-LIVERPOOL UNIVERSITY

COURSEWORK SUBMISSION COVER Page

Group Number						
Students' ID Number	2143132	2142298	2142253	2143271		
Module Code	CPT304					
Assignment Title	Assignment 1					
Submission Deadline	11/4/2025					

By uploading this coursework submission with this cover page, we certify the following:

We have read and understood the definitions of collusion, copying, plagiarism, and dishonest use of data as outlined in the Academic Integrity -Liverpool University.
This work is **entirely** our own, original work produced specifically for this assignment. It does not misrepresent the work of another group or institution as our own.
This work is not the product of unauthorized collaboration between ourselves and others.
This work has not been shared wholly or in part outside of the group.
It is a submission that has not been previously published, or submitted to any modules.
Students who would like to submit the same or similar work from previous years to the current module or other modules must receive written permission from all instructors involved in advance of the assignment due date.
All group members are **equally** and **collectively** responsible for the **entire** submission. Violations of academic integrity including failure to monitor group member contribution originality constitutes negligence.
Unreported suspicious academic misconduct by one group member will be attributed to ALL members in terms of penalties. **ALL members share the responsibilities**.
Use of generative AI is strictly prohibited for all assessments involved in this module.

We understand collusion, plagiarism, dishonest use of data, and submission of procured work are serious academic misconducts. **All** group members are held **jointly accountable** for the integrity of the **entire** submission. By uploading or submitting this cover page, we acknowledge that we are jointly subject to penalties and disciplinary actions if we are found to have committed such acts.

~~~~~~

We acknowledge that the university late submission policy will be applied if applicable.

~~~~~~

Please list the ID number of any group member NOT contributing to the submission:

___

Please indicate whether based on your individual contributions you meet the learning outcomes covered by this coursework and confirm your submission meets with all above requirements by signing your name, handwritten in ink, in English (Foreign students) or PINYIN (local students):

Student ID	(tick) box to confirm you contributed to all cw tasks	(tick) box to confirm you meet all LOs covered by this cw	Signed
2143132	✓	✓	顾齐斌
2142298	✓	✓	倪一
2142253	✓	✓	张琛亚婷
2143271	✓	✓	戴侦明

10/4/2025

Analyzing "No Silver Bullet" and Design Patterns

Qibin Gu	Yi Ni	Liangyuting Zhang	Zhenyu Dai
2143132	2142298	2142253	2143217

April 11, 2025

Abstract

This report delves into the fundamental challenges of software engineering as outlined in Fred Brooks' influential article "No Silver Bullet" and explores how the Decorator and Observer design patterns can effectively address these challenges. The inherent difficulties in software development, including complexity, conformity, changeability, and invisibility, often lead to project overruns and compromised quality. The Decorator pattern is analyzed through a case study of a coffee shop order system, demonstrating its ability to reduce complexity and manage changes dynamically by avoiding subclass explosion and adhering to the Open/Closed Principle. The Observer pattern is examined in the context of a weather monitoring system, highlighting its role in decoupling subjects from observers, enhancing system transparency, and supporting extensibility. The report concludes that integrating these design patterns into software engineering practices can significantly improve software productivity, maintainability, and quality, and recommends their continued use as part of a broader strategy to incrementally and sustainably tackle the inherent complexities of software development.

1 Introduction

Fred Brooks' seminal article, "No Silver Bullet", emphasizes fundamental challenges in software engineering that cannot be solved by any single technological breakthrough or methodology. These inherent difficulties—complexity, conformity, changeability, and invisibility—often result in projects exceeding budgets, delayed schedules, and compromised quality. This report explores in detail how two prominent design patterns, the Composite Pattern and the Observer Pattern, can effectively mitigate these critical challenges.

1.1 Key Challenges from "No Silver Bullet"

Brooks identifies the following four primary inherent challenges in software engineering [1]:

1. Complexity

   Complexity contains state complexity, function complexity and structure complexity

   As the system grows in scale, the number of possible states increases exponentially, far beyond what developers can fully enumerate and understand, which makes it extremely difficult to predict and test the system's conditions, directly contributing to its unreliability.

   At the same time, as the functionality becomes more complex, invoking becomes increasingly cumbersome, making it challenging for users to quickly and accurately access the desired features, thereby affecting usability.

   Moreover, the growing complexity of the software's structure means that adding and extending features often impact other parts of the system, easily leading to side effects, even affecting the stability of the whole system. In addition, complex structures often own hidden states that cannot be directly observed; if these states are not captured in a timely manner, they can become security trapdoors in the system.

2. Conformity

   Conformity refers to the requirement for a software system to adhere to external interfaces, standards, or past systems

   If the software is the latest system to appear on the market, it needs to be compatible with existing system interfaces; Sometimes the software is considered to be the easiest to achieve conformity across a variety of existing systems. This complexity, brought about by external interface requirements, cannot be eliminated by simply redesigning the software internally.

3. Changeability

   Changeability refers to the need for the software to make corresponding adjustments according to internal and external changes

   Software is essentially function embedded in system, and it is easier to modify than other physical products. In addition, the software is embedded in a constantly changing cultural matrix and may need to be adjusted at any time due to new needs, or other changes in the external environment.

   Successful software will inevitably undergo continuous evolution. When the software proves the practicality of its basic functions, users often apply it to new scenarios, resulting in demand for extended functions; secondly, when new computers, disks, etc. appear, the software needs to make adjustments to adapt to the new operating environment. These constant internal and external changes make the variability of software an indispensable feature.

4. Invisibility

   Invisibility refers to the fact that software does not have an inherent physical form

   As an abstract structure, software is different from buildings that can be visually expressed through blueprints, and needs to be represented by several directed diagrams with different representations. These diagrams are usually more complex, non-planar, and lack a natural hierarchical order. In order to manage and understand this complexity, developers usually have to artificially impose hierarchies through techniques such as "link cutting".

2 Pattern Analysis

2.1 Decorator Pattern

- Definition and Importance

  A structural design pattern called Decorator makes it easier to dynamically add behaviours to particular objects without changing how other objects of the same class behave. This technique is useful because it may improve object functions during runtime, offering a versatile substitute for sub-classes in the extension of functionalities.

- Justification for Selection

  The Decorator pattern, as outlined in the seminal work [1], serves as a dynamic alternative to subclassing for extending object functionality. By wrapping objects with decorators that adhere to the same interface, developers can add or remove features at runtime without modifying the core class structure. This approach enhances flexibility, reduces code duplication, and aligns with the Open/Closed Principle by allowing systems to evolve without altering existing code. The pattern is particularly advantageous in scenarios requiring selective enhancement, such as adding features to specific objects in a complex application.

2.2 Observer Pattern

- Definition and Importance

  The Observer Pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Justification for Selection

  The Observer Pattern decouples subjects from observers, enhancing system flexibility and scalability. It effectively addresses the challenges of complexity and invisibility discussed in Brooks' article.

3 Case study

3.1 Decorator Pattern

3.1.1 Case Study: Decorator Pattern in a Coffee Shop Order System

The Decorator pattern is widely used in applications requiring dynamic functionality enhancement. Consider a coffee shop order system where beverages can have various condiments (e.g., milk, soy,

whipped cream). Each condiment adds cost and description, but hardcoding combinations would lead to combinatorial explosion. The Decorator pattern elegantly addresses this challenge.

3.1.2 Code Implementation

See the code implementation in Listing 1.

Listing 1: Code implementation of the coffee shop order system.

```
// Core interface for beverages
interface Beverage {
    String getDescription();
    double cost();
}

// Concrete component: basic beverage
class Espresso implements Beverage {
    @Override
    public String getDescription() { return "Espresso"; }
    @Override
    public double cost() { return 1.99; }
}

// Decorator abstract class
abstract class CondimentDecorator implements Beverage {
    protected Beverage beverage;
    public abstract String getDescription();
}

// Concrete decorator: Milk
class Milk extends CondimentDecorator {
    public Milk(Beverage b) { beverage = b; }
    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Milk";
    }
    @Override
    public double cost() { return beverage.cost() + 0.50; }
}

// Usage
Beverage order = new Espresso();
order = new Milk(order); // Add milk dynamically
System.out.println(order.getDescription() + ": $" + order.cost());
```

3.1.3 Analysis of Challenges from "No Silver Bullet"

- Complexity

  Brooks highlights complexity as an inherent challenge in software. The Decorator pattern reduces complexity by avoiding subclass explosion. Instead of creating classes for every possible combination (e.g., 'EspressoWithMilk', 'HouseBlendWithSoyAndWhip'), decorators dynamically wrap objects, minimizing code duplication and maintaining scalability.

- Change Management

  Software must evolve over time. The Decorator pattern supports the Open/Closed Principle, allowing new condiments to be added without modifying existing code. This addresses Brooks' concern about managing changes in requirements, as functionalities can be extended incrementally without destabilizing the core system.

- Consistency

  Brooks emphasizes the need for consistent abstractions. The Decorator pattern maintains a uniform interface between components and decorators, ensuring all objects adhere to the 'Beverage' contract. This consistency simplifies maintenance and avoids the rigidity of subclassing, where changes propagate across multiple classes.

The Decorator pattern effectively tackles challenges like complexity, change management, and consistency by enabling dynamic, modular enhancements. It aligns with Brooks' assertion that while no single solution exists, thoughtful design patterns can mitigate software development challenges.

3.2 Observer Pattern

3.2.1 Case Study: Weather Monitoring System

In a weather monitoring system, multiple display devices (e.g., temperature gauges, humidity meters) need real-time updates on weather data. Hardcoding update logic in each display device would make the system difficult to maintain.

3.2.2 Code Implementation

See the code implementation in Listing 2.

Listing 2: Code implementation of the weather monitoring system.

```
// Subject Interface
interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

// Observer Interface
interface Observer {
    void update(float temperature, float humidity);
}

// Concrete Subject: Weather Data
class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    @Override
```

```
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity);
        }
    }

    public void setMeasurements(float temperature, float humidity) {
        this.temperature = temperature;
        this.humidity = humidity;
        notifyObservers();
    }
}

// Concrete Observer: Temperature Display
class TemperatureDisplay implements Observer {
    @Override
    public void update(float temperature, float humidity) {
        System.out.println("Current temperature: " + temperature);
    }
}

// Usage
WeatherData weatherData = new WeatherData();
Observer tempDisplay = new TemperatureDisplay();
weatherData.registerObserver(tempDisplay);
weatherData.setMeasurements(25.0f, 60.0f);
```

3.2.3 Analysis of Challenges from "No Silver Bullet"

- Complexity

  The Observer Pattern reduces complexity by decoupling subjects from observers, eliminating hardcoded update logic in each observer.

- Invisibility

  The Observer Pattern enhances system transparency by providing a uniform interface and notification mechanism, making state changes more visible.

- Change Management

  New observers can be added by simply implementing the observer interface and registering with the subject, supporting system extensibility without modifying existing code.

4 Group Task Allocation and Version Control

Team responsibilities were allocated clearly to optimize productivity and ensure accountability and The team employed Git for version control, providing transparency, accountability, and traceability throughout the project. Each member regularly committed their contributions, which facilitated seamless integration and easy tracking of changes and progress.

5 Conclusion

Through comprehensive analysis of Brooks' article "No Silver Bullet" and practical applications of the Composite and Observer Patterns, we determined effective strategies to mitigate core software engineering challenges. The Composite Pattern dramatically simplifies the management of complex hierarchical software structures, addressing the critical issue of complexity. Conversely, the Observer Pattern promotes adaptability and responsiveness, effectively handling frequent and continuous changes in software requirements.

Integrating such design patterns into regular software engineering practice can result in considerable improvements in software productivity, maintainability, and quality. Moving forward, software development should continue leveraging proven design patterns as part of a broader strategy to tackle inherent complexities incrementally and sustainably.

References

[1] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, April 1987.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Boston, MA: Addison-Wesley, 1994.

CPT304 Assignment 1

Individual Contribution

Group Number/Name:

Name	ID Number	Contribution (%)
1. Qibin Gu	2143132	25%
2. Yi Ni	2142298	25%
3. Liangyuting Zhang	2142253	25%
4. Zhenyu Dai	2143271	25%
5.		

Signed (physically) by all members:

顾齐文武 倪一 张梁玉婷 戴依羽