Journal of Computer Science

Original Research Paper

Impact of Design Principles and Patterns on Software Flexibility: An Experimental Evaluation Using Flexible Point (FXP)

[1,2]Muhammad Ehsan Rana, [2]Eddy Khonica, [1]Wan Nurhayati Wan Ab. Rahman, [1]Masrah Azrifah Azmi Murad and [1]Rodziah Binti Atan

[1]Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400, Serdang, Malaysia
[2]Faculty of Computing and Technology, Asia Pacific University of Technology and Innovation, Kuala Lumpur, Malaysia

Article history
Received: 24-06-2020
Revised: 13-08-2020
Accepted: 04-09-2020

Corresponding Author:
Wan Nurhayati Wan Ab. Rahman
Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400, Serdang, Malaysia
Email: wnurhayati@upm.edu.my

Abstract: Software flexibility is a crucial factor in designing and developing software as it reflects its capability to adapt to changes. It is a topic that has been discussed for a very long time which indicates its importance in software development. However, it is not easy to produce a flexible software design. Design principles provide fundamental concepts in designing good software. On the other hand, design patterns are proven solution to recurring problem. When used correctly, design principles and patterns can be used to improve software flexibility. However, it is necessary to evaluate its impact on software flexibility. For this purpose, this research will conduct an experiment by developing a simple application using Object-Oriented Programming (OOP) based on a case study. Based on the requirements of the case study, two SOLID design principles are chosen such as Single Responsibility Principle (SRP) and Open Closed Principle (OCP) while Strategy and Decorator for the patterns from the GoF. Then, Flexible Points (FXP) is used to measure the differences in software flexibility by comparing the solutions developed before and after applying design principles and patterns. This is aimed to prove that the chosen design principles and patterns have positive impact on software flexibility. Lastly, the result analysis shows that the use of the chosen design principles and patterns indeed improve the flexibility of the software. Therefore, the authors highly recommend adopting design principles and patterns in software development.

Keywords: Software Flexibility, Design Patterns, Design Principles, Flexible Points (FXP)

Introduction

In today's software development, it is extremely important to take software flexibility into system design consideration. It is a crucial factor among software quality attributes which focuses on the ability of a software to adapt to changes (Nelson *et al*., 1997; Subramaniam and Zulzalil, 2012). The concept of software flexibility was introduced in 1979, but the beginning of most of its literature were conducted in 1990's (Shen and Ren, 2006). Most of the literature review on software flexibility are rather old. However, till these days, software flexibility is still a must have quality in software development especially in object-oriented software projects. The existence of software design principles and patterns have encouraged the authors to evaluate its impact on software flexibility.

Shen and Ren (2006) highlighted that it is often unclear which and when software components require changes and should remain unchanged. Design principles and patterns are introduced over the years. The former provides the fundamental concepts to design a good software (Bräuer *et al*., 2018), while the latter provides proven solution to recurring problems. When used correctly, design principles and patterns can be used to improve software flexibility. This is vital because by taking flexibility into account during early software design can help to ease changes and save cost (Shen and Ren, 2006; Gorton, 2011). Although these principles and patterns can be used in producing more flexible software, but it still lacks an empirical evaluation on its impact on software flexibility. In existing literatures, there is yet to have a quantitative flexibility comparison in object-oriented software with SOLID design principles and GoF design patterns applied. For this reason, a simple object-oriented application is developed using JAVA programming language based on a case study. The initial implementation will then be compared to those after design principles and patterns are implemented. For this research, the design principles and patterns evaluated are limited to those from SOLID and Gang of Four (GoF) as these are the key core design principles and patterns being taught in academia. Then, the result will be empirically evaluated using Flexible Point (FXP) to check whether the chosen design principles and patterns improve software flexibility. This is the aim of this research.

The next section provides the review on existing literature on software flexibility, flexibility measurement techniques, SOLID design principles and GoF design patterns, followed by the methodology used in this research. The subsequent sections consist of the case study, design and implementation of the system. The remaining sections are namely analysis of the results, discussion and conclusion respectively.

Software Flexibility

Flexibility has become one of the key concerns in a software design. Flexibility is defined as *"the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed"* (IEEE, 1990). Loosely coupled components are the key factor in ensuring the flexibility of the system. Software architecture is another factor which greatly influence software flexibility as it decides the relationships between one component to one another (Lassing *et al*., 1999). In addition, software is flexible if it can easily adapt to changes during development or after software deployment (Shen and Ren, 2006). These statements highlight the crucial factors that must be taken into consideration to produce a flexible software design.

Zhao (1998) introduced two attributes related software flexibility such as "*system adaptability and system versatility"*. These two attributes basically require the system to handle changes in a way that it does not affect existing business operation. Moreover, Nurdiani *et al*. (2018) also identified three attributes that are related to software flexibility such as *"properties of change, flexibility perspectives and flexibility enablers"*. These three attributes are more towards the characteristics of change that affect software flexibility. Both researchers are conducting their research from different perspective, thus it may not be related to each other. However, software developers can still make use of these attributes as a reference to further understand software flexibility. Thus, enabling them to produce a highly flexible software.

Eden and Mens (2006) highlighted that it is not uncommon for software maintenance cost to exceed its development cost. This is due to there will be a high cost to implement a change to software without flexibility (Shen and Ren, 2006). As a result, it will be very expensive to maintain the system. However, by having a more flexible system, the maintenance cost can be greatly reduced (Gorton, 2011). Flexibility will also help in delivering high quality and reliable system within the constraints of cost and time (Abdullah *et al*., 2015). These statements basically prove how important flexibility is to a system as it affects the cost and quality of the system. Therefore, it is crucial to take flexibility into account in designing and developing a system.

Flexibility Measurement Techniques

Existing literatures have provided different techniques to measure software flexibility. Shen and Ren (2006) introduce a concept called Flexible Point (FXP) to measure software flexibility in a quantitative manner. FXP can be defined as *"a point or a location in software which can cause flexible changes to occur, upon which the external force F_e may apply"* (Shen and Ren, 2006). Shen and Ren (2006) highlighted five steps required to measure software flexibility using FXP such as:

- Identify flexible points which refer to things that can cause changes to software
- Calculate the flexible distance for every FXP to measure software change
- Determine FXP level and calculate flexible force
- Calculate flexible degree for every FXP
- Calculate flexible capacity

Eden and Mens (2006) introduced another method which takes evolution of software complexity metric into consideration. It involves the programming paradigms, architectural styles and design patterns chosen. In their research, only Visitor and Abstract Factory of the GoF design patterns are evaluated based on the case study provided. In their analysis, both patterns do help in providing better implementation policy when changes are required. However, the latter is technically worth it when the number of clients to implement is many. Despite providing better description on the nature of software flexibility, this method is not easy to use (Niu *et al*., 2011). For this reason, Niu *et al*. (2011) introduced another software flexibility measurement technique using *"Second-Order Cone Programming (SOCP) approach"*. The need to consider the consequences of external operation control force on top of software deformation size is the reasoning to the use of SOCP to measure software flexibility (Niu *et al*., 2011). The suitability of each technique depends on how software developers are going to use it.

Table 1: General flexible points (Shen and Ren, 2006)

NO	Change requirement	NO	Change requirement
1	Adjust the value range of data element	6	Add/delete business rules
2	Add/delete items in selection	7	Add calculation formulas
3	Add/delete data element	8	Adjust screen layout
4	Change data element type	9	Change external input interface
5	Modify calculation formulas	10	Change external output interface

Peng *et al.* (2009) research on software flexibility measurement using user-oriented approach has also included the five steps mentioned in (Shen and Ren, 2006) research along with user's manipulative capability and manipulative complexity. In their research, it is highlighted that FXP is easier to use to measure software flexibility and participants who are involved tend to be able to identify various problems related to software flexibility early. On the other hand, there are no other researchers who have made use of SOCP approach in software flexibility related research. Meaning that the SOCP approach has yet to be fully proven to be correct. Therefore, this research will utilize FXP for measuring the flexibility improvement in the object-oriented application developed before and after applying design principles and patterns.

Any changes to the requirements that can be satisfied by the software's internal adaptive mechanism can be conveyed using flexible point (Peng *et al.*, 2009). FXP is also a key indicator and the most basic unit of measurement of software flexibility (Niu *et al.*, 2011). Flexible point can be a function, a segment of codes, etc. where some general points are shown in Table 1. Based on the FXP, it can be further grouped into levels where lower FXP will have less changes and less technical while higher FXP will require more technical changes. This will be beneficial for software developers as they can use flexible points to predict the components that will have the least and most changes.

Impact of Design Principles and Patterns on Software Flexibility

Software design principles are fundamental concepts that helps in producing good design quality (Bräuer *et al.*, 2018). These principles are essential in creating a flexible and maintainable system. However, many difficulties are faced to develop software based on the design principles. This is due to the absence of clear instruction to correctly implement the principles (Bräuer *et al.*, 2018). As part of the research scope, only SOLID design principles will be evaluated in this research. SOLID is an acronym for five basic design principles such as *"Single Responsibility, Open Closed, Liskov Substitution, Interface Segregation and Dependency Inversion"* principle respectively (Martin and Martin, 2006). These five principles can be used as a guideline to develop a reusable and efficient system which is maintainable and sustainable for long term usage (Madasu *et al.*, 2015). By incorporating reusability and maintainability into the software development, it will also improve the flexibility of the software.

Design pattern can be defined as a tested solution to a programming problem with known pattern (Holzner, 2006; Walter and Alkhaeir, 2016). The concept of design patterns defines good practices to design software as opposed to antipatterns that represent bad practices (Arcelli and Di Pompeo, 2017). Gamma (1995) introduces GoF design patterns which has a total of 23 patterns divided into three categories such as Creational, Structural and Behavioral. Each pattern has its own intent and applicability, meaning that not all patterns are suitable to promote software flexibility. These patterns are set as the scope of patterns used in this research. Design patterns do not always improve the quality of the software developed (Khomh and Gueheneuc, 2018). In addition, lines of codes are expected to increase significantly by applying design patterns (Scanniello *et al.*, 2015). However, this does not pose as a disadvantage as in exchange, it will reduce code smell as well as improve code comprehensibility (Walter and Alkhaeir, 2016; Gravino and Risi, 2017). Therefore, it is necessary to select and apply the right pattern to solve the problem encountered.

Furthermore, most object-oriented software projects are using object-oriented principles and reusable design patterns to solve common recurring design problem (Oruc *et al.*, 2016; Thabasum and Sundar, 2012). With the use of design patterns, it is highly likely to result in a better and more maintainable system (Marouane *et al.*, 2018; Yu *et al.*, 2018). Moreover, design patterns also ease system modeling and improve the development process quality (Marouane *et al.*, 2018). Not only will it improve the quality of the system, but it also allows developers to have a rapid understanding on the software design, thus making maintenance easier (Yu *et al.*, 2018). As a result, the flexibility and maintainability of the system can be improved significantly. Due to these reasons, design patterns are still very popular in today's software development as it encourages flexibility, maintainability and reusability (Zhang *et al.*, 2014; Lano *et al.*, 2018).

Design Principles Chosen for this Research

In this research, the authors choose two SOLID principles that will be used in the experiments based on its characteristics to produce extensible and reusable software components which would result in promoting software flexibility. The chosen design principles are as follows.

Single Responsibility Principle (SRP)

Martin and Martin (2006) explained SRP as *"a class should have only one reason to change"*. Meaning that a single class must have one and only one responsibility. Sticking to this rule of thumb will prevent the software from having tightly coupled components as well as promoting clear responsibility. Thus, SRP can improve the flexibility of the system

Open Closed Principle (OCP)

"OCP is defined as software entities (classes, modules, functions, etc.) should be open for extension but closed for modification" (Martin and Martin, 2006). The former refers to the ability to extend a module based on new requirements, while the latter refers to no changes should be applied to the source upon extending the module. As a result, existing modules/components do not require recompilation and retest when an extension is added. Thus, this highly improves the flexibility of the software.

Design Patterns Chosen for this Research

It must be emphasized that wrongly used pattern may impact the quality of the software negatively (Zhu and Bayley, 2015). Therefore, not all patterns will be suitable for improving software flexibility. Based on the case study requirements and evaluation on the patterns' intent and applicability towards software flexibility, the authors choose two GoF design patterns which will be used in the experiments such as.

Strategy

Strategy pattern is one of the behavioral patterns from the GoF design patterns. The intention of this pattern is to *"define a family of algorithms, encapsulate each one and make them interchangeable"* (Gamma, 1995). This pattern is an alternative to inheritance, it breaks down volatile codes, encapsulates them as objects and uses them when needed (Holzner, 2006). This pattern resembles OCP where it is based on extensibility and reusability to promote software flexibility. No modification to existing classes is required, thus making the software flexible to changes. As a result, the flexibility of the software can be improved significantly.

Decorator

This pattern is a structural pattern from the GoF design patterns. It is intended to *"attach additional responsibilities to an object dynamically and provide a flexible alternative to subclassing for extending functionality"* (Gamma, 1995). As highlighted that decorator provides additional responsibilities, but those responsibilities can be removed when it is no longer required. In other words, adding or removing responsibilities will not affect the core modules, thus making the software flexible.

Methodology

There are many existing literatures conducted on software flexibility and design patterns from 1990's onwards. The years of study in this research will include literatures from 1990 to most recent. This is because software flexibility is an old topic which has very limited literatures from recent years. As a result, the authors would have to include these old references regarding software flexibility. However, design patterns seem to have more research interest as there are more and more related journal papers written. The authors will refer to existing researches as part of this research. This is to ensure that the literature review conducted is up to date.

Furthermore, this research uses quantitative research approach as flexible point is selected to measure software flexibility improvement in the OO application developed based on the case study. The application will first be developed using OOP language which is JAVA without applying any design principles and patterns. Then, adding design principles for the second comparison and adding design patterns for the final comparison. Only certain principles and patterns are chosen due to its suitability with the requirements of the case study. It is more accurate to measure the differences through experiment using FXP. The findings obtained from these experiments will then be used to justify whether the selected design principles and patterns have a positive impact on software flexibility.

Design and Implementation

To show the improvement in software flexibility, the authors consider the following case study which is implemented using JAVA. The design and implementation are focused on three parts such as simpler solution (without any design principles and patterns), after applying design principles and after applying design patterns.

Scenario/Case Study

Pyro Ice Cream Ordering System (PICOS) is an application that enables customer to order and customize ice cream based on the available toppings provided. Toppings are optional, customer can order ice cream without any topping. The toppings are provided by vendor, thus PICOS will not cover any modification to topping information. However, PICOS can be used by the Admin to update ice cream price and description. This system only accepts one customer at one time. A customer can only make a single order on one type of ice cream at any one time. Furthermore, PICOS provide a feature for customer to get ice cream recommendation such as random and most popular ice cream. The Admin of the system can update ice cream information, complete order and view report. The report feature only covers orders that are already completed.

Overall System Design

Figure 1 shows the Use Case diagram of PICOS system. There are two actors/users of the system such as Customer and Admin. Customer can order ice cream, add/remove topping and get ice cream recommendation. On the other hand, Admin can update ice cream info, complete order and view report. PICOS system does not keep track of customer information. Thus, customer does not require to login to the system. However, the system does keep track of order information to generate a sales report.

Figure 2 and 3 shows the workflow of PICOS system using Activity diagram for Customer and Admin respectively. Customer has two options such as order ice cream with or without topping and get ice cream recommendation. Once logged on, admin can perform three functions such as update ice cream info, complete order and view report.

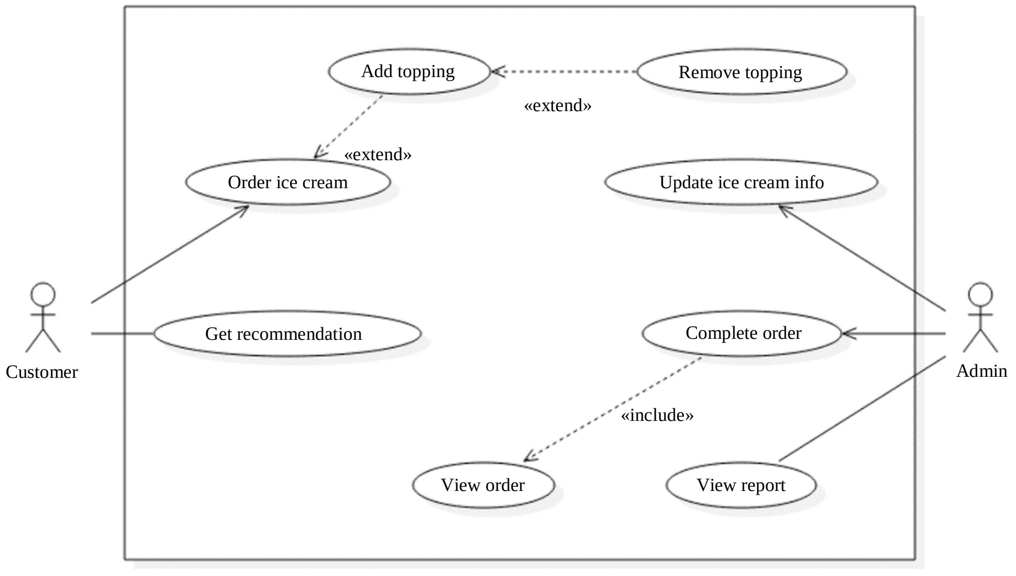


Fig. 1: Use case diagram

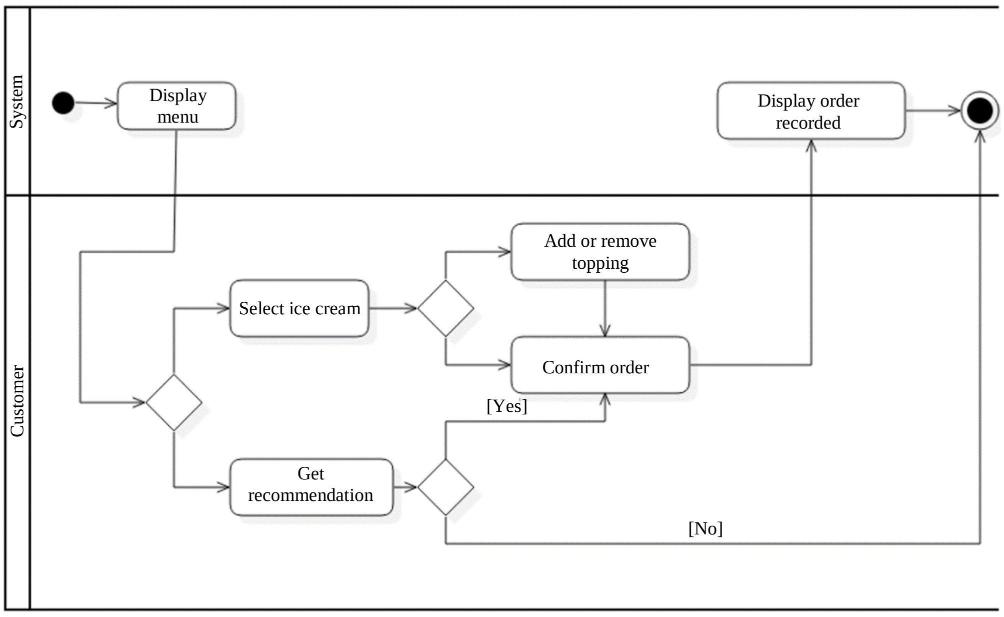


Fig. 2: Activity diagram - customer

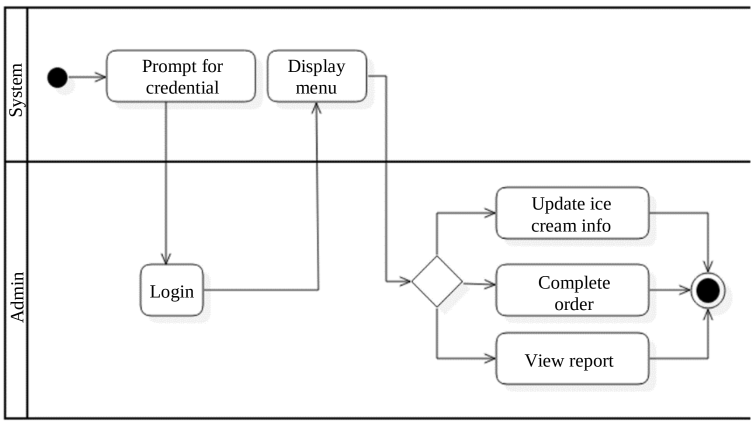
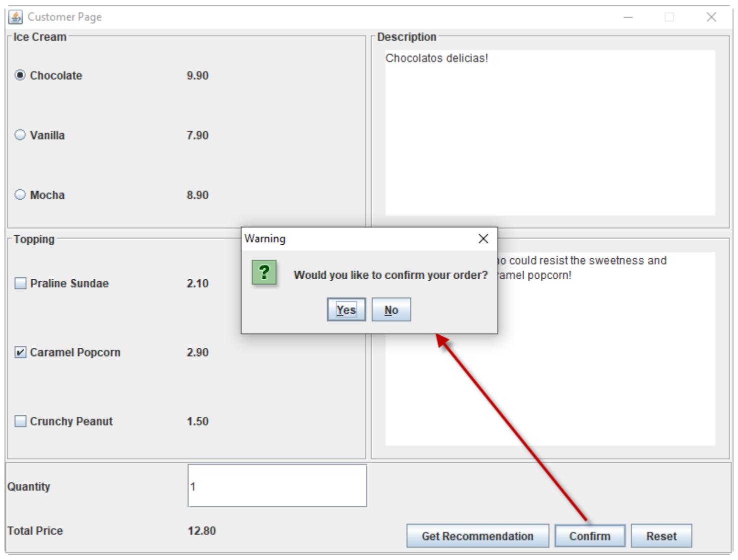


Fig. 3: Activity diagram - admin



Ice Cream

[✓ Chocolate]	9.90
Vanilla	7.90
Mocha	8.90

Description

Chocolatos delicias!

Topping

Praline Sundae	2.10
[✓ Caramel Popcorn]	2.90
Crunchy Peanut	1.50

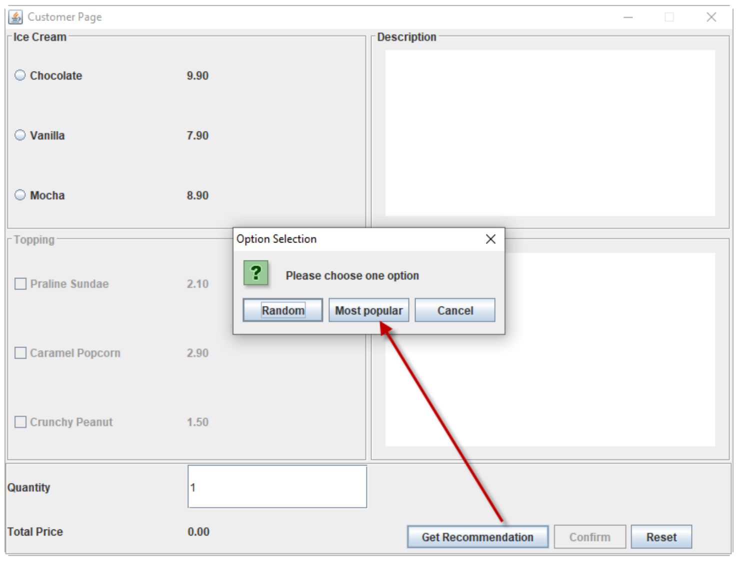
o could resist the sweetness and
amel popcorn!

Would you like to confirm your order?

Quantity 1

Total Price 12.80

Fig. 4: Customer - order ice cream



Ice Cream

Chocolate	9.90
Vanilla	7.90
Mocha	8.90

Description

Topping

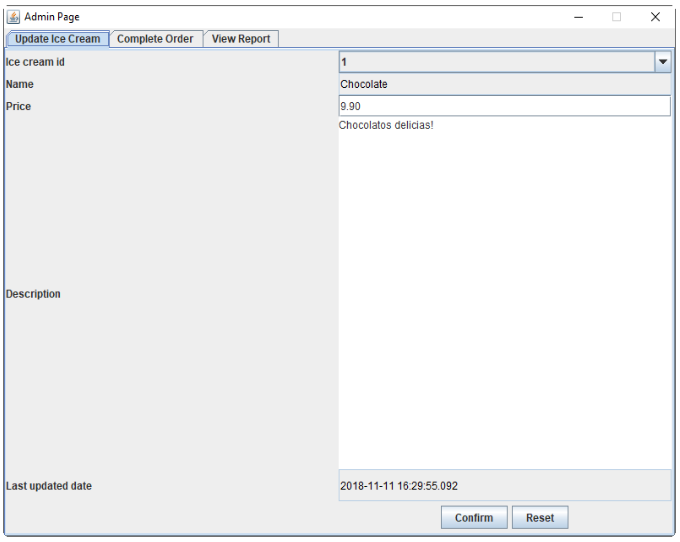
Praline Sundae	2.10
Caramel Popcorn	2.90
Crunchy Peanut	1.50

Please choose one option

Quantity 1

Total Price 0.00

Fig. 5: Customer - get recommendation


Admin Page

Update Ice Cream | Complete Order | View Report

Ice cream id	1
Name	Chocolate
Price	9.90
Description	Chocolatos delicias!
Last updated date	2018-11-11 16:29:55.092

Confirm Reset

Fig. 6: Admin - update ice cream info

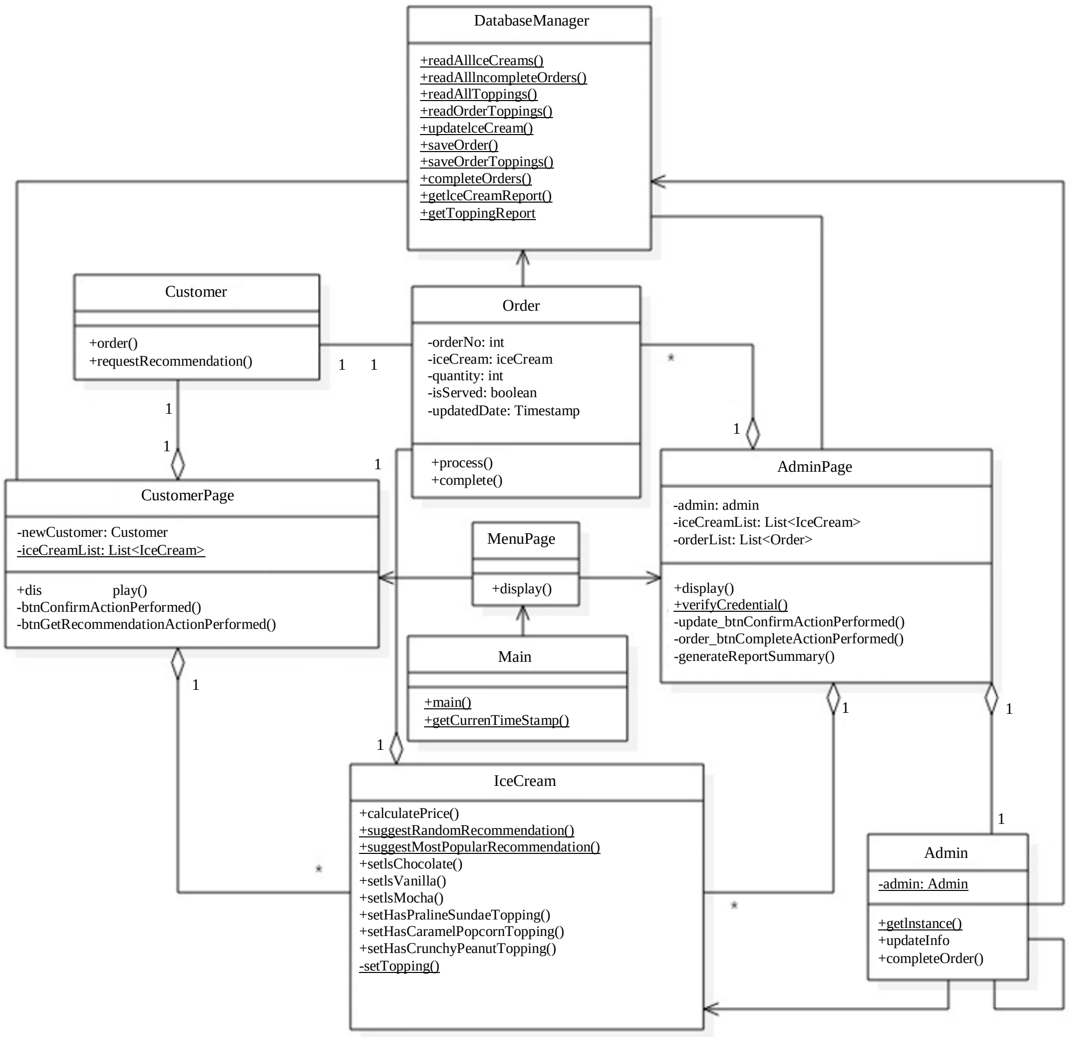


Fig. 7: Class diagram - initial design

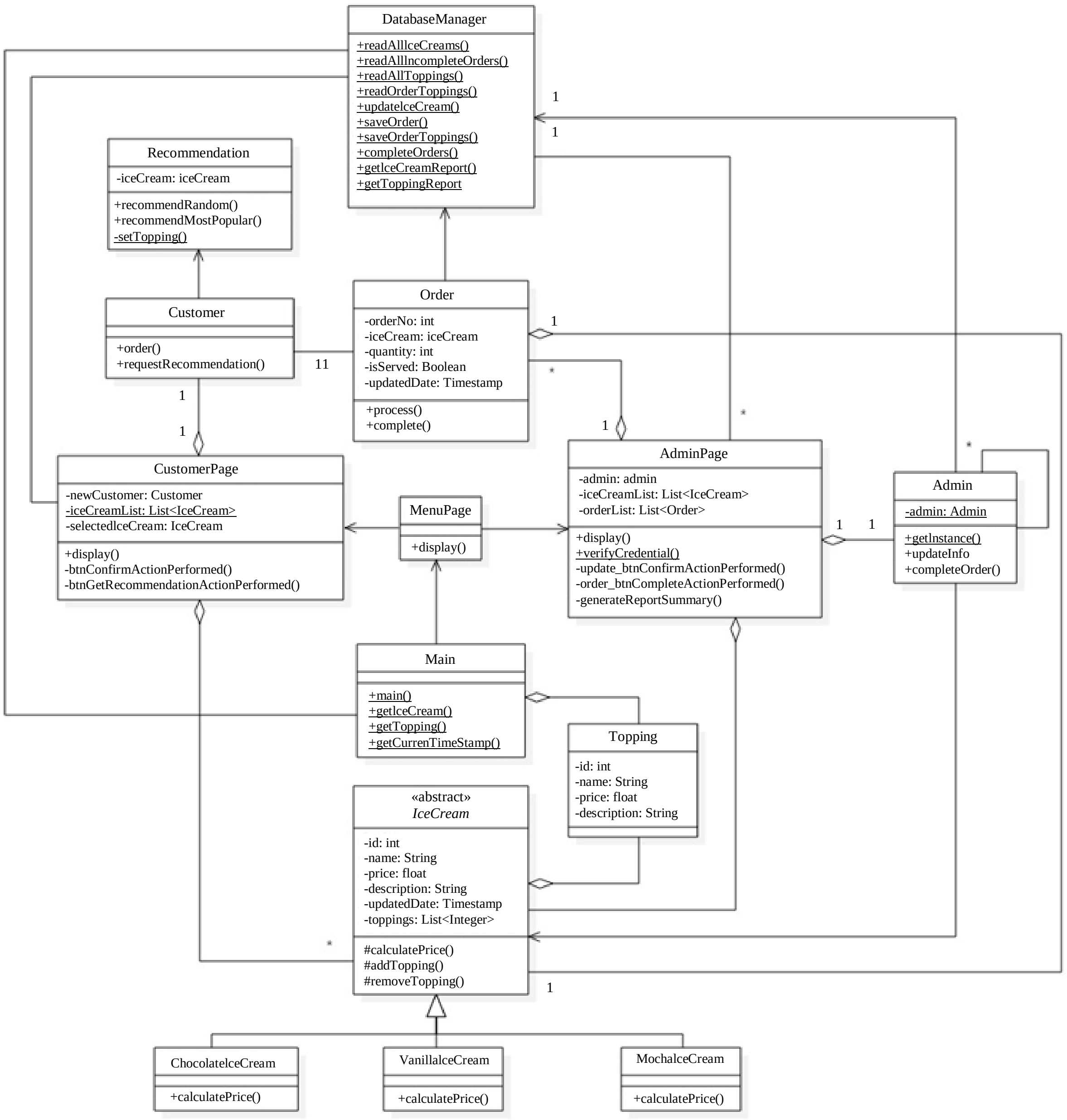


Fig. 8: Class diagram - after applying design principles

Using Simpler Solution

Figure 7 shows the initial design of PICOS system using Class diagram. Hereafter, all class name and method name will be written in italic. There is a total of 9 classes where the main classes are *IceCream*, *Customer*, *Order*, *Admin* and *DatabaseManager*. The remaining classes are GUI related and the main point of execution of the system.

After Applying Design Principles

Figure 8 shows the refined Class diagram after applying two SOLID design principles such as SRP and OCP. SRP is added by separating recommendation and topping from *IceCream* class. This resulting in more classes but offer greater flexibility as certain changes will only be applied to the right class. For instance, changes to the implementation of recommendation will only impact *Recommendation* class. On the other hand, OCP results in separating each type of ice cream into a child class of the parent *IceCream* class. This design promotes reusability and extensibility as the child class can reuse its parent attributes and methods. Extensibility in the sense that if a new type of ice cream is to be added, a new child class extending *IceCream* class can be added without affecting existing implementation.

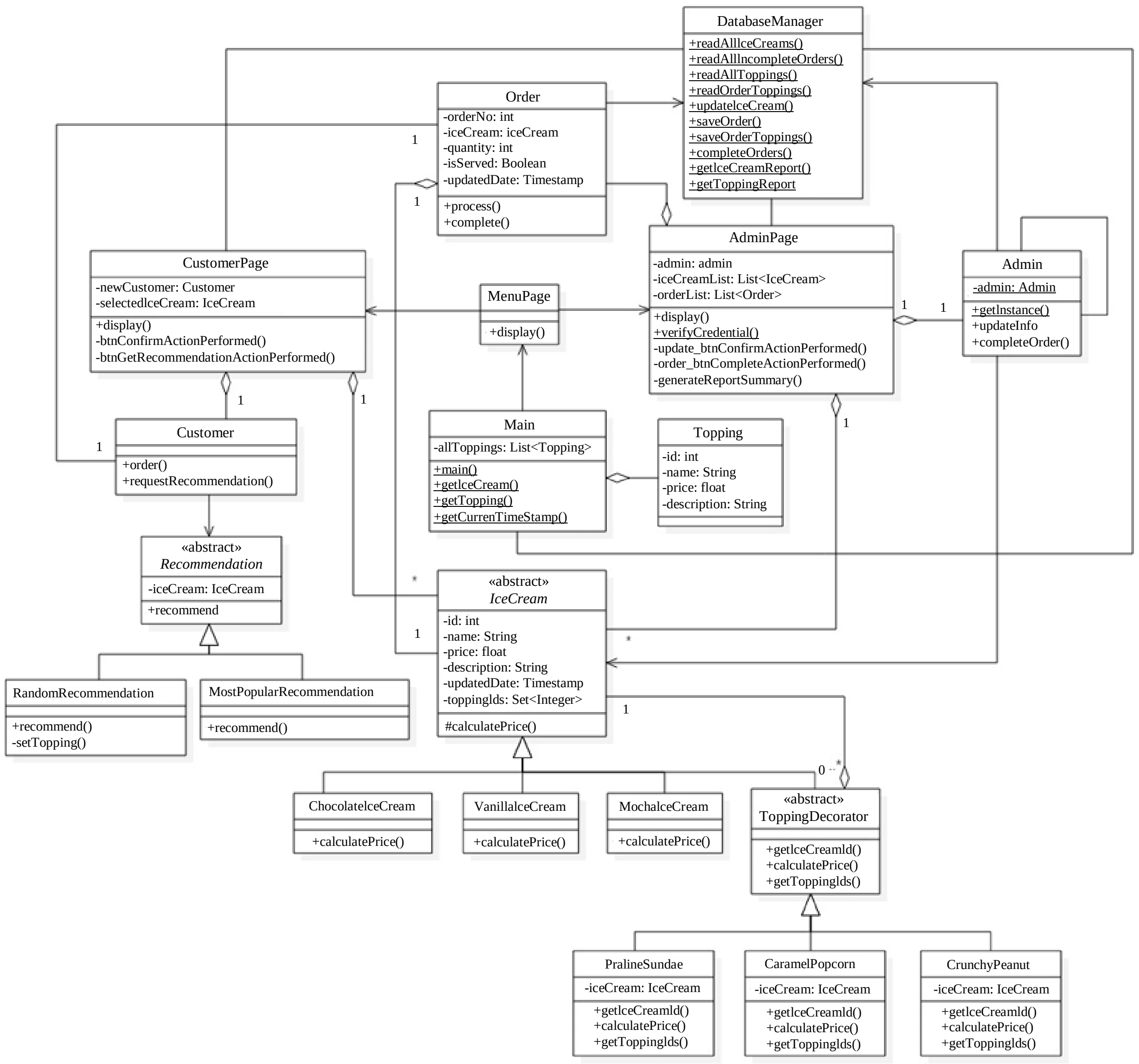


Fig. 9: Class diagram - after applying design patterns

After Applying Design Patterns

Two design patterns (i.e., Strategy and Decorator) are applied to the earlier design which can be seen in Figure 9. The former is applied to Recommendation class where each type of recommendation is now made as derived class of the super class. This results in more classes but offers greater flexibility in which new type of recommendation can be added without affecting existing implementation. On the other hand, decorator pattern is applied on the scenario where ice cream can have topping(s). This pattern is very suitable as it provides additional features on top of having ice cream. If topping is no longer required, the entire decorator can be removed without impacting ice cream related classes.

System Screenshots

Figure 4 shows the user interface of Order ice cream feature. It allows customer to order ice cream with or without toppings. Figure 5 shows the get recommendation feature which automatically selects the ice cream for the user. Figure 6 displays the Admin page where there are three tabs representing three functionalities. The first tab is to update ice cream info. The next tabs are to complete order and view report respectively.

Analysis of the Results

FXP level varies depending on the software manipulators (e.g., users, developers, etc.) where it can be categorized into Low-level User (LU), High-level User (HU) and Developer-level User (DU) (Shen and Ren, 2006). In PICOS system, the type of changes cannot be made by user, therefore all sort of changes requires developer involvement, thus limiting FXP level to DU. This kind of changes normally involves code change or reconfiguration (Shen and Ren, 2006). In addition, FXP based on DU can be categorized into three levels based on the technical complexity such as DUFXP1, DUFXP2 and DUFXP3 from low to high respectively (Shen and Ren, 2006). The flexible force for each level can be seen in Table 2.

FXP Identification

Table 3 shows the possible requirement changes to PICOS system which is the identification of the FXP. All the predicted changes refer to adding new type of item for ice cream, topping and recommendation. Therefore, the flexibility calculation will be based on these three changes.

Flexible Distance Calculation

The size of software change caused by a FXP is known as the flexible distance which can be measured using function points (Shen and Ren, 2006). Flexible distance calculation is based on Unadjusted Function Points (UFP) as the main consideration is about the change size (Shen and Ren, 2006). Function Point Analysis (FPA) is very popular in measuring the productivity of software (Vickers, 2014). There are five required components to calculate the function points such as *"External Input (EI), External Output (EO), External Enquiries (EQ), Internal Logical File (ILF) and External Interface File (EIF)"* (Vickers, 2014). Then, the sum of these five components form the UFP.

Table 4 provides the justification in determining the type and complexity. As identified from the source codes, three classes (IceCream, Order and CustomerPage) need to be changed upon having the requirement to add new ice cream. Furthermore, the function complexity and weightage for each category to calculate the total UFP are taken from (de Freitas Junior et al., 2015). There will be three sets of calculation such as for the simpler solution, after applying design principles and after applying design patterns.

Table 5 shows the impact of the code changes required to add new type of ice cream where File Type Reference (FTR) = 3 and Data Element Type (DET) = 11. As a result, the complexity of this EI is high. Whereas for ILF, the Record Element Type (RET) = 1 (ice cream table in database) and DET = 1 (insert new record), thus the complexity is low. Similarly, EQ has 2 FTR (*CustomerPage* and *DatabaseManager* class) and 2 DET (read records from ice cream table in database and convert result set to ice cream object). As a result, EQ has low complexity. EO has only 1 FTR which is CustomerPage class and 3 DET (displays radio button, price and description), thus EO complexity is low. The remaining calculation hereafter works the same way which can be seen in Table 6 to 13.

The add code extension in Table 8 basically add a new class extending the abstract *IceCream* class as well as adding new radio button and its event to CustomerPage class. Thus, FTR = 2 and DET = 3, resulting in low complexity for the EI. The same goes to Table 12 and 13. After applying design patterns, add new type of topping has FTR = 2 and DET = 3, while add new type of recommendation has FTR = 2 and DET = 2. Both results in low complexity for the EI.

Table 2: FXP level force value (Shen and Ren, 2006)

Flexible point level	Flexible force value	Manipulation
SAFXP	0	No need user's manipulation
LUFXP	10	Simple function manipulation
HUFXP	20	Complex function and business manipulation
DUFXP1	30	Low technical manipulation
DUFXP2	40	Average technical manipulation
DUFXP3	50	High technical manipulation

Table 3: Possible changes to PICOS

NO	Change requirement
1	Add new type of ice cream for order
2	Add new type of topping for order
3	Add new type of ice cream recommendation

Table 4: Function point components - simpler solution - add new type of ice cream

Activity	Type	Complexity	UFP
Add code changes to *IceCream, Order and CustomerPage* class	EI	High	6
Add new record to ice cream table	ILF	Low	7
Read record from database	EQ	Low	3
Display new ice cream info	EO	Low	4
Total			20

Table 5: EI complexity justification

Class name	Changes	DET
IceCream	Add new Boolean attribute, getter method and setter method. Modify calculatePrice() method, suggestRandomRecommendation(), setTopping() method.	6
Order	Modify *process()* method.	1
CustomerPage	Add new radio button and its event. Modify initComponents(), btnConfirmActionPerformed() method.	4

Table 6: Function point components - simpler solution - add new type of topping

Activity	Type	Complexity	UFP
Add code changes to *IceCream, Order, CustomerPage, AdminPage* class	EI	High	6
Add new record to topping table	ILF	Low	7
Read record from database	EQ	Low	3
Display new topping info	EO	Low	4
Total			20

Table 7: Function Point Components - Simpler Solution - Add New Type of Recommendation

Activity	Type	Complexity	UFP
Add code changes to *IceCream, Customer, CustomerPage* class	EI	High	6
Display recommended ice cream with topping(s) if any	EO	Low	4
Total			10

Table 8: Function point components - after applying design principles - add new type of ice cream

Activity	Type	Complexity	UFP
Add code extension	EI	Low	3
Add new record to ice cream table	ILF	Low	7
Read record from database	EQ	Low	3
Display new ice cream info	EO	Low	4
Total			17

Table 9: Function point components - after applying design principles - add new type of topping

Activity	Type	Complexity	UFP
Add code changes to *CustomerPage, AdminPage* class	EI	Medium	4
Add new record to topping table	ILF	Low	7
Read record from database	EQ	Low	3
Display new topping info	EO	Low	4
Total			18

Table 10: Function point components - after applying design principles - add new type of recommendation

Activity	Type	Complexity	UFP
Add code changes to *Recommendation, Customer, CustomerPage* class	EI	Medium	4
Display recommended ice cream with topping(s) if any	EO	Medium	4
Total			8

Table 11: Function point components - after applying design patterns - add new type of ice cream

Activity	Type	Complexity	UFP
Add code extension	EI	Low	3
Add new record to ice cream table	ILF	Low	7
Read record from database	EQ	Low	3
Display new ice cream info	EO	Low	4
Total			17

Table 12: Function point components - after applying design patterns - add new type of topping

Activity	Type	Complexity	UFP
Add code extension	EI	Low	3
Add new record to topping table	ILF	Low	7
Read record from database	EQ	Low	3
Display new topping info	EO	Low	4
Total			17

Table 13: Function point components - after applying design patterns - add new type of recommendation

Activity	Type	Complexity	UFP
Add code extension	EI	Low	3
Display recommended ice cream with topping(s) if any	EO	Low	4
Total			7

Table 14: Flexible degree calculation - simpler solution

Requirement	K_i calculation	Result
Add new type of ice cream	20/(1+50)	0.39
Add new type of topping	20/(1+50)	0.39
Add new type of recommendation	10/(1+50)	0.20

Table 15: Flexible degree calculation – after applying design principles

Requirement	K_i Calculation	Result
Add new type of ice cream	17/(1+30)	0.55
Add new type of topping	18/(1+40)	0.44
Add new type of recommendation	8/(1+40)	0.20

Flexible Force Calculation

The minimum external force required to cause software change in a FXP is known as the flexible force (Shen and Ren, 2006). The implementation using simpler solution requires high technical manipulation as the developer needs to understand the existing structure of the codes and make changes accordingly. On the other hand, after applying design principles, the implementation effort to make code changes can be reduced. Finally, applying design patterns mostly reduce implementation effort by extending the codes instead of modifying it. Therefore, the flexible force for each implementation varies depending on its technical manipulation. The flexible point level for Developer-level User (DU) can be categorized into three such as DUFXP1, DUFXP2 and DUFXP3 in which it represents low, average and high technical manipulation respectively (Shen and Ren, 2006). Therefore, the flexible force for each implementation respectively will be 50 for simpler solution as it requires higher technical knowledge. After applying design principles, the flexible force is 30 for the first FXP and 40 for second and third FXP. Lastly, after applying design patterns, most changes are to add code extension. Thus, only low technical manipulation required which gives a flexible force of 30 (Shen and Ren, 2006).

Flexible Degree Calculation

Table 14 to 16 shows the flexible degree calculation for the simpler solution, after applying design principles and after applying design patterns respectively. The values calculated are the used to calculate the flexible capacity.

Flexible Capacity Calculation

Table 17 provides the flexible capacity calculation for the three implementations.

Table 16: Flexible degree calculation-after applying design patterns

Requirement	K_i calculation	Result
Add new type of ice cream	17/(1+30)	0.55
Add new type of topping	17/(1+30)	0.55
Add new type of recommendation	7/(1+30)	0.23

Table 17: Flexible capacity calculation

Implementation	Flexible capacity calculation	Result
Simpler solution	0.39+0.39+0.20	0.98
After applying design principles	0.55+0.44+0.20	1.19
After applying design patterns	0.55+0.55+0.23	1.33

Discussion

It can be clearly seen in Figure 10 and 11 that both design principles and design patterns have improved the software flexibility of PICOS system. It has reduced a lot of implementation effort to implement the same type of changes which is shown by the degree of flexibility. After applying the chosen design principles (i.e., SRP and OCP), it is much easier to add new ice cream and new topping to the system, thus making the system more flexible to such changes. The system design has become more reusable and extensible as compared to its initial design. However, for add new recommendation, there is just slight difference where its flexible degree value is 0.2 after converted to the nearest 2 decimal points.

On the other hand, the chosen design patterns (i.e., Strategy and Decorator) are only applied to add topping and recommendation requirements. This is the reason behind it has the same flexibility degree with its previous two implementations. However, it boosts the degree of flexibility to the other two flexible points which proves that the chosen patterns have significantly improved the flexibility of the system. This experiment has proven that the use of the chosen design principles (i.e., SRP and OCP) and patterns (i.e. Strategy and Decorator) has positive impact on software flexibility for application developed using OOP. Therefore, the use of design principles and patterns is highly recommended to produce highly flexible object-oriented software.

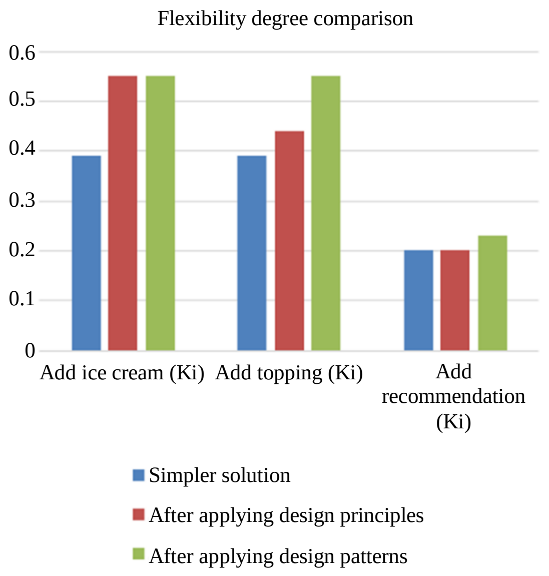


Fig. 10: Flexibility degree comparison

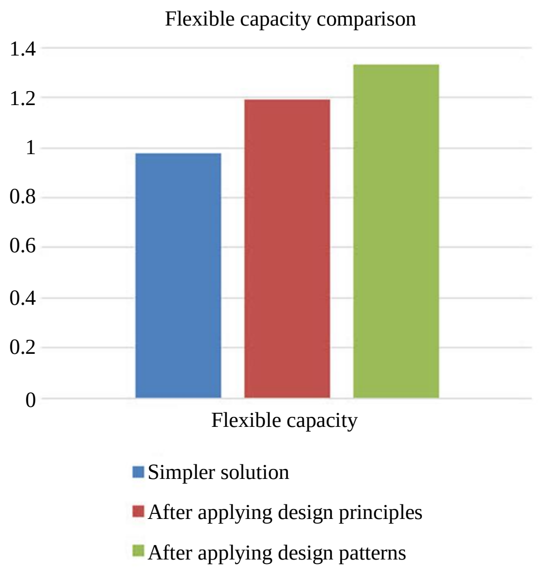


Fig. 11: Flexible capacity comparison

Conclusion

The main contribution of this research is to conduct an experiment to prove that the use of design principles and patterns can improve software flexibility of an application developed using OOP. Applying the chosen principles (i.e., SRP and OCP) and patterns (i.e., Strategy and Decorator) on the application developed have made the software design much more flexible to changes which can be seen from the results of this study. Changes can be added as extension rather than modifying existing components. This will significantly reduce the cost, time and effort to implement the changes. It will also ease maintenance in the long run. Therefore, the authors highly recommend the use of design principles and patterns in object-oriented software development. For future works, the authors would like to recommend researchers who are interested in this topic to conduct these experiments on programming paradigm other than OOP as well as utilizing different design principles and patterns used in this research to measure software flexibility or other quality attribute. In addition, similar experiments could also be made for different quality attributes.

Author’s Contributions

Muhammad Ehsan Rana: Conducting Research and revised outcomes and documentation.

Eddy Khonica: Conducting experiment to measure flexibility improvement using Flexible Point (FXP) based on the chosen design principles and patterns.

Wan Nurhayati Wan Ab. Rahman: Supervising the research by providing suggestions as well as continuously reviewing the manuscript.

Masrah Azrifah Azmi Murad: Reviewing and correcting the final manuscript.

Rodziah Binti Atan: Reviewing and correcting the final manuscript.

Ethics

The corresponding author confirms that all the other authors have read and approved the manuscript and no ethical issues involved.

References

Abdullah, D., Khan, M. H., & Srivastava, R. (2015). Flexibility: A Key Factor to Testability. International Journal of Software Engineering & Applications (IJSEA), 6(1).

Arcelli, D., & Di Pompeo, D. (2017, January). Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach. In ANT/SEIT (pp. 521-528).

Bräuer, J., Plösch, R., Saft, M., & Körner, C. (2018). Measuring object-oriented design principles: The results of focus group-based research. Journal of Systems and Software, 140, 74-90.

de Freitas Junior, M., Fantinato, M., & Sun, V. (2015). Improvements to the function point analysis method: A systematic literature review. IEEE Transactions on Engineering Management, 62(4), 495-506.

Eden, A. H., & Mens, T. (2006). Measuring software flexibility. IEE Proceedings-Software, 153(3), 113-125.

Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. Pearson Education India.

Gorton, I. (2011). Software quality attributes. In Essential Software Architecture (pp. 23-38). Springer, Berlin, Heidelberg.

Gravino, C., & Risi, M. (2017, August). How the use of design patterns affects the quality of software systems: a preliminary investigation. In 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 274-277). IEEE.

Holzner, S. (2006). Design patterns for dummies. John Wiley & Sons.

IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology. Office, 121990(1).

Khomh, F., & Guéhéneuc, Y. G. (2018, March). Design patterns impact on software quality: Where are the theories?. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 15-25). IEEE.

Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., & Sharbaf, M. (2018). A survey of model transformation design patterns in practice. Journal of Systems and Software, 140, 48-73.

Lassing, N., Rijsenbrij, D., & Vliet, H. (1999). The goal of software architecture analysis: Confidence building or risk assessment. In Proceedings of First BeNeLux conference on software architecture (pp. 47-57).

Madasu, V. K., Venna, T. V. S. N., & Eltaieb, T. (2015). SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP. Journal of Multidisciplinary Engineering Science and Technology, 2(2), 1–3.

Marouane, H., Duvallet, C., Makni, A., Bouaziz, R., & Sadeg, B. (2018). An UML profile for representing real-time design patterns. Journal of King Saud University-Computer and Information Sciences, 30(4), 478-497.

Martin, R. C., & Martin, M. (2006). Agile principles, patterns and practices in C# (Robert C. Martin). Prentice Hall PTR.

Nelson, K. M., Nelson, H. J., & Ghods, M. (1997, January). Technology flexibility: conceptualization, validation and measurement. In Proceedings of the Thirtieth Hawaii International Conference on System Sciences (Vol. 3, pp. 76-87). IEEE.

Niu, J., Shen, L., & Zheng, Q. (2011, August). A Measurement Method of Software Flexibility Based on SOCP. In International Conference on Computer Science, Environment, Ecoinformatics and Education (pp. 260-266). Springer, Berlin, Heidelberg.

Nurdiani, I., Börstler, J., & Fricker, S. A. (2018). Literature review of flexibility attributes: A flexibility framework for software developing organization. Journal of Software: Evolution and Process, 30(9), e1937.

Oruc, M., Akal, F., & Sever, H. (2016, April). Detecting design patterns in object-oriented design models by using a graph mining approach. In 2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT) (pp. 115-121). IEEE.

Peng, S., Shen, L., Liu, H., & Li, F. (2009, March). User-oriented measurement of software flexibility. In 2009 WRI World Congress on Computer Science and Information Engineering (Vol. 7, pp. 629-633). IEEE.

Scanniello, G., Gravino, C., Risi, M., Tortora, G., & Dodero, G. (2015). Documenting design-pattern instances: A family of experiments on source-code comprehensibility. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(3), 1-35.

Shen, L., & Ren, S. (2006). Analysis and measurement of software flexibility based on flexible points. Published in the Proceedings of Smef–2006.

Subramaniam, H., & Zulzalil, H. (2012). Software quality assessment using flexibility: A systematic literature review. International Review on Computers and Software, 7(5).

Thabasum, S. S., & Sundar, M. (2012). A survey on software design pattern tools for pattern selection and implementation. International Journal of Computer Science & Communication Networks (IJCSCN).

Vickers, P. (2014). An introduction to function point analysis. Northumbria University. https://pdfs.semanticscholar.org/5bed/ab771d97252 9bab804bca0a5ce88512df102.pdf

Walter, B., & Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. Information and Software Technology, 74, 127-142.

Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., & Chen, J. (2018). Efficiently detecting structural design pattern instances based on ordered sequences. Journal of Systems and Software, 142, 35-56.

Zhang, C., Wang, F., Xu, R., Li, X., & Yang, Y. (2014, May). A quantitative analysis of survey data for software design patterns. In Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies (pp. 48-55).

Zhao, L. (1998). Intelligent agents for flexible workflow systems. AMCIS 1998 Proceedings, 82.

Zhu, H., & Bayley, I. (2015). On the composability of design patterns. IEEE Transactions on Software Engineering, 41(11), 1138-1152.