План рефакторинга для проекта `PythonCode`

> Цель: вывести проект в устойчивое, модульное и тестируемое состояние, минимизировав риски для пользователей и ускорив дальнейшую разработку.

Краткий обзор целей

1. Разделить ответственность: UI (QML) ↔ Business logic ↔ Data access.
2. Перенести работу с БД в набор репозиториев; сократить `database.py` до менеджера соединения.
3. Ввести сервисный слой для бизнес-правил и единый `FileService` для работы с файлами.
4. Ввести тестируемость: pytest + модульные/интеграционные тесты для репозиториев и сервисов.
5. Наладить CI: lint, форматирование, тесты, security scan, dependency management.
6. Убрать артефакты IDE, закрепить зависимости, добавить LICENSE и README.md.

Высокоуровневый план (эпики)

Эпик A — Repo hygiene & infra (P0) - Add LICENSE, README.md (Quick Start) - .gitignore, удалить .idea и артефакты - Зафиксировать зависимости (requirements.txt или pyproject/poetry) - Настроить basic CI (lint + tests)

Эпик B — Разделение данных и репозиториев (P0) - Создать пакет `src/data` и `src/data/repositories` - Вынести connection-менеджер и транзакции - Перенести SQL-запросы в репозитории

Эпик C — Сервисный слой и Domain (P0-P1) - Добавить `src/services` и `src/domain/entities` (dataclasses) - Перенести бизнес-логику из моделей в сервисы - Ввести `FileService`, `SpecificationService`, `ItemService`

Эпик D — UI thin layer (P1) - Привести QML-модели к роли только представления - Внедрить DI/инверсию зависимостей (модели получают сервисы через параметры)

Эпик E — Tests & QA (P0-P1) - Добавить pytest, написать тесты для репозиториев и сервисов - Покрыть критические сценарии (CRUD, transactions, file ops)

Эпик F — Observability & Error handling (P1) - Логирование, уровни логов, централизованные исключения (AppError) - Документация по ошибкам и recovery steps

---

Детализированный план задач (с зависимостями и приоритетами)

1. Repo hygiene & initial infra (2–3 дня)

Задачи: - 1.1 Добавить `LICENSE` (MIT/Apache) — owner: repo-admin — 0.5d — AC: LICENSE в корне. - 1.2 Переименовать `Readme.txt` → `README.md` с Quick Start (run, deps, db init) — owner: dev — 0.5d — AC: runnable инструкцию можно выполнить. - 1.3 Добавить/обновить `.gitignore` и удалить `.idea`, `__pycache__` — owner: dev — 0.25d — AC: артефакты не коммитятся. - 1.4 Зафиксировать зависимости: преобразовать `requirements.txt` в pinned-формат или подготовить `pyproject.toml + poetry.lock` — owner: dev — 0.5–1d — AC: `pip install -r requirements.txt` восстанавливает окружение. - 1.5 Добавить pre-commit (black, ruff/flake8, isort) — owner: dev — 0.5d — AC: коммиты автоматически форматируются.

Зависимости: нет — можно начать сразу.

2. CI: basic pipeline (1–2 дня)

Задачи: - 2.1 Создать `.github/workflows/ci.yml`: checkout, set up python, install deps, run black --check, ruff, pytest. — owner: devops — 1d — AC: PRs показывают проверку, pipeline зелёный при корректных кодах. - 2.2 Включить Dependabot/Dependecy graph (opt) — owner: devops — 0.25d — AC: pull requests для обновлений deps.

Зависимости: 1.4 (фиксированные зависимости)

3. Data layer refactor — Connection & Repositories (3–7 дней)

Задачи: - 3.1 Создать `src/data/__init__.py`, `src/data/connection.py` — контекстный менеджер для sqlite (session/transaction). — owner: backend — 0.5d — AC: есть contextmanager с commit/rollback. - 3.2 Создать `src/data/repositories/base.py` с базовым CRUD-шаблоном и helper-методами (map row→entity). — owner: backend — 0.5d — AC: репозитории используют base. - 3.3 Мигрировать функционал из `database.py` в отдельные репозитории: `items_repo.py`, `suppliers_repo.py`, `specs_repo.py`, `categories_repo.py`. — owner: backend — 2–5d — AC: старые unit-тесты (если есть) проходят, код вызывается через новые репозитории. - 3.4 Написать интеграционные тесты для репозиториев (sqlite in-memory). — owner: backend — 1–2d — AC: pytest запускается и проверяет CRUD.

Зависимости: 1.4, 1.3

4. Domain & Service layer (4–8 дней)

Задачи: - 4.1 Добавить `src/domain/entities.py` (dataclasses для Item, Supplier, Specification, FileMeta). — owner: arch — 0.5d — AC: сущности описаны и используются репозиториями. - 4.2 Создать `src/services/*` — сервисы, оперирующие репозиториями: `ItemService`, `SpecificationService`, `FileService`. — owner: backend — 2–4d — AC: бизнес-логика вынесена из моделей. - 4.3 Переписать основные операции: создание/удаление спецификаций, связывание поставщиков, вычисление итоговых цен — owner: backend — 1–2d — AC: поведение совпадает с прежним при ручной проверке. - 4.4 Тесты для сервисов (pytest, моки репозиториев) — owner: backend — 1–2d — AC: тесты покрывают критичные сценарии.

Зависимости: 3.*

5. UI models → thin QML bridge (3–6 дней)

Задачи: - 5.1 Обновить QAbstract*Model классы так, чтобы они принимали готовые сервисы/репозитории (DI). — owner: frontend — 1–2d — AC: модели вызывают сервисы, не SQL. - 5.2 Удалить из моделей прямые вызовы SQL и преобразовать их в адаптеры для UI. — owner: frontend — 1–3d — AC: бизнес-логика отсутствует в моделях. - 5.3 Обновить QML-bindings при необходимости и проверка ручного сценария запуска UI. — owner: frontend — 1d — AC: UI функционирует как раньше.

Зависимости: 3,4

6. Tests coverage & Additional QA (ongoing)

Задачи: - 6.1 Достигнуть базового покрытия (30–50%) для core-modules — owner: QA — 3–7d - 6.2 Настроить покрытие в CI и fail-on-low-coverage (опционально) — owner: devops — 1d

Зависимости: 2,3,4

Миграционная стратегия (как внедрять без больших регрессий)

1. **Feature branches по эпикам** — каждый эпик = отдельная ветка.
2. **Инкрементные изменения** — не рефакторить всё сразу. Пример веток:
3. `infra/hygiene` (эпик A)
4. `data/repo-refactor` (эпик B)
5. `services/specification` (эпик C subset)
6. `ui/thin-models` (эпик D)
7. **Compatibility layer** — временно оставить «адаптеры», которые переводят старый API вызовов в новый репозиторный API.
8. **Integration tests** покрывают legacy flows — прогонять CI на каждом PR.
9. **Фаза переключения**: когда все сервисы и repo готовы, переключить UI на новые классы и удалить compatibility layer.

Acceptance criteria (общие)

- Все PR проходят CI (lint + pytest).
- Критические user flows работают: запуск приложения, CRUD для items/specifications, сохранение/загрузка файлов.
- Нет артефактов среды разработки в репо.
- Зависимости зафиксированы.
- Наличие README.md с Quick Start и описанием архитектуры.

Risks & Mitigations

- **Риск:** Рефакторинг database.py сломает сложные бизнес-правила.
- **Митигация:** покрыть ключевые сценарии интеграционными тестами (in-memory sqlite) до переключения.
- **Риск:** UI-баги после отделения логики.
- **Митигация:** использовать compatibility adapters и feature flags.
- **Риск:** Зависимости (PySide/PyQt) ведут себя иначе на CI.
- **Митигация:** тестировать репозитории и сервисы отдельно от UI (headless). UI smoke-test запускать вручную или в отдельной job с xvfb при необходимости.

Примеры изменений: шаблоны файлов

src/data/connection.py (контекстный менеджер для sqlite)

```
from contextlib import contextmanager
import sqlite3

class DBConnection:
    def __init__(self, path: str):
        self.path = path

    @contextmanager
    def session(self):
        conn = sqlite3.connect(self.path)
        try:
            conn.execute('PRAGMA foreign_keys = ON')
            yield conn
            conn.commit()
        except Exception:
            conn.rollback()
            raise
        finally:
            conn.close()
```

src/domain/entities.py (пример dataclass)

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class Item:
    id: Optional[int]
    name: str
    article: str
    price: float
    measure: str
```

src/data/repositories/items_repo.py (шаблон)

```
from typing import List
from ..connection import DBConnection
from ...domain.entities import Item

class ItemsRepository:
    def __init__(self, conn: DBConnection):
        self._conn = conn

    def get_by_id(self, item_id: int) -> Item:
        with self._conn.session() as c:
            row = c.execute('SELECT id, name, article, price, measure FROM
items WHERE id=?', (item_id,)).fetchone()
            if not row:
                return None
            return Item(*row)

    # другие CRUD
```

Оценка по времени и ресурсу

- Минимальный набор (эпик A + B + CI + 10 модульных тестов): **~2 нед. (1 dev, 0.5 devops)**
- Полный первичный рефакторинг (A–E + тесты, coverage ~40%): **~4–6 нед. (1–2 dev, 0.5 devops, 0.5 QA)**

Что я могу сделать прямо сейчас

- Сгенерировать skeleton репозитория с новой структурой и примерами файлов (по ветке `refactor/skeleton`).
- Выполнить автоматическую миграцию части `database.py` → `src/data/repositories/items_repo.py` и вернуть PR-патч.
- Настроить `ci.yml` и `pre-commit` конфиги.

Если устраивает — я сейчас автоматически **создам skeleton** и заполню первые файлы (connection, items_repo, domain/entities, basic CI). Дай подтверждение — и я инициирую генерацию файлов/PR прямо в репо (или подготовлю архив/патч для загрузки).