LINK TO GITHUB CODE:

https://github.com/kasperpagh/testClassAssignmentTwo

2) Static Code Analysis of Triangle Application

A) Since I use IntelliJ i decided to go with the plug-in called: MetricsReloaded[1].
B) I used standards in the way i think suited the project - the main points being:

- Validation returns booleans
- All variables are declared as private
- No null returns
- Exceptions thrown until they "can't go any further" (main)

C)

Metrics Complexity metrics for Project 'TriangleTestAssignment' from to, 15 feb. 2018 21:15:34 CET

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

method	ev(G)	iv(G)	v(G)
cphbusiness.pagh.IllegalTriangleException.IllegalTriangleException(String)	1	1	1
cphbusiness.pagh.InputValidator.angleSumChecker(int,int,int)	2	1	2
cphbusiness.pagh.InputValidator.inequalityChecker(int,int,int)	2	1	4
cphbusiness.pagh.InputValidator.naturalNumberChecker(int,int,int,int,int,int)	2	1	7
cphbusiness.pagh.Main.main(String[])	1	2	2
cphbusiness.pagh.Triangle.getTriangleType()	3	2	3
cphbusiness.pagh.Triangle.Triangle(int,int,int,int,int,int)	4	2	4
cphbusiness.pagh.TriangleTypeChecker.equilateralCheck(int,int,int,int,int,int)	2	1	5
cphbusiness.pagh.TriangleTypeChecker.isoscelesCheck(int,int,int,int,int,int)	2	1	7
test.TriangleTesterTest.angleValidationTester()	1	1	1
test.TriangleTesterTest.constructorTester()	1	1	1
test.TriangleTesterTest.constructorTesterWrong0()	1	1	1
test.TriangleTesterTest.constructorTesterWrong1()	1	1	1
test.TriangleTesterTest.constructorTesterWrong2()	1	1	1
test.TriangleTesterTest.constructorTesterWrong3()	1	1	1
test.TriangleTesterTest.constructorTesterWrong4()	1	1	1
test.TriangleTesterTest.constructorTesterWrong5()	1	1	1
test.TriangleTesterTest.inequalityValidationTester()	1	1	1
test.TriangleTesterTest.naturalNumberValidationTester()	1	1	1
Total	**29**	**22**	**45**
Average	1,53	1,16	2,37

Screendump of metricsReloaded's complexity metric.
Contains three different CC's for all methods, with warnings in red.

D) As you can properly guess from the above screenshot, metricsReloaded is kind enough to provide me with all three kinds of CC metrics by default.
In regards to the terminology of your slides they match up in the following way (I think, i'm actually not totally sure):

CC	ev(G) Essential Complexity

[1] https://plugins.jetbrains.com/plugin/93-metricsreloaded

CC2	v(G) Ordinary Cyclical Complexity
CC3	iv(G) Module Design Complexity Metric

E) The CC metrics you see above (in C) is the result of quite a bit of refactoring, namely i decided to put all my input validation in its own separate methods, since they really messed with the complexity:

```
3   public class InputValidator
4   {
5   @     public static boolean naturalNumberChecker(int a, int b, int c, int d, int e, int f)
6         {
7             if (a > 0 && b > 0 && c > 0 && d > 0 && e > 0 && f > 0)
8             {
9                 return true;
10            }
11            else
12            {
13                return false;
14            }
15        }
16
17  @     public static boolean angleSumChecker(int d, int e, int f)
18        {...}
28
29  @     public static boolean inequalityChecker(int a, int b, int c)
30        {...}
40
41
42  }
43
```

*Screenshot of my InputValidator class, in the first iteration of the program all these checks lived inside the same **if-statement.***
*(Funny note; at one point i accidentally wrote the above **naturalNumberChecker**, in such a way that **just one** of the inputs had to be above 0 for it to return true - I guess that goes to show the need for testing).*

```
if (!InputValidator.angleSumChecker(angleA, angleB, angleC))
{
    throw new IllegalTriangleException("Vinkelsum er ikke 180");
}
if (!InputValidator.naturalNumberChecker(sideA, sideB, sideC, angleA, angleB, angleC))
{
    throw new IllegalTriangleException("En vinkel eller side er 0 - det må den ikke!");
}
if (!InputValidator.inequalityChecker(sideA, sideB, sideC))
{
    throw new IllegalTriangleException("Siderne overholder ikke Triangle Inequality Theorem");
}
else
{
    System.out.println("Validation passed, constructing Triangle!!");
```

The use of the InputValidator, in an earlier version, all of the above was expressed as one single statement.

I also made another quite similar change in regards to the code that checks what type of triangle is provided. Like in the above example i had a couple of “chained” if-statements with all the boolean logic - which i divided up into two methods that returns either true or false, such that the following became possible:

```
public String getTriangleType()
{
    if (TriangleTypeChecker.equilateralCheck(sideA, sideB, sideC, angleA, angleB, angleC)) //ens sider og ens vinkler gør en equilateral trekant
    {
        return "equilateral";
    }
    else if (TriangleTypeChecker.isoscelesCheck(sideA, sideB, sideC, angleA, angleB, angleC))
    {
        return "isosceles";
    }
    else
    {
        //a != b && b != c && c != a Er altid sandt, hvis de ovenstående ikke er sande!
        //Det skyldes at der kun er tre typer trekanter!
        return "scalene triangle";
    }
}
```

3) Smartbear Checklist

I want to start off with the following quote from Smartbears webpage:

“It´s very likely that each person on your team makes the same 10 mistakes over and over. Omissions in particular are the hardest defects to find because it´s difficult to review something that isn´t there. Checklists are the most effective way to eliminate frequently made errors and to combat the challenges of omission finding.”
-Smartbear,
https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/

Since i think the above text paints a pretty good picture of the overall idea behind checklists I’ll add my own words in the form of examples of checklists that could be a good idea for a typical Java project. These are of course based on my own ideas for what could be useful (especially in regards the the above mentioned *errors of omission*)

Ex. 1 - A checklist for error handling

- Have you attached a message to your exception/error, explaining the cause and root?
- Have you logged the stack trace leading up to the error?
- Have you either thrown the error/caught the error with respect to company standard/policies?
- Have you determined how and why said error can occur (maybe if you fix the “root cause” you won’t have to bother with error handling?

Ex. 2 - A checklist for return values

- Have you ensured that none of you functions return null?

- Have you ensured that the range of possible return values are explicitly stated in documentation/comments, such that your code can be seamlessly integrated into the project as a whole?

Ex. 3 - A checklist for comments

- Are all your methods commented using the standard JavaDoc format, such that documentation can be auto generated (to save man hours)?
- Do you make sure to write comments in such a way that you explain **why** your code is the way it is, instead just explaining **how** it does it? (it must be assumed that everyone who reads your code understands Java, they may however not understand what's going on in your mind - making the "**why" much more important than the "how"!**)
- Did you remember to add your signature/name to the comment, such that your co-workers can establish who wrote the code, if they need any help?

EX. 4 - A checklist for "meta considerations"

- Did you remember to time your work, to make future time estimates more reliable?
- Did you remember to generate all the, by the organization required, artifacts to go with the code? (EG. diagrams, documentation etc.).

You could of course go on and on, but the point remains, that all of the above points are way easier (and thus cheaper) to do when you are writing the code in the first place, instead of when you are doing the code review.

Furthermore many of the above bullet points are of a nature where they don't break the code if you leave them out - meaning they can be hard to spot later, since they only really come to light when someone has to actually use the code you wrote.

Besides this the use of checklists simply means that you and your team of code reviewers have to spend less time making the review, which in turn means that the company saves money.

4) Review of failing unittest

This was a tough one, actually so though that I had to go and type the code into my own machine to figure out the problem.

What i found was that the following line of code seems to be the problem

```
assertEquals(0, underTest.getNrOfPeople());
```

This would be the one found in the *addingAMinorShouldThrowException* unittest (the topmost of the unit tests). If you remove that line the tests execute flawlessly (jUnit 4). Otherwise the LOC give an *java.lang.AssertionError* since the list has been populated by the adult from the *addingAnAdultSouldSucceed* method (even though it shouldn't have executed yet).

Furthermore from a theoretical standpoint, the assertion is useless since the purpose of the test is not to check the size of anything, but to ensure that the correct error is thrown in a given case.

Now for the reason this error happens, i really don't know. We talked about it in our group and decided that it looked like the test execute in an asynchronous fashion - this, however, is a odds with the jUnit 4 documentation that states all unit tests are executed in a sequential fashion (from the top down, so to speak). My best guess is that jUnit somehow delays the execution of the *@Test(expected = whatever.class)* until the very last moment, while it handles runs @Test (without the expected) first.

Why it does this (of if it's even true) is beyond me, possibly to prevent exceptions from messing with the work of the testing thread - I don't know.

To keep the tests as is, and still fix the problem - i would suggest making use of the *before* functions to ensure that all unit tests execute with a new instance of the Catalog class.

5) Coding Standards

The coding standards that's most important to a given team depends, in my opinion, heavily on the kind of work that's being done, what tools are being used and what language or languages they write in.

That being said there still is a couple of general points that's shared between many fields of software engineering.

- **Syntax formatting**: This is becoming less and less important, since most IDEs ship with tools to instantly format you code in a given way. However there is still value to be gained, by streamlining the formatting standard such that, for example all code pushed to the company's version control system (eg. Git) follows a given format. This will means that the developers can then use their own formatting for local work, and simply run it through some script to revert to a given format when they push. It's easy and everyone can view the code as they like.

- **Exception handling:** This is a big one, as there has to be a unanimous agreement between all the parts of the development team, to prevent any screw ups. For example you could imagine the following two ways of doing it:
    - Throw the exception as far as it can go (for backend that may mean that only the API people has to deal with error handling, since the API is often the “end” of a given process).
    - Always handle exceptions wherever they happen, in the case of Java that might mean that you always have to make a try catch when an exception can occur.

  While the above methods may be a little “over the top”, one thing is clear - if you're development team were to employ both methods at the same time it would be as mess to figure out what you had to do with your own exceptions, it would also be difficult to integrate your own code into the rest of the project, as you can’t know ahead of time how the rest of the code will behave in case of an error.
- **Error content**: The content/data of errors must conform to a certain specification. This could be that it must always log the stack trace, always contain a error message suitable for users or a number of other different things.
- **Return policies**: This one is important since it heavily impacts what code you have to write, for example; Do you have to null-check the return value of all methods? Or do you have a policy of returning empty objects instead (ie. forbid null returns) etc.
- **Version:** It’s very important for the team to make sure that all code fits the same compiler version. For example we once had to make a project, where the solution had to be hosted on OpenShift. However for some obscure reason OS only supported Java 6 or lower (this was well after Java 7 was released), this meant that a number of functions simply weren't available to use in the code (such as lambdas etc). Imagine if one of us wasn't aware of this, we would possibly have to discard/rewrite all of that person's code before we could deliver the assignment.

6) Highlights of Gitte’s presentation

In my opinion the single most important point Gitte made, was the one about how the monetary cost of rectifying bugs, increases by a factor of ten for every “stage” the bug is allowed to exist. This really drives home the point, that it’s much cheaper to *prevent” bugs* than to *fix bugs* (It might be obvious this is good, but not HOW good it is - so to speak).

I also quite liked her spiel about non functional requirements and their importance, since I know from myself that the pursuit of “cool code” sometimes overtakes the pursuit of the actual functionality. Which in turn can lead to subpar product in the end.

The point about being “T-shaped”, that’s to say having a deep technical root, but also branching out into other areas of knowledge was also really good.

This is because you often hear companies need specialists, but it seems (at least according to Gitte) that being a specialist doesn't mean that you should focus on just one point to the exclusion of everything else (which may be a trap people could fall into).