Article

Assessing System Quality Changes during Software Evolution: The Impact of Design Patterns Explored via Dependency Analysis

Kuo-Hsun Hsu *, Hua-Chieh Szu-Tu and Chia-Hsing Tsai

Department of Computer Science, National Taichung University of Education, Taichung 403514, Taiwan; bcs109111@gm.ntcu.edu.tw (H.-C.S.-T.); bcs112118@gm.ntcu.edu.tw (C.-H.T.)
* Correspondence: glenn@mail.ntcu.edu.tw

Abstract: Design patterns provide solutions to recurring problems in software design and development, promoting scalability, readability, and maintainability. While past research focused on the utilization of the design patterns and performance, there is limited insight into their impact on program evolution. Dependency signifies relationships between program elements, reflecting a program's structure and interaction. High dependencies indicate complexity and potential flaws, hampering system quality and maintenance. This paper presents how design patterns influence software evolution by analyzing dependencies using the Abstract Syntax Tree (AST) to examine dependency patterns during evolution. We employed three widely adopted design patterns from the Gang of Four (GoF) as experimental examples. The results show that design patterns effectively reduce dependencies, lowering system complexity and enhancing quality.

Keywords: design pattern; dependency relationships; abstract syntax tree; software quality; software evolution

check for updates

Citation: Hsu, K.-H.; Szu-Tu, H.-C.; Tsai, C.-H. Assessing System Quality Changes during Software Evolution: The Impact of Design Patterns Explored via Dependency Analysis. *Electronics* **2024**, *13*, 1444. https://doi.org/10.3390/electronics13081444

Academic Editor: Paulo Ferreira

Received: 31 January 2024
Revised: 6 April 2024
Accepted: 8 April 2024
Published: 11 April 2024

1. Introduction

Design patterns offer significant advantages in software development and have become essential standards for high-quality software design. Software engineers widely adopt them to enhance flexibility, reusability, and maintainability [1,2]. Current research mainly delves into pattern selection and implementation and their impact on systems [3–5]. Despite the extensive research on the impact of design patterns on software evolution, there remains a notable research gap in understanding the comprehensive influence of various design patterns on real-world software systems. Existing studies often focus on a limited set of design patterns within specific system versions, neglecting the broader spectrum of design patterns and their effects on long-term software evolution.

Dependency relationships represent the associations, interactions, and behaviors among program elements within a software system [6–8]. An evaluation of the impact and utility of design patterns in software systems often considers how design patterns influence dependency relationships, as such effects may have implications for software quality metrics such as complexity, maintainability, and scalability [9,10].

A total of 804 studies published between 2000 and 2018 were examined by Wedyan et al. [11]. The statistical analysis determined that the Factory Pattern, Decorator Pattern, and Observer Pattern are the most commonly used design patterns in three categories: Creational Pattern, Structural Pattern, and Behavioral Pattern. McNatt et al. [12] also mentioned that the Factory Pattern and Observer Pattern are the most frequently referenced design patterns. Similarly, Prechelt et al. [13] conducted research focusing on the Factory Pattern, Observer Pattern, and Decorator Pattern. Therefore, in this study, our attention was directed toward these three widely acknowledged design patterns sourced from the Gang of Four (GoF) [14]: the Factory Pattern, Decorator Pattern, and Observer Pattern. We designed individual experiments for each of the selected design patterns. To present the results generated by these design patterns in the experiments, we chose 39 dependency relationships from the 56 proposed by Huang [7] that are commonly associated with the three selected design patterns. By observing the variations in these dependency relationships in the experiment, the influence of these design patterns on different experimental environments can be determined. Additionally, we created a computational tool with Java Parser [15] to assist in minimizing the interference of the experimental environment during the calculation of dependency relationships. The assistance tool scans example programs employing the specified design patterns and those that do not and identifies code dependency relationships. Furthermore, our investigation included simulations of alterations in dependency relationships during program evolution. This allowed us to illustrate how design patterns can effectively diminish dependencies that arise throughout the evolution of a system, ultimately leading to the optimization of the software system.

In the following sections, we explore background knowledge (Section 2), detail our methodology for identifying dependency relationships (Section 3), present and discuss experimental results (Section 4), and conclude by outlining contributions and future research plans (Section 5).

2. Background Knowledge and Related Work

This section introduces the background knowledge and research context for this study, categorized into three subsections: Design Patterns, Dependency Relationships, and Abstract Syntax Trees (ASTs).

2.1. Design Patterns

Design patterns [14] are crucial principles in software engineering, providing reusable solutions to common design problems. They offer a structured approach to tackling typical design challenges in software development and bring various advantages to the process. Beyond resolving issues in system design, design patterns contribute to enhancing key software quality metrics, including readability, reusability, extensibility, maintainability, cohesion, and coupling [1,16,17].

Nanthaamornphong et al. [1] conducted a statistical survey on the Visitor Pattern, demonstrating that the complexity of software design was effectively reduced through the Visitor Pattern. Tahvildari et al. [16] found that utilizing design patterns in the system can enhance software system maintainability, whether creational, structural, or behavioral. In addition, Ampatzoglou et al. [17] revealed that, while incorporating GoF design patterns into a system may increase system size (including lines of code and the number of classes), all identified GoF design patterns improved system cohesion, reduced coupling, and decreased complexity. However, Khomh et al. [4] argued that not all design patterns are advantageous to system quality and may even reduce system extensibility or readability.

The Factory Pattern is a creational design pattern that defines an interface for creating objects in a superclass. However, it allows subclasses to modify the type of objects to be created. This pattern encourages loose coupling by removing the necessity to bind application-specific classes into the code directly. The Decorator Pattern is a structural design pattern that enables the addition of behavior to an individual object, either statically or dynamically, without impacting the behavior of other objects within the same class. This is accomplished by creating a series of decorator classes to encapsulate concrete components. The Observer Pattern is a behavioral design pattern in which an object, referred to as the subject, manages a list of dependents, known as observers, who should be informed of changes to the subject's state. It establishes a one-to-many dependency between objects, ensuring that all its dependents are notified when one object changes its state. In this study, the experiment focused on the Factory Pattern, Decorator Pattern, and Observer Pattern, the results of which will be presented in Section 4.

2.2. Dependency Relationships

Dependency relationships describe interactions between program elements, such as package names, field names, types, etc. These elements are nodes in the program. Bichsel [6] classified 28 types of dependencies into method relationships and structural relationships. Method relationships explain the semantic behavior of methods. For instance, in the method call field.foo().bar(), two receiver-type dependencies emerge: (foo, bar, receiver) and (field, foo, receiver). Structural relationships capture connections between nodes. For example, the study introduced "read-before" and "written-before" relationships to depict the order of reading or writing fields. Huang [7] increased the number of dependency relationships to 56 and classified them into three types based on object-oriented concepts: encapsulation, abstraction, and delegation.

Encapsulation is a method to protect data from unauthorized access and modification by bundling data and methods in the program into a single unit. In addition to making programs easier to maintain and extend, encapsulation provides a clean and user-friendly interface due to its ability to hide details. When there are more encapsulation dependency relationships in the program, it typically means that not only is the structure of the program more closed and independent, but unit testing and refactoring are also easier to perform, making the program easier to maintain, extend, and understand. However, excessive encapsulation dependency relationships may lead to a high degree of coupling in the program, making it more difficult to modify and extend. If all parts of the program depend on each other, changing one part may require changing other parts simultaneously. This kind of dependency relationship may not only lead to the program design becoming more complex but also increase the difficulty of maintenance.

Abstraction is the process of hiding complexity, focusing on key details, and simplifying the functionality of the object or system into interfaces. It is usually used to hide details, making the program easier to maintain and extend. When there are more abstraction dependency relationships, it typically means that the program has a higher level of abstraction and lower dependency. Because the program can be easily changed with little effect on others, it may be easier to maintain and extend. However, excessive abstraction dependency relationships may lead to an unduly high level of abstraction in the program, not only making the program difficult to understand and maintain but also making the dependency relationships hard to manage.

Delegation is the process of distributing and entrusting work from one object to another. When a proxy object needs to provide a specific work, it entrusts the work to another rather than implementing it by itself. This typically means that the structure of the program becomes more complex when there are lots of delegation dependency relationships in the program. Because each object needs to be understood and managed, a large number of objects that depend on each other may require more maintenance and testing. Furthermore, multiple layers of delegation may also lead to reduced execution efficiency because each object needs to process and transmit information. However, because it makes the implementation of proxy objects easy to replace with other objects, programs can be made easier to extend and modify by using delegation appropriately. Moreover, delegation can also separate different works, making the program more modular and readable.

In this paper, we have selected 39 dependency relationships for our experiment. These selected dependencies are re-categorized into three types, encapsulation, abstraction, and delegation, forming the basis of our research. When analyzing the quality of software systems, dependency relationships are often considered as one of the factors [9,10]. In [9], Iyapparaja et al. mention that dependency relationships influence the complexity of the program. The more complex the interactions within a system, the harder the readability of specific designs, leading to a higher likelihood of faults and making testing more difficult. Dependency relationships are also utilized in research on software system refactoring. Maruyama et al. [18] proposed a mechanism for automating the refactoring of object-oriented frameworks through a weighted dependency relationship graph. Experiments demonstrated that the number of statements developers need to write when creating multiple applications can be reduced by up to 22 percent.

2.3. Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) is an abstract representation used to describe the syntactic structure of code. It is a data structure employed by Java parsers to parse code into a format that is easier to understand and manipulate. ASTs represent the syntax of a programming language in a tree-like structure composed of nodes and edges. Each node represents an element in the code, such as a variable, operator, function call, and so on. Each node may also have additional attributes like values or child nodes to store the specific content or related sub-elements of that element. Edges denote the relationships between these elements, such as the parent–child relationships between nodes. The construction of an AST is implemented through the syntax analysis process, commonly known as parsing. A Java parser takes the source code and parses it into a series of syntactic units, such as tokens or abstract syntax units. It then uses these syntactic units to build the AST [19]. During the syntax analysis process, the code is validated according to the syntax rules of the programming language, and the corresponding AST structure is generated.

The AST can be used for syntax checking, error detection, code transformation, and more. For example, compilers can traverse the AST and perform various analyses and transformation operations, while parsers can use the AST to understand and perform a syntax analysis of program code. Developers can also utilize the AST for tasks like code refactoring and static code analysis [20]. Fauzi et al. [21] employed the AST to transform source code into sequence diagrams through reverse engineering. In [22], Tao et al. proposed a code plagiarism detection algorithm based on the AST. These examples demonstrate the versatility and utility of ASTs in various software engineering tasks.

3. Methodology

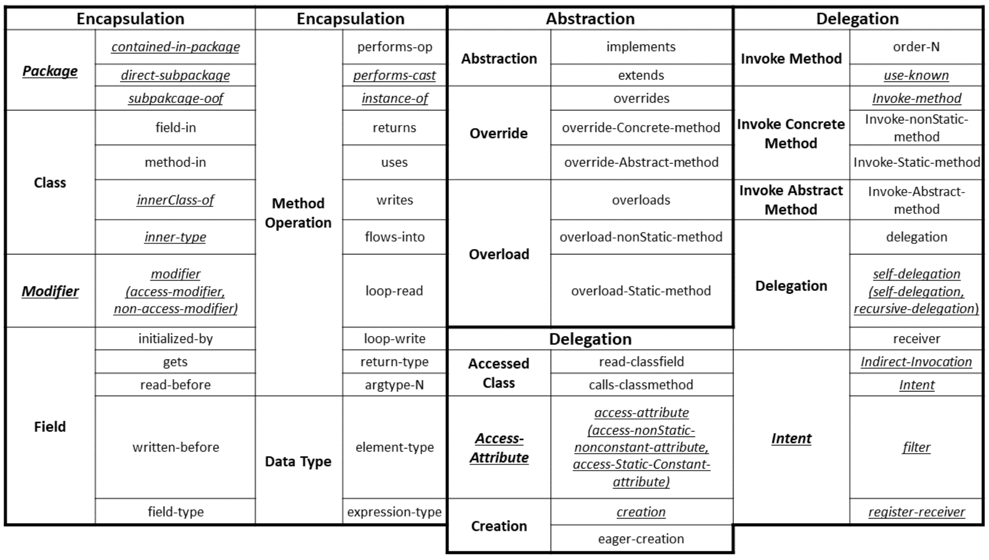
This section outlines our research methodology, structured into two parts: identifying dependency relationships and designing experimental procedures. The first part introduces dependency relationships related to design patterns and the computational tools developed to discover these relationships. Once the relationships are defined, the second part outlines the experimental procedure tailored to these relationships.

3.1. Identifying the Dependency Relationships

To assess software quality, this study evaluated dependency relationships in a software system. The upcoming subsection will detail the process of identifying these relationships.

3.1.1. Dependency Relationships Related to Design Patterns

This research specifically focuses on programs within the same package and design pattern. Therefore, certain types of dependency relationships are excluded: ‘contained-in-package’, ‘direct-subpackage-of’, ‘subpackage-of’, ‘innerClass-of’, ‘inner-type’, ‘performs-cast’, ‘instance-of’, and ‘use-known’. Additionally, dependencies related to the Android System, such as ‘indirect-invocation’, ‘intent’, ‘register-receiver’, and ‘filter’, will not be discussed. Furthermore, we categorized the dependency relationships with similar characteristics for the organization, such as ‘modifier’, ‘access-attribute’, and ‘self-delegation’. As a result, out of the 56 dependency relationships proposed by Huang [7], 39 relationships were selected in this research, as illustrated in Figure 1 (Dependency relationships with underlines indicate that they were excluded from this research). These chosen dependencies will be categorized into encapsulation, abstraction, and delegation.



Encapsulation	
Package	*contained-in-package*
	direct-subpackage
	subpakcage-oof
Class	field-in
	method-in
	innerClass-of
	inner-type
Modifier	*modifier (access-modifier, non-access-modifier)*
Field	initialized-by
	gets
	read-before
	written-before
	field-type

Encapsulation	
Method Operation	performs-op
	performs-cast
	instance-of
	returns
	uses
	writes
	flows-into
	loop-read
	loop-write
	return-type
	argtype-N
Data Type	element-type
	expression-type

Abstraction	
Abstraction	implements
	extends
Override	overrides
	override-Concrete-method
	override-Abstract-method
Overload	overloads
	overload-nonStatic-method
	overload-Static-method

Delegation	
Accessed Class	read-classfield
	calls-classmethod
Access-Attribute	*access-attribute (access-nonStatic-nonconstant-attribute, access-Static-Constant-attribute)*
Creation	*creation*
	eager-creation

Delegation	
Invoke Method	order-N
	use-known
Invoke Concrete Method	*Invoke-method*
	Invoke-nonStatic-method
	Invoke-Static-method
Invoke Abstract Method	Invoke-Abstract-method
Delegation	delegation
	self-delegation (self-delegation, recursive-delegation)
	receiver
Intent	*Indirect-Invocation*
	Intent
	filter
	register-receiver

Figure 1. Categorization of dependency relationships.

3.1.2. Dependency Relationship Calculation Tool

In order to calculate dependency relationships in the system, we developed a calculation tool using the API provided by JavaParser [15]. This tool is capable of not only parsing Java source code but also generating the Abstract Syntax Tree (AST). This AST is analyzed to identify whether the program contains the aforementioned 39 types of dependency relationships and calculate their quantity. Next, the following is elucidated through illustrative examples. In terms of the system architecture of this assistance tool, as shown in Figure 2, the AST of this program is generated by first passing the source code to Logic Positivizer. Next, the AST is passed separately to Encapsulation Finder, Abstraction Finder, and Delegation Finder for scanning. Finally, the number of dependency relationships for encapsulation, abstraction, and delegation types can be obtained.

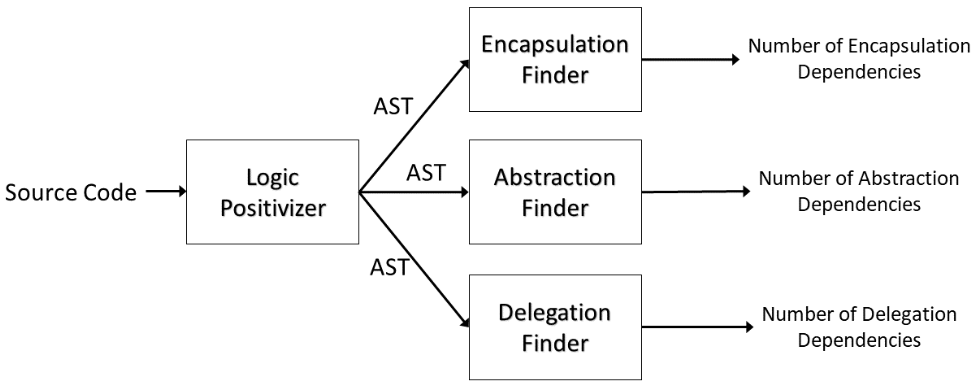

Source Code
Logic Positivizer
AST
AST
AST
Encapsulation Finder
Abstraction Finder
Delegation Finder
Number of Encapsulation Dependencies
Number of Abstraction Dependencies
Number of Delegation Dependencies

Figure 2. System architecture.

We elucidate the calculation method for each type of dependency relationship through mathematical expressions in Table 1, and the signification of the constituent elements within these mathematical formulations is shown in Table 2. Field-in in Table 2, for example, means all the fields declared in the package. By utilizing the mathematical expressions mentioned above within the tool, various dependencies within the code can be effectively identified, making subsequent experiments more convenient.

Table 1. The mathematical expressions of the dependency relationships.

Encapsulation dependency relationships	Mathematical expression
field-in	$\|x\|, x \in FD, x \in P$, *x is declared in P*
method-in	$\|x\|, x \in MD, x \in P$, *x is declared in P*
modifier	$A = MD \cup FD, \|(a,m)\|$, *m is the modifier of a*, $a \in A,\ b \in M$
initialized-by	$\|(a,b)\|$, *attribute a is initialized in class b*, $a \in Ab,\ b \in C$
gets	$A = Ab \cup MD \cup FD, \|(a,b)\|$, *b is an attribute assigned with a*, $a \in A,\ b \in Ab$
read-before	$\|(a,b)\|$, *a is a field which is read before b in the same method*, $a \in FD,\ b \in FD,\ a \neq b$
written-before	$\|(a,b)\|$, *a is a field which is written before b in the same method*, $a \in FD,\ b \in FD,\ a \neq b$
field-type	$\|x\|, x \in FD, x \in P$, *x is declared in P*
performs-op	$\|x\|, x \in op, x \subseteq C$
returns	$\|(a,b)\|$, *b has a return value a*, $a \in Vb,\ b \in M$
uses	$\|(a,b)\|$, *a is used in b*, $a \in A,\ b \in MD$
writes	$\|(a,b)\|$, *a is written in b*, $a \in FD,\ b \in MD$
flows-into	$\|(a,b)\|$, *a is an argument of b*, $a \in Vb,\ b \in MCD$
loop-read	$\|x\|, x \in Ab \cup MD \cup C$, *x is read in L*, $L \neq C$
loop-write	$\|x\|, x \in Ab \cup MD \cup C$, *x is read in L*, $L \neq C$
return-type	$\|(a,b)\|$, *b has a return value a*, $a \in Vb,\ b \in MD$
argtype-N	$\|(a,b)\|$, *a is an argument of b*, $a \in Vb,\ b \in MD$
element-type	$\|x\|, x \in FD \cap Ar$
expression-type	$\|x\|, x \in FD \cup Vb$
Abstraction dependency relationships	Mathematical expression
implement	$\|(a,b)\|$, *b implements interface a*, $a \in I,\ b \in C,\ a \neq b$
extends	$\|(a,b)\|$, *b extends a*, $a \in C,\ b \in C,\ a \neq b$
overrides	$\|(A,B)\|$, *a method in class B overrides a method in class A*, $A \in C,\ B \in C,\ A \neq B$
override-concrete-method	$\|(a,b)\|$, *method a overrides a concrete method b*, $a \in MD,\ a \subseteq C,\ b \in MD,\ b \subseteq C$
override-abstract-method	$\|(a,b)\|$, *method a overrides a abstract method b*, $a \in MD,\ a \subseteq C,\ b \in MD,\ b \subseteq C$
overloads	$\|(A,B)\|$, *a method in class B overloads a method in class A*, $A \in C,\ B \in C,\ A \neq B$
overload-static-method	$\|(a,b)\|$, *method a overloads a static method b*, $a \in MD,\ a \subseteq C,\ b \in MD,\ b \subseteq C$
overload-nonstatic-method	$\|(a,b)\|$, *method a overloads a non* $-$ *static method b*, $a \in MD,\ a \subseteq C,\ b \in MD,\ b \subseteq C$
Delegation dependency relationships	Mathematical expression
read-classfield	$\|(a,b)\|$, *method a reads a field of class b*, $a \in MD,\ b \in C$
call-classmethod	$\|(a,b)\|$, *method a calls a method of class b*, $a \in MD,\ b \in C$
access-attribute	$\|(a,b)\|$, *method a accesses a field of class b*, $a \in MD,\ b \in C$
creation	$\|(a,b)\|$, *method a creates an instance of class b*, $a \in MD,\ b \in C$
eager-creation	$\|(a,b)\|$, *field a creates an instance of class b*, $a \in FD,\ b \in C$
order-N	$\|(a,b)\|$, *b is an argument of method a*, $a \in MD,\ b \in Arg, Arg = FD \cup MD \cup C$
invoke-nonstatic-method	$\|(a,b)\|$, *non* $-$ *static method b is called by method a*, $a \in MD,\ b \in MDC$
invoke-static-method	$\|(a,b)\|$, *static method b is called by method a*, $a \in MD,\ b \in MD$
invoke-abstractc-method	$\|(a,b)\|$, *abstract method b is called by method a*, $a \in MD,\ b \in MD$
delegation	$\|(a,b)\|$, *an object of class a forwards an operation to an object of class b*, $a \in C,\ b \in C$
self-delegation	$\|(a,a)\|$, *an object of class a forwards an operation to itself*, $a \in C$
receiver	$\|(a,b)\|$, *a receives information from method b*, $a \in A,\ b \in MD, A = It \cup MD \cup Ab$

Table 2. Terms and Explanations table.

Term	**Definition**
Package (P)	the set of all program packages
Class (C)	the set of all program classes
Field (FD)	the set of all program fields across all classes
Method (MD)	the set of all program methods across all classes
Attribute (Ab)	the set of all attributes across all classes
Variable(Vb)	the set of all variables in the program
Loop (L)	the set of all loops in the program
MethodCall (MCD)	the set of all method calls in the program
Static-Method (SMD)	the set of all static methods across all classes
nonStatic-Method (nSMD)	the set of all non-static methods across all classes
Modifier (M)	the set of all modifiers in the program
Array (Ar)	the set of all arrays in the program
Interfaces (I)	the set of all program interfaces
Instance (It)	the set of all instances in the program

3.2. Design Experiment Steps

In order to investigate the relationship between design patterns and software quality, we designed the following experimental procedure. It is divided into four steps, as shown in Figure 3. The first step involves designing a sample program written without the use of design patterns. In the second step, a new sample program is designed by applying design patterns to refactor the initial program from step one. Step three simulates the evolution of both sample programs. In step four, the analysis tool is employed to scan the program to identify the number of dependency relationships in the code.

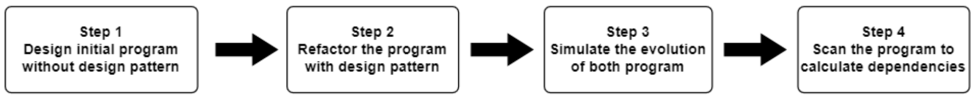

Step 1
Design initial program without design pattern
Step 2
Refactor the program with design pattern
Step 3
Simulate the evolution of both program
Step 4
Scan the program to calculate dependencies

Figure 3. Experimental steps.

Take the Decorator Pattern as an example. The program for the experiment implements a pizza-ordering system, as shown in Figure 4. First, an initial design without the use of design patterns is created, as shown in Figure 4a. In step two, the initial code is refactored using the Decorator design pattern, as shown in Figure 4b. Step three simulates the evolution of both programs, as shown in Figure 4c,d. Finally, the dependency relationship calculation tool is used to scan and calculate the number of dependency relationships in these programs for comparison.

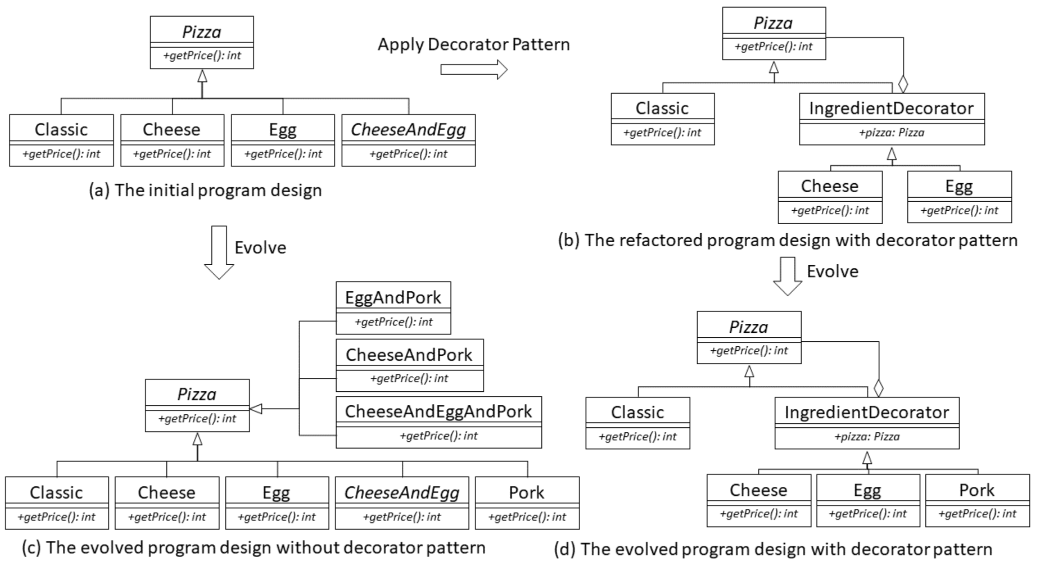

Apply Decorator Pattern
Pizza
+getPrice(): int
Classic
Cheese
Egg
CheeseAndEgg
(a) The initial program design
IngredientDecorator
+pizza: Pizza
(b) The refactored program design with decorator pattern
Evolve
EggAndPork
CheeseAndPork
CheeseAndEggAndPork
Pork
(c) The evolved program design without decorator pattern
(d) The evolved program design with decorator pattern

Figure 4. Decorator program design.

This study performed calculations for three types of dependency relationships: encapsulation, abstraction, and delegation. The total number of dependency relationships for each type is identified, as well as the average number of these contained in each class within the program. The results are presented as line charts in the next section.

4. Experiment and Results

This experiment focused on the aforementioned three types of design patterns: Factory Pattern, Decorator Pattern, and Observer Pattern. The purpose of this experiment was to compare the changes in dependency relationships within a program containing these design patterns with the changes in the program without these design patterns.

4.1. Factory Pattern

For the experiment with the Factory Pattern, this study designed a simple beverage store system as an example, with its program design illustrated in Figure 5. Figure 5a represents the initial design of the system, with a BeverageStore offering both GreenTea and BlackTea. With the continual addition of beverage items, this design requires frequent changes in the BeverageStore class to incorporate code for new items. The demand for frequent changes in program elements can be achieved by refactoring with the Factory Pattern, as shown in Figure 5b. Because of the abstraction of beverages through the use of the class SimpleBeverageFactory and the interface IBeverageProvider, it becomes unnecessary to modify the existing BeverageStores's code when adding new beverage items. Two continuous versions of the BeverageStore system are represented in Figure 5c,d.

The following section presents our experimental results for the dependency relationships (encapsulation, abstraction, and delegation) in the Factory Pattern.

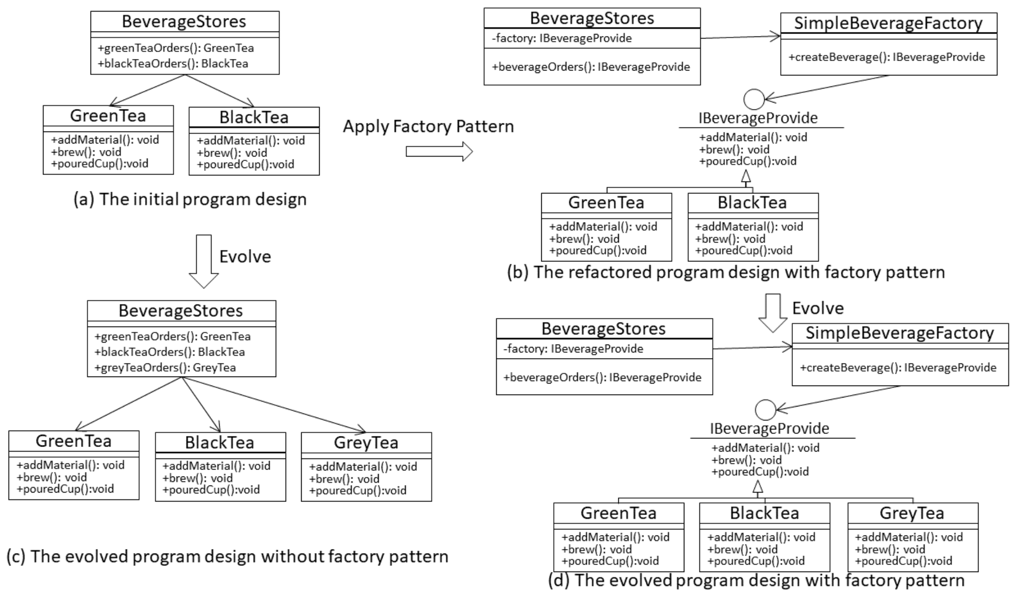

BeverageStores
+greenTeaOrders(): GreenTea
+blackTeaOrders(): BlackTea
GreenTea
BlackTea
GreyTea
+addMaterial(): void
+brew(): void
+pouredCup():void
(a) The initial program design
Apply Factory Pattern
-factory: IBeverageProvide
+beverageOrders(): IBeverageProvide
SimpleBeverageFactory
+createBeverage(): IBeverageProvide
IBeverageProvide
(b) The refactored program design with factory pattern
Evolve
+greyTeaOrders(): GreyTea
(c) The evolved program design without factory pattern
(d) The evolved program design with factory pattern

Figure 5. Factory program design.

4.1.1. Encapsulation Dependency Relationships in Factory Pattern

The total number of encapsulation dependency relationships is shown in Table 3, and the average number of encapsulation dependency relationships within each class in the program is displayed in Table 4. By visualizing the experimental results as line charts, it can be observed that, though using the Factory Pattern leads to more encapsulation dependency relationships in the early stages of development, as the program evolves into later stages, the Factory Pattern continues to effectively reduce the number of these relationships and results in lower system complexity. The line representing the Factory Pattern is color-coded in red, as shown in Figures 6 and 7.

Table 3. Total encapsulation dependencies in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	28	50
3	42	61
4	56	72
5	70	83
6	84	94
7	98	105
8	112	116
9	126	127
10	140	138
11	154	149
12	168	160
13	182	171
14	196	182
15	210	193

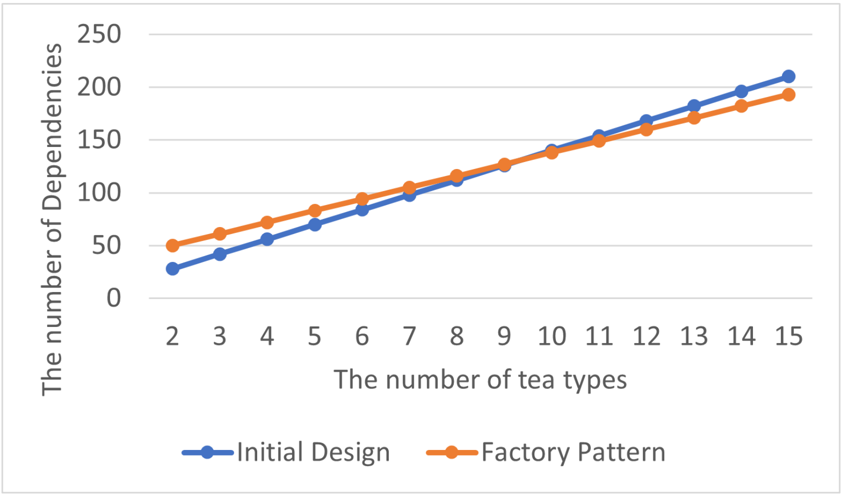


Figure 6. Factory Pattern: total encapsulation dependencies.

Table 4. Avg. encapsulation dependencies per class in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	9.333	10
3	10.5	10.167
4	11.2	10.286
5	11.667	10.375
6	12	10.444
7	12.25	10.5
8	12.444	10.545
9	12.6	10.583
10	12.727	10.615
11	12.833	10.643
12	12.923	10.667
13	13	10.688
14	13.067	10.706
15	13.125	10.722

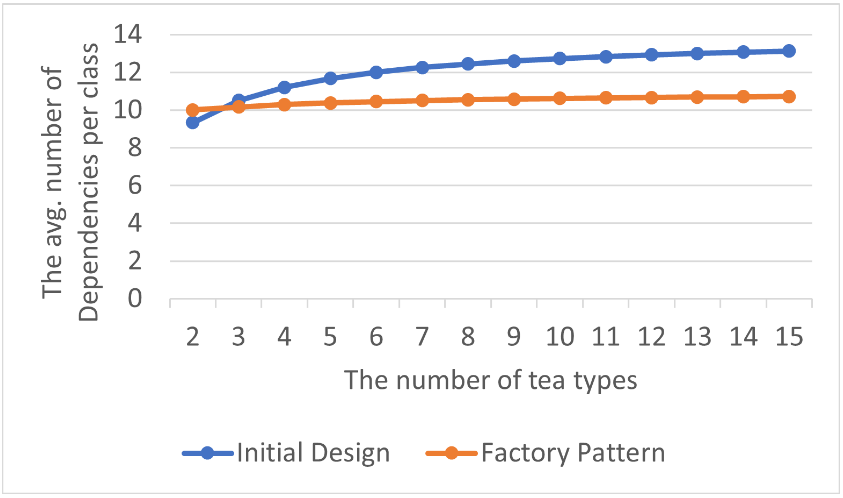


Figure 7. Factory Pattern: avg. encapsulation dependencies per class.

4.1.2. Abstraction Dependency Relationships in Factory Pattern

The total number of abstraction dependency relationships is shown in Table 5, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 6. Although the use of the Factory Pattern generates additional abstraction dependency relationships, the number of these dependency relationships still remains within a reasonable range, as shown in Figures 8 and 9. Furthermore, based on the easy modification of its details without causing significant impacts on other parts, moderate abstraction can make the program easier to maintain and extend.

Table 5. Total abstraction dependencies in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	0	2
3	0	3
4	0	4
5	0	5
6	0	6
7	0	7
8	0	8
9	0	9
10	0	10
11	0	11
12	0	12
13	0	13
14	0	14
15	0	15

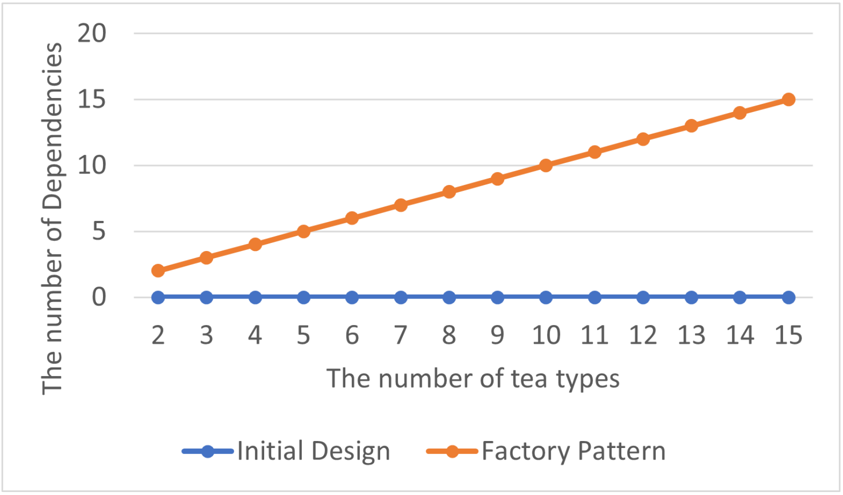


Figure 8. Factory Pattern: total abstraction dependencies.

Table 6. Avg. abstraction dependencies per class in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	0	0.4
3	0	0.5
4	0	0.571
5	0	0.625
6	0	0.667
7	0	0.7
8	0	0.727
9	0	0.75
10	0	0.769
11	0	0.786
12	0	0.8
13	0	0.813
14	0	0.824
15	0	0.833

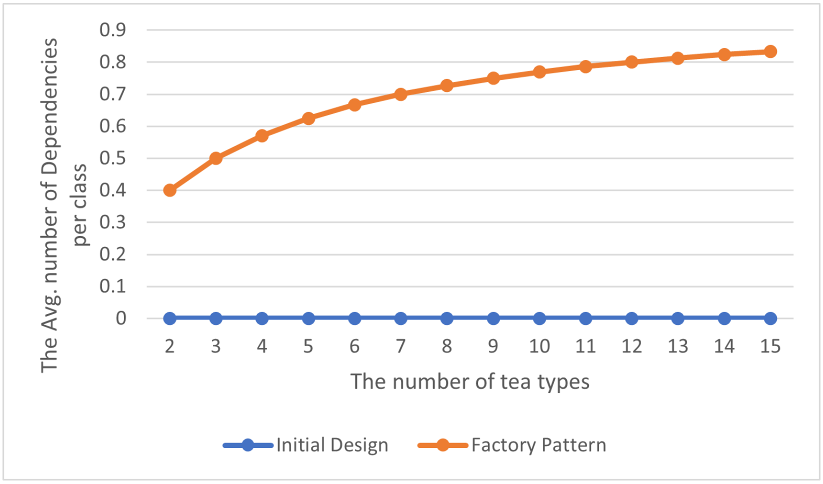


Figure 9. Factory Pattern: avg. abstraction dependencies per class.

4.1.3. Delegation Dependency Relationships in Factory Pattern

The total number of delegation dependency relationships is shown in Table 7, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 8. By visualizing the experimental results as line charts, it can be observed that a significant increase in delegation dependency relationships is caused in the program by not using the Factory Pattern, as shown in Figures 10 and 11. Moreover, excessive delegation dependency relationships may lead to higher program coupling, making modifications and extensions more challenging.

Table 7. Total delegation dependencies in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	16	9
3	24	10
4	32	11
5	40	12
6	48	13
7	56	14
8	64	15
9	72	16
10	80	17
11	88	18
12	96	19
13	104	20
14	112	21
15	120	22

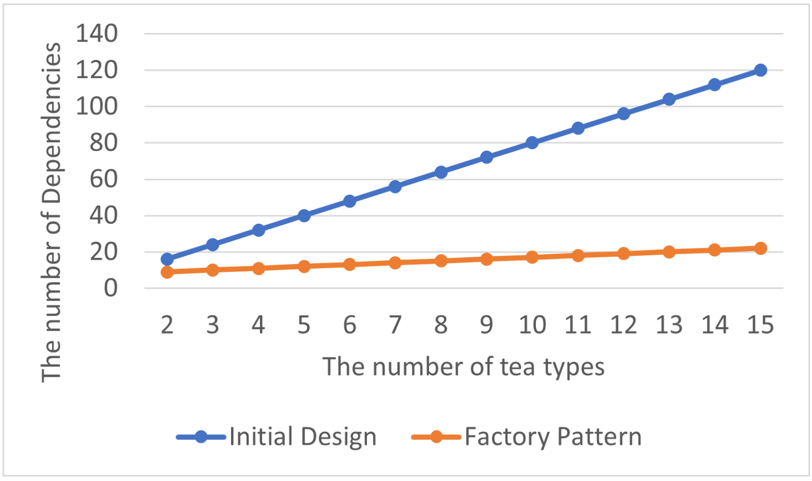


Figure 10. Factory Pattern: total delegation dependencies.

Table 8. Avg. delegation dependencies per class in Factory Pattern.

Tea Number	Initial Design	Factory Pattern
2	5.333	1.8
3	6	1.667
4	6.4	1.571
5	6.667	1.5
6	6.857	1.444
7	7	1.4
8	7.111	1.364
9	7.2	1.333
10	7.272	1.308
11	7.333	1.286
12	7.385	1.267
13	7.428	1.25
14	7.467	1.235
15	7.5	1.222

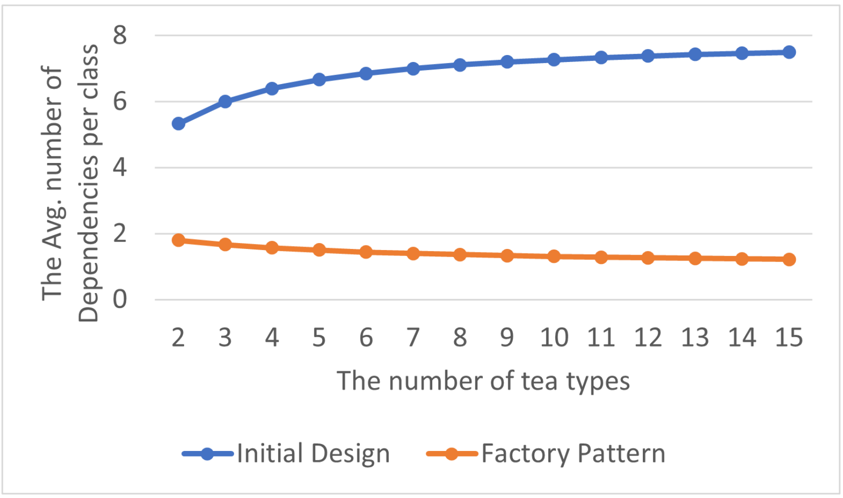


Figure 11. Factory Pattern: avg. delegation dependencies per class.

4.2. Decorator Pattern

For the experiment involving the Decorator Pattern, this study designed a pizza-ordering system calculating the corresponding price based on the selected topping as an example, with its program design illustrated previously in Figure 4a. There are two toppings available, Cheese and Egg, which results in four different combinations and their corresponding prices: Classic, Cheese, Egg, or Cheese and Egg. However, when the restaurant decides to add a third topping, the number of available combinations to choose from will increase significantly, leading to an overly large and excessively complex system, as shown in Figure 4c. This issue can be improved by using the Decorator Pattern. Treating various toppings as different decorators can prevent the proliferation of numerous classes during the initial program evolution, as shown in Figure 4b,d.

The following section presents our experimental results for the dependency relationships (encapsulation, abstraction, and delegation) in the Decorator Pattern.

4.2.1. Encapsulation Dependency Relationships in Decorator Pattern

The total number of encapsulation dependency relationships is shown in Table 9, and the average number of encapsulation dependency relationships within each class in the program is displayed in Table 10. By visualizing the experimental results as line charts, it can be observed that the number of encapsulation dependency relationships in the program that does not contain the Decorator Pattern is significantly higher than that in the one with the Decorator Pattern, as shown in Figures 12 and 13. As mentioned in Section 3.1.1, an excessive number of encapsulation dependency relationships leads to high program complexity, making modifications and extensions challenging.

Table 9. Total encapsulation dependencies in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	20	19
3	36	23
4	68	27
5	132	31
6	260	35
7	516	39
8	1028	43

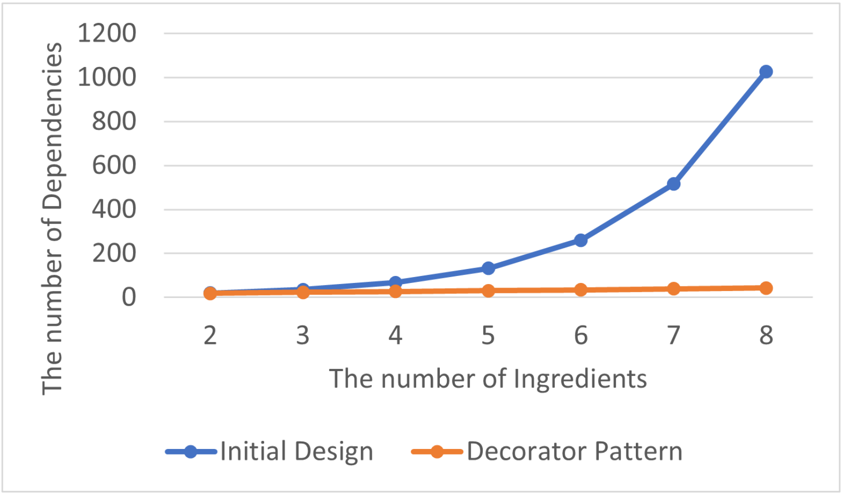


Figure 12. Decorator Pattern: total encapsulation dependencies.

Table 10. Avg. encapsulation dependencies per class in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	4	3.8
3	4	3.833
4	4	3.857
5	4	3.875
6	4	3.889
7	4	3.9
8	4	3.909

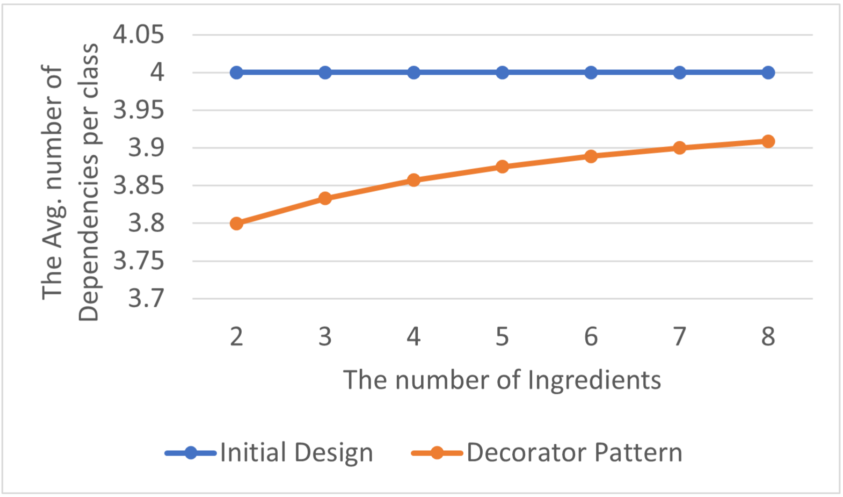


Figure 13. Decorator Pattern: avg. encapsulation dependencies per class.

4.2.2. Abstraction Dependency Relationships in Decorator Pattern

The total number of abstraction dependency relationships is shown in Table 11, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 12. By visualizing the experimental results as line charts, it can be observed that the number of abstraction dependency relationships in the program that does not contain the Decorator Pattern is significantly higher than that in the one with the Decorator Pattern, as shown in Figures 14 and 15. As mentioned in Section 3.1.1, high levels of abstraction dependency relationships make program maintenance and management challenging.

Table 11. Total abstraction dependencies in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	12	6
3	24	7
4	48	8
5	96	9
6	192	10
7	384	11
8	768	12

Table 12. Avg. abstraction dependencies per class in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	2.4	1.2
3	2.667	1.167
4	2.824	1.143
5	2.909	1.125
6	2.954	1.111
7	2.977	1.1
8	2.988	1.091

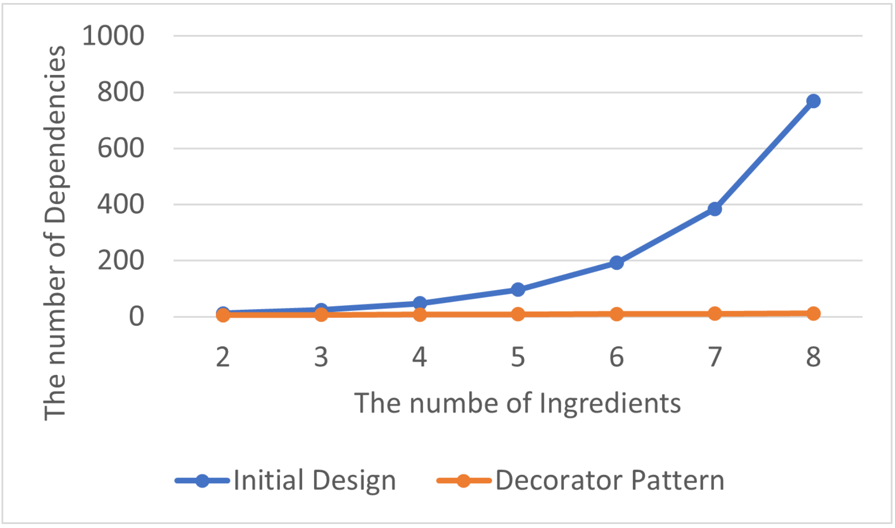


Figure 14. Decorator Pattern: total abstraction dependencies.

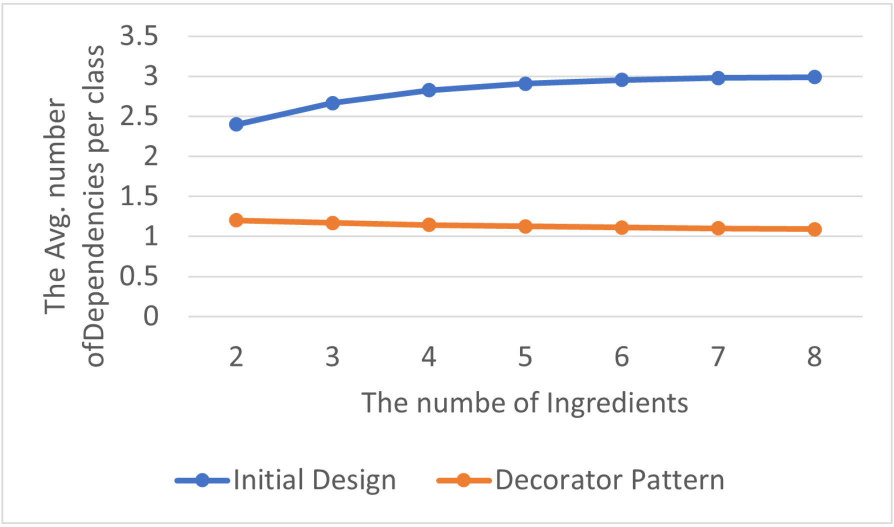


Figure 15. Decorator Pattern: avg. abstraction dependencies per class.

4.2.3. Delegation Dependency Relationships in Decorator Pattern

The total number of delegation dependency relationships is shown in Table 13, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 14. Although programs using the Decorator Pattern have more delegation dependency relationships, their number still remains within a reasonable range, as shown in Figures 16 and 17. As mentioned in Section 3.1.1, moderate delegation can make the program easier to evolve and modify while also enhancing modularity and readability.

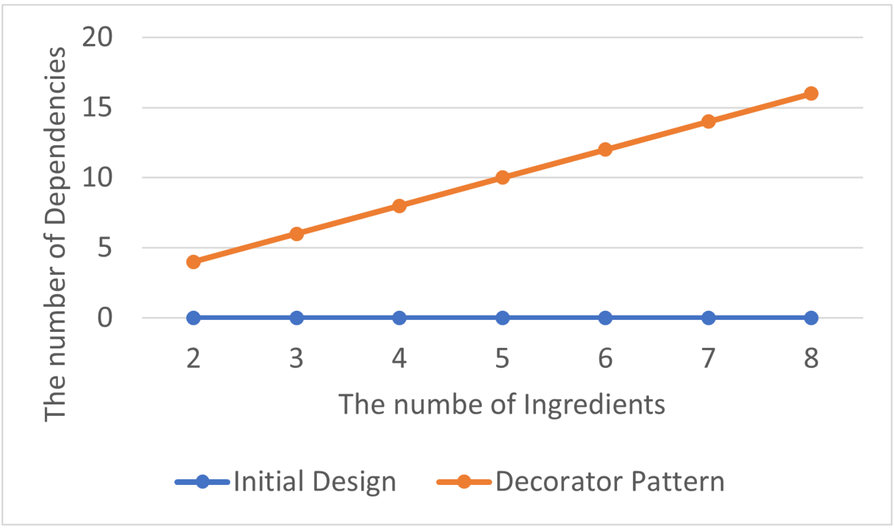


Figure 16. Decorator Pattern: total delegation dependencies.

Table 13. Total delegation dependencies in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	0	4
3	0	6
4	0	8
5	0	10
6	0	12
7	0	14
8	0	16

Table 14. Avg. delegation dependencies per class in Decorator Pattern.

Ingredient Number	Initial Design	Decorator Pattern
2	0	0.8
3	0	1
4	0	1.143
5	0	1.25
6	0	1.333
7	0	1.4
8	0	1.455

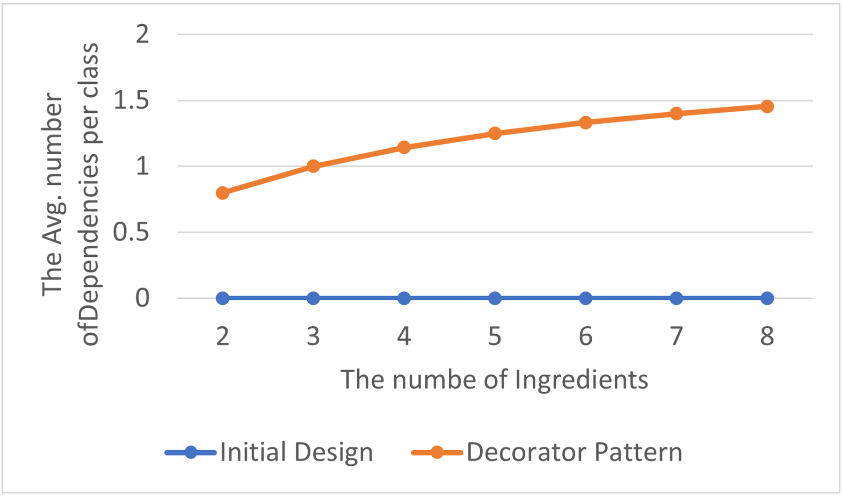
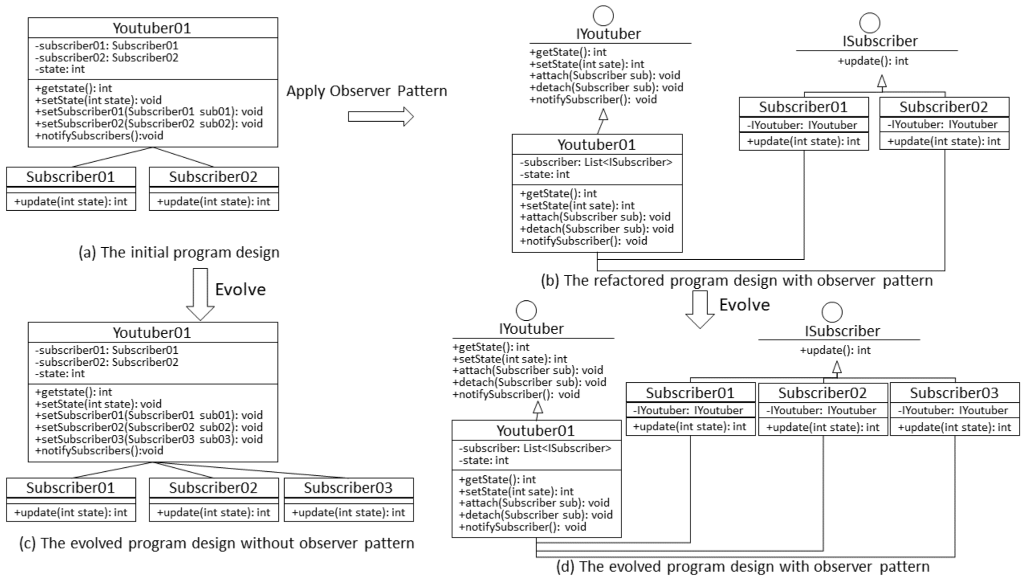


Figure 17. Decorator Pattern: avg. delegation dependencies per class.

4.3. Observer Pattern

For the experiment involving the Observer Pattern, this study designed a simple subscription notification system as an example, with its program design illustrated in Figure 18. When the Youtuber class updates its status, the Subscriber class will be notified, as shown in Figure 18a. However, the Youtuber class and Subscriber class are highly coupled in this design, and the Youtuber must know which Subscribers it has in order to notify them. Furthermore, as the number of Subscribers increases, the program will also become overly large.



Youtuber01
-subscriber01: Subscriber01
-subscriber02: Subscriber02
-state: int
+getstate(): int
+setState(int state): void
+setSubscriber01(Subscriber01 sub01): void
+setSubscriber02(Subscriber02 sub02): void
+notifySubscribers():void
Subscriber01
+update(int state): int
Subscriber02
+update(int state): int
Apply Observer Pattern
(a) The initial program design
Evolve
IYoutuber
+getState(): int
+setState(int sate): int
+attach(ISubscriber sub): void
+detach(Subscriber sub): void
+notifySubscriber(): void
Youtuber01
-subscriber: List<ISubscriber>
-state: int
ISubscriber
+update(): int
-IYoutuber: IYoutuber
+update(int state): int
(b) The refactored program design with observer pattern
Subscriber03
+setSubscriber03(Subscriber03 sub03): void
(c) The evolved program design without observer pattern
(d) The evolved program design with observer pattern

Figure 18. Observer program design.

This issue can be improved by using the Observer Pattern. The Youtuber can notify all Subscribers without needing to know the details of individual Subscribers through the interfaces IYoutuber and ISubscriber. Additionally, when adding new Subscribers, there is no need for repeated modifications to the existing program, as shown in Figure 18b. Finally, we depict the scenarios before and after system evolution following the refactoring in Figure 18c,d.

The following section presents our experimental results for the dependency relationships (encapsulation, abstraction, and delegation) in the Observer Pattern.

4.3.1. Encapsulation Dependency Relationships in Observer Pattern

The total number of encapsulation dependency relationships is shown in Table 15, and the average number of encapsulation dependency relationships within each class in the program is displayed in Table 16. By visualizing the experimental results as line charts, it can be observed that as the program extends into subsequent stages, using the Observer Pattern allows the refactored program to have a lower number of encapsulation dependency relationships compared to the initial program. It exhibits better performance in terms of system complexity, as shown in Figures 19 and 20.

Table 15. Total encapsulation dependencies in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	50	60
3	66	67
4	82	74
5	98	81
6	114	88
7	130	95
8	146	102
9	162	109
10	178	116
11	194	123
12	210	130
13	226	137
14	242	144
15	258	151

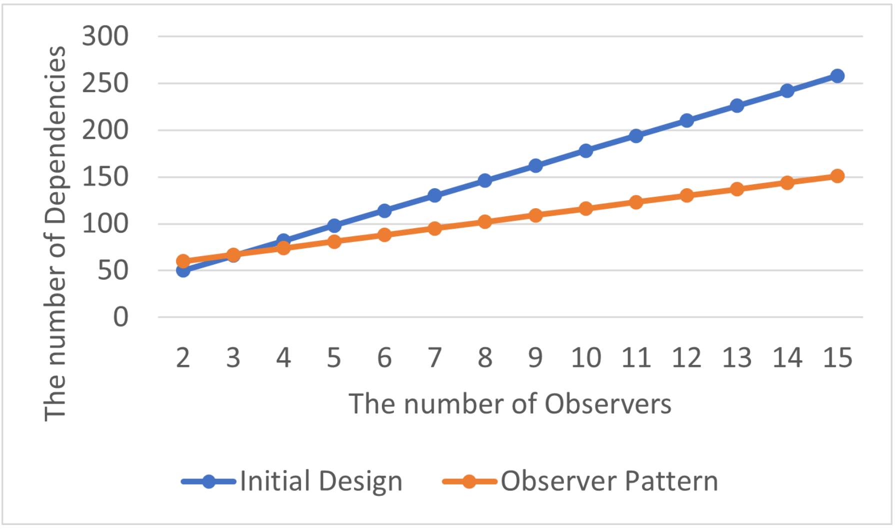


Figure 19. Observer Pattern: total encapsulation dependencies.

Table 16. Avg. encapsulation dependencies per class in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	16.667	12
3	16.5	11.167
4	16.4	10.571
5	16.333	10.125
6	16.286	9.778
7	16.25	9.5
8	16.222	9.273
9	16.2	9.083
10	16.182	8.923
11	16.167	8.786
12	16.154	8.667
13	16.143	8.563
14	16.133	8.471
15	16.125	8.389

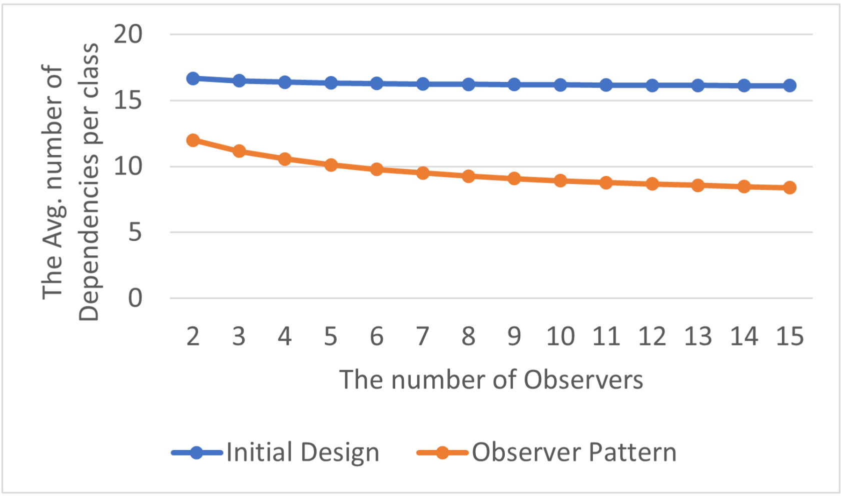


Figure 20. Observer Pattern: avg. encapsulation dependencies per class.

4.3.2. Abstraction Dependency Relationships in Observer Pattern

The total number of abstraction dependency relationships is shown in Table 17, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 18. By visualizing the experimental results as line charts, it can be observed that appropriately elevating the program's abstraction level not only reduces the dependence on the program but also makes it easier to extend and maintain, as shown in Figures 21 and 22.

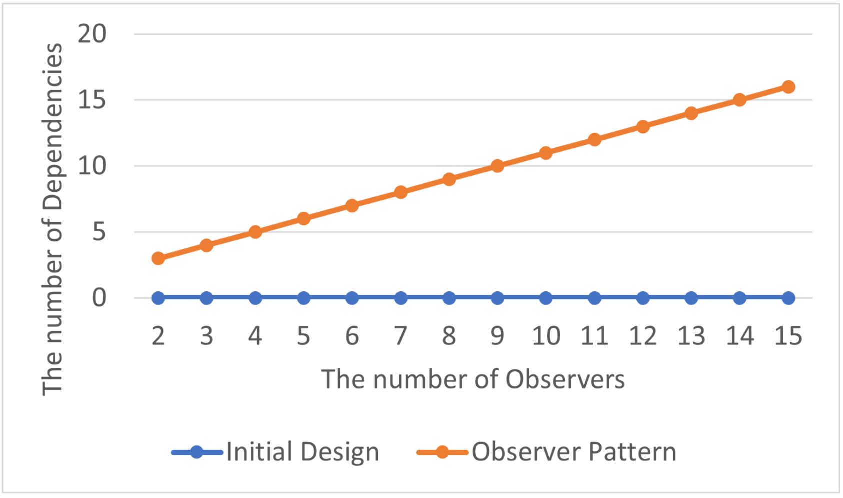


Figure 21. Observer Pattern: total abstraction dependencies.

Table 17. Total abstraction dependencies in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	0	3
3	0	4
4	0	5
5	0	6
6	0	7
7	0	8
8	0	9
9	0	10
10	0	11
11	0	12
12	0	13
13	0	14
14	0	15
15	0	16

Table 18. Avg. abstraction dependencies per class in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	0	0.6
3	0	0.667
4	0	0.714
5	0	0.75
6	0	0.778
7	0	0.8
8	0	0.818
9	0	0.833
10	0	0.846
11	0	0.857
12	0	0.867
13	0	0.875
14	0	0.882
15	0	0.889

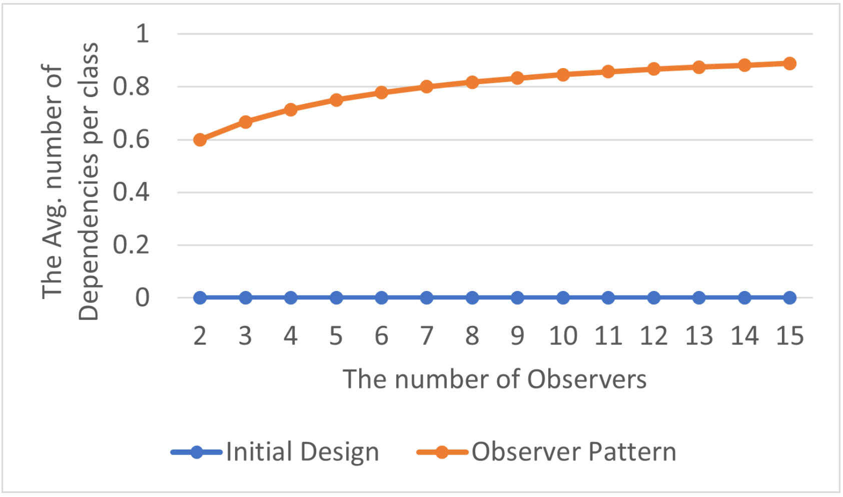


Figure 22. Observer Pattern: avg. abstraction dependencies per class.

4.3.3. Delegation Dependency Relationships in Observer Pattern

The total number of delegation dependency relationships is shown in Table 19, and the average number of abstraction dependency relationships within each class in the program is displayed in Table 20. Although the use of the Observer Pattern may result in more delegation dependency relationships during the early stages of system development, as its evolution progresses, these dependency relationships are still reduced compared to the beginning, as shown in Figures 23 and 24. This method can not only maintain the program's structure but also enhance its maintainability and scalability.

Table 19. Total delegation dependencies in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	14	20
3	19	22
4	24	24
5	29	26
6	34	28
7	39	30
8	44	32
9	49	34
10	54	36
11	59	38
12	64	40
13	69	42
14	74	44
15	79	46

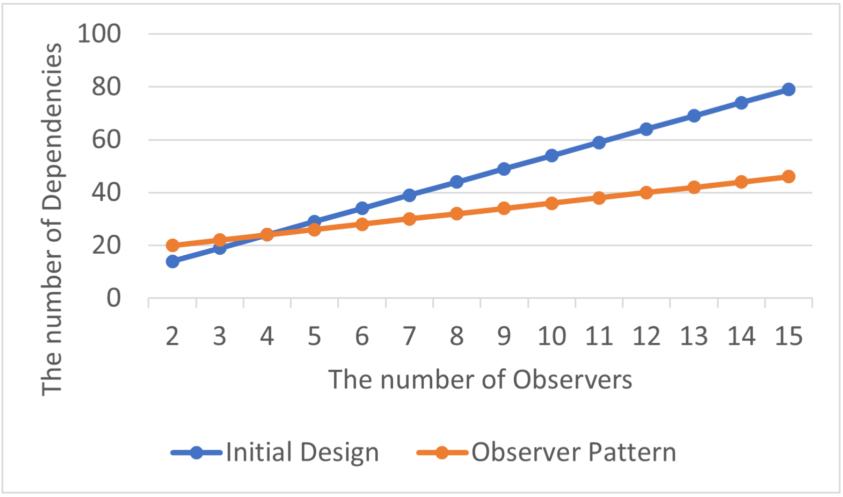


Figure 23. Observer Pattern: total delegation dependencies.

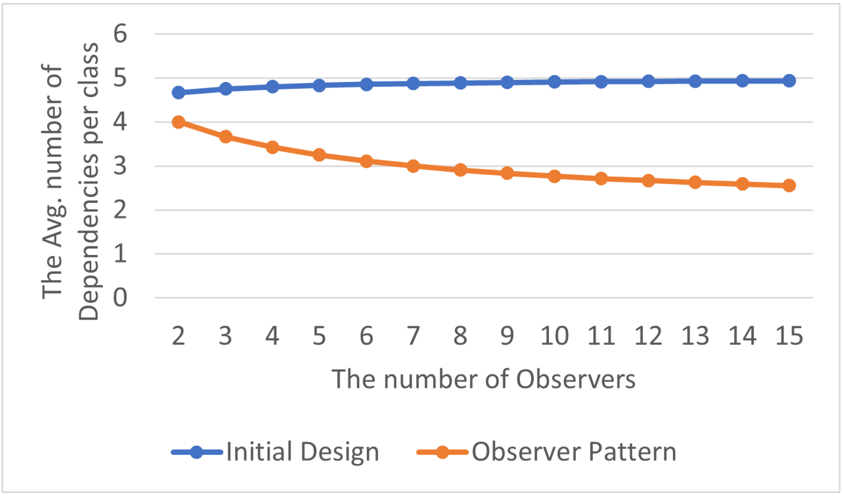


Figure 24. Observer Pattern: avg. delegation dependencies per class.

Table 20. Avg. delegation dependencies per class in Observer Pattern.

Subscriber Number	Initial Design	Observer Pattern
2	4.667	4
3	4.75	3.667
4	4.8	3.429
5	4.833	3.25
6	4.857	3.111
7	4.875	3
8	4.889	2.909
9	4.9	2.833
10	4.909	2.769
11	4.917	2.714
12	4.923	2.667
13	4.929	2.625
14	4.933	2.588
15	4.938	2.556

4.4. Threats to Validity

This research opted to examine only a restricted set of design patterns in the experiment and analyzed the results based on a subset of dependency relationships. Expanding the scope of the experiment would bolster the assertions. Furthermore, it is essential to select high-quality studies from the literature as references for the experimental sample programs and tools to yield more comprehensive results. In summary, there is still ample room for improvement regarding both credibility and comprehensiveness in this study.

5. Conclusions

The purpose of this study was to analyze the impact of design patterns on system quality during software system evolution. We employed dependency relationships as a gauge of program complexity. By designing a tool to compute the number of these dependency relationships, it became possible to analyze the changes in these dependencies throughout the evolution process. The evaluation of the three frequently utilized design patterns proposed by the Gang of Four (GoF) as experimental examples led to the following conclusions.

Firstly, the use of design patterns effectively reduces the dependencies generated during system evolution. By introducing appropriate abstractions and decoupling mechanisms, design patterns help reduce direct inter-module dependencies, thereby reducing system complexity. This reduction in dependency has a positive impact by enhancing system scalability and readability, making the system more amenable to modifications and extensions.

Secondly, design patterns also have a positive impact on system quality. Through the judicious use of design patterns, the system's structure becomes clearer and easier to understand, thus enhancing code readability. Furthermore, design patterns provide a generic solution that enables developers to more easily comprehend and design system architectures, thereby increasing system maintainability.

In summary, the results of this study indicate that design patterns play a significant role in software system evolution. By reducing dependencies and enhancing system quality, design patterns provide software developers with effective methods and guidelines to make systems more scalable, readable, and maintainable. In future research, it would be valuable to explore additional design patterns, conduct in-depth, real-world case studies, and develop refined methods for measuring dependencies, contributing to a more comprehensive understanding of the broader impact and benefits of diverse design patterns in software evolution.

Author Contributions: Conceptualization, K.-H.H.; Software, H.-C.S.-T.; Writing—review & editing, K.-H.H. and C.-H.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Nanthaamornphong, A.; Wetprasit, R. A controlled experiment: Do Visitor patterns improve design simplicity? In Proceedings of the 2014 8th Malaysian Software Engineering Conference (MySEC), Langkawi, Malaysia, 23–24 September 2014; pp. 90–95. [CrossRef]
2. Pree, W. *Design Patterns for Object-Oriented Software Development*; ACM Press/Addison-Wesley Publishing Co.: Boston, MA, USA, 1995.
3. Celikkan, U.; Bozoklar, D. A Consolidated Approach for Design Pattern Recommendation. In Proceedings of the 2019 4th International Conference on Computer Science and Engineering (UBMK), Samsun, Turkey, 11–15 September 2019; pp. 1–6. [CrossRef]
4. Khomh, F.; Guéhéneuc, Y.G. Do Design Patterns Impact Software Quality Positively? In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, 1–4 April 2008; pp. 274–278. [CrossRef]
5. Ampatzoglou, A.; Chatzigeorgiou, A.; Charalampidou, S.; Avgeriou, P. The Effect of GoF Design Patterns on Stability: A Case Study. *IEEE Trans. Softw. Eng.* **2015**, *41*, 781–802. [CrossRef]
6. Bichsel, B.; Raychev, V.; Tsankov, P.; Vechev, M. Statistical Deobfuscation of Android Applications. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 343–355. [CrossRef]
7. Huang, S.W. Towards a Solution to IoT Interoperability through Reverse Engineering. Master's Thesis, National Taiwan University, Taipei, Taiwan, 2017.
8. Huang, H.Y. Atomic Services Generation from Java-Based Open Source Software. Master's Thesis, National Taiwan University, Taipei, Taiwan, 2019.
9. Iyapparaja, M.; Sureshkumar, S. Coupling and cohesion metrics in Java for adaptive reusability risk reduction. In Proceedings of the IET Chennai 3rd International Conference on Sustainable Energy and Intelligent Systems (SEISCON 2012), Tiruchengode, India, 27–29 December 2012. [CrossRef]
10. Wand, A. Reuse Metrics and Assessment in Component-Based Development. In Proceedings of the 6th IASTED International Conference on Software Engineering and Applications, Cambridge, MA, USA, 4–6 November 2002; pp. 583–588.
11. Wedyan, F.; Abufakher, S. Impact of design patterns on software quality: A systematic literature review. *IET Softw.* **2020**, *14*, 1–17. [CrossRef]
12. McNatt, W.B.; Bieman, J.M. Coupling of design patterns: common practices and their benefits. In Proceedings of the 25th Annual International Computer Software and Applications Conference, COMPSAC 2001, Chicago, IL, USA, 8–12 October 2001; pp. 574–579. [CrossRef]
13. Prechelt, L.; Unger, B.; Tichy, W.F.; Brossler, P.; Votta, L.G. A controlled experiment in maintenance: Comparing design patterns to simpler solutions. *IEEE Trans. Softw. Eng.* **2001**, *27*, 1134–1144. [CrossRef]
14. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995.
15. van Bruggen Matozoid, D. javaparser. GitHub Repository. Retrieved from GitHub. Available online: https://github.com/javaparser/javaparser (accessed on 12 July 2023).
16. Tahvildari, L.; Kontogiannis, K.; Mylopoulos, J. Quality-driven software re-engineering. *J. Syst. Softw.* **2003**, *66*, 225–239. [CrossRef]
17. Ampatzoglou, A.; Chatzigeorgiou, A. Evaluation of object-oriented design patterns in game development. *Inf. Softw. Technol.* **2007**, *49*, 445–454. [CrossRef]
18. Maruyama, K.; Shima, K.i. Automatic method refactoring using weighted dependence graphs. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, USA, 16–22 May 1999; pp. 236–245. [CrossRef]
19. Terrence, P. *The Definitive ANTLR Reference*; The Pragmatic Bookshelf: Raleigh, NC, USA, 2007.
20. Fischer, G.; Lusiardi, J.; Von Gudenberg, J.W. Abstract Syntax Trees—and their Role in Model Driven Software Development. In Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007), Cap Esterel, France, 25–31 August 2007; pp. 38–38. [CrossRef]
21. Fauzi, E.; Hendradjaya, B.; Sunindyo, W.D. Reverse engineering of source code to sequence diagram using abstract syntax tree. In Proceedings of the 2016 International Conference on Data and Software Engineering (ICoDSE), Denpasar, Indonesia, 26–27 October 2016; pp. 1–6. [CrossRef]
22. Tao, G.; Guowei, D.; Hu, Q.; Baojiang, C. Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree. In Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies, Xi'an, China, 9–11 September 2013; pp. 714–719. [CrossRef]