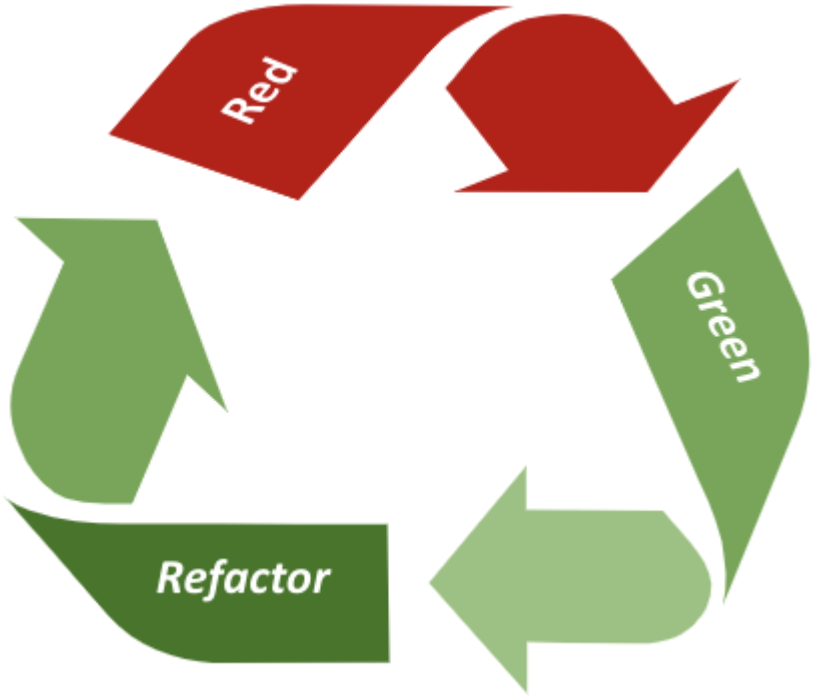
Guided Exercise 3 Statement



Refactoring and Simple Design

Software Development

April 2024

Degree in Computer Engineering - Double Degree in Computer Science and ADE

INDEX

1. OBJECTIVES AND CASE STATEMENT

The main goals of this guided exercise are:

- To practice refactoring techniques.
- To practice the principles of simple design and basic design patterns.

As a starting point for this guided exercise, we provide the source code that implements the functionalities proposed in the guided exercise 2. Therefore, as a first step, it will be necessary to ensure that you can verify the code with the test cases defined. It is very important to bear in mind that the code provided implements the functionalities described in the statement of the guided exercise 2, without taking into account the adaptations established during the practice classes.

First of all, you should reorganize the code in such a way that different predefined refactoring situations can be solved. These situations have been intentionally introduced into the code to facilitate the learning of this technique. Section 2 describes some cases in which Refactoring is necessary.

Subsequently, Section 3 will address several improvements to the technical architecture proposed for the solution of the guided exercise 3. These improvements will allow you to apply the fundamental principles of simple design and some basic design patterns. Section 3 of this statement describes the patterns recommended for the exercise.

2. REFACTORING

We have identified some situations in which it is necessary for the reorganization of the source code in the UC3M Travel component. You must follow the instructions below for the reorganization (refactoring) tasks. After every step, you must check that the unit tests still work as they should (in some cases it may be necessary to update the source code corresponding to such tests or even add new test cases).

Step 2.1: Identification and resolution of refactoring situations.

The refactoring cases that have been intentionally included in the code are the following:

Mysterious names. Writing a disconcerting text is appropriate when you are writing or reading a detective novel, but not when you are reading a code. One of the most important parts of the clear code are the good names, so we think a lot about the functions of nomenclature, modules, variables, classes, so that they clearly communicate what they do and how to use them.

The most common refactorings, in this case, are the renaming ones: Change Function Declaration (124) (to rename a function), Rename variable (137) and Rename field (244).

Duplicated code. If you see the same code structure in more than one place, you can be sure that your program will look better if you find a way to unify them. Duplication means that each time you read these copies, you should read them carefully to see if there are any differences. If you need to change the duplicate code, you must find and capture each duplication.

The simplest duplicate code problem is when it has the same expression in two methods of the same class. Then all you must do is Extract Function (106) and invoke the code from both places. If you have a similar code, but not quite identical, see if you can use Slide Statements (223) to organize the code so that similar items are all together for easy

extraction. If the duplicate fragments are in subclasses of a common base class, you can use the Pull Up Method (350) to avoid calling each other.

Long Functions. Under our experience, the programs that live better and longer are those with short functions.

Since the first days of programming, people have realized that the longer a function is, the harder it is to be understood.

Ninety-nine per cent of the time, all you have to do to shorten a function is Extract Function (106). Find parts of the function that seem to go well together and create a new one.

Divergent Changes. We structure our software to facilitate the change; After all, the software is meant to be smooth. When we make a change, we want to be able to jump to a precise point in the system and make the change.

Divergent change occurs when you frequently change a module in different ways for different reasons. If you look at a module and say, “I'll have to change these three functions every time I get a new database; I have to change these four functions every time there is a new financial instrument”, this is an indication of a divergent change.

If the two aspects naturally form a sequence, for example, it obtains data from the database and then applies its financial processing to it, then Split Phase (154) separates the two with a clear data structure between them. If there are more round trips in the calls, create the appropriate modules and use the Movement function (198) to divide the processing. If the functions combine the two types of processing within themselves, use the extraction function (106) to separate them before moving them. If the modules are classes, then Extract Class (182) helps formalize how to do the division.

Large classes. When a class is trying to do too much, it often appears as too many fields and functions. When a class has too many fields, the duplicate code cannot be far away.

You can Extract the Class (182) to group several variables. Choose variables to go together in the component that makes sense for each one.

If the inheritance makes sense, you will find Extract Superclass (375) or Replace type code with subclasses (362) (which essentially is to extract a subclass) are often easier.

Comments. The reason we mentioned the comments here is because it happens frequently that you find code with abundant comments and you realize that the comments are there because the code is bad. The comments lead to an erroneous code that has all the negative defects that should lead to refactoring. When we finish, we often find that the comments are superfluous.

If you need a comment to explain what a code block does, try Extract function (106). If the method is already extracted but you still need a comment to explain what it does, use Change function declaration (124) to change the name. If you need to set some rules about the required state of the system, use Submit assertion (302).

Step 2.2: Update the code according to the code regulations.

Finally, the corresponding updates must be made so that the source code conveniently satisfies the rules and recommendations defined in the team's code regulations.

Step 2.3: Execution of the defined test cases.

Finally, it must be verified that the result of the refactoring is satisfactory and meets the set expectations. For this purpose, the defined tests must be re-run to verify the correct operation of the Hotel Management component.

Consider that this step must be executed immediately after the completion of each of the code reorganization tasks established for this guided exercise.

It is possible that the code reorganization tasks involve the updating or the definition and implementation of new test cases. If this happens, the task of updating test cases must

be done before testing the Hotel Management functionality.

Step 2.4: Register the functional and test code in GitHub

As the resolution of the guided exercise will involve several work sessions, each session must end with the correct implementation of, at least, a test case.

At the end of the work session, the Push and Commit commands must be performed in GITHub, according to the instructions provided in Guided Exercise 2.

Remember that it is also necessary to register the output log that shows the correct execution of the test cases (XML) in GitHub.

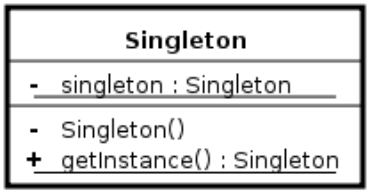
3. SIMPLE DESIGN

This section covers different technical upgrades related to the architecture proposed for the GE2 solution, allowing the application of basic principles of simple design and some basic design patterns.

Only one instance of the HotelManager, and the classes for managing the stores can be executed. To do so, you must apply the Singleton pattern.

Step 3.1: Usage of Singleton pattern to ensure that only one instance of a class can be generated

In Software Engineering, Singleton or single instance is a design pattern that allows you to restrict the creation of objects belonging to a class or the value of a type to a single object. Its purpose is to ensure that a class has only one instance that provide a global access point to it.



The singleton pattern is implemented by creating a method in our class that generates an instance of the object only if there is not an instance yet. To ensure that the class cannot be instantiated again, the scope of the constructor is regulated (with access modifiers as protected or private).

You can find below some Java source code corresponding to an example of Singleton pattern implementation [1]:

```
class SoyUnico(object):

     class __SoyUnico:

       def __init__(self):

            self.nombre = None

       def __str__(self):

             return `self` + ' ' + self.nombre

      instance = None


    def __new__(cls):

       if not SoyUnico.instance:

            SoyUnico.instance = SoyUnico.__SoyUnico()

       return SoyUnico.instance


    def __getattr__(self, nombre):

       return getattr(self.instance, nombre)


    def __setattr__(self, nombre, valor):

       return setattr(self.instance, nombre, valor)
```

[1] Example obtained from https://jarroba.com/patron-singleton-python-examples/

Step 3.3: Upload the functional and test source code to Git repository

As the resolution of the guided exercise will involve several work sessions, each session must end with the correct implementation of, at least, a test case.

At the end of the work session, the Push and Commit commands must be performed in GITHub, according to the instructions provided in Guided Exercise 2.

Remember that it is also necessary to upload the output log that shows the correct execution of the test cases (XML) in Github.

4. Publications related to refactoring.

Beyond its practical application, many researchers are working on Refactorig.

For example, the article "Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices" (https://www.mdpi.com/2079-9292/11/3/442) analyses the impact of refactoring on energy efficiency.

Search in Google Scholar (https://scholar.google.com) for publications related to refactoring. In a word document, each team member will include the reference (title, name of the journal or conference where it has been published, web address where to find it, authors and date of publication) to two articles that have been published in the last two years. Each student will include a brief summary of the articles (one paragraph per article), explaining briefly why they are relevant for this subject.

5. TASKS TO DO

1. Create a new repository on GitHub for this guided exercise. Name this repository according to the rules described in Guided Exercise 2 (GXX.2023.TYY.EG3) and invite your lab teacher.
2. Clone the project and include the code and folders structure available in Aula Global.
3. Apply the necessary refactoring and design techniques, the mentioned ones as well as those identified in the different practice sessions.
4. The code must comply with the PEP8 standard according to the pyLint tool.
5. Include the word document with the selected publications.
6. Upload the refactored and redesigned code, along with the tests and reports generated before each commit.
7. Download the code from GitHub and upload the ZIP to Aula Global GE3 delivery task. Follow instructions on this task.

6. RULES AND PROCEDURES

This exercise will be solved in groups (same groups as of GE2). During the exercise you should push in the corresponding branch of your project of the GitHub repository:

- The source code result of the refactoring and redesigning processes.
- The code of the tests with the necessary adjustments.
- The reports with the results of the execution of all the tests. These reports must be saved in a folder called "docs\test_reports".

The deadline for this exercise is April 26, 2023 before 23:59.

In accordance with the standards of continuous assessment established in this subject, if a team does not publish the solution of the exercise before the deadline, the exercise will be evaluated with a score of 0 points.