Software Design Patterns

Luke Foley  
T00224345

Table of Contents

[Introduction 3](#_Toc183021552)

[Existing System Overview 3](#_Toc183021553)

[Main problems 4](#_Toc183021554)

[Design and Architectural Pattern Exploration 4](#_Toc183021555)

[Design and Architectural Pattern Implementation 4](#_Toc183021556)

[Testing and Validation 4](#_Toc183021557)

# 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, enabling 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 valuable 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, providing 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

# Design and Architectural Pattern Implementation

## Implementation of Layered Architecture

## Implementation of Dependency Injection

## Implementation of Strategy Pattern

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.

# Further Improvements

Add interfaces to each of my service and data objects

# Testing and Validation

Sonarqube report

* Make utility functions that are stateless static.
* Make certain field readonly