

1 Test-Driven Development (TDD)

Test-Driven Development [1, 4], abbreviated as TDD, is a technique that concerns the development *process*, rather than the structure of the developed *product*. Usually you cannot see in the end product whether TDD was applied or not. As the name Test-Driven Development suggests, testing plays a leading role. In particular, automated test cases are an integral part of delivered software. Furthermore, test cases are developed *before* product code is written. In fact, the development of the product is *driven* by the test cases. Most of the documentation is incorporated in the source code.

We explain **how** to apply TDD in the context of this course and **why** to apply it. We also present some **examples** and **exercises** using the Java programming language and the JUnit testing framework.

2 How: TDD steps

In TDD as applied in this course, development steps are performed in the following order.

1. Gather and analyze requirements to be realized, e.g. use cases or user stories.
2. Decide on which requirement to develop next, and in what module¹ it will appear.
3. Specify the module informally.
4. Specify the module formally, by providing its *interface* and *contract* [2, 3].
5. Create a corresponding unit test module, to hold the automated test cases.
6. Design and implement a (new) test case; document its motivation in the source code. This test case will now fail, lacking a module implementation.
7. Design and implement module features, intended to make the test case pass.
8. Test the new module implementation, and fix defects found (back to step 7).
9. Where relevant, improve the structure of the code (*refactoring*), preserving correctness by running all test cases again (*regression testing*).
10. Repeat from step 6, until the entire module has been implemented and tested.
11. If a defect is detected while using the module *in vivo*, then, first, add a test case to detect that defect *in vitro*, and only then repair the defect.

It is recommended to write only **small fragments** of code at a time, and run all test cases after each change. This applies to both product code and test code.

¹Typically, this involves a new class, but possibly a single method, or a package of related classes.

3 Why: Motivation for TDD

First, let's look at why one should *test* software before delivery.

- When humans produce software (or anything else) of some complexity, it is inevitable that they make mistakes and introduce **defects**. The presence of defects reduces the product's **quality**. The longer a defect remains undetected, the higher the **cost**² of repairing it. Therefore, one should attempt to detect defects as soon as possible. Defects can be detected by *reviewing* intermediate artifacts, such as requirements documents and design documents, and also source code. But reviewing will not catch all defects. Testing is a complementary way of detecting defects, applied after reviewing. Both reviewing and testing are necessary.

Next, let's look at reasons for insisting on *automated* test cases as part of delivered software.

- **Manual testing is tedious, time consuming, error prone, and hard to reproduce.** Hence, it does not invite repeated application of the same test cases. Automated test cases are faster, more accurate, and reproducible.
- Automated test cases also serve as **documentation**; they operationalize the requirements. Note that each automated test case precisely establishes
 - how to invoke the tested feature, e.g., which inputs to provide;
 - how to observe the tested feature, e.g., which outputs to evaluate;
 - how to decide between passing and failing.

The purpose of a test case, in terms of its relationship to the requirements, must be documented separately in a comment.

- Automated test cases serve as a client to the module under construction, and their development may reveal problems with the module's interface and contract early in the development cycle, before releasing it to clients in the production code. Thus, development of automated test cases helps **stabilize interfaces and contracts before releasing them**.
- When defects are present in code that was not exercised in a unit test, it can be hard to localize and repair them, leading to frustrating and unpredictable debugging³ work. Ad hoc debugging work is a waste of time, because it does not help you to do better in the future. And systematic debugging boils down to writing automated test cases anyway. So, why not write them in the first place? Good automated test cases **obviate later debugging**.

²Experimental evidence shows that the cost of repair increases *exponentially* with time elapsed between injection of a defect and its subsequent detection.

³Testing concerns the *detection* of defects; debugging concerns the *repair* of detected defects.

- Automated test cases can be re-run anytime; in particular, they can be re-run after changing the software (*regression testing*). They serve as a **safety net** when *refactoring* the code. That way, one can see to it that production code always works.

Finally, let's look at why it is a good idea to write test code *before* product code. We already decided that test cases are needed, and that they must be automated in the form of program code. Choosing the order may seem like an irrelevant detail, but there are some important benefits to the TDD order.

- Because changes in a module's interface and contract lead to changes in its implementation, it is best to get the module's interface and contract in good shape *before* starting to implement the module. As remarked earlier, automated test cases serve as a client of the module. Therefore, working on the automated test cases will confront you with the module's interface and contract. Development of automated test cases helps **stabilize interfaces and contracts *before* implementing them**. That is why it is a good idea to develop automated test cases *before* implementing the module.
- While developing automated test cases, you familiarize yourself with the module's interface and contract, and that will put you in a better position to fulfil the contract in an implementation. Development of automated test cases **encourages analysis of interfaces and contracts *before* implementing them**. This prevents you from rushing into implementation work unprepared.
- After completing (another part of) the module's implementation, the whole module must be tested. If the automated test cases are completed *before* the module's implementation, then there is no interruption between completing some implementation work and testing it. If a test case would detect a defect, then you still have the implementation details fresh in your mind and can repair the defect quickly, and immediately retest it. Having automated test cases ready *before* implementing a module, allows you to **focus on implementing the module and getting it to work**, without being interrupted by test case development. This is also more satisfactory for a developer: once you have completed some code, you can immediately try it out.

4 TDD for a single function

To illustrate TDD in the small, we consider the development of

(requirement) a function that counts the number of decimal digits in a non-negative integer.

Give an **informal specification**, that is, provide a summary sentence in a *javadoc* comment, and a header (method name, parameter names and types, return type):

```
/**
 * Counts the decimal digits of a number.
 */
public static int countDigits(long n) {
}
```

Elaborate the informal specification, to remove ambiguities and imprecisions:

```
/**
 * Counts the decimal digits of a number.
 * This concerns a non-negative integer, assumed to be
 * written in decimal notation without leading zeroes.
 */
public static int countDigits(long n) {
}
```

Formalize the specification, that is, provide a **contract**⁴:

```
/**
 * Counts the decimal digits of a number.
 * This concerns a non-negative integer, assumed to be
 * written in decimal notation without leading zeroes.
 *
 * @param n the number whose digits are counted
 * @return the number of decimal digits in {@code n}
 * @pre {@code 0 <= n}
 * @post {@code \result = (\min int k; 1 <= k; n < 10 ^ k)}
 */
public static int countDigits(long n) {
}
```

A first test case, with the smallest allowed number:

```
/** Subject Under Test. Only static members are used. */
private static final TDDForCountDigitsMethod SUT = null;

@Test
public void testCountDigits0() {
    System.out.println("countDigits(0)");
    long n = 0L;
    int expectedResult = 1;
    int result = SUT.countDigits(n);
    assertEquals("result", expectedResult, result);
}
```

⁴Here, we do not yet aim for *robustness*.

It fails; in fact, it does not even compile, but that is easy to fix:

```
public static int countDigits(long n) {
    return 0; // to make it compile
}
```

Not surprisingly, it still fails. Here is an implementation that is easy to understand, using a *Linear Search*:

```
public static int countDigits(long n) {
    // Linear Search for smallest k >= 1 with n < 10 ^ k
    int k = 1;
    long p = 10L;
    // invariant: 1 <= k && p == 10 ^ k
    while (! (n < p)) {
        p *= 10;
        ++ k;
    }
    // n < p == 10 ^ k
    return k;
}
```

It passes the first test case. Add another test case, with the largest number for the smallest result, i.e. $n = 9$. We refactor the test cases to reduce code duplication:

```
/**
 * Invokes countDigits(n) and checks result against expectation.
 *
 * @param n the number whose digits are counted
 * @param expectedResult the expected result
 */
private void checkCountDigits(long n, int expectedResult) {
    System.out.println("countDigits(" + n + ")");
    int result = SUT.countDigits(n);
    assertEquals("result", expectedResult, result);
}

@Test
public void testCountDigits9() {
    checkCountDigits(9L, 1);
}
```

It passes. Add one more test case, the smallest number for result 2, i.e. $n = 10$.

```
@Test
public void testCountDigits10() {
    checkCountDigits(10L, 2);
}
```

This also passes. Time for a more systematic series of test cases, which includes all earlier test cases. All smallest and largest numbers with results up to 5:

```
@Test
public void testCountDigitsSmall() {
    long n = 1L;
    for (int r = 0; r <= 5; ++ r) {
        // n == 10 ^ r
        checkCountDigits(n - 1, Math.max(1, r));
        checkCountDigits(n, r + 1);
        n *= 10;
    }
}
```

All these tests pass. However, the implementation with a Linear Search has the risk of producing an overflow, because p is made to exceed n . So, let's add a test case for the largest possible value (the timeout is needed, because overflow⁵ may cause the Linear Search to loop endlessly):

```
@Test(timeout = 1000)
public void testCountDigitsMaxValue() {
    checkCountDigits(Long.MAX_VALUE, 19);
}
```

This test case fails with a timeout. Using the test cases as a safety net, we try to improve the implementation, by decreasing n , rather than increasing p :

```
public static int countDigits(long n) {
    int result = 1;
    while (n != 0) {
        n /= 10;
        ++ result;
    }
    return result;
}
```

This fails for $n > 0$, so the implementation is wrong (a *one-off* error). A fix:

```
public static int countDigits(long n) {
    int result = 1;
    while (10 <= n) {
        n /= 10;
        ++ result;
    }
    return result;
}
```

⁵Java will silently 'overflow' $p \ast = 10$ into the negative numbers.

This passes all test cases, including the largest value, so we can have some confidence in the new implementation.

Exercises Apply TDD to address the following requirements.

1. Introduce a second parameter for the radix r , which was hard coded as 10.
2. Make the function robust by checking that $0 \leq n$ and $2 \leq r$, throwing `IllegalArgumentException` if not. Also test for robustness.

5 TDD for a single class

Here is a summary of the steps for developing a class that will serve as an *Abstract Data Type*. Steps 1 and 2 are as before.

3. Specify the class **informally**:
 - (a) provide a **one-sentence summary** of the class, in a *javadoc* comment;
 - (b) choose an appropriate **class name**;
 - (c) when relevant, choose **generic type parameters** and their **constraints**;
 - (d) elaborate the informal description to obtain a more complete description of the class as a whole, from the perspective of a client of the class.
4. Specify the class **formally**, by providing
 - (a) a **model** defining the abstract set of values, in terms of specification variables, and/or basic queries and basic commands,
 - (b) **public invariants** constraining the model to intended values,
 - (c) **contracts** for its operations (constructors, queries, and commands), in terms of the model.

The result is a class without implementation: no data representation and empty method bodies. Consider putting this in an **abstract class** or **interface**. Note that not all operations have to be added at once.

5. Create a corresponding **unit test** class.
6. Add a new **test case** for an operation.
7. Implement (part of) a **data representation** (or **rep** for short) in terms of **private** instance variables, and also provide a **rep(resentation) invariant** and **abstraction function**. Implement the operation of the preceding step.
8. Test the implemented operation.
9. Where relevant, refactor code to improve the structure.
10. Repeat from step 6 until done.

5.1 Example

To illustrate TDD of a class, we consider the development of

[requirement] an Abstract Data Type to maintain a relation on small integers, in the range from 0 to a given upper bound.

Give an **informal specification**, that is, provide a summary sentence in a *javadoc* comment, and a header (class name; here, there are no generic type parameters):

```
/**
 * An {@code IntRelation} object maintains a relation on small integers.
 */
public abstract class IntRelation {
}
```

We make the class abstract, in order to provide multiple implementations. Elaborate the informal specification, to remove ambiguities and imprecisions:

```
/**
 * An {@code IntRelation} object maintains a relation on small integers.
 * The relation is a subset of {@code [0..n] x [0..n]},
 * where {@code n}, called the extent of the relation,
 * is given in the constructor and is immutable.
 * There are operations to test membership,
 * and to add and remove pairs.
 */
public abstract class IntRelation {
}
```

Formalize the specification⁶, that is, provide a **model** with **public invariants**, in the class javadoc:

```
* Model: subset of {@code [0..extent()] x [0..extent()]}
*
* @inv {@code NonNegativeExtent: 0 <= extent() }
```

And provide **contracts** for one or more operations. Here, we have decided to start with just the constructor, because that already gives us something to test and implement.

```
/**
 * Constructs an empty relation of given extent.
 *
 * @param n extent of the new relation
 * @pre {@code 0 <= n}
 * @post {@code this == [ ] && this.extent() == n}
 */
public IntRelation(final int n) {
}
```

⁶Again, we do not yet aim for robustness.

To test concrete implementations of the abstract class, we define general test cases in an abstract test class that can be reused for each concrete implementation:

```
public abstract class IntRelationTestCases {

    /** Test fixture. */
    protected IntRelation instance;

    /**
     * Sets instance to a newly constructed relation of given extent.
     * (This is a kind of factory method; cf. Factory Method Design Pattern.)
     *
     * @param n extent
     */
}
```

A concrete test class must override `setInstance` (see below). Here is a test case for the constructor. Because there are not yet any queries to inspect the constructed object, we only test whether construction succeeds without throwing anything (`Error` or `Exception`).

```
/** Tests the constructor with small values. */
@Test
System.out.println("constructor(int)");
for (int n = 0; n <= 3; ++ n) {
    assertTrue("isRepOk()", instance.isRepOk());
}
```

The test cannot yet be applied, because we do not have a concrete implementation. We first work on an implementation that uses a two-dimensional array of `boolean` to store the *incidence matrix* of the relation. We provide a javadoc comment, class name, the representation, representation invariants, and abstraction function:

```
/**
 * An implementation of {@link IntRelation} using nested arrays.
 * For relations with a small extent this may work faster.
 * However, relations with a large extent use more memory,
 * even when they are sparse (have few related pairs).
 */
public class IntRelationArrays extends IntRelation {

    /** Representation of the relation. */
    protected boolean[][] relation;

    /**
     * Representation invariants
     *
     * NotNull: relation != null
     * Extent: relation.length == extent()
     * ElementsNotNull: (\forall i; relation.has(i); relation[i] != null)
     * ElementsSameSize: (\forall i; relation.has(i);

```

```

    *   relation[i].length == relation.length)
    *
    * Abstraction function: set of (a, b) such that relation[a][b] holds
    */
}

```

We did not forbid *sharing*, where `relation[i] == relation[j]` for `i != j`. It is not a problem for correctness of the constructor if all `relation[i]` equal the same empty set. But do you see that it will make it harder—but not impossible—to implement `add`? We shall avoid sharing, although the rep invariant allows it.

Next we create a concrete test class for this implementation, overriding `setInstance`:

```

public class IntRelationArraysTest extends IntRelationTestCases {

    @Override
    protected void setInstance(final int n) {
        instance = new IntRelationArrays(n);
    }
}

```

When we run the test case, it fails, because we have not provided a constructor. So, we implement a constructor:

```

public IntRelationArrays(final int n) {
    super(n);
    relation = new boolean[n][n];
}

```

The test case passes.

Next, we intend to implement `isRepOk` to check the representation invariants. Note that developing this method early is valuable to prevent future debugging frustrations. If we would not do it now, and later would run into a problem that concerns the representation, then we would probably spend extra time doing some ad hoc inspections (using `println`), rather than improve the infrastructure.

We first provide an informal specification:

```

/**
 * Checks whether the representation invariants hold.
 */
public abstract boolean isRepOk() throws IllegalStateException;

```

and elaborate it into a formal specification:

```

/**
 * Checks whether the representation invariants hold.
 *
 * @return whether the representation invariants hold
 * @throws IllegalStateException if precondition violated
 * @pre representation invariants hold
 * @modifies None

```

```

    * @post {@code \result}
    */
    public abstract boolean isRepOk() throws IllegalStateException;

```

There are two reasons to throw an exception when the rep invariants do not hold:

1. Just returning **false** is not very informative, whereas the exception can carry more detailed information about the violation.
2. Proceeding with normal operation makes no sense, when a rep invariant fails.

Now, we redefine the constructor test case to invoke `isRepOk`:

```

/** Tests the constructor with small values. */
@Test
    System.out.println("constructor(int)");
    for (int n = 0; n <= 3; ++ n) {
        setInstance(n);
        assertTrue("isRepOk()", instance.isRepOk());
    }

```

Of course, the test fails, and we proceed to implement `isRepOk`

```

@Override
    public boolean isRepOk() {
        if (relation == null) {
            throw new IllegalStateException("relation == null");
        }
        for (int i = 0; i != extent(); ++ i) {
            if (relation[i] == null) {
                throw new IllegalStateException(
                    "relation[" + i + "] == null");
            }
            if (relation[i].length != extent()) {
                throw new IllegalStateException(
                    "relation[" + i + "].length != extent()");
            }
        }
        return true;
    }

```

Fortunately, it passes. Let's move on to introduce a query to retrieve the extent. First, the informal specification:

```

/**
 * Returns the extent of this relation.
 */
    public abstract int extent();

```

And then elaborated into a formal specification:

```

/**
 * Returns the extent of this relation.
 *
 * @return extent of this relation
 * @pre {@code true}
 * @modifies None
 * @post (basic query)
 */
public abstract int extent();

```

Because this concerns a basic query, we cannot provide a formal postcondition.

Now, add a test case for it:

```

/** Tests the extent method with small relations. */
@Test
    System.out.println("extent");
    for (int n = 0; n <= 3; ++ n) {
        setInstance(n);
        assertEquals("size", n, instance.extent());
        assertTrue("isRepOk()", instance.isRepOk());
    }

```

Again, it fails for lack of an implementation. We fix that:

```

@Override
public int extent() {
    return relation.length;
}

```

Test again, ... with success (it is hard to go wrong this way).

In a similar way, we can, one by one, specify, provide a test for, and implement other queries and commands:

```

public boolean isValidPair(int a, int b) {
}

public abstract boolean areRelated(int a, int b);

public abstract void add(int a, int b);

public abstract void remove(int a, int b);

```

See the source code for details.

A disadvantage of using arrays to store the relation is that it uses a lot of memory when the extent is larger. For *sparse* relations, which contain only relatively few pairs, another representation would be better. We briefly investigate an alternative implementation using a `List` of `Sets` from the Java Collection Framework.

First, we introduce a new class for the alternative implementation, with a javadoc comment, class name, representation, representation invariants, and abstraction function (the `import` section is not shown):

```

/**
 * An implementation of {@link IntRelation} using a {@link List} of {@link Sets}
 * For sparse relations with a large extent, this reduces memory usage.
 * However, there is a bit of performance overhead.
 */
public class IntRelationListOfSets extends IntRelation {

    /** Representation of the relation. */
    protected List<Set<Integer>> relation;

    /**
     * Representation invariants
     *
     * NotNull: relation != null
     * SetsNotNull: (\forall i; relation.has(i); relation.get(i) != null),
     *               where List.has(i) == 0 <= i < List.size()
     * SetsInExtent: (\forall i; relation.has(i);
     *               relation.get(i) subset of [0 .. relation.size()))
     *
     * Abstraction function: set of (a, b) such that
     *               relation.get(a).contains(b) holds
     */
}

```

We already have general test cases, and can reuse them again by overriding `setInstance`:

```

public class IntRelationListOfSetsTest extends IntRelationTestCases {

    @Override
    protected void setInstance(final int n) {
        instance = new IntRelationListOfSets(n);
    }
}

```

These tests all fail, for lack of implementations. All formal specifications are also already available. So, we can start implementing methods one-by-one, and re-run test cases after each method is done. For instance, here is the constructor:

```

public IntRelationListOfSets(final int n) {
    super(n);
    relation = new ArrayList<Set<Integer>>();
    for (int i = 0; i != n; ++ i) {
        relation.add(new HashSet<Integer>());
    }
}

```

Likewise, the remaining methods are implemented. See the source code for details. The Abstract Data Type is now ready for use by others. You see that it is easy to change the representation, once you have the safety net of a good set of test cases.

Exercises Apply TDD to address the following requirements.

3. Make `IntRelation` robust by checking preconditions of operations, and throwing `IllegalArgumentException` if they fail.

For each operation, first specify robustness in the abstract class, then develop a test case, and only then implement robustness in both concrete classes.

4. Provide a concrete class `IntRelationMapOfSets` extending `IntRelation`, based on a `Map of Sets`, to reduce memory usage further if an `a` is not related to any `b`. This can be formalized by the following representation invariant:

$$\text{relation.contains}(a) \iff (\exists b; ; a \text{ AF}(\text{relation}) b)$$

where `AF` is the abstraction function. Methods `areRelated`, `add`, and `remove` need to take this into account.

5. Allow clients to use a for-each statement to iterate over all related pairs. That is, make `IntRelation` iterable, by (i) defining a (static nested) class `IntPair` for a record consisting of two integers `a` and `b`, and (ii) making `IntRelation` implement the interface `Iterable<IntPair>` to yield all related pairs one by one.

First specify the operation in the abstract class, then develop a test case, and only then implement it in both concrete classes.

6. Introduce an operation `relatedToFirst(int a)` returning an (object that implements the interface) `Iterable<Integer>` that iterates over all `b` such that `areRelated(a, b)` holds.

Introduce auxiliary method `isValid(a)` to express the precondition of the new operation. Also use it for `isValidPair` to reduce code duplication.

7. Generalize `IntRelation` to the generic interface `Relation<A, B>`, so that `Relation<Integer, Integer>` can be used as a relation on integers with unbounded extent. Provide one implementation `RelationMapOfSets<A, B>` using a `Map of Sets`. For iterating, introduce a generic record type `Pair<A, B>`.

References

- [1] Martin Fowler. *Test Driven Development*. Bliki, <http://martinfowler.com/bliki/TestDrivenDevelopment.html>
- [2] T. Verhoeff. *Notation*. A separate note for Programming Methods (2IPC0).
- [3] T. Verhoeff. *Specification*. A separate note for Programming Methods (2IPC0).
- [4] *Test-Driven Development*, Wikipedia, http://en.wikipedia.org/wiki/Test-driven_development

Author: Tom Verhoeff, Department of Mathematics & Computer Science, Eindhoven University of Technology