EN.601.422 / EN.601.622

Software Testing & Debugging

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

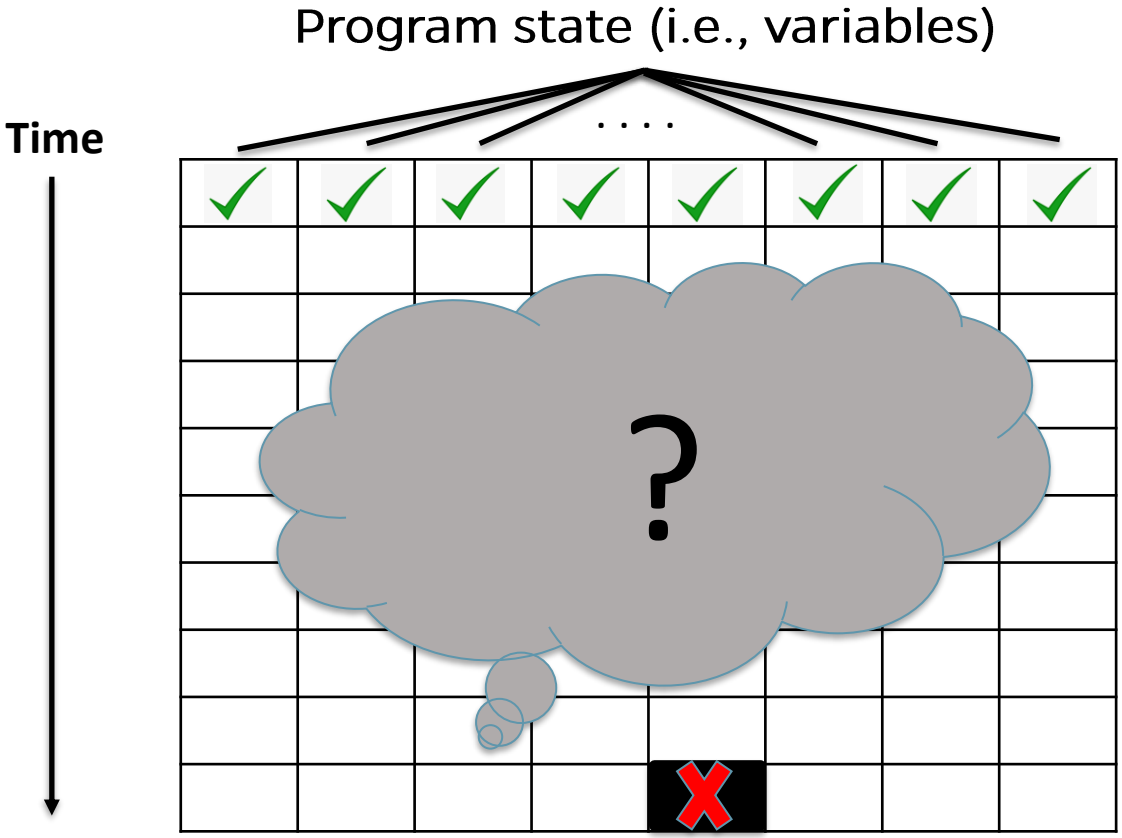
Six Stages of Debugging

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?

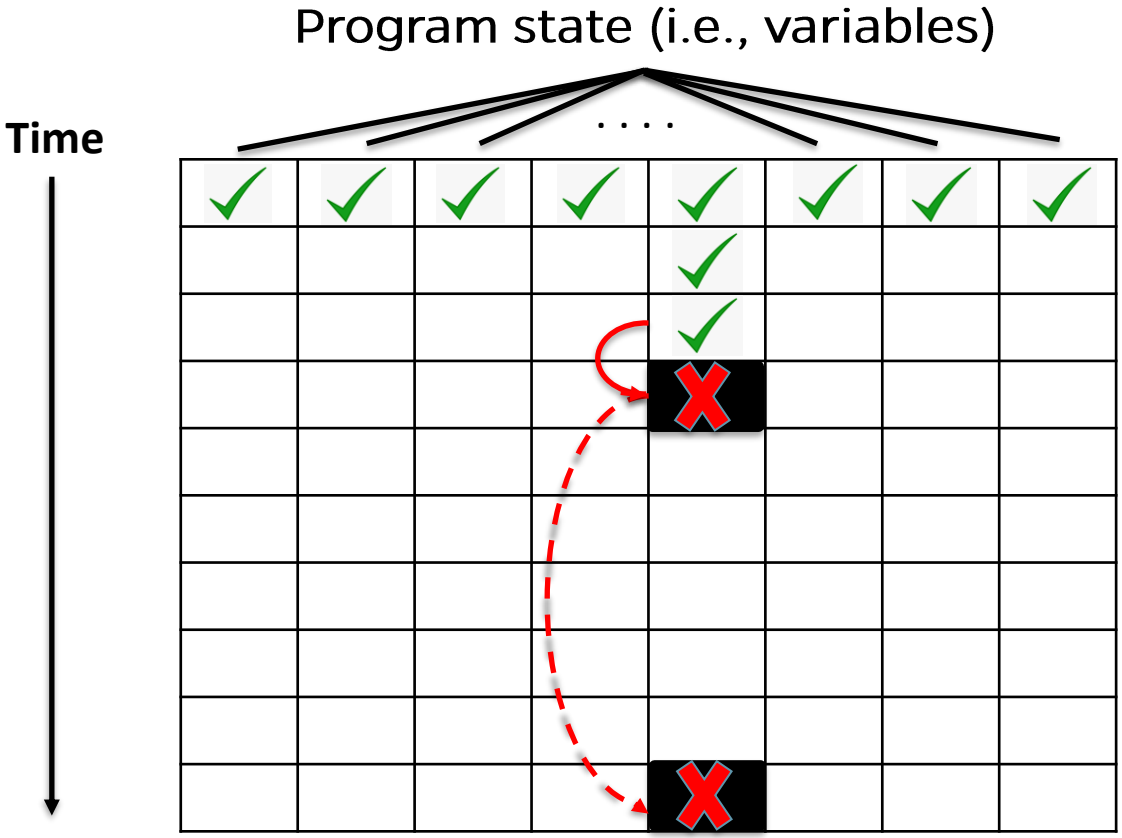
Debugging Steps

- 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



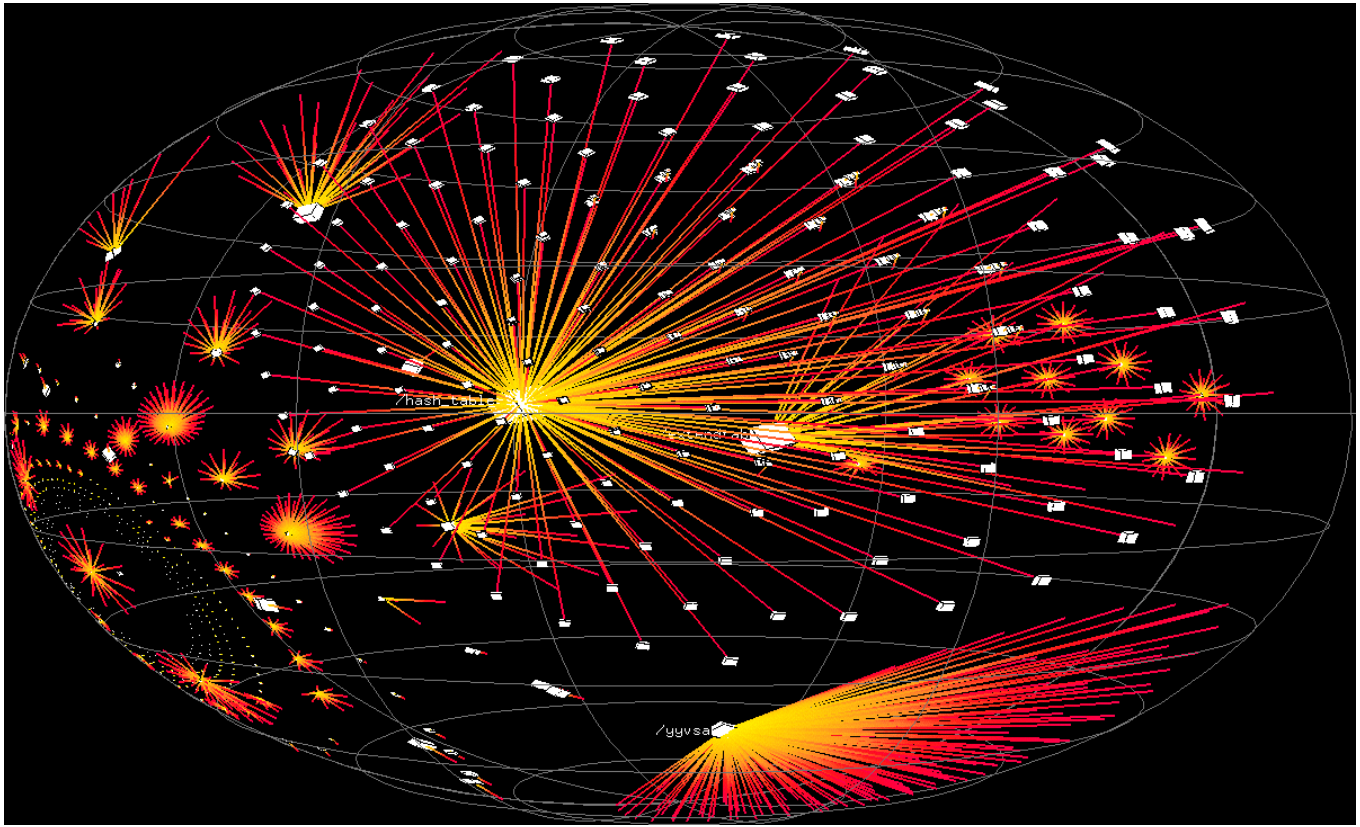
The Fault!



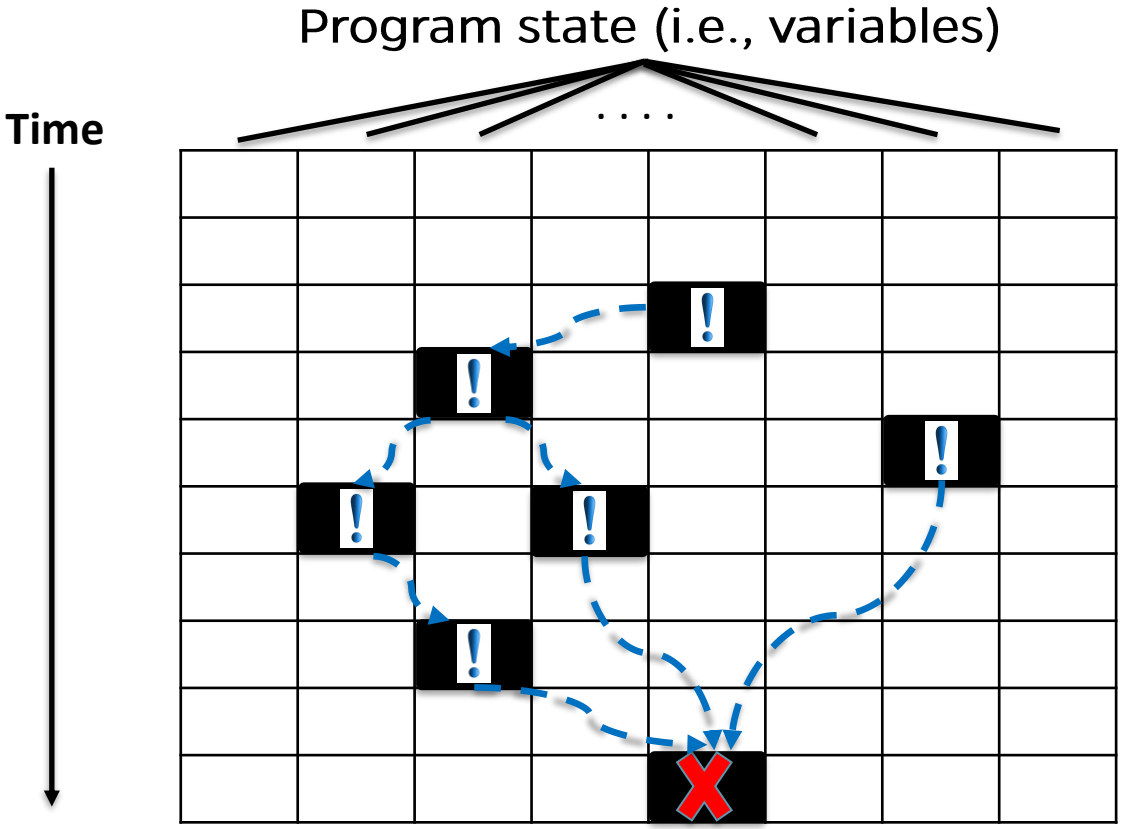
Debugging

- 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

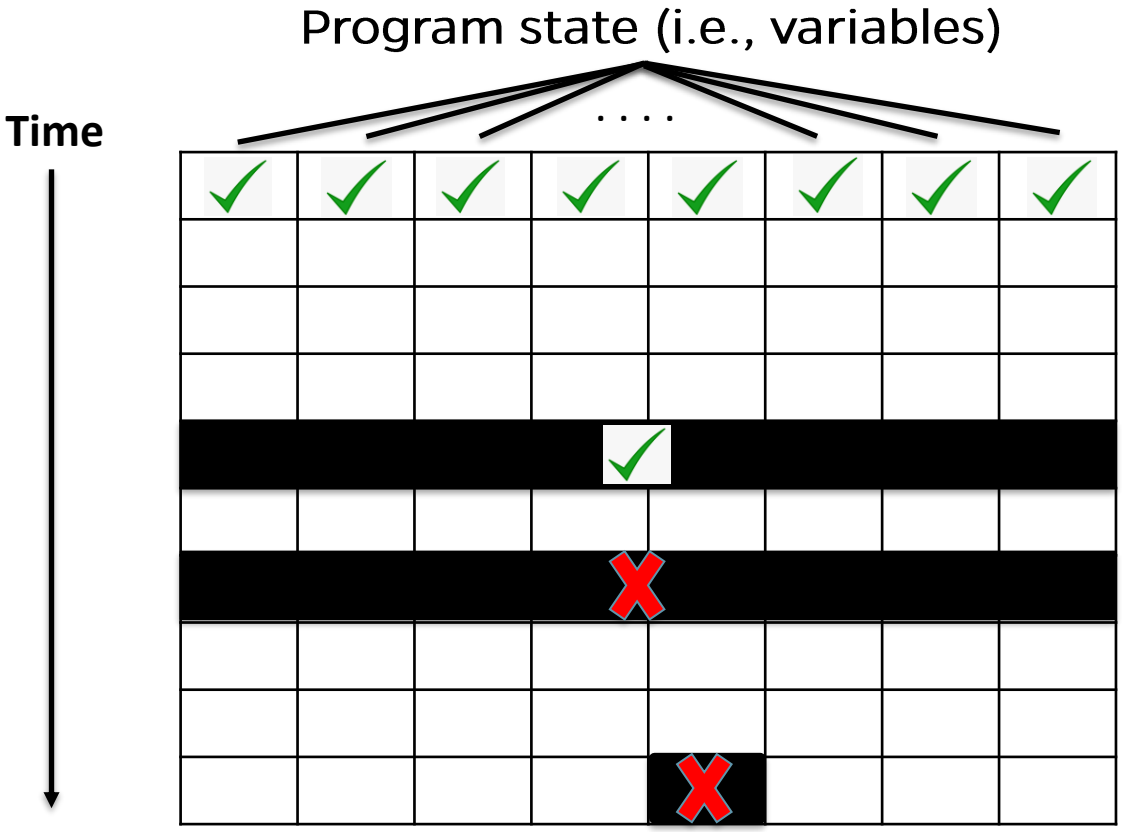
Program State Can Be Huge



Finding the Origins



Observing Transitions of States



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)
  - The infection causes a failure -- an externally visible error

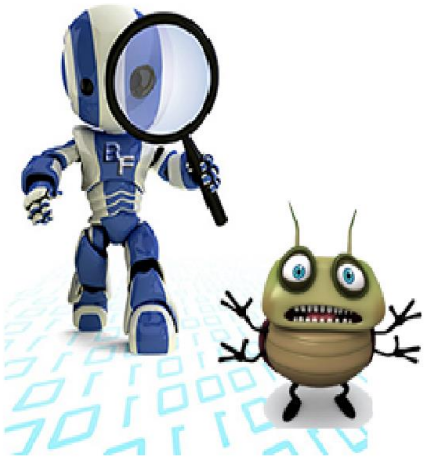
- Not every defect results in an infection, and not every infection results in a failure.

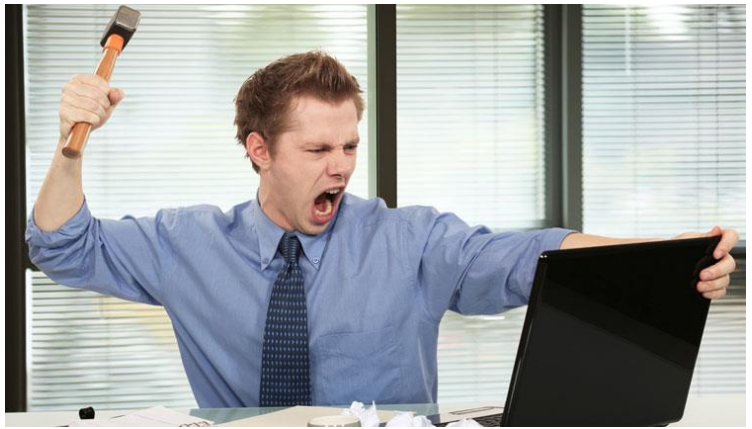
How To Debug Automatically

- 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”*





Problem Life Cycle

- The user informs the vendor about some problem.
  - “Problem” is a questionable property:
    - Failure
    - Requesting a missing feature
    - Normal behavior: *“it’s not a bug, it’s a feature!”*
- The vendor:
  - reproduces the problem
  - isolates the circumstances
  - locates and fixes the defect
  - delivers the fix to the user.

Problem Report

- A problem comes to life with a problem report
- A problem report includes all the information the vendor needs to fix the problem
- Also known as change request or *bug report*

Problem Report #1

From: me@dot.com
To: bug@product.org
Subject: Crash

Your program crashed. Please fix.

Yours faithfully,
Ali

Problem Report #2 (much better)

Revert factory settings crashes on Ubuntu 10.04

Steps to reproduce bug
1. Open Terminal in Ubuntu 10.04
2. Type mscore -F

Expected behavior: MuseScore should start with factory settings

Actual behavior: MuseScore starts and immediately crashes.

Discussion: MuseScore starts without any problems if I don't try to revert factory settings and type: mscore

MuseScore version: r 2949M, beta 2

Operating System: Ubuntu 10.04

Here's the terminal output:

```
ali@ali-laptop:~$ mscore -F
PulseAudio found, but no need to suspend. Starting
mscore.real...
Alsa_driver: the interface doesn't support mmap-based access.
init ALSA audio driver failed
init ALSA driver failed
bt_audio_service_open: connect() failed: Connection refused
(111)
bt_audio_service_open: connect() failed: Connection refused
(111)
bt_audio_service_open: connect() failed: Connection refused
(111)
bt_audio_service_open: connect() failed: Connection refused
(111)
Unable to open file "/usr/share/mscore-
0.9//sound/TimGM6mb.sf2"
Segmentation fault
```

ATTACHED: Snapshot.png, log.log, coredump.log

What To Report

- In 2008, a survey was conducted across 156 experienced Apache, Eclipse and Mozilla developers to see what makes a good bug report:
  1. **Problem history**: a description of what must be done to reproduce the problem
  2. **Diagnostic information**: e.g., log files, stack traces, core dumps etc.
  3. **Symptoms**: what happened in contrast to expected behavior (expected vs. experienced behavior)
  4. **One-line summary**: e.g., “File does not really get saved when I save a file through “File->Save” menu item”
  5. **The product release**
  6. **The operating environment**

“What Makes a Good Bug Report?” By Nicolas Bettenburg et al. in *16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008*

Problem History

- Steps needed to reproduce the problem:
  - `Create an empty file`
  - `Go to File->Print and print on the default printer…`
- If the problem cannot be reproduced, it is unlikely to be fixed
- Simplify if you can: which steps are relevant?

Diagnostic Information

- A failure might occur after a long series of events
  - The user might not remember all the necessary steps to reproduce the failure
- Many programs can be set up to record important events in a log file
- When reporting an issue, it is certainly helpful to submit such log files to the vendor

Core Dump

- a core dump (a.k.a crash dump, memory dump, or system dump) is a very useful debugging resource:
  - “consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.”

Expected Behavior

- What should have happened according to the user:
  - **Expected**: The program should have printed the document.

- **Reality check**: what's the understanding of the user?

Observed Behavior

- The symptoms of the problem — in contrast to the expected behavior
- The program crashed with the following information

```
*** STACK DUMP OF CRASH (LemonyOS)

 Back chain  ISA  Caller
 00000000    SPC  0BA8E574
 03EADF80    SPC  0B742428
 03EADF30    SPC  0B50FDDC  PrintThePage+072FC
SnicketPC unmapped memory exception at
      0B512BD0 PrintThePage+05F50
```

One-line Summary

- Captures the essence of the problem:
  - *“The program crashes when printing using the default printer”*

Product Release

- Typically, some version number or otherwise a unique identifier
- Required to reproduce the exact version:
  - e.g., Perfect Publishing Program 1.1 (Build 7E47)
- Generalize: Does the problem occur only in this release?

Operating Environment

- Typically, version information about the operating system
- Can be simple (”Windows 7”) or complex (”Debian Linux ‘Sarge’ with the following packages . . .”)
- Generalize: In which environments does the problem occur?

Things To Avoid

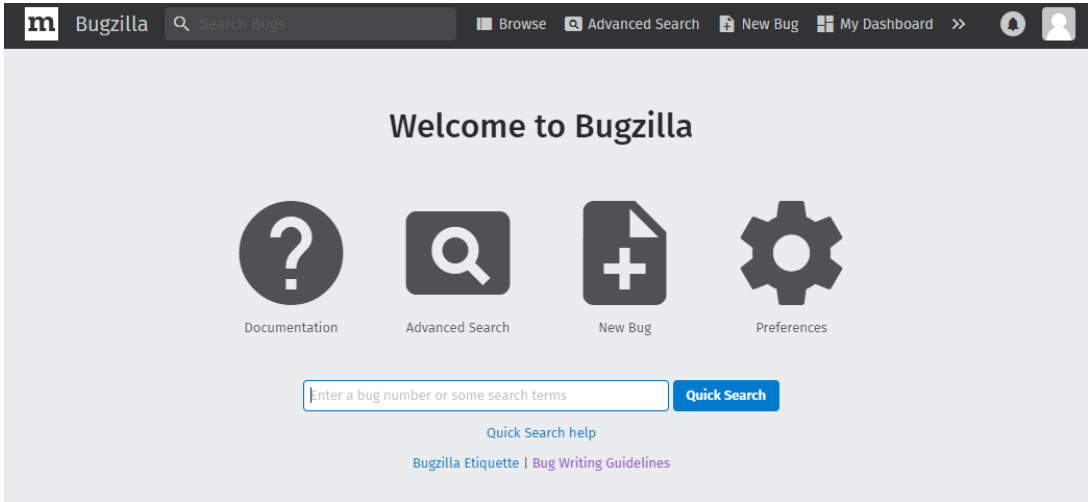
- Humor
  - PPP (oops, gotta go to the restroom :-) …
- Sarcasm
  - Here's yet another "never-to-be-fixed" bug
- Attacks
  - If you weren't too incompetent to grasp …

Managing Problems

- Alternative #1: A Problem File
  - Only one person at a time can work on it
  - History of earlier (fixed) problems is lost
  - Does not scale
- Alternative #2: A Problem Database

Bug Database – a.k.a Bug Tracking System

- A bug tracking system is a necessary component of a good software development infrastructure
- Consistent and proper utilization of bug tracking systems is considered one of the *"hallmarks of a good software team"*



Severity

- Enhancement. A desired feature
- Trivial. Cosmetic problem
- Minor. Problem with easy workaround
- Normal. “Standard” problem
- Major. Major loss of function
- Critical. Crashes, loss of data or memory
- Showstopper. Blocks development

Priority

- Every new problem gets a priority
- The higher the priority, the sooner the problem will be addressed
- Priority is independent from severity
- Prioritizing problems is the main tool to control development and problem solving

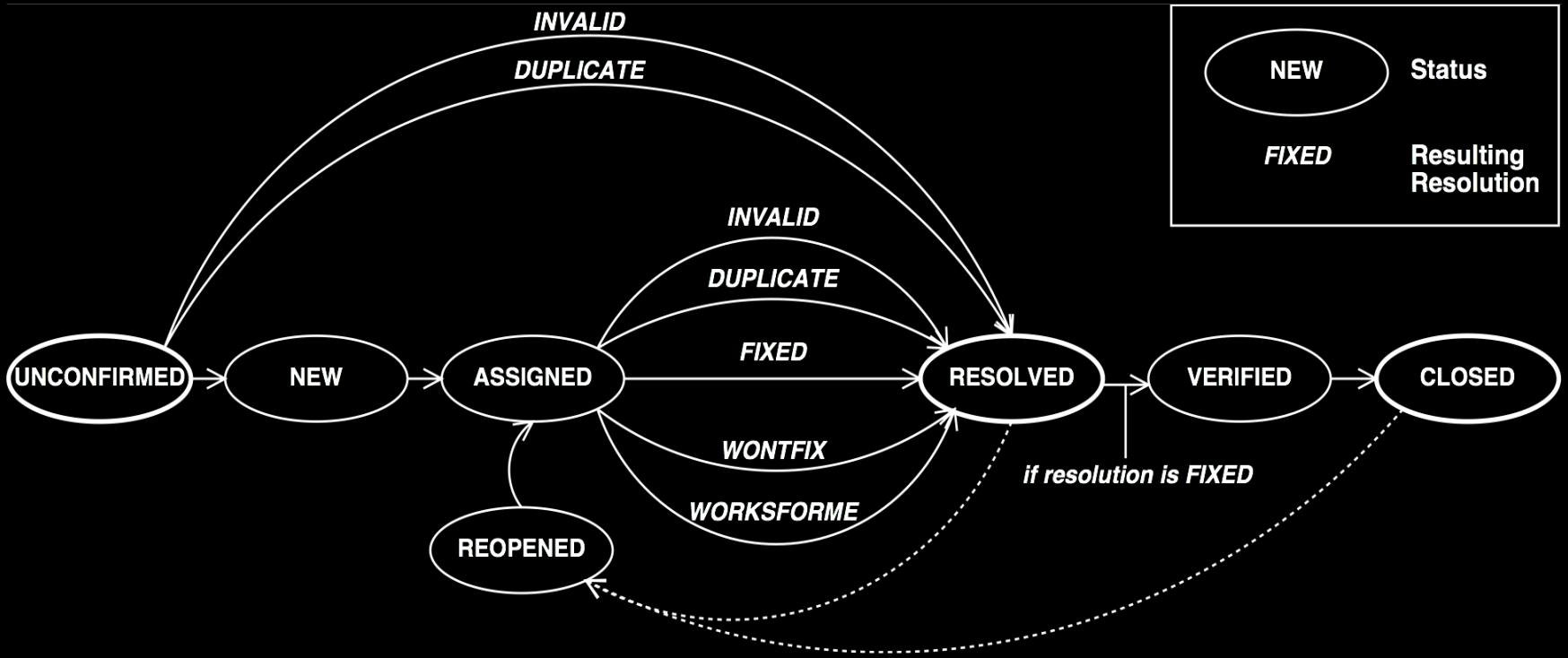
Identity

- Every new problem gets an identifier (also known as PR number, bug number, ticket number etc.)
- The identifier is used in all documents during the debugging process:
  - Subject: Is `PR #3427` fixed yet?

Notification

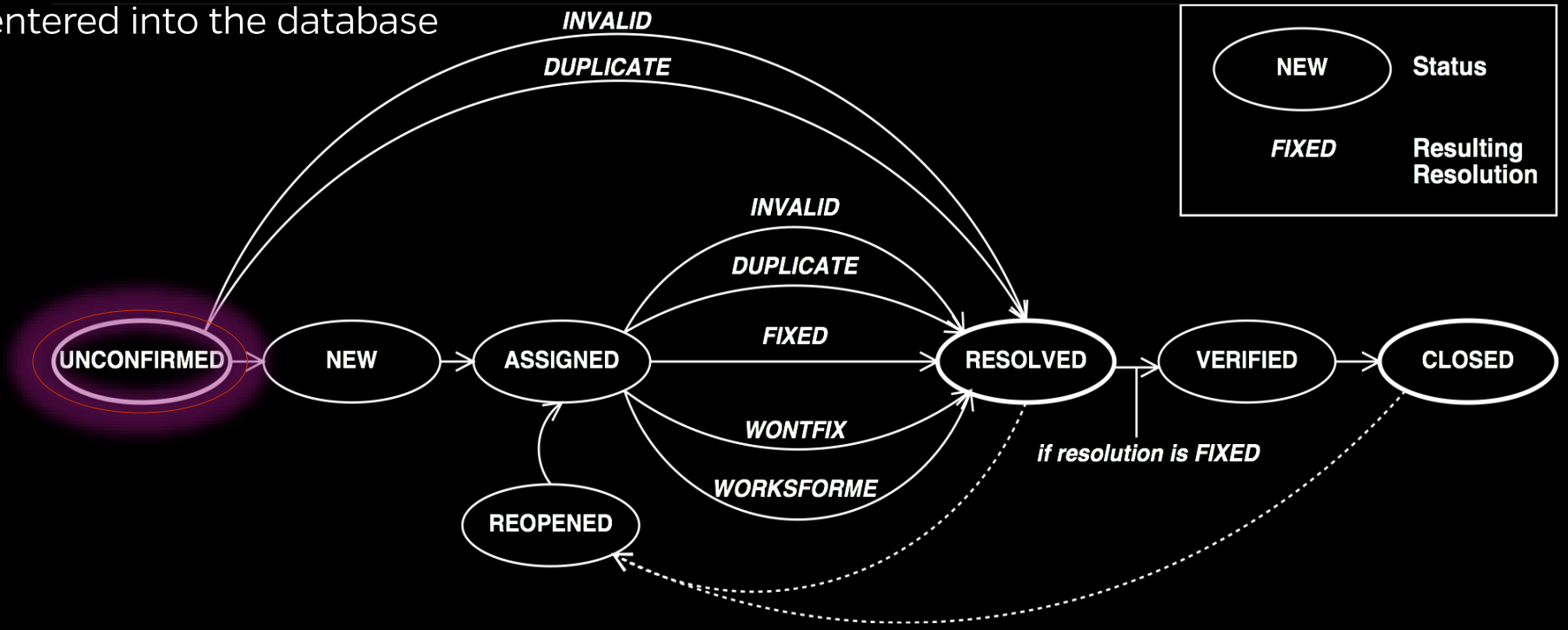
- Developers can attach an e-mail address to a problem report; they will be notified every time the report changes.
- Users can do so, too.

The Problem Lifecycle



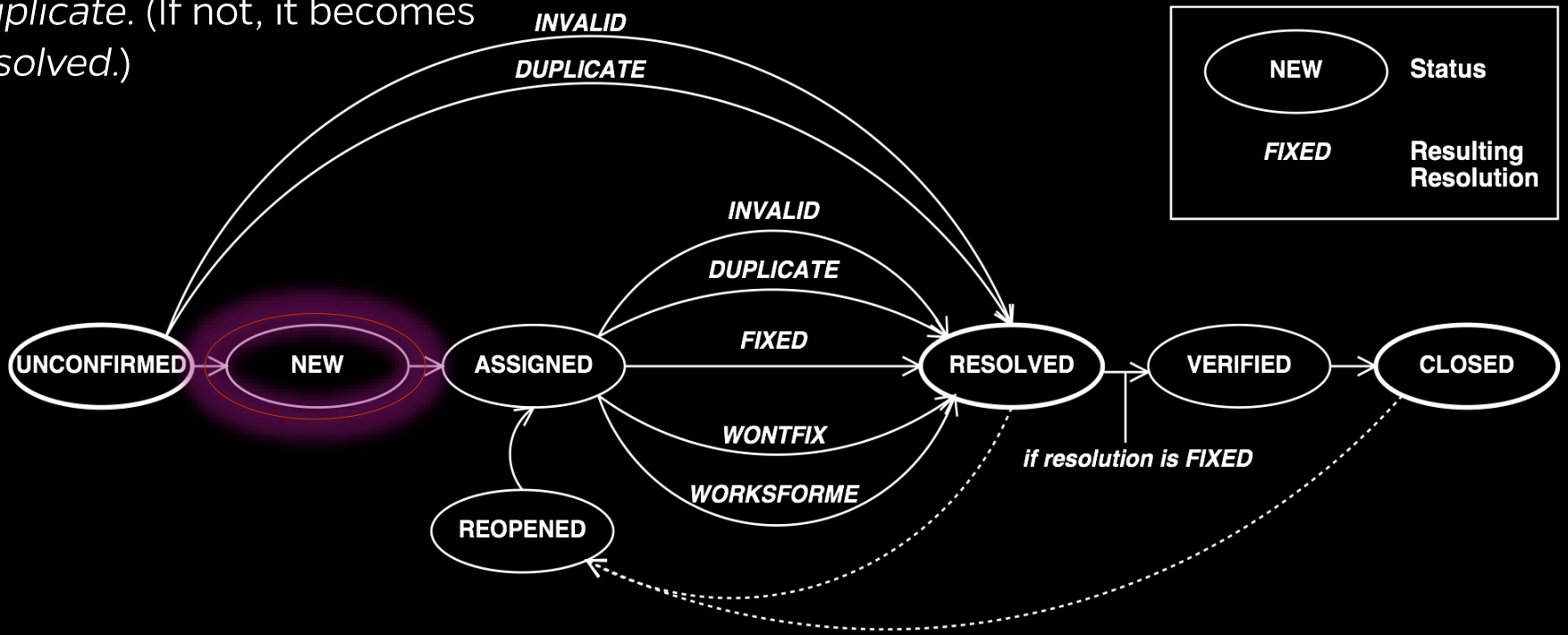
Unconfirmed

The problem report has just been entered into the database



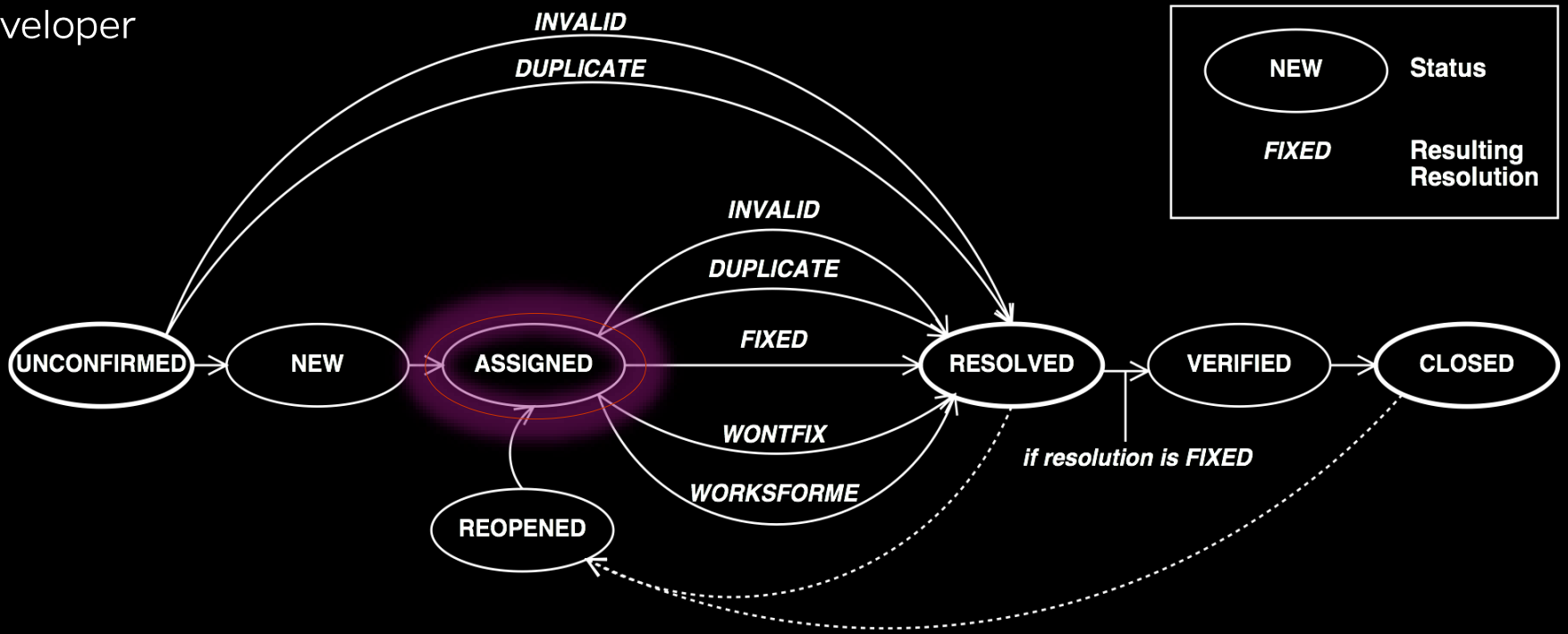
New Problem

The report is *valid* and not a *duplicate*. (If not, it becomes *resolved*.)



Assigned Problem

The problem is assigned to a developer

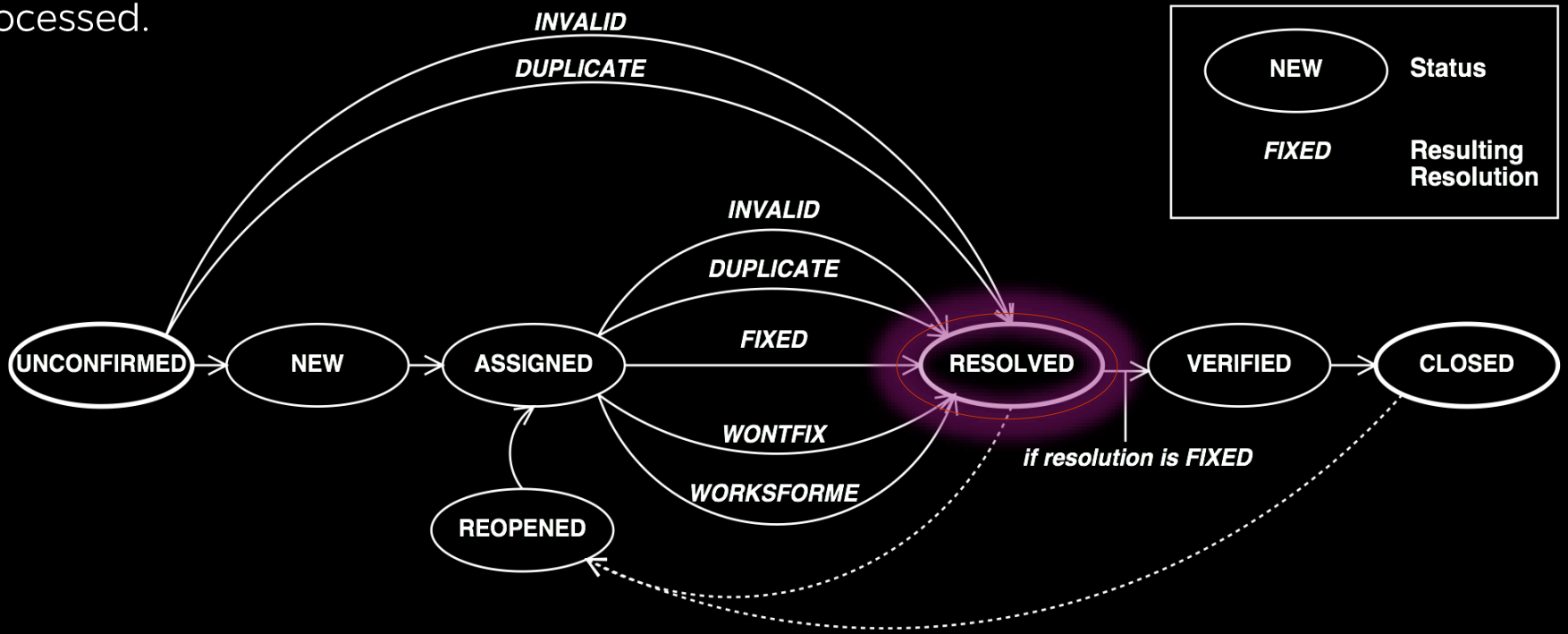


Resolution

- **FIXED**: The problem is fixed.
- **INVALID**: The problem is not a problem.
- **DUPLICATE**: The problem already exists.
- **WONTFIX**: Will never be fixed (for instance, because the problem is a feature)
- **WORKSFORME**: Could not be reproduced.

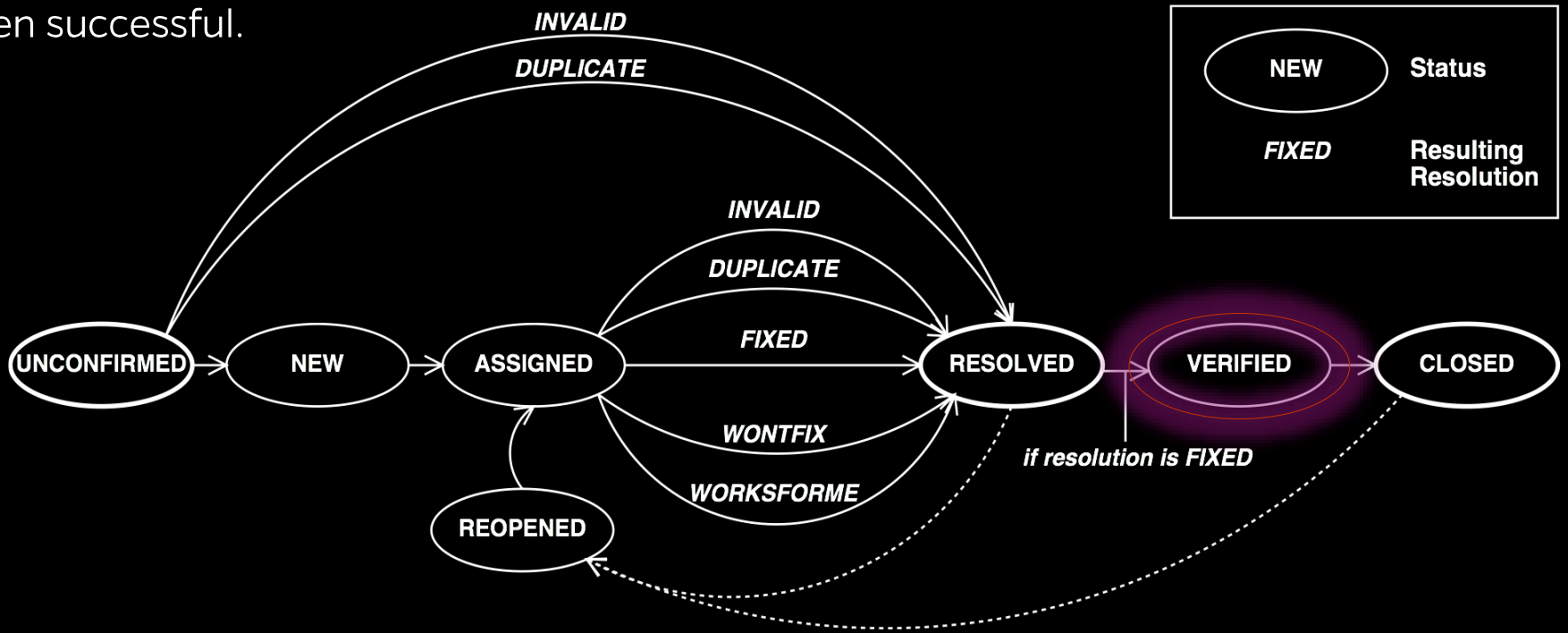
Resolved Problem

The problem report has been processed.



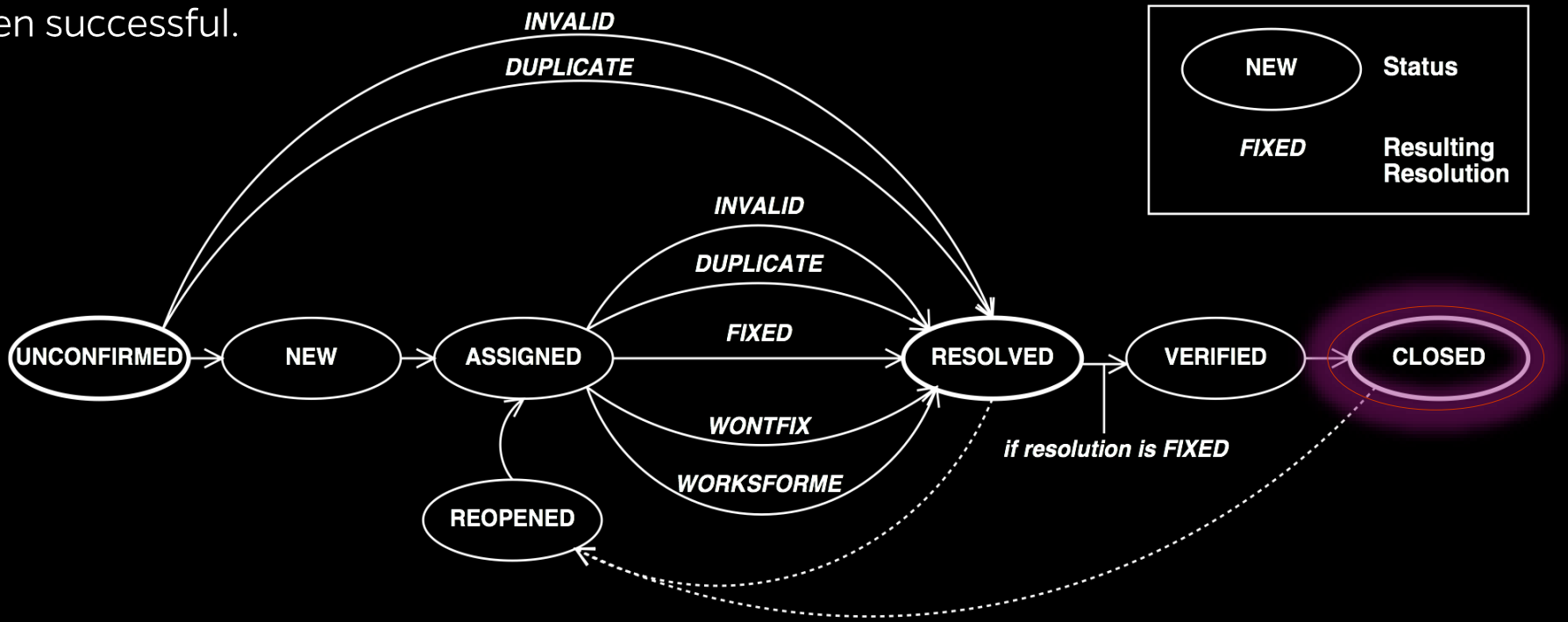
Verified Problem

The problem is fixed; the fix has been successful.



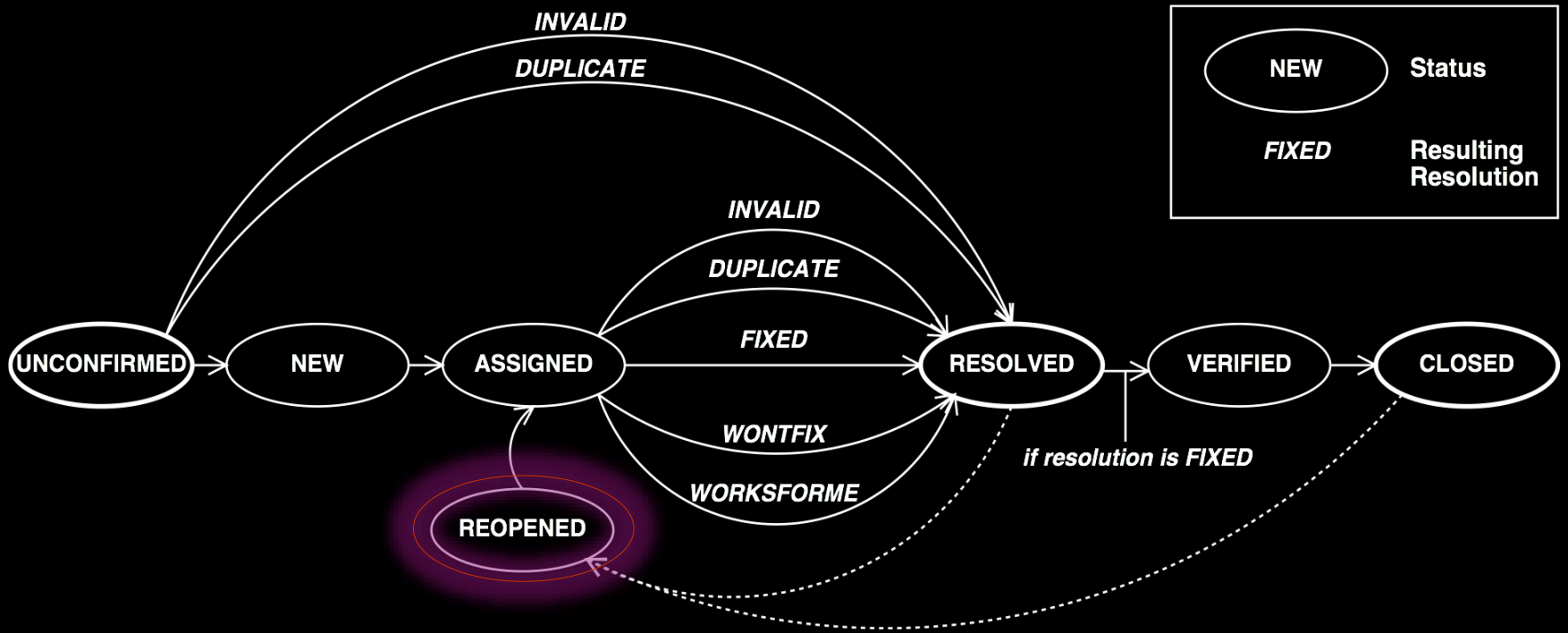
Closed Problem

The problem is fixed; the fix has been successful.



Reopened Problem

Oops – there we go again :-(



Management

- Who enters problem reports?
- Who classifies problem reports?
- Who sets priorities?
- Who takes care of the problem?
- Who closes issues?

The SCCB

- At many organizations, a **software change control board** is in charge of these questions:
  - Assess the impact of a problem
  - Assign tasks to developers
  - Close issues …

Problem-driven Development

- The whole development can be organized around the problem database:
  - Start with one single problem:
  - “The product isn’t there”
  - Decompose into sub-problems
  - Ship when all problems are fixed

BUG STORY 3: Tracking Milk Issues at Microsoft

The following bug report is purported to originate from Microsoft's Excel group from 1994. Aliases have been removed. The $T:$ indicates that the person was a tester, whereas $D:$ stands for developer and $P:$ for program manager.

```
------------------- ACTIVE - 05/12/94 - T:XXXXXX ----------------------
: Go to the kitchen
: Grab a Darigold chocolate milk carton
: Read the ingredients list

--! Either Darigold has discovered a chocolate cow, or something's
  missing from the ingredients list. It only lists milk, vitamin A,
  and vitamin D.  So where does the chocolate/sugar flavor come from?
------------------- ACTIVE - 05/12/94 - T:XXXXXX ----------------------
Moo info:
: Grab a Darigold 2% milk carton (NOT chocolate)
: Read the ingredients
--! Says it contains Cocoa, Sugar, Guar gum ...
Looks like the Chocolate
  and 2% ingredient lists have been swapped.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXXXXX ------------
looks like an internals problem?
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
UI Problem. I'll take it.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
They don't make milk at the Issaquah Darigold. Calling Ranier Ave.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
I can't repro. Do you have the wrong MILKINTL.DLL?
-------------- ASSIGNED to D:XXXXX - 05/12/94 - T:XXXXXXXX ------------
By design? I think new US health labeling went into effect this month.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
Wrong Department. Transferred from Distribution to Production.
Left voice mail for "Frank".
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
Reproduces in the Development Kitchen. Need a native
build of the Kitchen ...
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX ---------------
This is a feature. IntelliSense labeling knew that you didn't want to feel
guilty about the chocolate in the milk, so it didn't list it on the box.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXX ----------------
Recommend postpone. Reading the ingredients is not a common user
scenario ...
-------------- RESOLVED - WON'T FIX - 05/12/94 - P:XXXXX --------------
Fixing the package is just a band-aid. We need to come up with a solution
that addresses the real problem in 96. My recommendation is
chocolate cows.

Please close and assign to DARIGOLD.
```

Tests and Problem Database

- Some test fails. Should we enter the problem into the database?
  - You could, but in general it is better if you don't because test cases make problem reports obsolete:
    - Test failures occur way too more frequently than user reports → floods the database
    - If the test case is part of the standard test suite(s) → Ideally is part of continuous integration pipeline → no need to be entered in the database
    - Maybe a failed test case can be easily communicated with the/a developer and get fixed

Clutter Management

- Large problem databases contain garbage
- Get rid of duplicates by
  - simplifying bug reports
  - asking submitters to search first
- Get rid of obsolete problems by searching for old ones that are no longer relevant

Summary

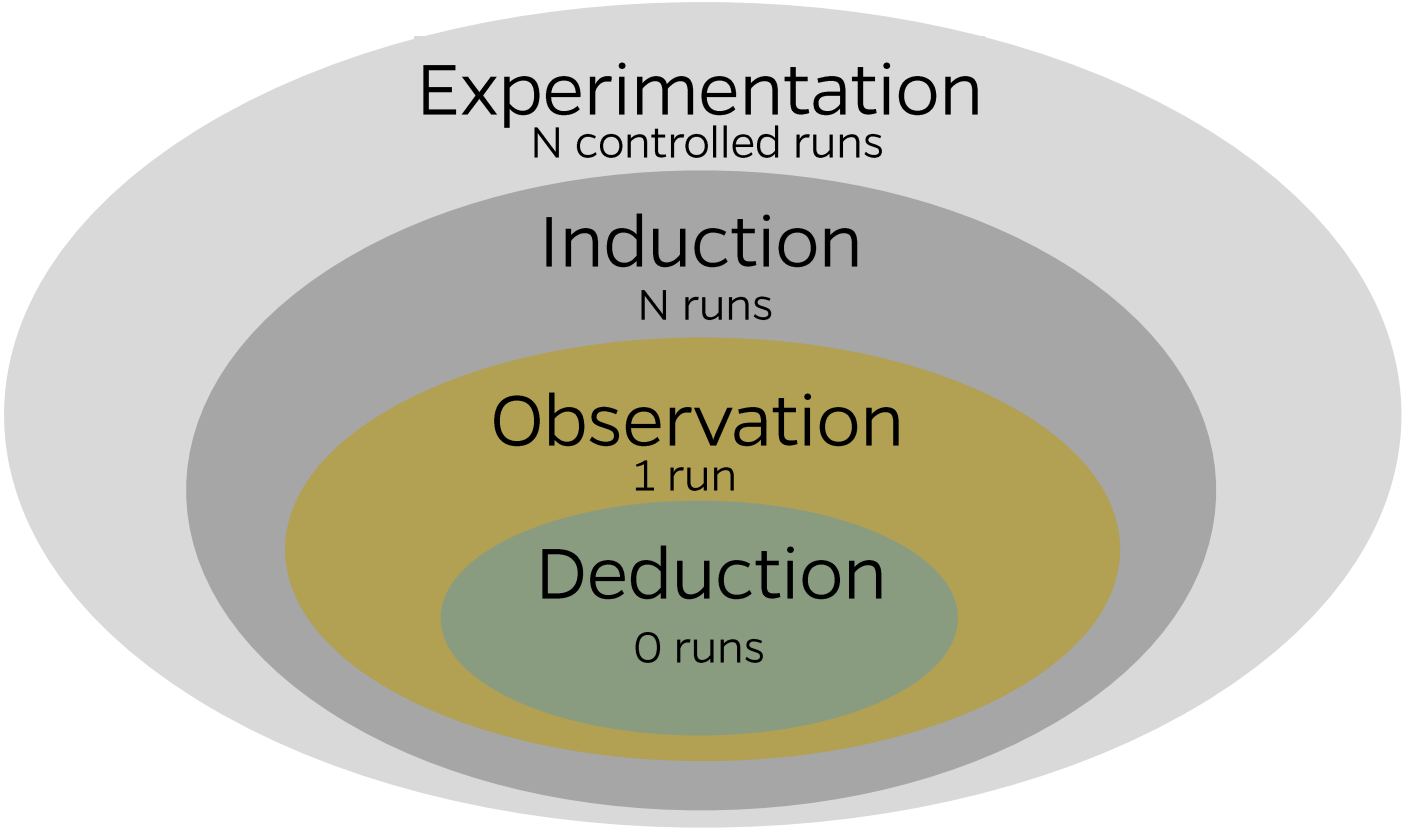
- An effective problem report ...
  - is well-structured
  - is reproducible
  - has a descriptive one-line summary
  - is as simple and general as possible
  - is neutral and stays with the facts
  - contains both expected and actual behavior/output
- A typical problem life cycle starts with an *unconfirmed* status, It ends with a *closed* status and a specific resolution (such as *fixed* or *worksforme*)
- Typically, a software change control board organizes priorities and assignments

Tool Support

- Bugzilla
- Jira
- TRAC
- ISSUETRACKER
- PHPBugTracker
- Github Issues
- Monday.com
- etc.

https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems

Debugging is Reasoning



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

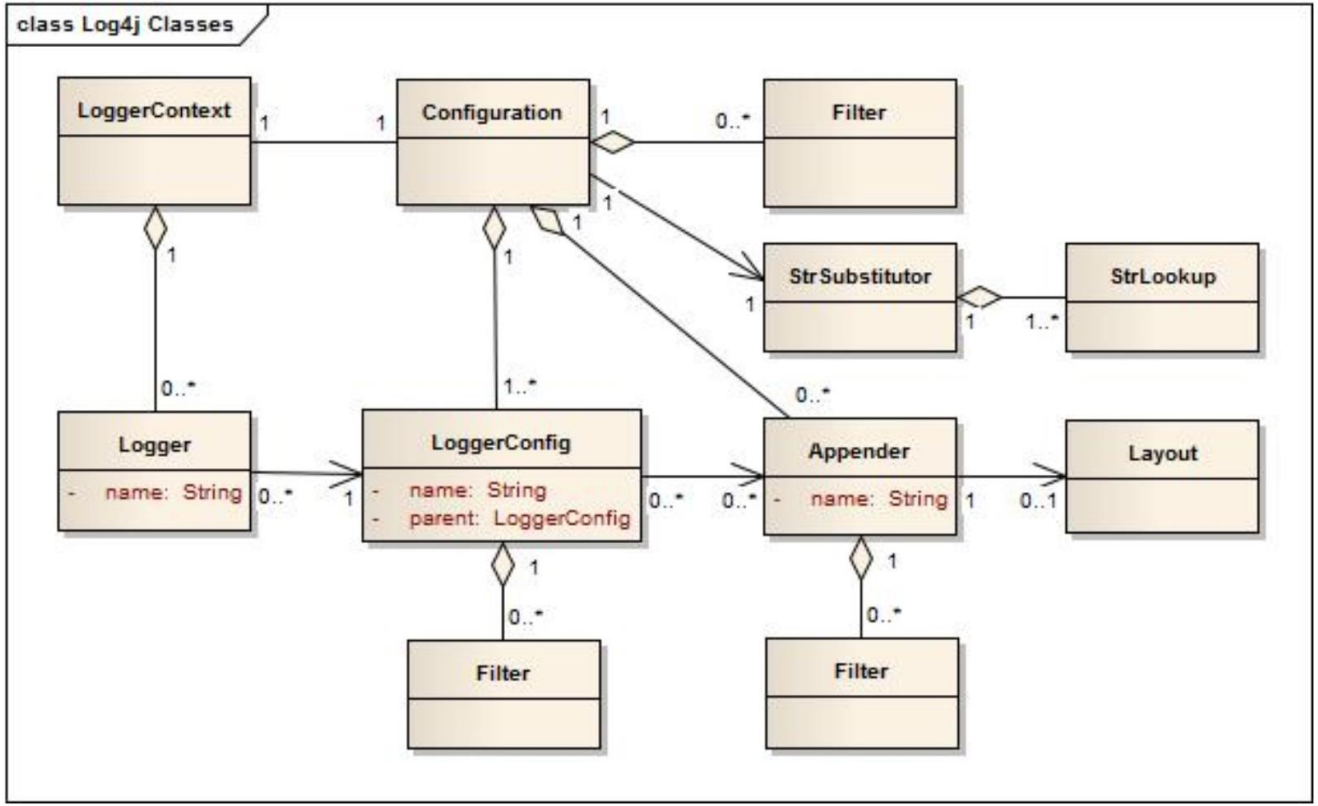
- To address some of the issues discussed:
  - Make a dedicated function to log the debugging-related messages
    - Example: write a function named dprintln(String) and call that when logging for debugging
  - Turn the function off when releasing/in production
    - Calculating arguments and calling 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



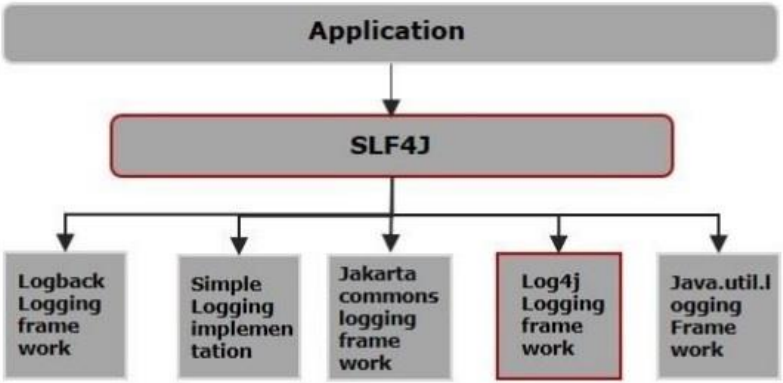
Log Levels

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



Relevant Reads and Resources

- Recommended Texts:
  - “Why Programs Fail”: ch1 and ch2
- https://bugzilla.mozilla.org/home
- https://www.bugzilla.org/download/