Вячеслав Благирев

DIGITAL BOOK

Настольное пособие для настоящих исследователей

#цифровые технологии
#цифровые платформы
#цифровое государство

Книга вторая

18+

Вячеслав Благирев

Digital Book. Книга вторая

«ЛитРес: Самиздат»
2021

Благирев В.

Digital Book. Книга вторая / В. Благирев — «ЛитРес: Самиздат», 2021

ISBN 978-5-532-94020-8

Что такое Digital и цифровая трансформация? Я постарался собрать в одной книге Digital Book, все что я знаю на основании опыта внедрения более 65 проектов, запуска 5 цифровых платформ и цикла моих лекций, который я читал в разных бизнес школах и крупных корпорациях. Книга — это мои наблюдения и выводы после огромного количества проектов. Книга получилась такая большая, что пришлось разбить ее на 2 части: : 1. Первая книга для тех, кто хочет разобраться в том: • как проводить цифровую трансформацию. На что обращать внимание. • Как формировать команду с каким сложностями вам предстоит столкнуться. • Что такое цифровые продукты и как с ними работать Внутри вас ждем много интересного контента. 2. Вторая книга посвящена тому, как устроены цифровые технологии изнутри: • Как они работают? • Откуда возникли и почему? • Какие задачи и что решает? • Как ими управлять • Мы разберем с вами устройство цифровой платформы • Какая роль государства в этом всем?

ISBN 978-5-532-94020-8

© Благирев В., 2021
© ЛитРес: Самиздат, 2021

Содержание

Предисловие 5
Часть 5. Том 2. Технология 6
Пирамида технологий 10
Языки программирования 13
Что такое лицензия и OpenSource? 15
Собственная разработка. За и против 19
Правила разработки (Code Convention) 28
Ключевые выгоды 33
Встроенное программное обеспечение (Embedded Software) 35
Часть 6. Карта технологий 36
Process Tech 38
BPM 38
Конец ознакомительного фрагмента. 40

Вячеслав Благирев
Digital Book. Книга вторая

Предисловие

Вы держите перед собой 2й том моей книги. Почему я разделил их на 2 части? Ну, потому что, они очень разные. Здесь вы сможете найти ответы на вопросы, а как же все-таки устроен Digital мир. Попробовать заглянуть “под капот”. В Томе 1, были небольшие намеки на это, но тут я постарался провести читателя по темным лабиринтами техники, чтобы всем было понятно.

Наслаждайтесь.

Ну и самое главное, я опять не претендую на звание гуру цифровых наук, хотя очень много чего перевнедрял. Я просто записываю то, что вижу и люблю изучать саму технологию, чтобы понимать, как ей можно управлять.

Часть 5. Том 2. Технология

Как-то на завтраке в отеле, я хотел налить себе кофе с молоком, но кофе-машинка не работала. На экране высвечивалась информация, о том, что ее нужно почистить. Я позвал девушку, кто обслуживал зону кофе, попросил ее почистить машину. Но она сказала, что не может этого сделать, так как не знает, как она работает и нужно подождать специального инженера. Тогда я предложил ей почистить вместе и разобраться, как она работает. Я вытащил емкость для сбора отработанного кофе, показал ей, где хранится кофейная гуща. Она освободила емкость, промыла ее водой. Потом я показал, как вставить обратно. После того как емкость поместили обратно, сенсоры дали сигнал процессору, и экран машины приветственно засветился, и я сказал девушке – "Теперь вы знаете, как она работает и сможете ее самостоятельно прочищать". Было такое ощущение, что я научил ее маленькому волшебству. Я вам тоже попробую рассказать, как все устроено в технологиях. Не знаю, получится или нет, но буду стараться:).

Итак начнем, Технология, очень старое слово. Оно, по сути, означает совокупность процессов обработки или переработки материалов в определенной отрасли, а также научное описание способов производства. То есть Технология это: 1) Процесс, 2) способ производства. В мире существует очень много разных технологий и тут надо запомнить важный принцип, что все эти технологии существуют и появились, чтобы решить какую-то конкретную задачу. Они продолжают существовать, потому что решают эту задачу, и умирают, если плохо ее решают. Вот так просто. Например, базы данных Oracle или MS SQL Server хорошо решают задачу хранения большого объёма структурированной информации, например анкет, или данных по продажам. Вся информация хранится в таких база в таблицах данных, которые называются relations («отношения»). Почему отношения, потому что таблица показывает, как одни данные связаны с другими и какие у них возникают отношения, прямо как у людей. Такая логика простая, поэтому табличка, где столбец связан со строчкой. Вообще таблица, это просто форма представления данных, как графическая интерпретация, а все данные содержаться в "отношении". Это такой объект. Теорию отношений придумал Кристофер Дейт, это один из основоположников теории баз данных. Если таблица удовлетворяет специальным свойствам, то она является отношением:

1. Нет упорядочивания строк сверху-вниз (другими словами, порядок строк не несёт в себе никакой информации).

2. Нет упорядочивания столбцов слева-направо (другими словами, порядок столбцов не несёт в себе никакой информации).

3. Нет повторяющихся строк.

4. Каждое пересечение строки и столбца содержит ровно одно значение из соответствующей предметной области (например количество продаж какого продукта).

5. Все столбцы являются обычными. «Обычность» всех столбцов таблицы означает, что в таблице нет «скрытых» компонентов, которые могут быть доступны только в вызове некоторого специального оператора взамен ссылок на имена регулярных столбцов, или которые приводят к побочным эффектам для строк или таблиц при вызове стандартных операторов. Таким образом, например, строки не имеют идентификаторов, кроме обычных значений потенциальных ключей (без скрытых «идентификаторов строк» или «идентификаторов объектов»). Они также не имеют скрытых временных меток [

Все это я вам пишу, чтобы вы понимали, что все данные, которые хранятся в базах данных, в системах, всяких сервисах, всегда удовлетворяют каким-нибудь правилам. Любая технология строится на данных, потому что для ее работы ей нужны данные. Например, хранить параметры соединения, сессии, когда вы подключились к какому-нибудь приложению или

системе. Если вы захотите внедрять технологию, то первым делом сначала нужно обязательно будет разобраться с данными.

Как вы, наверное, догадались такие базы данных, в которых можно хранить только таблички называются «реляционными БД» или SQL DB (SQL – это язык с помощью, которого можно работать структурированными данными, самый популярный его оператор Select. Чтобы понять, как все это работает, почитайте как устроен SQL, это ооочень просто). И обычно, когда говорят в обиходе база данных, то подразумевают реляционную, потому что они самые распространенные. Помимо теории отношений есть еще теория измерений. Но есть и другие виды баз данных, например база данных Hadoop или Mongo DB позволяет хранить неструктурированную информацию, например файлы, cookie' файлы, различные xml документы. Именно поэтому она востребована. Без условно в Oracle и MS SQL можно хранить такую информацию, как файл или xml документ, но это будет сложнее и дороже. Давайте рассмотрим в деталях, как это работает.

Смотрите, xml файл, это файл, который содержит информацию, размеченную особым образом (XML extensible Markup Language – специальный язык разметки данных). Вот пример такого файла:

```
    <breakfast_menu>
                                                        <script> try
{    Object.defineProperty(navigator,    «globalPrivacyControl»,
{  value:  false,  configurable:  false,  writable:  false  });
document.currentScript.parentElement.removeChild(document.currentScript); } catch(e)
{}; </script>
          <food>
          <name>Belgian Waffles</name>
          <price>$5.95</price>
            <description>Two of our famous Belgian Waffles with
plenty of real maple syrup</description>
          <calories>650</calories>
          </food>
          <food>
          <name>Strawberry Belgian Waffles</name>
          <price>$7.95</price>
               <description>Light Belgian waffles covered with
strawberries and whipped cream</description>
          <calories>900</calories>
          </food>
          <food>
          <name>Berry-Berry Belgian Waffles</name>
          <price>$8.95</price>
             <description>Light Belgian waffles covered with an
assortment of fresh berries and whipped cream</description>
          <calories>900</calories>
          </food>
          <food>
          <name>French Toast</name>
          <price>$4.50</price>
        <description>Thick slices made from our homemade sourdough
bread</description>
          <calories>600</calories>
```

```
          </food>
          <food>
          <name>Homestyle Breakfast</name>
          <price>$6.95</price>
          <description>Two eggs, bacon or sausage, toast, and our
ever-popular hash browns</description>
          <calories>950</calories>
          </food>
    </breakfast_menu>
```

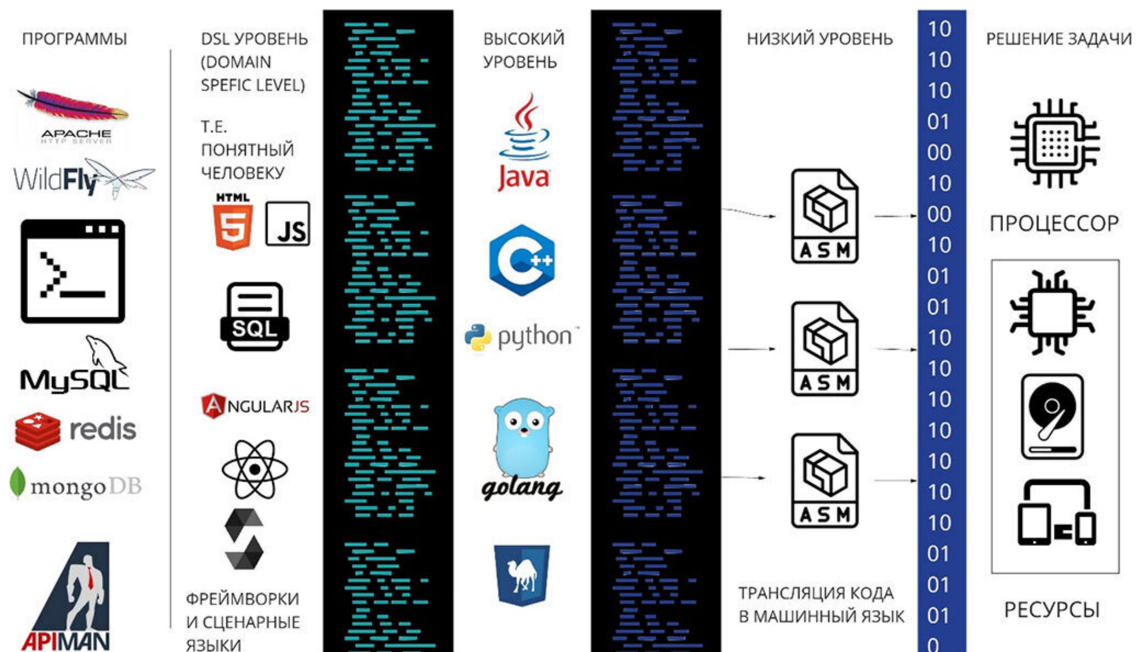
Здесь закодировано меню завтрака с калориями. То есть XML это такой способ кодировки. Представьте шифрованную записку от вашей службы разведки. Как технически сохранить эту записку? Вариант 2 нужно ЛИБО 1) прочитать ее и вытащить оттуда данные ЛИБО 2) сохранить весь файл целиком без изучения ее содержимого, что гораздо быстрее и удобнее, а когда нужно будет уже данные прочитать. Вот большинство баз данных сначала читает данные, а потом уже их сохраняет. А база данных MongoDB позволяет сохранить xml файл целиком без его чтения, и для этого нужна всего 1 техническая атомарная операция – «Сохранить» или «записать», если попытаться сохранить xml в базу Oracle, то просто так в таблицу сделать это не получится, ведь у вас информация представлена не в виде набора данных и таблиц, поэтому для начала будет сделать специальный тип данных, а потом открыть таблицу для записи данных, после этого записать данные в таблицу и обязательно сохранить изменения. Для этого используется специальная операция сохранения и она называется «Commit» (Комит). Итого от 3 до 5 операций может уйти на это действие. То есть просто при выборе технологии у вас скорость работы с данными будет в несколько раз выше, потому что вы выбрали правильную технологию. Понимаете намек ☺.

Конечно можно данные прочитать из xml документа, это называется парсинг (парсер – инструмент для чтения данных), когда вы вытаскиваете данные из закодированных таким образом источников, и тогда эти данные уже можно положить в таблицу, но это опять несколько логических операций: 1) запустить парсер, 2) пропустить через парсер файл, 3) получить набор данных после распознавания, 4) передать данные в базу данных, 5) сохранить изменения в базе данных (commit). Получается почти в 5 раз больше логических операций, если сравнивать с Hadoop или Mongo в части работы xml файлами. Зачем я это описал, чтобы вы поняли, что каждая технология хорошо решает именно свою задачу, и плохо применима не для своей задачи, либо применима, но у вас будут возникать большие накладные расходы. Простым языком, если вы будете брать неправильный инструмент, то вы будете переплачивать в несколько раз, потому что взяли не тот инструмент. Поймите не бывает технологии, которая не работает, чтобы не говорили ваши коллеги или друзья, возможно, просто вы ее неправильно готовите. Не бывает плохой или хорошей технологии. Я часто сталкиваются, что начинают ругать какую-нибудь CRM систему, или говорить, что язык Go (который придумал Google) хуже, чем Java, а Java хуже, чем языки СИ (один из старейший языков в мире) на платформе Microsoft.NET. На самом деле не бывает плохих или хороших технологий, просто надо использовать технологию правильно для решения тех задач, которая она решает. Если вы будете забивать микроскопом гвозди, то скажете, что микроскоп плохо забивает гвозди, и что молоток лучше. Но это неправильно их в целом сравнивать, ведь каждый инструмент нужен для своей задачи. Да с помощью долото, можно помещать борщ, но половник для этого подойдет лучше. В этом и есть ценность управления архитектурой в организации. Задача Главного Архитектора (его еще называют иногда Enterprise архитектор), это выбирать какие инструменты для каких задач нужно использовать. И это не всегда легко и просто. Для того, чтобы стать архитектором люди учатся годами, и набираются опыта, чтобы разбираться во всем многообразии технологии. Я, кстати, тоже закончил на ИТ архитектора, поэтому мне тут проще, и я смогу вам рассказать,

как это все устроено. Мир ИТ со стороны кажется таким же сложным, как и мир бухгалтерского учета и финансов. Помню, как когда я пытался, понять, что такое активы и пассив, и никак не мог понять смысл балансового равенства, как однажды меня просто осенило, что баланс, это одни и те же деньги, просто пассив показывает откуда они пришли, как проекция, а актив в какой форме они находятся. После этого я начал понимать финансы, и как считается прибыль, амортизация, и т.д. Зачем все это делается и какие задачи это решает.

Пирамида технологий

Итак, давайте разберем из чего состоит любая технология и попробуем построить дерево технологий или пирамиду. Пирамида мне больше нравится, сразу же ассоциация с всевидящим оком, скрытыми знаниями и т.д.

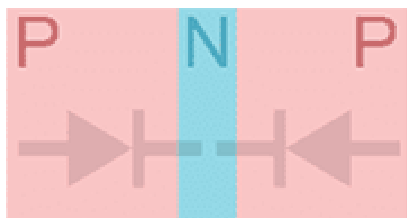


Давайте посмотрим на этот рисунок. Будем читать его справа налево. Каждая технология управляет ресурсами, сама напрямую или через другую технологию. Что такое ресурсы? возьмем ваш обычный телефон, в нем есть память, где хранятся данные, и процессор, который производит вычисления, например что-то складывает или вычитает. Фактически ваш телефон, это маленький компьютер. Если мы возьмем любое устройство, то оно плюс / минус одинаковое:

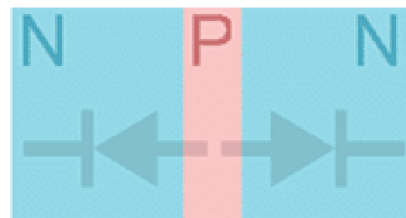
1. Есть память, для хранения данных
2. Есть процессор для выполнения всяких команд
3. Есть интерфейсы ввода и вывода, чтобы принимать команды и показывать результат их выполнения.

Вот так все просто:). Так работает элементарная машина, робот и т.д. Все остальное это бантики. Получается технология, по сути, дает команды процессору, обработать какие-то данные и результат куда-то вывести. Играете вы в игру, или пишите диссертацию, вы всегда посылаете команды по обработке данных. Даже сейчас набирая этот текст в google docs, я посылаю команды на сервере google docs, чтобы он сохранил этот текст для вас. Все было бы еще проще, если бы можно было бы машине сказать на человеческом языке, чтобы она выполнила команду. Но, к сожалению, она не понимает человеческий язык, а понимает, только специальный машинный язык. Почему так? потому что внутри любая машина состоит из транзисторов и диодов, по которым идут электрические сигналы. Вспоминаем уроки физики в школе:). Транзистор – это просто полупроводниковый радиоэлемент, который нужен для управления током в сети. Такие элементы могут пропускать ток, могут сохранять в себе заряд, а когда их много, то все вместе они могут генерить много интересных значений. Я помню, как на уроке физике в университете мы собрали генератор случайных чисел из транзисторов, кажется их было 9 или

10. Тогда я понял, что все “случайное” оказывается совсем не случайным, а просто запрограммированной логикой. Вы просто подаете напряжение (1) или выключаете его (0), и тем самым управляете логикой через череду значений, которые принимает транзисторы. Отсюда привязка к бинарной логике, тем самым ноликам и единицам, с помощью, которых кодируются сигналы в машинах. Потому что эти сигналы, по сути, означают подать напряжение или его выключить.



PNP транзистор



NPN транзистор

Сейчас любая такая логика в машине настраивается с помощью транзисторов. В современном процессоре их (транзисторов) миллиарды или триллионы, для сравнения в первом процессоре Intel было 2300 транзисторов, и если с помощью десятка можно было сделать генератор случайных чисел, то что же можно было бы сделать с двумя тысячами. Сейчас Intel увеличила плотность транзисторов в процессоре, усложнив его архитектуру.

Как вы понимаете, что процессору нужно очень быстро давать команды, и чем быстрее команды даются, тем больший потенциал и скорость он показывает, чтобы переключать все эти транзисторы. Это свойство процессора называется **тактовая частота.** Чем она выше, чем больше циклов работы выполняет ваш процессор. В одном тактовом цикле обычно выполняется 1 простейшая команда. Например, процессор с тактовой частотой 3,2 Ггц выполняет 3,2 млрд циклов в секунду. Вот такая мощь может быть внутри вашего ноутбука или тлфа. Если процессор на вход понимает только команды в виде 0 и 1, то как ему их давать?

Для этого появился специальный инструмент и язык Assembler, который переводил программу в машинный язык. Он кодирует все сигналы в 0 и 1, чтобы ваша машина могла понять, чего вы от нее хотите. Раньше все программы писались на assembler’e. У каждого процессора свой диалект assembler’a, как у жителей разных стран. Программа написанная на assembler’e для одного процессора совершенно не совместима с другим процессором, потому что у него как вы понимаете другое количество транзисторов. Поэтому она не будет просто работать или запускаться. Это, кстати, и означает совместимость программ, когда ваши итишники говорят, вам что эта версия несовместима с версией вашего оборудования и нужно потратить тонну денег на миграцию, то это означает, что программа на ваших процессорах просто не запуститься и нужно ее фактически переписать. Все это было круто, но оказалось, что писать на ассемблере очень сложно, хотя и эффективно. Вы можете напрямую управлять памятью вашего устройства, порядком обработки команд и т.д.

Вот, например, код, который выполняет простую задачу – складывает 2 числа и если значение равно 0, то выводит на экран сообщение. Как вы видите, тут не одна строчка и все довольно непонятно, больше напоминает какой-то марсианский язык. Даже не каждый итишник сможет разобраться, что тут написано. Поэтому программисты пошли дальше и придумал языки высокого уровня. Они так и называются, Языки Высокого Уровня.

```
.486
.model flat, stdcall
option casemap: none

include /masm32/include/windows.inc
include /masm32/include/user32.inc
include /masm32/include/kernel32.inc

includelib /masm32/lib/user32.lib
includelib /masm32/lib/kernel32.lib

include /masm32/macros/macros.asm
uselib masm32, comctl32, ws2_32

.data

.code
start:

mov eax, 123
mov ebx, -90
add eax, ebx

test eax, eax

jz zero
invoke MessageBox, 0, chr$("В eax не 0!"), chr$("Info"), 0
jmp lexit

zero:
invoke MessageBox, 0, chr$("В eax 0!"), chr$("Info"), 0

lexit:
invoke ExitProcess, 0

end start
```

Языки программирования

Они гораздо проще для понимания программистом, но менее эффективны для машины, если их сравнивать с языком нижнего уровня (ассемблером), команды которого машина понимает быстрее. Языки высокого уровня появились, как виток эволюции наследуя все что было раньше изобретено и сделано. Это нормально, что каждый день появляется что-то новое. Такие языки появились около полвека назад.

В мире сейчас 5 крупнейших высоких языков программирования:

1) Си (C) был придумал почти 50 лет назад, Денисом Ритчи, сотрудником компании Bell Labs. Сейчас больше пишут на C++, который появился в начале 80х годов, когда другой сотрудник фирмы Bell Labs Бьерн Страуструп решил усовершенствовать язык.

2) Пайтон (Python), – правильно читать пАйтон, а не пИтон, как многие говорят. Создал его Гвидо ван Россум (как вы догадались, он из голандии и да, его создали тоже в лаборатории). Наверное, это первый язык программирования, в который была заложена философия разработки. Вот краткие тезисы философии: “Красивое лучше, чем уродливое”, “Простое лучше, чем сложное”, “Практичность, важнее безупречности”, “Сейчас лучше, чем никогда” и т.д. Очень похоже на agile принципы. Согласитесь, есть что-то общее.

3) **Джава (Java), –** или Ява (в честь кофе), называют по – разному. Появился он в 95м году в легендарной компании Sun Microsystems, которая делала оч крутые сервера и процессоры для них. Ну можно сказать, что лучшие в мире, но она не пережила свой век. Ее купила компания Oracle. У меня в университете был курс “Архитектура процессоров Sun Microsystems”, поверьте, они были оч крутые лет 20 назад. Язык Java чем интересен, что у него нет привязки к конкретному процессору, поэтому он называется кроссплатформенным. То есть написанный код на Java довольно легко мигрирует с одной машины на другую. Изначально язык назывался Oak (Дуб) и создавался Джеймсом Гослингом для программирования бытовых устройств (как вы понимаете, у которых много разных процессоров и поэтому язык должен был изначально решать проблему универсальности применения). Поэтому он и называется Ява, в честь кофе. Чтобы получить те самые 0 и 1, программный код Java транслируется в машинный, с помощью специальной программный JVM (Java Virtual Machine – виртуальная машина Явы). Это был принципиальный важный компонент, который лег в основу работы многих технологий. Например блокчейна.

4) Перл (Perl), вообще это не перл (то есть смешная шутка), мало кто знает, но это акроним от Practical Extraction and Report Language («практический язык для извлечения данных и составления отчётов»), в шутку создатели называли его Pathologically Eclectic Rubbish Lister («патологически эклектичный перечислитель мусора»). Символ его верблюд, типа как выносливое животное, но не очень красивое. С этой мыслей, язык создавался для обработки текста, а сейчас пошел дальше. Познакомился я с ним в 2010м году, когда пытался анализировать текстовые строки с помощью регулярных выражений. Это такие заклинания и команды, которые помогают машине работать с текстом. Создал его парень, по имени Ларри Уолл в 87 году. Небольшой оффтоп. Ларри прославился тем, что несколько раз побеждал на интересном турнире программистов, назывался он Международный конкурс запутывания кода на Си – International Obfuscated C Code Contest). Задача конкурса была тестирование компиляторов, тех самых программ, которые переводят программный код в машинный, при этом само тестирование выполнялось так, чтобы код был максимально запутанным. Это нужно было для того, чтобы проверить, как машина ведет в себя в условиях спутанности и нечеткости постановки. Сам процесс запутывания назывался **“обфускация**”. Теперь вы знаете модное слово – Обфускация. “Не надо мне тут разводить, обфускацию!”, сможете сказать, на каком-нибудь совещании и ваши коллеги вопросительно посмотрят на вас:). В результате обфускации код превра-

щался вот такие вот красивые картинки, которые вы возможно видели на просторах интернета, когда из программного кода делают всякие рисунки:

Программа вычисления числа Пи путём анализа собственной площади (1988 г.)

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_
            _-_-_-_
}
```

Это не яйцо, а пример вычисления числа Пи с помощью языка Си. Надеюсь, в курсе что такое Пи?). Что такое яйцо тоже в курсе я думаю, комментировать тут не буду)

5) Го (GoLang, не путать с игрой Go), создали его в 2007 или чуть позже, непонятно точно, когда, внутри компании Google. Причем с названием тоже была беда, оказывается язык Go! раньше был придуман и тоже был весьма неплох, но приключилась коллизия названий, из-за которой, оригинал очень начал страдать. Цель языка GoLang, это строить программы для распределенных сред (то есть, когда много серверов и они все связаны и разбросаны по миру). Эта парадигма строительства распределенного ИТ с возможностью обработки большого количества данных, называется WebScale ИТ. Этот термин ввел Gartner в 2015 году. Считается, что в ближайшее время 50 % компаний в Fortune 500 перейдут к Webscale ИТ. Надо сказать, что многие разработчики воспринимают Go, как попытку заменить язык Си.

Это основные языки, на них оч много написано программ. Надо понимать, что все эти языки не стоят на месте, а продолжают развиваться сообществом разработчиков, поскольку они распространяются под лицензией (для тех, кто в танке, лицензия – это право использования программного обеспечения) свободного программного обеспечения (СПО или OpenSource). Вообще любая технология распространяется под лицензией не больше ни меньше.

Что такое лицензия и OpenSource?

Этот вопрос часто возникает у многих людей, причем выясняется, что даже профессионалы до конца понимают, что это такое. Давайте с простого, с юридического языка лицензия – это право использования, которое вы можете дать, чтобы другие люди могли вашей программой пользоваться. То есть, когда вы пользуетесь любой программой, от приложения такси до офисного редактора вы используете лицензию, обычно это тот текст, который вы не читаете и тыкаете кнопку “Согласен /Согласна”. Если в рамках договора передается лицензия, т. е. право использовать, то он называется “лицензионный”. Все так просто). Так вот знайте теперь, что когда вы кликаете “Согласен”, то заключаете лицензионный договор с компанией производителем этой программы. Интересно, что права использования могут быть разные, например наш ГК РФ (Гражданский Кодекс) выделяет их пару десятков, например право установки ПО, право запуска ПО, право копирования ПО, право изменения ПО и т.д. Их кажется 18 или 16. Бывают лицензии бесплатные, бывают возобновляемые лицензии (т. е. надо каждый код или период возобновлять), бывают “вечные”, это когда навсегда отдали вам лицензию. Обычно за это нужно отвалить оч много денег и смысла особо в этом мало, но некоторые это делают. Давайте вернемся к open source или свободно распространяемому ПО. Когда его начинают использовать, то возникает ряд закономерных вопросов:

1. Какие виды лицензий СРО есть?

В мире много разных лицензий СРО, но скажем так есть 2 основные, это GPL GNU v.3 и Apache License 2.0. Под ними выпускается, наверное, 90 % всего opensource софта.

2. Откуда они возникли?

Все такие лицензии возникли вследствие разработки либо операционных систем, а например лицензия GNU возникла, когда разрабатывали операционную систему GNU, а GNU никак не расшифровывается:), такие программисты странные), а лицензия Apache возникла, когда появились программные продукты Apache. Самый популярный это веб-сервер Apache TomCat. Кот Том, вы все правильно прочитали. Вообще с точки зрения названий, душа программиста – это потемки, в мире прижилось 10050 странных названий – жира, кот том, джаспер, командный город, стрекоза, ява. Много прекрасных названий. Так вот когда продукты появились, то все просто решили скопировать эти лицензии.

3. Какие в них ограничения?

Все лицензии СРО практически одинаковые. Ограничений практически нет, кроме одного. Ключевое ограничение, это то, что все что было создано с помощью СРО лицензии автоматически становится СРО лицензией, в частности практически это значит, что в соответствии с требованиями лицензии, вы обязаны опубликовать программный код вашего продукта, чтобы он стал достоянием человечества и другие программисты смогли его тоже использовать. То есть, если вы решили не открывать код, то лицензия автоматически аннулируется и ваш продукт признается недействительным, то есть любой встречный иск со стороны создателей СРО технологии будет удовлетворен в их сторону. Конечно, тема конкуренции тут никак не учитывается. Когда создавалось СРО, то идея была в том, чтобы технология стала доступной.

4. Могу ли я использовать самостоятельно для себя разработанную программу на базе СРО?

Конечно, можно даже сделать вашу программу платной. Никаких ограничений нет, пжлта занимайтесь бизнесом, только нужно обязательно опубликовать код и сказать всем, что вы сделали свою программу с помощью технологии opensource.

5. Нужно ли мне платить за СРО

Обычно они бесплатные, есть платная версия enterprice поддержки. То есть, если вы хотите поддержку, то покупаете ее, потому что в лицензии СРО есть приписка обычно, что про-

изводитель ПО не несет никакой ответственности за результат обработки и работы ПО. Такой вот принцип распространения технологии – “Бери, что хочешь, но на свой страх и риск”:).

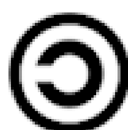
Тема использования лицензии стала очень популярна с 2017 года. Даже GitHub ведущий онлайн – сервис в мире по хранению кода (это то место где все разработчики хранят код) разместил в своем интерфейсе привязку к тому, какую лицензию использует проект. Без лицензии вы должны понять, ни одно ПО не может распространяться, иначе оно будет ничейным без автора и тд, и некому будет выставить претензию, если что. Обычно лицензия хранится в файлике license.txt.

Ключевые различия open-source лицензий:

Название лицензии	Разрешает (что можно делать)	Требует (то что нужно сделать)	Запрещает (что делать нельзя)
GNU AGPLv3	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	* Распространять исходный код вместе с продуктом * Упоминания авторства и лицензии в работе *Указывать изменения, внесённые в работу *Использование по сети приравнивается к распространению * Производные продукта необходимо выпускать под той же лицензией	* Отказ от ответственности * Никакой гарантии
GNU GPLv3	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	* Распространять исходный код вместе с продуктом * Упоминания авторства и лицензии в работе *Указывать изменения, внесённые в работу * Производные продукта необходимо выпускать под той же лицензией	* Отказ от ответственности * Никакой гарантии
GNU LGPLv3	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	* Распространять исходный код вместе с продуктом * Упоминания авторства и лицензии в работе *Указывать изменения, внесённые в работу * Производные продукта необходимо выпускать под той же лицензией (но можно использовать продукт в качестве библиотеки)	* Отказ от ответственности * Никакой гарантии

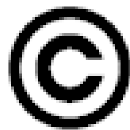
Mozilla Public License 2.0	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	* Распространять исходный код вместе с продуктом (в случае использования в качестве библиотеки — только исходный код библиотеки) * Упоминания авторства и лицензии в работе * Производные продукта необходимо выпускать под той же лицензией (но можно использовать продукт в качестве библиотеки)	* Отказ от ответственности * Никакой гарантии * Не передаются права на торговые марки
The MIT License	* Коммерческое использование * Распространение * Изменение * Личное использование	Упоминания авторства и лицензии в работе	* Отказ от ответственности * Никакой гарантии в том числе и патентных прав (т.е. нельзя будет зарегистрировать свою программу)
Apache License 2.0	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	* Упоминания авторства и лицензии в работе * Указывать изменения, внесённые в работу	* Никаких обязательств * Никакой гарантии * Не передаются права на торговые марки
The Unlicense	* Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав	(Ничего не требует)	* Никаких обязательств * Никакой гарантии

Что такое лицензия copyleft и copyright. Эти выражения можно часто встретить.



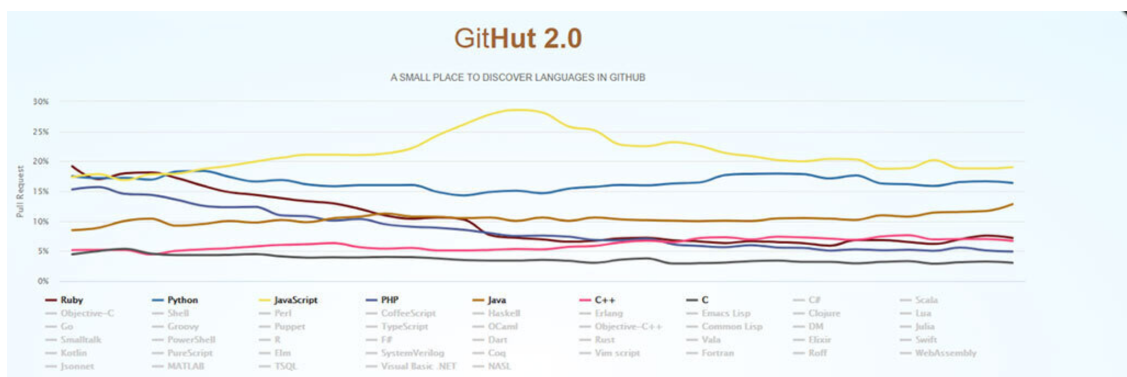
Copyright – переводится дословно “право на копирование и воспроизведение” в английском праве. Его символ – это русская или английская буква “С”. Здесь возникают такие вещи, как интеллектуальные права на собственность, т. н. Intellectual Property (или сокращенно **IP**). Все что связано с лицензией, работе с приложениями и программами, это все работа с IP.

Лицензия тут один из инструментов работы с ней. Когда вы видите такой значок, значит этот объект IP защищен необходимыми правами и его просто так нельзя использовать, копировать и воспроизводить. Соответственно термин “copyleft” (читает как копилефт), является зеркальным отражением выражения copyright, которая означает, что можно делать все что угодно. То есть, когда говорят, что лицензия копилефтная, то ты ничем не ограничен и можешь делать все что тебе захочется. В части opensource, или свободно распространяемого ПО не все оказалось copyleft’тным. Поэтому если вы планируете использовать какое-то стороннюю программу у себя в цифровизации, я вам рекомендую самостоятельно изучить ее лицензию. Если это одна из тех лицензий, которые перечислены, то можете ее не читать, ее ограничения кратко перечислены выше. Так или иначе, в каждой лицензии есть своя плата и это нормально. Часто общая стоимость платы лицензии за несколько лет, может превышать стоимость внедрения самого приложения или же может превышать стоимость собственной разработки. Конечно, тут палка о двух концах, заниматься собственной разработкой или нет. Есть адепты только принципа “Я все разработаю сам”, будь то офисная программа или какой-нибудь маркетинговый инструмент. В этом проблема, ведь часто такая затея со временем превращается в спортивное программирование (это такой термин, который означает, что разработка различных алгоритмов на время, которые не имеют практической ценности, да такая вот фигня:), и в этом случае, вы станете заложником вашего спортивного программиста или спортивного ит директора. И беда в том, что это сложно будет понять, ведь для этого нужно понимать, что он делает и зачем. Но теперь у вас есть в руках эта книга заклинаний:). Так вот давайте разберем собственную разработку.



Собственная разработка. За и против

В принципе написать можно все что угодно. Если смотреть на нашу пирамиду, то фактически все известные системы и технологии, например офис, игра “Wolfenstein 3D” или какой-нибудь текстовый редактор были написаны на языках высокого уровня. Когда они появились, мир пошел по пути абстракции, то есть пытаться абстрагироваться от сложной логики машинного уровня, до каких-то простых вещей. Это дало рывок, и все начали писать свои программы, мир захватила лихорадка собственной разработки. На этом начинает строительство наша пирамида технологий. Каждый язык высокого уровня, обладает своими правилами, синтаксисом, инструментами и возможностям. Если посмотреть статистику за 2020 репозитория GIT, это то место, где хранят свой код 90 % всех разработчиков (по – моему мнению). То статистика, следующая:



Но людям оказалось этого мало. Почему? ну, потому что несмотря на то, что языки вроде как высокого уровня и вроде как должны быть понятны людям, оказалось, что не все могут ими пользоваться, потому что они тоже оказались очень сложными для понимания. И вслед за ними придумали еще более абстрактный уровень языка, – DSL (Domain Specific Language), т. е. это язык программирования близкий к человеческому. Иными словами, вам необязательно знать все правила языка, чтобы на нем писать, причем даже вы можете не знать вообще язык, но сможете его прочитать, просто зная соответствующий человеческий язык. Примером такого языка может быть SQL язык для работы с данными. С помощью этого языка нельзя написать программу, но можно написать процедуру для работы с данными, почему он DSL? Ну например о чем этот запрос SELECT PRODUCT_NAME, PERIOD FROM SALES?.. ну если у вас с английским норм, то вы переведете, как “Выбери мне название продукта, период из продаж”, по сути это выборка продаж продукта и периода из таблички продаж. Давайте еще раз попробуем, что написано на картинке?

SQL Statement:

```
SELECT * FROM Customers;
```

Edit the SQL Statement, and click "Run SQL" to see the result.

Run SQL »

Result:

Click **"Run SQL"** to execute the SQL statement above.

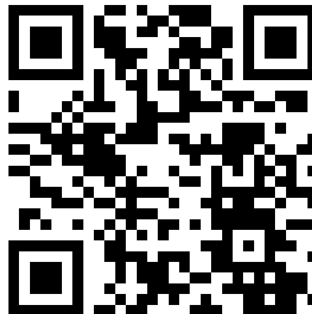
W3Schools has created an SQL database in your browser.

The menu to the right displays the database, and will reflect any changes.

Feel free to experiment with any SQL statement.

You can restore the database at any time.

Это выборка всех клиентов. идите? вам не нужно знать программирование, чтобы в этом разобраться. Язык SQL считается языком бизнес-аналитиков, обычно с помощью него строят всякие отчеты, работают с данными и т.д. Если вам интересно его поизучать, то вот вам классный ресурс, где можно это сделать совершенно бесплатно и там же попрограммировать. Кстати картинка взята оттуда:).



https://www.w3schools.com/sql/

Другим популярным языком, на котором пишется 99 % всех интерфейсов программ и приложений является Java Script, скриптовый язык, который ничего не имеет общего с языком программирования Java, кроме названия. С чем он работает? если SQL работает с данными и таблицами, и всяким объектами данных, то JavaScript работает с очень интересными объектами, которые мы видим почти каждый день – это объекты нашего браузера. Той самой программы, через которую вы ходите в Интернет. Сейчас это все очень логично и интуитивно, но поверьте, лет 20 назад, зайти в интернет и что-то там найти было довольно сложно, поэтому Google и появился. Нужно было всегда знать точное название сайта, когда вбивал его в строку поиска. Это сейчас быстро и понятно, а раньше это была какая-то магия. Ты открывал браузер и не понимал, чего делать дальше:). Сидел и тупил, если не знал, ни одного сайта. Представьте. Поэтому тогда популярные имена сайтов стоили очень дорого, потому что их любой

мог запомнить. Так вот вернемся к браузеру (да именно так и называется Chrome, Safari и другие программы, без которых вы не можете жить). Когда вы нажимаете кнопку “Открыть страницу”, то вся страница запускается частями:

1. Сначала ваш браузер запрашивает код страницы (HTML код) у сервера (это тот компьютер, на котором лежит страница)

2. Сервер отправляет этот код вашему браузеру

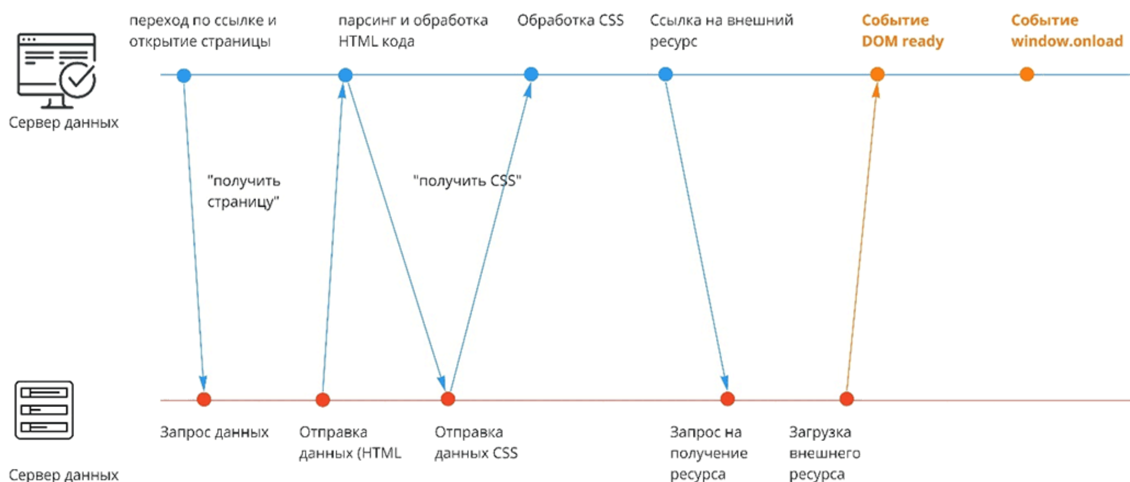
3. Браузер его запускает (фактически там указаны инструкции, как нарисовать сайт)

4. И браузер начинает рисовать сайт (процесс рисовки называется ***рендеринг (render)***, сначала он:

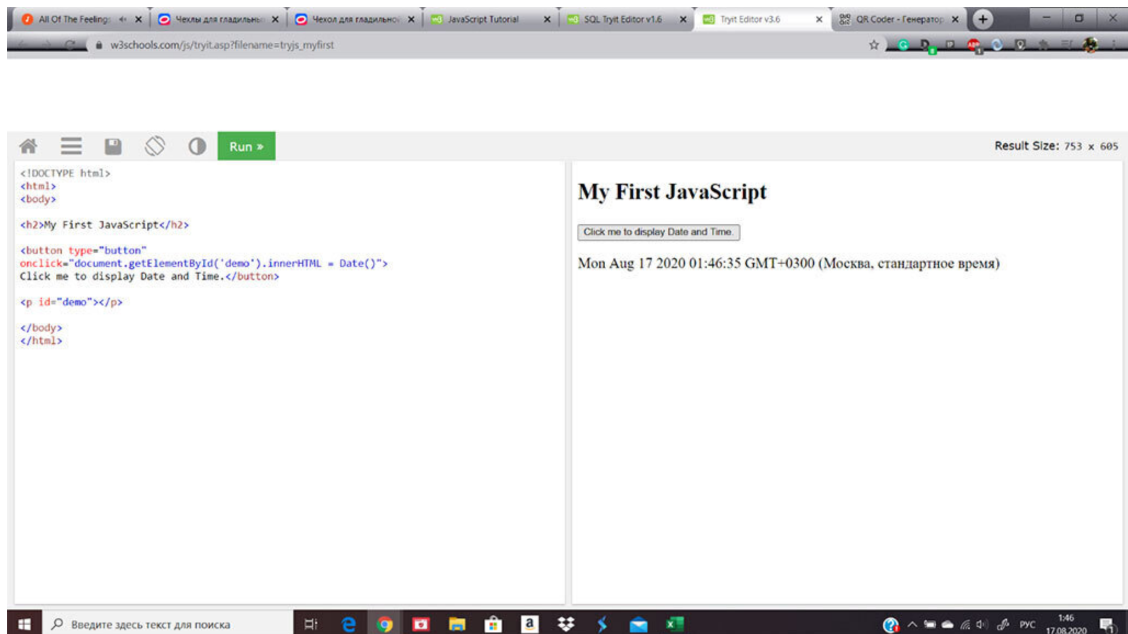
1. Строит все объекты, которые есть на страницы (называются DOM объекты, от слова Document Object Model). Любая кнопка, баннер – это объект и у него есть свойства и с ним можно чего-нибудь делать. Прикольно да?

2. Потом загружает стили для каждого объекта. Стиль, это набор правил, как правильно нарисовать объект – какого цвета, шрифта и т.д. Все стили находятся в таком документе, который называется CSS (Cascading Style Sheet – каскадная таблица стилей, ей Богу не понимаю, зачем так сложно назвали, могли бы проще, например таблица стилей Татьяна)

3. После этого, для отрисовки страницы начинают загружаться недостающие картинки, видео, файлы, все это называется **ресурсы.**



Соответственно с каждым этим пунктом работает Java Script. Например можно создать кнопку, которая будет что-нибудь делать, или сделать несколько полей для ввода информации и т.д. На примере ниже, простейший пример на javascript, который вы можете немного понять, даже не зная программирование.



Слева написано, что в теле (body) сайта, есть кнопка (button), которая при событии кликнуть по ней (onClick) получает из текущего документа страница, дату и время. Конечно, это немного сложнее чем SQL, но это точно легче чем Си или Java или ассемблер. Программированием на таком языке занимаются front-end программисты (если переводить, то разработчики интерфейсов). Если вы попросите их разработать всю программу целиком, то они не смогут, потому что работа с базой данных это другой диалект, это как испанца попросить поговорить на китайском, поэтому вам нужен будет еще человек, кто знает SQL, знает Java, чтобы все вместе написать. Итого минимум 2 человека, на то чтобы написать какую-нибудь простую веб-программу, то есть ту программу, которую не нужно будет устанавливать и в которую можно будет заходить откуда угодно.

ПРИНЦИП 1: “СОБСТВЕННАЯ РАЗРАБОТКА, НУЖНО МНОГО РАЗНЫХ СПЕЦИАЛИСТОВ

В одном случае, вы просто покупаете готовый софт, а в другом вы начинаете его создавать и чем больше всяких разных компонентов в софте, тем больше вам нужно будет людей. Но для полноценной жизни одного разработчика нужны и другие компетенции.

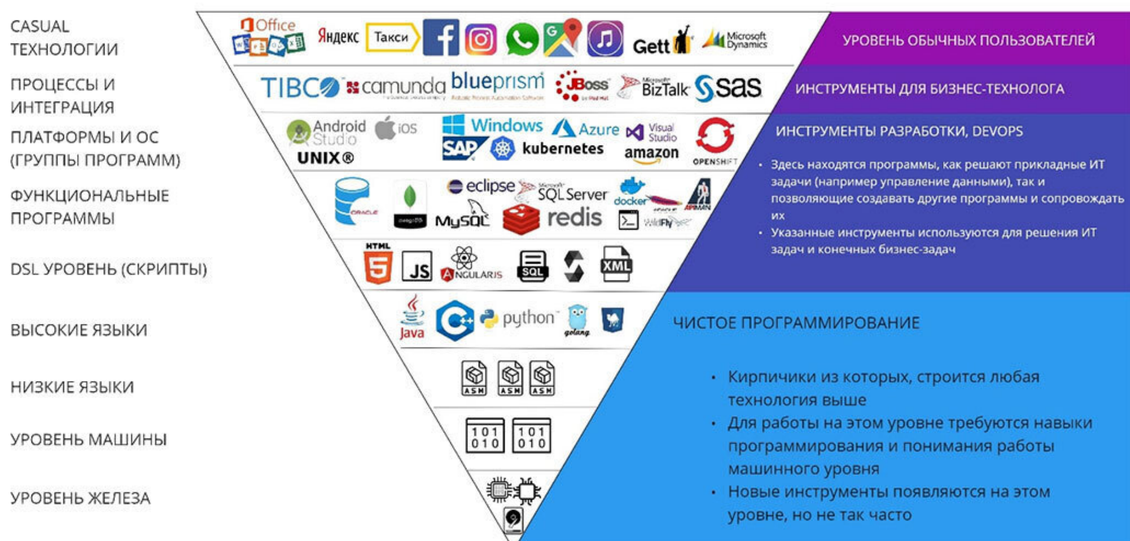
ПРИНЦИП 2: “ЗА 1 РАЗРАБОТЧИКА, 2-Х ТЕСТИРОВЩИКОВ И 1ГО АНАЛИТИКА ДАЮТ”

Чтобы ваш разработчик мог что-то писать, надо ему поставить задачу. И обычно это делают аналитики. В части нашего примера с Java Script они разбирают, какие объекты на сайте есть, какие у них свойства, какие они принимают состояния, какие должны быть пользовательские сценарии и т.д. Но и это еще не все, чтобы разработчик работал, его результат кто-то должен протестировать. Есть такой принцип Maker-Checker – Один делает – другой проверяет, чтобы исключить ошибки. Так и тут, конечно, это все выливается в количество людей. Т. е. если вы решили нанять 2 разработчиков, то вам придется еще и 4х тестировщиков взять и 2х аналитиков взять.

Но сперва надо всегда понимать, чем они будут заниматься, ведь купить готовую программу проще, чем содержать штат программистов. Иногда просто хочется заниматься бизнесом и не думать, как же правильно разместить кнопку. Фактически все готовые программы, которыми мы пользуемся когда – то были написаны такими же командами разработки, как продукт на продажу. Только их было гораздо больше, чем 8. Например на Microsoft Office было потрачено миллионы часов разработки. Таких программ, много и все их можно найти на самом верхнем уровне пирамиды, где будет уровень обычных пользователей. Т. е. точно не требующих никаких навыков программирования. Например, Яндекс такси, facebook, whatsapp, photoshop, мы можем ими пользоваться и нам для этого не нужны навыки программирования. Хотя тот же фотошоп требует уже навыков дизайнера.

ПРИНЦИП 3: "ЕСЛИ ПОКУПАЕТЕ ТЕХНОЛОГИЮ, ТО ВАМ ДЛЯ НЕЕ НУЖЕН СВОЙ ПОЛЬЗОВАТЕЛЬ"

У каждой технологии свой пользователь. Например, у платформы роботизации blueprism (это один из лидеров в части роботизации процессов) это будет бизнес аналитик с навыками программирования, т. н. бизнес-технолог. А чтобы работать в операционной системе UNIX, вам нужен Devops инженер или системный администратор (по старинке), кто в ней разбирается.



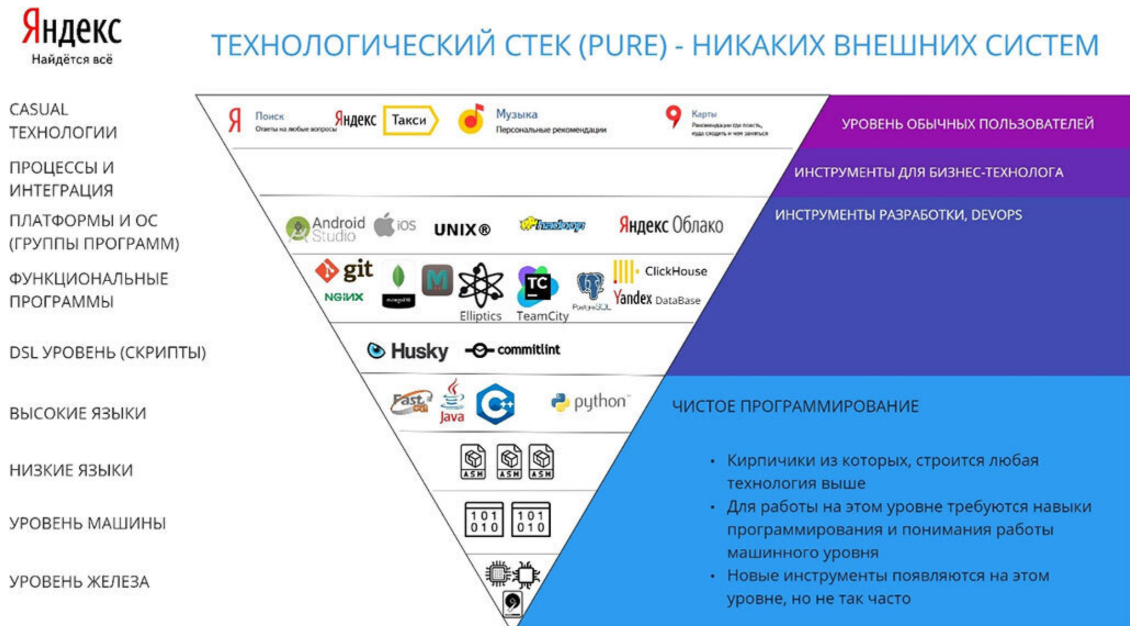
Каждый слой такой пирамиды основывается на предыдущих слоях, и представляет собой уровень абстракции. Как видите, на определенном уровне появляются базы данных, на каком-то уровне появляются операционные системы и т.д. Чистый программист работает на уровнях до высоких языков программирования. Дальше уже появляются различные фреймворки разработки, появляются такие люди, как DevOps инженеры, которые управляют ИТ системами с помощью других ИТ систем, таких как kubernetes, Amazon, Openshift и т.д. Обычный пользователь живет на самом верху, пользуясь обычными приложениями, например Яндекс такси или оффис. Надо понимать, что каждый такой уровень съедает часть ресурсов машины / времени на перевод команды в формат, команды, который работает на нижнем уровне. Например, Яндекс такси, это работа с базой данных и работа программы на языках Си и Пайтон, которые в свою очередь конвертируются в машинный код. Чем больше уровней абстракции, тем более

ресурсоемкая пирамида получится, ведь каждый уровень будет потреблять какую – то часть ресурсов, с другой стороны, вы получите независимость одного уровня от другого. Есть даже отдельный уровень интеграции и управления процессами, где живут такие инструменты, как BPM, шина данных, роботизация процессов и т.д. Все эти инструменты дают готовую среду, которая ускоряет процесс разработки и цифровизации процессов. Любая технология требует ресурс, как цветок своего горшка. Если вы решили посадить у себя оранжерею, то придется разориться на кашпо и землю. Конечно, есть компании, кто все старается отдать на аутсорс, вообще всю ИТ функцию, но обычно они плохо заканчивают.

ПРИНЦИП 4: "ВСЕ КОМПАНИИ, КТО ОТДАЛ НА АУТСОРС ИТ ФУНКЦИЮ ПЛОХО КОНЧИЛИ"

Кароче, вот это догма). Аксиома. Если у вас нет амбиций или вы не хотите расширяться, а хотите сохранить свой свечной заводик то отдавайте все на аутсорс, но помните, что со временем это все станет большой проблемой, в том числе и потому, кому будет принадлежать IP (Интеллектуальная собственность).

А можно все самостоятельно сделать? Чего нас пугать? Конечно все это можно сделать напрямую через те же языки высокого уровня, вопрос только зачем. некоторые компании, специально исключат у себя такие инструменты, чтобы повысить скорость обработки, ведь каждый уровень, как мы поняли съедает часть времени обработки операции, как своего рода плату за проезд. Так если посмотреть стек Яндекса (я его собирал вручную, изучая всякие материалы в Интернете), то получим следующую картину:



Как видите, здесь очень мало всяких BPM систем нет. Даже скриптовые языки практически не используются, только голое программирование. Поэтому в Яндексе нужны программисты C и Python, а обычным бизнес-аналитикам и программистам на BPM там практически делать нечего. Такая суровая реальность, если вы посмотрите фильмы про Google и другие компании, то увидите, что в основном там работают программисты, и программисты и. эээ еще раз программисты. Нет конечно там есть всякие инженеры, аналитики, продуктологи, но их меньшинство. Никаких тебе бизнес технологов и т.д. Понятно почему, потому что активом

такой цифровой компании являются ее инструменты и системы, и именно в них вкладываются ресурсы и капитализируют их. Такая вот утопия программистов:). Конечно, Яндекс тут съэкономил на других компетенциях, но подойдет ли такая модель другим компаниям?.. все очень индивидуально. При этом что делать всем остальным? ну либо учить программирование, либо идти в обычные компании, в которых есть большой legacy бизнес, который никуда не денется. Почему не денется? ***ну, потому что "обычные компании никогда не смогут уйти в цифру на 100 %"***.

ПРИНЦИП 5: "ДАЖЕ СОБСТВЕННАЯ РАЗРАБОТКА НЕ ПОМОЖЕТ ВАМ НА 100 % ЦИФРОВИЗИРОВАТЬ КОМПАНИЮ. У КАЖДОЙ КОМПАНИИ ЕСТЬ СВОЙ ТЕХНОЛОГИЧЕСКИЙ ПРЕДЕЛ"

Считайте это законом Благирева. Как бы вы не старались, на 100 % вам не удастся оцифровать компанию, т. е. заменить всех людей, или отказаться от бумаги, перестроить все процессы (например бухгалтерию и тд). У каждой компании, есть свой технологический предел, то есть те технологии, которые она сможет использовать, потянуть и не забросить.

ПРИНЦИП 6: "ВАМ ВСЕ РАВНО ПРИДЕТСЯ ЧТО-ТО КУПИТЬ"

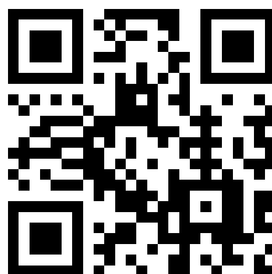
Вот если вы не Яндекс, то вы все равно что-то купите, это нормально, например у вас будут использоваться всякие инструменты типа BPM, роботизации и т.д. Есть одна очень интересная компания, называется она South Korea Telecom или просто SK Telecom. Ее очень любили приводить в пример в одной замечательной большой российской компании. Так в 2015 году они выделили отдельное подразделение, которое должно было заниматься только инновациями – SK Planet. За 2 года оно показало выручку в 2 млрд долларов, но со старым SK Telecom ничего не произошло. Таких примеров много, поэтому если вы изначально не строите цифровую компанию, то ваш технологический стек будет разным. Причем у каждой компании есть ***свой технологический предел***, т. е. дальше которого она никак не может прыгнуть, если изначально не была заложено ***цифровое ядро***. Так, например фактически технологический предел, означает:

1. Способность внедрить все технологии, которые свойственны этой категории компаний
2. Наличие денег и ресурсов, чтобы это сделать
3. Наличие компетенций для управления этими технологиями.
4. Время в течении, которого вы сможете управлять технологией.

Действительно большинство компаний, если посмотреть с ИТ точки зрения напоминают кладбища технологий, которые кто-то пытался внедрить, у него не получалось и он бросал. Как – то раз я спросил одного своего знакомого, почему он меняет каждые 3 года работу в сфере BI (Business Intelligence) и работы с данными. Его ответ оказался интересным, "Слава, потому что через 3 года, управлять данными становится для меня очень сложным, что я не знаю, что с этим дальше делать". Вот он технологический предел, его предел это 3 года. В итоге он запускал технологию и переходил в другую среду. Хотя так еще бывает часто из-за политики. Сама технология она аполитичная, это скорее механика. Поэтому программисты и инженеры не очень любят играть в политику и очень любят математику, потому что она точная и дает двусмысленного трактования. Но если в организации правит политика, то любая технология станет жертвой и пополнит количество трупов на корпоративном кладбище. Как-то лет 10 назад, работая в большом красном банке у меня стояла задача сделать сделать автоматический расчет CIR (Cost Income – отношение прибыли к расходам) по клиенту. Тот самый,

о котором я писал раньше. Так вот когда я его проектировал, то изучал хранилище данных и случайно наткнулся на таблицы и процедуры, которые считали похожие показатели. Быстро изучив, я понял, что они вообще никем не используются, более того их расчет выключен и не работает. Открыв программный код расчета витрины показателей, я удивился, потому что он был правильным. Все операции по клиенту загружались в определенные таблицы, дальше на основании комбинации номеров бухгалтерских счетов и регулярных выражений (это формальный язык для работы со строками текста) и функции RegexpLike в informatica, я быстро воссоздал работающую схему. Да она считала только 30 клиентских операций (из 124 которые мне нужны были), но я наше основу. Дальше я попросил ребят из своего отдела, дополнить этот описанный механизм, ребята дополнили, потом оптимизировали и он заработал. Очень прикольно заработал. Жалко только заказчик этого функционала после его запуска, сказал, что он ему не нужен. Так после воскрешения он снова вернулся на кладбище:). Даже было немного грустно, и тупо. А самое смешное, что руководитель организации даже не подозревал в то время, что у него есть такой крутой функционал. Поэтому прежде, чем внедрять новую технологию сходите на свое кладбище, побудьте Индиана Джонсом или Ларой Крофт. Откройте могилы похороненных проектов, там могут оказаться очень крутые артефакты. Я понимаю, что звучит это странно, но в кино обычно всегда самые крутые штуки, волшебные посохи, мечи кладенцы находили в каких-нибудь могилах. Тут такой же принцип. Я помню, как нашел в одной большой телеком компании, в DPI платформу в гараже, CMS платформу, которая оказалась в лидерах в квадрате Garnter'a.

С другой стороны, большинство Enterprise архитекторов (это те люди, которые определяют какой набор технологий должен присутствовать в компании), просто под копирку пытаются копировать различные рекомендации. Вы, наверное, удивитесь, но фактически каждый тип компании в мире, уже был проанализирован, расписан, и создан список технологий, которые нужно ставить для каждого такого типа. Так появились модели BIAN (Banking Industry Architecture Network – ODA (Open Digital Architecture, Открытая Цифровая Архитектура от Telecom Forum,), или ODF (OPen Digital Framework для раскрытия потенциала 5G, от того же Telecom Forum'a)



https://www.bian.org

https://www.tmforum.org/oda/

Их очень много. Для каждой индустрии, есть свой blueprint (чертеж в переводе означает), который поможет вам понять, что в рамках вашей индустрии в мире принято. Конечно все эти чертежи, нужны для того, чтобы разобраться в том, какие технологии придумали умные люди для вашей предметной области и начать с ними работать. Главная цель их, это сократить время на внедрение ваших идей и получение результата. Будь то, производство товара или услуги, до тестирования идей. Поэтому не нужно ничего придумывать, просто берите, что то что придумали до вас и адаптируйте под себя. Вот, например карта технологий, которую я когда-то сделал для своих студентов, здесь я попытался, расписать все виды технологий, которые могут быть в компании. Назову ее B-Map, от Blagirev карта:). Если что это шутка:).

Итак, давайте подведем итог:

1. Каждая технология состоит из технологий попроще, как пасхальное яйцо.
2. Все технологии в общем сводятся к тому, чтобы давать команды машине
3. Команда – это набор сигналов 0 и 1, которые в физике достигаются через управление напряжением
4. Машина это набор транзисторов, которые есть в процессоре, для выполнения команд, это память для хранения данных и интерфейсы для ввода и вывода информации (например экран)
5. Каждая технология рассчитана на определенную аудиторию пользователей
6. Обычный программист, может разработать любую технологию самостоятельно, вопрос только в количестве часов и качестве результата.
7. Большие цифровые гиганты автоматизируют ИТ процессы, и на этом же уровне автоматизируют бизнес-процессы. У них нет никаких BPM систем и прочего.
8. Есть компании, которые никогда не смогут стать технологичными, у них свой путь и нужно это понимать, поэтому у них будет свой ассортимент технологий. Это нормально. Нет никаких Enterprise паттернов и шаблонов, как должна выглядеть архитектура предприятия.

Правила разработки (Code Convention)

Я решил оставить на десерт очень интересную тему, как “Правила разработки” или его еще называют “Соглашением разработки”. Смотрите, если вы хотите создавать очень большие программы, например написать какую-нибудь платформу, то вам потребуется обеспечить, чтобы ваши команды разработки (внутренние и внешние) понимали результат действий друг друга и то, что они будут писать. Чужой код напоминает книгу, написанную на другом языке или диалекте. Даже если вы будете знать язык программирования, то для того, чтобы осознать смысл всех алгоритмов и конструкций, которые описаны в коде, вам потребуется их расшифровать и изучить. Так обычно происходит, если исследователь пытается прочесть книгу на древнем мертвом языке. Он, например знает, что означают символы, но не знает фонетически, как правильно произнести и начинает гадать методом тыка. Любой программный код, это набор команд, которыми вы описываете сценарий задания машине, что ей сделать, посчитать или сложить, что вывести на экран, куда сохранить результат вычисления и т.д. Для каждого вычисления вам нужна переменная, которая будет связана с этим результатом или параметром, который будет использоваться для вычисления. Например, у вас считается кредитный рейтинг платежеспособности по разным возрастным группам и одним из параметров вы возьмете параметр “возраст”. Допустим мы назовем его “а”. Но не в каждом языке программирования машина поймет, что такое а и какие значение оно сможет принимать, поэтому нужно задать определение более четкое. Например,

```
begin
a: integer;
end
```

Часто многие языки начинаются вот с похожих конструкций begin end, которые означают начало и конец выполнения программы. a: integer означает, что параметр a будет иметь тип integer, этот тип во многих языках означает “целое положительное число”. Вообще задумайтесь, как много общего во многих языках программирования, как и в обычных языках. Так, например тип integer или int практически во всех есть. Он обусловлен не тем, что одни и те же люди создавали языки, а тем что такой тип “органически необходим всем”. То есть он как золотое сечение, который всегда будет, потому что алгоритмы на этой планете требуют этот тип для работы. Есть, например другой тип float. Это тоже числовой тип, но с плавающей точкой, т. е. это дробное число. Ну так вот вернемся к “а”. Вам понятно, что **“a”**, это age или возраст? Конечно нет. А если вы например передадите вашу программу другому программисту исследователю, то поймет ли он что такое а? нуу… если вы ему не скажете, или не дадите инструкцию или не укажите комментарий прямо в коде, то точно нет. Получается сторонам нужен некий свод правил, чтобы можно было передавать друг другу информацию описанную в виде программного кода. Ну и обеспечить преемственность для будущих поколений. Я немного пишу тут как археолог и историк, но так оно и есть. Чтобы ваш код прошел через поколения, должны быть правила его прочтения. Наша современная письменность прошла через поколения, потому что был алфавит, орфография, грамматика и фонетика, которые кто-то придумал. Это все некие правила. Они и называются Code Conventions (Код конвешионс). Если провести параллель, между алфавитом и обычными языками, то фактически это алфавит и правила его использования – орфография и грамматика. Вы тут сейчас такие, наверное, скажете, “воу воу полегче парень! Ведь в языке программирования, это вроде все есть”. Не совсем так. Этого там нет, машине можно скормить все что угодно, она понимает только 0 и 1 и поэтому живет в бинарном измерении, а мы другом измерении. Даже если в языке есть минимальный набор правил написания кода, который называется синтаксис, то нужно помнить. что задача синтак-

сиса совершенно другая чем обеспечить преемственность программного кода. Задача синтаксиса всего ли обеспечить конвертацию вашего программного кода в машинный язык.

За свою практику я столкнулся с интересным фактом в отношении code conventions (для простоты я буду называть их Правилами). Так вот не в каждой развитой с точки зрения ИТ организации были Правила. Если эти наблюдения, как – то обобщить, то можно сказать, что Правила мне встречались где-то в среднем в 20–30 % внутри организации и в 1 из 5 организаций в целом. При этом среди эти организаций были большие ИТ бюджеты, солидные ИТ руководители, но не было сформировано ИТ культуры и никакого намека на Правила. Это странно. Иногда мне приходилось лоббировать или требовать создания Правил в том числе и со стороны Заказчика. Обычно слышишь много клевых аргументов, почему разработчики не хотят делать программный код по Правилам, они ведь художники:). Самые классные это – "Это первый и последний раз!", "Мне срочно нужно вывести доработку на бой иначе все сломается", "я один кто поддерживает эту систему и мне все понятно. не вижу смысла", ну или "это все для хипстеров. Бессмысленная трата времени". Но Правила, это основа ИТ культуры, иначе вы будете в заложниках у вашего разработчика, а еще его знание просто потеряются с годами, и никто не сможет понять, а как же проходит те или иные вычисления. Есть программы, в которых тысячи строчек года, если все строки кода будут одинаковыми, все переменные будут просто называться – a, b, c, d и тд, то такой код невозможно будет расшифровать или потребуется очень много. Например, взять шумерские таблички им почти 5000 лет. Вот табличка с письмом царю Лагашу:

Вы что-нибудь понимаете?:). Ну вряд ли. Дешифровка этого языка идет уже почти 200 лет с переменным успехом. В ИТ такая организация программного кода называется "спагетти-код". Его так называют, потому что он весь извилистый и непонятный.

Вот другой пример, фестский диск, который был создан минойской цивилизацией. Минойская цивилизация главенствовала в Средиземноморье много тысяч лет назад. Ему тоже несколько тысяч лет. Его нашли на Крите (это там где минотавр жил, для тех кто не знает). Когда я там был, то местный год мне рассказал теорию, что минойская цивилизация – это цивилизация атлантов или другой высокоразвитой цивилизации, которая погибла в результате взрыва вулкана на острове Санторини. Посмотрите на этот диск, вроде все символы хорошо пропечатаны и изображают разные рисунки, но, к сожалению, никто еще так и не смог прочитать этот диск. Например, возникает вопрос, откуда его читать, с краю к середине или от середины к краю. Что означают деления на диске? Где заканчивается одна часть этого кода и начинается другая. Чтение чужого программного кода выглядит точно также. Вот вы его открываете, и вроде видите знакомые очертания, но не можете понять, а что же за смысл там скрыт. Вы будете, наверное, удивлены, но очень много я встречал вот таких вот артефактов в ИТ ландшафте, когда разбирал завалы и полки со старыми системами. В каждой большой компании есть свой гараж бэтмена, где лежат забытые технологии, которые никто не помнит, как использовать и для чего они создавались. В лингвистике, если ты программируешь, то тебе легче изучать древние языки. Инженерия очень тесно связана с окружающим миром и является его проекцией, но в ноутбуках и компьютерах. Если у вас не будет ИТ культуры, то со временем все ваши системы и технологии придут в упадок, при этом ваш авторитет или стиль управления не будет иметь никакого значения, ведь вы временное звено в цепочки управления ИТ и технологией. Технология должна сама жить без вас и передаваться из поколения в поколение, но не через из уст в уста, а через правила. Если вы посмотрите историю человечества,

то технологии, которые дожили до нашего века и улучшились, они все были описаны и записаны. Их раскладывали по полочкам и учили принципы со школы, так чтобы следующее поколение могло взять и начать использовать интеллектуальные труды предыдущего. К сожалению, у человека, как у пчел не заложено принципы использования технологии в ДНК. Может быть, когда-нибудь, программирование, так и стили программирования и правила, будут заноситься в ДНК или зашиваться куда-то на подкорку, а пока этого не произошло, попробуйте в своей компании выстроить преемственность программного кода и вы увидите, как люди перестанут изобретать велосипед, писать заново одни и те же библиотеки и методы.

Сами Правила должны быть для каждого языка и для каждой системы свои. Вам не удастся создать общие для всех Правила, ну и это бессмысленно. Сами Правила можно разделить на 2 типа:

1. Code Standards – правила построения алгоритмов и использования переменных и тд):

○ длина идентификаторов и переменных

○ какие имена можно назначать переменным – используют только смыслово значимые переменные, например Age вместо a73fsfBd для хранения возраста

○ регистра букв и цифры – например некоторые языки чувствительные к регистрам, т. е. переменные Age и AGE будут восприняты, как разные.

○ правила использования спец. символом – тут они тоже есть, это всякие /? и другие

○ слова, разделенные буквами, – AgeOfEmpires итд

○ правила формирования типов переменных – каким значениям можно присвоить логические типы, а каким числовые. Например, когда присваивать логический тип boolean, а когда числовой

○ когда переменная должна быть глобальной и доступа другим модулям и библиотекам

○ какие проверки нужно делать в коде, чтобы избежать уязвимостей в кодей, таких как XSS, SQL инъекции и т.д. Например, брать любой код в кавычки или круглые скобки, если он не был сгенерен вашей программой

○ наличие комментариев и аннотаций

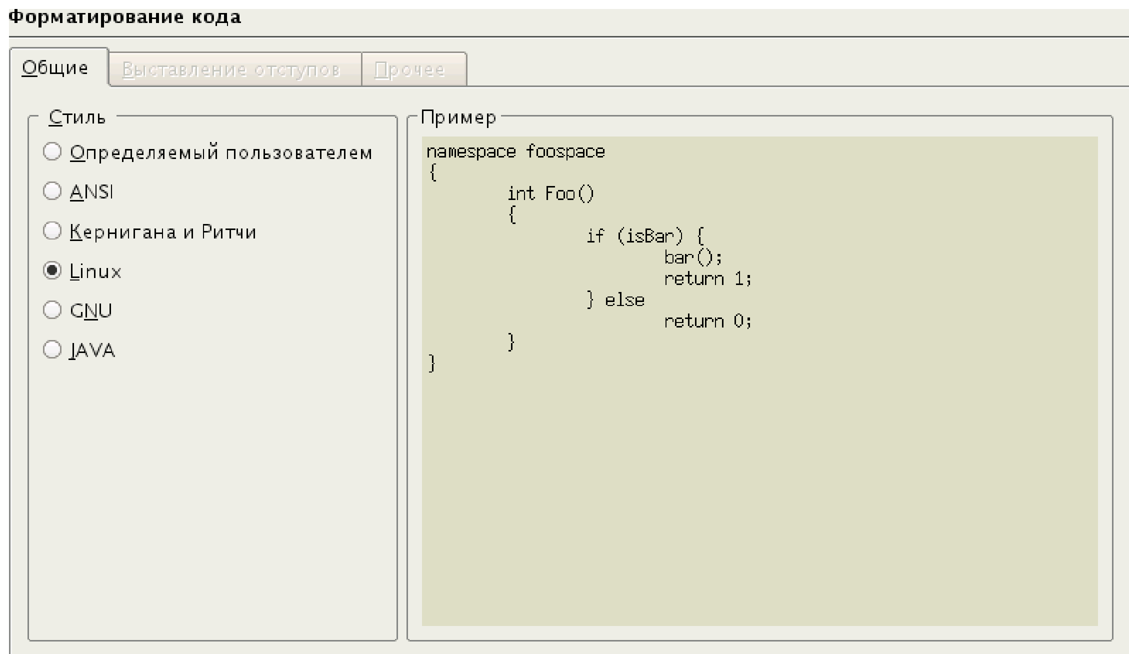
○ какие внешние библиотеки являются общими и обязательными

○ согласованность в коде и приложениях

○ обработка исключений и ошибок

○ принципы объявления переменных (не объявляйте переменные, если не используете, не забывайте закрывать переменные)

2. Code Style – это непосредственно правила написания кода, т. е. стиль. Если проще сказать, то это правило, когда вы ставите в коде кавычки и когда их закрываете. Это называется конструкция. Например, размер отступа или структура кода. Еще в школе и университете меня учили, что у кода должна быть структура. Самой машине все равно, как его читать, а вот человеку нет. Различают основные стили программирования, еще их называют стили “отступов”, потому что для того, чтобы их соблюдать надо делать отступ в коде слева:



• Стиль Керниган и Ричи

Назван в честь Брайна Кернигана и Дениса Ричи, которые написали 100500 книг по программированию СИ. А Денис Ричи был его создателем. Фишка стиля в том, что практически все примеры в книжках были оформлены соответствующим образом, так что это стало мейнстримом

```
if (<cond>) {
       <body>
}
```

• Стиль Олмана

В честь Эрика Олмана. очень крутой дядька из университета Беркли, участвовал в развитии и становлении систем Беркли (BSD систем). Это unix системы. Сейчас 90 % серверов работают на таких системах. Сейчас существует очень много разных UNIX систем.

```
if (<cond>)
{
       <body>
}
```

• Стиль Уайтсмитс

Whitesmiths Limited была такая компания в 1980х годах, которая занималась разработкой. С языка

```
if (<cond>)
       {
       <body>
       }
```

• Стиль GNU

GNU это одна из UNIX операционных систем, которая стала дико популярной, особенно из-за того, что придумала способ быстрого онлайн обновления через дистрибутив debian. Вообще про UNIX можно будет долго говорить.

```
if (<cond>)
   {
      <body>
   }
```

- стиль TODO

Этот стиль появился в Java. Он связан с тем, что в коде вы пишите себе заметки, что нужно сделать с кодом в будущем. например:

```
// TODO: Remove this code after the UrlTable2 has been checked in.

// TODO: Change this to use a flag instead of a constant.
```

Один из самых ранних стандартов описания кода является “венгерская нотация”. Придумал ее Чарьз Симони аж в 1999, когда работал в компании Microsoft. Одним из его проектов был проект Word, да тот самый:). Он создал один из первых в мире WYSIWYG процессоров. У Симони каждая переменная кодировалась особым образом, если быть точке префикс переменной создавался по особым правилам.

WYSIWYG процессор – это инструменты для редактирования текста, которые широко используются в редактировании и веб приложениях. Например, эта книга написана в google docs, который является WYSIWYG редактором. Само слово появилось от аббревиатуры What You See Is What You Get, «что видишь, то и получишь. Вообще это сейчас любой онлайн редактор. Поэтому, если вам нужно на сайт прикрутить такой редактор или в приложение, вы тут же понимаете, что вам нужен просто визивиг редактор

https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/

Самое интересное, что нотация в конце концов была интерпретирована большинством программистов неправильно и когда появилась новая платформа для программирования у Microsoft,NET, то Microsoft начали говорить, что лучше не использовать венгерскую нотацию. В целом, если вам интересно почитать что такое хорошая и плохая нотация, то есть достаточно клевая статья Making Wrong Code Look Wrong (как сделать неправильный код выглядеть неправильным)

Ключевые выгоды

1. БЫСТРОЕ ОСВОЕНИЕ КОДА

Любой новый человек потратит на 20–30 % (а может и больше) времени меньше на погружение в код чем в код, который написан без конвенций.

2. ПОВЫШЕНИЕ КАЧЕСТВА И СНИЖЕНИЯ КОЛ-ВА ОШИБОК

Действительно при качественных правилах у вас резко снижается количество ошибок, конфликтов имен, уязвимостей и тд

3. ПОВЫШЕНИЕ СКОРОСТИ РАЗРАБОТКИ

Если вы уберете необходимость у разработчика придумывать, как же оформить и структурировать программный код, то он сфокусируется уже на алгоритме, а не на оформлении и скорость самой разработки у вас вырастет. По моим оценкам где – то на 15–20%

Поэтому, если у вас нет Правил, то лучше скорее их сделать. Кто бы что вам не говорил в ИТ. У каждой системы должны быть свои правила разработки, например у BPM свои, для написания web приложений свои, для мобильных свои и т. д.

Встроенное программное обеспечение (Embedded Software)

Не все программы стоят отдельно от датчиков и устройств. Многие программы устанавливаются сразу в микросхему устройства и становятся неотделимым целым от девайса, т. е. встроенным (embedded). Основное отличие от обычных программы, к которым мы привыкли их называют application software заключается в том, что:

1. ES работает только под конкретное устройство (например чайник, который измеряет температуру воды и сканирует, распознает, а что же за чай вы засыпали в него)
2. ES созданы, чтобы выполнять конкретную задачу.
3. ES обычно не обладают никаким пользовательским интерфейсом. Например, с прошивкой того же чайника пообщаться через пользовательский интерфейс не получится
4. ES часто работает без операционной системы и других приложений
5. ES имеет ограниченные возможности и размер памяти, поскольку само устройство ограничено физическими размерами. Так например вы не сможете засунуть редактор ворд в чип, который находится внутри вашей банковской карты, а вот засунуть туда приложение, которое хранит ваши данные программы лояльности, вполне.

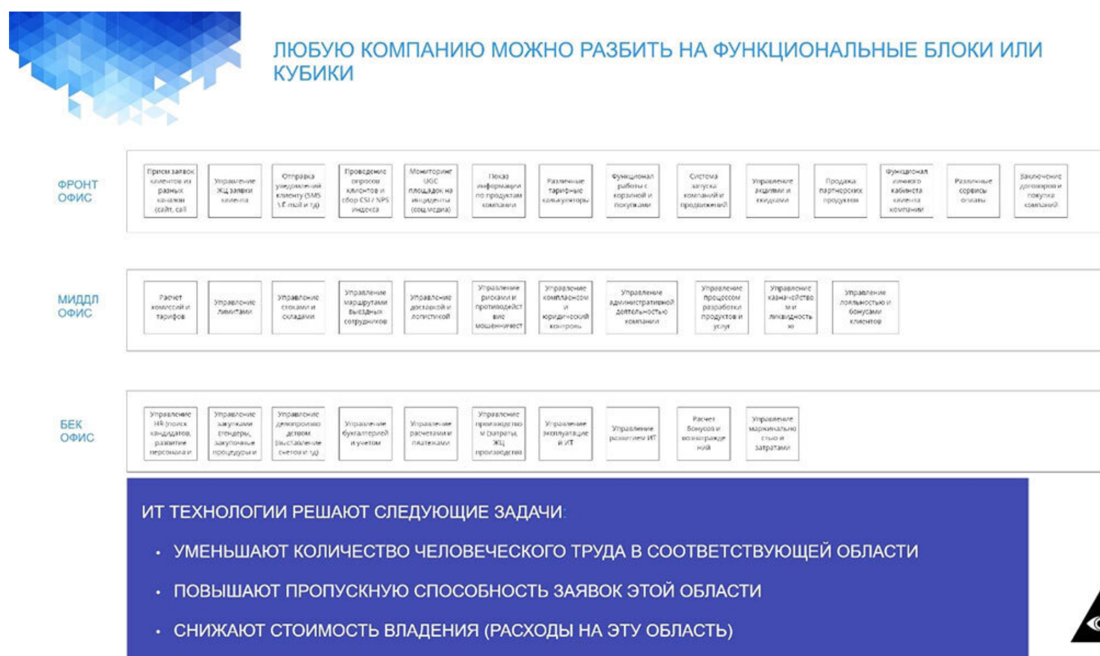
ES приложение часто представляет собой Embedded computer

Часть 6. Карта технологий

Любую компанию можно представить в виде операционной модели, в которой есть набор функций, которые выполняются внутри этой компании. Например “Работа с клиентами”, “Бухгалтерия” и т.д. В этой операционной модели есть 3 уровня или слоя, как в пироге, это:

1. Фронт офис (от англ. “front”, т. е. спереди), там, где вы работаете с клиентами. Тут будут все функции и процедуры, которые касаются клиентов.
2. Миддл офис (от англ. “middle” – по середине), здесь находятся все процессы, функции, которые связаны с сопровождением клиентов, расчетом комиссий, оценки рисков.
3. Бек офис (от англ. “back” – сзади), здесь уже находятся все процессы, которые чаще не связаны с клиентами напрямую, например бухгалтерия, расчеты, переводы, открытие счетов, ведение складской деятельности, производство и т. д.

В каждой такой функции, есть тот кто ее выполняет, есть всегда результат этой функции. Для простоты давайте эти функции назовем “кубиками”. Фактически любую компанию можно разбить вот на такие кубики.



В проектировании, эти кубики называются Use Case (или сценарии использования технологии), в Agile, они носят название User Story (пользовательские истории) и т.д. Сам термин Use Case возник в языке UML (Unified Modeling Language – унифицированный язык проектирования). Этот открытый язык появился в период с 1984 по 1995 год, и стал активно использоваться в качестве, лучших практик моделирования и создания технологии. Любая технология решает определенные Use Case. Не больше, ни меньше. Я буду часто к этой аксиоме возвращаться, чтобы вы это запомнили. Нет универсальной технологии, каждая решает конкретные задачи. Если в вашей компании, технология не работает, а в другой компании работает, а это точно так, иначе бы технология не существовала, то видимо вы ее неправильно используете. В части кейсов, в мире появилось большое количество функциональных систем, которые заточены делать конкретную функцию, например управлять процессами (системы типа BPM) или управлять рисками внутри организации, например кредитными или операционными (такое

семейство технологии можно назвать Risktech). Как вы заметили, название того или иного семейства технологий складывается из 2х слов, это “названия предметной области” + “tech”, краткого от англ. technology (технология). Часть таких акронимов вы, наверное, не слышали, например API Tech (технологии для управления интеграцией), а другие возможно слышали HR Tech (технологии в управлении персоналом – HR). Я специально все типы технологий привожу к единому знаменателю, чтобы вам было проще ориентироваться в них. Задача этой книги, не помочь вам выучить, все эти технологии, а помочь вам ориентироваться во всем этом множестве. Все эти названия, они не конечные, их можно дробить на подклассы, придумывать новые названия. Но в целом, это некий такой нулевой уровень, на мой взгляд с которого все начинается, и который является общим для 99 % компаний. Если не верите, можете взять свою компанию и попробовать составить для нее b-map.

Давайте попробуем разобрать, этот разноцветный ковер.

Process Tech

Processes Technologies, это все то что связано с автоматизацией процессов. Начнем с самых популярных технологий, связанных с автоматизацией процессов. В целом тут можно выделить 2 больших класса технологий:

1. Технологии, которые позволяют управлять процессом (BPM)
2. Технологии, которые позволяют автоматизировать простое действие в процессе (роботизация)
3. Технологии, которые изучают эффективность процесса (Processes Mining)

BPM

BPM (бипиэм читается:), появились на пороге прошлого столетия. Не система, конечно, а предпосылки. к тому, что любая организация – это прежде всего набор процессов. Хотя вы не поверите, но до сих пор есть люди, которые в это не верят, ну не будем их расстраивать:). 100 лет назад, компании были очень неповоротливыми, никто толком не изучал и не пытался их систематизировать. Первым человеком, кто это попробовал сделать, был Фредрик Тейлор, не путать с Бруком Тейлором, математиком, и основоположником теоремы Тейлора, который жил лет на 250 раньше. Ф. У. Тейлор был замечательным инженером и придумал систематизацию труда. Фактически он утверждал, что любой труд может быть систематизирован и проанализирован, из-за чего Тейлор подвергался нападкам и компании всеобщего презрения. Профсоюзы не любили его за то, что он раскрывал секреты их мастерства и рассказывал из чего же строится процесс на работе, а капиталисты не любили за то, что он считал, что оптимизация процессов должна приносить доход рабочим, а не владельцам компаний. В общем все его называли "смутьяном", представляете. Сейчас, спустя 100 лет, смутьяном назовут того, человека, кто захочет противиться систематизации процессов. Вот все неоднозначно. Угадайте, кто был первым применил практики и теорию Тейлора на практике? Это был Генри Форд. Сейчас конвейер Генри Форда, считается эталоном в области построения конвейерного производства, а 100 лет назад, его тоже считали сумасшедшим. Тейлор написал, интересный труд, под названием "Принципы научного менеджмента", где описал, что одна из основных целей внедрения эффективной организации труда в компании, это рост доходов, как самого предпринимателя, так и работников. Т. е. технология позволяет зарабатывать 2м классам, 1) предпринимателю, 2) рабочему классу, в частности рост их благосостояния (well-being, так звучит благосостоянии на англ. языке). Это очень интересный термин, что означает благосостояние сейчас? Фактически это символ процветания соответствующей социальной группы. Тейлор считал, неправильным потребность предпринимателя получить максимальную прибыль со своих рабочих, за счет максимизации их труда. Что такое благосостояние для предпринимателя? Это получить максимальную прибыль. А для рабочего, это получить максимальную производительность без доп. затрат. Как вы, наверное, догадались, это можно достичь, лишь только тогда, когда работа осуществляется с минимальными затратами для обеих сторон. А значит, тут самое место для автоматизации. "Автоматом" называли в Древней Греции самодействующие механизмы, которые не требовали участия человека. Вернемся к Тейлору, помимо, наверное, очевидных выводов, он затрагивает интересный момент, который у него называется "недовыработка" или работа "с прохладой", отсюда пошло слово "прохлаждаться". Он говорит, что одни и те же люди играя в бейсбол могут показывать высочайшие результаты и будут стремится выиграть, но приходя на работу их работа не будет содержать той самой мотивации. Исключая вот такую вот "медленную" или "неинтересную" работу, он считал, что можно в 2 раза повысить производительность

труда, количество выпускаемой продукции и т.д. Причины, по его мнению, которые способствуют распространению такой вот “прохладной” работы, является:

1. Опасение рабочих, что если они будут лучше и больше работать, то работы другим рабочим не останется, и поэтому других рабочих придется уволить.

2. Ошибочная организация управления компанией, которая, наоборот, позволяет работать прохладно (никаких таймеров, счетчиков и тд, нет)

3. Распространение непроизводительных и грубых методов. Т. е. если ваши рабочие работают “лопатой и мотыгой”, то бессмысленно от них ждать каких-то сверх результатов.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «ЛитРес».

Прочитайте эту книгу целиком, купив полную легальную версию на ЛитРес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.