**Static Test Techniques Exercise (5 Study points)**

* ***Static Code Analysis of Triangle program***
* Install Metrics software in your IDE (see tool examples in slides)

**Done.**

* Check coding standards in your Triangle program

**Only warning given is that it recommends for a [STAThread] attribute to be added to the Main() method.**

* Calculate central metrics in your Triangle program



<https://gyazo.com/8a694ee51d2759af74a315a0a8dc73a6>

* Find out what CC variation that your metrics tool uses

**CC1, as it does not increase on booleans, but increases by one for every switch case.**

* Possibly refactor your code based on static testing results
* ***Peer Review of your Triangle program***
* Exchange your Triangle solution with another student

**Switched with Emil Rosenius Pedersen.**

* Inspect the other student’s implementation and write down your comments

**The program has no actual user input, instead opting for just automatic input of integers into the triangle calculation method, which it does thrice, once for each triangle type. The checking of the sides is done without switch cases or if/else, instead opting for a more performance-centric early return model of checking, sectioning every type of side check into its own method method, meaning this is likely a later revision with cyclomatic complexity in mind.**

**The code does check for things such as whether the data input is higher than 0, and in the isosceles calculation does check whether or not the third side is too large to make the triangle real.**

**The code does not check whether the input is truly an integer or not, meaning there's no checking for if the input is a string, a double, or anything other than an integer. It also does not have checks for numbers that are too large to be utilized.**

* Hand over feedback result to each other

**Okay.**

* Possibly refactor your code based on review results
* ***Coding Standard Document***
* Create a coding standard document that describes the best practices and code conventions that you find most important for a team to follow.

**Naming Conventions**

Decide upon a naming convention and stick with it. Decide what sort of casing to use, and the acceptable length and complexity of a name. Make variable and function names as descriptive as possible within reason, such that it's easier to follow the intent and purpose of variables.

**Layout Conventions**

Decide on a shared indent setting, most likely just the default four spaces provided by the Tab key.

Prevent single lines from getting too long. If method chaining is taken into use, make a line break for every dot notation in the chain. This is important for readability. When making a line break, make sure that the break is indented once from the start of the original line.

Add whitespace between method definitions, and possibly after and before curly brackets, to improve readability and making the code seem less dense.

**Commenting Conventions**

Add the comment on its own line, preferably on top of the code it describes.

Write comment in such a way that it is understandable not just to yourself. Make it descriptive, explain the purpose of the code. The purpose of a comment is to explain code to someone else.

Write descriptive unit tests. Tests as documentation go a long way.

Add comments for the method, such as a javadoc, such that other developers can utilize the method more easily. Describe what it does, what it returns, and which parameters it receives. This is especially important for loosely-typed languages such as JavaScript.