



Labs

▼ Лабораторная 0. Isu

Цель.

Ознакомиться с языком C#, базовыми механизмами ООП. В шаблонном репозитории описаны базовые сущности, требуется реализовать недостающие методы и написать тесты, которые бы проверили корректность работы.

Предметная область.

В данной лабораторной используется только часть Информационной системы университета (ИСУ), отвечающая за хранение структурированной, актуальной информации о статусе студента, то есть за какой учебной группой, курсом, факультетом закреплен студент. Также добавим сюда функциональность переводов студентов между группами.

Функциональность.

- Хранение актуальной информации о студенте.
- Хранение актуальной информации о группе и ее составе.
- Реализация возможностей студента, учащегося в группе.

Техническое задание.

Перед написанием любой программы, приложения, сервиса, системы необходимо составить техническое задание. Что это вообще такое? Грубо говоря, перевод полета фантазии заказчика на сухой язык фактов и ограничений в реализации.

Давайте начнем с уточняющих вопросов.

- Что является актуальной информацией о студенте?
- Что является актуальной информацией о группе?
- Какие возможности имеет студент?
- Каким типом данных будут табельные номера студентов и уникальные идентификаторы групп?
- Мы делаем систему для всего университета или только для конкретного факультета?
- Какой формат названия у групп?
- Есть ли ограничение на количество студентов в группе?
- Студент может быть одновременно в нескольких группах?
- При переводе между группами как нужно решить вопрос единственности группы у студента?

Начинать выполнение лабораторной лучше после получения ответов на вопросы выше)

Также необходимо реализовать следующий интерфейс (находящийся в проекте Isu шаблонного репозитория), который описывает выполняемые студентами действия:

```
public interface IIsuService { Group AddGroup(GroupName name); Student AddStudent(Group group, string name); Student GetStudent(int id); Student FindStudent(int id); List<Student> FindStudents(GroupName groupName); List<Student> FindStudents(CourseNumber courseNumber); Group FindGroup(GroupName groupName); List<Group> FindGroups(CourseNumber courseNumber); void ChangeStudentGroup(Student student, Group newGroup); }
```

И протестировать написанный код:

```
[Test] public void AddStudentToGroup_StudentHasGroupAndGroupContainsStudent() { } [Test] public void ReachMaxStudentPerGroup_ThrowException() { } [Test] public void CreateGroupWithInvalidName_ThrowException() { } [Test] public void TransferStudentToAnotherGroup_GroupChanged() { }
```

FAQ

Нужно использовать `GroupName groupName` или `string groupName`? Можно использовать любой вариант. Первый предпочтительней.

▼ Лабораторная 1. Shops

Цель

Продемонстрировать умение выделять сущности и проектировать по ним классы.

Прикладная область

Магазин, покупатель, доставка, пополнение и покупка товаров. Магазин имеет уникальный идентификатор, название (не обязательно уникальное) и адрес. В каждом магазине установлена своя цена на товар и есть в наличии некоторое количество единиц товара (какого-то товара может и не быть вовсе). Покупатель может производить покупку. Во время покупки - он передает нужную сумму денег магазину. Поставка товаров представляет собой набор товаров, их цен и количества, которые должны быть добавлены в магазин.

Тест кейсы

- Поставка товаров в магазин. Создаётся магазин, добавляются в систему товары, происходит поставка товаров в магазин. После добавления товары можно купить.
- Установка и изменение цен на какой-то товар в магазине.
- Поиск магазина, в котором набор товаров можно купить максимально дешево. Обработать ситуации, когда товара может быть недостаточно или товаров может не быть нигде.
- Покупка партии товаров в магазине (набор пар товар + количество). Нужно убедиться, что товаров хватает, что у пользователя достаточно денег. После покупки должны передаваться деньги, а количество товаров измениться.

NB:

Можно не поддерживать разные цены для одного магазина. Как вариант, можно брать старую цену, если магазин уже содержит этот товар. Иначе брать цену указанную в поставке.

Пример ожидаемого формата тестов представлен ниже.

Используемые в тестах API магазина/менеджера/etc не являются интерфейсом для реализации в данной лабораторной. Не нужно ему следовать 1 в 1, это просто пример.

```
public void SomeTest(moneyBefore, productPrice, productCount,
productToBuyCount) { var person = new Person("name", moneyBefore); var
shopManager = new ShopManager(); var shop = shopManager.Create("shop name",
...); var product = shopManager.RegisterProduct("product name");
shop.AddProducts( ... ); shop.Buy(person, ...); Assert.AreEqual(moneyBefore -
productPrice * productToBuyCount, person.Money); Assert.AreEqual(productCount -
productToBuyCount , shop.GetProductInfo(product).Count); }
```

Цель

Научиться выделять зоны ответственности разных сущностей и проектировать связи между ними.

!! Реализация данной лабы должна переиспользовать логику Lab 0. Так же она должна поддерживать с ней обратную совместимость, что значит, что использование `IIsuService` отдельно от сервиса данной лабораторной не должно нарушать инвариант её логики.

Предметная область

Реализация системы записи студентов на ОГНП.

Курс ОГНП – дополнительные занятия, которые могут изучать студенты. Каждый курс реализуется определенным мегафакультетом. Курс изучается в несколько потоков с ограниченным количеством мест. У каждого потока есть свое расписание – список пар, которые проводятся в течение недели.

Пара – описание временного интервала, в который группа занимается. Пара должна быть ассоциирована с группой, временем, преподавателем и аудиторией.

Студенты могут записываться на два разных курса ОГНП. Студент не может записаться на ОГНП, которое представляет мегафакультет его учебной группы. Каждая учебная группа принадлежит определенному мегафакультету, который определяется из названия группы, например М3105 – ИТИП(ТИНТ), К3104 – СУИР(КТУ). Каждая учебная группа имеет список пар. При записи студента должна быть проверка на то, что пары его учебной группы не пересекаются с парами потока ОГНП.

Требуемый функционал

- Добавление нового курса ОГНП
- Запись студента на определенный ОГНП
- Возможность снять запись
- Получение потоков по курсу
- Получение списка студентов в определенной группе ОГНП
- Получение списка не записавшихся на курсы студентов по группе

▼ Лабораторная 3. Backups

Цель:

Применить на практике принципы из SOLID, GRASP.

Нотация

Concept Name – наименование концепции

Concept Name – тоже наименование концепции, но у меня закончились цвета текста

Concept Implementation Name – наименование реализации концепции

Предметная область

Backup Object – объект, который отслеживается системой для создания резервных копий. Объекты могут быть как файлами, так и папками.

Restore Point – снапшот набора отслеживаемых объектов. Должна как минимум хранить дату создания и коллекцию **Backup Object**, отслеживаемых на момент создания **Restore Point**.

Backup – в контексте данной системы, **Backup**, это коллекция **Restore Point**, то есть, полная история резервного копирования в рамках одной **Backup Task**.

Backup Task – сущность хранящая конфигурацию резервного копирования (текущий список **Backup Object**, способов хранения, способов сжатия) и информацию о созданных **Restore Point**. Так же **Backup Task** должна инкапсулировать логику своего выполнения, т.е. создания новых **Restore Point**.

Repository – абстракция над хранением и записью данных, будь то данные объектов, соответствующих **Backup Object**, либо данные о каком-либо **Storage**. Простейшая реализация репозитория в рамках данной лабораторной – абстракция над чтением/записью в некоторую директорию на локальной файловой системе.

Storage – файл, в котором хранится копия данных, соответствующих какому-либо **Backup Object**, созданная в конкретной **Restore Point**.

Пример логики работы

Выполняем следующие действия:

1. Имеем в **Repository** три объекта **File A**, **File B**, **Folder C**.
2. Создаём **Backup Task**, добавляем в неё три **Backup Object**, соответствующие объектам находящимся в репозитории.
3. Запускаем выполнение **Backup Task**, получаем **Restore Point**, он записывается в репозиторий, в соответствующей директории появляются **Storage** **File A(1)**, **File B(1)**, **Folder C(1)**.
4. Запускаем выполнение ещё раз, получаем **Storage** с версиями **(2)**.
5. Удаляем из **Backup Task** **File B**, запускаем выполнение ещё раз, получаем третий **Restore Point**, ему будут соответствовать два **Storage** – **File A(3)**, **Folder C(3)**.

Создание резервных копий

Под созданием резервной копии данных, подразумевается создание копии данных в другом месте. Система должна поддерживать расширяемость в вопросе выбора **Storage Algorithm**, используемых для хранения резервных копий (должна иметь возможность добавить новый алгоритм безболезненно, помним про OCP).

В данной лабораторной требуется реализовать два **Storage Algorithm**:

1. **Split Storage** – алгоритм раздельного хранения, для каждого **Backup Object** в **Restore Point** создаётся отдельный **Storage** – архив, в котором лежат данные объекта.
2. **Single Storage** – алгоритм общего хранения, для всех **Backup Object** в **Restore Point** создаётся один общий **Storage** – архив, в котором лежат данные каждого объекта.

Storage Algorithm не должен нести ответственность за реализацию архивации.

Хранение копий

В лабораторной работе подразумевается, что резервные копии будут создаваться локально на файловой системе. Но логика выполнения должна абстрагироваться от этого, должна быть введена абстракция – репозиторий (см. принцип DIP из SOLID).

В тестах стоит реализовать хранение в памяти (как вариант – [InMemoryRepository](#) с использованием паттерна Composite), так как при запуске тестов на настоящей файловой системе будет генерироваться много мусорных данных, а так же системы CI не дружат с запросами к файловой системе во время автоматического выполнения тестов.

Ожидаемая структура:

- Корневая директория
 - Директории различных **Backup Task**
 - Директории различных **Restore Point**
 - Файлы **Storage**

Создание Restore Point

Backup Task отвечает за создание новых точек восстановления, выступает фасадом, инкапсулируя логику выполнения этой операции.

При создании **Backup Task** должна быть возможность указать её название, **Repository** для хранения **Backup** (его данных), **Storage Algorithm**.

Backup Task должна поддерживать операции добавления и удаления отслеживаемых ей **Backup Object**.

Результатом выполнения **Backup Task** является создание **Restore Point** и соответствующих ей **Storage** в выбранном **Repository**.

Тест кейсы

1. Тест 1
 - a. Создаём **Backup Task**, использующую [Split Storage](#)
 - b. Добавляем в **Backup Task** два **Backup Object**
 - c. Запускаем выполнение **Backup Task**
 - d. Удаляем из **Backup Task** один **Backup Object**
 - e. Запускаем выполнение **Backup Task**
 - f. Проверяем то, что было создано две **Restore Point** и три **Storage**
2. Тест 2 (лучше оформить в виде консольного приложения, так как нормально проверить можно только на настоящей файловой системе)
 - a. Создаём **Backup Task**, использующую [FileSystemRepository](#) и [Single Storage](#)
 - b. Добавляем в **Backup Task** два **Backup Object**
 - c. Запускаем выполнение **Backup Task**
 - d. Проверяем то, что директории и файлы были созданы

▼ Лабораторная 4. Banks

Цель

Применить на практике принципы из SOLID, GRASP, паттерны GoF.

Предметная область

Есть несколько **Банков**, которые предоставляют финансовые услуги по операциям с деньгами.

В банке есть **Счета и Клиенты**. У клиента есть имя, фамилия, адрес и номер паспорта (имя и фамилия обязательны, остальное – опционально).

Счета и проценты

Счета бывают трёх видов: **Дебетовый счет**, **Депозит** и **Кредитный счет**. Каждый счет принадлежит какому-то клиенту.

Дебетовый счет – обычный счет с фиксированным процентом на остаток. Деньги можно снимать в любой момент, в минус уходить нельзя. Комиссий нет.

Депозитный счет – счет, с которого нельзя снимать и переводить деньги до тех пор, пока не закончится его срок (пополнять можно). Процент на остаток зависит от изначальной суммы, **например**, если открываем депозит до 50 000 р. - 3%, если от 50 000 р. до 100 000 р. - 3.5%, больше 100 000 р. - 4%. Комиссий нет. Проценты должны задаваться для каждого банка свои.

Кредитный счет – имеет кредитный лимит, в рамках которого можно уходить в минус (в плюс тоже можно). Процента на остаток нет. Есть фиксированная комиссия за использование, если клиент в минусе.

Комиссии

Периодически банки проводят операции по выплате процентов и вычету комиссии. Это значит, что **нужен механизм ускорения времени**, чтобы посмотреть, что будет через день/месяц/год и т.п.

Процент на остаток начисляется ежедневно от текущей суммы в этот день, но выплачивается раз в месяц (и для дебетовой карты и для депозита). Например, 3.65% годовых. Значит в день: $3.65\% / 365 \text{ дней} = 0.01\%$. У клиента сегодня 100 000 р. на счету – запомнили, что у него уже 10 р. Завтра ему пришла ЗП и стало 200 000 р. За этот день ему добавили ещё 20 р. На следующий день он купил себе новый ПК и у него осталось 50 000 р. – добавили 5 р. Таким образом, к концу месяца складываем все, что запоминали. Допустим, вышло 300 р. – эта сумма добавляется к счету или депозиту в текущем месяце.

Разные банки предлагают разные условия. В каждом банке известны величины процентов и комиссий.

Центральный банк

Регистрацией всех банков, а также взаимодействием между банками занимается центральный банк. Он должен управлять банками (предоставлять возможность создать банк) и предоставлять необходимый функционал, чтобы банки могли взаимодействовать с другими банками (например, можно реализовать переводы между банками через него). Он также занимается уведомлением других банков о том, что нужно начислять остаток или комиссию – для этого механизма не требуется создавать таймеры и завязываться на реальное время.

Операции и транзакции

Каждый счет должен предоставлять механизм **снятия, пополнения и перевода** денег (то есть счетам нужны некоторые идентификаторы).

Еще обязательный механизм, который должны иметь банки - **отмена транзакций**. Если вдруг выяснится, что транзакция была совершена злоумышленником, то такая транзакция должна быть отменена. Отмена транзакции подразумевает возвращение банком суммы обратно. Транзакция не может быть повторно отменена.

Счёта должны хранить в себе историю совершённых над ними транзакций.

Создание клиента и счета

Клиент должен создаваться по шагам. Сначала он указывает имя и фамилию (обязательно), затем адрес (можно пропустить и не указывать), затем паспортные данные (можно пропустить и не указывать).

Если при создании счета у клиента не указаны адрес или номер паспорта, мы объявляем такой счет (любого типа) сомнительным, и запрещаем операции снятия и перевода выше определенной суммы (у каждого банка своё значение). Если в дальнейшем клиент указывает всю необходимую информацию о себе - счет перестает быть сомнительным и может использоваться без ограничений.

Обновление условий счетов

Для банков требуется реализовать методы изменений процентов и лимитов на перевод. Также требуется реализовать возможность пользователям подписываться на информацию о таких изменениях - банк должен предоставлять возможность клиенту подписать на уведомления. Стоит продумать расширяемую систему, в которой могут появиться разные способы получения нотификаций клиентом (да, да, это референс на тот самый сайт). Например, когда происходит изменение лимита для кредитных карт - все пользователи, которые подписались и имеют кредитные карты, должны получить уведомление.

Консольный интерфейс работы

Для взаимодействия с банком требуется реализовать консольный интерфейс, который будет взаимодействовать с логикой приложения, отправлять и получать данные, отображать нужную информацию и предоставлять интерфейс для ввода информации пользователем.

Дополнения

- На усмотрение студента можно ввести свои дополнительные идентификаторы для пользователей, банков etc.
- На усмотрение студента можно пользователю добавить номер телефона или другие характеристики, если есть понимание зачем это нужно.

QnA

Q: Нужно ли предоставлять механизм отписки от информации об изменениях в условиях счетов A: Не оговорено, значит на ваше усмотрение (это вообще не критичный момент судя по условию лабы)

Q: Транзакциями считаются все действия со счётом, или только переводы между счетами. Если 1, то как-то странно поддерживать отмену операции снятия, а то после отмены деньги удвоются: они будут и у злоумышленника на руках и на счету. Или просто на это забить A: Все операции со счетами - транзакции.

Q: Фиксированная комиссия за использование кредитного счёта, когда тот в минусе измеряется в % или рублях, и когда её начислять: после выполнения транзакции, или до. И нужно ли при отмене транзакции убирать и начисленную за неё комиссию. А: Фиксированная комиссия означает, что это фиксированная сумма, а не процент. Да, при отмене транзакции стоит учитывать то, что могла быть также комиссия.

Q: Если транзакция подразумевает возвращение суммы обратно - но при этом эта же сумма была переведена на несколько счетов (например: перевод денег со счета 1 на счёт 2, со счёта 2 на счёт 3) Что происходит если клиент 1 отменяет транзакцию?

Подразумевается ли что деньги по цепочке снимаются со счёта 3? (на счету 2 их уже физически нет) Либо у нас банк мошеннический и деньги "отмываются" и возмещаются клиенту 1 с уводом счёта 2 в минус А: Банк не мошеннический, просто упрощённая система. Транзакции не связываются между собой. Так что да, можно считать, что может уйти в минус.