CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 6: Test Design and Unit Testing

Gregory Gay
DIT636/DAT560 - February 5, 2025

Today’s Goals

- More on test design
  - More practice
  - Using constraints to limit representative value selection.
- Unit testing

More on Test Design (Adding Constraints)

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Creating Functional Tests

Identify Testing Targets	Identify **function(s)** that can be tested in (relative) isolation.
Identify Choices	Identify **controllable aspects** of the input and environment that determine the outcome of that function.
Identify Representative Input Values	Identify **options for each choice** that lead to different function outcomes.
Generate Test Case Specifications	**Select a value for all choices** to form abstract test case “recipe”.
Generate Test Cases	Replace representative values with **concrete values**.

Identify Choices

- Examine parameters of function.
  - *Direct input*, *environmental parameters (i.e., databases)*, and *configuration options*.
- Identify characteristics of each parameter.
  - What aspects influence outcome? (**choices**)

Example - Set Functions

- Small function library related to Sets:
  - `POST /insert/SET_ID {“object”: VALUE}`
    - `Returns { “result”: VALUE (“OK” if success or error)}`
  - `GET /find/SET_ID {“object”: VALUE}`
    - `Returns { “result”: VALUE (TRUE or FALSE)}`
  - `GET /delete/SET_ID {“object”: VALUE}`
    - `Returns { “result”: VALUE (“OK” if success or error)}`
- We want to write tests for these three functions.

Example - Set Functions

Identify Testing Targets

POST /insert/SET_ID {“object”: VALUE}

- What are our choices?

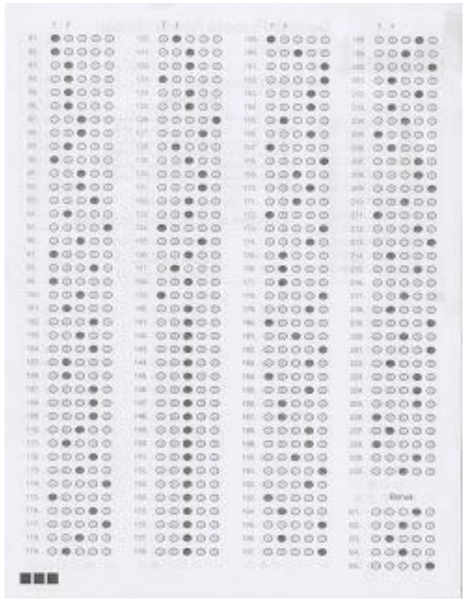
Identify Choices

```
// Set up the existing set, either empty or
with items.
POST /insert/ {“set”: [ …]}

// Insert an object
POST /insert/SET_ID {“object”: VALUE}

// Check the result
pm.test(“Insertion”, function() {
  var jsonData = pm.response.json();
  pm.expect(jsonData.result).to.eql(VALUE);});
```

- **Parameter:** Set ID
  - **Choice 1:** How many items are in the set? (performance may degrade with larger sets)
- **Parameter:** Object
  - **Choice 2:** Is obj already in the set?
  - **Choice 3:** Is the object valid? (e.g., not null)?

Identify Representative Values

- Many values can be selected for each choice.
- Partition values into **equivalence classes**.
  - Sets of interchangeable values.
  - Consider all outcomes of function.
  - Consider logical ranges or groupings.
- A test specification is a selection of values for all choices.
  - Concrete test case replaces equivalence class with a concrete value.



Example - Set Functions

Identify Representative Input Values

```
POST /insert/SET_ID {“object”: VALUE}
```

Parameter: Set ID

- **Choice:** How many items are in the set?
  - **Representative Values:**
    - Empty Set
    - Set with 1 item
    - Set with 10 items
    - Set with 10000 items

Parameter: object

- **Choice:** Is the object already in the set?
  - **Representative Values:**
    - obj already in set
    - obj not in set
- **Choice:** Is the object valid?
  - **Representative Values:**
    - Valid obj
    - Null obj

Test Specifications

- Test specification = selection a values for each choice.
  - May end up with thousands of test specifications.
  - Many specifications may be redundant or illegal.
  - **Identify constraints** to limit selection of values.

Example - Set Functions

Generate Test Case Specifications

Set Size	Obj in Set	Obj Status	Outcome
Empty	Yes	Valid	No change
Empty	Yes	Null	Error
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
1 item	Yes	Null	Error
1 item	No	Valid	Obj added to Set
1 item	No	Null	Error
10 items	Yes	Valid	No change
10 items	Yes	Null	Error
10 items	No	Valid	Obj added to Set
10 items	No	Null	Error

```
POST /insert/SET_ID
{“object”: VALUE}
```

- (4 * 2 * 2) = 16 specifications
- Each can become 1+ tests.
- Use constraints to remove impossible combinations.

Set Size	Obj in Set	Obj Status	Outcome
10000	Yes	Valid	No change (may be slowdown)
10000	Yes	Null	Error
10000	No	Valid	Obj added to Set(may be slowdown)
10000	No	Null	Error (may be slowdown)

Constraints Between Choices

- IF-CONSTRAINT
  - This representative value can only used if a certain value is used for a second choice (**if Choice 1 == X, Choice 2 can be Y**)
- ERROR
  - Selected representative value causes error regardless of values selected for other choices.
  - Error = can lead to abnormal outcome or exception.
- SINGLE
  - Corner cases that *should* give “normal” outcome.
  - Only a single test with this representative value is needed.

Example - Substring

`substr(string str, int index)`

Choice: Str length

length = 0

length = 1

length >= 2

Choice: Str contents

contains letters and numbers — if “Str length” != 0

contains special characters — if “Str length” != 0 — SINGLE

empty — if “Str length” = 0

Choice: index

value < 0 — ERROR

value = 0

value = 1

value > 1

Example - Set Functions

Identify Constraints

```
POST /insert/SET_ID {“object”: VALUE}
```

Parameter: set

- **Choice:** How many items are in the set?
  - **Representative Values:**
    - Empty Set
    - Set with 1 item
    - Set with 10 items single
    - Set with 10000 items single

Parameter: obj

- **Choice:** Is the object already in the set?
  - **Representative Values:**
    - obj already in set if “how many” != empty
    - obj not in set
- **Choice:** Is the object valid?
  - **Representative Values:**
    - Valid obj
    - Null obj error

Example - Set Functions

Apply Constraints

Set Size	Obj in Set	Obj Status	Outcome
~~Empty~~	~~Yes~~	~~Valid~~	~~No change~~
~~Empty~~	~~Yes~~	~~Null~~	~~Error~~
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
~~1 item~~	~~Yes~~	~~Null~~	~~Error~~
1 item	No	Valid	Obj added to Set
~~1 item~~	~~No~~	~~Null~~	~~Error~~
~~10 items~~	~~Yes~~	~~Valid~~	~~No change~~
~~10 items~~	~~Yes~~	~~Null~~	~~Error~~
10 items	No	Valid	Obj added to Set
~~10 items~~	~~No~~	~~Null~~	~~Error~~

```
POST /insert/SET_ID
{“object”: VALUE}
```

(4 * 2 * 2) = 16 specifications

Can’t already be in empty set, - 2

error (null), - 6

single (10, 10000), - 2

Set Size	Obj in Set	Obj Status	Outcome
~~10000~~	~~Yes~~	~~Valid~~	~~No change (may be slowdown)~~
~~10000~~	~~Yes~~	~~Null~~	~~Error (may be slowdown)~~
10000	No	Valid	Obj added to Set(may be slowdown)
~~10000~~	~~No~~	~~Null~~	~~Error (may be slowdown)~~

Example - Set Functions

Apply Constraints

Set Size	**Obj in Set**	**Obj Status**	**Outcome**
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
1 item	No	Valid	Obj added to Set
10 items	No	Valid	Obj added to Set
10000	No	Valid	Obj added to Set(may be slowdown)

```
POST /insert/SETID
{“object”: VALUE}
```

- From 16 -> 6 specifications
- Each can become 1+ tests.
- Can further constrain if needed.

Example - Set Functions

Create Test Cases

```
POST /insert/SET_ID {“object”: VALUE}
```

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Valid	Obj added to Set

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Null	Error

```
// Set up empty set.
POST /insert/ {“set”: []}
// Insert a valid object
POST /insert/SET_ID {“object”: “Test”}
// Check the result
pm.test(“Valid Insert”, function() {
  var jsonData = pm.response.json();
pm.expect(jsonData.result).to.eql(“OK”);
});
```

```
// Set up empty set.
POST /insert/ {“set”: []}
// Insert a null object
POST /insert/SET_ID {“object”: null}
// Check the result
pm.test(“Null Insert”, function() {
  var jsonData = pm.response.json();
pm.expect(jsonData.result).to.eql(“Null object
cannot be inserted into set”);});
```

Activity - `find` service

`find(pattern,file)`

- Finds instances of a pattern in a file
  - `find("john",myFile)`
    - Finds all instances of <u>`john`</u> in the file
  - `find("john smith",myFile)`
    - Finds all instances of <u>`john smith`</u> in the file
  - `find("“john” smith",myFile)`
    - Finds all instances of <u>`“john” smith`</u> in the file

Activity - `find` Service

- Parameters: pattern, file
- What can we vary for each?
  - What can we control about the pattern? Or the file?
- What values can we choose for each choice?
  - **File name:**
    - File exists with that name
    - File does not exist with that name
- What constraints can we apply between choice values? (if, single, error)

Example - `find` Service

$(2^2 * 3^3 * 4^1)$ = 108 test specifications

Pattern:

- Pattern size:
  - Empty
  - single character
  - many characters
  - longer than any line in the file
- Quoting:
  - pattern has no quotes
  - pattern has proper quotes
  - pattern has improper quotes (only one “)
- Embedded spaces:
  - No spaces
  - One space
  - Several spaces

File:

- File name:
  - Existing file name
  - no file with this name
- Number of occurrence of pattern in file:
  - None
  - exactly one
  - more than one
- Pattern occurrences on any single line line:
  - One
  - more than one

ERROR and SINGLE Constraints

4 (error) + 2 (single) + (1^2*2^3*3^1) = 30

- Pattern size:
  - [error] Empty
  - single character
  - many character
  - [error] longer than any line in the file
- Quoting:
  - pattern has no quotes
  - pattern has proper quotes
  - [error] pattern has improper quotes (only one “)
- Embedded spaces:
  - No spaces
  - One space
  - Several spaces

- File name:
  - Existing file name
  - no file with this name [error]
- Number of occurrence of pattern in file:
  - None
  - exactly one [single]
  - more than one
- Pattern occurrences on target line:
  - One
  - more than one [single]

IF Constraints

4 (error) + 2 (single) + (1^3*2^3) (quoted = true) + (1^4*2^2) (quoted = false) = 18

- Pattern size:
  - [error] Empty
  - single character
  - many character
  - [error] longer than any line in the file
- Quoting:
  - pattern has no quotes
  - pattern has proper quotes
  - [error] pattern has improper quotes (only one “)
- Embedded spaces:
  - No spaces
  - **[if quoting = proper]** One space
  - **[if quoting = proper]** Several spaces

- File name:
  - Existing file name
  - no file with this name **[error]**
- Number of occurrence of pattern in file:
  - None
  - exactly one **[single]**
  - more than one
- Pattern occurrences on target line:
  - One
  - more than one **[single]**

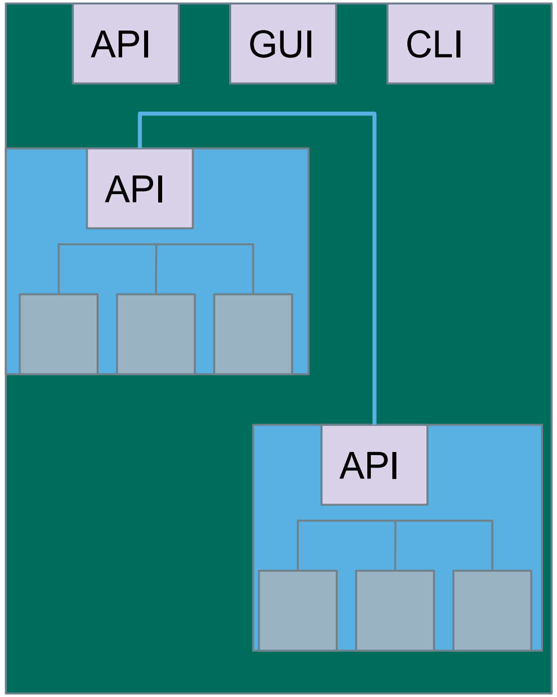
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Let’s take a break.

Unit Testing

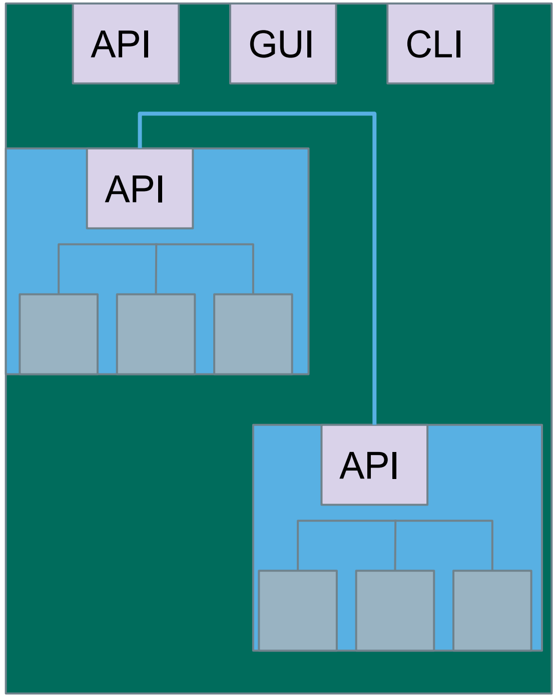
Testing Stages

- 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.



Testing Stages

- **Unit Testing**
  - Methods of a single class
- **System-level Testing**
  - **System (Integration) Testing**
    - (Subsystem-level) Collected units
    - (System-level) High-level interfaces
  - **Exploratory Testing**
    - Ad-hoc GUI testing method

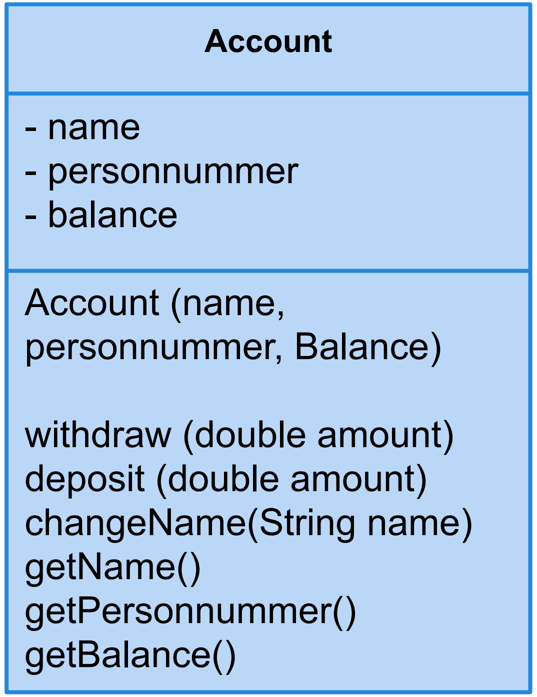


Unit Testing

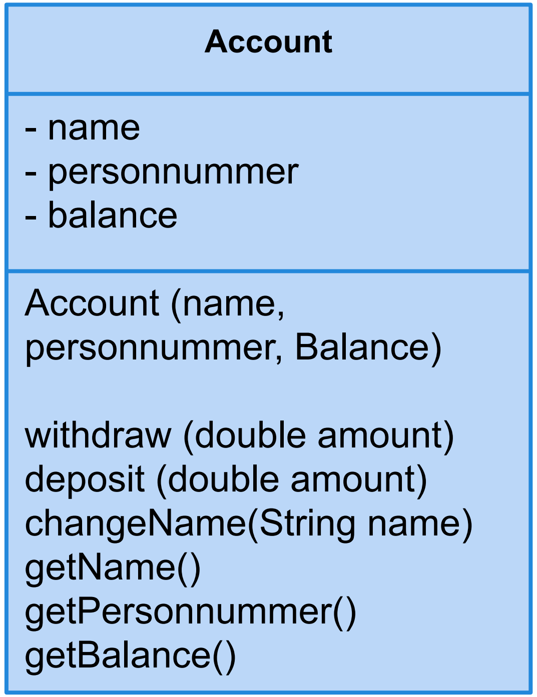
- Testing the smallest “unit” that can be tested.
  - Often, a class and its methods.
- Tested in **isolation** from all other units.
  - **Mock** the results from other classes.
- Test input = method calls.
- Test oracle = assertions on output/class variables.

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).



Unit Testing - Account



Account

- name
- personnummer
- balance

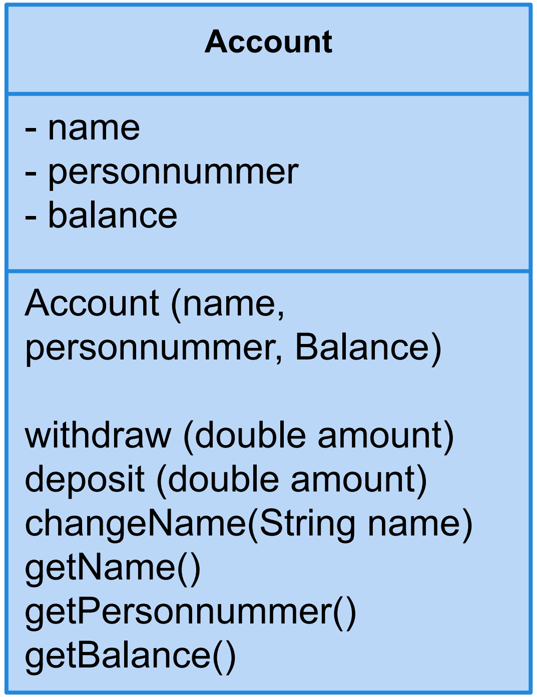
Account (name, personnummer, Balance)

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

Unit tests should cover:

- Set and check class variables.
  - Can any methods change name, personnummer, balance?
  - Does changing those create problems?
- 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).

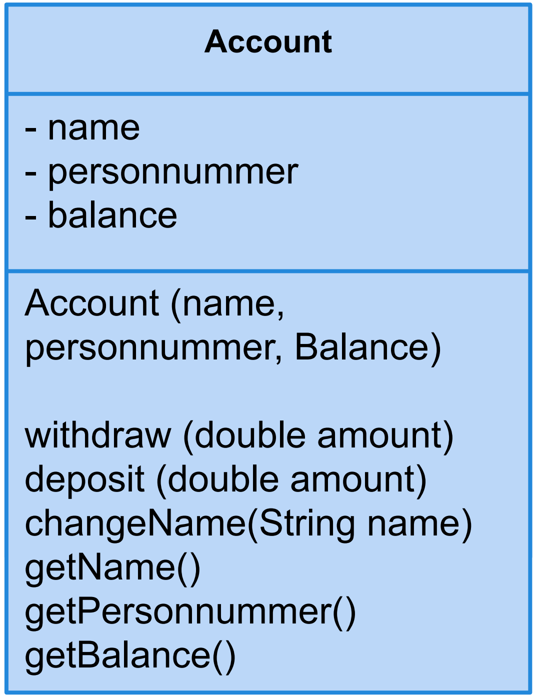
Unit Testing - Account



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.

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?)

Test Case Components

- Test Input
    - Any required input data.
- Expected Output (Test Oracle)
    - What *should* happen, i.e., values or exceptions.
- Initialization
    - Any steps that must be taken before test execution.
- Test Steps
    - Interactions (e.g., method calls), and output comparisons.
- Tear Down
    - Steps that must be taken after execution to prepare for the next test.

Writing a Unit Test

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

- Choose a target from the code base.
- Write a “testing class” containing a series of unit tests centered around testing that target.

```
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
```

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

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

```
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;
}
```

More Test Fixtures

- **@BeforeAll** defines initialization to take place before any tests are run.
- **@AfterAll** defines tear down after all tests are done.

```
@BeforeAll
  public static void setUpClass() {
     myManagedResource = new
            ManagedResource();
  }

  @AfterAll
  public static void tearDownClass()
throws IOException {
    myManagedResource.close();
    myManagedResource = 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("text", "text", "failure - 
strings are not equal");
}

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

- 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.

assertFalse, assertTrue

```
@Test
public void testAssertFalse() {
   assertFalse((getGrade(studentA,
“DIT635”).equals(“A”), "failure - should be
false");
}

@Test
public void testAssertTrue() {
      assertTrue((getOwed(studentA) > 0),
"failure - should be true");
}
```

- Take in a string and a boolean expression.
- Evaluates the expression and issues pass/fail based on outcome.
- Used to check conformance of solution to expected properties.

assertSame, assertNotSame

```
@Test
public void testAssertNotSame() {
   assertNotSame(studentA, new Object(),
"should not be same Object");
}


@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame(studentA, studentB, "should be
same");
}
```

- Checks whether two objects are clones.
- Are these variables aliases for the same object?
  - assertEquals uses .equals().
  - assertSame uses ==

assertNull, assertNotNull

```
@Test
public void testAssertNotNull() {
   assertNotNull(new Object(), "should 
not be null");
}

@Test
public void testAssertNull() {
   assertNull(null, "should be null");
}
```

- Take in an object and checks whether it is null/not null.
- Can be used to help diagnose and void null pointer exceptions.

Grouping Assertions

```
@Test
void groupedAssertions() {
  Person person = Account.getHolder();
  assertAll("person",
    () -> assertEquals("John",
person.getFirstName()),
    () -> assertEquals("Doe",
person.getLastName()));
}
```

- Grouped assertions are executed.
  - Failures are reported together.
  - Preferred way to compare fields of two data structures.

assertThat

ha tems,
bu

either - pass if one of these properties is true.

```
@Test
public void testAssertThat{
  assertThat("albumen", both(containsString("a")).and(containsString("b")));
  assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
                  everyItem(containsString("n")));
  assertThat("good", allOf(equalTo("good"), startsWith("good")));
  assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
  assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
  assertThat(7, not(CombinableMatcher.<Integer>
                  either(equalTo(3)).or(equalTo(4))));
}
```

Testing Exceptions

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

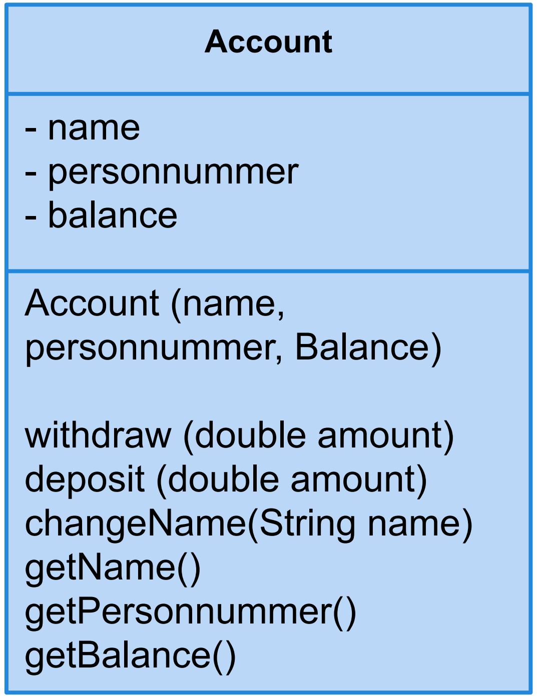
- When testing error handling, we expect exceptions to be thrown.
  - **assertThrows** checks whether the code block throws the expected exception.
  - **assertEquals** can be used to check the contents of the stack trace.

Testing Performance

```
@Test
void timeoutExceeded() {
  assertTimeout( ofMillis(10),
  () -> { Order.process(); });
}
@Test
void timeoutNotExceededWithMethod() {
  String greeting =
    assertTimeout(ofMinutes(2),
      AssertionsDemo::greeting);
  assertEquals("Hello, World!", greeting);
}
```

- **assertTimeout** can be used to impose a time limit on an action.
  - Time limit stated using `ofMilis(..)`, `ofSeconds(..)`, `ofMinutes(..)`
  - Result of action can be captured as well, allowing checking of result correctness.

Unit Testing - Account



Account

- name
- personnummer
- balance

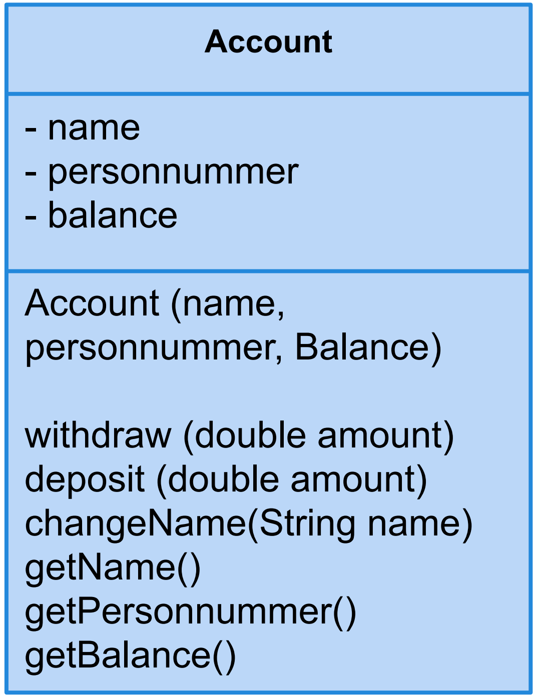
Account (name, personnummer, Balance)

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

- Withdraw money, verify balance.

```
@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
}
```

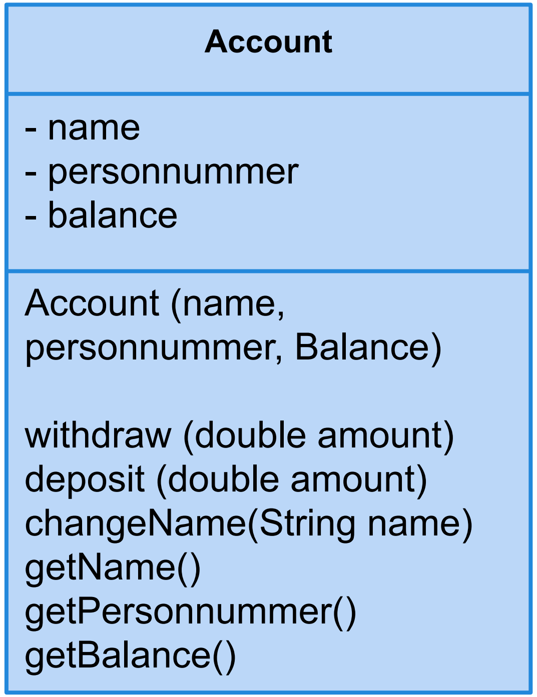
Unit Testing - Account



- Withdraw more than is in balance.
  - (should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_moreThanBalance() {
    // Setup
    Account account = new Account(“Test McTest”, “19850101-1001”, 48.5);
    // Test Steps
    double toWithdraw = 100.0; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals(“Amount 100.00 is greater than balance 48.50”,
                 exception.getMessage()); // Oracle
}
```

Unit Testing - Account



Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

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

- 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
}
```

Best Practices

- Use assertions instead of print statements

```
@Test

public void testStringUtil_Bad() {

    String result = stringUtil.concat("Hello ", "World");
    System.out.println("Result is "+result);
}

@Test
public void testStringUtil_Good() {
    String result = stringUtil.concat("Hello ", "World");
    assertEquals("Hello World", result);
}
```

- The first will always pass (no assertions)

Best Practices

- If code is non-deterministic, tests should give deterministic results.

```
public long calculateTime(){
    long time = 0;
    long before = System.currentTimeMillis();
    veryComplexFunction();
    long after = System.currentTimeMillis();
    time = after - before;
    return time;
}
```

- Tests for this method should not specify exact time, but properties of a “good” execution.
  - The time should be positive, not negative or 0.
  - A range on the allowed times.

Best Practices

- Test only one unit at a time.
    - Each scenario in a separate test case.
    - Helps in isolating and fixing faults.
- Do not use unnecessary assertions.
    - Specify how code should work, not a list of observations.
    - Generally, each unit test performs one assertion
        - Or all assertions are related.

Best Practices

- Make each test independent of all others.
  - Use `@BeforeEach` and `@AfterEach` to set up state and clear state before the next test case.
- Create unit tests to target exceptions.
  - If an exception should be thrown based on certain input, make sure the exception is thrown.

We Have Learned

- **Constraints** can be used in **functional test design** to limit test specifications we create.
  - Error, single, if
- **Unit testing** focuses on individual classes in isolation from the rest of the system.
  - Input = method calls
  - Oracle = assertions

Next Time

- Exercise Session: Functional test design
- Next class: System testing and test automation

- Assignment 1 - Feb 6
- Assignment 2 - Feb 16
  - Any questions?

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY