TDD — Let's Take a Deep Dive

Ilia Ilin
Sep 10, 2019 · 11 min read ★

Jump

Choose one:

1. I don't know what this term means at all or I've heard something about it but was lazy enough to read more, to understand it and try it in practice. Maybe at this point even TDD = **T**ake a **D**eep **D**ive sounds right? :)

2. TDD definition is clear — in theory, but when it comes to practice I don't know whether all my code could be written in TDD style and what type of tests to write (unit, integration, system, etc.). So far I am fine with approaches of writing tests first or tests after.

3. Oh, TDD — I use it already, from time to time. But, when it comes to refactoring or considering what test to write next I am stuck most of the time. Feeling a lack of creativity here and since it all is so time-consuming I tend to give up. Plus, I've heard that TDD is dead by the way.

4. Easy peasy. Let's see what you have here. By the way, I am Kent Beck.

Good news, regardless of your choice I have information that you will find useful and new. Yeah, maybe except #4.

Motivation behind TDD

Imagine, you received some coding task. What would you do? Usually, the starting point is a brainstorming process considering the main flow, edge cases, exception handling, some thoughts of ideal design on top of it. Ok, finished with thinking, implementation, oh wait thinking again and back to implementation. After the task is finally implemented you will write tests,won't you? All this is a classic and straightforward approach that requires a lot of time to start, especially when you have several implementation ideas and can't decide which one to choose. Or when you don't have any ideas at all.

What if you could write tests and implementation at the same time, eventually adding new test cases and improving your algorithm. **Tests** are the first 'customers' of the code, they can **drive** you through the process of **development** in order to achieve **better design** and more **confidence** in your decisions along the way by giving you instant **feedback**.

Ok, what is TDD then

TDD is a test-driven development technique which is really simple in theory. It boils down to following these steps:

1. Write new test and make code compilable. Run all the tests you have, this new test is **red**.

2. Fix red test and make it **green** as fast as possible. Just focus on all the tests being green at this point in time.

3. Clean up a bit after previous steps, **refactor**

Start with #1 just when you received a new task. Yes, there is nothing to test yet, whatever, create a test class first and #1->#2->#3 in a cycle until your code is correct. How you will understand that the code is correct? Assuming that all your tests are green and cover all important edge cases +happy flow, any new test won't bring much of added value and will be green right away.

Let's code!

We have a fairly straightforward task — develop service calculating attendance of employees in the office. It should be able to count time spent in the office taking into account lunchtime and edge cases when for instance time of leaving the office is not recorded. Let's agree if value representing either 'enter office' record or 'leave office' record is not specified result is just 0 in total. Plus, we will take into account only single presence 'window' — there shouldn't be more than one enter-leave record pairs. If there are some issues while counting lunchtime we'll just use default one = 30 min.

I don't have the exact solution in my head at the moment, let's see whether TDD can help us.

Before we start

You'll find the repository for this example here. Result of each iteration (red-green-refactor iteration) will be in the form of a separate commit. Let's assume before this task we already have some code around and our main goal would be to write one service in between like it usually happens in daily developers routine. I'm going to use the following stack for this task: Java 12, Junit5 + Mockito and Maven. So, let's go through the classes we already have.

```
1  public class AttendanceDao {
2
3      public List<Record> getRecords(long employeeId, LocalDate date) {
4          // Assume we have a good working code in here :)
5          return new ArrayList<>();
6      }
7  }
```

AttendanceDao.java hosted with ❤ by GitHub view raw

This data access object returns attendance records for the given employee on a given date from some storage. As we are going to mock this method in our test, its exact implementation doesn't matter for our task. So, let's move on to the 'Record' object.

```java
public class Record {
    private final Type type;
    private final LocalTime time;

    public Record(Type type, LocalTime time) {
        this.type = type;
        this.time = time;
    }

    public Type getType() {
        return type;
    }

    public LocalTime getTime() {
        return time;
    }
}
```

Record.java hosted with ❤ by GitHub view raw

Every record will represent some specific time point and attendance type. Example: entering the office at 10:30. And we will have the following types of records:

```java
public enum Type {
    ENTER_OFFICE,
    ENTER_LUNCH,
    LEAVE_LUNCH,
    LEAVE_OFFICE
}
```

Type.java hosted with ❤ by GitHub view raw

That was all from prerequisites. Time to start coding.

Iteration #1

If you recall theory at the beginning of the article, TDD consists of red-green-refactor steps. For the first iteration, we have to create a test class and then start with the 'red' step.

```java
@ExtendWith(MockitoExtension.class)
public class AttendanceServiceTest {

    @Mock
    private AttendanceDao attendanceDao;
    private AttendanceService attendanceService;
```

```
    private AttendanceService attendanceService;

    @BeforeEach
    public void init() {
        attendanceService = new AttendanceService(attendanceDao);
    }

    @Test
    public void pressEnter_and_pressLeave_withoutLunch() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(ENTER_OFFICE, LocalTime.of(10, 30)),
                                new Record(LEAVE_OFFICE, LocalTime.of(18, 30))));
        Duration expectedAttendance = Duration.ofHours(7).plus(Duration.ofMinutes(30));
        assertEquals(expectedAttendance, attendanceService.timeInTheOffice(1L, LocalDat
    }
}
```

AttendanceServiceTest_1.java hosted with ❤ by GitHub view raw

Our first test is the usual case when we have the enter record and leave record. As we don't record lunch it is counted automatically as 30 minutes and subtracted from office presence. We mock instance of AttendanceDao to simulate it and initialize our service with this mock. The only thing is missing right now — our implementation class!

```
public class AttendanceService {
    private final AttendanceDao attendanceDao;

    public AttendanceService(AttendanceDao attendanceDao) {
        this.attendanceDao = attendanceDao;
    }

    public Duration timeInTheOffice(long employeeId, LocalDate date) {
        List<Record> records = attendanceDao.getRecords(employeeId, date);
        Optional<Record> enterRecord = records.stream()
                .filter(record -> record.getType() == Type.ENTER_OFFICE)
                .findFirst();
        Optional<Record> leaveRecord = records.stream()
                .filter(record -> record.getType() == Type.LEAVE_OFFICE)
                .findFirst();
        return Duration.between(enterRecord.get().getTime(), leaveRecord.get().getTime(
            .minus(Duration.ofMinutes(30);
    }
}
```

AttendanceService_1.java hosted with ❤ by GitHub view raw

Implementation is simple, we get records from storage — find enter office record, leave office record and find duration between them substracting default lunchtime duration of half an hour.

Refactoring concern (still Iteration #1)

Now the test is green and we are good to go to the next step which is one of the most challenging sometimes IMO — refactoring. Every time we come to this third step there are the following questions we should answer:

1. do we need refactoring at this step? The code looks so simple …
2. ok, the code is messy, where to start from?
3. should I refactor all the issues at this step? Who knows, maybe the next iteration will organically resolve some of them.

After reading books and articles about TDD I understood one thing: **in this 3rd step (= refactoring) do what your gut feeling tells you**. There are uncounted ways of defining whether code needs refactoring and how — all are **subjective**. However, one of the approaches I liked the most as it formalizes the whole process quite well. Let me tell it to you shortly:

For refactoring step in TDD we can use connascence, which is the metric that describes what effort is required to keep everything working if a change in code occurs. Types of connascence are pretty much coupling flavors, ranging from not a big deal to a very error-prone situation. Our refactoring process would then boil down to detecting all connascences we have right now in our code and reducing them as much as possible.

Let's try it out and take a look once more on our latest code. List of all connascence types you can find here. So, we have connascence of Name (CoN) —it means that renaming of method will require renaming of its usages. Connascence of Type (CoT) is present as well — change of return type will require our attention as well. Both of these connascences are inevitable in usual Java code so we won't consider them while refactoring. As far as I can see we have Connascence of Algorithm (CoA) — we find the record by type in a list two times the same way. Changing it in one place would require changing it in the second. Plus you may have noticed that I don't check Optionals for value presence. We'll get back to it later. CoA resolution first.

```
1     public Duration timeInTheOffice(long employeeId, LocalDate date) {
2         List<Record> records = attendanceDao.getRecords(employeeId, date);
```

```
        Optional<Record> enterRecord = getRecordByType(records, Type.ENTER_OFFICE);
        Optional<Record> leaveRecord = getRecordByType(records, Type.LEAVE_OFFICE);
        return Duration.between(enterRecord.get().getTime(), leaveRecord.get().getTime(
           .minus(Duration.ofMinutes(30);
    }

    private Optional<Record> getRecordByType(List<Record> records, Type type) {
        return records.stream()
                .filter(record -> record.getType() == type)
                .findFirst();
    }
```

AttendanceService_2.java hosted with ❤ by GitHub view raw

Now the problem is solved and the refactoring step can be marked as finished. If you want to learn more about connascence and how it can be used while TDD-ing here is a very good video about it. I want to emphasize once more that **you don't have to use exactly this technique** — it just suits me the most of those I've seen before and I am using it only sometimes when the gut feeling is not very talkative:) I encourage you to choose your favorite approach(es) and not blindly follow what any of the articles/books/videos tell you to use. Enough of wisdom, time for the second iteration has come.

Iteration #2

All the groundwork and most of the boring explanations are well behind, let's add one more test and enjoy TDD more.

```
    @Test
    public void pressEnterOnly() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(ENTER_OFFICE, LocalTime.of(10, 30))));
        assertEquals(Duration.ZERO, attendanceService.timeInTheOffice(1L, LocalDate.now(
    }
```

AttendanceServiceTest_2.java hosted with ❤ by GitHub view raw

Test checks that we get 0 as a result if there is no record of leaving the office.

```
    public Duration timeInTheOffice(long employeeId, LocalDate date) {
        List<Record> records = attendanceDao.getRecords(employeeId, date);
        Optional<Record> enterRecord = getRecordByType(records, Type.ENTER_OFFICE);
        Optional<Record> leaveRecord = getRecordByType(records, Type.LEAVE_OFFICE);
```

```
        if (leaveRecord.isPresent() && leaveRecord.get().getTime() != null) {
            return Duration.between(enterRecord.get().getTime(), leaveRecord.get().getT
                    .minus(Duration.ofMinutes(30));
        }
        return Duration.ZERO;
    }
```

AttendanceService_3.java hosted with ❤ by GitHub view raw

Ok, now all tests are green. Have you noticed that this new test has forced us to solve one of the issues with value presence of Optionals? If we would again analyze our implementation class and test class we'll find connascence of value (CoV) in the test class — input parameters to the method of 'attendanceService' , namely 'employeeId' and 'date' repeat in two tests already and will most probably be used again in future tests.

```
    private final long EMPLOYEE_ID = 1L;
    private final LocalDate DATE_TO_CHECK = LocalDate.now();

    @BeforeEach
    public void init() {
        attendanceService = new AttendanceService(attendanceDao);
    }

    @Test
    public void pressEnter_and_pressLeave_withoutLunch() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(ENTER_OFFICE, LocalTime.of(10, 30)),
                                new Record(LEAVE_OFFICE, LocalTime.of(18, 30))));
        Duration expectedAttendance = Duration.ofHours(7).plus(Duration.ofMinutes(30));
        assertEquals(expectedAttendance, attendanceService.timeInTheOffice(EMPLOYEE_ID,
    }

    @Test
    public void pressEnterOnly() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(ENTER_OFFICE, LocalTime.of(10, 30))));
        assertEquals(Duration.ZERO, attendanceService.timeInTheOffice(EMPLOYEE_ID, DATE
    }
}
```

AttendanceServiceTest_3.java hosted with ❤ by GitHub view raw

The solution was to extract these values into the final fields of the test class.

Iteration #3

This iteration will be pretty much the same as the one before. We will add a test which will check that result is 0 if there is no record of entering the office.

```
1       @Test
2       public void pressLeaveOnly() {
3           when(attendanceDao.getRecords(anyLong(), any()))
4                   .thenReturn(
5                           List.of(new Record(LEAVE_OFFICE, LocalTime.of(18, 0))));
6           assertEquals(Duration.ZERO, attendanceService.timeInTheOffice(EMPLOYEE_ID, DATE_
7           ));
8       }
```

AttendanceServiceTest_4.java hosted with ❤ by GitHub view raw

Time for a 'green' step:

```
1       public Duration timeInTheOffice(long employeeId, LocalDate date) {
2           List<Record> records = attendanceDao.getRecords(employeeId, date);
3           Optional<Record> enterRecord = getRecordByType(records, Type.ENTER_OFFICE);
4           Optional<Record> leaveRecord = getRecordByType(records, Type.LEAVE_OFFICE);
5           if (enterRecord.isEmpty() || leaveRecord.isEmpty()) {
6               return Duration.ZERO;
7           }
8           LocalTime enterTime = enterRecord.get().getTime();
9           LocalTime leaveTime = leaveRecord.get().getTime();
10          if (enterTime == null || leaveTime == null) {
11              return Duration.ZERO;
12          }
13          return Duration.between(enterTime, leaveTime)
14                  .minus(Duration.ofMinutes(30));
15      }
```

AttendanceService_4.java hosted with ❤ by GitHub view raw

Yeah, if you've noticed I've also done small refactoring in this method, there were no connascences, just the code vulnerable to NPEs and nested constructions. In this iteration, I wanted to show you again that refactoring step is dependent a lot on a style of developer writing code, he decides when and how. Connascences search or any other technique can serve as an auxiliary tool to help when further improvement is not that evident.

Iteration #4

Now, let's add a test for the case when we log our lunchtime.

```
1       @Test
2       public void pressEnter_and_pressLeave_withLunch() {
3           when(attendanceDao.getRecords(anyLong(), any()))
4                   .thenReturn(
5                           List.of(new Record(ENTER_OFFICE, LocalTime.of(10, 30)),
6                                   new Record(ENTER_LUNCH, LocalTime.of(13, 30)),
7                                   new Record(LEAVE_LUNCH, LocalTime.of(14, 30)),
8                                   new Record(LEAVE_OFFICE, LocalTime.of(18, 30))));
9           assertEquals(Duration.ofHours(7), attendanceService.timeInTheOffice(EMPLOYEE_ID
10      }
```

AttendanceServiceTest_5.java hosted with ❤ by GitHub view raw

Here we have an ideal and at the same time usual scenario: everything is logged in a correct order and it all makes sense :) Test is red — lets' fix it. To save the space and since the refactoring on this step will be very natural again I'll just show the end result of this iteration:

```
1       private final Duration DEFAULT_LUNCH_DURATION = Duration.ofMinutes(30);
2
3       public AttendanceService(AttendanceDao attendanceDao) {
4           this.attendanceDao = attendanceDao;
5       }
6
7       public Duration timeInTheOffice(long employeeId, LocalDate date) {
8           List<Record> records = attendanceDao.getRecords(employeeId, date);
9           Optional<Record> enterRecord = getRecordByType(records, Type.ENTER_OFFICE);
10          Optional<Record> leaveRecord = getRecordByType(records, Type.LEAVE_OFFICE);
11          if (enterRecord.isEmpty() || leaveRecord.isEmpty()) {
12              return Duration.ZERO;
13          }
14          LocalTime enterTime = enterRecord.get().getTime();
15          LocalTime leaveTime = leaveRecord.get().getTime();
16          if (enterTime == null || leaveTime == null) {
17              return Duration.ZERO;
18          }
19          return Duration.between(enterTime, leaveTime)
20                  .minus(timeForLunch(records));
21      }
22
23      private Duration timeForLunch(List<Record> records) {
24          Optional<Record> lunchEnterRecord = getRecordByType(records, ENTER_LUNCH);
```

```
        Optional<Record> lunchLeaveRecord = getRecordByType(records, LEAVE_LUNCH);
        if (lunchEnterRecord.isEmpty() || lunchLeaveRecord.isEmpty()) {
            return DEFAULT_LUNCH_DURATION;
        }
        LocalTime lunchEnterTime = lunchEnterRecord.get().getTime();
        LocalTime lunchLeaveTime = lunchLeaveRecord.get().getTime();
        if (lunchEnterTime == null || lunchLeaveTime == null) {
            return DEFAULT_LUNCH_DURATION;
        }
        return Duration.between(lunchEnterTime, lunchLeaveTime);
    }
```

AttendanceService_5.java hosted with ❤ by GitHub view raw

We have created a new method for calculating lunch, in case of some value being empty or null we just use the default value of 30 minutes.

Iteration #5 (we are close to the end)

As our service is almost ready and covers 'happy scenario' it is time to go through edge cases. The case, when leave record is earlier than enter record, is one of them.

```
    @Test
    public void pressLeaveEarlierThanEnter() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(LEAVE_OFFICE, LocalTime.of(10, 30)),
                                new Record(ENTER_OFFICE, LocalTime.of(11, 30))));
        assertEquals(Duration.ZERO, attendanceService.timeInTheOffice(EMPLOYEE_ID, DATE_
    }
```

AttendanceServiceTest_6.java hosted with ❤ by GitHub view raw

Fix for this would fit in one line and won't require any refactoring step IMO.

```
        LocalTime enterTime = enterRecord.get().getTime();
        LocalTime leaveTime = leaveRecord.get().getTime();
        if (enterTime == null || leaveTime == null || leaveTime.isBefore(enterTime)) {
            return Duration.ZERO;
        }
```

AttendanceService_6.java hosted with ❤ by GitHub view raw

Iteration #6

This iteration will test the same edge case as in the previous one but for lunchtime — when 'leave lunch' record will happen earlier than 'enter lunch'.

```
1       @Test
2       public void pressLeaveLunchEarlierThanEnterLunch() {
3           when(attendanceDao.getRecords(anyLong(), any()))
4                   .thenReturn(
5                           List.of(new Record(ENTER_OFFICE, LocalTime.of(8, 0)),
6                                   new Record(LEAVE_LUNCH, LocalTime.of(11, 30)),
7                                   new Record(ENTER_LUNCH, LocalTime.of(12, 30)),
8                                   new Record(LEAVE_OFFICE, LocalTime.of(18, 30))));
9           assertEquals(Duration.ofHours(10), attendanceService.timeInTheOffice(EMPLOYEE_I
10      }
```

AttendanceServiceTest_7.java hosted with ❤ by GitHub view raw

The solution will be similar again and very straightforward — thanks to Java Time API appeared in Java 8.

```
1           LocalTime lunchEnterTime = lunchEnterRecord.get().getTime();
2           LocalTime lunchLeaveTime = lunchLeaveRecord.get().getTime();
3           if (lunchEnterTime == null || lunchLeaveTime == null || lunchLeaveTime.isBefore(
4               return DEFAULT_LUNCH_DURATION;
5           }
```

AttendanceService_7.java hosted with ❤ by GitHub view raw

If you recall in case of any problems while calculating time spent on lunchtime we just take the default of 30 minutes.

Iteration #7

Before we finish there is one more important type of edge case to cover. What will happen if we'll forget whether we've pressed 'enter office' or 'leave office' and duplicate the record with a different time? Let's start with duplicated leave office record.

```
1       @Test
2       public void pressEnter_and_pressLeaveTwoTimes() {
3           when(attendanceDao.getRecords(anyLong(), any()))
4                   .thenReturn(
5                           List.of(new Record(ENTER_OFFICE, LocalTime.of(8, 0)),
6                                   new Record(LEAVE_OFFICE, LocalTime.of(16, 0)),
7                                   new Record(LEAVE_OFFICE, LocalTime.of(17, 30))));
8           assertEquals(Duration.ofHours(9), attendanceService.timeInTheOffice(EMPLOYEE_ID,
```

```
9       }
```

AttendanceServiceTest_8.java hosted with ❤ by GitHub view raw

Why 9 hours? Imagine yourself leaving the office, recording your leave and suddenly in elevator, you remember about smth urgent and unfinished. So, you return to the office. With this in mind, in the case of leaving the office, we'll take the latest record. Minus default lunchtime. How do we implement this?

```
1       private Optional<Record> getRecordByType(List<Record> records, Type type) {
2           Stream<Record> recordStream = records.stream()
3                   .filter(record -> record.getType() == type);
4           return recordStream.max(Comparator.comparing(Record::getTime));
5       }
```

AttendanceService_8.java hosted with ❤ by GitHub view raw

Before we were taking every time the first record of the given type. The solution is to sort records of the same type by time and take the latest one. Code is so simple in all last iterations, so I am not even mentioning refactoring step.

Iteration #8

And one more test — when we have two enter records.

```
1       @Test
2       public void pressEnterTwoTimes_and_pressLeaveOneTime() {
3           when(attendanceDao.getRecords(anyLong(), any()))
4                   .thenReturn(
5                           List.of(new Record(ENTER_OFFICE, LocalTime.of(8, 0)),
6                                   new Record(ENTER_OFFICE, LocalTime.of(9, 0)),
7                                   new Record(LEAVE_OFFICE, LocalTime.of(16, 0))));
8           Duration expectedDuration = Duration.ofHours(7).plus(Duration.ofMinutes(30));
9           assertEquals(expectedDuration, attendanceService.timeInTheOffice(EMPLOYEE_ID, D
10      }
```

AttendanceServiceTest_9.java hosted with ❤ by GitHub view raw

Here the situation is a bit clearer. We could just forget that we've recorded our enter time and did it once more. So, we are taking the earliest record.

```
1       private Optional<Record> getRecordByType(List<Record> records, Type type) {
2           Stream<Record> recordStream = records.stream()
3                   .filter(record -> record.getType() == type);
```

```
        if (type == ENTER_OFFICE || type == ENTER_LUNCH) {
            return recordStream.min(Comparator.comparing(Record::getTime));
        }
        return recordStream.max(Comparator.comparing(Record::getTime));
    }
```

AttendanceService_9.java hosted with ❤ by GitHub view raw

Now, we are protected against duplicate records and take the earliest from all enter records and the last from leave records.

Iteration #9 (bonus one and the last one)

This iteration was not planned at all but while discussing this example with my colleagues one of them found a bug. Found a bug in a program that was driven by tests and should ideally be bulletproof at the end. In fact, it is never the case — TDD doesn't guarantee your algorithm will be bug-free :) Bug is in the subtraction of lunchtime, when our presence in the office is less than lunchtime duration (default or specified one).

```
    @Test
    public void durationBetweenRecordsIsLessThanLunchTime() {
        when(attendanceDao.getRecords(anyLong(), any()))
                .thenReturn(
                        List.of(new Record(ENTER_OFFICE, LocalTime.of(8, 0)),
                                new Record(LEAVE_OFFICE, LocalTime.of(8, 25))));
        assertEquals(Duration.ofMinutes(25), attendanceService.timeInTheOffice(EMPLOYEE_
    }
```

AttendanceServiceTest_10.java hosted with ❤ by GitHub view raw

The fix would be easy enough — prevent subtraction in case duration of lunch is longer than presence in the office.

```
        Duration inOffice = Duration.between(enterTime, leaveTime);
        Duration lunchDuration = timeForLunch(records);
        return inOffice.compareTo(lunchDuration) > 0 ? inOffice.minus(lunchDuration) : i
```

AttendanceService_10.java hosted with ❤ by GitHub view raw

Let's finish here with our code example. I think you got the idea. Code is not perfect at this stage, more tests could be written and most probably some bugs are still there. However, we have some level of confidence at this point as we have 9 passing tests.

'TDD is dead' thoughts + hundreds of subjective things

Every concept is either forgotten or misused. After the appearance of TDD, many people were very excited and wanted to use the technique for any new code piece they write. In some teams/companies usage of TDD was even enforced, which eventually led to misuse of the term. It became somehow connected to bad practices in testing like 'heavy mocking', brought some devs to the wrong design decisions because of the requirement to create testable components every time. A very good example of experiences like this you can find in this article and this one, both of them written by the creator of Ruby on Rails.

After all that, there was a very interesting set of discussions on TDD topic between the owner of the idea — Kent Beck, Martin Fowler, and David Heinemeier Hanson — creator of Ruby on Rails. So, at this point, I'd like to sum up all said by me in the article already, answer the questions for those who have chosen #2 at the beginning and at the same time point out important parts from that discussion between famous developers.

Summary :

1. TDD is a **simple red-green-refactor cycle** of development which helps you in building better design by breaking the task down to smaller units, gives you fast **feedback** on your crazy ideas during the development, makes you more **confident** in your code and as a result improves your sleep quality :D

2. This way of development is **not a silver bullet**, it shouldn't be used everywhere. I see it useful only for developing of some services with business logic. These services usually have inputs and outputs as some intermediate results in the system which simplifies the process of writing tests.

3. TDD **doesn't guarantee** you **bug-free** code (remember iteration #9?), but your codebase can benefit from its usage anyway.

4. Unit tests? Integration tests? Mock or not to mock? I would prefer unit tests as you'll have to run them many times while developing that required feature, the **less time** you wait for the **feedback** — the better it is. Regarding 'mocking', depends a lot on your habits. But keep in mind that if you have to mock literally everything, it is a sign of a poorly designed code.

5. **Refactoring** approach — as said before choose any that **works best for you**. I, personally, use common sense and sometimes analysis of present connascences in the code. Btw, you may have seen in the code example that I did green-refactor

together as one step many times. You choose the size of these steps, some prefer to move on in tiny steps while some in a faster manner, and it depends on the complexity of the task as well.

6. TDD may be '**dead**' for you as you are not used to this kind of thinking and it is absolutely fine. Go and use some other technique.

Hope it helped you on your way to excellence. Take care and happy coding!

Tdd Software Development Test Driven Development Testing Java