Лабораторная работа №1 по дисциплине Объектно-ориентированное программирование для студентов специальности “Информатика и технологии программирования”

Краткие теоретические сведения

Под ***предметной областью*** обычно понимают часть реального мира, которую описываю в программе.
При создании сложных программ или программных комплексов выполняется декомпозиция предметной области. При декомпозиции предметная область представляется как система.

Декомпозиция — это выявление структуры и организации системы. Система при декомпозиции представляется как нечто целостное, и вместе с тем составленная из простых компонентов.
Различают алгоритмическую (функциональную) и объектно-ориентированную декомпозиции.

Алгоритмическая декомпозиция основывается на разбиении системы на действия — алгоритмы. Система делится на подсистемы, которые делятся на более мелкие подсистемы и т.д. Конечными элементами такого деления являются процедуры и функции. Иллюстрацией такой декомпозиции служит функциональная схема.

Объектно-ориентированная декомпозиция обеспечивает разбиение на взаимодействующие объекты некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи.
В такой системе каждый объект, получив в процессе решения задачи некоторое входное воздействие (*сообщение),* выполняет заранее определенные действия. Например, он может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и в свою очередь воздействовать на другие элементы. Передавая сообщения от элемента к элементу, система выполняет необходимые
действия. При объектно-ориентированном проектировании определяются абстракции и механизмы, обеспечивающие правильное поведение модели. Шклаер и Меллор выделили следующих кандидатов на роль объектов:

- Материальные предметы - Датчики, автомобили, автоматы
- Роли - Учитель, контролер, отец, подчиненный
- События - Требование, запрос, прерывание
- Взаимодействие - Собрание, пресечение, заем.

Декомпозиция вообще и в частности объектная декомпозиция — работа интеллектуальная и лучший способ ее ведения — последовательный итеративный процесс. Сначала из возможных кандидатов строится приближенная модель системы. Затем по мере получения новых знаний

модель уточняется, вводятся новые объекты, исключаются ненужные, объекты объединяются или разбиваются.

В рамках данной лабораторной работы вам будет необходимо произвести алгоритмическую и объектную декомпозиции Банковской системы.

Отношения между классами

Рассмотрим сходства и различия классов для следующих объектов: цветы, маргаритки, красные розы, желтые розы и лепестки. Сделаем следующие выводы:

− Маргаритка — вид цветка.
− Роза — (другой) вид цветка.
− Красная и желтая розы — разновидность розы.
− Лепесток является частью обоих видов цветов.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. Наоборот, структура классов для конкретной области формируется на основе ключевых абстракций этой области и их связей. Отношение между двумя классами следует рассмотреть по двум причинам. Во-первых, отношения классов могут указывать на какой-либо вид общности. Например, маргаритки и розы являются разновидностями цветов, имеют яркую окраску лепестков, сильный аромат и т.д. Во вторых, отношения классов могут влиять на семантику связи между ними. Можно отметить, что между красными и желтыми розами больше сходства, чем между розами и маргаритками, а между маргаритками и розами больше, чем между лепестками и цветами.

Известно три основных типа отношений между классами. Первый тип называется отношением «разновидность» и отражает степень общности. Например, фраза «роза является разновидностью цветов» означает, что розе является более специализированным подклассом класса цветов. Второй тип отражает агрегатирование объектов и называется отношением «составная часть». Так, например, лепесток — не разновидность цветка, а его составная часть. Третий тип обозначает отношение ассоциативности, т.е. смысловую связь между классами, которые не связаны никакими другими типами отношений. Примером могут служить два достаточно независимых класс роз и маргариток, которые соответствуют объектам, пригодным для декоративного оформления обеденного стола.

Для покрытия основных отношений большинство объектно-ориентированных языков программирования поддерживает следующие отношения:

− ассоциация;
− наследование;
− агрегация;
− зависимость;
− конкретизация.

Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек.

Наследование — наиболее популярная разновидность отношения *обобщение специализация.* Альтернативой наследованию считается делегирование. При делегировании объекты *делегируют* свое поведение родственным объектам. При этом классы становятся не нужны.

Агрегация обеспечивает отношения *целое-часть,* объявляемые для экземпляров классов.

Зависимость часто представляется в виде частной формы — *использования,* которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Конкретизация выражает другую разновидность отношения *обобщение специализация.*

Ассоциации классов

Ассоциация обозначает семантическое соединение классов.
Пример: в системе обслуживания читателей имеются две ключевые абстракции — Книга и Библиотека. Класс Книга играет роль элемента, хранимого в библиотеке. Класс Библиотека играет роль хранилища для книг.

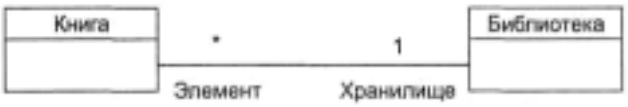


Рис. 1. Ассоциация.

Отношение ассоциации между классами изображено на рис.1. Очевидно, что ассоциация предполагает двухсторонние отношения:
− для данного экземпляра Книги выделяется экземпляр Библиотеки, обеспечивающий ее хранение;
− для данного экземпляра Библиотеки выделяются все хранимые Книги.

Здесь показана ассоциация *один-ко-многим.* Каждый экземпляр Книги имеет указатель на экземпляр Библиотеки. Каждый экземпляр Библиотеки имеет набор указателей на несколько экземпляров Книги. Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. С помощью создания

ассоциаций мы приходим к пониманию участников семантических связей, их ролей, мощности, (количества элементов).

Ассоциация *один-ко-многим,* введенная в примере, означает, что для каждого экземпляра класса Библиотека есть 0 или более экземпляров класса Книга, а для каждого экземпляра класса Книга есть один экземпляр Библиотеки. Эту множественность означает *мощность ассоциации.* Мощность ассоциации бывает одного из трех типов:
− один-к-одному;
− один-ко-многим;
− многие-ко-многим.

Примеры ассоциаций с различными типами мощности приведены на рис.6, они имеют следующий смысл:
− у европейской жены один муж, а у европейского мужа одна жена;
− у восточной жены один муж, а у восточного мужа сколько угодно жен;
− у заказа один клиент, а у клиента сколько угодно заказов;
− человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

Рис. 2. Ассоциации с различными типами мощности.

Наследование

Наследование — это отношение, при котором класс разделяет структуру и поведение, определенные в одном другом (простое наследование) или во многих других (множественное наследование) классах. Между *n* классами наследование определяет иерархию *«является» («is a»),* при которой подкласс наследует от одного или нескольких более общих суперклассов. Говорят, что подкласс *является* специализацией его суперкласса (за счет дополнения или переопределения существующей структуры или поведения).

Пример: дана система для записи параметров полета в «черный ящик», установленный в самолете. Организуем систему в виде иерархии классов, построенной на базе наследования.
Иерархическая структура классов системы для записи параметров полета, находящихся в отношении наследования, показана на рис. 3.

Рис. 3. Иерархия простого наследования.

Здесь Параметры Полета — базовый (корневой) суперкласс, подклассами которого являются Экипаж, Параметры Движения, Приборы, Кабина. В свою очередь, класс Параметры Движения является суперклассом для его подклассов Координаты, Скорость, Ориентация.

Полиморфизм

Полиморфизм — возможность с помощью одного имени обозначать операции из различных классов (но относящихся к общему суперклассу). Вызов обслуживания по полиморфному имени приводит к исполнению одной из некоторого набора операций.

Агрегация

Отношения агрегации между классами аналогичны отношениям агрегации между объектами. Для класса Вагон можно поставить в соответствие классы Тележка, Кузов, Рама. Класс Вагон является агрегатом, а экземпляр класса Тележка — это одна из его частей. Агрегация здесь определена как включение по величине. Это — пример физического включения, означающий, что объект тележка не существует независимо от включающего его экземпляра Вагон. Время жизни этих двух объектов неразрывно связано.
Графическая иллюстрация отношения агрегации по величине (композиции) представлена на рис. 4.

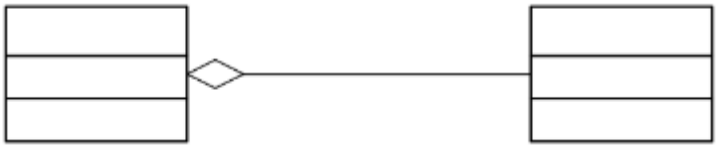


Рис. 4. Отношение агрегации по величине (композиция).

Еще два примера агрегации по ссылке и по величине (композиции) приведены на рис. 5. Здесь показаны класс-агрегат Дом и класс-агрегат Окно, причем указаны роли и множественность частей агрегата (соответствующие пометки имеют линии отношений).

Как показано на рис. 6, возможны и другие формы представления агрегации по величине — композиции. Композицию можно отобразить графическим вложением символов частей в символ агрегата (левая часть рис. 6). Вложенные части демонстрируют свою множественность (мощность, кратность) в правом верхнем углу своего символа. Если метка множественности опущена, по умолчанию считают, что ее значение «много».

Рис. 5. Агрегация классов.

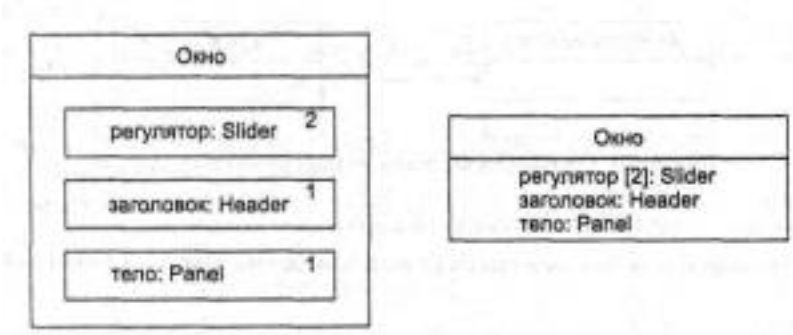


Рис. 6. Формы представления композиции

Зависимость

Зависимость — это отношение, которое показывает, что изменение в одном классе
(независимом) может влиять на другой класс (зависимый), который использует его. Графически зависимость изображается как пунктирная

стрелка, направленная на класс, от которого зависят. С помощью зависимости уточняют, какая абстракция является клиентом, а какая — поставщиком определенной услуги. Пунктирная стрелка зависимости направлена от клиента к поставщику.

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции.

Конкретизация

Г. Буч определяет конкретизацию как процесс наполнения шаблона (родового или параметризованного класса). Целью является получение класса, от которого возможно создание экземпляров.

Родовой класс служит заготовкой, шаблоном, параметры которого могут наполняться (настраиваться) другими классами, типами, объектами, операциями. Он может быть родоначальником большого количества обычных (конкретных) классов. Возможности настройки родового класса представляются списком формальных родовых параметров. Эти параметры в процессе настройки должны заменяться фактическими родовыми параметрами.

Процесс настройки родового класса называют конкретизацией.
Для Класс Очередь произведем настройку, то есть объявим два конкретизированных класса — Очередь Целых Элементов и Очередь Лилипутов: В первом случае мы настраивали класс на конкретный тип Integer (фактический родовой параметр), во втором случае — на конкретный тип Лилипут. Классы Очередь Целых Элементов и Очередь Лилипутов можно использовать как обычные классы. Они содержат все средства родового класса, но только эти средства настроены на использование конкретного типа, заданного при конкретизации. Графическая иллюстрация отношений конкретизации приведена на рис. 7. Отметим, что отношение конкретизации отображается с помощью подписанной стрелки отношения зависимости. Это логично, поскольку конкретизированный класс зависит от родового класса (класса-шаблона).

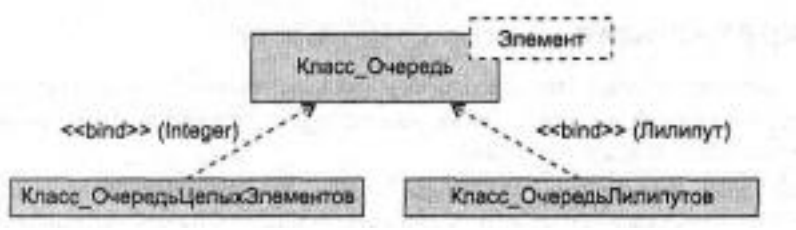


Рис. 7. Отношения конкретизации родового класса.

В данной лабораторной работе вам будет необходимо определить отношения между классами в вашей Банковской системе.

Объект

При объектной декомпозиции часть реального мира описывается в виде взаимодействующих объектов. Термин «объект» в программном обеспечении впервые введен в языке Simula и означал какой-либо аспект моделируемой реальности. Наиболее обще объект может быть определен как нечто, имеющее четко очерченные границы.

Существуют такие объекты, для которых определены явные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, процесс перевозок на железной дороге: его границы явно определены взаимодействием компонентов. Объекты могут быть осязаемыми, но иметь размытые физические границы. Например, реки, туман или толпы людей. Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только объекты для забивания, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоции (например, любовь или гнев). Все перечисленное является свойствами, которые присущи объектам. Можно, например, утверждать, что некоторый человек (объект) любит свою жену (другой объект), или определенный кот (еще один объект) имеет серую шерсть.

На основе имеющегося опыта можно дать следующее определение: Объект обладает состоянием, поведением и индивидуальностью; структура и поведение схожих объектов определяют общий для них класс; термины «экземпляр класса» и «объект» — взаимозаменяемы.

Состояние

Для состояния объекта дадим следующее определение:
Состояние объекта характеризуется перечнем всех возможных (обычно статических) свойств данного объекта и текущими значениями (обычно динамическими) каждого из этих свойств.

Так у файла имеется свойство длина. Этому свойству соответствует динамическое значение, характеризующее количество байт в файле. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской или инвентарный номер), поэтому в данном определении использован термин «обычно динамические».

К числу свойств объекта относятся присущие ему или приобретаемые характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для подъемника характерным является то, что он

сконструирован для подъема и спуска, но не для горизонтального перемещения. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу природы объекта. Все свойства объекта характеризуются значениями их параметров. Эти значения могут быть простыми количественными характеристиками, а могут означать другой объект.

Тот факт, что всякий объект характеризуется состоянием, означает, что он занимает определенное пространство (физически или в памяти компьютера).

Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.
Поведение характеризует то, как объект воздействует или подвергается воздействию других объектов с точки зрения изменения состояния этих объектов и передачи сообщений.
Другими словами, поведение объекта полностью определяется его действиями. *Операцией* называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, объект файл-менеджер может активизировать операцию ChangeAttr для того, чтобы изменить атрибуты файла. Существует также операция: FileSize, которая позволяет определить размер файла, но не может изменить значение этого размера. Применительно к таким языкам программирования, как Smalltalk, принято говорить о передаче сообщений между объектами. В основном понятие «сообщение» совпадает с понятием «операции над объектами», но механизм их действий различен.

Как правило, в объектных и объектно-ориентированных языках операции, выполняемые над данным объектом, называются *методами* (методической частью объекта) и входят составной частью в определение класса.

Из практики известно пять основных видов операций над объектами:
− Модификатор Операция, которая изменяет состояние объекта путем записи или доступа;
− Селектор Операция, дающая доступ для определения состояния объекта без его изменения (операция чтения);
− Итератор Операция доступа к содержанию объекта по частям (в определенной последовательности);
− Конструктор Операция создания и (или) инициализация объекта − Деструктор Операция разрушения объекта и (или) освобождение занимаемой им памяти.

Совокупность всех методов и общедоступных процедур, относящихся к конкретному объекту, образует протокол этого объекта. Протокол объекта, таким образом, определяет оболочку поведения объекта, охватывающую его внутреннее статическое и внешнее динамическое проявления.

Индивидуальность

Индивидуальность — это такие свойства объекта, которые отличают его от всех других объектов.

В большинстве языков программирования и баз данных для различения временных объектов, их взаимной адресации и идентификации используются имена переменных. Источником множества ошибок в объектно-ориентированном программировании является невозможность отличить имя объекта от самого объекта. Это связано с тем, что объекты всегда распределяются динамически и, следовательно, для них существуют: сам объект и ссылка (указатель) на объект, посредством которой производится доступ к объекту (Рис.8).

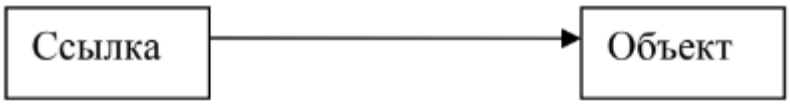


Рис. 8. Объект и ссылка на объект.

Индивидуальность связана с вопросами присвоения и тождественности, а также временем существования объектов.
Для того, чтобы при присвоении объект был продублирован необходимо либо переопределить операцию присвоения, либо создать специальную операцию, выполняющую дублирование. В противном случае будет производиться присвоение ссылок, что приведет к потери памяти и структурной неопределенности — ссылки на один и тот же объект (Рис.9).

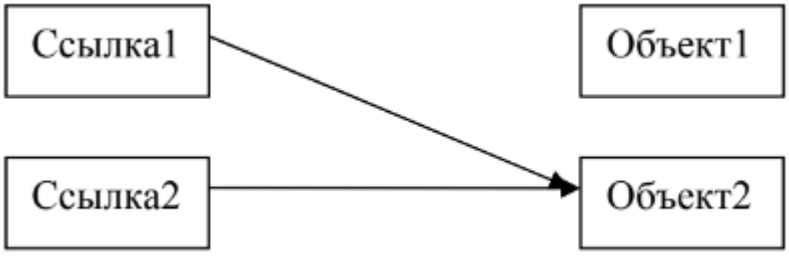


Рис. 9. Присвоение ссылок.

Тождественность представляется достаточно простой концепцией, но может обозначать одну из двух вещей. Во-первых, тождественность можно понимать как использование двух имен для обозначения одного и того же объекта. Во-вторых, тождественность может обозначать наличие одинакового состояния у двух разных объектов.

Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием — момент изъятия отведенного участка памяти. Объект продолжает существовать до тех пор, пока он занимает место в памяти, даже если будет потеряна ссылка на этот объект.

Отношения между объектами

Сами по себе объекты не представляют никакого интереса, только в процессе взаимодействия объектов между собой реализуется цель системы. По выражению Ингалса: «Вместо бессистемной кусочной обработки структур данных мы получаем объекты с ясным поведением, которые обращаются друг с другом по тщательно проработанному интерфейсу и выполняют нужные действия». Рассмотрим, например, структуру самолета, которая определяется как «совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевает эту тенденцию». Только за счет согласованных усилий всех компонентов самолета он имеет возможность летать. Отношения двух любых объектов основываются на предположении, что каждый объект имеет информацию о другом объекте, об операциях, которые над ним можно выполнять, и об ожидаемом поведении. Особый интерес представляют два типа иерархических соотношений объектов: отношение использования и отношение включения. Зейдевиц и Старк назвали эти два типа отношений отношениями старшинства и родства соответственно.

Отношение использования

Отношения использования между несколькими объектами подразумевает возможность передачи сообщений от одного объекта другому.
Пересылка сообщений между объектами обычно однонаправлена, но возможны и двунаправленные связи. Каждый объект, включенный в отношения использования, может выполнять следующие три роли:
− Воздействие Объект может воздействовать на другие объекты, но сам никогда не подвержен воздействию других объектов; в определенном смысле соответствует понятию активный объект.
− Исполнение Объект в этом случае может только подвергаться управлению со стороны других объектов, но никогда не выступает в роли воздействующего объекта.
− Посредничество Такой объект может выступать как в роли воздействующего, так и в роли исполнителя; как правило, объект посредник создается для выполнения операций в интересах какого-либо активного объекта или другого посредника.

Отношение включения

Понятие отношения включения между объектами. Логично предположить, что конкретный объект-подъемник состоит из других объектов, например из мотора и датчика перемещений. Другими словами, последние два объекта являются элементами состояния объекта-подъемника. Между отношениями включения и использования существует взаимная связь. Включение одних объектов в другие предпочтительнее в том плане, что при этом уменьшается число объектов, с которыми приходится оперировать на данном уровне описания. С другой стороны, использование одних объектов другими имеет преимущество: не возникает сильной зависимости между объектами, как в случае включения. В процессе проектирования необходимо тщательно взвешивать оба указанных фактора.

Класс

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие в этих двух понятиях. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию, «выжимку» из объекта.

Таким образом, можно говорить о классе «Млекопитающие», который включает общие характеристики для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать «это млекопитающее» или «то млекопитающее». *Класс — множество объектов, связанных общностью структуры и поведения.*
Любой объект является просто экземпляром класса.

Абстрагирование и обобщение

Для выявления свойств, присущих классу объекту, используются обобщение и абстрагирование.

Обобщение — это мысленное выделение, фиксирование каких-нибудь общих существенных свойств, принадлежащих только данному классу предметов или отношений.

Абстрагирование — это мысленное отвлечение, отделение общих, существенных свойств, выделенных в результате обобщения, от прочих несущественных или необщих свойств рассматриваемых предметов или отношений и отбрасывание (в рамках нашего изучения) последних.

Абстракция — это такие существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

В программировании абстракция — это класс объектов. Обобщение и абстрагирование применяются для формирования абстракции.

Абстракция, как понятие, имеет содержание и объем.
Содержание понятия — это совокупность свойств, мыслимых в понятии. *Объем понятия* — это совокупность объектов, мыслимых в понятии. Класс объектов чаще всего описывается как содержание понятия (Иногда класс объектов задается перечислением всех объектов, т.е. объемом понятия). Для формирования содержания понятия рассматривают объем понятия и изучают каждый составляющий объем понятия объект.
Абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом - вес, в третьем - материалы, из которых он сделан, в четвертом - закон движения предмета и т. д.

Если поставлена задача определения вида графического файла, то главной характеристикой файла будет его сигнатура (идентификационная последовательность символов, присущая только объектам определенного вида. Например, для файла *.png сигнатура состоит из цифр 137 80 78 71 13 10 26 10, для *.gif — символов GIF, для *.bmp — символов BM).
Если же задача заключается в отображении файла существенными будут: размер по вертикали, размер по горизонтали, цветовая палитра и т.п.

Абстракция концентрирует внимание на внешних особенностях объекта, позволяя отделить основное в поведении объекта от его реализации.

В лабораторной работе для всех классов, полученных в результате объектной декомпозиции, необходимо определить существенные свойства.

Ограничение доступа

Созданию абстракции какого-либо объекта должны предшествовать определенные решения о способе ее реализации. Выбранный способ реализации должен быть скрыт и защищен для большинства объектов-пользователей (обращающихся к данной абстракции). Как справедливо отметил Ингалс, «никакая часть сложной системы не должна находиться в зависимости от подробностей внутреннего устройства других частей системы». Ограничение доступа позволяет вносить в программу изменения, сохраняя ее надежность и минимизируя затраты на этот процесс.

Абстрагирование и ограничение доступа являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а ограничение доступа — или иначе защита информации — не позволяет объектам пользователям различать внутреннее устройство объекта.

Ограничение доступа, таким образом, определяет явные барьеры между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как корни растения или метахондрии клеток. Аналогичным образом при проектировании баз данных программисты не обращают внимание на физический смысл данных, а сосредоточиваются на схеме, отражающей логическое строение данных.

В обоих случаях объекты верхнего уровня абстракции не связаны прямо с подробностями их реализации на низком уровне. Лисков считает что «для «работы» абстракции, доступ к ее внутренней структуре должен быть ограничен». Практически это означает наличие двух частей в описании класса: интерфейса и реализации. *Интерфейс* отражает внешнее проявление объекта, создавая абстракцию поведения всех объектов данного класса. *Внутренняя реализация* описывает механизмы достижения желаемого поведения объекта. Принцип такого различения интерфейса и реализации соответствует разделению по сути: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Итак, понятие ограничения доступа можно определить следующим образом:
Ограничение доступа — это процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого.

На практике осуществляется защита как структуры объекта, так и реализации его методов.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через область памяти. В такой ситуации нет необходимости изменять интерфейсную часть объекта, а достаточно переписать реализацию. При правильном использовании ограничение доступа позволяет локализовать те особенности проекта, которые подвержены изменениям. По мере развития системы может оказаться, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют структуру объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать объемы используемой памяти. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов, связанных с данными.

В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации

этих условий для конкретных языков программирования является предметом постоянных обсуждений.

Модульность

Понятие модульности. По мнению Майерса, «Разделение программы на фрагменты позволяет частично уменьшить ее сложность. Однако гораздо важнее тот факт, что разделение программы улучшает проработку ее частей. Эти части весьма ценны для исчерпывающего понимания программы в целом». В некоторых языках программирования, модульность не реализована и классы составляют лишь физическую основу декомпозиции. В других языках, включая модульность является элементом конструкции и позволяет осуществлять на ее основе проектные решения. В таких языках классы и объекты составляют логическую структуру системы; эти абстракции организуются в модули, образуя физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.
По Лискову, «модульность — это разделение программы на раздельно компилируемые фрагменты, имеющие между собой средства сообщения». Можно использовать также и определение Парнаса: «Связи между модулями — это предположения о возможности их использования». В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, также реализуется разделение интерфейсной части и реализации. Таким образом, модульность и ограничение доступа идут неотделимо друг от друга.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зелковиц, утверждая: «поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной».
Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Это такая же ситуация, которая возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных вентилей типа не, и-не, или-не, но можно и объединить группы таких вентилей в стандартные интегральные схемы, например, серий 7400, 7402 или 7404. Отсутствие стандартизованных фрагментов дает программисту гораздо большую степень свободы — как если бы электронщик имел в своем распоряжении средства создания кремниевых кристаллов.

Для небольших задач допустимо описание каждого класса и объекта в отдельном модуле. В других, наиболее тривиальных, случаях в один модуль

лучше собрать все логически связанные классы и объекты и оставить незащищенными все те элементы, которые должны быть видимы для всей программы. Однако такой подход применяется в исключительных случаях. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации механизма передачи сообщений. В больших системах вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определение каждого класса сообщений в отдельном модуле. При этом не только возникает кошмарная ситуация с документированием, но чрезвычайно труден для пользователя даже просто поиск нужных фрагментов описания. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Отсюда можно понять, что защита информации может обернуться и обратной стороной. Деление программы на модули бессистемным образом гораздо хуже, чем отсутствие модульности вообще.

В традиционном структурном проектировании модульность связывается в первую очередь с логической группировкой подпрограмм на основе принципов взаимной связи и общей логики. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта и явно отличающиеся от подпрограмм.

Модульность — это свойство системы, связанное с возможностью декомпозиции ее на ряд тесно связанных модулей.

Таким образом, принципы абстрагирования, ограничения доступа и модульности являются взаимодополняющими. Объект определяет явные границы определенной абстракции, а ограничение доступа и модульность создают барьеры между абстракциями.

В процессе разделения системы на модули могут быть полезными два правила. Первое состоит в следующем: поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе многократно, распределение классов и объектов должно создавать для этого максимальные удобства. Второе правило вытекает из того факта, что многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому размер модуля должен быть соответственно ограничен. Объявление функций в модуле может оказать большое влияние на возможность их вызова из другого модуля, так как это связано с разделением памяти на страницы. Большое количество вызовов между сегментами загружает кэш-память и снижает характеристики всей системы.

Следует сказать о нескольких других моментах. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между

участниками проекта. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные — за остальную часть модулей. На более крупном уровне такие же соотношения справедливы для предприятий-соисполнителей. В последнем случае изменения в интерфейсе вызывают большое «напряжение» независимо от объема этих изменений, что оказывает сильное влияние на проектирование интерфейса. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу. К сожалению, иногда требования по документированию считаются главными при декомпозиции проекта (в большинстве случаев имея негативные последствия). Там, где один модуль требует большого объема документирования, делается десять модулей.
Могут сказываться требования секретности: часть кода может быть несекретной, а другая — секретной; последняя в виде отдельного модуля (модулей).

В лабораторной работе все абстракции обязаны иметь реализации. Для каждой абстракции необходимо описать внутреннею реализацию.

Принципы SOLID

Принципы SOLID это пять принципов объектно-ориентированного программирования. Это набор правил и наилучших подходов, которым нужно следовать при создании структуры классов.
Эти пять принципов помогают понять необходимость определенных шаблонов проектирования и разобраться в архитектуре ПО в целом.
Начнём с небольшого погружения в историю самого термина, а затем перейдем к подробностям: всем «почему?» и «как?» каждого принципа. Рассматривать все это будем на примере создания класса и его постепенного улучшения.

Бэкграунд

Принципы SOLID впервые были представлены знаменитым Дядюшкой Бобом — Робертом Мартином — в его работе «Design Principles and Design Patterns» в 2000 году. Но сам акроним SOLID ввел в оборот Майкл Фезерс, и случилось это несколько позже.
Дядюшка Боб также является автором книг-бестселлеров «Чистый код» и «Чистая архитектура», а кроме того он еще и один из членов «Agile Alliance».
В общем, не удивительно, что все эти концепции чистого кода, объектно-ориентированной архитектуры и шаблонов проектирования как-то связаны и взаимно дополняют друг друга.

Все они служат одной цели:
«Создавать понятный, читаемый, тестируемый код, над которым смогут совместно работать многие разработчики».

Рассмотрим принципы SOLID, а заодно расшифруем акроним:

- **S**ingle Responsibility Principle («Принцип единой ответственности», SRP)
- **O**pen-Closed Principle («Принцип открытости-закрытости», OCP)
- **L**iskov Substitution Principle («Принцип подстановки Барбары Лисков», LSP)
- **I**nterface Segregation Principle («Принцип разделения интерфейса», ISP)
- **D**ependency Inversion Principle («Принцип инверсии зависимостей», DIP)

Принцип единственной ответственности

Принцип единственной ответственности гласит, что класс должен делать какое-то одно действие и, соответственно, для его изменения должна быть только одна причина.

Можно сказать и более «техническим» языком: влиять на спецификацию класса должно только какое-то одно потенциальное изменение в спецификации программы (логика базы данных, логика журналирования и т. п.).

Это означает, что если наш класс — контейнер для данных, скажем, класс Book или Student, и в нем есть какие-то поля, относящиеся к этой сущности, они должны меняться только при изменении модели данных.

Следовать принципу единственной ответственности очень важно. Во-первых, когда несколько разных команд могут работать над одним проектом и редактировать один и тот же класс по многим причинам, это может привести к несовместимости модулей.

Во-вторых, следование этому принципу облегчает контроль версий. Скажем, у нас есть класс для обработки операций базы данных, и мы видим изменение в этом файле в коммитах на GitHub. Если мы последовательно придерживаемся принципа SRP, мы можем быть уверены, что это изменение касается хранения данных или вещей, связанных с базой данных.

Еще пример — конфликты слияния. Они происходят, когда разные команды меняют один и тот же файл. Но если мы следуем SRP, у нас таких конфликтов будет значительно меньше, поскольку файлы могут меняться только по какой-то одной причине. А если конфликты все же будут, их будет легче разрешить.

Распространенные ловушки и антипаттерны

В этом разделе мы рассмотрим некоторые частые ошибки, связанные с принципом единственной ответственности. И, конечно, мы поговорим о том, как такие ошибки исправлять.

Рассмотрим код простой программы для выставления счетов в книжном магазине. Начнем с определения класса `Book`, который будет использоваться в нашем инвойсе.

```
class Book {
 String name;
```

```
 String authorName;
 int year;
 int price;
 String isbn;
 public Book(String name, String authorName, int year,
int price, String isbn) {
   this.name = name;
   this.authorName = authorName;
   this.year = year;
       this.price = price;
   this.isbn = isbn;
 }
}
```

Это простой класс с несколькими полями, ничего особенного. В примере все поля НЕ приватные, чтобы не пришлось иметь дело с геттерами и сеттерами. Сейчас необходимо сосредоточимся на логике.

Создадим класс Invoice, где будет содержаться логика для создания инвойса и подсчета общей суммы. Предположим, что наш магазин торгует исключительно книгами.

```
public class Invoice {
 private Book book;
 private int quantity;
 private double discountRate;
 private double taxRate;
 private double total;
 public Invoice(Book book, int quantity, double
discountRate, double taxRate) {
   this.book = book;
   this.quantity = quantity;
   this.discountRate = discountRate;
   this.taxRate = taxRate;
   this.total = this.calculateTotal();
 }
 public double calculateTotal() {
         double price = ((book.price - book.price *
discountRate) * this.quantity);
   double priceWithTaxes = price * (1 + taxRate);
   return priceWithTaxes;
 }
 public void printInvoice() {
           System.out.println(quantity + "x " +
book.name + " " +          book.price + "$");
```

```
            System.out.println("Discount Rate: " +
discountRate);
            System.out.println("Tax Rate: " + taxRate);
            System.out.println("Total: " + total);
 }
       public void saveToFile(String filename) {
// Creates a file with given name and writes the
invoice
 }
}
```

Вот класс Invoice. В нём также содержатся несколько полей, касающихся выставления счета, и три метода:

- calculateTotal — для подсчета общей суммы,
- printInvoice — для вывода инвойса в консоль,
- saveToFile — для записи инвойса в файл.

Прежде чем читать дальше, определите для себя, что не так с дизайном этого класса.

Разбор проблемы.

Данный класс во многом нарушает принцип единственной ответственности.
Первое нарушение — наш метод printInvoice, содержащий логику вывода. SRP гласит, что наш класс должен иметь единственную причину для изменения. Таковой причиной для нашего класса должно быть изменение системы подсчета.
Но при существующей архитектуре, если потребуется изменить формат вывода, то придется изменить данный класс. По этой причине не следует смешивать в одном классе логику вывода с бизнес-логикой.

В данном классе есть еще один метод, нарушающий SRP, — метод saveToFile. Смешивание долгоживущей логики с бизнес-логикой это еще одна распространенная ошибка.
При построении архитектуры необходимо думать об этом не только как о записи в файл: это могло бы быть сохранение в базу данных, создание вызова API или еще что-нибудь подобное.

Для исправления необходимо создать новые классы для вывода и для долгоживущей логики, чтобы больше не пришлось изменять класс Invoice по этим причинам.
С
оздаем два класса — InvoicePrinter и InvoicePersistence — и перемещаем туда наши методы.

```
public class InvoicePrinter {
   private Invoice invoice;
```

```
    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }
    public void print() {
        System.out.println(invoice.quantity + "x " +
invoice.book.name + " " + invoice.book.price + " $");
        System.out.println("Discount Rate: " +
invoice.discountRate);
        System.out.println("Tax Rate: " +
invoice.taxRate);
        System.out.println("Total: " + invoice.total + "
$");
    }
}
```

```
public class InvoicePersistence {
    Invoice invoice;
    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }
    public void saveToFile(String filename) {
       // Creates a file with given name and writes the
invoice
    }
}
```

Теперь структура классов подчинена принципу единственной ответственности. Каждый класс отвечает за какой-то единственный аспект приложения.

Принцип открытости-закрытости

Принцип открытости-закрытости требует, чтобы классы были открыты для расширения, но закрыты для модификации.
Под модификацией подразумевается изменение кода существующих классов, а под расширением — добавление нового функционала.
В общем, этот принцип имеет в виду следующее: должна быть возможность добавлять новый функционал, не трогая существующий код класса. Это связано с тем, что когда происходит модификация существующего кода, есть риск создать потенциальные баги. Поэтому следует по возможности избегать прикасаться к протестированному и надежному (в целом) коду в процессе разработки ПО.
Чтобы добавлять к классам новый интерфейс и функционал без модификации, необходимо использовать интерфейсы и абстрактные классы.

Изменим предыдущую задачу. Отныне инвойсы должны сохраняться в базе данных, чтобы их можно было легко найти.
Создаем базу данных, подключаем ее и добавляем сохраняющий метод в наш класс InvoicePersistence:

```
public class InvoicePersistence {
    Invoice invoice;
    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }
    public void saveToFile(String filename) {
        // Creates a file with given name and writes the
invoice
    }
    public void saveToDatabase() {
        // Saves the invoice to database
    }
}
```

К сожалению, изначально не были спроектировали классы таким образом, чтобы в будущем они были легко расширяемыми. Поэтому, чтобы добавить новый функционал, необходимо модифицировать класс InvoicePersistence.

Если бы дизайн подчинялся принципу открытости-закрытости, то не понадобилось изменять этот класс.

На данном этапе необходимо произвести рефракторинг кода, чтобы он соответствовал принципу открытости-закрытости.

```
interface InvoicePersistence {
    public void save(Invoice invoice);
}
```

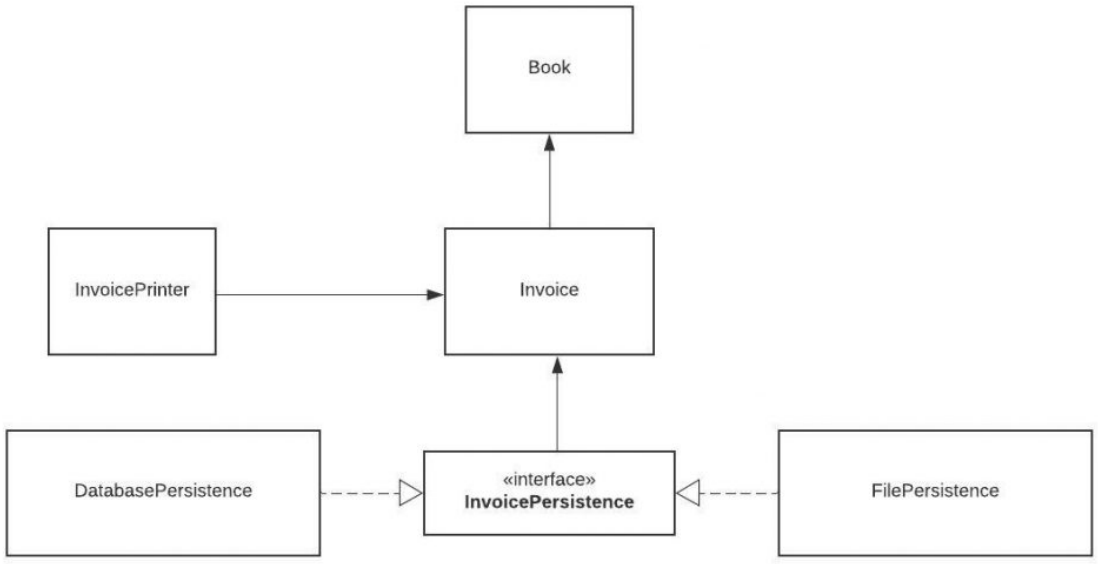
Изменяем тип InvoicePersistence на Interface и добавляем метод для сохранения. Таким образом этот метод будет реализован в каждом долгоживущем классе.

```
public class DatabasePersistence implements
InvoicePersistence {
    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}
```

```
public class FilePersistence implements
InvoicePersistence {
```

```
    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

Теперь классовая структура выглядит следующим образом:



Долгоживущая логика легко расширяема. Если будет необходимость добавить еще одну базу данных, причем одна из баз будет MySQL (реляционная), а другая — MongoDB (не реляционная), это можно будет сделать.
Один из вариантов решения данной задачи - создать несколько классов, обойдясь без интерфейса, и добавить сохраняющий метод в каждый из них.
Но представьте, как расшириться приложение, и появится несколько долгоживущих классов, например, InvoicePersistence, BookPersistence, и придётся создать класс PersistenceManager для управления всеми долгоживущими классами:

```
public class PersistenceManager {
    InvoicePersistence invoicePersistence;
    BookPersistence bookPersistence;

    public PersistenceManager(InvoicePersistence
invoicePersistence,
                              BookPersistence
bookPersistence) {
        this.invoicePersistence = invoicePersistence;
        this.bookPersistence = bookPersistence;
    }
```

Теперь при помощи полиморфизма можно передать в этот класс любой класс, реализующий интерфейс `InvoicePersistence`. Именно эту гибкость и обеспечивают интерфейсы.

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков гласит, что подклассы должны заменять свои базовые классы.
Это означает следующее. Если у есть класс B, являющийся подклассом класса A, то должна быть возможность передать объект класса B любому методу, который ожидает объект класса A, причем этот метод не должен выдать в таком случае какой-то странный output.
Это ожидаемое поведение, потому, что когда используется наследование, разработчики предполагают, что дочерний класс наследует всё, что есть у суперкласса. Дочерний класс расширяет поведение, **но никогда не сужает его**.

Если класс не подчиняется принципу подстановки Барбары Лисков, это приводит к неприятным ошибкам, которые трудно обнаружить.
Принцип Лисков понять легко, но разглядеть в коде трудно. Пример:

```
class Rectangle {
 protected int width, height;
 public Rectangle() {
 }
 public Rectangle(int width, int height) {
   this.width = width;
   this.height = height;
 }
 public int getWidth() {
   return width;
 }
 public void setWidth(int width) {
   this.width = width;
 }
 public int getHeight() {
   return height;
 }
 public void setHeight(int height) {
   this.height = height;
 }
 public int getArea() {
   return width * height;
 }
}
```

Существует простой класс Rectangle и функция getArea, возвращающая площадь прямоугольника.
Было принято решение создать другой класс — для квадратов. Квадрат - это частный случай прямоугольника, в котором ширина равна высоте.

```
class Square extends Rectangle {
 public Square() {}
 public Square(int size) {
   width = height = size;
 }
 @Override
 public void setWidth(int width) {
   super.setWidth(width);
   super.setHeight(width);
 }
 @Override
 public void setHeight(int height) {
   super.setHeight(height);
   super.setWidth(height);
 }
}
```

Класс Square расширяет класс Rectangle. Можно установить в конструкторе одинаковое значение высоты и ширины, но необходимо предусмотреть, чтобы какой-либо клиент (кто-то, использующий класс в своем коде) менял высоту или ширину, нарушая свойство квадрата.
Поэтому необходимо переопределить сеттеры, чтобы устанавливать оба свойства при изменении любого из них. Но таким образом произойдёт нарушение принципа подстановки Лисков.

Поэтому необходимо создать основной класс для тестирования функции getArea.

```
class Test {
  static void getAreaTest(Rectangle r) {
     int width = r.getWidth();
     r.setHeight(10);
     System.out.println("Expected area of " + (width *
10) + ", got " + r.getArea());
  }
  public static void main(String[] args) {
     Rectangle rc = new Rectangle(2, 3);
     getAreaTest(rc);
     Rectangle sq = new Square();
     sq.setWidth(5);
     getAreaTest(sq);
```

```
    }
}
```

Допустим, тестировщик команде разработки создал тестирующую функцию getAreaTest и предположил, что функция getArea не проходит тест для квадратных объектов.
В первом тесте программа создаёт прямоугольник с шириной 2 и высотой 3, а затем вызывает getAreaTest. Результат, как и ожидалось, 20. Но когда тестирование переходит к квадрату, что-то идет не так. Это происходит потому, что вызов функции setHeight в тесте устанавливает также ширину, и результат получается не тот, что ожидался.

Принцип разделения интерфейса

Разделение подразумевает, что вещи нужно хранить отдельно друг от друга, а принцип разделения интерфейса касается именно разделения интерфейсов.
Этот принцип гласит: много клиентоориентированных интерфейсов лучше, чем один интерфейс общего назначения. Клиенты не должны принуждаться к реализации функций, которые им не нужны.

Этот принцип прост как для понимания, так и для применения. Пример:

```
public interface ParkingLot {
 void parkCar(); // Decrease empty spot count by 1
 void unparkCar(); // Increase empty spots by 1
 void getCapacity(); // Returns car capacity
 double calculateFee(Car car); // Returns the price
based on number of hours
 void doPayment(Car car);
}
class Car {
}
```

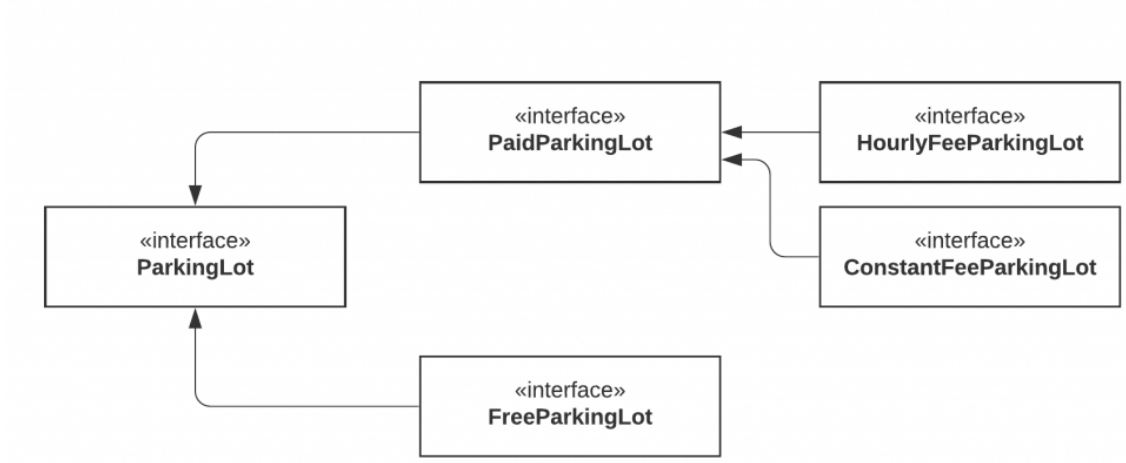
Здесь смоделирована очень упрощенная парковка. Это такая парковка, где клиент платит почасово. Теперь представим, что необходимо реализовать бесплатную парковку.

```
public class FreeParking implements ParkingLot {
 @Override
 public void parkCar() {

 }
 @Override
 public void unparkCar() {
 }
 @Override
 public void getCapacity() {
```

```
  }
  @Override
  public double calculateFee(Car car) {
    return 0;
  }
  @Override
  public void doPayment(Car car) {
    throw new Exception("Parking lot is free");
  }
}
```

Интерфейс нашей парковки состоит из двух частей: логики, касающейся парковки (паркующаяся машина, уезжающая машина, получение информации о вместимости), и логики, касающейся оплаты.
Но это слишком специфично. Из-за этого класс `FreeParking` вынужден реализовывать методы, касающиеся оплаты, а эти методы для него нехарактерны. Поэтому необходимо разделить интерфейсы.



При разделении интерфейсов с новой моделью можно пойти еще дальше: разделить PaidParkingLot, чтоб поддерживать разные типы оплаты.

Теперь данная модель куда более гибкая и расширяемая. Клиентам не нужно реализовывать никакую нерелевантную логику, потому что в интерфейсе парковки мы предоставили только функционал, касающийся самой парковки.

Принцип инверсии зависимостей

Принцип инверсии зависимостей гласит, что классы должны зависеть от интерфейсов или абстрактных классов, а не от конкретных классов и функций.

В своей статье (2000 год) Роберт Мартин (Дядюшка Боб) подытожил формулировку этого принципа так:

«Если OCP обозначает цель объектно-ориентированной архитектуры, то DIP — основной механизм». (OCP — принцип открытости-закрытости, а DIP — принцип инверсии зависимостей).
Эти два принципа, безусловно, связаны друг с другом, и ранее были приведены примеры этого шаблона, когда рассматривался принцип открытости-закрытости.

Всегда необходимо, чтобы классы были открыты для расширения, поэтому в примерах были реорганизовали зависимости, чтобы зависеть от интерфейсов, а не от конкретных классов. К примеру, класс PersistenceManager зависит от InvoicePersistence, а не от классов, реализующих этот интерфейс. Тем самым применяя главное правило Объектно-ориентированного программирования: **Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.**

Лабораторная работа №1.
Постановка задача:

Осуществить выбор языка и технологии для реализации системы межбанковский сообщений (C#, Java, C++ или Python).

Создать и оформить git репозиторий. Использовать заливку кода через сайт github запрещено. Вместо этого следует использовать утилиту командной строки git или графический клиент к нему (подробнее про git).

Наличие файла .gitignore для выбранной вами технологии обязательно (см. коллекцию). Попадание в репозиторий артефактов сборки (.dll, .obj, .class, .jar и т.д.) и служебных файлов среды разработки (.suo, .user и т.д.) недопустимо.

Создать и оформить проект для реализации приложения. Сборка проекта должна быть воспроизводимой (склонировал репозиторий, открыл проект, запустил сборку, запустил собранное приложение). Если для сборки необходимо выполнение дополнительных шагов, то они должны быть указаны в readme в корне репозитория. В C# проекты должны быть объединены в одно решение (один .sln файл).

Создать приложение «Управление финансовой системой», которая позволяет Пользователю выбрать любой банк, который зарегистрирован в системе, авторизоваться в нём под любой ролью и выполнить действия, разрешенные данной роли.

Приложение обязано быть спроектировано согласно **любому** паттерну проектирования ООП и следовать принципам SOLID. Паттерны можно комбинировать в зависимости от задачи модуля программы.

Приложение должно* выполнять следующие функции:

1. Имитировать работу банковской системы с возможностями работы с вкладами клиентов:
    a. Создание;
    b. Хранение;
    c. Снятие;
    d. Перевод;
    e. Накопление;
    f. Блокировка;
    g. Заморозка.
2. Реализовать возможности выдачи кредитов и рассрочек с индивидуальным и фиксированным процентом переплат по следующим условиям:
    a. 3 месяца;
    b. 6 месяцев;
    c. 12 месяцев;
    d. 24 месяца;
    e. Более 24 месяцев.
3. Реализовать возможность хранить базу данных (самописная) с применением любого алгоритма шифрования (к примеру, римское шифрование) и обеспечить сохранность ключей шифрования;
4. Реализовать функционал зарплатного проекта для предприятия;
5. Реализовать возможность авторизации пользователей с ролями (приведены обязательные роли, но можно дополнить):
    a. Клиент
        i. Может зарегистрироваться в системе (требуется апрув менеджера);
        ii. Может взаимодействовать со счетами (открывать, закрывать и т.д.);
        iii. Оформлять кредиты и рассрочки (требуется апрув менеджера);
        iv. Подать заявку на зарплатный проект от предприятия.
    b. Оператор
        i. Может просматривать статистику по движениям средств пользователей и 1 раз отменить** действие по счёту (любой перевод кроме снятия наличных);
        ii. Подтверждает Заявку на зарплатный проект после получения данных от предприятия.
    c. Менеджер
        i. Функционал оператора;
        ii. Подтверждение кредитов и рассрочек;

iii. Отмена** операций произведённым специалистом стороннего предприятия;

d. Специалист стороннего предприятия
   i. Подача документов на зарплатный проект;
   ii. Запрос на перевод средств другому предприятию или сотруднику его предприятия.

e. Администратор
   i. Просмотр всех логов действий (логи могут быть в отдельном файле и зашифрованы);
   ii. Отмена** всех действий пользователей.

6. Обязательные абстракции:
   a. Банк, пользователь, предприятие, счёт, кредит, рассрочка, перевод.

Минимальные данные по клиенту:

- ФИО;
- Серия и Номер паспорта;
- Идентификационный номер;
- Телефон;
- Email;

Предусмотреть возможность работы с иностранными клиентами (желательно отдельный механизм).

Минимальные данные по предприятию:

- ТИП (ИП, ООО, ЗАО и т.д.);
- Юридическое название;
- УНП;
- БИК банка;
- Юридический адрес;

Для демонстрации работы необходимо наполнить систему Тремя банками, Десятью предприятиями и 100 клиентами (во всех банках).

У предприятия только один банк и много счётов, у клиента много банков и много счетов. БАНК, по факту, может являться предприятием.

Разрешается демонстрация работы в консоли, но в таком случае необходимо продумать понятный и простой UX.

*Данный материал имеет минимальные и НЕ полные требования к функционалу проекта. Т.е. здесь не описаны очевидные функции взаимодействия объектов в системе. Дальнейшая реализация архитектуры лежит на плечах студентов.

**Для упрощения работы, можно отменить только ДВА последних действия по КАЖДОМУ пользователю.

Защита проекта:

1. Предоставить UML схему проекта;
2. Произвести алгоритмическую и объектную декомпозиции Банковской системы и отразить её в пояснительной записке к UML схеме;
3. Для всех классов, полученных в результате объектной декомпозиции, необходимо определить существенные свойства и отразить их в пояснительной записке к UML схеме;
4. Продемонстрировать, что все абстракции имеют реализацию. Для каждой абстракции необходимо описать внутреннею реализацию и отразить это в пояснительной записке к UML схеме.