

**Cornell University**  
**Computing and Information Science**

---

**CS 5150 Software Engineering**  
**20. Verification, Testing, and Bugs**

William Y. Arms

# Building Reliable Systems: Two Principles

---

## For a software system to be reliable:

- Each stage of development must be done well, with incremental verification and testing.
- Testing and correction do not ensure quality, but reliable systems are not possible without thorough testing.

# Static and Dynamic Verification

---

## Static verification:

Techniques of verification that do not include execution of the software.

- May be manual or use computer tools.

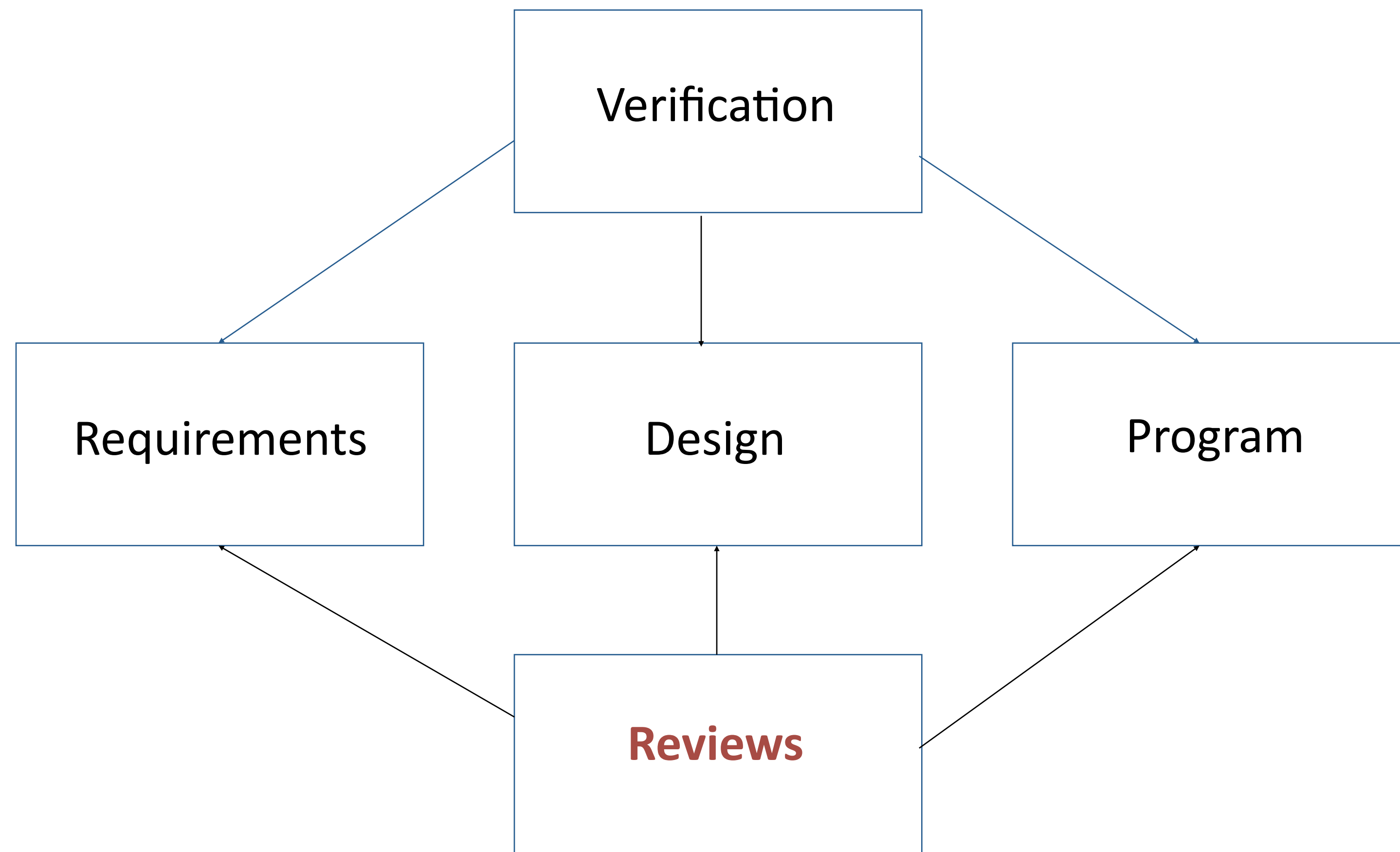
## Dynamic verification:

- **Testing** the software with trial data.
- **Debugging** to remove errors.

# Static Verification: Reviews

---

**Reviews** are a form of static verification that is carried out throughout the software development process.



# Reviews

---

Reviews are a fundamental part of good software development

## Concept

Colleagues review each other's work:

- can be applied to any stage of software development, but particularly valuable to review **program design** or **code**
- can be formal or informal

## Preparation

The developer(s) provides colleagues with documentation (e.g., models, specifications, or design), or code listing.

Participants study the materials in advance.

## Meeting

The developer leads the reviewers through the materials, describing what each section does and encouraging questions.

# The Review Meeting

---

A review is a structured meeting

## Participants and their roles:

- **Developer(s):** person(s) whose work is being reviewed
- **Moderator:** ensures that the meeting moves ahead steadily
- **Scribe:** records discussion in a constructive manner
- **Interested parties:** other developers on the same project
- **Outside experts:** knowledgeable people who are not working on this project
- **Client:** representatives of the client who are knowledgeable about this part of the process

# Benefits of Reviews

---

## Benefits:

- Extra eyes spot mistakes, suggest improvements
- Colleagues share expertise; helps with training
- Incompatibilities between components can be identified
- Gives developers an incentive to tidy loose ends
- Helps scheduling and management control

# Successful Reviews

---

## To make a review a success:

- Senior team members must show leadership
- Good reviews require good preparation by everybody
- Everybody must be helpful, not threatening

Allow plenty of time and be prepared to continue on another day.



# Static Verification: Pair Design and Pair Programming

---

**Concept: achieve benefits of review by shared development**

Two people work together as a team:

- design and/or coding
- testing and system integration
- documentation and hand-over

Benefits include:

- two people create better software with fewer mistakes
- cross training

Many software houses report excellent productivity

# Static Verification: Program Inspections

---

## **Formal program reviews whose objective is to detect faults**

- Code is read or reviewed line by line
- 150 to 250 lines of code in 2 hour meeting
- Use checklist of common errors
- Requires team commitment, e.g., trained leaders

So effective that it is claimed that it can replace unit testing

# Static Verification: Analysis Tools

---

**Program analyzers** scan the source of a program for **possible** errors and anomalies.

- **Control flow:** Loops with multiple exit or entry points
- **Data use:** Undeclared or uninitialized variables, unused variables, multiple assignments, array bounds
- **Interface faults:** Parameter mismatches, non-use of functions results, uncalled procedures
- **Storage management:** Unassigned pointers, pointer arithmetic

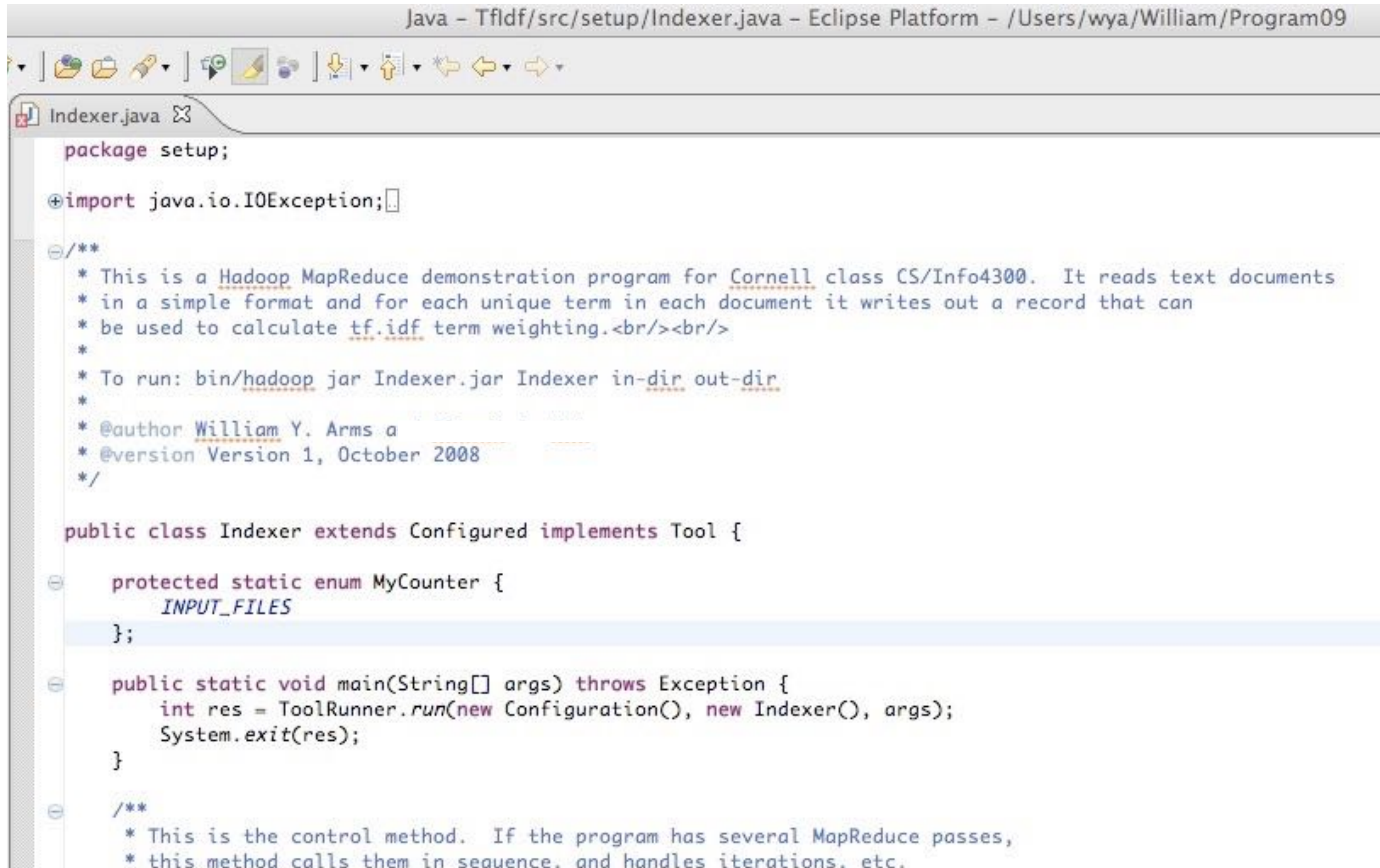
# Static Analysis Tools (continued)

---

## Static analysis tools

- **Cross-reference table:** Shows every use of a variable, procedure, object, etc.
- **Information flow analysis:** Identifies input variables on which an output depends.
- **Path analysis:** Identifies all possible paths through the program.

# Static Analysis Tools in Programming Toolkits



```
Java - TfIdf/src/setup/Indexer.java - Eclipse Platform - /Users/wya/William/Program09

Indexer.java

package setup;

import java.io.IOException;

/**
 * This is a Hadoop MapReduce demonstration program for Cornell class CS/Info4300. It reads text documents
 * in a simple format and for each unique term in each document it writes out a record that can
 * be used to calculate tf.idf term weighting.<br/><br/>
 *
 * To run: bin/hadoop jar Indexer.jar Indexer in-dir out-dir
 *
 * @author William Y. Arms a
 * @version Version 1, October 2008
 */

public class Indexer extends Configured implements Tool {

    protected static enum MyCounter {
        INPUT_FILES
    };

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new Indexer(), args);
        System.exit(res);
    }

    /**
     * This is the control method. If the program has several MapReduce passes,
     * this method calls them in sequence, and handles iterations, etc.
     */
}
```



# Static Analysis Tools in Programming Toolkits

```
public void map(LongWritable key, Text value, OutputCollector<Text, WordinDoc> output, Reporter rep1)

    /* Read the input document one token at a time and build the wordTree */

    String doc = "";
    String line = value.toString();

    StringTokenizer itr = new StringTokenizer(line);
    rep1.incrCounter(MyCounter.INPUT_FILES, 1);

    if (itr.hasMoreTokens()) doc = itr.nextToken();

    String word;
    int counter = 0;
    int maxF = 1;
    int lengthSq = 0;

    while (itr.hasMoreTokens() && counter <= MAXWORDS) {
        word = itr.nextToken().toLowerCase().replaceAll("[^a-z]", "");

        if (word.length() > 1 && !stopwords.contains(word)){
            WordinDoc tempNode = new WordinDoc();
            int tempF = 1;
        }
    }

    // Doc ID
    // First word of d
    // Next word from
    // Count of words
    // Max frequency o
    // Length of docum
    // Get next word a
    // Ignore one char
```

Problems @ Javadoc Declaration

1 error, 15 warnings, 0 others

Description	Resource
Errors (1 item)	
Warnings (15 items)	
The field GraphClean.MapClean.fromURL is never read locally	GraphClean.java
The field GraphClean.MapClean.hashFromURL is never read locally	GraphClean.java
The field GraphClean.MapClean.hashToURL is never read locally	GraphClean.java

# Dynamic Verification: Stages of Testing

---

**Testing is most effective if divided into stages**

User interface testing (carried out separately)

**Unit testing**

- unit test

**System testing**

- integration test
- function test
- performance test
- installation test

Acceptance testing (carried out separately)

# Testing Strategies

---

## Bottom-up testing

Each unit is tested with its own test environment. Used for all systems.

## Top-down testing

Large components are tested with dummy stubs. Particularly useful for:

- user interfaces
- work-flow
- client and management demonstrations

## Stress testing

Tests the system at and beyond its limits. Particularly useful for:

- real-time systems
- transaction processing

Most systems require a combination of all three strategies.



# Methods of Testing

---

## Open box testing

Testing is carried out by people who know the internals of what they are testing.

**Example:** Tick marks in a graphing package

## Closed box testing

Testing is carried out by people who do not know the internals of what they are testing.

**Example:** Educational demonstration that was not foolproof

# Testing: Unit Testing

---

- Tests on small sections of a system, e.g., a single class
- Emphasis is on accuracy of actual code against specification
- Test data is usually chosen by developer(s) based on their understanding of specification and knowledge of the unit
- Can be at various levels of granularity
- **Can be open box or closed box:** by the developer(s) of the unit or by special testers

If unit testing is not thorough, system testing becomes almost impossible. If you are working on a project that is behind schedule, do not rush the unit testing.

# Testing: System and Sub-System Testing

---

- Tests on components or complete system, combining units that have already been thoroughly tested
- Emphasis on integration and interfaces
- Trial data that is typical of the actual data, and/or stresses the boundaries of the system, e.g., failures, restart
- Carried out systematically, adding components until the entire system is assembled
- **Can be open or closed box:** by development team or by special testers

System testing is finished fastest if each component is completely debugged before assembling the next

# Dynamic Verification: Test Design

---

**Testing can never prove that a system is correct.**

It can only show that either (a) a system is correct in a single case, or (b) that it has an error.

- The objective of testing is to find errors or demonstrate that program is correct in specific instances.
- Testing is never comprehensive.
- Testing is expensive.

# Test Cases

---

**Test cases** are specific tests that are chosen because they **are likely to find specific problems**.

Test cases are chosen to balance expense against chance of finding serious errors.

- Cases chosen by the **development team** are effective in testing known vulnerable areas.
- Cases chosen by **experienced outsiders and clients** will be effective in finding gaps left by the developers.
- Cases chosen by **inexperienced users** will find other errors.

# Variations in Test Sets

---

A **test suite** is the set of all test cases that apply to a system or component of a system.

When running tests, there are some errors that occur only under certain circumstances, e.g., when certain other software is running on the same machine, when tasks are scheduled in specific sequences, or with unusual data sets, etc.

Therefore it is customary for each test run to vary some of the test cases systematically, and to change the order in which the tests are made, etc.

# Incremental Testing (e.g., Daily Testing)

---

## Spiral development and incremental testing

- Create a first iteration that has the structure of the final system and some basic functionality.
- Create an initial set of test cases.
- Check-in changes to the system on a daily basis, rebuild entire system daily.
- Run a comprehensive set of test cases daily, identify and deal with any new errors.
- Add new test cases continually.

Many large software houses, e.g., Microsoft, follow this procedure with a daily build of the entire system and comprehensive sets of test cases. For a really big system this may require hundreds or even thousands of test computers and a very large staff of testers.

# Dynamic Verification: Regression Testing

---

When software is modified, **regression testing** is used to check that modifications behave as intended and do not adversely affect the behavior of unmodified code.

After every change, however small, rerun the entire testing suite.



# Regression Testing: Program Testing

---

- Collect a **suite of test cases**, each with its expected behavior.
- Create scripts to run all test cases and compare with expected behavior. (Scripts may be automatic or have human interaction.)
- When a change is made to the system, however small (e.g., a bug is fixed), add a new test case that illustrates the change (e.g., a test case that revealed the bug).
- Before releasing the changed code, **rerun the entire test suite**.

# Incremental Testing: an Example

---

## Example

A graphics package consisting of a pre-processor, a runtime package (set of classes), and several device drivers.

## Starting point

A prototype with a good overall structure, and most of the functionality, but hastily coded and not robust.

## Approach

On a daily cycle:

- Design and code one small part of the package (e.g., an interface, a class, a dummy stub, an algorithm within a class.)
- Integrate into prototype.
- Create additional test cases if needed.
- Regression test.

# Documentation of Testing

---

Every project needs a **test plan** that documents the testing procedures for thoroughness, visibility, and for **future maintenance**.

The test plan should include:

- Description of testing approach.
- List of test cases and related bugs.
- Procedures for running the tests.
- Test analysis report.

# Fixing Bugs

---

## Isolate the bug

- Intermittent --> repeatable
- Complex example --> simple example

## Understand the bug and its context

- Root cause
- Dependencies
- Structural interactions

## Fix the bug

- Design changes
- Documentation changes
- Code changes
- Create new test case

# Moving the Bugs Around

---

## Fixing bugs is an error-prone process

- When you fix a bug, fix its environment.
- Bug fixes need static and dynamic testing.
- Repeat all tests that have the slightest relevance (regression testing).

## Bugs have a habit of returning

When a bug is fixed, add the failure case to the test suite for future regression testing.

## Persistence

Most people work around a problem. The best people track down the root cause and fix it for ever!

# Difficult Bugs

---

Some bugs may be extremely difficult to track down and isolate. This is particularly true of intermittent failures.

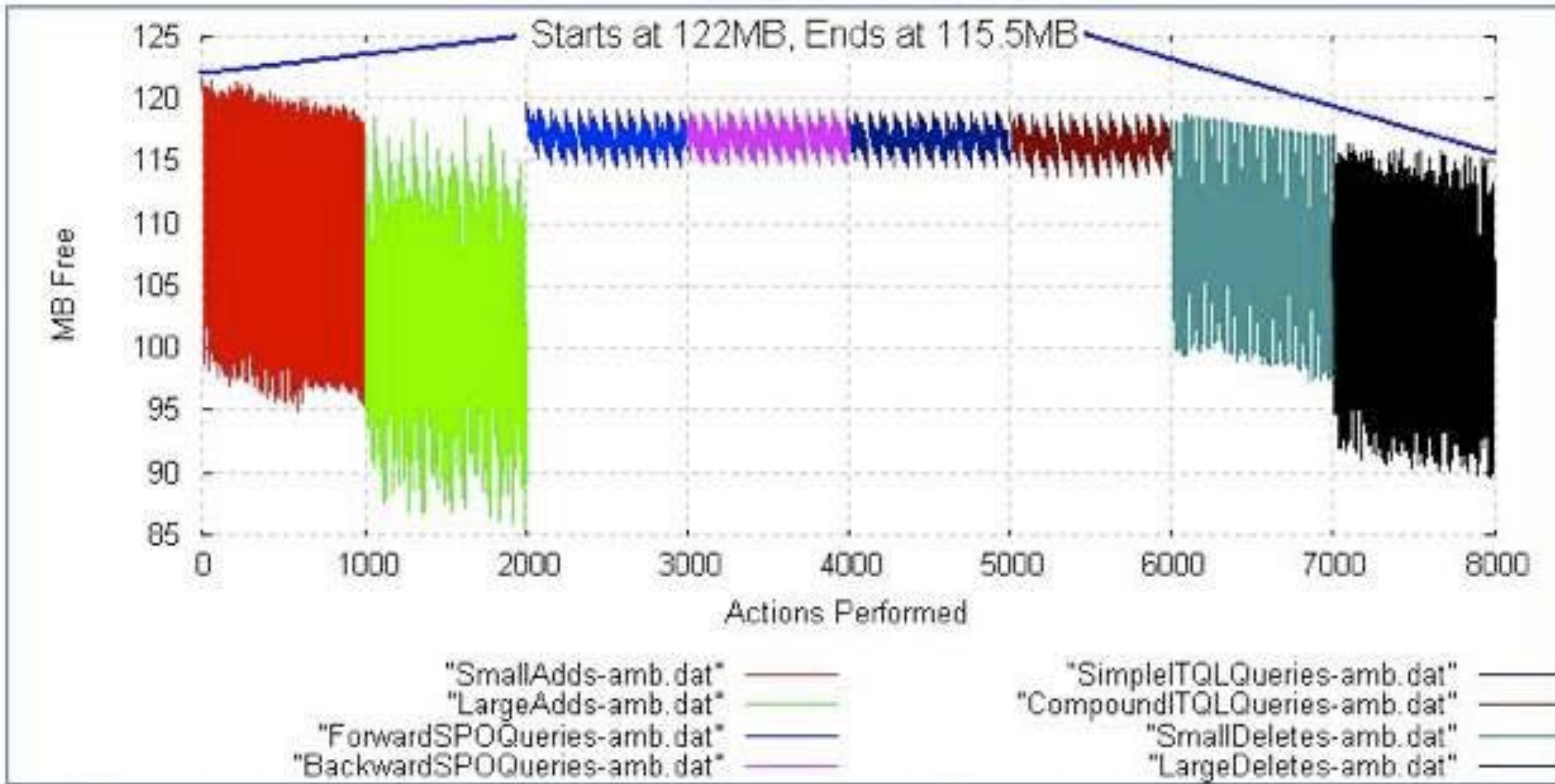
- A large central computer stops a few times every month with no dump or other diagnostic.
- A database load dies after running for several days with no diagnostics.
- An image processing system runs correctly, but uses huge amounts of memory.

Such problems may require months of effort to track down.

For a fictional example, see: Ellen Ullman, *The Bug: a Novel*, (Doubleday 2003).

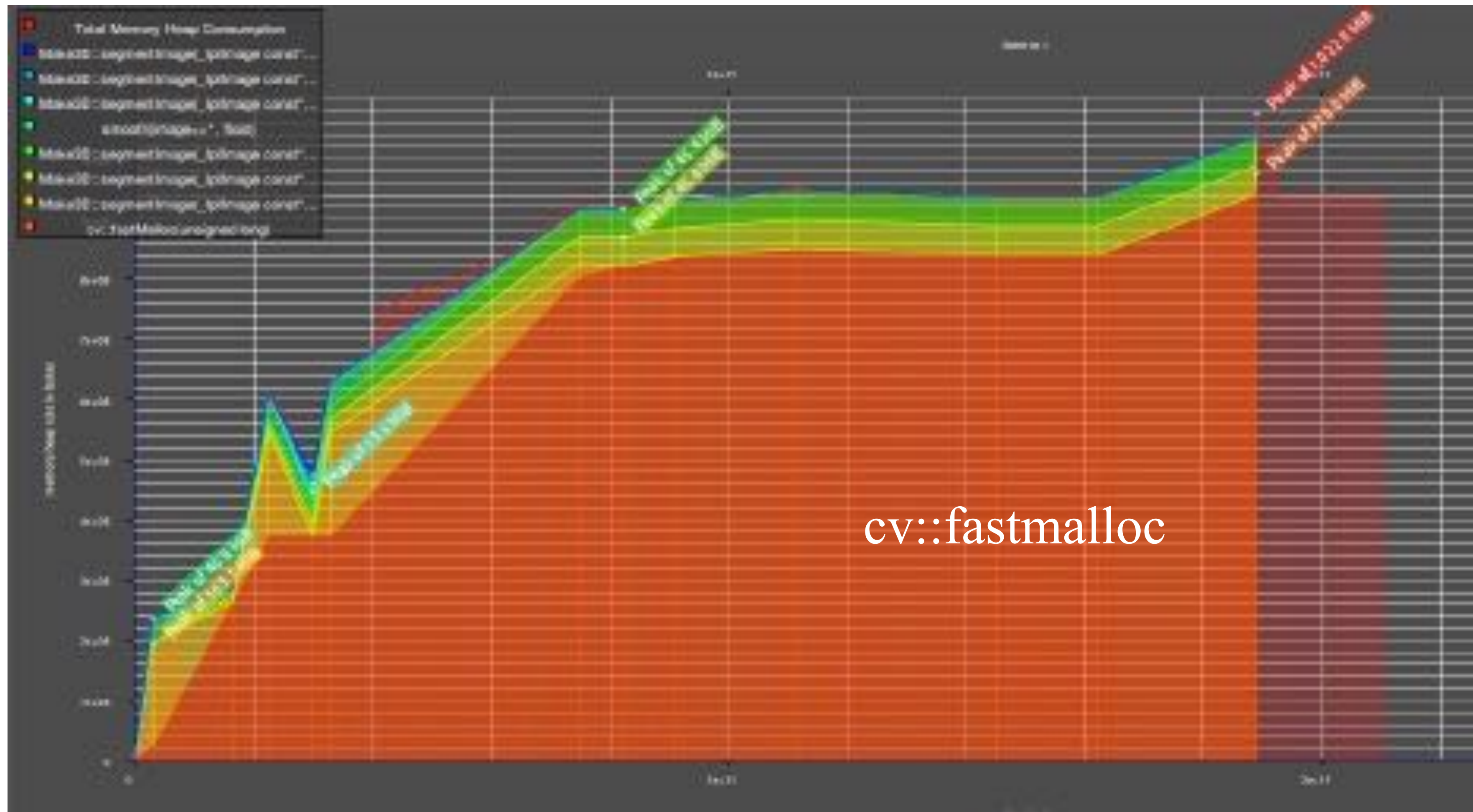


# Tracking Down a Difficult Bugs: The Heisenbug





# Tracking Down a Difficult Bug: Make3D



## Memory usage by function



# Bugs in System Software

---

**Even system software from good manufacturers may contain bugs:**

- Built-in function in Fortran run-time environment ( $e^0 = 0$ )
- The string-to-number function that was very slow with integers
- The preload system with the memory leak

# Bugs in Hardware

---

**Three times in my career I have encountered hardware bugs:**

- The film plotter with the missing byte (1:1023)
- Microcode for virtual memory management
- The Sun page fault

Each problem was actually a bug in embedded software/firmware

# When Fixing a Bug Creates a Problem for Customers

---

Sometimes customers will build applications that rely upon a bug. Fixing the bug will break the applications.

- The graphics package with rotation about the Z-axis in the wrong direction.
- An application crashes with an emulator, even though the emulator is bug free. (Compensating bug problem.)
- The 3-pixel rendering problem with Internet Explorer.

With each of these bugs the code was easy to fix, but releasing it would have caused problems for existing programs.

# A Note on Testing User Interfaces

---

## User interfaces need several categories of testing.

- During the **design phase**, user testing is carried out with trial users to ensure that the design is usable. Design testing is also used to develop graphical elements and to validate the requirements.
- During the **implementation phase**, the user interface goes through the standard steps of unit and system testing to check the reliability of the implementation.
- Finally, **acceptance testing** is carried out with users, on the complete system.