Quality Assurance 4

Testing 2, Program analysis

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

Testing

- Executing the program with selected inputs in a controlled environment (dynamic analysis)
- Goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

"Testing shows the presence,
not the absence of bugs

Edsger W. Dijkstra 1969

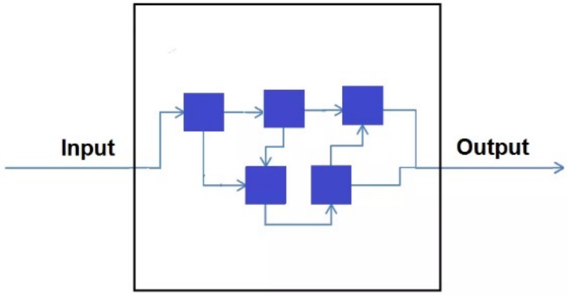
What are we covering?

- 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

Testing Levels

- Unit testing
- Integration testing
- System testing

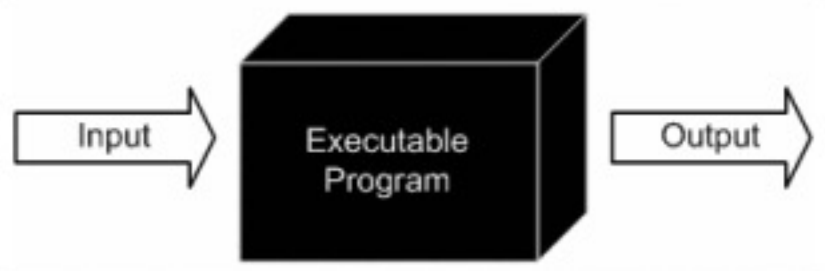
White box testing



Tests internal structures or workings of an application, as opposed to its functionality.

- Unit Test
- Testing for Memory Leaks
- Penetration Testing
  - ***"What would a cybercriminal do to harm my organization' computer systems, applications, and network?"***

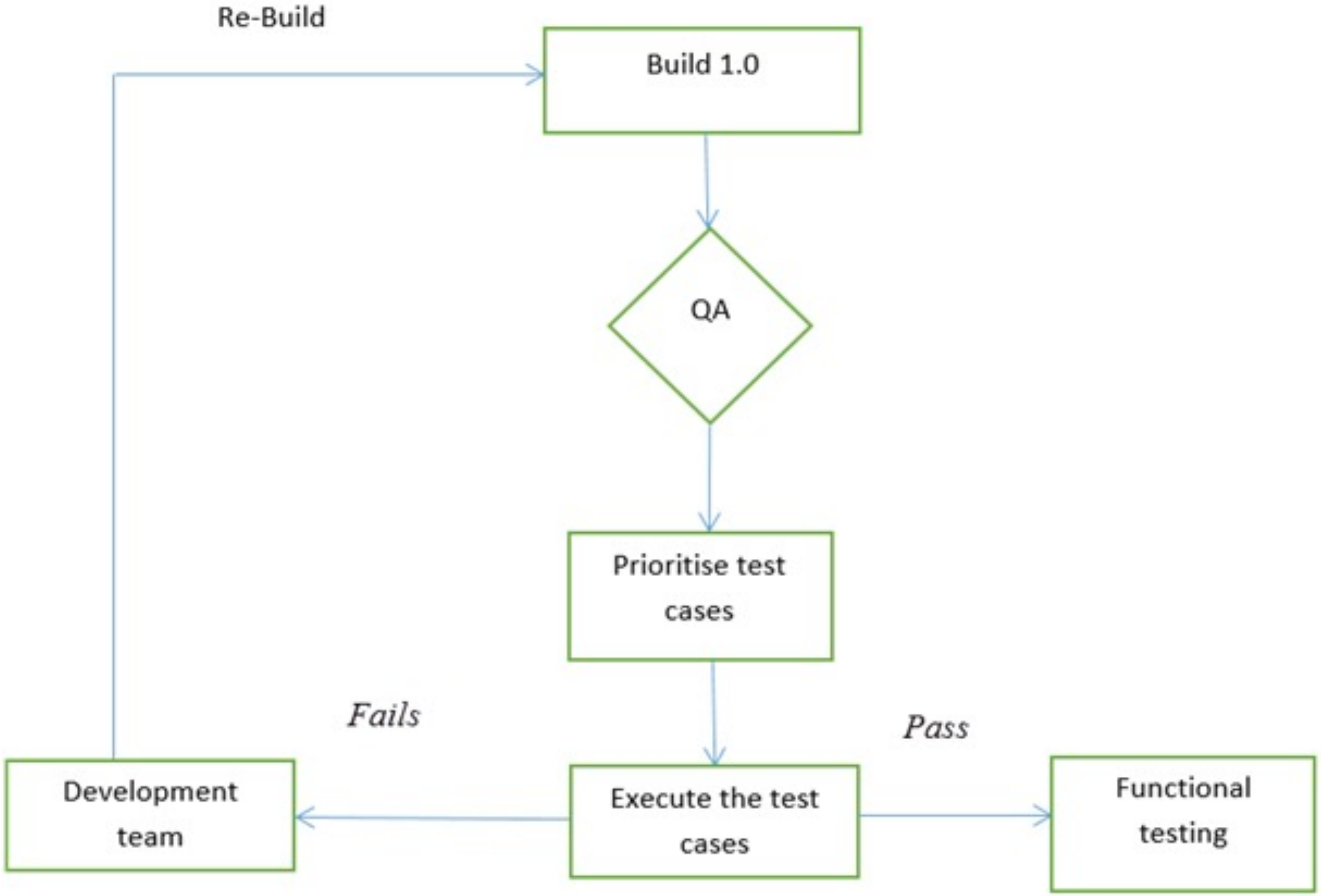
Black box testing



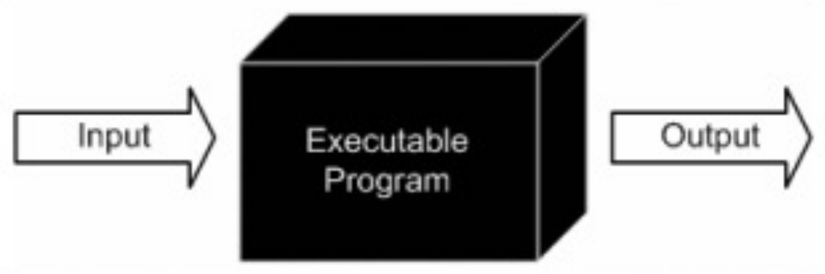
- Functionality of application is tested without looking at the implementation details
- Types
  - **Functional Testing**
    - Smoke Testing
    - Regression Testing
    - …
  - **Non-Functional Testing**
    - Performance Testing
    - Compatibility Testing
    - Stress Testing

Smoke Testing

- Determines whether the deployed software build is stable or not.
- We perform smoke testing on a new build.



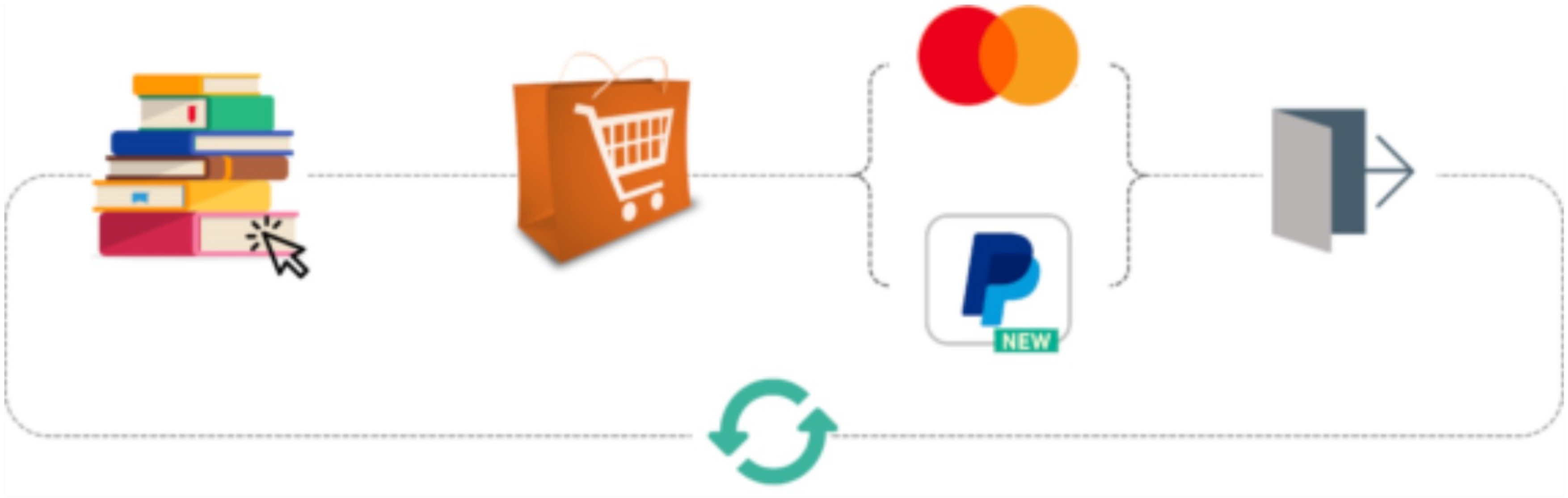
Black box testing



- Functionality of application is tested without looking at the implementation details
- Types
  - **Functional Testing**
    - Smoke Testing
    - Regression Testing
    - …
  - **Non-Functional Testing**
    - Performance Testing
    - Compatibility Testing
    - Stress Testing

Regression testing

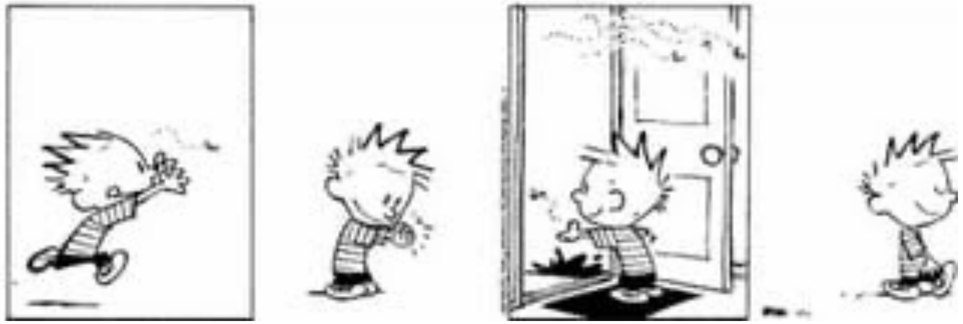
- Ensure that a small change in one part of the system does not break existing functionality elsewhere in the system.



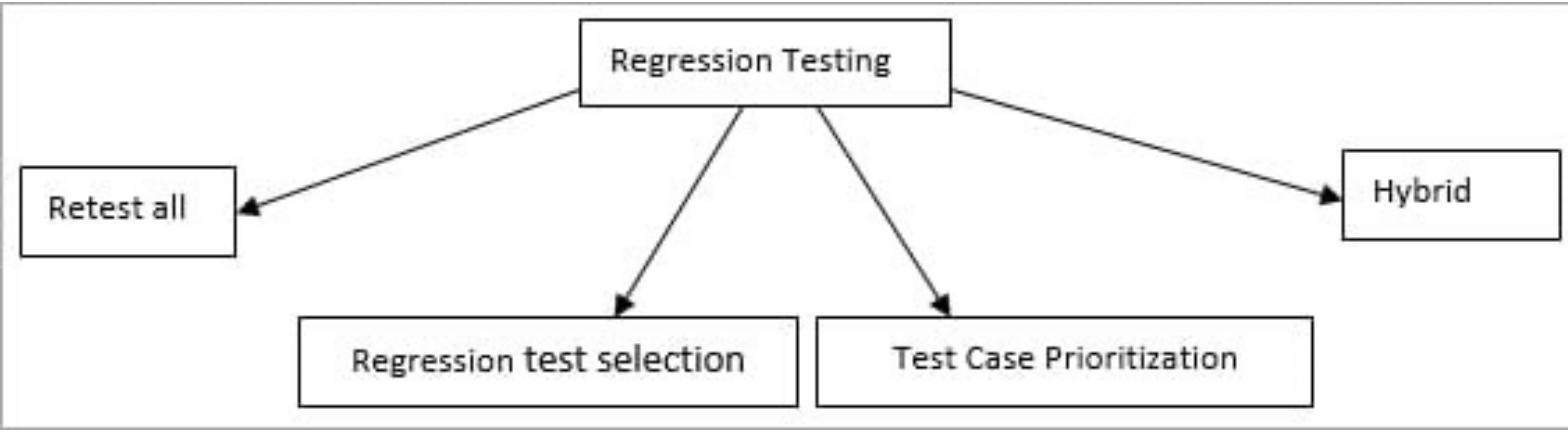
Regression testing

- Ensure that a small change in one part of the system does not break existing functionality elsewhere in the system.
- Application scenario:
  - When new functionalities are added
  - In case of change requirements
  - When there is a defect fix
  - When there are performance issues
  - In case of environment changes
  - When there is a patch fix

Regression:
"when you fix one bug, you introduce several newer bugs."



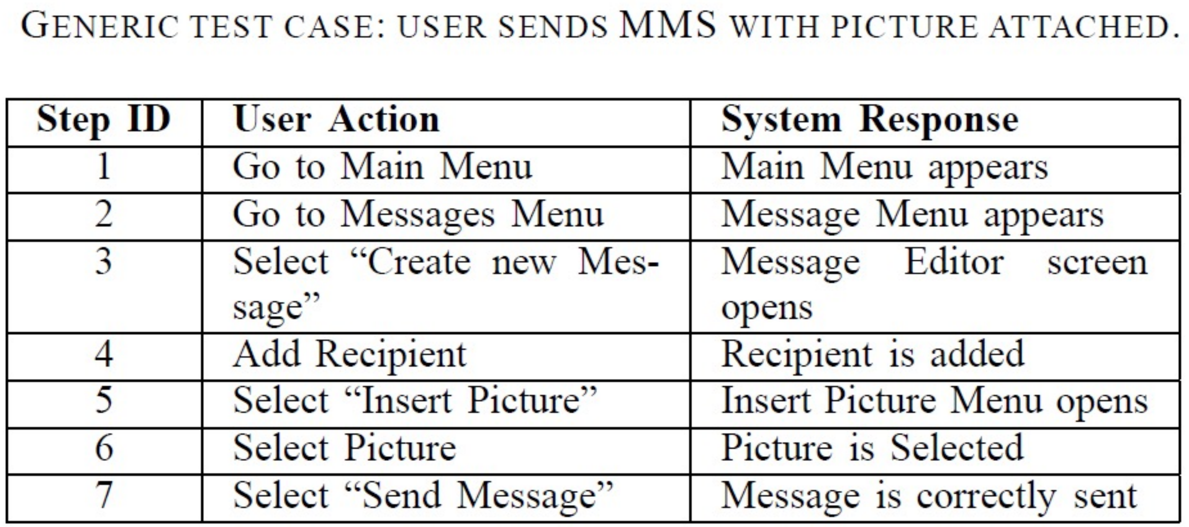
4 Types of Regression Testing



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

Manual Testing?



GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select "Create new Message"	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select "Insert Picture"	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select "Send Message"	Message is correctly sent

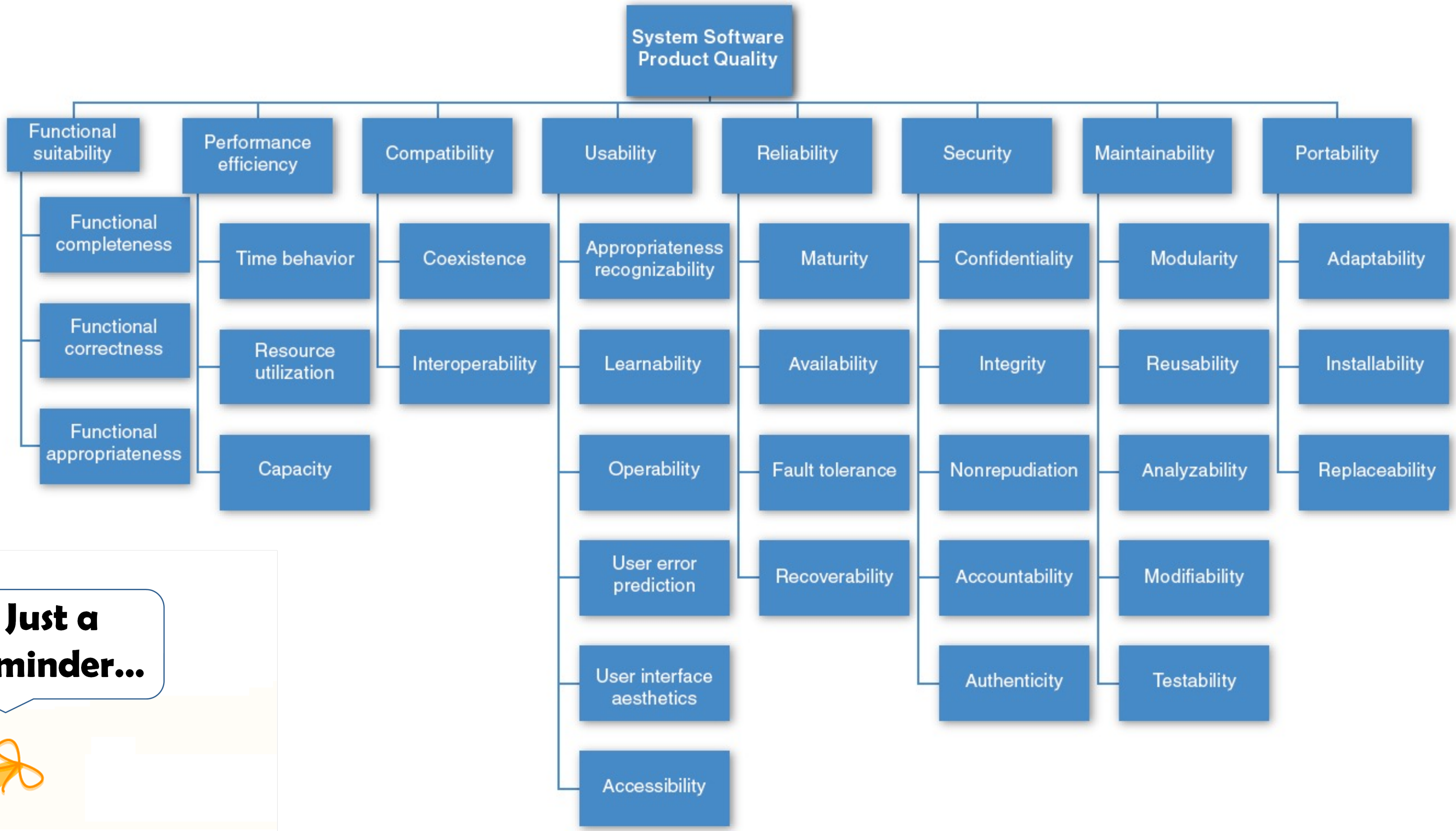
- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

A/B testing

What are we covering?

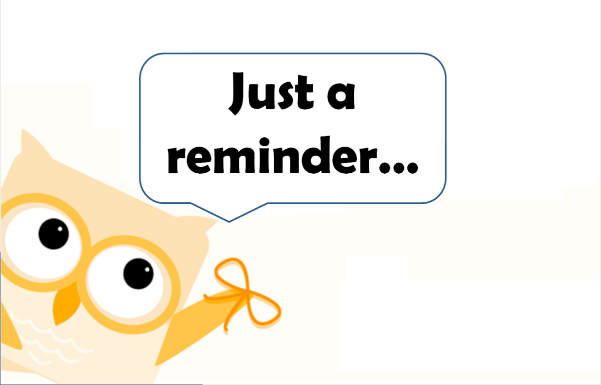
- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz,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

Quality Attributes



System Software Product Quality

- Functional suitability
  - Functional completeness
  - Functional correctness
  - Functional appropriateness
- Performance efficiency
  - Time behavior
  - Resource utilization
  - Capacity
- Compatibility
  - Coexistence
  - Interoperability
- Usability
  - Appropriateness recognizability
  - Learnability
  - Operability
  - User error prediction
  - User interface aesthetics
  - Accessibility
- Reliability
  - Maturity
  - Availability
  - Fault tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Nonrepudiation
  - Accountability
  - Authenticity
- Maintainability
  - Modularity
  - Reusability
  - Analyzability
  - Modifiability
  - Testability
- Portability
  - Adaptability
  - Installability
  - Replaceability

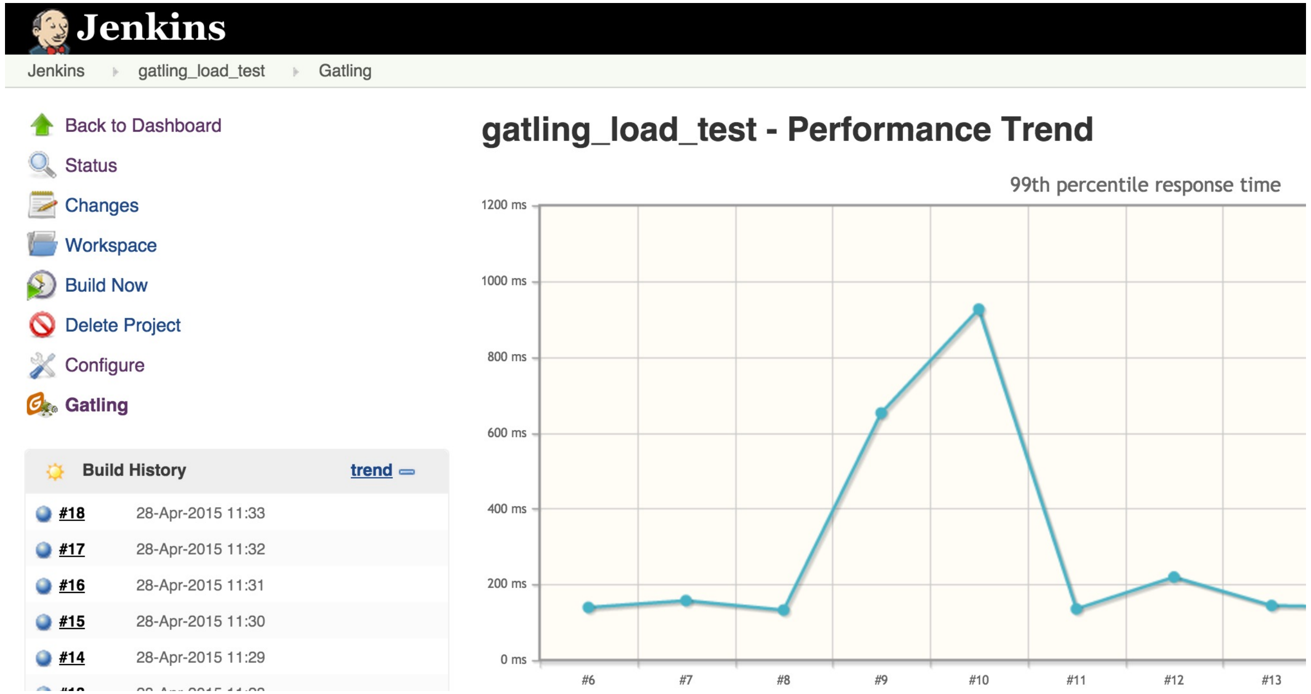


Performance Testing

- Specification? Oracle?
- Test harness? Environment?
- Nondeterminism?
- Unit testing?
- Automation?
- Coverage?

Unit and regression testing for performance

- Measure execution time of critical components
- Log execution times and compare over time

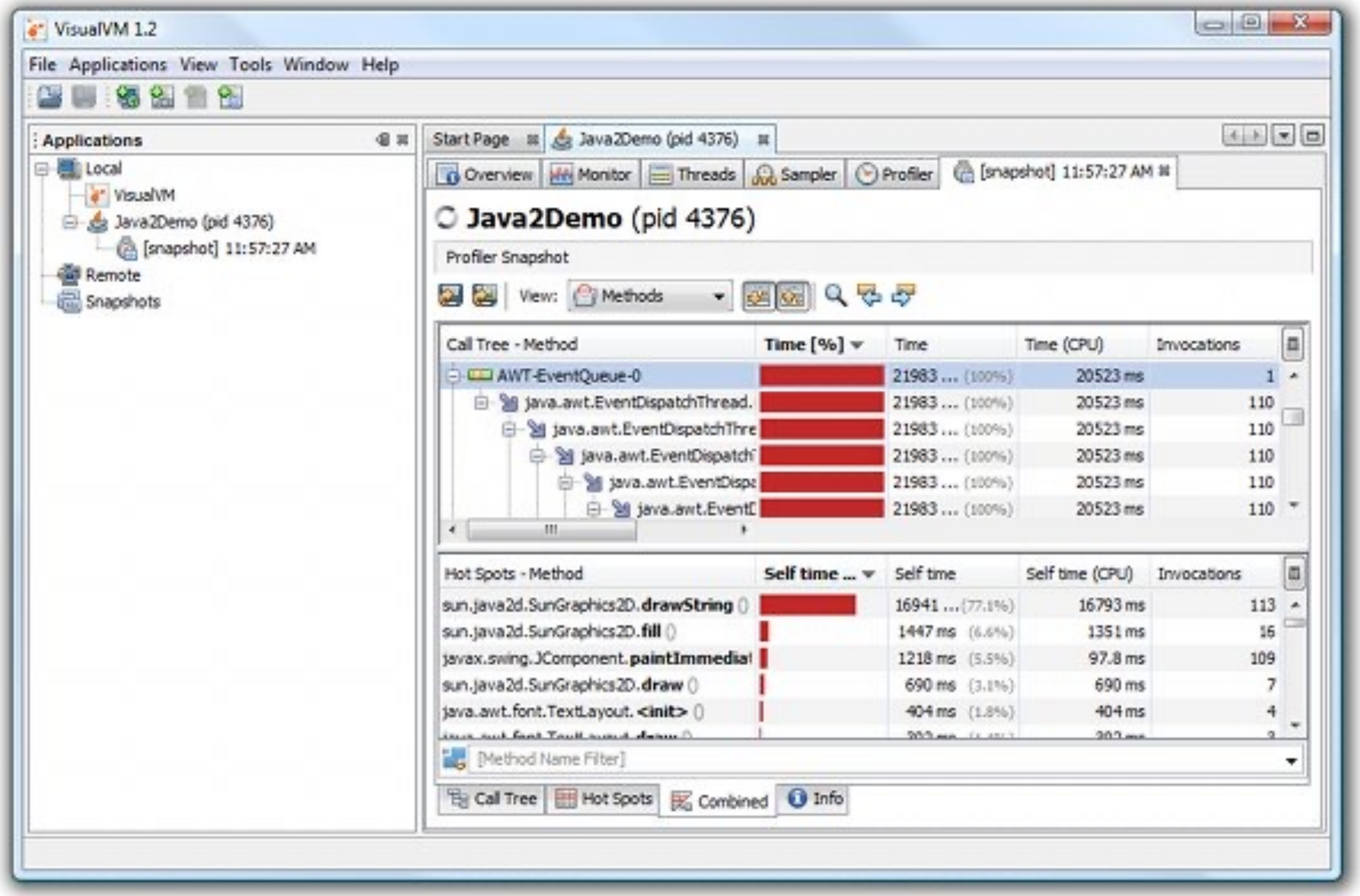


gatling_load_test - Performance Trend

#18	28-Apr-2015 11:33
#17	28-Apr-2015 11:32
#16	28-Apr-2015 11:31
#15	28-Apr-2015 11:30
#14	28-Apr-2015 11:29

Profiling

- Finding bottlenecks i execution time and memory



Java2Demo (pid 4376)

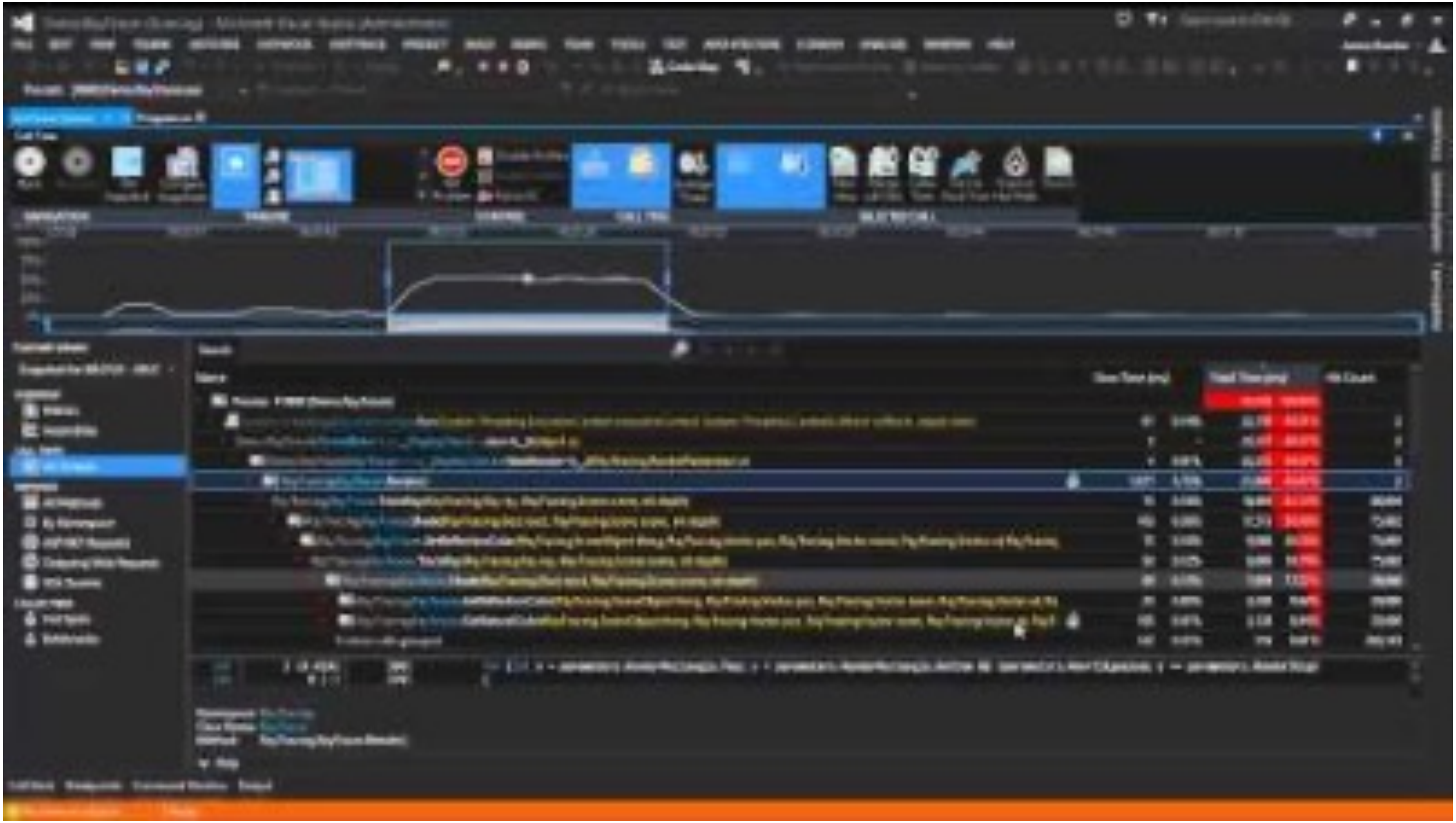
Profiler Snapshot

View: Methods

Call Tree - Method	Time [%]	Time	Time (CPU)	Invocations
AWT-EventQueue-0		21983 ... (100%)	20523 ms	1
java.awt.EventDispatchThread.		21983 ... (100%)	20523 ms	110
java.awt.EventDispatchThre		21983 ... (100%)	20523 ms	110
java.awt.EventDispatchT		21983 ... (100%)	20523 ms	110
java.awt.EventDispa		21983 ... (100%)	20523 ms	110
java.awt.EventD		21983 ... (100%)	20523 ms	110

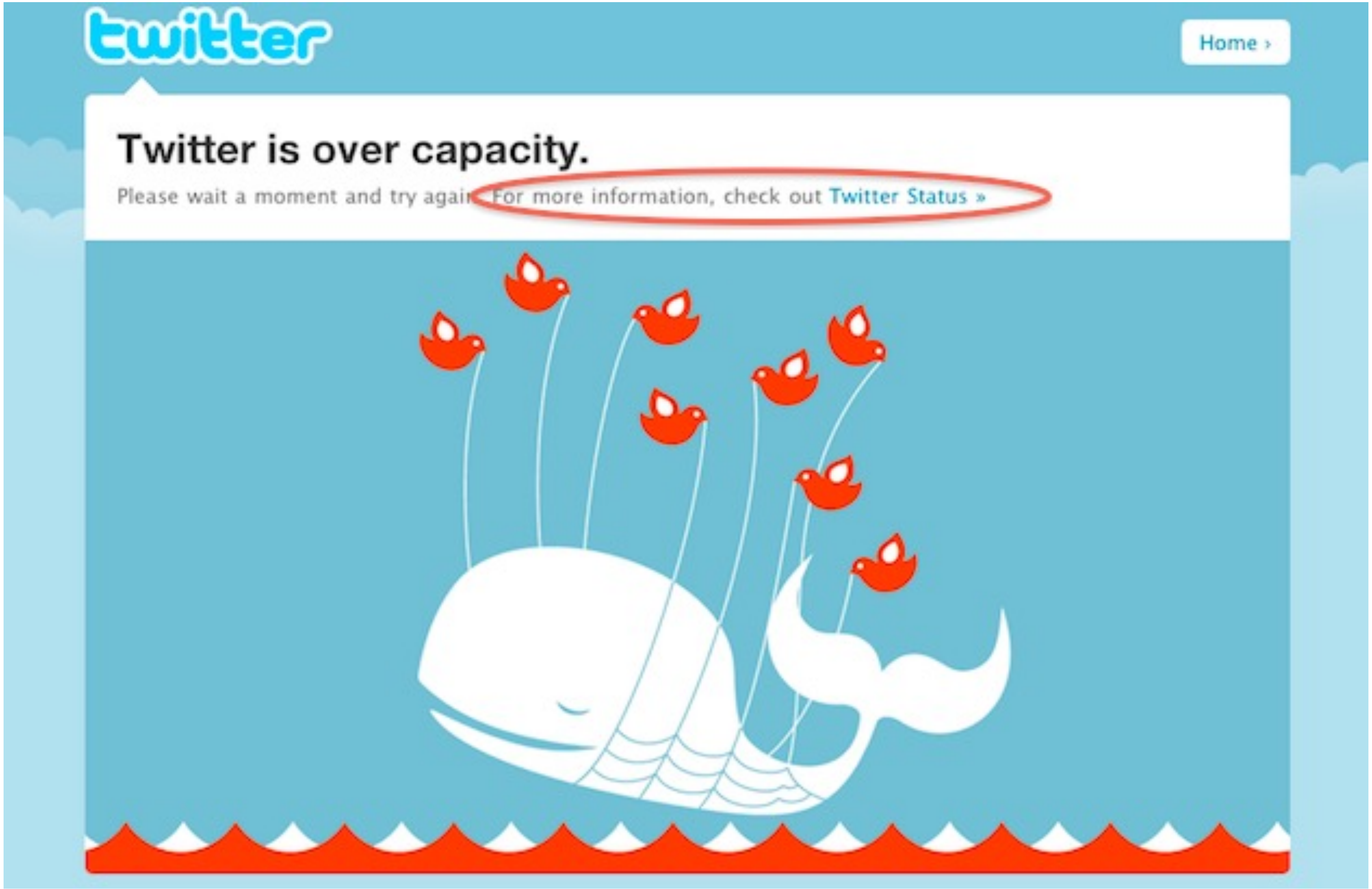
Hot Spots - Method	Self time ...	Self time	Self time (CPU)	Invocations
sun.java2d.SunGraphics2D.**drawString** ()		16941 ... (77.1%)	16793 ms	113
sun.java2d.SunGraphics2D.**fill** ()		1447 ms (6.6%)	1351 ms	16
javax.swing.JComponent.**paintImmediat**		1218 ms (5.5%)	97.8 ms	109
sun.java2d.SunGraphics2D.**draw** ()		690 ms (3.1%)	690 ms	7
java.awt.font.TextLayout.**<init>** ()		404 ms (1.8%)	404 ms	4
java.awt.font.TextLayout.**draw** ()		302 ms (1.4%)	302 ms	3

[Method Name Filter]



https://www.youtube.com/watch?v=0sEgkZ27gtY https://www.telerik.com/

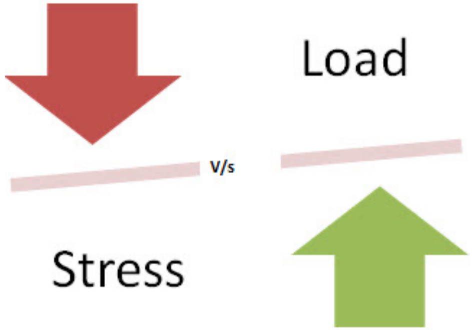
Robustness Testing



Twitter is over capacity.

Please wait a moment and try again. For more information, check out Twitter Status »

Robustness: Stress Testing

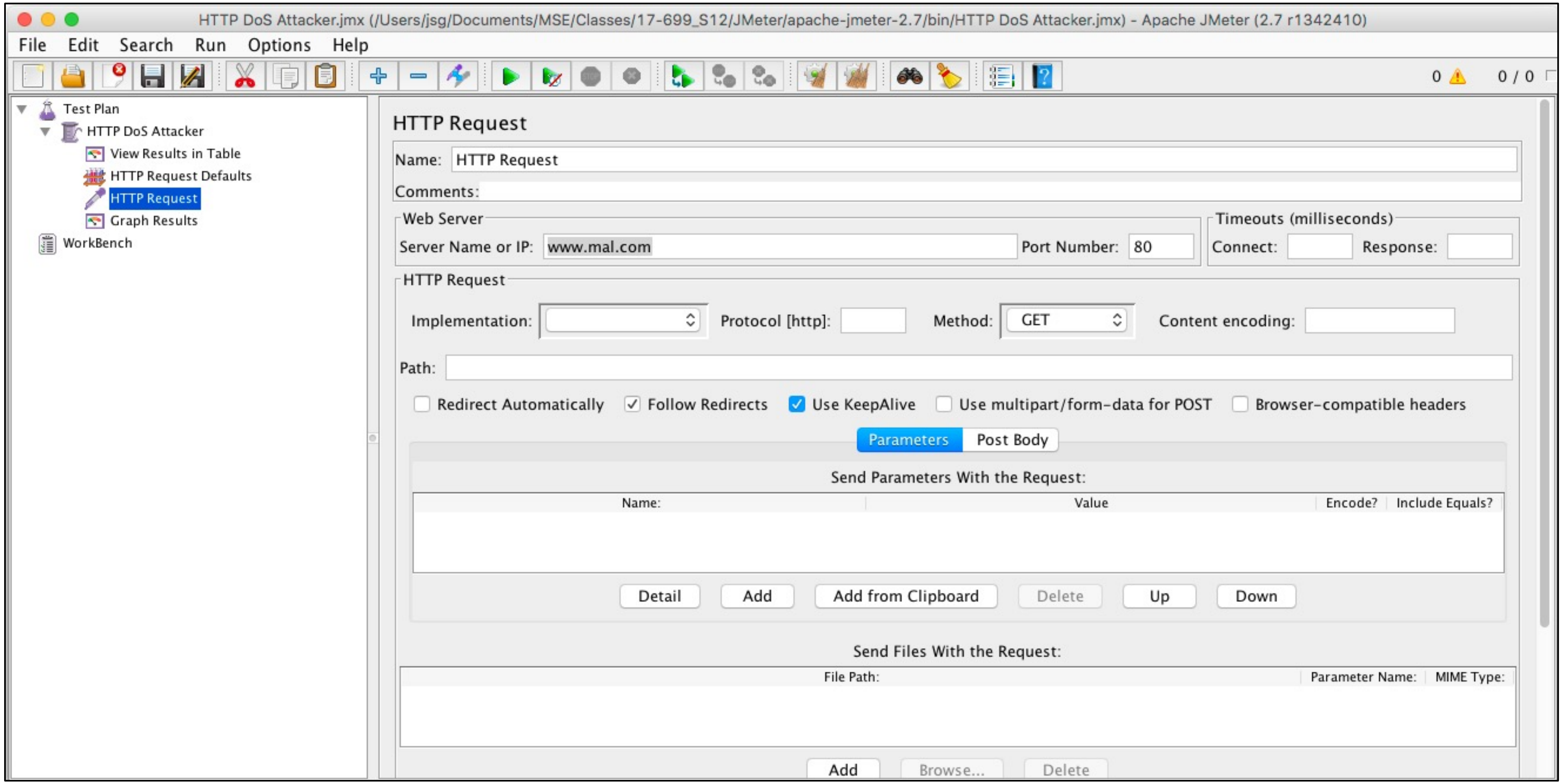


- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal 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
- **Soak testing:** testing a system with a significant load over a significant period of time
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

Performance testing tools: JMeter



HTTP DoS Attacker.jmx (/Users/jsg/Documents/MSE/Classes/17-699_S12/JMeter/apache-jmeter-2.7/bin/HTTP DoS Attacker.jmx) - Apache JMeter (2.7 r1342410)

HTTP Request

Name: HTTP Request

Comments:

Web Server

Server Name or IP: www.mal.com

Port Number: 80

Timeouts (milliseconds)

Connect:

Response:

HTTP Request

Implementation:

Protocol [http]:

Method: GET

Content encoding:

Path:

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters | Post Body

Send Parameters With the Request:

Name:	Value	Encode?	Include Equals?

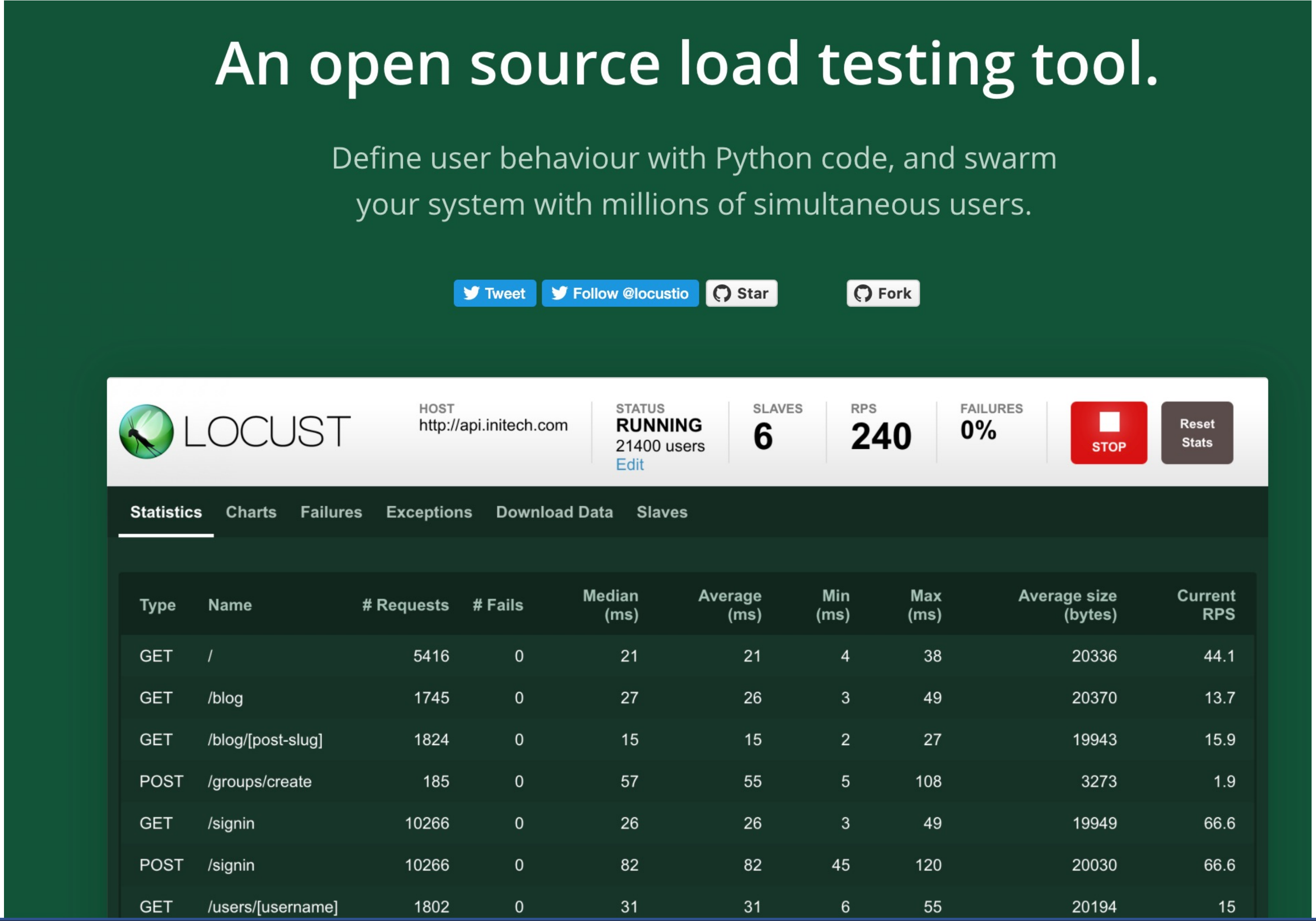
Send Files With the Request:

File Path:	Parameter Name:	MIME Type:

http://jmeter.apache.org

Performance testing tools: Locust

https://github.com/locustio/locust



An open source load testing tool.

Define user behaviour with Python code, and swarm your system with millions of simultaneous users.

HOST http://api.initech.com

STATUS RUNNING 21400 users Edit

SLAVES 6

RPS 240

FAILURES 0%

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/	5416	0	21	21	4	38	20336	44.1
GET	/blog	1745	0	27	26	3	49	20370	13.7
GET	/blog/[post-slug]	1824	0	15	15	2	27	19943	15.9
POST	/groups/create	185	0	57	55	5	108	3273	1.9
GET	/signin	10266	0	26	26	3	49	19949	66.6
POST	/signin	10266	0	82	82	45	120	20030	66.6
GET	/users/[username]	1802	0	31	31	6	55	20194	15

RELIABILITY
LEVEL
20
30
40
50
60
70
80
90
100
per cent
meter
%

Reliability: Fuzz testing

- To send anomalous data to a system in order to crash it, therefore revealing reliability problems.
- Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called **fuzzers**.
- Also known as **fuzzing** or **monkey testing**

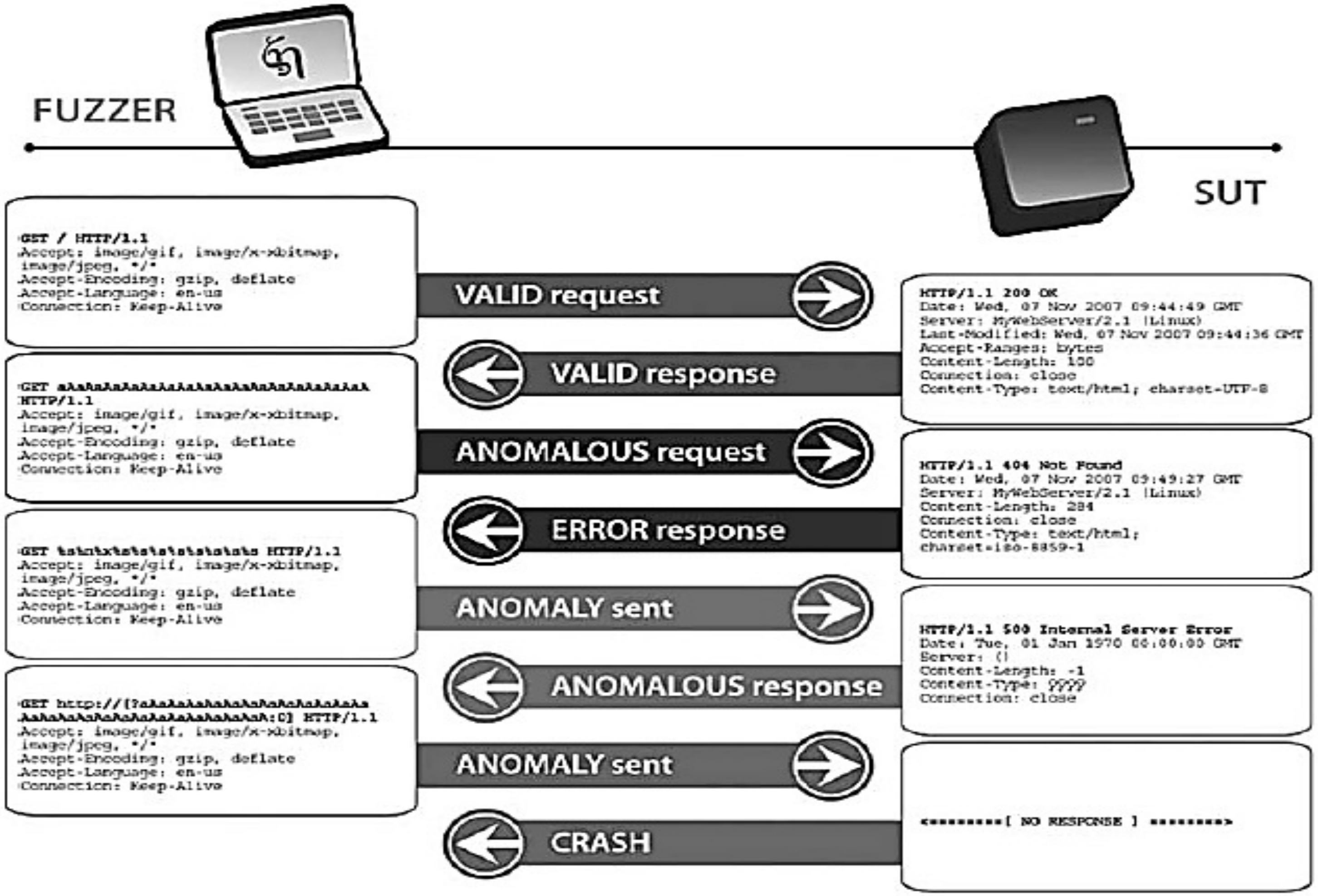
Monkey Testing

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
- black-box testing

https://www.youtube.com/watch?v=17ebHty54T4

Fuzzing Process



```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
HTTP/1.1 200 OK
Date: Wed, 07 Nov 2007 09:44:49 GMT
Server: MyWebServer/2.1 (Linux)
Last-Modified: Wed, 07 Nov 2007 09:44:36 GMT
Accept-Ranges: bytes
Content-Length: 100
Connection: close
Content-Type: text/html; charset=UTF-8
```

```
GET aAaAaAaAaAaAaAaAaAaAaAaAaAaAaAaAaAaA
HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
HTTP/1.1 404 Not Found
Date: Wed, 07 Nov 2007 09:49:27 GMT
Server: MyWebServer/2.1 (Linux)
Content-Length: 284
Connection: close
Content-Type: text/html;
charset=iso-8859-1
```

```
GET %s%n%x%s%s%s%s%s%s%s%s HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

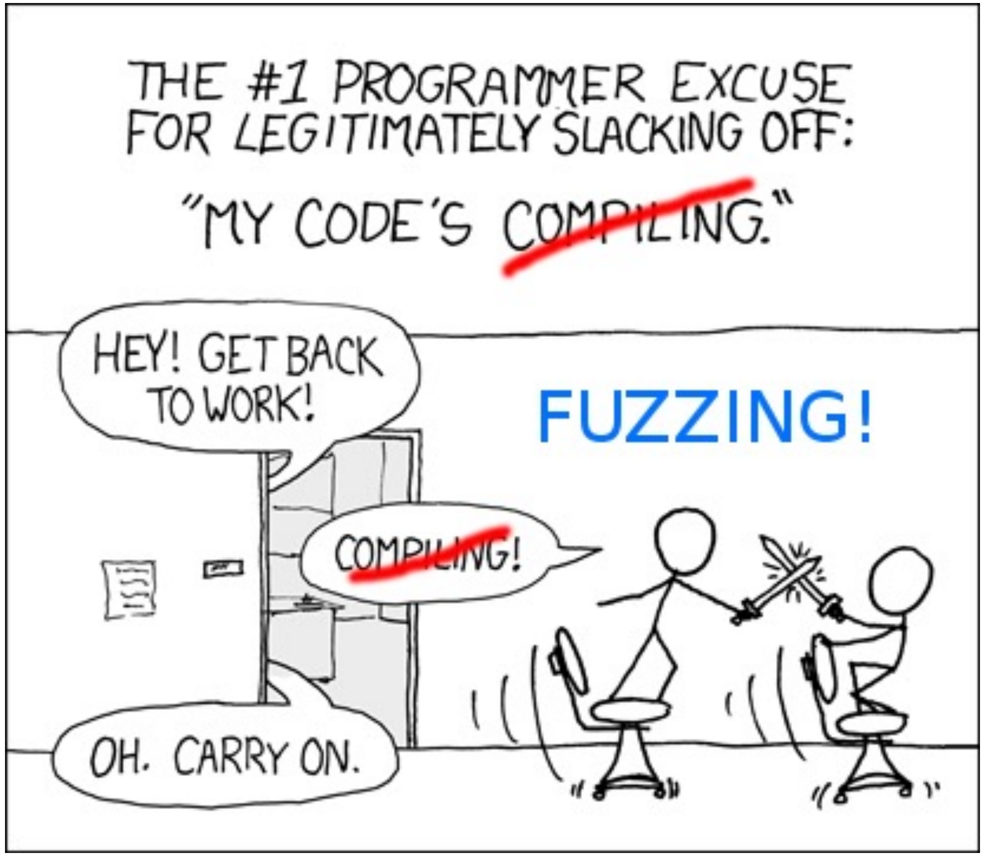
```
HTTP/1.1 500 Internal Server Error
Date: Tue, 01 Jan 1970 00:00:00 GMT
Server: ()
Content-Length: -1
Content-Type: ????
Connection: close
```

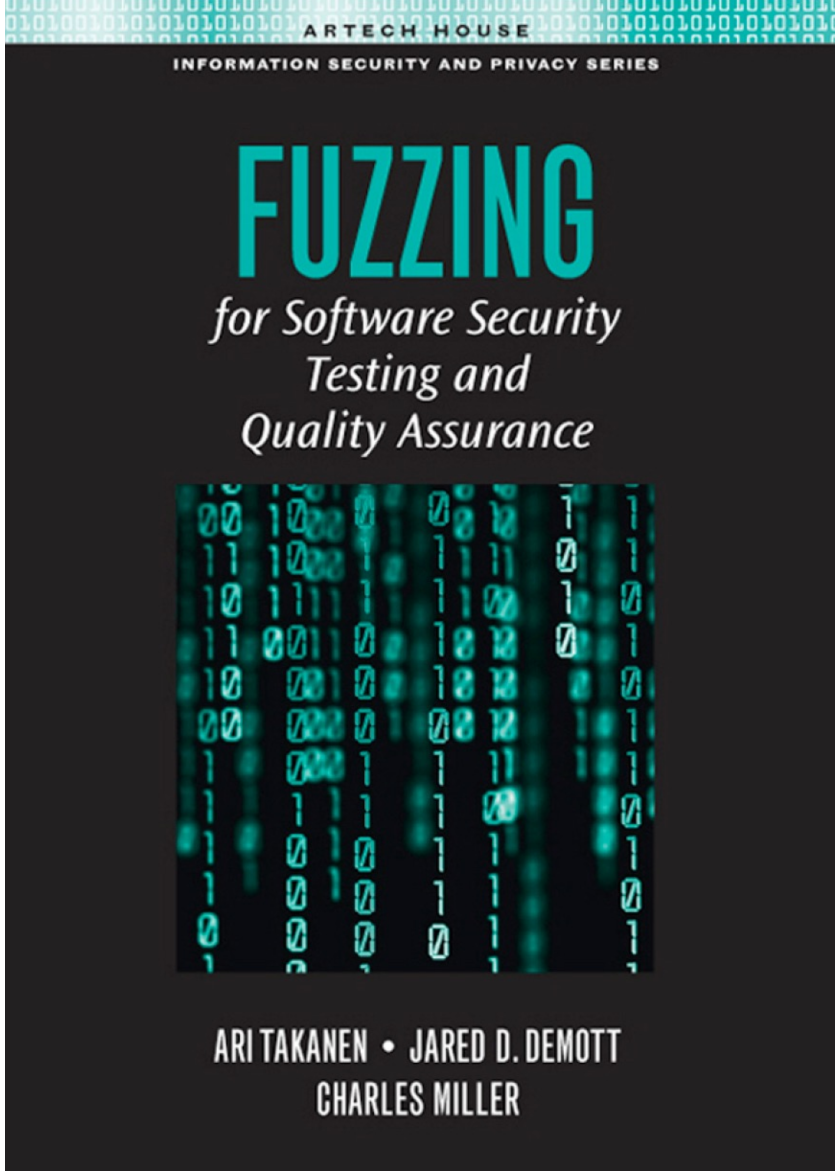
```
GET http://[?aAaAaAaAaAaAaAaAaAaAaAaAaAa
AaAaAaAaAaAaAaAaAaAaAaAaAaAaA:0] HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
<========[ NO RESPONSE ] ========>
```

System under test (SUT)

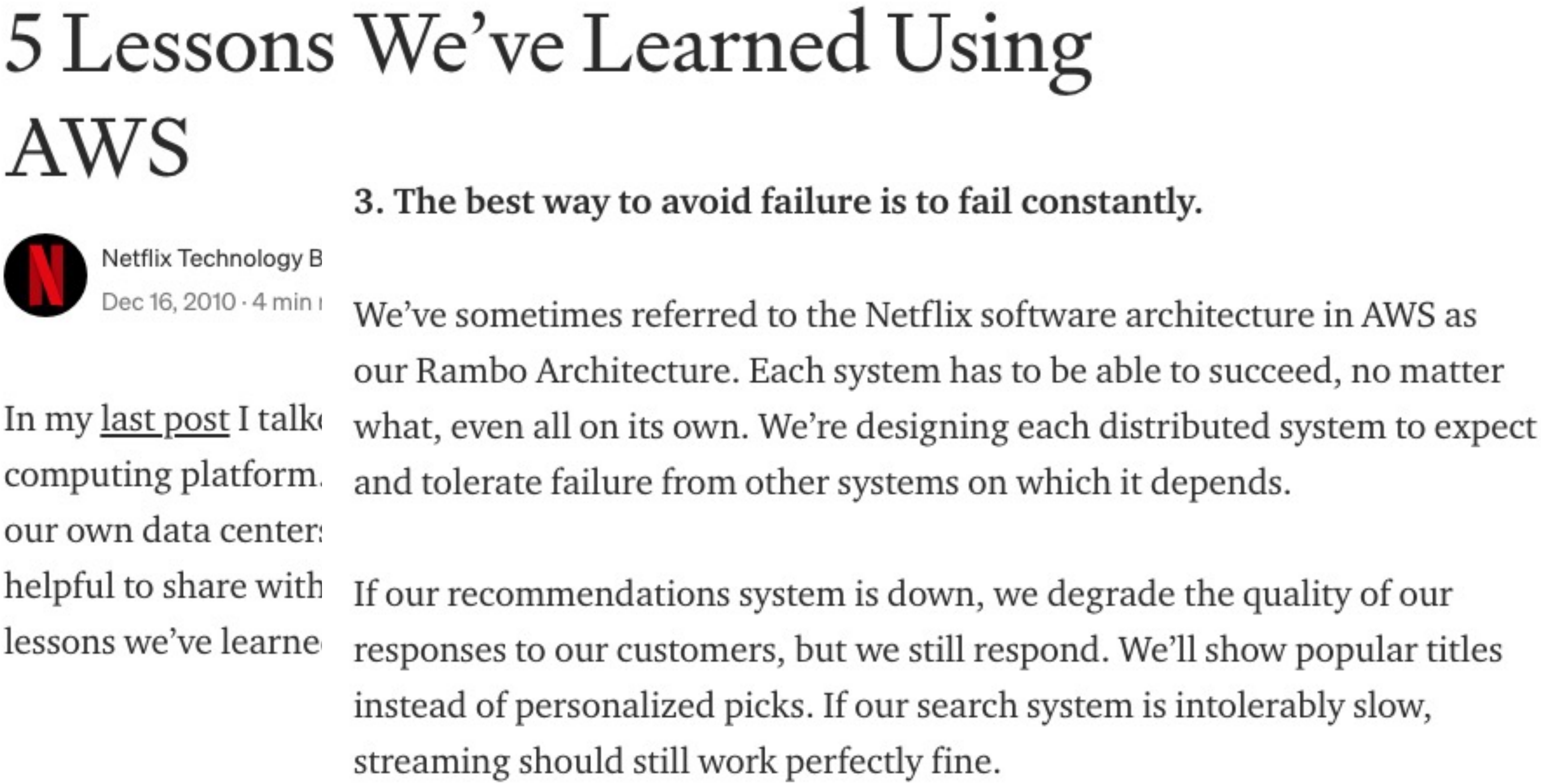
Reliability: Fuzz testing


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S ~~COMPILING~~."
HEY! GET BACK TO WORK!
FUZZING!
~~COMPILING~~!
OH. CARRY ON.


ARTECH HOUSE
INFORMATION SECURITY AND PRIVACY SERIES
FUZZING
for Software Security Testing and Quality Assurance
ARI TAKANEN • JARED D. DEMOTT
CHARLES MILLER

(A. Takanen et al, Fuzzing for Software Security Testing and Quality Assurance, 2008)

Chaos Engineering



5 Lessons We’ve Learned Using AWS

Netflix Technology B

Dec 16, 2010 · 4 min r

In my last post I talk
computing platform.
our own data center
helpful to share with
lessons we’ve learne

3. The best way to avoid failure is to fail constantly.

We’ve sometimes referred to the Netflix software architecture in AWS as our Rambo Architecture. Each system has to be able to succeed, no matter what, even all on its own. We’re designing each distributed system to expect and tolerate failure from other systems on which it depends.

If our recommendations system is down, we degrade the quality of our responses to our customers, but we still respond. We’ll show popular titles instead of personalized picks. If our search system is intolerably slow, streaming should still work perfectly fine.

NETFLIX

Principle of Chaos Engineering

Proactively inject failures in order to be prepared when disaster strikes.

“Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.”

Goal: To intentionally break things, compare measured with expected impact, and correct any problems uncovered this way.

NETFLIX

Principle of Chaos Engineering

4 steps:

- Define the system’s normal behavior
- Hypothesize about the steady state behavior of an experimental group, as compared to a stable control group.
- Expose the experimental group to simulated real-world events such as server crashes, malformed responses, or traffic spikes.
- Test the hypothesis by comparing the steady state of the control group and the experimental group.

The smaller the differences, the more confidence we have that the system is resilient.

https://www.youtube.com/watch?v=3WRVgC8SiGc

amazon
web services
CHAOS
MONKEY

SIMIAN
ARMY

NETFLIX

Chaos monkey/Simian army

- “Malicious” programs randomly trample on components, network, datacenters, AWS instances…
  - 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
  - open-sourced

Awesome Chaos Engineering

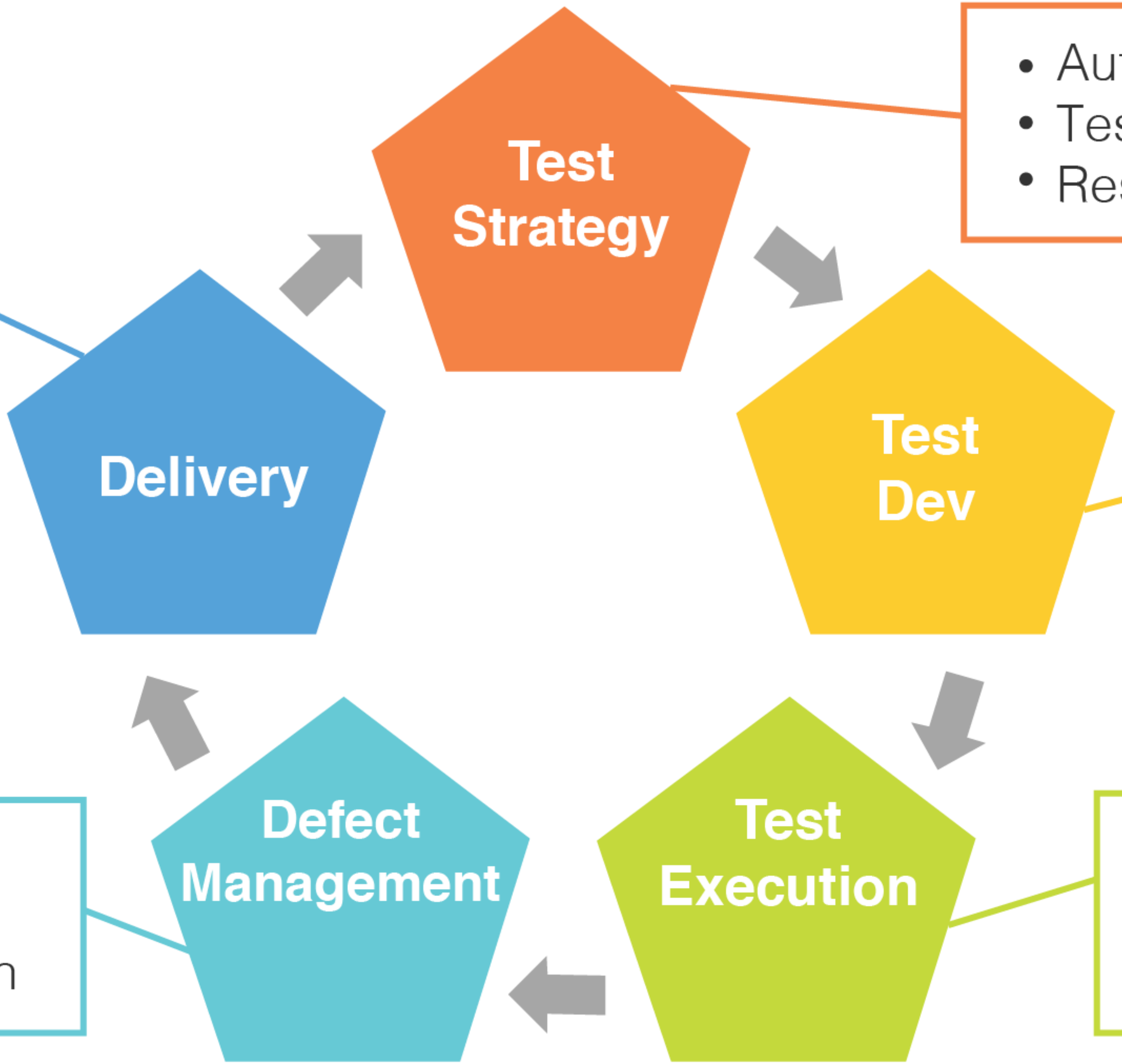
A curated list of awesome Chaos Engineering resources.

https://github.com/dastergon/awesome-chaos-engineering

Use Gremlin to validate your monitoring.

https://www.youtube.com/watch?v=VUwi5Jtw3ow&feature=youtu.be

GREMLIN



- Automation Strategy
- Test Schedule
- Resource Planning

- Test Plans
- Test Matrix
- Test Scripts
- Test Data

- Defects
- Test Reports
- Test Metrics

- Bug Tracking
- Bug Fixing
- Bug Verification

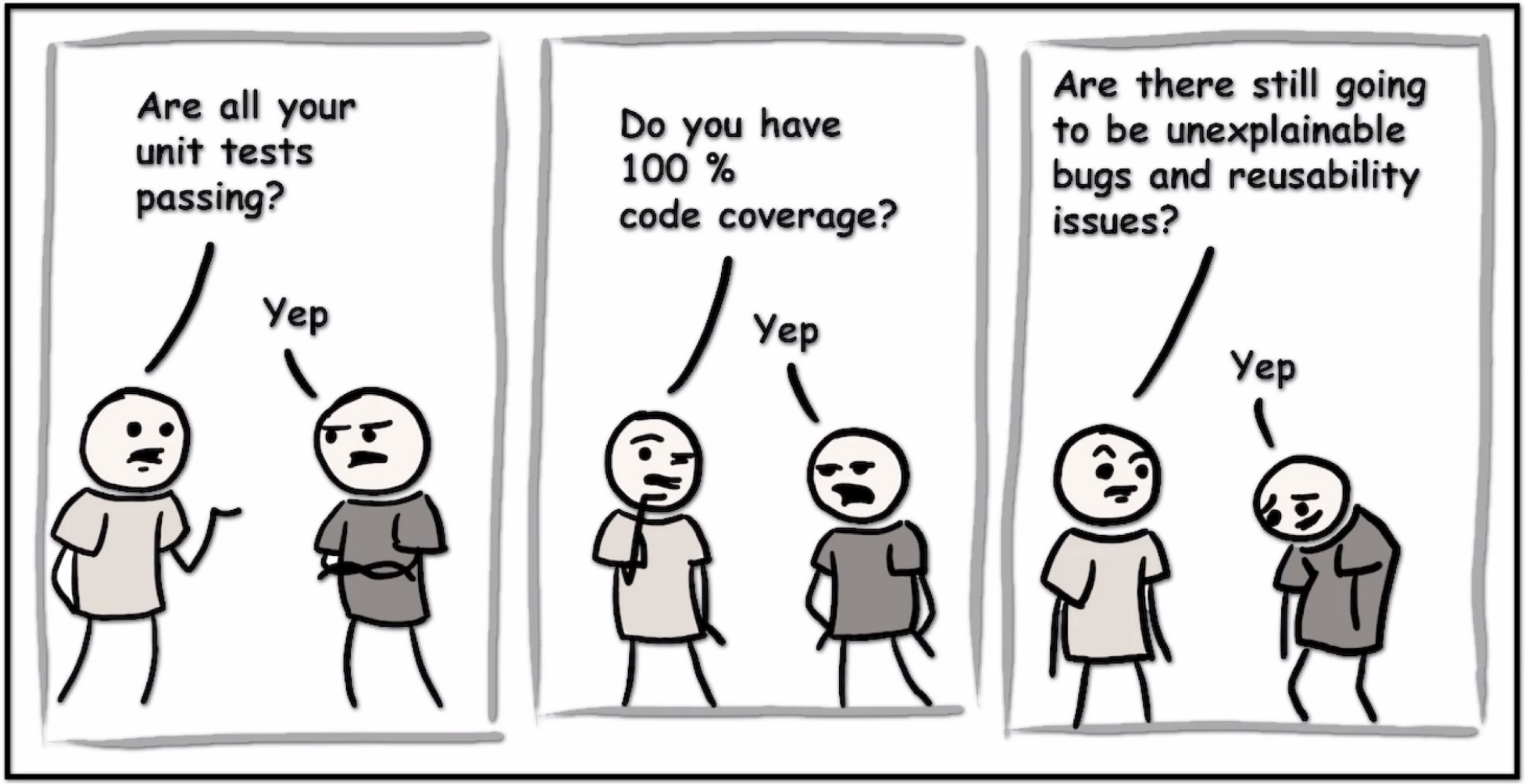
- User Acceptance Testing
- Operational Readiness Test
- Requirements Verification

Limits of Testing

- Cannot find bugs in code not executed, cannot assure absence of bugs
- Oracle problem
- Nondeterminism, flaky tests
  - Certain kinds of bugs occur only under very unlikely conditions
- Hard to observe/assert specifications
  - Memory leaks, information flow, …
- Potentially expensive, long run times
- Potentially high manual effort
- Verification, not validation
- …

Recording

But coverage has limitations.


Are all your unit tests passing?
Yep
Do you have 100 % code coverage?
Yep
Are there still going to be unexplainable bugs and reusability issues?
Yep

Low coverage means insufficient testing.

Summary

- Quality assurance is important, often underestimated
- Many forms of QA, testing popular
- Testing beyond functional correctness

Program Analysis

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

Definition: software analysis

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

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.

Principle techniques

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

What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.
  - Does not execute code! (like code review)
- **Abstraction:** produce a representation of a program that is simpler to analyze.
  - Results in fewer states to explore; makes difficult problems tractable.

Syntactic Analysis

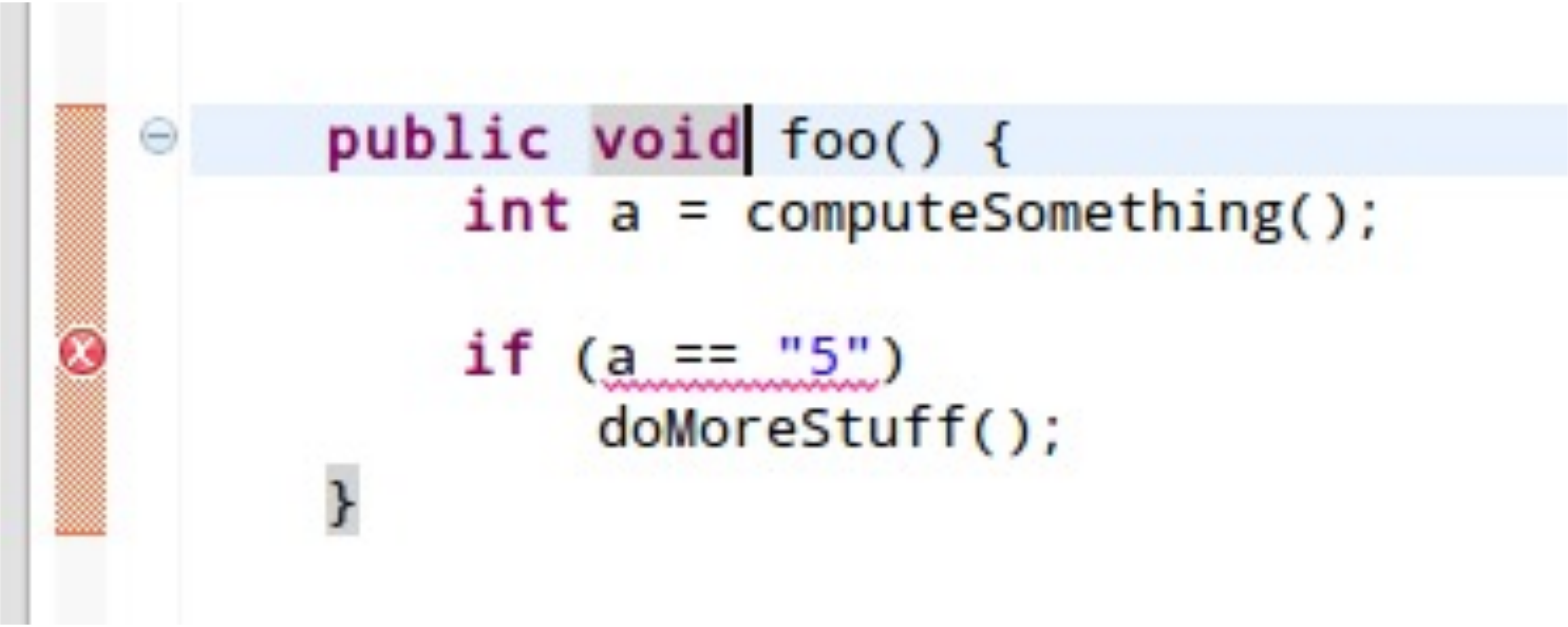
Find every occurrence of this pattern:

```
public foo() {
  …
   logger.debug(“We have ” + conn + “connections.”);
}
```

```
public foo() {
  …
  if (logger.inDebug()) {
    logger.debug(“We have ” + conn + “connections.”);
  }
}
```

grep "if \(logger\.inDebug" . -r

Type Analysis



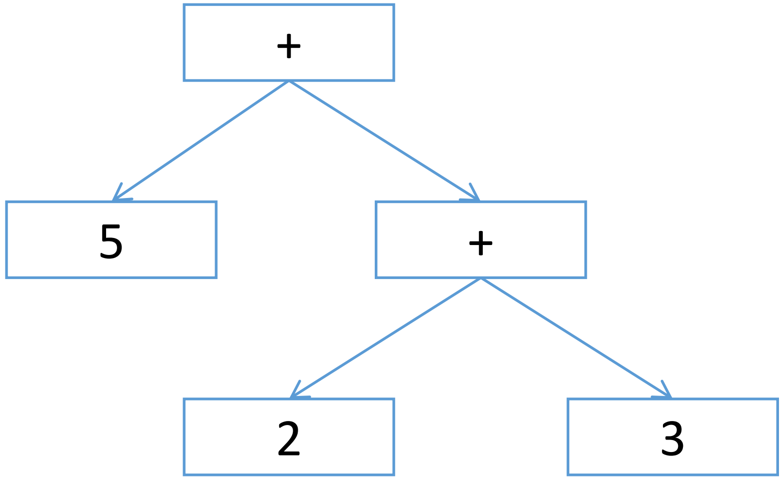
```
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

Abstraction: abstract syntax tree

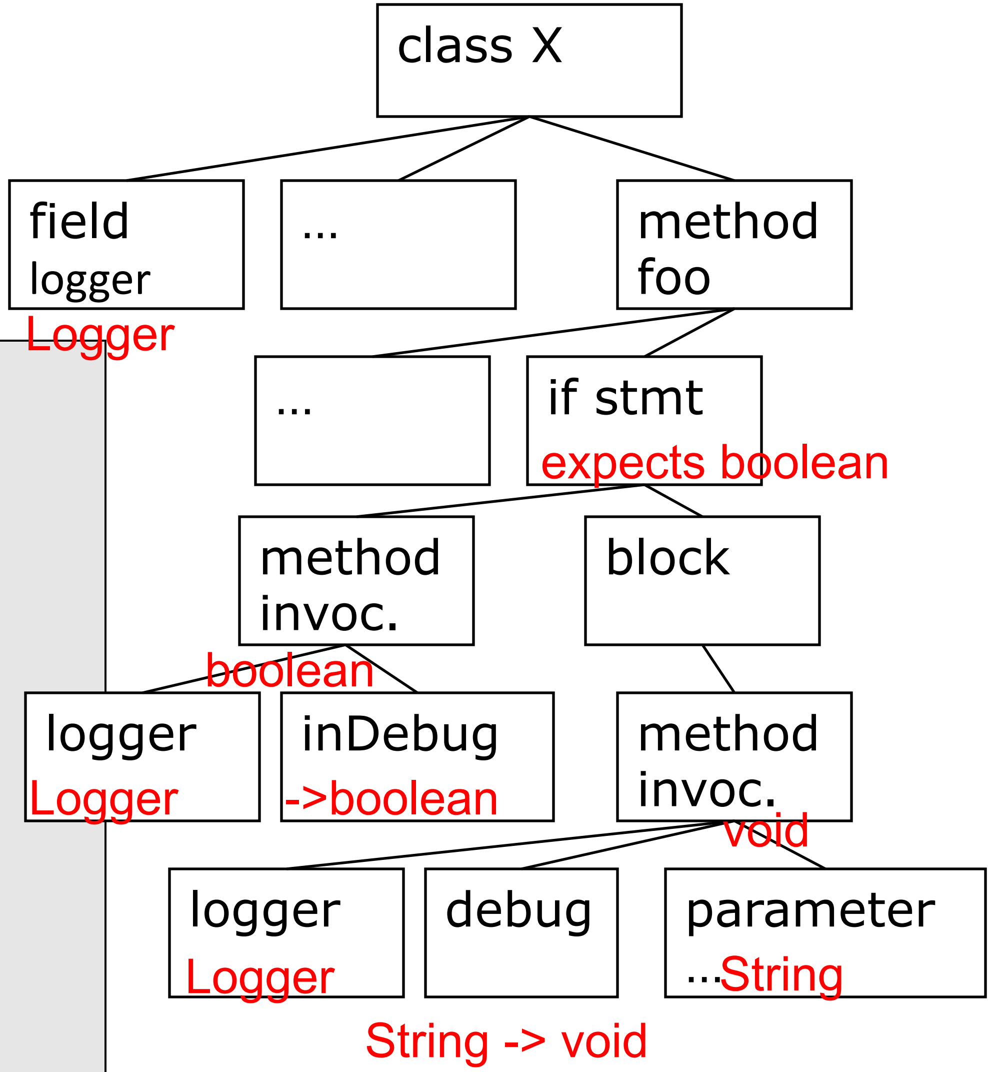
- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn't represent every detail (like parentheses); these can be inferred from the structure.
- (How to build one? Take compilers!)

- Example: 5 + (2 + 3)



Type checking

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug(“We have ” +
conn + “connections.”);
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```

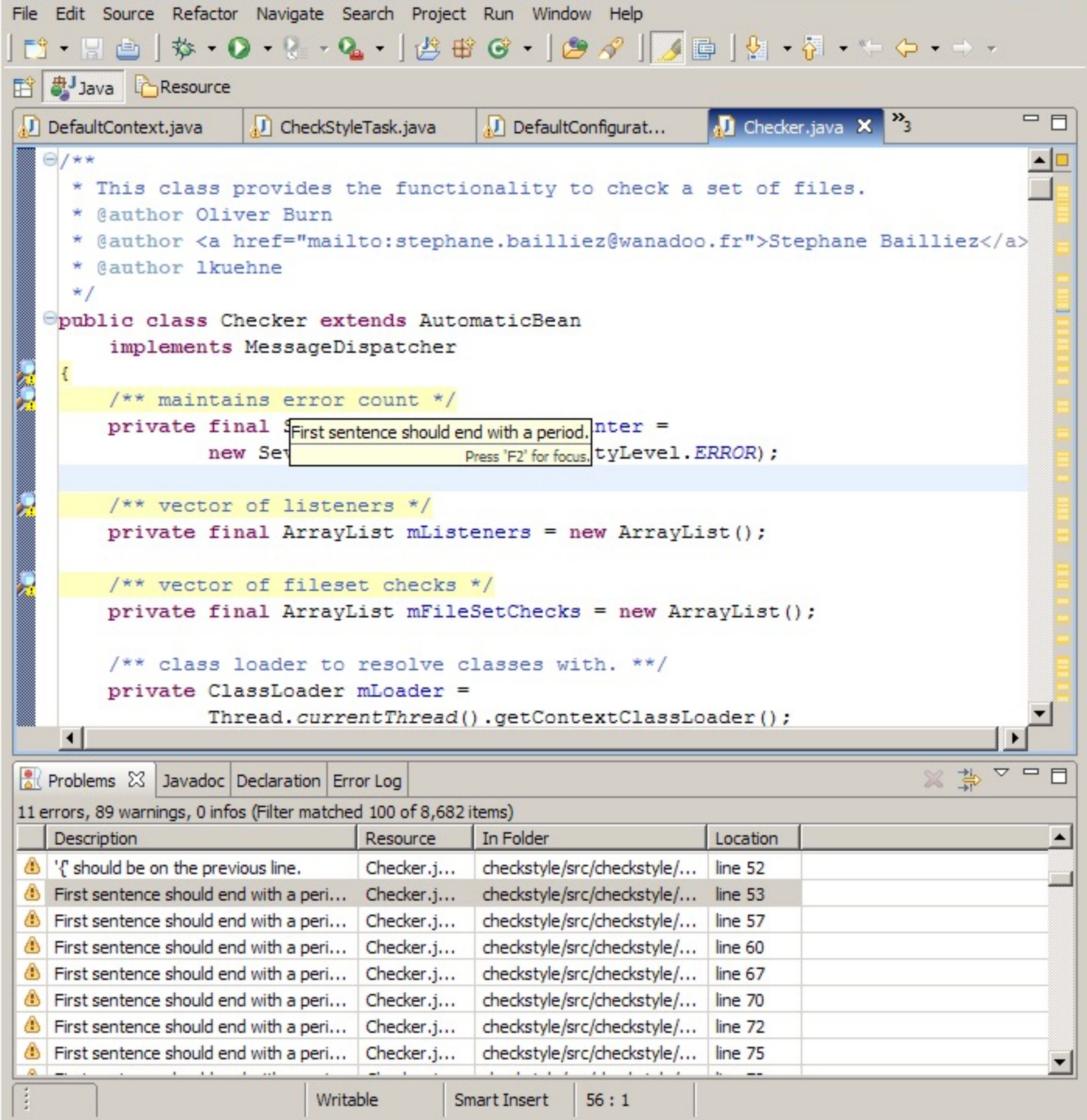


Summary:
Syntactic/Structural Analyses

- Analyzing token streams or code structures (ASTs)
- Useful to find patterns
- Local/structural properties, independent of execution paths

Tools

- Checkstyle
- Many linters (C, JS, Python, …)
- Findbugs (some analyses)



```
/**
 * This class provides the functionality to check a set of files.
 * @author Oliver Burn
 * @author <a href="mailto:stephane.bailliez@wanadoo.fr">Stephane Bailliez</a>
 * @author lkuehne
 */
public class Checker extends AutomaticBean
    implements MessageDispatcher
{
    /** maintains error count */
    private final S                      nter =
            new Se                       tyLevel.ERROR);

    /** vector of listeners */
    private final ArrayList mListeners = new ArrayList();

    /** vector of fileset checks */
    private final ArrayList mFileSetChecks = new ArrayList();

    /** class loader to resolve classes with. **/
    private ClassLoader mLoader =
            Thread.currentThread().getContextClassLoader();
```

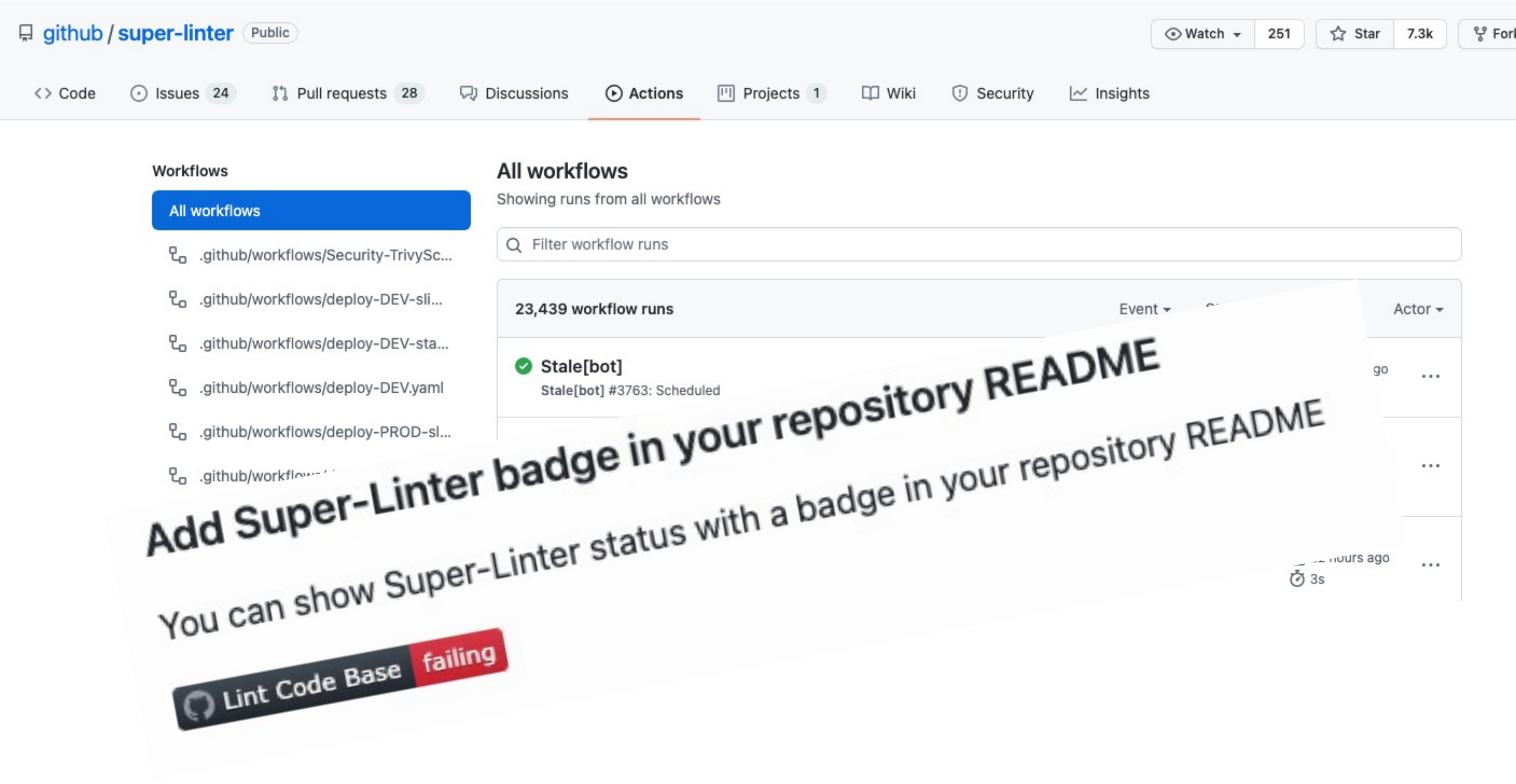
First sentence should end with a period.
Press 'F2' for focus.

11 errors, 89 warnings, 0 infos (Filter matched 100 of 8,682 items)

Description	Resource	In Folder	Location
'{' should be on the previous line.	Checker.j...	checkstyle/src/checkstyle/...	line 52
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 53
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 57
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 60
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 67
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 70
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 72
First sentence should end with a peri...	Checker.j...	checkstyle/src/checkstyle/...	line 75

Tool -- Linter

- **Lint**, or a **linter**, is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs. – [wikipedia]



Workflows

- All workflows
- .github/workflows/Security-TrivySc...
- .github/workflows/deploy-DEV-sli...
- .github/workflows/deploy-DEV-sta...
- .github/workflows/deploy-DEV.yaml
- .github/workflows/deploy-PROD-sl...
- .github/workflow...

All workflows

Showing runs from all workflows

23,439 workflow runs

Stale[bot]

Stale[bot] #3763: Scheduled

Add Super-Linter badge in your repository README

You can show Super-Linter status with a badge in your repository README

Lint Code Base failing

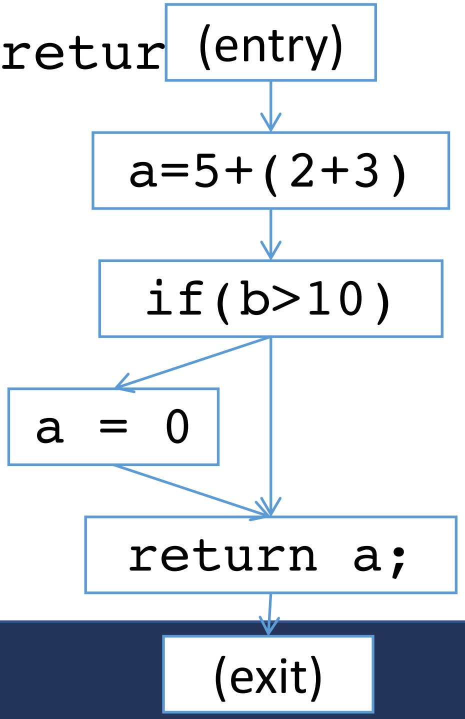
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
  - Including exception handling, function calls, etc
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Control flow graphs

- A tree/graph-based representation of the flow of control through the program.
    - Captures all possible execution paths.
- Each node is a basic block: no jumps in or out.
- Edges represent control flow options between nodes.
- Intra-procedural: within one function.
    - cf. inter-procedural

```
a = 5 + (2 + 3)
if (b > 10) {
  a = 0;
}
retur
```



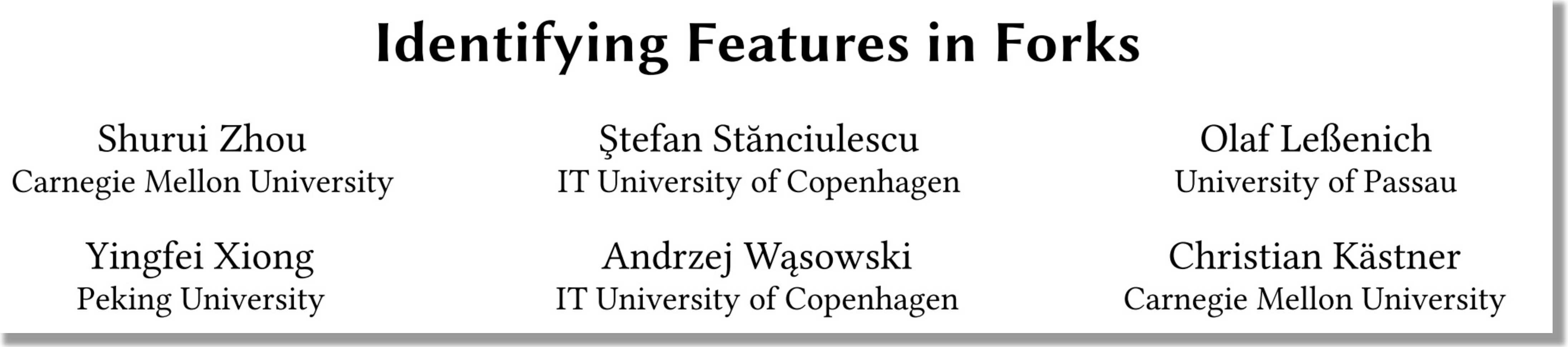
Data- vs. control-flow

- Dataflow: tracks abstract values for each of (some subset of) the **variables** in a program.
- Control flow: tracks state **global** to the function in question.

Tools

- Dead-code detection in many compilers (e.g. Java)
- Instrumentation for dynamic analysis before and after decision points; loop detection

Other application scenarios



Identifying Features in Forks

Shurui Zhou
Carnegie Mellon University

Ştefan Stănciulescu
IT University of Copenhagen

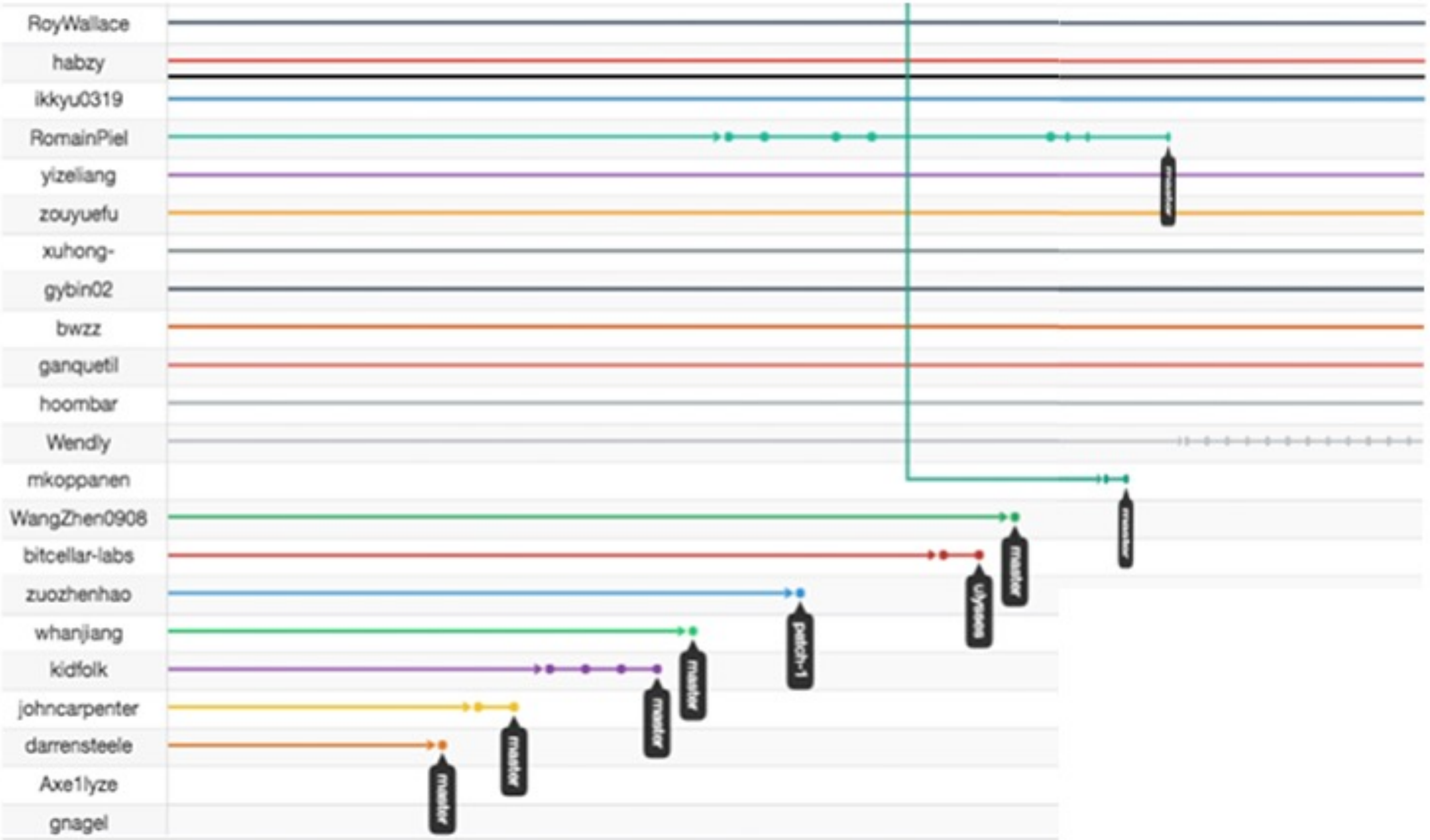
Olaf Leßenich
University of Passau

Yingfei Xiong
Peking University

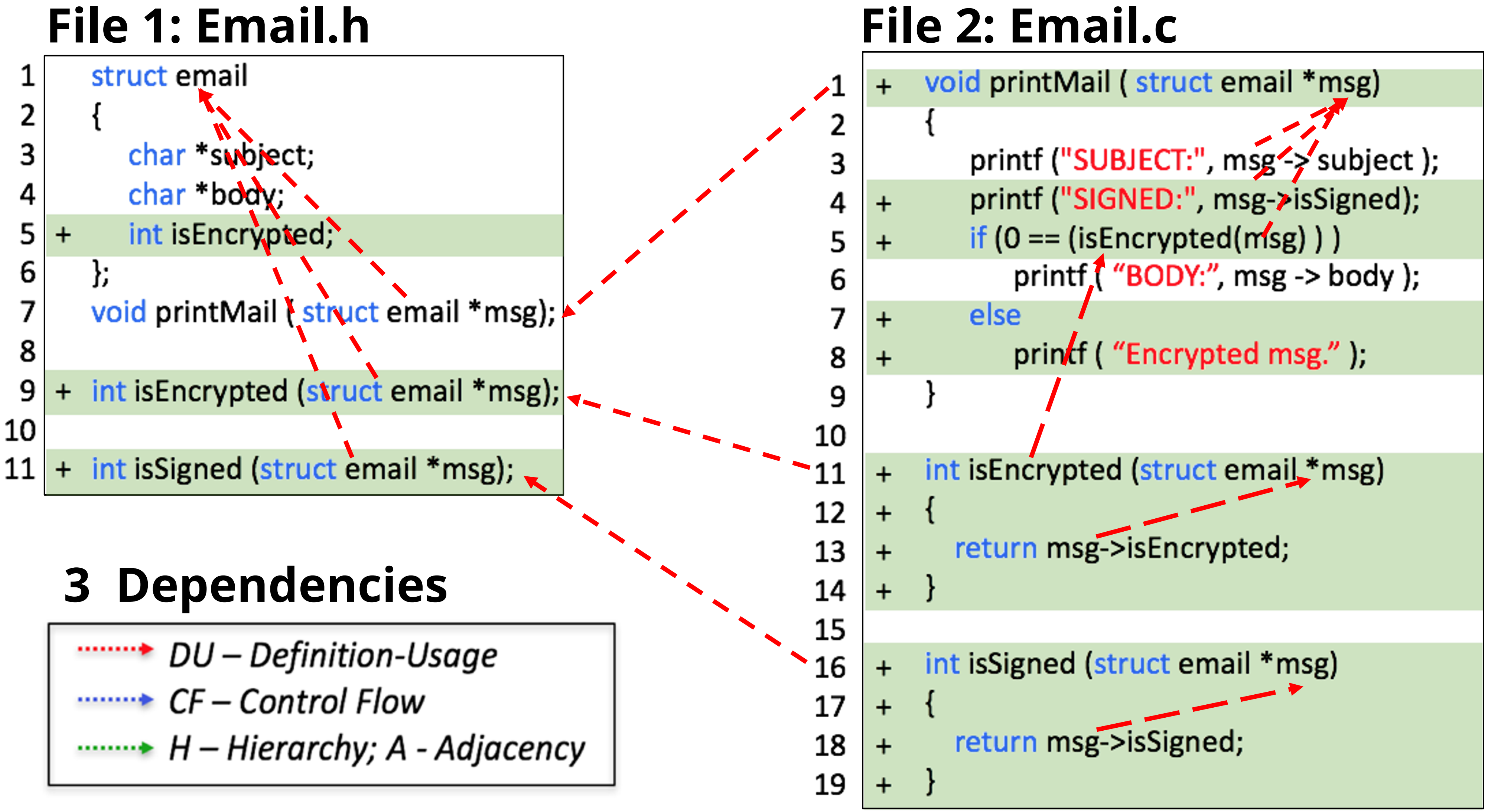
Andrzej Wąsowski
IT University of Copenhagen

Christian Kästner
Carnegie Mellon University

On GitHub, it is hard to figure out who has implemented what feature …



Dependency Graph



File 1: Email.h

```
   struct email
   {
       char *subject;
       char *body;
+      int isEncrypted;
   };
   void printMail ( struct email *msg);

+  int isEncrypted (struct email *msg);

+  int isSigned (struct email *msg);
```

File 2: Email.c

```
+  void printMail ( struct email *msg)
   {
       printf ("SUBJECT:", msg -> subject );
+      printf ("SIGNED:", msg->isSigned);
+      if (0 == (isEncrypted(msg) ) )
          printf ( “BODY:”, msg -> body );
+      else
          printf ( “Encrypted msg.” );
   }

+  int isEncrypted (struct email *msg)
+  {
+     return msg->isEncrypted;
+  }

+  int isSigned (struct email *msg)
+  {
+     return msg->isSigned;
+  }
```

3 Dependencies

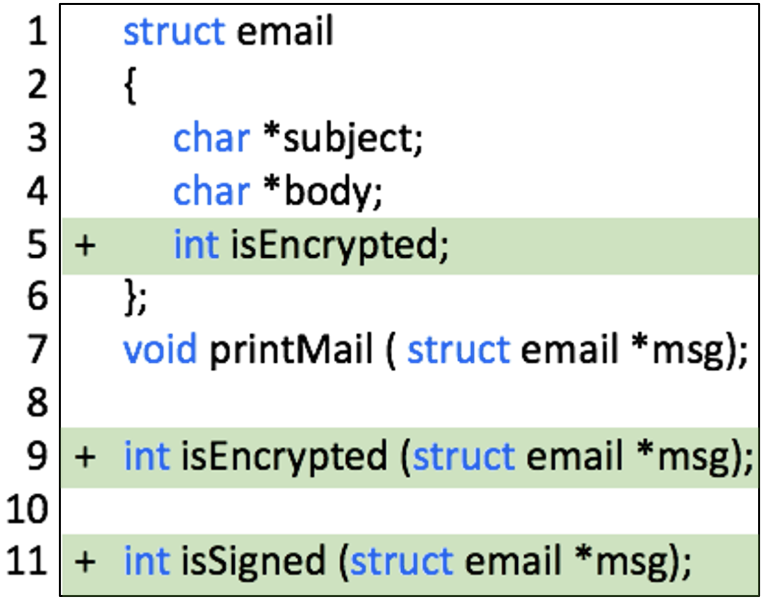
DU – Definition-Usage

CF – Control Flow

H – Hierarchy; A - Adjacency

Dependency Graph

File 1: Email.h

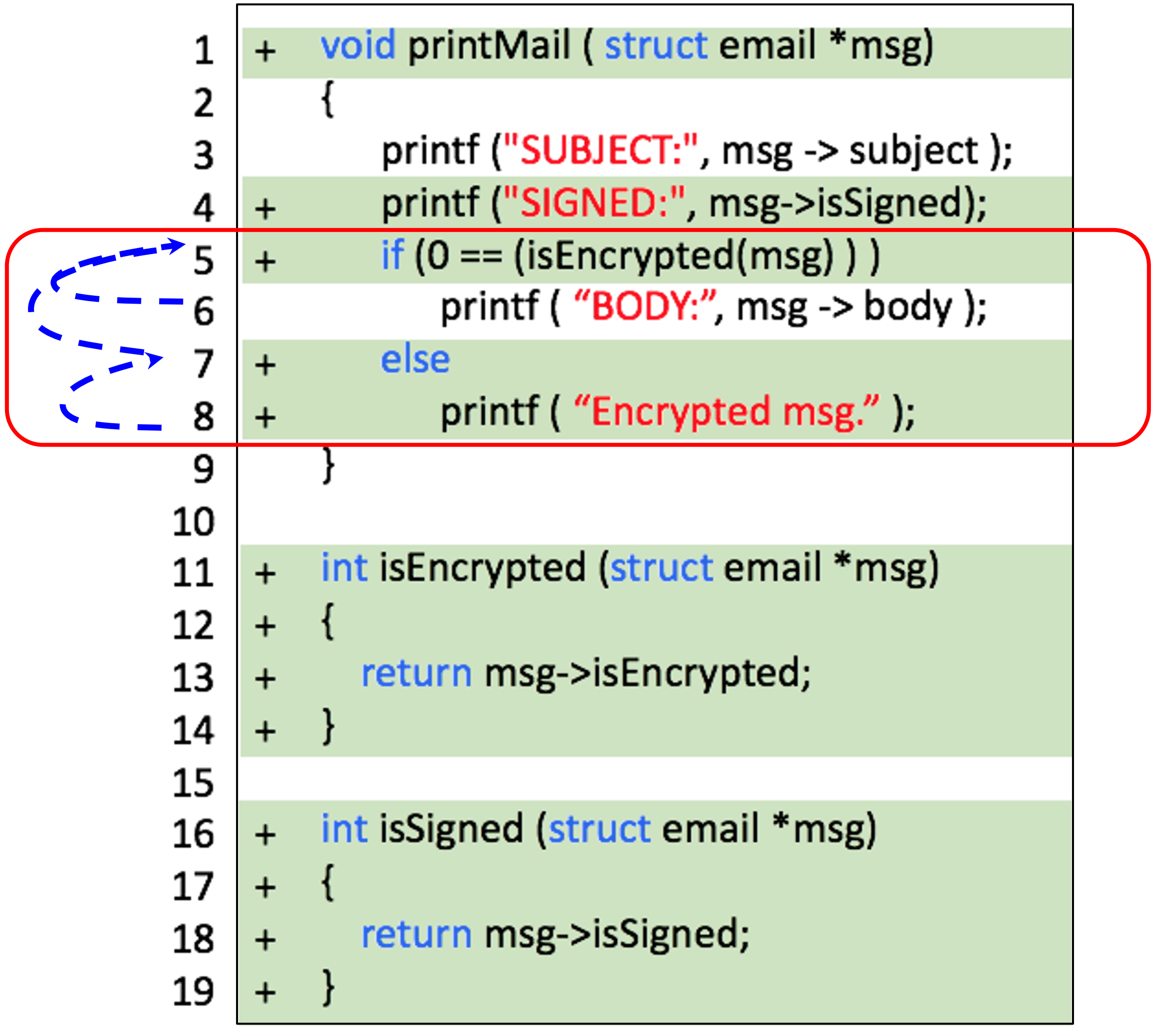


```
  struct email
  {
      char *subject;
      char *body;
+     int isEncrypted;
  };
  void printMail ( struct email *msg);

+ int isEncrypted (struct email *msg);

+ int isSigned (struct email *msg);
```

File 2: Email.c



```
+ void printMail ( struct email *msg)
  {
      printf ("SUBJECT:", msg -> subject );
+     printf ("SIGNED:", msg->isSigned);
+     if (0 == (isEncrypted(msg) ) )
          printf ( “BODY:”, msg -> body );
+     else
+         printf ( “Encrypted msg.” );
  }

+ int isEncrypted (struct email *msg)
+ {
+     return msg->isEncrypted;
+ }

+ int isSigned (struct email *msg)
+ {
+     return msg->isSigned;
+ }
```

3 Dependencies

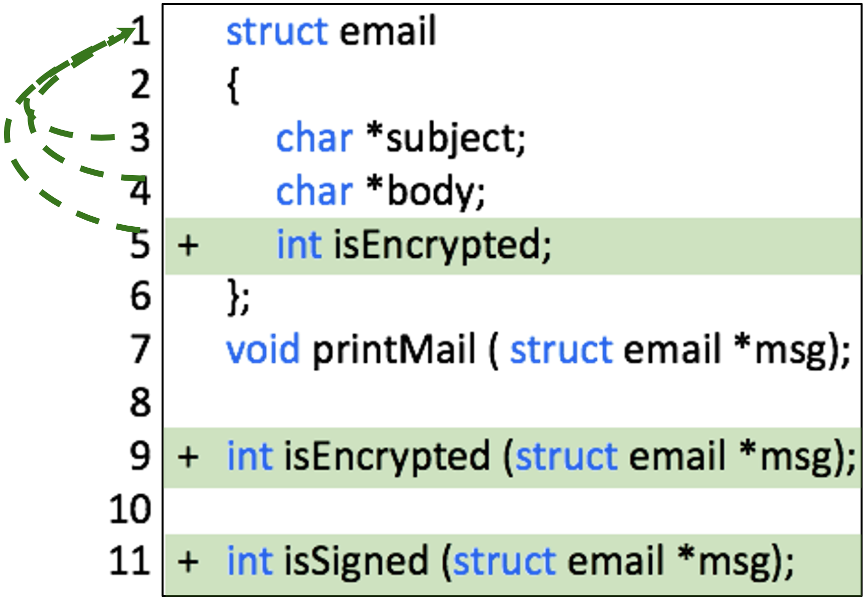
→ *DU – Definition-Usage*

→ *CF – Control Flow*

→ *H – Hierarchy; A - Adjacency*

Dependency Graph

File 1: Email.h

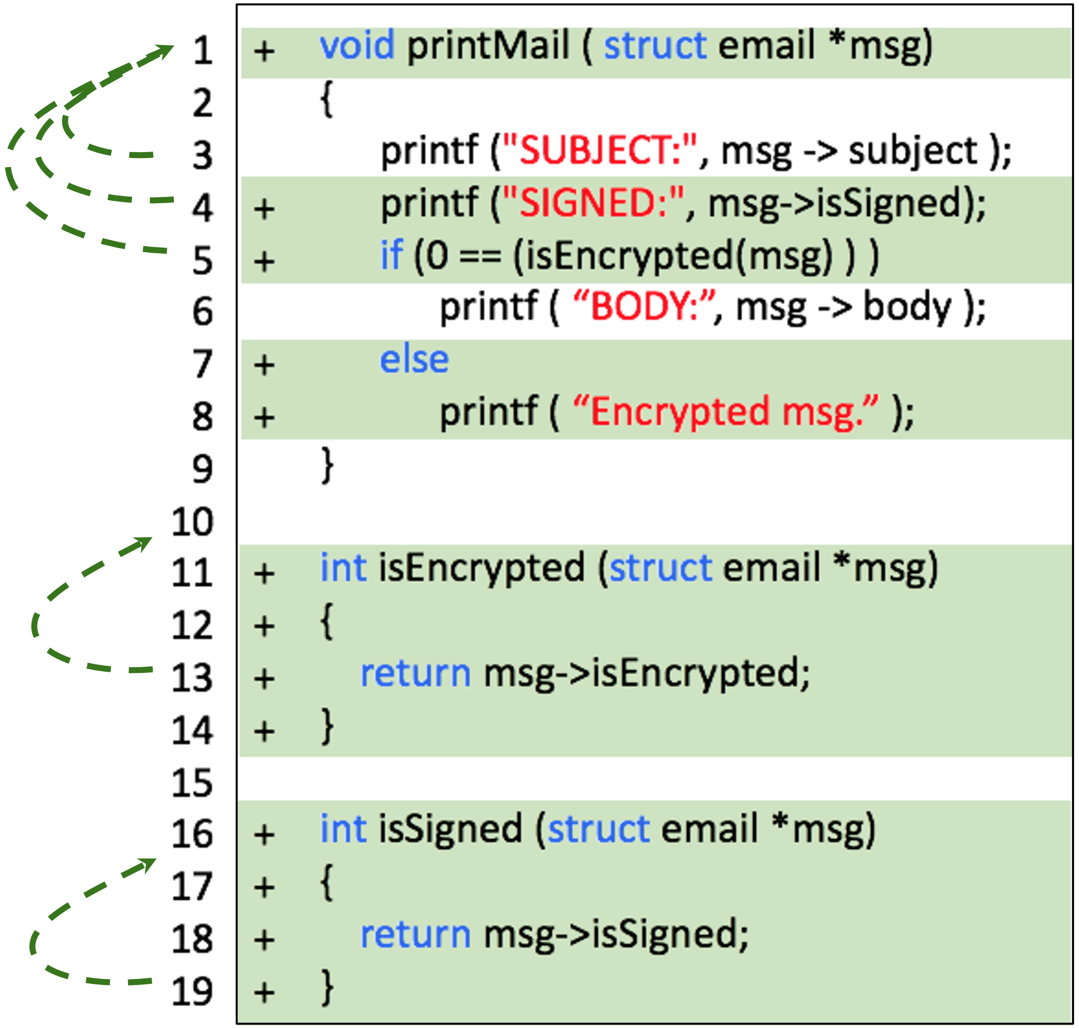


```
   struct email
   {
       char *subject;
       char *body;
+      int isEncrypted;
   };
   void printMail ( struct email *msg);

+  int isEncrypted (struct email *msg);

+  int isSigned (struct email *msg);
```

File 2: Email.c



```
+  void printMail ( struct email *msg)
   {
       printf ("SUBJECT:", msg -> subject );
+      printf ("SIGNED:", msg->isSigned);
+      if (0 == (isEncrypted(msg) ) )
           printf ( “BODY:”, msg -> body );
+      else
+          printf ( “Encrypted msg.” );
   }

+  int isEncrypted (struct email *msg)
+  {
+     return msg->isEncrypted;
+  }

+  int isSigned (struct email *msg)
+  {
+     return msg->isSigned;
+  }
```

3 Dependencies

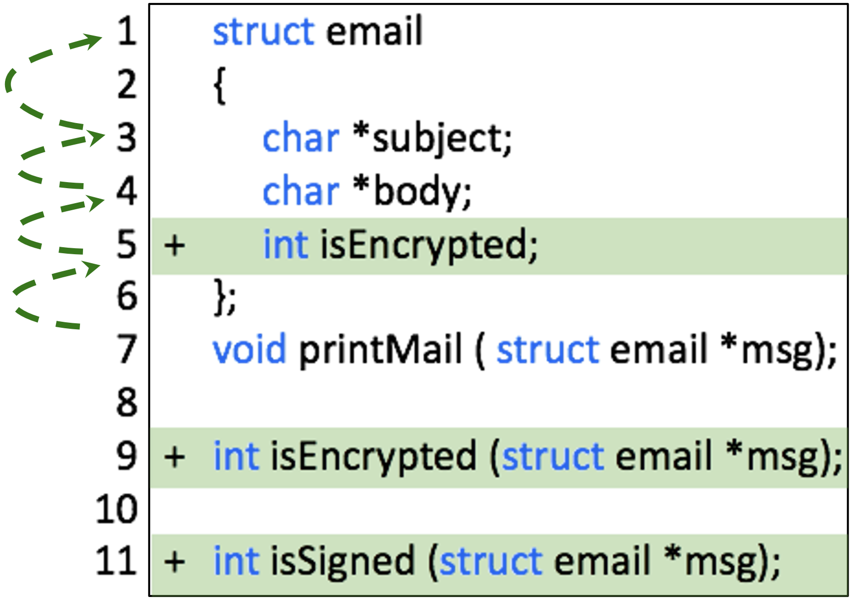
→ *DU – Definition-Usage*

→ *CF – Control Flow*

→ *H – Hierarchy; A - Adjacency*

Dependency Graph

File 1: Email.h

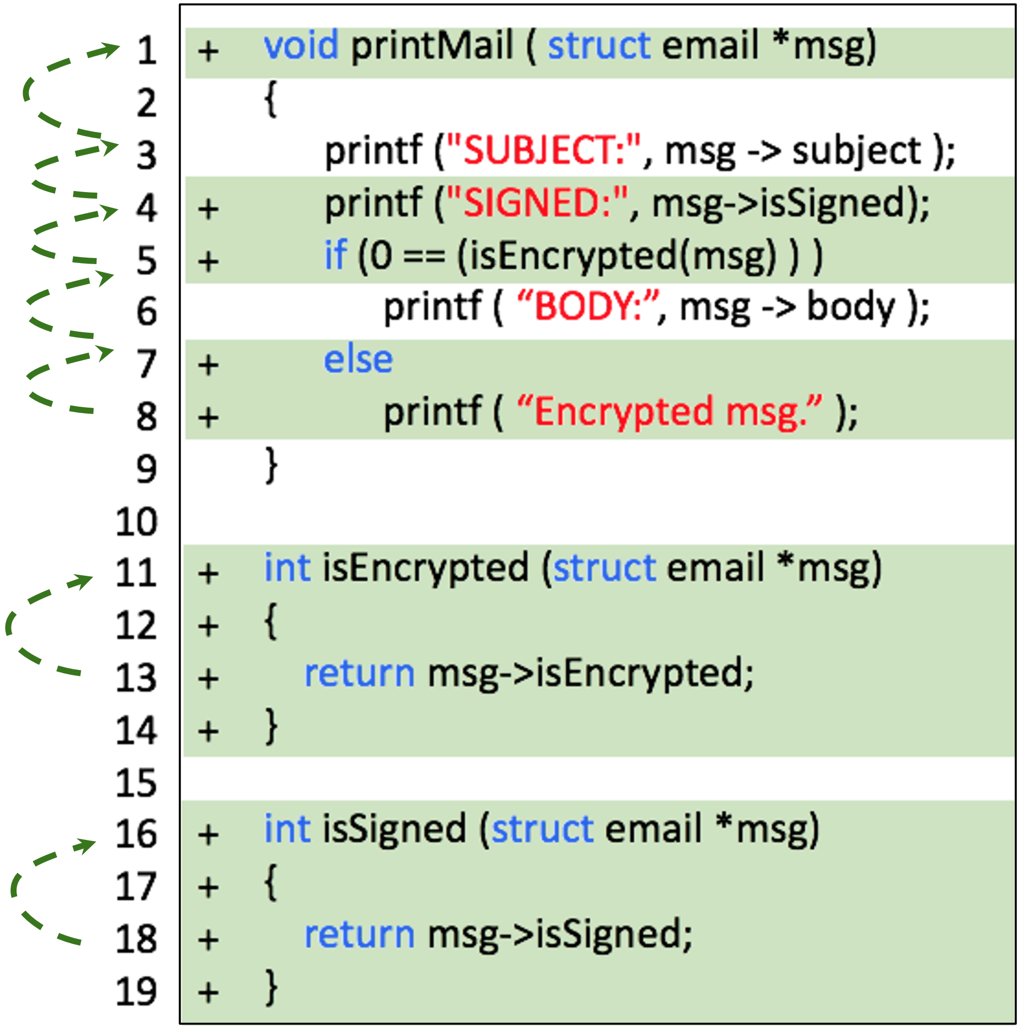


```
  struct email
  {
      char *subject;
      char *body;
+     int isEncrypted;
  };
  void printMail ( struct email *msg);

+ int isEncrypted (struct email *msg);

+ int isSigned (struct email *msg);
```

File 2: Email.c



```
+ void printMail ( struct email *msg)
  {
      printf ("SUBJECT:", msg -> subject );
+     printf ("SIGNED:", msg->isSigned);
+     if (0 == (isEncrypted(msg) ) )
          printf ( “BODY:”, msg -> body );
+     else
+         printf ( “Encrypted msg.” );
  }

+ int isEncrypted (struct email *msg)
+ {
+    return msg->isEncrypted;
+ }

+ int isSigned (struct email *msg)
+ {
+    return msg->isSigned;
+ }
```

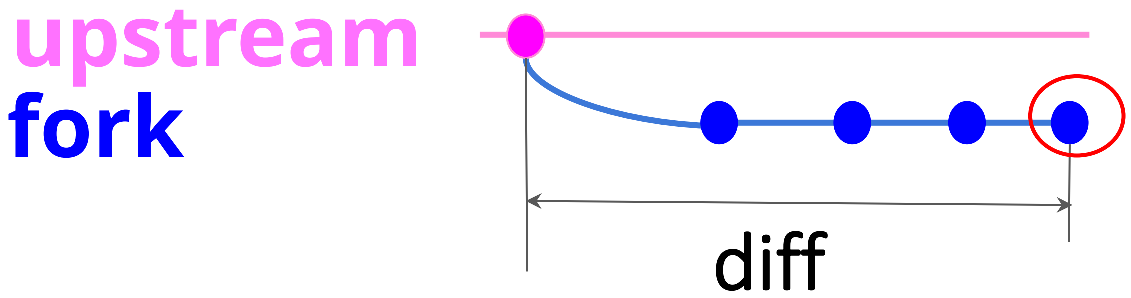
3 Dependencies

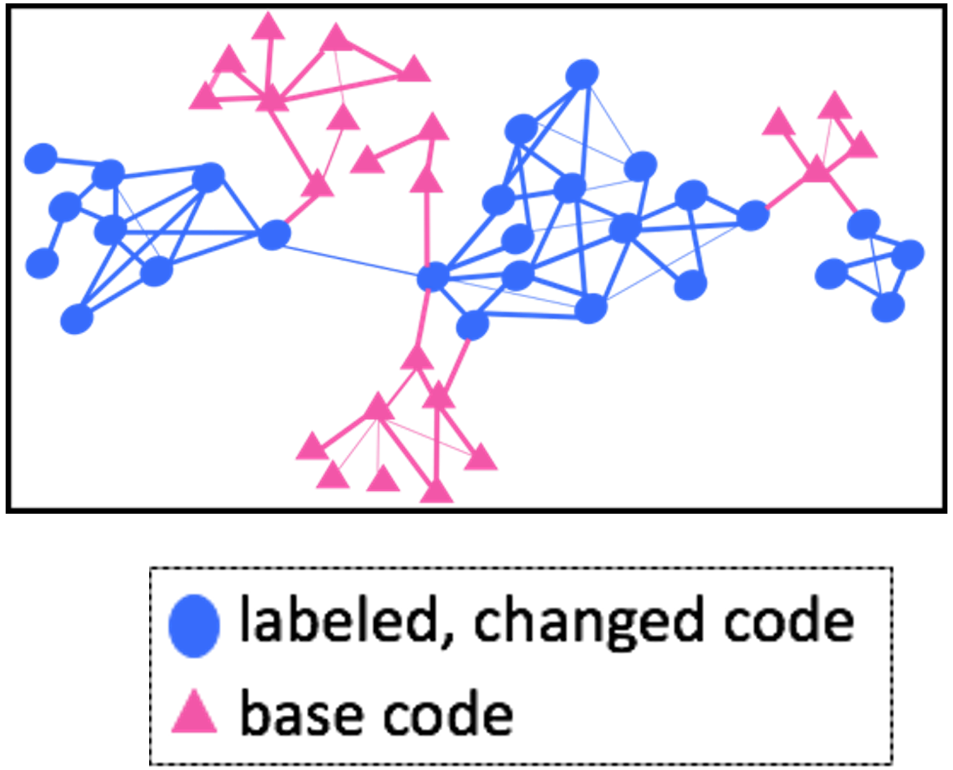
DU – Definition-Usage

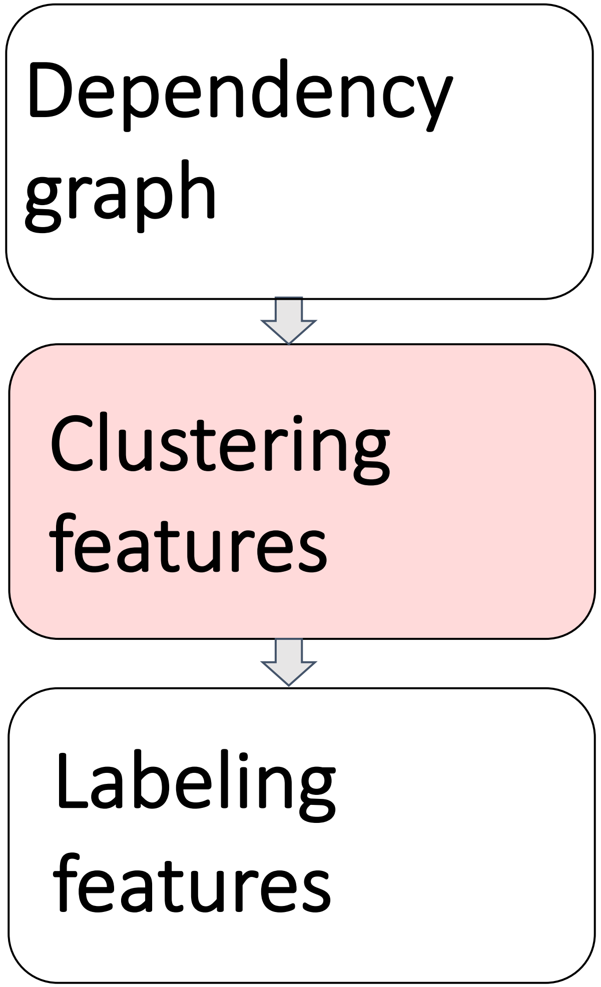
CF – Control Flow

H – Hierarchy; A - Adjacency

Dependency Graph

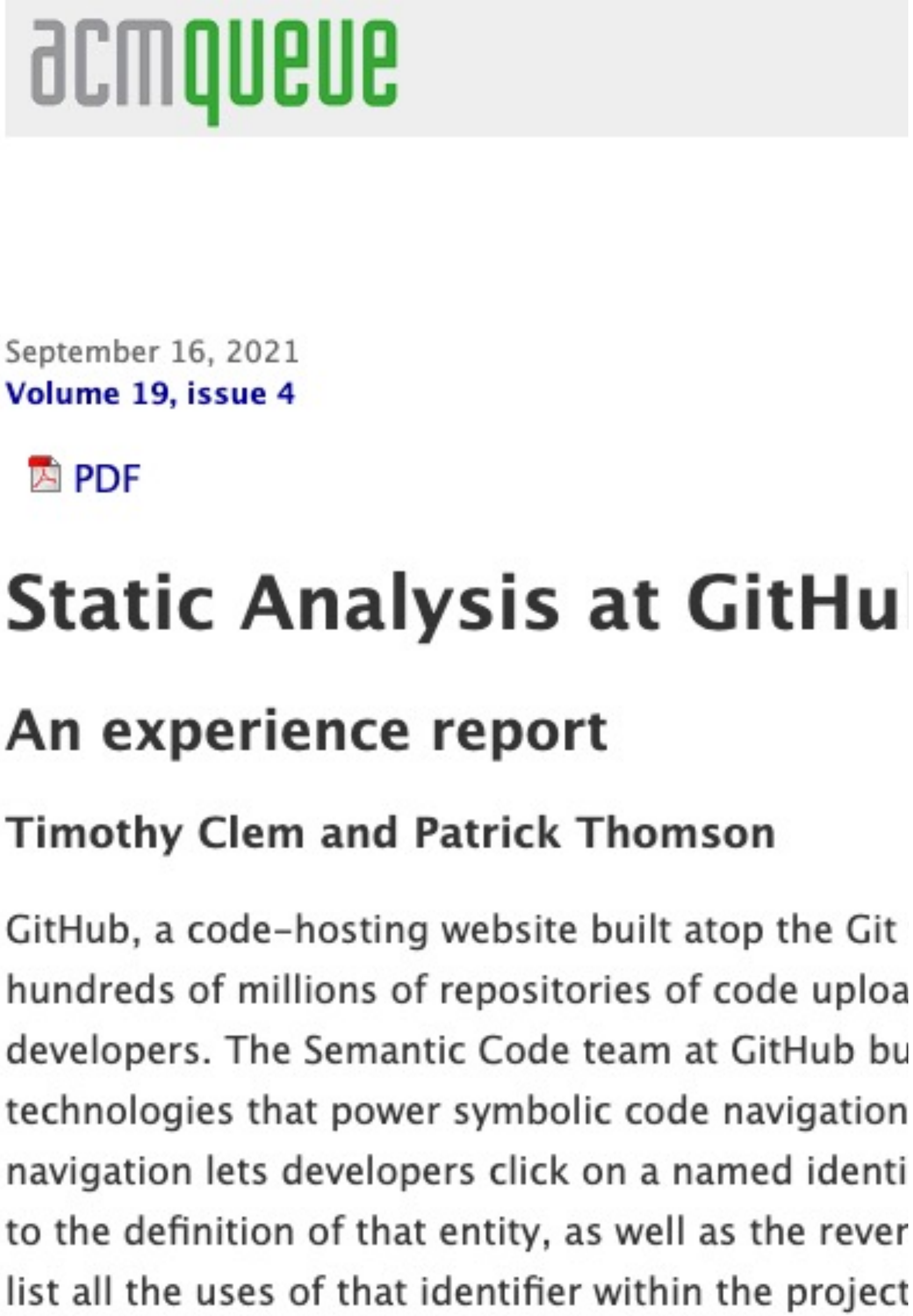





Dependency graph

Clustering features

Labeling features


acmqueue

September 16, 2021
Volume 19, issue 4

PDF

Static Analysis at GitHu

An experience report

Timothy Clem and Patrick Thomson

GitHub, a code-hosting website built atop the Git
hundreds of millions of repositories of code uploa
developers. The Semantic Code team at GitHub bu
technologies that power symbolic code navigation
navigation lets developers click on a named identi
to the definition of that entity, as well as the rever
list all the uses of that identifier within the project

Static Analysis @ GitHub

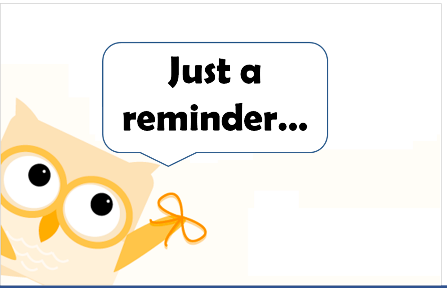
TIMOTHY CLEM AND PATRICK THOMSON

AN EXPERIENCE REPORT

GitHub, a code-hosting website built atop the Git version-control system, hosts hundreds of millions of repositories of code uploaded by more than 65 million developers. The Semantic Code team at GitHub builds and operates a suite of technologies that power symbolic code navigation on github.com. Symbolic code navigation lets developers click

Principle techniques

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



Wouldn't it be nice if we could learn about the program's memory usage as it was running?

How can we tackle this problem?

- Testing:

- Inspection:

- Static analysis:

Dynamic analysis:
learn about a program’s properties by executing it

- How can we learn about properties that are more interesting than “did this test pass” (e.g., memory use)?
- Short answer: examine program state throughout/after execution by gathering additional information.

Common dynamic analyses

- Coverage
- Performance
- Memory usage
- Security properties
- Concurrency errors
- Invariant checking
- Fault localization
- Anomaly detection

Collecting execution info

- Instrument at compile time
- Run on a specialized VM
- Instrument or monitor at runtime

Collecting execution info

- Instrument a
  - e.g., As
- Run on a specia
  - e.g., valgrind
- Instrument or m nitor at ru ne
  - a ires)
  - bols to profile/monitor
  - (

Note: some of these methods require a *static* pre processing step!

Avoid mixing up static things done to collect info and the dynamic analyses that use the info.

Example: Test Coverage

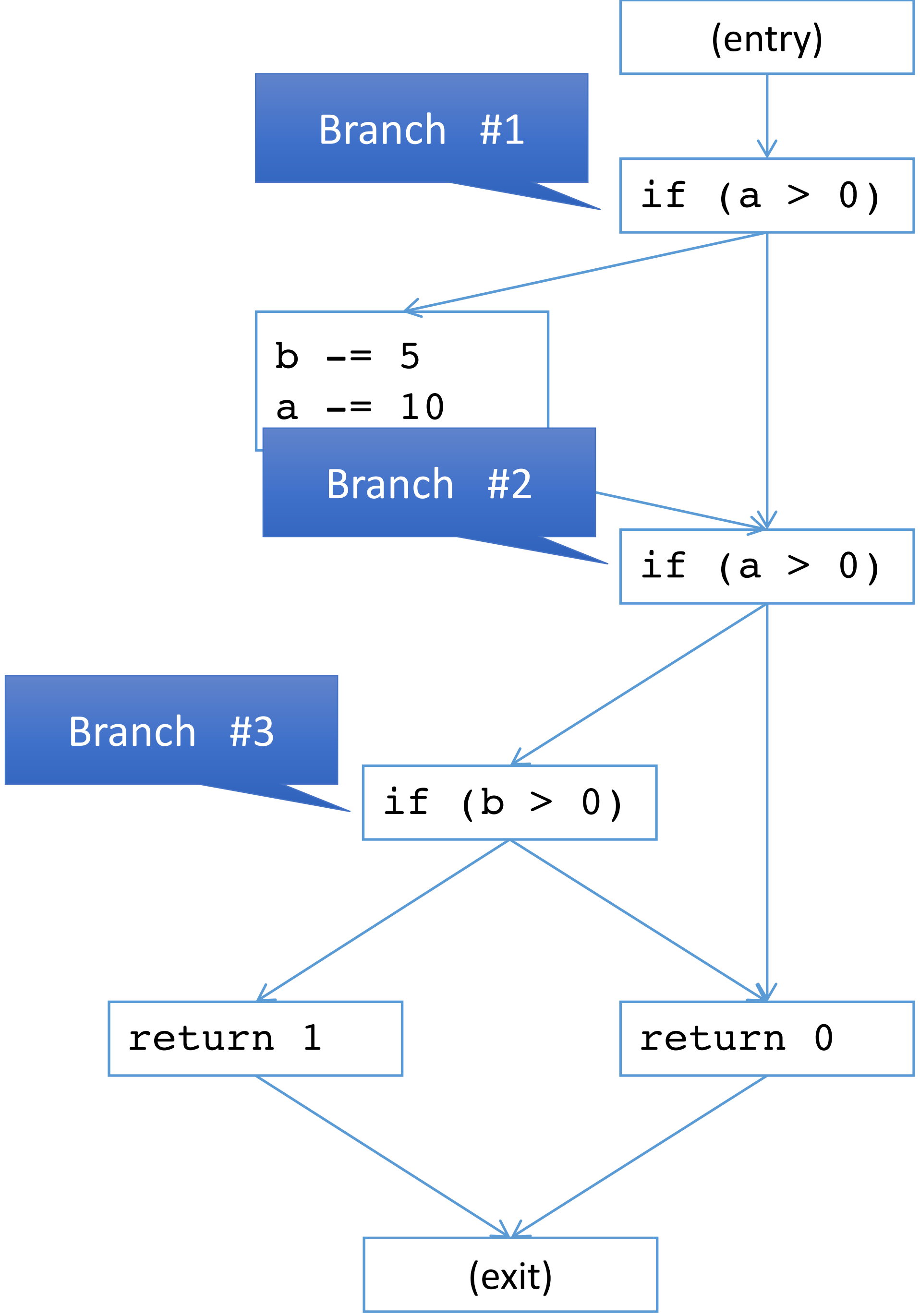
- Statement: Has each statement in the program been executed?
- → Branch: Has each of each control structure been executed?
- Function: Has each function in the program been called?
- Path: requires that all paths through the Control Flow Graph are covered.
- …

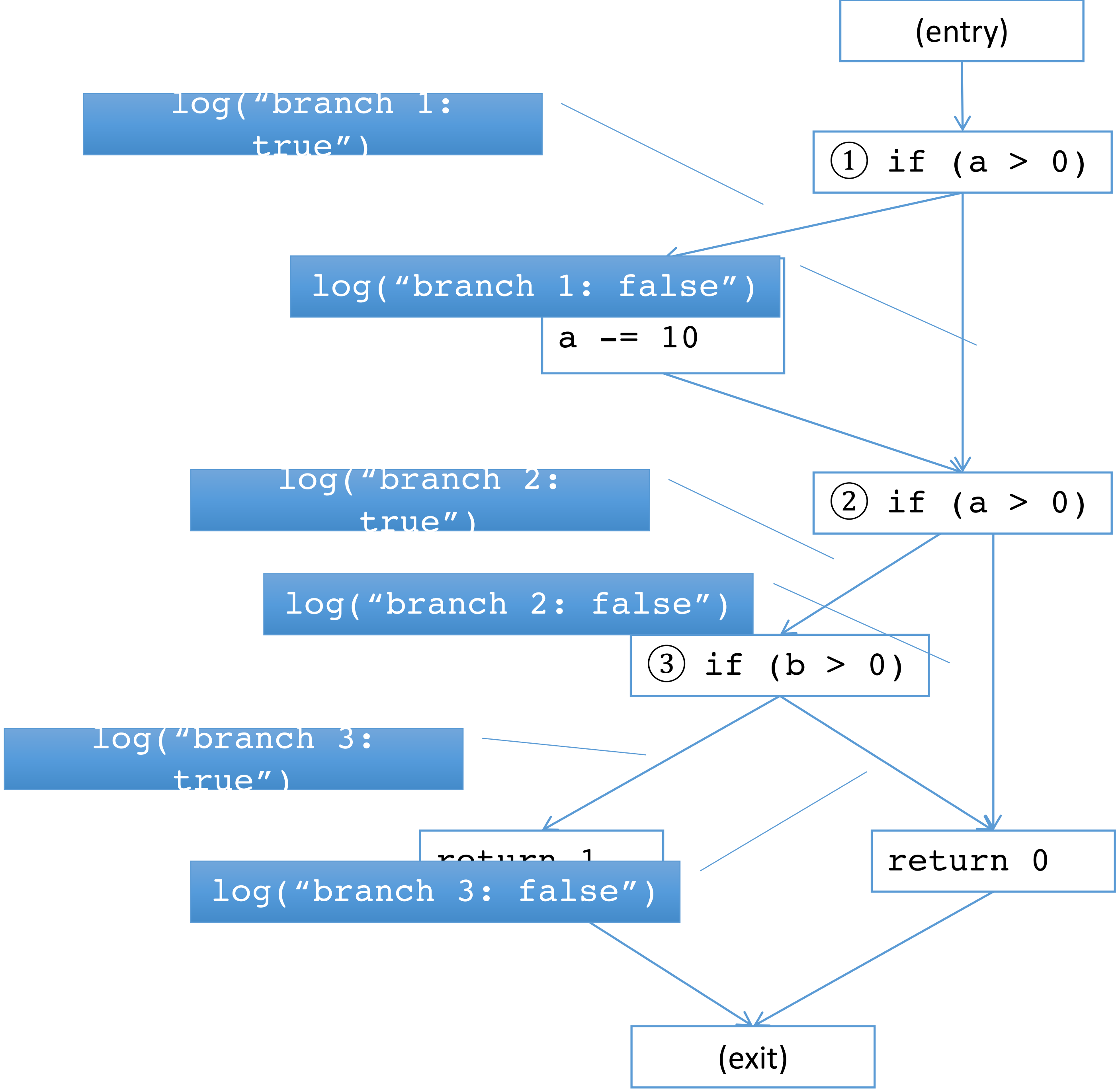
Q: How might tools that compute test suite coverage work?

Instrumentation: a simple example

- One option: *instrument* the code to track a certain type of data as the program executes.
  - **Instrument:** add of special code to track a certain type of information as a program executes.
  - Rephrase: insert logging statements (e.g., at compile time).
- What do we want to log/track for branch coverage computation?

```
int foobar(a,b) {
  if (a > 0) {
    b -= 5;
    a -= 10;
  }
  if(a > 0) {
    if (b > 0)
      return 1;
  }
  return 0;
}
```





(entry)

① if (a > 0)

log(“branch 1: true”)

log(“branch 1: false”)

a -= 10

② if (a > 0)

log(“branch 2: true”)

log(“branch 2: false”)

③ if (b > 0)

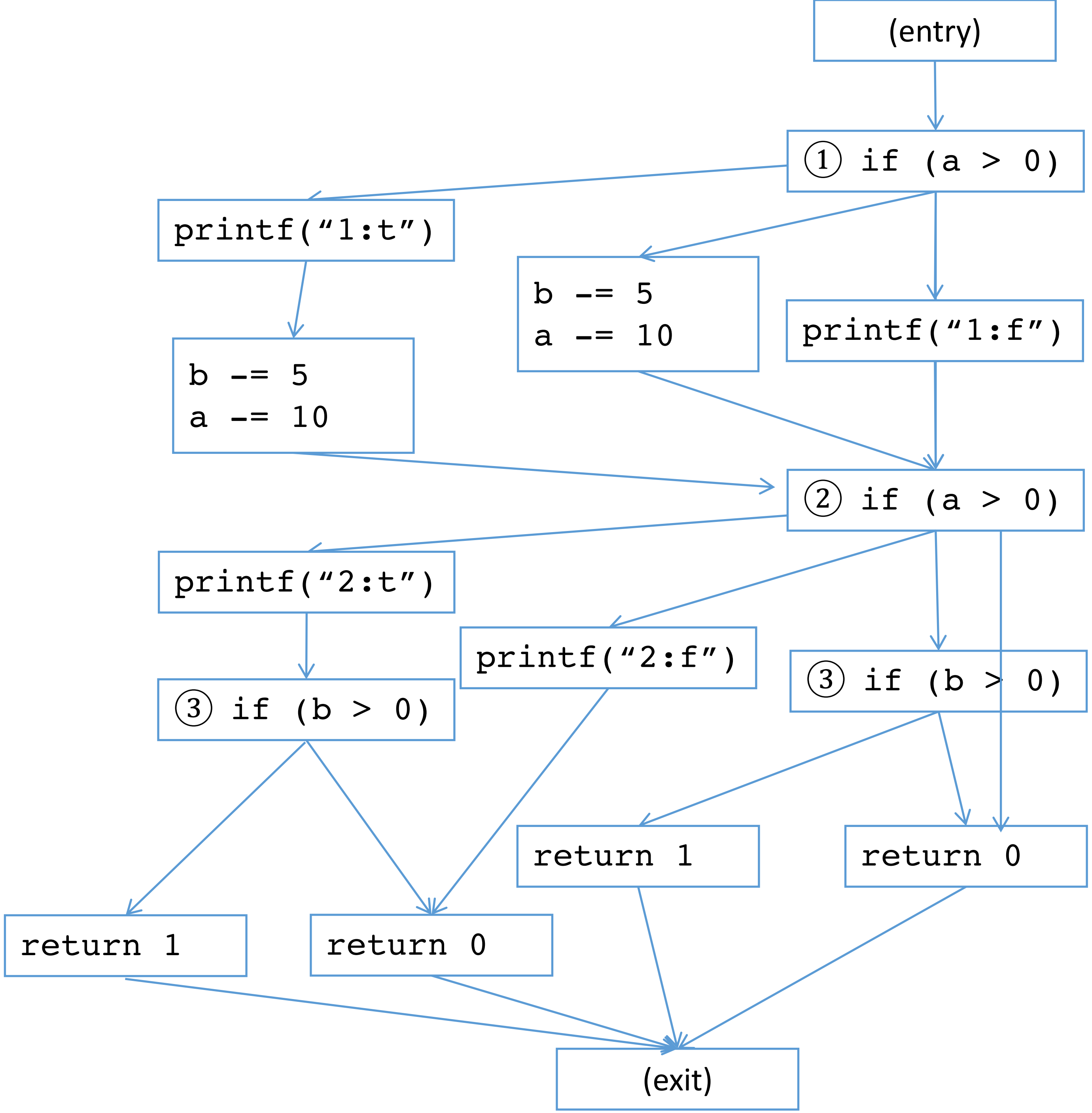
log(“branch 3: true”)

return 1

log(“branch 3: false”)

return 0

(exit)



(entry)

① if (a > 0)

printf(“1:t”)

b -= 5
a -= 10

b -= 5
a -= 10

printf(“1:f”)

② if (a > 0)

printf(“2:t”)

printf(“2:f”)

③ if (b > 0)

③ if (b > 0)

return 1

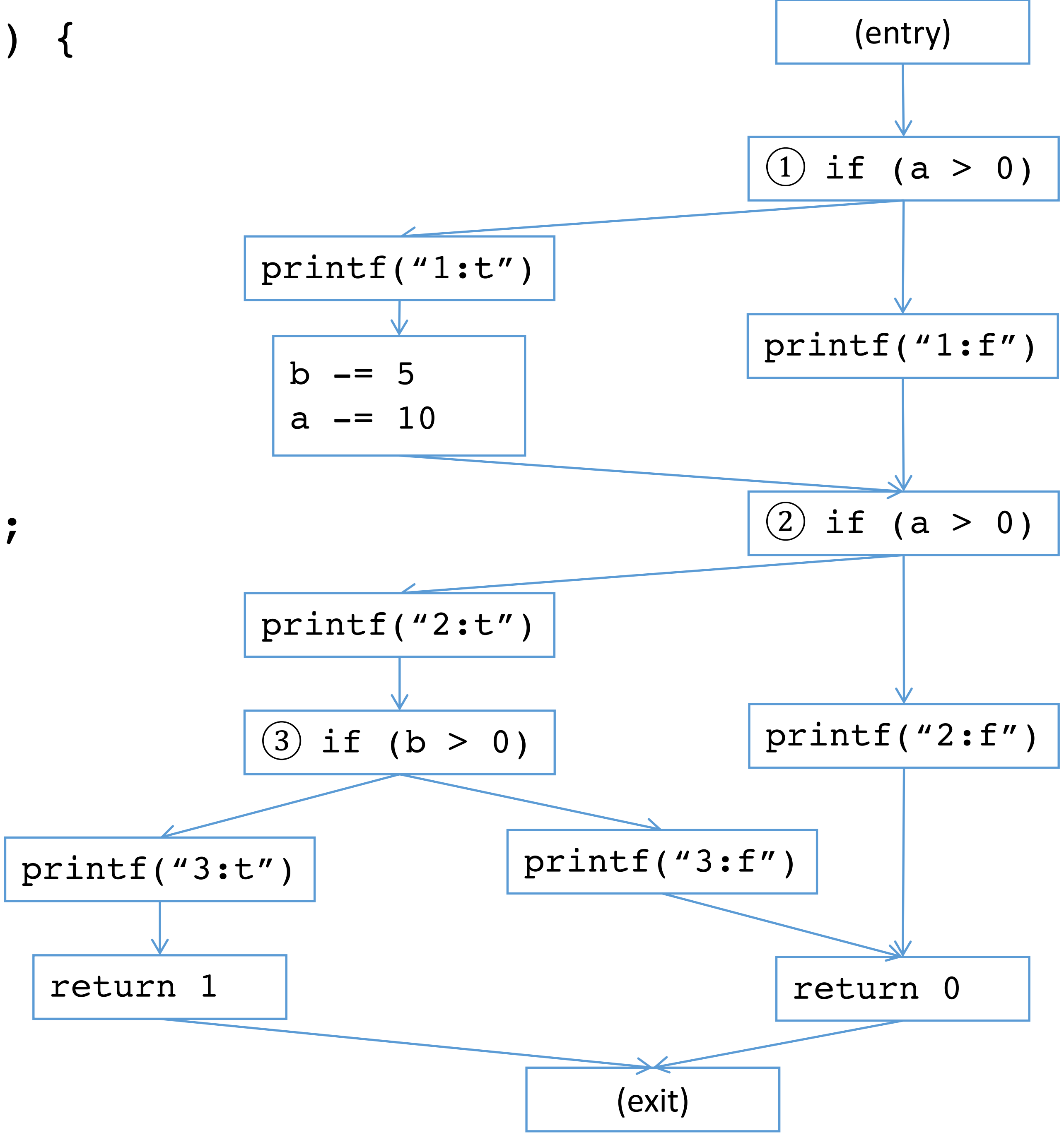
return 0

return 1

return 0

(exit)

```
int foobar(a,b) {
 if (a > 0) {
  b -= 5;
  a -= 10;
 }
 if(a > 0) {
  if (b > 0)
   return 1;
 }
 return 0;
}
```



(entry)

① if (a > 0)

printf(“1:t”)

printf(“1:f”)

b -= 5
a -= 10

② if (a > 0)

printf(“2:t”)

printf(“2:f”)

③ if (b > 0)

printf(“3:t”)

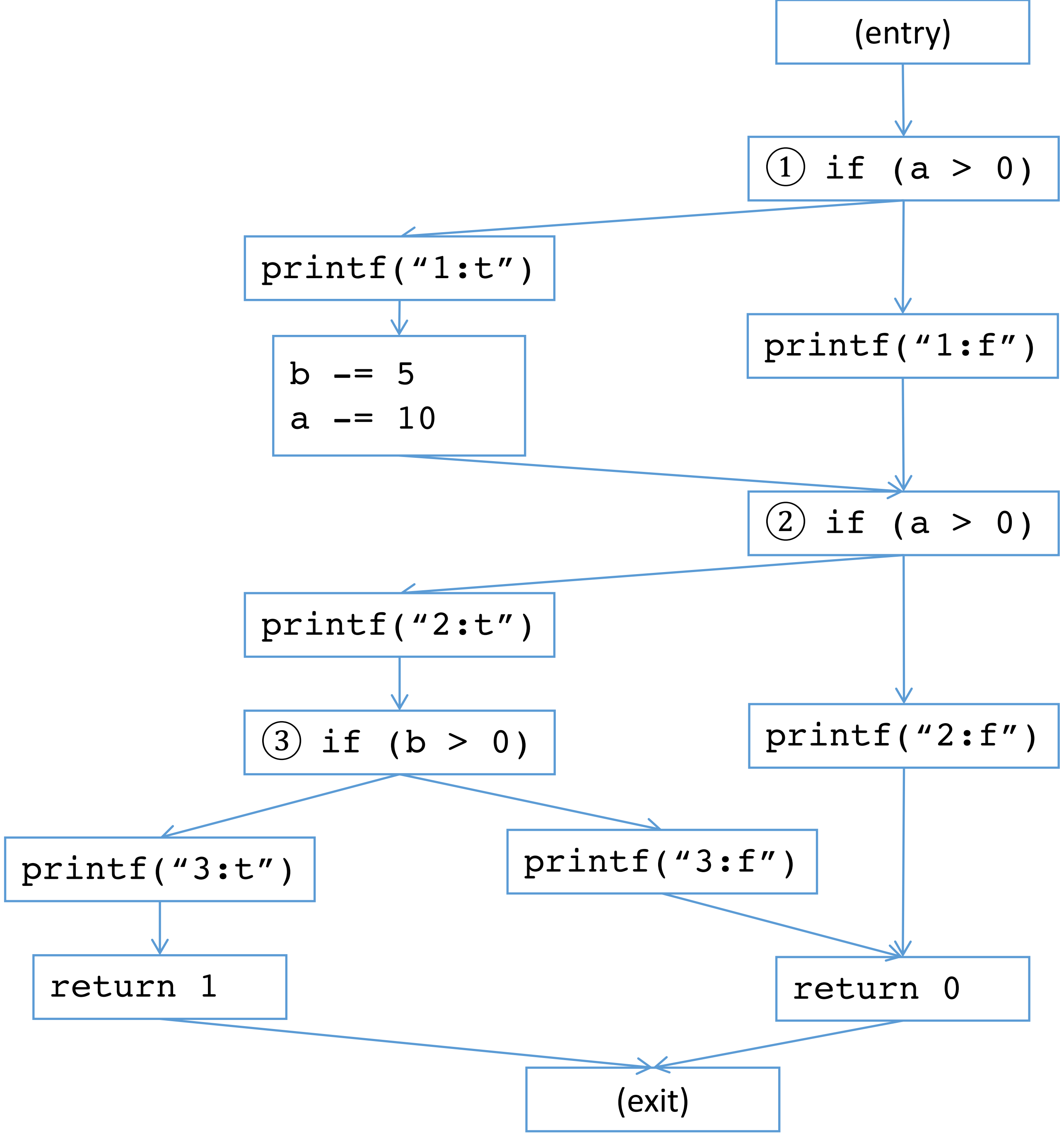
printf(“3:f”)

return 1

return 0

(exit)

```c
int foobar(a,b) {
  if (a > 0) {
    printf(“1:t”);
    b -= 5;
    a -= 10;
  } else {
    printf(“1:f”);
  }
  if(a > 0) {
    printf(“2:t”);
    if (b > 0) {
      printf(“3:t”);
      return 1;
    } else {
      printf(“3:f”);
    }
  } else {
    printf(“2:f”);
  }
  return 0;
}
```



(entry)

① if (a > 0)

printf(“1:t”)

b -= 5
a -= 10

printf(“1:f”)

② if (a > 0)

printf(“2:t”)

③ if (b > 0)

printf(“2:f”)

printf(“3:t”)

printf(“3:f”)

return 1

return 0

(exit)

```
int foobar(a,b) {
  if (a > 0) {
    printf(“1:t ”);
    b -= 5;
    a -= 10;
  } else {
    printf(“1:f ”);
  }
  if(a > 0) {
     printf(“2:t ”);
     if (b > 0) {
       printf(“3:t ”);
       return 1;
     } else {
       printf(“3:f ”);
     }
   } else {
     printf(“2:f ”);
   }
   return 0;
 }
```

- Test cases: (0,0), (1,0), (11,0), (11,6)
  - foobar(0,0): “1:f 2:f ”
  - foobar(1,0): “1:t 2:f ”
  - foobar(11,0): “1:t 2:t 3:f ”
  - foobar(11,6): “1:t 2:t 3:t “

Assuming we saved how many branches were in this method when we instrumented it, we could now process these logs to compute branch coverage.

Limitation: Dynamic analysis

- Cost

  Performance overhead for recording

  - Acceptable for use in testing?
  - Acceptable for use in production?

Costs

Performance overhead for recording

- Acceptable for use in testing?
- Acceptable for use in production?

Very input dependent

- Good if you have lots of tests!
- Can also use logs from live software runs that include actual user interactions (sometimes, see next slides).
- Or: specific inputs that replicate specific defect scenarios (like memory leaks).

Too much data

- Logging events in large and/or long-running programs (even for just one property!) can result in HUGE amounts of data.
- How do you process it?
    - Common strategy: sampling

Lifecycle

- During QA
    - Instrument code for tests
    - Let it run on all regression tests
    - Store output as part of the regression
- During Production
    - Only works for web apps
    - Instrument a few of the servers
        - Use them to gather data
        - Statistical analysis, similar to seeding defects in code reviews
    - Instrument all of the servers
        - Use them to protect data

Summary

- Dynamic analysis: selectively record data at runtime
- Data collection through instrumentation
- Integrated tools exist (e.g., profilers)
- Analyzes only concrete executions, runtime overhead