ЮРИЙ РЕВИЧ

bhv

НЕСТАНДАРТНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА DELPHI

РЕЗИДЕНТНЫЕ ПРОГРАММЫ В WINDOWS

ИНСТАЛЛЯЦИЯ И ДЕИНСТАЛЛЯЦИЯ ПРОГРАММ

ПОИСК В ДОКУМЕНТАХ

РАБОТА С ГРАФИКОЙ В WINDOWS

ЛЮБИТЕЛЬСКАЯ КРИПТОГРАФИЯ

РАБОТА С COM- И USB-ПОРТАМИ

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Юрий Ревич

НЕСТАНДАРТНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА DELPHI

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.26-018.2
Р32

Ревич Ю. В.

Р32 Нестандартные приемы программирования на Delphi. — СПб.: БХВ-Петербург, 2005. — 560 с.: ил.

ISBN 5-94157-686-2

Книга призвана помочь программистам разрабатывать полноценные, профессиональные Windows-приложения в Delphi. Показано, как предотвращать повторный запуск приложения, работать с нестандартными окнами, перехватывать нажатие клавиш, создавать резидентные программы в Windows, а также инсталляторы и деинсталляторы программ, осуществлять поиск в документах, работать с COM- и USB-портами, шифровать текст и многое другое. Рассмотрены примеры решения этих и многих других проблем, которые встают при создании программы, ориентированной на долговременное использование и распространение. Приведены приемы работы с Windows API. Изложение ведется на примерах поэтапного создания реально работающих практических приложений. Компакт-диск содержит исходные тексты разобранных в книге примеров.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.09.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,16.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-686-2

Оглавление

Введение

О чем и для кого написана эта книга

> Я вообще всю жизнь полагал, что ничего хорошего из исполнения чаяний неких гипотетических потребителей, "народа", никогда не выходило. Писать (стихи, романы, пьесы, сценарии и... программы) надо исключительно для себя. Ну, и для близкого круга друзей.
>
> *Е. Козловский, "Ниоткуда с любовью"*

По классификации ученого-химика А. Шкроба, создателя хорошего сайта о науке под названием "VivosVoco", программисты делятся на любителей, дилетантов и профессионалов[1]. Любители пишут программы для развлечения, дилетанты пишут программы по необходимости, профессионалы пишут программы для заработка. Вероятно (кто бы провел такое исследование?), любителей и дилетантов больше, чем профессионалов. Мало того, с распространением Интернета и появлением онлайновых сообществ грани между любителями, дилетантами и профессионалами при разделении их по признаку цели все больше и больше стираются — к какой категории, например, отнести добровольных членов сообщества создателей Linux? Сейчас самодеятельное (объединим таким названием область деятельности любителей и дилетантов) программирование у периодических компьютерных изданий несколько не в моде — просто в силу того, что значительную часть ниши, которую раньше занимали программы непрофессионалов (точнее, программирующих специалистов), ныне занимают универсальные фирменные продукты. Однако, как заявляют авторы одного старого пособия по программированию: *"...прикладные программы, созданные программирующим профессионалом (т. е. "дилетантом" в нашей классификации — Ю. Р.), с точки зрения профессионального программиста зачастую выглядят неуклюжими и неизящными. Но зато они обладают одним общим достоинством — они действи-*

[1] А. Шкроб. Я не любитель, я другой... — Компьютерра, № 24—25, 1998 (http://www.computerra.ru/offline/1998/252/1439/).

тельно работают..."[2]. Дополним данную мысль — это происходит потому, что у дилетантов нет выхода: они пишут программы для себя, и плохо работающие им просто не нужны.

Но согласно хорошему определению, услышанному автором этих строк от одного профессионального фотографа, профессионал отличается от любителя тем, что любитель всегда ищет ответ на вопрос "как", а профессионал — "зачем". Любой любитель или дилетант в конце концов доходит до той стадии, когда ему позарез требуются некоторые функции, которые Delphi (а большинство непрофессионалов использует именно Delphi) сама по себе дать либо не может, либо их осуществление не описано в обычных учебниках и пособиях. Причем среди таких функций есть очень распространенные и необходимые. Вот таким любителям и дилетантам и адресована эта книга. Не ждите от нее последовательного изложения основ объектно-ориентированного программирования (ООП) или построения Windows API. Подобно тому, как можно грамотно писать по-русски, не понимая разницы между существительным и сказуемым, создавать вполне работоспособные программы можно без глубокого знания ООП. Правда, как и в случае грамотности, ваши умения будут ущербными в том смысле, что выйти за рамки конкретных образцов вам будет сложновато, но на основе изложенного в этой книге материала вполне можно научиться делать программы не хуже фирменных — ну, а если вас программирование заинтересует, как самостоятельный предмет, то для этого нужно читать совсем другие пособия и, как правило, не на русском языке.

Зачем все это?

Индия планирует довести экспорт программного обеспечения к 2008 году до 50 миллиардов долларов — почти в два раза больше объема российского экспорта нефти. И, хотя официальные данные по экспорту ПО из России отличаются от этой цифры примерно на два порядка (полмиллиарда в 2004 году, по официальным данным), на самом деле есть основания полагать, что официальная статистика врет — некоторое представление о реальности может дать тот факт, что около 10% shareware в мире делается в России и Украине. К тому же индусам завидовать вообще не очень хочется: они не программисты, а кодеры, рабочие, которые лишь кладут кирпичи в здание, возводимое другими. Самостоятельно в Индии не создано, вероятно, ни одной хоть сколько-нибудь известной программы, мне, по крайней мере, о таких программах слышать не приходилось.

---

[2] Сташин В. В. и др. Проектирование цифровых устройств на однокристальных микроконтроллерах. — М.: Энергоатомиздат, 1990.

"Большое" программирование — индустрия, и без вот таких кодеров-пролетариев обойтись не в состоянии. Но как существование индустрии звукозаписи не отменяет музыкальное творчество, так и производство коммерческого программного обеспечения не противоречит существованию программиста-творца. На западе таких обычно, не особенно разбираясь, именуют "хакерами", но как мы увидим позже, это не совсем точное определение. Но дело не в терминах — такой программист вполне может добиться успеха и признания не будучи коммерсантом: свидетельств тому множество. Главному разработчику нашумевшего браузера FireFox Блейку Россу было всего 19 лет, когда версию 1.0 этого достойного конкурента Internet Explorer скачали за первый же месяц 5 миллионов человек. Большинство вошедших в обиход программных новинок последнего времени сделаны молодыми людьми в том же возрасте или чуть постарше, причем отнюдь не в рамках плановых разработок крупных софтверных компаний. Файлообменные сети Шона Фэннинга, сервис LiveJournal Брэда Фитцпатрика, проигрыватель WinAmp Джастина Франкеля, поисковая система Google Сергея Брина и Ларри Пейджа — список можно продолжать и продолжать. Придется сделать вывод, что время "компьютерных гениев" и "гаражных стартапов" (start up) отнюдь не прошло. Кстати, один из самых успешных отечественных программных продуктов — распознаватель текстов FineReader — также был создан под руководством совсем молодого тогда физика Давида Яна. Кто знает, может ваше имя недолгое время спустя будет звучать в этом ряду?

Но занятие под названием "программирование" — отнюдь не прерогатива только молодого поколения, с пеленок привыкшего держаться за мышку. Существует достаточное количество профессий, в которых молодые имеют преимущество благодаря своей высокой активности и стремлению сделать карьеру, и, наоборот, есть области деятельности, куда молодым до поры до времени дорога заказана. Программирование — как раз то занятие, где возраст не имеет никаких преимуществ, ни в ту, ни в другую сторону — было бы желание. Молодежь легко и быстро осваивает новое, зато человек постарше берет свое обстоятельностью и стремлением докопаться до глубин.

Единственное, что нужно заметить по этому поводу — особенности отечественного технического образования и, главное, содержание деятельности советского инженера были таковы, что он привык все делать сам, и даже "доводить напильником" то, что не было доделано разработчиками. Это с одной стороны хорошо — кругозор намного шире, и такой человек в отдельных случаях может создавать много лучше работающие программы, чем обычный среднестатистический западный специалист. Но в подавляющем большинстве случаев стремление сделать все самому от начала до конца, не затрудняя себя изучением того, что уже сделано до тебя, может только навредить — современные ОС чаще всего этого просто не позволят и правильно сделают. А почему — вы узнаете уже из первой главы этой книги.

Что можно найти в книге?

Предполагается, что читатель уже обладает некоторыми навыками создания приложений в среде Delphi и хочет расширить функциональность своих программ, придать им удобный интерфейс, законченность и профессиональные скоростные качества. Просто поместить на форму компонент Delphi, к примеру, `MainMenu` и написать обработчик щелчка мыши для пункта **Файл | Открыть** может каждый после десятиминутного ознакомления с соответствующим разделом в учебнике. Поэтому мы посвятим одну главу созданию проекта "с нуля", а в дальнейшем сосредоточимся на реализации функций, которые вы часто встречаете в различных программах, но приемы программирования для них в учебниках по Delphi не описаны вообще или вызывают трудности при практической реализации. Некоторые из таких функций могут служить просто для красоты или удобства, другие необходимы, если вы пишете программу не на один раз, а собираетесь ею часто пользоваться и тем паче распространять. Как предотвратить повторный запуск программы? Что такое потоковое чтение файлов и зачем оно нужно? Как правильно написать программу без окна, в виде иконки? Как перехватить нажатие клавиш? Как создать инсталляционный пакет? На подобные простые и не очень простые вопросы мы постараемся ответить в дальнейшем. Причем не удивляйтесь, если вы встретите подробное обсуждение некоей проблемы, которая может показаться совершенно второстепенной, скажем, как убрать с глаз долой текстовый курсор в компонентах-редакторах — хорошо сделанная программа отличается от плохо сделанной в первую очередь подобными мелочами.

В процессе создания программ мы постараемся избежать недозволенных приемов и недокументированных возможностей. В Windows 9*x* вам в принципе никто не запрещает, например, печатать документ прямым обращением к LPT-порту или использовать прямое обращение к регистрам последовательного порта. Но разработчики из MS не рекомендуют этого делать, и поступают совершенно правильно — с нашей точки зрения, хотя бы потому, что при переходе к семейству NT такие программы окажутся совершенно неработоспособными и их придется заново переписывать. В случае же, если мы все делаем, как рекомендовано инструкцией, мы, по крайней мере, снимаем с себя ответственность за происходящее. Это, конечно, шутка — с точки зрения пользователя, программы должны работать, а кто за это несет ответственность — дело десятое. И мы увидим, что при всех наших стараниях, нам все же не удастся следовать рекомендациям разработчиков в полной мере.

Кроме этого, я постараюсь, чтобы при использовании в своих программах примеров из этой книги вам не потребовалось бы использовать ничего, выходящего за рамки свежеустановленной Delphi. За некоторыми отдельными ис-

ключениями — это касается, во-первых, использования готового компонента AsyncFree для работы с COM-портом (но будет также и подробно рассказано, как все сделать без него), во-вторых, я для сокращения записи буду часто пользоваться собственной процедурой преобразования числа в HEX-форму (также будет показано, как ее можно легко заменить стандартными средствами).

В книге вы нередко встретите изложение личных предпочтений автора и его взглядов на ту или иную проблему, включая некоторые особенности Windows вообще. По мнению автора, в Windows есть много неоправданно сложных и просто ошибочных механизмов, для которых можно и нужно обсуждать целесообразность и границы их применимости. Сам автор отчетливо понимает, что его воззрения не есть истина в последней инстанции, и относится к этому в стиле "это мое мнение, и я его разделяю", но старается предоставить читателю самому судить о том, насколько автор прав в тех или иных случаях. В частности, реконструкция логики разработчиков языковых интерфейсов Windows *(см. главу 8)* представляет именно такой случай — вполне возможно, что среди профессионалов найдутся люди, которые считают эту систему стройной и логичной, находя в ней положительные стороны, ускользнувшие от автора этих строк. Прошу читателя иметь в виду, что дискутировать с автором в подобных вопросах совершенно не возбраняется.

Следуя за автором в его критике отдельных сторон Windows, следует иметь в виду одно обстоятельство — как и в любом другом деле, голоса критикующих здесь всегда звучат громче. В защите Windows не нуждается — ее доминирующая роль на рынке ПК сама по себе есть надежная защита. Однако справедливости ради следует отметить, что в продуктах Microsoft есть много положительных качеств — это и реализации некоторых функций Windows (например, буфера обмена), очень неплохая и изначально весьма продвинутая система работы с таблицами в Word, удобная работа с макросами в Office (кажется, вообще не имеющая аналогов), и т. п. Большинство же недостатков Windows (исключая откровенные недоработки) проистекает из того факта, что эту систему создавали и продвигали на рынок в условиях, весьма далеких от тепличных, в которых создавалась и продвигалась, например, MacOS. Обеспечить беспроблемную работоспособность ОС на миллионах *различных* конфигураций "железа" от неизвестных производителей — задача в принципе не решаемая, и, надо сказать, в Microsoft с ней справились не так уж и плохо. Это доказывает тот факт, что Linux при установке создает на порядок больше проблем (пока?), и с этим спорить трудно. Главный недостаток Windows — ее негибкость и ориентация на некоего "среднего" пользователя (так, как его себе представляют в Microsoft) — есть продолжение этих достоинств. Не надо забывать и другое — если бы не Windows, собиравшая и аккумулировавшая в себе все чужие передовые идеи, иногда искажая их до неузнаваемости,

но всегда доводя до состояния, относительно пригодного к использованию даже последними "чайниками", никакой компьютерной революции бы не произошло, компьютер так и остался бы дорогой игрушкой для инженеров и бизнесменов. Мало кто помнит, что правило, согласно которому последняя модель PC всегда стоит $1500, стало соблюдаться только с началом эпохи Windows. Первый IBM PC (с весьма ограниченными возможностями) был выпущен на рынок по запредельной для 1981 года цене примерно $3000, а продвинутая модель IBM PC AT (на основе 286-го процессора) в середине 80-х стоила порядка $5000—7000.

Есть и еще несколько обстоятельств, во многом извиняющих разработчиков из MS. Например, подавляющее большинство современных программ вполне может выполняться на "железе" трех-, пятилетней давности, и уже почти забыто, что еще лет пять-шесть назад в балансе аппаратных и программных средств аппаратные средства решительно отставали — очередные версии ОС требовали последних и достаточно дорогих конфигураций компьютера. Так как никому не хотелось тратить лишние деньги, был период, когда в эксплуатации одновременно находился почти весь парк компьютерного "железа", начиная с клонов IBM PC XT и до первых моделей Pentium MMX. Поэтому гораздо более остро стоял вопрос совместимости софта — пересаживаясь за новые компьютеры, пользователь хотел более комфортабельной работы, но вовсе не желал расставаться с любимыми Norton Commander и Lexicon (автор своими глазами наблюдал этот мучительный процесс). И в этом вопросе корпорации Microsoft, которая не стала ломать ситуацию "через колено", можно сказать только большое-большое спасибо. Но эти требования совместимости всего и вся имели и другую сторону — первые версии Windows оказались весьма далеки от идеала, который обычно обозначается словами "многозадачная операционная система".

Некоторые подробности о Windows и Delphi будут излагаться отдельно, в виде "заметок на полях", но хочется еще раз подчеркнуть, что книга эта ни в коем случае не призвана заменить учебник программирования, справку по Windows API или по работе в Delphi. Вам как минимум потребуется учебник-справочник по языку Object Pascal, приемам работы в Delphi IDE (Integrated Development Environment, интегрированная среда разработки) и основным компонентам. Некоторые подобные издания, а также ссылки на отдельные удачные, по субъективному мнению автора, интернет-ресурсы приведены в списке литературы. Отмечу, что и печатных пособий, и тем более сайтов, посвященных Delphi, так много, что никакой подобный список не может претендовать на полноту, так что эта задача даже и не ставилась. В качестве основного источника сведений по Windows API вполне достаточно официального сайта Microsoft MSDN [14] или даже встроенной в Delphi справки (файл win32.hlp, который располагается в папке C:\Program Files\Common Files

\Borland Shared\MSHelp\; может быть вызван и через меню **Пуск | Программы | Borland Delphi 7 | Help | MS SDK Help files | Win32 Programmer's Reference** — правда, в некоторых отношениях эта справка устарела). Есть и множество русскоязычных ресурсов, той или иной степени полноты, представляющих собой простой или комментированный перевод материалов с сайта Microsoft MSDN. Наиболее полные и систематизированные переводы принадлежат Владимиру Соковикову [16,18], другие вы без труда отыщете сами, просто набрав в Яндексе название той или иной функции API.

Знания и умения

Есть знания и умения. Обучение чтением теоретических курсов — приобретение знаний, как таковых — изобретение нового времени и составляет основное содержание классной системы образования, возникшей в период Реформации. До этого обучение происходило исключительно на личных примерах и в процессе реальной работы. И сейчас в тех областях, где закончивший некий курс обучения должен выйти в мир, обладая именно практическими умениями (как, например, в медицине), основное содержание обучения составляет практика. А есть области, в которых теоретических знаний не требуется вообще или нужно настолько мало, что без специального изучения теории можно обойтись — например, вождение автомобиля или, скажем, плотницкие работы. С другой стороны, такие области, как теоретическая физика или математика, наоборот, в значительной степени основаны именно на знаниях (хотя тоже далеко не полностью). Любопытно, что программирование, при том, что оно опирается на самые абстрактные области человеческого знания, по сути своей — типичное умение, ремесло. Можно спокойно начинать программировать, обладая математическим багажом в пределах средней школы — необходимые знания приобретаются в процессе отработки умений. А умения приобретаются только на практике.

Автора этой книги часто спрашивали: "Как можно научиться работать на персональном компьютере?" Ответ на этот вопрос единственный: надо его приобрести. Остальное сделает ваше желание научиться — если оно имеется, конечно. Точно так же и навыки программирования нельзя приобрести при чтении даже самых умных книжек. Включите компьютер, подберите себе кресло поудобнее и начинайте работать. Только если вы хотите стать Настоящим Программистом, не забывайте, что это очень особая профессия. Сама специфика деятельности программиста такова, что здесь не могут удержаться люди, привыкшие к делению жизни на работу с 9 до 17 и развлечения все остальное время. Этой особенности профессии посвящена добрая половина программистского фольклора:

У жены программиста спросили:
— А как он за тобой ухаживал?
Жена, после минутного раздумья:
— Компьютер показал...

Но, чтобы писать вполне работоспособные программы, Настоящим Программистом становиться вовсе необязательно. В этой книге я ориентировался на тех, кто делает программы, предназначенные для практической работы, а не для развлечения, постаравшись сосредоточиться на том, чтобы все "навороты" служили одной цели: сделать программу удобной и приятной для пользователя. Впрочем, грань между серьезным и развлекательным провести трудно. Никому сами по себе не нужные "приколы", несомненно, иногда неплохо помогают иллюстрировать тот или иной прием. Да и одно из основных наших приложений — SlideShow, которое мы будем создавать на протяжении почти всей книги, по своему назначению есть программа в основном развлекательная.

Кто такие хакеры?

Несколько слов об упомянутом ранее понятии "хакер". В компьютерной прессе не устают подчеркивать, что настоящий хакер — это не преступник, который рыщет по Интернету в поисках способов взломать сайт Пентагона или своровать базу кредитных карт клиентов онлайнового аукциона eBay. Точнее, часть таких преступников-"крякеров" — и любителей и профессионалов, занимающихся промышленным шпионажем — может относиться к хакерам, но уже обратное неверно: далеко не все хакеры, и даже не заметная их часть — преступники. Однако, когда доходит до объяснений, а кто же собственно такой хакер, дело обычно ограничивается следующей характеристикой: это человек, помешанный на компьютерах, желающий досконально знать, как все в них устроено, который отличается виртуозным владением соответствующими инструментами, позволяющими ему выделывать с компьютерами и сетями разные необычные штуки. Между тем, это определение является недостаточным: кроме виртуозного владения компьютером, принадлежность к хакерскому течению предполагает еще и определенный склад ума и особую социальную философию левацкого толка. Развитие хакерской субкультуры связано с Массачусетским технологическим институтом и появлением в 1961 году компьютеров PDP-1. Позднее, с появлением первой сети ARPAnet и, в дальнейшем, Internet (Интернет), хакерство интернационализировалось. Главными практическими результатами деятельности представителей хакерской субкультуры для всего остального мира стало рождение ОС Unix и языка C, а позднее сообщества "свободного софта" и соответствующего мировоз-

зрения. Характерной особенностью Unix (как и ее знаменитого клона — Linux) было то, что она была сделана полностью в рамках частной инициативы, без какого-то заказа — в дальнейшем это стало основным отправным пунктом идеологии "свободного софта" вообще, даже когда подобные проекты делаются в коммерческих целях, дух все равно тот же: "потому что интересно".

Основным в хакерстве является именно вот это либертарианское, отчасти анархистское мировоззрение, совпавшее по времени возникновения с расцветом движения хиппи со всеми сопутствующими атрибутами — цветочками в волосах, драными джинсами и способностью работать когда хочется, а не когда этим занимаются все остальные. Хакеры — это технологические хиппи, те из них, кто обрел, как ему показалось, именно в компьютерах желанную свободу от этого кошмарного мира "власти чистогана". Хакерское движение, как и молодежные бунты 60-х вообще, вне зависимости от практических результатов, сделало очень много хорошего. Прежде всего, в политическом смысле они стали действительно реальной оппозицией консервативной бизнес-верхушке — заняли то место, которое ранее безуспешно пытались занять коммунисты-догматики. У них это получилось потому, что, в отличие от политиков, они не болтали и не устраивали революций, а делали реальные дела ("Не пишите манифесты, пишите код!"), и чуть ли не впервые в истории показали всем, что, оказывается, серьезные вещи можно делать и так — без корпоративных структур, без формального подчинения, на основе добровольности и открытости. Современный компьютерный — и не только компьютерный — мир очень и очень многое у них заимствовал, как в плане методологии ведения разработок, так и в плане конкретных идей.

Эту небольшую политинформацию я провел вот зачем: хакер — совсем не синоним программиста-виртуоза, хакерство — течение социальное, а не профессиональное, поэтому термин этот без нужды примерять к себе не следует. Далеко не все профессиональные программисты и даже взломщики программ называют себя хакерами и являются ими. Вообще большинство из тех, кто профессионально "ломает" программы и средства защиты, являются серьезными учеными-криптографами, обладателями всяких академических званий и степеней, или, по крайней мере, составили себе имя публикациями на соответствующие темы. И взломом они занимаются не просто так, а потому что без их труда невозможно правильно оценить эффективность работы тех или иных алгоритмов. Согласитесь, в этом подходе есть некоторые отличия от очередных "подвигов" взломщиков сайтов, по какой-то странной причине называющих себя "хакерскими группами". Тем более, совсем не каждый, кто считает, что хорошо владеет компьютером, вправе считать себя хакером. Добавлю специально для тех, кто усматривает некую романтику в образе "крякера"-подпольщика: все громкие деяния, типа кражи исходного кода

Windows или остроумно написанного вируса, заразившего очередной миллион компьютеров, относятся к настоящему хакерству, примерно так же, как поведение водителя-лихача к манере вождения шофера-профессионала.

Как пользоваться книгой

Но вернемся к тому, с чего начали — играть лучше в практически полезные дела. Исходя из этих соображений, книга эта построена так, чтобы каждый пример, по возможности, являлся неким законченным проектом (или его частью), который может пригодиться на практике. Некоторые примеры (но не все, конечно) будут даны в развитии — вы увидите, как приложение постепенно отлаживается, с промежуточными экспериментами по использованию того или иного приема или функции. Все примеры (по главам) имеются на прилагаемом диске в том виде, в котором они описаны в тексте. Эти примеры в идеале нужно изучать следующим образом — вы берете исходный текст примера, такой, какой он есть к началу соответствующего раздела, и повторяете все, что описано в этом разделе по ходу мысли автора. А потом сравниваете то, что получилось, с окончательным вариантом, который имеется на диске. Если варианты совпали — замечательно, если нет — при сравнении легко установить, что именно (и у кого именно) не так. Все примеры тестировались на совместимость с Windows 98 и Windows XP (в последнем случае без установленных сервис-паков). При обнаружении каких-то особенностей в работе под указанными ОС, это оговаривается в тексте. Отразить в оглавлении все имеющиеся в книге приемы возможности, конечно, не было, поэтому в конце книги размещен предметный указатель в форме FAQ, где перечислены по возможности все имеющиеся в книге ответы на вопросы "Как?".

Запускать проекты прямо с диска неудобно, вам придется долго настраивать среду и вы не сможете вносить изменения, поэтому их нужно сначала перенести на жесткий диск вашего компьютера. При переносе, возможно, придется изменить путь к папке с проектом, который указан в исходном DSK-файле. Если вы устанавливали Delphi в режиме по умолчанию и папка Program Files находится на диске C:, то можно ничего не менять — просто скопируйте папку, относящуюся к данной главе, в папку Projects. В этом случае путь будет такой: C:\Program Files\Borland\Delphi7\Projects, и он зафиксирован в DSK-файлах проектов на диске. В противном случае можно вовсе не копировать DSK-файл в новую папку, но при этом вы также потеряете и все настройки. Лично я открываю DSK-файл в Блокноте и вношу исправления через пункт **Заменить | Заменить все**. Перенос можно осуществить и раздельным копированием через пункт **Save as** сначала проекта, потом каждого модуля программы по очереди.

Перед тем как приступить к делу, автор считает себя обязанным выразить благодарности:

- Валерию Васильевичу Фаронову — за его замечательные учебники по Pascal и Delphi;
- Евгению Сергеевичу Голомину, глубоко верующему человеку, за выложенные им для всеобщего ознакомления исходные тексты к прекрасной программе "Опечатка" (**http://come.to/golomin**);
- всем без исключения посетителям форумов по программированию в Рунете — без них эта книга вообще бы не увидела света.

Со всеми вопросами и пожеланиями пишите на **revich@homepc.ru**.

ГЛАВА 1

Ликбез

Некоторые сведения о программировании, Windows и Delphi

> Создайте систему, которой сможет пользоваться каждый дурак, и только дурак захочет ею пользоваться.
>
> *Принцип Шоу*

> Компьютерная программа выполняет то, что вы приказали ей делать, а не то, что вы бы хотели, чтобы она делала.
>
> *Третий закон Грида*

Современный персональный компьютер — устройство необычайно сложное. Причем рядовой ПК образца 2004 года отличается от IBM PC двадцатилетней давности гораздо больше, чем, к примеру, последняя модификация Ford Focus от легендарного Ford T образца 1913 года. И дело тут не в самой по себе скорости работы, которая возросла примерно на три порядка. Гораздо важнее принципиально возросшая функциональность — ни о какой 3D- и даже обычной многоцветной 2D-графике тогда и речи не шло, голосовые интерфейсы существовали разве что в фантастических романах, а предсказать нечто подобное Интернету, да еще и мобильному, фактически не смог никто. Если продолжить аналогию с первыми автомобилями, то современный ПК в сравнении с IBM PC скорее следует уподобить, если и не реактивному истребителю, то, по крайней мере, пассажирскому лайнеру.

Казалось бы, управлять такой сложной машиной — учиться и учиться. Но ПК не обосновались бы настолько прочно в наших домах и офисах, если бы разработчики не придумали программные средства, сводящие сверхсложные операции к двум-трем щелчкам мыши. Причем на сегодняшний момент сложилась довольно парадоксальная ситуация: разнообразие компьютерного "железа" настолько велико, что *правильно* подобрать комплектующие и *правильно* настроить современный ПК невозможно без определенного багажа специальных знаний. А вот технологии программирования ПК, наоборот, предельно упростились. Правда, это в полной мере справедливо только для

случая создания пользовательских программ — системные программы, разумеется, требуют для своего создания специальных знаний. Показательно, что в популярной книге С. В. Зубкова "Assembler для DOS, Windows и Unix" [13], выдержавшей несколько переизданий, программированию под Windows и Unix посвящено менее 100 страниц из почти 700 — настолько это простое по сути занятие.

Заметки на полях

Попробую пояснить эту парадоксальную мысль. Для начала хочу подчеркнуть разницу между понятиями "сложный" и "громоздкий" — очень простая по сути программа может быть очень громоздкой и наоборот. Например, расшифровать заголовок файла, содержащего изображение в формате BMP, и воспроизвести на экране содержащуюся в нем картинку — достаточно сложное занятие. Программа, которая это делает из-под DOS, окажется и сложной и достаточно громоздкой, учитывая еще тот факт, что для воспроизведения True Color придется где-то искать и присоединять к программе драйвер имеющейся в наличии видеокарты. Излишне говорить, что в силу большого разнообразия "железа" последняя задача может доставить и программисту и пользователю массу хлопот. А вот в Windows ни один из этих вопросов не стоит вообще — все драйверы уже установлены заранее, а для вывода на экран картинки формата BMP не требуется даже знать, что у нее есть какой-то там заголовок — даже на ассемблере это делается вызовом одной-двух функций. Вместе с тем, если простейшая ассемблерная программа под DOS может состоять из одной-единственной команды процессора, то любая программа под Windows обязана содержать некий минимум команд, так что решение упомянутой ранее задачи отображения картинки по объему кода может даже превысить DOS-вариант. Что делает Windows-программы более громоздкими, но отнюдь не более сложными, потому что этот необходимый минимум повторяется из программы в программу и заново его "изобретать" каждый раз не требуется (см. [13]).

Так получилось потому, что большинство функций в современных операционных системах скрыто за оболочкой, носящей название *пользовательские программные интерфейсы* (Application Programming Interface, API). Собственно процесс программирования сводится к тому, чтобы вовремя и правильно вызвать нужную функцию API. Поэтому такие крайне сложные для "обычного" программирования операции, как, к примеру, вывод на экран полноцветной графики или создание текстового редактора с поддержкой форматирования, шрифтов, операций с буфером обмена и прочих необходимых свойств, сводится к вызову нескольких функций API. Довольно большой прогресс по сравнению с прерываниями DOS, которые программисты в большинстве случаев старались обойти, не видя необходимости в лишних прослойках между программой и BIOS или "железом", не так ли?

Однако на этом разработчики современных систем не остановились. С помощью средств так называемого *визуального программирования* процесс вызова функций API максимально автоматизирован. Вам уже не нужно выполнять рутинную работу по написанию кода программы целиком. Достаточно

перетащить мышью нужный компонент на форму, и соответствующий код включается в текст программы автоматически. В результате процесс программирования стал воистину творческим занятием: всю "грязную" работу берет на себя компьютер (точнее, выбранный пакет программирования), а вам остается только правильно обработать возникающие в программе события. Специальных знаний об устройстве компьютера и построении ОС здесь требуется не больше, чем для обычного квалифицированного пользователя.

Разумеется, как в любом другом деле, за удобства приходится платить: вы полностью привязаны к существующим программным интерфейсам, которые писали для вас добрые дяди из Microsoft и других фирм-производителей системного софта, и шаг вправо-влево тут означает если и не расстрел, то, во всяком случае, необходимость приобретения *очень специальных* знаний. Однако такой выход за пределы стандартных функций в подавляющем большинстве случаев и не требуется: "добрые дяди" постарались предусмотреть по максимуму все, что требуется среднестатистическому пользователю. Другое дело, что поиск нужной функции и способа ее правильного использования может быть очень непростым занятием — ведь далеко не все можно сделать, перетаскивая компоненты мышью на форму. Именно в этом деле и призвана помочь книга, которую вы держите в руках.

О Delphi и Windows

Кстати, а почему именно Delphi? В принципе все современные пакеты визуального программирования позволяют делать одно и то же, а если какие-то вещи делать удобнее в одном пакете, а другие — в другом, то никто не мешает использовать их совместно. Скажем, библиотеки VCL (Visual Components Library) от Borland являются общими для Visual C++ и Delphi и написаны в основном на Object Pascal, а Windows API, наоборот, большей частью написаны на C (или C++), что не мешает использовать их в любой среде. Но Delphi, безусловно, является на сегодняшний день наиболее универсальной средой программирования, которая позволяет без лишних сложностей создавать как самые простые пользовательские программы, так и навороченные профессиональные пакеты.

Эта книга написана с расчетом на выполнение примеров в среде Delphi 7.0. Седьмая версия — последний релиз Delphi для платформы Win32, на которой основываются все версии Windows от 95-й до XP. В настоящее время Microsoft переходит на платформу .NET — на ней будет полностью основана Windows Longhorn, которую планируется выпустить на рынок в 2006 году. Частично на .NET переходит уже обновленная 64-разрядная Windows XP, готовящаяся к выходу в 2005 году. Между прочим, базовый язык программирования для платформы .NET под названием C# разрабатывает Андерс

Хейлсберг — человек, которому корпорация Borland во многом обязана самим своим существованием.

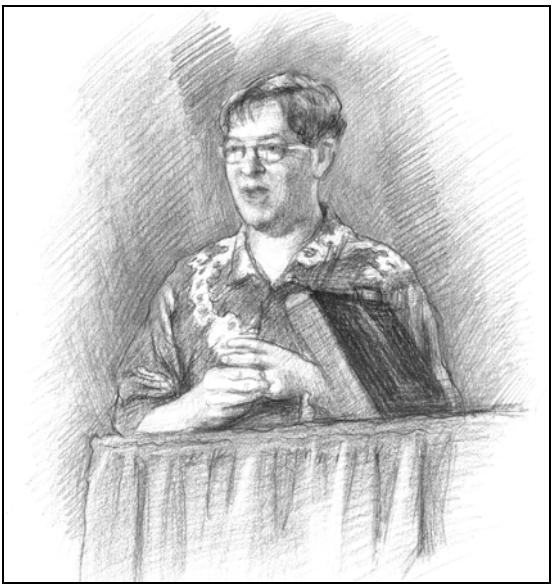

Рис. 1.1. Андерс Хейлсберг (рисунок Александры Дрофиной)

Андерс Хейлсберг (Anders Heilsberg) — создатель Turbo Pascal и один из главных архитекторов Borland Delphi с момента ее возникновения, человек, разработавший продукты, выведшие корпорацию Borland в ряды ведущих поставщиков программного обеспечения. Также главный архитектор языка C# для платформы Microsoft .NET. О себе Андерс рассказывает так (из интервью журналу "Домашний компьютер", #1, 2004):

Я родился в 1960 году в Копенгагене, Дания. Свое тихое и милое детство я провел в пригороде Копенгагена. Потом учился на инженера по электротехнике в Техническом университете Дании. В 1987 году переехал в США, а в 1994 — женился. Живу в Сиэтле. С 1983 по 1996 я работал в компании Borland, а теперь в Microsoft. В 1979 я основал компьютерную компанию в Дании под названием PolyData. Это было время, когда персональных компьютеров еще не существовало. Мы продавали компьютерные комплексы и писали для них программное обеспечение. Я написал такие вещи, как ассемблер, дизассемблер, небольшую операционную систему и несколько расширений для Microsoft ROM-Basic. Моим самым первым большим проектом стал компилятор с языка Pascal и редактор, который мог заменить ROM-Basic. После этого я написал еще одну реализацию Pascal для операционной системы CP/M. Она называлась PolyPascal. В 1983 году мы объединились с

ребятами, которые только что основали компанию Borland, они лицензировали наш компилятор Pascal, добавили туда свой собственный редактор и назвали все это Turbo Pascal. Я помню, как думал, что они сумасшедшие: эти парни продавали новый продукт по цене 49 долларов 95 центов, в то время как он стоил 500 долларов! Но достаточно быстро выяснилось, что я ошибался — Turbo Pascal стал очень популярным. Мы продали его столько, что в начале было невозможно представить.

Для того чтобы читатель немного сориентировался, приведем краткую сравнительную характеристику различных сред обработки от Borland, базирующихся на языке Pascal.

В начале всего был Turbo Pascal 1.0, вышедший на рынок 20 ноября 1982 г. Об искусстве Андерса Хейлсберга может говорить тот факт, что интегрированная среда разработки, встроенный редактор и библиотека времени выполнения умещались в файле turbo.com размером 33 280 байт. Правда, эта версия сама могла делать только COM-программы (если кто помнит, то в DOS существовал такой формат исполняемых файлов, отличавшийся от привычного EXE тем, что занимал только один 64-килобайтный сегмент памяти), но зато работала она на медленных ПК того времени очень быстро и давала очень компактный код, благодаря чему сразу вывела компанию Borland в лидеры. В дальнейшем эволюция Turbo Pascal привела к версии 7.0 (1992) — до сих пор популярному средству разработки программ под DOS. Благодаря непревзойденной по удобству среде разработчика (Integrated Development Environment, IDE) и простому для освоения языку, Turbo Pascal стал особенно популярным в непрофессиональной и полупрофессиональной среде. Строго говоря, последняя версия Turbo Pascal, как такового, называлась 6.0, седьмая версия называлась Borland Pascal и включала в себя две разных среды: Borland Pascal for DOS 7.0 (базирующийся на шестой версии с некоторыми существенными доработками) и Borland Pascal for Windows. Если вам удастся достать какое-нибудь пособие по последнему (например, раритетную книгу В. В. Фаронова "Паскаль и Windows"), то вы увидите, что разработка приложений под Windows "вручную" — довольно громоздкое занятие, требующее хорошего знания приемов объектно-ориентированного программирования (ООП). Чтобы облегчить это занятие, в Borland (под руководством все того же Хейлсберга) к 1995 году была разработана Delphi 1.0 — первая визуальная среда программирования, ориентированная на разработку 16-разрядных приложений под Windows 3*x*.

Начиная же с версии 2.0, Delphi ориентировалась на 32-разрядные версии Windows (95 и выше), использующие платформу Win32. Все использующиеся на практике на момент выхода этой книги версии Windows ориентируются

именно на эту платформу, поэтому в подавляющем большинстве случаев вам вообще не нужно думать о том, какая версия Windows у вас установлена. И все же многие API могут быть специфичными для того или иного семейства ОС либо той или иной версии. Поэтому работу программ, особенно тех, которые предназначены для распространения, следует проверять под различными версиями. Для этого удобно установить на компьютере сразу несколько систем, благо XP это позволяет без лишних сложностей.

На момент написания данной книги уже имеются версии Delphi 8.0 и Delphi 2005, ориентированные на .NET. Для совместимости в комплект поставки Delphi 8.0 входит Delphi 7.1, подобно тому, как в Borland Pascal входил Turbo Pascal, а Delphi 2 комплектовалась версией 1. Разница только в том, что вторая версия Delphi в свое время вышла с опозданием, уже после выхода Windows 95, а восьмая версия отчасти опережает события. Заметим, что наименование "Delphi 8.0" следует признать неудачным — это другой продукт для другой платформы (изменили же когда-то марку Turbo Pascal на Borland Pascal — почему бы и не следовать этому принципу и в дальнейшем?). Поэтому большинство примеров, которые вы встретите в этой книге, в среде Delphi 8.0 (и выше), скорее всего, просто не будут компилироваться — но сами по себе готовые программы будут пригодны еще долгие и долгие годы, в будущей Longhorn декларирована совместимость с Win32 (подобно тому, как версии Win95/98/ME были совместимы с DOS-программами).

Что же касается более ранних версий, то большинство примеров из этой книги совместимы с версиями Delphi, начиная с 4.0, в крайнем случае 6.0. При установке следует позаботиться, чтобы у вас оказались установленными файлы справки по Win32. Эта справка, к сожалению, практически не обновлялась с версии 3.0, так что можно просто скопировать файл win32.hlp в отдельный каталог и пользоваться им автономно, независимо от версии пакета.

Примечание специально для тех, кто только осваивает программирование: на пиратских развалах якобы появилась русифицированная версия Delphi. Ее устанавливать *не следует*, и не только потому, что она неизвестного происхождения, но и потому, что все без исключения пособия и интернет-ресурсы оперируют с английской версией, и в результате вы потратите гораздо больше времени на обратный перевод русских названий пунктов меню и текстов сообщений, чем на то, чтобы один раз выучить английские.

> Несколько слов о том, каковы особенности работы программ под теми или иными версиями Windows. Главное отличие семейства 9*x* (95/98/ME) от семейства NT (NT/2000/XP) заключается в реализации многозадачности. В забытых ныне 16-разрядных версиях Windows 3.*x* многозадачность называлась кооперативной — фактически весь процесс работы заключается в выполнении последовательного ряда событий Windows, и пока одно событие

не обработается, все остальные вынуждены ждать своей очереди. Таким образом, малейшая ошибка в одном приложении полностью подвешивает всю систему. Все программы, в том числе служебные, видимы друг для друга, модуль, содержащий ошибки обращения к памяти, может легко испортить ее содержимое, принадлежащее другому процессу.

В версиях семейства NT реализована настоящая — вытесняющая — многозадачность (напомним, что NT появилась раньше, чем 95-я). Такая система распределяет процессорное время и память между процессами (process) и потоками (thread)[1], как будто бы каждый из них выполняется в отдельном компьютере, поэтому "зависшая" программа теоретически не оказывает влияния на все остальные. Но за это приходится платить — в первую очередь тем, что прямой доступ к "железу" практически исключен, а если вы попытаетесь этот запрет обойти тем или иным способом, то такая попытка ничем хорошим ни для вашей программы, ни для системы не кончится. То есть требования к качеству программного кода сильно повышаются и можно смело утверждать, что все случаи "обрушения" Windows XP связаны именно с неправильно (как говорят программисты — "криво") написанными приложениями.

А в версиях 9*x* в целях совместимости с 16-разрядными приложениями (включая DOS-программы) реализовано что-то вроде промежуточного варианта. Все 32-разрядные прикладные программы выполняются в соответствии с моделью вытесняющей многозадачности. На уровне таких приложений формально все работает независимо, как и NT. Однако в области модели памяти (в основном из-за требований совместимости с 16-разрядными приложениями DOS и Win3*x*) есть серьезные дыры. Как и в 3*x*, ничто не может помешать программе, содержащей ошибку обращения к памяти, произвести запись в адреса, принадлежащие системным DLL, и вызвать крах всей системы.

В Windows 9*x* ситуация, когда выполняющийся поток все никак не освободит ресурсы компьютера, случается не так уж редко. Даже формально правильное приложение может быть источником неприятностей: так, многим знакома картинка, когда ресурсоемкая программа (архиватор, файловый менеджер при копировании больших объемов информации, или, например, поисковая программа) настолько "оккупирует" все ресурсы компьютера, что даже картинка рабочего окна не успевает прорисовываться. Один из механизмов таких задержек — но не единственный, конечно — связан с тем, что многие подобные процессы выполняются через 16-разрядные функции, а

[1] Сами по себе процессы ничего не делают, а лишь предоставляют ресурсы и контекст для выполнения потоков. Любой процесс должен иметь, по крайней мере, один поток, который и выполняет код процесса. Причем планирование переключения задач в Windows осуществляется на уровне потоков, а не процессов.

только один поток может обращаться к 16-разрядным DLL в каждый момент времени, потенциально затормаживая другие процессы, которым нужен к ним доступ. Другой — с неправильным распределением приоритетов одновременно выполняющихся потоков. Бороться с такими неприятностями можно только внутри самой программы, периодически искусственно вызывая обработчик системных сообщений, но самое правильное — по возможности не использовать потенциально "тормозных" процедур вообще.

О пользовательских интерфейсах компьютерных программ

Крупнейший в мире специалист по компьютерным интерфейсам Джеф Раскин (Jef Raskin, 1943—2005 гг.) говорил примерно так (за точность цитаты не ручаюсь): *"Пользователь обычно не понимает, насколько ему неудобно, механизм обратной связи от потребителя к производителю не работает, и единственное, что может заставить проектировщика создавать по-настоящему хороший интерфейс — это его совесть"*. Я не могу тут не остановиться на истории пользовательских интерфейсов, потому что непонимание, насколько неудобно работать с современными программами, к сожалению, характерно для программистов в еще большей степени, чем для пользователей. Невозможно делать удобные программы, если вы не очень хорошо представляете, с чем вам придется иметь дело.

Точкой роста почти всех идей, реализованных впоследствии в так называемом графическом интерфейсе пользователя (Graphic User Interface, GUI), стал Xerox Palo Alto Research Center (Xerox PARC), возникший на рубеже 60—70 годов. Любопытно, что одним из его основателей, а с сентября 1970 года — и руководителей, стал Боб Тейлор (Robert W. Taylor), который пришел в Xerox из DARPA, где в 1966—69 гг. возглавлял проект ARPAnet, предшественника Интернета. Одной из идей, родившихся в этом центре, была так называемая парадигма WIMP (Windows, Icons, Menus, Point-and-Click — "окна, пиктограммы, меню, укажи и щелкни"), которая переросла позже в концепцию GUI и продолжает эксплуатироваться в настоящее время. Эти разработки связывают с именем Алана Кея (Alan Key), также известного, как автора SmallTalk (первого объектно-ориентированного языка программирования). В 1975 году в Xerox была начата разработка нового проекта, закончившегося в апреле 1981 года представлением системы Xerox 8010 Professional Workstation, более известной под торговой маркой Xerox Star. Именно с нее и началось победное шествие GUI, внедрение которого, как промышленного стандарта, было произведено в 1988 году усилиями Sun Microsystems, Xerox и AT&T. Xerox Star мы также обязаны такими вещами, как иконки, окна, меню, двухбайтовые многоязычные шрифты (современный Unicode — см. *главу 8*), режим

WYSIWYG и др. За подробностями я отсылаю читателя к [25], а здесь нам важно, что распущенный после рыночного провала Xerox Star примерно в 1983 году коллектив ее разработчиков в основном оказался в Apple — за важными исключениями, о которых поговорим отдельно.

В числе прочих инноваций в интерфейсе Star было впервые использовано представление экранного пространства в виде "рабочего стола" (Desktop). Эта метафора основывается на том, что пользователь якобы не имеет представления о существовании, скажем, программы под названием "текстовый редактор", а просто "открывает документ". Альтернативой является концепция "инструментов" (Tools) или "приложений" (Applications), где пользователь запускает нужный инструмент (приложение), и с его помощью открывает документ, причем тип его идентифицируется обычно по расширению имени файла. Ответственность за результат, например, загрузки файла, содержащего изображение, в текстовый редактор при этом целиком ложится на пользователя. Разработчики Star сознательно шли на потерю универсальности, присущую инструментальной модели, в то же время предупреждая об ограниченности сферы применимости метафоры "рабочего стола" исключительно офисными системами. (Дуглас Энгельбарт, изобретатель мышинного интерфейса, позднее иронизировал: *"Весь мир был увлечен идеей "офисной автоматизации", будучи уверен, что "настоящие пользователи" компьютеров — это секретари, чьи задачи необходимо автоматизировать."*)

В 1978 году упомянутый ранее Джеф Раскин работал в Apple, где начал новый проект под названием Macintosh, однако уже в 1982 г. разругался с Джобсом и покинул фирму. Впрочем, это неудивительно — ни один из его проектов так и не стал успешным в коммерческом смысле, притом что выдвинутые им теории обязательно изучают на всех соответствующих курсах в мире. Основная заслуга Раскина в том, что он одним из первых осознал: самое важное ментальное ограничение человека — ограниченность внимания. Фокус внимания у человека один. Интерфейсы, переключающие на себя внимание человека, — это одна из ключевых причин неэффективности взаимодействия с машинами.

Заметки на полях

Вот пример: когда мне диктуют телефонный номер, я хочу сразу записать его, пока он не вылетел из памяти. В типичной современной ОС для этого требуется запустить специальное приложение (по крайней мере, переключиться на него), после чего создать новый контакт (это же надо еще знать, что то, что вам нужно, называется именно "контактами", см. *главу 16*), найти на появившейся форме подходящее поле, вписать туда номер... Автор этих строк и в бумажной-то телефонной книжке не вел записи по алфавиту, предпочитая для ускорения записывать их "внавал", и сейчас фиксирует номера, адреса и прочие полезные сведения в маленькой программке в стиле картотеки из Windows 3*x*. И даже забросил ведение адресной книги в The Bat! — слишком много усилий приходится

предпринимать для систематизации записей и извлечения оттуда нужного адреса, проще найти корреспондента через "Поиск".

Раскин выступил с радикальным предложением: отказаться от зоопарка разных приложений и заменить его на единую рабочую среду, которая всегда ведет себя одинаково. В такой среде, например, нажатие клавиш всегда приводило бы к вводу текста, поэтому человеку, желающему записать телефон, достаточно было бы просто его набрать. Разумеется, определить введенный номер в "правильное" место нужно было бы и в этой среде, но в ней это можно сделать после ввода номера, что принципиально удобнее. В результате Раскин придумал интерфейс, суть которого состоит в следующем: все, что хранится в компьютере, представляет собой единый документ, отдельные приложения — это команды и модули, запускаемые пользователем или подключающиеся автоматически. Причем все управление осуществляется с помощью клавиш — мыши Раскин не признавал, действительно, ведь для работы с текстом мышь в принципе не требуется, так зачем лишние сущности? Несмотря на понятные ограничения (а кроме инструментальной модели, предоставляющей пользователю полный доступ к "потрохам" системы, вплоть до ее перепрограммирования, любая другая модель является ограниченной и пригодной лишь в определенной области) эта система для тех, кто работает именно с документами, была бы, вероятно, более удобной. И в конце 80-х она даже была осуществлена на практике в компьютере Canon Cat, который разошелся в количестве 20 тыс. экземпляров, но затем проект был свернут.

Подход Раскина можно интерпретировать так: вся среда в компьютере есть одно универсальное приложение. Легко заметить, что черты этого подхода можно встретить, например, в MS Office, в попытках интегрировать все действия пользователя — или большинство их — в единую среду. Я не знаю, что могла бы представлять доведенная до логического завершения концепция Раскина (в Canon Cat был осуществлен фактически лишь текстовый редактор с функциями электронных таблиц), но есть сильное подозрение (основанное на практическом опыте), что ни один программист, даже такой корифей, как Джеф Раскин, не сумел бы сделать интегрированную среду во всех нюансах так, чтобы в ней было бы удобно работать всем без исключения.

Но вернемся к существующим интерфейсам. Для платформы PC (уже превратившейся к тому времени в Wintel) "рабочий стол" был реализован только спустя пятнадцать лет — с появлением Windows 95. Казалось бы, было время все продумать и учесть недоработки Xerox и Apple, но в Windows была предпринята эклектичная попытка сохранить все преимущества инструментальной модели, как наиболее гибкой, но рассчитанной на специально обученных пользователей, и в то же время склонить на свою сторону армию "чайников", обучаться не желающих или не имеющих возможности. В результате пророческие слова Раскина о том, что *"пользователь обычно не понимает, на-*

сколько ему неудобно" относятся к Windows, как ни к чему другому. Буквально любое незнакомое действие в Windows, несмотря на все попытки унификации и стандартизации, требует от "чайника" консультаций с более продвинутым собратом по несчастью[2]. Непоследовательное использование Desktop-метафоры может приводить к просто катастрофическим последствиям: т. к. первичным признаком для отнесения файла к тому или иному типу в Windows по-прежнему служит расширение его имени, то при неправильном переименовании файл может быть просто потерян для непросвещенного пользователя. При попытке скрыть расширения, как это происходит по умолчанию в Windows, часто может возникать забавная ситуация, когда в одной папке оказываются несколько абсолютно одинаковых с виду значков, что приводит к недоразумениям — одно дело послать по электронной почте многомегабайтный TIFF, другое — компактный JPEG, а ведь все иконки, относящиеся к изображениям (если вы не зададите специально обратного), будут связаны с одним приложением, и потому и обозначатся одинаково.

Рискну предположить, что правильным решением всех этих проблем был бы выпуск множества относительно автономных модификаций одной и той же ОС с интерфейсами, "заточенными" под нужды бухгалтера, секретаря, технического писателя, ученого, фотохудожника, журналиста, ребенка, наконец, с возможностью установки их в любой комбинации и переключения между ними. Глядишь, и модель Раскина вполне тогда вписалась бы в коммерческие продукты, и любителям прыгающих по экрану кнопок или "скрепок-помощников" тоже было бы где развернуться...

> Резюмируем то, что сказано ранее, попробовав сформулировать основную цель любого интерфейса — не только компьютерного. Подавляющее большинство действий совершается человеком бессознательно. Часть подобных действий человек умеет совершать от рождения, таких, как глотательные рефлексы или отдергивание руки от горячего предмета. Среди этих действий есть очень сложные *программы поведения* — например, половое поведение или стайные инстинкты. Другая часть — инстинкты и программы, приобретенные в процессе взросления и обучения, это навыки ходьбы, речи, письма и счета, или социальное поведение: стыдливость, способность к состраданию и т. п. Наконец, сравнительно небольшую часть действий составляют те, что контролируются сознанием. Однако именно эти действия требуют повышенного внимания, они человека, пользуясь расхожим выражением, "поглощают целиком". Ни один человек не может обдумывать две

[2] Причем по мере развития одни неудобства устраняются (так, реализация Plug&Play в Windows XP принципиально лучше всех предыдущих версий), зато множатся другие (в той же XP — непонятные проблемы с языком, атрибутами папок, активацией и т. п.).

вещи сразу. Поэтому любое обучение всегда преследует одну и ту же цель — перевести некие действия в область бессознательного, сделать так, чтобы они выполнялись автоматически. Именно на это направлены бесконечные тренировки спортсменов, балерин, специалистов по рукопашному бою, водителей автомобилей. Более того, на это также направлены и процессы умственного обучения — студентов заставляют решать учебные задачи с тем, чтобы в процессе практической деятельности нужное решение находилось как бы "само", на основе приобретенного опыта.

Интерфейс GUI менее всего приспособлен к эффективному обучению. Производители затвердили словосочетание "интуитивно понятный" применительно к интерфейсам, как будто это вообще что-то означает — интуитивно понятным является только тот интерфейс, к которому вы привыкли, и он не заставляет вас задумываться над каждым телодвижением. Обратите внимание, как ловко подростки манипулируют предельно "неинтуитивным" интерфейсом мобильников, и вы поймете, что это совершенно пустое понятие. Интерфейс может и должен быть *логичным* — хотя и это, в общем, имеет значение только для удобства обучения, но уж к "интуитивности" точно не имеет никакого отношения. Является ли "интуитивно понятным" интерфейс управления автомобилем? Отнюдь нет — обратите внимание, скажем, что сцепление нужно *нажимать*, чтобы его *выключить*. Или: если вы замрете при ходьбе, то остановитесь, а если вы все бросите в процессе езды, то автомобиль вовсе не остановится, как можно было бы предположить, исходя из *интуитивного* представления. Но никто ведь не протестует, правда? Потому что в принципе неважно, какие именно действия нужно вызубрить, "заложить в подкорку" (на самом деле — скорее в мозжечок) — важно, чтобы они не требовали в дальнейшем напряжения сознания. Если вы будете все время в процессе езды раздумывать над вопросом, какую педаль нужно нажать для снижения скорости — вряд ли вы вернетесь из поездки живыми.

А вот для существующего WIMP-интерфейса вызубрить действия даже в одной отдельно взятой программе нельзя. Кажущаяся простота действий при указании мышью пункта меню на самом деле оборачивается тем, что это действие невозможно автоматизировать — сам процесс таков, что требует полного переключения внимания. Нужно найти это меню зрительно, направить на него мышь, щелкнуть, потом найти нужный подпункт — все это настоящая полноценная задача, требующая обдумывания, занимающая сознание целиком, как требует обдумывания вопрос, куда именно направить автомобиль при движении. Но направление движения — это и есть настоящая цель управления автомобилем, а выбор пункта меню — вовсе нет. Отличный пример на эту тему приводит известный программист Михаил Донской: *"Такой интерфейсный элемент, как линейка прокрутки, находится в противоречии с одним из основных принципов психологии восприятия: у человека может быть только одна точка активного внимания. При использовании же линейки прокрутки приходится смотреть в две совершенно*

различные точки — на прокручиваемое изображение (не пора ли остановиться?) и на линейку. Всем знакомые неприятности с непопаданием мышью в нужную точку при прокрутке или с соскакиванием курсора мыши с линейки — очевидное следствие вышеуказанного противоречия."

Наглядность Windows, которая так привлекает начинающих, в сравнении с интерфейсом командной строки, оборачивается тем, что они и в дальнейшем вынуждены тратить огромное количество времени на обдумывание действий, которые в принципе никакого обдумывания не требуют. Это все равно, как если бы педаль тормоза в автомобиле каждый раз оказывалась на новом месте и ее нужно было бы искать, например, по тому, что на ней имеется табличка: "тормоз". Задача программиста — сделать таким образом, чтобы в идеале *каждое* действие можно было бы выполнять автоматически, "спинным мозгом", сосредоточившись именно на целевой задаче, а не над тем, в каком углу экрана в данный момент находится меню, и каким щелчком мыши — двойным или одинарным — оно в этой программе вызывается. Означает ли это, что надо вернуться к интерфейсу командной строки? Вовсе нет, нужно только четко себе представлять, что запомнить раз и навсегда последовательность нажатия двух-трех клавиш в конечном итоге намного проще, чем каждый раз целиться в меню. А использовать при этом WIMP, инструментально-командную модель, метафоры "рабочего стола", "вселенной" или что-то другое — это уже дело, в принципе, десятое. Главное не забывать, для чего все это делается вообще.

Страна советов

Конечно, в реальности программисту приходится идти на компромисс, подстраиваясь под существующие стандарты. Точно так же, как архитектор или скульптор при своей работе не может не принимать во внимание законы сопромата, программист должен соблюдать некие правила, которые в совокупности делают его творение программным продуктом. Это, разумеется, в первую очередь относится к ПО, которое предназначено для распространения — неважно, за деньги или просто так. Но даже если вы делаете некую программу для собственных нужд, и это не просто единовременная "проба пера", а некий продукт, которым вы собираетесь пользоваться в течение долгого времени, эти стандарты нужно соблюдать. Грубую ошибку делает тот, кто решает за пользователя (даже если этот пользователь — вы сами), в какой последовательности ему нужно нажимать экранные кнопки. Рассчитывать нужно на тот случай, что пользователь справку не читает (даже если она есть) и обязательно первым делом нажмет именно ту комбинацию клавиш, которая "повесит" вашу программу. И вы сами через пару месяцев гарантированно забудете, что и в какой последовательности надо нажимать. Есть несколько общих правил

составления программ — и далее вы увидите яркие примеры того, как многие из этих правил нарушаются даже в самых известных продуктах.

Но прежде, чем перейти к советам по соблюдению этих правил, замечу, что есть одно главное правило, справедливое для всех почти без исключения программ, по крайней мере таких, которые не осуществляют настолько уникальные функции, что им невозможно найти замену. Это эмпирически найденное правило гласит: если пользователь не может без обращения к справке в течение максимум получаса разобраться с основными функциями программы, он ее бросает. Имея это в виду, перейдем к советам по составлению программ более подробно.

Совет 1 — о справке

Программа должна иметь справку. Совершенно необязательно составлять разветвленный Help в неудобном до крайности стандартном стиле Windows. В простейшем случае достаточно просто описать необходимые действия в текстовом файле, а в соответствующей главе мы покажем, как легко и просто без всяких дополнительных инструментов можно создавать достаточно навороченную справку в HTML-формате. Должен отметить, что только создание справки и написание комментариев (наиболее ненавидимые "настоящими программистами" этапы создания программы), займет у вас почти столько же времени, сколько написание самой программы, а иногда даже больше. Но нужно иметь в виду вот какое обстоятельство. Процесс написания справки позволяет взглянуть на вашу программу со стороны, обобщить все, что вы о ней знаете, и увидеть такие возможные упущения и ошибки, которые в противном случае выявились бы только после долговременной эксплуатации. (Задумайтесь, почему хорошие лекторы заставляют писать студентов конспекты собственноручно?) Так что не надо относиться к этому только, как к раздражающей "обязаловке".

Совет 2 — о комментариях и именах переменных

Текст программы должен быть как можно подробнее комментирован. Это тот самый случай, когда требования ГОСТ'а полностью отвечают реальному положению вещей. Той же цели повышения читаемости программ служит требование, чтобы наименования идентификаторов переменных были как можно более осмысленными: программа не рухнет, если вы назовете несколько переменных типа **`file`** именами `f1`, `f2` и т. д. Но вам не придется каждый раз искать в тексте место с их инициализацией, чтобы вспомнить, что `f1` — это файл установок, а `f2` — журнал (log-файл), если вы с самого начала присвоите им имена, например, **`file`**`_set` и **`file`**`_log`. Хорошим методом изобретения

идентификаторов является присвоение "говорящих" имен по звучанию (iks, igrek) или записанных транслитом по-русски (stroka). При присвоении имен использование фантазии по максимуму не возбраняется, но тут главное — не переборщить. Так, для именования простых численных переменных удобно использовать имена, аналогичные обозначениям в обычной алгебре: x, y, i, n и т. д. Вряд ли целесообразно затемнять смысл текста программы чем-то принципиально более навороченным.

Заметки на полях

В Windows используется так называемая "венгерская нотация", названная в честь программиста Чарльза Саймони (Charles Simonyi), венгра по происхождению. Он начинал свою карьеру еще на советских "Уралах" в начале 60-х, а в 70-е годы в упоминавшемся Xerox PARC занимался разработкой WYSIWYG-редактора Bravo. Позднее он стал главным архитектором MS Office (отсюда понятно, почему сам по себе MS Word, в отрыве от системы и поздних наслоений — вполне приличная вещь). Придуманная им нотация (т. е. правила записи имен переменных) заключается в том, что каждый идентификатор должен содержать префикс, информирующий о типе переменной. Само же наименование любой переменной должно отражать ее назначение и смысл. Тогда названия переменных — при надлежащем опыте — легко расшифровать. Так, имя "wm_KeyDown" говорит, что это сообщение Windows (Windows message), означающее, что нажата некая клавиша, а имя "faAnyFile" — что это набор атрибутов (file attributes), идентифицирующий любой (any) файл. Никто не запрещает вам придумывать и использовать свои собственные префиксы. Так, автор этих строк за некоторыми понятными исключениями старается начинать идентификаторы файловых переменных с буквы "f", строковых — с букв "st", массивов — с буквы "a" (array) и т. п. Некоторые Unix-программисты резонно замечают, что венгерская нотация ведет к излишним сложностям (например, если в процессе перехода на другую платформу тип переменной изменится, то придется переписывать весь код), что в значительной степени справедливо.

Совет 3 — об исключениях

Нужно тщательно просматривать программу на предмет возможного возникновения исключительных ситуаций. Приведу один простой пример: предположим, вы открываете некий файл, который только что сами создали. То есть он, казалось бы, гарантированно существует — ну какая тут может быть исключительная ситуация? Но если вы перед открытием не проверили, действительно ли он существует, то в сложной программе с множеством событий вы легко можете попасть в совершенно дурацкую ситуацию: представьте себе, что некий "ламер" взял и удалил этот файл в промежутке между созданием и обращением к нему. "Сам виноват" — скажете вы, и будете категорически неправы. В вашей воле прервать выполнение программы с сообщением типа "Файл ... не существует", но если программа при этом виснет или выдает нечто вроде невнятного "Access denied" — "ламер" даже не поймет, где и что

он сделал не так. А такого допускать нельзя. Я специально заостряю ваше внимание на данном примере, потому что на практике таких экзотических проверок, конечно, никто не делает, полагаясь на системные обработчики исключений. И мы также этим заниматься не будем, но вы должны понимать, что в принципе это неправильно.

Одна из грубейших ошибок практически всех производителей софта, включая разработчиков Windows и самой Delphi — когда сообщение о возникновении исключительной ситуации не содержит никакой внятной информации. В лучшем случае создатели текстов таких сообщений рассчитывают на специалиста. Автор не видит никаких причин, по которым сообщение "Ошибка 103" не могло бы выглядеть, как "Ошибка доступа к файлу <*имя_файла*>" (то же относится и к известной "Ошибке 404" в серверном ПО для Интернета, про которую, правда, все уже все выучили). И тем более недопустимы сообщения такого рода, если они возникают перед глазами конечного пользователя. Это одна из причин, по которым все возможные исключительные ситуации нужно обрабатывать в программе, не полагаясь на системные обработчики.

Совет 4 — о функциональности

Программа не должна делать ничего лишнего. Типичный пример — запуск второго экземпляра программы. Если это специально не предусмотрено (подобно тому, как Internet Explorer можно запустить в любом количестве экземпляров и это отвечает его назначению), то программа обязана, по крайней мере, информировать пользователя о том, что он запускает второй экземпляр.

Программа не должна уметь ничего лишнего. Часто встречающейся концептуальной ошибкой, которой подвержены даже самые крупные и известные софтверные фирмы, является неудержимое стремление как можно больше расширить функциональность продукта. Это далеко не всегда означает, что разработчики так уж глупы — просто в этой среде *принято* выпускать каждый год по новой версии продукта в целях поддержания конкурентоспособности. И в ситуации, когда некая программа и так уже отлично делает все, что нужно и даже сверх того, разработчики просто вынуждены в чисто маркетинговых целях увеличивать ее функциональность — очень часто вместо того, чтобы просто поправить ошибки и довести программу до ума. Типичный пример — Word for Windows, который уже в 6-й версии (еще под Windows 3*x*), и даже в почти забытой ныне 2-й, содержал практически всю нужную самому продвинутому пользователю функциональность. Когда же его пытаются декларировать, например, как средство создания HTML-страничек, ничего, кроме улыбок, это вызвать не может — это совсем *другой* продукт и предназначен он для иных целей. Единственным разумным спосо-

бом как-то объединить под одной крышей разные функции является не нагромождение функций в новой версии "все в одном", а модульный принцип построения, например, использование плагинов — кому надо, тот и установит. Но на самом деле и это не всегда требуется — вполне нормально, если каждая программа будет работать в своей области, но зато делать это на пять баллов. Ну не получается в прокрустовых рамках Windows делать универсальные программы в стиле Раскина! Я не знаю примеров программ, которые осуществляли бы разные действия лучше, чем отдельные программы, "выглаженные" каждая для одного своего действия.

Заметки на полях

Здесь можно привести аналогию с производителями принтеров, сканеров и факсов — почти 10 лет они не могли понять, почему это пользователи настолько глупы, что не желают понимать своей выгоды и приобретать устройства "все в одном" (многофункциональные устройства, МФУ) в полтора-два раза дешевле, чем по сумме каждого из агрегатов в отдельности. Количество проданных МФУ в России в 2004 году составило всего чуть больше 10% от количества принтеров, но тенденции этого рынка очень показательны: он вырос за год на 140% (а продажи принтеров — всего на 12%). Что же произошло? Очень просто: в течение всего периода с момента выхода первого МФУ, каждая из его функций сама по себе была заведомо хуже по качеству, чем отдельные устройства. И только теперь, когда все функции делаются на основе лучших отдельных продуктов той же фирмы, пользователи начинают соглашаться — да, так выгоднее и удобнее. Применяя эти соображения к нашим программам, можно выразиться так: пока Word в роли HTML-редактора будет хоть в чем-то хуже, чем DreamWeaver, эта функция в нем никому не нужна. Так что если вы решили написать текстовый редактор для создания программ на ассемблере, совершенно необязательно заставлять его проигрывать MP3-файлы — WinAmp с этим справится однозначно лучше.

Совет 5 — об интерфейсе

То, что мы говорили об интерфейсах ранее, можно конкретизировать так: программа не должна заставлять вас делать что-то лишнее. Меню программы, а также все операции должны быть простыми и понятными и по возможности осуществляться стандартным способом. Надо четко представлять, что именно вы хотите сделать, и идти к цели наиболее коротким и по возможности стандартным путем. Сама Microsoft злостно нарушила это требование еще много лет назад, когда по совершенно необъяснимой причине вдруг задействовала клавишу <Alt> для входа в меню (кстати, в *главе 7* вы узнаете, как написать утилиту, которая эту функцию отключает). Хочу предостеречь читателей от подобных извращений — системные клавиши (в первую очередь клавиши-модификаторы <Alt>, <Ctrl> и <Shift>, но также и такие, как <Esc>, <Del>, <End> и пр.) предназначены для выполнения совершенно определенных действий и не должны использоваться в вашей программе ни для чего другого.

В противном случае вы обязательно создадите большие трудности пользователям. Единственное исключение — нестандартное использование правых (дополнительных) клавиш <Alt>, <Ctrl> и <Shift>, изредка — второго <Enter>, которое стало уже традиционным по той причине, что для осуществления основных функций они оказались просто лишними. Допустимо также задействовать для какой-то оригинальной операции практически неиспользуемую системную клавишу <ScrollLock>, если вас не раздражает лампочка, а вообще горячие клавиши должны быть только из набора специально для этой цели предназначенных функциональных клавиш <F*x*> или буквенно-цифровых в сочетании с клавишами-модификаторами (достаточный набор их имеется прямо в меню соответствующего компонента Delphi и обычно ничего специально придумывать не надо). Не следует также вслед за разработчиками некоторых графических редакторов и CAD использовать в качестве горячих клавиш "голые" буквенные или цифровые клавиши — просто по той причине, что их легко случайно задеть и что-то при этом испортить. Излишне говорить, что любая достаточно солидная программа обязана дублировать все пункты меню горячими клавишами.

Заметки на полях

В качестве типичного примера, как делать *не надо*, можно привести программы для записи на CD, например, WinOnCD или Nero. Когда мне случилось столкнуться с этими программами первый раз, я был просто ошарашен. Ладно, к тому, что просто записать файл не получится, а надо обязательно создавать некий "project", я был готов. Но WinOnCD (речь идет о версии 6), помимо идиотских вопросов про какие-то "мультисессии", встретила меня четырьмя смежными окнами, снабженными в общей сложности 19 пунктами меню (многие из которых имеют иногда с десяток подпунктов). Плюс еще семь кнопок без подписей, но и этого разработчикам показалось мало, и на экране наблюдается еще 8 (восемь) закладок. "Боже мой, — растерянно думает пользователь в моем лице — а я-то хотел всего лишь файл на диск записать...". Не думаю, что сильно ошибусь, если предположу, что программисты фирмы Roxio (или те, что продали все это Roxio) решили, что они сделают интерфейс "интуитивно понятным", если продублируют одну и ту же функцию во всех местах, где только осталось свободное место. Какая уж тут психология, эргономика и прочие премудрости...

Поползновения "настоящих программистов" сделать как можно сложнее наблюдаются, например, в таких популярных и необходимых программах, как WinZip, и, к еще большему сожалению, сделанному по ее образцу WinRar — программах, в основе которых лежат чудесные алгоритмы, но с точки зрения удобства интерфейса они заслуживают огромный жирный кол. WinZip (в версии 6.3) имеет в сумме 47 пунктов меню и сверх того панель из восьми кнопок, и к тому же представлена в двух почти ничем не различающихся вариантах ("классический" и еще какой-то, не помню). Легко сообразить, что для программ типа WinZip и WinRar в их основной функциональности меню не

требуется вообще (как оно не требуется для окна папки в Проводнике) — все спокойно можно реализовать через правую кнопку мыши. Если разработчик хотел добавить что-то еще, он был просто обязан спрятать это с глаз долой — для функций типа создания самораспаковывающихся архивов лично я бы просто сделал отдельную программу (кстати, сама по себе она реализована в WinRar просто отлично, и мы непременно этим воспользуемся, см. *главу 17*).

Заметки на полях

В WinRar есть одна концептуальная ошибка, которая может показаться мелкой, но мы остановимся на ней подробнее, т. к. она очень показательна. В диалоге распаковки файлов в строке с указанием пути указано одно, а в рядом расположенном окне с деревом папок — совершенно другое. Учитывая, что строка пути сделана на основе компонента Edit с неубранным выделением (к компоненту Edit с его синим выделением мы не раз вернемся), и по умолчанию там путь высвечивается абсолютно нечитаемым белым по синему фону шрифтом 8 кегля, господин Рошал простит меня за то, что я принципиально не пользовался графическим RAR лет шесть — только по крайней необходимости. (WinZip я вообще никогда не пользовался — в Disco Commander входит свой Zip, обращение с которым на много порядков быстрее.) И только потом я обнаружил, что эта программа, оказывается, запоминает последнюю папку и наверху еще есть кнопочка "Показать"! Если это не есть типичное "умножение сущностей без необходимости", то что тогда? Подумайте, сколько *в принципе* щелчков надо сделать для того, чтобы просто извлечь файл из архива?

Еще один отрицательный пример — демонстрация сокращенного пути к файлу вместо полного. Так, известный антивирус Касперского при сканировании диска может продемонстрировать путь к проверяемому файлу в таком, например, виде: C:\...\svhost.exe. Скажите, на кого рассчитана такая запись и зачем это демонстрировать вообще? Допустимо лишь сокращать путь *до* папок, местоположение которых очевидно: запись "..\Borland\Delphi7\Projects" или "..\Program Files\Adobe\Photoshop" в большинстве случаев не создаст неудобств. Такая запись, наоборот, может быть полезна, т. к. начальные папки ("Borland" и "Program Files" в данном примере) могут быть расположены в самых разных местах, и мы в этой книге, например, таким образом подчеркиваем, что нам это безразлично.

И еще одно замечание по поводу иконок. Их часто по-русски именуют пиктограммами, но это неправильно — пиктограмма есть то, что нарисовано на иконке, а не сама иконка. Более правильно именовать иконки значками, но это слово в русском языке имеет достаточно много значений, и чтобы сразу было понятно, о чем речь, в этой книге мы будем использовать кальку с английского "icon". Но вне зависимости от названия, главное — при использовании иконки надо отчетливо представлять, что пиктограмма на ней ни в коем случае не может заменить текстовую подпись. Информативность пиктограммы зависит даже не от художественной квалификации того, кто ее составлял — она зависит от того, насколько художник и пользователь настроены

"на одну волну". А *после того*, как вы выучите, что именно обозначает тот или иной рисунок, становится уже неважно, что именно там нарисовано — пусть это хоть ровный квадрат определенного цвета, и это будет ничуть не менее информативно, чем сложный многоцветный художественный образ (возьмите в качестве примера дорожный светофор). Поэтому основная задача при составлении иконок — сделать так, чтобы они максимально *отличались* друг от друга. Не пытайтесь нарисовать, как это делают в большинстве случаев, похожие иконки, отличающиеся в деталях — не столь эффектно, но куда лучше с точки зрения функциональности будет выглядеть простейший рисунок, но зато имеющий бросающиеся в глаза отличия от соседних.

Совет 6 — о пользовательских установках

Еще один вопрос, который может показаться второстепенным, но на самом деле он очень важен. Это вопрос запоминания пользовательских установок. Конечно, если у вас и запоминать-то нечего, то и думать об этом не нужно. Но если вы, к примеру, предусмотрели в меню возможность изменения цветов интерфейса, нужно обязательно позаботиться о том, чтобы данные установки сохранялись к следующему запуску — иначе это останется совершенно бесполезной "фичей". Это очевидный пример, но есть и не столь очевидные — типичным примером "как делать не надо" будут некоторые функции все тех же программ от Microsoft. Они никогда не запоминают рабочей папки дольше, чем на один сеанс, и всегда обращаются в "Мои документы" (или то, что ее заменяет). Редчайшее исключение — Internet Explorer, который запоминает папку, куда сохранялись файлы через пункт **Сохранить как**. Объяснение этому упрямству простое и кроется в упомянутой парадигме "рабочего стола" — когда пользователь о "приложениях" понятия не имеет, а запускает исключительно "документы", ему нормальный диалог открытия и не нужен. Сложнее понять, почему была напрочь утрачена функция открытия файла именно на том месте, на котором вы с ним прошлый раз расстались. Как показывает пример Delphi и множества других программ, это совсем несложно сделать в любом редакторе. Казалось бы, можно привести аргументы в пользу того, чтобы установки, которых может быть достаточное количество, сбрасывались по окончании текущего сеанса — пользователь может просто не помнить, что он там такое установил и как вернуть все обратно. Но это порочное рассуждение: если вас это волнует, возьмите пример с разработчиков BIOS и поставьте отдельную кнопочку "Set default" (каковая, кстати, в программах от MS обычно также отсутствует). На практике возможность автоматически запоминать все установки, типы файлов, папку, куда производилось последнее обращение, место в тексте документа, на котором вы остановились последний раз, намного сокращает количество пустых операций и общее время работы с программой. В идеале для каждого такого пункта

должна быть предусмотрена отдельная возможность выбора "запоминать — не запоминать", но в принципе достаточно и общей настройки "Запоминать установки".

Совет 7 — об украшательствах

Наконец, позвольте расширить сказанное ранее об интерфейсах пользователя и дать совет любителям разукрасить программу, как новогоднюю елку, чтобы все мигало, переливалось и бегало по экрану. Пожалуй, верно, что вкус не воспитывается, он либо есть, либо его нет, но что можно тут сделать, так это посоветовать всегда помнить, для кого и для чего вы все затевали. Украшательства допустимы и уместны, если вы делаете, например, свою версию медиаплеера для младшего школьного возраста, но будут смотреться достаточно дико в программе, предназначенной для профессиональной обработки музыкальных файлов. Найдите мне хоть одного человека, который использует MS Word для серьезной работы и которого при этом не раздражает пресловутая "скрепка-помощник". Заметим попутно, что все то же самое относится к злоупотреблениям Flash-роликами при построении сайтов[3]. Как нельзя более тут подходят слова классика:

Когда в делах — я от веселий прячусь,
Когда дурачиться — дурачусь.
А смешивать два этих ремесла
Есть тьма искусников — я не из их числа.

(А. С. Грибоедов, "Горе от ума")

Но, с другой стороны, украшательства иногда просто необходимы по делу. Пример: вы делаете программу, которая имитирует функции осциллографа. Тут имеет смысл приложить все усилия, чтобы "осциллограф" выглядел "как настоящий" — с сеточкой на экране, с зелененькой кривой, с "ручками" управления, с кнопками-переключателями. Так вы сильно облегчите задачу освоения программы теми, кто настоящий осциллограф знает, как свои пять пальцев, и время, потраченное на отработку интерфейса, уйдет не просто на упражнения в программировании с целью продемонстрировать (в основном своим коллегам-"ламерам") какой вы крутой, а по делу.

[3] По степени навязчивости и отсутствия элементарного вкуса в использовании Flash авторы некоторых сайтов и рекламных баннеров переплюнули даже телевизионную рекламу, создание которой по крайней мере предполагает какой-никакой уровень профессионализма. Автор этих строк однажды просто взял и избавился от Flash-плагина раз и навсегда, перешел на браузер FireFox (чтобы не тормозило) и с тех пор чувствует себя значительно лучше — хотя и понимает, что лишил себя доступа к редким удачным опытам в этом направлении.

Совет 8 — об автоматизации

Основная цель, для которой изначально были созданы компьютеры — автоматизировать и ускорить те операции, которые человеку самому проводить сложно. Это никогда не следует упускать из виду. Описанные ранее программы, типа WinOnCD, выглядят глупо именно потому, что их разработчики об этом напрочь забыли. Давайте подумаем: ведь CD-ROM есть не что иное, как просто еще одна разновидность памяти. Формально он ничем не отличается от, к примеру, дискет. Разве вы видели программу записи на дискету, которая вынуждала бы создавать какие-то "project"? Обращение к дискете происходит совершенно прозрачно для пользователя, единственное, что ему требуется делать — следить за тем, чтобы она была отформатирована (что, если вдуматься, тоже лишнее, вполне можно было бы обойтись и без перекладывания этого действия на плечи пользователя). Программы записи на CD — типичные образчики *программистского мышления*.

Все, что может быть автоматизировано, должно быть автоматизировано, и неважно, рассчитана ли ваша программа на виртуоза-профессионала или на "ламера", привыкшего использовать компьютер исключительно как печатную машинку. Возьмем упомянутую ранее обработку исключений. Корректно составленная программа в случае обнаружения ошибки при каких-то действиях пользователя должна выводить сообщение об этом — но при действиях именно пользователя, а ни в коем случае не самой программы! Если ошибку можно безболезненно обойти, то это исключительно внутреннее дело программиста. Пользователя о таких ситуациях нужно информировать только, если от него требуется принятие какого-то решения, в противном случае он устанет нажимать на кнопку **OK**, закрывая окна с совершенно бесполезной для него информацией.

Но и впадать в противоположную крайность, делая все действия программы полностью скрытыми от пользователя, тоже не стоит — помните, что решения, которые принимаете вы, мягко говоря, не всегда являются оптимальными, вы просто не можете предусмотреть всего. Поэтому настройки автоматизации нужно спрятать от "чайника", но сделать так, чтобы они были доступны квалифицированному пользователю по максимуму.

Заметки на полях

Вот пример: пользователи скажут вам большое спасибо, если вы не поленитесь сделать автозаполнение для полей какой-либо формы так, что их нужно только немного подправить. Это касается и диалогов и интернет-форм, и вообще любых действий, когда от пользователя требуется ввести с клавиатуры что-то конкретное. Например, если пользователю требуется ввести дату, то *обязательно* нужно, чтобы в соответствующем поле появлялся образец, иначе он будет долго гадать, что именно от него хотят: вводить ли дату, как "21.06.05", "21.06.2005", "06.21.05", "21/06/05", или, может быть, как "21 june 2005"? Но нет

ничего печальнее опыта разработчиков Word, которые включали автозаполнение даты в тексте по умолчанию — само по себе это нужно крайне редко, а такая функция может создать кучу неприятностей. Представьте себе, что вы открыли созданное полгода назад служебное письмо и вдруг все даты в тексте автоматически поменялись на текущую. Неопытная секретарша может даже и не заметить этого, пока не получит выговор от шефа — это в лучшем случае. Еще хуже выглядит "автоматизация" в стиле Excel, когда вы обнаруживаете, что вместо введенного вами числа 3005,97 в ячейке вдруг оказывается записано "30 мая 1997". Как говорится, это было бы смешно, если бы не было так грустно...

В общем, все рекомендации по этому поводу вполне укладываются в лозунг: "Сервис не должен быть навязчивым!".

Это не все, что мне бы хотелось отметить в качестве советов далеко не только начинающим программистам, но для начала достаточно. Ранее я замечал, что одно выполнение первых двух пунктов может значительно увеличить время вашей работы над программой, а если учесть все остальное, то можно прийти к выводу, что основное время будет потрачено на такие вещи, которые к собственно главной функциональности программы не относятся. Да, это так, и добавлять "к сожалению" я не считаю нужным. Потому что даже такая утилитарная вещь, как приготовление и употребление пищи, предполагает создание определенного комфорта, и хозяйке приходится тратить иногда гораздо больше времени на то, чтобы приготовить не как-нибудь, а вкусно, и еще красиво оформить стол, а потом помыть посуду и вычистить кухню. Все это — неотъемлемая часть процесса приготовления еды. И программирование тут ничем не отличается от любой другой области человеческой деятельности. Можно только помочь хозяйке, например, подарив ей посудомоечную машину — именно в роли подобного помощника и выступают современные среды визуального программирования.

Немного о стилях программирования

Принцип, который автор этих строк всегда старался соблюдать, как при составлении программ, так и при разработке электронных схем — "не умножать сущностей без необходимости". Должна быть четко поставлена цель, и она должна быть достигнута минимально возможными средствами. Однако помните, что на пути упрощения вас всегда подстерегает опасность что-то не предусмотреть или забыть, и если есть такая опасность, то лучше, как говорится, "перебдеть, чем недобдеть". Так, в *главе 20* при чтении из последовательного порта для сокращения текста программы мы пренебрежем рекомендуемыми проверками типа:

```
if not ClearCommError(hPort, dwError, @ComStat) then
raise Exception.Create('Error clearing port');
```

Автор этих строк много раз пытался искусственно вызвать ситуацию, когда указанная функция ClearCommError и аналогичные ей возвращают ошибку, и ему это не удалось. Поэтому он сделал вывод, что на практике такая ошибка крайне маловероятна. И все же, если вы делаете программу для широкого распространения, этими проверками ни в коем случае пренебрегать нельзя: мало ли что может случиться, если программа выйдет из-под вашего контроля. Однажды мне пришлось столкнуться со случаем (единственный, правда, раз в жизни), когда заведомо существующий коммуникационный порт не поддавался инициализации никакими способами. И если в программе не было нужных проверок, то она просто "висла", а это недопустимо.

Несколько слов об организации циклов. Те, кто хоть раз пробовал программировать на низкоуровневом ассемблере, знают, что никаких операторов **while**, **for**, **if**...**then** и других подобных атрибутов языков высокого уровня для процессора не существует. Любое подобное действие осуществляется на самом деле с помощью условных операторов перехода на метку, а еще точнее — на нужную команду по ее адресу (в этом смысле канонический Basic, столь нелюбимый профессионалами, был даже ближе к ассемблеру, чем C или тем более Pascal). Покажем, как, например, выглядит на Intel-ассемблере цикл **if**...**then**...**else**. Пусть есть переменная **var**_x, константа **const**_1 и две процедуры: pr_1 и pr_2. Нам требуется выполнить следующую часто встречающуюся задачу: ЕСЛИ переменная **var**_x больше **const**_1, ТО обнулить переменную **var**_x и перейти к процедуре pr_1, ИНАЧЕ выполнить процедуру pr_2. После выполнения той или иной процедуры требуется, естественно, продолжить выполнение основной программы. Соответствующий программный код может быть, например, таким:

```
. . . . .
cmp var_x,const_1 ;сравниваем
ja metka1 ;если больше, то на метку 1
call pr_2 ;иначе вызываем процедуру 2
jmp metka2 ;после нее на продолжение
metka1: xor ax,ax ;обнуляем регистр ax
mov var_x,ax ;переменная = 0
call pr_1 ;вызываем процедуру 1
metka2: <продолжение основной программы>
. . . . .
```

А вот та же самая задача, но реализованная на языке Pascal:

```
if (var_x>const_1) then begin var_x:=0; pr_1 end else pr2;
```

На этом примере очень хорошо видны преимущества языков высокого уровня. Знаменитый лозунг Эдгара Дейкстры "программирование без **goto**", таким образом, предполагает фактически использование эмуляторов действий, свя-

занных с использованием условных и безусловных переходов, но зато следование ему значительно повышает читаемость программ. И все же оператор **goto** включен во все современные языки, и никто вам не может запретить его использовать. Для людей, которые имеют образно-аналитический склад мышления, иногда проще разобраться в хитросплетениях условных и безусловных переходов, чем вникать, к примеру, в семантические различия между циклами типа **while**...**do** и **repeat**...**until**.

То же самое относится и к использованию флагов. Прием, когда по некоторому событию устанавливается некий флаг (в простейшем случае — всего один бит в некоторой переменной или регистре, но можно использовать и переменные целиком), является основным способом организации взаимодействия отдельных команд, целых процедур и даже узлов аппаратуры при низкоуровневом программировании. Так, в представленном ранее примере тоже неявно используется флаг — специальный бит в регистре флагов (бит переноса CF) устанавливается или сбрасывается в результате операции сравнения cmp (фактически — операции вычитания) и служит сигналом для последующей команды перехода. В высокоуровневых языках, как правило, можно обойтись без специальных флагов и институционально такой прием вообще "вне закона", но, как мы увидим, иногда его использование не только значительно облегчает составление программ, но и применяется самими разработчиками системы (скажем, булевская переменная CanClose, используемая в методе **Close** формы, есть не что иное, как типичный флаг по смыслу и назначению).

Немного о локальных и глобальных переменных. Автор этих строк предпочитает везде, где только можно, для передачи параметров пользоваться глобальными переменными — вместо того, чтобы организовывать процедуры с формальными параметрами. Это, разумеется, не руководство к действию, а просто привычка — на мой взгляд, так повышается читаемость и прозрачность программы. Разумеется, при использовании глобальных переменных для организации, например, циклов вы легко можете наделать ошибок, в причинах которых иногда непросто разобраться, недаром Delphi выводит даже специальное предупреждение на этот счет. Нет никаких особых причин, кроме указанных, для того чтобы следовать тому или иному стилю программирования, только при использовании локальных переменных автор настоятельно рекомендует по возможности присваивать им имена, не совпадающие ни с другими локальными переменными, ни с глобальными, иначе в тексте вашей программы не разберется и сам Валерий Васильевич Фаронов.

Еще одно замечание относится к собственно структурированию. Если текст процедуры состоит из одного-единственного оператора, то для повышения читаемости, на мой взгляд, лучше повторить нужный фрагмент несколько раз в основном тексте, каждый раз с новой переменной, чем выделять его в от-

дельную процедуру — хотя это и противоречит принципам структурного программирования. Не забывайте, что структурирование по процедурам и функциям удобнее с точки зрения программирования, но часто затрудняет чтение текста программы и к тому же ведет к раздуванию скомпилированного кода за счет организации стека при вызове процедуры.

ГЛАВА 2

Начинаем работу
Создаем типичное приложение

> При ремонте окон ни в коем случае нельзя вытаскивать шурупы клещами и забивать их молотком. Следует пользоваться только отверткой.
>
> *"1000 советов любителю мастерить"*

В качестве базового приложения мы создадим просмотрщик слайдов. Это, с одной стороны, функционально достаточно богатая вещь, и содержит в себе примеры многих типичных задач, возникающих при создании пользовательских программ. С другой стороны — само по себе приложение несложное, что позволит нам не отвлекаться на частности. В иллюстративных целях я опишу процесс создания достаточно подробно. Хотя в этой главе вы ничего, выходящего за рамки обычной работы в среде Delphi, не встретите, но подробности позволят даже самому неискушенному читателю избежать потерь времени на разрешение многих частных проблем, возникающих при создании приложений. А тем, кто процесс создания приложения уже знает достаточно хорошо, я все же рекомендую не просто скопировать проект с прилагаемого диска, а просмотреть эту главу хотя бы по диагонали — возможно, вы встретите здесь некоторые вещи, о которых ранее не знали.

Два слова по предварительной подстройке IDE (пункт меню **Tools**). Перед тем, как приступить к работе, я рекомендую сделать следующие установки:

- в пункте **Tools | Environment Options** на закладке **Preferences** установить флажки в обоих пунктах группы **Autosave options** и в пункте **Show compiler progress**. Первое даст вам возможность автоматически сохранить последний вариант проекта в случае, если Delphi при пробном запуске приложения "повиснет" (а в некоторых случаях, особенно при прямом обращении к API, это совсем не исключено), второе сделает процесс компиляции более наглядным;

- в пункте **Tools | Editor properties** на закладке **General** установить флажок в пункте **Undo after save**. Особенно актуально иметь такую возможность отмены изменений в тексте программы после неудачной компиляции.

Есть еще один пункт, на который стоит обратить внимание. Он находится на вкладке **Tools | Debbuger Options | Language Exceptions** и называется **Stop on Delphi Exceptions**. Если оставить его отмеченным (режим по умолчанию), то Delphi после запуска приложения будет "тормозить" на всех процедурах обработки исключений, что не слишком удобно на практике. Однако в некоторых случаях это может оказаться необходимым — не только, когда процедура обработки исключения работает не так, как запланировано, но и в случае необходимости отследить ошибки времени выполнения, которые иначе поймать не удается. Целесообразнее всего этот флажок снять, но на всякий случай помнить о такой возможности.

Теперь можно приступать к созданию проекта. Сначала надо кратко набросать его план — что именно мы хотим получить? В крупных проектах эта стадия — написание технического задания — может занять достаточно длительное время, а мы можем просто записать план на бумажке. Итак, наш простейший просмотрщик слайдов будет:

- открывать JPEG-файлы, находящиеся в нужной папке;
- демонстрировать картинки из этой папки по очереди от первой до последней по команде пользователя;
- демонстрировать их в режиме автоматического слайд-шоу, с неким интервалом времени, в закольцованном режиме.

Конечно, для того чтобы создать нормальную программу такого направления, следовало бы еще ввести:

- возможность просмотра картинок в виде набора "превьюшек" (preview);
- регулировку времени между демонстрацией отдельных слайдов;
- сортировку имен файлов по алфавиту;
- поиск файлов не только с расширением jpg, но и jpeg, а также, по возможности, и других форматов;
- отображение имени файла и размера изображения;

и т. п.

Все это не так уж и сложно, но в результате получится довольно громоздкая конструкция, а наша задача сейчас в том, чтобы создать законченное и компактное приложение-основу, которое мы потом будем постепенно дорабатывать. Все перечисленные функции, и даже несколько сверх того, мы введем в дальнейшем по ходу изложения *(законченный проект SlideShow см. в главах 15 и 16)*.

Начало

Запустим Delphi и создадим заготовку нового проекта (**File | New | Application**). Самое первое, что нужно сделать, — сохранить еще не рожденный проект, для чего выбираем пункт **File | Save project as**. У вас появится обычное окно сохранения файла с заголовком **Save Unit As**. Не забывайте, что введенное здесь имя — еще не имя проекта (программы), это только имя главного модуля с текстом, поэтому можно вести любое имя файла, и даже оставить предлагаемое по умолчанию — Unit1.pas, хотя это и неудобно (представляете, сколько у вас этих одинаковых "юнитов-1" потом наберется?). Введем имя, например, slide.pas. Перед тем, как нажимать кнопку **Сохранить**, в текущей папке Projects следует создать специальную папку для проекта, иначе потом файлы этого и других проектов перепутаются. Выбор папки и наименований — очень ответственный момент, потому что в Delphi переименовать что-то в проекте с перемещением его в другую папку — процедура не совсем тривиальная. Если вы в дальнейшем будете создавать еще модули, то обращайте внимание на то, в какой папке они будут сохраняться — не исключено, что они по умолчанию окажутся, например, в папке Мои документы, и вы потом потратите кучу времени на то, чтобы корректно собрать проект воедино.

Итак, создаем папку с именем Glava2, открываем ее и, наконец, сохраняем модуль. У вас тут же появляется аналогичное окно, но с заголовком **Save Project As** — тут уже надо вводить то имя, которым потом будет называться исполняемый файл программы. Пусть это так и будет SlideShow (пока этим именем будет называться файл проекта — SlideShow.dpr).

Компоненты

Заготовка будущего проекта, которую мы видим, представляет собой пока просто пустую форму `Form1` — будущее окно программы. Процесс программирования будет заключаться в том, что мы наполним форму компонентами, расставим их по местам, придадим им в Object Inspector необходимые свойства, затем напишем обработчики нужных событий. В процессе работы нам придется неоднократно запускать на выполнение еще недоделанный проект, так удобнее искать ошибки — *отлаживать программу*. При запуске на выполнение проект будет каждый раз *компилироваться* — в выбранной нами папке появится исполняемый файл программы SlideShow.exe, который, собственно, и будет запускаться по команде **Run** из меню Delphi (для этого нужно выбрать пункт меню **Run | Run** <**F9**>). Напомним, что если в программе есть критические ошибки периода компиляции, то компиляция не пройдет до конца, а на ошибки вам укажут. Если вы просто хотите проверить программу на отсутствие ошибок, не запуская ее, то нужно нажать клавиши

<Ctrl>+<F9> — тогда программа скомпилируется, не запускаясь (почему-то в 7-й версии Delphi этот пункт перенесен из меню **Run** в меню **Project**). При этом нужно учитывать два обстоятельства: во-первых, то, что при наличии ошибок EXE-файл не создается, а старый окажется стерт — т. е., если вы хотите сохранить промежуточный вариант скомпилированного файла, это следует делать до внесения изменений.

Первый компонент, который нам понадобится, — `MainMenu`, главное меню программы. Он расположен в палитре компонентов на закладке **Standard**, второй слева. Щелкаем по иконке, потом щелкаем на форме — иконка переносится на форму. Подобные компоненты можно располагать в любом месте формы (меню после компиляции все равно окажется там, где надо), но удобнее его поместить ближе к верхнему обрезу формы, иначе иконка будет "путаться под ногами" при установке других компонентов.

Затем нам понадобится диалог открытия файла. Он расположен на закладке **Dialogs**. Для того чтобы до нее добраться, нужно прокрутить палитру компонентов с помощью стрелочки у правого края верхнего окна. Компонент `OpenDialog`[1] самый первый слева. Переносим его на форму описанным ранее способом и располагаем где-нибудь рядом с меню. Далее таким же способом переносим на форму компонент `Timer` (нам ведь надо будет чем-то отмерять время при автоматическом показе) — он находится на закладке **System**.

Из визуальных компонентов нам в первую очередь понадобится кнопка, с помощью которой мы будем менять слайды. Используем также стандартную кнопку (компонент расположен на закладке **Standard**, иконка в виде маленькой кнопочки "Ok"). Переносим ее на форму и располагаем там, где она будет находиться в программе — внизу посередине. Кнопке можно придать любые размеры, растягивая ее мышью, но здесь нас устраивает и стандартный размер — если не нравится, можно подправить. Пока на ней написано `Button1`, но потом мы изменим надпись.

Теперь мы добавим один компонент для красоты — рамку будущего экрана для демонстрации. Картинки на экране несравненно лучше выглядят на черном фоне. Одно из крупнейших упущений составителей почти всех без исключения подобных программ под Windows — то, что они (скорее всего, даже не подозревая об этом) пытаются следовать рекомендациям для демонстрации именно картин, т. е. изображений на твердом носителе в отраженном свете. Такие изображения следует располагать на нейтральном сером или близком к нему фоне (именно так оформляют картинные галереи). А вот самосветящиеся изображения нужно помещать на черный и только на черный

[1] Напомню, что есть и специальный компонент-диалог открытия изображений (всем знакомый по MS Word: когда нам приходится вставлять рисунок в текст, именно этот диалог и используется), но в данном случае мы, исходя из нашей общей установки поступать как можно проще, будем использовать универсальный диалог.

фон (вспомните интерьеры кинотеатров), иначе они зрительно потеряются. Причем дополнительным достаточно сильно искажающим восприятие фактором будут окружающие картинку интерфейсные элементы — меню, кнопки, бордюр и т. п.; особенно это критично для "развлекательного" стиля Windows XP. В идеале нужно демонстрировать изображение "во весь экран", но это не всегда удобно для пользователя, поэтому придется хотя бы отделить изображение от интерфейсных элементов черным фоном-рамкой.

Рамку удобнее всего сделать из компонента `Panel` (находится на закладке **Standard**). Переносим ее на форму (она получит имя `Panel1`) и растягиваем за уголки так, чтобы от формы по боковым краям осталась только узенькая рамочка, а вверху и внизу оставалось небольшое пространство для меню и кнопки. Последний и главный компонент, который нам потребуется — собственно экран для отображения картинок. Этот компонент расположен на закладке **Additional**, носит незатейливое название `Image`, и его иконка выглядит, как сине-голубой квадратик (который, видимо, означает деревенский пейзаж на фоне голубого неба). Переносим его на форму, щелкнув на уже имеющейся там панели `Panel1` (компонент обозначится просто пунктиром и получит имя `Image1`), и тоже растягиваем, оставив поля, которые потом будут черными. К этому моменту наша форма должна иметь вид, показанный на рис. 2.1.

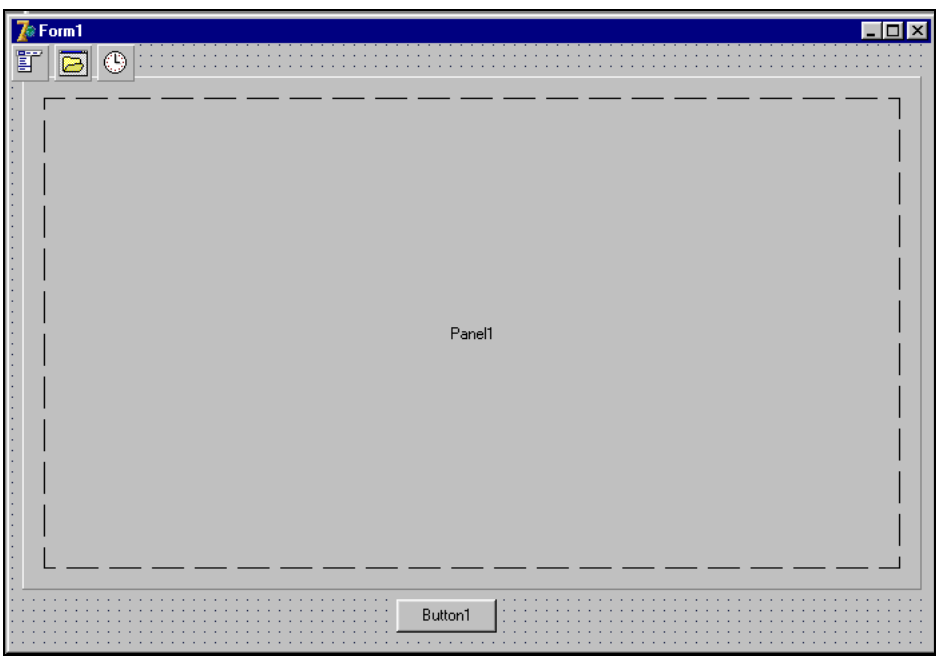

Рис. 2.1. Заготовка проекта SlideShow