МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ВЛАДИВОСТОКСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВВГУ»)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АНАЛИЗА ДАННЫХ
КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ

ОТЧЕТ
ПО ФИНАЛЬНОЙ ЛАБОРАТОРНОЙ РАБОТЕ

по дисциплине
«Информатика и программирование»

Студент
гр. БИН-25-02 _____ А.В. Воронин

Ассистент
преподавателя _____ М.В. Водяницкий

Владивосток 2026

Задание

Разработать консольную систему управления автозаправочной станцией на языке Python. Система должна имитировать реальные процессы работы заправочной станции и предоставлять оператору возможность управления всеми аспектами работы АЗС.

Основные требования к системе:

1. Система учитывает несколько типов топлива: АИ-92, АИ-95, АИ-98 и ДТ.

2. Реализовать систему подземных цистерн для хранения топлива. Каждая цистерна должна иметь следующие характеристики: тип топлива, максимальный объем, текущий уровень, минимальный допустимый уровень, состояние включена или отключена.

3. Реализовать систему заправочных колонок. Всего должно быть 8 колонок, каждая из которых поддерживает несколько типов топлива. Каждый тип топлива на колонке подключен к конкретной цистерне.

4. Схема подключений должна соответствовать следующим правилам: АИ-92 доступен на колонках 1-6 и подключен к цистерне Т1, АИ-95 доступен на колонках 1-4 цистерна Т2 и 5-8 цистерна Т3, АИ-98 доступен на колонках 3-6 и подключен к цистерне Т4, ДТ доступен на колонках 3-8 и подключен к цистерне Т5.

5. Реализовать меню управления со следующими функциями: обслуживание клиентов продажа топлива, просмотр состояния цистерн, пополнение запасов топлива, просмотр статистики продаж, просмотр истории операций, перекачка топлива между цистернами, управление состоянием цистерн включение или отключение, аварийный режим работы, снятие аварийного режима.

6. При продаже топлива необходимо выполнять следующие проверки: доступность выбранной колонки, состояние цистерны включена или отключена, достаточность топлива в цистерне, корректность введенных данных.

7. Реализовать автоматическое отключение цистерны при падении уровня топлива ниже минимального порога.

8. Включение цистерны после пополнения должно выполняться только вручную через меню управления.

9. Все данные должны сохраняться в файле формата JSON для сохранения состояния между запусками программы.

10. Реализовать ведение статистики продаж: общий доход, количество обслуженных автомобилей, объемы продаж по типам топлива, доход по каждому виду топлива.

11. Реализовать историю операций с временными метками для отслеживания всех действий в системе.

12. Реализовать аварийный режим, который полностью блокирует работу АЗС и фиксирует аварийное событие в истории.

13. Программа должна быть консольной и работать через текстовое меню с вводом команд с клавиатуры пользователем.

14. Код должен быть структурирован, использовать объектно-ориентированный подход и содержать обработку исключений для корректной работы при возможном некорректном пользовательским вводом.

Содержание

Введение

В современном мире автоматизация процессов управления различными объектами приобретает все большее значение. Одним из таких объектов является автозаправочная станция (АЗС), работа которой требует точного учета топлива, контроля оборудования и ведения статистики продаж. Разработка программного обеспечения для управления АЗС позволяет оптимизировать рабочие процессы, повысить эффективность обслуживания клиентов и обеспечить безопасность эксплуатации оборудования.

Данная лабораторная работа посвящена созданию консольной системы управления автозаправочной станцией на языке программирования Python. Целью работы является разработка программного комплекса, имитирующего реальные процессы работы АЗС и предоставляющего оператору инструменты для управления всеми аспектами деятельности станции. Система должна учитывать особенности работы с различными типами топлива, контролировать состояние подземных цистерн, управлять заправочными колонками, вести учет операций и обеспечивать безопасность через механизм аварийного режима.

Актуальность работы обусловлена необходимостью создания учебных моделей реальных систем управления для подготовки специалистов в области информационных технологий и автоматизации. Разрабатываемая система позволяет на практике изучить принципы объектно-ориентированного программирования, работу с файлами JSON, обработку исключений и создание пользовательских интерфейсов в консольных приложениях.

В процессе выполнения работы были использованы современные подходы к проектированию программного обеспечения, включая модульную архитектуру, разделение ответственности между компонентами системы и обеспечение целостности данных. Особое внимание уделено созданию удобного интерфейса для оператора, реализации механизмов проверки корректности ввода и обеспечению надежности работы системы в различных сценариях использования.

Система разработана в полном соответствии с техническим заданием и включает все требуемые функциональные возможности. Результатом работы является полностью функционирующее приложение, готовое к использованию в учебных целях и демонстрирующее практическое применение полученных знаний в области программирования и проектирования программных систем.

1 Выполнение работы

1.1 Структура программы

Система управления АЗС разработана на языке Python с использованием объектно-ориентированного подхода. Программа состоит из трех основных классов и главного управляющего цикла. Класс Tank моделирует цистерну для хранения топлива, класс Dispenser представляет заправочную колонку, а класс GasStation является центральным управляющим компонентом всей системы. Архитектура программы позволяет эффективно управлять всеми процессами на автозаправочной станции и соответствует требованиям технического задания.

Импорт необходимых библиотек и настройка основных констант программы представлены на рисунке 1.

```
import json
import os
from datetime import datetime

DATA_FILE = "data.json"

PRICES = {
    "АИ-92": 50.0,
    "АИ-95": 58.3,
    "АИ-98": 65.0,
    "ДТ": 55.0
}
```

Рисунок 1 – Импорт библиотек и настройка констант

Программа использует стандартные библиотеки Python для работы с файлами, JSON форматом и временными метками. Цены на топливо заданы в словаре PRICES, что позволяет легко изменять их при необходимости, что будет целиком и полностью влиять на экономику внутри симулятора.

1.2 Класс Tank цистерна

Класс Tank является одним из главных классов всей программы и представляет собой модель цистерны для хранения топлива. Каждая цистерна имеет уникальный идентификатор и характеристики, соответствующие техническому заданию. Основные характеристики цистерны включают идентификатор цистерны, тип хранимого топлива, максимальный объем цистерны, текущий объем топлива в цистерне, минимальный допустимый уровень топлива, состояние включена или отключена цистерна и флаг блокировки для аварийного режима, который можно включать или выключать, а также соответствующие функции для управления цистерной и топливом внутри неё. Конструктор класса принимает все необходимые параметры для инициализации цистерны. Параметры enabled и

blocked имеют значения по умолчанию, что упрощает создание новых объектов, если их не нужно преждевременно блокировать или выключать.

Конструктор и основные методы класса Tank, обеспечивающие ключевые операции с цистерной, представлены на рисунке 2.

```
class Tank:
    def __init__(self, id, fuel_type, max_volume,
                 current_volume, min_level, enabled=True, blocked=False):
        self.id = id
        self.fuel_type = fuel_type
        self.max_volume = max_volume
        self.current_volume = current_volume
        self.min_level = min_level
        self.enabled = enabled
        self.blocked = blocked
    def check_min_level(self):
        if self.current_volume < self.min_level:
            self.enabled = False

    def dispense(self, liters):
        if self.current_volume >= liters:
            self.current_volume -= liters
            self.check_min_level()
            return True
        return False

    def refill(self, liters):
        if self.current_volume + liters <= self.max_volume:
            self.current_volume += liters
            return True
        return False
```

Рисунок 2 – Конструктор и основные методы класса Tank

Метод check_min_level автоматически отключает цистерну при падении уровня топлива ниже минимального порога. Метод dispense отвечает за отпуск топлива с проверкой достаточности, а метод refill позволяет пополнять запас топлива с контролем переполнения.

Методы перекачки топлива между цистернами и сериализации объекта показаны на рисунке 3.

```
    def transfer_to(self, other, liters):
        if self.fuel_type != other.fuel_type:
            return False
        if self.current_volume < liters:
            return False
        if other.current_volume + liters > other.max_volume:
            return False

        self.current_volume -= liters
        other.current_volume += liters
        self.check_min_level()
        return True

    def to_dict(self):
        return self.__dict__
```

Рисунок 3 – Методы перекачки и сериализации

Метод transfer_to обеспечивает перекачку топлива между цистернами одного типа с выполнением всех необходимых проверок. Метод to_dict преобразует объект в словарь для сохранения в JSON формате, что обеспечивает сохранение состояния между запусками программы.

1.3 Класс Dispenser колонка

Класс Dispenser представляет заправочную колонку. Каждая колонка имеет словарь подключений, который определяет, к каким цистернам подключены различные типы топлива на данной колонке. Структура словаря подключений: ключ тип топлива, значение идентификатор цистерны.

Реализация класса Dispenser представлена на рисунке 4.

```
class Dispenser:
    def __init__(self, id, connections):
        self.id = id
        self.connections = connections

    def to_dict(self):
        return self.__dict__
```

Рисунок 4 – Класс Dispenser

Класс Dispenser имеет простую структуру, но играет важную роль в системе. Словарь connections определяет маршрутизацию топлива от колонки к соответствующим цистернам, что позволяет гибко настраивать схему подключений.

1.4 Класс GasStation АЗС

Класс GasStation является центральным компонентом системы и реализует все функции управления АЗС. Он содержит коллекции цистерн и колонок, а также данные статистики и истории операций.

Конструктор класса GasStation и процесс инициализации системы показаны на рисунке 5.

```
class GasStation:
    def __init__(self):
        self.tanks = {}
        self.dispensers = {}
        self.stats = {
            "income": 0,
            "cars": 0,
            "liters": {},
            "fuel_income": {}
        }
        self.history = []
        self.emergency_mode = False
        self.load()
```

Рисунок 5 – Конструктор и инициализация GasStation

При инициализации создаются пустые структуры данных для хранения информации о цистернах, колонках, статистике и истории. Метод load вызывается сразу после создания объекта для загрузки сохраненного состояния из файла.

1.5 Инициализация и сохранение данных

Программа использует файл data.json для хранения состояния АЗС между запусками. Данные сохраняются в структурированном формате JSON, что обеспечивает удобство чтения и редактирования. Метод load класса GasStation загружает данные из файла при инициализации, а метод save сохраняет текущее состояние системы. Этот подход соответствует требованию технического задания о сохранении состояния между запусками программы. Механизм сохранения и загрузки данных реализован с учетом особенностей работы автозаправочной станции. Система должна сохранять не только текущие объемы топлива в цистернах, но и всю историю операций, статистику продаж, а также состояние оборудования включение или отключение цистерн, аварийный режим. Использование формата JSON позволяет легко читать и анализировать сохраненные данные без необходимости запуска самой программы, что полезно для аудита и отладки.

Пример части структуры данных JSON, используемой для хранения состояния системы, показан на рисунке 6.

```
{
    "tanks": [
        {
            "id": "T1",
            "fuel_type": "АИ-92",
            "max_volume": 20000,
            "current_volume": 15000,
            "min_level": 1000,
            "enabled": true,
            "blocked": false
        }
    ],
    "dispensers": [
        {
            "id": 1,
            "connections": {
                АИ"-92": "T1",
                АИ"-95": "T2",
                АИ"-98": "T4",
                ДТ"": "T5"
            }
        }
    ],
}
```

Рисунок 6 – Пример структуры данных JSON

Файл данных содержит всю необходимую информацию для восстановления состояния системы. Структура файла хорошо организована и легко читаема, что упрощает отладку и модификацию данных при необходимости.

1.6 Основное меню программы

Главный цикл программы предоставляет пользователю меню с девятью основными функциями управления АЗС. Меню реализовано с использованием цикла while True, что обеспечивает непрерывную работу программы до явного выхода пользователя. Каж-

дый пункт меню вызывает соответствующий метод класса GasStation, что обеспечивает модульность и удобство поддержки кода.

Архитектура главного меню построена по принципу единой точки входа для всех операций управления станцией. Пользовательский интерфейс представляет собой текстовое меню с последовательной нумерацией опций от 1 до 9, где каждая цифра соответствует конкретной функции.

Начало главного цикла программы с отображением меню управления представлено на рисунке 7.

```
station = GasStation()

while True:
    print("\n=== СИСТЕМА УПРАВЛЕНИЯ АЗС ===")
    print("1) Обслужить клиента")
    print("2) Состояние цистерн")
    print("3) Пополнение")
    print("4) Статистика")
    print("5) История")
    print("6) Перекачка")
    print("7) Управление цистернами")
    print("8) EMERGENCY")
    print("9) Снять аварийный режим")
    print("0) Выход")

    choice = input("> ")
```

Рисунок 7 – Главный цикл программы

Меню организовано интуитивно понятно и предоставляет доступ ко всем необходимым функциям управления АЗС.

Обработка выбора пользователя в главном меню показана на рисунке 8.

```
    if choice == "1":
        station.serve_client()
    elif choice == "2":
        station.show_tanks()
    elif choice == "3":
        station.refill()
    elif choice == "4":
        station.show_stats()
    elif choice == "5":
        station.show_history()
    elif choice == "6":
        station.transfer()
    elif choice == "7":
        station.manage_tanks()
    elif choice == "8":
        station.emergency()
    elif choice == "9":
        station.exit_emergency()
    elif choice == "0":
        break
```

Рисунок 8 – Обработка выбора меню

Пользователь может легко выбирать нужные операции с помощью цифрового ввода. Каждый выбор пользователя обрабатывается соответствующей функцией, что обеспечивает

четкое разделение ответственности между различными модулями программы. Такой подход упрощает добавление новых функций и модификацию существующих.

1.7 Функция обслуживания клиентов

Функция serve_client реализует процесс продажи топлива клиенту. Алгоритм работы включает несколько этапов проверки и подтверждения. Этапы обслуживания клиента включают выбор номера колонки от 1 до 8, выбор типа топлива из доступных на выбранной колонке, проверку состояния подключенной цистерны, ввод количества литров с проверкой корректности, расчет стоимости на основе фиксированных цен, подтверждение операции пользователем и обновление данных с сохранением в файл. На каждом этапе обработки исключений программа обеспечивает безопасное продолжение работы, возвращаясь в главное меню при возникновении ошибок. Такой подход предотвращает аварийное завершение программы из-за некорректного пользовательского ввода и позволяет оператору повторить операцию с правильными данными. Конструкции try-except также используются в других функциях системы, таких как пополнение топлива, перекачка между цистернами и управление состоянием оборудования, обеспечивая надежность работы всего приложения.

Начальная часть метода обслуживания клиента, включающая выбор колонки и типа топлива, представлена на рисунке 9.

```
def serve_client(self):
    if self.emergency_mode:
        print("АЗС в аварийном режиме!")
        return

    try:
        col = int(input("Выберите колонку (1-8): "))
        dispenser = self.dispensers[col]
    except:
        print("Ошибка!")
        return

    fuels = list(dispenser.connections.keys())
    for i, f in enumerate(fuels, 1):
        print(f"{i}) {f}")

    try:
        choice = int(input("Выберите топливо: "))
        fuel = fuels[choice - 1]
    except:
        print("Ошибка!")
        return
```

Рисунок 9 – Выбор колонки и типа топлива

В начале функции проверяется, не находится ли АЗС в аварийном режиме. Затем пользователь выбирает колонку и тип топлива. Процесс определения цистерны представляет собой ключевой этап в цепочке обслуживания клиента. Система использует словарь connections объекта Dispenser, который содержит сопоставление типов топлива с иденти-

фикаторами цистерн. Проверка доступности цистерны и ввод количества топлива показаны на рисунке 10.

```
    tank = self.tanks[dispenser.connections[fuel]]

    if not tank.enabled or tank.blocked:
        print("Цистерна недоступна!")
        return

    try:
        liters = float(input("Введите литры: "))
    except:
        print("Ошибка!")
        return

    if liters <= 0:
        print("Некорректное количество!")
        return
```

Рисунок 10 – Проверка доступности и ввод количества

При запросе конкретного типа топлива программа выполняет поиск в словаре connections по ключу, соответствующему выбранному топливу, и получает идентификатор цистерны. Получение объекта цистерны происходит через обращение к словарю tanks класса GasStation по найденному идентификатору. Этот словарь содержит все цистерны станции, обеспечивая быстрый доступ к нужному объекту. Система гарантирует, что идентификатор, хранящийся в connections, всегда соответствует существующей цистерне, что проверяется на этапе загрузки конфигурации и при любых изменениях схемы подключений.

Расчет стоимости и подтверждение операции представлены на рисунке 11.

```
    if not tank.dispense(liters):
        print("Недостаточно топлива!")
        return

    cost = liters * PRICES[fuel]
    print(f"К оплате: {cost:.2f} ₽")
    confirm = input("Подтвердить? (y/n): ")

    if confirm.lower() != "y":
        tank.current_volume += liters
        return

    self.stats["income"] += cost
    self.stats["cars"] += 1
    self.stats["liters"][fuel] += liters
    self.stats["fuel_income"][fuel] += cost

    self.history.append(f"{datetime.now()} - Продажа {fuel} {liters} л")
    self.save()
    print("Успешно!")
```

Рисунок 11 – Расчет и подтверждение операции

Финальная часть функции выполняет расчет стоимости, запрашивает подтверждение у пользователя и обновляет все данные системы. При отказе от операции топливо возвращается в цистерну. При подтверждении обновляются статистика, история и сохраняется новое состояние системы.

В функции реализована обработка всех возможных ошибок: некорректный ввод, недоступность цистерны, недостаток топлива. При отмене операции производится возврат топлива в цистерну. Это обеспечивает надежность и безопасность операций продажи топлива.

1.8 Управление цистернами

Функция show_tanks отображает текущее состояние всех цистерн в удобном табличном формате. Для каждой цистерны выводится следующая информация: идентификатор цистерны, тип топлива, текущий и максимальный объем, статус ВКЛ или ВЫКЛ.

Реализация метода отображения состояния цистерн представлена на рисунке 12.

```
def show_tanks(self):
    for t in self.tanks.values():
        status = "ВКЛ" if t.enabled else "ВЫКЛ"
        print(f"{t.id} | {t.fuel_type} | {t.current_volume}/{t.max_volume} |
     {status}")
```

Рисунок 12 – Метод отображения состояния цистерн

Функция manage_tanks позволяет оператору вручную включать или отключать цистерны. Включение возможно только при достаточном уровне топлива, что соответствует требованиям безопасности. Это обеспечивает контроль над работой оборудования и предотвращает попытки использования пустых или неисправных цистерн.

1.9 Система пополнения и перекачки

Функция refill имитирует прибытие бензовоза и пополнение запасов топлива. Реализация метода пополнения топлива показана на рисунке 13.

```
def refill(self):
    for t in self.tanks.values():
        print(f"{t.id} - {t.fuel_type}")

    tank_id = input("Выберите цистерну: ")
    if tank_id not in self.tanks:
        return

    try:
        liters = float(input("Введите литры: "))
    except:
        return

    if self.tanks[tank_id].refill(liters):
        self.history.append(f"{datetime.now()} - Пополнение {tank_id} {
    liters} л")
        self.save()
        print("Пополнение выполнено.")
    else:
        print("Переполнение!")
```

Рисунок 13 – Метод пополнения топлива

Оператор выбирает цистерну и указывает количество литров для пополнения, после чего система проверяет, чтобы новый объем не превышал максимальную вместимость

цистерны. Функция transfer позволяет перекачивать топливо между цистернами одного типа. Это полезно для балансировки запасов или подготовки к обслуживанию. Проверяются следующие условия: цистерны должны содержать один тип топлива, в исходной цистерне должно быть достаточно топлива, целевая цистерна должна иметь достаточную вместимость.

1.10 Статистика и история операций

Система ведет подробную статистику продаж, которая включает общий доход от продаж, количество обслуженных автомобилей, объемы проданного топлива по типам, доход по каждому виду топлива. История операций сохраняет все значимые события с временными метками: продажи топлива клиентам, пополнение запасов, перекачка между цистернами, изменение состояния цистерн, аварийные события.

Метод отображения статистики продаж представлен на рисунке 14.

```
def show_stats(self):
    print(f"Доход: {self.stats['income']:.2f} ₽")
    print(f"Машин: {self.stats['cars']}")
    for fuel in PRICES:
        print(f"{fuel}: {self.stats['liters'][fuel]} л | {self.stats['
    fuel_income'][fuel]:.2f} ₽")
```

Рисунок 14 – Метод отображения статистики

История хранится в виде списка строк с датой и описанием события, что позволяет отслеживать все действия в хронологическом порядке.

1.11 Аварийный режим

Аварийный режим реализован для обеспечения безопасности на АЗС. При активации этого режима все цистерны блокируются, продажа топлива прекращается, событие фиксируется в истории и система переходит в заблокированное состояние.

Метод активации аварийного режима показан на рисунке 15.

```
def emergency(self):
    confirm = input("Подтвердить аварийный режим? (y/n): ")
    if confirm.lower() == "y":
        self.emergency_mode = True
        for t in self.tanks.values():
            t.blocked = True
            t.enabled = False
        self.history.append(f"{datetime.now()} - АВАРИЯ")
        self.save()
        print("АЗС остановлена!")
```

Рисунок 15 – Метод активации аварийного режима

Выход из аварийного режима требует ручного подтверждения оператора. После снятия аварийного режима цистерны остаются отключенными и требуют ручного включе-

ния, что соответствует требованиям безопасности. Такая двухэтапная система обеспечивает дополнительный уровень защиты от случайных или ошибочных действий.

1.12 Обработка ошибок и исключений

Во всех функциях ввода реализована обработка исключений для предотвращения сбоев программы при некорректном вводе пользователя. Используются блоки try-except для преобразования строк в числа и проверки диапазонов значений. Основные проверки включают корректность преобразования строк в числовые типы, проверку диапазонов номеров колонок и количества литров, проверку существования выбранных элементов и проверку достаточности ресурсов.

Такой подход обеспечивает устойчивость программы к ошибкам ввода и предотвращает ее аварийное завершение. Пользователь получает понятные сообщения об ошибках и возможность исправить ввод без потери данных или необходимости перезапуска программы.

1.13 Тестирование работы программы

Для проверки корректности работы программы было выполнено тестирование всех функций системы. Тестирование включало проверку продажи топлива с различными комбинациями колонок и типов топлива, проверку автоматического отключения цистерн при низком уровне, тестирование пополнения запасов с проверкой переполнения, проверку перекачки топлива между цистернами, тестирование аварийного режима и восстановления работы, проверку сохранения и загрузки данных из файла.

Все функции работают корректно и соответствуют требованиям технического задания. Программа успешно имитирует работу автозаправочной станции, предоставляя оператору полный контроль над всеми процессами. Система демонстрирует надежность, удобство использования и соответствие всем указанным требованиям.

Заключение

В результате выполнения лабораторной работы была успешно разработана консольная система управления автозаправочной станцией на языке Python. Система полностью соответствует всем требованиям технического задания и демонстрирует практическое применение объектно-ориентированного подхода в программировании.

Главным достижением работы является создание целостной программной системы, которая имитирует реальные процессы работы АЗС. Разработанное приложение включает все необходимые функции для управления станцией: обслуживание клиентов, контроль состояния оборудования, управление запасами топлива, ведение статистики и истории операций, а также обеспечение безопасности через механизм аварийного режима.

Архитектура системы построена на принципах модульности и разделения ответственности. Три основных класса - Tank, Dispenser и GasStation - четко распределяют функциональность между собой, что обеспечивает удобство поддержки и расширения кода. Класс Tank отвечает за моделирование цистерн с топливом, класс Dispenser представляет заправочные колонки, а класс GasStation интегрирует все компоненты в единую систему управления.

Важным аспектом реализации является механизм сохранения и загрузки данных в формате JSON. Это решение позволяет сохранять состояние системы между запусками программы, обеспечивает целостность данных и упрощает отладку приложения. Пользовательский интерфейс в виде текстового меню интуитивно понятен и обеспечивает удобный доступ ко всем функциям системы.

Особое внимание в разработке было уделено обработке ошибок и исключений. Все функции ввода защищены от некорректных данных, что повышает надежность работы системы и предотвращает ее аварийное завершение. Механизм аварийного режима реализован с учетом требований безопасности и обеспечивает защиту от случайных или ошибочных действий оператора.

Система демонстрирует высокую практическую ценность как учебный проект. В процессе ее разработки были применены и закреплены ключевые концепции программирования: работа с классами и объектами, обработка исключений, работа с файлами, создание консольных пользовательских интерфейсов. Полученный опыт может быть полезен при разработке более сложных систем управления в будущем.