Однажды меня попросили подготовить рекомендации по программированию на Scala для сотрудников нашей компании Solar. В поисках лучших рецептов приходилось заглядывать в самые пыльные уголки Интернета, и в одном из них, в старом проржавелом сундуке я заметил старинный конверт. Просто из любопытства решил заглянуть туда, благо конверт был уже вскрыт. Ветхие страницы, поблекшие чернила, но содержание настолько меня заворожило, что я просто не мог не поделиться со всеми!

К сожалению, не удалось установить ни автора, ни адресата. Местами текст совершенно невозможно было прочитать, так что пришлось додумывать самому. Но даже несмотря на потери, письмо всё ещё достойно Вашего внимания. Для удобства чтения разбил письмо на озаглавленные части, а свои комментарии оставил в [`таких скобочках`].

> Приятного чтения!

Кредо

Приветствую тебя, дорогой друг!

Приятно было получить твоё письмо и узнать, что у вас… [`Фрагмент утрачен`]

В письме ты интересуешься кредо, чем оно способно помочь в достижении наших целей. Я расскажу тебе всё, что знаю сам.

> **Ничто не истинно, всё дозволено.**

Это может показаться циничной доктриной. Но *кредо ##* всего лишь отражение природных вещей. Тезис «Ничто не истинно» подразумевает, что основы, на которых держится современная разработка, зыбки, и мы сами должны строить свои программы. Говоря «Всё дозволено», мы подразумеваем, что сами решаем, что нам делать, и несём ответственность за последствия, какими бы они ни были.

Кредо – это то что необходимо держать в голове и вспоминать, когда необходимо принимать какое-либо решение. В былые времена во время обряда посвящения произносилась такая формула:

> — Когда остальные слепо следуют за популярными парадигмами, помни…
> — Ничто не истинно…
> — Когда остальные ограничены набором паттернов и «лучших практик», помни…
> — Всё дозволено…

В каждый момент мы полагаемся только на собственное чутье и то, что нам известно в данный момент. Завтра ты можешь внезапно осознать, что ошибся вчера, но это

нормально! Такое открытие обязательно пригодится тебе в будущем. Принимая неудачные решения бессмысленно оправдываться чужим опытом, авторитетными мнениями, или просто привычкой. Ты несёшь персональную ответственность за все сделанные выборы и должен уметь обосновывать их каждому, а прежде всего, самому себе.

Ничто не истинно

В нашем цеху мы занимается автоматизацией различных процессов, что даёт людям возможность свободнее распоряжаться своим временем. Наше с тобой призвание – нести людям свободу.

Но нужно помнить, что сама наша работа отнимает время у всех, кто к ней причастен. Значит и здесь нам нужно обеспечивать максимальную свободу как товарищам по цеху, так и самим себе. В частности, стоит избегать любых необоснованных ограничений на написание программного кода.

В этом смысле, все паттерны программирования – зло. Нет, не сами по себе, конечно, среди них встречаются полезные примеры, с которыми стоит познакомится каждом неофиту. Но вот их навязывание, постоянное упоминание в книгах, публичных обсуждениях, приводит к тому, что в общественном сознании закрепляется ошибочное мнение, будто бы знание паттернов необходимо для качественного программирования! Будь осторожен – это всё происки [`Abstergo? Ubisoft? Невозможно разобрать`].

На самом деле частые мысли о паттернах приводит к формированию шаблонного мышления, развитию эффекта «туннельного зрения». Вместо прямого решения поставленной задачи, программист будет стараться натянуть её на полюбившийся ему паттерн. Это явление может показаться даже забавным, когда сталкиваешься с ним впервые, но такая беда встречается повсеместно! И если у опытных программистов получаются просто многословные и хрупкие решения, то начинающие способны сходу снести «правильными», но неуместными паттернами все заслоны тестирования.

Ещё в некоторых командах принято жёстко фиксировать правила форматирования кода – как расставлять скобки, отступы, табуляции внутри строки и т.п. Или хуже, настраивают в среде разработке механизмы автоматического форматирования. Оправдывается это зачастую тем, что строгие правила призваны сэкономить команде время на споры о «красоте кода», предотвратить возможные конфликты. Но так ли это ценно? Ведь, как известно, в споре рождается истина, а с «конфликтностью» стоит бороться путём повышения культуры общения! Встречал и такое, что опытные программисты теряют способность воспринимать чужой код, не укладывающийся в привычный формат.

То же самое касается и утверждения какого-либо конкретного стиля кодирования, что особенно критично для языка программирования Scala, допускающего множество разных стилей. Подобные ограничения не только приводят к невыразительному коду, что само по себе косвенно снижает качество программы, но, что гораздо хуже, ограничивают развитие навыков программирования.

Однако, не все «истины» навязываются извне. Нередко случается, что программист сам ограничивает себя выработанными привычками. И это не всегда плохо – полезные привычки бывают действительно… полезны! Вот только эту самую «полезность» стоит регулярно перепроверять, не потерялась ли она в связи с обретением новых знаний и навыков. И, к сожалению, такая привычка встречается не у каждого программиста.

Все подобные ограничения не оставляют места творческой составляющей, отнимают необходимость самостоятельно видеть, оценивать и выбирать лучше решения. Лишенный свободы выбора человек деградирует, теряет себя.

Всё дозволено

Если ничто не истинно, чем же тогда руководствоваться в принятии решений? Конечно же, в любом случае нужно выполнить техническое задание, удовлетворить всем функциональным и нефункциональным требованиям. Поэтому я расскажу только об оформлении этих решений – о стиле, формате и прочих мелочах.

Программируя, необходимо по-максимуму использовать все знания имеющиеся на данный момент. Но нужно пропускать эти знания через себя, каждый раз анализировать, на сколько они применимы для решения текущей задачи. Возможно уже завтра (после ревью, или здорового сна) ты убедишься, что можно было бы сделать лучше, но сейчас – это не важно.

Объективные критерии качества кода определяет многовековой опыт человечества. Но эти критерии не высечены на каменных скрижалях – они ежедневно меняются, подстраиваясь под всё новые знания. Причём в каждый момент времени сосуществуют несколько взаимоисключающих подходов. И всё же среди них можно найти общие черты.

Найдя решение задачи мы оформляем его таком виде, чтобы,

- во-первых, как можно понятнее донести до читателя суть описываемой логики,
- во-вторых, упростить и обезопасить внесение любых изменений.

Чтобы удовлетворить первому пункту нужно (всего лишь) уметь выражать свои мысли. Определяя свои термины, моделирующие конкретную предметную область, мы комбинируем их с помощью синтаксических конструкций языка программирования. В результате получается текст, одной из основных целей которого является простоя и

точная передача знаний о логике программы как другим программистам, так и самому себе. И тут могут выручить проверенные веками принципы написания грамотных литературных сочинений.

Выразительные идентификаторы

Прежде всего, программистам полезно научиться точно формулировать понятия предметной области, которые он использует в своих алгоритмах. Идентификаторы типов, сущностей, отношений должны легко читаться и однозначно указывать на свою семантику. Недопустимо предлагать читателям текст, составленный из безликих слов вида `Item`, `receive`, или вообще из бессмысленных наборов букв – `i`, `fo`, и т.п…

Так уж повелось, что термины мы формируем из слов [`какого-то «мёртвого языка»; латынь?`], разделяяИхЗаглавнымиБуквами. Поэтому полезно хоть немного знать лексику и грамматику этого языка, чтобы не написать `styleForm` вместо `formStyle`, или `forSeal` вместо `forSale`. Ну и всегда можно воспользоваться услугами толмачей.

Иногда не удаётся выразить смысл сущности в коротком идентификаторе. Это не страшно. Всё равно гораздо лучше читать какой-нибудь `selectedElephantsForAlpinCampaign`, нежели «ленивое» `yoprst`.

Сложные предложения

В классических художественных произведениях можно встретить громоздкие предложения, которые даже не всегда умещаются на одной странице. Не всякий читатель удержит в голове всё хитросплетение вложенных сложноподчинённых предрожений и поймёт основную мысль, которую хотел передать автор. Это достаточно стандратный художественный приём, применяемый совершенно осознанно, и его использовние совершенно уместно в некоторых художественных произведениях.

Но врядли кто-то найдёт оправдание этого приёма в продуктовой разработке. Наша задача состоит не в том, чтобы запутать читателя, но, наоборот, как можно короче и понятнее передать ему суть алогритма. Тут нужны короткие, но ёмкие фразы из всего нескольких терминов. Детали же этих терминах раскрываются уже в других «предложениях». Нужно стремиться избегать как перечисления большого количества операций, так и избыточной вложенности, которая обычно выглядит как «ёлочка» (лежащая на боку).

Один из самых лучших способов достижения простоты является декомпозиция. Достаточно разбить сложный алгоритм на небольшие предложения, имеющие самостоятельный смысл, назначить им подходящие по смыслу идентификатры, и тогда исходная программа представляется в виде простой комбинации вызовов всего

нескольких подпрограмм! Конечно, такой приём требует некторых усилий, но ведь это и есть наша работа - упрощать жизнь всем, кому достанутся результаты наших трудов.

Иногда встречается код, в котором вложенность удаётся устранить даже не прибегая к декомпозиции. Типичный пример - выполнение некоторых действий, только в случае, если выполняется некоторое условие. Частенько встречаются функции с телом вида

```
if (условие) {
  Большое,
  очень большое
  описание
  `счастливого пути`
} else
  `однотрочный несчастливый вариант`
```

В этом случае вложенность совершенно не обоснованна. Её можно устранить, применив механизм раннего выхода:

```
if (!условие)
  return `однотрочный несчастливый вариант`

Большое,
очень большое
описание
`счастливого пути`
```

Тавтология

Табулирование шаблонного кода

секреты скорочтения

Длинные строки – плохо

Чтение “сверху вниз”, а не “слева на право”

Отступы (перечисление параметров методов)

Неоправданные

знаки препинания и императивные операторы – это усложнение!

Вызов функции – самое простое

Инфиксная форма – “подлежащее-сказуемое-дополнение”

ООП и АККА

for/цепочки/комбинаторы

Стрелки

Операторы/инфиксная форма

Контексты, вывод типов и логическое программирование

Вычислительная эффективность, отказ от преждевременной оптимизации

Эклектика

Ракфторинг

PS

Нет абсолютных истин. В том числе это послание – лишь личное мнение, синтезированное на основе ограниченного опыта, как чужого, так и собственного. Возможно сам оспорю всё это уже завтра.

Нельзя навязать стремление к свободе. Общайся, убеждай, но всегда будь готовым к тому, что чужая истина может показаться тебе «более истинной».

Слова, вообще, лгут.

Когда тебя убеждают, что одна парадигма программирования лучше другой, помни:
Ничто не истинно!

Когда тебе не рекомендуют пользоваться шаблонными решениями, помни
Всё дозволено!

~~Да направит тебя Отец Понимания!~~
Да прибудет с тобой сила!
[Точно не уверен, но посыл примерно такой]