UI tests

In this section, we will go over the UI testing of IntelliJ plugin and implement UI tests for adding two connections with the same name using the by IntelliJ provided library called ui test robot. The library runs a Robot-Server, under which is the IDEA with the plugin (the plugin from IBA Group a.s. in out case). The UI test uses remote-robot to give instructions to the IntelliJ IDEA, where the robot-server and For Mainframe plugins are running. The communication is done via HTTP protocol.

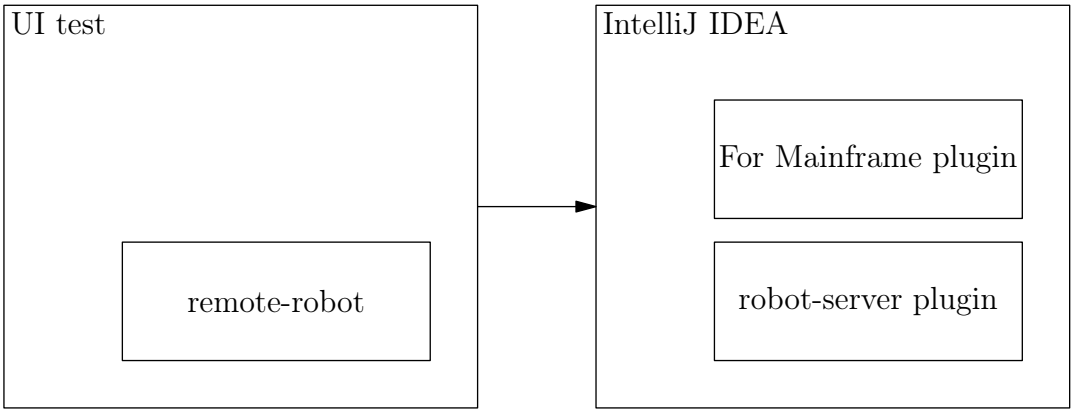
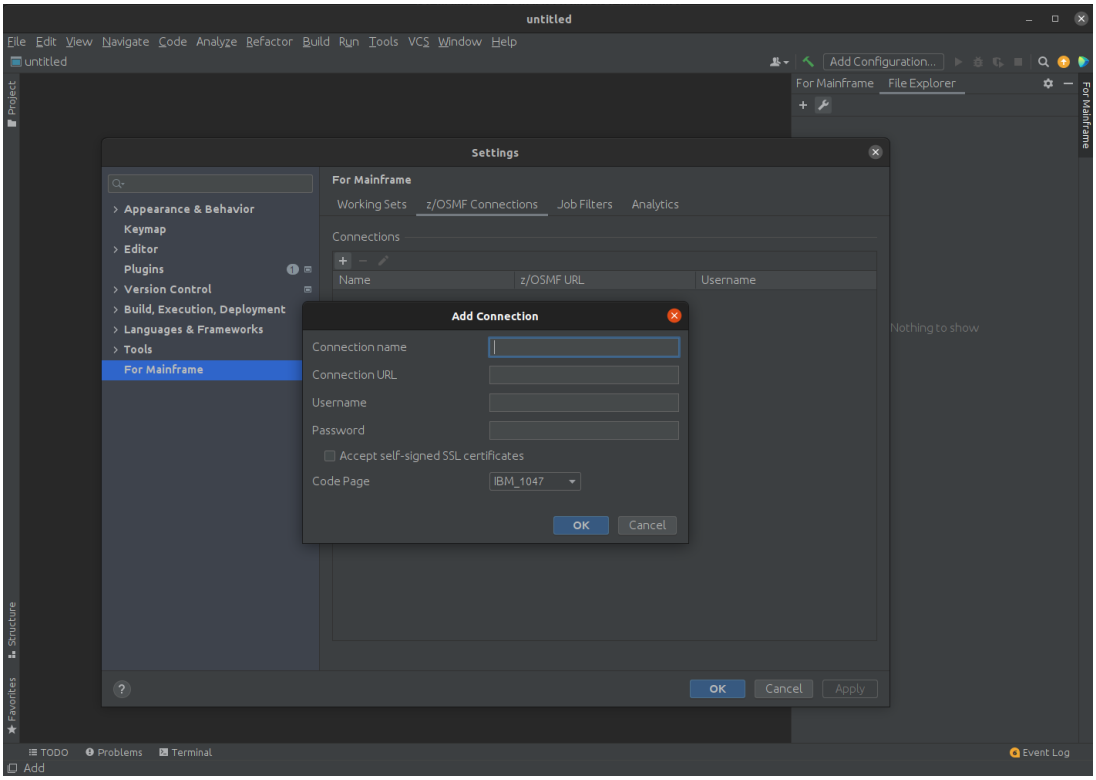
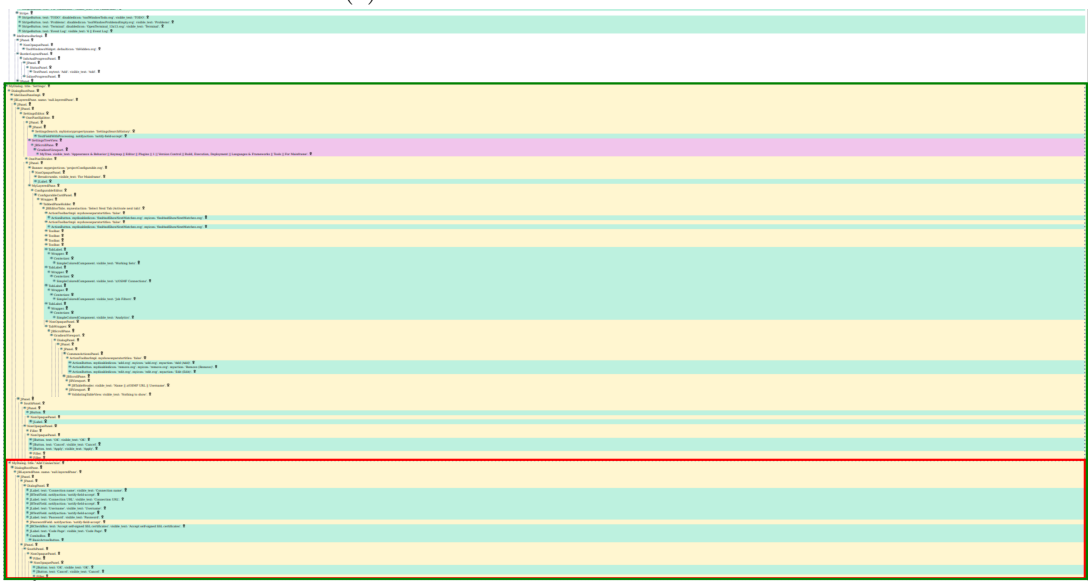


Fig. 1: The Remote-Robot library by IntelliJ. In the picture, we can see that the UI test uses remote-robot to give instructions to the IntelliJ IDEA, where the robot-server and For Mainframe plugins are running.

In order to locate the buttons, windows, etc. of the plugin, the UI test robot uses the XPath Query language. Once the robot-server plugin has been started, the IDEA UI components hierarchy can be seen on http://127.0.0.1:8580/, where 8580 is the default port. A hierarchy of IDEA untitled frame, Settings Dialog and Add Connection Dialog can be seen in Fig. 2. If we want to find a component fixture in the Add Connection Dialog, we just needs its XPath. However, both the Settings and the Add Connection dialogs have a Cancel button, which has the same XPath for both dialogs. That means that if you try to find the XPath right away, the robot-server will not know, which one you want. Because of this, we propose to first find the untitled frame. Then within the untitled frame you look for the Settings Dialog. Within the Settings Dialog you find the Add Connection Dialog and then finally you can find the component fixture you want.



(a) How it looks like in the IDEA.



(b) How it looks like in the browser on http://127.0.0.1:8580/. The Settings Dialog is in green frame. The Add Connection Dialog is in red frame.

Fig. 2: Add Connection Dialog opened from Settings Dialog in the For MAinframe plugin. Both dialogs are in an IDEA frame titled with the name untitled. The default port is being used.

Code 1: The top of the page http://127.0.0.1:8580/ in detail.

Also, from my limited experience, sometimes a uniform outcome of a test does not have to be guaranteed. The same test, with no changes to both the code of the test and code of the plugin, can result in both success and failure. That is due to the fact that there is an allocated time the robot-server searches for each component fixture. At some runs, the time has been exceeded. The solution for this unreliable behaviour was to increase time and add another step to the hierarchy. For example, we had troubles finding the add connection action button in the Settings Dialogue, so we have added another component fixture called ConfigurableEditor, which represents the For Mainframe option in the Settings Dialog.

Another thing to consider when proposing the approach for UI testing was what happens when a test fails and leaves opened unwanted container fixtures, mainly dialogs behind. Container fixtures are component fixtures, which contain other component fixtures, for example the Add Connection Dialog contains the OK and Cancel button and therefore it is a container fixture. These container fixtures can potentially result in the failure of the next test. In other words, there is a need for a tear down method, which would close unnecessary opened container fixtures after each test. In order to do that there is a need to track in which container fixture we are and its container path, by which we mean what container fixtures and in what order do we need to open in order to get to the current container fixture. We also need a storage, which stores the opened container fixtures. This is a good place to note that not all container fixtures, such as the ConfigurableEditor in our previous example need to be closed specifically. The ConfigurableEditor is closed along with the Settings Dialog.

Because of the above and other reasons we propose the following:

- Each search of any component fixture will have a long enough duration during which the robot-server looks for the component.
- Each container fixture function will have its components and actions that we need for the tests as either fields or methods.
- The "container path"will be tracked.
- The container fixtures, which will need to be closed after failed test will be tracked.

Component fixtures

Some basic component fixtures (that are not containers), like button are in the ui test robot library. Some, like Stripe Button, might need to be added manually. All manually added component fixtures are in
For-Mainframe/src/uiTest/kotlin/auxiliary/components. The stripe button can be seen in Code 2. For our purposes, it is enough to just create the StripeButtonFixture, annotate it with FixtureName and create a function, which would allow us to search for the stripe button within a container fixture. Unfortunately we do not completely understand why the FixtureName annotation is necessary.

```kotlin
/**
 * Function, which looks for the StripeButton.
 */
fun ContainerFixture.stripeButton(locator: Locator): StripeButtonFixture {
    return find(locator, Duration.ofSeconds( seconds: 60))
}

/**
 * This class represents the StripeButton.
 */
@FixtureName( name: "StripeButton")
open class StripeButtonFixture(
    remoteRobot: RemoteRobot,
    remoteComponent: RemoteComponent
) : ComponentFixture(remoteRobot, remoteComponent) {}
```

Code 2: The StripeButton component fixture.

Some components fixtures, such as ActionButton, have two states: enabled and disabled. When we want to click on these buttons, there is a need to wait a certain amount of time and check whether the button will not become enabled or disabled. Method checking whether the action button is enabled can be seen in Code 3 and found in
For-Mainframe/src/uiTest/kotlin/auxiliary/utils.kt.

```kotlin
/**
 * Waits 60 seconds for the action button to be enabled and then clicks on it.
 */
fun CommonContainerFixture.clickActionButton(locator: Locator) {
    val button = actionButton(locator)
    waitFor(Duration.ofSeconds( seconds: 60)) {
        button.isEnabled()
    }
    button.click()
}
```

Code 3: Checks whether the action button is enabled and clicks on it if it is.

Non closable container fixtures

Non closable container fixtures are the container fixtures, which do not need to be specifically closed like the ConfigurableEditor, which can be seen in Code 4. All closable and non closable container fixtures can be found in
For-Mainframe/src/uiTest/kotlin/auxiliary/containers.

In this class we define the filed conTab, which represents the z/OSMF Connections tab visible in Fig. 2a. Then we define the add method, which calls the Add Connection Dialog. This dialog is a closable container fixture and thus its XPath is added to the fixtureStack, which is a list of all XPaths of all currently opened closable container fixture ordered from the earliest to the latest. We also define its XPath as a static object.

```kotlin
/**
 * Calls the Configurable Editor, which is the For Mainframe section in the Settings Dialog.
 */
fun ContainerFixture.configurableEditor(function: ConfigurableEditor.() -> Unit) {
    find<ConfigurableEditor>(ConfigurableEditor.xPath(), Duration.ofSeconds( seconds: 60)).apply(function)
}

/**
 * The representation of the Configurable Editor, which is the For Mainframe section in the Settings Dialog.
 */
@FixtureName( name: "ConfigurableEditor")
class ConfigurableEditor(remoteRobot: RemoteRobot, remoteComponent: RemoteComponent) : CommonContainerFixture(remoteRobot, remoteComponent) {
    /**
     * The connection table
     */
    val conTab = tabLabel(remoteRobot,  name: "z/OSMF Connections")

    /**
     * Clicks on the add action and adds the Add Connection Dialog to the list of fixtures needed to close.
     */
    fun add(closableFixtureCollector: ClosableFixtureCollector, fixtureStack: List<Locator>) {
        clickActionButton(byXpath( xpath: "//div[@accessiblename='Add' and @class='ActionButton' and @myaction='Add (Add)']"))
        closableFixtureCollector.add(AddConnectionDialog.xPath(), fixtureStack)
    }
    companion object {
        /**
         * Returns the xPath of the Configurable Editor.
         */
        @JvmStatic
        fun xPath() = byXpath( xpath: "//div[@class='ConfigurableEditor']")
    }
}
```

Code 4: The ConfigurableEditor container fixture.

Closable container fixtures

Closable container fixtures are container fixtures which we might want to close in the tear down method. All of them have a close method, which closes them. When using them, we have to modify the fixtureStack. The approach we propose is to add their XPath to the fixtureStack in the method which calls them like we did in the add method in the ConfigurableEditor fixture and remove their XPath when the method, in which they are searched for, reaches its end. An example of this implementation is the SettingsDialog container fixture, which is being called from the Explorer container fixture and can be seen in Code 5.

```kotlin
/**
 * Finds the Settings Dialog and modifies the fixtureStack.
 */
fun ContainerFixture.settingsDialog(
    fixtureStack: MutableList<Locator>,
    timeout: Duration = Duration.ofSeconds( seconds: 60),
    function: SettingsDialog.() -> Unit = {}) {
    find<SettingsDialog>(SettingsDialog.xPath(), timeout).apply {  this: SettingsDialog
        fixtureStack.add(SettingsDialog.xPath())
        function()
        fixtureStack.removeLast()
    }
}

/**
 * Class representing the Settings Dialog.
 */
@FixtureName( name: "Settings Dialog")
class SettingsDialog(
    remoteRobot: RemoteRobot,
    remoteComponent: RemoteComponent
) : ClosableCommonContainerFixture(remoteRobot, remoteComponent) {

    /**
     * The close function, which is used to close the dialog in the tear down method.
     */
    override fun close() {
        cancel()
    }

    /**
     * Clicks on the Cancel button.
     */
    fun cancel() {
        clickButton( text: "Cancel")
    }
    companion object {
        const val name = "Settings Dialog"
        /**
         * Returns the xPath of the Settings Dialog.
         */
        @JvmStatic
        fun xPath() = byXpath( name, xpath: "//div[@accessiblename='Settings' and @class='MyDialog']")
    }
}
```

Code 5: The SettingsDialog container fixture.

Closable fixture collector

As mentioned above, the fixtureStack is a list of all XPaths that lead to the latest closable container fixture. Now we will introduce a class which collects the closable container fixtures and deletes them if needed. The class can be found in
For-Mainframe/src/uiTest/kotlin/auxiliary/closable/ClosableFixtureCollector.kt.

In the Code 6, we can see that the ClosableFixtureCollector has a field called items which is a mutable list of ClosableFixtureItem. Each ClosableFixtureItem has a name of a closable container fixture and list of all the XPaths needed to find it. The ClosableFixtureCollector then has methods for adding a new ClosableFixtureItem and closing it.

```kotlin
/**
 * The fixture that needs to be closed in the tear-down method.
 */
data class ClosableFixtureItem(
    var name: String,
    /**
     * List of all the xPaths of all its closable predecessors.
     */
    var fixtureStack: MutableList<Locator>
)

/**
 * Class which collects the fixtures that needed to be clsoed in the tear-down method and closes them.
 */
class ClosableFixtureCollector {
    /**
     * List of the fixtures that needed to be closed in the tear-down method.
     */
    var items = mutableListOf<ClosableFixtureItem>()

    /**
     * Adds closable fixture to the list.
     */
    fun add(xPath: Locator, stack: List<Locator>) {
        items.add(ClosableFixtureItem(xPath.byDescription, (stack + xPath).toMutableList()))
    }

    /**
     * Finds the closable fixture by its Locator.
     */
    fun findClosable(remoteRobot: RemoteRobot, locator: Locator) = with(remoteRobot) { this: RemoteRobot
        when(locator.byDescription) {...}
    }

    /**
     * Closes a single member of the items list.
     *
     * Is a recursive function. Should be called with i = 0.
     */
    fun closeItem(i: Int, item: ClosableFixtureItem, remoteRobot: RemoteRobot) {
        findClosable(remoteRobot, item.fixtureStack[i]).apply {...}
    }

    /**
     * Closes all closable fixtures, which we want to close.
     */
    fun closeWantedClosables(wantToClose: List<String>, remoteRobot: RemoteRobot) {
        for (item in items.reversed()) {...}
    }

    /**
     * Closes a single closable fixture by name if it exists.
     */
    fun closeOnceIfExists(name: String) {
        for (item in items) {...}
    }
}
```

Code 6: The ClosableFixtureCollector along with ClosableFixtureItem. The details of methods of the ClosableFixtureColelctor can be found in For-Mainframe/src/uiTest/kotlin/auxiliary/closable/ClosableFixtureCollector.kt.

Out of norm behaviour

Unfortunately, in the For Mainframe plugin there are some fixtures, which behave out of norm. We have identified one of them, which is the ErrorCreatingConnectionDialog. When we want to add connection to the plugin, but the input data are not alright, an error appears. In our tests, it is the Error Creating Connection Dialog. This dialog closes the Add Connection Dialog and if we hit the Cancel button in the Error Creating Connection Dialog, the Edit Connection Dialog opens. Because of this out of norm behaviour, we decided to modify the method, which searches for the dialog in the way we can see in Code

```
/**
 * Finds the Error Creating Connection Dialog.
 *
 * After this error, the Add Connection Dialog is changed into Edit Connection Dialog.
 * This function also takes this fact into account and
 * modifies the ClosableFixtureColelctor and fixtureStack accordingly.
 */
fun ContainerFixture.errorCreatingConnectionDialog(
    closableFixtureCollector: ClosableFixtureCollector,
    fixtureStack: List<Locator>,
    timeout: Duration = Duration.ofSeconds( seconds: 60),
    function: ErrorCreatingConnectionDialog.() -> Unit = {}) {
    find<ErrorCreatingConnectionDialog>(ErrorCreatingConnectionDialog.xPath(), timeout).apply { this: ErrorCreatingConnectionDialog
        for (item in closableFixtureCollector.items) {
            if (item.name == AddConnectionDialog.name) {
                item.name = EditConnectionDialog.name
                item.fixtureStack[item.fixtureStack.size-1] = EditConnectionDialog.xPath()
            }
        }
        closableFixtureCollector.add(ErrorCreatingConnectionDialog.xPath(), fixtureStack)
        function()
    }
}
```

Code 7: The method searching for the ErrorCreatingConnectionDialog container fixture.

Testing

Finally, in the ConnectionManager, we can find the UI tests. The testB checks whether it is possible to add a connection with an existing name. As we can see in Code 8, we go through container fixtures while modifying the fixtureStack. Once we find ourselves at the end of a closable container fixture, we delete it from the closableFixtureCollector.

```
@Test
fun testB(remoteRobot: RemoteRobot) = with(remoteRobot) {  this: RemoteRobot
    ideFrameImpl(projectName, fixtureStack) {  this: IdeFrameImpl
        explorer {  this: Explorer
            settings(closableFixtureCollector, fixtureStack)
        }
        settingsDialog(fixtureStack) {  this: SettingsDialog
            configurableEditor {  this: ConfigurableEditor
                conTab.click()
                add(closableFixtureCollector, fixtureStack)
            }
            addConnectionDialog(fixtureStack) {  this: AddConnectionDialog
                addConnection( connectionName: "a", connectionUrl: "https://a.com", username: "a", password: "a", ssl: true)
                ok()
            }
            closableFixtureCollector.closeOnceIfExists(AddConnectionDialog.name)
            errorCreatingConnectionDialog(closableFixtureCollector, fixtureStack) {  this: ErrorCreatingConnectionDialog
                yes()
            }
            closableFixtureCollector.closeOnceIfExists(ErrorCreatingConnectionDialog.name)
            configurableEditor {  this: ConfigurableEditor
                add(closableFixtureCollector, fixtureStack)
            }
            addConnectionDialog(fixtureStack) {  this: AddConnectionDialog
                addConnection( connectionName: "a", connectionUrl: "https://b.com", username: "b", password: "b", ssl: true)
                assertFalse(button( text: "OK").isEnabled())
                clickButton( text: "Cancel")
            }
            closableFixtureCollector.closeOnceIfExists(AddConnectionDialog.name)
            clickButton( text: "Cancel")
        }
        closableFixtureCollector.closeOnceIfExists(SettingsDialog.name)
    }
}
```

Code 8: UI test testing whether it is possible to add a connection with an existing name.

Lastly, let's point out the setUpAll method, which is tagged @BeforeAll. THis method opens the desired project from the welcome frame. The Thread.sleep method is being called to stop the execution of the program for a while. The reason is that sometimes, it took a bit longer for the IDEA untitled frame to be loaded and sometimes, tests failed. For now, we have solved it this way.

```
/**
 * Opens the project and Explorer.
 */
@BeforeAll
fun setUpAll(remoteRobot: RemoteRobot) = with(remoteRobot) { this: RemoteRobot
    welcomeFrame { this: WelcomeFrame
        open(projectName)
    }
    Thread.sleep(30000)
    ideFrameImpl(projectName, fixtureStack) { this: IdeFrameImpl
        forMainframe()
    }
}
```

Code 9: The setUp method.

final notes

You can run the IDEA needed for UI tests by going to the project root folder, building the plugin and then executing the following series of commands:

./gradlew runIdeForUiTest&

./gradlew uiTest

The first command will start up an IDEA, where the tests will be run. Please make sure the IDEA is visible at all times.