ECE444: Software Engineering
QA 2: Performance Testing, Chaos Engineering, Static & Dynamic Analysis

Shurui Zhou
Learning Goals

• Understand opportunities and challenges for testing quality attributes; enumerate testing strategies to help evaluate the following quality attributes: usability, reliability, security, robustness (both general and architectural), performance, integration.

• Discuss the limitations of testing

• Give a one sentence definition of static & dynamic analysis.
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
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
White box testing

Black box testing

Just a reminder...
Regression testing

• Ensure that a small change in one part of the system does not break existing functionality elsewhere in the system.
The Oracle Problem

Parameters
Input generator → SUT → Comparator → Fail → Pass
Golden standard → Comparator

Parameters
Input generator → SUT → Observer

Exception → Normal
Crash

System under test (SUT)
What are we covering?

• Program/system functionality:
  • Execution space (white box!).
  • Input or requirements space (black box!).

• The expected user experience (usability).
  • GUI testing, A/B testing

• The expected performance envelope (performance, reliability, robustness, integration).
  • Security, robustness, fuzz, and infrastructure testing.
  • Performance and reliability: soak and stress testing.
  • Integration and reliability: API/protocol testing
Quality Attributes
Performance Testing

• Specification? Oracle?
• Test harness? Environment?
• Nondeterminism?
• Unit testing?
• Automation?
• Coverage?
Specifications

• Textual
• Assertions
• Formal specifications

• JML (Java modeling language specification)

```java
debit(amount)
```
Test harness

Software system that contains test drivers, test scripts and other supporting tools that are required for the execution of any test case.

• Automation testing

• Integration Testing

Performance Testing

• Specification? Oracle?
• Test harness? Environment?
• Nondeterminism?
• Unit testing?
• Automation?
• Coverage?
Unit and regression testing for performance

• Measure execution time of critical components
• Log execution times and compare over time
Profiling

- Finding bottlenecks in execution time and memory
Profiling

• Memory profile as a function of time
Robustness: Stress Testing

• Robustness testing technique: test beyond the limits of normal operation.
• Can apply at any level of system granularity.
• Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.
Soak testing

- **Problem**: A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure

- **Soak testing**: testing a system with a significant load over a significant period of time

- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.
Reliability: Fuzz testing

• The purpose of fuzzing is to send anomalous data to a system in order to crash it, therefore revealing reliability problems.

• Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called fuzzers.

• Also known as fuzzing or monkey testing
Reliability: Fuzz testing

- Negative software testing method that feeds malformed and unexpected input data to a program, device, or system with the purpose of finding security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior
- black-box testing
Fuzzing Process

System under test (SUT)

FUZZER

VALID request

VALID response

ANOMALOUS request

ERROR response

ANOMALY sent

ANOMALOUS response

ANOMALY sent

CRASH

GET / HTTP/1.1
Accept: image/gif, image/vnd.microsoft.icon, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: keep-alive

HTTP/1.1 200 OK
Date: Wed, 07 Nov 2007 09:44:49 GMT
Server: Microsoft-IIS/6.0
Last-Modified: Wed, 07 Nov 2007 09:44:36 GMT
Content-Length: 0
Connection: close
Content-Type: text/html
charset=UTF-8

GET / HTTP/1.1
Accept: image/gif, image/vnd.microsoft.icon, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive

HTTP/1.1 404 Not Found
Date: Wed, 07 Nov 2007 09:45:27 GMT
Content-Length: 204
Connection: close
Content-Type: text/html

GET http://example.com/ HTTP/1.1
Accept: image/gif, image/vnd.microsoft.icon, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: keep-alive

HTTP/1.1 500 Internal Server Error
Date: Wed, 07 Nov 2007 09:46:30 GMT
Server: (IIS 6.0)
Content-Length: 4
Content-Type: text/html
Connection: close

********** [ 500 RESPONSE ] **********
Reliability: Fuzz testing

Performance testing tools: JMeter

http://jmeter.apache.org
Performance testing tools: Locust

An open source load testing tool.
Define user behaviour with Python code, and swarm your system with millions of simultaneous users.

https://github.com/locustio/locust

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

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

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

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

A curated list of awesome Chaos Engineering resources.

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

Your First Chaos Experiment
Use Gremlin to validate your monitoring.

https://www.youtube.com/watch?v=VUwi5Jtw3ow&feature=youtu.be
- User Acceptance Testing
- Operational Readiness Test
- Requirements Verification

- Automation Strategy
- Test Schedule
- Resource Planning

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

- Bug Tracking
- Bug Fixing
- Bug Verification

- Defects
- Test Reports
- Test Metrics
Limits of Testing

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

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
UNIVERSITY OF TORONTO
But coverage has limitations.

Are all your unit tests passing?

Yes

Do you have 100% code coverage?

Yes

Are there still going to be unexplainable bugs and reusability issues?

Yes

Low coverage means insufficient testing.
Summary

• Quality assurance is important, often underestimated
• Many forms of QA, testing popular
• Testing beyond functional correctness
Definition: software analysis

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

**Dynamic:**
- **Testing:** Direct execution of code on test data in a controlled environment.
- **Analysis:** Tools extracting data from test runs.

**Static:**
- **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
- **Analysis:** Tools reasoning about the program without executing it.
What’s a memory leak?
How can we tackle this problem?

• Testing:

• Inspection:

• Static analysis:

Wouldn’t it be nice if we could learn about the program’s memory usage as it was running?
Common dynamic analyses

• Coverage
• Performance
• Memory usage
• Security properties
• Concurrency errors
• Invariant checking
• Fault localization
• Anomaly detection
Collecting execution info

• Instrument at compile time
  • e.g., Aspects, logging, bytecode rewriting

• Run on a specialized VM
  • e.g., valgrind

• Instrument or monitor at runtime
  • also requires a special VM
  • e.g., hooking into the JVM using debugging symbols to profile/monitor (VisualVM)
Collecting execution info

- Instrument at compile time
  - e.g., Aspects, logging
- Run on a specialized VM
  - e.g., valgrind
- Instrument or monitor at runtime
  - also requires special VM
  - e.g., hooking into the JVM using debugging symbols to profile/monitor (VisualVM)

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

Note: some of these methods require a static pre processing step!
Example: Test Coverage

- **Statement:** Has each statement in the program been executed?
- **Branch:** Has each of each control structure been executed?
- **Function:** Has each function in the program been called?
- **Path:** requires that all paths through the Control Flow Graph are covered.
- ...
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?
1. `int foobar(a,b) {`
2.     `if (a > 0) {`
3.         `b -= 5;`
4.         `a -= 10;`
5.     `}`
6.     `if (a > 0) {`
7.         `if (b > 0)`
8.         `    return 1;`
9.     `}`
10.    `return 0;`
11. }`
\[
\begin{align*}
&\text{if } (a > 0) \\
&\quad \text{return 1} \\
&\quad \text{return 0}
\end{align*}
\]
if (a > 0)
    printf("1:t")
    b = 5
    a = 10

if (a > 0)
    printf("1:f")

if (b > 0)
    printf("2:t")

if (b > 0)
    printf("2:f")

return 1

return 0

(exit)
1. int foobar(a, b) {
2.     if (a > 0) {
3.         b -= 5;
4.         a -= 10;
5.     }
6.     if (a > 0) {
7.         if (b > 0)
8.             return 1;
9.     }
10. return 0;
11. }
```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;
}
```
• 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.
Common dynamic analyses

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

• Finding bottlenecks in execution time and memory
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