Lecture 11: System-Level Testing

Gregory Gay
TDA594 - December 8, 2020
Today’s Goals

• Discuss testing at the system level.
  • UI and Integration versus Unit Testing.

• Introduce process for creating System-Level Tests.
  • Identify Independently Testable Functionality
  • Identify Representative Values
  • Generate Test Case Specifications
  • Generate Concrete Test Cases
Software Testing

- An investigation into system quality.
- Based on sequences of **stimuli** and **observations**.
  - **Stimuli** that the system must react to.
  - **Observations** of system reactions.
  - **Verdicts** on correctness.
Axiom of Testing

“Program testing can be used to show the presence of bugs, but never their absence.”

- Dijkstra
Anatomy of a Test Case

Test Inputs
How we “stimulate” the system (method call, API request, GUI event).

Test Oracle
How we check the correctness of the resulting observation (assertions).

if $O_n = \text{Expected}(O_n)$
then … Pass
else … Fail
Anatomy of a Test Case

- **Initialization**
  - Any steps that must be taken before test execution.

- **Test Steps**
  - Interactions with the system, and comparisons between expected and actual values.

- **Tear Down**
  - Any steps that must be taken after test execution.
Testing Stages

- We interact with **systems** through **interfaces**.
- Systems built from **subsystems**.
  - With their own interfaces.
- Subsystems built from **units**.
  - Classes work with other classes through methods (interfaces).
Testing Stages

• Unit Testing
  • Do the methods of a class work?

• System Testing
  • Subsystem Integration Testing
    • Do the collected units work?
  • System Integration Testing
    • Do the collected subsystems work?

• UI Testing
  • Does interaction through UIs work?
Unit Testing

• Testing the smallest “unit” that can be tested.
  • Often, a class and its methods.
• Tested in isolation from all other units.
  • Mock the results from other classes.
• Test input = method calls.
• Test oracle = assertions on output/class variables.
Unit Testing

For a unit, tests should:

- Test all “jobs” associated with the unit.
  - Individual methods belonging to a class.
  - Sequences of methods that can interact.
- Set and check value of all class variables.
  - Examine how variables change after method calls.
  - Put the variables into all possible states (types of values).
Unit Testing - WeatherStation

Unit tests should cover:

- Set and check class variables.
  - Can any methods change identifier, temperature, pressure?

- Each “job” performed by the class.
  - Single methods or method sequences.
  - Vary the order methods are called.
  - Each outcome of each “job” (error handling, return conditions).

<table>
<thead>
<tr>
<th>WeatherStation</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
<tr>
<td>temperature</td>
</tr>
<tr>
<td>pressure</td>
</tr>
<tr>
<td>checkLink()</td>
</tr>
<tr>
<td>reportWeather()</td>
</tr>
<tr>
<td>reportInstrumentStatus()</td>
</tr>
<tr>
<td>restart(instrumentName)</td>
</tr>
<tr>
<td>shutdown(instrumentName)</td>
</tr>
<tr>
<td>reconfigure(instrumentName, commands)</td>
</tr>
</tbody>
</table>
Writing a Unit Test

Java-based unit testing (JUnit).

• Choose a target unit.
  • Ex. Calculator to right.
• Create a test class.
  • Unit tests are methods marked with @Test.

```java
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand : expression.split("\\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
```
JUnit Test Skeleton

@Test annotation defines a single test:

```java
@Test
public void test<Feature or Method Name>_<Testing Context>() {
    //Define Inputs
    try{ //Try to get output.
        }catch(Exception error){
            fail("Why did it fail?");
    }
    //Compare expected and actual values through assertions or through
    //if-statements/fail commands
}
```

Type of scenario, and expectation on outcome.
I.e., testEvaluate_GoodInput() or testEvaluate_NullInput()
Writing JUnit Tests

```java
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\+")) {
            sum += Integer.valueOf(summand);
        }
        return sum;
    }
}
```

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
    @Test
    void testEvaluate_Valid_ShouldPass() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
        calculator = null;
    }
}
```

**Convention** - name the test class after the class it is testing or the functionality being tested.

Each test is denoted with keyword `@test`.

- **Initialization**
- **Test Steps**
- **Input**
- **Oracle**
- **Tear Down**
Integration Testing

• After testing units, test their integration.
  • Integrate units in one subsystem.
  • Then integrate the subsystems.

• Test input through a defined interface.
  • Focus on showing that functionality accessed through interfaces is correct.
  • Subsystems: “Top-Level” Class, API
  • System: API, GUI, CLI, …
Integration Testing

Subsystem made up classes of A, B, and C. We have performed unit testing...

- Classes work together to perform subsystem functions.
- Tests applied to the interface of the subsystem they form.
- Errors in combined behavior not caught by unit testing.
Interface Errors

• Interface Misuse
  • Malformed data, order, number of parameters.

• Interface Misunderstanding
  • Incorrect assumptions made about called component.
  • A binary search called with an unordered array.

• Timing Errors
  • Producer of data and consumer of data access data in the wrong order.
Testing Percentages

- Unit tests verify behavior of a single class.
  - 70% of your tests.
- Integration tests verify class interactions in a portion of the app.
  - 20% of your tests.
- UI tests verify end-to-end journey over the app.
  - 10% of your tests.
Testing

• 70/20/10 recommended.
• Unit tests execute quickly, without emulator or devices.
• UI tests must run in Android, are very slow.
• Well-tested units reduce likelihood of integration issues, making high levels of testing easier.
Writing Integration and UI Tests

• Testing framework depends on language and interface type.
  • Android: JUnit (Integration - AndroidX, UI - Espresso)
  • RESTful API: Postman
  • Browser-based GUI: Selenium
Android UI Test

@Test
public void successfulLogin() {
    LoginActivity activity = ActivityScenario.launch(LoginActivity.class);
    onView(withId(R.id.user_name)).perform(typeText("test_user"));
    onView(withId(R.id.password)).perform(typeText("correct_password"));
    onView(withId(R.id.button)).perform(click());
    assertThat(getIntents().first()).hasComponentClass(HomeActivity.class);
}

Uses Espresso testing libraries to interact with Views and Intents. (Part of AndroidX)
RESTful API Test - Postman

Test Step + Input

Test Oracle

```javascript
pm.test("Status test", function () {
    pm.response.to.have.status(200);

    pm.response.to.not.be.error;
    pm.response.to.have.jsonBody("";
    pm.response.to.not.have.jsonBody("error");
});
```
System-Level Tests and SPLs

• Variability is a *system-level concept*.
  • Feature options tend to be entire classes or subsystems.

• **Unit testing during domain engineering.**
  • Assets tested in isolation.

• Many interaction errors between features, depending on chosen options.
  • System testing during application engineering.
Creating System-Level Test Cases
Creating System-Level Tests

1. Identify Independently Testable Functionality
2. Identify Choices
3. Identify Representative Input Values
4. Generate Test Case Specifications
5. Generate Test Cases

- Identify functionality that can be tested in (relative) isolation.
- Identify the choices you control when testing.
- Identify values for each choice that lead to different function outcomes.
- Identify abstract test cases based on choice combinations.
- Identify concrete input/expected output pairs.
Independently Testable Functionality

• A well-defined function that can be tested in (relative) isolation.
  • Based on the “verbs” - what can we do with this system?
  • The high-level functionality offered by an interface.
  • UI - look for user-visible functions.
    • Web Forum: Sorted User List can be accessed.
    • Accessing the list is a testable functionality.
    • Sorting the list is not (low-level, unit testing target)
Units and “Functionality”

• Many tests written in terms of “units” of code.
• An independently testable function is a capability of the software.
  • Can be at class, subsystem, or system level.
  • Defined by an interface.
Identify the Choices

• What choices do we make when using a function?
  • Anything we control when we test.
• What are the inputs to that feature?
• What choices did we make for variation points?
• Are there environmental factors we can vary?
  • Networking environment, file existence, file content, database connection, database contents, disk utilization, …
Ex: Register for Website

• What are the inputs to that feature?
  • (first name, last name, date of birth, e-mail)
• Website is part of product line with different database options.
  • (database type)
• Consider implicit environmental factors.
  • (database connection, user already in database)
Parameter Characteristics

• Identify choices by understanding how parameters are used by the function.
• Type information is helpful.
  • `firstName` is string, database contains UserRecords.
• … but context is important.
  • Reject registration if in database.
  • … or database is full.
  • … or database connection down.
Parameter Context

- Input parameter split into multiple “choices” based on contextual use.
  - “Database” is an implicit input for User Registration, but it is not *one* choice.
  - “Database Connection Status”, “User Record in Database”, “Percent of Database Filled” influence function outcome.
    - The Database input results in three choices.
    - Test cases will be based on these choices.
Examples

Class Registration System

What are some independently testable functions?

• Register for class
• Drop class
• Transfer credits from another university
• Apply for degree
Example - Register for a Class

What are the choices?

• Course number to add
• Student record
• What about a course database? Student record database?
• What else influences the outcome?
Example - Register for a Class

• Student Record is an implicit input.
• How is it used?
  • Have you already taken the course?
  • Do you meet the prerequisites?
  • What university are you registered at?
  • Can you take classes at the university the course is offered at?
Example - Register for a Class

• Choices:
  • Course to Add
  • Does course exist?
  • Does student record exist?
  • Has student taken the course?
  • Which university is student registered at?
  • Is course at a valid university for the student?
  • Can student record be retrieved from database?
  • Does the course exist?
  • Does student meet the prerequisites?
Let’s take a break.
Identifying Representative Values

• We know the functions.
• We have a set of choices.
• What values should we try? • For some choices, finite set.
  • For many, near-infinite set.
• What about exhaustively trying all options?
Exhaustive Testing

Take the arithmetic function for the calculator:
\[ \text{add}(\text{int } a, \text{ int } b) \]

- How long would it take to exhaustively test this function?

\[ 2^{32} \text{ possible integer values for each parameter.} \]
\[ = 2^{32} \times 2^{32} = 2^{64} \]
\[ \text{combinations} = 10^{13} \text{ tests.} \]

1 test per nanosecond
\[ = 10^{5} \text{ tests per second} \]
\[ = 10^{10} \text{ seconds} \]

or... about 600 years!
Not all Inputs are Created Equal

- Many inputs lead to same outcome.
- Some inputs better at revealing faults.
  - We can’t know which in advance.
  - Tests with different input better than tests with similar input.

Identify Representative Input Values
Random Testing

- Pick inputs uniformly from the distribution of all inputs.
- All inputs considered equal.
- Keep trying until out of time.
- No designer bias.
- Removes manual tedium.
Why Not Random?
Input Partitioning

• Consider possible values for a variable.
• Faults sparse in space of all inputs, but dense in parts where they appear.
  • Similar input to failing input also likely to fail.
• Try input from partitions, hit dense fault space.
Equivalence Class

• Divide the input domain into equivalence classes.
  • Inputs from a group interchangeable (trigger same outcome, result in the same behavior, etc.).
  • If one input reveals a fault, others in this class (probably) will too. In one input does not reveal a fault, the other ones (probably) will not either.

• Partitioning based on intuition, experience, and common sense.
Example

`substr(string str, int index)`

What are some possible partitions?

- index < 0
- index = 0
- index > 0
- str with length < index
- str with length = index
- str with length > index
- ...
Choosing Input Partitions

• Equivalent output events.
• Ranges of numbers or values.
• Membership in a logical group.
• Time-dependent equivalence classes.
• Equivalent operating environments.
• Data structures.
• Partition boundary conditions.
Look for Equivalent Outcomes

• Look at the outcomes and group input by the outcomes they trigger.

• Example: `getEmployeeStatus(employeeID)`
  • Outcomes include: Manager, Developer, Marketer, Lawyer, Employee Does Not Exist, Malformed ID
  • Abstract values for choice `employeeID`.
    • Can potentially break down further.
Look for Ranges of Values

• Divide based on data type and how variable used.
  • Ex: Integer input. Intended to be 5-digit:
    • < 10000, 10000-99999, >= 100000
    • Other options: < 0, 0, max int
    • Can you pass it something non-numeric? Null pointer?

• Try “expected” values and potential error cases.
Look for Membership in a Group

Consider the following inputs to a program:

- A floor layout
- A country name.
- All can be partitioned into groups.
  - Apartment vs Business, Europe vs Asia, etc.
- Many groups can be subdivided further.
- Look for context that an input is used in.
Timing Partitions

- Timing and duration of an input may be as important as the value.
  - Timing often implicit input.
    - Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
    - Close program before, during, and after the program is writing to (or reading from) a disc.
Operating Environments

- Environment may affect behavior of the program.
- Environmental factors can be partitioned.
  - Memory may affect the program.
  - Processor speed and architecture.
- Client-Server Environment
  - No clients, some clients, many clients
  - Network latency
  - Communication protocols (SSH vs HTTPS)
Data Structures

• Data structures are prone to certain types of errors.
• For arrays or lists:
  • Only a single value.
  • Different sizes and number filled.
  • Order of elements: access first, middle, and last elements.
Input Partition Example

What are the input partitions for:
\[ \text{max}(\text{int } a, \text{ int } b) \text{ returns (int } c) \]

We could consider \( a \) or \( b \) in isolation:
\[ a < 0, \ a = 0, \ a > 0 \]

Consider combinations of \( a \) and \( b \) that change outcome:
\[ a > b, \ a < b, \ a = b \]
Revisit the Roadmap

For each independently testable function, we want to:
1. Partition each choice into representative values.
2. Choose one partition for each choice to form a complete abstract test specification.
3. Assigning concrete values from each partition.
Forming Specification

Function insertPostalCode(int N, list A).

- Partition choices into equivalence classes.
  - int N is a 5-digit integer between 10000 and 99999.
    - Possible partitions: <10000, 10000-99999, >100000
  - list A is a list of length 1-10.
    - Possible partitions: Empty List, List of Length 1, List Length 2-10, List of Length > 10
From Partitions to Test Case

Choose concrete values for each combination of input partitions:

\[
\text{insertPostalCode}(\text{int } N, \text{list } A)
\]

\[
\begin{align*}
\text{int } N & \\
< 10000 & \\
10000 - 99999 & \\
> 99999 & \\
\text{list } A & \\
\text{Empty List} & \\
\text{List}[1] & \\
\text{List}[2-10] & \\
\text{List}[>10] & 
\end{align*}
\]

Test Specifications:

\[
\begin{align*}
\text{insert}(< 10000, \text{ Empty List}) \\
\text{insert}(10000 - 99999, \text{ list}[1]) \\
\text{insert}(> 99999, \text{ list}[2-10]) \\
\end{align*}
\]

... (3 * 4 = 12 abstract specifications)

Test Cases:

\[
\begin{align*}
\text{insert}(5000, \{\}) \\
\text{insert}(96521, \{11123\}) \\
\text{insert}(150000, \{11123, 98765\}) \\
\end{align*}
\]

... (Each specification = 1000s of potential test cases)
Generate Test Cases

substr(string str, int index)

Specification:
str: length \( \geq 2 \), contains special characters
index: value > 0

Test Case:
str = “ABCC!\n\t7”
index = 5
Boundary Values

• Errors tend to occur at the boundary of a partition.
• Remember to select inputs from those boundaries.
Choosing Test Case Values

Choose test case values at the boundary (and typical) values for each partition.

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

\(<10000, 10000-99999, >100000\)
Activity - System-Level Testing

- Microservice related to Sets:
  - void insert(Set set, Object obj)
  - Boolean find(Set set, Object obj)
  - void delete(Set set, Object obj)

- For each function, identify choices.

- For each choice, identify the representative values.

- Create abstract test specifications with expected outcomes.
Activity - System-Level Testing

• `insert(Set set, Object obj)`
  • Choices *(Number of Items)* and *(Object Status)*.

• One test specification might be:
  • **Input:** Set with One Item/Object Already in Set
  • **Expected output:** Object not Added
  • You can omit redundant test specifications.
Solution - Choices and Values

• (Number of Items in Set)
  • Empty
  • 1
  • 2 +
  • 100 + (may be slower - make sure it still works)

• (Object Status)
  • In set already
  • Not in set
  • Null pointer
# Solution - Test Specifications

<table>
<thead>
<tr>
<th>Insert</th>
<th>Empty/ Object not in Set</th>
<th>One element / Object not in Set</th>
<th>Multiple elements / Object not in Set</th>
<th>100+ / Object not in Set</th>
<th>(any choice) / Object in Set</th>
<th>(any choice) / Null Object</th>
<th>Exists</th>
<th>One element / Object in Set</th>
<th>Empty / Object not in Set</th>
<th>100 + / Object in Set</th>
<th>100 + / Object not in Set</th>
<th>(any choice) / Null Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>obj in container</td>
<td>obj in container</td>
<td>obj in container</td>
<td>obj in container</td>
<td>Error or no change</td>
<td>Error</td>
<td>True</td>
<td>obj no longer in set</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>One element / Object in Set</td>
<td>obj no longer in set</td>
<td>One element / Object not in Set</td>
<td>no change (or error)</td>
<td>Error</td>
<td>100 + / Object not in Set</td>
<td>obj no longer in set</td>
<td>Empty / Object not in Set</td>
<td>no change (or error)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table is used to test the behavior of an object insertion and deletion process in a set context, with various input scenarios and expected outcomes.
We Have Learned

• Unit testing centered around a single class.
• System-level tests centered around integration of components, through an interface.
  • Identifying independently testable functionality.
  • Identify choices that influence function outcome.
  • Partitioning choices into representative values.
  • Combining choice values into test specifications.
  • Choosing concrete values for specifications.
Next Time

• System-level testing and feature interactions
  • Handling infeasible combinations.
  • Selecting a valid subset of representative values.

• Assignment 3 due tomorrow!
  • Questions?
  • Assignment 4 posted. Due December 20.