|  |
| --- |
| Fontys ICT |
| Decorator pattern |
| Design patterns |

|  |
| --- |
| Jan-Niklas Schneider, Georgiana Manolache  9-20-2016 |

Contents

[1 Introduction 3](#_Toc462176976)

[2 Decorator pattern 3](#_Toc462176977)

[3 Implementation 4](#_Toc462176978)

[3.1 Explanation of classes 4](#_Toc462176979)

[3.2 Features 5](#_Toc462176980)

[4 Design choices 5](#_Toc462176981)

[5 Graphical User Interface 6](#_Toc462176982)

[6 Unit tests 7](#_Toc462176983)

[7 References 7](#_Toc462176984)

# Introduction

The goal of this document is to give an overview of the decorator pattern by giving an example implementation which displays a simplified pizza delivery service called *PizzaMe*. Furthermore, reusability, extensibility, and maintainability of this pattern are elaborated. Also, the implementation, its unit test and graphical user interface (GUI) are reviewed.

# Decorator pattern

The decorator pattern is a software design pattern which allows an object’s functionality to be extended during runtime. This allows the object to be changed when it’s being used compared to inheritance where functionality can be changed only statically, during compile time. Hence, the main goal of this pattern is to add additional responsibilities to an object dynamically.

The figure below depicts an UML diagram of the decorator pattern. The pattern consists of four classes.

Firstly, A *Component* interface which objects implement that can have responsibilities to them dynamically.

Secondly, A *ConcreteComponent* that represents a derived object of the *Component* interface that can add responsibilities dynamically.

Third, the *Decorator* holds a reference to the *Component* and implements its interface.

Lastly, the *ConcreteDecorator* which adds a certain behavior.

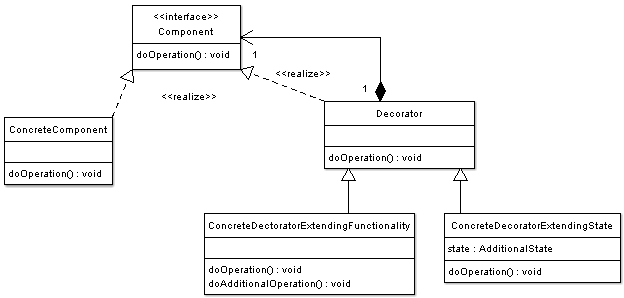


Figure 2‑1: UML diagram of decorator pattern (OODesign, 2016)

# Implementation

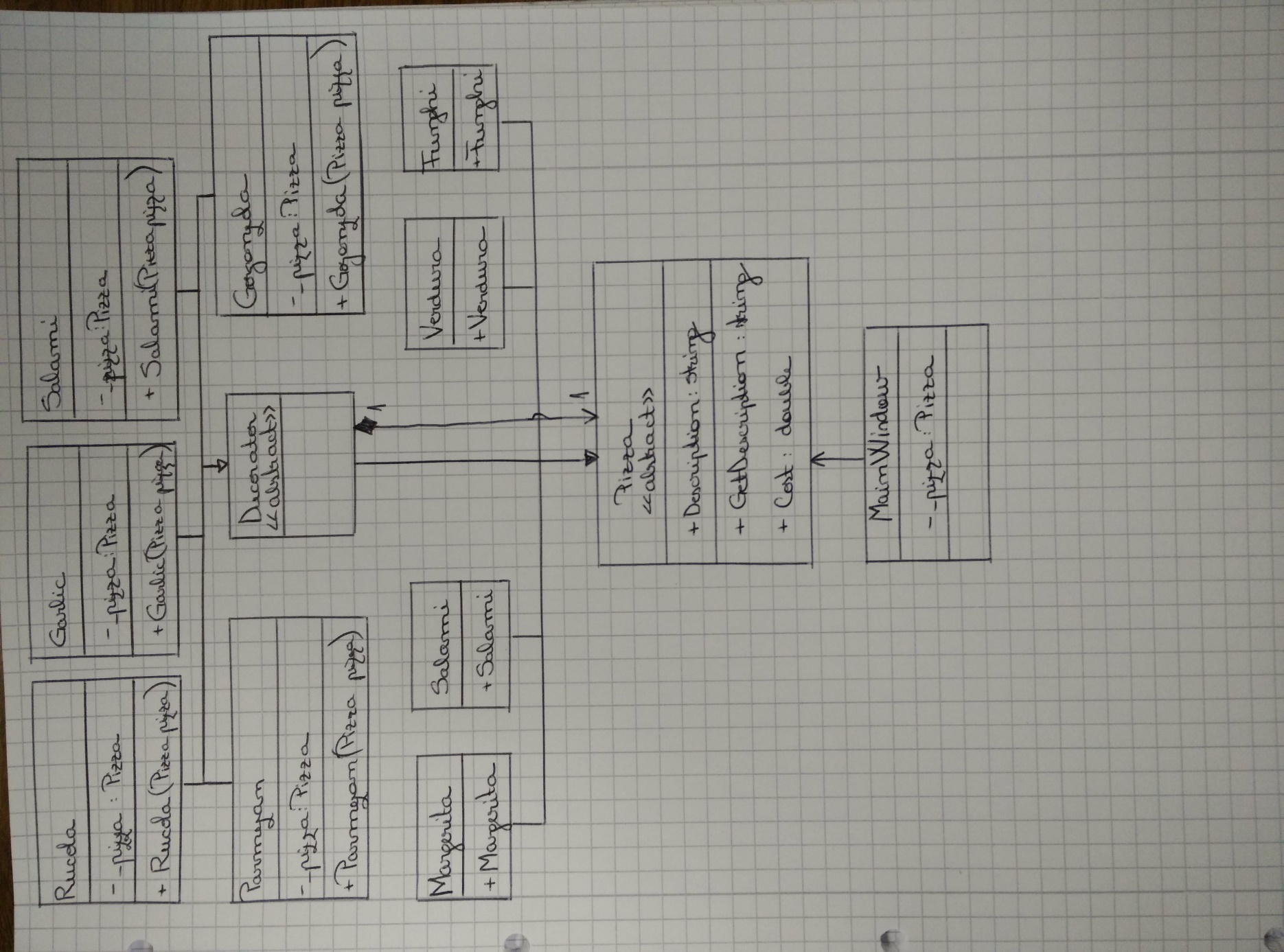
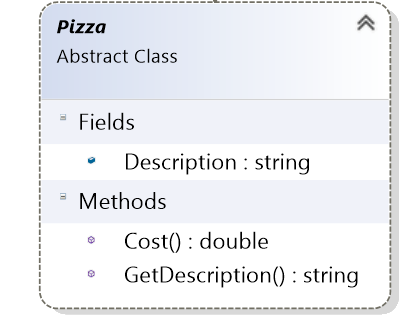
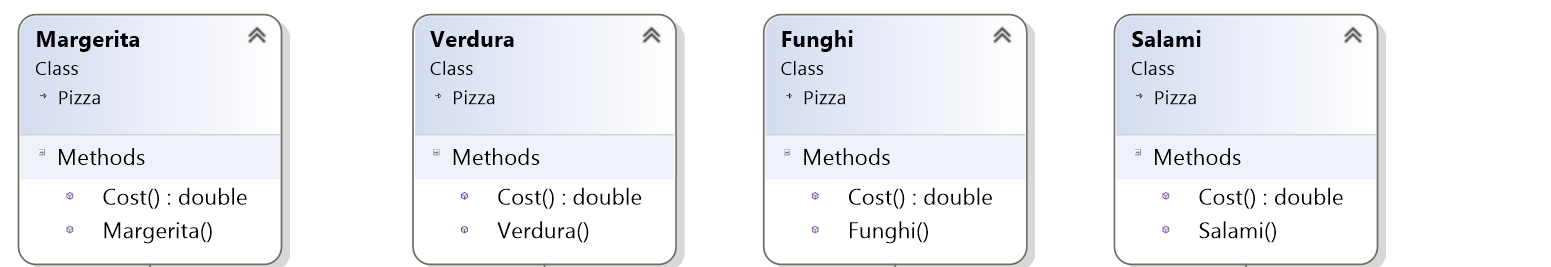


Figure 3‑1: Class diagram of Decorator pattern

## Explanation of classes

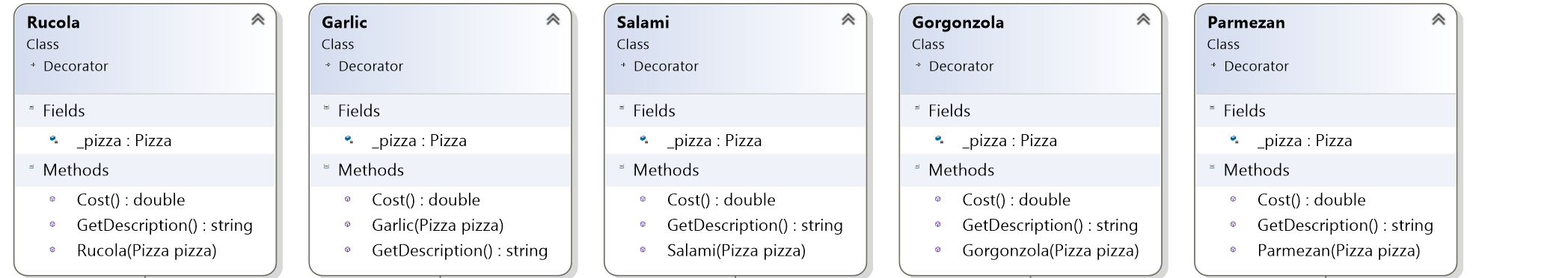
1. **Pizza** is an abstract class with two methods *Cost()* and *GetDescription()*. Attribute *Description* holds information about the name of the pizza and is set in the extended classes (2.,4.).



1. Four concrete components implement abstract class **Pizza**: **Margerita**, **Verdura**, **Funghi**, **Samali**. For each concrete class which extends **Pizza**, the cost is specified. The description of each pizza is set in the constructor of each concrete class. 
2. **Decorator** is an abstract class which implements **Pizza**, to ensure interchangeability.



1. Five concrete components implement abstract class **Decorator**: **Rucola**, **Garlic**, **Salami**, **Gorgonzola**, **Salami**, **Parmezan**. These classes hold an instance of the main abstract class **Pizza** which is initialized in the constructor of each concrete component.



## Features

The application has a simple and straightforward user interface. User can select its pizza of choice. Furthermore, user can any ‘extra’ condiment to its pizza of choice.

# Design choices

The implementation of the assignment has been done with regard to reusability, extensibility, and maintainability.

The **reusability** of the decorator pattern is rather low. This caused by the fact that pattern aims at extending the responsibility of an object during run-time. Depending on application of the pattern the interface can differ vastly.

In terms of **maintainability** the pattern is easy to maintain since logic is separated into independent units, such as *ConcreteComponents* and *ConcreteDecorators.*

The decorator pattern excels at **extensibility**. Functionality can be added by creating a new implementation of the *Component* interface or *Decorator* class which allows effortless integration of new functionality.

Since reusability of this pattern of this pattern is low the implementation of the *Component* interface has been done using an abstract class *Pizza*. By making use of an abstract class redundant and repetitive code is reduced.

# Graphical User Interface

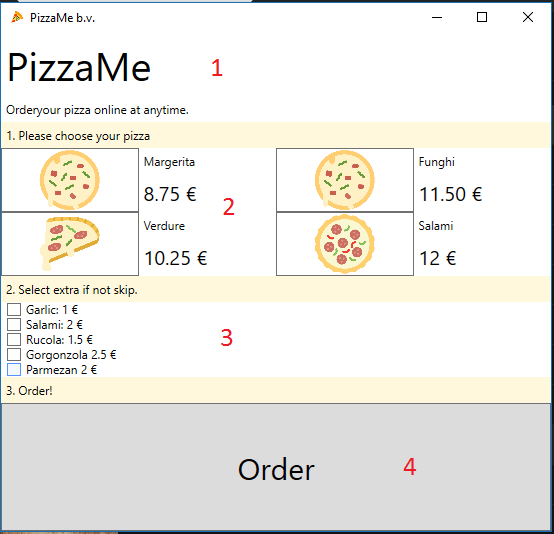


Figure 5‑1: Graphical user interface

The figure above depicts the user interface where red numbers indicate functionality or controls. More precisely these are:

1. Title of the delivery service
2. Choose from one of the pizzas using pizza specific icon. Right of the icon the description and price is shown.
3. None or many extras can be selected for the selected pizza.
4. Order the pizza when pizza is selected and extras are choosen.

# Unit tests

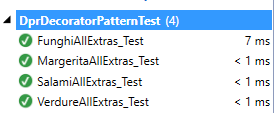
For each implemented Component unit tests have been defined to assert correct behavior. Four tests can be found in *DprDecoratorPatternTest*, namely *FunghiAllExtras\_Test, MargeritaAllExtras\_Test, SalamiAllExtras\_Test,* and *VerdureAllExtras\_Test*. Consequently, all test ran successfully.

Figure 6‑1: Unit test of DprDecoratorPattern

# References

OODesign. (September 2016). *Decorator Pattern*. Von OODesign.com: http://www.oodesign.com/decorator-pattern.html abgerufen

SourceMaking. (September 2016). *Decorator*. Von SourceMaking.com: https://sourcemaking.com/design\_patterns/decorator abgerufen