# Patter B – State

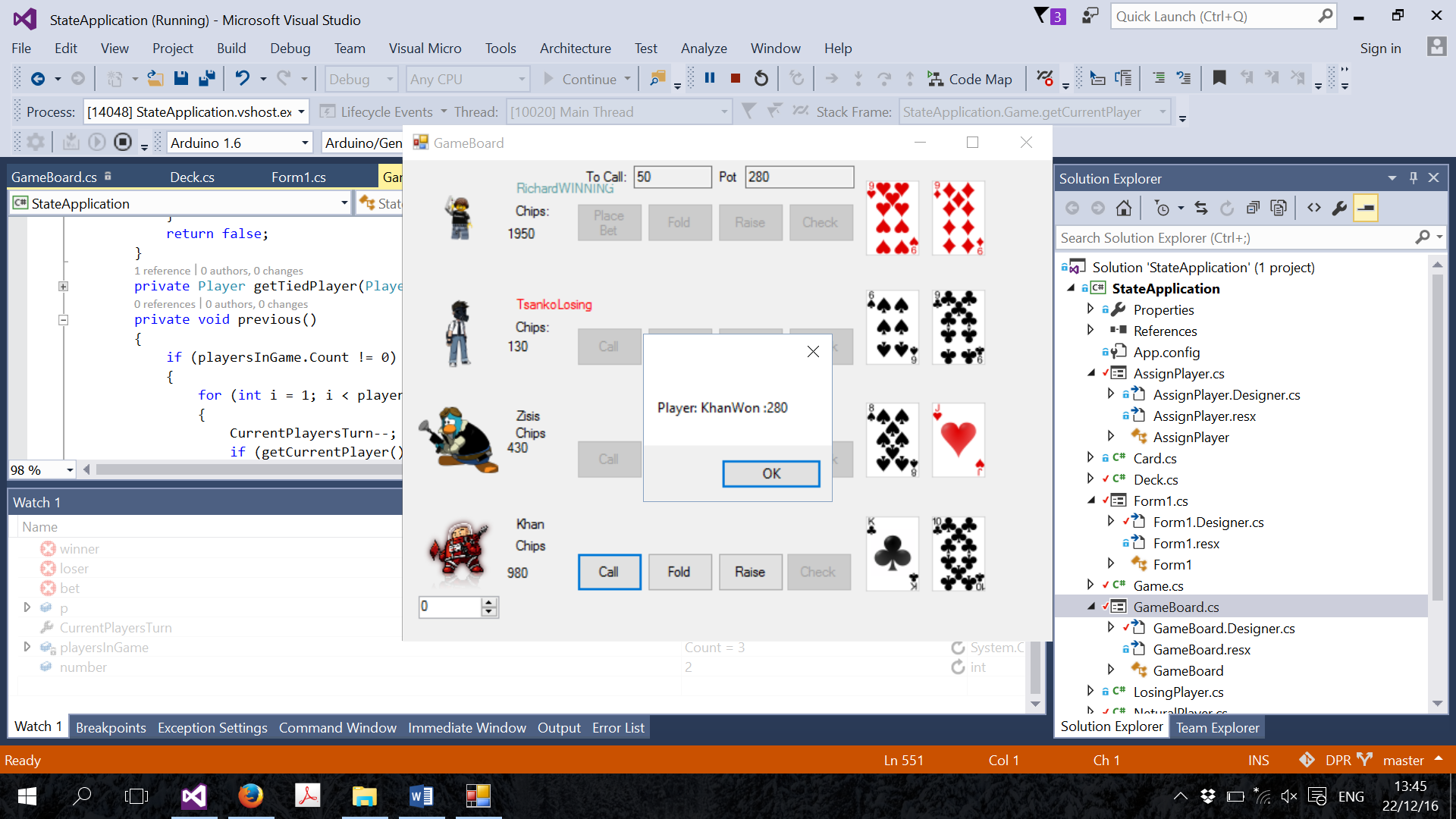
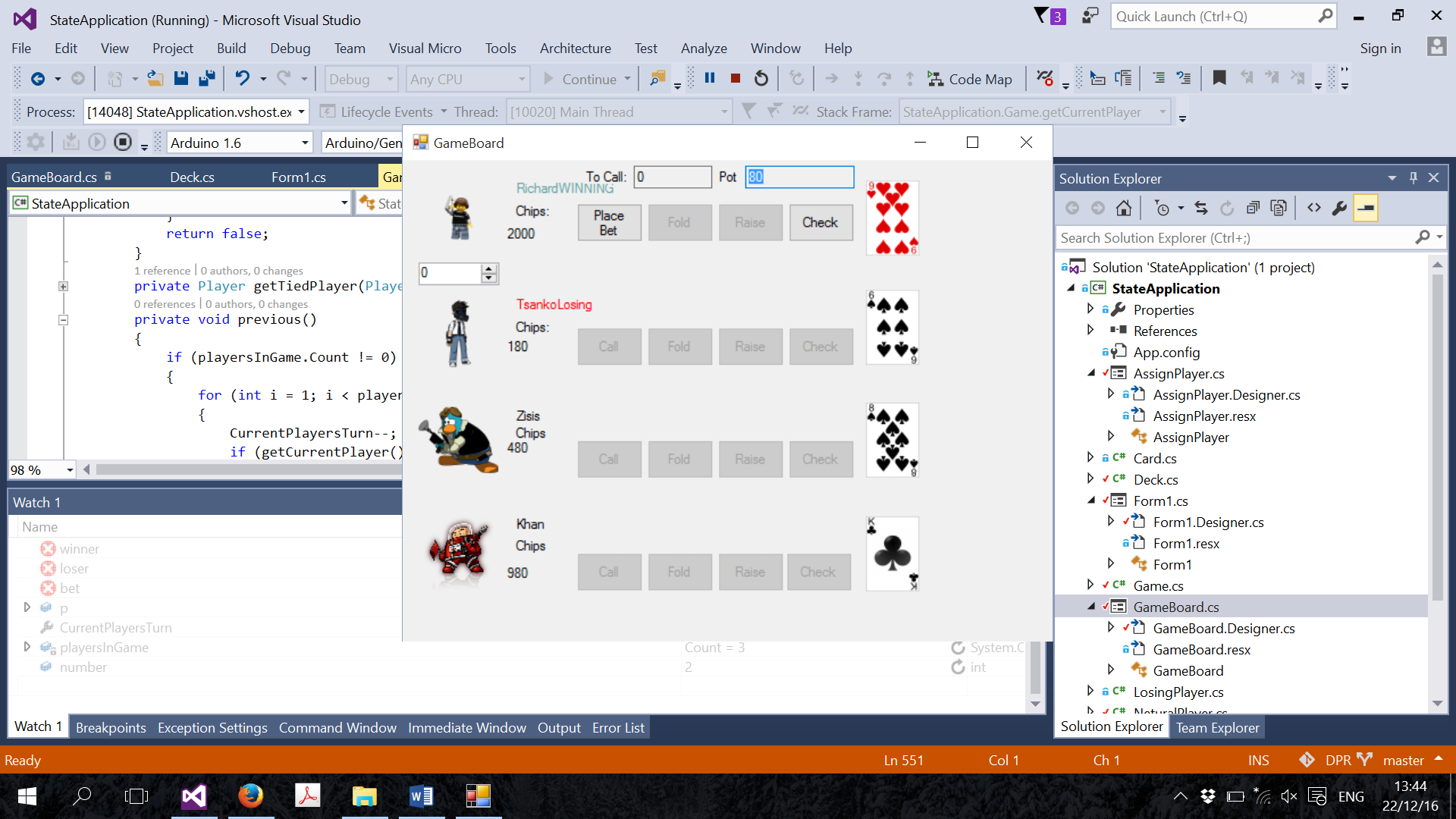
## Introduction

The state design pattern allows an object to change its state during run time. Meaning that it will behave differently based on the state it is in. Each state is derived by a state interface and is implemented separately. In a state pattern each state knows of the ‘wrapper’ class which is in our case the player. There are two ways to implement the State Design Pattern both of which reply on the Wrapper class to have a State object that can then change to its inherited classes during run time. We have however chosen to use the other way of implementing the State Design Patter by having the Player class contain all its possible states including it current state. When the Player class changes states its current state will change to one of those states belonging to the player class. We implemented it this way because there are only three states and each state takes the player itself as an argument in order to use the Players chips.

We created a small card game to demonstrate the use of states.

## Card Game Example

In our card game the state design is used to change the players state based on there ‘Winning’, ‘Losing’ or ‘Neutral’ status in the game. A player in the game has a limited set actions he can undertake, Call, Check, Raise, Fold. However, based on the Players state the behavior of these actions change. The challenge for the players in this game is that when a player folds, raises or checks they must play a fee/tax to the losing player. The winning player must pay an even high tax or fee to the losing player. However in the beginning of a new round the player with a winning state receives bonus chips. These behaviors are constantly changed during run time. The state design pattern was very much appropriate for this game as there are not too many states to change to which can make the state pattern a bit cluttered.



## Card Game State Class Diagram



## The Advantages and Disadvantages of the State Design Pattern

+ Object can dynamically change its behavior

+ Very efficient

+ Reduces duplication

+ Easy to extend certain behaviors

* Potential class explosion
* Better used when there are not too many state options
* Sometime better to use events.

## Relation to other Patterns

The State pattern shares similarities with the Strategy pattern. However they are used for different reasons. While they are both behavioral design patterns the behavior are both changed during run time. Concerning the Strategy pattern the behavior is selected during runtime where as in the State Pattern its state and thus entire way of acting can be changed to whatever circumstances

## Conclusion

In the right circumstances of course, the State Pattern is a preferable design pattern. If there are not a large amount of states then implementation is not so strenuous. Switch states and thus the behavior is consequently quite efficient. If there are a large amount of states then I would consider or recommend using another deign pattern as it the Pattern would encourage an unnecessary over use of similar classes.