|  |
| --- |
|  |
| Software Design Patterns |
| Luke Foley T00224345 |
|  |

Table of Contents

[Introduction 2](#_Toc183544690)

[Existing System Overview 3](#_Toc183544691)

[Main problems 4](#_Toc183544692)

[Design and Architectural Pattern Exploration 4](#_Toc183544693)

[Pros and Cons of Layered Architecture 4](#_Toc183544694)

[Pros and Cons of Builder Pattern 6](#_Toc183544695)

[Pros and Cons of Strategy Pattern 8](#_Toc183544696)

[Pros and Cons of Dependency Injection 10](#_Toc183544697)

[Pros and Cons of Façade Pattern 10](#_Toc183544698)

[System Refactoring 10](#_Toc183544699)

[Tools used 10](#_Toc183544700)

[Implementation of Layered Architecture 10](#_Toc183544701)

[Implementation of Dependency Injection and Centralising Form Navigation 13](#_Toc183544702)

[Implementation of Builder Pattern 13](#_Toc183544703)

[Implementation of Strategy Pattern 14](#_Toc183544704)

[Implementation Façade Pattern 14](#_Toc183544705)

[Further Improvements 15](#_Toc183544706)

[Testing and Validation 15](#_Toc183544707)

[Conclusion 15](#_Toc183544708)

# Introduction

The ToolSYS application is a management system designed to streamline operations for a tool rental business. The application was developed during the second year "Requirements Engineering" and "Software Engineering Project" modules and was built using Visual Studio with Windows Forms for UI, and an oracle database.

A screenshot of a computer

Description automatically generated

# Existing System Overview

The system is organised into five main sections: Rates, Tools, Customers, Rentals, and Admin.

* Rates Module – This module allows users to define categories of tools and their associated rental rates, ensuring that rental fees are calculated consistently.
* Tools Module – This module manages all aspects of tool management, including adding new tools, updating their details, removing unavailable tools, and viewing the complete inventory.
* Customers Module – This section focuses on customer information, allowing users to add, update, and view customer details.
* Rentals Module – This module facilitates the tool rental process, allowing users to rent tools, return them, and view rental records.
* Admin Module – This section provides analytical features, such as annual revenue reporting and tool type usage analysis, offering insights into business performance.

Functional Requirements Diagram:

A diagram of tools

Description automatically generated

While the original system successfully fulfilled its primary purpose, it was developed with a focus on functionality rather than architectural principles or scalability. This made it a suitable choice for this assignment, with opportunities to implement design and architectural patterns.

## Main problems

Limited Scalability: Adding new features, such as customer categorization or dynamic discounts, require extensive modifications to the existing codebase. This inflexibility made it difficult to adapt the system to evolving requirements

Violation of SRP: Several classes handled multiple concerns. Forms managed UI interactions, executed business logic, and sometimes communicated directly with the database. This mix of responsibilities made the code challenging to debug and reuse.

Tightly Coupled: The program has little separation. The UI, business logic, and database are very closely linked, with business logic in the UI

# Design and Architectural Pattern Exploration

## Pros and Cons of Layered Architecture

**Pros**

1. **Separation of Concerns**
   * Layers are designed to handle specific responsibilities, such as presentation, business logic, or data access. This separation makes the system easier to understand, maintain, and extend (Bass, Clements, & Kazman, 2003).
2. **Ease of Maintenance**
   * Changes in one layer typically have minimal impact on others, provided the interfaces are well-defined. This modularity simplifies debugging and updating parts of the application (Fowler, 2002).
3. **Scalability**
   * Components within layers can be scaled independently, allowing optimization of performance. For example, the database layer can be scaled separately from the business logic (Evans, 2004).
4. **Reusability**
   * Layers can often be reused in different systems. For example, the data access layer may be reused across different projects that use the same database schema (Pressman, 2005).
5. **Team Collaboration**
   * Different development teams can work on different layers simultaneously, increasing productivity and reducing development time (Larman, 2004).
6. **Testability**
   * The isolation of responsibilities allows for unit testing of each layer independently, ensuring reliability before integration (Martin, 2003).

**Cons**

1. **Performance Overhead**
   * Communication between layers adds overhead, which may lead to latency, especially in real-time or resource-constrained environments (Bass et al., 2003).
2. **Rigidity in Layered Rules**
   * Strict adherence to layer boundaries can become a bottleneck when cross-layer communication is necessary, reducing flexibility (Fowler, 2002).
3. **Potential for Over-Engineering**
   * For smaller projects, the added complexity of defining and managing multiple layers may outweigh the benefits, making the architecture unnecessarily cumbersome (Evans, 2004).
4. **Difficulty in Modifying Cross-Cutting Concerns**
   * Features like logging, security, or transaction management, which affect multiple layers, are harder to implement and maintain due to the strict separation of layers (Martin, 2003).
5. **Dependency on the Lower Layers**
   * A failure in a lower layer (e.g., the database) can cascade upwards, causing issues in the entire system. This can make recovery and fault isolation challenging (Larman, 2004).
6. **Tight Coupling Between Adjacent Layers**
   * Even though layers are loosely coupled overall, adjacent layers are inherently dependent on each other. This can lead to challenges when changes are made in these interfaces (Pressman, 2005).

**References**

* Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley.
* Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
* Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
* Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Pearson Education.
* Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
* Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

## Pros and Cons of Builder Pattern

**Pros**

1. **Encapsulation of Construction Logic**
   * The pattern encapsulates the construction process, making the creation of complex objects more manageable and reducing the complexity in client code (Gamma, Helm, Johnson, & Vlissides, 1994).
2. **Improved Readability and Maintainability**
   * The step-by-step construction approach results in more readable code, as each step in the construction process is explicitly defined (Fowler, 2002).
3. **Support for Immutability**
   * By building objects step-by-step, the Builder pattern supports creating immutable objects, which are particularly beneficial in multi-threaded environments (Bloch, 2008).
4. **Flexibility in Object Construction**
   * The same Builder can produce different representations or configurations of an object, allowing for greater flexibility (Gamma et al., 1994).
5. **Reduction of Constructor Overload**
   * Instead of using multiple constructors to handle different configurations, the Builder pattern organizes the configuration steps more cleanly (Martin, 2003).
6. **Ease of Extensibility**
   * Adding new steps or attributes to the construction process is straightforward, which makes it easier to extend the functionality without modifying existing code (Evans, 2004).

**Cons**

1. **Increased Complexity**
   * The implementation of the Builder pattern involves creating additional classes (e.g., Director and Builder), which may introduce unnecessary complexity for simple objects (Gamma et al., 1994).
2. **Overhead for Simple Objects**
   * For objects that don’t require complex construction, using the Builder pattern can lead to redundant code and unnecessary boilerplate (Fowler, 2002).
3. **Dependency on Director**
   * In some implementations, the Director class might tightly couple the construction process, reducing flexibility in object creation (Martin, 2003).
4. **Difficult to Integrate with Existing Systems**
   * Introducing the Builder pattern into an existing system may require refactoring and reorganization, which could be resource-intensive (Bloch, 2008).
5. **Not Always Intuitive**
   * For developers unfamiliar with the pattern, understanding and implementing the Builder pattern may require a steep learning curve (Gamma et al., 1994).

**References**

* Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
* Bloch, J. (2008). *Effective Java* (2nd Edition). Addison-Wesley.
* Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
* Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
* Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

This balanced view provides insight into the Builder pattern's suitability for applications with varying levels of object complexity.

## Pros and Cons of Strategy Pattern

**Pros**

1. **Promotes the Open-Closed Principle**
   * The Strategy pattern enables adding new strategies without modifying existing code, adhering to the principle of keeping code open for extension but closed for modification (Gamma, Helm, Johnson, & Vlissides, 1994).
2. **Encapsulation of Behavior**
   * By encapsulating algorithms in separate classes, the pattern improves modularity and separates concerns, making code easier to maintain and test (Larman, 2004).
3. **Flexibility in Behavior Changes**
   * The ability to dynamically switch between different strategies at runtime provides significant flexibility in adapting to changing requirements (Fowler, 2002).
4. **Simplifies Complex Conditional Logic**
   * Replacing conditional statements with polymorphic strategy implementations reduces code complexity and improves readability (Martin, 2003).
5. **Encourages Reusability**
   * Strategies can be reused across different contexts, reducing duplication and encouraging consistency in algorithm implementation (Evans, 2004).
6. **Testability**
   * Each strategy can be unit tested in isolation, ensuring reliability and making debugging easier (Beck, 2003).

**Cons**

1. **Increased Number of Classes**
   * The Strategy pattern often leads to the creation of many small classes, which can increase the complexity of the system and make navigation harder (Gamma et al., 1994).
2. **Overhead for Simple Algorithms**
   * For straightforward algorithms or when the set of strategies is unlikely to change, using this pattern may introduce unnecessary overhead (Fowler, 2002).
3. **Complex Client Code**
   * Clients must be aware of different strategies and ensure the appropriate one is chosen and used, which may increase their complexity (Martin, 2003).
4. **Limited Code Sharing**
   * Since each strategy encapsulates its own behavior, sharing common functionality between strategies may require extra effort or duplication (Evans, 2004).
5. **Difficulty in Managing Dependencies**
   * Strategies may have dependencies (e.g., shared data or configuration), and managing these dependencies across different strategy implementations can be challenging (Larman, 2004).
6. **Potential Runtime Errors**
   * Dynamically selecting strategies at runtime increases the risk of errors if an inappropriate or invalid strategy is chosen (Gamma et al., 1994).

**References**

* Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
* Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley.
* Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
* Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
* Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
* Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Pearson Education.

## Pros and Cons of Dependency Injection

## Pros and Cons of Façade Pattern

# System Refactoring

## Tools used

* ReSharper refactoring tool
* SonarQube
* NDepend Scanner, for measuring dependencies in the system, to reduce coupling

## Implementation of Layered Architecture

Due to the problems outlined previously, the chosen architecture pattern was the “Layered Architecture”. To reiterate, a major problem of the system is that all components were tightly linked. The UI often handled business logic, and the business logic and data access were very tightly coupled. For example, my Rates.cs file:

A screen shot of a computer

Description automatically generated

This architecture is a solid choice for this system as it would effectively separate the different elements of the system (data, business, UI), and ultimately make the system more maintainable and scalable.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

## Implementation of Dependency Injection and Centralising Form Navigation

Two

<https://learn.microsoft.com/en-sg/answers/questions/1654403/how-to-implement-layered-architecture>

<https://martinfowler.com/articles/refactoring-dependencies.html>

<https://www.geeksforgeeks.org/dependency-injectiondi-design-pattern/>

* Base Navigation form used to reduce a significant amount of duplicate code, which was showed by a SonarQube scan. This new centralised Navigation function was perfectly suited for injecting dependencies to the different functions during runtime.
* To further decouple the layers, the dependencies are injected at runtime instead of instantiating a new layer service. The implementation is similar to that of this martin fowler article, however the “injector” in my system is instead the form navigation service
* Interfaces were created for my business services and data access classes to facilitate dependency injection
* A diagram of a design process

  Description automatically generated

## Implementation of Builder Pattern

The builder pattern was chosen for the creation of the “Rental” object as this is the most complex object in the system and it highly likely for there to be additional (variables/components?) added to the system. For example, these additional components could be options for insurance, discounts, etc.

It also makes the creation of a rental object with a list of rental items cleaner.

## Implementation of Strategy Pattern

* The main problem here is that new reports would require a lot of new code. This is why merging and using strategy is a good fit.

A key functionality of this tool hire management system is report generation. Currently, the system uses two separate windows forms for report generation: one for annual revenue analysis and another for annual tool type analysis. While this setup works for these existing reports, it becomes increasingly bulky as the business grows and the need for additional reports arises. Each new report would require creating another form and writing similar blocks of code, leading to significant duplicate code and make the system harder to maintain.

To streamline this process, I made the decision to merge the two separate report forms into a single, "Analysis" form. This new form includes a dropdown menu, allowing users to select the type of report they want to generate. Implementing the **Strategy Pattern** is a perfect fit for this enhancement. By treating each report type as a strategy, the system can dynamically choose and execute the correct report generation logic based on the user's selection. This approach not only simplifies the user interface but also eliminates the need for duplicated code, as the core functionality for handling different reports is neatly encapsulated within separate strategy classes.

Adopting the Strategy Pattern has several advantages. It significantly improves maintainability by isolating the report generation logic, which makes it easier to update or modify individual reports without impacting others. Adding a new report becomes straightforward, a new strategy class without the having to alter or create forms or duplicate code. This aligns with the Open/Closed Principle, ensuring that the system remains flexible and scalable as new reporting needs emerge. In addition, this pattern improves code readability and organisation, as each report strategy is clearly defined and managed independently. Overall, merging the report forms using the Strategy Pattern results in a more efficient, flexible, and sustainable reporting system, well-equipped to support the evolving needs of our business.

## Implementation Façade Pattern

At this point in the system many of the UI layers interacted with multiple business layer services, in particular, the rental functionality. Due to time constraints this was only done for the rental functionality

# Further Improvements

# Testing and Validation

SonarQube report

* Make utility functions that are stateless static.
* Make certain field read-only

# Conclusion

The implementation of these design and architecture patterns was a valuable learning experience. Although implementing these patterns is such a small system seemingly overcomplicated things, it became clear how beneficial that patterns can be in a larger system in a real-world scenario.