Разработка → Я не знаю ООП

ООП*, Программирование*

Я не умею программировать на объектно-ориентированных языках. Не научился. После 5 лет промышленного программирования на Java я всё ещё не знаю, как создать хорошую систему в объектно-ориентированном стиле. Просто не понимаю.

Я пытался научиться, честно. Я изучал паттерны, читал код open source проектов, пытался строить в голове стройные концепции, но так и не понял принципы создания качественных объектно-ориентированных программ. Возможно кто-то другой их понял, но не я.

И вот несколько вещей, которые вызывают у меня непонимание.

Я не знаю, что такое ООП

Серьёзно. Мне сложно сформулировать основные идеи ООП. В функциональном программировании одной из основных идей является отсутствие состояния. В структурном — декомпозиция. В модульном — разделение функционала в законченные блоки. В любой из этих парадигм доминирующие принципы распространяются на 95% кода, а язык спроектирован так, чтобы поощрять их использование. Для ООП я таких правил не знаю.

Принято считать, что объектно-ориентированное программирование строится на 4 основных принципах (когда я был мал, их было всего 3, но ведь тогда и деревья были большими). Эти принципы:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Смахивает на свод правил, не так ли? Значит вот оно, те самые правила, которым нужно следовать в 95% случаев? Хмм, давайте посмотрим поближе.

Абстракция

Абстракция — это мощнейшее средство программирования. Именно то, что позволяет нам строить большие системы и поддерживать контроль над ними. Вряд ли мы когда-либо подошли бы хотя бы близко к сегодняшнему уровню программ, если бы не были вооружены таким инструментом. Однако как абстракция соотносится с ООП?

Во-первых, абстрагирование не является атрибутом исключительно ООП, да и вообще программирования. Процесс создания уровней абстракции распространяется практически на все области знаний человека. Так, мы можем делать суждения о материалах, не вдаваясь в подробности их молекулярной структуры. Или говорить о предметах, не упоминая материалы, из которых они сделаны. Или рассуждать о сложных механизмах, таких как компьютер, турбина самолёта или человеческое тело, не вспоминая отдельных деталей этих сущностей.

Во-вторых, абстракции в программировании были всегда, начиная с записей Ады Лавлейс, которую принято считать первым в истории программистом. С тех пор люди бесперерывно создавали в своих программах абстракции, зачастую имея для этого лишь простейшие средства. Так, Абельсон и Сассман в своей небезызвестной книге описывают, как создать систему решения уравнений с поддержкой комплексных чисел и даже полиномов, имея на вооружении только процедуры и связные списки. Так какие же дополнительные средства абстрагирования несёт в себе ООП? Понятия не имею. Выделение кода в подпрограммы? Это умеет любой высокоуровневый язык. Объединение подпрограмм в одном месте? Для этого достаточно модулей. Типизация? Она была задолго до ООП. Пример с системой решения уравнений хорошо показывает, что построение уровней абстракции не столько зависит от средств языка, сколько от способностей программиста.

Инкапсуляция

Главный козырь инкапсуляции в сокрытии реализации. Клиентский код видит только интерфейс, и только на него может рассчитывать. Это развязывает руки разработчикам, которые могут решить изменить реализацию. И это действительно круто. Но вопрос опять же в том, причём тут ООП? *Все* вышеперечисленные парадигмы подразумевают сокрытие реализации. Программируя на C вы выделяете интерфейс в header-файлы, Oberon позволяет делать поля и методы локальными для модуля, наконец, абстракция во многих языках строится просто посредствам подпрограмм, которые также инкапсулируют реализацию. Более того, объектно-ориентированные языки сами зачастую *нарушают правило инкапсуляции*, предоставляя доступ к данным через специальные методы — getters и setters в Java, properties в C# и т.д. (В комментариях выяснили, что некоторые объекты в языках программирования не являются объектами с точки зрения ООП: data transfer objects отвечают исключительно за перенос данных, и поэтому не являются полноценными сущностями ООП, и, следовательно, для них нет необходимости сохранять инкапсуляцию. С другой стороны, методы доступа лучше сохранять для поддержания гибкости архитектуры. Вот так всё непросто.) Более того, некоторые объектно-ориентированные языки, такие как Python, вообще не пытаются что-то скрыть, а расчитывают исключительно на разумность разработчиков, использующих этот код.

Наследование

Наследование — это одна из немногих новых вещей, которые действительно вышли на сцену благодаря ООП. Нет, объектно-ориентированные языки не создали новую идею — наследование вполне можно реализовать и в любой другой парадигме — однако ООП впервые вывело эту концепцию на уровень самого языка. Очевидны и плюсы наследования: когда вас *почти* устраивает какой-то класс, вы

можете создать потомка и переопределить какую-то часть его функциональности. В языках, поддерживающих множественное наследование, таких как C++ или Scala (в последней — за счёт traits), появляется ещё один вариант использования — mixins, небольшие классы, позволяющие «примешивать» функциональность к новому классу, не копируя код.

Значит, вот оно — то, что выделяет ООП как парадигму среди других? Хмм… если так, то почему мы так редко используем его в реальном коде? Помните, я говорил про 95% кода, подчиняющихся правилам доминирующей парадигмы? Я ведь не шутил. В функциональном программировании не меньше 95% кода использует неизменяемые данные и функции без side-эффектов. В модульном практически весь код логично расфасован по модулям. Приверженцы структурного программирования, следуя заветам Дейкстры, стараются разбивать все части программы на небольшие части. Наследование используется гораздо реже. Может быть в 10% кода, может быть в 50%, в отдельных случаях (например, при наследовании от классов фреймворка) — в 70%, но не больше. Потому что в большинстве ситуаций это просто *не нужно*.

Более того, наследование *опасно* для хорошего дизайна. Настолько опасно, что Банда Четырех (казалось бы, проповедники ООП) в своей книге рекомендуют при возможности заменять его на делегирование. Наследование в том виде, в котором оно существует в популярных ныне языках ведёт к хрупкому дизайну. Унаследовавшись от одного предка, класс уже не может наследоваться от других. Изменение предка так же становится опасным. Существуют, конечно, модификаторы private/protected, но и они требуют неслабых экстрасенсорных способностей для угадывания, как класс может измениться и как его может использовать клиентский код. Наследование настолько опасно и неудобно, что крупные фреймворки (такие как Spring и EJB в Java) отказываются от них, переходя на другие, не объектно-ориентированные средства (например, метапрограммирование). Последствия настолько непредсказуемы, что некоторые библиотеки (такие как Guava) прописывает своим классам модификаторы, запрещающие наследование, а в новом языке Go было решено вообще отказаться от иерархии наследования.

Полиморфизм

Пожалуй, полиморфизм — это лучшее, что есть в объектно-ориентированном программировании. Благодаря полиморфизму объект типа Person при выводе выглядит как «Шандоркин Адам Имполитович», а объект типа Point — как "[84.23 12.61]". Именно он позволяет написать «Mat1 * Mat2» и получить произведение матриц, аналогично произведению обычных чисел. Без него не получилось бы и считывать данные из входного потока, не заботясь о том, приходят они из сети, файла или строки в памяти. Везде, где есть интерфейсы, подразумевается и полиморфизм.

Мне правда нравится полиморфизм. Поэтому я даже не стану говорить о его проблемах в мейнстримовых языках. Я также промолчу про узость подхода диспетчеризации только по типу, и про то, как это могло бы быть сделано. В большинстве случаев он работает как надо, а это уже неплохо. Вопрос в другом: является ли полиморфизм тем самым принципом, отличающим ООП от других парадигм? Если бы вы спросили меня (а раз уж вы читаете этот текст, значит, можно считать, что спросили), я бы ответил «нет». И причина всё в тех же процентах использования в коде. Возможно, интерфейсы и полиморфные методы встречаются немного чаще наследования. Но сравните количество строк кода, занимаемое ими, с количеством строк, написанных в обычном процедурном стиле — последних всегда больше. Глядя на языки, поощряющие такой стиль программирования, я не могу назвать их полиморфными. Языки с поддержкой полиморфизма — да, так нормально. Но не полиморфные языки.

(Впрочем, это моё мнение. Вы всегда можете не согласиться.)

Итак, абстракция, инкапсуляция, наследование и полиморфизм — всё это есть в ООП, но ничто из этого не является его неотъемлемым атрибутом. Тогда что такое ООП? Есть мнение, что суть объектно-ориентированного программирования лежит в, собственно, объектах (звучит вполне логично) и классах. Именно идея объединения кода и данных, а также мысль о том, что объекты в программе отражают сущности реального мира. К этому мнению мы ещё вернёмся, но для начала расставим некоторые точки над i.

Чьё ООП круче?

Из предыдущей части видно, что языки программирования могут сильно отличаться по способу реализации объектно-ориентированного программирования. Если взять совокупность всех реализаций ООП во всех языках, то вероятнее всего вы не найдёте вообще ни одной общей для всех черты. Чтобы как-то ограничить этот зоопарк и внести ясность в рассуждения, я остановлюсь только одной группе — чисто объекто-ориентированные языки, а именно Java и C#. Термин «чисто объектно-ориентированный» в данном случае означает, что язык не поддерживает другие парадигмы или реализует их через всё то же ООП. Python или Ruby, например, не буду являться чистыми, т.к. вы вполне можете написать полноценную программу на них без единого объявления класса.

Чтобы лучше понять суть ООП в Java и C#, пробежимся по примерам реализации этой парадигмы в других языках.

Smalltalk. В отличие от своих современных коллег, этот язык имел динамическую типизацию и использовал message-passing style для реализации ООП. Вместо вызовов методов объекты посылали друг другу сообщения, а если получатель не мог обработать то, что пришло, он просто пересылал сообщение кому-то ещё.

Common Lisp. Изначально CL придерживался такой же парадигмы. Затем разработчики решили, что писать `(send obj 'some-message)` — это слишком долго, и преобразовали нотацию в вызов метода — `(some-method obj)`. На сегодняшний день Common Lisp имеет развитую систему объектно-ориентированного программирования (CLOS) с поддержкой множественного наследования, мультиметодов и метаклассов. Отличительной чертой является то, что ООП в CL крутится не вокруг объектов, а вокруг обобщённых функций.

Clojure. Clojure имеет целых 2 системы объектно-ориентированного программирования — одну, унаследованную от Java, и вторую, основанную на мультиметодах и более похожую на CLOS.

R. Этот язык для статистического анализа данных также имеет 2 системы объектно-ориентированного программирования — S3 и S4. Обе унаследованы от языка S (что не удивительно, учитывая, что R — это open source реализация коммерческого S). S4 по большей части соотвествует реализациям ООП в современных мейнстримовых языках. S3 является более легковесным вариантом, элементарно реализуемым средствами самого языка: создаётся одна общая функция, диспетчеризирующая запросы по атрибуту «class» полученного объекта.

JavaScript. По идеологии похож на Smalltalk, хотя и использует другой синтаксис. Вместо наследования использует прототипирование: если искомого свойства или вызванного метода в самом объекте нет, то запрос передаётся объекту-прототипу (свойство prototype всех объектов JavaScript). Интересным является факт, что поведение всех объектов класса можно поменять, заменив один из методов прототипа (очень красиво, например, выглядит добавление метода `.toBASE64` для класса строки).

Python. В целом придерживается той же концепции, что и мейнстримовые языки, но кроме этого поддерживает передачу поиска атрибута другому объекту, как в JavaScript или Smalltalk.

Haskell. В Haskell вообще нет состояния, а значит и объектов в обычном понимании. Тем не менее, своеобразное ООП там всё-таки есть: типы данных (types) могут принадлежать одному или более классам типов (type classes). Например, практически все типы в Haskell состоят в классе Eq (отвечает за операции сравнения 2-х объектов), а все числа дополнительно в классах Num (операции над числами) и Ord (операции <, <=, >=, >). В менстримовых языках типам соответствуют классы (данных), а классам типов — интерфейсы.

Stateful или Stateless?

Но вернёмся к более распространённым системам объектно-ориентированного программирования. Чего я никогда не мог понять, так это отношения объектов с внутренним состоянием. До изучения ООП всё было просто и прозрачно: есть структуры, хранящие несколько связанных данных, есть процедуры (функции), их обрабатывающие. выгулять(собаку), снятьс(аккаунт, сумма). Потом пришли объекты, и это было тоже ничего (хотя читать программы стало гораздо сложней — моя собака выгуливала [кого?], а аккаунт снимал деньги [откуда?]). Затем я узнал про сокрытие данных. Я всё ещё мог выгулять собаку, но вот посмотреть состав её пищи уже не мог. Пища не выполняла никаких действий (наверное, можно было написать, что пища.съесть(собака), но я всё-таки предпочитаю, чтобы моя собака ела пищу, а не наоборот). Пища — это просто данные, а мне (и моей собаке) нужно было просто получить к ним доступ. Всё *просто*. Но в рамки парадигмы влезть было уже невозможно, как в старые джинсы конца 90-х.

Ну ладно, у нас есть методы доступа к данным. Пойдём на этот маленький самообман и притворимся, что данные у нас действительно скрыты. Зато я теперь знаю, что объекты — это в первую очередь данные, а потом уже, возможно, методы их обрабатывающие. Я понял, как писать программы, к чему нужно стремиться при проектировании.

Не успел я насладиться просветлением, как увидил в интернетах слово stateless (готов поклясться, оно было окружено сиянием, а над буквами t и l висел нимб). Короткое изучение литературы открыло чудесный мир прозрачного потока управления и простой многопоточности без необходимости отслеживать согласованность объекта. Конечно, мне сразу захотелось прикоснуться к этому чудесному миру. Однако это означало полный отказ от любых правил — теперь было непонятно, следует ли собаке самой себя выгуливать, или для этого нужен специальный ВыгулМенеджер; нужен ли аккаунт, или со всей работой справится Банк, а если так, то должен он списывать деньги статически или динамически и т.д. Количество вариантов использования возрасло экспоненциально, и все варианты в будущем могли привести к необходимости серьёзного рефакторинга.

Я до сих пор не знаю, когда объект следует сделать stateless, когда stateful, а когда просто контейнером данных. Иногда это очевидно, но чаще всего нет.

Типизация: статическая или динамическая?

Еща одна вещь, с которой я не могу определиться относительно таких языков, как C# и Java, это являются они статически или динамически типизированными. Наверное большинство людей воскликнет «Что за глупость! Конечно статически типизированными! Типы проверяются во время компиляции!». Но действительно ли всё так просто? Правда ли, что программист, прописывая в параметрах метода тип X может быть уверен, что в него всегда будут передаваться объекты именно типа X? Верно — не может, т.к. в метод X можно будет передать параметр типа X *или его наследника*. Казалось бы, ну и что? Наследники класса X всё равно будут иметь те же методы, что и X. Методы методами, а вот логика работы может оказаться совершенно другой. Самый распространённый случай, это когда дочерний класс оказывается соптимизированным под другие нужды, чем X, а наш метод может рассчитывать именно на ту оптимизацию (если вам такой сценарий кажется нереалистичным, попробуйте написать плагин к какой-нибудь развитой open source библиотеке — либо вы потратите несколько недель на разбор архитектуры и алгоритмов библиотеки, либо будете просто наугад вызывать методы с подходящей сигнатурой). В итоге программа работает, однако скорость работы падает на порядок. Хотя с точки зрения компилятора всё корректно. Показательно, что Scala, которую называют наследницей Java, во многих местах по умолчанию разрешает передавать только аргументы именно указанного типа, хотя это поведение и можно изменить.

Другая проблема — это значение null, которое может быть передано практически вместо любого объекта в Java и вместо любого Nullable объекта в C#. null принадлежит сразу всем типам, и в то же время не принадлежит ни одному. null не имеет ни полей, ни методов, поэтому любое обращение к нему (кроме проверки на null) приводит к ошибке. Вроде бы все к этому привыкли, но для сравнения Haskell (да и та же Scala) заставлют использовать специальные типы (Maybe в Haskell, Option в Scala) для обёртки функций, которые в других языках могли бы вернуть null. В итоге про Haskell часто говорят «скомпилировать программу на нём сложно, но если всё-таки получилось, значит скорее всего она работает корректно».

С другой стороны, мейнстримовые языки, очевидно, не являются динамически типизированными, а значит не обладают такими свойствами, как простота интерфейсов и гибкость процедур. В итоге писать в стиле Python или Lisp также становится невозможным.

Какая разница, как называется такая типизация, если все правила всё равно известны? Разница в том, с какой стороны подходить к проектированию архитектуры. Существует давний спор, как строить систему: делать много типов и мало функций, или мало типов и много функций? Первый подход активно используется в Haskell, второй в Lisp. В современных объектно-ориентированных языках используется что-то среднее. Я не хочу сказать, что это плохо — наверное у него есть свои плюсы (в конце концов не стоит забывать, что за Java и C# стоят мультиязыковые платформы), но каждый раз приступая к новому проекту я задумываюсь, с чего начать проектирования — с типов или с функционала.

И ещё...

Я не знаю, как моделировать задачу. Считается, что ООП позволяет отображать в программе объекты реального мира. Однако в реальности у меня есть собака (с двумя ушами, четырмя лапами и ошейником) и счёт в банке (с менеджером, клерками и обеденным перерывом), а в программе — ВыгулМенеджер, СчётФабрика… ну, вы поняли. И дело не в том, что в программе есть вспомогательные классы, не отражающие объекты реального мира. Дело в том, что *поток управления изменяется*. ВыгулМенеджер лишает меня удовольствия от прогулки с собакой, а деньги я получаю от бездушного БанкСчёта (эй, где та милая девушка, у которой я менял деньги на прошлой неделе?).

Может быть я сноб, но мне было гораздо приятней, когда данные в компьютере были просто данными, даже если описывали мою собаку или счёт в банке. С данными я мог сделать то, что удобно, без оглядки на реальный мир.

Я также не знаю, как правильно декомпозировать функционал. В Python или C++, если мне нужна была маленькая функция для преобразования строки в число, я просто писал её в конце файла. В Java или C# я вынужден выносить её в отдельный класс StringUtils. В недо-ОО-языках я мог объявить ad hoc обёртку для возврата двух значений из функции (снятую сумму и остаток на счету). В ООП языках мне придётся создать полноценный класс РезультатТранзакции. И для нового человека на проекте (или даже меня самого через неделю) этот класс будет выглядеть точно таким же важным и фундаментальным в архитектуре системы. 150 файлов, и все одинаково важные и фундаментальные — о да, прозрачная архитектура, прекрасные уровни абстракции.

Я не умею писать эффективные программы. Эффективные программы используют мало памяти — иначе сборщик мусора будет постоянно тормозить выполнение. Но чтобы совершить простейшую операцию в объектно-ориентированных языках приходится создавать дюжину объектов. Чтобы сделать один HTTP запрос мне нужно создать объект типа URL, затем объект типа HttpConnection, затем объект типа Request… ну, вы поняли. В процедурном программировании я бы просто вызвал несколько процедур, передав им созданную на стеке структуру. Скорее всего, в памяти был бы создан всего один объект — для хранения результата. В ООП мне приходится засорять память постоянно.

Возможно, ООП — это действительно красивая и элегантная парадигма. Возможно, я просто недостаточно умён, чтобы понять её. Наверное, есть кто-то, кто может создать действительно красивую программу на объектно-ориентированном языке. Ну что ж, мне остаётся только позавидовать им.