|  |
| --- |
|  |
| Software Architecture Overview |
| Luke Foley T00224345 |
|  |

Table of Contents

[Introduction 2](#_Toc184026254)

[Existing System Overview 3](#_Toc184026255)

[Main problems 4](#_Toc184026256)

[Design and Architectural Pattern Exploration 4](#_Toc184026257)

[Pros and Cons of Layered Architecture 4](#_Toc184026258)

[Pros and Cons of Builder Pattern 6](#_Toc184026259)

[Pros and Cons of Strategy Pattern 8](#_Toc184026260)

[Pros and Cons of Dependency Injection 10](#_Toc184026261)

[Pros and Cons of Façade Pattern 10](#_Toc184026262)

[System Refactoring 10](#_Toc184026263)

[Tools used 10](#_Toc184026264)

[Implementation of Layered Architecture 10](#_Toc184026265)

[Implementation of Dependency Injection and Centralising Form Navigation 13](#_Toc184026266)

[Implementation of Stored Procedures 14](#_Toc184026267)

[Implementation of Builder Pattern 14](#_Toc184026268)

[Implementation of Strategy and Factory Patterns 14](#_Toc184026269)

[Implementation Façade Pattern 15](#_Toc184026270)

[Implementation of Validation Framework 15](#_Toc184026271)

[Testing and Validation 16](#_Toc184026272)

[Conclusion 16](#_Toc184026273)

# 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 in C# 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.
* 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 rental analysis, providing 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 handle 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

A diagram of a design process

Description automatically generated

## Pros and Cons of Façade Pattern

# System Refactoring

## Tools used

* ReSharper refactoring tool - ReSharper is a refactoring tool for MS Visual Studio that improves upon Visual Studio’s built in refactoring features. It offers automated refactoring options like renaming, extracting methods, and changing signatures, and ensures that refactorings are consistent. It also provides real-time code analysis which identify problems such as bugs and unused variables. ReSharper also has code generation features which save time.
* SonarQube for code quality
* NDepend Scanner, for measuring dependencies in the system, and code quality.

## Implementation of Layered Architectures

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 instance, my Rates.cs file:

A screen shot of a computer

Description automatically generated

In this instance, we have database logic in the Rates class, violating SOLID principles.

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.

In order to implement this architecture, a new project was created for each layer in the Visual Studio solution. These projects are ToolSYS.Presentation, ToolSYS.Business, ToolSYS.Data, ToolSYS.Entities.

All elements of the system were then moved and refactored to the appropriate layer. The figure below shows the various classes that were moved and refactored:

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

All windows forms were placed in the presentation layer. Business logic functions were placed in the business layer, each as its own service. Data access logic was placed in the data layer, with each object as its own repository. Finally, each core object was defined in entities.

In order for the application’s layers to communicate, each layer requires a reference to the layer above it. For example the Presentation layer communicates with the business layer, and should know nothing about the Data layer. The business layer only communicates with the Data layer. In addition, the entities layer is referenced with the sole purpose of defining the core objects of the system – Tools, Rentals, Customers etc.

The following project references were added to each project layer’s dependencies:

**ToolSYS.Presentation:**

A screenshot of a computer program

Description automatically generated

**ToolSYS.Business:**

A screenshot of a computer program

Description automatically generated

**ToolSYS.Data:**

A screenshot of a computer program

Description automatically generated

This formed the following layered architecture:A screenshot of a computer

Description automatically generated

To illustrate the flow of communication between layers, the following is the implementation of the tool search functionality across the 3 layers:

Firstly, in the presentation layer, a reference to the business layer is needed, in this case the services folder, to access the tool service. This is referenced via a “using” statement as seen on line 2 of the figure below. This is necessary for this class to be able to access classes in the business layer. An instance of the tool service class is instantiated on line 12, which allows access to all the methods within the ToolService class.

A screen shot of a computer program

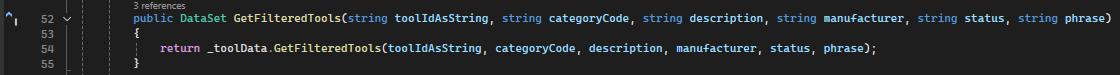
Description automatically generated

With the connection between presentation and business correctly set up, the GetFilteredTools() method is invoked to retrieve filtered tools, as seen on line 45 in the figure below.

A computer screen with text

Description automatically generated

The ToolService class in the business layer receives this request from the presentation laayer. As this function is a simple search operation, with no additional business logic, the request is then sent to the data layer.



The business layer implements a connection to the data layer in the same fashion as the presentation layer is connected to the business layer – a reference to the data layer with “using” and a ToolData() instance.

The data layer then receives the request and executes the SQL query to retrieve the filtered tools from the database. The results are returned to the business layer, which the business layer passes to the presentation layer for display:

A black background with many lines

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

## Implementation of Dependency Injection and Centralising Form Navigation

Two related refactorings to the system were the implementation of dependency injection between architecture layers, and the implementation of a centralised form navigation.

The implementation of dependency injection was inspired by a Martin Fowler article, in which he splits a simple application into a Presentation-Domain-Data layering and implements dependency injection patterns. <https://martinfowler.com/articles/refactoring-dependencies.html>

The current layered architecture relies on services or data repositories being instantiated directly in the class that is using them. For example, the tool search functionality that was previously outlined, directly instantiated the ToolService in the Presentation Layer. While this solution is adequate, the layers can be decoupled further by implementing dependency injection.

To start, interfaces were created for each service and repository. To illustrate the changes, I will continue to show examples for Tools.  
A black background with text

Description automatically generated

A computer screen with text

Description automatically generated

Each of the existing concrete classes will then implement these interfaces:





Each class was then altered to receive their dependencies through their constructors rather than being instantiated directly in the class.

A screen shot of a computer

Description automatically generated

These dependencies can then be defined at our application initiation – program.cs, and use the navigation class to pass dependencies to windows forms:

A computer screen shot of a program code

Description automatically generated

A black screen with text

Description automatically generated

<https://learn.microsoft.com/en-sg/answers/questions/1654403/how-to-implement-layered-architecture66>

<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

## Implementation of Stored Procedures

Stored procedures

## Implementation of Strategy and Simple Factory

### Description

A key functionality of the 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 analysis. While this setup works for these two 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 solve this, 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.

According to refactoring guru, the Strategy pattern suggests that you take a class that does something specific in different ways and extract all of these algorithms into separate classes called *strategies* (refactoring guru, 2024b)*.* Implementing the Strategy Pattern here is a good fit for this enhancement, as each report if going to be generated in a different way. 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 eliminates the need for duplicated code, as the core functionality for handling different reports is encapsulated within separate strategy.

Adopting the Strategy Pattern has several advantages. It 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 by adding an additional strategy. This aligns with the Open/Closed Principle, ensuring that the system remains flexible as the need for more reports arise. In addition, this pattern improves code readability and organisation, as each report strategy is clearly defined and managed independently.

The combination of strategy with simple factory was inspired by this YouTube video by Jono Williams: <https://www.youtube.com/watch?v=aBOrVRKK3fA>

The strategy and simple factory patterns work well together as the factory encapsulates the instantiation of the correct strategy without exposing the logic to the client.

### Code

Firstly, an additional dropdown was added to the Analysis Windows Form to allow users to select the type of report they wish to generate:  
A screenshot of a graph

Description automatically generated

A report strategy interface is defined, with a GenerateReport() method, which takes in a year and report type, and should return a ReportData object:  
A black background with white text

Description automatically generated

Each report type was then defined as a separate concrete strategy, implementing the IReportStrategy interface:

A screen shot of a computer program

Description automatically generated

A simple factory was then defined to instantiate the correct strategy based on the selected report type by the user:

A computer screen shot of a program code

Description automatically generated

Then, from the business layer analysis service, which the presentation layer will interact with, calls the simple factory and returns the correct strategy:

A screen shot of a computer code

Description automatically generated

Finally, from the presentation layer, GetReportData() can be invoked, which will go to the simple factory, return the correct report strategy, and generate the correct report data.

A screen shot of a computer

Description automatically generated

## Implementation of Unit of Work

### Description

The unit of work pattern is a behavioural pattern that groups one or more operations into single transactions or “Units” and executes them by applying a principle of do everything or do nothing. This means that if any of the transaction operations fail, it will roll back the transaction. If all of the operations are successful, then it will commit the transaction (DotNetTutorials, 2019).

In the context of the Tool Hire System, the unit of work pattern can be used for rental creation in the database. To create a rental, the system performs multiple database operations – creating a rental database entry, and several rental item entries. While effective, there is the risk that one of the operations could fail, for example due to network loss or power outage. By implementing the unit of work pattern, the operations are interdependent – if adding a rental item fails, you wouldn’t want the rental record to be created in the database without its associated items, and vice versa. They should both be executed within the same transaction. The unit of work pattern ensures this behaviour.

### Code

Firstly, a Unit of Work interface must be defined. The interface contains definitions BeginTransaction(), Commit(), and Roolback() to implement the Unit of Work pattern, as well as definitions for the repositories involved.

A screen shot of a computer program

Description automatically generated

A concrete implementation of the interface is then created. This Unit of Work class defines manages the database connection and transaction, and initialises the repositories with the same connection, so that they share the same transaction context:

A screen shot of a computer program

Description automatically generated

The repositories then needed to be altered to receive an oracle connection and transaction from the unit of work:

A screen shot of a computer program

Description automatically generated

The transaction then needed to be set in each of the queries:

A screen shot of a computer program

Description automatically generated

With the unit of work correctly set up, the unit of work could now be used by the business layer:

A computer screen shot of a program code

Description automatically generated

The process of adding rentals and rental items to the database is now completed in a single transaction, ensuring consistency.

## Implementation of Builder Pattern

### Description

The builder pattern is a creational design pattern that lets you construct complex objects step by step, and also allows you to produce different types or representations of an object using the same construction code (refactoring guru, 2024a).

In the context of the Tool Hire System, this pattern was chosen for the creation of Rental objects. The rental object is one of the more complex objects in the system, due to the rental object containing a list of rental items. While the builder pattern does not provide significant immediate benefits, it is likely that additional complexity added to rentals, for example insurance or discount options. The builder pattern is used here in anticipation of additions such as this, making rentals more maintainable and future proof.

### Code

A screenshot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

## Implementation of Façade Pattern

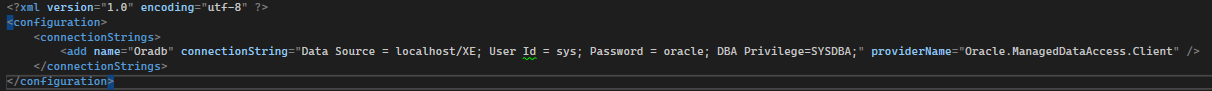
At this point in the system many of the UI layers interacted with multiple business layer services, in particular, the rental functionality. Due to time constraints this was only done for the rental functionality. The Façade was chosen for the rental function specifically because handles Rentals, Customers, Rates, and Tools. The RentalFacade hides in a single interface, which simplifies the function.

## Implementation of Fluent Validation Framework

# Testing and Validation

SonarQube report

* Make utility functions that are stateless static.
* Make certain field read-only
* Duplicate code, (base navigation)
* Long methods (DetermineSQLQuery)
* Security hotspot of the oracle connection string and password  
  A screenshot of a computer

  Description automatically generated  
  A black screen with blue and white text

  Description automatically generated

# 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.

# References

DotNetTutorials, 2019. *Unit Of Work in Repository Pattern*. [online] Dot Net Tutorials. Available at: <https://dotnettutorials.net/lesson/unit-of-work-csharp-mvc/> [Accessed 5 December 2024].

refactoring guru, 2024a. *Builder*. [online] Available at: <https://refactoring.guru/design-patterns/builder> [Accessed 4 December 2024].

refactoring guru, 2024b. *Strategy*. [online] Available at: <https://refactoring.guru/design-patterns/strategy> [Accessed 4 December 2024].