

COURSE WORK REPORT

EDIFU-23

STEPANENKO HLIB

ARTHUR BO

Object-Oriented Programming in Food Delivery Web Application

Introduction

What is your application?

Our application is a partial clone of Bolt Food, but it is written for one specific restaurant. That is, it can be used to deliver goods from one specific location.

It allows you to select products from a list, upload your location data there and order the product. This application allows you to create several product categories, upload there: price, product subtype, photograph and description.

A user who wants to buy this product will be able to select it from the list and buy them by entering his contact information.

Body/Analysis

OBJECT-ORIENTED PROGRAMMING PRINCIPLES

1.Polymorphism - Polymorphism in object-oriented programming allows objects of different types to be treated as objects of a common superclass. This means that a single interface or method can behave differently based on the specific implementation or type of the object it is called upon. This flexibility is achieved through concepts like method overriding, where a subclass provides its own implementation of a method defined in its superclass, enabling objects of the subclass to be treated interchangeably with objects of the superclass. Polymorphism promotes code reusability, flexibility, and easier maintenance by allowing code to operate on objects at a higher level of abstraction.

2.Abstraction - Abstraction simplifies complex systems by focusing on essential aspects while hiding unnecessary details. In object-oriented programming, it's achieved through classes and objects, where classes define properties and behaviors, and objects represent instances of those definitions. Encapsulation, which bundles data and methods within a class, helps in hiding implementation details. Abstraction helps manage complexity, allowing developers to focus on solving higher-level problems and building scalable, maintainable, and reusable software components.

3.Inheritance - In object-oriented programming, inheritance allows new classes (subclasses) to inherit attributes and methods from existing classes (superclasses). This promotes code reuse and creates a hierarchical relationship among classes. Subclasses can extend or modify the behavior of superclasses, facilitating the creation of modular and extensible software systems. Inheritance helps build conceptual clarity and promotes maintainability by organizing classes into a hierarchical structure.


4.Encapsulation - Encapsulation in object-oriented programming involves bundling data and methods within a class and controlling access to them. It hides implementation details, promoting modularity and information hiding. Encapsulation ensures data integrity and enhances code organization, leading to more robust and maintainable software.

ABSTRACTION:




```
1  #Abstraction
2  def post(self, request, *args, **kwargs):
3      name = request.POST.get('name')
4      email = request.POST.get('email')
5      street = request.POST.get('street')
6      city = request.POST.get('city')
7      state = request.POST.get('state')
8      zip_code = request.POST.get('zip_code')
9      order_items = {
10         'items': []
11     }
12
```

ENCAPSULATION:



```
1  #Incupsulation
2      def __str__(self):
3          return self.name
4
```

POLYMORPHISM:



```
1  #Polymorhism
2      def get(self, request, *args, **kwargs):
3          # get every item from each category
4          #Reading from
5          appetizers = MenuItem.objects.filter(category__name__contains='Appetizer')
6          entres = MenuItem.objects.filter(category__name__contains='Entre')
7          desserts = MenuItem.objects.filter(category__name__contains='Dessert')
8          drinks = MenuItem.objects.filter(category__name__contains='Drink')
9          soups = MenuItem.objects.filter(category__name__contains='Soup')
10         mainmeals = MenuItem.objects.filter(category__name__contains='mainmeal')
11
```

INHERITANCE:



```
1 class Category(models.Model):  
2     name = models.CharField(max_length=100)  
3
```

PATTERNS:

1.DECORATOR - The decorator pattern is a design pattern in object-oriented programming that allows behavior to be added to individual objects dynamically, without affecting other objects from the same class. It involves four main components: the Component interface, Concrete Component, Decorator abstract class, and Concrete Decorators. This pattern enables the flexible addition of responsibilities to objects at runtime, promoting code reuse and separation of concerns.

```
1  from django.contrib import admin
2  from .models import MenuItem, Category, OrderModel
3
4  @admin.register(MenuItem)
5  class MenuItemAdmin(admin.ModelAdmin):
6      list_display = ('name', 'price',)
7      search_fields = ('name', 'category__name')
8      list_filter = ('category',)
9
10
11 @admin.register(Category)
12 class CategoryAdmin(admin.ModelAdmin):
13     list_display = ('name',)
14     search_fields = ('name',)
15
16 @admin.register(OrderModel)
17 class OrderModelAdmin(admin.ModelAdmin):
18     list_display = ('created_on', 'price', 'name', 'email')
19     search_fields = ('name', 'email', 'items__name')
20     list_filter = ('items__name',)
```

2. ABSTRACT FACTORY- The abstract factory pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows the creation of objects to be abstracted away from the client code, providing a way to create families of related objects without exposing the concrete implementation details.

FACTORIES.PY:

```
1  from .models import MenuItem, Category, OrderModel
2  from abc import ABC, abstractmethod
3  #Abstract_method
4  class AbstractFactory(ABC):
5      @abstractmethod
6      def create_menu_item(self, **kwargs):
7          pass
8
9      @abstractmethod
10     def create_category(self, **kwargs):
11         pass
12
13     @abstractmethod
14     def create_order(self, **kwargs):
15         pass
16
17
18  class ConcreteFactory:
19     def create_menu_item(self, **kwargs):
20         return MenuItem.objects.create(**kwargs)
21
22     def create_category(self, **kwargs):
23         return Category.objects.create(**kwargs)
24
25     def create_order(self, **kwargs):
26         return OrderModel.objects.create(**kwargs)
```

USING OF FACTORY ABSTRACT PATTERN IN VIEWS.PY:

```
1 class Order(View):
2     def __init__(self, **kwargs):
3         super().__init__(**kwargs)
4         self.factory = ConcreteFactory()
5
```

READING FROM FILE AND WRITING TO FILE

In our program we read and write files from and to data base “db.sqlite3”. We use admin panel for basic CRUD operations.

WRITING:

```
1 #Writing
2     order = self.factory.create_order(
3         price=price,
4         name=name,
5         email=email,
6         street=street,
7         city=city,
8         state=state,
9         zip_code=zip_code
10    )
11    order.items.add(*item_ids)
12
```

READING:

```
1 #Reading from
2 appetizers = MenuItem.objects.filter(category__name__contains='Appetizer')
3 entres = MenuItem.objects.filter(category__name__contains='Entre')
4 desserts = MenuItem.objects.filter(category__name__contains='Dessert')
5 drinks = MenuItem.objects.filter(category__name__contains='Drink')
6 soups = MenuItem.objects.filter(category__name__contains='Soup')
7 mainmeals = MenuItem.objects.filter(category__name__contains='mainmeal')
8
```

UNIT TESTINGS

File with unit testings:

```
1 from django.test import TestCase
2 from .models import MenuItem, Category, OrderModel
3
4 class MenuItemModelTestCase(TestCase):
5     def setUp(self):
6         self.menu_item = MenuItem.objects.create(
7             name='Test Item',
8             description='Test description',
9             image='menu_images/test_image.jpg',
10            price=10.99
11        )
12        self.category = Category.objects.create(name='Test Category')
13
14    def test_menu_item_creation(self):
15        """Test the creation of a MenuItem instance"""
16        self.assertEqual(self.menu_item.name, 'Test Item')
17        self.assertEqual(self.menu_item.description, 'Test description')
18        self.assertEqual(self.menu_item.price, 10.99)
19        self.assertEqual(self.menu_item.category.count(), 0)
20
21    def test_category_association(self):
22        """Test associating a MenuItem with a Category"""
23        self.menu_item.category.add(self.category)
24        self.assertEqual(self.menu_item.category.count(), 1)
25        self.assertEqual(self.menu_item.category.first(), self.category)
26
```



```
1 class OrderModelTestCase(TestCase):
2     def setUp(self):
3         self.order = OrderModel.objects.create(
4             price=25.50,
5             name='Test Order',
6             email='test@example.com',
7             street='123 Test St',
8             city='Test City',
9             state='Test State',
10            zip_code=12345
11        )
12        self.menu_item1 = MenuItem.objects.create(
13            name='Item 1',
14            description='Description 1',
15            image='menu_images/item1.jpg',
16            price=10.00
17        )
18        self.menu_item2 = MenuItem.objects.create(
19            name='Item 2',
20            description='Description 2',
21            image='menu_images/item2.jpg',
22            price=15.50
23        )
24
25    def test_order_creation(self):
26        """Test the creation of an OrderModel instance"""
27        self.assertEqual(self.order.price, 25.50)
28        self.assertEqual(self.order.name, 'Test Order')
29        self.assertEqual(self.order.email, 'test@example.com')
30        self.assertEqual(self.order.street, '123 Test St')
31        self.assertEqual(self.order.city, 'Test City')
32        self.assertEqual(self.order.state, 'Test State')
33        self.assertEqual(self.order.zip_code, 12345)
34        self.assertEqual(self.order.items.count(), 0)
35
36    def test_order_item_association(self):
37        """Test associating MenuItem(s) with an OrderModel"""
38        self.order.items.add(self.menu_item1, self.menu_item2)
39        self.assertEqual(self.order.items.count(), 2)
40        self.assertIn(self.menu_item1, self.order.items.all())
41        self.assertIn(self.menu_item2, self.order.items.all())
```

TESTING RESULTS

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
-----
Ran 4 tests in 0.032s

OK
Destroying test database for alias 'default'...
```

RESULTS

Our program is written completely. This program allows you to completely complete an order for the buyer, as well as download all the necessary data for the user. In the future, our plans are to add functions so that you can track order statistics, restaurant information, as well as a payment system .

CONCLUSIONS

Summing up our results, we have made a workable program and achieved our initial goals. Writing this coursework improved our knowledge of Python and how OOP works. We also learned to work better with the Django framework, which will allow us to make projects on it in the future. This coursework also helped us develop our team skills and understanding of each other. In the future, we plan to bring our project to a separate server and connect it to one and a cafe in the city of Vilnius.