Une image contenant texte

Description générée automatiquement

**Design Patterns and Software Development Process: Final Project**

*By Benoît Cordier & Rodrigue Delgado, IOS1*

# Table of Content

Table des matières

[Table of Content 2](#_Toc61105124)

[Exercise 1 – CustomQueue – Generics 3](#_Toc61105125)

[Introduction 3](#_Toc61105126)

[Design hypotheses 3](#_Toc61105127)

[UML Diagrams 4](#_Toc61105128)

[Class diagram of the solution 4](#_Toc61105129)

[Sequence diagram 4](#_Toc61105130)

[Test Cases 5](#_Toc61105131)

[Exercise 3 – A MonopolyTM game (8 points) – Design patterns 6](#_Toc61105132)

[1/ Introduction 6](#_Toc61105133)

[2/ Design Hypotheses 6](#_Toc61105134)

[3/ UML Diagrams 8](#_Toc61105135)

[Class Diagram of the solution 8](#_Toc61105136)

[Sequence diagram 9](#_Toc61105137)

[Test Cases 9](#_Toc61105138)

[Additional Remarks 10](#_Toc61105139)

# Exercise 1 – CustomQueue – Generics

## Introduction

In this first exercise, we are asked to design and develop a CustomQueue data structure, represented by various linked nodes that carry a value of generic data type. This CustomQueue will work like a classic queue (enqueue and dequeue system) and needs to provide all the needed characteristics so to be used by a foreach loop.

## Design hypotheses

First, let’s talk about nodes. In the queue, every node will need a value, of generic data type. Therefore, nodes need to be represented by a generic public class, “Node<T>”. Moreover, these nodes need to be linked to another node, called “next” which is represents by another instance of “Node<T>”.

Using these hypotheses, we are able to design the “CustomQueue” class. We simply need to provide the front node and the back node to create an instance of the custom queue, as the attribute next will retrieve the different nodes.

Like this, CustomQueue will handle Enqueue and Dequeue methods. Indeed, to enqueue a node, we just need to set the value next of the back node to the new one and set this new node as the new back node. To dequeue a node, we will need to set the front node to null and set the next node of this front to the new front node.

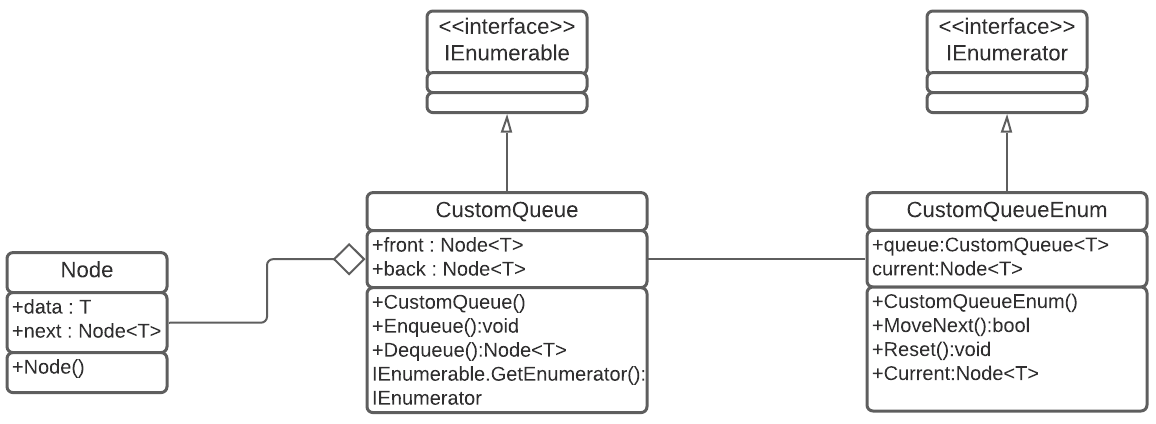
We need to be aware of a few cases. For example, if the queue is empty, the enqueue node becomes the front and the back at the same time. Then, when dequeuing the last node (when it didn’t have a next node), we need to also set the back to null.

Finally, we are asked to implement all the needed characteristics so to be used by a C# foreach loop. To do so, we will create a generic class, “CustomQueueuEnum<T>”, that will implement the interface “IEnumerator” in order to iterate through a queue to use the foreach, analyzing successively each node, called Current node.

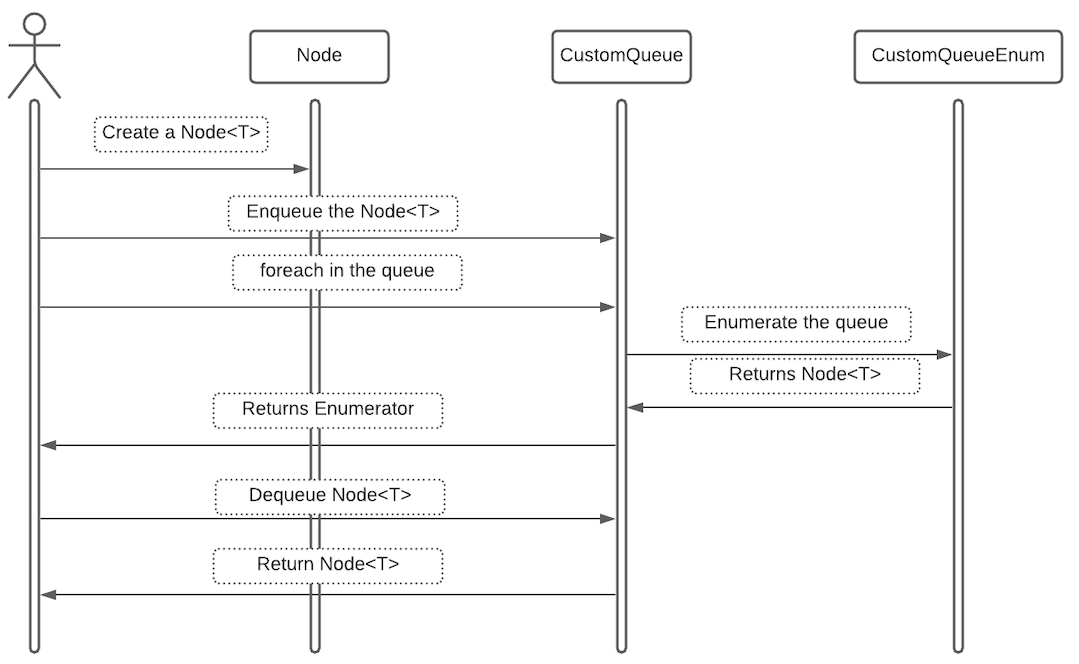
This class handle the IEnumerator.MoveNext(), which will set the Current to the next node of the Current one when enumerating the queue, and the IEnumerator.MoveNext(), which set the Current to null to reset the enumeration. IEnumerator will finally returns the Current node, and the interface IEnumerable will be implemented by CustomQueue to allow access to “CustomQueueEnum” and return the enumeration.

## UML Diagrams

### Class diagram of the solution



### Sequence diagram



## Test Cases

Once the hypotheses are implemented, we can test the CustomQueue, by enqueueing and dequeuing variables of generic data types.

For example, with Strings:

Une image contenant texte

Description générée automatiquement 

Or with Integers:

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Or as a last example, with a List<String>

Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement

# Exercise 3 – A MonopolyTM game (8 points) – Design patterns

## 1/ Introduction

In this last exercise, we are asked to simulate a simplified version of the Monopoly game, in other words, a Monopoly game without the sale-resale part, but only movements and the jail system. Players roll two dices to go through a board composed of 40 positions, linked (with position 39 linked to 0). Every turn, a player rolls the dice, moves, and an event happen, depending on the position.

In our code, the Board is represented by the class called “Plateau”, which is defined by a set of 40 positions. Those positions are represented through the class “BuyCase”, “JailCase” and “GoToJailCase”, which inherits from the abstract class Case (this will be detailed in the next part of the report, in the Strategy Pattern part). Players are represented with the class “Player”.

We manage to implement this game by using the **Singleton Pattern, the Strategy Pattern and the State Pattern**.

## 2/ Design Hypotheses

During this exercise, we faced a few difficulties. For instance, the fact that every position doesn’t have the same properties, or that a player’s turn depends sometimes on his previous turns (to go out of jail).

Firstly, the first issue was about the creation of the board. We wanted to make sure that the board can be created only once. To do so, we used the **Singleton Pattern**.

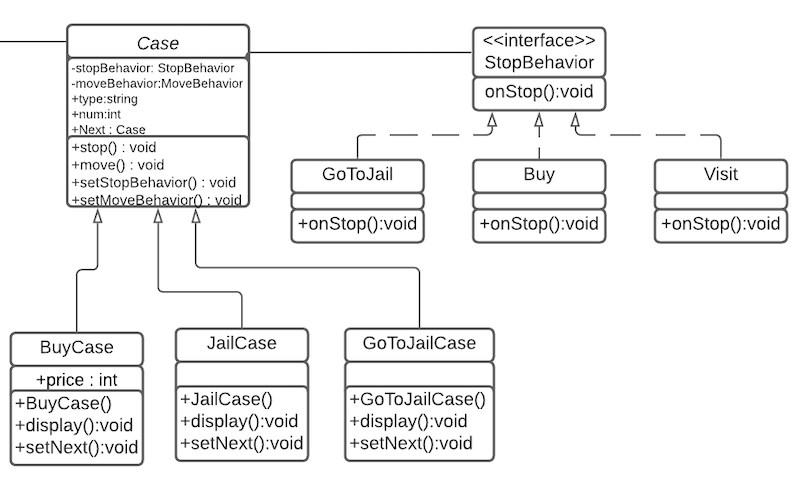
Une image contenant texte

Description générée automatiquementOur class “Plateau” will therefore need a private constructor (or a sealed class), and a private variable (“instance”) which will be used to check if an instance has already been created. Moreover, we decided to make it “**Thread Safe**”, using a lock to make sure two threads don’t create two boards.

Secondly, concerning the implementation of this board, the main issue was to make it circular, which means the board loops. To overcome this, the class “Case”, representing position, has an **attribute Next, which will point to another instance of “Case”**. On the board, position 0 will point to 1, then 1 to 2, … and finally 39 to 0. To implement this board, we have hesitated to use the **Prototype Design Pattern**, to clone every position around the board. But it didn’t seem relevant as we wanted to add a few points compared to the exercise, like a **price and a numbe**r for each position!

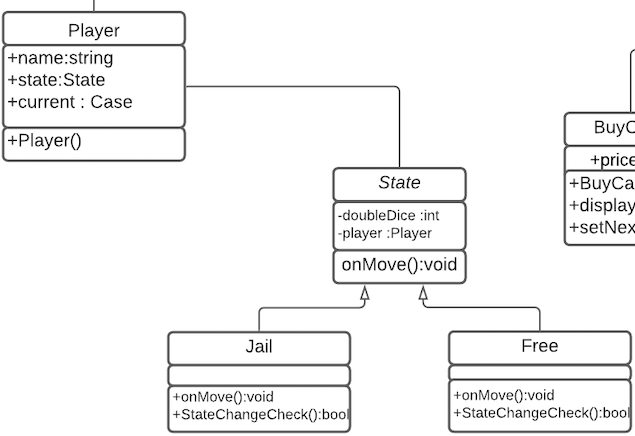
Then, we faced the main difficulty of this exercise: how to implement every position separately, knowing that positions don’t have the same effect, while they are instance of the same class? We struggle with this a while, but we realize that positions have two kinds of effect: when a player **stops** on the position, and when he **moves** from the position. These effects are what makes positions different, and therefore we decided to use the **Strategy Pattern** for the stop behavior, and a **State Pattern** for the move behavior, as it depends on the state of the player (detailed later).

In this strategy pattern, behaviors that may vary across positions are the StopBehavior and the MoveBehavior (these are our interfaces). Then, a **set of classes** will design our different behaviors. For example, StopBehavior will be inherited by classes “Buy” (when a player lands on the position he can buy it), “Visit” (nothing happens, for position 0 and 10), and “GoToJail” (when a player lands on it, he goes to jail). Using this, the class “Case” will delegate his StopBehavior and MoveBehavior instead of creating multiples methods.



Moreover, we will be able to create other behaviors if we want to implement the complete game!

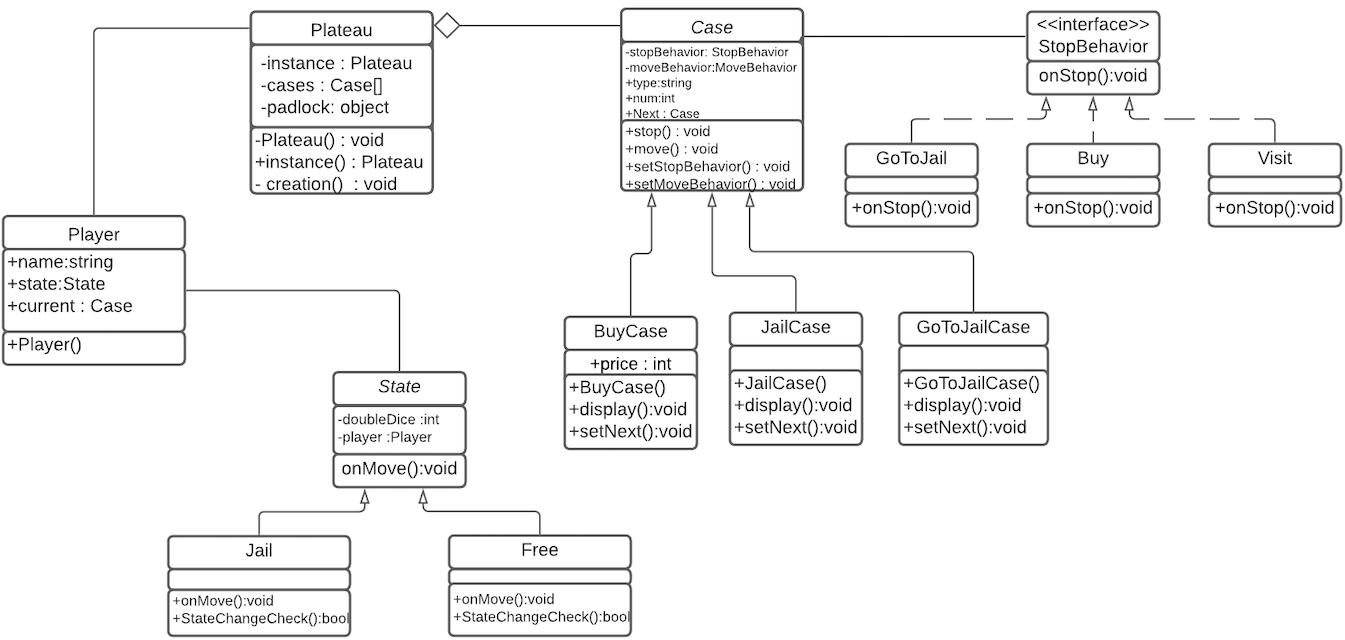
Concerning the move behavior, as we said previously, it depends on the state of the player. To implement this, we need to create two states that will define the player: a free state, and a jail state. Then the move behavior will depend on the state of the player: if he is free, then he moves normally, if he is in jail, he needs to do a double. Every double increments an integer and will check the state of the player, and change it if needed, every 3 doubles.



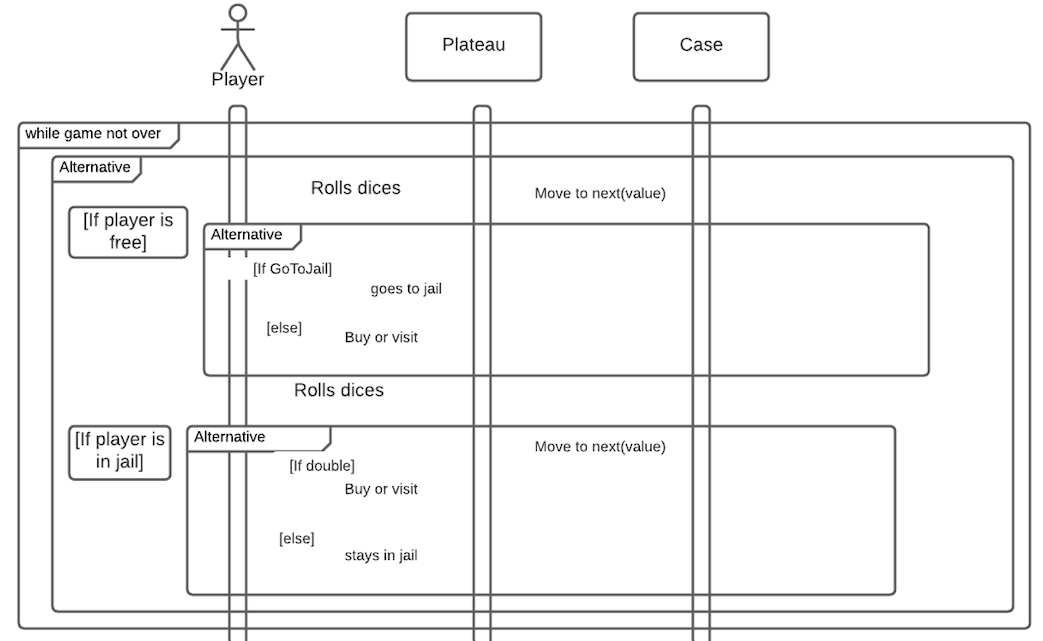
Finally, concerning the global workflow of the app, each player rolls the dice depending on his state, moves or not, and finally stops on a position to trigger a behavior

## 3/ UML Diagrams

### Class Diagram of the solution



### Sequence diagram



## Test Cases

Once we have implemented our patterns, we were able to test the game.

The board is created, circular, and unique thanks to the Singleton Pattern.

When a player lands on a position, it triggers an event depending on the position, which occurs thanks to the Strategy Pattern.

Once a player is in jail, he has to do a double to be free, which occurs thanks to the State Pattern.

Une image contenant texte

Description générée automatiquementWhen a player lands on a buy position, he can buy it with a price:

When a player lands on the “Go To Jail” position, he directly goes to jail and goes out if he does a double, or after 3 turns:

Une image contenant texte

Description générée automatiquement

When a player does a double, he plays again! But if he does three double in a row, he goes

to jail.

Une image contenant texte

Description générée automatiquement

To sum up, we indeed obtain the expected results.

## Additional Remarks

We decided for every position (except the go to jail, the visit, and the start positions) to add the possibility to buy it, with an ascending price. Every position is 2000$ more expensive than the previous one. Players can buy them, but it doesn’t have any impact on the rest of the game, it was just to add a bit of context of the game and show that we can still improve the game in order to make a full one!