***Programming : Assignment-1***

**Introduction -**

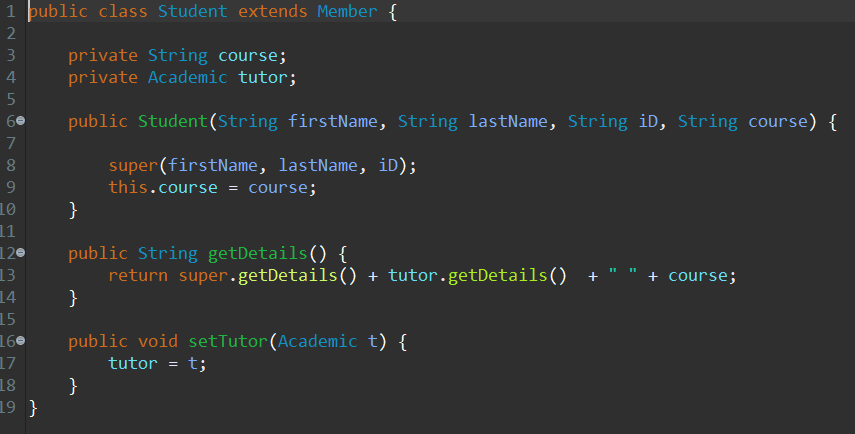
This report will examine the key components related to the object-oriented programming paradigm, analysing design pattern types. It will also examine the characteristics of the object-oriented paradigm as well as the various class relationships. It will show a design and build of a class diagram using UML tools. It will also Determine a design pattern from each of the creational, structural and behavioural pattern types. The definitions for specific designs using a UML tool. Analysis of the relationship between the object-oriented paradigm and design patterns.

**Task 1 -**

Term Descriptions -

**Encapsulation -**

It is an object- oriented feature that wraps data which can be variables and code into a single class. It is done so that the information is covered up to everything outside the class scope, and must be gotten to and altered through its own part capacities.

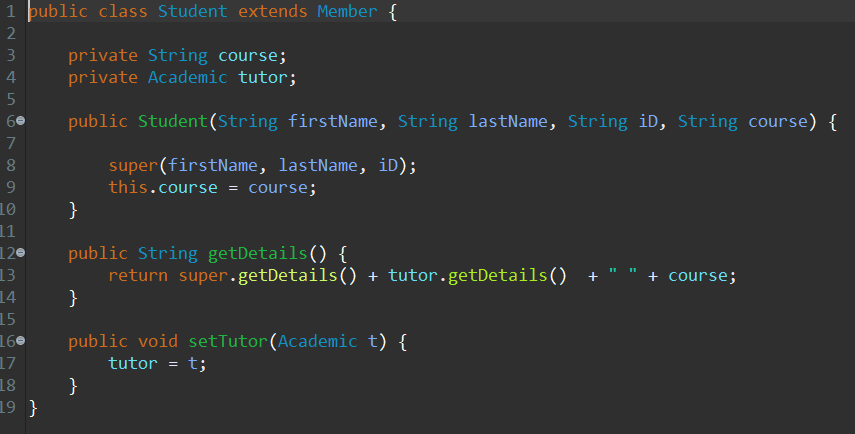


**Example -**

The code in the above image from **appendix B** shows an example of encapsulation in line 3 - 4 where there's classes which have variables declared as a private class and from line 6 - 9 are public classes which have setters and getter methods which can change the values of a variable. This is the thing that exemplification does, it conceals the execution and gives us the qualities we need.

**Inheritance -**

Inheritance is an object- oriented feature of a system where one class secures the property of another class. The concept behind Java's inheritance is that you can build new classes that are based on established classes. When you derive from an established class, you can reuse the parent class methods and fields. You may also add additional methods and fields to the existing class.

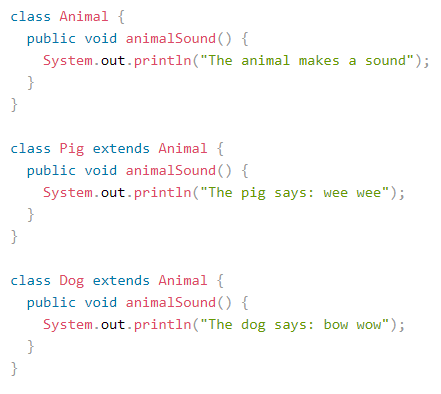


**Example -**

The code above image from **Appendix B** in line 1 demonstrates inheritance where The class student is inheriting the properties and methods of Member class. In this case the Member class is the parent class whose properties and functions are inherited by the student class and the student class is the child class that extends the features of the Member class.

**Polymorphism -**

Polymorphism is an object- oriented feature that is considered as one of the significant highlights of Object Oriented Programming. Polymorphism permits us to play out a solitary activity in various manners. It permits the characterization of one interface and has numerous executions. "Poly" signifies numerous and "transforms" signifies structures, So it implies numerous structures.

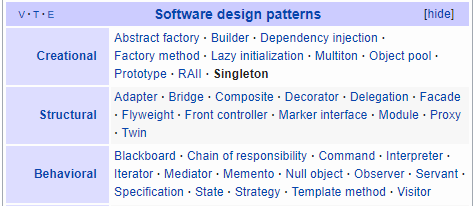


**Examples -**

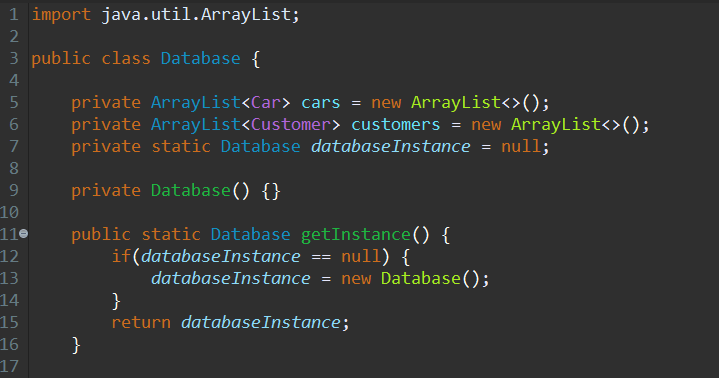
The code in the above example image demonstrates polymorphism where the class “Animal’ where sound() are made by different animals, which have varied implementations for each animal. It is the ability of a strategy to do various things dependent on the article that it is following up on. As it were, polymorphism permits you characterize one interface and have numerous executions.

**Singleton Pattern -**

Singleton pattern is a design pattern that has the simplest model which contains only one class which is shared by multiple other object classes with separate connections to other classes.



**Creational Design Pattern :** Singleton pattern follows a Creational design pattern that has only one object in a whole class. In the above image shows the categories that are listed in a Software design pattern. Creational patterns are patterns that offer a way to construct objects while covering the creational meaning, this allows us versatility to build objects depending on specific use cases, some examples are singleton, warehouse, etc, so depending on the use case, you can determine which type to use.

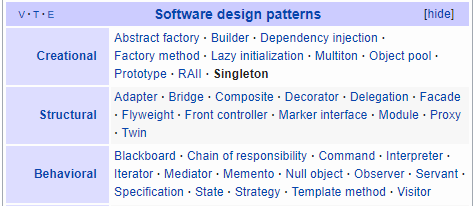


**Example -**

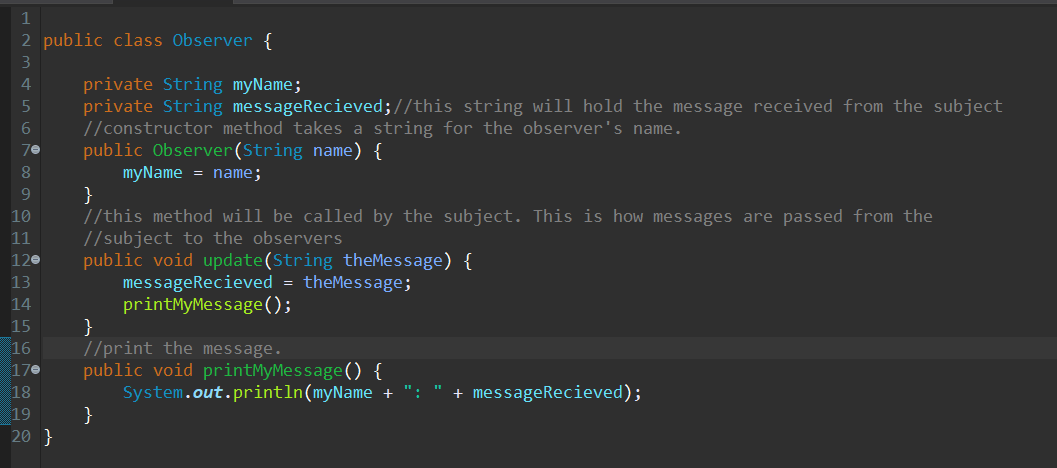
The code in the above example describes the singleton pattern which is a ‘Database’ class which stores multiple objects. A private arraylist is created for cars and customers and a private database instance is set to 0. These are all single objects linked to other classes within a design pattern.

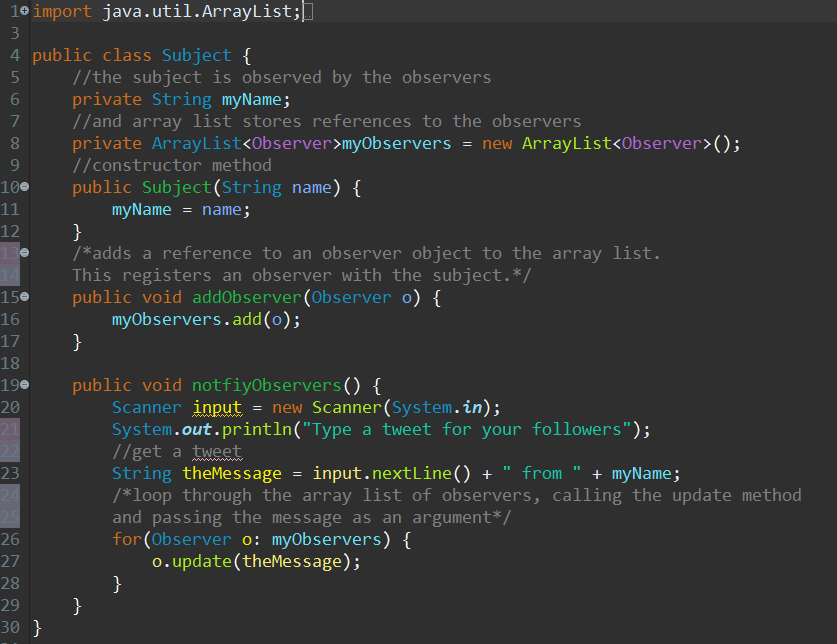
**Observer pattern -**

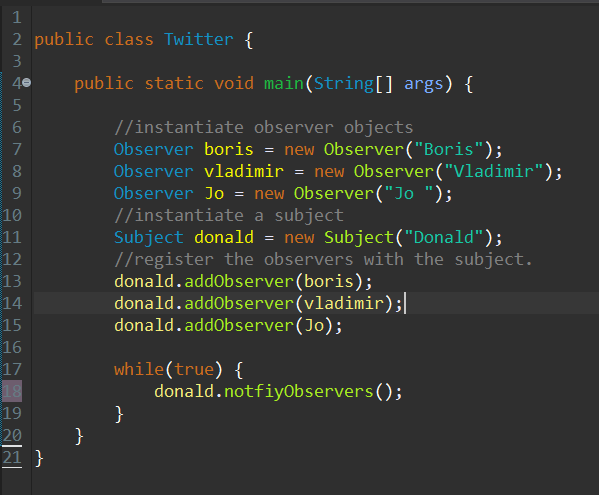
Observer pattern is a template of behavioural architecture. Specifies correspondence between objects: visible and observers. An observable entity is an entity that reminds observers about shifts in its environment.



**Behavioural Design Pattern :** Observer pattern follows a Behavioral design pattern that has only one object in a whole class. In the above image shows the categories that are listed in a Software design pattern. Behavioral design pattern involves coordination between items, hence increasing consistency between artefacts, some examples include perception patterns, chain of obligation patterns, etc.

****

****

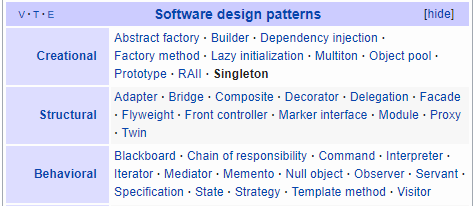
****

**Example -**

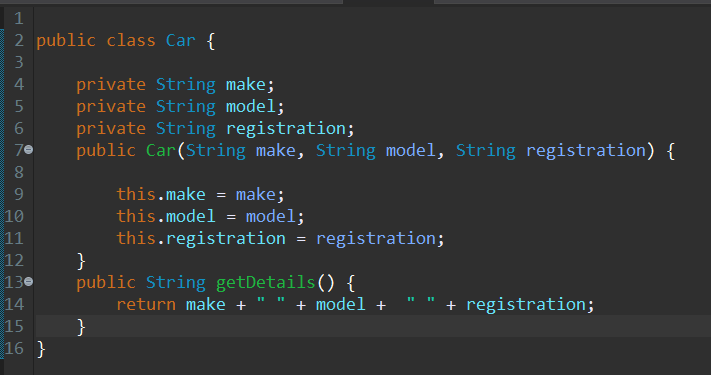
The code above image describes the observer pattern which observer observer which is an object and twitter is the client class. This occurs when a program has underground modification.

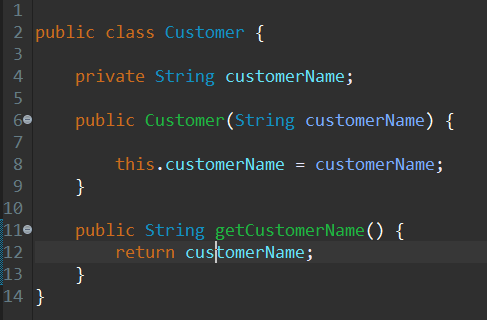
**Model-View-Controller pattern -**

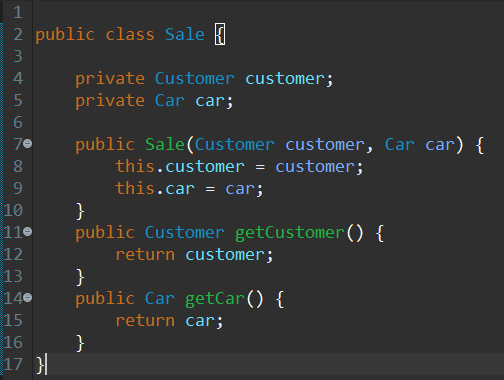
Model-View-Controller pattern where ‘Model’ is a one-relationship between the model and its components, on the one side, and the universe depicted, on the other side, as viewed by the owner of the concept.’View’ is a connected to the configuration (or part of the configuration) and gets the data required for the product presentation by asking questions. The layout can also be modified by submitting correct messages.’Controller’ is the interface between the consumer and the machine. It gives feedback to the consumer by creating plans for specific views to be shown in correct positions on the device. Provides means of user performance by providing menus or other ways of offering commands and details to the user.



**Structural Design Pattern :** Model-View-Controller pattern follows a Structural design pattern that has only one object in a whole class. In the above image shows the categories that are listed in a Software design pattern. Structural Design presents us with means of club groups and objecting together to shape large constructs, some instances are adapters, bridges, etc.

****

****

****

**Example -**

In the above example the code describes the Model-View-Controller which is the Car-Customer-Sales class. Here the car class is the ‘Model’ in which it contains the make, model and register, of the car. There ought to be a coordinated correspondence between the model and its parts from one viewpoint, and the world as seen by the proprietor of the model then again.

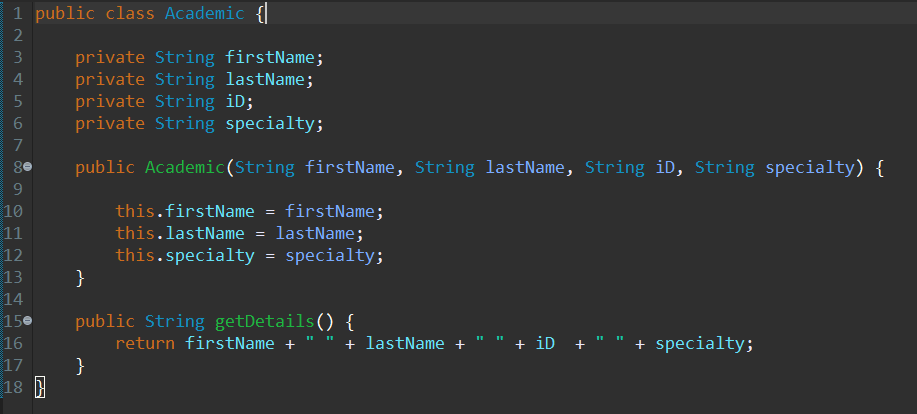
The ‘Customer’ is the ‘Viewer’ which contains the name of the customer’s name which is a view that is joined to its model (or model part) and gets the information fundamental for the introduction from the model by posing inquiries. It might likewise refresh the model by sending proper messages. Every one of these inquiries and messages must be in the wording of the model, the view will in this way need to know the semantics of the traits of the model it speaks to.

The ‘Sale’ class is the controller which ‘Controls’ the ‘Customer’ and the ‘Car’ class. It is a link between the user customer and the system. It gives feedback to the consumer by creating plans for specific views to be shown in correct positions on the device. Provides means of user performance by providing menus or other ways of offering commands and details to the user. The controller collects these user data, converts it into the correct messages, and transmits such messages to one or more views.

**Aggregation -**

Aggregation is an object- oriented feature that has a relationship between two different classes on the basis of their objects. They are linked to each other and can’t exist if there is no other class it depends on. It permits you to configure classes that follow great Object Oriented practices. It likewise gives code reusability.



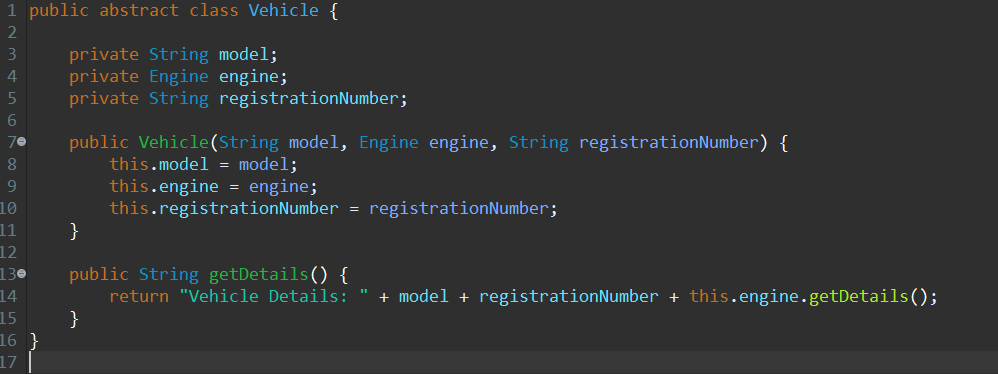


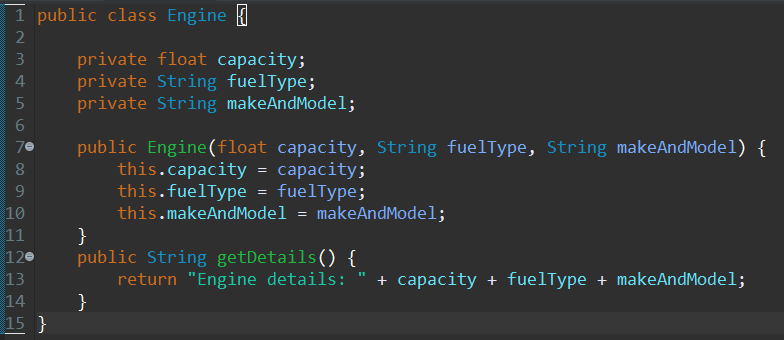
**Example -**

The code in the above example from **Appendix B** describes aggregation where there are two classes which are essentially dependent on each other to exist. Here Academic class depends on the Data repository class to exist, as the first name. Last name, ID, and speciality is stored into the data repository class to function. The Data Repository class depends on the Academic class to get variables to store in its class to function. Hence both are dependent on each other to function and are Aggregated with each other.

**Composition -.**

Composition is an object- oriented feature that helps to model objects that is made of other objects in class. It can control the perceivability of other objects to customer classes and reuse just what we needed to permit production of back-end class when it is required.





**Example -**

The code in the above example from **Appendix C** describes Composition which has a Vehicle class and an Engine class a class uses the functionality of another class as they are both using objects from each other, in this case the vehicle class is dependent on the engine class as a vehicle can’t function without whereas the engine class is dependent on the vehicle class as an engine can’t function without a vehicle. Therefore, both the classes need the other to function together using objects from each other

**Task 2 -**

**a) UML Diagrams -**

**Appendix B -**

****

In the above UML diagram use an **‘Model-View-Controller Design pattern’.** It follows a **“Structural Pattern”** type as it is the best way to create an object.

In the above example the UML describes the Model-View-Controller which is the Data Repository-College Database-User Interface class. Here the ‘Data Repository’ class is the ‘Model’ in which it contains the students and academics of the class.

The ‘College Database’ is the ‘View’ which contains the name of the student’s name which is a view that is joined to its model (or model part) and gets the information fundamental for the introduction from the model by posing inquiries. It might likewise refresh the model by sending proper messages. Every one of these inquiries and messages must be in the wording of the model, the view will in this way need to know the semantics of the traits of the model it speaks to.

The ‘User Interface’ class is the controller which ‘Controls’ the input and the academic. It is a link between the Data Repository, College Database and Academic classes. It gives feedback to the student by creating plans for specific views to be shown in correct positions on the device. Provides means of user performance by providing menus or other ways of offering commands and details to the user. The controller collects these user data,

**Appendix C -**

****

In the above UML diagram uses a **“Factory Design Pattern”.** It follows a **“Creational Pattern”** type as it is the best way to create an object. Which have many classes of a similar type that inherit from a parent class, we can use a factory class to instantiate objects, rather then instantiate the objects explicitly:

**Appendix D -**

****

In the above UML diagram use an **‘Observer Design pattern’.** It follows a **“Behavioral Pattern”** type as it is the best way to create an object. Several objects (called Observers) observing one object( called the Subject). Twitter is a conceptual example of the observer pattern. The subject tweets and the observers receive the tweets.

**B) Appendix D modified to represent a Singleton version -**

****

**Conclusion -**

In conclusion, This report examines the key components related to the object-oriented programming paradigm, analysing design pattern types. It also examines the characteristics of the object-oriented paradigm as well as the various class relationships. It shows the design and build of a class diagram using UML tools. It also Determines a design pattern from each of the creational, structural and behavioural pattern types. It defines specific designs using a UML tool. Analysis of the relationship between the object-oriented paradigm and design patterns.

**Resources -**

Guru99.com. 2020. *Encapsulation In Java Oops With Example*. [online] Available at: <https://www.guru99.com/java-oops-encapsulation.html> [Accessed 24 March 2020].

Javatutorial.net. 2020. *Java Encapsulation Example | Java Tutorial Network*. [online] Available at: <https://javatutorial.net/java-encapsulation-example> [Accessed 25 March 2020].

beginnersbook.com. 2020. *Inheritance In Java Programming With Examples*. [online] Available at: <https://beginnersbook.com/2013/03/inheritance-in-java/> [Accessed 25 March 2020].

www.javatpoint.com. 2020. *Encapsulation In Java - Javatpoint*. [online] Available at: <https://www.javatpoint.com/encapsulation> [Accessed 25 March 2020].

www.javatpoint.com. 2020. *Polymorphism In Java - Javatpoint*. [online] Available at: <https://www.javatpoint.com/runtime-polymorphism-in-java> [Accessed 25 March 2020].

beginnersbook.com. 2020. *Polymorphism In Java With Example*. [online] Available at: <https://beginnersbook.com/2013/03/polymorphism-in-java/> [Accessed 25 March 2020].

Atwood, J., 2020. *Understanding Model-View-Controller*. [online] Coding Horror. Available at: <https://blog.codinghorror.com/understanding-model-view-controller/> [Accessed 25 March 2020].

GeeksforGeeks. 2020. *Singleton Design Pattern | Implementation - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/singleton-design-pattern/> [Accessed 25 March 2020].

Baeldung.com. 2020. [online] Available at: <https://www.baeldung.com/creational-design-patterns> [Accessed 25 March 2020].

Sourcemaking.com. 2020. *Design Patterns And Refactoring*. [online] Available at: <https://sourcemaking.com/design\_patterns/observer> [Accessed 25 March 2020].

2020. [online] Available at: <https://www.quora.com/What-is-the-difference-between-Creational-Structural-and-Behavioral-Patterns> [Accessed 25 March 2020].

Sites.google.com. 2020. *Singleton Pattern With Example - Wcf Pandu*. [online] Available at: <https://sites.google.com/site/wcfpandu/singleton-pattern-with-example> [Accessed 27 March 2020].