Lecture 16 – Intro to QA, Testing

Claire Le Goues, Michael Hilton, Chris Meiklejohn

Architecture Recap

- "Think before implementing"
- Design and analyze for qualities of interest (e.g., performance, scalability, security, extensibility)
- From informal sketches to formal models; styles and tactics to guide discussion

Administrativia?

Learning goals

- Conceive of testing as an activity designed to achieve *coverage* along a number of (non-structural!) dimensions.
- Enumerate testing strategies to help evaluate the following quality attributes: usability, reliability, security, robustness (both general and architectural), performance, integration.
- Give tradeoffs and identify when each of those techniques might be useful.
- Integrate testing into your project's lifecycle and practices.
- Outline a test plan.

QA IS HARD

"We had initially scheduled time to write tests for both front and back end systems, although this never happened."

"Due to the lack of time, we could only conduct individual pages' unit testing. Limited testing was done using use cases. Our team felt that this testing process was rushed and more time and effort should be allocated."

"We failed completely to adhere to the initial [testing] plan. From the onset of the development process, we were more concerned with implementing the necessary features than the quality of our implementation, and as a result, we delayed, and eventually, failed to write any tests."

Time estimates (in hours):

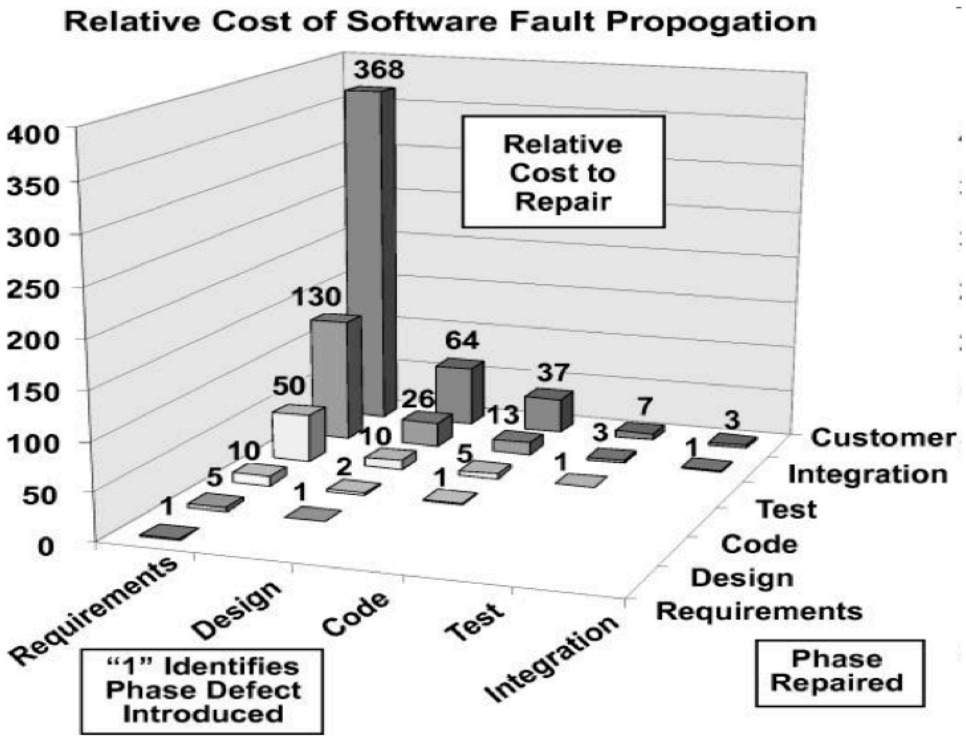
Activity	Estimated	Actual
testing plans	3	0
unit testing	3	1
validation testing	4	2
test data	1	1

"One portion we planned for but were not able to complete to our satisfaction was testing."

"[W]e did not end up using Github Issues and Milestones for progress tracking, because of our concern for implementing features. Additionally, once we started the development process, we felt that Github Issues and Milestones had too much overhead for only a week-long development process."

QA IS IMPORTANT (DUH!)

Cost


Relative Cost of Software Fault Propogation
400
350
300
250
200
150
100
50
0
368
130
64
50
37
26
13
10
10
7
5
5
3
3
2
1
1
1
1
Relative Cost to Repair
Requirements
Design
Code
Test
Integration
Customer
Integration
Test
Code
Design
Requirements
"1" Identifies Phase Defect Introduced
Phase Repaired

Cost

theguardian

News | US | World | Sports | Comment | Culture | Business | Money | Environment | Science

News › Technology › Heartbleed

Heartbleed: developer who introduced the error regrets 'oversight'

Submitted just seconds before new year in 2012, the bug 'slipped through' – but discovery 'validates' open source

Share 430
Tweet 269
+1 27
Share 103
Email

Alex Hern
Follow @alexhern Follow @guardiantech
theguardian.com, Friday 11 April 2014 03.05 EDT
Jump to comments (108)

Technology
Heartbleed · Open source · Programming · Software · Internet · Hacking · Data and computer security

More news

More on this story

QA HAS MANY FACETS

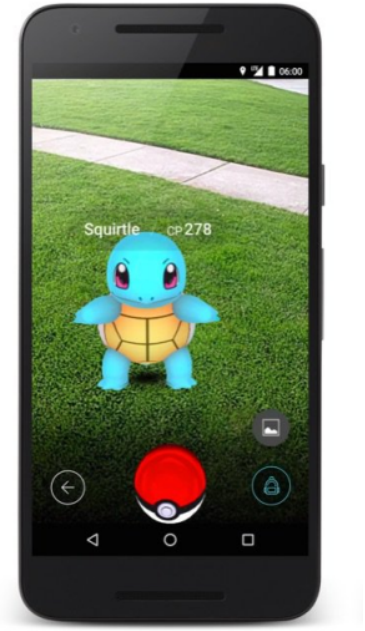
Questions

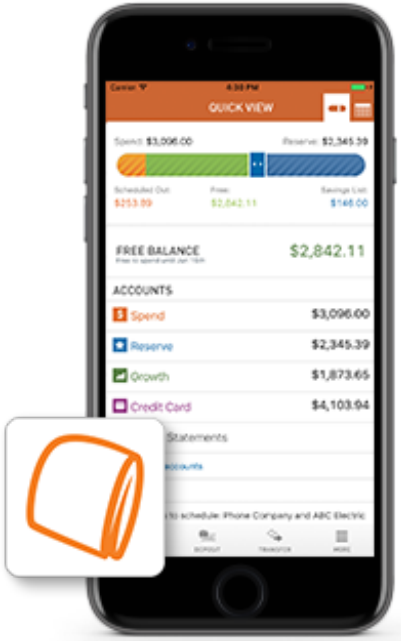
- How can we ensure that the specifications are correct?
- How can we ensure a system meets its specification?
- How can we ensure a system meets the needs of its users?
- How can we ensure a system does not behave badly?

Validation vs Verification

- **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?

Brief Case Discussion





What qualities are important and how can you assure them?

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…

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, …

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.

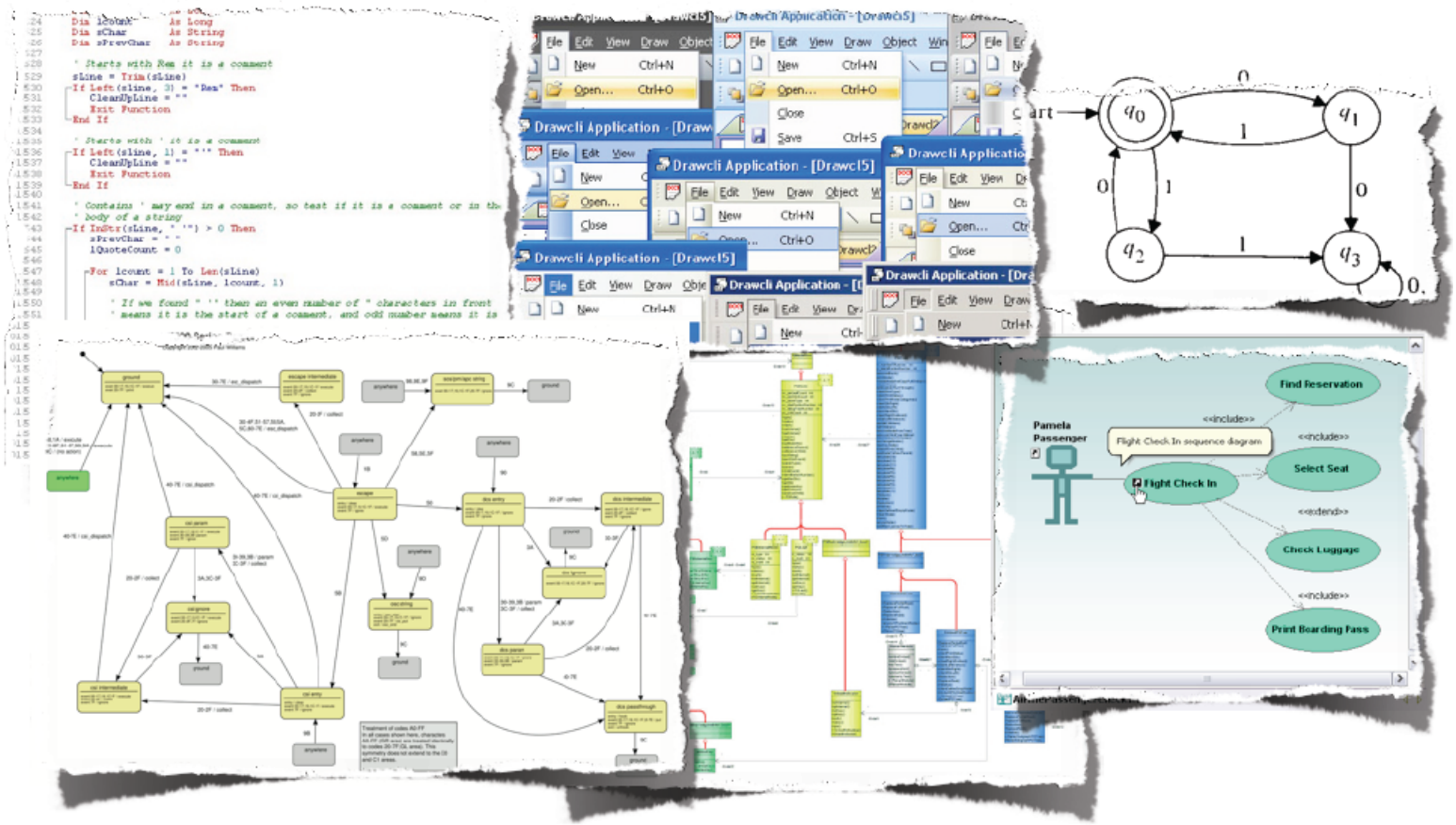
No Single Technique

- *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…

“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.

CLASSIC TESTING (FUNCTIONAL CORRECTNESS)

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:
  - 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

Software Errors

- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- …

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

Packages

All
net.sourceforge.cobertura.ant
net.sourceforge.cobertura.check
net.sourceforge.cobertura.coveragedat
net.sourceforge.cobertura.instrument
net.sourceforge.cobertura.merge
net.sourceforge.cobertura.reporting
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.xm
net.sourceforge.cobertura.util

All Packages

Classes

AntUtil *(88%)*
Archive *(100%)*
ArchiveUtil *(80%)*
BranchCoverageData *(N/A)*
CheckTask *(0%)*
ClassData *(N/A)*
ClassInstrumenter *(94%)*
ClassPattern *(100%)*
CoberturaFile *(73%)*
CommandLineBuilder *(96%)*
CommonMatchingTask *(88%)*
ComplexityCalculator *(100%)*
ConfigurationUtil *(50%)*
CopyFiles *(87%)*
CoverageData *(N/A)*
CoverageDataContainer *(N/A)*
CoverageDataFileHandler *(N/A)*
CoverageRate *(0%)*
ExcludeClasses *(100%)*
FileFinder *(96%)*
FileLocker *(0%)*
FirstPassMethodInstrumenter *(100%)*
HTMLReport *(94%)*
HasBeenInstrumented *(N/A)*
Header *(80%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Compl
All Packages	55	75%	1625/2179	64%	472/738	
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	
someotherpackage	1	83%	5/6	N/A	N/A	

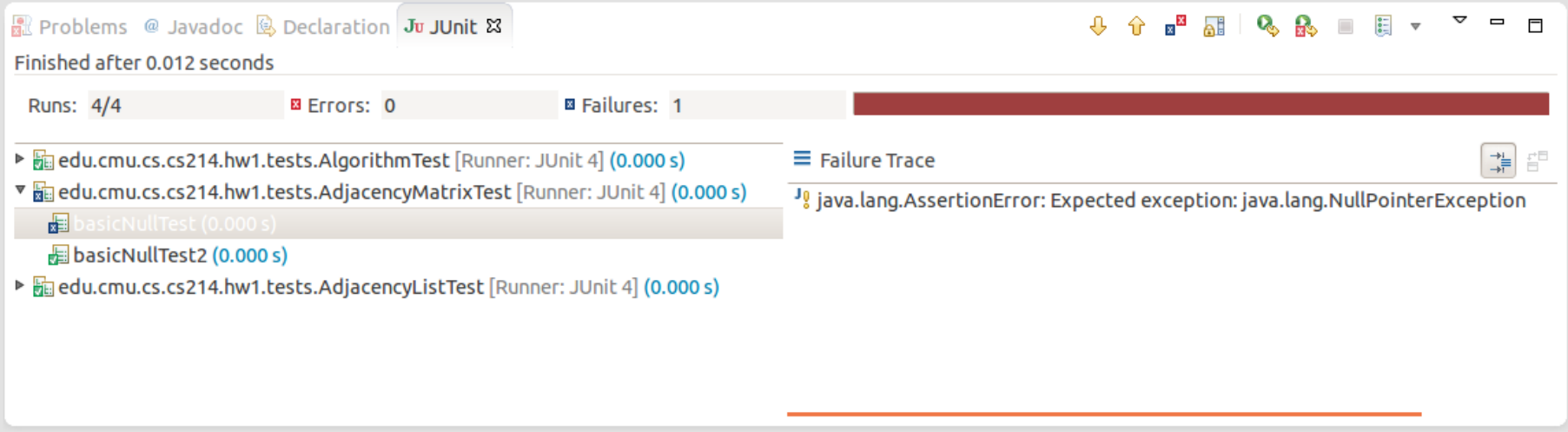
Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

Testing Levels

- Unit testing
- Integration testing
- System testing

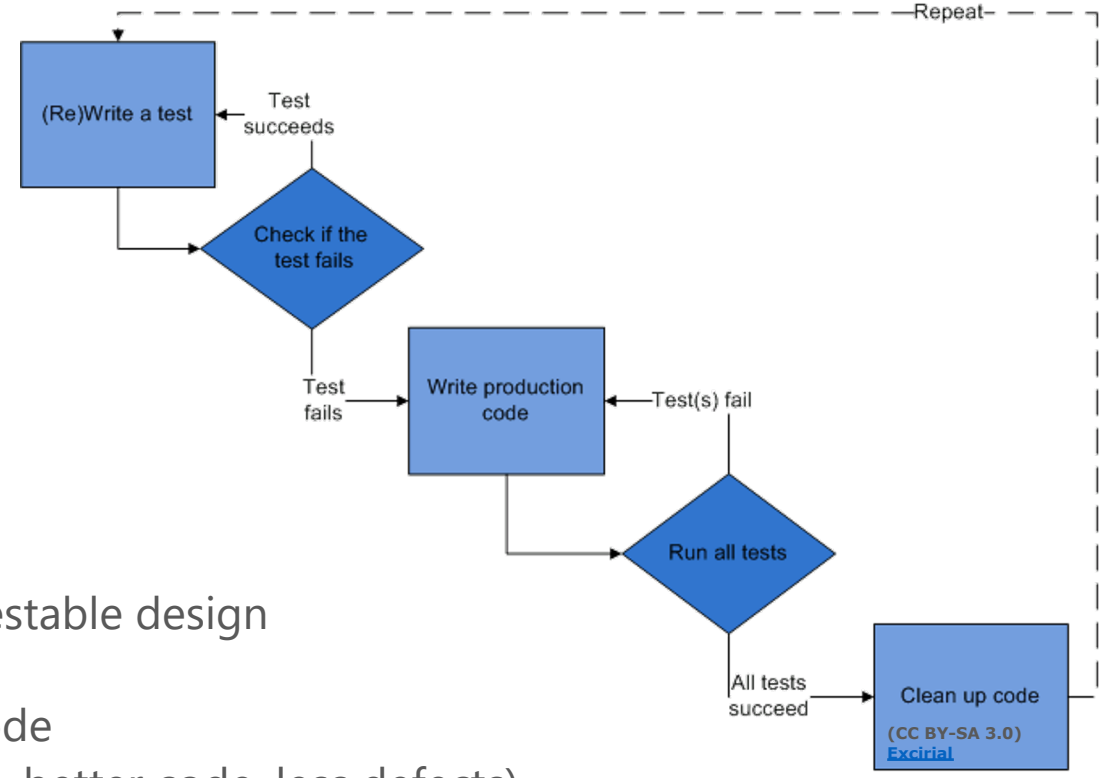
JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism

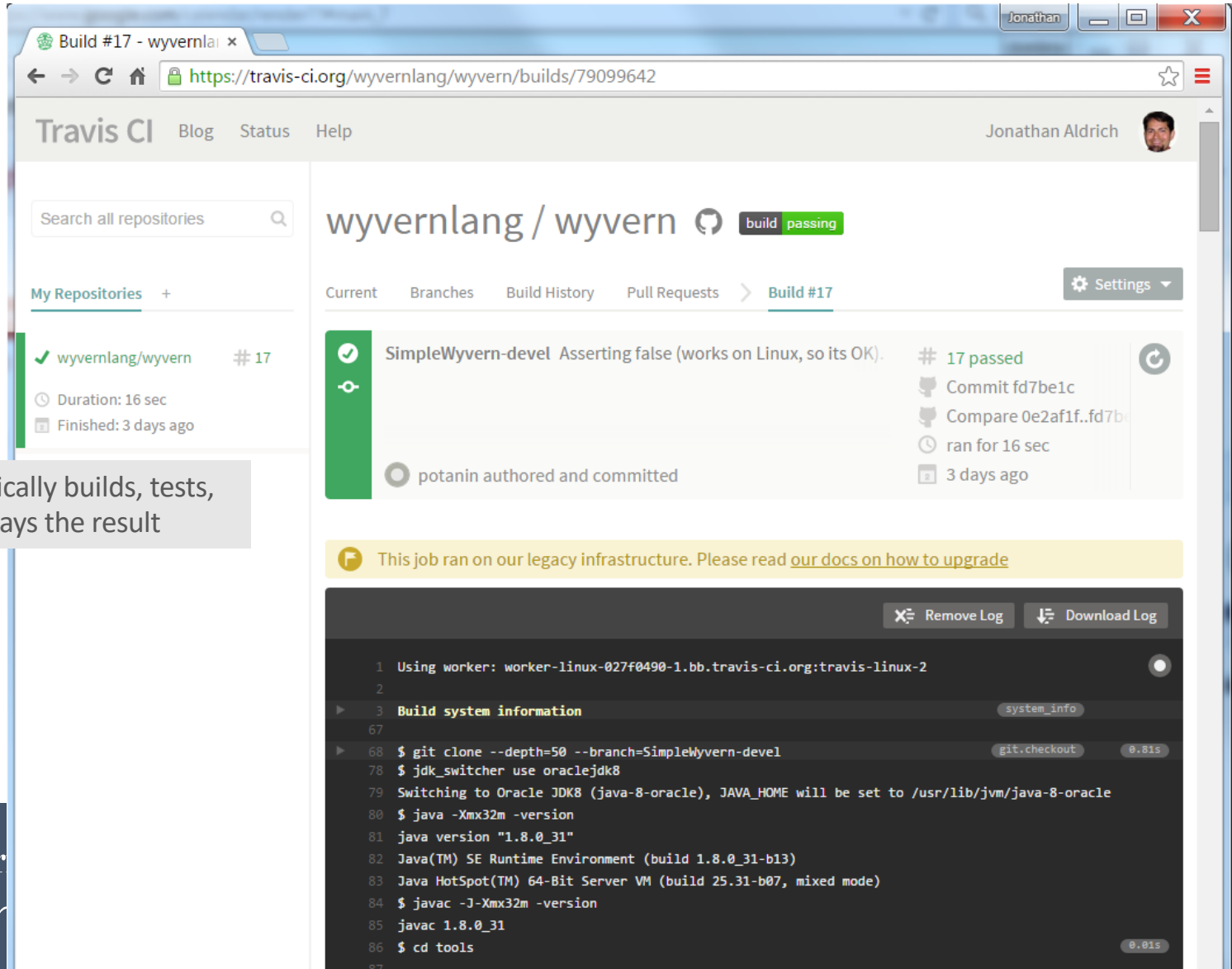


Test Driven Development

- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid writing unneeded code
  - Higher product quality (e.g. better code, less defects)
  - Higher test suite quality
  - Higher overall productivity

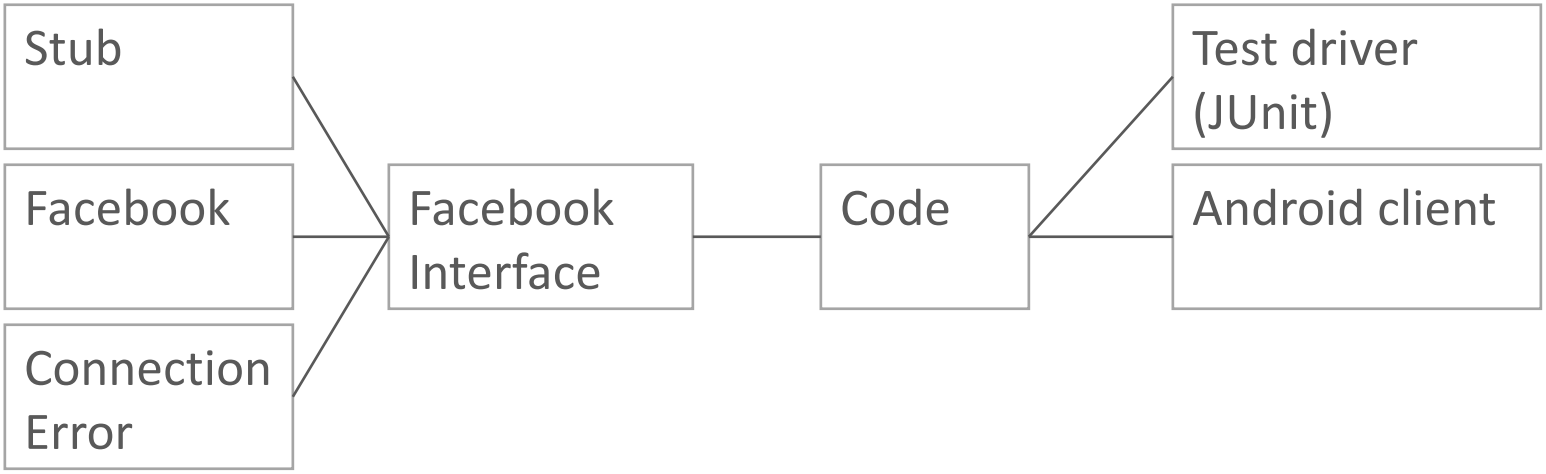


Continuous Integration



Automatically builds, tests, and displays the result

Testing with Stubs



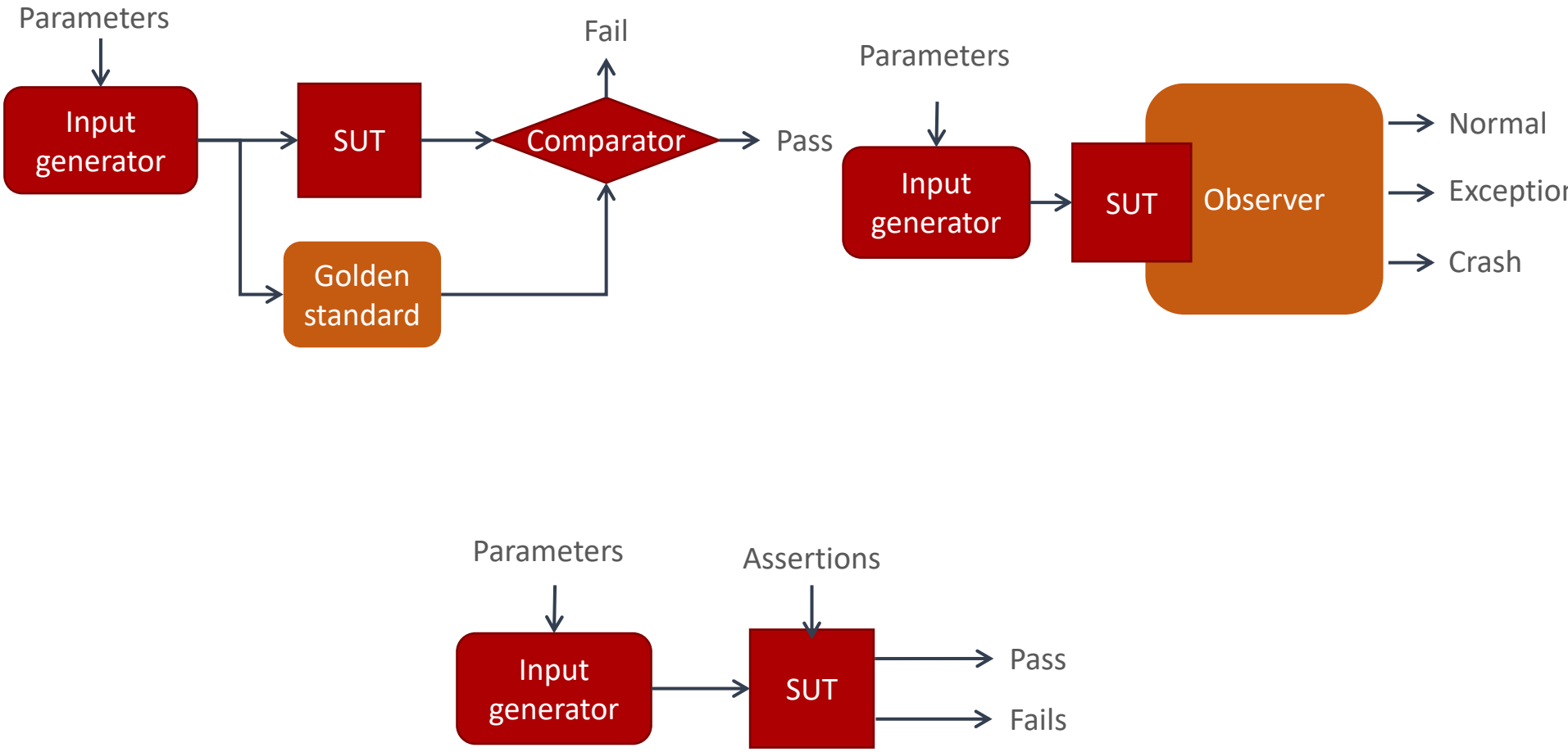

```
class ConnectionError implements FacebookInterface {
        List<Node> getPersons(String name) {
                throw new HttpConnectionException();
        }
}

@Test void testConnectionError() {
        assert getFriends(new ConnectionError()) == null;
}
```

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.

The Oracle Problem



TESTING BEYOND FUNCTIONAL CORRECTNESS

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

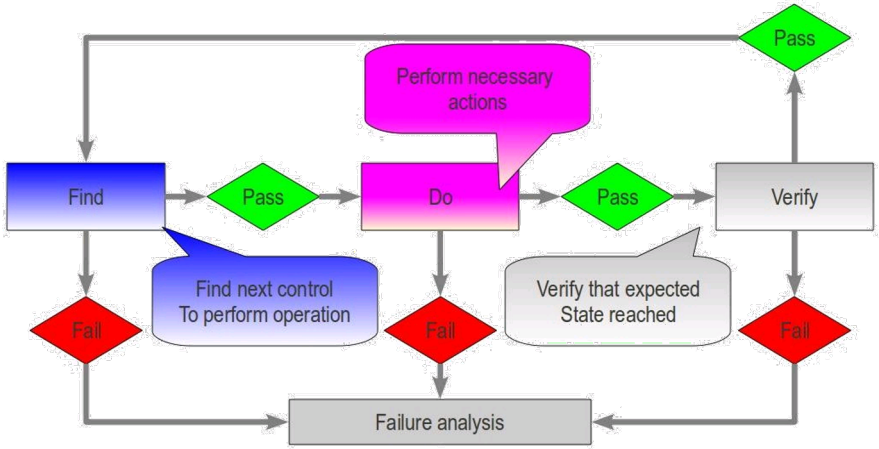
TESTING USABILITY

Usability Testing

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

Automating GUI/Web Testing

- This is 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)
- Beyond functional correctness?



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 Mes-sage"	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?

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

TESTING SECURITY/ROBUSTNESS

Security/Robustness Testing

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

Random testing

- Select inputs independently at random from the program's input domain:
  - Identify the input domain of the program.
  - Map random numbers to that input domain.
  - Select inputs from the input domain according to some probability distribution.
  - Determine if the program achieves the appropriate outputs on those inputs.
- Random testing can provide probabilistic guarantees about the likely faultiness of the program.
  - E.g., Random testing using ~23,000 inputs without failure (N = 23, 000) establishes that the program will not fail more than one time in 10,000 (F = 10^4), with a confidence of 90% (C = 0.9).

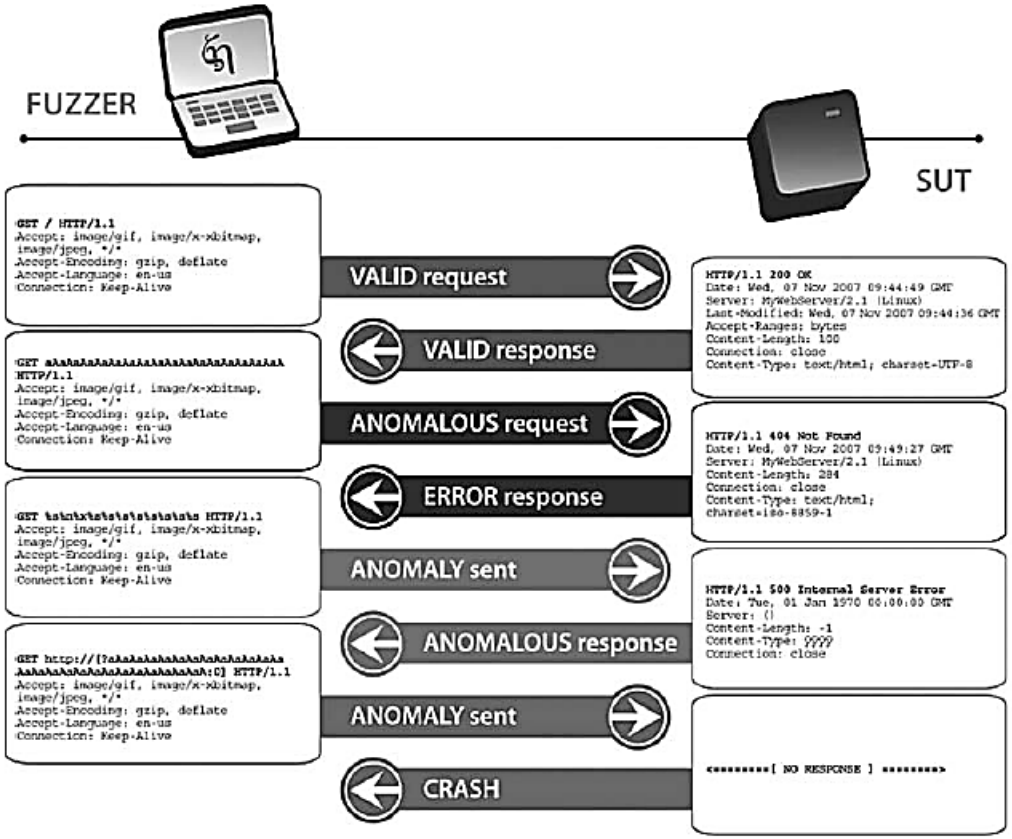
Reliability: Fuzz testing

- 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 (A. Takanen et al, Fuzzing for Software Security Testing and Quality Assurance, 2008)
- Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called **fuzzers**.

Types of faults found

- Pointer/array errors
- Not checking return codes
- Invalid/out of boundary data
- Data corruption
- Signed characters
- Race conditions
- Undocumented features
- …Possible tradeoffs?

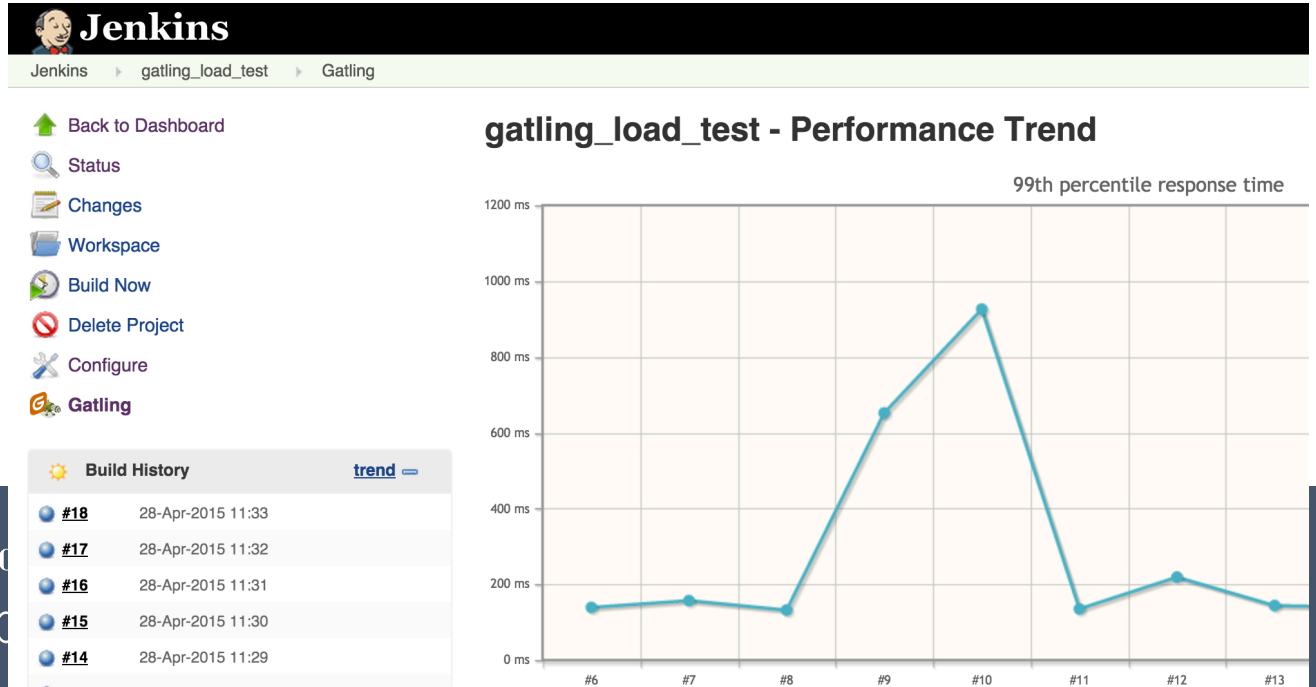
Fuzzing process



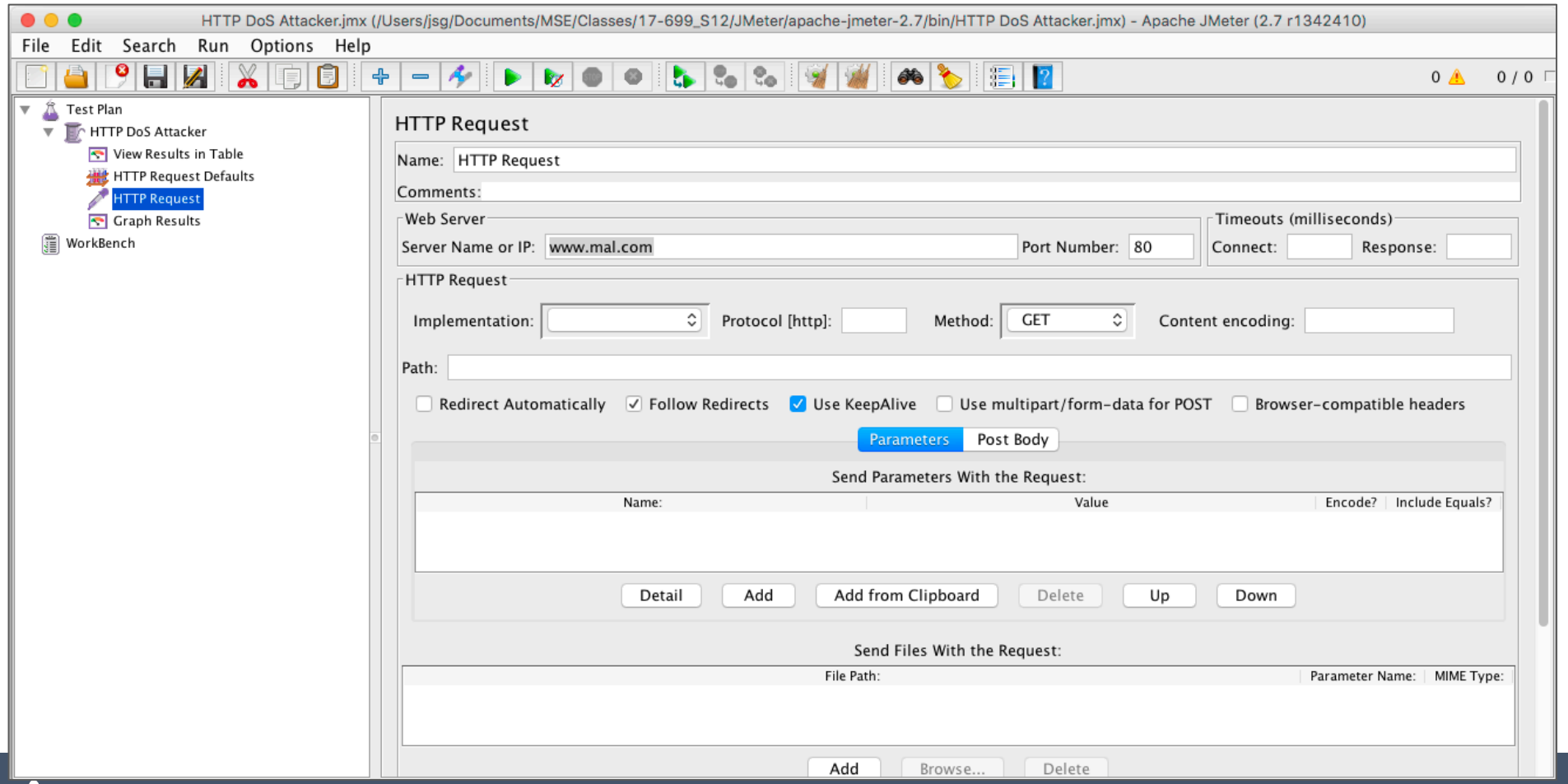
TESTING PERFORMANCE

Unit and regression testing for performance

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



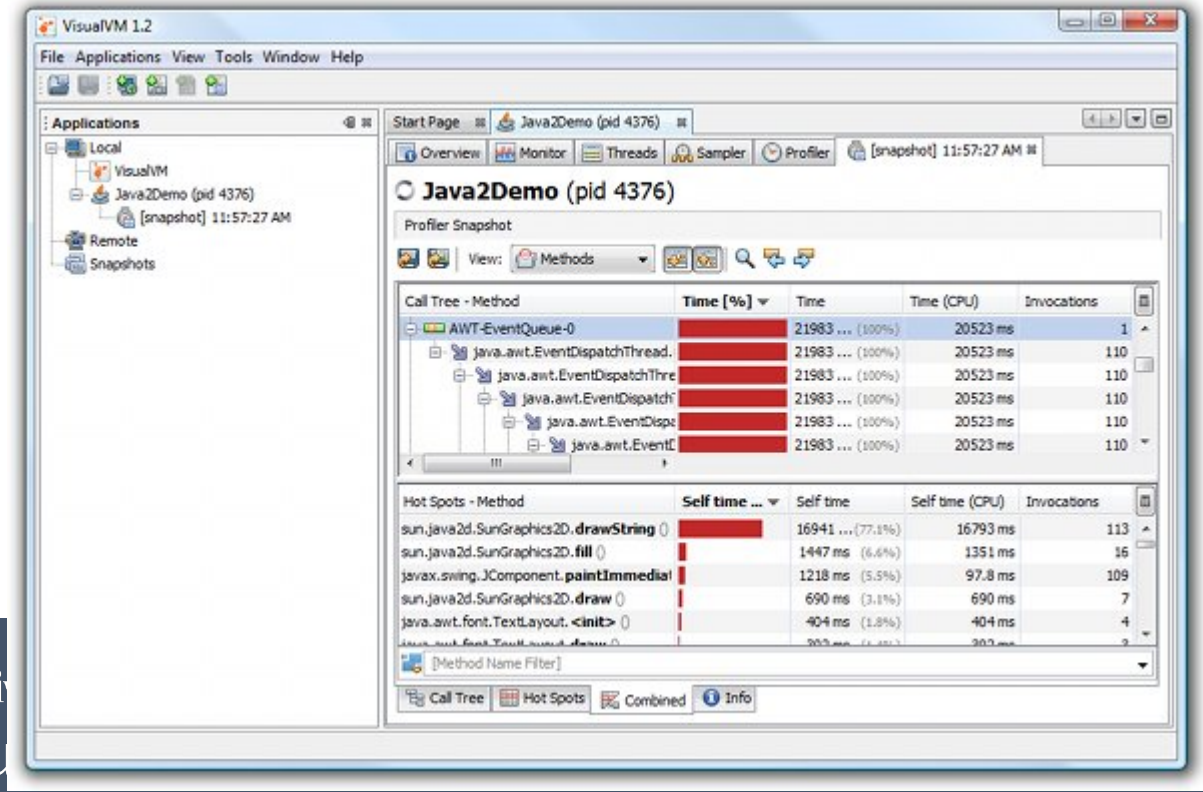
Performance testing tools: JMeter



http://jmeter.apache.org

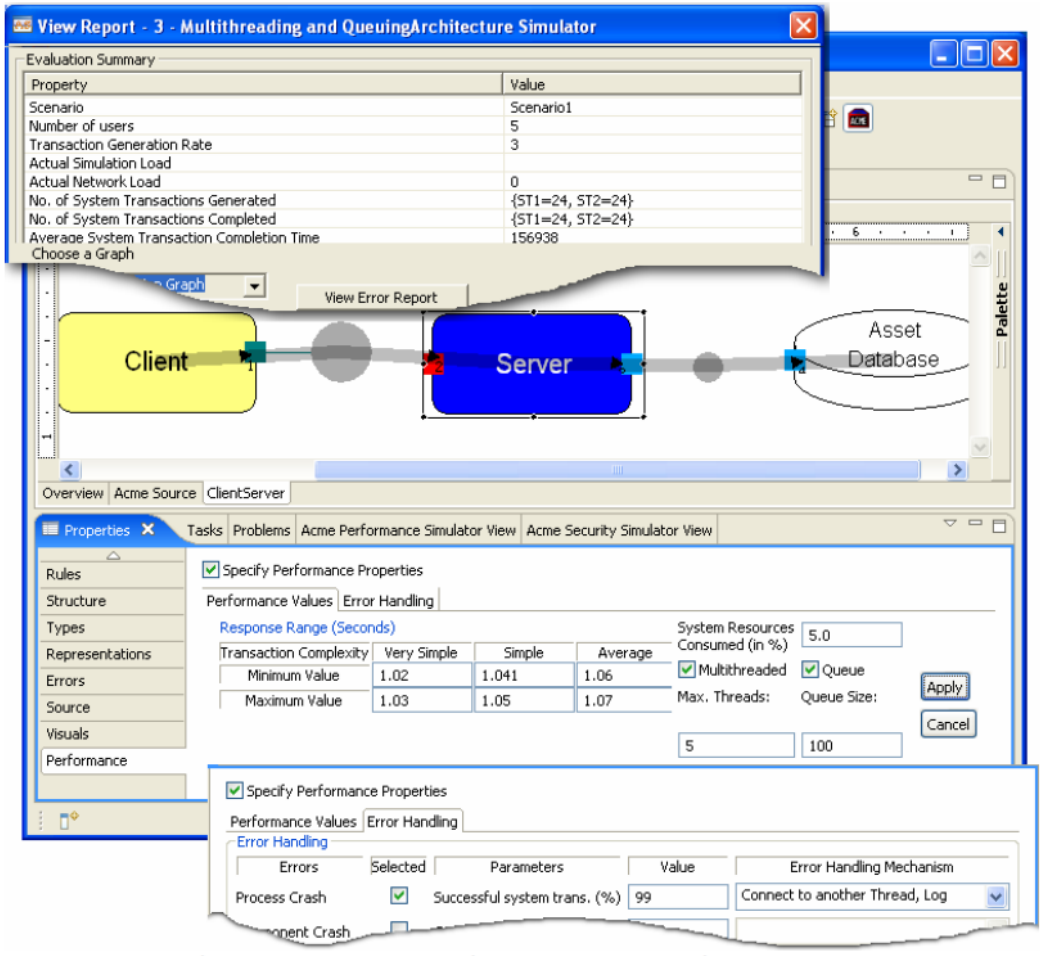
Profiling

- Finding bottlenecks in execution time and memory



Performance Testing during Design

- Modeling and simulation
  - e.g. queuing theory



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

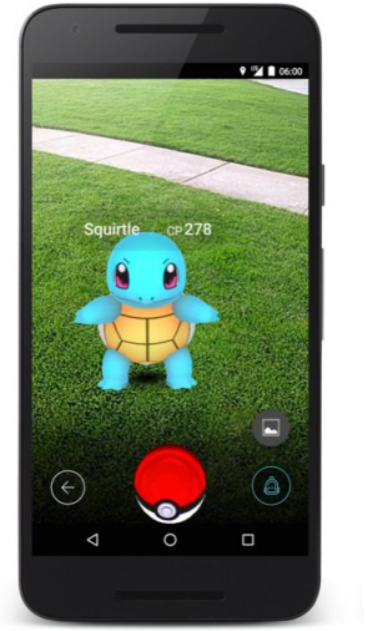
- **Problem:** A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we'll talk about that later).
- **Soak testing:** testing a system with a significant load over a significant period of time (*positive*).
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

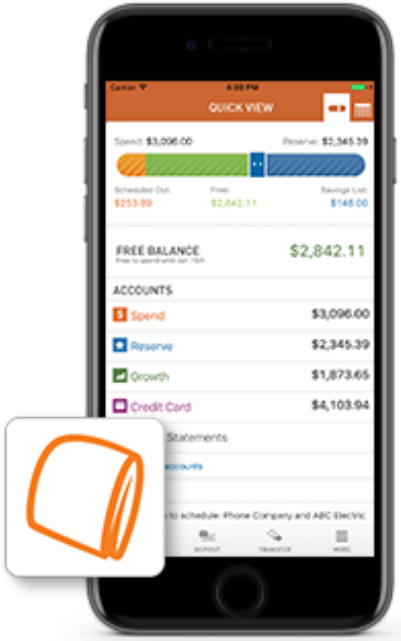

amazon
web services
CHAOS
MONKEY

Chaos monkey/Simian army

- A Netflix infrastructure testing system.
- "Malicious" programs randomly trample on components, network, datacenters, AWS instances…
  - Chaos monkey was the first – disables production instances at random.
  - 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.
- Netflix has open-sourced their chaos monkey code.

Brief Case Discussion





What qualities are important and how can you assure them?

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 purposes - 1

Technique	Description
Baseline testing	• Execute a single transaction as a single virtual user for a set period of time or for a set number of transaction iterations • Carried out without other activities under otherwise normal conditions • Establish a point of comparison for further test runs
Load testing	• Test application with target maximum load but typically no further • Test performance targets (i.e. response time, throughput, etc.) • Approximation of expected peak application use
Scalability testing	• Test application with increasing load • Scaling should not require new system or software redesign

Testing purposes - 2

Technique	Description
Soak (stability) testing	• Supply load to application continuously for a period of time • Identify problems that appear over extended period of time, for example a memory leak
Spike testing	• Test system with high load for short duration • Verify system stability during a burst of concurrent user and/or system activity to varying degrees of load over varying time periods
Stress testing	• Overwhelm system resources • Ensure the system fails and recovers gracefully

Completeness?

- Statistical thresholds
  - Defects reported/repaired
  - Relative proportion of defect kinds
  - Predictors on "going gold"
- Coverage criterion
  - E.g., 100% coverage required for avionics software
  - Distorts the software
  - Matrix: Map test cases to requirements use cases
- Can look at historical data
  - Within an organization, can compare across projects; Develop expectations and predictors
  - (More difficult across organizations, due to difficulty of commensurability, E.g., telecon switches vs. consumer software)
- Rule of thumb: when error detection rate drops (implies diminishing returns for testing investment).
- Most common: Run out of time or money