STATIC TEST TECHNIQUES EXERCISES (10 Study Points)

1. Recap book chapter 1-2 (1 SP – 55% must be passed)

Oversigt over dine besvarelser

Besvarelse	Tilstand	Point / 10,00	Karakter / 1,00	Gennemse
1	Færdig Afleveret lørdag, 17. februar 2018, 23:19	8,50	0,85	Gennemse

2. Static Code Analysis of Triangle Program (4 SP)

GitHub repository Link: https://github.com/cph-cs241/TEST_GetStartedExercise_Triangles.git

I used node module package jsmeter as a tool to generate code metrics for JavaScript.

OLD IMPLEMENTATION:

```
cherr@CJS-LenovoLaptop MINGW64 ~/Desktop/SoftDev_1st/TEST/GetStartedExercise (master)
$ node runjsmeter.js index.js
---------------------------
FUNCTION: typeOfTriangle
CYCLOMATIC COMPLEXITY: 13
MI: 106.01
Depth: 1
Lines: 15
---------------------------
FUNCTION: check
CYCLOMATIC COMPLEXITY: 7
MI: 104.96
Depth: 1
Lines: 18
---------------------------
```

NEW IMPLEMENTATION:

```
cherr@CJS-LenovoLaptop MINGW64 ~/Desktop/SoftDev_1st/TEST/GetStartedExercise (master)
$ node runjsmeter.js index.js
---------------------------
FUNCTION: typeOfTriangle
NODES: 6
EDGES: 9
CYCLOMATIC COMPLEXITY: 8
MI: 110.28
Depth: 1
Lines: 13
---------------------------
FUNCTION: check
NODES: 5
EDGES: 7
CYCLOMATIC COMPLEXITY: 6
MI: 110.07
Depth: 1
Lines: 14
---------------------------
FUNCTION: isValidSideLength
NODES: 5
EDGES: 7
CYCLOMATIC COMPLEXITY: 6
MI: 128.40
Depth: 1
Lines: 5
---------------------------
FUNCTION: getTriangle
NODES: 7
EDGES: 11
CYCLOMATIC COMPLEXITY: 9
MI: 121.59
Depth: 1
Lines: 7
---------------------------
```

Solution:

In order to reduce CC, I created new functions that would separate the logic of getting the result of the type of the triangle.

As a solution to the previous assignment last week (flattened triangle – invalid 1, 2, 3), I created a new function to check whether the side lengths are valid to form a correct triangle.

3. Peer Review Checklist (2 SP)

Smartbear: Best practices for Code Review

Tip #6: Use checklist in the review process

1. ***Set Early Expectations With The Developer About Annotating Their Source Code Before The Review***
   - The developer would have the chance to go through his code thoroughly and within this practice, the chances of finding defects is high before conducting the review itself. The pieces of codes was written or inserted when we are developing a requirement and we are seeing the need of it, I know that it was there but I might not be aware if it will be affecting the existing ones. And that is when, carefully reading it through would be useful.
2. ***Determine Quantifiable Goals Prior To Code Review For Increased Accountability***
   - Every team members would have the sense of responsibility by setting up realistic goals. Improvement is more visible as they could define such tasks done.
3. ***Have A System To Capture Metrics So You Can Improve Your Processes***
   - The metrics will show the complexity of the code, which means it would be easier to see the need of refactoring the code to make it more effective and efficient.
4. ***Plan Enough Time For A Proper, Slow Review But Not More Than 60-90 Minutes***
5. ***Review No More Than 200 Lines Of Code In One Sitting***
   - Based on studies, the accuracy of finding the defects within this limit is more effective
6. ***Take A 20 Minute Break In Between Review Sessions***

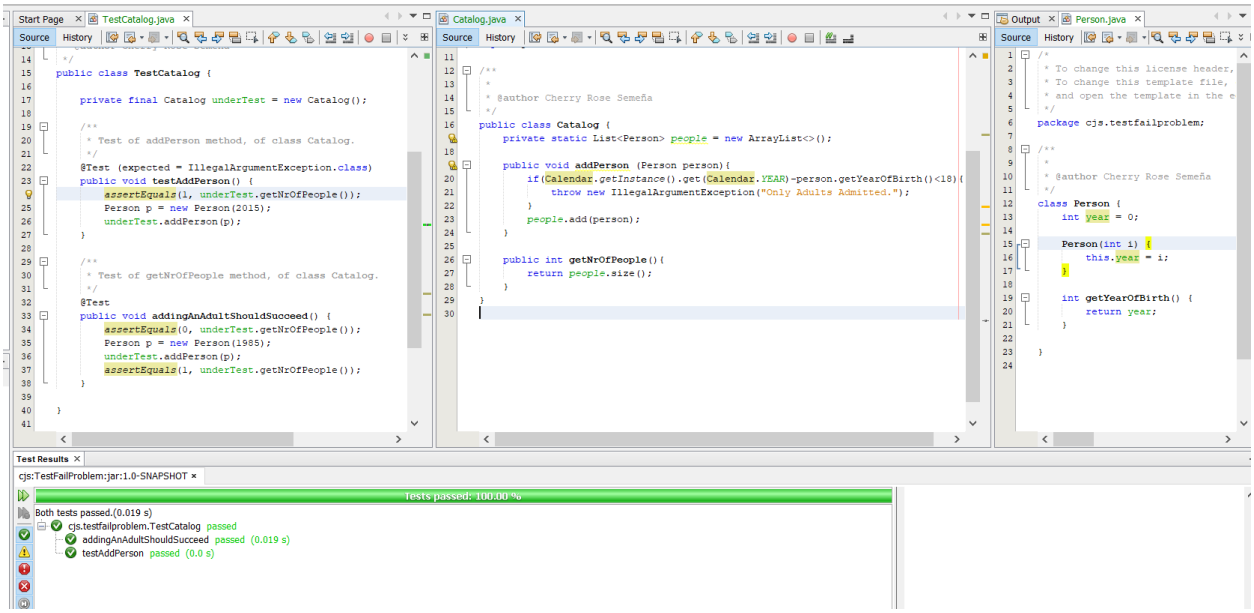
   - The checklist from 4-6 is fairly important, tiredness may lead to unproductive job. It is common way to keep yourself fresh and effective, in order to make things done with favorable results.

7. ***Verify That Defects Are Actually Fixed, Not Just Found***
   - Making a follow-up for the hanging fruits, meaning once the defect has been defined, it has to be fixed or else it doesn't make sense at all, leaving an open wound.
8. ***Use Code Review As A Team Building Activity. Remember, Finding Defects Before Launch Is The Point!***
   - Taking the coding review as a learning activity. Defects may occur because of the bad practices/routines that can be improved. Just like in programming, as we encounter errors

and fix it, we are also learning a strategy to prevent such error again. It is an open opportunity of learning.

9. ***Don't Be A Back Seat Coder***
    - You are in a team, and every member has their own responsibility. As an example, in pair programming, there is a driver and a navigator. It would only be a effective if two heads are working and not just constantly watching.
10. ***Do A Certain Amount Of Code Review Each Day, Even If It Is Not Every Line***
    - Continuously reviewing the code could help to get the developer attached to their own code. Finding some possible defects may also occur during this process.
11. ***Use A Code Review Tool For Even Greater Efficiency And Accuracy***
    - To gain efficiency and accuracy for code review, using a tool is recommended.
12. ***Use Checklists To Substantially Improve Results For Both Authors And Reviewers***
    - Being organized in order to prevent missing necessary things during code review. The checklist would act as a reminder on what has to be done/reviewed.

4. Review code that mysteriously fails its unit tests (1 SP)



Problem:

The ordering of test-method invocations is not guaranteed. The *addingAnAdultShouldSucceed()* was first executed, so the list of number of people wasn't '0' when the *testAddPerson()* was invoked.

Tests shouldn't be order dependent. I think that every test cases could be tested independently, so that it will make things easy to maintain and add more tests without affecting one another.

5. Coding Standard Document (1 SP)

We have been taught for some coding standards in the previous education, and that enables us to deal with it naturally. The more you put it in practice when coding, the more it will become your habit to use it.

The coding standards which I use most of the time, and I think is important for a team to follow are:

1. *Proper Naming*
    - The code must speak for itself. In that way, understanding the code wouldn't be such a pain especially when it has to be shared with the other team members or for maintainability purposes in the future. Proper naming convention includes the function name that states what it does and such variables that says what is it for, likewise with the modules and packages. Declarations and imports comes first before functions, and would only differ for locally declared inside functions.
2. *Indentations and break lines*
    - It will become more readable if indentations will be used to easily see the scope of the pieces of codes, whether it is just part of a function or if you are missing some parenthesis/ brackets. It would also define the scope by making new line after curly brackets and understand the logic of your code easily.
3. *Modular*
    - It is definitely a good practice to divide the source code into some modules that has its own responsibilities. For example, if they are using object-oriented programming, it is important to have the models separately from the controllers or database operators. This could be done by considering the design decisions or architectural layers.
4. *Camel Case or use of underscores*
    - Sometimes this may differ from the programming language you are using, or if it is a name of a file or for naming the functions or variables. I've been using small letters for naming packages, with underscores for files and camel case for function names and attributes. But, I also have seen and reviewed somebody else's code and they use different style. The good thing is both would give a common understanding when it comes to reading the code itself.
5. *Comments*
    - Leaving some comments/documentation is important for further development. It is much appreciated on more complex parts of the code.

6. Highlights from lecture by Gitte Ottosen, Gapgemini-Sogeti (1 SP)

Gitte Ottosen has provided a better understanding on the **importance of testing**, and I think it is very significant for us, students/developers, to invest time and prioritize testing on every project that we are doing wherein we often failed to do. The satisfaction of just seeing our code working the first time using the same test input all the time that we are executing it is obviously wrong attitude. She said something like, "The cost of defects found in production is a high risk because of no review… it is the developers responsibility to provide quality assurance". She explained how expensive it could be if there is no such

as **early testing** on the project, the later you found the defects, the more complicated and expensive it would be in order to be fixed. Sufficient test delivers known quality.

Another essential thing that she discussed is the **Agile Manifesto**. She made a good clarification why it uses "over" in defining the agile perspective. It means that "*<more value>* over *<also needed>*" such as "Individual & Interactions over Processes & Tools", and so on. She also mentioned the **challenges** that most of us are encountering in the development process such as test becoming bottleneck which means a team not taking responsibility, non-functional issues, common ownership, insufficient technical knowledge, system integration testing, too little focus on "done" which is sometimes becomes "done-done-done", role of test and test manager, test automation, and empowerment. In a cross-functional team, all skills are needed. The are some cases wherein the development team focuses on one user story like branching functionalities, and then merging it all once done, went through the test phase and nothing worked! She gave us ideas for testing strategies such as setting up points/boundaries when making test cases which I find it really useful and interesting, illustrated in pairwise example. It is not necessary to put all possible inputs when generating test cases, just having before, after and the boundary itself is enough since defects often occurs in data pairs.

Lastly, all the quarters in the **Agile Testing Quadrants** must be addressed. Another example she used is the apps development. Apps would be a success if it will provide good performance, security and usability. In consideration with the agile testing matrix/quadrants as a pattern, and reaching the UAT phase of the testing process would make sure that it can handle the expectations of the users.