

Проектное обучение

ОБУЧЕНИЕ МОБИЛЬНОЙ РАЗРАБОТКЕ НА DELPHI

ВСЕВОЛОД ЛЕОНОВ



ОБ АВТОРЕ

Леонов Всеволод Владимирович.....4

ОТ АВТОРА5**ГЛАВА 1****ЗНАКОМСТВО С DELPHI И ПЕРВЫЙ ПРОЕКТ**

1.1. Как пользоваться книгой.....	7
1.2. Что такое Delphi/C++Builder/RAD Studio.....	12
1.3. Из чего состоит и как работает Delphi	18
1.4. Библиотеки визуальных компонентов	22
1.5. Где взять Delphi с возможностью мобильной разработки.....	26
1.6. Установка пробной версии по шагам	29
1.7. Редакции продукта Delphi/C++Builder/RAD Studio	33
1.8. Подготовка к мобильной разработке	36
1.9. Первый проект–таймер	41

ГЛАВА 2**ОСНОВНЫЕ КОМПОНЕНТЫ ДЛЯ МОБИЛЬНОЙ РАЗРАБОТКИ**

2.1. Основные модели интерфейсов для мобильных устройств....	48
2.2. Компонент TTabControl	50
2.3. Прототип приложения для самоконтроля знаний учащихся..	54
2.4. Конструкция (Sender as TButton).....	64
2.5. Добавление новых вопросов	66
2.6. Лучшее—враг хорошего или подождём с улучшениями	71
2.7. Вывод результатов	74
2.8. Улучшение дизайна приложения	78
2.9. Варианты интерфейса пользователя	84
2.10. Приемы повышения качества кода.....	87

ГЛАВА 3

АНИМАЦИЯ

- 3.1. Основы анимации в Delphi/RAD Studio/C++Builder98
- 3.2. Исследовательский проект с анимацией..... 103
- 3.3. Эффективный код или как правильно скрыть объект 109
- 3.4. Эффективный способ групповой анимации 116

ГЛАВА 4

3D-ГРАФИКА

- 4.1. Основы 3D-графики в Delphi/RAD Studio/C++Builder 121
- 4.2. Управление объектам 3D-сцены..... 133
- 4.3. Создание сложных сцен 138
- 4.4. Интерактивные 3D-сцены..... 150
- 4.5 Групповое взаимодействие при работе над 3D-проектом 157

ГЛАВА 5

МАТЕМАТИКА И ПРОГРАММИРОВАНИЕ

- 5.1. Начало приложения для графиков функций..... 159
- 5.2. Отображения графика функции 163
- 5.3. Улучшение программного кода для построения графиков 165
- 5.4. Настройка компонента TChart..... 169
- 5.5. Новые возможности построения графика функции 171
- 5.6. Модификация интерфейса 179
- 5.7. Некоторые сведения об объектно-ориентированном
программировании 184
- 5.8. Дальнейшее развитие взаимодействия кода и интерфейса.... 188

ГЛАВА 6

ДНЕВНИК НАБЛЮДЕНИЙ

- 6.1. Постановка задачи 201
- 6.2. Прототип приложения..... 203
- 6.3. Структура данных..... 213
- 6.4. Интерфейс детального просмотра 221
- 6.5. Интерфейс добавления новой записи..... 225

6.6. Связывание интерфейса и структур данных в памяти.....	228
6.7. Динамическое создание компонентов	231
6.8. Реализация детального просмотра	238
6.9. Чтение сохраненных данных из файла.....	243
6.10. Сохранение данных в файл	255
6.11. Добавление файлов в проект при развёртывании.....	257
6.12. Экспорт накопленных данных.....	261
6.13. Объектно-ориентированная работа с файлами	266
6.14. Развитие проекта	269

ГЛАВА 7

МОБИЛЬНОЕ ПРИЛОЖЕНИЕ ДЛЯ ИЗУЧЕНИЯ ПОЭЗИИ

7.1. Прототип интерфейса	271
7.2. Алгоритмы заучивания и их реализация	275
7.2. Рассуждения о поиске наилучшего варианта.....	279
7.3. Добавление нового алгоритма	281
7.4. Развитие приложения	285
7.5. Запись голоса.....	287
7.6. Уведомления	294

ГЛАВА 8

СИСТЕМА «СУФЛЁР» НА ОСНОВЕ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

8.1. Публичные выступления: новости, драма, поэзия	304
--	-----

ГЛАВА 9

ЗАКЛЮЧЕНИЕ

9.1. Что делать дальше или планы на будущее.....	327
9.2. Полезная литература.....	330
9.3. Онлайн-источники и информация в Интернет.....	332

Об авторе

Леонов Всеволод Владимирович

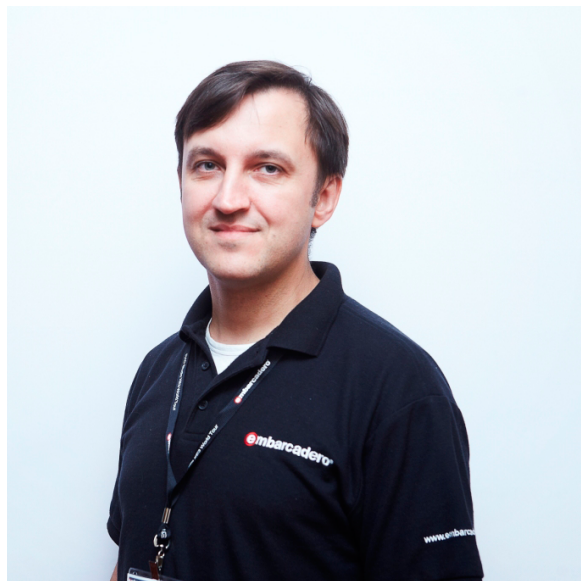
Кандидат технических наук, автор более 50 научных статей.

Занимался проблемами построения интеллектуальной САПР в МГТУ им. Н.Э. Баумана.

Преподавал Pascal, Delphi, C/C++, Microsoft Visual Studio, Microsoft SQL Server, InterBase и «Основы программирования» в Учебном центре «Специалист» при МГТУ им. Н.Э. Баумана.

Работал в представительстве Embarcadero по России и странам СНГ в качестве менеджера по взаимодействию с разработчиками.

В настоящий момент занимает должность менеджера по работе с ключевыми партнёрами департамента корпоративной мобильности Samsung Electronics.



От автора

Книга посвящена изучению программирования в среде Delphi. В качестве учебных примеров были выбраны приложения, которые могут быть использованы в проектном обучении. Материалом книги во многом сформировался в результате реализации программы **«Школа реальных дел 2014–2015»** под эгидой **Департамента Информационных Технологии Москвы** (ДИТ Москвы). Хочется поблагодарить **Игоря Степановича Марчака**, руководителя проектного офиса ДИТа Москвы, организовавшего данную программу и привлёкшего крупнейшие IT-компании к деятельному участию. Без Игоря Степановича данная книга вряд ли состоялась в том виде, в котором она заслуживает внимания с точки зрения проектного обучения.

Искренней благодарности также заслуживает **Сергей Александрович Терлецкий**, сотрудник компании Embarcadero за совместную работу с автором книги в реализации кейсов **«Школы реальных дел»**. Именно экспертиза Сергея Александровича в области реализации масштабных образовательных помогал выбрать проектное обучение в качестве магистрального направления при написании данной работы. Хочется выразить признательность **Екатерине Владимировне Макаровой**, менеджеру по маркетингу в Embarcadero, обладающей выдающимся талантом организатора. Без Екатерины Владимировны книга до сих пор не была бы закончена. Также следует отметить **Дениса Васильева**, менеджера по взаимодействию с разработчиками представительства Embarcadero по России и странам СНГ. Денис выполнил большую работу по описанию принципов настройки Delphi на мобильную разработку, ссылки на соответствующие публикации приведены в тексте книги.

Отдельно хочется поблагодарить **Елену Георгиевну Мякову**, педагога школы № 354 имени Д.М. Карбышева, под руководством которой команды учащихся достигли выдающихся результатов в **«Школе новых технологий 2014–2015»**, чем вдохновили автора приступить к написанию данной работы. Пусть она поможет всем желающим школьникам также эффективно создавать мобильные приложения.

Знакомство с Delphi и первый проект

1.1. Как пользоваться книгой

Вам не терпится начать разработку великолепных приложений для смартфонов или планшетов? Хотите удивить друзей и одноклассников, а также учителей и родителей? Тогда можете пропустить несколько разделов и начинать чтение с раздела «мой первый проект». Но не забудьте вернуться к изучению пропущенных разделов, как только появится свободное время между очередными атаками на программный код. Программирование это — не только умение вводить код и разрабатывать интерфейс. Это и понимание принципов работы среды разработки, и знание возможностей инструмента, и умение ориентироваться в различных версиях и редакциях продукта. Не забудьте найти время на проработку первых частей! Но лучше, конечно, читать последовательно, методично прорабатывая изложенный материал.

Есть много способов начать свой первый проект. Некоторые начинающие программисты делают первые шаги в увлекательный мир разработки приложений под руководством преподавателя или опытного старшего товарища. Во многом это — один из самых эффективных способов. Но мир программирования быстро меняется, не всегда можно найти наставника, тонко разбирающегося в особенностях последних версий сред разработки приложений. Даже если рядом с вами нет такого «гуру», но есть хорошая книга и доступ к сети интернет, то можно и самостоятельно в кратчайшие сроки овладеть техникой программирования приложений, включая мобильные. Основной целью данной книги как раз и является наглядная демонстрация приёмов создания программ в одной из самых популярных

и доступных сред разработки Delphi/C++Builder/RAD Studio. Почему используется такое сложное название, будет рассказано дальше.

Программирование давно считалось искусством. Отчасти это — правда, т.к. программистам приходится проявлять изобретательность, а многие навыки для непосвященных выглядят как элементы магии. Сами программисты с радостью поддерживали сложившиеся мифы, используя свой сложный профессиональный язык и принимая загадочный вид каждый раз, когда новички задают вопросы. Снято много художественных фильмов о приключениях всемогущих хакеров, каждый раз силой своего разума пытающихся или завоевать, или спасти мир. Информационные технологии в силу своего бурного развития, дающего человечеству новые возможности, действительно «творят чудеса». Но в их основе лежит достаточно обычный труд, больше похожий на работу инженера. Долгое время недостаток хороших книг, отсутствие глобальной сети для поиска информации и просто малое количество знающих людей возвели программирование в ранг мистической деятельности. Настоящая книга поможет вам разобраться во всем без всяких чудес, фокусов и трюков.

Программирование можно изучать разными способами. Уроки в школах и тщательное выполнение домашних заданий формируют надежную основу для самостоятельного развития. Школьные занятия преследуют цель поэтапного овладения базовыми навыками в индивидуальном порядке. Если они у вас уже есть, а также есть желание самостоятельно или под руководством наставника сделать резкий рывок вперед, обогнав не только школьную программу, но и программу обучения во многих среднеспециальных и высших учебных заведениях, то данная книга — отличный вариант. Предполагается, что с языком Pascal вы уже знакомы, среду разработки Delphi тоже знаете, как и понимаете основные принципы построения приложений. В последней главе будут приведены рекомендации по источникам информации, если в них ощущается дефицит. Но есть принципиальное отличие настоящей работы от большинства учебных курсов и справочных пособий. Данная книга полностью состоит из «проектов». Почему так важно изучать программирование в процессе выполнения проектов?

Владеющие техникой создания приложений в Delphi/C++Builder/RAD Studio помнят, что всё начинается с создания нового «проекта». В текстовом редакторе мы создаём «документ», в видео-редакторе — «видео-ролик», в среде разработки приложений — «проект». Проект по разработке приложения имеет начало, когда определена идея и функциональность будущей программы, реализация которой требует определенного количества времени. Конечным результатом является работающее приложение, а если таковое не готово, то проект считается незавершённым. В этом и есть основное отличие от других школьных предметов. Можно изучать историю, биологию или физику, где положительным итогом считается некая сумма накопленных знаний. То же самое можно сказать и про информатику. Но умение программировать означает способность создавать работающие приложения, выполняя конкретный проект. В книге представлено восемь таких проектов. Если вы выполните их, то можете считать себя уже весьма опытным программистом, способным самостоятельной к разработке достаточно сложных и даже коммерчески значимых. Никто не сможет в этом вас оспорить, т.к. в вашу пользу будут говорить ваши работающие программы.

Реализация проектов — единственный способ создавать программы, но проектная работа полезна и сама по себе. Она учит начинающего программиста последовательно идти к намеченной цели путём регулярных и интенсивных занятий. Вы научитесь определять глобальные цели и текущие задачи, распределять время и усилия, контролировать ход выполнение работ. Рассмотренные в книге проекты рассчитаны на индивидуальное выполнение, но не исключают и групповую работу. Если проект сложный, то на его реализацию может потребоваться значительные временные ресурсы. В течение этого периода может пропасть энтузиазм, но дело не только в этом. Большой проект требует участия различных специалистов, не только программистов. Дизайнер интерфейсов, тестировщик приложений, технический писатель, специалист по внедрению, а также «творец идей» — все они помогут не только ускорить реализацию проекта, но и повысить качество полученного результата. Можно, конечно, совместить все эти роли в одном человеке. Но реальной жизни создание приложений уже

давно стало коллективным трудом. Есть смысл попробовать работу в команде еще на стадии обучения.

Групповая работа требует прежде всего правильной организации. Лучше, если роль лидера возьмёт на себя учитель или преподаватель, авторитет которого непререкаем. Можно выбрать на эту должность и кого-то из учащихся, дав команде максимальную свободу действий. В этом случае преподаватель будет выполнять функции контроля, своевременной коррекции действий, выдачи рекомендаций и разрешения конфликтных ситуаций. Как и в первом, так и во втором случае совсем не обязательно, чтобы лидер был опытным программистом. Конечно, роль программиста в проекте, посвященном разработке приложений чрезвычайно важна. В таком случае руководитель организует остальных членов команды максимально помогать программисту, обеспечивая его рабочими материалами: текстами, картинками, макетами дизайна, подробным описанием требуемых функций, а также помогая в отладке и тестировании и апробации приложения в процессе создания.

Если проект выполняется одним человеком, то придётся все эти роли сочетать в себе. С одной стороны, это даст возможность попробовать все основные роли. Такая постоянная смена деятельности позволит понять, что является вашими самыми сильными сторонами. Современная отрасль разработки программного обеспечения очень разнообразна по спектру востребованных специальностей. Не надо огорчаться, если программирование не является вашей самой яркой способностью. Это вполне может компенсироваться креативностью или организаторскими способностями. Но в любом случае, уметь программировать надо. Без этого нельзя ни правильно поставить задачу другим программистам, ни проконтролировать их результаты. Также это важно при генерации идей и постановке задач. Есть ли смысл придумывать что-то новое, если это нельзя будет реализовать? Проработка материала книги очень полезна каждому, вне зависимости от его намерений стать именно разработчиком программного обеспечения.

Книга разбита на главы, каждая из глав практически пошагово описывает реализацию конкретного приложения, посвященного учебной тематике. По завершению каждой главы у нас должно появиться реальное работающее приложение. Основной акцент сделан на мобильной разработке, т.е. созданию приложений либо под смартфон, либо под планшет. Такой подход выбран исходя из стремительно растущей популярности мобильных устройств. Вам будет легко продемонстрировать созданное приложение одноклассникам, друзьям и знакомым. Также просто будет начать применение приложения в обычной жизни, т.к. практически каждый современный человек обладает каким-либо мобильным устройством. Представленные приложения это — не просто некие учебные примеры, они потом могут быть использованы в проектах, относящихся к другим предметам: физике, химии, биологии, математике, литературе. Создание приложения может стать частью другого большого исследовательского проекта.

Именно здесь и заключена самая главная идея книги. Не программирование ради программирования, но программирование для реальной жизни. Созданное приложение может быть использовано в биологическом проекте по наблюдению за ростом растений или в химическом проекте по наблюдению за ростом кристаллов. Приложение для построения графиков функций поможет навсегда разобраться в параболах или гиперболох. Литературное мобильное приложение сделает так, что вы выучите длинную поэму в считанные дни и без ущерба занятиям по другим предметам или вашему отдыху. Программирование меняет нашу жизнь, и вы сможете это доказать личным примером!

В завершении раздела хочется чуть более подробно остановиться на роли руководителя проекта. Не надо путать задачи лидера с программированием или генерацией идей. Работа над проектом должна постоянно приближать всю группу к его успешному завершению. Каждый шаг любого из члена команды должен быть спланирован, выполнен, проконтролирован и зафиксирован. Работа всех участников проекта должна быть синхронизирована. Но не надо сводить свои функции только к раздаче команд, причем в невежливой форме. Если у кого-то что-то не получается, то руководитель

должен уметь вникнуть в суть проблемы и помочь найти её решение. А ещё лучше — уметь избегать проблем за счет подбора команды и тщательного планирования. Оцените силы вашей рабочей группы и выберите проект, адекватный возможностям. Восемь проектов, представленные в книге, весьма разнообразны и позволят подобрать посильную, но интересную задачу. Соберите команду, вдохновитесь одной из описанных идей и приступаете к работе.

1.2. Что такое Delphi/C++Builder/RAD Studio

Мы будем разрабатывать приложения. Можно взять любое одно из восьми представленных проектов. Можно последовательно выполнять проекты один за другим, начиная с первого. Но в любом случае вам придётся воспользоваться средой разработки. В настоящий момент существует достаточное количество инструментов, позволяющих создавать приложения. В конкурентной среде всегда существует несколько близких технологий, дающих приблизительно одинаковый результат. Поскольку основной задачей данной книги является именно обучение программированию мобильных приложений, среда должна быть максимально простой и дружелюбной. Не каждый профессиональный инструмент может этим похвастаться. Если говорить об учебных средах, то чаще всего они позволяют создавать лишь «игрушечные» приложения в рамках ученических проектов. Но мы не видим смысла погружаться в игровую среду ради некой «поделки», а потом переучиваться для реальной разработки. Мы выберем такой инструмент, который будет сочетать в себе доступность для начинающих и возможности с точки зрения профессиональной разработки. Также выбранная среда должна позволять создавать мобильные приложения.

Безусловно лидирующим инструментом, соответствующим предъявленным требованиям, является Delphi/C++Builder/RAD Studio. Это группа продуктов изначально создавалась корпорацией Borland, но уже давно права на создания перешли к компании Embarcadero. Компания Embarcadero является основным производителем указанных сред

разработки, хотя и сейчас можно встретить упоминание Borland, особенно в связи с Delphi 7 или Turbo Pascal.

Turbo Pascal — в своё время, а это в начале 90-х годов прошлого века, была одной из самых популярных и чуть ли не единственной развитой средой разработки на языке Pascal. По ней было написано много книг, и она стала «родной» для обучения программированию для большого числа прикладных программистов. Сейчас её уже сложно встретить даже в учебных заведениях.

Delphi — среда визуальной разработки, в основу которой положен язык Object Pascal. Язык Object Pascal является «старшим братом» классического языка Pascal. Если вы знаете Pascal, то у вас уже есть значительные преимущества! Одной из самых ярких и мощных версий стала Delphi 7 от компании Borland. Вполне вероятно, что именно её вы уже изучили. Она вполне подходит для учебного процесса, но мобильные приложения в ней создавать нельзя. Современные версии Delphi поколения XE не только более развитые в плане удобства использования средой разработки, а также значительно улучшенным языком Object Pascal, они позволяют создавать приложения под Microsoft Windows, Mac OS, iOS и Android.

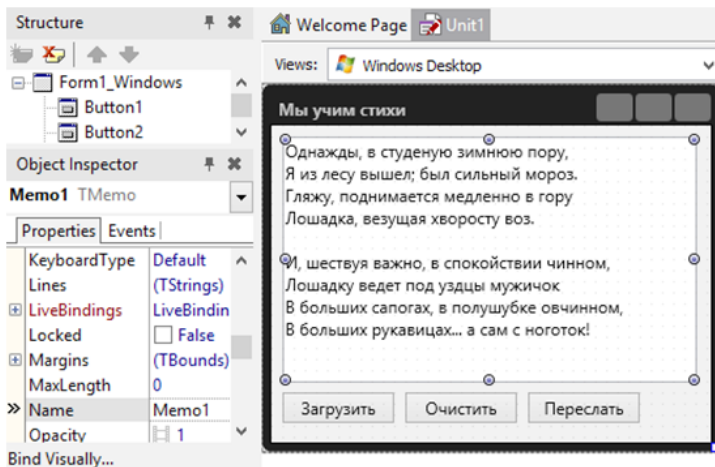
C++Builder — среда визуальной разработки, в основу которой положен язык C++. Язык C++ создан для профессионалов, поэтому лучше не начинать с него изучение программирования, если вы не готовы посвятить ему много сил и времени. Такое решение целесообразно, когда есть твёрдое желание стать профессиональным программистом. Однако C++Builder очень подходящая среда для начала изучения C++. Из предыдущего поколения можно отметить C++Builder 6, вышедший одновременно с Delphi 7. Но, как и Delphi 7, данная популярная версия позволяет создавать приложения только под ОС Microsoft Windows. C++Builder современного поколения XE по возможностям соответствует Delphi и также подходит для разработки мобильных приложений.

RAD Studio — продукт, включающий в себя и Delphi, и C++Builder. Очень часто профессиональные разработчики используют и Delphi, и C++Builder. Если речь идёт о покупке, то в этом случае нужно приобретать как Delphi, так и C++Builder. Проще и дешевле купить RAD Studio и пользоваться Delphi, и C++Builder, активировав их единым серийным номером. В учебных целях весьма полезно иметь RAD Studio, т.е. два этих мощных продукта одновременно. Delphi однозначно самая эффективная среда для изучения программирования, тогда как C++Builder отличный инструмент для изучения C++. С точки зрения среды разработки, методов визуального программирования, навыков разработки мобильных приложений эти продукты идентичны. Если вы умеете пользоваться Delphi, то перейти на C++Builder не составит труда. Единственное, что будет новым это — синтаксис языка C++. Но вы как минимум не потеряете время на освоение новой среды.

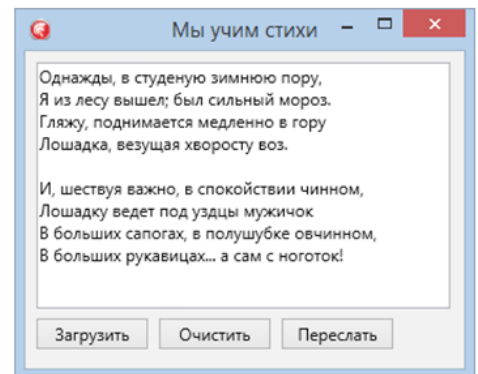
Delphi и C++Builder являются средами визуальной разработки приложений. Мы знаем, что для разработки программ или приложений нужно программировать, т.е. вводить программный код. Программный код представляет собой текст, написанный в соответствии с правилами конкретного языка. В визуальных средах не всё нужно программировать в виде текста, некоторую часть работы берёт на себя визуальный редактор. Он не является в полной мере графическим редактором, но принципы работы практически такие же. Интерактивными манипуляциями мы «рисует» или «моделируем» интерфейс пользователя, который не является «просто картинкой». Эта «картинка» после сборки проекта «оживает», превращаясь в интерфейс реального приложения. Итак, среда визуальной разработки сочетает в себе качества обычной программной среды, где нужно вводить код с клавиатуры, с визуальным редактором, создающий графические композиции интерактивно при помощи мыши.

Графическая композиция, которую создает разработчик в визуальной среде, это — интерфейс пользователя. Как мы «нарисуем» интерфейс, таким он и будет (рис. 1.1). Такой принцип был реализован в первой версии Delphi практически два десятилетия назад. Теперь визуальное создание интерфейсов является стандартом для любой профессиональной среды

разработки. Создание интерфейса визуальными средствами существенно экономит время. Но, наверное, нет смысла особым образом убеждать читателя использовать визуальные средства, такие как Delphi. Сейчас уже трудно встретить средства разработки, не обладающими средствами визуального прототипирования интерфейса. Среда C++Builder появилась несколько позже, но в основе лежит абсолютно та же технология. Поэтому во многих местах в книге будет встречаться перечисление сред через косую черту: Delphi/C++Builder/RAD Studio. Это будет означать, что и в Delphi, и в C++Builder, и в RAD Studio можно выполнить описанное действие или воспользоваться представленной возможностью.



Дизайн интерфейса при
создании приложения



Интерфейс работающего
приложения

Рис. 1.1. Визуальная разработка интерфейсов.

Принцип визуальной разработки остался неизменным, начиная с первой версии Delphi 1. С тех пор вышло уже очень много версий, рассмотрим кратко эволюцию Delphi как основной среды, а C++Builder сначала запаздывал по появлению новых возможностей, но сейчас они уже практически полностью сравнялись.

Delphi 1–7 — период зарождения и бурного развития среды разработки. Версия к версии добавлялись новые мощные возможности, но уже к версии 7 практически направления роста были исчерпаны с точки зрения разработки под Microsoft Windows. Даже сейчас Delphi 7 считается одной из самых лучших инструментов для разработки под данную операционную систему.

Delphi 8, 2006/7, 2009, 2010, XE — весьма сложный период для данного инструмента разработки приложений. Были предприняты различные и порой неудачные попытки придумать что-то новое. Компания Borland, которая тогда ещё владела технологией, весьма бессистемно добавляла различные сервисы и реализовывала функции. Но всё это проходило на фоне резкой потери стабильности среды и качества компилятора. Действительно полезным стала поддержка Unicode в версии 2009, но этот релиз выпустила уже компания Embarcadero. Embarcadero приобрела все права на Delphi у Borland, и с этого момента усилия по развитию продукта стали более продуманными и спланированными. Отдельно можно сказать про среду Kylix для разработки под операционную среду Linux. Но данная операционная система не обрела популярности, сравнимой с Microsoft Windows, поэтому судьба Linux сложилась невыразительно.

Delphi XE2 — XE8 — период начала и развития мульти-платформенных возможностей среды разработки. Не зря мы только что упомянули Linux, как еще одну целевую операционную систему, помимо Microsoft Windows. Как Borland, так и Embarcadero стремились выйти за рамки Microsoft Windows, но не было «достойных кандидатов». ОС Linux не оправдала надежд. К моменту планирования поколения XE явно выделялась Mac OS от компании Apple в качестве достойного конкурента доминирующей Microsoft Windows. Было принято решение поддержать именно Mac OS, что с успехом и было реализовано в версии XE2. После этого произошло второе рождение Delphi, а в дальнейших релизах была обеспечена возможность разработки приложений под iOS и Android. Главное, что приложения можно создавать на основе единого исходного кода. Мы создаём один проект, а сборку можно выполнять под различные платформы. Все эти платформы — Microsoft

Windows, Mac OS, iOS и Android будут существовать еще долгие годы, поэтому как в учебной, так и профессиональной среде нужно уметь создавать приложения под все эти операционные системы. Современная Delphi единообразно покрывает все четыре основные современных платформы.

Какой версией Delphi/C++Builder/RAD Studio нужно пользоваться? Конечно, самой последней. На момент написания книги таковой является XE8, но Embarcadero взяла достаточно жёсткий темп выпуска новых версий, поэтому в скором времени мы ожидаем новую версию. Почему важно использовать именно самую последнюю версию? Одной из главных причин является стабильность среды разработки. От версии к версии компания Embarcadero старается минимизировать количество ошибок или «глюков» в среде. Если вы будете использовать последнюю версию, то у вас точно будет самая мощная по возможностям и, вероятно, наиболее стабильная среда. Конечно, в рамках новых возможностей могут быть некоторые особенности, требующие особого отношения, но как минимум в данной книге мы будем беречь читателя и демонстрировать только проверенные методы разработки.

Материал книги можно использовать разными способами. В идеальном случае читатель последовательно прорабатывает все проекты, доводя их до работающего приложения. Если есть опытный преподаватель или руководитель в ранге учителя, то можно выбрать отдельный проект и сконцентрировать усилия на нём. В случае, когда базовые возможности Delphi не вызывают проблем, то выполнение проектов может сильно продвинуть читателя в область мобильной разработки с акцентом на Android. Ситуация такая же, если читатель знает классические версии C++Builder, поколения до XE2. Можно использовать данную книгу и для C++Builder. Единственно, что придётся переосмыслить и переработать, это — исходный код. Object Pascal при всём нашем уважении к нему всё-таки является более простым языком программирования, поэтому проблем с переводом его на C++ нет. Практически нет таких конструкций в Object Pascal, которые невозможно было бы выразить на C++. А поскольку аудитория пользователей Delphi значительно шире, чем C++Builder, умение переводить с Object Pascal на C++ является весьма полезным и даже необходимым навыком. Но в каче-

стве бонуса вы так или иначе будете уметь программировать и на C++, и на Object Pascal.

1.3. Из чего состоит и как работает Delphi

Можно выделить главные составные части Delphi с точки зрения её работы:

- Интегрированная среда разработки IDE (integrated development environment)
- Компилятор
- Библиотеки визуальных компонентов.

Интегрированная среда разработки IDE сама представляет собой приложение Win32, что означает её работоспособность в операционных системах семейства Microsoft Windows. Чтобы начать создавать программы нужно установить Delphi так, как мы устанавливаем любое другое приложение под Microsoft Windows. Установке Delphi/C++Builder/RAD Studio будет посвящен отдельный раздел. Запущенное приложение Delphi выглядит так, как показано на рис. 1.2:

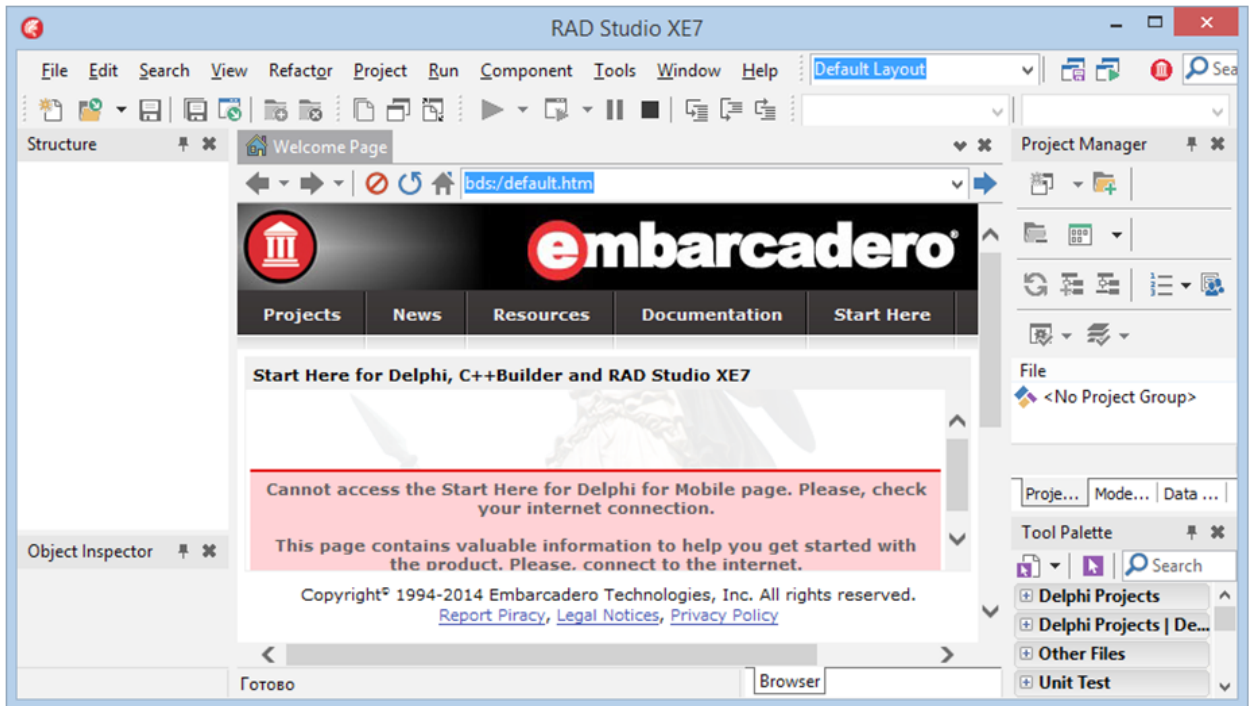


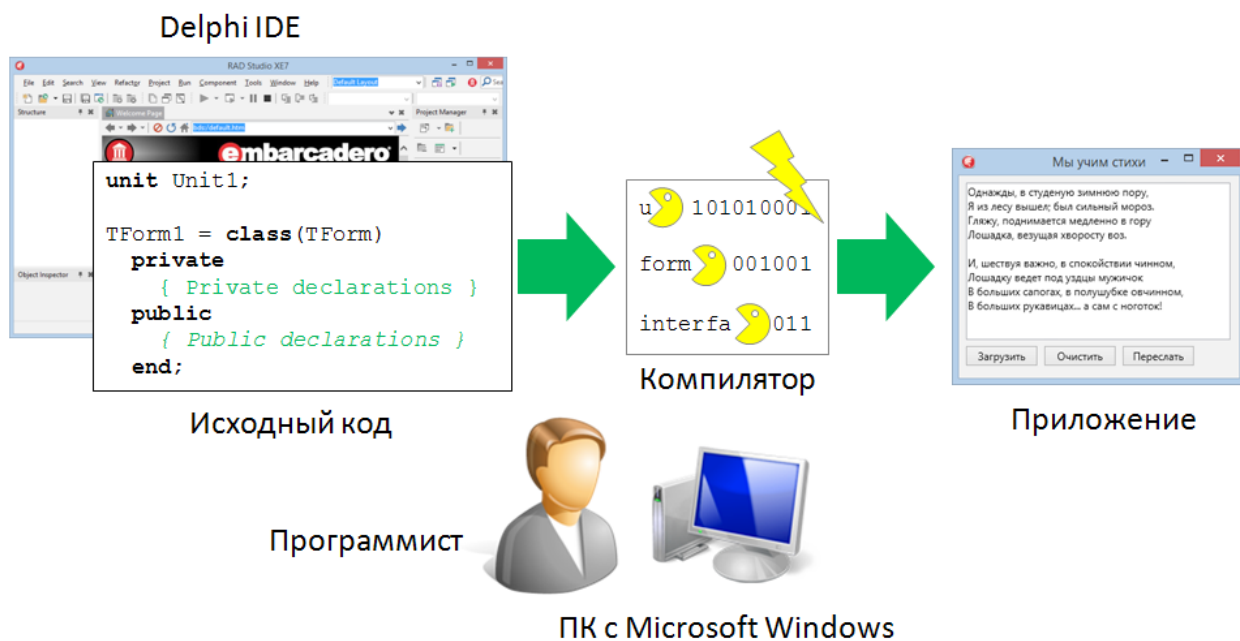
Рис. 1.2. Вид Delphi IDE после запуска

Если вы пользуетесь предыдущими версиям Delphi (что не рекомендуется), то среда может выглядеть иначе. Кроме того, если ваш компьютер подключен к сети интернет, то на стартовой страничке буду показаны новости компании Embarcadero, ссылки на обучающие видео-ролики и т.д. Среда IDE и есть то пространство, в котором программисты проводят своё активное рабочее время.

В интегрированной рабочей среде программисты создают программы. Происходит это следующим образом. Сначала создаётся «исходный код» в виде текста. Ввод текста программы или исходного кода ничем не отличается от работы в текстовом редакторе. На заре эпохи разработки программ для ввода текста использовали обычные текстовые редакторы. Программист вводил код, сохранял его в виде файла, а затем его преобразовывали в приложение. В дальнейшем текстовые редакторы, предназначенные для ввода исходного кода, получили функции, нужные только для программистам.

стов. Они стали отличаться от обычных редакторов типа Microsoft Word, Wordpad или Notepad. Эти функции коснулись удобства ввода и манипулирования исходным кодом.

При помощи редактора кода как части IDE (произносится как «ай-ди-и») вводится текст программы и сохраняется в файл. До этого момента разработка программы ничем не отличается от создания обычного текстового документа. Но затем включается вторая главная часть Delphi — компилятор. Компилятор — это тоже приложение Win32, т.е. приложение, которое может выполняться в операционных системах семейства Microsoft Windows. Компилятор перерабатывает исходный текст программы в исполняемый код. Исполняемый код попадает в файл, который имеет расширение «exe» для Windows-приложений. Файл с расширением «exe» может запускаться независимо от среды разработки. Схематично процесс создания приложения показан на рис. 1.3.



Создание большого серьезного приложения происходит несколько сложнее. Исходный код храниться не в одном файле, а в нескольких. Компилятор по очереди компилирует файлы исходного кода. Потом происходит так называемая «сборка», когда из нескольких переработанных кусков создаётся единый файл приложения. На этапе сборки добавляются ресурсы (некоторая дополнительная информация, необходимая для работы приложения). Для Microsoft Windows финальным результатом сборки является файл «exe», который можно копировать на другие компьютеры и там запускать. Для других операционных систем результатом сборки являются «пакеты», соответствующие понятию «исполняемое приложение». Например, для Android таким пакетом является файл с расширением «apk». Его также можно скопировать на смартфон или планшет, произвести установку приложения, а затем запускать и работать с ним.

Современная Delphi, как и C++Builder снабжены несколькими компиляторами. Для каждой из поддерживаемых операционных систем поставляется отдельный компилятор. Перечислим их:

- Компилятор для создания приложений Win32 (для 32-битных операционных систем или режимов поддержки исполнения 32-битного кода Microsoft Windows);
- Компилятор для создания приложений Win64 (для 64-битных операционных систем Microsoft Windows);
- Компилятор для создания приложений под Mac OS;
- Компилятор для создания приложений под iOS;
- Компилятор для создания приложений под Android.

Сначала Delphi обладала лишь одним компилятором для создания приложений Win32. Впоследствии, начиная с версии XE2, вместе с Delphi стали поставляться компиляторы для создания приложений и для других операционных систем. Следует иметь ввиду, что сама Delphi и её компиляторы являются приложениями Win32, т.е. для их работы нужен как минимум один ПК с операционной системой Microsoft Windows. Если мы собираемся

создавать только приложения под Microsoft Windows, то нам достаточно одного ПК. Для разработки приложений под другие операционные системы нужны соответствующие устройства. Есть варианты использования так называемых «эмуляторов», которые создают виртуальное устройство в Microsoft Windows. Однако практика показала, что производительность и качество работы таких «виртуальных» устройств на обычных ПК не выдерживает всякой критики по сравнению с реальными устройствами. Кроме того, создание удобного интерфейса для, например, мобильных приложений требует смартфона или планшета, т.к. иначе сложно полностью оценить качество интерфейса пользователя.

1.4. Библиотеки визуальных компонентов

Среда IDE нужна для создания исходного кода. Компилятор перерабатывает исходный код, а затем собирается результирующее приложение или установочный пакет в зависимости от целевой операционной системы или платформы. Мы не будем подробно останавливаться на процессе сборки, т.к. он происходит автоматически без необходимости вмешиваться и исправлять ошибки. Обычно параметры сборки приложения назначаются самой средой, поэтому задумываться над ними не приходится. Однако на этапе зарождения интегрированных сред разработки параметры сборки приходилось задавать вручную, вводя текстовые параметры практически так, как вводится исходный код.

В современных средах разработки, таких как Delphi, ручное кодирование или ввод исходного текста сведён к минимуму. Интерфейсы приложений, как настольных, так и мобильных, не кодируются в виде текста (а это имело место порядка 20 лет назад), а буквально «рисуются». Среда Delphi одной стороны похожа на текстовый редактор, когда речь касается ввода исходного кода. С другой — на графический редактор, позволяющий рисовать «картинки». Только в нашем случае картинки будут «оживать» при запуске приложений. Об этом мы уже говорили ранее, но не объясняли, как это происходит. Происходит процесс визуального создания интерфей-

са приложения следующим образом. Когда мы «рисует» интерфейс путём размещения на окне приложения, которое в Delphi называется «формой», различные визуальные компоненты или элементы управления, то автоматически генерируется программный код, а также файл, в котором хранится текстовое описание дизайна.

Каждая форма в Delphi описывается в двух файлах. Первый файл имеет расширение «pas» от слова Pascal, т.к. Object Pascal является языком программирования в Delphi. Второй файл имеет расширение «dfm» или «fmx». В первом случае название происходит от Delphi Forms, во втором — от FireMonkey. Схематично данный процесс показан на рис. 1.4.

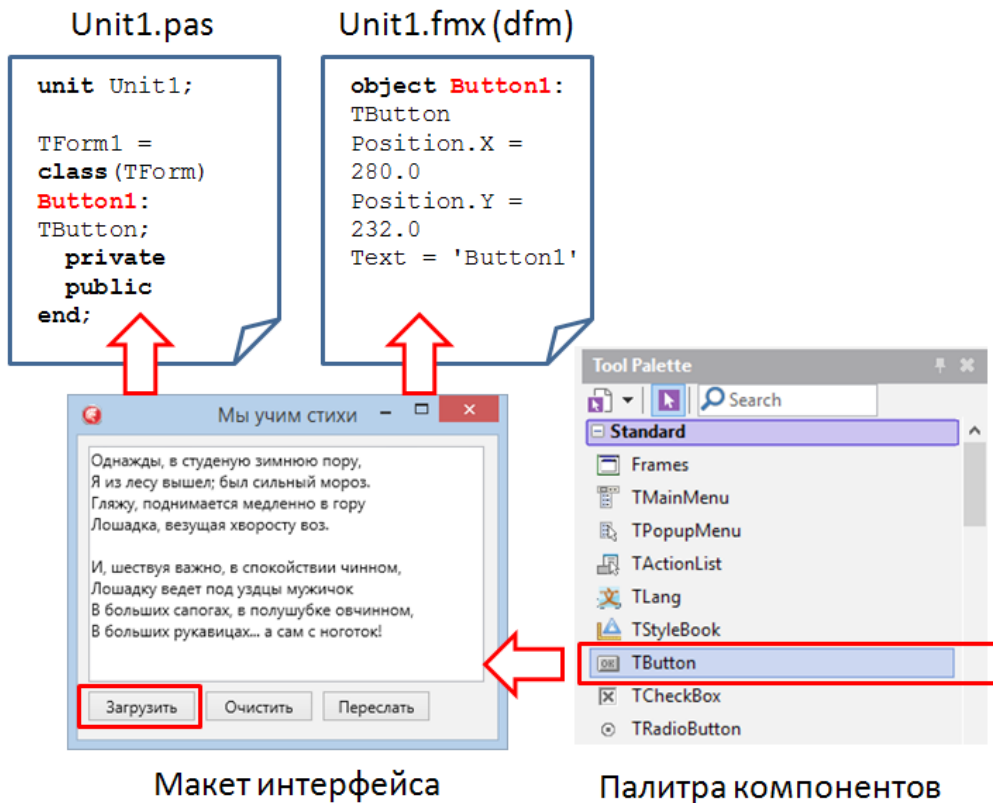


Рис. 1.4. Создание интерфейса визуальными методами

Компоненты на форму помещаются из «палитры компонентов» (ToolPalette). Визуальные компоненты называются «визуальными», т.к. они выглядят одинаково и в режиме разработки, и в режиме запуска готового приложения. Палитра отображает компоненты, которые находятся в «библиотеке». Раньше в Delphi была только одна библиотека VCL — visual component library. Она была ориентирована строго на создание Windows-приложений. Когда встала необходимость добавить в Delphi возможность создавать приложения и под другие операционные системы (их также называют «платформами»), то перед компанией Embarcadero встала проблема трансформации библиотеки VCL так, чтобы её компоненты могли быть использованы и для Mac OS, iOS и Android. Трансформация не получилась, и был выбран другой путь — создание новой, мульти-платформенной библиотеки, компоненты которой позволяют создавать интерфейсы сразу под несколько операционных систем. С этой задачей инженеры Embarcadero успешно справились, а Delphi и впоследствии C++Builder начали поставляться с новой библиотекой FireMonkey. В дальнейшем от бренда FireMonkey отказались в пользу «платформа FM».

Теперь если мы хотим создать приложение только под Microsoft Windows, то создаём проект приложения на основе VCL. Если мы хотим, чтобы создаваемое приложение можно было откомпилировать и собрать не только под Windows, но и Mac OS, iOS, а также Android, то нужно начинать проект на основе FM. При создании нового проекта приложения в Delphi или C++Builder:

- VCL Forms Application (только под Microsoft Windows)
- Multi-Device Application (под Microsoft Windows, Mac OS, iOS и Android)

Можно перечислить основные компоненты, из которых создаются интерфейсы приложений: кнопки, поля для ввода и редактирования, метки (надписи), панели, поля для многострочного ввода и т.д. Их названия с точки зрения программного кода соответствуют английским словам: TButton, TEdit, TLabel, TPanel, TMemo. По давней традиции со времён ещё первой

версии Delphi 1 в начало добавляется символ «Т». И в библиотеке визуальных компонентов VCL, и в мульти-платформенной FM есть идентичные по названию компоненты, часть их которых мы уже перечислили.

До появления FM и, как следствие, возможности создавать приложения не только под Microsoft Windows, возникла потребность конвертировать уже созданные проекты с VCL на новую платформу. Но при полном совпадении названий компонентов внутренние различия настолько велики, что в автоматическом режиме это крайне затруднительно. Существуют независимые решения, выполняющие данное преобразование на уровне исходного кода, однако их эффективность сильно зависит от сложности проекта. Если интерфейс очень сложный, то трансформация даже при наличии такого преобразователя все-таки требует кропотливого ручного труда. Поэтому, начиная новый проект, лучше сразу решить, будет ли это приложение ориентировано только на Microsoft Windows или сразу на целый спектр операционных систем. Поскольку данная книга посвящена разработке мобильных приложений, мы будем создавать проекты типа Multi-Device Application и использовать платформу FM.

У многих читателей вероятно уже есть опыт работы с Delphi по созданию VCL-приложений. В таком случае использование платформы FM не вызовет особых проблем, тем более, что Object Pascal в своей основе будет применяться идентичным способом. Сложность может вызывать проектирование интерфейсов мобильных приложений, т.к. придётся использовать визуальные компоненты, которых не было в VCL. Но в целом мобильные приложения с точки зрения интерфейса пользователя гораздо проще, поэтому после реализации небольшого числа проектов, описанных далее, вы сможете самостоятельно использовать платформу FM. Умение создавать мобильные приложения привычным для Delphi-программистов способом стоит того, чтобы освоить эту новую библиотеку.

1.5. Где взять Delphi с возможностью мобильной разработки

Прежде всего нужно посетить сайт компании Embarcadero по ссылке www.embarcadero.com, где можно ознакомиться с перечнем средств разработки и систем для работы с базами данных этого производителя. В разделе Products->Application Development можно найти Delphi/C++Builder/RAD Studio в категории Rapid Application Development Tools (быстрая разработка приложений). Также можно увидеть ссылки на скачивания бесплатных пробных версий (FREE TRIALS), однако следует знать особенности лицензионной политики компании Embarcadero.

На момент написания книги Embarcadero предоставляет разработчикам средства быстрой разработки Delphi/C++Builder/RAD Studio согласно различным лицензиям:

- Коммерческая
- Образовательная
- Пробная бесплатная

Коммерческая лицензия даёт право использовать выбранный продукт в различных целях для разработки без ограничений. Это может быть свободная продажа созданных программных продуктов, а также их использование внутри коммерческих и некоммерческих организаций. Последнее является очень важным — даже если вы не собираетесь продавать приложения, а использовать их внутри школы, института, государственного учреждения или коммерческой компании, лицензия все равно должна быть «коммерческой». Здесь вид организации роли не играет, единые правила работают для всех.

Образовательная лицензия даёт право использовать выбранное средство разработки в образовательных целях. Ограничением является то, что Delphi/C++Builder/RAD Studio или любой другой продукт используется в рамках учебного процесса по курсу, соответствующему назначению про-

дукта. В рамках курса «информатика», «программирование», «разработка приложений» применение Delphi согласно образовательной лицензии допустимо. Если выбранным продуктом является ER/Studio, то допустимо использовать его в курсах «моделирование», «разработка баз данных», «основы проектирования программных средств». При выборе типа лицензии на конкретный продукт лучше всего проконсультироваться непосредственно с представителями компании Embarcadero. Можно воспользоваться следующими контактами:

- Представительство Embarcadero по России и странам СНГ:
Телефон: +7 495 708–43–93
E-mail: russia.info@embarcadero.com
- Менеджер по работе с образовательными учреждениями,
Embarcadero:
Сергей Терлецкий
Телефон: +7 495 708–43–93
E-mail: sergey.terletskiy@embarcadero.com

На момент написания книги цена образовательной лицензии составляет 10% от стоимости коммерческой. Также на каждую приобретенную лицензию в рамках учебного процесса по соответствующему курсу образовательная организация получает дополнительно 10 лицензий для обеспечения возможности учащимся выполнять домашнее задание в удобном для них месте.

Бесплатная пробная версия имеет следующие ограничения:

- Действует ограниченный период времени
- Даёт право лишь на ознакомление с возможностями продукта

Типичной ошибкой, которая представляет собой прямое нарушение лицензионного соглашения, является использование бесплатной пробной лицензией в образовательных целях. Конечно, можно загрузить trial-вер-

сию Delphi и попробовать свои силы в разработке приложений. Однако систематическое использование в учебных целях, например, для выполнения проектов, описанных в данной книге, требует наличия образовательной лицензии.

Вполне допустимым является приобретение и использование образовательной лицензии для обучения школьников или студентов разработке мобильных приложений. Преподаватели также могут пользоваться данным типом лицензии. Но если образовательный проект перерастает в серьёзное приложение, которое начинает использоваться, например, в качестве обучающей среды для других предметов, предоставляется другим учебным организациям или распространяется через «магазины приложений», то здесь требуется коммерческая лицензия. Если у вас есть сомнения, лучше проконсультироваться с представителем Embarcadero.

Скачать и установить бесплатную пробную лицензию можно с сайта Embarcadero, ссылка на который указана выше. Для скачивания и установки вам придётся создать учётную запись (так называемый EDN-account), что не займёт много времени. При создании учётной записи следует указать корректный адрес электронной почты, т.к. именно он будет использован для отправки серийного номера. Серийный номер потребуется при установке продукта. Не следует вводить вымышленное имя, т.к. учётная запись будет однозначно идентифицировать вас как пользователя продуктов Embarcadero. При возникновении вопросов об авторстве созданного вами программного продукта или подтверждения правильности использования выбранной лицензии будет использоваться то имя, которое вы введёте.

Бесплатная пробная версия загружается автоматически при переходе по соответствующей ссылке на сайте. Загрузка требует подключения к интернет. Пока скачивается дистрибутив, вы проверяете электронную почту согласно адресу, введенного вами при создании учётной записи. На данный адрес должно прийти электронное письмо от Embarcadero, содержащее серийный номер для активации продукта. Данное письмо создаётся автоматически, поэтому в случае его отсутствия нужно проверить ящик для спам-сообщений. Установка пробной версии обычно не вызывает проблем,

достаточно своевременно нажимать кнопки «Next» (далее). Конечно, для экономии дискового пространства можно ограничить возможности устанавливаемой пробной версии, но для этого необходимо знание компонентов продукта. Если вы не уверены, с какими возможностями пробной версии вы хотите ознакомиться, выполняйте установку по-максимуму.

1.6. Установка пробной версии по шагам

В предыдущем разделе мы в общих чертах рассмотрели процедуру установки пробной версии Delphi. Для пользователей, знакомых с Delphi/C++Builder/RAD Studio этого будет достаточно. Для другой категории читателей опишем процесс установки по шагам:

Шаг № 1. Запустим браузер и введем в адресную строку: www.embarcadero.com/ru/products/rad-studio, затем кликнем на кнопку «Пробная версия» (рис. 1.5).

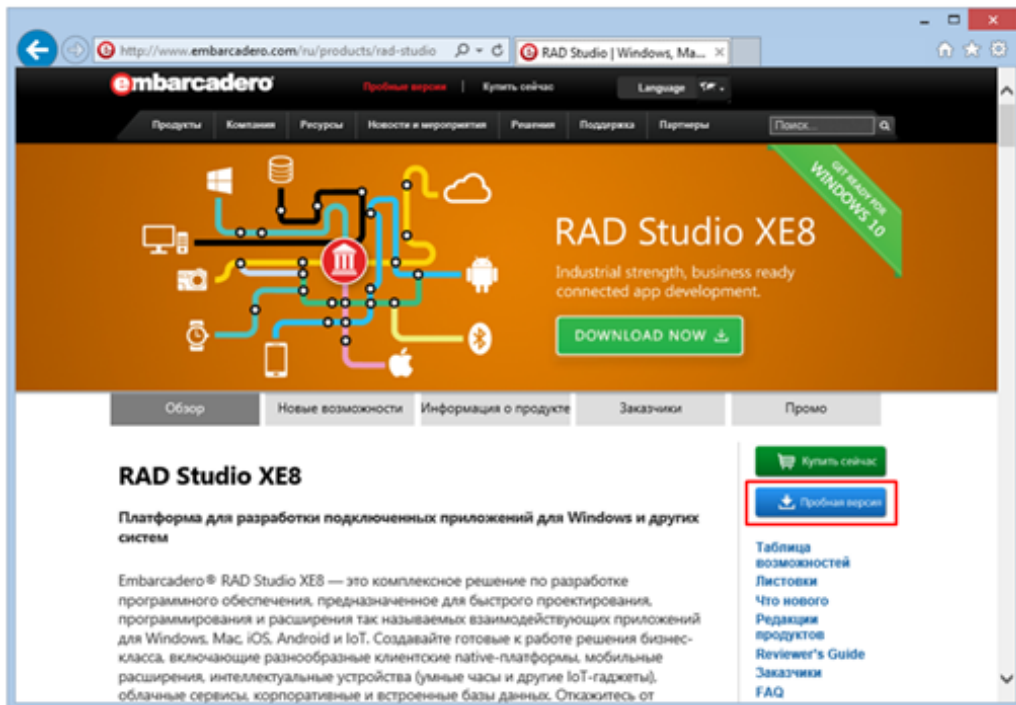


Рис. 1.5. Страница для скачивания пробной версии

Шаг № 2. После нажатия на кнопку «Пробная версия» на предыдущем шаге появится окно с предложением ввести параметры учётной записи. Если вы уже имеете учётную запись в Embarcadero Developer Network (edn.embarcadero.com), то введите её параметры (рис. 1.6, левая часть) и начните загрузку. Если у вас такой учётной записи нет, то кликните ссылку Register в правом верхнем углу появившегося окна, и оно увеличит свой размер. Теперь можно ввести свои данные и нажать DOWNLOAD TRIAL. Учётная запись Embarcadero Developer Network создастся автоматически. Потом уже независимо от процесса установки можно по адресу выше и ссылке LOG ON в правом верхнем углу в свой «личный кабинет». Там содержится много полезной для разработчика информации. С помощью своей учётной записи впоследствии можно получить доступ к различным ресурсам на сайте Embarcadero.

После нажатия DOWNLOAD TRIAL начнётся загрузка установщика после того, как разрешите браузеру это сделать (рис. 1.7). Можно нажать кнопку «Выполнить», тогда сразу после скачивания установщика будет выполнен его запуск. Установщик скачает на ваш компьютер инсталляционный пакет продукта.

FREE TRIAL DOWNLOAD
NO CREDIT CARD NEEDED

Not registered? [Register](#)

Email

Password

Product: RAD Studio
Version: XE8 Architect - 30 days trial

DOWNLOAD TRIAL

This trial download may include features and fixes only available with an [Update Subscription](#). If you purchase this product without an Update Subscription, you will receive the original release version.

FREE TRIAL DOWNLOAD
NO CREDIT CARD NEEDED

Already registered? [Login](#)

First Name

Last Name

Email

Password

Verify password

Company

Phone

Country

Product: RAD Studio
Version: XE8 Architect - 30 days trial

DOWNLOAD TRIAL

Рис. 1.6. Ввод или создание учётной записи

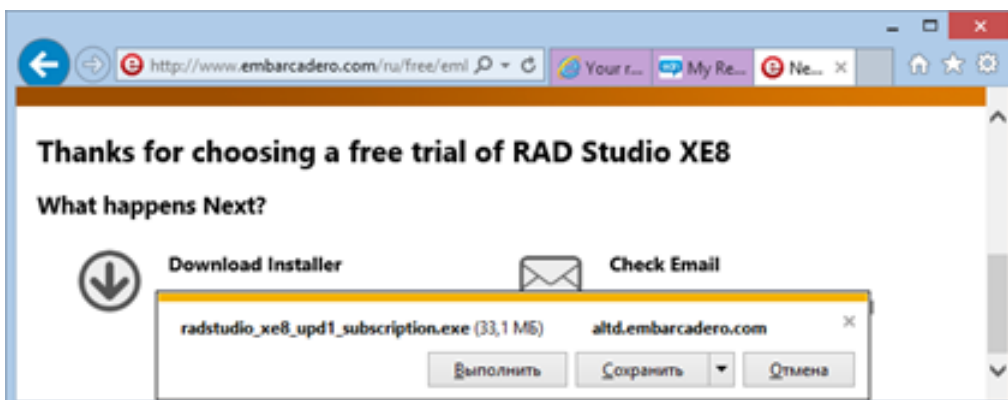


Рис. 1.7. Разрешение загрузки установочного пакета

Шаг № 3. После скачивания установщика автоматически начнётся инсталляция продукта, после принятия условий лицензионного соглашения. На экране будут появляться последовательно окна, показанные на рис. 1.9 и 1.10. Красными прямоугольниками отмечено место ввода серийного номера. Серийный номер должен придти по электронной почте в соответствии с введенным адресом в окне, показанном на рис. 1.6. Также красным прямоугольником помечено окно, где выполняется установка «пакета разработчика под Android» (Android SDK).

При необходимости можно задержаться на каждом из окон мастера установки для ознакомления с отображаемым текстом. Большинство опций выбрано по умолчанию, что вполне подходит для начинающих программистов. Кроме того, срок действия пробной версии ограничен по времени, по истечению которого продукт нужно деинсталлировать. В дальнейшем, ознакомившись с основными возможностями продукта при помощи пробной версии, вы уже сможете сконфигурировать аналогичным мастером установки продукт в точном соответствии с вашими требованиями и предпочтениями.

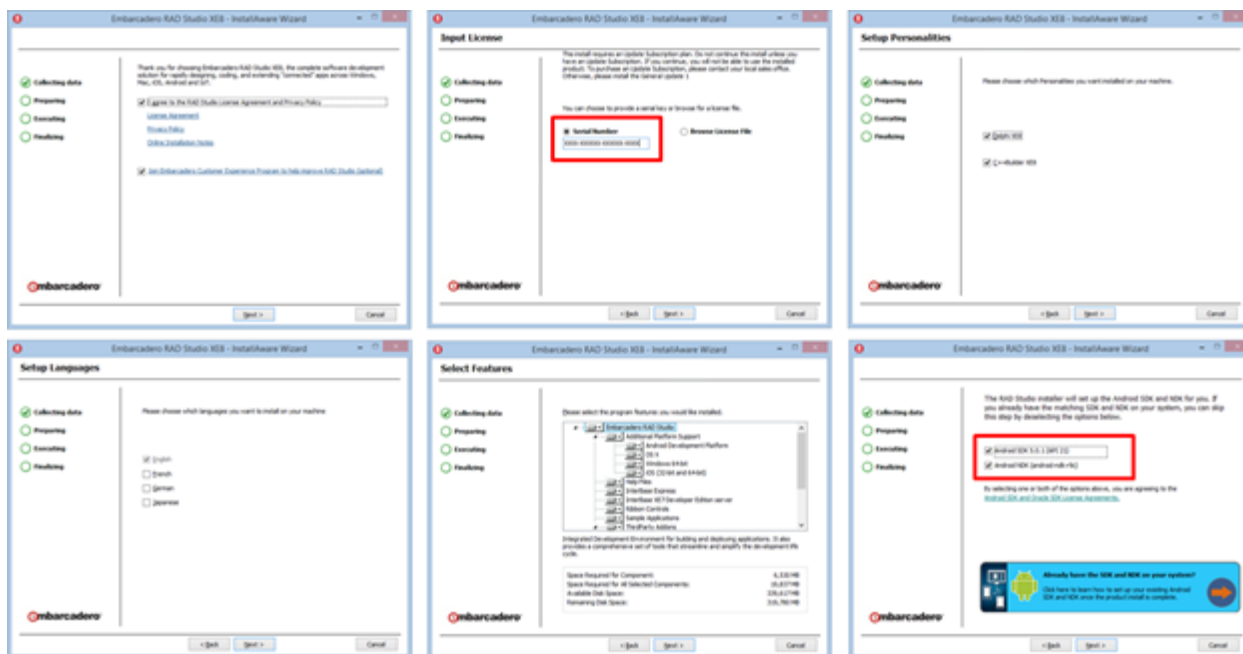


Рис. 1.9. Процесс установки, часть 1

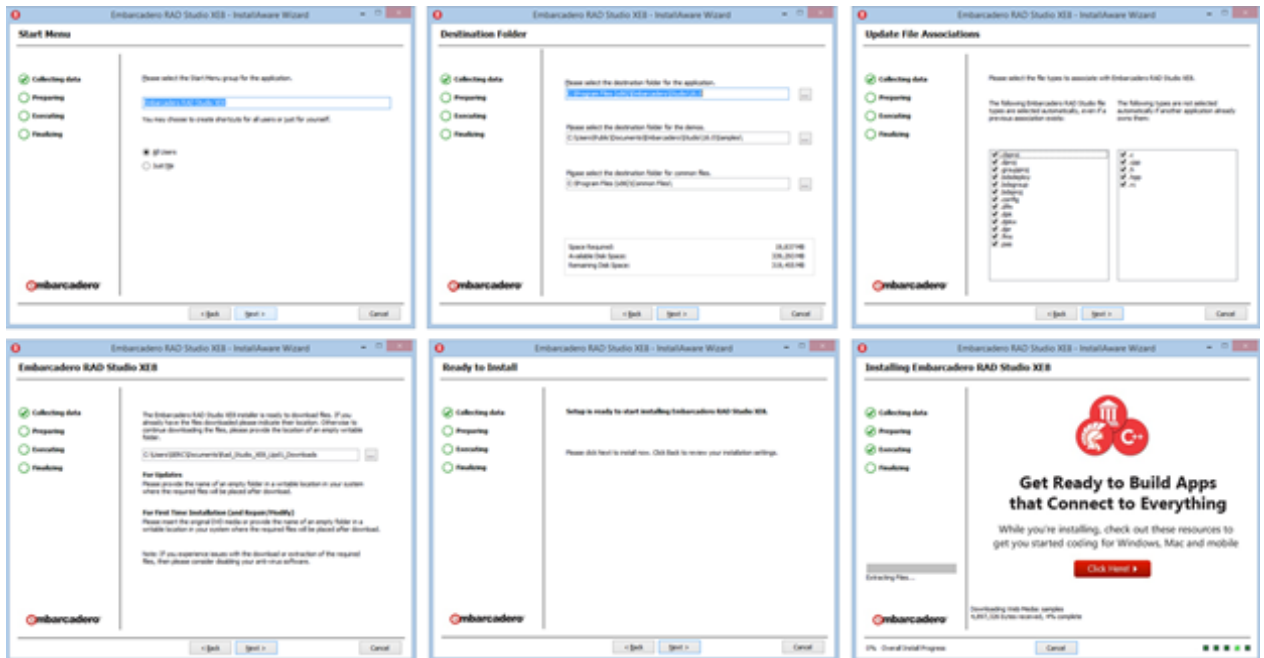


Рис. 1.10. Процесс установки, часть 2

После окончания установки продукт будет полностью готов к работе. Пробные версии Delphi/C++Builder/RAD Studio устанавливаются в самой мощной редакции Architect. Понятие «редакция» продукта будет рассмотрено в следующем разделе.

1.7. Редакции продукта Delphi/C++Builder/RAD Studio

Средства разработки Delphi/C++Builder/RAD Studio выпускаются в нескольких редакциях. Не следует путать «версию» и «редакцию». Версия тесно связана с датой выпуска продукта: чем старше версия, тем новее продукт. На момент написания книги последней версией была XE8. От версии к версии появляется все больше новых возможностей, и устраняются проблемы, найденные в период эксплуатации предыдущих. Если вы используете одну из предыдущих версий, то есть смысл ознакомиться с официальной документацией, описывающей новые возможности, появившиеся в последней

версии: <http://www.embarcadero.com/ru/products/rad-studio/upgrade-from-rad-xe2>. Но не только версия определяет набор возможностей продукта, но и редакция.

В рамках одной версии продукта существует несколько редакций. Мы уже рассмотрели, что Delphi/C++Builder/RAD Studio представляют собой сложные продукты, содержащие набор различных библиотек. Одна и та же версия продукта, например XE8, может содержать различный комплект библиотек. Для простоты представим себе, что речь идёт о билете на пассажирский поезд. Какой бы вы билет не купили — эконом класса, стандартный или повышенной комфортности, поезд отправится в одно и то же время. Но возможности пассажира будут разные. В зависимости от задач программиста ему может быть достаточно и самой простой редакции Delphi/C++Builder/RAD Studio, тогда как некоторые разработчики предпочитают не ограничивать себя в возможностях и использовать самую мощную.

С точки зрения изучения программированию нельзя дать однозначного совета, какую редакцию выбрать. С одной стороны, учащимся нужно твёрдо овладеть самыми базовыми навыками разработки. С другой, на этапе обучения важно охватить максимально большое число технологий для расширения горизонта мышления учащегося в области разработки программного обеспечения. Здесь Delphi/C++Builder/RAD Studio предоставляют значительное преимущества, т.к. даже очень сложные технологии, например, сетевого взаимодействия приложений и программирования многозвенных приложений, имеют компонентную основу. Такие приложения также можно программировать визуальными средствами, т.е. доступными для учеников школ.

Полную документацию по редакциям текущей версии продукта всегда можно узнать на сайте компании Embarcadero, выбрав в главном меню соответствующий продукт и скачав документ «Таблица возможностей» (рис. 1.11). В этом документе показаны различия между редакциями. Современные среды разработки Delphi/C++Builder/RAD Studio обладают очень большим количеством возможностей, разобраться в которых начинающим

разработчикам будет сложно. В таком случае можно обратиться к краткой таблице по ссылке: <http://www.embarcadero.com/ru/products/rad-studio/product-editions>, где также представлены краткие текстовые описания возможностей каждой из редакций.

Можно воспользоваться кратким мнемоническим правилом: чем выше редакция, тем больше возможностей. Редакция Professional позволяет создавать мобильные приложения, включая использование баз данных на смартфонах и планшетах, тогда как в реальной жизни требуется полномасштабная реализация клиент-серверных подключений и для этого случая используют редакцию Enterprise — все возможности Professional, а также компоненты и технологии работы с корпоративными базами данных и создания многозвенных приложений. Architect — все возможности Enterprise, а также средства моделирования баз данных.

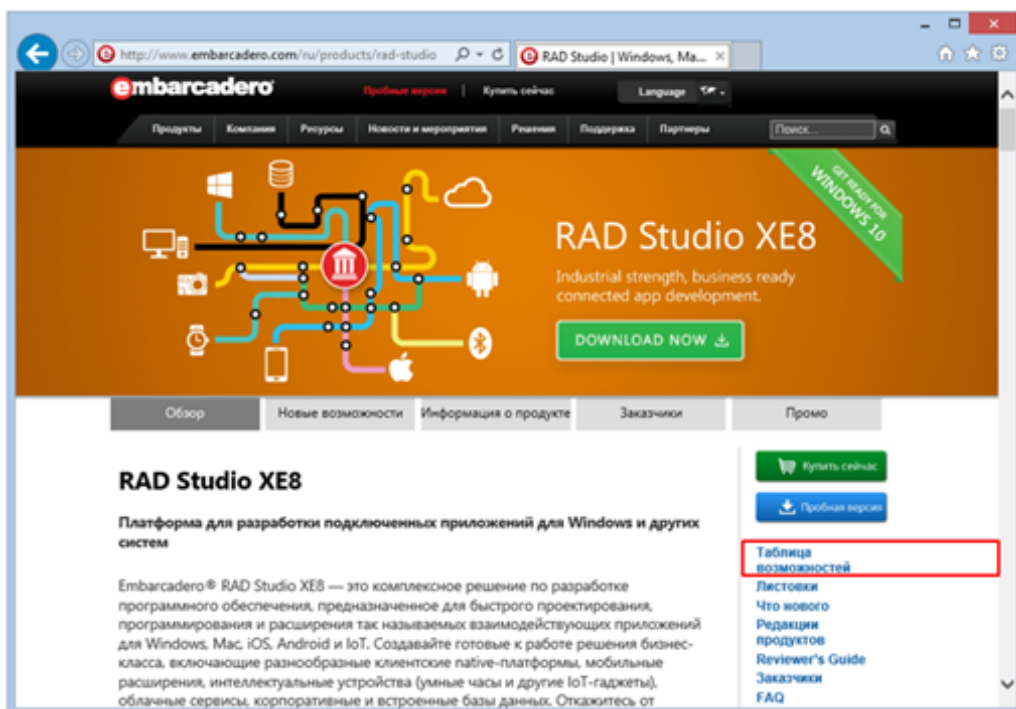


Рис. 1.11. Ссылка на документ «таблица возможностей»

1.8. Подготовка к мобильной разработке

После установки Delphi/C++Builder/RAD Studio полностью готовы к разработке приложений под Microsoft Windows. Если мы работаем на ПК под управлением операционной системы Microsoft Windows 8, то находим и запускаем Delphi XE8 (рис. 1.12).

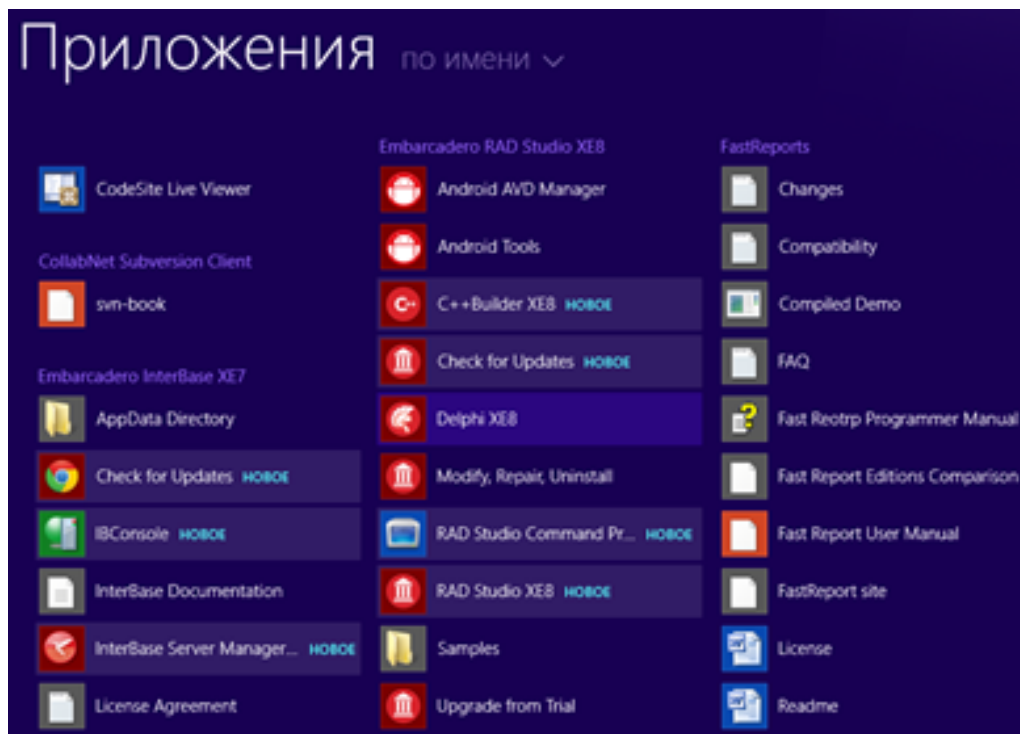
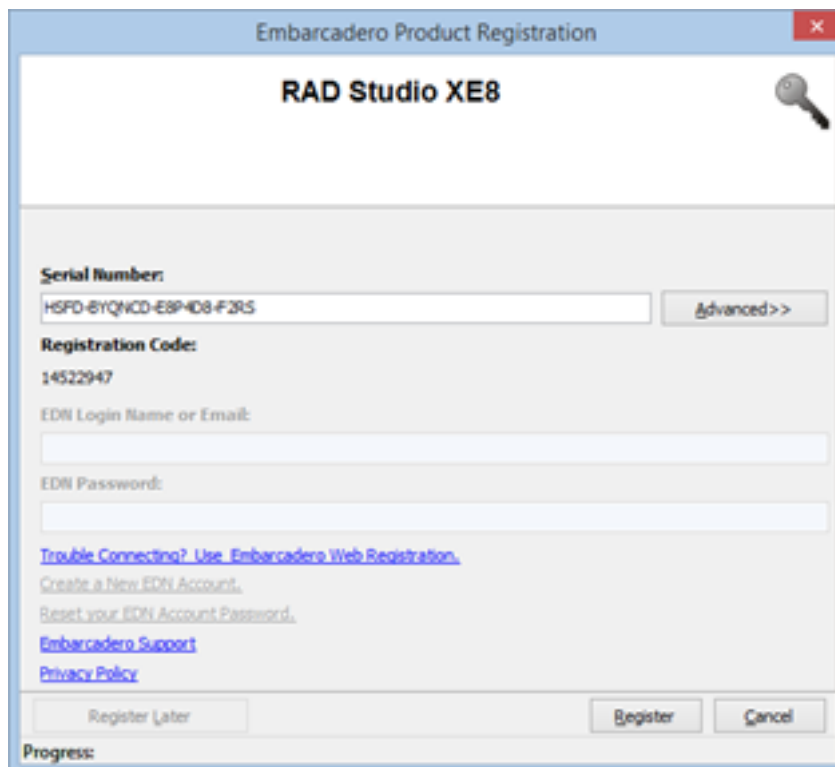


Рис. 1.12. Запуск Delphi 8 под Windows 8

При первом запуске появится окно для регистрации продукта (рис. 1.13), где нужно нажать кнопку «Register». Если используется уже не пробная бесплатная, а приобретённая лицензия (образовательная или коммерческая), то нужно предварительно ввести «EDN Login Name or Email» (учётная запись сети разработчиков Embarcadero или адрес электронной

почты) и «EDN Password» (пароль от учётной записи сети разработчиков Embarcadero), которые были введены при её создании (рис. 1.6).



The image shows a screenshot of the 'Embarcadero Product Registration' window for RAD Studio XE8. The window has a blue title bar and a close button in the top right corner. The main content area is white and contains the following elements:

- Serial Number:** A text box containing 'HSFD-8YQWCD-EBPND8-F3RS' and an 'Advanced >>' button to its right.
- Registration Code:** A text box containing '14522947'.
- EDN Login Name or Email:** A text box.
- EDN Password:** A text box.
- Four hyperlinks: 'Trouble Connecting? Use Embarcadero Web Registration.', 'Create a New EDN Account.', 'Reset your EDN Account Password.', and 'Embarcadero Support'.
- A 'Privacy Policy' link.
- At the bottom, there are three buttons: 'Register Later', 'Register', and 'Cancel'.
- A 'Progress:' label at the very bottom left.

Рис. 1.13. Регистрация продукта

При запуске может появиться окно с предупреждением Брандмаура Windows (рис. 14), где нам следует разрешить сетевую активность установленного нами продукта.

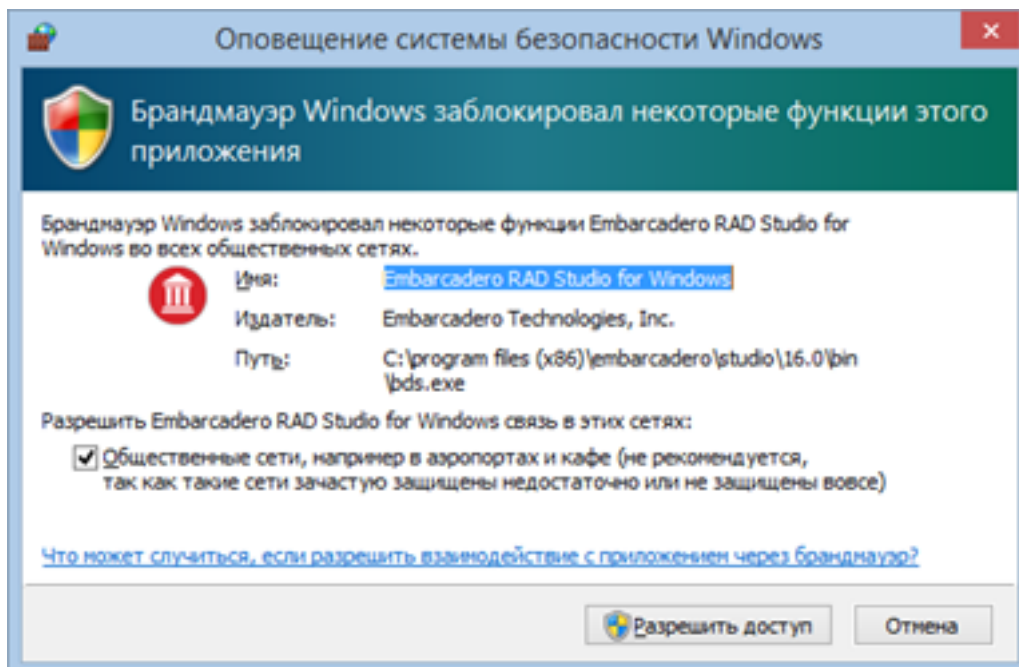


Рис. 1.4. Предупреждение Брандмауэра Windows

После успешного запуска на экране появится окно среды разработки Delphi XE8 (рис. 1.5). Центральное место занимает Welcome Page (страница приветствия), слева расположены две вертикальные панели Structure (структура) и Object Inspector. В левой стороне вертикально находятся также две панели Project Manager (менеджер проектов) и Tool Palette (палитра компонентов).

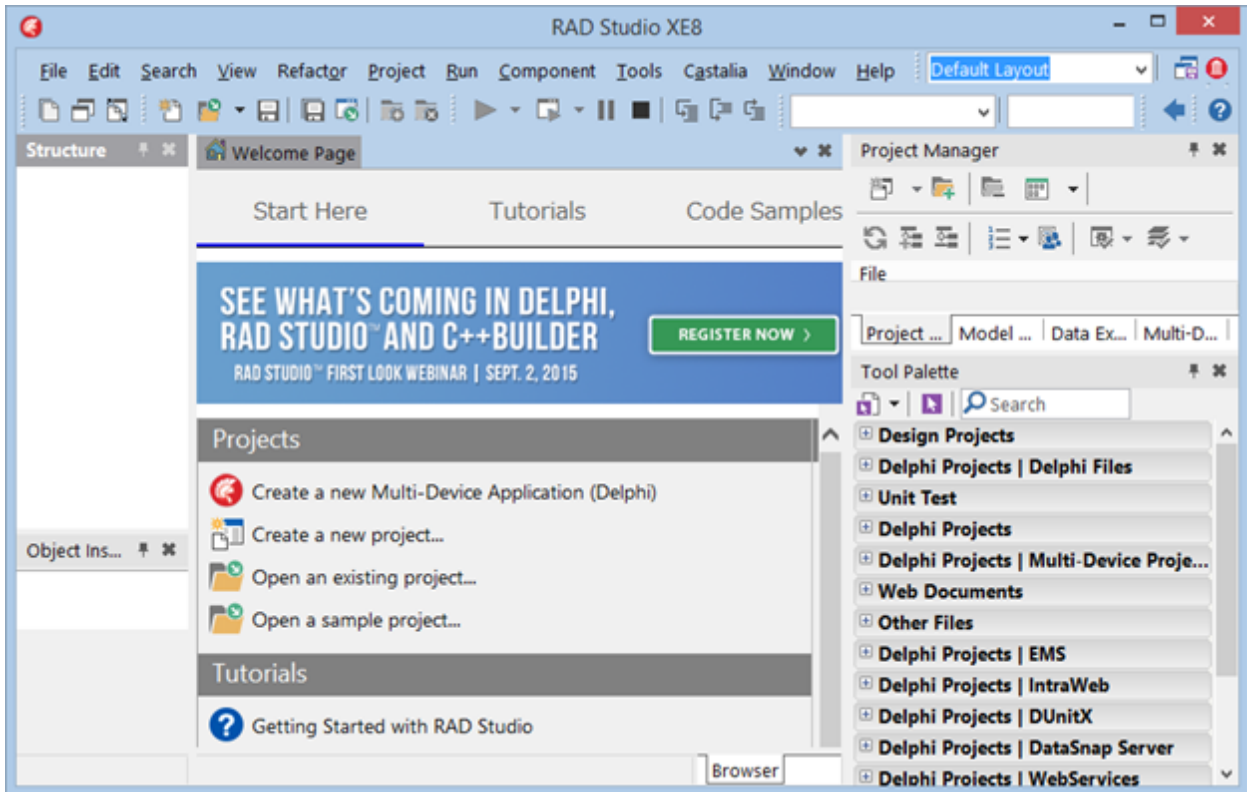


Рис. 1.15. Вид среды разработки после первого запуска

На странице приветствия в разделе Projects можно увидеть ссылку Create a new Multi-Device Application (создать новое приложение для различных устройств), но мы пока не будем торопиться с этим. Сначала подготовим среду для мобильной разработки. Для начинающих лучше всего подойдут простые и понятные инструкции на русском языке по ссылкам:

- <http://habrahabr.ru/company/delphi/blog/255721/> — настройка под iOS.
- <http://habrahabr.ru/company/delphi/blog/253929/> — настройка под Android.

Лучше всего воспользоваться официальным источником информации — онлайн-справкой от компании Embarcadero. Напомним, что Delphi/C++Builder/RAD Studio позволяют разрабатывать под мобильные платформы Android и iOS. Далее мы будем рассматривать разработку под мобильный устройства на Android, хотя за исключением первичной настройки среды всё остальное также подходит и для разработки под iOS.

Официальная документация по настройке и началу разработке от компании Embarcadero: http://docwiki.embarcadero.com/RADStudio/XE8/en/Android_Mobile_Application_Development — разработка под Android.

- http://docwiki.embarcadero.com/RADStudio/XE8/en/IOS_Mobile_Application_Development — разработка под iOS.

В связи с тем, что существует очень большое количество смартфонов и планшетов под управлением Android, то не всегда можно быстро настроить устройство на разработку. Часть сложно найти драйвера на подключаемое устройство. Ниже приведена полезная ссылка на подборку мини-статей про то, как подключаются различные устройства:

- <http://blogs.embarcadero.com/vsevolodleonov/category/android-devices/>.

1.9. Первый проект - таймер

В данном разделе для нас главное — начать и быстро получить результат. Поэтому будем выполнять последовательность действий строго по шагам:

1. В проводнике создадим папку и назовём её Project0.
2. В запущенной среде разработки Delphi выберем в главном меню File->New->Multi-Device Application — Delphi.
3. В появившемся окне кликнем на Blank Application.
4. Перед нами появится макет окна (рис. 1.16). Сохраним проект, выбрав File->Save All в созданную папку Project0.

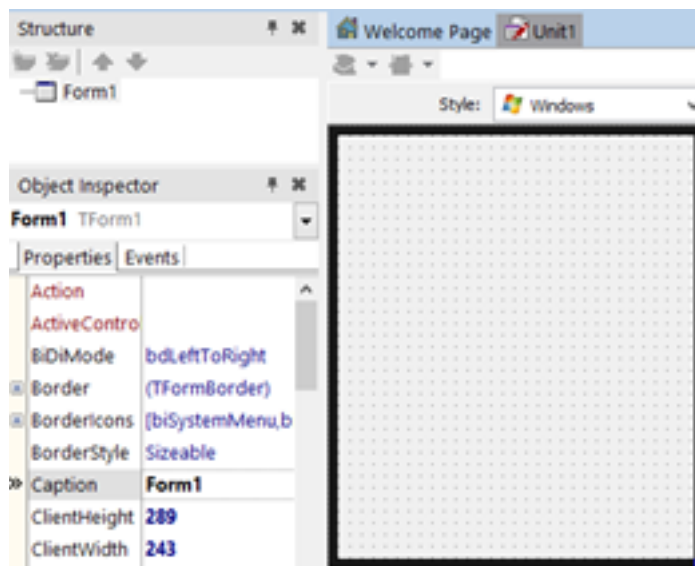


Рис. 1.16. Макет окна (формы)

5. Перейдём на палитру компонентов (Tool Palette), раскроем узел Standard и дважды щёлкнем на TLabel. На макете формы появится надпись Label1 (рис. 1.17).

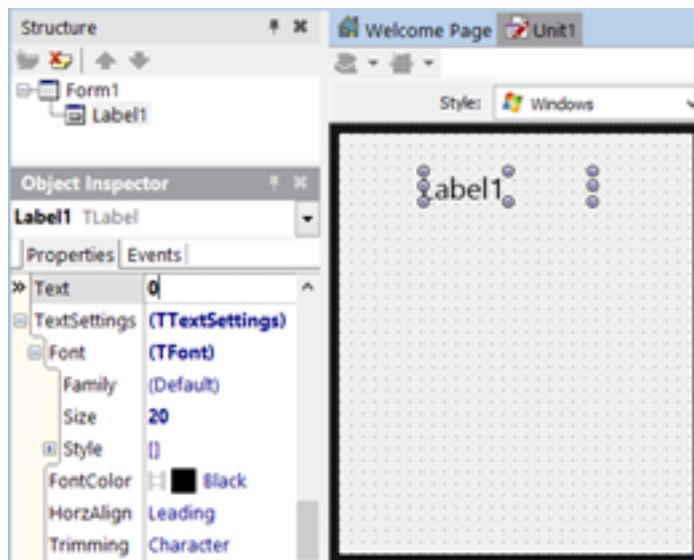


Рис. 1.17. Макет формы с надписью

6. Выберем Label1 на форме мышью, в Object Inspector найдём строчку Text и изменим его значение на 0. Затем раскроем узел TextSettings, затем узел Font и зададим свойство Size как 20.
7. Перейдём опять на палитру компонентов (Tool Palette), раскроем узел Standard, выберем TButton и перетащим на форму. Проделаем еще раз, чтобы на форме появилась вторая кнопка (рис. 1.18).

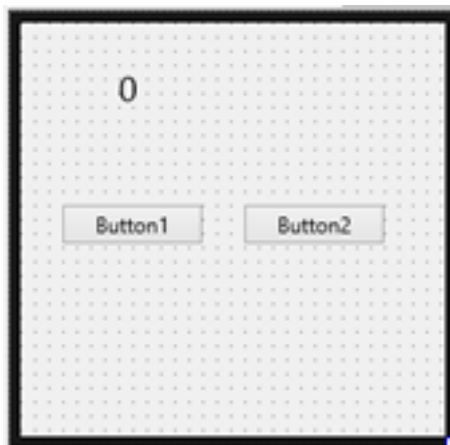


Рис. 1.18. Форма с надписью и кнопками

8. Поочерёдно выберем каждую из добавленных кнопок и в Object Inspector установим свойство Text в значение «Старт» и «Стоп», соответственно (рис. 1.19).

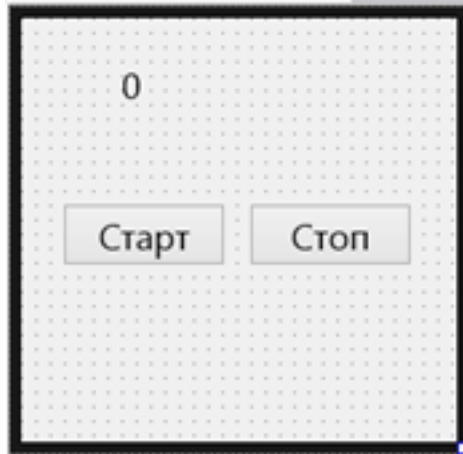


Рис. 1.19. Окончательный дизайн формы

9. В панели Project Manager (в правом верхнем углу) раскроем узел Target Platforms и выделим строку 32-bit Windows двойным кликом.
10. Выберем File->Save All.
11. Запустим приложение кнопкой с «зелёным треугольником».
12. Приложение запустится, и мы увидим его окно (рис. 1.20).

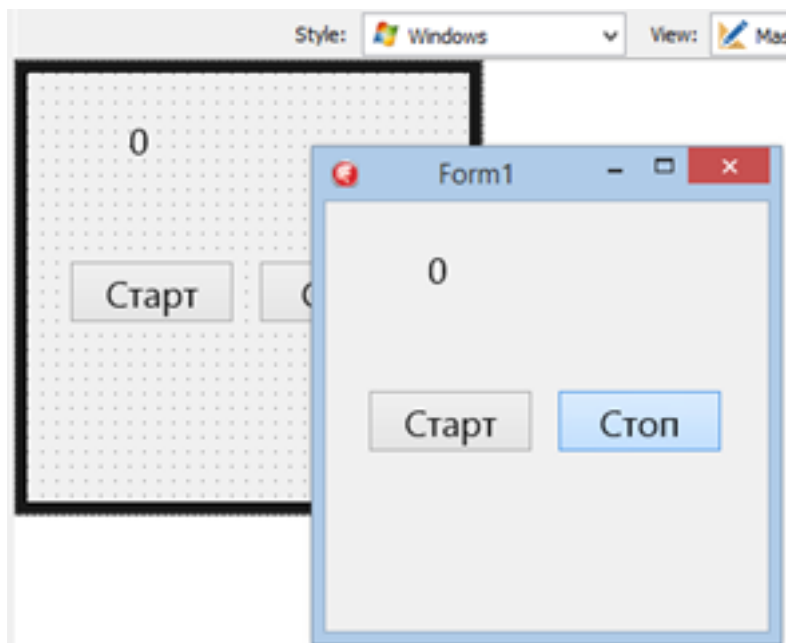
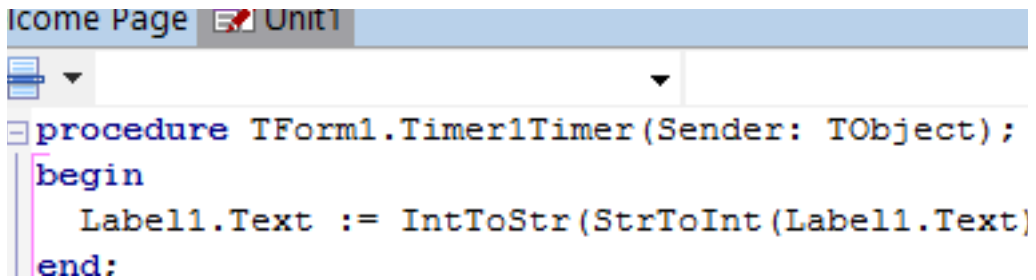


Рис. 1.20. Макет формы и запущенное приложение

13. Закроем приложение и вернёмся в среду разработки.
14. Добавим на форму компонент `TTimer` из палитры компонентов на форму.
15. Выделим его и в `Object Inspector` его свойств `Enabled` установим значение `false`.
16. В `Object Inspector` перейдём на закладку `Events`, найдём строку `OnTimer`, щёлкнем два раза в пустом поле рядом с этой надписью. Перед нами откроется редактор кода.
17. В пустой строке введём следующий код (рис. 1.21):

```
Label1.Text := IntToStr(StrToInt(Label1.Text) + 1);
```



```

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    Label1.Text := IntToStr(StrToInt(Label1.Text) + 1);
end;

```

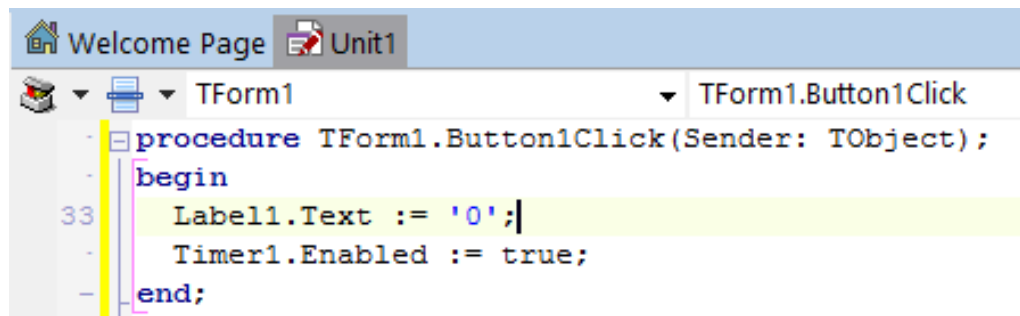
Рис. 1.21. Код для отображения количества пройденных секунд

18. Нажмём F12, чтобы перед нами опять появилась форма. Выберем кнопку с надписью «Старт», в Object Inspector на закладке Events найдём надпись с OnClick и дважды щёлкнем в пустом поле рядом с ней.
19. В появившемся редакторе кода введём следующее (рис. 1.22):

```

Label1.Text := '0';
Timer1.Enabled := true;

```



```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Text := '0';
    Timer1.Enabled := true;
end;

```

Рис. 1.22. Код для запуска таймера

20. Нажмём F12, перейдём опять в режим редактирования формы, выберем кнопку с надписью «Стоп», в Object Inspector на закладке Events найдём надпись с OnClick и дважды щёлкнем в пустом поле рядом с ней.
21. В появившемся редакторе кода введём следующее (рис. 1.23):

```

Timer1.Enabled := false;

```

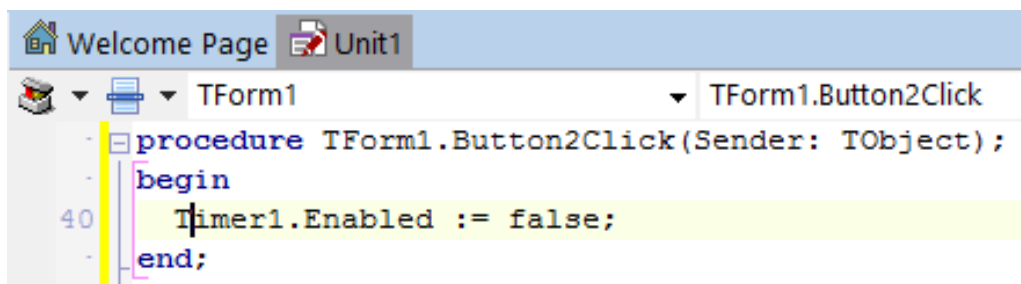


Рис. 1.23. Код для остановки таймера

22. Сохраним проект, выбрав в главном меню File->Save All.
23. Запустим проект на исполнение (кнопкой с «зелёным треугольником»). Протестируем приложение: нажатием на кнопку «Старт» мы будем запускать таймер, а кнопкой «Стоп» — останавливать его.
24. В панели Project Manager выберем узел Target Platforms, раскроем узел, соответствующий подключённому мобильному устройству. Дважды щёлкнем на устройстве (которое отображается после корректной настройки), выделив его таким образом.
25. Запустим приложение. Спустя некоторое время на устройстве должно запуститься уже мобильное приложение. Протестируйте его так, как мы это делали под Microsoft Windows.

Обратите внимание, что визуальный компонент типа «надпись» или «кнопка» в палитре компонентов (Tool Palette) имеет название, например, TButton, тогда как на форме оно преобразуется в Button1 или Button2. Пока удовлетворимся объяснением, что TButton — это «тип» компонента, тогда как Button1 или Button2 — это уже названия уже конкретных кнопок на форме. Представим себе, что есть понятие «котёнок», а если мы заведём парочку, то они уже будут иметь свои уникальные имена «мурзик» и «пушок». Приблизительно также «TButton» — это понятие, а «Button1» и «Button2» — конкретные объекты. Естественно, дальше по ходу изложения мы перейдём от таких упрощенных метафор к более точному и подробному объяс-

нению принципов работы объектно-ориентированной среды разработки с возможностями визуального прототипирования.

Следует запомнить, что пока мы занимаемся разработкой приложения — помещением на форму элементов управления типа «кнопка» или «надпись», заданием их свойств, а также вводом программного кода — это называется *design-time* (время разработки). Как только мы нажали кнопку с зелёным треугольником, и приложение запустилось, то мы перешли в фазу *runtime* (время исполнения). Чтобы не запутаться, форма или макет окна в режиме *design-time* имеет характерные «точечки», представляющие собой координатную сетку.

Только что мы сделали простое приложение, способное отсчитывать секунды на мобильном устройстве. Оно может использоваться для измерения:

- Времени при беге, плавании, катании на коньках.
- Периода прохождения химических реакций.
- Времени реализации физических процессов.
- Измерение скорости чтения.

Приложение можно усложнить, добавив, например, ввод дистанции и автоматического расчета средней скорости движения спортсмена. Также можно добавить текст, который будет читаться на время с мобильного устройства. Для физических процессов можно добавить функцию запоминания измерений, чтобы потом выполнять расчёт среднего значения.

Даже такое несложное приложение может быть эффективно использовано в проектных работах, включающих исследовательскую часть. Продолжим знакомство с техникой создания приложений в Delphi/C++Builder/RAD Studio, включая мобильные платформы, в следующих главах. По мере роста сложности выполняемых заданий будет расти и наше мастерство, а по завершению проработки материала книги вы сможете самостоятельно создавать мобильные приложения профессионального уровня.

Основные компоненты для мобильной разработки

2.1. Основные модели интерфейсов для мобильных устройств

Мобильные устройства можно разбить на два основных класса: смартфоны и планшеты. С точки зрения разработки приложений в RAD Studio и Delphi существенного различия между ними нет. Однако экраны планшетов больше, чем смартфонов, а их разрешение уже сопоставимо с разрешением мониторов обычных настольных ПК. Поэтому интерфейсы пользователя для планшетов и смартфонов заметно отличаются. Мы же в этой главе будем проектировать обобщенный интерфейс, соответствующий как планшету, так и смартфону.

Если вы являетесь опытным пользователем мобильных устройств, то при создании собственного мобильного приложения будет полезным вспомнить, какие интерфейсы показались вам наиболее удачными. Возможно, это какая-то обучающая программа или приложение для прослушивания аудио-записей. Это может быть офисной программой или даже интерфейсом настроек вашего смартфона. Если мобильные устройства для вас ещё не стали привычным, то целесообразно несколько абстрагироваться от дизайна приложений для настольных ПК под управлением Microsoft Windows и обратить внимание на готовые шаблоны в RAD Studio и Delphi. Создать приложение по шаблону можно, выполнив в IDE следующие действия: File->New->Multi-Device Application — Delphi. В результате появится следующее окно:

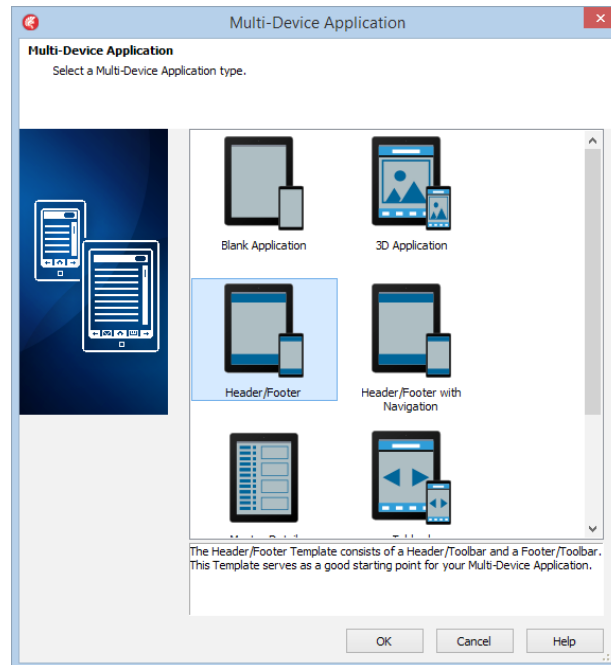


Рис. 2.1. Диалог выбора шаблона проекта

Произведём выбор одного из шаблонов. Первым является «Blank Application», что означает «пустое приложение», следовательно, интерфейс придётся создавать «с чистого листа». Остальные шаблоны (смысл которых легко понять из их названий) означают приложения, к главной форме которых добавлены уже некоторые визуальные компоненты. Желательно изучить эти шаблоны, выбрав их, нажав кнопку «Ок» (создав тем самым проект приложения), скомпилировав и запустив на мобильном устройстве. Посмотрите, из каких компонентов состоит интерфейс, как он выглядит на целевом планшете или смартфоне, а также какой уровень взаимодействия с пользователем он обеспечивает.

В большинстве случаев, если мы говорим не об игровых приложениях с уникальным дизайном, мобильные приложения строятся в соответствии с принципом многостраничности. Мы все привыкли к метафоре интерфейса приложений ОС Microsoft Windows (до версии 8). Там весь интерфейс строился на главном элементе — окне. Схожий подход наблюдается

и в операционных системах семейства Mac OS. Для мобильных же приложений в RAD Studio и Delphi основой является компонент TTabControl (Common Controls). Не менее важным также считается компонент TToolBar (Standard). Дизайн интерфейсов мобильных приложений мы будем строить, комбинируя и используя эти компоненты.

2.2. Компонент TTabControl

Изучим данный компонент непосредственно в проекте. Для этого выполним ряд действий:

- Создадим папку с названием «Project 2.1» в произвольном месте (например, в папке «Документы->Embarcaadero->Studio->Projects»).
- Запустим RAD Studio или Delphi.
- Создадим новое приложение, выполнив в IDE действия File->New->Multi-Device Application — Delphi.
- Выберем шаблон «Blank Application».
- Сохраним проект в созданную папку, выбрав File->Save All.

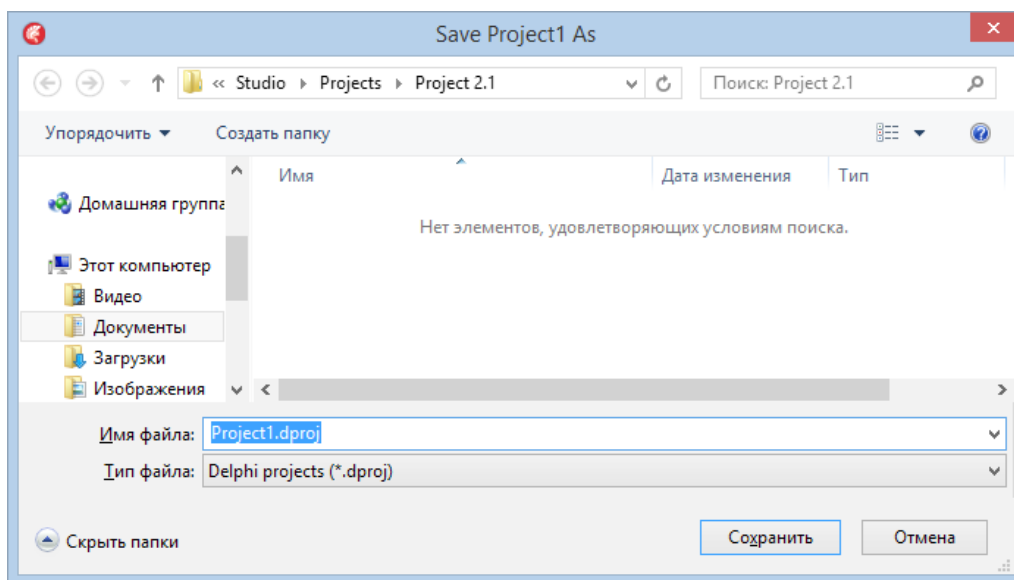


Рис. 2.2. Диалог выбора папки для сохранения проекта

Теперь мы видим главное окно мобильного приложения, на котором компонентов пока еще нет. В палитре компонентов выберем раздел Common Controls и разместим на форме компонент TTabControl. Данный компонент представляет собой набор «страничек» и «ярлычков» к каждой страничке. Пользователь уже созданного приложения может выбирать различные «ярлычки» или «закладки» (Tabs), тем самым переходя на нужную страничку (Рис. 2.3).

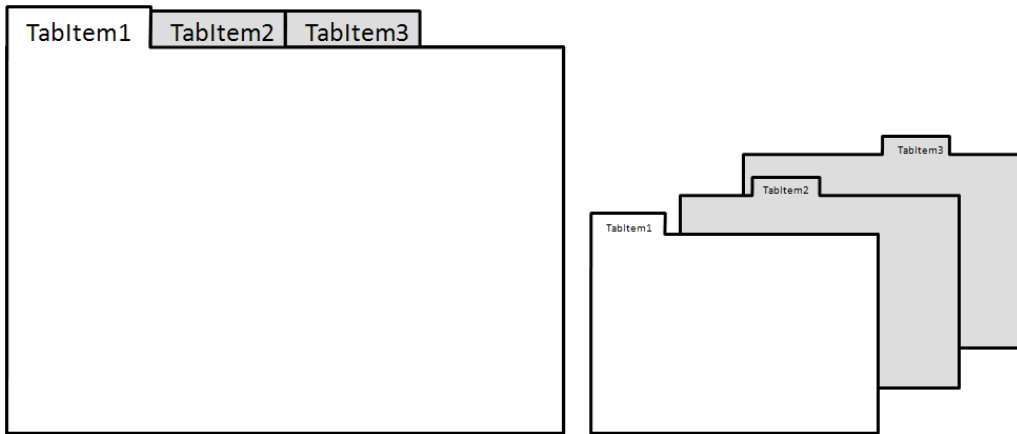


Рис. 2.3. Компонент TTabControl

На каждой «страничке» компонента TTabControl можно разместить любые элементы управления. Мобильные приложения с хорошо продуманным интерфейсом редко требуют «промотки» содержимого экрана, если его содержимое представляет собой небольшой объем текста, новостная лента и т.д. Гораздо эффективней разбить интерфейс на несколько «страничек» и разместить логически связанные визуальные компоненты на отдельных «страничках».

Компонент для «страничной» организации интерфейса пользователя появился ещё во времена, когда мобильные телефоны не были смартфонами, их экраны не позволяли использовать развитые интерфейсы пользователя. В операционной системе Microsoft Windows достаточно органично

смотрятся интерфейсы, оформленные данным образом. Если вы имеет опыт использования RAD Studio или Delphi для создания Windows-приложений, то можете вспомнить компонент `TPageControl` из библиотеки `VCL`, который по функциям сходен рассматриваемому компоненту `TTabControl` для мобильной разработки. Не следует его путать с одноименным `TTabControl` из `VCL`. Опыт использования компонента `TPageControl` может вам пригодится. Обратите внимание, что стиль «закладок» или `tab-ov` изменяется автоматически при сборке проекта под конкретную операционную систему. Переключите в дизайнера IDE платформу сборки и вы увидите, как изменяется внешний вид компонента (рис. 2.4).

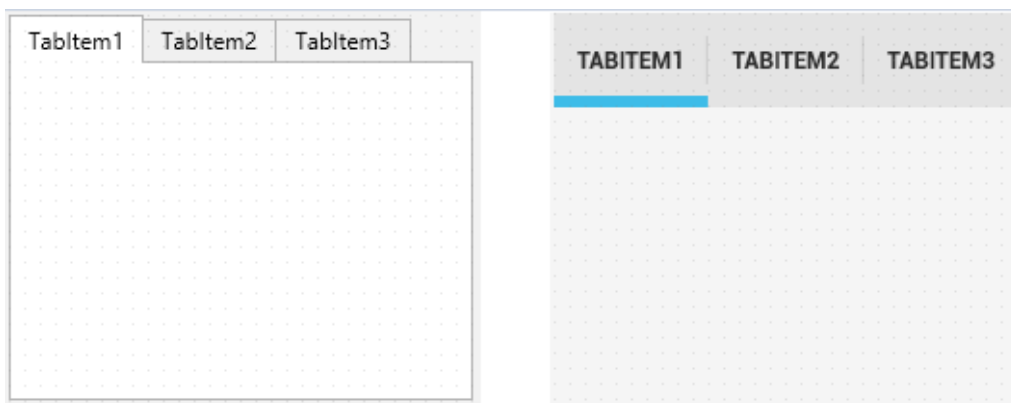


Рис. 2.4. Вид компонента *TTabControl* для различных ОС: *Windows* и *Android*

Вернёмся к нашему проекту. На главной форме мы только что разместили компонент `TTabControl`, но пока он не имеет никаких «закладок» и соответствующих им «страничек». Чтобы добавить «страничку» с «закладкой», наведите курсор на добавленный компонент, нажмите правую кнопку «мыши» и в появившемся всплывающем меню выберите пункт «Add `TTabItem`» («добавить элемент с закладкой»), как показано на рис. 2.5. Если очередная закладка не появляется, расширьте компонент.

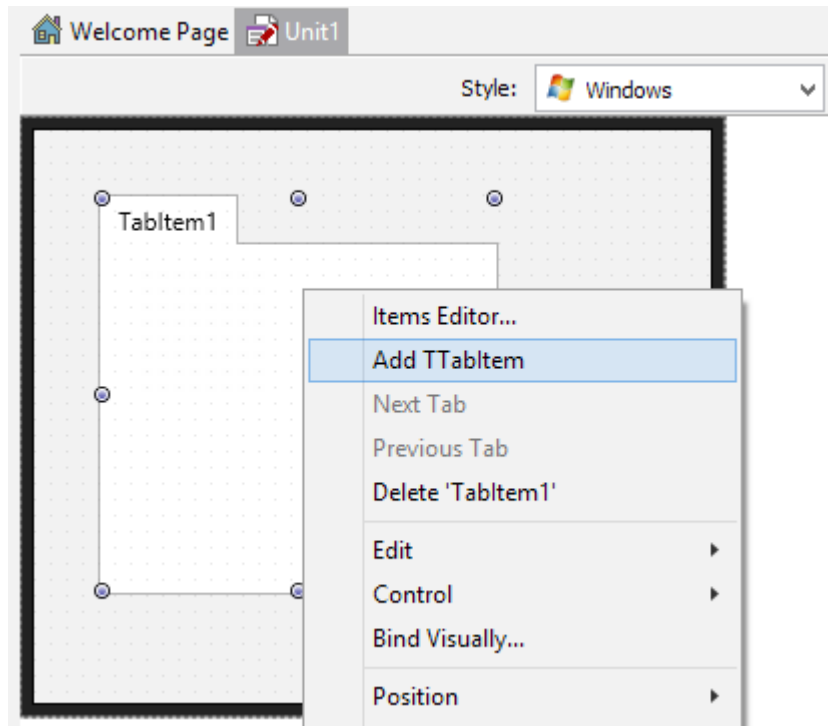


Рис. 2.5. *Добавление элементов к TTabControl*

Продолжим последовательность действий выше, закончившуюся на пункте 5.

- Добавим 5 элементов к компоненту TTabControl
- Разместим на форме компонент TToolBar (он автоматически займет верхнюю часть формы)
- Выделим компонент TTabControl (лучше всего в «дереве» на панели Structure) и установите его свойство Align в значение alClient.

На форме вы должны получить дизайн интерфейса, как показано на рис. 2.6 (слева). Если переключить стиль на Android, то дизайн будет соответствовать тому, что мы получим на мобильном устройстве.

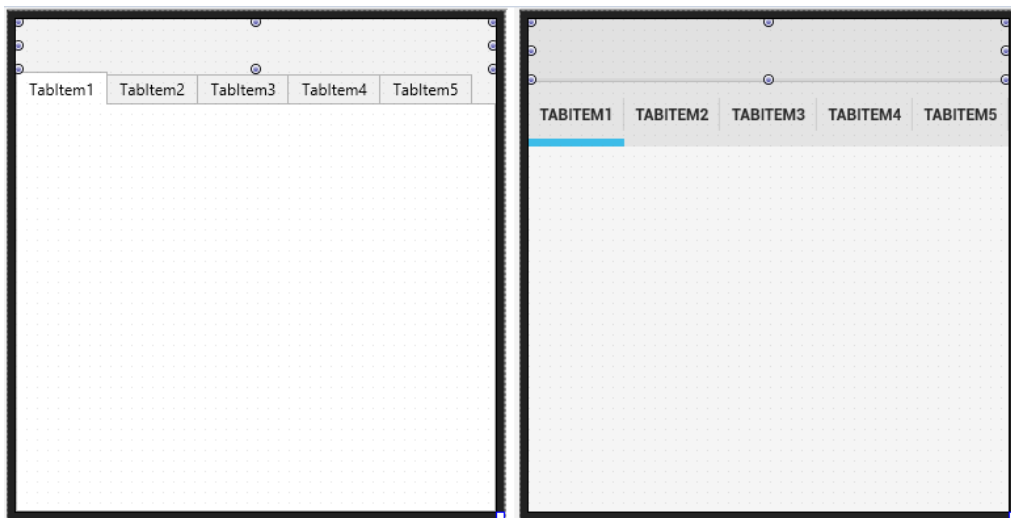


Рис. 2.6. Прототип интерфейса мобильного приложения

Теперь можно приступить к реализации приложения, на примере которого мы изучим основные свойства и методы компонента `TabControl`.

2.3. Прототип приложения для самоконтроля знаний учащихся

Большинство читателей, мы думаем, уже ощутили желание сделать какую-либо полезную программу. Одной из таких будет приложение для самоконтроля знаний учащихся. Современному школьнику приходится весьма интенсивно усваивать знания на каждодневной основе, поэтому небольшое мобильное приложение будет весьма полезным. Оно будет всегда с вами на смартфоне или планшете, поэтому в любой момент можно выполнить проверку своих знаний.

Возьмем, например, учебник биологии и выберем самый сложный на наш взгляд параграф. Затем по данному параграфу придумаем свои вопросы, ответы на которые помогут быстро вспомнить пройденный материал. Таким образом, нам удастся совместить две задачи: знакомство с базовыми

компонентами для мобильной разработки и создание полезной программы.

Вспомним, что у нас в проекте на главной форме добавлено 5 «закладок». Первые четыре мы будем использовать для размещения вопросов и вариантов ответа, а последняя закладка нам пригодится для демонстрации результатов тестирования. На закладках будут показаны вопросы с четырьмя вариантами ответов. Один из них будет правильный, а три — заведомо ошибочные.

Непосредственно на форме выделим первую закладку «TabItem1», кликнув на ярлычок. Проверим правильность выделения на панели «Structure». В «Object Inspector» найдем свойство «Text» и поменяйте его значение на «1» (рис. 2.7). Это будет означать «вопрос № 1». Затем в разделе «Standard» палитры компонентов найдем «TLabel» и перетащите его на первую страничку, как показано на рис. 2.8.

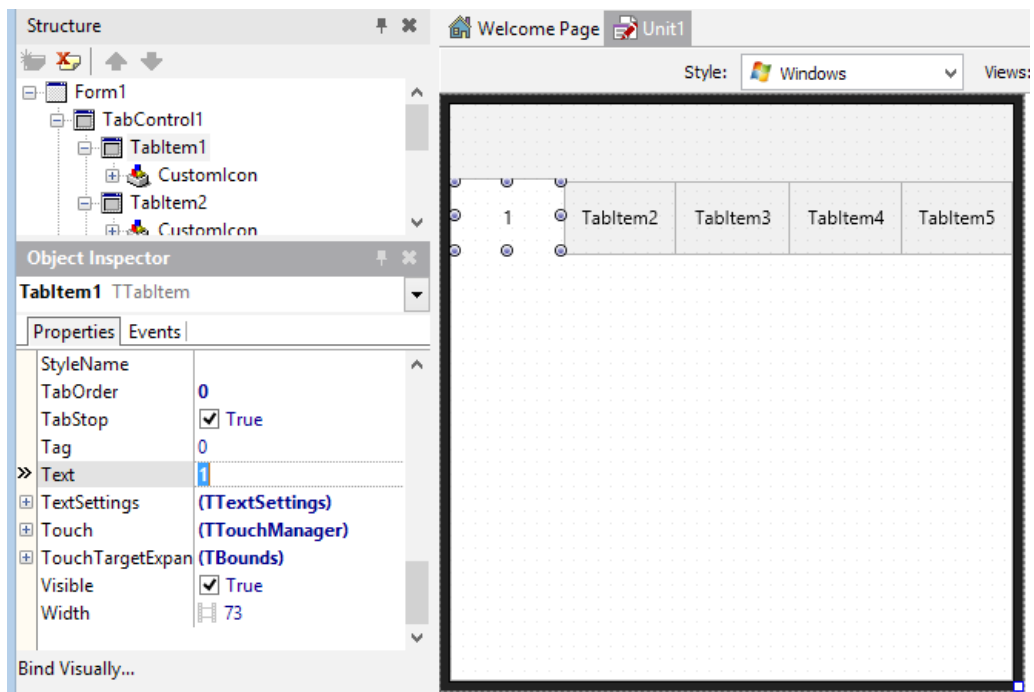


Рис. 2.7. Изменение свойства Text элемента TabItem1

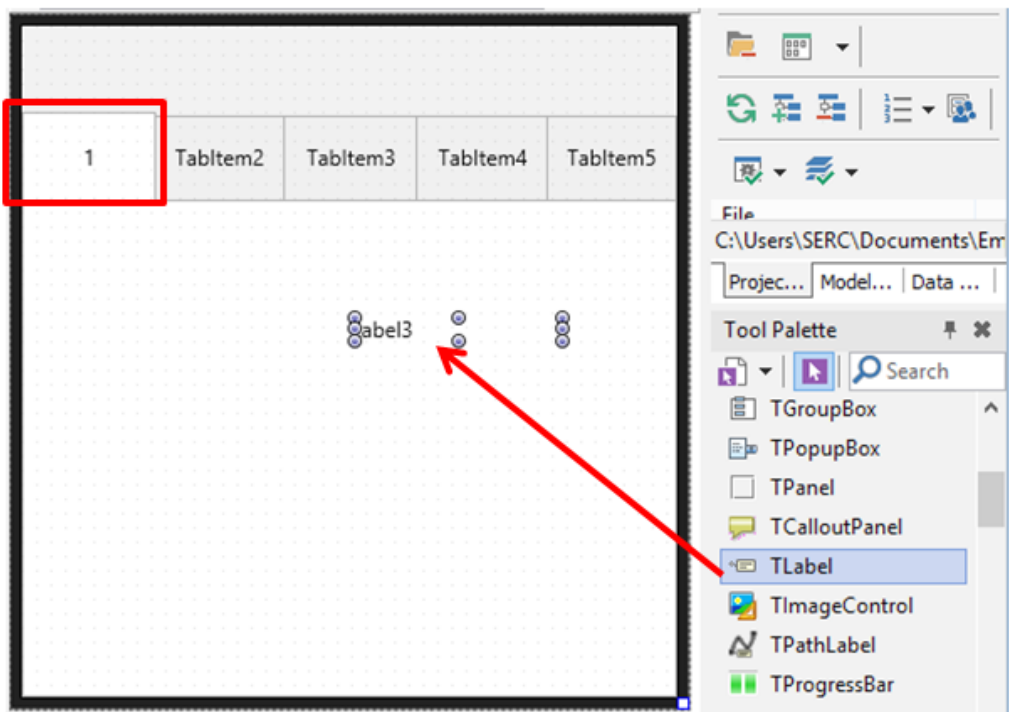


Рис. 2.8. *Добавление TLabel на страничку TabItem1*

Далее воспользуемся «Object Inspector» и таблицей ниже, чтобы нужным способом задать значения свойств компонента добавленного TLabel:

Label3: TLabel

Свойство	Значение	Объяснение
Align	alTop	Компонент автоматически распределится по верхней части
StyleLookup	Listboxitemlabel (выбрать из выпадающего списка)	Размер текста станет больше в соответствии выбранным стилем
Text	Как называются основные перья птиц?	Это — текст вопроса, который будет показан компонентом
TextSettings.HorzAlign	Center	Выравнивание текста по центру

Затем добавим на форму еще один компонент TLabel, свойство которого зададим аналогично первому (кроме свойства Text, в которое введем

«Ваш ответ»). Также добавим 4 компонента `TButton` и разместим их вертикально в ряд, а свойство `Text` каждой кнопки изменим так, чтобы они выглядели вариантами ответов на поставленный компонентом `TLabel` вопрос. Наконец, добавим на форму компонент `TToolBar`, а значение свойства `Align` выберем из списка как `alBottom`. Свойство `StyleLookup` также выберем из списка как `bottomtoolbar`. Все эти манипуляции проделываются в «Object Inspector». Окончательно наш интерфейс должен выглядеть аналогично представленному на рис. 2.9.

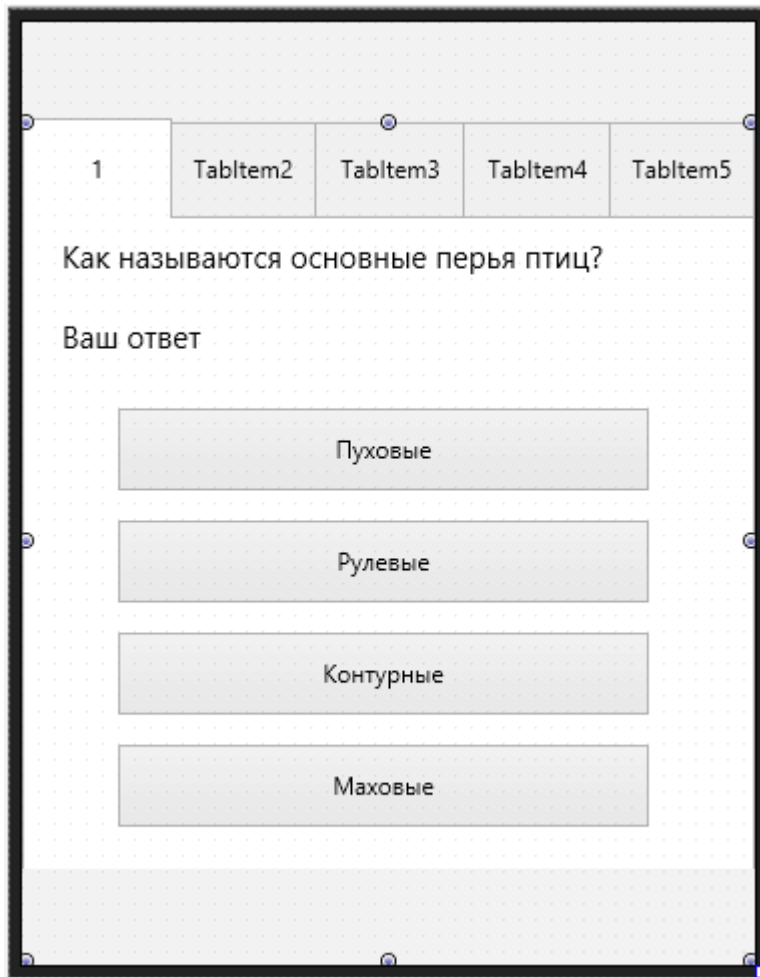


Рис. 2.9. Окончательный вид первой страницы

Сохраните проект, выбрав File->Save All в главном меню IDE. Нажмите на кнопку «Run Without Debugging». После этого ваша среда разработки соберёт ваше приложение под Windows (если вы не изменяли настройки среды) и запустит его. В появившемся окне вы можете нажимать кнопки и перелистывать страницы. С точки зрения интерфейса наше приложение выглядит вполне нормально, осталость ввести код, который будет определять логику работы программы в ответ на действия пользователя.

Сейчас нам нужно понять, как обращаться к визуальному компоненту из программного кода. К любому из компонентов на интерфейсе можно обратиться при помощи свойства Name, которое можно найти в «Object Inspector». На рис. 2.10 мы красным шрифтом подписали имена визуальных компонентов. Если у вас названия отличаются, то это нормально. Главное — знать имена элементов интерфейса. Выделим мышью нужный элемент интерфейса и узнаем его имя в окошках «Object Inspector» или «Structure».

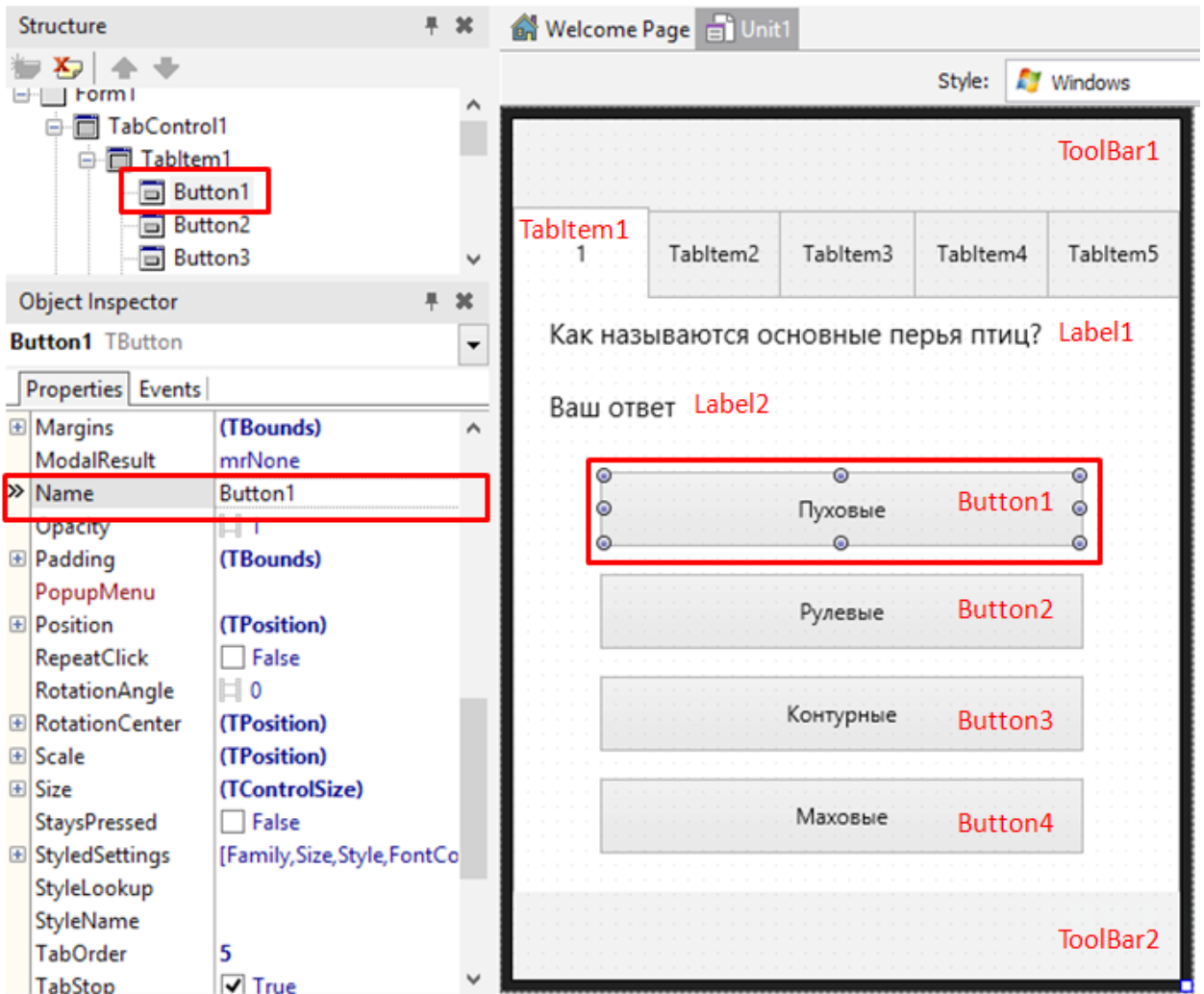


Рис. 2.10. Имена визуальных компонентов интерфейса

Теперь выделим Button1 как на рис. 2.10, перейдём в «Object Inspector», там на закладке Events найдем OnClick, щёлкнем там два раза, а IDE нам сгенерирует заготовку для кода. Данный код сработает, как легко догадаться, когда пользователь нажмёт на кнопку, т.е. сработает событие OnClick (рис. 2.11).

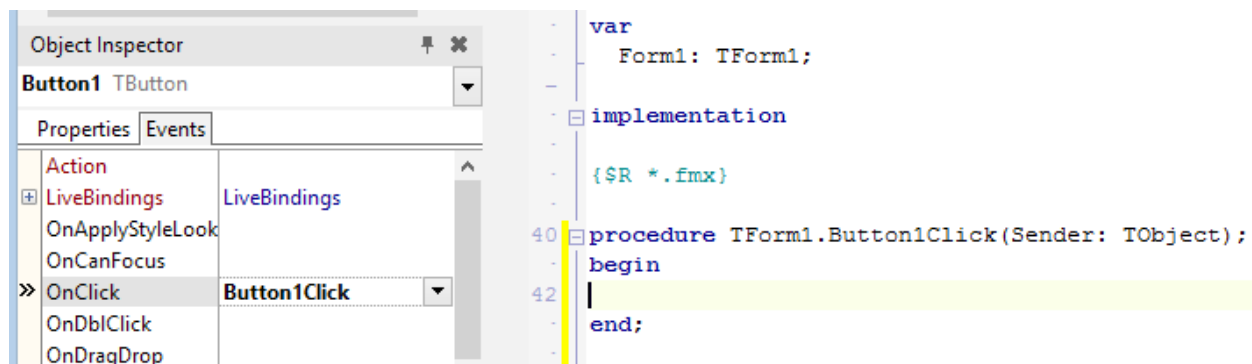


Рис. 2.11. Код срабатывания на событие *OnClick*

Первое, что приходит в голову — это написать следующий код, который будет заносить текст кнопки в Label2 как вариант ответа:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label2.Text := Button1.Text;
end;
```

При всей простоте и понятности данного кода (а вы можете проверить его работоспособность, скомпилировав, собрав и запустив проект на выполнение) он неэффективен. Это легко можно проиллюстрировать, если запрограммировать событие OnClick для следующей кнопки Button2:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label2.Text := Button2.Text;
end;
```

Два приведенных выше фрагмента кода практически идентичны за небольшим исключением. Изменяется лишь компонент-кнопка Button1 на Button2, а также название процедуры. Самое главное, что неизменной остаётся логика кода — присвоение свойству Label2.Text значению свойства Text нажатой кнопки Button1 или Button2. Если продолжить программировать кнопки данным способом, то для каждой из них у нас бу-

дет отдельная процедура отклика (рис. 2.12). Смысл всех процедур можно выразить так: значение свойства **Text** нажатой кнопки передать свойству **Text** компоненту Label2. Чтобы сделать код эффективным, нужно вместо Button1, Button2, Button3 и т.д. в качестве имени нажатой кнопки указать «та, которую нажали». Тогда мы сможем обойтись единой процедурой отклика на любую из кнопок.

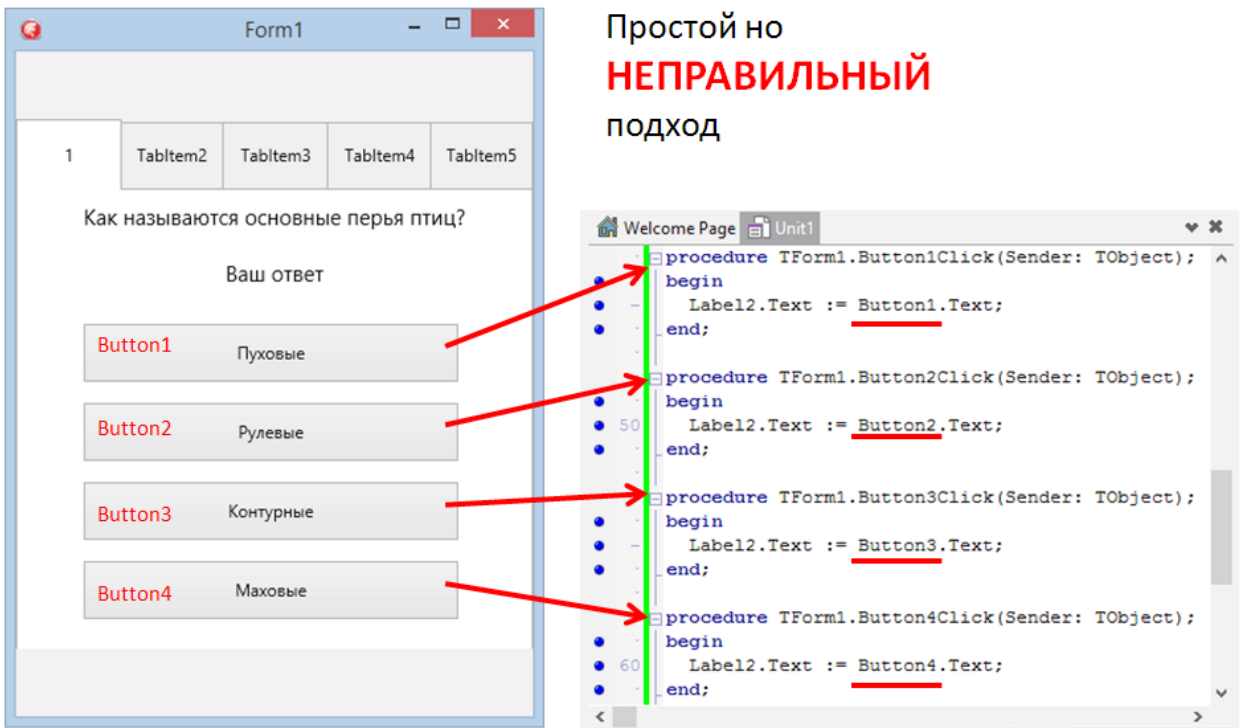


Рис. 2.12. Неправильная организация кода откликов на кнопки

У начинающих программистов часто возникают возражения: «Код же работает, почему он неэффективный?» или «Что плохого, что у меня много однотипного кода?». Представьте себе, что мы будем в дальнейшем развивать программу. Это не просто возможно, но и неизбежно. Допустим, мы захотим вносить в Label2 не только сам ответ, но и предваряющую фразу:

```
Label2.Text := 'Ваш ответ: ' + Button1.Text;
```

Тогда согласно рис. 2.12 придётся изменять коды всех четырех процедур. А если кнопок будет не четыре, а сорок четыре? А если изменения придётся вносить достаточно часто? Что мы получим в итоге? В итоге мы получим увеличение вероятности ошибки в коде из-за повторяющихся однотипных действий. Сделаем код максимально эффективным! Для этого нужно переформулировать код таким образом, чтобы он стал универсальным для всех четырёх кнопок (рис. 2.13).

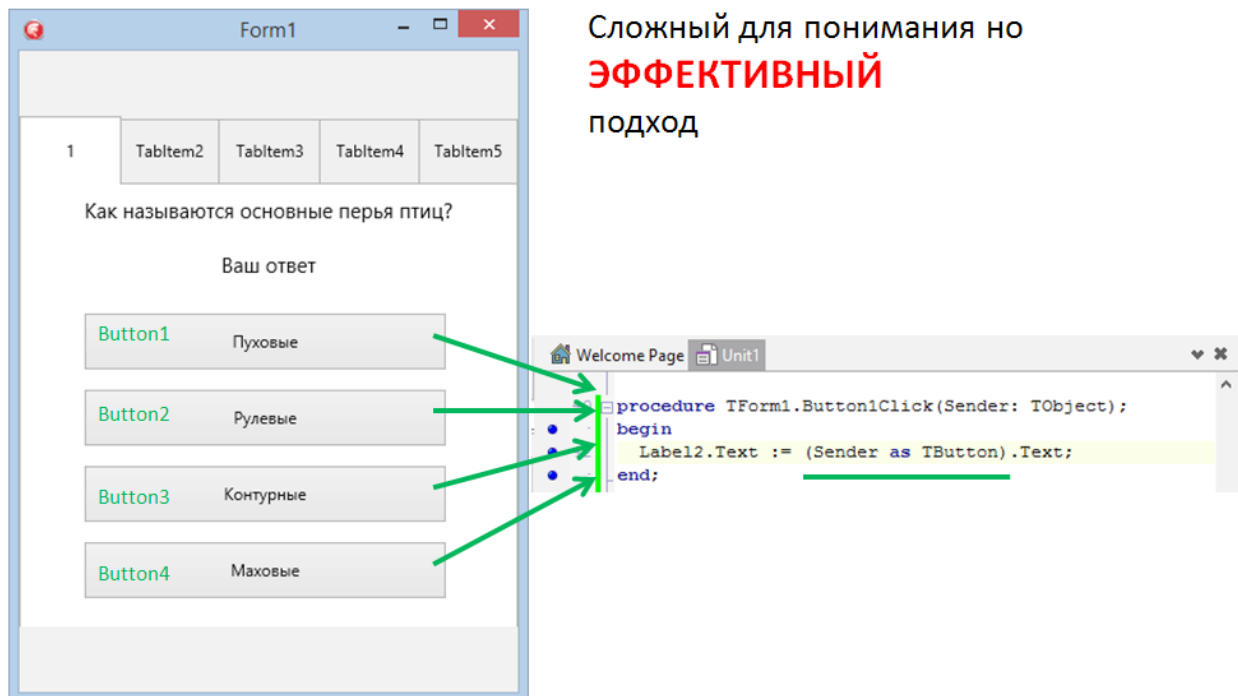


Рис. 2.13. Эффективная структура кода отклика для четырех кнопок

Если вы всё-таки сделали на каждую кнопку отдельную процедуру отклика, то необходимо выполнить следующие действия.

Удалите введенные строки, оставив заготовку процедуры, которую сгенерировала IDE:

```

procedure TForm1.Button2Click(Sender: TObject);
begin

end;

```

Сохраните проект, пустые заготовки процедур будут удалены автоматически. Операция сохранения сопровождается удалением кода пустых процедур. Таким образом IDE устраняет лишний код, исполнение которого лишено смысла.

Теперь в коде единственной процедуры сделайте следующие изменения:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label2.Text:= (Sender as TButton).Text;
end;

```

Обратите внимание, что Button2 мы заменили на (Sender **as** TButton). Теперь для каждой из трёх оставшихся кнопок в «Object Inspector» выберем отклик на событие OnClick из выпадающего списка, как показано на рис. 2.14. Сохранив проект, запустим приложение и протестируем его работоспособность.

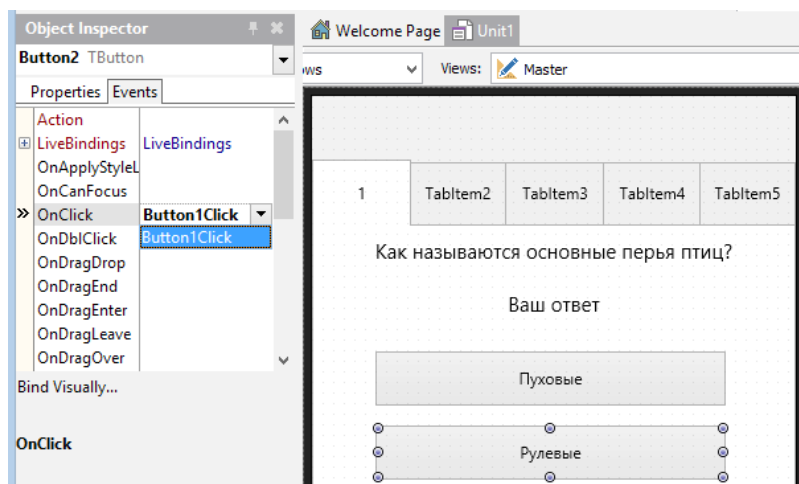


Рис. 2.14. Назначение процедуры отклика на событие OnClick

2.4. Конструкция (Sender as TButton)

Только что мы проделали крайне важный для всего объектно-ориентированного программирования трюк, придав нашему коду единообразие и устранив избыточность. Проведём анализ конструкции (`Sender as TButton`). При задании реакции на событие `OnClick` среда IDE автоматически генерирует следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Label2.Text := (Sender as TButton).Text;  
end;
```

Обратим внимание на параметр процедуры (`Sender: TObject`), который мы используем в конструкции (`Sender as TButton`). Считаем, что `Sender` — «объект, в жизни которого произошло событие `OnClick`». Если пользователь нажал на `Button1`, то `Sender` обозначает `Button1`. А если — `Button2`, то `Sender` обозначает кнопку `Button2`. В процедуру отклика на событие передаётся параметр `Sender`, представляющий собой ссылку на объект, к которому относится происшедшее событие (рис. 2.15).

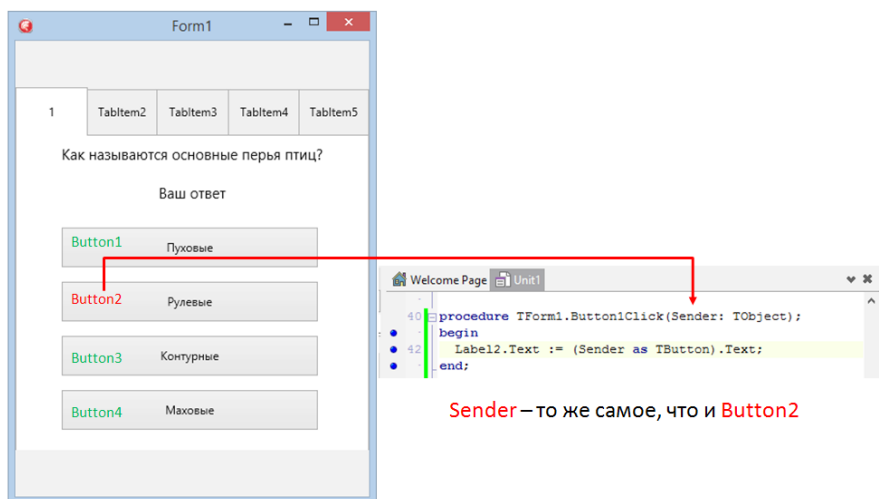


Рис. 2.15. Значение параметра `Sender` в процедуре отклика

В списке аргументов процедуры `Sender` объявлен как `TObject`. В переводе с языка `Delphi/Object Pascal` на обычный это означает «любой объект». Таким образом, `Sender` может быть любым объектом, не обязательно кнопкой `TButton`. Ну а поскольку нам нужно использовать данный параметр именно в качестве кнопки, то мы и применяем оператор **as**, который и позволит это сделать на уровне синтаксиса языка. `Sender`, будучи взятым «в качестве» `TButton` (именно «as» в буквальном переводе), теперь не только ссылается на кнопку, но и «есть кнопка». Более подробно данные вопросы рассмотрены в главе «Объектно-ориентированное программирование».

2.5. Добавление новых вопросов

Выполним небольшую самостоятельную работу — аккуратно доделаем остальные странички компонента TTabControl. Ниже приведены вопросы № 2, № 3 и № 4:

Киль грудины у летающих птиц нужен для:
крепления суставов плечевой кости;
крепления грудных мышц;
крепления суставов бедренной кости;
крепления маховых перьев.

Какой орган не входит в пищеварительную систему зерноядных птиц?
мускульный желудок;
зоб;
железистый желудок;
трахея.

Какой орган не входит в дыхательную систему птиц?
центральный бронх;
передние воздушные мешки;
средние воздушные мешки;
задние воздушные мешки.

Для облегчения задачи можно воспользоваться техникой копирования-вставки, применимой к компонентам, расположенным на форме в RAD Studio и Delphi. Выделим на первой закладке по очереди все элементы, начиная с первого, нажав и удерживая кнопку Shift на клавиатуре. Выбранные компоненты будут отмечены небольшими чёрными квадратами по контуру (рис. 2.16). После чего нажмём правую кнопку мыши и выберем Edit->Copy из всплывающего меню. Компоненты будут «скопированы» в буфер.

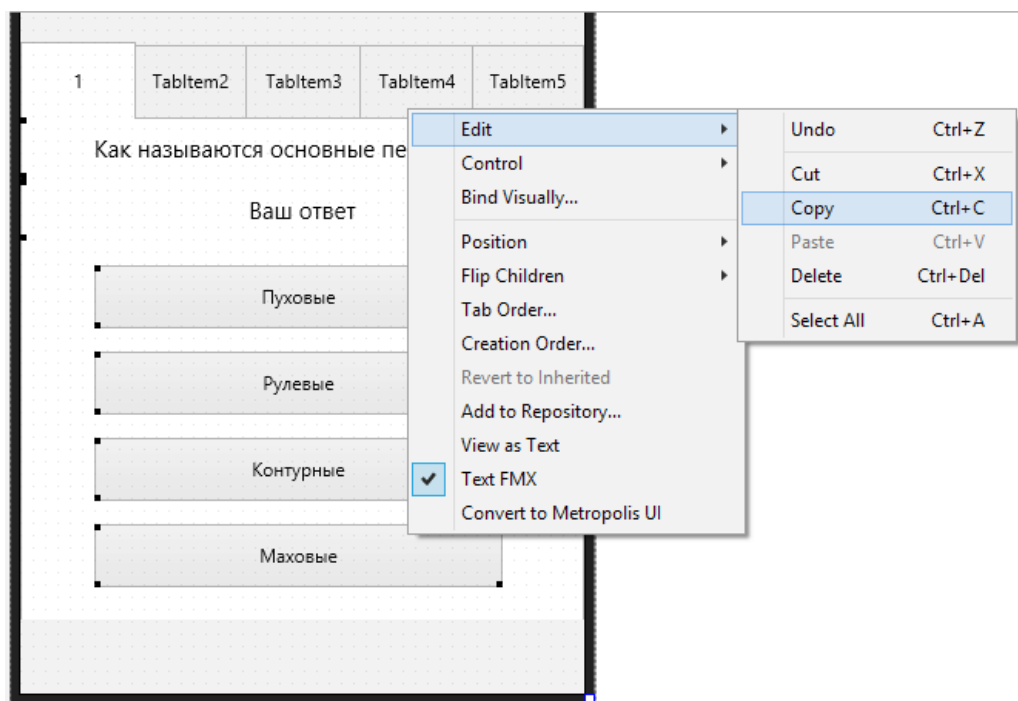


Рис. 2.16. Выделение и копирование компонентов на форме

Теперь перейдём на вторую закладку, нажмем правую кнопку мыши и из всплывающего меню выберем Edit->Paste. На страничке появятся визуальные компоненты, скопированные ранее (рис. 2.17).

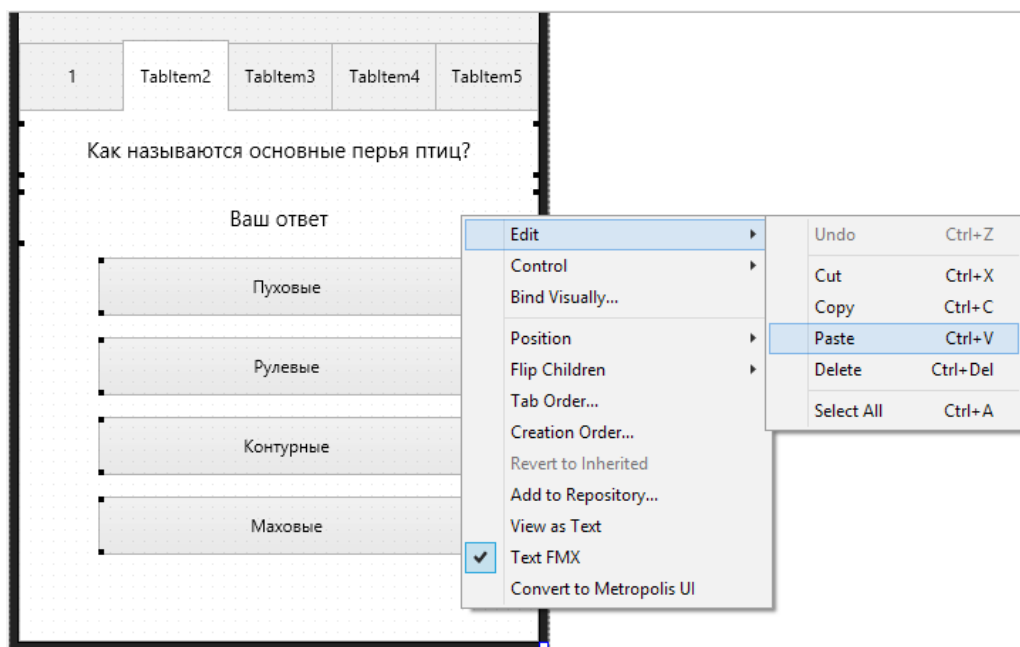


Рис. 2.17. Вставка компонентов на форме

Если какой-либо компонент случайно не попал в выделение и, соответственно, не появился на второй страничке, то вернемся обратно и еще раз сделаем операцию копирования-вставки. Теперь заменим свойства Text меткам и кнопкам в соответствии с вопросом № 2.

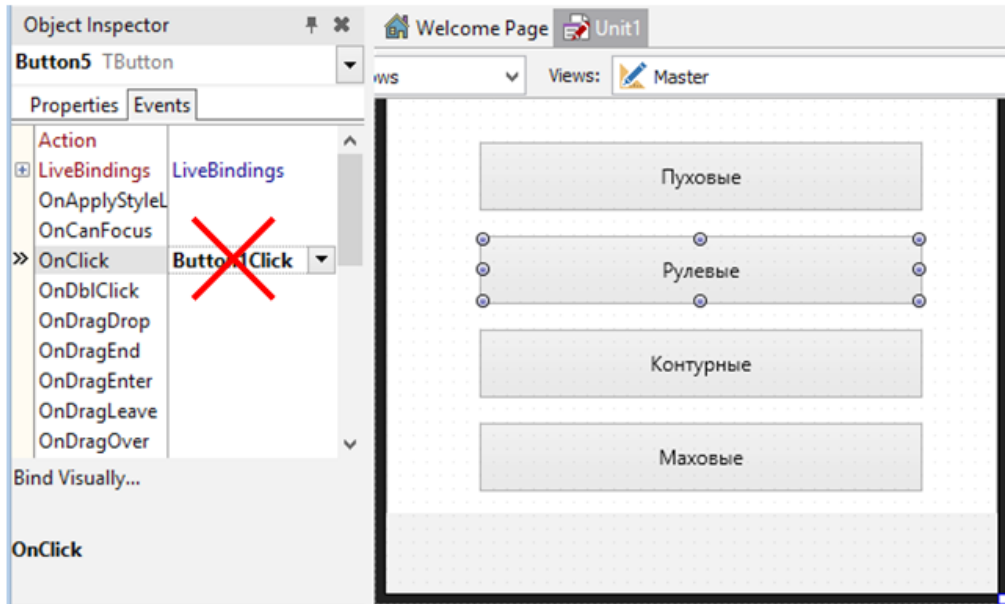


Рис. 2.18. Удаление события *OnClick* для кнопки

При копировании-вставке компонента копируются все значения его свойств и даже название процедуры отклика *OnClick*. Если мы хотим, чтобы новый компонент имел свою процедуру отклика, то нужно в «Object Inspector» найти данное событие на закладке *Events* и вручную удалить его (рис. 2.18). При необходимости новую процедуру можно создать двойным щелчком в пустом поле рядом с *OnClick*.

В случае, если текст вопроса не будет вмещаться в размеры компонента *TLabel*, нужно выбрать режим разбиения текста на строки. Достигается это установкой свойства *TextSettings.WordWrap* в значение *true* с помощью «Object Inspector» (рис. 2.19).

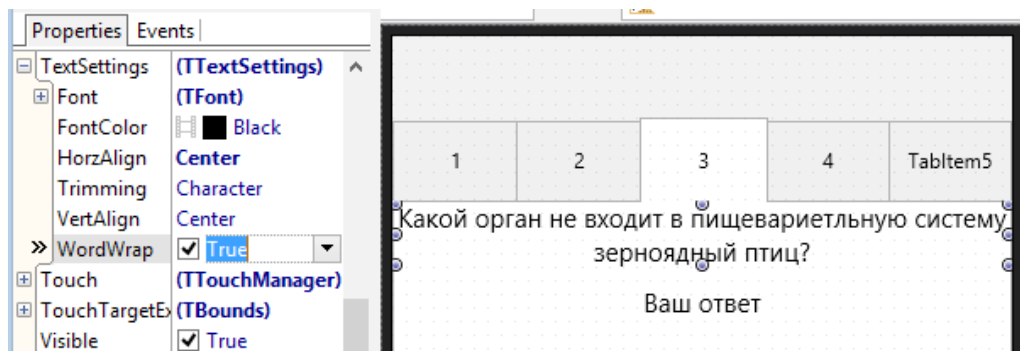


Рис. 2.19. Установка режима разбиения на строки в *TLabel*

Выберем по очереди кнопки на странице № 2 и удалим значение `OnClick` в «Object Inspector». Затем с нажатой клавишей `Shift` выделим все кнопки на странице № 2 и двойным щелчком в пустом поле `OnClick` сгенерировать при помощи IDE процедуру отклика. Затем в ручном режиме введём недостающую строку:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  Label4.Text:= (Sender as TButton).Text;
end;
```

Прделаем то же самое с закладками № 3 и № 4. Поскольку мы интенсивно использовали технику копирования-вставки с последующей коррекцией процедур отклика, необходимо протестировать готовый прототип. Типичной проблемой является отсутствие реакции на нажатие кнопки. В таком случае нужно проверить событие `OnClick` в «Object Inspector». Еще раз посмотрим на рис. 2.13. Для всех четырех кнопок на каждой страничке должна быть единая процедура отклика на событие `OnClick`. Если тестирование выявило ошибки, то таблица ниже поможет проконтролировать нужные значения в «Object Inspector»:

Закладка	Кнопка	Событие OnClick
TabItem1	Button1	Button1Click
	Button2	Button1Click
	Button3	Button1Click
	Button4	Button1Click
TabItem2	Button5	Button5Click
	Button6	Button5Click
	Button7	Button5Click
	Button8	Button5Click
TabItem3	Button9	Button9Click
	Button10	Button9Click
	Button11	Button9Click
	Button12	Button9Click
TabItem4	Button13	Button13Click
	Button14	Button13Click
	Button15	Button13Click
	Button16	Button13Click

Для каждой группы кнопок на одной и той же страничке компонента TabControl должна быть единая процедура отклика на событие OnClick.

2.6. Лучшее – враг хорошего или подождём с улучшениями

Введённое нами понятие «эффективность кода» могло заставить читателя задуматься на тему дальнейшего улучшения текста программы. Действительно, нам удалось объединить процедуры отклика в единый вариант для групп из четырёх кнопок для каждой странички (рис. 2.13). Процедуры отклика для разных страничек отличаются также совсем незначительно. Сравним:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label2.Text:= (Sender as TButton).Text;
end;
procedure TForm1.Button5Click(Sender: TObject);
begin
  Label4.Text:= (Sender as TButton).Text;
end;
```

где `Button1Click` и `Button5Click` — процедуры реакции на нажатие кнопок первой и второй страничек. Естественно возникает желание создать единый код для всех кнопок с выбором вариантов ответа в программе.

Отметим, что улучшать программный код можно очень долго. Существуют так называемые «шаблоны проектирования», применение которых позволяет кардинальным образом повысить качество структуры программного кода. Использование изоощренных шаблонов требует понимания всех тонкостей объектно-ориентированного программирования. Поэтому остановимся на текущей реализации проверки выбора ответа пользователем в виде одной процедуры отклика на каждую группу из четырёх кнопок, а не единой процедуры для всех.

Объединение всех процедур отклика в единый вариант возможно, но это далеко не единственный путь повышения качества кода. В «идеальном» случае вся система должна работать следующим образом:

- Отдельно приложение используется для ввода вопросов и ответов по выбранной теме.
- Созданные вопросы и ответы сохраняются в специализированной базе данных.
- При запуске мобильное приложение запрашивает или назначает пользователю тему для прохождения теста.
- После выбора или назначения темы мобильное приложение связывается с базой данных и загружает вопросы и ответы по выбранной теме.
- В зависимости от количества вопросов динамически создаются «странички» компонента `TabControl`.
- В зависимости от числа вариантов ответа на каждый вопрос создаются «кнопки с ответами» на каждой страничке компонента `TabControl`.
- В компоненты `TLabel` и `TButton` загружаются тексты вопросов и варианты ответов.
- На все кнопки автоматически назначается единая процедура реакции на событие `OnClick`.

- На каждое нажатие на кнопку с ответом также запускается глобальная процедура оценки правильности ответа.
- После прохождения теста автоматически подсчитывается результат, который заносится в базу данных для конкретного пользователя (рис. 2.20).

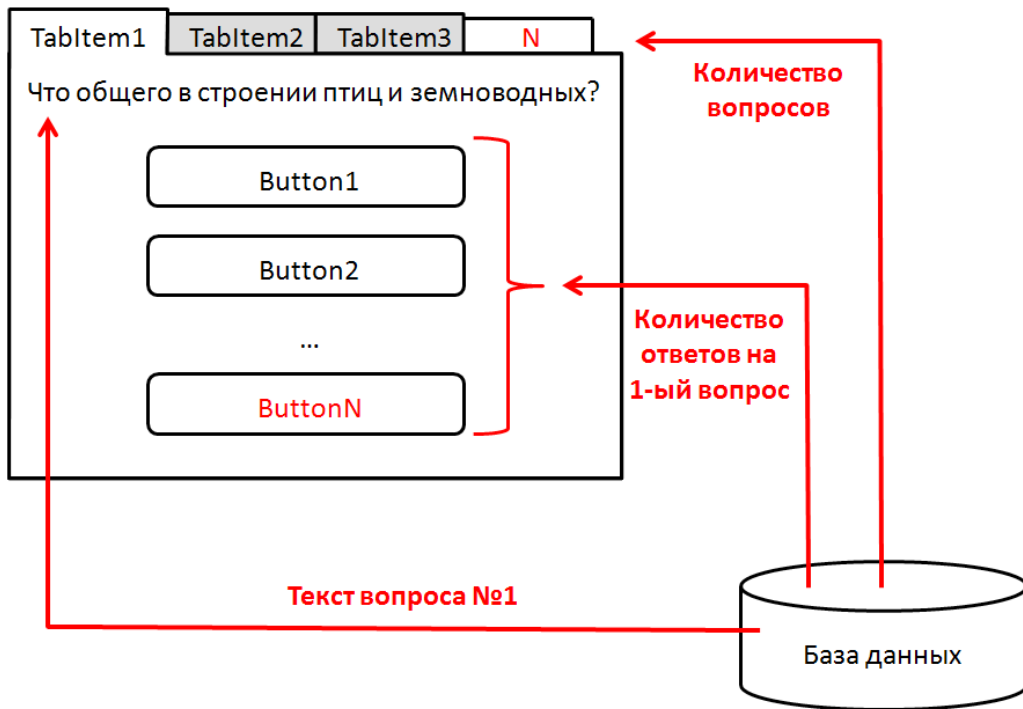


Рис. 2.20. Схема автоматической генерации интерфейса

В таком виде система уже приближается к профессиональному варианту, что требует достаточно глубоких знаний, включая разработку баз данных и методы динамического создания интерфейса. Однако конечному пользователю неважно, создаётся ли интерфейс динамически в процессе работы приложения или в режиме дизайнера при создании проекта. В таком «ручном» изготовлении интерфейса с помощью дизайнера есть ряд преимуществ:

- Такой подход не требует обширных знаний из области инженерии программного обеспечения, поэтому доступен даже для начинающих.
- Это самый быстрый способ создания приложения для реализации теста на проверку знаний.
- Для каждого конкретного вопроса позволяет добавлять поясняющее изображение или несколько изображений, видеоролик, звуковой файл, дополнительный текст, а также различные комбинации перечисленного.

Однако «ручной» подход к созданию приложения для тестирования менее универсален. Каждый раз новый тест придётся создавать на уровне разработки нового приложения или его части в отличие от универсального подхода с динамической генерацией интерфейса. Универсальный подход (рис. 2.20) более трудоёмок с точки зрения создания простого приложения с небольшим числом вопросов. Но потом добавление нового теста сводится к вводу текстов вопросов и ответов в базу данных. При этом эту работу может выполнять обычный пользователь без изменения проекта. Но в этом случае все тесты будут смотреться одинаковыми без каких-либо индивидуальных особенностей.

В этом и есть специфика разработки программного обеспечения: одну и ту же задачу можно решить разными способами, причем нельзя однозначно сказать, хороший данный способ или плохой. Каждый раз решение принимается с учетом времени на разработку, ресурсов и к квалификации. Для создания простого приложения для самотестирования знаний учащегося выбранный простой подход является наиболее эффективным.

2.7. Вывод результатов

Чтобы окончательно завершить наш проект, добавим алгоритм вывода результатов теста. В дизайнере IDE перейдём на закладку TabItem5 компонента TabControl. Поменяем значение свойства Text закладки на «Результат», добавим на страничку компонент TToolBar из раздела Standard палитры компонентов, а также TMemo того же раздела. Для TToolBar свойство Align установите в alBottom, а свойство StyleLookup в значение bottomtoolbar из списка в «Object Inspector». Свойство Align компоненту TMemo установим в значение alClient. В результате компонент TMemo, который представляет собой многострочный текст, распределится по всей оставшейся площади пятой странички (рис. 2.21).

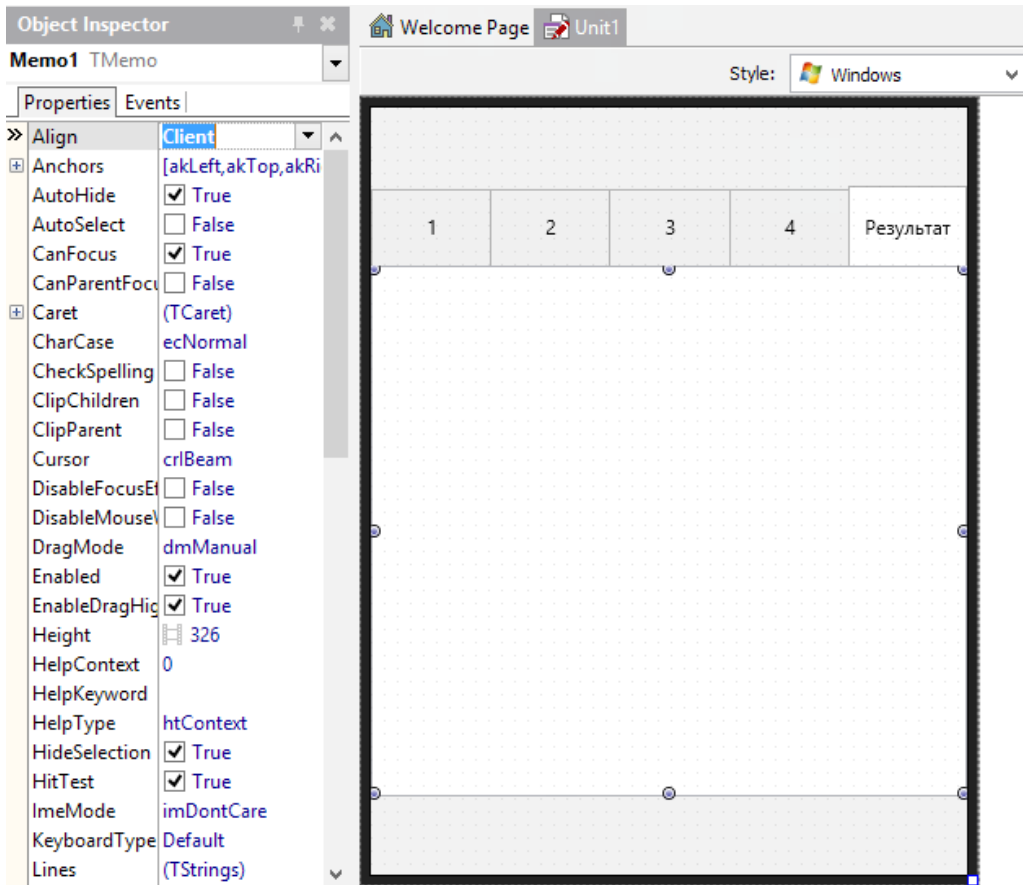


Рис. 2.21. Компоненты странички для отображения результатов

Логика работы данной части приложения будет следующей. Каждый раз, находясь на одной из первых четырёх страничек, пользователь выбирает вариант ответа, результат будет записываться в компонент TМемо. Войдём в код приложения и доработаем каждую из процедур по следующему образцу:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Label2.Text:= (Sender as TButton).Text;  
  if Sender = Button3 then  
    Memo1.Lines.Add('Вопрос № 1—правильно')  
  else  
    Memo1.Lines.Add('Вопрос № 1—неправильно')  
end;
```

Мы добавили код проверки, на какую кнопку произвёл нажатие пользователь. Вспомним, что Sender —ссылка на нажатую кнопку. Поскольку процедура Button1Click назначена сразу четырём кнопкам — Button1, Button2, Button3 и Button4, то именно по значению Sender можно определить, какую из них нажали. В нашем случае Button3 содержит текст правильного ответа, поэтому именно ней мы и сравниваем Sender. Подобным же образом доработаем остальные три процедуры отклика в нашем приложении. Готовое приложение позволяет пользователю выбирать последовательно закладки, отвечать на вопросы при помощи кнопок и видеть результаты ответов на вопросы (рис. 2.22).

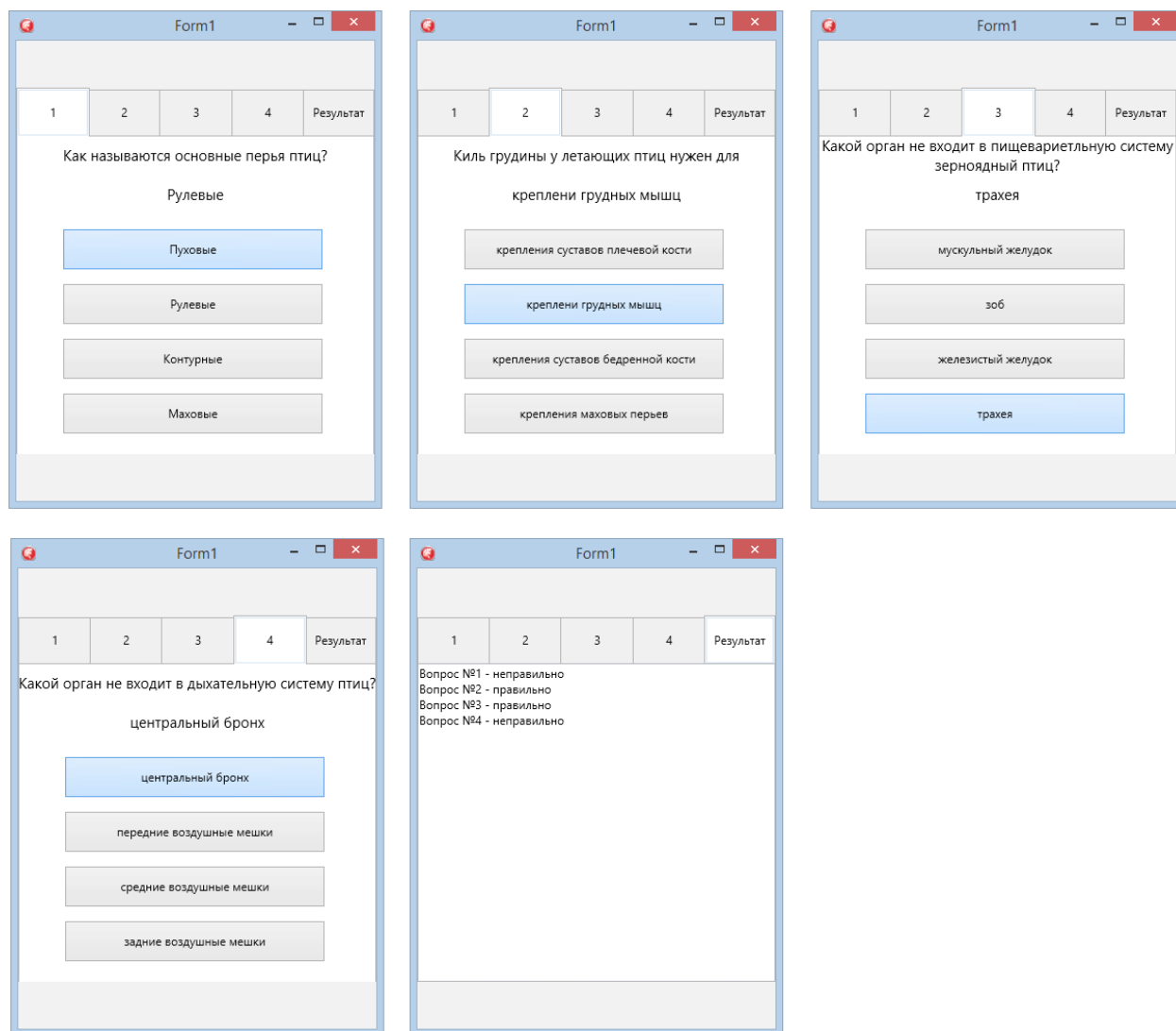


Рис. 2.22. Порядок следования рабочих экранов приложения

Мы решили основную задачу — создали работоспособное приложение, с которым пока может работать только его разработчик. Для обычного пользователя нужно повысить уровень так называемой «эргономики» интерфейса или «удобства использования», а также необходимо развивать приложение и в алгоритмической части, добавляя новые полезные функции. Этому мы посвятим ряд дальнейших разделов.

2.8. Улучшение дизайна приложения

Прежде всего обратим внимание на мелкие детали. Добавим компонент TLabel (Standard) на Toolbar1 (рис. 2.23). Воспользуемся «Object Inspector» и нужным образом зададим значения свойств TextSettings.WordWrap и Text. Можно поэкспериментировать самостоятельно с различными свойствами данного компонента, чтобы дизайн интерфейса полностью соответствовал вашим вкусам и понятиям красоты. Спросите мнение ваших друзей и коллег при условии, что они готовы к конструктивному взаимодействию.

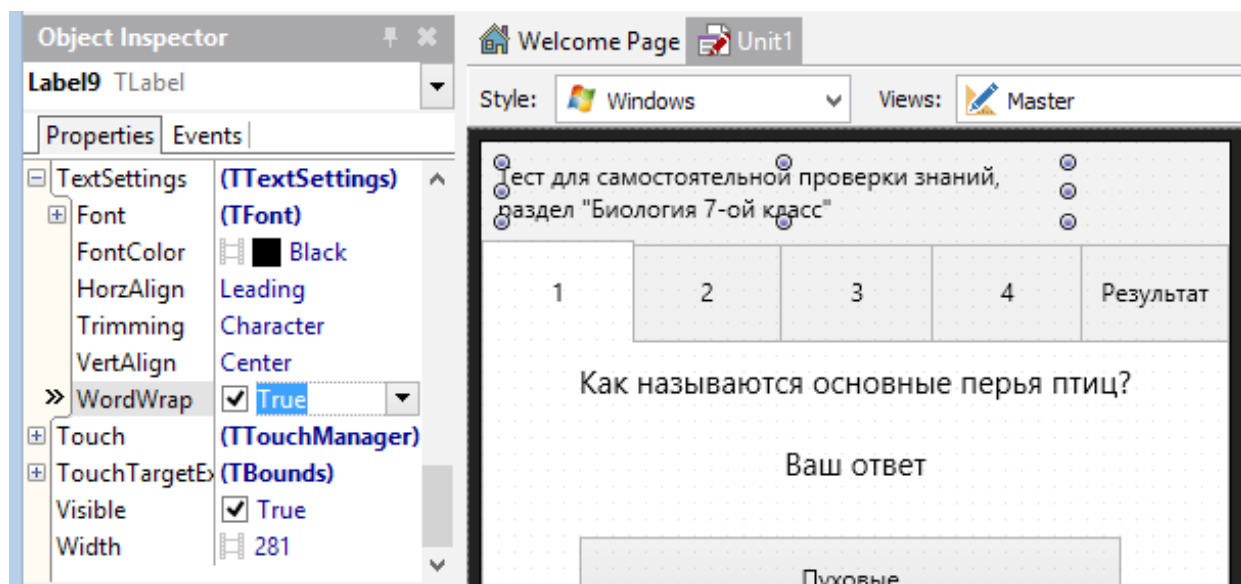


Рис. 2.23. Задание значений свойств компонента TLabel

Обсудим вопрос, связанный с навигацией по страничкам компонента TabControl при помощи закладок (Tab-ов). Пока мы сделали лишь мини-тест из четырёх вопросов. Если таких вопросов будет больше, то интерфейс потеряет свою изначальную простоту, превратившись в «многорядное нагромождение ярлычков». Кроме того, сейчас пользователь может переходить с вопроса на вопрос в произвольном порядке. Однако более эффективным

является сокрытие ярлычков и навигация по компоненту TabControl программным методом.

Выберем компонент TabControl и проанализируем варианты значения свойства TabPosition с помощью «Object Inspector». Изменим значения данного свойства и оценим полученный результат. Поскольку мы собираемся полностью взять контроль над навигацией в свои руки, зададим значение None. Тогда ярлычки исчезнут, внизу на интерфейсе появится ряд небольших кружков, с помощью которых можно переключать закладки в дизайнера. Также это можно делать посредством выбора нужной закладки по названию в панели Structure (рис. 2.24).

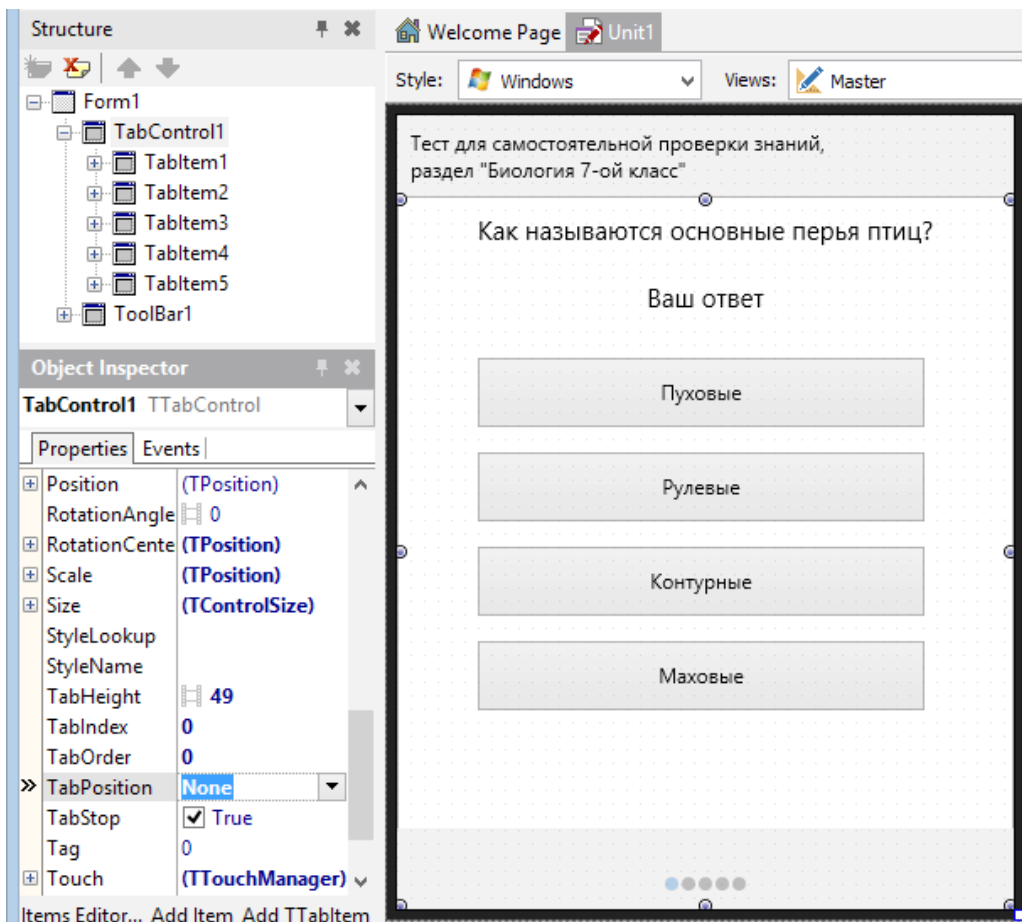


Рис. 2.24. Варианты размещения ярлычков

Выбор размещения ярлычков нужно делать, исходя из конкретной задачи. Если пользователь должен иметь возможность осуществлять навигацию при помощи ярлычков в произвольном порядке, то подойдёт любое свойство кроме None. Но если навигация реализуется программными методами с целью задания жесткого порядка следования страничек, то ярлычки нужно скрыть. Рассмотрим один из вариантов создания элементов управления навигацией из готовых компонентов.

Перейдём на первую страничку TabItem1 компонента TabControl1 либо при помощи маленьких кружочков внизу (которые будут всегда видны в режиме разработки), либо выбором соответствующего пункта в выпадающем списке свойства ActiveTab в «Object Inspector». На данной страничке выделим компонент ToolBar внизу и разместим на нём компонент TSpeedButton из раздела Additional палитры компонентов (рис. 2.25).

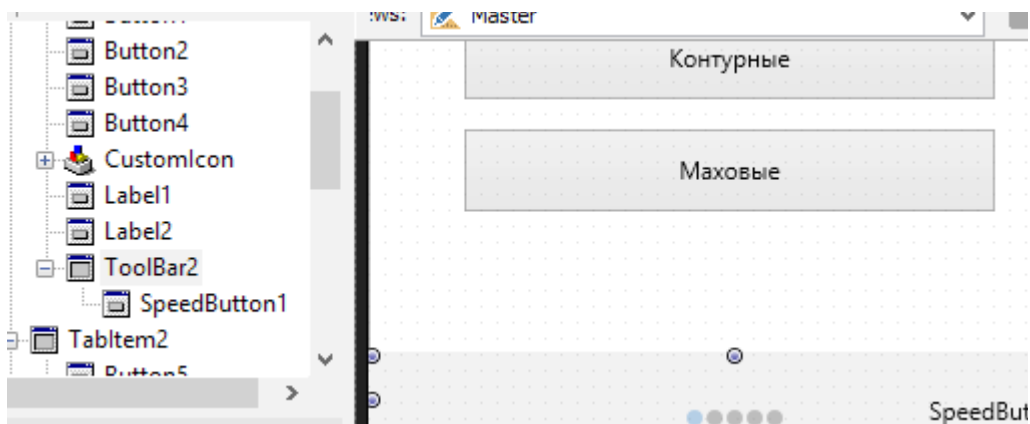


Рис. 2.25. Добавление компонента *TSpeedButton*

Место размещения добавленного компонента SpeedButton1 контролируем при помощи панели Structure. Он должен размещаться именно на ToolBar2, что выражается в иерархически-подчинительном расположении дочернего узла в дереве Structure. Следует пользоваться данным методом определения взаиморасположения компонентов и в других случаях.

Теперь выделим добавленный компонент SpeedButton1 и настроим его дизайн. Это можно сделать традиционным для RAD Studio и Delphi способом, задавая значения нескольким свойствам. Но с появлением мульти-платформенных возможностей разработки такие операции целесообразно проводить следующим способом. Воспользуемся «стилем», который единообразно задаёт весь дизайн компонента. Основное преимущество здесь не только в скорости настройки дизайна, а в автоматическом изменении дизайна при смене целевой платформы приложения (например, с MS Windows на Android).

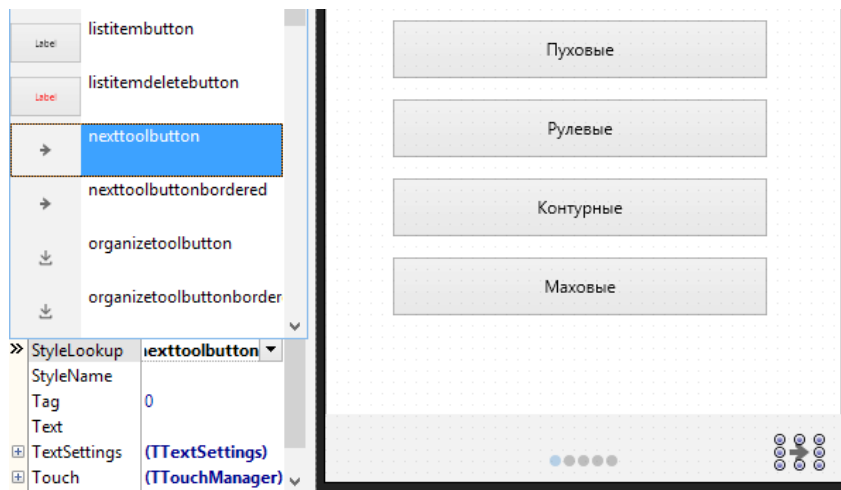


Рис. 2.26. Задание значения свойства *StyleLookup*

Обычно добавленный компонент жестко привязан к тому месту, где его разместили на форме. Если изменяется размер или ориентация формы с портретной на ландшафтную, или наоборот, то визуальный компонент должен автоматически сдвигаться для сохранения целостности интерфейса. Для этого выделим компонент SpeedButton1, найдём его свойство Align в «Object Inspector», и установим значение из выпадающего списка в Right. В результате компонент SpeedButton1 автоматически «прилипнет» к правой части нижнего компонента ToolBar. Теперь если мы теперь изменим

размеры запущенного приложения, то компонент SpeedButton будет постоянно находиться в правом нижнем углу формы (Рис. 2.27).

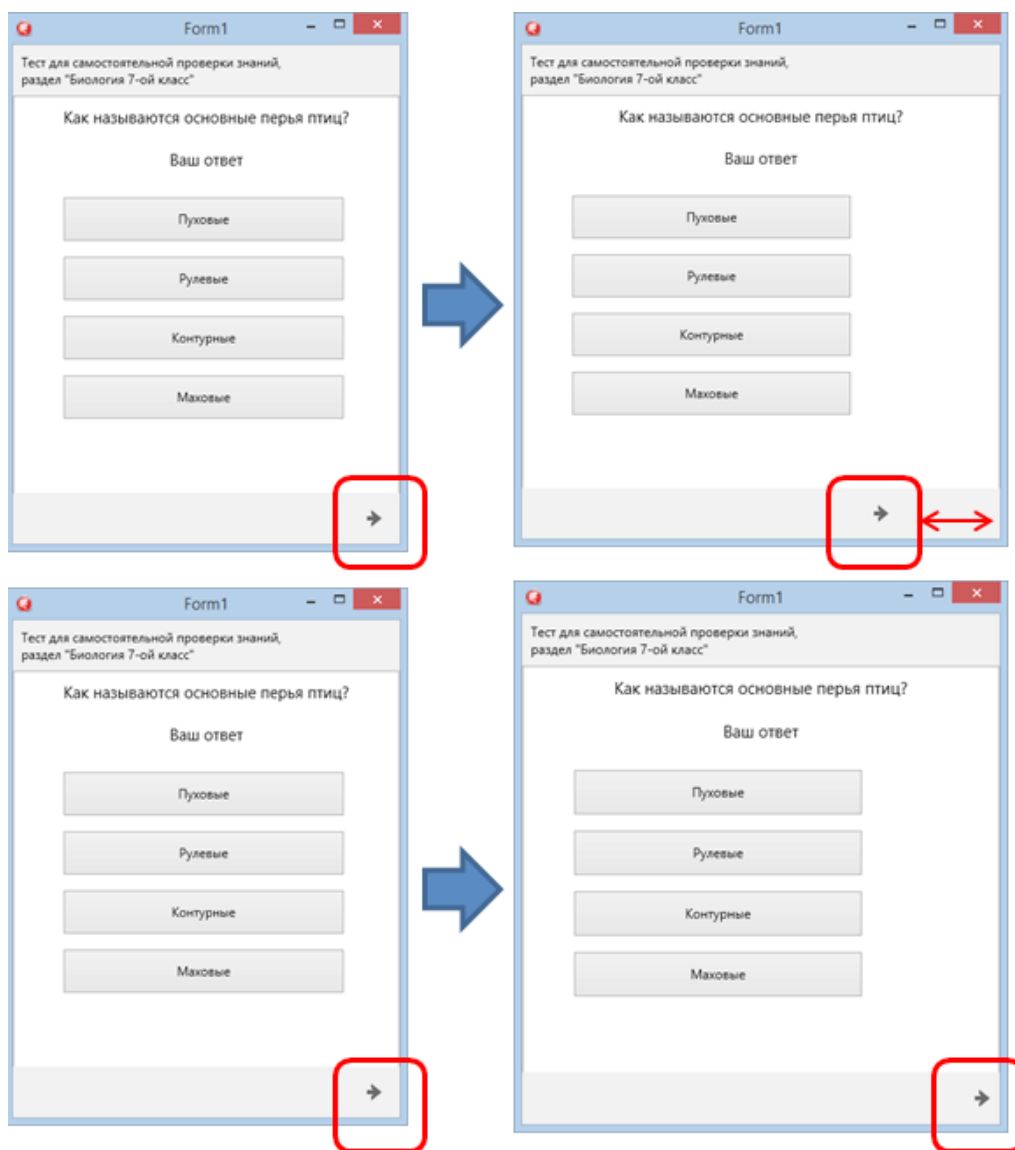


Рис. 2.27. Неправильное и правильное расположение кнопки при изменении размеров формы

Выделив кнопку SpeedButton1, перейдём в «Object Inspector» на закладку Events, кликнем 2 раза в поле OnClick и создадим прототип процедуры отклика на нажатие данной кнопки. В появившемся редакторе кода найдем процедуру, сгенерированную средой IDE, и запишем код для смены странички:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);  
begin  
  TabControl1.ActiveTab:= TabItem2;  
end;
```

В результате работы приведенного кода активной страничкой компонента TabControl1 назначает вторая TabItem2. Теперь вы можете уже самостоятельно на каждой страничке разместить аналогичные компоненты TSpeedButton и задать процедуры отклика на нажатие. Например, для второй странички и кнопки код будет выглядеть так:

```
procedure TForm1.SpeedButton2Click(Sender: TObject);  
begin  
  TabControl1.ActiveTab:= TabItem3;  
end;
```

Необходимо протестировать приложение для различных неповторяющихся вариантах выбора ответов на вопросы, чтобы убедиться в правильности отображения результатов теста на последней страничке.

2.9. Варианты интерфейса пользователя

В данном проекте мы рассматриваем создание мульти-платформенного приложения. Приложение можно собрать как для операционных систем Windows или Mac OS, так и для iOS или Android. При смене целевой платформы среда автоматически применит нужный «стиль», а собранное приложение будет выглядеть аутентично на всех перечисленных платформах. Нашей основной целью является создание мобильного приложения, поэтому мы будем прежде всего обращать внимание на то, как оно будет выглядеть на мобильном устройстве. Вариант сборки под Windows мы будем использовать для тестирования и ускорения компиляции проекта. Когда проект собирается под Windows, это происходит быстрее. Также быстро происходит и запуск проекта, т.к. нет необходимости развертывать приложения на мобильном устройстве, что само по себе является достаточно длительным процессом.

Основная сложность при проектировании интерфейса мобильного приложения заключена в особенности использования смартфонов или планшетов. Большинство пособий по дизайну приложений и web-дизайну, если они не касаются именно мобильных решений, ориентированы на работу мышью или клавиатурой настольного ПК. Мобильное приложение управляется при помощи касаний пальцев рук, что налагает определенные требования к размещению визуальных компонентов, их размерам, взаимному расположению, выравниванию (задаётся свойством Align). Практически всегда по-разному спроектировать интерфейс, основным правилом здесь является достижение максимального удобства пользователя. Рассмотрим следующие варианты улучшения интерфейса.

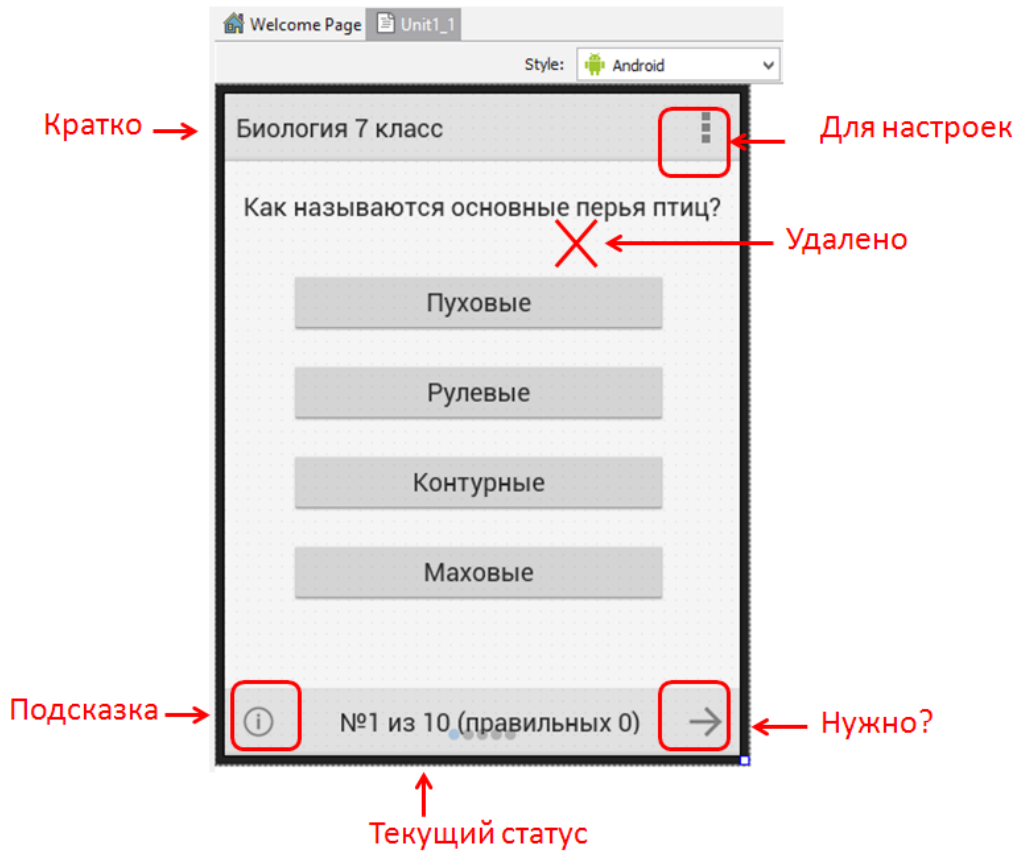


Рис. 2.28. Вариант интерфейса пользователя

Сначала мы переключим дизайнера в режим отображения для ОС Android, как показано на рис. 2.28. Именно в таком режиме мы сможем настроить интерфейс с учетом особенностей именно данной операционной системы. Рассмотрим основные изменения:

1. Название «Биология 7 класс» существенно короче своего предыдущего варианта. Для мобильных приложений характерны краткие названия. Это обусловлено малым абсолютным размером экрана, поэтому длинный текст будет воспроизводиться мелким шрифтом. Мелкий многострочный текст вызывает трудности при чтении с экрана мобильного устройства.

2. Добавился компонент `TSpeedButton` для настроек. Тестирование может проводиться в разных режимах. Например:
 - на каждый вопрос даётся ограниченное время — такой режим можно включить или выключить;
 - при работе «на время» можно включить дополнительные баллы за скорость ответа;
 - можно разрешить или запретить режим пропуска вопросов с возможностью затем вернуться к ним;
 - при необходимости создания «мягкого» режима тестирования можно разрешить пользователю воспользоваться подсказками при нажатии на соответствующую кнопку; подсказки также могут быть с различной степенью конкретизации — от одной до трех подсказок;
 - можно принудить пользователя к повторному прохождению теста по вопросам с ошибками сразу или после демонстрации учебного материала.
3. Удалён компонент `Label2`, имеющий вспомогательное значение (лучше избегать элементов интерфейса, «перегружающих» его дизайн).
4. Добавлена кнопка «подсказка», нажатие на которую демонстрирует дополнительный текст, помогающий пользователю сделать правильный выбор. Возможны варианты: не более 3 подсказок на весь тест; по одной подсказке на каждый вопрос; снижение суммарного балла в случае использования подсказок и т.д. Это уже зависит от логики теста и полноты подготовленных материалов.
5. Добавлен компонент `TLabel` для отображения статуса прохождения опроса в виде номера текущего вопроса из общего числа.
6. Кнопка «следующий вопрос» оставлена, но её назначение требует обсуждения. Если мы принимаем вариант, что пользователь в своих ответах жёстко следует заданной последовательности, то при ответе на текущий вопрос можно переключаться на следующий автоматически. Тогда данная кнопка не нужна. Если разрешить пользователю пропускать вопросы с последующим возвращением к ним, то данная кнопка уже означает не «следующий вопрос», а «пропустить вопрос».

Очевидно, что совершенствовать приложение можно достаточно долго и весьма продуктивно. Мы лишь привели некоторые пути улучшения и повышения удобства использования приложения.

2.10. Приемы повышения качества кода

Компонент `TTabControl` во многих случаях является основой интерфейса мобильных приложений, т.к. он позволяет весьма просто реализовывать и организовывать несколько страничек. Ранее мы рассмотрели достаточно легкий способ навигации, т.е. перехода с одной странички на другую, например:

```
TabControl1.ActiveTab:= TabItem2;
```

Для навигации по всем страничкам нам пришлось на каждой странице размещать нижнюю инструментальную панель `TToolBar`, а уже на ней — отдельную кнопку перехода на следующую страницу. Чем больше будет вопросов в тесте, тем больше будет страниц, тем больше кнопок придётся программировать. Если наше приложение предполагает наличие нескольких однотипных страниц, то эффективнее воспользоваться следующим методом.

Создадим папку «Actions» в «проводнике», затем перейдём в среду RAD Studio или Delphi IDE, создадим новый проект: File->New->Multi-Device Application | Blank Application. На форму добавим компонент `TToolBar`, а затем `TTabControl`. Четыре раза нажмем правой кнопкой мыши на компоненте `TabControl1` и выберите «Add TTabItem» из всплывающего меню. Таким образом, у нас появится 4 странички с закладками. На верхнюю инструментальную панель `ToolBar1` разместим компонент `TSpeedButton` (рис. 2.29).

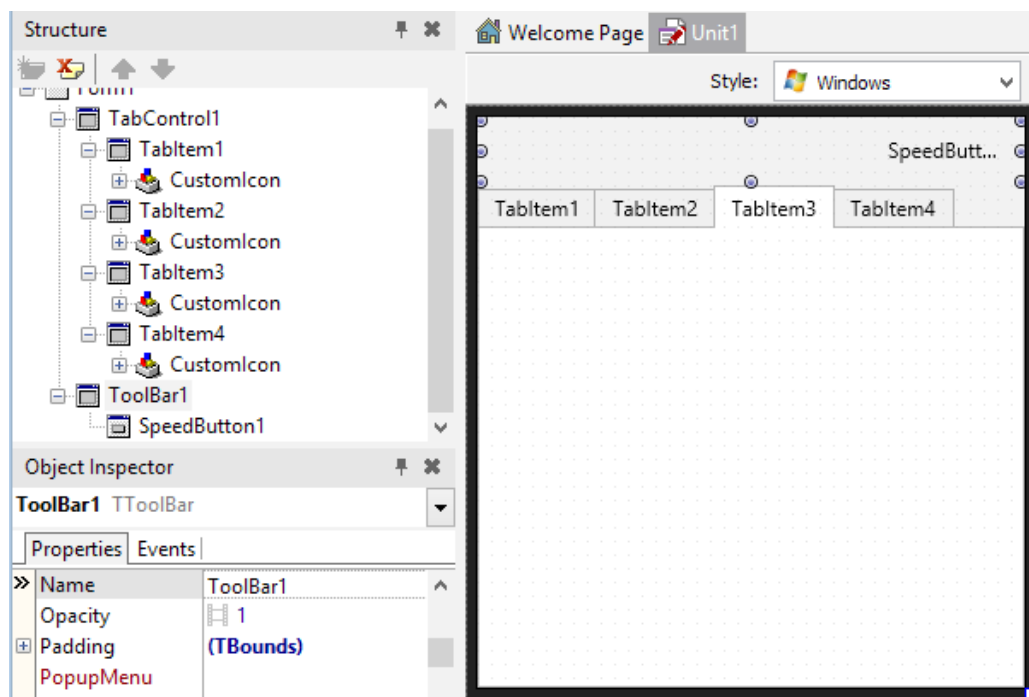


Рис. 2.29. Многостраничный интерфейс с кнопкой навигации

Обратим внимание, что инструментальная панель `ToolBar1` не принадлежит какой-либо страничке и является единой для всех, как и кнопка `SpeedButton1`. Чтобы сделать навигацию универсальной, воспользуемся «стандартным действием». Стандартное действие — это компонент, который уже заранее запрограммирован на выполнение определенных функций. «Стандартные действия» группируются в списки при помощи компонента `TActionList` («список действий»).

Разместим на форме компонент `TActionList`. Данный компонент не является визуальным, он будет виден только в режиме разработки. Затем кликнем на нем 2 раза и в появившемся окне выберем `New->New Standard Action` («добавить новое стандартное действие»), как показано на рис. 2.30. Затем выберем `TNextTabAction` для создания нового «стандартного действия», запрограммированного на переход к следующей страничке компонента `TabControl`.

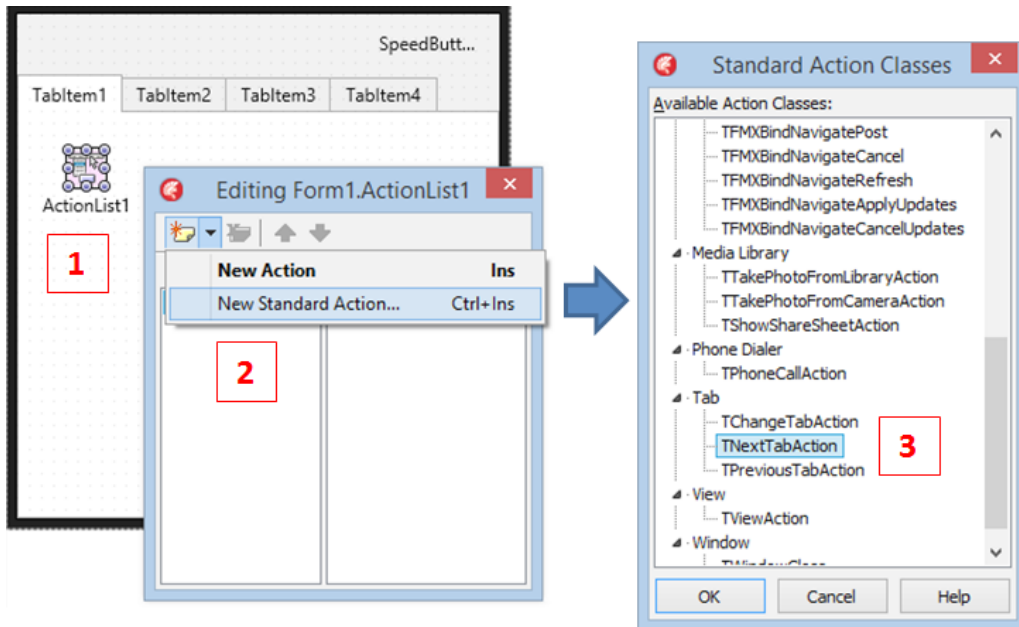


Рис. 2.30. *Добавление нового стандартного действия для навигации*

Мы добавили «стандартное действие», т.е. некую готовую функцию перехода на следующую страничку. Данное «действие» (Action) хранится в «списке действий» (ActionList), но для работы его нужно связать с целевым компонентом. Ошибочно предполагать, что добавленное действие TNextTabAction каким-либо образом самостоятельно начнёт листать TabControl1. Представьте себе, что «действие» есть некий «моторчик». Сам по себе он бесполезен, если его не связать с каким-нибудь механизмом, который он будет приводить в движение. Поэтому мы свяжем «действие» TNextTabAction с компонентом TabControl1.

Выберем в списке ActionList1 только добавленное «стандартное действие» (оно останется выделенным автоматически после добавления и будет называться NextTabAction1), а затем выберем его свойство TabControl как «TabControl1» (рис. 2.31). Нужно обратить внимание на свойства Transition и Direction, смысл которых можно понять из названия или по-

экспериментировать на нашем примере. Свойство `Transition` установлено в списочное значение `Slide`, поэтому при переключении страничек компонента они будут плавно сдвигаться на радость пользователю. Свойство `Direction` определяет направление перелистывания: от младшего номера странички к старшему (прямой порядок) или от старшего к младшему (обратный порядок). Процесс анимированного перелистывания странички, если выбрано значение `Slide` для свойства `Transition`, достаточно затратный с точки зрения ресурсов аппаратного обеспечения. Если потенциальный круг пользователей вашего приложения обладает маломощными дешевыми мобильными устройствами, то «скольжение» закладок может выполняться не всегда плавно. Тогда желательно пожертвовать красотой интерфейса и оставить значение свойства `Transition` как `None`, что приведёт к мгновенной смене страничек.

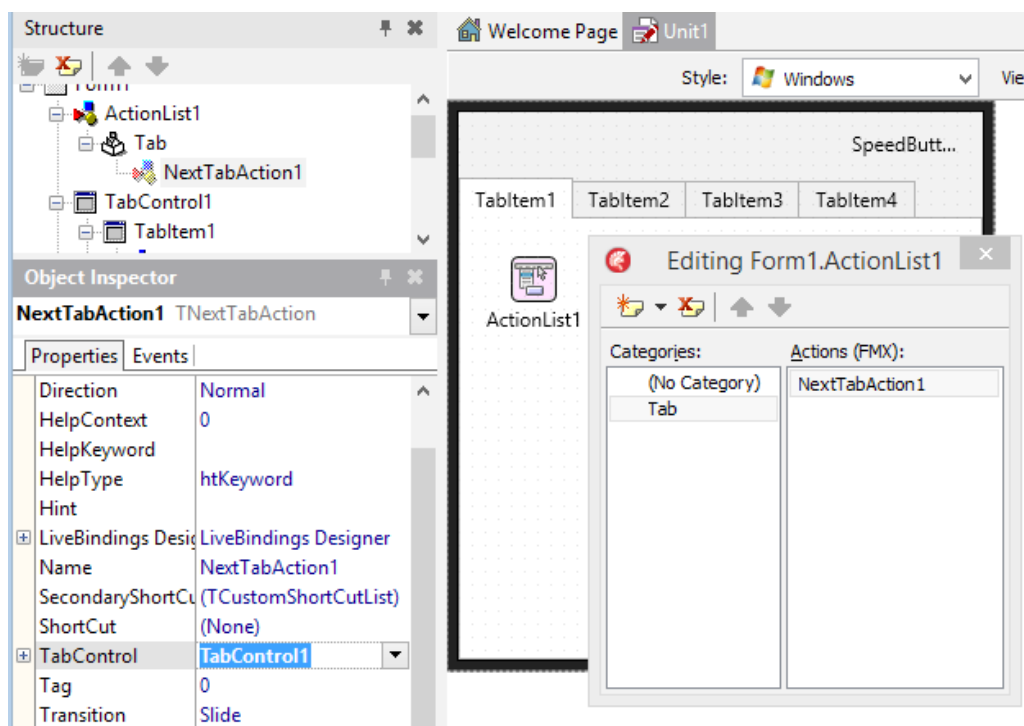


Рис. 2.31. Настройка стандартного действия

Для закрепления еще раз обсудим понятие «действие», воплощенное в виде Action списка TActionList. «Действие» есть некая функция, но не в виде именованной части кода, принимающей параметры, что в Delphi/Pascal вводится ключевым словом function или procedure. «Действие» или Action является компонентом, объектом. Вспомним аналогию с «моторчиком», которую мы представили выше. Моторчик может выполнять заданное действие, если его правильным образом прикрепить к целевому механизму. Для его включения необходимо подключить специальную кнопку.

Мы присоединили NextTabAction1 к PageControl1, теперь «моторчик» будет «листать странички». Осталось реализовать связь с кнопкой, запускающей «моторчик», т.е. NextTabAction1. Такой кнопкой будет SpeedButton1. Для неё мы не будем программировать событие OnClick, изменяющий активную страничку (как мы это делали ранее). Всё это уже реализовано в NextTabAction1, т.к. действие — стандартное. Мы используем связку: «Целевой компонент» — «Действие» — «Элемент интерфейса». В нашем случае это: TabControl1 — NextTabAction1 — SpeedButton1.

Выделим на интерфейсе SpeedButton1 и свяжем её с «действием». Для этого в Object Inspector в свойстве Action данного компонента из списка выберем NextTabAction1 (рис. 2.32). Мы не вводим надпись на кнопке, т.к. она получит её из связанного действия. Само «действие» имеет свойство «CustomText», которое будет передаваться кнопке, связанной с ним. Кликнем два раза на списке действий ActionList1, выберем действие NextTabAction1 и введем «Следующая» в свойство CustomText при помощи Object Inspector. Надпись на кнопке изменится автоматически.

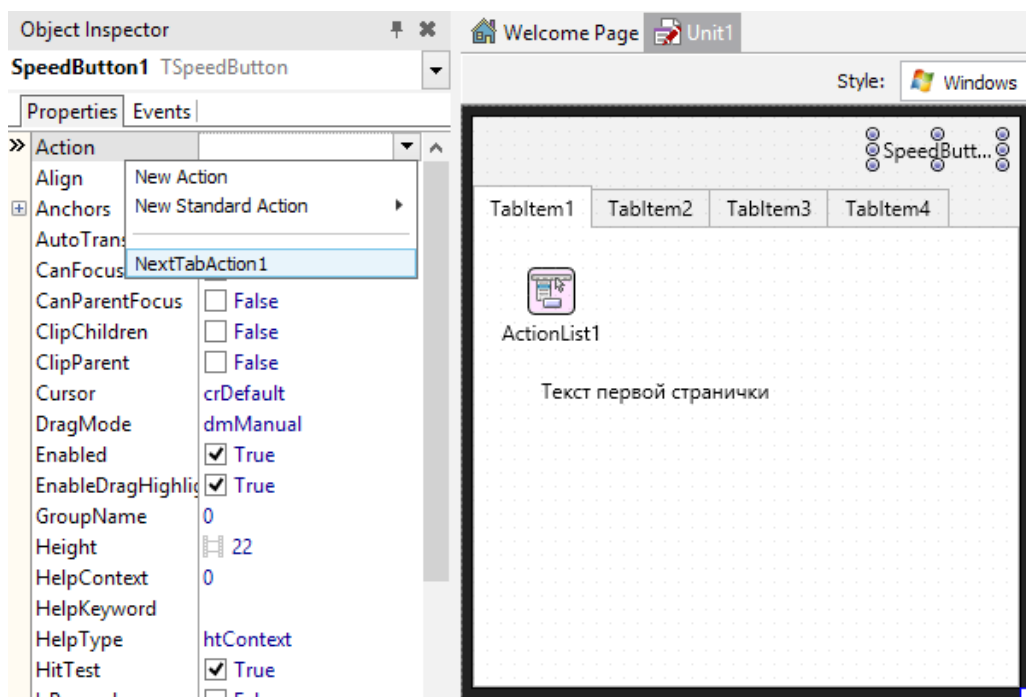


Рис. 2.32. Установка связи между действием и кнопкой

Теперь можно выполнить стандартную процедуру по сохранению и запуску приложения. Быстрая кнопка на верхней панели будет выполнять смену страничек в плавном режиме.

Рассмотренная техника использования стандартных действий (Standard Actions) является эффективной и популярной среди разработчиков. Выделим основные достоинства подобного подхода:

- возможность использования большого числа стандартных действий (как, например, «листание страничек»), что экономит время и силы на реализацию типовых функций;
- централизация многих, полезных функций в едином месте — списке действий (Action List), что позволяет легко ориентироваться даже в большом по объему кода проекте;
- единообразие в оформлении интерфейса на основе общепринятых подходов в программировании.

Для закрепления материала создадим «пользовательское действие», которое в отличие от «стандартного» придётся программировать самим. Щелкнем два раза на компоненте ActionList1, в результате откроется небольшое окно для редактирования действий. Нажмем на кнопку New Action, как показано на рис. 2.33.

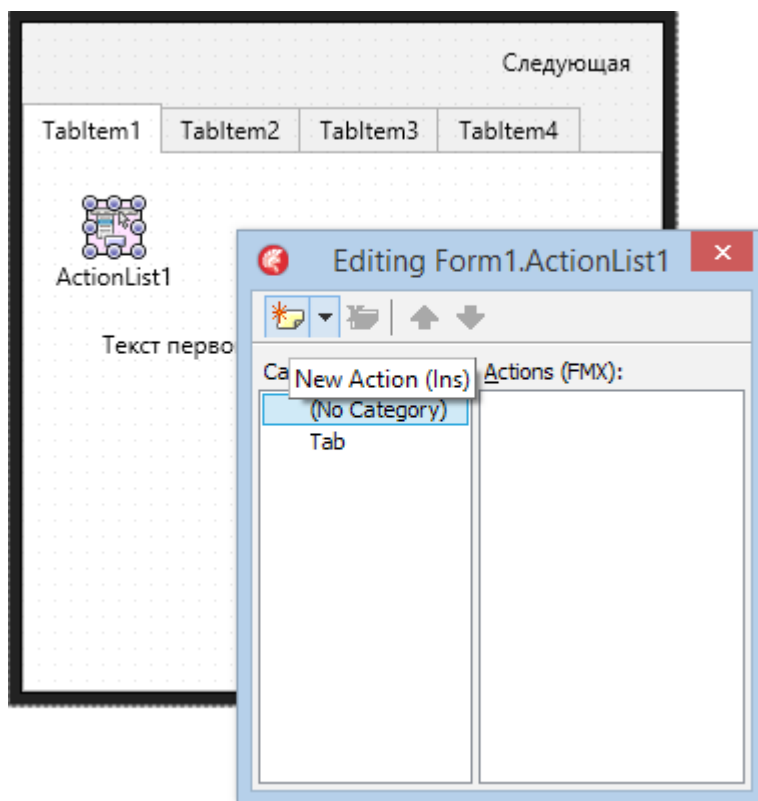


Рис 2.33. Добавление нового «пользовательского действия»

Только что добавленное действие по умолчанию имеет название Action1, оставим его как есть. В большинстве случаев, естественно, созданным действиям нужно задавать особые названия, чтобы не возникло путаницы при обращении к номерным названиям: Action1, Action2, Action3 и т.д. Также не следует путать название действия (свойство Name) с тем, что будет отображаться на связанных компонентах (свойство Text). Выберем свойство текст в Object Inspector и зададим ему значение «Действие 1». Потом перейдём на закладку Events и дважды щелкнем в поле OnExecute. В созданный прототип функции, реагирующей на событие OnExecute, введем следующий код:

```
procedure TForm1.Action1Execute(Sender: TObject);  
begin  
  ShowMessage('Действие выполнено');  
end;
```

Таким образом мы создали новое «действие» Action1, которое будет храниться в списке действий ActionList1 наряду с уже созданным стандартным NextTabAction1. Пока действие Action1 не связано ни с каким элементом управления — визуальным компонентом формы интерфейса, поэтому выполнить его будет нельзя. Чтобы выполнить готовое действие, разместим на форме несколько элементов управления и свяжем их с Action1.

На первую закладку поместим компонент TSpeedButton и компонент TButton, которые находятся в разделах Standard и Additional палитры компонентов. Далее, как показано на рис. 2.34, выберем поочередно эти компоненты и свойству Action зададим значение Action1 из списка при помощи ObjectInspector. Теперь кнопки получают не только надписи, соответствующие связанному «действию», но и при нажатии будут инициировать выполнение кода события OnExecute действия Action1.

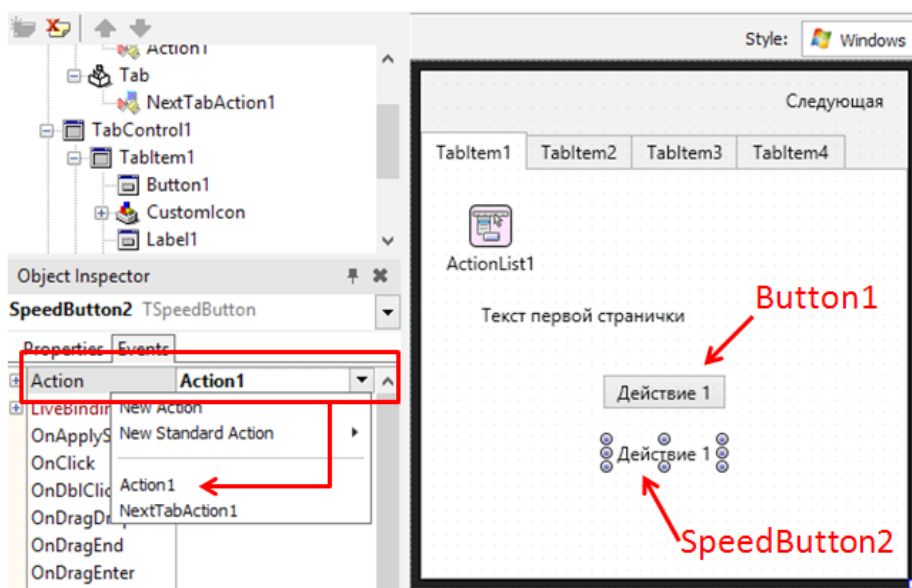


Рис. 2.34. Установление связи между действием и элементом управления

Скомпилируем и запустим приложение. При нажатии Button1 или SpeedButton2, сработает одно и то же связанное действие Action1 (рис. 2.35).

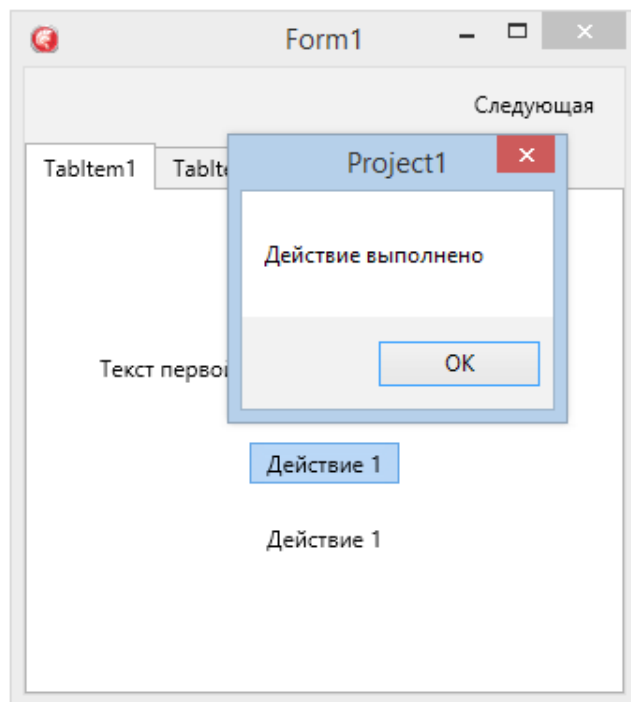


Рис. 2.35. Результат работы «действия»

Кратко перечислим последовательность операций при использовании «действий» Action:

- добавляем новое действие Action в список действий ActionList;
- определяем атрибуты его визуального представления в элементах управления (свойство Text);
- реализуем процедуру отклика на событие срабатывания действия OnExecute;
- в соответствии с дизайном интерфейса определяем или добавляем визуальный компонент (элемент управления), который будут связан с данным действием;
- связываем свойством Action в Object Inspector элемент управления и действие;
- контролируем правильность работы, запустив и протестировав приложение.

Описанная последовательность шагов позволит правильно применять «действия», что позволит иметь ясную структуру кода и компонентной модели приложения. В дальнейшем это значительно сократит усилия на развитие и сопровождение программного продукта.

Анимация

3.1. Основы анимации в Delphi/RAD Studio/C++Builder

Анимация с точки зрения программирования в Delphi/RAD Studio/C++Builder есть придание движения какому-либо визуальному объекту. Такой объект можно назвать «анимированным», т.е. приводимым в движение или движущимся. Анимировать объект в большинстве случаев означает изменить его местоположение, задаваемое координатами. Если объект двумерный или плоскостной, то его положение на форме или другом компоненте определяются значениями свойств `Position.X` и `Position.Y`. Если объект пространственный, то в комплексном свойстве `Position` добавляется ещё одна координата `Position.Z`. Займемся анимацией объектов на плоскости, т.к. с точки зрения придания движения нет различий, являются ли объекты двумерными (2D, 2-dimensional) или трехмерными (3D, 3-dimensional).

Анимация используется в игровых или развлекательных приложениях, но этим её применение не ограничено. Интерфейсы современных бизнес-приложений, научных программы, баз данных и операционных систем также содержат элементы анимации. Окна «всплывают», меню «раскрывается», кнопки изменяют свой цвет и размер и т.д. Анимация присуща любому визуальному компоненту или элементу управления. В Delphi/RAD Studio/C++Builder механизм придания движения объектам или анимация была добавлена прежде всего для создания современных и привлекательных интерфейсов. Данный механизм является универсальным, поэтому может быть использован и для создания простых игр и интерактивных приложений.

Рассмотрим технику анимации простых объектов. Создадим папку на жестком диске, запустим Delphi, в главном меню выберите `File->New Multi-Device Application` и сохраним проект в созданную папку «Project 3.1». На

форму добавим компонент TCircle, который представляет собой «кружок». Разместим его так, как показано на рис. 3.1.

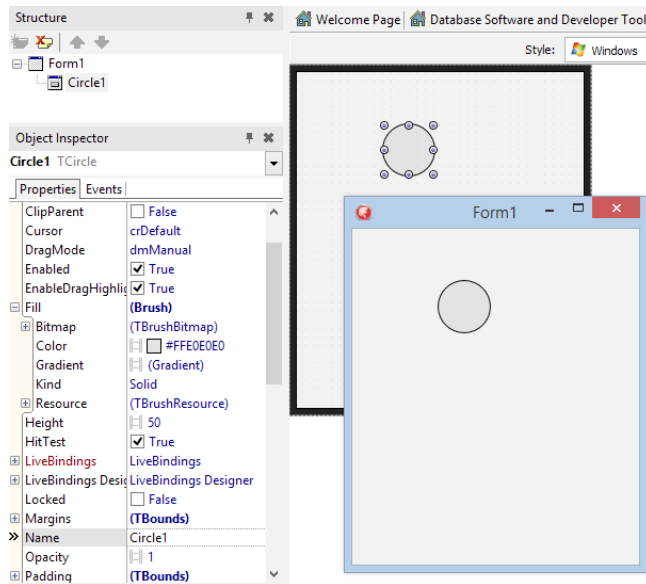


Рис. 3.1. Форма с компонентом TCircle в дизайнера и приложении

Мы будем анимировать «кружок», т.е. компонент Circle1: TCircle. Если мы перемещаем визуальный компонент по форме в design-time, то изменяется его свойство Position, представленное двумя составляющими Position.X и Position.Y. Соответственно, если мы в runtime будем изменять программно значение свойства Position.Y, то «кружок» будет двигаться по вертикали, т.е. вверх-вниз. В Delphi/RAD Studio/C++Builder для этого есть специальные компоненты, которые можно настроить в design-time во избежание ручного ввода кода.

В нашей задаче «кружок» изображает «мячик», падающий сверху вниз на твёрдый пол и отскакивающий обратно вверх. Посмотрим на рис. 3.2, чтобы понять, каким образом нужно изменять координаты для достижения описанного эффекта. Из рисунка ясно, что изменятся значения координаты Position.Y от крайнего верхнего в крайнее нижнее положение. Это из-

менение должно происходить во времени, чтобы создавался эффект движения. Если координаты изменятся мгновенно, то мы не увидим движение.

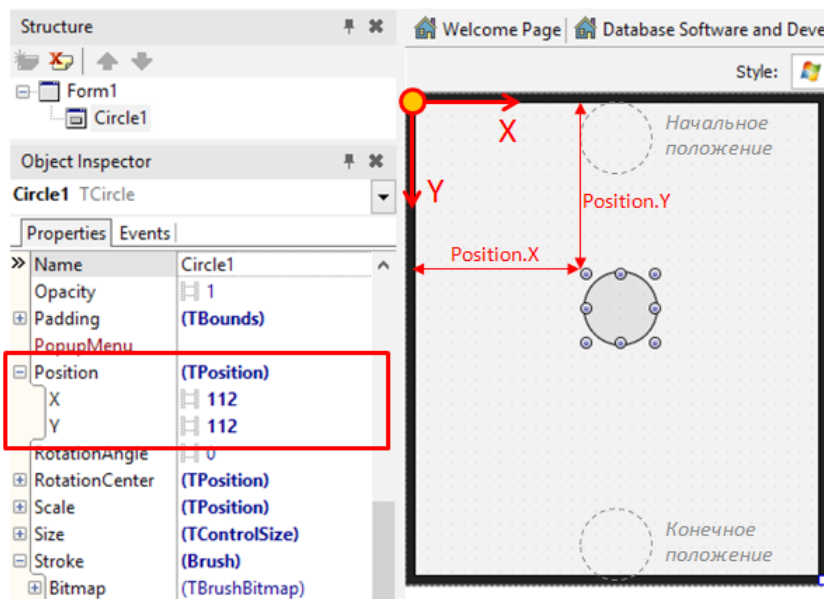


Рис. 3.2. Схема анимации

Поместим кружок в начальное положение, где значение координаты Position.Y равно 0. А вот с конечным значением координаты Position.Y не так все очевидно, т.к. размеры формы могут быть разными. Расположим компонент в конечном положении и запомним значение Position.Y. В данном примере это значение равно 264.

Теперь реализуем анимацию. Мы знаем начальное значение координаты Position.Y, её конечное значение, а длительность процесса изменения подберём так, чтобы кружок изображал падение реального сферического объекта. Существуют различные способы задать изменение координаты Position.Y во времени. Мы воспользуемся самым простым — поместим на форму специальный компонент TFloatAnimation. Следует обратить внимание, что при добавлении компонента TFloatAnimation он попадает на

главную форму. Но нам нужно, чтобы он был привязан именно к TCircle, т.к. именно его он и будет анимировать. После добавления компонента TFloatAnimation обратимся к панели Structure, захватим его мышью и переместим на узел TCircle, как показано на рис. 3.3.

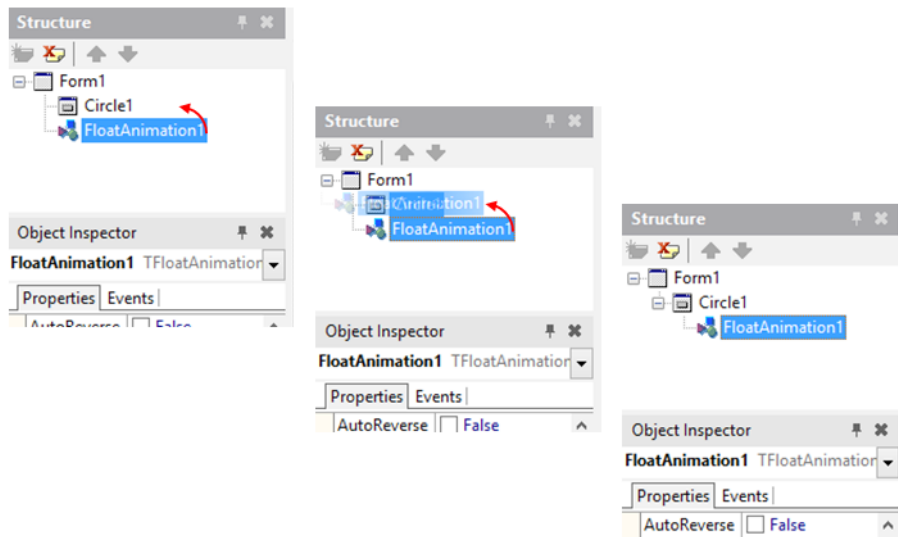


Рис. 3.3. Правильное размещение компонента для анимации

Для понимания взаимодействия «обычного» визуального компонента (в нашем случае — Circle1) и анимационного (FloatAnimation1) воспользуемся аналогией «трактор» — «объект». Как только трактор тросом привязан к какому-либо объекту («бревно», «бочка», «тележка», «швеллер» и т.д.), то при движении он будет перемещать связанный объект. FloatAnimation1 представляет собой «трактор», а Circle1 — движимый им объект. При этом компонент для анимации — TFloatAnimation — может быть связанным только с одним объектом. Слово Float в названии компонента TFloatAnimation означает, что изменяться во времени будет «вещественное» свойство, т.е. Position.Y.

Оставим выделение на компоненте FloatAnimation1 (рис. 3.3) и выполним настройку его свойств в Object Inspector, как показано в таблице ниже.

Свойство	Значение	Объяснение
AutoReverse	True	Как только «мячик упадёт вниз», то сработает «автоматическое изменение направления движения», и он «отскочит вверх»
Duration	1	Длительность анимации, т.е. время, за которое кружок сместится из начального положения в конечное; задаётся в секундах
Enabled	True	Анимация включена
Interpolation	Quadratic	Выбрать из списка «квадратичный» тип анимации
Loop	True	Анимация будет зациклена, т.е. «мячик» будет скакать бесконечно долго
PropertyName	Position.Y	Свойство связанного компонента (Circle1), которое будет изменяться во времени
StartValue	0	Начальное значение Position.Y анимированного компонента Circle1
StopValue	264	Конечное значение Position.Y анимированного компонента Circle1

Рассмотрим свойство Interpolation и его значение Quadratic. Изначально значение будет Linear (по умолчанию). Попробуем запустить приложение именно с этим значением. Мы увидим, что кружок равномерно движется сверху вниз и снизу вверх. Это выглядит вполне красиво, но не соответствует тому, как движутся объекты под действием силы тяжести. Вспомним простую формулу $y = -g \cdot t^2$, которая позволяет рассчитать координату тела в зависимости от времени при свободном падении. Степень 2 принято называть «квадратом числа», что соответствует слову Quadratic в английском языке. Именно такая «квадратичная» зависимость используется для моделирования падающего тела (в виде перемещения кружочка). Свойство Interpolation задаёт метод расчета промежуточных положений между начальным (StartValue) и конечным (StopValue) значением во времени.

3.2. Исследовательский проект с анимацией

Выполним интересный проект с исследовательской частью, где анимация будет использоваться в полном объеме. Попутно рассмотрим некоторые интересные и полезные особенности IDE, а также компонентной модели Delphi/RAD Studio/C++Builder. На рис. 3.4 представлена графическая композиция, иллюстрирующая оптическую иллюзию. Суть иллюзии будет видна, как только мы создадим соответствующее приложение.

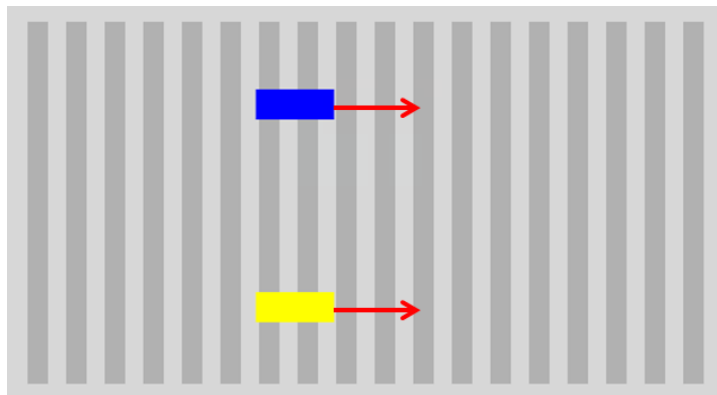


Рис. 3.4. Оптическая иллюзия неравномерности движения

На рис. 3.4 красными стрелками обозначены направления движения двух цветных прямоугольников («машинок»). Эти «машинки» движутся параллельно и с одинаковой скоростью. Если сосредоточить внимание на синем прямоугольнике и следить за ним неотрывно, то будет казаться, что нижняя «машинка» отстает и движется рывками. Данный эффект возникает вследствие «полосатости» дороги, по которой едут «машинки». Если полосы убрать, то пропадёт и сама иллюзия.

Создадим новый проект и сохраним его в соответствующую папку. Выберем главную форму, найдём свойство Fill в Object Inspector, а затем его вложенное свойство Color. Изменим значение данного свойства на LightGray. Найдём на палитре компонентов TRectangle в разделе Shapes

и поместим его на форму. Зададим его высоту (свойство Height) как 300, а ширину (свойство Width) как 17. Свойство Fill.Color установим в значение из списка DarkGray. Цвет контура прямоугольника зададим как DarkGray (свойство Stroke.Color). После этого создадим еще один такой же прямоугольник и разместите его на расстоянии 15 от первого. На форме должны быть видны две темные вертикальные полосы. Для получения 18 полосок (а именно это нам и нужно) придётся добавлять ещё 16 в ручном режиме. Но существуют как минимум ещё 2 способа. Один из них — создавать и размещать полосы динамически в цикле на этапе исполнения приложения.

Второй способ не требует программирования и подходит для начинающих. Выделим в дизайнере две полосы, кликнув на них поочередно мышью с нажатой кнопкой Shift на клавиатуре. Далее выполним типовую операцию копирования/вставки, например, при помощи всплывающего меню (правая кнопка мыши, Edit->Copy/Edit->Paste) или комбинации клавиш Ctrl+C/Ctrl+V. Затем нужно аккуратно «подцепить» новую пару полосок и передвинуть их в нужное место (рис. 3.5). Теперь повторим эту операцию уже для четырёх полосок. В считанные секунды у нас уже полностью заполненное поле!

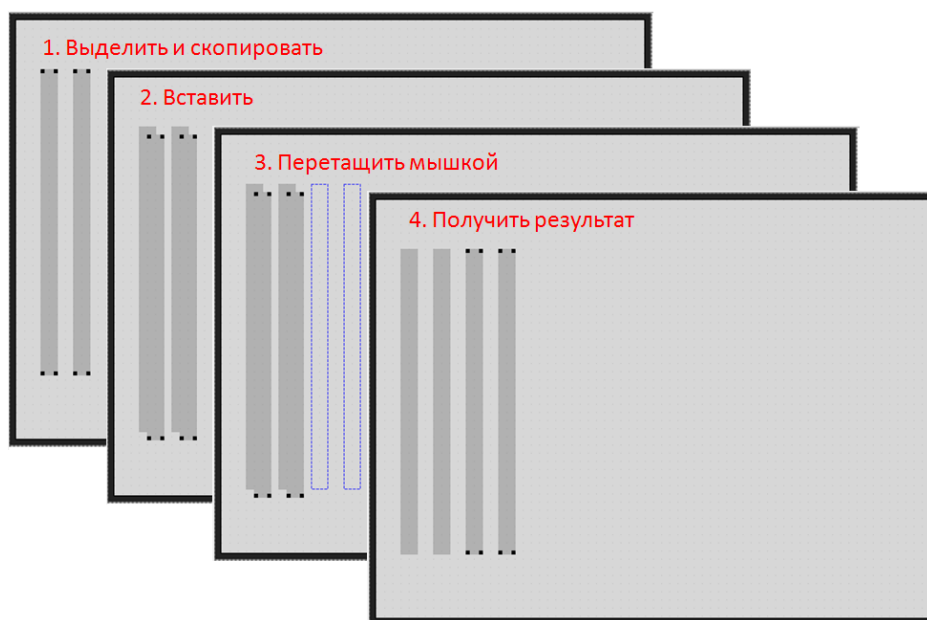


Рис. 3.5. Копирование-вставка визуальных компонентов

При перетаскивании компонентов мышью они перемещаются «скачками», от одного узла координатной сетки к другому. Координатная сетка — это еле заметные серые точки, равномерно покрывающие форму. Если нужно передвинуть объект не от одного узла к другому, а всего на единицу, то можно изменить вручную свойство `Position.X` или `Position.Y`, введя соответствующее число в Object Inspector. Такого же эффекта можно достичь, выделив компонент и нажав `Ctrl`+стрелка (на клавиатуре: вверх, вниз, вправо, влево). Если нажать кнопку `Shift`, а не `Ctrl`, то также можно изменить размер компонента на единицу.

После того, как форма будет заполнена полосками, добавим еще два прямоугольника, изменив их цвет при на синий и желтый и задав их размер соответствующим образом. Изменим их названия — свойства `Name` — на `recBlue` и `recYellow`. Начальное положение прямоугольников должно соответствовать рис. 3.6. Добавим на форму кнопку `Button1`. Сохраним и запустим приложение на исполнение, проверив тем самым готовность проекта к добавлению анимации.

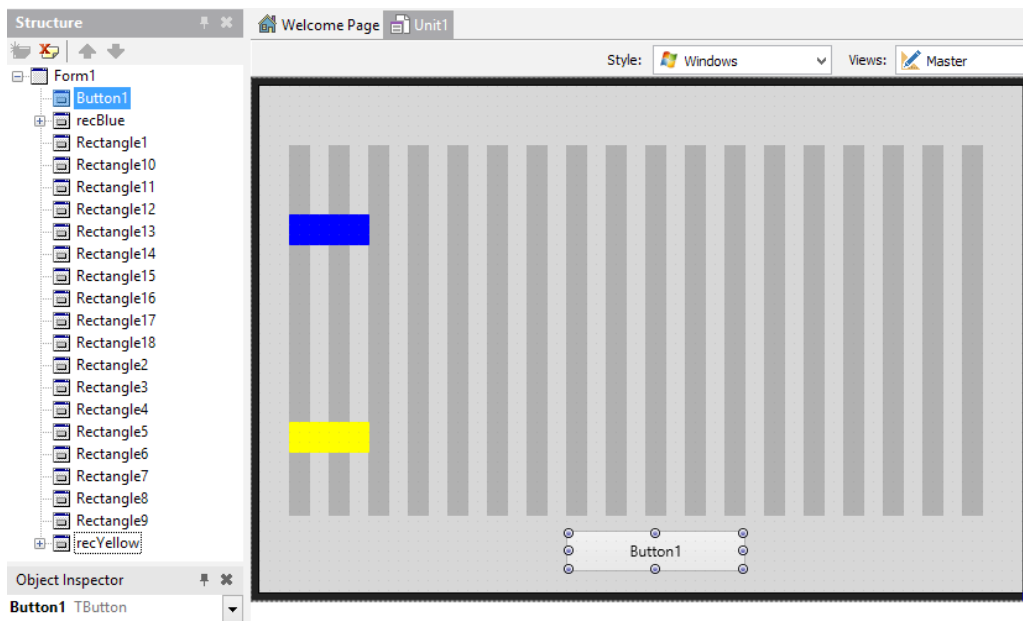


Рис. 3.6. Статичная картинка до добавления анимации

Наша задача — сделать так, чтобы «машинки» двигались одновременно слева направо. Очевидно, что для этого их свойство `Position.X` должно изменяться во времени. Тип значения свойства `Position.X` является вещественным, поэтому мы будем использовать уже привычный компонент `TFloatAnimation` с раздела палитры компонентов `Animations`. Можно ускорить процесс поиска нужного компонента по названию по первым символам. Палитра обладает полем, ввод первых символов в которое будет вызывать автоматический поиск нужного компонента. Если нескольким первым буквам соответствуют несколько компонентов, то они будут все показаны. Но искать среди черырёх вариантов будет гораздо проще (рис. 3.7).

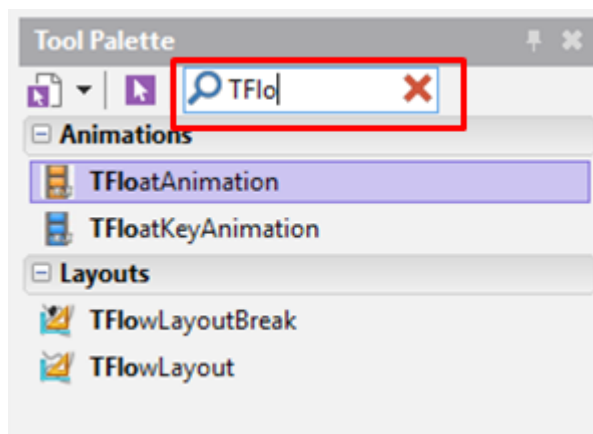


Рис. 3.7. Поиск компонента по первым символам названия

Щёлкнем два раза на компоненте `TFloatAnimation` в палитре, тогда он появится в структуре объектов формы (панель `Structure`). Изначально `TFloatAnimation` попадает непосредственно на форму, а затем его нужно аккуратно «оттащить» при помощи мыши в нужное место на панели `Structure`. У нас этим «нужным местом» является компонент `resTop`, который представляет собой верхний синий прямоугольник. Процедура перетаскивания компонента показана на рис. 3.8.

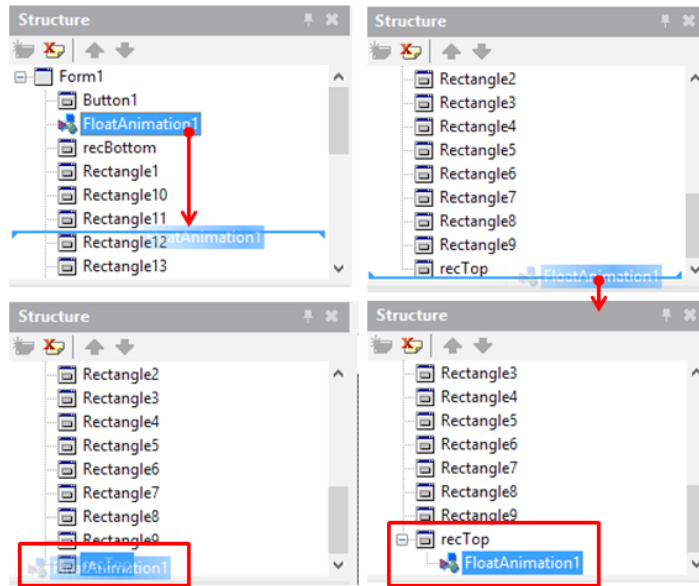


Рис. 3.8. Процедура перетаскивания компонента *FloatAnimation1*

После того, как компонент *FloatAnimation1* перемещен на *recTop*, он становится связанным с целевым прямоугольником. Далее заданием значения его свойств в соответствии с таблицей ниже:

Свойство	Значение	Объяснение
Duration	11	Длительность анимации, т.е. время в секундах, за которое прямоугольник переместится из начального положения в конечное
Enabled	True	Анимация включена
PropertyName	Position.X	Свойство связанного компонента (<i>recTop</i>), которое будет изменяться во времени
StartValue	16	Значение <i>Position.X</i> в начальном положении синего прямоугольника
StopValue	592	Значение <i>Position.Y</i> в конечном положении синего прямоугольника

Прделаем ту же самую процедуру со вторым компонентом *FloatAnimation2*, начиная от добавления его на форму и заканчивая связыванием с компонентом *recBottom*. Значения его свойств задаются по аналогии с тем, что показанов в таблице выше. Сейчас мы не будем изменять

значение *Interpolation* по умолчанию, установленное как *Linear* («линейная»). Это означает, что прямоугольник будет двигаться равномерно, без ускорений или замедлений.

Сохраним проект и запустим приложение на исполнение. Прямоугольники автоматически начнут двигаться, а если сконцентрировать взгляд на верхнем, то будет казаться, что нижний отстает от верхнего и перемещается рывками. Если удалить серые вертикальные полосы с формы, то иллюзия отставания и движения рывками пропадет.

Разместим на форме кнопку *Button1*. В качестве процедуры отклика на событие *OnClick* нужно ввести код, который будет скрывать прямоугольники-полоски. Выполните двойной щелчок в пустом поле *OnClick* на закладке *Events*, и интегрированная среда разработки сгенерирует шаблон процедуры.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  
    end;
```

Если мы хотим скрыть какой-нибудь компонент из виду, то для этого существует несколько способов. Сформулируем принципы выбора оптимального варианта:

- код должен быть максимально компактным (чем меньше кода, тем лучше);
- код должен быть удобочитаемым (максимально понятным);
- операторы, алгоритмы и компоненты должны использоваться по назначению.

Далее мы будем рассматривать методы сокрытия объекта в соответствии с указанными выше принципами.

3.3. Эффективный код или как правильно скрыть объект

Если выделить компонент `Rectangle1` и посмотреть в `Object Inspector`, то там можно обнаружить множество свойств даже для такого простого объекта, как прямоугольник (рис. 3.9). Начинающие программисты испытывают дискомфорт из-за незнания всех свойств компонентов, которыми им приходится оперировать. Но не стоит изучать все свойства подряд с использованием доступных источников информации, например, справочной системы. Такой путь ошибочный, т.к. многие свойства имеют свой смысл только в контексте решаемых задач. Лучше изучать свойства по мере необходимости реализации конкретных задач. Сейчас мы одну из таких задач: сокрытие прямоугольника.

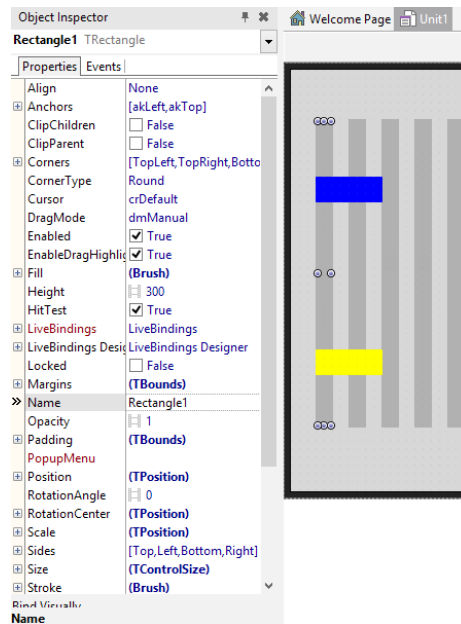


Рис. 3.9. Свойства объекта «прямоугольник»

Убрать или скрыть из виду прямоугольник можно различными способами. Исходя из принципов, указанных в разделе 3.2, мы должны стремиться-

ся к минимизации объема вводимого кода. Если можно решить задачу за счет одной строчки кода, то лучше так и сделать. Далее, код должен быть максимально простым. Есть такие профессионалы, которые, сокращая объем вводимого кода, выдают настолько замысловатые конструкции, что их весьма сложно понять. Для языка Pascal/Object Pascal/Delphi это не такая уж беда, в отличие от языка C++, где можно составлять настоящие «ребусы». Однако и здесь нужно писать программы максимально просто.

Современные программы пишутся многими людьми. При усовершенствовании программы приходится возвращаться к исходному коду и исправлять, модифицировать его, даже если код создавался другим разработчиком. Чем проще будет исходный код, тем меньше будут временные и финансовые затраты на эту работу.

Теперь проиллюстрируем третий принцип, рекомендующих использовать операторы, алгоритмы и компоненты по назначению. Чтобы убрать прямоугольник из виду можно его свойству `Position.X` присвоить какое-нибудь гигантское значение. Например, следующий код при нажатии кнопки `Button1` приведёт к такому эффекту:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Rectangle1.Position.X:= 10000;  
end;
```

При таком значении координаты `Position.X` прямоугольник переместится далеко за пределы формы (окна), и пользователь его не увидит. Будет ли этот код правильным? С точки зрения достигнутого эффекта — несомненно. С точки зрения понятности и логичности — сомнительно. Представим себе ситуацию, когда данный код написал один программист, а разобраться в нём должен другой. Допустим, мы видим такой код:

```
procedure TForm1.Button100Click(Sender: TObject);  
begin  
  Rectangle200.Position.X:= 100000;  
end;
```

Что может понять непосвященный программист? Что при нажатии на кнопку Button100 прямоугольник Rectangle200 перемещается в точку с координатой X, равной большому числу 100000. Зачем он перемещается туда, явно за границу формы? Почему «сто тысяч», а не «миллион» или «десять миллионов»? Что будет с логикой работы, если изменить данный код? Из данного кода неясно, планировалось ли перемещение или сокрытие прямоугольника.

Вот тут нам пригодятся другие свойства из Object Inspector. Как минимум, более логичным будет поискать подходящее свойство по названию. Например, свойство Visible, которое встречается практически у всех визуальных компонентов. Установка его значения в «ложь» Visible:= false будет скрывать объект от зрителя. Причем для этого необязательно программировать, достаточно в design-time «выключить» свойство (установив его в false). Даже запускать приложение не нужно!

Переработает код отклика на кнопку Button1 следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.Visible:= false;
  Rectangle2.Visible:= false;
  // ... и так далее до Rectangle 18
end;
```

Однако можно пойти дальше, и исследовать свойство Opacity. В переводе с английского оно означает «непрозрачность». По умолчанию для непрозрачных объектов Opacity равно 1. Можно догадаться, что при значении 0 объект будет полностью прозрачным, т.е. невидимым. Если Opacity равно 0.5, то объект будет полупрозрачным. Основываясь на этом свойстве, можно получить вполне работоспособный код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.Opacity:= 0;
  Rectangle2.Opacity:= 0;
  // ... и так далее до Rectangle18
end;
```

С точки зрения логики использование свойства Visible представляется более правильным. Изменение степени непрозрачности за счёт свойства Opacity нужно, когда за данным объектом скрывается другой, а нам его нужно слегка показать.

Ещё одним таким вариантом нецелевого использования является изменение цвета объекта (свойство Fill.Color). Если установить это значение равным цвету фона, то визуально объект также исчезнет:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.Fill.Color:= TAlphaColorRec.Lightgray;
  Rectangle2.Fill.Color:= TAlphaColorRec.Lightgray;
  // ... и так далее до Rectangle18
end;
```

В действительности приведенный код заставит прямоугольник слиться с фоном за счёт установки его свойства Fill.Color в значение Lightgray. На рис. 3.10 показано, как можно определить цвет формы, т.е. фона.

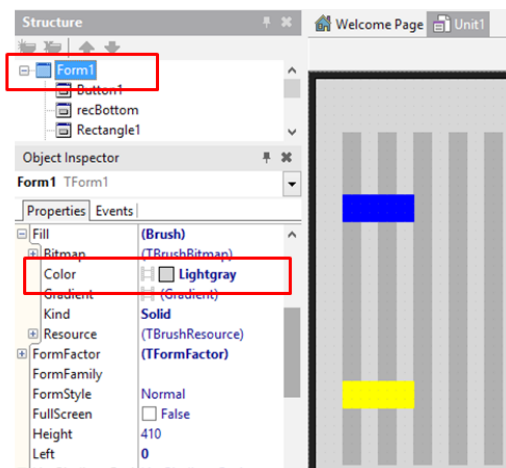


Рис. 3.10. Определение цвета формы в design-time

Рассмотренный код действительно скроет прямоугольник, изменив его цвет на серый. Но при изменении цвета формы, например, на зелёный, прямоугольник останется заметным. Можно попытаться исправить это, заменив явное указание цвета прямоугольника на непосредственно значение цвета формы:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.Fill.Color:= Form1.Fill.Color;
  Rectangle2.Fill.Color:= Form1.Fill.Color;
  // ... и так далее до Rectangle18
end;
```

Модифицированный код, также приведет к эффекту скрытия прямоугольников. Однако этот код содержит неявную ошибку, о чём будет сказано далее. Она связана с использованием Form1. Однако такой код хорошо иллюстрирует возможности по изменению цвета объекта, на основе которого мы добьемся эффекта плавного исчезновения прямоугольников.

Мы уже рассмотрели компонент TFloatAnimation, который реализовывал изменение вещественного значения во времени. Применительно к координате прямоугольника это вызвало его перемещение. Теперь рассмотрим компонент TColorAnimation, который теперь будет вызывать изменение цвета. Напоминаем, что цвет задаётся числом, поэтому переход от одного цвета к другому есть ничто иное, как изменение значения во времени. Визуально это будет выглядеть так, как будто прямоугольник плавно становится прозрачнее вплоть до полного исчезновения. Выполним ряд действий для первого прямоугольника-полоски:

- добавим с палитры на форму компонент TColorAnimation (раздел Animations);
- выделим его в панели Structure и перетащим на компонент Rectangle3 (как на рис. 3.8);
- настроим свойства, как показано на рис. 3.11;

Затем введем код для процедуры отклика на OnClick для Button1:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  ColorAnimation1.Enabled:= true;  
end;
```

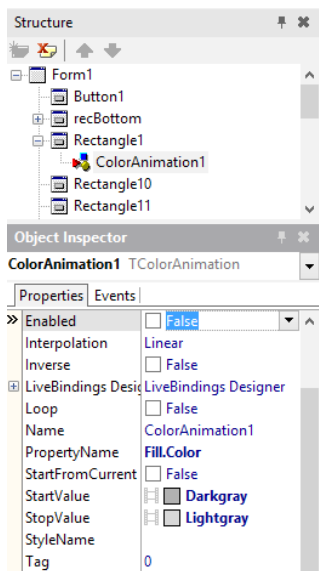


Рис. 3.11. Настройка анимации цвета

Сохраним и запустим проект. На интерфейсе приложения нажмем на кнопку Button1 и отметим плавное исчезновение прямоугольника. Если этого не произошло, то завершаем приложение (т.е. возвращаемся в IDE) и проверяем:

- начальное значение (StartValue);
- конечное значение (StopValue);
- PropertyName.

Анимацию можно запустить в автоматическом режиме, если в design-time установить свойство Enabled в true. Тогда анимация включится сразу после запуска приложения.

Если распространить данный метод для сокрытия каждого из оставшихся прямоугольников-полосок, то для каждого нужно добавить свой компонент TColorAnimation. Но есть и более эффективное решение без компонента TColorAnimation. Для этого удалим ColorAnimation1, выделив его в панели Structure и нажав кнопку Del. Перейдем к редактору кода и изменим код процедуры отклика на кнопку:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.AnimateColor('Fill.Color', TAlphaColorRec.
    LightGray, 1);
end;
```

Здесь первый параметр — это эквивалент PropertyName, далее идёт значение нового цвета, а последний параметр — период анимации в секундах, в течение которого который цвет плавно изменится в новое значение. Мы обошлись без отдельного компонента, сохранив анимацию цвета, т.е. плавное исчезновение. Этот подход также можно распространить на другие прямоугольники-полоски:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Rectangle1.AnimateColor('Fill.Color', TAlphaColorRec.
    LightGray, 1);
  Rectangle2.AnimateColor('Fill.Color', TAlphaColorRec.
    LightGray, 1);
  Rectangle3.AnimateColor('Fill.Color', TAlphaColorRec.
    LightGray, 1);
  // и т.д.
end;
```

Рассмотрим такую ситуацию, когда нужно будет удалить или добавить полоски. Тогда придётся дописывать код для скрытия новых прямоугольников. И в следующем разделе мы рассмотрим более эффективный способ групповой анимации.

3.4. Эффективный способ групповой анимации

У нас стоит задача — плавно скрыть множество прямоугольников наиболее эффективным способом. Воспользуемся компонентом TLayout. Этот компонент не имеет визуального изображения в runtime, а в design-time он также не заметен без выделения. TLayout — это просто область поверхности формы, контейнер, прозрачная панель, компонент без особых функций. Он выполняет важную группирующую функцию. Например, когда несколько компонентов, которые должны «всё делать одновременно». Это может быть совместное перемещение или одновременное изменение цвета и т.д. В переводе «layout» означает «разметка», что правильным образом описывает его назначение. Если речь идет о разметке интерфейса, то TLayout очень полезный компонент, т.к. различные смартфоны и планшеты могут иметь совершенно разные размеры экрана.

В этом разделе мы рассмотрим TLayout для решения задачи плавного и одновременного скрытия полосок с формы. Выделим все полоски, последовательно щелкнув на каждой из них мышью с нажатой кнопкой Shift (рис. 3.12, *а*). Все полоски выделены, что показано маленькими чёрными квадратиками по углам.

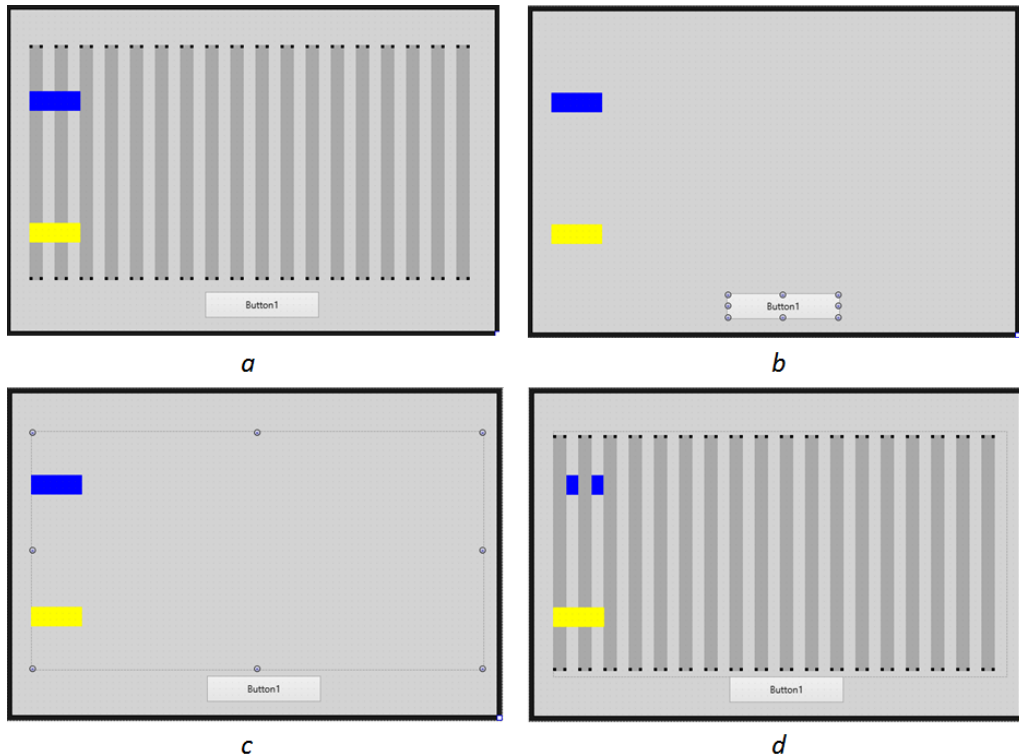


Рис. 3.12. Добавление полосок на TLayout

Теперь сделаем следующее действие с повышенным вниманием. Нажмем комбинацию клавиш Ctrl+X или выполним операцию «вырезать» из всплывающего или главного меню. Полоски исчезнут, но не бесследно. Они будут находиться в буфере обмена (рис. 3.12, *b*).

Теперь переместимся в палитру компонентов, найдем TLayout и разместим его на форме, как показано на рис. 3.12, *c*. А теперь при выделенном Layout1 нажмём Ctrl+V или выполним команду «вставить», полоски попадут из буфера обмена в наш компонент-контейнер, как показано на рис. 3.12, *d*. Может получиться так, что верхний (синий) или нижний (желтый) прямоугольники попадут под полоски. Чтобы выдвинуть их на передний план, выделим их поочередно в панели Structure и, нажав правую кнопку мыши, выберем команду BringToFront из всплывающего меню (рис. 3.13).

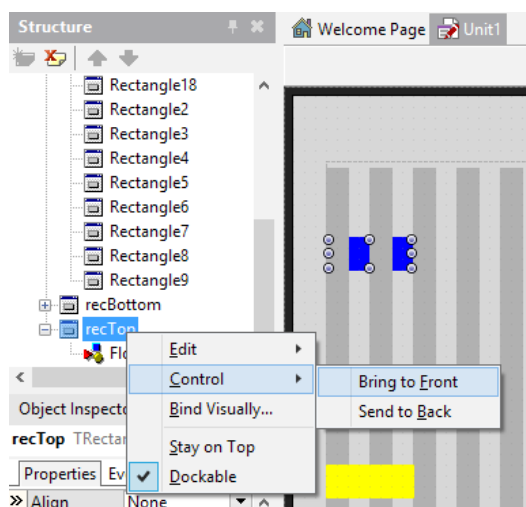


Рис. 3.13. *Перемещение компонента на передний план*

После таких нехитрых манипуляций у нас получается конфигурация компонентов, эквивалентная исходной, но теперь все прямоугольники будут расположены на контейнере TLayout. У всех компонентов TRectangle «отцовским» (Parent) компонентом является Layout1, в чём легко убедиться, обратившись к панели Structure (рис. 3.14).

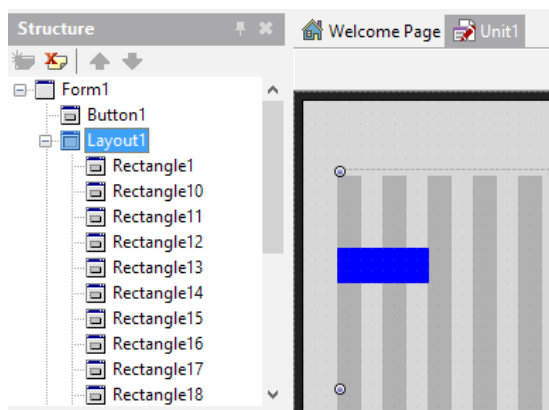


Рис. 3.14. *Взаимосвязь компонентов*

Если перемещать по форме компонент Layout1, то все расположенные на нём полоски также будут двигаться. Но это еще не все. Если мы будем изменять некоторые свойства Layout1, то это приведёт к автоматическому изменению всех идентичных свойств расположенных на нем компонентов. Перепишем процедуру отклика на кнопку Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Layout1.AnimateFloat('Opacity', 0, 1);
end;
```

Теперь наш контейнер Layout1 будет изменять свойство «прозрачность» не только самому себе, но и всем компонентам, расположенным на нём.

Мы потратили много времени, чтобы избавиться от однотипного кода типа:

```
Rectangle1.AnimateColor('Fill.Color', TAlphaColorRec.
  LightGray, 1);
Rectangle2.AnimateColor('Fill.Color', TAlphaColorRec.
  LightGray, 1);
Rectangle3.AnimateColor('Fill.Color', TAlphaColorRec.
  LightGray, 1);
// и т.д.
```

Если бы мы потом добавили еще пару прямоугольников-полосок (19 и 20), то пришлось бы дописывать код программы:

```
Rectangle19.AnimateColor('Fill.Color', TAlphaColorRec.
  LightGray, 1);
Rectangle20.AnimateColor('Fill.Color', TAlphaColorRec.
  LightGray, 1);
```

В варианте с `TLayout` в процедуре всего одна строчка, и её достаточно для «гашения» всех прямоугольники, сколько б их ни было на контейнере. Это — правильный вариант решения задачи, когда исходный код универсально работает на любом количестве добавленных на форму компонентов.

Мы решили задачу устранения однотипного кода в режиме `design-time`. Но это не избавило нас от необходимости добавлять полоски `TRectangle` в ручном режиме. Выходом из положения является динамическое создание компонентов, когда все они появляются на форме автоматически во время исполнения (`runtime`). Но для этого нужно использовать технику объектно-ориентированного программирования в полном объеме, чему будет посвящен один из следующих разделов.

3D-графика

4.1. Основы 3D-графики в Delphi/RAD Studio/C++Builder

Программирование 3D-графики всегда было крайне увлекательным занятием. Плоские картинki воспринимаются нами как схемы, чертежи, рисунки, диаграммы и т.д. В отличие от них создаваемые нами пространственные сцены имеют совсем другой уровень воздействия на наблюдателя. Умение создавать 3D-графику сродни какому-то волшебству, а если добавить еще и анимацию...

3D-графика создаётся при помощи разных технологий. Можно полностью программировать трёхмерные сцены, используя, специальные графические библиотеки и профессиональные среды разработки. Можно освоить графический пакет для 3D-моделирования и рисовать потрясающие пространственные картины. Есть и совсем сложные средства для создания 3D-фильмов, которыми пользуются мультипликаторы. Что из них выбрать? Для начинающего IT-специалиста лучше выбрать максимально простую технологию. Базовые принципы 3D-графики везде одинаковые, поэтому Delphi/RAD Studio/C++Builder является оптимальными средами.

Delphi/RAD Studio/C++ удачно сочетает базовые возможности графического пакета и среды программной разработки. С одной стороны, можно в визуальном режиме, как это мы делали с 2D-компонентами, моделировать 3D-сцены в design-time. С другой, манипулирование 3D-объектами можно производить на уровне программного кода. Конечно, визуальный редактор форм в Delphi по возможностям не может сравниться с профессиональными пакетами 3D-графики. Но и он позволяет создавать достаточно интересные пространственные сцены, динамизм которым придаётся при помощи программных средств.

Главным элементом приложения, использующего 3D-графику, является либо 3D-форма, либо 3D-viewport. Слово «viewport» является термином. Если форма сама по себе является 3D-формой, то любой пространственный объект может располагаться в любом её месте. Если на обычной плоской форме расположен прямоугольный «viewport», то он является 3D-частью 2D-формы. Пространственные объекты могут располагаться только в пределах его границ. Можно воспользоваться следующей аналогией: 2D-форма (обычная, плоская) является стеной, а 3D-viewport — окном в пространственный мир за стеной (рис. 4.1).

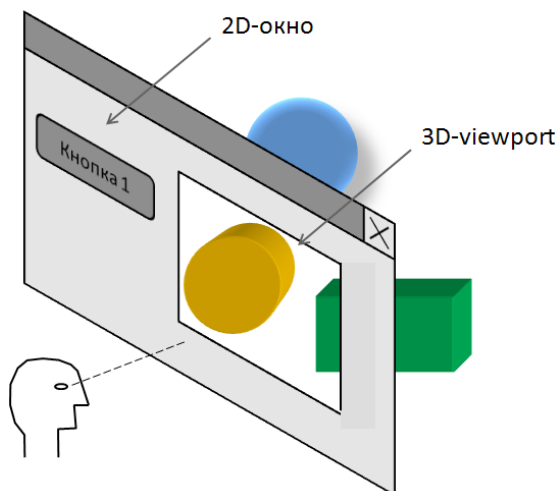


Рис. 4.1. Метафора окна в 3D-мир

Если мы планируем приложение, которое будет иметь только 3D-объекты в пространстве формы, то можно начать новый проект, где в качестве полностью трёхмерная форма является основной. Для этого нужно в мастере типа проекта выбрать 3D Application (рис. 4.2). Но мы воспользуемся схемой, когда создаётся обычное приложение Blank Application с плоской формой, а все пространственные объекты помещаем внутрь 3D-viewport-a.

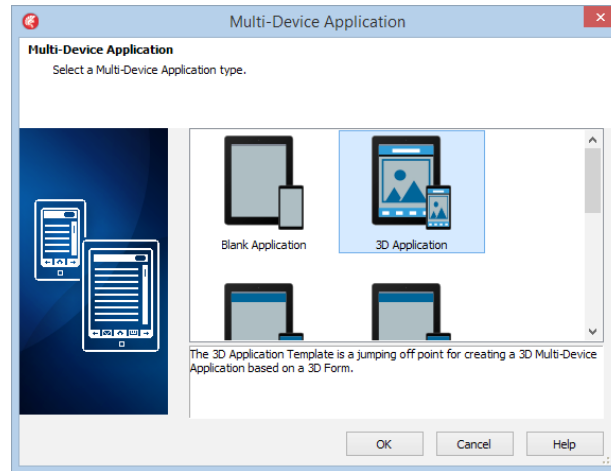


Рис. 4.2. Создание приложения с главной 3D-формой

Создадим приложение со «смешанным» типом интерфейса главной формы, когда обычные элементы управления будут сочетаться с трехмерными сценами. Это является более распространенным вариантом по сравнению с полностью 3D-приложением. Для нормального управления приложением, включая мобильное, нужны стандартные элементы типа «инструментальная панель», кнопки, метки, поля ввода и т.д. 3D-элементы будем размещать в специальном «видовом окне» — viewport-e.

Запустим среду Delphi и выберем в главном меню File->New->Multi-Device Application. После этого в мастере проектов кликнем на Blank Application (Рис. 4.3). Данный тип приложения не отличается принципиально от других типов, когда на главной форме уже есть некоторые элементы управления. Принципиальное отличие только в случае с 3D Application, т.к. там главная форма является полностью трёхмерной. Но мы выберем более универсальный путь, который позволяет смешивать 2D и 3D графику на одной форме.

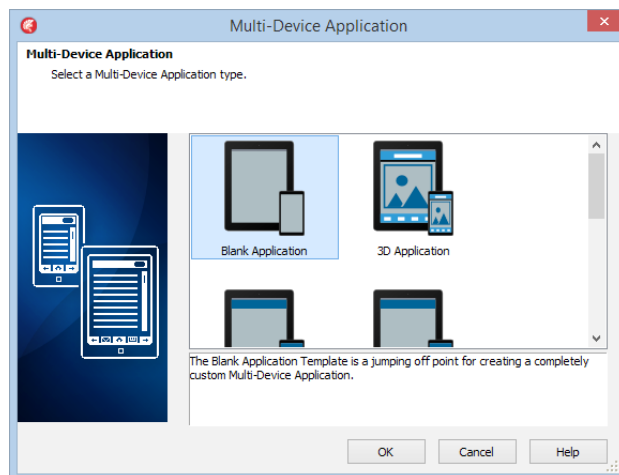


Рис. 4.3. Создание обычного приложения для 3D-графики

Перед нами появится обычная форма, на которую мы разместим компонент `TViewPort3D` из палитры компонентов в разделе Viewports. После размещения увеличим его размеры так, чтобы он занял практически всё пространство формы. Далее в разделе 3D Scene палитры компонентов найдём `TLight` и поместим его внутрь компонента `Viewport3D1` (рис. 4.4).

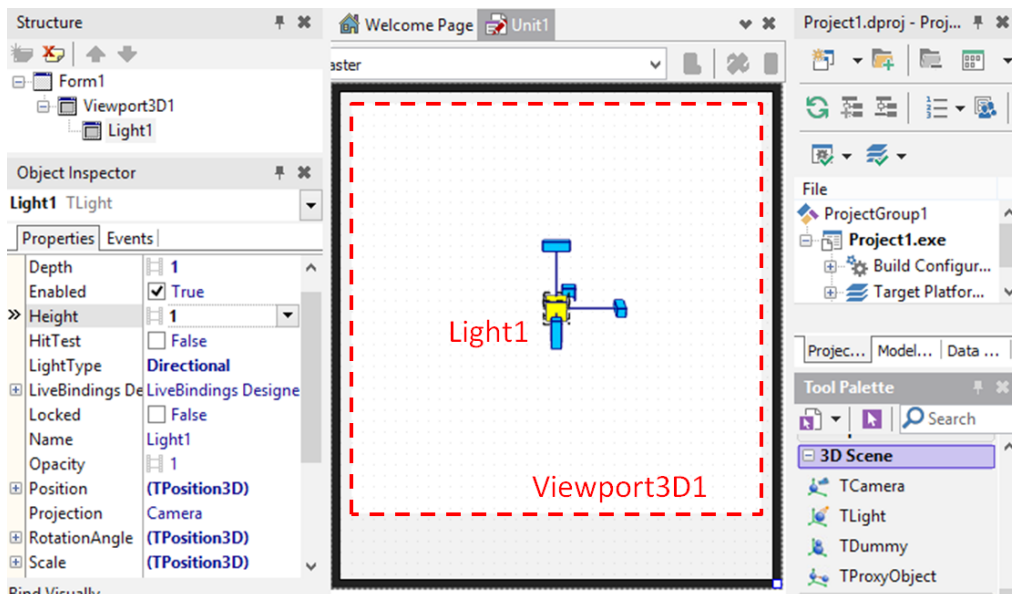


Рис. 4.4. Компонент `Light1` в `Viewport3D1`

Компонент TLight выполняет роль «источника света» для трёхмерной сцены. Если на TViewport компонент TLight не добавлен, то все 3D-объекты будут выглядеть неестественно красными и полностью лишёнными объёма. Таким образом, 3D-сцена будет сообщать, что не хватает «источника света» TLight.

Начинающим разработчикам не всегда бывает понятно, зачем нужен источник света в компьютерной графике. Источник света ассоциируется с настольной лампой или фонариком, которые не нужны в светлое время суток, чтобы увидеть объекты вокруг нас. Днём окружающие объекты освещены параллельными лучами солнечного света. В компьютерной графике мы будем использовать в качестве источника света TLight, дающего параллельные лучи, направление которых можно изменять. По умолчанию он размещается в центре TViewport3D и светит вглубь экрана.

Если выделить мышкой источник света, добавленный на ViewPort1, то можно увидеть небольшие синие «молоточки». Они называются «движками», которые можно задействовать мышью для вращения источника света вокруг одной из трёх пространственных осей (рис. 4.5). Данной возможностью следует пользоваться исключительно на начальном этапе, т.к. точность подобных манипуляций очень мала. Гораздо проще и точнее управлять положением, размерами и поворотом объектов при помощи свойств в Object Inspector.

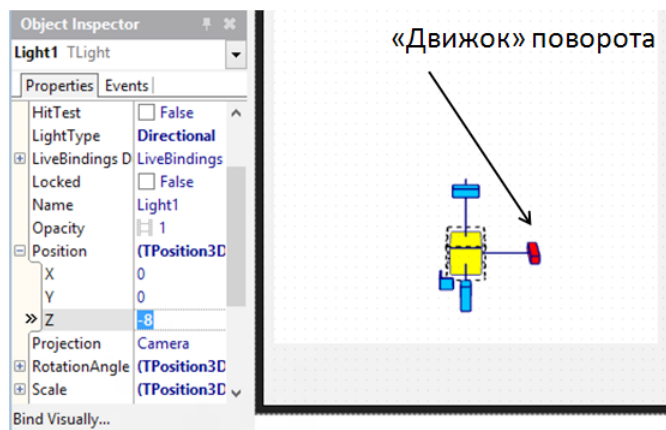


Рис. 4.5. Движок поворота источника света

Пространственная система координат представляется тремя осями: X , Y , Z . Начало системы располагается в центре `TViewport3D` (или 3D-формы). Ось X направлена из начала системы координат вправо, ось Y — вниз, ось Z — вглубь экрана (рис. 4.6).

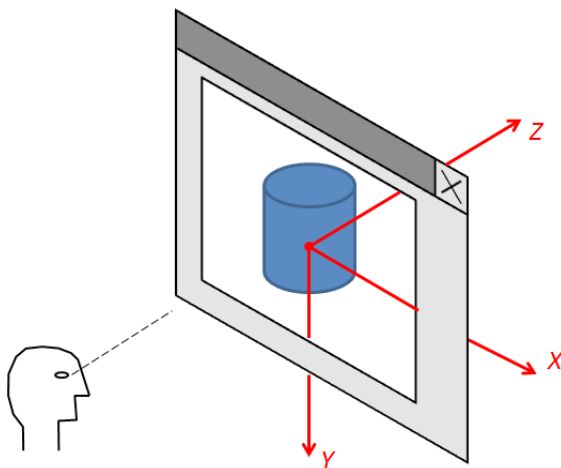


Рис. 4.6. Направление осей системы координат

Выше было сказано, что управлять объектами лучше при помощи свойств в Object Inspector. Посмотрим внимательно на рис. 4.5. При выбранном `Light1` отображаются вложенные свойства `Position`: X , Y , Z . Они задают координаты центра 3D-объекта. Выберем источник света и зададим его значение `Light1.Position.Z` равно -8 . Теперь источник света будет приближен к наблюдателю на 8 единиц (ось Z направлена вглубь экрана). Теперь `Light1` находится не в центре экрана, чтобы не заслонять собой видимые объекты. Объект-свет не виден непосредственно, его действие проявляется на других объектах. Если теперь запустить приложение, то перед нами будет пустое окно с белым квадратом viewport-a.

Добавим визуальные 3D-объекты. Первым из таких объектов будет сфера. Выберем `Viewport1` и добавим `TSphere` из раздела 3D Shapes палитры компонента (рис. 4.7). В данном разделе находится не только `TSphere`, но

и другие 3D-примитивы. Добавив сферу в 3D-сцену, обратим внимание на свойство Position. Оно состоит из вложенных свойств X, Y, Z, которые по умолчанию имеют значения 0, 0, 0, поэтому сфера появится в центре сцены. У неё, как и у источника света, показаны движки. Вращение сферы вокруг осей при помощи движков лишено смысла.

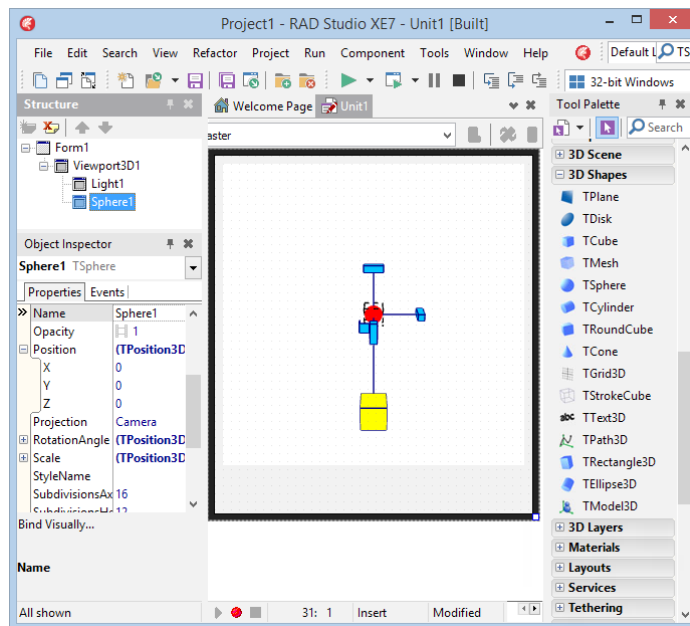


Рис. 4.7. Сфера в центре сцены

Помимо Position, основной является тройка свойств Depth (глубина), Height (высота), Width (ширина). Выберем их последовательно и зададим значения 10, 10, 10. Если задать неодинаковые значения, то сфера превратится в эллипсоид. Теперь у нас на форме будет большой красный плоский круг, вместо сферы. Для того, чтобы объект «сфера» выглядел привычным образом, ему нужно задать материал и осветить. Со освещением мы разобрались, добавив TLight. Теперь создадим материал.

Сам по себе любой 3D-объект является «бестелесным» в том плане, что у него нет материала. А раз нет материала, то нет и цвета. В простран-

ственной сцене материал представляется отдельным компонентом, причем в трёх вариантах. Мы возьмем самый сложный, создающий наиболее реалистичный эффект. Для этого добавим TLightMaterialSource из раздела Materials палитры компонентов.

У компонента LightMaterialSource1 задействуем свойство Diffuse, выбрав для него из списка значение Blue. После этого выберем сферу Sphere1, найдём в Object Inspector свойство MaterialSource и из выпадающего списка выберем LightMaterialSource1. Теперь наша картина должна выглядеть так, как показано на рис. 4.8. Когда 3D-объект с размерами является выбранным, то вокруг него отрисовывается «габаритная коробка» — прямоугольник, обозначающий границы объекта в его размерах.

Обратим внимание на взаиморасположение компонентов на панели Structure. По умолчанию LightMaterialSource1 попадает под узел Form1. Более правильным является его подчинительное расположение относительно Viewport3D1. «Уцепившись» мышью за LightMaterialSource1, перетаскиваем его на узел Viewport3D1. Теперь в случае необходимости мы можем скопировать всю сцену и перенести на другую панель, закладку, форму или даже в другой проект. При этом все необходимые компоненты, включая LightMaterialSource1, будут перенесены как одно целое.

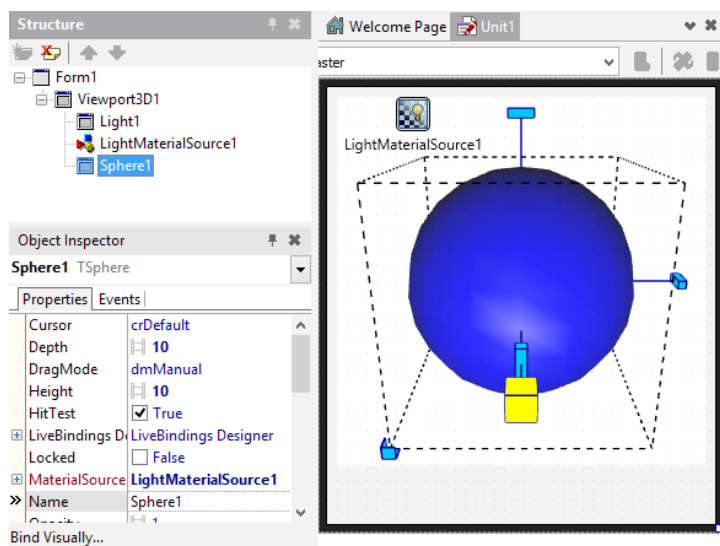


Рис. 4.8. Сфера с материалом и источником света

У сферы, как и у любого другого 3D-объекта, есть комплексное свойство Scale, состоящее из X, Y, Z. В переводе слово Scale означает «масштаб», его значение по умолчанию равно 1, 1, 1. Изменять масштаб размеров вдоль осей иногда бывает полезно, особенно когда сцена состоит из нескольких объектов и нужно изменить все сразу.

Если внимательно присмотреться к сфере на рис. 4.8, то можно заметить некоторую «угловатость» в её контуре. Происходит это потому, что вся поверхность сферы состоит из большого количества маленьких треугольников. Переход между ними достаточно плавный, поэтому «углы» заметны только на контуре, если размер объекта достаточно большой. Чтобы сделать сферу более гладкой, найдём в Object Inspector свойства SubdivisionAxes и SubdivisionHeight. Чем больше значения этих свойств, тем мельче будет размер треугольников, образующих поверхность сферы. Удвоим значения этих свойств. Теперь они будут 32 и 24, соответственно. Контур сферы стал весьма гладким. Но не стоит задавать неоправданно большие значения этих свойств. Чем больше треугольников в объекте, тем больше требуется времени для обработки. Сцена может начать «тормозить». Главное правило таково: используем стандартные значения. Если «угловатость» заметна, то увеличиваем их ровно настолько, чтобы получить визуально гладкую фигуру.

Выберем мышкой на форме или в панели Structure объект Sphere1 и добавим к нему еще две сферы — Sphere2 и Sphere3. Их не будет заметно, т.к. они попадают в центр сцены, т.е. внутрь большой синей Sphere1. Выберем их последовательно в Structure и зададим свойства, согласно таблице:

Свойства	Depth	Height	Width	Position		
Объекты				X	Y	Z
Sphere2	6	6	6	4	4	0
Sphere3	6	6	6	-4	4	0

Теперь малые сферы появятся по бокам большой. Добавим на форму еще один компонент TLightMaterialSource. Сразу перенесем его под узел Viewport3D1 в панели Structure. Свойство Diffuse выберем из списка как

Red. Последовательно или одновременно выделим Sphere1 и Sphere2, а затем в Object Inspector для свойства MaterialSource выберем из списка LightMaterialSource2. Мы получим пространственную модель молекулы водорода (рис. 4.9, *a*).

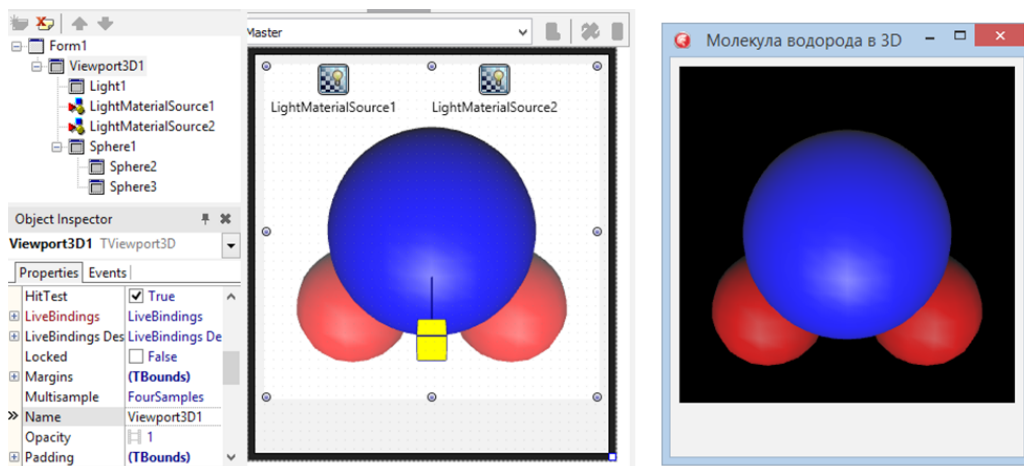
*a**b*

Рис. 4.9. Пространственная модель молекулы водорода:
a — в режиме design-time, *b* — в runtime и `Viewport3D1.Color = Black`

Сделаем рисунок более привлекательным. Выберем Sphere1 и изменим свойство Opacity на 0.8. Это добавит прозрачности не только синей сфере, но и двум красным. Теперь выберем ам TViewport1 и изменим его свойство Color на Black. Сохраним и запустим проект на исполнение. В результате у нас получится очень красивая 3D-молекула воды (рис. 4.9, *b*).

Обратите внимание, что Sphere2 и Sphere3 находятся в подчинительном положении относительно Sphere1 (панель Structure на рис. 4.9). Если мы будем что-либо изменять в свойствах Sphere1, то это отразится и на Sphere2 и Sphere3, что мы могли заметить по свойству Opacity. Если мы так-

же изменим масштаб Scale главного объекта, то и подчиненные объекты автоматически изменят свои размеры.

Реализуем вращение сцены в режиме design-time. Выделите Sphere1, уцепимся мышкой за «движок-молоточек», который направлен вправо вдоль оси *X*, и попытаемся подвигать им. При этом поворачиваться будет не только объект Sphere1, но и подчиненные Sphere2 и Sphere3. Как мы говорили раньше, «движки» хороши для экспериментов, но при разработке лучше использовать Object Inspector. Посмотрим на свойство Rotation, состоящее из X, Y, Z. Эта тройка свойств задает угол поворота в градусах вокруг осей координат, ранее показанных на рис. 4.6. В Object Inspector зададим значение вручную, например, RotationAngle.Y = 30. Молекула повернется вокруг вертикальной оси на 30 градусов, введенное явным образом (рис. 4.10).

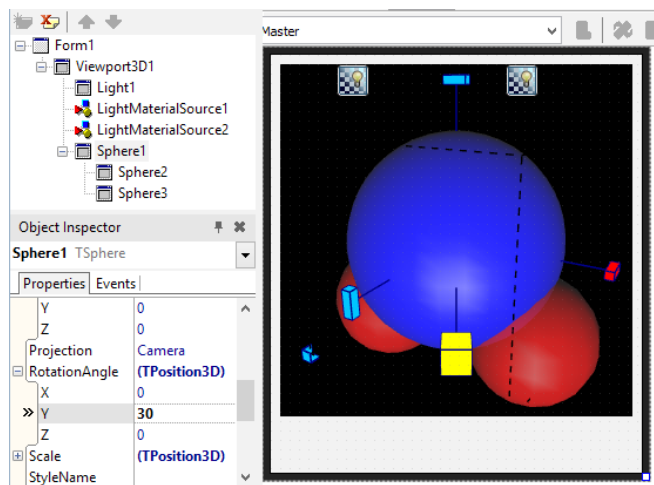


Рис. 4.10. Поворот сложного объекта

Завершим наш проект впечатляющим образом, реализовав автоматическое вращение молекулы. Выберем Sphere1 и добавим к ней компонент TFloatAnimation, с которым мы научились работать в предыдущей главе. Зададим свойства, согласно таблице:

Свойство	Значение	Объяснение
Duration	5	Длительность анимации, т.е. время, за которое молекула совершит полный оборот
Enabled	True	Анимация включена
Loop	True	Анимация будет зациклена, т.е. молекула будет вращаться, пока пользователь не остановит приложение
PropertyName	RotationAngle.Y	Свойство связанного компонента Sphere1, которое будет изменяться во времени (угол поворота вокруг вертикальной оси Y)
StartValue	0	Начальное значение угла поворота
StopValue	360	Конечное значение угла поворота (полный оборот)

Запустим проект на исполнение. Если мы всё сделали правильно, то молекула будет плавно вращаться.

В заключении проведём следующий эксперимент. В design-time перейдем на панель Structure, выделим узел FloatAnimation1, который вращал молекулу в runtime, и перетащим его на узел Light1. Теперь в runtime вращаться будет не молекула, а источник света (рис. 4.11). Для дальнейшего развития проекта вернём FloatAnimation1 в исходное положение, т.е. под Sphere1. Сохраним и запустим приложение, где как и прежде вращаться должна молекула.

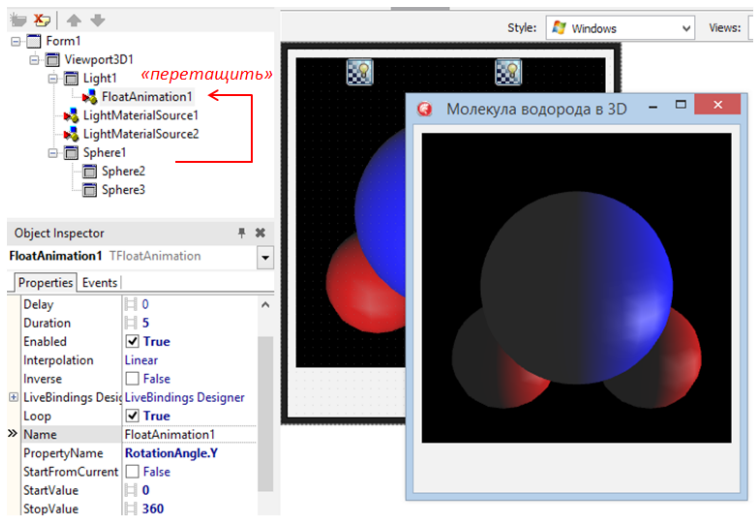


Рис. 4.11. Вращение источника света

При необходимости реализации сложной анимации можно добавить несколько компонентов `TFloatAnimation`. При воздействии с их помощью на значение `Sphere1.Position.Z` мы можем добиться эффектов приближения или удаления молекулы одновременно с вращением.

4.2. Управление объектам 3D-сцены

Приложения, связанные с использованием трёхмерной графики, весьма подходят для образовательных целей. Очень удобно изучать пространственные модели объектов при помощи приложений, поворачивая их различным образом, скрывая или показывая различные элементы. Мы уже знакомы с основными свойствами, определяющими размеры и положение 3D-объектов в виртуальном пространстве. Обобщим данную информацию в таблице:

Свойство	Объяснение
Position.X Position.Y Position.Z	Определяют смещение объекта относительно осей пространственной системы координат
RotationAngle.X RotationAngle.Y RotationAngle.Z	Определяют угол поворота объекта относительно осей пространственной системы координат
Depth Height Width	«Глубина» — размер объекта вдоль оси Z «Высота» — размер объекта вдоль оси Y «Ширина» — размер объекта вдоль оси X
Scale.X Scale.Y Scale.Z	Масштабный коэффициент для размеров объекта вдоль каждой из трёх пространственных осей. Если его значение 1, значит размер объекта соответствует Depth, Height и Width, соответственно.

Следует обратить внимание, что при добавлении фигуры её центр всегда попадает в начало системы координат. Следовательно вертикальный размер фигуры (свойство `Height` — «высота») как бы делится пополам в положительном и отрицательном направлении оси Y. Рис. 4.12 проясняет ситуацию. Таким же образом дела и с двумя другими габаритными свойствами — `Depth` и `Width`, (глубина и ширина).

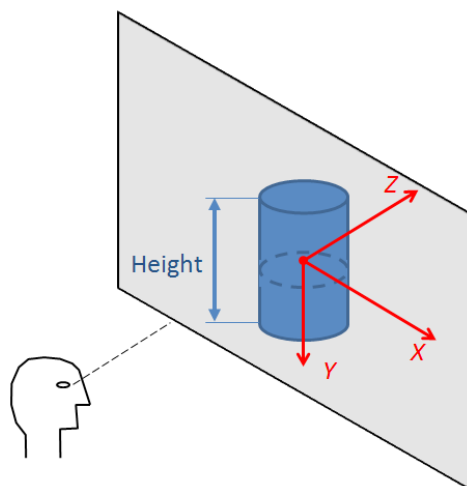


Рис. 4.12. Деление высоты пополам точкой начала координат

Рассмотрим свойство `Scale` (масштаб). По умолчанию оно для каждой из трёх осей задано как 1. Если мы изменим значение только `Scale.X` с 1 на 2, то объект вытянется вдоль оси `X` и будет выглядеть искаженным. Если нужно, чтобы объект сохранил пропорции, то изменять нужно все масштабы одновременно и пропорционально. Это сложное свойство `Scale` можно использовать для увеличения/уменьшения объекта и/или сцены.

Для реализации масштабирования молекулы добавим на форму две кнопки `TSpeedButton`. Перейдём в поле поиска палитры компонентов и начнём набирать первые символы слова `TSpeedButton` (Рис. 4.13). Срабатывает автоматический поиск, и мы увидим искомый компонент в палитре. Можно либо перетащить его на форму, либо просто щёлкнуть 2 раза мышкой на нём. При двойном щелчке кнопка появится на форме в произвольном месте. Нужно будет разместить её под формой. Также добавляется вторая кнопка.

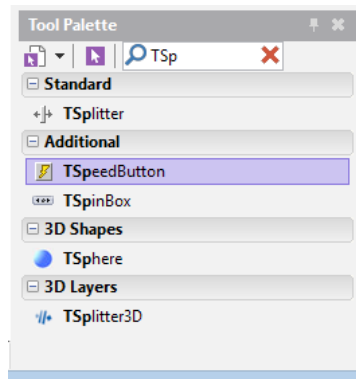


Рис. 4.13. Быстрый поиск компонента в палитре

После добавления «быстрые кнопки» (TSpeedButton) имеют дизайн по умолчанию, т.е. «никакой». Мы собираемся реализовать функцию увеличения/уменьшения объекта, поэтому нужно придать кнопкам соответствующий дизайн. Найдем у первой кнопки свойство StyleLookup и выберем из выпадающего списка значение «stepperbuttonleft», для второй — «stepperbuttonright». Расположим и сдвинем кнопки так, как показано на Рис. 4.14.

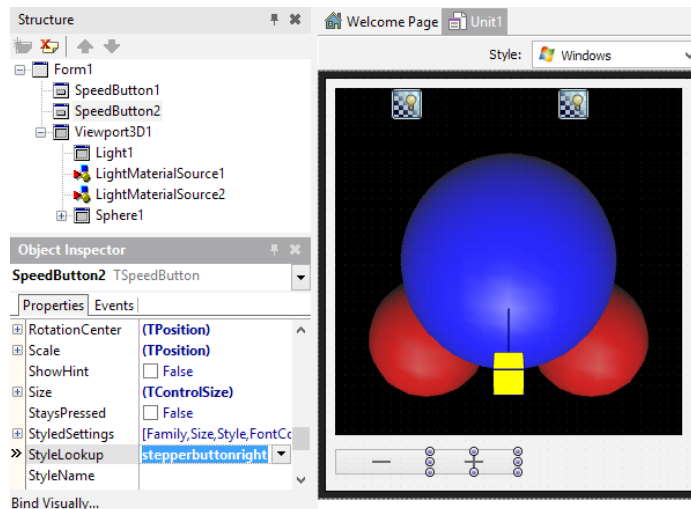


Рис. 4.14. Расположение кнопок уменьшения/увеличения объекта

Каждая кнопка нуждается в своей процедуре отклика. Есть два способа задания реакции приложения на действие пользователя. Первый из них, который мы собираемся реализовать, это непосредственно написание процедуры отклика на событие `OnClick`. Второй — связывание со специальным «действием» (Action). Пойдём по первому пути: выделим первую кнопку `SpeedButton1`, на которой изображён знак «минус», перейдём на `Object Inspector`, выберем закладку `Events` и дважды щёлкнем в пустом поле рядом с событием `OnClick`. Среда разработки автоматически сгенерирует программный код, в который мы впишем то, что будет увеличивать объект `Sphere1` и, соответственно, подчинённые объекты `Sphere2` и `Sphere3`:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);  
begin  
  Sphere1.Scale.X:= Sphere1.Scale.X * 0.9;  
  Sphere1.Scale.Y:= Sphere1.Scale.Y * 0.9;  
  Sphere1.Scale.Z:= Sphere1.Scale.Z * 0.9;  
end;
```

Практически то же самое сделаем с кнопкой `SpeedButton2`, на которой изображён знак «плюс»:

```
procedure TForm1.SpeedButton2Click(Sender: TObject);  
begin  
  Sphere1.Scale.X:= Sphere1.Scale.X * 1.1;  
  Sphere1.Scale.Y:= Sphere1.Scale.Y * 1.1;  
  Sphere1.Scale.Z:= Sphere1.Scale.Z * 1.1;  
end;
```

Еще раз подчеркнём, что мы изменяем масштаб лишь главного объекта, а подчинённые изменяются автоматически. Это удобно, когда нужно масштабировать все объекты в группе. Вспомним, что именно так мы и изменяли прозрачность (`Opacity`) сфер. Значение 0.8 было задано только для большой сферы `Sphere1`, но оно было автоматически распространено и на подчинённые сферы `Sphere2` и `Sphere3`.

Сохраним и запустим проект. После этого можно свободно нажимать кнопки «+» и «-», приближая и удаляя молекулу. Размер объекта можно изменять различными способами:

- изменять Depth, Height, Width;
- изменять Scale.X, Scale.Y, Scale.Z (значение 1 — базовый масштаб по умолчанию; значение 0.5 — меньше в два раза; значение 2 — больше в два раза);
- приближать/удалять объект «камера» (TCamera), если включена трансляция с неё.

Из перечисленных пунктов мы пока не владеем техникой использования «камеры», но оставим её рассмотрение до более подходящего случая. Камерой мы будем пользоваться, когда нужно будет осуществлять не масштабирование, а навигацию по пространственной сцене, состоящей из множества объектов. Задание свойств Width, Height, Depth обычно происходит изначально и в design-time. Это позволяет определить величину объекта относительно других в сцене. Для масштабирования объекта в runtime эффективно использовать свойство Scale, не забывая делать это одновременно для Scale.X, Scale.Y, Scale.Z.

К настоящему времени мы привыкли, что есть главный объект, а есть — подчиненные. Такого рода отношения между объектами задаются в панели Structure (рис. 4.15). Sphere1 — главный объект по отношению к Sphere2 и Sphere3, а они — подчинённые. Если мы будем перемещать (изменять Position) или масштабировать (изменять Scale) главный объект, то это автоматически скажется на подчиненных.

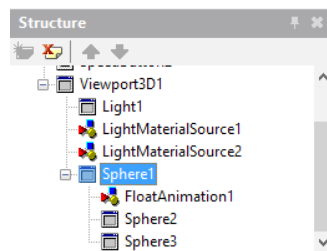


Рис. 4.15. Взаимоотношение объектов «главный-подчиненный»

В нашем случае главный объект — Sphere1 — находится в центре сцены, в начале системы координат (0, 0, 0). Такой случай является типичным для сцены с одним сложным объектом. Следующий раздел посвятим проекту со множеством таких объектов.

4.3. Создание сложных сцен

Начнём новый проект, выполнив File->New->Multi-Device Application — Delphi, затем Blank Application. Сохраним проект в специально созданную для этого папку. Поскольку сейчас мы будем делать достаточно сложное приложение в интерфейсной части, займёмся разметкой. Основой любой разметки, будь то приложение для настольных компьютеров или мобильных устройств, является компонент TLayout («разметка»), который доступен в Delphi/C++Builder/RAD Studio. Можно аккуратно разместить на форме различные компоненты: кнопки, поля ввода, «видовое окно» (TViewport), компоненты «многострочный текст» (TMemo), поля ввода (TEdit). Но если затем изменить размер окна в режиме runtime для настольных приложений или же запустить на мобильном устройстве, размер экрана которого будет отличаться от исходного, то мы увидим, что «разметка съехала» (рис. 4.16). Отсутствие разметки при размещении компонентов не выдерживает изменение размера формы. Поэтому всегда дизайн интерфейса приложения, если это не совсем уж учебный пример, начинается с разметки. Правильная разметка гарантирует сохранение порядка взаиморасположения компонентов.

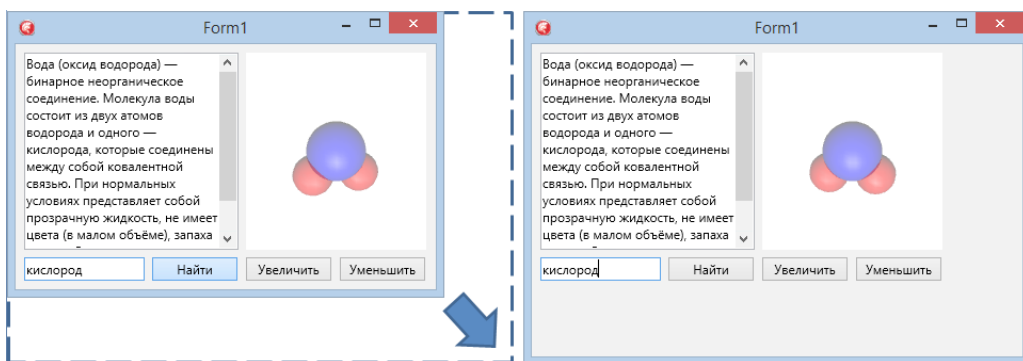


Рис. 4.16. Неправильная разметка интерфейса

Расположим на форме 3 компонента TLayout (рис 4.17, а). Они сами по себе не заметны в режиме runtime. В design-time они представлены пунктирными прямоугольниками, что не мешает обычным визуальным компонентам.

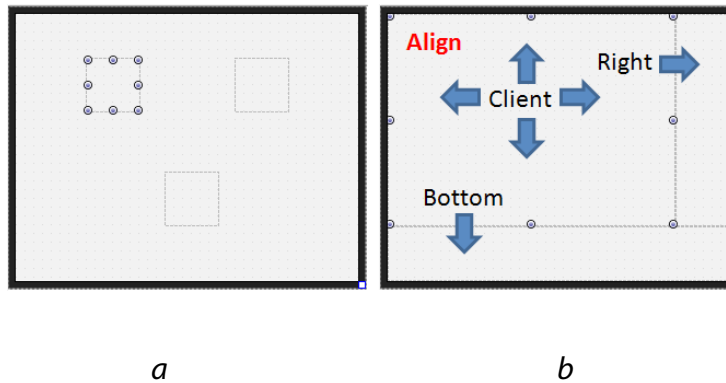


Рис. 4.17. Три компонента TLayout на форме

Теперь будем поочередно выбирать компоненты TLayout на форме и задавать им свойства Align (произносится как «э-лайн» с ударением на второй слог; в переводе — «настраивать»). Первому выбранному TLayout установим свойство Align в значение Bottom («низ»), воспользовавшись выпадающим списком (рис. 4.17, b). Теперь компонент TLayout всегда будет «приклеен» к нижней части формы и «размазан» по ней. Он всегда будет сохранять свою высоту, а ширина будет зависеть от ширины формы. Выберем второй TLayout, и его свойство Align установим как Right. Соответственно, данный компонент будет «прижат» к правой границе окна. Третьему компоненту TLayout установим свойство Align в Client. Он «растечётся» по оставшейся части формы, заполняя собой всё свободное пространство. В дальнейшем вы увидите, что все остальные визуальные компоненты формы будут строго следовать разметке, заданной тремя данными компонентами.

На центральный компонент TLayout поместим компонент TViewport3D. Он будет создавать нам «окно в трёхмерный мир». Его свойство Align в зна-

чение Client. Теперь и он «растечётся» по всей поверхности компонента TLayout. Будем создавать сложную пространственную схему, в которой не будет главного объекта (Sphere1 в предыдущем случае). Внутри TViewport3D добавим компонент TDummy (dummy — «пустышка»), как показано на рис. 4.17. Данный компонент обладает всеми свойствами пространственного объекта за исключением одного — он «невидимка» (runtime его заметно не будет). Нужен он лишь для того, чтобы быть главным для группы объектов.

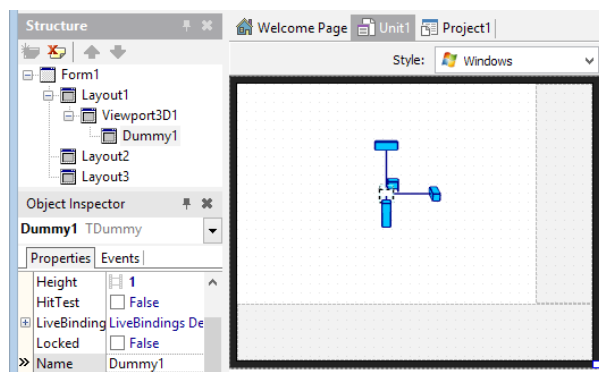


Рис. 4.17. Компонент-невидимка TDummy

Если теперь мы будем добавлять объекты на сцену, делая их подчиненными компоненту TDummy, то любое воздействие на него (поворот, масштабирование, изменение пространственных координат) будет влиять на всю сцену.

Приступим к созданию приложения, иллюстрирующее молекулы основных химических веществ, из которых состоит воздух. Модели молекул будут представлены в 3D-графике. Пользователь сможет их просматривать, используя вращение и анимацию.

На форме, представленную на рис. 4.17, добавим 4 компонента TLightMaterialSource. Важно правильно задать имена (Name) компонентов, чтобы

затем не путаться при их использовании. Значение свойств Name и Diffuse для данных компонентов представлены в таблице:

Компонент/свойство	Значение по-умолчанию	Новое значение
LightMaterialSource1.Name	LightMaterialSource1	lmsBlue
LightMaterialSource1.Diffuse	White	Blue
LightMaterialSource2.Name	LightMaterialSource2	lmsRed
LightMaterialSource2.Diffuse	White	Red
LightMaterialSource3.Name	LightMaterialSource3	lmsGreen
LightMaterialSource3.Diffuse	White	Green
LightMaterialSource4.Name	LightMaterialSource4	lmsGray
LightMaterialSource4.Diffuse	White	Gray

Новые значения имён компонентов теперь состоят из первых трёх букв lms (LightMaterialSource), к которым затем добавляется слово с большой буквы, обозначающее цвет. Выберем Dummy1 и к нему добавим еще один TDummy из палитры компонентов. Изменим его свойство Name в “duOxygen”. Выделим данный компонент.

С палитры компонентов добавим к нему два компонента TSphere, а их свойства установим в Object Inspector согласно таблице:

Компонент/свойство	Новое значение	Объяснение
Sphere1.Name	spOxygen1	Первый атом молекулы кислорода
Sphere1.Depth, Sphere1.Height, Sphere1.Width	4, 4, 4	Габаритные размеры одинаковы и равны 4
Sphere1.MaterialSource	lmsBlue (из выпадающего списка)	Сфера будет связана с компонентом-материалом, дающим синий цвет
Sphere1.Position.X	-1	Сфера будет смещена вдоль оси X влево
Sphere2.Name	spOxygen2	Второй атом молекулы кислорода
Sphere2.Depth, Sphere2.Height, Sphere2.Width	4, 4, 4	Габаритные размеры одинаковы и равны 4
Sphere2.MaterialSource	lmsBlue (из выпадающего списка)	Сфера будет связана с компонентом-материалом, дающим синий цвет
Sphere1.Position.X	1	Сфера будет смещена вдоль оси X вправо

Окончательно форма должна выглядеть так, как показано на рис. 4.19. На панели Structure чётко прослеживается иерархия объектов, т.е. их взаимоотношение типа «главный-подчиненный». В основе всего находится форма Form1. На ней расположен Layout1, всю поверхность которого занимает Viewport3D1. В центре «просмотрового окна» расположен главный «пустой объект сцены» Dummy1, который является главным для duOxygen. Поскольку молекула кислорода будет представлять собой группу объектов, точнее, две сферы, то главным будет специальный «пустой» объект duOxygen. В его подчинении находятся две сферы spOxygen1 и spOxygen2. Также на рисунке стрелками показано соответствие объектов в Structure и на форме.

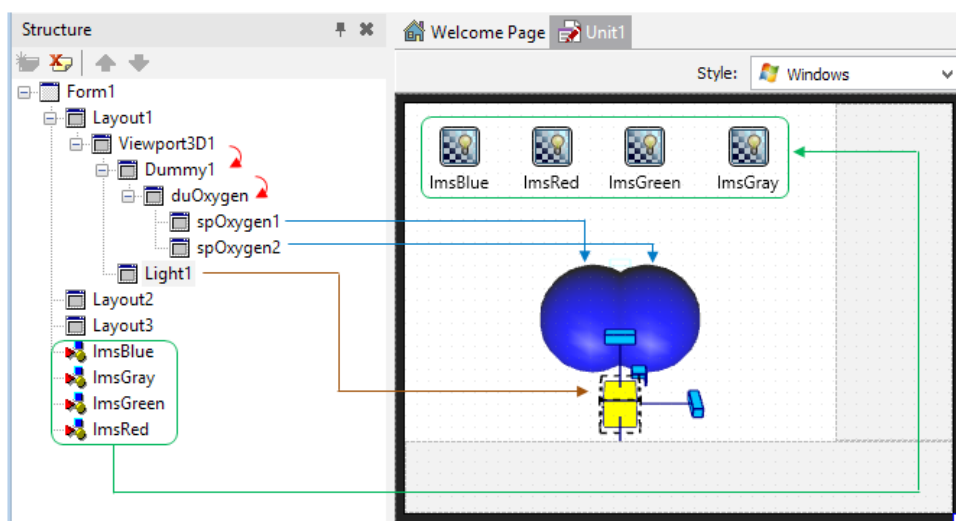


Рис. 4.19. Молекула кислорода из двух атомов — сфер

Пока в сцене присутствуют два объекта «пустышки» (TDummy). Первый — Dummy1 — является самым главным в сцене. Второй — duOxygen — главный в молекуле кислорода. Пока duOxygen находится в центре, совпадая с Dummy1. Выберем duOxygen и изменим в Object Inspector его свойство Position.Z, задав ему значение -5 . Особых изменений в сцене мы не увидим, т.к. молекула выдвинулась на зрителя, т.к. ось Z устремлена вглубь экрана от наблюдателя из центра системы координат.

Обеспечим возможность поворота сцены для пользователя программы в режиме runtime. В этот раз обойдёмся без анимации, будем вращать сцену вручную. Добавим на нижний TLayout компонент TTrackBar («движок» или «ползунок»). Зададим ему свойство Min равным 0, а свойство Max равным 360. При перемещении движка компонента значение Value будет изменяться от 0 до 360 в зависимости от его положения (рис. 4.20).

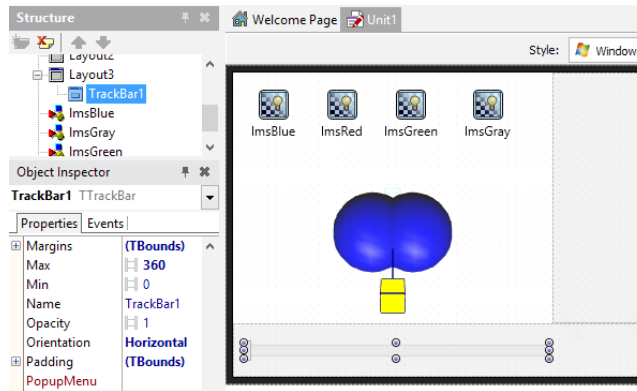


Рис. 4.20. Компонент TTrackBar на форме

Всякий раз, когда пользователь перемещает движок компонента, должно происходить вращение сцены. Перейдем на закладку Events панели Object Inspector, найдём событие OnChange и щелкнем два раза в пустом поле. Среда разработки сгенерирует шаблон процедуры отклика, который мы дополним собственным кодом:

```
procedure TForm1.TrackBar1Change(Sender: TObject);  
begin  
  Dummy1.RotationAngle.Y:= TrackBar1.Value;  
end;
```

Смысл введённой строчки очевиден — при каждом изменении положения движка новое значение Value будет присваиваться свойству «угол поворота вокруг вертикальной оси» главному Dummy1. Сцена будет вра-

щаться. Сохраним проект, запустим приложение на исполнение и проконтролируем результат.

Продолжим создание небольшой коллекции моделей молекул, из которых в основном состоит воздух. Для этого выберем объект `duOxygen`, сместим его вправо, задав `duOxygen.X = 5`. Далее переместим его обратно в плоскость экрана `duOxygen.Z = 0` и повернём вокруг вертикальной оси на 90 градусов, задав `duOxygen.RotationAngle.Y = 90`. Все эти манипуляции совершаем в Object Inspector. Тем самым мы убрали молекулу кислорода с переднего плана и из центра сцены, чтобы создавать очередную молекулу — молекулу воды (рис. 4.21), которую мы хорошо разобрали в предыдущем примере.

Начнём с добавления к сцене нового объекта `TDummy` при выделенном `Dummy1`. Тогда новый компонент, которому зададим имя `duWater`, станет подчинённым `Dummy1`. Если случайно новый компонент не стал подчинённым `Dummy1`, то нужно найти его и перетащить его на узел `Dummy1` в панели Structure. Затем добавляем одну сферу для обозначения атома кислорода и две сферы для атомов водорода. Назначаем им правильные `MaterialSource`, чтобы сферы обрели нужный цвет (Рис. 4.21).

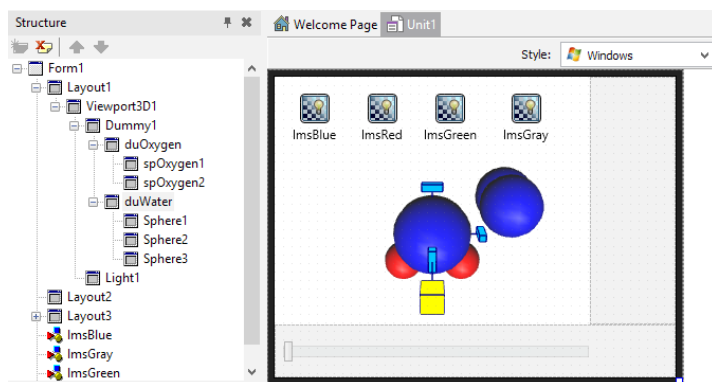


Рис. 4.21. Добавленная молекула воды

Наметим дальнейшие шаги: добавим молекулы углекислого газа и молекула азота. Необходимо добавлять TDummy для каждой новой молекулы в качестве главного объекта. По окончании работы с конкретной молекулой будем убирать её с переднего плана за счёт манипуляции со значениями свойств Position.X, Position.Y, Position.Z и RotationAngle.Y. В конце изменим свойство Color объекта ViewPort1 на Black, чтобы затенение объектов было адекватным фону. Запустим приложение. При перемещении движка компонента TrackBar1 будет наблюдаться эффект «карусели» (рис. 4.22).

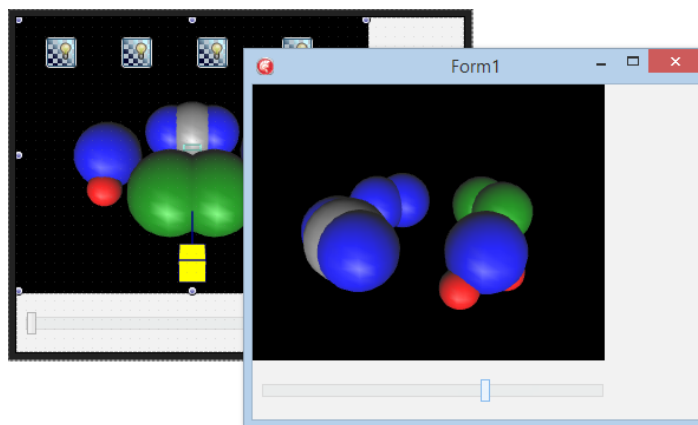


Рис. 4.22. Дизайн формы
(на заднем плане) и запущенное приложение «карусель молекул»

В начале данного проекта мы достаточно времени уделили первичной разметке формы, применив компоненты TLayout. В эффективности данного подхода легко убедиться, если запустить проект и «развернуть» окно или просто увеличить его размеры за счёт «перетаскивания» границы. При этом «просмотровое окно» TViewport3D также увеличивается. Однако «движок» TrackBar1 не увеличивается в размерах — не растягивается, но это легко исправить.

Каждый 2D-компонент или обычный плоский элемент управления имеет следующие геометрические свойства: ширина Width, высота Height, Position.X и Position.Y — координаты левого верхнего угла габаритного пря-

моугольника компонента. Если мы будем увеличивать форму или TLayout, на котором находится компонент, то он сам изменяться не будет, сохраняя свои Width и Height. Но если нам нужно, чтобы компонент «растягивался» вместе с увеличением размеров окна, тогда нужно применить «якоря». В Object Inspector найдём составное свойство Anchors («якоря»), раскроем его и поставим True рядом с akRight и akBottom. Теперь True стоит во всех строчках узла Anchors. Если мы запустим приложение и начнем изменять размеры окна, то TrackBar1 также будет изменяться, что нам и нужно было достичь. Рис. 4.23 подробно поясняет смысл значений составного свойства Anchors.

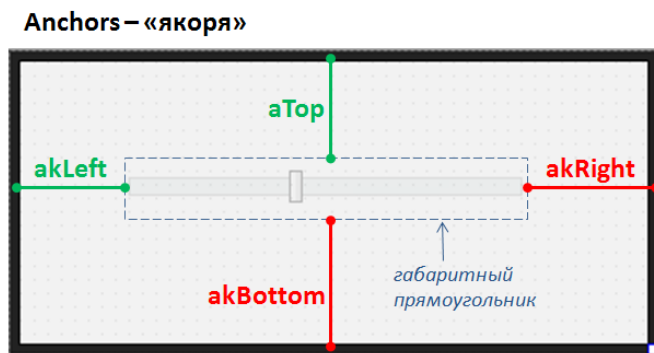


Рис. 4.23. Свойства Anchors для настройки компонента

При увеличении размера формы у пространственных объектов начинает проявляться некоторая «угловатость» контура, что было отмечено ранее (рис. 4.24, *a*). Пространственный объект «сфера» состоит из множества маленьких треугольников, как поверхность зеркального шара для диско-тек состоит из множества маленьких плоских квадратики (рис. 4.24, *b*). Чем меньше размер этих элементов, тем «круглее» кажется шар и тем менее заметны углы этих треугольников (в TSphere) или квадратики (в дискотечном шаре).

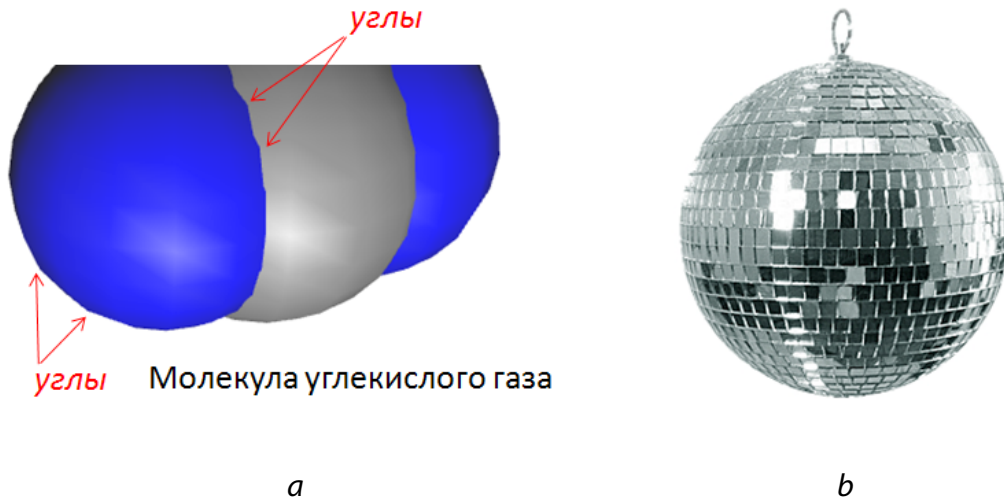


Рис. 4.24. Угловатость пространственной сферы *TSphere* и зеркального шара

Программист может варьировать размером этих треугольных элементов. В случае с *TSphere* мы в *Object Inspector* находим свойства *SubdivisionAxes* и *SubdivisionsHeight*, затем увеличиваем их значения, допустим, в два раза. Каждый раз для конкретного масштаба изображения данные значения подбираются визуально.

Доведём наше приложение до законченного вида. Для этого добавим ещё один компонент *TTrackBar* на правый *TLayout*, как показано на рис. 4.25. Чтобы он стал вертикальным, нужно изменить значения свойства *Orientation* с *Horizontal* на *Vertical*. После этого при помощи мышки зададим его размеры на правой панели *TLayout*. Чтобы новый вертикальный компонент *TTrackBar* адекватно реагировал на изменение размеров формы, изменим значения «якорей», как это было с предыдущим *TTrackBar* и как показано на рис. 4.25.

Изменим значение *Max* на 360, т.к. это будет максимальным углом поворота сцены вокруг горизонтальной оси. Процедуру отклика на событие *OnChange* для данного компонента напомним аналогично компоненту *TrackBar1* способом. Не забудем изменить в коде название компонента на

SeekBar2, а также ось вращения сцены на `Dummy1.RotationAngle.X`. Окончательно завершить приложение вы можете самостоятельно, добавив в правом нижнем углу кнопки «увеличить/уменьшить» с соответствующими процедурами отклика, описанными выше.

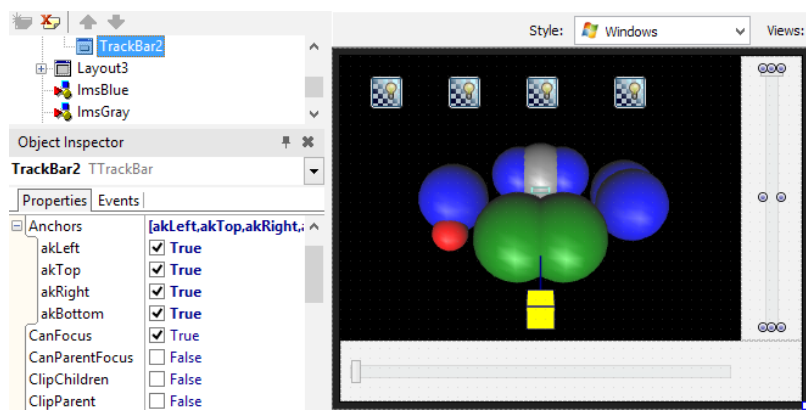


Рис. 4.25. Настройка вертикального компонента *TSeekBar*

В рассмотренном проекте мы оперировали тремя компонентами `TLayout`. Основным свойством, определяющим их расположение на форме, является `Align`. В зависимости от значения данного свойства компонент `TLayout` как контейнер для размещения других элементов управления может автоматически «приклеиваться» к правой или левой части интерфейса, вести себя как «вода на дне ёмкости» или подобно газу заполнять всё предоставленное пространство. Рассмотренный нами пример достаточно нагляден: внизу и слева расположены компоненты `TLayout` для размещения элементов управления. Они имеют свойство `Align` равным `Bottom` («низ») и `Right` («право»), соответственно. В таких случаях обычно есть и третий компонент `TLayout`, который имеет свойство `Align` равным `Client`. Он всегда заполняет оставшееся пространство, т.н. «клиентскую часть формы» (отсюда и название — `Client`, «клиент»).

После того, как мы сделали разметку (рис. 4.17), может потребоваться изменение размеров компонентов `TLayout`. Например, нам необходимо изменить ширину правого `TLayout`. Это можно сделать, как всегда, при

помощи Object Inspector и задать свойство Width с клавиатуры в численном виде. Гораздо быстрее это делать мышью. Типовая ошибка — попытка сделать это за счёт центрального TLayout, тогда как данные манипуляции нужно производить, выделив правый компонент TLayout, как показано на рис. 4.26.

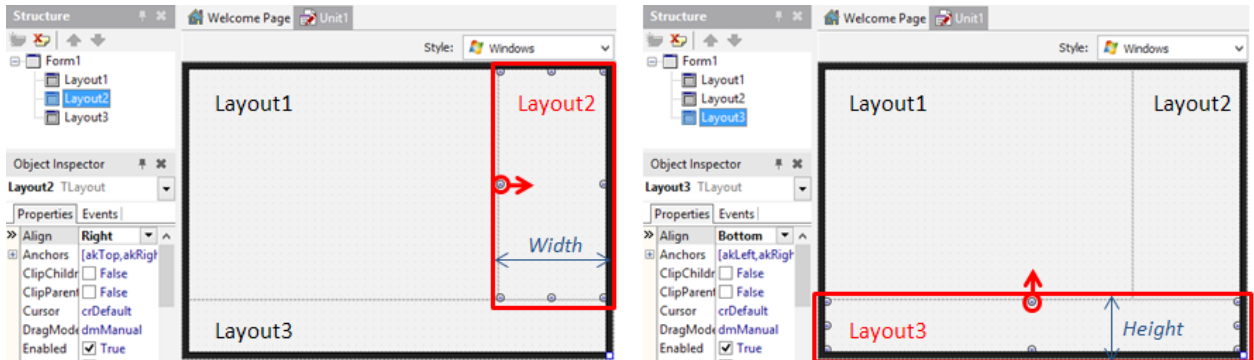


Рис. 4.26. Управление разметкой формы при помощи TLayout

При работе с разметкой формы при помощи компонентов TLayout и различных вариантов свойства Align следует пользоваться следующими правилами:

- мысленно разбить окно на зоны для близких по смыслу компонентов;
- в соответствии с зонами разместить на форме компоненты TLayout;
- в дальнейшем размещать визуальные компоненты группами на соответствующих TLayout;
- если сразу не удалось воспользоваться компонентами TLayout, то попытаться переделать интерфейс, используя «разметку»
- можно создать вторую форму в проекте с правильной разметкой, а затем перенести компоненты через буфер обмена со старой формы на новую;
- перенести процедуры отклика на события из модуля старой формы в модуль новой;
- сделать вторую форму главной, протестировать проект, а затем удалить первую «черновую» форму из проекта.

4.4. Интерактивные 3D-сцены

Пока мы реализовали трёхмерную сцену, состоящую из множества различных объектов, которыми можно управлять: поворачивать, сдвигать, масштабировать. Также мы усвоили, что для более удобной манипуляции с объектами сцены нужно применять компонент-«пустышку» `TDummy`. Воздействие как на объекты, так и на всю сцену мы выполняли при помощи обычных элементов управления — визуальных компонентов: «движков» и кнопок. Мы подбирали необходимые события — `OnChange` или `OnClick` — и реализовывали процедуры отклика в виде программного кода. Но наша 3D-сцепна не была интерактивной, т.е. не могла реагировала на непосредственные действия пользователя. Хотя сферы (и другие 3D-формы типа цилиндра, конуса, диска, куба и т.д.) непохожи на кнопки, они также могут воспринимать действия пользователя, и для них также можно создавать процедуры отклика на события.

А теперь мы создадим интерактивную сцену с обработкой метода копирования сцен и компонентов, описанного в конце раздела 4.3. Откроем наш последний проект и добавим к нему новую форму: `File->New->Multi-Device Form` — `Delphi`. Затем выберем `HD Form`, т.е. обычную «плоскую» двумерную форму. К сожалению, увидеть две формы одновременно в `Delphi/C++Builder/RAD Studio` нельзя, но зато можно быстро переключаться с одной на другую. На рис. 4.27 показана последовательность переключения между `Form1` и `Form2`. Также можно воспользоваться комбинацией клавиш `Shift + F12`, чтобы вызывать диалог переключения между формами.

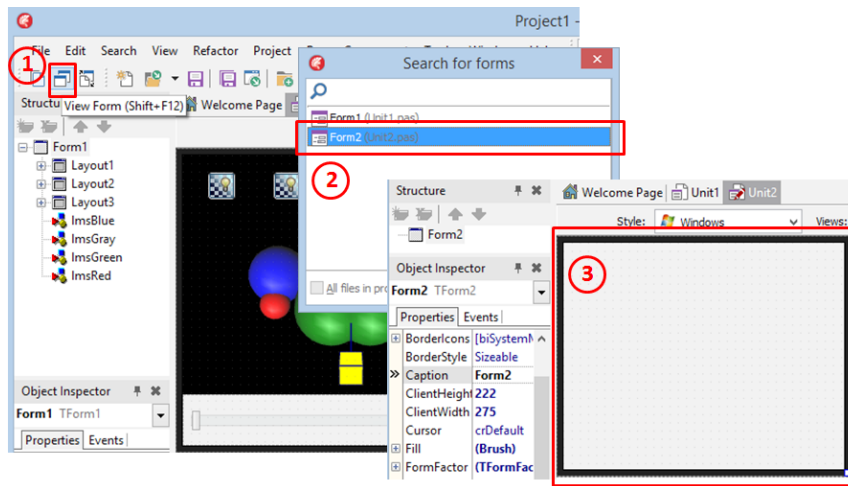


Рис. 4.27. Переключение между формами в проекте

Как только мы создали Form2, нужно сохранить весь проект. По умолчанию перед нами будет пустая новая форма. Можно сделать её дизайн «с нуля», но мы скопируем 3D-сцену с предыдущей формы.

Подготовим вторую форму. Добавим на неё компонент TLayout и установим ему свойство Align в значение Right. Добавим ещё один TLayout, а ему свойство Align установим в Client. Теперь переключимся на первую форму Form1, нажмём клавишу Shift и, удерживая её, выделим последовательно компоненты TLightMaterialSource: ImsBlue, ImsGray, ImsGreen и ImsRed. Скопируем в буфер обмена, нажав Ctrl+C или выбрав пункт меню Edit->Copy. Переключимся на вторую форму Form2, выделим тот TLayout, который находится в левой центральной части со свойством Align=Client, и выполним вставку из буфера обмена комбинацией Ctrl+V или пунктом меню Edit->Paste. Опять переключимся на Form1, выделим и скопируем весь Viewport1. Потом вернёмся на Form2 и вставим Viewport1 из буфера, предварительно выделив левый центральный TLayout. Результат должен соответствовать рис. 4.28.

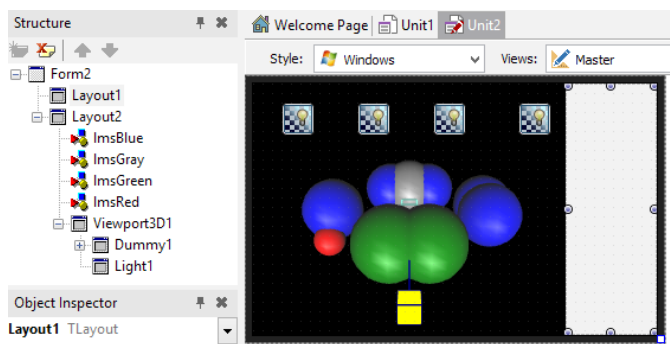


Рис. 4.28. Скопированная сцена на второй форме

Изменим свойство Color компонента Viewport1 на значение Gray, чтобы иметь явное визуальное отличие от первой формы. Разместим молекулы физических веществ, входящих в воздух, несколько иным способом. Мы не будем поворачивать сцену при помощи движков, поэтому все объекты будут находиться в одной плоскости — плоскости экрана. Последовательно выберем компоненты TDummy для каждой молекулы и изменим значения пространственных координат, согласно таблице:

Элемент	TDummy	Position		
		X	Y	Z
Углекислый газ	duCarboxide	-3	-3	0
Азот	duNitrogen	3	3	0
Кислород	duOxygen	3	-3	0
Вода	duWater	-3	3	0

Выделим правый TLayout и разместим на него TМемо из палитры компонентов. Изменим значение свойства Align на значение Client, понимая, что TМемо будет полностью занимать всю поверхность «родительского» TLayout. Полученный результат проконтролируем по рис. 4.29. У компонента TМемо найдём в Object Inspector свойство TextSettings.WordWrap и измените его значение на True. Теперь текст в данном компоненте будет переноситься, если его размер будет превышать одну строку.

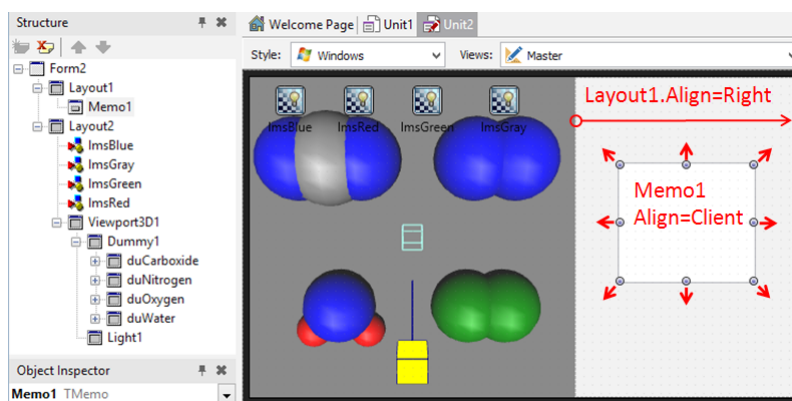


Рис. 4.29. Окончательный дизайн второй формы

Мы создали вторую форму в проекте. Но если мы запустим проект на исполнение, то мы увидим по-прежнему первую форму. В проекте может быть сколько угодно форм, но только одна является главной. Именно она автоматически отображается при запуске приложения, а остальные являются скрытыми. Однако в наших силах поменять формы местами, т.е. сделать вторую форму главной. Для этого выберем в главном меню Project->Options, затем в появившемся мастере настроек найдём пункт Forms в левой части. Поменяем значение в поле «Main form» на Form2, воспользовавшись выпадающим списком, как показано на рис. 4.30.

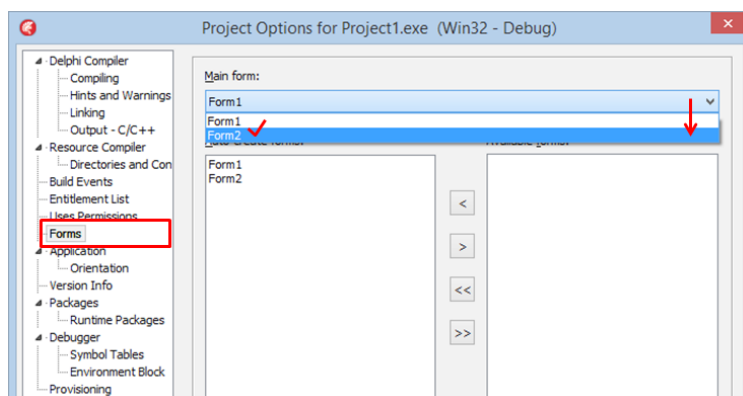


Рис. 4.30. Изменение главной формы в приложении

Сохраним и запустим приложение. Теперь появляется вторая форма, которую легко опознать по серому фону 3D-сцены.

Мы отработали технику создания дополнительных форм с практически готовой 3D-сценой за счет копирования. Как говорилось ранее, если нужно значительно переделать дизайн формы, то лучше не удалять-добавлять компоненты, а копировать их на дополнительную форму. Если новый дизайн окажется неудачным, то по крайней мере останется предыдущий вариант.

Теперь добавим интерактивность к объектам пространственной сцены, т.е. возможность реагировать на действия пользователя. Каждый 3D-объект имеет набор событий: `OnClick`, `OnMouseDown`, `OnMouseMove` и т.д. Не все из них будут работать адекватно, если мы соберём приложение под мобильную платформу. Для мобильного приложения характерны жесты, тогда как для «настольного» — пользовательских ввод при помощи мыши или клавиатуры. Для большинства учебных приложений вполне достаточно универсального события `OnClick`. В случае настольного приложения данное событие будет срабатывать при щелчке левой кнопкой мыши, а для мобильного — лёгкое касание пальцем или специальным цифровым пером.

Начнём с молекулы углекислого газа, которая располагается в левом верхнем углу. Она состоит из трёх сфер. Соответственно нужно написать

единую процедуру отклика для всех трёх сфер, как будто они представляют единый объект. Выделим первую сферу в Object Inspector, нажмём клавишу Shift и выберем последовательно две оставшиеся. Перейдем в Object Inspector на закладку Events и дважды щёлкнем в пустом поле рядом с OnClick (Рис. 4.31).

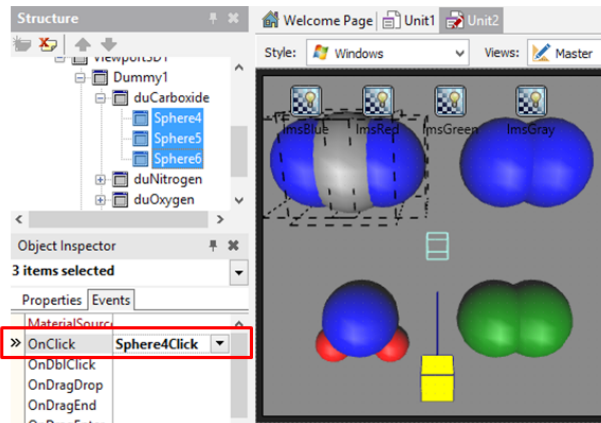


Рис. 4.31. Создание отклика на событие для нескольких объектов

Как обычно, среда разработки Delphi IDE сгенерирует шаблон процедуры-отклика на событие. На данное событие заставим молекулу сделать один оборот вокруг своей оси, а в поле TМемо выведем информацию о молекуле углекислого газа. Введем следующий программный код в шаблон процедуры отклика:

```
procedure TForm2.Sphere4Click(Sender: TObject);
begin
  duCarboxide.AnimateFloat('RotationAngle.Y', 360, 2);
  Memo1.Lines.Clear;
  Memo1.Lines.Add('Диоксид углерода (углекислый газ, двуокись углерода, '+
    ' оксид углерода (IV), угольный ангидрид) – бесцветный газ '+
    '(в нормальных условиях), без запаха, со слегка кисловатым вкусом.'
  );
end;
```


Сохраним и запустим проект. При щелчке мыши на любой из сфер, составляющих молекулу углекислого газа, она будет совершать оборот, а в TМето будет появляться поясняющий текст. Рассмотрим программный код в процедуре отклика. В первой строке тела процедуры мы реализуем анимацию для компонента-пустышки, который отвечает за молекулу углерода `duCarboxide`. Мы вызываем его метода `AnimateFloat` и передаём в него параметры, обеспечивающие оборот вокруг вертикальной осин на 360 градусов в течение 2 секунд. Далее мы очищаем содержимое строк компонента TМето, а затем добавляем в него текст. Поскольку текст длинный, то мы разбиваем его на строки, заключенные в кавычки, которые «склеиваем» в одну строку операторами «+».

Чтобы собрать приложение для мобильной платформы надо переключиться на панель Project Manager, поменять Target Platforms с Windows на Android, а затем выбрать подключенное через USB устройство. Автор подключал планшет Samsung Galaxy Tab 4 10.1, который отображался в подключенных устройствах как SM-T530, что являлось модельным кодом планшета. Сборка под Android занимает, более продолжительное время по сравнению с Windows, зато приложение на планшете более полезно, т.к. он всегда под рукой.

Для завершения проекта нужно проделать рассмотренную процедуру добавления интерактивности для оставшихся трёх молекул: кислорода, воды и азота.

Приложение можно развить и улучшить, воспользовавшись следующими советами:

- помимо поворота при клике на молекуле можно также анимировать значения `Scale` по отдельным осям (`Scale.X`, `Scale.Y`, `Scale.Z`); молекула будет не только вращаться, но и визуальнo увеличиваться;
- сделайте третью форму, на которую разместите `TTabControl`, а с первой и второй формы скопируйте 3D-сцены на отдельные странички; тогда в приложении будет единая форма для двух вариантов сцен;

третью форму сделайте главной, а первые две удалите из проекта (предварительно протестировав приложение);

- можно не ограничиваться воздухом и создать небольшой каталог наиболее часто встречающихся газов в обычной жизни или используемых в промышленности;
- правильным улучшением с точки зрения качества кода будет не хранение описательного текста в коде программы, как это реализовано сейчас, а его загрузка из текстового файла.

4.5 Групповое взаимодействие при работе над 3D-проектом

Одним из эффективных способов использования знаний о 3D-программировании мобильных приложений является создание проектов-каталогов или проектов-энциклопедий. Такого вида приложения нужны для просмотра различного рода готовых моделей с их описаниями. Можно создать энциклопедию пространственных геометрических фигур, анимированный иллюстративный справочник доказательств известных теорем, наглядное пособие по сборке скворечника из досок, трёхмерную схему спортивной площадки или школы и т.д. Разработка подобных проектов требует группового участия.

Один членов такой команды может быть программистом, т.е. использовать среду разработки Delphi/C++Builder/RAD Studio и создавать приложение. Другой — специалистом по компьютерной графике. Он будет создавать 3D-модели объектов в каком-либо трёхмерном редакторе для последующего использования в программе. Третий — подбирать модели для рисования, каталогизировать их, находить интересные текстовые описания. Четвёртый — тестировать программу.

Теперь рассмотрим более детально роли в команды с последовательностью их работы:

1. Аналитик, генератор идей, разработчик конвента: определяет предметную область энциклопедии/каталога/справочника, подбирает исходный материал в виде картинок и описаний. Также в его полномочия входит контроль и координация действий остальных участников.
2. Специалист по 3D-графике или 3D-дизайнер: создаёт трёхмерные модели по картинкам, эскизам или текстовым описаниям либо в среде разработки, либо во внешнем графическом редакторе.
3. Программист размечает интерфейс с элементами управления и вставляет готовые модели:
 - из другого проекта, созданного дизайнером, через буфер обмена;
 - из проекта, созданного дизайнером, импортируя готовые формы;
 - из файла, созданного во внешнем графическом редакторе в формате obj или dae, при помощи компонента TModel3D.
4. Тестировщик проверяет работоспособность приложениях по мере добавления новых компонентов, объектов или функций. Также тестировщик может несколько расширить свои полномочия и давать советы аналитику и разработчику в плане повышения удобства интерфейса, наглядности трёхмерного представления моделей и качества представляемых описаний.

Если не удаётся подобрать именно четырёх человек, объединённых общей идеей создания проекта, то можно создать и более тесную группу единомышленников. Тогда роли придётся совмещать. Однако не следует совмещать роль программиста и тестировщика. При проверке работоспособности приложения нужен человек со «свежим взглядом» и разумной долей критицизма в суждениях.

При коллективной работе над проектом важно соблюдать правильное распределение нагрузки и ритмичность в процессе его реализации. Нужно составить и стараться придерживаться проектного календаря. Немаловажную роль играет стиль общения в команде, он должен быть максимально дружеским, но деловым. И, конечно, все члены должны быть заряжены оптимизмом и сообща двигаться к конечной цели — работающему полнофункциональному приложению.

Математика и программирование

5.1. Начало приложения для графиков функций

Математические задачи часто вызывают сложности при решении, если их не проиллюстрировать наглядными графическими изображениями. Без них сложно проанализировать функцию, понять смысл решения уравнений и т.д. Если мы говорим о науках, интенсивно использующих математический аппарат, таких как физика, химия, биология, экономика, то графики функций помогают сделать правильные выводы о наблюдаемых законах, а также прогнозировать развитие событий. Сейчас нашей задачей будет создать приложение, в которых отображается график функции. Как и ранее, приложение будет как «настольным», так и «мобильным».

Создадим новую папку для проекта «Project 5.1». Запустим Delphi/RAD Studio IDE, выберем в главном меню File->New->Multi-Device Application — Delphi. Выберем Blank Application. Создаваемое нами приложение будет очень похоже своим интерфейсом на типовое. Новому пользователю будет комфортно с работать с таким интерфейсом, т.к. не придётся тратить время на знакомство с расположением основных кнопок или поиском инструментальной панели.

Добавим на форму компонент TToolBar. По умолчанию его свойство Align установлено в значение Top, поэтому он автоматически займёт самое верхнее пространство клиентской части формы. Добавим второй аналогичный компонент, он займет теперь уже второе по счету свободное место сверху формы. Выделим его, перейдём в Object Inspector и установим свойство Align в значение Bottom. Затем разместим на форме новый для нас компонент TChart, а его свойство Align зададим как Client. После данных

манипуляций наша форма должна выглядеть так, как показано на рис. 5.1. Сохраним проект и выполним пробный запуск приложения под Windows.

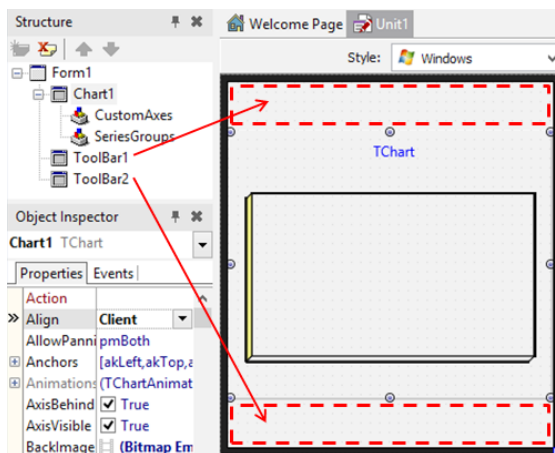


Рис. 5.1. Первичная разметка формы приложения

Разместим на нижнем ToolBar2 компонент TSpeedButton. Значение свойства StyleLookup зададим как actiontoolbutton, и на кнопке автоматически появится «флажок». Свойство Align определим как Right. Выделим данную кнопку, перейдем на закладку Event панели Object Inspector, найдём строку OnClick и щёлкнем два раза в пустом поле рядом с надписью. IDE сгенерирует уже привычную нам заготовку процедуры отклика. Дополним текст введённым нами кодом и обсудим полученное:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);  
begin  
  Chart1.Series[0].AddXY(0, 5);  
  Chart1.Series[0].AddXY(1, 2);  
  Chart1.Series[0].AddXY(2, 6);  
  Chart1.Series[0].AddXY(3, 3);  
end;
```

Не спешите запускать приложение, т.к. нам нужно ещё правильно настроить компонент TChart в design-time. Компонент TChart нужен для отображения различного рода графиков. Программисту не нужно размечать

плоскость изображения, проводить линии, рассчитывать масштаб, подбирать цвет и т.д. Всё это уже реализовано в TChart, нужно лишь уметь правильно настроить его в design-time.

После появления на форме компонент TChart не содержит никаких линий графиков. Для того, чтобы мы увидели графики в виде, например, линий, нужно сначала их создать. Щелкнем два раза на компоненте Chart1 — откроется мастер его настройки. Для добавления графика выберем пункт Series, нажмём кнопку Add и в открывшейся галерее графиков выберем самый простой первый шаблон Line. Для первого знакомства лучше снять «галочку» около 3D, наш график будет плоским. Потом нажмем ОК. На рис. 5.2 схематично показана последовательность действий.

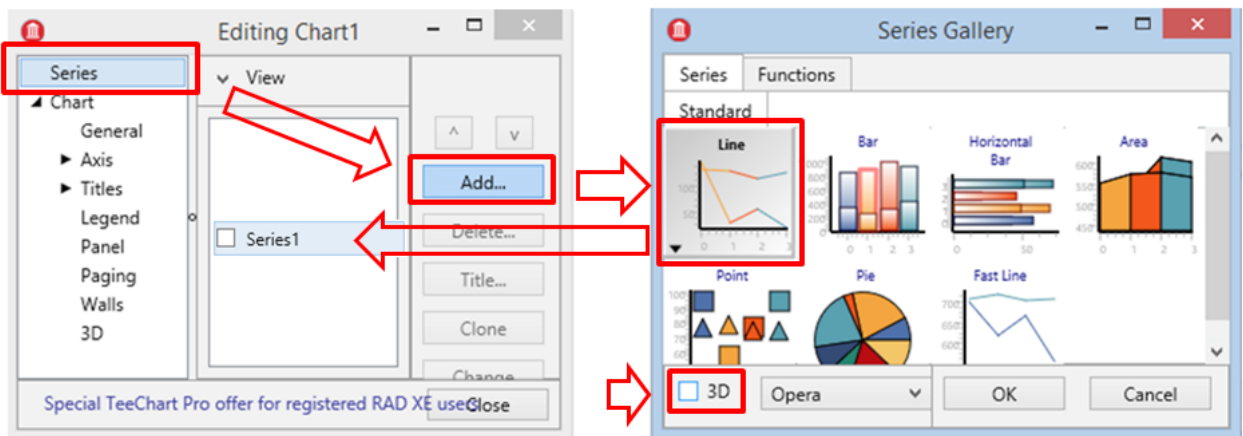


Рис. 5.2. Мастер создания графиков в TChart

После этого в design-time на форме появился график в виде ломаной линии. На самом деле он показывает лишь некие случайные данные. Запустим приложение и увидим, что график по-прежнему пуст. Но теперь если на интерфейсе мы нажмём на кнопку с «флажком», то на графике появится линия, соответствующая введенному выше коду. Теперь разберём код. Каждая конструкция `Chart1.Series[0].AddXY` в переводе с языка Object Pascal означает, что:

- компонент `Chart1` содержит массив графиков-линий `Chart1.Series`;
- следует обращение к первой графической линии по индексу «0» `Chart1.Series[0]`;
- в первую графическую линию добавляется точка с координатами (X, Y).

Полностью добавление точки выглядит так: `Chart1.Series[0].AddXY(..., ...)`, а вместо многоточия в последнем пункте подставляются числа — координаты точки на графике. Можно перевод с «дельфийского» (не путать с «эльфийским») начать с конца:

- добавляем точку с координатами (X, Y) методом `AddXY(..., ...)`;
- к первой графической линии `Series[0].AddXY(..., ...)`;
- который принадлежит к компоненту `Chart1` — `Chart1.Series[0].AddXY(..., ...)`.

Давайте повторим последовательность действий, когда нам понадобится добавить очередную графическую линию в компонент `TChart`:

- два раза щёлкнуть на компоненте `TChart`, чтобы вызывать мастер его настройки;
- добавить линию графика (или другой вид графика) в узле `Series`;
- в исходном коде к добавленной новой линии графика нужно обращаться как `Chart1.Series[n-1]`, где `n` — количество уже добавленных линий графиков.

5.2. Отображения графика функции

Отличным показательным примером использования компонента TChart будет построение графика какой-либо функции. Выше мы добавили некие произвольные данные, теперь же будем рассчитывать координаты каждой новой точки по формуле $y=a*x^2+b$, где a и b — некие произвольные числа. Впоследствии мы будем иметь возможность изменить их с интерфейса пользователя. Рассчитывать координаты мы будем последовательно для каждой точки, перебирая их в цикле. Рассмотрим рис. 5.3, чтобы затем правильно написать программный код.

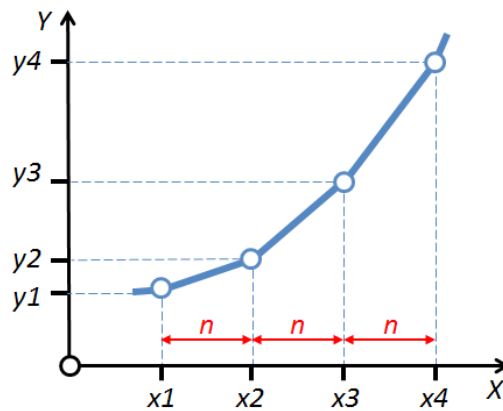


Рис. 5.3. Расчет координат для построения графика функции

Для простоты представим всё в действительных числах. Функция запишется как $y=x*x+2$. Координаты точек графика на рис. 5.3 представлены в таблице:

x	y
$x_1=1$	$y_1=1*1+2=3$
$x_2=2$	$y_2=2*2+2=6$
$x_3=3$	$y_3=3*3+2=11$
$x_4=4$	$y_4=4*4+2=18$

Чтобы построить график функции $y=x^2+2$ в интервале от $x=1$ до $x=4$ нужно изменить код процедуры отклика на нажатие SpeedButton1:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);  
begin  
Chart1.Series[0].AddXY(1, 3); // добавляем 1-ю точку  
Chart1.Series[0].AddXY(2, 6); // добавляем 2-ю точку  
Chart1.Series[0].AddXY(3, 11); // добавляем 3-ю точку  
Chart1.Series[0].AddXY(4, 18); // добавляем 4-ю точку  
end;
```

Сохраним и запустим проект на исполнение. После нажатия на кнопку с «флажком» в компонент Chart1 будут переданы 4 точки с соответствующими координатами. Они попадут в первую и единственную линию графика. Компонент Chart1 выполнит отрисовку отрезков, соединяющих точки. Можно проверить точность построения графика, сравнив график, построенный при помощи нашего приложения, с графиком, построенным при помощи, например, Microsoft Excel (рис. 5.4). Это очень полезный приём, когда математические алгоритмы и построения графиков проверяются при помощи сторонних программных продуктов.

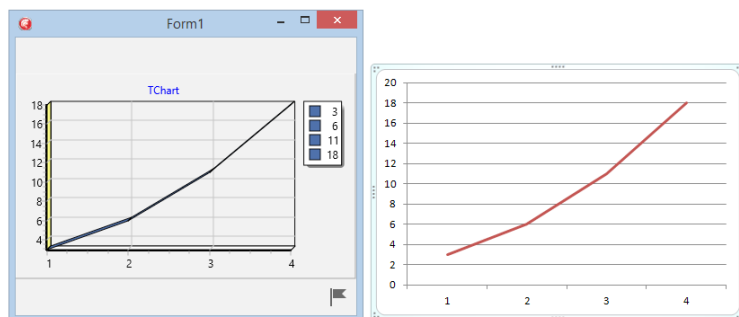


Рис. 5.4. График функции при помощи компонента TChart и Microsoft Excel

5.3. Улучшение программного кода для построения графиков

В процедуре отклика на событие мы использовали достаточно тривиальный код, который просто добавлял последовательно четыре точки. Такой код не несёт информации о типе функции, график которой мы построили. Если нам нужно изменить вид функции или увеличить диапазон построения, то придётся заново высчитывать точки вручную. Нам же нужно создать инструмент анализа графика функции, поэтому всё должно происходить автоматически.

Переделаем программный код следующим образом:

```
function MyFunction(xi: single): single;
var
  yi: single;
begin
  yi:= xi*xi + 2;
  result:= yi;
end;
procedure TForm1.SpeedButton1Click(Sender: TObject);
var
  x, y: single;
begin
  x:= 1; y:= MyFunction(x); // вычисляем 1-ю точку
  Chart1.Series[0].AddXY(x, y); // добавляем 1-ю точку
  x:= 2; y:= MyFunction(x); // вычисляем 2-ю точку
  Chart1.Series[0].AddXY(x, y); // добавляем 2-ю точку
  x:= 3; y:= MyFunction(x); // вычисляем 3-ю точку
  Chart1.Series[0].AddXY(x, y); // добавляем 3-ю точку
  x:= 4; y:= MyFunction(x); // вычисляем 4-ю точку
  Chart1.Series[0].AddXY(x, y); // добавляем 4-ю точку
end;
```

Проанализируем введённый код. Прежде всего, мы создали отдельный элемент программы — функцию. Функция — это изолированная часть кода, которая начинается функцией ключевым словом **function** и имеет тело, начинающееся словом **begin**, а заканчивающаяся словом **end** и точкой с запятой. Перед телом функции присутствует раздел переменных, который обозначен словом **var**, где объявлена переменная **yi**. После назва-

ния `MyFunction` в скобках указан единственный параметр `xi`, а после двоеточия обозначается тип возвращаемого значения `single`.

Сама по себе функция — некоторое действие, которое требует входной параметр и выдаёт результат. Уподобим её миксеру, который может что-то взбить. Если входной параметр — яйца, то после работы миксера мы получим результат — гоголь-моголь. Можно изменить входной параметр: вместо яиц поместить сливки. Тогда выходным результатом будут взбитые сливки. Между этими блюдами есть нечто общее — способ приготовления, т.е. интенсивное перемешивание или взбитие. Но результат зависит от входного параметра. Однако входной параметр может быть недопустимым. То результат может быть странным или нежелательным. Например, если положить в чашу миксера яблоко, то можно сломать миксер. Отсюда следует важный вывод: не всякий параметр можно передавать в функцию.



Рис. 5.5. Сравнение функции с миксером

В функцию `MyFunction` передаётся один параметр вещественного типа `xi`. В разделе переменных объявляется локальная переменная `yi`. Её значение вычисляется по формуле $y(x)=x^2+2$, а чтобы функция вернула результат, значение переменной `yi` присваивается специальной переменной

result. В функции мы используем обозначения x_i и y_i чтобы подчеркнуть — это не те переменные, которые используются в процедуре отклика OnClick.

Процедуру OnClick мы переделали так, чтобы все вычисления выполняла функция MyFunction, а мы только передавали в неё значения x последовательно для четырёх точек. Смысл инструкции $y := \text{MyFunction}(x);$ понятен просто. Мы передаём значение переменной x в функцию MyFunction, а возвращаемый результат копируем в переменную y . Потом добавляем в линию графика вычисленные координаты очередной точки: $\text{Chart1.Series}[0].\text{AddXY}(x, y);$. На рис. 5.6 показана схема передачи значения параметра в функцию и возврат результата.

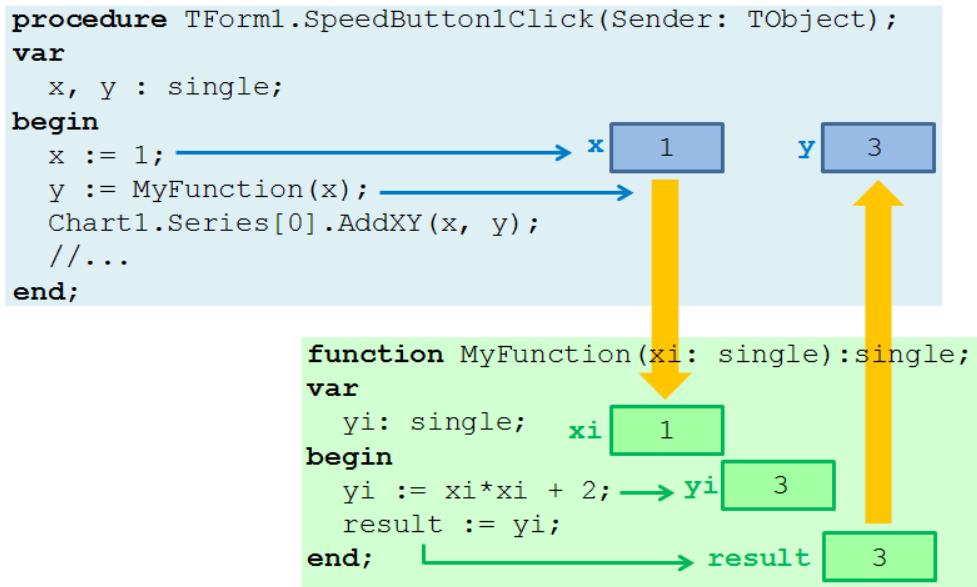


Рис. 5.6. Схема вызова функции и передача значений

Мы могли не вводить отдельную функцию MyFunction, написав всё код в процедуре отклика OnClick, но мы получили более структурированный

код. Теперь он состоит из процедуры отклика `OnClick` и чисто математической функции `MyFunction`, где мы можем изолированно изменить вид функции. Например, если мы изменяем вид функции с $y(x)=x^2+2$ на $y(x)=2x^2+3$, то нам не потребуется просматривать весь код, т.к. вид функции графика задан в `MyFunction`.

Перейдем к рассмотрению понятия локальных переменных. Вернёмся к рис. 5.6, где показана схема взаимодействия процедуры `OnClick` и функции `MyFunction`. И в процедуре, и в функции объявляются переменные в разделе `var`. Объявленные переменные в разделе `var` являются локальными. «Локальный» означает «местный», «не выходящий за определённые пределы». Локальные переменные существуют только в пределах функции или процедуры, где они объявлены. Рис. 5.6 цветом показано, что синие переменные «живут» в синем пространстве, а зелёные — в зелёном. Они принадлежат разным областям.

Теперь нужно рассмотрим передачу значений из процедуры отклика `OnClick` в функцию `MyFunction`, т.е. из синей области в зелёную, а затем обратно. При вызове функции значение «синей» переменной `x` передаётся в переменную `x1` из «зелёной» области. После этого выполняются строки кода `MyFunction`. В конце функции рассчитанное значение присваивается переменной `result`, и её значение будет передано обратно в синюю область в точку вызова `MyFunction (x)` и скопированно в переменную `y` за счёт оператора присваивания: `y := MyFunction (x) ;`.

5.4. Настройка компонента TChart

В режиме «по умолчанию» компонент TChart отрисовывает линию, подбирая масштаб графика и диапазон таким образом, чтобы максимально занять предоставленную площадь. Не всегда это является удобным и наглядным, поэтому настроим отображение графика на заданный диапазон. Для этого дважды щёлкнем на компоненте Chart1 и выберем в появившемся окне раздел Axis. После этого щёлкнем на сточке Bottom, что будет означать нижнюю ось, а потом обратим внимание на центральную часть окна с настройками (рис. 5.7).

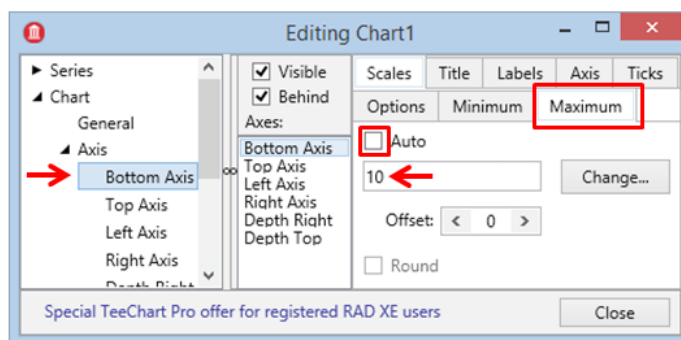


Рис. 5.7. Настройка осей TChart

На рис. 5.7 показано, как нужно настроить максимальное значение диапазона на нижней оси. Настроим диапазон значений левой вертикальной (OY) и нижней горизонтальной (OX) осей, как показано в таблице ниже:

Нижняя ось — ось OX (Bottom Axis)	
Minimum	0
Maximum	5
Верхняя ось — ось OY (Top Axis)	
Minimum	0
Maximum	20

Сохраним и запустим приложение, а затем нажать на кнопку с «флажком». Теперь график будет строиться в заданном диапазоне, однако в левой части появится «легенда», которую желательно убрать. Опять дважды щёлкнем на компоненте Chart1, затем выберем раздел Legend и снимем «галочку» Visible. Запустим приложение. Теперь график будет чётко вписываться в заданный диапазон.

Продолжим улучшение внешнего вида графика. Пока график отображается в трёхмерном виде, а нам нужно получить его в плоском виде. Войдём в режим редактирования графика привычным способом и выберем раздел 3D, после чего снимем «галочку» 3 Dimensions. После этого увеличим толщину линии. В редакторе графика войдём в раздел Series и выберем единственную Series1. В разделе Format перейдём на закладку Border и движком Width увеличим значения приблизительно до 3...4. Завершим настройку TChart следующим действием. Изменим заголовок графика, выбрав Chart->Titles->Title, а затем введя текст отображения функции $y(x)=x^2+2$, как показано на рис. 5.8.

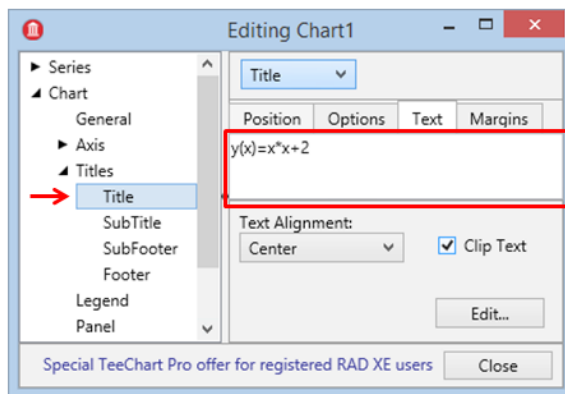


Рис. 5.8. Изменение заголовка графика

Работая в мастере настройки TChart, мы познакомились лишь с малой частью всех возможностей данного компонента в режиме design-time. Также можно изменять вид графика за счёт его свойств в режиме runtime.

5.5. Новые возможности построения графика функции

Конечная цель нашего проекта — дать возможность пользователю изменять и диапазон построения графика, и саму функцию. Реализуем возможность ввода диапазон построения графика в виде начальной и конечной точки. Модифицируем процедуру отклика следующим образом:

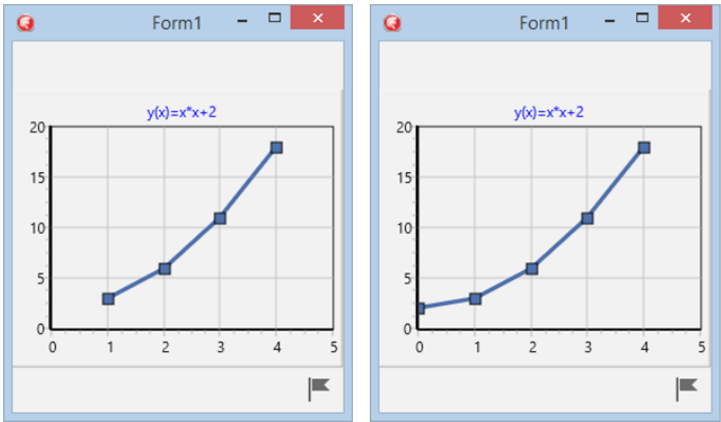
было
<pre> procedure TForm1.SpeedButton1Click(Sender: TObject); var x, y: single; begin x:= 1; y:= MyFunction(x); <i>// вычисляем 1-ю точку</i> Chart1.Series[0].AddXY(x, y); <i>// добавляем 1-ю точку</i> x:= 2; y:= MyFunction(x); <i>// вычисляем 2-ю точку</i> Chart1.Series[0].AddXY(x, y); <i>// добавляем 2-ю точку</i> x:= 3; y:= MyFunction(x); <i>// вычисляем 3-ю точку</i> Chart1.Series[0].AddXY(x, y); <i>// добавляем 3-ю точку</i> x:= 4; y:= MyFunction(x); <i>// вычисляем 4-ю точку</i> Chart1.Series[0].AddXY(x, y); <i>// добавляем 4-ю точку</i> end; </pre>
стало
<pre> procedure TForm1.SpeedButton1Click(Sender: TObject); var x, y: single; i: integer; <i>// переменная цикла</i> begin for i:= 1 to 4 do begin x:= i; y:= MyFunction(x); <i>// вычисляем i-ю точку</i> Chart1.Series[0].AddXY(x, y); <i>// добавляем i-ю точку</i> end; end; </pre>

В прежнем варианте мы 4 раза делали одно и то же: вычисляли и добавляли точку в график. А в новом варианте используем цикл, т.е. многократно повторяющуюся группу действий. Нельзя сказать, что мы «ходим по кругу», т.к. при каждом новом повторении изменяется значение переменной цикла *i* и, следовательно, значение переменной *x*, которая последовательно принимает значения 1, 2, 3, 4. Таким образом, новый код при исполнении даёт тот же результат, что и прежний, но при этом является более эффек-

тивным. Например, мы легко можем изменить количество проходов цикла и, следовательно, количество точек на графике:

было	стало
for i:= 1 to 4 do	for i:= 0 to 4 do

Скомпилируем и запустим проект на исполнение. Результат показан на рис. 5.9. Мы видим, что точек на графике стало больше за счет увеличения количества проходов в цикле. В первоначальном варианте кода нам пришлось бы добавлять дополнительные строки кода. А в новом –достаточно изменить пару числовых констант.



было: 4 точки

стало: 5 точек

Рис. 5.9. Увеличение числа точек на графике

Чтобы на графике появились красивые квадратики, обозначающие вычисленные точки, нужно в design-time дважды щёлкнуть на TChart, выбрать узел Series, кликнуть на единственной линии графика Series1, перейти на закладку Forman->Pointer и установите «галочку» Visible (рис. 5.10).

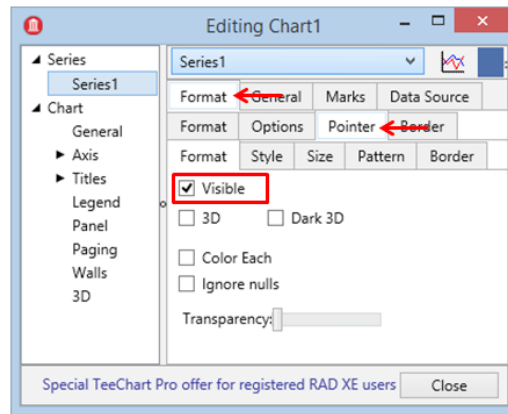


Рис. 5.10. Настройка отображения точек графика

Перейдём к следующему этапу реализации добавления точки к графику без изменения программного кода. Чтобы построить график функции на произвольном интервале с заданным количеством точек, нужно выполнить ряд шагов:

- задать количество точек на графике;
- определить начальную и конечную точки диапазона;
- рассчитать координаты каждой точки;
- добавить координаты каждой точки в линию графика.

На рис. 5.11 показано, как мы будем вычислять координаты, зная начальную и конечную точки, а также их количество на графике.

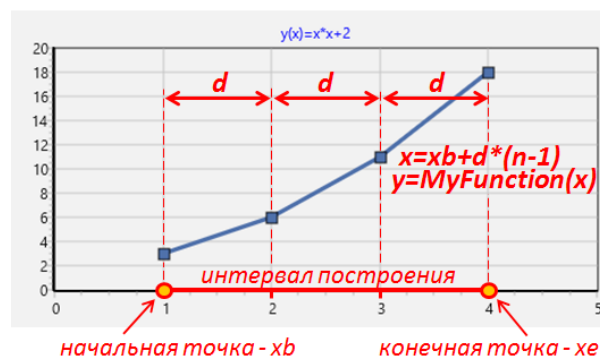


Рис. 5.11. Схема вычисления точек графика функции

```

procedure TForm1.SpeedButton1Click(Sender: TObject);
var
    xb, xe: single; // начальная и конечная точки интервала
    n: integer; // количество точек на графике
    d: single; // расстояние между точками
    x, y: single; // координаты текущей точки
    i: integer; // переменная цикла
begin
    xb:= 1; // задаем начальную точку интервала
    xe:= 4; // задаем конечную точку интервала
    n:= 4; // задаем количество точек на графике
    d:= (xe - xb) / (n - 1); // вычисляем расстояние между
    точками
    for i:= 1 to n do // тело цикла выполняется n раз
        begin
            x:= xb + d * (i - 1); //вычисляем x для i-ой точки
            y:= MyFunction(x); // вычисляем y для i-ой точки
            Chart1.Series[0].AddXY(x, y); // добавляем i-ю точку
        end;
    end;

```

Запустим готовое приложение. Результат нажатия кнопки SpeedButton1 инициирует построение графика для четырёх точек, как показано в левой части рис. 5.9. Чтобы построить график для пяти точек, как показано в правой части рис. 5.9, нужно задать *xb* равным 0, а количество точек *n* увеличить до 5. Предлагаем читателям сами внести изменения в код выше и контролировать результат по рис. 5.9.

Если по каким-то причинам что-то не получается, то следует воспользоваться отладчиком. Отладчик (debugger) встроен в интегрированную среду разработки Delphi/C++Builder/RAD Studio. В самом простом варианте отладка сводится к пошаговому выполнению программы и контролю значений переменных. Первым шагом в поиске ошибки является остановка приложения в нужной строке. Выберем эту строку и нажмем клавишу F5 или кликнем на синюю точку в поле левее начала строчки, как показано на рис. 5.12. Только что мы создали «точку останова» (breakpoint).

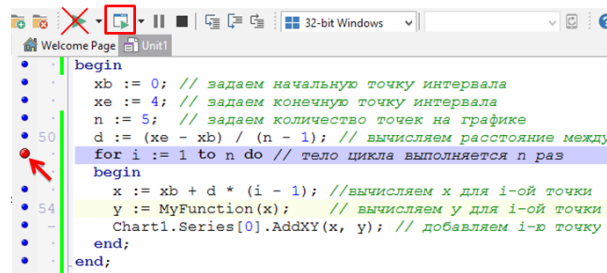


Рис. 5.12. Создание точки останова (breakpoint)

Чтобы точка останова сработала, т.е. наша программа остановилась, нужно запустить её на исполнение не обычной кнопкой с зелёным треугольником, а соседней, расположенной левее (показана также на рис. 5.12 в красном квадрате). Запустим приложение этой кнопкой, а затем на интерфейсе пользователя нажмём кнопку с «флажком» для построения графика. Поскольку мы установили точку останова именно в процедуре отклика на эту кнопку, где мы и остановимся. Автоматически откроется редактор кода, но уже в активном режиме, как показано на рис. 5.12. Теперь можно пошагово выполнять приложение и искать ошибку. Для это используются кнопки F7 и F8. Также можно использовать пункты меню Run->Trace Into и Run->Step Over.

Поясним различие между F7 (Trace Into) и F8 (Step Over). Первое действие можно перевести как «с заходом в», а второе как «перешагнуть через». Покажем это на примере. Сначала будем орудовать кнопкой F8. Нажмём её один раз, после этого подсветка строки сместится на одну ниже. Программа выполнит строку, на которой была поставлена точка останова, и после этого остановится на строке ниже (рис. 5.13).

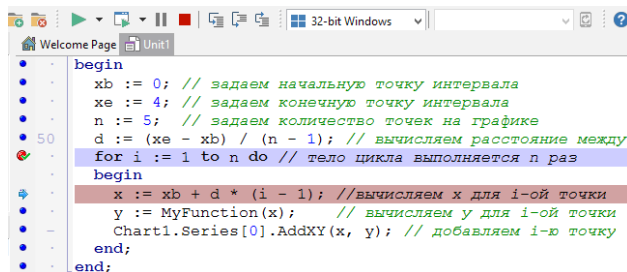


Рис. 5.13. Пошаговое выполнение программы

Отметим, что программа не остановилась на строке **begin**, т.к. она не маркирована синей точкой слева. Остановка на данной строке не произошла, т.к. она не содержит действий, а лишь обозначает начало тела цикла. Далее будем выполнять циклическое прохождение тела цикла. При работе с отладчиком в пошаговом режиме можно вернуться в режим обычного выполнения приложения, нажав F9.

Рассмотренным методом отладочного пошагового прохождения можно определить фактическое число повторений, выполненных в цикле. С одной стороны, в момент отладочного пошагового прохождения мы всегда видим значение локальных переменных в специальном окошке — Local Variables («локальные переменные»), как показано на рис. 5.14. В данном окне автоматически отображаются переменные в рамках данной процедуры или функции. Для просмотра значения переменных, причем не только локальных, можно использовать окошко Watches («просмотры»). Изначально оно пустое, но нажав правую кнопку мыши и выбрав пункт всплывающего меню «Add Watch...», можно добавить интересующую нас переменную. Еще одним способом является просто наведение курсора на какую-либо переменную в тексте, чтобы через мгновение всплыла подсказка (hint), где будет показано её значение. На рис. 5.14 показаны все эти возможности.

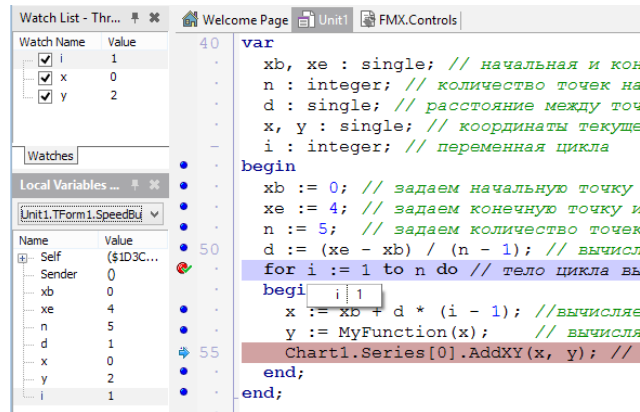
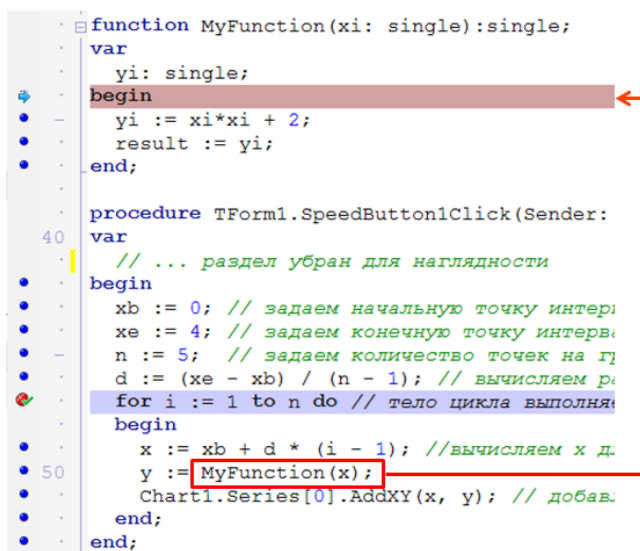


Рис. 5.14. Варианты просмотра значений переменных

Если по каким-либо причинам одно из рассмотренных окон не показано или было случайно закрыто, то их всегда можно отобразить при помощи пункта главного меню View->Debug Windows.

Только что мы рассмотрели несколько способов просмотра значения переменных. Если перед нами стоит задача быстрого просмотра значения переменной, то достаточно навести курсор. Окошко «Local Variables» отображается автоматически с локальными переменными, что вполне подходит для поиска типичных ошибок. Например, ошибка в математической формуле, неправильные имена переменных, проблемы с исправлением скопированного кода. Более целенаправленная и тщательная отладка обычно ведётся с помощью окна «Watches», когда подбирается правильная последовательность и ограниченный набор не только локальных, но и глобальных переменных. Иногда приходится комбинировать все три способа.

Перейдём к рассмотрению использования кнопки F7. Еще раз запустим приложение и нажмём на кнопку для построения графика («флажок»). Произойдёт прерывание исполнения процедуры отклика за счёт точки останова. Но теперь будем нажимать не кнопку F8 как в прошлый раз, а F7. Как только мы попадём на строчку с вызовом функции `MyFunction(x)`, то при следующем нажатии на F7 мы автоматически попадём внутрь данной функции, чего не происходило при работе с F8 (рис. 5.15).



```

function MyFunction(xi: single):single;
var
  yi: single;
begin
  yi := xi*xi + 2;
  result := yi;
end;

procedure TForm1.SpeedButton1Click(Sender:
var
  // ... раздел убран для наглядности
begin
  xb := 0; // задаем начальную точку интерв
  xe := 4; // задаем конечную точку интерв
  n := 5; // задаем количество точек на г
  d := (xe - xb) / (n - 1); // вычисляем р
  for i := 1 to n do // тело цикла выполня
  begin
    x := xb + d * (i - 1); //вычисляем x д
    y := MyFunction(x);
    Chart1.Series[0].AddXY(x, y); // добав
  end;
end;

```

Рис. 5.15. Пошаговая отладка «с заходом»

Отладка «с заходом» весьма полезна, когда мы точно не знаем, где происходят проблемы: в главной процедуре `OnClick` или в функции `MyFunction`. Следует правильным образом комбинировать отладку «перешагнуть через» и «с заходом». Поэтому в понятных местах следует идти крупными шагами «по F8», а в подозрительных — делая аккуратные мелкие шажки «по F7». В любом случае, данное умение приходит с опытом, который получается только при решении практических задач. Этим мы и займёмся в следующем разделе.

Для начинающих программистов также весьма полезно при отладке использовать методы ручного счёта. Нет ничего зазорного в том, чтобы на листке бумаги разлиновать небольшую табличку, куда вписать рассчитанные данные вручную. Можно также использовать электронную таблицу. Запишем рассчитанные вручную в таблицу или визуализируем их при помощи Microsoft Excel:

Номер прохода цикла Номер точки графика	i	x	y
1	1	0	2
2	2	1	3
3	3	2	6
4	4	3	11
5	5	4	18

Рассчитав и вписав данные один раз в таблицу, мы превратим утомительную отладку в лёгкую прогулку по коду с помощью пошаговой техники F7/F8 в комбинации с просмотром значений переменных в окне «Local Variables». И не забывайте запускать отладку кнопкой «Fun (F9)», а не «Run Without Debugging (Shift+Ctrl+F9)», иначе IDE не будет обращать внимание на точки останова. Если в какой-то момент отладку нужно завершить, то мы запускаем выполнение дальше при помощи F9 или просто завершаем работу приложения кнопкой «Program Reset (Ctrl+F2)», которую легко найти — она обозначена красным квадратом. С таким мощным инструментом поиск ошибок напоминает компьютерные игры в жанре Quest.

5.6. Модификация интерфейса

В предыдущем разделе мы добились работоспособности приложения, которое теперь делает главную полезную функцию — строит график. Теперь мы пойдем дальше и добавим интерактивности и дружелюбности нашему интерфейсу. Если в работе над проектом вы сделали перерыв, то нужно заново открыть проект, выполнив в главном меню File->Open Project.... Желательно предварительно сделать копию папки проекта, переименовав её в «Project 5.1». Тогда исходная версия проекта сохраниться в папке «Project 5». Открыв проект, займёмся ре-дизайном (изменением дизайна) главной формы. Вырежем и вставим SpeedButton1 на верхнюю инструментальную панель. На нижней инструментальной панели разместим 8 компонентов TSpeedButton. Первым четырьмя последовательно установим свойство Align в значение Right. Вторым четырём — Left. При помощи выпа-

дающего списка свойства StyleLookup поочередно выберем кнопкам такие стили, чтобы они по внешнему виду соответствовали рис. 5.16.

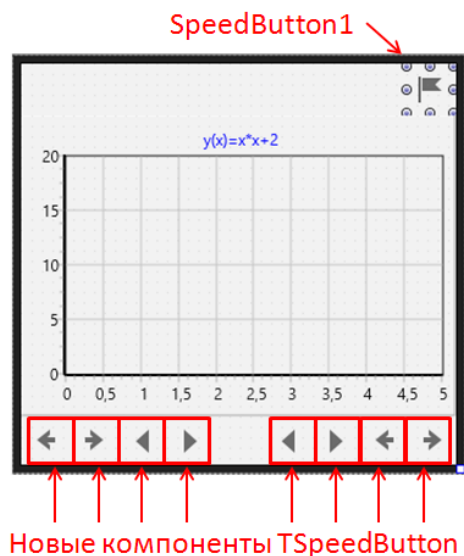


Рис. 5.16. Модифицированный интерфейс

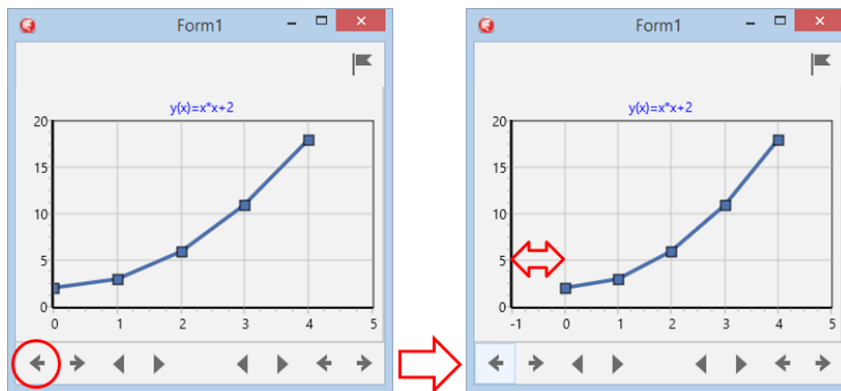
Крайние слева кнопки со «стрелочками» будут двигать левую границу поля графика влево и вправо, соответственно. Для самой левой кнопки с символом $\leftarrow$ на событие OnClick введём следующий код:

```
procedure TForm1.SpeedButton7Click(Sender: TObject);  
begin  
  Chart1.Axes[0].Minimum:= Chart1.Axes[0].Minimum-1;  
end;
```

Обратим внимание, что мы добавляли 8 кнопок подряд. Естественно, их названия назначались последовательно с точки зрения среды разработки IDE, но при размещении на форме они могли перемешаться. Самая левая кнопка может называться не SpeedButton7, как в коде выше, а по-другому. Об этой поправке нужно помнить. А лучше давать названия компонентам на основе их функций.

Запустим приложение, построим график кнопкой с «флажком», а затем нажмём самую левую кнопку с символом $\leftarrow$. Изменится координата левой границы на оси ОХ при неизменной ширине поля графика. В коде выше мы берём значение «минимума» (Minimum) первой оси из массива осей с индексом 0 (Axes[0]) нашего графика (Chart1), уменьшаем её на единицу и снова копируем в значение «минимума». Результат работы кода показан на рис. 5.17.

Таким образом мы решили задачу повышения уровня эргономики интерфейса или, как говорят, «юзабилити». Теперь пользователь может смещать границу поля графика, чтобы более наглядно представить вид линии. Далее вам предстоит самостоятельно запрограммировать события OnClick для оставшихся трёх кнопок со «стрелками» $\rightarrow$, $\leftarrow$ и $\rightarrow$. Как и в прошлый раз будем вводить код для процедур OnClick каждой из кнопок. Для двух последних вместо Minimum, будем вводить Maximum, добавляя или отнимая единицу. Тестирование позволит подтвердить корректность кода.



5.17. Логическое смещение левой границы поля графика

Теперь перейдём к более сложной задаче — добавлению точки к графику слева и справа при помощи левой группы кнопок $<$, $>$ и аналогичной правой группы $<$, $>$. Добавление или удаление точки графика означает изменение диапазона построения. Это мы решаем за счёт увеличения или

уменьшения количества проходов по циклу. Вернёмся к рассмотрению кода события OnClick для SpeedButton1 или кнопки с «флажком». Нет смысла приводить весь код здесь, его легко можно найти выше в разделе 5.3 или переключиться в IDE. Нас будут интересовать строчки кода:

```
xb:= 0; // задаем начальную точку интервала  
xe:= 4; // задаем конечную точку интервала  
n:= 5; // задаем количество точек на графике
```

Согласно коду выше значение начальной точки интервала, конечной точки интервала и количества точек на графике задаётся жёстко. Каждый раз при нажатии на кнопку SpeedButton1 всё повторяется снова. Переменные `xb`, `xe` и `n` — локальные, т.е. объявлены внутри процедуры. Их никак нельзя изменить снаружи. Выходом является вынесение этих переменных за пределы функции, чтобы их можно было изменить. Перечислим, какими способами это можно сделать:

- сделать переменные глобальными (а не локальными);
- преобразовать данные переменные во входные параметры с последующей модификацией остального кода программы;
- перенести переменные в класс формы.

Глобальные переменные использовать не рекомендуется в принципе без жёстких на то оснований, т.к. они становятся общедоступными и любой код может испортить их значение вследствие ошибки программиста. Преобразование локальных переменных в параметры функции — правильный подход, позволяющий сделать код более универсальным. Но мы выберем третий подход, как наиболее простой. Вырежем объявление переменных из раздела `var` процедуры OnClick:

```
xb, xe: single; // начальная и конечная точки интервала  
n: integer; // количество точек на графике
```

и вставим их в раздел `private` класса формы (символы `//...` означают опущенный для компактности код):

```

type
TForm1 = class(TForm)
ToolBar1: TToolBar;
...
private
{Private declarations}
xb, xe: single; // начальная и конечная точки интервала
n: integer; // количество точек на графике
public
{Public declarations}
end;

```

Далее вырежем уже упомянутые строчки кода с жёстким заданием значений из тела процедуры OnClick:

```

xb:= 0; // задаем начальную точку интервала
xe:= 4; // задаем конечную точку интервала
n:= 5; // задаем количество точек на графике

```

и вставим их в подходящее место.

Найдём то место программы, которое будет исполняться всегда перед построением графика. В противном случае *xb*, *xe* и *n* могут быть случайными и/или ошибочными, и график будет построен неправильно. Для соблюдения этого условия подходит событие OnCreate, которое происходит при создании формы. Выберем в панели Structure форму Form1, а затем в Object Inspector найдем событие OnCreate и дважды щёлкнем в пустом поле данной строчки. Среда IDE сгенерирует шаблон процедуры отклика, куда мы и вставим строчки из буфера:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
xb:= 0; // задаем начальную точку интервала
xe:= 4; // задаем конечную точку интервала
n:= 5; // задаем количество точек на графике
end;

```

Сохраним, запустим и протестируем приложение. В следующем разделе мы обсудим понятие «класс формы», куда мы поместили описания переменных.

5.7. Некоторые сведения об объектно-ориентированном программировании

Мы привыкли, что в проекте есть одна главная форма. Каждый раз, переходя в Unit1.pas для ввода кода процедур-откликов, мы считали его «средой обитания» нашей единственной формы. Но это — совсем не так. Unit1.pas содержит в себе описание «класса формы», некоего «чертежа» или «схемы сборки», по которому изготавливается та самая форма, которая видна в запущенном приложении. «Класс» формы или любого другого объекта в программном коде означает модель, шаблон, чертёж, рецепт, инструкция по изготовлению. Реальную форму можно назвать экземпляром или объектом. Класс — один, объектов — много: по одному чертежу можно изготовить много одинаковых деталей. Также класс можно уподобить рецепту, например, шоколадного коктейля, согласно которому можно изготовить сколько угодно одинаковых по вкусу порций рис. 5.18.



Рис. 5.18. Класс и объекты (экземпляры)

В нашем случае по рецепту (классу формы) изготавливается один коктейль (экземпляр формы), но в этом нет ничего странного. Нет ничего предосудительного в том, что вы решите побаловать только себя! Но в случае необходимости угостить друга, по рецепту вы сможете сделать точно такую же порцию. Но пока остановимся на создании одной формы в соответствии

с её классом. Можно, конечно, на основе одного класса изготовить несколько одинаковых форм, но в большинстве проектов этого не требуется.

Перейдём к рассмотрению класса формы уже с точки зрения объектно-ориентированного языка программирования Object Pascal:

```
type
TForm1 = class(TForm)
ToolBar1: TToolBar;
// ... код опущен для краткости
procedure SpeedButton6Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
private
  {Private declarations}
  xb, xe: single; // начальная и конечная точки интервала
  n: integer; // количество точек на графике
public
  {Public declarations}
end;
```

Здесь название класса TForm1 созвучно имени формы Form1, а ToolBar1 — название первой добавленной инструментальной панели. Далее мы убрали часть кода, а SpeedButton6Click представляет собой название процедуры отклика на событие OnClick компонента SpeedButton6. В разделе private мы встречаем переменные, которые уже не локальные, т.е. принадлежащие какой-либо процедуре или функции, а члены класса формы. И, наконец, процедура отклика на событие OnCreate под названием FormCreate. Класс формы по аналогии со «схемой сборки» содержит описание того, что входит в нашу форму Form1. Кое-что попало в класс формы автоматически, когда мы занимались созданием дизайна формы: ToolBar1 или SpeedButton6Click. Но мы также изменили класс вручную, добавив в него переменных. В классе формы находятся компоненты, процедуры отклика и переменные.

Отдельно остановимся на событии OnCreate. Поскольку в классе формы есть переменные, то каждый экземпляр формы, созданные по её описанию (классу) будет их содержать. Можно в рис. 5.18 рецепт коктейля заменить на класс формы, тогда мы получим рис. 5.19.

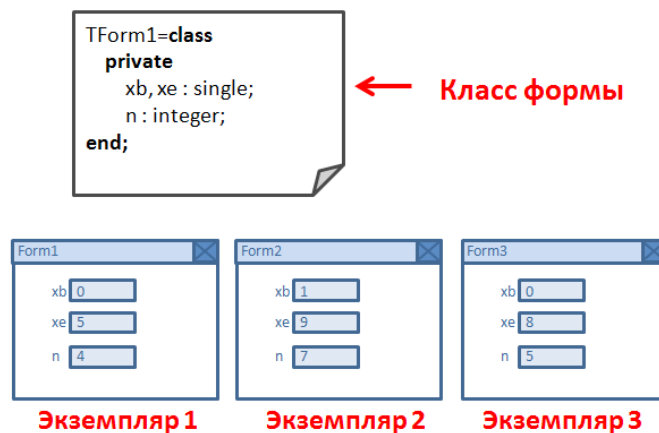


Рис. 5.19. Экземпляры формы с переменными

В классе формы `TForm1` объявлены переменные `xb`, `xe` и `n`. При создании разных экземпляров класса формы «Экземпляр 1», «Экземпляр 2» и «Экземпляр 3» у каждого из них будут свои переменные `xb`, `xe` и `n`. На рис. 5.19 показано, что значения переменных в каждом экземпляре могут быть различными, т.к. сами экземпляры также представляют собой различные объекты. Когда мы создаём объект с переменными, их значения непредсказуемы, например, случайные величины. Возможно, они будут хранить в себе нули. Нужно задавать конкретные значения переменным класса при создании объекта.

В жизни объекта-формы происходят различные события. Некоторые из них происходят по внешним причинам, например, вследствие действий пользователя. Другие происходят по внутренним причинам. Событие `OnCreate` происходит автоматически при создании экземпляра класса формы, что понятно из названия. Нашей задачей является задание правильных начальных значений переменным `xb`, `xe` и `n` при создании формы. Процедура-отклик на событие `OnCreate` является подходящей для этих действий. В проекте кликнем два раза в пустое поле строки `OnCreate` формы `Form1`. Если не получается выделить форму, т.к. она полностью закрыта располо-

женными на ней компонентами, то можно воспользоваться панелью Structure. Там форма доступна в виде самого верхнего узла дерева структуры компонентов.

Как только среда разработки сгенерировала нам процедуру отклика, мы можем ввести в неё следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  xb:= 0; // задаем начальную точку интервала
  xe:= 4; // задаем конечную точку интервала
  n:= 5; // задаем количество точек на графике
end;
```

Данная процедура принадлежит классу формы, что показано словом TForm1 с точкой перед названием процедуры FormCreate. Эта процедура запустится автоматически при создании экземпляра формы Form1. Переменные xb, xe и n, входящие в класс формы, получают заданные значения. Комментарии поясняют смысл выполняемых присвоений.

В завершении данного раздела систематизируем наши базовые знания об объектно-ориентированном программировании на примере класса и экземпляра формы в Delphi/C++Builder/RAD Studio:

- каждой форме, создаваемой в дизайнере, соответствует свой класс, название которого созвучно с названием формы: Form1 — форма, TForm1 — класс формы;
- класс формы находится в текстовом файле исходного кода программы, например, Unit1.pas;
- в классе формы перечислены визуальные компоненты — элементы интерфейса, которые расположены на форме;
- в классе формы присутствуют названия процедур-откликов на события;
- события могут происходить по внешним причинам из-за действий пользователя (например, нажатие на кнопку) или по внутренним

причинам (например, событие `OnCreate` при создании экземпляра формы);

- в описание класса формы можно добавлять нужный текст вручную;
- можно добавлять переменные в класс формы, для каждого экземпляра формы они будут свои;
- переменные нужно инициализировать, т.е. задавать им начальные значения; удобное для этого место — процедура отклика на `OnCreate`.

5.8. Дальнейшее развитие взаимодействия кода и интерфейса

В предыдущих двух разделах мы модифицировали интерфейс и код так, чтобы пользователь мог изменять интервал построения графика. Кнопки со «стрелочками» уже могут расширить или сузить поле построения. Теперь нам нужно запрограммировать кнопки с треугольниками, чтобы они могли добавлять или удалять точку к графику. Решим сначала задачу по добавлению дополнительной точки к графику справа при нажатии на кнопку с «треугольником» (рис. 5.20).

Выберем нужную кнопку, перейдём в `Object Inspector`, найдем событие `OnClick` и сгенерируем при помощи среды разработки процедуру отклика. Нам предстоит перестроить график функции, добавив ещё одну точку справа. Для этого нам выполним ряд шагов:

1. удалим точки текущего графика;
2. увеличим интервал построения графика (переменная `xe`);
3. увеличим количество точек (переменная `n`);
4. перестроим график заново с новыми значениями переменных класса формы с увеличенным количеством точек.

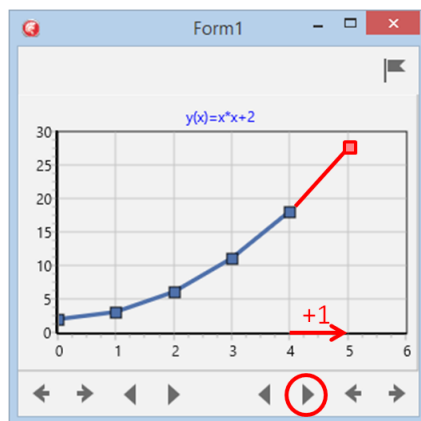


Рис. 5.20. Добавление точки справа

Если мы просто увеличим количество точек за счет увеличения значения переменной n , а интервал построения останется прежним, то график станет «мельче». Если мы увеличим интервал, но оставим прежнее количество точек, то график «растянется». Но нам нужно увеличить и интервал построения, и количество точек, как показано в коде ниже:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  Chart1.Series[0].Clear; // удаляем точки линии
  n:= n + 1; // увеличиваем кол-во точек
  xe:= xe + 1; // увеличиваем интервал
  SpeedButton1Click(Sender); // перестраиваем график
end;
```

Разберём код выше. В первой строчке процедуры мы очищаем линию графика, теперь в ней больше нет точек. Если запустить приложение только с этой строчкой, то после нажатия на кнопку линия графика исчезнет. Но мы не будем останавливаться на этом и увеличим значения переменных n и xe на единицу. Затем нам нужно выполнить построение графика «как-будто в первый раз», вызовом процедуры-отклика `SpeedButton1Click(Sender)`. Обычно эта процедура выполняется автоматически при нажатии пользо-

вателем кнопки SpeedButton1, но она входит в состав класса, и мы можем вызывать её программно.

Перед проверкой корректности работы введенного кода произведём дополнительную настройку графика. Двойным щелчком зайдём в мастер настройки графика и увеличим максимальное значение левой вертикальной оси, как показано на рис. 5.21. Сохраним, запустим проект на исполнение и проведём ряд экспериментов с построением графика. Сценарий проверки работоспособности приложения таков:

1. запустить приложение;
2. три раза последовательно нажать на кнопку увеличения интервала — самая правая кнопка со «стрелочкой» ;
3. три раза последовательно нажать на кнопку добавления точки на график — кнопка с «треугольничком»;
4. сравнить результат с рис. 5.22;
5. завершить работу с приложением.

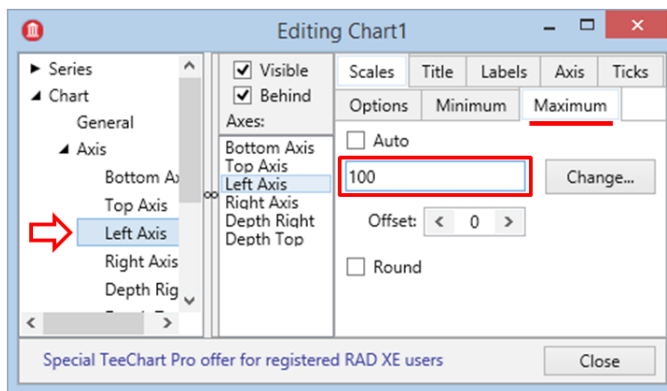


Рис. 5.21. Увеличение диапазона по высоте

Если в дальнейшем потребуется автоматически подбирать высоту, то нужно поставить «галочку» Automatic в окне мастера настройки компонента.

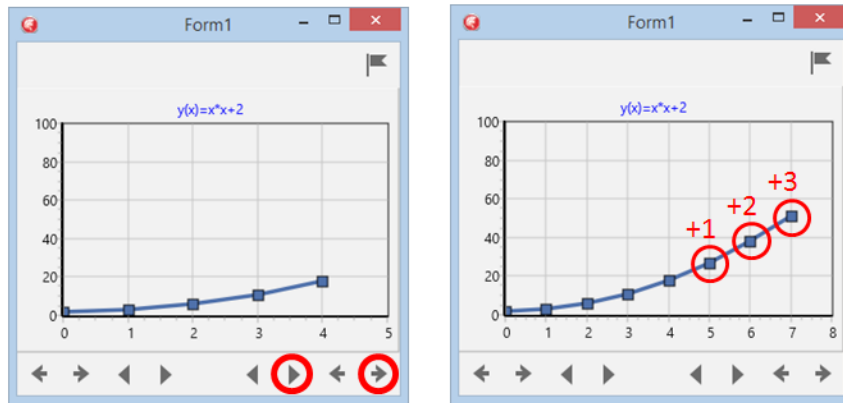


Рис. 5.22. Расширение интервала и добавление точек

Осталось доработать проект до конца. Этой доработкой станет реализация процедур отклика для оставшиеся кнопки нижней инструментальной панели. Сделайте это в самостоятельном режиме. Готовое приложение должно позволять получать результат, как показано на рис. 5.23.

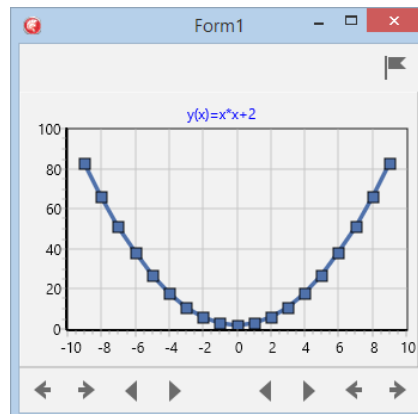


Рис. 5.23. Полный вид параболы

С помощью такого готового приложения можно анализировать более сложные графики функций, вид которых, в отличие от рассмотренной нами параболы, совсем неочевиден. Если мы сможем изменять вид уравнения

в приложении, то мы сможем изучать графики различных функций, становясь экспертами в области математического анализа. Также мы можем помочь друзьям и одноклассникам разобраться в этой интересной, но сложной теме.

Сейчас мы проведём значительную доработку интерфейса, поэтому сначала создадим рабочую копию проекта. Для этого выполним ряд действий:

- закроем среду разработки Delphi/C++Builder/RAD Studio, если она была открыта;
- создадим папку под названием «Project 5.2»;
- скопируем все содержимое папки «Project 5.1» (текущий проект) в нее;
- запустим среду разработки и откроем проект из папки «Project 5.2».

Данный приём — хороший способ сохранить предыдущую версию проекта. Каждый раз, когда мы планируем масштабные изменения, универсальным способом является копирование всего содержимого папки. Такой метод является самым простым. Главный его недостаток заключается в следующем. Если мы скопируем проект, а потом испортим текущую версию, то «откатиться назад» мы сможем целиком. Соответственно, пропадут все изменения, сделанные с момента копирования содержимого папки.

Однако есть и другие, более развитые способы управления исходным кодом без необходимости манипуляций в «Проводнике». Например, в саму среду разработки встроен сервис, позволяющий сравнивать различные версии исходного кода в одном проекте. Эти версии создаются автоматически в момент сохранения изменений в исходном коде. Как показано на рис. 5.24, если внизу окна среды разработки найти ярлычок «History» (история изменений), то можно переключиться в режим сравнения различных версий. С помощью данного сервиса можно найти и сравнить, что произошло на различных этапах работы с кодом между сохранениями.

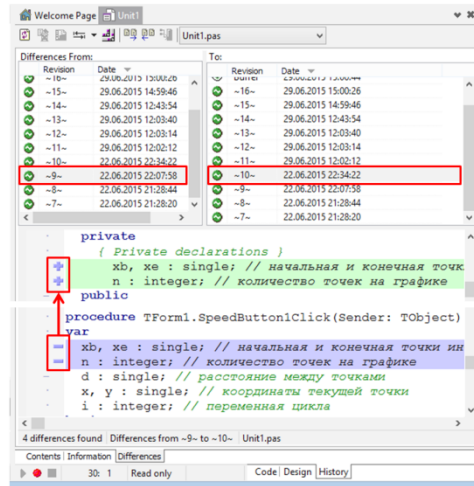


Рис. 5.24. Сервис сравнения вариантов исходного кода

Но вернёмся на то место, где мы создали новую папку, скопировали в неё весь исходный код и открыли проект. Переключите нижние ярлычки с «History» на «Code». Выполним ряд действий по расширению интерфейса:

- выделим компонент Chart1 и свойство Align установим в None;
- увеличим размер главного окна Form1, потянув его границу вправо;
- щёлкнем на пустом пространстве между графиком и правой границей формы разместите компонент TLayout;
- разместим четыре компонента TLabel и три TSpinBox на TLayout;
- установим свойство Align компоненту TLayout установить в значение Right;
- выберем компонент Chart1 и его свойству Align установим значение Client.

Проверим корректность размещения компонентов на TLayout по рис. 5.25. У самой верхней метки нужно изменить свойство Text так, чтобы она показывала уравнение параболы с коэффициентами a, b и c, соответственно. Свойства Text трёх остальных меток также настроим согласно рис. 5.25. Свойство TextSettings.Font верхней метки также нужно изменить,

щёлкнув на кнопку с тремя точками [...] в Object Inspector. Сделаем так, чтоб шрифт метки стал «жирным».

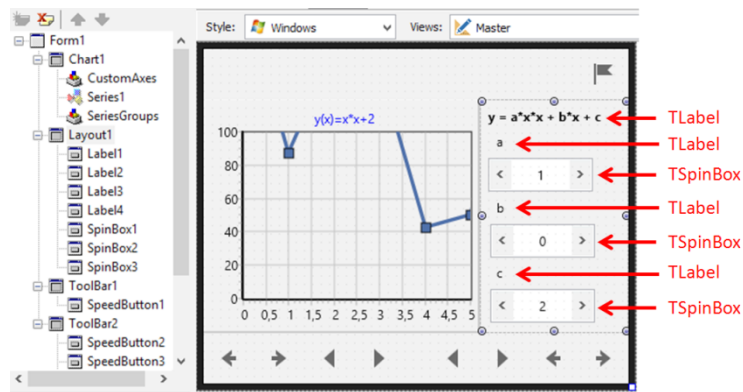


Рис. 5.25. Улучшенный интерфейс для анализа графиков

Изучим свойства компонента TSpinBox:

Свойство	Назначение
Increment	Определяет, с каким шагом изменяется значение в большую или в меньшую стороны при нажатии на кнопки [<] или [>]
Minimum	Минимальное значение
Maximum	Максимальное значение
Text	Отображаемое значение

Поскольку изначальный вид функции $y(x)=x^2+2$, что есть $y(x)=1*x^2+0*x+2$, то и изменить свойство Text для каждого компонент TSpinBox нужно соответствующим образом. Не забудем свойства Min и Max для каждого их компонентов задать как -10 и 10. Такого диапазона изменений будет достаточно. Свойство Increment по умолчанию равно 1. При нажатии маленьких кнопок со стрелочками компонента TSpinBox, хранимое в компоненте значение будет увеличиваться или уменьшаться на 1 (Increment).

Остаётся сделать так, чтобы на изменение значение коэффициентов в уравнении параболы при помощи компонентов TSpinBox происходи-

ло автоматическое перестроение графика. Выберем все три компонента TSpinBox, поочерёдно кликая на них и удерживая клавишу Shift. Перейдём в Object Inspector и найдём событие OnChange. Кликнем дважды в пустом поле рядом с этой надписью, а в сгенерированной процедуре отклика пока поставим две косые черты — признак комментария:

```
procedure TForm1.SpinBox1Change(Sender: TObject);  
begin  
  //  
end;
```

Поставив две косые черты, мы обозначили наши намерения реализовать код по отклику на событие OnChange для всех трёх компонентов, которое случится всякий раз, когда пользователь изменит значение в SpinBox1, SpinBox2 или SpinBox3. Если бы мы не поставили две косые черты, то процедура с пустым телом была бы автоматически удалена средой разработки всякий раз, когда мы сохраняли бы код. Теперь можно сохранять проект не опасаясь, что «пустая» процедура будет удалена.

Составим план модификации кода:

1. добавим в функцию MyFunction параметры a, b и c; изменим функцию так, чтобы она могла вычислять значения для произвольных значений коэффициентов;
2. добавим в класс формы переменные a, b и c;
3. добавим инициализацию новых переменных в событии формы OnCreate;
4. изменим процедуру запуска построения графика SpeedButton1Click;
5. доделаем процедуру отклика на событие изменение коэффициентов — OnChange для компонентов TSpinBox.

Пункт № 1. Поскольку мы будем изменять коэффициенты в уравнении, то и наша главная расчётная функция MyFunction должна принимать в ка-

честве параметров не только значение x , но и коэффициентов a , b и c . Найдём функцию в коде и изменим её следующим образом:

Было
<pre>function MyFunction(xi: single): single; var yi: single; begin yi:= xi*xi + 2; result:= yi; end;</pre>
Стало
<pre>function MyFunction(xi, ai, bi, ci: single): single; var yi: single; begin yi:= ai*xi*xi +bi*xi + ci; result:= yi; end;</pre>

В списке параметров функции добавились ai , bi , ci , через них будут передаваться значения соответствующих коэффициентов. Строка с расчётом y_i также подверглась изменению, теперь значение функции вычисляется согласно полной формуле со всеми коэффициентами.

Пункт № 2. Мы уже проделывали добавление xb , xe и n в класс формы, см. раздел 5.6. То же самое нужно сделать и с переменными a , b , c . Найдём в классе формы `TForm1` раздел `private` и добавим туда строки кода сразу после xb , xe и n :

```
Private
{Private declarations}
xb, xe: single; // начальная и конечная точки интервала
n: integer; // количество точек на графике
a, b, c: single; // коэффициенты в уравнении
```

Пункт № 3. В процедуре отклика на событие `OnCreate` формы удобно инициализировать значения переменных, т.е. задать им начальные значе-

ния. Найдём в коде соответствующую процедуру отклика и добавим инициализацию для a , b и c :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    xb:= 0; // задаем начальную точку интервала
    xe:= 4; // задаем конечную точку интервала
    n:= 5; // задаем количество точек на графике
    a:= 1;
    b:= 0;
    c:= 2;
end;
```

Пункт № 4. SpeedButton1Click или процедура отклика на нажатие кнопки «с флажком» тоже требует изменений. Теперь она должна вызывать расчёт функции MyFunction с учётом значений a , b и c :

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
var
    d: single; // расстояние между точками
    x, y: single; // координаты текущей точки
    i: integer; // переменная цикла
begin
    d:= (xe - xb) // (n - 1); // вычисляем расстояние между
        точками
    for i:= 1 to n do // тело цикла выполняется n раз
    begin
        x:= xb + d * (i - 1); // вычисляем x для i-ой точки
        y:= MyFunction(x, a, b, c); // вычисляем y для i-ой точки
        Chart1.Series[0].AddXY(x, y); // добавляем i-ю точку
    end;
end;
```

Здесь изменениям подверглась только строка с вызовом MyFunction: в вызов добавились дополнительные параметры.

Пункт № 5. Вернёмся к процедуре SpinBox1Change, куда мы добавили лишь две косые черты. Напишем полный текст данной процедуры:

```

procedure TForm1.SpinBox1Change(Sender: TObject);
begin
    a:= SpinBox1.Value; // новое значение a
    b:= SpinBox2.Value; // новое значение b
    c:= SpinBox3.Value; // новое значение c
    Chart1.Series[0].Clear; // удаляем точки линии
    SpeedButton1.OnClick(Sender); // перестраиваем график
end;

```

Расширим диапазон поля построения графика. Дважды щёлкнем на компоненте Chart1, перейдем в мастере настройки графика в раздел Axis (оси), выберем Left (левая) и ей установим свойство Minimum в значение -100.

Мы уже почти готовы не только проверить работоспособность приложения, но и провести анализ поведения графика параболы при изменении коэффициентов. Сохраним и запустим приложение. Нажмём кнопку с «флажком» для построения графика. Затем при помощи кнопок для расширения интервала построения графика добъёмся результата, показанного на рис. 5.26.

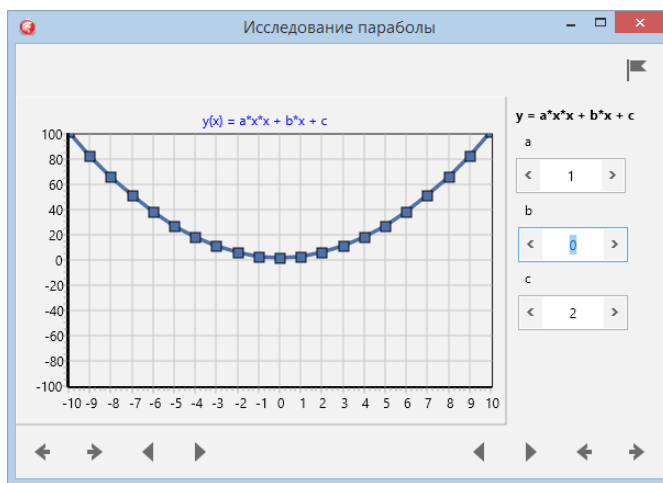


Рис. 5.26. Финальный вид формы с графиком

Теперь можно выполнить исследовательскую часть нашего проекта. Объектом исследования будет парабола, а предметом – её форма и вид при изменении коэффициентов в формуле: $y(x) = a \cdot x^2 + b \cdot x + c$. Такое упражнение несомненно углубит и расширит ваши математические познания. Потренируйтесь сами, ответив на следующие вопросы:

1. Что происходит с графиком функции при увеличении коэффициента a ?
2. Что происходит с графиком функции при уменьшении коэффициента a ?
3. Что происходит с графиком функции при увеличении коэффициента b ?
4. Что происходит с графиком функции при уменьшении коэффициента b ?
5. Что происходит с графиком функции при увеличении коэффициента c ?
6. Что происходит с графиком функции при уменьшении коэффициента c ?

Согласитесь, без такого инструмента, как наше приложение, достаточно сложно быстро ответить на данные вопросы. Чтобы пойти дальше в освоении Delphi/C++Builder/RAD Studio, а также в математическом анализе, предлагаем подумать или реализовать следующие дополнительные возможности:

- Добавить компонент TSwitch, при помощи которого можно включать/выключать автоматическую настройку интервала поля графика — свойство Chart1.Axes[2].Automatic.
- Добавить кнопку [+], при помощи которой можно добавлять новую линию графика. Например, построить один график, добавить другой и уже на нём выполнять исследования, оставив первый в качестве эталонного.
- Изменить код приложения так, чтобы строился график гиперболы, заданной уравнением: $y(x) = a \cdot x^2 + b \cdot x + c \cdot x + d$. Провести исследования поведения данного графика при изменении коэффициентов.

Рассмотренный проект целесообразно выполнять группой, где роли можно распределить следующим образом:

- Программист — разрабатывает проект;
- Математик — подбирает функции, выполняет ручной просчёт точек для тестирования (или при помощи Microsoft Excel);
- Математик-исследователь — составляет план исследований поведения графиков функций, разрабатывает методическое пособие по выполнению заданий при помощи разработанного приложения.

Слаженная работа указанных трёх членов команды позволит не просто создать хорошее учебное приложение, но и провести ряд интересных и полезных исследований. Для развития сбалансированных знаний у всех участников можно выполнять смену ролей. Например, для проекта по исследованию парабол назначается определенные роли конкретным участникам. После окончания первого проекта тут же стартует другой проект, где создаётся проект для изучения гипербол. Потом выполняется проект по изучению квадратичных парабол и т.д. Смена ролей при групповой работе позволяет долговременно поддерживать интерес к выполнению одинаковых проектов.

Дневник наблюдений

6.1. Постановка задачи

Одним из методов получения естественнонаучных знаний является наблюдение за природными явлениями и проведение экспериментов. Уже в школе весьма полезно уметь пользоваться данным методом, поэтому учащимся обязательно выдают подобные задания. Например, наблюдение за погодой. В порядке эксперимента многие занимались проращиванием семян. Рост химических кристаллов, колебание маятника — многие явления изучались методом наблюдений. Чтобы наблюдения имели научный характер, результаты наблюдений нужно записывать периодически и аккуратно.

Для аккуратной и регулярной записи наблюдаемого явления мы создадим небольшое приложение. Оно будет хорошо работать на мобильном устройстве, что сделает её особенно полезной. Если мы проводим измерения температуры воды в ближайшей реке, уровень выпавших осадков во дворе дома или скорость роста веток дерева по дороге из школы, то хорошо бы всегда с собой иметь некую «записную книжку». Можно купить блокнотик, можно делать записи на салфетке из-под «хот-дога», можно запомнить данные в уме... Но хорошим решением будет использовать свой смартфон.

Существуют различные мобильные приложения, в которых можно делать записи. Можно даже воспользоваться и обычным приложением «календарь». Но такую программу можно сделать самим, добавив в приложение уникальные функции, которых нет в уже готовых. Это и удобный формат хранения данных для последующей обработки, к примеру, в специальных математических пакетах. Это и заданный перечень наблюдаемых параметров. Также можно организовать групповые наблюдения, например, уровня осадков в различных частях города там, где живут ваши одноклассники.

Данные будут поступать одновременно от всех учеников в единую базу. В конечном итоге, даже просто удобное расположение кнопок, которое вы сделали сами под ваши требования, и есть та уникальность, которая определяет полезность приложения.

Если проанализировать современные мобильные приложения, то очень сложно найти действительно неповторимое. Практически все приложения имеют аналоги. Уже практически всё придумано и всё реализовано, причём не одной компанией, а целым множеством. Но это не означает, что нельзя сделать что-то полезное. Просто нужно найти какие-то уникальные задачи и решить при помощи мобильного приложения, максимально точно проработав мельчайшие детали. На момент написания данной книги в Google Play можно было легко найти много приложений типа «дневник» (diary). Они позволяют вести дневник для худеющих, занимающихся спортом, растящих младенцев. Также есть множество приложений, для записи имён понравившихся девочек или мальчиков, названий вкусных пирожных, пород милейших щенков, авторов любимых стихов и т.д. Конечно, есть серьёзные приложения типа «дневник роста растений» или «дневник рыбака». Но никто кроме вас не создаст приложение, идеально подходящее для вашей задачи.

При таком подходе важно правильно определить объект и предмет наблюдений. Если для вас «объект» исследований и «предмет» исследований звучит синонимично, то давайте разберёмся в этом на примере. Объект исследования — росток фасоли. Предмет — скорость роста. Помимо длины ростка мы можем делать записи о количестве листков и отростков, их цвета, толщине и т.п. В любой момент мы должны также уметь выгрузить данные из приложения для дальнейшей обработки или подготовки отчёта об исследованиях, например, в форме презентации. Теперь, когда постановка задачи сделана, пусть и в достаточно свободной форме, можно приступать к её реализации.

6.2. Прототип приложения

Прототип — это некий набросок того приложения, которое мы должны сделать. Можно сделать прототип в форме рисунка от руки на листке, вырванном из рабочей тетради. Можно создать его в специальной программе класса «app mockup tool» (инструмент прототипирования приложений). Но среда Delphi/C++Builder/RAD Studio настолько проста и легка в использовании для создания интерфейсов мобильных приложений, что есть смысл сделать прототип непосредственно здесь. Дальше мы сможем преобразовать эту «поделку» в реальное приложение, сэкономив время.

Запустим среду разработки Delphi/C++Builder/RAD Studio. При помощи главного меню создадим новый проект File->New->Multi-Device Application — Delphi. В появившемся мастере создания проектов выберем Blank Application (рис. 6.1).

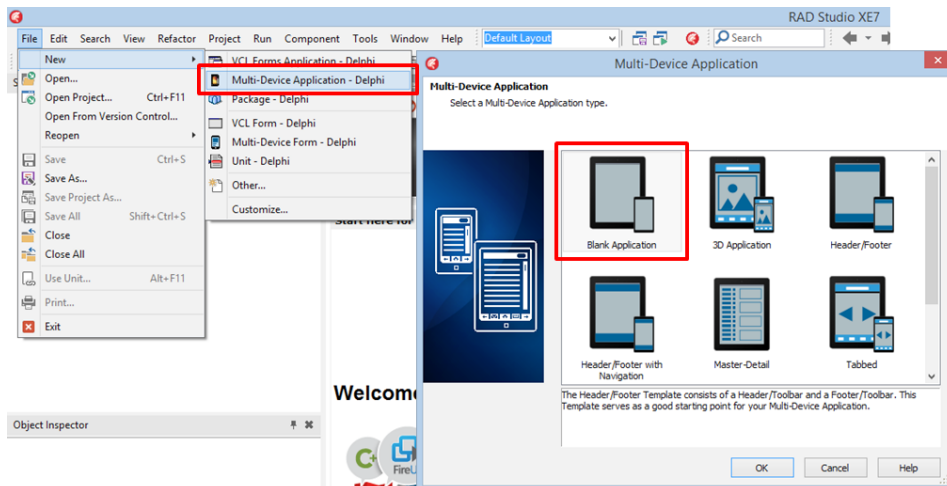


Рис. 6.1. Создание нового мульти-платформенного приложения

Можно было выбрать что-то иное, а не Blank Application. Готовые прототипы, следующие за Blank Application, весьма полезны для изучения и построения на их основе полнофункциональных приложений. Однако к теку-

щей главе вы уже достаточно опытны, чтобы самостоятельно разработать прототип из отдельных компонентов из палитры. Создайте папку «Project 6» и сохраните туда проект, выбрав File->Save All. Добавим на форму необходимые компоненты для интерфейса, обращая внимание на значение ключевых свойств:

1. Добавим TTabControl; Align -> Client; TabPosition -> None.
2. Кликнем правой кнопкой мыши на TTabControl, затем выберем из всплывающего меню Add TTabItem; продедем это 3 раза;
3. Выберем в Structure первую «страничку» TabItem1, добавим на неё TToolBar; Align -> Top.

Полученный макет интерфейса настолько тривиален, что пока нет смысла его иллюстрировать. У нас есть TTabControl с тремя «страничками». Первая страничка будет нам нужна для отображения списка с текущими наблюдениями. Вторая — для детального просмотра записи о наблюдении. Третья — для добавления новой записи. Но сначала нам нужно обеспечить навигацию по этим страничкам. На верхнюю инструментальную панель TToolBar первой «странички» разместим «быструю» кнопку TSpeedButton. Свойство Align зададим как MostRight. Она будет служить для добавления новой записи о наблюдении. Выберем свойство StyleLookup из выпадающего списка как «addtoolbutton». При нажатии на неё мы должны попадать на третью страничку, где будет происходить добавление новой записи. Сделаем навигацию — переход на нужную страничку — при помощи уже опробованного компонента TActionList.

Добавим на форму TActionList, щелкнем на него два раза, в появившемся мастере выберем New->New Standard Action (рис. 6.2). Из списка стандартных действий выберем TChangeTabAction. В Object Inspector настроим данное действие так, чтобы оно вело нас на третью «страничку»: свойству Tab из списка выберем значение TabItem3. Кнопке [+] в качестве Action выберем тоже из списка ChangeTabAction1. Сохраним и запустим приложение.

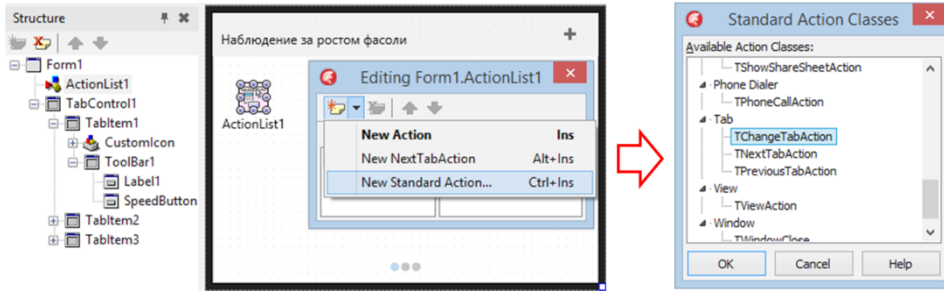


Рис. 6.2. Добавление нового стандартного действия

Если мы нажмём на кнопку с «плюсиком», то первая страничка сменится на третью в режиме «слайд-шоу». Если мы будем добавлять новую запись, то этот процесс может закончиться двумя вариантами. Пользователь может отказаться от сохранения новой записи или подтвердить её сохранение. В любом случае мы должны вернуться на первую «страничку». В визуальном дизайнере перейдём на третью страничку, нажав третий серый кружочек в нижней части формы. Также это можно сделать, выбрав `TabControl1`, а затем в `Object Inspector` свойство `ActiveTab` установить в `TabItem3` при помощи выпадающего списка.

На третьей страничке тоже разместим её собственный `TToolBar`, который, подобно хорошо надутому воздушному шару, взлетит вверх и приклеится к потолку. Здесь у нас будут три управляющих кнопки `TSpeedButton`. Первая со свойством `Align->MostLeft`, `StyleLookup->backtoolbutton`, а `Text->«Назад»`. Чтобы кнопка заработала, создадим ей нужное действие. Это можно сделать для данной кнопки непосредственно в `Object Inspector`, без привлечения `ActionList1`. Просто найдём свойство `Action` у кнопки, откинем вниз стрелкой выпадающий список, где выберем `New Standard Action->Tab->TChangeTabAction`. Потом раскроем свойства «действия» кнопкой `[+]` перед словом `Action`, где выберем нужный `Tab` и введём правильный `CustomText` (рис. 6.3). А теперь найдём свойство `Shortcut` и установим в значение из списка `HardwareBack` (оно будет последним). Теперь это действие будет срабатывать и от соответствующей аппаратной кнопки на платформе Android. Сохраним, запустим и проверим работоспособность

навигации между первой и третьей страничкой. Не забудьте переключить TabControl1 в исходное положение, выбрав в качестве активной первую «страничку» TabItem1.

Действие «<Назад» будет означать, что пользователь не хочет сохранять введённые данные. Добавим ещё две «быстрые» кнопки на инструментальную панель третьей «странички», задав первой из них свойство Text как «Сохранить», а второй «Отменить». Свойства Align обеим кнопкам выберем как MostRight. Далее для кнопки «Отменить» назначим действие Action как ChangeTabAction2.

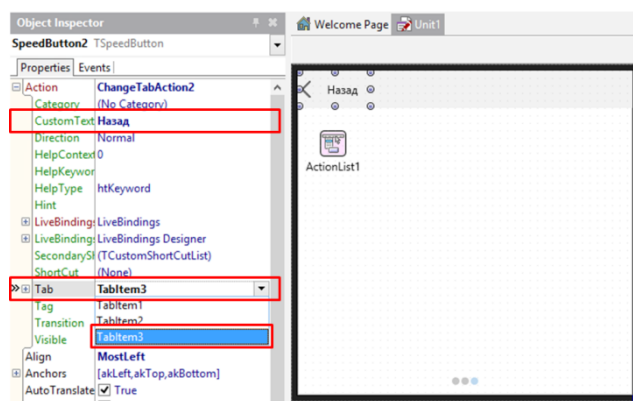


Рис. 6.3. Добавление действия для кнопки

С кнопкой «Сохранить» более интересная история. По факту мы должны также вернуться на первую страничку, но предварительно выполнив действия по добавлению описания наблюдения в список. Пока сделаем так: дважды щёлкнем на кнопку «Сохранить», перейдём в процедуру отклика, а там поставим «заглушку» с возвращением на первую страничку:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);  
begin  
  ShowMessage('запись сохранена');  
  ChangeTabAction2.ExecuteTarget(TabControl1);  
end;
```

«Заглушкой» называют некий код, который вместо реальных действий просто выполняет некую формальную демонстрацию активности (в нашем случае — показывает окошко с надписью). Вторую строчку тела процедуры понять не так просто. Но пока вполне можно удовлетвориться объяснением, что выполняется стандартное действие по смене «страничек» относительно целевого компонента `TabControl1`. Главное, что мы понимаем общий смысл данной строки кода: вызов созданного нами стандартного действия программно, а не в результате непосредственного нажатия на кнопку. Далее после создания структуры данных и ввода нужных алгоритмов их обработки мы прокомментируем демонстрацию окошка с надписью «Запись сохранена». Сейчас эта заглушка нужна нам для наглядности.

Продолжаем изготовление прототипа приложения. Вернёмся на первую «страничку» компонента `TabControl1` и разместим на неё компонент `TListBox`. Данный компонент является принципиально важным для мобильной разработки, т.к. большинство интерфейсов приложений имеют вид списка. «Контакты», «Сообщения», «Телефон» и т.д. выглядят как списки. Наше приложение также будет использовать компонент «список» `TListBox` в качестве главного. Для начинающих программистов очень важно не пытаться проявить ненужное творчество, а сделать приложение, похожее на уже существующие. Чем больше ваше приложение будет походить на наиболее популярные, тем проще пользователю будет к ним привыкнуть и пользоваться ими без ошибок.

Здесь возникает уместный вопрос. А если моё приложение будет слишком похожим на другие, то как мне завоевать новых пользователей? В чём будет уникальность именно моего приложения? Ответ достаточно прост: уникальность приложения достигается его функциональностью, т.е. набором новых и желательно неповторимых возможностей. Интерфейс важен, но только с позиции удобства использования. Также нужно учесть, что для начинающих разработчиков нужно не «закопаться» в реализации интерфейсов, которые могут быть созданы за счёт стандартных компонентов. `TListBox` — весьма мощный и универсальный компонент, поэтому лучше по-

тратить немного времени и разобраться с его настройками, чем пытаться выдумать что-то своё.

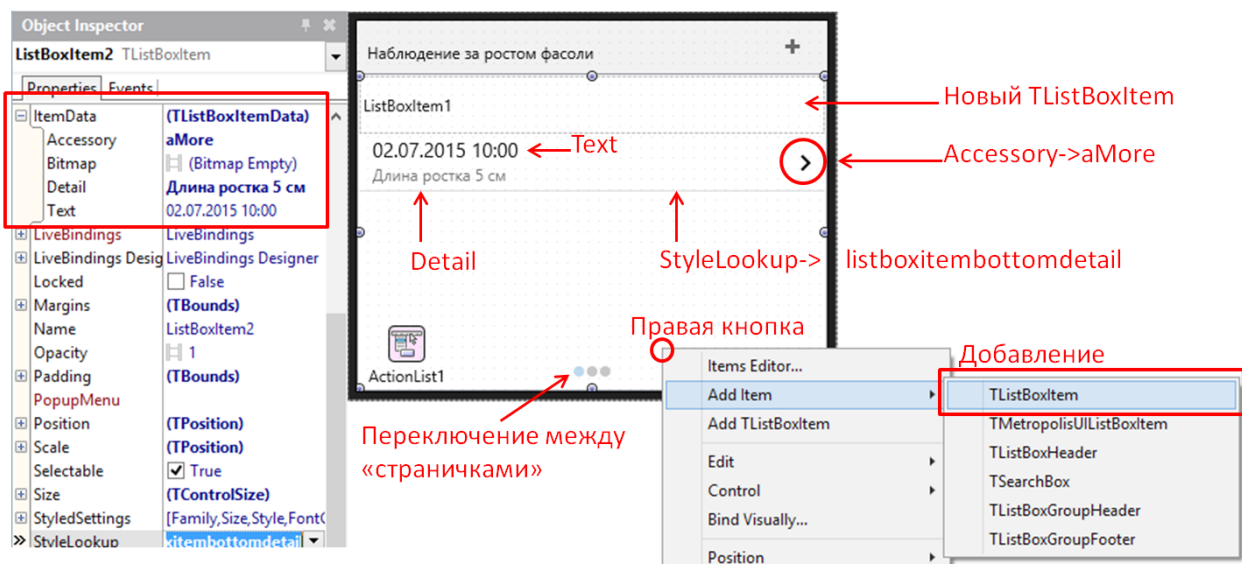


Рис. 6.4. Настройка списка

Внимательно изучим рис. 6.4. В центре внимания на форме находится «компонент-список», краеугольный камень современного интерфейса многих мобильных приложений. Очевидно, что список должен состоять и элементов списка. Чтобы добавить новый элемент, нужно кликнуть правой кнопкой мыши и в всплывающем меню выбрать пункт «Add Item», за которым опять откроется целый список возможных вариантов. Кратко перечислим их назначение:

Пункт всплывающего меню 2-го уровня	Описание
TListBoxItem	Универсальный элемент списка, который потом настраивается в Object Inspector множеством способов. Главные свойства — StyleLookup и ItemData
TMetropolisUIMListBoxItem	Элемент списка в стиле «метрополис». Был введён для стилизации интерфейсов под Windows 8. В мобильной разработке под Android не используется
TListBoxHeader	Формирует отдельный верхний элемент-заголовок списка
TSearchBox	Создаёт отдельный элемент списка для выполнения поиска, например, по списку контактов и т.д.
TListBoxGroupHeader	Если в списке нужно создать отдельную группу, то данный элемент будет заголовком группы
TListBoxGroupFooter	Если в списке нужно создать отдельную группу, то данный элемент будет «подвалом» или признаком завершения

В нашем случае мы будем добавлять обычные TListBoxItem с последующей настройкой их в Object Inspector. Когда мы добавляем универсальный элемент, он появляется в довольно простом виде, как показано на рис. 6.4 с меткой «Новый TListBoxItem». Дальше его внешний вид определяет свойство StyleLookup. Данное свойство позволяет одним действием применить к элементу целый шаблон оформления. Мы выбираем из списка для данного свойства значение listboxitembottomdetail (элемент списка с детализацией в низу). Можно поэкспериментировать с различными значениями данного свойства из списка. В основном, они отличаются лишь форматированием. Не забудем свойство Height элемента установить равным 44.

Стиль элемента списка с детализацией подразумевает, что у нас в элементе может находиться «главный» текст и «вспомогательный». Главный текст отображается шрифтом более крупного размера. Содержание текста определяет свойство Text. Свойство Detail содержит мелкий текст поясняющего характера. Свойство Accessory (аксессуар) тоже можно выбрать из списка. Мы воспользуемся значением aMore, что добавит иконку в виде треугольной скобки. Она будет означать возможность «кликнуть» на элементе списка, что приведёт к отображению более подробной информации о данном элементе списка. На рис. 6.4 второй элемент списка уже полностью настроен, тогда как первый отображает только что добавленный эле-

мент. Поменять порядок следование элементов списка можно при помощи опции «Items Editor...» контекстного меню.

Выбор правильного стиля для дизайна элементов списка является достаточно творческой задачей, т.к. нет готовых рекомендаций. Нужно пробовать различные варианты, а критерием истинности является визуальное восприятие. Изменяя стиль в среде разработки, можно создать несколько видов интерфейса, которые легко сравнить между собой (рис. 6.5). Желательно, чтобы список содержал хотя бы минимальное число элементов.

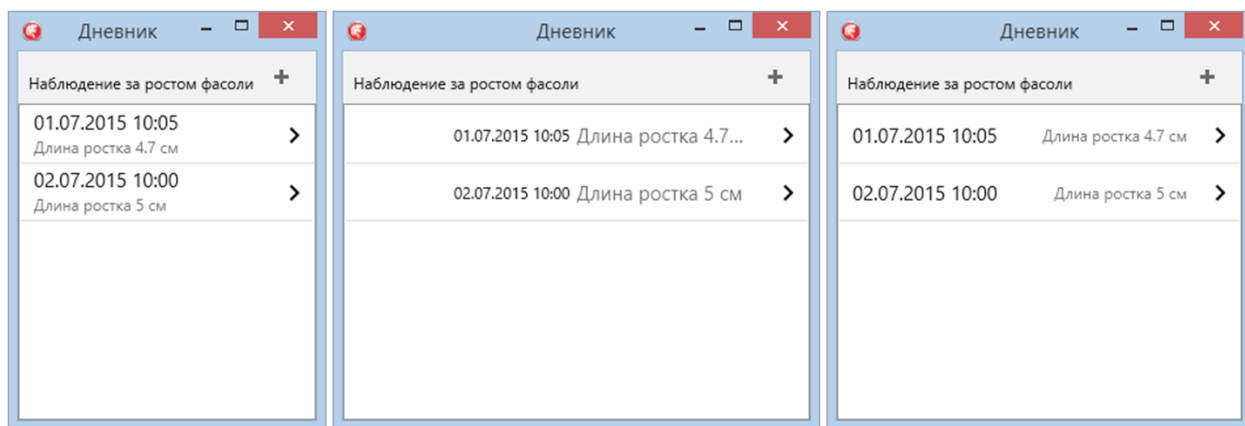


Рис. 6.5. *Различные варианты стиля списка с детализацией*

Как видно на рис. 6.5 для дневника наблюдений хорошо подходит первый и третий варианты со значениями свойства `StyleLookup` элементов списка `listboxitemrightdetail` и `listboxitembottomdetail`, соответственно. Delphi/C++Builder/RAD Studio позволяет создавать универсальные приложения, автоматически масштабирующие свой интерфейс под различные размеры экраны мобильного устройства. Но мы изначально решили, что дневник наблюдений более полезен на смартфоне, чем на планшете. Всё-таки смартфон находится большее количество времени в непосредственной досягаемости от пользователя. Тогда стиль с более компактным размещением информации в элементе списка, а это — первый вариант на рис. 6.5, явля-

ется самым подходящим. Если вы реально проделывали эксперимент со свойством `StyleLookup`, то верните его значение в `listboxitembottomdetail`.

Разберёмся с навигацией, когда пользователь кликает на элемент списка с тем, чтобы получить более детальную информацию. Детальная информация будет содержаться на 2-й «страничке» компонента `PageControl1`, поэтому мы должны обеспечить переход именно на неё. Как только нам нужна какая-либо навигация по страничкам компонента `TPageControl`, мы тут же вспоминаем о компоненте `TActionList` и стандартном действии `TChangeTabAction`. Щёлкаем два раза на `TActionList` и добавляем соответствующее новое стандартное действие. В качестве целевой «странички» или `Tab`-а мы выбираем в `Object Inspector` `TabItem2`. Всякий раз, когда будет вызываться данное действие, переход будет осуществляться на вторую «страничку» с подробной информацией о наблюдении.

Итак, пользователь просматривает список записей о наблюдениях. Потом кликает в какой-либо элемент списка. Приложение должно обработать событие `OnClick` компонента `ListBox1`. Ищем нужную строчку на закладке `Event` в `Object Inspector`, щелкаем два раза и попадаем в сгенерированную процедуру отклика на событие. Запишем следующий код:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  ChangeTabAction3.ExecuteTarget(TabControl1);
end;
```

Мы видим уже знакомый вызов метода `ExecuteTarget` для только что созданного стандартного «действия». Сохраняем, запускаем проект на исполнение. Убеждаемся, что при клике на одном из элементов списка автоматически появляется пустая вторая страничка. Вернёмся в среду разработки и минимальным образом улучшим вторую страничку. Как и в случае с третьей страничкой добавим инструментальную панель `TToolBar`. Как и в случае с третьей страничкой нам нужно обеспечить кнопку для возвращения на первую страничку. Можно начать всё заново, добавить кнопку `TSpeedButton`, добавить стандартное действие `TChangeTabItem` и связать

кнопку с действием. Но можно просто перейти на третью страничку, скопировать кнопку «<Назад» в буфер обмена, вернуться на вторую и вставить её из буфера на инструментальную панель. Поскольку и внешний вид, и функционал кнопок для этих двух страничек идентичен, можно воспользоваться простой операцией копирования-вставки. Сохраняем и запускаем проект на исполнение. В результате у нас должен появиться собранный прототип с реализованной функцией навигации по страничкам (рис. 6.6).

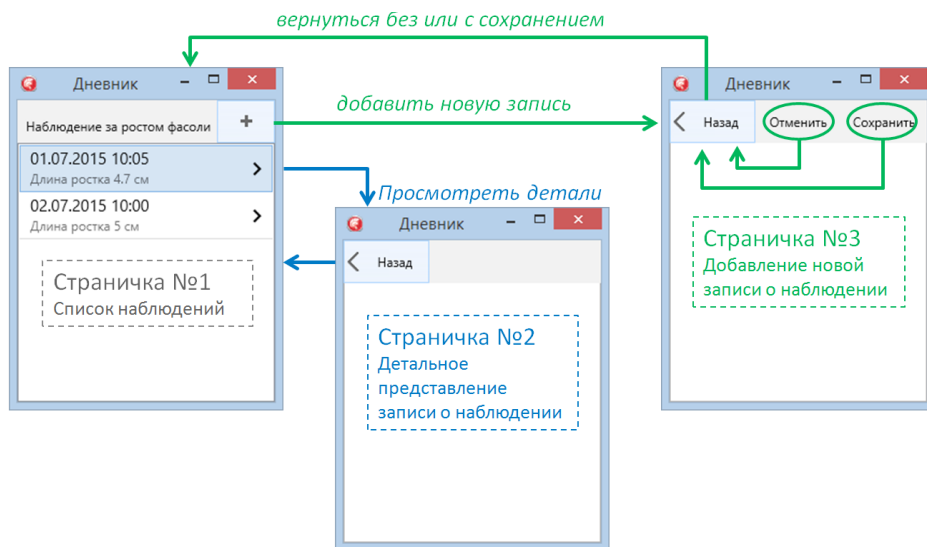


Рис. 6.6. Схема навигации между страничками

Вспомним особый случай навигации со странички № 3 на страничку № 1 при нажатии на кнопку «Сохранить». Переход на первую страничку будет происходить не автоматически, т.к. сначала будут выполнены действия по сохранению, а только потом переход. Схема навигации между элементами TPageControl показана на рис. 6.6. Если приложение требует тщательной проработки схемы навигации между страничками, то можно сначала нарисовать схему навигации, подобно указанной выше.

6.3. Структура данных

При создании приложений одним из самых важных аспектов является проектирование структур данных. То, что мы видим на интерфейсе пользователя, не является самым главным. Главное — это то, как информация хранится в памяти приложения. На интерфейс можно вывести данные о наблюдениях в строчку, в столбик, списком, поочередно, бегущей строкой, мигающими надписями и даже в виде отдельных летающих букв. Конечно, для дневника наблюдений подойдут только два способа: 1) в общем списке на первой страничке; 2) детально для каждой записи на второй страничке. Но в памяти приложения данные должны храниться в специальных структурах, не зависящих от интерфейса.

Хорошим пояснением данного подхода является система расположения товаров в магазине спортивных товаров. Непосредственно в магазине на витринах товары расположены так, чтобы посетителям и потенциальным покупателям было удобно их рассмотреть во всей красе. Нужно, чтобы у них появилось желание их взять, примерить, получить дополнительную информацию и, конечно купить. Очень важную роль здесь имеет дизайн витрины, схема выкладки, оформление и т.д. В противовес этому на складе товары размещены согласно совсем другим принципам. Здесь важны компактность и структурированность, чтобы складской персонал или сотрудники магазина в любой момент очень быстро могли найти нужный товар. Нетерпеливый покупатель ждать не будет! Лежать товары должны компактно, чтобы на склад поместилось как можно больше продукции. Никому не нужны затраты на аренду более вместительного склада только из-за того, что коробки и упаковки лежат неправильным образом.

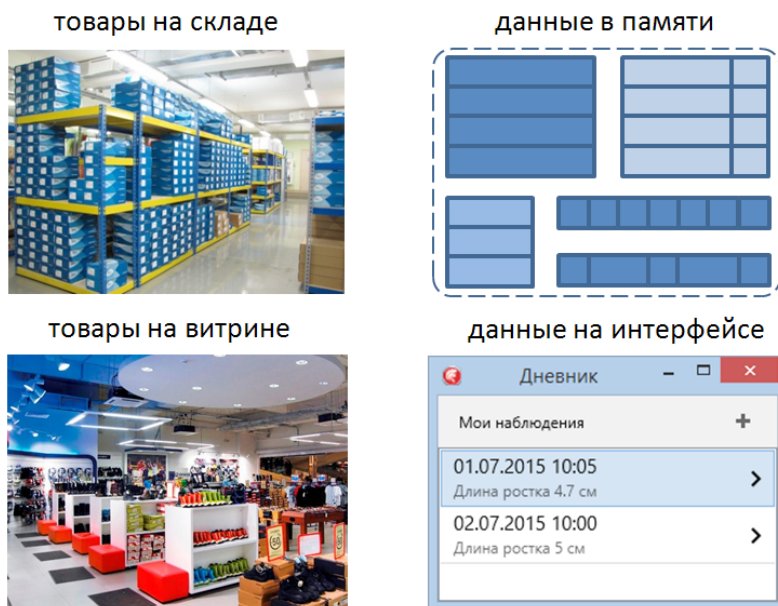


Рис. 6.7. *Хранение структурированное и пользовательское*

То же самое можно сказать и о данных применительно к приложениям (рис. 6.7). На интерфейсе они представлены при помощи визуальных компонентов. Визуальные компоненты и элементы управления подбираются таким образом, чтобы пользователь мог оценить качество и полезность данных. Также важно удобство манипулирования. Одни и те же данные могут быть представлены разным образом — и в списке, и в детальном описании на отдельной страничке. Развивая аналогию, обратимся к спортивной одежде в магазине. Она демонстрируется (как и данные на интерфейсе) разными способами. У входа она показана на манекене для наглядности. А на полке её можно взять в руки, рассмотреть, почитать состав материала ткани и даже примерить.

Логика размещения товаров на витрине (логика интерфейса) также может быть подчинена и такому правилу. Если покупатель интересуется роликовыми коньками, то на полке рядом нужно разместить защитную амуни-

цию. Таким образом, мы повышаем вероятность покупки дополнительного инвентаря, что полезно для бизнеса, а также делаем процесс приобретения товара более удобным. Покупателю не нужно метаться по магазину в поисках необходимого дополнительного товара. На складе же всё размещено совсем по-другому: коробки с роликовыми коньками совсем не обязательно должны соседствовать с наколенниками и шлемами.

Понимание необходимости разделить интерфейс пользователя и структуры хранения данных в памяти необходимо и из следующих соображений, которые мы опять проиллюстрируем на примере магазина. Часто внутренний интерьер магазина перестраивают, переоформляют витрины, изменяют схему выкладки товаров. Естественно, складское хранение от этого не изменяется. В случае с мобильным или обычным приложением данный подход тоже справедлив. По требованию пользователей или по мере развития функционала приложения интерфейс может изменяться, а структуры данных в памяти остаются неизменными.

В нашем приложении данные о наблюдениях будут храниться в специально спроектированных структурах, как для хранения коробок на складе есть специальные стойки и полки. Структуры для хранения данных каким образом не будут связаны с вариантами интерфейсов. Данные отражают, как говорится, предметную область. А интерфейс — потребности пользователя. Предметная область у нас — наблюдения за естественными процессами в природе. Будем ли мы фиксировать наблюдения на листке бумаги, в электронной таблице, зарубками на бревне или в мобильном приложении, вид данных от этого не изменится. Конечно, хранение данных в памяти компьютера (а современное мобильное устройство в полной мере соответствует этому названию) требует создания определённых структур. И нашей очередной задачей является проектирование данной структуры.

Сначала зададимся целью создать список параметров, которые мы будем записывать для единичного наблюдения. Вполне подойдёт следующий перечень:

- дата наблюдения (тип данных «дата»);
- время наблюдения (тип данных «время»);
- длина ростка в миллиметрах (тип данных «вещественное»);
- продолжительность светового дня на день наблюдения в часах (тип данных «целое»);
- температура в момент наблюдения в градусах Цельсия (тип данных «целое»);
- полив в день наблюдения (да/нет, тип данных «логическое значение», «булево»);
- комментарий (тип данных «строка»).

Важно решить, какие данные нужны, а какая информация нам не является полезной с точки зрения наблюдений. Сейчас не нужно рассуждать «влезет ли на интерфейс» или «будет ли красиво смотреться». Даже не стоит пока думать о типах данных Delphi или C++Builder. Сейчас главное — не потерять адекватность представления. Здесь под словом «адекватность» мы подразумеваем соответствие между реальным наблюдаемым явлением и тем описанием, которое будет храниться в приложении. Например, параметры в списке типа «цвет лейки, из которой производится полив» или «моё настроение в момент измерения длины ростка» явно будут неадекватными. Однако, параметр «тип музыки, который проигрывается в течение дня растению» может признаваться как неадекватным, так и вполне уместным (существуют вполне научные данные о том, что под классическую музыку растения растут быстрее). Напоминаем, здесь решение принимается с точки зрения предметной области: биологии или ботаники согласно задачам исследования.

Пока список данных выглядит вполне адекватным, поэтому можно перейти к его анализу с точки зрения программирования. Каждый пункт списка выше снабжён пояснением относительно предполагаемого типа данных. При всей кажущейся очевидности тут тоже могут быть варианты. Например, дату и время наблюдения можно хранить в одной переменной типа `DateTime`, что возможно в Delphi/C++Builder. Также можно хранить их от-

дельно в переменных типа TDate (дата) и TTime (время). Комментарий можно хранить в виде одной строки символов, а можно в виде многострочного текста или массива строк. Конечно, здесь вариантов уже гораздо меньше, но именно при проектировании структур данных начинают сказываться особенности языка программирования.

Переходим к непосредственному проектированию структур данных. С одной стороны, это — один из важнейших этапов разработки программного обеспечения. С другой — всегда можно придумать несколько вариантов, и перед нами встанет проблема выбора наилучшего. Вернёмся к нашей аналогии. Нам нужно оборудовать склад. Сделаем ли мы там обычные универсальные стойки, как показано на рис. 6.7, или построим сложное хранилище с автоматической подачей товара и его предварительной сортировкой и проверкой качества? Как всегда, нет однозначного ответа. Если магазин маленький, то подойдёт самый простой вариант хранения. Если это — гигантский склад, то строить его нужно по-другому. В нашем случае приложение является мобильным, небольшим и по размеру, и по функционалу. Нам подойдёт и самый простой принцип построения структур данных, основанный на массивах.

Массив представляет собой набор однотипных элементов. Если переменная `a: integer` представляет собой ячейку, в которую можно поместить целочисленное значение согласно типу, то массив `b: array [1...10] of integer` даёт нам 10 ячеек. В любую из этих 10 ячеек можно поместить опять же целочисленное значение. При этом мы пользуемся одним и тем же именем `b`, а во избежании путаницы для каждой ячейки из десяти мы указываем её номер. Например, `b[4] := 15`. Это означает, что в четвёртую ячейку или элемент массива `b` копируется число 15. Опять вспомним складское хозяйство. Представьте себе, что есть стеллаж `b`, состоящий из 10 полок. Очень удобно размещать там однотипные коробки. А когда нам понадобится коробка с четвёртой полки, мы поищем по адресу «стеллаж `b`, полка 4». По букве мы найдём стеллаж, а потом отсчитаем четвёртую полку от начала. Всё просто, как в обычной жизни, так и в программировании (рис. 6.8).

Перейдём к более сложному уровню. Нам нужно обеспечить хранение данных о повторяющихся наблюдениях. Если мы наблюдали за ростом фасоли один месяц, делая записи один раз в день, то нам понадобятся массивы с тридцатью одной ячейками. Один массив будет хранить даты наблюдений, другой — время. Третий — длину ростка, четвёртый — был ли полив в день наблюдения, пятый — комментарии. По сравнению со списком выше мы убрали некоторые данные. Сделано это было умышленно, чтобы не загромождать ни текст данной книги, ни код программы. Согласитесь, что добавить ещё пару-тройку массивов будет лёгкой задачей, как только мы решим всю задачу для пяти выбранных параметров наблюдений.

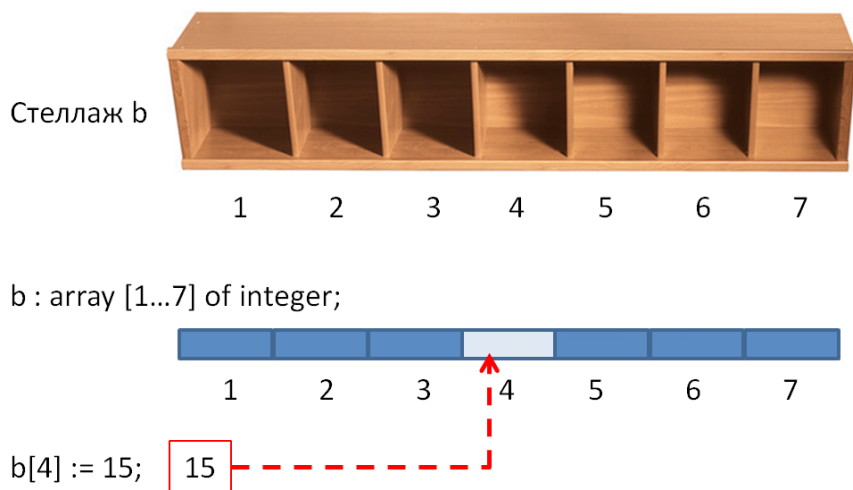


Рис. 6.8. Доступ к ячейкам стеллажа и массива

Зададим эти массивы в коде программы. Переключимся с дизайнера формы к текстовому редактору кода программы, нажав F12 или другим удобным вам способом. Найдём класс формы, в нём отыщем раздел `private`, куда поместим следующий текст:

```

TForm1 = class(TForm)
// код пропущен
private
{Private declarations}
arDate: array [1..31] of TDate;           // хранение даты
arTime: array [1..31] of TTime;          // хранение времени
arLength: array [1..31] of single;       // хранение длины
arWater: array [1..31] of boolean;       // хранение полива
arComment: array [1..31] of string;      // хранение
комментариев
public
{Public declarations}
end;

```

Как видно, мы добавили пять массивов для различных типов значений. У достаточно искушенных программистов, а также у просто думающих читателей может возникнуть вопрос: «а если мы будем наблюдать не один, а два месяца? В массивах же не хватит ячеек, чтобы хранить данные!». Вопрос уместен, т.к. размер массивов мы задали жёстко — от первой до тридцать первой ячейки. Методы организации хранения в «резиновых» массивах представляет собой отдельную, крайне интересную тему, которую нужно раскрыть в соответствующем тематическом разделе. Нет большой проблемы в реализации «динамических массивов» при помощи языка Object Pascal. Но для прототипа приложения будем считать, что продолжительность наблюдений задана фиксировано на момент разработки приложения. В дальнейшем можно будет либо увеличить размер массивов, либо перейти к более универсальному методу организации хранения данных с произвольным числом элементов.

При работе с массивами нужно грамотно оперировать циклами. Цикл — это такой метод организации исполнения, когда программный код делает одно и то же несколько раз. В обычной жизни это тоже часто встречается. Например, помыть тарелку — это совокупность отдельных операций типа:

1. взять грязную тарелку;
2. выдавить моющего средства на губку;
3. смочить тарелку;

4. намылить тарелку;
5. смыть моющее средство;
6. поставить тарелку в сушку.

Если мы «зациклимся», т.е. будем продолжать делать то же самое, каждый раз беря другую грязную тарелку, то мы перемоем всю посуду. Представьте, что вместо стопки грязных тарелок у нас есть последовательность элементов в массиве. Если нам нужно сделать какое-то однотипное действие с каждым элементом, то самое правильное — организовать цикл. Таким повторяющимся однотипным действием является инициализация элементов массива, т.е. присвоение им начального значения. Лучше всего это делать в процедуре отклика на событие `OnCreate`. Сгенерируем при помощи среды разработки данную процедуру и добавим код, инициализирующий массив для хранения длинны роста:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
  i: integer; // переменная цикла  
begin  
  for i:= 1 to 31 do // выполнится 31 раз  
    begin  
    arLength[i]:= 0.0; // присвоить значение текущему элементу  
    end;  
  end;
```

Если понимание данного кода вызывает большие сложности, значит нужно освежить базовые знания о языке Pascal/Dephi. Вначале объявлена целочисленная переменная `i`, которая затем использована для организации выполнения тела цикла, состоящего из одной строчки. Тело цикла выполнится 31 раз, причем каждый раз переменная `i` будет последовательно принимать значения 1, 2, 3, ... 31. Соответственно, `arLength[i]` каждый раз будет означать: `arLength[1]` — первый элемент, `arLength[2]` — второй элемент, `arLength[3]` — третий элемент, ... и, наконец, `arLegnth[31]` — последний, тридцать первый элемент. Теперь мы уве-

рены, что каждому элементу в процессе инициализации в цикле присвоено вещественное значение 0.

Для самоконтроля правильности ввода кода выше, сохраним и запустим приложение. Если были допущены ошибки, то компилятор тут же нам их покажет. Если всё прошло хорошо, то вернёмся в редактор кода и после строчки `arLength[i] := 0.0;` добавим инициализацию остальных массивов, естественно, в теле цикла:

```
arWater[i] := false; // изначально полива не было
arComment[i] := ''; // «пустая» строка
arDate[i] := 0; // «пустая» дата
arTime[i] := 0; // «пустое» время
```

Что означает «пустая» строка? Это строка, в которой нет символов. А что такое «пустая» дата? Это — некоторое число, не означающее реальной даты. То же самое и со временем. Если вас интересует, как представляется дата и время в компьютере и как с ними работать при помощи языков программирования, то рекомендуется самостоятельно ознакомиться с этим, т.к. данная книга не является справочником по языку программирования. Возможно, вы уже хорошо себе представляете систему хранения данных о дате и времени, поэтому сконцентрируемся на максимально быстром завершении проекта.

6.4. Интерфейс детального просмотра

Весь предыдущий раздел мы проектировали структуру хранения данных, наш «склад». Теперь мы уверены, что он готов для хранения записей о наблюдениях на протяжении целого месяца. Появилось время для подготовки «витрины», т.е. интерфейса пользователя. Первая страничка компонента `TabControl1` уже была оформлена нами в самом начале работы над проектом. Перейдём к страничке № 2, которая будет отображать детальную информацию. Пока она у нас совсем пустая, но сейчас мы начнём декорировать её, знакомясь с техникой быстрого прототипирования.

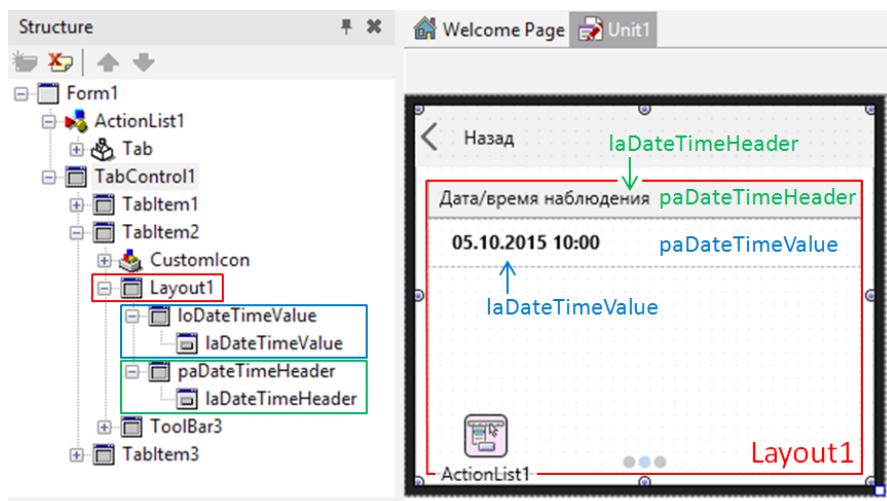


Рис. 6.9. Разметка странички детального просмотра

Выполним ряд действий в строгой последовательности, сверяя каждый свой шаг с рис. 6.9:

1. Добавим на страничку компонент TLayout. Он выглядит на форме как прозрачная панель. Его задача — быть областью размещения других компонентов. Растянем его на страничке так, чтобы остались небольшие поля, т.е. расстояния между его границей и границей странички. Чтобы при изменении формы он также растягивался и сужался, поставьте все четыре якоря Anchors в значение true. Найдём это комплексное свойство в Object Inspector, раскроем [+] и кликните на каждой из появившихся строк. Теперь можно мышкой растянуть форму в дизайнера, компонент Layout1 также будет изменять свои размеры.
2. Теперь разместим уже внутри Layout1 компонент TPanel (отмечен зелёным цветом на рис. 6.9). Свойство Align установим в значение Top. На данной панели разместим компонент TLabel, свойству Text присвоим значение «Дата/время наблюдения». По умолчанию названия компонентов будут что-то типа Panel1 или Label2. Представьте себе, что в приложении будет десять-двадцать компонентов с ничего не значащими названиями. Во избежание путаницы название ком-

понента нужно заменять на более значимое. Переименуйте Panel1 в paDateTimeHeader, а Label2 в laDateTimeHeader. От названия компонента остаётся лишь префикс (приставка) pa- или la- от слов panel и label. Дальше идут слова, поясняющие назначение компонента. Например, paDateTimeHeader можно расшифровать как «панель для заголовка даты и времени».

3. Для компонентов TLabel нужно устанавливать свойство Align в значение Client, а чтобы был отступ слева, свойство Margins->Left устанавливать в нужное значение. В нашем случае это либо 10, либо 20 в зависимости от величины отступа. Если этого не сделать, то компоненты могут «съехать» при переходе сборки проекта с Windows на Android.
4. Расположим на Layout1 компонент TLayout, свойству Align установим значение Top. Внутри него разместим компонент TLabel (синяя группа). Сразу же переименуем их в loDateTimeValue и laDateTimeValue («значение даты и времени»). На рис. 6.9 данная группа помечена синим цветом. Метке laDateTimeValue в качестве значения свойства Text в Object Inspector зададим строку с некой датой/временем.
5. Ещё раз обратим внимание на правильные имена компонентов, сверив данные значения с дерево компонентов на панели Structure, рис. 6.9.
6. Нажмём клавишу Shift и мышкой выделим компоненты paDateTimeHeader и loDateTimeValue. Затем нажмем Ctrl+C или выберем в меню Edit->Copy. Кликнем на Layout1 в пустом пространстве под выделенными компонентами, чтобы выделить именно его. Теперь нажмем Ctrl+V или выберем в меню Edit->Paste. Результатом должен явиться полный клон группы компонентов. Сделайте так ещё три раза, а результат сверьте с рис. 6.10.
7. Зададим всем новым компонентам имена согласно их функциям. Будем использовать буквы из названия компонента для префикса (приставки). Постфикс (окончание) должен быть либо Header, либо Value в зависимости от того, будет ли компонент «декоративным» или же он будет отображать значение наблюдаемого параметра.

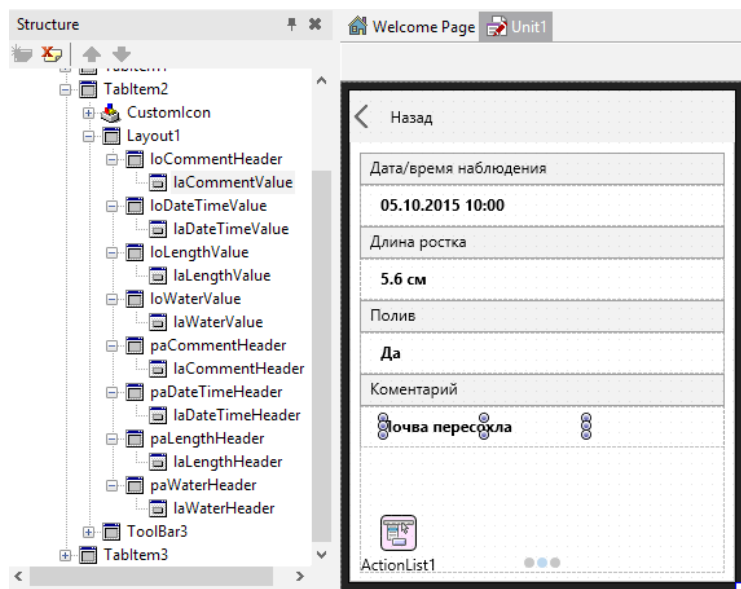


Рис. 6.10. Окончательный дизайн странички детального просмотра

Не забудем в дизайнера переключиться на первую страничку. Сохраняем и запускаем проект на исполнение. На первой страничке у нас есть список наблюдений в `ListBox1`. Для данного компонента мы уже прописали событие `OnClick`, которое пока просто выполняет навигацию с первой странички на вторую. Сейчас мы модифицируем данную процедуру, чтобы избежать ошибки, которая вполне вероятно у вас уже появлялась. Сначала запишем код:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  if ListBox1.ItemIndex < 0 then exit;
  ChangeTabAction3.ExecuteTarget(TabControl1);
end;
```

Добавлена всего одна строчка с оператором условия `if`, но она очень важная и требует пояснений. Компонент `ListBox1` со списком может нахо-

диться в состоянии, когда ни один из его элементов не выбран пользователем. Если пользователь кликнул на `ListBox1`, но не попал ни в один из элементов, то его свойство `ItemIndex` равно `-1`. Элементы начинают считаться с нуля, поэтому если `ItemIndex` равен 0 или другой положительной величине, значит, пользователь попал в какой-либо элемент, и нужно выполнять навигацию. В противном случае мы делаем `exit`: элемент не выбран, нет смысла переходить на вторую страничку.

6.5. Интерфейс добавления новой записи

Перейдём на третью страничку и доделаем оставшуюся часть дизайнерской работы. Последняя страничка будет похожа на вторую, однако мы изготовим её принципиально по-иному. Если в предыдущий раз мы использовали группы компонент `TPanel` и `TLayout` на общем большом `TLayout`, то сейчас мы применим другой метод. Разместим на третьей страничке уже знакомый нам компонент `TListBox`. Как и в предыдущем разделе, применим технику «якорей». Вручную при помощи мышки увеличим размер практически во всю предоставленную область, оставляя небольшие поля (рис. 6.11). После чего зафиксируем расположение `ListBox2`, установив в `true` все якоря комплексного свойства `Anchors` в `Object Inspector`.

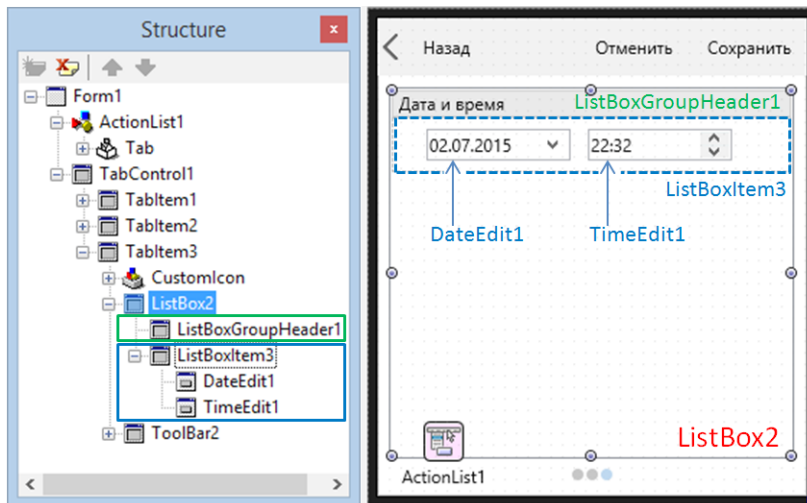


Рис. 6.11. Разметка странички добавления записи

Внимательно выполним следующие действия:

1. Щелчком правой кнопкой мыши на ListBox2, в появившемся меню выберем AddItem->TListBoxGroupHeader. Название компонента будет задано автоматически как ListBoxGroupHeader1 (зелёный цвет на рис. 6.11). Свойству Text зададим значение «Дата и время».
2. Кликнем правой кнопкой мыши на ListBox2 и в контекстном меню выберем Add TListBoxItem. Компонент будет автоматически назван ListBoxItem3 (помечен синим цветом на рис. 6.11). На данный компонент разместите TDateEdit и TimeEdit. Названия будут заданы также автоматически DateEdit1 и TimeEdit1.
3. Последовательно повторим шаги 1 и 2 выше, закончив построение интерфейса странички добавления в соответствии с рис. 6.12. Если при добавлении нового обычного TListBoxItem он будет слишком низким, то можно добавить высоты, увеличив значение свойства Height в Object Inspector или просто мышью.
4. Для раздела «полив» выберите компонент TSwitch. Компонент служит для «бинарного выбора», т.е. выбора одного значения булевского типа из вариантов «да» и «нет».
5. Для раздела «комментарий» найдем и разместим компонент TMemo. Данный компонент позволяет вводить многострочный текст. В Object Inspector раскройте свойство TextSettings и установите WordWrap как true. Теперь текст в этом компоненте будет автоматически переноситься на новую строку при вводе.

Как и в случае с TPanel и TLayout можно выполнять операции копирования/вставки компонентов. Но в нашем случае это не особо полезно, т.к. добавление через контекстное меню по времени соизмеримо, а то и быстрее операций копирования/вставки.

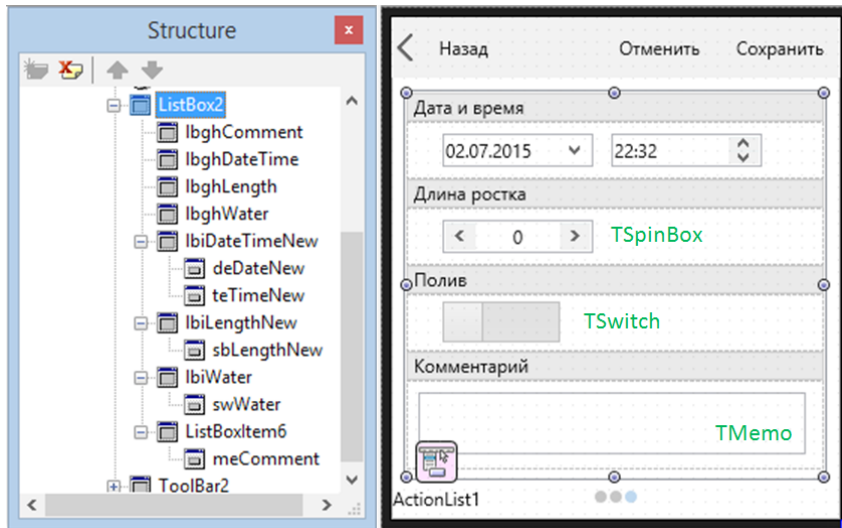


Рис. 6.12. Окончательный дизайн странички добавления новой записи

Рассмотрим компонент TSwitch. Компьютерные программы оперируют значениями определённых типов. Нельзя говорить о значении без какого-либо типа. Любое «просто число» имеет тип. Именно благодаря типу приложение знает, какую ячейку памяти нужно выделить и каким образом записать или считать оттуда значение. Мы уже знаем простые типы языка Pascal/Delphi: целый (integer), вещественный (single), логический (Boolean). Также мы упоминали строковый тип (string), типы «дата» (TDate) и «время» (TTime). Самый простой способ ввода любых значений — предоставить компонент «поле ввода» (TEdit) для всех типов значений. Но он подразумевает только использование клавиатуры и не уберегает от ошибок ввода, как-то ввод несуществующей даты (например, 30.02.2015 или «пять миллиметров» в поле для длины ростка). Для ввода различных типов значений и при помощи клавиатуры, и посредством пальца и стилуса, причем с высокой степенью защиты от ошибок доступны различные компоненты.

Для ввода данных типа «дата» TDate отлично подходит компонент TDateEdit. Если вы тестировали наше приложение, то уже увидели, как он работает. Если нет, запустите приложение и кликните на кнопке [+], чтобы

перейти на страничку с добавлением новой записи. В компонент TDateEdit можно ввести дату, причем только существующую. При попытке ввести 30.02.2015 сам компонент переправит день на 28, максимальное число в феврале 2015-го года. А если нажать кнопку выпадения списка данного компонента, то появится «календарик», где визуально можно выбрать нужную дату. То же самое можно сказать про компонент TTimeEdit, существенно облегчающий корректный ввод времени. Компонент TSpinBox эффективен для использования в мобильных приложениях, когда нужно число можно ввести при помощи кнопок компонента, а не клавиатуры.

В случае с вводом логических данных мы сталкиваемся с небольшой проблемой выбора. Классически для операционной системы Windows (и не только) был создан компонент типа «галочка» (check box), который весьма просто решал проблему ввода логического значения. Если «галочка» стояла (изначально или в результате действия пользователя), то это интерпретировалось как значение «истина» или «true». Если «галочку» снимали или её не было, то значение «ложь» или «false». Можно даже найти компонент TCheckBox и разместить его рядом с TSwitch для сравнения, т.к. последний выполняет ту же функцию. Для мобильных устройств компонент типа TCheckBox не очень подходит вследствие малых размеров. В конечном итоге нашей целью является именно мобильное приложение, поэтому мы и используем для интерфейса компонент TSwitch.

6.6. Связывание интерфейса и структур данных в памяти

Интерфейсная часть приложения готова, структуры данных в памяти в виде массивов тоже созданы. Скопируем проект в папку «Project 6.1», запустим среду разработки Delphi/C++Builder/RAD Studio и обеспечим связь между интерфейсом и данными. Начнём с добавления новой записи о наблюдении. Сначала разберём логику работы, а потом напишем соответствующий код. Итак, у нас есть несколько массивов для хранения данных, а также размечена страничка для добавления. На этой третьей страничке

есть компоненты, куда пользователь может ввести дату, время наблюдения, длину роста фасоли в миллиметрах и при помощи компонента Switch1 поставить «да» в случае полива или «нет», если его не было в день наблюдения. Сначала данные будем записывать во внутреннее хранилище, которым являются структуры данных типа массив, а затем занесём новый элемент в список на первой страничке.

Запишем последовательность действий по шагам:

1. определить порядковый номер новой записи («индекс»);
2. в массив дат по номеру новой записи поместить значения даты наблюдения из компонента deNew, свойство Date;
3. в массив времени по номеру новой записи поместить значение времени наблюдения из компонента teNew, свойство Time;
4. в массив длин ростка по номеру новой записи поместить значения длины из компонента sbLengthNew, свойство Value;
5. в массив признаков полива в день наблюдения поместим значение «истина» или «ложь» из компонента swWaterNew, свойство IsChecked.

На этом месте мы закончили ввод данных из интерфейса в массивы хранения. Рис. 6.13 содержит графическую иллюстрацию предложенного выше алгоритма. Обратим внимание, что значение «истина» или «ложь» в зависимости от положения «переключателя» компонента Switch1 хранится в свойствах IsChecked. Запомним это свойство. Для переключателей в реальной жизни более уместны значения «on»/«off» или «вкл»/«выкл», но нам следует оперировать значениями «true»/ «false» или «истина»/ «ложь». Само же название IsChecked унаследовано от компонента типа «галочка», где вполне было уместно свойство «выбрана» или «checked». Нужно привыкнуть, что иногда проще запомнить, чем искать аналогии в реальном мире, т.к. иногда они не работают. Еще раз внимательно изучим рис. 6.13 и начнём вводить первую часть кода на событие OnClick кнопки «Сохранить»:

```

procedure TForm1.SpeedButton3Click(Sender: TObject);
var
  NewIndex: integer;
  NewListBoxItem: TListBoxItem;
begin
  // ShowMessage('запись сохранена');
  // часть № 1, добавление данных в массивы
  NewIndex:= ListBox1.Count + 1;
  arDate[NewIndex]:= deDateNew.Date;
  arTime[NewIndex]:= teTimeNew.Time;
  arLength[NewIndex]:= sbLengthNew.Value;
  arWater[NewIndex]:= swWaterNew.IsChecked;
  arComment[NewIndex]:= meCommentNew.Text;
  // часть № 2, добавление элемента в список
  // будет дальше
end;

```

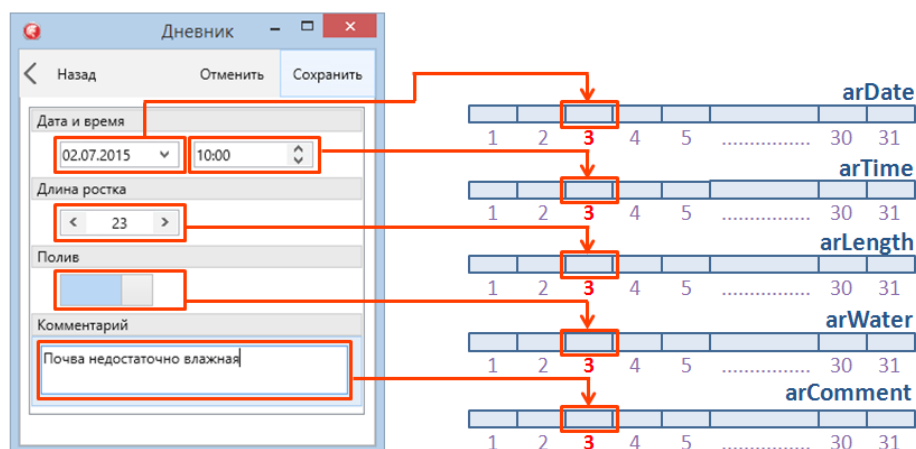


Рис. 6.13. Запись данных из интерфейса в массивы

Разберём представленный выше код. Первым делом мы, как и обещали, закомментировали строку, которая просто показывала уведомительное окно, но реально ничего не делала. В следующей строке мы вычисляем номер или индекс нового элемента. Конечно, мы помним, что добавили два элемента в список, поэтому новый номер будет «три». Но это пока у нас только два элемента, а нужно уметь вычислять данный номер. Номер или индекс нового элемента будет равен количеству уже существующих элементов плюс единица, что и написано в коде: `newindex:= ListBox1.`

`Count + 1;` где свойство `Count` содержит количество элементов в списке. Далее всё просто, мы в ячейку каждого массива по вычисленному номеру записываем введенные значения из соответствующих компонентов интерфейса.

После записи введенных пользователем данных в массивы нашей задачей является создание нового элемента в списке записей на первой страничке. Вспомним вкратце, как это мы делаем в режиме `design-time`. Мы щёлкаем правой кнопкой мыши на компоненте `LisbBox1`, а из всплывающего контекстного меню выбираем `Add TListBoxItem`. После этого в списке появляется готовый элемент, который будет отображаться в `Object Inspector`. Сейчас можно поочередно выбрать первый и второй добавленные элементы в списке и посмотреть на их свойства в `Object Inspector`.

Данное полезное упражнение освежило наше восприятие элемента списка как отдельного компонента. Это — не строка в электронной таблице, это — полноценный компонент. Он появляется в списке подобно тому, как мы располагаем на форме другие компоненты — `TSpeedButton`, `TLabel` и т.д. Пока мы умеем это делать с любым компонентом в режиме `design-time`, а теперь должны научиться делать это при помощи программного кода уже в процессе работы приложения.

6.7. Динамическое создание компонентов

Поскольку наш компонент `TListBoxItem` будет появляться на форме в результате действий пользователя на этапе работы приложения, как говорится, во время исполнения или `runtime`, то такой способ называется динамическим. Когда среды визуального программирования ещё только зарождались, объекты создавались преимущественно программно. Не было «режима визуальной разработки интерфейсов», а все компоненты размещались на форме не так, как это мы делаем сейчас. Программистам приходилось писать массу кода вручную для размещения элементов управления на форме. Современные средства визуальной разработки, такие как `Delphi/ C++Builder/RAD Studio`, избавляют нас от необходимости вводить текст там,

где можно обойтись визуальным редактором. Мы размещаем компоненты на форме, не задумываясь о том, как они создаются при запуске приложения. Но если какие-либо элементы интерфейса должны появляться на форме в процессе работы приложения, то без программирования тут не обойтись.

Большим преимуществом для реализации этой техники является хорошее знание базовых принципов объектно-ориентированного программирования. Но если у вас пока их нет, то развитие данного проекта поможет вам разобраться в основах. Каждый компонент, который мы видим как элемент интерфейса, представляет собой объект. Мы уже обсуждали с вами классы и объекты. Приведём ещё один пример, класс — это схема сборки велосипеда. Зная схему сборки и понимая функционал всегда можно изготовить велосипед или несколько велосипедов из отдельных частей. Но сама по себе схема сборки не является «объектом передвижения» или «транспортным средством». Чтобы воспользоваться схемой, нужно всё-таки с её помощью собрать экземпляр или объект.

Итак, за основу нужно взять класс. Класс того элемента интерфейса, который мы хотим изготовить. В нашем случае мы возьмём класс `TListBoxItem` и по нему, как по чертежу или схеме изготовим объект — элемент списка для `ListBox1`. Изготовление экземпляра или объекта по его классу производится при помощи «конструктора `Create`». Думайте о «конструкторе `Create`» как о некоем «мастере-на-все-руки» слесаре «дяде Васе», которому вы можете дать чертёж или схему сборки чего угодно, а он всё это аккуратно изготовит для вас. Если вы дадите ему чертёж велосипеда — будет велосипед. Если чертёж скутера, то скутер. Если чертёж автоматической поилки для кота, то он изготовит и её. Но нас в данный момент сильно интересуют не транспортные средства и агрегаты для ухода за домашними животными, а элементы списка `TListBoxItem`. Именно изготовление такого элемента мы и поручим нашему гениальному «конструктору».

Поскольку мы всё-таки говорим не о реальном физическом мире, а о приложениях, то здесь будет небольшая особенность. Как только «кон-

структор Create» изготовил для нас элемент списка, он тут же забывает о нём. Чтобы воспользоваться элементом списка, нам нужно знать, где его найти. Опять воспользуемся аналогией. У вас может быть друг Пети. Чтобы пообщаться с ним или попросить что-то сделать для вас, вам нужно каким-то образом его найти и обратиться к нему. Самый простой способ — позвонить по мобильному телефону. Звонок можно осуществить, если у вас есть телефонный номер Пети. У вас может быть много друзей, но каждый имеет свой уникальный номер. Единожды записав номер Пети, вы всегда можете к нему обратиться. В идеальном мире у каждого человека есть уникальный телефонный номер, зная который, всегда можно человека найти и обратиться к нему.

В мире объектно-ориентированного программирования у каждого объекта есть ссылка. Ссылка позволит всегда найти объект, вступить с ним в контакт и заставить выполнить нашу просьбу. Вы можете записать телефон Пети в разных местах — в книгу контактов на смартфоне, в бумажную записную книжку или в облачное хранилище. Так и ссылка на объект может храниться в разных местах. Но где бы вы ни записали телефон Наташи, это всегда будет одно и то же значение. Сейчас мы объявим переменную для хранения ссылки на объект, потом создадим объект при помощи конструктора и тут же запишем ссылку в переменную. Не будем приводить абстрактные примеры, а создадим объект класс TListBoxItem:

```
var
  // объявляется переменная-ссылка
  // на объект класса TListBoxItem
  NewListBoxItem: TListBoxItem;
begin
  NewListBoxItem:= TListBoxItem.Create(ListBox1);
  // ...
end;
```

Разберём приведённый выше код. В разделе var объявляется переменная, что характерно для языка программирования Pascal/Delphi. Обычно мы записываем имя переменной, а после двоеточия приводим тип. Если после двоеточия integer, то переменная может хранить значение типа «целочисленное». Если TDate, то переменная будет хранить значение

типа «дата». А если `TListBoxItem`, то переменная может хранить значение типа «ссылка на объект класса `TListBoxItem`». Именно для этого и будет использована наша переменная `NewListBoxItem`. Рис. 6.14 иллюстрирует приведённые объяснения.

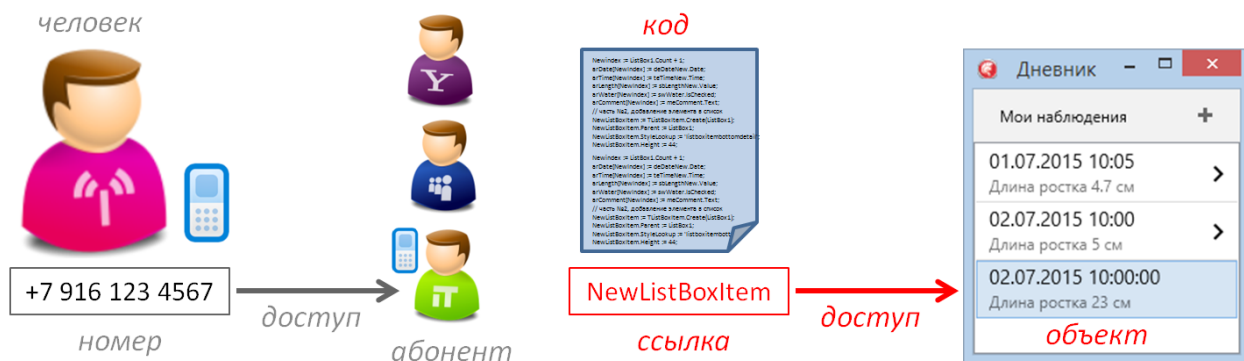


Рис. 6.14. Доступ по номеру телефона и по ссылке

Дальше после слова `begin` начинается самое интересное. Читаем строчку справа налево, как и нужно делать, когда мы разбираем какой-то код. Тут мы видим `Create`, т.е. вызов нашего мастера-самоделкина, который создаст новый элемент списка `TListBoxItem`. Теперь отбрасываем метафоры и чётко формулируем: код `TListBoxItem.Create(ListBox1)` создаёт объект класса `TListBoxItem`. Конструктору `Create` принимает в качестве аргумента `Listbox1`, чтобы созданный элемент списка сразу попал в `Listbox1`. Далее в полной строчке кода мы копируем ссылку на новый созданный объект в переменную `NewListBoxItem`. Теперь в любой момент мы через переменную `NewListBoxItem` можем обратиться к созданному объекту — элементу списка.

Зачем нам может потребоваться обратиться к новому созданному элементу списка? Во-первых, для настройки этого элемента. В design-time мы выбирали элемент и настраивали его при помощи Object Inspector. В режиме работы приложения (runtime) возможности использовать Object

Inspector у нас нет, поэтому настраивать свойства элемента будем посредством ссылки. Теперь можно рассмотреть и вторую часть кода процедуры [+] на добавление новой записи о наблюдении:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
var
  NewIndex: integer;
  NewListBoxItem: TListBoxItem;
begin
  // ShowMessage('запись сохранена');
  // часть № 1, добавление данных в массивы
  // код пропущен
  // часть № 2, добавление элемента в список
  NewListBoxItem:= TListBoxItem.Create(ListBox1);
  NewListBoxItem.Parent:= ListBox1;
  NewListBoxItem.StyleLookup:= 'listboxitembottomdetail';
  NewListBoxItem.Height:= 44;
  NewListBoxItem.ItemData.Accessory:= TListBoxItemData.
  TAccessory.aMore;
  NewListBoxItem.ItemData.Text:= DateToStr(arDate[NewIndex]) +
  ' ' + TimeToStr(arTime[NewIndex]);
  NewListBoxItem.ItemData.Detail:= 'Длина ростка ' +
  FloatToStr(arLength[NewIndex]) + ' мм';
  // переход на первую страничку
  ChangeTabAction2.ExecuteTarget(TabControl1);
end;
```

Часть № 1 кода можно найти выше. Теперь разберём код части № 2. Её задачей является передача данных из внутренних массивов на интерфейс пользователя (рис. 6.15). Строка с вызовом конструктора для создания объекта и копирования ссылки в `NewListBoxItem` сложности уже не вызывает. Далее следует `NewListBoxItem.Parent:= ListBox1`; где мы указываем `ListBox1` в качестве «родителя» (свойство `Parent`) у нового созданного компонента. «Родитель» или `Parent` — это компонент, на котором будет располагаться наш `NewListBoxItem`. `ListBox1` был передан конструктору `Create` в качестве аргумента, что означало «новый элемент принадлежит `ListBox1`». Теперь он не только принадлежит, но и располагается на `ListBox1`. Именно в такие моменты нужно поставить себе в план развития более глубокое знакомство с объектно-ориентированным программированием.

Следующие строчки кода не должны вызывать вопросов, т.к. мы практически делаем то, что и ранее при помощи Object Inspector. Мы задаём стиль, высоту элемента, вид картинки-аксессуара, а потом заносим значения, которые будут отображаться данным элементом. Завершающей инструкцией в данной процедуре будет программная навигация на первую страничку, которая уже будет готова продемонстрировать новый элемент в списке.

В завершении раздела хочется чуть глубже разобраться в строках, где происходит запись данных на интерфейс. Для отображения каких-либо данных в элементе списка при выбранном стиле 'listboxitembottomdetail' мы оперируем двумя свойствами: Text и Detail. Два этих свойства имеют строковый тип. Так или иначе, всё, что отображается на интерфейсе, это строчки. У нас есть два значения типа TDate и TTime, из которых нужно собрать одно свойство Text строкового типа. Для этого служит строчка кода: `NewListBoxItem.ItemData.Text := DateToStr(arDate[NewIndex]) + ' ' + TimeToStr(arTime[NewIndex]);`. В этой строке мы «склеиваем» строчку из трёх частей при помощи операторов «+». Применительно к строкам он означает «конкатенацию» или «склеивание» строк, а не арифметическое сложение. Первая часть из нашей склеенной строки формируется следующим образом. Мы берём из массива значение соответствующего элемента. Оно хранится как TDate. «Склеивать» даты нельзя, поэтому мы преобразуем значение `arDate[NewIndex]` в строковое при помощи функции `DateToStr` (date-to-string, «дату-в-строку»). Затем такое же преобразование мы делаем и со временем наблюдения, хранящимся в нужной ячейке массива `arTime`. Промежуточный пробел ' ' вклеивается посередине для правильного формата представления информации.

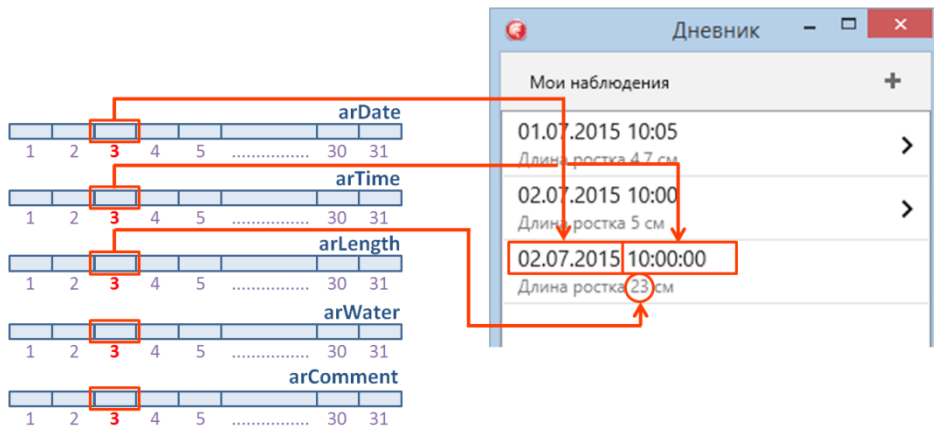


Рис. 6.15. Передача данных из массивов на интерфейс

Если с пониманием вышеизложенного проблем нет, то и строку свойствам Detail понять будет также просто: `NewListBoxItem.ItemData.Detail := 'Длина роста ' + FloatToStr(arLength[NewIndex]) + ' мм';`. Мы берём значение измеренной и введенной длины роста из ячейки массива, а затем преобразовываем это вещественное значение в строку функцией `FloatToStr`. Потом опять «склеиваем» одну длинную строку из трёх фрагментов.

Перечислим основные моменты, касающиеся динамического создания компонентов:

1. Подобно тому, как компоненты визуально размещаются на форме в design-time, их можно создавать во время исполнения приложения runtime при помощи программного кода.
2. Компоненты — это объекты классов.
3. Чтобы создать компонент, нужно знать его класс. Если мы знаем класс компонента, мы в любой момент можем его создать.
4. Компонент, объект, экземпляр класса — это всё синонимы.
5. Чтобы создать компонент (объект, экземпляр класса), нужно воспользоваться конструктором `Create`.

6. Конструктор для классов компонентов обычно принимает параметр. Этот параметр — владелец компонента.
7. Не путаем «владельца» компонента с его «родителем». «Родитель» — такой компонент, на котором появляется наш вновь созданный.
8. Чтобы управлять компонентом, нужно знать ссылку на него. Зная ссылку на компонент, можно получать доступ и изменять его свойства.
9. Ссылка на компонент хранится в специальной переменной, тип которой совпадает с классом. Например, `NewListBoxItem: TListBoxItem`. Переменная `NewListBoxItem` может хранить в себе ссылку на объект класса `TListBoxItem`.

6.8. Реализация детального просмотра

Только что мы научили наше приложение добавлять новую запись о наблюдении. Сначала данные, введённые пользователем при помощи компонентов на третьей страничке, попадают в массивы хранения, а затем уже информация из них поступает на интерфейс первой странички. Для этого создаётся и добавляется новый элемент в список. Перед тем, как информация поступит в новый элемент списка, она преобразовывается и приводится к виду, удобному для просмотра пользователем. Теперь нашей задачей является передача детальной информации из массивов на страничку № 2, спроектированную нами для детального просмотра (рис. 6.16).

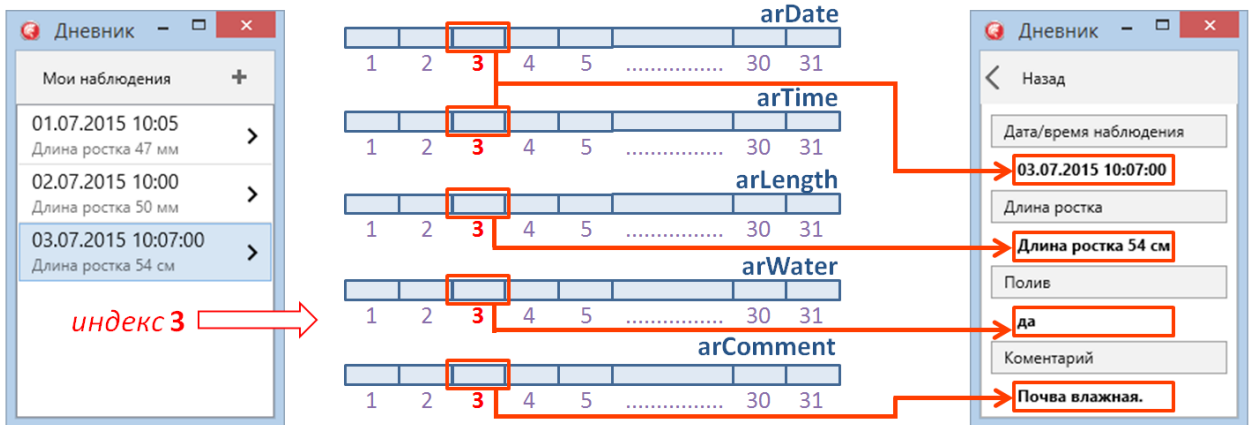


Рис. 6.16. Передача данных из массивов на вторую страничку

Необходимость показать вторую страничку возникает тогда, когда пользователь кликнул нужной ему элемент списка на первой страничке. Такая потребность обязательно возникнет, т.к. в общем списке наблюдений за событиями информация представлена очень кратко. Вспомним предыдущий раздел, когда нам пришлось «склеивать» строки, чтобы вписать их в формат представления элемента списка одной строкой. Естественно, что информативность такого представления невысока. Но взамен мы получаем возможность охватить взглядом самое основное у многих пунктов в списке. Если пользователь хочет рассмотреть конкретный элемент списка, то он кликает на него, а дальше получает детальное представление только одного выбранного элемента. Советуем запомнить такое построение интерфейсов приложений. Основой здесь являются две странички: 1) общее списочное представление многих элементов; 2) подробное представление для одного элемента.

Кратко запишем алгоритм показа детальной информации:

1. Определить номер элемента списка на первой странице, который пользователь хочет видеть подробно.
2. Обратиться к массивам хранения по номеру выбранного элемент.

3. Из нужных ячеек извлечь хранимую подробную информацию для нужного элемента.
4. Отобразить на компонентах второй странички хранимую в массивах информацию
5. Выполнить навигацию на вторую страничку.

Перейдём в код нашего приложения и найдём процедуру `ListBox1Click`, которая выполняется автоматически всякий раз, когда пользователь кликает на списке. В разделе 6.4 уже описан дизайн странички и некоторый код данной процедуры, которая пока выполняет проверку, выбран ли какой-то конкретный элемент. Также процедура делает навигацию на вторую страничку. Но теперь мы готовы ввести код процедуры полностью:

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
    ListBoxItemIndex: integer;
begin
    if ListBox1.ItemIndex < 0 then exit; // если элемент не
    выбран
    ListBoxItemIndex:= ListBox1.ItemIndex + 1; // индекс эле
    мента
    laDateTimeValue.Text:= DateToStr(arDate[ListBoxItemInd
    ex]) + ' ' + TimeToStr(arTime[ListBoxItemIndex]);
    laLengthValue.Text:= 'Длина ростка '
    + FloatToStr(arLength[ListBoxItemIndex]) + ' мм';
    if arWater[ListBoxItemIndex] then
        laWaterValue.Text:= 'да'
    else
        laWaterValue.Text:= 'нет';
        laCommentValue.Text:= arComment[ListBoxItemIndex];
        // навигация на вторую страничку
    ChangeTabAction3.ExecuteTarget(TabControl1);
end;
```

Начинаем анализировать код, начиная со второй строки, т.к. первую мы уже рассмотрели ранее в разделе 6.4. На второй строке мы определяем индекс или номер элемента, по которому будем искать его данные в массивах. В большинстве случаев в Delphi/C++Builder/RAD Studio номера элементов отсчитываются с 0. Нужно привыкнуть, что первый элемент имеет индекс 0,

второй 1, третий 2, т.е. на единицу меньше, чем его реальный порядковый номер. Поскольку мы объявили все наши массивы с диапазоном нумерации элементов [1..31], то для работы с ними нужна «нормальная» нумерация, т.е. с 1. Поэтому к значению `ItemIndex` компонента `ListBox1` нам именно в этом случае нужно прибавить 1. Прибавление единицы — вынужденная мера (а не постоянный ритуал), когда речь идёт о правильной индексации. Если хотите избежать прибавления 1, то всегда указывайте диапазоны массивов с 0.

Далее мы увидим уже знакомое извлечение значений из массивов и отправку их в форматированном виде в свойства компонентов, формирующих интерфейс второй странички. Обращаем внимание, что нам приходится «склеивать» преобразованные в строки данные, что и есть правильно. Никогда не надо жертвовать структурой или типизацией хранилищ данных в памяти ради красоты интерфейса. Как и наоборот, нельзя умышленно «обеднять» интерфейс, делать его не информативным только из-за того, что данные хранятся в памяти «некрасиво». Нужно лишь хорошо владеть преобразованиями типов данных и не лениться форматировать информацию, предназначенную для пользователя.

Настало время протестировать наше приложение. Зайдите на форму, аккуратно выделите и удалите элементы списка на первой странице: выделили элемент, правой кнопкой мыши, меню `Edit->Delete`. Не надо о них сильно жалеть, они были добавлены нами в `design-time` для прототипирования интерфейса, т.е. создания его работающего макета. Когда мы уже реализовали полностью функционал приложения, они нам стали не нужны. После «подчистки» интерфейса запускаем приложение и вводим туда тестовые данные. Лучше всего для тестирования иметь эталонные наборы данных. Нельзя вводить данные случайным «тыканьем» в клавиатуру. Могут быть ошибки склейки данных в строку, ошибки хранения, форматирования, передачи и отображения. Создайте в электронной таблице небольшой набор эталонных данных (рис. 6.17), а потом по нему аккуратно введите данные посредством интерфейса. После этого внимательно изучите список и детальное представление. Если для каждого пункта тестирование про-

шло успешно, то можно говорить о правильно реализованном функционале нашего приложения.

	A	B	C	D	E	F	G	H
1								
2		№	Дата	Время	Длина ростка	Полив	Комментарий	Тест
3		1	01.07.2015	10:03	47	да	Пасмурно	ок
4		2	02.07.2015	10:15	50	нет	Поставил поддерживающий прутик	ок
5		3	03.07.2015	10:04	54	нет	Грунт пересох	ок

Рис. 6.17. Эталонные данные для тестирования

Создание такой таблицы из эталонных данных — занятие весьма серьёзное. Причин ошибок в современных программах много, причём на уровне профессионалов они возникают не только из-за невнимательности, но из-за необходимости постоянно модифицировать приложения. Представьте себе, если мы решим добавить количество листьев на ростке в качестве параметра, а признак полива решим удалить. Нам придётся какое-то время очень интенсивно поработать с проектом, причём изменять придётся код фрагментарно. Фрагментарно, значит «то там, то здесь». Легко можно допустить ошибку. Но она никогда не дойдёт до конечного пользователя, если после каждого изменения программы мы будем тщательно его тестировать. Тщательно, значит в строгом соответствии с планом тестирования и эталонными данными. В самой правой колонке у нас стоит признак прохождения всего теста. Мы будем считать, что тест признан успешным, если пользователь ввёл данные на страничке 3, а затем они были корректно показаны в и общем списке на страничке 1, и на страничке 2 для детального просмотра.

Возможно, вы больше сконцентрированы на разработке приложений. Тогда обязательно пригласите в команду человека с ярко выраженным критическим мышлением. Составьте ему такую таблицу или дайте ему возмож-

ность сделать самому, а потом попросите проверить работу приложения. Если такого подходящего человека нет, то придётся тестировать приложение самим. Иначе возникающие в процессе работы с программой ошибки могут сильно повлиять на ваш имидж крутого разработчика. Каким бы красивым интерфейс не был, в конечном итоге приложение оценивается по качеству его работы. И это — один из самых важных критериев. Существуют системы для автоматического тестирования, как интерфейса, так и программного кода приложения. Профессионалы высокого уровня весьма интенсивно используют их. Часто даже сам программный код пишется так, чтобы потом его можно было протестировать в автоматическом режиме. Как только вы почувствуете, что разрабатываемые вами приложения становятся все сложнее, а их тестирование в ручном режиме требуют всё больше времени, обязательно воспользуйтесь такими системами.

Нашим приложением уже можно пользоваться в практических целях. Если его постоянно держать на смартфоне в запущенном состоянии, то оно вполне может хранить записи о серии наблюдений. Однако смартфон может быть перегружен, тогда данные пропадут. Следующей задачей является обеспечение долговременного хранения данных. Помимо этого нам останется решить задачу передачи данных во внешние системы. Например, если вам потребуется создать отчёт по проведенным исследованиям в бумажном варианте или в виде презентации. Также данные могут быть использованы для обработки в других системах или помещения в централизованное хранилище для накопления статистики наблюдений.

6.9. Чтение сохраненных данных из файла

Чтобы обеспечить постоянное хранение данных, нужно уметь записывать их в файл и считывать их оттуда. Мобильное приложение здесь не исключение. Если вы используете Delphi/C++Builder/RAD Studio, то техника работы с файлами для мобильных приложений является такой же простой, как и при работе в ОС Windows. Базовых знаний о работе с текстовыми файлами на языке Pascal вам будет вполне достаточно. При запуске приложе-

ния мы будем считывать информацию из файла и загружать данные в массивы хранения, а потом и заполнять интерфейс пользователя. При каждом добавлении записи мы будем дописывать новую информацию в файл.

Подготовим проект к существенным изменениям. Сделаем копию текущей папки с проектом «Project 6.1» и назовём её «Project 6.2». Запустим среду разработки IDE и откроем проект из новой папки. Скомпилируем и запустим проект на выполнение. Материал, изложенный ниже, будет гораздо проще объектно-ориентированного программирования.

Сначала возьмёмся за самую лёгкую часть задачи — загрузка данных из готового файла. Создадим текстовый файл, например, при помощи текстового редактора Notepad. Запустите Notepad и введите с клавиатуры следующие данные (можно воспользоваться тестами из раздела 6.8), а затем сохраните его под именем «diary.txt» в папку «Документы»:

```
01.07.2015 10:03 47 да «Пасмурно»  
02.07.2015 10:15 50 нет «Поставил поддерживающий прутик»  
03.07.2015 10:04 54 нет «Грунт пересох»
```

Файл — текстовый, значит всё хранимое там — символьные значения. В каждой строке отдельные группы символов разделены пробелами. Первое значение — дата, второе — время, третье — признак полива, четвёртое значение — комментарий. Обратите внимание, что комментарий взят в парные кавычки. Если этого не сделать, то, например, значение во второй строке Поставил поддерживающий прутик будет восприниматься как три отдельные строковые значения. Поэтому мы заключили данную группу слов в кавычки.

А теперь нам предстоит большая задача — написать код для чтения текстовой информации из файла и разбиения её на отдельные строковые значения. Представим себе, как это делаем мы. Мы смотрим на строчку слева направо и воспринимаем строковое значение целиком до пробела. Потом пропускаем пробел и начинаем читать снова. Так мы воспринимаем слова

в тексте. Попробуем сделать то же самое, но в программном коде. Сначала запишем алгоритм на нашем естественном языке:

1. Найти и открыть файл, в котором хранятся наши данные.
2. Считать строку из файла.
3. Разбить строку на значения «дата», «время», «длина», «полив», «комментарий».
4. Добавить данные в массивы хранения.
5. Добавить элемент в список на интерфейсе.
6. Прodelать пункты 2–6 до тех пор, пока в файле есть строки.

Перед тем, как приступить к программированию, найдём подходящее место для этого. Не в смысле удобного кресла или стола, а процедуры отклика на правильное событие. Загружать данные из файла нам нужно только один раз в момент запуска приложения. Мы уже познакомились с событием `OnCreate`, но оно происходит слишком рано, в самый начальный момент запуска. Теперь мы рассмотрим в качестве кандидата событие `OnShow`. Событие также происходит один раз в момент отображения приложения на экране. Выберем форму и сгенерируем процедуру отклика на данное событие. Сначала напомним код, который будет открывать файл и выполнять чтение строк из него:

```
procedure TForm1.FormShow(Sender: TObject);
var
    FileName: string; // полное имя файла
    MyFile: TextFile; // файловая переменная
    BigString: string; // строка для чтения из файла
begin
    // получить полное имя файла с путём
    FileName:= TPath.GetDocumentsPath + PathDelim + 'diary.
    txt';
    // связать файловую переменную с именем файла
    AssignFile(MyFile, FileName);
    // открыть файл на чтение
    Reset(MyFile);
    // пока не конец файла
    while not Eof(MyFile) do
```

```
begin
  // считать целиком строку из файла в переменную BigString
  Readln(MyFile, BigString);
end;
// закрыть файл
CloseFile(MyFile);
end;
```

Найдём первый верхний раздел **uses** и впишем туда через запятую `System.IOUtils`, чтобы код скомпилировался без проблем. Данная книга посвящена созданию проектов в Delphi/C++Builder/RAD Studio, поэтому мы не будем подробно останавливаться на приведённом коде. Достаточно и вполне подробных комментариев. При необходимости нужно отдельно проработать соответствующие источники информации по языку Pascal в разделе «работа с текстовыми файлами». Введём данный код, откомпилируем проект и приступим к отладке. При Естественном, на интерфейсе ничего отображено не будет, т.к. мы написали код загрузки из файла без отображения на интерфейсе. Поэтому берём в руки отладчик, работу с которым мы уже рассмотрели ранее, и начинаем пошагово проходить данный код. Поставим «точку останова» (break point) при помощи клавиши F5 на строку:

```
while not Eof(MyFile) do
```

и запустим программу на исполнение кнопкой F9. При запуске автоматически возникнет событие `OnShow`, запустится процедура `FormShow`, и выполнение прекратится на точке останова. Мы остановимся в начале цикла. Сделаем шаг вперёд кнопкой F8 два раза, пройдя тем самым тело цикла один раз. После этого наведём курсор на переменную `BigString` и посмотрим её значение. Также можно посмотреть её содержимое в окошке `Local Variables` (рис. 6.18). Выполняйте прохождение цикла столько раз, сколько нужно. Несложно догадаться, что количество проходов цикла будет соответствовать количеству строк в файле. Вы должны заметить, как все три строки из файла будут последовательно загружены в переменную `BigString`.

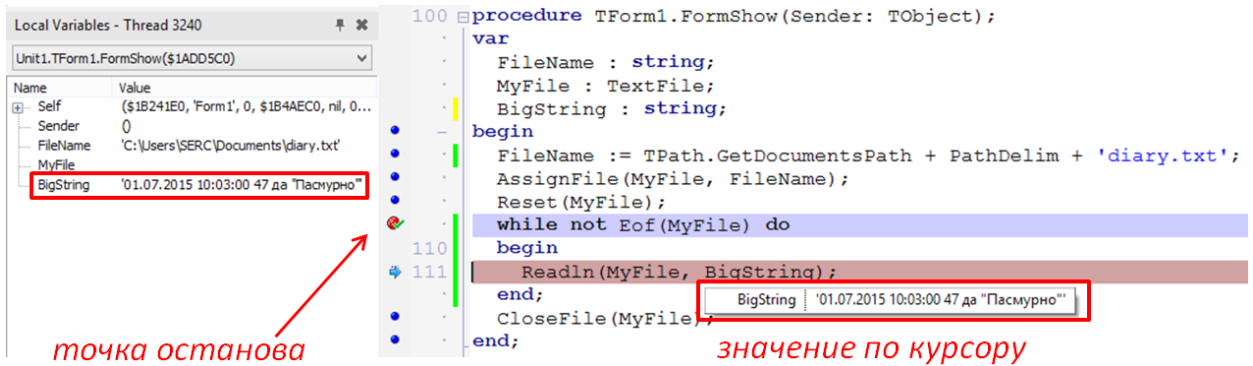


Рис. 6.18. Просмотр значения переменной

Обязательно проверим описанным выше способом значение переменной `FileName`. Если у вас возникает ошибка, то очень возможно, что вы промахнулись папкой и сохранили текстовый файл «diary.txt» в другое место. Значение переменной `FileName` «склеивается» из разных частей, где `TPath.GetDocumentsPath` есть путь к хранилищу документов. Поскольку мы создаём мульти-платформенное приложение, то на разных платформах это означает свою папку. Для ОС Windows мы получим папку «Документы», а для Android там будет нечто иное. Мы это узнаем, когда будем собирать приложение именно под мобильную платформу. `PathDelim` — тоже важный параметр. На разных платформах разделитель в пути к папке или полном имени файлов может быть различный, как «\», так и «/». Но об этом не надо задумываться, если использовать `TPath.GetDocumentsPath`.

Следующий пункт нашей обязательной программы — научиться разбивать длинную строку `BigString` на отдельные кусочки. Мы будем брать строку с самого начала и выбирать из неё символ за символом. Строка — это последовательность символов. Мы будем брать подряд символы до пробела и откладывать их в сторонку. Как только мы нашли первый пробел, отобранные символы есть первое строковое значение. Пропускаем пробел и опять выбираем символы. И так до самого конца, а последняя группа символов будет комментарием к записи наблюдения согласно нашей логике хранения. Рис. 6.19 иллюстрирует данный алгоритм.

Дополним код процедуры отклика FormShow в два приёма. Сначала добавим дополнительные переменные в раздел **var**:

```
// переменные для строковых значений
sDate, sTime, sLength, sWater, sComment: string;
i: integer; // целочисленная переменная
```

Мы будем дробить большую строку, а результаты дробления складывать для удобства в отдельные переменные строкового же типа. Префикс «s» перед названием означает «string» или «строка». Целочисленная переменная *i* будет играть важнейшую роль в разборе строки, храня в себе номер или индекс текущего символа. В теле цикла после строки Readln(MyFile, BigString); запишем следующий код, который будет идти до конца цикла:

```
// инициализация строк пустым значением
sDate:=''; sTime:=''; sLength:=''; sWater:=''; sComment:='';
// начинаем с 1-го номера в строке
i:= Low(BigString); // или i:= 1;
while BigString[i] <> ' ' do
    begin
        sDate:= sDate + BigString[i];
        i:= i + 1; // увеличиваем номер или индекс символа
    end;
i:= i + 1; // пропускаем пробел
while BigString[i] <> ' ' do
    begin
        sTime:= sTime + BigString[i];
        i:= i + 1; // увеличиваем номер или индекс символа
    end;
i:= i + 1; // пропускаем пробел
while BigString[i] <> ' ' do
    begin
        sLength:= sLength + BigString[i];
        i:= i + 1; // увеличиваем номер или индекс символа
    end;
i:= i + 1; // пропускаем пробел
while BigString[i] <> ' ' do
    begin
        sWater:= sWater + BigString[i];
        i:= i + 1; // увеличиваем номер или индекс символа
    end;
i:= i + 1; // пропускаем пробел
```

```

while i < High(BigString) do // или i < Length(BigString)
  begin
    if BigString[i] <> '»' then
      sComment:= sComment + BigString[i];
      i:= i + 1; // увеличиваем номер или индекс символа
    end;
  end;

```

Обсудим введенный код. Начало связано с инициализацией строковых переменных пустыми значениями. Присваивать переменным начальные значения — полезная и необходимая привычка. Всегда нужно быть уверенным в том, что храниться в переменных. «Пустая строка» означает отсутствие символов, её длина равна нулю.

Далее приступаем к разбору строки. Поочередно перебираем элементы, а `BigString[i]` есть текущий символ по номеру `i`. В классических строках Delphi/Pascal индекс элемента строки совпадает с его номером. На первом месте стоит элемент с индексом 1, на втором — с 2 и т.д. Но в других языках программирования принято индексировать массивы с 0. Чтобы никогда не ошибиться, лучше использовать функцию `Low(BigString)`, которая вернёт индекс самого младшего элемента в строке или массиве. Хотя в комментариях есть и другой, традиционный вариант, который можно встретить в учебниках по классическому языку Pascal.

Переходим к анализу первого цикла `while BigString[i] <> '»' do`. Пока очередной элемент строки не является пробелом, мы берём его и «приклеиваем» к строке `sDate`. Каждый раз мы присваиваем строке `sDate` её же, но с прибавкой в виде текущего символа. Таким образом, мы постоянно подрачиваем «хвост» `sDate` за счёт очередного элемента строки. Как только встретился первый пробел ' ', добавление символов прекращается. Текущим элементом остаётся пробел, поэтому выполняется `i:=i+1`, что есть увеличение текущего номера. Мы просто пропускаем пробел и не прибавляем этот бесполезный символ ни к одной из строк, т.к. он служит всего лишь разделителем слов в строке. Следующие циклы `while` работают абсолютно также за исключением название строк, к которым выполняется добавление символов.

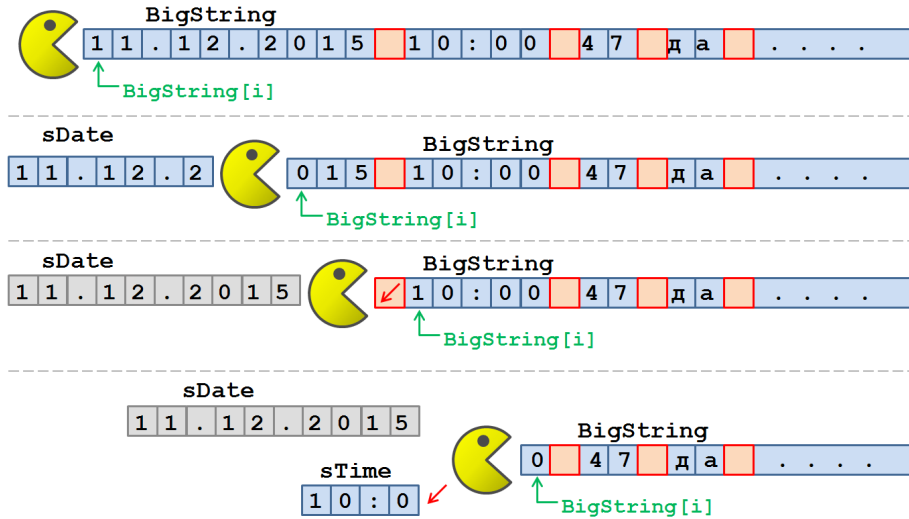


Рис. 6.19. Разбор строки на отдельные слова

Обратим внимание на последний цикл, где происходит склейка символов для комментария. Наша задача не просто взять все символы подряд, но отфильтровать парные кавычки ". Поэтому используется соответствующая конструкция с оператором `if`. По аналогии с использованием `Low(BigString)` применяется `High(BigString)`, что представляет собой индекс последнего элемента. В комментариях показано, что опять же классический вариант позволяет записать `Length(BigString)`. Но данный код будет работать только тогда, когда символы в строке считаются с 1. Тогда и длина строки, например, 5 совпадает с номером и индексом последнего элемента, что тоже 5. Однако для практических целей лучше использовать пару `Low(...)` и `High(...)`.

Обязательно воспользуйтесь техникой отладки, рассмотренной выше, для проверки работоспособности кода. Вооружаемся кнопками F5 (поставить точку останова) и F8 (шаг вперёд в режиме отладки) и начинаем выполнять обязательный этап проверки корректности работы алгоритм (рис. 6.20). Шаг за шагом мы видим, как наши строки накапливают символы, как точно и чётко работает алгоритм.

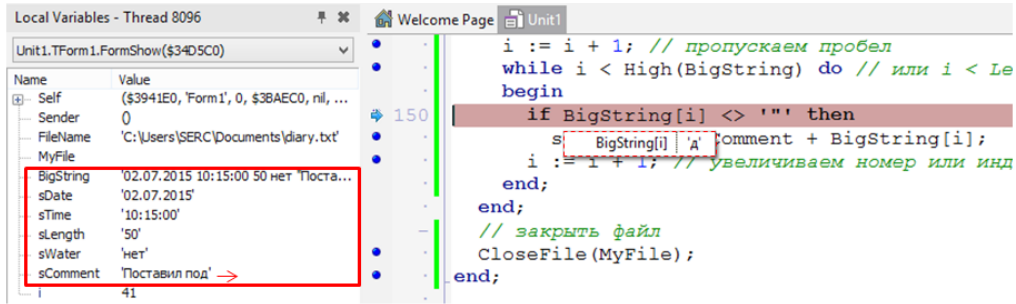


Рис. 6.20. Отладка алгоритма разборки строки

Только что мы освоили технику чтения строк из файла и разбор строки на элементы. У нас теперь в конце каждого прохода цикла есть строковые значения всех данных в переменных `sDate`, `sTime`, `sLength`, `sWater` и `sComment`. Нужно в конец каждого прохода занести их массивы хранения данных `arDate`, `arTime`, `arLength`, `arWater` и `arComment`, имея ввиду то, что значения мы получаем из файла в виде строк, а массивы хранят данные уже правильных типов — дата `TDate`, время `TTime`, вещественный `single`, логический `Boolean` и строковой `string`. Только последнее значение не требует преобразовывания, так как оно имеет совпадающий тип `string`. В остальных случаях потребуются преобразования. После преобразования и занесения данных в массивы можно будет приступить к созданию новых элементов списка `LisbBox1` на первой страничке `TabControl1`.

В раздел `var` введём новую переменную, которая будет хранить в себе количество считанных из файла строк. Соответственно, такое же количество новых элементов в список нужно добавить. Далее найдём ключевое слово `end;`, которое завершает тело большого цикла. Перед ним будем постепенно добавлять новый код:

```
procedure TForm1.FormShow(Sender: TObject);
var
    // пропущен код
    n: integer; // НОВЫЙ КОД—число считанных из файла строк
begin
    // пропущен код
    n:= 0; // НОВЫЙ КОД—инициализация
```



```

while not Eof(MyFile) do
  begin
    // пропущен код
    // здесь происходит чтение строки из файла
    // и её разбор на элементы
    // ...
    // НОВЫЙ КОД
    n:= n + 1; // увеличили кол-во прочитанных строк на 1
    // писать здесь!
  end;
// пропущен код
end;

```

Чтобы не приводить весь код процедуры снова, будем работать с его фрагментами. Это опять же полезное упражнение: не воспринимать код как «бесконечный поток слов», а видеть его структуру. Код выше показывает некий «скелет» или алгоритм «крупными мазками». Именно таким образом и нужно мыслить — видеть структуру кода, чувствовать главные принципы его работы, понимать общий ход выполнения программы. И лишь затем углубляться в нюансы конкретных операций. В теле цикла мы считали строчку, разобрали её по значениям и уже готовы заполнить массивы хранения. После строчки «писать здесь!» добавим:

```

// писать здесь!
// заполняем массивы данных с преобразованием
arDate[n]:= StrToDate(sDate);
arTime[n]:= StrToTime(sTime);
arLength[n]:= StrToFloat(sLength);
if sWater = 'да' then
  arWater[n]:= true
else
  arWater[n]:= false;
arComment[n]:= sComment;

```

Сохраняем, компилируем и запускаем проект. Логику работы интерфейса мы пока не сделали, поэтому воспользуемся средствами отладки IDE, чтобы проконтролировать работоспособность кода. В этот раз мы не будем пошагово исполнять приложение, т.е. нам требуется просто увидеть значения массивов хранения данных. Изучим рис. 6.21, где показаны различные способы проверки значений массива. Сначала поставим точку останова

на строчке `CloseFile(MyFile);`. Запустим приложение. При запуске сработает событие `OnShow` перед показом главной формы приложения. Процедура `FormShow` начнёт свою работу, файл откроется, строки считываются, произойдёт разбор и заполнение массивов. Остановимся мы на закрытии файла, когда вся работа уже проделана.

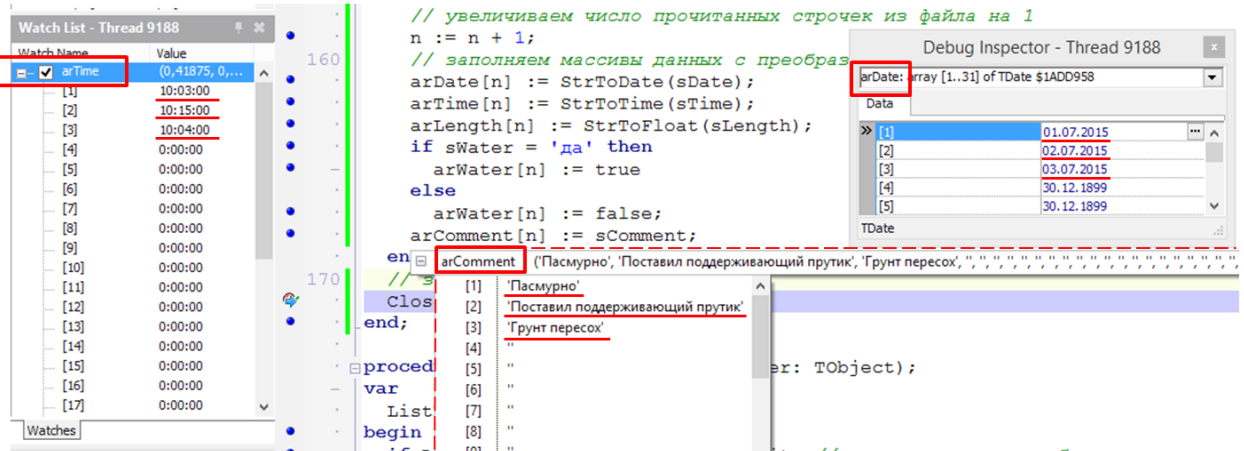


Рис. 6.21. Проверка загрузки данных в массивы

Теперь узнаем, что же такого у нас в массивах. Можно обратиться к панели Watch List. Если мы кликнем правой кнопкой мыши и выберем Add Watch, а затем введём `arTime`, то данная панель отобразит содержимое массива. Первые три значения совпадают с тем, что у нас есть в текстовом файле. Однако такую операцию можно проделать и быстрее. Наведём курсор мыши на массив `arComment` непосредственно в тексте. Сначала появится горизонтальная строка с содержимым массива, а если кликнуть `[+]` рядом с `arComment`, то раскроется и вертикальное окно, где будут показаны элементы массива с номерами. Но есть и еще один способ: вызывать окно Inspect посредством меню `Run->Inspect`. На рис. 6.21 его окно показано в правом верхнем углу. Там тоже можно увидеть значения массива. Выбор способа зависит от задачи. Если как у нас она простая, то достаточно и «курсорной» проверки. В сложных вариантах пошаговой отладки с одновременным контролем нескольких массивов, конечно, нужно применять

более развитые способы. Для большинства задач разработки приложений без сложных математических алгоритмов достаточно понимания кода при использовании самых простых технологий отладки.

Завершающим этапом в загрузке данных из файла будет заполнение списка на первой страничке. У нас есть несколько вариантов добавления кода на заполнение списка. Это можно сделать в главном цикле `while`, можно создать отдельный цикл с заданным числом повторений, т.к. мы уже знаем, сколько элементов в массиве. Можно затем ещё и создать отдельную функцию добавления нового элемента. Конечно, лучше максимально (в пределах разумного) дробить код на функции. Мы получаем более понятную структуру, а также повышаем универсальность. Если бы у нас уже была такая функция, то мы просто вызвали бы её. Был удачный момент, когда мы добавляли новый элемент в список по кнопке `[+]` на главной страничке, но мы этого не сделали. Теперь расплачиваемся написанием похожего кода. Но хотя бы не будем писать «всё в одну кучу», а создадим отдельный цикл. В раздел переменных процедуры `FormShow` добавим переменную `j: integer;`, а сразу после `CloseFile (MyFile)` введём следующий код:

```
// добавление элементов в список на 1-й страничке
for j:= 1 to n do
begin
  NewListBoxItem:= TListBoxItem.Create(ListBox1);
  NewListBoxItem.Parent:= ListBox1;
  NewListBoxItem.StyleLookup:= 'listboxitembottomdetail';
  NewListBoxItem.Height:= 44;
  NewListBoxItem.ItemData.Accessory:= TListBoxItemData.
    TAccessory.aMore;
  NewListBoxItem.ItemData.Text:= DateToStr(arDate[j]) + ' '
    + TimeToStr(arTime[j]);
  NewListBoxItem.ItemData.Detail:= 'Длина ростка ' +
    FloatToStr(arLength[j]) + ' см';
end;
```

Чтобы не было ошибок, также добавим в раздел `var` переменную `NewListBoxItem: TListBoxItem` для хранения ссылки на новый элемент в списке `ListBox1`. Если у вас был соблазн найти и скопировать код из процедуры `SpeedButton1Click`, то нет ничего плохого, если вы это сде-

ляли. Компилятор заставит вас переправить `NewIndex` на `j`. Но сам факт копирования–вставки фрагмента кода с минимальными изменениями есть повод серьезно задуматься. Дублирование кода порождает проблемы в будущем. Два почти одинаковых фрагмента кода будут постоянно путать программиста типа «где-то подобное я уже видел». А если нужно будет внести изменения, то придётся искать по всей программе эти «почти одинаковые» фрагменты и делать аналогичные правки. И если сейчас нам это «сошло с рук», то в дальнейшем лучше потратить некоторое количество времени и всё-таки создать функцию добавления нового элемента, которая будет получать три значения: дату, время наблюдения и длину роста.

После этой нотации вернёмся в проект, сохраним и запустим приложение. Всё должно работать хорошо. В процессе проверки работы уже при помощи интерфейса пользователя можно в текстовом файле добавить ещё одну строчку и ещё раз убедиться в работоспособности приложения. Не забудьте также проверить и кнопку добавления нового элемента [+], и просмотр детальной информации, т.е. выполнить полное тестирование. Лучше на каждом этапе быть полностью уверенным, чем через неделю интенсивной работы обнаружить «поломку» там, где раньше все работало. Тогда сложнее будет всё вспомнить и починить. Почему столько слов расходуется на убеждение читателя поддерживать хорошую структуру кода и постоянно тестировать приложение? Потому что такая «культура производства» и есть признак профессионализма. Но это не имиджевая составляющая, а основа качества и безошибочной работы создаваемых программных продуктов. А теперь с тем же уровнем качества перейдём к завершающей стадии — запись данных в файл при добавлении нового элемента.

6.10. Сохранение данных в файл

При добавлении новой записи о наблюдении, что выполняется по кнопке [+] на главной страничке, данные уже записываются в массивы хранения, а также добавляется новый элемент в список ListBox1. Не хватает лишь добавления записи в текстовый файл Diary.txt, чтобы при последующем запуске приложения были загружены новые данные. Кроме этого, при необходимости экспорта данных во внешнюю систему удобно иметь уже готовый текстовый файл.

Найдём в тексте программы процедуру SpeedButton3Click и заново изучим её код. Процедуру можно разделить на три логических блока: 1) запись данных в массивы хранения; 2) добавление нового элемента в список; 3) навигация. Перед навигацией самое время сделать до-запись нового элемента в файл. Добавим дополнительные переменные в раздел var:

```
MyFile: TextFile; // файловая переменная
FileName: string; // полное имя файла
MyString: string; // строка для записи
```

Затем перед навигацией вставим следующие строчки с комментариями:

```
// запись в файл
// получить полное имя файла с путём
FileName:= TPath.GetDocumentsPath + PathDelim + 'diary.txt';
// связать файловую переменную с именем файла
AssignFile(MyFile, FileName);
Append(MyFile); // открыть файл на до-запись
// склеить строку с данными для записи
MyString:= DateToStr(arDate[NewIndex]) + ' ' +
TimeToStr(arTime[NewIndex])
+ ' ' + FloatToStr(arLength[NewIndex]);
if arWater[NewIndex] = true then
MyString:= MyString + ' ' + 'да' + ' '
else
MyString:= MyString + ' ' + 'нет' + ' '
MyString:= MyString + '»' + arComment[NewIndex] + '»';
Writeln(MyFile, MyString); // записать строку в файл
CloseFile(MyFile); // закрыть файл
```

Комментарии к коду вполне полным образом описывают его смысл. Остановимся на некоторых моментах. Первое, это использование условного оператора **if** для подстановки «читаемого человеком» (human readable) значения. Можно было оставить и «true»/«false». Конкретно в нашем случае это вполне приемлемо, а «да»/«нет» не намного лучше. Но в общем случае нужно быть готовым и уметь переводить значения из «понятного компьютеру», т.е. «машинного представления» на «человеческий язык». Тогда файл Diary.txt будет понятен не только программистам, но и пользователям.

Второе, вспомним логику добавления парных кавычек " в начале и конце строки с комментариями. Таким образом, мы даём понять и человеку, и другой программе, что все эти разделённые пробелами слова представляют собой единое строковое значение. Попробуйте открыть файл Diary.txt в приложении, например, Microsoft Excel в варианте с кавычками и без кавычек и вы почувствуете разницу. Сначала откроем файл Diary.txt как есть. Потом откроем файл в Notepad и вручную уберем парные кавычки, затем снова откроем в Microsoft Excel. Во второй раз приложению Microsoft Excel будет гораздо сложнее справиться с задачей корректного чтения. Всегда при проектировании формата хранения данных в файле, особенно в текстовом, продумывайте степень «читабельности» информации и возможности открытия другими программными продуктами. Откроем файл в Notepad и вернём кавычки на место.

6.11. Добавление файлов в проект при развёртывании

Мы спроектировали наше приложение так, что при запуске оно обязательно выполняет загрузку данных из файла `Diary.txt`. А что будет, если наше приложение не найдёт данный файл при запуске? Нет ничего проще провести эксперимент: просто измените имя текстового файла на другое и запустите приложение. Всегда нужно пробовать самому, не дожидаясь, когда об ошибке вам сообщат «благодарные» пользователи! В нашем случае программа просто «вылетит». По-хорошему, перед открытием файла нужно было поставить проверку вида:

```
if not FileExists(FileName) then ...
```

Что поставить после `then` в случае, когда файл с данными не обнаружится? Вопрос интересный и не имеет однозначного ответа. Первое, что приходит в голову, это — создать файл программными методами и продолжить работу с приложением, где изначально никаких данных нет. Но будет ли это правильным? Представьте реальную ситуацию, когда велись наблюдения, а потом файл исчез. Возможно, был переименован или перемещён. Приложение опускается, заново создаёт файл и продолжает работу. Пользователь продолжает записывать свои наблюдения. Потом выясняется, что предыдущий файл нашёлся. Что делать? Можно ли как-то их объединить, чтобы ничего не пропало? Профессионалы задумываются и об этом. Чтобы не играть с читателем в прятки, мы приведём некий вполне уместный код вместо `Append(TextFile)`, но прежде использования сложных конструкций обязательно продумайте все возможные варианты работы приложения:

```
// вместо Append(TextFile)
if not Exists(FileName) then
  Rewrite(TextFile)
else
  Append(TextFile);
```

В случае, если файл не существует, то мы создадим его заново и откроем на запись. Если существует — откроем файл на дописывание записей в конец.

Теперь рассмотрим более общий вариант решения задачи, когда вместе с приложением должен поставляться текстовый файл. Для настольных приложений достаточно скопировать результирующий exe файл вместе с текстовым. Также можно создать инсталляционный пакет, который будет в себе содержать не только exe, но и txt-файл. Для мобильных приложений на платформе Android среда разработки Delphi/C++Builder/RAD Studio формируют файл apk, который можно скопировать на устройство и установить приложение. Размещается этот файл обычно в папке: [Project Folder]\Android\Debug\Project1\bin\Project1.apk. Однако нет никакой необходимости его искать, копировать и устанавливать на мобильное устройство. Среда разработки сделает это всё великолепно. Подключим смартфон на базе Android к компьютеру через USB, выберем целевое устройство и запустим приложение на исполнение (рис. 6.22).

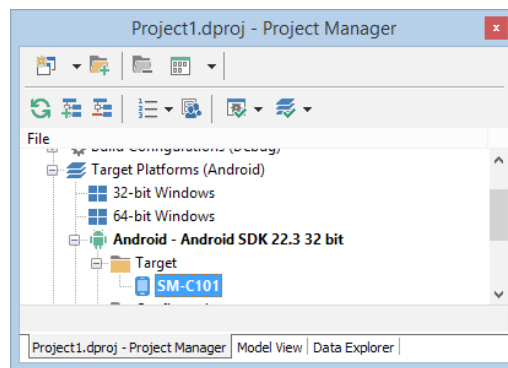


Рис. 6.22. Выбор целевого устройства

После запуска будет сформирован файл apk, а потом произойдёт установка и запуск приложения на мобильном устройстве. Но без Diary.txt приложение не будет работать, что мы уже обсудили. Нам необходимо сделать так, чтобы вместе с приложением на мобильное устройство был отправлен файл Diary.txt. Для этого в IDE есть специальный сервис Deployment Manager, вызывать который нужно из главного меню Project->Deployment. После этого нужно нажать кнопку добавления файлов, выбрать файл на ра-

бочем компьютере под управлением MS Windows, а затем вручную, поставив курсор в нужную ячейку ввести `assets\internal` (рис. 6.23).

добавить файл в проект

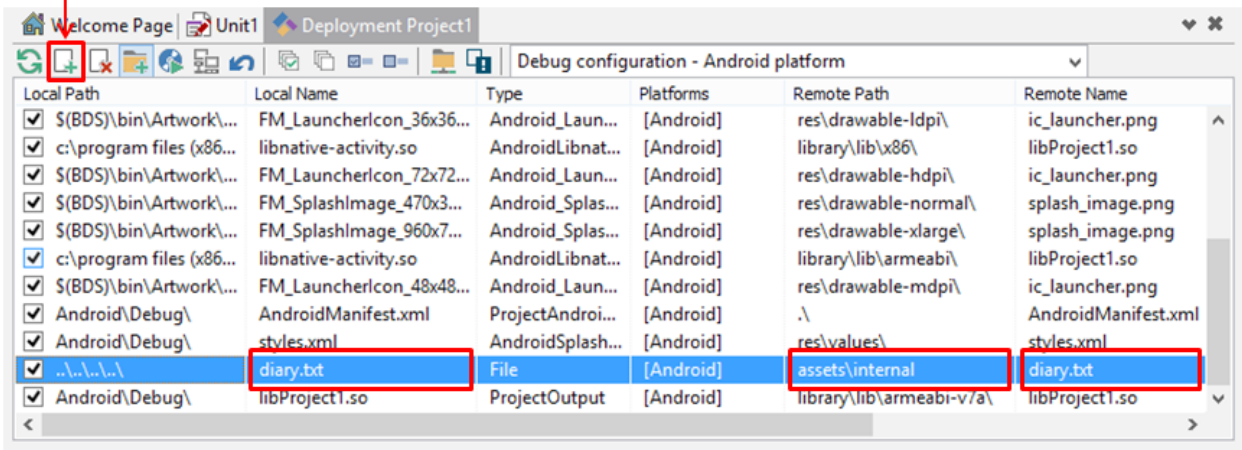


Рис. 6.23. Менеджер развёртывания — *Deployment Manager*

Теперь файл при развёртывании на мобильном устройстве попадёт в то место, которое определяется `TPath.GetDocumentsPath`. Если вам интересно, что же хранится в этой переменной, то можно поставить точку останова на нужной строчке и запустить приложение в режиме отладки, т.к. можно также пошагово отлаживать приложения, запущенные на мобильном устройстве. Однако не рекомендуется часто использовать отладку непосредственно на мобильном устройстве: развёртывание и запуск на смартфоне или планшете занимает много времени. Лучше разрабатывать и отлаживать под Windows, а уже финальное тестирование проводить на реальном устройстве.

6.12. Экспорт накопленных данных

Созданное нами мобильное приложение может с успехом использоваться для накопления данных о некоем явлении во времени. Это может быть рост фасоли, количество отжиманий, уровень воды в реке весной, количество осадков, количество уничтоженных в игре монстрах, построенных замков и т.д. Если мы планируем некую исследовательскую работу, то она, прежде всего, должна быть актуальной. Громкость лая собаки при виде сосиски не есть то, что приведёт в восторг преподавателя биологии. А вот большое хорошее исследование с использованием мобильного приложения, проведённое совместно с вашими одноклассниками, посвященное какому-либо социально-экономическому вопросу, могут не только прославить участников, но и реально принести пользу. Например, количество посетителей городского парка. Или число пешеходов, использующих конкретную дорогу, или пассажиро-поток на муниципальном транспорте. Каждый из учеников получает задание в определенный промежуток времени сидеть на лавочке у входа в парк и жать на одну кнопку каждый раз, когда посетитель заходит. И один раз, когда выходит. Потом данные от нескольких учеников объединяются. Получается общая картина, которую затем и интересно изучать и анализировать. Но для этого наше приложение должно уметь выгружать данные с мобильного устройства.

Для передачи данных из устройства есть несколько подходов:

- загрузка данных на web-сервер;
- использование облачных сервисов хранения данных;
- взаимодействие с базой данных;
- использование Share Sheet;
- выгрузка файла с данными;
- и др.

Сейчас мы рассмотрим самый универсальный из простых и самый простой из универсальных. Мы будем использовать Share Sheet, т.е. встроенные

возможности мобильной по передачи данных (изображений, текста и т.д.) в другие системы. Возможно, вы уже пользовались данным сервисом, когда делали фотографии и делились ими с друзьями через социальные сети. Конечно, мы не собираемся размещать отчёт о наблюдении в социальные сети. Нам всего лишь достаточно одной функции из Share Sheet — передачи текста через электронную почту. Мы просто отправим содержимое файла Diary.txt по почте на любой e-mail адрес. Потом уже на любом компьютере мы сможем получить данное электронное письмо и распорядиться его содержимым по своему усмотрению — вставить в электронную таблицу, проанализировать, построить график, вставить в презентацию и т.д.

Сейчас сделаем небольшое упражнение по модификации интерфейса. Неоднократно говорилось, что код никогда не пишется от начала до конца. Сначала пишется некий работающий прототип, а потом код постепенно добавляется. Часто достаточно трудно найти нужное место и аккуратно, ничего не испортив, вставить нужный код. Также и с интерфейсом, нужно уметь его улучшать, добавляя новые элементы управления. Перейдём на первую страничку нашего TabControl1 и сконцентрируем внимание на верхней инструментальной панели. Выделим Label1 и установим свойство Align в None, тем самым получив возможность сделать её немного уже. Теперь у нас появилось свободное место, куда мы и добавим новый компонент TSpeedButton (рис. 6.24). Установите ему свойство StyleLookup в значение actiontoolbarbutton, а затем верните свойство Align компоненту Label1 в значение Client. Теперь у нас новая кнопка, которая скоро будет выполнять полезную функцию.

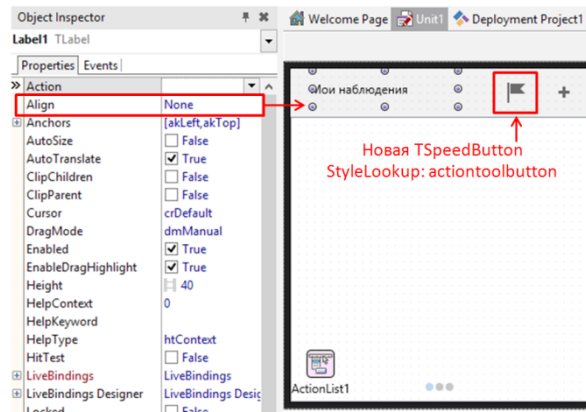


Рис. 6.24. Модификация интерфейса

Для реализации передачи данных из мобильного приложения посредством других систем, а в нашем случае это — e-mail, достаточно добавить одно стандартное действие. Выделим новую кнопку и из выпадающего списка Action в Object Inspector выберем New Standard Action->Media Library->TShowShareSheetAction (рис. 6.25).

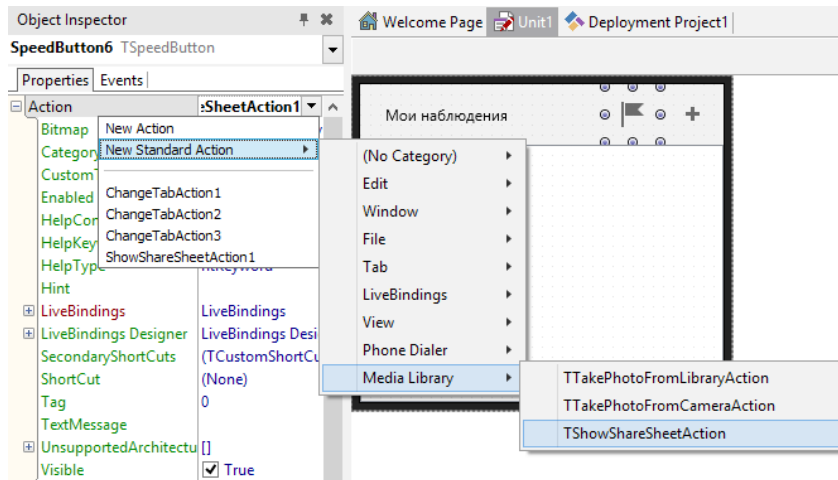


Рис. 6.25. Выбор стандартного действия Show Share Sheet

После этого раскроем [+] рядом со словом Action и ознакомимся со свойствами данного действия. Главным из них является TextMessage. Только не надо задавать его в Object Inspector, т.к. мы будем формировать его программно, загружая из файла Diary.txt. Перейдём на закладку Events и дважды щёлкнем в строчке с OnBeforeExecute. В созданном прототипе процедуры-отклика введём следующий код:

```
procedure TForm1.ShowShareSheetAction1BeforeExecute(Sender:
TObject);
begin
ShowShareSheetAction1.TextMessage:= 'Hello from mobile!';
end;
```

Вот тут протестировать приложение на Windows уже не получится. Можно скомпилировать и запустить приложение под данную платформу, но нажатие на кнопку ничего нам не даст в плане полезного эффекта. Выберем мобильную платформу, например, Android (рис. 6.22) и запустим приложение. После запуска мы увидим новую кнопку с несколько иной картинкой, т.к. стилизация интерфейса происходит в момент сборки под конкретную платформу. Для Android кнопка для вызова Share Sheet должна выглядеть именно так. Смело нажимаем на данную кнопку, выберем сервис электронной почты, вводим адресат и посылаем сообщение. Проверяем ящик входящих сообщений введённого адреса, сообщение должно быть доставлено. Рис. 6.26 показывает снимки экрана смартфона, на котором были проделаны описанные выше действия по передаче строки из приложения через Share Sheet.

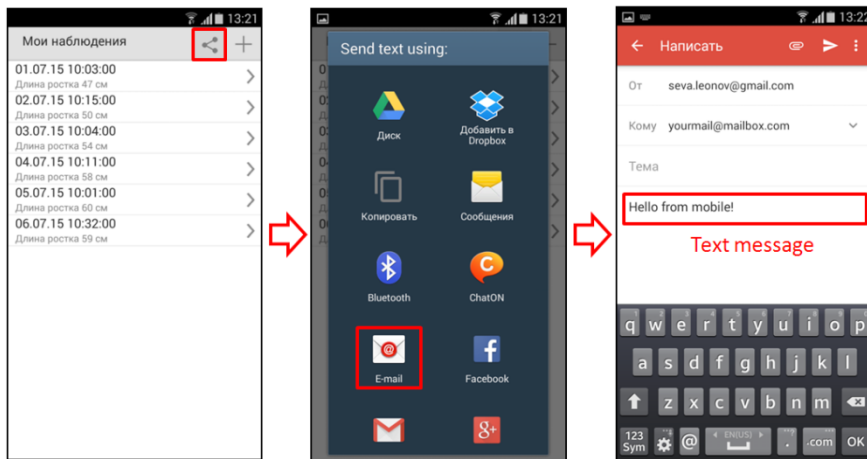


Рис. 6.26. Передача информации через Share Sheet

Остаётся сформировать не тестовое сообщение, а реальный отчёт о наблюдениях. Пойдём самым простым путём, который часто бывает самым эффективным. Откроем и прочитаем информацию из файла Diary.txt в текстовом виде, а затем пошлём её в теле письма. Проверим работоспособность приложения в данной части, а затем проведем его финальное тестирование.

Мы уже научились работать с текстовым файлом «классическими» методами Pascal. Но Delphi — объектно-ориентированная среда разработки, поэтому можно пользоваться и более мощными и универсальными средствами. В следующем разделе изучим объектно-ориентированный метод работы с текстовыми файлами.

6.13. Объектно-ориентированная работа с файлами

Что мы знаем об объектно-ориентированном программировании в Delphi? Мы уже выучили некоторые основные понятия, такие как «класс» и «объект» (экземпляр). Класс определяет чертёж или схему сборки, по которому «конструктор» создаёт объект. Если нам нужно получить полезный эффект, то мы создаём объект класса и пользуемся им в своих целях. Любой компонент это тоже объект определённого класса. Есть классы `TLabel`, `TSpeedButton`, `TListBoxItem` и многие другие. Нет большой разницы в том, объект какого класса создаётся. Когда объект создан, можно использовать его свойства и его функции.

В файле `Diary.txt` хранятся строки. Строк в файле много. Для работы с массивом строк есть специальный класс `TStringList`. Чтобы с его помощью можно было обработать массив строк, нужно создать объект данного класса. Для управления созданным объектом используются специальные переменные. Эти переменные хранят ссылки на объекты, посредством которых можно обращаться к объектам. Считайте, что ссылка — это пульт управления от объекта-робота. Через пульт управления всегда можно послать нужную команду роботу, а он её немедленно исполнит (рис. 6.27).

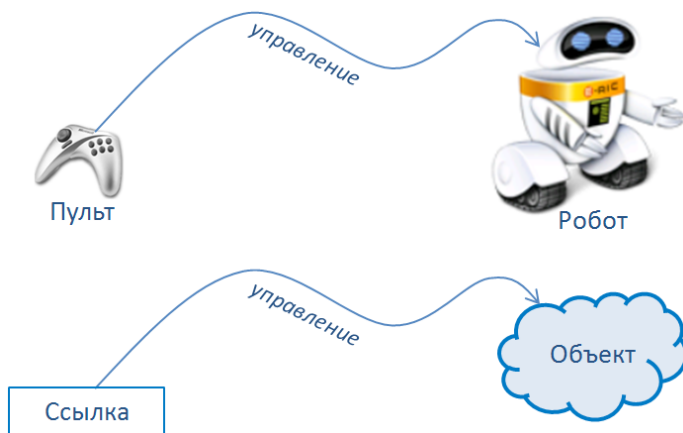


Рис. 6.27. Управление объектом через ссылку

Итак, действующие лица нашего спектакля таковы:

- имя файла
- класс TStringList
- свойства класса
- функции (методы) класса
- объект класса TStringList
- «создатель» объекта
- «уничтожитель» объекта

В этом списке мы видим несколько новых персонажей. Начнём с «уничтожителя» или терминатора. Объекты создаются и объекты уничтожаются. Созданный объект выполняет полезную работу, но он располагается в памяти, поэтому потребляет определённый ресурс. Обычно с объектами поступают так: создают объекты, используют их полезные свойства и функции, а затем уничтожают, освобождая тем самым ресурсы для других объектов и приложений. Если вспомнить раздел 6.7, то там мы создавали объект класса TListBoxItem при помощи «создателя» — «конструктора», но не уничтожали его. Действительно, мы не делали этого, т.к. он становился новым элементом списка, поэтому уничтожать его было нельзя. А в нашем случае объект «массив строк» после выполнения своих главных функций — подготовки текста к отправке — нам больше не нужен, и мы его уничтожим.

Появился ещё один незнакомый термин: метод класса. Если мы посмотрим на класс формы TForm1, а это легко сделать, поднявшись на самый верх Unit1.pas, то легко можно увидеть множество разных процедур, принадлежащих классу. Процедура или функция класса называется «метод» класса. Когда мы говорим «метод» или «вызов метода», то сразу понятно, что речь идёт о процедуре или функции какого-то класса. Так было сделано для удобства.

Теперь мы готовы записать код всех процедуры отклика на нажатие кнопки «поделиться»:


```

procedure TForm1.ShowShareSheetAction1BeforeExecute(Sender:
TObject);
var
  FileName: string; // полное имя файла
  StrList: TStringList; // переменная для ссылки
begin
  // получаем полное имя файла
  FileName:= TPath.GetDocumentsPath + PathDelim + 'diary.
txt';
  // создаём объект класса и запоминаем ссылку на него
  StrList:= TStringList.Create;
  // вызываем метод для загрузки строк из файла
  StrList.LoadFromFile(FileName);
  // формируем текст на передачу через Share Sheet
  ShowShareSheetAction1.TextMessage:= StrList.Text;
  // уничтожаем объект
  StrList.Free;
end;

```

Данный код достаточно прост для понимания, а первую строчку мы уже разбирали, когда работали с текстовыми файлами «классическим» для языка Pascal способом. Теперь мы работаем с помощью объектно-ориентированного программирования. Создаём при помощи «конструктора» Create объект нужно нам класса. Дальше вызываем его метод LoadFromFile, чтобы загрузить все строки из него в память. Затем формируется текст на отправку содержимого списка строк в Share Sheet. В конце работы процедуры объект уничтожается вызовом метода Free. Здесь мы рассмотрели книжный вариант работы с объектами классов. Более надёжное и предсказуемое выполнение приложения обеспечивает использование конструкций по обработке исключительных ситуаций. Данный материал выходит за рамки этой книги, но, пожалуйста, найдите информацию по исключениям try — except и try — finally в других источниках.

6.14. Развитие проекта

Мы только что завершили достаточно серьёзный проект, а созданное приложение можно с успехом использовать в исследовательской работе. Но если разобраться, то мы не закончили проект в том смысле, в котором создаются программные продукты. Единственная причина, по которой проекты завершаются и больше не развиваются, это — их ненужность. Если какой-либо продукт действительно нужен пользователям, то он постоянно совершенствуется. Конечно, после большой и интенсивной работы хочется какое-то время отдохнуть. Но всегда по завершению проекта нужно уметь посмотреть в будущее, что еще можно сделать. Не надо воспринимать это как «не успел», «не доделал», «не сумел». Постоянное добавление новых возможностей и есть смысл работы программиста. Представьте себе, что ваше приложение приобрело много фанатичных пользователей, которые жаждут новых более мощных в функциональном плане версий. Подумаем, какие возможности в таком случае мы могли бы добавить к нашему приложению:

- Редактирование записи о наблюдении в случае, если при внесении была допущена ошибка.
- Удаление записи, если она была внесена случайно.
- Возможность делать больше 31 записи о наблюдениях.
- Построение графика изменения наблюдаемой величины.
- Напоминание о необходимости провести наблюдение.
- Добавление фотографий к записи о наблюдении.
- Облачное хранение информации.

Некоторые возможности вы можете реализовать уже сейчас: редактирование и удаление записи. Редактирование — это смесь «просмотра» и «добавления новой». Увеличение числа записей о наблюдаемом явлении больше 31 требует умения пользоваться динамическими массивами. Динамические массивы или даже структуры хранения с изменяемым размером — очень интересная и полезная тема, прекрасное направление разви-

тия ваших навыков. Строить графики мы учились в одном из предыдущих разделов, а сервис напоминания мы освоим в следующем. Работа с фотографиями и облачное хранение выходят за рамки данной книги, также как и хранение информации в базе данных. В соответствующей главе будут приведены ссылки на полезную литературу, проливающую свет на данные вопросы.

Мобильное приложение для изучения поэзии

7.1. Прототип интерфейса

Мы все любим стихи. Декламация стихотворений — отличный способ произвести впечатление на эстетически развитую публику. Разучивание стихов наизусть развивает грамотность речи, пополняет словарный запас, улучшает дикцию и просто приносит массу положительных эмоций. Если у вас плохое настроение, то прочтите вслух или мысленно любимое стихотворение значительно улучшает мироощущение. Успокоить эмоции, отвлечься, расслабиться — для всего этого подходит декламация правильно подобранного стихотворения. А если речь идёт об активном участии в жизни учебного заведения или выполнения домашних заданий, то удобное мобильное приложение для облегчения заучивания стихов весьма полезно.

После сложного и длинного предыдущего проекта, посвященного решению исследовательской задачи, мы немного развлечёмся, сделав достаточно простую и весёлую программу с добавлением элемента игры. Но что самое главное, мы хорошо разберёмся с темой обработки текстовой информации. Каким бы ни было приложение, если только это не примитивная игра, всегда требуется работа с текстом. Особенно это касается серьёзных бизнес-приложений: текстовых редакторов, баз данных документов, электронных таблиц, автоматических переводчиков, систем распознавания текста или речи, словарей, баз знаний и т.д. А если речь идёт о создании искусственного интеллекта, то вопрос становится одним из наиболее актуальных. Чем бы вы в дальнейшем не занялись в области информационных технологий, полученные знания вам обязательно пригодятся.

Начнём с совсем простого приложения, которое отображает нам заранее введённый стих. Создадим папку «Project 7». Запустим Delphi IDE, далее File->New->Multi-Device Application — Delphi, а в появившемся мастере выберем Blank Application (рис. 7.1). Можно выбрать и другие шаблоны, но они лишь имеют уже некоторое количество размещённых компонентов на главной форме. Для развития собственных навыков лучше всё сделать самими с начала. Кроме того, наш текущий проект не будет иметь очень сложный интерфейс, поэтому работа по подготовке дизайна не будет долгой. Сохраните проект под именем «Роем.dproj» в папку «Project 7».

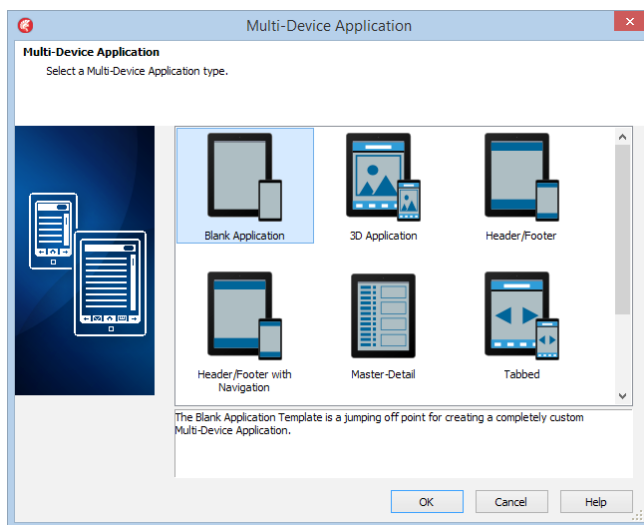


Рис. 7.1. Выбор шаблона проекта

Внимательно выполним последовательность действий по размещению и настройке визуальных компонентов:

- Разместите на форме TToolBar, он займёт верхнюю позицию;
- Выделите ToolBar1 и разместите на нём компонент TLabel;
- Свойству Align компонента Label1 выберите значение Left из выпадающего списка;
- Свойству Margin->Left компонента Label установите значение 10;

- Разместите на форме TMemo, свойство Align установите в Client;
- Найдите у TMemo в Object Inspector свойство Lines (строки) и нажмите кнопку [...];
- В открывшемся мастере скопируйте текст стихотворения;
- Измените свойство Text компонента Label1 так, чтобы оно содержало название стиха.

Сверим результаты с тем, что показано на рис. 7.2. Будем использовать стихотворение «Золотая осень», автор которого Борис Пастернак. Конечно, можно подобрать любое другое по вашему усмотрению. Выбранное стихотворение должны быть достаточно длинным и сложным для запоминания, чтобы интересно было тестировать приложение. Обратите внимание, что компонент TMemo автоматически добавляет полосу прокрутки, если текст не уместается в текущий экран. Когда вы будете добавлять текст из буфера обмена, возможно, придётся поработать в мастере добавления строк, который появляется при нажатии на [...]. Нужно будет отформатировать абзацы, убрать лишние строки и т.д. К сожалению, мастер добавления строк не работает безукоризненно, но проблемы решаются. Сохраним и запустим приложение. Также попробуйте запустить его на мобильном устройстве.

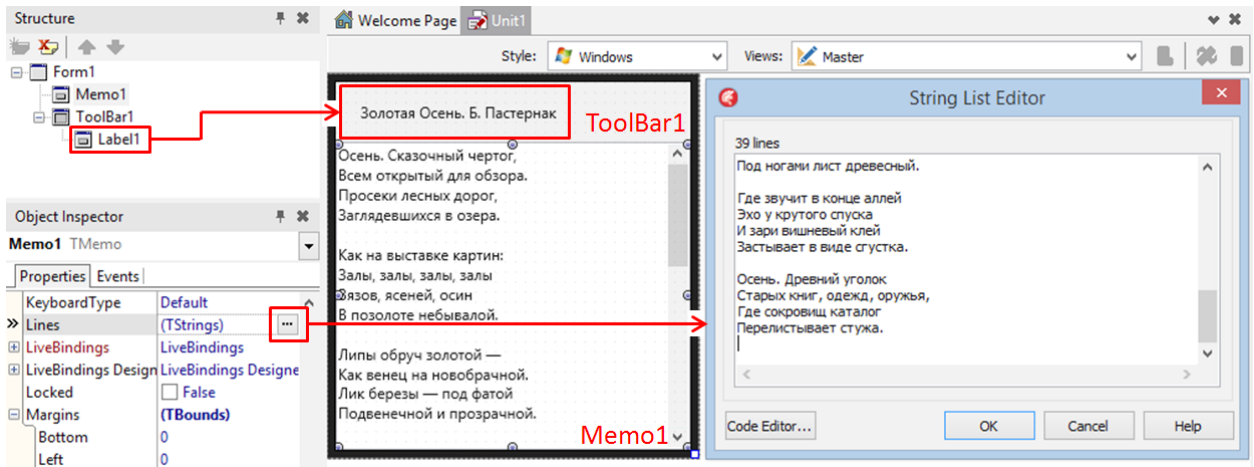


Рис. 7.2. Прототип интерфейса

В процессе работы с пробным вариантом приложения нужно обратить внимание, что показанный текст стихотворения в компоненте ТМето можно отредактировать, т.к. по умолчанию у него включен режим редактирования. Пока отключим данную возможность компонента ТМето, для чего в Object Inspector свойство ReadOnly (только чтение) установим в true.

После пробного запуска приложения мы обрели уверенность в своих силах. Помня о важности хорошего и понятного кода, сразу «почистим» проект. Это нужно делать сразу, т.к. по мере роста сложности проекта путаница будет только нарастать. Аккуратно пройдемся по именам компонентов в проекте и изменим их на более понятные согласно таблице ниже:

Свойство Name по-умолчанию	Корректное имя
Form1	fmMain
ToolBar1	tbTop
Label1	lbTop
Memo1	meFull

Далее мы уже будем придерживаться корректной формы записи имён компонентов.

7.2. Алгоритмы заучивания и их реализация

Смысл работы нашего приложения будет достаточно прост в плане алгоритма. Мы будем изначально показывать пользователю весь текст стиха, а затем скрывать часть текста, причем методику сокрытия можно придумать любую. Например, при заучивании стиха из обычной книги часто люди открывают лишь первые слова, загибая край следующей страницы. Мы же программно можем придумать любой алгоритм сокрытия или маскирования слов. Для начала сделаем два варианта:

- Оставляем только первые слова в каждой строчке;
- Скрываем текст через строчку.

Для этих целей наш интерфейс следует переделать. Выделим `meFullPоеm`, затем вырежем его в буфер обмена `Ctrl+X` или через контекстное меню. Разместим на форме компонент `TTabControl`, сразу переименуем его в `tcMain`, а свойству `Align` зададим значение `Client`. Свойству `TabPosition` зададим значение из выпадающего списка `Bottom`. После этого по правой кнопке мыши добавим три странички и на первую вставим `meFullPоеm` из буфера обмена. На вторую и третью странички также поместим по компоненту `TMemo`, задав им свойства `Align` как `Client`. Сразу же выполним правильные изменения имён новых компонентов:

Компонент	Новое значение Name	Text
TabItem1	tiFull	Полностью
TabItem2	tiEveryOtherLine	Через строчку
TMemo на TabItem2	meEveryOtherLine	
TabItem3	tiFirsWords	Первые слова
TMemo на TabItem3	meFirstWords	

Всё готово к реализации алгоритма. Как обычно, ищем нужно событие, т.к. любые действия в Delphi определяются срабатыванием какого-либо события. Оно может быть системным, оно может быть вследствие работы таймера, оно может инициироваться пользователем, как в нашем случае.

Реализуем часть алгоритма, связанную с маскированием всех слов в каждой строке, кроме первого, на событие смены страничек. Выделим форму `tcMain` в дереве компонентов панели `Structure`, затем перейдём в `Object Inspector` на закладку `Events` и дважды кликнем в строке `OnChange`. Данное событие случается каждый раз, когда пользователь переключает странички ярлычками в низу. Перед тем, как ввести код процедуры отклика на событие, опишем работу алгоритма.

Чтобы реализовать маскирование всех слов кроме первого слов в каждой строке, нужно выполнить следующие действия:

- Последовательно перебрать исходные строки из `meFull`;
- Для каждой исходной строки последовательно перебрать символы;
- В процессе перебора каждый символ сравнивается с пробелом;
- Как только встретился первый пробел, считаем, что первое слово пройдено;
- Начиная со следующего символа после пробела начинаем заменять любой символ на маскирующий знак «х»; пробелы не маскируем, чтобы сохранить слова;
- Измененную строку добавляем в `meFirstWords`.

Поясняющая схема работы алгоритма показана на рис. 7.3, а теперь мы готовы на событие `OnChange` написать следующий код:

```

procedure TfmMain.tcMainChange(Sender: TObject);
var
  Str: string; // текущая строка
  i: integer; // переменная для перебора по строкам
  j: integer; // переменная для перебора по символам строки
  Str
  GotFirst: boolean; // флаг нахождения первого слова
begin
  // если перешли на страничку tiFirstWords—"Первое слово"
  if tcMain.ActiveTab = tiFirstWords then
    begin
      // очистка содержимого Метод для «первых слов»
      meFirstWords.Lines.Clear;
      // перебор строк в «полном» Метод по i
      for i:= 0 to meFull.Lines.Count-1 do
        begin
          // берется текущая строка
          Str:= meFull.Lines[i];
          // «сбрасываем» флаг/признак прохождения первого слова
          GotFirst:= false;
          // перебор символов в текущей строке
          for j:= Low(Str) to High(Str) do
            begin
              // если первое слово уже найдено и символ не пробел
              if GotFirst and (Str[j] <> ' ') then
                // заменить текущий символ «крестиком»
                Str[j]:= 'x';
                // если первое слово ещё не нашли, а текущий символ—пробел
                if not GotFirst and (Str[j] = ' ') then
                  // значит первое слово нашли, флаг ставим в «истина»
                  GotFirst:= true;
            end;
          // добавление измененной строки в Метод для «первых слов»
          meFirstWords.Lines.Add(Str);
        end;
      end;
    end;

```

Разберем введенный код. В разделе переменных объявлены четыре, где Str строкового типа будет выполнять роль текущей строки. Переменные i и j будут использоваться для организации двух циклов или переборов, первый — по строкам meFull, второй — по символам в строке Str. Переменная GotFirst является «флагом». Обычно «флагом» называют переменную логического типа, которая изначально хранит значение «ложь»/«false», а с наступлением некоторого события или выполнения

Переходим к телу процедуры. Событие `OnChange` срабатывает для всего компонента `tcMain`, а рассмотренный алгоритм замены должен выполняться только при переходе на страничку «Первые слова», поэтому в начале стоит соответствующая проверка. После этого очищаем содержимое `meFirstWords` методом `Clear`, т.к. мы не знаем, что там осталось от предыдущего раза. Далее начинается первый цикл, в ходе выполнения которого мы копируем текущую по индексу `i` строку в переменную `Str`. Не выходя из первого цикла с перебором строк, попадаем во второй цикл для перебора символов в строке `Str`, и получается цикл-в-цикле. Второй цикл называется вложенным.



Во вложенном цикле перебираются символы в строке. Как только путём сравнения найден первый пробел, поднимается «флажок», а все дальнейшие символы маскируются. Так происходит до конца строки. Как только мы перебрали все символы в строке, то мы выходим из вложенного цикла. Переделанная строка добавляется в `meFirstWords`. Затем мы попадаем в начало внешнего цикла, берём следующую строку из `meFull`, заходим во вложенный цикл и процесс повторяется. Как только мы перебрали все строки, процедура завершается.

7.2. Рассуждения о поиске наилучшего варианта

Если вы хотите поскорее закончить приложение, то данный раздел можно временно пропустить. Не забудьте вернуться к нему, если у вас появится желание переосмыслить сделанное с точки зрения качества структуры кода потребления вычислительных ресурсов. Конечно, такие размышления не для новичков. Но, закончив данный проект, вы уже перестанете быть новичком, поэтому будет полезно проработать данный раздел.

Вдумчивый читатель может задать вопрос, а нет ли смысла один раз один раз преобразовать исходный текст стихотворения, чтобы не делать это каждый раз при переходе на страничку «Первые слова»? Действительно, событие `OnChange` срабатывает каждый раз при смене странички. Гораздо лучше выполнять алгоритм маскирования при изменении содержимого `meFull`, что фиксируется событием `OnChange` для данного компонента. Тогда такое маскирование будет происходить всего один раз для одного стихотворения.

Это замечание настолько серьезное, что мы остановимся на нём подробнее. Сейчас разговор идёт не о корректности работы алгоритма, в чем можно убедиться, протестировав приложение, а о поиске наилучшего варианта с точки зрения рационального расходования вычислительных ресурсов. В такой постановке задачи принятие решения зависит от множества факторов. Например, более правильный вариант в данных условиях может оказаться не таким уж привлекательным в других. Представьте, что

у нас будет несколько разных измененных текстов одного и того же стихотворения. Один — маскирование всех слов, кроме первого. Второй — маскирование целых строк, но через одну. Может быть третий и четвёртый алгоритм. Тогда наш вариант будет работать только при переключении на конкретную страничку. А алгоритм на событие изменения `meFull` — всегда для всех страничек, хотя вряд ли они понадобятся пользователю одновременно.

Возможно, наилучшим вариантом будет такой, когда при помощи системы флагов мы запускаем алгоритм изменения исходного текста: а) только один раз; б) только для конкретной странички. Подобный вариант кажется практически идеальным. Но мы добавляем минимум ещё одну переменную и усложняем код. Готовы ли мы пойти на это? А что мы сэкономим? Если приложение работает достаточно быстро, то пользователь и не почувствует ускорение работы. Вся наша работа будет направлена только на достижение некоего «мифического» идеального варианта, а нужно ли это?

Смысл изложенных выше рассуждений сводится к одному. Всегда тщательно продумывайте целесообразность поиска наилучшего решения. Идеал достичь сложно, потому что улучшая одно, вы ухудшаете другое. В конечном итоге можно потратить дни, месяцы или даже годы, делая код идеально красивым и максимально правильным. Но вы потеряете драгоценное время, которое можно было потратить на реализацию дополнительных интересных возможностей вашей программе. Вас обойдут конкуренты, приложение перестанет отвечать требованиям пользователей, закончится финансирование — и это только основные проблемы. Конечно, такой подход не означает, что не нужно думать над улучшениями. Просто всегда соизмеряйте затраченные ресурсы и ожидаемый результат. Поэтому следующим шагом после тестирования первого алгоритма будет реализация нового.

7.3. Добавление нового алгоритма

Новым запланированным алгоритмом у нас будет замена нечётных строк на аналогичные по длине и числу слов, но со всеми маскированными символами. Последовательность действий будет такая:

- Последовательно перебрать исходные строки из `meFull`;
- Если индекс строки нечётный, то перебрать символы и заменить на маскирующий знак «х»;
- Измененную строку добавить в `meEveryOtherLines`.

Как видно, данный алгоритм гораздо проще предыдущего, т.к. нам не нужно выделять отдельные слова. Иллюстрация алгоритма представлена на рис. 7.4. Единственно, к чему нужно отнестись с повышенным вниманием, это — индексация строк. В компоненте `ТМемо` строки содержатся в свойстве `Lines`. Доступ к конкретной строке осуществляется через индекс. Например, `Lines [ 3 ]` позволяет обратиться к четвертой строке. Индексация начинается с 0, поэтому индекс 3 соответствует номеру на единицу больше, т.е. 4. Поэтому, когда мы говорим о нечётной строке по номеру, например, третьей, то индекс у неё будет как раз чётный, т.е. 2. Чтобы не запутаться при кодировании алгоритма или его обсуждения с коллегами старайтесь не употреблять «восьмая строка», «третья строка» и т.д. Гораздо лучше говорить «строка с индексом семь» или «строка с индексом два». Конечно, ограничивать себя нужно лишь на этапе, пока вы не почувствуете лёгкость в придумывании и реализации алгоритмов обработки массивов строк.

При реализации алгоритма следует еще раз внимательно проконтролировать часть кода, где представлены вложенные циклы. Добавлять строчку в `meEveryOtherLines` нужно каждый проход, но изменять только через раз. При вводе кода легко допустить ошибку, и это касается не конкретно нашего случая, а вложенных циклов вообще. У нас же результирующий текст может показаться весь маскированный, как это было с автором при отладке


```

procedure TfmMain.tcMainChange(Sender: TObject);
var
  // код опущен для экономии места
begin
  // если перешли на страничку tiFirstWords—«Первое слово»
  if tcMain.ActiveTab = tiFirstWords then
    begin
      // код пропущен
    end;
  // если перешли на страничку tiFirstWords—«Через строчку»
  if tcMain.ActiveTab = tiEveryOtherLine then
    begin
      // очистка содержимого Метод для «через строчку»
      meEveryOtherLine.Lines.Clear;
      // перебор строк в «полном» Мето по i
      for i:= 0 to meFull.Lines.Count-1 do
        begin
          // берется текущая строка
          Str:= meFull.Lines[i];
          // если индекс строки нечётный
          if (i mod 2) = 1 then
            begin
              // перебор символов в текущей строке
              for j:= Low(Str) to High(Str) do
                begin
                  // если текущий символ не пробел
                  if Str[j] <> ' ' then
                    // заменить текущий символ «крестиком»
                    Str[j]:= 'x';
                end;
            end;
          // добавление строки в Мето для «через строчку»
          meEveryOtherLine.Lines.Add(Str);
        end;
      end;
    end;
  end;

```

Введённый код не требует долгих разъяснений. Основные его фрагменты очень похожи на структуру, рассмотренную для предыдущего алгоритма с маскированием всех слов кроме первого. Здесь принципиально новая часть посвящена проверки индекса строки на нечётность:

```

if (i mod 2) = 1 then

```


В приведённом выше фрагменте используется оператор вычисления остатка от деления нацело **mod**. Если число чётное, то оно делится на 2 без остатка, тогда оператор вычисления остатка целочисленного деления будет давать 0. В противном случае, когда число нечётное, результатом будет 1. Именно такая проверка и используется для выявления нужной строки по её индексу.

При тестировании можно выявить следующий факт, проиллюстрированный на рис. 7.5. Первая строка первого четверостишья показана полностью. Но во втором четверостишье первая строка изменена, а вторая — нет. Это происходит потому, что при переборе строк в Мето учитывается строка, разделяющая четверостишья. Такая строка — пустая, однако она тоже считается строкой. Поэтому первая строка второго четверостишья будет иметь индекс 5.

Если вам захочется сделать так, чтобы каждое четверостишье начиналось с исходной строки, а маскировались только вторая и четвёртая строка, то наш алгоритм придётся доработать. Принцип может быть достаточно прост, практически такой же, как мы определяем слова в первом алгоритме. Если пробел в текущем символе есть признак того, что слово завершилось, то и пустая строка есть признак завершения четверостишья. Не рекомендуется считать строки, так как достаточно большое число стихов состоят из пятистиший или трёхстиший. Поэтому нужно будет сравнивать текущую строку `Str` с пустой строкой `''`. Две одинарные кавычки означают «пустую строку», т.е. строку, не содержащую символов.

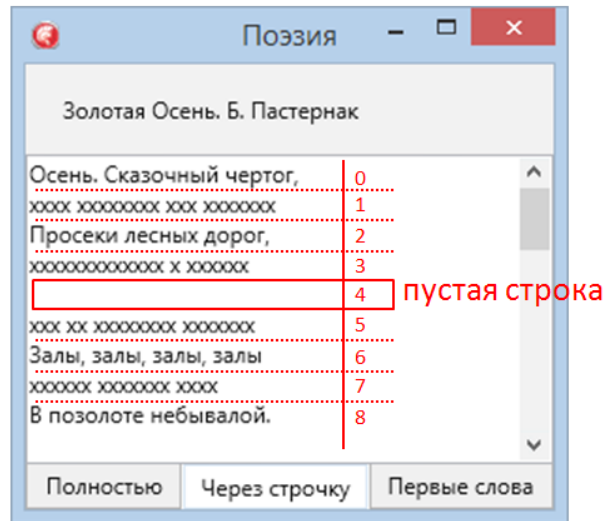


Рис. 7.5. Маскирование через строчку

7.4. Развитие приложения

Основной целью нашего учебного проекта было научиться работать с массивами строк на примере полезного мобильного приложения. Но если вы относитесь к этому чуть серьезней, то развить проект до уровня хорошей исследовательской работы. Полученных вами навыков вполне достаточно для реализации следующих идей:

- Комбинировать первый и второй алгоритмы, показывая нечётные строки полностью, а в чётных лишь первые слова;
- Случайным образом скрывать слова во всём стихотворении;
- Сделать процесс изучения контролируемым и ступенчатым с переходом на новый уровень: сначала пользователю даётся полный стих, затем скрывается каждое второе слово, затем каждое третье, потом — каждое четвёртое, но первые слова в строках всегда показаны целиком;
- Создать небольшой список предлогов и союзов; это может быть объект класса TStringList, который мы уже рассмотрели, а может быть и просто компонент TMemo со свойством Visible установленным

в false; в таком компоненте-невидимке можно хранить словарь (хотя профессиональные программисты могут и пожурить вас за нецелое использование визуального компонента); скрываются все слова, кроме предлогов и союзов;

- В каждой строчке показывать только первое и последнее слово;
- Добавить кнопку «быстро показать», при нажатии на которую стих показывается без маскирования, но на 1–2 секунды;
- Маскировать не все символы в слове, а оставлять первую и последнюю буквы;
- Сохранять стих в файле, что мы также умеем делать; лучше для этого воспользоваться следующим методом: у вас всегда есть файл `роем.txt`, где хранятся строки в виде:

“А.С. Пушкин” “Поэт” `роем1.txt`

“Б. Пастернак” “Золотая осень” `роем2.txt`

“С. Есенин” “Белая берёза” `роем3.txt`

при загрузке приложения на первой страничке возникает список `TListBox` стихов по названию и автору, когда выбран конкретный стих из файла `роем.txt` вычитываются строки и определяется, в каком файле хранится сам стих; он и загружается;

- Разрешить пользователю вставлять произвольный стих на первую страничку для заучивания, для этого по кнопке нужно установить свойство `ReadOnly` компонента `meFull` в значение false; введенный пользователем стих, например, за счёт копирования-вставки можно сохранить в файле, записав в `роем.txt` его название и автора.

И это — только небольшое количество возможных идей! Мы уверены, что вы способны придумать гораздо больше интересных алгоритмов и методов запоминания. Большое количество алгоритмов даст вам основу провести исследование. Если у вас есть друзья, готовые поучаствовать, то будет просто великолепно. Установите на их смартфоны приложение, раздайте один и то же стих, но пусть каждый использует свой алгоритм. Можно дать разные стихи, но один алгоритм. Варьировать стихами и алгоритмами

нужно обязательно, чтобы оценка была объективной. В результате исследования вы сможете определить самый лучший алгоритм для запоминания стихов. Приятным дополнительным эффектом будет масса выученных стихов вами и вашими друзьями. Полученным прогрессом можно делиться через социальные сети при помощи Share Sheets.

После заучивания стихов можно перейти к прозе. Есть такие фрагменты, которые по красоте не уступают рифмованным стихам. Это уже поле ваших самостоятельных исследований. Как разбить текст на фрагменты, как выполнять сокрытие. Такое приложение будет весьма полезно для заучивания текстов на иностранном языке. В любом случае, ваш труд не пропадёт даром, а помимо развития вашего уровня профессионализма в программировании будет расти и ваши литературные и даже артистические способности.

7.5. Запись голоса

Проект выполнен, приложение отлажено, первое длинное красивое стихотворение выучено благодаря нашей небольшой, но полезной программе. Но простое механическое заучивание и безэмоциональное прочтение не только не порадует слушателей, но даже может испортить хороший стих. Блёклый голос, невыразительная интонация, сбивчивый ритм — вот они, главные враги хорошей поэзии. Выучить стихотворение — еще полдела, вторую часть усилий нужно потратить, чтобы научиться его правильно «сыграть». Конечно, можно тренироваться на друзьях и родственниках, если у них хватит терпения. Гораздо лучше «послушать себя со стороны». Для этого мы добавим к нашей программе кнопки «запись», «стоп» и «воспроизвести». Микрофоны современных смартфонов достаточно хороши, чтобы получить вполне качественную аудио запись.

Как обычно, начнём с интерфейса. При разделении кода на интерфейсную и функциональную части они обе могут разрабатываться независимо, т.е. в любой последовательности. Но для начинающих разработчиков проще начинать с элементов управления. Как только мы разместим на интер-

фейсе нужные кнопки, функциональность станет для нас более отчётливой. Создадим клон проекта в папке «Project 7.2» и откроем проект из неё. Выделим tcMain и сменим Align с Client на None. Добавим TToolBar, свойство Align установим в Top, а Name в tbRecord. После этого вернём Align в значение Client для компонента tcMain.

Разместим на второй сверху tbRecord четыре быстрые кнопки TSpeedButton с Align = Right. Но не торопитесь создавать для них процедуры отклика. Лучше весь функционал создавать централизованно и максимально изолировано от интерфейса. Кнопки могут поменяться, а функционал останется. Для этого есть специальный компонент TActionList, который мы и разместим на форме. Дважды щелкнем на него и добавим четыре новых действия (рис. 7.6).

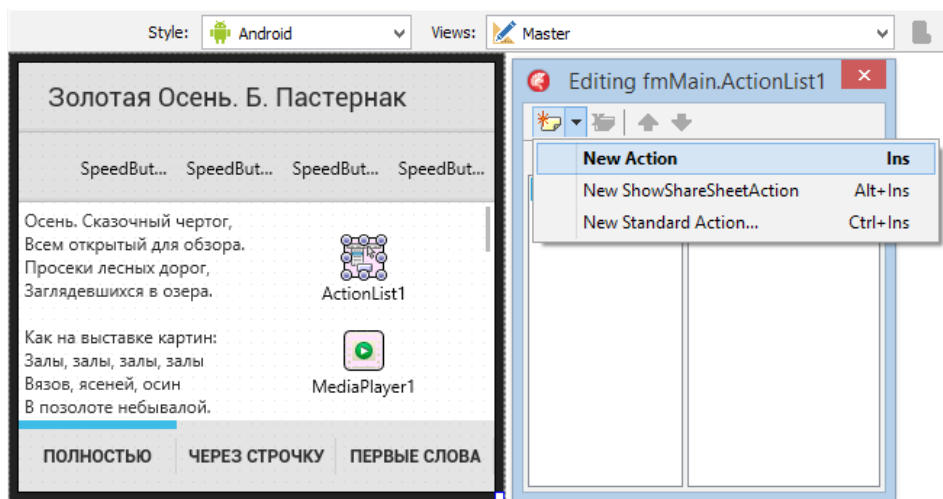


Рис. 7.6. Макетирование обновлённого интерфейса

Как только мы добавили новые компоненты, нужно сразу изменить их имена со стандартных по умолчанию на новые, наполненные смыслом:

Свойство Name по-умолчанию	Корректное имя
Action1	aStartRecord
Action2	aStopRecord
Action3	aStartPlayBack
Action4	aStopPlayBack
ToolButton1	tbStartRecord
ToolButton2	tbStopRecord
ToolButton3	tbStartPlayBack
ToolButotn4	tbStopPlayBack

Очень также полезно сразу выделить с кнопкой Shift все четыре действия и вручную ввести им свойство Category. Если мы выделяем группу компонентов и изменяем их свойство, то оно становится одинаковым для всех выделенных элементов. Данная техника весьма эффективна, когда количество «действий» становится больше десяти, что весьма характерно даже для мобильных приложений. Когда мы назначим «действиям» свою категорию, то их проще будет в этой категории найти в компоненте ActionList.

Поочередно выберем «быстрые» кнопки tbStartRecord, tbStopRecord, tbStartPlayBack и tbStopPlayBack и свяжем их с соответствующими действиями в Object Inspector через свойство Action для каждой из них. Не стоит изменять надпись на кнопки через их свойства. Попробуйте задать свойство Text у компонентов Action. Вы увидите, что автоматически изменяются и текст на кнопках. Для действий aStartRecord и aStopRecord задайте свойство Text как «Старт» и «Стоп», соответственно. Для действий и кнопок, которые начинают и останавливают воспроизведение, не будем вводить текст. Для tbStartPlayBack и свойства StyleLookup выберите значение из списка playtoolbutton, а для tbStopPlayBack и свойства StyleLookup значение stoptoolbutton. Переключите стиль в дизайнера на Android и сравните результат с тем, что показано на рис. 7.7.

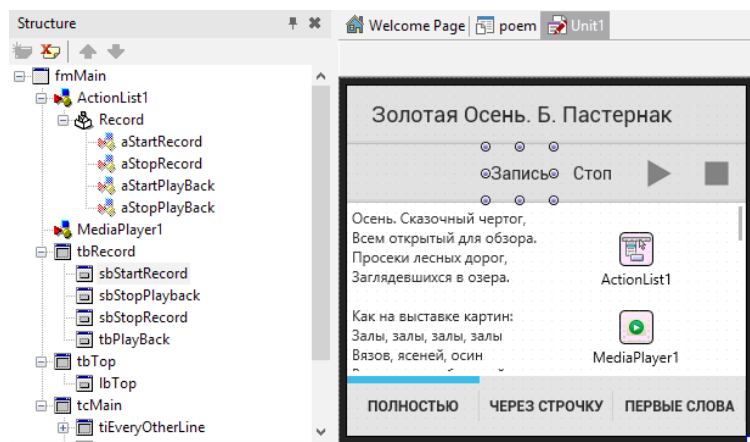


Рис. 7.7. Дизайн интерфейса для Android

Подготовимся к кодированию. Разместим на форме компонент TMediaPlayer, затем переключимся в редактор кода и в разделе `uses` добавим `System.IOUtils`. В разделе `private` класса формы добавим переменную `FMicrophone: TAudioCaptureDevice`. Опять вернёмся к `ActionList1`. Далее для каждого действия последовательно нажмём на `OnExecute`. Затем разом для всех процедур отклика введём следующий код для автоматически созданных процедур:

```
procedure TfmMain.aStartPlayBackExecute(Sender: TObject);
begin
    // задать имя файла для воспроизведения
    MediaPlayer1.FileName:= TPath.GetHomePath + '\поем.3GP';
    // запустить проигрыватель
    MediaPlayer1.Play;
end;
procedure TfmMain.aStartRecordExecute(Sender: TObject);
begin
    // инициализация микрофона
    FMicrophone:= TCaptureDeviceManager.Current.DefaultAudioCaptureDevice;
    // задание файла для записи голоса
    FMicrophone.FileName:= TPath.GetHomePath + '\поем.3GP';
    // начать запись голоса
    FMicrophone.StartCapture;
end;
```

```

procedure TfmMain.aStopPlayBackExecute(Sender: TObject);
begin
  // ОСТАНОВИТЬ ВОСПРОИЗВЕДЕНИЕ
  MediaPlayer1.Stop;
end;
procedure TfmMain.aStopRecordExecute(Sender: TObject);
begin
  // ОСТАНОВИТЬ ЗАПИСЬ ГОЛОСА
  FMicrophone.StopCapture;
end;

```

Перед тем, как запустить приложение, выберем платформу Android. Данный код будет работать только на ней. Приложения на данной платформе требуют разрешений для выполнения очень многих действий, включая доступ и использование микрофона. По умолчанию проекты мобильных приложений для Android в Delphi/RAD Studio/C++Builder не имеют данных разрешений. Поэтому перейдём в настройки проекта в главном меню Project->Options. В дереве настроек выберем узел User Permissions и разрешим приложение записывать голос, как показано на рис. 7.8 в левой части:

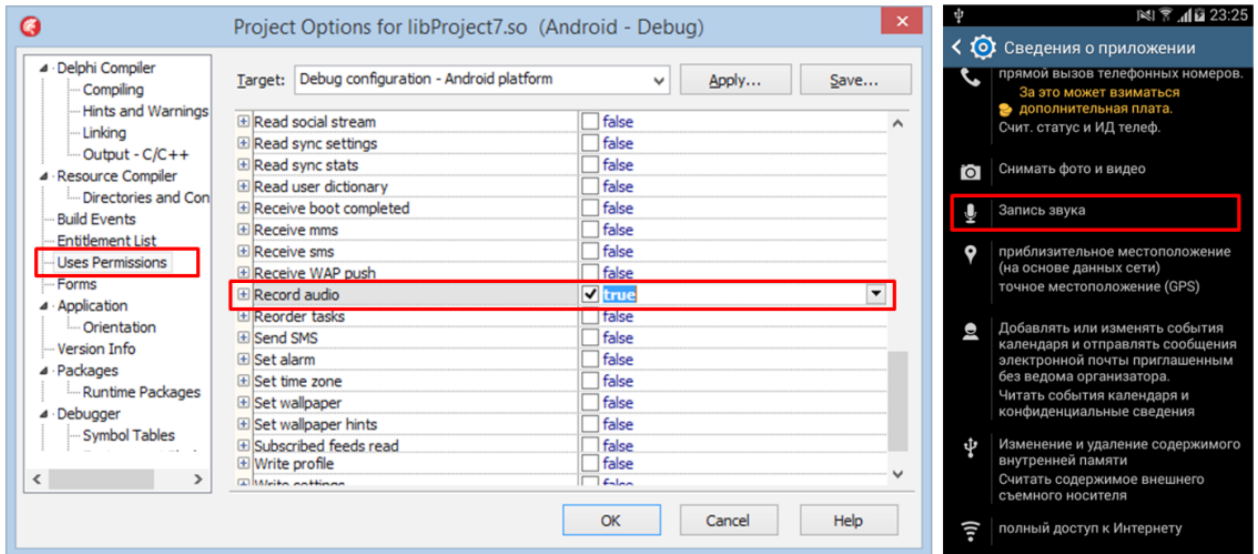


Рис. 7.8. Настройки разрешений для мобильного приложения

Когда приложение уже будет размещено на мобильном устройстве, можно зайти в настройки и также посмотреть разрешения (рис. 7.8, справа).

Соберём и запустим приложение на мобильном устройстве. Если вы внимательно тестировали приложение, то заметили некоторую особенность по сравнению с профессиональными приложениями. Изначально доступными могут быть только кнопки для начала записи и воспроизведения. Кнопки остановки записи и воспроизведения становятся активными только тогда, когда эти процессы уже запущены. Также, например, когда мы запустили запись, то данная кнопка блокируется и второй раз нажать её нельзя. Можем дать такой совет — используйте свойство `Enabled` у «действий». Например, `Action1.Enabled:= false;` заблокирует не только «действие», но и связанную с ним кнопку, а `Action1.Enabled:= true;` — наоборот. Очень рекомендуется изучить материалы по событию `OnUpdate` для «действий». Это полезно, когда можно проанализировать состояние готовности компонентов, например, код: **if** `Assigned(FMediaPlayer.Media)` **then** ... позволяет понять, выбран ли правильный файл для воспроизведения.

Описанные выше советы, даже если ими воспользоваться аккуратно, значительно усложнят код. А наша задача на начальном этапе сделать так, чтобы кода не было слишком много. Именно поэтому мы и используем Delphi/RAD Studio/C++Builder для создания мобильных приложений.

Две инструментальные панели вверху интерфейса — не самое красивое решение. Тем более, что запись нам потребуется только в самый последний момент, когда стих уже выучен. Предпримем ещё одно усилие для улучшения интерфейса. Сделаем так, чтобы инструментальная панель для записи была скрыта основное время, а открывалась только по нажатию на специальную кнопку. Начнём в этот раз с проектирования функциональности, т.е. с «действий».

Дважды щёлкнем на `ActionList1`, выберем категорию `Record` и добавим два новых действия. Одно назовём `aShowRecordToolBar`, другое `aHideRe-`

cordToolBar. Не надо избегать длинных имён. Наоборот, чем длиннее имя, тем больше оно говорит о своём компоненте. Добавим на верхнюю инструментальную панель TSpeedButton и назовём, как несложно догадаться, sbShowRecordToolBar. Свойству Align выберем значение Right. На вторую сверху инструментальную панель тоже добавим TSpeedButton, назовём её sbHideRecordToolBar, а свойство Align установим тоже в значение Right. Чтобы данная кнопка стала самой правой, визуальнo мышкой потащим её в самый правый край так, чтобы уже существующие кнопки подвинулись влево.

Как следует из названия, первая кнопка будет показывать инструментальную панель записи, а вторая — скрывать её. Выберем кнопку tbShowRecordToolBar и свяжем в Object Inspector её с «действием» aShowRecordToolBar, а кнопку tbHideRecordToolBar — с действием aHideRecordToolBar. Выделим кнопку tbShowRecordToolBar и установим ей свойство StyleLookUp в значение arrowdowntoolbutton, а для кнопки tbHideRecordToolBar — arrowuptoolbutton. Проверьте результат по рис. 7.9.

Изначально панель должна быть скрыта. Выберем панель tbRecord и установим свойство Visible в false. Осталось совсем немного — сделать так, чтобы при нажатии на кнопки панель появлялась и исчезала. Поскольку кнопки у нас связаны с действиями, tbShowRecordToolBar с действием aShowRecordToolBar, а tbHideRecordToolBar с действием aHideRecordToolBar, то реализовывать мы будем событие OnExecute для наших действий. А сделаем мы это так:

```
procedure TfmMain.aHideRecordToolBarExecute(Sender:
TObject);
begin
  tbRecord.Visible:= false;
end;
procedure TfmMain.aShowRecordToolBarExecute(Sender:
TObject);
begin
  tbRecord.Visible:= true;
end;
```

Смысл введённого кода ясен и без проверки. Однако мы сохраним и запустим проект на мобильном устройстве при выбранной платформе Android в Project Manager.

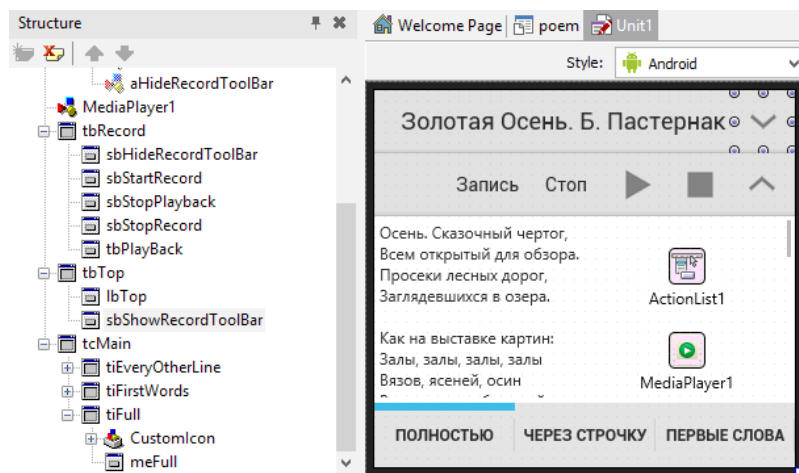


Рис. 7.9. Окончательный вариант интерфейса

7.6. Уведомления

В запоминании стихов очень важно делать это периодически, а не за один раз. Такой подход даёт более надежные результаты. Выучили стих, а потом с промежутком в один час повторяем. Для реализации этой функции на мобильных платформах существует такое понятие как «уведомление». Если вы используете смартфон в повседневной жизни, то уведомления вам хорошо знакомы. Наше приложение также может создавать и отправлять уведомления в специальный сервис, который и будет посылать их нам. Использовать мы их будем так: создаём и направляем сами себе уведомление о необходимости повторить стихотворение.

Мы уже исчерпали количество страничек с закладками в том смысле, что добавление новой сделает интерфейс более запутанной. Поэтому всё, что связано с уведомлениями будет размещено на отдельной форме. Для

Windows-приложений вторая (третья, четвёртая и т.д.) форма будут отображаться как отдельное окно. Такое окно может быть перемещено произвольным образом относительно главного. Для мобильных приложений второе окно появляться «поверх» главного, полностью закрывая его. В нашем приложении данный подход является вполне допустимым.

Добавим в наш проект новую форму. Сделаем это через главное меню File->New->Multi-Device Form — Delphi. На экране появится пустая форма с названием Form2. Сразу переименуем её в fmNotification. Свойства Height и Width установим такими же, как и для главной формы fmMain. Если нам нужно будет переключиться между формами, то сделать это просто. В главном меню выберем View->Forms или нажмём комбинация клавиш Shift+F12. Появится окно (рис. 7.10), где можно выбрать нужную форму по названию. Именно поэтому и нужно назвать формы по-разному, чтобы не приходилось терзать себя сомнениями: «нам нужна Form3 или Form33»?

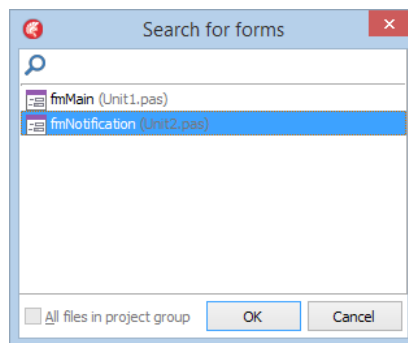


Рис. 7.10. Выбор форм в составе проекта

Теперь, когда мы можем свободно перемещаться между формами проекта, опять выберем форму fmNotification и займёмся её дизайном. Добавим на fmNotification компонент TToolBar и переименуем его в tbTop, а затем TListBox, который переименуем в lbNotification. Свойству Align компоненту lbNotification зададим значение Client. В список по правой кнопке добавим два элемента TListBoxItem. Сделаем их дизайн максимально простым: выбо-

рем свойство `StyleLookup` и установим значение `listboxitemnodetail`. На первый элемент списка разместим компонент `TSpinBox`, на второй — `TButton`. Свойство `Text` для первого зададим как «Через, мин», а второму «Повторить стих». Кнопке зададим `Text` как «Напомнить». На `Toolbar` разместим `TSpeedButton`, свойство `Align` зададим как `Left`, а `StyleLookup` как `backtoolbutton`. Проверьте дизайн второй формы по рис. 7.11. Заранее добавим туда `TActionList` и `TNotificationCenter`.

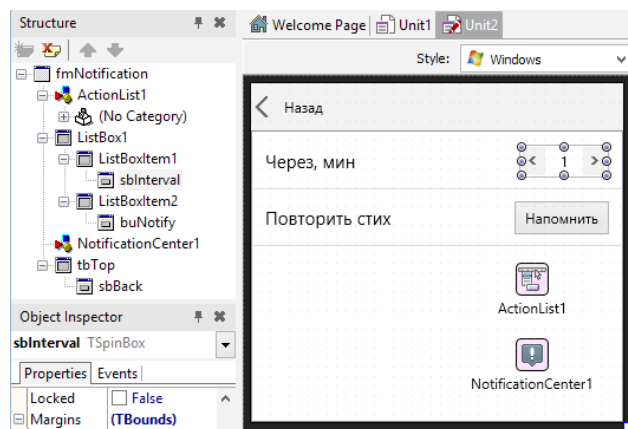


Рис. 7.11. Дизайн второй формы

Обратите внимание на компоненты `TSpinBox` и `TButton`. Они аккуратно выровнены по правой стороне с равномерными отступами сверху и снизу. Если мы будем изменять размеры формы, то их расположение останется таким же аккуратным. Свойство `Align` этих компонентов установлено как `Right`, поэтому они «прилипают» к правой стороне. Но это ещё не всё. Аккуратные отступы в 10 пикселей сверху, снизу и справа достигаются заданием соответствующих величин свойства `Margin` (поле). Выделим поочередно эти компоненты, раскроем свойство `Margin` в `Object Inspector` и вложенным свойствам `Right`, `Top` и `Bottom` установим значение 10 (рис. 7.12). Как бы теперь не изменялись размеры формы при запуске на устройствах с различными экранами, данные компоненты будут размещены практически идеально.

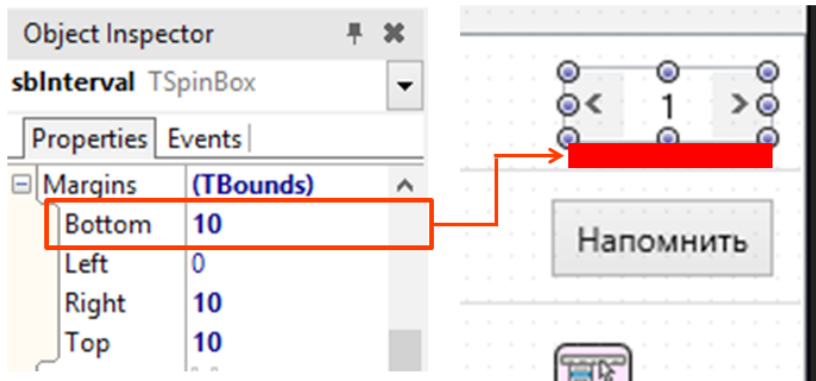


Рис. 7.12. Настройка свойства Margin

Дважды щёлкнем на `ActionList1` и добавим новое обычное «действие». Назовём его `aBackToMainForm`, по нему мы будем возвращаться обратно на главную форму. Событие `OnExecute` для данного действия закодируем как:

```
procedure TfmNotification.aBackExecute(Sender: TObject);  
begin  
  Close;  
end;
```

`Close` — это процедура. Она принадлежит классу формы, т.е. закрываться будет именно текущая вторая форма. Соответственно, мы автоматически возвратимся на главную. Свяжем теперь кнопку `sbBack` с действием `aBack`. После этого любое нажатие на данную кнопку будет вызывать запуск введённой процедуры. Процедура будет закрывать вторую форму, а мы — переходить на главную.

Гораздо более насыщенный код придётся ввести для реализации напоминания. Опять зайдём в `ActionList1`, добавим новое действие, которое переименуем в `aNotify`. Дважды щёлкнем на действии и введём в процедуру отклика следующий код:

```
procedure TfmNotification.aNotifyExecute(Sender: TObject);  
var  
    // переменная для ссылки на объект «уведомление»  
    myNotification: TNotification;  
begin  
    // создать «уведомление»  
    MyNotification:= NotificationCenter1.CreateNotification;  
    // текст уведомления  
    MyNotification.AlertBody:= 'повторить стих';  
    // установить время, через какое уведомление будет послано  
    MyNotification.FireDate:= Now + EncodeTime(0, Round(sbInter-  
        val.Value), 10, 0);  
    // поставить уведомление в расписание  
    NotificationCenter1.ScheduleNotification(MyNotification);  
    // удалить объект  
    MyNotification.Free;  
end;
```

Рассмотрим подробно, что происходит. Одним из главных компонентов является TNotificationCenter, который называется у нас NotificationCenter1 и располагается на форме. На форме он представлен в виде компонента, но при запуске он никак не отображается. Вся его мощь заключена в незримом функционале. Именно он отвечает за посылку уведомлений. Напоминание у нас будет приходить в виде уведомления — стандартного сервиса мобильных операционных систем. Чтобы задействовать данный компонент, надо сначала создать «объект» уведомления. Представьте себе, что центр уведомлений — это почтовая служба. Чтобы она доставила письмо адресату, его нужно создать как «объект». Создавать объекты мы уже умеем, для чего нужна переменная типа ссылка, чем и является myNotification: TNotification.

В самом начале процедуры первым же действием мы при помощи центра уведомления создадим новое уведомление и сохраним ссылку на него в переменной myNotification. Теперь через данную ссылку можно модифицировать созданный объект. Мы изменяем свойства нового объекта, задавая текст уведомления через свойство AlertBody и время срабатывания через FireDate. Здесь мы попадаем в область работы с датами и временем. Функция Now возвращает текущую дату, а функция EncodeTime — закодированное время. Зачем время нужно кодировать? Время из привычных

четырёх значений — часы, минуты, секунды, миллисекунды — всегда кодируется в одно число. В машинном представлении время — это всего лишь одно значение. Поэтому мы и кодируем его из четырёх значений. Вместо значений минут мы видим конструкцию `Round(SpinBox1.Value)`. `SpinBox1.Value` — это введенное пользователем значение интервала посылки уведомления. Грубо говоря, через сколько придёт напоминание. Мы это значение посылаем в функцию `Round(...)`, которая получает вещественное число, а возвращает целое. Целое значение нам нужно, т.к. количество минут — это целочисленная величина.

Остаются два действия. Первое — мы отправляем созданный объект «уведомление» в центр уведомлений. Второе и последнее — уничтожаем созданный объект, т.к. он своё дело уже сделал. Можно создавать уведомления, которые буду приходить периодически, тогда уведомления о необходимости повторять стихотворение будут приходить пользователю каждый час. Можно поступить по-иному в зависимости от плана повторения стиха: сразу создать несколько напоминаний. Одно будет напоминать перечитать стих через час, другое — через два, потом на утро следующего дня. Можно очень точно задать дату и время при помощи функции `EncodeDateTime`, куда в качестве параметров передаётся год, месяц, день, час, минута, секунда и миллисекунда. Но, конечно, более удобно использовать `Now` как текущую дату и уже дальше добавлять время, используя `EncodeTime`.

Каждой форме в проекте соответствует отдельный модуль или «Unit». Каждая форма, как мы знаем, это объект класса. Один модуль, один класс, одна форма. Всё достаточно просто. Но что делать, когда нужно из одной формы обратиться к другой? При запуске появляется главная форма, из которой нужно вызывать другую. Давайте изучим рис. 7.13, предварительно выбрав `Unit1.pas` посредством главного меню `View->Units`. Потом в главном же меню выберем `File->Save As`, дальше введём `MainUnit`. То же самое проделаем и с `Unit2`, сохранив его под именем `NotificationUnit`. Теперь у нас полная гармония в именах.

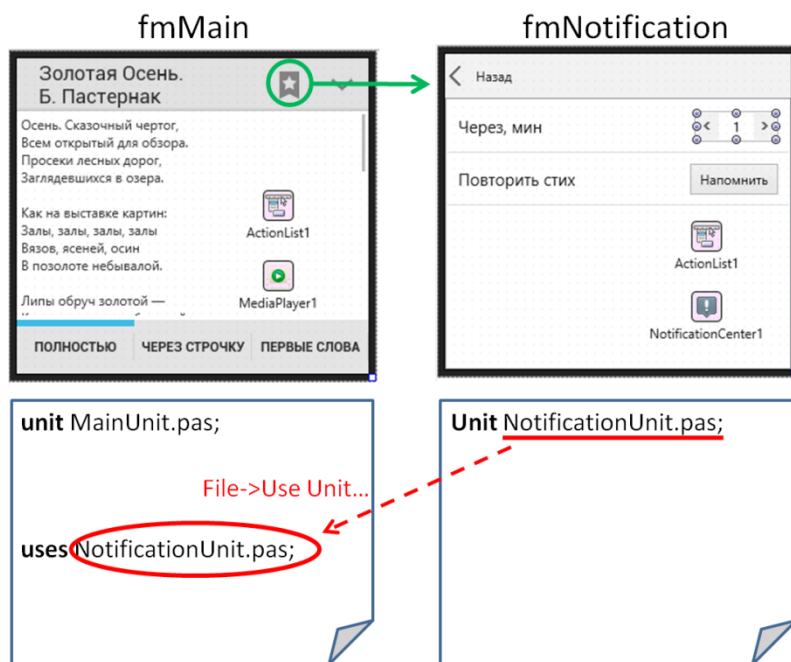


Рис. 7.13. Взаимодействие форм в проекте

Форма `fmMain` представлена своим классом в модуле `MainUnit`. Форма `fmNotification` — в своём модуле `NotificationUnit`. Каждая форма находится в своём отдельном модуле и ничего не знает о существовании другой. Для нас это не подходит, т.к. мы собираемся из главной формы вызвать вторую. Для этого нужно, чтобы главная форма «знала» о существовании второй. Сделать это просто. Выберем `MainUnit` через главное меню `View->Units...`. Потом опять в главном меню `File->Use Unit...`, а в появившемся окне (рис. 7.14) выделим единственный доступный элемент в списке и нажмём кнопку «ОК». В результате этого в главном модуле `MainUnit` появится строчка, показанные в левой части рис. 7.14.

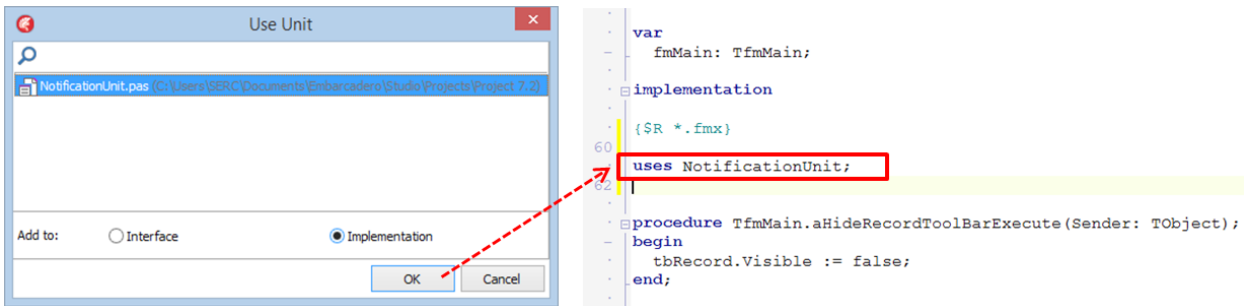


Рис. 7.14. Выбор второго модуля

Теперь гарантированно из главной формы можно вызывать вторую форму. Прежде чем сделать это и завершить проект, сделаем несколько странное упражнение. Его смысл будет в закреплении темы. Перейдём в главное меню в настройки проекта Project->Options, где затем в левой части в дереве опций найдём узел Forms (рис. 7.15). Внимательно изучим, что можно изменить. В левом списке мы видим список всех форм, доступных в приложении. Трогать его не будем. Зато посмотрим на верхний выпадающий список, озаглавленный MainForm. В качестве главной формы там стоит fmMain. Но если мы поменяем в этом компоненте fmMain на fmNotification, затем нажмём «Ок», потом сохраним и запустим проект, то мы увидим... что главной формой стала вторая! Отсюда делаем важный вывод: главной формой становится та, которая первая по счёту в мастере форм настроек проекта. Вернёмся в Project->Options и восстановим главенство формы fmMain.

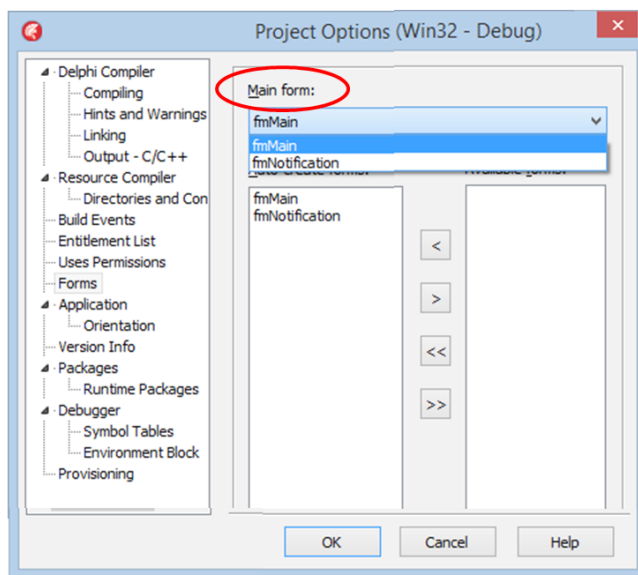


Рис. 7.15. Выбор главной формы в настройках проекта

Возвращаемся в главную форму `fmMain`, затем в дизайнере добавляем ещё одну «быструю» кнопку `TSpeedButton` с именем `sbShowNotificationForm` и свойством `LookupStyle = bookmarkstoolbutton`. В `ActionList1` добавим действие `aShowNotificationForm`, свяжем его с кнопкой `sbShowNotificationForm`. Затем «действию» дадим отклик:

```
procedure TfmMain.aShowNitificationFormExecute(Sender:
TObject);
begin
  fmNotification.Show;
end;
```

Код весьма прост. Мы просто показываем вторую формы. Дальше все действия будут происходить на ней. Пользователь создаст уведомление и нажмёт «Назад», закрыв тем самым вторую форму и вернувшись на главную. Не забудьте переключиться при сборке проекта на платформу Android, т.к. на платформе Windows мы сможем протестировать только общую часть интерфейса. Функционал, связанный с созданием уведомлений, работать не будет. Но для Android всё должно выглядеть так, как показано на рис. 7.16.

Вы можете запустить наше приложение, вызывать вторую форму, затем задать уведомление и потом даже выгрузить приложение. Про прошествии заданного интервала времени придёт уведомление, кликнув на котором мы запустим приложение с готовым для повторения стихом. Данное приложение обеспечивает вас прекрасным инструментом проведения действительно научных исследований. Попробуйте сначала на себе, тщательно фиксируя, какие стихотворения вы учили, сколько потратили времени, как повторяли, какие методы маскирования использовали и т.д. Мы очень надеемся, что кто-то из читателей, создав такое приложение и проведя с его помощью исследования, ощутит вдохновение. И это поможет выбрать свою будущую профессию, например, психолога или лингвиста, которая поможет затем реализовать себя в полной мере.

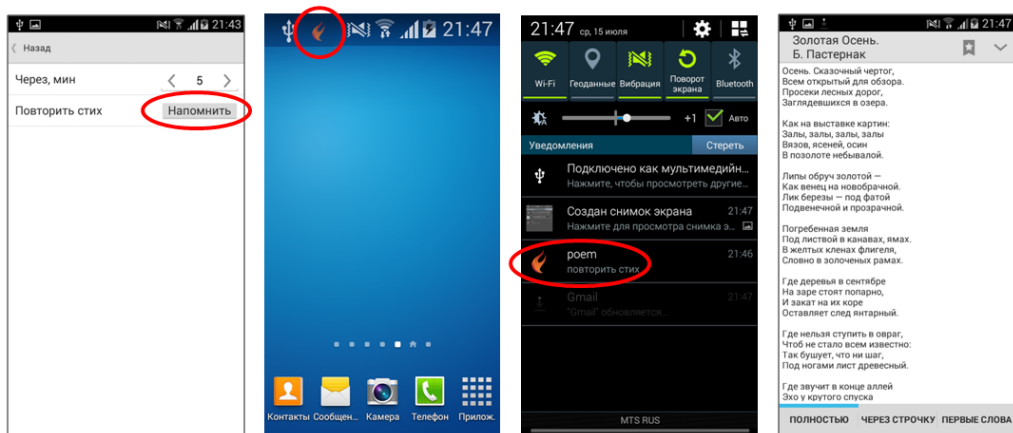


Рис. 7.16. Работа уведомлений в приложении

Система «суфлёр» на основе мобильного приложения

8.1. Публичные выступления: новости, драма, поэзия

Внимательное выполнение предыдущих проектов уже сделало из читателя достаточно опытного разработчика мобильных приложений. Техника визуального прототипирования позволяет быстро создавать программы для смартфонов и планшетов. Мы полностью сконцентрировались на отдельном приложении без связи с другими системами. Единственным исключением был проект «Дневник наблюдений», когда мы реализовали экспорт данных посредством Share Sheet. Такой подход очень прост, но позволял лишь отсылать данные при помощи сервиса электронной почты. Гораздо более мощным и универсальным способом является связывание приложений, когда из одного можно выполнять команды другого и наоборот. В данном разделе мы рассмотрим технологию «App Tethering» (связывание приложений).

«Связывание приложений» очень эффективный способ построения систем, когда в работе должно участвовать более одного устройства. Допустим, вам нужно разработать игру с участием двух игроков для мобильных устройств. Тогда придётся каким-то образом связать два устройства и передавать данные. На коммуникационном уровне есть как минимум два варианта: по сети WiFi и посредством Bluetooth. В первом случае нужно, чтобы устройства были подключены к общей «точке WiFi», что в наше время является делом обычным. Установление связи через Bluetooth также является простым и понятным методом связывания устройств для любого пользо-

вателя смартфонов или планшетов. Мы будем рассматривать связь по WiFi, но использование Bluetooth также просто и не потребует переделки кода.

Целью данной книги является показать, как с помощью Delphi/C++Builder/RAD Studio можно создавать мобильные приложения. Здесь не рассматриваются некоторые учебные примеры, лишённые возможности полезного применения, поэтому мы спланируем и реализуем систему на основе мобильных приложений, которую можно применять с успехом в жизни учебного заведения. Предлагаем следующую задачу: один пользователь с мобильном устройстве играет роль диктора или актёра, и ему нужно произнести достаточно длинный текст. Сейчас это вполне допустимо, когда, например, телеведущий рассказывает новости, держа планшет с текстовой подсказкой в руке. Конечно, именно чтение с планшета будет смотреться непрофессионально, но вполне можно иногда бросать взгляд на экран, и это выглядит достаточно красиво. Современные устройства улучшают имидж современного человека, поэтому диктор школьного телевидения может себе это позволить. Также планшеты могут помочь в прогоне школьных спектаклей, когда партии недостаточно твёрдо выучены. Вместо планшета в руке можно использовать компьютер с большим дисплеем. Наша среда разработки позволяет создавать приложения, которые могут быть собраны и для мобильных устройств, и для обычных компьютеров.

Итак, у нас будет одно мобильное устройство, которое будет дистанционно выполнять команды с компьютера или другого мобильного устройства (рис. 8.1). Возьмём в качестве исходной задачей прочтение выученного стихотворения. Если оно — сложное, то произнесение его на камеру или перед аудиторией без подсказок весьма рискованно. В предыдущем разделе мы разработали приложение для разучивания стихотворения. Но использовать его в качестве подсказки нельзя. Для поддержки выступающего не нужно видеть всё стихотворение, да оно будет показано слишком мелкими буквами. Идеальный вариант, когда мобильное приложение показывает текущую строку, а также следующую. Но это сделать автоматически? Автоматически никак, но если есть помощник, который видит весь текст и следит за выступающим. По мере произнесения слов помощник аккуратно выдаёт

подсказки. Он просто нажимает на строчку стиха или текста, которая тут же отправляется чере мобильное приложение на планшет выступающего.



Рис. 8.1. Работа системы подсказок выступающему

С точки зрения программирования сделать такие приложения несложно. Давайте перечислим функции приложения для помощника:

1. Отобразить стих/текст
2. Воспринять нажатие на строчку стиха/текста
3. Послать строчку в мобильное приложение выступающего

Функционал приложения для выступающего ещё проще:

1. Получить строчку текста из приложения помощника
2. Отобразить полученную строчку на экране

Данные отдельные пункты смехотворно просты в реализации, если не принимать во внимание, что пользователь инициирует посылает данные

команду, а выполняется команда в другом приложении, куда поступают данные (рис. 8.2). Чтобы обеспечить взаимодействие приложений, мы будем использовать специальный набор компонентов, реализующих технологию App Tethering, т.е. связывания приложений.

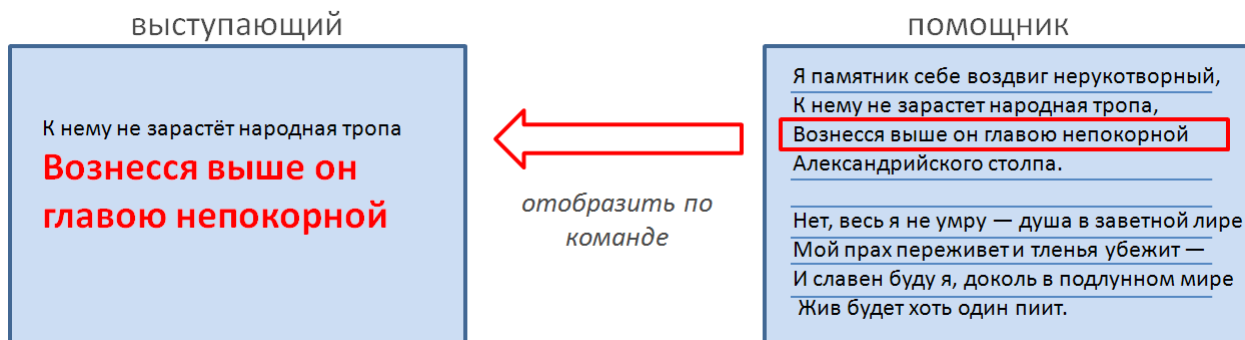


Рис. 8.2. Схема взаимодействия двух приложений

Для связывания двух приложений при помощи компонентов нужно понимать несколько принципов. Мы не будем углубляться в особенности сетевого взаимодействия, т.к. набор компонентов в Delphi/C++Builder/RAD Studio реализует связывание по WiFi или Bluetooth и позволяет нам сосредоточиться на прикладной задаче. Наша задача достаточно проста: одно приложение выполняет команду другого и получает строку. Принципы тут такие:

- одно приложение может определять наличие в сети другого приложения для взаимодействия;
- если одно приложение нашло другое, то оно устанавливает связь с ним;
- если связь между приложениями установлена, то первое приложение может запустить «действие» второго приложения;
- помимо выполнения «действий», первое приложение может послать некие данные второму приложению.

Указанные выше пункты выполняются парой компонентов: TTetheringManager и TTetheringAppProfile. TTetheringManager управляет соединением, а TTetheringAppProfile — выполнением команд и передачей данных.

Сейчас мы начнём создание не одного, а сразу двух приложений. Создадим папку и назовём её «Project 8». После этого запустим Delphi IDE и выберем File->New->Multi-Device Application — Delphi, затем начнём с пустого проекта Blank Application. Сохраним проект в созданную папку, а имена оставим по умолчанию. Сейчас именно такой подход поможет избежать путаницы. Пусть под номером 1 у нас будет обозначаться всё, что принадлежит к приложению для выступающего, т.к. именно он есть «номер 1». Помогаящий будет «номер 2». После сохранения проекта создадим второй, но не надо торопиться и отправляться в главное меню. Найдём панель Project Manager, которая расположена справа вверху, выделим мышью надпись ProjectGroup1 и кликнем правой кнопкой. После этого из контекстного меню выберем пункт Add New Project..., а затем в окне выбора типа проекта — Multi-Device Application (рис. 8.3). Второй проект тоже будет Blank Application.

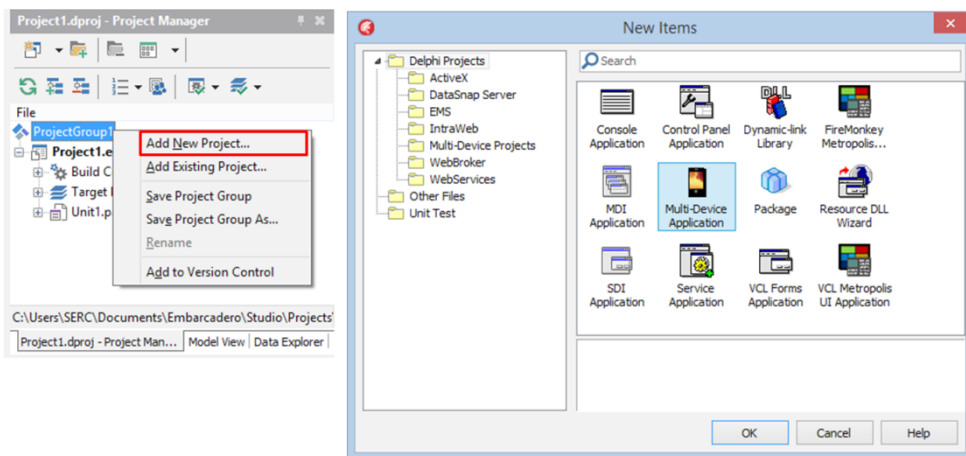


Рис. 8.3. Добавление нового проекта в проектную группу

Обратите внимание, что теперь мы не создаём новый проект «с нуля», а добавляем новый проект в «проектную группу» Project Group. Именно так делать правильно, когда мы одновременно создаём несколько приложений. Сохраним второй проект и проектную группу без изменения имён. Помним, что 2 цифра будет относиться к приложению для помощника. При необходимости мы теперь будем загружать не Project1 или Project2, а сразу всю группу проектов ProjectGroup1. В этом случае будут открываться сразу два наших проекта.

Сначала создадим два проекта как независимые приложения, а потом добавим средства взаимодействия App Tethering. Каждый раз, когда вы переключаетесь к первому или второму проекту, выбирайте его в Project Manager. Одного клика мыши недостаточно, нужно делать два щелчка (рис. 8. 4). Часто даже опытные программисты путаются в проектах, поэтому смотрите внимательно, Project1 или Project2 является активным в Project Manager.

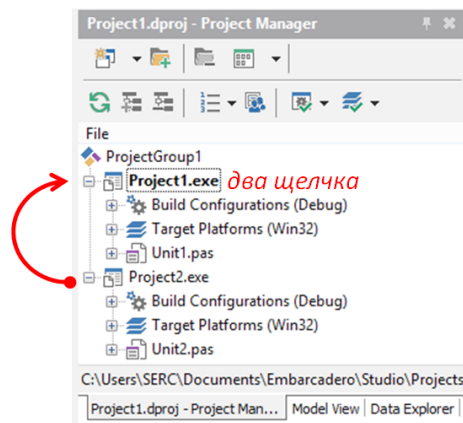


Рис. 8.4. Переключение между проектами в группе

Выберем Project 1 двойным щелчком и займёмся дизайном формы Form1. Конкретно для данного интерфейса минимализм станет нашим девизом. Выступающему точно будет не до кликов по элементам управления интерфейса. Всё должно происходить само собой, поэтому никаких инстру-

ментальных панелей, кнопок и списков с прокрутками. Поскольку праншет обычно держат одной рукой в такие минуты, приложение должно быть размечено точно под портретную ориентацию.

На Form1 проекта Project1 разместим две метки, «прижав» их к верху формы. Отступите слева на 20 пунктов за счёт свойства Margins, а размер шрифта подберите так, чтобы строчки хорошо читались с планшета на расстоянии с вытянутой руки (рис. 8.5), что легко проверить, запустив приложение на устройстве.

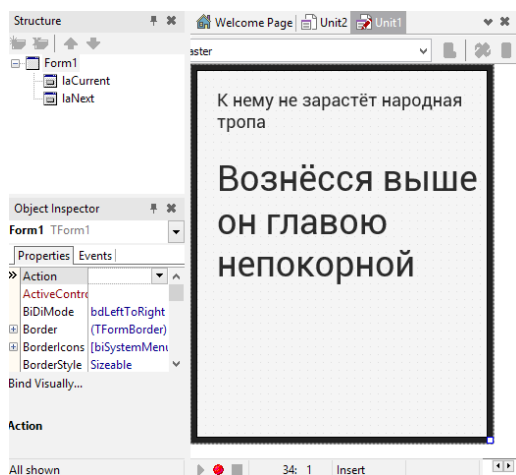


Рис. 8.5. Разметка интерфейса приложения № 1

Назовите верхнюю метку `laCurrent`, она будет отображать текущую строчку. Следующая метка `laNext` показывает текст, который диктор будет озвучивать в следующий момент. Еще раз освежим логику работы по рис. 8.2. Сохраним и запустим проект на реальном устройстве.

Теперь перейдём ко второму приложению. Его можно также разметить для запуска на планшете, что вполне приемлемо. Выступающий стоит на сцене перед аудиторией или камерой, а помощник находится за кулисами или в зале и последовательно отправляет подсказки. Такой вариант неплох, но только для работы с одним стихом или монологом. А теперь пред-

ставьте, если текст достаточно длинный, причём выступают два человека, и нужно подсказывать попеременно обоим. Или один человек будет читать несколько стихотворений. Тогда уже приложение для помощника должно быть достаточно сложным, а режим работы более напряженным. Для этого случая лучше подойдёт ноутбук и приложение под Windows или Mac OS. Но нам не надо особо думать над этим сейчас. Мы будем разрабатывать обычное мульти-платформенное приложение № 2, которое в любой момент можно перекомпилировать и под планшетное устройство, и под Windows, и под Mac OS. В этом вся сила Delphi/C++Builder/RAD Studio.

Логически интерфейс приложения № 2 будет разбит на две части. В левой части будет панель управления. Поместим на форму обычную панель и установим свойство Align в Left. На оставшееся место разместим компонент TListBox со свойством Align в Client. На панели расположим элементы управления, как показано на рис. 8.6. В свойство Items компонента списка загрузите текст стихотворения. Название и тип элементов управления зададим согласно по рис. 8.6. На панели paLeft формы Form2 должны быть две кнопки и два списка TListBox (которые пока выглядят как пустые прямоугольники), роль которых поясним ниже.

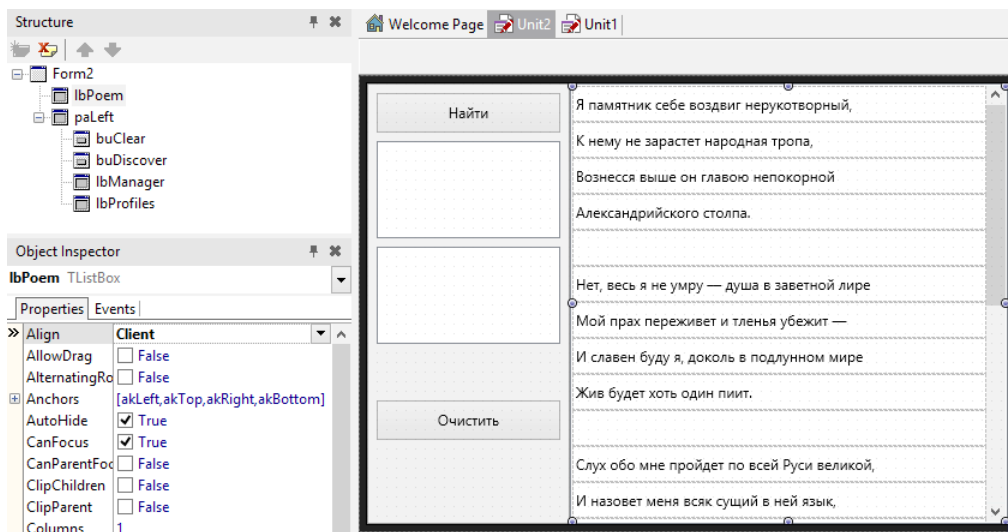


Рис. 8.6. Разметка интерфейса приложения № 2

Кнопка buDiscover будет искать устройства для подключения. Список lbManager покажет устройства, на которых запущено приложение № 1. Список lbProfiles отобразит профили выбранного приложения, кнопка с названием buClear будет очищать содержимое меток в приложении № 1. Пока назначение кнопок не так очевидно даже после такого объяснения, поэтому сейчас углубимся в изучение работы технологии App Tethering.

Всё сводится к работе двух компонентов: TTetheringManager, который устанавливает связь между приложениями, и TTetheringProfile, который отвечает за вызов методов и передачу данных. Добавим пару компонентов TTetheringManager и TTetheringProfile на Form1 первого проекта и Form2 второго проекта. Для второго проекта и Form2 поменяйте компонентам имена так, чтобы они имели цифру 2 в названии (рис. 8.7). Каждому из компонентов TetheringAppProfile выберите свойство Manager из списка. Вариант выбора будет один: TetheringManager1 для TetheringAppProfile1 и TetheringManager2 для TetheringAppProfile2.

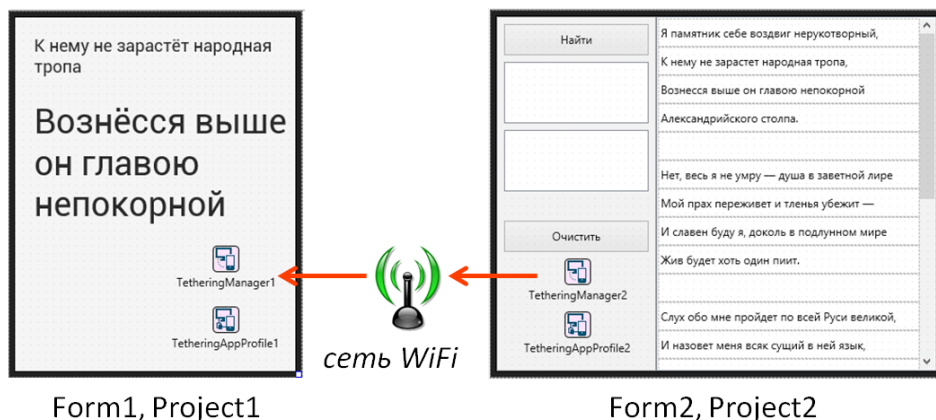


Рис. 8.7. Использование TTetheringManager и TTetheringProfile

Выберем Project1, в качестве целевой платформы для сборки приложения будем использовать именно ту ОС, которая соответствует подключённому мобильному устройству. Раскройте узел Android или iOS, а затем

выделите устройство. Скомпилируйте и запустите Project1 на мобильном устройстве. Проверьте, включён ли WiFi. Пусть пока мобильное приложение Project1 работает на устройстве, а мы займёмся Project2. На кнопку «Найти» введём следующий отклик:

```
procedure TForm2.buDiscoverClick(Sender: TObject);
var
  i: integer;
begin
  // проход по списку подключённых менеджеров внешних
  устройств
  for i:= TetheringManager2.PairedManagers.Count-1 downto 0 do
  begin
    // отключить менеджеры внешних устройств TetheringManager2.
    UnPairManager(TetheringManager2.PairedManagers[i]);
  end;
  // очистить список менеджеров внешних устройств
  lbManagers.Clear;
  // запустить поиск менеджеров внешних устройств
  TetheringManager2.DiscoverManagers;
end;
```

Особо пояснять код нет необходимости, всё написано в комментариях. Но имейте ввиду, код написан так, чтобы он всегда работал корректно вне зависимости от того, является ли это первым нажатием на кнопку или нет. Можно было ограничиться лишь единственной последней строчкой, где вызывается метод поиска менеджеров внешних устройств. Мы не ищем внешние устройства, мы ищем менеджеров внешних устройств. Это означает, что может быть много устройств в сети, но определяться будут только те, где запущено приложение Project1, а в нём есть компонент TTetheringManager. Но представьте, что мы нашли устройства, а потом ещё раз нажали на кнопку «Найти». Тогда сначала мы отсоединяем всех ранее подключённых менеджеров и очищаем список, а лишь затем (как в первый раз) запускаем поиск новых устройств. Отключаем мы их, естественно, от нашей программы Project2, а не от сети.

После этого выберем TetheringManager2 и реализуем отклик на событие OnEndManagerDiscovery. Откуда взялось данное событие? Зачем понадобилось запускать DiscoverManagers, а потом ждать события окончания

процесса поиска? Сделано это очень разумно и вот почему. Мы запускаем процесс поиска менеджеров внешних устройств, а данный процесс может быть длительным. Если это сделать командой с ожиданием результата, то наше приложение временно «повиснет» на продолжительный период поиска новых устройств и не будет ничего другого делать, включая отклик на действия пользователя. Поэтому мы запускаем поиск, а проверяем его результаты по его окончанию, когда наступит соответствующее событие. Закодируем отклик следующим образом:

```
procedure TForm2.TetheringManager2EndManagersDiscovery(const
Sender: TObject; const ARemoteManagers: TTetheringManagerIn-
foList);
var
i: integer;
begin
  // очистка списка менеджеров
  lbManagers.Items.Clear;
  // проход по найденным медежерам внешних устройств
  for i:= 0 to ARemoteManagers.Count-1 do
    begin
      // добавление текста текущего менеджера в список
      lbManagers.Items.Add(ARemoteManagers[i].ManagerText);
    end;
  end;
```

Теперь сохнаим и запустим Project2 как Windows-приложение. Вполне возможно, что ваша система покажет предупреждение, как на рис. 8.8. Действительно, сетевая активность может выглядеть подозрительной для брандмауэра операционной системы. Поскольку приложение наше и мы ему доверяем, разрешим доступ. После нажатия на кнопку «Найти» (спустя некоторое время) в списке lbManagers появится текст TetheringManager1. Это означает, что наше приложение Project2 по сети нашло мобильное приложение Project1, а компонент TetheringManager2 готов установить связь с TetheringManager1.

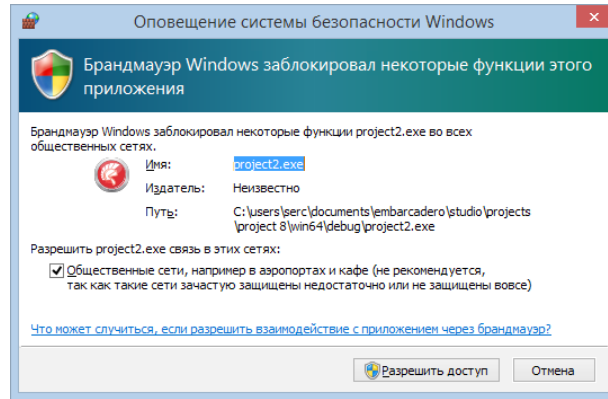


Рис. 8.8. Возможное предупреждение брандмауэра

Сейчас очень важно не потерять нить рассуждений, поэтому уделим некоторое время на повторное рассмотрения принципа работы системы, которая включает:

1. Мобильное приложение **Project1**
2. Компонент **TetheringManager1** в приложении **Project1**
3. Компонент **TetheringAppProfile1** в приложении **Project1**
4. Мобильное приложение **Project2**
5. Компонент **TetheringManager2** в приложении **Project2**
6. Компонент **TetheringAppProfile2** в приложении **Project**

Схема работы на начальном этапе представлена на рис. 8.9. Когда мы нажали на кнопку «Найти» в приложении **Project2**, то мы отсоединили предыдущие подключения и очистили список **lbManagers**. Затем поступила команда **TetheringManager2.DiscoveryManagers**. После этого начался поиск в сети приложений, которые содержат в себе компонент **TTetheringManager**. В результате этих поисков было найдено приложение **Project1**, т.к. в нём есть компонент **TetheringManager1**.

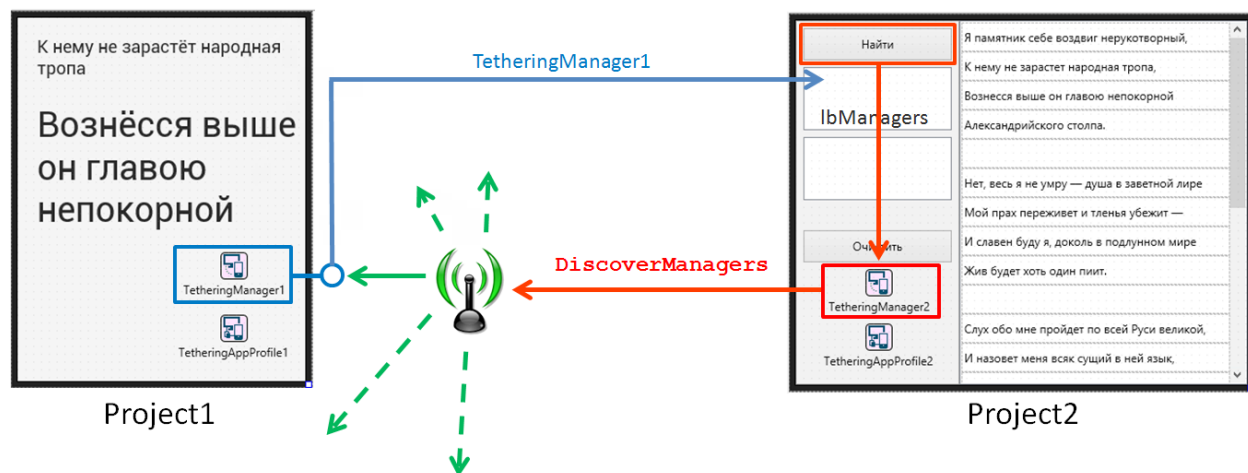


Рис. 8.9. Схема поиска менеджеров приложений в сети

Поиск закончился, и у компонента TetheringManager2 в приложении Project2 сработало событие OnEndManagersDiscovery. В процедуре отклика мы в качестве параметра получили список обнаруженных в сети менеджеров `ARemoteManagers: TTetheringManagerInfoList`. Далее мы просмотрели этот список, который расположен в памяти, и заполнили визуальный компонент `IbManagers` на форме `Form2` приложения Project2. Никаких подключений мы не сделали, а просто получили список возможных вариантов. Если в вашем распоряжении есть несколько мобильных устройств, можете провести эксперимент. Запустите на них приложение Project1, а затем на вашем компьютере Project2. Нажмите кнопку «Найти» и посмотрите, какие менеджеры будут найдены. Конечно, все устройства должны быть подключены к одной сети Wi-Fi.

Теперь пользователь Project2 в списке менеджеров `IbManagers` увидел один TetheringManager1, т.к. у нас в сети найдено одно приложение Project1. Теперь нужно их «спарить», т.е. подключить один TetheringManager1 к другому TetheringManager2. Выделяем компонент `IbManagers` и на его событие `OnClick` пишем следующий код:

```

procedure TForm2.lbManagersClick(Sender: TObject);
begin
  // «спаривание» менеджера TetheringManager2
  // с менеджером внешнего приложения по номеру в списке
  TetheringManager2.PairManager(
    TetheringManager2.RemoteManagers[lbManagers.ItemIndex]
  );
end;

```

Здесь всё очевидно, но чтобы было ещё проще, лучше начать читать код с конца. Конструкция `lbManagers.ItemIndex` означает индекс выбранного элемента в списке. Вспоминаем, что индексы считаются с 0, что легко проверить отладчиком. Конструкция `TetheringManager2.RemoteManagers [...]` даёт нам доступ к списку менеджеров внешних удалённых приложений по индексу в квадратных скобках. Наконец, `TetheringManager2.PairManager(...)` подключает выбранный менеджер из списка внешних удалённых приложений к нашему внутреннему `TetheringManager2`.

Как только произошло подключение, автоматически начинается поиск профилей внутри подключённого приложения. Этот процесс показан на рис. 8.10. Внутри `Project1` есть только один профиль `TetheringAppProfile1`, и он будет обязательно найден. Рис. 8.10 похож на рис. 8.9, только раньше происходил поиск по сети всех приложений с компонентами `TetheringManager` внутри, а теперь уже внутри выбранного приложения происходит поиск профилей приложения.

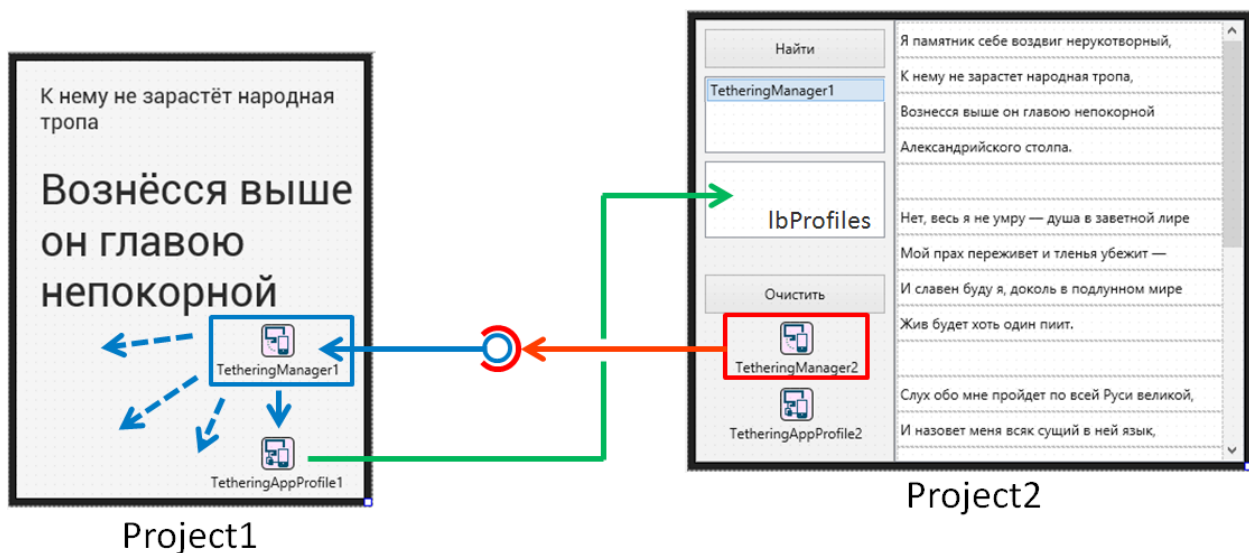


Рис. 8.10. Схема поиска профилей внутри приложения

Для получения информации о нём воспользуемся событием `OnEndProfilesDiscovery`. Событие происходит автоматически по завершению поиска профилей внутри приложения. Нам нужно просто в процедуре отклика на данное событие заполнить список `lbProfiles` найденными профилями:

```
procedure TForm2.TetheringManager2EndProfilesDiscovery(const
Sender: TObject; const ARemoteProfiles: TTetheringProfileInfo-
List);
var
i: integer;
begin
  // очистка списка профилей
  lbProfiles.Items.Clear;
  // проход по найденным профилям приложения
  for i:= 0 to ARemoteProfiles.Count-1 do
  begin
    // добавление текста текущего профиля в список
    lbProfiles.Items.Add(ARemoteProfiles[i].ProfileText);
  end;
end;
```

Объяснять данный код даже скучнее, чем набирать его при помощи клавиатуры. Однако эту работу тоже должен кто-то делать! Посмотрим на параметры процедуры. Нас интересует второй параметр. Он содержит список профилей, обнаруженных в приложении. Обычным образом просматриваем в цикле список этих профилей и добавляем уже в визуальный список `lbProfiles` на форме. А что будет нескучным, так это объяснение понятия «профиль».

Для начала ещё раз посмотрим на рис. 8.9 или 8.10. Каждое приложение имеет пару компонентов `TetheringManager` и `TetheringAppProfile`. Первый нужен для связи с другими приложениями, второй — для выполнения команд и передачи/получения данных. Одно приложение может иметь несколько профилей. Казалось бы — зачем? Рассмотрим обратную ситуацию, когда приложение имеет только один «профиль». Допустим, функционал приложения богатый. Оно может и получать подсказки для выступающего, загружать тексты целиком, получать вибро-оповещения для подготовки актёра к выходу, включать или выключать автоматическую запись голоса и т.д. Если мы все эти действия добавим в один профиль, то рано или поздно возникнет путаница. Чтобы приложение имело ясную и чёткую структуру, в таком случае нужно создать несколько профилей:

- профиль **proHints** для подсказок;
- профиль **proTexts** для текстов;
- профиль **proVoice** для записи голоса;
- ...

Логика рассуждений здесь такая же, как и для компонента `TActionList`. Обычно достаточно одного такого компонента, но это совсем не обязательно. Если функционал приложения большой, действий много (в профессиональных приложениях их порядка нескольких сот, да-да, именно несколько сот «полезных функций»), то и «списков действий» может быть несколько. Сейчас главное запомнить, что ничего странного в возможности иметь несколько профилей в одном приложении нет.

Когда пользователь увидел список профилей приложения, и даже если он всего один, то мы обработаем событие `OnClick` данного списка `lbProfiles`, чтобы соединить профиль `TetheringAppProfile1` в приложении `Project1` с профилем `TetheringAppProfile2` в приложении `Project2`:

```
procedure TForm2.lbProfilesClick(Sender: TObject);
begin
  TetheringAppProfile2.Connect (
    TetheringManager2.RemoteProfiles[lbProfiles.ItemIndex]
  );
end;
```

Не нужно пугаться того, что мы разбили одну строку кода тела процедуры на три. В противном случае у нас получилась бы очень длинная строка. Есть противники такого разбиения, есть сторонники. Выберите свой стиль форматирования сами. А в остальном опять всё понятно, если читать код с конца. Берётся индекс профиля в визуальном списке `lbProfiles.ItemIndex`, затем по данному индексу из списка в памяти находится нужный профиль `TetheringManager2.RemoteProfiles[...]`, а потом соединяется с ним профиль второго приложения `TetheringAppProfile2.Connect(...)`. Для окончательной ясности изучите схему соединения профилей на рис. 8.11.

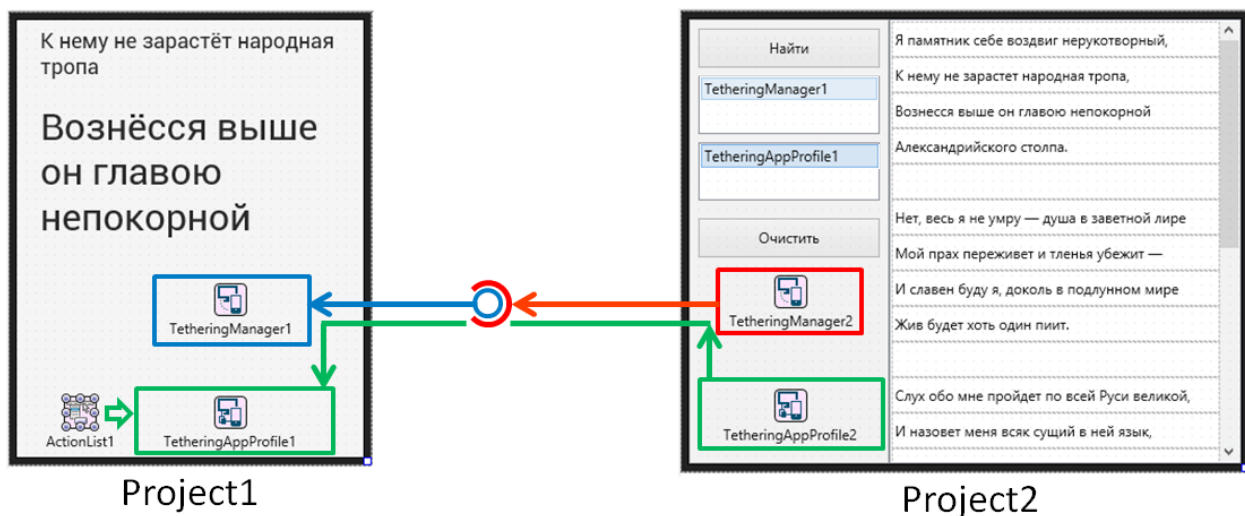


Рис. 8.11. Схема подключения профилей

Что-то новое появилось на форме Project1. Этот новый компонент — ActionList1. Сам по себе профиль TetheringAppProfile1 никаких действий не выполняет, зато через него можно получить доступ ко внутренним «действиям» приложения, хранящимся в ActionList1. Перейдём на Form1 проекта Project1 и добавим TActionList. В нём создадим новое действие Action1 с событием OnExecute:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
  laCurrent.Text:= ' ';
  laNext.Text:= ' ';
end;
```

Эта процедура очищает текст в метках, что нужно для сброса подсказок перед выступлением, а также в паузах. Теперь свяжем профиль с действием, т.е. TetheringAppProfile1 с ActionList1. Выделим TetheringAppProfile1, в Object Inspector найдём свойство Actions и добавим новое действие. Оно будет «профильным», а не «действующим», поэтому нам придётся свойством Action связать его с Action1. Подробно схему подключения можно посмотреть на рис. 8.12.

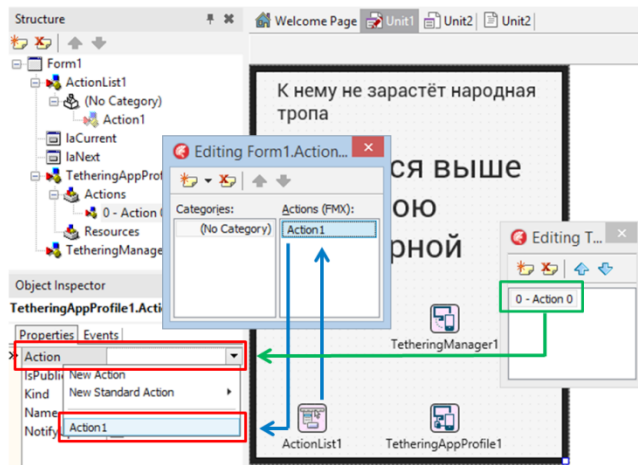


Рис. 8.12. Связывание «профильного» действия с «действующим»

Если ещё не понтно, то повторим снова. У нас есть `ActionList1` с действием `Action1`. Это действие на событие `OnExecute` выполняет некий код. Можно внутри приложения сделать кнопку и присвоить ей событие. Тогда процедура `OnExecute` будет срабатывать на нажатие кнопки. Теперь представим, что кнопка находится в другом приложении. Тогда действие `Action1` нужно связать с профилем, а само действие можно будет вызывать по кнопке из другого приложения. Сейчас простыми действиями докажем, что это работает. Перейдите в `Project2` и на кнопку «Очистить» введите следующий код:

```
procedure TForm2.buClearClick(Sender: TObject);
begin
  TetheringAppProfile2.RunRemoteAction(
    TetheringManager2.RemoteProfiles[lbProfiles.ItemIndex],
    'Action1'
  );
end;
```

Сначала запустим **Project1** на мобильном устройстве, а затем вернёмся в **Delphi**, выберем **Project2** и запустим его как приложение **Windows**. Посмотрим внимательно на рис. 8.11, вспомним, где какие компоненты расположены, а затем уверенно выполним следующие шаги:

- Нажмём кнопку «Найти»
- Выберем **TetheringManager1** из списка менеджеров
- Выберем **TetheringAppProfile1** из списка профилей
- Нажмём кнопку «Очистить»

В результате мы увидим, как метки мобильного приложения очищаются. Осталось дело за малым — послать строчки с подсказками в мобильное приложение. Здесь просто «действием» не обойтись, т.к. выполнение `Action` не предусматривает передачу данных. Опять вернёмся в `Project1`, выберем `TetheringAppProfile1`. Найдём событие `OnResourceRecieved`, оно произойдёт, когда из `Project2` в `Project1` будет что-то прислано по сети. Работаем в паре: `Project2` будет посылать что-то в `Project1`. Это «что-то» есть

некий ресурс, т.е. какой-то объем данных, который может быть различных типов. Но нас интересует посылка строки.

Project2 посылает строку. Project1 получает ресурс в виде строки автоматически, об этом заботится наша связка компонентов TetheringManager1 и TetheringAppProfile. Как только строка получена, происходит событие OnResourceRecieved, откликом на данное событие будет процедура, получающая ресурс в виде строки. Сначала научимся посылать строки из Project2. Как только помогающий пользователь кликнул на строчку стиха, срабатывает событие OnClick. В данном событии мы пошлём на планшет выступающего две строки — текущую и следующую. Напишем код процедуры так:

```
procedure TForm2.lbPoemClick(Sender: TObject);
begin
  TetheringAppProfile2.SendString(
    TetheringManager2.RemoteProfiles[lbProfiles.ItemIndex],
    'CurrentString',
    lbPoem.Items[lbPoem.ItemIndex]
  );
  TetheringAppProfile2.SendString(
    TetheringManager2.RemoteProfiles[lbProfiles.ItemIndex],
    'NextString',
    lbPoem.Items[lbPoem.ItemIndex+1]
  );
end;
```

Такую «ступенчатую» форму записи кода мы уже видели, она достаточно удобна. Тело процедуры содержит два одинаковых вызова метода SendString компонента-профиля TetheringAppProfile2. В качестве аргументов передаются три значения. Первое из них это — выбранный удаленный профиль в списке профилей RemoteProfiles по индексу в списке lbProfiles.ItemIndex. Второе значение — просто некая строка 'CurrentString', третье — текущая по индексу lbProfiles.ItemIndex строчка из стиха в компоненте lbPoem. Второй вызов аналогичен, только вторым аргументом посылается значение 'NextString'. Данные значения служат для того, чтобы в Project1 было понятно, какую строку передали: текущую или следующую.

Перейдём в Project1 и на событие OnResourceReceived компонента TetheringAppProfile1 введём следующий код:

```
procedure TForm1.TetheringAppProfile1ResourceReceived(const
Sender: TObject; const AResource: TRemoteResource);
begin
  if AResource.Hint = 'CurrentString' then
  begin
    laCurrent.Text:= AResource.Value.AsString;
  end;
  if AResource.Hint = 'NextString' then
  begin
    laNext.Text:= AResource.Value.AsString;
  end;
end;
```

В приведённом коде тоже всё понятно. Мы в качестве параметра получаем «ресурс» AResource. Что мы знаем, об этом некоем ресурсе? Во-первых, он строкового типа. Во-вторых, вместе со строкой мы передали подсказку или Hint. По значению этой подсказки Hint мы понимаем, какая это строка, текущая или следующая. Затем присваиваем ресурс в виде текста соответствующей метке. Можно сохранить и запустить сначала мобильное приложение Project1, а затем и Windows-приложение Project2. Пока собирается Project1, можно посмотреть схему отправки ресурса-строки (рис. 8.13).

Представьте себе, что в одном почтовом отделении упаковывают некую посылку, которая всего есть некий «ресурс». В коробку кладут строку с описанием, что это за строка. Потом посылают в другое почтовое отделение. На втором почтовом отделении работники сидят и скучают, пока им не доставили новую посылку. Когда она доставлена, срабатывает событие. Коробку распаковывают, достают оттуда строку и читают подсказку, что с ней нужно сделать. Ничего сверхестественного нет. Всегда можно придумать объяснение из реальной жизни. Конечно, можно представить себе почтовое отделение или красивую коробку с написанным на ней адресом. Можно представить себе почтового голубя с листочком, привязанным к лапке. Сухие строчки программного текста выглядят не так живописно, хотя суть их работы та же.

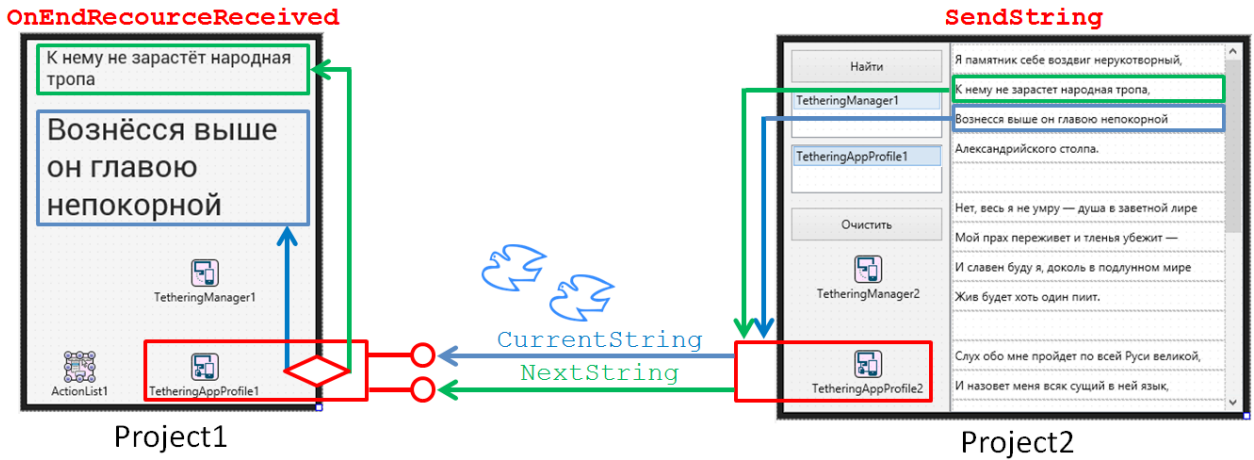


Рис. 8.13. Схема посылки ресурсов в виде строк

Сохраняем и запускаем мобильное приложение и приложение на настольном компьютере. После тривиальной операции спаривания менеджеров в приложениях можно приступить к его тестированию. Как только помощник (он же № 2) кликает на строчке текста в своём приложении, то из второго приложения посылается строка в первое для прочтения выступающего (он же № 1).

Для выполнения большого комплексного проекта можно использовать два проекта. Первый — помогает актёрам выучить их роли. Второй — обеспечивает их подсказками во время исполнения. В первом проекте можно выполнить исследовательскую работу, по поиску оптимального алгоритма запоминания. Во втором проекте найти место для проведения исследований не так просто: слишком утилитарную роль играет приложение «суфлёр». Однако второй проект позволяет овладеть техникой сетевого взаимодействия прикладных программ. После отработки данного задания можно вернуться к предыдущим проектам и добавить возможности совместного использования мобильного устройства и настольного ПК в единой системе:

- для дневника наблюдений можно передавать записанные данные с планшета на настольный компьютер;
- для запоминания стихов можно сделать так, когда по команде с одного устройства скрывается часть текста на другом, т.е. процесс запоминания стиха контролируется другим пользователем;
- для энциклопедии моделей молекул химических веществ можно использовать такую идею: на ПК в трехмерном исполнении демонстрируются модели молекул, а управление данным процессом осуществляется с мобильного устройства, играющего роль пульта дистанционного управления;
- для проекта с тестом можно использовать возможности взаимодействия приложений для автоматической отправки результатов тестирования в общее приложение, которое будет хранить данные обо всех опрошенных учениках.

Хотя каждая глава была посвящена отдельному проекту, рассмотренные в них методики решения задач по реализации определенных функций можно комбинировать между собой. Если вы проработали каждый из проектов, то теперь в вашем распоряжении мощный арсенал средств мобильной разработки. Пользуйтесь этим как для развития данных, так и создания новых проектов.

Заключение

9.1. Что делать дальше или планы на будущее

Мы дошли до последней главы данной книги. Кто-то из читателей скрупулезно и внимательно выполнил все проекты от начала до конца. Теперь о них можно говорить как о практически состоявшихся программистах. Конечно, нужно постоянно совершенствовать свои теоретические знания и практические навыки. Однако если даже школьник может реализовать один из проектов последних глав от начала до конца по памяти, то остаётся только пожелать профессиональных успехов. Как минимум вы уже смело можете считать, что программирование вам далось сравнительно легко. Не обязательно становиться именно разработчиком программ, есть ряд других профессий, тесно связанных с программированием. Работа в сфере информационных технологий требует понимания принципов создания программного обеспечения, а это невозможно без свободного владения хотя бы одним из языков программирования.

Некоторые читатели не имели времени, возможностей или соответствующей подготовки, чтобы самостоятельно проработать весь материал книги. Но даже в этом случае вы сделали важный шаг вперёд на пути профессионала в области информационных технологий. Любой выполненный проект, даже если это потребовало концентрации всех усилий, делает реализацию следующего проекта гораздо легче. Не останавливайтесь на достигнутом, идите дальше! Второй, третий, четвёртый проект — скоро вы почувствуете лёгкость и в создании интерфейсов, и в ручном кодировании. В какой-то момент вы поймёте, что главные вопросы не «как сделать», а «что сделать». Процесс обучения программированию можно сравнить с обучением плаванию или езде на велосипеде. В какой-то момент вы перестанете думать о блоках `begin-end`, потерянных операторах, ошибках компиляции, как вы уже не помните, как гребки руками согласуются с движением ног

при плавании или как удержать равновесие при езде на велосипеде. Всё начнёт происходить естественным образом.

Реализованные вами проекты вполне заслуживают показа друзьям, знакомым и даже потенциальным работодателям. Любой начинающий программист всегда имеет преимущества, если может показать созданные им программы. Но при этом следует отметить, что программист должен постоянно совершенствовать свои навыки. С одной стороны, есть ещё много, что следует выучить. С другой, любой перерыв в развитии означает потерю навыков. Теперь настало время подумать о том, куда идти дальше. Следует ли выучить ещё один язык программирования или продолжить применение Delphi/C++Builder/RAD Studio? Какие ещё технологии нужно изучить? Есть ли смысл продолжить развитие какого-либо проекта из книги или начать новый проект с самого начала? Сейчас мы подробно рассмотрим данные вопросы.

Работа над достаточно сложным проектом, а последние главы были посвящены именно таким, действительно требует значительных усилий. Программист должен быть исключительно внимательным при вводе кода или работе с интерфейсом, при этом держа в голове общий замысел и не теряя общую алгоритмическую нить. Часто возникают произвольные ошибки компиляции, которые требуют поиска и исправления. Иногда приложение запускается, но работает неверно. Приходится запускать отладчик и буквально заниматься процеживанием элементов кода через мелкое сито наших здравых рассуждений. После такой достаточно трудной работы хочется сделать большой перерыв и отдохнуть.

Здесь есть пара хороших новостей. Первое, желание немного забыть про проект — естественное. Спросите любого профессионального программиста, обычно он работает неравномерно. Периоды крайне интенсивного, почти экстремального ритма работы сопровождаются периодами достаточно неспешного обдумывания алгоритмов, поиска оптимального варианта интерфейса, реализации мелких «красивостей». Можно использовать данный приём. Если чувствуете, что процесс кодирования идёт вялый,

переключитесь на тонкую настройку интерфейса. Подвигайте кнопки, меняйте местами метки, обдумайте более короткие и звучные надписи. Соберите проект под разные типы устройств. В конечном итоге, составьте подробный план тестирования будущих, пока нереализованных функций. Как вариант, можно прочитать хорошую статью про программирование. Не обязательно всё понять с первого раза — мы же только учимся!

Любой язык программирования предоставляет большое количество различных возможностей. Операторы, типы данных, конструкции, принципы построения программ — всего этого достаточно много в каждом из существующих и активно используемых языков. Добавим библиотеки уже готовых функций или визуальных компонентов. Всё это представляет собой огромный объем знаний, и вряд ли есть какой-то один человек, кто полностью в этом разбирается. Если вы используете Delphi и Object Pascal, то здесь проще. Pascal задумывался как средство разработки для инженеров, которые раньше не программировали. Он должен был быть прост в обучении и эффективным в применении. Этот язык учится достаточно легко и его грамотное использование — вопрос нескольких занятий. Полагаю, многие по проработке материала книги в этом убедились.

Если вы начали использовать C++Builder как отдельно, так и в составе RAD Studio, то вполне могли ощутить заметные сложности. Язык C++ создавался для профессионалов, поэтому скорость его изучения и простота синтаксиса не были основной целью создателя. То же самое можно сказать о возможностях — они весьма и весьма разнообразны. Сам автор C++, Бьерн Страуструп, неоднократно заявлял, что не надо знать весь язык. Нужно владеть только тем, что требуется для реализации конкретного алгоритма, решения конкретной задачи. Отсюда можно сделать важный вывод, что лучше сначала ставить себе какую-то задачу, а потом изучать возможности языка или среды разработки, которая позволит её решить. Конечно, есть люди, которым нравится просто читать книгу, справочник или статьи, посвященные конкретному языку программирования или среде разработки. И это тоже полезно! Делайте так в периоды, когда требуется покой и расслабление.

Во многом ученики привыкли выполнять задания, данные учителями. На вопрос, а что бы вы хотели сделать сами, многие не знают ответа. Дело не в отсутствии творческих способностей, в молодом возрасте они у каждого проявляются достаточно ярко. Просто нет привычки ставить задачу самому себе и своим одноклассникам. При рассмотрении каждого проекта мы старались в конце дать направления дальнейшего развития, но конкретную форму каждый должен придумать сам. Старайтесь развивать свои продукты самостоятельно, т.к. именно это позволяет воспитать самих в себе не просто программистов, но талантливых творцов выдающегося программного обеспечения. Поставьте себя на место пользователя, а это сделать легко, если вы уже разработали приложение. Представьте, как можно улучшить программу, обогатив её новыми возможностями. Черпайте вдохновение от одного факта, что вы уже сделали работающий программный продукт.

Разработка современных программных продуктов это — бесконечное путешествие. Чтобы быть успешным, нужно не просто создать выдающийся продукт, но и поддерживать темп разработки на высоком уровне. Профессиональный программист должен не просто уметь программировать, он должен уметь делать это постоянно. Не пугайтесь, никто не заставляет вас мучительно вводить текст программ каждый день и подолгу. Но настройтесь на напряжённую интеллектуальную деятельность, которая позволит вам зарабатывать большие деньги и обрести уважение.

9.2. Полезная литература

После завершения данной книги весьма рекомендуется еще раз внимательно изучить синтаксис языка Pascal и базовые возможности среды Delphi. Поскольку Delphi и Object Pascal появились достаточно давно и активно использовались в обучении программированию, в Интернете можно найти очень много материалов для начинающих. Мы рекомендуем проработать «методичку», разработанную специально для школ. Её электронная версия расположена по адресу: https://yadi.sk/i/Uqpss_umih4S6 В данной «методичке» нет раздела, посвященного мобильной разработке,

что совсем не умаляет полезность издания. Как мы с вами убедились, при помощи Delphi/C++Builder/RAD Studio мобильные приложения разрабатывать также просто, как и обычные «настольные» приложения. Разница заключена лишь в умении правильно спроектировать интерфейс. А вот знания синтаксиса языка и способность реализовывать алгоритм работы действительно универсальны. Для повышения уровня в плане использования языка программирования весьма полезно временно отойти от мобильных проектов и обратиться к «чистому» знанию. Обязательно найдите время для проработки указанного выше материала.

После того, как вы почувствуете лёгкость в использовании языка программирования Object Pascal в среде Delphi, можно углубить знания о визуальных компонентах. А как понять, что вы уже достаточно опытни? Просто станет гораздо меньше синтаксических ошибок, выявляемых на стадии компиляции проекта. Через определенное время вы начнете писать код абсолютно верно, не задумываясь о «правилах пунктуации» или написании ключевых слов. Некорректный код будет выглядеть «некрасиво», как некрасивыми выглядят свола напесонные с ашипками (надеюсь, вы оценили шутку). Если вам захотелось узнать больше о мобильной разработке, то скачиваем и читаем еще одну книгу по ссылке: <https://yadi.sk/i/gigAvNn6ih4YU>. Эта книга предназначена для старших школьников и студентов. Багаж знаний формируют не только реализованные проекты, но и прочитанные книги. Даже если вы вполне уверены в собственных силах, чтение книг профессиональных авторов значительно повышает уровень программиста.

В качестве дополнительной литературы можно поискать какую-либо другую книгу, посвященную Delphi/C++Builder/RAD Studio на множестве сайтов онлайн-торговли. Однако здесь есть определённая проблема. Период слабого развития Интернет совпал с бурным ростом Delphi/C++Builder/RAD Studio, когда была написана основная масса книг. После этого Интернет стал поистине всеобъемлющим, появились форумы, тематические сайты, возможность публиковать книги в электронном виде, а в последнее время и порталы для размещения онлайн-видео, такие как YouTube. Сама идея написания «бумажных» книг начала себя изживать, т.к. гораздо проще

и быстрее использовать электронные источники информации. Количество издаваемых книг сократилось, и это сокращение совпало с вторым важным этапом в развитии Delphi/C++Builder/RAD Studio, связанным с мобильной разработкой. То же самое можно сказать и про любую другую среду разработки или даже любой программный продукт. Поэтому помимо указанных выше книг, скачиваемых бесплатно, или изданий, приобретаемых в онлайн или обычных книжных магазинах, нужно уметь пользоваться и другими источниками информации.

9.3. Онлайн-источники и информация в Интернет

Благодаря поисковым машинам можно быстро найти нужную информацию, достаточно лишь грамотно сформулировать вопрос. Большинство читателей, мы полагаем, знакомы с техникой использования поисковых машин на основе текстовых запросов. Не забывайте добавлять слово «Delphi», чтобы в найденные ответы касались использования именно этой технологии.

Рассмотрим уже более изощренный приём использования поисковых сервисов. Например, возникла ошибка на этапе компиляции или сборки проекта. Допустим, выбрана в качестве целевой платформы Android, а устройство не подключено. Тогда при запуске мульти-платформенного приложения мы увидим:

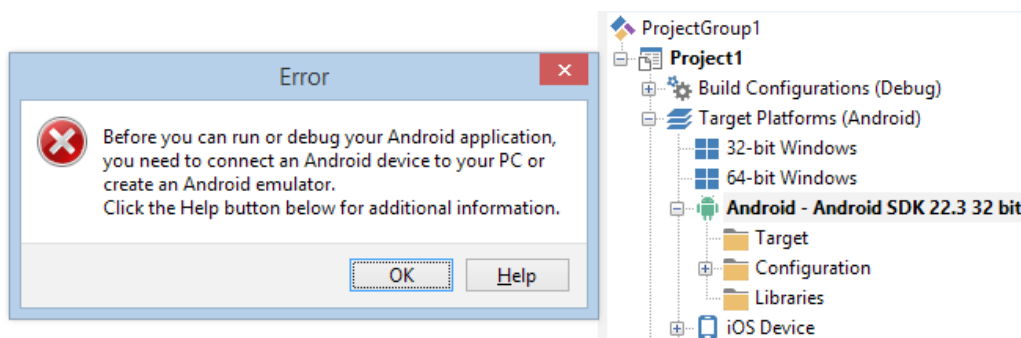


Рис. 9.1. Ошибка запуска при неподключенном устройстве

Не все могут читать с экрана по-английски. Иногда полезно воспользоваться онлайн-переводчиком. Может показаться, что текст ошибки невозможно скопировать в буфер обмена. Но это не так, просто нажмите Ctrl+C, когда окно активно. В буфер обмена попадут строчки, которые можно вставить в «Блокнот» или окно ввода онлайн-переводчика (рис. 9.2.).

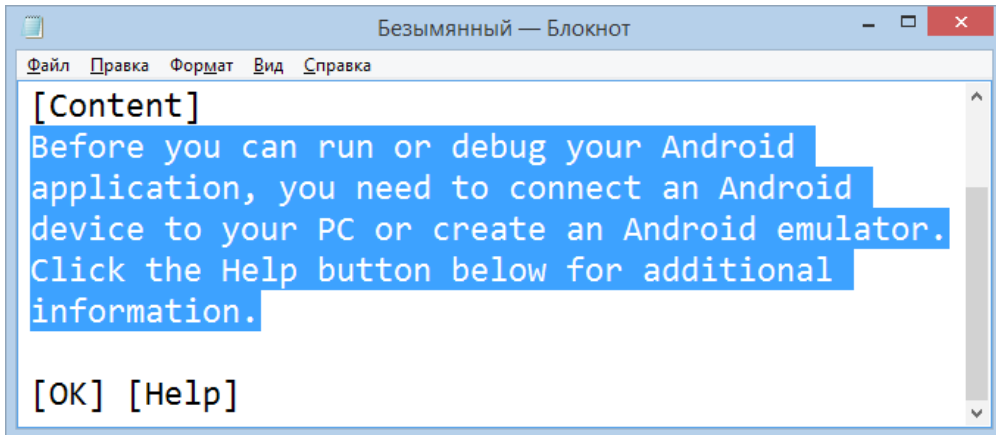


Рис. 9.2. Скопированный текст из окна с ошибкой

Если запустить поиск по введенной строке, предварив его словом «Delphi» (без кавычек), то в результате мы получим набор полезных ссылок. Их порядок может меняться, но в первой десятке мы получим ссылку на официальную документацию от компании Embarcadero, которая в загруженном виде показана на рис. 9.3. На показанной странице подробно описан процесс настройки среды IDE и необходимых элементов программного обеспечения для разработки мобильных приложений, включая под платформу Android. Выполнив указания на данной странице, мы избавимся от ошибки. Подобную процедуру можно проводить с любой непонятной ошибкой, возникшей при использовании Delphi.

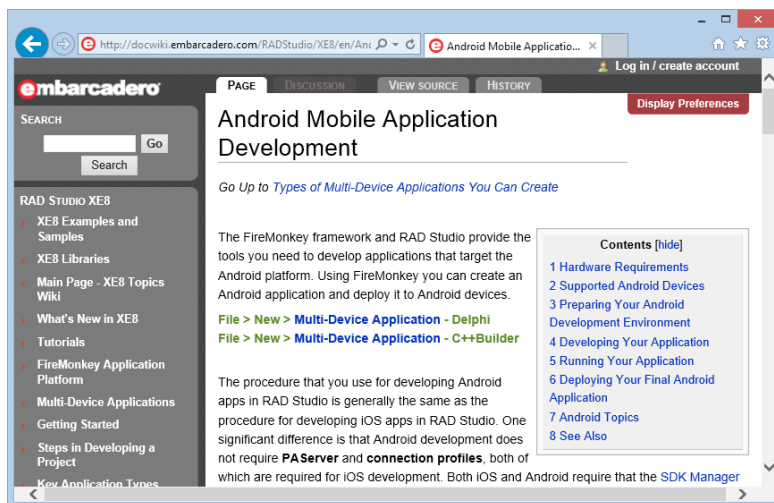


Рис. 9.3. Онлайн-документация от компании Embarcadero

В указанном выше примере поисковая машина выдаст также ссылку на ресурс YouTube, где будет приведён «видео-ответ» на поставленный вопрос об устранении ошибки во время сборки проекта (рис. 9.4).

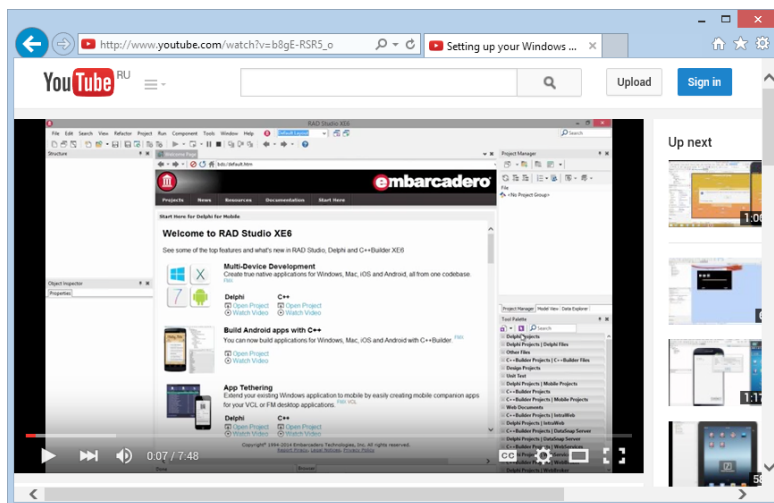


Рис. 9.4. Видео-инструкция по настройке среды

Данный видео-ролик опубликовала компания Embarcadero, которая создала специальный канал на YouTube. По ссылке <http://www.youtube.com/user/EmbarcaderoTechNet> можно получить доступ к этому каналу, где размещены интересные и полезные видео-материалы на тему использования Delphi/C++Builder/RAD Studio, а также мобильной разработки под iOS и Android.

Мы рассмотрели случай ошибки, связанный с настройкой среды. Теперь обсудим еще один пример, когда ошибка уже не в настройке среды, а уже в визуальном программировании. Такая ошибка часто бывает у новичков. Они располагают на форме компонент `TButton`, а потом хотят удалить ему надпись (свойство `Text`), но ошибаются и удаляют название (свойство `Name`). Может ли компонент существовать без названия? Может, хотя мы знаем, что название компонента является переменной ссылочного типа на него. Без имени компонент жить может, но весьма примитивной жизнью. Как только мы попытаемся добавить процедуру отклика через Object Inspector, то тут же получим ошибку, показанную на рис. 9.5.

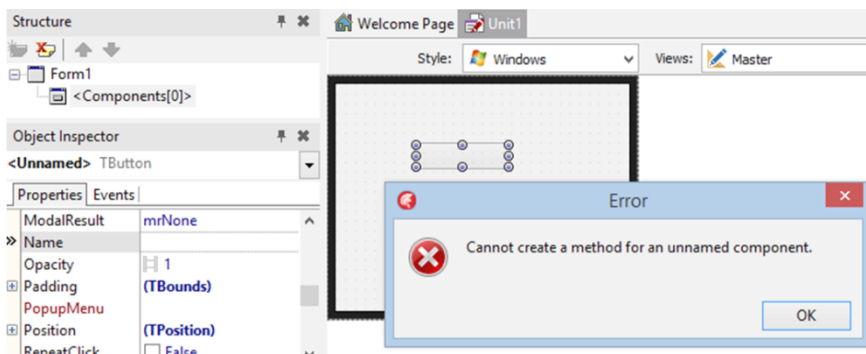


Рис. 9.5. Ошибка у компонента без имени

Конечно, мы только что разобрали причину данной ошибки, поэтому нет необходимости использовать «поисковик» для её диагностики и устранения. Но представим себе, что мы пока ничего не знаем о данной ошибке.

Нажимаем Ctrl+C, затем для контроля вставляем из буфера Ctrl+V в «Блокнот». Теперь мы можем убедиться, что скопирован текст ошибки (рис. 9.6).

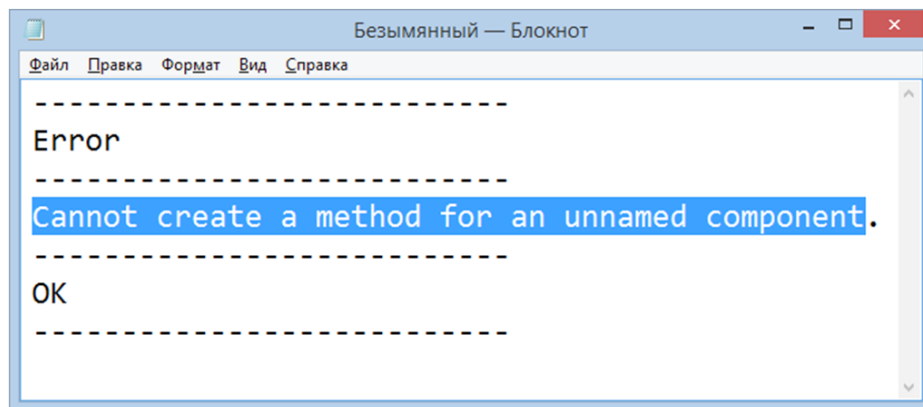


Рис. 9.6. Текст ошибки из буфера

Онлайн-переводчик легко справится с поставленной задачей — «Не удастся создать метод для безымянного компонента», и мы сразу поймём, в чём была причина ошибки.

Может возникнуть проблема компиляции на чисто языковом уровне. Тогда ошибки будут показаны не на отдельном окне, а в нижней части окна среды, закладка Messages (рис. 9.7). Нужно кликнуть на строке с ошибкой не в редакторе кода, а окошка Messages, затем щёлкнуть правой кнопкой мыши и в контекстном меню выбрать Copy. После этого опять текст ошибки попадёт в буфер обмена.

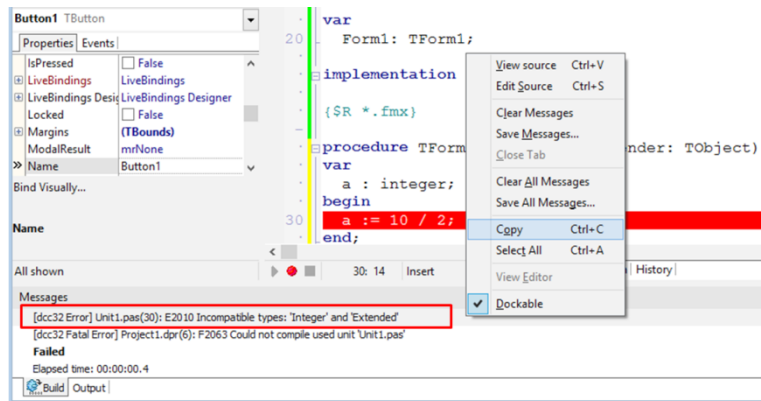


Рис. 9.7. Ошибка компилятора

Из буфера обмена вставляем текст в запрос поискового сервиса (рис. 9.8), затем получаем описание ошибки на русском языке (рис. 9.9). Конечно, это — не официальная документация Embarcadero, часто такие ссылки ведут на форумы программистов. Но не следует ограничиваться только форумами сообщества программистов, т.к. официальная документация на программный продукт компании-производителя предоставляет более достоверную информацию.

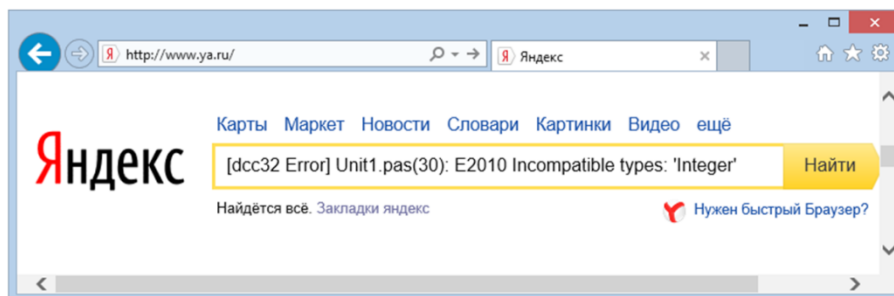


Рис. 9.8. Поиск ошибки при помощи «поисковика»

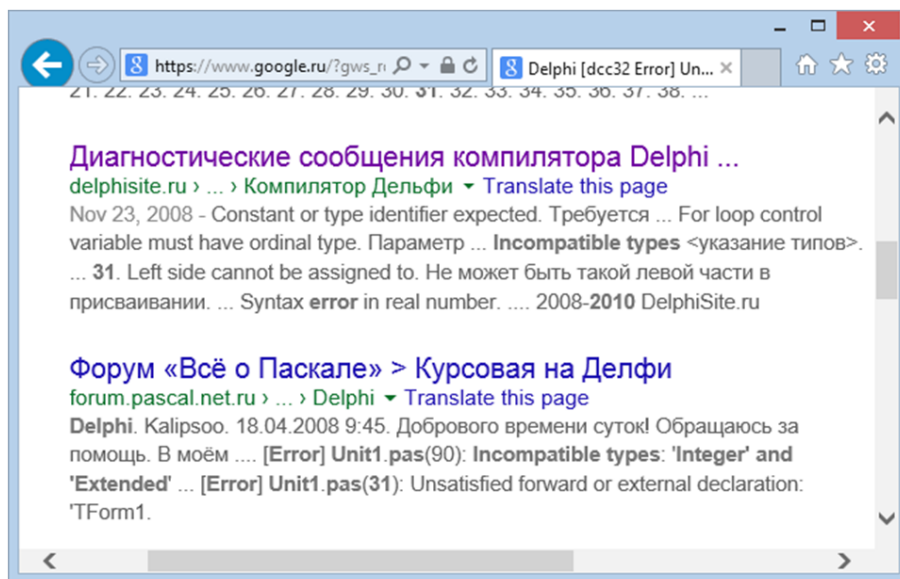


Рис. 9.9. Результат поиска

Раз уже речь зашла об официальной документации от Embarcadero, то обсудим данные проверенные источники. Только что мы уже упомянули технический канал Embarcadero на YouTube, но там опубликовано слишком много видео-роликов, причём большинство из них рассчитаны на профессиональных разработчиков и проектировщиков баз данных. Действительно, Delphi/C++Builder/RAD Studio представляют собой профессиональные инструменты, поэтому и информация о них соответствующего уровня. Специально для начинающих отличным началом будет изучение «приветственной странички» (welcome page) среды разработки. Если запустить Delphi/C++Builder/RAD Studio (при наличии подключения ПК к сети интернет), то сразу в IDE можно посмотреть подборку видео-материалов для начинающих. Если прокрутить стартовую страничку вниз, то появится раздел Code Snippets («фрагменты кода»), как показано на рис. 9.10. Так компания Embarcadero назвала небольшие проекты, показывающие отдельные возможности инструмента разработки Delphi/C++Builder/RAD Studio. Мы уже обсуждали ранее, что настоящее серьезное приложение собирается из различных функциональных блоков, и такими блоками являются Code Snippets.

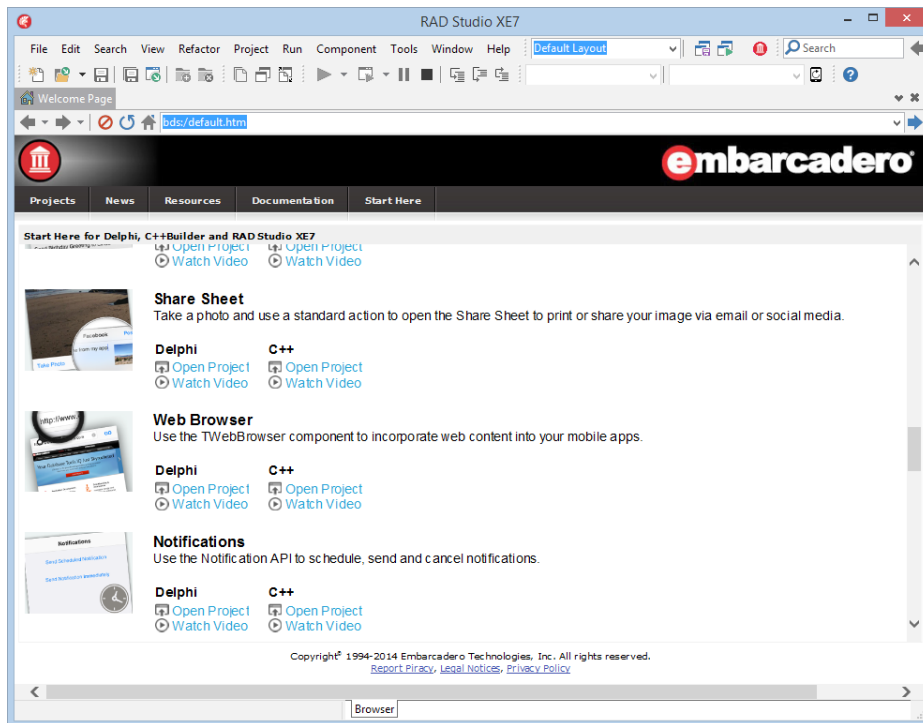


Рис. 9.10. Code Snippets на приветственной странице IDE

В разделе «фрагменты кода» на стартовой страничке присутствуют различные мини-проекты, там можно найти и то, что мы уже прошли: Notifications, Share Sheet, но есть и новые. К таким новым полезным возможностям относится использования компонента для определения позиции мобильного устройства Location, встраивания web-содержимого в приложение и т.д. Внизу страницы есть ссылка More Code Snippets, которая выдаст полный список. Также данные проекты с исходным кодом можно открыть в IDE обычным способом: File->Open Project..., если вы нашли соответствующую папку на жёстком диске после установки продукта Delphi/C++Builder/RAD Studio. Также можно прочитать про «фрагменты кода» и на русском языке по ссылке: <https://www.embarcadero.com/ru/products/rad-studio/android-ios-code-samples-xe8>

Справочная онлайн-система по продуктам Embarcadero находится по адресу: <http://docwiki.embarcadero.com/>, где, выбрав Delphi XE8, мы переходим в полный каталог официальной документации. Можно воспользоваться поисковым сервисом, введя запрос в окошко сверху слева. Но можно целевым образом выбрать нужный раздел. Например, выбрав Code Examples мы попадём на подборку всех доступных онлайн примеров кода. Так легко найти пример, где показаны все элементы управления платформы FMX: http://docwiki.embarcadero.com/CodeExamples/XE8/en/FMX.ControlsDemo_Sample, а также путь к папке с исходным кодом проекта на жёстком диске вашего ПК.

При возникновении сложных вопросов, можно задавать их на форумах разработчиков, например, <http://www.sql.ru/forum/delphi>. Нужно помнить, что профессиональные форумы — это сообщества для обмена опытом разработчиков высокого уровня. Обязательно указывайте в вопросе, что вы — начинающий программист и соблюдайте правила вежливости максимальным образом. Не стоит писать вопрос сразу на форум, если вы предварительно не «погуглили» и не почитали материал справочной системы.

Хороший сайт для фанатов платформы FMX, в котором допускаются вопросы от новичков: <http://fire-monkey.ru/>.

Большинство публикаций о Delphi на русском языке агрегируются сайтом <http://www.delphifeeds.ru/>, возьмите себе за правило регулярно его просматривать.

Пользуясь случаем, отправляю благодарных читателей на блог автора: <http://blogs.embarcadero.com/vsevolodleonov/>, где сразу после выхода данной книги будут публиковаться видео-уроки по изложенному материалу. Там же я жду вопросов и комментариев, а также пожеланий по дальнейшей публикации материалов.

ОБУЧЕНИЕ МОБИЛЬНОЙ РАЗРАБОТКЕ НА DELPHI

ВСЕВОЛОД ЛЕОНОВ



Во времена, когда все меняется быстрее, чем мы успеваем это заметить, лучший выход – адаптация учебной программы к реалиям жизни. Именно поэтому мейнстримом стало обучение школьников и студентов на реальных проектах. Книга именно об этом.

Страница за страницей бывший преподаватель университета им. Баумана и евангелист Embarcaadero раскрывает секреты обучения мобильной разработке на Delphi. Мне понравилось, что он на примерах рассказывает как преподавать мобильную разработку как начинающим, так и уже знакомым с языком Pascal. Самое интересное, что автор берет примеры приложений из разных областей, даже из литературы. Он, например, приводит пример создания приложения по изучению поэзии, захватывает биологию, физику и химию во время создания с нуля приложения по наблюдению за природными явлениями.

Насыров Наиль, преподаватель Санкт-Петербургского национального исследовательского университета информационных технологий механики и оптики (ИТМО).



Осваивая навыки программирования, участвуя в IT-проектах, посещая офисы IT-гигантов, школьники и студенты не только расширяют свой кругозор, но и формируют навыки прикладного применения информационных технологий. Уверен, что это издание поможет учащимся освоить среду программирования Delphi. Полученные знания пригодятся им во взрослой жизни, помогут им стать настоящими профессионалами, востребованными на рынке труда.

Марчак Игорь Степанович, руководитель проектного офиса "Школа новых технологий", совместного проекта Департамента образования и Департамента информационных технологий города Москвы.