Никлаус Вирт

Долой "жирные" программы

Источник: Открытые системы, #06/1996

Стало правилом: всякий раз, когда выпускается новая версия программного продукта, существенно — порой на много мегабайт — подскакивают его требования к размерам компьютерной памяти. Когда такие запросы превышают имеющуюся в наличии память, приходится закупать дополнительную. Когда же дальнейшее расширение невозможно, то надо приобретать новый, более мощный компьютер или рабочую станцию. Но идут ли большая производительность и расширенная функциональность в ногу со все увеличивающимися запросами на вычислительные ресурсы? В большинстве случаев ответ будет — нет.

Лет 25 тому назад, интерактивный текстовый редактор мог быть спроектирован из расчета всего лишь 8000 байтов памяти — современные редакторы текстов программ требуют в 100 и более раз больше. Операционная система должна была обслуживать 8000 байтов, и компилятор должен был умещаться в 32 Кбайт, в то время как их нынешние потомки требуют для своей работы многих мегабайтов. И что же: это раздутое программное обеспечение стало быстрее и эффективнее? Наоборот. Если бы не аппаратура с ее возросшей в тысячи раз производительностью, современные программные средства было бы просто невозможно использовать.

Считается, что расширенная функциональность и удобства пользователя оправдывают все возрастающие размеры программ; однако, более пристальный взгляд обнаруживает шаткость подобных оправданий. Тот же текстовый редактор все еще выполняет достаточно простую задачу по вставке, удалению и переносу фрагментов текста; компилятор по-прежнему транслирует текст в исполняемый код; и операционная система, как и в былые времена, управляет памятью, дисковым пространством и циклами процессора. Эти базовые обязанности вовсе не изменились с появлением окон, стратегии "вырезание-вставка" и всплывающих меню, так же как и с заменой текстовых командных строк изящными пиктограммами.

Столь явный взрывной рост размеров программного обеспечения был бы, конечно, неприемлем, если бы не ошеломляющий прогресс полупроводниковой технологии, которая улучшила отношение цена/производительность в степени, не имеющей аналогов ни в какой другой области технологии. Так, с 1978 по 1993 год для семейства процессоров Intel 80x86 производительность увеличилась в 335 раз, плотность размещения транзисторов — в 107 раз, в то время как цена — только в 3 раза. Перспективы для постоянного увеличения производительности сохраняются, при том, что нет никаких признаков, что волчий аппетит, присущий ныне программному обеспечению, будет в обозримом будущем утолен [1]. Такое развитие событий спровоцировало появление многочисленных правил, законов и выводов, которые, как это и бывает в таких случаях, выражаются в универсальных терминах; как таковые, их невозможно ни доказать, ни опровергнуть. Следующие два "закона" чрезвычайно хорошо (хотя и с долей иронии) отражают нынешнее положение дел:

- Программное обеспечение увеличивается в размерах до тех пор, пока не заполнит всю доступную на данный момент память (Паркинсон)
- Программное обеспечение замедляется более быстро, чем аппаратура становится быстрее (Рейзер).

Неконтролируемый рост размеров программ принимается как должное также и потому, что потребители имеют затруднения в отделении действительно существенных особенностей программного продукта от особенностей, которые просто "хорошо бы иметь". Примеры: произвольно перекрывающиеся окна, предлагаемые некритично воспринятой популярной метафорой "рабочий стол"; причудливые пиктограммы, декорирующие экран — антикварные почтовые ящики для электронной почты или корзинки для сбора мусора (которые, к тому же еще, остаются видимыми при движении к их новому местоположению). Эти детали привлекательны, но не существенны, и они имеют свою скрытую стоимость.

1. Причины громоздкости программного обеспечения

Итак, два фактора вносят вклад в приятие потребителями программного обеспечения все более растущих размеров:

- быстро увеличивающаяся аппаратная производительность
- игнорирование принципиальной разницы между жизненно важными возможностями и теми, которые "хорошо бы иметь".

Но, быть может, более важно, чем просто найти причины для такой толерантности, попытаться понять, что же обуславливает дрейф программного обеспечения навстречу сложности.

Основной причиной является некритичное принятие фирмами-поставщиками практически любого требования потребителей относительно новых возможностей программного продукта. Всякая несовместимость с первоначальными концепциями системы либо игнорируется, либо проходит нераспознанной. В результате, проектные решения усложняются, а в использовании продукт становится более громоздким. Когда мощность системы измеряется числом ее возможностей, количество становится более важным, чем качество. Считается, что каждая новая редакция продукта должна предлагать какие-нибудь дополнительные возможности, даже если некоторые из них реально не добавляют функциональности.

Другая важная причина, ответственная за программную сложность, лежит в "монолитном" дизайне, когда все мыслимые возможности сразу закладываются в систему. Каждый потребитель платит за все возможности, но реально использует лишь немногие из них. В идеале же, должна предлагаться только базовая система с заложенными в нее существенными возможностями, но эта система должна иметь потенциал для различных расширений. Тогда каждый потребитель мог бы выбирать функции, действительно необходимые для его задачи.

Возросшая производительность аппаратуры, несомненно, явилась стимулом для разработчиков при атаке на более сложные проблемы, а более сложные проблемы неизбежно требуют более сложных решений. Однако, речь идет не об этой внутренне присущей программным системам сложности, о которой и должна болеть голова; мы говорим здесь о сложности, искусственно привнесенной. Существует масса проблем, давно уже решенных, но ныне нам предлагаются "новые" решения тех же проблем, завернутые в более громоздкую программную оболочку.

Возросшая сложность по большей части является следствием наших недавно возникших пристрастий к "дружественному" пользовательскому интерфейсу. Я уже упоминал окна и пиктограммы; сюда же можно добавить цвет, полутона, тени, всплывающие меню, всевозможные картинки и диалоговые "реквизиты" различных типов.

1.1 Сложность как эквивалент мощности

Легкость использования системы всегда должна быть главной целью, но эта легкость должна опираться на лежащие в основе системы концепции, что и позволяет сделать работу с ней почти интуитивной. Кажется, однако, что чем дальше, тем больше люди склонны неверно истолковывать сложность как изощренность, которая сбивает с толку — а ведь непостижимость должна вызывать подозрение, а не восхищение.

Возможно, эта тенденция происходит от сомнительной веры в то, что до некоторой степени таинственное средство сообщает ауру чего-то сверхестественного пользователю (хотя что оно действительно "сообщает", так это чувство беспомощности, если не бессилия). Поэтому, соблазн сложности как стимула для продаж легко понятен; сложность способствует поддержанию зависимости потребителя от поставщика.

Ни для кого не секрет, например, что основные фирмы-производители программного обеспечения осуществили — и с успехом! — массированные инвестиции в сервисное обслуживание, наняв сотни консультантов, призванных круглосуточно отвечать на звонки пользователей. Было бы, однако, много более экономичней как для них, так и для их клиентов, если бы программный продукт основывался на систематических концептах (универсально справедливых правилах вывода, а не на таблицах правил, применимых только к специфическим ситуациям) в сочетании с систематической документацией и обучающими курсами.

Конечно, клиент, который платит — вперед! — за договорный сервис, являет собой более стабильный источник дохода, чем клиент, который полностью самостоятельно освоил продукт. Промышленность, вероятно, преследует цели, весьма отличные от принятых в академическом мире; следовательно, можно сформулировать еще один "закон": зависимость клиента более доходна, чем его обучение.

Что я нахожу истинно ставящим в тупик — так это руководства и документация — объемом в сотни страниц!, которые сопутствуют прикладным программам, языкам программирования и операционным системам. Безошибочно, они сигнализируют как об извращенном проектировании без четкой концептуальной базы, так и о намерении запудрить пользователю мозги.

Однако, одно лишь отсутствие ясной концептуальной основы не может отвечать за взрывной рост размеров программного обеспечения. Проектирование решений для сложных проблем, возникают ли они на программном или аппаратном уровне, это трудный, дорогой и требующий много времени процесс. Столь улучшенное аппаратное отношение цена/производительность достигнуто больше вследствие лучшей технологии промышленного копирования проектов, чем из-за более совершенного владения методами проектирования. Создание программного обеспечения, однако, и есть проектирование как таковое, а его копирование стоит производителю копейки.

Первоначальный проект программной системы для сложных приложений неизменно сам получается сложным, даже когда разрабатывается компетентными инженерами. Истинно хорошие проектные решения возникают после итеративных улучшений или после перепроектирования, которое опирается на качественно новое понимание проблемной области и методов решения задачи, и наиболее успешные итерации - это те, которые приводят к упрощению программ. Эволюции такого рода, однако, чрезвычайно редки в текущей практике — они требуют длительного мыслительного процесса, что весьма редко оценивается так, как подобает. Вместо этого, неадекватности в программах обычно корректируются предлагаемыми на скорую руку "дополнениями", которые неизбежно приводят к самым несоразмерным объемам.

1.2 Времени всегда не хватает

Постоянный недостаток времени — вот, вероятно, первейшая причина, приводящая к появлению громоздкого программного обеспечения. Спешка, в условиях которой работают проектировщики, не способствует тщательному планированию и усовершенствованию принятых решений; зато она потворствует возникающим на ходу программным добавлениям и корректировкам. Спешка мало-помалу понижает инженерные стандарты качества и совершенства и оказывает крайне вредное влияние на персонал, а значит и на разрабатываемые продукты.

Еще одной характерной чертой компьютерной индустрии является тот факт, что поставщик, которому удалось первым выбросить продукт на рынок, как правило, получает ощутимые преимущества над конкурентом, чей аналогичный — и лучший по качеству! — продукт появляется вторым. Тенденция принимать первый появившийся продукт в качестве de facto-стандарта — это крайне прискорбный феномен, вызванный к жизни все той же спешкой.

Хорошая инженерная практика характеризуется последовательным пошаговым усовершенствованием продукта, что и приводит к увеличению производительности при заданных ресурсах и ограничениях. Однако, ограничения на вычислительные ресурсы не считаются сколь-либо серьезными и с легкостью игнорируются: кажется, присутствует всеобщая вера в то, что быстрый рост скорости процессора и размеров памяти компенсируют допущенные при проектировании ПО небрежности. Тщательная инженерная работа не приносит дивидендов в лихорадочных гонках на короткие дистанции, и это одна из причин, почему программная инженерия имеет сомнительную репутацию среди устоявшихся инженерных дисциплин.

2. Языки и методология проектирования

Хотя исследования в области разработки ПО, являющиеся ключевыми для многих будущих технологий, пользуются неплохой финансовой поддержкой, их результаты, судя по всему, признаются не слишком уместными для использования в современной программной индустрии. Систематическое методичное проектирование, например, имеет репутацию не слишком подходящего, так как для выхода на рынок продуктов, таким образом разработанных, будто бы требуется слишком много времени. Еще хуже обстоят дела с аналитической верификацией и методами доказательства корректности; помимо прочего, эти методы требуют более высокого интеллектуального калибра, чем вошедший в привычку "пробуй и все получится" подход. Неудивительно, что в свете любви потребителей ко всякого рода "бубенчикам и свистулькам",

предложения о сокращении сложности с помощью концентрации на базисных принципах отметаются как заведомо абсурдные. Когда в качестве modus operandi выступает возглас "а все работает!", методологии и дисциплина разработки становятся первыми жертвами.

Методологии, связанные с языками программирования, до сих пор являются предметом дискуссий. В 1970-х было принято верить, что проектирование программ должно опираться на хорошо структурированные методы и слои абстракции с четко определенными спецификациями. Лучше всего эта мысль выразилась в концепции абстрактного типа данных, которая и нашла свое воплощение в новых тогда языках, прежде всего в Modula-2 и Ada. Сегодня программисты оставляют хорошо структурированные языки и мигрируют, в основном, к Си. Язык Си не позволяет компилятору даже выполнять контроль безопасности типов, а ведь именно эта функция в наибольшей степени полезна при разработке программ для локализации концептуальных ошибок уже на ранней стадии. Без контроля типов само понятие абстракции в языках программирования становится пустым и имеющим чисто академический интерес. Абстракция может работать только в языках, постулирующих строгий статический типовой контроль для каждой переменной и функции. В этом отношении Си несостоятелен и, в сущности, подобен ассемблерному коду, где, правда, "все работает".

Весьма примечательно, что абстрактный тип данных через 25 лет после своего изобретения появился вновь под названием "объектно-ориентированный". По своей сути этот современный концепт (принимаемый многими как панацея) более всего связан с построением иерархий классов или типов. Более старое понятие не было, в сущности, понято, пока не появился новый ярлык "объектно-ориентированный"; теперь же программисты признали присущую абстрактному типу данных мощь и обратились, наконец, к нему. Однако, чтобы об объектно-ориентированных языках можно было говорить всерьез, в них должна быть реализована строгая статическая типизация, которую нельзя было бы нарушить; это дало бы возможность программисту полагаться на компилятор в деле идентификации разного рода несогласованностей. К сожалению, наиболее популярный язык, C++, неудовлетворителен в этом отношении, потому что было изначально декларировано, что он должен быть совместим со своим предком — языком Си. Широкое принятие C++ подтверждает следующие "законы":

- Прогресс приемлем, только если он совместим с текущим состоянием.
- Приверженность стандарту — всегда безопаснее, чем даже мотивированный отход от него.

Принимая эту ситуацию как данную свыше, программисты вступают в борьбу с языком, который не поощряет структурное мышление и дисциплинированное построение программ, отрицая базовую поддержку компилятора. Они также прибегают к инструментам-паллиативам, которые еще более способствуют разрастанию размеров программ.

Что за мрачная картина; каков пессимист! — должно быть, думает читатель. И никакого намека на светлое компьютерное будущее, которое, вроде бы, считается само собой разумеющимся. На самом деле, этот мрачный взгляд является реалистическим; тем не менее, если имеется желание и воля, то можно найти способ улучшить нынешнее состояние дел.

3. Проект "Оберон"

Между 1986 и 1989 годами, Йорг Гуткнехт (Jurg Gutkneht) и я разработали и реализовали новую систему программного обеспечения — названную Оберон, предназначенную для рабочих станций, и которая опирается непосредственно на аппаратные средства и, соответственно, не использует каких-либо сторонних программ. Нашей основной целью было продемонстрировать — программное обеспечение может быть разработано так, чтобы использовать лишь небольшую часть памяти и процессорной мощности от обычно требуемых; при этом нет необходимости жертвовать гибкостью, функциональностью или удобствами пользователя.

Начиная с 1989 года, система Оберон широко используется во многих предметных областях: подготовка документов, разработка программного обеспечения и автоматизированное проектирование электронных схем. Система включает:

- управление памятью;
- файловую систему;

- оконную систему управления отображением данных;
- сеть с серверами;
- компилятор;
- редактор документов;
- текстовый и графические редакторы.

Спроектированный и реализованный — от нуля — коллективом из двух человек за три года, Оберон за прошедшее время был перенесен на несколько серийно выпускаемых рабочих станций и приобрел симпатии многих преисполненных энтузиазма пользователей, особенно с тех пор, как стал свободно доступным [2].

Дополнительной нашей целью было спроектировать такую систему, которую можно было бы легко объяснять и изучать во всех подробностях; подходящую для использования в качестве примера проектирования программной системы; и которая была бы "прозрачна" сверху донизу. Все основные проектные решения могут быть наглядно представлены и объяснены. (Действительно, практическое отсутствие опубликованных детальных примеров создания программных систем очевидно каждому, кто оказывался перед необходимостью чтения соответствующего учебного курса). Плодом наших усилий явилась книга, которая содержит полное описание системы вместе с исходными текстами всех модулей.

За счет чего же удалось, затратив всего пять человеко-лет, построить такую представительную программную систему, да еще и дать ее исчерпывающее описание в одной-единственной книге [3]?

3.1 Три базисных правила

Во-первых, мы сосредоточились на действительно существенных особенностях системы. Мы пренебрегаем всем, что не оказывает существенного влияния на необходимую мощность и гибкость системы. Например, взаимодействие с пользователем в базовой системе ограничивается режимом ввода/вывода текстовой информации — никакой графики, никаких картинок и пиктограмм.

Во-вторых, мы хотели использовать истинно объектно-ориентированный язык программирования — способный обеспечить безопасность типов. В сочетании с нашей верой в то, что первый принцип является даже более обязательным для инструментальных средств, чем для самой создаваемой системы, это вынудило нас спроектировать собственный язык и сконструировать для него компилятор. Так и возник Оберон [4] — язык, в сущности полученный из языка Modula-2 путем удаления из него "ненадежных" особенностей (таких как функции преобразования типа и вариантные записи) вместе с не слишком существенными (примеры — тип-диапазон и перечислимый тип).

Наконец, мы полагали, что система должна быть гибко расширяемой — это и позволит ей стать простой, эффективной и полезной. Следовательно, в нее могут быть добавлены новые модули, которые включают в себя новые процедуры на основе вызова уже существующих. Это также означает, что в новых модулях могут быть определены новые типы данных, совместимые с существующими типами. Мы называем их расширенными типами, и они представляют собой единственное фундаментальное понятие, которое было добавлено к Modula-2.

3.2 Расширение типа

Если, например, тип Viewer определен в модуле под названием Viewers, то тип TextViewer может быть определен как расширение Viewer (как правило, в другом модуле, который добавляется к системе). Всякая операция, применяемая к Viewers, может быть применена и к TextViewers, и всякое свойство, которое имеют Viewers, также относится и к TextViewers.

Расширяемость гарантирует, что модули могут быть добавлены к системе позже, и при этом не потребуется ни изменений, ни перекомпиляции. Очевидно, что типовая безопасность является в этом контексте критической и должна свободна распространяться через границы модулей.

Расширение типа является типичной объектно-ориентированной особенностью. Чтобы избежать вводящего в заблуждение антропоморфизма, мы предпочитаем говорить "TextViewers совместимы с Viewers", а не "TextViewers наследуют от Viewers". Мы также избегаем введения совершенно новых терминов для хорошо известных понятий; например, мы остаемся верными термину "тип", избегая слова "класс"; мы сохраняем термины "переменная" и "процедура", избегая новых терминов "экземпляр" и "метод". Несомненно, что наш первый принцип — сосредоточение на сути — равно применим и к терминологии.

3.3 Пример типа данных

Рассмотрим пример типа данных, который будет иллюстрировать нашу стратегию построения основной функциональности в базовой системе, а также особенности, добавленные в соответствии с принципом расширяемости системы.

В системном ядре тип данных Text определяется как последовательность символов с атрибутами font, offset и color. Базисные операции редактирования заложены в модуле с названием TextFrames.

Модуль электронной почты Email не включен в базовую систему, но может быть добавлен по запросу. Этот новый модуль, будучи надстройкой над базовой системой, импортирует типы Text и TextFrame, позволяющие работать с фрагментами текста. Это означает, что к получаемым по электронной почте сообщениям могут быть применены обычные операции редактирования. С помощью базисных операций эти сообщения могут быть модифицированы, скопированы и вставлены в другие фрагменты текста, высвечивающиеся на экране дисплея. Что же касается операций, обеспечиваемых самим модулем Email, то к ним относятся операции получения, отсылки и уничтожения сообщений, а также команды для работы с каталогом почтовых ящиков.

3.4 Активизация операций

Другим примером, иллюстрирующим нашу стратегию, будет активизация операций. В Обероне выполняются не программы а, отдельные процедуры экспортируемые из модулей. Если некоторый модуль M экспортирует процедуру P, то P может быть вызвана простым указанием на строку M.P, присутствующую в любом видимом на экране тексте — то есть с помощью позиционирования курсора на строку M.P с последующим щелчком клавишей мыши. Такая простая процедура активизации открывает следующие возможности:

1. Часто используемые команды представляются в виде небольших порций текста. Они называются "tool-texts" и напоминают настраиваемые меню, причем никакой специальной программной меню-поддержки не требуется. Обычно они отображаются в небольших просмотровых окнах.

2. Система может быть расширена простым графическим редактором, который обеспечивает заголовочные надписи, основанные на текстовых строках Оберона. Это позволяет выполнять подсветку команд либо украшать их рамками или тенями. Как результат, всплывающие и/или ниспадающие меню, кнопки и пиктограммы реализуются "бесплатно" — за счет многократного использования базисного механизма активизации команд.

3. Сообщение, получаемое по электронной почте, может наряду с обычным текстом содержать и команды. Эти команды могут быть выполнены путем все того же единственного щелчка клавишей мыши, выполняемого получателем (без копирования соответствующего участка сообщения в специальное командное окно). Мы используем эту особенность, например, при уведомлении о новых или модифицированных версиях модулей. При этом сообщение обычно содержит команды "receive", за которыми следует список имен модулей, которые нужно загрузить из сети. Весь процесс требует лишь нескольких щелчков клавишей мыши.

3.5 Сохраняйте систему простой

Стратегия поддержки системы простой, но расширяемой, с лихвой вознаграждает пользователей без больших амбиций. Ядро Оберона занимает меньше, чем 200 Кбайт — включая редактор и компилятор. Компьютерная система на основе Оберона нуждается в наращивании лишь в случае работы с особо требовательными к наличию вычислительных ресурсов приложениями, такими как системы автоматизированного проектирования с их значительными аппетитами к памяти. Если используются несколько таких приложений, система не требует их одновременной загрузки. Эта экономичность достигается за счет следующих свойств системы:

1. Модули могут загружаться по запросу. Сигнал на запрос дается либо когда активизируется команда (которая определена в еще не загруженном модуле), либо когда загруженный модуль импортирует другой модуль, еще не присутствующий. Загрузка модуля может инициироваться самим фактом запроса на доступ к данным. Например, когда к документу, содержащему графические элементы, просит доступа редактор, чей графический пакет не открыт, то этот доступ скрытым образом инициирует его загрузку.

2. В памяти присутствует лишь один экземпляр любого модуля. Это правило запрещает создание связанных загрузочных файлов (исполняемых модулей). Как правило, связанные загрузочные файлы вводятся в операционных системах из-за того, что процесс связывания является сложным и требующим много времени (иногда даже больше, чем компиляция). В Обероне связывание не может быть отделено от загрузки, что вполне приемлемо, так как эти трудноразделимые процессы легко зафиксировать: они происходят автоматически, когда встречается первая ссылка на модуль.

3.6 Цена простоты

Опытный инженер, понимая, что бесплатных обедов не бывает, несомненно спросит, а где же скрыта цена этой экономичности? Упрощенный ответ таков: в четкой концептуальной базе и в хорошо продуманной, адекватной структуре системы.

Для обеспечения расширяемости ядра системы (или любого другого модуля) проектировщик должен понимать, как будет использоваться система. Действительно, именно декомпозиция на модули является тем аспектом проектирования, который требует наибольшего внимания. Каждый модуль должен иметь точно определенный интерфейс, специфицирующий механизмы импорта и экспорта.

Каждый модуль также инкапсулирует методы реализации. Все процедуры должны быть совместимыми с точки зрения обработки экспортированных типов данных. Точное определение надлежащей декомпозиции — это непростая задача, которая редко может быть решена сразу, без последующих итераций. Конечно, итеративные (настраивающие) усовершенствования возможны вплоть до времени выпуска системы в свет.

Правила проектирования с трудом поддаются обобщению. При определении абстрактного типа данных следует тщательно обдумывать аккомпанирующий набор базовых операций; что касается составных операций, то их следует избегать. Можно с уверенностью сказать - общепринятое правило о необходимости законченного формулирования спецификации до реализации должно быть ослаблено. То, что спецификация неадекватна, может выясниться, когда реализация потерпит неудачу.

Заключение или девять уроков от Оберона

Следующие девять уроков, извлеченные из проекта Оберон, стоило бы иметь в виду каждому, собирающемуся взяться за новый программный проект:

1. Исключительное использование сильно типизированного языка было фактором, в наибольшей степени определившим саму возможность проектирования такой сложной системы в столь короткий срок. (Реализация сравнимых по размерам проектов с опорой на язык с ослабленной типизацией неизбежно потребовала бы значительного расширения и людских, и иных ресурсов). Статическая типизация, во-первых, позволяет компилятору с высокой степенью точности идентифицировать возможные несогласованности перед выполнением программы; во-вторых дает возможность проектировщику изменять определения и структуры с меньшей опасностью негативных последствий; в-третьих

ускоряет процесс усовершенствования системы, включающий, вероятно, такие изменения, которые в ином случае не могли бы рассматриваться как осуществимые.

2. Самой трудной проектной задачей является нахождение наиболее адекватной декомпозиции системы на иерархически выстроенные модули, с минимизацией функций и дублирования кода. Оберон способен оказать хорошую поддержку в этом отношении с помощью распространения механизма проверок типов поверх границ модулей.

3. Присутствующая в Обероне конструкция расширения типа оказалась весьма существенной для проектирования расширяемой системы там, где новые модули добавляли функциональность, и новые классы объектов безболезненно интегрировались в одно целое с уже существующими классами и типами данных. Расширяемость является предпосылкой для поддержания системы достаточно простой и рациональной с самого начала работы. Она также позволяет в любое время приспособить систему к любому специфическому приложению, причем без доступа к исходному коду.

4. Ключевой вопрос в расширяемой системе — вычленение тех базовых примитивов, которые обеспечивают наибольшую гибкость при расширении; в то же время следует избегать разрастания набора примитивов.

5. Ошибочно мнение, что сложная система требует целой армии проектировщиков и программистов. Если нет одного индивидуума, понимающего проект во всей его полноте или хотя бы в весьма значительной степени подробностей, то с большой вероятностью система построена не будет.

6. По мере разрастания команды проектировщиков растут и проблемы взаимодействия между ними. Когда эти проблемы начинают — явно или неявно — преобладать над содержательными, и команда, и проект погружаются в пучину больших проблем.

7. Сокращение сложности и размеров системы должно быть приоритетной целью на каждой фазе разработки — при специфицировании, проектировании и непосредственно программировании. О компетентности программистов следует судить по способности находить простые решения, а вовсе не по "производительности", измеренной "количеством строк кода, выданных на гора за день". Не в меру плодовитые программисты способны привести проект к катастрофе.

8. Единственный способ приобретения опыта — это собственная программистская практика. Команда, состоящая из менеджеров, проектировщиков, программистов, аналитиков и пользователей с едва пересекающимися функциями, едва ли будет жизнеспособна. Все должны участвовать (с различной степенью вовлеченности) во всех аспектах разработки. В частности, каждый — включая менеджеров — должен время от времени выступать в роли пользователя. Эта мера — лучшая гарантия корректировки ошибок и также, возможно, устранения избыточности.

9. Программы должны шлифоваться до тех пор, пока не приобретут достойное для публикации качество. Конечно, спроектировать программу, которую можно опубликовать, бесконечно более трудно, чем ту, которая просто способна "исполниться". Программы должны быть рассчитаны на людей в такой же степени, как и на компьютеры. Если это положение вступает в противоречие с некоторыми корпоративными интересами, принятыми в коммерческом мире, то по крайней мере в академической среде оно не должно встретить сопротивления.

С помощью проекта Оберон мы продемонстрировали, что гибкая и мощная система может быть создана с привлечением существенно меньших ресурсов и в более короткое время, чем обычно. Эпидемия неконтролируемого роста размеров программного обеспечения не обусловлена никаким "законом природы". Этого можно избежать, и именно инженер-программист призван отсекать всякую избыточность в разрабатываемых им программах.

Литература

- E.Perratore et al., "Fighting Fatware", Byte, Vol.18, No.4, Apr. 1993, pp.98-108.
- M.Reiser, The Oberon System, Addison-Wesley, Reading, Mass., 1991
- N.Wirth and J.Gutknecht, Project Oberon — The Design of an Operating System and Compiler, Addison-Wesley, Reading, Mass., 1992
- M.Reiser and N.Wirth, Programming in Oberon - Steps Beyond Pascal and Modula, Addison-Wesley, Reading, Mass., 1992