Глава 1

Обзор компонентов

В этой главе...

Общие свойства компонентов

Иерархия компонентов

Родительские классы компонентов

Компонент Form

Резюме

В Delphi 7 используется специальная *библиотека визуальных компонентов* (VCL — Visual Component Library), которая в своей основе практически такая же, как и во всех предыдущих версиях. Но в этой версии Delphi также имеется и новая *библиотека компонентов для различных платформ* (CLX— Component Library for Cross-Platform), которая впервые была введена в Delphi 6. Это библиотека компонентов следующего поколения и основа для разработки приложений и компонентов многократного использования, совместимых как с Linux, так и с Windows.

Обе библиотеки, и VCL и CLX, разработаны специально для графической среды разработки Delphi и во многом похожи друг на друга, хотя есть и значительные отличия.

Уровень знаний, необходимый для работы с VCL и CLX, зависит от способа их применения. Программистов Delphi можно разделить на разработчиков приложений и создателей визуальных компонентов. Первые создают приложения непосредственно в визуальной среде Delphi, где с помощью библиотек VCL и CLX они разрабатывают графический интерфейс и проектируют другие элементы приложения. Вторые создают новые компоненты, расширяя стандартные возможности VCL и CLX. Поэтому и специфика знаний у каждого из них должна быть разная. Если разработчики компонентов должны знать и понимать все тонкости той технологии, для которой они разрабатывают компоненты, то разработчики приложений должны иметь более широкий кругозор и иметь представление обо всех компонентах, при этом не углубляясь в детали реализации. В данной книге будет дано общее описание всех компонентов, которое может пригодиться разработчикам приложений. Как создавать собственные компоненты, рассказано в приложении А.

Общие свойства компонентов

Компоненты — это готовые фрагменты программы, используя которые разработчик создает графический интерфейс пользователя и включает в приложение необходимые невизуальные элементы. Для разработчиков приложений ком-

понент — это объект, который из палитры компонентов перенесен в форму. После помещения компонента в форму можно манипулировать его свойствами и добавлять обработчики событий для придания ему необходимого вида и требуемого поведения. Компоненты для разработчика компонентов — это класс, спроектированный для выполнения определенных задач. Поведение компонента полностью определяется кодом, написанным разработчиком компонента.

> *Компоненты* — более общее понятие, чем часто используемое в литературе по Windows определение *элемент управления*. Помимо того, что компоненты могут быть визуальными и обрабатывать сообщения системы Windows, существуют не отображаемые на экране и не получающие фокус ввода компоненты.

По своей сложности компоненты могут значительно отличаться друг от друга. Существуют как совсем простые компоненты, так и очень сложные. На степень сложности компонента нет никаких ограничений. Есть простые компоненты, например `TLabel` (Надпись), или компоненты, инкапсулирующие функциональные возможности целой электронной таблицы.

Для работы с VCL необходимо знать типы существующих компонентов, понимать их иерархию и особенности каждого уровня этой иерархии.

Иерархия компонентов

Ниже (рис. 1.1) представлена иерархическая диаграмма классов библиотеки VCL. На диаграмме выделены два типа компонентов: визуальные и невизуальные. Также обратите внимание, что все компоненты базируются на классах `TComponent` и `TPersistent`. Это обстоятельство и определяет общие свойства компонентов.

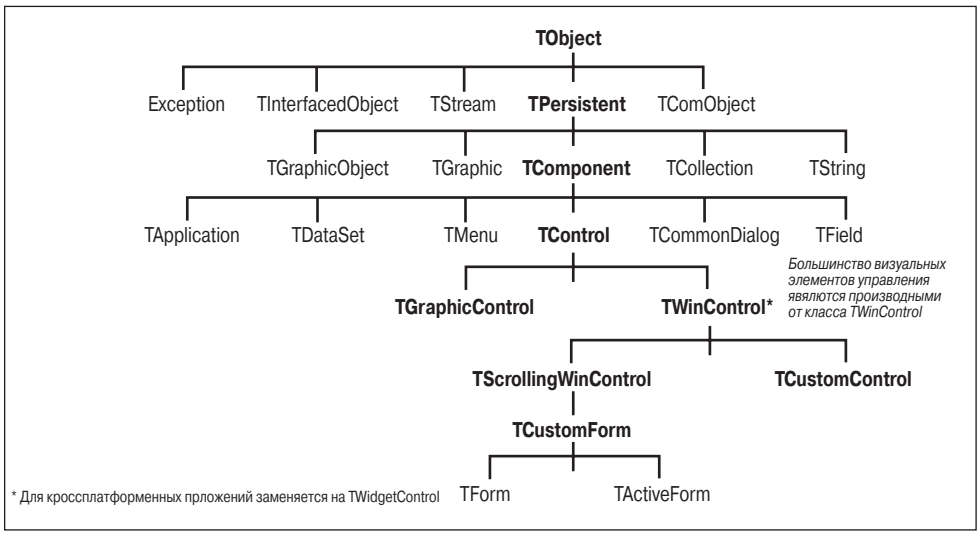

Рис. 1.1. *Диаграмма классов библиотеки VCL*

Невизуальные компоненты

Как можно понять из названия, *невизуальные компоненты* не отображаются на экране во время выполнения приложения. Но они выполняют определенные задачи, и в них заложены специализированные функциональные возможности, которые разработчик может несколько изменить, модифицировав с помощью инспектора объектов свойства компонента или создав обработчики его событий. В качестве примеров можно привести классы `TOpenDialog`, `TTable` и `TTimer`. Классы этих невизуальных компонентов наследуются непосредственно от класса `TComponent`.

Визуальные компоненты

Визуальные компоненты являются теми компонентами, которые пользователь видит на экране. Визуальные компоненты отображаются на экране во время работы приложения, они выполняют необходимые задачи по отображению информации, вводу данных и интерактивному общению, хотя не все из них способны взаимодействовать с пользователем. Классы этих компонентов происходят непосредственно от класса `TControl`. Фактически класс `TControl` был задуман как базовый для подобных компонентов. Он обладает такими свойствами и методами, которые необходимы для представления элемента управления на экране, задания его размеров, положения и внешнего вида.

Визуальные компоненты бывают двух видов: способные и не способные получать фокус.

Визуальные компоненты, способные получать фокус

Большинство элементов управления способно получать фокус ввода. Это означает, что пользователь может взаимодействовать с такими элементами управления. Эти типы элементов управления являются потомками класса `TWinControl` из библиотеки VCL. Потомки класса `TWinControl` являются аналогами стандартных элементов управления Windows и обладают следующими базовыми возможностями.

- Они способны получать фокус ввода и реагировать на события клавиатуры.
- Пользователь может взаимодействовать с ними.
- Они способны быть контейнерами, т.е. включать в себя другие элементы управления.
- В системе Windows им присваивается уникальный дескриптор.

> Дескриптор представляет собой 32-разрядное число, указывающее на определенный экземпляр объекта в системе Win32 (имеются в виду объекты Win32, а не Delphi). В Win32 существуют различные типы объектов: объекты ядра, пользовательские объекты и объекты GDI. Термин “объекты ядра” применяют к событиям, объектам отображения файлов и процессам. К пользовательским объектам относят объекты окон, например поля редактирования, раскрывающиеся списки и кнопки. К объектам GDI относят растровые изображения, кисти, шрифты и т.д.
> Delphi поддерживает работу с дескрипторами, поэтому если необходимо использовать функции API Windows, для выполнения которых требуется дескриптор, используйте потомки классов `TWinControl` и `TCustomControl`, так как оба эти компонента обладают свойством Handle (Дескриптор).

Визуальные компоненты, не способные получать фокус

Некоторые элементы управления хоть и видны на экране, но не обладают возможностью взаимодействия с пользователем. Такие элементы предназначены только для отображения определенной информации, поэтому их называют еще *графическими элементами управления*. Классы графических элементов управления происходят непосредственно от класса `TGraphicControl`.

В отличие от обычных элементов управления, графические элементы управления не получают фокус ввода. Они очень полезны, когда нужно отобразить что-либо на экране и не хочется расходовать так же много системных ресурсов, как для элементов управления. Графические элементы не занимают ресурсы Windows, и им не выделяется дескриптор. Но из-за того, что отсутствует дескриптор, графические элементы не способны получать фокус. К графическим элементам относятся, например `TLabel` и `Tshape`. Они не могут выступать в качестве контейнеров и не способны включать другие элементы управления. Также к графическим элементам управления относятся `TImage`, `TBevel` и `TPaintBox`.

Родительские классы компонентов

Класс TObject

Как читателю уже должно быть известно, базовым классом, от которого наследуются все остальные классы в языке Delphi, является класс `TObject`. Но разработчики компонентов не должны создавать свои компоненты непосредственно как производные (потомки) от класса `TObject`. Библиотека VCL предоставляет широкий выбор классов, производных от класса `TObject`, и разрабатываемые компоненты могут создаваться на их основе. Эти уже существующие классы обеспечивают большинство функциональных возможностей, необходимых для вновь создаваемых компонентов. Лишь при создании классов, не являющихся компонентами, имеет смысл делать их потомками класса `TObject`.

Класс `TObject` определяет фундаментальное поведение всех объектов и содержит методы, которые необходимы при создании, обслуживании и разрушении экземпляров класса, размещая, инициализируя и освобождая память. Методы класса `TObject` используются при:

- создании и уничтожении объекта;
- получении информации об объекте и типе класса, а также информации о типах времени выполнения;
- обработке сообщений;
- поддержке использования интерфейсов для объекта.

Базовые особенности объектов определяются методами, которые представлены в классе `TObject`. Но многие из этих методов используются только интегрированной средой разработки и не предназначены для непосредственного вызова в разрабатываемой программе. Другие методы могут быть переопределены в унаследованных классах для создания специфического поведения.

Хотя класс TObject лежит в основе всех компонентов, сами компоненты должны быть производными от класса TComponent. Ниже приводится описание всех методов класса TObject.

Procedure AfterConstruction; virtual; Метод AfterConstruction вызывается автоматически после создания объекта. Нельзя вызывать эту процедуру явно в разрабатываемой программе.

В классе TObject этот метод ничего не делает. При разработке нового класса переопределите этот метод для выполнения необходимых действий по созданию объекта. Например, в классе TCustomForm этот метод переопределен для генерации события OnCreate.

Procedure BeforeDestruction; virtual; Метод BeforeDestruction вызывается автоматически перед разрушением объекта. Нельзя вызывать эту процедуру явно в разрабатываемой программе.

В классе TObject этот метод ничего не делает. При разработке нового класса переопределите этот метод для выполнения необходимых действий для разрушения объекта. Например, в классе TCustomForm этот метод переопределен для генерации события OnDestroy.

Но если объект разрушается до того, как он полностью создан, то метод BeforeDestruction не вызывается. Это может случиться, если в конструкторе возникла исключительная ситуация и вызывается деструктор для ликвидации объекта.

Class function ClassInfo: Pointer; Метод ClassInfo используется для доступа к таблице RTTI объекта определенного типа. Некоторые классы не создают таблицы RTTI, и для таких классов метод ClassInfo возвращает значение nil. Все классы, унаследованные от класса TPersistent, содержат информацию о типах времени выполнения.

Class function ClassName: ShortString; Используйте метод ClassName для получения имени класса из объекта или ссылки на класс. Это помогает различать объекты во время присваивания значений переменным.

Class function ClassParent: TClass; Метод ClassParent возвращает имя родительского класса для объекта или ссылки. Для типа TObject возвращается значение nil. Не используйте метод ClassParent непосредственно в коде приложения, для этого предназначены операторы is или as.

Function ClassType: TClass; Метод ClassType динамически определяет тип объекта и возвращает ссылку на соответствующий класс или метакласс. Не используйте метод ClassType непосредственно в коде приложения, для этого предназначены операторы is или as.

Procedure CleanupInstance; Метод CleanupInstance вызывается автоматически при разрушении объекта, поэтому нет необходимости вызывать его непосредственно. CleanupInstance освобождает все длинные строки и варианты. Для строк он устанавливает пустое значение, а для вариантов — значение Unassigned.

Constructor Create; Конструктор Create необходим для создания объекта. Назначение, размер и поведение отдельных объектов значительно различаются. Конструктор, определенный в классе TObject, распределяет память, но не ини-

циализирует данные, поэтому обычно конструкторы определяются в производных классах для создания специфических объектов и инициализации данных.

Procedure DefaultHandler(var Message); virtual; Метод `DefaultHandler` вызывается методом `Dispatch` в том случае, когда `Dispatch` не может найти подходящий обработчик для отдельного сообщения. В методе `DefaultHandler` производится обработка всех сообщений, для которых в объекте нет необходимого обработчика. В производных классах необходимо переопределять метод `DefaultHandler` для работы с другими типами сообщений.

В Delphi производится вызов метода `DefaultHandler` родительского (базового) класса только в том случае, если в родительском классе также не определен подходящий обработчик сообщения. В противном случае вызывается обработчик родительского класса.

Destructor Destroy; virtual; Деструктор для разрушения объектов и освобождения занимаемой ими памяти. Не вызывайте этот метод непосредственно, это может привести к сбою программы. Вместо этого вызывайте метод `Free`, который более безопасен, так как проверяет существование ссылки на объект и только после этого вызывает метод `Destroy`.

Метод `Destroy`, определенный в классе `TObject`, освобождает память. В производных классах можно определять деструктор, в котором учитываются особенности определенного объекта, но после этого в деструкторе должен быть сделан вызов унаследованного метода `Destroy`. Так как метод `Destroy` класса `TObject` является виртуальным, в производных классах необходимо переопределять этот метод.

При объявлении метода `Destroy` в производных (дочерних) классах всегда добавляйте директиву `override` и вызывайте метод `Destroy` родительского класса в конце тела метода.

Procedure Dispatch(var Message); virtual; В методе `Dispatch` производится автоматическая диспетчеризация сообщений, т.е. поиск подходящего обработчика сообщения и передача сообщения этому обработчику. Для этого просматривается список всех обработчиков для данного объекта. Если в этом списке нет подходящего обработчика, то производится поиск в списках родительских классов. Если и там не найден нужный обработчик, автоматически вызывается метод `DefaultHandler`.

Идентификация сообщения производится по первым двум байтам, в которых содержится идентификатор сообщения ID, — уникальное целочисленное значение. Хотя в теле сообщения могут находиться любые данные, но обычно это запись типа `Tmessage` или специфические для среды Delphi структуры данных.

Function FieldAddress(const Name: ShortString): Pointer; Метод `FieldAddress` используется внутренними потоковыми системами компонентов для доступа к полям объекта в экспортируемом интерфейсе. Метод возвращает указатель на поле, если оно существует, Если в объекте необходимого поля не существует, то возвращается `nil`.

Для доступа к полям программист должен использовать свойства, а не метод `FieldAddress`.

Procedure Free; Используйте метод `Free` для уничтожения (ликвидации) объекта. В методе `Free` автоматически вызывается деструктор, если значение ссылки на объект не равно `nil`. Любой объект, созданный во время работы програм-

мы и не имеющий хозяина, должен быть корректно уничтожен с помощью метода Free для освобождения занимаемой им памяти. В отличие от метода Destroy, метод Free корректно работает даже в тех случаях, когда объект не был правильно инициализирован и ссылка на объект имеет значение nil.

Когда производится вызов метода Free для компонента, он вызывает методы Free всех входящих в данный компонент объектов в соответствии со списком. Так как форма владеет всеми элементами управления и другими компонентами, созданными во время проектирования, то все эти элементы будут автоматически уничтожены после ликвидации формы.

По умолчанию объект Application владеет всеми формами приложения, поэтому при закрытии приложения уничтожаются и все формы. Для объектов, которые не являются компонентами, или для объектов, не имеющих хозяина, необходимо вызывать метод Free после окончания работы с ними. Иначе они будут только занимать память до окончания работы приложения.

Запрещено явно ликвидировать компоненты в обработчиках событий. Например, не уничтожайте кнопку или форму, владеющую кнопкой, в обработчике события OnClick.

Для уничтожения формы вызывайте метод Release, который ликвидирует форму только после того, как освободит память, занимаемую всеми включенными в форму компонентами и обработчиками событий.

Procedure FreeInstance; virtual; Все деструкторы автоматически вызывают метод FreeInstance для освобождения памяти, которая была задействована для объекта методом NewInstance.

Не используйте непосредственно метод FreeInstance. Метод FreeInstance должен быть переопределен, если был переопределен метод NewInstance и изменен способ распределения памяти.

Подобно методу NewInstance, метод FreeInstance использует значение, возвращаемое методом InstanceSize.

Function GetInterface(const IID: TGUID; out Obj): Boolean; Метод GetInterface отыскивает интерфейс по глобально уникальному идентификатору или типу имени. Основная реализация GetInterface использует глобально уникальный идентификатор, передаваемый через параметр IID. Если определенный интерфейс поддерживается классом, то в параметре Obj возвращается ссылка, а сам метод GetInterface возвращает значение True. В противном случае значение параметра Obj будет равно nil, а возвращаемое методом значение равно False.

В коде Delphi параметр IID может принимать имя интерфейса. Компилятор автоматически транслирует имя в соответствующий глобально уникальный идентификатор.

Class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry; Метод GetInterfaceEntry возвращает точку входа в интерфейс, указанный в параметре IID.

В коде Delphi параметр IID может принимать имя интерфейса. Компилятор автоматически транслирует имя в соответствующий глобально уникальный идентификатор.

Объекты COM могут использовать GetInterfaceEntry для автоматизации диспетчеризации в дуальном интерфейсе IDispatch.

Class function GetInterfaceTable: PInterfaceTable; Метод `GetInterfaceTable` возвращает содержащиеся в классе интерфейсы. В списке присутствуют интерфейсы данного класса, но не родительских классов. Для получения интерфейсов, входящих в родительские классы, вызывайте метод `ClassParent` и используйте возвращаемое значение для вызова `GetInterfaceTable`. Для поиска точки входа отдельного интерфейса используйте метод `GetInterfaceEntry`.

Class function InheritsFrom(aClass: TClass): Boolean; Используйте метод `InheritsFrom` для определения принадлежности к определенному родительскому классу. Возвращаемое значение `True` говорит о том, что класс, указанный в параметре `aClass`, является родительским для данного класса. В противном случае возвращается значение `False`. Этот метод подобен оператору `is`, но может работать со ссылками на класс.

Class function InitInstance(Instance: Pointer): TObject; Нет необходимости вызывать метод `InitInstance` непосредственно. Он вызывается автоматически из метода `NewInstance` при создании объекта. При переопределении `NewInstance` последним утверждением должен быть вызов метода `InitInstance`.

Метод `InitInstance` не является виртуальным и его нельзя переопределять, поэтому все данные должны быть инициализированы в конструкторе.

Class function InstanceSize: Longint; Метод `InstanceSize` возвращает количество памяти в байтах, которое требуется для экземпляра класса. Этот метод вызывается из методов, которые размещают и освобождают память. Метод `InstanceSize` не является виртуальным, и его нельзя переопределять. Необходимость в вызове этого метода может возникнуть при переопределении метода `NewInstance`.

Class function MethodAddress(const Name: ShortString): Pointer; Метод `MethodAddress` используется внутри потоковых систем. Когда производится считывание из потока, метод `MethodAddress` преобразует имя метода, указанное в параметре `Name`, в указатель, содержащий адрес метода. Программисту нет необходимости непосредственно использовать этот метод.

Class function MethodName(Address: Pointer): ShortString; Метод `MethodName` используется внутри потоковых систем. В нем преобразуется адрес в строку с именем метода. Программисту нет необходимости непосредственно использовать этот метод.

Class function NewInstance: TObject; virtual; Все конструкторы автоматически вызывают метод `NewInstance`. В `NewInstance` вызывается метод `InstanceSize` для определения количества памяти в куче, необходимой для размещения объекта. Не вызывайте непосредственно метод `NewInstance`.

При необходимости самостоятельного распределения памяти метод `NewInstance` можно переопределить, например при необходимости одновременного размещения большого числа идентичных объектов в одном блоке памяти.

При переопределении метода `NewInstance` необходимо переопределить и метод `FreeInstance`.

По умолчанию метод `NewInstance` вызывает `InitInstance`.

Function SafeCallException(ExceptObject: TObject; ExceptAddr: Pointer): HResult; virtual; Метод `SafeCallException` обрабатывает исключения в методах, которые используют защищенное соглашение о вызовах. В некоторых классах, использующих интерфейсы, этот метод переопределен для обработки возможных ошибок. Реализация `SafeCallException` в классе `TObject` просто возвращает значение `E_UNEXPECTED`. Такая реализация характерна для классов, не использующих интерфейсы.

Используйте метод TObject.Free вместо метода TObject.Destroy. Метод Free вызывает метод Destroy, но перед этим проверяет, имеет ли указатель на объект значение nil, что позволяет избежать генерации исключения при попытке уничтожить несуществующий объект.

Класс TPersistent

Класс TPersistent происходит непосредственно от класса TObject. Особенностью класса TPersistent является то, что экземпляры происходящих от него объектов могут читать и записывать свои свойства в поток. Поскольку все компоненты являются потомками класса TPersistent, то все они обладают этой способностью. Класс TPersistent определяет не специальные свойства или события, а только некоторые методы, полезные как пользователям, так и разработчикам компонентов.

Методы класса TPersistent

Procedure Assign(Source: TPersistent); virtual; Метод Assign вызывается для копирования свойств и других атрибутов из одного объекта в другой. Стандартная форма вызова метода Assign в Delphi следующая:

```
Destination.Assign(Source);
```

здесь в целевой объект Destination копируется содержимое исходного объекта Source.

В большинстве объектов метод Assign переопределяется для более подходящего распределения свойств аналогичных объектов. При переопределении метода вызывайте унаследованный метод, если целевой объект не рассчитан на обработку определенных типов классов, передаваемых через параметр Source.

Если ссылка на исходный объект имеет значение nil, то генерируется исключительная ситуация EConvertError. Если метод Assign не может скопировать свойства исходного объекта, то вызывается метод AssignTo исходного объекта для копирования данных.

Если использовать оператор присваивания

```
Destination := Source;
```

то произведенное действие будет абсолютно другим, отличающимся от

```
Destination.Assign(Source);
```

Оператор присваивания просто присваивает объекту Destination значение ссылки на объект Source, в то время как метод Assign копирует содержимое, определяемое ссылкой объекта Source, по адресу, определяемому ссылкой для объекта Destination.

Но свойства и сами могут быть объектами. Если в этом случае свойства имеют методы, использующие метод Assign для присваивания значений свойствам, то действие оператора присваивания может быть подобно действию метода Assign.

Procedure AssignTo(Dest: TPersistent); virtual; Переопределите метод AssignTo для расширения функциональных возможностей метода Assign целевого объекта для обработки вновь создаваемых классов. Когда создается новый класс, переопределите метод Assign для того, чтобы объекты могли оперировать экземплярами вновь созданного класса. Переопределите метод AssignTo для всех существующих классов, которые будут копировать новый класс.

Метод Assign класса TPersistent вызывает метод AssignTo, если целевой объект не может скопировать свойства исходного объекта. Метод AssignTo, определенный в классе TPersistent, использует исключение EConvertError.

Например, используем следующий код для объектов A и B.

```
A.Assign(B);
```

Если объект A знает, как скопировать объект B, тогда он производит копирование. Если объект A не умеет копировать объект B, то используется версия Assign класса TPersistent, которая вызывается как

```
B.AssignTo(A);
```

Если B знает, как копировать A, то копирование завершается успешно, в противном случае генерируется исключение.

Procedure DefineProperties(Filer: TFiler); virtual; В производных от TPersistent классах метод DefineProperties переопределяется для адаптации к сохранению данных в потоках, подобных файлу формы. По умолчанию при записи объекта в поток переписываются значения всех экспортируемых свойств, а при считывании эти значения присваиваются соответствующим свойствам. В объектах также могут быть определены методы для чтения неэкспортируемых свойств, для чего переопределяется метод DefineProperties.

Function GetNamePath: string; dynamic; Метод GetNamePath предназначен только для внутреннего использования. Он определяет текст, который инспектор объектов отображает для имени редактируемого объекта. Так как метод GetNamePath представлен в классе TPersistent, то производные классы, подобные коллекциям, могут отображаться в инспекторе объектов. Нельзя непосредственно вызывать метод GetNamePath.

Для компонентов метод GetNamePath возвращает имя компонента. Для объектов типа TCollectionItem возвращается имя базового компонента, имя свойства и индекс, заключенные в скобки.

Класс TComponent

Класс TComponent является базовым классом для всех компонентов. Этот класс делает компоненты перманентными (постоянными), и они приобретают следующие возможности.

- Интегрируется в среду разработки. То есть способны отображаться в палитре компонентов и могут настраиваться в процессе проектирования
- Владеют другими компонентами. Например, если компонент A владеет компонентом B, то A способен ликвидировать B перед тем, как сам будет уничтожен.
- Сохраняются в потоках и файлах. Улучшены возможности сохранения, унаследованные от класса TPersistent.
- Поддерживают COM-объекты. Компоненты могут быть преобразованы в элементы управления ActiveX или COM-объекты с помощью мастеров. Компоненты могут служить оболочками для COM-объектов.

> Перманентными или постоянными считаются такие объекты, время жизни которых не ограничивается временем выполнения создавших и использующих их программ. То есть такие объекты могут сохраняться на физических носителях, в файле или базе данных.

В классе `TComponent` не реализованы возможности взаимодействия с пользователем и отображения на экране. Эти средства представлены в двух классах, являющихся непосредственными наследниками класса `TComponent`. Класс `TControl` из модуля `QControls` является базовым классом для визуальных компонентов в кроссплатформенных приложениях.

Класс `TControl` из модуля `Controls` является базовым классом для визуальных компонентов только в приложениях для Windows.

Модули `Controls` и другие, специфические для Windows модули, не поставляются со средой разработки для Linux.

Компоненты, которые отображаются на экране во время выполнения программы, часто называют *визуальными компонентами*. Остальные компоненты, которые не видны во время выполнения, называют *невизуальными компонентами*. Однако удобнее будет называть визуальные компоненты *элементами управления*, а *невизуальные компоненты* просто *компонентами*.

Не создавайте непосредственно объекты типа `TComponent`. Используйте `TComponent` как базовый класс при объявлении невизуальных компонентов, которые будут размещаться на палитре компонентов и использоваться при проектировании. Свойства и методы класса `TComponent` определяют базовое поведение производных классов, хотя можно и переопределять некоторые методы.

> Используйте слово *компонент* для общего определения компонентов или только для *невизуальных компонентов*. Для визуальных компонентов используйте словосочетание *элемент управления*.

Методы класса TComponent

Function _AddRef: Integer; stdcall; Метод `_AddRef` вызывается, когда приложение использует интерфейсы компонента. `_AddRef` является основной реализацией метода `AddRef` интерфейса `IInterface`.

Если компонент является оболочкой для COM-объекта, то `_AddRef` вызывает метод `AddRef` COM-объекта и возвращает результирующее значение количества ссылок.

Во всех других случаях `_AddRef` просто возвращает `-1` и не производит никаких действий. Это позволяет компонентам использовать интерфейсы, в которых не требуется счетчик ссылок. Для подсчета ссылок в унаследованных компонентах необходимо переопределить метод `_AddRef` для реализации счетчика.

Function _Release: Integer; stdcall; Метод `_Release` является базовой реализацией метода `Release` интерфейса `IInterface`.

Метод `_Release` возвращает результирующее значение счетчика ссылок для интерфейсов компонента.

Если компонент является оболочкой для COM-объекта, то `_Release` вызывает метод `Release` объекта COM и возвращает результирующее значение счетчика ссылок. Во всех других случаях метод `_Release` просто возвращает `-1` и не производит никаких действий. Это позволяет компонентам использовать интерфей-

сы, в которых не требуется счетчик ссылок. В унаследованных компонентах необходимо переопределить метод _Release для реализации счетчика ссылок.

Procedure BeforeDestruction; override; Метод BeforeDestruction вызывается автоматически сразу после вызова деструктора компонента и до выполнения деструктора. Не вызывайте этот метод явно в ваших приложениях.

Реализация метода BeforeDestruction в классе производит проверку вызова метода Destroying, и если этот метод не вызван, то BeforeDestruction вызывает метод Destroying. В производных классах, где переопределяется BeforeDestruction, необходимо сначала вызвать унаследованный метод для проверки вызова Destroying.

Procedure ChangeName(const NewName: TComponentName); Устанавливает закрытое внутреннее хранение значения для свойства Name, передаваемое через параметр NewName.

Не используйте метод ChangeName непосредственно, используйте свойство Name.

> Установки для свойств Name, SetName используют метод ChangeName для изменения имени компонента. Метод ChangeName не является виртуальным, не переопределяйте его.

Constructor Create(AOwner: TComponent); virtual; Конструктор распределяет память и создает надежную структуру экземпляра компонента. Все классы должны иметь метод Create для конструирования объекта. В классе TComponent метод Create переопределен для создания компонентов. Для того чтобы это произошло, выполните следующие действия.

1. Определите родительские отношения компонента и установите его владельца (свойство Owner), передаваемого через параметр AOwner.
2. Свойство ComponentStyle установите в состояние csInheritable, означающее, что компонент может быть унаследован производным от формы типом.

Нет необходимости явно создавать компоненты, добавляемые в форму. Такие компоненты создаются автоматически при запуске приложения и автоматически уничтожаются при его закрытии.

Для компонентов, создаваемых программно, т.е. для тех, которые не размещаются в форме во время проектирования, вызывайте метод Create и передавайте владельца компонента через параметр AOwner. Владелец предварительно ликвидирует все принадлежащие ему компоненты при своем уничтожении. Если у компонента нет владельца, то необходимо вызвать метод Free для уничтожения компонента.

> Если через параметр AOwner передается значение Self, то надо иметь четкое представление, на что ссылается Self. Если компонент создает другой компонент с помощью одного из своих методов, то Self ссылается на первый компонент, а не на создаваемый компонент.

> Конструктор класса TComponent является виртуальным для создания полиморфного поведения. Это необходимо для потоковых систем и при проектировании формы. Не забывайте использовать директиву override, когда объявляете конструктор Create для нового компонента.

Procedure DefineProperties(Filer: TFiler); override; Метод DefineProperties определяет способы хранения данных в потоковых системах, таких как файл формы.

В классе TComponent метод DefineProperties переопределяет метод класса TPersistent для получения "скрытых" свойств Top и Left. Это необходимо для компонентов, которые не являются элементами управления. Но эти свойства скрыты и не являются экспортируемыми, так как невизуальные компоненты не отображаются на экране во время выполнения программы.

Метод DefineProperties является виртуальным и может переопределяться в производных классах.

Не вызывайте непосредственно метод DefineProperties, он вызывается автоматически как часть потоковой системы компонента.

Destructor Destroy; override; Уничтожает компонент и все компоненты, которыми владеет уничтожаемый компонент.

Не вызывайте непосредственно метод Destroy, используйте метод Free, который проверяет значение ссылки на компонент.

Никогда явно не уничтожайте компонент в одном из его обработчиков событий или в компоненте, который владеет этим компонентом.

Для уничтожения формы вызывайте метод Release, который ожидает окончания всех событий, связанных с формой, и заканчивает выполнение перед уничтожением формы.

> Форма владеет всеми элементами управления и невизуальными компонентами, которые были размещены в ней во время проектирования. Когда форма закрывается, все эти компоненты автоматически уничтожаются. По умолчанию всеми формами владеет глобальный объект Application. Когда приложение заканчивает работу, уничтожается глобальный объект Application, который ликвидирует все формы. Для объектов, которые не являются компонентами или которые создаются без указания владельца, необходимо вызывать метод Free для уничтожения объекта, иначе не будет освобождена занимаемая ими память до окончания работы приложения.

Procedure DestroyComponents; Уничтожает все включенные компоненты. Метод DestroyComponents просматривает список включенных компонентов и последовательно уничтожает их. Нет необходимости непосредственно вызывать метод DestroyComponents, он вызывается автоматически при уничтожении компонента.

Procedure Destroying; Отмечает, что компонент и включенные в него компоненты готовы к уничтожению. Метод Destroying устанавливает флаг в состояние csDestroying в свойстве ComponentState и затем вызывает метод Destroying для каждого включенного компонента и устанавливает их флаги в состояние csDestroying. Если флаг уже находится в состоянии csDestroying, то метод Destroying ничего ни делает.

Нет необходимости непосредственно вызывать метод Destroying, он вызывается автоматически при уничтожении компонента.

Function ExecuteAction(Action: TBasicAction): Boolean; dynamic; Выполняет отдельную задачу. Когда пользователь запускает задачу, CLX выполняет ряд подготовительных действий. Сначала генерируется событие OnExecute из списка действий. Если в списке действий не обрабатывается событие OnExecute, выполнение передается методу ExecuteAction объекта Application, в котором

вызывается обработчик события OnActionExecute. Если и здесь нет возможности обработать событие, тогда вызывается метод ExecuteAction активного элемента управления.

В параметре Action определяется действие, которое должно быть выполнено. Метод ExecuteAction возвращает True, если задача отправлена на выполнение, и False, если компонент не может справиться с задачей. Если возвращается False для активного элемента управления, тогда CLX вызывает метод ExecuteAction активной формы. Если и в этом случае возвращается False, то перебираются все активные элементы управления в форме. Если и в этом случае будет возвращено False, то CLX повторяет процесс с главной формой, если она не является той же самой формой.

> Метод ExecuteAction вызывается только для тех компонентов и форм, которые являются видимыми.

В классе TComponent метод ExecuteAction проверяет, знает ли компонент, как выполнить задачу. Если да, то задача выполняется, и возвращается значение True. В противном случае возвращается False.

В производных классах метод ExecuteAction переопределяется для изменения функционального поведения, заданного по умолчанию.

Function FindComponent(const AName: string): TComponent; Проверяет, находится ли данный компонент в подчинении у другого компонента.

Метод FindComponent ищет компонент в массиве компонентов свойства Components по имени, передаваемому через параметр AName. Используйте FindComponent для определения вхождения компонента в состав определенного компонента. Имя компонента не чувствительно к регистру.

Procedure FreeNotification(AComponent: TComponent); Гарантирует уведомление компонента о том, что он должен быть уничтожен.

Используйте метод FreeNotification для регистрации компонента, передаваемого через параметр AComponent, как компонента, который необходимо автоматически уведомлять о том, что он должен быть уничтожен. Таким образом необходимо регистрировать только те компоненты, которые находятся в других формах или имеют другого хозяина. Например, компонент, определяемый параметром AComponent, находится в другой форме и используется для реализации отдельного свойства, поэтому необходимо вызвать метод FreeNotification для того, чтобы метод Notification уведомлял об уничтожении компонента.

Для компонентов, которые имеют одного и того же владельца, метод Notification вызывается автоматически, когда в приложении явно уничтожается компонент. Это уведомление не посылается за пределы компонента-владельца.

Procedure FreeOnRelease; Уничтожает ссылки интерфейса для компонентов, которые были созданы из классов COM. Метод FreeOnRelease вызывается, когда интерфейс, созданный компонентом, уничтожается. Нет необходимости вызывать метод непосредственно, так как он вызывается соответствующими методами интерфейса.

Function GetChildOwner: TComponent; dynamic; Возвращает владельца компонента.

Метод GetChildOwner используется средствами потоковой системы компонента, и необходимость вызывать его непосредственно в коде возникает крайне редко.

В классе TComponent метод GetChildOwner всегда возвращает nil, сигнализируя о том, что будет считан корневой компонент (обычно форма или модуль данных).

Владелец компонента способен уничтожить его.

Function GetChildParent: TComponent; dynamic; Возвращает родительский класс компонента. Если родительского класса нет, то возвращается владелец компонента.

Метод GetChildParent используется внутри потоковой системы компонента, и нет необходимости вызывать его непосредственно.

Реализация метода GetChildParent в классе TComponent возвращает значение Self. Если происходит возврат nil, то родительский класс является корневым компонентом (обычно это форма).

Procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic; Предусматривает интерфейс для методов, которые элементы управления используют для возврата производных компонентов.

Метод GetChildren реализован в классе TComponent для потоковых систем записи данных, которые загружают и сохраняют компоненты. В производных классах метод GetChildren переопределяется для выполнения обратного вызова, определенного в параметре Proc для каждого производного компонента. Таким образом производные компоненты могут использовать метод GetParentComponent.

Function GetIDsOfNames(const IID: TGUID; Names: Pointer; NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall; Метод GetIDsOfNames реализует интерфейс IDispatch. Для компонентов, которые поддерживают интерфейсы, вызывайте этот метод для интерфейсов, передавая соответствующие параметры. Возвращаемое значение может в дальнейшем использоваться для вызовов метода Invoke.

Function GetNamePath: string; override; Возвращает строку, используемую в инспекторе объектов.

Метод GetNamePath используется для получения текста, отображаемого в инспекторе объектов для имени редактируемого объекта. GetNamePath объявлен в классе TPersistent. В классе TComponent он переопределен для получения имени компонента. Не используйте метод GetNamePath непосредственно в коде.

Function GetOwner: TPersistent; override; Возвращает владельца компонента.

Метод GetOwner вызывается из метода GetNamePath для поиска владельца компонента. Методы GetNamePath и GetOwner представлены в классе TPersistent. Для класса Tcomponent метод GetOwner возвращает значение свойства Owner.

Function GetParentComponent: TComponent; dynamic; Возвращает родительский класс компонента.

Метод GetParentComponent используется в классе TComponent потоковыми системами, которые загружают и сохраняют компоненты CLX. Реализованный в классе TComponent метод GetParentComponent всегда возвращает nil. В производных классах метод GetParentComponent переопределяется для получения соответствующего родительского компонента.

> Не путайте родительские компоненты, которые отвечают за запись компонента в поток данных, и компонента-владельца, который отвечает за уничтожение компонента.

Function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HResult; stdcall; Извлекает информацию о типе объекта.

Метод `GetTypeInfo` реализует одноименный метод интерфейса `IDispatch`. Для компонентов, которые поддерживают интерфейсы, метод `GetTypeInfo` вызывает одноименный метод интерфейса, передавая соответствующие аргументы. Используйте возвращаемое значение для получения информации о типе интерфейса.

Function GetTypeInfoCount(out Count: Integer): HResult; stdcall; Метод `GetTypeInfoCount` реализует интерфейс `IDispatch` метода `GetTypeInfoCount`. Для компонентов, которые поддерживают интерфейсы, метод `GetTypeInfoCount` вызывает одноименный метод интерфейса, поддерживаемого компонентом. Параметр `Count` возвращает значение `1` или `0` в зависимости от того, есть ли информация о типе.

Function HasParent: Boolean; dynamic; Указывает на наличие родительского компонента.

Метод `HasParent` реализован в классе `TComponent` для потоковых систем загрузки и сохранения компонентов CLX. Для класса `TComponent` он возвращает `False`.

Возвращаемое значение говорит о том, может ли некоторый другой объект (родительский) записать данный объект в поток данных. Чаще всего это будет форма или панель, которые содержат элементы управления.

В любом производном компоненте, в котором метод `HasParent` возвращает `True`, должны быть также реализованы методы `GetParentComponent` и `SetParentComponent`.

Procedure InsertComponent(AComponent: TComponent); Устанавливает компонент как владельца другого компонента.

Метод `InsertComponent` добавляет компонент, передаваемый через параметр `AComponent` в конец массива компонентов свойства Components. Вставляемый компонент не должен иметь имени (не определено свойство Name) или имя должно быть уникальным.

Когда уничтожается владелец компонента, то предварительно автоматически уничтожается и сам компонент.

Компоненты автоматически вставляются и удаляются в момент проектирования при визуальном размещении. Используйте метод `InsertComponent`, если необходимо добавить компонент в массив компонентов владельца.

Function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer; Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall; Метод `Invoke` является стандартным механизмом для доступа к открытым свойствам и методам объекта автоматизации. Для объектов, которые заключаются в интерфейс `IDispatch` объекта автоматизации, метод `Invoke` вызывает одноименный метод интерфейса, поддерживаемого компонентом, передавая его через параметры.

Function IsImplementorOf(const I: IInterface): Boolean; Проверяет наличие определенного интерфейса в компоненте.

Используйте метод `IsImplementorOf` для проверки поддержки интерфейса, передаваемого через параметр `I` компонентом, даже при агрегации интерфейса с другими компонентами. Метод `IsImplementorOf` подобен методу `QueryInterface`, но он может работать со значениями `nil`, при этом не возвращая указатель на интерфейс.

Потоковые системы, которые загружают и сохраняют компоненты, используют IsImplementorOf для работы со значениями свойств, которые являются интерфейсами.

Procedure Loaded; virtual; Инициализирует компонент после считывания файла формы в память.

Не вызывайте непосредственно защищенный метод Loaded. Потоковые системы вызывают этот метод после загрузки формы из потока данных.

Когда потоковая система загружает форму или модуль данных из файла формы, сначала конструируется компонент формы с помощью вызова конструктора, затем считываются значения свойств из файла формы. После считывания свойств для всех компонентов потоковая система вызывает метод Loaded для каждого компонента в том порядке, в каком компоненты создавались. Это дает возможность компонентам инициализировать все данные, которые зависят от значений в других компонентах или других частях самого компонента.

> Все ссылки на родственные компоненты разрешаются во время вызова метода Loaded, который является первым "местом", где ссылки могут использоваться после считывания из потока.

Реализация метода Loaded в классе TComponent сбрасывает флажок csLoading в свойстве ComponentState, отмечая, что компонент уже загружен.

> Метод Loaded может вызываться несколько раз в производных формах. Он вызывается каждый раз при считывании формы из потока данных. Не распределяйте память в перегруженных методах Loaded без проверки того, что память уже распределена в предыдущем вызове.

Procedure Notification(AComponent: TComponent; Operation: TOperation); virtual; Рассылает уведомляющие сообщения всем включенным компонентам.

Не вызывайте метод Notification в разрабатываемом приложении. Notification вызывается автоматически, когда компонент, определяемый параметром AComponent, должен быть вставлен или удален, что определяется параметром Operation. По умолчанию компонент передает дальше уведомление тем компонентам, которыми владеет, если таковые имеются.

> Метод Notification не вызывается для компонентов, которые уничтожаются явно.

Procedure PaletteCreated; dynamic; Реакция на создание компонента из палитры компонент.

Метод PaletteCreated вызывается автоматически во время проектирования при использовании компонента из палитры компонентов. Разработчики компонентов могут переопределить этот метод для выполнения настроек, которые требуются при использовании компонента.

Реализация метода PaletteCreated в классе TComponent ничего не делает.

Function QueryInterface(const IID: TGUID; out Obj): HResult; virtual; stdcall; Возвращает ссылку на определенный интерфейс, если объект поддерживает этот интерфейс.

Метод QueryInterface проверяет, поддерживается ли интерфейс, определяемый параметром IID. Если да, то возвращается ссылка на интерфейс через параметр Obj. Если компонент не поддерживает интерфейс, то возвращается значение nil.

Для компонентов, которые являются оболочками для объектов COM, метод QueryInterface вызывает метод QueryInterface внутреннего объекта COM.

Procedure ReadState(Reader: TReader); virtual; Считывает данные компонента из потока.

Метод ReadState является частью последовательных вызовов, используемых потоковой системой для загрузки и сохранения компонентов CLX. В нем считываются значения всех экспортируемых свойств, сохраненных данных и входящих компонентов для объекта, переданного через параметр Reader. Не вызывайте метод ReadState непосредственно в коде.

Хотя метод ReadState является виртуальным, он почти никогда не переопределяется. В любом классе, где переопределен ReadState, необходимо вызвать унаследованный от класса TComponent метод ReadState.

Function ReferenceInterface(const I: IInterface; Operation: TOperation): Boolean; Создает или удаляет внутренние связи, которые определяют, как компонент будет уведомлен при уничтожении определенных интерфейсов.

Разработчики компонентов используют метод ReferenceInterface для того, чтобы убедиться, что свойства, значения которых являются интерфейсами, информированы о том, что объекты, реализующие интерфейс, будут уничтожены.

Параметр I является указателем на интерфейс. Параметр Operation определяет, должны ли уведомляющие связи, реализованные в интерфейсе, быть созданы (значение opInsert) или удалены (opRemove).

Метод ReferenceInterface возвращает True при успешном выполнении операции. Если возвращается значение False при значении параметра Operation равном opInsert, то указанный интерфейс не может быть сохранен как значение экспортируемого свойства.

> При возвращении значения False нет необходимости фиксировать ошибку, просто интерфейс не будет сохраняться потоковой системой. Например, ReferenceInterface возвращает False, когда определенный интерфейс реализует счет ссылок независимо от жизни компонента.

Procedure RemoveComponent(AComponent: TComponent); Удаляет заданный компонент из списка компонентов.

Компоненты автоматически удаляются и вставляются при визуальном манипулировании ими во время проектирования. Используйте RemoveComponent при программировании для удаления компонента, определяемого параметром AComponent, из списка компонентов хозяина.

Procedure RemoveFreeNotification(AComponent: TComponent); Блокирует уведомления об уничтожении, которые были установлены методом FreeNotification.

Метод RemoveFreeNotification удаляет компонент, определяемый параметром AComponent, из внутреннего списка объектов, которые должны быть уведомлены о предстоящем уничтожении компонента. Компоненты добавляются в этот список методом FreeNotification.

В большинстве приложений нет необходимости вызывать метод RemoveFreeNotification. Он используется в классе TComponent для выявления циклов, где два компонента уведомляют друг друга о предстоящем уничтожении.

Function SafeCallException(ExceptObject: TObject; ExceptAddr: Pointer): HResult; override; Обработчики исключений в методах, объявленных с использованием соглашений о вызовах.

Метод SafeCallException обрабатывает исключения в методах, которые используют соглашение о вызовах с директивой safecall. В некоторых классах, реализующих интерфейсы, переопределяют этот метод для обработки возможных ошибок. В классе TComponent вызывается реализация этого метода для интерфейсов, поддерживаемых компонентом. Если компонент не поддерживает интерфейсы, этот метод вызывает метод SafeCallException, унаследованный от класса TObject, который возвращает значение E_UNEXPECTED. Такое значение по умолчанию возвращается для классов, которые не поддерживают интерфейсов.

Procedure SetAncestor(Value: Boolean); Устанавливает или сбрасывает флажок csAncestor свойства ComponentState.

Метод SetAncestor используется потоковыми системами, которые загружают и сохраняют компоненты CLX. Не используйте метод SetAncestor непосредственно в коде.

Метод устанавливает или сбрасывает флажок csAncestor, который сигнализирует о том, представлен или нет компонент в родительской форме. Затем вызывается метод SetAncestor владеющего компонента для синхронизации свойств ComponentState.

Procedure SetChildOrder(Child: TComponent; Order: Integer); dynamic; Представляет интерфейс для метода по переупорядочению производных компонентов.

Метод SetChildOrder представлен в классе TComponent для потоковых систем, которые загружают и сохраняют компоненты CLX, но он там не реализован. В производных классах метод SetChildOrder переопределяется для изменения порядка, в котором появляются производные объекты при получении списка с помощью метода GetChildren.

Procedure SetDesigning(Value: Boolean; SetChildren: Boolean=True); Обеспечивает компоненту, использованному во время проектирования, установку флажка режима проектирования.

Метод SetDesigning используется внутренними средствами проектировщика форм, и нет необходимости его вызывать непосредственно в коде.

Метод SetDesigning устанавливает флажок csDesigning в свойстве ComponentState, если значением параметра Value является True. В противном случае флажок сбрасывается.

Если значением параметра SetChildren является True, тогда SetDesigning вызывает методы SetDesigning всех включенных компонентов для синхронизации свойств ComponentState.

Методы InsertComponent и RemoveComponent вызывают SetDesigning при вставке или удалении компонентов для проверки правильности установки флажка.

Procedure SetDesignInstance(Value: Boolean); Гарантирует установку флажка режима проектирования для компонентов, используемых при проектировании.

Метод SetDesignInstance предназначен для внутреннего использования проектировщиком форм. Не вызывайте его непосредственно. Метод SetDesignInstance устанавливает флажок csDesignInstance в свойстве ComponentState, если значение параметра Value равно True, в противном случае флажок сбрасывается.

Procedure SetInline(Value: Boolean); Устанавливает бит csInline свойства компонента ComponentState.

Метод SetInline предназначен для внутреннего использования и отмечает, может ли компонент использоваться в качестве базового (основного) компонента для проектировщика форм. Метод устанавливает флажок csInline для свойства ComponentState, если значение параметра Value равно True, в противном случае флажок сбрасывается.

Procedure SetName(const NewName: TComponentName); virtual; Устанавливает значение свойства Name.

Метод SetName является виртуальным и устанавливает значения свойства Name. Он вызывает метод ChangeName, который и выполняет всю работу.

Метод ChangeName не является виртуальным. Для изменения поведения свойства Name переопределите метод SetName.

> **Внимание!**
>
> Использование метода SetName для изменения имени компонента во время выполнения программы делает все ссылки на старые имена неопределенными. Любой последующий код, где встретится ссылка на старое имя, приведет к исключительной ситуации.

Procedure SetParentComponent(Value: TComponent); dynamic; Метод SetParentComponent представлен в классе TComponent для потоковых систем, загружающих и сохраняющих компоненты CLX. В производных классах метод SetParentComponent переопределяется, чтобы изменить значение, которое возвращает метод GetParentComponent для соответствующего значения параметра Value.

Procedure SetSubComponent(IsSubComponent: Boolean); Определяет, является ли компонент подкомпонентом.

Вызывайте метод SetSubComponent для определения того, является ли определенный компонент подкомпонентом. Подкомпонентом является компонент, владельцем которого является компонент, отличный от формы или модуля данных, в которых он находится. Если такой компонент вызывает метод SetSubComponentc с параметром IsSubComponent, имеющим значение True, его экспортируемые свойства не будут сохраняться в файле формы.

Параметр IsSubComponent говорит о том, что компонент является подкомпонентом (значение True) или не является (False).

Метод SetSubComponent можно вызывать во время проектирования в следующих случаях.

- Из конструктора компонента, который всегда выполняется как подкомпонент. В этом случае компонент вызывает собственный метод SetSubComponent из конструктора со значением параметра IsSubComponent равным True.
- Непосредственно после конструирования экземпляра подкомпонента. В этом случае владелец вызывает метод SetSubComponent компонента со значением параметра IsSubComponent равным True.

Function UpdateAction(Action: TBasicAction): Boolean; dynamic; Обновление действий компонента для отражения текущего состояния компонента.

Когда приложение простаивает, CLX делает серию вызовов для обновления свойств (достаточность, контроль и т.д.) для каждого действия, связанного с видимым элементом управления или меню. Сначала CLX генерирует событие OnUpdate

для списка действий. Если список действий не использует событие OnUpdate, то происходит обращение к методу UpdateAction объекта Application, который вызывает обработчик события OnActionUpdate. Если обработчик OnActionUpdate не производит обновление действия, то происходит обращение к обработчику OnUpdate. Если и в этом случае не происходит обновления, вызывается метод UpdateAction активного элемента управления.

Параметр Action определяет действие компонента, которое должно быть обновлено. Метод UpdateAction возвращает True, если действие компонента отражает состояние компонента, и возвращает False, если невозможно отреагировать на действие. Если UpdateAction возвращает False для активного компонента, CLX вызывает метод UpdateAction активной формы.

Не вызывайте UpdateAction непосредственно. Он вызывается автоматически при простаивании приложения.

Procedure Updated; dynamic; Сбрасывает флажок csUpdating в свойстве ComponentState после обновления.

Не вызывайте метод Updated непосредственно. Он используется внутренними средствами для сброса флажка csUpdating. Вызов Updated всегда следует за вызовом Updating, где флажок устанавливается.

Class procedure UpdateRegistry(Register: Boolean; const ClassID, ProgID: string); virtual; Обеспечивает интерфейс для методов, которые добавляют информацию о типе библиотеки и версии в регистр для компонентов, реализующих интерфейсы COM.

Используется только внутренними системами.

Procedure Updating; dynamic; Устанавливает флажок csUpdating в свойстве ComponentState.

Не вызывайте метод Updating непосредственно. Он используется внутренними системами для уведомления о том, что компонент обновляется. Непосредственно за Updating всегда следует вызов метода Updated.

Procedure ValidateContainer(AComponent: TComponent); dynamic; Определяет, может ли компонент быть вставлен в контейнер.

Метод ValidateContainer вызывается компонентом перед тем, как компонент вставляется в объект-контейнер. По умолчанию ValidateContainer вызывает метод ValidateInsert для компонента, передаваемого через параметр AComponent.

В производных компонентах метод ValidateContainer может быть переопределен для учета специфических ситуаций.

Procedure ValidateInsert(AComponent: TComponent); dynamic; Обеспечивает интерфейс для методов, которые проверяют дочерние компоненты перед тем, как они будут вставлены.

Метод ValidateInsert ничего не делает в классе TComponent. В производных классах его можно переопределять для контроля вставки дочерних компонентов. По умолчанию метод ValidateInsert разрешает всем объектам быть вставленными в компонент.

Если в компонент необходимо вставлять только объекты, отвечающие определенным требованиям, в производных классах необходимо переопределить метод ValidateInsert для фильтрации таких объектов. При запрете вставки должно генерироваться исключение в унаследованном методе.

Procedure ValidateRename(AComponent: TComponent; const CurName, NewName: string); virtual; Проверяет, что переименование компонента не создает конфликта имен.

Метод `ValidateRename` проверяет возможность изменение имени для компонента (параметр `AComponent`) одного из включенных компонентов, исходное имя которого передается через параметр `CurName`, на новое имя (параметр `NewName`). Если параметр `AComponent` имеет значение `nil`, или новое имя уже существует в списке компонентов свойства Components, метод `ValidateRename` генерирует исключительную ситуацию `EComponentError`.

Этот метод обычно используется внутренними средствами при модификации свойства Name, и нет необходимости вызывать его непосредственно.

Procedure WriteState(Writer: TWriter); virtual; Записывает параметры компонента в поток данных.

Метод `WriteState` является частью последовательных вызовов, используемых потоковыми системами для загрузки и сохранения компонентов CLX. `WriteState` является интерактивным методом, который записывает значения всех экспортируемых свойств и других сохраняемых данных в компонент, передаваемый через параметр `Writer`. Не вызывайте этот метод непосредственно в коде.

Хотя метод `WriteState` является виртуальным, он редко переопределяется. В любом производном классе, где переопределяется метод `WriteState`, он должен заканчиваться вызовом унаследованного от `TComponent` метода `WriteState`.

Свойства класса TComponent

Property ComObject: IUnknown; Используется для присваивания интерфейсов COM, реализуя ссылки на интерфейсы. Это свойство используется компонентами, которые поддерживают интерфейсы COM.

Если компонент не является оболочкой для COM, то считывание свойства ComObject заставит класс `TComponent` сгенерировать исключение `EComponentError`.

Property ComponentCount: Integer; Используется для получения или контроля количества компонентов, включенных в данный компонент.

> Свойство ComponentCount содержит такое же значение, как число элементов списка свойства Components и всегда на единицу больше, чем максимальное значение индекса, так как первый индекс всегда равняется `0`.

Property ComponentIndex: Integer; Используйте свойство ComponentIndex при работе со списком компонентов. Свойство ComponentIndex используется внутренними системами для итеративных процедур присваивания.

> Первый компонент в списке компонентов имеет значение `ComponentIndex` равное `0`, второй — `1` и т.д. Однако при использовании свойства ComponentIndex совместно со свойством ComponentCount не забывайте, что значение свойства ComponentCount всегда на единицу больше, чем максимальное значение свойства ComponentIndex.

Property Components[Index: Integer]: TComponent; Используйте свойство Components для доступа к любым включенным компонентам, например к компонентам, которыми владеет форма. Это свойство особенно удобно использовать, когда ссылка на включенный компонент производится по номеру, а не по имени. Оно также используется внутренними системами. Значения индекса изменяются от `0` до `ComponentIndex - 1`.

Property ComponentState: TComponentState; Свойство Components использует свойство ComponentState для определения состояния, при котором разрешены или запрещены определенные действия. Например, если для данного компонента определенное поведение возможно только во время выполнения, то во время проектирования производится проверка флажка `csDesigning`.

Свойство ComponentState используется только для чтения, и его флажки устанавливаются автоматически.

Property ComponentStyle: TComponentStyle; Свойство ComponentStyle регулирует взаимодействие компонента с потоковыми системами и инспектором объектов. Это свойство используется только для чтения. Обычно значение различных флажков определяется при описании компонента с помощью конструктора. Единственное исключение — это флажок `csSubComponent`, который можно устанавливать после вызова метода `SetSubComponent`.

Property DesignInfo: Longint; Свойство DesignInfo используется внутренними системами. Не применяйте его в приложениях.

Property Name: TComponentName; Используйте свойство Name для присваивания имени компоненту, отражающего его назначение. По умолчанию среда разработки присваивает имена в соответствии с типом компонента, например "`Button1`", "`Button2`" и т.д.

Ссылка на данный компонент в коде производится по имени.

> **Внимание!**
>
> Изменение имени в процессе выполнения делает все старые ссылки неопределенными. Если в последующем коде произойдет обращение по старому имени, то возникнет исключительная ситуация.

Property Owner: TComponent; Используйте свойство Owner для получения имени владельца компонента. Знание владельца необходимо по следующим причинам.

- Память для компонента будет освобождена после освобождения памяти владельца. Это означает, что когда форма уничтожается, то будут уничтожены и все входящие в нее компоненты.
- Владелец отвечает за загрузку и сохранение экспортируемых свойств всех включенных элементов управления.

По умолчанию форма владеет всеми компонентами, которые расположены в ней. Таким образом, когда приложение закрывается и память освобождается, память для всех форм и расположенных в них компонентов также освобождается. Когда форма загружается в память, она загружает все связанные с ней компоненты.

Владелец компонента определяется по значению параметра, передаваемого в конструктор при создании компонента. Для компонентов, помещаемых в форму во время разработки, форма автоматически становится их владельцем.

> **Внимание!**
>
> Если компонент имеет владельца, отличного от формы или модуля данных, он не будет сохраняться или загружаться, пока вы его не определите как подкомпонент. Для этого необходимо вызвать метод SetSubComponent.

Property Tag: Longint; Свойство Tag не имеет предопределенного значения. Оно введено для создания дополнительного удобства разработчикам. Его можно использовать для хранения дополнительных целочисленных значений или для приведения типов, имеющих 32 разряда, таких как ссылки или указатели.

Property VCLComObject: Pointer; Свойство VCLComObject используется только внутренними системами. Поэтому при доступе к интерфейсам для компонентов COM в CLX используйте свойство ComObject.

Структура компонентов

Как уже было сказано ранее, компоненты представляют собой классы языка Delphi, инкапсулирующие функции и поведение элементов, добавляемых разработчиком в приложение для придания ему необходимого поведения и свойств.

Рассмотренные выше базовые методы и свойства уже позволяют понять, как компоненты вписываются в технологию разработки программ в среде Delphi. Рассмотрим более подробно особенности построения компонентов.

Работа с потоками данных

Важной характеристикой компонентов является их способность работать с потоками данных, которая позволяет хранить компонент и значения относящихся к нему свойств в файле. При этом Delphi берет работу с потоками данных на себя, однако разработчикам компонентов иногда могут понадобиться возможности работы с потоками данных, выходящие за рамки автоматически предоставляемых Delphi. Создаваемый Delphi файл с расширением `.dfm` — не более чем файл ресурсов, содержащий информацию о потоках данных в форме и ее компонентах в виде ресурса RCDATA.

Рассмотрим результат сохранения простой программы, написанной для того, чтобы лучше представить работу потоковой системы.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    FData1: Integer;
    FStr1: string;
  published
    Button1: TButton;
    Memo1: TMemo;
    Label1: TLabel;
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
{ TForm1 }
procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
  Memo1.Clear;
  FData1 := 25;
  FStr1 := 'Строка для сохранения';
  Memo1.Lines.Add(IntToStr(FData1));
  Memo1.Lines.Add(FStr1);
end;
end.
```

Для такой программы данные будут сохранены в текстовом файле с расширением .dmf в следующем виде.

```
object Form1: TForm1
  Left = 606
  Top = 424
  Width = 507
  Height = 324
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Left = 16
    Top = 56
    Width = 32
    Height = 13
    Caption = 'Label1'
  end
  object Button1: TButton
    Left = 16
    Top = 16
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
  object Memo1: TMemo
    Left = 248
    Top = 16
    Width = 241
    Height = 265
    Lines.Strings = (
      'Memo1')
    TabOrder = 1
  end
end
```

Как видите, сохраняются все параметры самой формы и параметры объектов с экспортируемым интерфейсом (раздел published), но не сохраняются поля FData1 и FStr1. При необходимости нетрудно сделать так, чтобы сохранялись и они. Для этого необходимо переопределить метод DefineProperties класса TPersistent, который и отвечает за сохранение данных. Если просмотреть опи-

сание класса TPersistent, то видно, что этот метод виртуальный и поэтому допускает переопределение, что и сделано в следующей программе.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    FData1: Integer;
    FStr1: string;
    procedure ReadStrData(Reader: TReader);
    procedure WriteStrData(Writer: TWriter);
    procedure ReadIntData(Reader: TReader);
    procedure WriteIntData(Writer: TWriter);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  published
    Button1: TButton;
    Memo1: TMemo;
    Label1: TLabel;
  public
    constructor Create(AOwner: TComponent); override;
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
{ TForm1 }
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited;
  FData1 := 25;
  FStr1 := 'Строка для сохранения';
end;

procedure TForm1.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  Filer.DefineProperty('StringProp',ReadStrData,WriteStrData,True);
  Filer.DefineProperty('IntProp',ReadIntData,WriteIntData,True);
end;

procedure TForm1.ReadIntData(Reader: TReader);
begin
  FData1 := Reader.ReadInteger;
end;

procedure TForm1.ReadStrData(Reader: TReader);
begin
  FStr1 := Reader.ReadString;
end;
```

```
procedure TForm1.WriteIntData(Writer: TWriter);
begin
  Writer.WriteInteger(FData1);
end;

procedure TForm1.WriteStrData(Writer: TWriter);
begin
  Writer.WriteString(FStr1);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Lines.Add(IntToStr(FData1));
  Memo1.Lines.Add(FStr1);
end;
end.
```

Отношения владения

Компоненты могут владеть другими компонентами. Владелец компонента хранится в его свойстве Owner. При уничтожении компонента принадлежащие ему компоненты также уничтожаются, а занимаемая ими память освобождается. В качестве примера можно привести форму, которая обычно владеет всеми расположенными в ней компонентами. При помещении некоторого компонента в форму в окне конструктора форм она автоматически становится владельцем этого компонента. Если компонент создается динамически, то в его конструктор (метод `Create`) требуется передать параметр, указывающий владельца компонента. Это значение и будет присвоено свойству `Owner` вновь созданного компонента. В следующей строке кода показано, как передать неявную переменную `Self` в конструктор `TButton.Create`, в результате чего форма становится владельцем создаваемого компонента.

```
Button := TButton.Create(Self);
```

Когда форма закрывается, экземпляр класса `TButton`, на который ссылается компонент `Button`, также уничтожается. Это один "из краеугольных камней фундамента" VCL — форма определяет компоненты, которые подлежат уничтожению при закрытии в соответствии со списком свойства Components.

Можно также создать и "ничейный" компонент, без владельца, передав в конструктор компонента `Create` параметр `nil`. Но в этом случае можно считать, что сам разработчик является владельцем компонента, и именно разработчик должен побеспокоиться о последующем удалении такого компонента. Необходимо избегать создания "ничейных" компонентов за исключением тех редких случаев, когда компоненту просто невозможно назначить владельца.

Рассмотрим подробнее свойство Components. Это свойство представляет собой массив, содержащий список всех компонентов, принадлежащих данному компоненту. Простейшая программа, листинг которой приведен ниже, выводит информацию об именах всех компонентов, перечисленных в свойстве Components, а на рис. 1.2 показан результат работы программы.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
```

```
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Label1: TLabel;
    Memo1: TMemo;
    Button4: TButton;
    Label2: TLabel;
    procedure Button4Click(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Button4Click(Sender: TObject);
var
  i: integer;
begin
  for i:=0 to Form1.ComponentCount-1 do
    Memo1.Lines.Add(Form1.Components[i].Name);
end;
end.
```

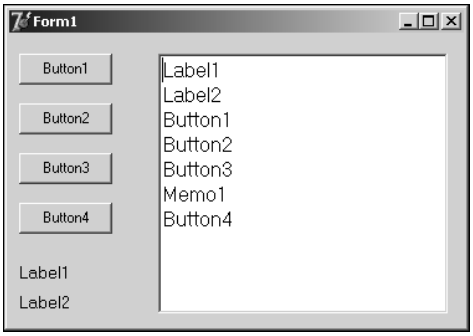

***Рис. 1.2.** Просмотр списка компонентов*

Отношения наследования

Не путайте понятия "владелец" и "родитель" компонента — это совершенно разные отношения. Некоторые компоненты могут быть родительскими для других компонентов. Обычно родительскими компонентами являются оконные компоненты, например потомки класса `TWinControl`. Родительские компоненты отвечают за функционирование методов отображения дочерних компонентов, а также за корректность этого отображения. Родительский компонент задается значением свойства Parent.

Родительский компонент не обязательно должен быть владельцем дочернего. Наличие разных родителя и владельца — совершенно нормальная ситуация для компонента. Как обычно, написание и анализ небольшой программы, подобно приведенной ниже, помогает лучше понять все тонкости терминологии. На рис. 1.3 показан результат работы программы.

Обычно в дереве объектов родительские отношения показываются соответствующими линиями, как показано на рис. 1.4 для приведенной программы.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Panel1: TPanel;
    Button2: TButton;
    GroupBox1: TGroupBox;
    Button3: TButton;
    LabeledEdit1: TLabeledEdit;
    Memo1: TMemo;
    Button4: TButton;
    procedure Button4Click(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Button4Click(Sender: TObject);
begin
  Memo1.Lines.Add(Button1.Name+':'+
    '  '+Button1.Parent.Name+'  '+Button1.Owner.Name);
  Memo1.Lines.Add(Button2.Name+':'+
    '  '+Button2.Parent.Name+'  '+Button2.Owner.Name);
  Memo1.Lines.Add(Button3.Name+':'+
    '  '+Button3.Parent.Name+'  '+Button3.Owner.Name);
  Memo1.Lines.Add(LabeledEdit1.Name+':'+
    '  '+LabeledEdit1.Parent.Name+'  '+LabeledEdit1.Owner.Name);
  Memo1.Lines.Add(GroupBox1.Name+':'+
    '  '+GroupBox1.Parent.Name+'  '+GroupBox1.Owner.Name);
end;
end.
```

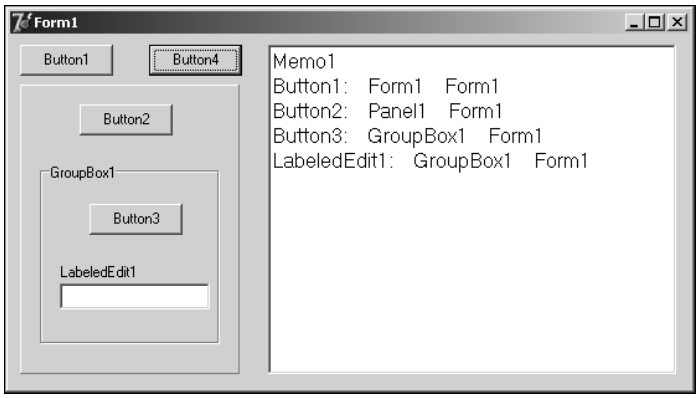

Рис. 1.3. *Демонстрация отношения компонентов*

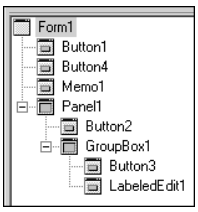

***Рис. 1.4.** Фрагмент дерева объектов*

Компонент Form

Хотя компонент `Form` (Форма) и не представлен на палитре компонентов, он используется чаще всего. Форма представляют собой видимое окно Windows, или элемент управления и используется практически в любом приложении.

С создания формы начинается конструирование приложения. В форме размещаются визуальные компоненты, образующие интерфейсную часть приложения, и системные, или невизуальные компоненты. Таким образом, в системе Delphi форма является контейнером для всех других компонентов. Конечно, возможно создание приложений и без окон, однако большинство приложений все же имеет отображаемое на экране окно, содержащее его интерфейсную часть.

Приложение может иметь несколько форм, одна их которых считается главной и при запуске программы отображается первой. При закрытии главной формы приложения прекращается работа всего приложения, при этом также закрываются и все другие окна приложения. В начале работы над новым проектом Delphi по умолчанию делает главной первую форму (с названием по умолчанию `Form1`). В файле проекта (расширение `.dpr`) эта форма создается первой, например:

```
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.CreateForm(TForm2, Form2);
Application.Run;
```

Программно можно сделать главной любую форму приложения, поставив метод `CreateForm` для этой формы первым. Например, задание формы `Form2` в качестве главной реализуется так:

```
Application.Initialize;
Application.CreateForm(TForm2, Form2);
Application.CreateForm(TForm1, Form1);
Application.Run;
```

При конструировании приложения более правильным будет указать главную форму в окне параметров проекта, вызываемом командой Project⇨Options... (Проект⇨Параметры). Главная форма выбирается в раскрывающемся списке Main Form на вкладке Form, после чего Delphi автоматически вносит соответствующие изменения в файл проекта.

Стандартная форма представляет собой прямоугольное окно с рамкой (рис. 1.5). Большинство окон содержит область заголовка, в которой расположены пиктограмма заголовка, заголовок и ряд кнопок, позволяющие свертывать, развертывать (восстанавливать) и закрывать окно, вызывать окно подсказки. Во многих формах отображаются также строка главного меню (под областью заголовка) и строка состояния (обычно в нижней части окна), для чего необходимо установить соответствующие компоненты. При необходимости в форме могут автоматически появляться полосы прокрутки, предназначенные для просмотра со-

держимого окна. Остальная часть пространства окна называется *клиентской областью*. В ней можно размещать элементы управления, выводить текст и графику, манипулировать дочерними окнами.

***Рис. 1.5**. Вид стандартной формы*

Форма может быть *модальной* и *немодальной*. Немодальная форма, оставшись открытой, позволяет переключиться в другую форму приложения. Модальная форма требует обязательного закрытия перед обращением к любой другой форме приложения.

Несложные формы, которые отображают различные сообщения и требуют от пользователя ввода какой-либо информации, часто называют *диалоговыми*. В свою очередь, диалоговые формы также могут быть модальными или немодальными.

В Windows имеется два основных типа приложений: однодокументные (SDI) и многодокументные (MDI). Однодокументные приложения состоят из одной или нескольких независимых друг от друга форм. В SDI-приложении ни одно окно на экране визуально не содержит в себе другие окна, поэтому иногда не ясно, какое из них является главным (родительским) окном приложения. В многодокументном приложении главное окно содержит дочерние окна, размещаемые в его пределах.

Свойства формы

Как и любой другой визуальный компонент, форма имеет свойства, методы и события, общие для всех визуальных компонентов. Однако форма имеет и свои специфические, определяемые ее особым значением, свойства, методы и события. Часть их характеризует форму как главный объект приложения, скажем свойство BorderIcons, другая часть присуща форме как контейнеру других компонентов, например свойства AutoScroll и ActiveControl.

В нижеприведенной таблице представлены все доступные для редактирования во время проектирования свойства формы.

Свойства класса Tform

Свойство	*Назначение*
Action	Связывает с формой объект типа `TBasicAction`
ActiveControl	Устанавливает фокус для элемента в форме
Align	Изменяет положение элементов в форме

Свойства класса Tform

Свойство	*Назначение*
AlphaBlend, AlphaBlendValue	Определяют прозрачность формы
Anchors	Определяет угол формы для привязки к координатам
AutoShroll	Если `True`, то полосы прокрутки появляются только при необходимости
AutoSize	Если `True`, то границы могут изменяться автоматически при изменении содержимого
BiDiMode	Двунаправленный режим для порядка считывания. Связано с локализацией программы
Active	Содержит значение `True`, если форма имеет фокус ввода
BorderIcons	Определяет пиктограмму в заголовке окна
BorderStyle	Вид границ формы
BorderWidth	Ширина рамки
Caption	Название формы, помещаемое в заголовке
Canvas	Область рисования формы
ClientHeight, ClientWidth	Размеры клиентской части формы (без заголовка)
Color	Цвет формы
Constraints	Ограничители, устанавливающие пределы автоматического изменения размеров формы
Ctl3D	Вид формы — объемный (3D) или нет
Cursor	Определяет вид курсора при наведении указателя мыши на форму
DefaultMonitor	Определяет монитор, в котором отображается форма
DockSite	Содержит значение `True`, если к форме разрешено “пристыковываться” другим окнам
DropKind, DragMode	Определяют возможности формы при операциях перетаскивания элементов
Enabled	Определяет реакцию формы на события мыши, клавиатуры и таймеров
Font	Установка шрифтов для формы
FormStyle	Стиль формы
Heigth	Высота формы с заголовком и рамкой
HelpContex	Используется для организации справочника
HelpFile	Название файла справки для формы
HelpKeyword	Ключевое слово для справочника
HelpType	Используется для организации справочника
Hint	Содержит текст этикетки, появляющейся при наведении на форму указателя мыши

Свойства класса Tform

Свойство	*Назначение*
HorzShrollBar	Свойства горизонтальной полосы прокрутки
Icon	Пиктограмма, обозначающая форму, когда она свернута
KeyPreview	Содержит значение `True`, если форма должна получать информацию о нажатых клавишах раньше, чем расположенные в ней объекты
Left	Координата угла привязки
Menu	Ссылка на главное меню формы (`Tmenu`)
ModalResult	Значение, возвращаемое формой, если она работает как модальное диалоговое окно
Name	Идентификатор (имя) формы для обращения к ней в программе
OldCreateOrder	Определяет момент выполнения событий `OnCreate` и `OnDestroy` относительно конструктора и деструктора
ParentBiDiMode	Использование режима, установленного в базовом классе. Применяется при локализации
ParentFont	Использование режима, установленного в базовом классе
PixelsPerInch	Число пикселей на дюйм. Применяется для настройки размера формы в зависимости от экранного разрешения
Position	Положение формы на экране в момент ее открытия в программе
PrintScale	Масштабирование формы при выводе на печать
Scaled	Содержит значение `True`, если размер формы будет подгоняться в соответствии со значением свойства `PixelsPerlnch`
ScreenSnap	Разрешение на стыковку с границей экрана
ShowHints	Разрешение на отображение подсказки (этикетки)
SnapBuffer	Установка зоны в пикселях для стыковки с границей экрана
Tag	Связывает определенное разработчиком числовое значение с формой
Top	Координата угла привязки
TransparentColor	Разрешает подсветку при установленном режиме прозрачности
TransparentColorValue	Определяет цвет подсветки
UseDockManager	Разрешение режима стыковки при перетаскивании
VertShrollBar	Свойства вертикальной полосы прокрутки
Visible	Содержит значение `True`, если форма будет видима во время работы программы
Wigth	Ширина формы с рамкой
WindowState	Состояние формы (свернута, развернута, нормальный размер)

События, поддерживаемые классом TForm

Событие	*Условия генерации*
OnActivate	При активизации формы
OnCanResize	При изменении размеров формы
OnClick	При щелчке мыши на форме
OnClose	При закрытии формы
OnCloseQuery	При запросе на закрытие формы
OnConstrainedResize	При выходе за пределы, установленные в ограничителях
OnCreate	При создании формы
OnDblClick	При двойном щелчке на форме
OnContexPopup	При вызове контекстного меню
OnDeactivate	При потере фокуса ввода
OnDestroy	При уничтожении формы
OnDockDrop	При стыковке с другим окном
OnDropOver	При расстыковке с другим окном
OnDragDrop	При перетаскивании объекта в пределы формы
OnDragOver	При перетаскивании объекта за пределы формы
OnEndDock	При расстыковке окон
OnGetSiteInfo	При стыковке окон
OnHelp	Форма получила запрос на выдачу справочной информации
OnHide	Форма стала невидимой (значение свойства Visible установлено равным False)
OnKeyDown	При нажатии клавиши клавиатуры и наличии фокуса
OnKeyPressed	При нажатии клавиши клавиатуры
OnKeyUp	При отпускании клавиши клавиатуры
OnMouseDown	При щелчке на кнопке мыши и перемещении мыши вниз окна
OnMouseMove	При щелчке на кнопке мыши и перемещении мыши над окном
OnMouseUp	При щелчке на кнопке мыши и перемещении мыши вверх окна
OnMouseWheel	При прокрутке ролика мыши
OnMouseWheelDown	При прокрутке ролика мыши вниз
OnMouseWheelUp	При прокрутке ролика мыши вверх
OnPaint	При перерисовке формы
OnResize	При изменении размеров окна
OnShortCut	Пользователь нажал комбинацию клавиш, которая пока не обработана
OnShow	Форма стала видимой (значение свойства Visible стало True)

События, поддерживаемые классом TForm

Событие	*Условия генерации*
OnStartDock	При перемещении элемента управления в режиме dcDock для DragKind
OnUpDock	При расстыковке элемента управления, подключенного средствами окна

Рассмотрим подробнее некоторые свойства и события. Для создания экземпляров форм служит конструктор Create. Сам класс формы обычно предварительно описывается при конструировании приложения, и для формы уже существуют файлы формы (.dfm) и программного модуля (.pas). Например, в процедуре:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
// Форма создается, но не отображается на экране
  Form2 := TForm2.Create(Application);
  Form2.Caption:='Новая форма';
end;
```

создается форма Form2, принадлежащая объекту Application и имеющая заголовок "Новая форма".

При создании и использовании формы генерируются следующие события типа TNotifyEvent, указанные в порядке их возникновения:

- OnCreate;
- OnShow;
- OnResize;
- OnActivate;
- OnPaint.

Событие OnCreate генерируется только один раз — при создании формы, остальные же события происходят при каждом отображении, активизации и каждой перерисовке формы соответственно.

В обработчик события OnCreate обычно включается код, устанавливающий начальные значения свойств формы, а также ее управляющих элементов, т.е. выполняющий начальную инициализацию формы в дополнение к установленным на этапе разработки приложения параметрам. Кроме того, в обработчик включаются дополнительные операции, которые должны выполняться однократно при создании формы, например чтение из файла некоторой информации и загрузка ее в список.

Например, в форме можно разместить компонент TPageControl с десятью вкладками.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to 9 do
    with TTabSheet.Create(Self) do
    begin
      PageControl := PageControl1;
      Caption := 'TabSheet #' + IntToStr(i);
    end;
end;
```

Из всех созданных форм Delphi при выполнении приложения автоматически делает видимой главную форму, для этого для свойства Visible этой формы устанавливается значение True. Для остальных форм значение данного свойства по умолчанию равно False, и после запуска приложения они на экране не отображаются. Если формы создаются вручную, то их отображение и сокрытие в процессе работы приложения регулируются программистом через свойство Visible. Даже если форма невидима, ее компонентами можно управлять, например, из других форм. Небольшая программа с двумя формами и расположенными в первой форме кнопками отлично демонстрирует эти возможности.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, Unit2, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Visible := True;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form2.Visible := False;
end;
end.
```

Щелчок на кнопках Button1 и Button2, расположенных в форме Form1, приводит, соответственно, к отображению и сокрытию формы Form2.

> Если одна форма выполняет какие-либо действия с другой формой, то в списке uses модуля первой формы должна быть ссылка на модуль второй формы.

Управлять видимостью форм на экране можно также с помощью методов Show и Hide. Процедура Show отображает форму в немодальном режиме, при этом свойству Visible придается значение True, а сама форма переводится на передний план. Процедура Hide скрывает форму, устанавливая для ее свойства Visible значение False.

Если окно видимо, то вызов метода Show переводит форму на передний план и передает ей фокус ввода. Несколько изменим использованные ранее процедуры обработки щелчка на кнопках.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form2.Hide;
end;
```

Щелчок на кнопках приводит к отображению на экране Form2 и удалению ее с экрана.

В момент отображения формы на экране ее свойство Visible принимает значение True, и возникает событие OnShow. Соответственно при сокрытии формы свойство Visible принимает значение False, и генерируется событие OnHide.

При получении формой фокуса ввода, например при щелчке на кнопке мыши в области формы, происходит ее активизация, и возникает событие OnActivate, а при потере фокуса — событие OnDeActivate.

Событие OnPaint генерируется при необходимости перерисовки формы, например при активизации формы, если до этого часть ее была закрыта другими окнами.

Для закрытия формы используется метод Close, который, если это возможно, удаляет ее с экрана. В случае закрытия главной формы прекращается работа всего приложения. Форма делается невидимой, но не уничтожается. Процедура Close не уничтожает созданный экземпляр формы, и форма может быть снова вызвана на экран с помощью методов Show или ShowModal.

Уничтожение формы происходит с помощью методов Release, Free или Destroy, после чего работа с этой формой становится невозможна, и при попытке обратиться к ней или ее компонентам будет сгенерирована исключительная ситуация. При закрытии и уничтожении формы генерируются следующие события, указанные в порядке их возникновения:

- OnCloseQuery;
- OnClose;
- OnDeActivate;
- OnHide;
- OnDestroy.

Событие OnCloseQuery типа TCloseQueryEvent возникает в ответ на попытку закрытия формы. Обработчик события получает логическую переменную-признак CanClose, определяющую, может ли быть закрыта данная форма. По умолчанию эта переменная имеет значение True, и форму можно закрыть. Если передать параметру CanClose значение False, то форма остается открытой. Такую возможность стоит использовать, например, для подтверждения закрытия окна или проверки, сохранена ли редактируемая информация на диске. Обработчик события OnCloseQuery вызывается всегда, независимо от способа закрытия формы.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form2.Close;
end;

procedure TForm2.FormCloseQuery(Sender: TObject;
    var CanClose: Boolean);
begin
```

```
  CanClose := MessageDlg('Вы хотите закрыть форму?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes;
end;
```

Здесь при закрытии формы `Form2` выдается запрос на подтверждение операции закрытия, который представляет собой модальное диалоговое окно с текстом и двумя кнопками — Yes и No. Щелчок на кнопке Yes вызывает закрытие формы, при щелчке на кнопке No закрытия формы не происходит.

Событие `OnClose` типа `TCloseEvent` возникает непосредственно перед закрытием формы. Обычно оно используется для изменения стандартного поведения формы при закрытии. Для этого обработчику события передается переменная `Action` типа `TCloseAction`, которая может принимать следующие значения:

- `caNone` — форму закрыть нельзя;
- `caHide` — форма становится невидимой;
- `caFree` — форма уничтожается, а связанная с ней память освобождается;
- `caMinimize` — окно формы минимизируется (значение по умолчанию для MDI-форм).

При закрытии окна методом `Close` переменная `Action` по умолчанию получает значение `caHide`, и форма становится невидимой. При уничтожении формы, например методом `Destroy` или `Free`, переменная `Action` по умолчанию получает значение `caFree`, и форма уничтожается.

Событие `OnClose` возникает при закрытии формы щелчком мыши на кнопке закрытия системного меню или при вызове метода `Close`. Когда закрывается главная форма приложения, все остальные окна закрываются без вызова события `OnClose`. Например, в нижеприведенной программе при закрытии формы `Form2` стандартным образом, т.е. щелчком на расположенной в заголовке окна кнопке закрытия, проверяется признак модификации содержимого редактора `Memo1`. Если информация в `Memo1` была изменена, то форма не закрывается.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Unit2;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
```

```
  Form2.Hide;
end;
end.


unit Unit2;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm2 = class(TForm)
    Memo1: TMemo;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;
var
  Form2: TForm2;

implementation
{$R *.dfm}
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if Memo1.Modified then Action:=caNone else Action:=caHide;
end;
end.
```

Событие `onDestroy` типа `TNotifyEvent` возникает непосредственно перед уничтожением формы и обычно используется для освобождения ресурсов.

При каждом изменении размеров формы в процессе выполнения приложения возникает событие `onResize` типа `TNotifyEvent`. В обработчике этого события может размещаться код, например, выполняющий изменение положения и размеров управляющих элементов окна, не имеющих свойства `Align`.

Проведем небольшой эксперимент. Создадим форму, и для нее определим обработчик события `OnReside`.

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Memo1.Left := 10;
  Memo1.Top := 10;
  Memo1.Height := Form1.ClientHeight - 20;
  Memo1.Width := Form1.ClientWidth - 120;
  Button1.Left := Form1.ClientWidth - 90;
  Button1.Top := Form1.ClientHeight - 35;
  Button2.Left := Button1.Left;
  Button2.Top := Button1.Top - 30;
end;
```

В форме `Form1` находятся три компонента: область ввода `Memo1` и кнопки `Button1` и `Button2`. Расположение этих компонентов показано на рис. 1.6. Размеры этих компонентов привязаны к размерам клиентской области формы и к друг другу, поэтому как бы вы не изменяли размеры формы, положение компонентов относительно формы и друг друга будет оставаться симметричным.

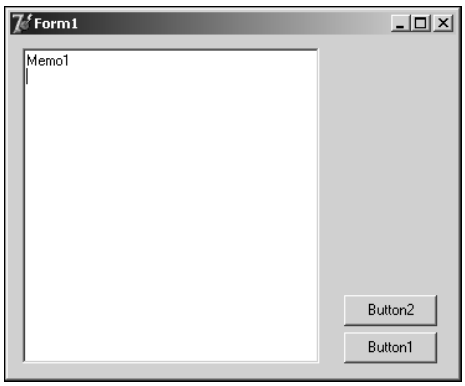

***Рис. 1.6.** Пример формы с взаимозависимым расположением компонентов*

Стиль формы определяется свойством FormStyle типа `TFormStyle`, принимающим следующие значения:

- `fsNormal` — стандартный стиль, используемый для большинства окон, в том числе и диалоговых;
- `fsMDIChild` — дочерняя форма в многодокументном приложении;
- `fsMDIForm` — родительская форма в многодокументном приложении;
- `fsStayOnTop` — форма, которая после запуска всегда отображается поверх других окон (обычно используется при выводе системной информации или информационной панели программы).

Форма может изменять стиль динамически — в процессе выполнения программы, например при выборе пункта меню. При изменении формой стиля возникает событие `OnShow`.

Ниже приведен пример динамического изменения стиля формы.

```
procedure TForml.mnuTopClick(Sender: TObject);
begin
  mnuTop.Checked := not mnuTop.Checked;
  if mnuTop.Checked then
    Forml.FormStyle:=fsStayOnTop
  else Forml.FormStyle: =fsNormal;
end;
```

При выборе пункта меню `mnuTop` форма переключает свой стиль в соответствии со значениями `fsNormal` и `fsStayOnTop`. Смена стиля отображается графически галочкой в заголовке этого пункта меню.

Каждая форма имеет рамку. Вид и поведение рамки определяет свойство BorderStyle типа `TFormBorderStyie`. Оно может принимать следующие значения:

- `bsDialog` — диалоговая форма;
- `bsSingle` — форма с неизменяемыми размерами;
- `bsNone` — форма не имеет видимой рамки и заголовка и не может изменять свои размеры (часто используется для заставок);

- bsSizeable — обычная форма с изменяемыми размерами (по умолчанию) имеет строку заголовка и может содержать любой набор кнопок;
- bsToolWindow — форма панели инструментов;
- bsSizeTooiWin — форма панели инструментов с изменяемыми размерами.

Невозможность изменения размеров форм некоторых стилей относится только к пользователю — нельзя с помощью мыши передвинуть границу формы в ту или другую сторону. Программно при выполнении приложения для формы любого стиля можно устанавливать любые допустимые размеры окна, а также изменять их.

В области заголовка могут отображаться четыре вида кнопок. Реализуемый набор кнопок определяет свойство BorderIcons типа TBorderIcons, которое может принимать комбинации следующих значений:

- biSystemMenu — окно содержит кнопки системного меню;
- biMinimize — окно содержит кнопку минимизации (свертывания);
- biMaximize — окно содержит кнопку максимизации (восстановления);
- biHelp — окно содержит кнопку справки, которая отображает вопросительный знак и вызывает контекстно-зависимую справку.

Системное меню представляет собой набор общих для всех окон Windows команд, например Свернуть или Закрыть. При наличии у окна системного меню слева в области заголовка отображается пиктограмма приложения, при щелчке на которой и появляются команды этого меню, а в области заголовка справа имеется кнопка закрытия формы.

Различные значения свойства BorderIcons не являются независимыми друг от друга. Так, если отсутствует системное меню, то ни одна кнопка не отображается. Возможность появления кнопок также зависит от стиля формы. Например, отображение кнопок максимизации и минимизации возможно только для обычной формы и формы панели инструментов с изменяемыми размерами.

> Обычно стиль формы и набор кнопок заголовка задаются на этапе разработки приложения через инспектор объектов. При этом на проектируемой форме всегда видны обычная рамка и три кнопки (минимизации, максимизации и закрытия формы), независимо от значения свойств FormStyle и BorderIcons. Заданные стиль формы и набор кнопок становятся видимыми при выполнении программы.

Обычно форму перетаскивают мышью, курсор которой устанавливается в любом месте области заголовка. При необходимости можно переместить форму, поместив курсор на ее клиентскую область, для чего требуется описать соответствующие операции программно. Одним из способов является перехват системного сообщения WM_NCHitTest. Для этого создается процедура FormMove, в которой анализируется, в каком месте формы находится указатель мыши при щелчке на кнопке. Код местоположения указателя мыши содержится в поле Result системного сообщения типа TMessage. Если значение Result равно единице, что соответствует щелчку в клиентской области, то полю Result присваивается новое значение, равное двум, имитирующее нахождение указателя мыши в области заголовка. В процедуре FormMove первый оператор inherited осуществляет вызов предопределенного обработчика перехватываемого события.

Для указания Delphi, что процедура FormMove является обработчиком события WM_NCHitTest, при ее описании в классе формы TForm1 используется синтаксис, включающий ключевое слово message. Как обработчик системного сообщения, процедура содержит один параметр типа TMessage.

Ниже приводится код модуля формы Form1, которую можно перемещать мышью, поместив курсор как в область заголовка, так и в клиентскую область.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
  public
    procedure MoveForm(var Msg: TMessage); message WM_NCHitTest;
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
{ TForm1 }
procedure TForm1.MoveForm(var Msg: TMessage);
begin
  inherited;
  if Msg.Result=1 then Msg.Result:=2;
end;
end.
```

Каждая форма отображает в левой стороне области заголовка свою пиктограмму, определяемую свойством Icon типа TIcon. Если форма не является главной в приложении, то эта пиктограмма отображается при минимизации формы. Для любой формы свойство Icon можно задать с помощью инспектора объектов или динамически (при выполнении приложения). Если пиктограмма не задана, то форма использует пиктограмму, указанную в свойстве Icon объекта Application. Последняя выводится также при минимизации и отображении в панели задач Windows значка главной формы приложения.

Размещение и размер формы при отображении определяет свойство Position типа TPosition. Оно может принимать значения, перечисленные ниже.

- poDesigned — форма отображается в той позиции и с теми размерами, которые были установлены при ее конструировании (значение по умолчанию). Положение и размеры формы определяются свойствами Left, Top, Width и Height. Если приложение запускается на мониторе с более низким разрешением, чем тот, на котором оно разрабатывалось, часть формы может выйти за пределы экрана.
- poScreenCenter — форма выводится в центре экрана, ее высота и ширина (свойства Height и Width) не изменяются.
- poDetauit — Windows автоматически определяет начальную позицию и размеры формы, при этом программист не имеет возможности контроли-

ровать эти параметры, поэтому данное значение не допускается для форм многодокументных приложений.

- `poDefaultPosOnly` — Windows определяет начальную позицию формы, ее размеры не изменяются.
- `poDefauitsizeOnly` — Windows определяет начальные ширину и высоту формы и помещает форму в позицию, определенную при разработке.
- `poDesktopCenter` — форма выводится в центре экрана, ее высота и ширина не изменяются.
- `poMainFormCenter` — форма выводится в центре главной формы приложения, ее высота и ширина не изменяются; это значение используется для вторичных форм если применять его для главной формы оно действует как значение `poScreenCenter`.
- `poOwnerFormCenter` — форма выводится в центре формы, которая является ее владельцем, высота и ширина формы не изменяются; если для формы не указан владелец (свойство Owner), то данное значение аналогично значению `poMainFormCenter`.

Приложение может запоминать расположение и размеры форм и при последующем выполнении правильно отображать формы на экране. Для этого программист должен записать соответствующие данные в файл инициализации приложения или в системный реестр Windows, а при последующем выполнении приложения считать эти данные и установить их для форм.

Свойства DockSite, DragKind, UseDockManager используются для получения эффекта объединения форм во время выполнения программы. Форма приобретает возможность “стыковаться” с другой формой и получается окно, объединяющее две формы, которым можно управлять. На рис. 1.7–1.9 показаны этапы стыковки окон. Причем в последнем случае форма `Form2` занимает все пространство панели `Panel1`. Двойной щелчок мыши на полосках в верхней части приведет к разъединению окон.

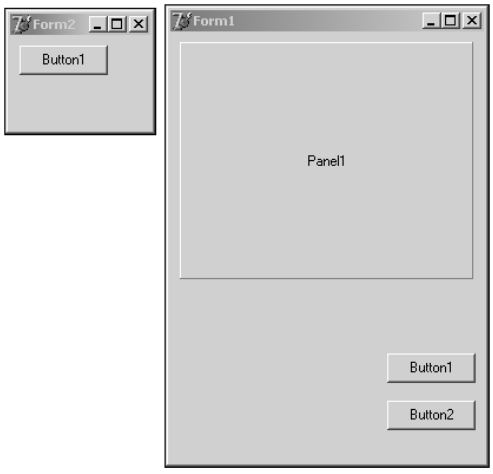

***Рис. 1.7.** Исходные формы*

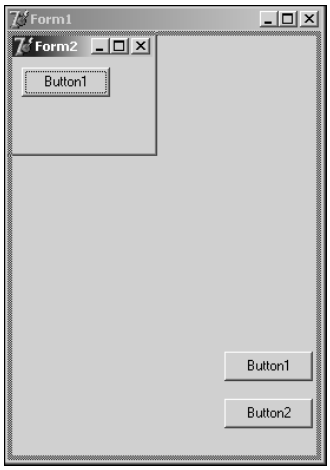

Рис. 1.8. *Возникающая рамка показывает возможное положение*

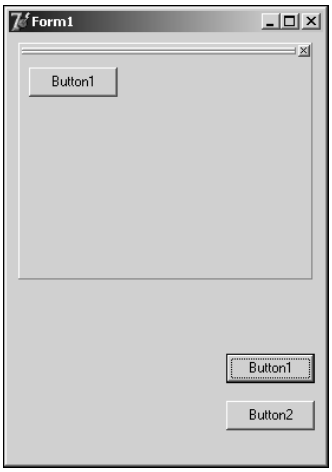

Рис. 1.9. *Форма* `Form2` *занимает место панели* `Panel1`

Свойства ScreenSnap и SnapBuffer обеспечивают "прилипание" окна формы к краю экрана. Величина зазора в пикселях, с которого это происходит, указывается в свойстве SnapBuffer, а возможность "прилипания" возникает при установке свойства ScreenSnap в состояние `True`.

Свойство Active типа `Boolean` позволяет определить активность формы. В любой момент времени активной может быть одна форма, при этом ее заголовок выделяется цветом, чаще всего синим. Если свойство Active имеет значение `True`, то форма активна (находится в фокусе ввода), если `False` — то неактивна. Это свойство доступно для чтения во время выполнения программы.

Свойство WindowState типа `TWindowState` определяет состояние отображения формы и может принимать одно из трех значений:

- `wsNormal` — обычное состояние (по умолчанию);
- `wsMinimized` — минимизация;
- `wsMaximized` — максимизация.

Форма, для которой изменяется состояние отображения на экране, предварительно должна быть создана методами `CreateForm` или `Create`. Если форма не создана, то при обращении к ней будет сгенерирована исключительная ситуация, несмотря на то что переменная формы объявлена в модуле. Если форма создана, но не отображается на экране, то изменения ее состояния (свойство WindowState) происходят, однако пользователь не видит этого до тех пор, пока форма не будет отображена на экране.

Будучи контейнером, форма содержит в себе другие управляющие элементы. Оконные элементы управления (потомки класса `TWinControl`) могут получать фокус ввода. Свойство ActiveControl типа `TWinControl` определяет, какой элемент формы находится в фокусе. Для выбора элемента, находящегося в фокусе ввода (активного элемента), можно придать этому свойству нужное значение в процессе выполнения программы.

```
Form1.ActiveControl := Edit2;
```

Аналогичную операцию выполняет метод SetFocus, который устанавливает фокус ввода на оконный элемент управления.

```
Edit2.SetFocus;
```

В случае когда размеры окна недостаточны для отображения всех содержащихся в форме интерфейсных компонентов, у формы могут появляться полосы прокрутки. Свойство AutoScroil типа Boolean определяет, появляются ли они автоматически. Если свойство AutoScroil имеет значение True (по умолчанию), то полосы прокрутки появляются и исчезают автоматически, без каких-либо действий программиста. Необходимость в полосах прокрутки может возникнуть, например, если пользователь уменьшит размеры формы так, что не все элементы управления будут полностью видимы.

Для программного управления полосами прокрутки можно использовать метод ScrollInView. Процедура ScrollInView(AControi: TControl) автоматически изменяет позиции полос прокрутки так, чтобы заданный параметром AControl управляющий элемент стал виден в отображаемой области.

В заключение отмечу редко используемую особенность — создание формы с непрямоугольным окном. Введите для обработчика события OnCreate следующий текст:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  rgn: HRGN;
begin
  rgn := CreateEllipticRgn( 10,10,400,400 );
  SetWindowRgn( Handle,rgn, True ); // Круглое окно
end;
end.
```

И получите круглое окно. Но при желании окну можно придать любой вид.

Можно сделать форму прозрачной, для чего установите значение свойства AlphaBlend в состояние True и измените значение свойства AlphaBlendValue.

Особенности модальных форм

Модальной называется форма, которая должна быть закрыта перед обращением к любой другой форме данного приложения. Если пользователь пытается перейти к другой форме, не закрыв текущую модальную форму, то Windows блокирует эту попытку и выдает предупреждающий сигнал. Запрет перехода к другой форме при незакрытой модальной форме относится только к текущему приложению, так что пользователь может активизировать любое другое приложение Windows.

Отметим, что программно возможен доступ к компонентам любой созданной формы приложения, несмотря на наличие в данный момент времени открытой модальной формы.

Модальные формы часто называют *диалогами*, или *диалоговыми окнами*, хотя существуют и немодальные диалоговые окна. Для выполнения различных операций в Windows часто используются стандартные диалоговые формы, с которыми пользователь имеет дело при работе с приложениями. Такие формы называются *общими диалогами*, или *стандартными диалогами*, для работы с ними Delphi предлагает специальные компоненты. Они будут рассмотрены отдельно.

Типичным примером модальной диалоговой формы системы Delphi является диалоговое окно About Delphi.

Диалоговые формы обычно используются при выполнении таких операций, как ввод данных, открытие или сохранение файлов, вывод информации о приложении, установка параметров приложения.

Для отображения формы в модальном режиме служит метод `ShowModal`. Например, в процедуре:

```
procedure TForm1.mnuAboutClick(Sender: TObject);
begin
  fmAbout.ShowModal;
end;
```

выбор пункта меню `mnuAbout` приводит к отображению формы `fmAbout` в модальном режиме. Пункт меню может иметь заголовок (например, "О программе"), устанавливаемый с помощью свойства Caption. Пользователь может продолжить работу с приложением, только закрыв эту модальную форму.

Многие формы можно отображать и в немодальном режиме, например следующим образом:

```
fmAbout.Show;
```

Напомним, что метод `Show` является процедурой и результат не возвращает.

При закрытии модальной формы функция `showModal` возвращает значение свойства `ModalResult` типа `TModalResult`.

Вообще говоря, код результата при закрытии возвращает любая форма. В данном случае его можно использовать для организации ветвления: возвращаемый после закрытия диалогового окна код результата анализируется, и в зависимости от значения выполняются те или иные действия.

Резюме

В данной главе рассмотрены базовые возможности создания компонентов, наличие которых позволяет компонентам функционировать в среде Delphi и отвечать определенным требованиям при выполнении программы. Разработчик может создать очень простой компонент или, используя при разработке любые, необходимые для решения поставленной задачи, функции, создать очень сложный объект, но в любом случае объект будет иметь возможности, определяющие его поведение как компонента. Именно поэтому необходимо хорошо изучить базовые функции компонентов.

В главе также довольно подробно описан наиболее часто используемый компонент — форма — и рассмотрены все его возможности.