



**Министерство науки и высшего образования Российской
Федерации Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

**Факультет «Информатика и системы управления»
Кафедра ИУ5 «Системы обработки информации и управления»**

Лабораторная работа №4
по дисциплине «Базовые компоненты интернет-технологий»

Выполнил:
студент группы ИУ5-33Б
Ахтамбаев Л.Н.

Проверил:
Канев А.И.

2021 г.

Оглавление

Постановка задачи:	3
Текст программы:.....	4
Файл builder.py	4
Файл command.py	7
Файл composite.py.....	9
Файл main.py	10
Файл TDD_test.py	11
Файл steps.py	11
Файл first_feature.feature	11
Файл mock_test.py.....	12
Экранные формы с примерами выполнения программы:	13

Постановка задачи:

1. Необходимо для произвольной предметной области реализовать от одного до трех шаблонов проектирования: один порождающий, один структурный и один поведенческий. В качестве справочника шаблонов можно использовать [следующий каталог](#). Для сдачи лабораторной работы в минимальном варианте достаточно реализовать один паттерн.
2. Вместо реализации паттерна Вы можете написать тесты для своей программы решения биквадратного уравнения. В этом случае, возможно, Вам потребуется доработать программу решения биквадратного уравнения, чтобы она была пригодна для модульного тестирования.
3. В модульных тестах необходимо применить следующие технологии:
 - TDD - фреймворк.
 - BDD - фреймворк.
 - Создание Mock-объектов.

Текст программы:

Файл builder.py

```
from abc import ABC, abstractmethod
from enum import Enum, auto
from collections import namedtuple

FullEngine = namedtuple('FullEngine', ['Type', 'Volume', 'Horsepower'])

'''
Классы составных частей
'''

class CarEngine(Enum):
    SUV_ENGINE = '2UZ-FE'
    SPORT_ENGINE = '2JZ-GTE'
    VAN_ENGINE = '15'

class CarBody(Enum):
    SUV_BODY = auto()
    SPORT_BODY = auto()
    VAN_BODY = auto()

class CarTires(Enum):
    SUV_TIRES = '245/70 R16'
    SPORT_TIRES = '275/35 R19'
    VAN_TIRES = '215/60 R15'

class CarDrive(Enum):
    FRONT_DRIVE = auto()
    REAR_DRIVE = auto()
    FULL_DRIVE = auto()

class CarInterior(Enum):
    BASIC_INTERIOR = auto()
    EXTENDED_INTERIOR = auto()
    PREMIUM_INTERIOR = auto()

'''
Класс компонуемого объекта
'''

class Car:
    def __init__(self, name):
        self.name = name
        self.engine = None
        self.body = None
        self.tires = None
        self.drive = None
        self.interior = None

    def __str__(self):
        info: str = f"Car name: {self.name} \n" \
```

```

        f"Body type: {self.body} \n" \
        f"Engine specifications: {self.engine.Type} " \
        f"{self.engine.Volume}" \
        f" {self.engine.Horsepower} \n" \
        f"Tires: {self.tires} \n" \
        f"Type of drive: {self.drive} \n" \
        f"Selected interior: {self.interior} \n"

    return info

'''
Абстрактный класс для задания интерфейса строителя
'''

class Builder(ABC):
    @abstractmethod
    def select_body(self) -> None: pass

    @abstractmethod
    def select_engine(self) -> None: pass

    @abstractmethod
    def select_tires(self) -> None: pass

    @abstractmethod
    def select_drive(self) -> None: pass

    @abstractmethod
    def select_interior(self) -> None: pass

    @abstractmethod
    def get_car(self) -> Car: pass

'''
Реализация конкретных строителей для трех типов машин
'''

class SuvCarBuilder(Builder):
    def __init__(self):
        self.car = Car("Toyota Tundra")

    def select_body(self) -> None:
        self.car.body = CarBody.SUV_BODY.name

    def select_engine(self) -> None:
        self.car.engine = FullEngine(CarEngine.SUV_ENGINE.value, '4.7 L',
'381 HP')

    def select_drive(self) -> None:
        self.car.drive = CarDrive.FULL_DRIVE.name

    def select_tires(self) -> None:
        self.car.tires = CarTires.SUV_TIRES.value

    def select_interior(self) -> None:
        self.car.interior = CarInterior.BASIC_INTERIOR.name

    def get_car(self) -> Car:
        return self.car

```

```

class SportCarBuilder(Builder):
    def __init__(self):
        self.car = Car("Toyota Supra")

    def select_body(self) -> None:
        self.car.body = CarBody.SPORT_BODY.name

    def select_engine(self) -> None:
        self.car.engine = FullEngine(CarEngine.SPORT_ENGINE.value, '3 L',
'280 HP')

    def select_drive(self) -> None:
        self.car.drive = CarDrive.REAR_DRIVE.name

    def select_tires(self) -> None:
        self.car.tires = CarTires.SPORT_TIRES.value

    def select_interior(self) -> None:
        self.car.interior = CarInterior.PREMIUM_INTERIOR.name

    def get_car(self) -> Car:
        return self.car

class VanCarBuilder(Builder):
    def __init__(self):
        self.car = Car("Volkswagen Transporter")

    def select_body(self) -> None:
        self.car.body = CarBody.VAN_BODY.name

    def select_engine(self) -> None:
        self.car.engine = FullEngine(CarEngine.VAN_ENGINE.value, '2.5 L',
'150 HP')

    def select_drive(self) -> None:
        self.car.drive = CarDrive.FRONT_DRIVE.name

    def select_tires(self) -> None:
        self.car.tires = CarTires.VAN_TIRES.value

    def select_interior(self) -> None:
        self.car.interior = CarInterior.EXTENDED_INTERIOR.name

    def get_car(self) -> Car:
        return self.car

"""
Класс Director, отвечающий за процесс поэтапной сборки машины
"""

class Director:
    def __init__(self):
        self.builder = None

    def set_builder(self, builder: Builder):
        self.builder = builder

    def build_car(self):
        if not self.builder:
            raise ValueError("Сборщик не установлен")

```

```
self.builder.select_body()
self.builder.select_engine()
self.builder.select_drive()
self.builder.select_tires()
self.builder.select_interior()
```

Файл command.py

```
from abc import ABC, abstractmethod
from typing import List

class ICommand(ABC):
    '''
    Интерфейс для выполняемых команд
    '''

    @abstractmethod
    def execute(self) -> None:
        pass

class Mechanic:
    def install_body(self) -> None:
        print("Installing car body")

    def install_engine_and_transmission(self) -> None:
        print("Installing engine and transmission")

    def install_interior(self) -> None:
        print("Installing interior")

class MechanicAssistant:
    def assemble_engine(self) -> None:
        print("Assembling engine")

    def assemble_tires(self) -> None:
        print("Assembling tires")

class Tools:
    def prepare_tools(self) -> None:
        print("Preparing all the tools")

    def remove_tools(self) -> None:
        print("Removing the tools")

class InstallBodyCommand(ICommand):
    def __init__(self, executor: Mechanic):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.install_body()

class InstallEngineTransmissionCommand(ICommand):
    def __init__(self, executor: Mechanic):
        self.__executor = executor

    def execute(self) -> None:
```

```

        self.__executor.install_engine_and_transmission()

class InstallInteriorCommand(ICommand):
    def __init__(self, executor: Mechanic):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.install_interior()

class AssembleEngineCommand(ICommand):
    def __init__(self, executor: MechanicAssistant):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.assemble_engine()

class AssembleTiresCommand(ICommand):
    def __init__(self, executor: MechanicAssistant):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.assemble_tires()

class PrepareToolsCommand(ICommand):
    def __init__(self, executor: Tools):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.prepare_tools()

class RemoveToolsCommand(ICommand):
    def __init__(self, executor: Tools):
        self.__executor = executor

    def execute(self) -> None:
        self.__executor.remove_tools()

class AutoRepairShop:
    def __init__(self):
        self.history: List[ICommand] = []

    def add_command(self, command: ICommand) -> None:
        self.history.append(command)

    def build(self) -> None:
        if not self.history:
            print("No commands in order")

        else:
            for executor in self.history:
                executor.execute()
            self.history.clear()

```


Файл composite.py

```
from abc import ABC, abstractmethod

class IPart(ABC):
    """
    Интерфейс частей автомобиля
    """

    @abstractmethod
    def name(self) -> str:
        pass

    @abstractmethod
    def cost(self) -> float:
        pass

class Part(IPart):
    """
    Класс части автомобиля
    """

    def __init__(self, name: str, cost: float):
        try:
            self.__cost = float(cost)
        except:
            self.__cost = 0
        self.__name = name

    def cost(self) -> float:
        return self.__cost

    def name(self) -> str:
        return self.__name

class ComplexPart(IPart):
    """
    Класс составных частей автомобиля
    """

    def __init__(self, name: str):
        self.__name = name
        self.parts = []

    def cost(self):
        cost = 0
        for i in self.parts:
            cost += i.cost()
        return cost

    def name(self) -> str:
        return self.__name

    def add_product(self, part: IPart):
        self.parts.append(part)

    def remove_product(self, part: IPart):
        self.parts.remove(part)

    def clear(self):
        self.parts = []
```

```

class Car(ComplexPart):
    '''
    Класс машины, состоящей из частей
    '''

    def __init__(self, name: str):
        super(Car, self).__init__(name)

    def cost(self):
        cost = 0
        for i in self.parts:
            cost_i = i.cost()
            print(f"Price for '{i.name()}' is {cost_i} dollars")
            cost += cost_i

        print(f"Price for '{self.name()}' is {cost} dollars")
        return cost

```

Файл main.py

```

from builder import Director, SuvCarBuilder, SportCarBuilder, VanCarBuilder
from composite import ComplexPart, Part, Car
from command import Mechanic, MechanicAssistant, Tools, AutoRepairShop,
PrepareToolsCommand, RemoveToolsCommand, \
    AssembleTiresCommand, AssembleEngineCommand, \
InstallEngineTransmissionCommand, InstallInteriorCommand, \
    InstallBodyCommand

if __name__ == '__main__':
    print('FIRST PATTERN: BUILDER \n')
    director = Director()
    for selected_car in (SuvCarBuilder, SportCarBuilder, VanCarBuilder):
        builder = selected_car()
        director.set_builder(builder)
        director.build_car()
        car = builder.get_car()
        print(car)

    print('SECOND PATTERN: COMPOSITE \n')
    engine = ComplexPart('Engine')
    engine.add_product(Part('Cylinders', 100))
    engine.add_product(Part('Pistons', 120))
    body = Part('SUV Body', 2300)
    tires = Part('SUV tires', 200)
    SUV = Car('Toyota Tundra')
    SUV.add_product(engine)
    SUV.add_product(body)
    SUV.add_product(tires)
    print(SUV.cost())

    print('THIRD PATTERN: COMMAND \n')
    mechanic = Mechanic()
    mechanic_assistant = MechanicAssistant()
    tools = Tools()
    auto_repair_shop = AutoRepairShop()
    auto_repair_shop.add_command(PrepareToolsCommand(tools))
    auto_repair_shop.add_command(AssembleTiresCommand(mechanic_assistant))
    auto_repair_shop.add_command(AssembleEngineCommand(mechanic_assistant))
    auto_repair_shop.add_command(InstallBodyCommand(mechanic))
    auto_repair_shop.add_command(InstallEngineTransmissionCommand(mechanic))
    auto_repair_shop.add_command(InstallInteriorCommand(mechanic))

```

```
auto_repair_shop.add_command(RemoveToolsCommand(tools))
auto_repair_shop.build()
```

Файл TDD_test.py

```
import unittest
import sys, os
from composite import *

sys.path.append(os.getcwd())

class TestPartCost(unittest.TestCase):
    def test_part_cost_is_working(self):
        engine = ComplexPart('Engine')
        engine.add_product(Part('Cylinders', 100))
        engine.add_product(Part('Pistons', 120))
        self.assertEqual(engine.cost(), 220)

    def test_part_cost_receives_string_is_working(self):
        engine = ComplexPart('Engine')
        engine.add_product(Part('Cylinders', '100'))
        engine.add_product(Part('Pistons', '120'))
        self.assertIsInstance(engine.cost(), float)

if __name__ == '__main__':
    unittest.main()
```

Файл steps.py

```
from behave import *
from TDD_test import *

@given(
    "I have pistons for 120 dollars and cylinders for 100 dollars")
def have_prices(context):
    context.a = TestPartCost()

@when("I put them into engine")
def engine_combine(context):
    context.a.test_part_cost_is_working()

@then("I expect engine to cost 220 dollars")
def check_result(context):
    pass
```

Файл first_feature.feature

```
Feature: Test
  Scenario: Test my function
    Given I have pistons for 120 dollars and cylinders for 100 dollars
    When I put them into engine
    Then I expect engine to cost 220 dollars
```

Файл mock_test.py

```
import unittest
import sys, os
from unittest.mock import patch, Mock

import composite

sys.path.append(os.getcwd())
from composite import *

class TestComposite(unittest.TestCase):
    @patch.object(composite.Car, 'cost')
    def test_car_cost(self, mock_cost):
        mock_cost.return_value = "100"
        SUV = Car("SUV")
        self.assertEqual(SUV.cost(), "100")
```

Экранные формы с примерами выполнения программы:

1. Результаты работы трех шаблонов из файла main.py:

```
FIRST PATTERN: BUILDER
```

```
Car name: Toyota Tundra
```

```
Body type: SUV_BODY
```

```
Engine specifications: 2UZ-FE 4.7 L 381 HP
```

```
Tires: 245/70 R16
```

```
Type of drive: FULL_DRIVE
```

```
Selected interior: BASIC_INTERIOR
```

```
Car name: Toyota Supra
```

```
Body type: SPORT_BODY
```

```
Engine specifications: 2JZ-GTE 3 L 280 HP
```

```
Tires: 275/35 R19
```

```
Type of drive: REAR_DRIVE
```

```
Selected interior: PREMIUM_INTERIOR
```

```
Car name: Volkswagen Transporter
```

```
Body type: VAN_BODY
```

```
Engine specifications: 15 2.5 L 150 HP
```

```
Tires: 215/60 R15
```

```
Type of drive: FRONT_DRIVE
```

```
Selected interior: EXTENDED_INTERIOR
```

```
SECOND PATTERN: COMPOSITE
```

```
Price for 'Engine' is 220 dollars
```

```
Price for 'SUV Body' is 2300 dollars
```

```
Price for 'SUV tires' is 200 dollars
```

```
Price for 'Toyota Tundra' is 2720 dollars
```

```
2720
```

```
THIRD PATTERN: COMMAND
```

```
Preparing all the tools
```

```
Assembling tires
```

```
Assembling engine
```

```
Installing car body
```

```
Installing engine and transmission
```

```
Installing interior
```

```
Removing the tools
```

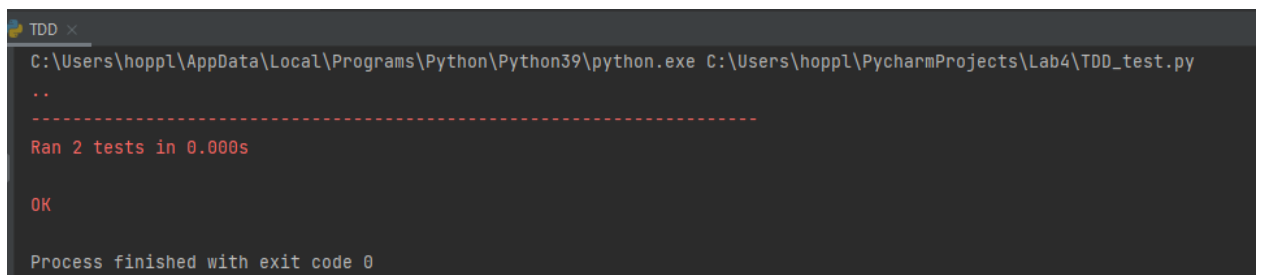
2. Тестирование.

2.1. TDD – фреймворк

Для реализации TDD – фреймворка подключим модуль unittest, создадим папку testing и файл TDD_test.

TDD – разработка на основе тестов, хорошо подходит для проверки работ отдельных модулей. Такое тестирование проходит в три этапа, Red, Green, Refactoring, т.е. сначала тест проваливается, потом мы должны реализовать код, чтобы тесты проходили, далее мы должны привести код к максимально короткому и правильному решению.

TDD тестирование проведем для шаблона Composite, а именно для классов Part и ComplexPart.



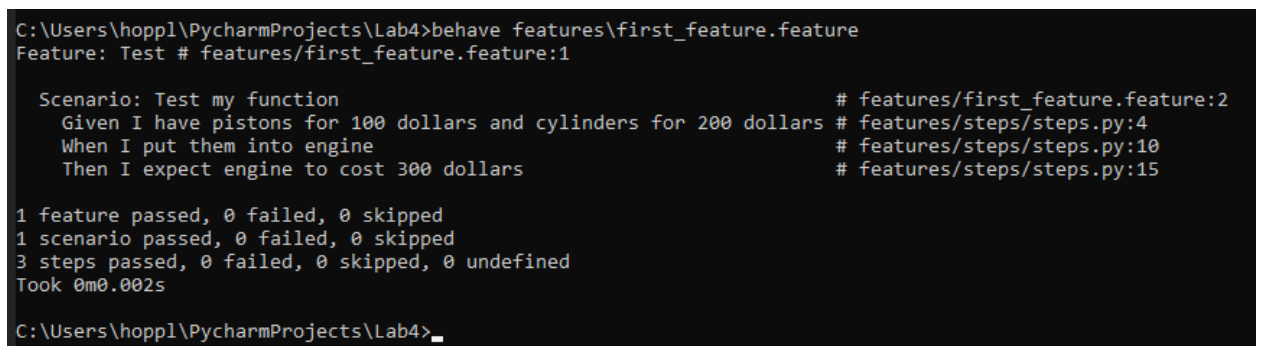
```
TDD x
C:\Users\hoppl\AppData\Local\Programs\Python\Python39\python.exe C:\Users\hoppl\PycharmProjects\Lab4\TDD_test.py
..
-----
Ran 2 tests in 0.000s

OK

Process finished with exit code 0
```

2.2. BDD – фреймворк

BDD – разработка на основе поведения. BDD является расширением TDD – подхода.



```
C:\Users\hoppl\PycharmProjects\Lab4>behave features\first_feature.feature
Feature: Test # features/first_feature.feature:1

  Scenario: Test my function # features/first_feature.feature:2
    Given I have pistons for 100 dollars and cylinders for 200 dollars # features/steps/steps.py:4
    When I put them into engine # features/steps/steps.py:10
    Then I expect engine to cost 300 dollars # features/steps/steps.py:15

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.002s

C:\Users\hoppl\PycharmProjects\Lab4>
```

2.3. Создание Моск-объектов.

