# Тема№12. Разработка текста программ

Цель: рассмотреть основы визуального программирования. Повторное использование текста. Событийное программирование. Интегрированные визуальные среды разработки ([Delphi](http://ru.wikipedia.org/wiki/Delphi), Visual Basic и [C++ Builder](http://ru.wikipedia.org/wiki/C%2B%2B_Builder)). Автоматизация разработки – мастера (Wizards), библиотеки разработки, построение собственных компонент и их оформление с использованием инструментальных средств, инспектор свойств и событий, создание обработчиков событийных прерываний, установку и редактирование свойств.

План:

# **Визуальное программирование.**

# **Повторное использование - текста.**

# **Событийное программирование.**

# **Интегрированные визуальные среды разработки (**[**Delphi**](http://ru.wikipedia.org/wiki/Delphi)**, Visual Basic и** [**C++ Builder**](http://ru.wikipedia.org/wiki/C%2B%2B_Builder)**).**

# **Автоматизация разработки – мастера (Wizards)**

1. Библиотеки разработки - готовые к повторному использованию тексты программ (компоненты).
2. Инспектор свойств и событий.

**1 Визуальное программирование**

**Визуальное программирование** — способ создания программы для компьютера путём манипулирования графическими объектами вместо написания её текста.

Необходимо различать:

* [графический язык программирования](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) – который, прежде всего язык программирования (со своим синтаксисом);
* [визуальные средства разработки](http://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%B7%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B2%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8&action=edit&redlink=1) - как правило, под ними подразумевают средства проектирования интерфейсов или какую-либо [CASE](http://ru.wikipedia.org/w/index.php?title=CASE&action=edit&redlink=1)-систему, для быстрой разработки приложений или [SCADA](http://ru.wikipedia.org/wiki/SCADA)- систему для программирования микроконтроллеров.

Языки визуального программирования могут быть дополнительно классифицированы в зависимости от типа и степени визуального выражения, на следующие типы:

* языки на основе объектов, когда визуальная среда программирования предоставляет графические или символьные элементы, которыми можно манипулировать интерактивным образом в соответствии с некоторыми правилами;
* языки, в [интегрированной среде разработки](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) которых на этапе проектирования интерфейса применяются формы, с возможностью настройкой их свойств. Примеры: [Delphi](http://ru.wikipedia.org/wiki/Delphi) и [C++ Builder](http://ru.wikipedia.org/wiki/C%2B%2B_Builder) фирмы [Borland](http://ru.wikipedia.org/wiki/Borland), [С#](http://ru.wikipedia.org/wiki/C_Sharp)
* языки схем, основанные на идее «фигур и линий», где фигуры (прямоугольники, овалы и т. п.) рассматриваются как субъекты и соединяются линиями (стрелками, дугами и др.), которые представляют собой отношения. Пример: [UML](http://ru.wikipedia.org/wiki/UML).

Визуально-преобразованные языки являются невизуальными языками с наложенным визуальным представлением (например, среда [Visual C++](http://ru.wikipedia.org/wiki/Visual_C%2B%2B) для языка [C++](http://ru.wikipedia.org/wiki/C%2B%2B)). Естественно-визуальные языки имеют неотъемлемое визуальное выражение, для которого нет очевидного текстового эквивалента (например, графический язык G в среде [LabVIEW](http://ru.wikipedia.org/wiki/LabVIEW)).

В современных разработках делаются попытки интегрировать подход визуального программирования с программированием потоков данных (англ. dataflow programming), чтобы иметь непосредственный доступ к состоянию программы для онлайновой отладки, или автоматизированная генерация и документирование программы. Языки потоков данных также позволяют автоматическое распараллеливание, которое может стать одним из величайших достижений программирования в будущем.

## Графические (Визуальные) языки программирования

Это незаконченный список, который может быть никогда не будет удовлетворять каким-либо стандартам по своей завершенности. Вы можете дополнить его, ссылаясь на источники.

* [Дракон-схемы](http://ru.wikipedia.org/w/index.php?title=%D0%94%D0%A0%D0%90%D0%9A%D0%9E%D0%9D_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)&action=edit&redlink=1) - графический язык программирования, используется для программирования в ракетно космической технике ("[Буран](http://ru.wikipedia.org/wiki/%D0%91%D1%83%D1%80%D0%B0%D0%BD_(%D0%BA%D0%BE%D1%81%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D1%80%D0%B0%D0%B1%D0%BB%D1%8C))", "[Морской старт](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%80%D1%81%D0%BA%D0%BE%D0%B9_%D1%81%D1%82%D0%B0%D1%80%D1%82)", "[Тополь](http://ru.wikipedia.org/wiki/%D0%A0%D0%A2-2%D0%9F%D0%9C)"). Существует бесплатный Дракон-редактор. Этот язык имеет наиболее строгое теоретическое обоснование.
* Язык последовательных функциональных схем [SFC](http://ru.wikipedia.org/wiki/SFC) (Sequential Function Chart) – данный [графический язык программирования](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) широко используется для программирования промышленных логических контроллеров [PLC](http://ru.wikipedia.org/wiki/PLC). В SFC программа описывается в виде схематической последовательности шагов, объединенных переходами.
* [LD](http://ru.wikipedia.org/wiki/LD) — язык релейно-контактных схем.
* [FBD](http://ru.wikipedia.org/wiki/FBD) — язык Функциональных блоковых диаграмм.
* Язык [CFC](http://ru.wikipedia.org/w/index.php?title=CFC&action=edit&redlink=1) ([Continuous Flow Chart](http://ru.wikipedia.org/w/index.php?title=Continuous_Flow_Chart&action=edit&redlink=1" \o "Continuous Flow Chart (страница отсутствует))) – еще один высокоуровневый язык графического программирования. CFC – это дальнейшее развития языка FBD. CFC был специально создан для проектирования систем управления непрерывными технологическими процессами.
* [G](http://ru.wikipedia.org/w/index.php?title=G_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)&action=edit&redlink=1), язык, используемый в среде разработки [LabVIEW](http://ru.wikipedia.org/wiki/LabVIEW)

## Визуальные средства разработки

1. [AgentSheets](http://ru.wikipedia.org/w/index.php?title=AgentSheets&action=edit&redlink=1), легкая в использовании авторская система для игр и научных расчётов
2. [Alice](http://ru.wikipedia.org/wiki/Alice)
3. [Analytica](http://ru.wikipedia.org/w/index.php?title=Analytica&action=edit&redlink=1)
4. [AppWare](http://ru.wikipedia.org/w/index.php?title=AppWare&action=edit&redlink=1), also known as MicroBrew, программирование, на основе иконок для Mac OS и Microsoft Windows
5. [AudioMulch](http://ru.wikipedia.org/w/index.php?title=AudioMulch&action=edit&redlink=1), основанная на потоке звукового сигнала, среда для создания музыки
6. [Macromedia Authorware](http://ru.wikipedia.org/w/index.php?title=Macromedia_Authorware&action=edit&redlink=1)
7. [Automator](http://ru.wikipedia.org/wiki/Automator)
8. [Aviary Peacock](http://ru.wikipedia.org/w/index.php?title=Aviary_Peacock&action=edit&redlink=1), визуальная лаборатория на основе просмотра файлов
9. [Baltie](http://ru.wikipedia.org/w/index.php?title=Baltie&action=edit&redlink=1)
10. [Befunge](http://ru.wikipedia.org/wiki/Befunge), изотерический текстовый язык программирования, в котором команды размещаются графически в текстовом файле
11. [CODE](http://ru.wikipedia.org/w/index.php?title=CODE&action=edit&redlink=1)
12. [DRAKON](http://ru.wikipedia.org/w/index.php?title=DRAKON&action=edit&redlink=1), язык, созданный для разработки Советского Бурана (космического корабля)
13. [eXpecco](http://ru.wikipedia.org/w/index.php?title=EXpecco&action=edit&redlink=1), основанный на графическом потоке язык программирования и среда разработки для автоматизации тестирования
14. [Flow](http://ru.wikipedia.org/w/index.php?title=Flow&action=edit&redlink=1)
15. [Kwikpoint](http://ru.wikipedia.org/w/index.php?title=Kwikpoint&action=edit&redlink=1), пиктограммный визуальный транслятор, созданный Аланом Стилманом
16. [LabVIEW](http://ru.wikipedia.org/wiki/LabVIEW), среда графического программирования, разработанная для инженеров и учёных
17. [HiAsm](http://ru.wikipedia.org/wiki/HiAsm), конструктор программ

**2 Событийно-ориентированное программирование**

Событи́йно-ориенти́рованное программи́рование (англ. event-driven programming; в дальнейшем СОП) — это способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события.

**Сфера применения**

Событийно-ориентированное программирование, как правило, применяется в трех случаях:

- при построении пользовательских интерфейсов (в том числе графических);

- при создании серверных приложений в случае, если по тем или иным причинам нежелательно порождение обслуживающих процессов;

- при программировании игр, в которых осуществляется управление множеством объектов.

**Применение в серверных приложениях**

Событийно-ориентированное программирование применяется в серверных приложениях для решения проблемы масштабирования на 10000 одновременных соединений и более.

В серверах, построенных по модели «один поток на соединение», проблемы с масштабируемостью возникают по следующим причинам:

- слишком велики накладные расходы на структуры данных операционной системы, необходимые для описания одной задачи (сегмент состояния задачи, стек);

- слишком велики накладные расходы на переключение контекстов.

Философской предпосылкой для отказа от потоковой модели серверов может служить высказывание Алана Кокса: «Компьютер — это конечный автомат. Потоковое программирование нужно тем, кто не умеет программировать конечные автоматы».

Серверное приложение при событийно-ориентированном программировании реализуется на системном вызове, получающем события одновременно от многих дескрипторов (мультиплексирование). При обработке событий используются исключительно неблокирующие операции ввода-вывода, чтобы ни один дескриптор не препятствовал обработке событий от других дескрипторов.

**Мультиплексирование**

Для мультиплексирования соединений могут быть использованы следующие средства операционной системы:

- select (большинство UNIX систем). Плохо масштабируется, из-за того, что список дескрипторов представлен в виде битовой карты;

- poll и epoll (Linux);

- kqueue (FreeBSD);

- /dev/poll (Solaris);

- IO completion port (Windows);

- POSIX AIO на текущий момент только для операций дискового ввода-вывода;

- io submit и eventfd для операций дискового ввода-вывода.

**Применеие в интерфейсах**

СОП может быть как расширением структурного, так и объектно-ориентированного программирования (в дальнейшем ООП). Эпостасиями СОП являются cобытия и их приоритеты. События — специльные процедуры (для структурного програмирования) или методы (для ООП). События можно разделить на обычные, выполнение которых начинается по их прямому вызову (пользователя или программы) и события-демоны, находящихся в фоновом режиме и выпняющихся при совершении события, например, выполнения условий. При наступления ситуации, когда событие должно начать работу оно уступает событиям, имеющий больший приоритет. Приоритет может быть выражден от динамического до статического, крайним случаем которого является порядок следования событий.

## 3 Повторное использование кода

## **Повторное использование кода (англ. code reuse) — методология проектирования компьютерных и других систем, заключающаяся в том, что система (компьютерная программа, программный модуль) частично либо полностью должна составляться из частей, написанных ранее компонентов и/или частей другой системы**

## **Повторное использование — основная методология, которая применяется для сокращения трудозатрат при разработке сложных систем.**

## **Самый распространённый случай повторного использования кода — библиотеки программ. Библиотеки предоставляют общую достаточно универсальную функциональность, покрывающую избранную предметную область.**

## **Примеры: библиотека функций для работы с комплексными числами, библиотека функций для работы с 3D-графикой, библиотека для использования протокола TCP/IP, библиотека для работы с базами данных.**

## **Иногда повторное использование кода представляет собой простое копирование некоторой части кода из существующей программы в другую (англ. copy-paste). Это один из самых низкоуровневых подходов к повторному использованию. Но и он имеет место, особенно когда речь идет о повторном использовании кода «в малом» («reuse в малом»).**

## **Кроме немногочисленных, но очень важных достоинств метод повторного использования кода имеет ряд недостатков. Подключение к проекту сторонних библиотек автоматически приводит к необходимости контроля совместимости версий компонент создаваемой системы и версий используемых библиотек. Самым характерным примером такой ошибки считается Авария ракеты-носителя Ариан 5 (4 июня 1996), вызванная использованием программного модуля, разработанного для ракеты Ариан - 4. Важно также отметить, что многие библиотеки коммерческие и требуют денежных затрат (с развитием движения свободного ПО это постепенно теряет актуальность). Кроме того, часто библиотеки недостаточно универсальны и не реализуют той функциональности, которая требуется создаваемой системе, либо, наоборот, слишком универсальны и в результате неэффективны, неудобны или содержат много избыточной (для данного проекта) функциональности**

**4 Инструментальные средства разработки программ**

Инструментальное программное обеспечение (Software tools) — программное обеспечение, используемое в ходе разработки, корректировки или развития других программ: редакторы, компиляторы, отладчики, вспомогательные системные программы, графические пакеты и др. Сюда входят языки программирования, интегрированные среды разработки программ, CASE-системы и др.

**Выбор языка программирования**

Существующие на сегодняшний день языки программирования можно выделить в следующие группы:

* универсальные языки высокого уровня;
* специализированные языки разработчика программного обеспечения;
* специализированные языки пользователя;
* языки низкого уровня.

В группе универсальных языков высокого уровня безусловным лидером на сегодня является язык C++. Действительно, он имеет ряд достоинств:

* масштабируемость. На языке C++ разрабатывают программы для самых различных платформ и систем;
* возможность работы на низком уровне с памятью, адресами, портами, что при неосторожном использовании может легко превратиться в недостаток;
* C++ имеет мощный препроцессор, унаследованный от С, но, как и любой другой мощный инструмент, требует осторожного использования;
* возможность создания обобщенных алгоритмов для разных типов данных, их специализация и вычисления на этапе компиляции, используя шаблоны.

При этом язык C++ обладает рядом существенных недостатков:

* подключение интерфейса внешнего модуля через препроцессорную вставку заголовочного файла (#include) серьезно замедляет компиляцию при подключении большого количества модулей;
* недостаток информации о типах данных во время компиляции;
* сложность для изучения и компиляции;
* некоторые преобразования типов неинтуитивны. В частности, операция над беззнаковым и знаковым числами выдает беззнаковый результат.

Для C++ существует большое количество библиотек классов, поддерживающих создание пользовательского интерфейса, клиент-серверных приложений, работу с базами данных и т. д., поэтому пока альтернативы C++ нет. Для второстепенных проектов иногда используется Visual Basic. Язык Java рассматривался как альтернатива Basic, но из-за отсутствия визуального средства разработки форм он пока остается малопригодным. Современный Object Pascal, как и Pascal, предложенный Н. Виртом в середине 70-х годов XX в., остается наиболее привлекательным для обучения основам программирования в силу своей простоты, структурированности и обнаружения компилятором большого количества не только синтаксических, но и семантических ошибок.

В нынешнее время в отличие от 60-х годов XX в. языки программирования создаются крайне редко. За последние 15 лет можно отметить лишь две новинки, получившие широкое распространение — это Java (Sun Microsystems, 1995 г.), ставший популярным во многом благодаря технологии его использования в Интернете и появления такого понятия, как виртуальная Java-машина, и С# (Microsoft, 2000 г.), созданный на основе C++.

Создателем языка является сотрудник Microsoft Андреас Хейлсберг. Он стал известным в мире программистов задолго до того, как пришел в Microsoft. Хейлсберг входил в число ведущих разработчиков одной из самых популярных сред разработки — Delphi. В Microsoft он участвовал в создании версии Java — J++, так что опыта в написании языков и сред программирования ему не занимать. Как отмечал сам Андреас Хейлсберг, С# создавался как язык компонентного программирования, и в этом одно из главных достоинств языка, направленное на возможность повторного использования созданных компонентов.

Другие достоинства языка С#:

* сохраняет лучшие черты популярных языков программирования C/C++, на основе которых он создан. В связи с этим облегчается переход программистов от C++ к С#;
* является проще и надежнее C++. Простота и надежность главным образом связаны с тем, что на С# хотя и допускаются, но не поощряются такие опасные свойства C++, как указатели, адресация, разыменование, адресная арифметика;
* является полностью объектно-ориентированным языком, где даже типы, встроенные в язык, представлены классами;
* реализует возможности наследования и универсализации;
* учитывает все возможности Framework .Net, так как С# создавался параллельно с данной средой;
* благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты С# получают те же преимущества работы с виртуальной машиной, что и программисты Java. Эффективность кода даже повышается, поскольку исполнительная среда CLR представляет собой компилятор промежуточного языка, в то время как виртуальная Java-машина является интерпретатором байт-кода;
* мощная библиотека каркаса поддерживает удобство построения различных типов приложений на С#, позволяя легко строить Web-службы, другие виды компонентов, достаточно просто сохранять и получать информацию из базы данных и других хранилищ данных;
* является источником надежного и эффективного кода.

Кроме вышеописанных языков к группе универсальных принадлежат также Modula, Ada, COBOL, FORTRAN и некоторые другие. Каждый из вышеописанных языков имеет свои особенности и, соответственно, свою область применения. В настоящее время универсальные языки программирования применяются в самых различных областях человеческой деятельности, таких как:

* научные вычисления (языки C++, FORTRAN, Java);
* системное программирование (языки C++, Java);
* обработка информации (языки C++, COBOL, Java);
* искусственный интеллект (LISP, Prolog);
* издательская деятельность (Postscript, TeX);
* удаленная обработка информации (Perl, PHP, Java, C++);
* описание документов (HTML, XML).

С течением времени одни языки развивались, приобретали новые черты и остались востребованными, другие утратили свою актуальность и сегодня представляют в лучшем случае чисто теоретический интерес (Focal, PL/1 и др.). В значительной степени это связано с такими факторами:

* наличие среды программирования, поддерживающей разработку приложений на конкретном языке программирования;
* удобство сопровождения и тестирования программ;
* стоимость разработки с применением конкретного языка программирования;
* четкость и ортогональность конструкций языка;
* применение объектно-ориентированного подхода.

Специализированные языки разработчика используют для создания конкретных типов программного обеспечения. К ним относят:

* языки баз данных;
* языки создания сетевых приложений;
* языки создания систем искусственного интеллекта и т. д.

Специализированные языки пользователя обычно являются частью профессиональных сред пользователя, характеризуются узкой направленностью и разработчиками программного обеспечения не используются.

Языки низкого уровня позволяют осуществлять программирование практически на уровне машинных команд. При этом получают самые оптимальные как с точки зрения времени выполнения, так и с точки зрения объема необходимой памяти программы. Недостатком их является то, что они не поддерживают принципов структурного программирования.

В настоящее время языки типа ассемблера обычно используют:

* при написании сравнительно простых программ, для обращения к техническим средствам, например драйверов;
* в виде вставок в программы на языках высокого уровня, например, для ускорения преобразования данных в циклах с большим количеством повторений.

В большей степени выбор языка программирования определяется опытом разработчика, требованиями ведущей разработку организации или просто устоявшимся мнением.

**Выбор среды программирования**

Интегрированной средой разработки программного обеспечения называют систему программных средств, используемую программистами для разработки программного обеспечения. Обычно среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, компоновщик, отладчик и справочную систему. Иногда также содержит систему управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя.

Многие современные среды разработки также включают инспектор объектов, браузер классов и диаграмму иерархии классов, которые используются для объектно-ориентированной разработки ПО. Обычно среда разработки предназначается для одного определенного языка программирования, как, например, Visual Basic или Deiphi, но существуют среды разработки, предназначенные для нескольких языков, такие как Eclipse или Microsoft Visual Studio.

Примеры сред разработки — Turbo Pascal, Borland C++, GNU toolchain, DrPython.

В последнее время, с развитием объектно-ориентированного программирования, широкое распространение получили упоминавшиеся ранее среды визуального программирования, в которых наиболее распространенные блоки программного кода представлены в виде графических объектов.

Наиболее часто используемыми являются визуальные среды Delphi, C++ Builder фирмы Borland (Inprise Corporation), Visual C++, Visual Basic фирмы Microsoft, Visual Ada фирмы IBM и др.

Большую популярность в наши дни получила технология .NET Framework, предложенная фирмой Microsoft в качестве платформы для создания как обычных программ, так и веб-приложений. Основным преимуществом .NET является совместимость различных служб, написанных на разных языках. Например, служба, написанная на C++ для .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на С# можно написать класс, наследующий от класса, написанного на Visual Basic .NET, а исключение, выброшенное методом, написанным на С#, может быть поймано и обработано в Delphi. Так же как и в случае с выбором языка программирования, выбор среды программирования определяется характером проекта, привычками и навыками разработчика, веяниями времени, требованиями заказчика и просто общественным мнением.

Интегрированная среда разработки представляет собой полнофункциональный набор инструментов для осуществления полного цикла разработки программ (написание программ, компиляция и визуальная отладка).

Среда может находиться в двух состояниях**.**

Рисунок – 47. Перспектива разработки и компиляции проектов C, С++ или ассемблера

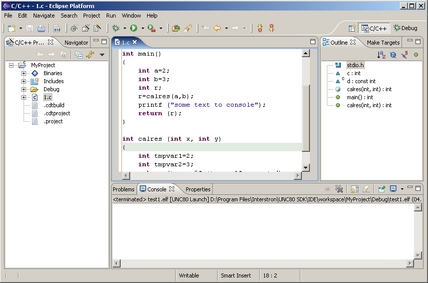
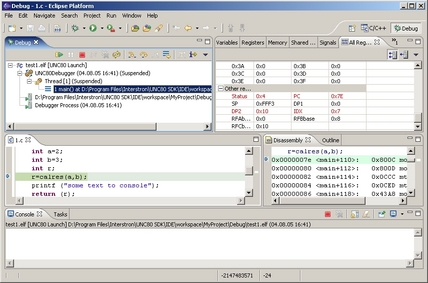


Рисунок 48. Перспектива отладки проектов



Каждая перспектива представляет собой набор окошек, несущих ту или иную функциональность. Расположение и состав этих окошек может быть легко изменён по желанию пользователя среды.

Перспектива разработки и компиляции проектов предназначена для создания, изменения и компиляции проектов. В ней могут содержаться окно структуры проектов, отображающее все имеющиеся в наличии проекты, полнофункциональный редактор исходных файлов с настраиваемой подсветкой текста, консоль работы компилятора, окно демонстрации ошибок и предупреждений работы компилятора (а также ассемблера и линкера) с привязкой к редактору исходных текстов. Имеется также окно, отображающее структуру исходного файла С или С++, открытого в редакторе.

Создание проектов производится с помощью специального менеджера, который последовательно, шаг за шагом запрашивает информацию о новом проекте. Здесь можно установить пути к включаемым файлам, пути к библиотекам, установить опции компилятора, ассемблера и линкера.

**5 Автоматизация разработки Master Wizard**

Мастера – это общий класс инструментов Windows, упрощающих выполнение часто повторяющихся задач, в том числе в пошаговом режиме взаимодействия с пользователем.

Master Wizard – это средство для автоматизации разработки и поддержки сайта.

Master Wizard предназначен для создания Wizards в среде Visual Basic. Можно сказать что, Master Wizard это несколько последовательных шагов, с помощью которых в удобном для пользователя виде собирается информация, на основании которой производятся определенные действия – создаются пользователи, импортируются данные, настраиваются параметры.

Master Wizard используется в среде Project Guide-2000. MW даёт новый и значительно более простой программный интерфейс, чем интерфейс VB IDE. Это значит, что появилась возможность значительно более простыми средствами и с минимальными затратами времени создавать для себя среду разработки для VB6. Теперь у разработчика появилась возможность усовершенствовать свою среду разработки не начиная новый Add-Ins проект, а просто написать скрипт для MW и вставить его в меню MW. Вам не надо знать объекты VB IDE для работы с проектом. Вы просто используете функции MW, которые скрывают всю сложность объектов VB IDE для вас. Кроме того вы всегда можете усовершенствовать скрипт, написанный вами или другими разработчиками ранее. Используя возможности MW можно быстро создавать различные Wizards и использовать их в VB6. Это достигается за счёт простых средств создания диалога, небольшого количества мощных функций, работающих с объектами VB IDE, простотой языка MW(VB-Script) и удобством используемых полученных скриптов.

Вызовы Скриптов вставляются в оглавление древовидной структуры и легко находятся для использования. Диапазон использования MW лежит от простой установки множества свойств управляющих элементов (что позволяет переходить от установки свойств к установке стиля) для создания полноценных Wizards, создающих Forms, Controls, CodeModules или модифицирующих ваши Templates.

MW является интерпретатором скриптов, аналогично ASP скриптам, но результатом интерпретации является не HTML страница, а объекты среды разработки VB IDE.

Скрипт MW может состоять из трёх видов скриптов:

**- скрипт** **Menu.** Используется для создания шаблонов меню. Используя затем диалог Paste можно вставить меню на форму.

**- скрипт для создания диалога.** Скрипт заключается в скобки FORM{…} Скриптов FORM{} может быть несколько. С помощью них организуется выполнение серий диалоговых окон (Диалогов) для сбора информации, которая является входной для выполнения скрипта CODE{…}. Скрипты типа FORM{} не являются обязательными. Скрипты типа FORM{…} могут быть трех типов:

- неименованный скрипт. Скрипт не содержит свойства Name и исполняется всегда при запуске скрипта MW. Его нельзя повторно вызывать из кода и порядок его вызова определяется порядком написания в скрипте МW.

-именованный скрипт. Содержит свойство Name. Исполнение данного скрипта всегда происходит программно с помощью функций Show. При этом всегда можно управлять свойствами контролов диалога с помощью функции Modify.

-динамический скрипт. Функции Show передается не имя формы диалога, а строка описания формы диалога.

-**скрипт CODE{…}** для получения конкретных результатов на основе скрипта диалога FORM{…} и/или шаблона VB IDE. Скрипт содержит последовательность операторов VB-Script, позволяющих реализовать логику вашего Wizard. Скрипт может вызывать именованные или динамические FORM-скрипты для получения входной информации для вашего Wizard. С помощью мощного набора функции вы можете создавать и модифицировать объекты среды разработки, такие как Controls, Forms, User Controls, Code Modules и др.

**Элемент управления Wizard(Мастер). Серверный элемент управления Wizard в ASP.NET2.0, функциональность работы с несколькими формами**

Очень часто всю информацию, которую нужно принять от пользователя, сложно уместить на одной Web- форме (странице HTML). В этой ситуации обычно:

* Создаётся несколько Web- форм, на каждой из которых настраиваются свои элементы управления;
* Обеспечиваются средства управления состоянием и сохранения значений, которые пользователь ввёл на каждой форме;
* Создаются элементы управления, которые обеспечивают переход между формами и т.п.

Применение элемента управления Wizard (Мастер) позволяет обеспечить функциональность при работе с несколькими формами, которую в противном случае пришлось бы реализовать вручную. При использовании этого элемента управления на странице автоматически создаётся набор вкладок (по умолчанию-Step1, Step2 и т.п.), при помощи которых производится переход между формами. В нижней части автоматически создаётся набор кнопок для навигации(Next, Previous и т.п.), а на последней вкладке-кнопка Finish, при нажатии на которую должны выполняться какие-либо действия с информацией, введённой пользователем. Справа от вкладок и сверху от кнопок находится область, в которой вы можете помещаться обычные элементы управления.

К каждому этапу вы можете применить своё оформление. В вашем распоряжении также полные возможности по настройке этого элемента управления. Например, вы можете переименовывать шаги мастера, кнопки и т.п.

### **6 Библиотеки разработки - готовые к повторному использованию тексты программ (компоненты)**

### **Библиотека DLL (dynamic-link library)** представляет собой коллекцию подпрограмм, которые могут быть вызваны на выполнение приложениями или подпрограммами из других библиотек. Подобно модулям, библиотеки DLL содержат разделяемый (sharable) код или ресурсы. Однако, в отличие от модулей, библиотеки содержат отдельно откомпилированный исполняемый код, который подключается к приложению динамически на этапе его выполнения, а не компиляции.

### Для того чтобы библиотеки можно было отличить от самостоятельно выполняемых приложений, они имеют расширение .dll.

### С самого рождения (или чуть позже) операционная система Windows использовала библиотеки динамической компоновки DLL (Dynamic Link Library), в которых содержались реализации наиболее часто применяемых функций.

### Практически невозможно создать приложение Windows, в котором не использовались бы библиотеки DLL. В DLL содержатся все функции Win32 API и несчетное количество других функций операционных систем Win32.

### Вообще говоря, DLL - это просто наборы функций, собранные в библиотеки. Однако, в отличие от своих статических родственников (файлов . lib), библиотеки DLL не присоединены непосредственно к выполняемым файлам с помощью редактора связей. В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также контролировать размеры ЕХЕ-файлов.

### **Предпосылки к появлению DLL**

Свою историю DLL ведут с средины 60-х годов прошлого столетия, однако по-настоящему широкое распространение динамически линкуемые библиотеки получили после появления операционной системы Windows.

По крайней мере, доподлинно известно, что операционные системы Windows 3.1/3.11 уже содержали программы, использующие DLL.

В связи с тем, что в те времена емкости оперативной памяти и жесткого диска были значительно меньше, чем сейчас, использование DLL предоставляло ряд преимуществ:

- экономия дискового пространства за счет многократного использования кода (reusing). Если приложения используют один и тот же код, нет необходимости поставлять его в коде каждого приложения. Достаточно разработать DLL.

- экономия физической памяти (RAM) за счет загрузки в нее единственного экземпляра DLL.

- изолирование и модификация кода DLL независимо от остального кода программы. Например, код визуализации изолируется от математической части. При изменении математического аппарата (например, при разработке нового, более быстрого алгоритма) перекомпиляция кода клиентского приложения (отвечающего за визуализацию результатов) не требуется.

**Экспорт и импорт идентификаторов библиотеки**

С формальной точки зрения DLL - особым образом оформленный относительно независимый блок исполняемого кода.

Особый способ оформления предполагает наличие в DLL так называемых секций импорта и экспорта. Секция экспорта указывает те идентификаторы объектов (функций, классов, переменных), доступ к которым предоставляет данная DLL. В этом случае мы говорим об экспортировании идентификаторов из DLL. В общем случае, именно секция экспорта предоставляет особый интерес для разработчиков. Хотя ничто не мешает реализовать DLL, которая не имеет данной секции, но, тем не менее, выполняет полезную работу.

Относительная независимость связана с наличием/отсутствием секции импорта у DLL (т.е. секции, в которой описываются внешние зависимости данной DLL от других). Подавляющее большинство DLL (за исключением, быть может, DLL ресурсов) импортирует функции из системных DLL (kernel32.dll, user32.dll, gdi32.dll и др.). В большинстве случае при создании проекта в его опциях автоматически проставляется стандартный набор таких библиотек.

Исполняемый" код в DLL не предполагает автономного использования. Перед тем, как можно будет приступить к использованию, необходимо загрузить DLL в область памяти вызывающего процесса (т.е. DLL не может выполняться сама по себе - ей обязательно нужен клиент). Это явление носит название "проецирование DLL на адресное пространство процесса". И это не удивительно, если вспомнить тот факт, что процессор работает не только с регистрами, но и с адресами памяти. Поэтому каждому объекту DLL требуется свое место "под солнцем", чтобы иметь возможность быть выполненным при вызове. В конечном коде exe-файла, который генерирует компилятор, не будет инструкций процессора, соответствующих коду данной функции. Вместо этого будет сгенерирована инструкция вызова соответствующей функции (call). Так как DLL отображена на адресное пространство процесса, то код DLL будет легко доступен по call-вызову.

**Вызов функций из DLL**

Существует два способа загрузки DLL: с явной и неявной компоновкой.

При неявной компоновке функции загружаемой DLL добавляются в секцию импорта вызывающего файла. При запуске такого файла загрузчик операционной системы анализирует секцию импорта и подключает все указанные библиотеки. Ввиду своей простоты этот способ пользуется большой популярностью; но простота - простотой, а неявной компоновке присущи определенные недостатки и ограничения:

- все подключенные DLL загружаются всегда, даже если в течение всего сеанса работы программа ни разу не обратится ни к одной из них;

- если хотя бы одна из требуемых DLL отсутствует (или DLL не экспортирует хотя бы одной требуемой функции) - загрузка исполняемого файла прерывается сообщением "Dynamic link library could not be found" (или что-то в этом роде) - даже если отсутствие этой DLL некритично для исполнения программы. Например, текстовой редактор мог бы вполне работать и в минимальной комплектации - без модуля печати, вывода таблиц, графиков, формул и прочих второстепенных компонентов, но если эти DLL загружаются неявной компоновкой - хочешь не хочешь, придется "тянуть" их за собой.

- поиск DLL происходит в следующем порядке: в каталоге, содержащем вызывающий файл; в текущем каталоге процесса; в системном каталоге %Windows%System%; в основном каталоге %Windows%; в каталогах, указанных в переменной PATH. Задать другой путь поиска невозможно (вернее - возможно, но для этого потребуется вносить изменения в системный реестр, и эти изменения окажут влияние на все процессы, исполняющиеся в системе).

Явная компоновка устраняет все эти недостатки - ценой некоторого усложнения кода. Программисту самому придется позаботиться о загрузке DLL и подключении экспортируемых функций (не забывая при этом о контроле над ошибками, иначе в один прекрасный момент дело кончится зависанием системы). Зато явная компоновка позволяет подгружать DLL по мере необходимости и дает программисту возможность самостоятельно обрабатывать ситуации с отсутствием DLL. Можно пойти и дальше - не задавать имя DLL в программе явно, а сканировать такой-то каталог на предмет наличия динамических библиотек и подключать все найденные к приложению. Именно так работает механизм поддержки plug-in’ов в популярном файл-менеджере FAR (да и не только в нем).

Таким образом, неявной компоновкой целесообразно пользоваться лишь для подключения загружаемых в каждом сеансе, жизненно необходимых для работы приложения динамических библиотек; во всех остальных случаях - предпочтительнее явная компоновка.

**Создание библиотеки dll в Delphi**

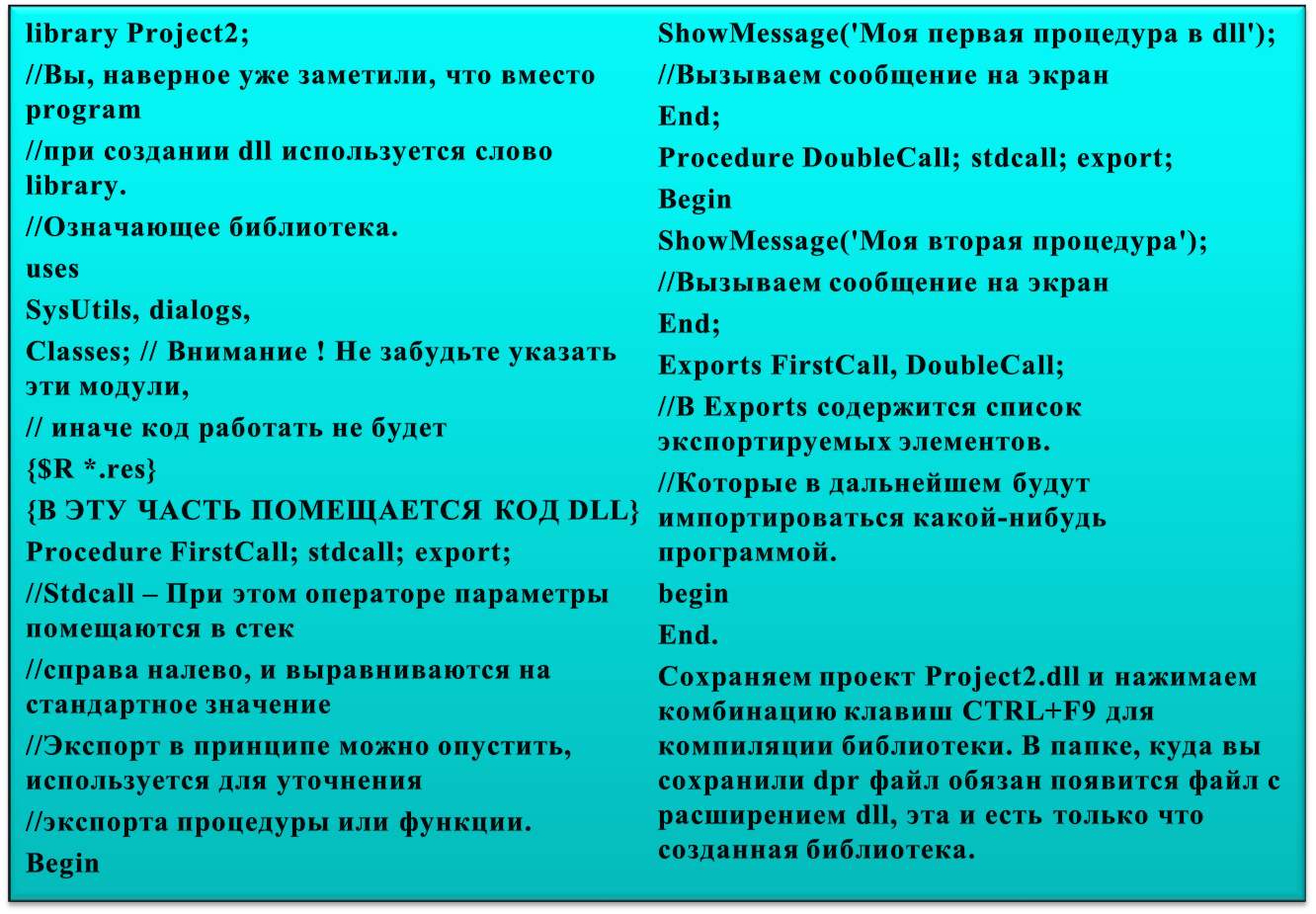
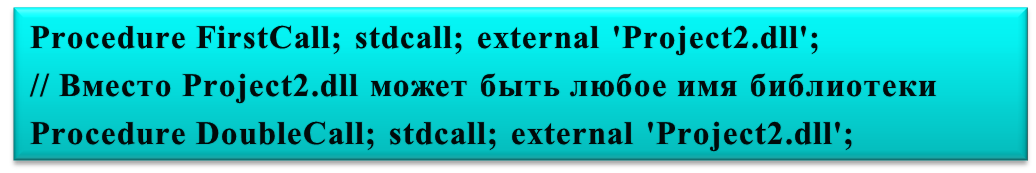


Рисунок – 49. Создание библиотеки dll в Delphi

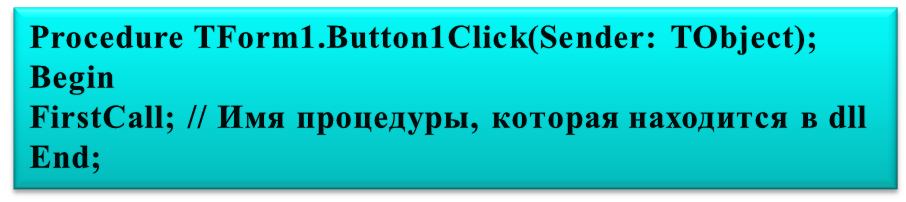
Вызов процедур находящихся в библиотеке. Статическая загрузка (неявная компоновка)

После ключевого слова implementation прописываем следующий код:

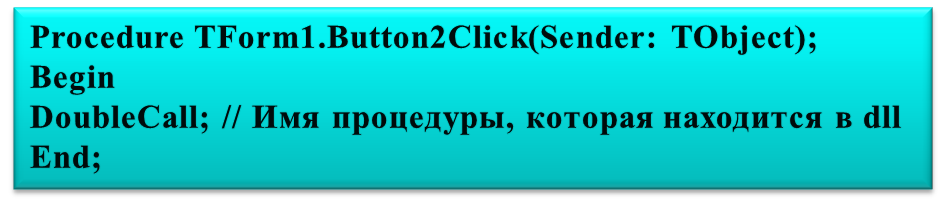


Теперь для того, что бы вызвать данные процедуры нам необходимо лишь вставить их названия в любое место кода.

OnClick первой кнопки:



OnClick второй кнопки:



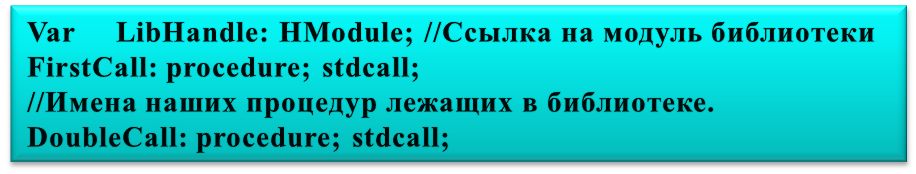
**Динамическая загрузка (явная компоновка)**

Описанный выше способ определения функций и процедур DLL (с помощью директивы External) заставит компилятор поместить в заголовок программы список всех DLL, и загрузчик загрузит библиотеки в память одновременно с загрузкой самой программы. Программа может загружать DLL и без External с помощью трех

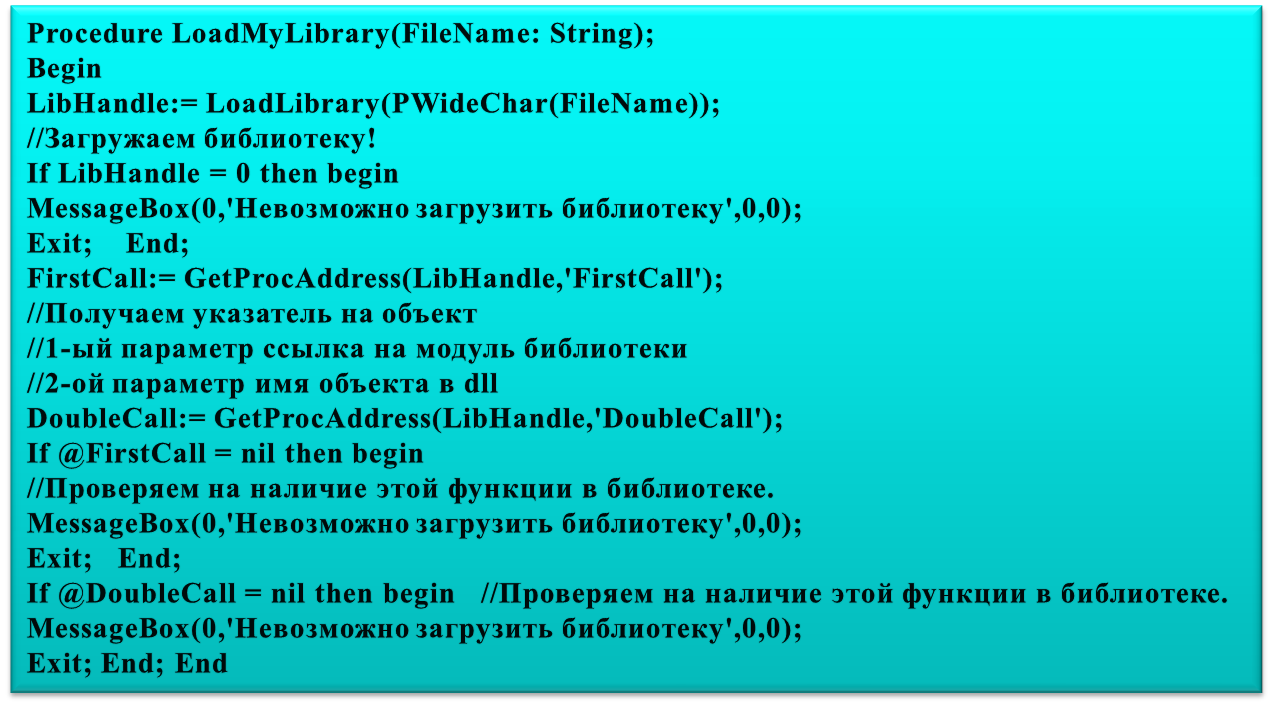
Стандартных функций : **LoadLibrary, GetProcAddress** и **FreeLibrary**.

Следующий пример иллюстрирует технику такой загрузки DLL

Для применения данного метода, первым делом объявляем несколько глобальных переменных:



Затем после ключевого слова implementation напишем процедуру которая будет загружать нашу библиотеку:



### После чего на форме создаем, обработчик событий OnCreate, в котором с помощью только что созданной процедуры мы загрузим нашу библиотеку

### 

### OnClick первой кнопки:

### 

### OnClick второй кнопки:

### 

### Ну и напоследок создаем обработчик событий OnDestroy на форме, в котором выгружаем dll библиотеку из памяти

### 

### **Области применения DLL**

Зачем же нужны библиотеки DLL и где они используются?

Перечислим некоторые из областей их применения:

* **Отдельные библиотеки**, содержащие полезные для программистов дополнительные функции. Например, функции для работы со строками, или же - сложные библиотеки для преобразования изображений.
* **Хранилища ресурсов**. В DLL можно хранить не только программы и функции, но и всевозможные ресурсы - иконки, рисунки, строковые массивы, меню, и т.д.
* **Библиотеки поддержки**. В качестве примера можно привести библиотеки таких известных пакетов, как: *DirectX*, *ICQAPI* (API для ICQ), *OpenGL* и т.д.
* **Части программы**. Например, в DLL можно хранить окна программы (формы), и т.п.
* **Плагины** (Plugins). Плагины - дополнения к программе, расширяющие ее возможности. Например, в этой статье мы рассмотрим теорию создания плагина для собственной программы.
* **Разделяемый ресурс**. DLL (*Dynamic Link Library*) может быть использована сразу несколькими программами или процессами (т.н. *sharing* - разделяемый ресурс)

**Простой пример создания DLL**

Так как DLL – это отдельный файл, то и создаётся он в Delphi как отдельный проект. Для создания новой динамической библиотеки нужно выбрать из меню *File* пункт *New* и затем *Other*. В окне создания нового проекта (рисунок 50) нужно выбрать на закладке *New* пункт *DLL Wizard*. Выберите этот пункт и нажми *ОК*.



Рисунок - 50. Окно создания нового проекта

Несмотря на то, что выбран *DLL Wizard* (слово Wizard говорит о том, что должен запуститься мастер), будет просто создан пустой проект с одним только модулем. Модуль будет содержать следующий текст:

library Project2;

uses

SysUtils,

Classes;

{$R \*.res}

begin

end.

Если открыть менеджер проектов (Меню *View->Project Manager*), то в окне вообще не будет видно ни одного модуля. Это потому что код, который видели выше относиться к самой библиотеки. Выберите из меню *File* пункт *Save All*, и будет предложено сохранить только один проект и никаких модулей. Сохраните проект под именем FirsDLLProject. Потом открыйте файл проекта FirsDLLProject.dpr с помощью блокнота и увидите тот же самый код.

Теперь давайте добавим в нашу библиотеку одну функцию с именем *Summ*. У этой функции будет два параметра в виде целых чисел, и возвращать она будет сумму этих чисел:

library FirsDLLProject;

uses

SysUtils,

Classes;

function Summ(X,Y:Integer):Integer; StdCall;

begin

Result:=X+Y;

end;

exports Summ;

{$R \*.res}

begin

end.

Обратите внимание, что функция у нас объявлена не так как всегда. В конце строки объявления, после типа возвращаемого значения стоит ключевое слово **StdCall**. Оно говорит о том, что для вызова процедуры нужно использовать стандартный тип вызова. Уже говорили, что все параметры, передаваемые в процедуры и в функции, передаются через стек. Если не указать ключевое слово **StdCall**, то параметры будут передаваться способом, заложенным фирмой *Borland*. Этот способ работает быстрее, но он не совместим со стандартными правилами. Если уверены, что к процедуре будут обращаться только программы скомпилированные компиляторами фирмы Borland, то можете не ставить это ключевое слово. Но если библиотека будет выложена на всеобщее использование или к ней будут обращаться программы сторонних разработчиков, то желательно ставить **StdCall**, иначе у программистов на языках Visual C++ или других языках будут проблемы.

Теперь напишем программу, которая будет использовать написанную функцию из динамической библиотеки. Создайте новый проект простого приложения (*File->New- >Application*). На форму поместите только одну кнопку и по её нажатия напиши следующий код:

procedure TForm1.Button1Click(Sender: TObject);

var

418

r:Integer;

begin

r:=Summ(10,34);

Application.MessageBox(PChar(IntToStr(r)), 'Результат функции Summ');

end;

В первой строчке вызываем функцию *Summ* с двумя числовыми параметрами. Результат записывается в переменной *r*. Вторая строка всего лишь выводит окно с результатом. Если попытаетесь сейчас откомпилировать проект, то ничего не выйдет. Компилятор Delphi скажет, что он не знает такой функции *Summ*. Мы должны показать Delphi, что это за функция и где её искать. Для начала покажем компилятору, что это за функция. Для этого в разделе type, после описания объекта *TForm1* (нашей главной формы) нужно написать следующую строку:

function Summ(X,Y:Integer):Integer;StdCall;

В принципе, это такое же объявление функции, которое описано в библиотеке, только здесь нет **begin** и **end** и самого кода процедуры. По этой строке Delphi узнаёт, что где-то существует такая функция *Summ*, у неё есть два параметра, и она должна вызываться стандартным вызовом.

Теперь нужно сказать компилятору, где же искать эту загадочную функцию. Для этого после слова **implementation** напиши следующий код:

function Summ; external 'FirsDLLProject.dll' name 'Summ';

Здесь написано, что есть такая функция *Summ*. После точки с запятой стоит ключевое слово **external**, которое говорит о том, что функция внешняя, не принадлежит программе. После этого слова указывается имя динамической библиотеки, где нужно искать функцию. Далее идёт ключевое слово **name**, которое означает, что функцию надо искать по имени. После этого ключевого слова указывается точное имя функции в библиотеки. Вот теперь проект готов и его можно компилировать, запускать и проверять результат.

Давайте подкорректируем пример на индексы. Откройте проект динамической библиотеки *FirsDLLProject.dpr*. Найдите ключевое слово **export** и напишите код:

exports Summ index 10;

После имени функции стоит ключевое слово **index** и числовой индекс функции. Индексы и имена должны быть уникальными!!! Вот несколько примеров:

exports

Func1 index 10 name 'Fun',

Func2 Insert,

Func3 index 11,

Func4 index 11,//*Ошибка, такой индекс уже существует*

Func5 name 'Don';

В объявлении последней процедуры явно использовано ключевое имя **name**, чтобы указать экспортной функции новое имя. Теперь внутри библиотеки эта функция реализована как *Func5*, но внешние приложения должны обращаться к ней по имени *Don*. Объявлять можно и так:

exports Func1 index 10 name 'Fun',

exports Func2 Insert,

exports Func3 index 11,

Перекомпилируйте проект, чтобы изменения вошли в силу (нажмите Ctrl+F9). Теперь возвращаемся в проект, где используем функцию. В разделе **implementation** подправляем описание нашей функции:

function Summ; external 'FirsDLLProject.dll' index 10;

Теперь вместо ключевого слова **name** стоит слово **index** и тот же номер. Запустите проект и убедитесь, что он работает корректно.

**Замечания по использованию библиотек**

В принципе, библиотеки действительно нельзя запускать, но Delphi может сделать это. Откройте нашу библиотеку и выберите из меню *Run* пункт *Parameters*. Откроется окно (рисунок 51).



Рисунок - 51. Окно параметров запуска программы

В строке *Host Application* нужно указать имя приложения, которое умеет загружать библиотеку. Теперь попробуйте запустить проект (клавиша F9). Запустится указанная программа.

Зачем нужен этот способ? Если попытаетесь запустить программу, и она показала ошибку в коде, где вызывается функция из динамической библиотеки, то можно попытаться запустить библиотеку таким образом. Если снова произойдёт ошибка, то Delphi покажет строку с ошибкой.

Функции и процедуры из динамической библиотеки не могут прямо влиять на ход основной программы. Это значит, что мы не можем получить доступ к окнам основной программы, изменить какие-то переменные или ещё чего-нибудь. Функции библиотеки – как бы изолированы от всего остального, хотя и выполняются в одном адресном пространстве с основной программой. Они могут использовать только переданные параметры, а результат работы возвращать в качестве результата работы функции.

Имена библиотек пишите полностью, вместе с расширением. Без расширения DLL может быть не найдена в Windows NT/2000/ХР.

Обязательно соблюдайте индексы и параметры процедуры, иначе могут возникнуть ошибки. Лучше лишний раз проверить, чем потом долго искать опечатку.

**Хранения формы в динамических библиотеках**

Очень удобно, когда редко используемые окна, находятся в динамической библиотеке. В этом случае основной файл очень сильно разгружается от лишнего кода.

Ещё одно преимущество такого кода – библиотека может использовать u1076 для вывода информации на экран своё окно. Из dll файла нельзя получить доступ к переменным и данным основной программы. Это значит, что из DLL файла нельзя ничего вывести в окна основной программы. Но библиотека может создать собственное окно и использовать для вывода необходимых данных именно его.

Итак, создавайте новую DLL библиотеку и сохраните её под именем *ProjectDLL*. Теперь добавим к нашей библиотеки одну экспортную процедуру *ShowAbout*:

library ProjectDLL;

uses

SysUtils, Classes;

{$R \*.RES}

exports ShowAbout index 10;

begin

end.

Добавили только одну строку *exports ShowAbout index 10;*. Будет только одна процедура *ShowAbout* с индексом 10. Эта процедура будет показывать окно «*О программе*».

Теперь щёлкаем File->New Form , чтобы создать новую форму. Нарисуйте на ней что-нибудь (рисунок 52).



Рисунок - 52. Окно «О программе»

Переходите в текст модуля. В разделе var, после объявления формы опишите процедуру ShowAbout:

var

Form1: TForm1;

procedure ShowAbout(Handle: THandle);export;stdcall**;**

Опять присутствует ключ **export** и добавлен ещё **stdcall**, указывающий на обязательность использования стандартного вызова процедуры.

Теперь напишем саму функцию после ключевого слова **implementation** и ключа *{$R \*.DFM}*:

procedure ShowAbout(Handle: THandle);

422

begin

//*Установить указатель на приложение*

Application.Handle := Handle;

//*Создать форму*

Form1:= TForm1.Create(Application);

//*Отобразить*

Form1.ShowModal;

//*Очистить*

Form1.Free;

end;

Эта процедура получает в качестве параметра указатель на главное приложение. В первой строке устанавливаем этот указатель в свойство *Handle* объекта *Application*. Этот объект хранит настройки всего приложения, и этим присваиванием связали оба приложения. Во второй строке кода создаем окно *TForm1.Create(Application)*, в результате чего будет возвращён указатель на это окно. Результат сохраняем в переменной *Form1*. Эта переменная объявлена в разделе **var** проекта. Следующей строкой отображаем модально созданное нами окно. Как только оно закроется, будет выполнена последняя строка кода этой процедуры, а именно, окно будет уничтожено из памяти и процедура закончит своё выполнение.

В процедурах DLL библиотек будь более внимателен к высвобождению памяти. По моей практике могу сказать, что ошибки в библиотеках переносятся программами более критично, потому что тут основная программа практически бессильна.

Откомпилируйте библиотеку (*Ctrl+F9*) и DLL - файл готов. Можно закрывать этот проект (*File->Close All*) и создавать новое приложение, из которого будем вызывать созданную в библиотеке процедуру (*File->New Application*). В новом проекте переходим в текст формы и объявляем функцию ShowAbout:

unit Unit2;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls,

Forms, Dialogs, StdCtrls;

procedure ShowAbout(Handle: THandle)stdcall;

type

TForm1 = class(TForm)

Button1: TButton;

procedure Button1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

procedure ShowAbout;external 'ProjectDLL.dll' index 10;

implementation

Обратите внимание, что первое описание процедуры написано не в разделе type, а до него:

procedure ShowAbout(Handle: THandle)stdcall;

Это не является ошибкой, и можете выбрать любой из этих способов.

Теперь ставим на форму кнопочку и пишем по её событию *OnClick* следующий код:

procedure TForm1.Button1Click(Sender: TObject);

begin

ShowAbout(Handle);

end;

Запустите пример и убедитесь в том, что пример работает корректно. Как видите окно не так уж трудно поместить в библиотеку, и в нём могут быть свои события и свои процедуры и функции. К тому же окно может создавать дочерние окна по отношению к себе и той же динамической библиотеки.

**7 Инспектор свойств и событий**

**Редакторы свойств**

Как Вы знаете, во время дизайна для настройки внешнего вида и поведения объекта нужно пользоваться Инспектором Объектов. Например, можно изменить цвет фона у объекта TLabel на форме.

Перейдем в окно Инспектора Объектов (рисунок 53) и выберем свойство Color - отметьте, что справа есть маленькая стрелка, она означает, что мы можем выбрать цвет из списка. Нажмите мышкой на эту стрелку.

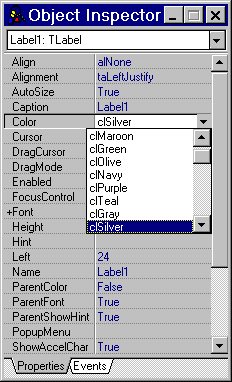


Рисунок – 53. Окно свойств

Столкнувшись с этим в первый раз Вы могли подумать, что этот список цветов является некоей функцией, жестко заданной разработчиками среды программирования Delphi. В действительности, для свойства Color используется соответствующий Редактор Свойств. И Вам вовсе не нужно работать в компании Borland, чтобы создать подобные Редакторы Свойств. Точно так же, как Вы добавляете новые компоненты в Delphi, Вы можете добавить свой собственный Редактор Свойств в среду разработки.

**Стандартные Редакторы Свойств**

Методы, приведенные ниже, можно переопределять (override) для изменения поведения Редактора свойств. ( "SetXxxValue" используется для представления одного из методов SetFloatValue, SetMethodValue, SetOrdValue или SetStrValue. "GetXxxValue" обозначает GetFloatValue, GetMethodValue, GetOrdValue или GetStrValue)

* **Activate**

 Вызывается, когда свойство выбирают в инспекторе объектов. Может быть полезно позволить некоторым атрибутам свойства определяться в каждый момент выбора этого свойства.

* **AllEqual**

 Вызывается всякий раз, когда на форме выбирается более чем один объект. Если этот метод возвращает True, то вызывается GetValue, иначе в Инспекторе Объектов показывается пустая строка. AllEqual вызывается при условии, что GetAttributes возвращает paMultiSelect.

* **Edit**

 Вызывается при нажатии кнопки '...' или по двойному щелчку мыши на свойстве. Этот метод может, к примеру, показать какое-нибудь диалоговое окно для редактирования свойства (пример - свойство Font).

* **GetAttributes**

 Возвращает необходимую Инспектору Объектов информацию для того, чтобы тот смог отобразить свойство в подходящей манере. GetAttributes возвращает множество (set) значений типа TPropertyAttributes:

 **paValueList:** Редактор свойств может возвращать список значений для этого свойства. Если этот атрибут установлен, то нужно определить GetValues. В Инспекторе объектов справа от свойства появится кнопка для выпадающего списка.

**paSortList:** Инспектор объектов будет сортировать список, полученный от GetValues.

**paSubProperties:** Свойство имеет подсвойства, которые будут показываться ниже в виде иерархии (outline). Если GetProperties будет генерировать объекты-свойства, то этот атрибут должен быть установлен.

**paDialog:** Показывает, что метод Edit будет вызывать диалог. Если данный атрибут установлен, то появится кнопка '...' справа от свойства в Инспекторе Объектов.

**paMultiSelect:** Позволяет свойству оставаться в Инспекторе Объектов, когда на форме выбрано сразу несколько объектов. Некоторые свойства не годятся для множественного выбора, например, Name.

**paAutoUpdate:** Если этот атрибут установлен, то метод SetValue будет вызываться при каждом изменении, произведенном в редакторе, а не после завершения редактирования (пример - свойство Caption).

**paReadOnly:** Значение менять нельзя.

* **GetComponent**

 Возвращает компонент под номером Index в случае множественного выбора объектов (multiselect). GetAttributes должен возвращать paMultiSelect.

* **GetEditLimit**

 Возвращает число символов, которые пользователь может ввести при редактировании свойства. По умолчанию 255 символов.

* **GetName**

 Возвращает имя свойства. По умолчанию это имя получается из информации о типе, все подчеркивания замещаются пробелами. Данный метод Вам нужно переопределять только в том случае, если имя свойства отличается от того, которое нужно отображать в Инспекторе Объектов.

* **GetProperties**

 Должен быть переопределен для вызова PropertyProc для каждого подсвойства (или вложенного свойства) редактируемого свойства и передачи нового TPropertyEdtior для каждого подсвойства. По умолчанию, PropertyProc не вызывается и подсвойства не ожидаются. TClassProperty будет передавать новый редактор свойств для каждого свойства, объявленного **published** в классе. TSetProperty передает новый редактор для каждого элемента множества.

* **GetPropType**

 Возвращает указатель на информацию о типе редактируемого свойства.

* **GetValue**

 Возвращает значение свойства в виде строки. По умолчанию возвращает '(unknown)'. Этот метод нужно переопределять с тем, чтобы возвращать правильное значение.

* **GetValues**

 Вызывается, если GetAttributes возвращает paValueList. Должно вызвать Proc для каждого значения, которое приемлемо для данного свойства.

* **Initialize**

 Вызывается при создании Редактора свойств.

* **SetValue(Value)**

 Вызывается для того, чтобы установить значение свойства. Редактор свойств должен уметь разобрать строку (Value) и вызвать метод SetXxxValue. Если строка имеет некорректный формат или неверное значение, то редактор Свойств должен сгенерировать исключительную ситуацию (exception), описывающую данную проблему. SetValue может вообще проигнорировать все изменения и оставить всю обработку изменений методу Edit (как в свойстве Picture).

Свойства и методы полезные при создании нового класса Редактора свойств:

* **PrivateDirectory (**свойство**)**

 Это директория, в которой находится .EXE, либо рабочая директория, указанная в DELPHI.INI. Если редактор должен сохранить какую-то информацию (установки), то лучше в этой директории.

* **Value (**свойство**)**

Текущее значение свойства, то же самое возвращает GetValue.

* **Modified (**метод**)**

 Вызывается для того, чтобы показать, что значение свойства изменилось. Методы SetXxxValue вызывают Modified автоматически.

* **GetXxxValue (**метод**)**

 Возвращает значение первого из редактируемых свойств.

* **SetXxxValue (**метод**)**

 Устанавливает значения свойства для всех выбранных объектов.

**Создание Редактора Свойств**

При создании нового Редактора Свойств, конечно, не нужно всегда переписывать его заново от базового класса TPropertyEditor. Может оказаться достаточным выбрать в качестве предка уже существующий для данного свойства редактор и переопределить некоторые его методы. Давайте рассмотрим простейший пример нового Редактора Свойств. Как Вы знаете, у всех видимых объектов есть свойство Hint - подсказка, появляющаяся во время выполнения программы, если задержать на некоторое время мышь на объекте. Это свойство имеет тип String и во время дизайна для его редактирования используется Редактор типа TStringProperty. Обычно, подсказка бывает однострочной, но в некоторых случаях ее нужно сделать многострочной. В принципе, здесь проблемы нет, достаточно во время выполнения программы присвоить свойству Hint нужное значение, например:

Button1.Hint:=’Line1’#13#10’Line2’;

Теперь подсказка будет состоять из двух строк. Но это достаточно неудобно, более удобно было бы формировать многострочную подсказку во время дизайна, однако редактор TStringProperty такой возможности не дает. Давайте создадим новый редактор, который мог бы это сделать.

В нашем случае будет достаточно выбрать в качестве предка редактор TStringProperty и переписать некоторые методы. Во-первых, нужно переопределить метод Edit, в котором будет вызываться диалог для ввода строк подсказки. Во-вторых, нужно переопределить функцию GetAttributes, которая возвращает набор параметров, описывающих данное свойство. В частности, должен быть установлен атрибут paDialog, при этом в Инспекторе Объектов у свойства появится кнопка ‘…’ для вызова диалога. И вообще-то нужно изменить метод GetValue, который используется для отображения значения свойства в Инспекторе Объектов.

Назовем новый Редактор Свойств THintProperty, декларация нового класса:

THintProperty = class(TStringProperty)

public

function GetAttributes: TPropertyAttributes; override;

function GetValue : String; override;

procedure Edit; override;

end;

Рассмотрим по порядку методы нового класса.

Функция GetAttributes добавляет к унаследованному множеству атрибуты paDialog (появляется кнопка ‘…’) и paReadOnly (свойство нельзя редактировать непосредственно в Инспекторе Объектов, а только в диалоге, вызываемом через кнопку ‘…’):

function THintProperty.GetAttributes: TPropertyAttributes;

begin

Result := inherited GetAttributes + [paDialog,

paReadOnly];

end;

Функция GetValue заменяет “неправильные” символы #10 и #13 (перевод каретки и переход на новую строку) на символ “>”:

function THintProperty.GetValue : string;

var

i : Byte;

begin

result:=inherited GetValue;

for i:=1 to Byte(result[0]) do

if result[i]<#32 then result[i]:='>';

end;

Процедура Edit вызывает диалог для ввода строк подсказки. Диалог можно было бы нарисовать свой собственный, однако можно воспользоваться уже готовым. Несколько разных диалогов лежит в директории X:\DELPHI\SOURCE\LIB. Мы воспользуемся модулем STREDIT.PAS, в котором есть необходимый диалог редактирования строк. Итак, процедура Edit:

procedure THintProperty.Edit;

var

HintEditDlg : TStrEditDlg;

s : string;

begin

HintEditDlg:=TStrEditDlg.Create(Application);

with HintEditDlg do

try

Memo.MaxLength := 254;

s:=GetStrValue+#0;

Memo.Lines.SetText(@s[1]);

UpdateStatus(nil);

ActiveControl := Memo;

if ShowModal = mrOk then begin

s:=StrPas(Memo.Lines.GetText);

if s[0]>#2 then Dec(Byte(s[0]),2);

SetStrValue(s);

end;

finally

Free;

end;

end;

Строка if s[0]>#2 then Dec(Byte(s[0]),2) нужна, так как Memo.Lines.GetText возвращает все строки с символами #13#10.

**Регистрация Редактора Свойств**

Новый Редактор Свойств готов, осталось только его зарегистрировать в среде Delphi. Для этого в интерфейсной части модуля с нашим редактором требуется поместить декларацию процедуры Register, а в части **implementation** написать следующее:

procedure Register;

begin

RegisterPropertyEditor(TypeInfo(String), TControl, 'Hint',

THintProperty);

end;

Как уже сообщалось выше, один и тот же редактор свойств можно “привязать” к свойствам, в зависимости от их названия или типа объекта. Это определяется параметрами (второй и третий), которые передаются во время регистрации в процедуре RegisterPropertyEditor. Возможны четыре варианта:

Таблица – 4. Класс компоненты, имя свойства

|  |  |  |
| --- | --- | --- |
| Класс компоненты | Имя свойства | Для каких свойств |
| Nil | ‘’ | совпадает тип свойства |
| Nil | ‘Name’ | Тип свойства + Имя свойства |
| TClass | ‘’ | Тип свойства + класс компоненты |
| TClass | ‘Name’ | Тип свойства + Имя свойства+ класс компоненты |

Пояснение к таблице 4. Если вы зарегистрировали Редактор и указали как класс компоненты, так и имя свойства, то данный редактор “привязывается” ко всем свойствам, которые:

* + имеют тип, указанный в первом параметре процедуры;
  + принадлежат компоненте, которая относится к классу (или его потомкам), указанному во втором параметре;
  + имеют имя, совпадающее с указанным в третьем параметре;

Если вместо типа класса в процедуре регистрации стоит **Nil**, а вместо имени свойства - пустая строка ‘’, то данный редактор “привязывается” ко **всем** свойствам, которые имеют тип, указанный в первом параметре, независимо от их имени или принадлежности к объекту какого-либо класса.

Если указан только класс, то редактор относится ко всем свойствам указанного типа для объектов указанного класса.

Если указано только имя, то редактор относится к свойствам указанного типа, которые имеют указанное имя.

В нашем случае Редактор Свойств зарегистрирован для всех свойств, которые имеют тип **String**, относятся к компоненте класса TControl или наследника от него и имеют имя ‘Hint’.

**Установка Редактора свойств**

После того, как модуль с новым редактором свойств подготовлен, его нужно подключить к среде Delphi. Установка Редактора Свойств абсолютно аналогична установке новых объектов в палитру компонент и происходит следующим образом:

* выберите пункт меню "Options|Install Components...."
* нажмите кнопку “Add”
* укажите имя подключаемого модуля (или воспользуйтесь кнопкой “Browse”)
* нажмите “OK” и еще раз “OK”

После успешной перекомпиляции библиотеки проверьте, как действует новый редактор свойств. Для этого создайте новый проект, положите на форму какой-либо видимый объект, например TButton, установите ShowHint для него в True, вызовите редактор подсказки (рисунок 54) (кнопка ‘…’ в свойстве Hint), редактор выглядит примерно так:

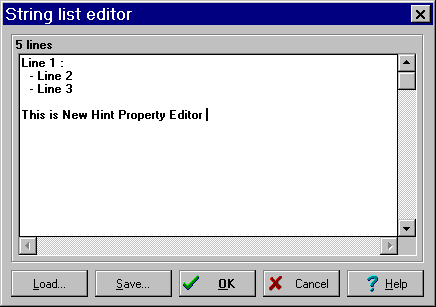


Рисунок – 54. Окно редактора подсказки

В диалоге нажмите “OK” и запустите программу.

Полный текст модуля с Редактором Свойств см. в примерах к данному уроку.

Методы и свойства TComponentEditor:

**Create**(**AComponent**, **ADesigner**): Конструктор Редактора Компонент. AComponent - редактируемый компонент. ADesigner - интерфейс к Дизайнеру среды Delphi.

**Edit**: Вызывается при двойном щелчке мышью на компоненте. Редактор Компонент может вызвать какой-нибудь диалог или эксперт.

**ExecuteVerb**(**Index**): Выполняется, когда был выбран пункт номер Index из контекстного меню. Считается, что Редактор Компонент знает, как проинтерпретировать это значение.

**GetVerb(Index)**: Редактор Компонент должен вернуть в этом методе строку, которая будет показана в виде пункта контекстного меню. Можно использовать обычные для пунктов меню символы, например &.

**GetVerbCount**: Возвращает число, которое определяет на какие значения будут отвечать методы GetVerb и ExecuteVerb. Например, если это число равно 3, то в меню будет добавлено три пункта со значениями Index от 0 до 2.

**Copy**: Вызывается, когда компонент нужно скопировать в Clipboard. На самом деле, образы полей компонента уже находятся в Clipboard. Просто предоставляется возможность скопировать различные типы форматов, которые игнорируются Дизайнером, но которые могут быть распознаны другими приложениями.

**Инспектор объекта**

Окно Инспектора объекта отображается по левому краю экрана (рисунок 55). Он является тем инструментом среды Delphi, который в полной мере позволяет контролировать компоненты в составе приложения. С его помощью можно устанавливать требуемые значения свойств компонентов и задавать их реакцию на стандартные события.

Внутри Инспектора объекта имеется две страницы, переход со страницы на страницу осуществляется щелчком левой кнопки мыши по вкладкам. В верхней части имеется комбинированная панель выбора компонентов, размещенных на форме. Она содержит список компонентов активизированной формы в алфавитном порядке. Страницы имеют названия Properties (свойства) и Events (события). Выбирая какой-либо компонент из списка или активизируя (выделяя) компонент на форме, страницы Инспектора объекта заполняются значениями его свойств и именами обработчиков событий (рисунок 55).

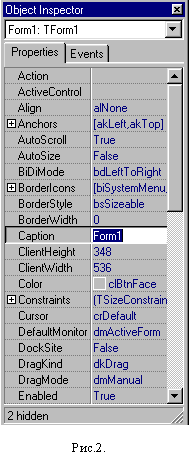


Рисунок – 55. Окно Инспектора объекта

**Страница свойств (Properties).**

Свойства компонентов, отображаемые на странице Properties, могут быть комплексными или простыми. Комплексными свойствами называют те свойства, которые состоят из набора нескольких свойств. Они обозначаются знаком "+". Для просмотра комплексного свойства необходимо дважды щелкнуть на его имени, и в колонке имен свойств разворачивается дополнительный список, при этом знак перед именем комплексного свойства меняется на "-".

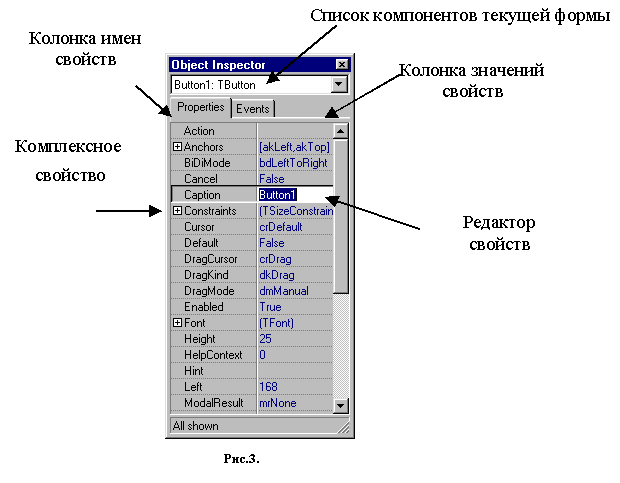


Рисунок – 56. Окно Инспектора объекта

В Delphi 5 для Инспектора объекта введена возможность фильтрации свойств и событий, группирования их по категориям. Для того чтобы воспользоваться этими возможностями, щелкните в Инспекторе объекта правой кнопкой мыши. Во всплывшем меню выберите раздел View. Вы увидите ряд категорий свойств и событий (рисунок 57). Около каждой категории имеется индикатор, который можно активизировать, поставив галочку в окне.

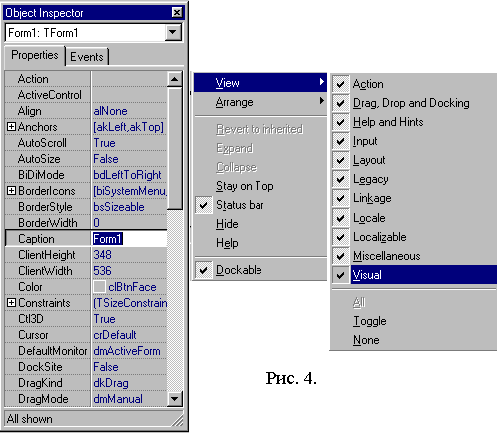


Рисунок – 57. Окно Инспектора объекта. Категории свойств и событий

**Контрольные вопросы:**

1. Что называют визуальным программированием?
2. Чем отличается графический язык программирования от визуальных средств разработки?
3. Языки визуального программирования могут быть дополнительно классифицированы в зависимости от типа и степени визуального выражения, на следующие типы…
4. Что называют визуально-преобразованными языками программирования?
5. Что называют естественно-визуальными языками программирования?
6. Приведите примеры графических языков программирования.
7. Приведите примеры визуальных средств разработки.
8. Инструментальное программное обеспечение это….
9. Какие группы можно выделить из существующие на сегодняшний день языков программирования?
10. Что называют интегрированной средой программного обеспечения?
11. Что входит в состав интегрированной среды разработки?
12. Что включает в себя современная интегрированная среда разработки?
13. Для чего предназначен Master Wizard?
14. Из скольких видов скриптов может состоять Master Wizard?
15. Скрипты типа Form {…} могут быть:
16. Для чего служит скрипт Menu?
17. Перечислим некоторые из областей их применения динамических библиотек.
18. Приведите пример создания DLL.
19. Посторное использование текста программ, что это?
20. Поясните что, означает событийно-ориентированное программирование?
21. Что представляет собой инспектор событий в языках программирования?
22. Что представляет собой редактор свойств в язвках программирования?