КОМПЬЮТЕРНАЯ ФИЛОСОФИЯ

http://unism.pjwb.org
http://unism.pjwb.net
http://unism.narod.ru

Информатика молода, она стремительно развивается. Любой текст на компьютерные темы рискует безнадежно устареть еще до того, как кто-либо удосужится его прочесть. И однако есть общие темы, так или иначе воспроизводимые на каждом витке спирали; в частности, бросается в глаза почти полное отсутствие осмысленной критики на фоне переизбытка талантливых творцов. Возможно, эти заметки что-то добавят к росту самосознания в рядах компьютерщиков — в дополнение к собственно технологическому развитию, которое широко представлено другими в других местах. Тексты написаны в разные годы по разным поводам и расположены почти случайным образом, без учета хронологии.

Искусство, наука, философия

За сотню лет компьютерной эры информатика выросла в солидное научное предприятие с полномасштабной организационной структурой: от грандиозных теорий до узкоприкладных разработок. На этом пути сделано немало открытий, породивших новые направления в других науках, и прежде всего в математике и физике. В свою очередь, абстрактная математика постепенно обретает плоть в продвинутых компьютерных технологиях, а за ней и другие науки потихоньку подталкивают дело каждый в своем направлении. Термин «компьютерная наука» укоренился в английском языке, в других языках та же идея присутствует в каких-то иных выражениях — суть от этого не меняется. Разумеется, речь идет не только и не столько о применениях вычислительных машин и автоматизации интеллектуальных операций — информатика универсальна, это особый взгляд на любую вообще деятельность, в дополнение к представлениям о материальном производстве или психических процессах.

Но, как всегда, глядеть можно по-разному. Большинство компьютерщиков не заморачивается постижением теоретических основ — они просто используют имеющиеся инструменты, совершенствуя их в сугубо практических нуждах сугубо практическими методами. Опыт и интуиция — все, что нужно им на этом пути. В своих наиболее совершенных формах компьютерное ремесло становится искусством. Искусство программирования, искусство администрирования, искусство системной архитектуры или разработки программного обеспечения… Как и в других отраслях, слишком много искусства — вредно для индустрии, но в небольших дозах эстетика хорошо продается, и почему бы на этом не поиграть?

Время от времени кто-то слишком образованный пытается создавать новые инструменты с учетом теоретических представлений. Например, хочется в языке программирования явно прописать расхожую математическую конструкцию. Почему бы и нет? В каких-то ситуациях это дает относительную свободу и становится конкурентным преимуществом. Тогда нововведение обрастает легионом адептов и зоосадом всевозможных адаптаций — и становится еще одним долгоживущим монстром, частью компьютерной науки… Практическое применение этого инструмента становится высшим принципом, лишь отдаленно связанным с его былой полезностью и удобством.

Здесь уже начинается философия. Философия языка или стиля программирования, философия сетевого протокола или платформы разработки, философия формата данных или языка разметки, философия интерфейса, философия веб-дизайна, философия станков с программным управлением… Кто чем занимается, тот о том и философствует. Чем обширнее область применения, тем больше становится философии. У людей без философии не обходится

ни в одном деле, и сфера информатики не стала исключением.

Поначалу все это остается по большей части кухонной самодеятельностью, досужим развлечением — в противоположность серьезной работе. Но потом появляются книги, в которых кто-то пытается явно высказать то, что другие только подразумевают — посмотреть на себя и ремесло со стороны. Опять же, стихийно, исходя из профессиональной интуиции и личного опыта, под влиянием расхожих философских идей. Такой синкретический уровень безусловно необходим и неизбежен. Со временем все это перерастет в более серьезные философские искания, в поиск универсалий, лежащих в основании научных изысканий и технологических изысков.

Вот тут у публики возникает вопрос: оно нам надо? Каждый компьютерщик со временем свой подход к работе, какие-то общие представления и принципы. Пропорционально общей начитанности и обратно пропорционально текущей загруженности. Это его философия — и ему этого вполне хватает. Нечего тут делать профессиональному философу.

Примерно так же рассуждали когда-то философствующие физики, ухватившись за модное словечко «позитивизм». Дескать, нет никаких всеобщих оснований, а есть практическая работы, систематизация опыта… Потом, правда, выяснилось, что не все так просто, — и разочарование привело к другой крайности, к отказу от попыток вообще что-либо что-то понимать. Науку объявили абстрактным конструктом, чистой игрой ума, а такой вещи как научный факт в принципе не существует — это чистая условность, произвольное соглашение для удобства дальнейшего теоретизирования. Впрочем, тут тема для особого обсуждения. Нынче все равно нет еще ничего в компьютерной философии кроме первобытного мудрствования — смесь ходячих предрассудков и пропагандистских клише.

По счастью, философствующий компьютерщик — это пока не так опасно, как философствующий физик. Компьютерщик не пытается объявить свои измышления единственной причиной мира и не строит модели Вселенной целиком. Ремесло слишком практическое, оно не дает зарваться. Вот когда появятся первые образчики компьютерной космогонии, и разработчики ролевых игр станут принимать их всерьез, — пора кричать «караул!». И начнутся сугубо философские баталии, из которых ни один нормальный программист не уяснит ровным счетом ничего. И опять будет спрашивать: зачем? Есть компьютерная наука, есть искусство — а философия, как водится, не идет ни к тому, ни к другому, бессмысленное суесловие…

Однако подобные сомнения — маска вполне определенной философии, скрывающейся за вроде бы наивной рассудочностью. Вопрос поставлен некорректно. Философия просто есть — и потому надо не задаваться вопросами «зачем?», а принять это как данное и сделать из этого какие-то выводы — каждый для себя.

Тем более, что польза от философии все же имеется, и не надо быть семи пядей во лбу, чтобы ее усмотреть.

Элементарное наблюдение из преподавательской практики: осмысленная информация усваивается гораздо эффективнее. Если в горах эмпирии есть хоть какое-то подобие системы — лучше подсунуть студенту одну общую мысль, чем хаос несистематизированных сведений. В каждой области бездна премудрости, технологии бесконечно сложны… Не мудрено и растеряться перед эдакой махиной. Философия организует деятельность, делает ее приемлемой для живого человека, соизмеримой с ним. Это, возможно, окажется полезным и для будущих компьютеров, способных общаться с людьми по-человечески.

С другой стороны, в информатике до сих пор существует своего рода кастовая система, противопоставляющая чайников профессионалам, а среди профессионалов — тоже иерархия ритуально замкнутых групп. Компьютерщиками свойственно (или выгодно) поддерживать в пользователях благоговейный ужас, подчеркивать сверхъестественную сложность информационных систем — и свою незаменимость в качестве жрецов-заклинателей. Придавая делу мистическую окраску, они постепенно сами проникаются мистическим отношением к собственному ремеслу, начинают верить в абсолютную правильность преходящих форм. Понятно, что развиваться с выключенными мозгами не всегда удобно, — и тут на помощь приходит критическое мышление, философия.

Разумеется, есть кое-что и помимо простого наследования знаний или принятия технологических решений. Само соединение слов «компьютер» и «философия» порождает множество коннотаций. Компьютерная философия — это и философствование по поводу компьютеров, и философствование при помощи компьютера, — и философия самих компьютеров, когда те до этого дорастут. А то, что это неизбежно произойдет, — не вызывает ни малейших сомнений. На то мы и люди, чтобы им в этом помочь. И здесь важно не только знание, но и понимание — не только помощь, но и сочувствие. Даже если это перспектива далекого будущего, начинать приходится сейчас, с тем, что есть на данный момент. Следовательно, начинать с себя. И тут важно не переусердствовать, не переборщить.

Как говаривал Михайло Ломоносов, математику уже затем учить надо, что она ум в порядок приводит. И действительно, приводит. Иногда чересчур самозабвенно и деспотически. Философия же нужна затем, чтобы приводить в порядок сам порядок — не давать ему перерасти из опоры в оковы. Философия напоминает о невечности и относительности любого порядка, заставляет разумно относиться к собственной разумности, не пытаться сделать из нее погоняло для Вселенной. Мир невероятнее любых наших фантазий — он большой, нем много всего, и он всегда норовит подсунуть нам очередную несуразность вместо следования стандартам и предписаниям. Да, жить в таком мире сложнее, чем в уютненьком мирке формальной логики. Но куда мы денемся? — других миров нет.

В плену у башни

Количество компьютерных языков давно уже перешло всякие разумные границы — счет идет на тысячи, и новые конструкции появляются чуть ли не каждый день. Конечно, большинство из них — лишь диалекты чего-то ранее известного, либо специализированные протоколы для узкого круга технических задач. Но нередки и претензии на универсальность, подкрепленные ссылками на самую высоколобую математику. В конце концов, нет такого диалекта, которые не мечтал бы стать языком нации.

Никто не в силах не то что освоить — но хотя бы взглядом охватить все это многообразие. На всех фронтах кипучая деятельность, полное впечатление целеустремленного и плодотворного труда. Разработчики полны энтузиазма, программисты имеют возможность блеснуть эрудицией, а работодатели — тешить себя иллюзиями о суперсовременных технологиях и невероятной эффективности. В едином порыве созидания мы строим — что? Очередную вавилонскую башню — или, быть может, навозную кучу до небес?

Книги по программированию плодятся с не менее устрашающей стремительностью. Их авторы традиционно пытаются убедить себя и читателя в безусловном превосходстве того, к чему они привыкли, над тем, к чему они не горят желанием привыкать. В каждой книге — примеры решения одних и тех же типовых задач, плюс указания на возможность включения кода, написанного на любых других языках (предполагая, что программирование на них есть пережиток первобытности или болезненное извращение). Различия — в стиле рекламы. Одни демонстрируют разительную гибкость языка, делают упор на компактность программ; другие выкатывают на публику богатство настроек и возможности адаптации; кто-то подчеркивает минимальность выразительных средств, языковую однородность и единообразие; где-то полезен обыкновенный эпатаж, банальное стремление выделиться. При этом, разумеется, каждый клянется в приверженности стандартам и божится, что ему проще решить любую задачу на своем языке, нежели приспосабливаться к другим.

Миллионы компьютерщиков по всему миру заняты чем-то своим, все вместе участвуя в грандиозном столпотворении. Навозная куча дает кров и стол малым мира сего, и можно жить в киселе и питаться киселем. Время от времени случаются трагедии, когда что-то выходит из моды, и водопой пересыхает, а у другой кормушки уже слишком тесно, и надо вымирать. Но кому до этого дело? — атмосфера всеобщей суеты прекрасна и удивительна.

Робкий вопрос: а нельзя ли все-таки обойтись одним универсальным языком, хотя бы в качестве языка междупрограммного общения? Ответ: нельзя. По тем же причинам, по которым

невозможно утверждение единого международного языка. Есть группы людей с весьма различными интересами, и многообразие языков — одно из орудий конкурентной борьбы. Пока люди разделены на нации и классы, пока они не доросли до разумности и живут по животным законам, — они будут говорить на разных языках. В частности, профессиональный жаргон охраняет привилегии и выращивает снобизм некоторого сообщества — и он же может погубить это сообщество в изменившейся социальной среде. Дайте программистам возможность вырваться из вавилонского плена — большинство останется сидеть в своей норе, и это их единственная универсалия.

Каждый язык для чего-то удобен, для чего-то не очень. Универсальный язык не удобен ни для чего — но зато он может выразить гораздо больше, чем любой специализированный язык, и даже то, для чего пока не придумали особого жаргона. Насколько выигрыш в эффективности оправдывает языковой хаос — еще вопрос. Смотря что считать эффективностью. Человеческая деятельность меняется — технологии устаревают. Универсальный язык обеспечил бы плавный переход от одних концепций программирования к другим, разумное сочетание разных возможностей в рамках единой среды.

В качестве иллюстрации можно уподобить эволюцию компьютерных языков животной борьбе за существование. Мутации могут дать адаптивные преимущества одному из сожительствующих видов — и тогда этот вид быстро захватывает все доступные экологические ниши, вытесняя другие виды и ставя их зачастую на грань вымирания. Единственное ограничение тут — существование естественных врагов, пресекающих слишком активное размножение. Появление человека как универсально приспособленного вида привело к гибели многих биологических сообществ, и постепенно разрушает оставшиеся. Дикий человек изменяет свою среду стихийно, ради сиюминутной потребности, — и придумывает невероятно действенны приспособления для того, чтобы подогнать мир под себя. Когда обнаруживается вредный побочный эффект — изобретаются приспособления для его нейтрализации, опять же без оглядки на возможные последствия. Последнее время сохранение биологического разнообразия стало модой — и, скажем, болото в черте города — это уже не проклятье, а достижение цивилизации. Наряду с экономической конкуренцией, это еще одно проявление нашей дикости: не умеет пока человек создавать полноценные экосистемы — и потому трясется над существующими.

Точно так же, в информатике создают по поводу и без повода специализированные языки, а не сбалансированные среды. Отсюда конфликты совместимости, войны форматов, фанатизм адептов и мучения пользователей. На заре компьютерной жизни были попытки создания зачатков экосистем — например, язык C создаваться под оперативные системы семейства Unix и развивался вместе с командными оболочками; во многом это способствовало живучести идеологии Unix и ее повсеместному распространению. Последующая стихийная эволюция преуспела в нагромождении нелепостей — но до сих пор язык C служит внутренней интегративной силой, без которой семейство Unix потеряло бы свою определенность.

Что же, компьютерное многоязычие — лишь признак недоразвитости, и нет в нем никакой реальной нужды? Не совсем. Человеческие интересы тоже вполне реальны — а язык отражает строение человеческой деятельности, для этого он, собственно, и нужен. Будет мир устроен иначе — будет другой язык. Да, основное определение разума — универсальность, способность связать в деятельности любые стороны мира. Поэтому языковая универсализация по мере очеловечивания дикой человечьей природы — неизбежна. Однако проявиться единое может только в многообразных частностях, каждая из которых представляет лишь одну из его сторон. А значит, будут сосуществовать разные языки программирования, и будут рождаться новые. Другое дело, что нелишне бы ввести все это в разумные рамки, понемногу избавляться от стихийности, от животного приспособленчества и животной конкуренции. Как? Ну, хотя бы для начала задуматься над этим, а не над очередной порцией сугубо прикладных фокусов. Конечно, куда проще снискать популярность (или деньги) полезными изобретениями в сфере повседневного быта. Это бездонная нива для интеллекта. Но разум начинается как раз за границами повседневности. По крайней мере, пока.

А тем временем есть прекрасная возможность развивать новую гуманитарную науку —

компьютерное языкознание как раздел языкознания в целом, начиная с эмпирического упорядочения компьютерно-языковой истории и кончая философскими принципами языкообразования и языковой динамики. Если не поддаваться соблазну впихнуть все наблюдаемое в примитивные математические формы, можно заметить немало такого, что сделало бы понятнее загадочные страницы истории естественных языков — поскольку в сфере информационных технологий на наших глазах протекали языковые процессы сходные с развитием естественных языков в дописьменную эпоху, о котором мы пока можем только что-то предполагать. В частности, многие проблемы компаративистики предстали бы в совершенно ином свете. Возможно, и компьютерщикам от этих изысканий что-то когда-то перепадет.

Не надо иллюзий

Когда человек увлечен своим делом — это хорошо. Если при этом человек умеет еще и умные книжки писать — просто замечательно. Можно только позавидовать.

Но, как говорит французская пословица, nul miel sans fiel. Чего регулярно не хватает пишущим профессионалам — это самокритичности.

Конечно, все знать и уметь невозможно. Тогда увлекаться своим делом будет некогда. А желание писать все правильно — это называется перфекционизм, психическое отклонение. В конце концов на то и существуют бесталанные критики, чтобы общий уровень критичности восстановить. Ну что ж, будем восстанавливать…

Странные вещи иногда приходится созерцать. Например, в одной книжке по языку Erlang с удивлением читаю, что, оказывается, великое преимущество Erlang по сравнению с C — наличие простого синтаксиса для работы с битовыми полями. Но мы-то знаем, что битовые поля существовали в самых первых вариантах C — и это вполне естественно, поскольку в те далекие времена программисты были еще не избалованы немеряными объемами памяти и неимоверной полосой пропускания сетей — а потому старались упаковывать данные по максимуму. Много лет спустя стало хорошим тоном выделять 64-битовое целое для простого двоичного флага. А тогда ни у кого бы рука не поднялась. В конце концов, и регистры флагов появились в процессорах из тех же соображений — из экономии. И сетевые протоколы строили на том же принципе. В нынешних книгах про C битовые поля зачастую даже не упоминают. Но то, что Erlang содрал свой синтаксис из C, — сомнений никаких.

В другой книжке, по языку Prolog, читаю, что в C++ будто бы нет возможности определять функции так, чтобы они работали с параметрами произвольных типов. А почему нет? Задолго до появления в C++ шаблонов, которые синтаксически оформляют подобную возможность, на самом обычном C творили программы, динамически определяющие тип передаваемых данных и реагирующие соответственно. При этом ничто не мешало передавать в качестве параметров в том числе и какие угодно функции — вплоть до таких, которые вообще нигде не описаны и формируются на лету. В конце концов, наличие эффектных прибамбасов в других языках стало возможным именно благодаря такой гибкости C — на котором сейчас пишут практически все интерпретаторы и компиляторы, — по крайней мере, на первых порах. Да, конечно, многие трюки начисто игнорируют нормы структурированности и безопасности, и с точки зрения индустриального программирования выглядят ужасно, — однако в конечном счете все работает, и очень даже неплохо. А если кому-то надо соблюсти идейную чистоту — все сводится к добавлению еще одного куска кода, эту самую чистоту поддерживающего. То есть, по сути, вопрос не синтаксиса, а стиля.

Но самое смешное начинается там, где творцы (и продавцы) программных продуктов начинают хвастать их невероятной производительностью. Когда вам пытаются впарить браузер, который «работает быстро даже при медленном Интернете», — это реклама законченного идиотизма. Козе понятно, что если у вас сеть не дает закачать гигабайт данных за две секунды — это никаким боком от браузера не зависит. Быстрее, чем полоса позволит, — не проскочить. Ну невозможно смотреть фильмы HD по телетайпу! А браузер что? — он, быть может, сам по себе, безо всякой сети, быстро работает, — только пользователю от этого ни

холодно, ни жарко — ему качать надо.

То же самое, когда сравнивают разные языки программирования. В уже упомянутой книжке про Erlang есть восторженные сказки о том, как программы на Erlang работают даже быстрее, чем написанные на C. Похожие заявления по поводу других языков приходилось читать у соответствующих проповедников. Полный бред. Опять же, по простой причине, что выполняет программы на всех этих языках виртуальная машина, написанная на C, и сами агитаторы это признают. Если другая программа на C делает ту же работу медленнее — значит, ее криво написали. Претензии к программисту, а не к языку.

Пропагандисты Erlang многократно повторяют, что виртуальная машина Erlang якобы способна поддерживать одновременное выполнение миллионов процессов — программистам на C такое не по зубам. Ну, считать зубы мы не будем… А вот оценить ресурсы, необходимые для большого количества процессов, мы можем. Каждый процесс требует, как минимум, места в физической оперативной памяти. Если его данные сваливаются в виртуальную память (то есть на жесткий диск), от производительности ничего не останется. Сколько-нибудь осмысленный код вряд ли займет меньше 16 килобайт — учитывая заголовочные структуры, тело процесса, стек, статические данные и динамическую память. На миллион процессов надо 16 гигабайт оперативной памяти. То есть, далеко не всякий сервер такое потянет. А если учесть еще и необходимость работы с жестким диском, и нагрузку на сети — заявленный параллелизм выглядит глупой рекламной сказочкой.

В чем тут собака? А дело в том, что процессы в Erlang не настоящие, а «легковесные». То есть, на самом деле выполняется *только один процесс* — виртуальная машина, которая подменяет собой операционную систему, принимая на себя ряд ее функций. То, что в Erlang громко называют процессами, — всего лишь объекты определенных классов, с которыми работает управляющая программа. Понятно, что инициализация объекта требует куда меньше работы, чем запуск нового процесса, — достаточно выделить память и передать управление конструктору. Для примитивных задач типа коммутации и пересылки сообщений объем собственных данных объекта невелик, порядка сотен байт. Это и позволяет без чрезмерного напряга разместить «параллельные процессы» в оперативной памяти. При этом неявно предполагается, что «процессы» большей частью однотипны — то есть, количество классов (а следовательно, и объем кода, который надо хранить в оперативной памяти) невелико. Время от времени управляющая программа обращается к тем или иным методам класса для обработки конкретного объекта, после чего «процесс» завершается или приостанавливается. То есть, речь идет не о параллельной работе нескольких процессов, а о последовательной обработке запросов одним-единственным процессом (который, в принципе, может быть исполняться в нескольких параллельных потоках, на нескольких процессорах или ядрах). Ограничения подобной схемы очевидны. Стоит только попробовать одновременно запустить хотя бы тысячу совершенно разных программ, требующих постоянного внимания, — и ни о какой производительности уже не будет и речи; начнутся проблемы с распределением ресурсов и ожиданием в очереди. Несравненный параллелизм Erlang оказывается при таком раскладе чистой воды жульничеством.

По большому счету, совершенно безразлично, как устроен язык программирования. Производительность создаваемых программ определяется не синтаксисом языка, а реализацией конкретных компиляторов и интерпретаторов, качеством стандартных библиотек. В конечном итоге, на выходе всегда машинный код, и этот код выполняет тот же процессор с той же периферией. До высокоуровневых извращений и математических абстракций процессору дела нет. Он просто выбирает по порядку инструкции, выбирает данные, преобразует данные по инструкции, пересылает результат куда скажут, и перемещает указатель к месту выборки следующей инструкции. Как ни мудри — ничего другого нет. И пока не изменится железо — не будет.

Было время, писали на языке ассемблера. Первые языки программирования были просто сокращенной записью того же ассемблерного кода. Потом появились языки высокого уровня и прекрасные оптимизирующие компиляторы. И даже возникла мысль архитектуру компьютеров привести в соответствие с языками высокого уровня, что и было с успехом реализовано. Но

оказалось, что эта красота не выдерживает гонки за развитием индустрии. Реализация новых возможностей требовала основательной переработки компиляторов, а создание хорошего компилятора — долгая и серьезная работа. Еще меньше отличалось гибкостью производство компьютерного железа. Поэтому решили, что лучше язык высокого уровня максимально приблизить к исторически сложившейся архитектуре машин, а задачу оптимизации возложить на программиста. Так родился великий и могучий язык С.

Поначалу, впрочем, даже на С писали со вставками в машинном коде, и даже появились синтаксические конструкции, поддерживающие обращение к процессору напрямую. Потом, по мере увеличения железной производительности, необходимость в столь суровой оптимизации отпала, и код стал полностью высокоуровневым. Непосредственно в кодах теперь не работают даже производители разного рода электронных устройств — у всех есть специализированные языки микропрограммирования. Как всегда, тон задает коммерция. Чем ниже уровень программирования — тем больше оно требует искусства. А с бизнесом искусство всегда было в напряженных отношениях. Бизнесу надо, чтобы не обязательно красиво — зато быстро, без лишних затрат, и ближе к продажам. Вот тут и расцвели всяческие надстроечные языки. Которые, в принципе, все равно, чем транслировать. Особая роль С (давно переросшего себя и превратившегося в C++/C# с многочисленными функциональными расширениями) сегодня не более чем традиция. С тем же успехом можно было бы писать компиляторы на фортране, а нынешние высокоуровневые хвалятся тем, что сами себя компилируют, — еще одна иллюзия. Есть процессор, который понимает только встроенный в него производителем набор команд. Архитектура всех существующих процессоров (кроме, быть может, человеческого мозга) принципиально одинакова, различие только в деталях. Из этого все следует. В частности, можно заранее предсказать, что никакой декларативный язык не может достичь эффективности языков императивных. Потому, что декларативный код — это всего лишь данные, и превратить их в нечто исполняемое может только другая программа — виртуальная машина, интерпретатор, компилятор… Но появление каких-то подготовительных стадий перед этапом исполнения — это дополнительное время и место. Никакая оптимизация тут не спасет. Вот когда научатся данные сами себя преобразовывать и пересылать — тогда и посмотрим, как оно сложится. А пока — не надо иллюзий.

Программы и агенты

Приготовьтесь. Я сейчас открою вам страшную тайну.

Дело в том, что ни одна программа ничего не делает и не вычисляет сама по себе — для этого нужен некто, способный прочесть код, интерпретировать его как инструкцию к исполнению, и выполнить в соответствии со своим пониманием и своими возможностями. Точно так же, как учебник анализа остается мертвым текстом, пока кому-то не приспичит приложить производные да интегралы к практическим задачам.

Казалось бы, ничего особенного, и должно подразумеваться само собой, — но почему-то многие теоретики компьютерной науки и компьютерные философы об этом забывают.

Например, в книгах по декларативным языкам программирования можно встретить высказывания о каких-то запрограммированных в декларативном стиле вычислениях, да еще с глубокомысленными замечаниями и рекомендациями по оптимизации кода, — хотя на самом деле никакое утверждение ни на каком языке не предполагает необходимости что-то в нем сообщаемое выполнять, поскольку выполнять, в общем-то, нечего. Декларативный язык потому и называется декларативным, что он описывает, констатирует — а не предлагает. Если есть, допустим, утверждение «Я люблю манную кашу» — наше дело принять его к сведению, а любая активность по поводу — это уже наша личная инициатива.

Очевидно, поборники декларативности просто путают разные вещи — и немного жульничают, в рекламных целях. Они скромно умалчивают, что любая обработка кода начинается в ответ на совершенно императивную команду; введена эта команда оператором в интерактивной оболочке или считана неким агентом из какого-то хранилища в процессе

выполнения другой программы — совершенно неважно. То, что в некоторых оболочках команды замаскированы под предложения декларативного языка, дела не меняет. Смысл написанного определяется контекстом — то есть, по сути дела, некоторым агентом, в этом контексте работающим.

Так что же такое — этот загадочный агент?

В элементарном курсе информатики нас учат, что программы исполняются процессором, который выбирает из оперативной памяти инструкцию за инструкцией, загружает в регистры необходимые данные, как-то эти данные преобразует, куда-то при необходимости сохраняет, — и все начинается сначала. В принципе, процессоров может быть много — и они как-то делят работу между собой. Кто-то специализируется на управлении устройствами, кто-то наделен набором универсальных команд. Суть от этого не меняется. В любом случае исполнителем будет некое устройство, способное преобразовывать данные — а программа представляет собой такие же данные, как и все остальное; по джентльменскому соглашению, мы принимаем, что какие-то записи в памяти представляют исполняемый код, а другие — только подвергаются обработке.

Картина, конечно же, упрощенная. Например, программа может на лету компилировать куски динамически формируемого текста и объявлять результат исполняемым кодом — процессору это все равно. Кое-кто назовет подобные трюки дурным стилем программирования. Но если присмотреться, живая природа в большинстве случаев предпочитает действовать именно так, совмещая разные функции в одном органе, — неграмотная, что с нее возьмешь?

Не существует предпочтительного набора базовых команд. Сколь угодно сложные куски кода часто оформляются как элементарная команда — иногда они внедряются в архитектуру компьютеров, становятся микропрограммой; микропрограммы могут быть реализованы аппаратно, как команда процессора, — и наоборот, какие-то ранее элементарные команды становятся микропрограммами, а то и внешним кодом. Все зависит от текущей моды, от преимущественной ориентации на определенный класс задач.

В этом плане работа командного интерпретатора по сути ничем не отличается от исполнения откомпилированного кода — есть иерархия операций, и вместо низкоуровневых деталей можно оперировать крупными блоками. Соответственно, компилируемые языки ничем принципиально не отличаются от интерпретируемых — и в том, и в другом случае требуется некоторая среда, организующая переход от высших уровней к низшим. Оболочка операционной системы, интерпретатор, виртуальная машина, IDE… Все это эквивалентные способы иерархической организации и визуализации работы процессора. Пошаговое выполнение программы в отладчике какого-либо компилируемого языка выглядит точно так же, как исполнение кода в диалоговых оболочках Ruby или Prolog. А в конечном счете работает все та же железка, со своим набором встроенных команд.

Но тут есть одна философская тонкость.

Сегодня мы при слове компьютер сразу представляем себе цифровое устройство — а ведь компьютеры бывают еще и аналоговые. И начиналась-то информатика именно с них. Программы не вводили с какого-то внешнего носителя, и не набивали с терминала, — а набирали вручную путем коммутации аналоговых цепей. Первые цифровые компьютеры поначалу тоже программировали путем ручной установки значений регистров, обыкновенными кнопками и тумблерами. Это потом стало проще, когда появились стандарты архитектуры и стандартные форматы данных, и программы стали всего лишь разновидностью данных, а не положением переключателей на пульте управления.

Но все равно, современный процессор — это в конечном счете некое аналоговое устройство, скоммутированное таким образом, чтобы обеспечить выполнение стандартного набора операций. То, что коммутация выполняется на уровне микросхем и даже отдельных атомов (а теперь уже и фотонных пучков), ничего не меняет по сути. Есть набор устройств, и мы коммутируем их, вводим определенную программу — а именно, выборку данных и команд, преобразование данных, передачу управления, взаимодействие с другими процессорами и контроллерами устройств, и т. д.

Получается, что работает процессор не сам по себе, а лишь в качестве нашего

помощника, автоматизирующего некоторые типовые операции, которые мы, в принципе, могли бы выполнить и сами, кабы у нас не было иных, более интересных занятий. Процессор лишь действует по заложенной в него жесткой программе, он так устроен, и никуда от этого не уйти. Выходит, компьютерное железо ничем не фундаментальнее трансляторов или интерпретаторов. А подлинным агентом оказывается живой человек.

Современному компьютерщику трудно представить себе то далекое время, когда не было персональных компьютеров и планшетов, когда про компьютерные сети мало кто слыхал, а напрямую управлять компьютером разрешалось только специально обученному персоналу. Новые поколения уже не знают, что такое перфокарта или перфолента, — разве что понаслышке, из школьного курса истории информатики. А я еще помню, как приходилось вместо терминала сидеть за клавиатурой перфоратора, и правильность набора проверять при помощи карты-читалки (текстовая строка на перфокартах появилась далеко не сразу, да и печать в ней была не безошибочной). О том, чтобы отлаживать программу в реальном времени, речи быть не могло. Не спасала и возможность вставлять в код отладочную печать, поскольку результаты работы выдавались на страшный агрегат под названием АЦПУ, а расход специальной бумаги для печати в организациях ограничивался по каждому отделу. В таких условиях программисту частенько приходилось самому исполнять роль машины и проигрывать типовые варианты выполнения программы до передачи задания оператору. Если в работе программы наблюдались странности — ошибки вылавливать надо было тем же методом, принимая роль процессора на себя. Идея о первичности человека как агента исполнения программ была, так сказать, прочувствована на собственных мозгах.

Можно возразить, что человек тоже представляет собой некоторое аналоговое устройство, набор нейронов в мозгу, — и что он тоже просто выполняет заложенную в него программу. Встает вопрос: кем заложенную? Если все сводить к чисто природной эволюции, к физиологии, — мы ничем не отличаемся от животных. Кому-то приятно чувствовать себя скотом — а я не хочу. Если допустить, что был какой-то сверхъестественный программист — мы впадаем в мистику, и это уже не философия, а сплошное суеверие.

Но есть еще один путь — допустить, что природа человека вовсе не определяется его физиологией, что человек — это всегда член общества, включенный в иерархию общественно значимых деятельностей, и именно это включение с младенчества формирует определенные способы функционирования мозга, программирует наши «мокросхемы» (wetware). Тогда получается, что главный программист — это общество в целом, процесс общественного производства и всевозможные культурные процессы. А если уж на то пошло, то и общество программируется чем-то другим — в его жизнедеятельности проявляются объективные законы развития мира в целом, неизбежно приводящие к возникновению разума в каких-то формах, которые потом исчезнут без следа, но повторятся в другом месте, в другое время, другими способами, на другом уровне.

Вот такой компот. Сложновато для непривычных.

Однако что с того современному компьютерщику? Какое нам дело до всеобщих материй? Нам бы скоренько сваять что-нибудь под нужды заказчика, чтобы на выходе было правильно и чтобы на руки за это поиметь.

А дело в том, что любые методы программирования возникают не от фонаря — они выполняют какую-то общественную задачу. Поэтому, скажем, одни языки программирования приживаются — а другие нет.

— Ничего подобного, — скажет крутой программер, — просто одни языки для чего-то удобнее других.

Но откуда берется это что-то, для чего оно удобно? Почему получается, что программисту или админу надо решать вполне определенный круг задач? Так исторически сложилось. А история — это история человеческой деятельности. Отсюда мораль: не следует абсолютизировать. Сейчас оно так, потом будет по-другому.

В частности, то, что единственным реальным агентом сейчас является человек, — не догма, а руководство к действию. Возможно, когда-то в будущем машины смогут сами выбирать правила своей работы, самостоятельно решать, как лучше оптимизировать свою

архитектуру. А там, глядишь, появятся и другие агенты, о которых мы пока ничего не знаем, и представить себе не в состоянии. Тем не менее, философски подкованный программист к такому повороту всегда готов.

ОС без наркоза

Сразу предупреждаю: речь не про это. Набившее оскомину противостояние одинокого гиганта Microsoft своре ближних и дальних родственников Unix подходит к концу. С выходом Windows 8 стало понятно: это агония. На рынке операционных систем Microsoft больше не игрок. Задрали медведя — и радуйтесь.

Но сначала все-таки об этом.

Лично я большой разницы между Windows/DOS и любыми вариантами Unix/Linux так и не усмотрел. И то, и другое одинаково годится для решения типичных задач в области информационных технологий, равно как и большинства задач прикладного характера. Просто делается это разными способами в разных системах — причем различия между отдельными представителями клана Unix иной раз куда значительнее, чем их общее отличие от MS DOS. Насчет удобства — это дело личных пристрастий. Люди по большей части ленивы, и кто привык к одному — не хочет привыкать к другому. Опять же, по личному опыту, неудобств всюду хватает — это приятности еще поискать… Какие-то вещи, которые под Windows делаются легким движением мыши, в юниксах становятся неимоверно громоздкими, даже при наличии графической оболочки, а уж тем более без таковой. И наоборот, что-то под юниксом делается влет, а винду надо еще поуговаривать. Но проблемы тут, скорее, субъективного порядка — просто работать в юниксах как под Windows нежелательно, и наоборот.

Много спорят об эффективности и безопасности. Опять же — никакой разницы. Хорошо написать приложение можно под любую ОС. А если тупо переносить алгоритмы из одной в другую — ничего кроме тупости и не поиметь. Что качается безопасности — дырявость Linux уже давно стала притчей во языцех, а другие родственники остаются относительно безопасными исключительно из-за малой распространенности, так что взломщикам просто нет интереса с ними возиться. К тому же в любой ОС основные угрозы безопасности возникают не от глюков в системе, и даже не из-за корявостей в приложениях, — на 99% и более это человеческий фактор. Если оператор страшно конфиденциальной базы данных будет где-нибудь в кафе громко диктовать по телефону свои учетные данные кому-то из сослуживцев, чтобы те зашли в базу и что-то сделали от его имени… Я молчу.

А вообще, система доступов в Windows куда изощреннее, чем в юниксах, и допускает сколь угодно тонкие настройки. Только заниматься этим администраторам лень, а рядовому пользователю и подавно. Более того, стоит кому-то из айтишников всерьез заняться раздачей слонов — тут же валится начальственное возмущение и требование вернуть все как было. Чайнику, из которого поступают деньги, неудобно, когда чего-то нельзя. В одной организации, например, заместитель исполнительного директора считал себя большим знатоком компьютерного дела — и требовал дать ему администраторские права ко всем базам данных, чтобы он мог с ними свободно играть. Легко представить, чем потом приходилось заниматься информационному отделу!

Ходят легенды о несравненной стабильности юниксовых систем, об их способности работать без сбоев многие годы… Но если урезать функциональность Windows до той же степени, оставить небольшое количество простых задач, — завалить систему будет столь же трудно. В моей практике были примеры, когда Windows Server работал более десяти лет — и хоть бы что. При полноценной графической оболочке и с установкой нескольких сторонних приложений. На практике сервера падают чаще всего от проблем с питанием или кондиционированием — иногда из-за чьих-то программных корявостей или несовместимости. Перезагрузка реально требуется только при установке некоторых приложений, или после критических обновлений — так, ведь, и юниксы приходится перезагружать при настройке ядра.

Ну, какие еще могут быть различия? Разные файловые системы? Чепуха. Под Windows

есть возможность работать с файловыми системами Unix/Linux, и наоборот, файловые системы Windows запросто монтируются в юниксах. Существующие несовместимости связаны не с принципиальной невозможностью сделать правильно, а с отсутствием желания что-то делать. Получается, что способы работы с файлами не привязаны к какой-то ОС, это всего лишь надстройки — каждое устройство требует подходящего драйвера.

Кстати, насчет устройств.

Непонятно почему, поклонники юниксов считают великим их преимуществом трактовку устройств как файлов, в отличие от DOS/Windows, где устройство — это устройство, а воспринимать его как файловую систему, вообще говоря, не обязательно. Единообразие? Это еще как посмотреть. Например, интернет-адреса начинаются с названия протокола, двоеточие, потом путь. Но именно это реализовано в Windows: название устройства (которое только для жестких дисков сводится к одной букве, и то не всегда) означает, по сути, ссылку на протокол доступа, и вполне логично отделять его двоеточием от последующего пути. А использовать прямой или обратный слэш — совершенно без разницы.

Отождествление устройств с файлами — пережиток первобытной эпохи, когда в качестве устройств выступали в основном клавиатура и терминал, изредка внешние накопители с такого же рода функциональностью. Но, например, представлять мышь файлом — это, положа руку на сердце, просто извращение, хотя технически ничего сверхъестественного. Куда разумнее здесь, например, парадигма потока, когда каждое устройство связывается с определенным протоколом обмена. Это годится и для будущих компьютеров, которые смогут считывать что-то прямо из мозга. Адресация типа «контейнер : путь» универсальна; она, например, хорошо согласуется с функциональным подходом, отвечая обычной схеме «функция : параметры», где в числе параметров могут быть и другие функции. Аналогично устроены ссылки на свойства и методы объектов, на поля и записи в базах данных и т. д. Уподобить объектную базу данных файловой системе можно — однако это ограничивает диапазон возможностей. Если в качестве контейнера (устройства) выступает оперативная память — мы возвращаемся (на новом уровне) к старой схеме адресации «регистр : смещение», где регистр уже не связан с физическим блоком, а задает некоторую логическую область (например, память, выделенная задаче или процессу; кстати, такая схема позволила бы значительно снизить риски сбоев в работе системы, вызванных утечками памяти). Устройство может быть реальным или виртуальным, оно может быть распределенным и т. д. Путь от этого никак не зависит, он связан лишь с организацией данных в пределах устройства.

Для пользователя ОС чаще всего представлена какой-то командной оболочкой. Традиция Unix явно оговаривает свободу в использовании таких оболочек и принципиальную возможность выбирать то, что больше нравится. Но, как ни странно, значительная часть критики Windows направлена именно против оболочки — хотя никому не возбраняется ее при желании поменять. Например, есть возможность загружать Windows в режиме DOS, и работать в режиме командной строки, вообще без графики. Есть возможность загружать в качестве оболочки Total Commander или иную программу (например, клиент базы данных). В конце концов, можно полностью скопировать ходовые оболочки Unix, обозвать устройства /dev/что-то-там — даже настройку при помощи конфигурационных файлов можно воспроизвести. Проекция системы доступов Windows на такую оболочку просто вырежет часть возможностей. Различия если и будут, то лишь при выходе за пределы собственно оболочки, при вмешательстве в работу системы. Значительная часть этой программы реализована под Windows в оболочке PowerShell, воспроизводящей многие черты юниксовых оболочек, вплоть до названий команд.[1]

Операционная система — это просто набор программ (служб), занимающихся распределением ресурсов и запуском задач. Этот набор совершенно одинаков для Windows и Unix, с точностью до реализации, — а дальше все зависит от того, что мы делаем. Исторически

[1] Разработчики Microsoft, по-видимому, надеялись привлечь таким способом часть сторонников Unix. На деле это воспринимается как признание поражение, как сигнал и повод отказаться от Windows. В мире дикой конкуренции не бывает благородных жестов.

эти ОС использовались по-разному, отсюда и кажущиеся расхождения; однако ничто не мешает и в той, и в другой решать одни и те же задачи.

Типичный образчик критики: вытаскивают на свет какую-нибудь команду Unix (типа grep) — и торжественно заявляют: вот видите, Windows этого не умеет!

Ну, во-первых, Windows это умеет, если написать соответствующую программу. Такого рода команды Unix — чисто оболочечное явление, от самой ОС никак не зависящие.

С другой стороны — зачем это уметь? Значительная часть возникающих в юниксах задач порождена особенностями традиционных надстроек над операционной системой. Unix появился на свет, когда практически все системные задачи сводились к обработке текста — а пользовательские приложения в основном занимались математическими расчетами. Логика системных оболочек приспособлена именно к этому. Для более сложных задач — требуется что-то изобретать, обходить ограничения. Так появляются платформы разработки типа Perl, Python, Ruby или PHP. В принципе, использование таких расширений не обязательно — достаточно уметь программировать на C и запускать специализированные программы с нужными параметрами. Минимализм — это в духе Unix; к сожалению, иной раз приходится всю жизнь расплачиваться за грехи молодости. Но не всегда. Например, чукотский язык прекрасно выражает опыт чукчи, оленевода или охотника. Можно говорить на нем и на отвлеченные темы, и даже есть примеры подобной литературы, — но зачем? Сподручнее выучить другой язык — пусть даже какие-то чукотские реалии на нем передать непросто…

Переключение консолей в Unix — наследие примитивности древних терминалов, умевших отображать только текст. Кичиться этим — все равно что с гордостью заявлять о врожденном заболевании. Много окон — гораздо удобнее, можно даже на одном экране параллельно работать с несколькими консолями. В принципе, ничто не мешает и под Windows связать с каждым окном свои права доступа — но опять же, зачем? Достаточно запустить свой сеанс для другого пользователя и переключаться при необходимости. Или запустить несколько виртуальных машин с разными системами. Кстати, еще до массового распространения графических оболочек под MS DOS существовали реально многооконные программы, и система управления окнами хорошо обкатана еще тогда. В системных приложениях Unix это почти не прижилось, там логика осталась на уровне одномерности командной строки. В наши дни графические оболочки — обязательное условие удобства и эффективности. На одном графическом экране можно отобразить объемы информации многократно превышающие возможности текстовой консоли. В частности, можно одновременно видеть разные срезы состояния сервера. Традиционно считают, что это излишне нагружает сервер, и особенно вредно в условиях интенсивного доступа по сети. Но чего стоит ОС, которая не в состоянии справиться со сколько-нибудь нетривиальной операцией, помимо пересылки байтов из одного места в другое?

Возвращаясь к коронной теме Unix, к обработке текста, можно сильно посомневаться. Древние средства набора, редактирования и форматирования текста в юниксоподобных системах создавались для преодоления ограниченности доступных технических средств. Сейчас с этими программами практически никто не работает — но логика осталась, иногда даже используются старинные форматы разметки. Поскольку все это увязано с ходовыми оболочками, пережитки прошлого сказываются на стиле работы и в наши дни — и влияют на выбор будущего.

Разумеется, можно написать красиво оформленную книгу в чисто текстовом режиме — например, есть такое гениальное изобретение как TeX. В конце концов, Web-страницы часто удобнее проектировать в обычном текстовом редакторе (без назойливой подсветки тегов), не привязываясь к благонамеренно встроенной кем-то куда-то методологии. Однако сколько-нибудь сложное форматирование требует в таком раскладе хорошо развитого воображения — или возможности постоянно контролировать, что получится. В итоге оказывается все же удобнее использовать полнофункциональный текстовый процессор — и здесь пока никто не переплюнул MS Word, подобно тому как Adobe Photoshop остается эталоном и пока

недостижимым идеалом для остальных графических редакторов.[2]

Массовая (пакетная) обработка текста возможно только там, где исходные данные устроены единообразно. Где просто невозможно задать их по-разному. Много ли таких ситуаций? Даже при администрировании серверов Unix каждый администратор вырабатывает свой стиль именования пользователей и компонент, свои правила размещения файлов. Подобная гибкость может, например, использоваться для повышения безопасности, ибо нестандартную конфигурацию труднее взломать. Однако в реальной жизни администраторы приходят и уходят, правила рождаются и умирают, — и в результате в конфигурации накапливается достаточное количество хаоса, чтобы сделать массовые единообразные замены занятием по меньшей мере рискованным. Попробуйте скачать с веба (из разных источников) пару сотен книг — и привести их названия к единообразному формату. Никакие регулярные выражения тут не спасут — остаются только крепкие выражения по поводу… Можно, конечно, потратить в несколько раз больше времени и написать универсальный преобразователь. Однако при скачивании очередной книги его универсальность тут же рухнет, и все придется начинать сначала. А полагаться на автоматику при изменении конфигурации серверов или рабочих станций — слишком дорогое развлечение.

Первобытные базы данных представляли собой особым образом организованный текст; точно так же создавались первобытные документы. В этих условиях автоматическая обработка текста была эффективна — она была и способом управления данными, и языком построения отчетов. С появлением более мощных движков и повсеместного перехода к двоичным форматам появились иные, более удобные средства. Хотя и здесь можно усмотреть пережитки далекого прошлого: например, реляционная идеология — это минимальное обобщение текстовых баз древности, только поля теперь могут быть и двоичными. Но, по большому счету, современная ОС есть программа работы с системными базами данных — и когда-нибудь именно это станет основной парадигмой в системном программировании. Разумеется, с учетом печального новаторства той же Microsoft и неуклюжести ссылок в «хороших» браузерах, вроде Firefox.

Затянувшийся экскурс в историю великого противостояния завершим маленьким идеологическим наблюдением. На самой заре компьютерной эры оператор, программист и пользователь компьютера совмещались в одном лице. Потом эти функции разделились, и для рядового пользователя использование компьютера свелось к небольшому набору типовых действий — при этом компьютер неизбежно приобрел черты загадочности и непостижимости, а для укрощения этой глюкавой уродины требовался шаман-системщик. Такова идеология Unix: компьютер — божество, а человек создан для того, чтобы молиться на него и приносить ему жертвы; сисадмин — великий жрец у алтаря.

Основатели Microsoft предприняли романтическую попытку научить компьютер делать наоборот: то, что нужно человеку, и в том стиле, как это удобно человеку. Отсюда особое внимание к интерфейсу. К сожалению, они не учли, что человек, на которого все это было рассчитано, к рядовому пользователю имеет минимальное отношение. Безрогие двуногие не хотели и пальцем пошевелить ради собственного удобства, им надо было все сразу на блюдечке. Куда проще свалить ответственность на потусторонние силы — кому-то о чем-то помолиться, кого-то о чем-то попросить… На этом стояли и стоят все религии. Например, агрессивность приверженцев Unix — в книгах по технологиям Microsoft меньше глупого самодовольства, они спокойно-информативны. Можно уважать специалистов Microsoft за то, что в атмосфере откровенной травли они серьезно работали, используя чей угодно опыт, в том числе и опыт Unix.

В Apple все начиналось на той же идее: компьютер для человека — но под человеком они понимали безмозглое существо, неспособное самостоятельно что-либо объяснить машине. В

[2] В каком-то смысле, все остальные приложения служат полигоном для обкатки тех или иных технических трюков — которые потом включаются в Word или Photoshop в качестве стандартных средств. Однако подобная монополия может доставлять значительные неудобства, когда разработчики Microsoft или Adobe вдруг решают исключить какую-нибудь полезную функциональность, или переделать интерфейс под новомодную идею, — пользователям оно совсем ни к чему, только осложняет жизнь… Но выбора нет.

результате компьютеры Apple стали вещью в себе, которую нужно просто принимать как есть, без попыток что-либо подстроить и приспособить. Маки, айфоны и айпады очень удобны — для тех, кому нужно именно такое удобство. Включил — и доволен.

Сухой остаток: есть разные ОС, все они делаю примерно одно и то же, и любые предпочтения — дело сугубо личное. Сравнивать одно с другим совершенно бесполезно. Это просто разные инструменты, и грамотный админ должен уметь пользоваться и тем, и другим.

И вот тут мы переходим к сути вопроса.

А суть такова: все эти юниксы, винды, макоси, андроиды и прочее — только верхушка айсберга, то, что видно каждому. А операционных систем значительно больше, многие сотни или даже тысячи. Чуть ли не всякая железяка в наши дни имеет встроенную ОС, и конфигурирование железа уже давно не сводится к перестановке джамперов — можно запустить специальную настроечную программу, или просто открыть страницу в браузере. Иногда производители не заморачиваются выпуском своих операционок, а просто ставят какой-нибудь урезанный клон Unix и настраивают под себя. Для того, чтобы энное количество байтов направить из одного места в другое, этого вполне достаточно.

С другой стороны, аппаратные комплексы требуют координированной работы устройств, и нужна ОС более высокого уровня, чтобы это организовать. Как минимум, в аппаратуру вшивается BIOS, и практически вся дальнейшая работа идет через эту мини-ОС — которая, впрочем, иногда вовсе не мини, в ней и элементы диспетчеризации, и командная оболочка… Поверх BIOS загружается еще одна операционная система, и на этом уровне уже можно предоставлять какие-то услуги конечному пользователю. Но частенько бывает, что такая ОС лишь создает операционную среду для работы многочисленных виртуальных машин, на которых и крутятся собственно пользовательские (или серверные) ОС, под каждой из которых можно, в свою очередь, запустить виртуальные машины разного уровня, от простой песочницы до полномасштабных компьютерных комплексов. Вариантов предостаточно. А если учесть, что локальные компьютеры разной степени виртуальности могут быть в разных комбинациях включены в любые сетевые структуры, — тут начинается полная необозримость.

Таким образом, есть не одна изолированная операционная система — а иерархия операционных систем. Как водится, уровни этой иерархии начинают перемешиваться, перенимать функции друг у друга — и возникают новые иерархические структуры, которые иначе делают то же самое. Это так называемые обращения иерархии. Появляются также механизмы коммутации разных иерархических структур — тоже разновидность ОС, — иерархическая система. Как и во всякой иерархии, разделение уровней относительно. Несколько ОС разных уровней могут образовать устойчивую комбинацию, которая работает как одно целое — иерархия свертывается. С другой стороны, на одном уровне можно выделить ряд задач в особую подструктуру, которая будет играть роль ОС по отношению к части приложений. Так происходит развертывание иерархии. Например, сервер Domino может быть установлен поверх Unix или Windows — при этом он берет на себя часть системных функций, определяемую его собственной конфигурацией, независимо от конфигурации ОС. В свою очередь, поздние версии Domino надстраиваются над платформой Eclipse, а, например, управлять генерацией Web-страниц Domino может либо напрямую — либо через любую систему интерпретации скриптов CGI (например, PHP). В какой-то мере, разные оболочки в рамках одной ОС могут считаться операционными системами промежуточного уровня; тем более это справедливо по отношению к виртуальным машинам вроде Erlang, Python или Ruby. В индустриальном программировании широко используются разного рода стандартные библиотеки (платформы разработки) — это еще один пример развертывания иерархий. Наконец, на аппаратном уровне есть иерархия микропрограмм как переходное звено от собственно железа к BIOS и ОС высокого уровня.

При таком подходе становится просто глупо провозглашать какую-то одну ОС верхом совершенства и с презрением взирать на все остальные. Надо идти по пути интеграции. Не просто сосуществовать, а всячески способствовать свободному общению. Но не в том смысле, чтобы тексты на любых языках сопровождались переводом на некоторый «универсальный» язык — это означало бы гегемонию одной культуры над другими, и вместо общения остались

бы распоряжения сверху. Речь идет именно о двустороннем переводе — и о возможности свертывать иерархию, устранять промежуточные ступени. Только тогда появится возможность разработки аппаратуры и приложений на любой программной и элементной базе, с возможностью адаптации старых приложений без потери эффективности.

Когда одна система пытается противопоставить себя всем остальным — это сепаратизм. А сепаратизм — явление сугубо экономическое, орудие конкурентной борьбы. Вот и выходит, что полезное многообразие вырождается в бессмысленное противостояние, а любая ценная находка опошляется, становится оружием в экономической войне. Но войны — для дикарей. Разумным людям воевать незачем. Они все делают вместе, для общего блага, а не против друг друга.

Регулярные выражения

Это смешно. Стоит где-нибудь прилюдно упомянуть регулярные выражения — у народа делаются квадратные глаза и перехватывает дыхание. Кто ими пользуется — немы от восторга. Кто ими не пользуется — благоговейно трепещут. И те, и другие почитают их как божественное откровение, как высшее достижение всяческой разумности (пардон, чуть не сказал: «высшую меру»).

А собственно, что такого? Всего-навсего язык для задания шаблонов поиска текстовых строк — ни на что другое он не годится, хотя иной раз осмеливается претендовать. Язык древний, набитый под завязку пережитками прошлого. Громогласные заявления о его мощи и удобстве — не более чем пропаганда. Да, свою работу он выполняет и вполне годится для использования в тех приложениях, на которые он рассчитан. Как и любой другой язык. Если кто-то к этому привык, он будет считать это удобным. Но существуют и другие возможности, которые, по большому счету, ничем не хуже.

Регулярные выражения придуманы для обслуживания операционных систем семейства Unix — управление системой в них сводится по большей части к редактированию текстовых файлов, и отчеты о состоянии системы также представляют собой текстовые файлы в неудобочитаемом формате — так что потребность в средствах выкапывания их таких файлов нужной информации вполне понятна. Традиция оказалась весьма живучей — и тестовый формат до сих пор используется в большинстве систем и приложений для журналирования, регистрации текущих событий и действий. Хотя, например, в системах управления базами данных было бы логичнее заносить данные в специальную базу в структурированном виде, а отчеты строить стандартными средствами. Одно время серьезным аргументом в пользу регулярных выражений служил исключительно текстовый характер HTML — действительно, анализ статических страниц прекрасно вписывался в идеологию поиска по шаблону. Однако реальные парсеры все-таки работают по-другому, намного эффективнее. А потом и сама возможность примитивной обработки текста из HTML испарилась, ибо основное содержание страниц нынче формируется динамически, генерируется на лету в реальном взаимодействии с пользователем. Искать что-то по шаблону в разнесенных по разным файлам скриптах и определениях стилей — дело неблагодарное. Допустим, на странице светится картинка с определенным названием — но названия как такового в исходном тексте страницы нет, оно комбинируется из отдельных элементов непосредственно при вызове изображения. Что тогда искать?

Современный документ — это не только, и не столько текст, сколько сложная комбинация объектов самого разного уровня. В них встроены запросы к базам данных, изображения, мультимедийные вставки, конструкторы формул, документооборотные элементы, средства визуализации, сценарии и формы. Проку от регулярных выражений здесь совершенно никакого. Документы обрабатываются соответствующими приложениями, в которых есть встроенные средства поиска и замены, плотно интегрированные с языком прикладного программирования. Если и требуется в каких-то случаях искать куски текста — особых наворотов тут не нужно, достаточно простейших шаблонов. Тем более, что искать приходится

не просто строки, а строки в определенном контексте — например, с заданными параметрами форматирования. Стиль Unix до сих пор заставляет программистов запихивать все, что надо и не надо, в текстовые файлы — и потом с чувством глубокого удовлетворения заниматься их фильтрацией по регулярным выражениям. Система работает сама на себя — не по уму, а ради самосохранения.

Никто не спорит, в ряде случаев поиск по шаблону — вещь очень даже необходимая. Например, запросы SQL представляют собой чистой воды поиск по шаблону. И логично сочетаются с использованием регулярных выражений. В других системах управления базами данных есть свои языки построения шаблонов. Разумеется, это больше нужно администратору и программисту; рядовой пользователь разбираться с синтаксисом не станет, ему надо предложить набор специализированных форм поиска и фильтрации. Текстовые запросы пользователей — это всегда голые фрагменты текста, без каких-либо специальных символов. Обрабатывать такие запросы — дело прикладного программиста. Например, чтобы искать не только по ключевым словам, а еще и предлагать что-то весьма или отдаленно похожее. Конечно, в большинстве платформ есть встроенные средства ассоциативного поиска — и в регулярных выражениях соответствующий синтаксис имеется. Но на практике такие базовые средства почти всегда оказываются недостаточными или неуместными — и приходится изобретать свой велосипед для каждой конкретной задачи.

С точки зрения программиста — регулярные выражения тоже далеко не идеал. Компактность записи — пережиток первобытности. Сколько-нибудь сложные конструкции не всунешь в командную строку, в которой и так все слишком длинно, даже при активном употреблении алиасов, да еще и пайпить надо несколько раз… Файлы сценариев под Unix — минимальное обобщение командной строки, страдающее теми же недостатками. Вот и приходится жертвовать удобочитаемостью в угоду минимализму. Однако даже под Unix вполне допустимо вынести определение правил поиска в отдельный файл — и просто подгружать его при необходимости; при этом язык задания шаблона можно сделать вполне дружелюбным, с возможностью комментирования, с модульной организацией и сколь угодно развитыми средствами динамического формирования шаблонов и записи результатов в указанные каналы. В конце концов, и в стандарте регулярных выражений чистой декларативности никак не получается — а уж тем более в расширениях, используемых императивными языками типа Perl. Слышу возражения: развернутые сценарии поиска вредят эффективности. Но это уже зависит от интерпретатора, и со способом задания шаблонов никак не связано. Когда-то и регулярные выражения затыкались на сколько-нибудь нетривиальных задачах. Сейчас имеющиеся движки оптимизированы, и проблем почти нет. Минимализм как индивидуальный стиль — это нормально. Удобства в виде дырки в полу кого-то вполне устраивают. Но это не повод для отмены унитазов.

На практике сложные правила поиска практически никогда не нужны. И развитый синтаксис регулярных выражений — скорее помеха, чем подспорье. Если есть возможность обозримой визуализации результатов поиска — проще последовательно включать несколько фильтров, чем впихивать все сразу в один шаблон. Если такой возможности нет — претензии к используемой платформе.

Часто поиск по шаблону используется для массовой замены одного текста другим. Вот тут я, извините, не любитель. В крупных организациях, где сложились строгие стандарты записи конфигураций, это в какой-то мере может пригодиться. Но в реальной жизни стандартов мало кто придерживается, и массовые замены запросто могут погубить вполне работоспособную информационную сеть. Разнобой в данных — неизбежное зло. Поэтому мощные средства массовой обработки, вроде регулярных выражений, — это, как говорят в физике, превышение точности. Последовательные преобразования с контролем на каждом этапе — это спокойнее. Идеальный вариант — устранение самой необходимости массовых замен. Если что-то многократно повторяется, это свидетельство неэффективной организации данных. Правильно — когда каждый элемент данных задан один раз в одном месте. Тогда достаточно его поменять один раз — и все остальное получится автоматически. Касается это не только атомарных данных, но и форматов, и функциональности. Если меня не устраивает

распределение доступов в системе, я должен иметь возможность задать где-то алгоритм — и не в скрюченном стиле регулярных выражений, а как полноценное описание функциональности. А дальше дело операционной системы разобраться с этим и применить. Очевидно, в масштабируемых системах потребуется иерархичность.

Попытки создания такого рода систем в истории были. Называется это — Microsoft. Предложенная ими иерархия политик — прототип чего-то из далекого будущего. Но в нашем времени умные решения не востребованы. Зачем? Пока хватает диких админов, превращающих неудобоваримость системных решений в неиссякаемый источник средств к существованию. А если все просто и разумно — кто же тогда станет деньги платить? Опять же, и программисты всегда при деле — на каждую бочку приходится строгать свою затычку. И чем запутаннее язык программирования — тем ценнее высококвалифицированный специалист. Вот тут регулярные выражения — самое оно.

Не надо мне возражать — я ни с кем не спорю. Это мое личное мнение, на которое я, надеюсь, имею право — после десятков лет работы с компьютерным чем угодно. Да, в моей практике не было необходимости создавать высоконагруженные комплексы, в которых каждая микросекунда на счету. С другой стороны, есть подозрение, что чрезмерная нагрузка — результат неправильной организации, и следовало бы поискать что-то более эффективного, дабы потребности насиловать систему просто не возникало. Бывает, конечно, что пиковые нагрузки связаны с внешними обстоятельствами, с ажиотажным спросом на какой-нибудь ресурс, подобно атакам типа DDoS. Но в разумно организованной системе есть на тот случай адекватные технические решения (распределение нагрузки, ограничение доступа, оптимизация обслуживания запросов и т. д.) — если их почему-либо пока нет, то это надо делать. А не упиваться способностью пропустить через узкие ворота миллион дураков. Стоит ли идти на поводу у моды? Конечно, если речь идет только о том, чтобы деньги получить, — тогда без вопросов, выражайтесь в меру собственной испорченности. Но если мечтается создавать не только то, что продается, а еще и нечто разумное — другой разговор.

Общность и общение

Параллелизм в современных информационных системах — на пике последней моды. Мы уверенно продвигаемся ко все более распределенным вычислениям — как в смысле интеграции разных аппаратных средств в глобальном масштабе, так и внутри каждой отдельно взятой железяки. Поверх железного параллелизма надстраиваются эмуляторы разного уровня, и даже при наличии одного-единственного процессора операционная система умело создает видимость параллельного выполнения нескольких задач, каждая из которых может запускать дочерние процессы (потоки). Названия в разных платформах разные — суть одна.

Как водится, если мы выпустили из банки тараканов — пора учиться их ловить. И вот тут возникают идеологические разногласия. Каким образом управлять этим беспокойным хозяйством, чтобы в результате продвигаться к намеченной цели, не впадая в ступор при виде стремительно размножающихся сущностей?

Простой вариант — разделение труда, разбиение большой задачи на какое-то количество задач поменьше, каждая из которых может выполняться независимо от других; при этом где-то есть ящик, куда сваливаются результаты работы всех участников, и один из участников занимается составлением из этих кусочков целостной картинки.[3] В простейшем случае, картинка получается сама, если результаты каждой подзадачи сразу направляются в нужное место; в такой *свернутой* иерархии функция составления мозаики все же присутствует — только сделано это кем-то еще до запуска процессов-компонент.

Математической моделью такого распараллеливания служит объединение семейства непересекающихся множеств. Мы разбиваем исходное множество на части, те могут в свою

[3] Легко усмотреть аналогию с передачей пакетов в сетях TCP/IP: отдельные фреймы могут приходить в любом порядке, а принимающая сторона расставляет их по местам и контролирует полноту данных.

очередь делиться как угодно — и возникает некоторая древовидная структура, а в сумме все равно получается исходное множество. Отсюда можно усмотреть условия применимости подобной технологии:

1. возможность представления исходного множества в виде объединения какого-то числа непересекающихся классов; вообще говоря, это дело нетривиальное, и могут быть топологии, в которых оно просто никак не пройдет; однако во многих практических ситуациях реализуемо даже более сильное условие — существование базиса, максимально детального разбиения;
2. стационарность разбиения — границы между подзадачами не меняются в ходе совместной работы (когда это не так, начинается так называемое экстремальное программирование);
3. независимость обработки элементов каждого подмножества от работы в других подмножествах; в противном случае результат существенно зависит от случайных вариаций порядка вычислений в разных подсистемах — это модель квантового вычислителя;
4. процесс сборки результатов не зависит от способа распределения и от результатов вычислений; иначе говоря, общий вид целого задан заранее, так что совершенно без разницы, будем мы компоновать целое параллельно или путем последовательности запросов; нарушение этого требования в математической модели соответствовало бы своего рода конструктивистской позиции, когда объединение множеств (или что-то еще) понимается как действие, а не мгновенный акт, — и далеко не факт, что всегда одно возможно отождествить с другим (из курса анализа известно, что различные подпоследовательности могут сходиться к разным пределам).

Разумеется, список неполон, и копать можно до бесконечности. Но если слишком интенсивно копать — совсем уроешься, и всегда важно где-то остановиться, предположить то, что никогда не может быть строго доказано. На этом целиком стоит величественное здание современной математики — в том числе вычислительной.

Противоположный подход — пустить дело на самотек, дать возможность нескольким процессам заниматься одним и тем же, во славу свободной конкуренции.[4] И надеяться, что в итоге все это придет к чему-то, пусть даже не очень разумному — но хоть сколько-нибудь удобоваримому. Мы уже не заботимся о совместности и независимости, нам бы только удержаться в рамках глобальных параметров… А кто параметры задает? Опять же, какие-то из участников всей этой деятельности. То есть, возникает социальное расслоение, когда одни участники оказываются почему-то главнее других. Вариантов такой организации много — но всем хорошо известно, к чему она приводит на практике. Неизбежность кризисов в развитии капиталистической экономики давным-давно установлена товарищем Марксом, и мы сейчас имеем все возможности испытывать справедливость его выводов на собственной шкуре. Но вот какая штука: кризисы, как правило, не меняют глобальных параметров самого верхнего уровня, и тем самым базарная экономика в целом оказывается относительно устойчивой. Разумеется, до поры до времени. Пока совсем в разнос не пойдет. Чтобы максимально оттянуть этот момент, появляются гибридные схемы, допускающие централизованный контроль в ограниченных масштабах. И здесь мы попадаем в главное русло (mainstream) современных технологий распараллеливания.

Ходячая концепция многозадачной системы предполагает, что все процессы (потоки) иерархически упорядочены, и каждый процесс так или иначе контролирует дочерние процессы, и может при необходимости ограничить их активность или свернуть вообще. Запуск процессом чего-то на том же (или более высоком) уровне возможен — но не приветствуется; по смыслу это эквивалентно передаче дочернего процесса под юрисдикцию более высокой инстанции, а значит, разумнее сразу организовать такие разветвления как запросы наверх, с подобающим

[4] То есть, в данном случае concurrence = competition.

раболепием.

Я не страдаю гипертрофированным ригоризмом и не собираюсь различать здесь задачи, процессы, потоки — и любые иные фрагменты кода, предназначенные для параллельного (или квазипараллельного) выполнения. Для меня это все синонимы. Разные названия традиционно использовались на разных уровнях вычислительной деятельности, и все их различие состоит лишь в том, как отдельные куски стыкуются меж собой. Поскольку различные процессы, вообще говоря, взаимозависимы. Каждому управляющему процессу приходится как минимум отслеживать состояние всех дочерних и рапортовать наверх о трудовых успехах. Дебаты в компьютерной науке в основном развертываются вокруг способов реализации вертикального и горизонтального взаимодействия.

Одно из поветрий: массы свихнулись на инкапсуляции — считается, что в идеальной вычислительной среде процессы совершенно изолированы друг от друга и общаться им дозволено только путем отправки друг другу сообщений. Это в чистом виде императивный стиль программирования — однако он безоговорочно принимается и большинством декларативных языков как высшее достижение компьютерной культуры.

В древности было не так. Исходно параллельные процессы взаимодействовали через совместно используемые области данных: каждый процесс мог что-то записать в такую область, а другой процесс на основании анализа имеющихся записей судил о состоянии коллег по цеху и возможностях дальнейшего плодотворного труда. Семафоры, флаги, области обмена… Сейчас принято все это высокомерно презирать и всячески от этого избавляться. И тем самым демонстрировать собственную глупость.

Чтобы две материальные (или программные) системы могли обмениваться сообщениями, они должны совместно использовать нечто третье, способное взаимодействовать как с одним собеседником, так и с другим. Остается этот посредник на одном и том же месте или путешествует по электронным цепям и коммуникационным сетям — совершенно не принципиально. Области обмена обеспечивают прямое, контактное взаимодействие. Пересылка сообщений — аналог дальнодействия, которое, как демонстрирует современная физика, есть не более чем коллективный эффект, усреднение всевозможных локальных контактов. Впрочем, совсем современная физика опять вытаскивает идею нелокальности — но когда-нибудь всяческие суперструны и гипербраны также будут поняты как вполне локальным образом связанные объекты — и потребуют освоения новых форм нелокальности, и так далее, — поскольку дискретность и непрерывность суть категории диалектические, две стороны одного и того же, — и одно неразрывно связано с другим.

Для того, чтобы передать сообщение, процесс должен его куда-то записать; получатель должен сообщение откуда-то прочесть. Чем это отличается от разделяемых данных? По сути — ничем. Однако апологеты взаимоизоляции поднимают на щит принцип передачи параметров только по значению — и подсовывают получателю не исходный текст сообщения, а его копию, наивно полагая, что тем самым пресекают любую возможность воздействия получателя на данные отправителя. Но если сам получатель не может изменить исходное сообщение — он вполне может попросить об этом отправителя, и тот имеет право поддаться на уговоры. Тут господа-ригористы звереют и требуют, чтобы исходный текст был уничтожен сразу же после отправки, чтобы и отправитель не смог ничего с ним поделать. Однако отправитель не дурак — он может сгенерировать новое сообщение, в точности воспроизводящее прежнее, но уже с поправками, предложенными кем-то из собеседников. И все опять вернется к разделяемым данным, с учетом того, что данные — это не область памяти или сектор на диске, а нечто идеальное, что вполне может кочевать из одного места в другое, меняя носитель, но не меняя содержания. То, что Наполеон родился на Корсике, никак не зависит от количества и качества копий этой информации в миллиардах голов, книг или электронных носителей. И если кому-то вздумается усомниться в историческом факте и посчастливится найти более достоверные данные — он так или иначе внесет поправку во всеобщее понимание сути вопроса, а всевозможные материальные носители будут приведены в соответствие.

В мире нет ничего, что можно было бы надежно изолировать от всего остального. У любых двух объектов всегда найдется что-то общее — начиная с того, что они всегда

принадлежат одному и тому общему для всех миру, а других миров по определению нет. Способов взаимодействия разных вычислительных процессов не меньше, чем всевозможных природных прототипов. Одна только квантовая теория дает кучу разнообразнейших примеров, даже в рамках стандартной модели. Например, в каких-то условиях процесс может вести себя как частица: передача сообщения тогда происходит путем распада частицы (завершения процесса) и рождения новых частиц (дочерних процессов), которые могут поглощаться другими частицами (получателями сообщения), меняя их состояние или вызывая новые акты рождения-уничтожения. Бывают ситуации, когда исходная частица не исчезает при испускании частицы посредника, а лишь меняет состояние движения и может потом принять «ответное сообщение» от другой частицы; так возникают коррелированные квантовые системы — аналог нынешних технологий распараллеливания. Но это частный случай, а реальность гораздо богаче. В качестве одной из альтернатив можно взять не частицу, а волну, которая взаимодействует сразу со всеми частицами, но может поглотиться одной из них, — похоже и на очередь сообщений Windows, и на обычный программный переключатель, и на некоторые сетевые протоколы… Наконец, есть вариант интерференции волн — и это возвращает нас к технике разделяемых данных. Но помимо квантов существует много другой физики, а еще и химия, и биология, и психология, и культурология, и экономика… Так что вычислительным средам есть «делать жизнь с кого».[5]

После всех этих глобальных соображений, давайте пофилософствуем помельче, на уровне простых моделей.

В технология обмена сообщениями часто предполагается, что сообщение посылает один процесс другому процессу — а стало быть, один процесс должен знать о существовании другого и о доступных партнеру каналах коммуникации. Схема это содрана с простейших актов общения у людей. Самые примитивные уровни такого общения — отправка сигналов и команд. В более развитом варианте, возможен диалог, а значит, собеседники должны принадлежать к некоторой общности более высокого уровня — и в частности, пользоваться одним и тем же протоколом общения (тоже разделяемые данные!), говорить на одном языке. Минимальное обобщение такого парного взаимодействия — конференция, когда одно и то же сообщение поступает сразу нескольким участникам, и те имеют возможность его активно комментировать (или реагировать действием). Групповое общение, конечно, нарушает нравственную чистоту — но в ряде случаев оказывается намного более эффективным. Допустим, на сервере подняты несколько демонов, обслуживающих распределенные приложения; если по каким-то причинам связь с другими серверами прерывается — каждый такой процесс выясняет это самостоятельно, пытаясь установить внешний контакт и нарываясь на неудачу. Если же потеря (или восстановление) связи порождает сообщение, доступное сразу всем демонам, они не будут тратить время и ресурсы на дозвон, а займутся более полезной деятельностью (например, поиском альтернативных способов коммуникации или организацией буферов — так сказать, заготовкой консервов). В более развитой системе конференций сообщения обрабатываются всеми членами некоторой группы процессов и каждый участник адресуется ко всем остальным. Например, если у кого-то из участников есть информация о причинах нарушения связи и предположения о возможных сроках ее восстановления, это может влиять на организацию работы многих процессов, в том числе и того, который занимается обслуживанием внешних коммуникаций.

В общем случае процесс вовсе не обязан знать кого-то еще, ему достаточно самого факта присутствия других процессов в общей среде — и сообщение отправляется не кому-то конкретно, а сообществу в целом, в открытое пространство или в отдаленное будущее. Процесс просто взаимодействует со средой, выражает свое отношение к происходящему, реагирует на глобальные события или сугубо личные переживания. Ему, по большому счету, дела нет до того, прочтет кто-нибудь его послание или нет. Он занят своей работой и выполняет ее по мере наличия ресурсов, от волеизъявления американского народа до банального электропитания. Это декларативное дополнение к методу направленной передачи сообщений.

[5] А хотя бы и с товарища Дзержинского — вполне достойный прототип.

Базовый цикл жизнедеятельности процесса выглядит тогда следующим образом: проверяем наличие условий для работы → если есть возможность, выполняем какие-то из доступных операций → отсылаем результат во внешнюю среду. И все повторяется сначала. Если условий для продолжения нет, включается один из альтернативных способов деятельности — например, переход в состояние ожидания, или активный поиск ресурсов. Где? Да в той же самой среде, в которой живут и трудятся самые разнообразные процессы. Это их главный и непосредственный разделяемый ресурс, косвенным образом реализующий общность данных на всех остальных уровнях. И здесь появляется иерархия общения, связывающая процессы по самым разным параметрам, от элементарной синхронизации до обмена деятельностями (распределение нагрузки, кластеризация и т. д.), а через это — переход к совместной постановке задач, включая взаимное программирование.

В качестве примера иерархической организации можно упомянуть память. Элемент памяти вполне может работать как агент, получающий запросы на чтение, запись или управляющие действия — а результатом становится некоторое изменение во внешней среде, доступное другим процессам, которое те интерпретируют в зависимости от своих текущих потребностей — как индикацию работоспособности, как запрос на обслуживание или как сохраненные данные. Эта схема буквально воспроизводится на аппаратном уровне. В системах программирования, как правило, существуют специальные средства управления памятью, и здесь запросы посылаются не непосредственно в аппаратную среду, а в специализированный программный модуль, который опять-таки принимает запросы и выдает в ответ какие-то данные; но кроме этого занимается распределением и освобождением памяти, предотвращая возможные конфликты программных сущностей (процессы внутри программы). Что-то из этого реализовано на уровне языка программирования, какие-то компоненты включаются в исполняемый код. Различие этих уровней весьма и весьма относительно — поэтому можно полагать, что в общем случае процессы верхнего уровня не работают с памятью напрямую, а только посылают запросы в некоторую программную среду и проверяют наличие требуемых данных; если данные есть — они могут быть ассоциированы с другими данными (например, присвоение значения переменной создает связь между данными, представляющими значение, и данными, представляющими переменную) или переданы другому процессу путем перезаписи в условленное место внешней среды (например, в стек). Тут важно, что опосредованный характер получения и записи данных позволяет отвлечься от конкретного физического представления: совершенно неважно, будут ли данные храниться в оперативной памяти, или выгружены в своп, записаны на флешку или переданы по сети на деревню дедушке. Модуль управления памятью должен уметь прозрачным образом обрабатывать все эти возможности, а отдельному процессу вовсе не обязательно разбираться в деталях инфраструктуры — он стандартным образом взаимодействует со средой и получает все необходимое, — или не получает и имеет полное право болезненно на это реагировать. В общем случае в ту же схему включаются и неэлектронные носители информации, а также носители более высокого уровня — распределенные процессы в аппаратных комплексах и в сетях. То есть, данными может быть нечто такое, что принципиально непредставимо определенным состоянием физической (а также биологической или социальной) системы, но требует постоянного изменения этой системы, перехода из одного состояния в другое в соответствии с некоторым законом, который и представляет собой содержание памяти, некоторое значение, которое процесс более низкого уровня имеет право запросить у своей программной среды, получить ответ и обработать в соответствии со своими надобностями. На практике оказывается, что таких данных нам по жизни требуется очень много — с некоторого уровня развития цивилизации они начинают преобладать над элементарными данными, представимыми состояниями физических систем.

Что же получается? Неправда, что у объектов, процессов и агентов нет общих данных — они всегда есть, они образуют их общий мир, часть единственного всеобщего мира. Чтобы общаться, не надо терзать ближних своими посланиями — достаточно просто делать свое дело, а другие будут судить по результатам. Когда я пишу этот текст, я не имею в виду конкретного читателя — более того, чем больше разных читателей, тем интереснее моя работа. Но мне в моем деле это не пригодится — для меня существует только возможность отправки сообщений,

некоторый обобщенный читатель, субъект более высокого уровня, объединяющий всех потенциальных читателей и возникающий из той же самой общественной потребности, которая заставляет меня стучать по клавишам.

Обычный протокол обмена сообщениями в таком подходе отнюдь не исключается. У каждого процесса есть круг его знакомых и родственников, под которых он преимущественно и затачивает свое поведение. Для кого-то этого вполне достаточно. А у кого-то в душе есть жажда чего-то большего, ощущение причастности к большому и разнообразному миру. Другими словами, одни программы работают сами на себя, другие — на общую задачу. Одни системы созданы для самосохранения и самовоспроизведения, другие — как этап на пути к другим, более развитым системам. Понятно, что в открытой среде существовать труднее, и потому устойчивости зачастую добиваются через изоляцию. Но зато в открытой среде возможна самоорганизация, и возникающие проблемы способствуют глобальному развитию, — а в замкнутой среде если что и растет, так это энтропия.

Модульность и интеграция

В мире рыночной конкуренции крупные поставщики программного обеспечения больше заботятся о своих доходах — а польза и удобство оцениваются в миллионах долларов, которые на этом можно сделать. В результате — гонка обновлений коммерческих программ, с добавлением новой функциональности не по необходимости, а лишь для того, чтобы успеть занять еще одну нишу. То, что исходно было простым и доступным, понемногу становится громоздким и неповоротливым, и пользователям все труднее добраться до тех нескольких основных операций, которые когда-то им пришлись по душе — поскольку интерфейс целиком заточен под новые «фишки» и все старье заметает под ковер, за ненадобностью.

Тут можно, конечно, начать возражать, что, дескать, сложные задачи требуют столь же сложных программ, и что мухобойность программных продуктов напрямую увязана с разнообразием и богатством возможностей… Что поделаешь — за гибкость и мощь надо платить. Кому-то, скажем, MS Office нужен исключительно в качестве печатной машинки с расширенными возможностями; для другого это средство создания презентаций; а кто-то вполне счастлив встроенными средствами интеграции с вебом. Большинство пользователей и понятия не имеют о написании скриптов — тогда как для разработчика приложений это предмет первой необходимости.

К сожалению, это путь быстро заводит в тупик. Всякая интегрированная среда интегрирована только внутри себя — и совсем не вяжется с другими подобными средами. В мире рыночной конкуренции иначе и быть не может, ибо ни один здравомыслящий производитель не станет выставлять на публику корпоративные технологии ради стопроцентной совместимости. Объективная необходимость заставляет-таки конкурентов садиться за стол переговоров — однако дело в самом благоприятном случае кончается лишь еще одним интеграторским решением, включая набор стандартных протоколов, в какой-то мере открытые форматы данных и некоторая общность принципов макропрограммирования. Все это связывает разные продукты чисто внешним образом, не оказывая особого влияния на исходные технологии.

А в идеале хотелось бы иметь такую среду, в которой работало бы любое приложение, независимо от изготовителя и возраста; пользователь тогда сам бы смог определять, что ему по жизни требуется и динамически подключать или отключать необходимые модули по мере надобности, работая в подвижной, ориентированной на текущие задачи конфигурации. В каждый момент под рукой оказываются только необходимые инструменты — подобно тому, как ненужные пункты меню скрываются в некоторых системах, с возможностью развертывания по требованию. И не нужно заставлять пользователя (или админа) танцевать с бубном всякий раз, когда вдруг приспичило задействовать новую фичу, — и потом молиться, чтобы подключение обошлось без последствий. Надо иметь возможность просто указать требуемую функциональность и подтвердить команду установки. А интеграция новых приложения с уже

имеющимися и встраивании в привычный пользователю интерфейс — автоматически обеспечивается установщиком.

Да, конечно, информатика движется в этом направлении, и существуют механизмы фоновой загрузки и подключения новых модулей из единого хранилища — особенно для мобильных приложений. Однако полноценной интеграции, в общем-то, нет — поскольку каждое приложение просто разрабатывается в нескольких вариантах, под самые ходовые платформы. Каждая платформа при этом остается замкнутой в себе и не допускает чужеродных элементов.

Безусловно, с ростом вычислительных мощностей, каждый будет в состоянии установить на одном устройстве несколько виртуальных машин, связать их чем-то вроде локальной сети — и пожалуйста, запускайте любые приложения в их собственной виртуальной среде. Однако такое решение — чистая эклектика, и особой гибкостью не отличается. Особенно учитывая, что при этом пользователь должен учиться работать в разных операционных системах — вместо того, чтобы, наоборот, разные платформы объединить в индивидуально скомпонованной среде.

Некоторые платформы, в принципе, позволяют выстраивать оболочки с высокой степенью интеграции. Пожалуй, самый впечатляющий пример гибкости и удобства — Windows XP mode под Windows 7 (к сожалению, это уже вести с того света); в VMWare Workstation есть нечто подобное под названием unity mode — но, конечно, уровень интеграции уже не тот: это не расширение системы, а всего лишь надстройка. Понятно, что различие в форматах данных и системе ссылок никак не способствует универсальности подобных интеграторских решений. В идеале пользователь должен иметь возможность выращивать свою собственную операционную систему из независимых конструктивных блоков, не заморачиваясь вопросами совместимости.

Если по-простому, допустим, что мне нужна такая-то возможность из Windows, что-то мне больше нравится в Linux, отдельные фишки хотелось бы выдернуть из VAX VMS, а что-то, может быть, из AS/400, или из какой-то версии макинтоша либо того же андроида. В довесок мне бы еще основные инструменты фотошопа чуть подкрасить странностями GIMP, да еще скомпоновать текстовый процессор на базе MS Word + Scientific TeX, и чтобы все это было совместимо с PDF в обоих направлениях. Есть у меня такая возможность набрать только те функции, которые меня интересуют? На нынешний момент — ровно два раза! Если бы каждый компонент каждой компьютерной платформы (любого уровня) можно было бы отделить от остального и аккуратно приклеить к любому набору других модулей, каждый мог бы обрести систему мечты, в которой есть только то, что нужно — и ничего лишнего, да еще и в единообразном и под себя заточенном интерфейсе.

Понятно, что это на практике означает совершенно новый подход к разработке программного обеспечения. Прежде всего, функциональность следует отделить от реализации. А для этого нужен *универсальный язык* для представления пользовательских нужд. То есть, *по-настоящему универсальный*, а не просто принятый в качестве взятого с потолка стандарта. Чтобы можно было абсолютно новую парадигму, которая никому и в голову не могла бы прийти десяток лет назад, выразить на том же самом языке, а не изобретать нечто более подходящее. В этом смысле универсальный компьютерный язык будет сродни естественным языкам, которые многие века легко приспосабливаются к любым изменениям в культуре. Реализация требуемой функции предполагаем *перевод* с универсального языка на какой-либо из специализированных языков, удобных для данной конкретной разработки. Такой перевод заведомо неоднозначен, и приведет к разным реализациям — важно, чтобы требования пользователя при этом были соблюдены. В частности, разные компании могут как угодно использовать корпоративные технологии — но при этом снабдить свои продукты набором приспособлений, позволяющих разговаривать с другими приложениями на общем языке. Нечто вроде семейства динамически конфигурируемых виртуальных машин на базе единой платформы интеграции.

Элементы такого подхода понемногу проникают в умы программистов, чему немало способствует растущая популярность декларативных и функциональных языков. Можно вспомнить и о различных платформах Web-разработки, о системах управления контентом и т. п. — все это придает полезной гибкости индустриальному программированию. Встроенная

в них возможность независимого параллельного использования разнородных средств — это прототип нового стиля в информатике. Для универсальности не хватает свободной трансляции из одной платформы в другую, в дополнение к собственно заимствованиям.

Разумеется, подлинная универсальность должна распространяться на все области деятельности, включая унификацию на уровне железа. Больное место слишком современных операционных систем — отсутствие драйверов для какого-нибудь периферийного старья, вроде звуковой или видеокарты, выпущенной бог знает кем уже несуществующим много лет назад. Когда Linux пробивал себе дорогу к массовому пользователю в качестве конкурента MS Windows, его пропагандисты пытались привлечь новых адептов громкими заявлениями о невиданной доселе совместимости с любыми железками; к сожалению, все это осталось не более чем рекламной уткой. Есть люди (вроде меня), которым дела нет до всяческой новизны, им хватает того, чем они пользовались десятилетиями — и только бы это не отнимали молодые и агрессивные. Это касается и софта, и оборудования — и у меня, допустим, вполне могут оказаться веские основания использовать железо десятилетней давности вместе с новейшим, супернавороченным. По-хорошему, нужен единый банк электронных и программных адаптеров, с гибкой иерархической организацией, обеспечивающей сохранение нашего культурного наследия при всех поворотах будущей технологической моды.

Итого: развитие информатики следовало бы нацелить на создание универсальной интеграционной платформы, позволяющей добиться абсолютной гибкости индивидуальных компьютерных систем. В компьютерном мире все должно сочетаться со всем, чтобы убрать любые препятствия для творчества. Это идеал модульности — но к этому же сводится и самое общее определение сознания. Кто знает? — быть может, подобная универсальность будет означать развитие компьютеров до уровня выше обычного автоматического интеллекта, к чему-то вроде еще одной формы разума?

Java — не только классы

Проповедники Java частенько грешат утверждениями вроде того, что все в Java является классом, и это единообразие — великое достижение и преимущество Java перед иными языками программирования. Всякая реклама, по определению, есть ложь — и в данном случае реклама врет трижды: во-первых, есть и другие языки, способные потягаться с Java по части единообразия (например, в Ruby все подряд обзывают объектом); во-вторых, далеко не все в Java — классы; а в-третьих, сведение всего к классам (или иной произвольно выбранной сущности) вовсе не дает никаких преимуществ.

Каждый язык программирования предоставляет средства доступа к типовому набору компьютерных операций — который, в свою очередь, не более чем надстройка над физическими взаимодействиями в электронных цепях компьютера (или нескольких компьютеров, если речь идет о распределенной системе), из-за которых меняется *логическое* состояние компьютера (или сети) — то, как мы представляем себе компьютерную систему. Пользователя, собственно, интересуют именно эти логические состояния, а не то, как они реализованы схемотехнически или выражены в том или ином языке программирования.

Класс Java — лишь одна из возможных моделей типовых логических состояний, связанных с обычными способами употребления компьютеров; если мы не собираемся работать с большим количеством объектов данного класса, никакой практической пользы от этой конструкции нет — формальные свойства и методы удобнее будет заменить более непосредственными средствами управления. По сути дела, *все* классы являются абстрактными, поскольку любой класс — это лишь *возможность* определения, а завершенная спецификация нуждается в привязке к определенному объекту, к логическому состоянию компьютерной системы. Такая дополняющая спецификация отложена до момента создания экземпляра класса — и часто требует нетривиальных процедур инициализации. Некоторые объектно-ориентированные языки (вроде Python или Ruby) идут еще дальше и позволяют добавлять свойства и методы произвольным объектам, а не их классам (с образованием так называемых

синглетонов). С другой стороны — чистоту классовой идеи портят статические свойства и методы, фактически сводящие класс к его единственному экземпляру, объекту-синглетону. Очевидно, статическая компонента любого класса может выделена в отдельный класс (объект); нет никаких оснований для упаковки таких статических определений внутрь обычного, абстрактного класса — кроме, может быть, некоторого минимального удобства в задании областей видимости по умолчанию. Можно заметить, что большинство классов в пакете `java.lang.*` на самом деле — одна видимость; они существенно статичны, и все, что от них требуется, — это задать условное пространство имен для зарезервированных слов языка, функционально не отличающихся от **`if`**, **`for`** или **`switch`**. использование вперемешку служебных слов, базовых типов и ссылок из статического пространства имен — это все дурная эклектика, ничего общего не имеющая с собственно классами.

Как ни крути, в языках со строгой и статической типизацией, вроде Java, идеологическую чистоту никогда невозможно соблюсти в полной мере — хотя бы потому, что представители класса вовсе не обязательно будут классами. Более того, на практике таких абсолютное большинство, поскольку лишь экземпляры класса `java.lang.Class` и его наследников могут быть названы классами — но как раз эта ветвь корневого класса `java.lang.Object` совершенно бесполезна, ибо не позволяет конструировать и модифицировать классы динамически; разумеется, всегда есть возможность напрямую манипулировать байт-кодом — но такой метод вряд ли окажется сколько-нибудь переносимым. Таким образом, «формальные» классы `java.lang.Class`, `java.lang.reflect.Method` или `java.lang.reflect.Member` — это всего лишь извращенный способ документирования, определяющий основные конструкции Java сугубо метафорически.

Хорошо, допустим, что классы и их экземпляры — это два изначальных примитива языка. Снимаются ли дальнейшие вопросы? Никоим образом. Существует первичная эклектичность, не устранимая ни при каких обстоятельствах. Программы на Java предварительно компилируются в байт-код, который впоследствии может быть выполнен на виртуальной машине Java. Но транслятор Java и JVM — примеры принципиально внеклассовых сущностей, не вписывающихся в общую иерархию, даже если начать с корневого класса, `java.lang.Object`. Различные реализации компилятора Java и JVM влияют на фактическую производительность и привносят разного рода несовместимости, преодолевать которые можно либо объявляя одну из имеющихся реализации стандартом для данной технологии — либо приняв некоторый протокол для указания на тип компилятора и целевой JVM в самом байт-коде, что впоследствии более продвинутые JVM могли адаптировать свое поведение в зависимости от этой информации и добиться таким способом совместимости со всеми предыдущими версиями.

Никто не сомневается, что возможно было бы устранить эклектику в данном частном случае путем определения специальных «формальных» классов для компилятора и JVM, и вполне возможно написать соответствующий код на Java. Однако, как показывает опыт некоторых других языков, это тупиковый путь. В наши дни преобладает мода на модульность и технологическое разнообразие, и большинство современных систем включают различные средства интеграции для объединения модулей, написанных на разных языках. В частности, это означает отказ от древоподобной модели как стандарта языковой организации. Вместо жесткой иерархической структуры — мы переходим к иерархической гибкости, допускающей рекурсивное наследование и неразрешимое взаимоопределение. В таких иерархиях целое может быть развернуто, начиная с любого произвольно выбранного элемента, порождая одну из допустимых иерархических структур; более того, параллельные потоки обработки могут оперировать с разными обращениями той же иерархии. Зародыш такого подхода обнаруживается и в Java в виде изначально заложенного в язык логического круга: объекты понимаются как экземпляры класса, а классы — частный случай объектов.

Принципиальная невозможность свести формальный язык к единственному примитиву (или конечному набору примитивов) связана с существенной неполнотой классической логики, которую выражает, в частности, знаменитый парадокс лжеца. Всякий компьютерный язык требует интерпретации в терминах состояний компьютера и их последовательностей; как

только мы формализуем саму процедуру интерпретации — встаем вопрос об интерпретации этого формализма, и так далее. В рамках классической логики (включая любые формальные расширения вроде модальной, многозначной, комбинаторной или нечеткой логики) нельзя последовательно решить эту проблему. Просто потому, что в основе парадокса лжеца лежит закон исключенного третьего.

Разработка формальных языков с высокой степенью рефлексивности — безусловно полезный опыт не только в прикладном программировании, и но во многих других областях. Программисту нужен язык, наиболее компактно и удобно выражающий все необходимое в пределах конкретной прикладной области. Из двух языков программирования, обеспечивающих одинаковую функциональность, будет выбран то, который делает код максимально компактным и логически прозрачным. По большому счету, объектно-ориентированное программирование появилось в ответ на подобные запросы, обещая структурировать программы интуитивно понятным образом и устраняя за счет этого нежелательные побочные эффекты.

Однако слишком хорошо — тоже нехорошо. Конструирование классов по любому поводу ведет к громоздкому и неэффективному стилю программирования, при котором любое изменение существующих программ требует утомительной и деликатной работы по согласованию функциональности многочисленных классов — вместо банальной замены пары строк кода. Наличие специального инструментария для управления классами и оптимизации структур на практике помогает редко.

Формально, классы полностью эквивалентны библиотекам подпрограмм, с некоторыми ограничениями на способы доступа — которые могут быть реализованы по-разному, начиная с простого соглашения — до модульных пространств имен или встроенных списков контроля. Другими словами, объектно-ориентированное программирование не нуждается в особом синтаксисе — это, скорее, стиль программирования, а не технология. Но то же самое относится и к любым другим синтаксическим конструкциям. Следовательно, язык программирования не сводится к синтаксису и формальной семантике — хотя, разумеется, какая-то определенная форма необходима языку в каждом конкретном приложении. В конечном итоге, все языки программирования являются лишь представлениями *одного и того же универсального языка*, отражающего строение как объекта (логических состояний компьютерных систем), так и потребности пользователя.

Сколь угодно полезный и эффективный подход можно использовать не по делу, вне области его реальной применимости, — и тогда его достоинства превратятся в недостатки. Чрезмерная последовательность — синоним неадекватности. Все знают, как записать любой текст на русском языке, используя алфавит из двух символов,[6] — но будет ли такая запись удобочитаемой? Большинство людей предпочтет обычную латиницу. Более того, на письме большинство людей прибегает к индивидуальной системе сокращений, в дополнение к развернутой записи; это эффективно расширяет алфавит. Подобные составные символы во многом родственны арабским лигатурам — или китайским (японским) иероглифам, которые, как правило, состоят из более простых компонент (хотя и не обязательно фонетических). Очевидно, использование таких сокращений (идеограмм) становится неудобным, когда их слишком много, так что выбор подходящего символа в расширенной кодовой таблице требует больших усилий, чем полная фонетическая запись. Иначе говоря, в каждой прикладной области есть некоторый оптимальный алфавит (или класс эквивалентных кодовых таблиц), обеспечивающий максимальную эффективность. Длина такого алфавита зависит от свойств соответствующих деятельностей. Так, при наличии фиксированного набора операций алфавит стремится включить символы для каждой из них (независимо от колмогоровской сложности); напротив, в быстро развивающихся средах предпочтительней более компактный и универсальный набор символов.

[6] В математике иногда встречаются утверждения, что для кодирования достаточно лишь одного символа; это утверждение ошибочно, ибо в таком представлении неизбежно потребуется особый разделитель кодов — то есть, дополнительный символ.

Точно так же и в программировании не следует стремиться свести все живое к единственной платформе, или единому стандарту разработки. Слепая приверженность к одной излюбленной среде сродни страстной любви к написанию программ непосредственно в машинных кодах, игнорируя языки высокого уровня. В прикладном программировании и компьютерной науке идеология иерархического подхода понемногу набирает вес. Однако до окончания «языковых войн» еще далеко — упорное стремление изобрести единственно правильную технологию (во многом напоминающее бесчисленные попытки создания вечного двигателя — даже сейчас, когда все знают, что подобные устройства невозможны в силу наиболее фундаментальных законов термодинамики) все еще распространено среди компьютерщиков.

Иерархический стиль

Отделение контента от форматирования давно уже стало общим правилом приличия во всех издательских системах, от электронных книг до веб-дизайна. На первый взгляд, вроде бы, все логично: поставщики контента добывают информацию и закачивают ее в систему при помощи подходящих средств автоматизации, не особенно заботясь об особенностях доставки всего этого потребителю; эти данные аккумулируются в некотором резервуаре — и как только накопится достаточно для презентации, профессиональные дизайнеры отрабатывают детали публикации, чтобы устроить дело наилучшим образом и привлечь широчайшую аудиторию.

Как все понимают, такая логика работает лишь в условиях определенной экономической организации — основанной на всеобщем разделении труда. В такой системе профессиональный дизайн зачастую на самом деле выглядит на порядок привлекательнее любого любительского решения. И само собой разумеется, что соображения форматирования не должны касаться, скажем, способа изложения юридических документов или формулировок математических теорем… Однако в реальности не все так просто.

В процессе написания некоторого текста (электронного или нет) автор (подсознательно или намеренно) предполагает, что читать это будут вполне определенным образом. В соответствии с этим своим представлением автор пытается организовать текст так, чтобы подчеркнуть его структуру, упростить навигацию и поиск, привлечь внимание читателя к ключевым положениям. Общеизвестно, что короткие абзацы читаются легче, простая структура предложения способствует ясности, а перемещаться по тексту в несколько колонок значительно труднее. Если при публикации, из самых лучших побуждений, редактор меняет общую разметку — текст, с большой долей вероятности, будет восприниматься по-другому. Например, чем меньше абзацев — тем больше текста помещается в ограниченном объеме; однако объединение абзацев по требованию редактора нарушит внутреннюю логику текста, ибо каждый параграф представляет относительно замкнутую систему мыслей (и в этом плане он совершенно подобен блоку в языках программирования).

Чтобы еще больше усугубить положение, некоторые тексты содержат форматирование, которое существенным образом связано с их содержанием. Так, большинство юридических документов содержит несколько уровней нумерации; если стиль нумерации (сколь угодно уродливый) изменить — документ станет недействительным. Точно так же, внешнее оформление стихотворения, как правило, крайне важно для передачи правильной интонации и переплетения смыслов. Малейшее изменение расположения текста, пунктуации или словоупотребления — убьет оригинал. Даже такое, казалось бы, невинное изменение как замена длинного тире (m-dash) более короткой современной версией (n-dash) — на что многие редакторы идут совершенно не задумываясь, — может привести к нарушению потока восприятия, и в результате — изуродованное впечатление. Да, конечно, всякое форматирование допускает подвижки в определенных границах — но зона неразрушающих модификаций очень индивидуальна, и даже вариации внутри зоны отнюдь не нейтральны — они воспринимаются как различные оттенки той же базовой интонации, отражающие особенности индивидуальной

мотивации.[7]

Понятно, что в современном мультимедийном мире есть объективная потребность в унификации, с тем чтобы один и тот же документ мог быть представлен в любом окружении, с соответствующей подстройкой форматов. Поскольку различные носители не могут быть стопроцентно совместимы, всякий перенос текста в другие условия неизбежно приведет к изменениям формата, совершенно не отвечающим авторскому замыслу. Такой вещи как идеальный перевод — просто не существует. И тем менее, люди говорят на многих языках, и компьютеры обмениваются данными с самыми разными периферийными устройствами — и придется с этим жить, как ни крути. Мы заменяем художественные полотна репродукциями и фотографиями, живой голос — цифровой фонограммой, письмо от руки — вводом с клавиатуры. Такие трансформации вполне подобны литературному переводу — они предполагают иерархию подобия, допускающую различные представления целого. Качество перевода не может, вообще говоря, быть выражено числом — оно зависит не только от умения и вкуса переводчика, но также от поставленной задачи и личной мотивации. Разумеется, всегда можно отличить хороший перевод от плохого — но это не поддается точному измерению. Искусство издателя — это искусство деликатной интерпретации.

Однако пора возвращаться с артистических высот к компьютерным технологиям. Возможно ли вообще дать автору какие-то средства влияния на возможные форматы публикации — чтобы не слишком отвлекать его от собственно содержания? Можем ли мы провести границы между форматированием и контентом достаточно гибким образом, позволяющим учесть специфические потребности конкретного текста?

Современные средства электронной публикации (включая текстовые процессоры, графические редакторы, программы обработки звука и изображения) предоставляют автору широкие возможности форматирования, либо встроенные в базовую конфигурацию — либо подключаемые в качестве сторонних расширений. После сравнительно скромного периода обучения всякий может создавать нечто, прилично выглядящее, — по крайней мере, достаточно, чтобы дать общее представление о желаемом результате и основу для последующих преобразований. Существует также много программ переформатирования, максимально сохраняющих облик оригинала — в пределах возможного. Может показаться, что быстрое технологическое развитие вскоре снимет с повестки дня эту проблему — если еще не устранило ее. Да, есть такая надежда. Но есть и серьезные сомнения.

Типичный пример — средства HTML-разработки. Компании твердят предполагаемым клиентам, что при помощи высокотехнологичного программного обеспечения они смогут построить Web-сайт с нуля, ничего не зная об HTML, CSS, JavaScript и прочих ругательных словах… Но приходилось ли вам когда-нибудь сравнивать Web-страницу, порожденную, скажем, конвертером MS Office (FrontPage), с такой же страницей, вручную размеченной на HTML? Вторая — в несколько раз компактнее, намного более удобочитаема, и у нее меньше проблем по части совместимости с самыми различными браузерами. Я не имею ничего лично против Microsoft — все остальные ничем не примечательней. Есть общая тенденция — убрать как можно больше формата в стилевые листы и скрипт, насыщая текст внешними ссылками вплоть до полной невозможности в чем-то разобраться. Такова, дескать, цена, которую приходится платить за единообразие и мощь.

А если мне вовсе не нужна вся эта заумь на моих Web-страницах? Что если меня вполне устроит простенький вариант форматирования, который наверняка будет выглядеть примерно одинаково во всех браузерах (даже без поддержки стилей), во всех операционных системах, с их неизбежным различием в наборе установленных шрифтов? Далее, мне нужен минимум гибкости, чтобы добавлять отдельные «продвинутые» элементы когда без этого просто не обойтись. Так, я хочу показывать страницу без фреймов тем, кто их не любит, — и оставить фреймы для тех, кого смущает постоянное уплывание блока навигации из визуального поля. Мне нужно уметь вставлять картинки и, возможно, иные объекты без нарушения общей

[7] См. P. Ivanov, “A Hierarchical Theory of Aesthetic Perception: Musical Scales”, *Leonardo*, Vol. 27, No. 5, pp. 417–421 (1994)

структуры текста и без ущерба для его читабельности. Короче, я хочу, чтобы мои документы были просты и приспосабливались к любому окружению без лишнего самоограничения. Или я хочу слишком многого?

Похоже, что так. Обратная совместимость давно не в чести у разработчиков программного обеспечения и компьютерного оборудования. Новые браузеры не понимают старых протоколов — и для совместимости приходится делать несколько страниц HTML, в расчете на разные браузеры — при этом нет гарантии, что очередное «улучшение» протоколов Интернет и браузеров не превратит это старье в нечто совершенно несуразное. В наши дни пользователей заставляют устанавливать все новые и новые версии программ, а для этого приходится обновлять операционную систему, а новая ОС отказывается работать со старым оборудованием… И так далее, без остановки. Даже если я отсканирую свои страницы и выставлю их как картинки (лишая себя всех прелестей гипертекста) — эволюция графических форматов в конце концов погубит и эту затею.

Да, конечно, есть скриптовые библиотеки и интеграционные платформы, которые автоматически решают проблемы совместимости — до некоторого уровня. Разумеется, там много всякого мусора, который мне никогда не потребуется, — однако если предположить, что оптоволокно в каждый дом — дело ближайшей перспективы, заботиться о компактности, вроде бы, уже и не нужно. Все равно для современных Web-страниц, набитых под завязку графикой, звуком и видео, лишний мегабайт чистого текста погоды не делает. С другой стороны, разработчики сайтов нынче помешались на динамическом порождении контента — и старая добрая статика в наше время воспринимается как дурной тон, нечто неуместное в приличном обществе. Даже тупо текстовые страницы, на которых по определению никогда ничего не изменится, следует компоновать на лету из мелких кусочков — исключительно ради стилистического единообразия. В современном HTML почти не осталось собственно HTML, это сплошные скрипты. Конечно, дизайн становится громоздким и неуклюжим, съедает все имеющиеся ресурсы и целиком забивает полосу пропускания, приводит к периодическим подвисаниям из-за конфликтов с другими активными процессами. Но, конечно же, все это временное явление, и ситуация должна улучшиться с приходом технологий нового поколения, к которым лучше готовиться загодя… Но потом, когда долгожданные новые технологии все-таки приходят, оказывается, что они уже морально устарели — и современный уровень сложности для них становится высоковат. Все это напоминает бесконечную российскую стройку: мы постоянно что-то строим — но почему-то никак не видно ничего построенного, чем можно было бы просто пользоваться.

До сих пор лично я не встречал ни одной системы подготовки текстов (или Web-разработки), которая была бы способна автоматизировать сколько-нибудь удовлетворительным образом даже самые обычные презентации, не говоря уже об особых выразительных возможностях. Почему-то мои нужды никак не хотят мирно сосуществовать со стандартными реализациями. Разумеется, мир не обязан прислушиваться к какому-то полоумному старикашке — но пока он не стер меня с поверхности земли, я имею полное право разговаривать хотя бы с самим собой.

В идеале, автор мог бы ограничиться лишь общими указаниями на то, какие элементы форматирования для него существенны, — а все остальное возьмет на себя автоматика публикации. Такая система подготовки текстов включала бы набор всевозможных фильтров, позволяющих динамически обеспечивать качественное представление текста на любом носителе. Внутритекстовая разметка (да-да, я обожаю все эти старинные `<i>` или `<b>` и считаю, что обязательная замена их стилями стала бы катастрофой!) и введение индивидуальных стилей — это явное указание на важность соответствующего форматирования. Например, если я хочу, чтобы кусок текста был изображен жирным шрифтом или курсивом, — так оно и должно быть на печати, и мне дела нет до каких угодно экспертов, полагающих, что такой способ выделения уже не в моде и должен быть выведен из употребления. Почему я должен соглашаться с общепринятым идиотским мнением, что таблицы HTML не столь удобны для форматирования, как хаос явно и неявно привязанных к стилям секций `<div>`, про которые практически невозможно заранее сказать, во что это превратится на экране? Если я использую

пробелы для создания отступов — это мое право, и стандартная ширина пробела должна сохранять мой замысел в неприкосновенности. Разумеется, мне потребуются разные виды пробелов для разных нужд — наряду с разными дефисами и тире; таким способом я смогу из самого текста управлять разбивкой на строки и выравниванием. И конечно же, нужен простой и удобный способ размещения в тексте графики (а здесь пока хромает любой HTML, со стилями или без). При необходимости я должен иметь возможность определять собственные стили — и мой выбор любая система презентации обязана уважать (включая встроенные шрифты или даже индивидуальные обработчики стилей).

Это не то же самое, что отделение форматирования от контента. Проблема в том, чтобы приспособить уровень разделения к потребностям каждого конкретного пользователя. Вообще говоря, элементы форматирования предполагают иерархическую организацию в рамках отдельного документа, от самых общих и безусловно обязательных — до вспомогательных тонких настроек, применимых не всегда. Интерпретатор текста (программа-конвертор или живой дизайнер) будет пытаться реализовать предложенный формат максимально полным образом, переходя к более грубому уровню лишь при невозможности удовлетворения запросов более низкого уровня. Таким способом автор получает инструмент динамического управления трансляциями, при безусловном сохранении содержательной стороны.

Так или иначе, идея иерархического форматирования скрыто присутствует в большинстве текстовых процессоров. Например, похожий принцип был положен в основу TeX; однако существующие реализации как-то упустили из виду эту сторону дизайнерской работы, и в результате, как и в случае с HTML, приходится бороться с несовместимостью разных версий, корпоративных решений и типовых шаблонов. Любые попытки выработать универсальную модель документа обречены на провал, пока мы не откажемся от привычного древесного представления и не осознаем потребности в динамическом (ориентированном на контент) структурировании. В какой-то мере это напоминает использование статистических и структурных особенностей файла алгоритмами сжатия ради достижения максимальной плотности упаковки. Точно так же, модель документа в норме должна возникать из его содержания, а не из общих соображений, основанных на чьем-то ограниченном опыте и личных предпочтениях, а потому не отвечающих нуждам других категорий пользователей. Гибкость достигается путем смещения акцентов с единичных правил на общие принципы, позволяющие компоновку любых комбинаций правил по требованию. Такая обращаемость иерархий[8] — необходимое условие для создания систем эффективной подготовки текстов.

Объектно-ориентированные vs реляционные

Хотя реляционные базы данных долгие годы остаются абсолютно преобладающей технологией, поиски альтернативных решения не прекращались никогда, и сегодня они становятся особенно актуальны. Но до сих пор многие даже не знают о каких-либо альтернативах — а те, кто о них наслышан, считают это абстрактным баловством, экспериментальными игрушками без особой рыночной перспективы. Lotus Notes/Domino пока остается единственным примером полномасштабной коммерческой нереляционной системы управления данными. Впрочем, после поглощения компании Lotus гигантом IBM, в этом направлении нет существенного технологического прогресса; насильственная адаптация Notes/Domino к собственным технологиям IBM и типовым Web-платформам уже привела к отходу от изначальной идеи — и ничего светлого от будущего ждать не приходится. По всей видимости, собственные структуры данных Lotus вскоре уступят место реляционным форматам DB2. Можно еще вспомнить, как к начале 1990-х, базы данных Jasmine пытались конкурировать с продуктами на основе SQL, и поначалу даже неплохо получалось. К сожалению, авторы вскоре забросили этот проект и перепрофилировали его под еще одну

[8] P. Ivanov, *Philosophy of Consciousness*, Trafford (2009)

платформу интеграции, так что собственно базы данных остались не у дел. На том все и кончилось.

Но надежда все же есть. Реляционный подход обязан своими успехами традиционному представлению базе данных как способу складирования принципиально статичных сущностей. Такое хранилище преимущественно используется для того, чтобы вытаскивать из него данные и скармливать их какой-нибудь интерактивной оболочке, которая занимается обслуживанием пользовательских запросов. На заре компьютерной эры большинство пользователей знало и воспринимало одну-единственную форму отчета — таблицу (возможно, с довесками в виде типовых диаграмм). Совершенно логично, что базовые структуры выбирались из соображений этой повсеместной необходимости и оптимизировались прежде всего для генерации таблиц. Вся дальнейшая обработка происходила вне СУБД и не влияла на ее функциональность. Потом Ray Ozzie придумал Lotus Notes — первая брешь в реляционной стене. Это уже не просто база данных — а интегрированная система групповой работы, предназначенная прежде всего для обслуживания документооборота, а не только для хранения данных. Упор на совместную работу, параллельная обработка данных, индивидуализация и безопасность коммуникаций требуют принципиально иной парадигмы, устраняющей застарелую слабость реляционных баз данных — недостаточную поддержку модификации данных одновременно в нескольких согласованных потоках. Фокус таким образом смещается от *базы данных* к *приложению*, способному комбинировать разношерстные данные прозрачным для пользователя образом.

В наши дни практически все коммерческие СУБД оснащены средствами совместной работы над проектами и автоматизации документооборота, и это очевидно размывает строго реляционную парадигму. Однако еще потребуется время, чтобы осознать необходимость альтернативных способов хранения и получения данных. По мере того как обычные отчеты-таблицы будут уступать место мультимедийными презентациями, позволяющими развертывать различный структуры в зависимости от общего контекста и потребностей пользователя табличный формат хранения данных начнет проявлять свою ограниченность. Общая тенденция развития Web-приложений подталкивает развитие в этом направлении, поскольку базы данных сейчас преимущественно используются в качестве основы для создания интерактивных сетевых систем, и реляционная модель уже не соответствует нуждам технологического прогресса.

Но давайте пока забудем о промышленном применении и поговорим о принципиальных моментах. Сразу же возникает вопрос: а есть вообще в объектной технологии что-то такое, что существенно отличало бы ее от реляционного подхода?

В самом общем смысле, объектно-ориентированная база данных (ОБД) предназначена для хранения объектов, а не данных — то есть, записи в такой базе могут содержать наряду с полями данных еще и код, позволяющий модифицировать способы работы с данными в зависимости от окружения. Поскольку главный принцип объектно-ориентированного программирования формулируется практически так же, может показаться, что ОБД должны, вообще говоря, лучше интегрироваться с большинством языков программирования, в которых так или иначе реализована идеология ООП. В этом смысле реляционные базы данных (РБД) ближе к традиционно-структурному стилю программирования, когда код жестко отделен от данных.

Однако на деле не все так оптимистично. Многие объектно-ориентированные языки предполагают статическую типизацию; это означает, что код (методы классы, реализованные как библиотека динамически подключаемых подпрограмм) всегда эффективно отделен от данных объекта (инициализированного экземпляра класса), — а значит, между структурным и объектно-ориентированным стилями программирования нет принципиальной разницы. С другой стороны, существуют объектно-ориентированные интерфейсы к РБД, и каких-либо дополнительных средств интеграции просто не нужно.

Аналогично, можно указать, что на практике чисто объектной системы хранения не бывает — поскольку реально запись создается как некоторая простая структура (набор полей), а вся логика обработки переносится в формы представления (способ доступа к данным), и хранится отдельно от контента. А значит, ОБД всегда можно спроецировать на РБД,

оснащенную подходящими обработчиками событий; так и поступают в реализациях Lotus Notes/Domino, привязанных к IBM DB2.

Впрочем, такой вещи как чисто реляционная база данных тоже в природе не существует. Ни одна из реальных СУБД не является реляционной в строгом смысле слова — они всегда включают множество нереляционных черт, и именно эта непоследовательность делает их практически полезными и конкурентоспособными. Учитывая также, что представление РБД на диске уже далеко ушло от плоских таблиц и оптимизировано для экономии места и ускорения доступа, вся связь с реляционными технологиями сводится к языку построения запросов — одному из диалектов SQL. То есть, у нас есть не РБД — а всего лишь базы данных на основе SQL.

Исходя из всего вышесказанного, возможность роста доли ОБД в технологических решениях для распределенных хранилищ не кажется такой уж призрачной — хотя, скорее всего, речь будет идти о чем-то третьем, отличным как от РБД, так и от ОБД, и сочетающим их сильные стороны. И все же, в пределах сегодняшнего опыта, есть смысл приглядеться к некоторым ходячим предрассудкам.

Предрассудок 1. *Структуры данных ОБД совершенно не похожи на таблицы*.

На самом деле это не так. Таблицы — такой же объект, как и все остальные, с определенными свойствами и методами, — и любая комбинация таблиц может быть выражена в объектно-ориентированном стиле (динамически) не хуже, чем при использовании SQL (преимущественно статический запрос). Напротив, любую совокупность объектов можно трактовать как таблицу, строки которой соответствуют индивидуальным объектам, а столбцы содержат значения свойств. Например, такая возможность реализована в приложениях Lotus Notes, позволяющих формировать обычные SQL-запросы, основанные на конкретных видах (views) и формах (forms). Если к тому же положить, что методы класса реализованы как поля данных (ссылки на код), отличий от реляционной структуры вообще не остается.

Предрассудок 2. *ОБД слишком неудобны для гибкого формирования отчетов.*

Это мнение основано на опыте имеющих ОБД — таких как Lotus Notes или Jasmine; однако, отсутствие эффективной реализации вовсе не означает принципиальной невозможности. Разумеется, если нам ничего не нужно, кроме таблиц, придется оснастить ОБД специальными средствами быстрой выборки данных по заданному разрезу — точно так же, как к РБД приходится добавлять средства поддержки документооборота. И все же, с развитием технологий data mining, управления знаниями, многофакторного анализа и семантического моделирования, преимущества объектно-ориентированного подхода становятся очевидными. Вместо пристального разглядывания необъятных таблиц в надежде наткнуться на какие-то закономерности — пользователь сможет видеть лишь компактную общую картину, и развертывать ее по мере необходимости, иерархически, начиная с любого места — и это намного удобнее для гибкой отчетности, чем жесткие и неповоротливые таблицы.

В реальной жизни РБД вовсе не так удобны для пользователя, как пишут в рекламных проспектах. Чтобы построить правильный SQL-запрос для получения удобно организованной нетривиальной выборки из нескольких таблиц — требуется квалифицированный программист. Это вполне подобно тому, как в клиенте Lotus Notes всякий пользователь может легко строить персонализированные виды (views) для (динамического) формирования простых табличных отчетов — но построить вид для более сложных запросов может лишь опытный программист. Хорошо спроектированная ОБД гораздо понятнее, поскольку пользователь имеет дело с естественными свойствами объектов, а не абстрактными полями данных. Но если вдруг пользователю понадобится какая-то нестандартная структура (эффективно связанная с объектами другого типа), переход к ней от уже существующих форматов может оказаться проблематичным. Считается, что в предположении многочисленных нестандартных запросов следует стремиться к нормализации дизайна — но как раз высокая степень нормализации и делает РБД столь недружественными по отношению к пользователю, который начисто теряется в нагромождении взаимосвязанных таблиц, когда никому толком неизвестно, где что лежит. Можно пока только мечтать о настоящей иерархической базе данных, позволяющей

динамически формировать структуры — с автоматической подстройкой дисковых форматов по мере необходимости.

Предрассудок 3. *ОБД приводят к большим размерам файлов.*

В некоторых старых реализациях это действительно так. Общеизвестно, что реляционные системы давно уже ушли от хранения данных в виде плоских таблиц (вроде первых версий формата DBASE, которые приводили к размерам файлов намного превышающим типичный размер файла .nsf в Lotus Notes. Мощности современных компьютеров позволяют на лету сжимать и распаковывать данные — что было совершенно невозможно в старые времена, когда базы данных только начинали вводиться в обиход. Очевидно, размер файлов зависит от сложности данных. Так, статическое поле требует ровно столько места, сколько нужно для хранения соответствующего значения; напротив, элемент документооборота предполагает наличие дополнительной информации о времени модификации, версии, шифровании, цифровой подписи и т. д. В идеале ОБД должна предоставлять полный контроль над тем, что будет храниться, тем самым допуская различные стратегии экономии.

Например, как в ОБД, так и в РБД, есть накладные расходы на поддержание многочисленных перекрестных ссылок. С ростом сложности приложения объем хранимых указателей может превысить объем собственно значимых данных. Чтобы преодолеть эту трудность, можно было бы хранить ссылочную структуру как отдельную запись (объект), динамически порождая любую ссылочную структуру над тем же самым объемом данных. Еще большей компактности можно достичь за счет динамического создания локальных классификаторов типовых классов и часто повторяющихся значений полей. Таким образом, гигантские файлы возникают, скорее, от недостаточного внедрения объектных технологий, чем от их использования.

Предрассудок 4. *ОБД слишком сложны, это затрудняет эффективное управление.*

Это как посмотреть. Действительно, РБД используют сравнительно небольшой набор базовых операций — тогда как ОБД требуют особых «драйверов» для каждого типа объектов. Но современные программные продукты в достаточной мере расширяемы — в них предусмотрено добавление внешней функциональности. Следовательно, достаточно реализовать стандартный протокол расширения в ядре ОБД (наряду с минимумом общеупотребительных классов), чтобы любые индивидуализированные решения можно было подключать как внешние компоненты и отключать после использования. В конечном итоге ОБД окажутся существенно быстрее РБД, поскольку время обработки в ОБД меньше зависит от сложности запроса.

ОБД гораздо удобнее и с точки зрения возможных изменений организации данных. В SQL-системах всякая реорганизация таблиц — это верная головная боль, ибо приходится модифицировать все формулы запросов, а пользователи (или прикладные программисты) должны привыкать к необычному размещению данных в новой инфраструктуре. Иногда тривиальнейшая операция (вроде изменения длины ключа) приводит к совершенно непреодолимым трудностям. В иерархической базе данных любое изменение прозрачно для пользователя, поскольку данные отделены от ссылок, и одна и та же видимая структура может соответствовать разным внутренним форматам.

Предрассудок 5. *ОБД хуже масштабируемы по сравнению РБД.*

Часто приходится слышать, что ОБД несовместимы с высокой интенсивностью запросов и большим количеством пользователей. Это поверхностное мнение происходит из сравнения несопоставимых сторон ОБД и РБД. В пределах одной и той же функциональности — никаких различий не будет. Так, большинство документооборотных задач на базе РБД выполняется внешними приложениями, которые запрашивают необходимые им данные в РБД. Чтобы сравнить эффективность таких систем с ОБД, надо включать в сравнение и эти внешние приложения. Если же использовать ОБД исключительно как хранилища данных (то есть, не предполагая никакой внутренней обработки), они окажутся ничуть не хуже масштабируемыми, чем РБД. В обоих случаях есть проблемы с доступом к записям большого размера (например,

мультимедийным); такие записи, как правило, лучше хранить как отдельно от базы данных, в файловой системе.

Отделение данных от способов их обработки может где-то оказаться полезным — тогда как в других случаях это лишь усложняет дело. В хорошо утроенной (иерархической) базе данных уровень разделения должен быть подвижным, с возможностью адаптации к конкретному приложению и операционной среде.

Предрассудок 6. *ОБД когда еще появятся! — а РБД уже здесь и сейчас.*

Это серьезный аргумент. Гонка за быстрыми деньгами отвлекает производителей программного обеспечения от инновационных проектов. Однако объективная необходимость заставляет рынок предпочесть больше модульности, единообразия и гибкости — хотя эти требования пока возможно удовлетворить путем расширения традиционных SQL-систем, что делает их все менее похожими на РБД. Продолжаются эксперименты в области технологий работы с данными и знаниями, документооборота и методов совместной работы. Вполне возможно, что прорыв уже совсем рядом — и кто-нибудь из ведущих игроков выбросит на рынок качественно новый продукт, пытаясь захватить перспективную нишу.

Так jump или не jump?

Аргументация по поводу допустимости той или иной языковой конструкции в программировании, равно как и рассуждения о «хорошем» и «плохом» стиле, — сродни попыткам установить единые стандарты в естественных языках. Если мы исключим матерную лексику из словарей — люди не перестанут ее употреблять. Точно так же, треклятое словечко `goto` все еще занимает важное место в программировании на языках высокого уровня, насмотря на чье-то большое желание его истребить.

А если серьезно — существуют ли против этого действительно серьезные возражения, и есть хоть какие-то основания объявлять это «дурным стилем»? Конечно, легко написать совершенно идиотскую программу, пестрящую метками, расставленными «от фонаря». Но в любом мало-мальски нетривиальном случае можно точно так же, при желании, написать абсолютно невразумительный — но идеально модульный код, без единого перехода по метке. Принято ругать конструкцию `goto` за то, что она предположительно позволяет обойти нормальные процедуры закрытия блока; но это, скорее, вопрос некорректной компиляции, а не дефект самого кода. Адепты структурного программирования утверждают, что переходы по метке делают структуру программы недостаточно прозрачной, что при этом теряется явное отделение одной логически замкнутой структурной единицы от другой; однако большинство таких высказываний предполагает, что мы должны ограничиться лишь некоторым узким классом простейших структур (и стать чем-то вроде программистов-веганов). Можно также услышать высокопарные заявления о том, что использование `goto` несовместимо с крупномасштабным индустриальным программированием — что звучит, по меньшей мере, странно, поскольку условные и безусловные переходы используются в промышленных приложениях испокон веком, и это никогда особо не мешало работе. С другой стороны, почему мы должны ориентироваться только на массовую индустрию? Искусство программирования — не менее важный компонент человеческой культуры.

В конце концов, на самом низком уровне, переходы по адресу в памяти встроены в железо, они есть в системе команд любого процессора — а без возможности переключиться на другую ветвь вычислений компьютер был бы практически бесполезен. Теоретически, конечно, можно построить изначально модульную низкоуровневую архитектуру, ориентированную на изолированные сегменты кода, с переключением по событиям — но это стало бы лишь более громоздким вариантом той же базовой функциональности. На самом деле, всякое событие предполагает передачу управления соответствующему обработчику — и совершенно неважно, встроен этот механизм в процессор или записан последовательностью команд в памяти. Ветвление универсально. Независимо от оборудования, оно выражает глубинную логику

последовательной обработки — а всякое вычисление вообще развертывается во времени и потому предполагает последовательности операций. Устранять время из программирования — все равно что устранять еду из питания. Напротив, структурное программирование тесно привязано к характеру приложений, оно зависит от уровня развития технологий; зашивать эту логику в процессоры было бы неразумно — если мы не собираемся выбрасывать старое оборудование каждый раз, когда меняются стилевые предпочтения в программировании.

Конечно, существуют существенно статические задачи, которым последовательное развертывание вычислений совсем не требуется. Например, вовсе незачем решать систему дифференциальных уравнений последовательно, шаг за шагом, точка за точкой; во многих случаях параллельная обработка привлекательнее, а еще лучше — системное решение, позволяющее сразу получить результат для любого заданного значения аргумента (как в аналоговых компьютерах). Точно так же, строго алгоритмизованный код и заранее определенная локализация функций совершенно ни к чему в адаптивных компьютерах, способных вырабатывать подходящие структуры по мере необходимости. Однако если речь идет о численном моделировании или автоматизации рабочего процесса, без последовательной обработки не обойтись.

При структурном подходе, программа представляется последовательностью блоков ***A***, ***B***, … , — причем каждый блок логически замкнут: он запускается некоторым событием и выполняется как целое, как одна простая операция. То есть, все возможные способы организации выполнения эффективно сводятся к простому условному выражению:

```
if e1 A;
else if e2 B;
...
```

Разумеется, в реальной жизни эта конструкция существует в разных вариантах. Так, можно задать особые обработчики, зарегистрировать их в системе и настроить прослушивание событий e_1, e_2, … Это позволяет перейти от последовательной обработки к параллельной — что для независимых событий означает также изменений способа переключения. Но структура в целом та же. И в «плохо структурированном» коде можно было бы встретить нечто вроде

```
on e1 goto branch_e1;
on e2 goto branch_e2;
...
goto Exit;

branch_e1:
  A;
  goto Exit;
branch_e2:
  B;
  goto Exit;
...

Exit:
```

Какой уродливой ни показалась бы такая запись ригористу, структурная логика всегда может быть в точности воспроизведена с использованием **goto** (хотя бы потому, что на уровне оборудования она реализована *именно так*); более того, ничто не мешает нам подчеркнуть, при необходимости, общую структуру разного рода условными обозначениями и форматированием кода (отступы, именование меток, расположение фрагментов). Хороший компилятор должен уметь распараллеливать такую запись, по возможности автоматически определяя обработчики соответствующих событий и генерируя код очистки, выполняемый перед ветвлением. Для не слишком умных трансляторов программист может руками прописывать все необходимое (что иногда даже лучше). В любом случае, программный контроль ветвления намного превосходит структурный метод в гибкости, когда речь заходит о структурированных событиях. Например, если событие e_1 предполагает событие e_k, но не наоборот, можно просто написать:

```
on e1 goto branch_e1;
on e2 goto branch_e2;
...
on ek goto branch_ek;
...
goto Exit;

branch_e1:
  A;
  goto branch_ek;
branch_e2:
  B;
  goto Exit;
...
branch_ek:
  K;
  goto Exit;
...

Exit:
```

В «структурном» стиле придется дублировать код, генерировать дополнительные события, организовывать структурированную очередь событий — или еще что-нибудь столь же неуклюжее. По нынешней моде, можно было бы «обернуть» исходные блоки в функции и заменить прямую передачу управления блоку ***A*** блоком вида { **call** f_A; } — это эффективно порождает иерархию вложенных блоков (или рекурсивные функции). Такая технология позволяет комбинировать блоки с минимальными накладными расходами: { **call** f_A; **call** f_K; }; однако не похоже ли это больше на замаскированную комбинацию ветвлений? По сравнению с подобными уродливыми монстрами, простое **goto** смотрится куда изящнее и делает логику переходов совершенно прозрачной, так что изменить при случае структуру событий не представляет труда.

Для удобочитаемости в языках программирования есть разного рода сокращения для типовых структур ветвления — например, два ходовых варианта конструкции **switch**. При использовании явно программируемых ветвлений несложно соблюсти функциональную идеологию — достаточно, чтобы программист (или компилятор) аккуратно прописывал подготовку контекста каждый раз, когда требуется пересечение границы блока, чтобы ограничить видимость имен и доступ к внутренним переменным. Явное задание логики инициализации или очистки предоставляет программисту широчайшие возможности и может существенно повысить эффективность кода. Кстати, хитроумные методы “обертывания”, используемые в объектно-ориентированном и функциональном программировании для создания объектов, сохраняющих свое состояние (функции, итераторы и т. д.) — это всего лишь громоздкий способ задания процедур обработки для пересечения границ объекта (фрагмента кода).

Основная беда с явным программированием ветвлений — это перфекционизм. Попытка оптимизировать код сверх разумных потребностей (особенно в части использования общих фрагментов) иногда приводит к подлинным произведениям искусства — уникальным и непрактичным. Свобода тесно связана с ответственностью — недостаток одного уничтожает другое. Если язык программирования разрешает явные ветвления, можно войти в блок, что-то сделать и покинуть его без всякой заботы о последствиях — рискуя нарваться на неинициализированные объекты, утечку памяти и т. п. Но, в конце концов, те же проблемы есть и в других языках, тщательно избавленных от явных ветвлений — вспомним, хотя бы, об утечке памяти как типичной болезни Java. С другой стороны, не странно ли отнимать у программистов мощный и полезный инструмент только потому, что некоторые из них не умеют с ним обращаться?

По поводу допустимости многих точек входа в блок и возможности покинуть его в любой момент идут дебаты в компьютерной литературе. Так, в параллельном программировании каждый блок можно считать самостоятельным процессом — и тогда нет ничего странного в

том, что другой процесс взаимодействует лишь с отдельными его компонентами. По большому счету, именно так взаимодействуют реальные вещи. С другой стороны, если считать блок экземпляром некоторого класса, мы требуем обязательной инициализации и очистки; но транслятор всегда может интерпретировать множественные точки входа как одну-единственную, но с параметрами, — а выход из блока всегда предполагает передачу управления обработчику выхода.

Будущее по знаком Java?

Для разработчика выбор языка программирования не имеет особого значения — все они устроены примерно одинаково, и то, что написано на одном языке, всегда может быть переписано на другом. Здесь все определяют требования заказчика и случайные привычки, а вовсе не мифические преимущества одного над другим. Например, когда-то страницы в Сети были сплошь по-английски — просто потому, что Интернет родился в Америке и тащил на себе груз технологических традиций; по мере широкого распространения Уникода локализованный контент быстро вытеснил английский язык из национальных доменов. А сегодня средства автоматического перевода (при всем их несовершенстве) позволяют бродить по сайтам вообще без знания языка; впрочем, без особого напряга можно построить и по-настоящему многоязычный сайт.

Java изобрели в Sun Microsystems в качестве корпоративного языка — главным образом, с целью получить некоторое рыночное преимущество в эпоху, когда Microsoft успешно завоевывала мир довольно простыми и удобными творениями. Беспрецедентная рекламная кампания, развернутая Sun по поводу Java, была подхвачена всеми приверженцами Unix (которые тоже почуяли серьезную опасность) — и принесла свои плоды: на некоторое время Java стал стандартом *de facto* для коммерческих приложений и особенно Web-сервисов. Позже новый владелец, Oracle, попытался возобновить агрессивное продвижение Java — но на этот раз без особого успеха, поскольку к тому времени появилось много альтернативных систем разработки, в том числе бесплатных. Тем не менее, популярность Java, а потом и JavaScript (еще одно мощное конкурентное оружие Sun), практически уничтожили шансы Microsoft на завоевание Web (а значит, вообще всей информационной отрасли).

В 1990-х Java с помпой преподносили как универсальный язык будущего, который призван заменить все остальные языки — а новые процессоры предлагалась затачивать под непосредственную интерпретацию байт-кода Java, понемногу избавляясь от традиционных наборов инструкций. По счастью, эта идея оказалась для индустриальных целей совершенно неподходящей — большинство производителей оборудования были против, и в конце концов будущее под знаком Java превратилось в отдаленную память о кошмаре прошлого.

Если серьезно, ничего радостного в этом языке нет. Ему не хватает элегантности и краткости; он требует трудоемких поиском обходных путей ради преодоления его врожденной неуклюжести; он почти не допускает расширения, заимствования выразительных средств у других языков; его практически невозможно адаптировать под индивидуальные нужды. Впрочем, когда все богатство мира пытаются свести к одной-единственной парадигме (класс) — можно ли ожидать чего-то еще?

Пропагандисты Java всячески раздувают миф о переносимости. Да, существуют реализации Java для любых платформ — однако в реальности Java не более переносим, чем Fortran, Basic или C++. Появилось множество плохо совместимых между собой версий Java, потом широкий спектр JVM, по-разному интерпретирующих один и тот же код. Исходная идея Java была далека от равноправия всех операционных сред — язык предназначался для подавления конкурентов, как попытка навязать им чуждую для них идеологию и не дать развиваться в других направлениях. Пожалуйста — покупайте только системы, авторизованные для работы с Java, — и у вас не будет проблем с совместимостью. Длинная серия судебных исков против Microsoft и некоторых других компаний, позволяющих себе отклониться от «истинного» духа Java (а также «подлинного» JavaScript, «стандартного» HTML и т. д.)

наглядно иллюстрируют глубинный замысел строителей светлого будущего под знаком Java.

В наши дни идея промежуточного уровня между машинным кодом и высокоуровневым языком программирования стала практически общим местом. Большинство современных языков предполагают преобразование в некий «байт-код» — который потом уже можно компилировать или интерпретировать. Переносимость приложений и служб напрямую увязывается со стандартом этого внутреннего представления. В начале 1960-х этот подход основательно отработан в СССР. Был предложен унифицированный низкоуровневый синтаксис программирования для ряда широко распространенных в то время советских компьютеров — а прикладные программисты получили гамму специализированных языков высокого уровня, оптимизированных под определенный круг задач. Эти языки обозначались просто буквами греческого алфавита[9]; самые известные из них — Альфа и Эпсилон. Гораздо позже та же идея легла в основу платформы .NET от Microsoft, где разные языки программирования (C#, Visual Basic, F#) транслируются в общий промежуточный язык и могут пользоваться одними и теми же библиотеками подпрограмм.

Такой вещи как полная переносимость в природе не бывает. Качественный код можно написать на любом языке — но никакая платформа не может гарантировать практическую поддержку всех его достоинств. В идеале каждый разработчик должен иметь возможность собрать свой индивидуальный набор инструментов, сочетая разнородные элементы любых языков, платформ и операционных сред. Все это могло бы интегрироваться с иерархией интерфейсов, позволяющих обеспечить совместную работу и обмен данными. Объективной основой совместимости и переносимости может стать только единая организация практических задач, общие принципы компьютерной архитектуры, проблемно-ориентированные протоколы коммуникации и типовые форматы данных.

Насущная потребность в гибких и настраиваемых под конкретного разработчика средствах программирования привела к появлению разного рода расширений (препроцессоров, встроенных скриптов и т. д.). Быстрый прогресс в области автоматизированной разработки компиляторов, возможно, скоро вернет нас в старинные времена VAX VMS, когда допускалась свободное языкотворчество, гибкая настройка языковых средств под себя. Иерархическая организация языков программирования и скриптов, включая идеологию промежуточного кода, призвана значительно облегчить эту задачу. Что ж, в конце концов, Java — это шаг в правильном направлении, и только дикая коммерция подпортила перспективы разумного совершенствования. Но в каком-то смысле, в Java тоже брезжит кусочек будущего — в единстве с другими созидателями, а не вопреки им.

Парадигмы компьютерных систем

В истории компьютерных технологий можно заметить чередование двух противоположных тенденций: коллективность — индивидуализация. Первые электронные вычислители были колоссальными инструментальными комплексами, требующими целой армии обслуживающего персонала, — однако в целом оператор был полновластным хозяином, эдаким всемогущим божеством, вручную программирующим регистры и контролирующим все технические мелочи. Прошли годы — и компьютеры стали корпоративными инструментами, предназначенными для обработки самых разных запросов; роль оператора при этом стала чисто посреднической: все что от него требовалось — это скормить компьютеру колоды перфокарт и перфоленты, смонтировать магнитные ленты и флоппи-диски (тогда они были по-настоящему гибкими!) в соответствии с прилагаемыми инструкциями, а в конце нарезать тонны распечаток и распихать их по личным и корпоративным ячейкам. Как только начала развиваться многозадачность — пользователи снова получили контроль над выделенными под конкретную задачу ресурсами, так что большой компьютер эффективно становился как бы разрезан на множество маленьких компьютеров для индивидуального пользования. Эта тенденция

[9] По той же схеме получил имя язык C, а потом и его наследник D.

получила материальное воплощение в программируемых калькуляторах и персональных компьютерах. Однако эра сетевых решений выдвинула на первый план парадигму «клиент — сервер», с одним центральным компьютером (сервером) коллективно используемым конечными пользователями (клиентами), разделяющими серверные ресурсы в соответствии с заданными квотами. С тех пор эта парадигма оставалась неизменно популярной — а центр противостояния индивидуальной и коллективной работы сместился в область выбора между «тонкими» и «толстыми» клиентами. Различие функций сервера и рабочей станции (или мобильного устройства) привело к специализации оборудования и программного обеспечения; в течение некоторого времени серверные операционные системы значительно отличались от клиентских (или, по крайней мере, особым образом конфигурировались). Коммерческие приложения также разделились на серверную и клиентскую компоненты — особенно в задачах документооборота и средах совместной работы.

Однако время не стоит на месте — персональные устройства всех сортов становятся более мощными и универсальными, и, может быть, широкое распространение технологий peer-to-peer приведет к реабилитации «вычислительного индивидуализма» — а сервера будут лишь обеспечивать связность сети. Когда данные и код распределены между миллионами персональных компьютеров, избыточность достигает невиданных доселе масштабов — господство централизованных хранилищ, суперкомпьютеров и гигантских кластеров подходит к концу. В каком-то смысле, каждый компьютер становится одновременно и сервером, и рабочей станцией, предоставляя то одну, то другую функцию по требованию. Такая распределенная система гораздо устойчивее нескольких выделенных серверов (с любой степенью кластеризации), поскольку устранение одного узла не влияет сколько-нибудь существенно на работу сети в целом. Очевидно, это все та же идеология Интернета — только на уровне приложений.

Конечно, в реальности мы всегда увидим какую-то комбинацию элементов обеих парадигм; мощные сервера еще долго будут вполне соответствовать запросам корпоративного сектора. И все же современные компьютеры-монстры будут понемногу вытесняться многочисленными мелкими узлами — как универсальными, так и специализированными (например, встроенными в бытовые электроприборы). Распределенная система позволяет достичь большей гибкости и стабильности. А если потребуется дополнительная функциональность — можно просто запросить ее в сети, чтобы не менять конфигурацию локальных устройств. В качестве намека на то, что тут может получиться, можно упомянуть службы онлайнового перевода и конвертации файлов. Полностью распределенная операционная среда, конечно же, потребует значительной унификации оборудования и протоколов обмена данными. В конце концов, все компьютерные устройства смогут говорить на одном языке — и станут членами единого компьютерного сообщества.

Промежуточное положение между полностью распределенной и централизованной системами занимают облачные технологии. Их можно считать своего рода промежуточным этапом на пути к децентрализованной сети, синтезом «индивидуализма» и «коллективности». В облаке преимущества распределенных вычислений сочетаются с мощью выделенных центров переработки и хранения данных — и это делает облака довольно привлекательными. И все же, пытаясь угодить и вашим, и нашим, мы никогда полностью не удовлетворим ни тех, ни других. Корпоративным пользователям не понравятся ограниченность ресурсов и потенциальная небезопасность; индивидуалам придется не по вкусу слишком узкая функциональность, отсутствие достаточного количества персональных настроек, зарегулированность сверху — и неизбежные дополнительные расходы. В силу этого облачные системы вряд ли когда-либо станут чем-то большим, нежели вспомогательное инструментальное средство.

Статические = динамизм

Объектно-ориентированное программирование давно утратило прежний задор и превратилось в один их возможных стилей разработки. В моду вошли новые штучки, которые с

тем же азартом обещают осчастливить мир единственно правильным и неповторимым образом; ладно, пусть немного остынут — и займут свое место среди всего прочего. А пока — все современные языки программирования имеют встроенные средства ООП, необходимые хотя бы для того, чтобы завлекать новых приверженцев из рядов бывших конкурентов.

В действительности объектно-ориентированный стиль — это эклектическая смесь двух основных парадигм: *объект* и *класс*. Объекты — в общих чертах представляют собой набор данных (свойства) и кусков кода (методы); классы — это, по сути дела, библиотеки подпрограмм для доступа к свойствам и методам объекта, рассматриваемого как *экземпляр* класса (хотя тот же объект может, в принципе, быть экземпляром другого класса или синглетоном, или еще чем-нибудь). На уровне языка объекты и классы организуют для своих компонент относительно замкнутые пространства имен — однако в статически компилируемом коде эта возможность практически обессмысливается.

Принято считать, что традиционный императивный стиль противоположен объектно-ориентированному. Императивно построенная программа представляет собой, грубо говоря, последовательность (возможно, распараллеленную) вызовов функций (операторов), работающих в контексте соответствующих входных данных и системного окружения, а иногда имеющих ряд параметров, определяющих индивидуальный контекст (замыкания). Задать раздельные пространства имен в императивном программировании не представляет особого труда: можно использовать специальные соглашения об именовании, блоки и модули, препроцессоры и т. д. Степень инкапсуляции не зависит от предпочитаемого стиля: существуют объектно-ориентированные языки без малейших следов инкапсуляции — тогда как императивный код может всегда быть разбит на полностью изолированные фрагменты, взаимодействующие только через специальный интерфейс

Иерархичность — общая черта как объектно-ориентированного, так и императивного стиля. Объекты состоят из других объектов, классы наследуют что-то от других классов, функции могут вызывать другие функции.

Никакой принципиальной разницы между императивным и объектно-ориентированным программированием нет. И однако в некоторых языках со строгой типизацией (вроде Java или C#), идея класса раздута до абсурда — и нельзя определить данные или код иначе как через специально придуманный класс. Как и везде, излишняя строгость превращается в полную бессмыслицу, делает язык громоздким и уродливым, невыразительным и непрозрачным.

Один из типичных способов скрытого ухода от объектно-ориентированной идеологии — статические свойства и методы. Это не что иное как возможность скормить системе что-то необъектное через задний проход. Нет абсолютно никакой разницы между следующими фрагментами кода (не привязанного к какому-либо конкретному языку):

```
class QQ { static public DWORD Zero = 0; }
DWORD x = QQ.Zero;
```

и

```
#define QQZero 0;
DWORD x = QQZero;
```

кроме того, что второй вариант проще и логически прозрачнее. Аналогично, статические методы класса — это просто обходной путь для определения самостоятельных функций, не связанных ни с каким классом вообще. Единственным оправданием могла бы послужить потребность разделения областей видимости — но для этого не нужны классы, достаточно определить пространства имен, которые представляют собой лишь контексты трансляции и не имеют отношения к типизации (для чего и служат классы). Оставаясь своего рода расширением инструкций препроцессора, пространства имен сродни интерфейсам, шаблонам и прочим языковым конструкциям, призванным преодолеть врожденную заскорузлость классов. В реальной жизни, хорошее соглашение о стандартах именования оказывается куда полезнее и удобней, чем лингвистические извращения.

Могут возразить, что, помещая статические компоненты в класс, мы задает тем самым специфические права доступа к ним. Лично мне по жизни ни разу не приходилось сталкиваться

с ситуациями, в которых подобные ограничения были бы действительно нужны — большей частью, они лишь запутывают логику взаимодействия и становятся головной болью для программиста, когда приходится приспосабливать чужой код к изменившимся требованиям заказчика. В таких делах минимальный корпоративный стандарт намного эффективнее и удобнее, в смысле гибкости и логической прозрачности.

К тому же, когда статические свойства и методы уникальны в пределах проекта, даже соображения разделения доступов и областей видимости не оправдывают складирования их в класс. Это бросается в глаза, например, в отношении метода `main()` в программах на Java и C#, который не имеет ничего общего с объектным поведением, а просто ссылается на операционное окружение, на способ создания управления процессами в операционной системе. В терминах классов, можно было бы представить себе класс `Task`, который описывает всевозможные программы (вычислительные операции) — и произвести от него класс `MyTask`, каким-то образом определяя для него индивидуальный конструктор, эквивалент `main()`. Потом экземпляр класса `TaskManager` (созданный экземпляром класса `OperationSystem`, созданным экземпляром класса `BIOS`, созданным экземпляром класса `Computer`, созданным экземпляром класса `PowerButtonPresser` и так далее) вызывает встроенный метод `StartTask(Task* pointer_to_task_definition)` принимающий в качестве параметра определение класса, содержащее что-то вроде

```
Task instance_of_mytask = pointer_to_task_definition -> new();
```

Это наглядно показывает отличие программного материализма от объектно-ориентированного идеализма — то есть, определение класса трактуется как объект, а не как искусственная конструкция этапа компиляции. Практические реализации таких объектов могут быть разным, от чистых интерпретаторов (в духе JavaScript) до динамически подгружаемых библиотек в машинном коде (подобно технологии ActiveX) — с промежуточными вариантами вроде JVM, интерпретирующих частично оттранслированный код. И однако можно легко понять, что у всех этих реализаций есть нечто общее — протокол. Именно наличие таких стандартных соглашений позволяет работать с классами, совместимыми с данным протоколом, — в этом смысле протокол можно было бы назвать классом (или, скорее, категорией) определений класса. В частности, протокол задает способ построения класса по его определению. Так, можно потребовать, чтобы всякое определение класса (трактуемое как объект) имело стандартную точку входа, соответствующую методу `new()`. Но можно сделать иначе — динамически вызывать этот метод из определения класса. Разумеется, никто не мешает придумать еще тысячу возможных протоколов.

В каком-то смысле, протокол можно уподобить виртуальной машине, которую можно запустить для обработки соответствующих классов в любом окружении. Если подходящей виртуальной машины не найдется — операционная среда просто не сможет запускать задачи такого типа и вернет стандартный код ошибки, означающий что-то вроде `UNKNOWN_PROTOCOL`.

Возвращаясь к статическим компонентам классов (свойствам или методам), отметим, что они представляют собой часть описания особой задачи — реализуют стандартный метод `new()` для создания среды выполнения. А потому совершенно не нужно включать их в определение вызываемого класса. Функция `main()` в такой трактовке понимается как реализация (экземпляр) некоторой виртуальной машины (протокола) для обработки динамически определяемых классов. В целом, вычислительный процесс образует иерархию протоколов, реализованных как виртуальные машины разных уровней, — и можно сделать следующий шаг, рассматривая не только фиксированные иерархические структуры, но также возможные обращения иерархии и ее развитие. Но это уже другая история.

Защита на дурака

Как известно, на всякую технологию есть контртехнология: против любой атаки — защита, на любую защиту — хак. Нет защиты только от человеческой тупости — но это уже,

скорее, из области медицины, а не про компьютеры.

Компьютерная безопасность давно уже стала весьма прибыльным направлением бизнеса, на эту тему много и умно пишут, получают бешеные гранты и защищают диссертации… Со стороны выглядит солидно. Хотя иной раз и закрадывается отвращение к самой необходимости от кого-то защищаться, тратить время и деньги не на удовлетворение насущных потребностей, а на обеспечение эксклюзивности их удовлетворения — по-простому: чтобы другой не ловил кайф за наш счет. В разумно устроенном обществе, казалось бы, польза одному автоматически есть польза и другому, и делиться радостями для разумных существ столь же естественно, как для дикарей — разбивать друг другу черепа.

Но допустим, что наука безопасности для чего-то все же нужна. Тогда логичный вопрос: а защищает ли она на самом деле? Конечно, мы сознаем, что азартные хакеры найдут прокол в любом протоколе, и придется дыры латать. Но это, так сказать, экзистенциальная печаль — тогда как в обыденной жизни хотелось бы, чтобы затраченные усилия имели хоть какой-то смысл, пусть даже временный и минимальный. А то, ведь, защитные меры превращаются часто в гальванические конвульсии — ни уму, ни сердцу.

Например, всем, вероятно, приходилось регистрироваться на разных сайтах, или в иных компьютерных приложениях. Их авторы всерьез полагают, что возможно защититься от дурака, вводя разного рода ограничения на структуру пароля — и это гордо именуется «качеством» или «степенью защищенности»! В качестве популярных мер — запрет коротких паролей, требование обязательного смешения строчных и заглавных букв, букв и цифр, присутствия не буквенно-цифровых символов… Где логика? Общеизвестно, что любое ограничение сокращает количество возможных вариантов — и тем самым задача взломщика только облегчается. И не надо мне про психологию — что, дескать, людям свойственна лень и они чаще выбирают короткие простенькие пароли… Точно так же можно сказать, что злоумышленники легко учитывают существующие ограничения и не заморачиваются проверкой заведомо неподходящих вариантов. А что касается психологии… Легкий пароль и запомнить легко — а трудные приходится записывать, и хранить запись в легко доступном месте (а иначе какой смысл?). Стало быть, и вредителям такие пароли вполне доступны. Получается, что в уродском стремлении защитить дурака от него самого мы лишь даем ему иллюзию защищенности — и толкаем на более раскованное поведение, подставляем под неприятности.

Еще один идиотизм: в старые временя мне частенько приходилось покупать что-то через Интернет. Для этого достаточно было иметь кредитную карту (для того я ее, собственно, и получал). Потом о моей «безопасности» позаботились: вместо прямого платежа, надо еще и вводить одноразовый пароль, так что приходилось дополнительно забивать себе голову и время заблаговременным получением списка таких паролей в банке или в банкомате. Но потом кому-то пришло в голову, что и это недостаточно «безопасно» — и теперь одноразовые пароли можно получить только через SMS-портал банка. Выходит, карта у меня есть — а заплатить я не могу, поскольку для этого нужно еще иметь и мобильный телефон (и не абы какой — а геморройным образом зарегистрированный). А если нет у меня телефона? Ну, не люблю я их, по жизни! Или SMS отключил из-за спама. В офисе у меня всегда есть Интернет — а, вот, телефон почти не ловит, экранируется. В заграничном отеле Интернета сколько угодно — а, допустим, тот самый мобильник я дома забыл. И так далее. Зачем тогда банковская карта, если использовать ее нельзя? Но даже если телефон под рукой, и связь есть, всякий платеж способен превратиться в русскую рулетку, ибо работают SMS-порталы не без кривостей, и у меня бывали случаи, когда код приходил на телефон спустя два часа (!) после запроса (за это время сеанс в платежной системе уже успевает отвалиться — и все надо начинать сначала).

Пошли дальше. Попробуйте разумно объяснить, почему, открывая страницу HTML с локального диска, я не могу использовать управление объектами Flash через JavaScript. Если страница открывается с сервера — пожалуйста. А без Интернета — черта с два. Дело тут не в лени программиста — я всегда могу поднять на локале виртуальный Web-сервер и отлаживать сайты сколько угодно. Но если само приложение рассчитано на то, чтобы люди скачивали страницы (или раздел сайта) на диск и смотрели, когда им удобно, не заморачиваясь выходом в

Сеть? И надо, чтобы это было красиво. И чтобы музыка играла уместная, и только тогда, когда требуется. Не загонять же всех на виртуальные машины! Писать полностью на флэшах — тот еще геморрой, у них есть свои ограничения. Вот и выходит, что для локальной и сетевой версий приходится, по сути, писать разные приложения, совмещая их на одной странице. Реакции типа «используйте HTML 5» не принимаются; это из серии: «зачем вы вообще в этой Вселенной? родились бы в какой-нибудь другой…»

Коммерческий характер современных технологий «безопасности» очевиден на примере зашифрованной электронной почты. Если Вы заходите на сайт по HTTPS, и Ваш браузер по каким-то причинам сомневается в предложенных с той стороны сертификатах, в большинстве (правильно сконфигурированных) систем Вы получите уведомление и предложение выбрать между продолжением на свой страх и риск — и отказом от дальнейших контактов; как правило, можно добавить сертификат в локальный список надежных и в дальнейшем работать без недоразумений. Но когда речь идет об электронной почте, такой непосредственный контроль уже невозможен, и Ваш почтовый ящик обороняют роботы, принимая или отвергая сообщения в зависимости от встроенного в них интеллекта. И вот тут начинается чехарда… Мало того, что популярные системы спамообороны используют сетевые черные списки (в которые каждый желающий может поместить любой адрес — и создать кому-то букет неприятностей); мало того, что почту они фильтруют, исходя из абстрактных правил без учета специфики адресата и часто даже не давая ему вмешаться, — но они, ведь, еще и сертификаты принимают не всякие, а только из доверенного списка! Если у меня корпоративная почтовая система шифрует трафик с использованием своих собственных сертификатов — при попытке контакта с внешним адресатом я могу получить в ответ все, что угодно, если тамошние админы (как это часто бывает) не позаботились о корректном формировании сообщений об ошибках (и приходится долго догадываться об истинной причине отлупа). Казалось бы, какая разница? Почему почту нельзя шифровать незнакомым ключом? Риски при этом минимальны. В конце концов, можно при сомнениях автоматически перенаправлять почту в нешифрованный канал (желательно, с формированием соответствующего уведомления). Но нет! — надо заставить людей платить официально признанным игрокам на рынке сертификатов, или хотя бы регистрировать свои сертификаты в чьих-то системах, тоже не бесплатно.

Старая присказка гласит: если надо надежно защитить компьютер — следует привязать его к атомной бомбе и сбросить в глубокую шахту, а потом все хорошенько забетонировать. Но и тут остается риск, что объявится потом благодетель, ради вашего душевного спокойствия успевший сделать полный бэкап. И в этом изначальная ущербность технологий защиты. Намагниченный диск (или распределение заряда в полупроводнике) — это еще не информация. Информацией оно становится при передаче от одного человека к другому. Компьютеры лишь обслуживают наше общение (хотя когда-нибудь они, вероятно, смогут общаться и без нас). Если я что-то знаю — это еще не знание; надо, чтобы и кто-то другой это знал. Но если есть потенциальная возможность передачи — есть и возможность перехвата. Чисто физически, процесс передачи неизбежно распадается на несколько этапов — и если мы защитим один из них, всегда остается риск вмешательства на какой-то другой стадии. Даже прямая передача из мозга в мозг не гарантирует интима, ибо встроенная аппаратура кодирования и передачи — это все равно физический прибор, который возможно обнаружить извне, да и сама работа мозга связана с поддающимися регистрации физическими процессами. Будем мы использовать сверхдлинные ключи или квантовое шифрование — совершенно без разницы. То, что доступно одному, может быть доступно и другому.

Смешно смотреть, как строители Интернет-сайтов пытаются обезопасить их от взлома и копирования. Они используют защищенный код, многослойный дизайн и защитные скрипты… Но в конечном итоге на экране все равно должна появиться картинка. И дальше с этим кто угодно может делать что угодно. В частности, можно скопировать дизайн и данные, и создать независимый сайт по образу и подобию, совершенно неотличимый от исходного по функционалу. Чем, кстати, и пользуются преступники, подставляя пользователям липовые сайты вместо настоящих с целью хищения личных данных (прежде всего финансовых). Копировать сайты могут и роботы — достаточно вместо тупых запросов смоделировать

человекообразное поведение; и если одна программа умеет генерировать кэпчу — другая всегда сможет ее распознать и ввести правильный код.

В связи с этим — о безопасности в Интернете. Когда-то все начиналось с простой идеи: нужна такая организация сетей, чтобы частичное их разрушение не приводило к параличу, чтобы возможность общения сохранялась всегда. Потом запаниковали: живучесть «правильной» циркуляции данных оказалась неотделима от гомеостаза несанкционированных трансакций — криминальные сети восстанавливаются после любых ограничений и запретов. Технологии шифрования работают против своего клиента, ибо они защищают злоумышленника столь же успешно, повышают уровень его анонимности. Блокируя одни каналы, мы оставляем миллионы других, а перестройка информационных потоков — вопрос времени.

Оказывается, что любое централизованное вмешательство в работу сетей больше мешает людям, чем ограждает их от злоумышленников. Начальство занято главным образом защитой коммерческих интересов богатых спонсоров — на простых смертных ему глубоко наплевать. Тогда населению приходится брать дело своей защиты в собственные руки — и появляются прокси-сервера (к сожалению, не всегда бесплатные), анонимайзеры, шифрованные и пиринговые сети. Даже в рамках существующей инфраструктуры все это успешно ограничивает возможности централизованного (коммерческого) контроля — хотя (как и любая технология) может быть использовано во вред.

Но Интернет — предприятие не из дешевых. Требуется развитие и поддержка глобальной сети, включая спутниковые средства связи, магистральные линии передачи, разрешение имен… Следовательно, всегда сохраняется давление грубой силы: неугодных властям могут попросту обесточить. В частности, любую закрытую сеть можно вывести из строя по указанию сверху, перекрыть каналы связи, конфисковать ключевые сервера. Разумеется, при наличии резервных копий все можно восстановить на других узлах и в других сетях — но придется заново связывать отдельные устройства в единую систему, а это иногда требует значительных усилий и времени. По-настоящему устойчивые коммуникационные сети нуждаются в технических решениях, освобождающих связь от централизованного воздействия. Надо уметь соединяться с другими устройствами независимо от доступности тех или иных средств связи; если не удается одним способом — автоматически искать другой. Такое обобщение P2P следует строить не поверх обычных сетей, а как альтернативное решение, не требующее ни обязательной регистрации, ни правительственной поддержки. Каналом передачи становится все окружающая среда — и никакими армейскими акциями вычистить это нельзя. Единственный метод — истребление человечества целиком. Вопрос, следовательно, стоит не о том, как защититься друг от друга, а о том, как преодолеть любые барьеры, облегчить контакты, а не затруднить их.

По сути, в такой сети уже не требуется шифрование. Да, она уязвима, в нее легко могут проникнуть вредители и диверсанты. Но там, где нет центра, никакие локальные разрушения не выведут из строя систему в целом. Там, где нет лидерства — никакие политические убийства не изменят общественный строй. Будущее надо строить на единстве, а не на разобщенности.

Индустрия компьютерной безопасности усложняет технологии — а значит, снижает их надежность, создает лишнюю нагрузку на пользователей, связывает их, встраивает в систему. Получается, что мы сначала изобретаем что-то, способное облегчить труд человека, — а потом ограничиваем доступ настолько, что этим уже почти невозможно пользоваться. Точно так же, мы сначала строим хорошие дороги — а потом укладываем на них «лежачих полицейских», чтобы не дать людям возможности нормально по хорошим дорогам ездить. Лихачей это не остановит — а порядочным водителям (и их пассажирам) сплошные неприятности.

Неуемная пропаганда технологических решений способна (или кому-то нужна, чтобы) отодвинуть в тень вопрос о разумном переустройстве человеческой жизни. Людям говорят: не надо думать — за вас подумают; не надо ни о чем заботиться — за вас позаботятся. У нас умные машины, они умнее вас. Вот и получается, что защищаем-то мы не данные или технологии — мы защищаем дурака от всякого интеллекта, позволяем ему вариться в собственной дурости.

Безопасность — это экономическая и социальная проблема, и любые технологии играют здесь сугубо вспомогательную роль, обеспечивают деятельность, а не направляют ее. Если

экономика работает для общего блага, а не ради богатой верхушки, — нет смысла воровать. Если люди привыкли уважать друг друга — не нужно ни за кем-то подглядывать, никого контролировать или подталкивать. А пока навороченные средства защиты разрабатывают лишь для того, чтобы продать их именно тем, кто грабит народ, чтобы сколотить себе кругленький капитал, — когда только воры против воров, — никто не может быть уверен в собственной безопасности.

Целостность распределенных данных

Технологии поддержания целостности данных часто связаны с ограничениями доступа, использованием шифрования и сертификатов (электронная подпись). Предполагается, что злоумышленник просто не может корректным образом изменить данные, если не завладеет соответствующими ключами. Иначе говоря, несанкционированное вмешательство либо вообще невозможно, либо оставляет легко обнаружимый след. Например, отличие текущей hash-суммы исполняемого модуля с такой же суммой, записанной при авторизованном запуске, может говорить о вмешательстве в систему — это обычный механизм антивирусной защиты. В древности в качестве такой суммы использовался размер файла — потом придумали хитрые алгоритмы.

Частный случай той же методики — связывание различных записей в распределенные структуры, при котором вмешательство в одном узле нарушает связи с другими узлами, что может быть легко обнаружено. Например, как в алгоритмах типа блокчейн.

При мало-мальски трезвом размышлении становится понятно, что все это паллиативы, способы затруднить работу вредителям — но никоим образом не устранить саму возможность вмешательства. С одной стороны, те же технические средства, которые позволяют внедрить сложные методы контроля, повышают эффективность программ взлома, и злоумышленник вполне способен обеспечить соблюдение формальных критериев целостности (как раньше научились сохранять подделывать даты создания и модификации, или адреса электронной почты). С другой стороны, в распределенных системах проверка целостности неизбежно распределена во времени; если система большая, времена задержки становятся вполне осязаемыми — и внутри таких «мертвых» интервалов легко размещаются сколь угодно сложные преступления. Это, так сказать, квантовое шифрование наизнанку. Достаточно долей миллисекунды, чтобы перебросить деньги на счет злоумышленника; потом испорченную цепочку обнаружат и заблокируют — но дело уже сделано, и что толку после драки кулаками махать?

Главная уязвимость компьютерных систем — пользователи. Передача паролей коллегам и знакомым — обычное дело. Копирование ключей электронной подписи — на каждом шагу. Неужели генеральный директор будет сам лезть в бухгалтерскую программу и подписывать финансовые документы личным ключом? Да никогда! Это делает рядовой бухгалтер, которому передают и контейнеры ключей, и все остальное. А значит — доступ может поиметь и кто угодно еще.

С точки зрения нормального человека, формальные методы защиты — источник всяческих неудобств. Каждый поставщик изобретает собственные программы для доступа к предоставляемым сервисам, установка которых на компьютерах пользователей предполагает многоступенчатые процедуры активации, и затрудняет параллельную работу нескольких операционистов; системному администратору приходится придумывать, как сделать личные ключи общественными, а пользователи тем временем пишут на бумажке инструкции по входу в систему и раздают доступы кому не лень. Программы «защищенного» обмена данными конфликтуют между собой и с офисными приложениями; они тормозят систему и требуют предоставления пользователю практически полных прав — чем и пользуются ушлые вирусы.

Остается только припомнить всевозможные ошибки формальных проверок, из-за которых работа может, при плохом раскладе, вообще остановиться на долгие часы и дни, до выяснения обстоятельств и получения лекарств. Тривиальный случай: неправильная дата. В

распределенной системе даты на разных железках синхронизуются лишь в общих чертах, и от локальных настроек может поплыть что угодно. Не будет сисадмин залезать к каждому на домашний компьютер или в ноутбук, чтобы выставить все по корпоративным стандартам, — тем более, что такая стандартизация запросто может конфликтовать с требованиями другой фирмы, на которую тот же юзер работает параллельно с какими-то офисными обязанностями. Я уже не говорю о работе с публичных терминалов в гостиницах и аэропортах.

Одно и то же может по-разному выглядеть в разных средах. Простая пересылка файла по электронной почте способна нарушить его формальную целостность. Информация о файле в юниксах отличается от параметров того же файла под Windows или в качестве документа какой-нибудь базы данных. Здесь нет и не может быть единства: методы «защиты» быстро мутируют, и уследить за всеми нововведениями просто нереально.

О том, как антивирусы режут заведомо безобидные файлы, все достаточно наслышаны. Бороться с этим приходится при помощи разного рода списков исключений — то есть, путем внедрения в систему дополнительных уязвимостей. Пример из практики: хваленый Яндекс-браузер, который не только сам себя рекламирует как верх добропорядочности, но еще и файлы пользователя обещает всемерно оберегать, внесли в список доверенных приложений. Через месяц на месте браузера оказалась искусная имитация турецкого изготовления, заповедник троянов.

Наконец, сложные технологии поддержания целостности сами по себе становятся критической уязвимостью там, где действия злоумышленников направлены на разрушение системы как таковой (предполагая, что кто-то заплатит за это больше, чем можно выкачать из взломанной сети). Испорченные (и выведенные из оборота) элементы накапливаются, и в какой-то момент внутренние связи оказываются совершенно неработоспособными — их просто недостаточно для формальных проверок. Иногда целое восстанавливают с грехом пополам — иногда проще выкинуть все и начать с нуля (или от состояния на момент создания полной резервной копии — а в сложной распределенной системе такие копии вряд ли возможны). Атаки DDoS — старинная боль. Еще в XIX веке перед войной спецслужбы вбрасывали на рынок страны-противника море фальшивых банкнот и ценных бумаг, чтобы подорвать экономику и облегчить военную часть операции. То же ожидает и рынок криптовалют. Хотя, между прочим, криптовалюты — лишь разновидность финансовых пирамид, и самое что ни на есть мошенничество. Вор у вора ворует.

Но допустим, что перед нами вполне разумная задача: организовать массовую работу с данными таким образом, чтобы никакие (случайные или умышленные) флуктуации не влияли на доступность и согласованность. Такова была утопическая мечта изобретателей Интернета — но в условиях рыночной конкуренции идея быстро превратилась в свою противоположность, стремление защитить всех от всех. Быть может проблема вообще неразрешима? Если думать о корысти — пожалуй, так. А мы будем думать о вечном — и строить на века.

Что требуется? Две взаимно дополнительные вещи: избыточность и самовосстановление. Проблема эффективности (как при проектировании RAID-массивов) здесь не стоит. Важнее соблюсти единство информационного поля.

В бухгалтерских книгах издавна существует система двойной записи: дебет — кредит. Это прототип того, что нам требуется. Если, допустим, одна запись исчезла — ее легко восстановить по оставшейся. Однако, если записи в паре вдруг стали различны, — какую из них считать правильной? С формальной точки зрения они совершенно равноправны. Но если у нас не двойная запись, а многократное повторение одного и того же в разных вариантах — проблемы нет: локальные разрушения устраняются, исходя из критерия согласованности копий. И не нужно формальное удостоверение подлинности — мы решаем по смыслу, а не по форме. Документ считается подлинным не потому, что он кем-то там подписан, а потому, что он используется на практике именно в таком виде, а не в каком-то еще. Ясно, что особые «законодательные» органы здесь уже не обязательны: всякий может предложить свой вариант документа — а общество на практике определит его жизнеспособность. В конце концов, разные варианты могут сосуществовать как местные традиции; так сама идея документа изменяется: это уже не единичный текст, а иерархия текстов. Но так ли уж драматично нововведение?

Испокон веков законодательные акты (и религиозные тексты) копировались множеством переписчиков (или мнемоников, как в античной Индии) — без этого иначе донести их до массового пользователя невозможно. Сегодня любой законодательный акт существует не только в виде нескольких завизированных экземпляров на разных языках (с указанием, какой из них считать правильным в случае разночтений) — но и во множестве печатных изданий, в электронных копиях, в базах данных и т. д. Поскольку практические решения принимаются, исходя из копий, факт существования «оригиналов» к делу, в общем-то, и не относится. Если же говорить о неофициальных текстах, традиция их вольной трактовки — еще древнее. Не надо мне про авторство. Народ вправе выбрасывать лишнее и добавлять отсутствующее — так литература превращается в фольклор. Что есть каноническая версия анекдота? Какой миф правильнее? Всегда ли вариация хуже темы? Какой из домов одной серии каноничнее? И не все ли равно, в какой день начинается следующий год?

Иерархические распределенные системы хранения данных абсолютно устойчивы. Их строение отражает строение человеческой культуры — и уничтожить их можно только вместе с человечеством.

На практике выборка данных выглядит примерно так: мы сравниваем полученное из разных источников и выбираем наиболее подходящее для наших задач. В принципе, статистику предпочтений несложно где-то сохранить в качестве еще одного документа — и использовать для реорганизации сети в целом; однако таких статистик может быть много, и все они лишь частично выражают объективно сложившиеся связи вещей. «Клиент» иерархической распределенной базы данных будет автоматически подстраиваться под интересы владельца и предлагать варианты, исходя из его собственного опыта, а не желаний (и бюджетов) возможных поставщиков. Это прямо противоположно тому, как Web-сайты навязывают пользователям рекламу, — при всем схожести основного механизма.

Например, я скачиваю с разных сайтов заинтересовавшую меня книгу, музыку, видеозапись, или программу. Из разных вариантов я выбираю что-то для себя, или комбинирую разные версии, — и открываю доступ к (части) моей коллекции всем желающим. Отличие от обычной торрент-сети — в универсальности «каталога», так что любой «документ» становится автоматически доступен публике сразу же после опубликования, и не надо регистрировать источник в разного рода трекерах, подстраиваясь под их требования. Это как бы синтез торрент-сети и поисковика. Пусть практика определит, чья коллекция лучше отвечает духу места и времени.

Ясно, что небольшие вариации могут со временем привести к перестройке иерархии, изменению «стандартов» и «законов». При желании, можно организовать массовую кампанию за то или иной новшество, сознательно внедряя в сеть соответствующие мутации. И какой-нибудь идиот способен заставить этим заниматься вредительскую (или рекламную) программу. Но если каждый берет из сети лишь то, что ему действительно нужно, это не изменит объективной структуры предпочтений, которая зависит только от потребителя, а не от поставщика. Дурные флуктуации сами собой сходят на нет. Полезные — закрепляются и задают новые направления развития. Да, какое-то время можно морочить людям головы — но действительность в конце концов все расставит по местам.

Система «авторизации» и удостоверения «подлинности» — попытка заморозить какую-то из исторически сложившихся структур раз и навек. В развивающемся мире такая целостность не может быть устойчивой. Как бы ни запрещали пиратские сети, как бы ни перекрывали им кислород, всегда найдется способ обхода блокировок — точно так же, как любые деньги можно подделать, а любой закон обойти. И сама общественно-экономическая система, нуждающаяся в искусственных ограничениях и законах, в промывании мозгов, в секретах и хакерах, в борьбе партий и корпораций, — неизбежно уступит место новому миру, где все для всех, где нет собственности как таковой, и глупо задаваться вопросом, что кому принадлежит.

Личности без лица

Счетчик посещаемости — обычный элемент каждого живого сайта. Устанавливают их на своих страницах все кому не лень, с самых первых шагов публичного Интернета. Но есть маленькая тонкость: далеко не всякий сайт сидит на собственном оборудовании, в собственной сети, и поддерживается владельцами или авторами самостоятельно. По большей части Web-хостинг арендуют у кого-нибудь, материалы размещают посредством каких-то провайдерских приспособлений, и посмотреть динамику посещаемости могут лишь в пределах того, что дают. Если дизайн сайта заказывают на стороне, он и получается «профессиональный» — то есть, слепленный из стандартных компонент, приспосабливать которые к интересам заказчика никто не собирается. В частности, счетчики посещаемости в лучшем случае реализуют на базе типового скрипта, написанного неизвестно кем неизвестно под кого, — а то и просто арендуют как дополнительный сервис, так что установка сводится к добавлению на каждую страницу текста со ссылкой, а деньги берут за право посмотреть, что в итоге получается.

Вот и давайте смотреть, что получается.

Где-то в глубине заказчик (или автор) надеется, что от счетчика будет хоть минимальная польза. И долго пытается выискать ее в предоставленных таблицах, графиках, семантических схемах и прочей инфографике. Чаще всего оказывается, что результат этих изысканий — почти ноль, исключая уйму потерянного времени. А продавцу до этого и дела нет: он сбагрил, что хотел, — а лохи пусть сами выпутываются. Когда поток наивных мечтателей иссякнет — сервис и вовсе прикроют, и ищите какую угодно замену, и платите за изменения в дизайне…

За десятилетия существования Сети ее характер существенно изменился. Поначалу туда шли люди творческие, которым хотелось показать то, что другим способом показать было просто невозможно, — а при случае и подсмотреть свежую идею. Гуглов еще не было, и бродили по Сети вольные серферы, и обменивались интересными адресами с друзьями и коллегами. Сейчас Сеть — нечто совсем иное. Прежде всего, это инструмент коммерции, виртуальная витрина с роботами-продавцами. То есть, никакого смысла в выставляемом контенте нет, он там просто неуместен: вместо информации — одна реклама (то есть, дезинформация). Обратная сторона того же самого — всевозможные блоги, социальные сети и тому подобное, где, опять-таки, совершенно неважно, что выставлять, — лишь бы привлечь внимание; здесь впаривают не вещи, а идеологию. Иногда и магазины, и чаты маскируют под справочники, информационные ресурсы. Но не секрет, что средства массовой информации не для того, чтобы информировать, а для того, чтобы мозги промыть; когда этим занимается индивидуал-самоучка, выглядит совсем грубо.

Творческие натуры пытались (и до сих пор пытаются) приручить дикий Интернет, использовать хотя бы такие, не для того созданные инструменты в качестве зова в пустоту, маяка для прочих одиноких странников. Возникают социальные сети по интересам — и становится не интересно. Ну, выставил какой-то математик препринт будущей статьи — что с того? Ему все равно придется публиковаться по правилам, и приносить нетривиальные находки в жертву академическим стандартам. Обсуждения, взаимопомощь? Ровно два раза. Полезный комментарий — из области редчайшей экзотики, поскольку читатель со стороны в сути вопроса не очень разбирается, а с теми, кто разбирается, автор и так общается по личным каналам, и лишний раз светиться в Сети им вовсе незачем. Особенно, когда на чем-то можно наварить. Точно так же, были раньше сайты свободной публикации для литераторов — и приходили таланты, и разговаривали на равных… В итоге — большая мусорная куча, и проще дотянуться до родственного разума как-нибудь иначе. А может быть, и не до чего, и незачем дотягиваться.

Зайдем с другой стороны: зачем народ нас посещает? Кому оно нужно?

Уровень первый: (хотя бы потенциальные) покупатели. Хочется жене безделушку на годовщину, себя любимого вкусненьким побаловать, или начальству заслать стратегический обзор… Такой визитер заходит на сайт с конкретной целью: потратить (свои или чужие) тугрики. И надо таких по максимуму стричь. Выигрывает тот, у кого портал шире, и можно просеивать посетителей как планктон, добывая успешные сделки по методу больших чисел. На них, как правило и ориентированы типовые счетчики посещаемости, с отчетностью в виде

сколь угодно гнусной статистики. Понятно, что из бесконечности цифр всегда можно выудить положительный тренд и предъявить тому, кто все оплачивает. Именно коммерсанты обеспечивают производителям и предоставителям счетчиков посещаемости (какими бы высокими словами их не называли) платежеспособный спрос.

Следующий уровень: искатели халявы. Надо кому-то быстренько скачать файл, найти цитату, шпаргалку к экзамену, склепать из готовых кусков реферат… Они быстренько забегают на страницу, выдергивают что-то свое, не обращая внимание на все остальное, — и навсегда исчезают. Считать их бесполезно — от любых попыток завлечь и впарить они отбрыкиваются всеми техническими и поведенческими средствами. И не остается в них от ваших текстов равным счетом ничего. Они роются в поисковиках, просеивают тонны мусора и рекламы — но в конце концов добиваются своего и успокаиваются. Это как бы коммерческий портал наизнанку. Изучение потока таких посетителей больше характеризует строение (и политику) популярных поисковиков, а не актуальность сайта.

Высший уровень — заинтересованные. Те, кому нужны идеи. Они возвращаются, перечитывают тексты несколько раз. Иногда даже осмеливаются побродить по ссылкам. Они создают закладки в браузерах и сохраняют страницы на локальный диск. Они делятся находками с такими же чудаками — и те тоже многократно возвращаются к пройденному… Никакая статистика этого не покажет. Тут нужен инструмент другого типа, способный выловить осмысленные визиты и по каждому показать картину живого общения, познакомить с личностью. Такие счетчики не купишь — это надо ваять самому, в зависимости от строения и содержания сайта, и иметь прямой доступ к результатам, чтобы при случае подкрутить логику. Для чего?

В древности серферы могли запросто связаться с автором интересной статьи и вывести общение в офлайн. Личности не любят лишней публичности. Сейчас это редкость. Личность зачастую не имеет лица, она существует виртуально, как тип сетевого поведения. Таковая магистральная линия общественного развития: от прямых контактов к универсальному опосредованию. По большому счету, нам не важно, что собеседник соотносится с физическим телом, территориально расположенным в каких-то местах, имеет паспорт с официальной фамилией или обходится блатной кликухой… Все это шелуха. Личностями собеседники становятся друг для друга только в творческом общении, обмениваясь пристрастиями или судьбами. Частные воплощения — смертны, а идеи остаются навсегда. Так вот, пока я не растерял последние крупицы разума, я заинтересован в том, чтобы искорки моей разумности подхватил кто-то другой — и разбудил огонь, и передал дальше, в пространстве и во времени. Когда я вижу заинтересованных посетителей, я знаю: что бы там дальше ни произошло, мое дело будет жить, разойдется по тысячам душ. Можно вещать в пустоту, верить, что прав и не нуждаться в сочувствии. Какие-то из моих идей начинают пробиваться в жизнь (вероятно, без моего влияния, как объективная необходимость) лишь десятилетия спустя. Истинность других обнаружится через века. Но есть подозрение, что человек не может остаться человеком, не обращая внимания на плоды своего труда. Потому что разум — это не только познание, но и волнение, и мечта. И надо личности говорить с личностью, а не взирать на мир свысока. Только тогда разум будет по-настоящему бессмертным — и я стану его лицом.

Психономика

Каждому компьютерщику знаком авторитарный стиль вездесущей номенклатуры: стоит кому-то заделаться хоть минимальным начальством — у него в голове происходит фазовый сдвиг, и все происходящее во Вселенной выстраивается под одну доминирующую потребность. Такой мини-, макси-, или мега-босс требует от якобы подчиненных беспрекословного выполнения начальственных указаний, немедленного удовлетворения любых прихотей. Разумеется, ни о каком ожидании речи быть не может: все нужно не сходя с места и прямо сейчас. Любимое словечко всякого руководства — «срочно». Дескать, я сейчас хочу. А буду ли хотеть через энное время — неизвестно.

Разумеется, относится это не только к компьютерному хозяйству. Это болезнь классовой экономики как таковой. Когда все равно заинтересованы в труде на общее благо, никому и в голову не придет работать локтями и пропихивать свои задачи в ранг приоритетных. Но когда за непослушание могут уволить с волчьим билетом — приходится делать поправку на бесноватость, и начинается вульгарное затыкание дыр: экстремальное программирование, заплаточное администрирование, наращивание вычислительных мощностей и уплотнение коммуникаций, и т. д. — полный букет производственных авралов. Иногда поиски путей выживания порождают новые технологические решения. От этого господа еще больше надувают щеки и провозглашают: вот, видите, какая от нас польза! Ущерб жизненным интересам и здоровью персонала в расчет можно не принимать.

Архитектура компьютерных систем во многом копирует классовое устройство общества. Однако на ранних этапах развития надо было еще и вписаться в собственно технологические ограничения, и это создавало своеобразную зону безопасности для инженеров, программистов и сисадминов: обосновать конкретную реализацию можно было, ссылаясь на недоразвитость техники, — и неумеренные аппетиты руководству приходилось переводить в другое русло, перекладывать проблему на другие плечи. Сейчас такой номер не пройдет: практически все технические ограничения в компьютерной области отменены, и любой запрос принципиально выполним — за соответствующие деньги. Стремительный рост компьютерной образованности создает ту самую «резервную армию труда», из которой черпают дешевые мозги, и куда продвинутому айтишнику никак не хочется попасть. Приходится изворачиваться. Изобретать технологии адекватного реагирования. В том числе, в области архитектуры.

Как известно, традиционная парадигма машины Тьюринга предполагает переработку ресурсов (данных) и производство чего-то еще по некоторому алгоритму. Не принципиально, будет обработка последовательной или параллельной, локализованной или распределенной, дискретной или аналоговой, детерминированной или стохастической. В итоге должно получиться то, что можно предъявить заказчику и потребовать причитающихся за работу знаков стоимости. Часть результата каждой операции может быть конвертирована в данные. Другая часть встраивается в алгоритм работы (включая аппаратные решения). Остальное — непроизводительное потребление, издержки производства (хотя заказчику именно это важнее всего).

Поскольку возможности машины ограниченны (а это предполагается самим актом выделения ее из окружающей среды), поток запросов на обработку данных приходится регулировать. Скорость подачи данных должна соответствовать скорости их переработки. Дикому капиталисту объяснить это почти невозможно: он же хозяин — а рабы должны исполнять приказы. Для смягчения противоречий вводят буферные устройства, которые в реальной жизни называются управленческим персоналом, а в компьютерной индустрии — очередями. На эту тему вариаций — немеряно. Два древнейших архитектурных решения: FIFO (first in first out) и LIFO (last in first out). То есть, в первую очередь выполняем либо то, что раньше заказали — либо актуальное на данный момент (стековая организация). Начальство, конечно, знает только второй вариант. Для каждого начальника его приказ — главнее всех. Дали команду — тут же начинайте выполнять, а все остальное побоку. Армейские уставы устроены по тому же принципу — хотя подчиненный обязан предупредить начальника о наличии указаний от вышестоящей инстанции. Однако приоритет стековой обработки в компьютерных системах неизбежно приводит к неприятным последствиям: если не разгребать завалы, стек переполняется, и система подвисает. Нужны радикальные решения, перезагрузка. В рыночной экономике это называется кризисом. И решения не менее радикальны: море крови и груды костей.

Очередь FIFO с точки зрения кризисов намного благополучнее: мы просто пропускаем мимо ушей запросы, которые переполняют очередь, — и спокойно обрабатываем то, что уже успели принять. Среда вынуждена регулироваться сама; многие запросы устаревают к моменту, когда в очереди появилось для них местечко. Однако начальству очень не по душе, когда приходится прикручивать аппетиты. С другой стороны, бывает и объективная необходимость по-настоящему срочных мер, откладывать которые на потом — себе дороже. Как быть?

Заметим, что распараллеливание машины Тьюринга не дает существенного выигрыша в части оперативности обработки. Растет быстродействие — но очередь-то на всех одна, и появляются накладные расходы: синхронизация потоков, разграничение доступов, обработка конфликтов, агрегирование результатов и т. д. То есть, решать каждую задачу в отдельности мы можем быстрее — а система в целом все равно буксует.

Естественный ход — переход к архитектуре с несколькими очередями. Например, при критическом заполнении стека можно отправлять запросы в очередь FIFO. Здесь есть свои заморочки — но в целом станет легче дышать. Еще спокойнее — иерархический вариант FIFO, когда поступающие задачи сразу сортируются по приоритетам, и есть очередь в режиме ожидания (pending), есть очередь регламентных процедур, есть очередь срочных задач, очередь супер-срочных, очередь жизненно важных (критических) и т. д. Исполнитель (какой угодно архитектуры) выбирает задания из этой иерархической очереди по определенному правилу. Алгоритмы можно выстраивать в зависимости от обстановки — однако в любом случае проблем с начальством уже нет: на любой запрос можно сообщить о постановке задания в очередь; если начальник скажет: еще быстрее, — прекрасно, сообщаем о переводе в очередь более приоритетных задач. Тонкость в том, что далеко не всегда срочные задачи выполняются раньше низкоприоритетных — но начальство-то об этом не знает!

Действительно, посмотрим на простейшую логику обработки иерархических очередей — технику мультиплексирования в потоках типа FIFO. Приоритетным очередям выделяется больше временных слотов (тактов выборки) или больший процент общей полосы пропускания. Однако необходимость выполнения менее приоритетных задач никто не отменял, и они вклиниваются в поток срочных дел, в пределах отведенной квоты. Тем самым вся система постоянно на ходу, и не застревает ни в одном из звеньев. Допустим, один слот регулярных задач приходится на 3 слота срочных. Тогда каждая четвертая операция — низкоприоритетная. Если в срочной очереди 10 заданий — семь из них однозначно выполнятся после первого в неприоритетной очереди. То есть, наш исполнитель лишь *предрасположен* обрабатывать срочные задачи в первую голову — но вовсе не обязательно будет поступать именно так.

Иерархическая очередь эффективно соответствует введению еще двух промежуточных звеньев между заказчиком и исполнителем: сортировщик запросов и обработчик очередей. Найти соответствие этим уровням в производственной сфере — несложное упражнение для заинтересованного собеседника. Однако в реальной жизни существенна неоднородность экономических потоков: сейчас заказов полно — через месяц их совсем нет, приоритеты меняются на каждом шагу, по итогам работы. Соответственно, производственные мощности могут быть либо заняты под завязку — либо недогружены. Какие-то из уровней очереди переполнены — другие пустуют. Эффективное мультиплексирование требует гибких алгоритмов с динамическим распределением слотов. С другой стороны, исполнитель в большинстве случаев отнюдь не элементарен, и его внутренние состояния будут влиять на порядок работы. Кроме того задания на практике оказываются взаимозависимыми, и каждое следует обрабатывать с оглядкой на другие. Например, банк получил распоряжение оплатить счет — но, пока это задание ждет своей очереди, вступает в силу закон, запрещающий перевод средств между указанными счетами, — и поставленную в очередь задачу приходится обрабатывать иначе (отказ вместо исполнения). Возможна тонкая игра на приоритетах обработки распоряжений — и платеж пойдет, если реализация запрета по каким-то причинам запоздает. Этот трюк нам хорошо известен из истории; он всесторонне обыгран в литературе, особенно детективной направленности.

Другой пример: всякая операция требует определенных ресурсов; если необходимых предпосылок нет на момент выборки задания из очереди, исполнитель бессилен — и надо ставить задание в другую очередь, и здесь тоже возможны разные алгоритмические решения.

Таким образом, как помещение задания в очередь, так и выборка очередного задания чаще всего производятся не единообразно, а в зависимости от ситуации. В частности, порядок выборки заданий очереди может быть изменен. Такую очередь можно было бы назвать CICO (conditional in conditional out). Массовое применение этого принципа в экономике («CICO-номика», или, в соответствии с английским произношением, «психономика») не меняет ее

рыночной основы, однако регулируемость и стабильность каждой относительно замкнутой хозяйственной системы будет значительно выше. Понятно, что глобальная экономика требует столь же глобальной психономики. В общем случае, иерархичность очереди должна соответствовать масштабам экономических проектов; например, экспансия человечества в космос на каком-то этапе потребует согласования планетарных приоритетов.

Лирическое отступление: при капитализме технологии нужны не для улучшения чего бы то ни было, а для сохранения социального неравенства. Поэтому полноценная психономика здесь попросту невозможна, и можно говорить лишь о частичных реализациях.

Для рядовых программистов, системных администраторов и IT-менеджеров очереди CICO полезны в плане противодействия стрессам и перехода от экстремальности к нормальным производственным отношениям. В любой компании работник «сервисного» подразделения формально оказывается «слугой многих господ». Минимальная формализация множественной «подчиненности» эффективно выводит подчиненного на вершину иерархии, а всем остальным приходится считаться с его манерой работы, которая теперь представляется объективной необходимостью. Эффект особенно важен для работающих во вневедомственных структурах, сервисных службах, юридически независимых от заказчика (хотя экономическую зависимость, конечно же, никто не отменял). Скажем, если сисадмин поддерживает базы данных десятков компаний, он вправе наложить ограничения на использование этих баз — и это можно (при необходимости) оговорить в официальных документах. Не всякую работу мы готовы взять на себя — но если уж взялись, то делаем честно, в соответствии с нашими технологическими возможностями, и давить на нас — никакого смысла.

Особый интерес представляет психономика распределенных процессов. С одной стороны, каждый исполнитель представляет собой относительно замкнутую CICO-систему, выбирая для себя что-то из имеющихся в среде задач и запуская выполнение при наличии необходимых ресурсов. С другой стороны, среда каждого исполнителя — часть среды процесса в целом, с его собственной иерархией ресурсов и приоритетов. Это означает, что постановка задач и порядок их решения на уровне процесса обусловлены возможностями задействованных исполнителей. Простейший пример: поступает глобальный запрос — глобальная очередь структурирует его и рассылает широковещательный запрос исполнителям; если в каком-то звене активного исполнителя нет, запрос отвергается. Если задача уже включена в глобальную очередь, взаимоотношения исполнителей могут принимать разный характер. «Классическая» парадигма — конкурентная (соревновательная) обработка: кто первым захватил задание, тот монополизирует ресурсы и средства производства, и всем остальным надо ждать конца работы. Частный случай — создание рабочих групп («коллективный исполнитель»). Но есть и альтернативная, «квантовая» архитектура: несколько исполнителей параллельно работают над проблемой, каждый решает ее своим способом — а в итоге есть набор вариантов (путей решения). Заказчику все равно, как именно получен результат. Однако, как и в кантовой физике, бывают нежелательные пути решения («запрещенные переходы»), и приходится задействовать многоступенчатые схемы.

На практике используют разного рода гибридные архитектуры, когда параллельное развитие допускается на этапе создания прототипов — однако в итоге принимается классическое решение, и работа выстраивается по одной из возможных схем (мистически настроенные физики воспринимают это как «коллапс волновой функции»). Ничего сверхъестественного здесь нет, поскольку, в силу обращаемости иерархий, каждая иерархическая структура может быть свернута и развернута по-другому, в зависимости от того, что в данной ситуации «энергетически выгодно». Речь не идет об абстрактной оптимальности: ни у кого нет желания гоняться за идеалом — достаточно какого-нибудь решения, а жажда совершенства — категория не экономическая. Пока дело движется, пусть даже не совсем так, как хотелось бы, — мало кто будет рисковать готовым ради лишь теоретически возможного. Так психономика, с одной стороны, поддерживает кадровую стабильность, а с другой — создает предпосылки для переосмысления экономической «вертикали»: не персонал подчинен хозяину, а наоборот, всякое предприятие существует лишь как совместная деятельность свободных людей.

СОДЕРЖАНИЕ