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 РАЗРАБОТКИ СИСТЕМЫ УПРАВЛЕНИЯ АВТОЗАПРАВОЧНОЙ СТАНЦИЕЙ НА PYTHON

по дисциплине «Информатика и программирование»

Студент
гр. БИН-25-3 _____ М.В. Кирийчук

Ассистент
преподавателя _____ М.В. Водяницкий

Владивосток 2025

Содержание

1 Введение

Автоматизация процессов на автозаправочных станциях (АЗС) играет ключевую роль в современном бизнесе. Консольные системы, используемые для этих целей, позволяют эффективно контролировать запасы топлива, вести учет продаж, оперативно реагировать на технические неисправности и обеспечивать бесперебойную работу станции.

Наша цель — разработать простую и надежную систему управления АЗС, которая будет функционировать без сбоев, будет интуитивно понятной и сможет эффективно выполнять все возложенные на нее задачи.

1.1 Принципы проектирования

При разработки были заложены следующие принципы:

– Надежность: все операции проверяются на корректность, некорректный ввод не приводит к аварийному завершению.

– Состоятельность: система сохраняет полное состояние между запусками.

– Безопасность: аварийный режим блокирует все операции и требует ручного восстановления.

– Модульность: код разделён на логические компоненты, каждый из которых отвечает за свою зону ответственности.

Эти принципы обеспечивают соответствие требованиям промышленного программного обеспечения.

2 Выполнение работы

2.1 Архетектура программы

Программа реализована в виде единого класса FuelStation, который инкапсулирует всю логику управления станцией. Основные компоненты:

– Данные цистерн: словарь cisterns с параметрами (тип, объём, состояние).

– Цены на топливо: словарь fuel_prices.

– Колонки: словарь pumps с привязкой видов топлива к цистернам.

– Финансы и статистика: баланс, количество обслуженных автомобилей, объёмы продаж.

– История операций: список history для аудита.

– Аварийный режим: флаг emergency_mode.

На рисунке 1 представлен код структуры класса FuelStation.

```
class FuelStation:
    def __init__(self):
        self.cisterns = {}
        self.fuel_prices = {
            'AI-92': 47.50,
            'AI-95': 51.20,
            'AI-98': 58.30,
            'DT': 56.00
        }
        self.pumps = {}
        self.balance = 0.0
        self.stats = {
            'cars_served': 0,
            'fuel_sold': {'AI-92': 0, 'AI-95': 0, 'AI-98': 0, 'DT': 0},
            'income_by_fuel': {'AI-92': 0, 'AI-95': 0, 'AI-98': 0, 'DT': 0}
        }
        self.history = []
        self.emergency_mode = False

        self.initialize_data()
```

Рисунок 1 – Основная структура класса

Такой подход позволяет управлять всеми аспектами станции в рамках одного объекта, что упрощает сериализацию и восстановление состояния.

2.2 Инициализация станции

При первом запуске создаётся стандартная конфигурация, соответствующая техническому заданию:

1) 5 цистерн с разными типами топлива (AI-92, AI-95, AI-98, DT) и начальными уровнями.

2) 8 колонок с заданными привязками к цистернам.

На рисунке 2 представлена инициализация цистерн.

```
def initialize_data(self):
    self.cisterns = {
        'AI-92_1': {
            'type': 'AI-92',
            'max_volume': 20000,
            'current_volume': 12400,
            'enabled': True,
            'min_level': 1000
        },
        'AI-95_1': {
            'type': 'AI-95',
            'max_volume': 20000,
            'current_volume': 9800,
            'enabled': True,
            'min_level': 1000
        },
        # ... остальные цистерны
    }
```

Рисунок 2 – Инициализация цистерн

Метод initialize_data заполняет структуры начальными значениями, после чего вызывается проверка уровня топлива для автоматического отключения цистерн с низким запасом.

2.3 Обслуживание клиентов

Метод serve_customer реализует полный цикл продажи:

– Проверка аварийного режима.

– Выбор доступной колонки.

– Выбор типа топлива с учётом состояния цистерны.

– Ввод количества литров и проверка достаточности запаса.

– Расчёт стоимости и подтверждение оплаты.

– Списание топлива, обновление статистики, запись в историю.

На рисунке 3 представлен код логики продаж.

```
def serve_customer(self):
    if self.emergency_mode:
        print("Ошибка: Станция находится в аварийном режиме!")
        return

    # Выбор колонки
    for pump_num in sorted(self.pumps.keys()):
        available_fuels = []
        for fuel_type, cistern_id in self.pumps[pump_num]['
    fuels'].items():
            if self.cisterns[cistern_id]['enabled']:
                available_fuels.append(fuel_type)

    # Проверка оплаты и списание топлива
    confirm = input("Подтвердить оплату? (y/n): ").lower()
    if confirm == 'y':
        self.cisterns[cistern_id]['current_volume'] -=
    liters
        self.balance += price
        self.stats['cars_served'] += 1
        self.log_operation('SALE', f"Колонка {pump_choice}, {
    fuel_type}, {litersл}, {price:.2f₽}")
```

Рисунок 3 – Основная логика продажи

Все этапы защищены проверками ввода, что исключает аварийное завершение при ошибках пользователя.

2.4 Пополнение и перекачка топлива

Операции пополнения refuel_cistern и перекачки transfer_fuel реализованы с учётом бизнес-ограничений:

1) Рисунок 4: проверка на превышение максимального объёма цистерны.

2) Рисунок 5: разрешена только между цистернами одного типа топлива; контролируются достаточность топлива в источнике и свободный объём в приёмнике.

```
def refuel_cistern(self):
    new_volume = cistern['current_volume'] + liters
    if new_volume > cistern['max_volume']:
        print(f"Превышение максимального объема! Максимум: {
    cistern['max_volumeл']}")
        return
    self.cisterns[cistern_id]['current_volume'] = new_volume
    self.log_operation('REFUEL', f"Цистерна {cistern_id},
    добавлено {litersл}")
```

Рисунок 4 – Пополнение цистерны

```
def transfer_fuel(self):
    # Проверка типа топлива
    if self.cisterns[source_id]['type'] != self.cisterns[dest_id]['type']:
        print("Ошибка: перекачка возможна только между цистернами одного типа!")
        return

    # Проверка объемов
    if liters > source_cistern['current_volume']:
        print("Недостаточно топлива в источнике!")
        return

    if dest_cistern['current_volume'] + liters > dest_cistern['max_volume']:
        print("Превышение максимального объема в приемнике!")
        return

    # Выполнение перекачки
    self.cisterns[source_id]['current_volume'] -= liters
    self.cisterns[dest_id]['current_volume'] += liters
    self.log_operation('TRANSFER', f"Из {source_id} в {dest_id}, {litersл}")
```

Рисунок 5 – Перекачка топлива

Обе операции логируются в истории.

2.5 Аварийный режим

Аварийный режим имитирует критическую ситуацию (утечка, пожар). При активации emergency_mode_handler

1) все цистерны немедленно отключаются;

2) блокируются все коммерческие операции;

3) фиксируется событие в истории;

4) выводится сообщение о вызове служб.

Код аварийного режима предаставлен на рисунке 6

```
def emergency_mode_handler(self):
    print("ВНИМАНИЕ! Все цистерны будут заблокированы!")
    confirm = input("Подтвердить аварию? (y/n): ").lower()

    if confirm == 'y':
        for cistern_id, cistern in self.cisterns.items():
            if cistern['enabled']:
                cistern['enabled'] = False
                self.log_operation('EMERGENCY_DISABLE', f"Цистерна {cistern_id} заблокирована при аварии")

        self.emergency_mode = True
        self.log_operation('EMERGENCY', "Аварийная ситуация активирована")
```

Рисунок 6 – Аварийный режим

Выход из аварийного режима возможен только вручную, при этом цистерны не включаются автоматически — требование ТЗ выполнено строго.

2.6 Сохранение данных

Все данные сохраняются в файл station_datajson в формате JSON. Структура включает:

1) состояние цистерн;

2) баланс и статистику;

3) историю операций;

4) флаг аварийного режима.

Код сохранение данных изабражен на рисунке 7

```
def save_data(self):
    data = {
        'cisterns': self.cisterns,
        'balance': self.balance,
        'stats': self.stats,
        'history': self.history,
        'emergency_mode': self.emergency_mode
    }
    with open('station_data.json', 'w', encoding='utf-8') as
    f:
        json.dump(data, f, ensure_ascii=False, indent=2)
```

Рисунок 7 – Сохранение данных

Методы save_data и load_data обеспечивают полное восстановление состояния программы после перезапуска.

3 Тестирование

В процессе разработки и после её завершения была проведена всесторонняя проверка функциональности системы. Тестирование включало следующие основные сценарии использования:

Было проведено комплексное тестирование всех функций системы. Проверялась корректность основных операций: обслуживания клиентов (полный цикл от выбора колонки до фиксации продажи), пополнения цистерн с валидацией объёмов и перекачки топлива между цистернами одного типа.

Система устойчива к некорректному вводу — обрабатывает нечисловые значения, отрицательные величины и выбор несуществующих пунктов. Аварийный режим корректно блокирует все операции, а после его отключения цистерны остаются заблокированными, как требует ТЗ.

Механизмы сохранения данных работают надёжно: состояние полностью восстанавливается после перезапуска. Автоматическое отключение цистерн при низком уровне топлива выполняется корректно. Все тесты подтвердили соответствие системы техническому заданию.

4 Заключение

Разработан прототип системы управления АЗС на Python, полностью соответствующий ТЗ. Реализованы:

1) полный цикл обслуживания клиентов;
2) управление запасами с автоматическим контролем уровня;
3) операции пополнения и перекачки топлива;
4) аварийный режим с ручным восстановлением;
5) сохранение состояния между запусками.

Архитектура на основе класса FuelStation обеспечивает связность и простоту поддержки. Код устойчив к ошибкам ввода, логирует все операции и легко расширяем.

Система готова для учебного использования и может служить основой для дальнейшего развития (веб-интерфейс, интеграция с учётными системами). Работа подтвердила возможность создания надёжных консольных систем управления на Python.