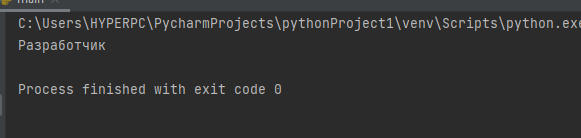
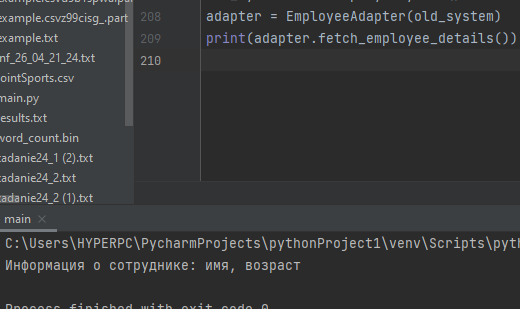
from abc import ABC, abstractmethod  
  
# Абстрактный класс для сотрудника  
class Employee(ABC):  
 @abstractmethod  
 def get\_role(self):  
 pass  
  
# класс для менеджера  
class Manager(Employee):  
 def get\_role(self):  
 return "Менеджер"  
  
#класс для разработчика  
class Developer(Employee):  
 def get\_role(self):  
 return "Разработчик"  
  
#класс для дизайнера  
class Designer(Employee):  
 def get\_role(self):  
 return "Дизайнер"  
  
  
class EmployeeFactory:  
 @staticmethod  
 def create\_employee(employee\_type: str) -> Employee:  
 if employee\_type == "Manager":  
 return Manager()  
 elif employee\_type == "Developer":  
 return Developer()  
 elif employee\_type == "Designer":  
 return Designer()  
 else:  
 raise ValueError("Неизвестный тип сотрудника")  
  
  
employee1 = EmployeeFactory.create\_employee("Developer")  
print(employee1.get\_role()) # Вывод: Разработчик



# Старая система  
class OldEmployeeSystem:  
 def get\_employee\_info(self):  
 return "Информация о сотруднике: имя, возраст"  
  
# Новый интерфейс  
class NewEmployeeSystem:  
 def fetch\_employee\_details(self):  
 return "Детали сотрудника: должность, зарплата"  
  
# Адаптер для совместимости старой системы с новой  
class EmployeeAdapter(NewEmployeeSystem):  
 def \_\_init\_\_(self, old\_employee\_system: OldEmployeeSystem):  
 self.old\_employee\_system = old\_employee\_system  
  
 def fetch\_employee\_details(self):  
 return self.old\_employee\_system.get\_employee\_info()  
  
  
old\_system = OldEmployeeSystem()  
adapter = EmployeeAdapter(old\_system)  
print(adapter.fetch\_employee\_details()) # Вывод: Информация о сотруднике: имя, возраст



from abc import ABC, abstractmethod  
  
# Абстрактный класс стратегии расчета зарплаты  
class SalaryStrategy(ABC):  
 @abstractmethod  
 def calculate\_salary(self, base\_salary: float) -> float:  
 pass  
  
# Конкретная стратегия для менеджера  
class ManagerSalaryStrategy(SalaryStrategy):  
 def calculate\_salary(self, base\_salary: float) -> float:  
 return base\_salary \* 1.5  
  
# Конкретная стратегия для разработчика  
class DeveloperSalaryStrategy(SalaryStrategy):  
 def calculate\_salary(self, base\_salary: float) -> float:  
 return base\_salary \* 1.2  
  
# Конкретная стратегия для дизайнера  
class DesignerSalaryStrategy(SalaryStrategy):  
 def calculate\_salary(self, base\_salary: float) -> float:  
 return base\_salary \* 1.1  
  
# Контекст использующий стратегию  
class Employee:  
 def \_\_init\_\_(self, name: str, role: str, salary\_strategy: SalaryStrategy):  
 self.name = name  
 self.role = role  
 self.salary\_strategy = salary\_strategy  
  
 def calculate\_salary(self, base\_salary: float) -> float:  
 return self.salary\_strategy.calculate\_salary(base\_salary)  
  
  
employee1 = Employee("Иван", "Менеджер", ManagerSalaryStrategy())  
print(employee1.calculate\_salary(1000)) # Вывод: 1500.0  
  
employee2 = Employee("Анна", "Разработчик", DeveloperSalaryStrategy())  
print(employee2.calculate\_salary(1000)) # Вывод: 1200.0

