Alan Carter, Colston Sanger

The Programmers' Stone

Программистский камень

Русский перевод — Сергей Козлов

Оригинал: http://progstone.narod.ru
LaTeX'ed by be9, 2003

Оглавление

Предисловие переводчика

Печальная ирония факта: некоторые люди отличаются от мыслящих машин только тем, что уже давно ни над чем не задумываются.
Феликс Кривин

Изначально проект Programmers' Stone возник у Алана Картера *(Alan Carter)* и Колстона Сэнджера *(Colston Sanger)* в попытках осознать, как же научить программированию, и почему некоторые люди оказываются в программировании гораздо более полезными, чем остальные. И это при том, что, если посмотреть трезво, сама программная индустрия находится в кризисе. Ведь программы-монстры мы видим каждый день. Ведь неспроста такой интерес к той же Linux.

Похоже, что проблема зарыта глубоко, в способе мышления человека при решении задач. Людям свойственны две стратегии мышления: «паковка» *(packing)* и «картостроение» *(mapping)*. По ряду причин люди оказались в ловушке первой стратегии. Переход же между стратегиями не плавный, преодоление барьера требует усилий, а необходимость его преодоления еще нужно осознать...

Programmers' Stone позволяет посмотреть на программирование с немного непривычной стороны, и я надеюсь, что думающим людям он даст качественную пищу для ума. Более того, как это видит Алан Картер, «мы должны изменить парадигму, если действительно хотим перейти из индустриальной эры в эру информационную». И PS может помочь в этом.

Дальнейшее развитие идеи PS получили в проекте Reciprocality (Взаимность). Алан Картер идет дальше и глубже, в поле зрения оказываются время, пространство, разум, причинность, жизнь, мироздание. Хотя многие идеи могут показаться спорными и непривычными, выстроена потрясающе целостная и согласованная картина. Но многое в ней требует дальнейшего исследования и развития. Еще раз — ничего не нужно принимать на веру.

Теперь о переводе. Когда я сам пытался читать в оригинале, то это оказалось трудным занятием, хотя обычно технический английский дается без большого труда. Как мне потом пояснил Алан, PS — это расшифровка магнитофонной записи лекций. Да и носители языка ворчат на трудности при чтении. Заглядывать через слово в словарь и одновременно понимать идеи этой работы оказалось довольно утомительно, поэтому мне и пришла в голову идея перевести. За два года работы перевод стабилизировался. И явных ляпов, я надеюсь, в нем не осталось. Заодно хочу поблагодарить немногочисленных помощников, обсуждение перевода с которыми оказалось очень полезным (в том числе для перевода).

Существует коммуникационный барьер между картостроителями и паковщиками, но существует и языковый барьер. Я постарался языковый барьер убрать; надеюсь, мне это удалось.

Перевод основных текстов закончен, поэтому сейчас идет работа по переводу материалов из переписки, с книгами и ссылками. Материалов много, а я один. Поэтому планы примерно такие — сначала интегрировать ссылки, а потом дополнить их выдержками (а где возможно — полным переводом). Я чувствую, что из «Ресурсов по темам» может получиться что-то действительно интересное.

Не посчитайте меня фанатом, но мне показалось, что идеи проекта очень интересны. Поэтому я считаю важным распространять их. Если окажется, что вы думаете так же, расскажите

об этом месте своим друзьям, выскажите свое мнение, давайте это обсуждать... Вы можете свободно копировать, цитировать, создавать зеркала. Информация должна быть свободной.

Сергей Козлов
2 февраля 2002 г.

Глава 1

Мысли о мышлении

❒ Истоки подхода

На эту работу нас подвигло желание узнать, почему в программной инженерии некоторые люди на порядок, а то и два, продуктивнее большинства остальных. Если бы так было у каменщиков, то строительная индустрия очень заинтересовалась бы причиной. Проблема, конечно, в том, что о каменщике за работой можно снять фильм, а затем в спокойной обстановке разобраться в происходящем. Но невозможно даже хотя бы увидеть, что делают великие программисты, да и они сами не смогли бы объяснить, в чем дело, хотя большинство из них и желало бы это сделать.

Мы знали, что дело не только в использовании лучших крупиц опыта, накопленных в индустрии. Мало одних лишь инвестиционных вливаний и обучения. Не помогают одни только инновационные программы Качества, явно включающие в себя такие целостные концепции, как «Дзен и искусство ухода за мотоциклом» Роберта Пирсига, которые в большей части индустрии считают слишком радикальными, чтобы с ними экспериментировать. Не получается и опереться исключительно на многолетнюю практику или на многие годы академического образования.

Оставался единственный способ продолжить исследования, если уж приверженная объективным метрикам индустрия не смогла найти этот X-фактор, — необходимо было рассмотреть субъективный опыт работающих в области программной инженерии людей.

Достижение понимания происходящего заняло долгое время, хотя ключевые идеи большинству из нас уже были известны. Попутно мы изучили склад мышления успешных программистов и смогли разработать упражнения, которые определенно помогли многим.

Этот материал создавался несколько лет и представляет собой смесь идей, эмпирически подправленных экспериментами и позднее собранных в единую логичную картину, а также материал, полученный на основе этой картины.

Цель этой работы — донести те элементы «опыта» или «взглядов», на которые повсеместно ссылаются, но редко описывают. Многие темы входят в круг вещей, обсуждаемых программистами за пивом. Кажется странным, что никто не задается вопросом: как проблемы, которые программисты считают наиболее важными, соотносятся с «формальными» структурами современной инженерии. Здесь мы делаем именно это.

Выяснилось, что, как только мы обнаруживаем себя на этой волне, большинство программистов получает удобный случай поместить волновавшие их долгие годы проблемы в ясный рабочий контекст. Мы предлагаем вам расслабиться и хорошо провести время!

❒ Картостроение и программная инженерия

Программная инженерия находится в ужасно плачевном состоянии. Так называемый «кризис программного обеспечения» был сформулирован в 1968 г., но, несмотря на тридцатилетние уси-

лия и сотни опубликованных фундаментально новых, как предполагалось, концепций, общее состояние индустрии остается ужасающим. Проекты выходят за рамки бюджетов или превращаются в неразгребаемые кучи. Делать оценки — это черная магия, и многие проекты решают вчерашние проблемы пользователей, а не сегодняшние. Техническое качество большей части кода отвратительное, что вызывает проблемы с надежностью в процессе сопровождения и высокие затраты на эксплуатацию. В индустрии все еще существуют отдельные программисты и группы, которым нравятся ошеломляющие и воспроизводимые успехи. Есть множество способов измерения продуктивности программистов, но некоторые программисты оказываются в сотни раз более продуктивными, чем большинство, независимо от выбранного способа подсчета. Если бы вся индустрия работала так же, как малая доля великолепных программистов, то экономический эффект был бы потрясающим. Если бы получалось писать сложные, надежные программы быстро и дешево, увеличился бы интеллектуальный потенциал общества и стало бы возможным многое: от совместного использования автомобилей до реалистичного контроля за общественной безопасностью.

Эту проблему можно понять в рамках предлагаемой модели. То, что мы представляем как социально обусловленное общепринятое мышление (называемое *паковкой* — packing), основано на действии. Чтобы быть хорошим каменщиком, паковщик должен знать, что делает каменщик. Но что же делает программист? Самая развитая из используемых паковщиками моделей программирования — концепция *программной фабрики*. В ней пункты требований от заказчиков входят в одну дверь и обрабатываются рабочими по описанной в руководствах процедуре. Когда же производственная линия завершает свою работу, программы появляются из другой двери. Эта модель работает на автомобильных заводах.

Беда в том, что аналогия с автозаводом не слишком хороша. Большая часть автозавода заполнена работниками, применяющими различные механизмы для изготовления автомобилей, но где-то на задворках есть небольшой офис, где другой работник определяет, как использовать ресурсы фабрики, чтобы сделать как можно больше одинаковых автомобилей.

Работники программной фирмы не похожи на работников цехов автозавода. Сегодня последних можно заменить роботами, но люди, использующие творческий подход для управления заводом, были и остаются нужны. Программисты делают ту же работу, что и офис на заводских задворках, и мы не можем научиться ничему из того, что они делают, бегая по цехам.

Паковщики, защищающие основанные на процессе и не склонные к компромиссам программные фабрики, на самом деле заявляют, что способны реализовать искусственный интеллект, который бы имитировал разработчика, работающего на поток. Они хотят сделать это с помощью людей, перемещающих бумажки туда-сюда и играющих тем самым роль компьютера. К сожалению, паковка просто не достигла понимания производства в сфере программного обеспечения и приводит к ужасной путанице. Это означает, что иногда паковщики говорят очень глупые вещи.

Чтобы понять, что же на самом деле делают программисты, требуется другая стратегия мышления (называемая *картостроение* — mapping), поскольку вся суть программирования состоит в том, чтобы впитать в себя возможности системы, природу задачи, потребность — и сформулировать свое глубинное понимание на некотором языке программирования. Детально исследовать потребности и понять их так, чтобы это помогло уследить за всей сложностью — вот где собака зарыта. Решение задачи картостроителем может привести к красивым, компактным, элегантным программам, в которых ошибкам будет негде спрятаться. Мышление картостроителя способно писать программы, но способ, которым оно это делает, невозможно описать на языке паковщика, ориентированном на действия.

А паковщики заключают, что хакеры «необъяснимы», и отвергают их работу, говоря, что сложность по своей внутренней сути неподвластна человеческому пониманию и что мы должны создавать все более сложные процедуры, чтобы отречься от ответственности.

К счастью, многие управленцы в организациях продолжают поддерживать мышление, основанное на интуиции и эмпирических соображениях, не пытаясь поместить его в прокрустово

ложе процедурных инструкций. Это трудный путь, но единственный, приводящий к конечному результату.

Очень важно понять, что картостроение — это не еще одна процедурная методология, которую нужно загрузить в голову паковщика. Это другой способ посмотреть на все вещи сразу. Необходимо осознать, что это действительно возможно — взять на себя личную ответственность за работу, вместо того чтобы переложить ее на используемую процедуру.

Программирование настолько близко к чистому картостроению, насколько вы сможете выбраться за пределы своего черепа. Именно поэтому оно приносит удовольствие. Это путь, ведущий к бесконечным открытиям, пониманию и обучению.

Объектно-ориентированный подход (ООП) и картостроение связаны друг с другом интересным образом. ООП зачастую очень по-разному воспринимается картостроителями и паковщиками. Карта картостроителя — это такой род объектной модели, где существует множество объектов и взаимосвязей. Картостроители видят ООП как элегантный способ разрабатывать программы, когда они уже поняли задачу. Паковщик же смотрит на ООП как на способ бродить вокруг проблемной области, создавая программные объекты, а затем просто соединить их, как получится. Таким образом, ООП воспринимается как процедурный механизм для перехода от задачи к готовой программе, без вмешательства понимания. Если бы было возможно учесть абсолютно все аспекты проблемной области и не нужно было бы заботиться об эффективности, то такой подход мог бы даже и сработать. Но в действительности при проектировании и категоризации объектов всегда требуется хороший вкус, поскольку необходимо создавать такие программные объекты, которые хорошо бы соответствовали объектам реального мира, но которые можно было бы соединить вместе и получить жизнеспособную компьютерную систему. Здесь нужно понимание, и такая работа является строго работой картостроителя. Теперь становится ясно, почему некоторые проекты, использовавшие ООП, зашли в тупик, породив кучу реальных и вспомогательных объектов, использующих множественные избыточные схемы адресации для взаимодействия *через* Брокеры Объектных Запросов (ORB), без ясной концептуальной целостности способов рождения объектов, их уничтожения и протоколирования. Программисты-паковщики часто настолько плохо контролируют свои объекты, что вообще теряют их, и все заканчивается утечками памяти, приводящими к падению программы. Решение паковщика в этом случае — покупка средств обнаружения утечек памяти, а не восстановление контроля над своими объектами, чтобы все работало как надо.

❒ Картостроение и тотальное управление качеством (ТУК)

После Второй мировой войны американцы послали в Японию д-ра Д. Э. Деминга, чтобы он помог японцам привести в порядок их промышленность, являвшую собой странную смесь средневековья и индустриальной эпохи и вдобавок разрушенную войной. Деминг предложил несколько идей: собрать статистику деятельности при массовом производстве; попросить работников, занятых этой деятельностью, подумать о том, как ее можно улучшить; удостовериться в том, что каждый работник понимает, что он делает. Эти идеи в дальнейшем развились в то, что мы сейчас называем *тотальное управление качеством* (Total Quality Management) — ТУК.

Результат, как нам говорят, получился сверхординарный. За одно поколение японская промышленность взмыла ввысь и перешла от кустарного производства велосипедов к мировому лидерству в высокотехнологичных производствах — кораблестроении, автомобилестроении и электронике. «Японский метод» был реимпортирован на Запад и узаконен в ISO 9001, международном стандарте «Качества», на который бизнес потратил целое состояние. ISO 9001 фокусируется на задании процедур для чего бы то ни было, с большим количеством проверок и простановки галочек. В общем плане ожидаемых выгод от его использования пока не наблюдается; не достигли ошеломительного успеха и некоторые организации, взявшие на вооружение наработки Деминга и его последователей.

Осознание важности картостроения предлагает иной взгляд на то, что произошло в Японии. Картостроение определенно можно пробудить травмой. Вот некоторые возможные способы травмировать людей:

1. Скинуть на них атомную бомбу. Дважды.
2. Похоронить их застойное, предсказуемое феодальное общество.
3. Сказать им, что завтра придут захватчики.
4. Оставить их без ужина.

Чтобы вечером поесть, человек вынужден пробудить свою способность к воображению. Поэтому к тому моменту, когда Деминг прибыл в Японию, люди, с которыми ему пришлось работать, уже стали картостроителями. Поголовно все и сразу. Возможно, все, что нужно было сделать Демингу, это взять листок из «Необычайных приключений Билла и Теда», встать на чайной церемонии и закричать: «Будьте как можно внимательнее друг к другу!»

Когда это сработало так эффективно, то, наверное, Деминг вместе со своими коллегами был очень впечатлен и начал работу над методами, используя которые, его работники смогли бы стать еще внимательнее, создавая мощную индустриальную культуру. Но методы при этом имели скрытое требование: все это работает только для картостроителей!

Когда «японский метод» стал только-только появляться, картостроители из Японии приехали в Америку и с присущим им энтузиазмом и картостроительными привычками показали американским рабочим, как задавать интересные вопросы о своей работе, как собирать информацию, интерпретировать ее с пользой и улучшать рабочий процесс. Они показали им, как составлять описания своей деятельности, изучать эти описания и находить проблемные места.

Это работало прекрасно, но дело было все-таки в том, что людей случайно обучили картостроению.

Когда идеи ТУК нашли широкое применение, такое обучение картостроению на случайной основе просто исчезло. Идеи были проданы индустрии паковщиков, воодушевившейся результатами, но она не смогла увидеть главного в этом приобретении — дани, отдаваемой мудрости и размышлениям.

Даже креативные менеджеры из индустрии высоких технологий могут натолкнуться на коммуникационный барьер. Для многих работников идеи ТУК напоминали то, что провозглашал отец научного менеджмента Фредерик Тейлор. Тейлор дал нам массовое производство еще до того, как у нас появились роботы, — сделав роботами людей. Наверное, не стоит так смотреть на вещи, но в Лос Аламосе имитировали работу электронной таблицы, усаживая людей рядами за столами с арифмометрами! Он был таким наркоманом контроля, что имел обыкновение каждую ночь привязывать себя к кровати, борясь со своим патологическим страхом выпасть из кровати. Его лозунг был таков: «Оставь свои мозги за дверью и внеси сюда только тело». Наша культура, от школы до законодательства и правовых концепций, по-прежнему находится в тисках тейлоризма. В этой ситуации при внедрении ТУК без ясного понимания необходимости картостроения в худшем случае получится тупой тейлоризм. А в лучшем случае мы удивимся, почему мы делаем то, что делаем.

В некоторых организациях результат был трагичным. Появилась одержимость в составлении детальных отчетов, упрощенчестве и создании отвратительных описаний работы, принимаемых за абсолютные нормы поведения. Все нужно было делать в рамках конкурентной борьбы, предлагаемой моделью паковки, а не сотрудничества, естественного для картостроителей. Аудиторы ISO 9001 стали рыскать по офисам с проверками правильности оформления бумаг, стремясь подловить работников на мелких нарушениях инструкций, совсем как у Кафки. В некоторых организациях работники стали больше озабочены тем, как избежать наказания

за микронарушения правил составления бумаг, чем выполнением самой работы, смысл которой терялся за соблюдением ритуалов. Вспомните историю Фейнмана про твердотопливные ускорители на Челленджере. Некоторые люди до сих пор считают, что так и надо.

Хорошее ТУК собирает опыт на рабочих местах и концентрирует это знание в виде списков тех вещей, на которые стоит обратить внимание. Эти списки просто напоминают картостроителям о тех моментах, для рассмотрения которых им понадобится применить свой здравый смысл там, где это нужно. Паковщики же вместо этого считают, что нужно просто проставить галочки с той скоростью, с которой возможно находить оправдание для того, чтобы это сделать. Сколько рассмотрения «достаточно» паковщику?

Поскольку процедуралистская оргия распространилась под флагом «Качества» в слишком многих местах, это привело к тому, что реальное управление качеством, которое нацелено на выполнение для потребителя настолько хорошей работы, насколько можно себе представить, полностью выпало из поля зрения.

Что самое смешное, есть некоторые организации (которые, похоже, знают толк в разумном использовании информационных технологий), изобретшие своего рода «настоящий процедурализм». Компании, осуществляющие банковское обслуживание по телефону, перестали претендовать на предоставление «разумного» сервиса реальными людьми и открыто признали анонимную, процедурную природу своего бизнеса. Это позволило им грамотно продумать свои процедуры и сделать их очень хорошими: круглосуточно и без больших затрат удовлетворяющими потребности клиентов. Такая практика сильно контрастирует в глазах у многих с образом грубого клерка-столоначальника, являющего собой карикатуру на помпезного диккенсовского начальника похоронного бюро, который считает глупые «постановления», которыми он руководствуется, проблемой его клиентов, а не своей собственной.

Очень успешные финансовые организации поняли, что есть процедуры, которые хорошо исполняются компьютерами, в других же обстоятельствах лучше проявляют себя опытные работники. Эти организации анализируют свой рынок сбыта математическими методами с помощью компьютеров, а последнее слово оставляют за человеком. Они могут использовать различные критерии для описания обоих аспектов системы и оценивать эффективность различных алгоритмов и торговцев.

Это дает возможность применить подход картостроителей. Если у нас есть «настоящее ТУК», «ложное ТУК» и «настоящий процедурализм», то мы можем сказать вот что:

Настоящее ТУК	Настоящий процедурализм
Ложное ТУК	Ложный процедурализм

и спросить, есть ли какие-нибудь примеры организаций с «ложным процедурализмом», клянущихся, что они безмозглые автоматы, а на самом деле предающихся безумству картостроения? Как насчет операции британской армии в Порт Стенли в 1982 г.? Вспомним, что армия — это организация, сталкивающаяся с достаточно сложными задачами. Даже те, кому отвратительны любые конфликты, могут извлечь урок того, как сплотить мир, разобравшись, что делает более сплоченной армию. Являются ли британские военные «ложными процедуралистами»? Интересно взглянуть на эти вещи с позиции картостроителя, потому что в таком случае мы сможем за официальными документами и казенным языком увидеть то, что делает эта организация. Идея, что каждый в любой момент следует правилам, затрудняет понимание британской армии в действии. Но если мы предположим, что в британской армии много картостроителей, следующих правилам до того момента, пока они не увидят, что правила перестают действовать, ситуация проясняется. Мы можем также сравнить обычаи британской армии и армии США. Американцы всегда открыто предпочитали подход, больше напоминающий «настоящий процедурализм» банковского обслуживания по телефону. Они явно предпочитают действовать по процедуре и заставляют своих картостроителей составлять эти процедуры наилучшим образом. Когда этот

подход работает, то действительно работает очень хорошо (как, например, в Персидском заливе во времена «Бури в пустыне»). Тем не менее, он неустойчив, ибо не дает паковщикам, использующим процедуры, достаточно свободы, чтобы реагировать на изменяющиеся условия. Это приводит к неэффективности, как было при вторжении на Гренаду.

Урок прост. Если в основе не лежит картостроение, ТУК превращается в трагикомедию. А если лежит, то Качество может учить и побуждать. Энтузиазм и удовольствие от работы, о которых говорят защитники TQM есть не что иное, как обычное настроение картостроителя!

В этой модели системное мышление — подход, защищаемый Питером Сенге в «Пятой дисциплине», — может рассматриваться как набор полезных концепций и технологий картостроения, оптимизированных для вопросов управления персоналом.

❒ Заставь себя!

В наше время живет гораздо больше паковщиков, чем картостроителей. Одна из целей этой работы — показать эффективные методы картостроения, но другая состоит в том, чтобы объяснить, почему озарения многих из нас не находят поддержки среди остальных. Нам нужно научиться определять те моменты, когда наши заботы как творцов-программистов непонятны для коллег-паковщиков, чтобы дать им возможность освоиться со сложными явлениями, требующими некоторого времени на обдумывание. Мы также должны понять, что правота не всегда ведет к популярности, но что личная заинтересованность в качественной работе зачастую приносит больше удовлетворения и меньше стресса, чем страусиное поведение.

Мы должны также осознать, что возможно эффективное взаимодействие с картостроителями, даже с теми, кто далек от нашей области. Признавая существование специфического коммуникационного барьера при общении с одними, нам также нужно уяснить, что с другими взаимодействие часто оказывается гораздо проще, чем можно было бы предположить.

Необходимо также четко осознавать границы нашей ответственности. Если в разговоре с заказчиком выясняется, что он не ухватывает существенные моменты вопроса, помните, что наша личная, поставленная нами самими цель — найти наилучший ответ — не обязательно означает заставить заказчика принять один лишь этот ответ. Любое размышление, которое допускает одну стратегию, обычно допускает несколько других стратегий, каждая из которых имеет свои достоинства и недостатки. Их всегда можно принять во внимание и получить удовлетворение от хорошо проделанной работы по изучению возможностей и обоснованию своего выбора заказчику. Если, обладая полным пониманием, заказчик делает выбор, который вам кажется глупым, то каким еще способом организация заказчика может чему-то научиться?

Вы не должны спасать весь мир, только свой небольшой кусочек и еще чуть-чуть, если сможете!

❒ Terra incognita

В своей книге *Peopleware* Том де Марко и Тим Листер предполагают, что великие программы создаются «однородными» *(gelled)* командами, и предлагают брать на себя инициативу по повышению сплоченности команд. Глядя на «однородные» команды, мы можем заметить присутствующую там легкость во взаимоотношениях и эффективность работы. Но добавим в уравнение концепцию картостроения, и картина изменится. «Однородные» команды выглядят скорее как группы эффективно взаимодействующих друг с другом картостроителей, поскольку они могут ссылаться на общую мысленную карту ситуации с помощью нескольких, возможно, странно звучащих слов. (Как-то раз разработчики назвали подсистему буферизации в системе связи с гарантированной доставкой «Фабрикой спагетти». Дело было в неких колечках, свободно летающих в воздухе.)

Они не могут просто быстро обмениваться информацией о своих картах — они могут брать отдельные кусочки своих карт и перемещать их. Они могут обмениваться кусочками своих карт. Они вместе, как команда, обладают очень быстрой реакцией. Все знают, что происходит, и упорная работа ведется в голове у каждого. Они не петушатся и не тратят время на несинхронные действия. Они уважают друг друга, хотя их вкусы в музыке, политике и еде могут разительным образом отличаться. От суммарного роста производительности захватывает дух; тот, кому посчастливилось работать в такой команде, знает, о чем идет речь.

На что нужно тратить время, так это на то, чтобы каждый обладал пониманием происходящего. Жить тогда станет приятнее, поскольку в пять вечера каждый будет испытывать чувство успеха.

Эта ситуация не случайна, оно воспроизводима.

❒ Пакеты знаний, фантазии, карты и понимание

Как инженеры-программисты, мы могли бы описать обучение как формирование ассоциаций между объектами. Небо голубое. Дождь в Испании выпадает главным образом на равнине. Эти элементы познания мы можем назвать *пакетами знаний*: маленькими кусочками истины (или заблуждений), которыми мы обладаем.

Пакеты знаний можно накапливать долго. Раннее обучение (направляемое взрослыми) для большинства детей практически полностью фокусируется на их приобретении. Вещи, которые следует и не следует делать. Методы выполнения задач. Данные, которые следует запомнить, чтобы потом в нужный момент вспомнить.

Тонкость с пакетами состоит в том, чтобы идентифицировать ключевые особенности ситуации и определить действие, которое нужно выполнить. С помощью пакетов знаний можно получать «пятерки» и научные степени, управлять автомобилем и даже знакомиться с представителями противоположного пола. Высокопрофессиональные пользователи пакетов знаний могут набить свои головы мегабайтами налогового законодательства и стать бухгалтерами, получая при этом кругленькую зарплату. Некоторые политики опускают стадию распознавания паттерна и используют один единственный универсальный пакет знаний для всего.

Конечно, мы не складываем пакеты знаний в своей голове стопкой, как тарелки. С самого раннего возраста наша естественная реакция на каждый новый пакет — это вопрос «почему?».

Мы стремимся связать друг с другом разрозненные пакеты, чтобы создать знаниевую структуру — мысленную карту, которая дает нам понимание причин и следствий внутри ситуации. Это понимание позволяет нам получать решение для любой проблемы в рамках этой ситуации, вместо того, чтобы использовать бездумно заученную реакцию.

В дальнейшей жизни мы должны периодически проводить время в размышлениях или фантазиях, путешествуя по связям между теми вещами, которые мы знаем. Это расширяет нашу интегрированную карту и позволяет идентифицировать на карте те структуры, которые могут быть применены в различных областях. Затем мы можем получить более подробную карту, в которой то, что математики называют «изоморфизмом», дает то, что программисты называют «наследованием», позволяя нам повторно применять знание.

Мы реорганизуем наши мысленные карты, чтобы получать более простые выражения и иметь возможность удержать в уме больше понимания. Когда мы находим более простой способ посмотреть на вещи, становится трудно вспомнить, как это было, когда предмет казался сложным, и это означает, что мы растем. Если есть понимание, то где кончается личность и начинаются собственно данные? Что касается пакетов знаний, там граница очевидна.

Мы становимся экспертами в применении способов мышления, позволяющих исследовать наши карты и те пакеты знаний, которые пока еще никуда не подсоединены. Вероятно, в основе мышления лежат неврологические механизмы, но при этом должна происходить некоторая деятельность по распознаванию абстрактных структур. Мы учимся использовать наши мозги.

Без понимания не может быть разумных действий. Без мысленных карт не может быть понимания. Без размышления не может быть мысленных карт, только пакеты знаний.

Существуют компьютерные структуры данных — их называют *онтологиями*, — содержащие массу утверждений (истин), объединенных в сетевую структуру на основе определенной логики предикатов. Например, база данных CYC может использовать карты значений для естественного языка, чтобы интерпретировать подписи к фотографиям и подобрать образцы для тех снимков, которые требуются журналистам.

❒ Картостроители и паковщики

Или, по крайней мере, все это стоит того, чтобы оказаться правдой. К сожалению, мы происходим из индустриальных и аграрных обществ, где один день был очень похож на другой. Эффективность зависела от согласованной групповой работы по выполнению простых задач. С другой стороны, не слишком требовалась изобретательность. Мы создали общественный порядок, который учит людей складировать пакеты знаний и фокусироваться на действии. Размышление («фантазии») в начальной школе подавляется. Мы внимательно наблюдаем за детьми и принимаем во внимание малейшие отклонения от норм поведения, основанных на действии. Некоторые родители даже считают своих детей психически неполноценными, если те не желают заниматься определенным видом спорта.

Нельзя легко научить ребенка мыслить. В отличие от выполнения физических упражнений, субъективный опыт должен обсуждаться.

Нелегко установить, насколько успешно проистекает мышление в человеке. Только посредством тщательного обсуждения или, наблюдая за долговременными результатами мыслительного процесса ребенка, можно выявить способность к эффективному фантазированию.

Итак, в нашей общественной истории нет ничего, что побуждало бы родителей или учителей учить мышлению. Нет ничего, что сделало бы обучение размышлению школьным приоритетом.

На самом деле верно обратное. Когда дети пытаются размышлять, являющееся следствием этого отсутствие физической активности наказывается. Когда процесс мышления в ребенке требует задать вопрос и получить на него ответ, занятые взрослые редко откликаются. Из того места, где размышления оказались успешными и было достигнуто понимание, для ребенка может возникнуть помеха. Если нужно сделать еще 15 простых примеров на сложение, ребенок заскучает, будет наказан и назван неспособным выполнить простую задачу, при том, что ничто не может быть дальше от истины, чем это.

Обратите внимание, что, хотя взрослые в каждом конкретном случае наказывают за разные проступки, дети-то заняты одним делом — размышлением. На самом деле таким способом приучили многих людей считать, что рефлексивное мышление само по себе социально неприемлемо!

Считается, что мышлению учат в университетах, но, когда полный курс обучения тридцатилетней давности ужимается в один год, как это происходит в большинстве технических предметов, такое случается редко.

На работе образованных людей до сих пор считают способными мыслить, и, действительно, все программисты должны в какой-то степени уметь это делать, просто для того чтобы сделать хоть что-нибудь. Мы принадлежим к числу наиболее мыслящих людей в обществе, но нам все еще далеко до однородной группы. Некоторые из нас достигли в этом больших успехов, или переживают по этому поводу меньше других. Еще раз — этому не учат, а на рабочем месте, которое встроено в общество, культурная среда часто остается основанной на пакетах знаний и действиях, а не на мысленных картах и понимании.

Это приводит к двум различным группам в обществе. Картостроители главным образом применяют когнитивную стратегию составления и интеграции мысленных карт, для того чтобы впоследствии считывать с них решения для конкретных задач. Они быстро находят методы

достижения своих целей, сверяясь со своими картами. Паковщики становятся экспертами в поддержании большого набора пакетов знаний. Их единственная цель — выполнить «правильное» действие. Стратегии для разрешения «хэш-конфликтов», когда при заданных условиях существует несколько вариантов действий, представляют собой особый случай.

❒ Как восстановить картостроение

Принципиальное преимущество нашего вида над другими состоит в нашей универсальности. Мы можем выживать в более широком интервале температур, чем прочие создания, но, что более важно, мы изобретательны. Артур Кларк и Стенли Кубрик отметили эту изобретательность в знаменитой заставке «от берцовой кости к космическому кораблю» в фильме «2001».

Мы все картостроители, независимо от того, насколько мало мы используем эту способность. Те из вас, кто, проводя время в одиноких прогулках, барах для металлистов или где там еще, чувствует небольшой дискомфорт, пока вдруг внезапно не находится решение для какого-то вопроса (а вы даже не знали, что его ищете!), уже пользуются ей. Вы знаете, кто вы!

В противном случае, есть простой способ начать. Такой простой, что дети, упорно пытающиеся быть естественными картостроителями, часто его находят. Заведи себе воображаемого друга, такого же умного, как ты сам, но не имеющего ни малейшего представления о мире. Кого угодно, с кем вы могли бы общаться — вам никому не надо рассказывать о том, что легче всего вам говорить с персонажем из мультфильма 1960-х по имени Astronut с антенной на голове. Или больше подойдет хитрый средневековый алхимик Шона Коннери в «Имени розы». Объясни все своему воображаемому другу. Для чего это. Откуда это. Где это происходит.

Сначала для этого упражнения потребуется полное внимание, но через некоторое время нахождение логических связей между пакетами знаний становится таким же автоматическим, как вождение автомобиля, и ваше внимание начинают привлекать только необычные ситуации: кусочки вашей карты, которые нужно заполнить, либо противоречия, которые нужно разрешить. Это работает. Теперь, когда ваши карты строятся, можно обсудить методы, поскольку мы все знаем, о чем говорим.

❒ Пути картостроителей и паковщиков

Будет неожиданностью обнаружить, что вокруг нас существуют два отличных друг от друга состояния ума. Это похоже на то, как если бы вы узнали, что кто-то, кого вы давно знаете, неграмотный. Сначала вы изумитесь: это невозможно! Но потом поймете, что кто-то другой может жить совсем другой жизнью, хотя на поверхностный взгляд это выглядит почти так же.

В этом разделе мы рассмотрим характерные черты этих двух стратегий. Когда мы это сделаем, многие беды современной жизни, особенно в области высоких технологий, станут просты и понятны — вот признак полезной теории! Помните — большинству людей, картостроители они или паковщики, нет повода думать, что есть какой-то другой склад ума — не такой, как у них.

Что такое паковка? Это когда ты перестаешь задаваться вопросом «Почему?». Перестаешь обновлять свою карту мира и поэтому не выявляешь многие глубинные структуры, которые картостроители используют для «плутовства». Ты учишься медленнее, поскольку усваиваешь маленькие пакеты знаний, которые ты не можешь проверить до конца, и поэтому неожиданно возникает множество маленьких проблем. Ты редко доходишь до той точки, когда бóльшая часть карты отсортирована, и можно просто увидеть, что получится в остальной ее части. В таких требующих интенсивного мышления областях, как физика и математика, картостроители могут понять достаточно, чтобы получить аттестат с хорошими оценками за две недели, хотя в большинстве школ тратится более трех лет на зубрежку пакетов знаний, которые остаются лежать в памяти необработанными, ибо дети послушны и не предаются фантазиям. В наш век информации это не самый эффективный путь.

Без карты мира, выверяемой относительно самой себя и объясняющей буквально все вокруг, очень трудно быть уверенным в том, что делать. Подход, который ты вынужден брать на вооружение в любой ситуации — яростно сканировать память, пока не найдешь маленький пакет знаний, более-менее подходящий к ситуации (в основе всего лежит немного размышления, но оно прекращается как можно скорее). Затем ты перечисляешь те элементы, которые вроде бы подходят, и убеждаешься в том, что это именно та ситуация, так что ответное действие определяется твоим «знанием».

А вот твой приятель извлек другой пакет «знаний», и вы начинаете «спор», в котором твой друг перечисляет все те элементы твоего знания, которые не подходят к ситуации, и говорит, что ты ошибаешься, а он прав; и ты делаешь то же самое. Ты не пытаешься построить карту, которая включила бы в себя элементы обоих ваших знаний и сделала бы правильный ответ наглядным, поскольку у тебя нет доступа к необходимым для этого навыкам картостроения, и, кроме того, не имея подобного опыта, очень трудно поверить, что это возможно сделать за приемлемое время. Без ясности, появляющейся при наличии даже наполовину упорядоченной карты, ты скорее сделаешь что-то неэффективное к сроку, чем создашь что-то, что могло бы даже работать. Когда позже возникнут серьезные проблемы, ты скажешь, что тебе не повезло.

Следствия идут дальше. То, что у тебя нет большой карты, означает, что ты часто не понимаешь происходящего даже в знакомой обстановке (дома или на работе). По твоим предположениям, это означает, что ты не усвоил соответствующий пакет знаний, так что представляется как внутренний просчет с твоей стороны. В конце концов, с детства тебе внушали, что хорошие мальчики и девочки приобретают пакеты знаний и складывают их стопкой в голове как тарелки, а не делают этого ленивые.

Ты также слишком озабочен достоверностью. Картостроители обладают богатой, сильной, связной структурой, которую они могут детально исследовать, по которой могут выверять ситуацию и собственные действия. Логика для них заключается в том, чтобы соответствовать карте и честно сознавать, когда она перестает работать. Это не проблема, они просто изменяют ее до тех пор, пока она вновь не становится «логичной». Без картостроения ты вынужден использовать хрупкие цепочки вывода, которые поддерживаются только с одного конца. Поскольку они хрупки, тебя очень волнует, чтобы каждая связь была абсолютна, достоверна, полностью корректна (чего на самом деле никогда не достичь). Тебе приходится отказываться от недостоверных аргументов (хотя, по иронии судьбы, они могли бы быть таковыми, если бы твоя карта была побольше) и зачастую ограничиваться действиями, по поводу которых ты можешь убедить в себя в том, что они абсолютно правильны в мире, изменчивом по своей натуре.

Вопрос достоверности затем становится доминирующим. Люди не желают думать о чем-либо (строить хрупкие цепочки) до тех пор, пока они не станут «уверены», что «процедура» даст гарантированный результат, потому что, как им кажется, мудрые люди поступают именно так.

Тебя поглощает страх оказаться «в заблуждении» из-за идеи о том, что «добро» достигается правильным пакетом знаний, действующим в любой ситуации. Мысль о том, что мир замкнут и полностью постигаем (но не тобой), принимается по умолчанию. Мысль о новом становится настолько нежелательной, что ты редко обращаешь внимание на это новое, когда оно происходит, хотя картостроители постоянно подмечают новизну. Твой подход фокусируется на действиях, которые ты не смеешь «критиковать», даже если их бесполезность, или даже вредность (в смысле снижения производительности), очевидна. Ты настаиваешь на своих специфических (ритуальных) действиях, предписанных в твоей работе, даже когда твоя карта уже достаточно хороша, чтобы принять под личную ответственность достижение целей, что более приличествовало бы твоему достоинству.

У некоторых люди настолько мало опыта непосредственного понимания, добываемого с помощью картостроения, что они не могут поверить, что можно что-то понять, до тех пор, пока кто-то еще не расскажет им в подробностях, как делать абсолютно все. Они верят, что един-

ственной альтернативой всеобщей регламентации может быть лишь полная анархия, но только не группа людей, способная получить результат.

Теперь, когда ты привык беседовать с воображаемым другом о своей карте мира, продолжаешь находить в ней дыры и латать их, ты стал менее зависимым от ее текущего состояния в любой конкретный момент времени. Иногда ты все же зависишь от карты, если находишь абстракцию, которая была прекрасным сюрпризом, когда ты ее обнаружил, и была полезной, но теперь должна уйти. Важно всегда помнить, что удовольствие только нарастает: если открытие чего-то доставило удовольствие, то открытие чего-то более глубокого будет еще большим удовольствием. Обычно ты даже не возражаешь, когда твой воображаемый друг откалывает кусочки карты, если они не работают. Поэтому ты не возражаешь, если настоящие друзья тоже это делают! Когда вы видите вещь с разных сторон, то вы пытаетесь понять карты друг друга и прорабатываете различия. Две несогласованные карты часто показывают путь к более глубокому взгляду на вещи.

Великие мыслители — картостроители. Они редко занимаются возведением зданий огромной концептуальной сложности. Скорее, они показывают нам, как посмотреть на мир проще.

Картостроители организуют свое обучение как внутренний процесс в ответ на внешние и самостоятельно генерируемые стимулы. Паковщики строят свое обучение как еще одну задачу, которую требуется выполнить, обычно в классе, используя соответствующее оборудование. Особенно в ранние годы эффективное обучение картостроителя требует внутренних методов для исследования взаимосвязей концепций и распознавания истин, в то время как эффективное обучение паковщика сконцентрировано на навыках запоминания.

Особенности обучения картостроителя требуют более высоких затрат, чем обучение паковщика, и у этого есть последствия. Концентрация на сжатом, структурированном знании означает, что плохо структурированные, не имеющие отношения к предмету соображения могут побудить к решению непропорционально большого числа проблем, не имеющих прямого отношения к проблеме, над которой размышляет картостроитель. Если ребенок пытается понять новую идею с точки зрения как можно большего из того, что он знает, то очень вероятно, что разум ребенка будет рассредоточен над стольким «базовым знанием», насколько это уже возможно. Необходимость рассмотреть вопрос: «Отнесу ли я сегодня книги в библиотеку?» приносит с собой концептуально связанные вопросы: «Где мой ранец?», «Будет ли дождь?», «Будет ли дождь завтра?» и т. д., которые становятся дополнительной нагрузкой для ума, которая у ребенка-паковщика при аналогичных обстоятельствах просто не возникает. У ребенка-паковщика (например) просто никогда не возникает такой формы растекающихся потоков, являющихся следствием из кривых спроса и предложения в экономике (которые могут на самом деле иметь такое же представление, которое используется для хранения, скажем, составляющих понимания термодинамики), которые будут замещены таким простым вопросом о библиотечной книге.

Восприятие факта и готовность к следующему также различны в картостроении и паковке. Ум картостроителя должен исследовать этот факт и сравнить его с базовым знанием, чтобы увидеть: либо он является следствием, уже имеющим место в концептуальной карте мира, либо этот факт представляет новое знание, которое требует структурных изменений.

Картостроители, вероятно, гораздо более обеспокоены сравнительной надежностью информации. В то время как паковщики стремятся рассматривать знание плоско, как ряд имеющих место быть утверждений, картостроители стремятся связать утверждения взаимными ссылками, чтобы проверить их и свести к более глубоким истинам. Более чем вероятно, что картостроители работают с совокупностью утверждений типа: «Если X истинно, то Y также должно быть истинно, Z несомненно истинно, а W — чепуха, хотя все говорят, что это правильно». Картостроители, скорее всего, недоумевают над отсутствием у паковщиков доводов.

Особенность склада ума паковщиков, которая заставляет картостроителей лезть на стену, заключается в том, что паковщики редко замечают изъяны своей логики и не слышат их, даже когда произносят. Хуже того, когда на эти изъяны им указывают, они с большой вероятностью

оправдают свою логику, бодро признав изъяны, административной целесообразностью. Обоснованность их собственного мнения не так важна, как закрепленное обучением поведение, и кажется, что они не видят смысла соотношений, когда проводят анализ стоимость/полезность. Это происходит потому, что паковщики не создают обобщенные концептуальные картинки из максимального количества знаний, которым они обладают. Картостроитель может обратить внимание на факт, но это один факт среди многих других. У паковщика нет концептуальной картины ситуации, которая выделяет важные положения, поэтому в своих действиях он в основном руководствуется набором процедурных реакций, которые задают выбираемое к исполнению действие. Процедура выбора действия напоминает лотерею. Для картостроителя наличие факта, который должен быть помещен на карту, но не подходит к ней, вызывает подозрения к карте. Эта неувязка может перемещаться как складка на ковре, пока не придет к чему-нибудь важному. Обе стороны согласны, что им следует делать «логичные» вещи, но два человека могут не сходиться в логике, когда один видит соотношения, которые другой старается не замечать.

У картостроителей много хороших идей, основанных на глубоком погружении в соотношения, которые паковщик редко когда принимает во внимание.

Частично необычайная гибкость и скорость обучения картостроителей происходит из преимуществ поиска понимания, а не данных, но остальное происходит из-за тщательного изучения того, с чем они имеют дело. Для картостроителя совершенно естественно проводить каждый свободный миг в верчении предмета в голове (обдумывании проблемы), а все выходные — полностью фокусируясь на нем. Фокусирование картостроителя — страшная вещь. За несколько часов он может достичь таких результатов, которых группа паковщиков будет достигать месяцами. Каждый IT-менеджер, который видел картостроителя в действии, это знает.

Картостроители демонстрируют тенденцию говорить в терминах концентрированного знания, в которое они прессуют свой опыт. Хотя картостроители часто используют различные внутренние представления области рассуждений, они эксперты во взаимном согласовании терминологии при обсуждении предмета между собой, и это один из способов картостроителей распознать друг друга. Это распознавание возникает вследствие последовательности взаимодействий, в которых один прослеживает маршрут на карте, останавливается и приглашает других выяснить, где же они расходятся. Цель этого упражнения — согласовать мысленные карты, но это также выявляет, прежде всего, наличие карт у других!

Картостроители склонны к частому изменению описаний и подходов, поскольку видят в этом способ упрощения, которое очень способствует пониманию, — ведь кто хозяин карты? В общественных или административных ситуациях это может привести к недоразумениям, поскольку картостроитель не догадывается, что у паковщика нет карты, которую он мог бы передавать по кусочкам. Для картостроителей паковщики выглядят упрямыми невежами, для паковщиков картостроители — большие путаники. В контексте программной инженерии этот коммуникационный барьер приводит к стычкам. Картостроитель хочет двигаться от массивного программного обеспечения к чему-то компактному, более надежному из-за структуры, которая необходима и достаточна. Паковщики не пытаются разобраться в этой новой структуре, а видят лишь маньяка, который стремится изменить каждый файл, встречающийся ему на пути.

Картостроители напрямую чувствуют эффективность своих размышлений и поэтому они в большинстве случаев чувствуют, что природа неведомым путем «играет честно» с ними, и даже награждает их чудесными сюрпризами, если они копнут достаточно глубоко. Это часто усиливает «спиритические» и «мистические» элементы их характера, даже в ситуациях, когда паковщики падают духом.

Прежде чем приступить к проблеме, картостроители убеждаются, что известные элементы этой проблемы содержатся в их сознании. Они берут на вооружение самые сильные черты своего характера, чтобы найти мотивы для выполнения тяжелой работы, заключающейся в исследовании происходящего в фоновом режиме. Чтобы найти решение проблемы, картостроитель напрягает все свои силы и получает в награду восторг, либо ощущение измены, если

что-то не получается так, как хотелось. Картостроители испытывают «страсть» к «скучным» предметам.

Картостроители превосходны за концептуально сложной работой, такой как решение сложных задач со многими взаимосвязанными элементами. Они могут проявлять требуемое задачами чутье или воображение, чего совершенно нельзя ожидать от паковщика. Высококачественная программная инженерия, математика и физика, с присущими им сложностью и уникальностью, — наиболее привлекательные для картостроителя научные дисциплины. Традиционно считающиеся искусством поэзия и музыка — области, где способности картостроителя манипулировать структурой являются существенным преимуществом, хотя может иметь смысл переопределить «Искусство» как то, что картостроители делают хорошо. Мощь великого искусства подвластна только мышлению картостроителя, поскольку художник использует массу звуков или цветов, самих по себе неинформативных, но затрагивающих глубокие пласты сознания. Концентрация на структуре может перенести эту структуру в сознание, и художник включается в карты зрителей!

Все эти различия — это просто следствия того, что у одного человека есть большая карта, построенная путем упорядоченного (дисциплинированного) обдумывания, а у другого ее нет. То, что эти различия между двумя группами людей существуют — главный сюрприз предлагаемого подхода. Это означает, что маловероятно взаимопонимание между людьми из разных категорий (картостроитель плохо понимает образ мыслей паковщика и наоборот).

❒ Паковка как самоподдерживающееся состояние

Мы живем в обществе, ориентированном на действие. Так пошло со времен изобретения сельского хозяйства и создания стабильной среды обитания, в которой выполняемые задачи определялись самой средой. Пропала потребность в интенсивном размышлении. У нас мало опыта обсуждения и управления субъективными, внутренними состояниями — хотя в субъективном имеется столько же общего (для разных людей) опыта, как и для внешних видимых всеми объектов. У нас есть общая эвристика, которая говорит, что следует ограничить наши наблюдения лишь видимым вовне, которое врывается к нам, чтобы предотвратить исследование субъективных явлений прежде, чем у них будет шанс получить результаты и скорректировать себя.

Когда все происходит не так, как нужно, мы ищем объяснения событию, и добываем более хорошие описания более эффективных действий. В ситуациях, где гибкость — ценное качество, это ведет к ослаблению желаний. Если все происходит в соответствии с процедурой, описанной на бумаге, считается, что все идет хорошо, а цена усилий просто не принимается во внимание.

Хуже того, поведение людей, лишенных понимания, может подстегивать друг друга. Если человек просто не понимает того, что происходит, он смотрит на других людей (по его мнению, явно знающих, что они делают) и чувствует себя уязвленным, поскольку считает отсутствие у него нужного пакета знаний своим просчетом. Они держат носы по ветру и лопочут о «соответствующих соображениях» и «подходящем действии», как будто «несоответствующие соображения» и «неподходящие действия» у них тоже есть, но не предполагают даже, каким же могло бы быть подходящее действие на самом деле.

Есть дело — все его делают! Так развивается эта стыдливая конспирация поддержания этикета хвастовства. Если кто-то пренебрегает этим этикетом, то он будет атакован внутренне невежественными возражениями и прочими методами давления, чтобы «соответствовать», очевидно ради этого. Это не может быть выражено ориентированными на действие терминами; только ссылками на случайные соотношения, полностью знать которые может только один человек. Картостроение в пакующем мире может быть мучительным и тягостным занятием, особенно если кто-то не понимает действительность, в которой живут паковщики.

В патологических ситуациях это может вести к постоянной деградации, когда решение любой проблемы пытаются спихнуть кому-то еще, на процедуру или механизм распределения

ответственности. Это напоминает попытку держать в одной руке зубную щетку и палочки для еды — если вы держите палочки в точности как на рисунке, то, что щетка упирается в нос и мажет пастой все вокруг, вас не должно беспокоить!

Напоминаем, мы описали причины этой мистики, не лопоча о «человеческом факторе» или «струнах души» наших коллег, а, определенно, социально обусловленным избеганием «фантазирования»!

❒ Коммуникационный барьер картостроитель/паковщик

Стоит повторить некоторые ключевые положения:

- Картостроение и паковка — очень разные стратегии.
- Паковка — это усердно насаждаемая общественная норма.
- Мир приспособлен для паковщиков.
- Язык бизнеса — это язык паковщиков.
- Результаты картостроения называют «здравым смыслом».
- Здравый смысл не так распространен.
- Картостроители считают паковщиков циниками или лентяями.
- Паковщики считают картостроителей иррациональными.
- Паковщики проводят большую часть своего времени, играя в политику.
- Последнее, что учитывают в политике, — разум.
- Картостроители часто заблуждаются насчет психологии паковщика.
- Паковщики обычно правильно понимают психологию паковщика.
- Картостроители часто заблуждаются насчет психологии картостроителя.
- Паковщики всегда неправильно понимают психологию картостроителя.
- Картостроители не имеют руководящей ими культуры.
- Большинство картостроителей учатся сами, как Маугли.
- Картостроители могут обучить себя сами!
- Картостроители могут научиться у других.
- Картостроители часто сталкиваются с вызовом со стороны общества.
- Картостроители в наше время редко реализуют свой потенциал.
- Если уж ситуация понята, она может быть изменена.

Глава 2

Мысли о программировании

❒ Для чего служит программная инженерия?

Всякий раз, когда мы запутываемся, мы должны оказаться способными увидеть, куда мы идем, чтобы знать, какое действие предпринять. Мы должны знать, чего мы пытаемся достичь.

Мы инженеры-программисты. Почему? Для чего служит программная инженерия? Что делают инженеры-программисты? Мы получаем самые курьезные ответы на этот вопрос. Один чудак сказал: «Они следуют процедурам Стандартов Программной Инженерии!» Другой добавил: «Они переформулируют *(transliterate)* требования!»

Боже. Мы предполагаем, что инженеры-программисты просто обеспечивают работу программ, которые нужны пользователям на их компьютерах. Это означает, что наши программы должны делать правильные вещи. Они должны быть устойчивыми. Иногда мы должны знать вполне определенно, что они устойчивы (надежны), а иногда нам требуется уметь это доказать. Нам всегда бы понравилось оказаться способными все это делать! Необходимые программы должны работать также и завтра, что обычно означает, что наши программы сегодня должны быть поддерживаемыми. Мы должны делать нашу работу эффективно по стоимости, иначе не получим шанс написать эти программы вообще. Доставка должна быть вовремя.

Мы используем всю нашу изобретательность и опыт, содержащийся в нашей дисциплине, чтобы добиться этих целей. Все наши технологии, стандарты, инструменты, языки предназначены помочь нам добиться этих целей.

Мы ничего не делаем ради этого.

❒ Программная инженерия — распределенное программирование

Традиционный взгляд на работу состоит в том, что команда выполняет работу, а отдельный человек вносит вклад в общие усилия. Но как картостроители мы можем попытаться посмотреть на вещи всеми возможными способами, чтобы проверить насколько они информативны. Мы можем обрисовать границу системы вокруг программирующей команды и заметить, что там нет ничего, что не смог бы сделать отдельный программист. Такие действия, как формулировка требований, проектирование, реализация, тестирование, управление, рецензирование, компилирование *(build)*, архивирование и управление конфигурацией, должны быть выполнены отдельным программистом даже для выполнения небольшой работы. Поэтому мы можем рассматривать деятельность в программной инженерии как распределение того, что один человек мог делать совершенно эффективно в «любительском» («непрофессиональном») режиме во время обучения!

Мы распределяем программирование по тем же причинам, по которым распределяем любой вид обработки: пригодность *(availability)*, параллелизм и специализация.

Такой взгляд приносит понимание. Мы должны аккуратно выделить различия между задачами. Иногда мы можем получать преимущества от выполнения двух задач одним человеком,

когда нас не должно волновать, что они объединены. Например, во многих организациях принята практика разделения идентификации требований и выбора архитектуры, но когда они переходят на технологию моделирования объектов в стиле Буча, то внимают совету и объединяют эти задачи. Когда мы разделяем навыки разработки и тестирования, мы можем извлечь из этого дополнительные преимущества, контролируя взаимодействие между стадиями таким образом, что мышлению инженера-тестера не угрожает мышление проектировщика. Был менеджер проекта, скорее всего паковщик. Он не имел ясного понимания того, что он делал и почему, а отсутствие какой-нибудь позитивной модели своей работы привело его к мысли, что ключевая цель состоит в предотвращении какого бы то ни было взаимодействия. Тестеры не должны были знать, как установить (создать) условия для компонентов, которые они должны были тестировать, а проектировщикам не дозволялось об этом говорить. Яростные споры продолжались днями. Это реально произошло тогда, когда мы потеряли ощущение большой картины.

Мы должны удостовериться, что взаимодействие между распределенными задачами эффективно, и это означает, что мы должны, помимо соответствия протоколу, держать в голове потребности друг друга. Все, что вам нужно держать в голове для выполнения своей задачи и передачи ее другому, также должны держать в голове ваши коллеги. Ваш результат не поможет никому, если он не говорит о том, что им нужно для выполнения следующего действия. Нам нужно использовать наши собственные способности выполнять работу друг друга, неважно, насколько неумело, чтобы контролировать собственную работу.

Наконец, мы должны понять, что в команде все еще существует черный ящик отдельного программиста. Поток информации — это не линейная последовательность преобразований, как на конвейере автозавода; для проектировщика это скорее расходящийся веер возможностей, сводящийся к единственному решению. Интуиция проектировщика пока еще не распределена. Такое достижение было бы самым значительным результатом искусственного интеллекта (ИИ).

❒ Что такое программирование?

Чтобы понять программную инженерию, мы должны понять программиста. Давайте позволим программисту определять требования (идентичные требованиям пользователя) и исследуем сценарий, который заканчивается созданием наипростейшей возможной программы.

> Ада сидит в комнате.
>
> Вечером в комнате становится темно.
>
> Ада включает свет.

Это фундаментальное действие программирования. Есть проблемная область (комната), которая динамична (становится темной). В динамике проблемной области есть порядок (темно будет до утра), который можно анализировать. Есть система, которая может функционировать в проблемной области (лампочка), и у этой системы есть семантика (состояние выключателя).

Есть желание (в комнате должно быть светло), и есть понимание (что воздействие на выключатель удовлетворит желание).

Динамические предметные области, системы и семантика детально где-то обсуждаются. Но здесь мы концентрируемся на лучшем осознании, что есть желание и что есть понимание.

Здесь стоит отметить, что мы подразумеваем под словом «программист». Робот, пишущий все ту же RPG 3 для распечатки счетов, все еще не делает никакого реального программирования вообще, но менеджер проекта, используя Excel для получения интуитивного понимания того, когда бюджет сократится и в чем главные причины, несомненно занимается реальным программированием.

❒ Программирование — игра картостроителя

Мы имеем разумное, имеющее смысл описание того, что на самом деле делают программисты. Два ключевых слова — «желание» и «понимание» — это вещи, которые трудно обсуждать осмысленно на бизнес-языке паковщика, концентрирующегося на «объективных» явлениях. Хотя это очень хорошая идея там, где это возможно, но она может тормозить прогресс, когда применяется как абсолютное правило (как паковщики часто и применяют правила).

Здесь стоит обратить внимание на философский аспект. Для того чтобы произошло взаимодействие, я должен ссылаться на то, что уже есть в твоей голове. Один из способов, чтобы вещь попала в твою голову, — попасть туда в виде образа чего-то из внешнего мира, а другой — быть частью твоего собственного опыта. Если часть твоего опыта уникальна (например, ассоциация между дымом трубки и вкусом рождественского пудинга, из-за визитов к родным), мы не можем говорить об этом без первоначального определения терминов. Даже после этого у меня нет опыта такой ассоциации, только представление о такой ассоциации. Но если часть твоего опыта разделяется всеми людьми (наша реакция на крик птенца альбатроса[1]), мы можем говорить об ее «объективности», как если бы реакцию на птенца можно было получить с самим птенцом, чтобы взвесить и измерить.

Необходимость ограничиться на работе «объективным» языком аргументируют тем, что это ограничение исходит из структуры организации работы[2]. Это просто глупо. Как работают журналисты, архитекторы (гражданского строительства) или даже судьи? Это область, где менеджеры вынуждены использовать свое понимание для уменьшения риска из-за ошибок.

Мы предполагаем, что реальный вывод отсюда — это то, что мы еще плохо умеем делать программы. Вероятно, мы никогда не научимся этому — наши аппетиты будут постоянно расти. Мы ограничены культурой и все больше подвержены влиянию тщательно продуманных объективных метрик, которые обычно используют наши коллеги из физики, а не из информационных дисциплин.

Чтобы достичь чего-либо в программировании, мы должны быть вольны обсуждать и улучшать субъективные факторы, а объективные метрики оставлять для отчетов об ошибках.

Первое, желание. В вышеприведенном примере Ада, вероятно, не начала с четкого желания увеличить освещенность. Ее среда становилась неоптимальной, вероятно, дискомфортной, и ей пришлось искать точное описание того, что же на самом деле она хочет. Прояснение желания — это обычно опыт, который допускает постепенное уточнение, и выполняется в тандеме с проектированием. Позднее мы более подробно остановимся на «Требованиях Пользователя» — сейчас же напомним, что уточнение желаний всегда содержит потенциальную возможность отправиться в исследование вместе с пользователем.

Следующее, понимание. Это момент распознавания, когда мы видим, что взаимодействие проблемы и желания может быть удовлетворено определенным использованием семантики. Это как сложение абстрактных векторов в бесконечном пространстве решений. Или, иначе говоря, это напоминает собирание мозаики, в которой можно изменять как форму кусочков, так и их расположение. Это сверхинтеллектуальное занятие.

Здесь есть паттерн, который соотносит программирование с любым другим требующим творчества занятием (искусством). У нас есть три явления: **Проблема**, **Семантика** и **Желание** (заглавные буквы напоминают о сущностях Платона). Проблема и Семантика не очень интересны для искусственного интеллекта (ИИ) или изучения сознания человека, а Желание — это вообще что-то странное. Эти три сущности выделены или соединены вместе из-за трех видов деятельности программиста. **Взгляд** заключается в изучении внутренних свойств Проблемы. Смотреть, чтобы понять значение Желания. **Описание** выявляет Семантику. Взгляд и Описание зависят от предметной области. Поэт может наблюдать за пассажирами, а эколог

[1] Наверное, очень противно кричит — С.К.

[2] Определяемой сводом положений — законами, правилами, инструкциями и т. д. — С.К.

за образцами популяций. Поэт выстраивает структуру из слов, а эколог описывает тщательно отобранный вид. Взгляд один и тот же у всех. Расскажите любому художнику о хороших моментах своей работы.

Чтобы с этим обращаться, нам нужны все эти прекрасные способности картостроителя.

Программирование — это игра картостроителя.

❒ Общие советы по картостроению

Паковщики обладают целой процедурализированной культурой, которая навязывает поведенческую колею просто для всего. Это настолько всеохватно, что вы даже не замечаете этого до тех пор, пока однажды не решите проблему очень эффективно, но способом, которого нет в списке. Это может быть нечто настолько же простое, как выйти из машины и купить билет «Плати и Показывай», перед тем как прокатиться по автостоянке и поставить авто на свободное место. Очевидно, некто «предполагает» припарковать машину, идет к автомату, возвращается обратно.

Картостроители с трудом представляют эти культурные соображения, но когда это действительно происходит, то может оказаться забавным. Паковщик давал званый обед и так оказалось, что более половины его гостей были картостроители, работники IT и другие. Хозяин достал стопку теплых тарелок из духовки и стал передавать их парню слева от себя. «Просто раздай их всем вокруг!» — сказал он бодро. И все было хорошо, пока он не передал последнюю тарелку. Затем его лицо выразило растерянность, потом веселье и в отдельный момент даже страх, пока он не догадался крикнуть: «Стоп!»

Или, может быть, он просто повеселился от души.

Картостроители не имеют общего культурного контекста, из которого можно добывать знания, поэтому мы почти все самообученные. Здесь мы собрали некоторые наблюдения, полученные в беседах с картостроителями. Разговаривая с другими картостроителями, мы можем узнать очень многое о картостроении.

Трясти надо

После того, как вы рассказали себе о том, что хотят в конце концов получить заказчики, походите вокруг элементов проблемы, рассмотрите, как они связаны, какие физические возможности имеются в системе, и проблема вдруг сожмется в нечто гораздо более простое. Из некоторых соображений, в этот неожиданный момент понимания мы редко получаем что-то совершенно правильное. Приготовьтесь покрутить это новое понимание и сделать побольше, развивая это понимание. Это хорошее время, чтобы рассказать о своем новом понимании коллегам и дать им возможность взглянуть свежим взглядом на вещи, на которые вы перестали смотреть, привыкнув к ним.

Изменения: последовательные или катастрофические?

Неожиданные реализации приходят тогда, когда они готовы, и мы можем оптимизировать условия для их получения. У них свои проблемы. Они веселят, убеждают, и иногда они ошибочны. Когда вы получаете их, проверьте их, обдумывая с точки зрения всего, что вам уже известно, и попытайтесь их расколоть. Большая встряска всегда полезна, даже если она не принесет решения непосредственно. С другой стороны, часто мы можем упростить проблему, просто разбив ее на куски и изучив их подробнее. Не смущайтесь думать «грубо» — начните это делать, и вы обнаружите нечто в следующий вторник. До тех пор, пока человек смотрит на вещи очень серьезно, он ничего не узнает.

Границы

Сфокусируйтесь на границах. Здесь имеется три класса составляющих вашей проблемы. Это вещи, о которых вы беспокоитесь, вещи, которые влияют на вещи, о которых вы беспокоитесь, и вещи, которые вас не волнуют. Одна из причин, почему жизнь картостроителей легче жизни паковщиков, в том, что они проявляют инициативу и пытаются идентифицировать все внешние проявления, которые дает им проблема, а не смотрят только на вещи, которые перечислены в бумажке, которую им всучили. Если вы в состоянии определить границы, то ваша проблема хорошо определена, и вы можете приступить к ее решению. Если это не получается, то, вероятно, нужно еще поговорить с заказчиком, либо обрисовать собственные границы, включающие сделанные предположения, которые должны быть явно проверяемыми.

Исследуйте перестановки

Когда у вас есть зеленая утка, фиолетовый лев и зеленый лев, спросите себя, где должно быть место фиолетовой утки. Осознание примитивных и невозможных перестановок может привести к лучшему общему пониманию, а некоторые перестановки полезны просто сами по себе.

Поработайте в обратном направлении

Мы все знаем, как найти выход из лабиринта, нарисованного в детской книжке, не правда ли!

Верчение тарелок

Вы знаете, когда ваша неосознанная способность к картостроению действует, из-за ощущения беспокойства, дискомфорта, даже досады. Когда это ощущение ослабляется, это сигнал вам. Если у вас назначено свидание — идите! Но если вы хотите добиться результатов, просто совершите короткое путешествие вокруг вашей проблемы и осмотрите ее с разных точек или направлений, и беспокойство возвращается. Это похоже на то, как жонглер возвращается к каждой тарелочке и подкручивает ее вновь до того, как она упадет с трости.

Расслабляйтесь

После тяжелой физической работы вы можете попытаться поднять что-то, но ничего не получается. Неожиданное бессилие там, где вы ожидали от себя способность применить силу, обескураживает. Аналогичное состояние мышления ощущается очень похоже. Нет абсолютно никакой точки опоры, но переключение в режим отдыха, вместо того чтобы продолжать пытать свои слабенькие маленькие нейроны — непростой процесс. Это происходит на автопилоте. Вы должны получить стимуляцию органов чувств. Душ, шумный бар, концерт. Смените обстановку. Вы сможете восстановить мыслительную энергию за несколько часов, если вы останавливаетесь, когда осознаете, что дальше продолжать не получится.

Рвите порочные круги

Ощущение беспокойства, возникающее из-за эффективного фонового размышления, отличается от ощущения переутомления (когда чувствуешь, что выдохся), которое иногда даже описывают как тошноту. Ваш мозг исчерпал все возможности, которые удалось найти, и вам требуется новая эмпирическая информация. Получите побольше данных. Поговорите с кем-нибудь. Очевидно, что у вас нет какой-то ключевой информации, либо ваша модель в целом перекосилась. Поэтому, может быть, вам нужно более тщательно исследовать проблему (пройтись частой сетью). Если это программа с ошибками, установите диагностику после каждой строчки и направьте вывод в файл. Затем прочитайте его тщательно за чашкой кофе. Да, это займет уйму времени — но что, у вас есть идея получше? Если это ужасная мешанина асинхронных событий,

которые нужно обработать, выпишите их на листочке вручную. Это усилит ваше внимание к последовательности событий, и у вас, вероятно, появится несколько новых вопросов прежде, чем вы дойдете до середины.

Сбрасывайте в «файл подкачки»

Есть виды глупости, к которым имеют доступ только картостроители. Картостроитель может быть парализован при попытках оптимизировать последовательность, которая слишком велика, чтобы поместиться в его голове. Вероятно, он хочет понести свадебный торт до того, как установит запасное колесо на машину, чтобы руки были чистыми, но запасное колесо здесь, а торт у Фреда. Когда такое случается с современными операционными системами со страничной организацией памяти, они освобождаются от ненужных страниц — переходят к стратегии подкачки страниц *(swap)*. Они просто сбрасывают в файл подкачки целые процессы, пока не освободится нужный объем памяти, а затем возвращаются к размещению страниц. Не дай себя парализовать — просто выполняй какую-то работу, а затем снова посмотри на проблему.

Упражнение с одеялом

Выверните пододеяльник, засуньте в него руки и захватите дальние углы изнутри. Затем возьмитесь за углы одеяла углами пододеяльника и потрясите. Немного практики, и вы сможете заправить одеяло в пододеяльник менее чем за 30 секунд.

❒ Картостроение и Процесс

Назначение программной инженерии — гарантировать, что программы, которые нужны нашим пользователям, работают на их компьютерах. Программная инженерия — это распределенное программирование. С этой позиции, мы можем определить процесс, как протокол для взаимодействия с нашими коллегами во времени и пространстве. Он обеспечивает структуру *(framework)*, которая говорит тем, кто идет за нами, где найти информацию о проектных решениях, нужную им для выполнения их работы. Изменяя процесс, мы передаем наш опыт в будущее. Он говорит нашим коллегам из другой части команды, когда мы встретимся, и предоставляет структуру для наших дискуссий. Он обеспечивает общие точки в наших проектах, в которых можем сравнивать подобное с подобным, и поэтому можем обсуждать аспекты нашего подхода, которые мы изменили.

Процесс — это не предварительно написанная метапрограмма для изготовления других программ. Хотя наша деятельность должна отображаться на процесс, самого по себе его недостаточно для изготовления программ. Мы думаем в рамках структуры процесса, но всегда должна быть стадия интерпретации определений процесса в свете заданной проблемы. Помните о необходимости интерпретировать определения — игнорирование этой деятельности просто приведет к выбору произвольной интерпретации. Тогда прекращаются попытки действовать методами, свойственными разработке, скажем, системы торговли фьючерсами, при решении проблем, возникающих при создании, например, системы рендеринга графики. Поэтому ты прекратишь споры о том, как будут удовлетворяться требования к трассировке журналирования транзакций, а вместо этого обеспокоишься дополнительными битами, которые понадобятся для зеркальных отражений!

❒ Ангелы, драконы и философский камень

Наши предки были не глупее нас, и, когда в четыре часа наступала темнота, им ничего не оставалось делать, как играть с мыслями в своих головах. Разгадывание таких античных головоломок, как мышление древних картостроителей, полезно не только потому, что оно интересно,

но и потому, что оно показывает нам, на что способен беспомощный человеческий интеллект. Это то, что нам нужно понять, если мы собираемся вернуть контроль над нашей работой из процессов, которым мы доверили свои жизни и карьеру.

Бесконечность была горячей темой, и наши предки разбили это понятие на три вида. Концептуальная бесконечность проста — ты просто говоришь «навсегда» и получаешь ее там, где это нужно. Еще есть потенциальная бесконечность. Ты можешь дать кому-нибудь указание типа «считай непрерывно». Теоретически в результате можно получить бесконечную последовательность чисел, но произойдет ли это на самом деле? Можешь ли ты на самом деле сделать так, чтобы перед тобой оказалось бесконечное число вещей, чтобы проделывать с ними удивительные фокусы? Они пришли к выводу, что если бы коллекция из бесконечного числа чего-нибудь, начиная с капусты и кончая королями, существовала на самом деле, то она заняла бы бесконечный объем, поэтому если бы существовала какая-нибудь бесконечная коллекция каких-нибудь вещей любого размера, то нас бы здесь попросту не было. Не было бы ничего, кроме кочанов капусты — повсюду. Мы здесь, поэтому бесконечной коллекции чего-нибудь любого конечного размера не существует — нигде. Но по-прежнему остается возможность бесконечной коллекции чего-нибудь бесконечно малого. Если что-то может быть бесконечно малым, то Бог (тот, кого удобно иметь под рукой в мысленных экспериментах, поскольку он способен сделать все, что может быть сделано в этом мире) смог бы заставить танцевать на кончике иглы бесконечное число ангелов.

Наши предки почувствовали, что эта идея смешна, и, таким образом, в этом мире не существует настоящей бесконечности. Сегодня у нас есть две великие физические теории. Одна работает при больших масштабах и использует гладкие кривые для описания пространства. Другая работает в микромире и использует ступеньки (квантование). Нам еще не удалось объединить эти теории вместе, поэтому мы не знаем, использует ли лежащая в основе их обеих более общая теория ступеньки для построения кривых (как фотография в газете), либо она использует кривые для построения ступенек (как ковер на лестнице). Это может быть нечто, что мы еще не можем представить, но если бы пришлось выбирать из этих двух, наши предки выбрали бы ступеньки, из-за ангелов на кончике булавки.

А как насчет драконов? Они ревут и изрыгают пламя. Их шумный полет быстрее ветра. Они добывают из-под земли драгоценные камни. Они живут в Южной Америке, Китае, Уэльсе. Они едят людей. Они червяки, а древний символ для мира — большой червяк. Они — это концептуальное ведро, в которое наши предки слили то, что мы называем тектоническими явлениями. У них не было мысли, что мир состоит из твердых плит, плавающих по жидкому ядру, но посредством картостроения они собрали вместе все эффекты, доступные для непосредственных наблюдений. Дракон занимал место реальных вещей в их мысленных картах до тех пор, пока они не открыли реальные явления, приводящие к тем эффектам, которые они обозначали «драконом».

А алхимия? Тщетный поиск процедуры для превращения основных металлов в золото и быстрого обогащения? Алхимический рецепт состоит из выполняемой оператором последовательности операций (которые могут иметь, а могут не иметь физической интерпретации в виде чертежа или описания эксперимента, или могут быть просто мысленным экспериментом). Рецепт заканчивается на том же месте, где и начинался, а по мере исполнения рецепта изменяется восприятие мира оператором. Самосознание оператора становится глубже и лучше, и трансформируется именно он, а не вещи на его столе. Возврат к началу необходим, поскольку только там он может увидеть — то, что прежде было искажено, теперь стало ясным. Алхимия — это картостроение.

В величественных соборах Европы много арок, поддерживающих своды. В наше время для выполнения расчетов методом конечных элементов мы наверняка использовали бы симметричный мультипроцессор, но у строителей не было компьютеров и алгоритмов. У них не было ни тех замечательных уравнений, которые у нас есть в механике, ни даже написанного Ньютоном на латыни текста. Большинство из них были неграмотны. Но если вы вычислите оптимальную

с точки зрения нагрузок/массы форму арки, то обнаружите, что они ее нашли. Они сделали это с помощью только тех инструментов, которые у них были — их собственным опытом и способностью получить ощущение чего-либо с помощью нейронной сети, расположенной между ушей.

Убедитесь, что у вас есть реалистичное представление о собственных способностях. Обычно требуется его подкорректировать! Достижение успехов в чем-то требует практики, но учитывая, что вы в любом случае будете выполнять работу, хорошо бы знать, чего можно достичь.

Творческий хак и ответственный инженерный труд ортогональны, а не несовместимы. Мы можем получить удовольствие, развивая наши способности до предела, и продолжать выполнять наши обязательства перед коллегами.

❒ Литературная критика и паттерны проектирования

Существует важное различие между намерением и действием. У писателя может быть намерение показать нам, как ужасен этот плохой парень, и он будет это делать, описывая сцены, содержащие отвратительные подробности. Наше намерение может состоять в сигнализации о том, что страница памяти в кэше более не доступна, наше действие состоит в установке флага «мусор» *(dirty flag)*.

Чтобы окончательно прояснить эту позицию, рассмотрим язык ассемблера. Код операции *(opcode)* может выполнять большинство специфических установок выходов процессора по заданным входным значениям, но мы рассматриваем код операции посредством его мнемоники, например DAA *(Десятичная Коррекция Аккумулятора — Decimal Adjust Accumulator)*. Даже когда есть взаимное соответствие между кодом операции и мнемоникой, более высокий уровень абстракции мнемоники может скрывать действие кода операции на аккумулятор, который просто преобразует биты в соответствии с алгоритмом. Если мы видим возможности обработки, предоставляемые кодом операции, можем ли мы «злоупотреблять» этим? Ответ зависит от обстоятельств.

Всякий раз, когда мы выясняем различие между намерением и действием, мы имеем возможность посмотреть на эффективность действия и задаться вопросом, что мы можем узнать о намерении или области намерения, исходя из структуры выбранного действия. Может быть, другое действие было бы лучше? Выявляют ли проблемы в действии проблемы в намерении? Когда мы проделываем это с книгами, то называем литературной критикой и беремся за дело всерьез. Если мы должны научиться лучше писать программы, то нам необходимо как можно лучше разобраться в некотором роде литературной критики, поскольку это единственный способ, который у нас есть, чтобы осознанно обсудить взаимодействие структуры и детали, которая характеризует стиль. По настоящему хорошо то, что, в отличие от литературной критики прозы, литературная критика программ черпает знания из экспериментальных данных, таких как протоколы ошибок. Это увеличивает удовольствие и отсекает болтовню, но оставляет возможность обучения.

Мы можем получить строгую и элегантную дисциплину из различия между намерением и действием. Рассмотрим следующий фрагмент:

```
// Search the list of available dealers and find those that
// handle the triggering stock. Send them notification of
// the event.

for (DealerIterator DI(DealersOnline); DI.more(); DI++)
    if (DI.CurrentDealer()->InPortfolio(TheEvent.GetStock()))
        DI.CurrentDealer()->HandleEvent(TheEvent);
```

Определения объектов содержат допускаемые намерением варианты использования, выраженные в сжатой форме. Однако реально нет более мелкого дробления, когда мы можем заключить намерение в комментарий, а действие в код, без того чтобы комментарии не стали глупыми.

Если мы перемежаем комментарии и код на этом естественном уровне дробления, мы можем гарантировать, что все строки в программе проинтерпретированы в комментариях. У нас есть стимул проектировать объекты (или функции), которые при таком способе мы можем использовать экономно. Мы находим, что гораздо легче исправить некоторую неэлегантность, чем объяснить ее кому-нибудь.

Осознавая различие между намерением и действием, мы можем сделать их оба экономными одновременно и удовлетворить цели детального псевдокода проектной документации и комментариев реализации, в то же время способствуя верификации реализации. Размещая все в одном месте, мы содействуем согласованности этих уровней.

Эта концепция развита далее в идее Дональда Кнута *(Donald Knuth)* о «грамотном программировании» *(Literate Programming)*, которое, чтобы его сделать хорошо, требует средств системной поддержки, вроде его Сетевой среды *(Web environment)* — прообраза WWW *(World Wide Web)*. Но вам не нужно скупать все гири, чтобы получить удовольствие от спорта. Грамотное программирование — это скорее отношение (позиция), а не инструмент.

На этом уровне литературной критики мы можем получить серьезные выгоды от изучения **паттернов** проектирования *(design patterns)*. Это компоненты архитектурной технологии, которая гораздо сложнее, чем обычный поток управления *(flow control)* с обработкой ошибок и другими типичными идиомами. Они чрезвычайно мощные и платформонезависимые. Почитайте прекрасную книгу Гаммы, Хелма, Джонсона и Влиссидеса *(Gamma, Helm, Johnson, Vlissides)*, в которой они описывают паттерн как нечто, что

«...описывает проблему, которая возникает вновь и вновь в нашей среде, и затем описывает основу решения этой проблемы таким образом, что вы можете использовать это решение миллион раз, не выполняя одни и те же действия дважды.»

Тема, которая лежит в основе обсуждаемых в этом разделе предметов — **Эстетическое Качество**. Мы все знаем о той беде, когда мы видим свою часто грозящую парализовать неспособность действовать на основании собственных ощущений, поскольку нет процедурного перевода для: «Это работает, но оно безобразно». Когда опытный профессионал чувствует эстетический дискомфорт и пытается об этом сказать, нам следует всегда обращать на это внимание. Наши стандарты красоты меняются от поколения к поколению, и по какой-то причине всегда соответствуют функции. В этом причина того, что делание кода красивым использует огромную базу знаний, которую мы не можем сознательно интегрировать, и ведет к эффективным по стоимости решениям. Если вещи красивы, то очень маловероятно, что они приведут к громадным затратам на поддержку. В этом состоит красота. Эстетическое качество — это, вероятно, единственный критерий, против которого можно честно спорить, утверждая, что использован неправильный язык. Попытка изобразить рассвет в стиле импрессионизма, но акриловыми красками, будет выглядеть ужасно, даже если пульверизатор работал прекрасно.

Мы должны смотреть на исходный код, который мы создаем, не как на конечный продукт более интересного процесса, а как на явление со своими собственными правилами. Он должен хорошо выглядеть, если его повесить на стену. Затраты на то, чтобы действительно взглянуть на наш код и изучить закономерности геометрических узоров черного и белого, узоры в синтаксисе и узоры в проблемной области, сами по себе не так уж велики, но иногда позволяют буквально увидеть ошибки с расстояния шести футов.

С точки зрения всей этой литературной критики, что можно сказать о религиозных войнах? Конечно, часто они происходят развлечения ради, и мы не хотим препятствовать удовольствию от нелепых особенностей любимых инструментов и технологий наших друзей! Но иногда разумные программисты впадают в изматывающие и снижающие производительность перебранки, которые просто идут по кругу. Мы забываем, что строго между собой мы можем использовать

структурные аргументы. Когда возникает нежелательная религиозная война, задайте следующие вопросы:

1. Какова глобальная позиция, включающая оба специальных случая?
2. Имеется ли различие намерений между позициями?
3. В чем состоит общая цель?

Например, вы оцениваете возможности мощной интегрированной среды. Вы используете *Emacs* на работе и доработали его, чтобы он управлял вашей кофеваркой. Я работаю на многих машинах, и, как это ни эксцентрично, я знаю, что на них всегда есть *vi*. Мы устанавливаем *Emacs* на новых машинах и обучаем *vi* новичков. Ваш LISP, конечно, sucks.

Это соотнесение возможностей и целей часто дает великолепное примирение мнений у квалифицированных людей. Возможность прийти к соглашению о наилучших идиомах, чтобы сделать работу в хорошо понимаемой среде, не означает, что все должны изменить свое мнение — они просто соглашаются. В противоположность популярному мнению, часто это правильный ответ. Разговаривая с квалифицированным человеком об идиомах новой среды, можно научиться очень многому очень быстро, тогда как использование идиом из старой среды в новой приведет к постоянным стычкам.

❒ Атомы познания

В любой задаче, требующей понимания, мы всегда будем находить по крайней мере один «**атом познания**». Атом познания — это часть проблемы, которая может быть **адекватно** рассмотрена только при условии загрузки ее элементов, черт, знаков и т. п. в сознание отдельного картостроителя и получения наилучшего возможного результата. Слово «адекватно» здесь очень важно — существует целый букет проблем, которые, если бы имелись неограниченные ресурсы, могли быть разрешены играючи, но нужно очень хорошо подумать, чтобы решить их в реальных условиях. Например, любая толпа идиотов могла бы справиться с задачей смены декораций, которая возникает во время большого музыкального шоу, если на это дано несколько недель. Но сделать то же самое за время, пока конферансье что-то меланхолично бубнит в свете единственного прожектора, требует гениальности.

Опытные планировщики проектов обнаружили, что распознавание и управление атомами познания в рамках проекта — решающий начальный шаг. Сначала мы должны распознать атомы познания. Существует взаимосвязь между архитектурой системы и атомами познания, которые она содержит — архитектор должен применить интуицию и опыт для выявления решаемых, но еще не решенных проблем. Проблемы, которые, как надеется архитектор, могут быть решены при разработке, будут влиять на дизайн, поскольку никто не хочет создавать архитектуру, которую нельзя реализовать!

Поэтому архитектор может очертить границы атомов познания вокруг проблемы. Например, в системах добычи знаний *(data mining system)*, практические комбинаторные проблемы могут быть сконцентрированы в базе данных, либо на более высоком уровне прикладной логики. Правильная идентификация атомов познания будет управлять как архитектурой, так и рабочими пакетами, порученными отдельным членам команды. Каждый атом должен быть передан одному человеку или подгруппе для решения, но они могут обнаружить, что работают над более чем одной частью системы, чтобы разрешить свою проблему. Поэтому части должны быть хорошо разбиты на уровни, так, чтобы модули не сталкивались в бестолковых сражениях. Идентификация атомов обычно требует учета баланса времени, пространства, связи, риска, возможностей команды, переносимости, времени разработки, и все это должно быть проделано при наличии атомов, разрешимость которых неочевидна. Поэтому архитектор должен суметь

увидеть ключевую проблему и выразить, по крайней мере, в своей собственной голове природу условий компромиссов. Вполне возможно распознать набор очевидных компромиссов, о котором очень трудно рассказать другому, не обладающему, как картостроитель, способностью видеть структуру. Превращение мысленной модели в последовательность (действий) всегда тяжело, поскольку мы не думаем на языке технических бумаг, которые загружаем по ftp.

Во время идентификации атомов познания очень важно избежать специфических заблуждений, которые повторяются раз за разом. Часто возможно раздробить атом на более мелкие части не сильно задумываясь, и таким образом достигнуть этапа кодирования без больших усилий. Но когда дело доходит до реализации, все ввергается в хаос. Реальные проблемы никуда не деваются, они просто оборачиваются уродливыми API подсистем, проблемами производительности, ненадежностью и т. д. Границы атомов познания сжимаются все сильнее и сильнее, до... Хоп! Они вновь возникают на уровне всей системы! Идеология упрощающего пошагового уточнения без регулярной сверки с действительностью и попыток найти логические ошибки в проекте ответственна за великое множество трагедий, включая потерю большей части отведенного на проект времени на попытки выполнять текущую проектную работу с чистосердечной неформальной желчностью, за которой следуют отчаянные попытки залатать дыры в программе.

Определение границ атома познания может происходить циклически, а квалифицированный архитектор укажет для них правильное место, где верхний уровень, где нижний, а что посередине. На начальных стадиях это может быть огромный, единый атом познания, который нужно передать в руки ответственного работника и сказать: «Попробуй разобрать эту мешанину, пожалуйста!»

По определению мы не знаем, каков наилучший подход к атому познания. Если бы мы знали, то он не был бы атомом. Из этого следует, что это не может планироваться на основе диаграмм планирования проектов (диаграмм Ганта) в терминах подцелей. Это должно быть одной задачей, а о длительности можно только догадываться. Опытные картостроители поднаторели в догадках, но они не могут объяснить, почему проблема тянет на два дня, неделю, полгода. Поэтому у того, кто дал наилучший прогноз, очень мало аргументов в его защиту. Боязнь последующих объяснений — важный фактор, который часто отбивает у картостроителей охоту проявлять свои интуитивные способности и выдавать необходимые для планирования проекта цифры.

В результате расщепления атома познания работник обычно может выложить очень детальный набор описаний задач (целей), основанный на твердом понимании того, что должно быть сделано. Таким образом, во многих проектах следует поправить диаграммы Ганта, добавив туда расщепление атомов познания. Мы предполагаем, что большая часть проектов, пытающихся распланировать по Ганту все по дням, демонстрирует всеобъемлющую линейную модель производства. Программисты, работающие по таким диаграммам Ганта, не могут получить выгод из разумного управления атомами познания. Вместо того, чтобы повернуть свой разум к решаемой проблеме, они будут доказывать, что они хорошие работники, под «прессом», как будто, унижая, их можно заставить думать более ясно. Это грозит стрессом и снижает производительность.

❒ Плато качества

Когда принята стратегия формирования мысленной карты проблемной области и попыток упростить ее, обычно сталкиваются с проблемой определения момента окончания работы над картой. Эта проблема возникает на каждом уровне проектирования. Сверхъестественно, но почти всегда есть глубокое решение, которое значительно проще всех остальных и, очевидно, минимальное. (Есть много способов это выразить, но потом этот вывод станет очевидным.) Хо-

тя проповеди типа: «Ты узнаешь это, когда увидишь!» — несомненная истина, но они не говорят, куда посмотреть.

Единственный честный аргумент, который мы можем здесь предложить — это обещание, что это действительно произойдет. И хотя это ничего не доказывает, все что мы можем сделать — это показать работающий пример приведения в минимальное состояние. Но это работает — спросите любого, кто пытался.

В качестве примера возьмем код из прекрасной книги Джеффри Рихтера *(Jeffrey Richter's* Advanced Windows*)*. Эта книга — полезное чтение для любого, кто пытается писать программы для Win32 API *(Application Programming Interface)* (поскольку иначе у вас не появится мысленная карта семантики системы).

Рихтер очень четко раскладывает по полочкам вопросы использования Win32, но даже в его примерах (и, в частности, как результат соглашений, которым он следует) появляется сложность, которую мы попробуем убрать. На странице 319 имеется функция `SecondThread()`. Мы просто посмотрим на эту функцию, опустив остальную программу и некоторые глобальные определения:

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm) {
  BOOL fDone = FALSE;
  DWORD dw;

  while (!fDone) {
    // Wait forever for the mutex to become signaled.
    dw = WaitForSingleObject(g_hMutex, INFINITE);

    if (dw == WAIT_OBJECT_0) {
      // Mutex became signalled.
      if (g_nIndex >= MAX_TIMES) {
        fDone = TRUE;
      } else {
        g_nIndex++;
        g_dwTimes[g_nIndex - 1] = GetTickCount();
      }

      // Release the mutex.
      ReleaseMutex(g_hMutex);
    } else {
      // The mutex was abandoned.
      break;// Exit the while loop.
    }
  }

  return(0);
}
```

Для начала просто упростим стиль скобок, уберем пробел между ключевым словом и открывающей скобкой, а также многословный комментарий к `ReleaseMutex`. Мы в курсе, что идет религиозная война между последователями Кернигана и Ритчи *(K&R)* и последователями Вирта *(Wirth)* по поводу стиля скобок, но симметрия обрамления блока действительно позволяет лучше увидеть некоторые вещи. Дополнительная строка, которая при этом появляется, даст выигрыш чуть позднее — следуйте за нами!

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
```

```
{
    BOOL fDone = FALSE;
    DWORD dw;

    while(!fDone)
    {
        // Wait forever for the mutex to become signaled.
        dw = WaitForSingleObject(g_hMutex, INFINITE);

        if(dw == WAIT_OBJECT_0)
        {
            // Mutex became signalled.
            if(g_nIndex >= MAX_TIMES)
            {
                fDone = TRUE;
            }
            else
            {
                g_nIndex++;
                g_dwTimes[g_nIndex - 1] = GetTickCount();
            }

            ReleaseMutex(g_hMutex);
        }
        else
        {
            // The mutex was abandoned.
            break;// Exit the while loop.
        }
    }

    return(0);
}
```

Очень легко можно избавиться от одной локальной переменной: *dw* присваивают значение, а в следующей операции тестируют. Инвертирование смысла проверки помогает локализовать ссылку (проверка, затем изменение `g_nIndex`). А пока мы здесь, нет смысла инкрементировать `g_nIndex` просто для того, чтобы вычесть 1 из текущего значения в следующей операции! Мы уже использовали постфиксную форму оператора инкремента языка Си, который как раз для этого и предназначен.

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;

    while(!fDone)
    {
        // Wait forever for the mutex to become signaled.
        if (WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
        {
            // Mutex became signalled.
            if(g_nIndex < MAX_TIMES)
```

```
            {
                g_dwTimes[g_nIndex++] = GetTickCount();
            }
            else
            {
                fDone = TRUE;
            }

            ReleaseMutex(g_hMutex);
        }
        else
        {
            // The mutex was abandoned.
            break;// Exit the while loop.
        }
    }

    return(0);
}
```

Прерывание цикла (`break`) зависит только от результата `WaitForSingleObject`, поэтому естественно переместить проверку в управляющее выражение, избавляясь от прерывания цикла и одного уровня вложенности:

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
{
    BOOL fDone = FALSE;

    while(!fDone && WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
    {
        // Mutex became signalled.
        if(g_nIndex < MAX_TIMES)
        {
            g_dwTimes[g_nIndex++] = GetTickCount();
        }
        else
        {
            fDone = TRUE;
        }

        ReleaseMutex(g_hMutex);
    }

    return(0);
}
```

Теперь просто сожмем... Мы знаем — многие стандарты кодирования говорят, что мы всегда должны ставить фигурные скобки, поскольку иногда у глупых людей получается нечитаемая мешанина, но посмотрите, что получается, когда мы пренебрегаем этим правилом и концентрируемся на повышении читаемости кода.

```
DWORD WINAPI SecondThread (LPVOID lpwThreadParm)
```

```
{
    BOOL fDone = FALSE;

    while(!fDone && WaitForSingleObject(g_hMutex, INFINITE)==WAIT_OBJECT_0)
    {
        if(g_nIndex < MAX_TIMES)
            g_dwTimes[g_nIndex++] = GetTickCount();
        else
            fDone = TRUE;
        ReleaseMutex(g_hMutex);
    }
    return(0);
}
```

Теперь немного настоящей ереси. Черт возьми, в момент когда мы покончим с этой полной безответственностью, результат окажется совершенно неочевидным. (Здравый смысл поможет сделать лучше, чем правила.)

Ересь в том, что если мы знаем, для чего наши переменные, то мы знаем их типы. Если мы не знаем, для чего предназначена переменная, знание ее типа мало поможет. В любом случае, компилятор все равно сделает проверку типов. Поэтому избавимся от венгерской записи, а заодно и от переопределений типов, которые просто определены (`#define`), но не для нас. Сокрытие разыменования используя `typedef` — другое бесцельное упражнение, поскольку хотя и позволяет выполнить некоторую инкапсуляцию валюты, этого совершенно недостаточно, чтобы избавиться от беспокойства по этому поводу, поэтому аккуратные программисты вынуждены держать настоящие типы в голове. Поддержка концепции дальних указателей в именах переменных для 32-битного API с плоской адресацией — тоже довольно глупое занятие.

```
DWORD SecondThread (void *ThreadParm)
{
    BOOL done = FALSE;

    while(!done && WaitForSingleObject(Mutex, INFINITE)==WAIT_OBJECT_0)
    {
        if(Index < MAX_TIMES)
            Times[Index++] = GetTickCount();
        else
            done = TRUE;
        ReleaseMutex(Mutex);
    }
    return(0);
}
```

Теперь смотрите. Мы достигнем Плато Качества...

```
DWORD SecondThread (void *ThreadParm)
{
    while(Index < MAX_TIMES &&
          WaitForSingleObject(Mutex, INFINITE)==WAIT_OBJECT_0)
    {
        if(Index < MAX_TIMES)
            Times[Index++] = GetTickCount();
        ReleaseMutex(Mutex);
```

```
    }
    return(0);
}
```

Одиннадцать строк против 26. На один уровень меньшая вложенность, но структура полностью прозрачна. Две локальных переменных исчезли. Нет блоков. Совсем нет вложенных `else`. Меньше мест, где могут скрываться ошибки.

(Если вы еще не программировали используя потоки *(threads)*, то повторная проверка значения `Index` внутри тела цикла кажется грубой и ненужной. Если же *программировали*, то повторная проверка естественна и очевидна. Это очень важно: пока текущий поток приостановлен в `WaitForSingleObject`, другой поток скорее всего будет активен и изменит значение. То, что для вас очевидно, зависит от вашего опыта: еще одна мораль из этого примера — рассмотрения только структуры куска кода недостаточно.)

Наконец, текст делает совершенно ясным, что разные потоки выполняют функции в разных контекстах. Поэтому совершенно не нужно определять функцию с именем `FirstThread()`, в точности такую же, как `SecondThread()`, и вызывать их так:

```
hThreads[0] = CreateThread(..., FirstThread, ...);
hThreads[1] = CreateThread(..., SecondThread, ...);
```

Когда можно просто:

```
hThreads[0] = CreateThread(..., TheThread, ...);
hThreads[1] = CreateThread(..., TheThread, ...);
```

Почти треть этого примера получена клонированием! Если мы обнаружим ошибку в одной реализации, нам нужно будет не забыть исправить аналогичные ошибки везде. Зачем беспокоиться, когда можно просто слить их в одну. Это хороший способ, когда поджимают сроки.

❒ Знание, а не число строк кода *(KLOCS)*

Программисты дороги. Результаты их работы должны быть собраны и использованы к выгоде их организации. Проблема в том, что традиционный способ паковщика подбить результаты — подсчитать то, что они могут увидеть. Результаты изучения проблемы командой программистов, пришедшей к пониманию и проверившей это понимание, получив надежный код, выражаются не в числе строк кода *(KLOCS)*, которые они набрали во время изучения. Результаты состоят в окончательном понимании, к которому они пришли, когда закончили.

Причина, по которой важно оценивать результат таким образом, в том, что понимание показывает гораздо более простой способ реализации, чем тот, с которого команда начинала. Классическое поле сражения картостроителей с паковщиками в программировании состоит в том, что картостроители видят, что с тем, что они узнали, повторная реализация может быть сделана быстрее и не будет страдать от проблем сопровождения, вырисовывающихся в существующем коде. Паковщики видят, что картостроители безумствуют, пытаясь разгромить всю их работу (как будто нет резервной копии) и повторить работу нескольких последних месяцев, которые были ужасны, поскольку они, очевидно, не знали, что они делали (они хранят изменяющиеся вещи). Паковщики настраивают одного из своих защитников остановить картостроителей, и организация вынуждена забыть о достигнутом понимании, которое не может быть использовано в контексте существующего кода.

Разумная организация хочет максимального понимания и минимального размера кода, которого только можно достигнуть. Организация, увязшая в модели производства разбухающего кода не учитывает понимание, а подсчитывает свои активы в растущих грудах кода.

❒ Хорошая композиция и экспоненциальный рост продуктивности

Определение **хорошей композиции**, которое часто используют в художественных школах, звучит так: «Если убрать или изменить любой элемент, то изменится и целое». Вероятно, это морской пейзаж с маяком, задающим с одной стороны сильную вертикаль, приковывающую взгляд и позиционирующуюся относительно волн позади. Ситуация с маяком (и волнами) — то, что мы распознаем, и тут живопись показывает свою мощь. Если бы вместо величественного маяка стояло приземистое здание, картина говорила бы о чем-то другом. Если бы поверхность воды была гладкой или там резвились купальщики, то картина несла бы еще какие-то сообщения.

Дело в том, что вокруг не должно быть ничего, что не имело бы ясно означенной цели по отношению к другим элементам композиции. Художнику нужно сохранять контроль за сообщением, и если картина содержит случайные предметы, то они будут вызывать в уме зрителя непредсказуемые ассоциации и искажать отношения между важными элементами.

Логики при проверке наборов аксиом сталкиваются с той же самой проблемой. Для этого у них есть гораздо более точный термин, но он происходит просто из компактных формальных структур, в рамках которых делаются наблюдения и доказываются теоремы. Они говорят, что набор аксиом должен быть **«необходимым и достаточным»**. Необходимый и достаточный набор позволяет ясно увидеть «природу» рассматриваемого «мира». Это позволяет удостовериться, что обнаруженные следствия — истинные следствия исследуемой области, а не какие-то произвольные предположения.

Ни в одной из этих сфер деятельности не нужно напоминать людям о важности сохранения вещей настолько малыми, насколько это возможно, в противоположность программированию. К сожалению, практическая полезность нашего искусства означает, что люди часто стремятся как можно быстрее увидеть новую функциональность, которую мы пытаемся создать. Будучи создана, эта функциональность становится частью фона, и каждый из нас, от корпораций до отдельных людей, становится заложником наших собственных унаследованных систем.

И хотя это может выглядеть, как вечная безысходность (неизбежность) Закона Программиста, здесь появляются люди, разрывающие порочный круг деградации, и мы опишем, как это делать, с этой точки зрения на программирование как на творческий процесс.

Фундаментальная трудность в сохранении контроля над унаследованными структурами, будь то артефакты стратегии доставки потребителю, возникшие из спецификаций с фиксированной стоимостью амортизации, либо древняя система индексирования CODASYL, которую требуют воссоздать в объектной базе данных — это время. Иногда это выражается как «стоимость», но время редко имеет цену. Это крайние сроки *(deadlines)*. Нет другого способа избежать крайних сроков, кроме как крикнуть «Волк!». Это реалии коммерции, которыми мы не управляем. Все в порядке — мы просто думаем об этом реалистично и скорее учитываем это в своей работе, а не используем для оправдания плохого качества продуктов.

Первая точка приложения усилий против крайних сроков — осознание того, что работа спорится в чистой среде без странных флагов у функций, без противоречивых соглашений о вызовах, без многочисленных соглашений об именовании и прочего барахла. Дни после чистки весомее дней до нее. Поэтому сделайте чистку в самом начале, когда каждый может видеть перед собой большой проект, это время окупится позднее. Вы почти всегда должны делать чистку — код, который большинство организаций помещают в репозитарии, обычно первый кандидат. И не важно, что он прошел все этапы тестирования. Делайте чистку, проверку на старость и даже не обсуждайте сроки, пока не увидите порядка.

Следует сделать предупреждение — будьте реалистами насчет того, сколько времени займет чистка. Чем запутаннее клубок, тем больший эффект даст чистка, но тем больше риск, что у вас не будет времени на его распутывание и выполнение работы. Часто полезно задать вопрос: «Насколько сложно поведение этого предмета как черного ящика?» Если ответ: «Не очень!»,

то вы знаете, что по мере последовательного вычесывания сложности, он ужмется до чего-то простого, даже если вам вообще не видно, куда идти.

Вторая точка приложения усилий исходит из экспоненциального уменьшения сложности программы. Если у вас более ясный алгоритм, минимальная реализация будет проще. Чем меньше у вас кода, тем легче увидеть структуру кода, и уменьшается шанс появления искажающих концепцию ошибок. В то же время меньшее количество кода означает и меньше возможностей для синтаксических ошибок, описок в именах переменных и т. д. Меньше ошибок — меньше затраты времени, меньше затраты времени — меньше тестирования. Это не займет много времени в любой команде из более полудюжины человек для большей части их работы, чтобы опуститься до нанесения увечий взаимного перепатчения, когда доступ к репозитарию станет узким местом. Позволить пропустить вещи через процесс на более поздних стадиях — заложить бомбу с часовым механизмом, которая рванет тогда, когда уже поздно что-то делать. С другой стороны, безумие отказа посреди такой ситуации может восстановить спокойствие в оставшиеся дни.

Третья точка приложения усилий — **«дела скунса»** *(skunkworks)*, название пошло от названия местечка Skunkworks, основанного корпорацией Lockheed Martin вдали от корпоративного центра, «потому что скунс».[3] Эта страшная технология может быть использована исключительно инициативными командами по секрету на рождественской вечеринке, либо спровоцирована просвещенным руководством. Как повелось во всей этой работе, мы проясним, почему «дела скунса» срабатывают.

В такой деятельности индустриальной эры, как строительство, имеются физические объекты (кирпичи), с которыми трудно управляться. Вместо того чтобы нагромоздить кирпичи стопкой, чтобы увидеть, сколько их понадобится для строительства дома, мы подсчитываем их. Эта абстракция от физического к информационному дает нам сверхъестественные способности в обращении с кирпичами. В конце концов у нас становится так много чисел, говорящих о поставках, транспортировке и потребностях, что мы вынуждены организовать наши числа в структуры, чтобы работать с ними. Мы используем электронные таблицы, а абстракция от информационного к концептуальному вновь дает нам сверхъестественные способности.

Выполняя такую работу над информацией, как программирование, мы не начинаем с физического и не получаем явное преимущество, когда переходим к информационному. Мы начинаем с информационных требований, списков по пунктам и т. п., и нам приходится обрабатывать их с помощью информационных инструментов. Нам приходится делать это из хороших соображений, таких, как информационные контракты с заказчиками, информационные соглашения на собраниях с включенными в наш процесс коллегами. Иногда также мы делаем это из таких плохих соображений, как слишком подробные инструкции информационных технологий по преобразованию кирпичей в область информации — например, измерение производительности с помощью строк кода *(KLOCS)*.

Беда в том, что при нормальной работе у нас нет рычагов. Информационное содержимое нескольких минут совещания может быть больше информационного содержимого требований, которые на нем обсуждаются! Как деятельность, выполняемая людьми, собрания не способствуют достижению цели! Мы выигрываем только потому, что мы можем продать наше новое знание много раз, или потому, что в соединении с другими знаниями оно дает существенно больший вклад в процесс.

Это лишает возможности использовать понимание для увеличения власти над информацией. «Дела скунса» иногда выглядят как отказ от процесса в интересах творчества. Ничто не может быть дальше от истины. Необходима высокая пропорция опытных людей, чтобы исключить этот эффект, поскольку, чтобы что-то завершить, они должны положить в основу высокоинформативные динамические индивидуальные процессы. То, от чего отказываются — это понимание, содержащееся в изнурительном процессе, ради понимания, которым владеют

[3]Насколько я слышал, скунс — это такая вонючка, что даже кошек выворачивает — С.К.

опытные люди. Отсюда проистекает предусловие для «дел скунса». Отказываясь от детализованного процесса, принимают, что риск неизбежен, а потеря личной безопасности компенсируется простыми хорошо определенными целями. Каждый должен осознать, что «дела скунса» могут потерпеть неудачу, надежды могут не оправдать ожиданий, а могут быть соображения, по которым все вернется к традиционным методам управления. Но когда они срабатывают, то срабатывают великолепно!

Все успешные начинания — «дела скунса». Поэтому есть неудачные начинания. Усилия «дел скунса» могут сменить большой риск потери управляемости[4] на малый риск «пионерства»[5]. В таких ситуациях, это может быть эффективным средством управления риском.

[4]Из-за разбухания проекта — С.К.

[5]В смысле, быть первым — С.К.

Глава 3

Программист за работой

❒ Подходы, методологии, языки

Когда мы рассматривали простейшую программу, мы увидели, что необходимо найти взаимосвязь (карту) между проблемной областью и семантикой системы, которая удовлетворит желание. Очевидно, чем меньше возможный набор взаимосвязей, тем легче найти нужную, если, конечно, она существует. Любая заданная проблемная область будет обладать своей собственной присущей ей сложностью, а каждая проблема внутри нее будет иметь свои уникальные сложности. Когда мы рассматриваем проблему, она — вещь в себе. Мы редко можем изменить ее определение, чтобы управлять ее сложностью (хотя иногда это и можно и нужно делать, и поэтому это **хорошо**). Поэтому в поисках точки приложения усилий, наиболее эффективного способа сделать работу, все чем мы можем поиграть — это семантика системы.

На одном конце этого спектра расположены продукты COTS (Commertial Off The Shelf — коммерческие, взятые с полки). Загрузи его, запусти его, работа сделана. На другом — набор команд процессора, который позволяет нам выжать из железа все, на что оно физически способно. Между этими крайностями располагаются различные промежуточные семантики, которые упрощают поиск взаимосвязей, ограничивая семантику.

В этих терминах, язык — это произвольный набор *(ящик инструментов — kitbag)* семантик. Си — язык, но также и Excel — язык, как и средства создания интерфейса пользователя *(GUI builders)*. Эти наборы лежат, но не дают никакого намека, как использовать их содержимое. Языки специализируют по проблемным областям, чтобы достигнуть большой вероятности получения более простых взаимосвязей для любой проблемы из выбранной области. Чтобы решить, какую из двух семантик (языков) выбрать, обычно нужно спросить, какая из них требует более простых взаимосвязей (более простой программы), чтобы выполнить работу. За исключением самых тривиальных случаев, это требует знакомства с обоими наборами семантик на практике.

И хотя мы можем получить ясное понимание того, что из себя представляет язык, методология гораздо труднее поддается рассмотрению в таких терминах. Мы предполагаем, что причина состоит в том, что идея «методологии», как ее обычно рассматривают, включает неявные предположения, что это процедурный подход к решению программных проблем, а мы знаем, что это не так. То, что мы можем описать вместо этого, немного иное — **подход**.

Подход заключается в совете, даваемом одним опытным картостроителем другому, о том, как лучше всего решать проблему какого-то типа. Это приглашение посмотреть на мир определенным образом, даже если это выражено как процедурный рецепт. Предписание «Получи диаграмму *(Data Flow Diagram)* изменения начислений по неделям» в книге «Как построить систему печати платежной ведомости» *(How To Build A Payroll System)* на самом деле выглядит так «Ограничь рассмотрение до систем с понедельным начислением, и выведи перечень этих начислений».

Это совет для создателя систем печати платежной ведомости, при условии, что все ситуации можно свести к понедельным начислениям. Как и язык, подход получается тем проще, чем больше он специализирован на заданной предметной области. Кроме того, как и язык, подход трудно выбрать правильно без понимания «цен» имеющихся подходов и проблемы. При большом числе умных разработчиков, каждый год пишущих коммерческие *(COTS)* продукты, которые автоматизируют подход в виде высоко специализированного языка, с которым сможет работать любой дурак, в будущем наиболее вероятным приложением сил для хороших программистов станет изучение более глубоких, перспективных подходов и выработка новых подходов перед лицом новых проблем. Но, вероятно, всегда будут толпы людей, использующих один и тот же подход для ритуального производства еще одной точно такой же бухгалтерской системы для еще одного клиента. Они могут постоянно «переобучаться новым методологиям». Но они есть и останутся клерками, а разрыв в производительности и отдаче между клерками и программистами будет расширяться. Вот что означает быть игроком в эру информации.

Имеются языки, которые специализируются на определенном подходе. Smalltalk требует от пользователя рассматривать мир в виде объектов. Lisp требует применения неудобоваримых манипуляций с лямбда-исчислением, которые приводят к появлению «собаки еды» *(dog of food)* вместо «накормленной собаки».

Следует подчеркнуть, что языки объективны, подходы объективны, но методологии — плод нашего коллективного воображения и не существуют. Пренебрежение этим положением и неудачный выбор подхода может привести к ситуациям, в которых критические составляющие проблемы не учтены, поскольку оказывается, что подход их не рассматривает, в то время как тем, кто пытается разобраться в проблемах, вставляют палки в колеса коллеги, которым кажется, что те действуют «непрофессионально», поскольку не «применяют методологию». Это пример барьера между картостроителями и паковщиками.

Интересные методологии состоят частично из подхода, частично из языка. Структурный дизайн Джексона *(Jackson Structured Design — JSD)* ограничивает свою область применения проблемами с ясно идентифицируемыми чертами (свойствами) и поэтому в состоянии предложить очень детальную инструкцию, как решать проблемы такого типа. Держите глаза открытыми, и JSD хорошо послужит в своей области. Однако за пределами этой области он может вызвать проблемы, поскольку если задачи не имеют нужных Джексону черт, то бесконечное латание дыр *(kludging)* будет делать хорошую систему из плохого понимания. Это не ошибка Джексона, поскольку он никогда не говорил, что JSD — это ритуализованная панацея для решения всех компьютерных проблем.

На выходе JSD мы видим нечто совершенно неожиданное, артефакт того времени. Джексон описывает, как перейти от его диаграмм к коду, делая это вручную! Он осознает, что он делает, и поясняет, что автоматизм этой работы позволяет нарушить правила структурного программирования и использовать `goto`. Сегодня он не стал бы этого делать — он бы просто направил диаграммы в генератор кода, как это многие и делают. Дело в том, что нотацию диаграмм JSD лучше всего рассматривать как язык программирования! Джексон создал язык, который был ориентирован на некий подход к проблемной области.

То же самое справедливо для подходов и языков Буча *(Booch)*, Румбаха *(Rumbaugh)* и UML *(Unified Modelling Language)*. В действительности, для каждой интересной методологии. В ранних публикациях Буча и Румбаха они не пропускали диаграммы через генераторы кода, но показали, что трансляция большинства диаграмм в большой степени механична. Не переживайте слишком сильно о реализации этих методов вручную — в целом это не сложно!

Создание языка и подхода, более или менее специализированного к какой-то предметной области — крупное достижение. Поступая таким образом, авторы должны долгое время продумывать, как лучше всего управляться с проблемами, разделять их, исследовать их, смотреть на них с разных точек зрения и в соответствии с этим разрабатывать свой подход или язык. Но многое оказывается запутанным из-за пресловутого языкового барьера картостроитель/паковщик, и приходится отказываться от акцента на творческое мышление, необходи-

мое для построения карты между проблемой и ее языком. Вместо того, чтобы представлять свой подход как структуру и предлагать некоторые эвристики для рассмотрения проблемы в терминах этой структуры, они вынуждены использовать процедурный язык и описывать действия, которые необходимо предпринять, императивно[1]. Если кто-то не обладает даром думать творчески, *то есть* конструировать мысленную карту своей проблемы посредством размышления и затем исследовать ее, то им не остается выбора, кроме как следовать этим (вероятно, некорректным) указаниям, а их результаты будут существенно зависеть от везения. Джексон тут хорош. Он специфически ограничивает свою область и говорит читателю, какие свойства искать. Читатель начинает с обследования проблемы в поисках (ключевых) признаков. Буч включает интересный раздел, посвященный поиску объектов, которого, если бы только он был выполнен глубоко и широко, достаточно, чтобы сделать путь императивов ненужным, поскольку он опирается на сугубо картостроительные моменты. Наконец, то, как книга Страуструпа *(Stroustrup)* описывает объектный подход и язык C++ — праздник стиля, интуиции, структуры, глубины и творчества. Это тяжелая книга, описывающая сложный язык программирования, но она написана великим картостроителем в действии, у которого нет внутренней неразберихи относительно предмета.

❒ Как писать документы

С точки зрения многих инженеров-программистов, большая часть их жизни состоит в написании документов. С точки зрения настоящей работы, мы предпочли бы сказать, что их жизнь состоит в выполнении работы по повышению понимания, которое будет донесено до их коллег в соответствии с протоколом, определенным в их процессе. Следовательно, мы подразумеваем, что работа — это всегда понимание, а процесс говорит, какое понимание нам нужно передать коллегам. Это определяет приемлемый для каждого документа язык. Данные соображения помогут сформировать описание реальной работы, которую требуется выполнить при работе над каждым из создаваемых инженером документов, как-то: Требования Пользователя, Требования к Программному Обеспечению, Архитектурный Проект, Детальный Проект и Спецификация Тестирования *(User Requirement, Software Requirement, Architectural Design, Detailed Design and Test Specification)*.

Следует отметить еще два момента. Во-первых, работа не заключается в создании кучи невразумительных бумажек, которые никто даже не будет читать, но которые по виду напоминают «конструкторскую документацию». Те, кто выплескивают технические подробности (все слэши и десятичные точки) в основном тексте, вместо того, чтобы поместить их в приложении, если уж это так нужно, просто издеваются над читателями и дискредитируют наше искусство в целом. Используя простой, нормальный язык (включая где необходимо специальную терминологию, но не злоупотребляя ею) расскажите читателю то, что ему нужно знать.

Второй момент касается формата. На любой стадии процесса программирования люди используют понимание, чтобы находить и предлагать паттерны. Если бы они знали, что они собираются найти, у них не было бы работы, поскольку их мог бы заменить настроенный кем-нибудь коммерческий *(COTS)* продукт. Раз мы не знаем наверняка, что работнику нужно представить, то как мы можем сказать ему, как он это должен представить? Стандартные форматы в процессах не должны восприниматься как догмы. Все приличные процессы в ISO 9001 содержат стандарты размещения необходимых разделов. Используй их соответственно, и если во время написания возникает структура документа, можно сделать вставку в План Управления Проектом, чтобы описать выбранный формат. Вот и вся суть ISO 9001.

[1] Строгим командным голосом — С.К.

Требования пользователя

Недавно возник большой интерес к «Реинженирингу бизнес-процессов» *(«Business Process Re-Engineering» — BPR)*. Это практика изучения бизнес-процессов для определения, можно ли их улучшить, и часто это необходимо сделать просто потому, что со временем изменилась природа бизнеса организации. Иногда превозносят точку зрения, что программная инженерия всегда включает значительный компонент BPR, поскольку в противном случае потребитель обнаружит, что компьютерная система автоматизирует бизнес-процесс, не учитывающий появление компонентов, которые руководство реализовало в ответ на потребности бизнеса, и систему ожидает крах.

Таким образом, первая обязанность инженера-программиста — помочь потребителю понять природу их собственных требований. В примере простейшей программы — это кристаллизация желания из общего ощущения дискомфорта в специфическую потребность в большей освещенности. Инженер-программист руководствуется в этой работе дисциплиной, накладываемой необходимостью писать программу для компьютера. В работающем коде невозможно скрыть неоднозначности, как это можно сделать в текстовом отчете. Полезные ТП, таким образом, составляют настолько ясное понимание потребностей пользователя, насколько это возможно в начале проекта, как это понято пользователем и инженером, на языке пользователя. ТП обязательно потребуют уточнения в дальнейшем, по мере того, как дисциплина программирования выявит неоднозначности, независимо от того, будут ли эти поправки включены в документ или нет.

Причина, которая вызывает большую путаницу — двоякое назначение ТП. С точки зрения инженера ТП должны быть живым документом, но из соображений коммерческих и правовых он занимает место эталонного документа на все время работы над проектом. Эти две цели совершенно различны. Когда они смешиваются, мы получаем комедию инженеров, не обремененных юридическими знаниями (что на самом деле так), пытающихся написать «Декларацию независимости», в то время как критические моменты бизнес-процессов остаются неисследованными.

Иногда единственный способ выйти из этого положения — написать два документа. Один определяет контрактный минимум и, если придерживаться некоторых методик, может быть с успехом написан полностью пользователем. Другой — это живой, внутренний документ, который говорит нам, что же «приведет пользователя в восторг». Это то, чем мы стараемся руководствоваться в его интересах. Как удовлетворить пользователя, если единственной подсказкой, как это сделать, является нечто, служащее нашим коллегам из коммерческого подразделения на поле закона? Рамки, до которых потребитель видит «настоящие ТП», зависят от коммерческих соображений.

Будьте очень осторожны по отношению к «средствам интегрированного отслеживания свойств», призванных получить ваши ТП и пропустить их положения через проектирование и тестирование, превращая в код. Такие средства часто забывают, что требования можно удовлетворить, **не делая** что-то, что ряд требований может быть реализован в нескольких сегментах кода, без прямого соответствия между требованиями и сегментами, и что очень трудно протестировать важные общие требования отдельными тестовыми инструментами. Это не значит, что такими инструментами не следует пользоваться — для задач конфигурирования и ввода данных они подходят прекрасно. Некоторые даже могут отследить свойства отдельных групп классов для GUI. Но для обычных требований черного ящика на «уровне пользователя» они искажают то, что могло быть написано в ТП, или навязать стиль разработки, требующий утомительного ручного кодирования отдельных свойств, вместо использования абстракций везде, где это возможно.

Требования к программному обеспечению

Если ТП описывают требуемую систему на языке пользователя, то ТПО описывают ее на языке инженера. Поэтому в этом документе сначала могут оказаться расчеты размера системы. Что особенно характерно для современных объектных методологий, потребность в ТПО уменьшилась, поскольку архитектура будет состоять из прикладных классов, у которых ясные взаимосвязи с языком ТП. В этой ситуации ТПО и АП могут быть объединены.

Говорят, скульптор думает о своей законченной работе как о заключенной внутри глыбы камня или куска дерева, которую нужно оттуда извлечь, отсекая лишнее. Это помогает. Точно так же, мы можем представить себя глядящими глазами пользователя на наше творение в один из дней в будущем. Если мы смотрим на него, используя свойства (черты) системы, мы можем спросить себя: «Как это должно быть реализовано?» Тогда очень легко получить описание потребностей пользователя на языке инженера-программиста.

Архитектурный проект

Если уже приходится делать работу, заключенную в АП, то может показаться, что самая тяжелая работа по проектированию к этому моменту завершена. Существует также большой соблазн вообще уклониться от написания Архитектурного Проекта. В то время как мы умышленно опускаем детали проекта в АП, иногда чтобы удовлетворить требованиям независимости от платформы *(portability)*, иногда просто чтобы развеять туман вокруг большой картины, мы по-прежнему должны быть уверены в том, что наш проект действительно реализуем. Инженер должен знать по крайней мере один приемлемый способ реализации каждого свойства до того, как искать его, и должен подумать о концептуальной целостности кода, требуемого для реализации этих свойств.

Утверждение, что архитектурный проект не должен касаться детального проекта, мы считаем ошибочным. Если мы не можем рассматривать реализацию, мы не можем быть хорошими инженерами, поскольку любой идиот может спроектировать нереализуемое. Только учитывая реализацию мы обнаруживаем ограничения наших проектов и находим разницу между хорошим и плохим. Мы способны увидеть альтернативы, сравнивать их и выбирать лучшее. Если мы не можем учитывать реалии реализации, то один дизайн так же хорош, как и другой, и эта критическая стадия познания становится упражнением в письме, кто быстрее может «написать документ», нимало не задумываясь над написанным!

АП — это дидактический документ. Он учит читателя тому, как посмотреть на проблему и решение так же, как смотрел автор.

Детальный проект

ДП — это записка в бутылке. Он говорит читателю о том, как автор планировал реализацию, поэтому код можно понять. Детальность изложения должна прояснить те места, которые остались за кадром в АП, и привести читателя в точку, где уже должен быть сам код. Иногда, это объяснение может быть выражено псевдокодом, но не обязательно. Следует допускать возможность исправления ДП. Во время реализации будут возникать такие детали проекта, как организация кода в модули. Если такие детали не передать с помощью ДП нашим коллегам, то как еще это сделать? Это простое упущение вызывает в дальнейшем слишком много ненужных проблем, поскольку инженеры посчитают составляющие системы хорошо документированными только в случае, если они знали, с чего начать! Окончательный вариант ДП должен говорить последователям, что они должны знать, чтобы понять систему и изменить ее.

План тестирования

План тестирования — наиболее чувствительный к контексту тип документа, но очень полезно руководствоваться следующими наблюдениями в рамках требований ситуации. Стратегия тестирования имеет целью напрячь систему. Не будет никакой пользы, если делать тестирование хаотично, поэтому необходимо найти одну или несколько моделей системы, которые могут дать нам индикатор для типичных и стрессовых ситуаций. Таким образом, полезная структура — описать модель, выяснить стрессовые условия и затем перечислить их.

❒ Ход конем («Вилка»)

Снова и снова в этой работе мы видим эхо глубокой структуры, которую мы выявили, написав простейшую программу. У нас есть проблемная область, семантика системы и карта взаимосвязей между ними, созданная программистом при рассмотрении желания. Этот паттерн — центральное действие программирования компьютеров. В нем самом может и не содержаться понимания, но способность это делать — единственное доказательство, что проблема на самом деле понята в терминах заданной семантики. Если семантика строгая и проверяемая, как семантика цифрового компьютера, можно заявить о «глубоком» или «истинном» понимании, но это только предположение, поскольку кто-нибудь всегда может заглянуть за горизонт и сказать: «Посмотри на это так!»

Этот паттерн настолько важен, что мы хотим сфокусировать на нем внимание. Хотя мы стараемся избегать замысловатого жаргона без стоящего за ним реального смысла, мы хотим ввести термин **«Ход конем»** («Вилка»), чтобы обозначить этот паттерн. Мы позаимствовали этот термин из шахмат. Там конь стоит на доске и может совершать последовательность Г-образных перемещений. Другие фигуры могут перемещаться только по горизонтали, вертикали или диагонали, а Г-образные ходы коня позволяют напасть сразу на две фигуры, каждая из которых ограничена своим собственным миром, и таким образом добиться чего-нибудь полезного в любом случае.

Такого вида паттерн появляется снова и снова, но мы всегда можем свести его к написанию простейшей программы. Компьютерная система может находиться во многих состояниях и развиваться в соответствии со своей собственной внутренней логикой. Окружающий мир, за которым следит компьютер, также может находиться во многих состояниях и изменяться (эволюционировать). Используя интуицию, дизайнер может абстрагировать и внести в компьютер критические аспекты проблемы, применяя одну и ту же структуру в обоих случаях, поэтому компьютер и реальность будут согласованы. Тестовые ситуации, формируемые моделью проблемы и системы, будут охватывать допустимые (и, вероятно, недопустимые) состояния пространства входных воздействий, в соответствии с интуицией автора, таким образом, что в любом случае можно будет проверить изменение состояния системы. Дизайнер, при необходимости выполнения манипуляций с данными, будет использовать свойства самих данных, определяемые структурой данных, и отражать эти свойства на свойства языка, как в каноническом:

```
while((c = getchar()) != EOF)
    putchar(f(c));
```

Все проектирование архитектуры заключается в прощупывании проблемы, рассматривая требования с максимально возможного числа направлений, до тех пор, пока в ней не обнаружится структура, которую архитектор системы сможет использовать для решения проблемы.

«Ход конем» всегда использует внутреннюю (присущую ей) глубокую структуру проблемной области. Проверка того, что эта глубокая структура реальна, а не плод воображения и случайных совпадений, очень важна. Если дизайнер использует случайные совпадения, результат будет скорее «заумным», чем «элегантным», и все будет хрупким, полагающимся на специальные меры предосторожности по всему результирующему коду системы, с потерей целостности

проекта. Вейнберг *(Weinberg)* приводит пример программиста, пишущего на ассемблере. Тот обнаружил, что мог бы делать поиск по таблице,[2] основываясь на номере кода операции и спроектировал свою программу исходя из этого. Но разработчики аппаратуры («железа») не считали схему нумерации кодов операций священной, и когда они произвели допустимые изменения, весь дизайн программы развалился.

❒ Персональный послойный процесс

Дзенская притча говорит о мудром монахе, пришедшем к старому учителю. Он вошел в комнату учителя и сел перед ним. «Когда ты входил, с какой стороны двери ты оставил свой посох?» — спросил учитель. Монах не знал. «В таком случае, ты потерял свой Дзен».

После того как вы рассмотрели структуру своей программы и готовы реализовать ее, все еще остается важная задача сохранить над ней контроль. Даже если вы уже написали критические строки кода, еще нужно написать много других. Требующаяся для этого дисциплина гораздо важнее, чем любой формальный процесс, и в каждой новой ситуации его нужно применять разумно.

Ваш процесс будет разбивать задачу на подзадачи, а затем вы должны собрать все вместе. Как машина-укладчик железнодорожного полотна, вы должны выстраивать структуру своей работы по мере ее развития. По прошествии времени вы достигнете способности делать это в своей голове, и на самом деле очень быстро, поскольку задачу можно упростить двумя способами.

Вы можете разворачивать только часть вашего плана, над которым вы работаете. Вот как можно спланировать изменения некоторой программы в своей голове:

1. Идентифицировать все файлы, которые включают функции:

   `ModelOpen()`,

   `ModelRead()`,

   `ModelWrite()`,

   `ModelClose()`.

2. Отключить контроль версий этих файлов.

3. Внести изменения.

   3.1. Изменить `modread.c`

      3.1.1. Исправить `ModelOpen()`

      3.1.2. Исправить `ModelRead()`

      3.1.3. Исправить `ModelWrite()`

      3.1.4. Исправить `ModelClose()`

   3.2. Изменить `appfile1.c`

   3.3. Изменить `appfile2.c`

4. Обратно включить контроль версий

5. Перекомпилировать

[2] Table lookups — например, массив адресов перехода или массив указателей на функции, когда выбор осуществляется по целому индексу — С.К.

Тот факт, что это описание процесса может не описывать каждый маленький шажок и поэтому не перегружает ваш интеллект бесполезными попытками это делать, не освобождает вас от обязанности делать эту работу самому. И это позволяет устанавливать необходимый порядок в собственной голове по своему усмотрению — так, как вам удобнее. Некоторым людям нравится записывать небольшие списки файлов, которые нужно модифицировать, на клочках бумаги и рвать их, когда дело сделано, оставляя остальной процесс в своих головах. Они могут помнить, где они находятся на большой картине, но если их прервать посредине большого списка, они могут растеряться.

Другой важный метод — вы можете изменять свои планы. Ключевая концепция TQM заключается в том, что мы должны понимать, чего мы хотим достигнуть, если уж мы хотим знать, когда мы этого достигнем. Это значит, что нам нужно оказаться в состоянии сказать честно, что, как мы думаем, мы делаем в любой момент времени, но это не остановит нас, если мы передумаем! Например, мы могли бы добавить в вышеприведенный пример:

3.1.5. Разобраться со всеми заголовочными файлами :-(

...в любой момент, поскольку мы изменяем определения функций и наши бедные маленькие умишки блуждают туда-сюда и выясняют, что прототипы тоже потребуют корректировки.

Нам не нужно помнить, в какое мусорное ведро мы выбросили стаканчик из-под выпитого утром кофе, чтобы иметь полное понимание, где мы находимся в нашей организованной работе. Вместо этого мы можем проникнуться духом TQM и организовать самих себя с полным осознанием того, что мы делаем. Если мы это сделаем, проявятся все обычные преимущества от обдумывания того, что мы делаем. Мы сможем увидеть возможности автоматизации скучного набора текста с помощью скриптов и макросов, а в рамках Персонального Послойного Процесса *(PLP)* мы всегда можем задать вопрос: «Как бы я мог отменить это действие?»; это то, что отличает от людей, которые случайно стирают свой код (удалив по ошибке файл) и которым приходится два часа ждать администратора, чтобы восстановить этот файл со сделанной ночью резервной копии.

И, в качестве последнего комментария этого раздела, скажем, что в условиях профессиональной инженерной среды часто нам нужно использовать PLP для управления сложностью даже простейшей работы. Ритуализация PLP может стать гипнотической. Чтобы выдержать пропорции, всегда спрашивайте себя, не хватит ли 30 секунд, чтобы завершить эту работу, и если ее можно просто сделать, не теряйте время на выполнение ритуалов. Всегда делайте резервную копию!

❒ Увидеть мир в строчке кода

Мы описали центральную проблему проектирования программного обеспечения как нахождение оптимальной карты взаимосвязей между проблемой и семантикой системы. Мы также обсуждали деятельность, которую обычно называют «написание документов», как выполнение необходимой работы и отражение ее результатов в документе. Итак, что входит в выполнение работы, но не показывается в документе? Это нечто связано с поиском оптимальной карты взаимосвязей.

Правда в том, что еще никто, взявшись за работу, не сел и не выписал наилучшее решение иначе, чем не задав некоторый набор проверочных вопросов. Разработчик эффективного решения всегда посмотрит на проблему с нескольких различных направлений и обычно сможет увидеть несколько вариаций возможных решений. Решения должны быть выверены, чтобы убедиться, что они удовлетворяют всем требованиям и могут быть реализованы на практике. Только победитель будет описан в документе. К сожалению, обычная практика — опускать в документе детали того, почему документированное решение было выбрано из других альтернатив.

Эта позиция очень важна, когда наш доминирующий подход, который обычно обеспечивает базовую структуру нашего процесса, включает разработку сверху вниз. Идея разработки сверху вниз — это то, что позволяет нам увидеть лес за деревьями. На ранних стадиях мы можем увидеть общую цель системы. Мы можем затем правильно сконцентрироваться на получении деталей внутри каждой подсистемы, зная, что общее направление выбрано правильно. Эта позиция отличается от подхода, когда разработка сверху вниз выбирается для того, чтобы не зависеть от деталей реализации нижних уровней, хотя оба мотива часто принимаются во внимание вместе.

В обоих случаях замысел должен быть реализован, поэтому разработчик должен убедить себя в том, что проект действительно реализуем. Если цель — увидеть лес за деревьями, то вероятно появится идея о выборе целевого языка, операционной системы, решении проблем управления командой. Критерий успешной разработки — это обычно оптимизация использования системных ресурсов. Если цель — независимость, то критерий — выполнять разработку, реализуемую на всех возможных целевых платформах. В идеале, этого можно достичь, используя модель, общую для всех целевых платформ.

Это означает, что разработчик должен учитывать в процессе разработки реализацию, хотя обычная практика — упускать требования реализации, что приводит к тому, что разработчик предпочитает одно решение другому.

Размышляя о проекте, совершенно естественно, что разработчики видят у себя в голове высокоуровневое описание внешних частей системы, вероятно I/O, более детальное описание внутренних частей, вероятно группу определений таблиц базы данных, а прямо посередине, в точке, где выполняется основная работа системы, они часто знают лишь критические участки кода, которые могут быть очень сложными. С этой позиции они могут убедить себя, что детали внешней части системы в порядке, тем не менее, даже не продумав их. Не всегда при проектировании в центре внимания оказывается факт, что существуют ошибки при передаче (сбойные биты) — разработчик мог бы заметить критическую часть протокола низкоуровневого восстановления ошибок и осознать необходимость продумывания его реализации. Нет лучшего способа ощутить безопасность того, что ты разрабатываешь, чем найти хотя бы один практический способ его реализации.

Мы не говорим, что вы обязаны видеть, как в процессе проектирования проскакивают в голове строки. Мы говорим о том, что это может быть очень полезным способом прояснения ваших размышлений о предмете, а если ваши мысли поворачиваются к коду, следуйте за ними. Не отметайте эти размышления только потому, что ваша задача — высокоуровневый документ. На этом пути вы получите проектный документ, который можно эффективно использовать, а люди будут называть вас волшебником процесса разработки. Помните, каково держать зубную щетку и палочки для еды? Люди, которые имеют такую привычку, скорее поверят, что вы хорошо владеете палочками для еды, чем то, что вы просто зажали зубную щетку в кулаке.

Другая область, где на этапе высокоуровнего проектирования действительно очень полезны небольшие фрагменты кода — получение реального ощущения семантики системы, которую вы собираетесь использовать. Мы всегда должны изучить новые API для операционных систем, GUI, библиотек и т. д. Потребуются годы, чтобы действительно полностью освоить правильное использование API. Поэтому заглядывайте в книги, которые обсуждают API, и пишите небольшие программки, демонстрирующие свойства, которые, как вы думаете, вам понадобятся. Это на самом деле помогает сконцентрировать ваш ум на том, что вам нужно, чтобы проследить путь снизу вверх, в то время, пока ваша разработка происходит сверху вниз, и гарантирует, что вы не пытаетесь использовать семантику, которой на самом деле нет. Попытка разработать проект, требующий другого дизайна операционной системы, может оказаться очень сложной, но если вы провели несколько минут за написанием маленькой программки с целью поупражняться с какой-то особенностью, то вы будете использовать ее как есть, и никогда не вспомните, что говорится в документации. Вы отыграете эти минуты во время реализации, поскольку вы можете просто скопировать ваши заготовки в исходный код и немного их поправить.

Посвятите некоторое время просмотру дизайна используемых вами API. Взгляните на их валюту — значения, передаваемые из/в API. Как части этого интерфейса стыкуются друг с другом? Хорошо ли они спроектированы? Какие идиомы предлагают вам использовать их разработчики? API обычно проектируют опытные разработчики, и они похожи на небольшие послания от очень ярких людей о том, как они видят мир. Стиль UNIX API Кена Томпсона *(Ken Thompson)* прекрасно сохранился за 30 лет. Он сам говорил, что единственное изменение, которое он бы сделал, это вот какое: «Я бы написал `creat()` с `e`!» В UNIX API есть что-то очень близкое к тому, как работают компьютеры.

Этот раздел целиком посвящен тому, насколько важно уметь смотреть на один уровень ниже, чем тот, на котором работаешь. Это справедливо несмотря даже на то, что сокрытие деталей реализации — постоянная цель нашей работы. Чем лучше мы этого добиваемся, тем больше мы выигрываем, но мы еще не настолько умелы, чтобы забывать о нижних уровнях. Понимание того, где компилятор резервирует кучу и стек, позволяет нам обнаружить грубые ошибки, когда мы нарушаем модель языка. Знание имеющегося в нашем распоряжении объема физической памяти (и файла подкачки) позволяет нам писать программы, которые будут работать в ситуациях реального мира. Даже истинно виртуальные машины, такие как виртуальная Java машина, предоставляют сервисы настолько низкого уровня, что мы можем полагаться на то, что их создатели реализуют их разумно, чтобы мы могли предсказать эффективность наших операций.

❒ Концептуальная целостность

В «Мифическом человеко-месяце» Фредерик Брукс[3] подчеркивает важность концептуальной целостности проекта. Наш глубокий взгляд на программирование предполагает некоторые практические пути достижения концептуальной целостности.

Во-первых, мы знаем о важности мысленных карт. Если каждый член команды разделяет внутренне согласованную мысленную карту создаваемой системы, то возможный вклад каждого будет соответствовать духу всей разработки. Если не разделяет, то не будет, поскольку руководство по стилю, достаточно подробное, чтобы позволить кому-либо сделать все правильно без знания того, что они делают, окажется гораздо сложнее в написании, чем сама система.

Во-вторых, у нас есть картина того, как программист оптимизировал последовательность проектных решений для получения минимального решения и управления сложностью. Поэтому нам нужно посмотреть на виды конструкций, из которых строятся программы, и убедиться, что они общедоступны. Такое руководство по стилю проекта указывает на согласованный набор соглашений об именовании переменных, стратегий обработки ошибок, примеров идиом использования API подсистем и даже стиль комментариев. Можно сказать, что, управляя формой кирпичей, архитектор может задать форму здания, оставляя в то же время гибкость в руках конструктора. Структура такого кода, таким образом, гарантирует, что код канонических примеров предсказуем и элегантен. Поэтому примеры кода в руководстве по стилю управляют структурой, а структура управляет кодом. Здесь мы видим еще одно эхо «Хода конем» — если мы используем правильные структуры, то мы можем ввести в игру необходимый и достаточный синтаксис и написать минимум текста. И наоборот, чем более запутанные берутся приемы, тем больше их нужно, чтобы переплести их вместе.

И, наконец, последнее преимущество концептуальной целостности, ценимое профессиональными программистами — очень практичное. Представьте, что вы в работе. Вы увидели способ распределить функциональность, вы нашли элегантные методы обхода всех тех способов, какими ОС может сигнализировать об ошибке, и уже наполовину сделали работу по кодированию, и вам понадобилось имя для новой переменной. В вашей голове застопорилось от перегрузки тривиальностью! Экспоненциальная выгода от сосредоточенности внимания и удержании этого

[3] *The Mythical Man Month* by Fred Brooks

состояния так же велика, как экспоненциальная выгода от минимизации размера кода, поэтому стоит оградить себя от всяких глупых раздражителей, отвлекающих внимание. В местах, где каждый прекращает работу через каждые десять минут для объяснений с руководством, преимущества настоящей концентрации внимания никогда не проявятся, но там, где внешние обстоятельства тщательно отсеиваются, наличие руководства по стилю позволяет часто делать подобные штучки[4] на лету и может существенно увеличить эффективность работы.

❒ Управление настроением

У паковщиков есть правила ведения дискуссий, включающие подсчет очков каждого и демонстрирующие на словах и на деле полную незаинтересованность в конечном результате. Картостроители также имеют правила ведения дискуссий, но они несколько другие.

Картостроителям разрешается подпрыгивать и много кричать. Но это не означает, что они собираются убить друг друга, это означает их причастность. Вероятно, они прервутся, только сходив вместе на ланч, продолжив свои выкрики по возвращении.

У каждого будет свой собственный способ говорить о свойствах проблемы, и понадобится согласовать общий жаргон проекта. Сделайте эти согласования по совместной мысленной модели и нацельте группу на созидание и оспаривание части достижений группы, а не на воздвижение стен вокруг замков из песка. Ненавидьте грех, а не грешника!

Если коллега говорит что-то, что вы не понимаете или кажется вам парадоксальным или нелепым, спросите себя, а что если человек пытается рассказать о части карты, которую вы видите перед собой, с совершенно другой точки зрения? Узнайте, что это значит на языке, который вас устраивает. Начните с предположения, что у него имеется кое-что интересное в голове, и попытайтесь выяснить, что это такое. Этот стиль дискуссий, о котором много думали апологеты **зететики** *(Zetetics)*, вышедшие из Общества по Исследованию Паранормальных Проявлений *(Society for the Investigation of Claims of the Paranormal — SICOP)*, чтобы исследовать, какими могли бы быть правила доказательства существования паранормальных явлений.

Просто быть группой картостроителей с совместно используемой мысленной моделью совершенно недостаточно, чтобы начать вместе ее изменять. Как и во всех остальных случаях, мы должны явно осознать правильность того, что мы пытаемся сделать. На других этапах проекта команде потребуются другие размышления. Иногда вам захочется собрать вместе все трудности и усложнить модель. В другой раз стратегическими станут организация и упрощение. Иногда вам захочется описать свои потребности, а в другой раз вам нужно будет решить, как объяснить модель заказчику (пользователю).

Если в дискуссии члены команды имеют разные цели, то мало чего можно достичь. Никто не сможет сконструировать разумное описание технических моментов, если его прерывают люди, которые думают, что цель заключается в максимизации приемлемости для пользователя *(maximising customer acceptability)*.

Это не означает, что все собрания без ясно определенной цели с неизбежностью вырождаются в переход на личности — это происходит только, когда произвольно выбранные цели взаимно исключают друг друга. Но даже дискуссии со многими целями могут быть прояснены, если сначала явно сформулировать, в чем цели состоят. И ни в коем случае внимание группы не должно удерживаться с ритуальной одержимостью паковщиков, поскольку идея заключается в прояснении дискуссии, а не в избегании ее. Как всегда, мы должны служить цели, а не микрополиции процедуры. Если член команды видит, что что-то уводит от темы (и засоряет весь вид), он должен сказать об этом. И наоборот, если он видит обстоятельство, которое нужно обсудить, но оно не настолько критично, он может записать его на клочке бумаги и заявить о нем как о теме для обсуждения на следующих собраниях.

[4]Например, выбор имен для переменных — С.К.

Управление настроением также распространяется на весь цикл проекта. Идентифицируя специфические настроения и их изменения, лидер команды может обеспечить структуру для деятельности команды и избежать ситуаций, когда каждый день каждый приходит на работу и занимается каким-то кодированием, без какого бы ни было ясного понимания того, как должен проходить нормальный день.

За рамками проекта настроение организации в целом также может влиять на проект. Главная угроза может исходить от того, как организация смотрит на взаимодействие (общение) внутри себя. Некоторые организации установили высокоритуализированные границы между группами, что приводит к значительным затратам времени на управление внутри организации. Когда велика сила устаревших административных процедур, приводящих к росту сложности и, следовательно, уменьшающих эффективность, существует лишь немного сил, которые могут их упростить. Это происходит потому, что от последствий страдают только люди, связанные с действительностью и реально выполняющие работу, в то время как остальные достигают прогресса в создании круговой поруки, говоря себе при этом, что они делают работу.

Коммуникационный барьер между картостроителями и паковщиками часто заставляет людей говорить, что эффект от навязанных административных накладных расходов ограничен. Есть три эффекта, которые может вызвать неэффективное администрирование. Они находятся на высоком уровне абстракции и, следовательно, как знают картостроители, имеют большую разрушительную силу.

Это отнимает время у настоящей работы. Некоторые организации требуют, чтобы работники заполняли отчеты о командировках, такие сложные, что люди на самом деле выделяют полдня в месяц только на заполнение этих отчетов. Да что там — 10% времени (и зарплаты) тратится на тупой ритуальный административный процедурализм! Данные отчетов могли бы собирать гораздо проще, а остальную конторскую обработку, если это так необходимо, могли бы делать клерки, которым платят меньше и которые многочисленнее.

Это прерывает нормальный ход дела. Часто требуется несколько часов, чтобы загрузить проблему в свое сознание, и если некто из отдела кадров постоянно прерывает по поводу проблем с их файловой системой, то разработчик за несколько рабочих дней не найдет нескольких секунд, чтобы упорядочить свои мысли о проблеме. Очень скоро это превращается в пытку водой, когда вывихнутый мозг программиста уходит от обдумывания проблемы, поскольку каждый раз, когда он вкладывает эмоциональную энергию, необходимую для загрузки требующей рассмотрения трудной, неструктурированной проблемы, его уводят в сторону. Это очень неприятный опыт. К алкоголикам подключали электроды и пускали ток, когда они прикасались к бутылке виски. Это то же самое.

Это забивает голову. Быть картостроителем — значит искать ясности и рассматривать множество вещей. Если назойливый и некомпетентный администратор обращает рабочее место в сюрреалистичный кошмар, то удержание фокуса на высоких стандартах ясности, необходимой для программирования, становится очень трудным делом, а если никогда нельзя предсказать, как долго займет покупка нужного программного пакета, то не может быть и речи ни о каком планировании.

Команды могут многое сделать, чтобы изолировать себя от административного хаоса в своих организациях, позволив людям, знающим правила игры, защитить остальных. Как хороший менеджер ограждает от внешнего давления и раздражителей команду разработчиков, чтобы она смогла сконцентрироваться, так и хороший администратор ограждает команду от паршивого администрирования.

Помните, что паковщики в организации не поймут описанные выше эффекты, поскольку они не признают существование такого подхода и состояния ума, с которым мы программируем. Это проблема офиса с открытой планировкой[5]!

[5]Когда все находятся в одном помещении — С.К.

❒ Моделирование ситуаций

Эффективный способ поддержания совместно используемой мысленной карты проблемы, проекта и деятельности группы состоит в регулярном проведении моделирования ситуаций. Это короткие собрания, где один человек за десять минут излагает текущее понимание положения группы. Как и все остальное, это не ритуал, который необходимо выполнять неукоснительно как неизбежную часть работы — он имеет свое назначение. Это означает, что моделирование ситуаций стоит проводить, даже если присутствуют не все члены команды, и, более того, его стоит начинать экспромтом, если уж собрались интересные люди.

Книга «Справочник рейнджера Слоана» *(Sloane Ranger's Handbook)* включает карту мира для рейнджеров Слоана. Около 50% всей карты занято Площадью Слоана, Шотландия соединена с Лондоном узенькой дорожкой M1, а континенты отодвинуты к краям карты. Шутка заключается в том, что у всех нас собственная искаженная карта мира, но карта рейнджеров уж очень сильно искажена по сравнению с географической. Для рейнджеров Слоана это не шутка — это правильное представление их мира, и они утверждают, что их представление не более нереальное, чем какое бы то ни было другое. (Некоторые из них купили книгу, чтобы иметь возможность проверить правильность. Она прошла проверку.)

Точно так же, поскольку у нас всех есть своя собственная карта мира, у каждого из нас есть своя точка зрения на проблему и деятельность группы. Выявление различий точек зрения на проблему у разных людей дает больше преимуществ группе в целом, чем если бы просто позволить членам выверять свою точку зрения на своих картах меньшего размера и выявлять качественные и фактические отличия (что моделирование ситуаций тоже делает) в отдельных разговорах. Если взгляд на проблему с разных сторон приносит понимание, выслушивание того, как описывает прикладную программу занимающаяся связью команда, может дать прикладным программистам вещи, о которых они даже не догадывались со своей точки зрения.

Чтобы понять, почему полезно прерываться для моделирования ситуаций на несколько минут в день для общего собрания группы, очень полезно подумать о двух различных физических принципах работы систем хранения изображений. Традиционные фотопленки хранят разные части всего изображения в разных частях поверхности пленки. Соответствие между плоскостью изображения и плоскостью пленки прямое. Оторвите кусочек, и этот кусочек изображения пропадет. Голографические же пластинки хранят преобразование целого изображения в каждой части поверхности пластинки. Отломите кусочек, но изображение сохранится, хотя и понизится разрешение, поскольку кусочек содержал информацию о распределении определенных частотных компонентов изображения.

Команде не нужно концентрировать знание о предмете в отдельных личностях для исключения всего другого знания. Если делать так, то результат будет печальным, поскольку команда не сможет общаться внутри себя. Распределение знания по всей команде должно больше походить на голограмму, чем на фотографию. Я должен знать многое о своей работе и немного о вашей. То немногое, что я знаю, должно быть правдивым и истинным, неважно, как причудливо я бы ни выражал это с вашей точки зрения. Тогда вы и я можем говорить друг с другом.

В моделировании ситуаций очень важно выдерживать строгий регламент (лимит времени), иначе вы безнадежно увязнете. Это означает, что у докладчика должно быть лишь несколько минут, чтобы кратко выразить Что На Самом Деле Произошло, оставляя вытекающие последствия за рамками. Это могут быть сравнения точек зрения, в которых члены команды не согласны с докладчиком, поиск возможностей упрощения, когда я узнаю, что я делаю на своем уровне такого, с чем вам приходится бороться на своем, либо получение знания от специалиста.

Также помните, что модель, где каждый имеет свою точку зрения — это групповая модель. Если в свете точки зрения кого-то обнаружился изъян в совместно используемой модели, то нападки на модель не означают нападки на человека, чей новый подход позволил обнаружить изъян, который вы не заметили со своей точки зрения.

Если группа освоится с подходом **зететики**, то из моделирования ситуаций можно извлечь дополнительные выгоды. Докладчика можно выбирать случайным образом. Это значит, что у каждого будет стимул регулярно прокручивать в своей голове весь проект, чтобы выглядеть элегантным и изобретательным, когда наступит его очередь рассказывать. Результат может быть потрясающим.

Когда последний раз вы делали работу, которая требовала от вас продумывания вашей работы, если вам требовалось писать отчеты о выполнении работы, заполнять многочисленные бланки и формы по контролю качества, заполняющие хранилище истории проекта?

Глава 4

Привычки и практика

❒ Руководства по кодированию

Суть Тотального Управления Качеством (TQM) — в сознательности. Осознание того, что мы делаем, когда выполняем повторяющиеся процедуры или подобную работу, позволяет нам ухватить те редкие моменты озарения, когда мы видим способы сделать работу более эффективно и сообщаем о них нашим коллегам, изменяя наш процесс. Это означает, что определение процесса может содержать скучные вещи, необходимые для поддержания взаимодействия (связи) между группами, но стилистические или определяемые подходом моменты не должны быть великой ценностью, которую стоит передавать в таких документах. Идея этого аспекта процесса — не задавать каждую маленькую операцию, а сохранить знание. Это дает нам критерий, хотя и субъективный, определяющий, какие пункты стоит включать в документ. Например, микроспецификации упорядочивания заголовочных файлов не подходят для включения в стандарт кодирования, поскольку помимо прочего, ими почти всегда пренебрегают на каждой реальной платформе на этапе компиляции кода. Однако, метод использования макросов условной компиляции вокруг всего содержимого включаемого модуля для предотвращения многократного включения является маленьким сокровищем, которое нужно поместить там, где каждый его может увидеть и будет ему следовать.

На автозаводах сборочные цеха подвергаются постоянному совершенствованию, поскольку люди, делающие непродуктивную работу, осознают, что они непродуктивны, и исправляют это. Оригинальная параллель с программной инженерией должна заключаться в том, что руководства по кодированию должны совершенствовать процесс. Одно из наиболее дорогостоящих последствий барьера между картостроителями и паковщиками состоит в том, что паковщики, паникуя из-за «кризиса программного обеспечения», отстаивают утверждение, что «процесс» — это необъяснимый и мистический источник всего хорошего и, будучи таковым, он правилен. В некоторых организациях процесс становится механизмом принудительных попыток внедрить тупой роботизм паковщиков на всех рабочих местах, поскольку он кажется правильным состоянием ума на пути к магическому успеху.

Чтобы осознать масштабы проблемы, рассмотрим эволюцию языков программирования и моделей, средств разработки, инструментов CASE и так далее за последние тридцать лет. На самом деле нет никакого сравнения между концами этого интервала ни в чем, за исключением стандартов программирования. Некоторые аспекты структурного программирования, которые начал обсуждать Дейкстра *(Djikstra)*, были превращены в ритуализированную догму, а затем эта догма перекочевывала из стандарта в стандарт и дошла до наших дней. Действительно, главная черта большинства стандартов кодирования, страстно пропагандируемых их сторонниками, состоит в том, что в них скопированы вещи, взятые откуда-то еще. Это история о том, как продать хлам запуганным и невежественным. Говоря, что угадал нечто, кому-то еще — хороший способ без всяких оснований добавить к чему-нибудь фальшивый источник. Стандарты кодирования создаются менеджерами для программистов, а не открываются во время коди-

рования и не передаются наверх. Такой оживленной и квалифицированной (если примитивно) дискуссии, которая вела к первоначальным стандартам кодирования, которые были великолепны для своего времени, с тех пор не повторялось. Как только был принят первый стандарт и были замечены улучшения, мир бизнеса паковщиков ухватился за них, обратил их в камень и объявил, что установлена «соответствующая процедура». Споры о стиле скатились до религиозных войн, не были повернуты лицом к потребностям развивающейся индустрии и потому были глупыми. Между тем, существование этой «соответствующей процедуры» неявно отвергало существование новых стилистических явлений, возникавших с появлением новых языков и моделей, которые нуждались в квалифицированных обсуждениях, каких же интенсивных, как обсуждения, развернувшиеся вокруг структурного программирования, для того, чтобы научиться использовать эти новые языки.

Программист, использующий среду разработки типа ParcWorks Smalltalk, получает столько же преимуществ от ханжеских разглагольствований о не использовании `goto` и помещении данных, на которые никто не смотрит, в стандартных строках комментариев в начале файла, как и авиадиспетчер, прочитавший во Второзаконии о наказании за секс с верблюдами.

❒ Кто украл мою мышку?

Этот раздел посвящен сложности. Мы начнем с мысленного эксперимента, касающегося воображаемой марсианской экологии.

На Марсе (как все знают) есть скалы. Там есть также два вида живых существ. Это марсиане, которые едят мышей, и, что кстати для марсиан, есть мыши. Мыши скрываются среди скал и грызут их. Мало что происходит на Марсе. В основном марсиане проводят свое время, сидя в пустыне и выслеживая снующих между скал мышей. Поскольку скалы на Марсе повсюду, куда ни кинь взгляд, по всем направлениям, то марсианам нужно уметь хорошо видеть сразу во всех направлениях. Поэтому у марсиан характерные направленные в разные стороны четыре больших глаза, каждый на стебельке.

Мало что происходит на Марсе, поэтому эволюция марсиан полностью направлена на совершенствование обнаружения мышей. У каждого огромного глаза сзади расположена огромная мозговая корка, которая может обнаружить мышь за несколько миль в любое время суток (при любой освещенности). Большая часть мозга марсианина состоит из зрительной коры, и эти четыре подмозга имеют многочисленные связи, чтобы компенсировать неблагоприятные условия освещенности. Марсиане интенсивно обрабатывают изображение ландшафта в полуавтономных подмозгах, поэтому им на самом деле не требуется «внимание», как людям — вместо этого они сосредоточены на переключении входов между своими «вниманиями».

Обнаружив мышку, марсианин должен подкрасться к ней. Это означает, что нужно преодолеть скалы. Это требует Разума. Вскоре после обретения Разума марсиане изобрели Великую Литературу. Ее выбивали на больших скалах с помощью маленьких обломков скал. Она подчиняется правилам марсианской грамматики и использует Северный голос для эмоций, Южный голос для действия, Восточный голос для речи и Западный голос для обстоятельств. Мало что происходит на Марсе, поэтому наиболее близкий марсианский эквивалент нашего «Преступления и наказания» называется «Кто украл мою мышь?»:

Emotion	**Action**	**Speech**	**Circumstance**
Grumpy	Sneak	Horrid Martian	Cold, Gloomy
Determined	Bash	Die! Die! Die!	Old Martian's Cave
Ashamed	Steal Vole		Dead Martian
Эмоция	**Действие**	**Речь**	**Обстоятельства**
Сердитый	Подкрадывается	Ужасный марсианин	Холодный, Мрачный
Определенный	Бить	Умри! Умри! Умри!	Старая марсианская пещера
Стыдно	Украсть мышь		Мертвый Марсианин

Захватывающе, не правда ли? Этот неожиданный пробел в Восточном голосе, сменяющий Южный голос — непроизнесенное проклятие, очень реальное само по себе! Я утверждаю, что это помогает получить правильную структуру мозга...

В чем смысл этой Странной Истории? Хорошо, представим, что марсианский программист мог бы создать теорию взаимодействующих последовательных процессов Ч. Хоара — CSP[1]. Их мозг (легко избегаем рода — у марсиан 16 полов и для организации брачной ночи требуется десять лет) уже создан таким *(hardwired)*, чтобы позволить им почувствовать сложные отношения между независимыми действиями, поэтому процессы, которые Хоар располагает линейно, используя последовательности символических преобразований и делая их таким образом понятными человеку, становятся непонятными марсианам.

Попытки разработать систему управления совместного человеко-марсианского космического корабля, в которой множество взаимодействующих последовательных процессов управляют двигателями, фотонными пушками и т. п. столкнутся с проблемами, напоминающими коммуникационный барьер между картостроителями и паковщиками, даже несмотря на то, что доказыватели теорем с помощью CSP могли бы предложить автоматические трансляторы на основе своих идей.

Дело в том, что сложность находится в глазу зрителя. Нам не требуется инопланетная философия, чтобы заметить различие между тем, как отдельные личности оценивают сложность, — это полностью сфера мысленных карт. Когда мы открываем структуру, которая позволяет нам понять, что происходит, то можем понять истоки гораздо большего числа явлений с одного взгляда. Поразмышляйте над любой сложной ситуацией, в которой вам нужно разобраться, будь то палуба яхты или сцена любительского драматического театра. Когда вы впервые посмотрите на нее, она будет выглядеть как хаос из тросов, блоков, ящиков, трапов, железяк совершенно непонятного назначения. Когда вы получите картину, выяснив назначение каждого предмета, палуба, сцена и все что угодно кажется уже просторнее, аккуратнее, чем когда вы первый раз посмотрели на нее. Но там ничего не изменилось, изменились вы.

Ни один здравомыслящий шкипер или режиссер не стал бы пытаться действовать таким образом, чтобы любой самый зеленый новичок смог с первого раза понять, что происходит. Даже плавание по бухте или поднятие занавеса требуют некоторого навыка.

В программной индустрии метод рычага, коммуникационный барьер картостроитель/паковщик и языковая специализация маскируют этот момент. Метод рычага говорит, что преимущество, которое мы получаем, перемещая числа вместо кирпичей, означает, что мы можем приложить усилия, чтобы простые числа, описывающие кирпичи, были представлены в доступной всем форме. Индустриальный и коммерческий контекст многих операций программирования подразумевает, что раз специалист организовал свою информацию, то ее сложность управляется сама собой, а, исходя из этого, прогресс кирпичей гарантирован. Все это так, и это вполне правильный подход к информации о кирпичах. Но когда мы подменяем информацией кирпичи, то получаем проблему. Мы не можем абстрагировать от большого безобразного кирпича к информационной «1». Конечно, мы можем абстрагировать, но каждый раз, когда мы делаем это, мы теряем важную информацию, которая позднее может сыграть с нами злую

[1] C.A.R. Hoare's Communicating Sequential Processes

шутку. Мы не можем просто сказать, что специалист упорядочил свои данные, поскольку работа в наше время состоит в организации огромного потока данных, который сваливается на нас. Нам больше не нужны навыки оператора автопогрузчика, но нам необходимы новые навыки. И мы не можем управляться с представлением сложности, просто требуя, чтобы представление было простым. Коммуникационный барьер между картостроителями и паковщиками делает обсуждение ситуации с такой неадекватной аналогией между информацией и кирпичами еще более сложным просто потому, что почти в каждом шаге логики кирпичей есть информационный аргумент, но вместо 1% управления и 99% полезной нагрузки, проблема менеджмента скорее в 90% управления и 10% полезной нагрузки. Это соотношение — то, что отличает все эвристики управления кирпичами, и это то отношение, которое, к сожалению, задают паковщики. Они думают, что они распознали ситуацию, щеголяют пакетом знаний, аккуратным и документирующим все, и на этом останавливаются. Идея о том, что, может быть, они изобрели многоступенчатую ракету, которой никогда не оторваться от земли, поскольку двигатели слишком неэффективны, и понадобится гораздо больше менеджмента, чтобы компенсировать недостаток искусности, очень тяжела для осознания теми, кто не обучен рисовать мысленные модели и видеть в них структуру. Наконец, существование специализированных языков (профессиональных жаргонов) способствует появлению такой возможности. Если бы все было так же просто, как SQL. Нужно помнить:

1. Это заняло 30 лет.
2. Область применимости очень ограничена.
3. Это требует большой вычислительной мощности процессоров.

SQL не прост. Это в точности то, что мы описали — управление известными вещами с помощью идиом — каждый понимает ужасы внешних связей таблиц *(outer joins)*.

И вновь здесь возникает «ход конем». Что лучше: разработать действительно сложный код, послать его, и затем получить короткий ответ, или послать простой код и получить длинное сообщение. Каковы предположения о квалификации и опыте администратора базы данных? Насколько мы можем расширить их понимание в документации, чтобы мы могли использовать более «сложные» идиомы? Огромный `switch()` — обычно довольно ужасный способ управлять процессом, если только вы не пишете цикл обработки событий GUI, где мы все его предполагаем и ищем его.

Нет абсолютной меры «сложности». Это должно родиться в мозгу, в обсуждениях сложности алгоритмов, и стиля кодирования, и того, что выдают средства анализа сложности после графического представления системы, которое может оказаться очень полезным. Сложность в этих картинках (после того как систему упростили до необходимой достаточности) не есть Плохая Вещь — это сущность лежащей перед вами проблемы. Нам не следует пытаться избегать внутренней (присущей) сложности проблем. Это бесполезная стратегия, уводящая нас от достижения абстрактного понимания, которое позволяет нам упорядочить сложность.

❒ Рецензии и анонсы

Фундаментальный элемент взгляда паковщика на работу — управление посредством угрозы. Выполни действие так и так, и не иначе. Чтобы сделать угрозу эффективной, исполнение правил должно контролироваться (полицейскими методами), поэтому мы должны следить, гарантируя этим следование правилу. Тогда ленивые и ненадежные работники будут знать, что их уволят, и будут выполнять действия как предписано, из страха наказания. Действительно, важное соображение, из-за которого отвергают стиль работы картостроителя, поддерживается заблуждением, что работа картостроителя может быть специфицирована по шагам *(microspecified)*, а единственная причина, почему это необходимо делать — то, что правила

должны быть специфицированы. Только для записанных на бумаге правил можно выявить их нарушение программистами, и возможность этого выявления поставлена в центре всей модели!

Конечно, есть еще одна цель — в том, что рецензия также может выявить небольшие оплошности, которые необходимо поправить перед тем, как работу можно будет отдать заказчику. Это напоминает осмотр «Роллс-Ройса» и стирание пятнышек на капоте мягкой тряпочкой — но это не превратит какую-нибудь колымагу в «Роллс-Ройс».

В программной индустрии у нас есть процессы, которые уделяют значительное внимание рецензиям, что само по себе либо тривиальная, либо полицейская акция, но ничего не делают, чтобы обратить основное внимание команды и опыт группы на нахождение наилучшего способа сделать работу. Это приводит к рецензиям, на которые отдельные работники или подгруппы тратят свои лучшие силы, но которые оказываются откровенно плохими, особенно если за дело берутся неопытные программисты. Время на выполнение этой работы потрачено, поэтому если для решения задач, с которыми способна справиться сама операционная система (бесплатно), были выбраны какие-нибудь древние библиотеки процессов уровня приложений со сложными правилами инициализации, то на перепроектирование просто не остается времени. Членам группы рецензирования приходится сговариваться не замечать изъяны логики и сконцентрировать свои усилия на ритуальных предметах, касающихся просто стиля. Это ничего не дает для качества.

Решение, конечно, картостроительское. Мы должны принять, что программисты любят делать настоящую хорошую работу, если им дать шанс, и нужно инвестировать в помощь им, а не в их запугивание. Вместо того, чтобы тратить полдня на рецензирование, когда уже поздно что-то менять, следует проводить это время над анонсом *(preview)*, в котором мы могли бы оценить условия и согласовать общее направление до того, как сделана работа. В такой ситуации от работы по рецензированию можно просто отказаться, поскольку если уж получился «Роллс-Ройс», то финальное рецензирование нужно лишь для проверки, не осталось ли пятнышек на капоте.

❒ Инспектирование кода и пошаговые проверки

Инспектирование кода составляет важную часть процесса многих организаций. Изначальная причина, почему его делают — удовлетворить здравый смысл. TQM требует: «Если вы сделали работу, посмотрите на то, что вы сделали, и убедитесь, что с этим пунктом все в порядке». Но в инспектировании кода появляется нечто забавное, часто искажающее его назначение. Это вызвано тем, что инспектирование кода похоже на Рождество. Это старое действо, которое старше структур, с которыми оно имеет дело. И, как и Рождество, содержит множество мишуры, происходящей из Старой Религии.

Когда-то давно программы приходилось писать на специальных листах, которые передавались операторам для набивки перфокарт. Результат распечатывался для проверки, затем перфокарты загружались в компьютер. И все это имело смысл из-за необычайно высокой стоимости работы компилятора. И это была не просто стоимость работы и амортизации — время цикла одного прогона могло составлять неделю. Поэтому было мудрым решением выработать привычку посидеть и проверить кодовые листы друг друга перед тем, как отправить их на перфорирование.

Сейчас у нас на столе мощные редакторы и процессоры, поэтому изначальные мотивы больше неприменимы, но нам по-прежнему нужно продолжать проверять наш код, чтобы можно было идентифицировать логические ошибки до того, как они уронят систему. Здесь заключается причина помрачения сознания. У некоторых организаций так же помрачено сознание, как у того IT-менеджера, который недавно сказал, что работники должны выполнять инспектирование кода по листингу, перед тем как его компилировать. Чтобы избежать неприятностей, у нас есть стадия проектирования для получения правильной большой картины и компилятор для

обнаружения синтаксических ошибок. Ручной просмотр кода в поиске синтаксических ошибок не дает ничего полезного, а является очень дорогим и ненадежным методом поиска синтаксических ошибок, несмотря на то, что так делали всегда! Баланс изменился, и нам нужно свериться с нашими мысленными картами, как и во всем, что нам приходится делать в этой насыщенной информацией игре.

Хотя в большинстве организаций людям разрешают компилировать и тестировать код перед инспектированием, падубовая веточка все еще висит над очагом. Почему это полдюжины людей вдруг вскочили, бросили свои навороченные браузеры классов, средства абстрактного моделирования, среды разработки с графическим интерфейсом и удалились с пачкой листинга на целый день, организация за организацией, день за днем? Пусть это делается на компьютере, чтобы можно было, например, запустить поиск и получить ответ на вопрос, всегда ли эта величина положительна.

Инспектирование кода очень дорогое удовольствие, и нам следует избавиться от него. Очень хороший способ сделать это (при наличии символического графического отладчика) — разбить работу на две части. Сначала отдельный программист, который хорошо знаком со структурой и назначением кода, использует отладчик в пошаговом режиме и тестирует программу. Это может казаться очень трудоемким и малопродуктивным, но эффект потрясающий. Программа сама естественным образом отслеживает путь, который проверяет разработчик, а глаза разработчика фокусируются на каждом отдельном шаге логики. И не придется трассировать ошибочную часть, чтобы ее увидеть, поскольку ум работает гораздо быстрее пальца на кнопке мыши и быстро выявляет такие вещи, если устремлен в правильном направлении. Может оказаться полезным распечатать листинг, разбить его горизонтальными линиями по функциям и блокам кода, и рисовать вертикальные линии на полях листинга по мере проверки. При этом используется знание разработчика, помогает машина, и для этого требуется один человек, хотя выявляется множество проблем. А полная групповая инспекция кода может затем сфокусироваться на вещах, которые может привнести свежая точка зрения, например обнаружение неявных предположений, которые могут оказаться неверными, по мере того, как разработчик объясняет логику.

Инспектирование кода, организованное таким образом и совмещенное с индивидуальной пошаговой проверкой, имеет свое назначение, и маловероятно, что оно деградирует до маразма религиозных войн по поводу стиля комментариев, на которые сейчас тратят свое время многие высокооплачиваемые программисты.

❒ Стандарты кодирования и руководства по стилю

Стандарты кодирования и руководства по стилю уже несколько раз появлялись в этой работе, и уже сказанное могло сильно отличаться от общепринятых убеждений. Будет полезно изучить эти различия и понять в точности, о чем мы говорим.

Мы показали, что индустрия программирования часто пытается расположиться между двумя очень различными взглядами на мир. Паковщиков выдрессировали структурировать свои доводы и рассуждения вокруг идентификации ситуации с заученными ответами, а затем исполнения того действия, которое та предписывает. Их поведение в мире основано на том, чтобы делать правильные вещи. На работе им говорят, что делать. Картостроители стремятся получить общее понимание ситуации, применяя известные паттерны там, где они применимы, и создавая новые там, где их нет. Картостроители, если им задана цель, в своих действиях руководствуются своими картами. На работе они понимают проблему и находят оптимальное решение. Мы также увидели, что картостроительный подход — это то, как создается новое программное обеспечение, и его нельзя создать паковкой.

Поэтому стандарты кодирования отражают частично мотивацию картостроителей и частично мотивацию паковщиков, то в результате цели создателей стандартов перепутываются. Про-

является коммуникационный барьер между картостроителями и паковщиками, поэтому картостроители рвут на себе волосы, ибо паковщики растирают сложность, пока она не станет невидимой на каждой отдельной странице, цитируя Пакет Знаний 47684 (Ясность — Это Хорошо) и в то же самое время вытирая ясность из работы.

Если мы примем, что картостроение и TQM есть основы хорошего программирования и что суть картостроения и TQM — понимание и управление (контроль), то можем посмотреть на цели и увидеть, что возможно сделать для улучшения стандартов кодирования и руководств по стилю.

Сначала о **ясности**. Есть идея, что использование синтаксического богатства языка — это Плохая Вещь, поскольку это «сложно». Ни когда составной синтаксис образует идиому. Ни даже когда эта идиома вводится и обсуждается в документации. И вообще не нужна ни при каких условиях. Когда Ньютон писал «Принципы» *(Newton's* Principia*)*, то написал их словами, хотя мог использовать алгебраические символы, поскольку был сооткрывателем флюксий, или исчислений. В наше время мы решили, что в математике лучше иметь дело с алгебраической нотацией даже при описании. Сейчас то, что потребовало бы нескольких страниц текста, в алгебраической нотации займет всего страницу, и хотя при этом скорость чтения страницы снижается, общее время чтения стопки текста больше, поскольку в стопке текста гораздо труднее проследить мысль. Итак, следует ли нашим программам быть более похожими на прозу по концентрации сложности на страницу или на выражение, либо им следует быть ближе к математике? Мы предполагаем, что если человек — новичок в языке, то лучше, если он сможет отчетливо увидеть всю структуру и тратить на каждую страницу несколько минут, чем читать каждую идиотскую строчку и не понимать, для чего это написано! Как часто нам приходится видеть напряженных людей, сидящих перед стопкой кода, не имеющих понятия, с чего начать, и думающих, что это их вина.

Второй момент — **соглашения**. Прежде чем принять соглашение, убедитесь в том, что оно даст больше, чем будет стоить. Мы приводили ранее пример, в котором если кто-то не знает назначение переменной, то он ничего не получит от знания ее типа[2]. Но это не просто вклад в перегрузку мозга соглашениями, которые, как предполагается, «хорошие» программисты должны хранить в виде пакетов знаний, что само по себе проблема. Дело еще и в том, что наличие слишком большого количества соглашений делает код некрасивым и даже уродливым. Если стремиться к красоте минимального кода, то этого сложнее достичь постоянно натыкаясь на обвешанных мусором уродцев типа `gzw_upSaDaisies`. Никогда не делайте того, что подавляет стремление команды создать великий продукт. Было место, где думали, что соглашения — Хорошая Вещь. Несколько людей были назначены Создавать Правила, и они делали это должным образом. Один из них объявил, что длина имен переменных должна ограничиваться 31 символом. Очень разумно — многие компиляторы могут различать имена не длиннее указанного. Другой объявил, что переменные, описанные в подсистеме, должны начинаться с трехбуквенного альфа-кода подсистемы. Любая переменная, даже если никто не использует ее глобально. (Еще один предложил помечать глобальные переменные.) Еще один произвел причудливую схему типов, параллельную собственным составным типам языка, и потребовал включения информации об этих типах в имена. Зачем — мы не знаем. Еще один опубликовал список сокращений имен модулей кода, известных менеджеру конфигурации и сказал, что это тоже необходимо включить в каждое имя переменной. И т. д. Веселье на самом деле началось, когда оказалось, что все эти обязательные штучки плюс пометка типа «указатель на функцию, возвращающую указатель на запись» превысили 31 символ. Создатели Законов просто сказали, эта конструкция очень сложна и ее не следует использовать, хотя это было главным для архитектуры, и топтались вокруг описывания промежуточных переменных и приведения типов во время присваивания им значений, что совершенно не помогало ни ясности, ни эффективности. Поэтому, наконец, пузырь лопнул и мы разработали некоторые прагматические стандарты, которые хо-

[2]Тут идет речь о венгерской записи — С.К.

рошо выглядели и говорили нам, как с помощью словаря проекта быстро образовать имя и некоторые разумные сокращения. Если посмотреть с точки зрения картостроитель/паковщик, мы видим, что ситуация развивалась подобным образом потому, что Создатели Законов были Объявителями Полезных Правил, что есть Хорошая Вещь, но цена, цена...

Третий момент — о природе строительных кирпичей. Для целей сохранения хорошего порядка руководство по стилю, содержащее многослойный винегрет из соглашений об именовании, идиом и примеров модулей для разработчиков будет служить вам лучше, чем коллекция императивных (обязательных к исполнению) инструкций, ограничивающих способность программиста искать элегантность самостоятельно. Если вы не можете доверить команде самостоятельно решить, когда заключать блок в скобки, а когда нет, как вы можете доверить ей написать ваш суперпроект?

Четвертый момент — об этих императивах. Есть некоторые средства, которые вообще не стоило изобретать, например `scanf()` и `gets()` в UNIX. Указания никогда не использовать их в разрабатываемом коде разумны. Но есть цели, которых просто невозможно безопасно достичь другим, лучшим путем. И всегда остается проблема баланса ясности. Мы рассмотрим два конкретных примера, из которых станет видно, что в Си все же существует хороший повод для использования `goto` — хотя вам могли говорить, что таких поводов нет.

Вот первый, здесь нет другого способа. Представьте рекурсивный обход двоичного дерева:

```
void Walk(NODE *Node)
{
    // Do whatever we came here to do...

    // Shall we recurse left?
    if(Node->Left) Walk(Node->Left);

    // Shall we recurse right?
    if(Node->Right) Walk(Node->Right);
}
```

По мере обхода дерева, мы начинаем делаем вызовы, ведущие нас влево, влево, влево, влево... и так до самого низа. Затем просматриваем каждую комбинацию влево, влево, вправо, вправо, влево... и т. д., до тех пор, пока не просмотрим все ветви, и не проделаем все возвраты из нашего последнего визита, который был вправо, вправо, вправо, вправо...

Каждый шаг влево или вправо включает в себя открытие нового кадра стека, копирование аргумента в этот кадр стека, выполнение вызова и возврат. Для некоторых задач с многочисленной навигацией, но небольшой обработкой в узле, эти накладные расходы могут оказаться чрезмерными. Но посмотрите на такую мощную идиому, известную как устранение хвостовой рекурсии:

```
void Walk(NODE *Node)
{
Label:// Do whatever we came here to do...

    // Shall we recurse left?
    if(Node->Left) Walk(Node->Left);

    // Shall we recurse right?
    if(Node->Right)
    {
        // Tail recursion elimination used for efficiency
        Node = Node->Right;
```

```
        goto Label;
    }
}
```

Мы используем стек для отслеживания того, куда мы попали слева, но после того, как мы обошли левую часть, переход на правую ветвь не требует хранения положения. Поэтому мы устраняем 50% вызовов и накладные расходы на возвраты. Это может изменить ситуацию при выборе между реализацией на Си или добавлением ассемблерного модуля. Чтобы увидеть другой пример, посмотрите на конструкцию Даффа *(Duff's Device)* в книге Страуструпа «Язык программирования C++»[3].

Второй пример касается чистоты стиля — вообще нет необходимости применять язык ассемблера. Помните, что когда Дейкстра посчитал `goto` вредным, он имел в виду привычку использовать `goto` для организации управления в неструктурированном коде 60-х годов. Идея заключалась в том, что, меньше используя `goto`, мы могли бы улучшить ясность. Идея не состояла в жертвовании ясностью, избегая `goto` любой ценой. Представьте программу, которой нужно открыть порт, инициализировать его, инициализировать модем, установить соединение, зарегистрироваться *(logon)* и загрузить файл *(download)*. Если что-то не так, в любом месте, нам нужно вернуться обратно в самое начало. Доморощенный структуралист мог бы написать нечто вроде:

```
BOOL Done = FALSE;

while(!Done)
{
  if(OpenPort())
  {
    if(InitPort())
    {
      if(InitModem())
      {
        if(SetupConnection())
        {
          if(Logon())
          {
            if(Fetch())
            {
              Done = TRUE; // Ouch! Hit the right hand side!
            }
          }
        }
      }
    }
  }
}
```

...что нам кажется просто глупым. Есть более понятная альтернатива, использующая то, что оператор `&&` прекращается сразу, как только встречается выражение, принимающее значение `FALSE` — «неправильное использование» языка, обычно запрещаемое в большинстве стандартов кодирования:

```
while(!(OpenPort()&&
```

[3] Stroustrup's *The C++ Programming Language*

```
        InitPort()&&
        InitModem()&&
        SetupConnection()&&
        Logon()&&
        Fetch()));
```

Здесь все ясно и удобно, поскольку мы можем инкапсулировать каждый шаг в функцию. Проблема в коде такого рода заключается в том, что требуется правильно сделать целый ряд ужасных вещей, например инициализацию строк и т. п., и чтобы работать с таким кодом, нужно выполнить его в очень похожем на скрипт виде. Например, так:

```
Start:  if(!OpenPort())goto Start;
        if(!InitPort())goto Start;
        if(!InitModem())goto Start;
        if(!SetupConnection())goto Start;
        if(!Logon())goto Start;
        if(!Fetch())goto Start;
```

Это в точности то, что позволяют нам делать специализированные скриптовые языки, разработанные для такого вида работ!

Не забывайте: если вы хотите, чтобы предмет вашего обожания понял ваше любовное письмо, вы не позволите педантизму правописания и грамматики исказить его, а если вы хотите, чтобы ваши коллеги поняли вашу программу, не перекручивайте ее структуру во имя «чистоты».

❒ Значимые метрики

Мы можем повернуть линзы практического понимания цели на получение и интерпретацию метрик, на которые в некоторых местах тратится огромное количество денег, что заставляет нас думать, что они правильные. Имеется три мотива коллекционирования чисел. Все они ценные, но всегда важно понимать, в чем состоит наше намерение. Их три:

Описательная наука. Сюда относятся получение и коллекционирование данных о предмете, чтобы увидеть, можно ли найти какие-либо интересные свойства данных. При этом не нужно знать, что ожидают найти. Нецензурированные сырые данные — источник всего. Современная этимология в огромном долгу перед леди времен Виктории и Эдварда, которые проводили свое время в создании очень детальных акварелей каждой бабочки или насекомого, которых они только могли найти. Уже стало традицией, что действительно интересные кометы одновременно открываются профессионалами и астрономами-любителями. Наша дисциплина пострадала от непродуманного переноса «метрик» массового производства на интеллектуальную, трудоемкую деятельность. Если мы хотим построить аналогии с фабриками, то нам нужно задать вопрос о том, что оптимизирует сложные человеческие факторы в нашем производстве. Нам нужно проводить больше времени у истоков. Например, много ли дает знание того, что проверка потерпела неудачу в коде, написанном летом, когда есть много других вещей, которые мы могли бы сделать вместо того, чтобы поставить время проверки? Можно заложить окна кирпичом или заложить сезонный фактор в планы работы предприятия, чтобы максимизировать качество. Что есть индикатор качества? Внутренние или внешние отчеты об ошибках на одну функцию? Число строк кода на одну функцию?

Экспериментальная наука. Сюда относится внесение изменений в иным способом управляемую среду, чтобы посмотреть, получается ли тот результат, который ожидается. Это

позволяет нам проверять и улучшать нашу мысленную карту рабочего места. Это очень легко делать на массовом производстве и очень трудно в программной инженерии, где время цикла может составлять месяцы, может меняться состав команды и ни одна работа не похожа на другую в точности. Можно также нанять хорошего статистика с реальным пониманием искусства программирования или поискать настоящего большого победителя, который издает шум. Мы знаем, что есть большие победители, поскольку существуют хакеры. Эта работа делалась, чтобы указать профессионалам области для исследования, где прячутся большие победители.

Кибернетическая технология. Это область, где мы действительно знаем, что мы делаем. Перед тем как измерить, мы знаем, как мы будем это интерпретировать, и какую переменную мы будем подстраивать с помощью записываемого значения. Если программная инженерия на самом деле сидит в луже, то это то, что нам следует делать. Но, к сожалению, не делаем. Эта область настолько сложна, что, вероятно, никогда и не будем делать, но мы можем разработать некоторые очень хорошие эвристики. Мы должны принимать во внимание, что культура паковщиков вынуждена прикидываться, что мы уже все полностью контролируем, но это не должно останавливать нас на пути к достижению лучшего частичного контроля, окружая таинственностью наши действия и интерпретацию той статистики, которую мы можем получить.

Здесь проявляется паттерн: не ставь телегу впереди лошади. Если мы занимаемся сбором статистики без ясного понимания того, что мы делаем, то важный инструмент превращается в упражнение по подсчету зерен. Без разумной интерпретации люди занимаются получением артефактов в статистике, а не улучшением выполнения работы, используя статистику в качестве индикатора этого улучшения. Это не порок машинных инструментов. Без ясной кибернетической модели «плохая» статистика становится палкой, которая бьет людей: «Это люди плохие и следует посмотреть на их грехи. Это наверняка поможет добиться улучшения». Начнутся собрания, на которых будут пытаться разными способами подсчитать ошибки, чтобы «улучшить ситуацию», но создаваемая программа все равно не будет работать.

С помощью метрик, как и со всем остальным, мы ничего не сможем сделать, отвергая нашу ответственность в пользу процедуры.

❒ Надежда на инструменты

Использует ли разработчик социально нормальную стратегию паковщика, или хорош в картостроении, он будет подвержен сильному влиянию надежды на инструменты. Паковщик смотрит на инструмент как на машину, которая выполняет работу, как копир в углу офиса. Действительно, таким образом большинство из нас использует компиляторы — засунь исходник с одной стороны, исполнимый код выйдет с другой. Обычно здесь все нормально, хотя день, проведенный за чтением руководств по компилятору и компоновщику многократно окупится.

Паковщики любят большие дорогие инструменты со сложным интерфейсом и неимоверно сложным внутренним состоянием. Они обещают делать все и требуют недель на установку и настройку. В них содержится множество сложных технологий. Все эксперименты заканчиваются плачевно. Среди шума легко потерять этот замалчиваемый глянцевыми рекламными проспектами факт. Программирование — это операция по обработке данных, которую этот продукт автоматизирует для вас, поэтому вы можете носить галстук, много улыбаться и быть «профессионалом» — вот что написано в рекламе.

Картостроители не рассматривают инструменты как копиры, они рассматривают их как протезы ума. Они — мыслительный эквивалент Рипли из фильма «Чужие» *(Ripley in the film Aliens)*, взятой в рубку космического корабля, чтобы победить главное чудовище. Картостроители оставляют за собой ответственность за все и используют инструменты для расширения

своего кругозора и силы мысли. Картостроители не любят помещать все свои штучки в один инструмент, где их другой и не найдет. Им нравятся коллекции инструментов и чтобы ввод/вывод был текстовым и анализируемым *(parseable)*, так чтобы они могли объединять (соединять) инструменты вместе.

Картостроители считают разумным писать небольшие программы на лету, чтобы манипулировать исходным текстом. Они отдают себе полный отчет в том, что делают хорошо они, а что хорошо делают компьютеры, используя свои собственные суждения, когда проверяют каждый вызов, скажем, функции, чье определение они изменяют, а компьютер гарантирует, что они проверили каждое появление вызова функции в тексте.

Имеются великолепные инструменты картостроителя — браузеры, инструменты восстановления исходного текста *(reverse engineering — дизассемблеры)*, даже некоторые «интегрированные среды разработки» *(IDE)*. Однако, стоит держать в уме, чего можно достичь на большинстве систем просто с помощью нескольких скриптов и системного редактора. Одна команда пришла в волнение, когда им показали инструменты, которые давали им все возможности для просмотра и средства получения перекрестных ссылок. Но отличия между этими инструментами и пучком скриптов, которые у них уже были, и на написание которых понадобилось одно утро заключались в том, что:

1. Инструменты нельзя было модифицировать.
2. Инструменты стоили 20,000 фунтов стерлингов плюс 5,000 фунтов стерлингов за рабочее место.
3. Инструменты требовали несколько недель на установку и настройку.
4. Инструменты были с графическим интерфейсом.

Когда кто-то с энтузиазмом рассказывает о новом продукте, остановитесь на минутку и проверьте, может быть правильной реакцией будет: «Ну и что?»

❒ Структуры программы — структуры проблемы

Можете себе вообразить Марго Фонтейн *(Margot Fonteyn)* с ногами и руками в гипсе и парой кандалов на лодыжках? Очень не грациозно.

Одна из наигрустнейших вещей — позволить молодым разработчикам начать работу с пошагового подхода, который позволяет им начать создание программы, а затем по мере работы знакомить их с идиомами, позволяющими отобразить проблему на язык и подход. Основной момент в том, что используемые ими языки и подходы предназначаются для отражения предметных областей. С большим удивлением наблюдаешь, как они начинают рассматривать и описывать проблемы в терминах программных структур. Черта, разделяющая хороший способ описания проблемы и хороший способ ее решения, размыта, а использование объектных подходов и языков максимизируют это размытие.

Но как раз в момент, когда они могут стать умелыми и изобретательными, эти разработчики начинают чувствовать себя виноватыми, поскольку им следует «видеть проблему, а не решение». Поэтому они начинают своеобразный спектакль, где они утверждают, что не могут видеть сути вещи, о которой они говорят. Если перед ними поставлена специфическая цель, например, обеспечение независимости от реализации, этот маневр может быть проделан с большой ловкостью, поскольку они точно знают, чего они не знают, и это становится упражнением в строгости, но если они просто стараются быть глупее, чем они есть на самом деле, когда они собираются остановиться?

Если ты хороший, то ты просто находка для своей организации, поскольку можешь сжато изложить, что решение Y хорошо для проблемы X и почему, следующий вопрос, пожалуйста!

Это не преступление — быть умелым и квалифицированным.

❒ Анализ причин

Анализ причин формализован как составная часть процесса многих организаций. При этом организация распознает ситуации, где все пошло наперекосяк, смотрит на то что произошло, чтобы понять, что же произошло и гарантировать, что это не произойдет снова. У картостроителей нет проблем с этим — это обычное дело. Но для паковщиков — это совершенно необычная вещь, с которой приходится сталкиваться в работе.

Чтобы понять, что же самое важное в анализе причин, мы можем посмотреть, как картостроитель делает то же самое, но под другим названием, в выявлении требований.

Представим себе транспортную фирму, которая по своей политике классификации традиционно разделяет все работы на сельские и городские. Деление на сельские и городские может хорошо работать в каждом аспекте бизнес-процесса. Когда инженер-программист начинает изучать требования для новой системы, очень важно, чтобы он посмотрел на закономерности потока данных и не растерялся от постоянного упоминания заказчиком о сельском и городском, что не влияет на большинство требований, чтобы не внести в разработку удвоенную сложность.

Урок в том, что необходимо видеть то, что есть, а не то, что, как тебе говорят, ты должен увидеть. Это значит, что ты должен отстраниться от того, как видит систему заказчик.

При выполнении анализа причин на рабочем месте, важно видеть то, что действительно происходит, а не выражать явления на языке процесса. На автозаводе добавление резинового улавливателя к конвейеру может предотвратить повреждение деталей, но мы редко владеем всеми элементами ситуации, как в случае деталей на сборочной линии. Если явления всегда выражаются в терминах процесса, то наиболее вероятное заключение будет состоять в том, что козел отпущения не смог отследить процесс, который служит двоякой цели паковщиков: уклонению от ответственности и празднованию совершенства процесса.

На самом деле, источники проблем можно классифицировать с точки зрения вовлеченности в процесс как:

Несвязанные. В течение месяца команду косит ветрянка. Мы ничего не можем с этим поделать, изменяя процесс, но, вероятно, мы можем собрать некоторые интересные наблюдения для передачи руководству — о недостатках управления рисками.

Операционные. Заказ нужно было ввести в систему, но этого не произошло. Обычно это рассматривается как отдельная проблема, на самом деле редко возникающая. Даже когда она возникает, предположение паковщика о недобросовестности клерка и закрытие таким образом проблемы неверно, поскольку на самом деле не выявлена причина. Вероятно, клерк был плохо обучен. Вероятно, процесс очень сложен. Вопрос — почему не введен заказ. Эта проблема может иногда быть решена, если перевернуть вверх дном описание процесса, но обычно все сводится к вопросам морали или заинтересованности в работе. Т. е. к социологическим эффектам, которые сильны на рабочем месте, но не включены в процесс.

Эргономические. В принципе, с процессом все в порядке, но реализация нежизнеспособная. Клерку приходится также продавать за наличные, а постоянное отвлечение влияет на аккуратность ввода данных. Оставьте определение процесса таким как есть, но добавьте немного здравого смысла в реализацию.

Процедурные. Процесс плохо определен. Потребители заказывают комплектующие на основании накладных, которые мы получаем от наших поставщиков, поэтому потребители платят за комплектующие, которые нам еще не доставлены, и получают сбой системы. Измените процесс.

❒ Уменьшение сложности и последовательное ужимание

Все интересные системы, от игровых до расчетных, содержат операции, которые делают состояние более сложным, а другие его упрощают. Большинство формальных процессов и обучение программированию фокусируются на накоплении массы. Для сбрасывания лишнего веса и получения преимуществ, которое оно приносит, мы должны проявить инициативу. Нам может быть поставлена задача написать функцию, но никто не поставит задачу обобщить ее, чтобы она решала три других.

Огромные опухоли усложнения всегда ходят парами, одна для закручивания вещей, другая для их раскручивания. Это распространяется на выявление требований, когда у пользователя обычно появляется желание воспроизвести функциональность существующей системы, включая процедурные следствия ограничений существующей системы и их обхода! Иногда это называют «деградирующей практикой».

Проблема, возникающая вместе с объектными библиотеками, состоит в том, что объекты очень хорошо инкапсулируют (скрывают) свою реализацию. Это позволяет нам использовать иерархии классов, в действительности ничего не зная о их внутренностях. Поэтому приходится продираться через несколько слоев классов, каждый из которых (возьмем пример из Географических Информационных Систем) имеет дело с координатной системой просто для того, чтобы поставить значок на карте. Никаких проблем не возникает до тех пор, пока мы впервые не попытаемся поставить несколько сот значков и не обнаружим, что «простое присваивание», которым мы так восхищались, требует столько вычислительной мощности, что для перерисовки экрана нужно полчаса.

Проект, который регулярно не проверяет свою иерархию классов, чтобы гарантировать, что внутренние представления естественны и стандартны и что дизайн соответствует встречающимся на практике ситуациям, может внезапно оказаться по горло в воде из-за скрытой цены повторного использования кода.

Как всегда, самое лучшее оружие от распухания — это концептуальная целостность, достигаемая сравнением мысленных моделей проекта.

❒ Бесконечная деградация «программных архитектур»

Была команда, которую заказчик попросил создать среду реального времени, которая могла бы поддерживать небольшие процессы, стыкуемые через средства ввода/вывода, что-то типа графической оболочки для управления сложной системой трубопроводов. Если классифицировать эту задачу, это работа по системному программированию. Эта команда решила быть Профессиональной, и стала использовать Надлежащие Методы. Поэтому они пошли и сделали объектно-ориентированную модель этих вещей из Требований Пользователя используя инструменты, автоматизирующие подход Шлера-Меллора *(Shlear-Mellor)*. Но была небольшая трудность, поскольку проблема была из другой области, зато они подогнали ее под Надлежащую Методологию. Они ведь могли использовать генератор кода! Они нажали кнопку и получили целую кучу склеенных классов. Каждый из них просто вызывал процедуру из API из «Программной Архитектуры», слоя, который требовался в подходе, что позволяло приложению работать на определенном типе компьютерной системы. В этом случае Программная Архитектура стала бы системным средством предоставления графического интерфейса для оболочки, работающей со сложной системой трубопроводов!

Как картостроители, мы можем видеть, что произошло с этим образчиком тупого Профессионализма и Следования Процедуре паковщиков. Команда извлекла не тот пакет знаний и использовала хорошо продуманный и замечательно автоматизированный подход и язык в совершенно неправильном контексте. Подход Шлера-Меллора предназначен для задания функционирования прикладных программ, а не для системного программирования. Говоря простым языком, они не смогли увидеть, что классификация проблемы была проведена неправильно,

поскольку у них не было здравого смысла, что есть результат недостатка морали или врожденного идиотизма. Мы предпочли бы сказать, что они были принуждены думать, что им нельзя думать непосредственно о работе, но сначала все смести и найти костыль. Потом они могли видеть проблему только через подход, даже если он плохо соответствовал. Затем подход уровня приложений представил их проблему на прикладном языке. Поэтому им в голову не приходило подумать, какого сорта программу они пишут, и заметить, что это системная оболочка, а не прикладная программа. Для паковщиков Профессионализм всегда означает взять зубную щетку и палочки для еды в руку и не замечать, что зубная щетка попала в нос, хотя все в полном соответствии с церемонией!

По мере того, как все становилось все более абсурдным, росло напряжение и команда все настойчивее обороняла свой Профессионализм. Один раз на встрече некто, ожидая хотя бы немного прагматизма, представился как «программист компьютеров» *(computer programmer)*. Результат был замечательный. Вся команда глядя в пол бубнила под нос «Программный Инженер... Компьютерный Ученый...» (Software Engineer... Computer Scientist...), как будто тот совершил значительную социальную ошибку. Два члена команды позднее объясняли, что они не нашли создание «просто кода» интересным, и их Профессиональный Интерес заключался в Применении Процесса.

Очень тяжело заниматься системным программированием применяя некие лежащие перед тобой инструкции к компьютеру. Это ведет к стрессу, поскольку невозможно. Поэтому очень важно многословно демонстрировать свой Профессионализм руководству, чтобы никто не смог отделить зерна от плевел в этом вечном хаосе.

Вы не сможете писать компьютерные программы, если ваша стратегия состоит в замысловатой игре по передаче пакетов. Даже если придерживаться характерной стратегии паковщиков передавать документ соседу слева «для рецензии». Менеджер проекта, который пишет описание задачи, показывая входы и выходы задачи, часто может держать такие вещи под контролем, но только в том случае, если действия соответствуют атомам познания проекта и хорошо разбиты на части и при условии, что выходы на самом деле ближе к обработчику, либо непосредственно, либо путем информирования другой стороны, чем входы.

Картостроители тоже могут извлечь интересный урок из подобных историй. Нужно быть предельно осторожным, включаясь в игры функциональной передачи пакетов. Сложность никуда не исчезает, и функциональность не появляется из ничего. Сложность может быть выражена более просто, если копнуть глубже. Ты узнаешь, когда ты этого добился. Ее можно также избежать, удалив часть сложности. Ты также узнаешь, когда ты этого добился. Когда показалось, что огромная глыба неуклюжести куда-то пропала, вероятно ты просто переместил ее куда-то в другое место своей структуры. И это хорошо, поскольку когда ты обнаружишь ее вновь, то ты получишь две точки зрения на ту же проблему, а это может привести к большим открытиям. Но не питайте привязанность к какому-то решению, которое создает видимость, что вся сложность магически исчезает — она просто неожиданно проявится в другом месте и испортит весь день. Это проявляется на любом уровне, начиная с выявления требований и заканчивая отладкой. Ошибки никуда не деваются. Те ошибки, которые появились, а потом неожиданно исчезли сами собой — самые коварные ошибки. Возьми старую реализацию, найди их и убедись, что таких же ошибок нет в последней реализации.

В качестве практического примера того, что функциональность не появляется по мановению волшебной палочки, рассмотрим проблему дробления *(atomicity)*. В многозадачных системах многие процессы исполняются без возможности управления временем приостановки, чтобы дать шанс выполниться другим процессам. Иногда двум процессам требуется координация, например для управления периферийными устройствами. Проблема в том, что один из процессов может обнаружить, что периферийное устройство свободно и сделать пометку о его захвате. Но в промежутке между этими двумя событиями второй процесс тоже может сделать проверку и обнаружить, что устройство свободно и тоже начнет делать пометку о захвате. Один процесс просто перепишет пометку другого и оба будут пытаться получить доступ

к периферийному устройству одновременно. Не имеет значения, как это организуется в пользовательских процессах, невозможно совместить проверку и установку как единую, «атомарную» операцию, независимо от заумности пользовательского процесса. Ее можно инкапсулировать в функцию `GetLock()`, но она все равно должна добыть атомарность из операционной системы. Если процессор поддерживает многозадачность на уровне железа, как большинство современных процессоров, нам понадобится атомарная операция, содержащаяся в наборе инструкций, такая как инструкция TAS в процессорах серии 68000 фирмы Motorola.

Ничего этого нельзя увидеть предполагая, что нельзя переходить на использование расслоения или применять специализированные языки. Действительно, если некто полагается на специфический код операции просто для захвата принтера, то, по-большому счету, необходимо разбиение на слои! Это просто означает, что для достижения максимального упрощения мы используем оптимальный уровень специализации — мы не делегируем невозможное призрачным ящикам и не основываем дизайн на ложных предпосылках.

❒ Аудит качества

Для картостроителя процесс — это протокол взаимодействия со своими коллегами во времени и пространстве, а не предварительно написанный свод правил, исполнение которых контролируется. Он предоставляет средства, а не требует тупого подчинения. Чтобы отчетливо подчеркнуть различие, нам следует в правильном духе рассмотреть аудит качества.

В обычной модели паковщика аудит — это суровое испытание. По мере приближения аудита менеджеры работают с мрачным предчувствием, работники стремятся сбежать в отпуск или в командировку, либо прикидываются больными, лишь бы избежать аудита. Когда аудиторы набрасываются как коршуны, они смотрят на членов команды как на врагов, и принуждают их к подразумеваемому подтверждению совершенства процесса и того, что все недостатки, которые они обнаружат, — нарушения, совершенные людьми. Отдельные работники становятся мальчиками для битья, несущими наказание за ошибки организации, которые они не контролируют. Это классическая ситуация, заставляющая людей глотать успокоительное перед приходом на работу. Нет сомнения, что это модель, не смотря на то, что ее редко так характеризуют, поскольку стандартное для менеджеров «Краткое руководство для погонщика» *(Bogeyman Briefing)* включает советы типа: «Не выдавай информацию добровольно. Коротко формулируй. Если тебя спрашивают, где план управления проектом, скажи, что в Хранилище *(Registry)*». Адвокат должен давать похожие советы клиенту, подвергаемому перекрестному допросу!

В ISO 9001 нет абсолютно ничего, что требовало бы оправдать ритуал такого вот пути совершенствования. У нас вообще нет проблем с ISO 9001. В действительности мы говорим, что разговор о «следовании ISO 9001» в то время, когда мы даже не способны применять с положительной мотивацией сам ISO 9001, превращает ту же самую старую глупость в еще один «радикально новый прорыв в науке менеджмента». Ну, а что думает об аудите качества команда счастливых картостроителей?

Во-первых, объектом пристального внимания однозначно должен стать процесс сам по себе. Мы должны оказать любезность работникам, предполагая, что они делают свое дело добросовестно, поскольку это всегда так, даже когда их держат за полных идиотов. Мы, таким образом, предполагаем, что любая проблема, по умолчанию, — это системный просчет процесса. Нехорошо сваливать повторяющиеся крушения самолетов на «ошибки пилота» — очевидно, что нуждается в перепроектировании эргономика кабины.

Во-вторых, выполняемые аудиторами сравнения, должны проводится между возможностями процесса и тем, что требует от пользователей бизнес. Одна из худших вещей, возникающая от враждебного аудита, — это то, что разногласия при определенных обстоятельствах могут легко вылиться в оргвыводы. Например, большинство процессов содержат пункт, говорящий, что в записях у кадровиков должны делаться отметки о прохождении обучения. Это вполне

соответствует полуквалифицированной работе, когда рабочие могут получать «билетики», позволяющие им выполнять специфические задачи. В транспортном отделе, часто требуется получать сертификаты из-за законов, и там проблема немного другая. В программирующей команде формальные курсы обучения дают очень мало по сравнению с навыком, получаемым в команде, поскольку почти никогда не имеют отношения к делу. Повышение квалификации происходит либо на работе, либо за счет свободного времени работника. Многие программисты проводят по нескольку сотен часов в год за самообучением дома. Авторы основанных на «или иначе» процессах должны помнить, что наниматель даже не смеет потребовать от работника разглашения, что же он изучает дома, поэтому не стоит грубить ему, уходя от вопроса, поскольку менеджерам проектов действительно нужна эта информация и им приходится спрашивать вежливо. Так что проверка бессловесной покорности глобальному механизму бесполезна по сути и приведет только к спорам об интерпретации. Вместо этого аудитор должен оценить локальные потребности и удостовериться в применимости процесса в свете этих потребностей.

В-третьих, аудитора нужно рассматривать как коллегу по бизнесу со своим положительным вкладом в работу. Эти люди видят сотни потребностей бизнеса и картотек, автоматизированных и ручных. Если бы глупейшая война между проверяющим и проверяемыми смогла превратиться в совместную критику процесса, проверяемые получили бы возможность открыто говорить о своих проблемах, вместо игры в молчанку, как это советует большинство менеджеров. Только тогда, когда они знают, в чем заключаются настоящие проблемы, аудиторы смогут покопаться в обширном опыте и предложить решения, которые, как они видели, срабатывали у других.

Аудиторам качества не следует быть учетчиками зерна, единственный положительный вклад которых заключается во внедрении изысканных ритуалов для компенсации неестественной сложности процесса. Их роль должна идти гораздо дальше. Роберт Хайнлайн сказал, что цивилизация строится на библиотечной науке, а аудиторы качества это современные библиотечные ученые инженерной индустрии. Они могут сказать нам, как дешевле всего сохранить данные так, чтобы мы смогли найти их позднее, при условии, что мы знаем что ищем.

Глава 5

Принципы разработки

❒ Простая и надежная среда

Средства разработки состоят из всех тех инструментов (включая текстовые процессоры), которые используют программисты, а так же машин и сетевой инфраструктуры, на которой они работают. Хорошие средства разработки должны быть как можно проще. Как и в программах, которые вы разрабатываете: чем больше сложности, тем больше места для проблем, тем выше стоимость эксплуатации.

Хорошее общее правило — держать всю свою работу, включая средства конфигурации (в файлах скриптов, если нужно) в виде простого текста. Имейте возможность при необходимости стереть все, кроме исходного текста, и автоматически перекомпилировать.

Самая главная задача репозитария и системы управления конфигурацией — обеспечить вашу безопасность. Каждый добавляемый элемент сложности увеличивает опасность падения системы. Команда, перекладывающая контроль на систему управления конфигурацией с клиент-серверной архитектурой, рискует потерять проект из-за потери ссылочной целостности в системе. Получающийся хаос, когда вся работа останавливается, понуждает найти способ обеспечения безопасности в создании резервной копии, люди начинают выяснять, что необходимо переделать заново, и дух падает. Не так сложно построить хорошую систему управления конфигурацией на основе скриптов с использованием списков рассылки, на основе чего-то простого, типа SCCS или RCS, для обеспечения контроля версий.

Та же самая логика, что целью является безопасность, и поэтому простота увеличивает надежность и безопасность, когда система и пользователи в стрессе, применима и к созданию резервных копий. Когда что-то не так, главное — это знать, как откатить систему в нормальное состояние, в котором она была некоторое время назад. Инкрементное резервирование с гибкой стратегией отслеживания изменений дерева каталогов может уменьшить доверие к тому, что все работает нормально, даже когда это так. Или, скорее, уменьшит. Команда, перешедшая к простому тексту, записывающая все на ленту и меняющая ленту каждую ночь, знает, где все это находится, и на этом надежном фундаменте способная достигать прогресса в выполнении настоящей работы, действительно умнее, чем любящая сложности команда, проводящая месяц только в попытках организовать работу.

Делай проще. Никогда не латай ничего — ты никогда не знаешь, нашел ли ты все проблемы. Сотри и загрузи заново. Всегда будь способен переформатировать свой диск, переинсталлировать свои инструменты, восстановить тексты из репозитария, переконфигурировать и перекомпилировать. Это дает полную безопасность и сохраняет все то время, которое ушло бы на суету по поводу вирусов. Кого это волнует?

❒ Типы систем

Один из самых важных вопросов, который нужно задать о новой системе, или даже о старой системе, с которой пришлось столкнуться, — какого вида эта система? Никто не станет даже пытаться расположить колонию хиппи вокруг плаца или военный лагерь в виде хаотически расположенных вигвамов, соединенных тропинками! Любая система может обладать более чем одним признаком из перечисленных ниже, хотя некоторые взаимно исключают друг друга. Эти, вероятно полезные, признаки — грубые категории, встречающиеся на практике. Они не выведены из какой-то лежащей в основе теории. Примеры типов систем такие:

Монолитная. Централизованная обработка, а также оффлайновые соединения с пользователями либо примитивные терминалы — вот что такое монолит. Пользователи находятся либо в том же здании либо подключены по постоянной выделенной линии. Это прекрасный способ проделать обработку громадного количества коммерческих данных, когда к системе подключены промышленные средства ввода документов, огромные хранилища данных на дисках и лазерные принтеры, выплевывающие страницы на несколько футов в воздух. У монолитов свои проблемы — пользователи, например, часто лишены возможности доступа к резервной копии! Но это сглаживается высокой степенью контроля над системой. Устойчивое к сбоям оборудование (включающее RAID и резервирование питания) и сложные технологии серверов баз данных — вот где имеют преимущество такие системы.

Клиент-Сервер. Распределяет обработку среди пользователей для работы, которая должна быть сосредоточена в одном месте. Предоставляет хорошее разбиение на уровни, разделяя хранение, обработку и взаимодействие с пользователем[1]. Допускает локальную специализацию по функциям и нескольких поставщиков, и, следовательно, сохраняет инвестиции при последующем расширении. Требует (и, следовательно, использует) более сложную сеть, чем монолит, но лучше приспособлена к интерактивной работе, давая максимально возможному числу пользователей выполнять обработку локально, используя средства, наиболее соответствующие потребностям пользователей. Все клиент-серверные архитектуры внутри себя прячут монолит.

Интерактивная. Позволяет пользователю вступать в диалог с системой. Незаменима, когда требуется быстрый отклик, либо при работе с широкой публикой. В наше время обычно построены на графическом интерфейсе. Состояние системы постоянно изменяется, поэтому требуется журналирование, либо периодически будет возникать риск потери данных. Проблема распределения нагрузки — важный момент, поскольку число пользователей меняется, и нагрузка регулярно скачет от нормальной до пиковой.

Пакетная. Хотя их часто считают устаревшей стратегией с использованием перфокарт, пакетные системы просты, надежны, великолепно себя ведут при неустойчивой связи и масштабируются лучше, чем интерактивные системы.

Управляемая событиями. Реагирует на события из внешнего мира. Приложения с графическим интерфейсом большую часть времени находятся в ожидании, пока пользователь что-нибудь нажмет, чтобы отреагировать на событие. «Однорукие бандиты» (игровые автоматы) — тоже управляемые событиями системы, как и сигнализация. Управляемые событиями системы имеют сложное пространство состояний и большую чувствительность к вводимым данным. Часто для них ограничивается время ответа на событие. Если проблему можно представить как не управляемую событиями систему, то лучше так и сделать.

[1] HCI — Human-Computer Interface, человеко-машинный интерфейс

Управляемая данными. Аналогична управляемой событиями и пакетной, но с ясными потоками данных через каждую подсистему, где наличие входных данных — это событие-триггер, заставляющее каждую подсистему выполнять свой цикл обработки. Управляемые данными системы гибче пакетных, поскольку размер пакета может изменяться динамически, но они имеют надежность пакетных систем, поскольку мы всегда можем узнать, как далеко мы зашли в обработке каждого пакета. Каждая подсистема может организовывать элементарную операцию, выдающую информацию и удаляющую входные данные, так что даже при восстановлении после сбоя питания система сразу готова к работе, поскольку состояние системы всегда устойчиво. Системы электронной почты — пример систем, управляемых данными.

Оппортунистическая (рассчитывающая на благоприятную возможность). Этот тип систем никогда не пострадает от ошибок при передаче, поскольку они используют каналы только тогда, когда это возможно. В действительности, большинство офисов оппортунистические, поскольку они построены на основе Ethernet. Данные хранятся в буфере до тех пор, пока локальный передатчик не передаст их без конфликтов *(collision)*.

Штурманская (Dead reckoning). Пытается проследить каждый шаг пользователя. Часто представляется желательной, поскольку позволяет обеспечить сильную проверку данных пользователя, но может оказаться очень хрупкой, что приводит к шуткам типа знаменитой: «Бесполезно нажимать на это, компьютер говорит, что этого тут нет!»

Сходящаяся в одну точку (Convergent). Тут не интересно отслеживать каждый шаг происходящего в реальном мире процесса, здесь внимание направлено на регистрацию изменений состояний в ключевых точках и интегрировании данных в виде аккуратной картины реального мира в некоторый момент в прошлом, и прогрессивно ухудшающейся аппроксимацией по мере приближения к настоящему. Мобильные пользователи, которые закачивают данные со своих ноутбуков в корпоративную сеть — хороший пример. Мы очень точно знаем, как много мы продали на прошлой неделе, очень приблизительно знаем о вчерашнем дне, но мы не получили еще данные от Джона и Джилла, поэтому не знаем почти ничего о сегодняшнем дне.

На гребне волны (Wavefront). Системы, работающие по мере появления событий. Управление освещением или телефонная коммутация — вот примеры. При сбоях мы больше заинтересованы в быстром восстановлении работы, чем обеспокоены потерей данных.

Ретроспективная (Retrospective). Связанная с поддержанием аккуратной записи прошлого. Избежание потерь данных обычно очень важно. Пример — бухгалтерские системы.

❒ Обработка ошибок — лимфатическая система программы

Часто говорят, что следует перехватывать сообщения об ошибках, но в этом мало пользы, если неизвестно, что же с ними собираются делать. Обработка ошибок составляет такую же часть структуры программы, как и логика обработки нормальных ситуаций, но не настолько же почитаема. Эта связь скорее похожа на взаимодействие кровеносной и лимфатической систем в теле. Вам необходимо предусматривать обработку ошибок на каждой стадии проектирования. Например, нет никакого смысла регистрировать сообщения об ошибках записи в журнал ошибок в процедуре обработки ошибок!

Концептуальная целостность требует, чтобы вы определили общий подход к обработке ошибок, используемый по всему проекту. Как вы будете сообщать об ошибках? Какие идиомы программисты будут использовать для элегантной проверки ошибок без нарушения главного

потока управления? Совершенно необязательно делать второй вызов, чтобы проверить, возникла ли ошибка или что там было — иначе код будет распухать. В идеале ошибки должны проверяться при выходе из функции, чтобы использовать краткие идиомы, что приводит к получению понятного кода и, как следствие, плато качества:

```
if((fp = fopen(...)) == NULL)
{
    // Error
}
```

или

```
if(!DoTheBusiness())
{
    // Error
}
```

Можно слышать множество жалоб на распухание логики обработки ошибок до состояния, когда строго написанный вызов функции может занимать столько строк, что уже невозможно увидеть всю картину в целом. Как картостроители, мы знаем, что обрастание таким количеством Достойных стандартов кодирования, что невозможно написать замечательную программу — ошибочный путь.

В процедурном (и, в некоторой степени, объектном) подходах к коду ведется дискуссия по поводу стратегии обработки ошибок, где их следует обрабатывать. В дискуссии рассматриваются два подхода. Первый говорит, что вызываемая подпрограмма не должна возвращать управление, пока она не выполнила то, что ее просили, и это можно назвать «полным делегированием». Другой говорит, что вызываемая подпрограмма должна выполняться, пока не столкнется с проблемой, и тогда аккуратно возвращать наверх весь мусор, который она получила и сообщать вызывавшему, что произошла ошибка — это мы назовем «ложная готовность».

Привлекательность **полного делегирования** состоит в том, что при этом получается очень чистый код с вызывающей стороны, и он может быть очень эффективным, и он препоручает ответственность за поддержание состояния более низких уровней самим уровням. Недостаток заключается в том, что он работает только при условии, что вызывающей стороне на самом деле не требуется обработка последствий ошибки в ее собственном контексте. Это ограничивает его системами для типовых ситуаций, где приложение действительно может не знать о том, что происходит на нижних уровнях, и если нижний уровень реально не сможет устранить проблему, то попытка возврата будет недопустима, поскольку приведет либо к зависанию процесса либо к фатальной ошибке.

Ложная готовность всегда позволяет вызывающей стороне отреагировать на проблемы, и вложенные вызовы смогут откатить стек, пока не будет достигнут уровень, способный разобраться с проблемой. Логика обработки ошибок пронизывает каждый уровень, но может быть минимизирована аккуратным кодированием, если автор и сопровождающий знают, как работать в этой схеме. Кроме того, можно обеспечить трассировку вызовов, показывающую, как процесс напоролся на проблему, так что всегда можно воспроизвести ситуацию.

Мы не думаем, что нужно дискутировать на эту тему, поскольку когда полное делегирование применимо, оно оказывается на самом нижнем уровне. Смешивание этих подходов приводит к кошмару в коде, поскольку нарушает концептуальную целостность.

Некоторые объектные языки предоставляют исключения, которые позволяют автоматически схлопывать стек до уровня, ответственного за обработку данной ошибки. Это замечательный способ освободить основную логику от деталей обработки ошибок. Важный момент, который нужно помнить, заключается в том, что иногда исключение может быть передано гораздо выше, если ты хочешь не просто обработать его, а хочешь знать, из-за чего оно произошло. Сообщение об ошибке с нижнего уровня, говорящее:

`Could not write() datafile ftell() = 246810,`

если за ним не следует другое, говорящее:

`Could not Save World`

... при отладке просто бесполезно. Ты можешь передавать исключения на более высокий уровень без нарушения главной логики управления, и следует подумать, как это сделать.

Не злоупотребляй в рабочее время исключениями для создания замысловатого потока управления. В особенности не скрывай `longjmp()` в макросах и не вызывай их из обработчиков. Если ты желаешь поэкспериментировать с Силами Тьмы, делай это дома. Нам всем приходится это делать, но что печальнее всего, и наши коллеги могут ухватиться за неправильную идею и начнут ее рационализировать. Не кажется ли странным, что мы производим сегодня языки, которые страдают запорами по этому поводу, когда заняло годы просто правильно переопределить с помощью `const` описания прототипов функций, но при этом нам позволено фокусничать с потоком управления так, как мы не пытались делать даже на ассемблере?

По возможности избегайте расставлять захламляющие ваш код `assert()` и условную компиляцию отладочных макросов. Вам не достичь необходимой достаточности плато качества, если все вокруг завалено мусором.

❒ Увлечение формой (а не содержанием) и комбинаторный взрыв

По некоторым причинам существует мнение, что для того, чтобы системы были робастными (устойчивыми к ошибкам), им требуются нормальные режимы, режимы сбоя, в которые они попадают при сбое, и режимы восстановления, в которые они переходят после попадания в режим сбоя для возврата в нормальный режим. Частично это провоцируется потерявшими ориентировку пользователями, которые пытаются описать цели в случае сбоя, но делают это рассуждая о «режимах» системы. Это деликатная область, поскольку при обсуждении сбоя пользователи должны думать о составляющих реальной системы, которые могут давать сбой, и они должны обсуждать сбои заранее, раз они вынуждены подписывать Требования Пользователя, которые потом могут быть использованы как палка, которой их будут бить. Это значит, что они должны пытаться изучить финальную реализацию лучше, чем ее знают сами разработчики, чтобы суметь описать, что нужно делать при сбое компонентов.

Подчеркивая важность диалога, необходимо также отметить часто упускаемый момент. Действительно ли пользователь хочет, чтобы вы реализовали режим сбоя, детально описанный в Требованиях Пользователя? Может будет достаточно системы, которая просто работает? Конечно, скорее всего так и есть, но многие команды сломя голову бегут и реализуют эти сбои, как и сказано в Требованиях Пользователя.

Современная легенда в ICL гласит, что когда они покупали первую партию плат от Fujitsu, то сделали оценку, что надежность будет составлять 1% отказов. Поэтому прямо перед отправкой первой сотни один из директоров Fujitsu взял сверху из ящика плату и, перед тем как положить ее обратно, стукнул по ней молотком.

Помимо необходимости управлять переключениями состояний и исполнением редко когда нужного кода, в системах такого рода есть более глубокая проблема.

Сначала мы находимся в нормальном режиме. Затем попадаем в режим сбоя. Затем в режим восстановления. Что случится, если опять произойдет сбой? Что, у нас приключился сбой во время восстановления из режима сбоя? Восстановления из сбоя во время восстановления из сбоя? Тут очень легко появляется необходимость бесконечного расползания системы режимов, а не просто распознавание сбоев. Конечно, если дизайн всех уровней одинаков, то ничего страшного — вам остается лишь доказать, что это именно тот случай.

Если вы смогли остановить бесконечное расползание, то, вероятно, сделали и следующий шаг — устранили нормальный режим и режим восстановления и оставили только режим сбоя!

(Или устранили нормальный режим и режим сбоя и оставили режим восстановления, если вам так больше нравится.) При этом отпадает необходимость управлении скоординированными переключениями на многих платформах в моменты, когда гремлины шевелят контакты в сетях питания. Системе даже не нужно знать, что она находится под непрерывной атакой реального мира, и что это уже четвертый раз, как она пытается обработать пучок транзакций. При этом, если вы достаточно аккуратно определили Правильные Вещи, для выполнения Правильных Вещей не нужно знать их контекста.

Наличие множества режимов для обработки сбоев на самом деле гораздо менее нужно, чем думают большинство людей, а избавление от них очень сильно увеличивает управляемость сложностью. Если мы желаем сохранить контроль и понимание наших проектных решений, мы должны минимизировать сложность всего, что мы можем. На стороне победителя в этом уравнении находится плато качества. На стороне проигравшего — взаимодействие одной сложности с другой сложностью, дающее невообразимый рост пространства состояний системы, называемый «**комбинаторным взрывом**».

❒ Избегайте избыточности представления

Каждый проектировщик баз данных знает о нормальных формах. Дело становится очень сложным, если пытаться проводить полный анализ предмета в условиях реального мира, но базовая идея очень проста. Избегайте избыточности в представлении. Если вам понадобилась запись заказа и запись счета, каждая из которых требует названия заказчика, храните запись о заказчике в одной таблице, и используйте уникальную ссылку на таблицу заказчиков в записи заказа. Затем вставьте ссылку на таблицу заказов в записи счета. Тогда вещи никогда не встанут наперекосяк, когда вам придется не забывать удостовериться в загрузке разных мелочей каждый раз, когда вы хотите изменить данные.

Дело в том, что концепция нормализации базы данных из тех же соображений применима везде. Никогда не храните одну вещь и, отдельно, еще одну неподсоединенную вещь, которая полагается на то, что первая существует. Пусть данные управляют своей собственной структурой, и не появится никаких перекосов. Ритуальное использование структур данных часто включает претензию на управление держа зубную щетку и палочки для еды в одной руке. Если мы создаем структуру финансовых отчетов в Ящике A, а сложное описание Ящика A в Ящике B, то мы можем провести много времени копаясь в Ящике B, и ни разу не обратим внимания на тот факт, что на самом деле мы вообще не понимаем того, что происходит в Ящике A!

Не попадайте в эту ловушку. Пусть данные представляют сами себя, или, как Лори Андерсон *(Laurie Anderson)* сказал в книге «Большая наука» *(Big Science)*:

```
Let X = X
```

❒ Посмотри на состояние всего этого!

Точно также как важно избегать избыточности представления данных в контексте вашей системы, важно также избегать избыточности представления данных вашей системы в контексте платформы. Это истинно, поскольку из-за сбоев глобальные ресурсы могут оказаться в непредсказуемых состояниях. Проект всегда должен предусматривать освобождение всех системных ресурсов, особенно частично записанных файлов, которые съедают пространство диска, даже если они не нарушают работы системы.

Осознавайте, какие системные ресурсы освобождают себя сами (такие, как семафоры), когда процесс-владелец погибает, и предпочитайте их.

Избегайте «процессов очистки», которые срабатывают самопроизвольно по системным часам и захватывают права на все ресурсы вашей системы. Пытайтесь использовать протоколы

инициализации, которые начинаются с определения известного состояния и только потом двигаются вперед. Вот это может стать примером:

1. Найти входной файл.
2. Если выходной файл уже существует, удалить входной файл и завершить работу.
3. Открыть входной файл.
4. Открыть временный выходной файл со стандартным именем.
5. Направить результат обработки входного файла в выходной файл.
6. По окончании записи изменить атомарной операцией имя временного файла на имя выходного файла.
7. Удалить входной файл.

Или, крепко возьмитесь левой рукой, прежде чем отпустить правую!

❒ Реальность системы как объекта

Этот раздел прежде всего обращен к проектировщикам объектных систем, поскольку проблема, которой он касается, прежде всего проявляется в объектном подходе. Это происходит из-за строгой инкапсуляции, которую предоставляет объектная модель. Мы уже обсуждали два подхода к проектированию объектных систем, которые предпочитают картостроители и паковщики. Подход картостроителя включает понимание природы желания, а затем в процессе итераций выявляется адекватная динамика системы и создается оптимальное взаимное соответствие (карта) между динамикой проблемы и семантикой системы.

Объектные проекты нацелены на создание формализованного Хода Конем («вилки») с использованием подхода, который явно связывает объекты реального мира с жизнеспособной семантикой системы посредством объектных языков программирования (будь это Eiffel или генератор кода UML). При разработке этих проектов их создатели стремятся представить все, что есть сейчас в реальном мире, а не будет завтра, когда система будет использоваться. Главное отличие в том, что завтра система будет существовать в мире пользователя — сегодня это не так. Поэтому аналитики регулярно создают маленькие картинки мира пользователя в будущем, которые содержат все, кроме самой компьютерной системы, которая является центром всего сценария.

Тем не менее, внутренний проект системы также испытывает затруднения из-за недостаточного представления самой системы. Кто-то мог бы сказать, что система реального мира и внутренняя система — это одно и то же как в реальном, так и в абстрактном мирах, и, следовательно, эта идентичность в объектных проектах формирует появление «Хода Конем» в наиболее фундаментальной форме. Вот два глубоких вопроса, возникающих при поиске объектов и попытках найти их взаимосвязи:

1. Кто кого порождает?
2. Кто чьи методы использует?

Имея в проекте чистый класс System (Система), гораздо легче построить иерархию наследования и увидеть, откуда приходят такие вещи как GUI и ввод/вывод на ленту, не говоря уже о событиях, запускаемых по таймеру! Это не означает, что на более поздней стадии проектирования функциональность нельзя поместить в специализированные классы, но в проекте

это дает реальности мира пользователя равный вес с реальностью системы, поэтому результат будет удовлетворять обоим критериям.

Битва с абстрактным набором беспорядочно летающих вокруг классов без способа сверки с реальностью так же безнадежна, как и любая другая деятельность, если вы не знаете, что же вы делаете.

Конечно, потребность в классе System (Система) исчезает, если кому-то нужно просто моделирование, а не автоматизация бизнес-процессов, в которых не представлены управляющие системы. Каким мог бы быть подход в таком случае? Здесь мы напираем на то, что проектирование, формируемое стратегией картостроения, включает в себя больше, чем просто набор процедурных действий. Это означает очерчивание проблемы, прояснение желаний, нахождение точки оптимального приложения сил между динамикой проблемы и семантикой системы. Если ваш проект не получает преимуществ от класса System (Система), не используйте его!

❒ Детекторы утечки памяти

На рынке имеется ряд продуктов, которые на основе различных стратегий обнаруживают утечки памяти в приложениях. Утечка памяти — это то, что случается, когда программа запрашивает блок памяти из кучи (используя, например, `malloc()` в Си под UNIX или DOS , или оператор `new` в C++), а затем забывает вернуть его обратно по своему завершению. Это иногда приводит к нарушению работы других процессов, работающих на той же платформе, поскольку некоторые операционные системы позволяют одному процессу захватить всю имеющуюся память системы и файл подкачки.

Даже если операционная система достаточно разумна, чтобы ограничить выделяемую отдельному процессу память, приложение может вскоре исчерпать свою квоту, что обычно заканчивается сбоем с точки зрения пользователя.

Поэтому утечки памяти — это Плохая Вещь.

Это причина того, почему люди продают, и покупают, детекторы утечки памяти. Неприятность в том, что утечки — это симптом проблемы, а не причина проблемы. Не так трудно вызвать `free()` или удалить объекты, которые больше не нужны. Применение `delete` к коллекции указателей на активные объекты — вот опасное занятие, как и перезаписывание их адресов. Что если эти объекты, скажем, зарегистрировали функции обратного вызова *(callbacks)* в GUI? Как же их освободить? Деструкторы вне контроля — это программист вне контроля. Если программист не может продемонстрировать контроль над объектами, которого достаточно для избежания утечек памяти, то как мы можем думать, что правильно еще хоть что-нибудь?

Концептуальная целостность — это одна из самых сильных подпорок в сохранении контроля над объектами. Полезное общее правило (хотя, как и все правила, не всегда применимое) должно говорить, что уровень, который конструирует модуль, должен отвечать за его уничтожение. Это, по крайней мере, фокусирует внимание на жизненном цикле объектов, а не на всего лишь нескольких аспектах их поведения, которые можно проследить на диаграммах использования.

❒ Таймауты

Один из наиболее эффективных способов получения «затравки» для генератора случайных чисел — посмотреть на системные часы. Аналогично, если два процесса работают на одном и том же процессоре, то мы никогда не сможем предсказать, сколько времени пройдет от запуска программы до достижения заданной точки в программе. Мы не можем даже точно предсказать, сколько процессорного времени будет выделено каждому процессу.

Таким образом, таймауты — это Плохая Вещь. Есть мнение, что они значительно увеличивают пространство состояний системы, что делает поведение системы гораздо хуже предсказуемым для проектировщика. При отладке они могут привести к условиям, при которых невоз-

можно будет воспроизвести сбой. Не используйте их, кроме тех случаев, когда они абсолютно необходимы.

Коммуникационные уровни часто обязаны использовать таймауты, поскольку когда приходит время их работы, единственный способ обнаружить, что удаленное устройство готово к взаимодействию — послать ему сообщение и дождаться, посылает ли оно ответ. Как долго нужно ждать? «Проблема Византийских полководцев» *(Byzantine Generals' Problem)* это иллюстрирует. Поэтому в большинстве современных систем есть таймауты на уровне коммуникации, но нет никакого оправдания использовать их повсюду, и там, где они должны использоваться, они должны быть сокрыты внутри инкапсулирующего объекта, который для целей отладки может быть заменен на детерминированный генератор событий (например, нажатие клавиши).

❒ Проектируй для тестирования

Редко бывает достаточно, что наши системы работают. Обычно нам также нужно знать, что они работают хорошо. Это положение может показаться тривиальным, но из него есть следствия для того, как нам организовать свою работу.

На уровне выявления требований мы можем обследовать определенный участок проблемной области, который, как мы предполагаем, может стать источником проблем, даже не зная, собрали ли мы всю относящуюся к проблеме информацию. Для повышения уверенности, что мы ничего не упустили, нам нужно пристально посмотреть пошире, чтобы увидеть, куда что уходит, и откуда что приходит. Мы должны найти способ представления этих потоков, чтобы можно было охватить одним взглядом большую картину. Стопки прозы или толстые папки Диаграмм Потоков Данных здесь совсем не помогут (хотя они могут пригодиться где-то в другом месте проекта), поскольку они не позволят нам увидеть одним взглядом, что нет потерянных концов. Если мы можем убедиться, что нет потерянных концов, то мы с основанием можем быть уверены, что здесь нет скрытых ужасов, которые мы обнаружим во время реализации. Это пример технологии картостроителя для очерчивания проблемы.

На архитектурном и детальном уровнях проектирования применяется та же идея. Рассматривая наш проект, мы представляем себе наши идеи как можно большим числом способов и проверяем, сможем ли мы их разрушить. Это важно, что мы используем некоторые черты проекта, такие как число возможных состояний на входе, чтобы показать, что система, которую мы проектируем, будет устойчива во всех случаях, показав, что мы рассмотрели все случаи. Это не означает, что мы пытаемся просмотреть все варианты — вместо этого мы находим средства сгруппировать эти варианты, и показать, что мы рассмотрели все группы.

При пошаговой отладке с помощью символьного графического отладчика в каждой точке принятия решения нам следует рассмотреть все обстоятельства, при которых принят путь, который мы выбираем, и также следует проследить все другие варианты путей.

Во всех этих ситуациях проектирование для тестирования начинается с разбиения нашей работы на уровни так, чтобы ее корректность была доступна для проверки. В свете этого интересно рассмотреть, что мы подразумеваем под математическим доказательством. Предназначение доказательства обычно описывают как то, что показывает, что утверждение имеет место. Это действие-центрический взгляд на вещи, присущий паковщикам. Описание предназначения доказательства с точки зрения картостроителя состоит в том, что оно показывает нам утверждение в новом свете, в котором истинность утверждения очевидна для проверки. Для картостроителей доказательство не просто устанавливает факт, оно так же увеличивает наше понимание. Недавно мы видели полученные с помощью компьютера доказательства, которые удовлетворяют целям паковщика, но не дают ничего для целей картостроителя. Эти доказательства, поскольку они не используют силу, которая приходит от понимания, слабее. Так ли необходимо, чтобы корректность кода (оставим в стороне архитектуру компьютера), который выполняет поиск, была очевидна для проверки?

Мудрые архитекторы обычно разделяют свои проекты на уровни так, что прослеживаются отдельные дискретные стадии при переходе от кода, взаимодействующего с пользователем, к коду, взаимодействующему с операционной системой. Каждый из этих уровней предоставляет возможность написать небольшое тестовое приложение. Обычно этими возможностями следует воспользоваться, поскольку хотя это, как может показаться, и вызывает рост стоимости, выявление ошибок, которые не имеют хорошо определенных тестовых точек, может неимоверно осложнить фазу окончательного тестирования и отнимет кучу времени. Чтобы полностью воспользоваться этими тестовыми возможностями, нам следует предусмотреть тестирование при определении API для наших уровней. Можно ли упростить определения API так, что мы сможем уменьшить количество всех возможных вызовов, которые не имеют смысл? Каждый уровень обязан проверять свой вход, либо, если время очень критично, требовать предварительно выверенных входных значений. Тестирование должно гарантировать, что эта логика работает в соответствии условиями функционирования этого уровня. Если API можно упростить, то одновременно автоматически упрощаются требования к тестированию.

Соображения, которые применимы к уровням, также применимы к процессам времени исполнения. Большинство нетривиальных систем требуют нескольких процессов для взаимодействия как внутри отдельной платформы, так и через сеть. Функциональность этих процессов должна быть распределена таким образом, чтобы их было можно протестировать, в идеале изолированно — из командной строки или с помощью скрипта.

Иногда нам не удается избежать внесения неоднородности (разрывности) решения, которая отсутствует в проблеме. Например, если наша база данных настолько велика, что мы должны распределить ее по нескольким машинам (и имеющаяся у нас коммерческая СУБД не обеспечивает такой возможности), то нам нужно распознать те точки, где должна измениться логика наших программ, чтобы работать на другой системе, и убедиться (тестированием), что это изменение проведено правильно.

У разработчиков объектных систем есть особенно простая стратегия автоматизации тестирования. Каждый класс (или ключевые классы, по решению архитектора) может иметь соответствующий класс, заданный так, что подменяет (эмулирует) методы системного класса. Это так хорошо работает потому, что описание этого класса стимулирует тестирование внешнего интерфейса класса в стандартизованном и хорошо определенным формате (в чем заключается суть объектов). Поэтому каждый класс может сопровождаться собственным тестовым кодом, который просто нужно заключить в небольшую обрамляющую программку для автоматизации тестирования. Эти тестовые классы иногда называют классами «янь» *(yang)*, а поставляемые классы называют соответственно классами «инь» *(yin)*.

Когда имеется автоматизированный тест, можно получить два преимущества. Первое заключается в том, что тесты можно прогонять каждую ночь, как составляющую процесса компиляции. Это позволяет программистам оставаться в хорошем настроении, когда прийдя на работу они находят e-mail от среды разработки, говорящий, что все, что команда разработала до сего момента, все еще правильно работает. Когда сообщение говорит, что есть проблемы, то не приходится тратить дни на то, чтобы найти, что же не так с их новым уровнем, когда проблема на самом деле лежит двумя уровнями ниже. Второе преимущество автоматизированного тестирования кода состоит в том, что оно не запаздывает относительно разработки, как это бывает с документацией. Если автоматизированный тест прошел компиляцию, компоновку и выполнение, то мы знаем, что описание поведения протестированного кода правильное.

Эти идеи определения и исполнения автоматизированных тестов особенно важны для очень сложных проектов, где динамическое управление конфигурацией и средства инкрементной компиляции из научно-фантастических книг позволяют сотням разработчиков работать как проклятым мартышкам на кокаине без сна и отдыха. (Сказанное — авторская риторика).

Контрольные проверки и прогон автоматизированных тестов снизу доверху не следует рассматривать как помеху в работе — это очень дешевый путь получения подтверждения прочности фундамента. Как дополнительное преимущество, такие события становятся праздниками

команды, по мере того как модуль за модулем, уровень за уровнем говорят об успешной компиляции и прохождении теста на рабочей станции менеджера. На таких праздниках команда может естественным образом взглянуть на все, чего они достигли к этому моменту, поскольку самый первый праздник может состоять просто в компиляции и запуске программы «Hello world!» и доказательстве, что компилятор работает правильно, а последний дает в результате работающий продукт, который поставляется заказчику и в котором достигнуты все цели.

❒ Даты, деньги, единицы измерения и проблема Y2K

Область, где тестирование (и сбои) могут быть существенно уменьшены уменьшением сложности системы — это распознавание неоднородностей (прерывности) в проблемной области и избежание ее более глубоким представлением. Что это означает на практике? Один из примеров — это отсчет времени и переход на летнее время. Время обычно считается секунда за секундой, а планета не дергается на орбите каждую весну и осень. Поэтому, даже если нормативные документы говорят о переходах на зимнее и летнее время, нет необходимости отражать эти переходы в системе на уровнях более нижних чем уровень интерфейса пользователя, в котором должна быть предусмотрена функция, метод или нечто, названное `LocalAdjustTime()` или как-то похоже. UNIX содержит прекрасную поддержку этих вещей, но печально мало мест, где она правильно используется.

Тот же подход применим и к временным зонам. Ваши пользователи могут прекрасно работать по всему миру и желать использовать местное время, но ваша сеть повсеместно должна использовать гринвичское время (GMT, или UTC), и все даты и времена файлов должны проставляться соответственно. Проблемы упорядочивания по дате файлов, созданных на компьютерах с по-разному установленными часами, поглощают у программистов много часов. Однажды менеджер международной сети пошла в магазин и купила сорок отвратительных, в стиле 50-х годов, одинаковых часов и коробку батареек. Весь следующий год, при посещении очередного удаленного офиса, она устанавливала на часах правильное время по Гринвичу (GMT) и вешала их на стену. В конце того года постоянные проблемы, связанные с трудностями упорядочения файлов, магически исчезли, поскольку каждый оператор имел перед глазами огромных размеров напоминание, какое время нужно установить на системных часах при перезагрузке.

Другим примером того, как избежать неоднородности в проблемной области, могут послужить два вида денег, которые нравится иметь большинству стран. В фунте всегда 100 пенсов, а в долларе 100 центов, или просто храните пенсы или центы, а если пользователь хочет, чтобы перед последними двумя цифрами печаталась десятичная точка, достаточно поставить 2 где-то в базе данных и использовать процедуру, которая просматривает базу данных. Но такие чувствительные валюты, как песеты и лиры не имеют вторых денег, вызывая проблемы, поскольку нужно помнить, что не нужно печатать десятичную точку...

Трудности, ассоциирующиеся с проблемой 2000 года, постоянно создают порог, о который то и дело вновь и вновь спотыкаются программисты. Языки программирования предоставляют типы данных, поскольку внутри типа имеется произвольный набор операций, которые можно использовать или нет, и если приходится читать код, а операции там встречаются, то их можно увидеть. Объектно-ориентированные языки расширяют эти средства для любых данных, которыми мы желаем таким образом управлять, предоставляя Абстрактные Типы Данных *(ADT)*. Реальная трудность с 2000 годом не в том, что многие программисты кодировали год двумя цифрами — в те времена требовалось экономить память, а многие дошедшие до 2000 года программы очень старые. Проблема в том, что некоторые программисты берут эти две цифры и вставляют их где ни попадя без использования подходящей подпрограммы, макроса или даже фрагмента кода, чтобы сделать поправку. Это означает, что для того, чтобы решить проблему 2000 года в этих ужасно переплетенных старых программах требуется прочитать и понять каждую отдельную строку.

❒ Безопасность

В разных местах разная потребность в безопасности. Для некоторых это неизбежное следствие природы бизнеса. Многие военные и коммерческие операции действительно нуждаются в предотвращении раскрытия того, что происходит. Но многие различия происходят из-за путаницы в понимании назначения безопасности, и это предмет обсуждения этого раздела. Как это было в этой работе уже много раз, ситуации и технологии усиления безопасности проявляются повсюду. Здесь мы сконцентрируемся на некоторых курьезах обычной безопасности.

Во-первых, следует различать требования безопасности продуктов, которые исходят из потребностей пользователя, и безопасности, требующейся собственно в среде разработки. Это может быть взаимосвязано, например, безопасность продукта зависит от конфиденциальности исходного кода, но взаимосвязь не означает эквивалентности. Не добавляйте в продуктах средства «засекречивания» принудительно или по привычке. Так ли необходимо связывать пароль с каждым идентификатором пользователя в вашем продукте? Нужен ли вам идентификатор пользователя вообще? Нельзя ли использовать для доступа к несекретной информации идентификатор «guest», не требующий пароля? Каждый пароль в вашем продукте вы должны запоминать и поддерживать, уменьшая эргономическую живучесть и увеличивая стоимость владения, ведь эти красотки наверняка забудут свои пароли.

Далее, существует два вида угроз безопасности: злонамеренные и по небрежности. Вашему продукту может потребоваться защита от злонамеренных угроз, но если вам требуется защита вашей собственной среды разработки от злонамеренных угроз изнутри (мы предполагаем, что вы растете и у вас уже есть брандмауэр), то у вас гораздо большие проблемы, и они не решаются просто установкой запрета доступа к нескольким файлам. Поэтому забудьте о злонамеренных угрозах на работе. Что касается угроз по небрежности, типа случайного уничтожения всего дерева исходных текстов проекта, то ведь у вас есть резервная копия, не правда ли? Навязывание дорогостоящих накладных расходов на безопасность каждой операции в среде разработки для защиты от «катастроф», которые, если произойдут, оказываются пустяками — это ложный путь. По мере того, как программисты все лучше осваивают персональный послойный процесс, даже эти незначительные ошибки происходят все реже, а разработка совместно используемых мысленных моделей и картостроительный жаргон в команде означают, что неформальный «этикет» разработки уже усвоен, как возглас «Реинициализация тестовой базы данных — все в порядке?» перед очисткой засоренных тестовых данных. Эти вопросы как элементы этикета — единственный приемлемый возглас в ненавистных офисах с открытой планировкой, и это единственный разумный довод их существования. Но, тем не менее, это недостаточный довод.

Поэтому не блокируйте вашу среду разработки до состояния, когда изменение хоть чего-нибудь требует присутствия каждого члена команды, чтобы ввести свои пароли. Не создавайте и не приспосабливайте системы управления конфигурацией, которые делают разработчиков беспомощными в 8 часов вечера, когда они все еще на работе, но не могут получить исправляемый файл, чтобы прочитать и разобраться. Это не только напрямую тормозит ваш проект: это также дает печальный эмоциональный опыт, который вы навьючиваете на самое высокомотивированное животное в коммерческом мире — программиста в Режиме Глубокого Хака. Чем он вас так обидел?

И наконец, не позволяйте затуманить ваши мозги ни одному элементу из веры паковщиков, в то, что мы должны знать точно кто, когда и что абсолютно обо всем. Если ваш проект — это команда, координируемая этикетом и формализованная по необходимости, у вас есть шанс. Если это базар, регулируемый детальными инструкциями, то вы обречены.

Глава 6

Техника безопасности

❒ Перегрузка мозга

Как мы уже сказали в самом начале, картостроение и паковка отличаются. Это отличие можно увидеть в организации рабочего места и в поведении человека на нем. Организация паковщиков видит всю работу состоящей из механической последовательности действий, выполняемых в определенном месте. Дело не в том, будто циничные менеджеры верят в то, что рассаживание всех в офисы с открытой планировкой, лишая таким образом возможности сконцентрироваться и, следовательно, губя продукт, не имеет значения, поскольку их цели более краткосрочные — дело в том, что они не верят, что есть такая вещь как концентрация (как ее знают картостроители), которая должна стоять на первом месте!

Некоторые рабочие среды могут быть настолько ориентированы на паковщика, что деятельность картостроителя в них невозможна. Там будут постоянные отвлечения, не дающие картостроителю возможности сосредоточиться. Собрания будут построены из серии докладов в стиле «положение обязывает», делаемых людьми, подсчитывающими очки в поиске победивших и проигравших, что никак не связано с произносимым. В этой ситуации, картостроитель, для выражения мысли которого может потребоваться два, ну три, отчетливо произнесенных предложения, будет выглядеть как трогательный неудачник.

Хуже того, эта самая неэффективность когнитивной стратегии паковщиков ведет к тому, что ее приверженцы становятся в яростную оборонительную позицию, когда критике подвергается скрывающий весь недостаток понимания паковщиков этикет. Если в коллективе мала пропорция картостроителей (или даже велика, но они не знают, что происходит), то могут возникнуть напряженные ситуации.

Ключевой момент в этой картинке состоит в том, что нет смысла в попытках картостроителей убедить своих коллег-паковщиков в значении строгого и полного подхода даже с помощью более осторожных аргументов — проблема в том, что паковщики с самого начала не готовы принять любой вид детализованного рассуждения! Поэтому картостроитель может просто устать, пытаясь доказать что-то людям, которые просто не слушают.

Можно на самом деле переутомиться, занимаясь картостроением, и очень важно это заметить и избежать. Первое, что нужно сделать — это распознать ситуации, где интенсивный подход картостроителя наиболее подходящ.

Картостроение в целом можно рассматривать как поиск, а особенность поиска состоит в том, что неизвестно, где это искомое лежит. Поэтому необходимо продолжать поиск до тех пор, пока объект не будет найден. Это гораздо легче сделать, если есть уверенность, что объект поиска действительно существует! В противном случае нужно предусматривать некоторый искусственный ограничитель, например, лимит времени. Тут приходит упоминавшаяся в начале «вера картостроителя» — картостроители снова и снова открывают, что мир вокруг нас всегда проще, чем он кажется, при условии, что на него смотрят правильно. Иногда приходится раскрывать и изучать огромную скрытую сложность, но простота, необходимая достаточность

«плато качества» в конце концов будет достигнута. Во всех ситуациях, которые можно найти в реальном мире, инвестиции картостроителя будут небесполезными, поскольку чем глубже видится скрытое, тем более стоящим (мощным) будет результат. Ситуации, которые не включают «естественный мир» в этом смысле (в конце концов, все в этом мире «естественное») — это те ситуации, где сознание пытается создать локальную область иррациональности. Другими словами, где вы играете против другого ума, который случайно или осознанно старается привести вас в растерянность, стараясь показать только части всей системы (которая рациональна) так, что она выглядит для вас полной, но иррациональной. Поэтому паковщики, используя тот же самый язык, что и картостроители, при разговорах о мышлении, но подразумевая нечто другое, оказываются иррациональными. Когда картинка дополняется барьером картостроитель/паковщик, то мы снова возвращаемся в естественный мир, который включает противодействующий разум, и рациональность восстанавливается.

Есть такая игра на радио: *Mornington Crescent*, правила которой по некоторым соображениям никогда не публиковались. Если кто-то слушает эту игру и подразумевает, что существует лежащая в ее основе рациональная система правил, то он может быстро свихнуться. Нет таких правил. На самом деле игра состоит в ловкости, с которой игроки делают вид, что правила все же существуют, и это придает игре своеобразный характер.

Итак, если в рассматриваемой ситуации присутствует еще один ум, всегда нужно учитывать его потенциальное противодействие (капризность, извращенность), чтобы гарантировать, что ситуация остается естественной. Может показаться, что нас как компьютерных программистов это повергает в паралич паранойи, поскольку многие проблемы, с которыми мы имеем дело, включают пользователей. Но дела не настолько плохи, поскольку если бизнес просуществовал столько времени, то это полностью естественное явление, которое подвластно картостроительному анализу, даже если никто из игроков на самом деле не понимает, что же на самом деле происходит. Однако помните, что короткоживущие транзакции бизнеса, такие как выставляемые на биржах только в короткий период быстрого изменения рынка заявки, не могут быть источником существования, и потому могут рассматриваться как продукт извращенных умов. Это не означает, что такие транзакции нельзя автоматизировать, — просто это означает, что единственное, что приходится делать в этом случае — это кодировать с помощью 4GL, или какое там еще средство быстрой разработки *(RAD)* вы используете, глупость, раз уж биржевые игроки просят вас это сделать, и пусть они беспокоятся о восстановлении собственного нормального поведения по отношению к таким же извращенным коллегам. Иногда организации, занимающиеся такими вещами, просят картостроителей взглянуть на ситуацию в целом и посмотреть, могут ли они найти какую-нибудь логику, если границы достаточно широки. Эти работы могут оказаться исключительно интересными и благодарными.

Идентифицировав ситуации, где нам следует отказаться от картостроения или переопределить проблему, мы остаемся с проблемой того, как долго мы будем идти к пониманию. Опыт картостроения приносит причудливую интуицию относительно интуиции, которая дает хорошее чутье на эти вещи. Не высказывайте оценки до того, как вы сами лично не убедитесь, что набрались достаточно опыта в их получении. Запишите вашу личную оценку перед началом картостроительной работы, и посмотрите потом, правильной ли она оказалась. Вы можете ошибиться несмотря на десятилетия опыта. Если бы работа была понята, то был бы создан автоматизирующий эту работу коммерческий продукт *(COTS)*, не правда ли?

Картостроители часто сталкиваются с проблемами за много лет до их решения — исследования, собранные в этой работе, заняли тридцать или шесть лет, в зависимости от того, как установить границы! Важно, что хотя есть ограничение на силы, которые можно направить на решение проблемы, нет ограничения на нахождение в состоянии готовности к этой проблеме, увеличивающем продолжительность жизни картостроителя. Нет позора в осознании длительности решения проблемы и уменьшении интенсивности попыток ее решения. Эта мотивация — один из наиболее забавных примеров коммуникационного барьера картостроитель/паковщик. Для паковщика проект состоит из последовательности действий, которые требуется выполнить.

Скорость, с которой выполняются действия, индицирует эффективность работающего. Работа над проблемой, от которой приходится отказываться (оставлять в покое), есть доказательство неорганизованности части работников. Следовательно, паковщики способны посмеиваться над «курьезными проектами» картостроителей, осознавая и, тем не менее, одновременно отвергая выдающиеся творческие результаты, которые картостроители регулярно выдают, поскольку им ясно, что они не были достигнуты «правильно», хотя у паковщиков нет предположений о том, каким должен быть «правильный» подход. Плохи дела. Образование за это в большом ответе.

Руководствуясь интересами дела, мы должны обратить внимание на то, что при интенсивном занятии картостроением нужно себя беречь. Основы физического здоровья могут быть подорваны двумя способами. Во время интенсивной работы некоторые картостроители обнаруживают, что без должного внимания оказываются еда и необходимые упражнения. Гордитесь тем, кто вы и что вы можете делать, но перед тем, как войти в это состояние, убедитесь, что поблизости лежит много свежих фруктов и холодильник полон. Тогда поесть — это не проблема. Каждый картостроитель, похоже, в состоянии хорошо работать, прогуливаясь в одиночку, поэтому прогуливайтесь.

Второй способ подорвать свое здоровье — перегрузить свой мозг. Следующий совет не нужно воспринимать буквально, поскольку у нас нет нейрологического основания для него, но это то, что происходит с картостроителями, которые столкнулись с трудностями. Если проблема очень большая, то ее сложность, которую нужно поместить в ум, перед тем как произойдет ее коллапс, старается заполнить весь мозг картостроителя и занять те части, которые содержат образ тела картостроителя. Когда это происходит, физическое состояние за несколько дней может ухудшится, человек очень быстро может стать похожим на высохшую картофелину. Мы не собираемся советовать не делать этого — это само приходит к вам. Но если вы остановитесь посреди недели из-за внезапно появившейся беспомощности и попытаетесь восстановить образ своего тела физической работой или получив некую обратную связь, то вы сможете восстановить свое состояние так же быстро, как и потеряли его. Если выгрузить образ своего тела надолго, то восстановить его будет гораздо труднее.

Помните, что мы говорили о напрасности трат энергии на повторяющиеся непродуктивные циклы размышлений.

Помните, что кое-что также происходит и во внешнем мире. Нужно поддерживать ваши личные взаимоотношения, и хотя кое-кто возле вас знает о вашем состоянии и ждет вашего возвращения, другие будут нуждаться в некотором внимании. Практикуйтесь работать в фоновом режиме, используя ранее описанную технику «верчения тарелки». Со временем вы обнаружите, что можете менять объем сознания, выделяемый на размышление над проблемой, и объем, оставляемый свободным для поддержания учтивой беседы. Если вы находитесь в стадии, когда требуется мобилизовать весь ум, и вы не хотите останавливаться, поскольку потребуется неделя на восстановление той частичной картины, которая у вас уже есть, и у вас есть функция, которую вы предполагаете применить, вы всегда можете перейти в состояние счастливого идиота. Вы знаете, что вы не обращаете внимание на болтовню вокруг вас, но, что невероятно, болтуны редко это замечают!

Не обращайте внимание на мнения паковщиков о ваших вредных для здоровья способах. Такие комментарии — пустяки по сравнению с теми советами о здоровье, которые мы обсудили в этом разделе. Для паковщиков «думать слишком много» — это расстройство (болезнь) само по себе!

❒ Переутомление мозга

Картостроение — это интенсивное, поглощающее эмоции занятие. Каждый коллапс проблемы — праздник. Проблема в том, что мы достигаем пика интенсивности и состояния праздника, а затем эта проклятая штука раскалывается, и больше нечем заняться. В конце проекта это

может привести к опасности депрессии, поскольку веселье прекращается. Это также может привести к тому, что ум рыщет и рыщет вокруг, пропуская то, в чем теперь имеется очень простая структура, кружась в непродуктивном цикле мышления.

Все это плохо для вас. Если вы работали в команде, обменивайтесь впечатлениями. Позвольте себе поговорить о чем-то полезном, например о своем подходе к проблеме, и чему вы научились в решении этого класса проблем, о специфике, с которой вы столкнулись. Что вы узнали о своей платформе? Это круто или так себе? Во время бесед, помните о контроле за настроением. Идея в том, чтобы спланировать вниз, а не взлететь наверх. Замените удовольствие от раскалывания проблемы празднованием вашего успеха.

Если вы не в команде, попытайтесь как можно скорее включиться в совершенно другую деятельность, в которой участвуют другие. Всегда найдется проблема в которой, если вы включитесь в работу, вы не будете ждать официальных праздников, а когда вы ее решите, то можете испугаться того, как быстро вы это сделали. Поэтому имейте рассудок и пригласите к себе друзей или сами пойдите к ним в гости.

Если худшее идет к худшему, и вы остаетесь наедине с решенной проблемой, отбросьте ее как можно скорее. Возьмите свои трофеи, листинги, диаграммы, продукт, полейте отравой по вашему выбору и проведите вечер тайно злорадствуя. Делать так может показаться слишком снисходительным по отношению к себе, но это дает оригинальную эмоциональную встряску, связанную с тяжелой утратой. Но не повторяйте еще раз — только один вечер, затем продолжайте жить!

❒ Переработка

Не путайте удивительную работоспособность картостроителя с деятельностью паковщика по накручиванию очков. Помните, картостроение — это целесообразное приложение сил, и пусть это выработается в вашей голове, так что действительно выполняемое вами в физическом мире ограничится необходимым и достаточным правильным действием.

Держите в уме персональный послойный процесс и оценивайте вашу реакцию на ситуацию, интересуясь, подходит ли имеющийся план. Этот путь, на работе или дома, практичен и объективен, и не имеет отношения к морали.

❒ Управление межкультурным интерфейсом

Осознайте необходимость обеспечения интерфейса между ценностями картостроителей, необходимыми для производства программного обеспечения и ценностями паковщиков, которые обычно окружают ваш проект в коммерческой среде.

Избегайте вовлечения в дискуссии без установленных базовых правил рационального мышления. Требуйте пространства для структурного маневра.

Когда требуется сделать выбор, не пытайтесь выложить всю логику, как вам бы хотелось. Помните, необходимость знания вами всех фактов для принятия собственного решения, не разделяется паковщиками. Также не пытайтесь объяснить, почему оптимальное решение правильное. В спорах с вами это подстрекает паковщиков на набор политических очков (единственный способ выжить в бесконечном хаосе). Вместо этого, защитите себя, оперируя несколькими соображениями относительно цены и преимуществ, и ограничьте себя тем, чтобы убедить окружающих в понимании этих соображений. Паковщики в состоянии это сделать — именно так они покупают стиральные машины и Двойные Шоколадные Бургеры.

Выявляйте неоднозначности и разрешайте их. Полезное звучное словечко для обозначения этого упражнения — «Парад Риска». Идентифицируйте неизученное и обнародуйте его с оценкой, может ли это стать проблемой. Корректируйте Парад Риска по мере изменения ситуации.

Представляйте эти данные формально и неформально, но постарайтесь, чтобы все знали, где они находятся.

Будьте готовы использовать фразу: «Я не знаю». Этот простой честный поступок может положить конец помпезности и принуждению паковщиков, оставляя вам ясное понимание того, где вы находитесь.

Все эти методы работают по отношению к основной проблеме. Паковщики хотят избавиться от сложности сваливая ответственность на других. За исключением случаев, когда вы пытаетесь это предотвратить, вы можете оказаться «ответственным» за отличие реального мира от тех фантазий паковщиков по поводу того, каким они хотели бы его видеть. Действуя так, чтобы поместить реальности в общедоступное место, но не всучивая их конкретному человеку, вы на самом деле помогаете восстановить порядок вокруг себя, уберегая при этом собственную задницу.

❒ Личная ответственность и лидерство

Картостроителям свойственно обмениваться и согласовывать свои мысленные модели. Тогда для увеличения взаимного знания они могут легко ссылаться на аспекты этих моделей нестрогим языком. Они также делают акцент на получении оптимального решения, им более комфортна модель совместной работы победитель/победитель, а не победитель/проигравший.

Все эти факторы ведут к общей тенденции, которая возникает, когда картостроители собираются вместе поделиться проблемами и решениями и научить друг друга. Эта кооперативная тенденция — важная часть хакерской культуры.

Простой факт состоит в том, что методы картостроения, особенно картостроения в данной области, — мощное искусство. То, что на формальных курсах мы быстро загружаем новые языки и обозначения в мозги программиста — это только крем на торте[1]. Реальное обучение происходит на работе, по мере того, как опытные люди показывают новичкам методы, которые они могут найти полезными. Новички сами потом оценивают, что использовать и как использовать, в свете состояния предмета, в который они вошли. Это один из способов быстро войти в курс дела и причина, по которой наша область так быстро эволюционирует.

Мы можем работать с этим знанием и культивировать его, чтобы взять контроль над нашими собственными разработками, либо можем игнорировать его и играть в «перечни навыков» *(skills summaries)*, перечисляющие языки программирования. Мы предполагаем, что разумный способ взять управление уже был найден как естественное следствие из проблем. Наша индустрия опутана формальными, но произвольными классификациями, но та, которую мы предлагаем, неформальна, но реальна, и она уже существует, нужно только открыто рассмотреть ее на рабочем месте.

Традиционно, тех, кто начинает учиться навыкам ремесла, называют подмастерьями. Им с самого первого дня поручают реальную работу, но всегда под присмотром более опытного работника. Когда становится очевидным, что присмотр более не требуется, подмастерья признают квалифицированным работником, на которого можно положиться, что он хорошо сделает работу и сможет руководить подмастерьями, которые могут понадобиться ему в помощь.

Многим компетентным работникам нравится работа ремесленника, использующего свои навыки, и они остаются ремесленниками на всю оставшуюся жизнь. Они предпочитают, чтобы другой человек, возможно с другим темпераментом, взял ответственность за успех проектов. Такой человек не может быть назначен номинально. У него может быть высокая квалификация, напор и знание природы профессии, а может и не быть. В то время, как мастерство разработчика может расти под руководством других, это новый мастер, который должен найти свой собственный голос. Следующие мысли направлены студентам, а не учителям. Чтобы стать

[1] Вершина айсберга — С.К.

признанным мастером, ремесленник должен создать образец искусства. В нем он демонстрирует свою способность создать качественный образец. В старые времена, когда работа была связана с материальными изделиями, этот образец был чем-то выдающимся, поскольку новый мастер хотел продемонстрировать уровень мастерства, и, вероятно, никогда больше не стал бы делать ничего столь же причудливого. Более поздние изделия были бы направлены на удовлетворение реальных потребностей, и поэтому более соответствовали бы своему назначению. Таким образом, образец мастера — это на самом деле самый нижний уровень работы мастера, а не высший, как это можно было бы предположить из общих соображений. В наше время, образец мастера — эта целая система, выведенная на плато качества, и единственное отличие состоит в том, что мы питаем отвращение к ненужным бантикам и бубенчикам (наворотам и примочкам). По прежнему, образец мастера — это самая первая система, а все последующие должны быть лучше, как это и происходит у всех хороших программистов, опыт которых мы всегда изучаем. Один из доводов в пользу того, что программистам легче учиться друг у друга, состоит в том, что мы одновременно и учителя и ученики, и очень быстро переходим из одного состояния в другое.

Из этой модели мастерства проистекает ряд следствий. Во-первых, она максимально повышает одновременно проработанность и продуктивность. Управляющий проектом мастер должен гарантировать, что каждый член команды работает в пределах своей квалификации, но на самом пределе. Для компетентных программистов нет недостатка работы, поэтому поиск приложения своего мастерства не является проблемой. Это требует от работников приложения усилий, которые не только приносят непосредственные дивиденды, но также гарантируют, что ресурсы используются максимально эффективно, это как раз то, чего хотят добиться бухгалтеры, но чего не может быть в процедурной модели, скопированной с индустрии паковщиков, основанной на повторении. Нет двух похожих церквей, нет двух похожих систем.

Другое соображение, о котором уже знают все программисты, но которое стоит повторить в обществе паковщиков с их победителями/проигравшими, заключается в том, что страх учиться на работе, который овладевает многими профессионалами, к нам не имеет никакого отношения. Мы стоим на пороге новой культурной эры. Оглядитесь и посмотрите, можете ли вы найти какой-нибудь способ сделать общество более интеллигентным. Вы когда-нибудь пытались купить лошадь? Еще долгое время не будет недостатка в работе для программистов, а когда будет, пусть все сделают роботы, а мы просто запрограммируем забавную графику, которая будет крутиться на наших карнавалах.

❒ Фальшивая цель деквалификации

Этот раздел явно остановится на том, что несколько раз упоминалось в этой работе, поскольку это составляет существенное отличие во взглядах картостроителей и паковщиков на рабочее место.

Видение мира паковщиком неестественно — ему приходится тренироваться быть ребенком вместо того, чтобы развивать естественные для детей способности познавать мир. Вероятно, это была самая дешевая форма минимального образования и максимальной организации с начала аграрной эры и до конца индустриальной. В это время люди выполняли повторяющиеся задачи в материальном мире.

Видение мира картостроителем использует и развивает естественные человеческие мыслительные способности по исследованию идей и в информационную эру является для людей уникальным резервом.

Программирование — это деятельность картостроителей. Если мы на самом деле вынуждены снова и снова писать одну и ту же программу, какая-нибудь светлая голова разработает готовый продукт *(COTS)*, и программист будет не нужен.

Традиционный взгляд паковщика на любую работу состоит в том, что он предполагает, что это повторяющаяся лишенная смысла деятельность, и пытается найти способ, как сделать ее как можно проще, оптимизировать ее, и если не полностью автоматизировать, то снизить требования к квалификации, чтобы уменьшить затраты на работников.

Утверждать, что стратегия паковщика, которая применима к тюкам сена, также применима и к любой другой работе, просто потому, что в нее вовлечено много людей — значит останавливать движение от примитивной материальной экономики, когда человек — это убогая замена лошади или паровой машины, или даже станка с программным управлением, к информационной экономике, когда черная работа менее необходима, чем понимание или творчество.

Фильм «Субботний вечер и воскресное утро» *(Saturday Night and Sunday Morning)* начинается с показа рабочего в механическом цехе, занятом массовым производством компонентов на станках. «Девятьсот девяносто восемь... девятьсот девяносто, блин, девять...», бормочет он. Безысходность в том, что это на самом деле замечательно отражает положение в наше время. В те времена менеджеры платили рабочим за произведенные детали, передавая рабочим реальную силу и инициативу оптимизировать. В контексте программного обеспечения, нам не следует пытаться контролировать каждый аспект ввода 1000 строк одного и того же кода каждый день, нам следует спросить, почему работник до сих пор не написал макрос.

Концепция деквалификации буквально пронизывает наше общество, и это делает ее очень вредной. Одурманенное сознание может положить ее перед вами, но все аргументы в ее пользу фальшивые. Никогда не забывайте об этом.

Держа в уме недопустимость деквалифицирующих программ, мы можем исследовать ряд мифов. Даже в сфере массового материального производства трудно сравнивать похожее с похожим. Например, мы можем сравнивать производство автомобилей за этот месяц и за прошлый месяц, и посмотреть в каком из них оно было лучше. А за прошлый год? Мы использовали тогда три разных вида блоков фар и предлагали пять окрасок. А десять лет назад? Каждый аспект технологии — антиблокировка тормозов, система управления автомобилем, воздушные мешки, состав топлива — все изменилось. Требуется настоящее озарение, чтобы просто рассказать, насколько мы богаче, чем предшественники! Академические попытки сравнения отношения заработной платы к цене за большой период времени скатываются к затратам на покупку кирпичей и хлеба, поскольку на протяжении многих столетий только эти вещи можно было всегда пойти и купить! Ну что мы можем поделать с поразительным экспоненциальным ростом «улучшения производительности», связанным с каждым новым «прорывом в технологии», который позволит вам набрать в штат проекта орангутангов и получать от них по 10 миллионов строк кода *(10 MLOCS)* в секунду. С чем на Земле можно сравнить этот рост? Нам приходится делать вывод о том, какие ужасные кучи мусора вываливают на нас в каждой статистически убогой работе.

А что такое «дружественные пользователю метафоры», означающие, что орангутанги теперь могут делать все, что им нравится, не требуя квалификации? Мы предполагаем, что истинная ситуация заключается в том, что некоторые сегменты рынка эксплуатируют миф о том, что возможна деквалификация в управлении сложностью, и предлагают продукты, которые при поверхностном изучении в течение короткого времени на самом деле кажутся «простыми в использовании». Трагедия в том, что пользователям на самом деле приходится выполнять операции типа конфигурирования адресов IP, брандмауэров, дисков, сканнеров, принтеров, совместно используемых дисководов, бюджетов и т. д. В этом месте мы обнаруживаем, что вместо компьютера, не требующего никакой квалификации, поскольку он претендует на роль еще одного предмета мебели типа стола, у нас оказывается компьютер, к которому обычные навыки работы с компьютером не применимы, поскольку, в конце концов, столу не нужно иметь сконфигурированных бюджетов, поэтому при его использовании нет необходимости в навыках конфигурирования бюджета пользователя стола. Мы неожиданно обнаруживаем, что даже в привычных ситуациях, когда вдруг решили установить новый IP без перезагрузки всей машины, «шароварные» системы, не отказывающиеся от звания компьютеров, оказываются более

дружественными пользователю, чем так называемые «дружественные пользователю» штуковины.

❒ Пути к отступлению

Как работающие в реальном коммерческом окружении профессиональные программисты, мы часто работаем в жестких временных рамках, так что мы не можем гарантировать достижения настоящего плато качества решения в пределах этих рамок. Важная часть персонального послойного процесса и неформальная часть плана управления проектом в достаточно зрелой организации состоит поэтому в определении и постоянном переопределении наших непредвиденно изменившихся планов.

Наиболее общим случаем корректировки плана является, к сожалению, сокращение функциональности. Это редко бывает эффективным способом экономии времени, поскольку большая часть низкоуровневой функциональности обычно все равно нужна для поддержки работы ужатых слоев приложения, которые в любом случае должны быть дешевыми в реализации, если нижние уровни обеспечивают правильную специализацию прикладного уровня. Мы предполагаем, что более эффективен следующий подход:

- Во-первых, правильно разбейте на уровни. Выясните суть API каждого выявленного уровня.
- Во-вторых, примените совет Кена Томпсона *(Ken Thompson)*: «Если сомневаетесь, применяйте грубую силу». Определите раздутый, дорогой, неэффективный, тяжелый в реализации и ужасный способ обеспечения функциональности на каждом уровне. Не важно, что вся система в целом может просто не работать, если реализовать ее именно таким способом, поскольку этого не будет.
- В-третьих, обеспечьте достижение плато качества на каждом уровне. Пересматривайте время от времени свою незрелую методику и добавляйте в нее те хорошие части, которые у вас уже есть, а остальное заполняйте по возможности различными грубыми методами.
- В-четвертых, когда начнутся трудности, исходя из краткосрочных и среднесрочных потребностей заказчика и имеющегося времени сделайте на свой риск оптимальный выбор, какие части будут поставлены сырыми, а какие стоит сделать максимально хорошо.

Этот подход имеет огромное преимущество, позволяющее делать наилучшие возможные в данный момент вещи. Вы не сможете сделать для вашего заказчика что-либо лучшее, чем это.

Когда уровни могут быть реализованы вчерне, и если у вас уже есть фрагменты, которые вы написали для тестирования операционной системы и API специальных библиотек, то вы на самом деле в состоянии очень быстро создать сырую версию. Это дает каждому программисту общий набор тестовых заглушек, значительно уменьшающих риск из-за одновременного создания всех уровней.

❒ Новички в команде

Будьте внимательны к присоединяющимся к команде новым работникам. Как и во всей этой работе, мы не подразумеваем под этим сентиментальную болтовню ритуала «добро пожаловать к нашему шалашу»: мы подразумеваем нечто более практичное.

Команда обладает мысленной моделью работы. Посвятите в нее новичков. Убедитесь, что они понимают, что наступает Моделирование Ситуации и пригласите их. Объясните цель проекта, затем поясните весь используемый в проекте внешний (со стороны заказчика) и внутренний

(мысленная модель) язык. Дайте обзор среды разработки, включающей инструменты, средства управления конфигурацией, компиляторы и т. д. Не заставляйте их спрашивать о каждой стадии.

Никогда не совершайте ошибку, заботливо обеспечивая им стол, стул, рабочую станцию, но не предоставляя бюджет *(account)* и конкретного дела. Хуже всего, когда приходящий на новый проект становится похожим на лимон[2], а каждая следующая минута кажется длиннее предыдущей.

На BT *(British Telekom ?)* очень эффективно использовалась очень удачная практическая идея, которая состоит в представлении новичка официальному «назначаемому другу». Этот назначаемый друг — равный ему по должности, кто уже давно работает в команде, и явно представляется как источник информации, которого «можно беспокоить» по поводу всего, что нужно узнать новичку. Одно из замечательных свойств этого подхода в том, что будучи равным, назначаемый друг будет знать правильные ответы на вопросы новичка. Бумага обычно лежит в коричневом шкафу, а листы A3 для больших диаграмм — в зеленом ящике внизу.

[2]В смысле, киснет — С.К.

Глава 7

Некоторые забавные вещи

❒ Ричард Фейнман *(Richard Feynman)*

Для любого, кто желает принести на рабочее место силу картостроителя, будет очень полезным изучение жизни и работы физика Ричарда Фейнмана *(Richard Feynman)*. Он рассказывал историю. Отец показал ему одну из певчих птиц *(Spencers' Warbler — певчая птица Спенсера)*. Имя ее было придуманным. Затем отец привел ему названия этой птицы на многих других языках, и оказалось, что молодой Фейнман узнал не больше, чем в начале. Механически запомненные названия вещей ничего не значат. Только посмотрев на саму птицу можно хоть что-то сказать о ней.

Он был чрезвычайно честным и видел сквозь искусственную сложность, всегда настаивая на простоте и фактах. Посмотрите на его личную версию Сообщения Претендента *(? The Challenger Report)*, которая содержится в его книге «Почему тебя беспокоит, что думают другие?» *(What Do You Care What Other People Think?)*

У него был простой, веселый, любопытный язык, заполненный маленькими картинками и вдохновением. Его методы прикалывания помпезности вызывают естественный смех.

Недавно были опубликованы его «Лекции по вычислениям» *(Lectures on Computation)*, и их стоит почитать, как и все что он писал, начиная с «Шести легких частей» *(Six Easy Pieces)* до «Красной книги лекций» *(Red Book Lectures)*. Книги Джеймса Глейка «Гений» *(James Gleik's* Genius*)* и Гриббена «Ричард Фейнман» *(Richard Feynman)* — прекрасно написанные биографии.

Добудьте эти книги и прочтите.

❒ Джорж Спенсер-Браун *(George Spencer-Brown)*

«Законы формы» *(The Laws of Form)* Джоржа Спенсера-Брауна — это небольшая книжка по математике с комментариями, которая, по мнению современных логиков, содержит форму «модальной логики» *(modal logic)*, характеризуемую тем, что в ней есть правила логической системы, которые по-разному применяются в разных местах в манере, определяемой правилами самой логики.

С точки зрения программиста, есть два аспекта этой книги, которые несомненно будут стимулировать мышление. В основном тексте автор показывает, как создать логику предиката только с одним символом, предлагая более глубокий, чем можно предположить, взгляд на «фундаментальные» логические и вычислительные операции типа NOT, OR, AND, XOR.

Затем идут примечания, простые и глубокие, к которым возвращаешься вновь и вновь, часто сформированные с помощью методов логики предикатов с одним символом, которые можно рассматривать как простое разделение плоскости на две части, таким образом, что они становятся двумя отдельными вещами, и при этом есть о чем сказать. Например, автор говорит:

> На некоторой стадии во всей математике становятся явными шоры, которые оказываются на нас при следовании в течение некоторого времени правилу без осознания этого

факта. Это можно описать как использование завуалированного соглашения. Существенный аспект развития математики состоит в развитии понимания того, что же мы делаем такого, что при этом тайное становится явным. Математика в этом аспекте психоделична.

Или:

> При обнаружении доказательства мы должны совершать нечто более искусное, чем поиск. Мы должны посмотреть на уместность некоторого факта по отношению к утверждению, которое мы хотим доказать, с точки зрения общей картины и тех фактов, в справедливости которых мы уверены. Несмотря на то, что мы можем знать, как осуществлять поиск того, что мы не можем видеть, утонченность метода «нахождения» того, что мы уже можем видеть, может гораздо проще завершить наши усилия.

Или:

> Любые великие открытия в математике и других дисциплинах, если уж они совершены, кажутся чрезвычайно простыми и очевидными, и делают всех, включая открывателя, глупцами, которые до сих пор не могли этого открыть. Слишком часто забывается, что древним символом источника зарождения мира является шут (дурак), и эта глупость, будучи божественным состоянием, не является обстоятельством, которым можно гордиться или его осуждать.
>
> К сожалению, мы находим, что сегодня системы образования настолько далеко отошли от этой простой истины, что учат нас гордиться тем, что мы знаем и стыдиться невежества. Это порочно дважды. Это порочно не только потому, что гордость сама по себе смертный грех, но, кроме того, учить гордости за знания — значит воздвигнуть эффективный барьер перед развитием того, что мы уже знаем, поскольку он делает стыдными попытки посмотреть по ту сторону стены, воздвигнутой собственным невежеством.
>
> Для любого человека, готового с уважением вступить в царство своего великого и всеобщего невежества, секреты бытия в конце концов раскроются, и они будут раскрываться в мере, соответствующей его свободе от естественного или внушенного стыда в его отношении к их раскрытию.
>
> Перед лицом сильного и действительно яростного социального давления на них, немногие люди готовы принять этот простой и благодарный путь к здравомыслию. И кто в обществе, где видный психиатр может заявлять, что, если бы был шанс, то подверг бы Ньютона электрошоковой терапии, может упрекать того, кто боится так делать?
>
> Чтобы постигнуть такую наипростейшую истину, которую знал и применял Ньютон, требуются годы размышлений. Не деятельности. Не доказательств. Не расчетов. Не упорного труда. Не чтения. Не разговоров. Не прикладывания усилий. Не думания. Просто удержания в мозгу того, что нужно знать. А еще те, кто смело идет этим путем к реальному открытию, не только не получают практически никакой поддержки в том как это делать, они активно подавляются и вынуждены хранить свое открытие в секрете, притворяясь, между тем, что они вовлечены в безумные заблуждения и соответствуют тем тупиковым взглядам, в которые окружающие не перестают верить.

Какое великолепное описание так долго обсуждаемого нами коммуникационного барьера между картостроителями и паковщиками, и вряд ли кто скажет лучше! Наконец, вот видение силы картостроительной стратегии познания как продолжающей поиск в основе изучаемого явления еще более глубокой структуры, что обусловлено способом, которым мы выбираем, выполняя отдельное различение в пустоте:

> Мы обдумываем, сконцентрировавшись полностью на этом, форму отдельной конструкции... основываясь на самом первом отличии. Суть результата нашего обдумывания — как оно может появиться, в свете различных состояний ума, в которые мы себя вводим.

Еще он говорит:

> Таким образом, мы не можем отбросить факт, что мир, который мы знаем, создан таким, чтобы видеть себя (и, таким образом, чтобы быть способным видеть).

Богатство из максимальной простоты. Предел сокращения сложности, и искусство использования треугольника творчества, чтобы поместить «вилку» (сделать «ход конем») нашего восприятия на правильный для наших целей уровень абстракции. Как программисты, мы работаем, и каждым своим действием доказываем это, в точно таком же творческом пространстве, как и большинство абстрактных математиков и поэтов. Помня слова Джорджа Спенсера-Брауна, взгляните на это стихотворение Лори Ли *(Laurie Lee)*, и задайтесь вопросом, ваш код когда-нибудь рисовал структуру предметной области, делал все, что он должен был сделать, и выглядел ли при этом так же совершенно?

Fish and Water

A golden fish like a pint of wine
Rolls the sea undergreen,
Glassily balanced on the tide
Only the skin between.
Fish and water lean together,
Separate and one,
Till a fatal flash of the instant sun
Lazily corkscrews down.
Did fish and water drink each other?
The reed leans there alone;
As we, who once drank each other's breath,
Have emptied the air, and gone.

❒ Книга по физике как продукт культуры

Нас регулярно приглашают посмотреть на мир определенным образом, как пользователей, которые верят, что они понимают свой мир, в стиле и с подходами гуру, через наши собственные концепции. Мы постоянно стремимся увидеть мир таким, какой он есть, таким образом, что мы делаем его представления в наших системах как можно проще. Точно также как нужно (хотя бы раз) увидеть плато качества перед тем, как мы сможем его распознавать, поэтому приходится «ходить кругами вокруг Защитной Полосы и смотреть, сколько раз они зажигают огонь»; приходится подвергать сомнению предположительно целостную реальность, пока не появиться возможность узнать, в чем тут дело.

Не может быть ничего более целостного чем Уровень Физики: у любого, кто говорит, что это продукт культуры, общественное соглашение между циничными физиками, чтобы сделать мир непонятным для цивилизованных людей с гуманитарным образованием, определенно поехала крыша. Странная вещь, некоторые люди талантливо доказывают, что законы физики создаются физиками, а не открываются, и следует пресечь их попытки сделать законы другими!

Настоящая трагедия этих глупых болтунов в том, что если бы им удалось хоть немного изучить физику, они смогли бы обнаружить, что хотя законы физики существовали задолго до изучавших их физиков и совершенно независимы от воззрений физиков, понимание пространства, которое мы рисуем исходя из этих законов, может, тем не менее, быть продуктом культуры.

Чтобы пояснить это замечательное утверждение, нам нужно сослаться на трех физиков. Исаак Ньютон *(Isaac Newton)* открыл современную механику, и записал свои открытия в основном на латыни, а не в символическом виде, как мы это сегодня делаем. То, что было изобретено

викторианцем Оливером Хевисайдом *(Oliver Heavyside)*, и что мы обычно называем «ньютоновской» физикой, было бы ближе к действительности называть интерпретацией Хевисайдом физики Ньютона. Ричард Фейнман был физиком нашего времени, который в Красных Книгах *(Red Books)* попытался обобщить то, что уже было известно, настолько элегантно, насколько он смог это сделать. Вещи становятся интереснее, когда мы сравниваем содержание в «Принципах» гениального Ньютона *(Principia)*, главы «Красных книг» Фейнмана, в том, что было известно Ньютону, и главы «Современной физики» *(Advanced Level Physics)* Нелкона и Паркера *(Nelkon and Parker)* (стандартный британский учебник), тоже в пунктах, которые были известны Ньютону.

Principia:

- Три закона движения Ньютона.
- Траектории в гравитации (включая движения вверх/вниз).
- Движение в среде с трением.
- Гидростатика.
- Маятники.
- Движение в жидкостях.

Red Books:

- Энергия.
- Время и расстояние.
- Гравитация.
- Движение.
- Три закона Ньютона.
- Движение вверх/вниз.
- Маятники.
- Гидростатика и движение жидкости.

Advanced Level Physics:

- Три закона Ньютона.
- Маятники.
- Гидростатика.
- Гравитация.
- Энергия.

Что отличает *Advanced Level Physics* — механика в ней увеличивает сложность уравнений в системе Хевисайда, в то время как в двух других работах намерения иные.

Ньютон начинает со своих трех законов, в то время как Фейнман вводит энергию раньше и оставляет три закона на потом. Но как только они определили некоторые термины, с которыми работают, оба гения начинают говорить нам о пространстве, где все всегда в движении, а затем заполняют картину. Они делают это задолго до обсуждения маятников, которые математически более просты, но являются специальным случаем по сравнению с планетами на орбитах.

Advanced Level Physics рассматривает маятники до гравитации и рассуждает о гидростатике, которую оба гения рассматривают гораздо позже, до того как вообще упоминается гравитация; к этому времени, как мы предполагаем, студенты уже научились очень эффективно выполнять вычисления, но их мысленная модель мироздания полна ссылок с неясными связями между ними.

Хотя это может показаться неудобным с точки зрения математики (хотя текст Ньютона, казалось бы, не зависит от математической интерпретации, но у Фейнмана мы должны принять ее во внимание), оба гения хотят перенести идею о всеобщем движении как можно ближе к началу.

Возможно ли изучить физику ошибочным путем и закончить изучение, умея выполнять вычисления, касающиеся происходящего в пространстве, но по-прежнему с ограниченным и запутанным взглядом на это происходящее?

❒ Думают ли электроны?

В *The Quantum Self* Данах Зохар *(Danah Zohar)* рассматривает некоторые вопросы, имеющие отношение к природе сознания. Одна идея из науки о сознании предполагает, что феномен сознания возникает из сложных взаимосвязей между вещами, которые сами по себе не являются сознательными. Возникает вопрос, а каким может быть самое маленькое сознание? Может ли электрон, летая вокруг и делая эти волново-корпускулярные штучки, быть маленьким кусочком сознания?

Мы поставили вопрос Зохар не для того, чтобы прямо на него ответить, но чтобы попытаться подойти к нему с другой стороны. И, как и со всеми этими «Забавными Вещами», мы стремимся не предоставить информацию, а просто продемонстрировать, насколько тесно каждодневная работа программиста реально приближается к высочайшему искусству и глубочайшему волшебству.

Мы начнем с того, что оказываем вам любезность предположением вашей разумности. Представим, что вы изучаете синхронные процессы совместного использования ресурсов. Как хороший картостроитель, вы изучаете литературу, и размышляете над тем, что сказали другие. Вы также пытаетесь экспериментировать сами. Очень скоро вы начинаете видеть глубокие инвариантные структуры, как удачные, так и неудачные. Вы приходите к заключению, что ситуация потенциального тупика *(deadlock)* — это потенциальный тупик, неважно, замаскирован ли он сложностью. Вы также можете распознать потенциальный динамический тупик *(livelock)*, когда с ним сталкиваетесь.

Для тех читателей, которые еще не пытались это изучить, мы советуем сделать это, поскольку слишком много часов работы программиста теряются именно на таких вещах, но кратенько расскажем о тупике и бесконечном цикле. Тупик *(deadlock)* возникает, когда два (или больше) процесса останавливаются во взаимном ожидании друг друга. Например, один процесс может захватить в исключительное пользование базу данных заказчиков, а другой захватывает в исключительное пользование базу данных о складе. Затем каждый процесс пытается получить в исключительное пользование базу, которой у него нет. Ни один из запросов не может быть удовлетворен, поскольку другой процесс уже получил запрошенное исключительное пользование. Поэтому менеджер базы данных оставляет оба запроса в подвешенном состоянии, оба процес-

са засыпают до момента, когда запрос сможет быть удовлетворен. Конечно же, этого никогда не произойдет, поскольку ни один из спящих процессов не освободит базу данных, которой он уже владеет, и спать им вечно. Самый простой способ избежать такой ситуации в реальных проектах — сделать это случайно, и это не особенно мудро. Слово customer после сортировки по алфавиту стоит перед словом stock, поэтому при массовой закупке напитков стоит обратиться сначала к базе данных склада до обращения к базе данных заказчиков, даже если это означает, что возникают ситуации, где только один уже имеет доступ к базе данных склада, и поэтому приходится освобождать склад, запрашивать заказчика, запрашивать склад. Это стоит делать и пусть так и будет, либо доступ будет предоставляться одновременно либо некоторый другой нужный процесс будет попадать туда и циклы будут хорошо использоваться.

Динамический тупик *(livelock)* — это вариация тупика, когда (например) каждый процесс возвращает код ошибки вместо впадения в спячку, и пытается помочь, освобождая уже захваченные ресурсы, а затем начинает с начала своего списка покупок. Поэтому оба процесса гоняются за хвостами друг друга до тех пор, пока один или другой не сделает достаточно кругов и не получит оба ресурса сразу и не прервет кружение.

Итак, теперь вы знаете о динамическом тупике. По горькому опыту вы узнали динамический тупик, и вы распознаете потенциальный динамический тупик, когда с ним столкнетесь. Теперь представьте, что вы собираетесь встретиться с другом. Вы не уверены, в каком из двух баров вы хотите встретиться, поскольку всегда один оживленный, в то время как другой похож на морг, и вы никогда не можете сказать как все будет на этот раз. Вы не знаете, в какой из них пойдете сначала. Эти два бара находятся в разных концах квартала. Конечно, вы знаете о динамических тупиках. Вы же не собираетесь, как бегающий по планете Земля мохнатый зверек[1], впадать в динамический тупик, накручивая с товарищем круги между этими двумя барами в поисках друг друга. Когда придет время назначать встречу, вы — тот, кто скажет: «А если ты захочешь проверить другой бар, погуляй по берегу реки у квартала, так что я увижу тебя, если попаду в ту же ситуацию!»

Это вы. Это вы человек такого типа. Человек, с которым вы собираетесь встретиться, уже восхищен вашим воображением и заботливостью, и одобряет этот план.

Итак, что мы поняли и что заплели. Когда вы понимаете динамический тупик, то понимание динамического тупика — осознание того, что в мире происходят подобные вещи, и что вам приходится с ними сталкиваться — становится частью вашего сознания.

Теперь представим, что вас попросили рассмотреть информационные потоки в большой корпорации, чтобы разработать алгоритм управления сетью, оптимизирующий пропускную способность. Вы выполняете изучение как картостроитель, как это было с динамическим тупиком, и в конце концов вы испытываете озарение (проблема схлопывается), что позволяет вам увидеть элегантную, устойчивую и расширяемую стратегию управления сетью.

Теперь эта стратегия, точно так же как динамический тупик, является частью вас. Когда вы видите, что составляющие проблемы где-то повторяются, а составляющие вашей стратегии можно с очевидностью применить хоть сейчас, вы можете поклясться страшной клятвой «Это именно так!» и не сможете объяснить почему. Поэтому когда вы впоследствии излагаете свое элегантное, компактное понимание на языке программирования и заставляете его работать, то какого размера копия малой частички вас обеспечивает работу корпоративной сети 24 часа в сутки?

Это глубокий вопрос, и его не так просто понять. Чтобы увидеть, в чем тут суть, почитайте фантастическую повесть Марвина Мински и Гарри Гаррисона «Выбор по Тьюрингу»[2].

Для обладающих традиционным философским складом мышления мы в этой связи можем привести дополнительные наблюдения. Обычно сущность, такую как абстракция Платона «двойственность», нельзя увидеть непосредственно, а только как проявление: как двух собак,

[1] Тут, по-видимому, намек на бешеную собаку, которой семь верст не крюк — С.К.

[2] *The Turing Option* by Marvin Minsky and Harry Harrison

две ноги или два глаза. Обычно считается, что сущность обнаруживается некоторым образом в проявлениях, поскольку абстракция двойственности остается даже тогда, когда перед нами нет пары чего-либо. Проявление обычно, если завуалированно (covertly — в том смысле слова, как его использует Спенсер-Браун), выглядит как результат сущности.

Теперь посмотрим, что происходит при написании простейшей программы. Треугольник творчества, включающий в себя динамику проблемы, семантику системы и желание, определенно является проявлением, поскольку располагается в голове программиста, который должен действительно и физически существовать. Однако треугольник творчества остается в виде своего продукта, «вилки» («Хода Конем»), которая есть сущность построения карты (взаимосвязей) от динамики проблемы к семантике системы. «Вилка» («Ход конем»), которая есть сущность, в этом случае является образом треугольника творчества (и происходит из него), который есть проявление. Можно ли эту смену на противоположное обычно принимаемого направления онтологического приоритета соединить со странным путем, которым микросхема ПЗУ *(ROM chip)* получает специфицескую порцию отрицательной энтропии, побывав в наших руках?

❒ Тейяр де Шарден и Вернор Винж *(Teilhard de Chardin and Vernor Vinge)*

Пьер Тейяр де Шарден *(Pierre Teilhard de Chardin)*, палеонтолог и иезуит, в середине 1950-х написал книгу «Феномен человека» *(The Phenomenon of Man)*. Строя картину исходя из ископаемых находок и заполняя свойства черных ящиков частей своей модели, которые он не понимал, полуаллегорическими, полурелигиозными предположениями, он пришел к необычному взгляду на эволюцию, предлагающему предсказуемое направление будущего развития. Хотя мысли Тейяра де Шардена были очень необычны в то время, его идеи переместились в центр взгляда некоторых людей на то, что случится с технологиями и миром вообще. Работа не изменилась, просто мы получили свидетельство, что ментальная модель эволюции, предлагавшаяся там, оказалась очень близкой к истине.

Тейяр де Шарден обнаружил рост сложности форм, сначала путем агрегации атомного вещества при образовании планет (геосферы), затем на геосфере с появлением жизни (биосферы), затем развитием жизни до появления разума. Он предположил, что следующая стадия — это интеграция единиц разума и создание «ноосферы», которая будет новым действующим игроком, использующим входящие в него сознания как платформу, также как сознание использует мозг, а мозг использует молекулы. Свойства и соответствующее влияние на окружающую среду для сознаний, мозгов и молекул полностью различны, и мы можем ожидать, что новая стадия не будет исключением.

Он утверждает, что для образования агрегированной формы не нужно никакого принуждения к требуемой адаптации скоординированных состояний достаточного числа отдельных сознаний — возможно, это то, что мы видим в «однородной команде» *(gelled team)*, которая совместно использует мысленную модель происходящего. Он предположил, что окончательное слияние будет тем, что он называет «Точкой Омега» *(Omega Point)*, где скоординированное взаимодействие составляющих ноосферу сознаний подавит нескоординированное действие и возникнет новое состояние.

Он не обошелся без критиков — сэр Питер Медавар *(Sir Peter Medawar)* написал злобную атаку, сосредоточенную на изменениях в языке на стыках между убедительными доказательными частями аргументации и процессами неизвестного происхождения, помещенными между ними. Особенно Медавар возмущался использованием слова «вибрация» *(vibration)* там, где с очевидностью следовало использовать слова «совокупление» *(coupling)* или «принуждение» *(constraint)*, которые не так возмущали бы Медавара. Проблема в том, что картостроителям приходится работать с вещами, которые они не понимают, поэтому местами язык неизбежно становится немного расплывчатым. Именно отсюда приходят новые теории (и можно сказать,

что программа — это теория программиста относительно проблемной области). К сожалению, этот вид языка приводит некоторых людей в состояние сумасшествия, даже не смотря на то, что большинство хороших вещей временами нарушают каноны, хорошо если только в форме высказывания о том, что вещи «хотят» делать то или это, а то и наполняя неизвестный механизм антропоморфизмом, что просто глупо, будучи применено к электрону, не говоря уже о муравье, демонстрирующем «невыразимый дух», но это из некоторых предположений более приемлемо.

В качестве последнего примера, прислушайтесь к Ньютону, который описывал явления, которые он мог наблюдать, но которые не находили места в его собственной картине физики, и механизм которых он не мог объяснить (и которым был посвящен целый раздел в этой работе)...

Конечно, это хорошо известный факт, что Ньютон проводил значительную часть своей жизни «путаясь с теологией»! Вернор Винж — профессор математики в Университете Сан Диего и один из лучших писателей-фантастов. В своей знаменитой работе «О сингулярности» *(Singularity Paper)*[3] и научно-фантастические книги «Сквозь время» *(Across Realtime)* и «Огонь над бездной» *(A Fire Upon the Deep)* он предположил, что разум обитателей этой планеты будет возрастать как путем генетического усовершенствования человеческого мозга, так и расширения его возможностей с помощью «железа» и построения новых компьютерных архитектур, включающих человека. После этого сети и новые реалии, происходящие из дальнейшего развития, создадут мир, который мы не в состоянии даже представить с позиций нашего нынешнего состояния.

Есть поразительная схожесть идей Тейяра де Шардена и Винжа, только, помещая эволюцию в быстрый реактор программного обеспечения, мы сжимаем миллионы лет органической эволюции, которые нужны (по Тейяру де Шардену) для создания ноосферы, в тридцатилетие (о чем говорит Винж).

Но не принимайте эти наши слова на веру — проверьте (изучите), посмотрите, дает ли это новую перспективу тому, что происходит с миром, когда вы программируете, и, главное, думайте об этом лишь для упражнения!

❒ Общество разума

Марвин Мински *(Marvin Minsky)* в «Обществе разума» *(The Society of Mind)* предположил, что феномен человеческого разума возникает из взаимодействия множества неразумных обрабатывающих агентов, которые действуют в мозгу как со-процессы, каждый со своими собственными триггерами и программами действий. Агенты затем соединяются и подвергаются арбитражу посредством «сетикета» *(nettiquette)*, что позволяет им определять направление деятельности организма как целого. Когда мы чувствуем себя проявляющими свободу воли в потакании нашим прихотям, мы на самом деле просто следуем решению, которое уже возникло в коллективе агентов. Эта модель определенно имеет свои привлекательные стороны и дает основу для стимулов, для удовлетворения которых мы применяем творчество и интеллект, но, как кажется, не дает полезного описания самих творчества и интеллекта. С этими обобщенными средствами познания, мозг, как кажется, должен использоваться как управляемый прибор распознавания образов общего назначения, внутренние представления которого подсоединяются к сенсорным компонентам опосредованно (косвенно), по крайней мере так, чтобы абстрактное и конкретное можно было рассматривать с одинаковых позиций.

Соотношения между моделью общества разума (для познания и мотивации) и средствами общего назначения отражают отношения между тем, что мы называли стратегиями паковки и картостроения, и существует дальнейшая параллель с двумя простыми подходами к управлению данными при проектировании компьютерных систем.

[3]Используйте WWW. Например, `http://www-rohan.sdsu.edu/faculty/vinge/misc/singularity.html`

Хэширование действует путем абстрагирования (извлечения) некоего ключа из данных — возможно, беря 20 символьное имя из поля и складывая численные значения всех символов. Это число затем может использоваться для индексирования таблицы и нахождения полной записи. Реальные хэширующие алгоритмы разрабатываются так, чтобы максимизировать разброс получающихся из типичных входных данных чисел, и должны учитывать ситуации, когда хэш уже переполнен, сохраняя ссылки на несколько записей, так что извлечение включает проверку полного ключа для каждой записи. В простых ситуациях хэш часто очень эффективен, и напоминает паковку, где некоторая абстракция ситуации используется для извлечения «подходящего действия». Как представляется, при паковке плохо обрабатываются переполнения хэша. Они даже не будут замечены, пока один или несколько участников не обнаружат явные, быстро проявляющиеся *(short term)* потери от «подходящего действия». За этим последует «спор», в котором паковщик укажет на один путь абстрагирования ключа из ситуации и утверждать, что это «именно тот случай», в то время как другой укажет на другой алгоритм хэширования и будет утверждать, что нет, это «тот случай». Это не продуктивно и показывает изъян описанной стратегии при превышении некоторого уровня сложности проблемы, где мы пытаемся буквально впихнуть слишком много вариаций в слишком маленький хэш и не развиваем навыки выполнения при необходимости большого числа проверок полного ключа.

Объектные модели позволяют хранящимся в компьютере структурам данных усложняться и изменяться, подчиняясь семантике моделируемых объектов. Вид всей структуры данных в процессе обработки может полностью измениться, а извлечение данных всегда остается «естественным» в том, что данные всегда там, где они «должны» быть — они все непосредственно связаны с соответствующими другими данными. Следовательно, там нет сложности, внесенной чужеродным алгоритмом типа хэширования, которую требуется устранить с помощью чего-то еще, типа дополнительного сравнения ключей. Выше определенного уровня сложности объектные модели более удобны чем хэш, но, несомненно, их труднее реализовать. Причина, по которой мы можем использовать их сегодня за малую цену, в том, что мы получаем большую поддержку от языков для описания объектов, и у нас есть операционные системы, управляющие свободной памятью. Объектные модели представляются настолько похожими на стратегию картостроения, что мы описываем картостроение как попытку сконструировать жизнеспособную объектную модель проблемной области.

Эти параллели между функциональной (общество разума и распознавание шаблонов), субъективной описательной (паковка и картостроение) и вычислительной (хэширование и объектное моделирование) моделями сознания предполагают, что вполне может существовать нейрологическая корреляция с описанными нами стратегиями картостроения и паковки. Мы определенно знаем, что ранняя стимуляция детей приводит в увеличению роста нейронов и их переплетению в мозгу ребенка, и это коррелирует с большей «интеллектуальностью» *(intelligence)* (что бы это ни значило) во взрослой жизни. Что бы ни было «интеллектуальностью», навыки познания и решения проблем, что тестировалось, слабо проявлялись на тейлористском паковочном рабочем месте, где основная идея заключается в деквалификации и ограничении поведения.

Вероятный вопрос в начале информационной эры — это: «Какая часть вашего мозга предназначена для использования на работе?»

❒ Картостроение и мистицизм

В самом начале, мы посмотрели на два разных пути решения проблем. Паковка характеризовалась как социально обусловленная привычка обрастать «пакетами знаний», которые определяют «подходящее действие», а не исследовать или переконфигурировать соотношения между пакетами знаний. Эта стратегия деградирует в практику подгонки реальности под известные пакеты и возложение ответственности на фортуну, когда что-то происходит не так. Картостроение, наоборот, включает в себя инвестирование в построение внутренней объектной модели

мира по мере его восприятия и получение рычагов путем обнаружения глубокой структуры. Картостроение можно совершенствовать изучением методик, которые помогают исследованию концептуальных пространств и помогают действительно увидеть происходящее перед глазами, распознавая в происходящем глубокие структурные паттерны. Картостроители могут гибко реагировать и являются единственными людьми, которые в состоянии предложить новые подходы. Они могут учиться гораздо быстрее паковщиков, и пока они ищут глубокую структуру, они смотрят на нее как на еще неразгаданную тайну. Опыт картостроителей и паковщиков в совершенно одинаковых обстоятельствах может оказаться совершенно различным.

Картостроение — естественное состояние человека, и каждый в душе — картостроитель. К сожалению, сообщества по всему миру развили альтернативу, которую мы называем паковкой, примерно в то же время, когда появилось земледелие, т.е. около 6000 лет назад. Она не могла возникнуть раньше — доаграрный паковщик, борющийся с диким животным на охоте, мог тяжело поплатиться, если бы обращался к небу и кричал, что животное плохо следует процедуре!

Альтернатива включает людей, убежденных, что хорошая жизнь заключается в следовании предварительно заданным процедурам и подавлении любых альтернатив. Она должна была принести преимущества построенным вокруг выращивания урожая новым сообществам, когда на полях должна быть проделана значительная скучная работа, и если дела идут тяжело, то единственное что можно делать — это трудиться, трудиться, трудиться до сбора урожая. Таким образом, паковка включает социализацию (приобщение) молодежи к мировоззрению паковки и построению общества, где реальность состоит в подходе паковщика и наборе пакетов знаний, и ничего больше. Любой человек, предполагающий, что могут быть другие способы взглянуть на вещи, окажется в оппозиции с каждым членом общества, в котором он оказался, в оппозиции к неэффективности общества паковщиков и ритуализированному образу жизни, даже когда социальные обстоятельства дают сбой. Этому раскольнику могут быть приписаны загадочные свойства, типа магической силы, если ему повезет реализовать некоторые идеи здравого смысла, либо безумность, если окружающие паковщики вовремя задумают саботировать его отклонения. Большинство людей не смогут поверить, что может существовать какой-то другой, отличный от паковки, подход к мирозданию.

Сегодня в развитом мире спрос на работу в три погибели невелик, но есть огромная потребность в сознательных людях для создания новых программ автоматизации. Только естественные картостроители обладают способностями распознавания образов (структур), существенных для написания компьютерных программ.

В течение большей части времени своего существования стратегия паковщика, вероятно, хорошо послужила своим пользователям, поддерживая порядок на полях и на первых фабриках, обеспечивая условия для выполнения простого ручного труда, который был необходим для выживания. В тех условиях это было важнее, и хотя литературе можно было бы уделить побольше внимания, со времен изобретения печатного пресса всегда было гораздо больше поэтов, чем печатных прессов, поэтому, вероятно, эта способность стоила не много. Но в начале 20 века эра индустриализации сделала паковку опасно неэффективной стратегией. Мы просто были слишком богаты. Наши машины позволяли нам делать вещи, о которых не могли мечтать предыдущие поколения, и нам нужно было понимание, чтобы их использовать. Попав в сети мировоззрения паковщиков и обладая неприменимыми в индустриальном обществе пакетами знаний, Европа была разорвана на части, поскольку миллионы пошли воевать, вооружившись двигателями внутреннего сгорания, гусеничными машинами, колючей проволокой, пулеметами, горчичным газом, аэропланами и другим оружием, что извращало доиндустриальный пакет знаний «Война — это продолжение дипломатии другими средствами» в любом разумном дипломатическом определении целей на основе одной лишь цены.

Теперь все начало трещать по швам. Мы достигли мечты веков, исчезла потребность работы миллионов, освободив их время на то, чтобы делать то, что они пожелают. Пока мы называем это безработицей, и по-прежнему занимаем миллионы не требующей большого напряжения ума работой, просто манипулирующей предметами аграрной экономической системы. Существует

так много такой непродуктивной работы, что ее трудно увидеть сразу, но каждый кассир в супермаркете, кассир в банке, билетный контролер, сборщик налогов, бухгалтер и многие, многие другие на самом деле заняты непродуктивной работой. Лишь небольшая прослойка населения выполняет работу, действительно необходимую для поддержания нашего материального благосостояния, и мы до сих пор убеждены, что живем в нужде!

Даже видимые сейчас противоречия, гораздо худшие, чем когда-либо в истории (паковка всегда приводит к определенной глупости, когда приходится принимать решения в необычной обстановке), не могут указать путь обратно к картостроению на языке паковщика. Можно сказать, что предотвращение обсуждения картостроения — это развитая за тысячелетия функция языка паковщика! Поэтому в прошедшие времена возврат к картостроению должен был быть действительно очень редким событием. Если эффективность или приемлемость картостроительного подхода к проблеме становится объектом обсуждения, то мнение большинства паковщиков будет всегда таким, что не важно, что результат был получен, поскольку он был получен «неправильно».

Только сегодня у отдельных людей есть реальная возможность поупражняться в картостроении и получить так важную для обучения реалистичную обратную связь. Это происходит потому, что только картостроители могут программировать компьютеры. Если человек, все еще с мировоззрением паковщика, следует процедуре и переводит требования, то результат скорее всего будет ужасным. В этот момент он мог бы материть компилятор, операционную систему или пользователя, но, возможно, следовало бы просто осознать, что компьютер на самом деле с абсолютной точностью отражает то, что ему сказал человек. Поэтому человек может принять, что он и только он сам, должен понять динамику проблемы и семантику системы. Здесь начинаются бессонные ночи, открывающие дорогу к реальному мышлению, а не исполнению бессмысленных ритуалов, в чем заключается паковка, и что большинство паковщиков считают нормальным.

С этой перспективы интересно посмотреть на некоторые нити предыдущих размышлений, которые пытались описать опыт картостроения в культуре, где нельзя просто сказать «Программа работает!», а нужно обладать достаточно сильными аргументами в свою пользу на языке, который картостроители могут понять в терминах совместно используемого субъективного опыта игры с представлениями реальности в чьей-то голове, пока они не выправятся настолько, что станут полезными, и который паковщики вообще не могут понять. Возможно, это не удивительно, что многим великим программистам интересны эти нити предыдущих размышлений...

Мы уже обсуждали природу алхимии как внутреннего путешествия, изменяющего взгляд оператора на мир — базовую технику картостроения. Традиции алхимии, скорее всего, распространились по Европе из Мавританской Испании благодаря таким людям, как Роджер Бэкон.

В книге «В поисках чудесного» П. Д. Успенский описал беседы с Г. И. Гурджиевым, которые происходили в России в 1915 году. Успенский приводит слова Гурджиева, странной и влиятельной фигуры, утверждавшего, что провел многие годы в изучении мистических традиций:

> В целом, есть четыре возможных для человека состояния сознания... но обычный человек... живет только в двух самых низших состояниях сознания. Два высших состояния сознания недостижимы для него, и хотя он может сталкиваться со вспышками таких состояний, он не способен понять их, и он оценивает их с точки зрения тех состояний, в которых он обычно находится.
>
> Эти два обычных (и низших) состояния сознания — это, во-первых, сон, или, другими словами, пассивное состояние, в котором человек проводит треть и даже часто половину своей жизни. И, во-вторых, состояние, в котором человек проводит остальную часть своей жизни, в котором он ходит по улицам, пишет книги, рассуждает о возвышенных материях, участвует в политике, убивает других, которое он считает активным и называет его «ясным сознанием» или «пробужденным состоянием сознания». Термин «ясное сознание» или «пробужденное состояние сознания», как кажется, дан в шутку, особенно когда вы

осознаете, что ясное сознание действительно должно существовать, и каково на самом деле состояние, в котором человек живет и действует.

Третье состояние сознания — это само-вспоминание или само-осознание или сознание собственного бытия. Обычно считают, что у нас есть это состояние сознания или что оно может у нас быть, если мы этого захотим. Наша наука и философия упускают факт, что мы не обладаем этим состоянием сознания и что мы не можем создать его в нас самих по желанию или одному лишь решению.

Четвертое состояние сознания называется объективным состоянием сознания. В этом состоянии человек может видеть вещи такими, какие они есть. Вспышки этого состояния сознания также возникают в человеке. В религиях всех народов есть указания на возможность состояния сознания этого типа, что называют «озарением» *(enlightenment)* и разными другими именами, но которое не может быть описано словами. Но единственный правильный путь к объективному сознанию лежит через развитие самосознания. Если обычный человек искусственно введен в состояние объективного сознания и затем возвращен обратно к своему обычному состоянию, то он ничего не будет помнить и будет думать, что в это время он потерял сознание. Но в состоянии самосознания у человека могут проявляться проблески объективного сознания и он может их запомнить.

Четвертое состояние сознания у человека означает совершенно другое состояние бытия; это результат долгой и трудной работы над собой.

Но третье состояние сознания составляет естественное право человека, раз он есть, и если человек не обладает им, то только вследствие неправильных условий своей жизни. Без всяких преувеличений можно сказать, что в настоящее время третье состояние сознания появляется в человеке только в виде кратковременных вспышек и что его можно сделать более или менее постоянным только посредством специальных тренировок.

Очень похоже, что паковка соответствует второму состоянию, картостроение третьему состоянию, а то, что происходит при разрешении проблемы — это четвертое состояние. К счастью, описанные Гурджиевым трудности уменьшены сегодня добрыми нанимателями, которые желают платить нам высокие зарплаты, чтобы мы целый день сидели перед обучающими машинами. Если мы сможем приспособить к работе язык третьего уровня вместо языка второго уровня, то сможем отплатить этим добрым людям написанием для них множества прекрасных компьютерных программ.

По справедливости нам следует сказать, что хотя значительная часть описанного в книге «В поисках чудесного» непосредственно достижима в терминах модели картостроитель/паковщик, многое — нет. Существует также система «Водородов» *(Hydrogens)*, которая, оказывается, вообще не имеет отношения к физике элементарных частиц, которая предположительно описывает структуру пространства. Но она, однако, вносит в сознание фрактальную структуру и атракторы и претендует на то, чтобы стать видением мира, которое позволяет личности восхищаться необычайно расширившимися возможностями посредством «освобождения себя от общих законов» в не подчиняющейся сокращенному описанию форме. Мы не можем найти у этого ни головы, ни хвоста, но посмотрев на картостроителей и паковщиков на работе, только обнаружив их сидящими за компьютерами, мы начинаем задумываться, что может быть правда стоит заняться... созерцанием.

В Исламе есть концепция двух Кораном. Есть написанный Коран, записанный Пророком под диктовку Бога, и обнародованный Коран, который являет собой мир вокруг нас, созданный Богом. И обязанность каждого человека, который получает удовольствие от роскоши совершенствования себя, проводя свое время в изучении этих творений Бога, состоит в том, чтобы продолжать свои поиски доступным каждому способом. Возможно, это прекрасная идея — позволить ученику убедиться в своем невежестве, затеяв безнадежное прямое соревнование с Богом, которое у любого закончится поражением, и затем научить тому, что духовный долг

ученика состоит в уменьшении этого невежества, и этой идеей Ислам может внести некий полезный вклад в нашу область. Ведь все мы знаем, откуда к нам пришли алгебра и алгоритмы!

В Китае есть древняя даосская традиция, которая также страдает от коммуникационной проблемы — «Дао дэ цзин» *(Tao Te Ching)* начинается с:

> «Дао, которое можно произнести — это не вечное Дао.»

Даосы сосредоточены на нахождении глубокой структуры в глубокой структуре и получении максимального воздействия «правильным действием». Даос не ковыляет «по течению», у него есть ясное (и, следовательно, непротиворечивое и правильное) понимание того, что он желает осуществить, и ищет правильную точку приложения воздействия, глядя на структуру взаимосвязанных явлений, которыми он заинтересовался. Правильное действие тогда может быть легким толчком в нужное место! Как и во всех мистических традициях, у даосов просто нет времени на превознесение чего бы то ни было.

Когда дао встретилось с буддизмом, появился дзен. С позиций картостроителя/паковщика, дзен можно описать как специальный набор картостроительных методик и строительных блоков, которые позволяют исследование глубокой структуры, что часто может идти в разрез с интуицией того, кто поражен (болен) мировоззрением паковщика. Когда дзен спрашивает «Как звучит хлопок одной руки?», то он говорит, что хлопок нельзя обнаружить ни для левой руки, ни для правой, но только во взаимодействии между ними. Многие великие программисты, особенно работающие над искусственным интеллектом, любят ублажать себя дзен-притчами (Zen koans)[4].

Алхимия, дао и дзен — это все мистические техники, которые вообще не несут в себе сверхъестественного компонента. Они обсуждают состояние ума практикующего и таким образом усиливают имеющиеся способности, освобождая их от закаменевшей корки стереотипов, создаваемых при паковке. Как сказала Кейт Буш *(Kate Bush — еще один фаворит у программистов)*,

> Don't fall for a magic world
> We humans got it all
> Every one of us
> Has a heaven inside.
> * * *
>
> Не гнись пред мира волшебством.
> Мы, люди, получили его целиком.
> Каждый, бережно храня,
> Таит небеса внутри себя.

Но несмотря на свою нацеленность на практику, при обсуждении субъективного опыта картостроителя все они вынуждены использовать аллегорический язык. Древний аллегорический язык, который не имеет никакого смысла для паковщиков, легко путают с религией, и в 19 веке некоторые люди пытались буквально интерпретировать древние описания происходящего.

Философ Фридрих Ницше *(Frederic Nietzsche)* проделал большой путь в преодолении коммуникационного барьера картостроитель/паковщик и вызвал сильное волнение в своем местном обществе паковщиков тем, что заявил, что Сверхчеловек *(Superman)* не ограничен простыми законами. Он умер в психушке, но успел оказать значительное воздействие на философию.

Ницше интересовался разницей между человеком, который реализовал свой потенциал, и тем, кто живет в общественной реальности паковщиков. Ему на самом деле не нравились хныкающие, завистливые, злобные, неумные обыватели, которых он противопоставлял своему Сверхчеловеку. Вновь он вызвал интерес у людей, занятых Тотальным Управлением Качеством (TQM).

[4] См., например, THE JARGON FILE — `http://www.jargon.org` — С.К.

Зигмунд Фрейд *(Sigmund Freud)* опросил в Вене большое число женщин среднего класса и создал оригинальный психоанализ, который включал своеобразный *(idiosyncratic)* взгляд на мотивацию и импринты *(preoccupations)* человека. Не все его последователи поддержали эти импринты, но осталась его концепция «отчуждения» *(alienation)*. Это такая ситуация, когда человек играет роль, вместо того, чтобы вести себя «подлинно» *(authentically)*, и является, таким образом, отдалением, отчуждением от своих товарищей, которые также играют роли. Возможно, мировоззрением этого человека становится показная, фиктивная реальность, так что он становится отчужденным от самого себя и больше не может идентифицировать и осознавать свои собственные желания и заботы.

Сорен Кейркегард *(Soren Keirkegaard)* озаботился тем, как мы можем вообще что-то знать в том сумасшествии, которое нас окружает, и создал философскую позицию экзистенциализма, где величину и значение действия может оценить только действующее лицо, на основе имеющейся у него информации. Этот вид социального релятивизма определенно выделяет личность из группы, в которой можно избежать условия самоцензуры картостроения, и человек может носить черное и плодиться. Он, однако, коварно предполагает, что нет такой вещи как объективная, внешняя реальность (или, если она есть, то не имеет значения, поскольку никто не знает, что это такое). Это лживо до абсурда, поскольку означает, что стоять на углу и корчить рожи — это такое же дельное занятие, как облегчение ужасных страданий или строительство домов, если этот идиот говорит, что они есть. Этот аспект экзистенциализма противоречит опыту картостроителя, который приводит картостроителей к убеждению, что существует великой искусности внешняя реальность, и хотя никто из нас еще не представил ее себе полностью, она прекрасна. И если один из нас открывает явление, то оно очевидно подтвердит совместимость с любым другим уже открытым нами явлением. В этом смысле, внешняя реальность важна, даже если она не постигаема.

За Кейркегардом последовал Жан-Поль Сартр *(Jean-Paul Sartre)*, который написал условие членства в обществе, которое само себя отрицает, и Лэинг *(R. D. Laing)*, который привнес идеи экзистенциализма в психиатрию, где он наблюдал целые семьи, тайно сговорившиеся поддерживать одного человека, которому был поставлен диагноз «шизофреник» *(schizophrenic)*, в условиях полного неведения, которые тратили значительную часть своих ресурсов, финансовых и временных, на защиту реальности паковщиков своих несчастных семей от угрозы со стороны оказавшегося среди них картостроителя. С точки зрения картостроителя, «пациент» находится в центре сложной сети мистификации и насилия, распределенной по всей семье, которую необходимо разорвать, если они хотят найти счастье. Точка зрения паковщика в том, что Леинг «осуждает» родителей за «навязанную болезнь». Это означает, что работа Лэинга не встречает поддержки в клинической ситуации, где эффективная сила находится в руках родственников пациента (или он не был бы пациентом). Однако, коллега Лэинга, Мелани Клейн *(Melanie Klein)*, которая выдвинула множество собственных идей, поработала в индустриальном секторе, и идеи экзистенциализма, выдвинутые Клейн, в промышленной психологии по-прежнему интересны.

Недавно Питер Сенге *(Peter Senge)* из Sloan Business School при MIT написал о Системном мышлении *(Systems Thinking)*, которое является подходом к решению проблем на основе формирования мысленных моделей и использовании таких вещей, как обратная связь.

Когда мы попытались понять, почему некоторые люди так хороши в программировании, мы знали, что ответ будет очень интересным, но мы никогда не предполагали получить простую модель, которая могла бы также выявить универсальную тему у такого количества мистических и философских школ. Вероятно, так сделать было очень ценно, поскольку с таким количеством совершенно различных способов сказать об одном предмете ситуация для любого, кто безуспешно пытается пробить мышление паковщика, вместо того чтобы остановиться и изучить несколько предметов, очень запутанна. Чьи-нибудь друзья могут даже подумать, что он впал маразм *(wierdo)*!

❒ Картостроение и ADHD (Болезнь гиперактивного дефицита внимания)

Говорят, что существует расстройство, называемое болезнью гиперактивного дефицита внимания *(Attention Deficit Hyperactivity Disorder — ADHD)*, которое поражает 3% населения. Страдающие от нее, как можно предположить, влачат трудную жизнь, она им мешает, но принимая соответствующие лекарства и получая поддержку, они могут надеяться на некоторую интеграцию в общество.

В терминах модели картостроитель/паковщик, мы подозреваем, что ADHD может быть просто результатом естественного детского картостроения, когда дети, будучи умнее своих ровесников, становятся все более и более холодны к окружающим паковщикам, по мере того как они все упорнее и упорнее думают, пытаясь понять что от них хотят учителя-паковщики, ровесники и родственники, в то время, как взрослые считают этих детей непослушными или больными, поскольку им не видна очевидная дисфункция, требующаяся для того, чтобы сидеть и непрерывно выполнять одни и те же простые, бессмысленные, тупые действия, пока не станешь себя вести как стадное животное.

❒ Как развивался этот подход

Создание этой работы — само по себе пример картостроения, поэтому будет показано, как была собрана вся картинка.

Необходимость этой работы стала ясна после взгляда на то, что произошло, когда в компьютерную индустрию был внедрен стандарт ISO 9001. Оказалось, что в лучшем случае, он обеспечивал гарантию, что сертифицированная по ISO 9001 организация была по крайней мере выше уровня «Лаурел и Харди» *(Laurel and Hardy)*, где кто-то смог потерять исходный код работающих у заказчиков программ, но не давал ничего позитивного для улучшения способностей программировать у занимавшихся программированием людей. Несколько лет назад был инцидент, когда работникам организации, которая производила программное обеспечение для управления огромными мельницами, пришлось посетить заказчика под каким-то предлогом, вынуть ПЗУ *(ROM)* и скопировать его для того, чтобы дизассемблировать содержимое и обеспечить поддержку программы. Никто из тех, кто оказывался в такой ситуации, никогда такое не забудет. Поэтому ISO 9001 был хорош, но реальная работа нуждалась в «инженерном чутье» и «здравом смысле», которые упоминались во всех лучших документах — составляющих, которые мы не могли получить подражая автозаводам.

Но затем мы увидели, что в некоторых организациях, существовала необъяснимая, но почти религиозная вера в то, что сведением всего к упрощающему процедурализму можно достигнуть совершенства, и что метры книжных полок заключают в себе необходимые простые процедуры. Окруженное процессом ограниченное обдумывание того, что нужно делать, могло быть заброшено или лучше уничтожено и каждый мог бы бегать кругами, будучи «профессионалом», не достигая на самом деле вообще ничего. В те старые времена, даже нищие организации реально имели исходный код достаточно длинный, чтобы продавать его заказчику и платить арендную плату!

Нам было нужно выяснить, в чем состоит настоящее программирование, чтобы противостоять как негативным последствиям плохо применяемого ISO 9001, так и как важный ингредиент, дополняющий хорошо применяемый ISO 9001. Основываясь на том, что в ISO 9001 было нечто упущено из описания рабочего места, и в честь сюрреалистического объявления лондонской подземки, эта работа получила в тот момент рабочее название «Брешь в сознании» / «Напоминаем о перерыве в движении» *(Mind the Gap)*.

Мы начали с наблюдения, что есть некоторые программисты, которые гораздо лучше большинства, и что они согласны между собой, в том, кто они. Они могут разговаривать друг с

другом о программировании, и хотя они часто не приходят к согласию в количественных оценках, они часто соглашаются в главном.

Конечно, прямо с самого начала, мы столкнулись с трудностями описания на «языке управленцев» того, что мы увидели, разговаривая с великими программистами. Мы проводили много времени споря в кружках, пытаясь поместить двумерное создание в третье измерение, показывая его последовательностью шагов, каждый меньше предыдущего. В целом, конечно, представление было с изъяном, поскольку не важно, насколько тонким был срез шага, он по-прежнему оставался трехмерным объектом, недоступным для двумерной твари. Но тогда мы этого не знали.

В то же самое время мы смотрели на образ мыслей великих программистов изнутри, анализируя наш собственный процесс мышления в процессе работы, и наблюдая за другими. Это дало гораздо больший прогресс, и мы очень быстро идентифицировали «Программиста-Ремесленника» *(Artisan Programmer)* как фигуру, больше напоминающую мастера-ремесленника прошлого, чем современного рабочего на конвейере.

Мы также интересовались лежащей в основе акта программирования когнитивной нейропсихологии, но вряд ли вообще что-то получили. Мы не смогли найти много работ, связывающих субъективный опыт с его платформой, и одна из областей, на которую нам особенно хотелось взглянуть, это отличия в программировании у разных полов, оказалась особенно редко исследуемой. Нейропсихологи сообщали нам приватно, что различия полов в мышлении иногда очень трудно исследовать вследствие соображений «политкорректности». Однако, при отсутствии полезных психологических исследований, мы делали попытки сконструировать работающие определения субъективного опыта. Это случайно привело к мысленному эксперименту с простейшей программой, который окончательно похоронил внешний процесс, и заставил нас сконцентрироваться на субъективном опыте.

Период с весны 1992 до осени 1995 г. мы провели в беседах с программистами, обсуждая и обдумывая то, чему мы научились. Мы должны были испытать сотни способов «рассказать историю», каждый из них погибал на языковом барьере. Однако, мы обнаружили, что некоторые немногие вопросы возникали вновь и вновь, в одном месте за другим, и на эти вопросы имелись правильные ответы. Они были сформулированы как Принципы Проектирования. Мы также обнаружили, что некоторые идеи и истории, которые нам рассказали великие программисты, оказывали очень положительный эффект на новичков, когда мы их рассказывали. Этот материал также был включен в Камень.

Затем осенью 1995 г.экстраординарная работа Фредерика Кантора *(Frederick W. Kantor)* в области физики «Информационная механика» *(Information Mechanics)* дала главное вдохновение. В ней Кантор отбросил все костыли и попытался построить целостную картину физики чисто на концепциях информации. Вероятно, решением нашей проблемы стало бы отбрасывание всего того языка, который, как мы знали, не работал, и использование языка, который, как мы знали, работал. Возможно, посредством такой онтологической строгости, мы смогли построить самодостаточную картину, даже если она шла в разрез «главному направлению» реальности, что мы видели достаточно отчетливо.

Очень быстро мы сфокусировались на движении сознания и увидели связь с алхимией. Быстро проявились связи с другими мистическими традициями, и мы пытались использовать инспирированный мистикой язык к новичкам, и объяснили цикличность алхимических путешествий. Мы обнаружили, что мы можем улучшить производительность программистов лучше прежнего, но мы все еще не могли объяснить на обычном языке — почему. С этого момента мы называли наш проект «Развернутое сознание» *(Deployed Conciousness)*.

Летом 1997 мы столкнулись с ADHD, и сразу распознали в чертах характера детей с ADHD великих программистов, с которыми мы разговаривали. Мы могли видеть, что делали дети, но нам казалось совершенно очевидным то, чего не видели психологи и другие профессионалы, и они могли бы научить их реальным вещам типа теории чисел вместо того, чтобы пичкать амфетаминами, чтобы те сидели тихо и выполняли бессмысленную, в духе школы паковщи-

ков, «работу». Это было большое потрясение, но оно дало нам важный ключ: здесь должно присутствовать некоторое ослепление сознания, что означает, что эти психологи были просто не способны понять детей и не могли даже осознать, что происходит нечто, что они не могут понять.

Это показало нам, почему существует проблема языка — восхитившись, когда это проявилось, нам пришлось заключить, что все наши коллеги действительно находились в одном из двух (и только двух) возможных состояний, и мы могли описать различия между ними. Мы быстро это записали, предполагая некую лежащую в основе нейрологию в виде черного ящика, возможно включающую сдвиг в стратегии обработки, раз некий ресурс или что-то другое достигло критического уровня и сделало переключение на другую стратегию оптимальным решением, и распространили среди друзей, которые с самого начала обсуждали с нами эти темы.

Мы получили много откликов, в большинстве случаев положительных, но один комментарий оказался критическим. Нас спрашивали, не могли бы мы найти какую-нибудь связь между нашей работой и ME *(aka CFIDS)*, истощающее послевирусное расстройство *(debilitating post-viral disorder)*, которое разрушило жизни многих активных творческих людей. Многим картостроителям казалось, что они знают нескольких людей, пострадавших от ME, и мы составили список. Да, это все были активно думающие люди, а не наглые неинтеллектуальные «юппи», о которых говорили бы, что они подхватили «грипп юппи» *(yuppie flu)*. Но далее, они все были думающими людьми, чьи чрезвычайно вежливые характеры вынуждали их отвечать на действия величайшей глупости, набрасывающейся со всем презрением, на какое может мобилизоваться паковщик, унынием, а не, скажем, гневом или презрением. Побейте обезьяну палкой подольше, и у нее повыпадают волосы. Это психологический эффект от постоянной психологической жестокости. ME проявлялась в период, когда фундаментализм паковщиков сокрушил все в мире, приводя к гигантской глупости и жестокости. ME с успехом могла бы быть следствием. Но почему только у вежливых? Они все были очень активны, они бы покрыли черепицей дом, потому что стало тепло, или объездили бы Канаду на велосипеде, чтобы отпраздновать ее открытие, хотя все они были мечтателями. Это не могло быть мечтательством, поскольку мы все мечтатели *(daydreamers)*... и упал пенс.

Различие между паковщиками и картостроителями заключается в том, что паковщики социально обусловленно подавляют свои естественные способности строить мысленные модели мечтая (фантазируя), и вместо этого скатываются к механически заучиваемым процедурным, ориентированным на действие реакциям. Мы могли отбросить нейрологические черные ящики, и просто сказать «мечтание» *(daydreaming)*, чтобы навести мост к обычному языку. И тогда эмпирическая работа, как и понимание природы проблемы языка, все стало на место.

Так завершился путь нашего исследования. Когда мы начинали, мы не знали, что мы найдем, но мы чувствовали уверенность, что это будет полезным. За первых три с половиной года мы помогали нескольким новичкам разрабатывать, но мало чего достигли еще. Мы собирали материал и искали закономерности.

Вся работа заняла около шести лет, но это неплохо для глубокого результата. Если бы мы не начали, мы никогда бы не достигли результата, который теперь мы предлагаем вам прочитать за гораздо более короткий срок, чем шесть лет!

❒ Сложность космологии

Этот повторяющийся опыт больших инвестиций картостроителей в стратегию познания, когда они не знают, окупятся ли эти инвестиции. Почему так? Есть ли у нас зацепки к глубокому ответу на этот вопрос?

Одна, вероятно, заключается в том, что касается сложностей космологии. Мы знаем, что начиная с самого первого момента, в пространстве возникала структура. Мы знаем, что физические константы в природе просто справедливы для атомов, звезд, планет, сложных химических

соединений. Нами не доказано, что возникновение жизни было неизбежно, но как мы теперь знаем, поместите что-нибудь в банку, дайте толчок в правильном направлении, и получите самоорганизацию.

Мы могли бы принять подход, развивающий идеи Тейяра де Шардена и Вернора Винжа, которые обсуждались ранее, и задаться вопросом, а что если наше поведение в увеличении сложности мироздания путем написания программ — это просто эволюция, действующая в космическом масштабе, вновь увеличивающая скорость изменения. Тогда мы могли бы сказать, что получаем выигрыш в двух направлениях — по-первых потому, что сложность, которую мы видим, была построена на основе более простых уровней, поэтому очерчивая наши произвольные границы вокруг системы, мы будем часто находить возможности для снижения сложности в пределах этих границ, и во-вторых, потому, что добавляя больше сложности, мы просто делаем то, что происходит естественно.

Создание (построение) сложности вполне могло бы быть естественной осью времени, как и энтропия, и, таким образом изначально достижимо в этом мире, из соображений, которые мы еще не понимаем. Мы вполне можем не знать, к чему это ведет, но, вероятно, у нас есть шанс узнать это до того, как мы туда доберемся.

❒ Дилемма заключенных, свободное программное обеспечение и доверие *(The Prisoners' Dilemma, Freeware and Trust)*

Дилемма заключенных интенсивно изучалась как модель стратегии первого удара ядерными баллистическими ракетами. Там два заключенных содержатся отдельно, и обоим предлагается следующая сделка: «Если никто не сознается, вас обоих освободят. Если вы оба сознаетесь, то получите большие сроки. Если сознается только один, он получит маленький срок, а другой получит срок в два раза больше».

Дело в том, что пока я могу быть уверенным, что ты не хочешь сознаться, лучшее, что я могу сделать — это сознаться и отсидеть короткий или длинный срок, но при этом избежать двойного срока. Ты думаешь также. Поэтому, пока мы оба уверены (вспомните «уверенность» — «certainty» старых паковщиков), чего может и не быть, мы оба закончим тем, что будем сидеть длинный срок, хотя могли бы не сидеть вовсе.

Этот результат угнетал во время холодной войны, когда значительное стратегическое преимущество могло быть достигнуто упреждающим ударом. В то время, как теоретики игр настаивали на том, что двойной обмен ударами неизбежен, человеческая раса перед лицом полного уничтожения оказалась способной вести себя рационально и избежать обмена ядерными ударами вообще, избежав ужаса, предсказанного теорией игр.

В чем ошибалась теория игр? Все возвращается к проблеме обоснования уверенности, в терминах паковщика, в уверенности другого, что вы уверены... Это так сложно. Чтобы сделать это, оба игрока должны суметь оказаться рациональными — что теории называют «сверхрациональными» *(super-rational)* — сами по себе и по отношению к рациональности другого. Кажется, нет другого очевидного способа этого достигнуть, кроме обращения Ганди *(Ghandi)* к людям, что не казалось слишком надежным людям, вооружившимся ядерным оружием.

В модели картостроитель/паковщик, вещи гораздо проще. Если мы оба паковщики, мы оба получим большие сроки. Если ты паковщик, я должен сознаться, поскольку ты сознаешься. Но если мы знаем друг друга, то нам легко осознать способность друг друга конструировать мысленные карты и достигать прямого и полного их понимания. Если я знаю, что ты картостроитель, я знаю, что ты будешь в состоянии обнаружить хитрость Дилеммы, поскольку это не очень большая карта. Ты знаешь, что я картостроитель, поэтому ты также будешь в состоянии предсказать мое полное понимание этой простой хитрости. Поэтому нас выпустят. Другими словами, мы победим, поскольку разница между идиотами и здравомыслящими людьми вещь не сильно заметная, но видна только здравомыслящим людям. Для паковщиков это града-

ция от душевнобольных (картостроители, которые не могут думать правильно и поэтому... ош... выход *(escape)*), через людей, которые могут запомнить лишь несколько пакетов знаний и поэтому глупы, до Ответственных Персон, которые применяют пакеты знаний с точностью роботов. Знаменательно, что хотя мы будем убивать миллионы и терять огромные богатства, играя в спасающие лицо игры, держа носы по ветру, как род человеческий мы удержались от уничтожения планеты.

Озабоченнось паковщика уверенностью, без видения, как ее достигнуть, связаная с игрой с нулевым результатом в условиях материальной экономики и дефицита, ведет к очень ограниченному набору возможных транзакций. Чтобы заниматься программной инженерией, мы должны быть картостроителями, и Дилемма Заключенных показывает нам, что мы имеем возможности увидеть эффективные стратегии, которые недоступны паковщикам. Создание программного обеспечения — это не игра с нулевым результатом — если я копирую вашу программу, то она есть у обоих. И у нас нет дефицита, если только программирование хорошо оплачивается. Поэтому нам доступно гораздо больше видов взаимодействий (транзакций), чем у любой другой группы в истории. Мы уже начали находить примеры в совместной разработке стандартов, в коммерческих организациях, помещающих ключевые исходные коды в открытый доступ, и в росте рынка свободного программного обеспечения. Люди, которые совершают эти вещи, не беднеют — преимущества лидерства часто перевешивают стоимость даваемого на сторону, которое вы все равно получите любым способом. Решение проблем музыкального бизнеса состоит в правильной оценке этой новой среды бизнеса, и пока мы делаем это, мы будем страдать от установленных по сговору цен, которые конкурирующие организации устанавливают сами для себя.

Рынок программного обеспечения должен оставаться интересным достаточно долго!

❒ Предопределенность *(Predeterminism)*

В работе «Структура научных революций» *(The Structure of Scientific Revolutions)* Томас Кун *(Thomas Kuhn)* ввел концепцию **парадигмы** — лежащей в основе теории мира, которую даже не считают теорией, а вместо этого называют «реальностью». Целые общества совместно используют парадигмы, и они могут оказать необычайное влияние на поведение членов общества. Одно время была такая философская парадигма, названная «предопределенность» *(predeterminism)*. Она говорит, что Бог распланировал жизнь каждого при рождении, и испытаний, которые выпадают на долю кого-либо, невозможно избежать, поскольку они были волей Бога и поэтому должны приниматься с благодарностью. Затем были религиозные споры, закончившиеся тем, что эта противоречивая свобода воли победила, а предопределенность была повержена. Это были хорошие новости, поскольку предопределенность мало устраивает людей. Если на все воля Бога, то наши слабенькие усилия мало чего стоят.

Отбросив предопределенность мы стали свободны верить, что мы можем немного управлять своей судьбой, что мы и делаем.

Даже после этого, мы все ждали, когда же упадет другой ботинок. Хотя мы верим, что результаты возможны, и поэтому мы прикладываем усилия, чтобы улучшить нашу жизнь, большинство из нас по прежнему не верит, что возможно понимание, поэтому мы не делаем усилий, чтобы понять. Теперь, когда автоматизация сделала понимание необходимым и доказала его возможность, мы имеем шанс войти в новую эру человеческого знания — настоящую Эру Информации.

Приложение А

Дополнительные материалы

❒ Автоформализация знания

Здесь краткое резюме моего опыта в замечательном мире «Автоформализации знания». К сожалению, нет ни книги, ни статей, опубликованных на английском.

Алексей[1]

Краткое резюме книги

1.1. Проблемная область

- Действительно трудная сторона разработки программного обеспечения — формализовать проблему, которую собираешься решить.
- На языке progstone картостроители — те, кто выполняет формализацию.
- Существуют виды человеческой деятельности, которые чрезвычайно сложно формализовать. Любимый пример - автопилот для внедорожного автомобиля[2]: даже не шибко умный человек обычно в состоянии выбрать подходящий маршрут движения вне дороги, чтобы не врезаться, в то же время самые умные на Земле инженеры-программисты тратят годы на то, чтобы только приблизиться к созданию программы, делающей то же самое.
- Во многих случаях нет надежды, что инженер-программист будет когда-либо способен понять проблемы заказчика (пользователя) достаточно хорошо, чтобы выполнить формализацию самостоятельно.
- Во всех случаях заказчик (пользователь) не полностью понимает, что он хочет.
- Во всех случаях пользователь не способен адекватно выразить свое понимание в виде спецификации, достаточной для успешного проектирования.

  Другими словами, очень трудно заниматься картостроением за пределами непосредственно технической области.

Итак, проблема в том, как расколоть трудное, не поддающееся построению карты приложение?

1.2. Процесс автоформализации знания

Общая идея в том, что единственный способ решить ее - создать среду, где продвинутый пользователь может выразить себя через программирование. Черт сидит в деталях *(There devil is in the details)*.

[1] Наш соотечественник, но, к сожалению, ничего, кроме имени — С.К.

[2] Или танка? — С.К.

1.2.1. Пользователь-пилот (Pilot User)

Обычно можно найти «пользователя-пилота»: пользователя с продвинутым знанием в предметной области, который также имеет некоторый опыт программирования (например, в пределах институтского курса) или способный усвоить азы программирования.

1.2.2. Инженер поддержки (Support Engineer)

Инженер поддержки создает и поддерживает среду для пользователя-пилота. Очевидно, что мы не можем ожидать многого от пользователя-пилота, кроме способности организовывать готовые вызовы функций в программу. Ключевой элемент в том, что он не обязан делать ничего за пределами этой очень ограниченной области. Всю трудную работу делает инженер поддержки.

1.2.3. Процесс

Инженер поддержки оценивает начальные потребности пользователя-пилота (например, интерфейс к аппаратуре, с которой придется начинать, какие функции ему придется вызывать и т. д.).

Например, в моем собственном опыте, я начал с простой программы с меню, позволяющей пользователю-пилоту выполнять элементарные функции управления оборудованием посредством набора меню, а пользователь-пилот модифицировал ее шаг за шагом до полнофункционального прототипа.

(*)

Пользователь-пилот начал играть с этими вещами. Каждый раз, когда он сталкивался с программной/аппаратной проблемой, то решать ее было ответственностью инженера поддержки: добавлять новые компоненты, улучшать производительность, обсуждать результаты и т. д. После этого шел следующий цикл работы пользователя-пилота и т. д.

Другими словами, пользователь-пилот разбивает проблему на части. Некоторые из них он решает самостоятельно, некоторые передаются инженеру поддержки. Ключевой момент в том, что инженер поддержки не обязан понимать всю проблему, а пользователь-пилот не обязан хорошо программировать.

В конце получится (грубый и медленный, но работающий) прототип разрабатываемой вещи, который можно использовать как основу для формальной спецификации и затем команда паковщиков с ней разберется.

Личный опыт использования этой вещи

По моему личному опыту результаты были на удивление хороши. У меня был очень красноречивый эксперимент в этой области. Были ребята, которые разрабатывали новый измеритель вязкости стекла (вискозиметр), и им нужно было управлять им посредством компьютера (это было ооооооочень давно, во времена, когда некоторые химики были не в ладах с компьютерами, не то что сейчас). Проблема была не такой уж и тривиальной, поскольку стекло — очень странная жидкость с сильной зависимостью от температуры и большими разбросами от образца к образцу (кроме того, все процессы мучительно медленные, поэтому требуется полчаса, чтобы температура равномерно распределилась по образцу, чтобы перемещение стало линейным во времени и т. п.).

Я нашел пользователя-пилота, который много программировал на FORTRANe, делая научные расчеты, однако он никогда не занимался управлением с помощью компьютеров. Как я говорил (*), я собрал и откалибровал все элементы аппаратуры и написал подпрограммы, выполняющие операции уровня пользователя: изменение мощности нагревателя, измерение температуры, загрузка (образцов ?), измерение серии перемещений. И я поместил их в простую управляемую через меню программу. Меня несколько раз вызывали помочь ему улучшить ту или другую часть контрольно-измерительной системы и все - он делал остальное сам, работая

несколько часов в день месяц. Он получил прибор, который работал при любом человеческом вмешательстве в геометрию образца и был достаточно приспособлен, чтобы справляться с широким диапазоном температур/вязкостей.

В то же самое время, я не знал и до сих пор не знаю ничего о вязкости стекла, кроме самого элементарного понимания. Он не знал и до сих пор не знает ничего об управлении с помощью компьютера. А мы вместе были способны делать вещи с действительно минимальными усилиями с обеих сторон.

Здесь идет интересная часть. Через некоторое время после того, как я начал работать над проектом, они в течение нескольких месяцев разрабатывали спецификации для других парней. После завершения нашего проекта я попросил его сравнить его с последней спецификацией, которую он написал имея реальную программу, которую он разработал. Он нашел 13 отличий:

- Он решил, что может прожить без некоторых вещей, которые не дают большого эффекта. Это засчиталось за 2 отличия.
- Шесть следующих были на самом деле тривиальны в решении, и он был уверен, что в нормальном процессе спецификация-тестирование-спецификация-тестирование эти моменты были бы решены легко.
- Два были менее тривиальными, однако, эти пункты еще можно было решить в процессе спецификация-тестирование-спецификация-тестирование-спецификация-тестирование.
- Два были нетривиальными, его ощущение было, что он не смог бы получить их достаточно быстро не играя с вещами сам.
- Этот проект был маленьким, однако, с моей точки зрения это красноречивый пример.

Другие примеры

Unix — это система, созданная разработчиками программного обеспечения для разработчиков программного обеспечения, и поэтому она так удобна как среда разработки.

❒ Экстремальное программирование

From: Colston Sanger <colston@shotters.dircon.co.uk>
Этим утром на Amazon появилась новая книга Кента Бека «Экстремальное программирование: смена содержания»[3].
Из предисловия:

> Для некоторых людей XP представляется просто хорошим здравым смыслом. Поэтому почему в названии «экстремальное»? XP поднимает принципы и практику здравого смысла на экстремальный уровень.
>
> - Если обзоры кода хороши, мы будем все время делать обзоры кода (парное программирование).
> - Если тестирование хорошо, каждый все время будет тестировать (тестирование компонентов), и даже пользователи (функциональное тестирование).
> - Если проектирование хорошо, мы сделаем его частью каждодневной деятельности каждого *(умножение усилий — refactoring)*.

[3]Kent Beck's *Extreme Programming explained: embrace change*, Addison-Wesley

- Если простота хороша, мы всегда будем оставлять систему с простейшим дизайном, который обеспечивает нужную функциональность (простейшая вещь, которая может работать).
- Если важна архитектура, каждый все время будет работать над определением и уточнением архитектуры *(метафора — metaphor)*.
- Если тестирование интеграции важно, то мы будем интегрировать и тестировать несколько раз на дню *(непрерывная интеграция — continuous integration)*.
- Если хороши короткие итерации, мы будем делать очень, очень короткие итерации — секунды и минуты и часы, не недели, месяцы и годы *(Игра планирования — the Planning Game)*.

У меня было много контактов с людьми XP за последние несколько месяцев и идеи произвели на меня хорошее впечатление. Они продвинулись с того места, где мы оставили Programmers' Stone в конце 1997 г.

❒ Неожиданная рекомендация

From: "Philip W. Darnowsky" <pdarnows@qis.net>

On Tue, 2 Nov 1999, Alan Carter wrote:

> Слоган: The Programmers' Stone способствует вашей половой жизни. Это не абстрактный этический аргумент, я знаю, но это реально.

Я буду свидетельствовать этому. Две недели назад я еще работал в среде с высокой концентрацией паковщиков. Я занимался разработкой программного обеспечения, но это делалось для крупного государственного заказчика, поэтому паковщики суетились. Когда я ушел и нашел нынешнюю работу, где много картостроителей, я начал испытывать чувство, которого у меня не было давно. Страсть. Мой сексуальный настрой (драйв) восстановился до естественного уровня.

Приложение Б

Ссылки

❒ Указатели на ресурсы

1. **Mining Usefulness** (`http://www.geeknews.org/features/articles/usefulness.html`)

   Как противоположность согласию. Например.

2. **The Jargon File** (`http://www.tuxedo.org/~esr/jargon`)

   Классический праздник хакерской культуры, поддерживаемый Эриком Рэймондом (Eric S. Raymond).

3. **Design Patterns in MFC** (`http://www.cs.hut.fi/~kny/patterns`)

   Интересное исследование паттернов дизайна, которые можно увидеть в MFC и других графических библиотеках (инструментах).

❒ Библиография

1. Adams, Scott. **The Dilbert Future** («Дильбертово будущее»), Boxtree, ISBN 0-7522-1118-8.

   Очень забавно и проницательно. Об Адамсе говорят много чепухи. Но он просто очень смешной карикатурист. Говорят, что он ужасен и циничен. Это потому, что он с поразительной точностью документировал происходящую на рабочих местах глупость. Всю помпезность, недобросовестность, тупость и ритуализм. При просмотре завершающего книгу раздела ваши волосы встанут дыбом.

2. Brookes, Frederick P. **The Mythical Man-Month** («Мифический человеко-месяц»), Addison Wesley, ISBN 0-201-00650-2.

   Обычно считается самым разумным руководством по управлению практичными эффективными программными проектами, каждое высказывание Брукса, кажется, взято из практики блюдущих ритуал ISO 9001 зомби. Именно поэтому производство коммерческого программного обеспечения в таком застое.

3. DeMarco, Tom & Lister, Timothy. **Peopleware: Productive Projects and Teams** («Peopleware: продуктивные проекты и команды»), Dorset House, ISBN 0-932633-05-6.

   Наблюдения с точки зрения здравого смысла, касающиеся того, как делать эффективные программные проекты. Лучшие места посвящены нападкам на офисы с открытой планировкой. В Reciprocality (Взаимность) показано, что открытую планировку можно считать желательной, поскольку потребители ритуала любят наблюдать ритуальные движения друг друга весь день, а бесконечно звонящие телефоны не вызывают проблемы, поскольку все равно никто не думает. Посмотрите также комментарии об «однородных

командах» («jelled teams») и «профессионализме», который предлагается в качестве синонима для самодовольной помпезности.

4. Degrace, Peter & Stahl, Leslie Hulet. **The Olduvai Imperative**, Prentice Hall, ISBN 0-13-220104-6.

   Авторы настроились написать книгу об инструментах CASE, и обнаружили множество ждущих своего исследования белых пятен, когда мы спрашиваем, что же мы в действительности делаем, когда разрабатываем программы. Я не думаю, что предлагаемое ими разделение «греки и римляне» («Greeks vs. Romans») слишком хорошо работает, но они действительно открыли, что есть два отличающихся подхода.

5. Feynman, Richard P. **Feynman Lectures on Computation** («Фейнмановские лекции по исчислению»), Addison Wesley, ISBN 0-20148991-0.

   Все отлично, но особенно разделы, посвященные Чарлзу Беннетту (Charles Bennett) и энергетическому эквиваленту информации. Эта книга была предметом бесконечных споров в течение 10 лет, но сейчас мы можем положиться на слова Фейнмана по поводу этого замечательного результата, так важного для Reciprocality.

6. Gamma, Erich et. al. **Design Patterns: Elements of reusable Object-Oriented Software** («Приемы объектно-ориентированного проектирования. Паттерны проектирования»), Addison Wesley, ISBN 0-201-63361-2.

   Лучшая книга по паттернам дизайна (design patterns). Делает акцент на аспектах композиции программного дизайна – том, что не могут делать носители M0. Очень полезна в местах, где упрощенная M0 неправильная интерпретация ISO 9001 полностью вышла из-под контроля. Вы просто ссылаетесь на шаблон (по имени) в документе «Проект Архитектуры» и говорите о деталях в документе «Детальный Проект». Это дает полезный документ, который *не мешает* хорошей композиции, требуя, чтобы проект укладывался в дебильную обязательную структуру документа, созданного человеком, который не может понять, что такое композиция, но должен его написать!

7. Goldratt, Eliyahu M & Cox, Jeff. **The Goal** («Цель»), Gow, ISBN 0-566-07418-4.

   Сказочки о том, как наши герои умеют думать с M0 и решать проблемы, вместо того, чтобы бежать с работы со всеми вещами, что скорее всего и произошло бы на самом деле.

8. Goldratt, Eliyahu M. **It's Not Luck** («Нет счастья»), Gower, ISBN 0-566-07637-3.

   Еще несколько сказочек.

9. Hohmann, Luke. **Journey of the Software Professional** («Путь профессионального программиста»), Prentice Hall, ISBN 0-13-236613-4.

   Очень близко к Programmers' Stone, насколько это возможно, если придерживаться парадигмы и языка M0. Самая близкая к Programmers' Stone из напечатанных книг. «Путь» (Journey) в названии, конечно же, в алхимическом смысле.

10. Levy, Steven. **Hackers** («Хакеры»), Penguin, ISBN 0-14-023269-9.

    Как «очевидно очень глупые» люди изменяют мир. В главной роли — маленький Билл Гейтс (Bill Gates), играющий молодого Дарта Вадера (Darth Vader). (Факт: В 1978 я купил продукт Microsoft, называвшийся «EDAS for TRS-80 Model I». Он оказался настолько плохим, что я написал с его помощью замену, а потом просто выбросил. Лента в магнитофонной кассете, на которой он поставлялся, была настолько короткой, что не могла хранить что-нибудь полезное).

11. Naur, Peter. **Computing: A Human Activity** («Вычисления: человеческая деятельность»), ACM Press, ISBN 0-201-58069-1.

    Мудрые слова из давних времен. Как стало возможно, что программирование смогло стать чем-то *другим*, чем человеческой деятельностью, но люди об этом забыли.

12. Schwartz, Howard S. **Narcissistic Process and Corporate Decay** («Процесс нарциссизма и распад корпорации»), New York University Press, ISBN 0-8147-7938-7.

    Описывает M0 в коммерческих обстоятельствах с позиций модели Фрейда. Модель, конечно, во многом справедлива — M0, а не детские умы — место, откуда приходит мотивационная и иллюзорная структура.

13. Senge, Peter M. **The Fifth Discipline** («Пятая дисциплина»), Random House, ISBN 0-7126-5687-1.

    Свободное от M0 мышление в бизнесе. Вводятся «Сенгианские Структуры» (Sengian Patterns), которые, я уверен, пораженные M0 не способны заметить в ситуациях реального мира.

14. Spencer-Brown, George. **Laws of Form** («Законы формы»), E. P Dutton, ISBN 0-525-47544-3.

    Культовая классика среди хакеров около 30 лет назад, на которую также ссылается в книге «Вселенная за следующей дверью» Роберт Уилсон (Robert Anton Wilson's *Universe Next Door*).

15. Weinberg, Gerald M. **The Psychology of Computer Programming** («Психология программирования компьютеров»), Van Nostrand Reinhold, ISBN 0-442-20764-6.

    Этот старинный текст никому не удалось превзойти. Никто почему-то не отваживается.

16. White, Michael. **Isaac Newton — The Last Sorcerer** («Исаак Ньютон — последний алхимик»), Fourth Estate, ISBN 1-85702-416-8.

    Уайт, как кажется, не понимал, что алхимия — это преобразование оператора — картостроение, но его изложение великолепно, поэтому вы можете сделать собственные выводы на основе его данных.

17. Yourdon, Edward. **Decline and Fall of the American Programmer** («Закат и падение американского программирования»), Prentice Hall, ISBN 0-13203670-3.

    Я еще не видел второго издания. Проблема оффшора не наступила, поскольку программирование не тот свободный от контекста процедурализм, которым, как думают, можно с успехом заниматься в офисах с открытой планировкой. Показывает тоскливую предопределенность глупость ритуалов управления по стандарту в фирмах M0.