Журнал "Мир ПК", #08, 2001 год // Издательство "Открытые Системы" (www.osp.ru)
Постоянный адрес статьи: http://www.osp.ru/pcworld/2001/08/136.htm

Критика Си++. Виртуальные функции*

Ян Джойнер

01.10.2001

Язык программирования работает на многих уровнях и выполняет различные функции, а потому должен критически рассматриваться по отношению именно к этим уровням и функциям. Именно виртуальные функции — основной объект критики языка Си++.

Си++ представляет собой интересный эксперимент по адаптации возможностей объектной технологии к традиционному языку программирования. Бьерн Страуструп вполне достоин аплодисментов за то, что ему в голову пришла мысль слить обе технологии воедино. В то же время в Си++ сохранились проблемы старого поколения средств программного производства. Язык Си++ обладает тем преимуществом перед Си, что поддерживает некоторые аспекты объектной технологии, которые могут быть использованы для ограниченного проведения анализа требований и проектирования. Однако процессы анализа, проектирования и реализации проекта все еще в значительной степени остаются внешними по отношению к Си++. Таким образом, в Си++ не реализованы важные преимущества объектной технологии, которые прямо бы привели к экономичному производству программной продукции.

Виртуальные функции

Полиморфизм — основополагающая концепция объектно-ориентированного программирования. В языке Си++ ключевое слово virtual предоставляет функции возможность стать полиморфической, если она будет переписана (переопределена) в одном классе-потомке или более. Однако слово virtual отнюдь не является необходимым, так как любая функция, переопределенная (overriden) в классе-потомке, может быть полиморфической. Компилятору только требуется генерировать коммутирующий код для истинно полиморфических процедур.

Если автор родительского класса в языке Си++ не предвидит, что класс-потомок захочет переопределить функцию, то он не сможет сделать ее и полиморфической. В этом заключается наиболее серьезный порок Си++, поскольку снижается гибкость программных компонентов, а следовательно, и

способность создавать адаптируемые и расширяемые библиотеки.

Си++ также позволяет функциям быть перегруженными (overloaded); в такой ситуации вызов нужной функции зависит от аргументов. Различие между перегруженными и полиморфическими (переопределенными) функциями состоит в том, что в перегруженных функциях нужная определяется при компиляции, а в случае полиморфических определяется при выполнении.

Когда родительский класс проектируется программистом, то приходится только догадываться, может ли быть функция переопределена или перегружена. Класс-потомок способен перегружать функцию в любое время, но это не соответствует более важному механизму полиморфизма, когда автор родительского класса должен точно задавать, что процедура будет виртуальной. Тогда компилятор установит коммутирующую запись для данной функции в таблице переключения классов. Значит, на программиста ложится забота обо всем том, что автоматически должно выполняться самим компилятором и что совершает компилятор в других языках. Такой пережиток унаследован Си++ из-за того, что первоначально он был реализован с помощью инструментария UNIX, а не с помощью специальной поддержки компилятора и компоновщика.

Виртуальные функции предоставляют один из возможных путей реализации полиморфизма. Разработчик языка может сделать выбор в пользу определения полиморфизма либо в родительском, либо в наследующем классе. Так что же из них имеет смысл выбрать разработчику языка? Здесь можно выделить несколько вариантов для родительских классов и классов-потомков, которые не станут взаимоисключающими и смогут достаточно легко найти себе место в любом объектно-ориентированном языке.

Существуют три варианта, связанные с переопределением, которые описываются словами «не должно», «может» и «должно».

1. Переопределение процедуры запрещено. Классы-потомки должны использовать процедуру в том виде, как она есть
2. Процедура может быть переопределена. Классы-потомки могут применять процедуру такой, как она есть, или же обеспечить свою собственную реализацию в строгом соответствии с первоначальным описанием интерфейса
3. Процедура является абстрактной. Реализация не предоставляется, и каждый неабстрактный класс-потомок должен обеспечивать свою собственную реализацию. Это и есть полиморфизм.

Разработчик базового класса должен принять варианты 1 и 3, а классов-потомков — 2. Для всех вариантов язык обязан предоставлять соответствующий синтаксис.

Вариант 1

Язык Си++ не запрещает переопределение процедуры в классе-потомке. Даже приватные виртуальные процедуры могут быть переопределены. Саккинен [1] указывает на то, что класс-потомок может переопределять приватную виртуальную функцию и тогда, когда к ней нет никакого доступа.

С ним тесно связан отказ от использования виртуальной процедуры. Здесь процедура может быть полностью заменена, что вызывает две проблемы. Первая заключается в том, что процедуру можно ненароком заменить в классе-потомке, и тогда компилятор будет генерировать синтаксическую ошибку из-за дублирования описаний. Это логично, так как классы-потомки являются частью того же самого пространства имен, что и классы, от которых они наследуются. Переопределение имени внутри одной и той же области видимости будет вызывать конфликт имен, т. е. если там две сущности будут иметь одинаковое имя, то это приведет к неоднозначности и другим схожим проблемам.

Вторую проблему можно рассмотреть на следующем примере (Листинг 1).

В этом примере класс B имеет расширенные или взятые из класса A замененные процедуры. Для объектов типа B должна быть вызвана процедура B::nonvirt. Программисту-пользователю класса Си++ придает гибкость, т. е. можно вызывать либо A::nonvirt, либо B::nonvirt. Но того же можно было добиться гораздо проще и более прямым путем. A::nonvirt и B::nonvirt следует дать разные имена. В этом случае программист вызывает нужную ему процедуру явно, а не за счет каких-то темных махинаций языка, приводящих к возможности ошибок (Листинг 2).

Теперь разработчик класса B имеет прямой контроль над интерфейсом этого класса. Приложение требует, чтобы клиенты класса B могли вызывать и A::nonvirt, и B::nonvirt. Разработчик класса B обеспечивает вызов явным образом, что можно считать хорошим объектно-ориентированным проектированием, при котором предоставляются четко определенные интерфейсы. Си++ позволяет программистам-пользователям класса совершать различные трюки с интерфейсами, внешними по отношению к данному классу, и разработчик класса B не в силах предотвратить вызов A::nonvirt. Объекты класса B содержат свои собственные специализированные процедуры nonvirt, но разработчик класса B не имеет достаточного контроля над интерфейсом этого класса B, чтобы гарантировать вызов корректной версии этой процедуры.

Си++ также не защищает класс B от других изменений, вносимых в систему. Предположим, что нам требуется создать такой класс C, у которого процедура nonvirt должна быть виртуальной. Для этого nonvirt в классе A

также должна быть виртуальной, что сводит на нет трюк с B::nonvirt. Требование класса C иметь виртуальную процедуру заставляет вносить изменения в базовый класс, затрагивающие всех остальных потомков данного базового класса. Это делается вместо локализации нового специфического требования в новом классе. Подобные действия идут вразрез с идеей объектно-ориентированного программирования иметь слабосвязанные классы, для того чтобы новые требования и изменения обладали локальным характером и не заставляли вносить коррективы повсюду. Потенциально это может привести к конфликту с другими существующими частями системы.

Еще один аргумент состоит в том, что любой оператор должен постоянно иметь одну и ту же семантику. Полиморфическая интерпретация оператора вида a->f() заключается в том, что для объекта, на который ссылается a, вызывается наиболее подходящая реализация f(), независимо от того, принадлежит ли этот объект типу A или же потомку класса A. Однако в языке Си++ для того чтобы четко представлять себе, что же вызывает a->f(), программист должен знать, определена ли функция f() как виртуальная или как невиртуальная. Следовательно, про оператор a->f() нельзя сказать, что он не зависит от реализации, и принцип сокрытия деталей реализации нарушается. Изменение в описании f() будет влиять на семантику вызова. А вот независимость от реализации означает, что ее изменения не затрагивают семантику исполняемых операторов.

Если правка в описании изменяет семантику, то будет генерироваться ошибка компиляции. Программист должен создавать оператор семантически целостным с измененным описанием, что отражает динамическую природу разработки программного обеспечения, когда текст программы постоянно изменяется.

Чтобы познакомиться еще с одним примером нарушения целостности семантики оператора a->f(), посмотрите раздел 10.9c в руководстве [2, с. 232].

Ни в Eiffel, ни в Java подобных проблем не возникает. Их механизмы проще и понятнее, они не приводят к сюрпризам, которых хватает у Си++. В языке Java все является виртуальным, и это приводит к тому, что метод должен быть не переопределен или определен с квалификатором final. Язык Eiffel позволяет специфицировать процедуру как замороженную, и тогда она не может быть переопределена в классах-потомках.

Вариант 2

Этот вариант переопределения открыт для разработчиков классов-потомков, однако в языке Си++ решение должно приниматься в базовом классе. Если говорить об объектно-ориентированном проектировании, то решения от чего-

либо отказаться столь же важны, как и на что-то согласиться. Окончательные же решения нужно оттягивать до самого последнего момента. Такая стратегия предотвращает от появления ошибок на самых ранних стадиях программирования. При принятии скоропалительных решений вы зачастую будете попадать в тупик, опираясь на то, что впоследствии окажется некорректным. Си++ требует, чтобы родительский класс специфицировал потенциальный полиморфизм через виртуальные методы (хотя промежуточный класс в цепочке наследования тоже способен вводить виртуализацию). Из этого следует, что процедура может быть переопределена в классах-потомках. Такой подход весьма проблематичен, поскольку процедуры, которые фактически не являются полиморфическими, доступны не через прямой процедурный вызов, а через несколько менее эффективную схему виртуальных таблиц. (Это не столь уж значительные накладные расходы, однако объектно-ориентированные программы имеют тенденцию использовать большое количество маленьких процедур, что в сумме ведет к повышению накладных расходов.) Политика языка Си++ состоит в том, что процедуры, которые могут быть переопределены, должны быть объявлены как виртуальные.

Вот какая критика в адрес виртуальных функций звучит из уст Румбо и его коллег [3]: «Си++ обладает механизмами наследования и динамического разрешения методов (run-time method resolution), а в то же время структура данных Си++ не является по своей природе объектно-ориентированной. Разрешение метода и способность переопределять операцию в подклассе доступны только в том случае, когда операция объявлена как виртуальная в суперклассе. Таким образом, следует предвидеть необходимость перегружать метод и заносить эту возможность в исходное описание класса. К сожалению, разработчик класса может не учесть потребности задавать специализированные подклассы и не знать, что те или иные операции будут в подклассе переопределены. Значит, суперкласс часто должен подвергаться изменениям, когда задается подкласс. Отсюда следует существенное ограничение на многоразовое использование библиотечных классов при создании подклассов, особенно когда недоступна библиотека в исходных текстах. (Безусловно, можно объявлять все операции как виртуальные, но тогда значительно возрастут накладные расходы на использование памяти и вызов функций.)»

Однако виртуализация — неудачный механизм для имеющего с ним дело программиста. Другая проблема в Си++ связана с ошибочным переопределением методов.

Компилятор может обнаруживать полиморфизм и генерировать соответствующий код виртуализации там и только там, где это нужно. Необходимость определять виртуальные функции — лишняя забота для программиста. В основном именно по этой причине Си++ и является ущербным объектно-ориентированным языком. Программист должен постоянно заботиться о низкоуровневых деталях, которые могли бы

автоматически обрабатываться компилятором.

Процедура базового класса может быть переопределена без всякого умысла. Компилятор должен сообщать об ошибочном переопределении имени в рамках одного и того же пространства имен, если только разработчик класса-потомка явно не укажет, что он делает это намеренно. Может быть использовано и то же самое имя, но тогда программист должен идти на это сознательно и задавать его явно, особенно в тех средах, где системы собираются из уже существующих компонентов. До тех пор пока программист явно не перегрузит исходное имя, будет генерироваться ошибка о том, что имеет место дублирование имен. В то же время Си++ унаследовал первоначальный подход языка Симула. В других языках он был усовершенствован, и они пошли по более удачному и более четко выраженному пути, чтобы избежать непреднамеренных ошибок переопределения.

Решение состоит в том, что virtual не должен задаваться в родительском классе. Когда требуется динамическое полиморфическое связывание на этапе выполнения, то класс-потомок должен специфицировать переопределение функции, а когда необходимо статическое полиморфическое связывание на этапе компиляции, то он будет задавать перегрузку функции. Подобная схема имеет значительные преимущества: в случае полиморфических функций компилятор может проверять соответствие заголовков функций, а в случае перегруженных — их некоторые различия.

В языке Java нет ключевого слова virtual, а все методы — потенциально полиморфические. Java использует прямой вызов вместо поиска в таблице динамических методов, где метод может быть статическим, приватным или финальным. Значит, будут неполиморфические функции, которые должны вызываться динамически, а из-за динамической природы Java дальнейшая оптимизация невозможна.

Языки Eiffel и Object Pascal решают эту проблему таким образом, что разработчику класса-потомка нужно задавать переопределение осознанно. Дополнительная выгода от подобного решения состоит в том, что впоследствии люди, изучающие или сопровождающие данный класс, могут легко выявить переопределенные процедуры и что данное конкретное описание относится именно к классу-потомку, а не к родительским классам. Таким образом, вариант 2 реализуется там, где ему и место, т.е. в классах-потомках. И Eiffel, и Object Pascal оптимизируют вызовы: они только генерируют для динамического связывания коммутирующие записи в таблице методов, когда процедура истинно полиморфическая.

Вариант 3

В варианте 3 используется чистая виртуальная функция. Процедура здесь

является неопределенной, а класс абстрактным и не допускающим непосредственного порождения. Поэтому класс-потомок должен доопределить эту процедуру. Все потомки, которые не делают этого, также относятся к абстрактным классам. Такая концепция корректна, но все-таки следует посмотреть виртуальные функции, где подвергается критике их синтаксис. Java также имеет абстрактные методы, а в языке Eiffel реализация помечается как deferred (отложенная).

Чистые виртуальные функции

Чистые виртуальные функции (pure virtual functions) предоставляют механизм для сохранения функции в виде неопределенной и абстрактной. Класс, содержащий такую абстрактную функцию, не может быть порожден напрямую, а неабстрактный класс-потомок должен определить ее. Синтаксис чистой виртуальной функции в Си++ выглядит так: virtual void fn() = 0.

Это заставляет любого, кто знакомится с текстом, догадываться о ее значении, причем даже тех, кто хорошо разбирается в объектно-ориентированных концепциях.

Запись =0 была бы полезна только для разработчика компилятора, так как при реализации в таблицу виртуальных вызовов заносится 0. Таким образом, лишний раз демонстрируется, как детали реализации, не имеющие отношения к программисту, в языке Си++ видны снаружи.

С точки зрения математики 0 отнюдь не обозначает отсутствие. Обычно 0 — это просто другое число. Использование нуля в значении «отсутствует» приводит к смысловому разрыву. В мире баз данных применяется значение «не известно». Если 0 используется в этом значении, то возникает проблема, когда значение известно и равно 0. Запись вида =0 вызывает накопление ошибок. Перегружаются не только такие ключевые слова, как virtual и static, но и 0 начинает означать то, что математически не имеет смысла. Java и Eiffel используют гораздо более понятный синтаксис. В Java это выглядит как abstract void fn(), а в Eiffel процедуру нужно задавать как отложенную (deferred). Следовательно, детали реализации откладываются до решения в классе-потомке: r is deferred end. Вы можете специфицировать и другие абстрактные свойства в виде пред- и пост-условий. В Eiffel используется более удачная терминология, так, под deferred понимается, что реализация откладывается. Процедура, имеющая реализацию, все еще обладает абстрактной формой, причем под термином «абстрактный» вовсе не следует понимать «нереализованный».

«Виртуальный» — это понятие, сложное для восприятия. Более просты для понимания соответствующие концепции полиморфизма и динамического связывания, переопределения и перегрузки, поскольку они ориентированы на предметную область. Виртуальные процедуры представляют по сути

механизм реализации полиморфизма. Полиморфизм — «что», а виртуальный — это «как». Языки Smalltalk, Objective C, Java и Eiffel используют совершенно иной механизм для реализации полиморфизма. Виртуальные процедуры — пример того, как язык Си++ нарушает концепции объектно-ориентированного программирования. Программиста заставляют работать на основе низкоуровневых концепций, а не пользоваться высокоуровневыми объектно-ориентированными. Подобные механизмы могут быть интересны для теоретиков и разработчиков компиляторов, тогда как практикам нет никакой надобности в них вникать, а нужно применять лишь в качестве высокоуровневых концепций. Неизбежность использования этих механизмов на практике приводит к весьма кропотливой и чреватой ошибками работе, что может затормозить адаптацию программного обеспечения к новым достижениям в соответствующих технологиях и механизмах ее поддержки.

Лучшим решением было бы предоставить какое-нибудь ключевое слово, например abstract. Подобный механизм должен иметь синтаксическую значимость, так как абстрактные функции — очень важная концепция в объектно-ориентированном проектировании.

Решение Си++ заключается в том, что он следует философии языка Си избегать ключевых слов, причем это зачастую идет в ущерб пониманию. Ключевое слово будет реализовывать такую концепцию гораздо яснее, например pure virtual void fn() или abstract void fn(). Математическая нотация, применяемая в Си++, предполагает, что могут использоваться величины, отличные от 0. Так, если функции будет присвоено значение 13: virtual void fn() = 13, то функция либо реализована, либо не определена. Любой аналитик все понял бы, если бы это было булево значение, которое выражалось бы одним ключевым словом. Устранить такую неясность можно, если определить =0 как abstract:

```
#define abstract = 0
virtual void fn() abstract;
```

Термин «чистая виртуальная» — некорректное языковое выражение. Это сочетание слов, несущих в себе противоположные значения. «Чистый» означает нечто реальное, что-то без примесей. Например, «чистое золото». «Виртуальный» — нечто такое, чего нет в природе. Например, «виртуальная память». Правда, «виртуальное золото» может быть фальшивым. Как уже ранее отмечалось, «виртуальный» — для понимания концепция сложная. И когда она встречается в сочетании со словом «чистый», становится совсем не по себе.

Бьерн Страуструп дает такое объяснение странному виду записи =0: «Курьезный синтаксис =0 был выбран как очевидная альтернатива введению ключевых слов pure или abstract, поскольку в то время я не видел возможности вводить еще одно ключевое слово. Я предполагал, что в Release

2.0 абстрактные классы не войдут. Чтобы не рисковать с задержкой в принятии решения и не вступать в длительные дискуссии, я использовал традиционное соглашение Си и Си++ о том, что 0 носит значение «отсутствует»».

Заключение

Наследование предусматривает тесные взаимоотношения. Оно предлагает фундаментальный путь для сборки программных компонентов. Объекты, являющиеся экземплярами некоторого класса, также будут и экземплярами всех родителей данного класса. Для эффективного объектно-ориентированного проектирования целостность этих отношений должна четко соблюдаться. Каждое переопределение в подклассе должно быть проверено на целостность с исходным описанием в родительском классе. И подкласс должен соблюдать требования, установленные в родительском классе. Требования, которые не могут быть выполнены, сигнализируют об ошибке проектирования или о том, что наследование в данном случае не подходит. Целостность в смысле наследования — это фундаментальный принцип объектно-ориентированного проектирования. Реализация в языке Си++ невиртуальной перегрузки (non-virtual overloading) и перегрузки по заголовку функций (overloading by signature) означает, что компилятор не может проверять эту целостность. Си++ не поддерживает данный аспект объектно-ориентированного проектирования. Это приводит к большому и весьма дорогостоящему разрыву между анализом требований, проектированием и реализацией.

Наследование может быть классифицировано как синтаксическое или как семантическое. Сэйк и его коллеги в работе [5] формулируют это следующим образом: «Синтаксическое наследование обозначает наследование структуры или описания методов и потому относится к многоразовому использованию программного кода (и к перегрузке кода для наследуемых методов). Семантическое наследование обозначает наследование семантики объекта, т.е. самих объектов. Этот вид наследования известен из семантических моделей данных, где он используется для моделирования такого объекта, который выступает в приложении сразу в нескольких ипостасях». Сэйк с коллегами сосредоточили свое внимание именно на семантической форме наследования. Поведенческое или семантическое наследование выражает роль объекта внутри данной системы.

Вегнер [6], однако, убежден, что наследование кода должно представлять большую практическую ценность. Он соотнес различие между синтаксическим и семантическим наследованием с иерархиями кода и поведения и сделал вывод о том, что они редко совместимы друг с другом и зачастую входят в противоречие. Вегнер также задается вопросом: «Каким образом должна быть ограничена модификация наследуемых атрибутов?» Наследование кода обеспечивает основу для модульности. Поведенческое же

наследование поддерживает моделирование через отношение «является» (is-a). И то и другое играет полезную роль. Оба требуют проверки целостности, при которой комбинации в плане наследования имеют определенный реальный смысл.

Представляется, что наследование — наиболее мощный механизм в крайне ограниченной форме семантики, сохраняющий отношения наследования. Подкласс не должен нарушать условий, установленных его родительским классом.

Программные компоненты подобны кусочкам собираемой картинки. Когда решается подобная головоломка, то необходимо, разумеется, соблюсти форму кусочков, но гораздо важнее, чтобы собранная картинка не получилась бессмысленной. Сборка программных компонентов — задача куда сложнее, ведь собираемая картинка до перемешивания ее кусочков была законченной. А сборка программных компонентов — это возведение системы, которой никогда не существовало.

Наследование в языке Си++ подобно такой собираемой картинке, где кусочки-то друг с другом стыкуются, а вот компилятор не в силах проверить, имеет ли собранная картинка какой-то смысл. Другими словами, Си++ задает для классов и наследования синтаксис, но не дает им семантики. Таким образом, язык Си++ не может обеспечить поддержку основополагающей цели объектно-ориентированного проектирования и программирования.

Литература

1. Sakkinen M. Inheritance and Other Main Principles of C++ and Other Object-Oriented Languages // Structured Programming. 1992. V.13.
2. Ellis, Stroustrup B. The annotated C++ Reference Manual. AT&T, 1990.
3. Rumbaugh, Blaha, Premerlani, Eddy, Lorensen. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
4. Stroustrup B. Design and Evolution of C++. Addison Wesley, 1994.
5. Saake, Jungclaus, Ehrich. Object-Oriented Specification and Stepwise Refinement // IFIP Workshop on Open Distributed Processing. Berlin, 1991.
6. Wegner P. Concepts and Paradigms of Object-Oriented Programming // ACM SIGPLAN OOPS Messenger. 1990. V.1, № 1.

---

* Здесь публикуется фрагмент обширной работы Яна Джойнера, впоследствии положенной в основу его книги «Objects Unencapsulated: Eiffel, Java, and C++» (Prentice-Hall, 1999). В русский вариант вошли фрагменты, взятые из 2-й и 3-й редакции работы Яна Джойнера «C++? A Critique of C++».

От автора (к русскому изданию в «Мире ПК»)

Мне очень приятно, что вы решили опубликовать в своем журнале фрагмент моей работы «C++? A Critique of C++» (1996), поскольку Си++ — язык, требующий весьма интенсивной критики. Он представляет собой не слишком удачную реализацию объектно-ориентированной технологии, и поэтому его недостатки просто необходимо подвергать критическому анализу.

Бесспорно, один из немногих, кто обозначил четкую границу для объектно-ориентированной технологии, — Бертран Мейер (Bertrand Meyer), автор языка Eiffel. Он написал несколько книг, посвященных объектной технологии, и с ними вы обязательно должны познакомиться. В своей критике Си++ ему удается обходиться без кулачного боя. Среди тех его книг, которые можно порекомендовать вашим читателям, следует отметить две, выпущенные издательством Prentice-Hall: «Object-oriented Software Construction» и «Object Success: A Managers guide to object orientation, its impact on the corporation and its use for reengineering the software process».

Я убежден в том, что объектно-ориентированная технология может многое предложить для решения проблем, стоящих перед современными сложными программными системами. Но одновременно Си++ приносит всем колоссальное разочарование — он вобрал в себя все плохие и старые средства, а также привнес в объектную технологию абсолютно ненужную сложность. Зачем нужен язык, насквозь пропитанный низкоуровневыми конструкциями? Наиболее верный путь к успеху — использование чистого объектно-ориентированного языка, обладающего интерфейсом с языком Си или с другими низкоуровневыми языками. Это обеспечит хорошие поддержку и сопровождение, переносимость и качество.

Мне хочется, чтобы вы добились успеха в создании эффективного и качественного программного обеспечения, и я считаю, что вы найдете часть ответов на стоящие перед вами вопросы именно в объектной ориентации, а не в языке Си++. Не надо думать, что критики Си++ — религиозные фанатики или чудаки, какими их пытаются представить его сторонники. Мы раскрываем недостатки Си++ и советуем обратить внимание на гораздо более совершенные подходы, которыми можно воспользоваться уже сейчас.

Журнал "Мир ПК", #08, 2001 год // Издательство "Открытые Системы" (www.osp.ru)
Постоянный адрес статьи: http://www.osp.ru/pcworld/2001/08/136.htm