Integration Testing from the Trenches

Nicolas Fränkel

Integration Testing from the Trenches

Nicolas Fränkel

This book is for sale at http://leanpub.com/integrationtest

This version was published on 2016-08-04

ISBN 978-2-9550214-0-8

Tweet This Book!

Please help Nicolas Fränkel by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Read ”Integration Testing from the Trenches”, have a look at it https://leanpub.com/integrationtest #integrationtesting #testing

The suggested hashtag for this book is #integrationtesting.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#integrationtesting

Contents

1. Infrastructure Resources Integration

Applications are seldom found in the void: they interact with resources such as file systems, databases, SMTP servers, FTP servers and the like. Infrastructure integration refers to how application interactions with those resources can be tested and validated. While mocks or fakes can be used in place of infrastructure resources, they will not help validate the behavior of those resources.

This chapter covers strategies and tools to achieve testing resources in the following domains:

- System Time
- Filesystem
- Databases
- Mail servers
- FTP servers

Similar strategies can be used for other resource types. However, Web Services integration will be the subject of the next chapter given its scope.

1.1 Common resource integration testing techniques

This section describes techniques that enable and ease Integration Testing resources integration.

1.1.1 No hard-coded resource reference

Most infrastructure resources have something in common: they can be located through a unique identifier and offer a set of commands to be called. The following table displays some resources with an example identifier:

Table 5.1 - Resource identifiers sample

Â Resource	Identifier	Example
Relational DataBase Management System	Java Database Connectivity URL	`jdbc:mysql://localhost:3306/myapp`
SMTP server	Domain	`smtp.gmail.com`
FTP server	URL	`ftp://ftp.ietf.cnri.reston.va.us/`

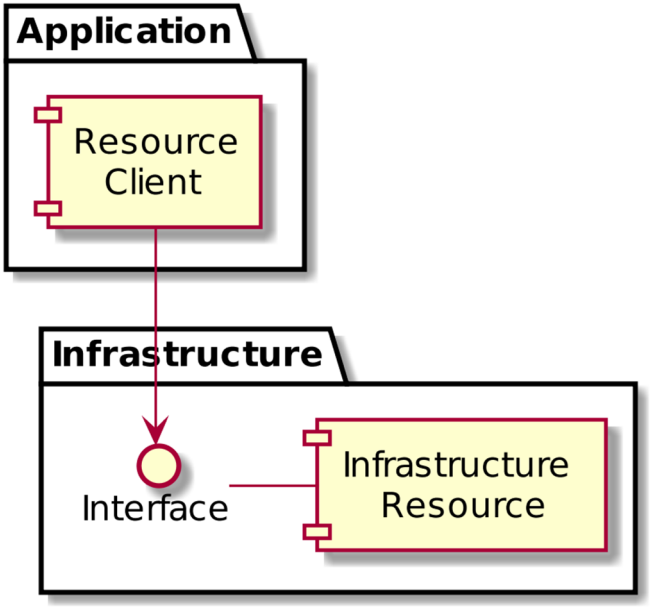

Fig. 5.1 - Resource usage modeling

To enable testing, those URLs and domains should neither be hard-coded nor packaged into the deployed application in any way, so as to be able to change them without redeploying.

Java EE JNDI resources

Java EE application servers already provide references on resources through their Java Naming and Directory Interface (JNDI) resources tree:

- Developers reference resources through their JNDI locations, that are compiled and thus stay the same throughout different environments
- Server administrators bind physical resources to those locations, and of course those resources are different from environment to environment. For more information on JNDI, please check the JNDI Oracle trail (http://docs.oracle.com/javase/tutorial/jndi/).

In pure Java, not hard-coding resource references is typically achieved by using Properties (http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html) file(s). Properties are plain key value pairs, formatted as `key=value`. This is a basic example of reading such a file:

Code 5.1 - Reading from a Properties file

```
1  import java.io.*;
2  import java.util.Properties;
3  import java.util.logging.Logger;
4
5  public class LoadProperties {
6
7      private static final Logger LOGGER =
8              Logger.getLogger(LoadProperties.class.getName());
9      private Properties properties = new Properties();
10
11     public LoadProperties() {
12         File file = new File("/path/to/file.properties");
13         try (FileInputStream stream = new FileInputStream(file)) {
14             properties.load(stream);
15         } catch (FileNotFoundException e) {
16             LOGGER.severe("Properties file does not exist");
```

```
17              throw new RuntimeException(e);
18          } catch (IOException e) {
19              LOGGER.warning("Could not close Properties file");
20          }
21      }
22
23      public String getValueOf(String key) {
24          return properties.getProperty(key);
25      }
26  }
```

Adding an indirection level

In the previous sample, the Properties file path is hard-coded. In most cases though, a second indirection layer is required to have a path dependent on the context (testing, development, production, etc.). This becomes a requirement when the developer machines Operating System is different from the deployed environments OS. There are many ways to achieve this, but the most common way is through a System Property: the Virtual Machine is started with an agreed-upon System parameter *e.g.* `-Dapp.properties=/path/to/file.properties`.

In this case, the following snippet should replace line 13 above:

```
12  String path = System.getProperty("app.properties",
13                                    "/default/file.properties");
14  File file = new File(path);
```

1.1.2 Setting up and cleaning data

When integration testing resources have state (*e.g.* filesystems, databases, JMS queues, etc), data sometimes has to be set up. There are two different cases:

- Repository data that is to be set up and never (or very rarely) removed. For example, a list of countries

- Contextual data that provides data to be manipulated. For example, emails to test the POP3 receiving feature

Most of the time, sample data is set up before running tests and all data cleaned after the run. But, if a test fails, and the resource state has been wiped clean, there is no clue as to the reason why it failed. Thus, it is much more convenient to **first clear data then set sample data during initialization** and do nothing during clean up (at least nothing regarding data) to keep the state if needed.

1.2 System Time integration

Time-dependent components are among the hardest ones to test, since time is something that cannot easily be mocked/faked/stubbed for testing purposes. Fortunately, there are a couple of answers to this:

Manually
: Manually changing the system time is a possible option when testing manually. However, this is completely out of the question in regard to automated testing.

Compatible design
: In Java, time usage is based on `System.currentTimeMillis()` but there is no counterpart programmatic setter for this getter. Directly using this API couples the calling code to the system clock. In order to improve testability, the call should be embedded inside an abstraction, with a default implementation using `System.currentTimeMillis()`. Here is a proposed design:

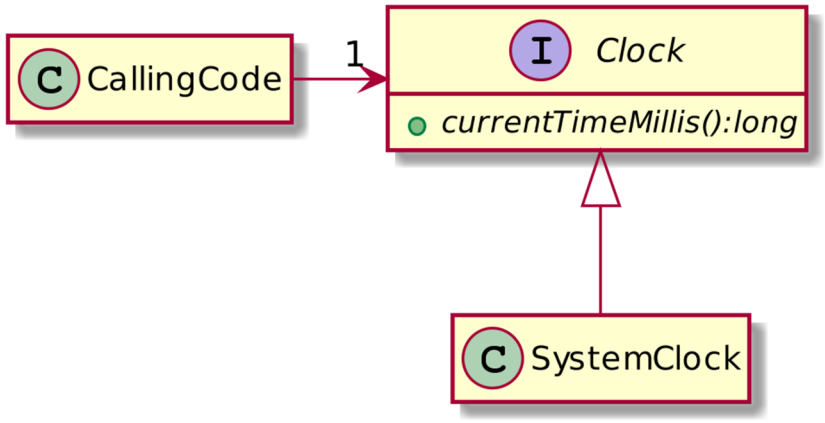

Fig. 5.2 - Design compatible with testing

This way, the `Clock` can be replaced by a Mock in tests.

Joda Time

Joda Time is a library that aims to provide a better `Date` class, but offers a whole API for time management (for happy Java 8 users, it is integrated into the `java.time` package). It also offers a wrapper around `System.currentTimeMillis()` out-of-the-box: by default, `DateTimeUtils.currentTimeMilli` delegates to the former. However, unlike the native API, Joda Time's can be plugged with alternative implementations.

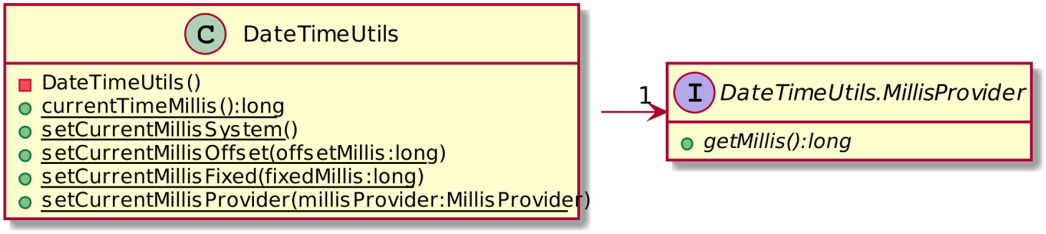

Fig. 5.3 - Joda Time API for system time testing

Alternative implementations include:

- A fixed time implementation with `DateTimeUtils.setCurrentMillisFixed(long fixedMillis)`

- `DateTimeUtils.setCurrentMillisOffset(long offsetMillis)` provides an offset time implementation *e.g.* a fixed time difference with the system time.

Getting back to using the system time is achieved by calling `DateTimeUtils.setCurrentMillisSystem()`. If the provided alternative are not enough, a custom `MillisProvider` implementation can be developed.

Not Invented Here syndrom

Using design option #2 above shows off one's design skills and lets different classes use different system times. However, such manual development takes time and brings nothing to the table in comparison to Joda Time. Besides, requirements to have different system times is not common (to say the least). In most cases, going the Joda Time way is the most worthwile; chances are it will already be used in the project anyway.

1.3 Filesystem integration

Some applications expect file(s) as input(s) and produce file(s) as output(s); these are commonly known as **Batch**-type applications. The goal of this section is to describe how to provide the former and to assert the existence and content of the latter.

Before Java 7, files were handled through the `java.io.File` class. Here is a class dedicated to reading from and writing to files using this legacy class.

Code 5.2 - Legacy file reading/writing features

```
import java.io.*;

public class LegacyFileComponent {

    public void writeContent(String content, File file) {

        if (file.isDirectory()) {
            throw new RuntimeException("Cannot write content to directory");
        }

        if (!file.exists()) {
            try {
                file.createNewFile();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        } else if (!file.canWrite()) {
            throw new RuntimeException("Cannot write to file");
        }

        try (FileWriter writer = new FileWriter(file)) {
            writer.write(content);
            writer.flush();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public String readContent(File file) {

        if (file.isDirectory()) {
            throw new RuntimeException("Cannot read directory content");
        }

        if (!file.canRead()) {
            throw new RuntimeException("Cannot read file");
```

```
        }

        try (FileReader reader = new FileReader(file)) {
            StringBuffer buffer = new StringBuffer();
            int ch;
            while ((ch = reader.read()) != -1) {
                buffer.append((char) ch);
            }
            return buffer.toString();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Given this code, testing requires the injection of `File` instances. Since `File` depends on the file system, referencing a real file is necessary. This is possible by using the `java.io.tmpdir` system property that points to the system temporary directory (it should be writable on most if not all systems with no privileges). Testing code would look something like the following:

Code 5.3 - Testing legacy `File`

```
@Test
public void read_file_should_retrieve_content() throws Exception {

    String dir = System.getProperty("java.io.tmpdir");
    File file = new File(dir, "dummy.txt");

    file.createNewFile();
    FileOutputStream fos = new FileOutputStream(file);
    fos.write("Hello world!".getBytes());

    String content = legacy.readContent(file);

    assertEquals(content, "Hello world!");
}
```

Things are easier since Java 7, as the API offers the new `java.nio.file.Path` interface to provide an abstraction over the file system. The previous component can be replaced with this one while still keeping the same test code. Just replace the `File` parameter with a `Path`, this is easily achieved with `file.getPath()`.

Code 5.4 - New file reading/writing features

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;

public class NewestFileComponent {

    public void writeContent(String content, Path path) {

        if (!Files.exists(path)) {
            try {
                Files.createFile(path);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }

        if (Files.isDirectory(path)) {
            throw new RuntimeException("Cannot write into directory");
        }

        if (!Files.isWritable(path)) {
            throw new RuntimeException("No write permissions");
        }

        try {
            Files.write(path, content.getBytes());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

```

```
32      public String readContent(Path path) {
33
34          if (!Files.exists(path)) {
35              throw new RuntimeException("File doesn't exist");
36          }
37
38          if (Files.isDirectory(path)) {
39              throw new RuntimeException("Cannot read from a directory");
40          }
41
42          if (!Files.isReadable(path)) {
43              throw new RuntimeException("No read permissions");
44          }
45
46          try {
47              List<String> lines =  Files.readAllLines(path);
48              StringBuffer buffer = new StringBuffer();
49              lines.stream().forEach(line -> buffer.append(line));
50              return buffer.toString();
51          } catch (IOException e) {
52              throw new RuntimeException(e);
53          }
54      }
55  }
```

Stubbing files

Stubbing file behavior when using either `File` or `Path` is possible, but it achieves nothing since the behavior that needs to be tested is the one to be stubbed.

1.4 Database integration

Different Java projects use various persistence strategies (SQL vs NoSQL), different database vendors and different ways to access their persistence store(s).

In the SQL realm, RDBMS are standardized enough to provide a common abstraction layer so there are many more ways to access them from Java. From the oldest to the most recent, these are:

- Plain old Java DataBase Connectivity, JDBC
- EJB 2.x Entity Beans, CMP & BMP
- Java Data Objects (http://db.apache.org/jdo/index.html), JDO
- Java Persistence API (http://docs.oracle.com/javaee/5/tutorial/doc/bnbpz.html), JPA
- Hibernate (http://hibernate.org/orm/)
- EclipseLink (http://projects.eclipse.org/projects/rt.eclipselink)
- MyBatis (http://mybatis.github.io/mybatis-3/)
- Spring Data (http://projects.spring.io/spring-data/)
- jOOQ (http://www.jooq.org/)

Spring Data and Hibernate OGM are the frameworks that bridge the traditional SQL realm with newer NoSQL datastores.

In general, for NoSQL solutions, the vendor offers both the store and the connector(s) to access it. For example, MongoDB (http://www.mongodb.org/) is fully under the vendor control and consists of the persistence store itself, a command-line interface and a Java driver. For the record, there is also a third-party Jongo (http://jongo.org/) driver.

1.4.1 SQL Database integration

Regarding Database Integration Testing, a benefit of SQL standardization is that the following section applies to any database product.

1.4.1.1 Datastore environment strategies

Integration Testing with a data store cannot use a single shared instance as is the case for deployed environments (Development, Q&A, Production, etc.). There is probably more than one developer, and they each need to test on their own database, without the possibility of stepping on their colleagues toes. This requires each developer to have a dedicated database. Basically, there are 3 different strategies to address this:

In-memory database

In-memory databases are dedicated data stores, running in-memory on the developer machine. Available in-memory databases include Apache Derby (http://docs.oracle.com/javadb/10.8.3.0/getstart/index.html) (also released by Oracle under the name "Java DB"), HSQLDB (http://hsqldb.org/) and H2 (http://www.h2database.com/). In-memory DBs greatest advantage are their ability to be set up (and discarded) quickly and easily, thus making **test execution completely independent from infrastructure**.

Using a in-memory database for testing means handling the mismatch between the testing DB and the deployed DB(s) capabilities. At a minimum, the portion of SQL syntax used should stick to the syntax supported by both, which usually limits development options.

All above referenced in-memory databases also support persistent storage on the file system. Then, in case of failure, the DB state can be checked after test execution. Since it is very easy to configure - with the only requirement a change in the JDBC URL - there is no reason not to use this. For example, H2 persistent file storage can be configured with a URL such as `jdbc:h2:[<path>]/<databaseName>`.

H2 competitive advantages

In most cases, H2 should be the preferred in-memory database:

1. It supports a compatibility mode that makes it understand some proprietary SQL idioms from major RDBMS vendors. Although this has to be checked on the deployed DB as early as possible, it relaxes somewhat constraints on the SQL syntax used.
2. Its JAR contains an embedded webapp that allows to connect to any JDBC-compatible database, complete with schema browsing and interactive querying.
3. Finally, H2 documentation is top-notch. Just have a look at http://www.h2database.com/html/features.html

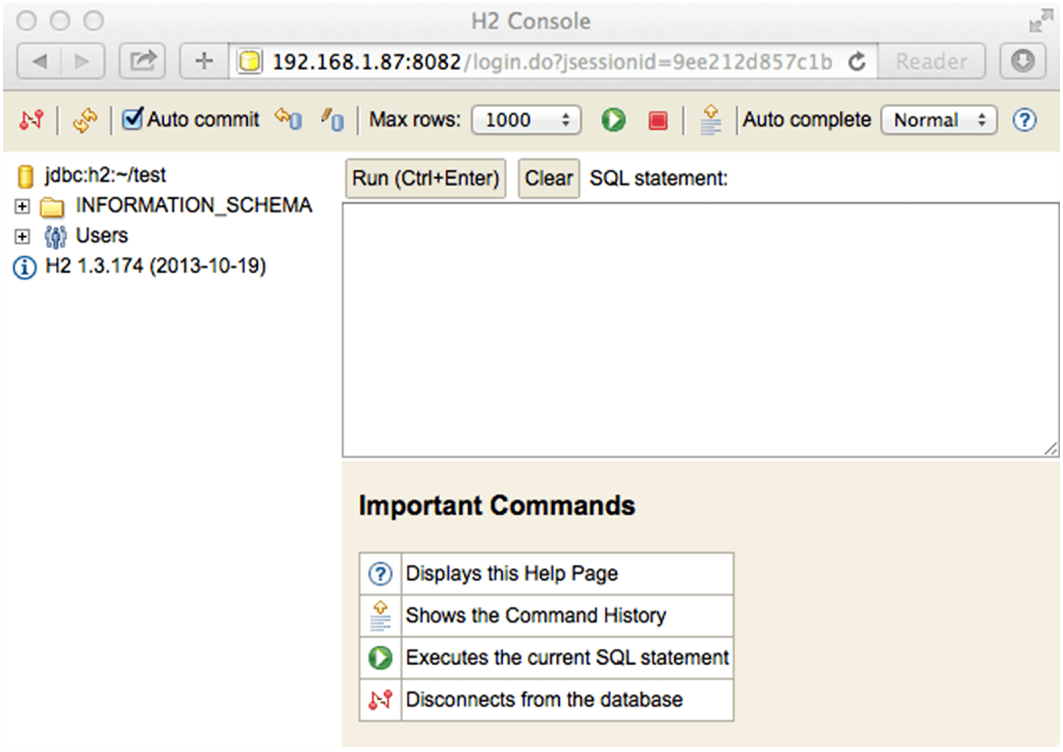

Fig. 5.4 H2 embedded console

Persistent or in-memory?

So called in-memory databases can run entirely in memory **or** be persisted on the filesystem. Each option has its advantages: with in-memory, there is no need to clean up the database at the beginning of a new test suite, while the persisten option enables to check the database state.

Local database

With a local database, each developer installs a copy of the deployed RDBMS on his local development machine. This strategy guarantees 100% compatibility with the deployed platform, at the cost of always keeping the local software version synchronized with the deployed one. Moreover, software installation becomes a prerequisite as it cannot be automated (or very painfully) during application build.

Licensing for local installation

With Open Source RDBMS *e.g.* MySQL or PostgreSQL, an unlimited number of copies on an unlimited number of machines can be installed free of charge. This is not the case with commercial products such as Oracle. The good news: Oracle provides Oracle Database Express Edition *aka* Oracle XE, a database that can be freely installed on all developer machines; the bad news, it is *based* on the commercial product but is not the same product and therefore compatibility issues have to be tested (albeit to a lesser extent than with another different database).

Single remote database per developer

This strategy is about creating a dedicated database per application and per developer. Following this strategy mandates that the Database Administrator(s) has to create one database per developer on the deployed platform (in general, the deployed development environment) and each developer to use it for their tests. It is the best strategy to ensure the exact same behavior during testing **and** in production. On the downside, it either require developers to have admin rights on a database or a working process enrolling the DBA.

Oracle schemas

Oracle Database allows you to have more than one schema under the same JDBC URL. When using it, instead of having one database per developer it is advised to use one dedicated schema instead - this is much easier to set up.

Choosing one strategy over the others really is a matter of context. If you enjoy a good relationship with the DBAs in the organization and plenty of disk space is available, you should probably go for a single remote database. If the project is Open Source and anyone can get the sources and execute the tests, you should use in-memory databases.

All the aforementioned strategies require the same flexibility: to be able to configure the mapping between a developer and a URL (or a schema) during test runs. There

are different ways to achieve this, but the primart way should be to use a properties file, as with file system integration.

1.4.1.2 Data management with DBUnit

From a database integration point of view, requirements are twofold:

1. Put the database in a specific state during initialization
2. Assert the database is in an expected state at the end of the test

For example, for testing the whole order process, initialization has to put customers and products in setup, and after it has run, we check a new order line has been written in the database.

[DBUnit (http://www.dbunit.org/)] is a Java framework that dates back to 2002, has seen no code-related activity since January 2013 and is based on JUnit v3.x. However, it is the only one of its kind and offers both the features required above.

Creating datasets

In DBUnit, datasets can be either created "by hand" or exported from an existing database; they both mimic the database structure. They are available in two different XML file formats, standard and flat.

Code 5.5 - Standard file format sample

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <table name="TABLE_1">
    <column>COL_0<column>
    <column>COL_1<column>
    <column>COL_2<column>
    <row>
      <value>a</value>
      <value>b</value>
      <value>c</value>
    </row>
  </table>
  <table name="TABLE_2">
    <column>COL_A<column>
    <column>COL_B<column>
    <row>
      <value>d</value>
      <value>e</value>
    </row>
    <row>
      <value>f</value>
      <null />
    </row>
  </table>
</dataset>
```

Code 5.6 - Flat file format sample

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <TABLE_1 COL_0="a" COL_1="b" COL_2="c" />
  <TABLE_2 COL_A="d" COL_B="e" />
  <TABLE_2 COL_A="f" />
</dataset>
```

As can be seen, standard format is much more verbose but its Document Type Definition grammar is the same across all databases:

Code 5.7 - Standard file format DTD

```
<!ELEMENT dataset (table+) | ANY>
<!ELEMENT table (column*, row*)>
<!ATTLIST table name CDATA #REQUIRED>
<!ELEMENT column (#PCDATA)>
<!ELEMENT row (value | null | none)*>
<!ELEMENT value (#PCDATA)>
<!ELEMENT null EMPTY>
```

On one hand, this makes it reusable from test suite to test suite; on the other hand, this also makes it almost useless, as it will not catch any table or attribute mistyping in the XML. To do that, it is necessary to use flat files **and** create the DTD. If by chance the database schema already exists, this snippet (http://www.dbunit.org/faq.html#generatedtd) connects to the database, reads the schema and writes it in a DTD file. It can then simply be referenced by the XML.

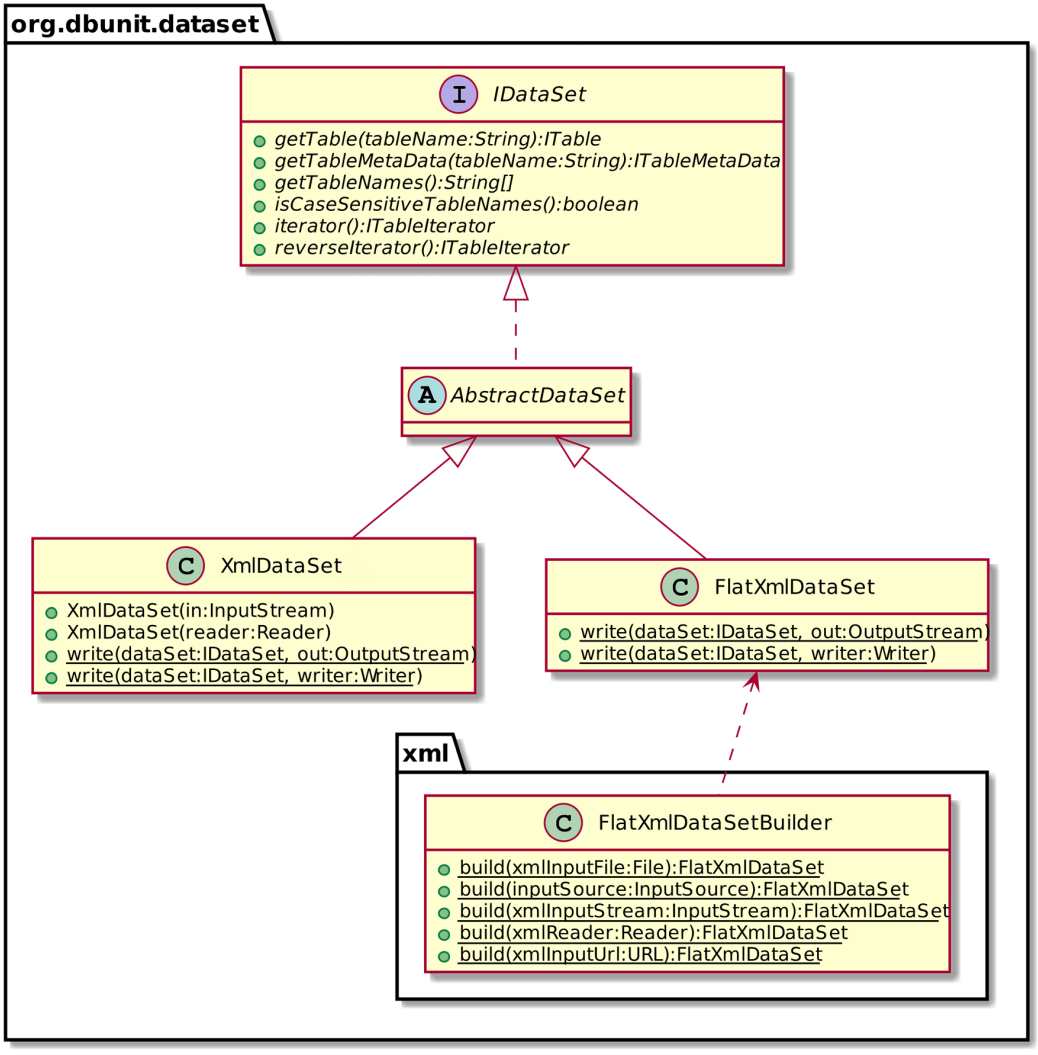

Fig. 5.5 - Dataset class hierarchy

Flat or standard format?

Flat format should be preferred over standard format as it is more concise and easier to write. However, if the schema seldom changes, it is better to create a DTD; if changes are more frequent, a DTD will usually bring more pain than it's worth.

Whether using standard or flat XML files, exporting a sample dataset from an existing database can be achieved with the following snippet: http://www.dbunit.org/faq.html#extract. In essence, it shows how to connect to the database, execute a query and write results and dependent the results in an XML file.

Keep datasets small and dedicated

Not only are datasets a bore to create, they become complex to handle when they grow in size, for various reasons: not `NULL` columns, insertion order with foreign key constraints, etc. Therefore, it is well advised to have small dedicated datasets for each testing use-case to let you better manage them. Naming of the dataset should be done in accordance with the class name.

```
public class ThisIsAnExampleTest {

   @Before
   protected void setUp() throws Exception {
       IDataSet dataset = new FlatXmlDataSetBuilder()
           .build(new File("thisIsAnExampleTest.xml"));
       // Do something with dataset
   }
}
```

Inserting datasets

Handling of datasets requires a DBUnit `DatabaseOperation` instance. The `DatabaseOperation` interface has a single method `execute(IDatabaseConnection, IDataSet)`. Implementations of those are found as constants of the same class. Among them, some are of particular interest:

- `DELETE` deletes the dataset (and only the dataset) from the database
- `DELETE_ALL` deletes all data from the database, but **only from tables referenced in the dataset**
- `INSERT` inserts the dataset into the database
- `CLEAN_INSERT` is an ordered composite operation of `DELETE_ALL` then `INSERT`

Given that test data should be kept after test execution, it is advised to use only `CLEAN_INSERT`. This way, data put in reference tables (such as zip codes, countries, etc.) will not be erased.

True clean inserts

By using `CLEAN_INSERT`, only data from tables referenced in the dataset will be removed. However, some tests may create data in other tables, and this will not be removed. If they are to be tested, it may cause problems in assertions. Thus, it might be worthwile to create a custom `DatabaseOperation` implementation that drops data from **all relevant** tables (of course, reference tables are excluded).

Code 5.8 - True clean inserts possible implementation

```
1  import org.dbunit.DatabaseUnitException;
2  import org.dbunit.database.IDatabaseConnection;
3  import org.dbunit.dataset.IDataSet;
4  import org.dbunit.operation.DatabaseOperation;
5
6  import java.sql.*;
7
8  public class TrueDeleteAllOperation extends DatabaseOperation {
9
10     @Override
11     public void execute(IDatabaseConnection iconn, IDataSet dataSet)
12           throws DatabaseUnitException, SQLException {
13         StringBuffer sql = new StringBuffer("DELETE FROM X; ");
14         sql.append("DELETE FROM Y; ");
15         sql.append("DELETE FROM Z; ");
16         // Append other tables if necessary
17         Connection connection = iconn.getConnection();
18         PreparedStatement statement =
19             connection.prepareStatement(sql.toString());
20         statement.executeBatch();
21     }
22 }
```

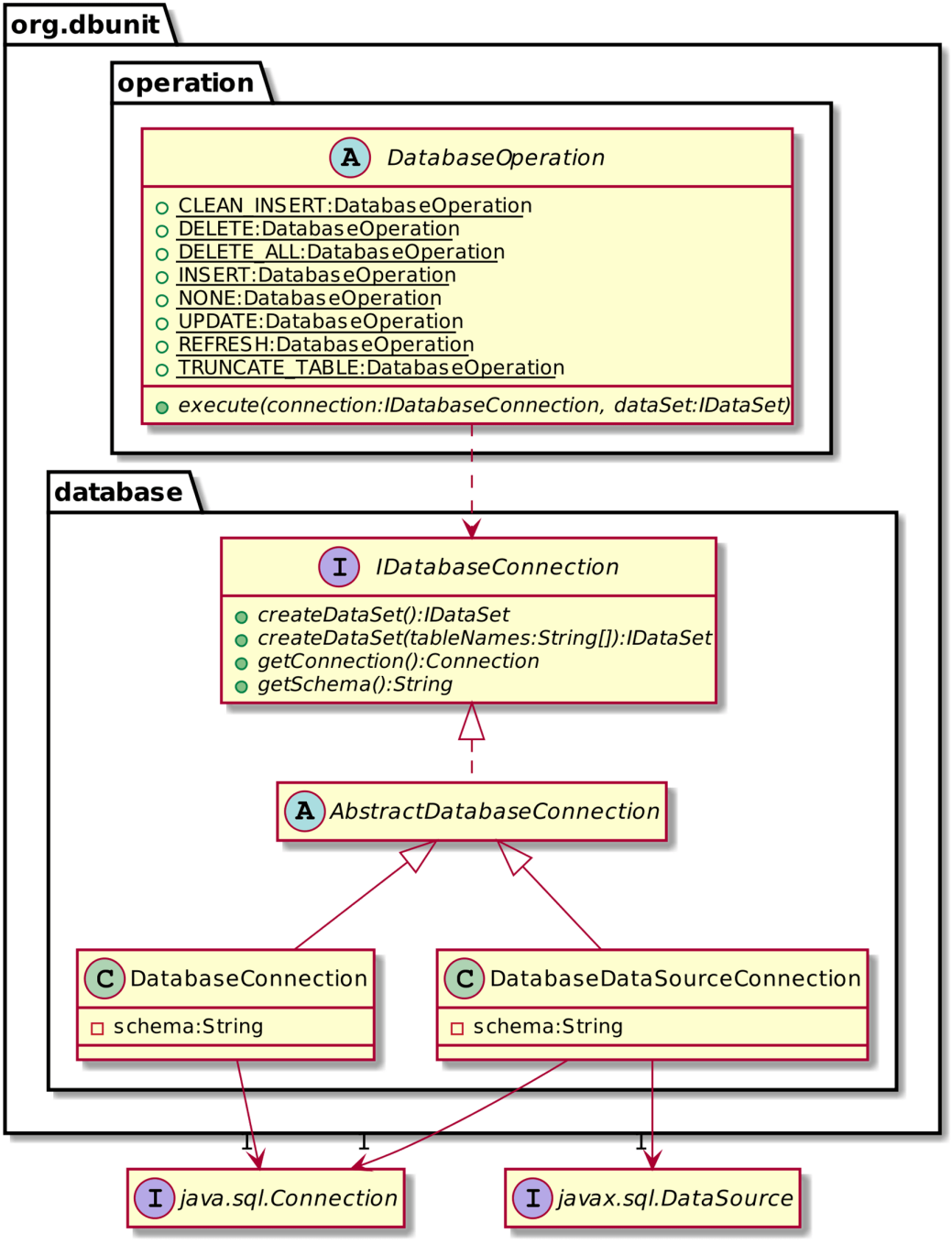

Fig. 5.6 - Connection and operation class hierarchy

Asserting datasets

Finally, when a test has run its course, the database has to be in the expected state. Doing that manually would require connecting to the database, querying tables and checking data line by line.

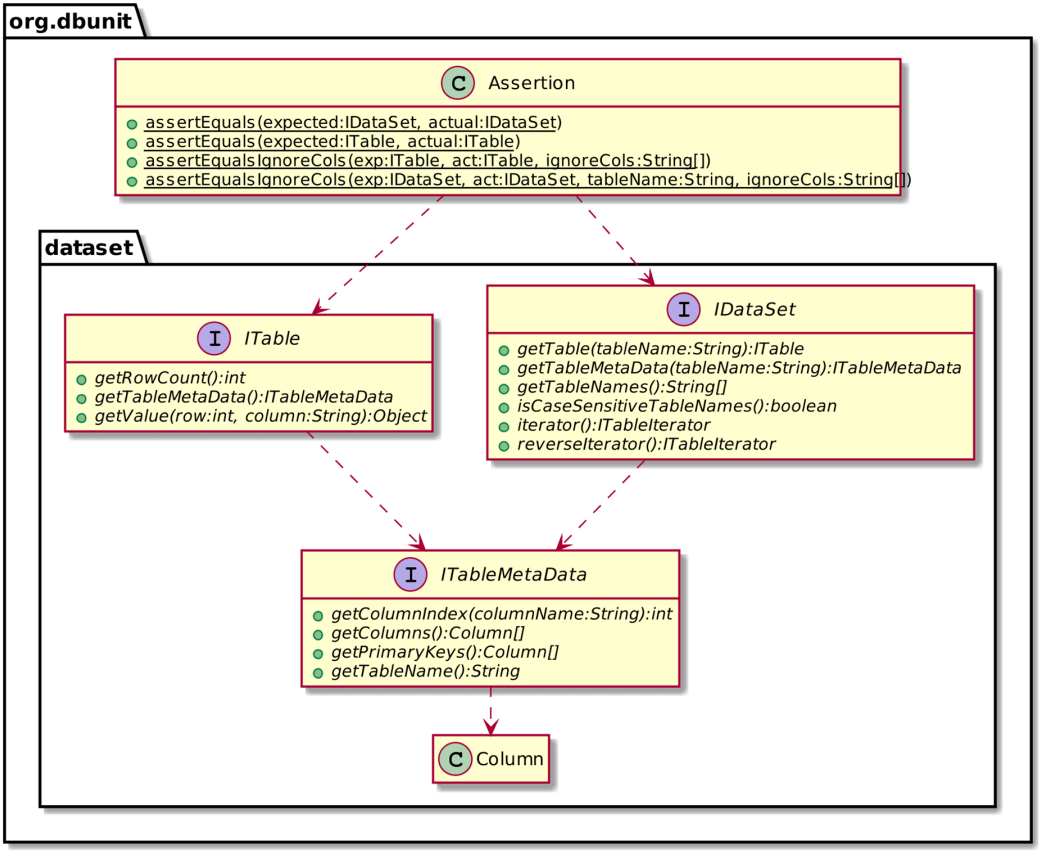

Fig. 5.7 - Assertion class hierarchy

DBUnit is able to handle all of that for us by:

1. Caching the connection in the `IConnection` implementation
2. Creating the dataset from the connection through the `createDataSet()` method
3. Providing an `Assertion` class that can compare both dataset and table contents.

Imagine a full-fledged example to test the former `OrderService`. This would require setting up reference customers and products, passing the order, then checking there is another new order. Let's use H2 as a persistent database (file-based) as it shows more complex setup logic.

Code 5.9.1 - DBUnit comprehensive sample code

```
public class OrderServiceIT {

    private OrderService service;
    private IDatabaseConnection connection;

    @BeforeClass
    protected void setUpBeforeClass() throws Exception {
        Class.forName("org.h2.Driver");
        Path dbPath = FileSystems.getDefault().getPath("target",
                "chapter5.h2.db");
        Files.deleteIfExists(dbPath);
        Connection connection =
                DriverManager.getConnection("jdbc:h2:file:target/chapter6");
        this.connection = new DatabaseConnection(connection);
        DatabaseConfig cfg = this.connection.getConfig();
        cfg.setProperty(PROPERTY_DATATYPE_FACTORY, new H2DataTypeFactory());
        Statement statement = connection.createStatement();
        statement.addBatch("CREATE TABLE PRODUCT(ID BIGINT PRIMARY KEY)");
        statement.addBatch("CREATE TABLE CUSTOMER(ID BIGINT PRIMARY KEY)");
        statement.addBatch("CREATE TABLE ORDERS (" +
                "CUSTOMER_ID BIGINT NOT NULL, " +
                "PRODUCT_ID BIGINT NOT NULL, " +
                "PRIMARY KEY (CUSTOMER_ID, PRODUCT_ID))");
        statement.executeBatch();
        statement.close();
    }
```

This part displays the setup done once before all the tests in this class. The steps are the following:

Line 8

As for any standard direct database connection, the H2 driver is loaded in memory, so JDBC knows which driver to use in order to communicate with H2 databases.

Lines 9-11

Existing database files are removed so test(s) can run from a clean state.

Line 12-13

The JDBC connection to the database is created.

Line 14

The JDBC connection is wrapped in a DBUnit connection wrapper and the latter stored as an attribute for future use.

Lines 15-16

DBUnit's default compatibility is with Apache Derby. DBUnit is explicitly configured to use H2 in order to prevent potential data type mismatch(es) between Derby and H2. Other data type factories for major database vendors are available in the library (see all packages (http://dbunit.sourceforge.net/apidocs/) starting with `org.dbunit.ext`).

Lines 17-25

Database tables used in the tests are (finally) created. This is done through standard SQL DDL statements.

Code 5.9.2 - DBUnit comprehensive sample code

```
1     @AfterClass
2     protected void tearDownAfterClass() throws Exception {
3         connection.close();
4     }
```

It is a good practice to close the connection to the database after the tests have completed.

In-memory databases are only accessible during a JVM run. In a test context, this means this step could well be omitted with no side-effects. However, closing the connection should be a habit and makes the above snippet a template reusable in other scenarios.

Code 5.9.3 - DBUnit comprehensive sample code

```
    @BeforeMethod
    protected void setUp() throws Exception {
        Connection connection = this.connection.getConnection();
        CustomerRepository customerRepository =
                new CustomerRepository(connection);
        ProductRepository prdRepository = new ProductRepository(connection);
        OrderRepository orderRepository = new OrderRepository(connection);
        service = new OrderService(
                customerRepository, orderRepository, prdRepository);
    }
```

As in previous tests, before each test case, the setup initializes the instance under test so as to have no side-effect from state changed in previous test cases.

Code 5.9.4 - DBUnit comprehensive sample code

```
    @Test
    public void order_should_save_order_with_product_and_customer()
            throws Exception {
        FlatXmlDataSetBuilder builder = new FlatXmlDataSetBuilder();
        Path initPath = FileSystems.getDefault().getPath("target",
                "test-classes", "chapter5", "OrderServiceText-init.xml");
        IDataSet dataSet = builder.build(initPath.toFile());
        CLEAN_INSERT.execute(this.connection, dataSet);
        service.order(2L, 2L);
        Path expectedPath = FileSystems.getDefault().getPath("target",
                "test-classes", "chapter5", "OrderServiceText-expected.xml");
        IDataSet expected = builder.build(expectedPath.toFile());
        IDataSet actual = connection.createDataSet(new String[]{"ORDERS"});
        assertEquals(expected, actual);
```

```
15      }
16  }
```

The test itself does the following:

Line 4-8:
The database is set in the expected state. In this case, an existing file is loaded to populate the database. Since there are no other test cases, this could also have been done during setup. Also, it would have been enough to use `INSERT` instead of `CLEAN_INSERT`. However, as for closing the database connection, it makes sense to support more advanced usages, such as having multiple test cases (a likely occurrence in real-life projects).

Line 9:
The method to be tested (at last!)

Line 10-14:
An expected data set is created from a provided file. Then the actual data is read from the database and compared. If both are equal, the test passes; if not, it fails.

Evaluate ROI carefully

This example demonstrates that integration testing Return Over Investment should carefully be evaluated as the setup code is quite long and error prone compared to the feature under test. Trivial data access code should probably not be tested.

1.4.1.3 Setting up data with DBSetup

Using DBUnit to set up data at test initialization requires tons of XML which is:

Time-consuming
DBUnit XML files mimic the database tables structure. Columns are repeated

as XML attributes on each line, so that writing an XML file involves copy-pasting and changing the value.

Also, XML is declarative only and prevents tests and loops.

Error-prone
: XML is not compiled. At best, it can be validated against a Document Type Definition (or an XML schema). Anyone who has ever generated a DTD (http://dbunit.sourceforge.net/faq.html#generatedtd) for DBUnit knows the configuration to use in both tests and IDE with no error takes time.

DBSetup (http://dbsetup.ninja-squad.com/) is an initiative aimed at correcting those drawbacks for inserting data. However, it does not provide ways to validate data after test execution.

1.4.2 NoSQL Database integration

While Integration Testing benefits from SQL standards, NoSQL offers no such standardization: each product might have an associated product dedicated to help with Integration Testing… or not. At the time of this writing, only MongoDB (https://www.mongodb.org/) offers such a testing framework.

1.4.2.1 MongoDB integration

MongoDB is a NoSQL document database. From a developer point of view, once MongoDB has been installed on a server, using this particular database instance looks as follows:

```
MongoClient client = new MongoClient("localhost", 27017);
DB main = mongoClient.getDB("main");
// Now use main to get collections and stuff
```

This snippet assumes a MongoDB instance is running on localhost and listening on port 27017. For testing purposes, this means one has:

- To make sure the MongoDB binaries are available on the system

- If not, download and extract them
- To have permissions to setup MongoDB
- To get it up and running during test startup
- To shut it down during cleanup

It is possible, but not exactly worthwile from a ROI perspective. This process can however be replaced by using Fongo (https://github.com/fakemongo/fongo).

> Fongo is an in-memory java implementation of Mongo. It intercepts calls to the standard mongo-java-driver for finds, updates, inserts, removes and other methods. The primary use is for lightweight unit testing where you don't want to spin up a Mongo process.

Missing features and limitations

Not all MongoDB features are included in Fongo and in some cases, the ones that are included might be limited, so caution is advised. As Fongo and MongoDB are changing on a very rapid basis, it is advised to check the gaps before relying on Fongo.

Fongo usage is very straightforward: it provides a `Fongo` class that offers more or less the same methods as the standard `MongoClient` but the former does not share any contract with the latter. However, Fongo also provides a `MockMongoClient` that both inherits from `MongoClient` and wraps a `Fongo` instance.

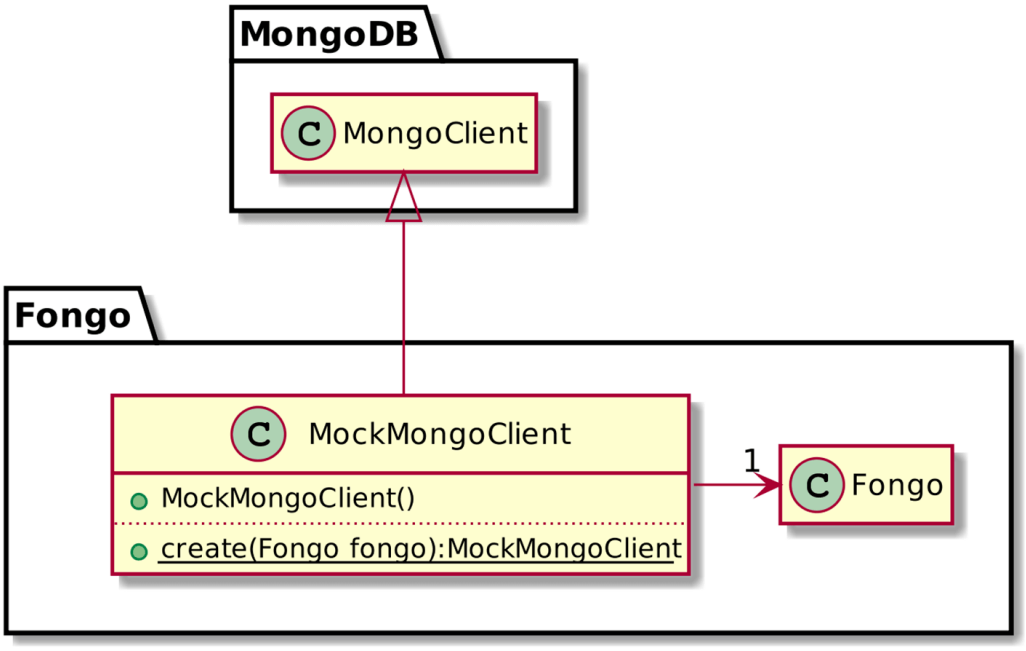

Fig. 5.8 - Fongo core classes

That requires application design to allow for the Mongo client to be injected. The starting point would look something like the following:

Code 5.10 - MongoDB starting design

```
1  import com.mongodb.*;
2
3  import java.net.UnknownHostException;
4  import java.util.*;
5
6  public class MongoDbRepositoryNaiveSample {
7
8      private DB database;
9
10     public MongoDbRepositoryNaiveSample(String host, int port)
11           throws UnknownHostException {
12         MongoClient client = new MongoClient(host, port);
13         database = client.getDB("customerDb");
```

```
    }

    public List<Customer> getAllCustomers() {
        DBCollection collection = database.getCollection("customer");
        DBCursor cursor = collection.find();
        List<Customer> customers = new ArrayList<>();

        try {
            while (cursor.hasNext()) {
                DBObject object = cursor.next();
                Integer id = (Integer) object.get("_id");
                Customer customer = new Customer();
                customer.setId((long) id);
                customers.add(customer);
            }
        } finally {
            cursor.close();
        }

        return customers;
    }
}
```

Updating the design would involve replacing the above constructor code with this:

Code 5.11 - MongoDB testable design

```
    private DB database;

    public MongoDbRepositoryTestableSample(MongoClient client) {
        database = client.getDB("customerDb");
    }
    ...
}
```

Nothing fancy here: simply that the lessons for achieving a testing-friendly design from *Chapter 3 - Test-Friendly Design* have been applied. At this point, testing becomes very easy:

Code 5.12 - MongoDB test

```
import com.github.fakemongo.Fongo;
import com.mongodb.*;
import org.testng.annotations.*;

import java.net.UnknownHostException;
import java.util.List;

import static com.mongodb.WriteConcern.ACKNOWLEDGED;
import static org.testng.Assert.*;

public class MongoDbRepositoryIT {

    private MongoDbRepositoryTestableSample repository;

    private MockMongoClient client;

    @BeforeTest
    protected void setUp() throws UnknownHostException {
        Fongo fongo = new Fongo("main");
        client = MockMongoClient.create(fongo);
        repository = new MongoDbRepositoryTestableSample(client);
    }

    @Test
    public void should_return_all_customers() {
        DB database = client.getDB("customerDb");
        DBCollection collection = database.getCollection("customer");
        for (int i = 0; i < 5; i++) {
            collection.insert(ACKNOWLEDGED, new BasicDBObject("_id", i));
        }

        List<Customer> customer = repository.getAllCustomers();
        assertNotNull(customer);
        assertEquals(customer.size(), 5);
    }
}
```

Here is the explanation of the code.

Line 19
A new `Fongo` instance is created. Notice the constructor requires a `String` parameter: it is the instance's name but plays no role whatsoever.

Line 20
Create the Mongo client required for the class under test at line 24, using the `MockMongoClient.create()` factory method.

Lines 26-30
Set up the database state by adding a new `DBObject` to the Mongo collection.

1.5 eMail integration

Many applications have email-sending capabilities, while some have email-receiving ones. This is generally achieved by connecting to a Simple Mail Transfer Protocol server (respectively POP3/IMAP) and issuing the relevant commands.

A huge Integration Testing issue occurs when application use-cases require to send or receive emails. As with databases, the components responsible for those features can be mocked, but this does not guarantee that they will work correctly.

Fortunately, two fake SMTP servers are available for use during tests:

- Dumbster (http://quintanasoft.com/dumbster/) is a fake SMTP server, where outbound mails are stored for later retrieval and asserts
- Greenmail (http://www.icegreen.com/greenmail/) is an SMTP, POP3 and IMAP server with SSL support all rolled into one

Greenmail has two big advantages over Dumbster: it is feature-complete regarding emails and is available as a Maven dependency (refer to *Chapter 4 - Automated testing* for a refresher if needed) on repo1.

Greenmail main classes are:

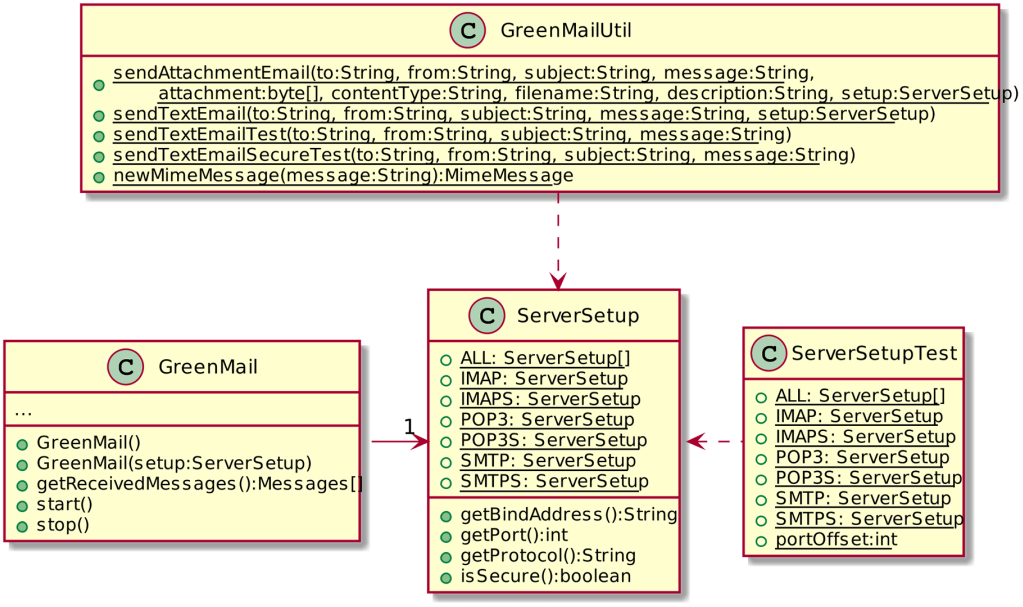

Fig. 5.9 - Green mail core classes

Greenmail is Grails favorite

Greenmail is also available as a plugin for the Grails platform.

The Greenmail root class is `GreenMail` which represents the email server itself. It can respectively be started and stopped with its `start()` and `stop()` methods.

Port numbers restriction

On *nix systems, launching processes that bind ports below number 1024 requires root privileges. Calling `start()` with no arguments calls `start(ServerSetup.ALL)`, using default ports (*e.g.* 25 for SMTP, 11P for POP3, etc.) that are below this limit. In order to be portable across all Operating Systems, one should call `start(ServerSetupTest.ALL)` instead, that set all default protocol ports to their default value **plus** the `ServerSetupTest.offset` value (which is settable).

1. To get the messages received by the GreenMail fake server, call the `receiveMessages()` method on the `GreenMail` instance

Code 5.13 - Receiving messages with Greenmail

```
@BeforeClass
protected void setUpBeforeClass() {
        greenMail = new GreenMail();
        greenMail.start();
}

@AfterClass
protected void tearDownAfterClass() {
        greenMail.stop();
}

@Test
public void send_should_send_mail_message() throws Exception {
        sender.send("Hello world!", "This is a dummy message",
                "from@dummy.com", "to@dummy.com");
        Message[] messages = greenMail.getReceivedMessages();

        assertNotNull(messages);
        assertEquals(messages.length, 1);

        Message message = messages[0];
        Address[] froms = message.getFrom();

        assertNotNull(froms);
        assertEquals(froms.length, 1);
        assertMessageEquals(froms[0], "from@dummy.com");
}
```

The previous code gets the messages from the fake server, checks the number of received messages, get the single message and then checks the sender. Other data could also be checked (sender, subject, body).

2. To send messages to the Greenmail fake server, call the static `GreenMailUtil.sendTextEmailTest()` method (or its secured protocol ports

counterpart `GreenMailUtil.sendTextEmailSecureTest()` instead)

Code 5.14 - Sending messages with Greenmail

```
@BeforeClass
protected void setUpBeforeClass() {
        greenMail = new GreenMail();
        greenMail.setUser("test", "test");
        greenMail.start();
}

@AfterClass
protected void tearDownAfterClass() {
        greenMail.stop();
}

@Test
public void receive_should_receive_all_mail_messages() throws Exception {
        GreenMailUtil.sendTextEmailTest("to@dummy.com", "from@dummy1.com",
                "Hello world!", "This is a dummy message");
        GreenMailUtil.sendTextEmailTest("to@dummy.com", "from@dummy2.com",
                "Hello world!", "This is another dummy message");
        Message[] messages = receiver.receive("test", "test");

        assertNotNull(messages);
        assertEquals(messages.length, 2);
}
```

The above code uses Greenmail to send two dummy email messages.

Reuse the Fake throughout your tests

As for other infrastructure resource Fakes, it is advised to keep the Fake instance up throughout the test suite for even Fakes have startup cost. TestNG is definitely your friend here as the Fake can be started in any of the lifecycle hooks depending on the context.

1.6 FTP integration

Some applications require interacting with an FTP server, either uploading or downloading files. As for other components interacting with infrastructure resources, FTP-responsible components have to be tested by using the techniques already seen above.

In this area, the MockFTPServer (http://mockftpserver.sourceforge.net/) project is a great library that offers two different capabilities depending on the entry-point class used:

1. `org.mockftpserver.fake.FakeFtpServer` represents a fake FTP server
2. `org.mockftpserver.stub.StubFtpServer` is an abstraction over an FTP server that can be stubbed with behavior required for testing

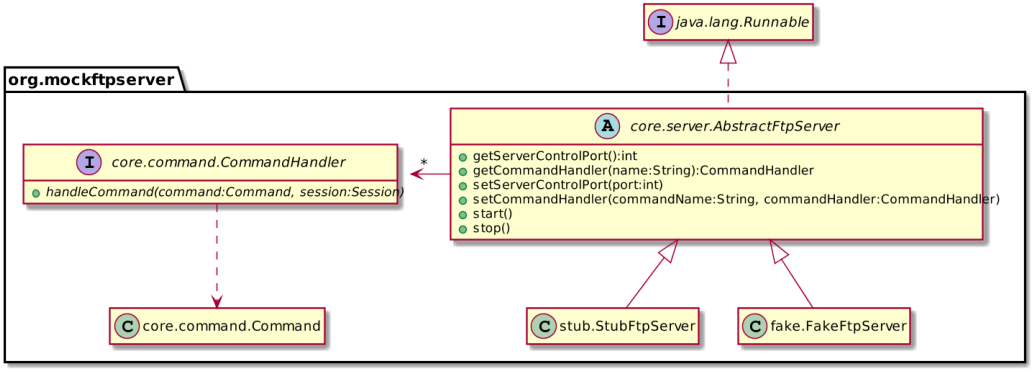

Fig. 5.10 - MockFTPServer root classes

As with the Greenmail server, the server's lifecyle can be managed through the `start()` and `stop()` methods.

Port numbers restriction

By default, the server will attempt to bind to port 21 (FTP's default port). As it requires root privileges on *nix systems, it is more than recommended to call the `setServerControlPort()` method with a port number higher than 1024 before starting it.

1.6.1 Fake FTP server

`FakeFtpServer` is one of the two components of MockFTPServer: the Fake server is set a virtual filesystem, complete with accounts and permissions.

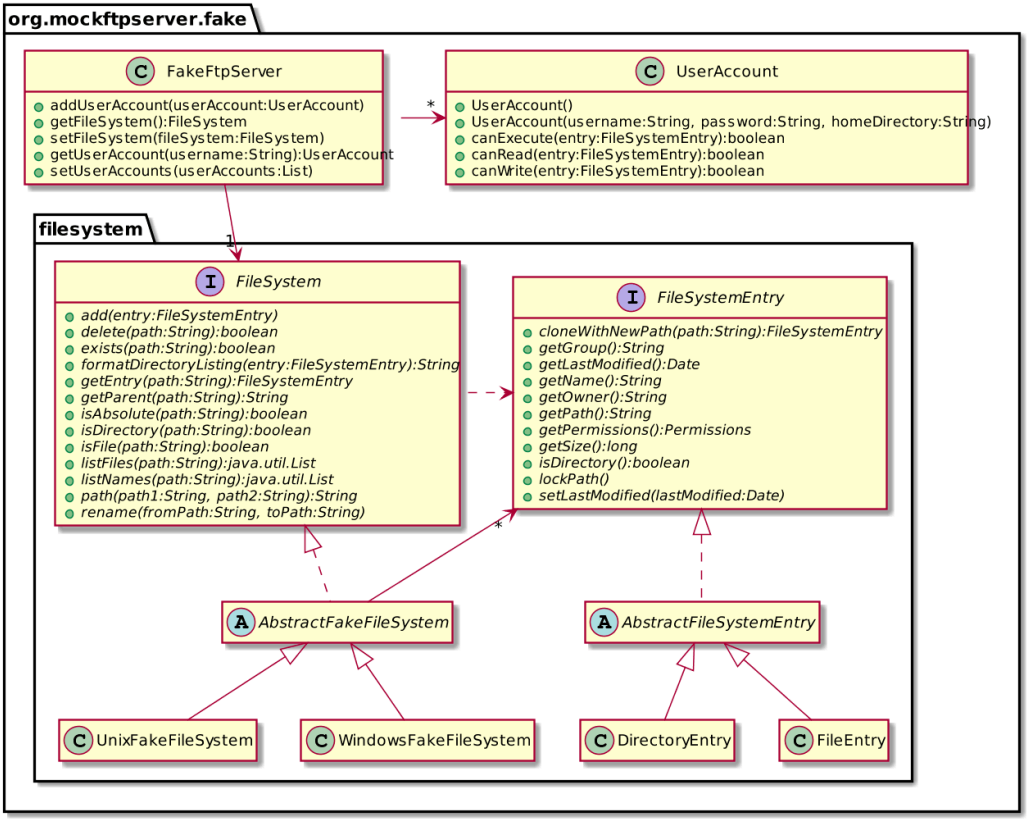

Fig. 5.11 - Fake FTP server class hierarchy

When the server is issued FTP commands, it is the filesystem that is used. The API offers two concrete filesystem implementations, one *nix-like and one Windows-like. The good thing is that since the filesystem is virtual, either of the two implementations can be plugged-in **regardless** of the physical Operating System that the tests are executed on, meaning one should target the production filesystem.

As an example, imagine a component that gets text files from FTP servers and that

needs to be tested. Here is some sample code to achieve that, using the Fake server:

Code 5.15 - Setting up a Fake FTP server

```
@BeforeClass
protected void setUpBeforeClass() {
    client = new FtpGetComponent("localhost", 2021, "test", "test");
        fs = new UnixFakeFileSystem();
    server = new FakeFtpServer();
    server.setServerControlPort(2021);
    server.setFileSystem(fs);
    server.addUserAccount(new UserAccount("test", "test", "/"));
    server.start();
}

@AfterClass
protected void tearDownAfterClass() {
        server.stop();
}

@Test
public void get_file_should_contain_hello_world() throws IOException {
        String tmpDir = System.getProperty("java.io.tmpdir");
        Path path = Paths.get(tmpDir, "dummy.txt");
        Files.deleteIfExists(path);
        fs.add(new FileEntry("/dummy.txt", "Hello world!"));
        client.getTextFile(path.toString(), "/dummy.txt");
        assertTrue(Files.exists(path));
        List<String> lines = Files.readAllLines(path);
        assertNotNull(lines);
        assertFalse(lines.isEmpty());
        assertEquals(lines.size(), 1);
        String line = lines.iterator().next();
        assertEquals(line, "Hello world!");
}
```

The Fake server is good enough for typical FTP server behavior. However, it cannot simulate error behavior. For this, custom-made stubs are required.

1.6.2 Stub FTP server

In order to provide the desired behavior during test execution, it is possible to provide custom `CommandHandlers` that are invoked when calling a specific FTP command.

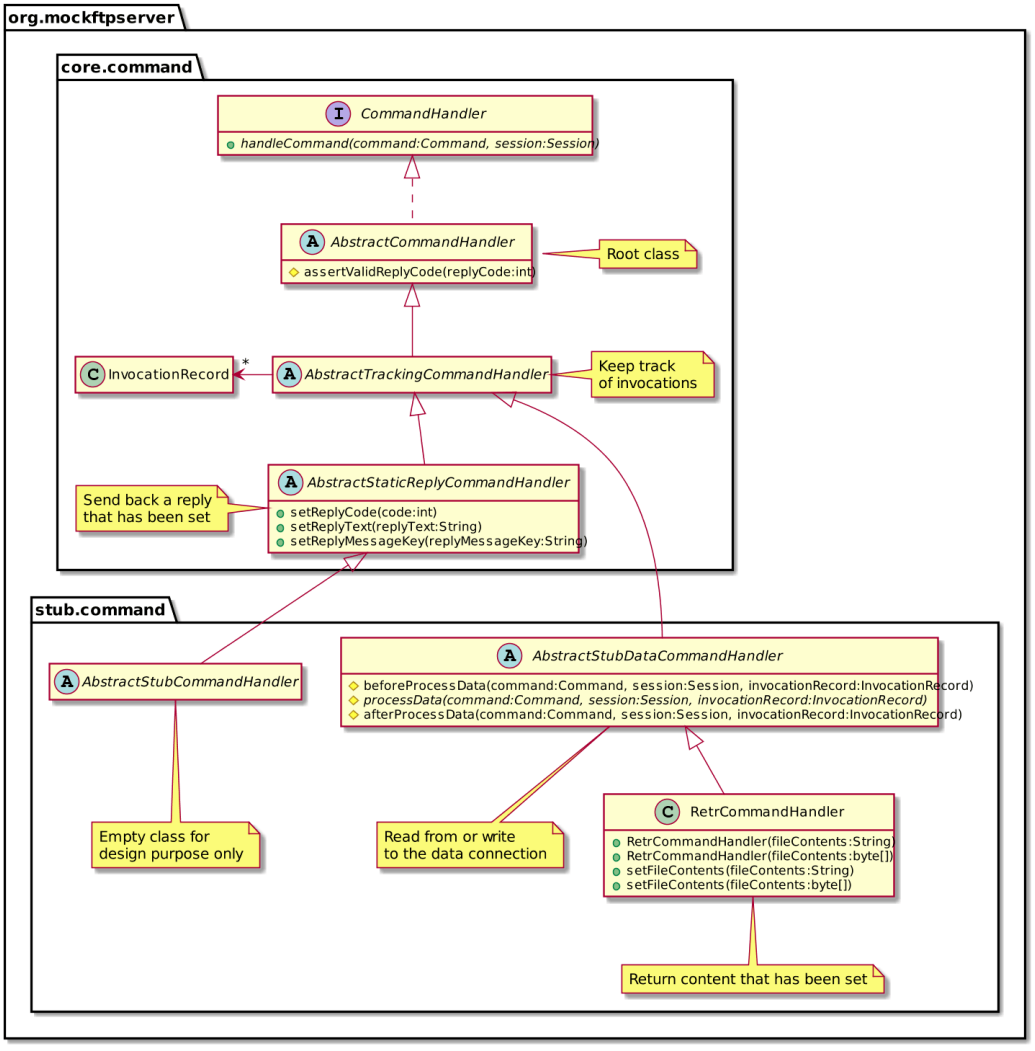

Fig. 5.12 - Command handler class hierarchy overview

In this context, FTP commands are not those that are typed by users on the command line (`ls`, `cd`, etc.) but those of the FTP **protocol** http://en.wikipedia.org/wiki/List_of_FTP_commands.

MockFTPServer provides a default command handler implementation for each FTP command, and there are many. Note that command handler classes are also used internally in the Fake server. The following diagram displays the upper levels of the `CommandHandler` class hierarchy.

In regard to the previous example, instead of using a Fake server with a file added at the root, one could provide a command handler that returns the file on `RETR` commands. Line 28 from the previous code can be replaced with the following:

Code 5.16 - Using the RETR default command handler

```
46      (RetrCommandHandler) server.getCommandHandler(RETR);
47  handler.setFileContents("Hello world!");
48          client.getTextFile(path.toString(), "/dummy.txt");
49          assertTrue(Files.exists(path));
```

Notes:

1. The Stub server does not allow a filesystem to be set.
2. A more advanced (and expensive) setup would be to extend `RetrCommandHandler` and override `processData()` to analyze the parameters of the `Command` argument. If it is `"/dummy.txt"` then return the content, otherwise send the appropriate reply (`500`).

Code 5.17 - A custom command handler

```
import org.mockftpserver.core.command.*;
import org.mockftpserver.core.session.Session;
import org.mockftpserver.stub.command.RetrCommandHandler;

import static org.mockftpserver.core.command.ReplyCodes.READ_FILE_ERROR;

public class AdvancedRetrCommandHandler extends RetrCommandHandler {

    private byte[] fileContents = new byte[0];

    public AdvancedRetrCommandHandler() {}

    public AdvancedRetrCommandHandler(String fileContents) {
        setFileContents(fileContents);
    }

    public AdvancedRetrCommandHandler(byte[] fileContents) {
        setFileContents(fileContents);
    }

    @Override
    protected void processData(Command command,
        Session session, InvocationRecord invocationRecord) {
        String arg = command.getParameter(0);
        if ("/dummy.txt".equals(arg)) {
            super.processData(command, session, invocationRecord);
        } else {
         sendReply(session, READ_FILE_ERROR, null, null, new Object[]{arg});
        }
    }

    @Override
    public void setFileContents(byte[] fileContents) {
        super.setFileContents(fileContents);
        this.fileContents = fileContents;
    }
```

```
37 }
```

At this point, using the handler is a simple as calling `server.set(RETR, new AdvancedRetrCommandHandler())`.

Stub or Fake FTP server?

Having more than one option always forces one to make a choice. In most contexts, it is better to use the Fake server as it a) requires less effort and b) makes code more readable. It is recommended to use the Stub server only if the testing requirements go beyond the Fake's features.

1.7 Summary

This chapter detailed how to test integration with some important infrastructure resources, such as filesystems, system time, databases, email servers and FTP servers, as well as tools that help in doing so.

There is no common lesson here, just to be aware of the Faking tool appropriate for each resource. However, in this day and age, most resources that need integrating with are web services. Those will be covered in the next chapter.