Лекция 2: Декомпозиция, объектно-ориентированное проектирование

Юрий Литвинов
yurii.litvinov@gmail.com

14.02.2022

1. Декомпозиция и модульность

Обсуждение архитектуры программного обеспечения имеет смысл начать с самого начала — что такое сложность и как ею можно управлять. Этот курс больше про объектно-ориентированное программирование, поэтому в этой лекции будет довольно много объектно-ориентированной специфики, но сложность и декомпозиция как способ борьбы со сложностью свойственны всем парадигмам программирования вообще.

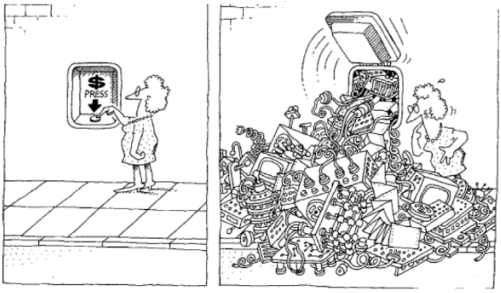
1.1. Сложность

Архитектура как таковая нужна для разработки сложных программных систем. Интересная особенность программного обеспечения в том, что сложность является неотъемлемой его частью и если попытаться от неё избавиться, разрабатываемое ПО потеряет и свою ценность. Это отличает программную инженерию от многих других видов человеческой деятельности — если физики пытаются по сложному природному явлению построить его простую модель, чтобы исследовать её свойства и применить полученные знания обратно к природным явлениям, то у программистов даже модели ПО сложны. Почему так — потому что, как правило, каждая строчка кода уникальна и требует некоторого мыслительного процесса, чтобы её написать; иначе общие части можно было бы вынести или сгенерировать. Так что каждая строчка кода несёт в себе содержательную информацию, которая необходима для работоспособности программы, а поскольку типичное разрабатываемое ПО имеет впечатляющие размеры (см. предыдущую лекцию), то и сложность ПО велика и от неё никуда не деться.

При этом выделяют два «вида» сложности — *существенную сложность* (essential complexity) и *случайную сложность* (accidental complexity). Существенная сложность присуща самой решаемой проблеме, случайная сложность — это сложность, привнесённая способом решения. От случайной сложности можно и нужно избавляться, для этого существует масса полезных приёмов, некоторые из которых будут упомянуты в этой лекции (приёмов, как правило, тактического характера — как спроектировать API, как разделить функциональность по классам, как сделать удобными вызовы методов и т.д.). От существенной сложности избавиться нельзя, она объективно существует в предметной области, а программное обеспечение лишь «моделирует» её в коде. Например, если мы решаем

дифференциальное уравнение, вряд ли наш код может быть проще, чем математический метод его решения.

Поэтому деятельность архитектора направлена не на борьбу со сложностью, а на управление ею. И основной приём управления сложностью — это её сокрытие. Вот рисунок из замечательной книги G. Booch, «Object-oriented analysis and design», который иллюстрирует ситуацию:



Система сложна, существенная сложность может быть впечатляющей, но пользователю ничего знать про это не надо, ему система предоставляет возможно более простой интерфейс для решения его задач. Причём под пользователем тут понимается как конечный пользователь системы, так и коллеги-программисты, которые будут пользоваться написанным кодом, вызывать веб-сервисы и т.д.

Интересно, что все сложные системы имеют общие свойства:

- иерархичность — свойство системы состоять из иерархии подсистем или компонентов;
- наличие относительно небольшого количества видов компонентов, экземпляры которых сложно связаны друг с другом;
- сложная система, как правило, является результатом эволюции простой системы.

На самом деле, скорее всего, не все сложные системы обладают этими характеристиками, но ими обладают те системы, о которых мы можем хоть как-то рассуждать. Сложность, которая в принципе не иерархична или не поддаётся разбиению на виды элементов, остаётся «невидимой» для нас, поскольку у нас просто нет механизмов исследования таких систем. Но ежели система всё-таки обладает вышеперечисленными свойствами, то к ней можно применить следующие подходы:

- декомпозиция — разбиение иерархичной системы на компоненты, каждый из которых состоит из более мелких компонентов и т.д., при этом про каждый компонент можно более-менее осмысленно рассуждать в отдельности;
- абстрагирование — выделение общих свойств компонентов, их классификация, рассуждение не о конкретных элементах системы, а об их типах.

Всё это, естественно, вовсю используется в программировании. С декомпозицией и абстрагированием всё и так понятно, а вот то, что сложные системы появляются из простых,

влечёт тот печальный факт, что даже изначально небольшие и простые программы желательно проектировать сразу так, будто они превратятся в монстров на много миллионов строк кода.

Ещё одно важное соображение состоит в том, что сложность создаваемых систем вполне может превосходить человеческие возможности. Декомпозиция позволяет рассматривать отдельно каждый компонент или аспект системы, так, чтобы по отдельности в них можно было разобраться, но для реально используемого ПО вполне возможно, что систему целиком не знает никто, и даже никто не в состоянии знать все её детали.

1.2. Подходы к декомпозиции

Есть разные способы декомпозиции задачи на подзадачи. Первый — восходящее проектирование, когда сначала создаются отдельные компоненты, а потом из них, как из кирпичиков, собираются более сложные компоненты и, в итоге, система целиком. Такой подход целесообразно применять в исследовательских проектах, когда непонятно, что в конечном итоге может получиться, либо в случае, когда сама предметная область содержит небольшие обособленные задачи, которые можно по-разному комбинировать. Так имеет смысл проектировать библиотеки — рассматривать их как набор кирпичиков, из которых пользователь может сам сложить всё, что ему нужно. Язык программирования FORTH предполагал восходящее проектирование как основной способ создания программ — там программа состояла из *слов*, которые строились из других слов, и т.д. до очень простых конструкций стандартной библиотеки. И, поскольку объявление новых слов было очень простым и их можно было тут же запустить и отладить, рекомендовалось сначала реализовать и отладить простые компоненты системы, а из них постепенно собирать более сложные, тоже запускаясь и отлаживаясь на каждом этапе. В общем-то, современные методологии типа Test Driven Development предполагают похожий подход.

Второй подход — нисходящее проектирование, более «традиционный» в программистском сообществе. Это когда мы сначала рассматриваем задачу целиком, разделяем её на подзадачи, реализуем общую логику, вставляя *заглушки* вместо реализации подзадач, проверяем, что оно не то чтобы работает, но делает всё, что надо, и в правильной последовательности, потом точно так же рассматриваем каждую отдельную подзадачу, пока не придём к окончательному решению. Такой подход уступает восходящему проектированию в том, что пока мы не дописали систему (или подсистему) до самого низа иерархии декомпозиции, мы не можем её внятно тестировать, зато лучше тем, что направляет процесс проектирования и разработки — меньше шансов реализовать что-то, что потом не понадобится, и вообще, всё более предсказуемо. Такой подход используется, когда есть чёткое видение конечного результата и надо минимальными усилиями этого результата достичь (то есть, на самом деле, почти всегда).

Важным понятием для нисходящего проектирования являются *модули* — структурные единицы кода, которые соответствуют подзадачам, на которые разбита система. Модули в объектно-ориентированных языках могут быть классами или компонентами (иногда целыми отдельными подсистемами, например, веб-сервисом), в функциональных языках — отдельными функциями или какими-либо способами их группировки (в F# они так и называются, module), в структурных языках они тоже есть — пара из .h и .c-файлов в C, модули в Паскале, пакеты в Аде и т.д. Модули характеризуются своим *интерфейсом* и *реализацией*. Интерфейсы необходимы модулям в том числе для того, чтобы можно было реализовать

заглушки и чтобы упростить дальнейшую интеграцию — перед тем, как заниматься отдельными подзадачами, мы должны чётко зафиксировать интерфейс, по которому будут общаться реализации разных подзадач, и строго ему следовать. Заглушки тоже должны следовать интерфейсу, но могут иметь пустую или максимально простую реализацию.

Для модулей есть следующие общепринятые правила их проектирования:

- чёткая декомпозиция — каждый модуль занимает своё место в системе, знает, какую задачу он решает; должно быть понятно, как им пользоваться;
- минимизация интерфейса модуля — вечный принцип хорошей архитектуры «меньше знаешь — крепче спишь»; использование модуля должно быть максимально простым, но всё ещё позволять решать задачу;
- один модуль — одна функциональность; если модуль делает что-то и что-то ещё, разбейте этот модуль на два;
- отсутствие побочных эффектов — очень желательное, но не всегда достижимое свойство — чтобы вызов одной и той же функции модуля с одними и теми же параметрами приводил к одним и тем же результатам, и уж тем более не влиял каким-то неочевидным образом на работу других модулей;
- независимость от других модулей — естественно, модуль может использовать другие модули для своей реализации, но он не может знать о деталях реализации других модулей и как-то рассчитывать на эти детали; модуль не имеет права делать предположения о том, кто и как будет его использовать;
- принцип сокрытия данных — внутреннее представление всех данных модуля должно быть известно только ему самому, наружу могут быть видны только абстрактные типы данных, манипулировать которыми можно только через функции модуля.

Модульность как способ разбиения системы на компоненты, вообще говоря, позволяет создавать сколь угодно большие системы, поскольку модуль независим от других и общается с другими только через строго определённые интерфейсы. Разработку каждого модуля можно рассматривать как отдельную задачу, которая меньше по размеру, чем исходная. И проектировщик модуля уровнем выше может вообще ничего не знать про модули уровнем ниже кроме того, как их использовать — что, если соблюдается принцип сокрытия сложности, не должно быть проблемой. Более того, разные модули могут разрабатываться разными людьми или командами, независимо друг от друга, пока все выполняют соглашения, прописанные в интерфейсах (как синтаксические, так и семантические — неправильные типы аргументов поймает компилятор, а вот то, что, например, сначала надо вызвать Connect(), а потом уже GetData() — вряд ли).

При разбиении на модули возникает вопрос, какого размера должен быть модуль, и даже принцип «один модуль — одна функциональность» не всегда даёт ответ на этот вопрос. Начинающие программисты могут пихать вообще всё в один модуль, либо, столь же часто, делать кучу модулей по одной функции в каждом, думая, что вот, красивый модульный дизайн, минимизация интерфейсов и всё такое. Проблема в том, что если модули слишком большие, то сложность каждого модуля неоправданно велика, и, соответственно, неоправданно велики затраты на его разработку. Если модули слишком маленькие, то

неоправданно велики затраты на их интеграцию — модулей становится слишком много, взаимодействия между модулями становятся слишком сложны. В вырожденных случаях (один большой модуль или куча модулей по одной функции) мы никаких преимуществ от модульности вообще не получаем. Поэтому должен соблюдаться некоторый баланс:

Есть знаменитое правило «7+-2», говорящее, что человек может одновременно удерживать в голове порядка семи сущностей (плюс-минус сколько-то, в зависимости от индивидуальных способностей). Это правило может быть хорошей отправной точкой при проектировании системы и разбиении на модули.

Ещё хорошим показателем качества разбиения на модули могут быть численные метрики сопряжения и связности, которые можно просто посчитать (естественно, автоматически) для данного куска кода, или оценить на глаз. *Сопряжение* (Coupling) — мера того, насколько взаимосвязаны разные модули в программе (то есть насколько часто один модуль дёргает другие, насколько много этих других и насколько много они должны знать друг о друге). *Связность* (Cohesion) — мера того, насколько взаимосвязаны функции внутри модуля и насколько похожие задачи они решают. Целью при проектировании является слабое сопряжение (опять-таки, «меньше знаешь — крепче спишь», можно независимо менять компоненты системы, не боясь всё сломать) и сильная связность (функции внутри модуля должны быть связаны друг с другом, иначе их следует разложить в отдельные модули — прежде всего для того, чтобы модуль было проще понять и использовать).

Отдельно обращу внимание, что все эти хорошие практики направлены на упрощение понимания системы. И речь в этом курсе идёт не о понимании, которое может получить внимательный читатель, аккуратно разбираясь с каждой строчкой кода, а о понимании, которое необходимо, когда завтра релиз, а где-то в этих сотне тысяч строк кода есть критический баг, без исправления которого релиз не состоится, а код этот писала соседняя команда из Чехии, которая всем составом ушла в отпуск, и все комментарии на чешском.

2. Объектно-ориентированное проектирование

2.1. Объекты

В объектно-ориентированном проектировании роль модулей играют объекты и классы, а поскольку этот курс ориентирован в основном на объектно-ориентированное программирование, дальше речь пойдёт про них. Впрочем, бывают объектно-ориентированные языки, не имеющие вовсе понятия «класс», так что поговорим сначала именно про объекты. Вот несколько разных определений из нескольких разных источников:

- Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods — **Wikipedia**
- An object stores its state in fields and exposes its behavior through methods — **Oracle**
- Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform — **Thinking in Java**
- An object is some memory that holds a value of some type — **The C++ Programming Language**
- An object is the equivalent of the quanta from which the universe is constructed — **Object Thinking**

The C++ Programming Language подходит к определению объекта наиболее прагматично, потому что, вообще говоря, там это понятие нужно для описания семантики языка, а не для философских рассуждений, но вместе с тем такое определение самое бесполезное, поскольку перекладывает всю ответственность на понятие «type». Определения из Википедии и из доков по Java слишком механистичны и следуют традиции ошибочного понимания объектов как структур с методами — технически это так, но не в этом суть понятия «объект». Больше всего соответствует этому курсу определение из Thinking in Java, объект как изолированная сущность, которая обладает каким-то поведением и может иметь состояние (что отличает объектно-ориентированное программирование от функционального). Object Thinking даёт слишком философское, хотя и совершенно в духе этого курса определение — объект как единица декомпозиции объектно-ориентированной программы.

В любом случае, объекты имеют три важных свойства: состояние, поведение и идентичность. С состоянием связано такое важное понятие, как *инвариант* — набор логических условий, которые должны исполняться всё время жизни объекта. Инвариант важен, поскольку позволяет реализовывать методы будучи уверенными в том, что эти условия выполнены — например, что количество элементов в связном списке равно значению поля length, что позволяет во всех методах, требующих просмотра всего списка, бежать просто от 0 до length и не думать, что будет, если следующий указатель null. Каждый объект сам отвечает за поддержание своего инварианта и обязан не давать возможности нарушить его извне (поэтому public-поля запрещены, например).

Поведение объекта принципиально отличается от вызова функции модуля тем, что объект вправе сам решать, как обработать пришедший к нему запрос, так что правильнее всего считать, что объекты не вызывают методы друг друга, а отправляют друг другу сообщения. Более конкретно, методы объекта могут быть полиморфными, так что код, который будет реально вызван, неизвестен вызывающему (и может даже не существовать на момент, когда вызывающий написан и скомпилирован). То, какой код будет исполнен при полиморфном вызове, определяется типом времени выполнения объекта. Можно сделать даже так, чтобы это определялось типами времени выполнения сразу двух разных объектов, это называется «двойная диспетчеризация» и подробности про это будут, когда дойдём до паттерна «Посетитель». Объекты также могут существовать в разных потоках и обрабатывать вызовы асинхронно, теоретически это можно сделать прозрачно для вызывающего (и есть архитектуры, где каждому объекту всегда сопоставляется отдельный поток).

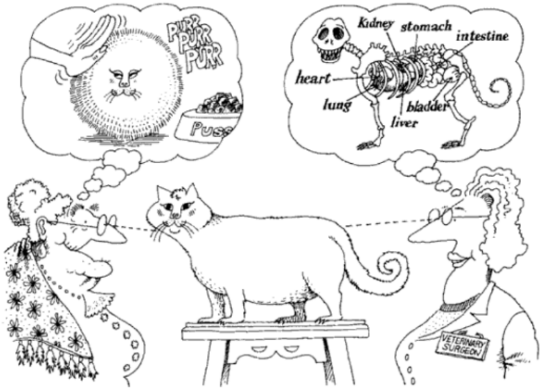
Класс — это тип объекта. Классы определяют структуру данных, которые хранит объект, и методы (с их реализацией, обратите внимание), которые есть у каждого объекта этого класса. Класс — это сущность времени компиляции (и именно с классами в основном работают программисты), объект — сущность времени выполнения (с ними программисты работают, например, отлаживая систему). Но, как уже упоминалось выше, есть языки, не имеющие понятия «класс» (например, JavaScript до версии ES6, Smalltalk).

2.2. «Три кита» объектно-ориентированного программирования

2.2.1. Абстракция

Есть три основных принципа объектно-ориентированного программирования, про которые обычно рассказывают на первом курсе и пишут в любой книге — инкапсуляция, наследование, полиморфизм. Обсудим некоторые из них ещё раз, уже с архитектурной точки зрения. Но начнём мы с абстракции — на самом деле, самого важного понятия ООП, хотя она и не относится к «китам». Потому что она касается не только ООП, но и всего остального — структурного, функционального программирования, алгебры и вообще способности людей к мышлению.

Абстракция выделяет существенные характеристики объекта, отличающие его от остальных объектов, с точки зрения наблюдателя. Один и тот же физический объект может обладать несколькими разными абстракциями одновременно, как на ещё одной картинке из книги Буча «Object-oriented analysis and design»:

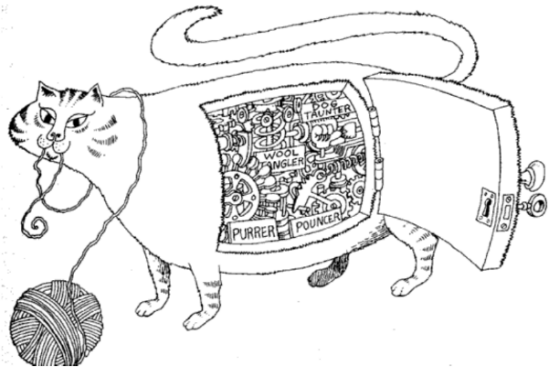


Наличие хорошей и аккуратно спроектированной абстракции в разы упрощает работу с системой. Абстракция может иметь несколько взаимозаменяемых реализаций, и если абстракция достаточно содержательна, на ней могут базироваться реализации других абстракций. Хороший пример использования абстракций — вся математика. Например, в алгебре вводятся алгебраические структуры, такие как кольцо, поле, моноид — абстракции, декларирующие наличие определённых свойств у соответствующих этим абстракциям объектов. Для алгебраических структур доказываются теоремы, которые ничего не знают про реализацию структуры, а знают только то, что декларируется абстракцией (например, для любого моноида верно то-то). Реализации могут быть самые разные — множество целых чисел и операция сложения образуют моноид, строки и операция конкатенации тоже его образуют, функции из int в int и операция композиции — тоже моноид. Значит, для

них верны все теоремы, которые доказаны для моноидов. Теорема — это внешний код, пользующийся абстракцией, моноид — абстракция, композиция функций — реализация абстракции.

2.2.2. Инкапсуляция

Инкапсуляция на самом деле разделяет интерфейс (*контракты*) абстракции и её реализацию. Инкапсуляция опять-таки реализует принцип «меньше знаешь — крепче спишь», позволяя пользователю не знать про реализацию вообще. Но главное, что инкапсуляция защищает инварианты абстракции от их порчи извне. И ещё одна картинка из книжки Буча по этому поводу:



2.2.3. Наследование

Наследование часто определяют как то, что потомок получает все public- и protected-поля и методы предка. Это неправильно (то есть правильно технически, обычно, но неправильно по сути). Наследование — это отношение «является» (is-a) между типами, ну или более формально, один из способов реализации *сабтайпинга* в системе типов. Правильнее всего понимать наследование как «Объект типа-потомка является одновременно объектом типа-предка, поэтому может использоваться везде, где может использоваться предок». Это важно понимать, чтобы избегать семантических нарушений иерархии наследования («традиционное» определение наследования этому только мешает).

Простой пример на понимание наследования — это загадка, которая иногда всё ещё упоминается на собеседованиях — что от чего надо наследовать, прямоугольник от квадрата или квадрат от прямоугольника? Конечно квадрат от прямоугольника, не задумываясь отвечает собеседуемый, ведь квадрат — это такой прямоугольник, у которого все стороны равны. Можно и прямоугольник унаследовать от квадрата, в конце концов и тот и другой имеют длину и ширину, у них можно посчитать площадь, с public- и protected-полями и методами всё ок. Но если мы напишем код, который увеличивает площадь прямоугольника вдвое, удвоив его длину, то для квадрата этот код работать уже не будет — у квадрата есть инвариант, которого нет у прямоугольника — длина всегда равна ширине. Так что квадрат не может использоваться везде, где может использоваться прямоугольник, и наследовать его от прямоугольника нельзя. Но и наоборот нельзя, опять-таки из-за

инварианта — код, увеличивающий сторону квадрата, может сломаться, получив прямоугольник. Итого, внезапно, прямоугольник и квадрат вообще не должны быть связаны наследованием. Важная мораль этой истории — потомок наследует все инварианты предка и не вправе их нарушать (даже если это такие неявные инварианты, как «при увеличении одной из сторон вдвое площадь увеличится вдвое»).

Всему этому противоречит наличие private-наследования в C++. Дело в том, что во времена C++ наследование рассматривалось как способ переиспользования кода — выносим общий код в предка, наследуемся несколькими потомками, получаем в каждом этот общий код. Современная архитектурная мысль рекомендует так не делать (и правда, private-наследование в настоящем коде на C++ встречается крайне редко). Наследование применяется только с двумя целями — как средство классификации и абстрагирования в соответствии с уже существующими системами классификации в предметной области и как средство обеспечения полиморфизма. И для того, и для другого случая достаточно наследования интерфейсов, так что наследование конкретных классов друг от друга нынче большая редкость, а абстрактные классы применяются с сугубо прагматическими целями — чтобы лишний раз поля не писать (и тоже редко).

Вместо наследования для переиспользования кода нынче модно использовать *композицию* — отношение «имеет» (has-a) между типами, которое реализуется обычно с помощью private-полей. Общая функциональность выносится в отдельный класс, и все, кому она нужна, просто включают объект этого класса себе как поле (если повезёт, класс окажется статическим, тогда даже поле не нужно). Композиция позволяет создавать, менять и удалять связи прямо во время выполнения, а также делегировать запросы к объекту-«хозяину» объекту, который внутри, что позволяет делать всякие хорошие вещи типа паттернов «Декоратор», «Прокси», «Адаптер» и т.д., про которые будет дальше в этом курсе.

Интересно, что наследование и композиция более-менее взаимозаменяемы. Технически, наследование и есть композиция — объект-потомок всегда включает в себя объект класса-предка. Можно переделать одно в другое, особенно если язык поддерживает множественное наследование, правда, для этого часто приходится довольно радикально менять точку зрения на систему. Композиция, как уже говорилось, считается более предпочтительной, чем наследование, поскольку наследование фиксировано во время компиляции, а композицию можно менять во время выполнения, что даёт большую гибкость.

2.3. Выделение абстракций

Теперь надо обсудить, откуда, собственно, брать объекты и классы, если система ещё не написана. Допустим, у нас есть техническое задание, требующее разработать приложение для какой-то (возможно, незнакомой нам) предметной области. Тогда нашей первой задачей будет построение объектной модели предметной области и решаемой задачи, для чего обычно выполняются следующие действия:

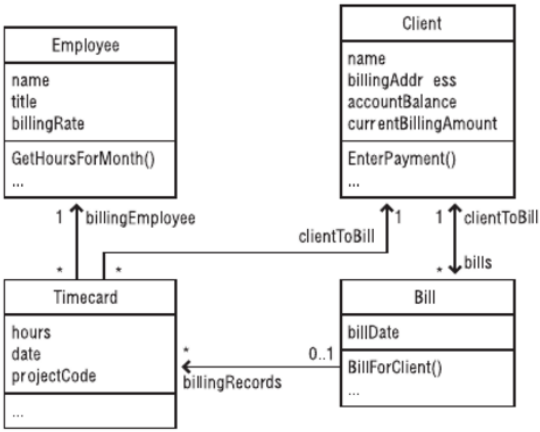
1. определение объектов и их атрибутов, чаще всего на основе общения с экспертами, самостоятельного изучения предметной области и здравого смысла — часто встречающиеся существительные в речи экспертов, скорее всего, станут классами;

2. определение действий, которые могут быть выполнены над каждым объектом (назначение ответственности) — так же, слушаем экспертов и собираем информацию о предметной области, действия — это глаголы;

3. определение связей между объектами, задавая вопросы в духе «а про это кто знает», «а это где используется» и т.д.

4. определение интерфейса каждого объекта, когда фиксируется и формализуется всё, что удалось сделать на предыдущих этапах.

Результатом этой деятельности будет первый грубый набросок архитектуры системы, выражаемый обычно в виде диаграммы классов:



Классы, конечно, появляются не только из предметной области, но и могут появляться в процессе реализации. Часто встречающиеся источники абстракций таковы:

- изоляция сложности — имеет смысл делать отдельными классами сложные алгоритмы, нетривиальные структуры данных, целые сложные подсистемы прятать за простыми фасадами. Делается это для того, чтобы, во-первых, спрятать сложность от остальной системы, во-вторых, иметь возможность менять алгоритмы, оптимизировать внутреннее представление и т.д., зная, что это ничего не сломает. При этом надо тщательно проектировать интерфейсы абстракций, скрывающих сложность, потому что они имеют тенденцию сами становиться слишком сложными, что убивает все преимущества инкапсуляции.

- Изоляция возможных изменений — прятать всё, что имеет большие шансы поменяться, в отдельный класс или подсистему, чтобы их можно было выкинуть и переписать, если изменение таки произойдёт. Однако не надо переусердствовать, хорошая архитектура позволяет себя менять и без специальной поддержки изменчивости. Основные источники изменений:

  – бизнес-правила, то есть алгоритмы из предметной области, которые реализует программа, они меняются на удивление часто;

  – зависимости от оборудования и операционной системы — меняются редко, но изменение может быть критично для жизни программного продукта — ваше любимое железо через пять лет, скорее всего, никто не будет выпускать, что при неаккуратном проектировании может потребовать переписать всю систему

с нуля. Если вы думаете «кому мой программный продукт будет нужен через пять лет», вспомните (или почитайте) про проблему 2000 года;

- ввод-вывод, как формат файлов сохранения, так и формат сетевых пакетов, набор допустимых команд и т.д., они наверняка будут почему-то меняться;
- нестандартные возможности языка — система должна иметь возможность пережить смену компилятора;
- сложные аспекты проектирования и конструирования — это обсуждалось выше, сложность полезно прятать, но не только сложность реализации, но и сложность архитектуры — сложные решения часто оказываются неверными, их должно быть можно легко поменять;
- третьесторонние компоненты — нельзя критически зависеть от чего-то, что перестанет поддерживаться через полгода или станет стоить бешеных денег.

- Изоляция служебной функциональности — сущности, которые нужны для работы других сущностей. Это различного рода фабрики, репозитории, диспетчеры, медиаторы, сервисы (группы статических методов, полезных для работы) и т.д. и т.п. Классы, отвечающие за бизнес-логику, должны содержать только бизнес-логику, не захламляя её деталями реализации. Детали должны жить отдельно.

2.4. Принципы SOLID

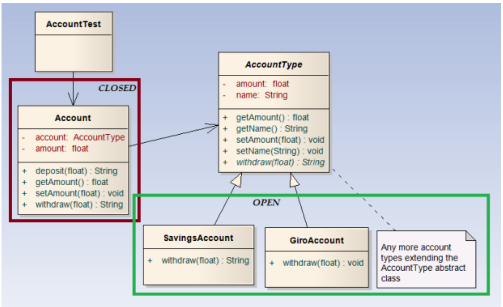
Есть пять базовых принципов объектно-ориентированного проектирования, известные как принципы SOLID. Эти принципы должны применяться при проектировании всех объектно-ориентированных систем, их любят спрашивать на собеседованиях, и вообще, их следовало бы рассказывать ещё на первом курсе, но обычно этого не делают.

Принципы таковы:

- Single responsibility principle, принцип единственности ответственности — каждый класс должен делать что-то одно. Кажется привлекательным иметь «швейцарский нож», который бы один решал все возможные проблемы, но такой класс, во-первых, тяжёл в сопровождении, а во-вторых, сложно понять его роль в системе, сложно объяснить её новым людям в проекте. Поэтому про каждый класс должно быть можно в одном предложении сказать, зачем он нужен, например, «утилиты для работы со строками», «вычислялка процентов по вкладу». И поэтому важно писать комментарии к самому классу — если у вас не получается сформулировать кратко, что делает этот класс, значит, это плохая абстракция и её надо разбить на несколько классов. Ещё надо следить, чтобы ответственность класса была полностью инкапсулирована в этом классе. То есть, например, если у вас есть «утилиты для работы со строками» и «ещё утилиты для работы со строками», дела ваши плохи — вы никогда в жизни не запомните, где нужный вам метод. Это всё касается не только классов, но и функций, и целых подсистем.

- Open/closed principle, принцип открытости/закрытости — абстракция (класс, модуль, функция) должна быть открыта для расширения, но закрыта для изменения. То есть, если интерфейс уже стабилизировался и им начали пользоваться, менять

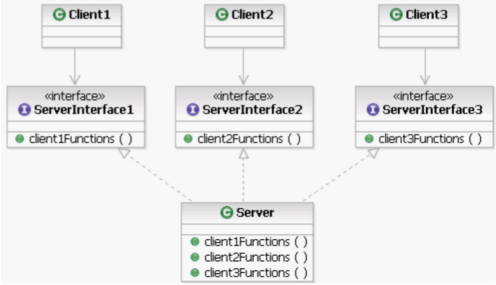
его нельзя. Если надо добавить в абстракцию новую функциональность, можно использовать наследование и/или заранее подготовленные точки расширения, как на картинке:



Если это правило не соблюдать, интерфейс абстракции начнёт увеличиваться в размерах, сам собой начнёт нарушаться принцип единственности ответственности, и в результате мы получим несопровождаемого и неюзабельного монстра (см. антипаттерны «God object» и «Swiss army knife» далее в этом курсе).

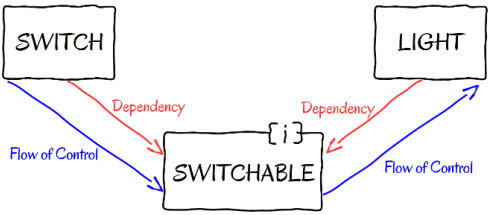
- Liskov substitution principle, принцип подстановки Барбары Лисков — то самое определение наследования, о котором шла речь выше. Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Это означает, в частности, что классы-потомки должны реализовывать интерфейсы классов предков (как чисто синтаксически, так и семантически — делать то, что ожидается от предка) и, про что часто забывают, выполнять все инварианты классов-предков, явные или неявные. Хороший пример проблем с принципом подстановки — проверяемые исключения в Java. Методы класса-предка могли задекларировать исключения, которые они могут бросать, тогда потомок, переопределяющий эти методы, имел право бросать только эти исключения. Почему — вызывающий мог использовать try/catch для обработки ошибок, и если там были написаны catch-блоки для исключений предка, а потомок бросал что-то новое, это ломало вызывающего и тем самым нарушало принцип подстановки. Компилятор проверял такие ситуации и сообщал об ошибке, если потомок пытался бросить незадекларированное в предке проверяемое исключение. Проблема в том, что в большинстве случаев заранее предсказать все исключительные ситуации, которые могут возникнуть в потомках, при написании предка невозможно, поэтому проверяемые исключения особо не используются нынче в Java и так и не попали ни в один другой язык.

- Interface segregation principle, принцип разделения интерфейсов — клиенты не должны зависеть от методов, которые они не используют. Если у абстракции есть группы методов, нужные разным клиентам, то имеет смысл сделать разные интерфейсы, по одному на каждую группу методов. Например, у нас есть модем (например, 4G/LTE), который умеет устанавливать соединение и при наличии соединения передавать данные. У нас есть системная утилита установки соединения, которая следит за сотовой сетью и устанавливает соединение, если это возможно, и у нас есть сетевой стек, который передаёт данные, если его попросят. Если просто сделать интерфейс IModem, который будет иметь методы Connect() и Send(), это как раз нарушит принцип разделения интерфейсов, потому что утилиту установки соединения надо будет пересобирать каждый раз, когда меняются параметры метода Send(), который ей совсем не нужен, а сетевой стек придётся пересобирать каждый раз, когда меняется Connect(), который в этом примере сетевой стек не использует (например, потому, что не хочет знать подробности сотовой связи). Необходимость разделять интерфейсы абстракции, впрочем, может указывать на нарушение принципа единственности ответственности, но может и нет (в нашем примере «модем» был единой сущностью и в этом плане всё было ок, просто клиенты использовали его по-разному). Ещё важно не переусердствовать, ведь формально у класса должно быть ровно столько интерфейсов, сколько клиентов его используют, но это бессмысленно и опасно в плане излишних зависимостей. Группировать в интерфейсы надо методы, которые реально кластеризуются по смыслу, а не чисто механически.



- Dependency inversion principle, принцип инверсии зависимостей — модули верхних

уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций. Эту фразу, бывает, студенты выучивают наизусть и повторяют на экзамене как заклинание, совершенно не понимая её смысла. Проще всего пояснить суть дела на примере:



Положим, у нас есть выключатель, который управляет лампочкой. Наивным решением было бы сделать класс Switch, у которого было бы поле типа Light (лампочка), и он бы вызывал её методы on()/off(). Чем это плохо — выключатель может управлять только лампочкой, выключатель надо перекомпилировать, когда меняется лампочка, работу выключателя невозможно понять, не зная про лампочку. Если выключатель ещё и сам создаёт лампочку, невозможно подсунуть вместо неё mock-объект, что затрудняет тестирование. Правильное решение (та самая инверсия зависимостей) — сделать интерфейс «то, что можно включать/выключать» (Switchable), сделать так, чтобы лампочка реализовывала этот интерфейс и сделать в классе Switch поле типа Switchable, так, чтобы он вообще ничего про лампочку не знал. Теперь выключатель сам объект-лампочку создать в принципе не может (он просто не знает, что она есть), так что должен получать свою зависимость либо параметром в конструктор, либо в метод-сеттер (что называется Dependency Injection, внедрение зависимости). Кстати, существуют и активно используются целые библиотеки, занимающиеся управлением зависимостями и инициализацией объектов, так называемые IoC-контейнеры (Inversion of Control), без них не обходится ни одна современная библиотека для разработки веб-приложений, например. Тем не менее, и без библиотек это хорошая практика, существенно уменьшающая зависимости между модулями программы.

2.5. Закон Деметры

Ещё одно хорошее правило, не являющееся частью принципов SOLID — это закон Деметры, кратко формулирующийся как «Не разговаривай с незнакомцами». Более формально, правило говорит о том, что объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B, и у объекта B есть доступ к объекту C. Например,

```
book.pages.last.text  // Плохо
book.pages().last().text()  // Лучше, но тоже не супер
book.lastPageText()  // Идеально
```

Почему так — вызовы «по цепочке» раскрывают внутреннюю структуру данных класса и не дают её потом изменить. В нашем примере с книгой цепочка вызовов говорит, что

в книге есть коллекция страниц, у которой обязательно есть метод last() (и любая другая коллекция без этого метода не подойдёт), и что там лежат страницы, у которых есть метод text(). Если мы хотим просто получить текст последней страницы, это ну очень много знаний, которые нам не нужны и заставляют нас зависеть от автора класса-книги. В книжке Р. Мартина «Чистый код» проводится аналогия с крушением поезда: где-то посреди структуры меняется внутреннее представление — и всё, все эти цепочки вызовов надо переделывать.

Как обычно, главное не переусердствовать. Код вида book.firstPageText(), book.secondPageText(), book.thirdPageText() и т.д. гораздо хуже, чем «паровозик». Закон Деметры говорит просто, что не надо выставлять напоказ структуру данных, которая не является частью абстракции, и если мы реально ожидаем доступ к каждой странице (что, вообще говоря, для книги вполне ок), то имеет смысл предоставить-таки коллекцию pages(). Более того, с нарушением закона Деметры часто путают очень полезный приём программирования «Fluent API», когда через точку пишется последовательность действий, которые надо выполнить над объектом (хороший пример — это Java Stream API или LINQ в .NET), этот же приём используется в паттерне «Строитель», про который тоже будет в этом курсе. Здесь с законом Деметры всё ок, потому что цепочка fluent-вызовов, как правило, возвращает один и тот же объект, не раскрывая его внутренней структуры (любители функционального программирования могут провести параллели с монадами, и на самом деле LINQ проектировался как самые настоящие монады на C#).