




# семестровый проект по ООП на Python: Консольная система управления автосервисом "CarFix"

**Цель проекта:** разработать полнофункциональное консольное приложение для управления автосервисом (или автомойкой, СТО, шиномонтажом) с использованием принципов ООП. Все данные хранятся в текстовых файлах. студенты должны реализовать иерархию классов, взаимодействие объектов, сохранение/загрузку состояния, обработку ошибок и удобный консольный интерфейс.

 **Срок выполнения:** 1 семестр (рекомендуется разбить на 4–5 этапов)

 **Хранение данных:** только текстовые файлы (JSON или CSV — на выбор студента, но без внешних БД)

 **Интерфейс:** полностью консольный (меню, ввод/вывод через `input()/print()`)

## Структура проекта

```
carfix/
├── main.py           # точка входа, меню
├── models.py        # все классы
├── data/            # папка с данными
│   ├── clients.json
│   ├── services.json
│   ├── appointments.json
│   └── staff.json
├── storage.py       # модуль для загрузки/сохранения
└── test_checker.py  # проверочный файл для преподавателя
```

## Классы и их методы

### 1. Класс `Client`

**Описание:** Представляет клиента автосервиса.

**Атрибуты:**

- `client_id` (int) — уникальный ID (генерируется автоматически)
- `name` (str)
- `phone` (str)
- `car_model` (str)
- `license_plate` (str) — гос. номер
- `registration_date` (str) — дата регистрации в формате "YYYY-MM-DD"

**Методы:**

- `__init__(name, phone, car_model, license_plate)`
  - `__str__()` → возвращает строку: "Клиент [ID: 1] Иван Петров, BMW X5, a123Bc"
- 

## 2. 🛠 Класс Service

**Описание:** Услуга, которую предоставляет сервис (мойка, замена масла, шиномонтаж и т.д.).

**атрибуты:**

- `service_id` (int)
- `name` (str) — например, "Полировка кузова"
- `duration_minutes` (int) — длительность в минутах
- `price` (float) — стоимость

**Методы:**

- `__init__(name, duration, price)`
  - `__str__()` → "Услуга [ID: 3] Замена масла (30 мин) – 1500 руб."
- 

## 3. 👤 Класс Staff

**Описание:** сотрудник автосервиса.

**Атрибуты:**

- `staff_id` (int)
- `name` (str)
- `position` (str) — "Мойщик", "Механик", "администратор"
- `is_available` (bool) — свободен ли сейчас

**Методы:**

- `__init__(name, position)`
  - `set_availability(status: bool)` — меняет статус
  - `__str__()` → "сотрудник [ID: 2] сергей Иванов (Механик) – свободен"
- 

## 4. 📅 Класс Appointment

**Описание:** Запись на приём — связывает клиента, услугу, сотрудника и время.

**Атрибуты:**

- `appointment_id` (int)
- `client_id` (int)
- `service_id` (int)
- `staff_id` (int)
- `date_time` (str) — формат "YYYY-MM-DD HH:MM" (например, "2025-04-10 15:30")

- `status` (str) — “запланировано”, “выполнено”, “отменено”

#### Методы:

- `__init__(client_id, service_id, staff_id, date_time)`
  - `cancel()` — меняет статус на “отменено”
  - `complete()` — меняет статус на “выполнено”
  - `__str__()` → “Запись [ID: 5] Клиент 1 → Услуга 3 → сотрудник 2, 2025-04-10 15:30 – запланировано”
- 

## 5. Класс `CarService`

**Описание:** основной класс-менеджер, управляющий всей системой.

#### Атрибуты:

- `clients: list[Client]`
- `services: list[Service]`
- `staff: list[Staff]`
- `appointments: list[Appointment]`

#### Методы:

##### ◆ Загрузка и сохранение

- `load_data()` — загружает все списки из файлов в папке `data/`
- `save_data()` — сохраняет все списки обратно в файлы

##### ◆ Управление клиентами

- `add_client(name, phone, car_model, license_plate)` → возвращает объект `Client`
- `find_client_by_phone(phone: str)` → `Client` или `None`
- `get_all_clients()` → `list[Client]`

##### ◆ Управление услугами

- `add_service(name, duration, price)` → `Service`
- `get_service_by_id(service_id: int)` → `Service` или `None`
- `get_all_services()` → `list[Service]`

##### ◆ Управление сотрудниками

- `add_staff(name, position)` → `Staff`
- `get_available_staff(position: str = None)` → `list[Staff]` (если `position` указан — только этой должности)
- `assign_staff_to_appointment(staff_id: int, appointment_id: int)` → `bool` (успех/неудача)

##### ◆ Управление записями



- `create_appointment(client_phone, service_id, date_time) → Appointment` (автоматически подбирает свободного сотрудника нужной специальности, проверяет, нет ли пересечений по времени)
- `cancel_appointment(appointment_id) → bool`
- `complete_appointment(appointment_id) → bool`
- `get_appointments_for_date(date: str) → list[Appointment]` (формат "YYYY-MM-DD")
- `get_client_appointments(client_id) → list[Appointment]`

#### ◆ Вспомогательные

- `validate_time_slot(staff_id, date_time, duration) → bool` (проверяет, свободен ли сотрудник в это время с учётом длительности услуги)
- `generate_id(data_list) → int` (генерирует уникальный ID на основе максимального существующего + 1)

## 📁 Формат хранения данных (JSON — пример)

Файлы хранятся в папке `data/` в формате JSON. Пример `clients.json`:

```
[
  {
    "client_id": 1,
    "name": "Иван Петров",
    "phone": "+79001234567",
    "car_model": "BMW X5",
    "license_plate": "a123Bc",
    "registration_date": "2025-03-01"
  },
  {
    "client_id": 2,
    "name": "Мария сидорова",
    "phone": "+79007654321",
    "car_model": "Kia Rio",
    "license_plate": "Б456Де",
    "registration_date": "2025-03-05"
  }
]
```

аналогично — `services.json`, `staff.json`, `appointments.json`.

## 🖥 Консольный интерфейс (main.py)

Программа должна запускаться через `main.py` и предоставлять меню:

```
=== Добро пожаловать в CarFix ===
1. Добавить клиента
```

2. Добавить услугу
3. Добавить сотрудника
4. Записать клиента на приём
5. отменить запись
6. Завершить приём
7. Показать записи на дату
8. Показать всех клиентов
9. Показать все услуги
0. Выход

При выборе пункта — запрашиваются необходимые данные, вызываются методы `CarService`, выводятся результаты.

🔗 Пример сценария записи:

```
Выберите действие: 4
Введите телефон клиента: +79001234567
Выберите услугу:
  1. Мойка кузова (20 мин) – 500 руб.
  2. Замена масла (30 мин) – 1500 руб.
Введите ID услуги: 2
Введите дату и время (YYYY-MM-DD HH:MM): 2025-04-10 15:30
→ Подбираем свободного механика...
→ Запись создана! ID: 5
```

## ✓ Требования к реализации

- Все классы должны быть реализованы в `models.py`.
- Логика сохранения/загрузки — в `storage.py` (можно использовать `json` модуль).
- Главное меню и логика взаимодействия — в `main.py`.
- обработка ошибок: неверный ввод, отсутствие сотрудника, пересечение времени — должна выводиться понятная ошибка, а не приводить к падению программы.
- При запуске — автоматически загружать данные из файлов.
- При выходе — сохранять изменения.
- ID генерируются автоматически и не должны повторяться.
- Время хранится как строка в едином формате.

## 🔧 Проверочный файл: `test_checker.py`

Этот файл должен **автоматически проверять работоспособность программы**. Преподаватель запускает его — и видит, всё ли работает.

## 📅 рекомендуемое распределение по этапам (на семестр)

Этап	срок	Что реализовать
------	------	-----------------

Этап	срок	Что реализовать
1	Неделя 3	Классы <code>Client</code> , <code>Service</code> , <code>Staff</code> + загрузка/сохранение в JSON
2	Неделя 6	Класс <code>Appointment</code> + методы <code>create/cancel/complete</code> + проверка времени
3	Неделя 9	Класс <code>CarService</code> с полной логикой управления + консольное меню
4	Неделя 12	Добавление валидации, обработки ошибок, красивого вывода
5	Неделя 15	Финальная сборка, тестирование, подготовка к защите + прохождение <code>test_checker.py</code>

## Критерии оценки

Любую строчку кода нужно знать и уметь объяснить, что она выполняет!

Критерий	Баллы
реализация всех классов	5
Корректная работа с файлами	5
Полноценное меню и UX	5
обработка ошибок и валидация	5
автоматическое назначение персонала и проверка времени	5
Прохождение всех тестов из <code>test_checker.py</code>	5
<b>Итого</b>	<b>36</b>