JOHNS HOPKINS
UNIVERSITY

EN.601.422 / EN.601.622

Software Testing & Debugging

The material in this video is subject to the copyright of the owners of the material and is being provided for educational purposes under rules of fair use for registered students in this course only. No additional copies of the copyrighted work may be made or distributed.

Testing vs. Debugging

- So far: Testing
  - Look for inputs that cause failures
    - Coverage criteria
    - Test generation
    - Test Oracle
- Program fails, now what? → **Debugging**
  - Failures are typically discovered by
    - Tests
    - Real user

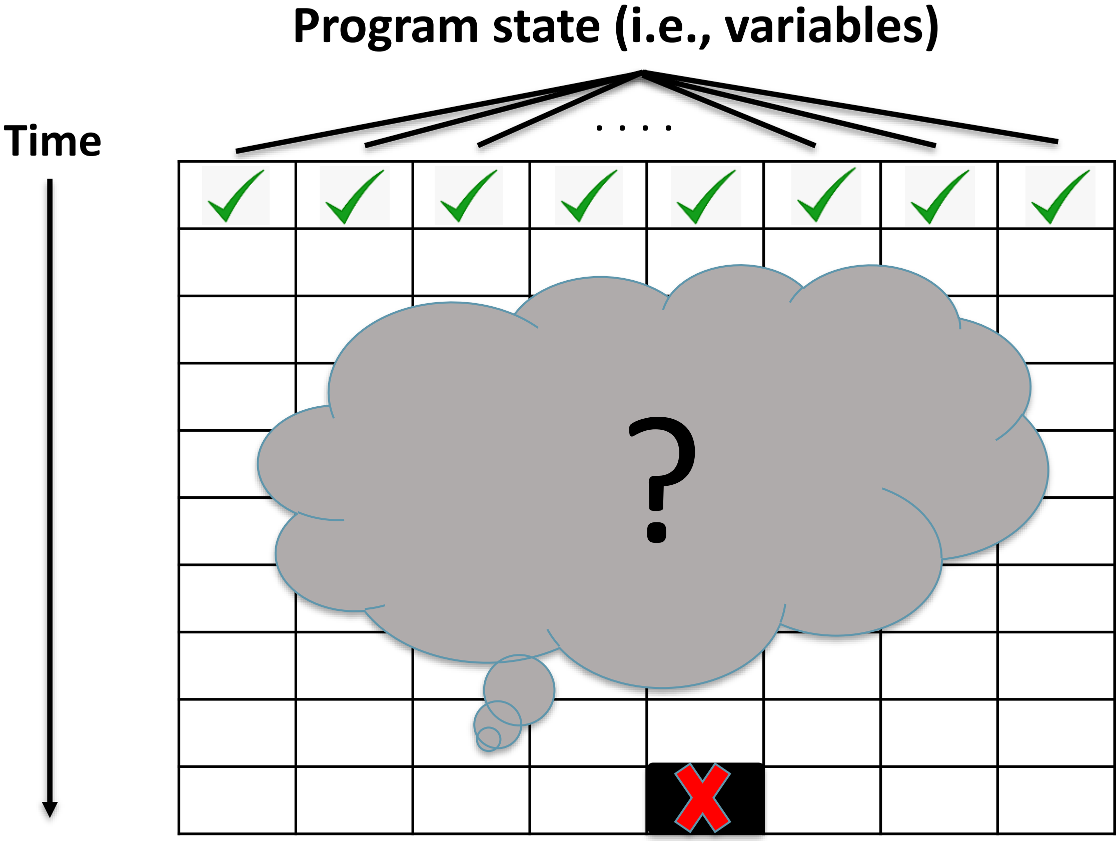
Six Stages of Debugging

1. That can’t happen.
2. That does not happen on my machine.
3. That should not happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

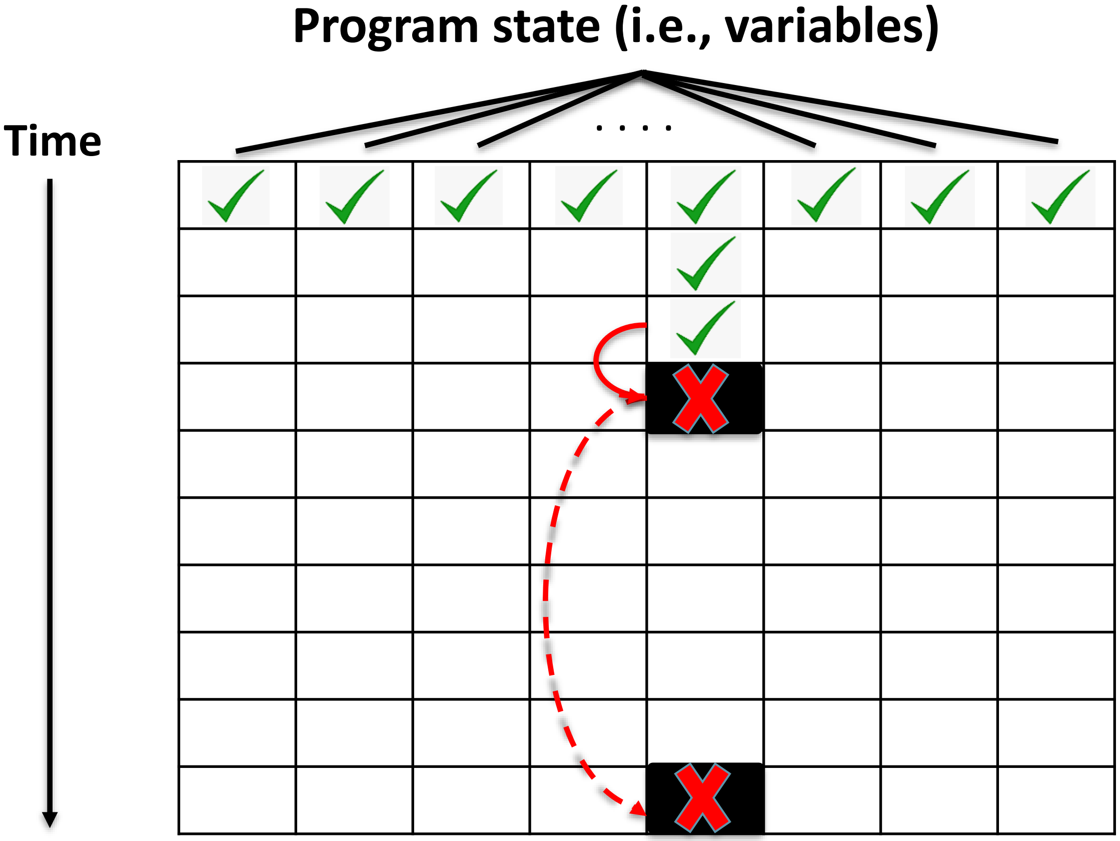
Debugging Steps

- Debugging Steps:
  1. Reproduce the error, understand
  2. Isolate and Minimize (shrink)– Simplification
  3. Eyeball the code, where/what could it be? Reason backwards
  4. Devise and run an experiment to test your hypothesis
  5. Repeat 3 and 4 until you understand what is wrong
  6. Fix the Bug and Verify the Fix
  7. Create a Regression Test

Search in Time and Space



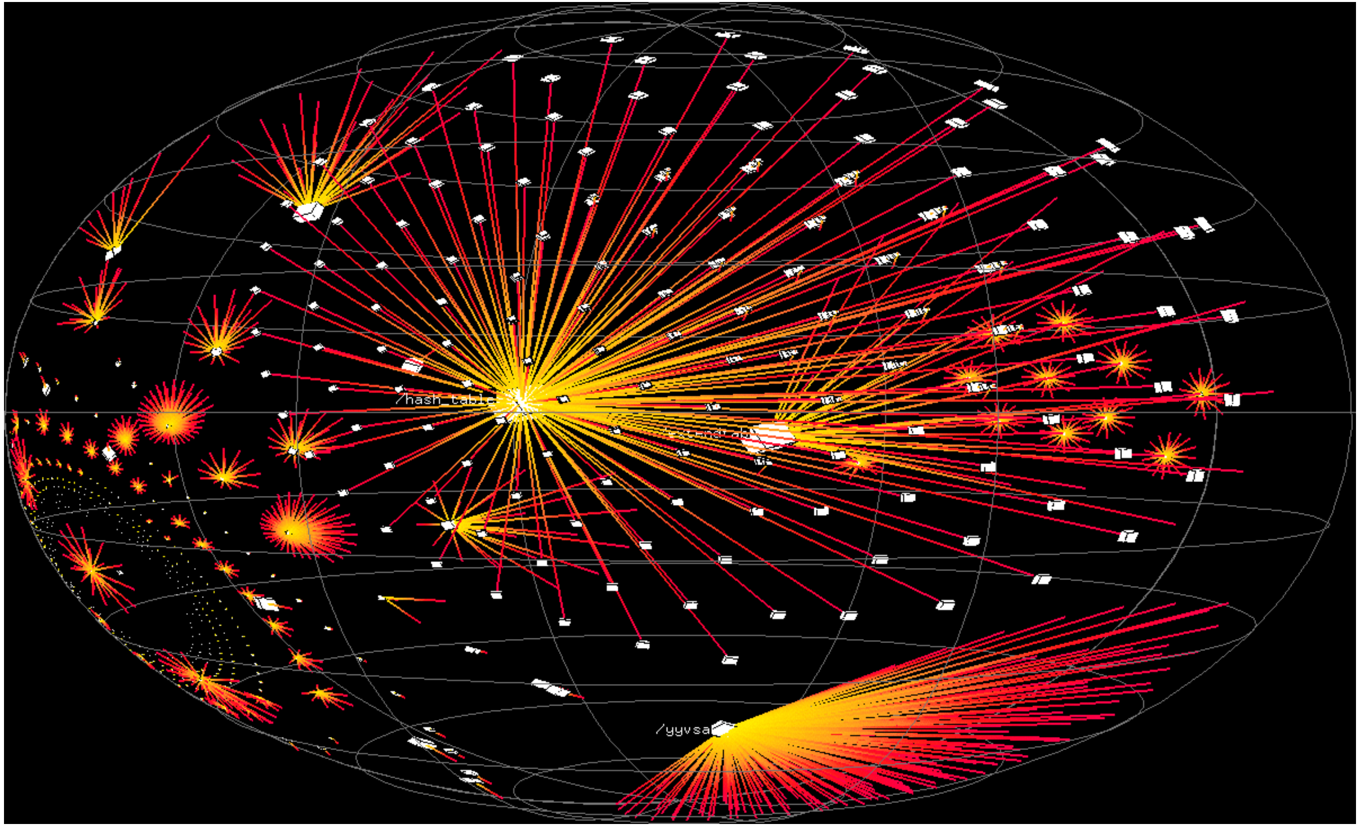
The Fault!



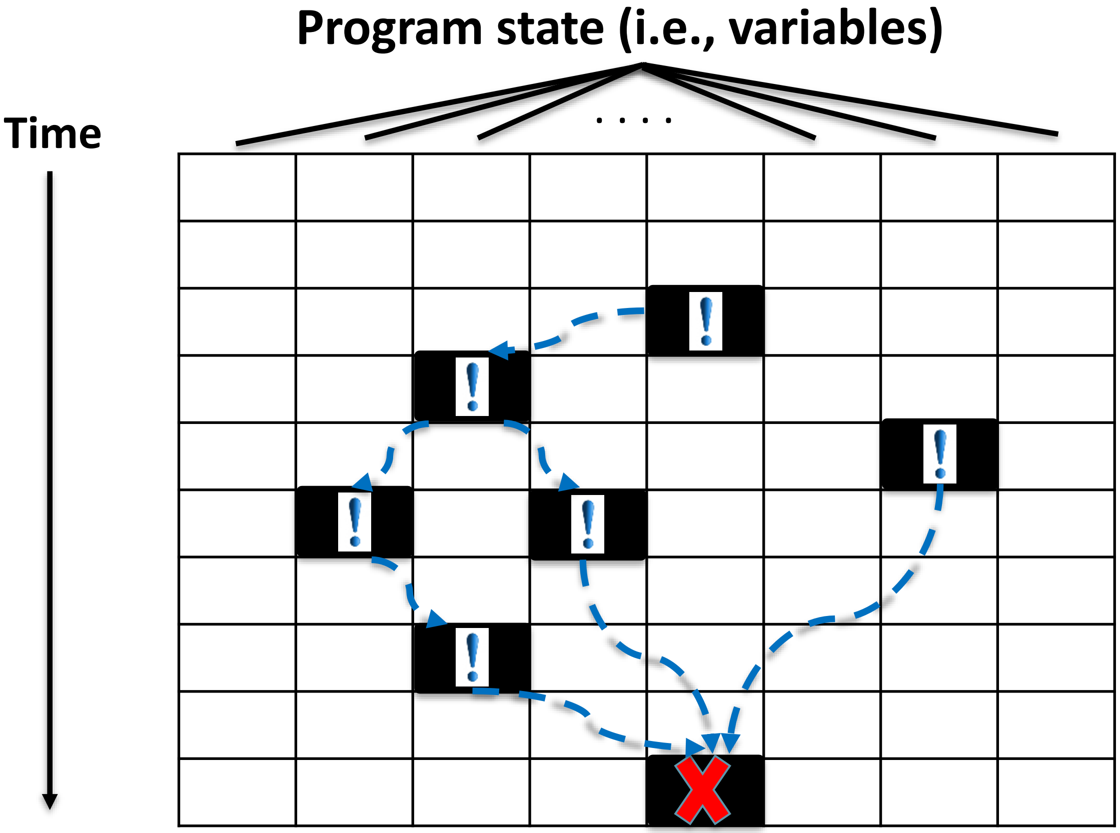
Debugging

- Debugging is a search in space (a given state of the program) and time (all the states that the program goes through) to find and resolve faults in a computer program
  - Each single program state may involve a large number of variables
  - A program may pass through millions of states before failure occurs
- This may seem like searching for a needle in endless rows of haystack

Program State Can Be Huge

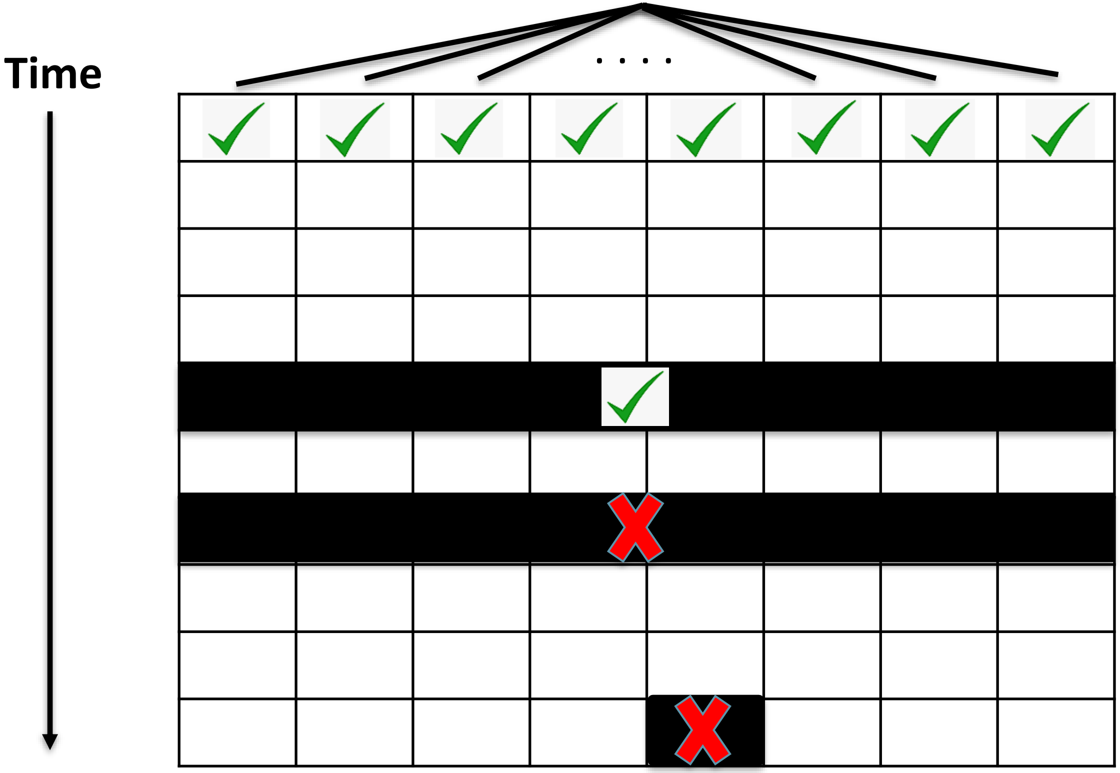


Finding the Origins



Observing Transitions of States

Program state (i.e., variables)



How Failures Come To Be

- A failure comes to be in three stages:
  - The programmer makes a *fault* by creating a *defect* in the code
  - The defect causes an infection (i.e., incorrect state or error)
  - The infection causes a failure -- an externally visible error

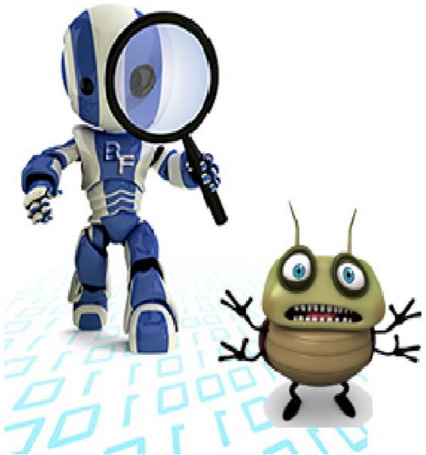
- Not every defect results in an infection, and not every infection/error results in a failure.

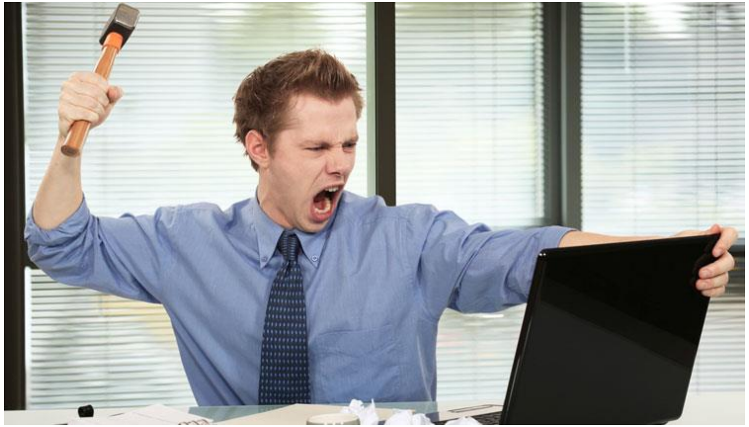
How To Debug Automatically

- A variety of tools and techniques are available to *automate debugging:*
  - Program Slicing
  - Observing & Watching State
    - logging
    - using debugger tools
  - Asserting Invariants
  - Detecting Anomalies
  - Isolating Cause-Effect Chains

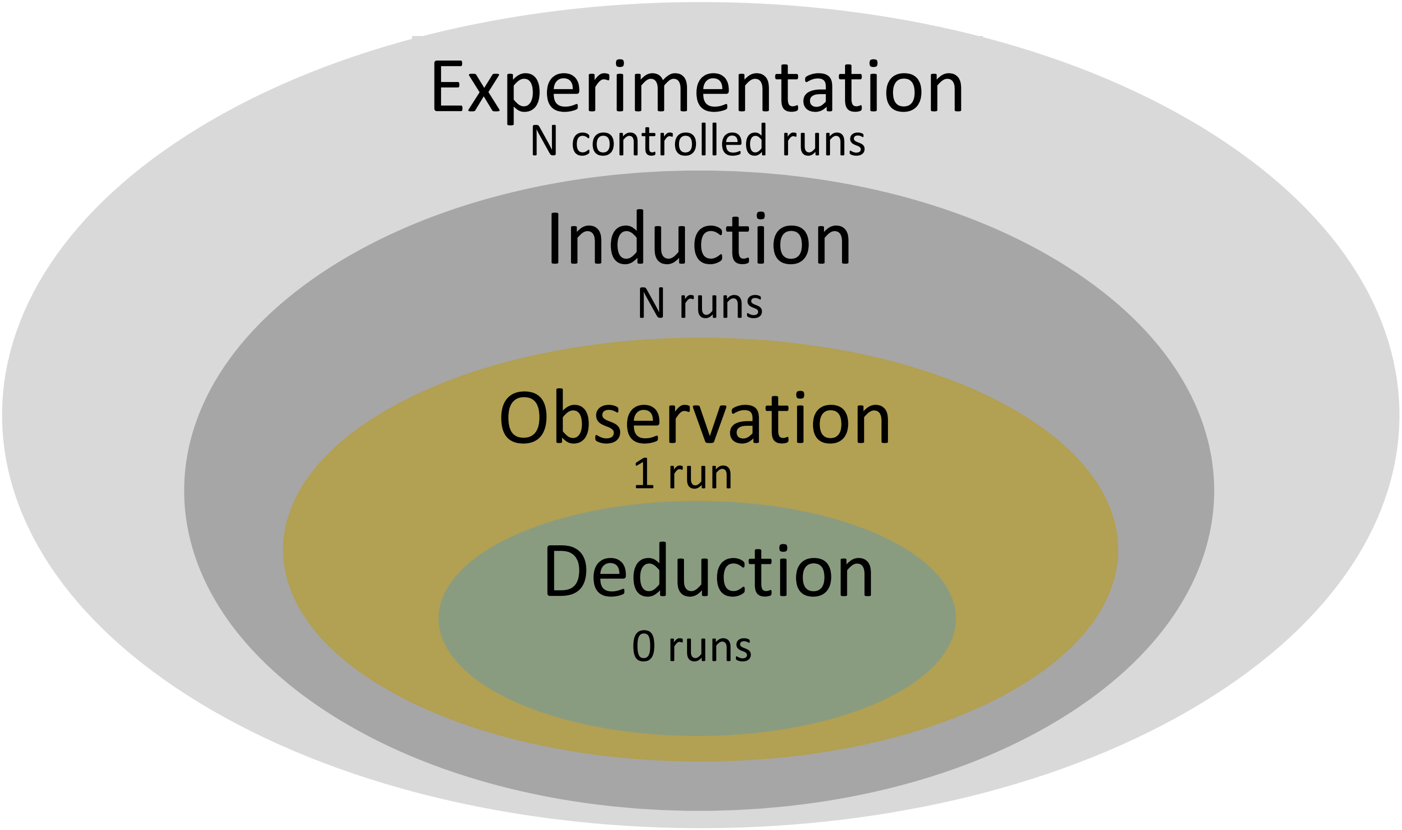
Debugging Initiation

- Typically, debugging process is initiated when:
  - An automated test case causes a failure
  - A user experiences a failure and submits a *"Bug Report"*





Debugging is Reasoning



Experimentation

N controlled runs

Induction

N runs

Observation

1 run

Deduction

0 runs

How to Debug Automatically

- A variety of tools and techniques are available to *automate debugging:*
  - Program Slicing
  - **Observing & Watching State**
    - **logging**
    - **using debugger tools**
  - Asserting Invariants
  - Detecting Anomalies
  - Isolating Cause-Effect Chains

Debugging by Observation

- Determine facts based on what has happened in a concrete run
- Know what to observe and when to observe in a systematic way
- Debugging by Observation techniques:
  - Logging
  - Interactive debugging
  - Postmortem debugging

Observation Principles

- **Proceed systematically:** Rather than observing values at random, search scientifically → develop hypotheses
- **Know what to observe and when to observe:** program run is a long succession of huge program states (i.e., large number of variables), so it is impossible/impractical to observer everything all the time
- **Do not interfere:** Whatever you observe should be the effect of the original program run rather than an effect of your observation

Debugging by Observation

- How can we observe the software state:

Logging the execution

Logging the Execution

- General idea: Insert output statements at specific places in the program
- Also known as *println* debugging

```
public void quickSort(int arr[],
                      int low, int high) {
    if (low < high) {
        /* pi is partitioning index,
           arr[pi] is now at right place */
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
public void quickSort(int arr[],
                      int low, int high) {
    if (low < high) {
        /* pi is partitioning index,
           arr[pi] is now at right place */
        int pi = partition(arr, low, high);
        System.out.println("pi is: " + pi);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

“println” Debugging Issues

- Cluttered code: logging statements serve no purpose other than debugging
- Cluttered output: logging statements can produce a large amount of output which gets interleaved with ordinary output
  - designate a separate channel for logging (e.g., error channel, a separate logfile etc.)
- Slowdown: huge amount of logging statements can slow down the program
- Loss of Data: for performance reasons, outputs are buffered before being outputted
  - if the program crashes, output data will be lost
  - do not buffer or buffer less frequently → **Slowdown**

Better Logging

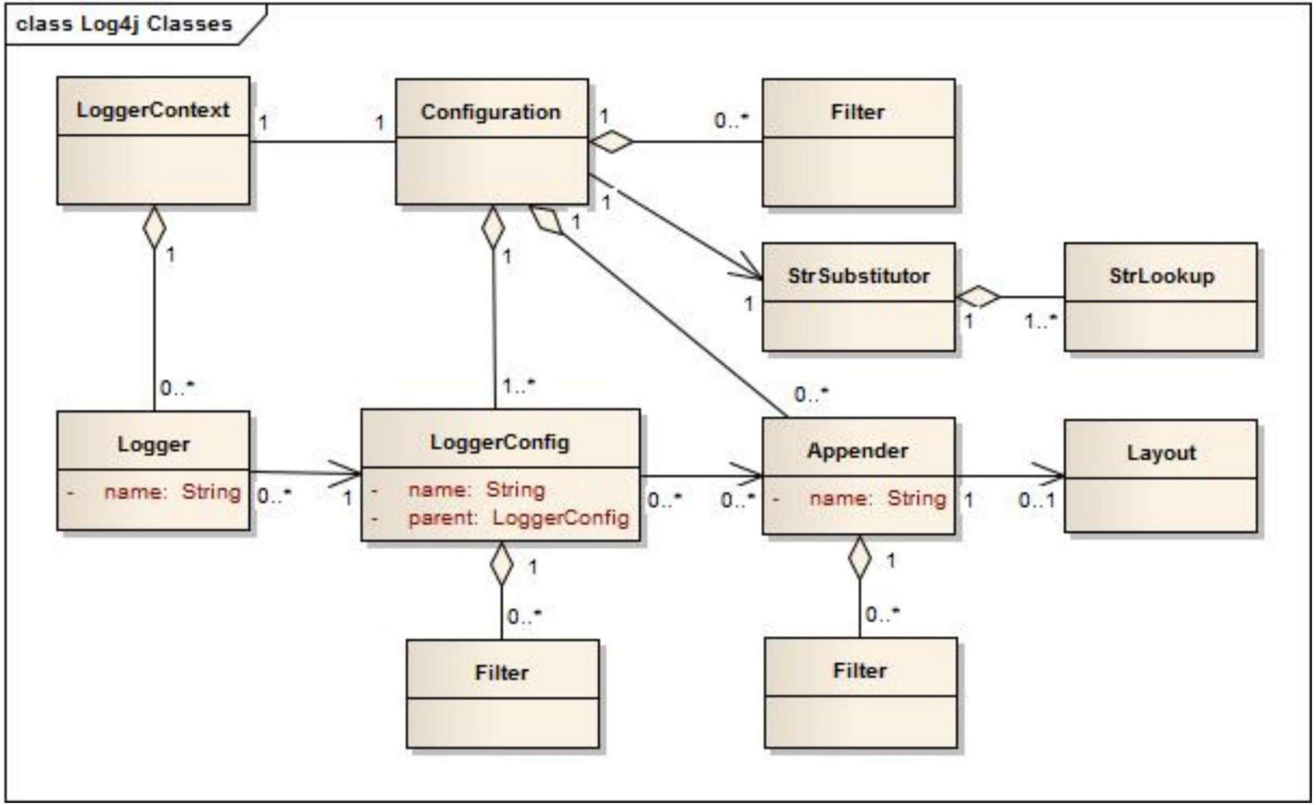
- To address some of the issues discussed:
  - Make a dedicated function to log the debugging-related messages
    - Example: write a function named dprintln(String) and call that when logging for debugging
  - Turn the function off when releasing/in production
    - Calculating arguments and calling an empty function is still costly
    - In languages that support *macros* like C/C++, it is not an issue
  - Better yet, make use of dedicated “logging libraries”
    - The first and foremost advantage of any logging API over plain System.out.println resides in its ability to disable certain log statements while allowing others to print unhindered
    - Tools available: Log4j, Log4net, Log4c, etc.

Apache Log4j 2

- A full-fledged logging framework
- Offers more functionality compared to `java.util.logging`
- Many open-source applications utilize log4j

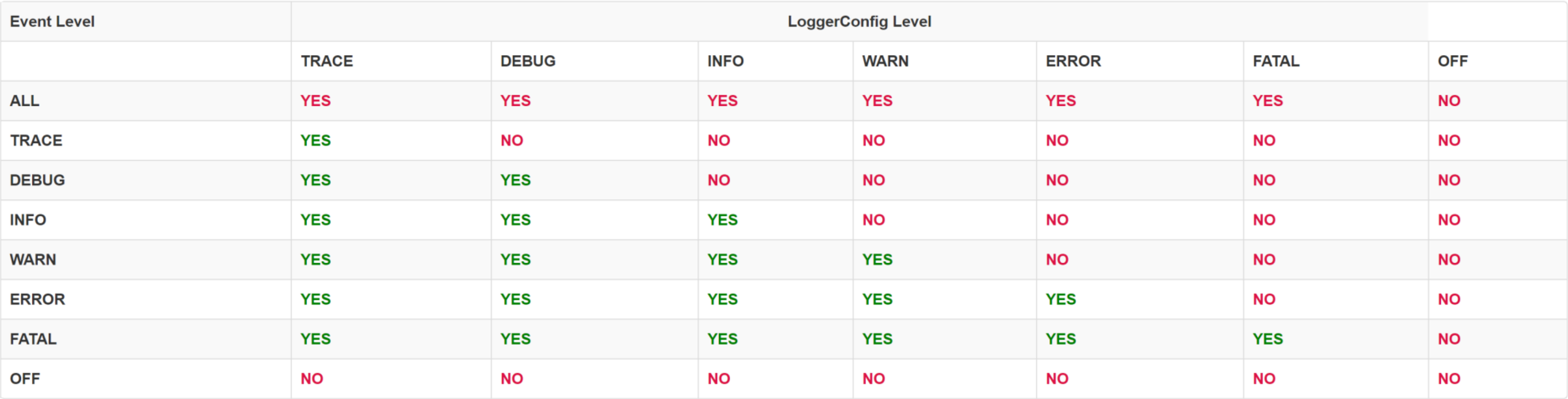
https://logging.apache.org/log4j/2.x/

Log4j Architecture



Log Levels

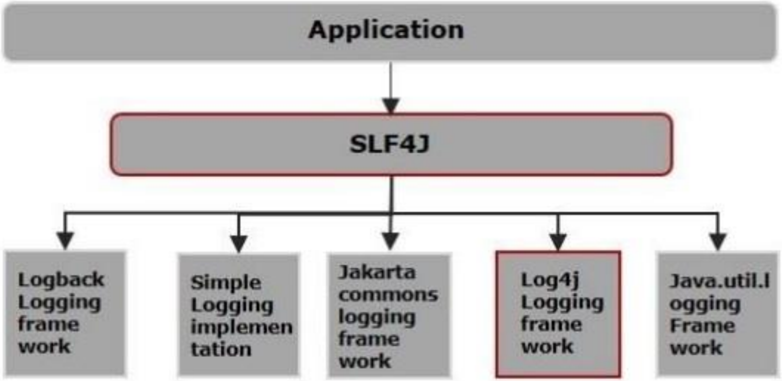
All < Trace < Debug < Info < Warn < Error < Fatal < Off



Event Level	LoggerConfig Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

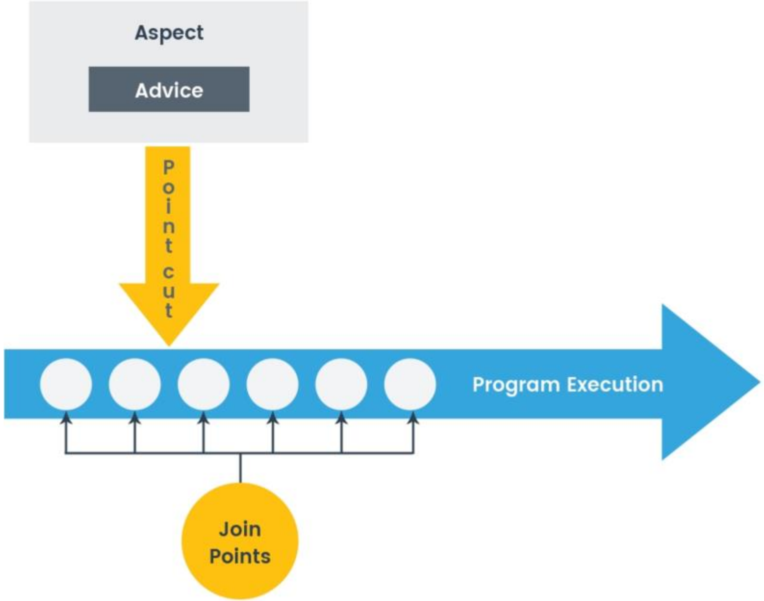
SLF4J

- Simple Logging Facade for Java (abbreviated SLF4J):
  - acts as a facade for different logging frameworks e.g., java.util.logging, logback, Log4j 2).
  - The underlying logging framework can be plugged in at run-time


Application
SLF4J
Logback Logging frame work
Simple Logging implemen tation
Jakarta commons logging frame work
Log4j Logging frame work
Java.util.l ogging Frame work

Log with Aspects

- **Aspect-Oriented (AO) Programming:** a programming paradigm that *aims to increase modularity* by allowing the separation of ***cross-cutting concerns*** from the ***core concern***
- *Basic idea: Separate concerns into individual syntactic entities (aspects)*
- *Aspect code (advice) is woven into the program code at specific places (join points)*
- *The same aspect code can be woven into multiple places (pointcuts)*
  - *e.g.,* log all function calls when the function's name begins with “set”



Logging as a Crosscutting Concern

- Logging is a cross-cutting concern: does not have anything to do with the logic of the program → **logging is a separate concern**
- Implement "logging" as a separate *aspect* that cuts through points of interest in the code and logs events/activities/variable values/etc. of interest.
- Other cross-cutting concerns (i.e., aspects) might be security, data validation, authentication system, synchronization, optimizations etc.

Using Debuggers (aka Observation Tool)

- Logging requires writing and integrating extra code into the program
- Debugger:
  - Getting started fast – without altering the program code at hand
  - Flexible observation of arbitrary events
  - Transient sessions – no code is written

Debuggers

- Debugger: an external observer tool that hooks itself into the execution of the program and observes (possibly manipulates too) the state of the program.
- Debuggers functionalities:
  - Execute the program and make it stop under specific conditions
  - Observe the state of the stopped program
  - Change the state of the program

Debugging Session Using a Debugger

- Before starting the session:
  - Try to develop a *hypothesis* or several hypotheses: explanations for what might be wrong
  - Make note of parts of the code and variables that are involved (i.e., should be investigated) based on your hypothesis/hypotheses
    - What part(s) of program state should be checked
  - Decide on particular points of interest in the program where you like to stop and check things out:
    - **Breakpoint:** when program reaches a breakpoint, it stops (i.e., hands over the control to the debugger) giving you a chance to check things out

Watchpoints and Conditional Breakpoints

- Watchpoints: a data breakpoint
  - Program execution stops and execution control is handed over to the debugger if a variable (or an expression) is read and/or is changed
  - Useful when you want to focus on a specific variable/expression
- Conditional Breakpoint:
  - Program execution stops, and execution control is handed over to the debugger if a certain condition evaluates to true
- Watchpoints and conditional breakpoints are expensive:
  - The debugger must verify the value of watched variable/expression and/or a condition after each instruction
  - Slows down program execution by a factor of 1000

Interactive vs. Postmortem Debugging

- Interactive debuggers allow step-by-step execution and inspection/modification of state
- Postmortem debuggers analyze an application after it has crashed:
  - Analyze the core dump
  - Automated tools available: WinDbg, LLDB, GDB etc.

Simplifying

- Once one has reproduced a problem, one must find out what’s relevant:
  - Does the problem really depend on 10,000 lines of input?
  - Does the failure really require this exact schedule?
  - Do we need this sequence of calls?

Why Simplify

- An airplane crashes:
  - Remove passenger seats, does it still crash?
  - Remove coffee machine, does it still crash?
  - Remove the engines, it does not move

⬇

engines are relevant!

Simplifying and Circumstances

- For every circumstance of the problem, check whether it is relevant for the problem to occur.
- If it is not, remove it from the problem report or the test case in question.
- Any aspect that may influence a problem is a circumstance:
  - Aspects of the problem environment
  - Individual steps of the problem history

Simplifying by Experimentation

- By experimentation, one finds out whether a circumstance is relevant or not:
- Omit the circumstance and try to reproduce the problem.
- The circumstance is relevant iff the problem no longer occurs.

Mozilla Gecko and a Reported Bug

- Gecko: Mozilla HTML layout engine
- In 1999, there were 370 open problem reports
- Loading an 896-lines HTML crashed the browser
- Much better to work with the smallest possible HTML input file that contains the “failure cause”

Why Simplify

- Ease of communication:
  - A simplified test case is easier to communicate.
- Easier debugging:
  - Smaller test cases result in smaller states and shorter executions.
- Identify duplicates:
  - Simplified test cases subsume several duplicates.

bugzilla.mozilla.org

```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<
98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="
VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac Syste
VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System 8.5<OPTION VALUE="Mac S
VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTIO
VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION VALUE="OpenBSD">OpenBSD<OPTION VALUI
VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutr
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION VALUE="OSF/1">OSF/1<OPTION VALUE="Solari
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>

</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="

</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION VALUE=
VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
```

What’s relevant in here?

The Gecko BugAThon

- New problem reports came in way faster than the Mozilla developers could possibly simplify them or even look at them

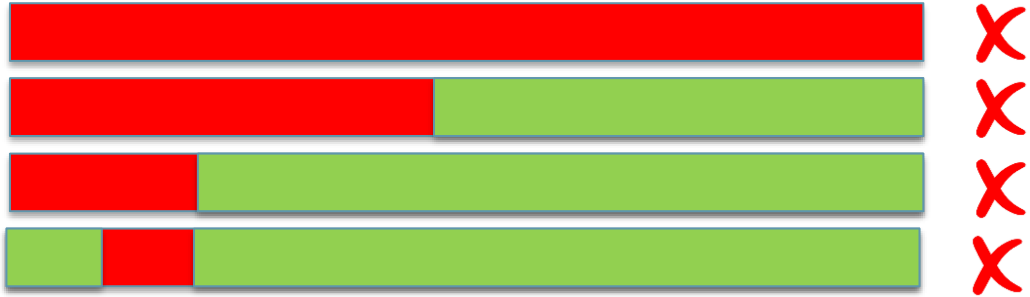
- Eric Krock, the Mozilla product manager, came up with a brilliant idea
  - Download the Web page to your machine.
  - Using a text editor, start removing HTML from the page. Every few minutes, make sure it still reproduces the bug.
  - Code not required to reproduce the bug can be safely removed.
  - When you’ve cut away as much as you can, you’re done.

Rewards

- Asked the users themselves to help with simplifying the bugs:
  - 5 bugs - invitation to the Gecko launch party
  - 10 bugs - the invitation, plus an attractive Gecko stuffed animal
  - 12 bugs - the invitation, plus an attractive Gecko stuffed animal autographed by Rick Gessner, the Father of Gecko
  - 15 bugs - the invitation, plus a Gecko T-shirt
  - 20 bugs - the invitation, plus a Gecko T-shirt signed by the whole raptor team

Binary Search

- Proceed by binary search. Throw away half the input and see if the output is still wrong.
- If not, go back to the previous state and discard the other half of the input.



1: ✗ 896 lines

2: ✗ 448 lines

3: ✗ 224 lines

4: ✗ 112 lines

.

.

.

12: <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗ 40 characters

Simplified Input

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

- Simplified from 896 lines to one single line
- Required 12 tests only

Benefits

- Ease of communication:
  - All one needs is “<SELECT> tag causes a crash”
- Easier debugging:
  - We can directly focus on the piece of code that renders <SELECT>
- Identify duplicates:
  - Check other test cases whether they’re <SELECT>-related, too.

Automated Simplification

- Manual simplification is slow & boring.
- We have machines for mechanical tasks.
- Basic idea:
  - We set up an automated test that checks whether the failure occurs or not e.g., Mozilla crashes or not
  - We implement a strategy that realizes the binary search

Automated Test

- Launch Mozilla
- Replay (previously recorded) steps from problem report
- Wait to see whether
  - Mozilla crashes (= the test fails)
  - Mozilla still runs (= the test passes)
- If neither happens, the test is *unresolved*

Binary Search

- What do we do if both halves pass?
  - Increase granularity, i.e., break the input into smaller pieces

Example

```
public static int checkSum(int[] a)
```

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever “a” contains two identical consecutive numbers, **but we don’t know that yet**
- we have a failed test case, e.g., from protocol transmission:
  - {1, 3, 5, 3, 9, 17, 44, 3, 6, 1, 1, 0, 44, 1, 44, 0}

Another Example (N is number of chunks)

N = 2

1 | 3 | 5 | 3 | 9 | 17 | 44 | 3 | 6 | 1 | 1 | 0 | 44 | 1 | 44 | 0 ✗

1 | 3 | 5 | 3 | 9 | 17 | 44 | 3 ✓

6 | 1 | 1 | 0 | 44 | 1 | 44 | 0 ✗

6 | 1 | 1 | 0 ✗

44 | 1 | 44 | 0 ✓

6 | 1 ✓

N = 4

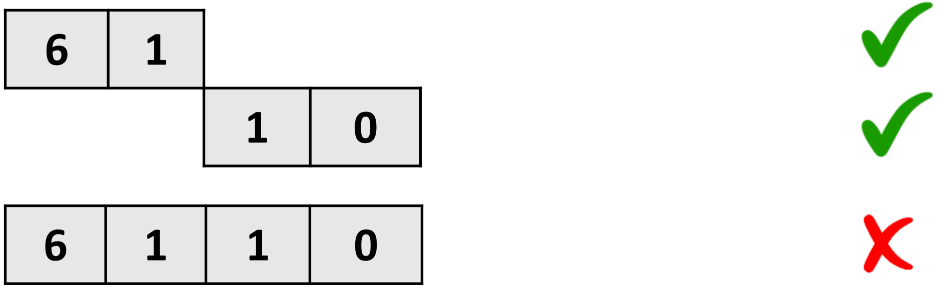
Increase granularity

1 | 0 ✓

Another Example - Continued

N = 4

Increase granularity



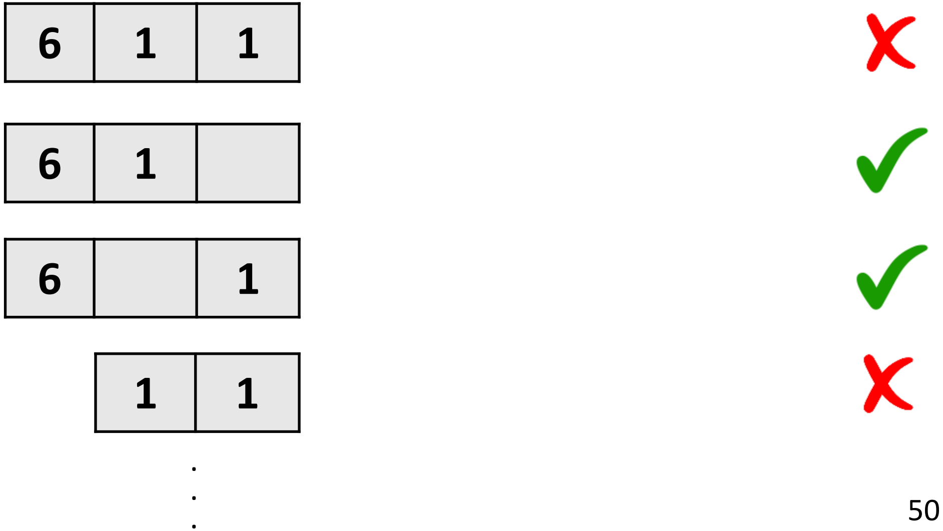
6 1 ✓

1 0 ✓

6 1 1 0 ✗

N = 3

Adjust granularity to input size



6 1 1 ✗

6 1 ✓

6 1 ✓

1 1 ✗

.
.
.

ddmin Algorithm

- Let **`c`** be a failing input configuration (sequence of individual inputs)
- **`test(c)`** runs a test on **`c`** with possible outcome PASS or FAIL
- **`n`** is the number of chunks to split **`c`** into (initially **`n = 2`**). We will remove one chunk at a time and test the remaining input.

```
ddMin(c, n) :
1. If |c| = 1 return c
```

ddmin Algorithm

- Let `c` be a failing input configuration (sequence of individual inputs)
- `test(c)` runs a test on `c` with possible outcome PASS or FAIL
- `n` is the number of chunks to split `c` into (initially `n = 2`). We will remove one chunk at the time and test the remaining input.

```
ddMin(c, n) :
1. If |c| = 1 return c
```

Otherwise, systematically remove one chunk c_i at the time. Test the remaining input c \ c_i :

```
2. If there exist some $c_i$ such that test(c \ $c_i$ ) = FAIL
return ddMin(c \ $c_i$ , max(n-1, 2))
```

ddmin Algorithm

- Let **c** be a failing input configuration (sequence of individual inputs)
- **test(c)** runs a test on **c** with possible outcome PASS or FAIL
- **n** is the number of chunks to split **c** into (initially **n = 2**). We will remove one chunk at the time and test the remaining input.

```
ddMin(c, n) :
1. If |c| = 1 return c
```

Otherwise, systematically remove one chunk c_i at the time. Test the remaining input **c** \ c_i :

```
2. If there exist some $c_i$ such that test(c \ $c_i$ ) = FAIL
return ddMin(c \ $c_i$ , max(n-1, 2))
3. Else, if n < |c| return ddMin(c, min(2n, |c|))
```

ddmin Algorithm

- Let **c** be a failing input configuration (sequence of individual inputs)
- **test(c)** runs a test on **c** with possible outcome PASS or FAIL
- **n** is the number of chunks to split **c** into (initially **n = 2**). We will remove one chunk at the time and test the remaining input.

```
ddMin(c, n) :
1. If: |c| = 1 return c
// Otherwise, systematically remove one chunk $c_i$ at the
   time. Test the remaining input c \ $c_i$ :
2. If there exist some $c_i$ such that test(c \ $c_i$ ) = FAIL
   return ddMin(c \ $c_i$ , max(n-1, 2))
3. Else if: n < |c|
   return ddMin(c, min(2n, |c|))
4. Else: // (can't split into smaller chunks)
   return c
```

Delta Debugging

- The technique is an instance of *delta debugging*:
  - An approach to isolate failure causes by narrowing down differences (deltas) between runs
- Delta Debugging can be applied to various types of inputs such as:
  - failure-inducing program input, e.g., HTML page
  - failure-inducing user interactions e.g., the key/mouse strokes that make a program crash
  - failure-inducing changes to the program code, e.g., after a failing regression test
  - etc.

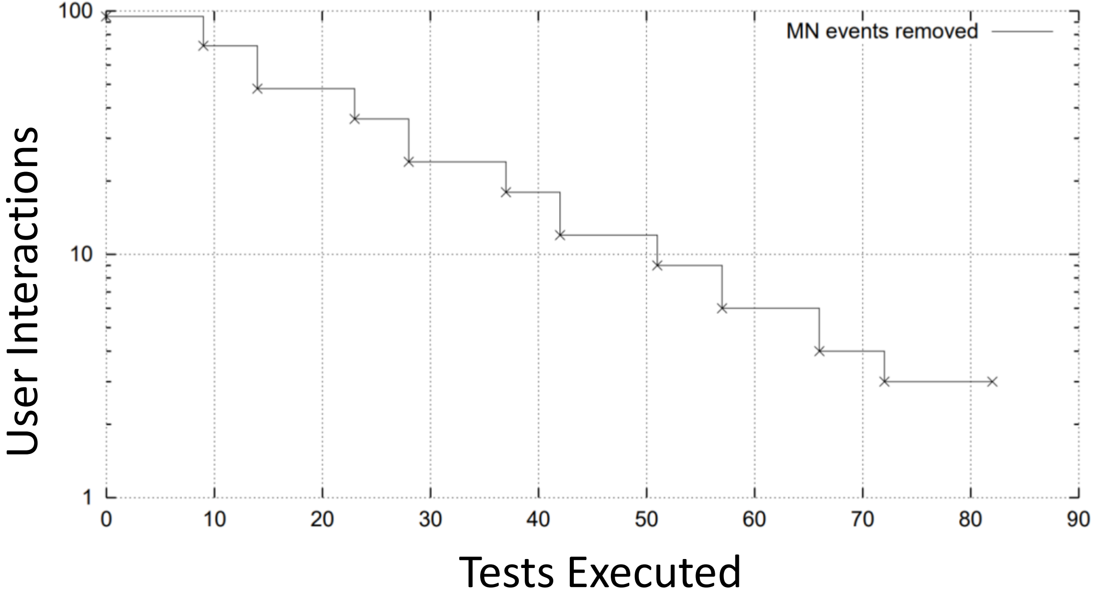
Delta Debugging

Delta Debugging gives an alternate mindset to the debugging problem. Normally, when something does not work as expected (i.e., you have a “failure”), you would naturally think: “hmmm, what’s wrong here?”. Delta Debugging takes an alternate approach: “what could NOT be wrong here?” In other words, “what is irrelevant here?” so that I can exclude those parts and put them aside to simplify things.

Delta Debugging

- After 82 tests, **`ddmin`** has simplified the user interactions to 3 events:

  1. Press `P` while holding `Alt`
  2. Press the `left mouse button` on the `Print` button
  3. Release the `left mouse button`



Relevant Reads and Resources

- Recommended Texts:
  - “Why Programs Fail”: ch1 and ch2
- https://bugzilla.mozilla.org/home
- https://www.bugzilla.org/download/
- https://logging.apache.org/log4j/
- https://www.slf4j.org/
- Recommended Texts
  - “Why Programs Fail”: ch5
- https://www-archive.mozilla.org/newlayout/bugathon.html
- TDA567/DIT082 Chalmers University of Technology http://www.cse.chalmers.se/edu/year/2018/course/TDA567/