TuCSoN refactoring

Stefano Bernagozzi
stefano.bernagozzi@studio.unibo.it

August 2019

The purpose of this project is to refactor the TuCSoN Java library [8] [9]. In particular the main goal is to make all of the main five modules independent and to split the core into four sub-modules. Subsequently we focus on the dependencies of the main five modules, and especially all the external libraries that need to be imported from Maven and must be included in the jars. A side goal is to create JUnit test for end-to-end.A first attempt has been made in [10].

Contents

1. Goal/Requirements

The main goal of this project is to effectively divide TuCSoN Java library into 5 modules:

- **core**: The core module is the main module of the project, it contains classes and interfaces common to client, cli, inspector and service;
- **client**: The client module contains all the necessary classes to interact with service and tuple centres;
- **cli**: (command line interpreter) The cli module depends on the client and contains the necessary classes to start a command line interpreter that provides an interface to the service;
- **inspector**: The inspector module contains all the necessary classes to start the inspector and to dynamically inspect the tuple centers of a node;
- **service**: The service module contains all the necessary classes to start a service and to manage tuple centres and tuple spaces.

Each module would have the least dependencies upon others (ideally only upon core) and should be available as a standalone library.

Once the above separation is completed, the next steps are:

1. to create a fat-jar for each module,
2. to upload all the jars to Maven Central,
3. to automate the testing of end-to-end examples by using JUnit test suites,

4. to rename packages so as to avoid confusing developers with redundant names and to divide core into four different modules (tuple-template, tuplecentre, tucson and respect).

2. Work Plan

The work will be divided as (during all the phases, the project is continuously tested and all the tests in [10] must pass.):

2.1. Phase A - Module Division

1. At first a graph of classes and imports is made:
   - graphs of inner module dependencies
   - graphs of outer module dependencies

   The program, explained in [1], examines all the import rows in the classes, and in parallel it checks all the rows for the name of the classes (case sensitive) that are in the same module and that are not declared in the imports with the verbose mode flag. The verbose mode can have many false outcomes, in particular if a class contains in its name the name of another class (e.g. Inspector4Gui and Inspector). Despite that this allows to search for all dependencies, and the false positives are easily detected by using intellij tools to find usages of the classes.
2. The second step is to examine each module and check if some classes can be moved or are not used.
3. Then all modules are made standalone and all the common classes are put in core.

2.2. Phase B - Dependencies and Fat Jars

The second phase is related to the gradle build tool, in particular its goals are:

1. all the dependencies that aren't imported via gradle are updated and imported from maven central,
2. a task is added in order to create a fat jar for each module,
3. fat jars are tested and checked,
4. fat jars are uploaded to Maven Central.

2.3. Phase C - Test Automation

In the third phase the focus is on testing, with the purpose to automate it with the help of JUnit test suite.

2.4. Phase D - Core Division

The last phase is an ideal division of the core. Its final purpose is to have 4 submodules:

- Tuple-Template
- Tucson
- Tuplecentre
- Respect

This division however is almost impossible to achieve since there are many inter dependencies between those modules.

The last step of this phase is to rename all the packages under pika-lab.tucson.

2.5. Documentation

A side phase that belongs to the whole project is to document every step taken and the usage of the software. This allows future users to continue the work and to better understand the software.

3. Phase A - Module Division

3.1. Step 1 - Graphs

The first step of this phase is to generate all the graphs necessary to better understand the composition of the software. The diagrams and the program that generates them can be found at [1].

There are 3 types of graphs:

- **Inner Dependencies Graphs:** these graphs contain all the classes inside the module and the dependencies between them. This is useful to determine which classes can be isolated and extracted from the module.
- **Outer Dependencies Graphs:** these graphs contain all the classes inside the module and in addition the classes imported from other modules. They show the dependencies between inner module and outer module classes. The following outer module classes are excluded:
  - slf4j classes
  - core classes
  - java core classes
  - tuprolog classes

  This is useful to understand which classes are imported from other modules and can be moved to core.
- **Directories Graphs:** those graphs contain only the modules inside the main modules (the five ones in Goal/Requirements)

3.2. Step 2 - Analysis

From the analysis of the graphs mentioned in previous section it appears that all the classes imported from other modules are in service and they are:

- **AbstractTucsonAction:** used in service and client;
- **InspectorProtocolDefault:** used in inspector and service;
- **InspectorContextListener:** used in service and inspector;
- **InspectorContext:** used in inspector and core;
- **Inspector4GuiContextEvent:** used in service and inspector;
- **SearchableOpsQueue:** used in client and service;
- **CompletedOpsQueue:** used in client and service;
- **TucsonOpWrapper:** used in client and service;

Other important elements noticed are:

- The following classes are not used[1][2]:
  - respectReactionParser
  - Tucson2PLibrary
  - UnknownACCException
  - TucsonInvalidCommandException
  - IRespectProxy
  - inspector4gui
  - SpawnedWorkingActivity
  - RootACCProxy
  - OutputEventListener
  - GetS

All the classes found can be moved to the core module. No problems are found since there are no particular dependencies. Subsequently all the the Gradle files inside the modules are updated with core as unique dependency. In the end all the builds are executed with no errors.

[1] The analysis of the unused classes was first made with the python program and then with the intellij tool.

[2] N.B. Some of them are only libraries imported when needed by the programmer.

3.3. Step 3 - Stable separated version

Now all the various module are separated, each module has core as unique dependency and it can be launched standalone.

In order to launch The various modules the user needs to add 3 other libraries:

- **slf4j api** that can be downloaded from [6] (The version of slf4j api used was 1.7.28)
- **slf4j jdk** that can be downloaded from [6] (The version of slf4j jdk used was 14-1.7.28)
- **tuprolog** that can be downloaded from [11] (N.B. the version of tuprolog needed is 3.3.0)

After the download of the additional jars, each module can be launched as follow [3] [4]:

- **cli:** java -cp cli.jar;2p.jar;core.jar;slf4j-api-1.7.28.jar;slf4j-jdk14-1.7.28.jar alice.tuplecentre.tucson.service.tools.CommandLineInterpreter
- **inspector:** java -cp inspector.jar;core.jar;2p.jar;slf4j-api-1.7.28.jar;slf4j-jdk14-1.7.28.jar alice.tuplecentre.tucson.introspection.tools.InspectorGUI
- **service:** java -cp service.jar;core.jar;2p.jar;slf4j-jdk14-1.7.28.jar;slf4j-api-1.7.28.jar alice.tuplecentre.tucson.service.TucsonNodeService

For further information on how to use TuCSoN see [5]

4. Phase B - Dependencies and Fat Jars

4.1. Step 1 - Imports via Maven Central

The first step in order to create fat jars is to import all the libraries mentioned above from maven central. All of them are on maven central and can be imported via Gradle.

The libraries used are:

- version 1.7.9 of slf4j api
- version 14-1.7.25 of slf4j jdk
- version 3.3.0 of tuProlog [5]

[3] N.B. all the jars must be in the same directory

[4] These commands are for Windows users, for Linux and MacOs users the **;** must be replaced by **:**

[5] For tuProlog the newer libraries were tested, but the the Parser class was moved and in tuProlog 4.0.3 is hidden to the developer. Alongside this some exceptions classes are moved in a package exceptions.

4.2. Step 2 - Fat Jars

The subsequent step is to create a fat jar, a jar with all the dependencies inside. This allows users to launch the modules with the only command **java -jar modulename.jar**. The plugin used is the gradle shadow plugin [2].

To complete this process a new task was added to each build.gradle file, named customFatJar, that creates the fat jar under modulename/build/libs/modulename.jar.

4.3. Step 3 - Testing

The testing is focused on 5 tests, since they are the most suitable to check the correctness of the project. [6] These tests are:

- situatedness
- AsynchAPI
- distributedDiningPhilos
- diningPhilos
- timedDiningPhilos

Errors: The situatedness test blocks on a **IN** operation and it doesn't proceed.

4.4. Step 4 - Upload to Maven Central

Since not all the necessary requirements are satisfied and some tests don't pass, the upload on maven central was not carried out. For further information see appendix B

5. Phase C - Test Automation

In the third phase it is analysed if the tests in 4.3 can be refactored using JUnit to allow the user to check whether the test is passed or not. The tests are edited as below:

- in the 3 dining philosophers tests (distributed, timed and normal) the number of eats for each philosopher is limited to 5. This is done with a CountDownLatch. When a philosopher ends all the 5 eats, the latch (that counts the philosophers ended) is decreased. Meanwhile the main thread is waiting the latch for completion with a timer: if the latch ends before the timeout the test passes, otherwise it fails.
- in the asynchAPI test a similar method is used, but the latch counts the number of responses received from the master agent.

[6] All the tests need a service on the default port (20504) except for distributedDiningPhilos that needs also a service on port 20505. The programs were launched via intellij GUI and all the processes are executed on the same laptop.

- The situatedness test has not been modified since it didn't pass before.

Tests are executed in four different versions, as can be seen in appendix C: the service was launched both from terminal and from code, and the tests were launched both from a main class and from test suite. [7] [8]

6. Phase D - Core Division

The last stage of this project is to divide the core into 4 parts:

- **tuple-template:** with all the classes related to logic tuple and tuple templates.
- **tuplecentre:** with all the classes related to tuplecentre (tuplecentre.core and tuplecentre.api packages), excluded tucson and respect classes.
- **tucson:** with all the classes related to tucson core (tuplecentre.tucson package).
- **respect:** with all the classes related to respect core (tuplecentre.respect package).

Currently the tuple-template package is almost detached from the rest of the core and with few changes it can be moved to a new standalone module. Obviously the remaining part of the core should depend from it, as can be seen in the graph core-graphs/outer-dependencies-tuplecentre.svg in [1].

In order to complete the isolation, the following 2 classes need to be moved or edited:

- **InvalidOperationException** that is used in 4 classes
- **InvalidTupleException** that is used in 5 classes

For the relation between those classes and the other part of the core please refer to graphs-core/outer-dependencies-tuple.svg in [1]

The division of the remaining packages of the core is more complex since there is a high inter dependency between them, as can be seen in the outer-dependencies graphs of core, api, tucson and respect in [1]

7. Future Work

Some future improvements that can be done after this project are:

- complete the core, in particular the isolation of the tuple package, and then check if tuplecentre can be splitted in tucson, respect and tuplecentre.
- make JUnit tests for each class and method, starting from service basic operations (like IN, OUT, ...) to get to the examples given in 4.3.

[7] The suite used for the tests is JUnit 5 jupiter [7]. All the tests can be found in RunTests, LaunchWithService and RunTestsWithService classes.

[8] The tests with the service launched from code are in separate git branch (test-with-service-error).

- insert the missing information of the artifacts (group id, domain ownership), rename the packages and upload them to maven central.
- go deeper in test errors and find where they are wrong (in particular the situatedness test).

References

[1] Stefano Bernagozzi. Classes and packages graphs, 2019. URL https://github.com/ste93/classes_graph.

[2] John Engelman. Gradle shadow plugin, 2019. URL https://plugins.gradle.org/plugin/com.github.johnrengelman.shadow.

[3] The Apache Software Foundation. maven upload, 2019. URL https://maven.apache.org/repository/guide-central-repository-upload.html.

[4] The Apache Software Foundation. maven upload specific, 2019. URL https://central.sonatype.org/pages/requirements.html.

[5] Andrea Omicini and Stefano Mariani. The tucson model & technology. a guide, 2015. URL https://www.slideshare.net/andreaomicini/the-tucson-coordination-model-technology-a-guide.

[6] slf4j. slf4j download site, 2019. URL https://www.slf4j.org/download.html.

[7] The JUnit Team. Junit 5 website, 2019. URL https://junit.org/junit5/docs/current/user-guide/.

[8] TuCSoN. Tucson home, 2015. URL http://apice.unibo.it/xwiki/bin/view/TuCSoN/.

[9] TuCSoN. Tucson repository, 2015. URL https://bitbucket.org/smariani/tucson/src/master/.

[10] TuCSoN. Tucson refactored repository, 2019. URL https://gitlab.com/pika-lab/tuples/tucson.

[11] Tuprolog. Tuprolog download site, 2019. URL https://bitbucket.org/tuprologteam/tuprolog/downloads/.

A. Project Graphs

A.1. Directories Graph

The directories graphs show for each module the main package and its sub packages connected by arrows.

A.2. Inner Dependencies Graph

The inner dependencies graphs show for each module all the classes inside it and their the relative imports that are in the same module. The arrow A $->$ B means that the class A imports the class B.

A.3. Outer Dependencies Graph

The outer dependencies graphs show for each module all the classes inside it and their relative imports that are not in the same module. The following classes are excluded from the graph:

- classes of the SLF4J library;
- classes of the Java standard library;
- classes of the tuprolog library;
- classes of the core module.

As above, the arrow A $->$ B means that the class A imports the class B.
The colors meaning are:

- BLUE: the class belongs to the module;
- RED: the class belongs to the project but is located in another module (the path is specified in the name);
- PURPLE: the class doesn't belongs to the project or to the categories mentioned above.

B. Upload on Maven Central

As per maven's instruction [3] [4] the needed in order to upload an artifact to maven central are:

- **releases:** Only releases can be uploaded to the Central Repository, that means files that won't change and that only depend on other files already released and available in the repository,
- **javadoc and sources** for IDE lookup,
- **PGP signature,**
- **minimum POM information:** There are some requirements for the minimal information in the POMs that are in the Central Repository.
- **coordinates:** Picking the appropriate coordinates for your project is important, particularly on groupId and domain ownership.

This project, in order to be uploaded to maven central, must have in addition:

- a javadoc
- a release for each of the jars (service, core, cli, client and inspector)
- a PGP signature
- all the metadata: group id, artifact id, version, projectName, description, url, license information, developer information and SCM(source control system) information. Those information must be added to the POM of the project.

C. Test Log

As can be seen in Table 1 not all the tests performed have passed. Errors and the logs obtained from them are reported below and for the relation between tests, logs and errors please refer to Table 1.

Problem 1: the logs are all correct but the program doesn't terminate.

Problem 2: it throws an unreachable node exception but the test passes.

Problem 3: the master doesn't receive any response from the workers as can be seen in C.1

Problem 4: sometimes it performs an infinite loop of **INP** operations as in C.2

Problem 5: sometimes there is a high delay, maybe some operations are pending. TestDDiningPhilo after 5 minutes failed due to timeout, normal time is about 1m30s. The delay happens with any test randomly, no pattern is found. Timeouts are not considered and it doesn't terminate (as in C.3).

Problem 6: sometimes the second test gives no more solutions, also with timeouts between the two tests (as in C.3).

C.1. Log 1

[master]: Handlers registered, received 0 in successful completions, 0 inp successful completions, total copyOf 0

This log is repeated for each of the 50 tasks, then wait there for the termination

N.B. when this log appears the test fails

C.2. log 2

```
alice.tuplecentre.tucson.service.OperationHandler log
    INFO: ....[OperationHandler (helper4worker2)]:
    requesting op INP, calcprime(X), '@'(default,':'(localhost,'20504'))
alice.tuplecentre.tucson.asynchSupport.AsynchOpsHelper log
    INFO: ....[AsynchOpsHelper (helper4worker1)]:
    doing op inp(calcprime(X)) to '@'(default,':'(localhost,'20504'))
alice.tuplecentre.tucson.service.OperationHandler log
    INFO: ....[OperationHandler (helper4worker1)]:
    requesting op INP, calcprime(X), '@'(default,':'(localhost,'20504'))
alice.tuplecentre.tucson.asynchSupport.AsynchOpsHelper log
    INFO: ....[AsynchOpsHelper (helper4worker0)]:
    doing op inp(calcprime(X)) to '@'(default,':'(localhost,'20504'))
```

```
alice.tuplecentre.tucson.service.OperationHandler log
    INFO: ....[OperationHandler (helper4worker0)]:
    requesting op INP, calcprime(X), '@'(default,':'(localhost,'20504'))
```

C.3. Log 3

```
INFO: [RespectVMContext ('\$OBS'@localhost:20504)]: No more solutions.
alice.tuplecentre.tucson.service.TucsonNodeService install
INFO: Setting up Management Service...
alice.tuplecentre.respect.core.RespectVMContext log
INFO: [RespectVMContext ('\$ORG'@localhost:20504)]: No more solutions.
```

C.4. Log 4

```
Exception in thread "pool-1-thread-7" java.lang.NullPointerException
at alice.tuple.logic.LogicMatchingEngine.propagate
    (LogicMatchingEngine.java:46)
at alice.tuple.logic.LogicTupleDefault.propagate
    (LogicTupleDefault.java:112)
at alice.tuplecentre.respect.core.RootInterface.unify
    (RootInterface.java:37)
at alice.tuplecentre.respect.core.OrdinarySynchInterface.inp
    (OrdinarySynchInterface.java:120)
at alice.tuplecentre.tucson.service.TupleCentreContainer.doBlockingOperation
    (TupleCentreContainer.java:107)
at alice.tuplecentre.tucson.service.ACCProvider.shutdownContext
    (ACCProvider.java:229)
at alice.tuplecentre.tucson.service.ACCProxyNodeSide.run
    (ACCProxyNodeSide.java:466)
```

C.5. Other Errors

- For testPrimeCalculator there is also a null pointer exception in the service (see log 4 C.4).
- When the services are stopped in testDDiningPhilo, some pending actions are executed (they are executed also after the end of the test).

Table 1: Table with all the tests executed

	LAUNCHED FROM MAIN		**LAUNCHED FROM TEST**	
test	**service from terminal**	**service from main**	**service from terminal**	**service from test**
testPhilo	passed	Problem 1 C	passed	passed
testDDiningPhilo	passed	Problem 1 C	passed	Problem 2 C
testTimedPhilo	passed	Problem 1 C	passed	passed
testPrimeCalculator	passed	Problems 1 C and 3 C	Problem 4 C	Problems 2 C and 3 C
all in sequence	Problem 5 C	Problems 5 C and 6 C *	Problem 5 C *	Problems 5 C and 6 C *

N.B. when the test are executed all together the order is random.
* The errors mentioned are the ones that results from the sequential execution, each test keeps the errors referenced in the rows above.