Practice TDD With Kotlin

Why test-driven design/development (TDD) in your development?

- **Reduce possibles production regressions**

  What saves us time when troubleshooting

- **Validate functional design**

  Make sure we respect the target solution

- **Increase the overall quality of your program**

  The program is more reliable and we are more confident to push a new version on production

TDD as Test-Driven Development

Test-driven development refers to a programming style favoring testing before writing production code. Basically, you have to follow the steps including:

- Write Test
- Run Test (the test should fail)
- Write Production Code
- Run Test (the test should pass)
- Refactor until the code is conform
- Repeat, “accumulating” unit tests over time

Kotlin for TDD

Kotlin has many advantages compared to Java for writing TDD:

- Kotlin reduces java boilerplate thanks to function extension which can improve your test readability.
- Kotlin supports the infix notation allowing to write a code in more natural English.

```
   val canOrder = user.isAuthenticated and user.hasACreditCard
```

- Kotlin supports backticks function name, really convenient to write tests

```
class MyTestCase {
     @Test fun `ensure everything works`() { /*...*/ }

     @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

- Kotlin is interoperable with Java meaning that you can use Kotlin to write your tests whereas your production code is written in Java.

using Kotlin we can build a library to set up a TDD environment:

```
@Test
fun `I should be able to insert a new item in my todo list`() {
    given {
        `a todo list`
    } and {
        `an item`("Eat banana")
    } `when` {
        `I add the last item into my todo list`
    } then {
        `I expect this item is present in my todo list`
    }
}
```

- Kotlin-TDD provides two flavors
    - `GivenWhen`Then exposing the `given`, and, `when` and then infix functions.
    - `AssumeActAssert` exposing the `assume`, and, `act` and `assert` infix functions.

Indeed the same test previously can be written by following the AAA pattern.

```
@Test
fun `I should be able to insert a new item in my todo list`() {
    assume {
        `a todo list`
    } and {
        `an item`("Eat banana")
    } act {
        `I add the last item into my todo list`
    } assert {
        `I expect this item is present in my todo list`
    }
}
```

Example

Let’s assume you have a requirement to create a Todo List application and one of the acceptance criteria is. As a user, I should see my new item when it has been added to my To-do list.

1 – Setup Kotlin-TDD

Let’s start by importing this dependency into your project. I’m assuming Junit 5 is installed in your project.

With Gradle:

```
testCompile "io.github.ludorival:kotlin-tdd:1.1.2"
```

With Maven:

```
<dependency>
<groupId>io.github.ludorival</groupId>
<artifactId>kotlin-tdd</artifactId>
<version>1.1.2</version>
<scope>test</scope>
</dependency>
```

2 – Write Your Interface Action

Inside of each step, you can access a field named action which is the instance you will pass to your TDD configuration (you will see in the next step). It has no real use in the library but it allows you to use a common instance throughout your tests. Here we will use it to expose different possible actions in the application:

```
// src/test/kotlin/com/example/kotlintdd/Action.kt
package com.example.kotlintdd

interface Action {

  fun createTodoList(): TodoList

  fun createItem(name: String): Item

  fun addItem(todoList: TodoList, item: Item): TodoList
}
```

3 – Configure an Instance of Givenwhenthen To Use in All Our Unit Tests

```
// src/test/kotlin/com/example/kotlintdd/UnitTest.kt
package com.example.kotlintdd

import io.github.ludorival.kotlintdd.GWTContext // it is an alias of GivenWhenThen.Context
import io.github.ludorival.kotlintdd.GivenWhenThen

object UnitTest : GivenWhenThen<Action> {
    override val action: Action = object: Action {
        override fun createTodoList() = TodoList()
        override fun createItem(name: String) = Item(name)
       override fun addItem(todoList: TodoList, item: Item) = todoList.add(item)
    }
}
// defines the entrypoint on file-level to be automatically recognized by your IDE
fun <R> given(block: GWTContext<Action, Unit>.() -> R) = UnitTest.given(block)
fun <R> `when`(block: GWTContext<Action, Unit>.() -> R) = UnitTest.`when`(block)
```

4 – Write Your Custom DSL

> A Domain Specific Language is a computer language specialized to a particular application domain.

This step is optional but it helps to describe your action in a natural language. Here we will create a file that will host this DSL.

```
// src/test/kotlin/com/example/kotlintdd/DSL.kt
package com.example.kotlintdd

import io.github.ludorival.kotlintdd.GWTContext

val GWTContext<*, *>.`a todo list` get() = action.createTodoList()

fun GWTContext<*, *>.`an item`(name: String) = action.createItem(name)

val GWTContext<*, *>.`I add the last item into my todo list` get() =
    action.addItem(first<TodoList>(), last<Item>())

val GWTContext<*, *>.`I expect this item is present in my todo list` get() =
    assertTrue(first<TodoList>().contains(last<Item>()))
```

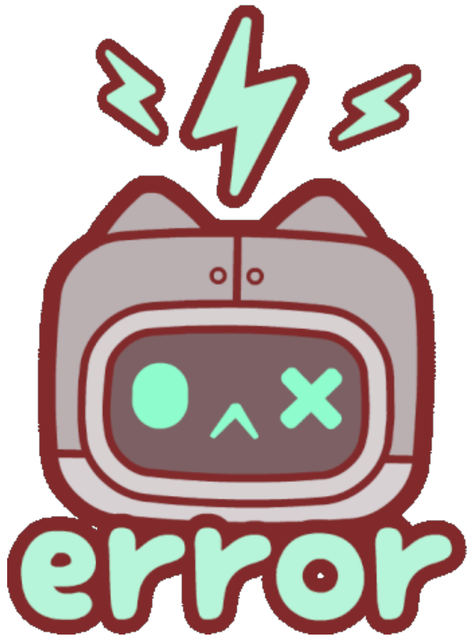
5 – Write Your Unit Test

> Now we have configured our TDD and our custom DSL, let's put it all together in a test:

```
// src/test/kotlin/com/example/kotlintdd/TodoListTest
package com.example.kotlintdd

import org.junit.jupiter.api.Test

class TodoListTest {

  @Test
  fun `I should be able to insert a new item in my todo list`() {
      given {
          `a todo list`
      } and {
          `an item`("Eat banana")
      } `when` {
          `I add the last item into my todo list`
      } then {
          `I expect this item is present in my todo list`
      }
  }
}
```

6 – Run the Test

> Of course, the test is failing due to a compilation error. We did not write any production code. Don’t worry this is part of the TDD process.



7 – Write Production Code

> Now it is time to make the test green.

```
   // src/main/kotlin/com/example/kotlintdd/Item
   package com.example.kotlintdd

   data class Item(val name: String)
```

- Create the TodoList class:

```
// src/main/kotlin/com/example/kotlintdd/TodoList
package com.example.kotlintdd

class TodoList {
 val list = mutableListOf()

   fun add(item: Item) {
      list.add(item)
      return this
    }

   fun contains(item: Item) = list.contains(item)
}
```

8 – Run the Test Again

> your test is Green!

9 – Next Steps: Acceptance Test

> We can continue to add more tests by combining the Given When Then pattern and our custom DSL. The DSL can be enhanced for more use cases. The advantage of Kotlin-TDD is that you can reuse the same process for writing Acceptance Test as well.

Let's assume that you have a Spring application where the TodoList and Item are saved in a database. The creation and the update should be done through the database for those entities.

We expect to have three endpoints in our Rest API:

```
POST /v1/todo/list // Create a new Todo list -> return the TodoList with an id
POST /v1/todo/item // Create a new Item -> return the Item with an id
PUT /v1/todo/list/{listId}/add // add the item defined by {itemId} in the list {listId}
```

We can write a different implementation of our Action interface:

```
// src/test/kotlin/com/example/kotlintdd/acceptance/RestActions.kt
package com.example.kotlintdd.acceptance

import com.example.kotlintdd.Action
import com.example.kotlintdd.Item
import com.example.kotlintdd.TodoList
import org.springframework.http.HttpEntity
import org.springframework.http.HttpMethod
import org.springframework.http.ResponseEntity
import org.springframework.web.client.RestTemplate

class RestAction : Action {

    private val url = "http://localhost:8080/spring-rest/v1"
    private val restTemplate = RestTemplate()
    override fun createTodoList(): TodoList {
        val response: ResponseEntity<TodoList> = restTemplate
            .exchange("$url/todo",
                      HttpMethod.POST,
                      HttpEntity(TodoList()),
                      TodoList::class.java)
        return response.body!!
    }

    override fun createItem(name: String): Item {
        val response: ResponseEntity<Item> = restTemplate
            .exchange("$url/item",
                      HttpMethod.POST,
                      HttpEntity(Item(name)),
                      Item::class.java)
        return response.body!!
    }

    override fun addItem(todoList: TodoList, item: Item): TodoList {
        val response: ResponseEntity<TodoList> = restTemplate
            .exchange(
                "$url/todo/${todoList.id}/add",
                HttpMethod.PUT,
                HttpEntity(item),
                TodoList::class.java
            )
        return response.body!!
    }

}
```

And you need to setup this new action for a different instance of GivenWhenThen:

```
// src/test/kotlin/com/example/kotlintdd/acceptance/AcceptanceTest.kt
package com.example.kotlintdd.acceptance

import com.example.kotlintdd.Action
import io.github.ludorival.kotlintdd.GWTContext
import io.github.ludorival.kotlintdd.GivenWhenThen

object AcceptanceTest: GivenWhenThen<Action> {
    override val action: Action = RestAction()

}
fun <R> given(block: GWTContext<Action, Unit>.() -> R) = AcceptanceTest.given(block)
```

Then I can literally copy my unit test as an acceptance test:

```
// src/test/kotlin/com/example/kotlintdd/acceptance/TodoListAT.kt
package com.example.kotlintdd.acceptance

import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest
class TodoListAT {

  @Test
  fun `I should be able to insert a new item in my todo list`() {
      given {
          `a todo list`
      } and {
          `an item`("Eat banana")
      } `when` {
          `I add the last item into my todo list`
      } then {
          `I expect this item is present in my todo list`
      }
  }
}
```