[STATE PATTERN]

Pizza Shop Application

Abstract

This document continues the implementation of design patterns. This document presents the state pattern. We have decided to implement our design patterns based on an application theme. In this current assignment we decided to show how a state pattern can be implemented to a pizza shop.

Nurjahan Akter & Alexandros Gkalkin

Introduction

Before we start with the state design pattern, we should explain the methodology in applying the stated design pattern. For the last couple of weeks, we have been making themed application for most of the design patterns. Our theme is making a pizza shop application. In this way we learn to apply different design pattern on the same kind of application. We feel that in this way it is easier and clearer to understand a patterns, since we are not obstructed by the vast amount of possibilities for implementation purposes. Additionally, we get to see the actual differences of the design patterns, since we no longer need to study different themes every time.

Finally, our application shows the process of ordering a pizza from a shop with a tracking capability. In other words, we can track the progress from the ordering part till the final delivery.

State Pattern

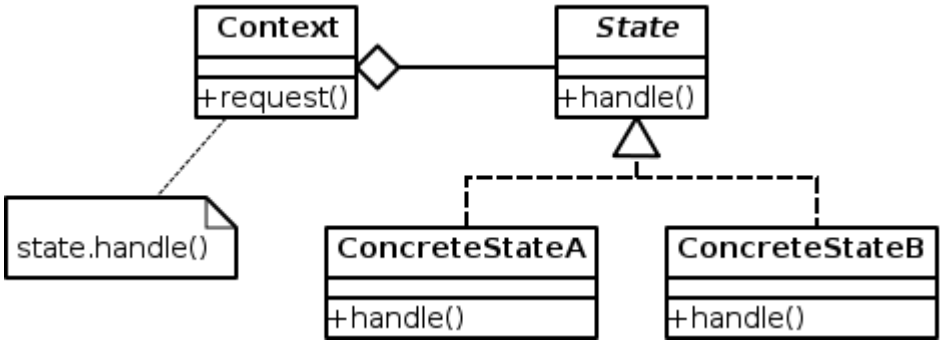

Figure 1 - State Pattern

The state pattern resembles the state machine methodology used in computer programs and logic circuits. However, the state pattern implements this logic in an object oriented way. Usually state machines are used in an embedded programming environment where you have a microcontroller or a board or even a computer working in ways of multiple states where the machine can remain in a specific state for a very long time. For example, a state machine of an elevator can contain such states as moving, waiting, which basically show the two functionalities of an elevator: to move between floors and wait till used. We can additionally expand these into such states as waiting for a button press, service mode etc. In our case however as you will see in the UML diagram, we have separated into 4 different stages that one can go through when ordering a pizza. These states are:

- Product gathering which is the state when the chef gathers the products that he needs for every pizza.
- Cooking, which is the state where the products are formed into a raw pizza and baked.
- Packing, which is the state where the pizza is taken out of the oven and placed into a cardboard box.
- And finally, Delivering, which is the state in which a pizza arrives from the shop to your doorway.

Now that we know, what to expect we are going to discuss a bit about such important values as reusability, maintainability and extensibility.

Reusability

One of the advantages of design patterns is that we could reuse them in different applications. As far as the state pattern is concerned, every behavior can be reused independently from the others and in collection with the others. However, the main problem occurs that most of the behaviors are linked to a specific object that actually has them. Then we would say that the reusability of this pattern would be medium.

Extensibility

Design patterns such as state can be extended only if we use a correct design. In general, most of the state classes in our application could be extended or even broken down in smaller substrates. For example, the product gathering state could be extended into such states as gathering materials, making dough, placing materials on dough, etc. Furthermore, we can also include additional states such as payment. In our application we are only limited by the amount of work that needs to be done to produce a pizza. Taking as an additional example, if we apply the state pattern to a car factory then again we are going to be limited only by the amount of the actual states a factory has for creating a car.

Maintainability

In the pizza application, we do not use any if statements in the design pattern implementation. However, the main problem is the fact that every behavior of the pattern implements its own way for specific functions. If we see for example, in the productGathering state of the pizza application we get implementation for all of the function described in the interface. However during this state, the only function that makes sense to use if the gathering(). All the other functions are implemented as a case when someone tries to use an inapplicable to the state function. Therefore, in our application in case of maintenance it would be extremely hard to achieve that because every separate function can be implemented differently depending on the state it is in.

UML

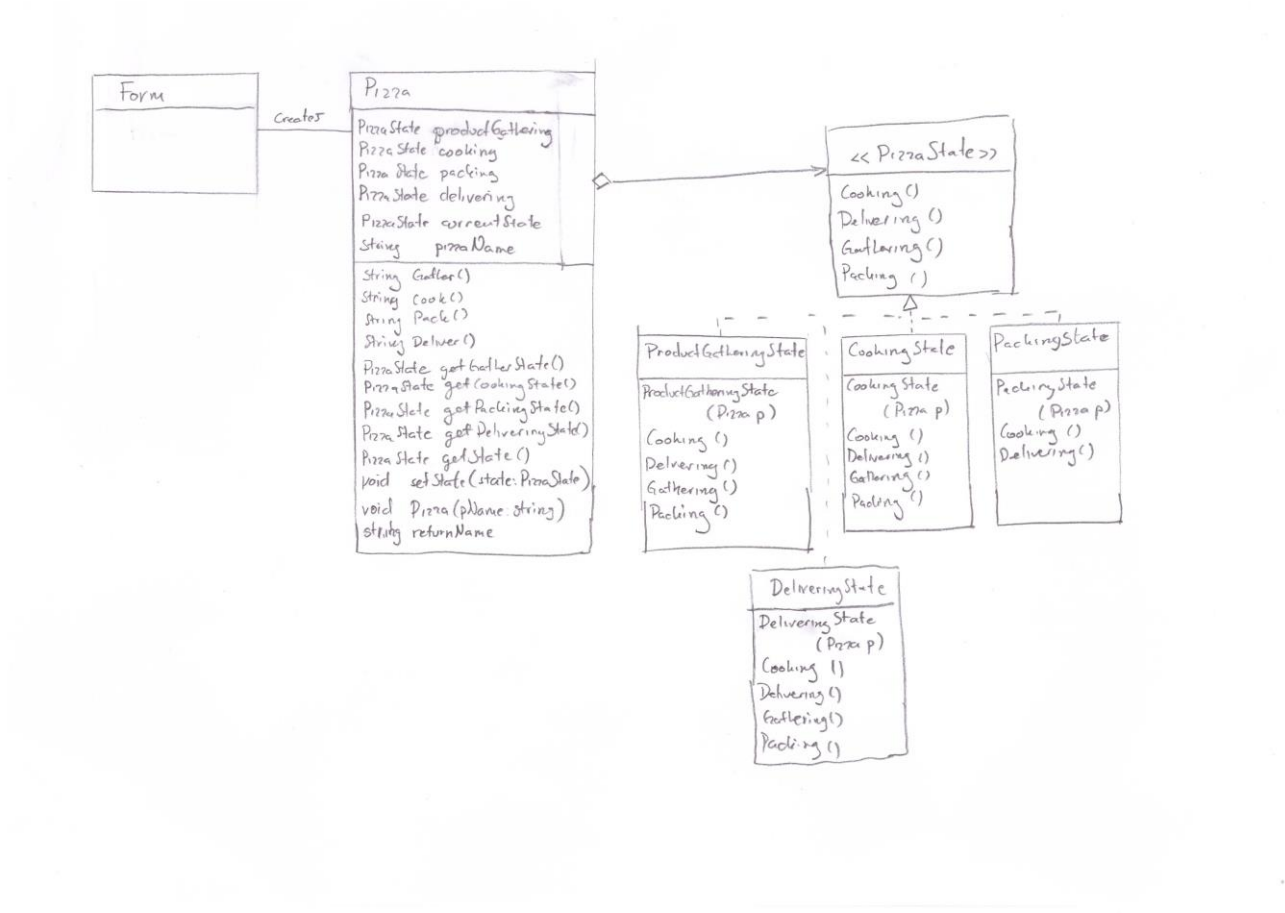

Figure 2 - Pizza Shop UML

Test

We have created a number of test for the pizza application. All of them test the correctness of the transition from one state to the other. However, since the object created by the unit test does not maintain the application's flow of events, we had to manually alter the states per test.

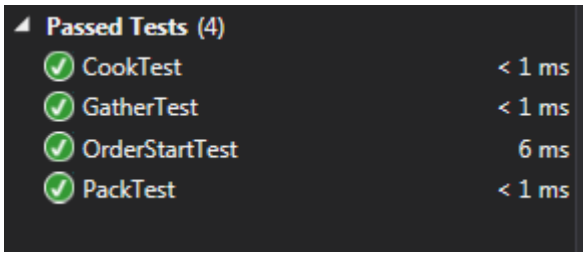

Figure 3 - Test results

Video: https://youtu.be/3ypVb9xo12s (sorry for the mic quality)