DUBLIN INSTITUTE of TECHNOLOGY
Institiúid Teicneolaíochta Bhaile Átha Cliath

Assignment 1
Project Report

DT8265
Higher Diploma in Computing

Michele Ercolino

School of Computing
Dublin Institute of Technology

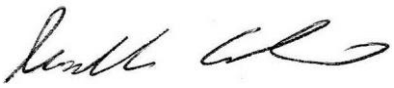
13/10/2017

School of Computing
Scoil na Ríomhaireachta

Declaration

I hereby declare that the work described in this assignment is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:



_____

Michele Ercolino

13/10/2017

Table of Contents

1. Introduction

The project is about a software program that manages the attendees of an event, storing details of each attendee into a file and/or into a database. The main purpose of this project is to apply the SOLID design principles creating a program which it is reusable, maintainable and extensible.

2. SOLID Design Principles

SOLID design principles are five guidelines that have been promoted by Robert Martin to make software easier to maintain, to reuse and to improve readability and flexibility.
Design principles are recommendations to design better software and they help to define patterns. Design patterns are used to simplify the identification of common problems and solutions. They exists to give a formal name and definition to common practices in software development (Galloway et al., 2014), enhancing communication among developers and maintainability and reusability of applications.

2.1 Single Responsibility Principle (SRP)

This principle states that a class should have only one reason to change. This means that a class should have only one responsibility. If a class has more than one responsibilities, this implies that this class will have more than a reason to change. For instance if a class has two responsibilities, this creates a coupling between them and changes in one responsibility will affect also the other one. This principle is closely related to the Separation of Concerns which declares that each component in a software should have only one concern and therefore different concerns should be separated and distributed among different components in the system. Basically it suggests to build several classes that hold their specific own behaviour. In this way, each module is independent to other modules in the system and therefore it will be easily reusable and the system itself will be easily maintainable.

2.2 Open-Closed Principle (OCP)

The Open-Closed Principle affirms that each entity in a system should be open to extension but closed to modification. This principle is very important regards of the extensibility and thus the longevity of a software. Each software component should be closed to modification, in other words it would not need to be modified after being created and at the same time it will be open to extension having a longer life without modifying its internal structure. Following the rules of OCP would help to build a software which it will not need to be changed when new features are added to it. This is what nowadays allows add-ons and plug-ins to be applied to existing software.

2.3 Liskov Substitution Principle (LSP)

This principles comes from a concept formulated by Barbara Liskov, which states that each child class can be substitutable for its base class without any side effects.

This principle is strictly related to OCP in a way that violating LSP principle will cause the violation of OCP's rules. This would happen, for instance, when a function which references to a class wants to replace this class with its child class. If the subclass violates LSP, because it has some method which is not behaving like parent class (i.e. overridden methods behave differently), this will make the function not working correctly for all derivatives of the base class. This implies that the base class needs to be modified due to the creation of a new derivative class. Obviously this violate Open-Closed Principle as well.

2.4 Interface Segregation Principle (ISP)

Interface Segregation principles indicates that interfaces should be not made by many methods but it is preferable to create many small interfaces instead. This is because a large interface will be difficult to use among several classes as it will enforce the classes to implement methods that probably they do not need to. As consequences this would break also the single responsibility of the class itself and the interface too. So the principle suggests to create many small interfaces made by specific methods and then allowing the classes to implements as many interfaces as they need to build their own behaviour.

2.5 Dependency Inversion Principle (DIP)

This principle states that objects do not need to be concerned about the creation of their dependencies and therefore be dependent of them. As Robert Martin explains in his works, high level modules should not depend on low-level modules but they should depend on abstractions (2003). High level modules can represent the identity of an application and they cannot be dependent of details because this would imply that changes in details would lead to a modification to the system structure. Basically this principles suggests to not initialize any other object inside a class but let someone else handle it. This will prevent a strong dependency that will make objects not reusable unless along with their dependencies. Violation of this principle would bring to a strongly coupled components and therefore not maintainable and reusable system. The creation of some kind of interface which will handle the dependency between objects would make the single components interact only to the interface, not be tightly coupled to each other, and consequently building independent and reusable objects.

3. Implementation

The program consists of 11 classes and 4 interfaces.
Classes are:

- Person: which holds generic person details, like name, email and phone number and implements IPersonDetails interface.
- General Employee: derived from Person.
- Contractor: derived from Person.
- Guest: derived from Person.
- ConsoleOutput: implements IOutput and display list of attendees on console.
- SmsService: implements IMessageService interface to send message by mobile phone.
- EmailService: implements IMessageService interface to send message by email.
- FileStorage: implements IStorage interface and it holds the logic to save data on file.
- SqliteDbStorage: implements IStorage interface and it holds logic to store data on SQLite database.
- EventMeeting: this class handles events, creating the event, listing all attendees who take part of event and closing the event itself sending messages to all attendees.
- EntryPoint: in this class is located the main method to launch the application.
  An event is created called “Job Event”, then few attendees are added to the event. They are two guests, one general employee and a contractor.
  After that the program stores the attendees first to a csv file and subsequently to a SQLite database. To accomplish this task, the storage system needs first to be initialized, which means create headline for a csv file and create a table for a database instead. Afterwards the list that holds attendees is uploaded to the storage.
  Then the program will display the list of attendees to the Console and at the end when the event is over all attendees will receive an end event message.

Interfaces are:

- IPersonDetails: defines methods to retrieve person details, like name and contacts details.
- IStorage: defines functionalities necessary for a storage class, such as storeData and initializeStorage
- IOutput: defines a single method to display elements from a list of IPersonDetails.
- IMessageService: defines a single method to send a message.

4. How the application has followed the SOLID Principles

Each class in the application has one responsibility, applying the SRP. For instance the EventMeeting class is responsible to create an event collecting the attendees of the event itself. It is not responsible of displaying the list of attendees as this functionality is held by any IOutput object. Moreover it is not directly responsible of sending a message as this task is accomplished by calling an IMessageService object.
Similarly, EventMeeting could be shown as an example of OCP. It is open to extension as for instance can include in the list of attendees any object that implements IPersonDetail interface and can send any kind of message no matter which service is used. So if the application needs to extend the list of attendees to another object which is not a Person, this object needs only to implement IPersonDetails interface and it can be added to the event without modifying EventMeeting class.
In addition, also FileStorage and SqliteDbStorage classes can be considered an example of OCP. They are not strictly coupled to this specific application, they can be used to store data of different object, not necessarily IPersonDetail objects.
Analysing the Liskov substitution principle, it is evident that all subclasses of Person class can be substitutable for it without any side effects as they inherited directly from Person without overriding any particular behaviour.
The dependency inversion principles is applied in EventMeeting class. EventMeeting class is not concerned about the creation of sms or email objects to send messages. In fact, in EntryPoint an IMessageService object is created and then it is passed as argument to send message method of EventMeeting and accomplish the task avoiding EventMeeting cares about its dependencies. So EventMeeting will depend on an abstraction (IMessageService) and not on concrete classes.
About the Interface segregation principle, the application has no evident examples of the principles. Every interface is defined by few related methods as it supposed to be. However as there are no classes that implement more than an interface, the purpose of ISP cannot be shown easily by this program. The chance of violating the principle is very low due to the nature of the application requirements. For instance, there is no example of class which needs to implement many interfaces and it could have violated the principle declaring all the methods in one large interface instead.

5. Conclusion

The application can be considered a good implementation of SOLID design principles. However some behaviours have been enforced to follow the requirements and therefore reducing the level of abstraction that the application itself could have achieved. For instance, the displayItems method of IOutput objects could have been more abstract and not be applied only for IPersonDetails objects. This has been done due to the specific requirements of listing the attendees with only first name, last

name and mobile number. For this reason, it could not be applied the toString() method of Person class as it includes also email attribute and changing this method would have broken the abstraction of storeData method in IStorage. Eventually, for the project purpose, the current level of abstraction has been evaluated reasonable.
Furthermore it could have been implemented a new interface which holds the storage connection (i.e. open and close behaviours) but again for the purpose of the project it has not been considered necessary. Initially the IStorage interface included a definition of close connection method. After a detailed observation, it comes out that the method itself in the IStorage interface violates the single responsibility and interface segregation principles and thus it has been deleted from IStorage.

References

Galloway, J., Wilson, B., Allen, K.S. and Matson, D. (2014), *Professional ASP.NET MVC 5*, Wrox, a Wiley brand, Indianapolis, IN.

Martin, R.C. (2003), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, Upper Saddle River, NJ.