

CS456/556 Software Design & Development

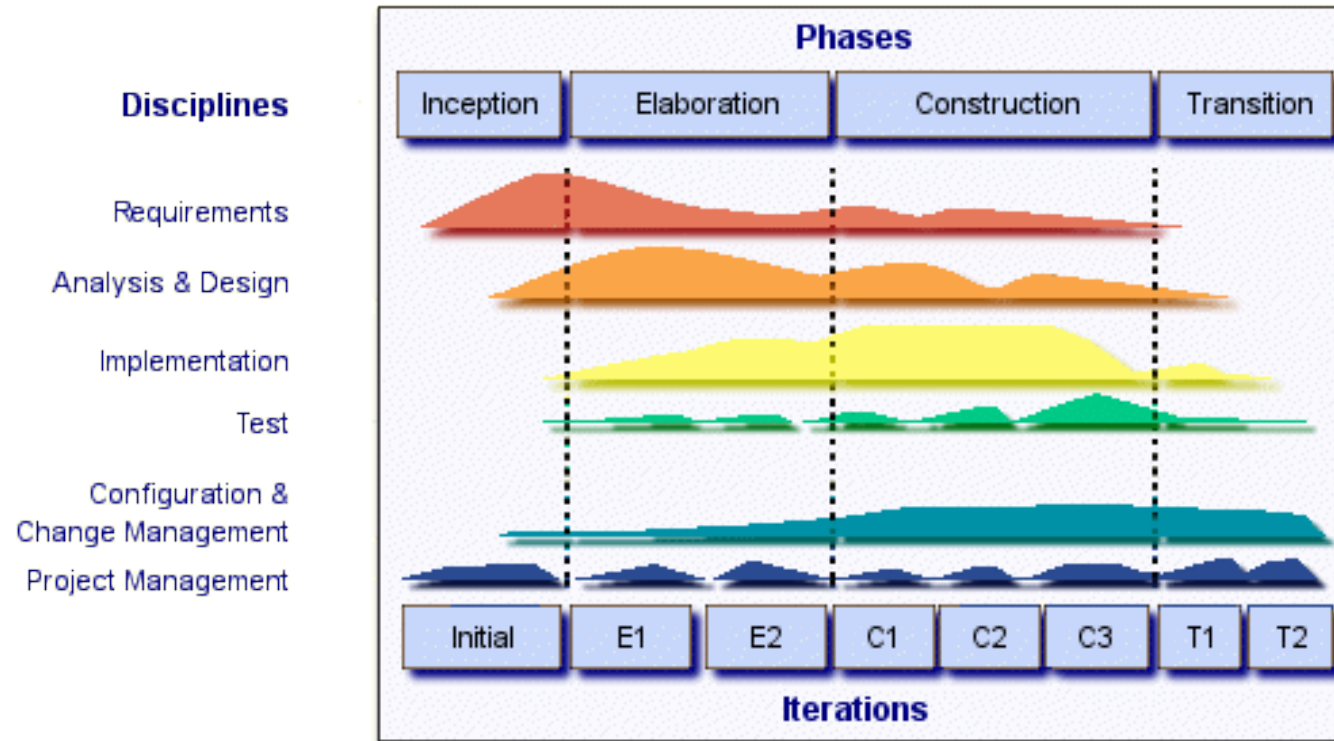
Test

Chang Liu

liuc@ohio.edu

<http://vital.cs.ohiou.edu>

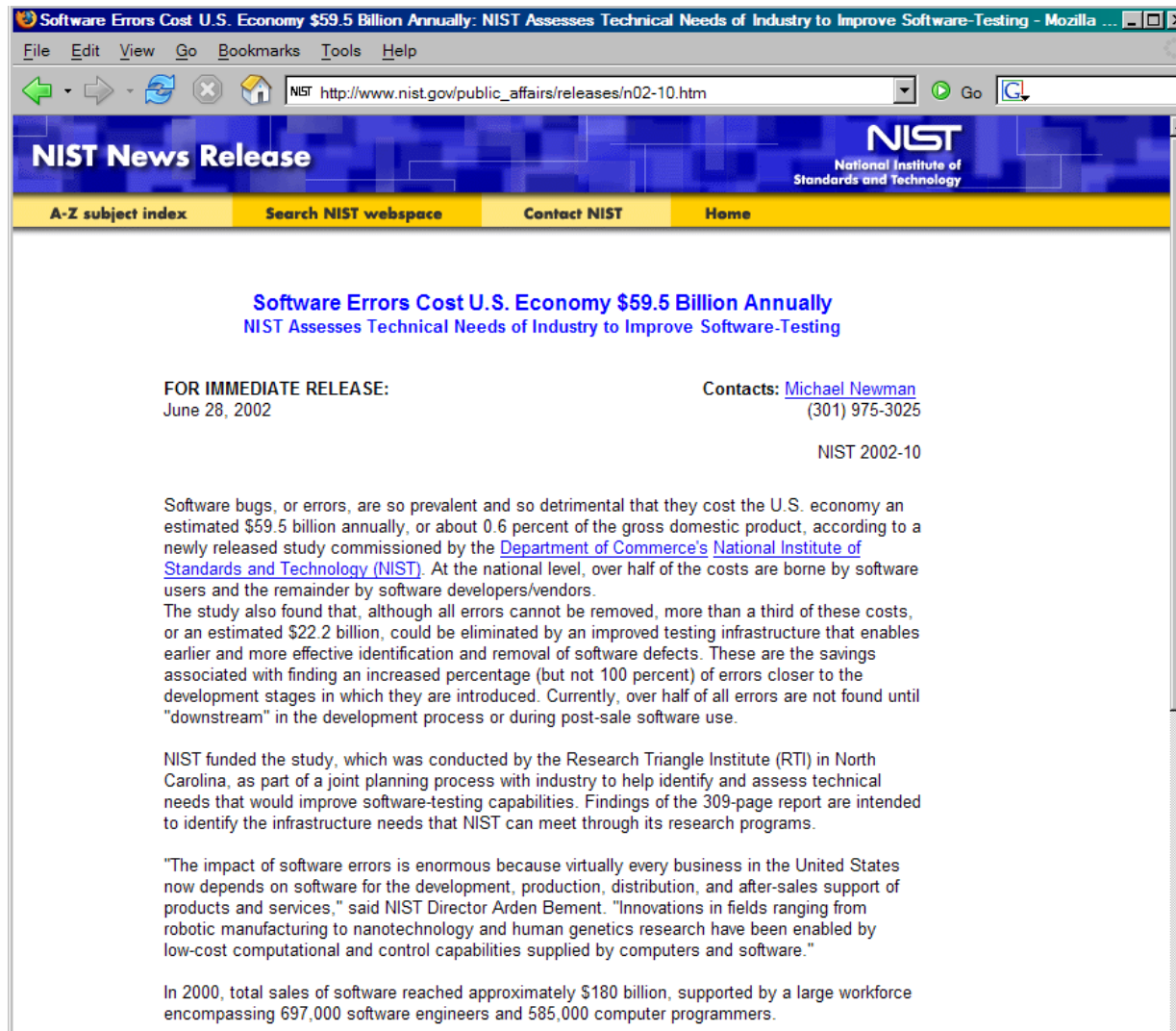
When Does Testing Happen?



Software Testing

- Plan the tests required for each iteration
- Design and implement the tests (create test cases)
- Perform the various tests at various stages and evaluate the test results

Why is Software Testing Important?



Software Errors Cost U.S. Economy \$59.5 Billion Annually
NIST Assesses Technical Needs of Industry to Improve Software-Testing

FOR IMMEDIATE RELEASE:
June 28, 2002

Contacts: [Michael Newman](#)
(301) 975-3025

NIST 2002-10

Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated \$59.5 billion annually, or about 0.6 percent of the gross domestic product, according to a newly released study commissioned by the [Department of Commerce's National Institute of Standards and Technology \(NIST\)](#). At the national level, over half of the costs are borne by software users and the remainder by software developers/vendors.

The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated \$22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage (but not 100 percent) of errors closer to the development stages in which they are introduced. Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.

NIST funded the study, which was conducted by the Research Triangle Institute (RTI) in North Carolina, as part of a joint planning process with industry to help identify and assess technical needs that would improve software-testing capabilities. Findings of the 309-page report are intended to identify the infrastructure needs that NIST can meet through its research programs.

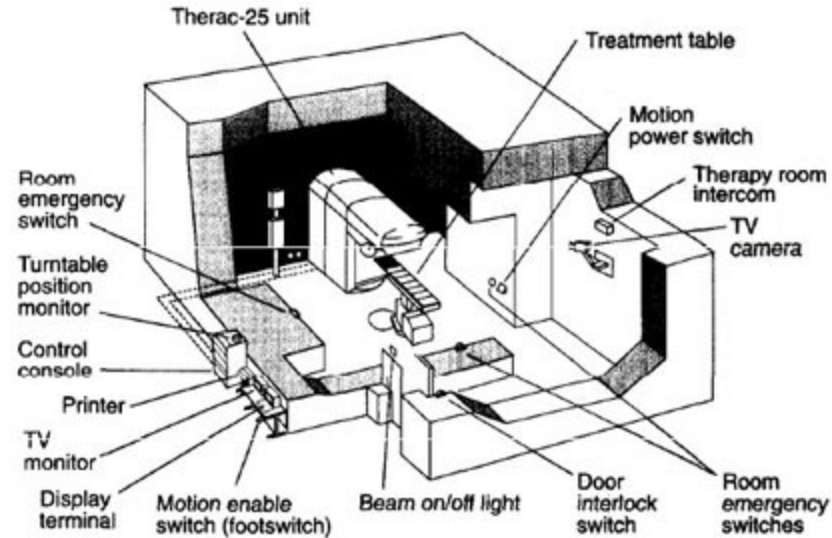
"The impact of software errors is enormous because virtually every business in the United States now depends on software for the development, production, distribution, and after-sales support of products and services," said NIST Director Arden Bement. "Innovations in fields ranging from robotic manufacturing to nanotechnology and human genetics research have been enabled by low-cost computational and control capabilities supplied by computers and software."

In 2000, total sales of software reached approximately \$180 billion, supported by a large workforce encompassing 697,000 software engineers and 585,000 computer programmers.

NIST Study

- Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated \$59.5 billion annually, or about 0.6 percent of the gross domestic product, according to a newly released study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST). At the national level, over half of the costs are borne by software users and the remainder by software developers/vendors.
- The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated \$22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage (but not 100 percent) of errors closer to the development stages in which they are introduced. Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.
- In 2000, total sales of software reached approximately \$180 billion, supported by a large workforce encompassing 697,000 software engineers and 585,000 computer programmers.

1985-1987 Therac-25



Why are defects in software difficult and expensive to detect?

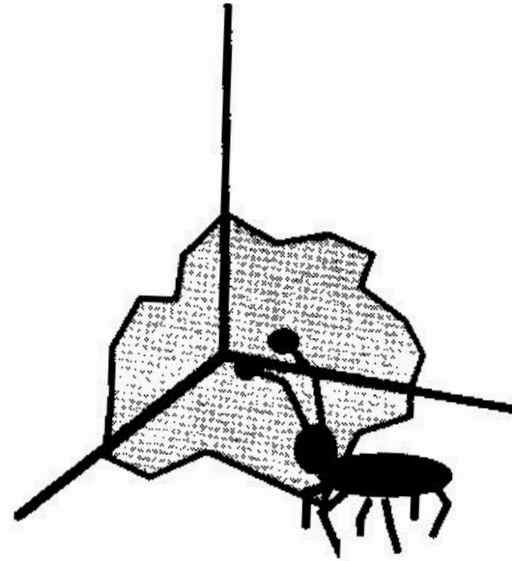
- Example: A bug in Visual Studio 6

```
class name| {  
    int var1;  
    int var2;  
  
    int method1(int arg1,int arg2)  
    {  
        //...  
    }  
}
```

Test Case Design

"Bugs lurk in corners
and congregate at
boundaries ..."

Boris Beizer



OBJECTIVE

to uncover errors

CRITERIA

in a complete manner

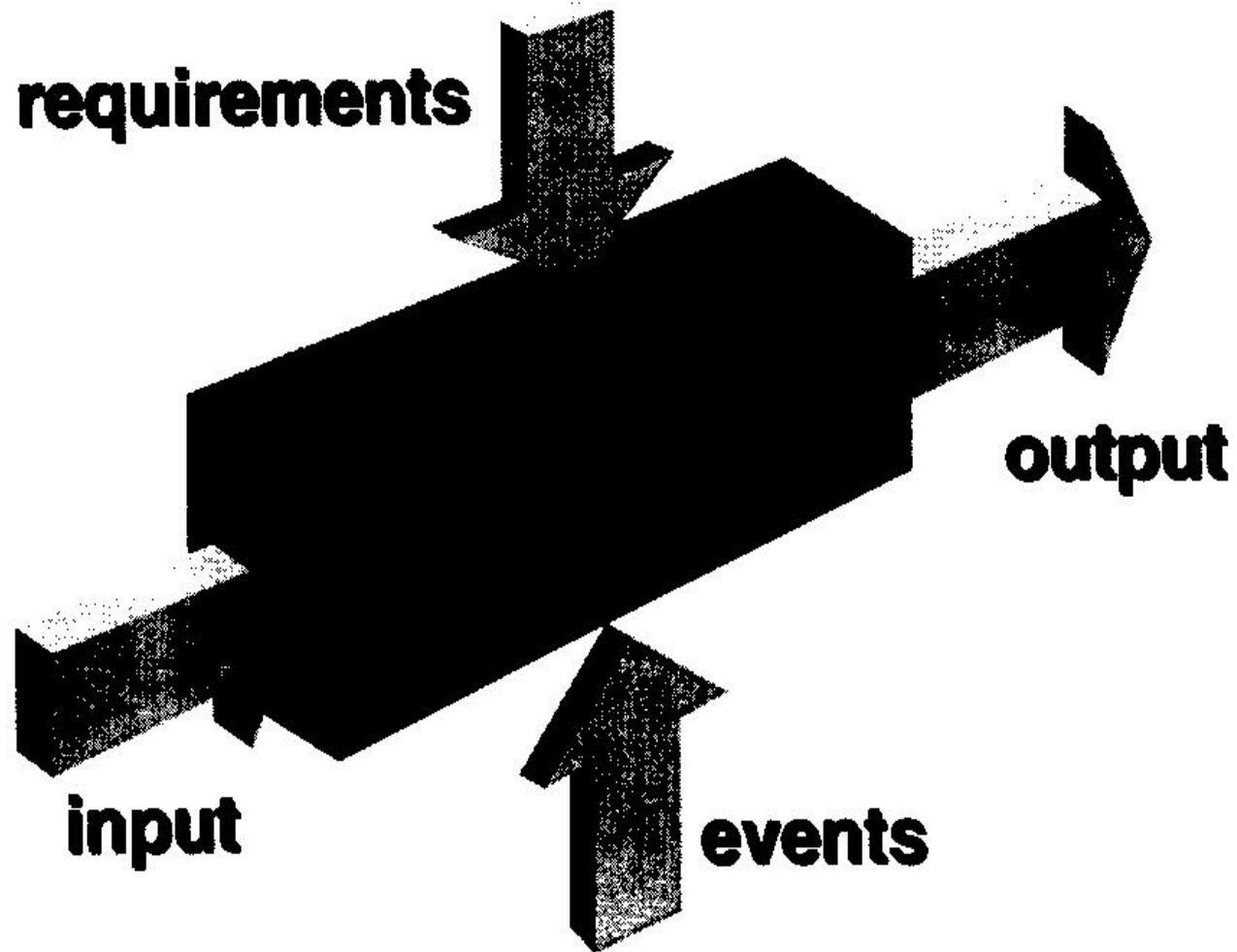
CONSTRAINT

with a minimum of effort and time

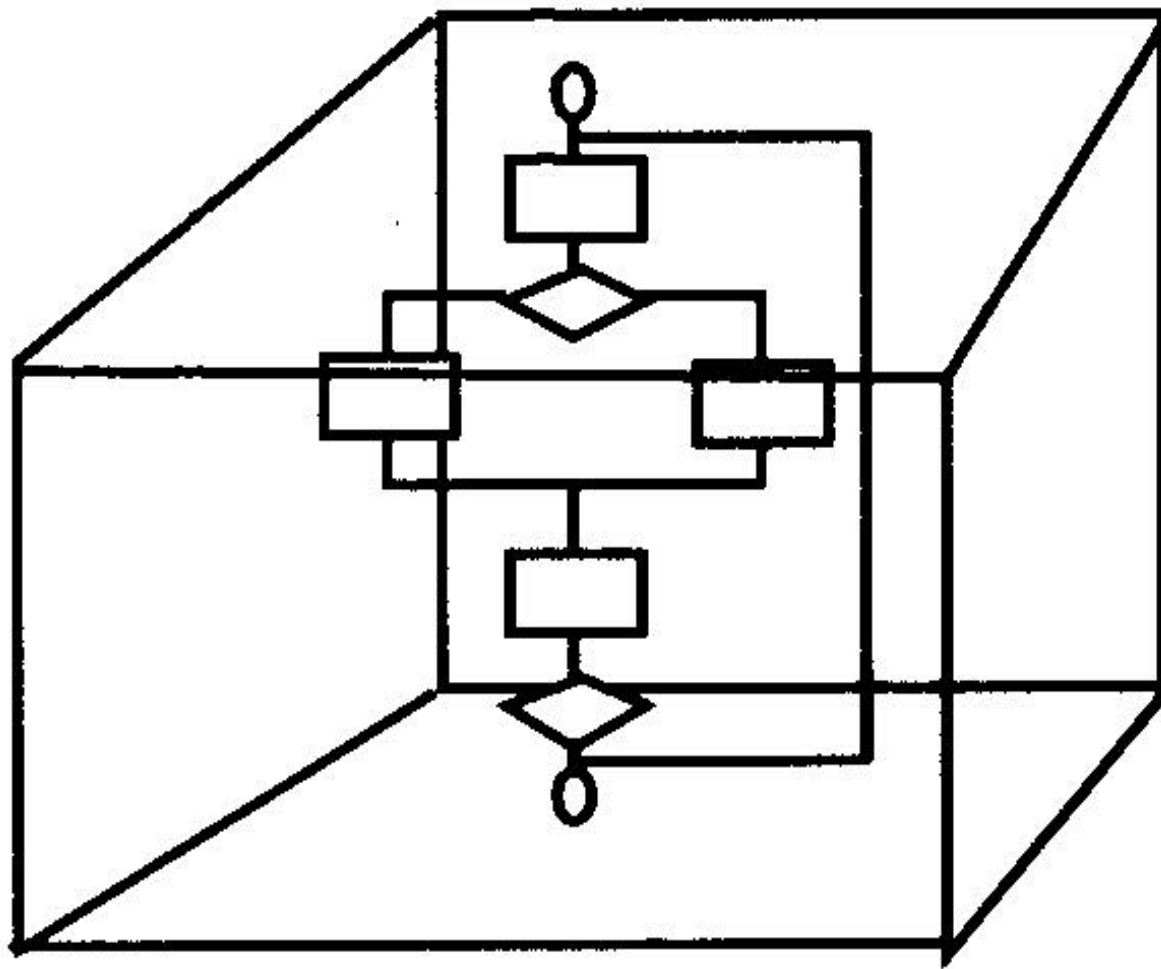
Unit Test

- **Specification Testing**
 - Black-box testing
- **Structure Testing**
 - White-box testing

Black Box Testing

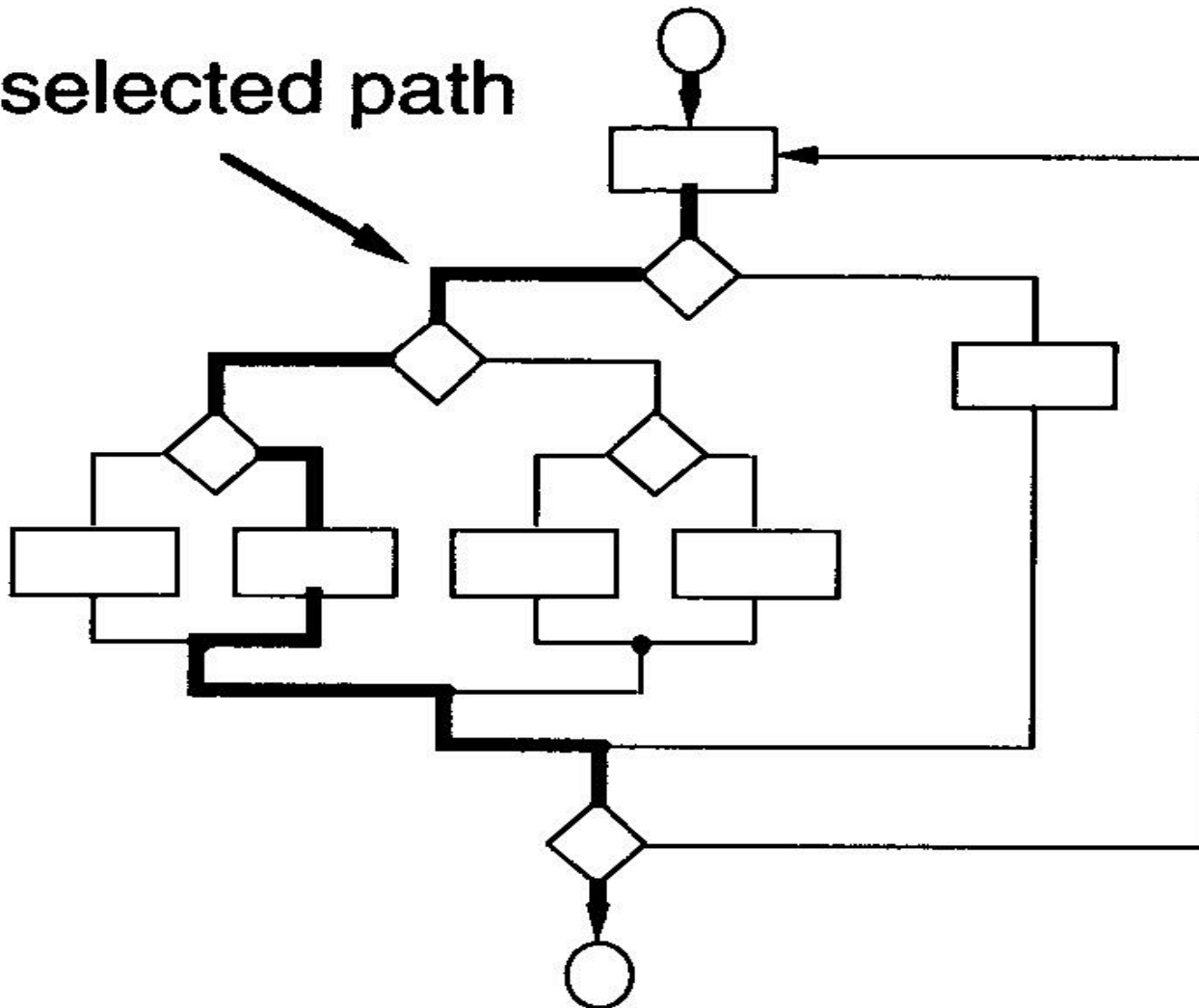


White Box Testing



White Box Testing

a selected path



loop $\leq 20 \times$

Concepts: Fault, Error, Failure, Bug, Defect

- Fault - A bug. A defect.
- Error - A mistake made by a person that results in a fault.
- Failure - a failure is said to occur in a system when the system's environment observes an output from the system that does not conform to its specification.
- Failure is the run-time manifestation of a fault.
- A fault may or may not lead to failure.

Test Coverage (<http://www.bullseye.com/coverage.html>)

- **Statement coverage** (each statement)
- **Decision coverage** (each boolean expression)
- **Condition coverage** (each boolean sub-expression)
 - Better sensitivity
 - Full condition coverage does not guarantee full decision coverage
- **Multiple condition coverage** (every possible combination of all conditions in a decision)
 - Full multiple condition coverage does not guarantee full decision coverage
- **Condition/decision coverage**
- **Path coverage**
- **Data flow coverage** (def-use path)
- **Loop Coverage** (0, 1, 2+) or (1, 2+)

Sample Code

```
public class Account {  
    private Money balance = new Money(0);  
  
    public Money withdraw(Money amount) {  
        if (balance >= amount) {  
            if (amount >= 0) {  
                balance = balance - amount;  
                return amount;  
            } else {  
                return 0;  
            }  
        } else {  
            return 0;  
        }  
    }  
};
```

- **Full statement coverage**
 - Withdraw \$50 from an Account with \$100
 - Withdraw \$50 from an Account with \$10
 - Withdraw \$-50 from an Account with \$10
- **Full decision coverage**
 - The same test cases

Another example

```
int* p = NULL;  
if (condition)  
    p = &variable;  
*p = 123;
```


Another Example

```
public class Class {
    public void Func(int a, int b)    {
        if (a>10 || b>10) {
            printf("statement 1\n");
        } else {
            printf("statement 2\n");
        }
        if (a>b && b>30) {
            printf("statement 3\n");
        } else {
            printf("statement 4\n");
        }
        if (a<100) {
            printf("statement 5\n");
        }
    }
};
```

- Full statement coverage
 - [a=50, b=2]
 - [a=1, b=2]
 - ?
- Full decision coverage
 - ?
- Full condition coverage
 - ?
- Full multiple condition coverage
 - ?
- Full condition/decision coverage
 - ?
- Full Path coverage
 - ?

Data Flow Testing

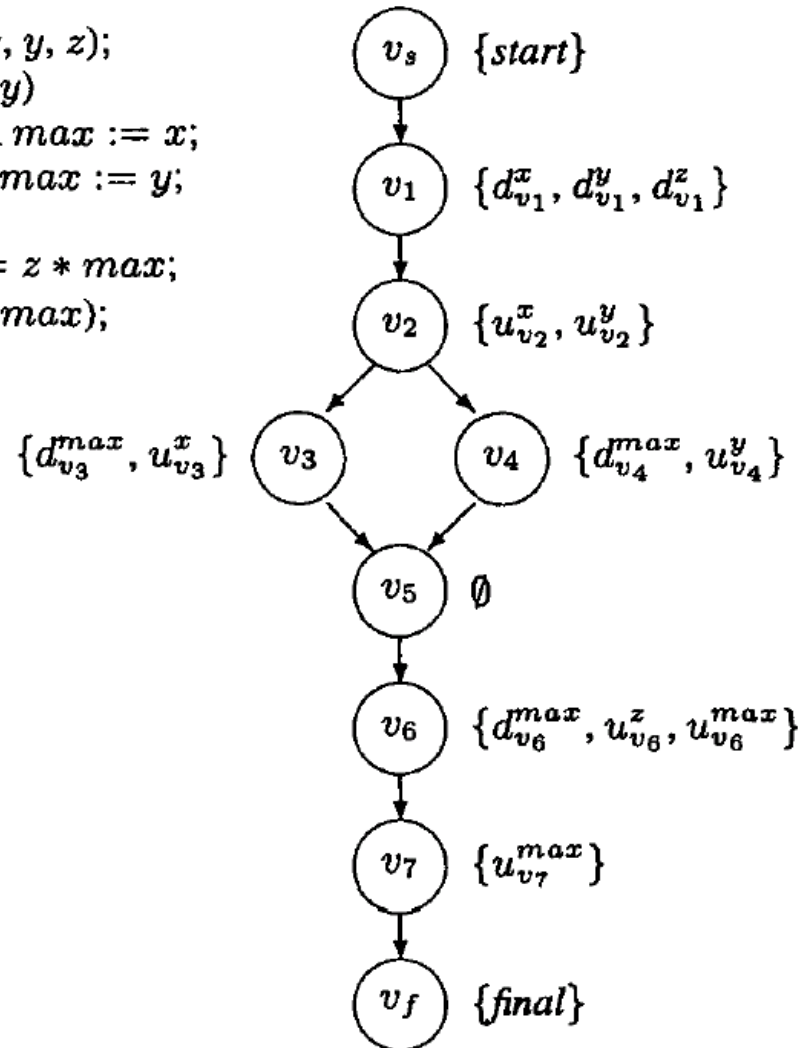
- Definition occurrence
- Use occurrence
 - Computational use occurrence
 - Predicate use occurrence
- Def-use pair
- All definitions criterion
- All uses criterion
- All definition-use-paths criterion

Def-Use Pairs

```

v1: input(x, y, z);
v2: if (x > y)
v3:   then max := x;
v4:   else max := y;
v5: endif
v6: max := z * max;
v7: output(max);

```



Equivalence Classes

- **An equivalence class is a set of input, state, or output values for which an object is supposed to behave similarly.**

Equivalence Classes

- **Valid Data**

- User supplied commands
- Responses to systems prompts
- File names
- Computational data
 - Physical parameters
 - Bounding values
 - Initiation values
- Output data formatting commands
- Responses to error messages
- Graphical data (e.g., mouse clicks)

Equivalence Classes

● Invalid Data

- Data outside bounds of the program
- Physically impossible data
- Proper value supplied in wrong place

Two Types of Questions to Answer

- **1. Does the system work for the most common scenarios or situations?**
- **2. Does the system behave as expected under all possible scenarios and in all situations?**

Two Types of Tests

- **Initial Validation Tests** (for each build)
 - “smoke test”:
 - A rudimentary form of testing applied to electronic equipment following repair or reconfiguration, in which power is applied and the tester checks for sparks, smoke, or other dramatic signs of fundamental failure.
 - By extension, the first run of a piece of software after construction or a critical change.
- **Acceptance Tests** (for milestone builds, internal release builds, alpha builds, beta builds, release candidates, or official releases)

Scenarios in Test

- **Smoke Test: One or a few common scenarios**
- **Differences with scenarios in use case descriptions:**
 - **Scenarios in tests much have concrete data.**
 - **Expected outputs must be described.**

Concepts

- **Test:** Verify the result from implementation by executing each build (including internal and intermediate ones) with pre-defined inputs
- ***Test model:*** a collection of *test cases*, *test procedures*, and *test components*.
 - Test model
 - Test case
 - Test procedure
 - Test component

Concepts

- **Test case**

- **Input**
- **Expected result**
- **Execution conditions**
 - **Environmental conditions**
 - **Activities that take place simultaneously**

Concepts

- **Test procedure**

- A test procedure specifies how to perform one or several test cases or parts of them.

- **Test component**

- A test component automates one or several test procedures or parts of them.
- A test component enables testing of incomplete program units.
- Test components are also sometimes called test drivers, test harnesses, and test scripts.

The Test Workflow

- **Plan Test**
- **Design Test**
- **Implement Tests**
- **Perform Integration Tests**
- **Perform System Tests**
- **Evaluate Tests**

Plan Test

- Testing strategy.
- Required level of test code coverage.
- Success criteria.
- Scheduling the testing effort.
- Example System Test Strategy:
 - System Test Strategy for the Last Iteration in the Elaboration Phase
 - At least 75% of the test should be automated, and the rest should be manual. Each subject use case will be tested for its normal flow and three alternative flows.
 - Success criteria: 90% of test cases passed. No medium-to-high priority defects unresolved.

Design Test

- Design Integration Test Cases
- Design System Test Cases
- Design Regression Test Cases
- Identify and Structure *Test Procedures*

Design Integration Test Cases

- **Specification (Black Box) Tests for Use Cases**
(based on the use case descriptions in the requirement model)
- **Structure (White Box) Tests for Use Cases**
(based on the use case realization in the analysis and design model) (Important clues: interaction diagrams)
- **Consider: combinations of actor input, output, and system start state**
- **Important: focus on new features**

Design System Test Cases

- **Test the system functions as a whole**
- **Different configurations**
- **Different levels of system loads**
- **Various numbers of actors**
- **Different sizes of the database**

Design System Test Cases

- Consider combinations of use cases that:
- Are required to function in parallel
- Are likely to be performed in parallel
- Are likely to influence each other
- Involve multiple processes

System tests

- **Installation tests**
- **Configuration tests**
- **Negative tests** (try to cause the system to fail in order to reveal its weakness)
- **Stress tests**

An Example of Negative Test

- **Cyber Security**

- **ICSE2003 Invited Paper**
- **By Richard Kemmerer**
- <http://www.cs.ucsb.edu/~kemm/>
- <http://eecs.oregonstate.edu/icse2003/events/fosp.html#topic3>
- <http://www.acm.org/dl>
- **A real-world example of negative tests of on-line banking system security**

Example: Microsoft EEC (Enterprise Engineering Center)



Design Regression Test Cases

- What is regression test?
- Why is regression test important?
- Can any test case be used for regression testing?
 - Some test cases are more effective and/or efficient than others.

Regression Testing

```
input(x,y);  
if (x > y)  
    then output(x);  
    else output(y);
```

Test case 1: (2,1)

Test case 2: (2,2)

```
input(x,y);  
if (x > y)  
    then output(x);  
    else  
        if (y>0)  
            then output(y);  
            else output(0);
```

Test case 1: (2,1)

***Test case 2: (2,2)**

***Test case 3: (-2,-1)**

Identify and Structure Test Procedures

- Reuse existing test procedures as much as possible.
- Limit test procedures' dependency on subsystems.

Implement Tests

- Create test components to automate test procedures.
- Automation is optional.
- Not all test procedures can be automated.
- Not all automateable test procedures should be automated.
- Good candidate: some regression tests.
- Test automation tools: VisualTest (Microsoft/Rational/IBM), SilkTest (Segue), QARunner, ...

Perform Integration Test

- Perform the tests relevant to the current build
- Compare the test results with the expected results
- Report defects

Perform System Test

- Should not wait until the release build
- Probably should not perform on the first build
- Should perform for several iterations

Evaluate Tests

- Evaluate the testing efforts within an iteration.
- Refer to the goals outlined in the test plan
- Prepare metrics
 - **Test completeness**
(% of test cases completed; % of code covered)
 - **Reliability** (by analyzing and predicting defect trends)
(defects over time; successful tests over time; MTBF)

Test Checklist

- **Test strategy**
- **Integration tests**
 - **Black box tests for use cases**
 - Equivalence classes (for input, output and state)
 - Test cases, procedures and results
 - **White box tests for use cases**
 - List of interesting paths (most commonly followed, most critical, least known, high risk, etc.) through the sequence diagram
 - Test cases, procedures and results
- **System tests**
 - **Installation tests**
 - **Configuration tests**
 - **Negative tests**
 - **Stress tests**
 - **Test cases, procedures and results**
- **Test evaluation**
 - **Testing completeness**
 - **Reliability**