INTERNSHIP PROJECT

ASTracker

Ana Roman
S2763753

supervised by
Darius Sas & Paris Avgeriou

May 12, 2020
University of Groningen

1 Introduction

In software development, architectural smells and code smells are symptoms of bad code or design that can give an indication over quality problems, such as errors, technical debt, or difficulties with maintenance and evolution [1]. In order to study the evolution of architectural smells not only on a coarse-grained level, but along the system's evolution, the tool **ASTracker** has been developed. This tools makes use of the results found by an open source tool named Arcan [2], which tracks individual smell instances in each version of a system, and uses them to measure and monitor the evolution of the properties of each detected instance throughout the evolution of a system.

ASTracker exists as a standalone application developed in Java and implemented using the Spring [1] framework. However, in order to run the current version of the tool, the project needs to be compiled into a `.jar` file which then can be invoked through the Command Line Interface (CLI) with a list of flags provided which can customize the way the program is executed. To make the usage of the tool more accessible and user-friendly, a web interface could be used that should have the same effect as running the tool through the CLI.

The present document describes the changes and additions made to the tool in order to make it possible to provide access to the tool and the analysis, by providing a list of Application programming interfaces (APIs) that a client application, such as an interface, could communicate with.

2 Problem Description

The following section will take a look at the functionalities of the provided tools that are involved when using ASTracker, as well as investigate the design of the system. Afterwards, the requirements will be analysed, together with the changes that should be implemented when trying to satisfy the set of requirements.

2.1 Arcan

As mentioned previously, ASTracker makes use of Arcan in order to link architectural smells between the versions of a system. In this context, different versions will be the contained within the different commits found in a repository, in the case of a project found in a VCS system.

Arcan is a tool that analyses a system's versions and, for each version, produces a `.graphml` file containing all the architectural smells detected in the respective version. In the current system, Arcan is provided as a standalone application, and more explicitly, a `.jar` file inside the ASTracker project directory. ASTracker can invoke it and run it on a given project, which will trigger Arcan to analyse the respective project. The output of Arcan will be one or more `.graphml` files, which in turn will serve as input for ASTracker.

Based on the run command parameters, Arcan can have different behaviors. for example it can analyse a single version of a project instead of the whole repository; it can also be given a certain start date, and then only the commits made after that date will be analysed.

[1] Spring official site: `https://spring.io/`

As mentioned before, Arcan can be executed as a standalone application through a CLI command. An example of such a command can be seen below. A similar command is used inside ASTracker to trigger Arcan.

```
java -Xmx5G
    -jar Arcan-1.4.0-SNAPSHOT.jar
    -git -p pyne
    -out arcanOutput\pyne
    -branch master
    -startDate 1-1-1
    -nWeeks 2
```

Arcan can analyse projects that are written in both Java and C; however, this can be done by using two different versions of the application. As such, two `.jar` files are present inside ASTracker, one for each type of project. Both of them will be used by ASTracker, and both have almost identical functionalities and options that can specify how the analysis should be performed. Furthermore, both versions will output files in the same `.graphml` format, which can be further analysed by ASTracker.

2.2 ASTracker

ASTracker is a tool written in Java that parses Arcan's output and tracks the architectural smells detected in each version analysed by Arcan. It is written with the help of the Spring framework, and uses Maven for dependency management. The source code of the tool can be found on the GitHub repository [2], together with more explanations regarding the structure of the project, its modules and how it can be used.

In its initial state, the tool can analyse and track architectural smells throughout multiple versions of a project situated locally, on the same file system as the source code of the project. The source code can be packaged as an application, and the analysis can be triggered through the CLI. However, starting an analysis like this is not very straight-forward or intuitive, since a lot of parameters have to be specified (for example, input and output directories). At the same time, the tool can only analyse projects written in Java, and does not allow the analysis of a single version of a project.

As mentioned on the official repository, an example of such a CLI command that could trigger an analysis on the `antlr` project is the following:

```
java
    -jar target/astracker-0.9.0-jar-with-dependencies.jar
    -i sample-data/antlr
    -p antlr
    -o sample-data/antlr
    -pC
```

A general overview of the communication process between Arcan and ASTracker can be seen in figure 1. Due to the fact that the project was already implementing using the Spring framework, some functionality was already present that allowed the application to be ran as a standalone local server. As such, some endpoints were already existing, for example an endpoint that would allow the user to view the results of previous analyses.

[2] GitHub URL: `https://github.com/darius-sas/astracker`

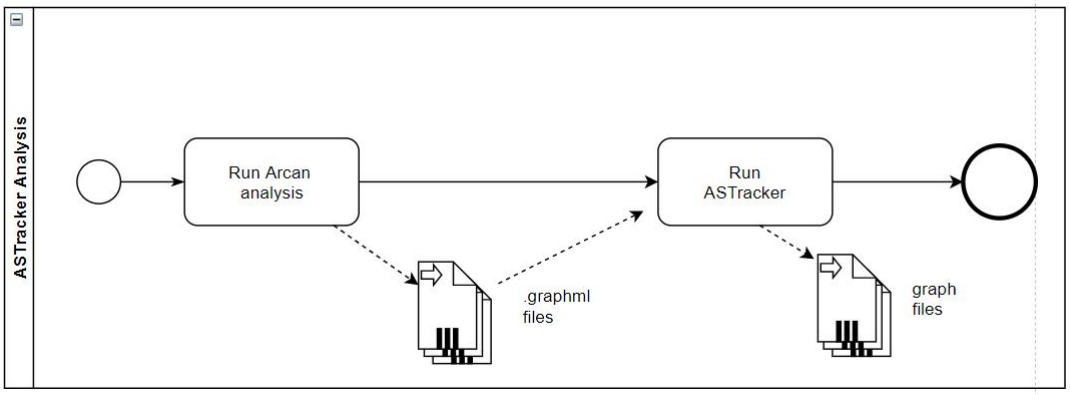

Figure 1: Overview of the interactions between Arcan and ASTracker

2.3 Requirements

When developing the code for integration of the tool, there was a list of requirements that had to kept in mind. The majority of the requirements were already specified in the project description, and some of the smaller specifications came up during the development process.
The list of requirements is presented below.

2.3.1 Functional requirements

1. Implement a way in which the tool could start the analysis of a project when a HTTP request is being sent to the locally running application.
   According to this requirement, the tool should be able to start a complete analysis of a project when a request is being sent to a locally running server which hosts the ASTracker tool. The request should specify, through it's argument list, which project should be analysed, the programming language of the project, and all the other arguments required for the analysis to take place, for both Arcan and ASTracker. The behaviour of the tool should be, as much as possible, identical to its behavior when invoked through the CLI, and the results of the analysis should be identical in both cases.

2. The request sent to the application should follow a set of given guidelines.
   The request should only contain a few parameters, and the rest of the options should be deduced from the parameters or set inside the tool. This implies that the tool should be able to locally download a project from a remote repository and perform a clean analysis on it.

3. Add the functionality to the tool that would allow it to analyse a single version of a system, instead of multiple, and allow the system to check for previous analyses.
   The initial version of the tool can analyse multiple versions of a system. It does so by checking out a certain version, analysing it, and moving on to the next ones until the most current version is reached. The new system should be able to perform the analysis on a single version only, namely the currently checkout one. In order to do this, the system should also be able to save and load the state of an already analysed project, such that the analysis of newer versions can be started from the last analysed version instead of starting from the beginning.

4. Extend the functionality of the tool by adding the possibility of analysing a remote project from a public VCS repository, such as GitHub or GitLab.

5. Extend the current functionality of analysing Java projects with the possibility of analysing projects written in C.
   This can be made possible thanks to the fact that both the Java and C Arcan runners output files in the same format, and both can be interpreted by ASTracker. As such, ASTracker should be able to invoke both the Java and C Arcan runners and then perform its analysis on the outputted `.graphml` files.

2.3.2 Non-Functional requirements

- **Testability**
  Testability is an important requirement of the project. In order to ensure the correct functionality of the code, unit tests should be written for all the new classes and additional functionality.

- **Quality & Maintainability**
  The project should maintain the same level of code quality throughout the whole process, without introducing bugs or faulty features, so that it can be easily maintained and further extended upon.

3 Problem Design and Implementation

The following section will give a broad explanation and argumentation of the solution design, when implementing the functionality mentioned in section 2.3.1. This will be done by documenting the new components that have been added to the system, together with an explanation of their functionalities as well as tracing back to the requirements that they help achieve.

Since ASTracker was an already existing system with a broad set of functionalities, the focus of the section will be more on the newly added features rather than the existing ones. When developing the new features, the non-functional requirements mentioned in section 2.3.2 were kept in mind as strict guidelines to be followed.

Figure 2 gives a broad overview of how the system will process an incoming request to the server. The components, together with the interactions between them will be explained in the following section.

3.1 Implementing the requirements

In order to start the analysis by sending a request to a server, we need to set up ASTracker to run within a local server. Since the tool already had the foundation built in Spring, this task was rather straightforward to implement since Spring has certain functionalities that allow the mapping of a URL to a specific function inside a class that has to be annotated as a `@RestController`.
Below, a list of the newly added classes can be found.

- `WebAnalysisController`
  This will act as a central point for the incoming HTTP requests to the server. As mentioned previously, Spring will see it as a REST controller, and will direct all the incoming requests that match the mapped URLs to their specific functions.
  Here we will define the `/analyse` endpoint, which clients could call in order to start the analysis of a project. If, in the future, more endpoints should be added to the project, this is the class where the endpoints could be defined.

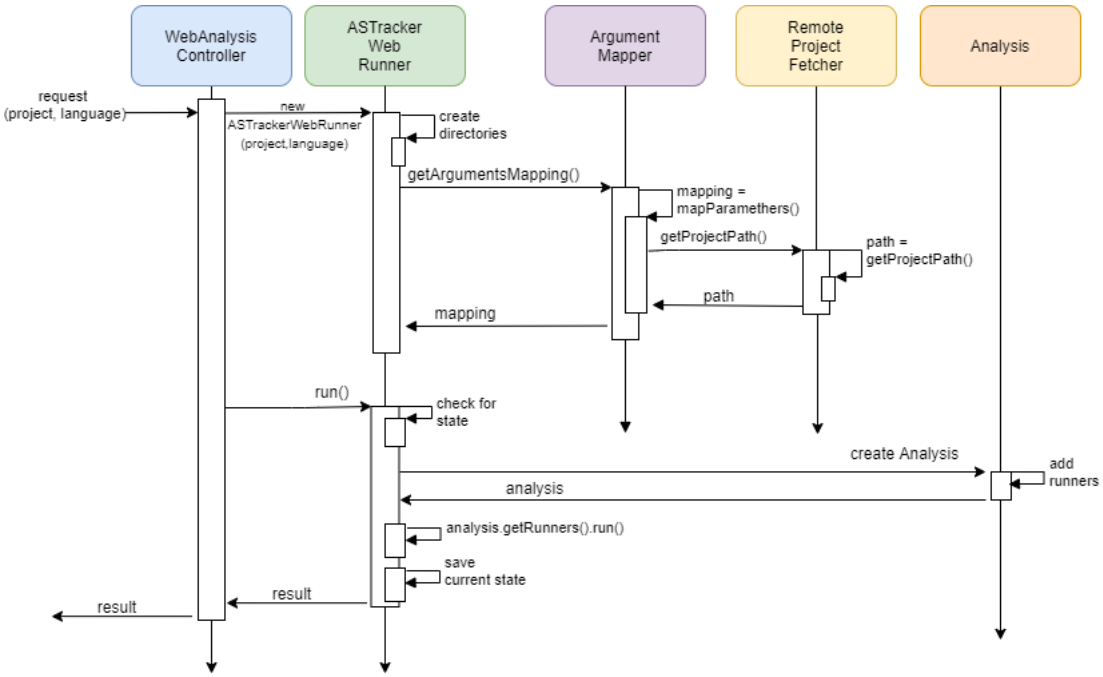

Figure 2: System Sequence Diagram

This will be the API that will start ASTracker together with all its' sub-processes. It instantiates the `ASTrackerWebRunner`, calls its' `run()` method, and sends back to the client the result that the method returns.
There are three possible responses that this endpoint can return:

1. `SUCCESS` - when the analysis has been successfully performed, and the results can be found in the expected files/folders.
2. `FAILURE` - if an error has occurred or the analysis could not be performed on the given project.
3. `SKIPPED` - if the analysis has been performed before on a project that has no new versions.

- `ASTrackerWebRunner`
  This class is responsible for the 'heavy-lifting' of performing the analysis, using the parameters received from the client. The results obtained after running this class should be in line with all the specifications of requirements 1.
  The class performs a few steps, which are enumerated below:

  1. When instantiated, the object creates all necessary directories and files, and also instantiates an `ArgumentMapper` object.
  2. When ran, it first checks whether the given project has been analysed before, and based on this decision it instantiates a `Analysis` object.

     It does so by looking at whether there exists a `version.seo` file in the project saved files directory, for the project that is currently analysed. For example, the presence of the file

`states/pure/version.seo` would indicate a previous analysis of the Pure project.
There are two possible scenarios in this case:

- If the project has never been analysed before, a new `Analysis` object will be created, which will contain a list of newly instantiated runners needed for the analysis.
  In this context, runners are the objects that will 'dig' into the project and perform the actual analysis, for example, tracking code smells.
- When the project has been analysed before, the state of the old analysis will be loaded, and a runner will be instantiated with the information about the previous analysis. The runner will be then used to create a new `Analysis` object, and the analysis can now be performed from the last state analysed, when there are new versions present, without having to re-analyse the old versions.
  If there are no new versions detected, the analysis will stop and a message will be returned to the user letting them know that the analysis has been skipped.

3. Executes the `run()` methods of all the runners inside the `Analysis`
4. Saves the current state of the analysed project and returns a message to the user.

- `ArgumentMapper`
  Since the original version of the project worked with CLI arguments, a challenge was to create a mapping between the arguments received from a request and the CLI arguments. If such a mapping could be found, the analysis could be performed in the same way as before, by reusing the same code but by passing the arguments mapped from the request. The `ArgumentMapper` class is a helper class that performs this mapping.
  According to requirement 2, the requests that the clients make should follow a given format. The request should have at least two parameters, namely `project` and `language`, which should point to the project name/link and the programming language, respectively. An example of such a request is the following:

```
http://localhost:8080/analyse?
       language=java&
       project=https://github.com/darius-sas/pyne.git
```

  The `ArgumentMapper` looks at the `project` request parameter:

```
project=https://github.com/darius-sas/pyne.git
```

  and uses it to extract from it information. For example, it can extract the name of the project and it can identify whether the value is a URL to a VCS repository or the name of a project that is already present. If the value of the parameter is a valid link to a VCS repository, a `RemoteProjectFetcher` object is instantiated.

- `ArcanArgsHelper`
  In order to analyse only the currently checked out version of a system, the first step is to let Arcan perform the analysis on that version of the project. This can be done by passing the `-singleVersion` flag to the list of arguments that are used when running Arcan. The `ArcanArgsHelper` class handles this case, by mapping the third value of the HTTP request (if present) to the Arcan command. The class also adds all the rest of the required parameters that

are needed for running Arcan in this case.
An example of such a request is the following:

```
http://localhost:8080/analyse?
        language=java&
        project=https://github.com/darius-sas/pyne.git&
        singleVersion=true
```

- `RemoteProjectFetcher`
  According to requirement 4, ASTracker should allow the clients to specify a link to a remote project that they would like analysed. In this case, the project should be cloned locally and the analysis performed on it.
  The `RemoteProjectFetcher` will handle the downloading of the files and will return the path to the project on the file system, in the case when the project is not present locally. If the project is already downloaded, then the path will be returned without performing any other actions.

  To note here is that it doesn't matter whether the latest version of the project is present on the file system or an older version, since Arcan can check out to specific commits by itself. As such, it is only important that the a version of the project is present, and that the file system location is a local version of a VCS repository.

- `AbstractGitArcanRunner`

  According to requirement 5, ASTracker should be able to analyse projects written in both Java and C, by invoking the correct Arcan application on a project repository and then analysing the outputted files. In order to implement this requirement, two new classes have been defined that will handle each type of project: `GitArcanJavaRunner` and `GitArcanCRunner`. Having two separate classes was required due to the fact that the two Arcan applications need different parameters to be specified during execution, and as such, different lists of arguments have to be extracted. Both of these classes will inherit from an abstract parent, `AbstractGitArcanRunner` since both objects are initialized with the same list or arguments which could be extracted in a parent class. An overview of this structure can be seen in figure 3.
  When an `Analysis` object is created in the `ASTrackerWebRunner`, it will use the parameters from the argument mapping to decide what runners it should add - this functionality was already present. However, in the new version of the system, the Analysis will look at the `language` parameter in order to decide which of the two new runners it should add - either the Java or the C Arcan runner.

4 Evaluation

The tool can be evaluated by analysing projects and comparing the output of the old version of the system (where the tool was invoked through the CLI) with the output of the new version (starting the analysis when a request is sent to the server). The evaluation consists in comparing the outputted files and examining whether they are identical or not. Identical outputs would indicate that the tool's output

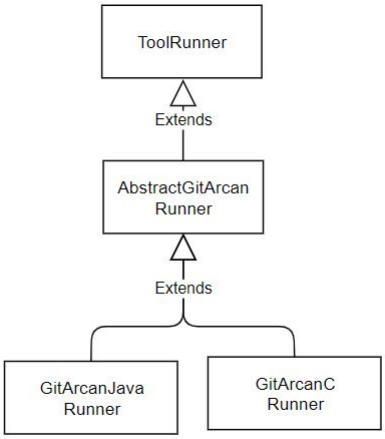

Figure 3: The structure of the two different Arcan runners

when the analysis is triggered through an HTTP request has not changed and is still correct, which would indicate that requirement 1 has been successfully satisfied.

The command to run ASTracker through the CLI on the Pyne project is the following:

```
java -jar target/astracker-0.9.0-jar-with-dependencies.jar -pC
    -i cloned-projects/pyne
    -p pyne
    -o cli-output-folder
    -runArcan arcan/Arcan-1.4.0-SNAPSHOT/Arcan-1.4.0-SNAPSHOT.jar
    -gitRepo cloned-projects/pyne
```

The request that needs to be sent to the server is the following:

```
http://localhost:8080/analyse?
        language=java&
        project=https://github.com/darius-sas/pyne.git
```

Both methods result in two `.graphml` files, whose contents are indeed, identical. As such, we can conclude that our evaluation is successful and that the tool's behavior is correct.

5 Conclusion

The most important requirement of this project was to change the behavior of an already existing system, to allow the usage of the system through a request sent to a server instead of running the tool through the CLI. The present document detailed the steps taken when bringing those changes, together with the other requirements and the evaluation of the new version.

Improvements and future work consist in building an interface for the tool, creating more endpoints and perhaps further customizing the arguments of the analysis.

References

[1] F. A. Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi, "Are architectural smells independent from code smells? an empirical study," *CoRR*, vol. abs/1904.11755, 2019.

[2] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 282–285, 2017.