



Software Testing

Prof. Mark Micallef
mark.micallef@um.edu.mt

[People tell me that testing is...]

- Boring
- Not for developers
- A second class activity
- Not necessary because they are very good coders

[Testing for Developers]

- As a developer, you are responsible for a certain amount of testing (usually unit testing)
- Knowledge of testability concepts will help you build more testable code
- This leads to higher quality code

[Testing for Developers]

What does this method do? How would you test it?

```
public String getMessage(String name) {  
    Time time = new Time();  
    if (time.isMorning()) {  
        return "Good morning " + name + "!";  
    } else {  
        return "Good evening " + name + "!";  
    }  
}
```

[Testing for Developers]

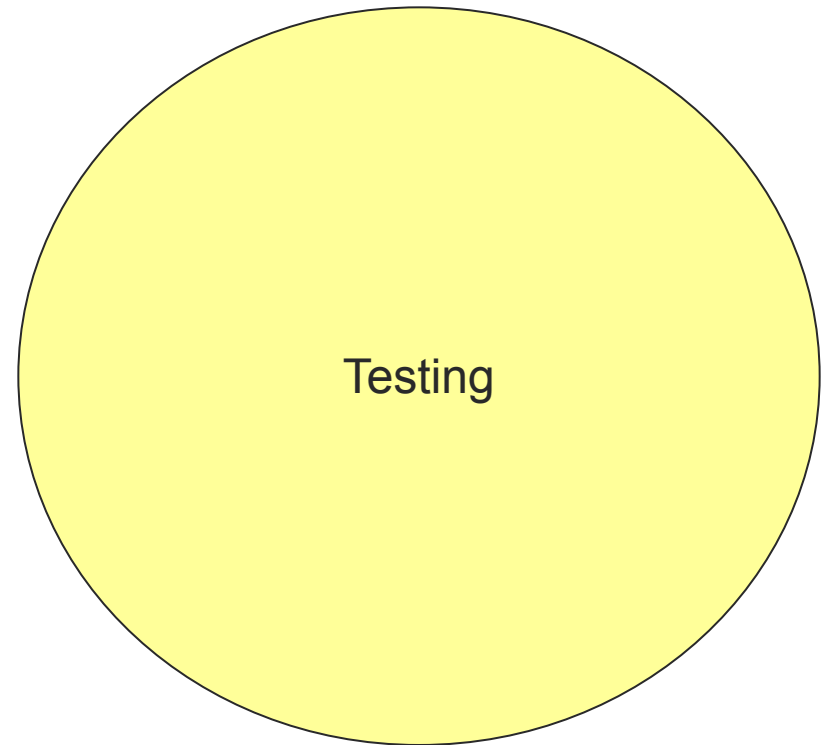
Why is this more testable?

```
public String getMessage(String name, Time time) {  
    if (time.isMorning()) {  
        return "Good morning " + name + "!";  
    } else {  
        return "Good evening " + name + "!";  
    }  
}
```

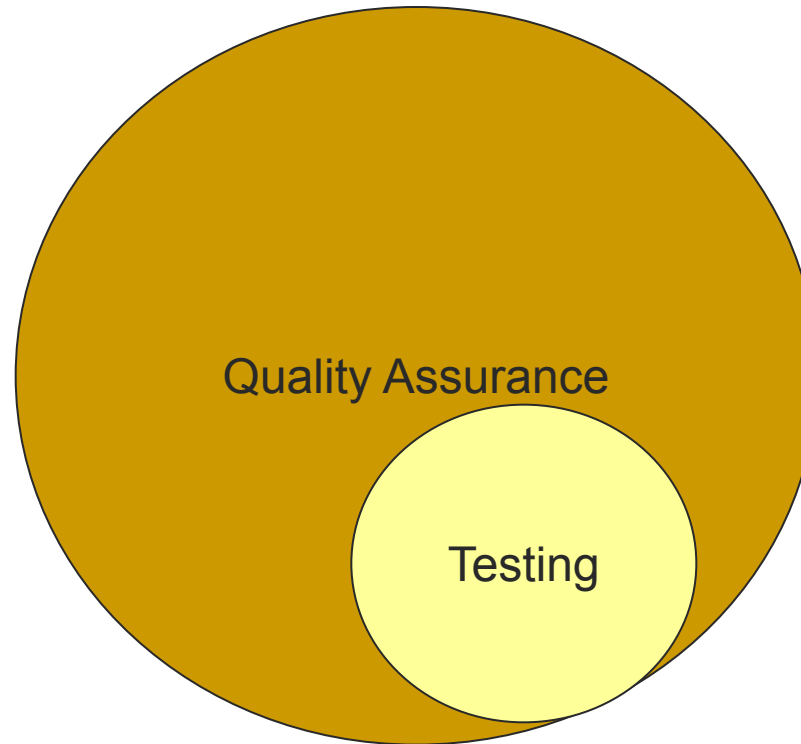
[What is testing?]

- Testing is a process of executing a software application with the **intent of finding errors and to verify that it satisfies specified requirements** (BS 7925-1)
- Testing is the process of exercising or evaluating a system or a system component by manual or automated means to verify that it **satisfies specified requirements** or to **identify differences between expected and actual results**. (IEEE)
- Testing is a **measurement of software quality** in terms of defects found, for both functional and non-functional software requirements and characteristics. (ISEB Syllabus)

[Quality Assurance vs Testing]



[Quality Assurance vs Testing]



[Quality Assurance]

- Multiple activities throughout the dev process
 - Development standards
 - Version control
 - Change/Configuration management
 - Release management
 - Testing
 - Quality measurement
 - Defect analysis
 - Training

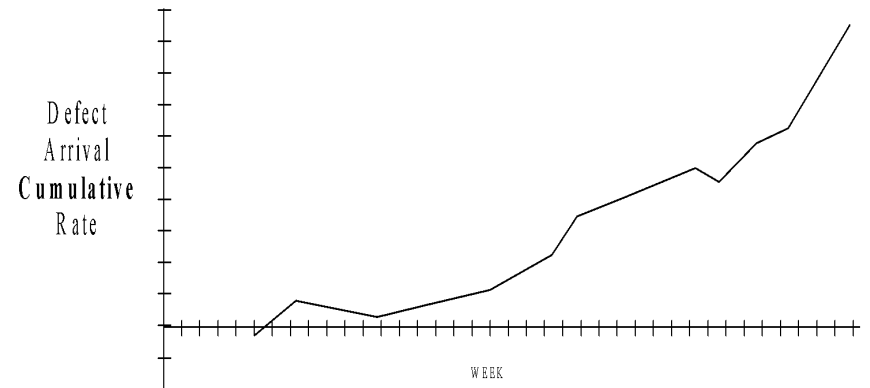
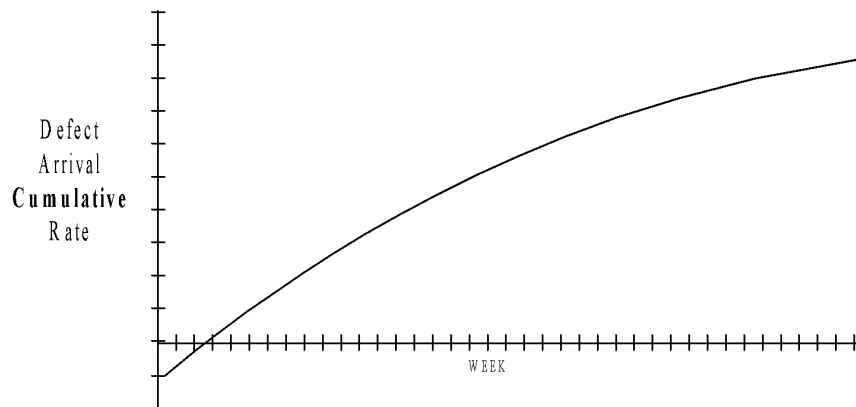
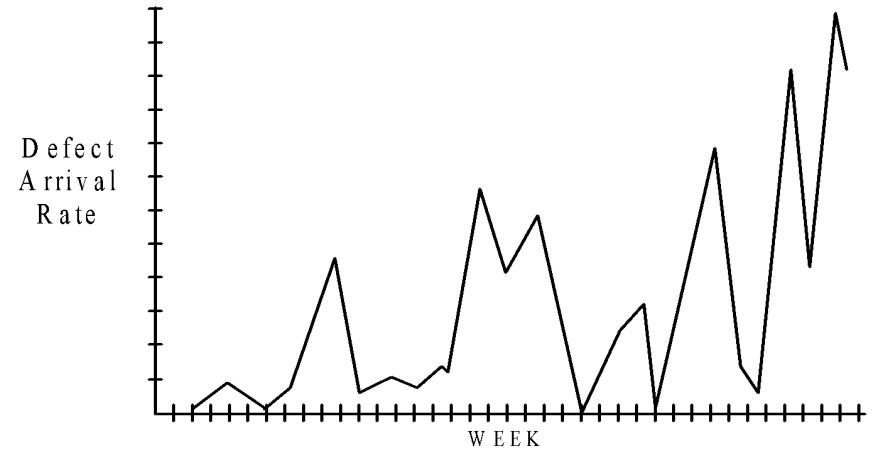
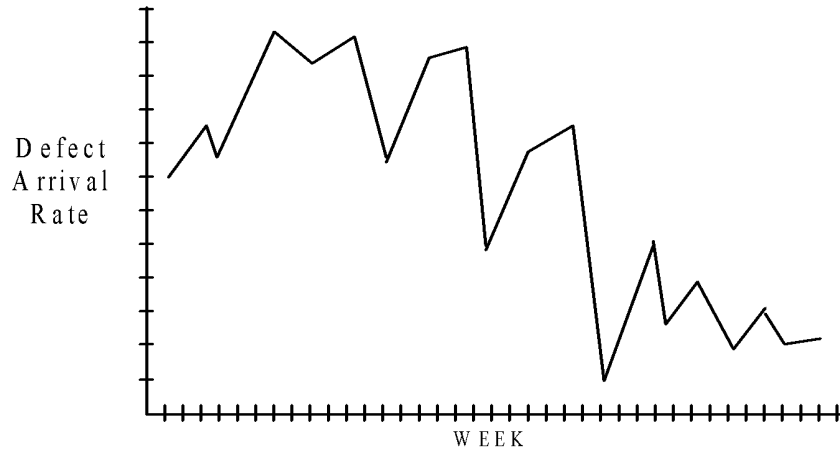
[Testing]

- Also consists of multiple activities
 - Unit testing
 - Whitebox Testing
 - Blackbox Testing
 - Data boundary testing
 - Code coverage analysis
 - Exploratory testing
 - Ad-hoc testing
 - ...

[Testing Axioms]

- Testing cannot show that bugs do not exist
- Exhaustive testing is impossible for non-trivial applications
- Software Testing is a Risk-Based Exercise. Testing is done differently in different contexts, i.e. safety-critical software is tested differently from an e-commerce site.
- Testing should start as early as possible in the software development life cycle
- The More Bugs you find, the More bugs there are.

Bug Counts vs Defect Arrival Patterns



Project A

Project B

[Errors, Faults and Failures]

- Error – a human action that produces an incorrect result
- Fault/defect/bug – an incorrect step, process or data definition in a computer program, specification, documentation, etc.
- Failure – The deviation of the product from its expected behaviour. This is a manifestation of one or more faults.

[Common Error Categories]

- Boundary-Related
- Calculation/Algorithmic
- Control flow
- Errors in handling/interpreting data
- User Interface
- Exception handling errors
- Version control errors

[Testing Principles]

- All tests should be traceable to customer requirements
 - The objective of software testing is to uncover errors.
 - The most severe defects are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins
 - Detailed tests can be defined as soon as the system design is complete
- Tests should be prioritised by risk since it is impossible to exhaustively test a system.
 - Pareto principle holds true in testing as well.

[What do we test? When do we test it?]

- All artefacts, throughout the development life cycle.
- Requirements
 - Are they complete?
 - Do they conflict?
 - Are they reasonable?
 - Are they testable?

[What do we test? When do we test it?]

■ Design

- Does this satisfy the specification?
- Does it conform to the required criteria?
- Will this facilitate integration with existing systems?

■ Implemented Systems

- Does the system do what is it supposed to do?

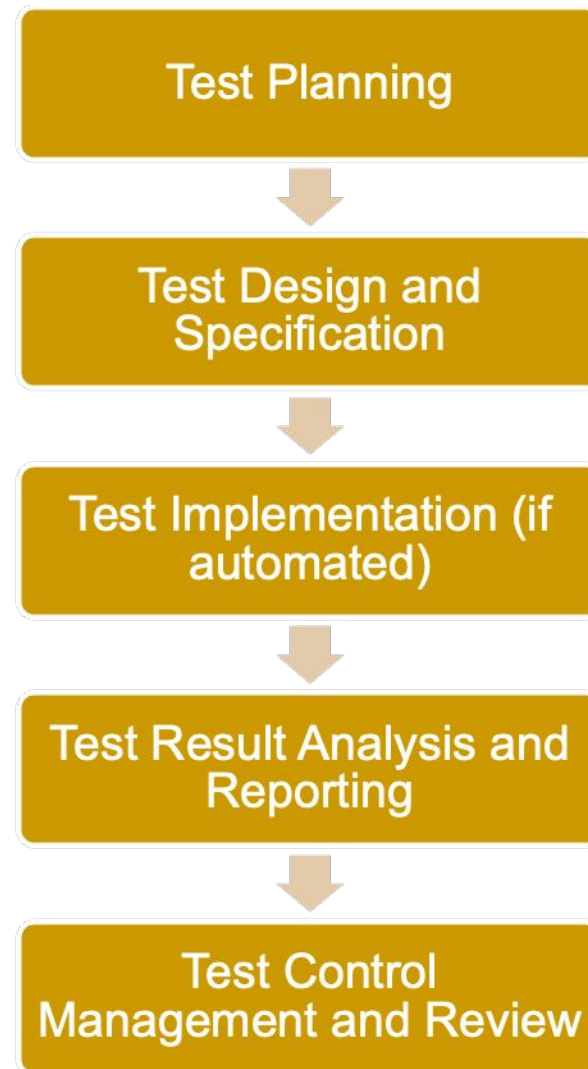
■ Documentation

- Is this documentation accurate?
- Is it up to date?
- Does it convey the information that it is meant to convey?

[Summary so far...]

- Testing is an important part of the software development process
- It is the responsibility of multiple stakeholders, including developers, testers and product owners
- Testing should be done throughout
- Basic Definitions

[The Testing Process]



[Test Planning]

- Test planning involves the establishment of a test plan
- Common test plan elements:
 - Entry criteria
 - Testing activities and schedule
 - Testing tasks assignments
 - Selected test strategy and techniques
 - Required tools, environment, resources
 - Problem tracking and reporting
 - Exit criteria

[Test Design and Specification]

- Review the test basis (requirements, architecture, design, etc)
- Evaluate the testability of the requirements of a system
- Identifying test conditions and required test data
- Design the test cases
 - Identifier
 - Short description
 - Priority of the test case
 - Preconditions
 - Execution
 - Post conditions
- Design the test environment setup (Software, Hardware, Network Architecture, Database, etc)

[Test Implementation]

- Only when using automated testing
- Can start right after system design
- May require some core parts of the system to have been developed
- Use of record/playback tools vs writing test drivers

[Test Execution]

- Verify that the environment is properly set up
- Execute test cases
- Record results of tests (PASS | FAIL | NOT EXECUTED)
- Repeat test activities
 - Regression testing

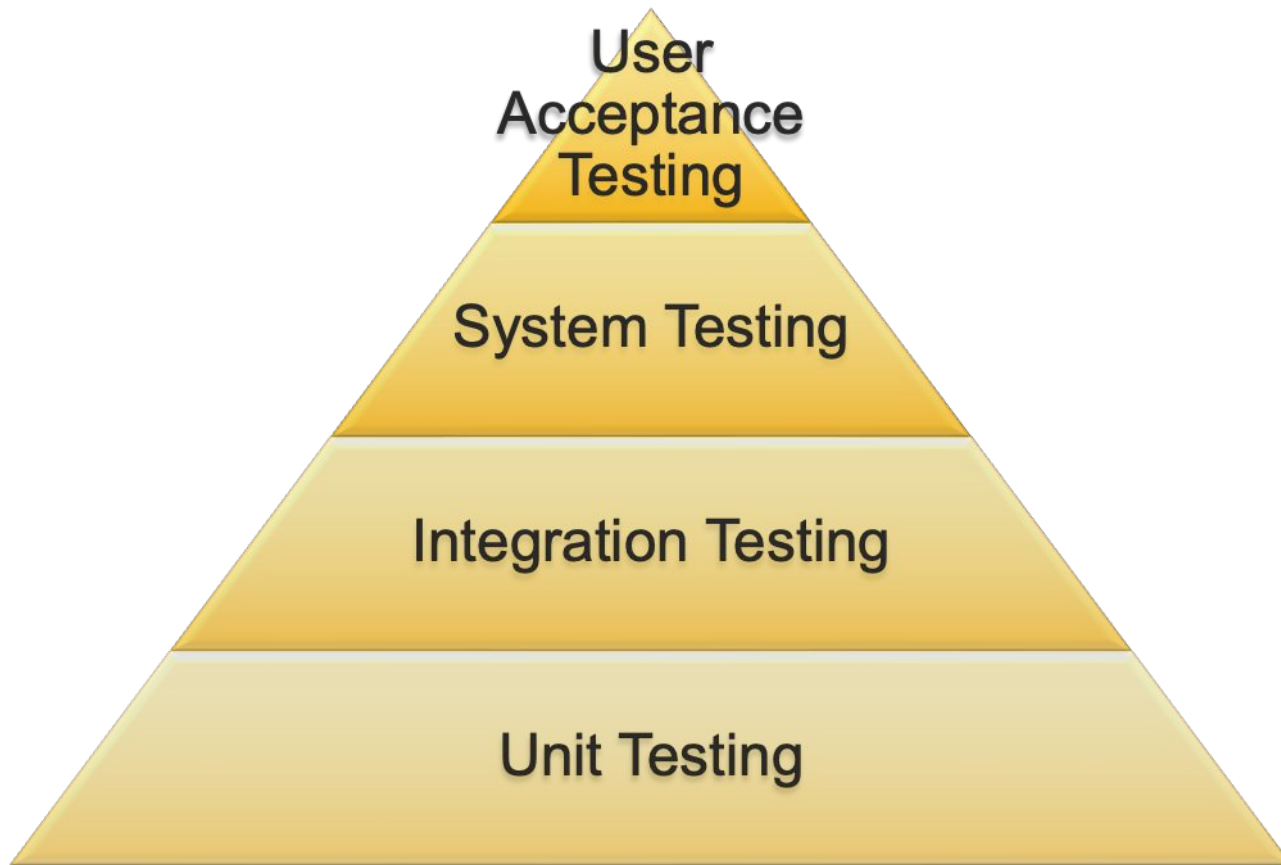
[Result Analysis and Reporting]

- Reporting problems
 - Short Description
 - Where the problem was found
 - How to reproduce it
 - Severity
 - Priority
- Can this problem lead to new test case ideas?

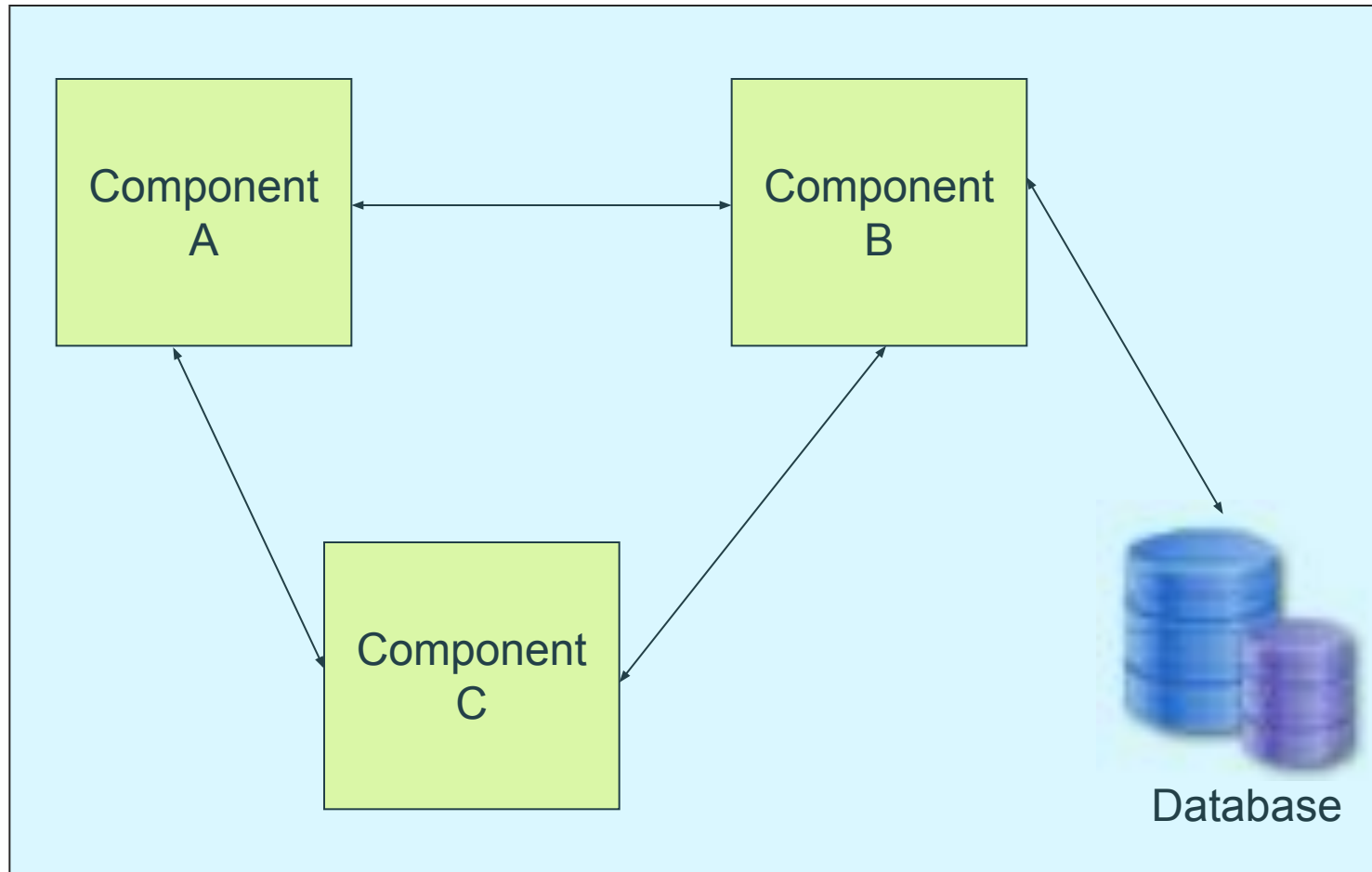
Test Control, Management and Review

- Exit criteria should be used to determine when testing should stop. Criteria may include:
 - Coverage analysis
 - Faults pending
 - Time
 - Cost
- Tasks in this stage include
 - Checking test logs against exit criteria
 - Assessing if more tests are needed
 - Write a test summary report for stakeholders

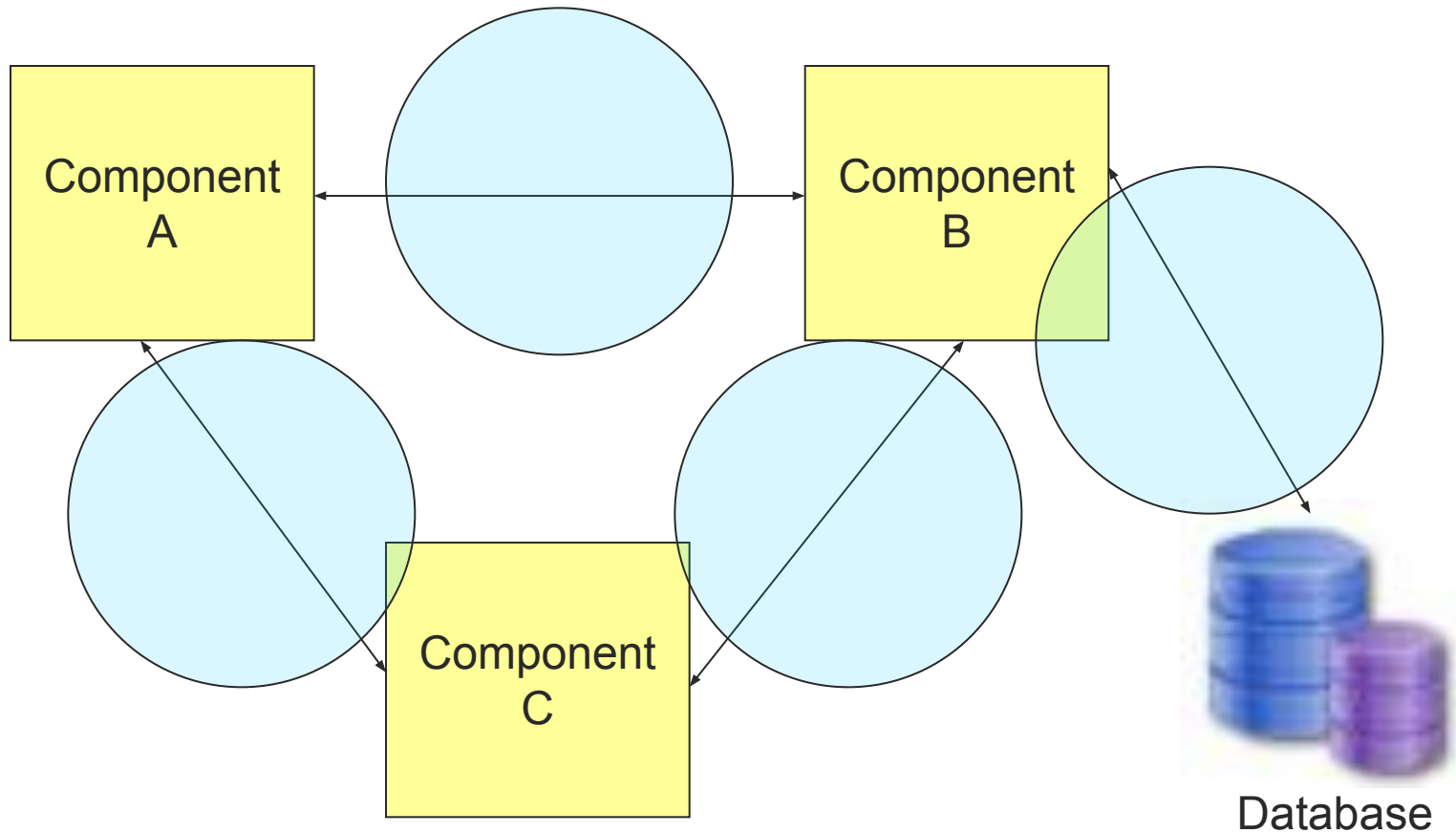
[Levels of Testing]



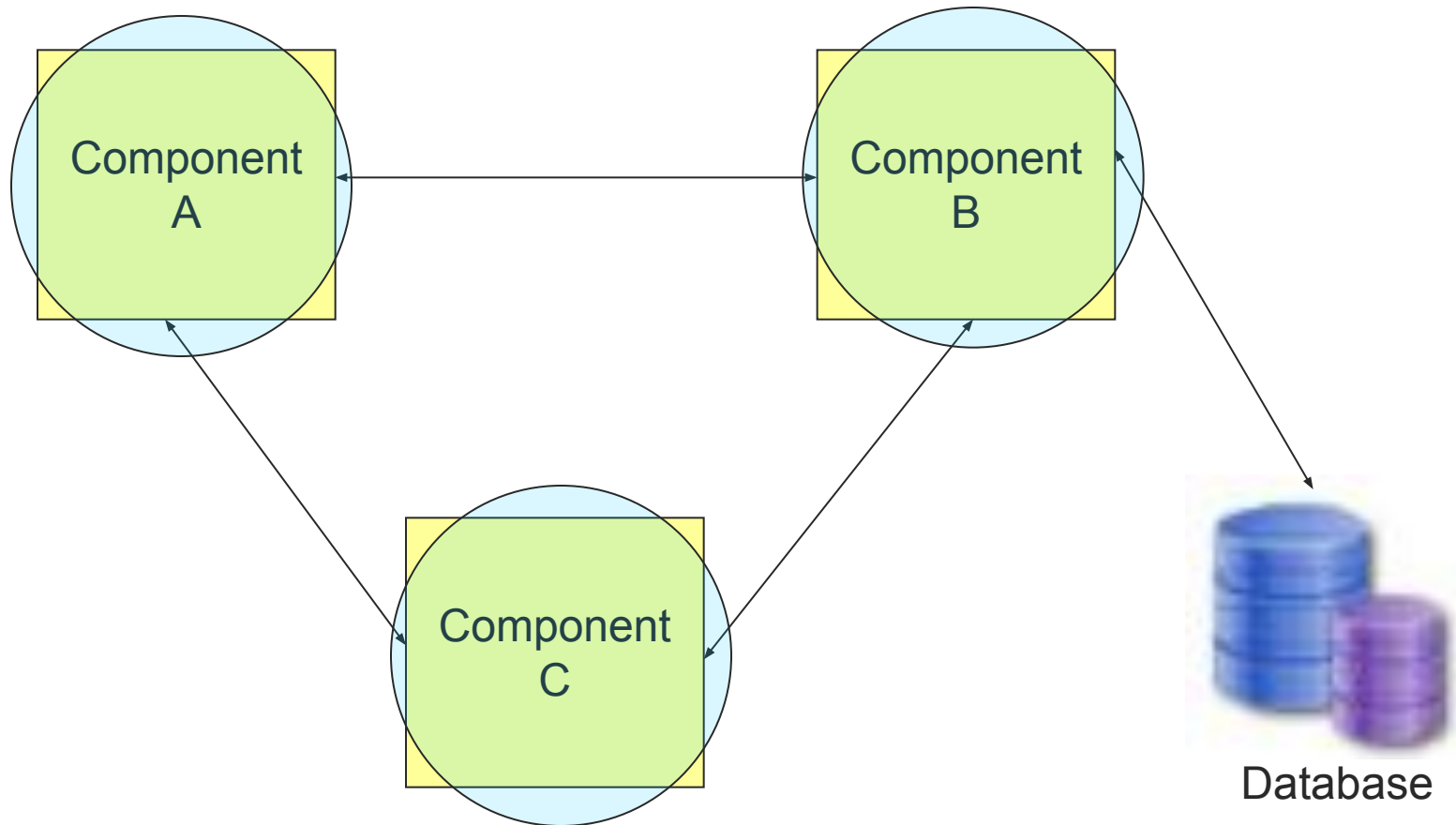
[System Testing]



[Integration Testing]



[Unit Testing]



[Unit Testing]

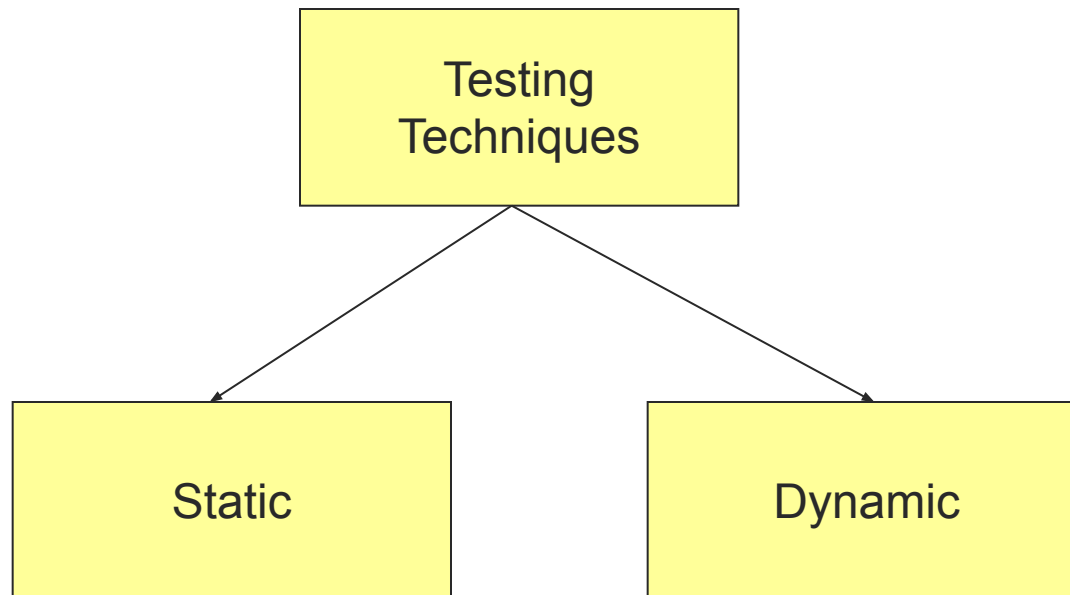
- calculateAge("01/01/1985")
 - Should return: 25
- calculateAge("03/09/2150")
 - Should return: ERROR
- calculateAge("55/55/55")
 - Should return: ERROR
- calculateAge("Bob")
 - Should Return: ERROR
- calculateAge("29/02/1987")
 - Should Return: ERROR

```
calculateAge(String  
dob)
```



Testing Techniques

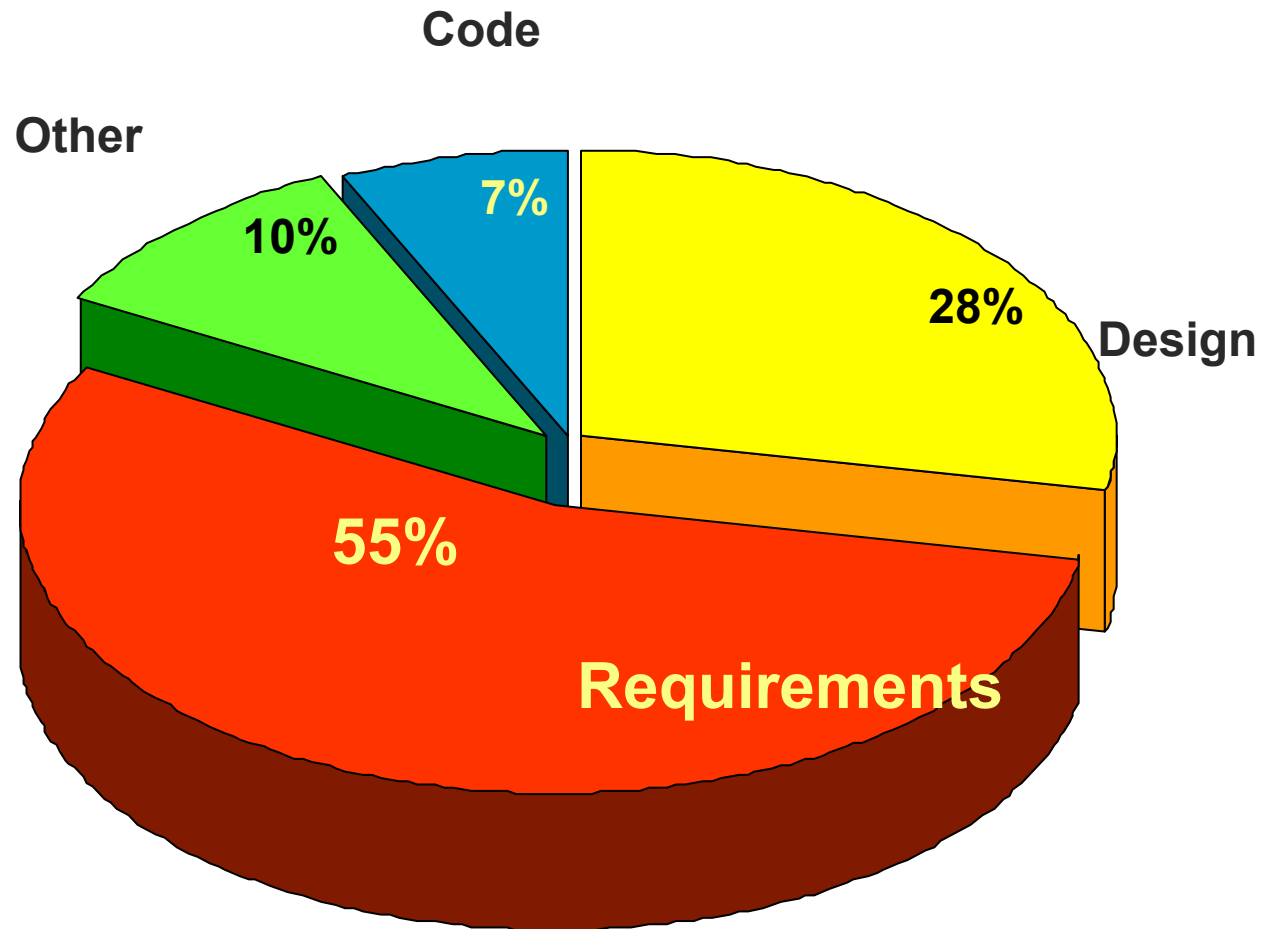
[Testing Techniques]



[Static Testing]

- Testing artefacts without actually executing a system
- Can be done from early stages of the development process
- Can include:
 - Requirement Reviews
 - Code walk-throughs
 - Enforcement of coding standards
 - Code-smell analysis
 - Automated Static Code Analysis
 - Tools: FindBugs, PMD

[Origins of software defects]



Typical faults found in reviews

- Deviation from Coding Standard
- Requirements defect
- Design Defects
- Insufficient Maintainability
- Lack of error checking

Java Coding Conventions on One Page

William C. Wake (William.Wake@acm.org), 2-17-2000

General conventions: see <http://java.sun.com/docs/codeconv/index.html>

Specify a package (not default)

```
package com.some.display;
```

May use "*"

```
import java.util.*;
```

Each word capitalized

```
public class NounPhrase implements Interfaceable {
```

Each word capitalized: noun phrase + optional "able" or "ing"

```
    public static final int MIN_SIZE = 22;
```

All caps with "_" separator

```
    static int height;
```

Constant

No special prefix for "static" variables

```
    String lastName;
```

Noun phrase. Fields not "public".

Javadoc conventions: see <http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html>

```
    /** @deprecated Use getLastName(), not getName() */
```

One-liners are OK

```
    public String getName() { return lastName; }
```

JavaBeans naming conventions: see <http://java.sun.com/beans/docs/beans.101.pdf>

```
    public int getScore() {
```

Braces here and here

```
        if (height < MIN_SIZE) {
```

```
            return MIN_SIZE;
```

```
        }
```

```
        return height;
```

```
    }
```

4-space indent throughout

```
protected void paste(String filename, Vector v) throws IOException {
```

```
    Writer out;
```

```
    try {
```

```
        out = new FileWriter(filename);
```

```
        super.save(v);
```

```
        out.println(v.size());
```

```
    } catch (FileNotFoundException fnf) {
```

```
        Log.log(fnf);
```

```
        throw fnf;
```

```
    } finally {
```

Exceptions either re-thrown or handled

"Finally" clause to release resources

```
        if (out != null) try { out.close(); } catch (Exception ignored) {}
```

```
    }
```

```
}
```

Catch any exceptions in "finally" clause so original exception is reported

[Code Smells]



[Code Smells]

- An indication that something may be wrong with your code.
- A few examples
 - Very long methods
 - Duplicated code
 - Long parameter lists
 - Large classes
 - Unused variables / class properties
 - Shotgun surgery (one change leads to cascading changes)



[Pros/Cons of Static Testing]

- Pros

- Can be done early
- Can provide meaningful insight

- Cons

- Can be expensive
- Tends to throw up many false positives

[Dynamic Testing Techniques]

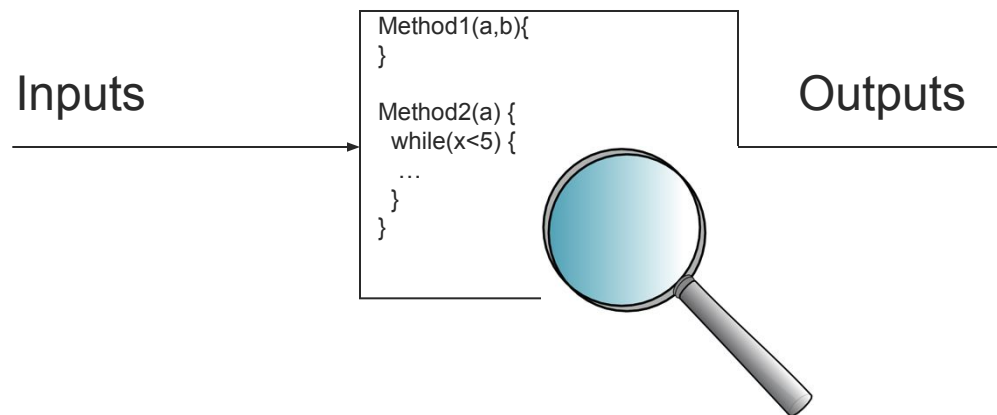
- Testing a system by executing it
- Commonly used taxonomy:
 - Black box testing
 - White box testing

[Black box Testing]



- Confirms that requirements are satisfied
- Assumes no knowledge of internal workings
- Examples of black box techniques:
 - Boundary Value Analysis
 - Error Guessing
 - State transition analysis

[White box Testing]



- Design tests based on your knowledge of system internals
- Examples of white box techniques:
 - Testing individual functions, libraries, etc
 - Designing test cases based on your knowledge of the code
 - Monitoring the values of variables, time spent in each method, etc-
 - Code coverage analysis – which code is executing?

[Test case design techniques]

- A good test case
 - Has a reasonable probability of uncovering an error
 - Is not redundant
 - Is not complex
- Various test case design techniques exist

[Test to Pass vs Test to Fail]

- Test to pass

- Only runs happy-path tests
- Software is not pushed to its limits
- Useful in user acceptance testing
- Useful in smoke testing

- Test to fail

- Assumes software works when treated in the right way
- Attempts to force errors

[Various Testing Techniques]

- Experience-based
 - Ad-hoc
 - Exploratory
- Specification-based
 - Functional Testing
 - Domain Testing

[Experience-based Testing]

- Use of experience to design test cases
- Experience can include
 - Domain knowledge
 - Knowledge of developers involved
 - Knowledge of typical problems
- Two main types
 - Ad Hoc Testing
 - Exploratory Testing

[Ad-hoc vs Exploratory Testing]

- Ad-hoc Testing
 - Informal testing
 - No preparation
 - Not repeatable
 - Cannot be tracked
- Exploratory Testing
 - Also informal
 - Involves test design and control
 - Useful when no specification is available
 - Notes are taken and progress tracked

[Specification-Based Testing]

- Designing test-cases to test specific specifications and designs
- Various categories
 - Functional Testing
 - Decomposes functionality and tests for it
 - Domain Testing
 - Random Testing
 - Equivalence Classes
 - Combinatorial testing
 - Boundary Value Analysis

[Test Design Techniques]

How would you test these methods?

- `String getMonthName(int month)`
- `void plotPixel(int x, int y)`
- `void plotPixel(int x, int y, int r, int g, int b)`

[Analogy – CockroachHunt]

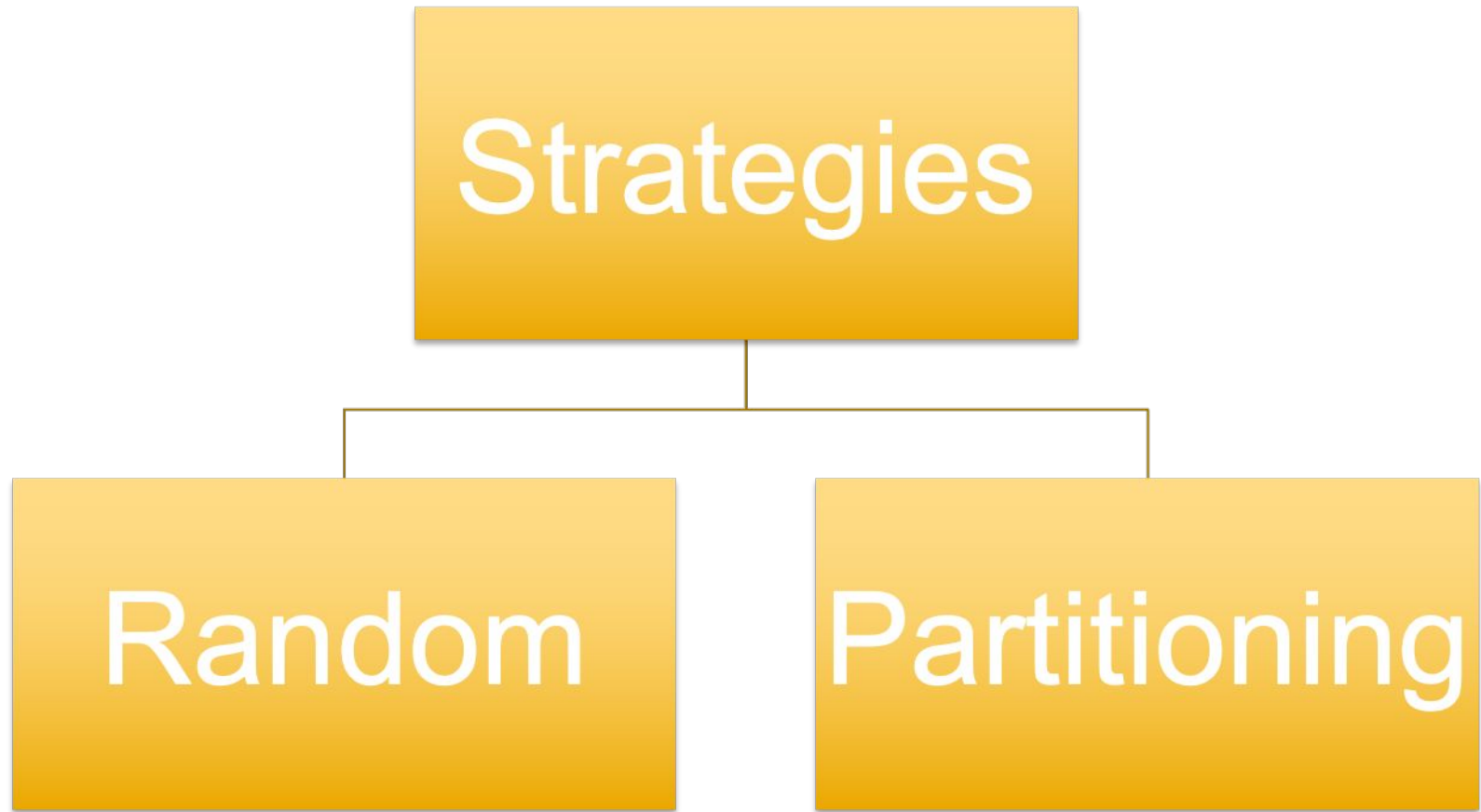


[Analogy – Cockroach Hunt]



Analogy – Cockroach Hunt

[Test Design Strategies]



Random Strategies

1



[Partitioning Strategies]



[Equivalence Classes]

- Also referred to as:
 - Equivalence partitions or
 - Equivalence class partitioning
- Classifies ranges of inputs
- Each class/partition is known/assumed to be treated in the same way by the software
- Representative test cases can then be derived from each class

[Equivalence Classes]

Input Rules	Valid Equivalence Classes	Invalid Equivalence Classes
A two digit positive number	$10 \leq x \leq 99$	$X < 10$ $X > 99$ X is not a number
Positive odd prime numbers less than 19	$\{1, 3, 5, 7, 11, 13, 17\}$	$X < 0$ $X > 0 \ \&\& \ X$ is even $\{9, 15\}$ $X \geq 19$

[Examples]

- What equivalence partitioning would you assign to parameters in these methods:
 - `String getDayName(int day)`
 - `boolean isValidHumanAge(int age)`
 - `boolean isHexDigit(char hex)`
 - `String calcLifePhase(int age)`
 - E.g. `calcLifePhase(1)` = “baby”
 - E.g. `calcLifePhase(17)` = “teenager”

[Boundary Value Analysis]

*“Bugs lurk in corners and
congregate at boundaries.”*

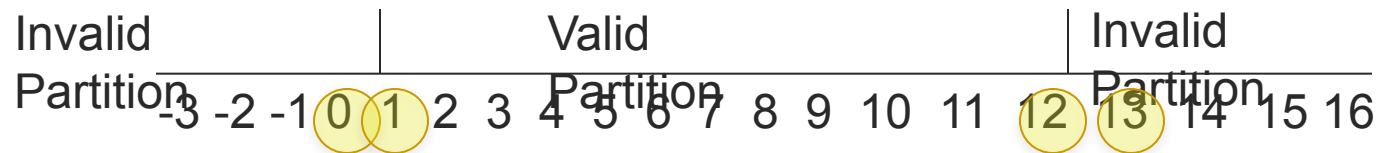
-Boris Beizer

Boundary Value Analysis

- Used to enhance test-case selection with equivalence partitions
- Attempts to pick test cases which exploit situations where developers are statistically known to make mistakes □ boundaries
- **General rule:** When picking test cases from equivalence partitions, be sure to pick test cases at every boundary.
- More test cases from within partitions can be added if desired.

[Boundary Value Analysis]

```
public String getMonthName(int  
month);
```



[Analysing the Quality of Tests]

- How can we provide some measure of confidence in our tests?
- Coverage analysis techniques:
 - Used to check how much of your code is actually being tested
 - Two main types:
 - Structural Coverage
 - Data Coverage
 - In this course we will only cover structural analysis techniques but you are expected to know the difference between structural and data coverage

[Structural Coverage]

- Measures how much of the “structure” of the program is exercised by tests
- Three main structures are considered:
 - Statements
 - Branches
 - Conditions

Java - CursorableLinkedList.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

JUnit

Finished after 34,898 seconds

Runs: 13009/13009 Errors: 0 Failures: 0

Failures

Hierarchy

junit.framework.TestSuite

junit.framework.TestSuite

TestBagUtils

org.apache.commons.collections.TestClos

org.apache.commons.collections.TestColle

TestBufferUtils

TestEnumerationUtils

org.apache.commons.collections.TestFact

TestListUtils

TestMapUtils

org.apache.commons.collections.TestPrec

TestSetUtils

org.apache.commons.collections.TestTran

TestArrayStack

TestBeanMap

org.apache.commons.collections.TestBina

TestBoundedFifoBuffer

TestBoundedFifoBuffer2

TestCursorableLinkedList

TestDoubleOrderedMap

org.apache.commons.collections.TestExte

TestFastArrayList

TestFastArrayList1

TestFastHashMap

TestFastHashMap1

TestFastTreeMap

TestFastTreeMap1

Failure Trace

CursorableLinkedList.java

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index || size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Problems Javadoc Declaration Console Coverage

TestAllPackages (31.10.2006 15:04:14)

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Writable Smart Insert 149 : 28

Statement vs Branch Coverage

```
void checkEvenNumber(int num) {  
  
    if ( (num % 2) == 1) {  
        System.out.println("not ");  
    }  
    System.out.println("an even number");  
}
```

- In this example:
 - You achieve 100% statement coverage with one test case
 - You achieve 100% branch coverage with 2 test cases

[Conditional Coverage]

```
void checkRealisticAge(int age) {  
  
    if ( age >= 0 && age <= 105) {  
        System.out.println("age is realistic");  
    } else {  
        System.out.println("not realistic");  
    }  
}
```

[Conclusion]

- This was an crash introduction to testing
- Testing is an essential part of the development process
- Even if you are not a test engineer, you have to be familiar with testing techniques

