Impact of Abstract Factory and Decorator Design Patterns on Software Maintainability: Empirical Evaluation using CK Metrics

Aisha Kurmangali
School of Computing
Asia Pacific University of Technology and Innovation
Kuala Lumpur, Malaysia
aisha.kurmangali@gmail.com

Muhammad Ehsan Rana
School of Computing
Asia Pacific University of Technology and Innovation
Kuala Lumpur, Malaysia
muhd_ehsanrana@apu.edu.my

Wan Nurhayati Wan Ab Rahman
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
Selangor, Malaysia
wnurhayati@upm.edu.my

***Abstract*— Gang of Four (GoF) design patterns are the famous set of twenty-three reusable and robust solutions to certain established problems in object-oriented programming that still guides software developers and designers despite having been published in mid-90's. However, it is relatively difficult to explicitly determine the overall appropriateness and suitability of a design pattern in a software system. Thus, this work attempts to select one attribute from the McCall's software quality model – maintainability, and empirically evaluate it before and after applying selected design patterns (Abstract Factory and Decorator) within a relatively simple problem scenario to keep the focus on and prove the positive effect of design patterns in object-oriented software solutions in terms of their maintainability.**

Keywords*— Design Patterns, Software Measurement, Object Oriented Design, Software Performance Evaluation, Software Maintenance, Software Metrics, Software Quality

I. INTRODUCTION

A part of McCall's software quality model, maintainability is defined as the "effort required to locate and fix an error in an operational program" [1]. There are four types of software maintenance [2]:

- Corrective – related to error or bug discovery and correction
- Adaptive – related to any future change anticipation and adaptation, like the enhancement of technologies used
- Perfective – related to the reaction of requirements changes, like improvement of features or maintenance of an elevated number of users
- Preventive – related to adopting already established methods for bug prevention and quality augmentation

As argued by [2], maintainability is a quality attribute that has utmost importance in every system and should always be accounted for from the beginning, due to the following two major reasons. It is extremely consequential not only in software terms but in business too – maintenance staff and costs are important business factors. Moreover, maintainability has a way of facilitating other software attributes like security or performance. Thus, in order to test one approach to improving such an important quality factor, such as applying design patterns, this study will use a fictional scenario to demonstrate their influence. Furthermore, the problem scenario will be solved with UML class diagrams and Java implementations, both with and without design patterns to empirically prove their effect on maintainability.

II. EMPIRICAL EVALUATION

To demonstrate the impact of design patterns on software maintainability, a simple scenario was derived. Firstly, a simple solution without any design patterns applied would be designed and implemented in Java. Then, design patterns like Abstract Factory and Decorator will be applied in attempt to improve the first solution's maintainability. It should be noted that the application of the design patterns was focused on the object-oriented aspect of the system and its main classes – the class Client is merely there for user input and output.

A. Scenario

A car manufacturing company requires a system for its customers to choose from the various car types available, as well as customize their appearance. The user would be able to choose a car that runs either on electricity or fuel, as well as select a paint finish for it: solid (default), matte, and metallic.

B. Design Refinements

1) Simple Solution

This simple solution uses inheritance and creates several classes, each for every possible combination between car types and paint finishes, depending on what the customer wants. This approach has several maintainability issues. For example, what if the company adds on hybrid types of cars? Or pearlescent paint finish? That would imply the creation of classes like HybridCarSolidFinish.java or FuelCarPearlescentFinish.java. The maintenance of so many classes is extremely complicated, and this solution is not feasible in the long run. Moreover, each such class would handle both the functionality of car types and paint finishes, which is not a good practice either. Fig. 1 represents simple solution using UML class diagram.

Thus, Abstract Factory would handle the distinction between choosing the car type and choosing the paint finish. Meanwhile, Decorator would have a slightly different approach – the customer would first choose the type of car, and then how to "decorate" it. Many manufacturers provide solid finish by default and sometimes free, so that's how the car would be created first. Then the customer would be given a choice to apply a matte or a metallic finish, and even change their mind – with Decorator design pattern, the car can be refinished multiple times.

978-1-6654-9501-1/22/$31.00 ©2022 IEEE

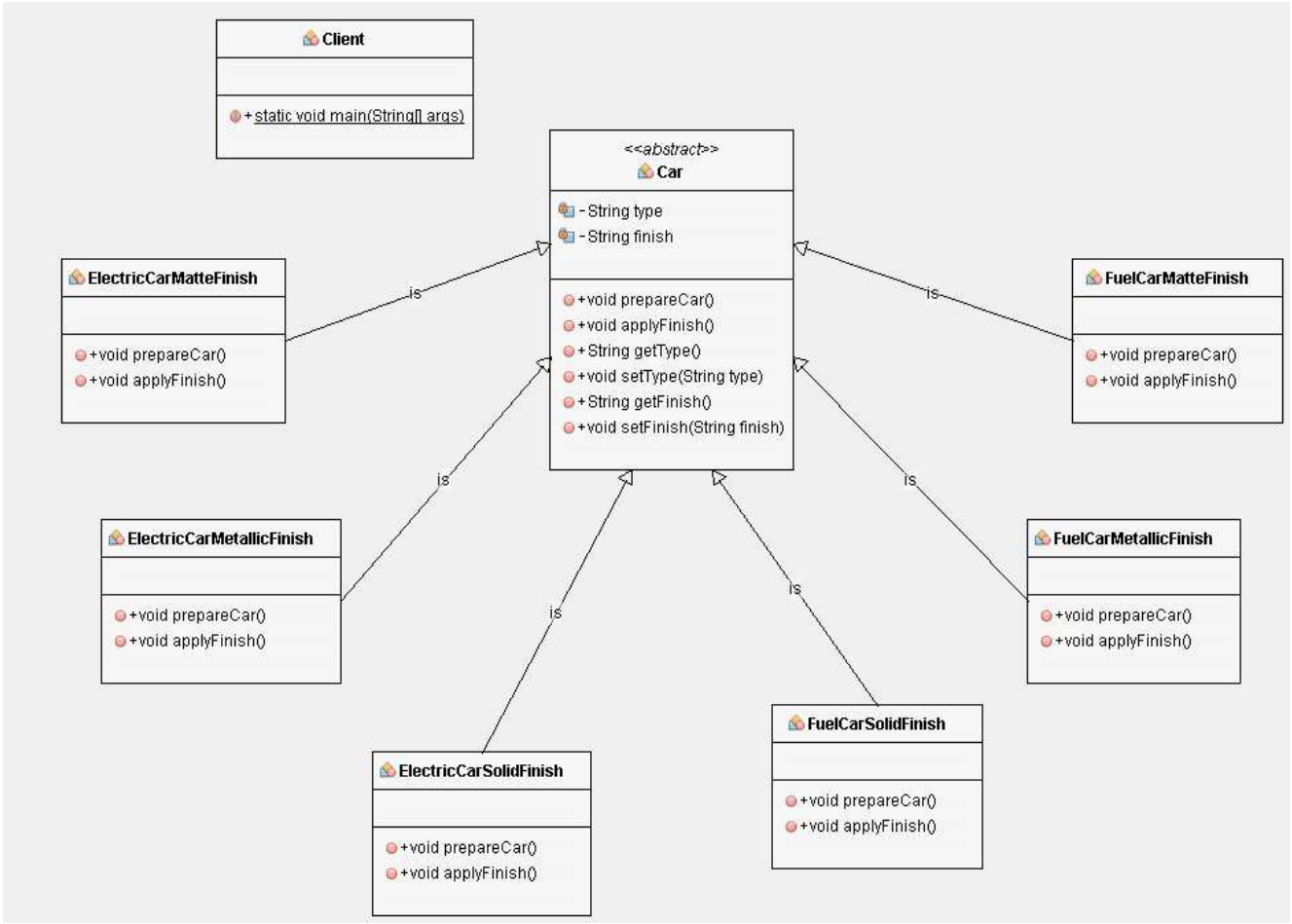


Fig. 1: Simple solution UML class diagram

2) Abstract Factory based Solution

Abstract Factory is an object creational design pattern, a step-up from a Factory Method, inside the family of 23 Gang of Four (GoF) design patterns [3]. This approach allows an encapsulated creation of objects that are related to each other and are within common families. This way, the client application would be shielded from creating a concrete instance, the factory responsible for that specific product family would do it instead. As its name suggests, this pattern brings on an additional level of abstraction to a system, which could be good for its maintainability. Moreover, it ensures a clean and orderly code with the fulfilment of open-closed and single responsibility principles [4]. Fig. 2 represents Abstract Factory based solution using UML class diagram.

In the case of the given scenario, this pattern is applicable due to there being two possible groups or families – types of cars and types of paint finishes. Thus, since the exact combination of the final car is not known upfront and solely depends on the user's choice, Abstract Factory is a good way to solve the problem scenario. The client code would only provide the selected options, and the correct factory would be called and handle the creation of a car. Additionally, if the number of combinations rises (e.g., the car company introduces a pearlescent paint finish), then there would be an established, consistent, and maintainable way of adding and managing new products – though, admittedly, it would be a trade-off in complexity, as with many design patterns.

In the case of the given scenario, this pattern is applicable due to there being two possible groups or families – types of cars and types of paint finishes. Thus, since the exact combination of the final car is not known upfront and solely depends on the user's choice, Abstract Factory is a good way to solve the problem scenario. The client code would only provide the selected options, and the correct factory would be called and handle the creation of a car. Additionally, if the number of combinations rises (e.g., the car company introduces a pearlescent paint finish), then there would be an established, consistent, and maintainable way of adding and managing new products – though, admittedly, it would be a trade-off in complexity, as with many design patterns. Hence, following is the mapping of all Abstract Factory requisites to this specific scenario:

- Product Family One – Car types:
  - Abstract Product 1 – Car.java
  - Concrete Product 1 – ElectricCar.java
  - Concrete Product 2 – FuelCar.java
- Product Family Two – Paint finishes:
  - Abstract Product 2 – Finish.java
  - Concrete Product 1 – SolidFinish.java
  - Concrete Product 2 – MatteFinish.java
  - Concrete Product 3 – MetallicFinish.java

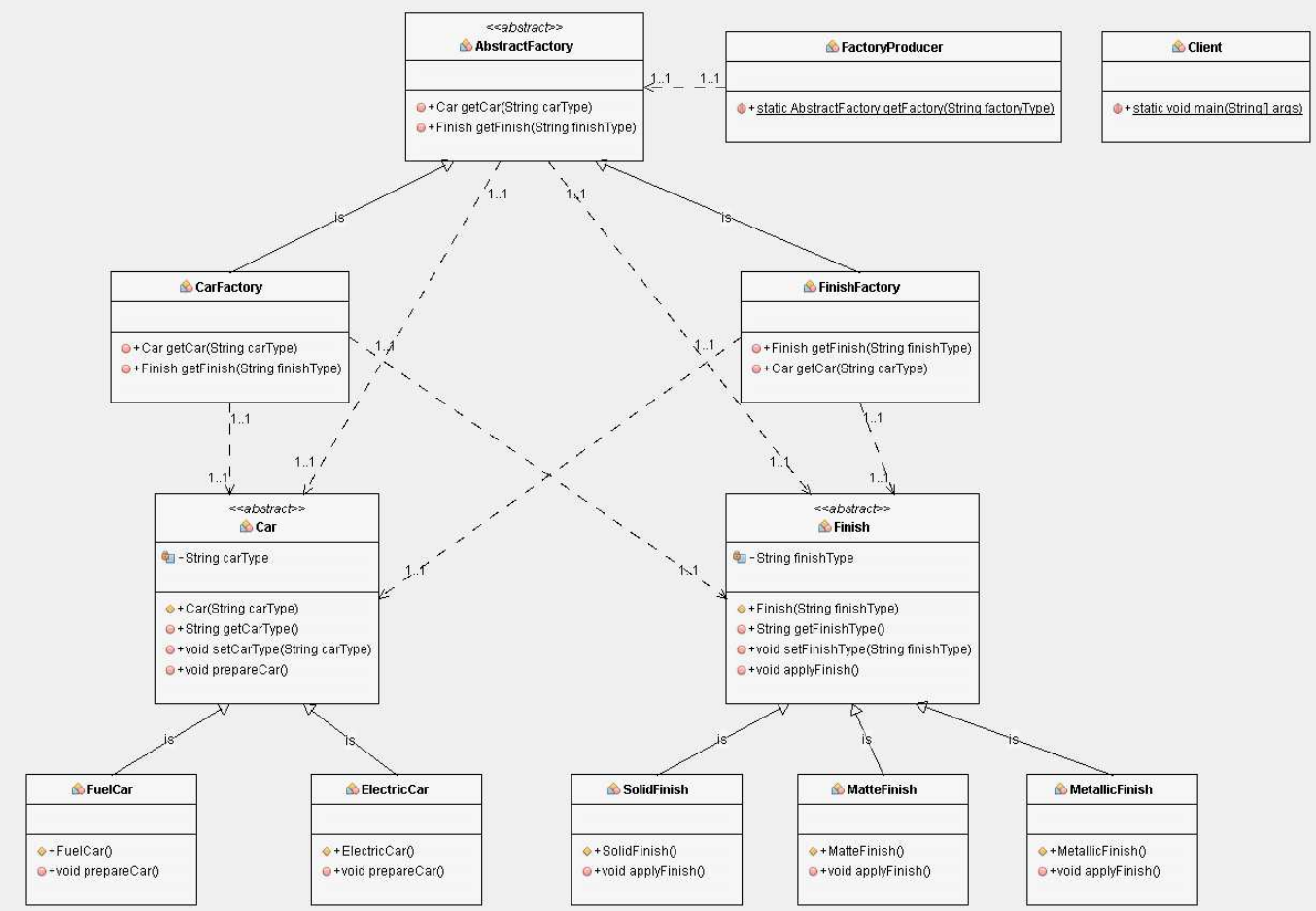


Fig. 2: Abstract Factory solution UML class diagram

- Abstract Factory – AbstractFactory.java
    - Concrete Factory 1 – CarFactory.java
    - Concrete Factory 2 – FinishFactory.java
- The selection of Product Family – FactoryProducer.java and Client.java

3) Decorator based Solution

Unlike its counterpart, Decorator or Wrapper design pattern is of structural nature [3]. It permits "attach additional functionalities to an object dynamically" [5] and makes the extension of features or behavior more maintainable than inheritance. To compare, the simple solution had six classes just to represent each car variation, which could turn into much more if the company changes their requirements and new options come in – "class explosion" [6]. The instances of base classes like ElectricCar.java and FuelCar.java would be open for extension and closed to modification. Also, classes like MatteFinishDecorator.java would only handle one task – applying matte paint finish. With Decorator, the customer would have more flexibility to add on paint finishes to their car, or even redo it after. Moreover, the finishes would be applied only when needed, meaning the system wouldn't necessarily have to foresee every variation, just add another decorator like PearlescentFinishDecorator.java. Thus, this approach brings on more flexibility and extendibility, which influences maintainability. Fig. 3 represents Decorator based solution using UML class diagram. The following is the mapping of Decorator design pattern elements to the problem scenario:

- Abstract Component – Car.java (interface)
- Concrete Components – ElectricCar.java, FuelCar.java
- Base Decorator – CarDecorator.java (applying solid finish to every car by default)
- Concrete Decorators – MatteFinishDecorator.java, MetallicFinishDecorator.java (customers will apply other additional finishes, if they want)

C. Measuring Technique for Maintainability Calculation

A previous research was carried out by applying Maintainability Estimation Model (MEM) based on QMOOD (Quality Model for Object-Oriented Design) metrics to see the impact of State and Proxy design patterns on maintainability [7]. Moreover, another study [8] indicate the use of a formula-based technique to measure design patterns impact on software maintainability. However, several academic articles and works analysed and indicated the effect of Chidamber and Kemerer object-oriented metrics (CK metrics) developed in 1994 on overall software quality, and on its maintainability in specific [9] [10] [11] [12] [13]. Thus, the proposed empirical evaluation criteria for maintainability will be the CK metrics, evaluated using the open-source tool called CKJM - Chidamber and Kemerer Java Metrics [14] [15]. It evaluates the java output files with the extension of *.class, as depicted in Fig. 4. The author of this tool developed it considering the complex nature of the existing open-source tools to provide an easy to use and simple enough tool to measure only the CK metrics with no

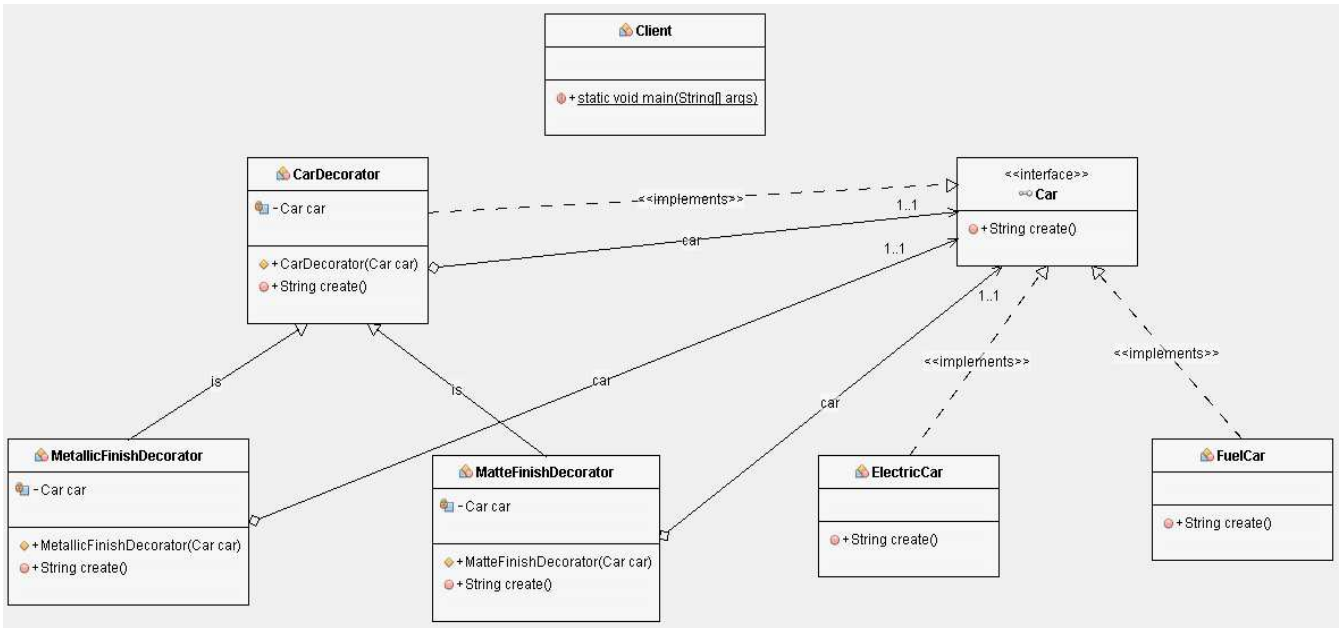


Fig. 3: Decorator solution UML class diagram

other unrelated features. Although, there are two additional metrics in this tool that are not strictly original CK: Ca (Afferent coupling) and NPM (Number of Public Methods for a class), which will not be focused on in this study. The following metrics remain:

- WMC – weighted methods for a class, linked with complexity and maintainability, the low value of which indicates better software quality
- DIT – depth of inheritance tree, indicates the proximity of a measured class to its root class
- NOC – number of children of a class
- CBO – coupling between object classes
- RFC – response for a class, meaning the number of possible methods invoked in response to any message received by the class
- LCOM – lack of cohesion between methods, calculates the number of methods that are unrelated to each other or dissimilar

It is considered that the lower the above metrics are, the less complex and the more maintainable the software is. However, [8] indicated that LCOM does not have the same effect on maintainaibility as other metrics do, which was supported by [12]. On the other hand, WMC and CBO were highlighted as especially related to maintainability [8]. The desired threshholds for each metric in terms of software maintainability were produced by [10], guided by recommendations from NASA:

- WMC – 5 to 10 (influence on maintainability is 85% to 67%)
- DIT – 1 to 3 (99% to 74%)
- NOC – 2 to 4 (70% to 10%)
- CBO – 1 to 3 (92% to 86%)
- RFC – 5 to 10 (96% to 84%)
- LCOM – 0.05 to 0.18 (94% to 75%)

Thus, a higher software maintainability in the code is more probable when the values of the metrics are lower. Although, in case the last metric shows the opposite, it does not have the same influence as if the others would. The CKJM tool shows the CMD output for each class on every metric in the following order: WMC, DIT, NOC, CBO, RFC, LCOM. The latter two values shown will be disregarded as they are the same as CBO and WMC, respectively. The overall performance of the solution with regards to the CK metrics will be the average value of the values of its classes. The next section showcases and analyses the results of each solution.

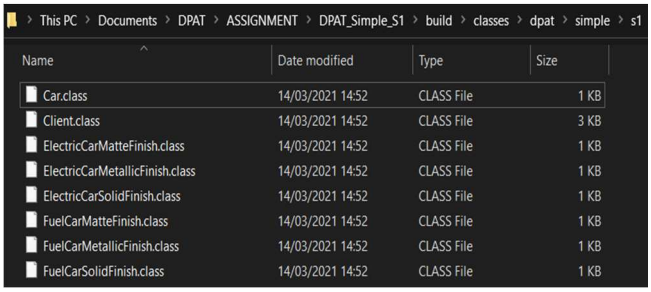


Fig. 4: Research Methodology

III. Results and Discussion

A. Simple Solution

The CMD command that allows to measure these metrics has the following structure:

java -jar "ckjm.jar directory" "class files directory*.class"

This java command runs the ckjm.jar file and all the *.class files located in the "build/classes" directory to calculate CK metrics for each class, as it was explained in the previous section. Each solution will then be accompanied by a table with a calculation of averages for each metric per class. To evaluate the first simple solution (where "your directory" implies the directory to which the provided files will be downloaded):

java -jar "your directory\ckjm-1.9\build\ckjm-1.9.jar

"your directory\DPAT_Simple_S1\build\classes\dpat\simple\s1*.class"

```
dpat.simple.s1.FuelCarSolidFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.ElectricCarMetallicFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.FuelCarMatteFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.FuelCarMetallicFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.ElectricCarMatteFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.ElectricCarSolidFinish 3 0 0 1 7 3 1 3
dpat.simple.s1.Car 7 1 6 0 8 17 7 7
dpat.simple.s1.Client 2 1 0 7 21 1 0 2
```

Fig. 5: Simple solution (S1) CKJM results

Table I below shows the CKJM average results for Simple Solution (S1).

TABLE I: SIMPLE SOLUTION (S1) CKJM AVERAGE RESULTS

Class	WMC	DIT	NOC	CBO	RFC	LCOM
FuelCarSolidFinish	3	0	0	1	7	3
ElectricCarMetallic Finish	3	0	0	1	7	3
FuelCarMatteFinish	3	0	0	1	7	3
FuelCarMetallicFinish	3	0	0	1	7	3
ElectricCarMatteFinish	3	0	0	1	7	3
ElectricCarSolidFinish	3	0	0	1	7	3
Car	7	1	6	0	8	17
Client	2	1	0	7	21	1
Average:	3.38	0.25	0.75	1.63	8.88	4.5

B. *Abstract Factory based Solution*

Following command is used to represents the evaluation of the Abstract Factory based solution:

java -jar "your directory\ckjm-1.9\build\ckjm-1.9.jar"

"your directory\DPAT_AbstractFactory_S2\build\classes\dpat\simple\s2*.class"

```
dpat.abstractfactory.s2.MatteFinish 2 0 0 1 4 1 1 2
dpat.abstractfactory.s2.Car 4 1 2 0 5 0 6 4
dpat.abstractfactory.s2.MetallicFinish 2 0 0 1 4 1 1 2
dpat.abstractfactory.s2.Client 2 1 0 4 18 1 0 2
dpat.abstractfactory.s2.Finish 4 1 3 0 5 0 7 4
dpat.abstractfactory.s2.FuelCar 2 0 0 1 4 1 1 2
dpat.abstractfactory.s2.SolidFinish 2 0 0 1 4 1 1 2
dpat.abstractfactory.s2.CarFactory 3 0 0 5 8 3 1 3
dpat.abstractfactory.s2.AbstractFactory 3 1 2 2 4 3 4 3
dpat.abstractfactory.s2.FinishFactory 3 0 0 6 9 3 1 3
dpat.abstractfactory.s2.FactoryProducer 2 1 0 3 7 1 1 2
dpat.abstractfactory.s2.ElectricCar 2 0 0 1 4 1 1 2
```

Fig. 6: Abstract Factory solution (S2) CKJM results

Table II below shows the CKJM average results for the Abstract Factory based solution (S2).

TABLE II: ABSTRACT FACTORY SOLUTION (S2) CKJM AVERAGE RESULTS

Class	WMC	DIT	NOC	CBO	RFC	LCOM
MatteFinish	2	0	0	1	4	1
Car	4	1	2	0	5	0
MetallicFinish	2	0	0	1	4	1
Client	2	1	0	4	18	1
Finish	4	1	3	0	5	0
FuelCar	2	0	0	1	4	1
SolidFinish	2	0	0	1	4	1
CarFactory	3	0	0	5	8	3
AbstractFactory	3	1	2	2	4	3
FinishFactory	3	0	0	6	9	3
FactoryProducer	2	1	0	3	7	1
ElectricCar	2	0	0	1	4	1
Average	2.58	0.42	0.58	2.08	6.08	1.33

C. *Decorator based Solution*

Following command is used to represents the evaluation of the Decorator based solution.

java -jar "your directory\ckjm-1.9\build\ckjm-1.9.jar"

"your directory\DPAT_AbstractFactory_S2\build\classes\dpat\simple \s2*.class"

```
dpat.decorator.s3.Car 1 1 0 0 1 0 6 1
dpat.decorator.s3.ElectricCar 2 1 0 1 3 1 1 2
dpat.decorator.s3.FuelCar 2 1 0 1 3 1 1 2
dpat.decorator.s3.CarDecorator 2 1 2 1 4 0 2 2
dpat.decorator.s3.MetallicFinishDecorator 2 0 0 2 7 1 1 2
dpat.decorator.s3.MatteFinishDecorator 2 0 0 2 7 1 1 2
dpat.decorator.s3.Client 2 1 0 5 14 1 0 2
```

Fig.7: Decorator solution (S3) CKJM results

Table 3 shows the CKJM average results for the Decorator based solution (S3).

TABLE III: DECORATOR SOLUTION (S3) CKJM AVERAGE RESULTS

Class	WMC	DIT	NOC	CBO	RFC	LCOM
Car	1	1	0	0	1	0
ElectricCar	2	1	0	1	3	1
FuelCar	2	1	0	1	3	1
CarDecorator	2	1	2	1	4	0
MetallicFinishDecorator	2	0	0	2	7	1
MatteFinishDecorator	2	0	0	2	7	1
Client	2	1	0	5	14	1
Average:	1.86	0.71	0.29	1.71	5.57	0.71

Judging by the results displayed in Fig. 8 and Fig. 9, design pattern solutions S2 and S3 outperformed the simple solution S1 in 4 out of 6 metrics, excluding DIT and CBO where S1 scored lower values. However, it can be seen in Fig. 8 that S1 did not score that much lower, and the difference is not as considerable or telling. On the other hand, between S1 and S2, the Decorator design pattern proved to be more suitable to increase the maintainability of the car manufacturing system. Although, Fig. 9 shows that the lines of Abstract Factory and Decorator are quite close together and that might indicate that applying either one would have approximately similar positive effect on maintainability.

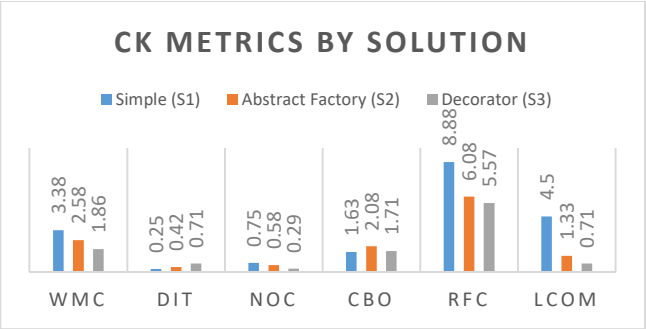
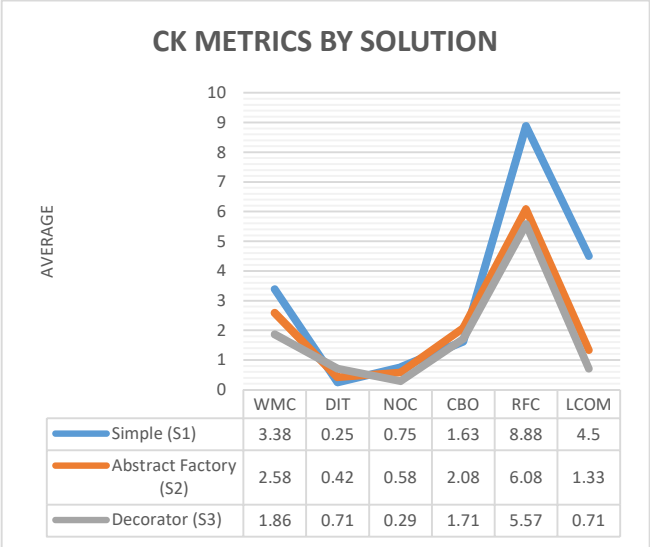


Fig. 8: Bar chart of average CK metrics by solution

Notably, some metric values were out of the recommended optimal ranges stated before, only the latter three came relatively close. This might be due to the small scale of the system and the scenario in general – the results should show clearer results if the scenario is expanded to a real-world one. All in all, it can be said that applying design patterns to an object-oriented design and code has produced positive effects on improving software maintainability, and the Decorator design pattern might prove to be more suitable to the specified problem scenario in the long run.



	WMC	DIT	NOC	CBO	RFC	LCOM
Simple (S1)	3.38	0.25	0.75	1.63	8.88	4.5
Abstract Factory (S2)	2.58	0.42	0.58	2.08	6.08	1.33
Decorator (S3)	1.86	0.71	0.29	1.71	5.57	0.71

Fig. 9: Line chart of average CK metrics by solution

IV. CONCLUSION

This research attempts to prove the impact of Abstract Factory and Decorator design patterns on maintainability. Firstly, A sample problem scenario of car manufacturing was devised to illustrate the requirement of applying two design patterns simply and understandably, namely Abstract Factory and Decorator. Secondly, UML class diagrams and Java implementations were produced for three variants of problem scenario solutions: simple solution with no particular design patterns applied (S1), Abstract Factory solution (S2), and Decorator solution (S3). Finally, each approach was empirically evaluated using well-known and established object-oriented software metrics created by Shyam R. Chidamber and Chris F. Kemerer. The actual tool that was used for the evaluation of said metrics is the CKJM tool created by Diomidis Spinellis. The results were displayed in the form of two graphs, a bar chart and a line chart. Analysis showed that the application of design patterns positively influenced software maintainability, thus proving the initial intentions and hypotheses of the researcher. Remarkably, there was a slightly better performance demonstrated by the Decorator design pattern, which might mean that this solution would be more suitable in this scenario, if it were on a larger and a real-world scale.

ACKNOWLEDGMENT

The work described in this paper was supported by the Asia Pacific University of Technology & Innovation (APU) Malaysia. The authors of this paper are also thankful to the Mae Fah Luang University for providing the opportunity to present this paper under IEEE platform.

REFERENCES

[1] J. P. Cavano and J. A. McCall, "A Framework for the Measurement of Software Quality," in *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, New York, NY, USA, Association for Computing Machinery, 1978, p. 133–139, 1978.

[2] J. Visser, S. Rigal, R. V. D. Leek, P. V. Eck and G. Wijnholds, "What is maintainability? Introduction," in *Building Maintainable Software, Java Edition*, O'Reilly Media, 2016.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns Abstraction and Reuse of Object-Oriented Design," in *ECOOP '93: Object-Oriented Programming*, Kaiserslautern, Germany, 1993.

[4] C. Caballero, "Understanding the Abstract Factory Design Patterns | by Carlos Caballero | Jan 2021 | Better Programming," 2015. [Online]. Available: https://betterprogramming.pub/understanding-the-abstract-method-design-patterns-bc416aaaf076. [Accessed 13 March 2021].

[5] Poyias, "Design Patterns — A quick guide to Decorator pattern. | by Andreas Poyias | Medium," 2019. [Online]. Available: https://medium.com/@andreaspoyias/design-patterns-a-quick-guide-to-decorator-pattern-2159b97863f. [Accessed 13 March 2021].

[6] M. Rafla, "Decorator Design Pattern — miraf. The goal of the Decorator Design… | by Mina RAFLA | Analytics Vidhya | Medium," 2020. [Online]. Available: https://medium.com/analytics-vidhya/decorator-design-pattern-miraf-d978735df38c. [Accessed 13 March 2021].

[7] M. E. Rana, W.N.W. Ab Rahman and M. Ahmed, "Evaluating Design Pattern Based Solutions with their Equivalent Simpler Solutions to Promote Maintainability in Software," *Advanced Science Letters*, vol. *24, no.* 3, pp. 1702-1707, 2018.

[8] H. K. Jun and M. E. Rana, "Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation," 2021 Third International Sustainability and Resilience Conference: Climate Change, Nov. 2021, pp. 539-548.

[9] U. L. Kulkarni, Y. R. Kalshetty and V. G. Arde, "Validation of CK metrics for Object Oriented Design Measurement," Goa, India, 2010.

[10] V. Rai, A. M. Srivastava, H. Pandey and D. V. K. Singh, "Estimation of Maintainability in Object Oriented Design Phase: State of the art," *International Journal of Scientific & Engineering Research,* vol. 6, no. 9, pp. 25-35, 2015.

[11] T. R. G. Nair, S. Aravindh and R. Selvarani, "Design Property metrics to Maintainability estimation – A virtual method using functional relationship mapping," *ACM SIGSOFT Software Engineering Notes,* vol. 35, no. 6, pp. 1-6, 2010.

[12] B. Anda, "Assessing Software System Maintainability using Structural Measures and Expert Assessments," Paris, France, 2007.

[13] D. S. Chawla and G. Kaur, "Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development," *(IJACSA) International Journal of Advanced Computer Science and Applications,* vol. 4, no. 9, pp. 161-164, 2013.

[14] D. Spinellis, "Tool writing: a forgotten art? (software tools)," *IEEE Software,* vol. 22, no. 4, pp. 9 - 11, 2005.

[15] D. Spinellis, "CKJM - A Tool for Calculating Chidamber and Kemerer Java Metrics," 2005. [Online]. Available: https://www.spinellis.gr/sw/ckjm/doc/index.html. [Accessed 12 May 2021].