Intro to QA, and Static Analysis

Claire Le Goues

October 22, 2019

Learning goals

- Define software analysis.
- Reason about QA activities with respect to coverage and coverage/adequacy criteria, both traditional (structural) and non-traditional.
- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Explain at a high level why static analyses cannot be sound, complete, and terminating; assess tradeoffs in analysis design.
- Give tradeoffs and identify when various techniques might be useful.

HOW DO YOU KNOW THAT YOUR PROGRAM WORKS?

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…

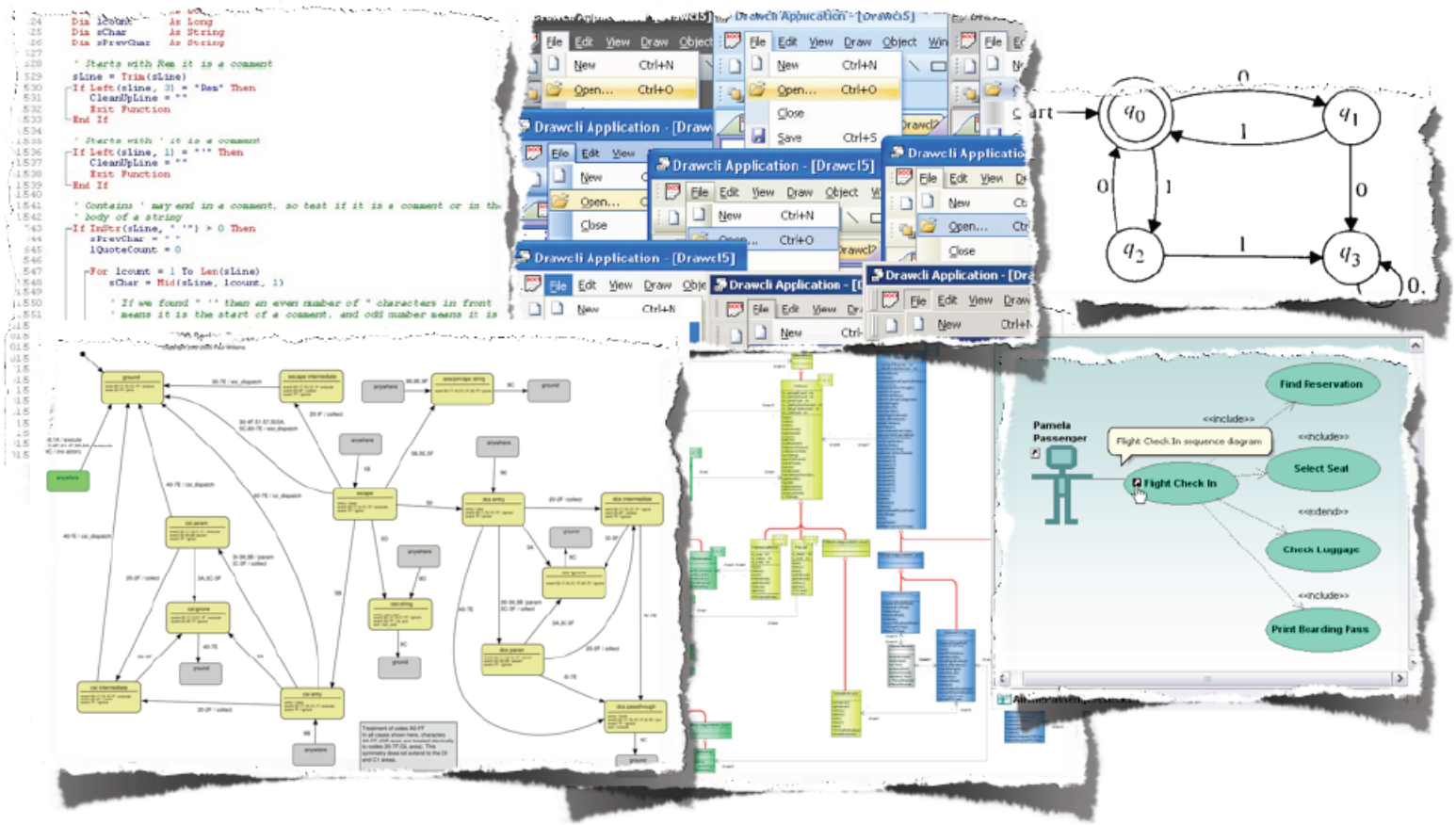
Principle techniques

- **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.

“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?

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

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

Regression testing (redux)

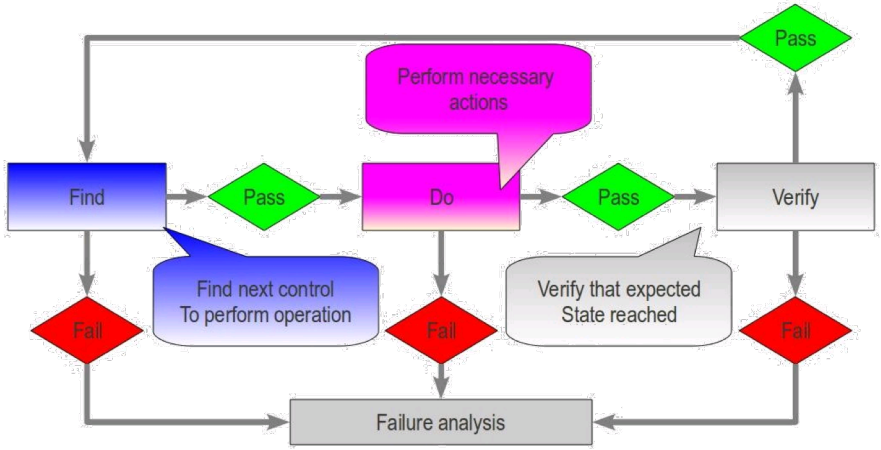
- What is “covered” by a set of regression tests?
- Why do we do regression testing?
- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - **Code1** + **TestSet** □ **TestResults1**
    - **Code2** + **TestSet** □ **TestResults2**
  - As code evolves, 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.

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

- 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!

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

Completeness?

- Statistical thresholds
  - Defects reported/repaired
  - Relative proportion of defect kinds
- Coverage criteria
  - E.g., 100% coverage required for avionics software, but distorts the software
  - Matrix: Map test cases to requirements use cases
- Can look at historical data
  - Rule of thumb: when error detection rate drops (implies diminishing returns for testing investment)
- Best practice: continuous quality assessment, testing, controls, and gates throughout development.

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.

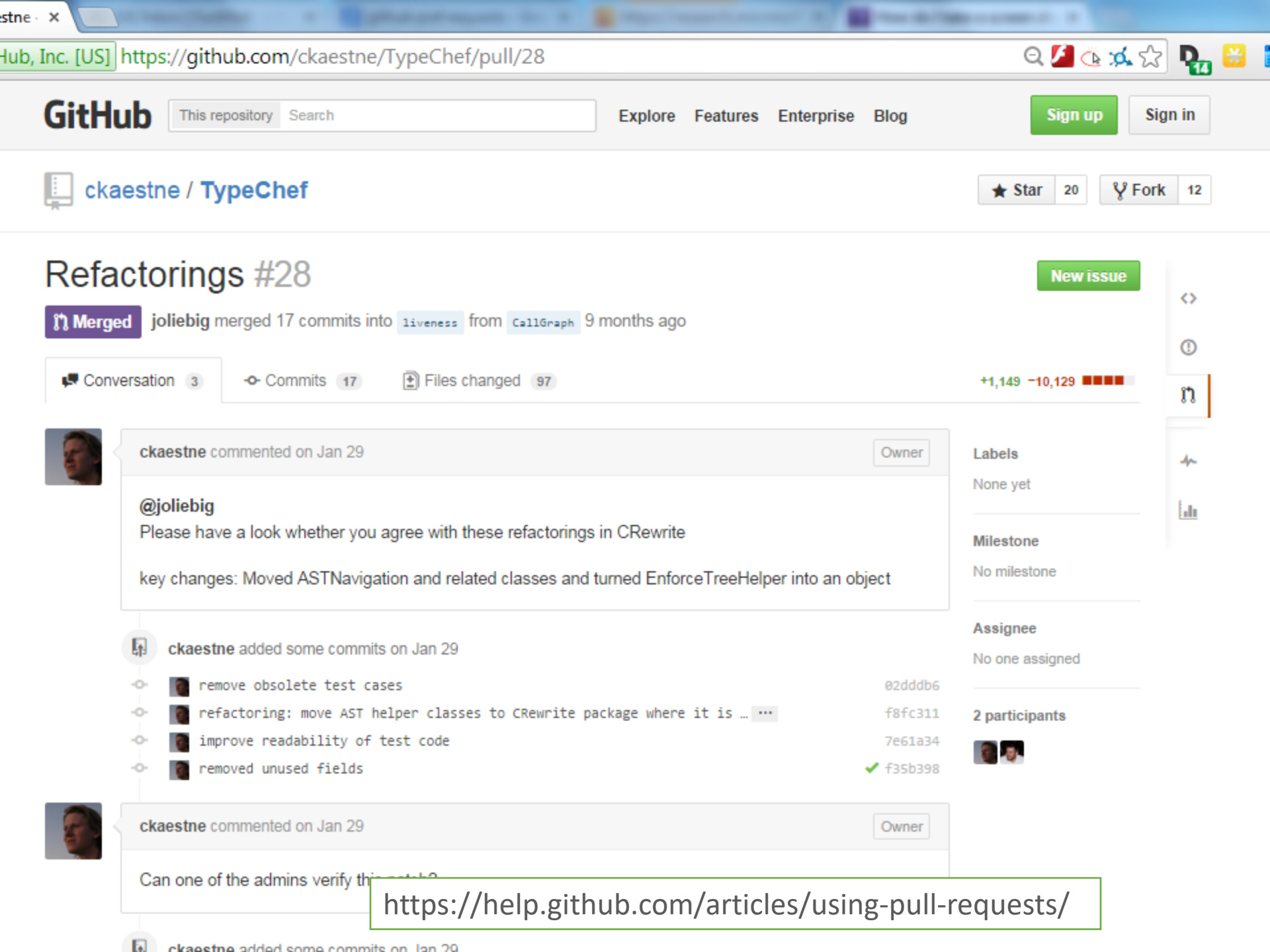
SOFTWARE PEER REVIEWS

"Many eyes make all bugs shallow"

Standard Refrain in Open Source

"Have peers, rather than customers, find defects"

Karl Wiegers


https://github.com/ckaestne/TypeChef/pull/28
GitHub
This repository
Search
Explore
Features
Enterprise
Blog
Sign up
Sign in
ckaestne / TypeChef
Star 20
Fork 12
Refactorings #28
New issue
Merged
joliebig merged 17 commits into liveness from CallGraph 9 months ago
Conversation 3
Commits 17
Files changed 97
+1,149 −10,129
ckaestne commented on Jan 29
Owner
@joliebig
Please have a look whether you agree with these refactorings in CRewrite
key changes: Moved ASTNavigation and related classes and turned EnforceTreeHelper into an object
ckaestne added some commits on Jan 29
remove obsolete test cases
02dddb6
refactoring: move AST helper classes to CRewrite package where it is …
f8fc311
improve readability of test code
7e61a34
removed unused fields
f35b398
ckaestne commented on Jan 29
Owner
Can one of the admins verify this patch?
Labels
None yet
Milestone
No milestone
Assignee
No one assigned
2 participants
https://help.github.com/articles/using-pull-requests/

Isn’t testing sufficient?

- Errors can mask other errors
- Only completed implementations can be tested (esp. scalability, performance)
- Design documents cannot be tested
- Tests don’t check code quality
- Many quality attributes (eg., security, compliance, scalability) are difficult to test

Static Analysis as "Automated Inspection"

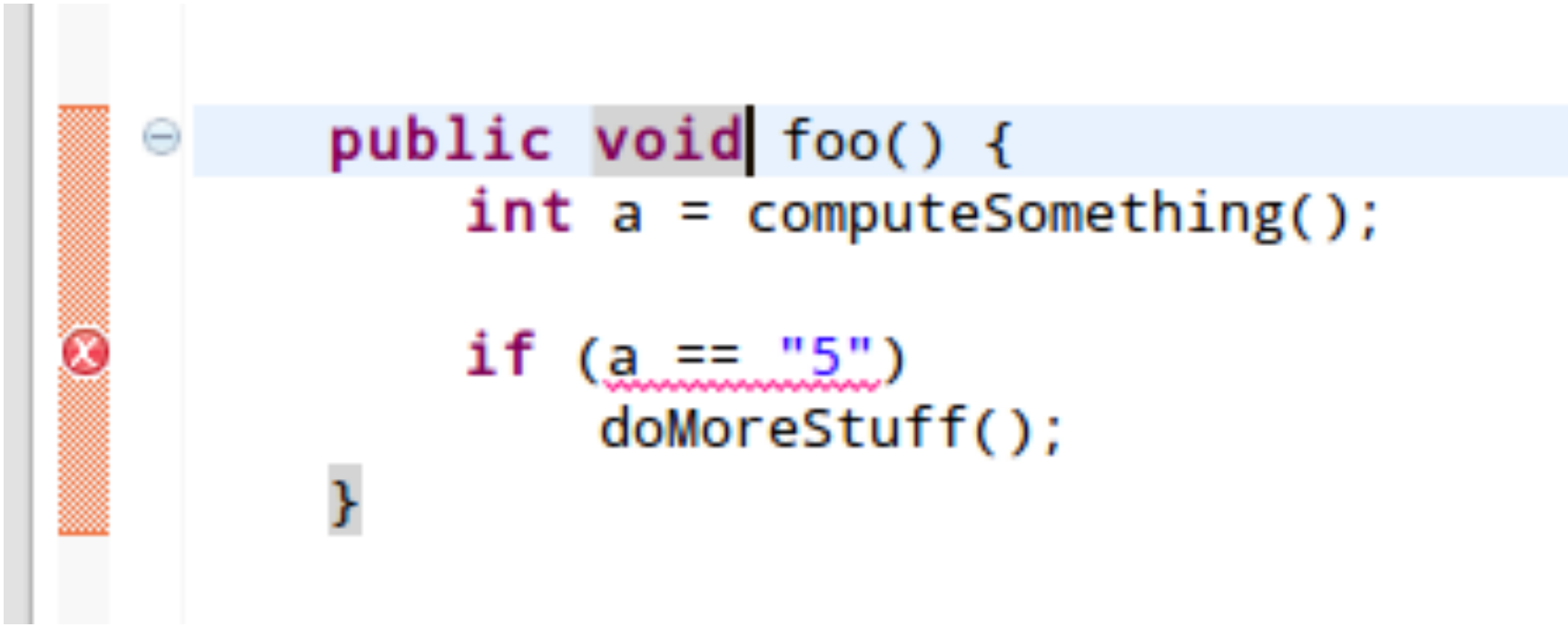
- Low-level issues often checked by compiler or static analysis tool
  - Initializing variables; providing correct number of parameters
  - Closing file handles; freeing memory
  - Code style issues
- Root cause analysis -> Build new static checkers
- Enables inspections to focus on important issues

Which rules should ALWAYS be enforced?

What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.
  - Does not execute code!
- **Abstraction:** produce a representation of a program that is simpler to analyze.
  - Results in fewer states to explore; makes difficult problems tractable.
- Check if a **particular property** holds over the entire state space:
  - Liveness: “something good eventually happens.”
  - Safety: “this bad thing can’t ever happen.”

Abstraction?



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

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```

ERROR: function returns with interrupts disabled!

With thanks to Jonathan Aldrich; example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Could you have found them?

- How often would those bugs trigger?
- Driver bug:
  - What happens if you return from a driver with interrupts disabled?
  - Consider: that's one function
    - …in a 2000 LOC file
    - …in a module with 60,000 LOC
    - …IN THE LINUX KERNEL
- **Moral:** *Some defects are very difficult to find via testing, inspection.*

Defects of interest…

- Are on uncommon or difficult-to-force execution paths.
  - Which is why it's hard to find them via testing.
- Executing (or interpreting/otherwise analyzing) all paths concretely to find such defects is infeasible.
- **What we really want to do is check the entire possible state space of the program for particular properties.**

Defects Static Analysis can Catch

- **Defects that result from inconsistently following simple, mechanical design rules.**
  - **Security:** Buffer overruns, improperly validated input.
  - **Memory safety:** Null dereference, uninitialized data.
  - **Resource leaks:** Memory, OS resources.
  - **API Protocols:** Device drivers; real time libraries; GUI frameworks.
  - **Exceptions:** Arithmetic/library/user-defined
  - **Encapsulation:** Accessing internal data, calling private functions.
  - **Data races:** Two threads access the same data without synchronization
- **Increasingly used at commit time or otherwise throughout QA in practice.**

Key: check compliance to simple, mechanical design rules

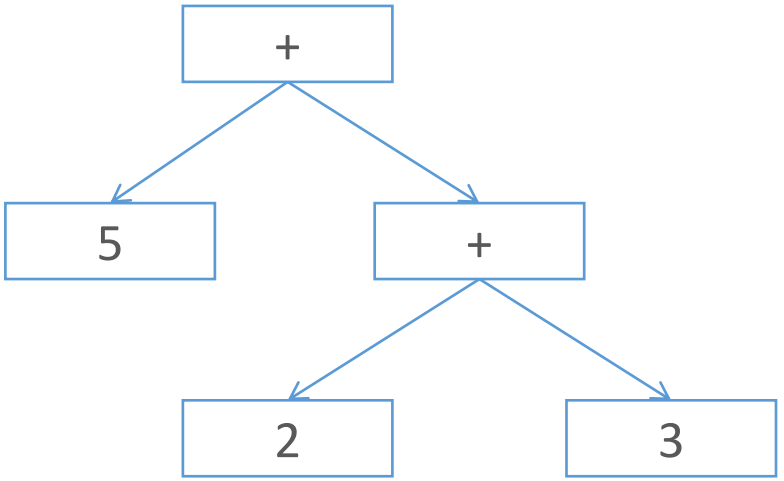
Two fundamental concepts

- **Abstraction.**
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- **Programs as data.**
  - Programs are just trees/graphs!
  - …and we know lots of ways to analyze trees/graphs, right?

Abstraction: abstract syntax tree

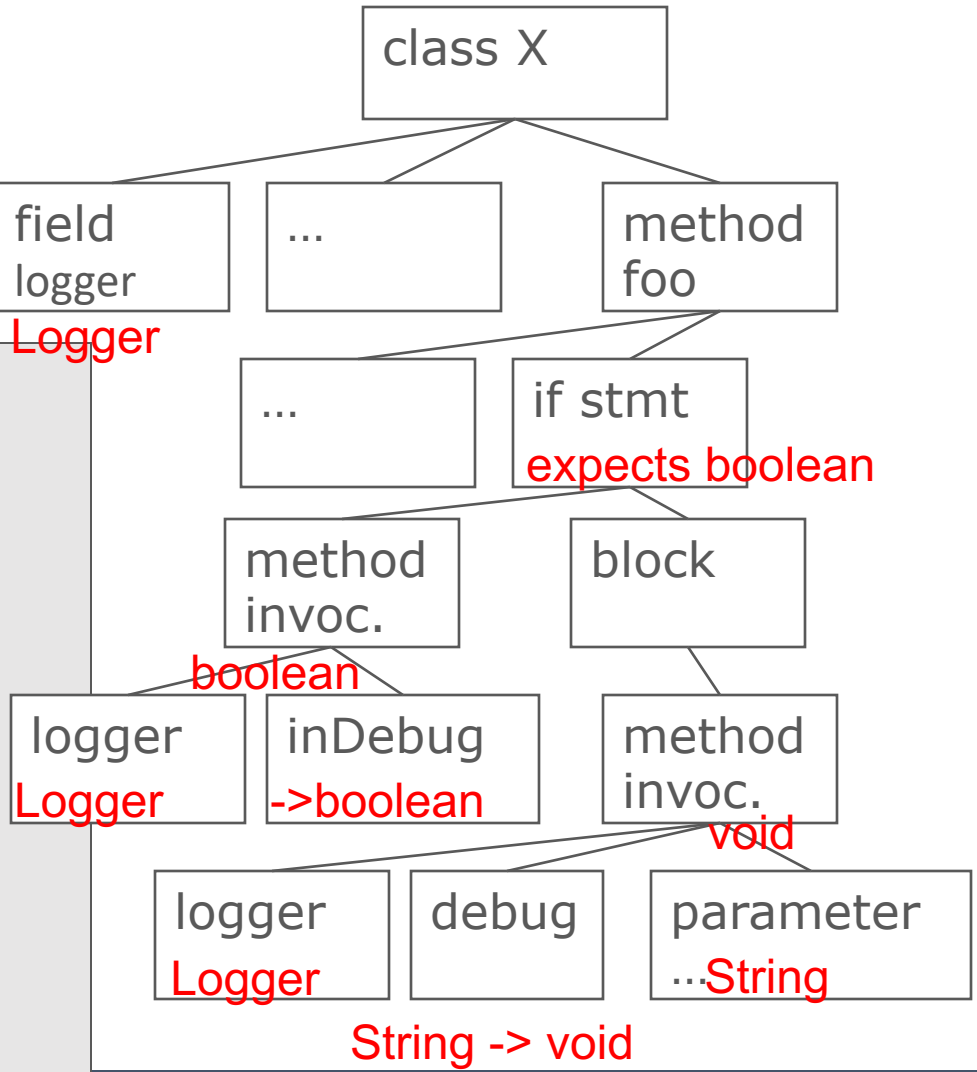
- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn't represent every detail (like parentheses); these can be inferred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

- Example: 5 + (2 + 3)

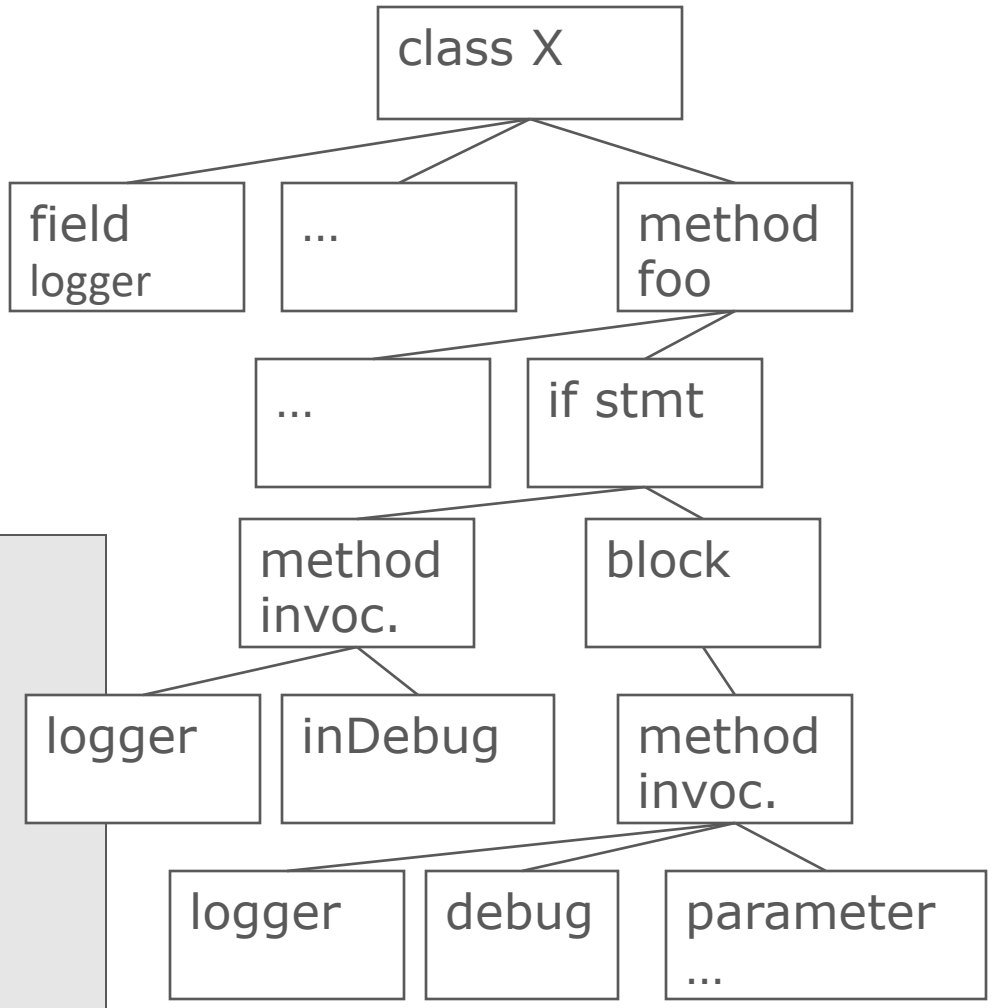


Type checking

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



Structural Analysis

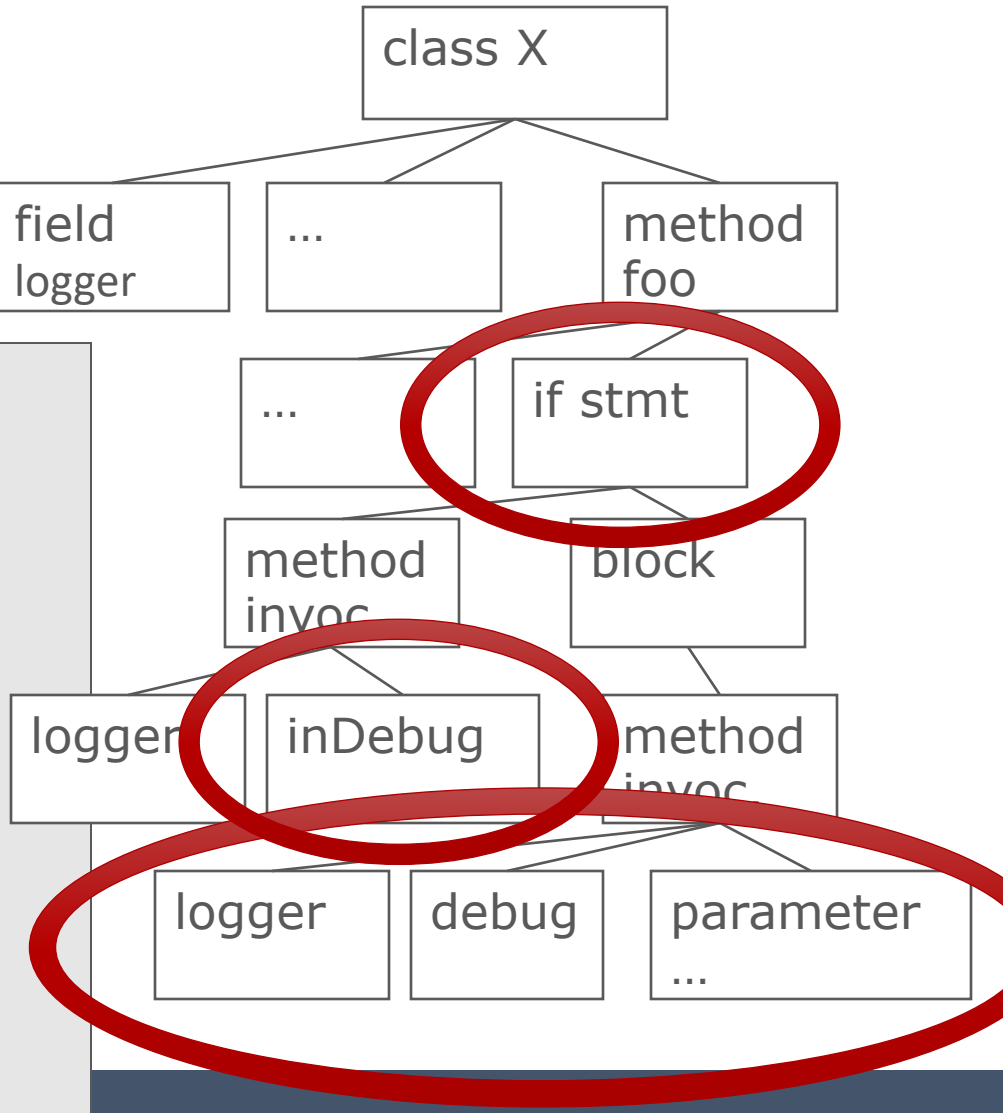



```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug(“We have ” +
conn + “connections.”);
    }
  }
}
```

Abstract syntax tree walker

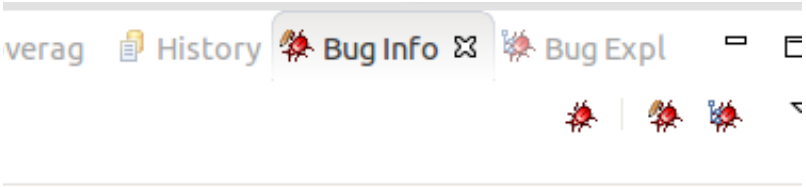
- Check that we don't create strings outside of a `Logger.inDebug` check
- Abstraction:
  - Look only for calls to `Logger.debug()`
  - Make sure they're all surrounded by if (`Logger.inDebug()`)
- Systematic: Checks all the code
- Known as an Abstract Syntax Tree (AST) walker
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way

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



Bug finding

```
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```



Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank**: Troubling (14)
Pattern: NP_BOOLEAN_RETURN_NULL
Type: NP, **Category**: BAD_PRACTICE (Bad practice)

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*.
- 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.

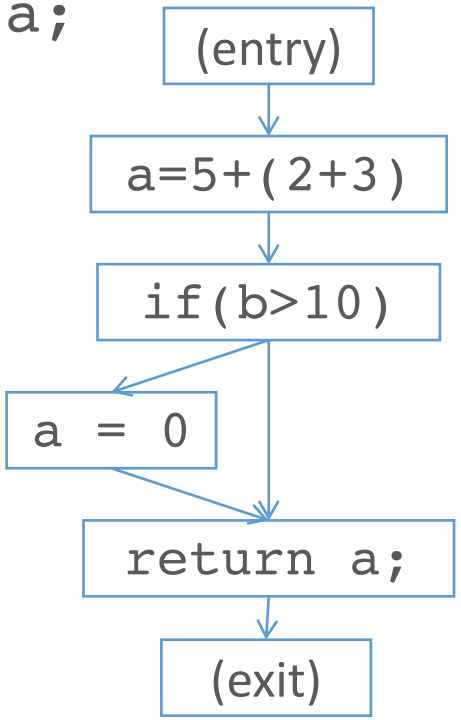
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- 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.

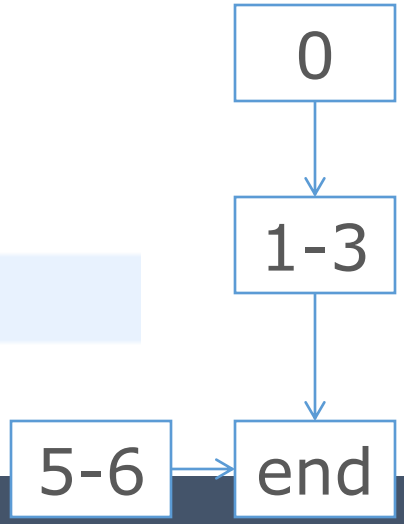
Abstraction: Control flow graphs

- 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

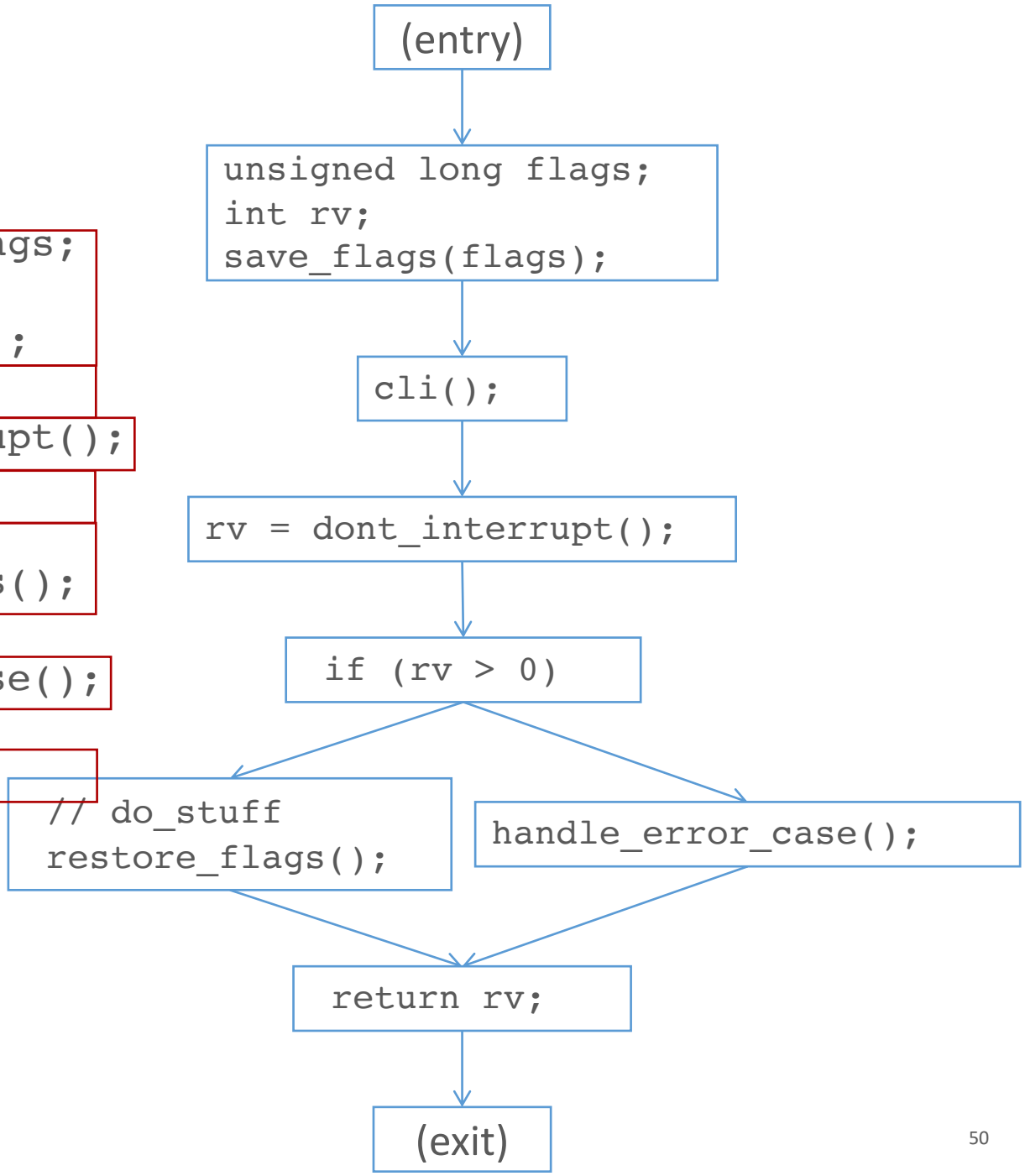
```
1. a = 5 + (2 + 3)
2. if (b > 10) {
3.    a = 0;
4. }
5. return a;
```



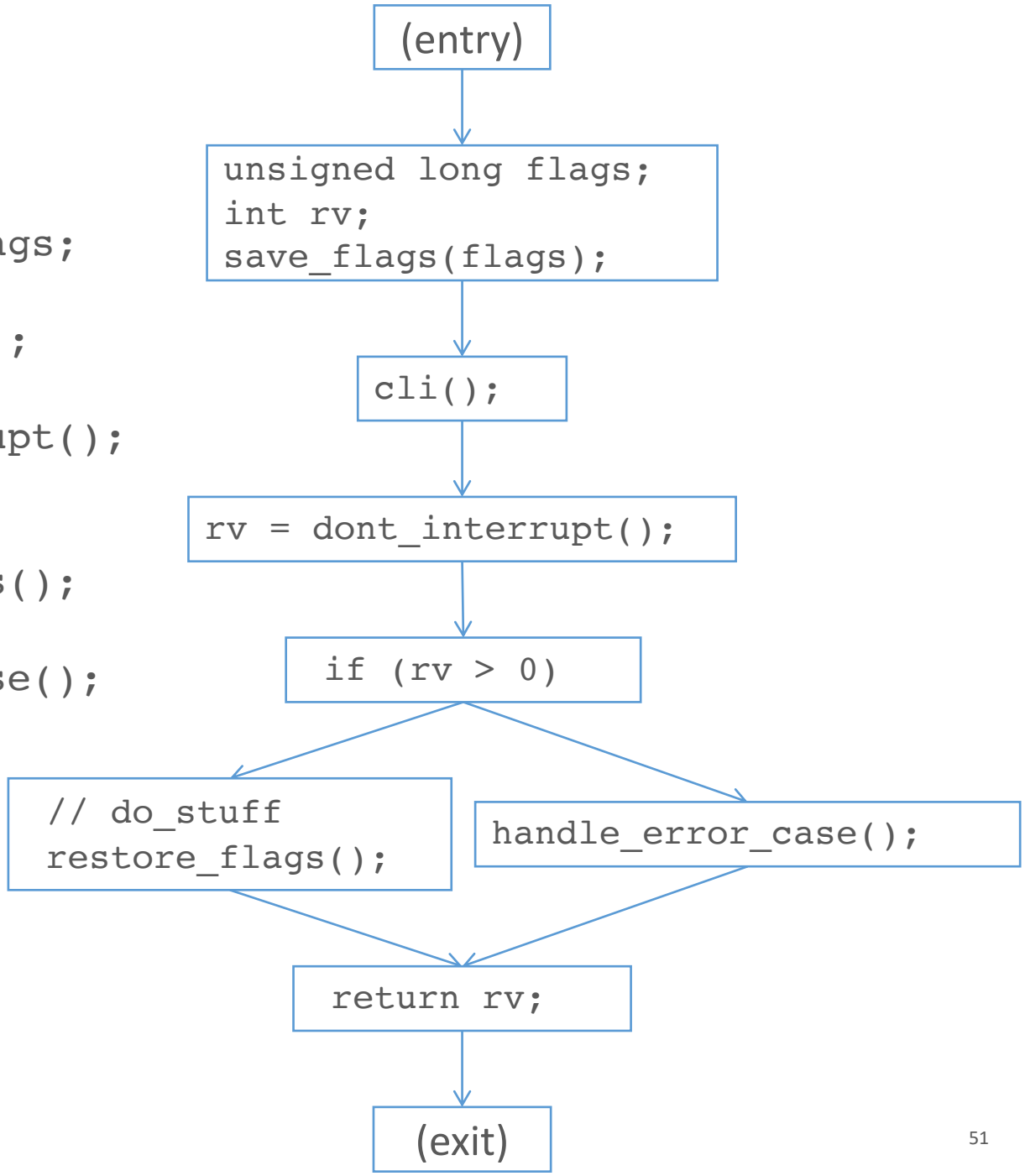
```
public int foo() {
    doStuff();

    return 3;

    doMoreStuff();
    return 4;
}
```



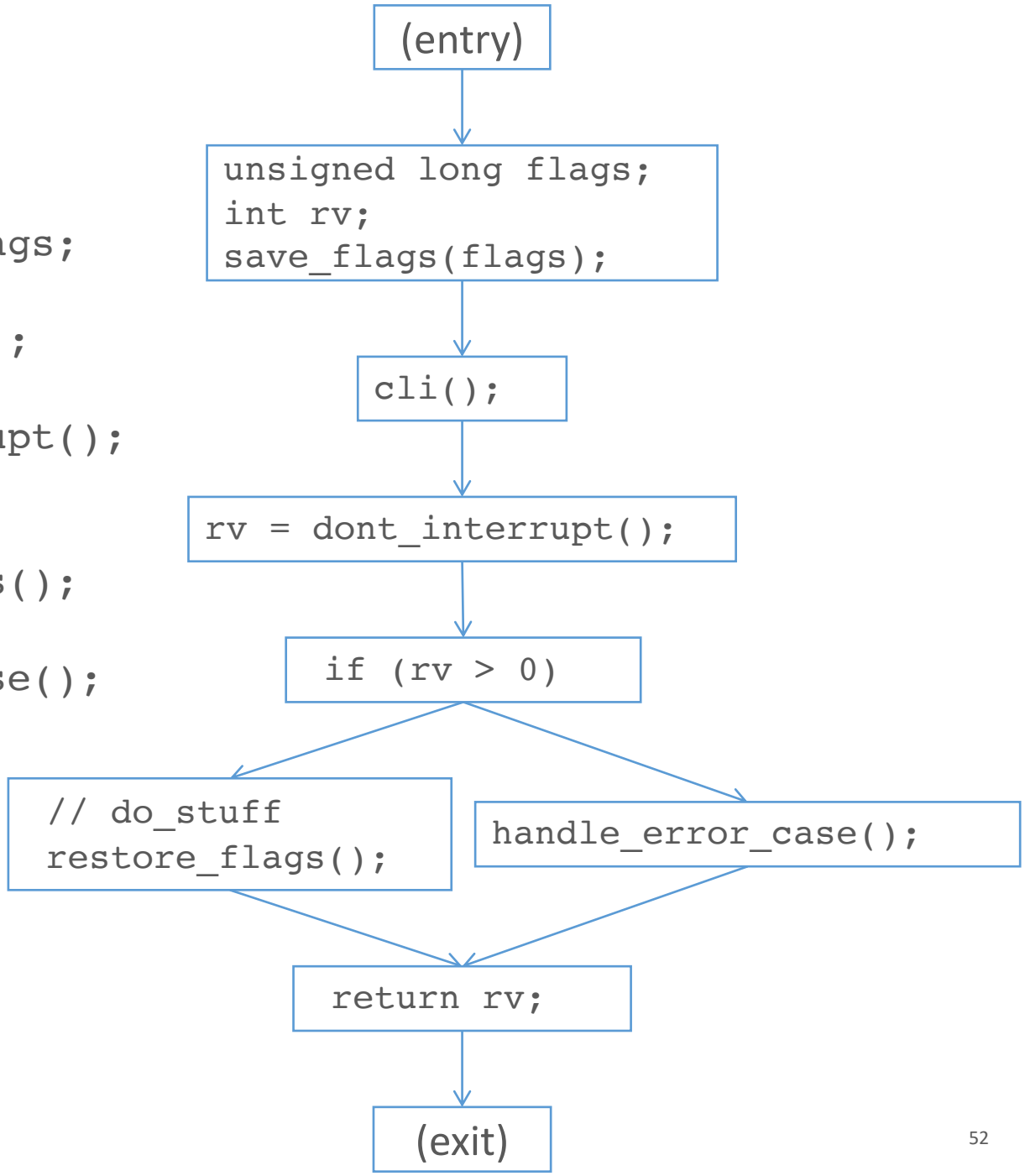
```
1.  int foo() {
2.    unsigned long flags;
3.    int rv;
4.    save_flags(flags);
5.    cli();
6.    rv = dont_interrupt();
7.    if (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.   } else {
11.    handle_error_case();
12.   }
13.   return rv;
14. }
```



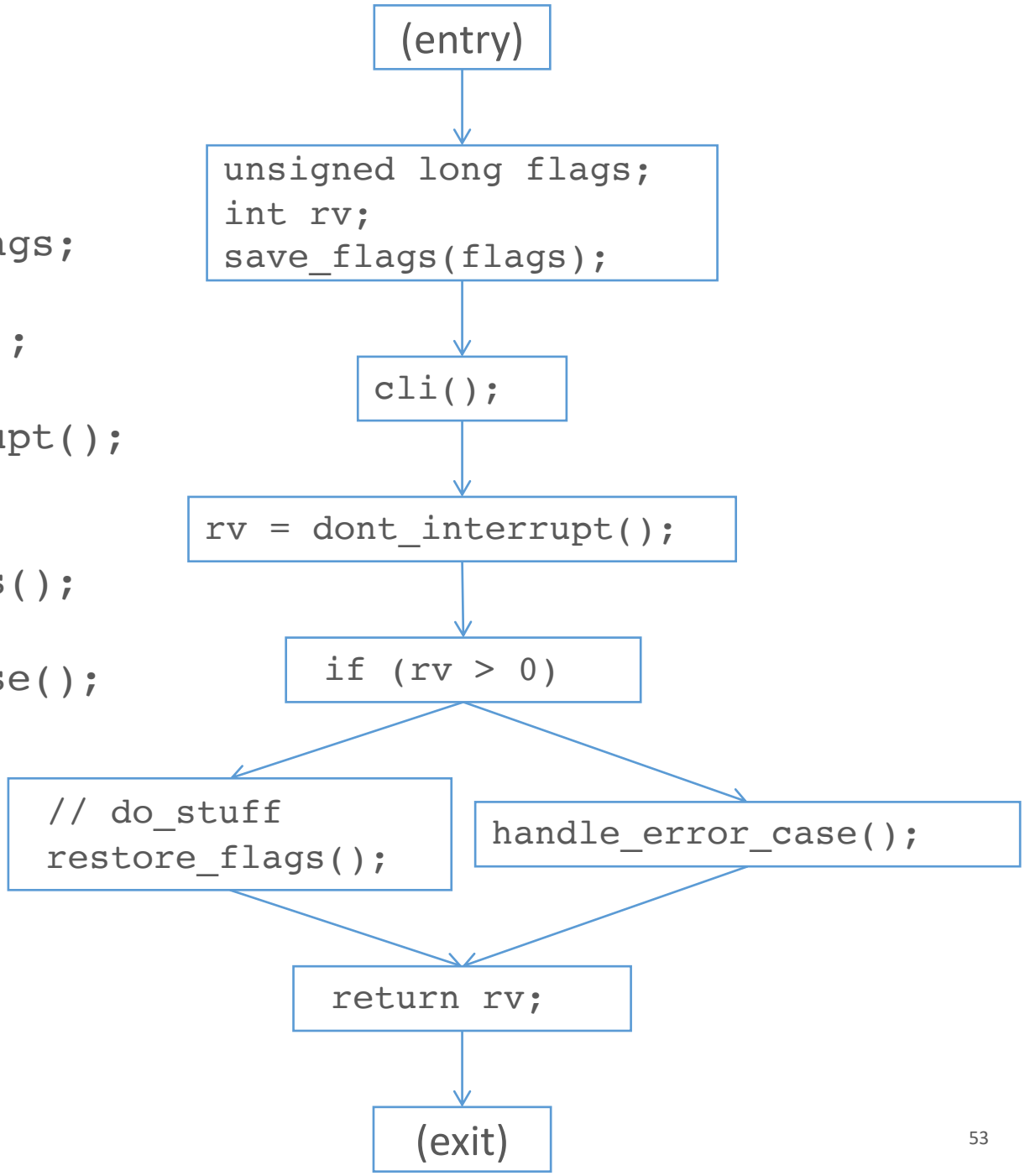
```
1.  int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     if (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.    } else {
11.     handle_error_case();
12.    }
13.    return rv;
14. }
```



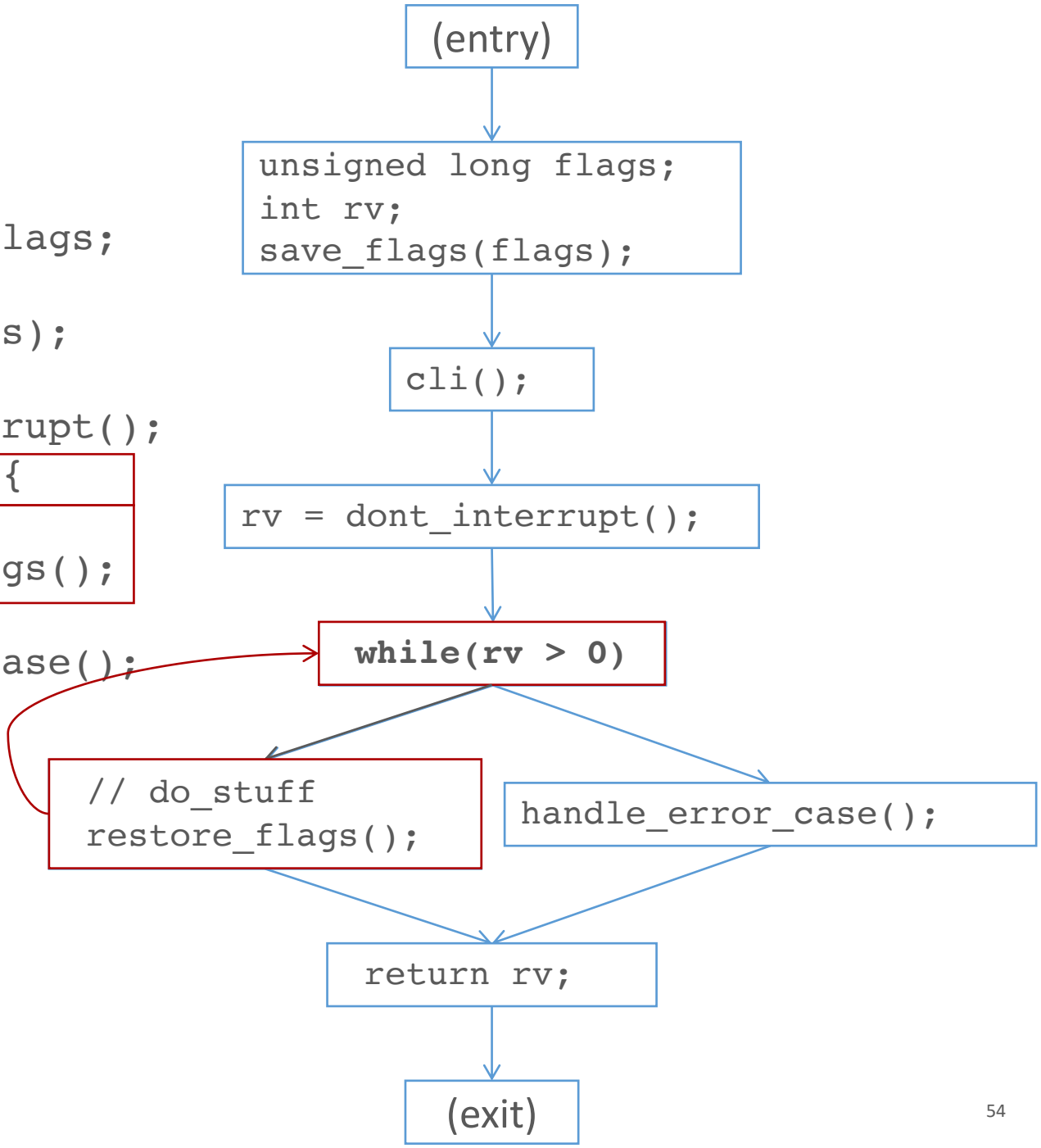
```
1.  int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     while (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.    } else {
11.     handle_error_case();
12.    }
13.    return rv;
14. }
```



```
1.  int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     while (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.    } else {
11.     handle_error_case();
12.    }
13.    return rv;
14. }
```



```
1.  int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     while (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.    }
11.     handle_error_case();
12.
13.    return rv;
14. }
```



The Bad News: Rice's Theorem

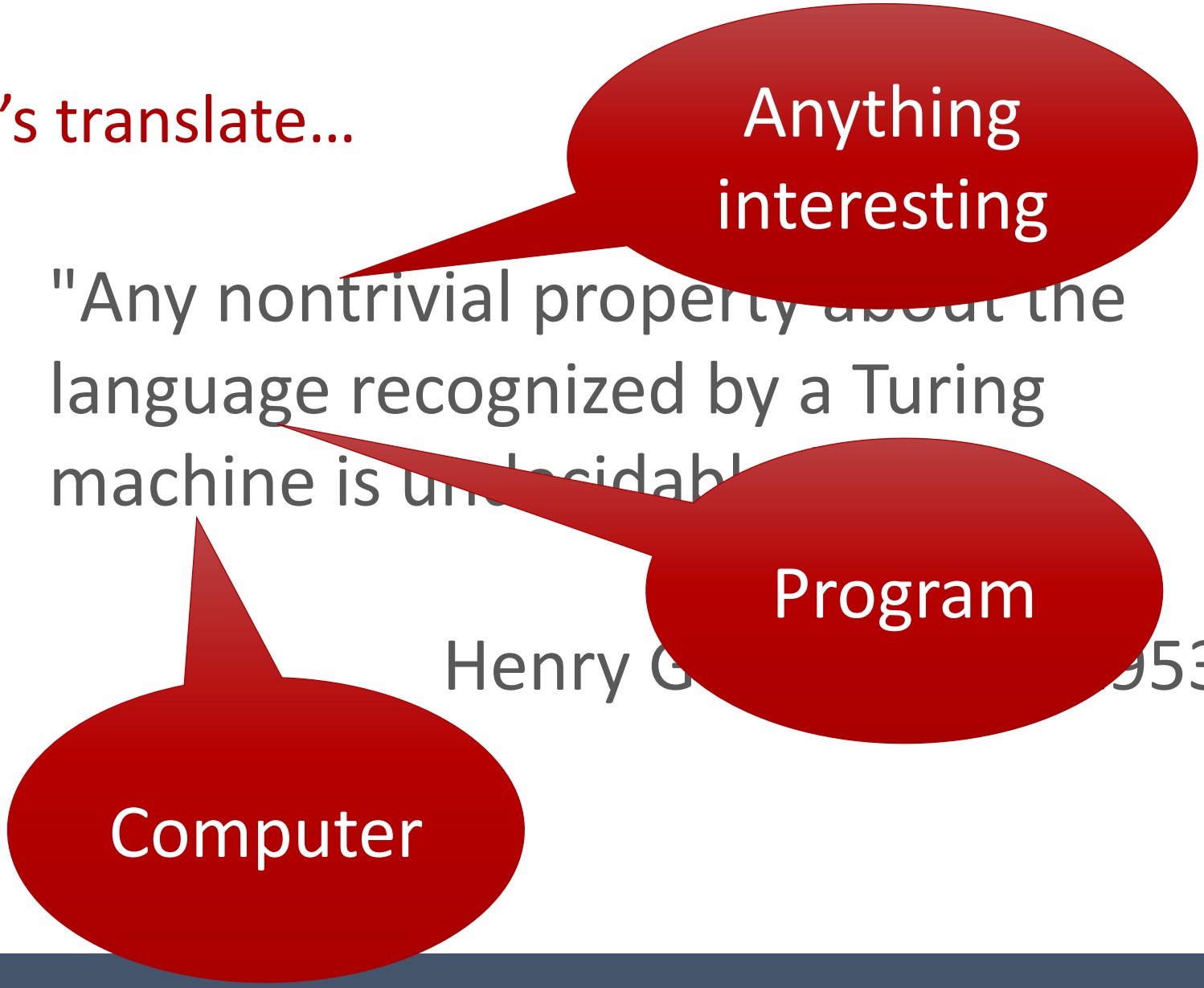
"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

WHAT DOES THAT MEAN, AND ALSO, WHY?

Let's translate…



Why? Infinite loops.

- I have a program, and it takes input.
- That program is written in a reasonable programming language, so it has loops.
- One way a program with loops can go horrifically awry is that it can loop infinitely.
- It's often hard to tell the difference between a program that just takes a long time to execute, and a program that's stuck in an infinite loop.

Computability theory says…

- **Halting problem:** the problem of determining whether a given program will halt/terminate on a given input.
- A *general* algorithm that solves this problem is impossible.
  - More specifically: it's undecidable (it's possible to get a *yes* answer, but not a *no* answer).
  - (sometimes you can use heuristics, but solving it generally for all programs is still out.)
- The proof here is very elegant. But trust me: this problem is extremely impossible.

OK, so?

- If you could always statically tell if any program had a non-trivial property (never dereferences null, always releases all file handles, etc, etc), you could also generally solve the halting problem.
- …but the halting problem is *definitely* impossible.
- So: no static analysis is perfect. They will always have false positives or false negatives (or both).
- *All tools make tradeoffs.*

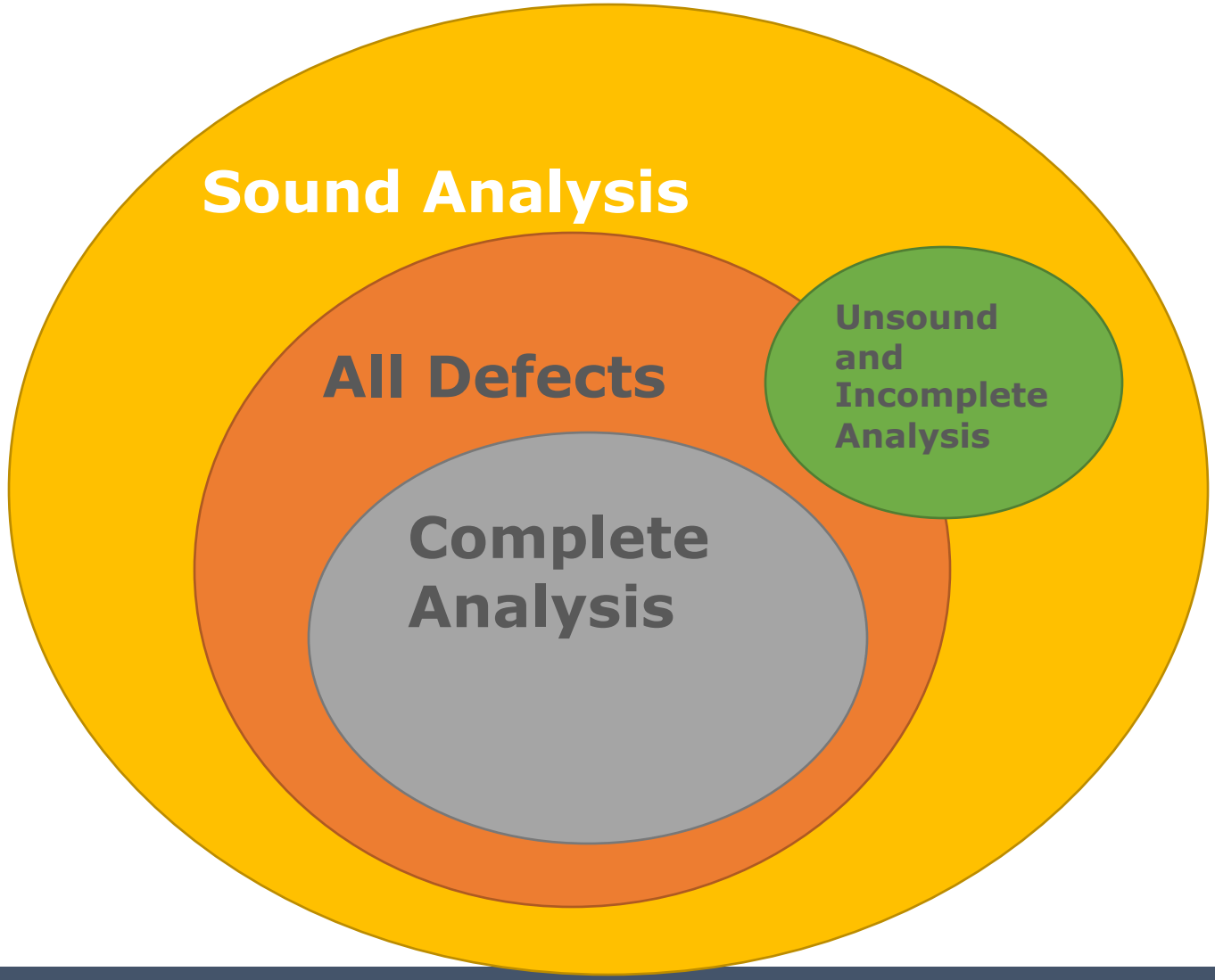
	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

Sound Analysis:

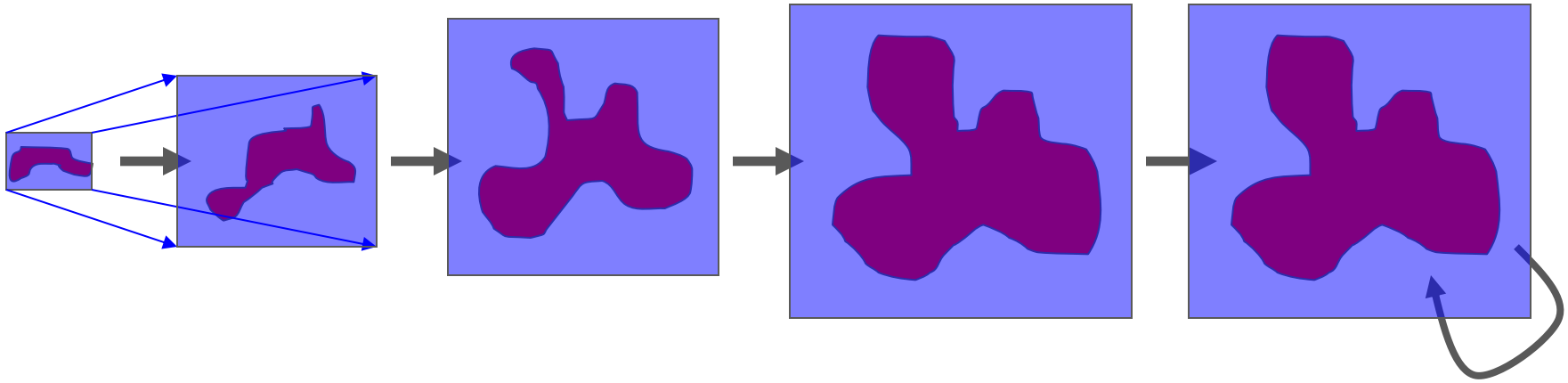
reports all defects
-> no false negatives
typically overapproximated

Complete Analysis:

every reported defect is an actual defect
-> no false positives
typically underapproximated

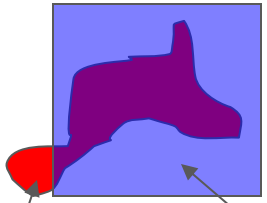

Sound Analysis
All Defects
Unsound and Incomplete Analysis
Complete Analysis

Soundness and precision



Program state covered in actual execution

Program state covered by abstract execution with analysis



Sound vs. Heuristic Analysis

- Heuristic Analysis
  - FindBugs, checkstyle, …
  - Follow rules, approximate, avoid some checks to reduce false positives
  - May report false positives and false negatives
- Sound Static Analysis
  - Type checking, Not-Null, … (specific fault classes)
  - Sound abstraction, precise analysis to reduce false positives

Upshot: analysis as approximation

- Analysis must approximate in practice
  - False positives: may report errors where there are really none
  - False negatives: may not report errors that really exist
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct code
    - which is feasible to check (analysis terminates quickly),
    - covers most correct code in practice (low false positives),
    - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A sound tool has no false negatives
    - Never misses an error in a category that it checks