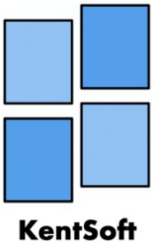
Developer Guidelines



Here at KentSoft we have specific developer guidelines that we expect all our developers to adhere to. We have implemented these procedures to maximise efficiency, code readability and general correctness within our development process.

Developer Environment

All developers can use any environment they want to suit their needs. We would recommend all developers stay away from Docker as this has proven to create issues in the past, however any use of VM's is perfectly fine. Since we are a small company we like to leave these decisions up to developer preference. We do however expect everyone to be working on their own git branch and committing to their branch only.

Programming Language

The programming Language of choice we use at KentSoft is heavily impacted by the Client Requirements. As a team we have decided to use Java for our first client (Yuconz), this is because the client requires a software and because of Java's versatility of working on all operating systems, we think this would best suit their needs.

IDE's

Since we are programming in Java, we have decided to all work with Eclipse. There are a couple of reasons for using this IDE instead of Intellij. However, the main reason is the fact that building a GUI is a lot easier in Eclipse, and we need this as we don't have any inhouse designers and this would make the product look a lot better for our client. Our team of developers had initially intended to use Intellij. However, during development they quickly noticed that it was not satisfying their needs due to the GUI creation system not being as user-friendly in their opinion.

Programming Paradigm

We will be using Object-Oriented Programming. This means that we will be using concepts like Inheritance, Encapsulation, Polymorphism and Abstraction a lot.

Design Pattern

The chosen Design pattern will be TDD. This stands for Test-Driven Development. We will be using this, because it we want to provide a bug-free and usable software to our client and this agile development pattern is something that has been proven to work within the industry. Be prepared for a lot of specific JUnits to be written!

For those of you that are new to this, this is the process.

1. Start by writing a test
2. Run the test and any other tests. At this point, your newly added test should fail. If it doesn't fail here, it might not be testing the right thing and thus has a bug in it.
3. Write the minimum amount of code required to make the test pass
4. Run the tests to check the new test passes
5. Optionally refactor your code
6. Repeat from 1

We have actually switched over to use BDD from Stage 4 onwards. This is because we realised most of our tests require a Test Database as we don't want to compromise the actual database when running the tests. This has slowed us down in terms of manually having to Test the System, however it has allowed us to be more thorough.

Automated Testing & Integration

Using GitLabs CI CD Pipelines, and their generated Docker Images, we are able to continuously integrate functionality into the main repository. Everyone a push or merge occurs, the pipeline is run. The Docker Image tries to build our application on the first stage, and then test it on the second stage. This is done using Gradle, because of this we have had to wrap our Original Java project, and essentially transition it into a Gradle Project. However it was worth doing this, because of the increase in efficiency with automated Testing. This workflow seamlessly coincides with our Software Development Lifecycle's Waterfall-Agile Hybrid.

Coding Style

Class Names

Class Names will be written like this:

```
public class ShowMeThisClass
```

Method Names

Method Names should be written like this:

```
public void thisMethod() {

}
```

Or a longer name like this:

```
public void thisMethodIsRight() {

}
```

Indentation

We will be using a 4-space tab, and we will not use any spaces.
Indentation should be structured like this:

```
public void thisMethodIsRight(){
    if(true){

    }else{

    }
}
```

Not like this:

```
public void thisMethodIsRight()
{
    if(true)
    {

    }
    else
    {

    }
}
```

Error Handling

We should use “Try and Catch” in order to ensure Error Handling along with usability via the pre-built tests we have created by Use Cases, etc.

Variable Declaration

Local Variables should be declared only when needed and should be declared at the top of the Method only if they are important variables. Temporary, Holder and Counter variables can be declared near where they are used instead.

There should only be one variable per declaration.
Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

Comment Style

We will use method comments like this:

```
/**
 * Constructor for objects of class showMe
 */
public ShowMeThisClass(){
    // initialise instance variables
    x = 0;
}
```

These should be well written and detailed. The only other thing we should comment are the local variables declared at the top of the method or class variables declared at the top of classes. These should be done with inline comments like shown below:

```
private int userNum; //Number of all users
```

Variable Names

These should be short and meaningful and should leave a detailed explanation for the variable with the inline comment instead. Like the image shown above.