Database Testing

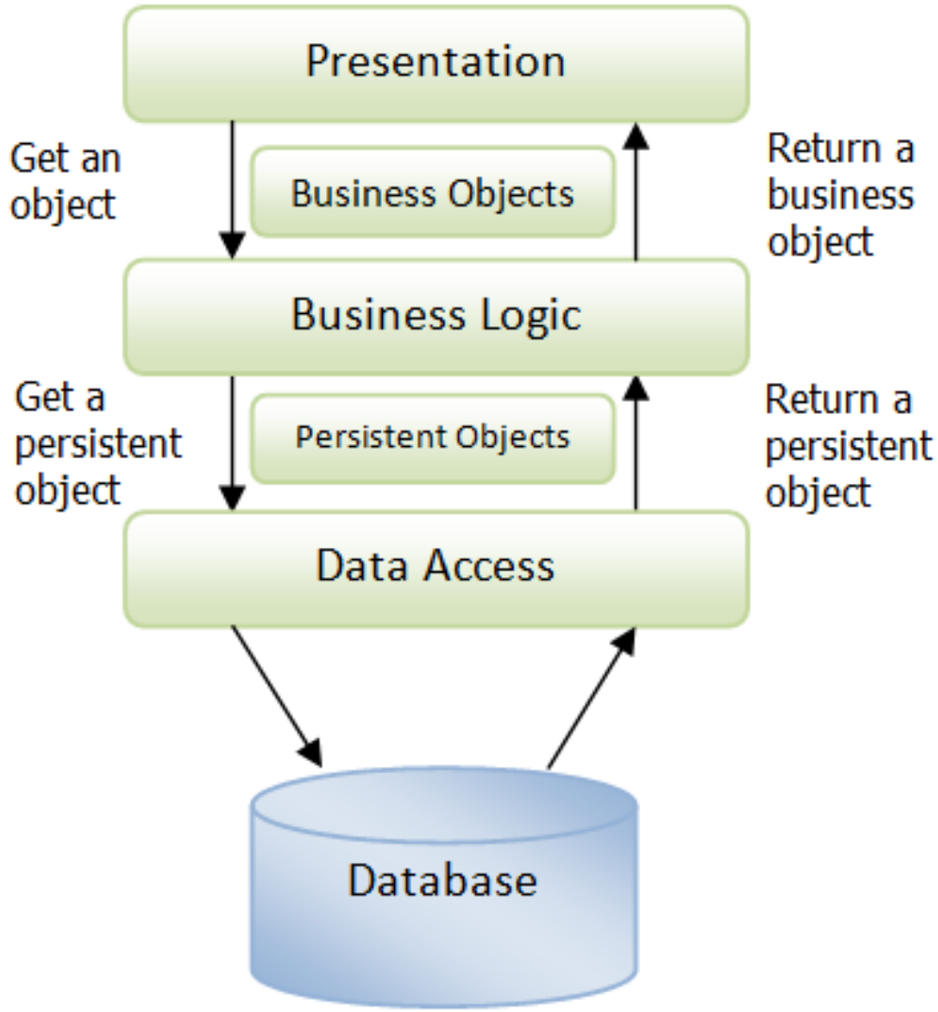
- Difficulties in database testing
- Database testing patterns
- Tools/libraries

Ciências ULisboa

slides: Eduardo Marques, João Neto

Database testing

- Database use is common …
- Testing code that accesses databases must be done carefully.
- A few questions:
  - Can we test the SUT without the DB. If so, how/at what level of testing granularity?
  - Should we let DB state be affected by a test persist?
  - Should we use an unique DB for developing&testing?
  - What DB data should we use?
  - How do we define test data?
  - How do we verify the database state ? Database access is normally “hidden” by the presentation layer, the business logic, etc.



Issues & advice by Meszaros [chap 13]

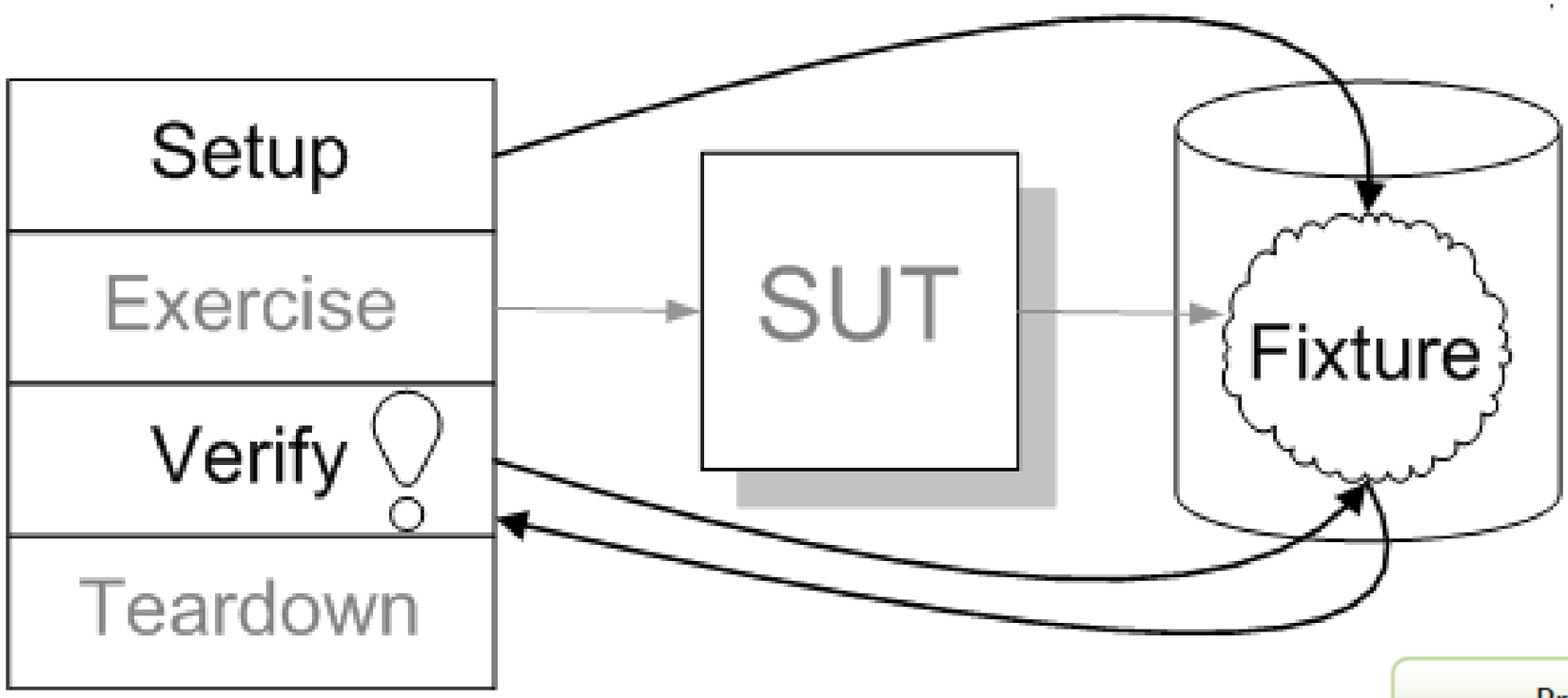
- "*My first and overriding piece of advice on this subject is: When there is any way to test without a database, test without the database!*"
- However… "*The database is a necessary part of the application and verifying that the database is used properly is a necessary part of building these applications.*"
  - "Because the data in a database may potentially persist long after we run our test, we must pay special attention to this data to avoid creating tests that can be run only once or tests that interact with one another."
  - "Databases are much slower than the processors used in modern computers. As a consequence, tests that interact with a database tend to run much more slowly than tests that can run entirely in memory."
  - Databases have more than just data: PL/SQL stored procedures, triggers, …

A few patterns

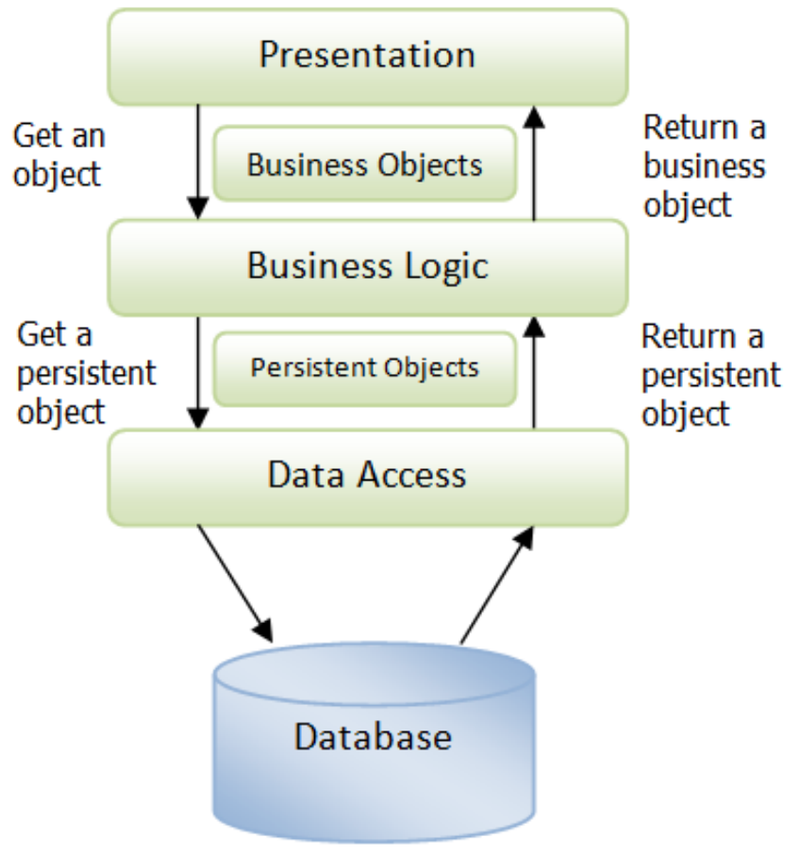
- Testing without the DB: Test Doubles (e.g., mock objects) can be useful. This makes sense for testing modules in isolation from database access.
- But what about integration testing and database validation…?
  - "Backdoor manipulation: we set up the test fixture or verify the outcome by going through a back door (such as direct database access)".
  - "Database sandbox pattern: we provide a separate test database for each developer or tester."
  - "Transaction rollback pattern: We rollback the uncommitted test transaction as part of the teardown." [not always feasible: commits may be performed by internal transactional logic of the SUT]
  - "Table Truncation Teardown": "Truncate the tables modified during the test to tear down the fixture." [handles inner transactional logic issues, but clears up all tables at stake]

Click the links above to access detailed descriptions at xunitpatterns.com

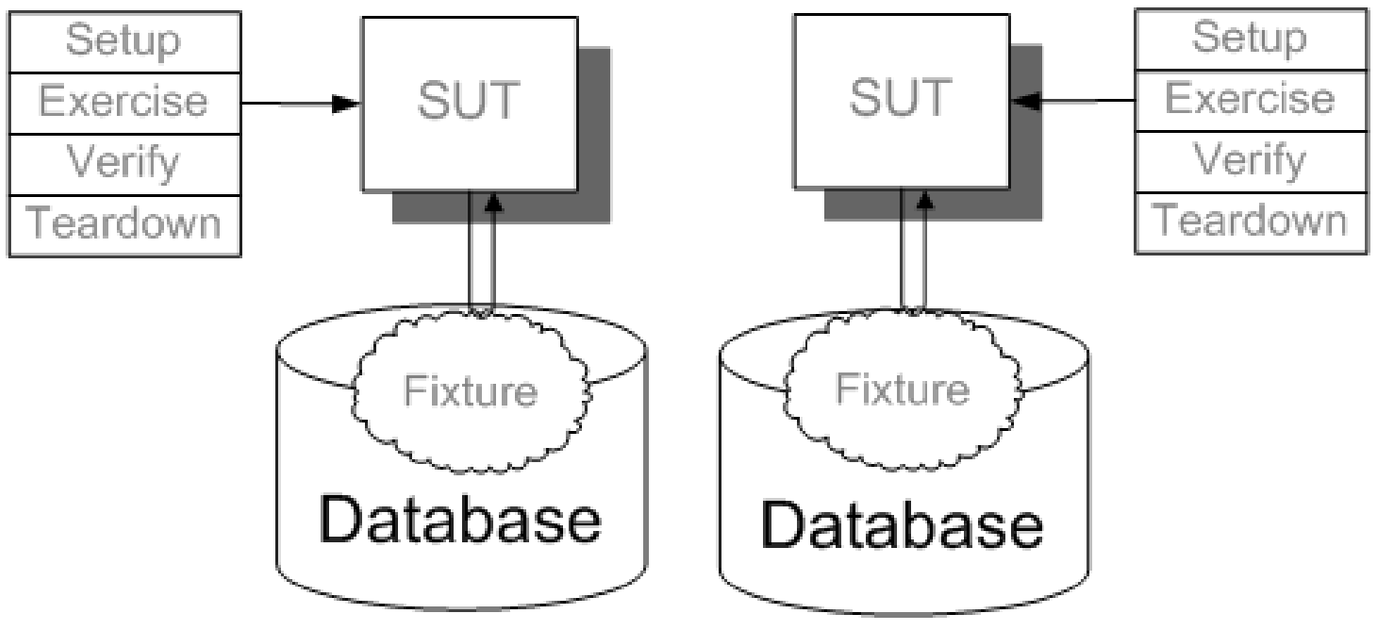
Backdoor manipulation



- SUT does not expose the DB directly
- We need some “backdoor” …
  - usually a direct connection to the database

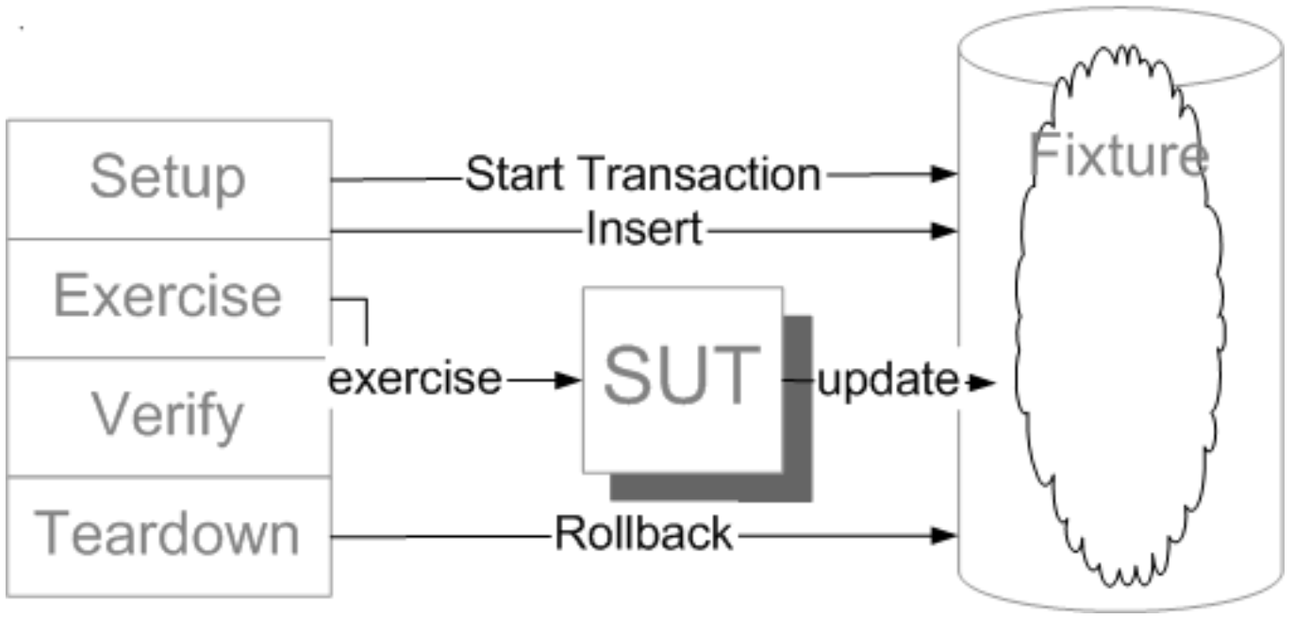


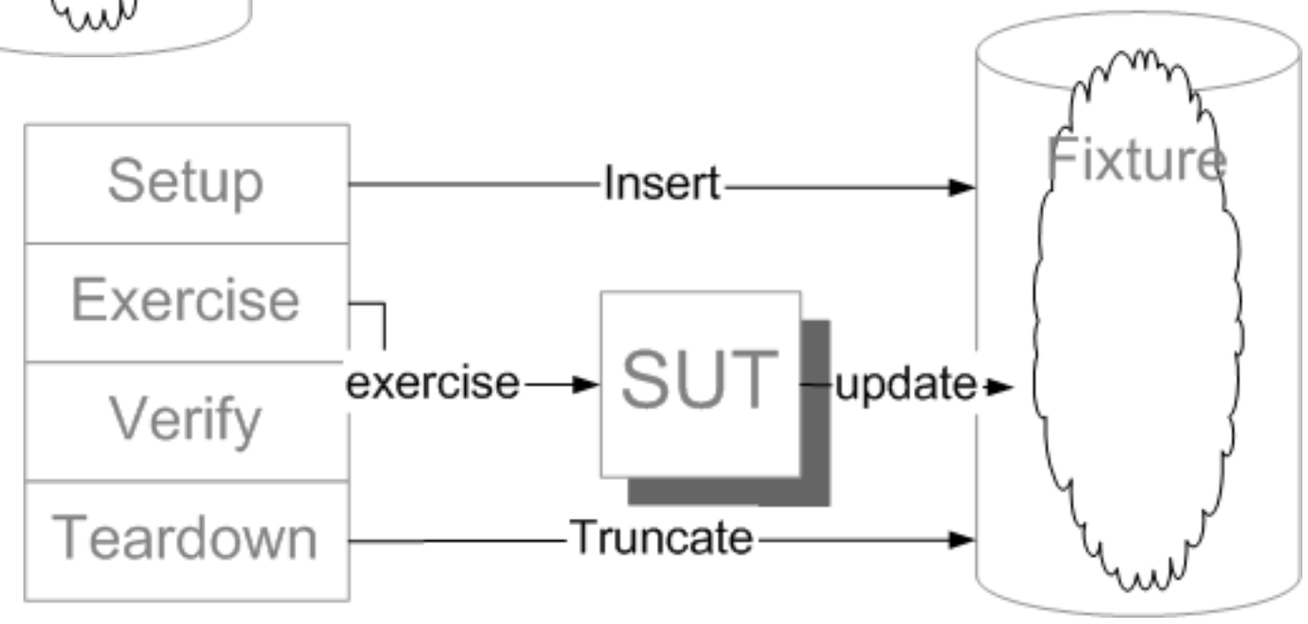
Database sandbox



- “We provide a separate copy of the database for each developer or tester”
  - Concurrent testing / developing / application use do not interfere with one another
  - A single suite of tests executes independently.
- Possibilities:
  - Dedicated Database Sandbox (each developer/tester uses a separate database)
  - DB Schema per TestRunner (each runner uses a separate schema)
  - Database Partitioning Scheme (each developer/tester uses a separate set of data within a single database)

Transaction rollback / Table truncation





- Each test case must leave the database in a clean state.
- It is also common to perform table truncation during setup, before inserting test data.

Complementary notes

- What data should we use?
  - The relevant data that is convenient for your test criteria …
  - Data sampled from a real system DB is normally useful …
- System/Acceptance testing: how much effort?
  - Meszaros: “Suppose we have done a good job layering our system and made it possible to run most of our tests without using the real database. *What kinds of tests should we run against the real database?* The answer is "*as few as possible, but no fewer*"! In practice, we want to run *at least a representative sample of our customer tests* against the database to ensure that the SUT behaves the same way with a database as without.”

Complementary notes

- Data Access tests should follow these rules:
  - They must use the real database schema
  - Be deterministic
    - Otherwise, do it methodically, like using QuickCheck
  - Must assert for the right thing
    - When writing data via a testing library, read it using the application API
    - If the data is retrieved by an external framework or library, we should not make tests for it -- we are only testing our code, our application
    - If we can cover most test situations with unit tests – and the dependencies with the persistence layer are small – probably we will not need integration tests concerning data access

Complementary notes

- Do not initialize your database with a single dataset for all tests
  - Divide and conquer also matters in data testing
- Your application has lots of functions and tables
  - A single dataset to cover them all will be very big and unmanageable, slower to initialize, and harder to find the reason why some test fails
- For each test set (e.g., the tests inside a class), build a minimal dataset expressive enough to be usable
  - It should only have data that can be asserted by the given tests (unused tables should be empty)
- Problem: if the database schema is not stable, it will be harder to maintain!
  - Possible solution: build data samples by combining them from a set of smaller, 'data blocks' samples
  - This way, we only need to maintain these basic data blocks

A few tools

- In-memory/standalone databases
  - Database data is entirely stored in memory or in stand-alone files
  - Database sandboxing, test repeatability, improved test performance
  - Many SQL relational database written in Java: we use HSQLDB
  - DbSetup is used for database setup/teardown
    - DbUnit can additionally be used for asserting the database state (somewhat limited and inflexible)
- utPLSQL, PL/SQL unit testing frameworks
- HammerDB, tool for stress/load testing
- Arquillian, an integration testing framework for Java EE

DBSetup

- Java library for database setup in unit tests
- Its task is to populate the database with test data
- Full Java. Does not use XML, SQL, or other data files like DBUnit
- http://dbsetup.ninja-squad.com/index.html

DBSetup Philosophy

- Classical way:
  - Start with empty database @ setup
  - Populate database
  - Run test
  - Cleanup data
- DBSetup:
  - Clear database @ setup
  - Insert small data sample @ setup
  - Run Test
- Pros:
  - If test fails, the data is still there
  - Tests do not assume the database starts empty
  - A good setup does not need cleanup!

DBSetup eg

- In this example, class `DBSteupUtils` includes information, processes and data samples to be shared by all tests

```
public class DBSetupUtils {

  public static final String DB_URL =
     "jdbc:hsqldb:file:src/main/resources/data/hsqldb/cssdb";
  public static final String DB_USERNAME = "SA";
  public static final String DB_PASSWORD = "";

  ...
```

DBSetup eg

- In this example, class `DBSteupUtils` includes information, processes and data samples to be shared by all tests

```
private static boolean appDatabaseAlreadyStarted = false;

public static void startApplicationDatabaseForTesting() {

  if (appDatabaseAlreadyStarted) // do it once
    return;

  try {
    webapp.persistence.DataSource.INSTANCE.connect(
            DB_URL, DB_USERNAME, DB_PASSWORD);
    appDatabaseAlreadyStarted = true;
  } catch (PersistenceException e) {
    throw new Error("DataSource could not be started");
  }
}
```

DBSetup eg

- DBSetup includes type `Operation` which encloses a database operation
- Class `Operations` is a factory for operations (herein, it is static imported to improve code readability)

```
////////////////////////////////////////////
// Operations for populating test database

public static final Operation DELETE_ALL =
  deleteAllFrom("CUSTOMER", "SALE", "ADDRESS", "SALEDELIVERY");

public static final int NUM_INIT_CUSTOMERS;
public static final int NUM_INIT_SALES;
public static final int NUM_INIT_ADDRESSES;

public static final Operation INSERT_CUSTOMER_SALE_DATA;
public static final Operation INSERT_CUSTOMER_ADDRESS_DATA;
```

DBSetup eg, data samples

- Since class `DBSteupUtils` only includes static elements, we initialize the data samples in a static field
- Operation `Insert` inserts one or several rows into a table

```
static {

  Insert insertCustomers =
    insertInto("CUSTOMER")
     .columns("ID", "DESIGNATION", "PHONENUMBER", "VATNUMBER")
     .values(   1,   "JOSE FARIA",     914276732,   197672337)
     .values(   2,  "LUIS SANTOS",     964294317,   168027852)
     .build();

  NUM_INIT_CUSTOMERS = insertCustomers.getRowCount();
```

DBSetup eg, data samples

- Each database table can have its data sample(s)
- Data samples can be combined using `Operations.sequenceOf`

```
Insert insertAddresses =
    insertInto("ADDRESS")
    .withGeneratedValue("ID",
        ValueGenerators.sequence().startingAt(100L).incrementingBy(1))
    .columns(                                   "ADDRESS", "CUSTOMER_VAT")
    .values(            "FCUL, Campo Grande, Lisboa",       197672337)
    .values(          "R. 25 de Abril, 101A, Porto",       197672337)
    .values( "Av Neil Armstrong, Cratera Azul, Lua",       168027852)
    .build();

  NUM_INIT_ADDRESSES = insertAddresses.getRowCount();

  INSERT_CUSTOMER_ADDRESS_DATA = sequenceOf(insertCustomers, insertAddresses);
} // static
```

DBSetup eg, test class

- Classes `Destination` and `DriverManagerDestination` are used to connect DBSetup with the application database

```
public class CustomersDBTest {

  private static Destination dataSource;

  private static DbSetupTracker dbSetupTracker =
     new DbSetupTracker();

  @BeforeClass
  public static void setupClass() {
    startApplicationDatabaseForTesting();
    dataSource = DriverManagerDestination.with(
      DB_URL, DB_USERNAME, DB_PASSWORD);
  }
```

DBSetup eg, setup

- Classes `Destination` and `DriverManagerDestination` are used to connect DBSetup with the application database
- For each test, we clear the database and insert a small data sample
- This test class should only include tests about customers and addresses

```
@Before
public void setup() throws SQLException {

  Operation initDBOperations = Operations.sequenceOf(
     DELETE_ALL
   , INSERT_CUSTOMER_ADDRESS_DATA
    );

  DbSetup dbSetup = new DbSetup(dataSource, initDBOperations);

  // Use the tracker to launch the DBSetup.
  // This will speed-up tests that do not change the DB.
  dbSetupTracker.launchIfNecessary(dbSetup);
}
```

DBSetup eg, test

- Some tests are just queries, they will not change the database
- Method `skipNextLaunch()` skips initialization for the next test
  - This technique produces faster tests

```
@Test
public void queryCustomerNumberTest() throws … {
  // read-only test: unnecessary to re-launch setup
  // after test has been run
  dbSetupTracker.skipNextLaunch();

  int expected = NUM_INIT_CUSTOMERS;
  int actual =
     CustomerService.INSTANCE.getAllCustomers().customers.size();

  assertEquals(expected, actual);
}
```

DBSetup eg, test

- For tests that change the database, we cannot skip initialization
- Notice that all the changes will be reset for the next test
- This guarantees stable testing, with no side effects

```
@Test
public void addCustomerSizeTest() throws … {

  CustomerService.INSTANCE
                 .addCustomer(503183504, "FCUL", 217500000);

  int size = CustomerService.INSTANCE
                            .getAllCustomers().customers.size();

  assertEquals(NUM_INIT_CUSTOMERS+1, size);
}
```

DBSetup eg, test

```
private boolean hasClient(int vat) throws … {
  CustomersDTO customersDTO =
      CustomerService.INSTANCE.getAllCustomers();

  for(CustomerDTO customer : customersDTO.customers)
    if (customer.vat == vat)
      return true;
  return false;
}

@Test
public void addCustomerTest() throws ApplicationException {

  assumeFalse(hasClient(503183504));

  CustomerService.INSTANCE
                 .addCustomer(503183504, "FCUL", 217500000);

  assertTrue(hasClient(503183504));
}
```

Exercises

- Import maven project `vvs_databases`
- Add tests about addresses to `CustomersDBTest`
- Create a new class with tests about sales
- Add delivery tests (including new data samples)