**Refactoring Strategy**

A good point to start describing the refactor strategy of TransportManager could be read what Martin Fowler says about what is a refactor: “*whenever you make a change to the code to make it clearer, without changing the software’s functionality, you are performing a refactoring*”.

After that, and before start coding something, we need to analyze several aspects:

* The quality attributes in our Software Architecture we want to achieve (in this case Maintainability and Extensibility).
* Uses Red-Green-Refactor technique.

Analysis Phase

**Maintainability**

The definition of this quality attribute is the ability of the system to support changes.

One thing is useful for detecting if our code is hard to maintain or not is analyzing if we are using one or many anti-patterns or if we follow the SOLID principles.

To achieve that, it’s important if we do an exhaustive code review trying to detect the aspects described above. For example:

* Check if there are classes with more than one responsibility.
* If we are using IoC to achieve loose coupling classes which make more testable.
* If we have code repeated.
* Detect if we are doing overengineering in whatever point.
* Detect if we have an anemic domain model.
* Detect if we have classes with a huge amount of code lines.
* Detect if we have unstructured code (Spaghetti Code).
* Detect if we are using magic strings and numbers in the code.

**Extensibility**

The definition of this quality attribute is the ability of the system to support new requirement with the less effort possible.

After the description of this quality attribute we can detect that both are very related. If your code is hard to maintain could will be hard to extend too. And if you solve most of the problems related to the maintainability it’s sure the software will be more extensible.

**Red-Green-Refactor**

This technique is based on Test-Driven Development and the steps we need to follow are:

* Check if all current functionality is covered by tests. If we find some requirement is not covered we implement a non pass test.
* Get the development to pass tests.
* After this point we can develop new improvements follow the first two steps.

Coding Phase

During the development we can follow an Agile approach like I’ve explained before, applying Red-Green-Refactor technique.

Breaking down the refactoring process into manageable chunks and performing timely testing before moving on to other updates.

Could be helpful if we try to do automatic refactors. As with most processes, the more it can be automated, the easier and faster refactoring becomes. Normally, the IDEs have many automated refactoring support to achieve that. For example, in Visual Studio IDE support many refactors directly or plugins like Resharper.

And finally, It’s very important to do frequently code reviews to detect as soon as possible if during the coding phase we are achieving the improvements.

After Refactoring Phase

In this point is very important to share the knowledge and the improvements with all the team, because if ALL the people we are involved not understand the purpose and the benefits of this effort could be in the future we detect the same situation.

**Project Structure**

We are assuming that TransportManager is a Microservice with its main purpose is to manage different operations with the blood samples.

Another assumption is that it’s a skeleton of a real Microservice because in the real world, TransportManager should receive Commands (operations to perform) from external devices.

And the communication between external components and TransportManager should be, for example, through by a message broker.

In the PoC I construct a Net Core Console Application that receive different parameters (it emulates the communication between external components and TransportManager) and this is the way to check how is the behaviour of the microservice when receive an external stimulation.

If we are assuming that TransportManager performs complex tasks, it requires separation of concerns. This is where the concept of a *Layered Architecture* comes in. The basic idea is to structure an application conceptual layers:

TransportManager.Application

This layer contains the bootstrap of our Microservice.

Here I configure the dependency injection using the dependency container provided by Microsoft by default in namespace *Microsoft.Extensions.DependencyInjection*. This is how I’ve implemented the IoC principle.

Another thing is configured in this point is logging (only shows info messages in console application) and the structure of parameters the service can receive.

TransportManager.Services

In this layer the service manage the different interactions between domain entities.

In a real service could be the point we add message broker handlers to receive commands from external components.

Here, thinking of quality attributes are more important for this service, I’ve applied different pattern I will explain below.

I’ve used a Factory Pattern combined with Strategy Pattern to encapsulate the creation of a new operation to achieve a high maintainability and extensibility.

Using the strategy pattern is easy in the future add a new operation. If we compare with UML diagram before refactor, when we need to add new operation could need to add a new if statement to check if apply one or other operation. Follow the old approach, adding more operations we add more complexity and less maintainability.

Otherwise, the advantage of using Factory Pattern is getting the responsibility of creating an *OperationStrategy* to the Factory class. Who calls factory not need take care about the dependencies need each strategy.

TransportManager.Domain

In this layer the service manager the entities where is encapsulated the business logic.

In this layer I’ve wanted to show how and new behaviours of an entity avoiding changes inside the entity. Achieving this, we get the purpose of a high extensibility and in consequence high maintainability.

Here I’ve used a Visitor Pattern, its purpose is exactly the same as we can get, add new behavior without changing the class on which it operates.

Applying this pattern the microservice will be more extensible because adding a new visitor (new behaviour) is not needed change the class on it’s applied.

For this PoC has been skipped the access layer for simplicity, the *Laboratory* entity is register like a singleton dependency to maintain the state of all samples, it’s only an example.

Other questions

The best option when we detect that our system or part of it has several code smell and not achieve the original quality attributes to get a successful product is the sincerity. What it means? Assuming I have a Principal role in the project I should speak with the stakeholders what are our problems and this is the point to manage and take care of it.

One of the most important stakeholders is the SA, normally is out of daily implementation, daily code problems. The Principal role has to be a connection point between development and software architecture. The Principal needs to explain accurately what are the problems to the SA. But, at the same time, say that the team are strong involved in achieving this challenge.

After SA and Principal are aligned, this is the point when we need to communicate the situation to rest of stakeholder (Project Manager, Team Lead, …). Explain, how more sincerely better, what is the situation, how it will impact in the future if not manage and try to solve. But, at the same time, confident that the team has the ability to solve all the challenges we have.

After that, Principal assumes the responsibility (with SA help) to create a refactor plan and after that it should be shared with the stakeholders. This is the point we need to align and think how fit all these changes.

The only way to ensure the design and implementation follow the Software Architecture is periodically does a checkpoint with SA, revise if the technical design achieve the quality attributes define in Software Architecture if there are any misunderstanding or misalignment.

The key point is, how more often this checkpoints happen, less probability to have huge refactors in consequence of misalignments with the Architecture.

When we speak about how we can maintain the code quality steady, we have some mechanism to achieve that, some of them automatically and others manually.

If speak about automatic processes we can follow to have a good code quality is, for example, a well defined source code pipeline for continuous integration. For example, before commit code in a development branch, the tests have to pass, the code analysis checks have to pass and some reviewers need to approve the code before it’s committed successfully. Another important thing could be added code analysis rules and apply that in the CI pipeline. And finally define SonarQube rules (in the CI pipeline) to avoid code smells.

On the other hand, it’s important the mentoring and communication with the team, all people need to understand the importance of a good code and how it can impact on the success of our product. Another important thing is doing, for example, every week, code reviews with the team sharing knowledge, doubts and detecting lacks or code smells.

If we speak in a specific situation with 2 team (6 people each one) I will establish these points to follow:

* Every pull request is needed to approve for 2 people.
* If the pull request is a core functionality a quick code review in Pair Programming (for example) to detect problems it’s so difficult to see in a PR.
* In the planning, detect tasks are core functionality. These tasks will develop in Pair Programming.
* Establish a protocol to do Code Reviews every week:
  + Each week 2 volunteer choose a topic
  + For this topic try to find 3 aspects:
    - Exceptional piece of code (It will be a reference for all of us).
    - Good piece of code (It’s with enough quality but detect some improvements).
    - Code smell (It need a complete refactor).

And for the last question, *Which steps/process would you get in place in order to ensure the team does not steps away from the original architecture*, if we accomplish all I explained before with a continuous checkpoints (code reviews, technical design review with SA) is the best way to be aware the development follows the right way.