## High level refactoring strategy

When we are trying to refactor an existing code the more important part is to assure that the application keeps same behavior as before our changes, just with the desired modifications and changes we decide to introduce.

**Before**

1. Analyze behavior of current code. Write all requirements we are fulfilling and the behavior it has in every situation.
2. Document the new features that we would like to add in our system. Detect conflicts with the current behavior defined in step 1 and mark the previous steps as will be deprecated soon.
3. Create a set of black box tests or integration tests (depending of the technical possibilities to cover all documented behaviors described in step 1).
4. Create a continuous integration strategy that runs all this set of tests with the higher frequency possible.
5. Create a different branch of code with existing application. Test set shall also be branched and being executed against this new branch.
6. Identify the points of legacy code where the communication will be done with the new component

**During**

1. Add new tests to check new requirements. Let others as before.
2. Using LegacyWrapper pattern, create a class in old code where focus all calls from legacy that should be changed. By the moment will be targeting to existing functions. Original set of tests must be continuing working after this step.
3. Implement new component. Cover all new code with IT’s and UT’s.
4. When all tests are working as expected, modify Legacywrapper class to target to the interfaces defined in new component. Execute all tests you have.
5. Correct in new code the issues detected in the test execution.

**After**

1. Document all new implementation.
2. Perform the merge of the branches including the modified tests
3. Execute again UT / IT and the blackbox tests to assure merge operation was successful.

## Proposed project structure

My proposed TestManager module project can be split in two big areas: Test and code. Test will be located under a folder called source packages, and test under Test packages. It also will be a two extra main packages containing the libraries.

New TestManager module shall be split in different packages depending of the type of the entities. In our dummy implementation these packages are called test and operations. Each one of these packages contains the abstract entity with common properties/operations for each, then the implementation in different classes of each one of the inherited elements, and finally a factory.

With this structure we can easily add new test types or operations types just adding a new class that inherits from the existing one and adding the operation.

To manage and store the tests and operations while are not used I created a class called TestManager core. Core now contains a simple array where test are stored but in a more realistic implementation shall store this information in a persistence layer.

For test area, the files structure must be a replica of the file structure defined in the source code area. Each one of the classes must be covered by a unit test class called with same name as the class it is covering, just adding word “Test” on the end, and in a same folder structure. In this example I developed the unit tests using the Junit technology, but any else can be valid too. Moreover the factory method should be covered by integration test. It will be located in same path as unit test, with same name, but the suffix added in the class name will be “IT” and not “Test”.

Finally the last parts of the project structure are the libraries. It will be two folders to store these references, one for normal compilation and other one with the references used in the test environment. In this dummy implementation I just use the reference to the JDK (java library) and Junit. If the system was more realistic, my proposal should be to use Maven tool which simplifies the creation and maintenance of the libraries, but in this dummy case I believe this was not adding advantage.

## Design patterns used

1. **Legacy wrapper:** This wrapper helps us to abstract old logic from the new implemented logic. It allows us to encapsulate and keep without interruptions the legacy code, in order to reduce points where new implementation is being mixed with old one. It gives us more security in our implementation, and avoids need to check every time the non-refactored pieces of code

1. **Abstract factory design:** This pattern allows us to create different instances of a family of entities without having the need of doing the declaration by itself.
2. **Singleton pattern:** Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. The singleton class must provide a global access point to get the instance of the class.