

# Foundations of Software Engineering

Lecture 12 – Testing

Claire Le Goues

# Learning goals

- Define software analysis.
- Reason about QA activities with respect to coverage and coverage/adequacy criteria, both traditional (structural) and non-traditional.
- Conceive of testing as an activity designed to achieve *coverage* along a number of (non-structural!) dimensions.
- Enumerate testing strategies to help evaluate the following quality attributes: usability, reliability, security, robustness (both general and architectural), performance, integration.
- Give tradeoffs and identify when each of those techniques might be useful.

**HOW DO YOU KNOW THAT YOUR  
PROGRAM WORKS?**

# Questions

- How can we ensure a system meets its specification?
- How can we ensure a system meets the needs of its users?
- How can we ensure a system does not behave badly?

# Two kinds of analysis questions

- **Verification:** Does the system meet its specification?
  - i.e. did we build the system correctly?
- **Verification:** are there flaws in design or code?
  - i.e. are there incorrect design or implementation decisions?
- **Validation:** Does the system meet the needs of users?
  - i.e. did we build the right system?
- **Validation:** are there flaws in the specification?
  - i.e., did we do requirements capture incorrectly?

# Definition: software analysis

The **systematic** examination of a software artifact to determine its properties.

Attempting to be comprehensive, as measured by, as examples:  
Test coverage, inspection checklists, exhaustive model checking.

# Definition: software analysis

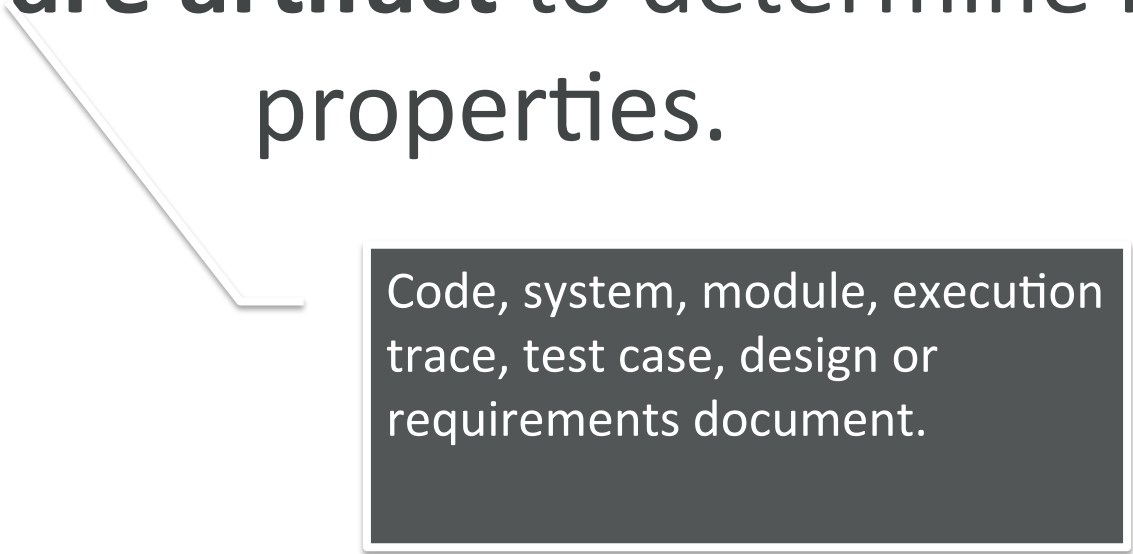
The systematic **examination** of a software artifact to determine its properties.

**Automated:** Regression testing, static analysis, dynamic analysis

**Manual:** Manual testing, inspection, modeling

# Definition: software analysis

The systematic examination of a **software artifact** to determine its properties.




Code, system, module, execution trace, test case, design or requirements document.



# Definition: software analysis

The systematic examination of a software artifact to determine its **properties.**



**Functional:** code correctness  
**Non-functional:** evolvability, safety, maintainability, security, reliability, performance, ...

# VERY IMPORTANT

- *There is no one analysis technique that can perfectly address all quality concerns.*
- Which techniques are appropriate depends on many factors, such as the system in question (and its size/complexity), quality goals, available resources, safety/security requirements, etc etc...

# Principle techniques

- **Dynamic:**
  - **Testing:** Direct execution of code on test data in a controlled environment.
  - **Analysis:** Tools extracting data from test runs.
- **Static:**
  - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
  - **Analysis:** Tools reasoning about the program without executing it.

# One slide with a bunch of ideas you should remember from 15-214

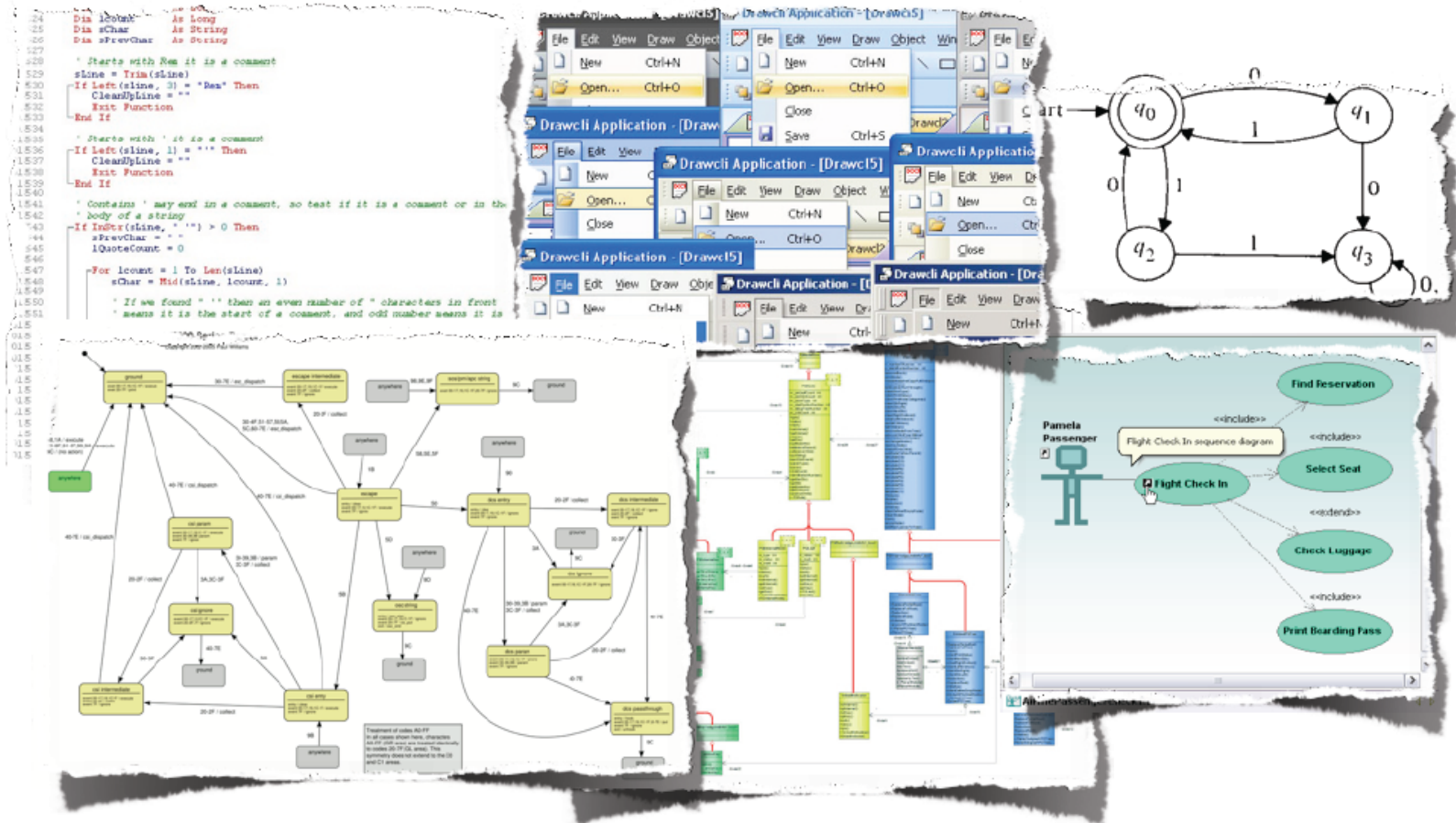
- Verification vs. testing
- Black box
- TDD
- Testing harness, scaffolding, stubs
- Unit testing/Junit
- Nightly vs. smoke tests
- **Coverage**



# “Traditional” coverage

- Statement
- Branch
- Function
- Path (?)
- MC/DC

# We can measure coverage on almost anything



A. Zeller, Testing and Debugging Advanced course, 2010

# We can measure coverage on almost anything

- Common adequacy criteria for testing approximate full “coverage” of the program execution or specification space.
- Measures the extent to which a given verification activity has achieved its objectives; approximates adequacy of the activity.
  - *Can be applied to any verification activity, although most frequently applied to testing.*
- Expressed as a ratio of the measured items executed or evaluated at least once to the total number of measured items; usually expressed as a percentage.

# Covering quality requirements

- How might we test the following?
  - Web-application performance
  - Scalability of application for millions of users
  - Concurrency in a multiuser client-server application
  - Usability of the UI
  - Security of the handled data
- What are the coverage criteria we can apply to those qualities?



# Principle techniques

- **Dynamic:**
  - **Testing:** Direct execution of code on test data in a controlled environment.
  - **Analysis:** Tools extracting data from test runs.
- **Static:**
  - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
  - **Analysis:** Tools reasoning about the program without executing it.

# What is testing?

- *Direct execution of code on test data in a controlled environment*
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
  - Defect testing: reveal failures.
- Other goals:
  - Clarify specification: Testing can demonstrate inconsistency; either spec or program could be wrong
  - Learn about program: How does it behave under various conditions? Feedback to rest of team goes beyond bugs
  - Verify contract, including customer, legal, standards

**"Testing shows the presence,  
not the absence of bugs  
Edsger W. Dijkstra 1969**

# What are we covering?

- Program/system functionality:
  - Execution space.
  - Input or requirements space.
  - Use cases.
  - Defect space.
- The expected user experience (usability).
- The expected performance envelope (performance, reliability, robustness, integration).

# Regression testing (redux)

- What is “covered” by a set of regression tests?
- Why do we do regression testing?
- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - **Code1 + TestSet**  $\diamond$  **TestResults1**
    - **Code2 + TestSet**  $\diamond$  **TestResults2**
  - As code evolves, compare **TestResults1** with **TestResults2**, etc.
- Benefits:
  - Ensure bug fixes remain in place and bugs do not reappear.
  - Reduces reliance on specifications, as **<TestSet,TestResults1>** acts as one.

# Integration: object protocols

- Covers the space of possible API calls, or program “conceptual states.”
- Develop test cases that involve representative sequence of operations on objects
  - Example: Dictionary structure: Create, AddEntry\*, Lookup, ModifyEntry\*, DeleteEntry, Lookup, Destroy
  - Example: IO Stream: Open, Read, Read, Close, Read, Open, Write, Read, Close, Close
  - Test concurrent access from multiple threads
    - Example: FIFO queue for events, logging, etc.



- Approach
  - Develop representative sequences – based on use cases, scenarios, profiles
  - Randomly generate call sequences
- Also useful for protocol interactions within distributed designs.

# Practices – integration testing

- Do incremental integration testing
  - Test several modules together
  - Still need scaffolding for modules not under test
- Avoid “big bang” integrations
  - Going directly from unit tests to whole program tests
  - Likely to have many big issues
  - Hard to identify which component causes each
- Test interactions between modules
  - Ultimately leads to end-to-end system test



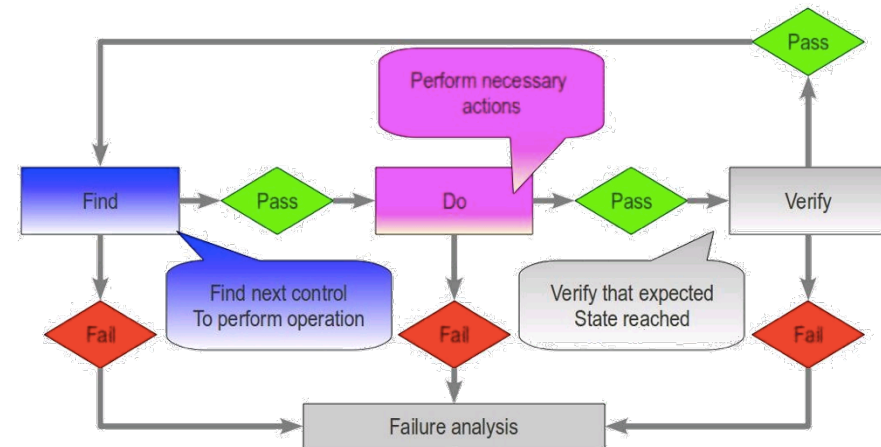
# What are we covering?

- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- **The expected user experience (usability).**
  - **GUI testing, A/B testing**
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing



# Automating GUI/Web Testing (from 214)

- First: why is this hard?
- Capture and Replay Strategy
  - mouse actions
  - system events
- Test Scripts: (click on button labeled "Start" expect value X in field Y)
- Lots of tools and frameworks
  - e.g. JUnit + Jemmy for Java/Swing
- (Avoid load on GUI testing by separating model from GUI)



# Usability: A/B testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.

# Example

- A company sends an advertising email to its customer database, varying the photograph used in the ad...

# Example: group A (99% of users)



- Act now!  
Sale ends soon!

# Example: group B (1%)



- Act now!  
Sale ends soon!

**HOW DOES THIS TECHNIQUE  
GENERALIZE, ESPECIALLY TECHNICALLY?**

# What are we covering?

- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- The expected user experience (usability).
- **The expected performance envelope (performance, reliability, robustness, integration).**
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

# (214 review) Random testing

- Select inputs independently at random from the program's input domain:
  - Identify the input domain of the program.
  - Map random numbers to that input domain.
  - Select inputs from the input domain according to some probability distribution.
  - Determine if the program achieves the appropriate outputs on those inputs.
- Random testing can provide probabilistic guarantees about the likely faultiness of the program.
  - E.g., Random testing using  $\sim 23,000$  inputs without failure ( $N = 23,000$ ) establishes that the program will not fail more than one time in  $10,000$  ( $F = 10^4$ ), with a confidence of 90% ( $C = 0.9$ ).



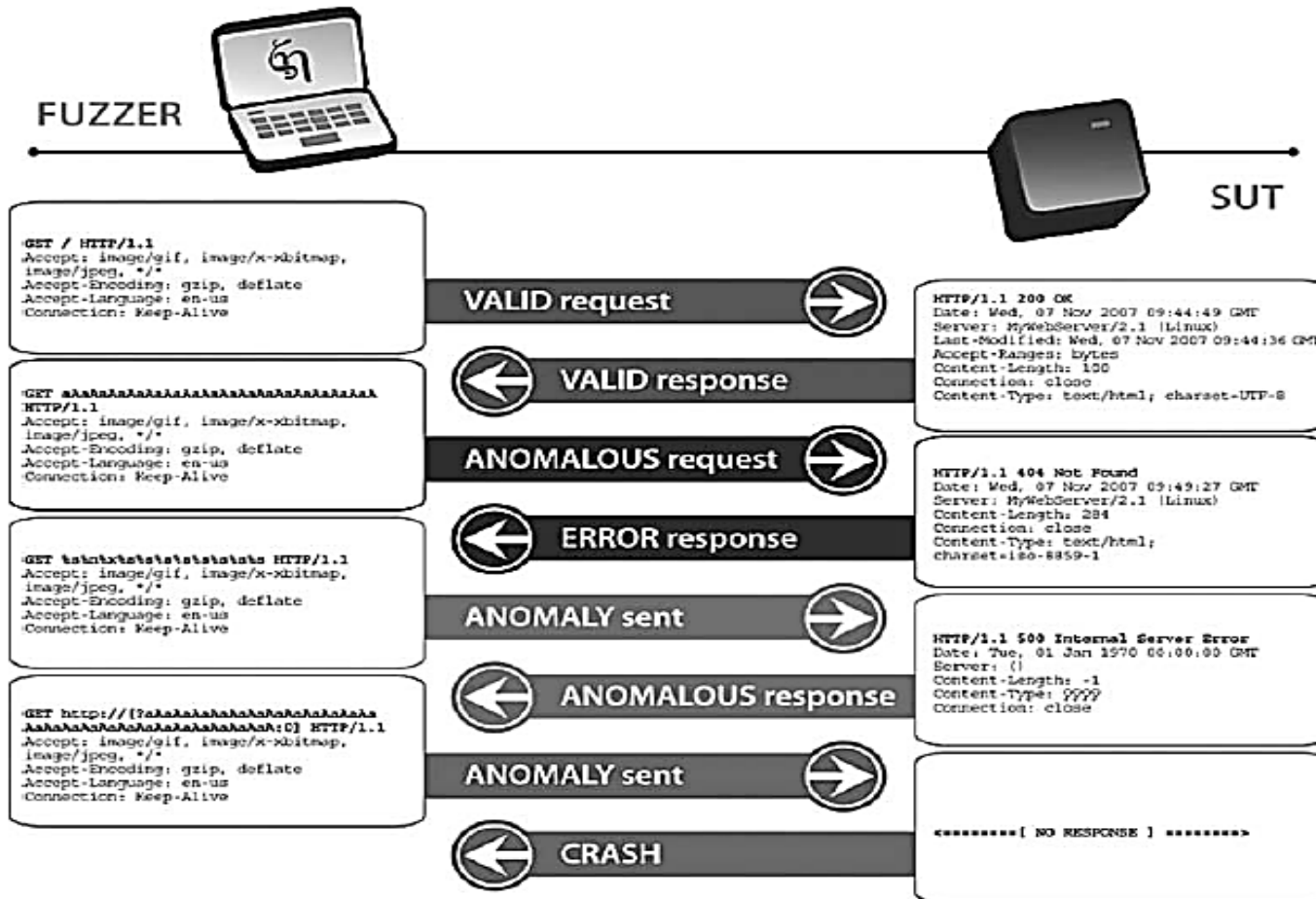
# Reliability: Fuzz testing

- Negative software testing method that feeds malformed and unexpected input data to a program, device, or system with the purpose of finding security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior (A. Takanen et al, Fuzzing for Software Security Testing and Quality Assurance, 2008)
- Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called **fuzzers**.

# Types of faults found

- Pointer/array errors
- Not checking return codes
- Invalid/out of boundary data
- Data corruption
- Signed characters
- Race conditions
- Undocumented features
- ...Possible tradeoffs?

# Fuzzing process



# Fuzzing approaches

- **Generic:** crude, random corruption of valid data without any regard to the data format.
- **Pattern-based:** modify random data to conform to particular patterns. For example, byte values alternating between a value in the ASCII range and zero to “look like” Unicode.
- **Intelligent:** uses semi-valid data (that may pass a parser/sanity checker’s initial line of defense); requires understanding the underlying data format. For example, fuzzing the compression ratio for image formats, or fuzz PDF header or cross-reference table values.
- **Large Volume:** fuzz tests at large scale. The Microsoft Security Development Lifecycle methodology recommends a minimum of 100,000 data fuzzed files.
- **Exploit variant:** vary a known exploitative input to take advantage of the same attack vector with a different input; good for evaluating the quality of a security patch.

# Assessing robustness

- Test erroneous inputs and boundary cases
  - Assess consequences of misuse or other failure to achieve preconditions

```
public static int bsrch (int[] a, int key) {
```

Java

```
    int low = 0;
```

```
    int high = a.length - 1;
```

```
    while (true) {
```

```
        if ( low > high ) return -(low+1);
```

```
        int mid = (low+high) / 2;
```

```
        if ( a[mid] < key ) low = mid + 1;
```

```
        else if ( a[mid] > key ) high = mid - 1;
```

```
        else return mid;
```

```
    }
```

```
}
```

What if the array reference a is null?

# Assessing robustness

- Test erroneous inputs and boundary cases
  - Assess consequences of misuse or other failure to achieve preconditions
  - Bad use of API
  - Bad input data, files (e.g., corrupted), or communication connections
  - Buffer overflow (security exploit) is a robustness failure (deliberate interface misuse)
- Test apparatus needs to be able to catch and recover from crashes and other hard errors (e.g., Ballista tool)
  - Sometimes multiple inputs need to be at/beyond boundaries
- The question of responsibility
  - Is there external assurance that preconditions will be respected?
  - This is a design commitment that must be considered explicitly.

`a[mid]`

What if the array reference `a` is null?

# Stress testing

- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.

# Soak testing

- **Problem:** A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we'll talk about that later).
- **Soak testing:** testing a system with a significant load over a significant period of time (*positive*).
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.





# Completeness?

- Statistical thresholds
  - Defects reported/repaired
  - Relative proportion of defect kinds
  - Predictors on “going gold”
- Coverage criterion
  - E.g., 100% coverage required for avionics software
  - Distorts the software
  - Matrix: Map test cases to requirements use cases
- Can look at historical data
  - Within an organization, can compare across projects; Develop expectations and predictors
  - (More difficult across organizations, due to difficulty of commensurability, E.g., telecon switches vs. consumer software)
- Rule of thumb: when error detection rate drops (implies diminishing returns for testing investment).
- Most common: Run out of time or money

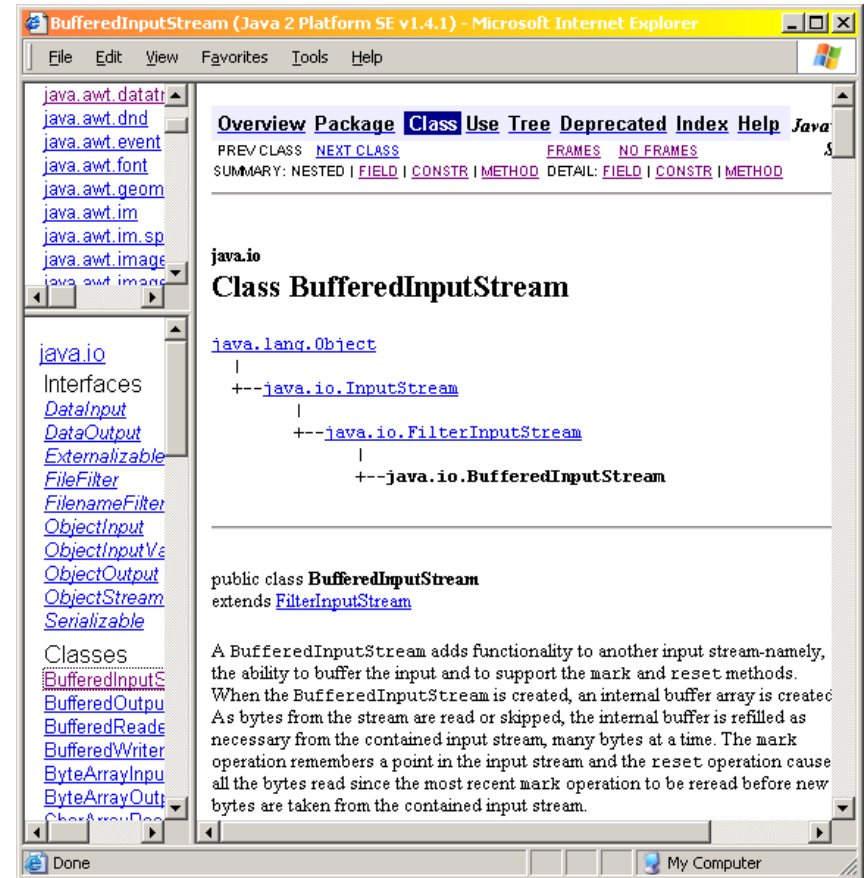
**QUICK ASIDE ON BUG FIXING AND THE  
TRICKY RELATIONSHIP BETWEEN  
DESIGN, INTENT, IMPLEMENTATION,  
AND YOUR CRANKY USERS...**

# Race conditions

- Races can occur when:
  - Multiple threads of control access shared data
  - Data gets corrupted when internal integrity assumptions are violated.
- How we protect against races
  - Use “lock” objects that enable access by one thread at a time
    - E.g., event dispatch
    - A language feature in Java, Ada95, etc.
  - Follow a thread discipline in which only one thread can access critical data (Common in GUI APIs e.g., graphical toolkit redraw)
- *Issue*: Basically the hardest bugs to find, fix, and protect against.
  - Why?

# java.io.BufferedInputStream

- Buffering wrapper for unbuffered stream input: **read**, **close**, reset, skip, mark, etc.
- JDK < 1.2: Race condition between methods **read** and **close**: interleaved execution could cause **read** to throw `NullPointerException`
  - But not always; concurrency → non-deterministic!
- JDK1.2 fixes by synchronize-ing the methods, preventing **close** and **read** from interleaving.



The screenshot shows a Microsoft Internet Explorer browser window displaying the Java documentation for the `java.io.BufferedInputStream` class. The browser's address bar shows the URL: `BufferedInputStream (Java 2 Platform SE v1.4.1) - Microsoft Internet Explorer`. The page content includes a navigation menu with options like Overview, Package, Class, Use Tree, Deprecated, Index, and Help. Below this, there are links for 'PREV CLASS', 'NEXT CLASS', 'FRAMES', and 'NO FRAMES'. A summary section lists 'NESTED', 'FIELD', 'CONSTR', 'METHOD', 'DETAIL: FIELD', 'CONSTR', and 'METHOD'. The main content area shows the class hierarchy: `java.io` package containing `Class BufferedInputStream`. It lists the superclass `java.lang.Object` and the interfaces `java.io.InputStream` and `java.io.FilterInputStream`. Below the hierarchy, the class declaration is shown: `public class BufferedInputStream extends FilterInputStream`. A detailed description follows: 'A BufferedInputStream adds functionality to another input stream—namely, the ability to buffer the input and to support the mark and reset methods. When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The mark operation remembers a point in the input stream and the reset operation cause all the bytes read since the most recent mark operation to be reread before new bytes are taken from the contained input stream.'

# Reaction to bug fix

“This really sucks. Now just to convert to [JDK 1.2] I’ve got to rewrite code that has worked since JDK 1.02... It’s pretty obvious that syncing close would break things.”

*Comment in Bug ID #4225348:  
“Attempt to close while reading  
causes deadlock”*



# Why was everyone so mad?

- Java socket programming idiom that requires the ability to **close** mid-**read**: “Hung” socket stream: Use separate thread to **close** and interrupt “hung” **read** or **write**
- In other words: clients assumed **read** and **close** can interleave!
  - Bug fix *prevents* interleaving.
  - Intent inferred — is it correct?
- Design choices — **What is/was the design intent?**
  - Interleaving **intended** — Fix race while allowing interleaving
  - Interleaving **not intended** — Provide alternative idiom to get the same effect.
- What should the Java designers have done? What’s a good solution to this problem? Whose fault was it?

# Upshot

- Fix was *undone* in JDK1.3
  - Re-enabled socket idiom.
  - Compromises safety of the class by re-enabling the race condition
- **BufferedInputStream** was fixed to both prevent the race and allow socket idiom for JDK 1.5
- Issue #1 – Race condition in deployed production library code
- Issue #2 – Lack of documentation of design intent with respect to concurrency.
- **Moral: bugs are hard, and correctness depends on context and user expectations.**



# Learning goals

- Conceive of testing as an activity designed to achieve *coverage* along a number of (non-structural!) dimensions.
- Enumerate testing strategies to help evaluate the following quality attributes: usability, reliability, security, robustness (both general and architectural), performance, integration.
- Give tradeoffs and identify when each of those techniques might be useful.
- Integrate testing into your project's lifecycle and practices.
- Outline a test plan.

# How can software fail?

## Faults of omission

- Specified behavior that for some reason is not present in the software, e.g. that the programmer forgot to include.
- 22—54% of software faults (B. Marick, 2000).

## Faults of commission

- An *unintended* behavior, that is, behavior that is not part of specification. i.e., implementing the wrong thing.

Correcting (isolating and fixing) a fault of commission requires more effort than a fault of omission (V. Basili, 1994).





- Two modes: high-power beam, low power beam.
- High power beam should only activate with beam spreader in place.
- Previous versions of Therac had hardware interlocks to prevent high beam without beam spreader; Therac 25 used software interlocks.
- Race condition + integer overflow: if operator provided manual input exactly when a one-byte counter overflowed, the interlock failed.
- Result: high power beam without spreader, radiation burns and sickness.

- A system is a series of externally observable states (remember requirements?).
- **Correct service** is delivered when the the **system function is implemented**, as observed externally.
- Things go wrong when the system enters into an externally visible erroneous state:
  - **Hazard:** the error may result in a failure (ex: a patient dies from radiation) (The Therac-20 had the same fault, and the same error, but there was a hardware override that prevented the failure).
  - **Failure:** the system enters into one or more erroneous states; the system enters one or more erroneous states; the system believes that the data it has is valid (radiation)
  - **Error:** the system enters into an erroneous state (the correct to an incorrect execution) (integer overflows and goes to 0)
  - **Fault:** the static problem in the code, hypothesized cause of an error (when data is invalid in memory, a variable called "error counter" instead of setting it to 0)

An error may not always result in a failure (The Therac-20 had the same fault, and the same error, but there was a hardware override that prevented the failure).

Faults in dead code do not cause errors.

Multiple faults may cause an error together.

# Terminology

# Basic definitions

- **Test case:** given a program, a configuration (including environment), a set of inputs to that program, a set of expected outputs, and an *oracle comparator* that determines whether the actual output of the program matches the expected output.
  - Note the difference between a *test input* and a *test case*!
- **Test suite:** a collection of test cases.
- **Oracle problem:** figuring out what the output of the program should be for a given input.



# Oracle comparator: why is this hard?

- For simple programs, oracle comparator can just be, for example, diff.
- For more complicated programs, an exact match may not be necessary.
- Example: a webserver where the exact response time varies with some random range. If web pages are returned within 5 milliseconds, we say everything is fine.
  - The oracle comparator for a test of the web server's performance needs to do more than just check the output with diff!

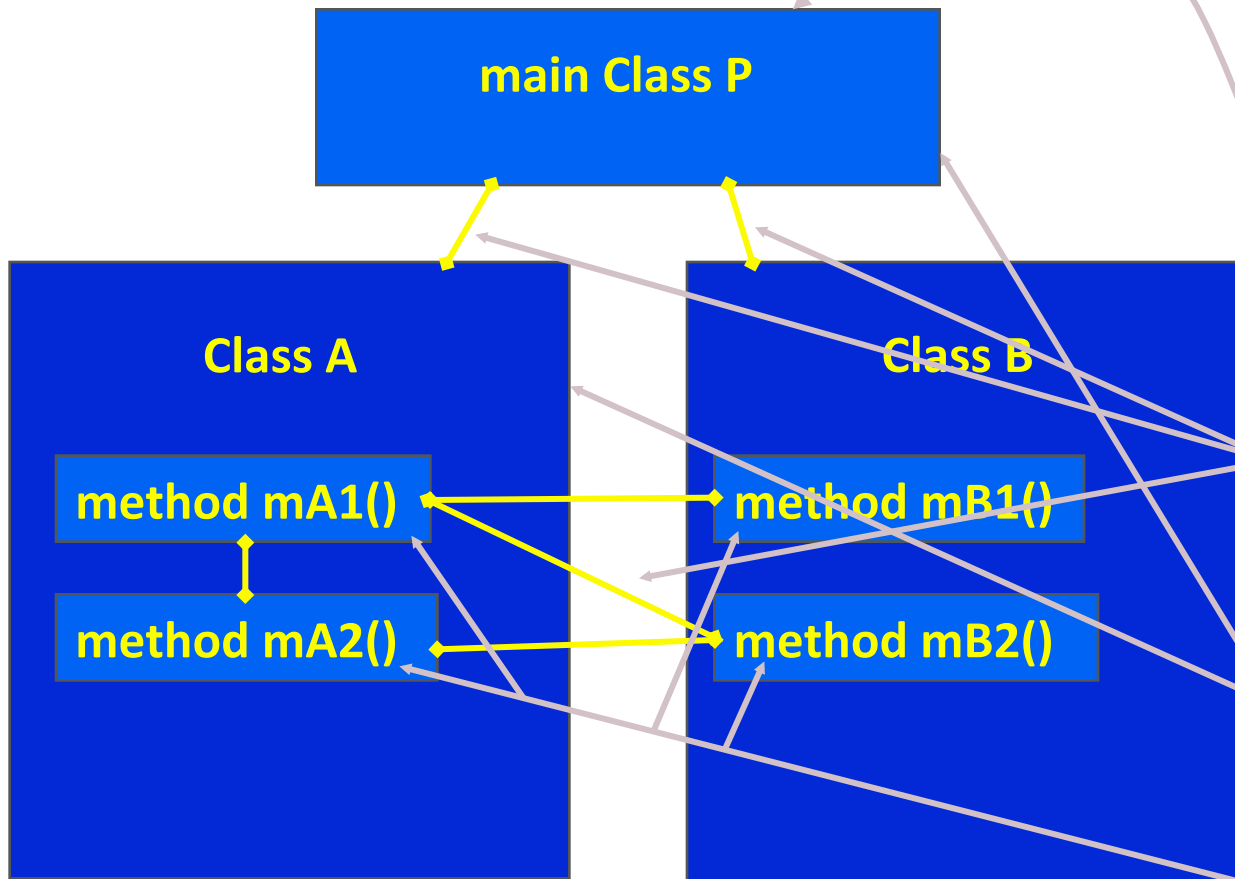
# Differential Testing

- When multiple implementations are available, search for differences between them (hand-selected or random inputs).
  - Also works for multiple "equivalent" configurations
- Example Csmith:
  - $n$  C Compiler and the same compilers at different optimization levels.
  - Generate random short code fragments
  - Compile code with each compiler, execute and compare output
  - On output differences the "majority" wins
  - Found 325 previous unknown compiler bugs





# Old: Verification at Different Levels



- Acceptance testing: Is the software acceptable to the user?
- System testing: Test the overall functionality of the system
- Integration testing: Test how modules interact with each other
- Module testing: Test each class, file, module or component
- Unit testing: Test each unit (method) individually

**This view obscures underlying similarities**

# Functional Correctness

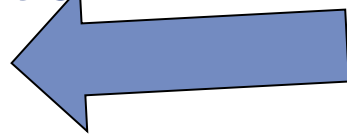
- Specification

- Formal Verification

- Unit Testing

- Type Checking

- Statistic Analysis



15-214

- Requirements definition

- Inspections, Reviews

- Integration/System/Acceptance/Regression/GUI/Blackbox/  
Model-Based/Random Testing

- Change/Release Management

15-313

# Class Integration and Test Order

- **Old programs** tended to be very hierarchical. Integration order was easy.
  - Test the “**leaves**” of the call tree, integrate up to the **root**
  - Goal is to minimize the number of **stubs** needed
- **OO programs** make this more complicated
  - Lots of kinds of **dependencies** (call, inheritance, use, aggregation)
  - **Circular** dependencies : A inherits from B, B uses C, C aggregates A
- **CITO** : *Which order should we integrate and test ?*
  - Must “**break cycles**”
  - Common goal : **least stubbing**
- **Designs** often have few cycles, but cycles creep in during **implementation**

# Practices – testing throughout the lifecycle

- Favor unit testing over integration and system testing
  - Top-down testing
    - Test full system with stubs (for undeveloped code).
    - Tests design (structural architecture), when it exists.
  - Bottom-up testing
    - Units -> Integrated modules -> system
- Unit tests find defects earlier (less cost and less risk)
  - During design, make API specifications specific
    - Missing or inconsistent interface (API) specifications
    - Missing representation invariants for key data structures
    - What are the unstated assumptions?
      - Null refs ok? Integrity check responsibility? Thread creation ok?
- Over-reliance on system testing can be risky
  - Possibility for finger pointing within the team
  - Difficulty of mapping issues back to responsible developers
  - Root cause analysis becomes blame analysis

# Practices – reporting defects

The screenshot shows the Eclipse bugzilla web interface. The browser window title is "Bug 141261 - crash - Shell create, RepositionWindow() - Unexpected Eclipse crash RC3 (JavaNati - Microsoft Intern...". The address bar shows "https://bugs.eclipse.org/bugs/show\_bug.cgi?id=141261". The page header includes the Eclipse logo and "Eclipse bugs version 2.56.1". The main content area displays the following information:

**Bugzilla Bug 141261** crash - Shell create, RepositionWindow() - Unexpected Eclipse crash RC3 (JavaNati... Last modified: 2006-11-14 17:46:58

**Bug List:** (31 of 200) [First](#) [Last](#) [Prev](#) [Next](#) [Show last search results](#) [Search page](#) [Enter new bug](#)

**[Eclipse] Bug#:** 141261 **Hardware:** Macintosh **Reporter:** Igor Goldenberg <igor@igaspaces.com>  
**Product:** Platform **OS:** Mac OS **Add CC:**  
**Component:** SWT **Version:** 3.2 **Priority:** P3 **CC:** grant\_gayed@ca.ibm.com, Mike\_Wilson@ca.ibm.com, paper@animecity.nu, steve\_northover@ca.ibm.com  
**Status:** NEW **Severity:** major **Target Milestone:**  
**Resolution:**  
**Assigned To:** Silenio Quarti <Silenio\_Quarti@ca.ibm.com>  Remove selected CCs

**QA Contact:**  
**URL:**  
**Summary:** crash - Shell create, RepositionWindow() - Unexpected Eclipse c  
**Status:**  
**Whiteboard:**  
**Keywords:**

Attachment	Type	Created	Size	Actions
<a href="#">Create a New Attachment</a> (proposed patch, testcase, etc)				<a href="#">View All</a>

**Bug 141261 depends on:**   
**Bug 141261 blocks:**  [Show dependency tree](#)  
**Votes:** 0 [Show votes for this bug](#) [Vote for this bug](#)  
**Additional Comments:**

- Reproducible defects
  - Easier to find and fix
  - Easier to validate
  - Built-in regression test
  - Increased confidence
- Simple and general
  - More value doing the fix
  - Helps root-cause analysis
- Non-antagonistic
  - State the problem
  - Don't blame

# Practices – track defects

- Issue: Bug, feature request, or query
  - May not know which of these until analysis is done, so track in the same database (Issuezilla)
- Provides a basis for measurement
  - Defects reported: which lifecycle phase
  - Defects repaired: time lag, difficulty
  - Defect categorization
  - Root cause analysis (more difficult!)
- Provides a basis for division of effort
  - Track diagnosis and repair
  - Assign roles, track team involvement
- Facilitates communication
  - Organized record for each issue
  - Ensures problems are not forgotten
- Provides some accountability
  - Can identify and fix problems in process
  - Not enough detail in test reports
  - Not rapid enough response to bug reports
- Should not be used for HR evaluation!

----- [Comment #4 From Clare Carty 2006-10-11 15:28 \[reply\]](#) -----  
 (In reply to [comment #3](#))  
 > I'm sorry but we really don't have enough details to be able  
 > problem. Could you try with another VM?  
 >  
 Problem didn't happen with another JRE - just the sun JRE.

----- [Comment #5 From Oleg Besedin 2006-10-11 15:38 \[reply\]](#) -----  
 This looks like a duplicate of the [bug 92250](#). Could you try if  
 with `-XX:MaxPermSize=256m` ?

----- [Comment #6 From Pascal Rapicault 2006-10-12 12:57 \[reply\]](#) -----  
 After further investigation, setting the permgen space to 1024 M  
 problem.  
 \*\*\* This bug has been marked as a duplicate of [92250](#) \*\*\*

----- [Comment #7 From Clare Carty 2006-10-12 15:18 \[reply\]](#) -----  
 This problem is still occurring on the dependent product with M  
 to 1024M. Please investigate.

----- [Comment #8 From John Arthorne 2006-10-12 17:24 \[reply\]](#) -----  
 What version of the Sun JRE are you using? I suggest trying with  
 later, as there are known memory leak problems with 1.5.0\_06 or

**Bug List:** (48 of 200) [First](#) [Last](#)

[Eclipse] Bug#: [160502](#) Hardware: PC  
 Product: Platform OS: Linux  
 Component: Runtime Version: 3.2.1  
 Status: REOPENED Priority: P3  
 Resolution: Severity: blocker  
 Assigned To: platform-runtime-inbox Target: Milestone:  
 <platform-runtime-inbox@eclipse.org>

QA Contact:   
 URL:   
 Summary: JVM crash at random intervals on SUSE 9 with Sun JRE 1.5  
 Status:   
 Whiteboard:   
 Keywords: vm

Reporter: [Clare Carty](#)  
 <ccarty@ca.ibm.com>  
 Add CC:   
 CC: [ccarty@ca.ibm.com](#)  
[john\\_arthorne@ca.ibm.com](#)  
 Remove selected CCs

Attachment	Type	Created	Size	Actions
<a href="#">screenshot of crash</a>	image/jpeg	2006-10-11 12:14	131.55 KB	<a href="#">Edit</a>
<a href="#">Create a New Attachment</a> (proposed patch, testcase, etc.)				<a href="#">View All</a>

Bug 160502 depends on:  [Show dependency tree](#)  
 Bug 160502 blocks:

Votes: 0 [Show votes for this bug](#) [Vote for this bug](#)

# Practices –social issues

- There are differences between developer and tester culture. Acknowledge that testers often deliver bad news.
- Avoid using defects in performance evaluations
  - Is the defect real?
  - Bad will within team
- Work hard to detect defects before integration testing
  - Easier to narrow scope and responsibility
  - Less adversarial
- Issues vs. defects

[Reassign](#) bug to

Reassign bug to default assignee and QA contact of selected component

[View Bug Activity](#) | [Format For Printing](#) | [Clone This Bug](#)

---

**Description:** [\[reply\]](#) **Opened:** 2005-07-25 07:03

I didn't even know that there was an undo feature inside the GUI editor, but today I accidentally pressed CTRL-Z instead of CTRL-S and the undo started... crashed.

Try adding some extension in the extensions page and then press CTRL-Z.

This is actually two bugs imho;

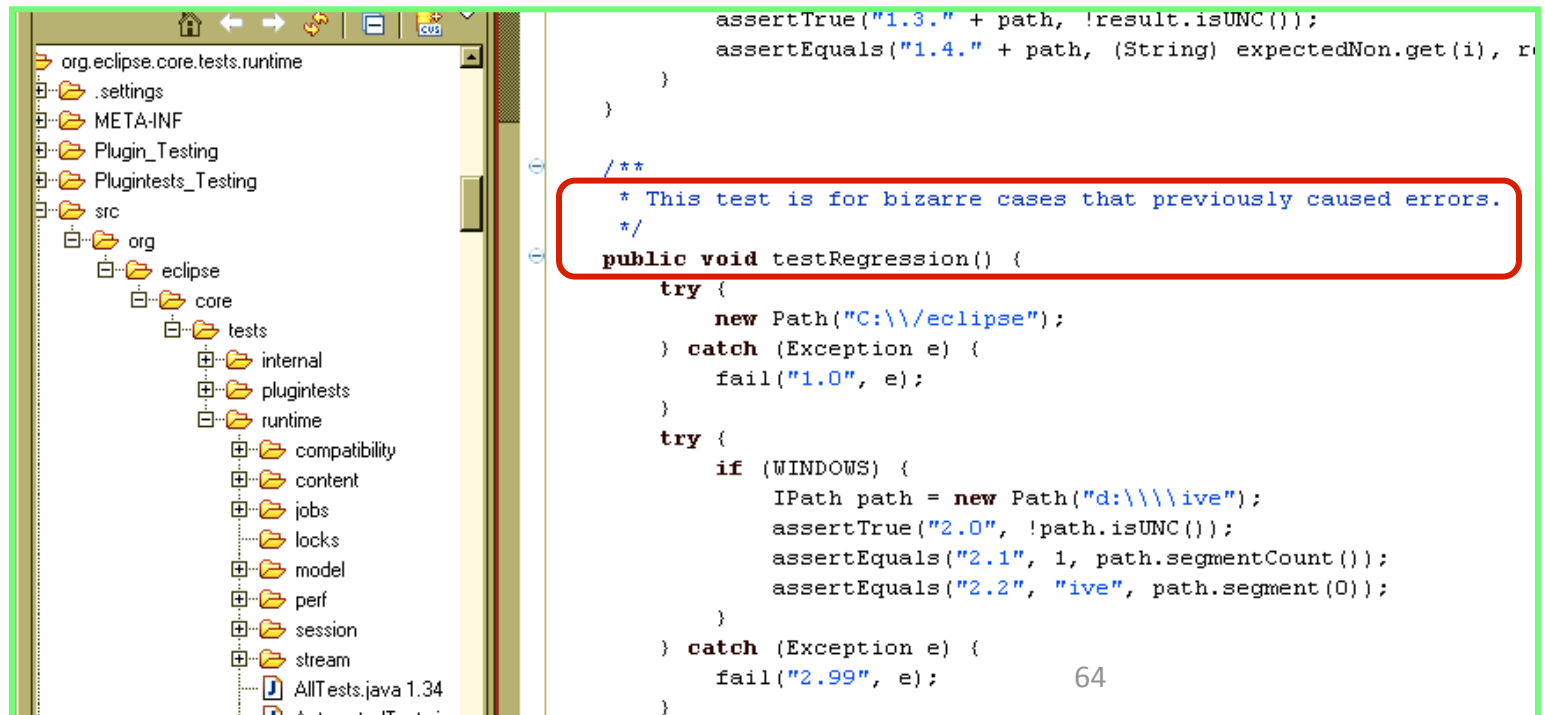
1. The details is very very poor so I really don't know what happened. A stacktrace would be great for debugging.
2. The undo obviously does not work correctly.

here is a screenshot of the crash:  
[http://mmemo.minimum.se/eclipse\\_crashes/eclipse\\_undo\\_crash.png](http://mmemo.minimum.se/eclipse_crashes/eclipse_undo_crash.png)

I don't have time for extensive reprod testing atm, maybe someone else can assist with this and see if they can get the plugin.xml editor to crash using weird combinations of editing and CTRL-Z undoing.

# Practices – use regression tests

- Goal: catch new bugs introduced by code changes
  - Check to ensure fixed bugs stay fixed
  - New bug fixes often introduce new issues/bugs
  - Incrementally add tests for new functionality
- Which new tests to run when adding new functionality?



```
org.eclipse.core.tests.runtime
├── .settings
├── META-INF
├── Plugin_Testing
├── Plugintests_Testing
├── src
│   └── org
│       ├── eclipse
│       └── core
│           └── tests
│               ├── internal
│               ├── plugintests
│               └── runtime
│                   ├── compatibility
│                   ├── content
│                   ├── jobs
│                   ├── locks
│                   ├── model
│                   ├── perf
│                   ├── session
│                   └── stream
│                       └── AllTests.java 1.34
└── ...

assertTrue("1.3." + path, !result.isUNC());
assertEquals("1.4." + path, (String) expectedNon.get(i), r
)
}
}

/**
 * This test is for bizarre cases that previously caused errors.
 */
public void testRegression() {
    try {
        new Path("C:\\\\eclipse");
    } catch (Exception e) {
        fail("1.0", e);
    }
    try {
        if (WINDOWS) {
            IPath path = new Path("d:\\\\ive");
            assertTrue("2.0", !path.isUNC());
            assertEquals("2.1", 1, path.segmentCount());
            assertEquals("2.2", "ive", path.segment(0));
        }
    } catch (Exception e) {
        fail("2.99", e);
    }
}
```

64



# When a Regression Test Fails

- Regression tests are **evaluated** based on whether the result on the new program  $P$  is **equivalent to** the result on the previous version  $P-1$ 
  - If they **differ**, the test is considered to have **failed**
- Regression test failures represent **three possibilities** :
  - The **software** has a fault – *Must fix the fix*
  - The **test values** are no longer valid on the new version – *Must delete or modify the test*
  - The **expected output** is no longer valid – *Must update the test*
- Sometimes **hard to decide** which !!

# Evolving Tests Over Time

- Changes to **external interfaces** can sometimes cause all tests to fail
  - Modern **capture / replay** tools will not be fooled by trivial changes like color, format, and placement
  - **Automated scripts** can be changed automatically via global changes in an editor or by another script
- Adding **one test** does not cost much – but over time the cost of these small additions start to pile up

# Practices - Updating Test Suites

- Which **tests to keep** can be based on several policies
  - Add a new test for every **problem report**
  - Ensure that a **coverage criterion** is always satisfied
- Sometimes harder to choose tests **to remove**
  - Remove tests that **do not contribute** to satisfying coverage
  - Remove tests that have **never found a fault** (risky !)
  - Remove tests that have found the **same fault** as other tests (also risky!)
  - (no one ever got fired for leaving a test suite in...)
- **Reordering** strategies
  - If a suite of  $N$  tests satisfies a coverage criterion, the tests can often be reordered so that the first  $N-x$  tests satisfies the criterion – so the remaining tests can be removed

# Practices: Root cause analysis

- Identify the “root causes” of frequent defect types, locations
  - Requirements and specifications?
  - Architecture? Design? Coding style? Inspection?
- Try to find all the paths to a problem
  - If one path is common, defect is higher priority
  - Each path provides more info on likely cause
- Try to find related bugs
  - Helps identify underlying root cause of the defect
  - Can use to get simpler path to problem
  - This can mean easier to fix
- Identify the most serious consequences of a defect

# Test Process Maturity Levels

- Level 0: There's no difference between testing and debugging
- Level 1: The purpose of testing is to show correctness
- Level 2: The purpose of testing is to show that the software doesn't work
- Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software.

# Test plans

- What quality techniques are used and for what purpose. In terms of testing:
  - What we will test
  - How we will test
  - When we will do so, when we will stop.
  - Who will write the tests, who will run them.
  - Why we know it's a good set of tests.

<b>1</b>	<b>Scope</b>	
1.1	System Overview . . . . .	
<b>2</b>	<b>Reference Documents</b>	
<b>3</b>	<b>Software Test Environment</b>	
<b>4</b>	<b>Test Identification</b>	
4.1	General Information . . . . .	
4.1.1	Test Level . . . . .	
4.1.2	Test Classes . . . . .	
4.2	Planned Tests . . . . .	
4.2.1	Test 1 – Linear Operators . . . . .	
4.2.2	Test 2 – Convergence of Multifluid Project . . . . .	
4.2.3	Test 3 – Fixed-boundary diffusion solver . . . . .	
4.2.4	Test 4 – Upwind advection . . . . .	
4.2.5	Test 5 – Fixed-boundary projection test . . . . .	
4.2.6	Test 6 – Surface Tension Test . . . . .	
4.2.7	Test 7 – Multifluid system test . . . . .	
4.2.8	Test 8 – Multifluid AMR test . . . . .	
4.2.9	Test 9 – Multifluid system regression test . . . . .	
<b>5</b>	<b>Test Schedules</b>	
<b>6</b>	<b>Bug Tracking</b>	
<b>7</b>	<b>Requirements Traceability</b>	

# ANSI/IEEE 829-1983

- Test plan: “A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.”
- Emphasizes documentation, not actual testing: a **well documented vacuum**.

# Why Produce a Plan?

- Ensure the test plan addresses the needs of stakeholders
- Customer: may be a required product, for operations and support
  - E.g., Government systems integration, safety-critical certification: avionics, health devices, etc.
- A separate test organization may implement part of the plan
  - “IV&V” – Independent verification and validation
- May benefit development team
  - Set priorities: Use planning process to identify areas of hazard, risk, cost
- Additional benefits – the plan is a team product
  - Test quality
    - Improve coverage via list of features and quality attributes
    - Analysis of program (e.g. boundary values)
    - Avoid repetition and check completeness
  - Communication
    - Get feedback on strategy
    - Agree on cost, quality with management
  - Organization
    - Division of labor
    - Measurement of progress



# Managing Test Artifacts

- Don't fail because of **lack of organization**
- Keep **track** of :
  - Test design documents
  - Tests
  - Test results
  - Automated support
- Use **configuration control**
- Keep track of **source of tests** – when the source changes, the tests must also change

# Chaos monkey/Simian army

- A Netflix infrastructure testing system.
- “Malicious” programs randomly trample on components, network, datacenters, AWS instances...
  - Chaos monkey was the first – disables production instances at random.
  - Other monkeys include Latency Monkey, Doctor Monkey, Conformity Monkey, etc... Fuzz testing at the infrastructure level.
  - Force failure of components to make sure that the system architecture is resilient to unplanned/random outages.
- Netflix has open-sourced their chaos monkey code.

# QA throughout lifecycle

