Assignment series 2

Thijs Klaver & Carly Hill

Perspective of a maintainer

I could not find the article named, since there were just some writers provided and no title…
What I think software should be like to be maintainable:

- easy to understand
- have meaningful tests
- not be extremely complex

Easy to understand

The software is easy to understand because of the structure. The software is divided in modules which all have their own task, and in the modules the order gives structure to the functions, functions that work the same are close together. Also, documentation explains what the code does. This might not be necessary because the methods speak for themselves, but it is a good way to explain again what and why you are doing something.

Have meaningful tests

Testing the software is of course very important. It makes sure the software works according to its specification. Apart from some unit test methods (you do need to call them **after** doing main) the testing is done by a test class. Because the software isn't easy to completely test in unit tests, the main test method also gives you the visualization and the generation of the report. This way the user can also check the output of the software.

Not be extremely complex

The software will be complex in Cyclomatic Complexity, but as Carly has argued in her position paper, case statements are psychologically not very complex. The FileSplitter has a lot of case statements, but they are easy to understand. By breaking up the functions the difficult functionality in separate methods, the software is not so complex.

The exact type of clones the software finds

The software was supposed to find type 1 and type 2 clones. However, the clone detection doesn't work correctly, probably because the hashing isn't done correctly. Apart from finding actual clones, it finds clones that are not actually clones. So if we should improve this code I would redesign the hashing algorithm. Specifically the part where sequential statements are being hashed together.

core of clone detecting algorithm (pseudocode)

The clone detecting algorithm works as follows:

First get a few values we need to calculate things.

```
getValues(loc projectLocation, int dupType){
	M3 model = createM3FromEclipseProject(projectLocation);
	<LOC, LOCbyFile> = getLoc(model);
	map[loc, Declaration] ASTsbyFile = getAsts(model, dupType);
	return <LOC, LOCbyFile, ASTsbyFile>;
}
```

With these values we can detect clones the following way by hashing.

```
getHash(value ASTnode, acc){
	foreach child <- ASTnode
		calculateHash(child);
	foreach(sublists of children){
		calculateHash(sublist);
		acc += (hash : ASTnode@src + res);
	}
	calculateHash(ASTnode);
	if(hash already exists)
		acc += (hash : ASTnode@src + rest);
	else
		acc += (hash : ASTnode@src);
	return <hash, acc>;
}
```

After the hashing we filter every list of locations for duplicates and delete any hash that only has one element after that.

```
for(i <- acc){
	list[loc] l = dup(acc[i]);
	if(size(l) > 1)
		acc += (i : l);
	else
		acc = delete(acc, i);
}
```

After this, range(acc) contains all lists of locations with clones.

Visualizations implemented

We have implemented a tree view of the source code, because this will give an easy overview of the files in an understandable way. The files underneath a folder are, for viewer purposes, visualized as a list. This because the tree would get too wide for big programs. The files will get a color from green to red based on the percentage of duplicated lines. Green and red because these are the colors people naturally see as ‘good’ and ‘bad’. It is important to take a look at the files that have a lot of duplication. Furthermore, the lines of duplication are revealed on hover for the files, and the locations of the duplications are printed on click. Because they are printed in the terminal, you can move to the location of the duplication easily.