INF3121
Project 1: Interpret the metrics offered by the static analyzer

Husein Mehmedagic[*] Even Langfeldt Friberg[†]

March 10, 2015

Introduction

In this project we will make use of one of the authors' first Java code, slightly more than 500 lines, which was given as the first mandatory assignment in the course INF1010 taught in spring 2012. The original assignment text can be found at http://heim.ifi.uio.no/inf1010/v12/oblig/1/oblig1_.html.

The GitHub repository for our work can be found at http://github.com/elfriberg/inf3121/tree/master/project1.

1 Metrics Kiviat Graph on Checkpoint and Significant Other (Step 3)

What does the graph tell you? How do you interpret the metrics applied on your project?

For the metrics applied on the checkpoint (total project) we observe that although comments are sparse, they are still over a minimum defined by the inner boundary. Methods per class says here that this small project follows good object-oriented principles in not overloading a class with too many methods. In this case we probably could have done with fewer classes, some of them are very small. Average statements per method seems to be quite okay, within boundaries. Max complexity is more interesting: it seems to be 18 points off the outer boundary. Note that this is for one particular method. Considering this is the first assignment in an introductory Java subject it is hardly a surprise. The student probably didn't know smarter ways of doing things, and stuck to the most basic of structures, and was happy that the code run at all. Note that even though average complexity is on the higher end, it is considerable closer to the Kiviat graph than the maximum metric. This means (since there are few

[*]huseinm@student.matnat.uio.no.
[†]evenlf@student.matnat.uio.no.

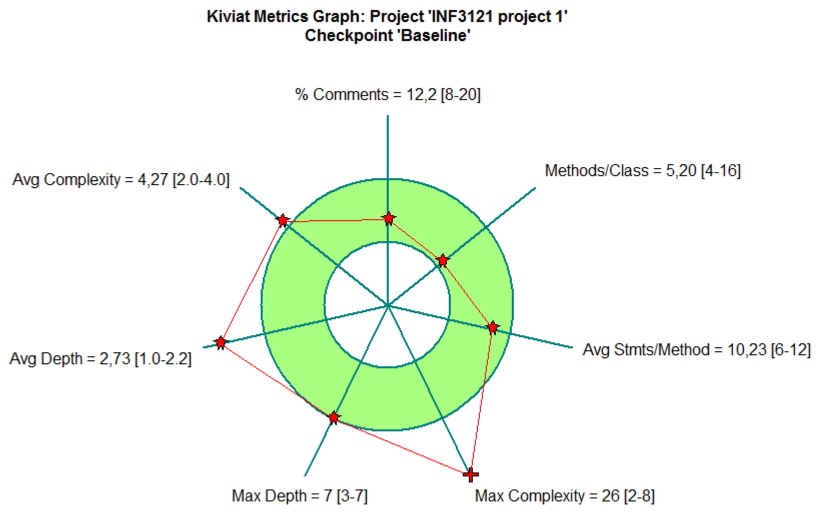


Figure 1: We readily see the complexity and the depth both exceed the boundaries given by the Kiviat graph. Note that this is metrics on all java files in the project.

lines of code) that there is one (or a few methods) that is much more complex than the average.

The depth metric is also outside the outer graph boundary, and it means some methods could have been doing less. If the novice programmer made each method perform just a minimum, but ensured that it did what it was supposed to do after he finished it, it might have been easier to debug during development.

We now turn our attention to the two most significant files of the project (`PersonListe.java`: 197 lines; `Program.java`: 204 lines) and one really small (`Person.java`: 80 lines).

What does the graph tell you? How do you interpret the metrics applied on your file? How are they different the metrics you obtained on the whole project, compared with the metrics on this file?

The Kiviat graph for `Program.java` helps us identify the family's black sheep. This file consists of virtually no comments. If one can read Norwegian this can be excused as the method names make quite a lot of sense. This file contains the code for the terminal user-interface (`public void kommandoProgram`) and commands for e.g. adding a new person (`public void cmdNyperson`) to the system, a super-simple 'Facebook' system.

Max depth and max complexity is the same as for the checkpoint already considered. Methods per class is well inbetween the limits and average state-

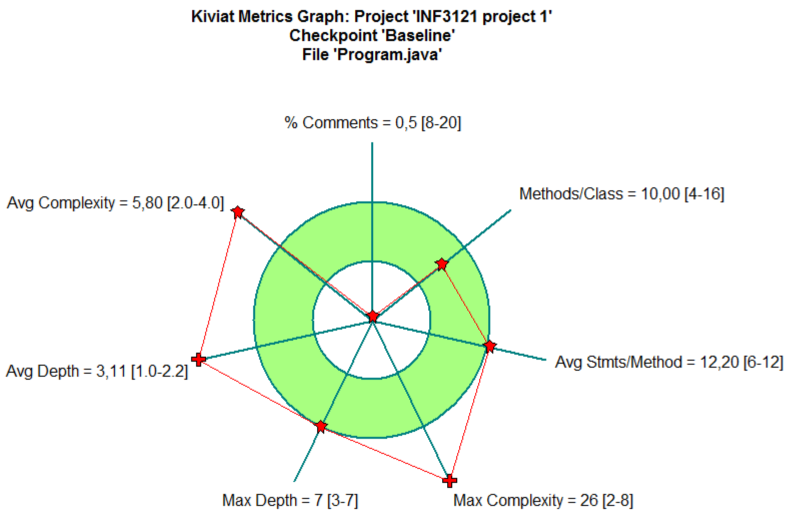


Figure 2: Kiviat graph for the largest file in the project.

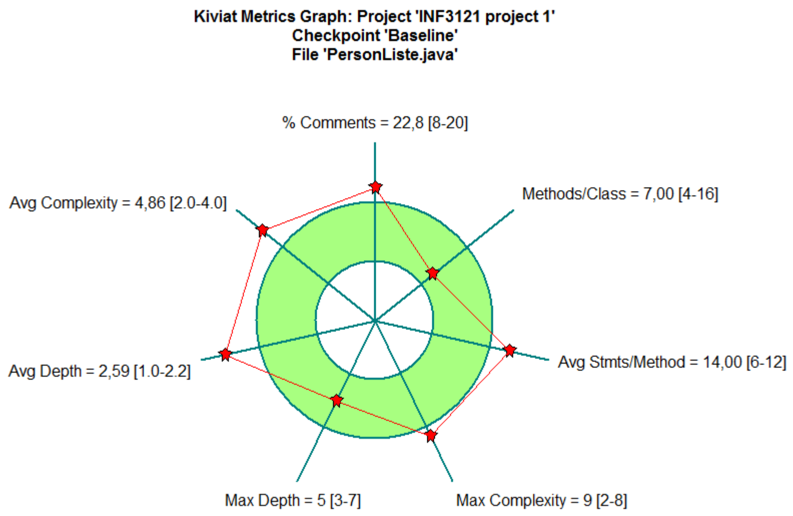


Figure 3: Kiviat graph for the next-largest file in the project.

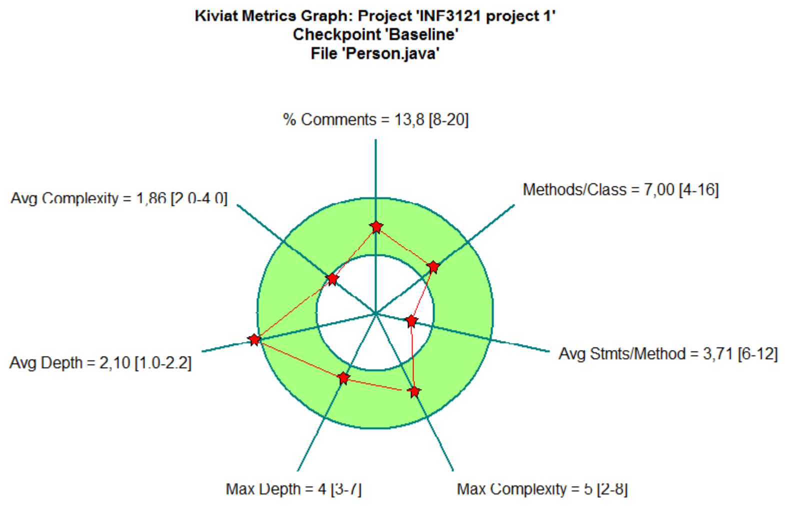


Figure 4: Kiviat graph for a smaller file in the project.

ments per method is just on the upper limit.

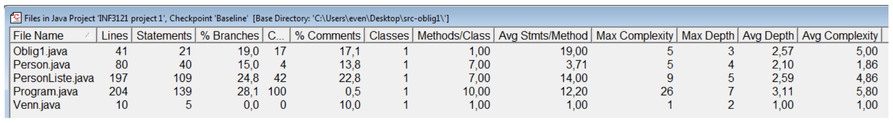
Offset points from checkpoint:

1. Comments: $0.5 - 12.2 = -11.7$
2. Methods/Class: $10 - 5.2 = 4.8$
3. Avg Stmts/Class: $12.2 - 10.23 = 1.97$
4. Max Complexity: $26 - 26 = 0$
5. Max Depth: $7 - 7 = 0$
6. Avg Depth: $3.11 - 2.73 = 0.38$
7. Avg Complexity: $5.8 - 4.27 = 1.53$

2 Quantative Data (Step 4)

What is the biggest file you have in your project? How long the file? How many methods in it?

The largest file is `Program.java` consisting of 204 lines (6.9 KB), and 10 methods, including the constructor method.


Files in Java Project 'INF3121 project 1', Checkpoint 'Baseline' [Base Directory: 'C:\Users\even\Desktop\src-oblig1\']

File Name	Lines	Statements	% Branches	C...	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity
Oblig1.java	41	21	19,0	17	17,1	1	1,00	19,00	5	3	2,57	5,00
Person.java	80	40	15,0	4	13,8	1	7,00	3,71	5	4	2,10	1,86
PersonListe.java	197	109	24,8	42	22,8	1	7,00	14,00	9	5	2,59	4,86
Program.java	204	139	28,1	100	0,5	1	10,00	12,20	26	7	3,11	5,80
Venn.java	10	5	0,0	0	10,0	1	1,00	1,00	1	2	1,00	1,00

Figure 5: Step 4

3 Method Metrics (Step 5)

Which is the most complex method in your project? How many statements does it have? - Would you refactor any of the methods you have in your project? - What can you say about how coupled/decoupled the program is?

1. The most complex method is `kommandoProgram`, with a complexity of 26. It consists of 40 statements.

2. Yes, for sure. `kommandoProgram` should have been refactored, to get rid of many redundant logical statements. This is also true for `fjernVenn` who suffers under the linkedlist-structure.

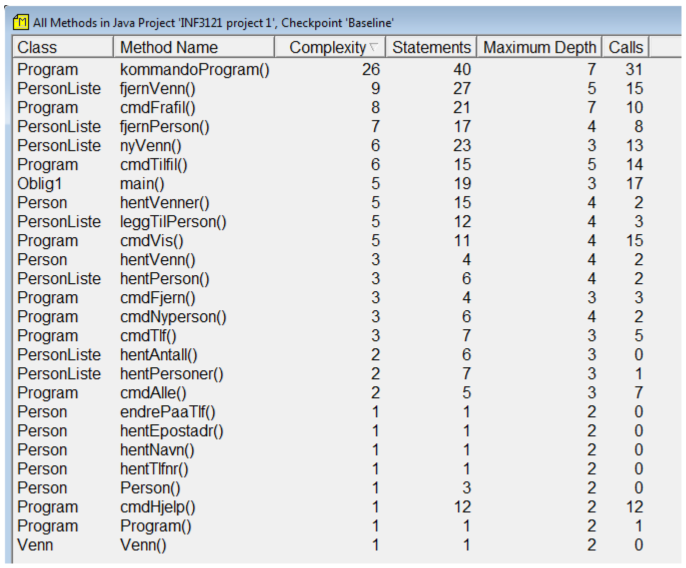
3. We have 8 methods with more than 10 method-calls. The two most highly-coupled methods make respectively 31 and 17 calls. This is quite a lot for so few lines of code. Changing the structure should be done carefully because several methods are that coupled.

4 Suggestions for Improvements (Step 6)

How would you improve your code, based on the metrics you have obtained with this analyzer?

```
(...)

} else if (innKommando.equalsIgnoreCase("tlf") && (words.length > 2)){
cmdTlf(words[1], words[2]);
} else if (innKommando.equalsIgnoreCase("fjern") && (words.length > 1)){
cmdFjern(words[1]);
} else if (innKommando.equalsIgnoreCase("alle")){
cmdAlle();
} else if (innKommando.equalsIgnoreCase("hjelp")){
cmdHjelp();
} else if (innKommando.equalsIgnoreCase("venner") && (words.length > 2)){
personliste.nyVenn(words[1], words[2]);
} else if (innKommando.equalsIgnoreCase("uvenner") && (words.length > 2)){
personliste.fjernVenn(words[1], words[2]);
} else if (innKommando.equalsIgnoreCase("vis") && (words.length > 1)){
cmdVis (words[1]);
```


All Methods in Java Project 'INF3121 project 1', Checkpoint 'Baseline'

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
Program	kommandoProgram()	26	40	7	31
PersonListe	fjernVenn()	9	27	5	15
Program	cmdFrafil()	8	21	7	10
PersonListe	fjernPerson()	7	17	4	8
PersonListe	nyVenn()	6	23	3	13
Program	cmdTilfil()	6	15	5	14
Oblig1	main()	5	19	3	17
Person	hentVenner()	5	15	4	2
PersonListe	leggTilPerson()	5	12	4	3
Program	cmdVis()	5	11	4	15
Person	hentVenn()	3	4	4	2
PersonListe	hentPerson()	3	6	4	2
Program	cmdFjern()	3	4	3	3
Program	cmdNyperson()	3	6	4	2
Program	cmdTlf()	3	7	3	5
PersonListe	hentAntall()	2	6	3	0
PersonListe	hentPersoner()	2	7	3	1
Program	cmdAlle()	2	5	3	7
Person	endrePaaTlf()	1	1	2	0
Person	hentEpostadr()	1	1	2	0
Person	hentNavn()	1	1	2	0
Person	hentTlfnr()	1	1	2	0
Person	Person()	1	3	2	0
Program	cmdHjelp()	1	12	2	12
Program	Program()	1	1	2	1
Venn	Venn()	1	1	2	0

Figure 6: Step 5

```
} else if (innKommando.equalsIgnoreCase("tilfil") && (words.length > 1)){
cmdTilfil(words[1]);
} else if (innKommando.equalsIgnoreCase("frafil") && (words.length > 1)){
cmdFrafil(words[1]);
} else if (innKommando.equalsIgnoreCase("slutt")){
System.out.println("Ha det bra!");
}
else {
System.out.println("Kommandoen du gav er ikke på gyldig form");
cmdHjelp();
}

(...)
```

This is a snippet of the `kommandoProgram` method of `Program.java` with a lot of logical statements. We could put all valid command words (i.e. `hjelp`) in an array, check if the `innKommando` (or simply `words[0]`) is found in the array, and call the relevant command, or give an error message if the user input a typo. Thus we could reduce 10 else-if statements to 1.

```
public boolean fjernVenn(String navn, String vnavn){
(...)
else {
  for (Venn peker = a.foersteVenn; peker.nesteVenn != null; peker = peker.nesteVenn) {
    if (peker.nesteVenn.minVenn.hentNavn().equalsIgnoreCase(enVenn.hentNavn())) {
```

```
peker.nesteVenn = peker.nesteVenn.nesteVenn;
System.out.println("OK: " + vnavn + " er nå fjernet fra vennelisten til " + navn);
a.venneTeller--;
return true;
(...)
```

This is a snippet of method `fjernVenn` from `PersonListe.java`. An objective of this mandatory assignment was to implement a self-written LinkedList-structure. The code above could have been much simpler if one made use of the available LinkedList class from the Java library (which implements a lot of useful interfaces).