Today’s Goals

• Discuss testing at the system level.
  • System (Integration) Testing versus Unit Testing.

• Introduce process for creating System-Level Tests.
  • Identify Independently Testable Functionality
  • Identify Choices (AKA variation points)
  • Identify Representative Values for each Choice
  • 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.

![Software Testing Diagram]

- Test Input
- SUT
- Output
- Test Oracle (Expected Output)
- Verdict (Pass/Fail)
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.
  • APIs, GUIs, CLIs

• Systems built from subsystems.
  • With their own interfaces.

• Subsystems built from units.
  • Communication via method calls.
  • Set of methods is an interface.
Testing Stages

- **Unit Testing**
  - Do the methods of a class work?

- **System-level Testing**
  - **System (Integration) Testing**
    - (Subsystem-level) Do the collected units work?
    - (System-level) Does high-level interaction through APIs/UIs work?
  - **Exploratory Testing**
    - Does interaction through GUIs 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 class variables.
    - Examine how variables change after method calls.
    - Put the variables into all possible states (types of values).

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- personnummer</td>
</tr>
<tr>
<td>- balance</td>
</tr>
<tr>
<td>Account (name, personnummer, Balance)</td>
</tr>
<tr>
<td>withdraw (double amount)</td>
</tr>
<tr>
<td>deposit (double amount)</td>
</tr>
<tr>
<td>changeName(String name)</td>
</tr>
<tr>
<td>getName()</td>
</tr>
<tr>
<td>getPersonnummer()</td>
</tr>
<tr>
<td>getBalance()</td>
</tr>
</tbody>
</table>
Unit Testing - Account

Some tests we might want to write:

- Execute constructor, verify fields.
- Check the name, change the name, make sure changed name is in place.
- Check that personnummer is correct.
- Check the balance, withdraw money, verify that new balance is correct.
- Check the balance, deposit money, verify that new balance is correct.

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- personnummer</td>
</tr>
<tr>
<td>- balance</td>
</tr>
<tr>
<td>Account (name, personnummer, Balance)</td>
</tr>
<tr>
<td>withdraw (double amount)</td>
</tr>
<tr>
<td>deposit (double amount)</td>
</tr>
<tr>
<td>changeName(String name)</td>
</tr>
<tr>
<td>getName()</td>
</tr>
<tr>
<td>getPersonnummer()</td>
</tr>
<tr>
<td>getBalance()</td>
</tr>
</tbody>
</table>
Unit Testing - Account

Some potential error cases:

- Withdraw more than is in balance.
- Withdraw a negative amount.
- Deposit a negative amount.
- Withdraw/Deposit a small amount (potential rounding error)
- Change name to a null reference.
- Can we set an “malformed” name?
  - (i.e., are there any rules on a valid name?)
Unit Testing - Account

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- personnummer</td>
</tr>
<tr>
<td>- balance</td>
</tr>
</tbody>
</table>

Account (name, personnummer, Balance)
withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

- Withdraw money, verify balance.

Each test is denoted with keyword `@test`.
Name based on type of scenario, and expectation on outcome.

```java
public void testWithdraw_normal() {
    // Setup
    Account account = new Account(“Test McTest”, “19850101-1001”, 48.5);
    // Test Steps
    double toWithdraw = 16.0; //Input
    account.withdraw(toWithdraw);
    double actual = account.getBalance();
    double expectedBalance = 32.5; // Oracle
    assertEquals(expected, actual); // Oracle
}
```
• Withdraw a negative amount.
  • (should throw an exception with appropriate error message)

```java
@Test
public void testWithdraw_negative() {
    // Setup
    Account account = new Account("Test McTest", "19850101-1001", 48.5);
    // Test Steps
    double toWithdraw = -2.5; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals("Cannot withdraw a negative amount: -2.50", exception.getMessage()); // Oracle
}
```
System Testing

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

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

• Unit tests focus on a **single class**.
  • Simple functionality, more freedom.
  • Few method calls.

• **System tests** bring **many classes** together.
  • Focus on testing through an interface.
  • One interface call triggers many internal calls.
    • Slower test execution.
  • May have complex input and setup.
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.
- System tests verify class interactions.
  - 20% of your tests.
- GUI tests verify end-to-end journeys.
  - 10% of your tests.
Testing

• 70/20/10 recommended.
• Unit tests execute quickly, relatively simple.
• System tests more complex, require more setup, slower to execute.
• UI tests very slow, may require humans.
• 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
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 an Independently Testable Function
2. Identify Choices
3. Identify Representative Input Values
4. Generate Test Case Specifications
5. Generate Test Cases

- Identify a function that can be tested in (relative) isolation.
- Identify controllable aspects of the input and environment that determine the outcome of the function.
- Identify types of values for each choice that lead to different function outcomes.
- Combine values to form “recipes” for test cases.
- Replace representative values with concrete values.
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 Input Choices

• What choices do we make when using a function?
  • Anything we control that can change the outcome.
• What are the *inputs* to that feature?
• What *configuration choices* can we make?
• 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 leads to **more than one** choice.
  • “Database Connection Status”, “User Record in Database”, “Percent of Database Filled” influence function outcome.
    • The Database “input” results in three input choices when we design test cases.
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 we make when we design a test case?

• 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 choice.
• 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

- Potential Test 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?

<table>
<thead>
<tr>
<th>2(^{32}) possible integer values for each parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = 2^{32} \times 2^{32} = 2^{64} )</td>
</tr>
<tr>
<td>combinations = 10(^{13}) tests.</td>
</tr>
</tbody>
</table>

1 test per nanosecond

\( = 10^{5} \) tests per second

\( = 10^{10} \) 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.
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:
max(int a, int b) 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 testing choice for a function, we want to:
1. Partition each choice into representative values.
2. Choose a value for each choice to form a test specification.
3. Assigning concrete values from each partition.
Forming Specification

Function `insertPostalCode(int N, list A)`.

- **Choice:** `int N`
  - 5-digit integer between 10000 and 99999
  - **Representative Values:** `<10000, 10000-99999, >100000`

- **Choice:** `list A`
  - list of length 1-10
  - **Representative Values:** Empty List, List of Length 1, List Length 2-10, List of Length > 10
Forming Specifications

Choose concrete values for each combination of input partitions:

```
insertPostalCode(int N, list A)
```

- **int N**
  - < 10000
  - 10000 - 99999
  - > 99999

- **list A**
  - Empty List
  - List[1]
  - List[2-10]
  - List[>10]

### Test Specifications:

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

```
insert(< 10000, Empty List)
insert(10000 - 99999, list[1])
insert(> 99999, list[2-10])
```

...  

### Concrete Test Cases:

```
(Each specification = 1000s of potential test cases)
```

```
insert(5000, { })
insert(96521, {11123})
insert(150000, {11123, 98765})
```

...
Generate Test Cases

substr(string str, int index)

Specification:
str: length >=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.
Boundary 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
Example - Set Microservice

- 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 representative values.
- Create test specifications with expected outcomes.
Example - Set Microservice

```java
void insert(Set set, Object obj)
```

- What are our choices?

  - **Parameter:** `set`
    - **Choice 1:** Number of items in the set
  - **Parameter:** `obj`
    - **Choice 2:** Is `obj` already in the set?
    - **Choice 3:** Type of `obj` (e.g., valid, invalid, null)
Example - Set Microservice

```java
void insert(Set set, Object obj)
```

Parameter: set

- **Choice:** Number of items in the set
  - Representative Values:
    - Empty Set
    - Set with 1 item
    - Set with 10 items
    - Set with 10000 items

Parameter: obj

- **Choice:** Is obj already in the set?
  - Representative Values:
    - obj already in set
    - obj not in set

- **Choice:** Type of obj
  - Representative Values:
    - Valid obj
    - Null obj
Example - Set Microservice

void insert(Set set, Object obj)

- (4 * 2 * 2) = 16 specifications
  - Some are invalid (null in set). Can remove/ignore those.
  - Each can become 1+ test cases.

<table>
<thead>
<tr>
<th>Set Size</th>
<th>Obj in Set</th>
<th>Obj Status</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>No</td>
<td>Valid</td>
<td>Obj added to Set</td>
</tr>
<tr>
<td>Empty</td>
<td>No</td>
<td>Null</td>
<td>Error or no change</td>
</tr>
<tr>
<td>1 item</td>
<td>Yes</td>
<td>Valid</td>
<td>Error or no change</td>
</tr>
<tr>
<td>1 item</td>
<td>No</td>
<td>Valid</td>
<td>Obj added to Set</td>
</tr>
<tr>
<td>1 item</td>
<td>No</td>
<td>Null</td>
<td>Error or no Change</td>
</tr>
<tr>
<td>10 items</td>
<td>Yes</td>
<td>Valid</td>
<td>Error or no change</td>
</tr>
<tr>
<td>10 items</td>
<td>No</td>
<td>Valid</td>
<td>Obj added to Set</td>
</tr>
<tr>
<td>10 items</td>
<td>No</td>
<td>Null</td>
<td>Error or no Change</td>
</tr>
<tr>
<td>10000</td>
<td>Yes</td>
<td>Valid</td>
<td>Error or no change (may be slowdown)</td>
</tr>
<tr>
<td>10000</td>
<td>No</td>
<td>Valid</td>
<td>Obj added to Set (may be slowdown)</td>
</tr>
<tr>
<td>10000</td>
<td>No</td>
<td>Null</td>
<td>Error or no Change (may be slowdown)</td>
</tr>
</tbody>
</table>
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 microservice, identify choices.

● For each choice, identify the representative values.

● Create four abstract test specifications with expected outcomes.
# Solution - Test Specifications

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

<table>
<thead>
<tr>
<th>Exists</th>
<th>One element / Object in Set</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Empty / Object not in Set</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>100 + / Object in Set</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>100 + / Object not in Set</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>(any choice) / Null Object</td>
<td>Error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delete</th>
<th>One element / Object in Set</th>
<th>obj no longer in set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One element / Object not in Set</td>
<td>no change (or error)</td>
</tr>
<tr>
<td></td>
<td>(any choice) / Null Pointer</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>100 + / Object in Set</td>
<td>obj no longer in set</td>
</tr>
<tr>
<td></td>
<td>Empty / Object not in Set</td>
<td>no change (or error)</td>
</tr>
</tbody>
</table>
We Have Learned

• Unit testing focus on a single class.
• System tests focus on high-level functionality, integrating low-level components through a UI/API.
  • Identify an independently testable function.
  • Identify choices that influence function outcome.
  • Partition choices into representative values.
  • Form specifications by choosing a value for each choice.
  • Turn specifications into concrete test cases.
Next Time

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

• Assignment 4 - Dec 12
  • Any questions?