# COMP3304 Advanced Object Oriented Programming

## Portfolio Design Documentation

Student Name: Ryan Coles

Student Number: 18010943

### Introduction

*All documentation has some kind of introductory text. We usually write this last. Don’t spend too long on it, just a brief summary of the contents of the document*.

This project includes a detail design and the analysis of the image/video asset manipulation for the client ‘PetSimsRUs’. The software allows the user to load images into memory and display them on a windows form. You can also cycle through each image to change the currently displayed image.

### Problem Analysis

*This should be your interpretation of the client brief. This section is essentially setting-out the problem that your software design is intended to solve.*

Outline of problems given in client brief

The main problem that the client brief asks to solve is create a simple image/video asset manipulation tool with a GUI to display images and manipulation functionality. The Client also requires me to make use of the *ImageProcessor OSS library,* which is an open source .Net library used to manipulate images on-the-fly*.*

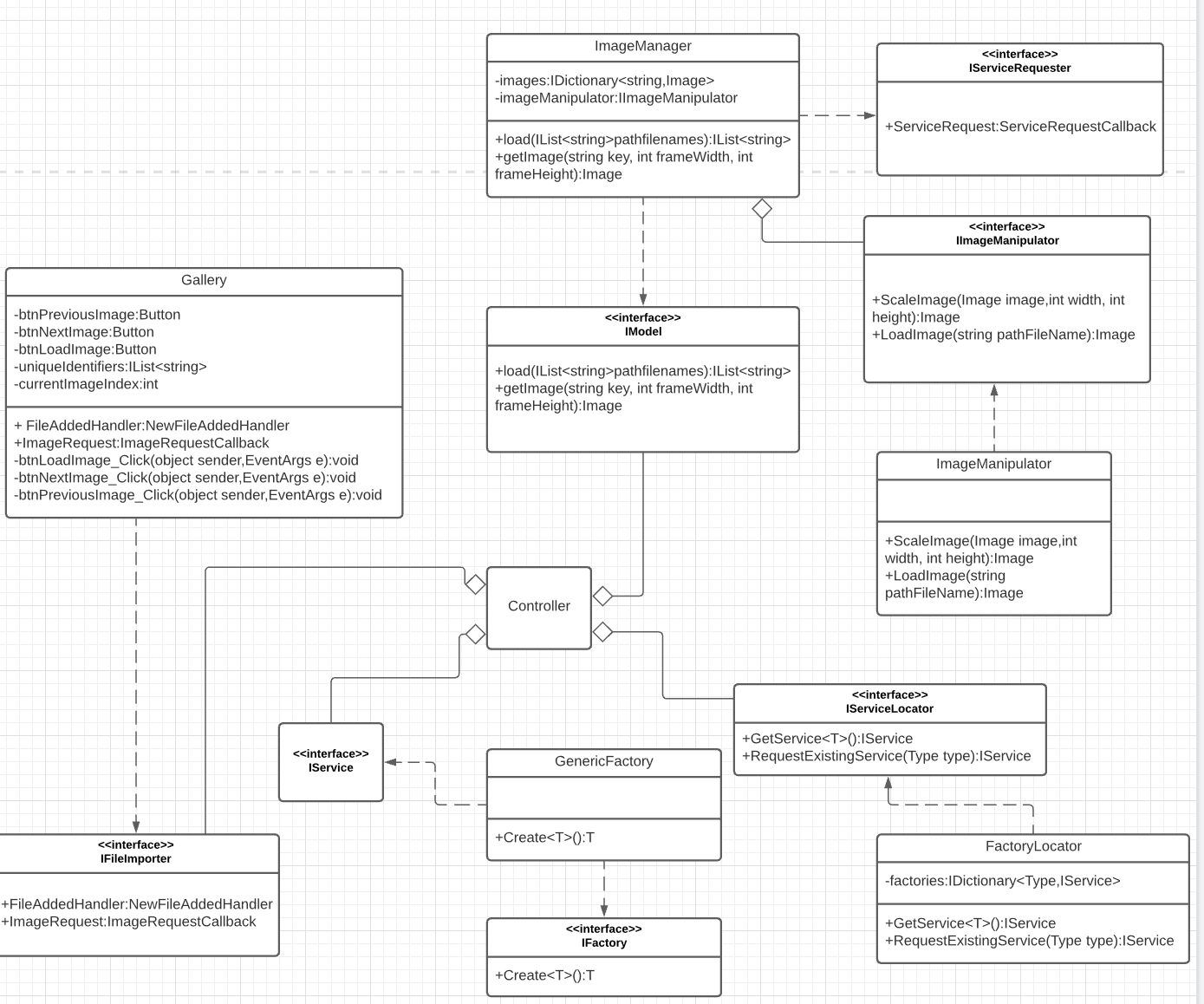
This back-end library that I am creating must include the following functionalities:

* An implementation of the IModel interface
* Hold a collection of images in memory
* A button that loads an image into the collection of images via a file chooser pop-up
* Arrow buttons to cycle through each image in the collection
* Able to store more than one image in the collection at a time
* Take an image from the collection and return a scaled version of the image as a .NET System,Drawing.Image.
* A windows form to display all the functionality in a GUI
* Minimise and close functionality on the windows form

Class Discovery

|  |  |  |
| --- | --- | --- |
| Proper noun / Instance(object) | Common noun / Class( or attribute) | Doing Verb / Operation |
| Arrow Button | **Button** | **Cycle through** |
| Load Button | **Form** | **Display images** |
| Scaled Image | **Image** | **Minimise** |
|  | **File** | **Close** |
|  | **Video** | **Scale** |
|  | **Image Manipulator** | **Popup** |
|  |  | **Store Images** |
|  |  | **Load Image** |

## OO Software Design

*This should include a brief discussion of your software, and should make use of a class diagram to help convey the overall design. This section is essentially setting-out the solution to the above problem. You may want to use citations to refer to any design patterns that are used.* 

The design of my software consists of a controller class, which instantiates the factory locator which uses the Service locator design pattern. The service locator pattern is used as a middleman between the client and the services that it uses(Kanjilal, 2016). This decouples the service from the client. The factory locator creates a generic factory, which is used to create the image manager and the Gallery class. The Abstract Factory design pattern is also used to decouple the client. It does this by taking an object as a generic type and returning a new object of that type(Abstract Factory, n.d.). This way the client only needs to know about the type of the object to create a new instance of that object. The gallery class inherits the Form class and is used to display the images from the collection of images. The collection of images is stored inside the imageManager class, which implements the IModel interface. The IModel interface is used to load and image into the collection and return a scaled image.

One problem I faced when designing the software was communicating to the IModel interface from the Gallery class, without being tightly coupled to it. To remove this issue, I implemented 2 delegate events that sit inside the Gallery class, which the ImageManger subscribes to. When the Gallery class invokes these events, imageManager will send the required information through that event. This removes any knowledge the classes have of eachother and completely removes any dependencies between the classes.

## Learning Journal

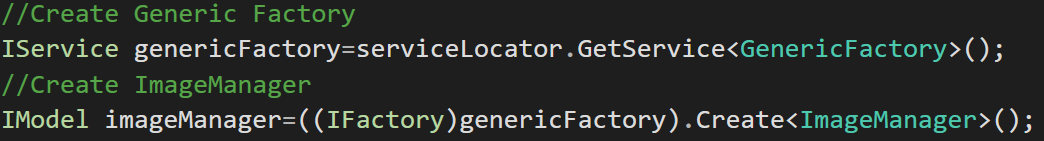
*Follow the guidance under each of the sub-headings below to evidence knowledge and application of the concepts in question.*

### Polymorphism & Interfaces

*Explain how the use of Interfaces facilitates Polymorphism, and refer to an example of this in your software.*

An interface is similar to an abstract base class with only abstract members. Any class that chooses to implement the interface must implement all of its members(Interfaces - C# Programming Guide, 2020). You cannot directly instantiate an interface. Upon implementing an interface, a class must override all its methods(polymorphism) (C# Interface, n.d.). To call upon its members, you can cast the instance of the object to an interface that it implements(C# Interface, n.d.). Alternatively, you can store the instance of the class that is implementing the interface as the interface, so you can only communicate with the portion of that class that is open to the interface. A class can inherit from its base class and also implement multiple interfaces at the same time(Interfaces - C# Programming Guide, 2020) (if the language you are using supports this), which can prove to be extremely useful when using subtype polymorphism(C# Interface, n.d.).

Polymorphism can occur when we have many classes that relate to each other through inheritance or Interfaces. By using inheritance one class can inherit members and functions from its base class(C# Polymorphism, n.d.). When we want the child classes to have the inherited function do something different to what it is doing in its siblings and/or parent class, we use polymorphism. This type of polymorphism is called method overriding(C# Polymorphism, n.d.). An example of where I have programmed to the interface and used subtype polymorphism in my projects is…



In this example I am using the IFactory interface to talk to specific methods inside the GenericFactory Class. Then I create the imageManager and store it in IModel, which means imageManager is a polymorphic object.

Interfaces facilitates polymorphism by allowing us to define polymorphism in a declarative way, unrelated to implementation. Without using interfaces, there is no way to enforce the polymorphism onto the class, except in informal ways, or language specific ways. By using interfaces, you can guarantee that there is no mystery behind which classes are to be using the abstract methods through polymorphism.

### Design Principles

*Explain what Cohesion and Coupling are, and how Cohesion and Coupling of a software system can be assessed using the five SOLID Design Principles.*

Cohesion refers to how focused a classes purpose is(Cohesion in Java - GeeksforGeeks, 2017). The more focused the purpose, the higher the cohesion and the better the OO design(Cohesion in Java - GeeksforGeeks, 2017). One design pattern that closely relates to cohesion is the single responsibility principle, as when your code has a single responsibility, it is automatically cohesive.

Coupling is the principle of “separation of concerns” (Lambert, 2012). When one or more of your software components depend on each other, then they are seen as tightly coupled together. The opposite of this, which is what we want to aim for in software design is called loose coupling(Lambert, 2012).

Open/Closed Principle means that software entities should be open for extension but closed for modification(Janssen, 2018). So, by providing a solution that doesn’t require modification every time a change to the system needs to be made, we can extend the behaviour of a class without modifying that class. By using Open Closed Principle in your design, you can make sure there are no tight links between modules and the dependency between entities will be at the minimum(Janssen, 2018). When you use interfaces with the open/close principle instead of super classes, this adds another level of abstraction which enables loose coupling(Janssen, 2018).

To follow the Liskov substitution principle, objects of a superclass must be replaceable with objects of its subclasses, without breaking the application(Awad, 2020)This principle helps with creating a good, loosely coupled design because any instance that extends the base class can be used without needing to know what the behaviour of the replaced object does and how it functions and removes the dependencies of these classes(Janssen, 2018).

The interface segregation principle states that a class should not be forced to depend upon interfaces which they don’t use(Awad, 2020). This makes a design highly cohesive, as it keeps the entities as focused as possible. It also loosely couples your code, because it reduces the number of dependencies that class has to others, as they are not publicising any methods for no reason(Awad, 2020).

Dependency Inversion principle has two main principles:

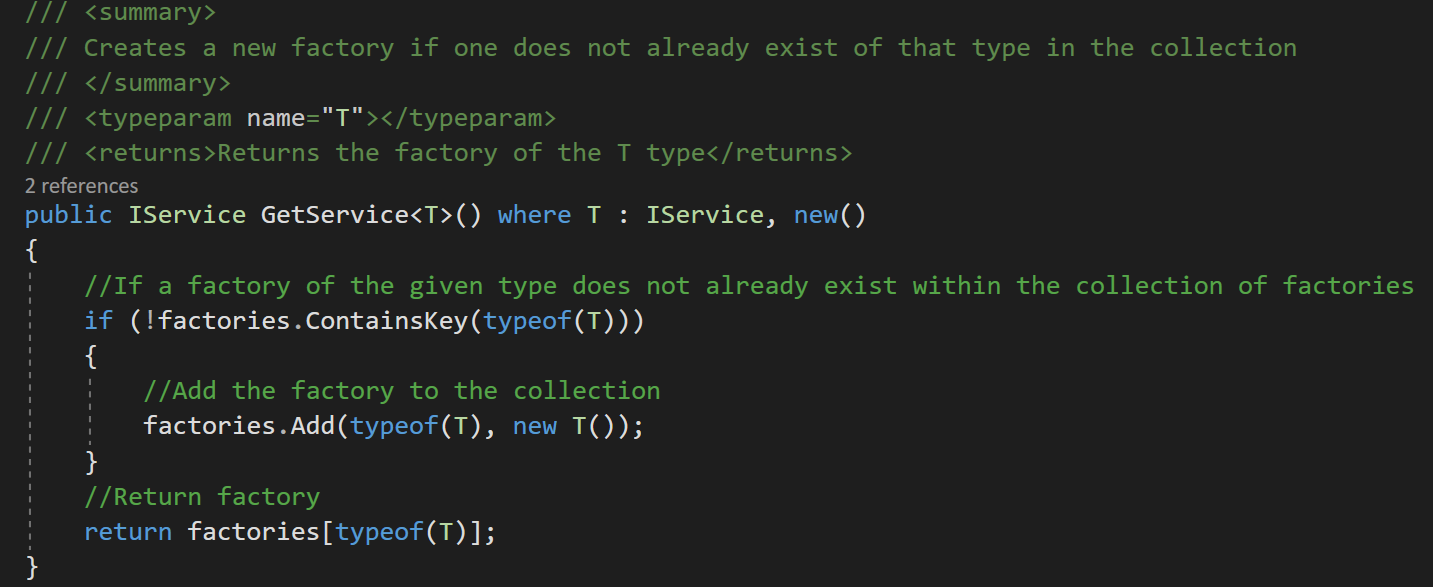
1. “High level modules should not depend on low-level modules. Both should be dependent on the abstractions.”
2. Abstractions should not depend on details. Details should depend on abstractions.” (Janssen, 2018)

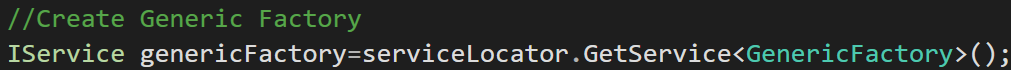
This principle enforces decoupling high-level components from the low ones and implements a layer of abstraction between each component(Janssen, 2018).

### Design Patterns

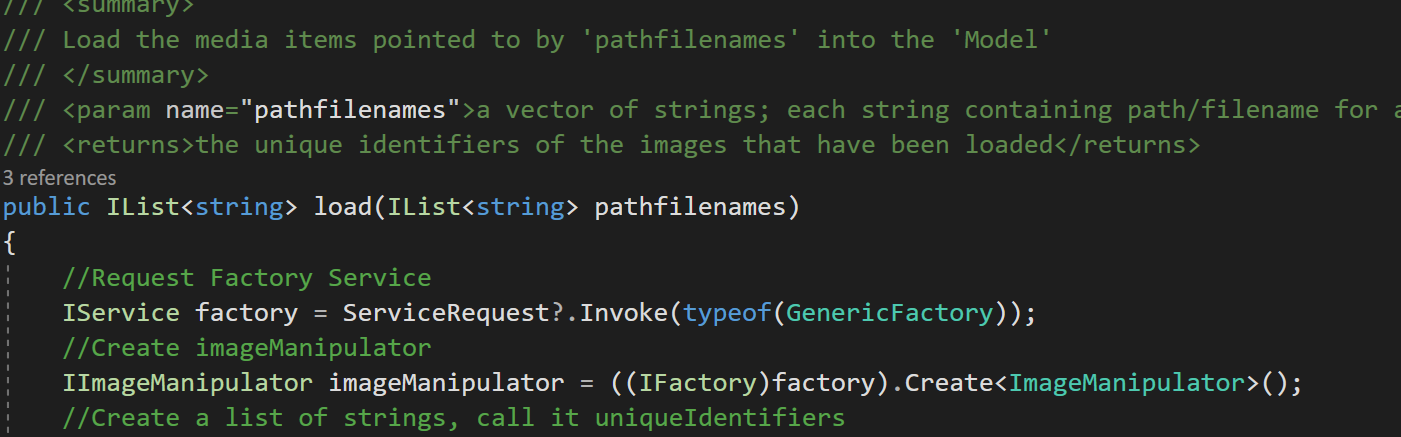
*Explain at least two Design Patterns you have used in your software design to strengthen Cohesion and/or loosen Coupling – for each pattern, name the pattern, explain its structure, and explain how/why it helped to strengthen Cohesion and/or loosen Coupling.*

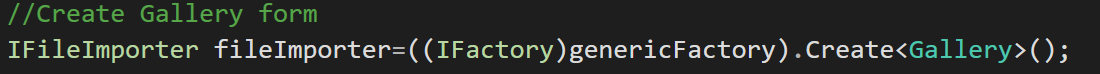
To strengthen cohesion and loosen coupling, I have used a few certain design patterns to help me remove some dependencies and focus the scope of classes. The first design pattern I have used is called the service locator design pattern. The service locator pattern is used to encapsulate the processes involved in accessing a service(Service Locator Pattern - GeeksforGeeks, 2018). A service locator class will return an instance of a service when a client requests that service. This removes the dependency between the client and the service(Service Locator Pattern - GeeksforGeeks, 2018). In my software I made a FactoryLocator, which creates, stores and returns an instance of a factory. To further the decoupling of the service location process, I have created an interface called IServiceRequester. This interface contains an event which the service locator subscribes to and when this event is invoked, the service locator returns a service of the passed type. By using the type as a parameter, I am not affecting the coupling of the class that is using the service locator. This is an example of using reflection. This is when you dynamically create an instance of a type, then link this to an object that already exists within the application (Altvater, 2017). I also used this design pattern to strengthen the cohesion of the controller class, as it takes the job of storing a collection of services away from the controller class and puts it into a more focused class. Below is example of an implementation of the Service locator.





The other design pattern I used is the abstract factory design pattern. The way this design pattern works is it takes a generic type from the client and then creates a new instance of that type and returns it to the client, usually stored as a more abstract type like an interface(Abstract Factory, n.d.). This removes any dependency the client would have to that class that they wish to create, decoupling it from the client. It also strengthens the cohesion of the client, as they no longer have the responsibility of creating entities, this now belongs to the factory class. This design pattern was perfect for my software as it decouples my controller class, which main job is creating instances of objects and decouples + strengthens the cohesion of the imager manager class, as it no longer has the job of creating the ImageManipulator instance and removes the dependency between the ImageManipulator class and the ImageManager class.





## References

*List your references here using the Harvard referencing style.*

Docs.microsoft.com. 2020. *Interfaces - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/> [Accessed 10 March 2021].

W3schools.com. n.d. *C# Polymorphism*. [online] Available at: <https://www.w3schools.com/cs/cs\_polymorphism.asp> [Accessed 10 March 2021].

Lambert, S., 2012. *Quick Tip: The OOP Principle of Coupling*. [online] Game Development Envato Tuts+. Available at: <https://gamedevelopment.tutsplus.com/tutorials/quick-tip-the-oop-principle-of-coupling--gamedev-1935> [Accessed 10 March 2021].

GeeksforGeeks. 2017. *Cohesion in Java - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/cohesion-in-java/> [Accessed 10 March 2021].

W3schools.com. n.d. *C# Interface*. [online] Available at: <https://www.w3schools.com/cs/cs\_interface.asp> [Accessed 10 March 2021].

Janssen, T., 2018. *SOLID Design Principles Explained: The Open/Closed Principle with Code Examples*. [online] Stackify. Available at: <https://stackify.com/solid-design-open-closed-principle/> [Accessed 10 March 2021].

Janssen, T., 2018. *SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples*. [online] Stackify. Available at: <https://stackify.com/solid-design-liskov-substitution-principle/> [Accessed 10 March 2021].

Awad, H., 2020. *Contribution of SOLID principle to low coupling and high cohesion pattern*. [online] Medium. Available at: <https://medium.com/dev-genius/contribution-of-solid-principle-to-low-coupling-and-high-cohesion-pattern-973c33574572> [Accessed 10 March 2021].

Janssen, T., 2018. *SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples*. [online] Stackify. Available at: <https://stackify.com/dependency-inversion-principle/> [Accessed 10 March 2021].

Refactoring.guru. n.d. *Abstract Factory*. [online] Available at: <https://refactoring.guru/design-patterns/abstract-factory> [Accessed 11 March 2021].

Kanjilal, J., 2016. *How to implement the ServiceLocator design pattern*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3104888/how-to-implement-the-servicelocator-design-pattern.html> [Accessed 11 March 2021].

GeeksforGeeks. 2018. *Service Locator Pattern - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/service-locator-pattern/> [Accessed 11 March 2021].

Altvater, A., 2017. *How C# Reflection Works With Code Examples*. [online] Stackify. Available at: <https://stackify.com/what-is-c-reflection/> [Accessed 11 March 2021].