Design Patterns

Table of Contents

[Types of Design Patterns 2](#_Toc100620141)

[J2EE Design Patterns 3](#_Toc100620142)

[MVC Architecture 3](#_Toc100620143)

[Singleton Pattern 4](#_Toc100620144)

[Front Controller Pattern 5](#_Toc100620145)

[Data Access Object Pattern 6](#_Toc100620146)

A **software design pattern** is a general, reusable solution to a commonly occurring problem. These patterns are specific to particular scenarios, and the solutions are observed through experience. However, the patterns are not code. They are specifications about how to write the code.

Using design patterns prevents us from reinventing the wheel. Using an existing well-known and optimized algorithm makes it easier to communicate the design. Multiple design patterns can also be used together to solve the larger problem.

## Types of Design Patterns

There are mainly three categories of design patterns, Creational patterns, Structural patterns and Behavioural patterns.

**Creational patterns** provide ways to create objects while hiding the creation logic, i.e. we do not directly use constructors. This allows us to have better control over how the creation mechanisms work. Some examples of creational patterns include the Singleton pattern, the Factory Method pattern, etc.

**Structural patterns** use things like interfaces to define how objects will be composed. Examples include the Adapter pattern, the Bridge pattern, the Façade pattern, etc.

**Behavioural patterns** deal with communication methods between objects. Examples include the Mediator pattern, the Observer pattern, the State pattern, etc.

## J2EE Design Patterns

The **J2EE Design Patterns** are design patterns specific to applications that use J2EE/JEE technology. These patterns are classified based on their logical tier.

|  |  |
| --- | --- |
| **Tier** | **Pattern Name** |
| Presentation Tier | Intercepting Filter, Front Controller, View Helper, Composite View, Service to Worker, Dispatcher View |
| Business Tier | Business Delegate, Value Object, Session Façade, Composite Entity, Value Object Assembler, Value List Handler, Service Locator |
| Integration Tier | Data Access Object, Service Activator |

From these, we will be looking into the Front Controller pattern, the Session Façade pattern and the Data Access Object pattern specifically.

## MVC Architecture

We have previously seen the **MVC Architecture**, which is used to separate business logic from the presentation layer. Such separation allows multiple views to share the same data model, making it easier to support multiple clients and frequent changes to the view.

## Singleton Pattern

The **Singleton Pattern** is a Creational Design Pattern which essentially ensures that only a **single object** of a class exists at any given moment of time.

public class SingleObject {  
 private static volatile SingleObject *instance* = null;  
 private SingleObject() {};  
 public static SingleObject getInstance() {  
 if (*instance* == null) {  
 synchronized(SingleObject.class) {  
 *instance* = new SingleObject();  
 }  
 }  
 return *instance*;  
 }  
}

JAVA

In the above code, we have made the main constructor **private**. This ensures that other classes are unable to create new objects of this class. They must use the getInstance() method instead.

The *instance* object is initially null. This is called **lazy initialization**, when a property is not initialized at the time the class is created.

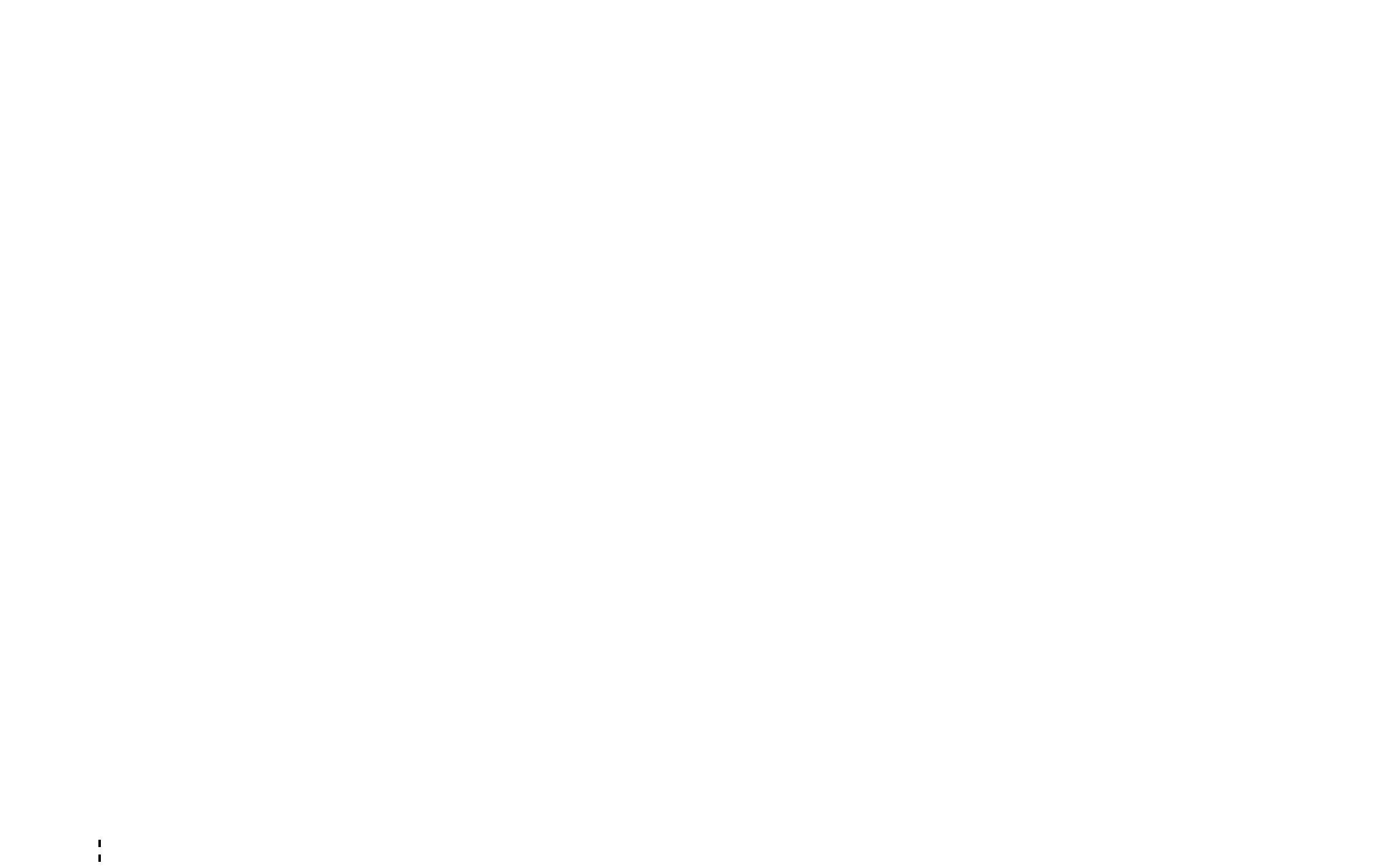
The synchronized keyword is used to make the process **thread safe**. Imagine that two different threads access the getInstance() method at the same time, both finding that the *instance* object is currently null. In this case, they might both initialize the *instance* object, which will lead to two objects being created. Making the process thread safe means that two threads cannot access that section of the method simultaneously. Essentially, there is a **lock**.

Now consider that one thread is in the middle of initializing the class. At this moment, another method finds that the *instance* object has been initialized and tries to access some other properties of the class. However, those properties have not yet been initialized. To make sure this does not happen, we use the volatile keyword, which ensures that all the other properties are also initialized before the lock is released.

## Front Controller Pattern

For a variety of reasons, we need to make sure that there is a **centralized access point** for all requests. We have previously done this for things like user authentication, but it can also be used to handle navigation, i.e. instead of directly telling views which buttons take them where, we can direct them to a controller instead and make the controller decide. This is the **Front Controller Pattern**.

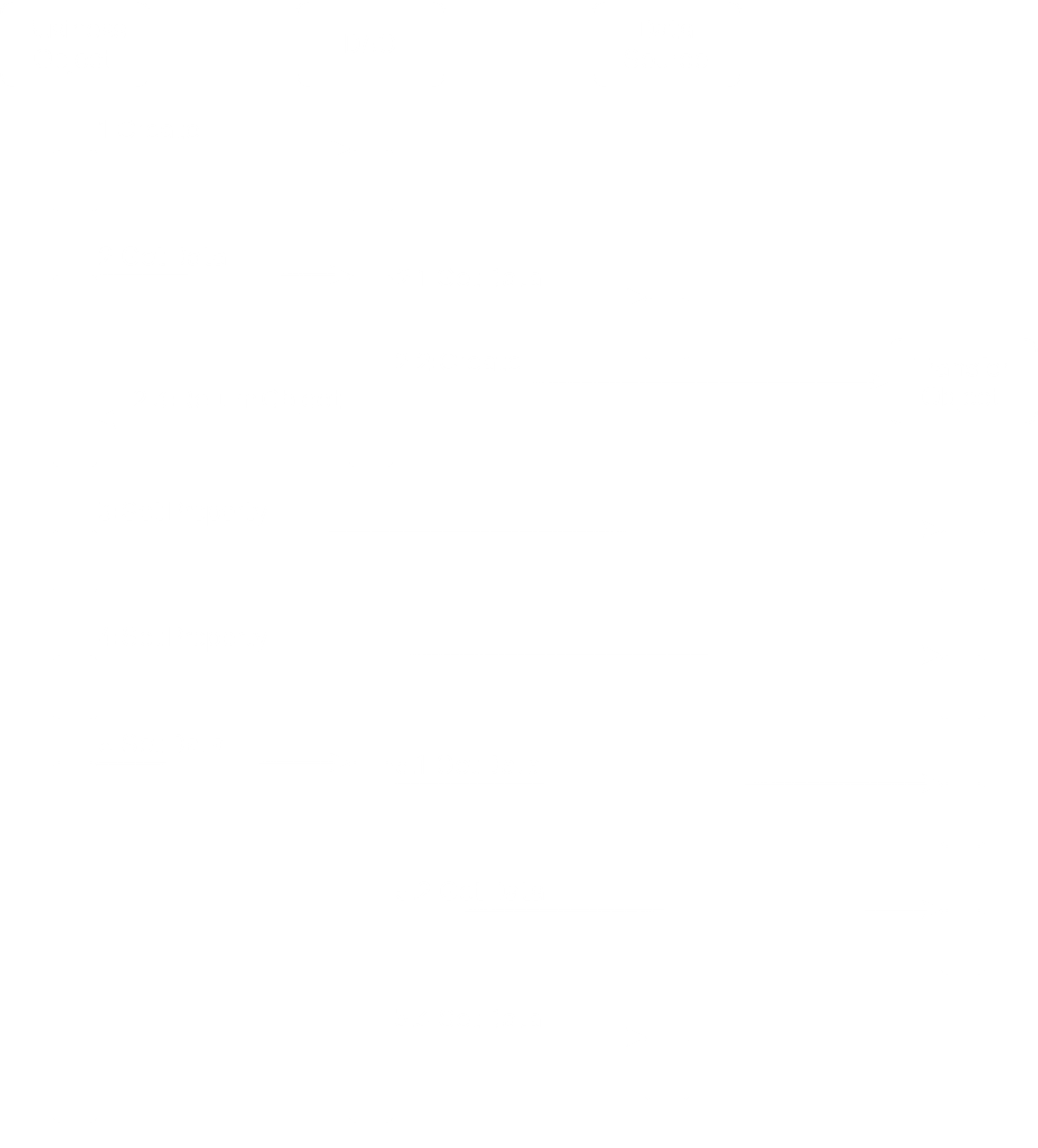
Note that the controller itself, which is a **servlet**, does not do much work. In simply delegates the work. To do this, it may use a **dispatcher**, which handles navigation, or a **helper class**, which handles processing.



## Data Access Object Pattern

When using **persistent data** (data that must be remembered even if the application shuts down), different storage mechanisms can be used. The easiest interpretation of this is just that we might use different database types. Each of the database types have their own ways of accessing the data.

If we have separate methods that each handles the process of access the data on a different database, and we use whichever method is appropriate, we are constantly having to deal with the implementation logic. Not separating this from our business logic makes our lives more complicated. The **Data Access Object Pattern** can be used to abstract and encapsulate the implementation logic so that the business logic does not need to become tightly coupled with it.



In the diagram above, the **Business Object** represents the object handling business logic, the **Data Access Object** provides the abstraction, the **Data Source** represents the different databases and the **Data Transfer Object** represents an object used to carry data.

Example

Consider that we have a Student class, which has the properties id and name. This class is the Data Transfer Object. It will be used to store the data from a single record from the database and transfer that data.

We will have an **interface**, say the *SampleDaoInterface*, which defines the methods that will be available to clients. For example, we could have one method, getStudent(int id), which retrieves a record from the database using the id and returns a Student object based on it.

Finally, we will have **multiple implementations** of the interface, suppose SampleDaoImplOracle, SampleDaoImplMongo, etc., each of which implements the logic to access the different databases. Client-side applications will use whichever object they need, but will be able to use the same methods due to the interface implementation.