AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

SOFTWARE ENGINEERING PRINCIPLES
SOFTWARE VERIFICATION

STEFAN HALLESTEDE
PETER GORM LARSEN
CARL SCHULTZ

UNI VERSITET

Contents

Contents

Which problems are we trying to solve?

- Are we building the right product?

  Validation

  Does the product satisfy the customer?

- Are we building the product right?

  Verification

  Does the product satisfy the specification?

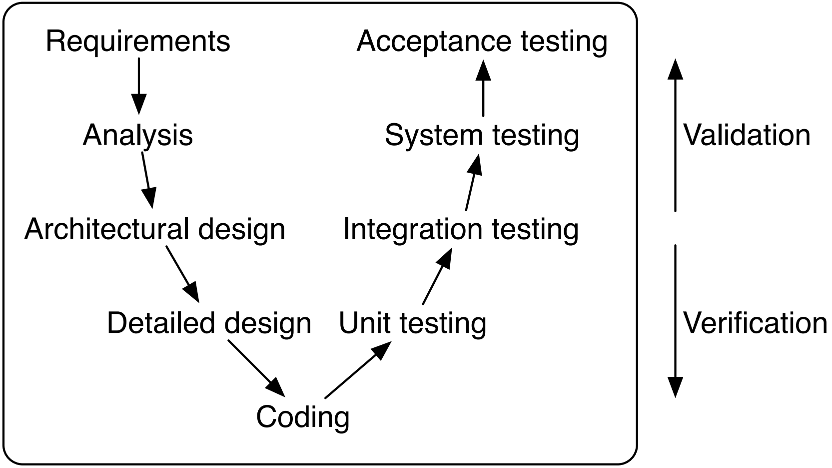
Verification And Validation Testing Overview

Activity	**Verification**	**Validation**
Inspection or review		
Unit testing		
Integration testing		
System testing		
Acceptance testing		
Regression testing		

Verification And Validation Testing Overview

Activity	Verification	Validation
Inspection or review	✓	✓
Unit testing	✓	
Integration testing	✓	
System testing		✓
Acceptance testing		✓
Regression testing	✓	✓

Verification And Validation Testing In The V-Model



What Are The Objectives Of Testing?

- Find errors?
- Check for compliance with requirements?
- Break the software?
- Reduce risk? (Which risk?)
- Check performance?
- Show absence of defects?

What Are The Objectives Of Testing?

- Find errors?
- Check for compliance with requirements?
- ~~Break the software?~~
- Reduce risk? (Which risk?)
- Check performance?
- Show absence of defects?

Program Complexity Causes Problems

- Estimate time to test addition of two 32 bit integers thoroughly!

Program Complexity Causes Problems

- Estimate time to test addition of two 32 bit integers thoroughly!
- 2^{64} values to inspect
- How many seconds, minutes, years, centuries, millennia?

Program Complexity Causes Problems

- Estimate time to test addition of two 32 bit integers thoroughly!
- 2^{64} values to inspect
- How many seconds, minutes, years, centuries, millennia?
- A reasonable estimate would be approx. 600,000 millennia

  if we run 1000 tests per second

What Are The Objectives Of Testing?

- Find errors?
- Check for compliance with requirements?
- ~~Break the software?~~
- Reduce risk? (Which risk?)
- Check performance?
- Show absence of defects?

What Are The Objectives Of Testing?

- Find errors?
- Check for compliance with requirements?
- ~~Break the software?~~
- Reduce risk? (Which risk?)
- Check performance?
- ~~Show absence of defects?~~

"Program testing can be used to show the presence of defects, but never their absence"
Edsger W. Dijkstra

What Are The Objectives Of Testing?

- Find errors?
- Check for compliance with requirements?
- ~~Break the software?~~
- Reduce risk? (Which risk?)
- Check performance?
- ~~Show absence of defects?~~

"Program testing can be used to show the presence of defects, but never their absence"
Edsger W. Dijkstra

- Formal methods can complement testing to verify absence of certain defects

Ad-Hoc Testing

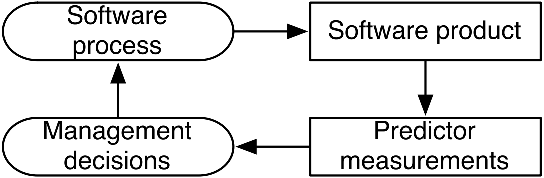
Belongs to the **Code & Fix** development “method”:

- You have implemented something
- You run it
- You look whether you are satisfied with the result
- If not, you fix the program

We Need To Be Systematic About Testing

- Analogy: *Scientific experiment*
  - To *find out* whether some process works
  - Need to state the expected result *before* the experiment
  - Must know the *precise conditions* under which the experiment runs
  - The experiment must be *repeatable*
- *What do we expect of good testing?*

Software Measurement And Metrics



Aims

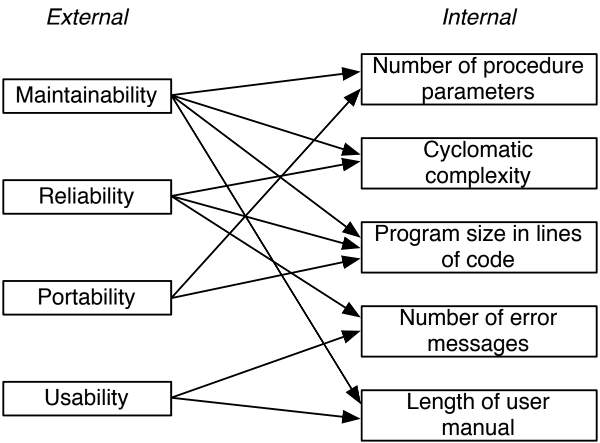
- Make general *predictions* about a system
- Identify *anomalous* components

Classes of metrics

- *dynamic*: measured during program execution
  (e.g. execution times of specific functions)
- *static*: measured by means of the software artefacts
  (e.g. program size in lines of code)

External And Internal Software Attributes

External Attributes Difficult To Measure: Use Relationships To Internal Attributes



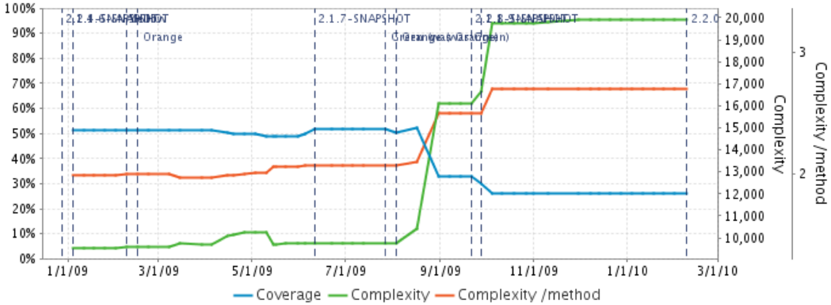
- Internal attributes must be measured *accurately*
- A well-understood *relationship* between the attributes must exist

Static Software Product Metrics

- **Fan-in** *number of functions calling a function*
  A high value suggests tight coupling to the rest of the design
- **Fan-out** *number of functions called by a function*
  A high value suggests overall complexity of the calling function
- **Length of code** *size of the program*
  Length reliably predicts error-proneness in components
- **Cyclomatic comlexity** *control complexity of the program*
  Affects program understandability and test complexity
- **Length of identifiers** *average length of distinct identifiers*
  Longer identifiers likely to be meaningful and understandable
- **Depth of condition nesting** *nesting of if-statements*
  Deeply nested they are hard to understand and error-prone
- **Fog index** *average length of words in documents*
  A high value suggests the document is difficult to understand

Continual Measurements[1]

Analysing Measurements

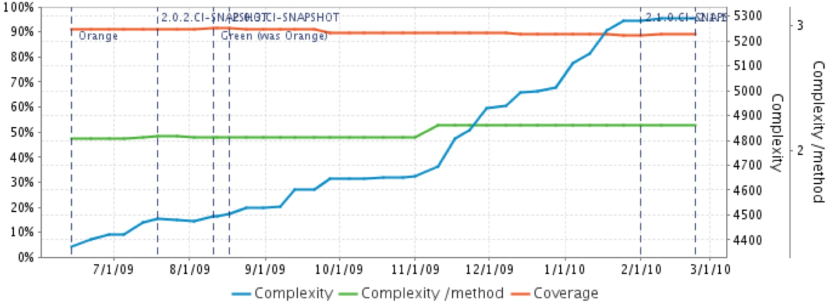


Measurement analysis is open to interpretation
but measurements provide a basis on which to judge quality

[1] Neil Ford (2009) *Evolutionary architecture and emergent design*. IBM developerWorks. Slides.

Continual Measurements[1]

Analysing Measurements



Measurement analysis is open to interpretation
but measurements provide a basis on which to judge quality

[1] Neil Ford (2009) *Evolutionary architecture and emergent design*. IBM developerWorks. Slides.

Contents

Planning Software Tests

- Devise a test plan:
  - **Testing process:** describe major testing phases
  - **Requirements traceability:** ensure test of requirements
  - **Test items:** specify artefacts to be tested
  - **Testing schedule:** integrate into development schedule
  - **Test recording procedures:** how test results are archived
- Keep in mind the *Pareto principle*:

80% of the errors in 20% of the components

What Should A Test Measure?

Achieve an acceptable **level of confidence**
that the system **behaves correctly**
under all **circumstances of interest.**

What

- is a level of confidence?
- is correct behavior?
- are circumstances of interest?

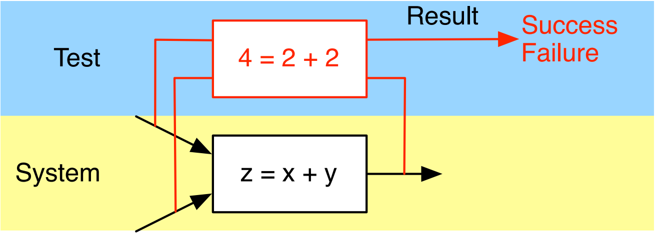
Level Of Confidence

- Usually specified as "*residual defect discovery rate*"
  - Number of defects found in a *given (series of) tests*, or
  - Number of defects found in a *given time*
  - "*Less than 10 defects discovered in last 7 days*"

- An alternative
  - *Reliability specification*
  - "*Mean time between failures is at least 5000 hours*"

Correct Behavior

- *Specification* required
- Derived from user *requirements*
- *Component specifications*
- Compare test result with *expected result*
- A test can either *succeed* or *fail*



Circumstances Of Interest

- *Realistic inputs* *(How can this be judged?)*
- University has 10,000 students growing a little each year:
- Test student registration system with
  - 10,000 students
  - 12,000 students
  - 15,000 students
- *Never* going to be 100,000 students

Circumstances Of Interest

- *Realistic inputs* *(How can this be judged?)*
- University has 10,000 students growing a little each year:
- Test student registration system with
  - 10,000 students
  - 12,000 students
  - 15,000 students
- *Never* going to be 100,000 students
- Part of the *requirements!*

Remember The Design Principle “Ensure Testability”

What is “Testability”?

Remember The Design Principle "Ensure Testability"

What is "Testability"?

?

Some keywords:

- **Operability:**
- **Controllability:**
- **Simplicity:**
- **Understandability:**

Remember The Design Principle “Ensure Testability”

What is “Testability”?

Some keywords:

- **Operability:** the program “works”
- **Controllability:** the program is easily predicable
- **Simplicity:** the program’s function, structure, and code
- **Understandability:** the program is documented

Contents

What Is A Test Case?

“A set of test inputs,
execution conditions,
and expected results developed for a particular objective,
such as to exercise a particular program path
or to verify compliance with a specific requirement”

(IEEE standard)

Simple Example Of A Test Case

- **Title:** “Open account without error”
- **Input:** Customer data
- **Conditions:** Enough storage space for account record
- **Execution:** Fill in mask 23.7.10 … and confirm
- **Expected results:**
  New account number is generated and shown within mask 23.7.10

Simple Example Of A Unit Test Case

- **Title:** "Length of empty linked list"
- **Input:**
- **Conditions:**
- **Execution:**
- **Expected results:**

}

Simple Example Of A Unit Test Case

- **Title:** "Length of empty linked list"
- **Input:** Empty linked list
- **Conditions:** None
- **Execution:** Call method `size()`
- **Expected results:** returned value is `0`

The unit test case in Java (JUnit):

```
/* method size() must return 0 for the empty linked list */
void testEmptyLinkedListSize()                 // descriptive name
throws Exception {
  List<Object> x = new LinkedList<Object>(); // input: empty list
  int s = x.size();                            // execution
  assertEquals("length not 0", 0, s);         // comparison to expected result
}
```

Testing More Complex Situations

- Test cases can be collected into *test scenarios*
- Scenarios show a *typical use* of the system
- Each test case stays *simple* but scenario can be complex
- Possible *sources* for scenarios: Use Cases, Interactions

No.	Input	Initial State	Output	Final State
1	User enters valid card	Idle	ATM reads it successfully, displays “Enter PIN” prompt	Await PIN
2	User enters valid PIN	Await PIN	ATM displays *Transaction Section Screen*	Get TxNo
3	User selects withdrawal for £ 50	Get TxNo	ATM checks balance, debits account, dispenses cash	Take cash
4	User takes card and money	Take cash	ATM displays “Enter your card”	Idle

Contents

Testing Methods

- *White box* testing
  - Follows *control structure* of procedural design
  - Exercises *program*: conditions, loops, data structures
- *Black box* testing
  - *Ignores* implementation details
  - Exercises all *functional requirements*

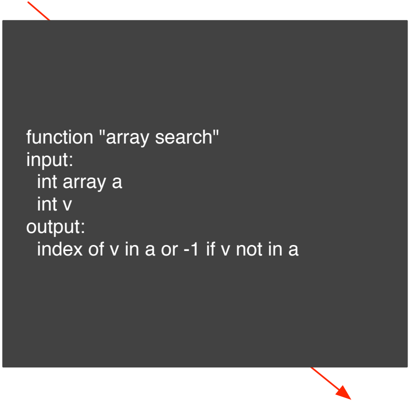
White Box Testing

- Examines *paths and decisions* by looking inside the program
- Typically used in *early* testing stages
  - Unit tests
  - Integration tests

```
int binsearch (int[] a, int v) {
        int low = 0;
        int high = length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (a[mid] > value)
                high = mid - 1;
            else if (a[mid] < value)
                low = mid + 1;
            else
                return mid;
        }
        return - 1;
    }
```

Block Box Testing

- Examines all *functions* and compares
  actual to expected result
- Typically used in *later* testing stages
  - System tests
  - Acceptance tests



Contents

Whitebox Testing

- White box testing relies *internal structure* of programs
  - One such technique: *basis paths testing*
- *Basis paths* can be used to derive test cases
- *Cyclometric complexity*:
  - Metric for *complexity of software*
  - Upper bound on *number of test cases*

Basis Path Testing

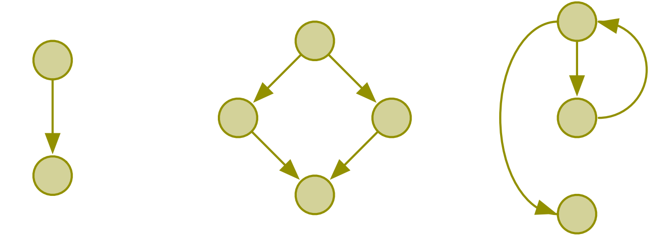
- A path is

  "a sequence of instructions that may be performed in the execution of a computer program"
  (IEEE definition)

- Path testing:
  - *Selecting different paths* through the code,
  - and checking that they *work correctly*

Representing Control Flow By Flow Graphs

Graphical notation that helps visualising control flow:



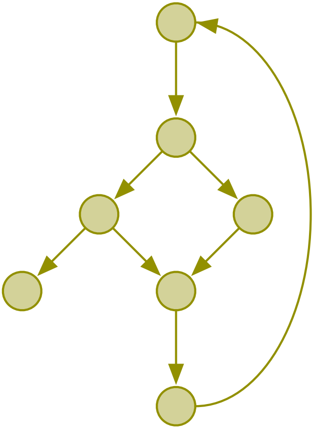
Sequence

If

While

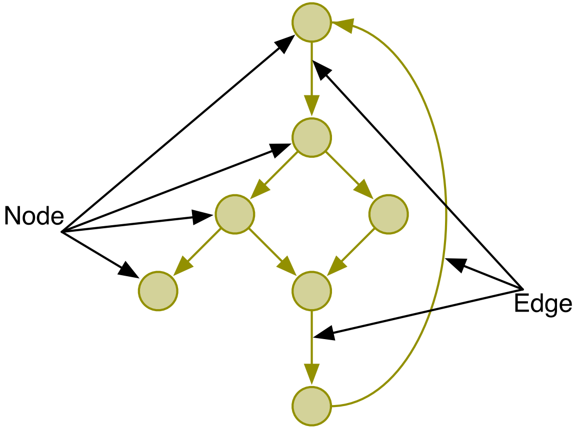
Properties Of A Flow Graph

A Flow Graph



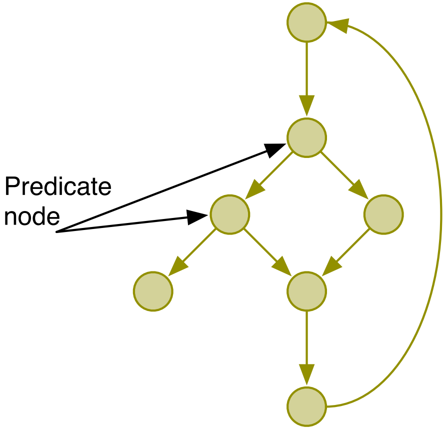
Properties Of A Flow Graph

Nodes And Edges



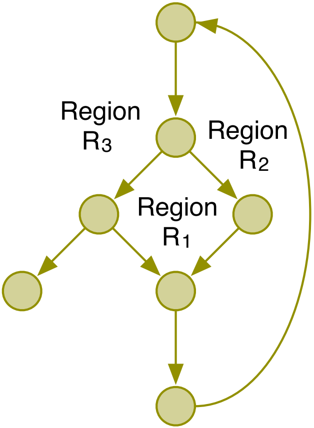
Properties Of A Flow Graph

Predicate Nodes



Properties Of A Flow Graph

Regions

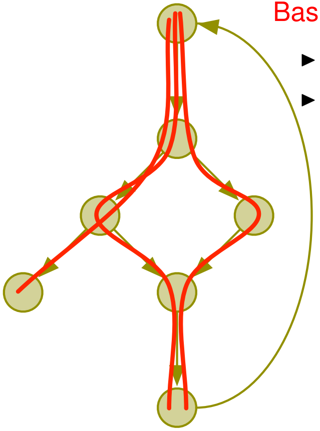


Properties Of A Flow Graph

Basis Set

Independent paths:

- 1–2–3–6–7
- 1–2–5–6–7
- 1–2–3–4



Basis set:

- Each node visited
- Use for test case design

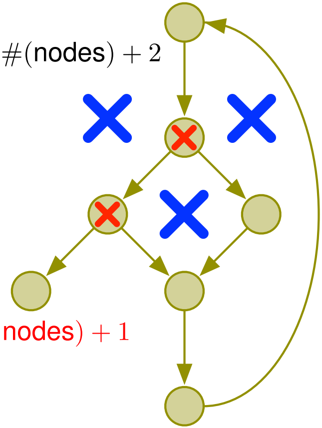
Properties Of A Flow Graph

Cyclomatic Complexity

$$
\begin{aligned}
V(G) &= \#(\text{edges}) - \#(\text{nodes}) + 2 \\
&= 8 - 7 + 2 \\
&= 3
\end{aligned}
$$

$$
\begin{aligned}
V(G) &= \#(\text{regions}) \\
&= 3
\end{aligned}
$$

$$
\begin{aligned}
V(G) &= \#(\text{predicate nodes}) + 1 \\
&= 2 + 1 \\
&= 3
\end{aligned}
$$



For What Is Cyclomatic Complexity Used?

- upper bound on required *number of test cases*
- upper bound on *size of the basis set*
- software metric for *program complexity*

Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length - 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

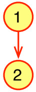
control flow graph?

Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

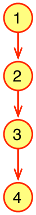


Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

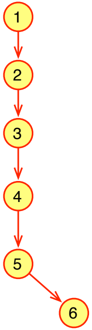


Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

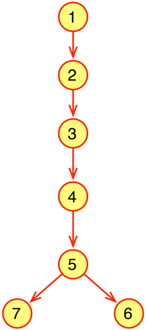
Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



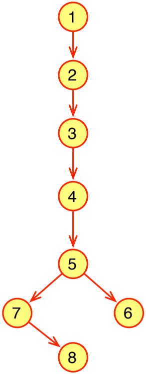
Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



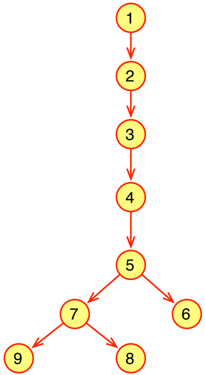
Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



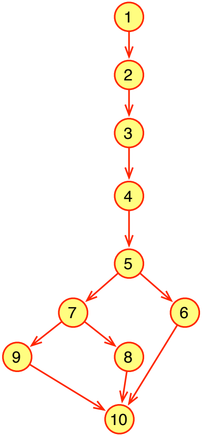
Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



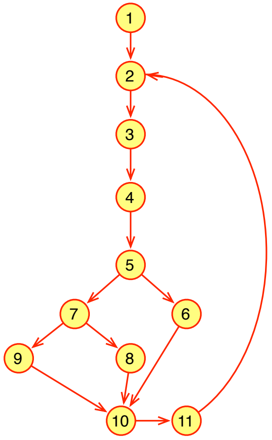
Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



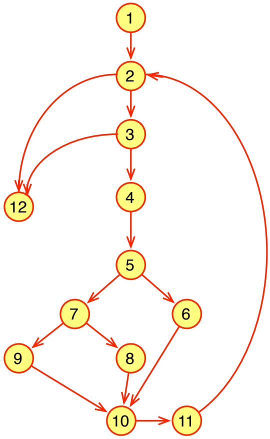
Worked example: Binary search

```
int binsearch (int[] a, int value) {
  int low = 0;
  int high = length { 1;
  int r = -1;
  while (low <= high && r == -1) {
    int mid = (low + high) / 2;
    if (a[mid] > value) {
      high = mid { 1;
    } else if (a[mid] < value) {
      low = mid + 1;
    } else {
        r = mid;
    }
  }
  return r;
}
```



Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```

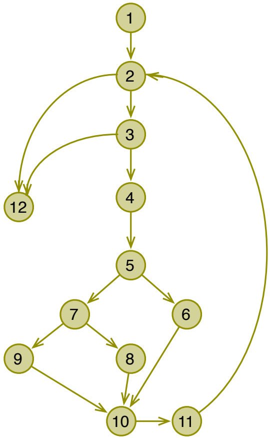


cyclomatic complexity?

Worked example: Binary search

Cyclomatic complexity V(G)

$$
\begin{aligned}
&= \quad 15 - 12 + 2 \\
& \#(\text{edges}) - \#(\text{nodes}) + 2
\end{aligned}
$$



Worked example: Binary search

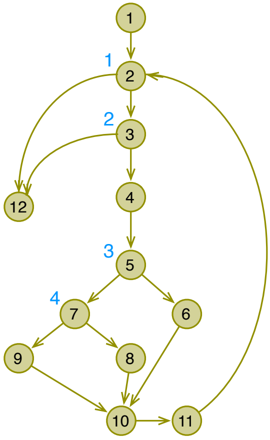
Cyclomatic complexity V(G)

$$= \quad 15 - 12 + 2$$

$$\#(\text{edges}) - \#(\text{nodes}) + 2$$

$$= \quad 4 + 1$$

$$\#(\text{predicate nodes}) + 1$$



Worked example: Binary search

Cyclomatic complexity V(G)

$$= \quad 15 - 12 + 2$$

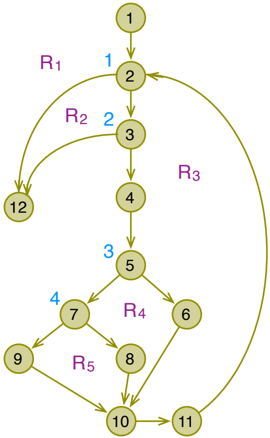
$$\#(\text{edges}) - \#(\text{nodes}) + 2$$

$$= \quad 4 + 1$$

$$\#(\text{predicate nodes}) + 1$$

$$= \quad 5$$

$$\#(\text{regions})$$

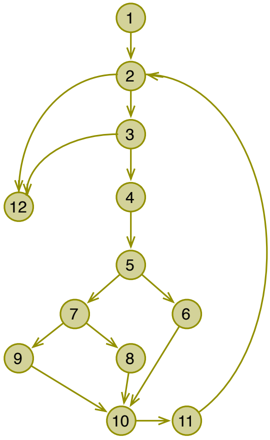


basis paths?

Worked example: Binary search

Basis paths:

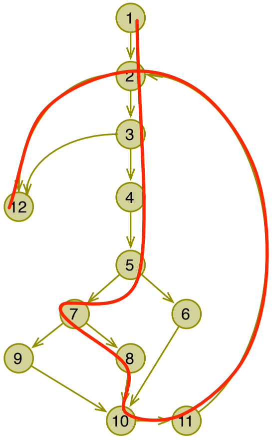
- 1–2–12
- 1–2–3–12
- 1–2–3–4–5–7–9–10–11–2–12
- 1–2–3–4–5–7–8–10–11–2–12
- 1–2–3–4–5–6–10–11–2–12



Worked example: Binary search

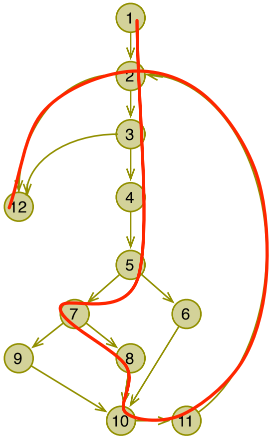
Basis paths:

- 1–2–12
- 1–2–3–12
- 1–2–3–4–5–7–9–10–11–2–12
- 1–2–3–4–5–7–8–10–11–2–12
- 1–2–3–4–5–6–10–11–2–12



Worked example: Binary search

```
int binsearch (int[] a, int value) {
    int low = 0;
    int high = length { 1;
    int r = -1;
    while (low <= high && r == -1) {
        int mid = (low + high) / 2;
        if (a[mid] > value) {
            high = mid { 1;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            r = mid;
        }
    }
    return r;
}
```



test case for this path?

Worked example: Binary search

```
int binsearch (int[] a, int value) {
   int low = 0;
   int high = length { 1;
   int r = -1;
   while (low <= high && r == -1) {
      int mid = (low + high) / 2;
      if (a[mid] > value) {
         high = mid { 1;
      } else if (a[mid] < value) {
         low = mid + 1;
      } else {
         r = mid;
      }
   }
   return r;
}
```

Test case

Path:

1–2–3–4–5–7–8–10–11–2–12

Input:

`a == { 4 }`

`value == 7`

Output:

`r == -1`

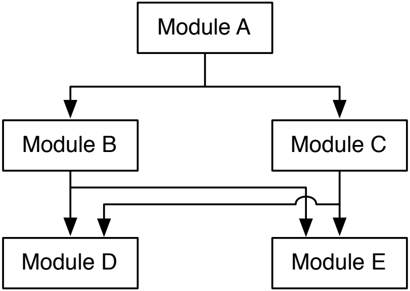
Contents

Unit Testing

- A unit is the *smallest piece* of testable software
  - *One* program module
  - *One* method or *one* class
- Typically the work of *one* programmer
- Unit is based on *detailed design* specification
  - The specification determines the unit tests
- Separate modules
  - *Executed in isolation* from the rest of the system

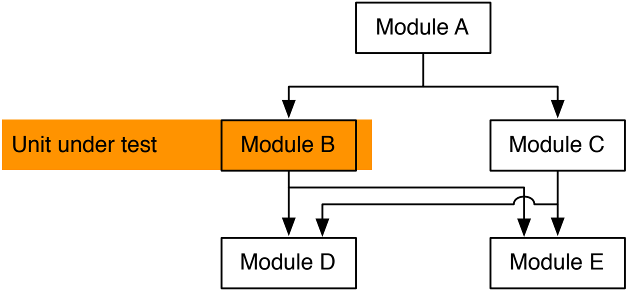
Unit Testing

- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



Unit Testing

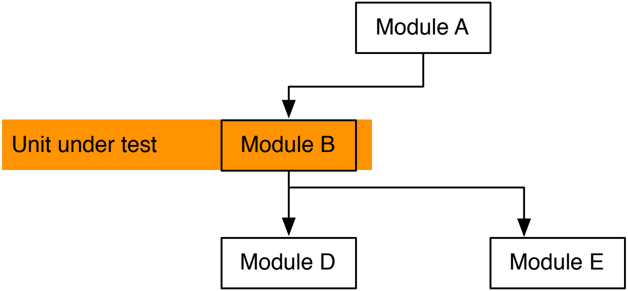
- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



drivers? stubs?

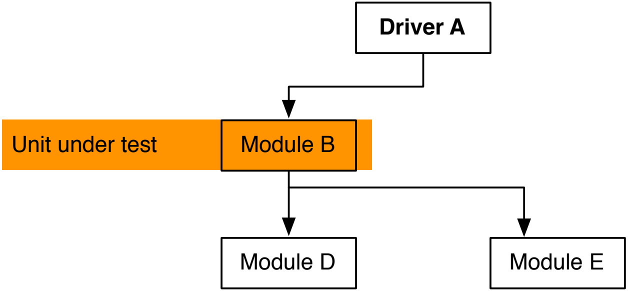
Unit Testing

- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



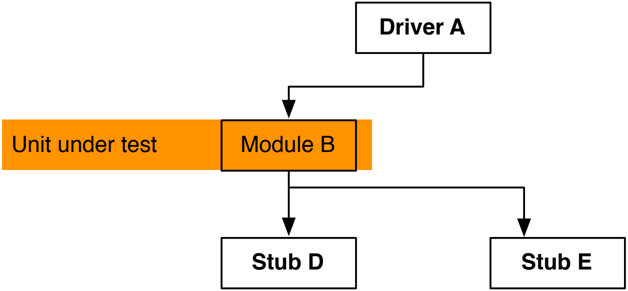
Unit Testing

- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



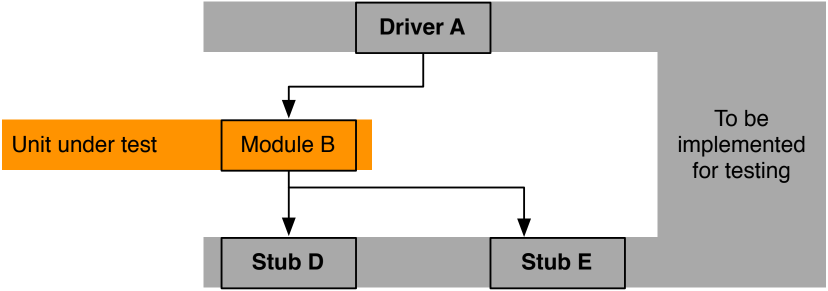
Unit Testing

- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



Unit Testing

- *Drivers* take the place of calling programs
- *Stubs* take the place of subordinate programs



What A Driver Does

- *Simulates the calling component*
- May need a little more function
  - *Initialize* the environment
  - Load values passed to the module under test
  - Set global variables
  - ...
- Some drivers may provide additional services
  - Running a series of tests
  - Conditioning of stubs
  - Logging and reporting results
  - ...
- Should be kept *simple*

What A Stub Does

- *Simulates behaviour of subordinate program or hardware*
  - Subroutine, function, procedure
  - Hardware interrupts, sending and receiving data
- Has the *same interface* as the real thing, but not the logic
  - Provides pre-canned responses with a constant value
  - Generates same interrupt every time
- *Used in place* of the real thing
  - Compiled and linked instead of the real program

What A Stub May Do

- A stub is intended to be *simple*
  - We do not want to have to set up elaborate tests to ensure that the stub code is correct!
- Many stubs *not much more than one line*
- But *may* have a little more function
  - Return one response for a number of times, and then return a different response
- Stubs *may* be designed to read a ‘constant value’ out of a small file, which can be easily edited
  - Can change test values without recompiling
- Stubs *may* be ‘conditioned’ by a driver before execution

What And How Much To Unit Test?

- Unit testing will normally use *white box* test methods
  - *Path testing*
  - *Complete coverage*
  - Tests derived from *program specification*
    - Verification, not validation
    - Examining internal workings
- Unit test may occasionally include *black box* methods
  - Additional to white box testing

What To Do With Defects Found?

- For *tests that fail*, the programmer needs help – information about *what happened*
  - *Inputs* and *outputs* (expected and actual)
  - Program variables which *changed*
  - *Paths* traveled / *decisions* made
  - Failure *symptoms* / messages
    - Illegal instruction
    - Branch to strange address
    - Divide by zero
    - . . .
- All tests should be *recorded*

Object-Oriented Unit Tests

- In OO testing, the “module” is a *class* or *method*
- Additional considerations for OO
  - *Drivers* need to take account of OO environment
    - For example, constructors and destructors
  - *Stubs* may replace more than one class
  - *Inheritance* hierarchies can complicate the situation
    - Data types defined elsewhere have to be accessible
    - Interactions can get quite complicated to isolate
    - *Control flow not explicit* (dynamic dispatch)

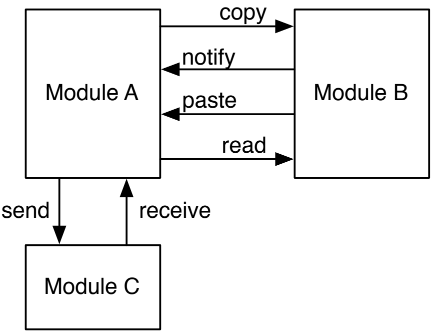
Contents

Integration Testing

- *When assembling* the system from components
- Usually carried out *step by step*
  - Various strategies,
    e.g., *bottom-up* or *top-down* or *incremental*
- Can begin as soon as concerned *components are ready*
  - Others can be added *gradually* as they become available
  - Consider order of integration during *design*
    - *Delay to complete* vital module delays whole testing

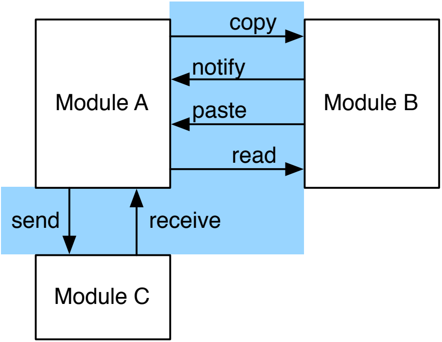
Testing At The Interfaces

- Integration testing focuses on *interfaces*
  - Program “Calls” in structured systems
  - Messages being passed in OO systems



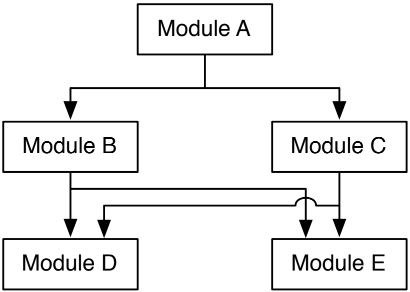
Testing At The Interfaces

- Integration testing focuses on *interfaces*
  - Program “Calls” in structured systems
  - Messages being passed in OO systems



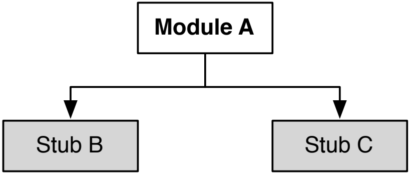
Top-Down Integration

- Start at the *top level* of the system architecture
  - with stubs below,
  - and *work downwards*



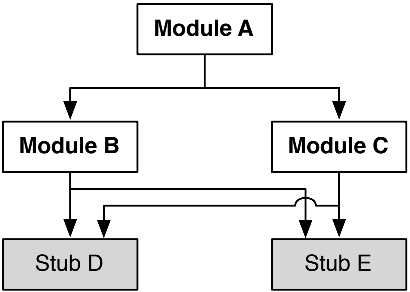
Top-Down Integration

- Start at the *top level* of the system architecture
  - with stubs below,
  - and *work downwards*



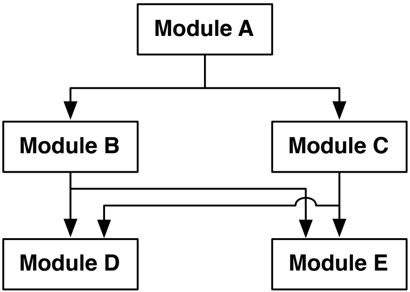
Top-Down Integration

- Start at the *top level* of the system architecture
  - with stubs below,
  - and *work downwards*



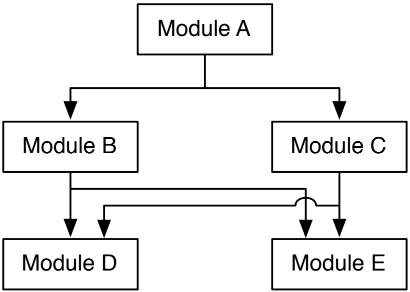
Top-Down Integration

- Start at the *top level* of the system architecture
  - with stubs below,
  - and *work downwards*



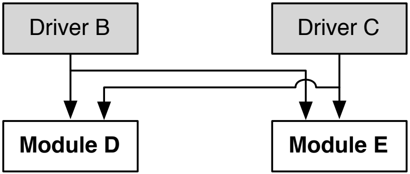
Bottom-Up Integration

- Start at the *lowest level* of the system architecture
  - with drivers above,
  - and *work upwards*



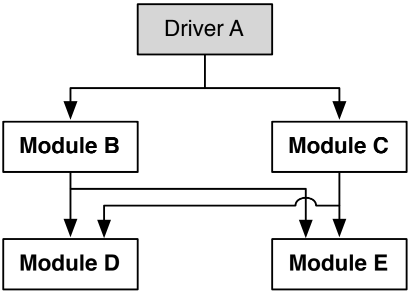
Bottom-Up Integration

- Start at the *lowest level* of the system architecture
  - with drivers above,
  - and *work upwards*



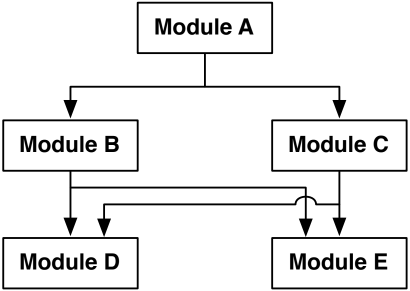
Bottom-Up Integration

- Start at the *lowest level* of the system architecture
  - with drivers above,
  - and *work upwards*



Bottom-Up Integration

- Start at the *lowest level* of the system architecture
  - with drivers above,
  - and *work upwards*



Contents

Problems Addressed By Black Box Testing

- Incorrect or missing *functions*
- Interface *errors*
- *Errors* in data structures
- Behaviour and performance *errors*
- Initialisation and termination *errors*

Two Black Box Testing Techniques

- Equivalence partitioning
- Boundary value analysis

How Many Black Box Tests?

- We have to *pick specific values* to use in test cases
  - (*Not enough time* to test all possible values)
- *Equivalence partition* analysis
  - Helps to select a sensible number of *test cases*
  - *Boundary value* testing ties in with this

Equivalence Partitioning

- Equivalence class
  - Sets of *inputs* that result in *similar program behaviour*
  - Sets of valid or invalid *states for input conditions*
- One may consider
  - *Input* equivalence classes
  - *Output* equivalence classes
- One approach is to consider *output* “messages” produced
  - May indicate what the *input* equivalence partitions are
- *Guidelines* for deriving equivalence classes:

Input condition	#(Equivalence classes)	
	Valid	Invalid
Range of numeric values	1	2
Specific numeric values	1	2
Set of values	1	1
Boolean	1	1

Examples Of Equivalence Classes

- **Valid input** is a month number $[1 \dots 12]$
  Equivalence classes are: $[-\infty \dots 0], [1 \dots 12], [13 \dots \infty]$
- **Valid input** is the name of a formal modelling method
  Two equivalence classes
  - [“Z”, “ASM”, “VDM”, “B”, “Event-B”,“TLA+”]
  - All other strings

Equivalence Classes For Dependent Inputs

- Two inputs: measure unit and speed value
- **One valid input** is either ‘Metric’ or ‘US/Imperial’
  - Equivalence classes are: [“Metric”], [“US/Imperial”], Other
- **Other valid input** is speed: 1 to 750 km/h or 1 to 500 mph
  - Validity depends on whether metric or US/imperial
  - Equivalence classes are:

    $[-\infty \ldots 0], [1 \ldots 500], [501 \ldots 750], [751 \ldots \infty]$
- Some test combinations

Equivalence Classes For Dependent Inputs

- Two inputs: measure unit and speed value
- **One valid input** is either ‘Metric’ or ‘US/Imperial’
  - Equivalence classes are: [“Metric”], [“US/Imperial”], Other
- **Other valid input** is speed: 1 to 750 km/h or 1 to 500 mph
  - Validity depends on whether metric or US/imperial
  - Equivalence classes are:

    $[-\infty \ldots 0], [1 \ldots 500], [501 \ldots 750], [751 \ldots \infty]$
- Some test combinations

Metric	$[-\infty \ldots 0]$	*invalid*
Metric	$[1 \ldots 500]$	*valid*
US/Imperial	$[501 \ldots 750]$	*invalid*
Metric	$[501 \ldots 750]$	*valid*
US/Imperial	$[751 \ldots \infty]$	*invalid*

Which Values To Select From A Class?

- Once *equivalence classes* are identified,
  need to *determine test data values* for each class
- Select values
  at the *boundaries* and
  somewhere in the middle of each class
- This is called *Boundary Value Analysis*
- Errors tend to occur at boundaries of input values
  (Reason unknown)

Boundary Values Of Equivalence Classes

- **Valid input** is a month number $[1 \ldots 12]$

  Equivalence classes are:

  Boundaries:

- **Valid input** is the name of a formal modelling method

  Two equivalence classes

  -
  -

  Boundaries:

Boundary Values Of Equivalence Classes

- **Valid input** is a month number $[1 \ldots 12]$

  Equivalence classes are: $[-\infty \ldots 0], [1 \ldots 12], [13 \ldots \infty]$

  Boundaries: $0, 1, 12, 13$

- **Valid input** is the name of a formal modelling method

  Two equivalence classes

  - [“Z”, “ASM”, “VDM”, “B”, “Event-B”,“TLA+”]
  - All other strings

  Boundaries: ?

Contents

System Testing

- *System*: software, hardware, network, ...
- *Not* under control of software engineer
- Must be *planned ahead*
- System test *focus*
  - Does the system do what it should?
  - The users' point of view
  - *Requirements*
- We are now carrying out *validation*
- *Black box* testing

Contents

Acceptance Testing

- *Done by users* at user's site
  - Developers are *not involved*
- Prepare
  - *Requirements* satisfied?
  - Involve users *during design* where possible
    - Usability!
- *Select and train users* for acceptance testing
- Environment needs to be set up — *first time at users' site?*

Contents

Regression Testing

- Ensure *changes* do not cause errors in already tested code
- *New code* likely to introduce *new defects*
- Regression testing may be required at *all stages*
  - During *development* and during *maintenance*
  - At all levels : UT, IT, ST
- If possible, *automate*

Contents

Concluding Remarks

- Testing must be *systematic*
- Testing begins with *requirements*
- Testing must be taken into account *during design*
- *White box testing* looks inside the program
- *Unit tests* tests the smallest units of a system
- *Integration tests* test communication between components
- *Black box testing* ignores the inside of the program
- Testing does *not* address all correctness problems
- Need for *complementary techniques*
  - Inspection
  - Formal methods