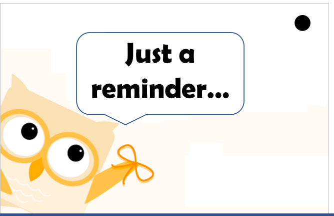
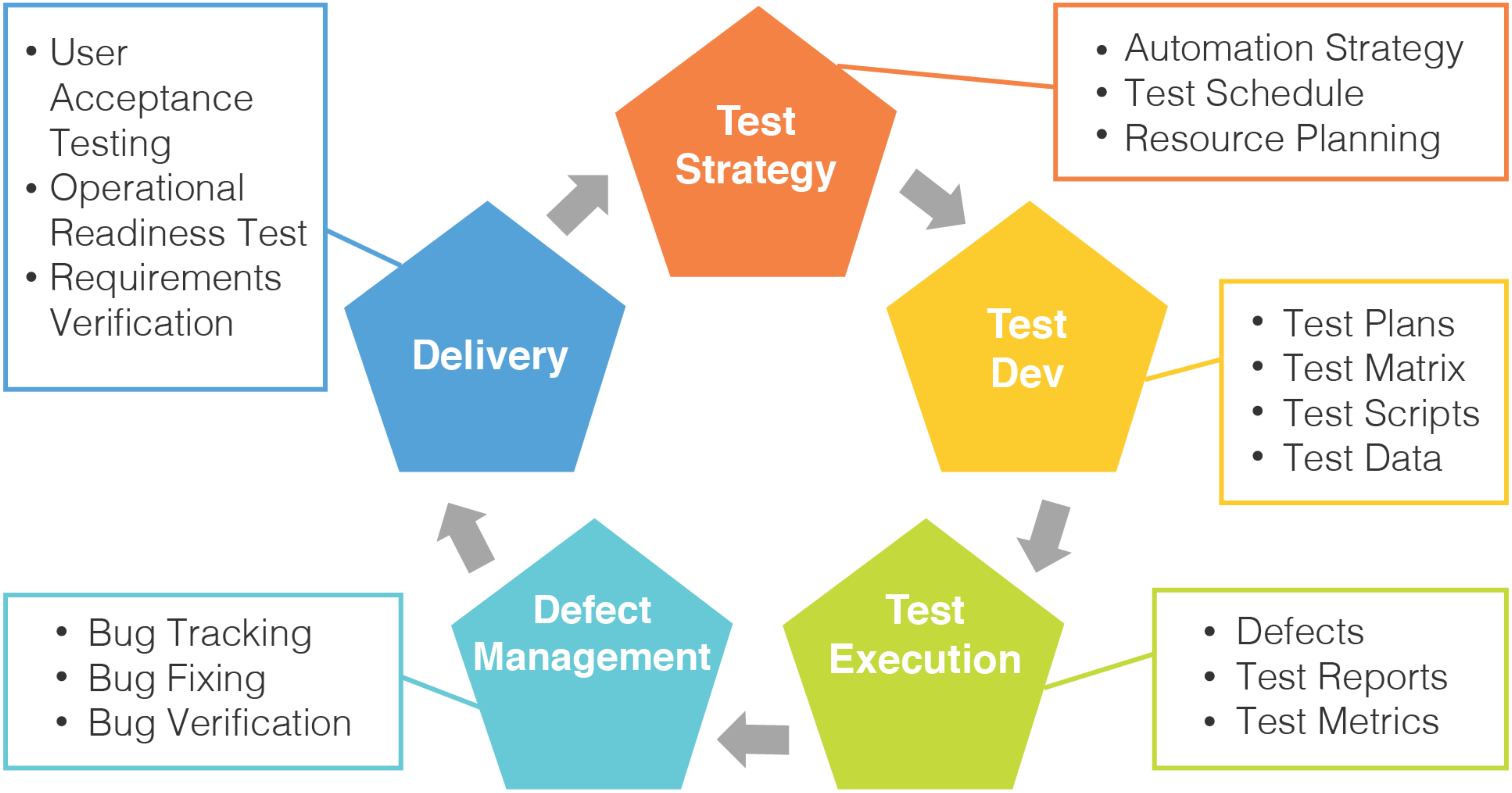
What is testing?

- *Direct execution of code on test data in a controlled environment*
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
- 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




• User Acceptance Testing
• Operational Readiness Test
• Requirements Verification
Test Strategy
• Automation Strategy
• Test Schedule
• Resource Planning
Delivery
Test Dev
• Test Plans
• Test Matrix
• Test Scripts
• Test Data
Defect Management
Test Execution
• Bug Tracking
• Bug Fixing
• Bug Verification
• Defects
• Test Reports
• Test Metrics

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.

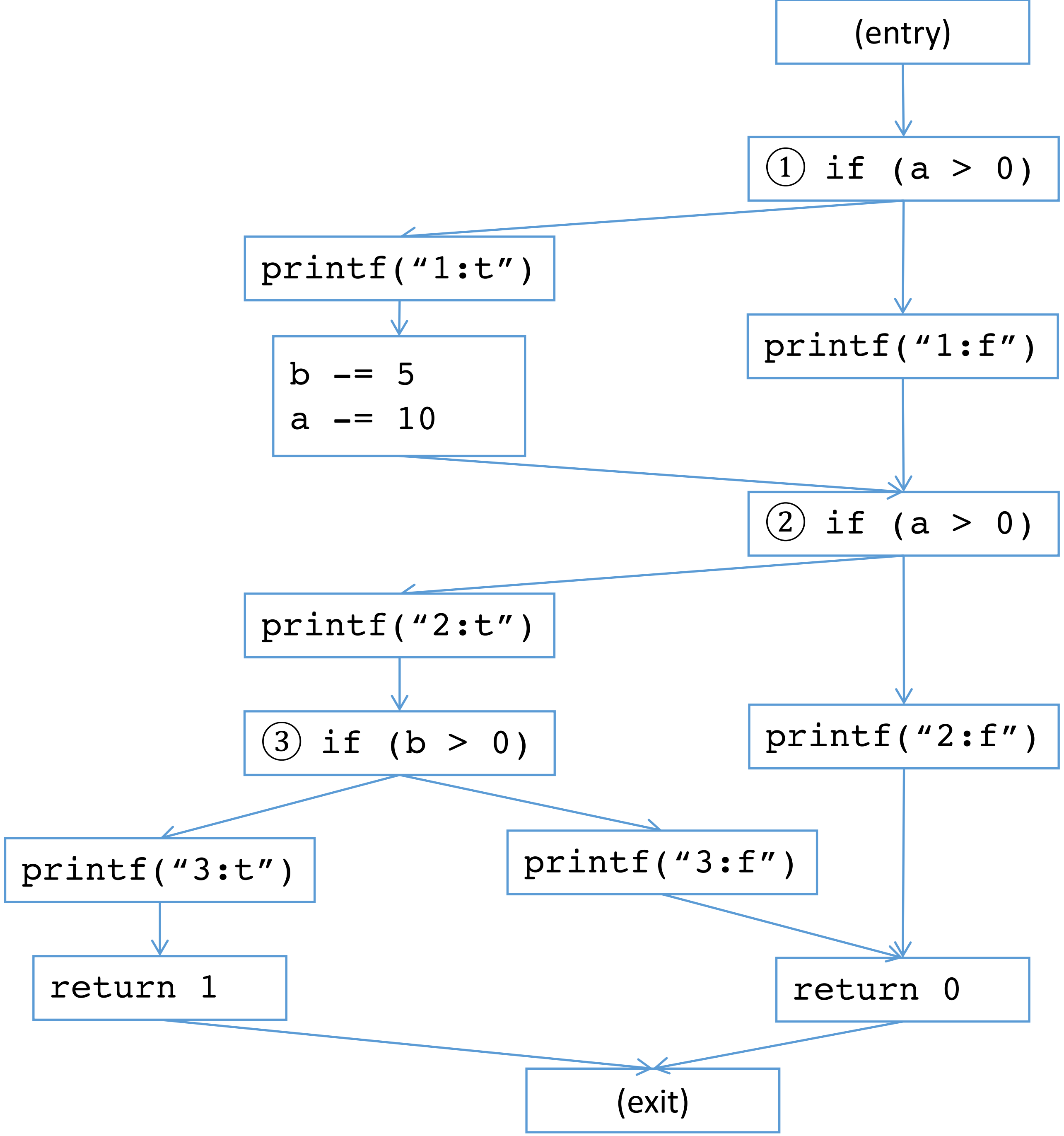
Common dynamic analyses

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

Instrumentation: a simple example

- How might tools that compute test suite coverage work?
- 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?

```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;
}
```



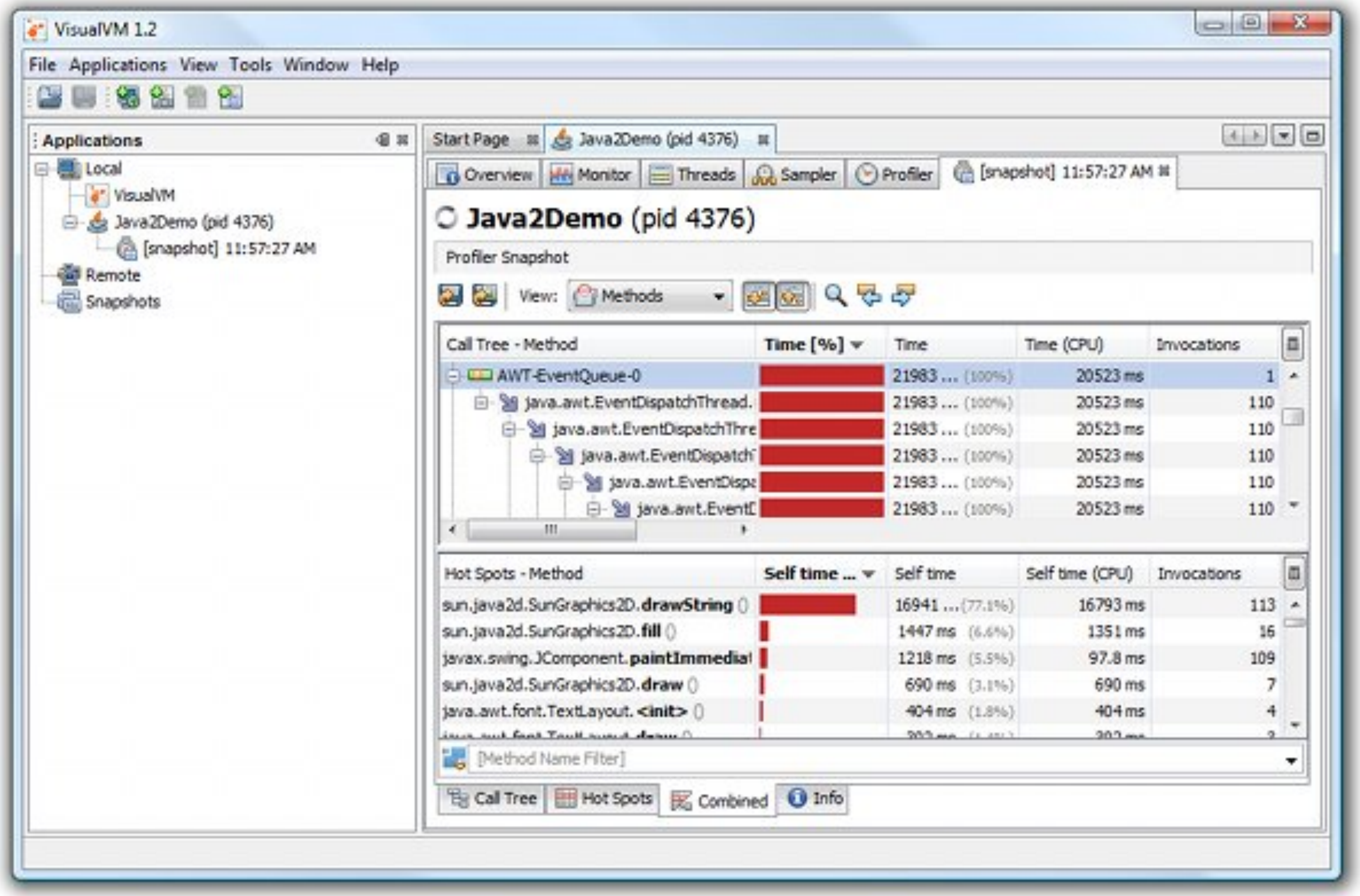
```
int foobar(a,b) {
  if (a > 0) {
    printf(“1:t ”);
    b -= 5;
    a -= 10;
  } else {
    printf(“1:f ”);
  }
  if(a > 0) {
    printf(“2:t ”);
    if (b > 0) {
      printf(“3:t ”);
      return 1;
    } else {
      printf(“3:f ”);
    }
  } else {
    printf(“2:f ”);
  }
  return 0;
}
```

- 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.

Profiling

- Finding bottlenecks in execution time and memory



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

Limitation: Dynamic analysis

- Cost

  Performance overhead for recording

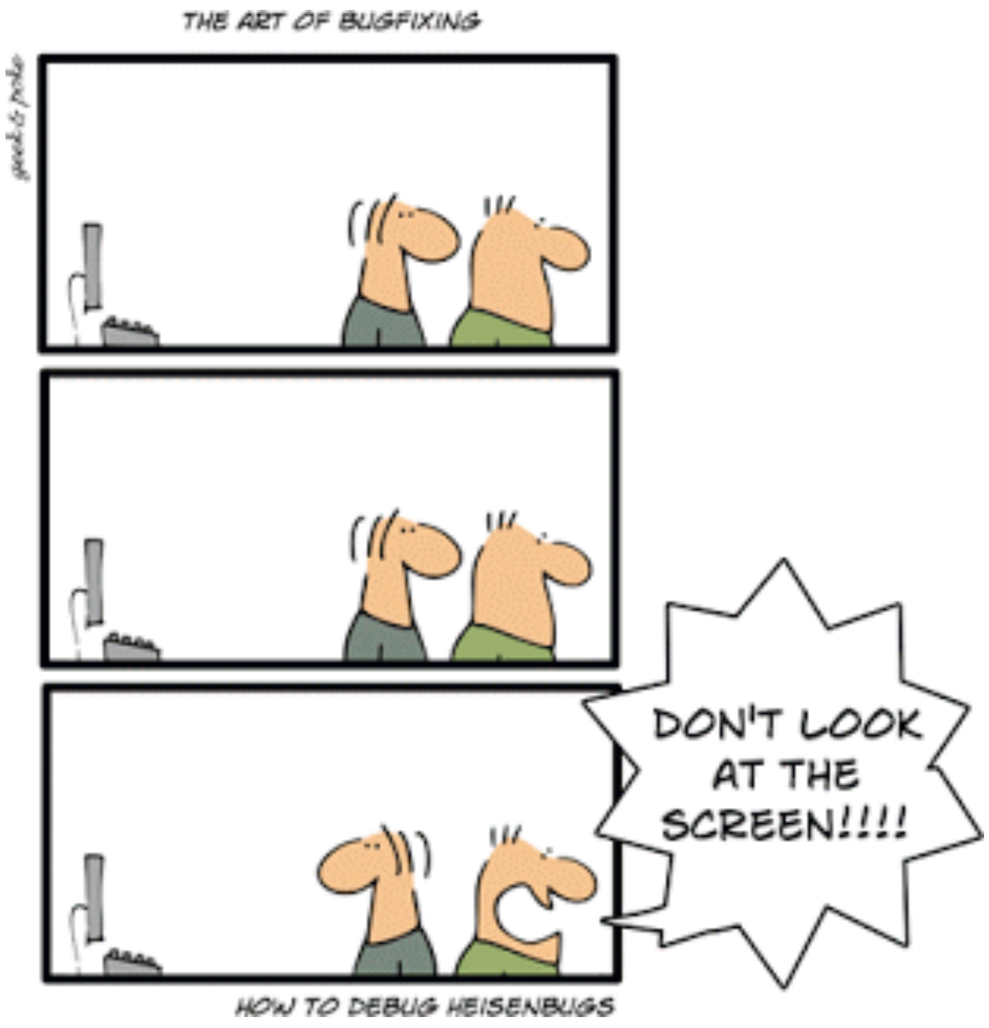
  - Acceptable for use in testing?
  - Acceptable for use in production?

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).

Heisenbugs

- Heisenbugs occur because common attempts to debug a program, such as inserting output statements or running it with a debugger, usually have the side-effect of altering the behavior of the program in subtle ways, such as changing the memory addresses of variables and the timing of its execution.



https://www.testing-whiz.com/blog/heisenbug-elusive-bug

Heisenbuggy behavior

- Instrumentation and monitoring can change the behavior of a program.
  - e.g., slowdown, memory overhead.
- **Important question 1:** can/should you deploy it live?
  - Or possibly just deploy for debugging something specific?
- **Important question 2:** *Will the monitoring meaningfully change the program behavior with respect to the property you care about?*

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

Principle techniques

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

What is 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.
- Check if a **particular property** holds over the entire state space:
  - Liveness: “something good eventually happens.”
  - Safety: “this bad thing can’t ever happen.”

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

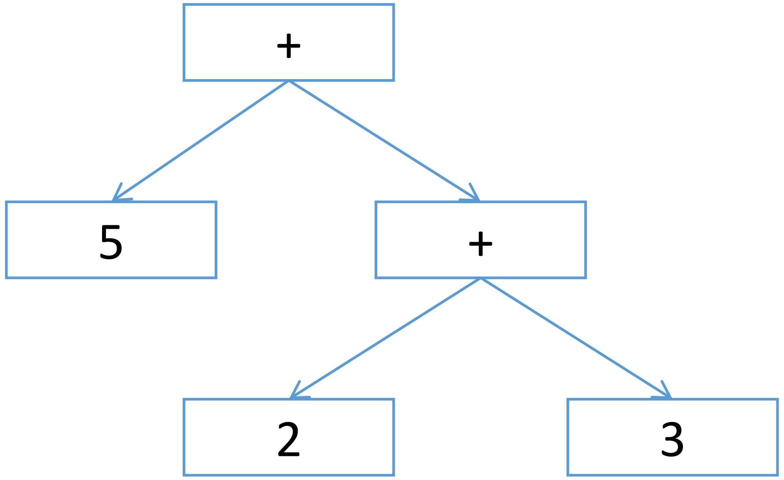
```java
public void foo() {
    int a = computeSomething();

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

Abstraction: abstract syntax tree

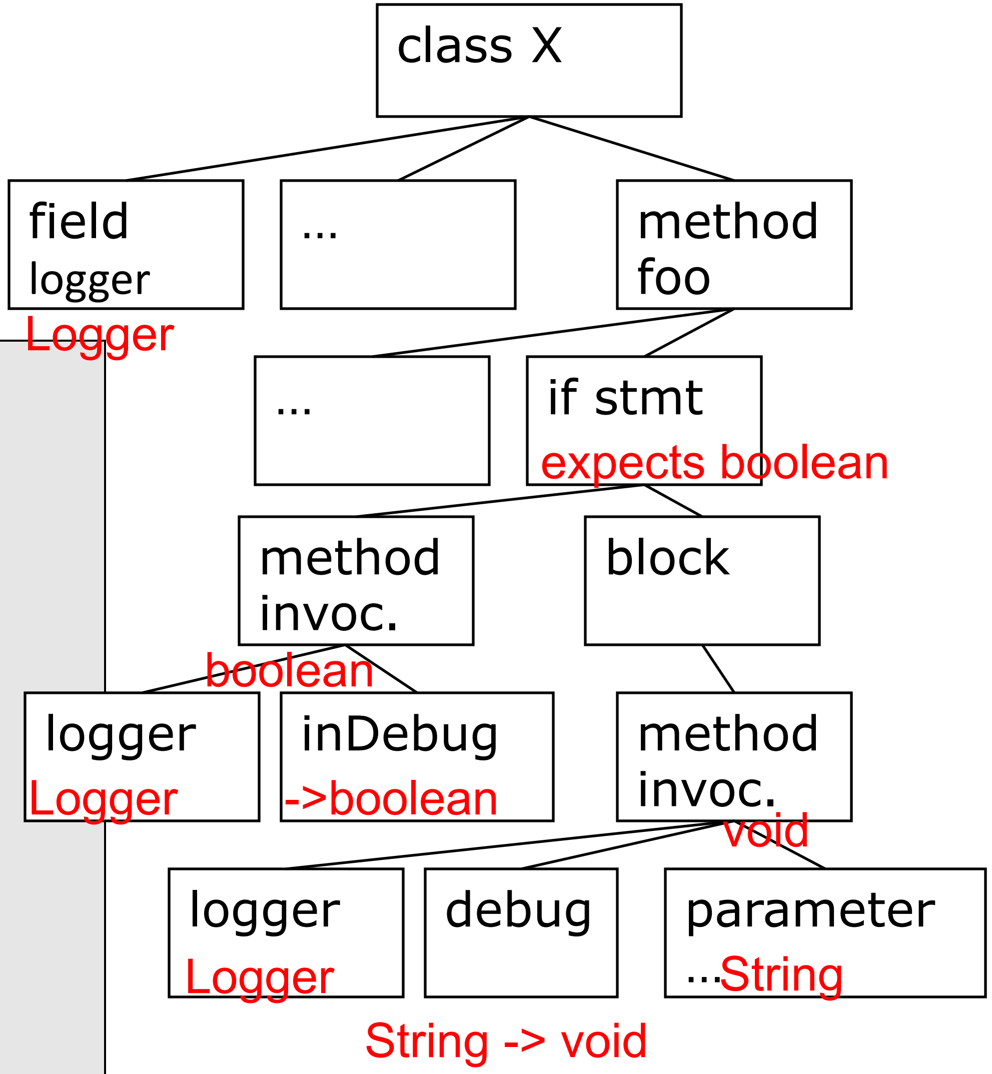
- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn't represent every detail (like parentheses); these can be inferred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

- Example: 5 + (2 + 3)

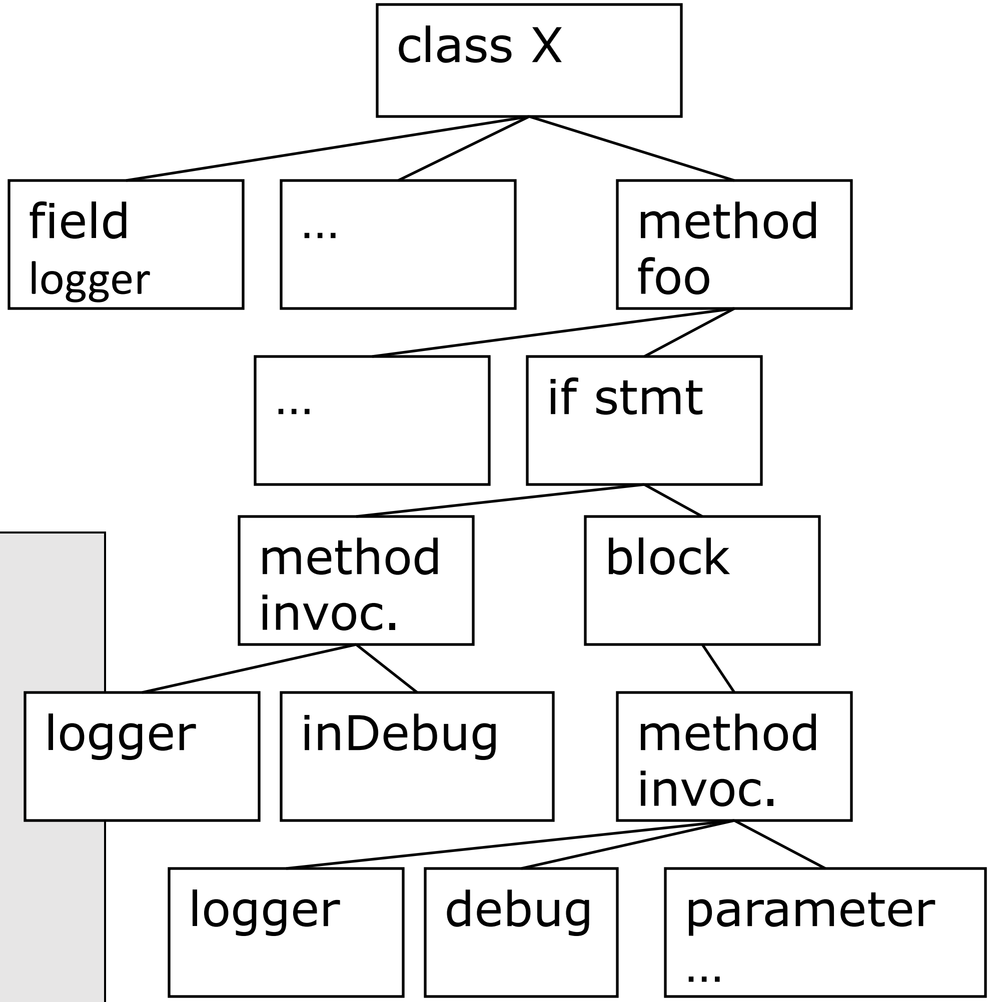


Type checking

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



Structural Analysis



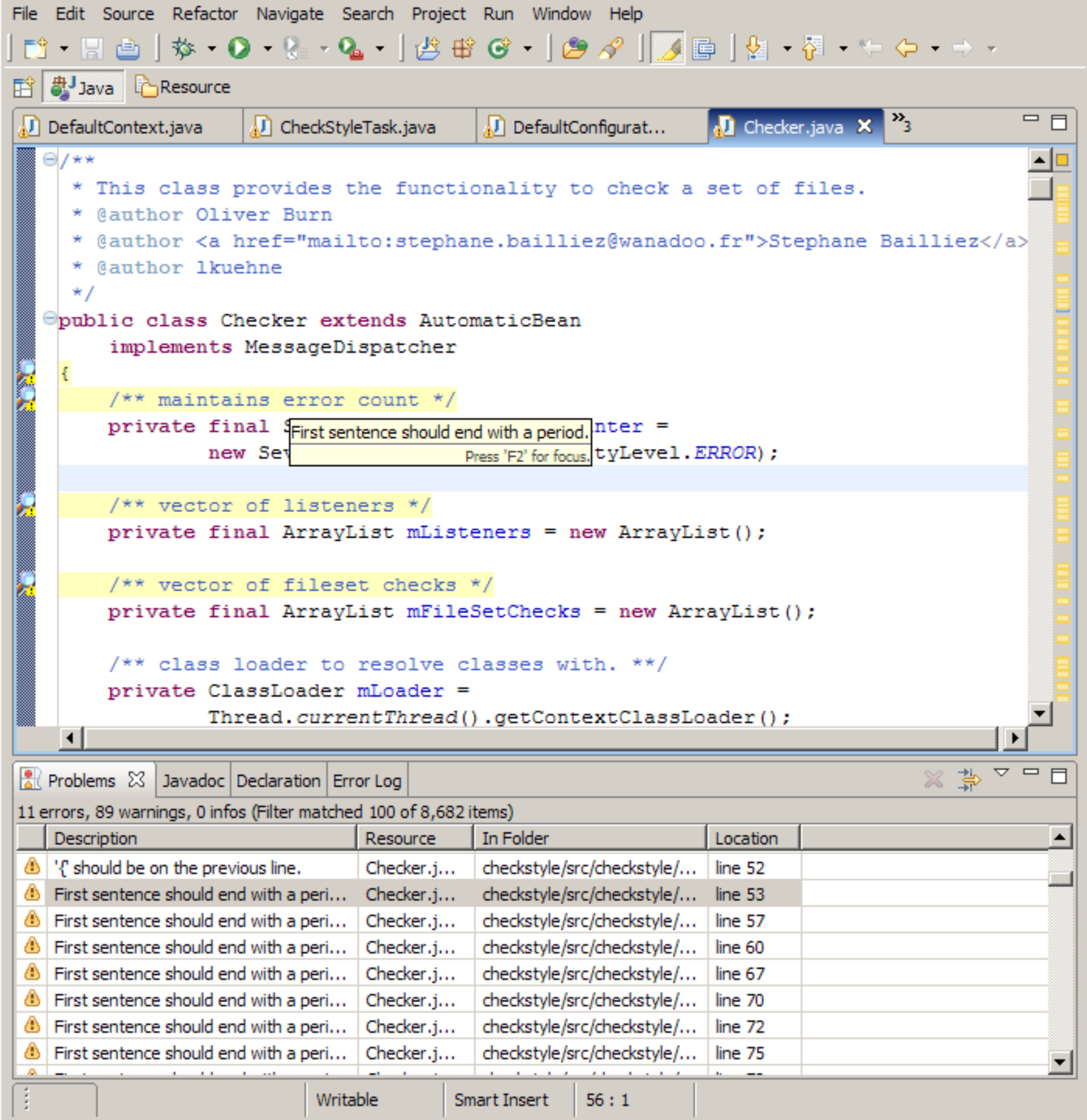

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " +
conn + "connections.");
    }
  }
}
```

Summary: Syntactic/Structural Analyses

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

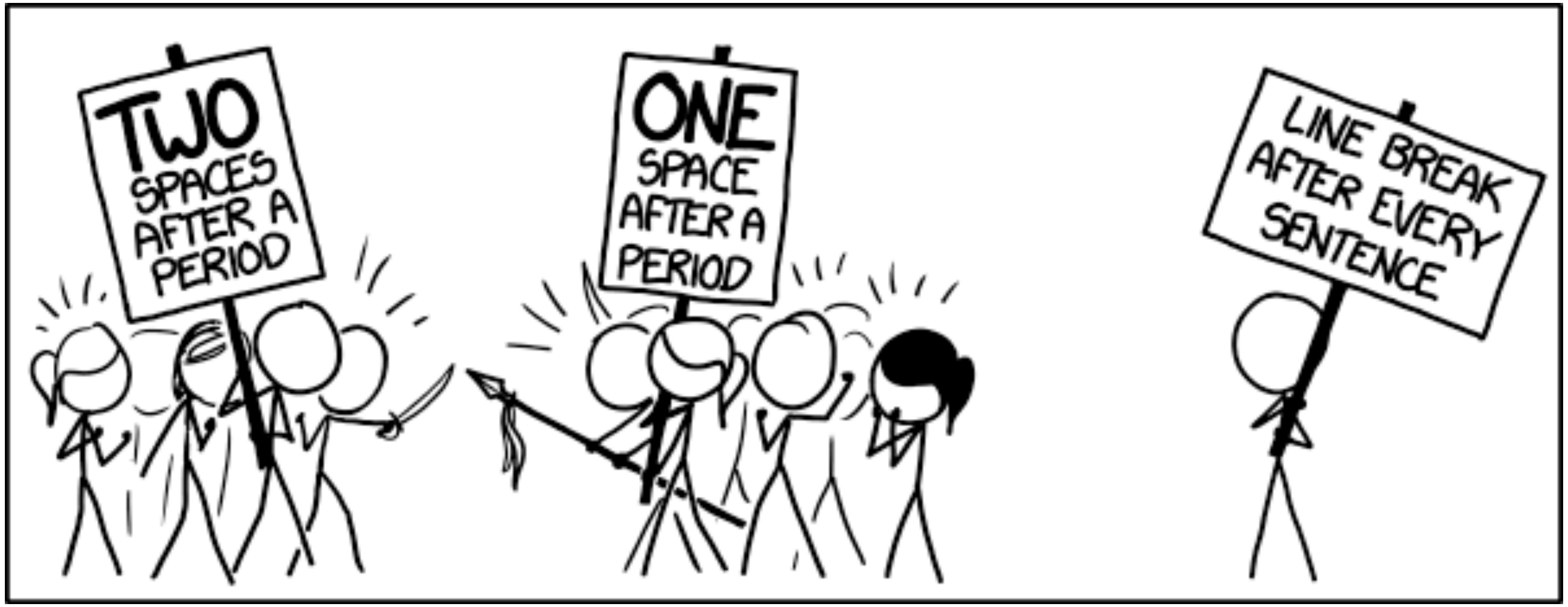
Tools

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



Linter

- *is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.*



https://xkcd.com/1285/

https://www.pylint.org/

Pylint

Star your Python code!

Read the doc | Install it | Contribute | Get support

Features

Coding Standard

- checking line-code's length,
- checking if variable names are well-formed according to your coding standard
- checking if imported modules are used

Python's PEP8 style guide

Error detection

- checking if declared interfaces are truly implemented
- checking if modules are imported
- and much more (see the complete check list)

Full list of codes (wiki)

Refactoring help

Pylint detects duplicated code

About Refactoring (on wikipedia)

Fully customizable

Modify your `pylintrc` to customize which errors or conventions are important to you. The big

Editor integration

Run it in emacs , vim (pylint.vim, syntastic), eclipse, etc

IDE integration

Pylint is integrated into various IDEs:

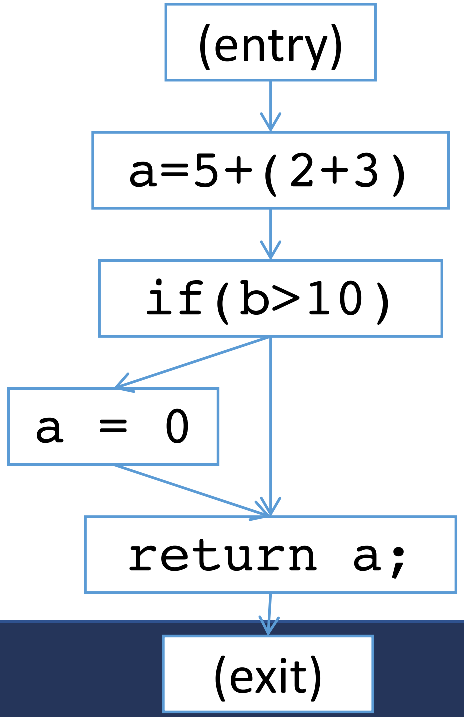
Control/Dataflow analysis

- **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;
}
return a;
```



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

ECE444: Software Engineering
QA 3: Quality Assurance Process, Case Studies

Shurui Zhou

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

QA Process Considerations

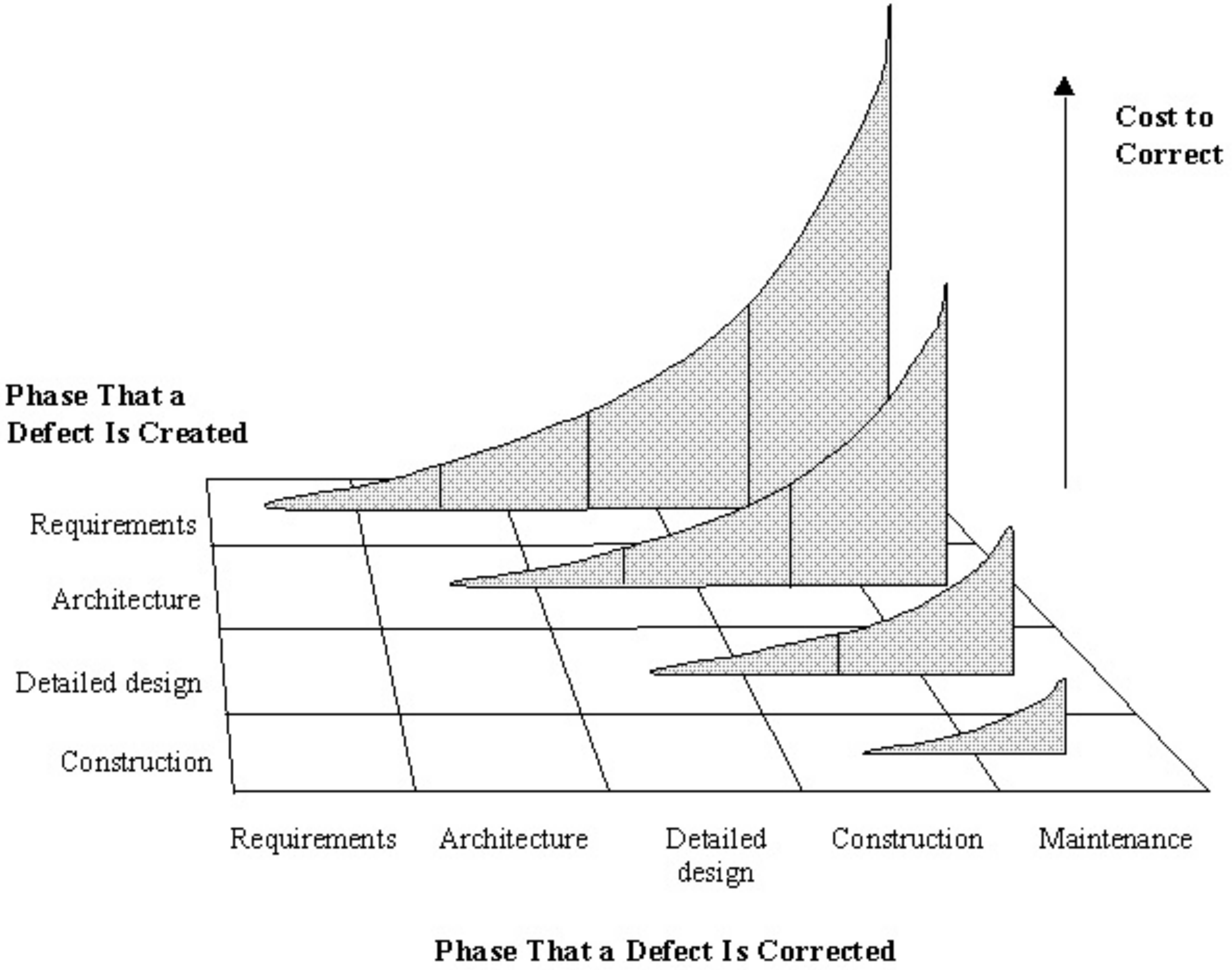
- We covered several QA techniques:
  - Formal verification
  - Unit testing, test driven development
  - Various forms of advanced testing for quality attributes (GUI testing, fuzz testing, …)
  - Static analysis
  - Dynamic analysis
  - Formal inspections and other forms of code reviews
- But: When to use? Which techniques? How much? How to introduce? Automation? How to establish a quality culture? How to ensure compliance? Social issues? What about external components?

Learning Goals

- Understand process aspects of QA
- Describe the tradeoffs of QA techniques
- Select an appropriate QA technique for a given project and quality attribute
- Decide the when and how much of QA
- Overview of concepts how to enforce QA techniques in a process
- Select when and how to integrate tools and policies into the process: daily builds, continuous integration, test automation, static analysis, issue tracking, …
- Understand human and social challenges of adopting QA techniques
- Understand how process and tool improvement can solve the dilemma between features and quality

QA Process

How to get developers to
[write tests|use static analysis|appreciate testers]

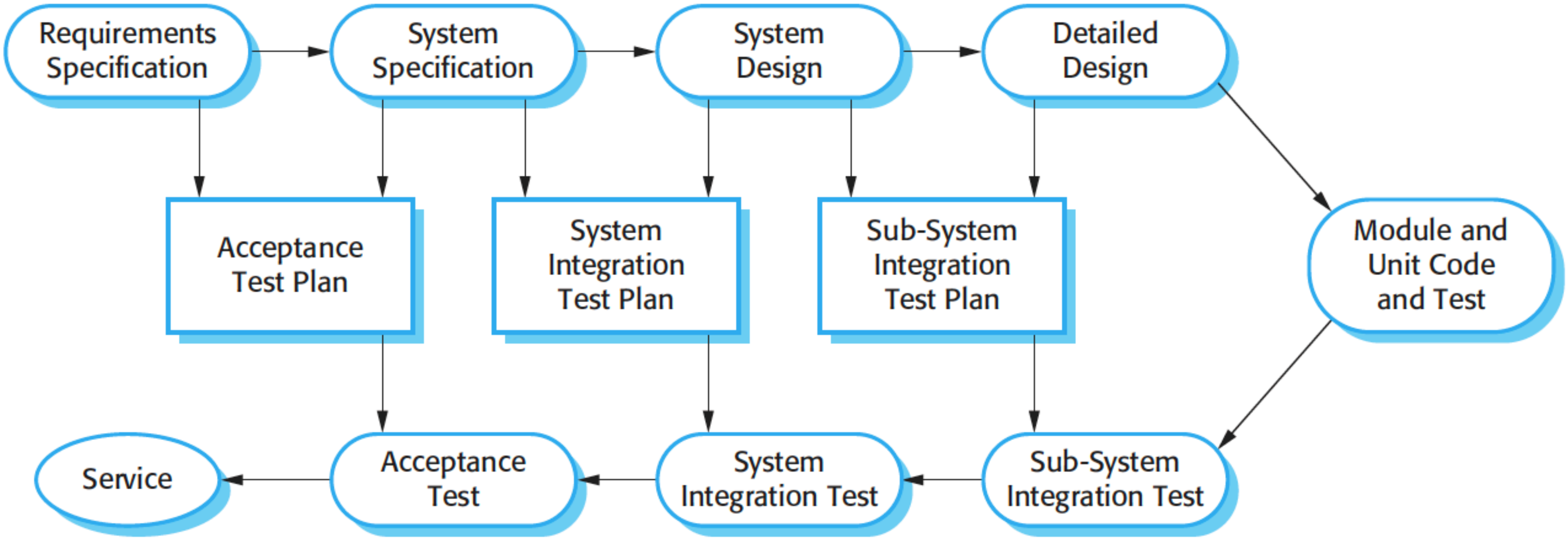

Cost to Correct
Phase That a Defect Is Created
Requirements
Architecture
Detailed design
Construction
Requirements
Architecture
Detailed design
Construction
Maintenance
Phase That a Defect Is Corrected

Qualities and Risks

- What qualities are required? (requirements engineering)
- What risks are expected?

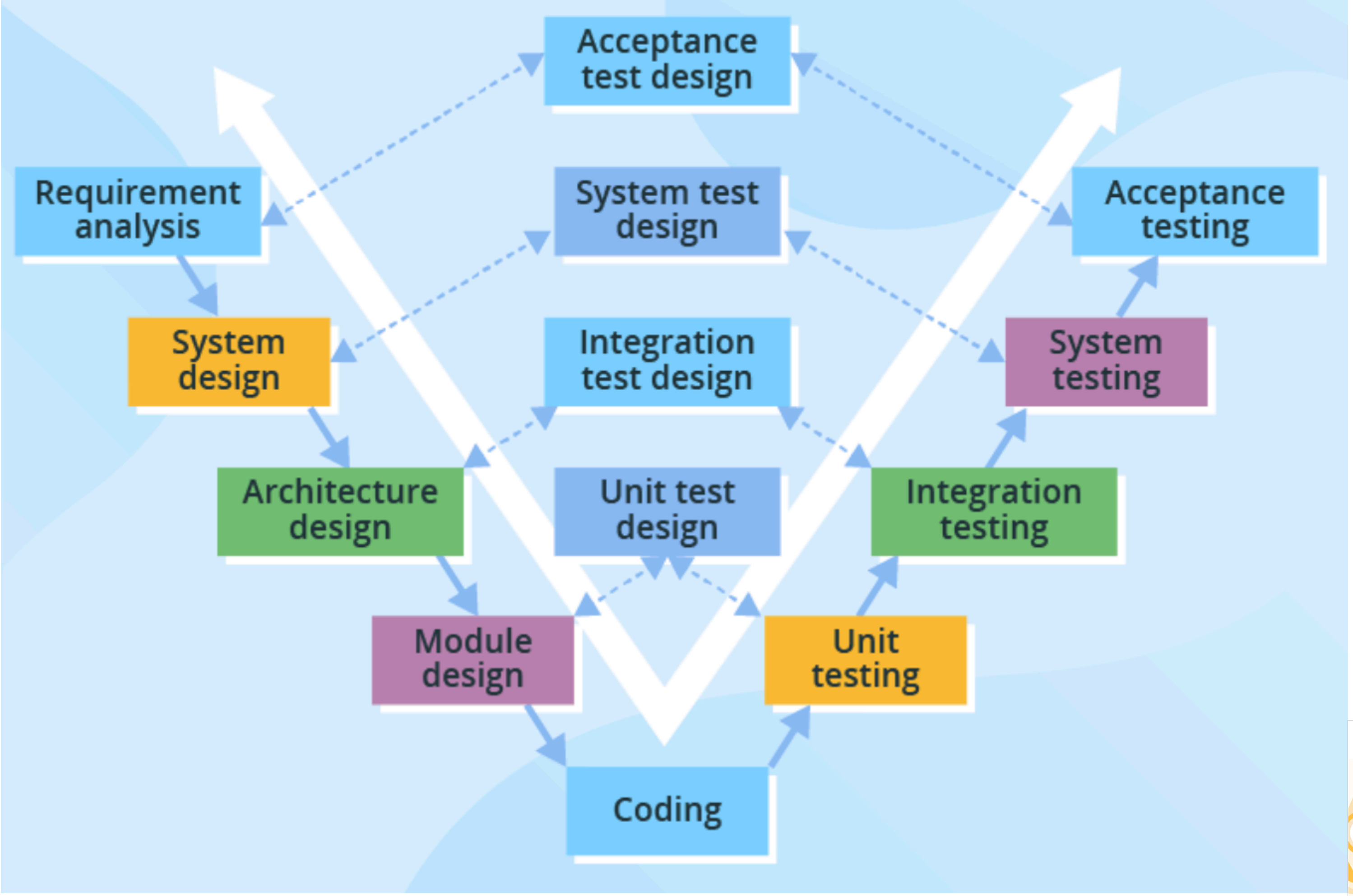
- Align QA strategy based on qualities and risks

Example:
Test plans linking development and testing



Sommerville. Software Engineering. Ed. 8, Ch 22

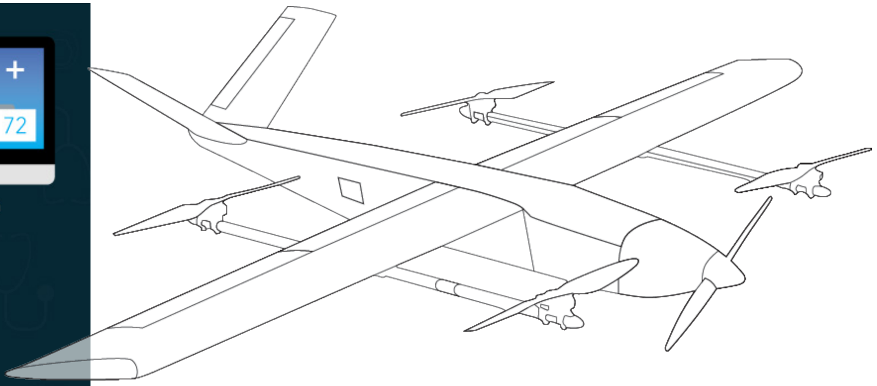
V-Model

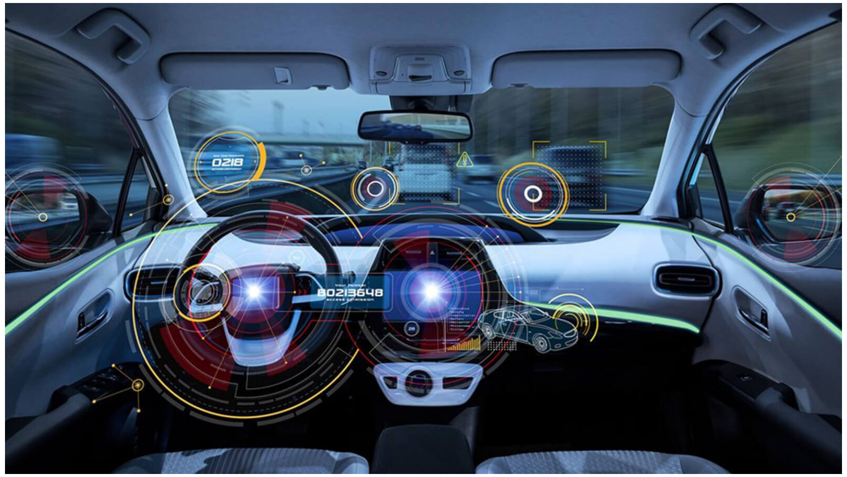


Expensive and time-consuming

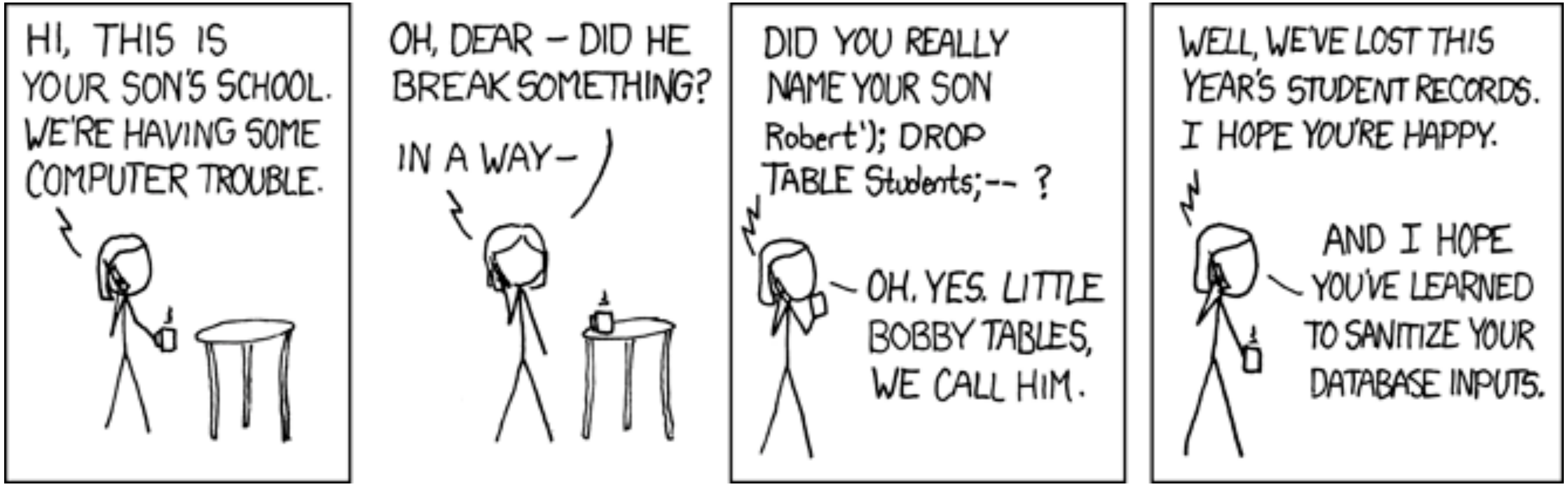
Use cases

Projects where failures and downtimes are unacceptable (e.g., medical software, aviation fleet management software).





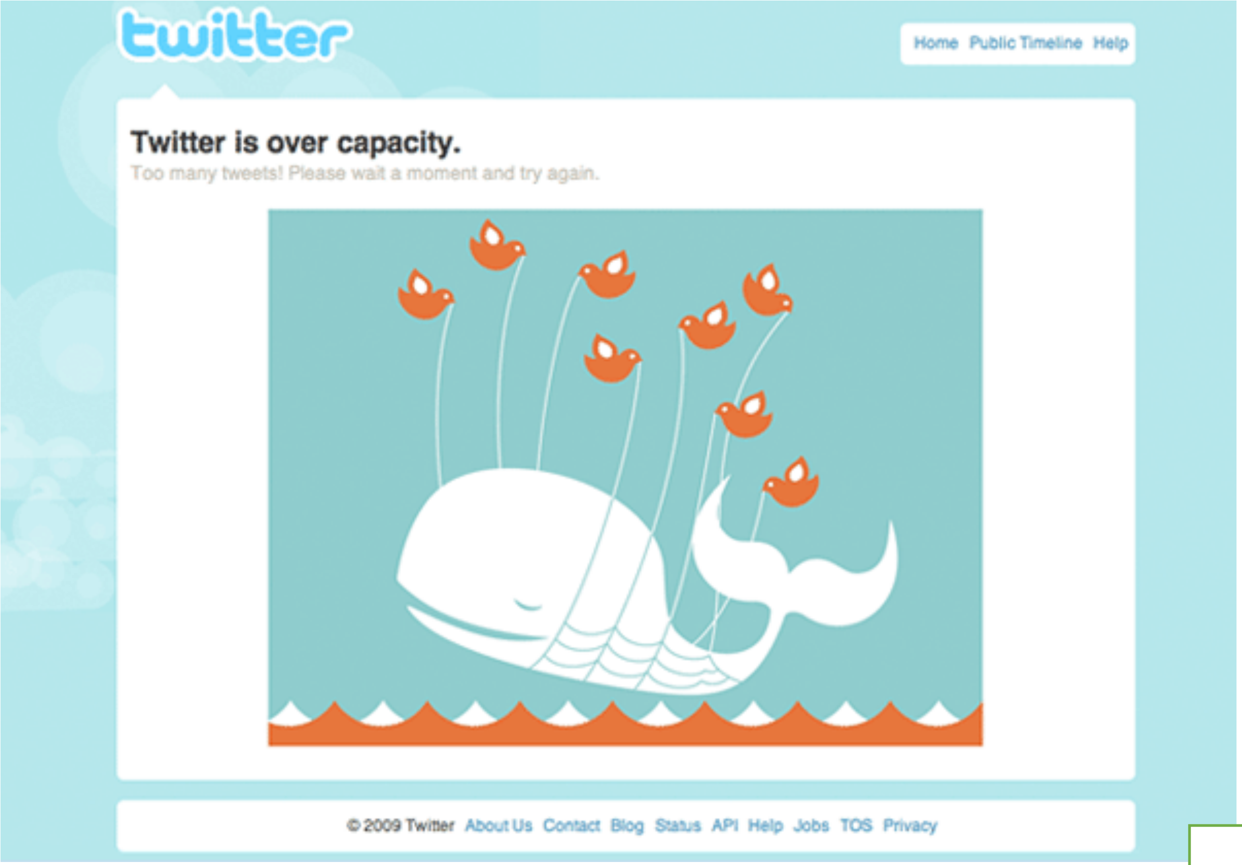
Example: SQL Injection Attacks



Which QA strategy is suitable?

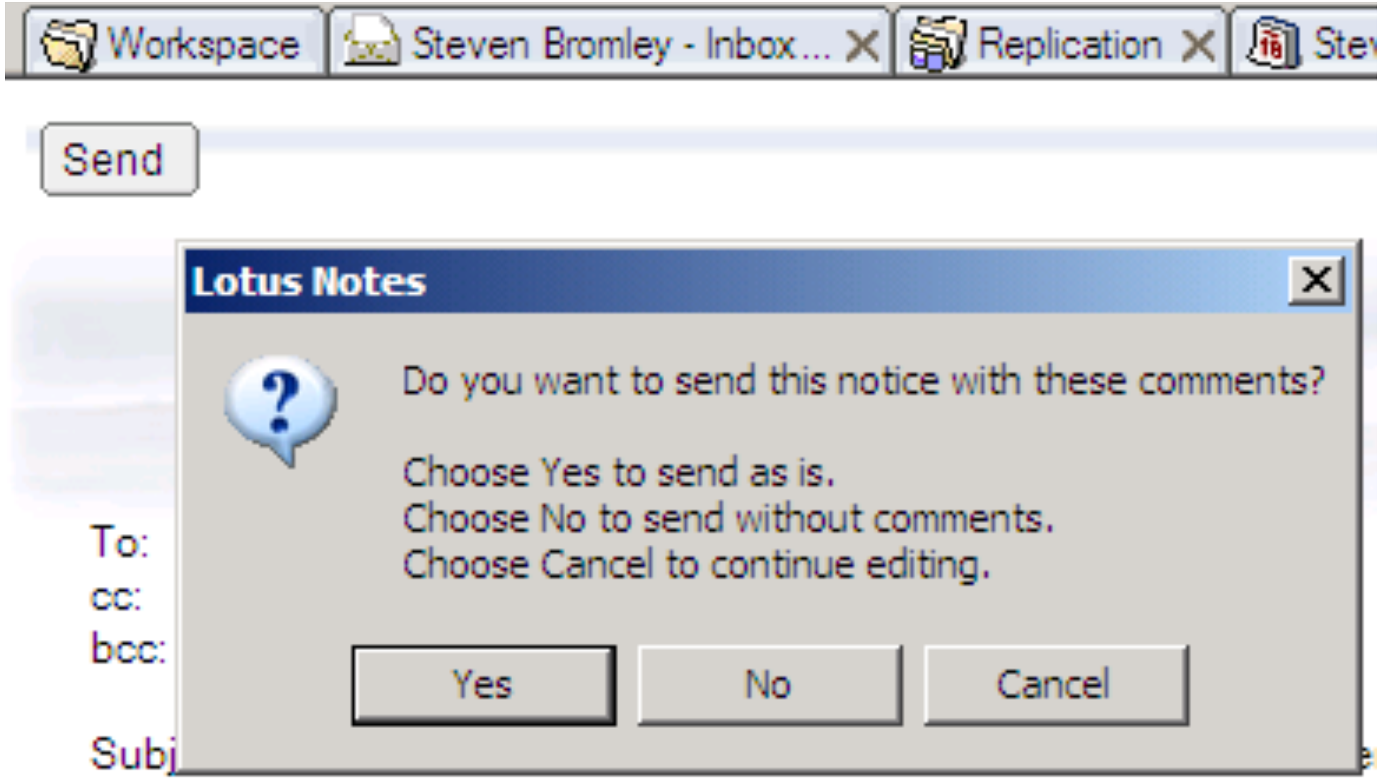
http://xkcd.com/327/

Example: Scalability



Which QA strategy is suitable?

Example: Usability



Which QA strategy is suitable?

QA Tradeoffs

- Understand limitations of QA approaches
  - e.g. testing vs static analysis,
    formal verification vs inspection, …
- Mix and match techniques
- Different techniques for different qualities

Case Study: QA at Microsoft

A problem has been detected and windows has been shut down to prevent damage
to your computer.

THREAD_NOT_MUTEX_OWNER

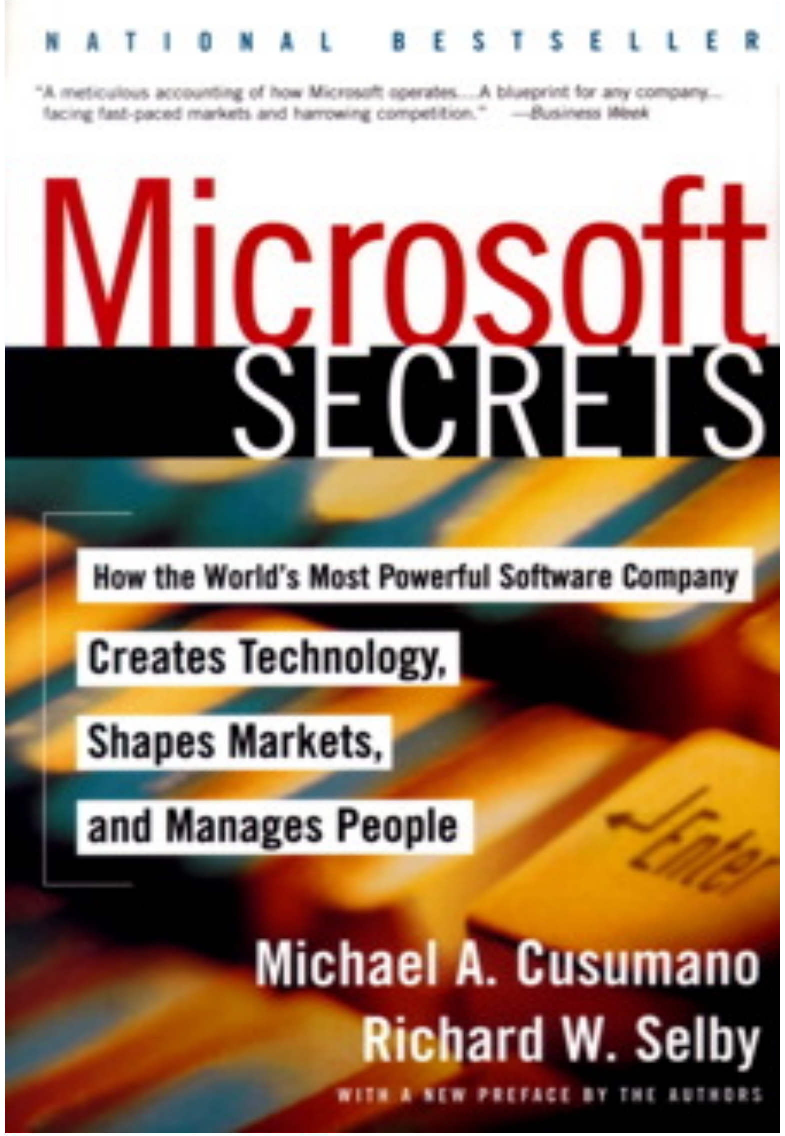
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000011 (0x00234234,0x00005345,0x05345345,0xFFFFFFFF)


NATIONAL BESTSELLER
"A meticulous accounting of how Microsoft operates....A blueprint for any company... facing fast-paced markets and harrowing competition." —Business Week
Microsoft
SECRETS
How the World's Most Powerful Software Company
Creates Technology,
Shapes Markets,
and Manages People
Enter
Michael A. Cusumano
Richard W. Selby
WITH A NEW PREFACE BY THE AUTHORS

Throughout the case studies, look for nontechnical challenges and how they were addressed (social issues, process issues, …)

Microsoft's Culture

- Hiring the best developers
  - “Microsoft can achieve with a few hundred top-notch developers for what IBM would need thousands”
- Giving them freedom
- Teams for products largely independent
- Relatively short development cycles
  - Version updates (eg. Excel 3->4) 1-2 month
  - New products 1-4 years
  - Driven by release date
- Little upfront specification, flexible for change and cutting features

Early Days (1984): Separate testing from development

- after complaints over bugs from hardware manufacturers (eg. wrong computations in BASIC)
- customers complained about products
- IBM insisted that Microsoft improves process for development and quality control
- Serious data-destroying bug forced Microsoft to ship update of Multiplan to 20000 users at 10$ cost each
- Resistance from developers and some management (incl. Balmer): “developers could test their own products, assisted on occasion by high school students, secretaries, and some outside contractors”
- Hired outside testers
- Avoided bureaucracy of formal inspections, signoff between stages, or time logging
- Separate testing group; automated tests; code reviews for new people and critical components

Early Days (1986): Testing groups

- “Developers got lazy”, relied on test team for QA
- “Infinite defects” - Testers find defects faster than developers can fix them
- Late and large integrations (“big bang”) - long testing periods, delayed releases
- Mac Word 3 desaster: 8 month late, hundreds of bugs, including crashing and data destroying bugs; 1M$ for free upgrades
- Pressure on delivering quality grew

1989 Retreat and “Zero defects”

- see memo

Microsoft Memo

To: Application developers and testers

From: Chris Mason

Date: 6/20/89

Subject: Zero-defects code

Cc: Mike Maples, Steve Ballmer, Applications Business Unit managers and department heads

---

On May 12th and 13th, the applications development managers held a retreat with some of their project leads, Mike Maples, and other representatives of Applications and Languages. My discussion group investigated techniques for writing code with no defects. This memo describes the conclusions which we reached. . . . *There are a lot of reasons why our products seem to get buggier and buggier. It's a fact that they're getting more complex, but we haven't changed our methods to respond to that complexity.* . . . The point of enumerating our problems is to realize that our current methods, not our people, cause their own failure. . . . Our scheduling methods and Microsoft's culture encourage doing the minimum work necessary on a feature. When it works well enough to demonstrate, we consider it done, everyone else considers it done, and the feature is checked off the schedule. The inevitable bugs months later are seen as unrelated. . . . When the schedule is jeopardized, we start cutting corners. . . . *The reason that complexity breeds bugs is that we don't understand how the pieces will work together.* This is true for new products as well as for changes to existing products. . . . I mean this literally: your goal should be to have a working, nearly-shippable product every day. . . . Since human beings themselves are not fully debugged yet, there will be bugs in your code no matter what you do. When this happens, you must evaluate the problem and resolve it immediately. . . . Coding is the major way we spend our time. Writing bugs means we're failing in our major activity. *Hundreds of thousands of individuals and companies rely on our products; bugs can cause a lot of lost time and money. We could conceivably put a company out of business with a bug in a spreadsheet, database, or word processor. We have to start taking this more seriously.* [italics added]

Zero-Defect Rules for Excel 4

- All changes must compile and link
- All changes must pass the automated quick tests on Mac and Windows
- Any developer who has more than 10 open bugs assigned must fix them before moving to new features

Testing Buddies

- Development and test teams separate, roughly similar size
- Developers test their own code, run automated tests daily
- Individual testers often assigned to one developer
  - Testing their private releases (branch), giving direct, rapid feedback by email before code is merged

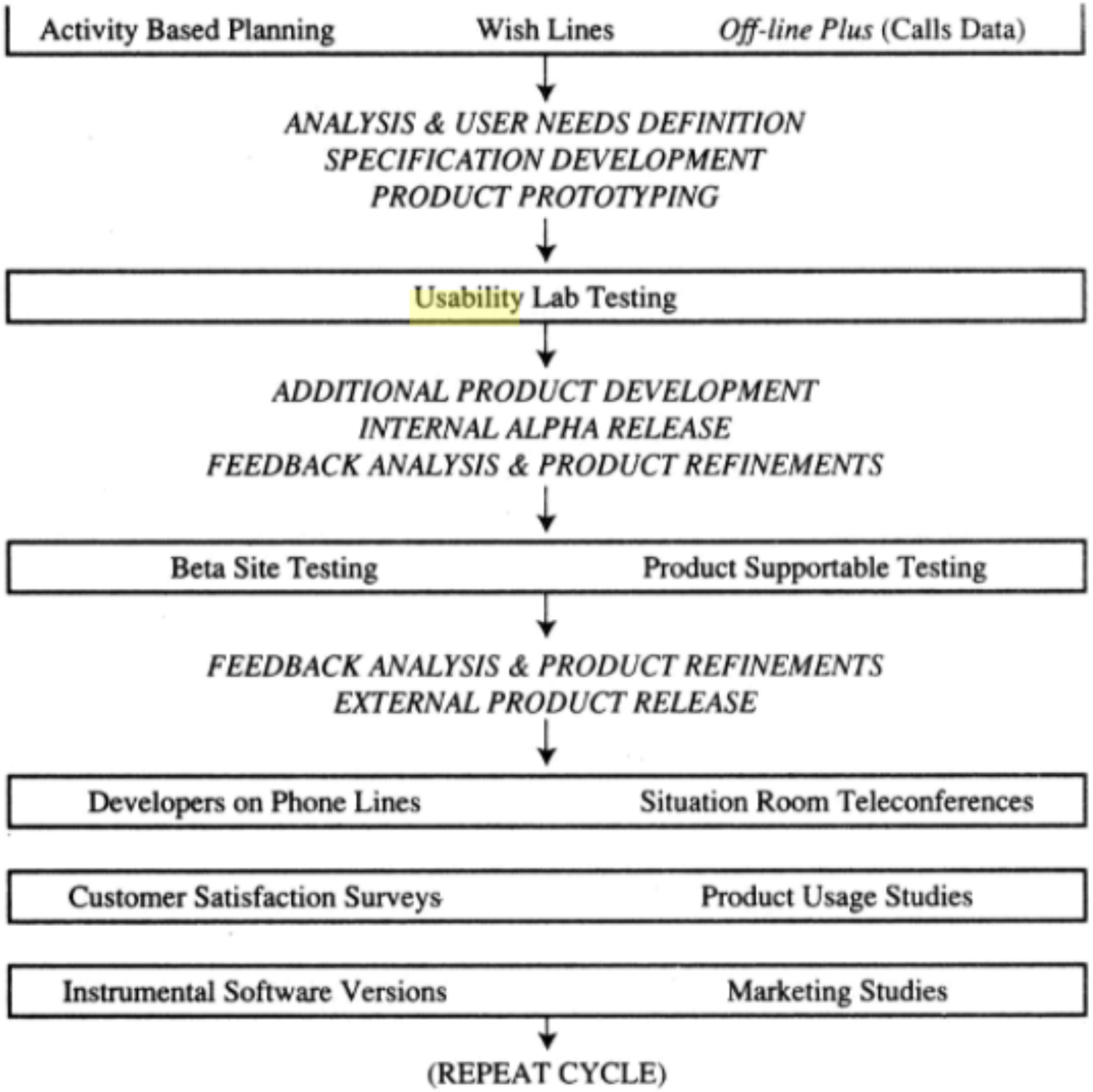
Testers

- Encouraged to communicate with support team and customers, review media evaluations
- Develop testing strategy for high-risk areas
- Many forms of testing (internally called): unstructured testing, ad hoc testing, gorilla testing, free-form Fridays

Early-mid 90s

- Zero defect goal (1989 memo)
- Milestones (first with Publisher 1.0 in 1988)
- Version control, branches, frequent integration
- Daily builds
- Automated tests ("quick autotest") - must succeed before checkin
- Usability labs
- Beta testing (400000 beta testers for Win 95) with instrumentation
- Brief formal design reviews; selected code reviews
- Defect tracking and metrics
- Developers stay in product group for more than one release cycle

Figure 6.1 Customer Input During Product Development



Metrics

- Number of open bugs by severity
  - Number of open bugs expected to decrease before milestone
  - All know severe bugs need to be fixed before release
  - Severity 1 (product crash), Severity 2 (feature crash), Severity 3 (bug with workaround), Severity 4 (cosmetic/minor)
  - Metrics tracked across releases and projects
- Performance metrics
- Bug data used for deciding when “ready to ship”
  - Relative and pragmatic, not absolute view
  - “The market will forgive us for being late, but they won't forgive us for being buggy”

Challenges of Microsoft's Culture

- Little communication among product teams
- Developers and testers often "not so well read in with software-engineering literature, reinventing the wheel"
  - Long underestimated architecture, design, sharing of components, quality metrics, …
- Developers resistant to change and "bureaucracy"

Project Postmortem

- Identify systematic problems and good practices (10-150 page report)
  - document recurring problems and practices that work well
  - e.g.,
    - breadth-first → depth-first & tested milestones
    - insufficient specification
    - not reviewing commits
    - using asserts to communicate assumptions
    - lack of adequate tools → automated tests
    - instrumented versions for testers and beta releases
    - zero defect rule not a priority for developers
- Circulate insights as memos, encourage cross-team learning

Process Audits

- Informal 1-week audits in problematic problems
- Analyzing metrics, interviewing team members
- Recommendations to pick up best practices from other teams
  - daily builds, automated tests, milestones, reviews

The 2002 Trustworthy Computing Memo

There are many changes Microsoft needs to make as a company to ensure and keep our customers' trust at every level -- from the way we develop software, to our support efforts, to our operational and business practices. As software has become ever more complex, interdependent and interconnected, our reputation as a company has in turn become more vulnerable. Flaws in a single Microsoft product, service or policy not only affect the quality of our platform and services overall, but also our customers' view of us as a company.

http://news.microsoft.com/2012/01/11/memo-from-bill-gates/

Code Reviews

- Own code review tools (passaround style)
- Internal studies on how effective reviews are
- Internal tools to improve code reviews

SLAM/SDV (since 2000)

- Goal: Reducing blue screens, often caused by drivers
- Driver verification tool for C
- Model checking technology
- Finds narrow class of protocol violations
  - Use characteristics of drivers (not general C code)
  - Found several bugs in Microsoft's well tested sample drivers
- Fully automated in Microsoft compiler suite
- Available for free
- Enforcement through driver certification program

Ball, Thomas, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM." Communications of the ACM 54.7 (2011): 68-76.

SLAM

- Compelling business case: eliminated most blue screens
- Based on basic science of model checking: originated in university labs with public funding

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability."
--- **Bill Gates, April 18, 2002**

2010: Agile

- Web-based services and C++ evolution requires faster iteration
- Embrace of agile methods
- Massive reduction of testing team (from two testers per developers toward one): developers now expected to do their own testing

Have Agile Techniques been the Silver Bullet for Software Development at Microsoft?

Brendan Murphy
Microsoft Research
Cambridge, UK
bmurphy@microsoft.com

Christian Bird
Microsoft Research
Redmond, USA
cbird@microsoft.com

Thomas Zimmermann
Microsoft Research
Redmond, USA
tzimmer@microsoft.com

Laurie Williams
NCSU
Raleigh, USA
williams@csc.ncsu.edu

Nachiappan Nagappan
Microsoft Research
Redmond, USA
nachin@microsoft.com

Andrew Begel
Microsoft Research
Redmond, USA
andrew.begel@microsoft.com

Table 1: Do you current Use Agile Development Techniques

	2006	2007	2008	2009	2012
Using Agile	34%	51%	56%	49%	57%

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Agile20Trends20ESEM20Master.pdf

Case Study 2:
Static Analysis at Google

Integrate Static Analysis in Review Process

- Static analysis as bots in code review tool
  - Automatically applied on each commit
  - Results visible to author and reviewers
- Lightweight checkers, easy to add and modify
- Feedback buttons to indicate ineffective checkers

Sadowski, Caitlin, et al. "Tricorder: Building a program analysis ecosystem." 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE, 2015.

```java
package com.google.devtools.staticanalysis;

public class Test {
```

Lint
Java
1:02 AM, Aug 21

Missing a Javadoc comment.

Please fix

Not useful

```java
  public boolean foo() {
    return getString() == "foo".toString();
```

ErrorProne
StringEquality
1:03 AM, Aug 21

String comparison using reference equality instead of value equality
(see http://code.google.com/p/error-prone/wiki/StringEquality)

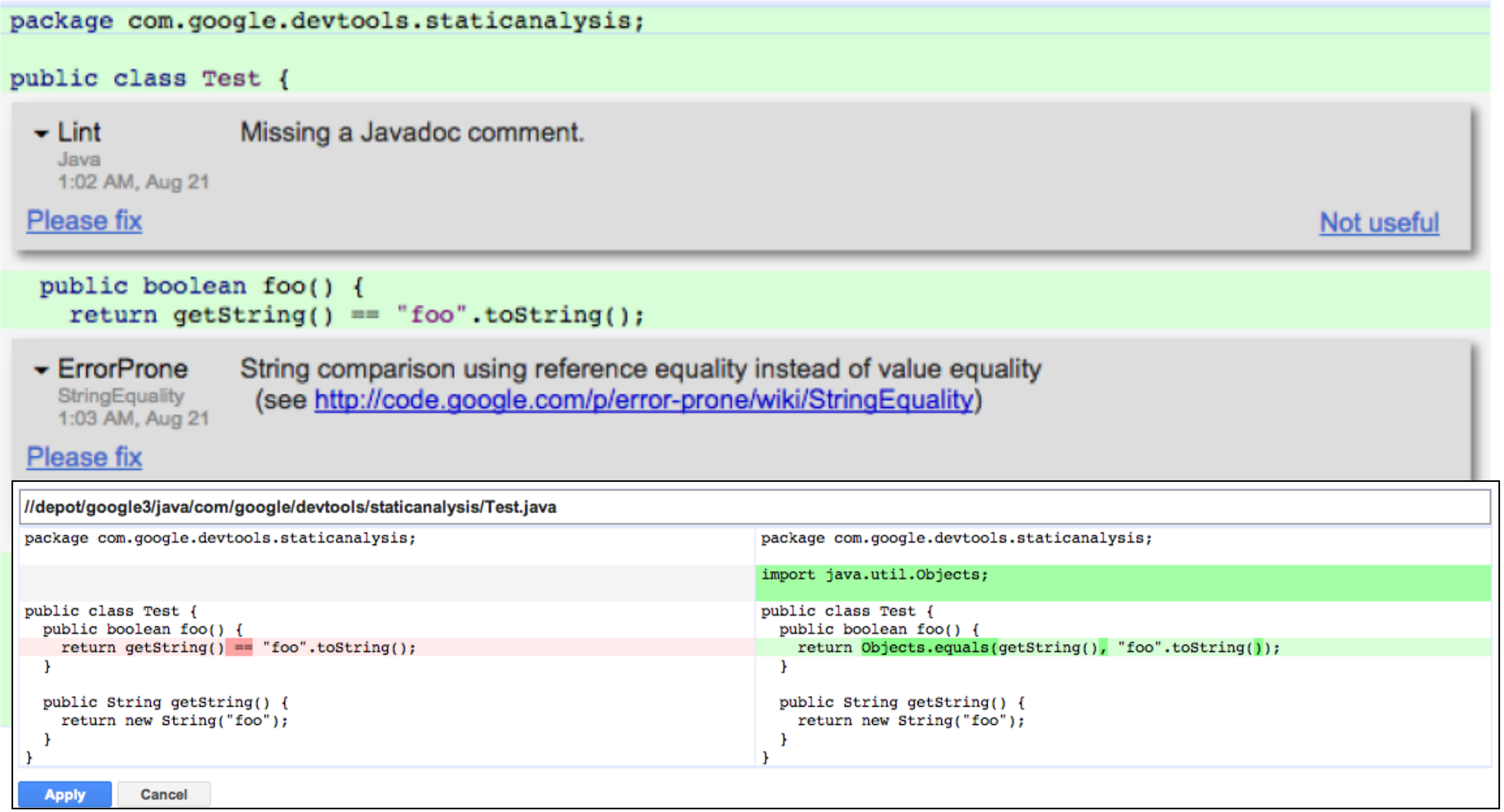
Please fix

Suggested fix attached: show

Not useful

```java
  }

  public String getString() {
    return new String("foo");
  }
}
```


package com.google.devtools.staticanalysis;
public class Test {
Lint
Java
1:02 AM, Aug 21
Missing a Javadoc comment.
Please fix
Not useful
public boolean foo() {
return getString() == "foo".toString();
ErrorProne
StringEquality
1:03 AM, Aug 21
String comparison using reference equality instead of value equality
(see http://code.google.com/p/error-prone/wiki/StringEquality)
Please fix
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;
public class Test {
public boolean foo() {
return getString() == "foo".toString();
}
public String getString() {
return new String("foo");
}
}
package com.google.devtools.staticanalysis;
import java.util.Objects;
public class Test {
public boolean foo() {
return Objects.equals(getString(), "foo".toString());
}
public String getString() {
return new String("foo");
}
}
Apply
Cancel

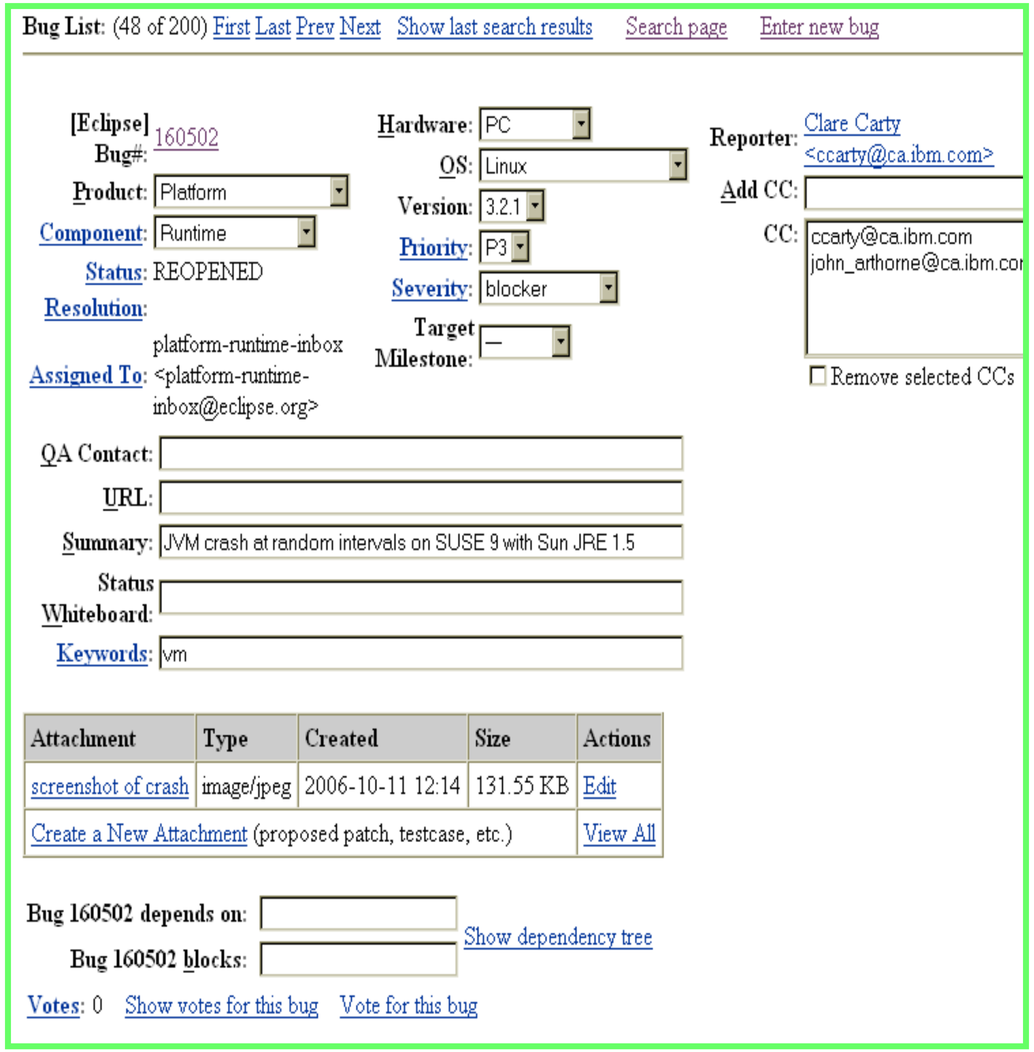
QA within the Process

QA as part of the process

- Have QA deliverables at milestones (management policy)
  - Inspection / test report before milestone
- Change development practices (req. developer buy-in)
  - e.g., continuous integration, pair programming, reviewed checkins, zero-bug static analysis before checking
- Static analysis part of code review (Google)
- Track bugs and other quality metrics

Defect tracking

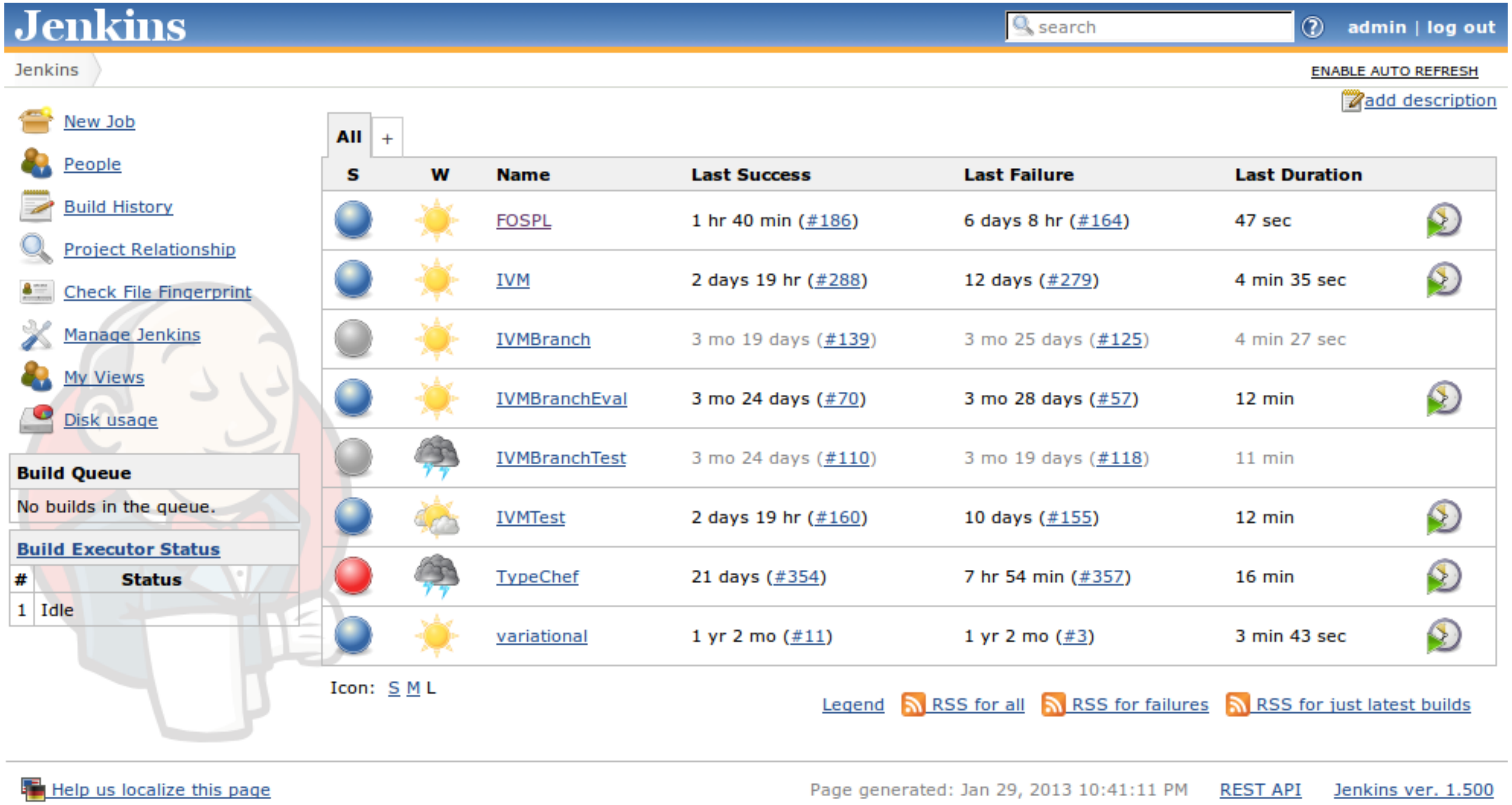
- Issues: Bug, feature request, query
- Basis for measurement
  - reported in which phase
  - duration to repair, difficulty
  - categorization
    -> root cause analysis
- Facilitates communication
  - questions back to reporter
  - ensures reports are not forgotten
- Accountability



Enforcement

- Microsoft: check in gates
  - Cannot check in code unless analysis suite has been run and produced no errors (test coverage, dependency violation, insufficient/bad design intent, integer overflow, allocation arithmetic, buffer overruns, memory errors, security issues)
- eBay: dev/QA handoff
  - Developers run FindBugs on desktop
  - QA runs FindBugs on receipt of code, posts results, require high-priority fixes.
- Google: static analysis on commits, shown in review
- Requirements for success
  - Low false positives
  - A way to override false positive warnings (typically through inspection).
  - Developers must buy into static analysis first

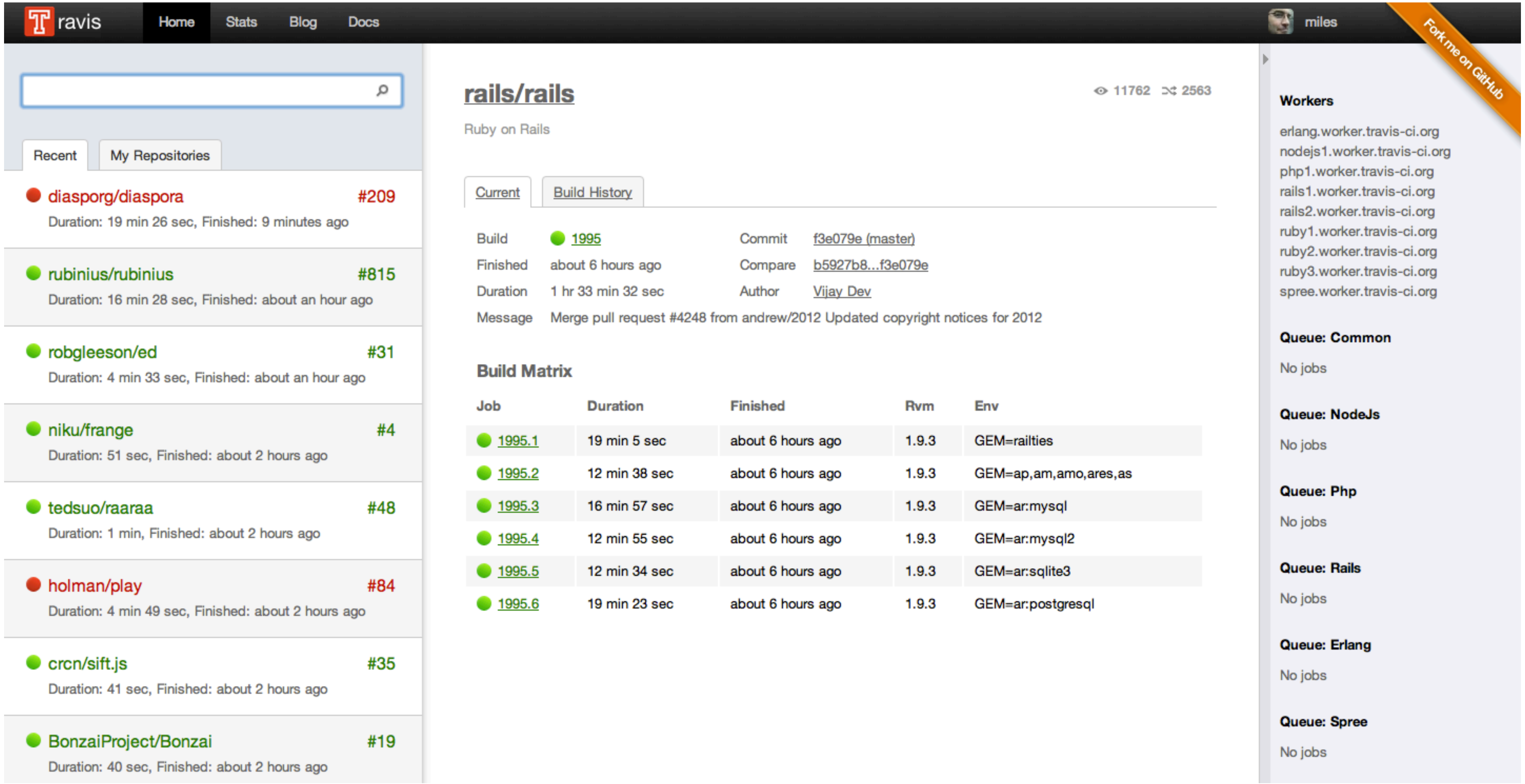
Reminder: Continuous Integration



Automating Test Execution

```
ckaestne@kastner-desktop:~/work/TypeChef/TypeChef$ sbt "project FeatureExprLib" test
Detected sbt version 0.12.2
[info] Loading global plugins from /usr0/home/ckaestne/.sbt/plugins
[info] Loading project definition from /usr0/home/ckaestne/work/TypeChef/TypeChef/project/project
[info] Loading project definition from /usr0/home/ckaestne/work/TypeChef/TypeChef/project
[info] Set current project to TypeChef (in build file:/usr0/home/ckaestne/work/TypeChef/TypeChef/)
[info] Set current project to FeatureExprLib (in build file:/usr0/home/ckaestne/work/TypeChef/TypeChef/)
[info] Compiling 10 Scala sources to /usr0/home/ckaestne/work/TypeChef/TypeChef/FeatureExprLib/target/scala-2.10/test
-classes...
[info] + FeatureExpr.parse(print(x))==x: OK, passed 100 tests.
[info] + FeatureExpr.and1: OK, passed 100 tests.
[info] + FeatureExpr.and0: OK, passed 100 tests.
[info] + FeatureExpr.andSelf: OK, passed 100 tests.
[info] + FeatureExpr.or1: OK, passed 100 tests.
[info] + FeatureExpr.or0: OK, passed 100 tests.
[info] + FeatureExpr.orSelf: OK, passed 100 tests.
[info] + FeatureExpr.a eq a: OK, passed 100 tests.
[info] + FeatureExpr.a equals a: OK, passed 100 tests.
[info] + FeatureExpr.a equivalent a: OK, passed 100 tests.
[info] + FeatureExpr.a implies a: OK, passed 100 tests.
[info] + FeatureExpr.creating (a and b) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.creating (a or b) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.creating (not a) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.applying not twice yields an equivalent formula: OK, passed 100 tests.
[info] + FeatureExpr.Commutativity wrt. equivalence: (a and b) produces the same object as (b and a): OK, passed 100
tests.
[info] + FeatureExpr.Commutativity wrt. equivalence: (a or b) produces the same object as (b or a): OK, passed 100 te
sts.
[info] + FeatureExpr.taut(a=>b) == contr(a and !b): OK, passed 100 tests.
[info] + FeatureExpr.featuremodel.tautology: OK, passed 100 tests.
```

Continuous Integration with Travis-CI



Summary

- Developing a QA plan:
  - Identify quality goals and risks
  - Mix and match approaches
  - Enforce QA, establish practices
- Case study from Microsoft
- Integrate QA in process

Further Reading

- Cusumano, Michael A., and Richard W. Selby. "Microsoft secrets." (1997).
  - Book covers quality assurance at Microsoft until the mid 90s (and much more)
- Ball, Thomas, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM." Communications of the ACM 54.7 (2011): 68-76.
  - An overview of SLAM at Microsoft
- Jaspan, Ciera, I. Chen, and Anoop Sharma. "Understanding the value of program analysis tools." *Companion OOPSLA*. ACM, 2007.
  - Description of eBay evaluating FindBugs
- Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., & Winter, C. Tricorder: Building a Program Analysis Ecosystem. ICSE 2015
  - Integrating static analysis into code reviews at Google in a data-driven way
- Sommerville. Software Engineering. 8th Edition. Chapter 27
  - QA planning and process improvement, standards