Конструирование программного обеспечения

Домашняя работа №2. Паттерны проектирования.

Формат сдачи предлагается семинаристом. По умолчанию работа выполняется на языке программирования, который является основным на семинарских занятиях, но по согласованию с семинаристом можно выполнить на другом ЯП.

Предисловие

Многие компании активно развивают собственные финтех-направления, а некоторые и вовсе открывают свои банки. НИУ ВШЭ не собирается оставаться в стороне!

ВШЭ-Банк в своем будущем приложении сделает акцент на модуле финансового учета. Этот раздел будет включать инструменты, упрощающие управление личными финансами: отслеживание доходов и расходов, анализ трат и планирование бюджета. Такой подход поможет пользователям более осознанно распоряжаться средствами и достигать своих финансовых целей.

Перед вами поставлена задача *разработать классы* доменной модели ключевого модуля будущего приложения – *модуль «Учет финансов». Доменная модель классов должна быть реализована* с соблюдением принципов SOLID, ключевыми идеями GRASP: High Cohesion и Low Coupling, а также рядом паттернов GoF: порождающих, структурных и поведенческих.

В ходе общения с заказчиком были выявлены:

Основные требования к функциональности модуля «Учет финансов»

1. Работа с доменной моделью: создание, редактирование и удаление счетов, категорий, операций (доходов/расходов).

Опциональная функциональность модуля «Учет финансов». В каждом пункте можно пропустить/модифицировать любые подпункты, а также добавить свои, если это поможет вам уместнее применять паттерны.

1. **Аналитика**:
    a. Подсчет разницы доходов и расходов за выбранный период.
    b. Группировка доходов и расходов по категориям.
    c. Любая другая аналитика.
2. **Импорт и экспорт данных**:
    a. Экспорт всех данных в файлы: CSV, YAML, JSON.
    b. Импорт данных из файлов: CSV, YAML, JSON.
3. **Управление данными**:
    a. Возможность пересчета баланса в случае выявленного несоответствия (предусмотреть процедуры автоматический или ручной пересчет баланса).
4. **Статистика**:
    a. Измерение времени работы отдельных пользовательских сценариев.

На встрече с доменными экспертами были определены три основных класса: BankAccount, Category и Operation:

1. **BankAccount**:
   - id: уникальный идентификатор счета.
   - name: название счета (например, "Основной счет").
   - balance: текущий баланс счета.
2. **Category**:
   - id: уникальный идентификатор категории.
   - type: тип категории (доход или расход).
   - name: название категории (например, "Кафе", "Зарплата").

   Примеры категорий: "Кафе" или "Здоровье" – для расходов. "Зарплата" или "Кэшбэк" – для доходов
3. **Operation**:
   - id: уникальный идентификатор операции.
   - type: тип операции (доход или расход).
   - bank_account_id: ссылка на счет, к которому относится операция.
   - amount: сумма операции.
   - date: дата операции.
   - description: описание операции (необязательное поле).
   - category_id: ссылка на категорию, к которой относится операция.

На встрече с командой вы продумали и пришли к выводу, что будут уместны следующие паттерны (эти идеи можно проигнорировать и реализовать свои):

1. **Фасад** – поможет вам объединить по смыслу несколько методов. Например, можно упаковать в отдельный фасад всю работу с Category (создание, изменение, получение, удаление). Также отдельные фасады для BankAccount, Operation. Всю работу с аналитикой тоже можно вынести в отдельный фасад.
2. **Команда** + 3. **Декоратор** - вы сможете каждый пользовательский сценарий представить в виде паттерна Команда. Это позволит вам проще применить над каждой из команд паттерн Декоратор, для измерения времени работы пользовательского сценария. Паттерн декоратор в свою очередь позволит вам написать измерение времени работы пользовательского сценария один раз, а дальше просто оборачивать в этот декоратор сами сценарии - Команды.
4. **Шаблонный метод** – пригодится для импорта данных из файлов различных форматов. В сценарии загрузки из файла отличается только часть парсинга данных - из json / yaml / csv, а остальная часть будет одинаковой (когда данные из файла уже прочитаны).
5. **Посетитель** – подойдет при выгрузке данных в файл.
6. **Фабрика** – подойдет для того, чтобы гарантировать, что все доменные объекты создаются в одном и том же месте кода. Это в свою очередь поможет вам поддержать валидацию этих объектов - например запретить создание Operation с отрицательным amount. Если создание объектов в вашем коде через new происходит в нескольких местах, то в каждом из этих мест вам нужно валидировать ваш свежесозданный объект - происходит дублирование кода и со

временем вы забудете внести очередные изменения в валидацию в одном из этих мест, тем самым допустив создание невалидных доменных объектов.

7. **Прокси** - если вы решите использовать базу данных для персистентности данных после перезапуска приложения, вы можете использовать паттерн Прокси для реализации in-memory кэша. При инициализации приложения вы выгружаете все данные в кэш и дальше при чтении данных читаете их оттуда. При записи данных вы пишете их в кэш и БД.

8. …

ТРЕБУЕТСЯ

1. Разработать классы доменной модели модуля «Учет финансов». Доменная модель классов должна быть реализована с соблюдением принципов SOLID и ключевыми идеями GRASP: High Cohesion / Low Coupling, а также рядом паттернов GoF: порождающих, структурных и поведенческих (их варианты описаны выше).
2. Создать консольное приложение, в котором продемонстрировать функциональность разработанной доменной модели.
3. Написать отчет*, в котором отразить:
    a. Общую идею вашего решения (какой функционал реализовали, особенно если вносили изменения в функциональные требования).
    b. Опишите какие принципы из SOLID и GRASP вы реализовали, скажите в каких классах (модулях).
    c. Опишите какие паттерны GoF вы реализовали, обоснуйте их важность, скажите в каких классах (модулях) они реализованы.

* Отчет пишется в свободной форме (например, redme-файл к проекту в формате md). Задача отчета – помочь проверяющему увидеть всё, что вы реализовали.

4. Добавить инструкция по запуску вашего приложения.

Критерии оценки

- **+ 3 балла** за полную реализацию основных требований к функциональности
- **+ 0.5 балла** за каждый реализованный паттерн (max 4 балла).
- **+ 2 балла** за соблюдение принципов SOLID и GRASP
- **+ 1 балл** за использование DI-контейнера

Штрафы

1. до – 2 баллов за наличие ошибки во время выполнения кода;
2. до – 5 баллов, если программа не собирается;
3. – 1 балл за каждый день просрочки дедлайна
4. – 1 балл за грубое игнорирование кодстайла.