Writing a contract test

This tutorial will take you through the steps required to write a contract test using Kotlin and Java.

The testing DSL allows one to define a piece of the ledger with transactions referring to each other, and ways of verifying their correctness.

Testing single transactions

We start with the empty ledger:

Kotlin Java

```
class CommercialPaperTest{
    @Test
    fun emptyLedger() {
        ledger {
        }
    }
    ...
}
```

The DSL keyword `ledger` takes a closure that can build up several transactions and may verify their overall correctness. A ledger is effectively a fresh world with no pre-existing transactions or services within it.

We will start with defining helper function that returns a `CommercialPaper` state:

Kotlin Java

```
fun getPaper(): ICommercialPaperState = CommercialPaper.State(
    issuance = MEGA_CORP.ref(123),
    owner = MEGA_CORP,
    faceValue = 1000.DOLLARS `issued by` MEGA_CORP.ref(123),
    maturityDate = TEST_TX_TIME + 7.days
)
```

It's a `CommercialPaper` issued by `MEGA_CORP` with face value of $1000 and maturity date in 7 days.

Let's add a `CommercialPaper` transaction:

Kotlin Java

```
@Test
fun simpleCPDoesntCompile() {
    val inState = getPaper()
    ledger {
        transaction {
            input(inState)
        }
    }
}
```

We can add a transaction to the ledger using the `transaction` primitive. The transaction in turn may be defined by specifying `input`-s, `output`-s, `command`-s and `attachment`-s.

The above `input` call is a bit special; transactions don't actually contain input states, just references to output states of other transactions. Under the hood the above `input` call creates a dummy transaction in the ledger (that won't be verified) which outputs the specified state, and references that from this transaction.

The above code however doesn't compile:

Kotlin Java

```
Error:(29, 17) Kotlin: Type mismatch: inferred type is Unit but EnforceVerifyOrFail was
expected
```

This is deliberate: The DSL forces us to specify either `this.verifies()` or `this `fails with` "some text"` on the last line of `transaction`:

Kotlin Java

```
@Test
fun simpleCP() {
    val inState = getPaper()
    ledger {
        transaction {
            input(inState)
            this.verifies()
        }
    }
}
```

Let's take a look at a transaction that fails.

Kotlin Java

```
@Test
fun simpleCPMove() {
    val inState = getPaper()
    ledger {
        transaction {
            input(inState)
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this.verifies()
        }
    }
}
```

When run, that code produces the following error:

Kotlin Java

```
net.corda.core.contracts.TransactionVerificationException$ContractRejection:
java.lang.IllegalArgumentException: Failed requirement: the state is propagated
```

The transaction verification failed, because we wanted to move paper but didn't specify an output - but the state should be propagated. However we can specify that this is an intended behaviour by changing `this.verifies()` to `this `fails with` "the state is propagated"`:

Kotlin Java

```
@Test
fun simpleCPMoveFails() {
    val inState = getPaper()
    ledger {
        transaction {
            input(inState)
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this `fails with` "the state is propagated"
        }
    }
}
```

We can continue to build the transaction until it `verifies`:

Kotlin Java

```
@Test
fun simpleCPMoveSuccess() {
    val inState = getPaper()
    ledger {
        transaction {
            input(inState)
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this `fails with` "the state is propagated"
            output("alice's paper") { inState `owned by` ALICE_PUBKEY }
            this.verifies()
        }
    }
}
```

`output` specifies that we want the input state to be transferred to `ALICE` and `command` adds the `Move` command itself, signed by the current owner of the input state, `MEGA_CORP_PUBKEY`.

We constructed a complete signed commercial paper transaction and verified it. Note how we left in the `fails with` line - this is fine, the failure will be tested on the partially constructed transaction.

What should we do if we wanted to test what happens when the wrong party signs the transaction? If we simply add a `command` it will permanently ruin the transaction... Enter `tweak`:

Kotlin Java

```
@Test
fun `simple issuance with tweak`() {
    ledger {
        transaction {
            output("paper") { getPaper() } // Some CP is issued onto the ledger by
MegaCorp.
            tweak {
                command(DUMMY_PUBKEY_1) { CommercialPaper.Commands.Issue() }
                timestamp(TEST_TX_TIME)
                this `fails with` "output states are issued by a command signer"
            }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Issue() }
            timestamp(TEST_TX_TIME)
            this.verifies()
        }
    }
}
```

`tweak` creates a local copy of the transaction. This makes possible to locally "ruin" the transaction while not modifying the original one, allowing testing of different error conditions.

We now have a neat little test that tests a single transaction. This is already useful, and in fact testing of a single transaction in this way is very common. There is even a shorthand top-level `transaction` primitive that creates a ledger with a single transaction:

Kotlin Java

```
@Test
fun `simple issuance with tweak and top level transaction`() {
    transaction {
        output("paper") { getPaper() } // Some CP is issued onto the ledger by MegaCorp.
        tweak {
            command(DUMMY_PUBKEY_1) { CommercialPaper.Commands.Issue() }
            timestamp(TEST_TX_TIME)
            this `fails with` "output states are issued by a command signer"
        }
        command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Issue() }
        timestamp(TEST_TX_TIME)
        this.verifies()
    }
}
```

Chaining transactions

Now that we know how to define a single transaction, let's look at how to define a chain of them:

Kotlin Java

```
@Test
fun `chain commercial paper`() {
    val issuer = MEGA_CORP.ref(123)

    ledger {
        unverifiedTransaction {
            output("alice's $900", 900.DOLLARS.CASH `issued by` issuer `owned by`
ALICE_PUBKEY)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output("paper") { getPaper() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Issue() }
            timestamp(TEST_TX_TIME)
            this.verifies()
        }


        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output("borrowed $900") { 900.DOLLARS.CASH `issued by` issuer `owned by`
MEGA_CORP_PUBKEY }
            output("alice's paper") { "paper".output<ICommercialPaperState>() `owned by`
ALICE_PUBKEY }
            command(ALICE_PUBKEY) { Cash.Commands.Move() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this.verifies()
        }
    }
}
```

In this example we declare that `ALICE` has $900 but we don't care where from. For this we can use `unverifiedTransaction`. Note how we don't need to specify `this.verifies()`.

Notice that we labelled output with `"alice's $900"`, also in transaction named `"Issuance"` we labelled a commercial paper with `"paper"`. Now we can subsequently refer to them in other transactions, e.g. by `input("alice's $900")` or `"paper".output<ICommercialPaperState>()`.

The last transaction named `"Trade"` exemplifies simple fact of selling the `CommercialPaper` to Alice for her $900, $100 less than the face value at 10% interest after only 7 days.

We can also test whole ledger calling `this.verifies()` and `this.fails()` on the ledger level. To do so let's create a simple example that uses the same input twice:

Kotlin Java

```
@Test
fun `chain commercial paper double spend`() {
    val issuer = MEGA_CORP.ref(123)
    ledger {
        unverifiedTransaction {
            output("alice's $900", 900.DOLLARS.CASH `issued by` issuer `owned by`
ALICE_PUBKEY)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output("paper") { getPaper() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Issue() }
            timestamp(TEST_TX_TIME)
            this.verifies()
        }

        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output("borrowed $900") { 900.DOLLARS.CASH `issued by` issuer `owned by`
MEGA_CORP_PUBKEY }
            output("alice's paper") { "paper".output<ICommercialPaperState>() `owned by`
ALICE_PUBKEY }
            command(ALICE_PUBKEY) { Cash.Commands.Move() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this.verifies()
        }

        transaction {
            input("paper")
            // We moved a paper to another pubkey.
            output("bob's paper") { "paper".output<ICommercialPaperState>() `owned by`
BOB_PUBKEY }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this.verifies()
        }

        this.fails()
    }
}
```

The transactions `verifies()` individually, however the state was spent twice! That's why we need the global ledger verification (`this.fails()` at the end). As in previous examples we can use `tweak` to create a local copy of the whole ledger:

Kotlin Java

```
@Test
fun `chain commercial tweak`() {
    val issuer = MEGA_CORP.ref(123)
    ledger {
        unverifiedTransaction {
            output("alice's $900", 900.DOLLARS.CASH `issued by` issuer `owned by`
ALICE_PUBKEY)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output("paper") { getPaper() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Issue() }
            timestamp(TEST_TX_TIME)
            this.verifies()
        }

        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output("borrowed $900") { 900.DOLLARS.CASH `issued by` issuer `owned by`
MEGA_CORP_PUBKEY }
            output("alice's paper") { "paper".output<ICommercialPaperState>() `owned by`
ALICE_PUBKEY }
            command(ALICE_PUBKEY) { Cash.Commands.Move() }
            command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
            this.verifies()
        }

        tweak {
            transaction {
                input("paper")
                // We moved a paper to another pubkey.
                output("bob's paper") { "paper".output<ICommercialPaperState>() `owned
by` BOB_PUBKEY }
                command(MEGA_CORP_PUBKEY) { CommercialPaper.Commands.Move() }
                this.verifies()
            }
            this.fails()
        }

        this.verifies()
    }
}
```