Adoption of Design Principles and Design Patterns for Developing Software Application

R Prajna
Dept. Of Information Science and & Engineering
RV College Of Engineering®
Bengaluru, INDIA
prajna.rb93@gmail.com

Prof.Kavitha S N
Dept. Of Information Science and Engineering
RV College Of Engineering®
Bengaluru, INDIA
kavithasn@rvce.edu.in

Abstract - **Design principles and design patterns play a vital role in software development. Over recent years research and development team from both academia and industry have focused on issues related to handling technical debt. Poor design and implementation choices are main reason for high technical debt. Increase rate of technical debt results in code smelling. Adopting suitable design principles and correct design patterns plays a major role to prevent the code smelling. Solving the design problems that occur recurrently in object-oriented software development, several design principles are proposed. Proper understanding and implementing design patterns help indirectly to achieve design principle. In this paper we have briefly explained one of the widely used "SOLID" design principles in object-oriented applications. Discussed applied design patterns to accomplish "SOLID" design principles in our project.**

Keywords - technical debt, code smelling, design pattern, design principles, object oriented.

I.INTRODUCTION

During implementation of the software application, it's not sufficient for code to work. Implementation features not following the proper code structure will result in code duplication, difficult for other developers to understand the code and debugging is challenging. The demand for software application with righteous object-oriented design structure is common in industry and academia. Righteous design structure guarantees application going to become more "understandable, reusable, flexible and maintainable". Typically, developers design their applications using either their own preferred styles or the framework followed in their organisation. This procedure takes a long time to understand and is difficult to maintain in the future. The major challenge is to write the quality software by following good software design practices.

For the last few decades, the object-oriented based software design approach has become very common. Good objected oriented design results in low coupling and high cohesion. Number of design principles and design patterns are specified to improve the design quality of object-oriented based software applications [4]. Many real-time applications are implemented using the object-oriented design patterns. In the initial stage of development, it is very essential to understand the features of design principles and design patterns.

One of the significant features of object-oriented program is abstractions and reusability of functionalities. Initial stage of development when developer follows their own way of designing the program, many a time designs are not up to the standard and lack of documentation. These may cause issues at the later stages of development life cycle. Understanding and adopting the proven design solutions is always a better choice. In the object-oriented based software applications, many design principles and design patterns are defined and adopted in large scale.

Software design principles represents a set of guidelines that help developers to avoid having a bad design [5]. Unlike design patterns which are highly dependent on the project or domain context, design principles are in general suitable for designers assisting to obtain a shared understanding of overall architecture of application [3]. Generally, design principles provide best practises for designing software while keeping in mind the project's long-term maintenance and expansion. Object oriented based design principles generally demonstrate the essence of a programming language concept. Couple of proposed object-oriented design principles are Single responsibility principle (SRP), Open Closed Principle, don't repeat yourself (DRY) [5], "You aren't gonna need it" (YAGNI)

In software engineering adopting design patterns are widely accepted as a good practice. A design pattern is proven general repeatable solutions to a design problem with the intention of making the solution reliable and reusable. To implement in software development several design patterns are proposed, and each design pattern offers the solution to a specific problem. From the developer's viewpoint design pattern improves the code readability and maintainability [6]. In object-oriented applications, design pattern gives the approach to address a recurrent design problem, also describes the problem, and provides the hint when to apply it and post implications.

Adopting the right design principles and design patterns is very challenging, improper selection and forcefully trying to add the design patterns may cause complexity in code maintainability. The main goal of this paper is to explain the effect of "SOLID" design principles, dependency injection pattern and singleton design pattern to achieve the abstractness, which indirectly impact on software quality of our project. The project is named as Inpatient pharmacy medication management.

This paper is categorized into total 6 sections. Section II describes the literature survey, section III concepts of "SOLID" design principles, Section IV describes the implementation work, section V outlines the evaluation result and section VI conclusion of the research work.

II.LITERATURE SURVEY

For the past three decades, the object-oriented procedure has become the most common software design practice. Survey paper [8] focused on describing significance of following object-oriented design best practices. Based on survey authors have come up with 49 design measures such as avoid duplicates, avoid long methods etc which indirectly contributes to build the design best practices while developing the application.

Harmeet Singh, Syed Imtiyaz Hassan [9] described impact of SOLID design principles on software quality. Overview of “SOLID” principles and empirical analysis of these principles by using a working prototype, applying the design principles and later stage evaluated the prototype with measuring metrics. Authors have focused on evaluating the quality using CKJM metrices considering the cohesion and coupling measurements in the payroll system application.

Patterns were first used in the field of architecture by Christopher Alexander, who documented reusable architectural plans for manufacturing high-quality designs [1]. Object-oriented software developers began to use themes in the mid-1990s. The so-called GoF (Gang of Four, Gamma, Helms, Johnson, and Vlisides) catalogued 23 design trends aimed at addressing certain commonly-recurring object-oriented design needs [2].

R.Subburaj Professor et al. [7] , focused on explaining the advantage and disadvantage of design patterns for object oriented based software development. Design pattern catalog were described considering three basic classes of design patterns: structural design patterns, creational design patterns and behavioural design pattens. Authors have also proposed different ways to select design patterns and avail those in software development.

F. Khomh and Y. Guéhéneuc [10] proposed a qualitative research on theory of design patterns actually applied to raise the abstraction level of programming. Survey was Performed mainly on types of patterns proposed by Gamma et al[2] , and positive impacts and negative impacts of those design patterns. Authors have explained impact of design patterns on software quality as well as described other factors of software engineering like development tools, knowledge sharing, reverse engineering, forward engineering and documentation.

MU Huaxin, JIANG Shuai[6] focused on study of object oriented design patterns in development of software. Observer pattern , factory method pattern and decorator pattern were considered for research work. Authors have referred these design pattern from 23 design patterns theory mentioned by Gamma ET al[2].Based on understanding of design patterns working way design principles were suggested by researchers.

III. OVERVIEW OF DESIGN PRINCIPLES

Design principles are basic standards that help to create and manage a software framework for software designers and developers. Design principles help software architecture beginners to eliminate pitfalls and mistakes in object-oriented design.

To handle the design issues of application “SOLID” principles provides right structure. “SOLID” design principles are widely used in object-oriented based application design. In the 1990s Robert C. Martin defined and explained these principles [12]. “SOLID” design principles provided greater scope for developers to move from tightly coupled code to loosely coupled code with encapsulation to meet the business requirement properly.

SOLID is an acronym of the following,
S: Single Responsibility Principle (SRP)
O: Open closed Principle (OSP)
L: Liskov substitution Principle (LSP)
I : Interface Segregation Principle (ISP)
D: Dependency Inversion Principle (DIP)

3.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is considered to be no more than one reason for each Class to alter. Every class should have one responsibility, class that has more than one responsibility or chance of adding more responsibility to the class indicates high coupling. High coupling may cause unstable designs that can break down any conditions for improvement in unpredictable ways.

3.2 Open Closed Principle (OSP)

According to this principle software entities namely modules, classes or functions should be open for extension but it should be closed for modification [9]. Alteration of single software component may impact dependent components, as a result there is a possibility of unacceptable program behaviour. The program becomes compact, static, unstable and unrefusable. This is solved in a very simple way by the open-closed theory. According to OSP one can expand the actions of classes, functions or modules by adding new features code but making changes or updating to the existing code is not allowed.

3.3 Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) related to substitution property. Functions or modules which going to use references or pointers to super classes can be replaceable with its object of child classes [9]. This shouldn’t break the application. This is related to the substitution property.

3.4 Interface Segregation Principle (ISP)

Modification in an unrelated interface will result in the client code being modified unintentionally. This leads to coupling of all the clients. Cohesive interfaces which can be derived from abstract classes is better approach, Interface Segregation Principle (ISP) says the same.

3.5 Dependency Inversion Principle (DIP)

Dependency of high-level model on lower-level modules is not a good programming practice. Through abstractions they can depend on. Abstractions should not depend on details, rather details should depend upon abstractions. Adding an abstraction layer in order to separate the higher-level modules from lower-level modules is a solution [9]. We need to separate this abstraction from the problem's details in order to adhere to the concept of dependency inversion.

IV. IMPLEMENTATION

In order to further comprehend adopting “SOLID” design principles and design patterns to achieve these principles, we have taken up project on pharmacy management for inpatient.

Application is developed in .NET framework and programming language is C#. Basically, followed the object-oriented pattern while designing the application development.

To avoid facing the common design flaws like placing more responsivities on single class, tight coupling between the other classes followed below steps.

4.1Choosing the design principles

Compare with the design principles like Don't repeat yourself (DRY), Keep it simple stupid (KISS), however SOLID principles benefits are suitable for real time applications and advantages are evidently and easily reflected.

Application consists of many classes such as Patient Demographics, Patient Allergy, Patient clinical notes, Patient medication dose and Patient dose calculation class. All classes exhibit single responsibility. Each class implements the interface which indirectly achieves the open closed principle. Example the class named cPatientDose closed for adding the methods related to dosage time interval calculation, but open for adding the functionalities like quantity per dose, indicator used in dose calculation.

The main challenge is to implement the dependency inversion between every class which contains the several methods within it. According to dependency inversion principle higher level modules and lower-level modules must communicate through abstractions. Abstraction shouldn't depend on the details rather details should depend on abstractions [12].

Development started with realizing "SOLID" design principles primary approach, below are the list of those

- ➢ Creating more classes and made sure every class has single responsibility to handle.
- ➢ Created the abstract types i.e., interfaces and abstract class wherever there is scope to add abstraction.
- ➢ Followed to implement the class with small cohesive types

Fig.1. shows the activities we have implemented to follow the design principles in our work. To achieve the dependency inversion principle and to reduce the coupling between the classes we have adopted design patterns.

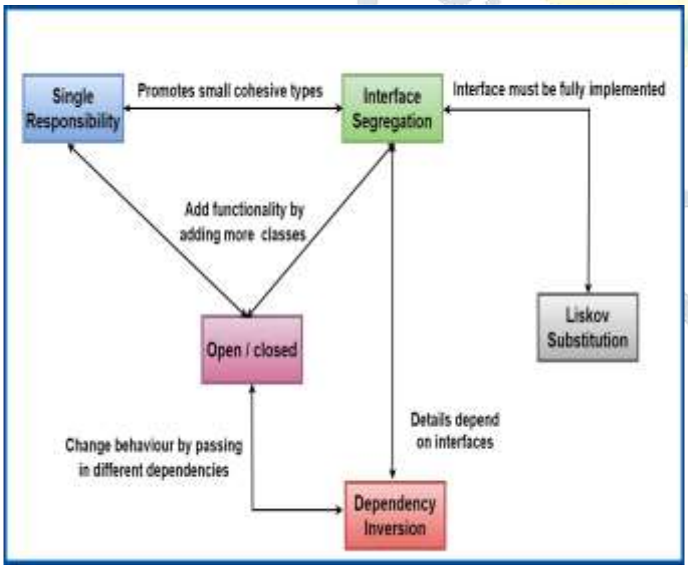
Single Responsibility
Promotes small cohesive types
Interface Segregation
Interface must be fully implemented
Add functionality by adding more classes
Liskov Substitution
Open / closed
Details depend on interfaces
Change behaviour by passing in different dependencies
Dependency Inversion

Fig. 1 Implementation approach followed for "SOLID" principles

4.2 Choosing correct design patterns to address the design issues

A design pattern describes the core features of a typical design structure that make it useful for designing reusable object-oriented designs by naming, abstracting, and identifying them. It identifies the classes and their instances, as well as their tasks and collaborations, and the responsibility allocation. Each design pattern focuses on a recurring design problem which occurs while implementing object-oriented design structure. We have chosen below design patterns to solve and implement the design problem occurred during development.

4.2.1 Factory Method Pattern

The Factory Method Pattern specifies instant of object is created through interface but subclasses that is factory class decides which class to instantiate [2]. Factory Method lets a class defer instantiation to subclasses. This comes under the creational pattern section of Gang of Four (GOF) [2].

In our application, there is a scenario in which client class i.e., consumer class can create the instance of the other dependent classes without knowing the details of how they are created. Fig 2 indicates the scenario of factory pattern method implemented for suggested dose route. Dose route basically indicates the path by which medication is taken into the body through oral, nasal inhalation, mouth inhalation or buccal.

- ConcretePatientDoseRoute indicates the different medication consumption route classes.
- Those Classes implement methods declared in interface IPatientDoseRoute.
- The cPatientDoseRouteFactory is responsible for creating one or more concretePatientDoseRoute.This subclass has the knowledge of creating instance of patient dose route.
- The consumer class cPatientDoseCalculator will create object of factory class instead of concrete class, through factory class refers the implementation of methods in concrete class.

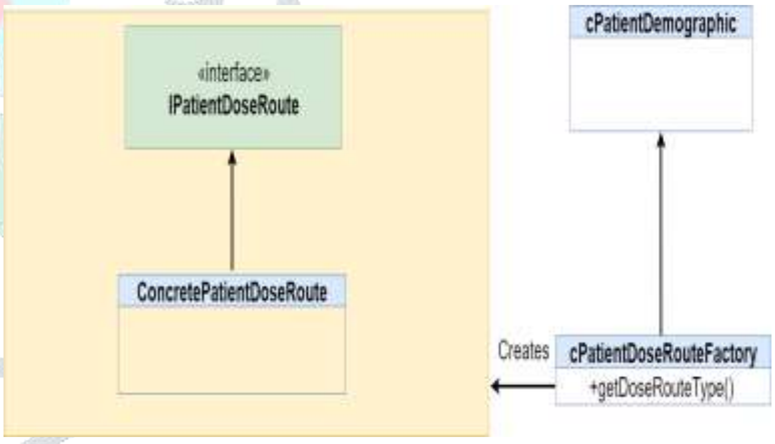

Fig. 2 Class Diagram for Factory Method Pattern

Same way we have implemented other scenarios in the application wherever there is scope of implementing factory method pattern.

4.2.2 Dependency Injection (DI) Design Pattern

Dependency injection design pattern provides a technique to create applications that are loosely coupled. Dependent objects can be injected into class through methods, properties or constructor [13].

Adopting interfaces-based implementation, & communication between the higher level and lower-level classes through abstractions improves the reusability of code. But still instantiation of lower-level classes has to be done at someplace. The client class will depend on interface instead of depending on concrete classes. This makes even easier to use classes which has different implementation for same methods in interfaces. Next introducing injector class which basically sets the instant of lower-level class to higher level class either through a constructor, property or a method.

In our application we have accessed data from different sources. Consider Fig.3 in which client class is using service classes via interface reference to get the required data. Rather than directly creating objects of classes, interfaces are used. These service class instance will be created in the injector class. So, constructor dependency injection pattern provides the objects reference. DI accomplishes complete decoupling between these two client and service classes.

- The cPatientService class is the injector class, which sets the object of a service classes to the PatientBuisnessLogic class i.e., client class.
- cPatientBuisnessLogic class instead of directly creating instance of service class, via interface IPatientData gets object reference.
- Constructor injection pattern used to provide the dependency through injector class.

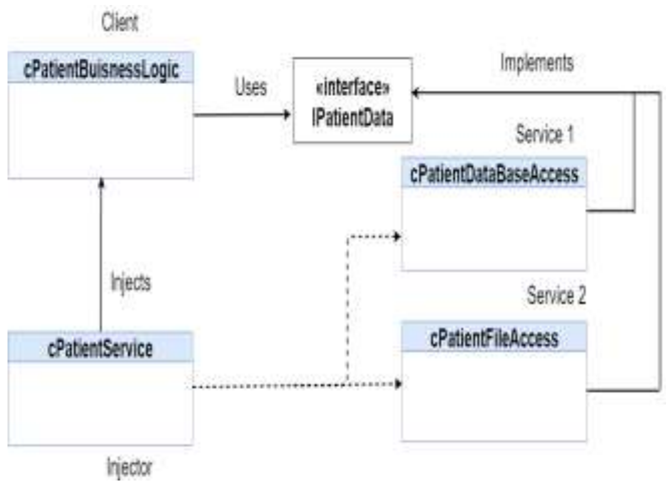

Fig. 3 Class diagram for dependency injection pattern

4.2.3 Singleton Design Pattern

The singleton design pattern specifies that a class will have only one instance throughout the program and provides a global point of access to it [12].

Designing the objective of Patient properties and related variables should maintain the same values across the all classes. Considering patientAge properties value which is required in patientDose and patientDemographics class, whenever there is change in patientAge variable value should reflect in referring classes. So, creating singleton class which has variables and properties declaration and definition solved the inconsistent values referring issue. Fig.4 shows the singleton class we have implemented.

- cPatientPropertiesSingleton is a singleton class, which has static method instance.
- Instance method when called first time singleton object will be created.

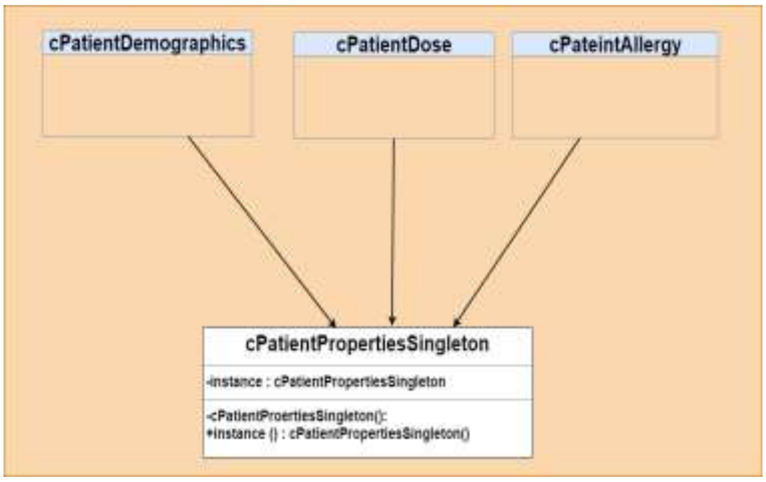
cPatientDemographics
cPatientDose
cPateintAllergy
cPatientPropertiesSingleton
-instance : cPatientPropertiesSingleton
-cPatientProertiesSingleton():
+instance () : cPatientPropertiesSingleton()

Fig. 4 Class diagram for singleton design pattern

V. EVALUATIONS

The object-oriented software application is difficult to change if it shows high coupling, low cohesion and it is difficult to write the unit test cases. After applying the design patterns in our working prototype, we are able to follow the design principles in our development components. We have conducted code analysis using NDepend tool. By finding the abstractness metrics of the project which follows the theory and metrics proposed by Robert C Martin [11].

To evaluate our project exhibits good range of abstractness, NDepend tool evaluated code for stability metrics. Tool indicates code is abstract assembly consists of many abstract types like interfaces and abstract classes and a smaller number of concrete types.

Fig.5 shows the abstractness vs instability graph, it detects assemblies that are concrete and stable which indicates painful to maintain. Assemblies that are abstract and instable indicates potentially useless.

5.1 Abstractness (A)

The ratio of abstract types to the total number of classes. Nc and Na indicates the number of classes and abstract types in the project respectively. Equation (1) shows finding the abstractness in component [12]. The value range for this metric is 0 to 1.

$$A = \frac{Na}{Nc} \tag{1}$$

5.2 Instability (I): The ratio of efferent coupling (Ce) to total coupling. Equation (2) is used to find the instability [12]. Basically, calculation depend on number of dependencies component associated.

Ca (afferent couplings): The number of classes outside the component to that depend on classes within this component.
Ce (efferent couplings): The number of classes inside this component that depend on classes outside component.

$$I = \frac{Ce}{(Ca + Ce)} \tag{2}$$

In the Fig.5, dotted line indicates the main sequence. Component near to area of zone of pain is stable and concrete i.e., very difficult to modify. Components near to zone of usefulness is very abstract i.e., no dependents at all so it is not useful. If the Assemblies normal distance from main sequence is higher than 0.7, it is considered as problematic. Our case assembly is in the desired range of distance from main sequence.

TABLE 1
Metrics on Inpatient Pharmacy Medication Management

Project Name	Parameters	Values
Inpatient Pharmacy Medication Management	Abstractness	0.17
	Instability	1
	Distance	0.12

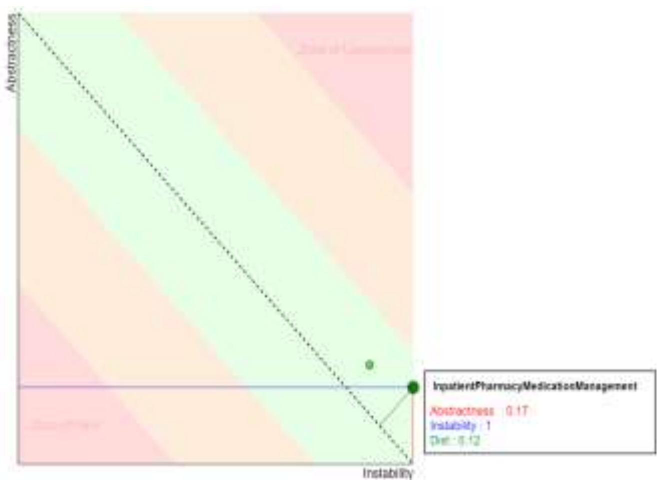

Fig .5 Abstractness vs Instability graph

VI. CONCLUSION

This paper provides software developers with information on the importance of design principles and design patterns and how they can influence the overall success of object-oriented applications. Our approach of design pattern selection process started with understanding the design principles, next we studied the design patterns which are indirectly helped to achieve the "SOLID" principles as well as other design issues we faced while developing the applications. This paper analyses the "SOLID" design principles importance in real-time object-oriented application and implementing three design patterns to achieve the same and to solve the other design issues.

REFERENCES

[1] C. Alexander, "The origins of pattern theory: The future of the theory, and the generation of a living world," IEEESoftware,vol.16,no.5,pp.71–82, September/October 1999.

[2] Gamma, E., Richard Helm, Ralph Johnson and John Vlissides,"Design Patterns: Elements of Reusable Object-Oriented software,"Addison-Wesley, 1995

[3] W. Haoyu and Z. Haili, "Basic Design Principles in Software Engineering," 2012 Fourth International Conference on Computational and Information Sciences, 2012, pp. 1251-1254, doi: 10.1109/ICCIS.2012.91.

[4] M. Oruc, F. Akal and H. Sever, "Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach," 2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT), 2016, pp. 115-121, doi: 10.1109/CONISOFT.2016.26.

[5] J. Braeuer, "Measuring Object-Oriented Design Principles," *2015 30th* IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 882-885, doi: 10.1109/ASE.2015.17

[6] H. Mu and S. Jiang, "Design patterns in software development," 2011 IEEE 2nd International Conference on Software Engineering and Service Science, 2011, pp. 322-325, doi: 10.1109/ICSESS.2011.5982228.

[7] R.Subburaj Professor, Gladman Jekese, Chiedza Hwata, "Impact of Object Oriented Design Patterns on Software Development", March 2015,International Journal of Scientific and Engineering Research , Volume6(Issue 2), ISSN 2229-5518

[8] J. Bräuer, R. Plösch, M. Saft and C. Körner, "A Survey on the Importance of Object-Oriented Design Best Practices," 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vienna, 2017, pp. 27-34, doi: 10.1109/SEAA.2017.14.

[9] Harmeet Singh, Syed Imtiyaz Hassan, "Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment", International Journal of Scientific & Engineering Research, Volume 6, Issue 4, April-2015 1321 ISSN 2229-5518

[10] F. Khomh and Y. Guéhéneuc, "Design patterns impact on software quality: Where are the theories?," 2018 IEEE 25th InternationalConference on Software Analysis, Evolution and *and Reengineering (SANER)*, 2018, pp. 15-25, doi: 10.1109/SANER.2018.8330193.

[11] Agile Software Development: Principles, Patterns, and Practices in C# Robert C. Martin (Prentice Hall PTR, 2006)

[12] R. C. Martin, Design Principles and Design Patterns, 2000. [Online]. URL: http://www.objectmentor.com