Report for assignment 4 - DD2480

Anders Sjöbom, asjobom@kth.se
Albin Remnestål, albinre@kth.se
Johan Bergdorf, bergdorf@kth.se
Marcus Wengelin, wengel@kth.se
Robert Wörlund, worlund@kth.se

Project

Name: Jarvis
URL: https://github.com/remnestal/Jarvis

Project description:
Described as: “Personal Assistant for Linux and macOS”.

Index of relevant data produced in the project

- coverage metrics before tests were added
- coverage metrics after tests were added
- CCN analysis before refactoring
- CCN analysis after refactoring
- Branch coverage analysis using ad-hoc measurements

Complexity

We used lizard and manual calculations to check the complexity of the Jarvis project.

Figure 1 shows a run of lizard and some of the most complex functions that we decided to work with during the assignment.

```
================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
     75     27    552      1      96 parse_date@64-159@./utilities/textParser.py
     70     21    379      3      70 main@8-77@./packages/weather_pinpoint.py
     65     20    503      1      70 score@101-170@./packages/cricket.py
     48     16    279      1      53 main@7-59@./packages/translate.py
     42     14    302      2      53 parse_number@9-61@./utilities/textParser.py
     43     11    237      2      57 request_news@138-194@./packages/news.py
     21     11    224      4      26 organise@72-97@./packages/file_organise.py
     15     11    146      2      21 precmd@54-74@./Jarvis.py
     29     10    174      1      35 reminder_handler@156-190@./packages/reminder.py
     29     10    151      1      32 get_opt@90-121@./packages/news.py
```

Figure 1. Output from running lizard -s cyclomatic_complexity -C 9 -x "./tests/"

In Table 1 below we compare the CCNs we have calculated by running lizard and by manual calculations from flow-chart diagrams. The github issues contain links to the flow chart

diagrams. For the *precmd@Jarvis.py* and *organise@file_organise.py* functions the two persons originally assigned got a different result and hence a third person made an additional flow chart and a final CCN was decided upon by all three persons.

Table 1. Comparison between CCN from lizard and manual calculations

Method	Lizard CCN	Manual CCN	Github issue
request_news@138-194@./packages/news.py	11	10	#3
organise@72-97@./packages/file_organise.py	11	10	#2
precmd@54-74@./Jarvis.py	11	8	#4
reminder_handler@156-190@./packages/reminder.py	10	9	#1
get_opt@90-121@./packages/news.p	10	9	#5

1. **Did all tools/methods get the same result?**
   We only used lizard as a computer-aided tool. Most of the times lizard and our manual calculations yielded a fairly similar result, but never exactly the same. It seems that lizard in many cases has a CCN of one number higher than our manual calculations.
2. **Are the results clear?**
   The results, i.e. the output from the lizard tool, is very clear.
3. **Are the functions just complex, or also long?**
   The functions are both complex and long. This is, of course, subjective, but we feel that functions which occupy more than half a page can be considered long.
4. **What is the purpose of the functions?**
   The functions are quite diverse, and deal with many different issues. The function *"request_news"* requests news articles from a given source, *"organise"* organises the data on the filesystem, *"precmd"* is a hook which executes before every given command, *"reminder_handler"* handles the parameters for the reminder function call and *"get_opt"* reads user input used for selecting a news source. See the section below for a more detailed description of all the functions.
5. **Are exceptions taken into account in the given measurements?**
   Yes, exceptions were taken into account when creating the flowcharts and calculating the CCN.
6. **Is the documentation clear w.r.t. all the possible outcomes?**
   No, unfortunately the functions are barely documented at all. For example, the *parse_number* function has a description but contains a lot of hidden functionality not mentioned in any description of the function regarding what should happen for different mixtures of numerals and literals. A clear example of this is: When given the string "10 thousand" it results in 10000 as expected. However there is no mention of what the expected results of "10 thousand thousand" should be or if you chain it even more, which resulted in 11000. However, it is not consistent as the input string "10 hundred thousand" behaves differently and results in 1000000. When analyzing

the code carefully it is clear that it adds the results if a scale > 100 is encountered rather than scale the previous value. This decision is unclear if it is intentional and should be documented to avoid misconceptions regarding what output to expect from given input strings.

The 10 most complex functions

parse_date @ textParser.py

This function deals with parsing strings for several formats when asking for a date-object given passed string-argument. You can for example get the date for “next monday” or “2017-01-01 in two year and 3 month”. It is no surprise that a parsing method like this has high complexity since parsing usually have many if cases. The code is not particularly nested, there is only one for loop. The rest of the complexity comes from many if else if cases which is expected for a parsing function. The function is also long(75 lines of code) which is okay for a function which deals with parsing.

Lines of code	CCN	Params	Parent file
75	27	1	jarviscli/utilities/textParser.py

main @ weather_pinpoint.py

This function attempts to pinpoint the users location in order to obtain weather data from some external APIs. When the function is called, location data may already be stored in memory. The user will then be prompted whether this is their actual location, if not, the function will attempt to find the user's location. If this is not successful, the user may input their own location. One of two weather apis are then called, depending on a function argument ‘s’ (terrible name). The code is fairly nested and the function ‘main’ is large (71 lines of code), however it is not very difficult to understand.

Lines of code	CCN	Params	Parent file
70	21	3	jarviscli/packages/weather_pinpoint.py

score @ cricket.py

This function allows you to check for the latest cricket match data from the service cricbuzz. The score-method shows the 3 latest matches and allows you to show information about them. The user is able to show a full scorecard from the match, see the latest commentary or get the latest summarized match score info. So the user has a number of choices. First a number between 1-3 for which match to look at. Then a number between 1-4 to select an action for the match. And then a yes or no if you want to keep refreshing the information from selected action about the match. The function is fairly simple to understand but contains a lot of duplicated code which makes it unnecessary long and more complex than needed.

Lines of code	CCN	Params	Parent file
65	20	1	jarviscli/packages/cricket.py

main @ translate.py

The *translate.py* file only has one function; *main*, whose purpose is to translate arbitrary sentences from one language to another. This function had very little documentation, although most of the content was fairly easy to understand due to descriptive variable names and such, although I feel that some of the code was hard to grasp. After some analysis I found that large parts of the function contained *dead code*, which would execute under no circumstances at all. This only added to my confusion and it took some time before we were actually convinced that the code would never execute (since you wouldn't want to jump to conclusions with such a claim). Translating natural language from one to another seems like a complex task, but most of the real logic is delegated to the *Google Translate API* which makes the functions large CCN value even less motivated. In short, this function really doesn't have to be this complex.

Lines of code	CCN	Params	Parent file
48	16	1	jarviscli/packages/translate.py

parse_number @ textParser.py

The purpose of this function is to parse strings to valid integer. It supports inputs both as numerals with ',' separations and whitespace as well as mixed numerals and literals. Valid literals for parsing is specified in a default dictionary. Returns the parsed integer and amount of words parsed. The code has nested for loops to do the parsing to deal with inputs separated in different ways by splitting them. Most of the complexity comes from cases where it tries to match words in dictionary as well as exceptions and edge cases. The outcomes of the branches consist of if a word is in the dictionary or not as well as if it can parse the numeral directly to an integer or case a ValueError Exception. As well as some additional logic to handle cases of mixing numerals and literals. e.g "24 thousand" => 24000

Lines of code	CCN	Params	Parent file
42	14	2	jarviscli/packages/textParser.py

request_news @ news.py

The request_news function is used to aquire quick news from https://newsapi.org. It takes a url as input parameter and uses this to query for a json message with news articles. The function contains quite a lot of possible exceptions but not that many if/elif statements. it could fail if a url is not passed as an own parameter or passed in the self object. It also includes the possibility to open a browser for the user which implies a lot of things could go wrong.

Lines of code	CCN	Params	Parent file
43	11	2	jarviscli/packages/news.py

organise @ file_organise.py

This function takes a path to a directory that the user wants to "organize" according to given extensions passed. For example if the user wants all java and python files, in the directory to

be organized, in separate subfolders(also passed by user) then the user can pass “java” and “py” as extensions together directory names. The function is a bit complex since it has to deal with directory path finding and exceptions when directories don't exist etc.

Lines of code	CCN	Params	Parent file
21	11	4	jarviscli/packages/file_organise.py

precmd @ Jarvis.py

precmd is a function that is called before every command is executed. It appends certain keywords and checks that the command is correctly formatted. It is not super easy to understand exactly what happens in the function but it has some comments to help the understanding. It is fairly complex as it is mainly made up of if/elif statements.

Lines of code	CCN	Params	Parent file
15	11	2	jarviscli/Jarvis.py

reminder_handler @ reminder.py

The system uses a kind of *“trigger”* mechanism to determine when a reminder (i.e. some kind of todo event) is set to go off and this function is the one that handles those. To just say that the function *handles* these triggers is a very vague description, but it is actually very difficult to understand what is happening in this function; due to a combination of poorly chosen variable names and a lot of nested loops and if-statements, in addition to the lack of documentation. But from what we can gather, this method continuously looks through the queue of active triggers and removes those who has already been activated. There is also some kind of heuristic for determining if a trigger has been activated but this functionality has no documentation at all. Therefore, it is very difficult to determine if this function has a fair amount of complexity.

Lines of code	CCN	Params	Parent file
29	10	1	jarviscli/packages/reminder.py

get_opt @ news.py

The get_opt function gives the user the option to set a specific news provider for the articles given by the news command in jarvis. It is very forward and not that complex to understand at all! There’s a lot of if/elif statements but they are not that nested which decreases the complexity of understanding what is going on.

Lines of code	CCN	Params	Parent file
29	10	1	jarviscli/packages/news.py

Coverage

Output from *coverage.py* (before any aditional added tests):
https://gist.github.com/remnestal/467ea648b76def9c4153e4632fb05b93

Tools

Coverage.py was very simple to use in our opinion. It had a good documentation and did not require any extensive setup to work. It was quite easy to integrate with how the tests were run in Jarvis. We just wrote a simple shell script (can be seen below in figure 2) that ran the unit tests and gave the result to coverage.py.

```
cd ~/Jarvis/jarviscli/
touch tests/test_manual/__init__.py
echo "running coverage run on tests"
coverage run --branch --source=. -m unittest discover -s tests/
echo "reporting coverage"
coverage report -m
```

Figure 2. Shell script to check coverage of tests

DIY

The instrumented code was developed in this branch.
A shell script 'covtest.sh' was used to generate the instrumented measurements and store it in files.

An initiator function was created for each of the ten functions that created an empty set, to not store duplicate information, where branch data is to be stored. This function then made a function call to the real function with this branch-set as an extra parameter. An element was added to the set at every branch in the ten functions with its line number. The set was then written to file as well as the amount of total branches for the function and how many were activated when the function was invoked; for each of the ten functions. These changes can be seen in the ten file references below.

The python script 'diycov.py' was then used to compile all the generated branch coverage results from the generated files after running all tests and give it as readable output which contains the coverage results for each function measured.

Files instrumented for branch-coverage measurement:

- ➔ parse_date@85-217@./utilities/textParser.py
- ➔ main@7-116@./packages/weather_pinpoint.py
- ➔ score@100-202@./packages/cricket.py
- ➔ main@6-85@./packages/translate.py
- ➔ parse_number@7-83@./utilities/textParser.py
- ➔ request_news@153-238@./packages/news.py
- ➔ organise@71-116@./packages/file_organise.py

- ➔ precmd@55-89@./Jarvis.py
- ➔ reminder_handler@154-208@./packages/reminder.py
- ➔ get_opt@90-141@./packages/news.py

The results of the DIY Coverage Tool can be found here. These results were taken from after our tests were added to the functions as there existed no tests for these functions previously and would have resulted in 0% coverage for all these functions which was also observed by running Coverage.py.

Evaluation

Report of old coverage
Report of new coverage

Test cases added:

- Test cases for main@weather_pinpoint.py - code coverage improved 7% → 63%
- Test cases for parse_date@textParser.py - code coverage improved 3% → 76%
- Test cases for score@cricket.py - code coverage improved 5% → 89%
- Test cases for main@translate.py - code coverage improved 8% → 70%
- Test cases for parse_number@textparser.py - code coverage improved 3% → 34% (100% coverage of the parse_number function though)

Total test suite coverage improvement: 37% → 53%

Refactoring

Refactoring of score@cricket.py (#16 on github)

There are some improvements that possibly could be made to decrease the CCN number.There is duplicated code in quite many places:

- At two points the user could input a faulty number to choose from a menu and the same logic is used in both the places. This could be refactored!
- At the same two points the user is faced with a choice dialogue that looks identical but is duplicated in the code. Improvements could be made!
- At three points when looking up different types of data of a match the user is asked if he/she wants to refresh the information. The dialogue options are exactly the same for all three data types and the code probably be refactored to be less complex.

The refactoring was carried out by moving the many duplicated segments of code into own methods and can be seen on the github repo. It resulted in a change of the CCN from 20 to 13 when tested with lizard which means the complexity was reduced by 35%!

Refactoring of main@translate.py (#17 on github)

As previously mentioned, the *main* function in *translate.py* contains a lot of dead if-statements that simply cannot execute under any circumstances. This was the main issue with this function and also the main thing that was changed with the refactor. Another big issue with the code was that if-statements to change between logic for python2 and python3 was duplicated anywhere user input was requested. Since this function asks for the user to input some text many times, the code quickly becomes cluttered with these. This was easy to fix and only required that if-statement to be broken out into its own function within the *main*-function.

Refactoring of main@weather_pinpoint.py (#15 on github)

There are a number of significant improvements that could be made to decrease the CCN.

- To maintain backwards compatibility, the authors check if the program is running on python2, and if that is the case they use raw_input instead of input. This check only needs to be done once.
- The authors have decided to use if-else constructs everywhere instead of a more flat structure with return-statements in the middle of the function. Switching to a more flat structure will improve readability and make the code shorter.
- Some code is regularly repeated, which can be replaced with a function.

These changes mentioned above were carried out, the complexity calculations can be found on the github issue. The changes resulted in a reduction of complexity by 46%.

Refactoring of parse_number@textParser.py (#18 on github)

The refactor that was carried out consisted of moving a big chunk of default initializations to its own method that were contained in if constructs in the original function. This reduced the functions CCN from 14 to 9. The main parsing block consisting of a nested for-loop could probably be refactored or rather remade as well. However, for its current functionality there is no parts of it that can easily be or necessary to refactor.

Refactoring of parse_date@textParser.py (#14 on github)

A refactoring was carried which can be seen in the link to gihub. The complexity measure of parse_date function was decreased from 27 to 15(~45% CCN decrease) by refactoring out 3 parts of the code that were suitable for extracting. Namely the cases for handling 1: date-formats, 2: time-formats and 3:time-spans. It should be noted however that since the function handles parsing it may not be very necessary to refactor it with the point of decreasing complexity in mind since parsing is usually straightforward anyways(there is not much nesting for example).

Effort spent

Anders

Task	Time (h)

Setup project, fix tools and errors	3
Calculating CCNs	4
Writing tests	4
Refactor	2
Github	1
group discussions	2
report writing	4
TOTAL	20

Marcus

Task	**Time (h)**
Project setup	2
CCN/Flowcharts	5
Report	1
Writing tests	1
Refactor	1
Attempting automated coverage analysis	3
TOTAL	13

JOHAN

Task	**Time (h)**
Choosing project + project setup	2
Lizard and coverage learning simple commands	1
Flowchart and CNN	5
Discussion within group	3.5
Github	1.5
Running and testing code	2
Report	2
Writing tests and testing	2
Refactor parse_date	2
Total	21

Albin

Task	Time (h)
Project setup	2
CCN count / flowcharts	4.5
Github administration, issues etc	4
Analysing coverage/lizard results	2
Writing testcases for jarviscli/packages/translate.py	3.5
Calculating CCN by hand for 'organise' and 'reminder_handler'	3
Writing script for analysing result of ad-hoc branch coverage tool	1.5
Refactoring main@translate.py	2
discussions in project group	3
report writing	2
compiling data from lizard, coverage.py and ad-hoc tool	1
TOTAL	28.5

Robert

Task	Time
Setup project/tools/fix errors	2
Use tools/analyze results/	2
Reading docs	1
CCN/FlowChart for 2 functions	3
Improve Coverage (tests)	2
Refactoring	2
coverage tool	5
fixing bugs in code	0.5
report writing	1.5
TOTAL	19

Overall experience

This project was quite interesting overall. It was interesting to see that there are many simple tools available to check things like coverage and complexity. The coverage tools can be a great aid when you want to check how complete the test suite is, but a high coverage does not of course mean that everything works as it should, you still need good oracles for the tests. The coverage tools could potentially also be of great help when analysing the software to find dead code to remove.