

**Abdi-rahman Musse | @00493736 | SGB315 | Design Patterns**

**| Number Quiz Game**

Table of Contents

[Summary 2](#_Toc5978794)

[Introduction 2](#_Toc5978795)

[Use Case Diagram 2](#_Toc5978796)

[UML Class Diagrams 3](#_Toc5978797)

[Design Patters 5](#_Toc5978798)

[**MVC (Model-view-conTRoller)** 5](#_Toc5978799)

[**delegation** 5](#_Toc5978800)

[**Factory** 5](#_Toc5978801)

[**Strategy** 6](#_Toc5978802)

[Conclusion 6](#_Toc5978803)

[**improvements** 7](#_Toc5978804)

[References 7](#_Toc5978805)

# Summary (Abstract)

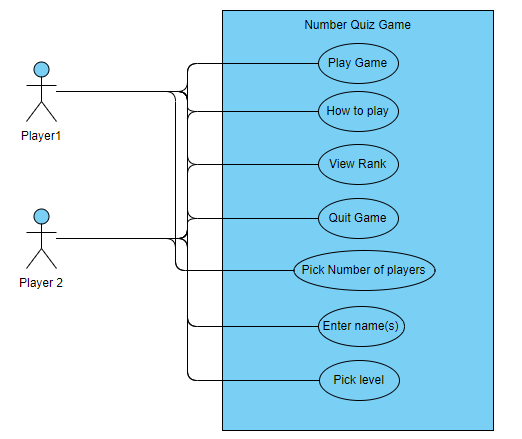
Design patterns are a crucial aspect to consider when making large applications. It helps developers solve problems and decomposes the system into different appropriate objects. This allows for encapsulation, improves flexibility, reusability and performance. Another important reason to use design patterns is to improve the communication between developers so that other developers can better understand what you’re working on. This report illustrates my understanding of design patterns and shows how I’ve implemented them into my application which is building an educational game (Number Quiz) for primary school children. Use Case Diagrams and Class Diagrams will be used to further explain the process and my break down of the game which will show the design patterns that I have used.

# Introduction

The main objective of this project is to build an educational game for young children by using eclipse and javaFX2. This is to test my understanding of design patterns and its application. The design patterns that were used are: Factory, Strategy, Delegation, and MVC. Each one will be talked about in depth in further sections. Lambda expressions were also used since it is concise, improve readability, and reduces the JAR file size (McKenzie, 2018). Another goal of this project is to demonstrate responsibility driven design. Therefore, a use case diagram will be used to represent what the user can do and a class diagram to represents the structure of the game and the classes that it holds.

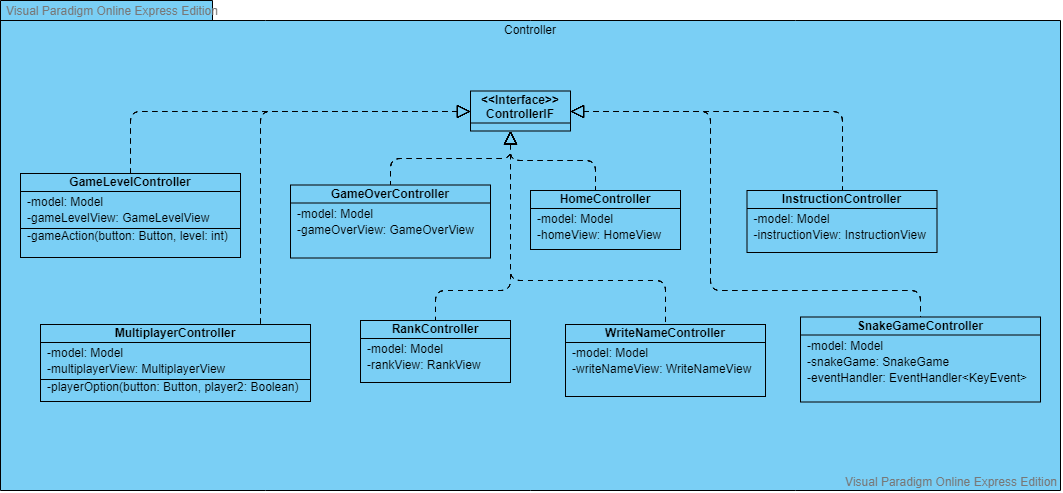
# Use Case Diagram

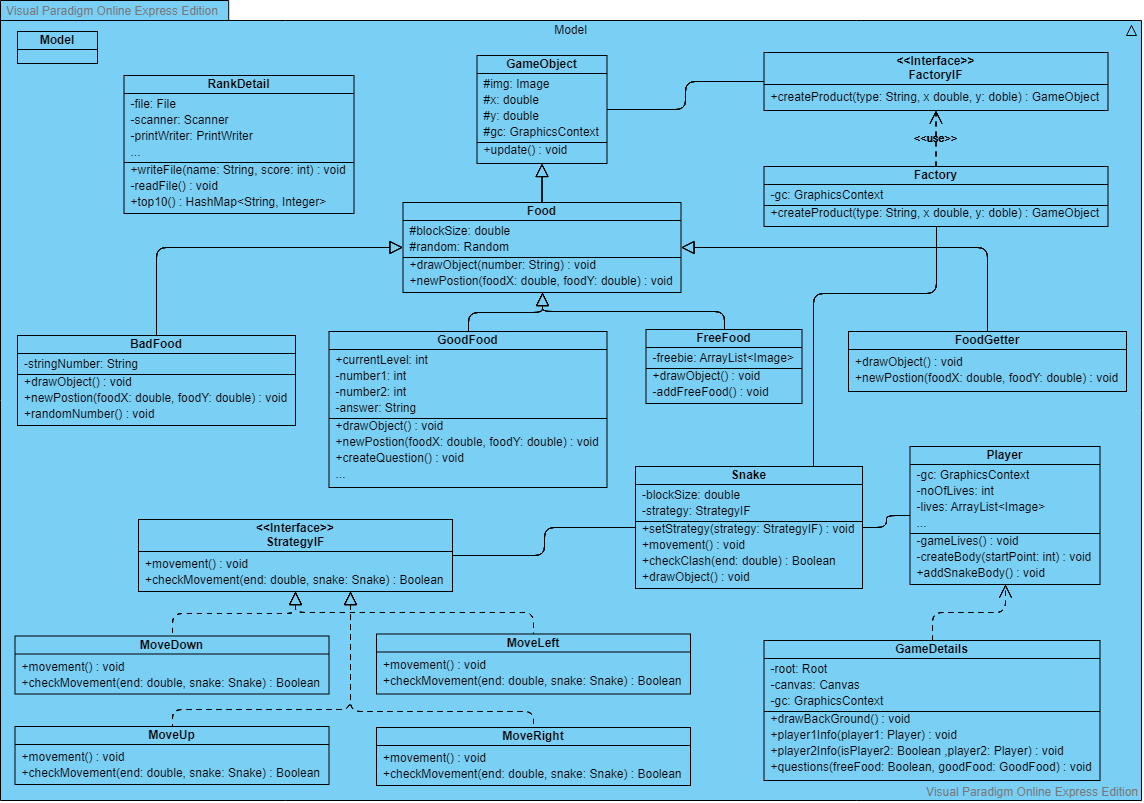
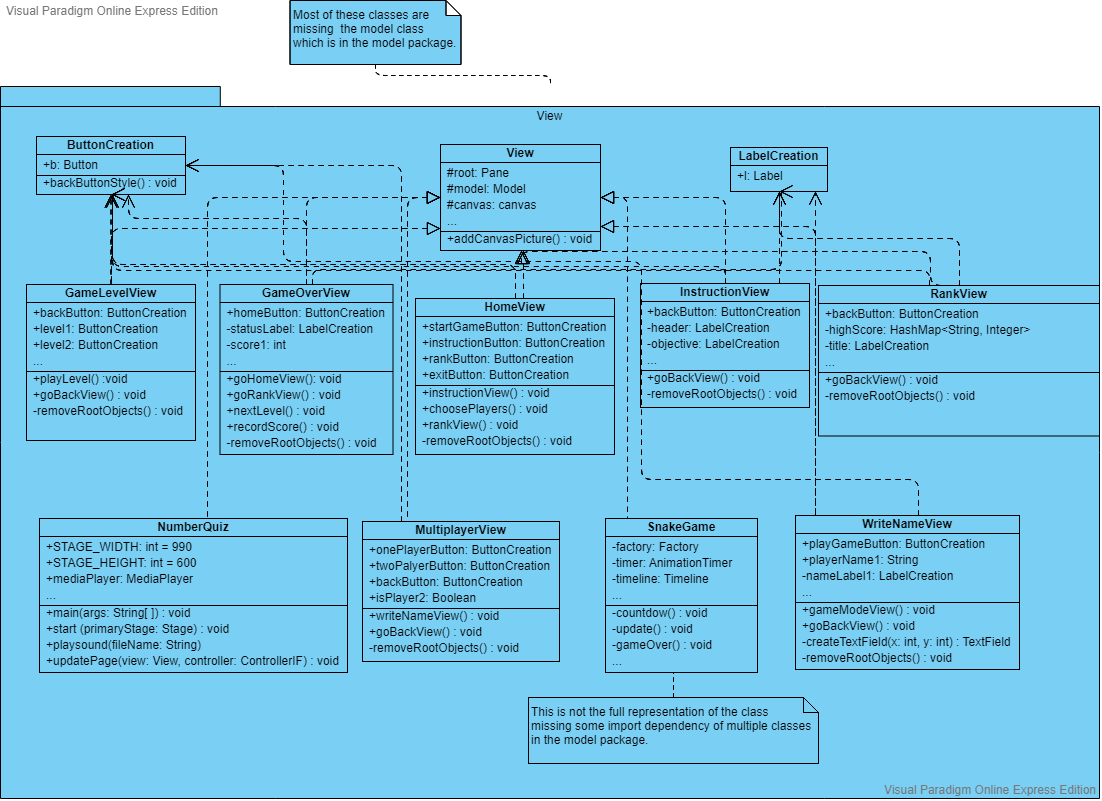
As shown in the diagram below, we have 2 primary actors which are the players (player 2 is optional). These are shown on the left-hand side of the diagram. The right-hand side is empty since we don’t have any secondary actors. The box in the middle represents the game and the actions that take place within the game. The basic structure was found on lucid chart along with lots of useful information on how to represent use case diagrams (2018). Both players have the same actions. This can be seen by the fact that there is line connecting every action (the use cases) with each actor.



# UML Class Diagrams

The diagrams below represent the entire game with all of its classes as it shows the static state of the application. This allows the reader and people looking at the diagram to get a better understanding of the application as it is more visual and allows for the documentation of different aspects of the application while at the same time imposing on some constraints (UML - Class Diagram, n.d.). After doing some research on the general structure of class diagrams (Bass, n.d.), I was able to draw the class diagrams show below. Due to space constrians, these are not the most accurate as there are some links to the model pacakge from both the view and controller package that are not shown here.





# Design Patters

## **MVC (Model-view-conTRoller)**

MVC is a structural pattern that was used in the game to “separate the modelling of the domain, the presentation, and the actions based on user input into three separate parts (Drumm, 2018-2019).” This allows for the decoupling of logic from the UI which creates a loose coupling between the classes. Therefore, the model classes are a standalone and don’t depend on either the view or controller. However, the view and controller and both dependent on the model which might mean that there might be a some updates required if the model changes.

MVC is used throughout the whole of the project, and each class is stored in a package that is dedicated to those three parts (model, view, controller). This allows for a high cohesion as it allows related classes with similar logic to be grouped together and for each class to have a single well-defined task (math-cs.gordon, 2008). This approach was chosen as it made it easier to support multiple views and allowed for adding new functionality with ease. This meant that the code was more efficient as it would easily allow the re-use of code when needed (Development, 2018).

## **delegation**

The goal of the delegation pattern is to have an alternative to inheritance by adding an instance of a class you want to use to your class. As mention in the book of four, this allows an object to forward (delegate) a request to another object. The delegate then carries out the request on behalf of the original object (Erich Gamma, 21 October 1994). In other words, “delegation is an abstract mechanism which centralizes object (method) behavior (Delegation Pattern, 2013)”.

The delegation pattern was used in the project to delegate the responsibly of changing the controller by assigning a common interface to all the appropriate classes and using that to create the needed classes. It is a bit different from the standard use but still falls under the delegation pattern. The principle is the same where a class is being called which knows what to do. The one calling that class doesn’t need to know what is being done or how (Abhishek Gupta, n.d.).

## **Factory**

The factory pattern is a creational pattern which gives us a way of creating objects. Factory can be used when a project has a superclass (Game object) with many sub-classes, and one of those sub-classes are required to be instantiated. The factory pattern delegates the responsibly of instantiation from the client to the factory class (PANKAJ, n.d.).

This allows the code to be robust and reusable as the client doesn’t need to know about the actual classes that are being created. Therefore, it would be easy to swap out classes without the client knowing. This is beneficial as it creates a high cohesion due to it being a standalone, and loose coupling between the classes due to its independency (Drumm, 2018-2019).

This was very beneficial for this application as the snake, and different types of foods for the snake were all sub-classes of the game object and this was used to create the objects in the factory class. This therefore allowed for an abstraction between the implementation and client classes through inheritance.

## **Strategy**

The strategy pattern is a behavioral pattern that encapsulates a collection of similar algorithms that have a common superclass or interface. In our case, it was used to encapsulate the code that allowed the snakes to move. This allowed the algorithm that is used to change at run time depending on the user input (Drumm, 2018-2019).

When the user changed the direction of the snake, the object delegates the request to an object that represents that specific strategy to carry out the request. This object, along with all the other strategies, are concrete relations of the strategy interface.

The reason I used the strategy pattern was due to the fact that I noticed that the movement of the snake was a family of related algorithms and that the strategy pattern could be used to help decouple the class and make it more independent so that the class using these algorithms doesn’t need to worry about the complex details and let the strategy class do that job (Erich Gamma, 21 October 1994).

# Conclusion

Going through the process of learning design patterns and understanding their importance in building applications has given me a greater insight into the importance of planning and implementing a responsibility driven design. It also showed me the difficulties that comes with using design patterns and how to choose the right one to help solve the problems you’re dealing with. I believe it has given me an amazing experience and strengthened my programming knowledge and software principles such as loose coupling, high cohesion, and encapsulation.

## **improvements**

This educational game was a massive project for me but using design patterns and UML diagrams made it manageable and structured. However, there were many things that I didn’t manage to complete and/or could’ve been improve upon. Some of these are: better use of getters and setters instead of directly accessing the fields, getting rid of the lag/pause in the game, fixing the issue of the snake leaving the grid when it’s two players, and better use of the strategy pattern as I don’t believe I’ve used it as efficiently as I could have. Overall, I believe I was able to demonstrate my understanding of design patterns and responsibility driven design which was the main goal.

# References

Abhishek Gupta, A. (n.d.). *Delegation vs Inheritance in Java*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/delegation-vs-inheritance-java/

*Delegation Pattern*. (2013, 10 04). Retrieved from Best-Practice Software Engineering: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/delegation.html

Development, D. N. (2018, 04 26). *Six Benefits of Using MVC Model for Effective Web Application Development*. Retrieved from brainvire: https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/

Erich Gamma, R. H. (21 October 1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

math-cs.gordon. (2008, 07 24). *CPS211 Lecture: Cohesion and Coupling; the MVC Paradigm*. Retrieved from math-cs.gordon: http://www.math-cs.gordon.edu/courses/cs211/lectures-2009/Cohesion,Coupling,MVC.pdf

McKenzie, C. (2018, 12 01). *Benefits of lambda expressions in Java*. Retrieved from theserverside: https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Benefits-of-lambda-expressions-in-Java-makes-the-move-to-a-newer-JDK-worthwhile

PANKAJ. (n.d.). *Factory Design Pattern in Java*. Retrieved from journaldev: https://www.journaldev.com/1392/factory-design-pattern-in-java

*UML - Class Diagram*. (n.d.). Retrieved from tutorialspoint: https://www.tutorialspoint.com/uml/uml\_class\_diagram.htm