Architecture Recap

• “Think before implementing”
• “Secret sauce”
• 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
Administrativa: HW3

• Decompose HW1 into microservices
• Design and justify architecture
• Discuss architectural tradeoffs
Learning goals

• Define software analysis
• Distinguish validation and verification
• Understand a range of QA techniques
• Apply testing and test automation for functional correctness
• 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
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):

<table>
<thead>
<tr>
<th>Activity</th>
<th>Estimated</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing plans</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>unit testing</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>validation testing</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>test data</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
“One portion we planned for but were not able to complete to our satisfaction was testing.”
QA IS IMPORTANT (DUH!)
A software crash has occurred and Windows has been shut down.

PAGE_FAULT_IN_NONPAGED_AREA

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: 0x00000050 (0x800000f2, 0x00000000, 0x00000000, 0x00000000)

Beginning dump of physical memory
Physical memory dump complete.

Contact your system administrator or technical support group for further assistance.
Cost

Relative Cost of Software Fault Propogation

“1” Identifies Phase Defect Introduced

Relative Cost to Repair

Phase Repaired
Cost

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
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?
QA HAS MANY FACETS
What qualities are important and how can you assure them?
Cases (Continued)
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
- ...

Institute for Software Research
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.
<table>
<thead>
<tr>
<th>Error exists</th>
<th>No error exists</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Reported</strong></td>
<td><strong>False positive</strong></td>
</tr>
<tr>
<td>True positive (correct analysis result)</td>
<td></td>
</tr>
<tr>
<td><strong>No Error Reported</strong></td>
<td><strong>True negative</strong></td>
</tr>
<tr>
<td>False negative</td>
<td>True negative (correct analysis result)</td>
</tr>
</tbody>
</table>

**Sound Analysis:**
- reports all defects
- -> no false negatives
- typically overapproximated

**Complete Analysis:**
- every reported defect is an actual defect
- -> no false positives
- typically underapproximated
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...
CLASSIC TESTING
(FUNCTIONAL CORRECTNESS)
Testing

- Executing the program with selected inputs in a controlled environment (dynamic analysis)
- 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
Who’s to blame?

Algorithms.shortestDistance(graph, “Tom”, “Anne”);

> ArrayOutOfBoundsException
Specifications

• Textual
• Assertions
• Formal specifications
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
Blackbox vs Whitebox Testing
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

(Creative Commons Attribution-ShareAlike 3.0) Excirial
## Coverage Report - All Packages

<table>
<thead>
<tr>
<th>Package</th>
<th># Classes</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>55</td>
<td>75%</td>
<td>64%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.ant</td>
<td>11</td>
<td>52%</td>
<td>43%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.check</td>
<td>3</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.coveragedata</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.instrument</td>
<td>10</td>
<td>90%</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.merge</td>
<td>1</td>
<td>86%</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting</td>
<td>3</td>
<td>87%</td>
<td>88%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html</td>
<td>4</td>
<td>91%</td>
<td>77%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html.files</td>
<td>1</td>
<td>87%</td>
<td>62%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.xml</td>
<td>1</td>
<td>100%</td>
<td>95%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.util</td>
<td>9</td>
<td>60%</td>
<td>69%</td>
<td></td>
</tr>
<tr>
<td>someotherpackage</td>
<td>1</td>
<td>83%</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Report generated by [Cobertura](http://cobertura.org) 1.9 on 6/9/07 12:37 AM.
“Traditional” coverage

• Statement
• Branch
• Function
• Path (?)
• MC/DC
Continuous Integration

Automatically builds, tests, and displays the result
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 $\rightarrow$ TestResults1
    • Code2 + TestSet $\rightarrow$ 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 $<\text{TestSet},\text{TestResults1}>$ acts as one.
The Oracle Problem

Parameters

Input generator → SUT → Comparator

Golden standard

Fail

Pass

Parameters

Input generator → SUT → Observer

Exception → Crash

Parameters

Input generator → SUT → SUT

Assertion

Pass

Fails
TESTING BEYOND FUNCTIONAL CORRECTNESS
Testing Relevant Qualities
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?
TESTING PERFORMANCE
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
Performance testing tools: JMeter

http://jmeter.apache.org
Profiling

• Finding bottlenecks in execution time and memory
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.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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                                                                                       |
TESTING USABILITY
Usability Testing

• Specification?
• Test harness? Environment?
• Nondeterminism?
• Unit testing?
• Automation?
• Coverage?
What is testing?

• **Direct execution of code on test data in a controlled environment (dynamic analysis)**

• 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
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.

<table>
<thead>
<tr>
<th>Create</th>
<th>Put</th>
<th>Put</th>
<th>Get</th>
<th>Get</th>
<th>Get</th>
<th>Get</th>
<th>Put</th>
<th>Put</th>
<th>Get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Get</td>
<td>Get</td>
<td>Put</td>
<td>Get</td>
<td>Put</td>
<td>Get</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Approach
  – Develop representative sequences – based on use cases, scenarios, profiles
  – Randomly generate call sequences

• Also useful for protocol interactions within distributed designs.
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)
Manual Testing?

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

**Generic test case: user sends MMS with picture attached.**

<table>
<thead>
<tr>
<th>Step ID</th>
<th>User Action</th>
<th>System Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Go to Main Menu</td>
<td>Main Menu appears</td>
</tr>
<tr>
<td>2</td>
<td>Go to Messages Menu</td>
<td>Message Menu appears</td>
</tr>
<tr>
<td>3</td>
<td>Select “Create new Message”</td>
<td>Message Editor screen opens</td>
</tr>
<tr>
<td>4</td>
<td>Add Recipient</td>
<td>Recipient is added</td>
</tr>
<tr>
<td>5</td>
<td>Select “Insert Picture”</td>
<td>Insert Picture Menu opens</td>
</tr>
<tr>
<td>6</td>
<td>Select Picture</td>
<td>Picture is Selected</td>
</tr>
<tr>
<td>7</td>
<td>Select “Send Message”</td>
<td>Message is correctly sent</td>
</tr>
</tbody>
</table>
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
**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.
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.
We can measure coverage on almost anything
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.
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
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
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
• ...

...
Summary

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