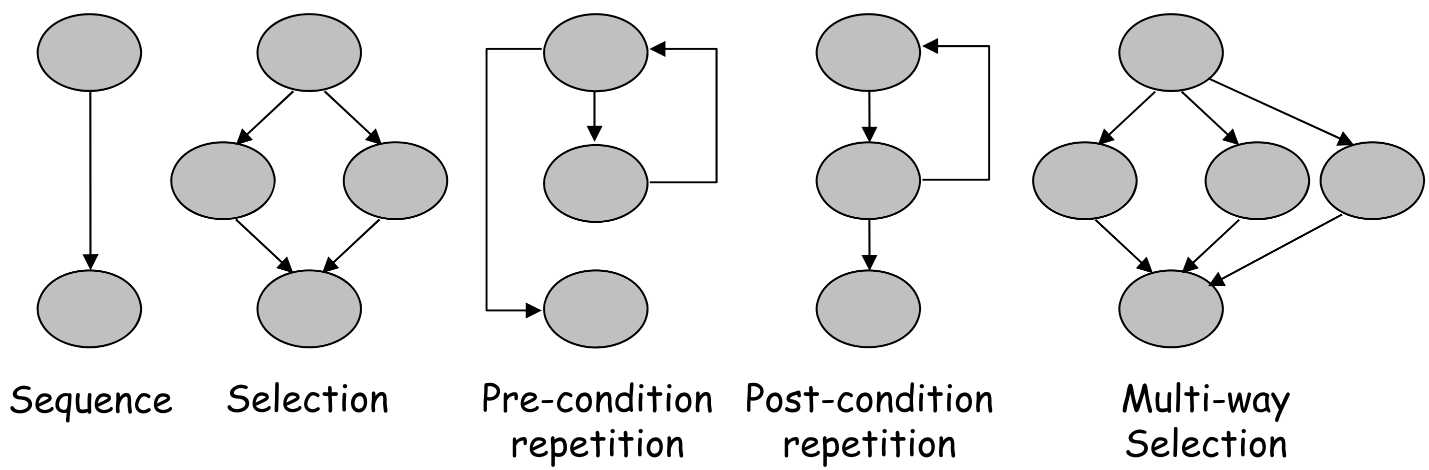
- A skeletal model of all execution paths through a program unit [Sommerville, 2001]
- Also referred to as Control Flow Graph
- Consists of
  - Nodes
    Represent computational statements or expressions in a program unit
  - Edges
    Represent transfer of control between nodes

Control Flow Graphs

- Program unit
  - A software module with a single entry and exit point and used as an individual component via a call-return mechanism
- In a flow graph
  - Source node represents entry point of the program unit
  - Sink node represents exit point of the program unit

Control Flow Graphs (Cont.,)

- How to construct a flow graph?
  - Replace structured programming constructs in a program with equivalent flow graph notations

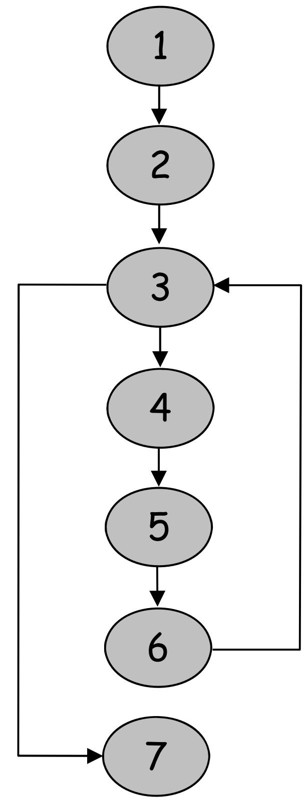


Structured programming constructs in flow graph form

Control Flow Graphs (Cont..)

```
1  int GCD(int m, int n)
   {
2      int r;
3      while(n != 0)
       {
4        r = n;
5        n = m % n;
6        m = r;
       }
7      return m;
   }
```

Source code of "GCD" function



Control flow graph of "GCD" function

Computing Cyclomatic Complexity

- By definition [McCabe, 1982]

  $$v(G) = e - n + 2$$

- Indicates an quantitative measure of the program's complexity
- McCabe has suggested $v(G) = 10$ as a practical upper limit

Cyclomatic Complexity

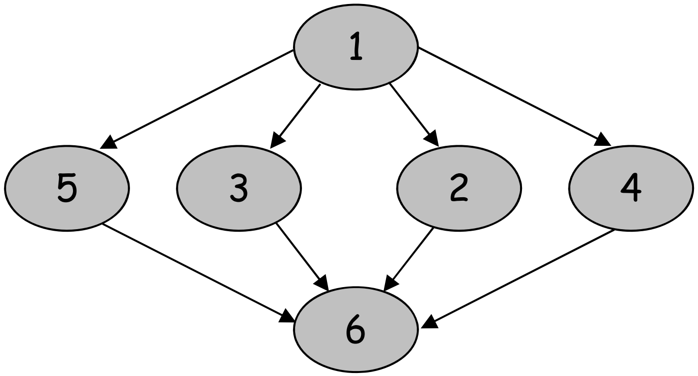
- If a program exceeds the upper limit for v(G)
  - It becomes extremely difficult to understand and thereby it would
    - Prone for more errors
    - Be very hard to test
    - Not be reliable
- Solution
  - Limit v(G) of the program unit
  - Distribute functionality among several program units instead of one unit

Computation of v(G)

- By definition
  - Requires the control flow representation of the program
  - Not convenient
  - Suitable for automated computation
- Predicate nodes or decision constructs based computation
  - For binary selection and repetition constructs
    - $v(G) = 1 + p$
  - For multi-way selection constructs
    - v(G) = 1 + (no. of outgoing flows - 1)
    - Also valid for binary selections and repetitions

Computation of v(G)

```
1  switch(n)
2     case 0: n = -1;
3     case 1: n = n + 1;
4     case 2: n = n – 2;
5     default: n = 0;
6  end switch
```

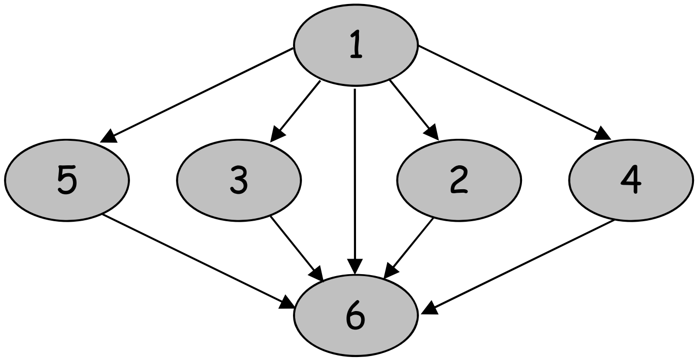



```
v(G) = 1 + (no. of outgoing edges – 1)
     = 1 + (4 – 1)
     = 1 + 3
     = 4
```

Multi-way selection construct with an explicit default branch

Computation of v(G)

```
1  switch(n)
2     case 0: n = -1;
3     case 1: n = n + 1;
4     case 2: n = n – 2;
5     case 3: n = 0;
6  end switch
```




```
v(G) = 1 + (no. of outgoing edges – 1)
     = 1 + (5 – 1)
     = 1 + 4
     = 5
```

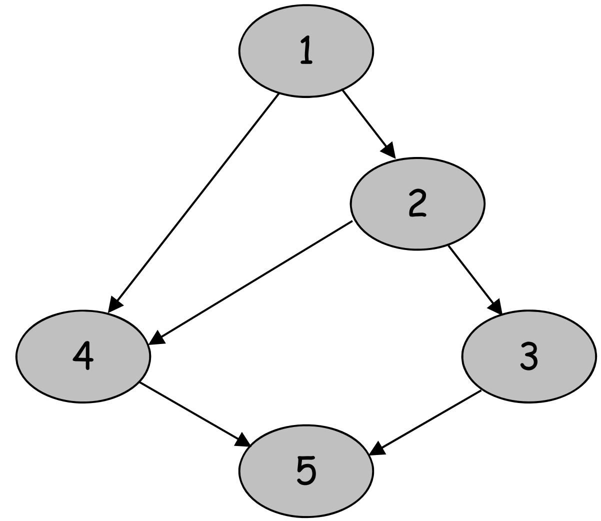
Multi-way selection construct with an implicit default branch

Computation of v(G)

- For complex selection or repetition expressions composed with Boolean operators
  - Use v(G) = 1 + p and then
  - Add an additional 1 to v(G) for each Boolean operator in use if it is a short-circuit operator
  - Add nothing additionally to v(G) if the Boolean operator is a full-evaluation operator

Computation of v(G)

```
1, 2   if (exp1 && exp2)
3          // do something
       else
4          // do something else
5      end if
```

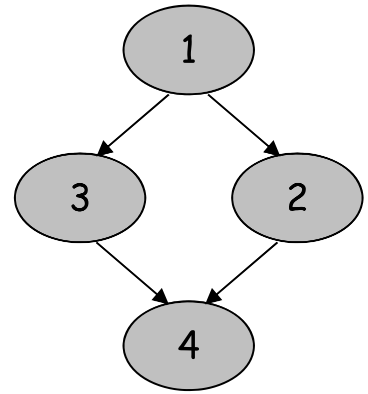



```
v(G) = 1 + p + (1 for each Boolean operator in use)
     = 1 + 1 + 1
     = 3
```

Contribution to v(G) from short-circuit Boolean operators

Computation of v(G)

```
1    if (exp1 & exp2)
2        // do something
     else
3       // do something else
4    end if
```




```
v(G) = 1 + p
     = 1 + 1
     = 2
```

Contribution to v(G) from full-evaluation Boolean operators