CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 10: System-Level Testing

Gregory Gay
TDA 594/DIT 593 - December 1, 2022

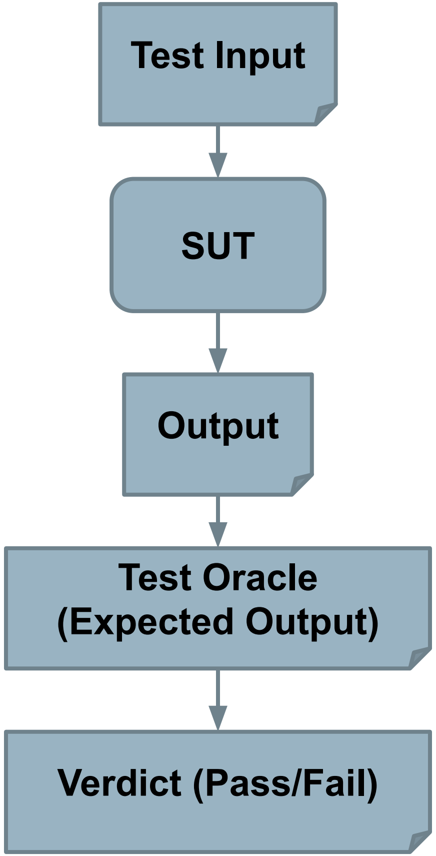
Today’s Goals

- Introduce software testing.
- Introduce process for creating test cases.
  - Identify Independently Testable Functionality
  - Identify Choices (AKA variation points)
  - Identify Representative Values for each Choice
  - Generate Test Case Specifications
  - Generate Concrete Test Cases

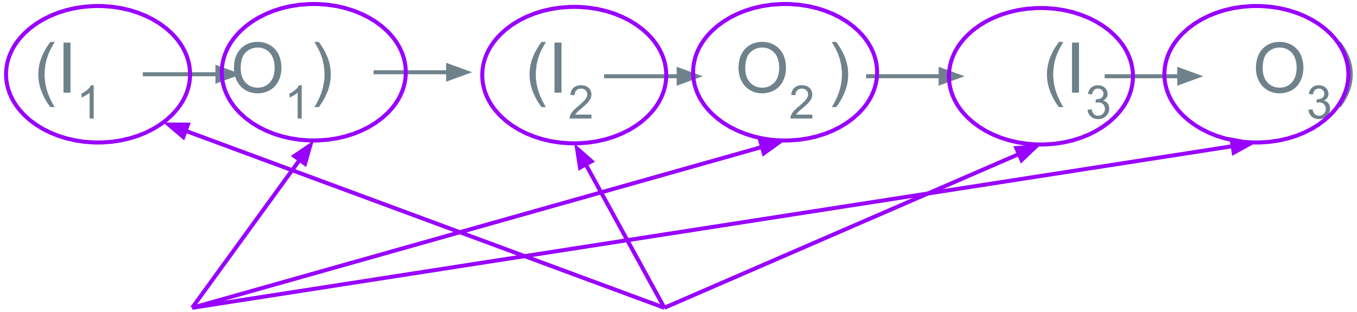
Testing Fundamentals and Test Case Structure

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.



Anatomy of a Test Case



$(I_1 \rightarrow O_1) \rightarrow (I_2 \rightarrow O_2) \rightarrow (I_3 \rightarrow O_3)$

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 = 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.

JUnit Test Cases

JUnit is a Java-based toolkit for writing executable tests.

- Create “testing class” centered around a common target or theme.
- Test cases written as methods.

```
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:

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

```
@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
}
```

Writing JUnit Tests

Convention - name the test class after the class it is testing.

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

Each test is denoted with keyword **@test**.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  void testEvaluate_Valid_ShouldPass(){
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Initialization

Test Steps

Input

Oracle

Test Fixtures - Shared Initialization

@BeforeEach annotation defines a common test initialization method:

```
@BeforeEach
public void setUp() throws Exception
{
   this.registration = new Registration();
   this.registration.setUser(“ggay”);
}
```

Test Fixtures - Teardown Method

@AfterEach annotation defines a common test tear down method:

```
@AfterEach
public void tearDown() throws Exception
{
    this.registration.logout();
    this.registration = null;
}
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

- `assertEquals, assertArrayEquals`
- `assertFalse, assertTrue`
- `assertNull, assertNotNull`
- `assertSame,assertNotSame`

assertEquals

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not
equal", "text", "text");
}

@Test
public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays
not same", expected, actual);
}
```

- Compares two items for equality.
- For user-defined classes, relies on `.equals` method.
  - Compare field-by-field
  - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`
- assertArrayEquals compares arrays of items.

Testing Exceptions

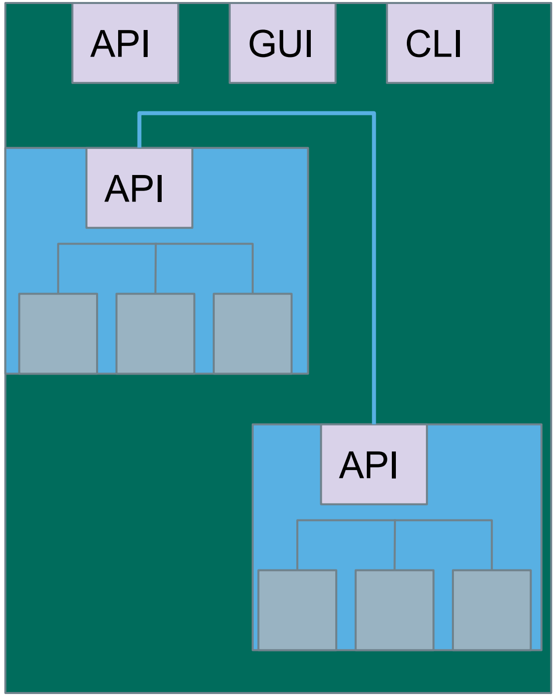
```
@Test
void exceptionTesting() {
  Throwable exception =
    assertThrows(
      IndexOutOfBoundsException.class,
      () -> { new ArrayList<Object>().get(0);}
    );
    assertEquals("Index:0, Size:0",
      exception.getMessage());
}
```

- When testing error handling, we expect exceptions to be thrown.
    - **assertThrows** checks whether the code block throws the expected exception.
    - **assertEquals** can be used to check the contents of the stack trace.

Levels of Granularity

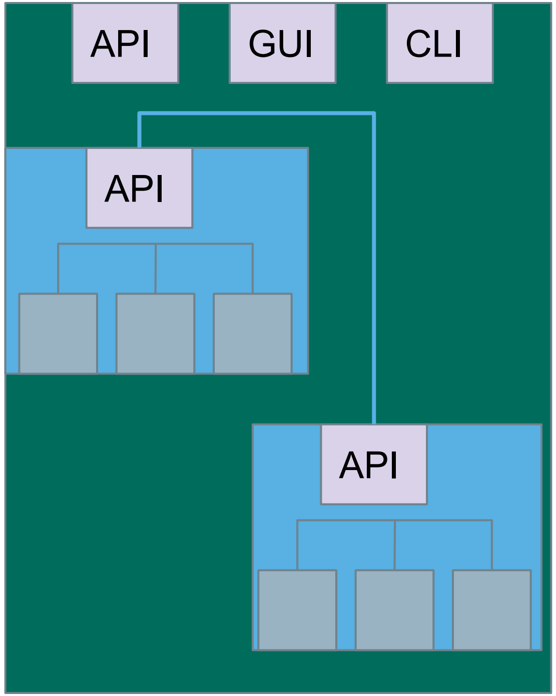
Testing Stages

- 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?

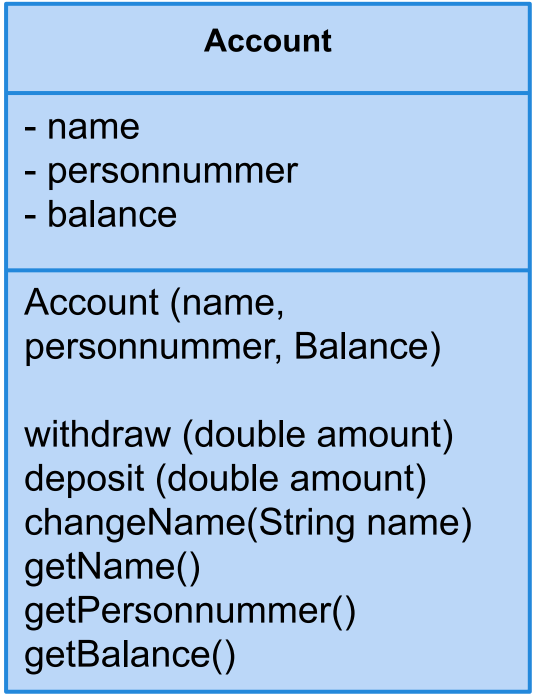


Unit Testing

- 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).



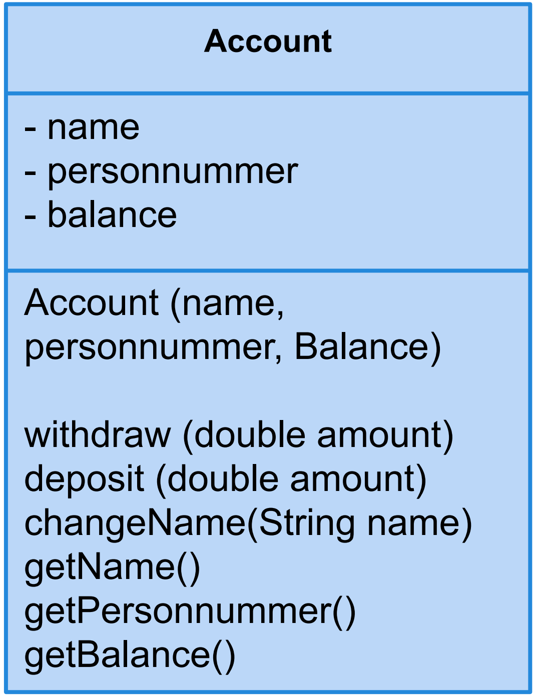
Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

Unit Testing - Account



Account

- name
- personnummer
- balance

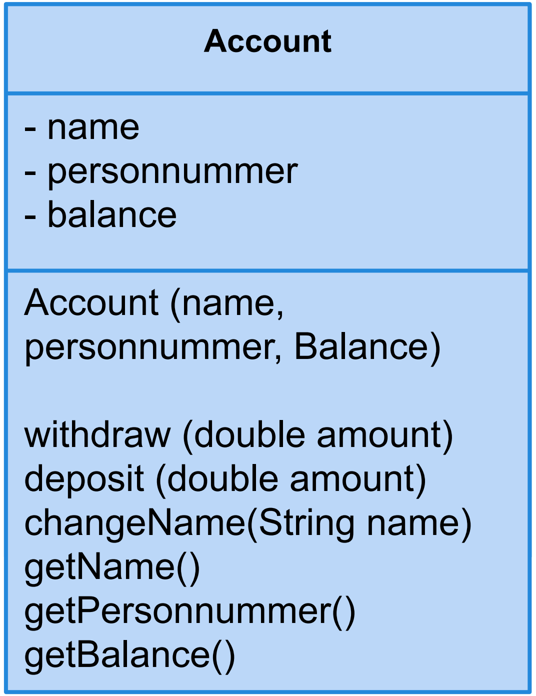
Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

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.

Unit Testing - Account



Account

- name
- personnummer
- balance

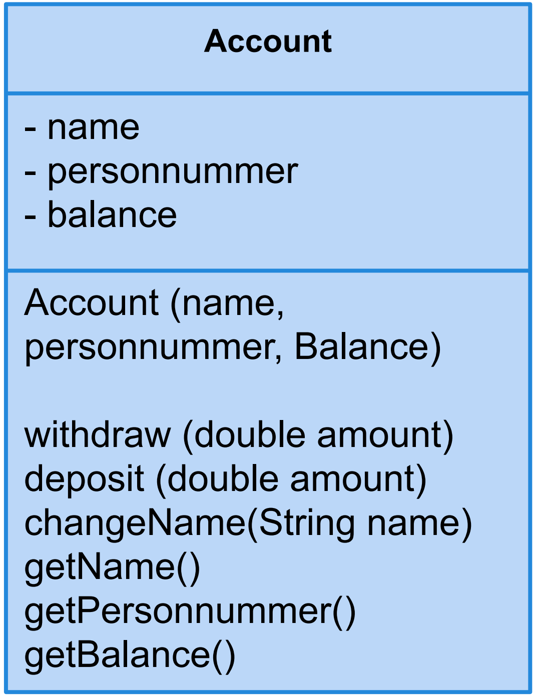
Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

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



Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

- Withdraw money, verify balance.

Each test is

Name based on type of scenario, and expectation on outcome.

```
@Test
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
}
```

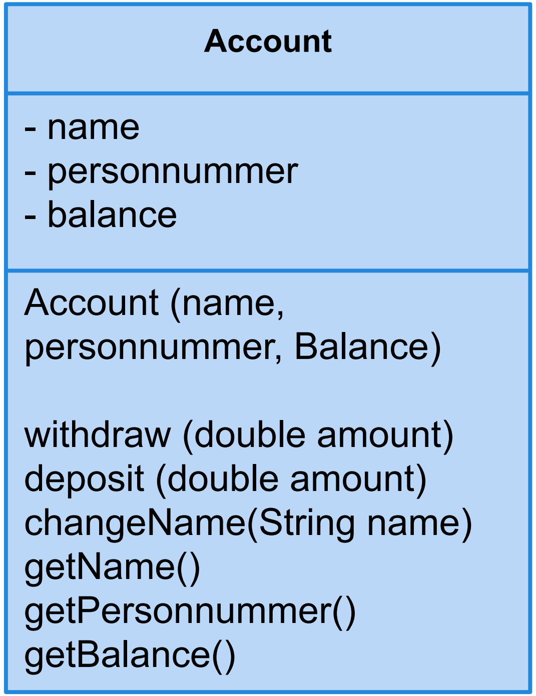
Initialization

Input

Test Steps

Oracle

Unit Testing - Account



- Withdraw a negative amount.
  - (should throw an exception with appropriate error message)

```
@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
}
```

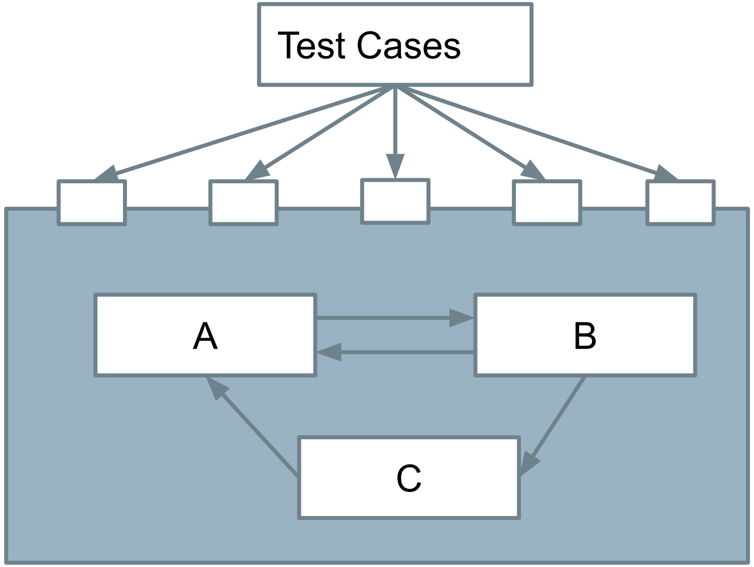
System Testing

- 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

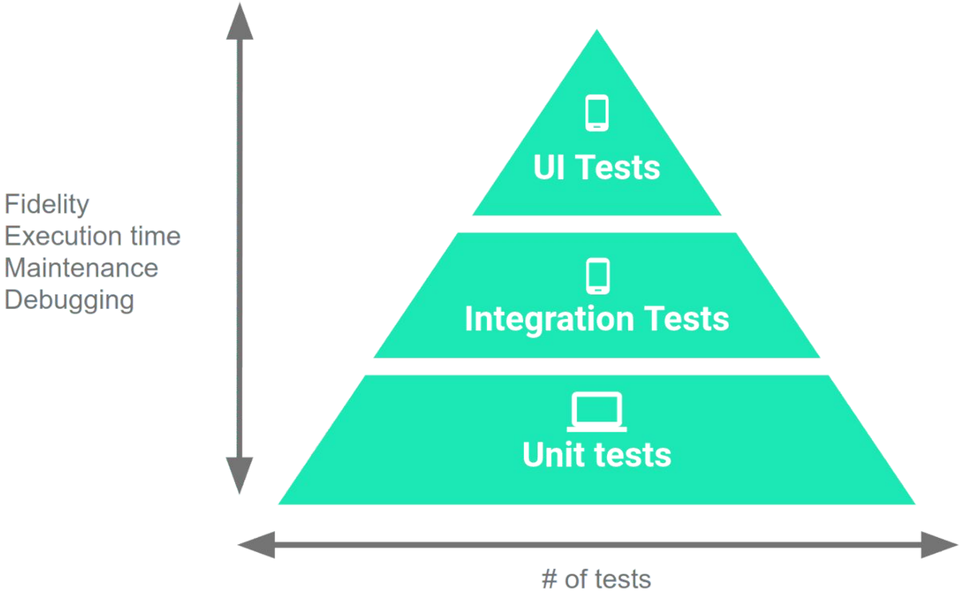
- Unit tests focus on a **single class**.
    - Simple functionality, more freedom.
    - Few method calls.
- System tests **bring many classes together**.
    - Focus on testing through an interface.
    - One interface call triggers many internal calls.
        - Slower test execution.
    - May have complex input and setup.

Interface Errors

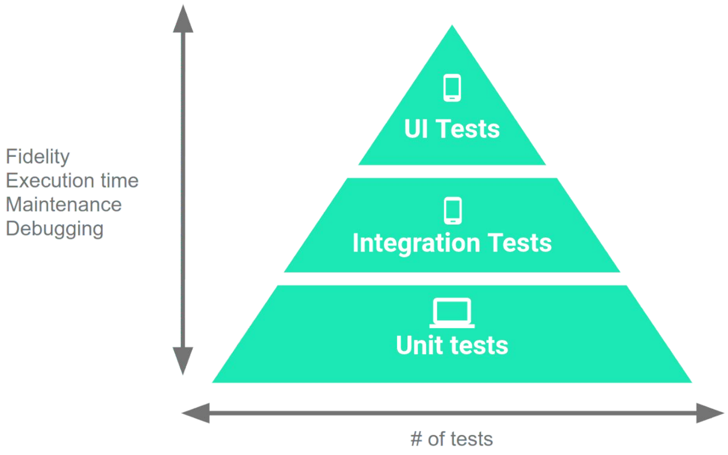
- Interface Misuse
    - Malformed data, order, number of parameters.
- Interface Misunderstanding
    - Incorrect assumptions made about called component.
    - A binary search called with an unordered array.
- Timing Errors
    - Producer of data and consumer of data access data in the wrong order.

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.

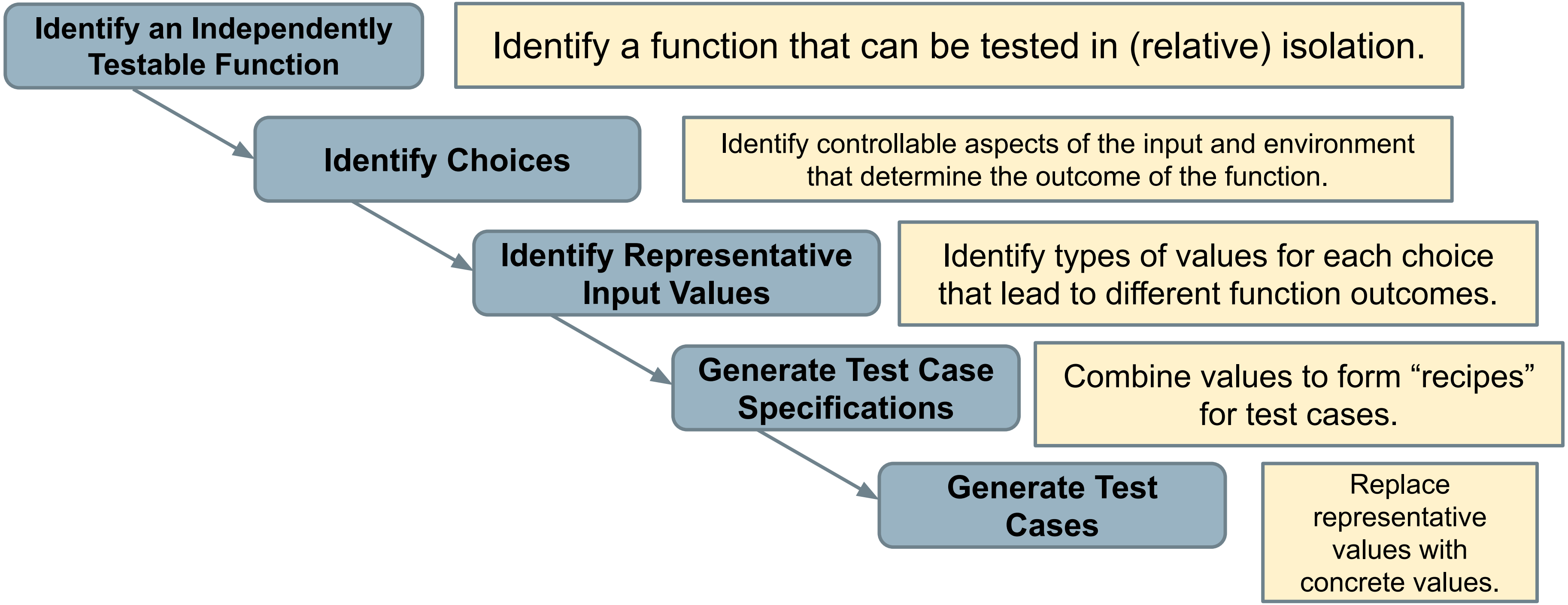
System-Level Tests and SPLs

- 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.**

Let’s take a break.

Creating Test Cases

Creating System-Level Tests



Identify an Independently Testable Function

Identify a function that can be tested in (relative) isolation.

Identify Choices

Identify controllable aspects of the input and environment that determine the outcome of the function.

Identify Representative Input Values

Identify types of values for each choice that lead to different function outcomes.

Generate Test Case Specifications

Combine values to form “recipes” for test cases.

Generate 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"

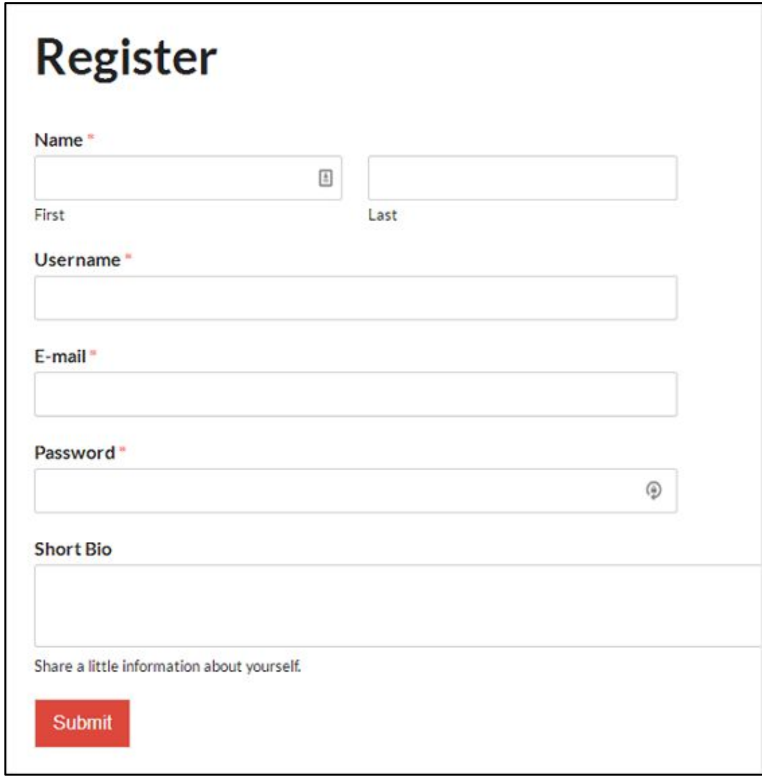
- Many tests written in terms of "units" of code.
- An independently testable function is a *capability* of the software.
  - Can be at class, subsystem, or system level.
  - **Defined by an interface.**

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)`



Register

Name *

First

Last

Username *

E-mail *

Password *

Short Bio

Share a little information about yourself.

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.**

Example

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

```
@Test
public void testRegistration() {
    // Set Up
    // Attempt to register for a course
    Boolean outcome = registerForCourse(studentID, courseID);
    Boolean expected = … ; // Set expected value, true or false
    // Check the result of registration
    assertEquals(expected, outcome);
}
```

Example - Register for a Class

What are the choices we make when we design a test case?

```
@Test
public void testRegistration() {
    // Set Up
    // Attempt to register for a course
    Boolean outcome = registerForCourse(studentID, courseID);
    Boolean expected = … ; // Set expected value, true or false
    // Check the result of registration
    assertEquals(expected, outcome);
}
```

- Does student meet prerequisites?
- Does the course exist?
- **What else influences the outcome?**

Identify Choices

Example - Register for a Class

- During setup, we can influence a student’s record and the course records.
  - These are “inputs” to consider.
- How are they used?
  - Has a student already taken the course?
  - Do they meet the prerequisites?
  - Does a course exist?
  - What are the prerequisites of a course.

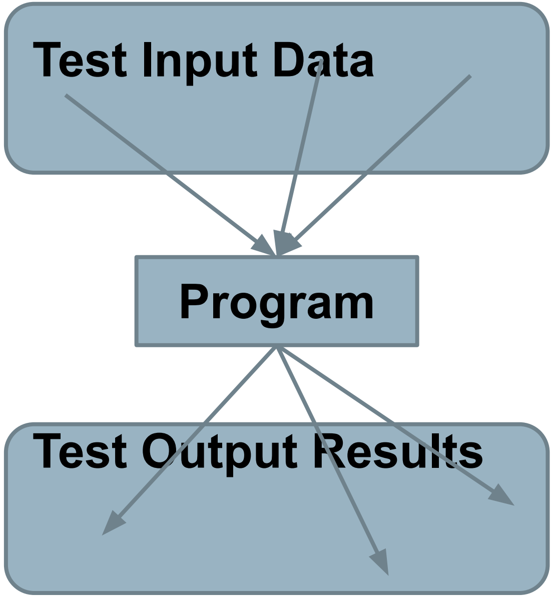
Example - Register for a Class

Test Choices

- **Parameter: studentID**
  - Validity of Student ID
  - Courses Student Has Taken Previously
- **Parameter: courseID**
  - Validity of Course ID
  - Prerequisites of Course ID

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:

```
add(int a, int b)
```

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

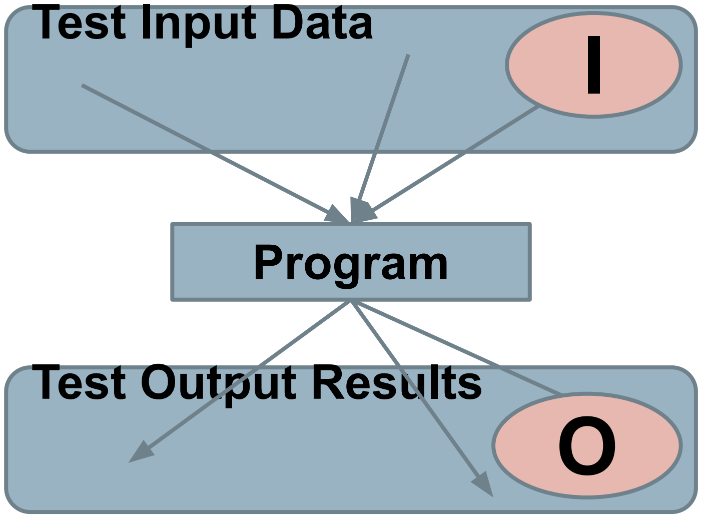
2^{32} possible integer values for each parameter.
$= 2^{32} \times 2^{32} = 2^{64}$
combinations $= 10^{13}$ tests.

1 test per nanosecond
$= 10^{5}$ tests per second
$= 10^{10}$ seconds

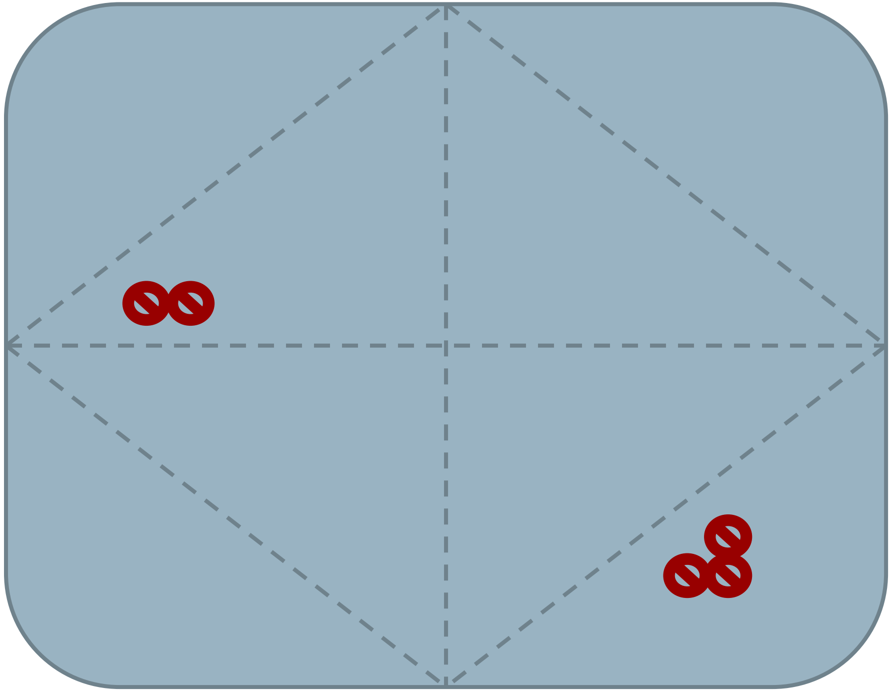
or… about 600 years!

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.

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

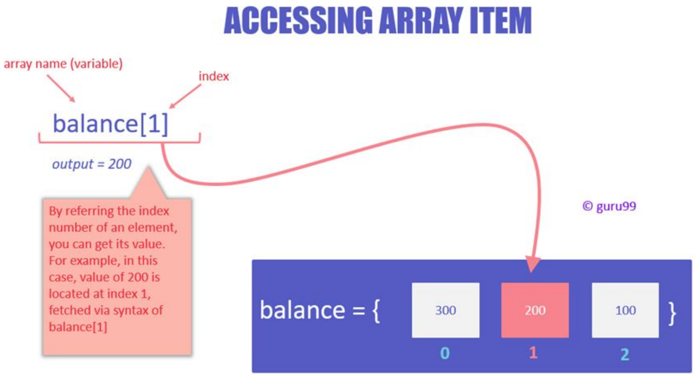
- Look at the outcomes and group input by the outcomes they trigger.
- Example: **`getEmployeeStatus(employeeID)`**
  - Outcomes include: Manager, Developer, Marketer, Lawyer, Employee Does Not Exist, Malformed ID
  - These are representative values for choice `employeeID`.
    - Can potentially break down further.

Data Type

- Divide based on both data type and how parameter used in function.
  - Ex: Integer
    - Basic Split: < 0, 0, >0
    - If conversions take place from String -> Integer, use a non-numeric string.
    - Other splits based on context.
      - Ex: Integer intended to be 5-digit: < 10000, 10000-99999, >= 100000
      - Try “expected” values and potential error cases.

Data Type

- Data structures are also 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.



By referring the index number of an element, you can get its value. For example, in this case, value of 200 is located at index 1, fetched via syntax of balance[1]

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)

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`

Example - Register for a Class

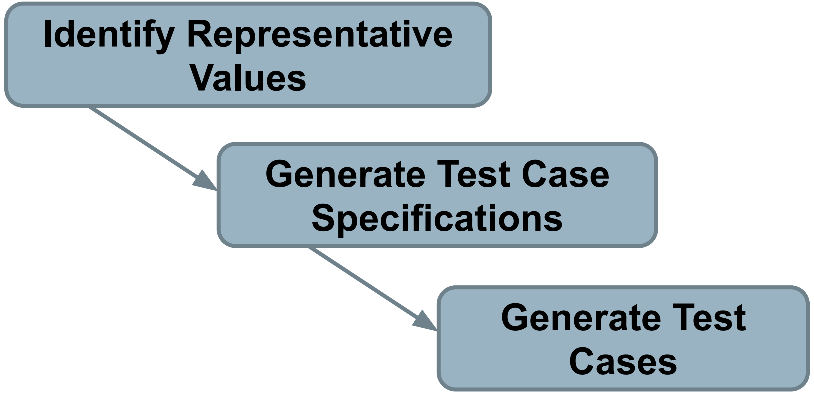
Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Revisit the Roadmap



Identify Representative Values

Generate Test Case Specifications

Generate Test Cases

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

```
@Test
public void testRegistration() {
    // Set Up
    setupStudentRecord(studentID, status, coursesTaken);
    setupCourse(courseID, prerequisites),
    // Attempt to register for a course
    Boolean outcome = registerForCourse(studentID, courseID);
    Boolean expected = … ; // Set expected value, true or false
    // Check the result of registration
    assertEquals(expected, outcome);
}
```

Forming Specification

Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Test Specifications:

- Active, Matches, Existing, Only Taken
- Active, Does Not Match, Existing, Only Not Taken
- Active, Does Not Match, Existing, Some Taken
- Active, - , Non-Existing, -
- Inactive, Matches, Existing, Only Taken
- Inactive, Does Not Match, Existing, Only Not Taken
- Inactive, Does Not Match, Existing Some Taken
- Inactive, - , Non-Existing, -
- Non-Existing, -, Existing, -
- Non-Existing, -, Non-Existing, -
- …

# Specifications: 3 * 2 * 2 * 3 = 36 - Illegal Combinations

Generate Test Cases

Specification:

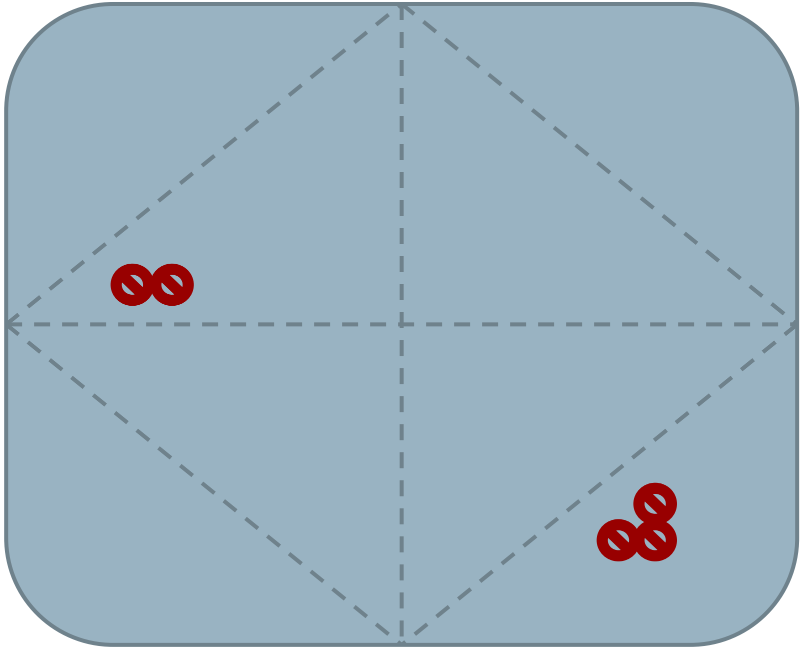
Active, Matches, Existing, Only Taken

```
@Test
public void testRegistration() {
    // Set Up
    setupStudentRecord(ggay, active, [TDA050, TDA360]);
    setupCourse(TDA594, [TDA360]),
    // Attempt to register for a course
    Boolean outcome = registerForCourse(ggay, TDA594);
    Boolean expected = true;
    // Check the result of registration
    assertEquals(expected, outcome);
}
```

- Fill in concrete values that match the representative values classes.
- Can create MANY concrete tests for each specification.

Boundary Values

- Errors tend to occur at the boundary of a partition.
- Remember to select inputs from those boundaries.

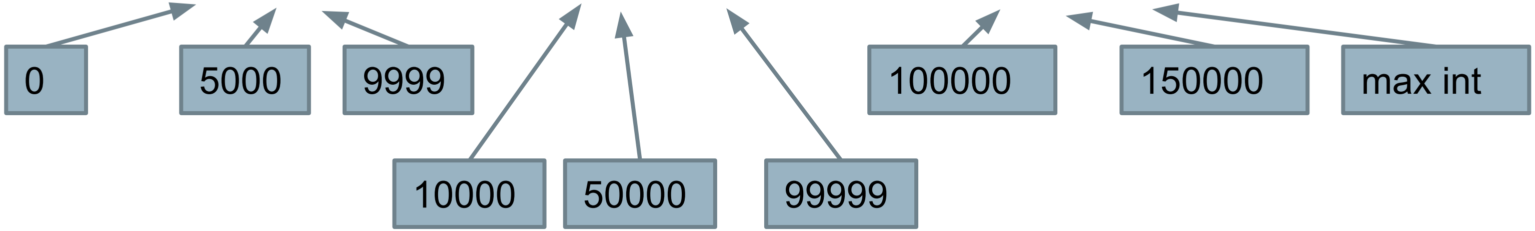


Boundary Values

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



0

5000

9999

10000

50000

99999

100000

150000

max int

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 3 - Dec 4
- Assignment 4 - Out Now - Dec 18
  - Any questions?

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY