Text

Description automatically generated with low confidenceSoftware Architecture Overview

Course: Software Design Patterns

Name: Kaitlin Fish

Student Number: T00243922

Date: Sept21,2023

Table Of Contents

[Table Of Figures 2](#_Toc152153322)

[1 Background Information 4](#_Toc152153323)

[2. Lab 1 4](#_Toc152153324)

[2.2 Implementation 5](#_Toc152153325)

[2.3 Simple Factory Pattern 8](#_Toc152153326)

[2.3.2 Field Validation Test Cases 9](#_Toc152153327)

[3. Lab 2 9](#_Toc152153328)

[3.1 Implementation 10](#_Toc152153329)

[3.2 Strategy Pattern 14](#_Toc152153330)

[4. Lab 3 15](#_Toc152153331)

[4.1 Implementation 15](#_Toc152153332)

[4.2 Issues within the Lab 19](#_Toc152153333)

[4.3 Layered Architecture 19](#_Toc152153334)

[4.4 Repository Pattern 20](#_Toc152153335)

[5. SonarQube 21](#_Toc152153336)

[Project referencing defect analysis 22](#_Toc152153337)

[6. Reflection 23](#_Toc152153338)

# Table Of Figures

[Figure 1 Design diagram flow 4](#_Toc152510929)

[Figure 2 5](#_Toc152510930)

[Figure 3 6](#_Toc152510931)

[Figure 4 6](#_Toc152510932)

[Figure 5 7](#_Toc152510933)

[Figure 6 7](#_Toc152510934)

[Figure 7 7](#_Toc152510935)

[Figure 8 8](#_Toc152510936)

[Figure 9 Simple factory class diagram 9](#_Toc152510937)

[Figure 10 10](#_Toc152510938)

[Figure 11 logical decoupling diagram 11](#_Toc152510939)

[Figure 12 11](#_Toc152510940)

[Figure 13 12](#_Toc152510941)

[Figure 14 Physical coupling diagram 12](https://ittralee-my.sharepoint.com/personal/kaitlin_fish_students_ittralee_ie/Documents/Kaitlin%20Fish%20-%20Software%20Architecture%20Overview.docx#_Toc152510942)

[Figure 15 13](#_Toc152510943)

[Figure 16 13](#_Toc152510944)

[Figure 17 13](#_Toc152510945)

[Figure 18 14](#_Toc152510946)

[Figure 19 Strategy pattern class diagram 15](#_Toc152510947)

[Figure 20 15](#_Toc152510948)

[Figure 21 16](#_Toc152510949)

[Figure 22 17](#_Toc152510950)

[Figure 23 18](#_Toc152510951)

[Figure 24 18](#_Toc152510952)

[Figure 25 19](#_Toc152510953)

[Figure 26 19](#_Toc152510954)

[Figure 27 20](#_Toc152510955)

[Figure 28 Layered architecture diagram 21](#_Toc152510956)

[Figure 29 Repository pattern Diagram 22](#_Toc152510957)

[Figure 30 23](#_Toc152510958)

[Figure 31 23](#_Toc152510959)

[Figure 32 23](#_Toc152510960)

[Figure 33 23](#_Toc152510961)

[Figure 34 23](#_Toc152510962)

[Figure 35 24](#_Toc152510963)

# 1 Background Information

The route chosen for this overview is to develop a pet management system in C# on visual studio 2022. This project will reference a course on Udemy instructed by Shivsprasad Koirala. Full course can be found here: https://www.udemy.com/course/designpattern/learn/lecture/12758597#content

Shivsprasad’s course demonstrates the use of multiple design patterns by creating a customer billing system. The development of the pet management program will be executed in three labs, each lab will implement at least one design pattern.

This pet management program will be implemented in the user story of a vet clinic when a pet’s bill is being registered into their system after an appointment. Or, if a pet is being added into the system for the first time and has not had a vet appointment yet. Vet techs can select which type of entry they would like to register. The program will validate for blank fields depending on which type of entry the vet tech would like to register. If all fields have been filled in, the vet tech can then register the pet into the system, which sends the pet information to a database.

A computer screen shot of a computer

Description automatically generated

Figure Design diagram flow

#### Functional requirements:

1. Program must validate if fields have been left blank. Program will validate certain fields depending on entry type selected.
2. Program must store registration entries into a database and show the tables in the user interface
3. Implement 1 design pattern of each category.

#### Non-Functional Requirements

1. Program could have a clean or “aesthetically pleasing” user interface.
2. Achieve as much decoupling in the program as possible.
3. Code smells should be used to improve the maintainability of the code.

# 2. Lab 1

The objectives of lab one is to develop the basic Gui of the pet management system, and to implement basic entry validations in the system based off different entry algorithms with the assistance of the simple factory design.

‘Basic’ entry type is for owners that want to enroll their pet into the vet clinic but have not had any visits yet. Therefore, there is no need to fill in the billing info yet. ‘Pet’ entry type that will be selected when a pet has had a service provided by the clinic and is time for the bill to be recorded in the pet’s medical records.

## 2.2 Implementation

First, a class library called ‘MiddleLayer’ is created with a class called ‘Pet.cs’ . The purpose of this class is to define our base classes and fields that the program will be validating for. Base classes ‘PetBase’,’Pet’, and ‘basic’ will be defined. In ‘PetBase’ the entry fields will be defined, and a method called ‘Validate’ will be declared.

A screen shot of a computer program

Description automatically generated

Figure

In the ‘Pet’ class, the validate method will validate if fields ‘PetName’,’ownerPhoneNumber’,’vetBillAmount’,’vetBillDate’, and ‘ownerAddress’ are left blank, and will throw respective exception messages. In the ‘Basic’ class, the validate method will validate if fields ‘PetName’ and ‘ownerPhoneNumber’ have been left blank and will throw respective exception messages.

A screenshot of a computer program

Description automatically generated

Figure

In Figure 2, both classes inherit from ‘PetBase’ This is because both classes need to validate using the same fields. Rather than creating duplicate fields in different classes with slightly altered names, The Pet class and the Basic class will validate using the same fields from the PetBase Class.

A screenshot of a computer

Description automatically generated

Figure

Next, The GUI design side of the project is developed. Labels for each field is created, along with entry text boxes. A validate button is also created. A data grid view is included for the database layer. The database layer will not be implemented until lab 3.

A screenshot of a computer

Description automatically generated

Figure

A new class library is created called ‘FactoryPet’, with the class inside it called ‘FactoryPet’. It is given a project reference to ‘MiddleLayer’ so classes can be used. In ‘FactoryPet’, a collection of base type Dictionary referencing ‘PetBase’ is created. It is used to store instances of objects created in ‘PetBase’ where the data type is string. It is defined in the field of type string called ‘Createpets’. A method of type ‘PetBase’ called ‘Create’ is created, with a parameter of type string called ‘petType’. It checks if the dictionary is empty. If it is, the dictionary becomes populated to whichever class object the pet type is selected as. It returns the dictionary field of the whichever object is associated with ‘petType’.

A computer screen shot of a program

Description automatically generated

Figure

Finally, the user interface layer will be enhanced to use the business logic. Inside the combo box method that allows vet techs to choose the entry type, an instance of ‘petBase’ will be declared and initialized to call the create method from the Factory class.

A screen shot of a computer

Description automatically generated

Figure

A private void method is created called ‘setPet’. This method assigns fields created in the ‘PetBase’ class to their respective Textboxes.

A screen shot of a computer program

Description automatically generated

Figure

In the validate button method, a try catch is implemented. In the try, the ‘setPet’ method is called, and the ‘createPet’ instance calls the validate method. In the catch clause, it also calls the ‘Validate’ method using the ‘createPet’ instance. Depending on the type of entry the vet tech selects, the validate button will create an instance of whichever object corresponds with the entry type selected and check if the fields conditions in that class have been met. If not, the program will give an exception message.

## 2.3 Simple Factory Pattern

The simple factory design pattern falls under the creational design pattern category. Simple factory is essentially, a class that contains a single method which creates and returns different instances of objects depending on given input (refactoring guru, n.d.). Simple factory can be given the analogy of a juice machine. If you put apples into the juicer, its going to give you apple juice. If you put oranges into the juice machine, its going to give you orange juice. The juice you will get from the machine depends on what fruit you put in.

In Shivsprasad’s customer billing program, he justifies implementing the simple factory pattern by explaining how the simple factory pattern takes away the need for creating duplications of the create method for each entry type.

While the simple factory pattern is one of the more limited creational patterns in terms of flexibility for handling complex object creation scenarios, it is a beneficial design pattern to implement in this stage of the pet program. Implementing the simple factory design pattern takes away the need for code duplication by centralizing object creation into one class method, instead of needing to create similar class methods under different scenarios. If the simple factory wasn’t implemented, a create method would need to be duplicated for each entry type the program has. Currently, the pet program only handles two entry types. But if the program were to expand later by wanting to handle other entry types, maintainability could become more difficult by having duplicate class methods.

The only drawback to the simple factory pattern is that its not suitable for more complex hierarchies. Because of its limitations in terms of object creation variation, real world applications would rarely use this pattern in finished results. Real world applications would most likely use the abstract factory pattern. Simple factory might be used for testing purposes in development instead.

A diagram of a pet base

Description automatically generated

Figure Simple factory class diagram

## 2.3.2 Field Validation Test Cases

The following test cases were executed in the program to verify the program points to correct validation code under the appropriate circumstances.

|  |  |  |
| --- | --- | --- |
| **Test Case Scenario’s if type is Basic** | **Expected Outcome** | **Actual Outcome** |
| Fills in name field, does not fill in phone number field | program crashes, outputs phone number exception message | program crashes, outputs phone number exception message |
| Fills in phone number field, does not fill in name field | program crashes, outputs name exception message | program crashes, outputs name exception message |
| Fills in name and phone number field | Nothing happens | Nothing happens |
| Does not fill in name and phone number field | program crashes, outputs name exception message a | program crashes, outputs name exception message |

|  |  |  |
| --- | --- | --- |
| **Test Case Scenario’s if type is Pet** | **Expected Outcome** | **Actual Outcome** |
| Does not fill in any fields | Program crashes, outputs first exception message which is name exception message | Program crashes, outputs first exception message which is name exception message |
| All fields are filled in | Nothing happens | Nothing happens |
| All fields are filled in but name | Program crashes, outputs name exception message | Program crashes, outputs name exception message |
| All fields are filled in but phone number | Program crashes, outputs phone number exception message | Program crashes, outputs phone number exception message |
| All fields are filled in but vet bill amount field | Program crashes, outputs vet bill amount exception message | Program crashes, outputs Owner address exception |
| All fields are filled in but vet bill date field | Program crashes, outputs vet bill date exception message | Program crashes, outputs vet bill date exception message |
| All fields are filled in but owner address field | Program crashes, outputs owner address exception message | Program crashes, outputs owner address exception message |

After careful evaluation, it was discovered that ‘vetBillAmount’ field was going to a different exception message because it needed a default value defined. When a default value of 0 was given, the program pointed to the ‘vetBillAmount’ exception message as it was meant to before.

# 3. Lab 2

The objective of lab two is to do a few housekeeping tasks to better decouple the program and use the strategy pattern to improve the validation feature.

## 3.1 Implementation

First, the nugget packet ‘Unity’ is imported into the project. Unity is an extension packet available in visual studio 2022 that allows developers to use dependency injection to manage the construction of objects in their applications. ’FactoryPet’ is modified to include ‘IUnityContainer’ injection.

Shivsprasad explains that implementing dependency injections into the project allows the object creation to become more automated. The programs functionality doesn’t change by implementing unity. Personally, no difference in performance was noticed by introducing dependency injections.

A screen shot of a computer

Description automatically generated

Figure

Next, the architecture of the class references in the program will be slightly altered to achieve a better decoupled architecture. Currently, the middle layer class serves as the middle point between the user Interface layer (‘FrmPet’), and the business layer (‘FactoryPet’). Shivsprasad explains that the current architecture is more decoupled logically than physically. The middle layer contains our ‘PetBase’ class definitions. There are three classes. ‘PetBase’, ‘Pet’, and ‘Basic’. Basic and class inherit from petBase class. In the UI layer, ‘PetBase’ class is being referenced. There is no reference to the Pet class or the basic class.

A diagram of a factory

Description automatically generated

Figure logical decoupling diagram

Logical decoupling is not necessarily a bad thing in a program, but excessive logical decoupling can add challenged in program behavior prediction, especially in a program that undergoes constant modification. While the pet programs development life cycle will not last as long as Shivaprakads, it is an issue he acknowledges he can run into in the later labs.

To prevent this issue in the early labs before it festers, Shivsprasad would like to achieve some physical decoupling from the middle layer and the UI layer. Physical decoupling could improve the scalability of the program as features can be added or removed without affecting other components of the program.

To achieve this, the middle layer reference is completely removed from the UI layer. A new class library is added to the project titled ‘InterfacePet’, with its class name being ‘IPet’ The purpose of this class is to separate some responsibility from the middle layer by holding our field definitions and define the validate method. The UI layer will add ‘InterfacePet’ to its reference list. ‘MiddleLayer’ will also add ‘InterfacePet’ as a reference.

A screen shot of a computer

Description automatically generated

Figure

By doing this, the ‘InterfacePet’ layer now becomes the connection point between the UI layer and the Base business layer. The ‘PetBase’ class inside the middle layer will now implement ‘IPet’.

A screen shot of a computer

Description automatically generated

Figure

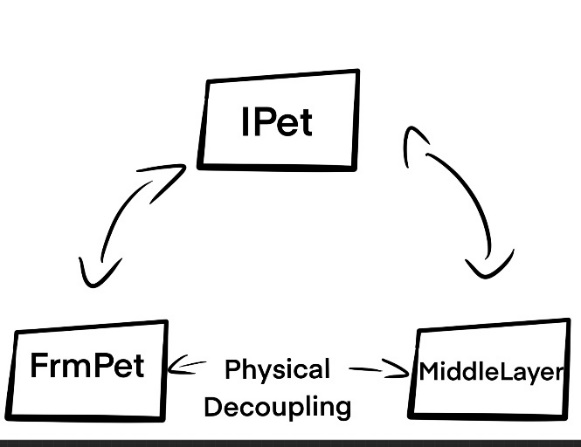
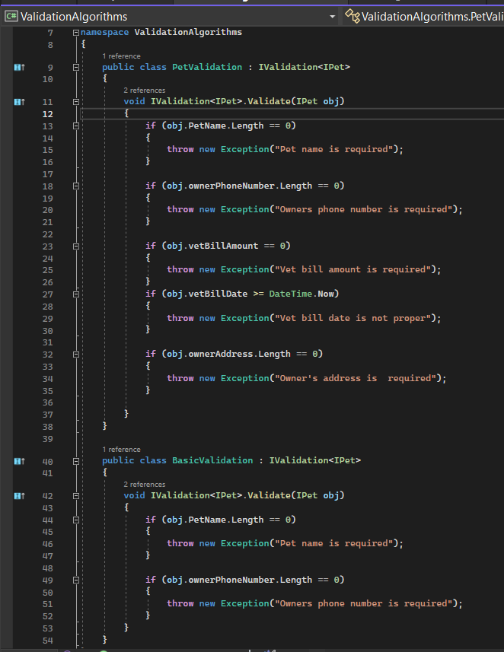
The factory class will also now add a reference to ‘InterfacePet’. The create method is modified to use the Validate method defined in the ‘Ipet’ class.

Figure 14 Physical coupling diagram

The architecture has now been modified to no longer use the middle layer for the UI but use Interface as the communication point between the two instead. The middle layer and UI layer are now separate from each other. This modification doesn’t alter how the program runs from the user end but separating the two has now better decoupled the program.

Next, the strategy pattern will be used to improve the functionality of the validation feature in the program.

To implement the strategy pattern into the program, a new class library is created titled ‘ValidationAlgorithms’ with the class being titled the same. The purpose of this class is to handle all our validation algorithms and conditions. Currently, the validation algorithms are in the ‘MiddleLayer’, where the base classes and class instances are defined. The validation logic is taken out of ‘MiddleLayer’ and placed into the ‘ValidationAlgorithms’ class. This class defined two validation classes. ‘PetValidation’, and ‘BasicValidation’. Inside these classes, different validation algorithms are defined. These would be considered the concreate strategies that the strategy interface created in the next step will use.



Figure

The class adds a reference to ‘interfacePet’, where the instance definitions are being held.

Next, in ‘IPet’, an interface method titled ‘IValidation’ is created. It takes in a parameter of ‘anyType’. Inside the method, it calls the validate method, bringing in an object of ‘anyType’. In figure 15 the ‘PetValidation’ and ‘BasicValidation’ inherits from this interface. This serves as the interface which will declare the common method that all strategies will use. This interface represents the family of algorithms.

A screen shot of a computer

Description automatically generated

Figure

Next, ‘MiddleLayer’ will be adjusted to use ‘ValidationAlgorithms’. A private variable of type ‘IValidation’ is declared. In the ‘petBase’ class, ‘validation’ is assigned to ‘obj’. This variable is defined so it can point to the validation fields in the validation algorithms.

A screen shot of a computer code

Description automatically generated

Figure

Below, the base classes for the entry types take in parameters of type ‘IValidation’ and ‘IPet’ into the obj variable.

A computer screen shot of a program code

Description automatically generated

Figure

Finally, in ‘FactoryPet’, the ‘Create’ method is altered to implement the strategy interface. In Figure 10, the type of the method is changed to ‘IPet’. Also, at the end of the ‘registerType’ method, the classes ‘PetValidation’ and ‘BasicValidtion’ are used. It now points to the class library that holds the validation algorithm logic defined in the earlier stages of the lab.

## 3.2 Strategy Pattern

The strategy pattern is a behavioral design pattern pattern that defines a set of algorithms and encapsulates each algorithm (Eric Freeman, 2004). It allows algorithms to be selected at runtime. The strategy pattern is commonly used when a program has multiple algorithms that can run depending on user entry.

A small case study that was conducted by Ikhiloya Imokhai on how the strategy pattern can be used in an email application to handle multiple email clients and storage types depending on which environment or profile the application is running (Imokhai, 2020). The application is either run in dev or prod. The goal being to switch between Gmail and SendGrid email clients, and between local and AWS storage. The case study emphasizes the ease of switching between dev and prod files without having the handle the specific implementations, thanks to the runtime flexibility provided by the strategy pattern.

In Shivsprasad’s customer billing program, he justifies implementing the strategy pattern to solve the problem of coupling between base classes and algorithm. Before implementing the strategy pattern, Both the base classes and validation logic were coupled in one class. This takes away the ability for the algorithms to dynamically switch at runtime. Having the base and algorithm nested together can heavily limit its extensibility. It would be more difficult to add additional algorithms or modify existing algorithms without disrupting other parts of the program.

In the pet program, the strategy pattern is implemented by defining a set of concrete strategies (figure 15), create an interface that defines the family of algorithms (figure 16), and modify the base class and create method to use the interface, allowing the program to dynamically select objects at runtime depending on the entry type the vet tech selects (figure 10, 17, 18).

Personally, adding the strategy pattern into the pet program for the field validation feature will greatly benefit the program as it evolves or changes later in its lifecycle. In any program, there is always the potential for things to change. The validation conditions of entry types could be changed anytime. For example, instead of just validating pet name and phone number for basic entry type, the vet clinic might also want the basic entry type to also validate the owner’s address. Implementing the strategy pattern will enhance maintainability and extensibility.

One drawback of the strategy pattern is the potential for runtime overhead. Runtime overhead is when a program is using excess amounts of processing power or memory to perform a specific task. Depending on the complexity of the strategies and the frequency of changes happening each time the program is run, this can have a negative impact on the program’s performance.

A diagram of a strategy

Description automatically generated

Figure Strategy pattern class diagram

# 4. Lab 3

The goal of lab three is to introduce a database layer into the project and use the repository design pattern to assist in decoupling the business logic from the data access logic of the database.

## 4.1 Implementation

First, the database needs to be created in a database software. The chosen database software to use was SQL server management studio 2019, as that version is still having compatibility capabilities to .NET 2.0. A database table titled “petTBL” is created. All validation fields that are in the program are included in the table. The database is connected to the program by using the ‘add new data source” feature in visual studio 2019. A register button is also added to the UI titled ‘Register’.

A screenshot of a computer

Description automatically generated

Figure

Next, a new class library is created titled ‘Interface Dal’. This class will hold crud operations for the database feature. The purpose of having a DAL layer is to provide a way to interact with a database. This class also introduces the repository design pattern.

A screen shot of a computer program

Description automatically generated

Figure

Next, a new class library titled ‘CommonDAL’ is created . It adds a reference to the InterfaceDal class library. The purpose of this class is to serve as an abstract class for the data access later. This class also defines the specific functionality of the CRUD operation methods that were created inside the InterfaceDAL layer.The constructor of the class takes in a connection\_string parameter and is provided a value. Connection string properties are used to connect to database storage systems. A protected List property called “Anytypes” is defined. This field will be used to store instances of “Anytype” that the DAL layer is designed to work with

A computer screen shot of a program

Description automatically generated

Figure

Another class library is added to the project called ‘TemplateADO’. The purpose of this class is to serve as a template for the actual implementation for the data access layer using ADO.NET in a fixed sequence of steps. This class inherits from the ‘AbstractDAL’ class in the ‘commonDAL’ layer. The constructor uses the properties ‘SqlConnection’ and ‘SQLCommand’ to manage the database connection and any needed commands. The open and close methods are responsible for opening and closing the database connection. The two execute methods use a fixed sequence of database operations.

They may look the same at first but have different purposes. The ‘Execute(AnyType obj)’ method is used for inserting operations, while the ‘Execute()’ method is used for select operations. They both open the connect, perform their specific operations that were defined in the ‘CommonDAL’ layer, and close the connection.

A screenshot of a computer program

Description automatically generated

Figure

A screenshot of a computer program

Description automatically generated

Figure

A class titled ‘PetDAl’ is also defined in this class. It inherits from ‘AbstractDAL’. It uses the object of ‘IPet’ from the ‘InterfacePet’ layer. The method ‘ExecuteCommand()’ specifies SQL commands to insert and select data from the pet table created in SQL server management studios. It uses Factory to create instances of the ‘IPet’ interface.

A computer screen shot of a program

Description automatically generated

Figure

Next, a new class library is created called ‘FactoryDal’. It references ‘AdoNetDAL’, ‘InterfaceDAL’,’InterfacePet’, and ‘CommonDal’. This class uses the simple factory pattern to create objects like ‘Factorypet’, except this object creation is for the database side of the program, not the validation side.

In Figure 25, it is trying to use Factory to create an instance of the pet to insert into the database. However, if an attempt to add a project reference from ‘FactoryPet’ is made, it will not work. This is because the ‘FactoryPet’ class also uses ‘PetDAL’. What ends up happening is circular dependency.

Circular dependency is when two or more classes depend on each other, either directly or indirectly. This creates a loop of dependency that can create a lot of issues later and are tricky to solve. To navigate this, a new Factory class needs to be created. The architecture of the program will now have a factory class for the’ MiddleLayer’, and a factory class for the data access layer. The ‘FactoryDal’ class will also use dependency injection to automate the simple factory design pattern. In the resolve method, it takes in a parameter of the database connection string.

A screen shot of a computer program

Description automatically generated

Figure

Finally, inside the ‘FrmPet’ class, a new method called ‘LoadGrid’ will be created, this method will fetch data from the data access layer and populate the data grid view with that data. The ‘LoadGrid’ method is called in the onload event when the program first runs.

A computer screen with text on it

Description automatically generated

Figure

## 4.2 Issues within the Lab

In figure 25, there’s an error on the line that has they keyword Factory. This error is due to the class needing a reference to the ‘factoryDAL’ class. However, the project will not allow a project reference to be added to the class library. This could be because of some referencing issues, or circular dependency has already occurred that hasn’t been picked up on yet. A deeper evaluation on the referencing issues with the classes will need to take place in page 19-22.

## 4.3 Layered Architecture

The layered architecture is a software architecture pattern that splits a program or application into different component layers. Each layer has its own functionality and responsibilities.

The top layer is the presentation layer, which would be the user interface. This is the layer the client sees and interacts with. In the pet management system on the database feature, the presentation layer would be the FrmPet class library, as it contains the Gui and the register button the end user sees and interacts with.

The layer below would be the business logic layer. This layer contains the back-end logic of the program. The business layer can process user input, perform calculations, enforce certain rules or validations, and helps guide the flow of data between the data layer and the presentation layer. In the context of the database feature of the pet program, the business layer would be ‘FactoryDAL’. ‘FactoryDAL’ is responsible for populating the grid view with data once the register button has been selected.

The layer below the business layer would be the persistence layer. The persistence layer is responsible for managing the storage and retrieval of data. It serves as the middle ground between the business layer and the database layer by abstracting the interactions wit the database. In the context of the database feature in the pet management program, the persistence layer would include ‘TemplateADO’, ’AbstractDAL’, and ‘InteraceDal’. These three classes are responsible for abstracting the data from the database and managing how that data gets processed.

The bottom layer of the layered architecture is the database or data storage system itself. In the context of the database feature of the pet management program, this layer would include the ‘petTble’ created in SQL server management studios 2019.

A benefit of a program being designed with a layered architecture is reusability. The separation of components allows them to be reused for different applications. In my work placement, I had projects that reused the same business logic layer with a few small tweaks. In some applications, the presentation layer was identical throughout different projects. They just had a few labels changed.

One drawback in a program with a layered architecture is maintainability. To make a change in one layer of the program, a deeper analysis of how the change would impact other layers of the program would need to be considered.

A real-world example of the layered architecture being used is Gmail. A journal article written by Madhusha Prasad discusses the different layers that make up Gmail’s architecture (MadhushaPrasad, 2023). Their presentation layer communicates with users, their business layer handles the processing, and there’s a persistence layer that interacts with a database to store email messages.

A diagram of a company

Description automatically generated

Figure Layered architecture diagram

## 4.4 Repository Pattern

The repository pattern is a structural design pattern that is commonly used in programs that have a data access layer. The repository pattern is used to separate the business logic from the database. It serves as a middle point between the two.

In the context of the pet management program, the repository pattern is implemented by defining the repository interface ‘Idal’, creating concrete repository implementation ‘CommonDAL’ and ‘TemplateADO’, then introducing the business layer that speaks with the repository interface ‘FactoryDAL’. ‘Idal’, ‘TemplateAdo’ and ‘CommonDal’ are able to abstract the data access logic and operations from the rest of the program

In shivsrapads course, he justifies using this design pattern to prevent data access logic issues and adhere to the ‘seperation of concern’ rules. When a program has directly embedded data access logic in different parts of the program, it raises the chances of running into issues later if the program needs to switch out databases. The program would have to search through and modify all areas in the program that’s handling data access logic.

By implementing the repository pattern, the program will have far less areas in the program to modify. Personally, adding the repository pattern in the database feature of the program has improved the programs maintainability and gives the program more flexibility if the program wants to work with multiple databases. The repository has also helped create a layered architecture in the program, which better separates components for better personal understanding of the program.

One drawback the repository pattern brings to the program is increasing difficulty in handling complex queries. Repositories might struggle with handling complex queries that need multiple joins. If the program requires more complex queries later in its life cycle, the repository might no longer be suitable for the program.

A real-world example of the repository pattern being used could be a web application that manages a library of books. The application would need to be able to add new books, retrieve book details, update book information, or delete books from the library. The repository pattern could be implemented in this application to abstract away the details of the database interactions. The repository pattern is a popular design pattern. It is often found in applications that have a database component.

A diagram of a data flow

Description automatically generated

Figure Repository pattern Diagram

# 5. SonarQube

SonarQube is a code analytics tool used to detect code smells and bugs. The purpose of SonarQube is to assist in providing cleaner code. SonarQube will be used in this project to detect any bugs or maintainability issues to help improve the programs’ structure. Scanning the project through SonarQube has picked up 13 code smells in the maintainability section. It doesn’t pick up the referencing bug experienced in the third lab, however. There is a possibility that bug is also preventing other code smells from being picked up by SonarQube. Most of the code smells picked up by SonarQube are maintainability related.

A screenshot of a test

Description automatically generated

Figure

A screenshot of a chat

Description automatically generated

Figure

A screenshot of a computer

Description automatically generated

Figure

A screenshot of a computer

Description automatically generated

Figure

A screenshot of a computer

Description automatically generated

Figure

After making a few refactoring adjustments based off the given code smells, the project is scanned once again in Sonarqube.

A screenshot of a computer

Description automatically generated

Figure

The code smells picked up in the maintainability section decreased down from 13 to 10. The persistent maintainability code smells that remained are ‘System Exception’ should not be thrown by user code. This code smell was not altered due to this method of error message handling being the only way to verify the validation feature was working. It’s acknowledged that throwing exception message is not a user-friendly way of telling the user if they didn’t fill in a field properly. You would normally use a ‘MessageBoxShow’ method. However, null exceptions would come up when that method was used. The object was clearly not null if it was able to point to the proper conditional statement. If there was more time to develop the project, there would have been more time spent on improving how the error message appears for users.

## 5.1 Project referencing defect analysis

Both class libraries ‘FactoryDAL’ and ‘TemplateADO’ need to be referencing eachother. ‘FactoryDAL’ is referencing ‘TemplateADO’ as it should, but ‘TemplateADO’ isn’t able to add a reference to ‘FactoryDAL’. To see what would happen, project reference from ‘FactoryDAL’ was taken away and then added the ‘FactoryDAL’ reference to ‘TemplateADO’. The reference came through successfully, but now ‘TemplateADO’ reference cannot be added to ‘FactoryDAL’. It seems neither class library can reference eachother at the same time.

Upon closer inspection, ‘TemplateADO’ needed to reference ‘FactoryPet’, not ‘FactoryDAL’. While ‘FactoryDAL’ needed to reference ‘TemplateADO’. ‘TemplateADO’ reference error has been fixed. However, ‘FactoryDAL’ is now having issues referencing issues to ‘TemplateADO’. There are a few reasons that this can be occurring, a in depth analysis of each defect scenario will now take place.

|  |  |  |  |
| --- | --- | --- | --- |
| **Possible Cause** | **Possible Solution** | **Execution** | **Solved?** |
| Project .NET compatibility issues | Ensure both class libraries were created in .NET 2.0 | Investigate config file of class libraries. Both are created under .NET 2.0 | no |
| Build error | Ensure the project build ran without errors. Sometimes if a class library has an unsuccessful build, other class libraries will have issues referencing it | Re-build TemplateADO class since initial referencing error has been resolved | no |
| Clean solution | Clean solution command might need to be run on project | Clean solution ran succesfully | no |
| Project Type | Ensure both class libraries are of type class library | Checked in class library properties if both class libraries are of type class library. Both are | no |
| Circular Dependency | Review all class references to ensure no circular dependency has sneakily occurred | In depth review comparing references to shiv’s project to ensure references were not missed | no |

The factoryDal class was created to prevent circular dependency. However, there is now a ‘diamond’ problem in the program. Diamond problem can occur when two other classes cannot be referenced at the same time, commonly due to having an ancestor class. ‘TemplateAdo’ can now reference ‘factoryPet’ when the reference to ‘factoryDal’ was taken away. Now, ‘factoryDal’ cannot reference ‘TemplateAdo’

Due to time limitations, the defect was unsolved.

# 6. Reflection

One thing I liked about this project was it helped wrap my head around design patterns as a general concept. I felt very lost in the first few weeks of this course. I remember sitting down on the weekends trying to write my own notes to better understand software design pattern concepts. It still wouldn’t click for me. Approaching this project in the form of a follow along course helped me better understand software design patterns by having someone explain the patterns as I was developing it. Design patterns are one of those things where I needed to go in and do it myself to see how it was interacting with the program or changing the flow of the program.

I also learned other things throughout this project not necessarily related to design patterns. I hadn’t touched C# or the program visual studios 2022 in a few years so my developing skills in that section were rusty. This project allowed me to re-learn some concepts from my C# course back in 2nd year of college and re-learn how visual studio’s GUI application side works.

Something I really struggled with at the beginning of this project was the way Sivsraprad approached this project. Shivsraprad developed his customer billing system in visual studio 2010. He justified this by explaining working in an earlier version of visual studio 2010 made the lab project folders he was uploading along with this project easier to migrate for students that just wanted to download the lab folders. The trade-off was that it made the course harder to follow along with. The version of the .NET framework files he was using when creating the project were not compatible with the project when I would try to follow along. I spent a few days playing around with project creation, figuring out what versions of .NET framework class libraries would work with my project.

One thing I wish I did differently was comments in my code screenshots. In the implementation section of the first lab, you’ll notice the screenshots don’t have any comments in them, but code screenshots in later labs have a short summary of the purpose of the class. This was an oversight on my part. I didn’t think to add comments in my code until later in the project. I would have gone back and added comments to the screenshots, but the code from the first lab had been altered too much for it to make sense for the implementation of the first lab.

Another thing I wish I had done differently was thought out the details of the subject of my project better. I’m not happy with the naming conventions of the entry types of the pet registration. “pet” and “basic” doesn’t tell the user anything about what those entry types could mean. I wish I had watched through the three lab videos and brainstormed those types of details before starting development, instead of brainstorming as I developed along with the video. Changing the naming conventions of those details would require re-naming in many areas in the project, which is too much of a headache. If I had more time I would’ve gone back and changed those things.

Another thing I wish I had done differently was deciding on my project earlier on in the semester. I was originally going to add to an existing restaurant management system in C++.It wasn’t until two weeks after we were meant to pick a project that I realized the project wouldn’t work due to lack of code complexity. Two weeks may not seem like a lot of time, but it is still time I lost in the project. Those two weeks could have helped make a difference in the outcome of the project.

If I had more time to work on the project, I would have additionally added a 2nd database that holds pet registration for first time clients if the vet tech selected the entry type as ‘Basic’. Currently, entries are stored in one database regardless of entry type. It would have been nice to have separate databases for entry types for better data organization.

One of my biggest takeaways from this assignment was understanding design patterns on a higher level as I continued in the assignment. Why we use design patterns. When to use a design pattern or when to not use a design pattern. While design patterns aren’t an essential thing to have in a program, design patterns tie into software quality management. Software design patterns can be used to assist in defect prevention. There is an interesting case study performed by Apostolos Ampatzoglu, Georgia Frantzeksou, and loaniss stamelos where they compare the quality of a system with and without design patterns during maintenance versus using alternative designs to address the same problem (Apostolos Ampatzoglu, 2011).

For this case study, they chose to use the bridge pattern, abstract factory, and visitor pattern as the example design patterns. Metric scores were calculated under eleven quality attributes. The case study found that in most scenarios it was evaluating under, the design pattern solution approach was the more effective approach, as design patterns are narrower and are meant to solve specific problems or prevent specific problems.

The conclusion of the case study was more unbiased. The paper recognized their findings can’t be definitive for all patterns, quality attributes, or pattern sizes. For each problem a developer is trying to solve, a deeper analysis needs to be made under different categories to determine the best solution.

This paper helped re-enforce the idea that design patterns can be effective in terms of software quality, as the design pattern scored higher than alternative general design solutions under certain software quality metrics.

In my research about the design patterns, I used for this assignment and their drawbacks, performance issues often came up. I looked at a few other design patterns not used for the assignment and their drawbacks. Performance issues also came up. The main reason for performance issues is due to misuse of the design pattern. Design patterns are meant to solve specific problems. They won’t perform as well, or perform at all, if they aren’t being used properly.

I found a research paper written by Riccardo Pinciroli, Aldeida Aleti, and Catia Trubiani. The purpose of their research paper was to assist architects in understanding how different workloads can affect the performance of design patterns in the context of microservice systems (Riccardo Pinciroli, 2023).

They conducted this study by identifying several design patterns to use. Then they use these design patterns under a variety of tasks and measure the performance of each design pattern. They utilized performance models, focusing on Queuing networks to collect their findings.

The findings were a variety of higher and lower performances based on different workloads given to each design pattern.

This research paper helped re-enforce the idea that the quality of a design patterns performance is dependent on the workload scenario given to it.

Although the outcome of my pet program wasn’t fully functional, I felt I the approach taken for this assignment greatly helped make the content taught in class click better.

# References

Apostolos Ampatzoglu, G. F. (2011, June 8). *A methodology to assess the impact of design patterns on software quality.* Retrieved from science direct assets.com: https://pdf.sciencedirectassets.com/271539/1-s2.0-S0950584912X00028/1-s2.0-S0950584911002151/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEK%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLWVhc3QtMSJHMEUCIHC0pl6eZ%2FgyBBHkBO%2BQXNu5JovsfTp0mYvn%2Fr8Z0PpdAiEA57GV

Eric Freeman, E. F. (2004). Head First Desgin patterns. In E. F. Eric Freeman, *Head First Desgin patterns* (p. 681). O'Reilly Media .

Imokhai, I. (2020, May 14). *Application of The Strategy Design Pattern.* Retrieved from Medium.com: https://ikhiloyaimokhai.medium.com/practical-application-of-the-strategy-design-pattern-dbf50b2952ce

MadhushaPrasad. (2023, July 6). *Layered Architecture.* Retrieved from medium.com: https://madushaprasad21.medium.com/layered-architecture-84afbfcef10f

refactoring guru. (n.d.). *Factory Comparison*. Retrieved from refactoring guru: https://refactoring.guru/design-patterns/factory-comparison#:~:text=The%20Simple%20factory%20pattern,of%20the%20creational%20design%20patterns.

Riccardo Pinciroli, A. A. (2023, April 24). *Performance Modeling and Analysis of Design Patterns for Microservice Systems.* Retrieved from IEEExpolore: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10092637