AU4J

Application Understanding for Java

From the eighties of the last century, Legacy Applications have always been a serious issue of the IT. A decade ago, “Legacy” meant COBOL or something similar to COBOL, as the Natural language of Software AG, or the RPG of IBM. In the last ten years, however, the mainstream technology has changed dramatically, also affected the Java community. The Cloud has rendered some aspects of Java EE useless, and eventually Java EE was removed from Oracle Java portfolio. Microservices are going to replace monolithic servers deployed in heavyweight containers.

This encouraged us to extend our existing Application Understanding technology to the Java world, and the result of that effort is the AU4J product.

Wherever you want to go with your legacy code, either redevelop it from scratch, or continue the brownfield development, or just decide between the above two, you will need to understand the business logic the application embodies. Extracting the business logic from an existing code base is not an easy task, however. Developers have to examine hundred thousand or even million lines of code.

Application Understanding provides a brand-new approach to this process[1], to extract the business logic from the application code. Code analysis traditionally goes top-down, starting from the entry point(s), and traversing the directed graph of the control flow. Things quickly become unclear, get out of hand, because complexity grows exponentially on the way. But Application Understanding takes the opposite direction. It detects the particular points in the code, where a piece of persistent data is created (called *output endpoint*’s). From there, the traverse goes backward, gathering the code fragments that are involved in creation of the persistent data. The result is the pure business logic, excluded any other aspects of the architecture, e.g. logging or management.

As a process, the business rule extraction is somehow similar to *angiography*, a medical imaging technique used to visualize the blood vessels[2]. Developer inject paint to one or more endpoints of the system, and look at what has become colored.

The AU4J tool is integrated with the IDE (actually a JetBrains IntelliJ IDEA plugin is available), so developers can process the result list in their familiar environment.

In addition to the business rule extraction, the AU4J tool (which is a Web application) provides several functions to support application maintenance and brownfield development.

- *Module hierarchy* tool – Shows the module hierarchy of the system as an expandable tree. Displays module statistics as well.
- *Entities* tool – Lists the persistent entities of the application. On click it reveals the structure of an entity. If a property is a reference, on click it shows it’s structure, too.
- *Paint* tool – extracts the business logic by exploring the codes and datasets that are connected to the selected endpoints

[1] cf. Business Rule Extraction in Application Modernization Projects © Copyright 2015, Don Estes
[2] in Wikipedia https://en.wikipedia.org/wiki/Angiography

- *Dead code detection* – Lists the unused methods in the application . The algorithm is recursive: dead code is uninvoked or only invoked by dead codes.
- *Cyclomatic complexity3* – Displays total and average complexity4 of the elements of the module–class–method hierarchy. The tool helps resource planning for testing, and points to methods that are difficult to maintain in the code.
- *Find SQL* tool – identifies the line(s) in the code that result in the specified SQL or JPQL query.
- *Queries* tool – Lists the named, native and inline queries within the code. On click it shows the formatted source of a query both in JPQL and SQL language.
- *Impact analysis* – AU4J is integrated with the version management system that manages the application's code. If you enter a commit identifier, it returns the list of use cases and interface functions affected. The tool reduces the resource requirements for regression testing , and allows you to evaluate the security impact of the change.
- *Vocabulary analysis* – Collects the words used as names in the application (and the decomposition of the compound names). Enables clustering of application concepts. You can go through all usages of a particular word. It helps to standardize the language of the system, and develop a common vocabulary for the domain experts and the developers. Allows you to match phrases (in mixed language codes).
- *Security analysis* – The domain expert can assign *value of damage* to the data files and tables. The tool handles damages of *data leak* and *data hack* separately. The tool aggregates the damage (that can be caused through them) to entry points of the application.

[3]https://en.wikipedia.org/wiki/Cyclomatic_complexity

[4]The rules of our calculation:

- Methods have a base complexity of 1
- +1 for every control flow statement (if, case, catch, do, while, for), and conditional expression
- +1 for every boolean operator (&&, ||) in the guard condition of a control flow statement
- For classes and modules the complexity is an aggregation of the methods below.