Foundations of Software Engineering

Lecture 11 – Intro to QA, Testing

Claire Le Goues

institute for SOFTWARE RESEARCH

Learning goals

- Define software analysis.
- Reason about QA activities with respect to coverage and coverage/adequacy criteria, both traditional (structural) and non-traditional.
- 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.

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

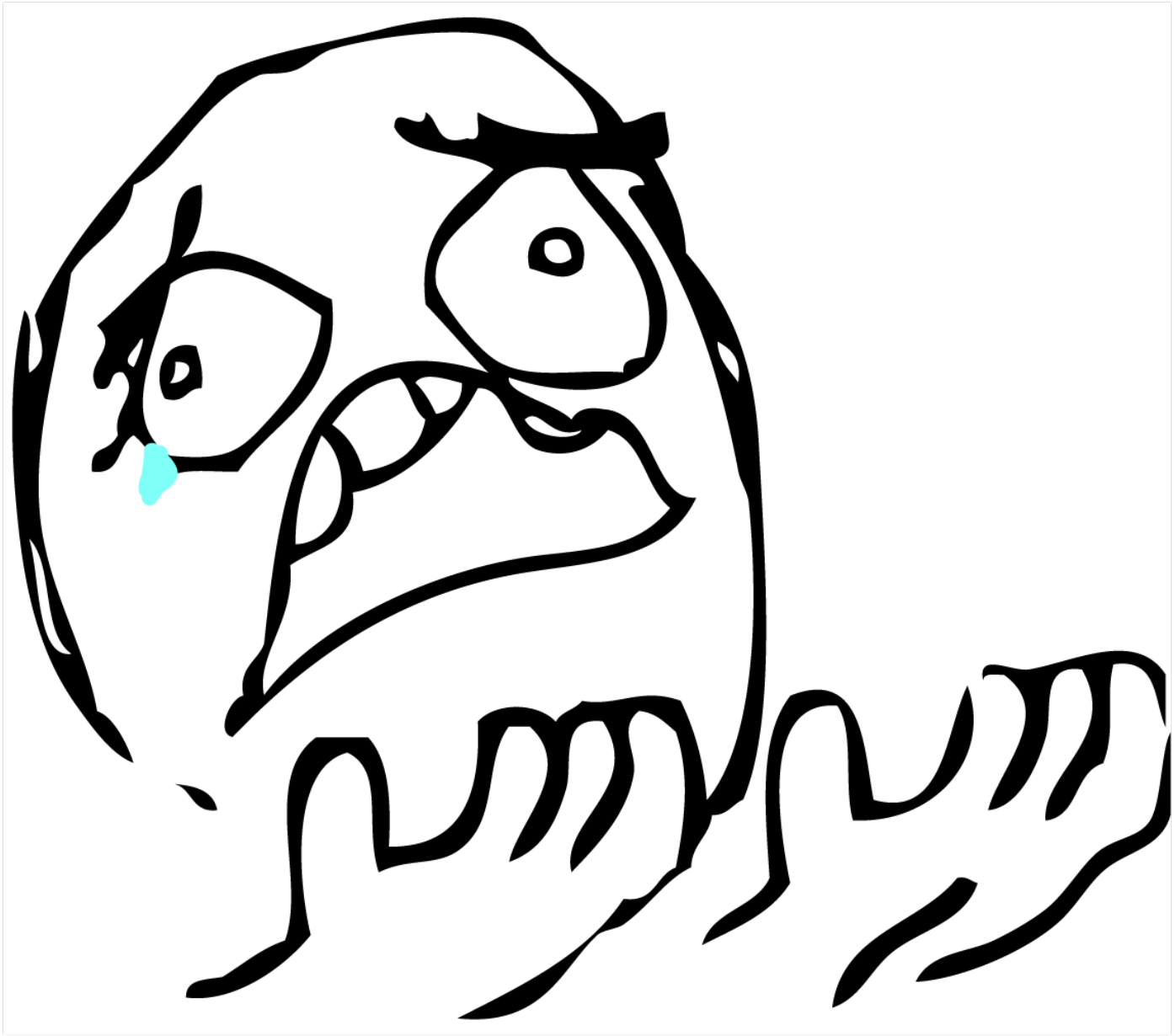
"Due to the lack of time, we could only conduct individual pages' unit testing. Limited testing was done using use cases. Our team felt that this testing process was rushed and more time and effort should be allocated."

"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

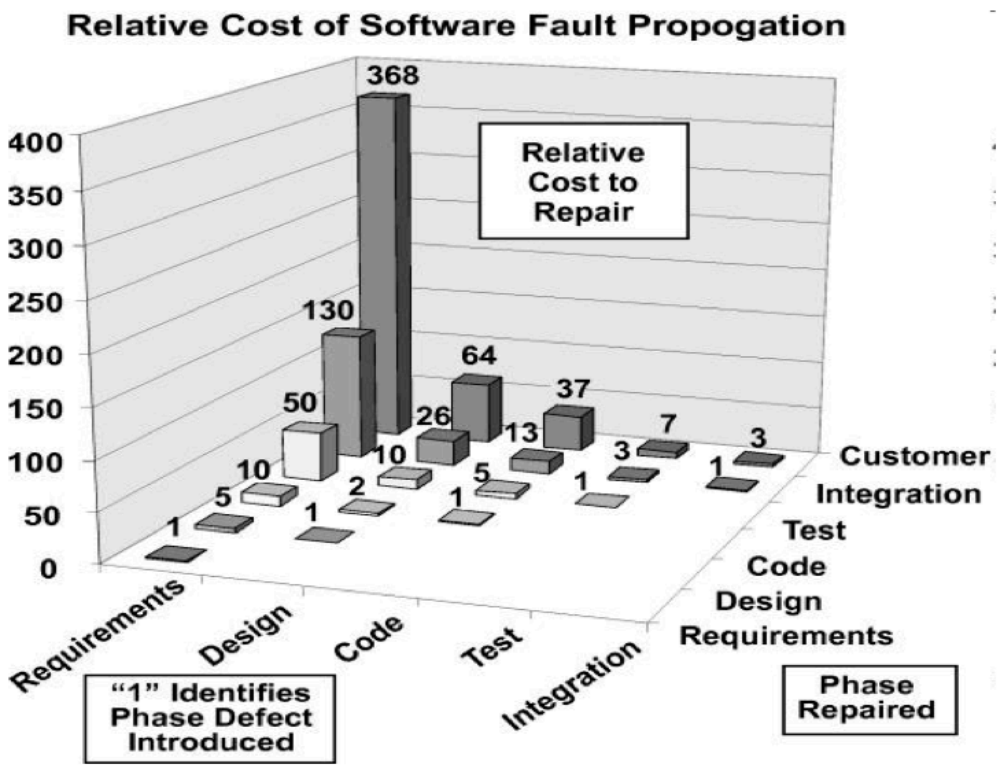
"One portion we planned for but were not able to complete to our satisfaction was testing."



…WHY??!!!!1!!!11

"[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."

Cost



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

HOW DO YOU KNOW THAT YOUR PROGRAM WORKS?

Questions

- 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?

QUALITY ATTRIBUTES??

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.

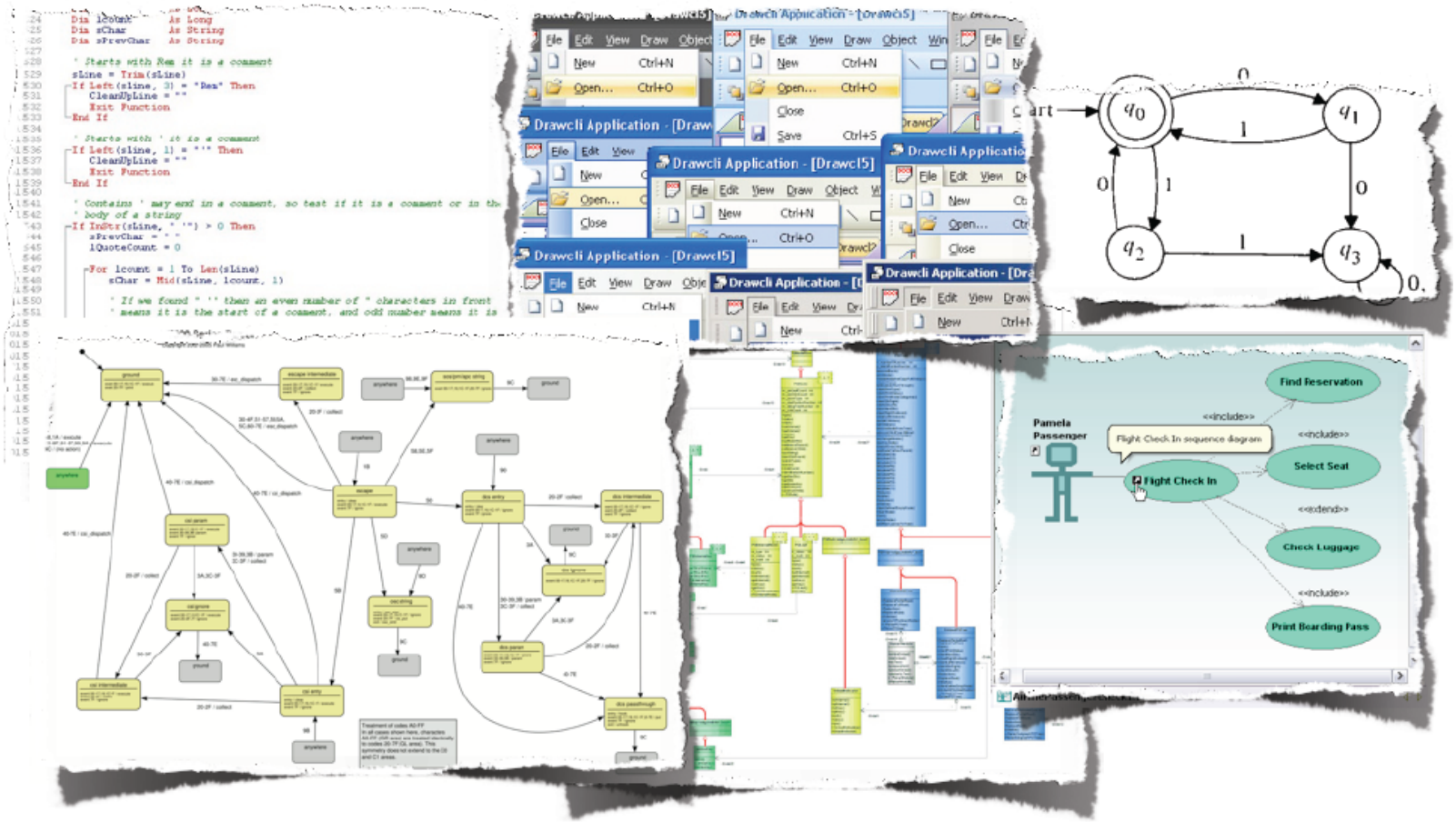
One slide with a bunch of ideas you should remember from 15-214

- Verification vs. testing
- Black box
- TDD
- Testing harness, scaffolding, stubs
- Unit testing/Junit
- Nightly vs. smoke tests
- **Coverage**

“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.

Integration: object protocols

- Covers the space of possible API calls, or program "conceptual states."
- Develop test cases that involve representative sequence of operations on objects
  - Example: Dictionary structure: `Create, AddEntry*, Lookup, ModifyEntry*, DeleteEntry, Lookup, Destroy`
  - Example: IO Stream: `Open, Read, Read, Close, Read, Open, Write, Read, Close, Close`
  - Test concurrent access from multiple threads
    - Example: FIFO queue for events, logging, etc.

```
Create      Put      Put      Get      Get
      Put      Get      Get      Put      Put      Get
```

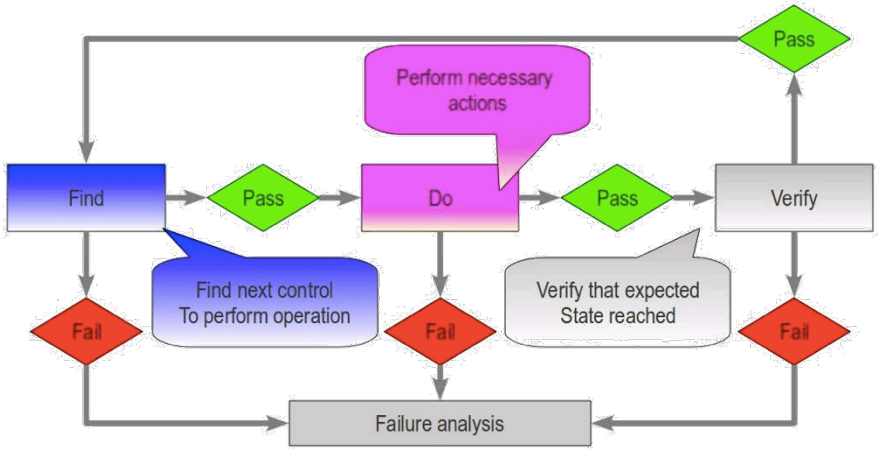
- Approach
  - Develop representative sequences – based on use cases, scenarios, profiles
  - Randomly generate call sequences
- Also useful for protocol interactions within distributed designs.

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 (from 214)

- 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!

HOW DOES THIS TECHNIQUE GENERALIZE, ESPECIALLY TECHNICALLY?

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

(214 review) 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).

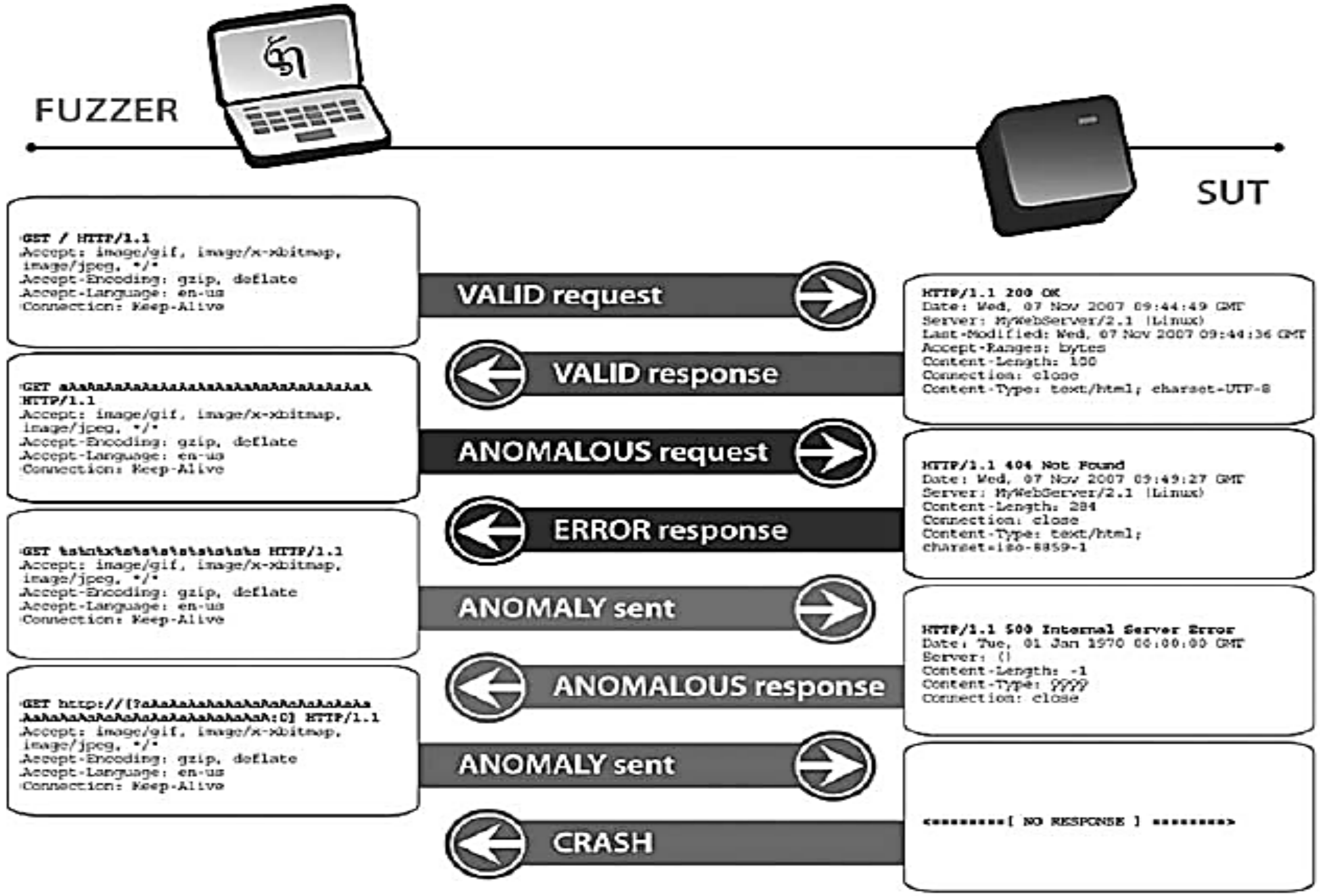
Reliability: Fuzz testing

- 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



institute for SOFTWARE RESEARCH

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.

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

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.