## Забытая история ООП

Большинство парадигм программирования, которые мы используем сегодня, были впервые математически изучены в 1930-х годах с использованием идей лямбда-исчисления и машины Тьюринга, которые представляют собой варианты модели универсальных вычислений (это формализованные системы, которые могут выполнять вычисления общего назначения). Тезис Чёрча-Тьюринга показал, что лямбда-исчисление и машины Тьюринга функционально эквивалентны. А именно, речь идёт о том, что всё, что можно вычислить с использованием машины Тьюринга, можно вычислить и с использованием лямбда-исчисления, и наоборот.

Есть распространённое заблуждение, в соответствии с которым машины Тьюринга могут вычислить всё, поддающееся вычислению. Существуют классы проблем (например — проблема остановки), которые могут быть вычислимыми с использованием машин Тьюринга лишь для некоторых случаев. Когда в этом тексте используется слово «вычислимо», имеется в виду «вычислимо машиной Тьюринга».

Лямбда-исчисление демонстрирует подход применения функций к вычислениям по принципу «сверху вниз». А ленточная машина Тьюринга представляет императивный (пошаговый) подход к вычислениям, реализуемый по принципу «снизу-вверх».

Низкоуровневые языки программирования, вроде машинного кода или ассемблера, появились в 1940-е, а, к концу 1950-х, возникли первые популярные высокоуровневые языки, которые реализовали и функциональный и императивный подходы. Так, диалекты языка Lisp до сих пор широко используются, среди них можно отметить Clojure, Scheme, AutoLisp и так далее. В пятидесятых появились и такие языки, как FORTRAN и COBOL. Они являются примерами императивных высокоуровневых языков, которые всё ещё живы. Хотя надо отметить, что языки семейства C, в большинстве сфер, заменили и COBOL, и FORTRAN.

Корни императивного и функционального программирования лежат в формальной математике вычислений, они появились раньше цифровых компьютеров. Объектно-ориентированное программирование, или ООП (Object Oriented Programming, OOP), пришло позже, оно берёт начало в революции структурного программирования, которая происходила в шестидесятых-семидесятых годах прошлого века.

Первый известный мне объект был использован Айвеном Сазерлендом в его судьбоносном приложении Sketchpad, созданном между 1961 и 1962, описанном им в этой работе в 1963 году. Объекты представляли собой графические знаки, выводимые на экране осциллографа (возможно это — первый в истории случай использования графического компьютерного монитора), и поддерживающие наследование через динамических делегатов, которые Айвен Сазерленд назвал в своей работе «мастер-объектами» (masters). Любой объект мог стать мастер-объектом, дополнительные экземпляры объекта были названы «реализациями» (occurrences). Это сделало систему Sketchpad обладателем первых из известных языков программирования, который реализовал прототипное наследование.

Первым языком программирования, широко известным как «объектно-ориентированный», был язык Simula, спецификации которого были разработаны в 1965 году. Как и Sketchpad, Silmula предусматривал работу с объектами, но также включал в себя классы, наследование, основанное на классах, подклассы, и виртуальные методы.

> *Виртуальным методом называется метод, определённый в классе, который предназначен для того, чтобы подклассы его переопределяли. Виртуальные методы позволяют программам вызывать методы, которые могут не существовать на момент компиляции кода, благодаря задействованию динамической диспетчеризации для определения того, какой конкретный метод нужно вызывать во время выполнения программы. JavaScript имеет динамические типы и использует цепочку делегирования для определения того, какой метод нужно вызвать.В результате, этот язык не нуждается в представлении концепции виртуальных методов программистам. Другими словами, все методы в JavaScript используют диспетчеризацию во время выполнения программы, в результате методы в JavaScript не нужно объявлять как «виртуальные» для обеспечения поддержки этой возможности.*

### Мнение отца ООП об ООП

*«Я придумал термин «объектно-ориентированный», и могу сказать, что я не имел в виду C++». Алан Кэй, конференция OOPSLA, 1997.*

Алан Кэй придумал термин «объектно-ориентированное программирование», имея в виду язык программирования Smalltalk (1972). Этот язык разработали Алан Кэй, Дэн Инглз и другие сотрудники научно-исследовательского центра Xerox PARC в рамках проекта по созданию устройства Dynabook. Язык Smalltalk был более объектно-ориентированным, чем Simula. В Smalltalk всё является объектом, включая классы, целые числа и блоки (замыкания). Первоначальная реализация

языка, Smalltalk-72, не имела возможностей создания подклассов. Эта возможность появилась в Smalltalk-76.

В то время, как Smalltalk поддерживал классы, и, в итоге, создание подклассов, в Smalltalk эти идеи не ставились во главу угла. Это был функциональный язык, на который Lisp повлиял так же сильно, как Simula. По мнению Алана Кэя, отношение к классам как к механизму повторного использования кода — это ошибка. Индустрия программирования уделяет огромное внимание созданию подклассов, отвлекаясь от настоящих преимуществ объектно-ориентированного программирования.

У JavaScript и Smalltalk много общего. Я сказал бы, что JavaScript — это месть Smalltalk миру за неправильное понимает концепции ООП. Оба эти языка поддерживают следующие возможности:

- Объекты.
- Функции первого класса и замыкания.
- Динамические типы.
- Позднее связывание (функции и методы можно заменять во время выполнения программы).
- ООП без системы наследования, основанной на классах.

*«Я сожалею о том, что давным-давно придумал термин «объекты» для этого явления, так как его использование приводит к тому, что многие люди уделяют основное значение идее, которая не так важна, как основная. Основная идея — это обмен сообщениями». Алан Кэй*

В переписке по электронной почте 2003-года Алан Кэй уточнил то, что он имел в виду, когда называл Smalltalk «объектно-ориентированным языком».

*«ООП для меня означает лишь обмен сообщениями, локальное сохранение, и защита, и скрытие состояния, и крайне позднее связывание». Алан Кэй*

Другими словами, в соответствии с идеями Алана Кэя, самыми важными ингредиентами ООП являются следующие:

- Передача сообщений.
- Инкапсуляция.
- Динамическое связывание.

Важно отметить, что Алан Кэй, человек, который изобрёл термин «ООП» и принёс его в массы, не считал важнейшими составными частями ООП наследование и полиморфизм.

***Сущность ООП***

Комбинация передачи сообщений и инкапсуляции служит нескольким важным целям:

- Уход от разделяемого мутабельного состояния объекта благодаря инкапсуляции состояния и изоляции других объектов от локальных изменений его состояния. Единственный способ повлиять на состояние другого объекта заключается в том, чтобы попросить его (а не отдать ему команду) об изменении, отправив ему сообщение. Изменения состояния контролируются на локальном, клеточном уровне, состояние не делается доступным другим объектам.
- Отделение объектов друг от друга. Отправитель сообщения слабо связан с получателем посредством API для работы с сообщениями.
- Адаптируемость и устойчивость к изменениям во время выполнения программы посредством позднего связывания. Адаптация к изменениям во время выполнения программы даёт множество значительных преимуществ, который Алан Кэй считал очень важными для ООП.

Источниками вдохновения Алана Кэя, высказавшего эти идеи, стали его познания в биологии, и то, что ему было известно об ARPANET (это — ранняя версия интернета). А именно, речь идёт о биологических клетках и об отдельных компьютерах, подключённых к сети. Даже тогда Алан Кэй представлял себе, как программы выполняются на огромных, распределённых компьютерах (интернет), в то время как индивидуальные компьютеры действуют как биологические клетки, независимо работая со своим собственным изолированным состоянием и обмениваясь данными с другими компьютерами путём отправки сообщений.

*«Я понял, что метафора клетки или компьютера поможет избавиться от данных[...]». Алан Кэй*

Говоря «поможет избавиться от данных», Алан Кэй, конечно, знал о проблемах, вызванных разделяемым мутабельным состоянием, и о сильной связанности, причиной которой является общий доступ к данным. Сегодня эти темы у всех на слуху. Но в конце 1960-х программисты ARPANET были недовольны необходимостью выбирать представление модели данных для своих программ до начала разработки программ. Разработчики хотели уйти от этой практики, так как, заранее загоняя себя в рамки, определяемые представлением данных, сложнее изменить что-то в будущем.

Проблема заключалась в том, что разные способы представления данных требовали, для доступа к ним, разного кода и разного синтаксиса

в используемых в некий момент времени языках программирования. Святым Граалем здесь был бы универсальный способ для доступа к данным и для управления ими. Если все данные выглядели бы для программы одинаково, это решило бы множество проблем разработчиков, касающихся развития и сопровождения программ.

Алан Кэй пытался «избавиться» от идеи, в соответствие с которой данные и программы были, в каком-то смысле, самостоятельными сущностями. Они не рассматриваются таковыми в List или в Smalltalk. Тут нет разделения между тем, что можно делать с данными (со значениями, переменными, структурами данных, и так далее) и программными конструкциями вроде функций. Функции — это «граждане первого класса», а программам разрешено меняться во время их выполнения. Другими словами, в Smalltalk к данным нет особого, привилегированного отношения.

Алан Кэй, кроме того, рассматривал объекты как алгебраические структуры, что давало определённые, математически доказуемые, гарантии их поведения.

*«Моё математическое образование позволило мне понять, что каждый объект может иметь несколько алгебраических моделей, связанных с ним, что могут быть целые группы подобных моделей, и что они могут быть очень и очень полезными». Алан Кэй*

Было доказано, что так оно и есть, и это сформировало базу для объектов, таких, как промисы и линзы, причём, и на то, и на другое оказала влияние теория категорий.

Алгебраическая природа того, как Алан Кэй видел объекты, позволила бы объектам обеспечить формальную верификацию, детерминистическое поведение, улучшило бы тестируемость, так как алгебраические модели — это, в сущности, операции, которые подчиняются нескольким правилам в форме уравнений.

На жаргоне программистов «алгебраические модели» — это абстракции, созданные из функций (операций), которым сопутствуют определённые правила, приводимые в жизнь модульными тестами, которые эти функции должны пройти (аксиомы, уравнения).

Эти идеи были на десятилетия забыты в большинство объектно-ориентированных языков семейства C, включая C++, Java, C# и так далее. Но эти идеи начинают поиски обратного пути, в свежие версии наиболее широко используемых объектно-ориентированных языков.

По этому поводу кто-то может сказать, что мир программирования открывает заново преимущества функционального программирования, и привести рациональные доводы в контексте объектно-ориентированных языков.

Как JavaScript и Smalltalk ранее, большинство современных объектно-ориентированных языков становится всё более и более «мультипарадигменными». Нет причины выбирать между функциональным программированием и ООП. Когда мы смотрим на историческую сущность каждого из этих подходов, они выглядят не только как совместимые, но и как дополняющие друг друга идеи.

**Что, в соответствии с мыслями Алана Кэя, является самым главным в ООП?**

- Инкапсуляция.
- Передача сообщений.
- Динамическая привязка (возможность программ развиваться и адаптироваться к изменениям во время их выполнения).
-

**Что в ООП несущественно?**

- Классы.
- Наследование, основанное на классах.
- Особое отношение к объектам, функциям или данным.
- Ключевое слово `new`.
- Полиморфизм.
- Статическая типизация.
- Отношение к классам как к «типам».

Если вы знаете Java или C#, вы можете подумать, что статическая типизация или полиморфизм — это важнейшие ингредиенты ООП, но Алан Кэй предпочитает иметь дело с универсальными шаблонами поведения в алгебраической форме. Вот пример, написанный на Haskell:

```
fmap :: (a -> b) -> f a -> f b
```

Это — сигнатура универсального функтора map, который работает с неопределёнными типами a и b, применяя функцию от a к b в контексте функтора a для того, чтобы создать функтор b. «Функтор» — это слово из математического жаргона, смысл которого сводится к «поддержке операции отображения». Если вы знакомы с методом [].map() в JavaScript, то вы уже знаете о том, что это значит.

Вот пара примеров на JavaScript:

```
// isEven = Number => Boolean
const isEven = n => n % 2 === 0;
const nums = [1, 2, 3, 4, 5, 6];
// метод map принимает функцию `a => b`
// и массив значений `a` (через `this`)
// он возвращает массив значений `b`
// в данном случае значения`a` имеют тип `Number`,
// а значения `b` тип `Boolean`
const results = nums.map(isEven);
console.log(results);
// [false, true, false, true, false, true]
```

Метод map() является универсальным, в том смысле, что a и b могут иметь любой тип, и этот метод без проблем справляется с подобной ситуацией, так как массивы — это структуры данных, которые реализуют алгебраические законы функторов. Типы для map() не имеют значения, так как этот метод не пытается работать с соответствующими значениями напрямую. Вместо этого он использует функцию, которая ожидает и возвращает значения соответствующих типов, корректных с точки зрения приложения.

```
// matches = a => Boolean
// здесь `a` может быть любого типа, поддерживающего сравнение
const matches = control => input => input === control;
const strings = ['foo', 'bar', 'baz'];
const results = strings.map(matches('bar'));
console.log(results);
// [false, true, false]
```

Взаимоотношения универсальных типов может быть сложно правильно и полно выразить в языках вроде TypeScript, но это очень просто сделать в системе типов Хиндли-Милнера, применяемой в Haskell, поддерживающей типы высших родов (типы типов). Большинство систем типов предусматривают слишком сильные ограничения для того, чтобы позволить свободное выражение динамических и функциональных идей, таких, как композиция функций, свободная композиция объектов, расширение объектов во время выполнения программы, применение комбинаторов, линз и так далее. Другими словами? статические типы часто усложняют написание ПО с использованием методов компоновки.

Если ваша система типов отличается слишком большим числом ограничений (как в TypeScript или в Java), то вы, для достижения тех же

целей, вынуждены писать более сложный код, чем при использовании языков с более свободным подходом к типизации. Это не значит, что использование статических типов — это неудачная идея, или что все реализации статических типов характеризуются одинаковыми ограничениями. Я, например, сталкивался с гораздо меньшим количеством проблем, работая с системой типов Haskell.

Если вы — фанат статических типов и не против ограничений — желаю вам семь футов под килем. Но если вы обнаружили, что некоторые из высказанных здесь идей сложно реализуемы из-за того, что непросто типизировать функции, полученные путём композиции других функций, и составные алгебраические структуры, тогда вините систему типов а не идеи. Водителям нравятся удобства, которые дают им рамные внедорожники, но никто не жалуется на то, что они не летают. Для полёта нужно транспортное средство, у которого больше степеней свободы.

Если ограничения упрощают ваш код — это замечательно! Но если ограничения принуждают вас к написанию более сложного кода, то, возможно, что-то не так с этими ограничениями.

**Что такое «объект»?**

Слово «объект», со временем, приобрело множество побочных оттенков значения. То, что мы называем «объектами» в JavaScript — это просто составные типы данных, без намёков на что-то из программирования, основанного на классах, или на идеи Алана Кэя о передаче сообщений.

В JavaScript эти объекты могут поддерживать, и часто поддерживают, инкапсуляцию, передачу сообщений, разделения поведения через методы, даже полиморфизм с использованием подклассов (хотя и с использованием цепочки делегирования, а не диспетчеризации, основанной на типе).

Алан Кэй хотел избавиться от различия между программой и её данными. JavaScript, в некоторой степени, достигает этой цели, помещая методы объектов туда же, где находятся свойства, хранящие данные. Любому свойству, например, можно назначить любую функцию. Конструировать поведение объекта можно динамически и менять смысловое содержание объекта во время выполнения программы.

Объект — это всего лишь составная структура данных, и ему не нужно ничего особенного для того, чтобы считаться объектом. Однако программирование с использованием объектов не ведёт к тому, что такой код оказывается «объектно-ориентированным», так же, как использование функций не делает код «функциональным».

**ООП больше не является настоящим ООП**

Так как понятие «объект» в современных языках программирования означает гораздо меньше, чем означало для Алана Кэя, я использую слово «компонент» вместо слова «объект» для описания правил настоящего ООП. Многими объектами владеет и управляет напрямую некий сторонний по отношению к ним код на JavaScript, но компоненты должны инкапсулировать собственное состояние и контролировать его.

**Вот что такое настоящее ООП:**

- Программирование с использованием компонентов (Алан Кэй называет их «объектами»).
- Состояние компонента должно быть инкапсулировано.
- Для коммуникации между сущностями используется передача сообщений.
- Компоненты можно добавлять, изменять и заменять во время выполнения программы.

Большинство поведений объектов можно задать в универсальном виде с использованием алгебраических структур данных. Тут нет необходимости в наследовании. Компоненты могут повторно использовать поведения из общедоступных функций и импортируя модули, при этом у них нет необходимости делать общедоступными свои данные.

Манипулирование объектами в JavaScript или использование наследования, основанного на классах, не означает, что некто занимается ООП-программированием. А вот использование компонентов такими способами — означает. Но от устоявшихся представлений о терминах очень сложно отвязаться, поэтому, возможно, нам надо оставить термин «ООП» и назвать то, для чего используются вышеописанные «компоненты», «программированием, ориентированным на сообщения» (Message Oriented Programming, MOP)? Ниже мы будем пользоваться термином «MOP», говоря о программировании, ориентированном на сообщения.

По случайности, английское слово «mop» переводится как «швабра», а их, как известно, используют для наведения порядка.

**На что похоже хорошее MOP?**

В большинстве современных программ имеется некий пользовательский интерфейс (User Interface, UI), ответственный за

взаимодействие с пользователем, некий код, занятый управлением состоянием приложения (данными пользователя), и код, работающий с системой или отвечающий за обмен данными с сетью. Для обеспечения работы каждой из этих систем могут понадобиться долгоживующие процессы, такие, как прослушиватели событий. Тут понадобится и состояние приложения — для хранения чего-то вроде сведений о сетевых соединениях, о положении дел с элементами управления интерфейса и о самом приложении.

Хорошее МОР означает, что, вместо того, чтобы все подобные системы имели бы доступ к состоянию друг друга и могли бы им напрямую управлять, они взаимодействуют друг с другом через сообщения. Когда пользователь щёлкает по кнопке «Сохранить», может быть диспетчеризовано сообщение `"SAVE"`. Компонент приложения, отвечающий за управление состоянием, может интерпретировать это сообщение и перенаправить его к обработчику, ответственному за обновление с состояния (такому, как чистая функция-редьюсер). Возможно, после обновления состояния, компонент, отвечающий за управление состоянием, диспетчеризует сообщение `"STATE_UPDATED"` компоненту пользовательского интерфейса, который, в свою очередь, интерпретирует состояние, решит, какие части интерфейса нужно обновить, и передаст обновлённое состояние подкомпонентам, которые ответственны за работу с конкретными элементами интерфейса.

Между тем, компонент, отвечающий за сетевые соединения, может наблюдать за подключением пользователя к другому компьютеру в сети, прослушивать сообщения и диспетчеризовать обновлённое представление состояния для сохранения его на удалённой машине. Подобный компонент отвечает за работу с сетевыми механизмами, знает о том, работает соединение или нет, и так далее.

Подобные системы приложения не должны знать подробности о других его частях. Они должны заботиться лишь о решении собственных задач. Компоненты системы можно разбирать и собирать как конструктор. Они реализуют стандартизированные интерфейсы, а это значит, что они могут взаимодействовать друг с другом. До тех пор, пока общеизвестные требования к интерфейсу компонентов выполняются, такие компоненты можно заменять другими, с такими же интерфейсами, но делающими то же самое по-другому, или выполняющими, принимая те же сообщения, нечто совершенно иное. Менять одни компоненты на другие можно даже во время выполнения программы — это её работу не нарушит.

Компоненты некоей программной системы даже не должны находиться на одном и том же компьютере. Система может быть

децентрализованной. Сетевое хранилище может разместить данные в децентрализованной системе хранения данных вроде IPFS, в результате пользователь оказывается независимым от исправности некоей конкретной машины, которая обеспечивает сохранность его данных. При таком подходе данные оказываются надёжно сохранёнными и защищёнными от злоумышленников.

ООП, отчасти, появилось под воздействием идей ARPANET, а одной из целей этого проекта было создание децентрализованной сети, которая будет устойчива к атакам наподобие ядерного удара.

Хорошая MOP-система может характеризоваться похожим уровнем устойчивости, используя компоненты, которые поддерживают «горячую замену» во время работы приложения. Она сможет продолжить функционирование в том случае, если пользователь работает с ней с сотового телефона и оказался вне зоны действия сети из-за того, что въехал в туннель. Если ураган нарушил электропитание одного из дата-центров, в котором расположены её серверы, она тоже продолжит функционировать.

Настало время, чтобы мир программного обеспечения освободился бы от неудачного эксперимента с наследованием, основанным на классах, и принял бы математические и научные принципы, которые стояли у истоков ООП.

Пришло время, чтобы мы, разработчики, создавали бы более гибкие, устойчивые, красивые программы, используя гармоничное сочетание MOP и функционального программирования.

Кстати, акроним «MOP» уже используется, описывая «программирование, ориентированное на мониторинг» (Monitoring Oriented Programming), но эта концепция, в отличие от ООП, просто тихо исчезнет.

Материал взят из источника – «Забытая история ООП»:
https://vk.com/wall-143503361_89507