8. Переопределение методов. Полиморфные (виртуальные) методы

Переопределение методов является одним из проявлений так называемого **полиморфизма** - важного и, в то же время, самого трудного для понимания принципа объектной технологии. Дословный перевод термина «полиморфный» – это «имеющий много форм», когда одна и та же сущность в разных ситуациях может по-разному проявлять себя.

Одним из самых простых проявлений полиморфизма можно считать возможность **перегрузки** (**overloading**) методов в классе, т.е. наличие в классе **нескольких** методов с **одним** именем, но **разным** набором формальных **параметров** и разной программной реализацией. Например, перегруженные конструкторы класса (напомним, что в языках Java, C# и C++ ситуация перегрузки конструкторов возникает автоматически при объявлении более одного конструктора) можно рассматривать как разные формы реализации одной и той же сущности – метода, отвечающего за создание и инициализацию объекта.

Перейдем к подробному рассмотрению другого проявления полиморфизма применительно к сущностям-методам, а именно – к **переопределению** методов. При этом надо ответить на следующие вопросы:

- в чем смысл механизма переопределения и что он дает
- что лежит в основе программной реализации этого механизма

Начнем с первого достаточно простого вопроса.

Переопределение (overriding) методов
– это возможность объявления в дочерних классах методов, заголовки которых полностью совпадают с базовым родительским методом, но этим методам в дочерних классах дается своя программная реализация

Несколько важных замечаний:

- использование механизма переопределения **не является обязательным** во всех объектных программах, можно создавать достаточно функциональные объектные программы без этого механизма
- переопределение неразрывно **связано** с механизмом наследования, без него оно не имеет смысла, что еще раз подчеркивает **центральное** место, которое в объектной технологии отводится принципу наследования
- полное совпадение заголовков включает в себя совпадение **имен** методов (вплоть до регистра для С-подобных программ!), **числа, порядка** и **типов** формальных параметров, если они есть (этот набор атрибутов подпрограмм часто называют сигнатурой)
- переопределяемые методы принято называть виртуальными

В результате в рамках некоторой подиерархии классов (а это хотя бы два класса – родительский и дочерний) появляются **одноименные** (правильнее сказать – **односигнатурные**) методы, т.е. методы с **одинаковой** сигнатурой (имя + параметры) но с **разным** программным кодом. Тем самым реализуется следующая интересная возможность:

В дочерних классах можно видоизменять, модифицировать поведение унаследованных родительских методов, приспосабливать их к особенностям объектов конкретных классов.

В целом это позволяет создавать более гибкие объектные программы, которые могут изменять свое поведение в процессе своего выполнения в зависимости от текущей ситуации.

Здесь **ОЧЕНЬ** важно подчеркнуть следующее: в объектной программе, созданной с помощью механизма переопределения, существует **несколько** вариантов реализации одного и того же виртуального метода

(многоформность, однако!), и решение о том какой из них вызывать принимается **во время выполнения** программы в зависимости от того**, объекту какого класса** требуется этот метод. Отсюда следует, что часть работы по связыванию программного кода переносится с этапа компиляции/компоновки на этап выполнения. Поскольку связывание программного кода во время выполнения программы требует определенных затрат, то при проектировании класса возникает вопрос о **соотношении** переопределяемых и не переопределяемых (т.е. обычных) методов. Ответ на этот вопрос сильно зависит от специфики решаемой задачи и часто является весьма непростым.

Разобьем этот вопрос на два подвопроса:

- **будут ли** в дочернем классе переопределяться унаследованные методы (если это вообще разрешено родителем) и **какие** именно
- какие **новые** методы, **впервые** вводимые в разрабатываемом классе, **имеет смысл разрешить** к переопределению в последующих производных классах

В первом случае в проектируемый класс **полностью** копируется сигнатура родительского метода (или нескольких методов), в заголовке метода он описывается как переопределяемый (виртуальный) с помощью специальной **директивы** и дается его программная реализация (разумеется, отличающаяся от родительской).

Во втором случае метод(ы) также в заголовке помечается специальным образом как **виртуальный** и либо приводится его базовая программная реализация, либо (достаточно часто) он объявляется **абстрактным**.

В итоге, в самом общем случае проектируемый класс может содержать следующие разновидности методов:

- обычные унаследованные методы, которые присутствуют в классе неявно
- переопределенные унаследованные методы (виртуальные)
- вновь введенные в классе обычные невиртуальные методы

- вновь введенные в классе методы, разрешенные к переопределению, т.е. объявленные виртуальными

Вся эта информация о методах необходима для правильной обработки описания класса компилятором/компоновщиком. При этом обычные и виртуальные методы обрабатываются по-разному.

Для обычных методов используется схема **раннего связывания** (**early binding**, статическая компоновка), когда замена (связывание) имени подпрограммы необходимым набором команд выполняется при **разработке** программы. В результате создается полностью готовый к выполнению программный код, в котором все адресные связи настроены необходимым образом, и поэтому выполнение такого кода происходит максимально быстро. Но с другой стороны, такая жесткая фиксация адресных связей не позволяет менять траекторию выполнения программы.

Для виртуальных методов используется другая схема – так называемое **позднее связывание** или **динамическая компоновка**. Здесь компилятор/компоновщик выполняет лишь некоторую **подготовительную** работу, а **окончательное** связывание имени подпрограммы с исполняемым кодом происходит при **работе** программы. В этом случае создаваемый программный код имеет «**полуфабрикатный**» вид, в нем отсутствуют некоторые необходимые программные фрагменты, что позволяет при выполнении программы в одной и той же ее точке **динамически** подключать **разные** фрагменты программного кода. Это безусловно повышает гибкость объектной программы, но несколько замедляет ее выполнение.

Эти факторы обязательно надо учитывать при проектировании классов и решении вопроса о степени использования виртуальных методов. Если в классе **все** методы будут объявлены как **обычные**, то степень гибкости программы будет минимальной, но зато скорость выполнения – максимально возможной (при прочих равных условиях). Наоборот, если **все**

методы будут **виртуальными**, то гибкость программы будет максимальной, но время выполнения – больше. На практике обычно стремятся найти разумный **компромисс** между гибкостью и скоростью выполнения.

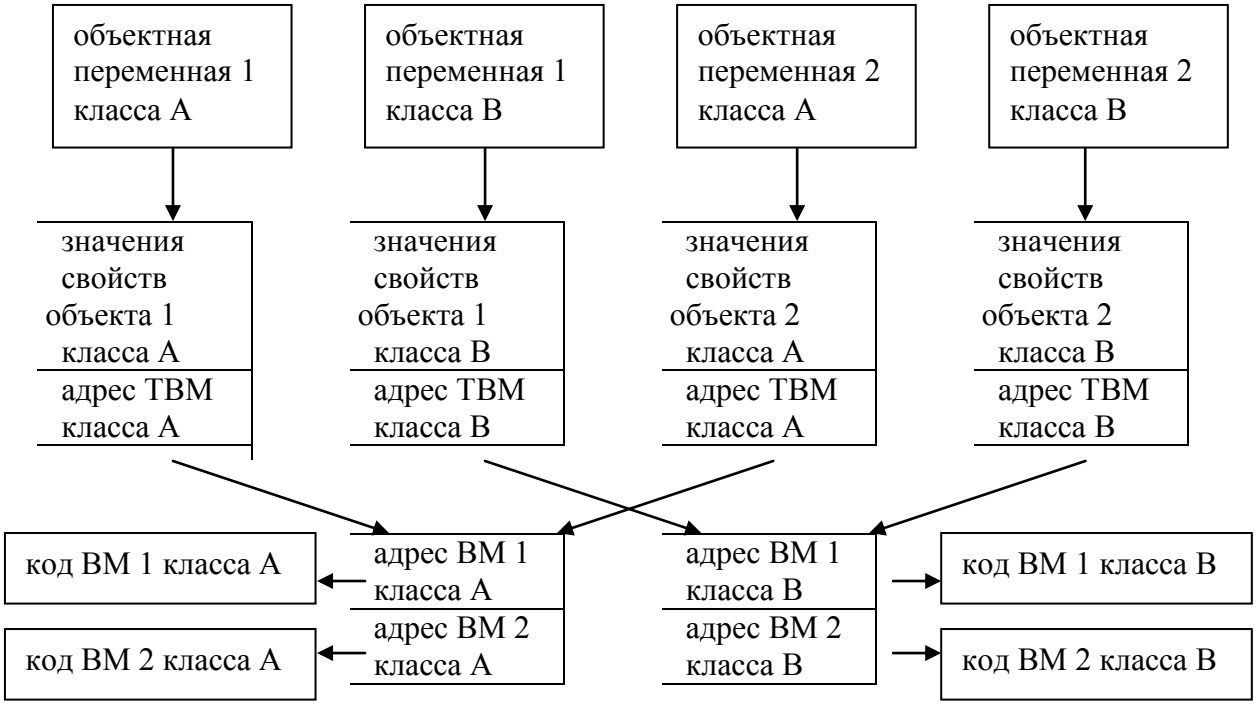
Действительно, далеко не все методы следует объявлять виртуальными. Например, вряд ли имеет смысл объявлять виртуальными методы доступа к закрытым свойствам класса. Также виртуальными не объявляют конструкторы в силу их особого предназначения. А вот метод прорисовки графических объектов разумно объявить виртуальным: все такие методы имеют **одинаковое** имя (поскольку параметров нет, то сигнатура сводится просто к имени), но **свою** программную реализацию в каждом классе. Это открывает целый ряд интересных возможностей, которые описываются в следующих разделах пособия.

Теперь немного поговорим о тех внутренних механизмах, на которых основана динамическая компоновка. Если в классе объявлены виртуальные методы, то компилятор/компоновщик в точках вызова этих методов формирует специальные конструкции, часто называемые «**заглушками**». Кроме того, создаются специальные **структуры данных** с информацией об используемых в классе виртуальных методах. Часто такие структуры называют **Таблицами Виртуальных Методов** (**VMT, Virtual Method Table**). В разных языках эти таблицы реализованы немного по-разному, но общий принцип их использования одинаков: предоставить механизму динамической компоновки информацию о виртуальных методах класса.

Для каждого класса, где есть хотя бы один виртуальный метод создается **своя** отдельная таблица. Каждая такая таблица во время выполнения программы находится в оперативной памяти и содержит **адреса** размещения в памяти кода соответствующих виртуальных методов. Для доступа к этим таблицам у каждого объекта есть **внутреннее** скрытое от разработчика поле, которое используется для хранения **адреса** соответствующей таблицы. Другими словами, каждый объект **связан** со своей таблицей виртуальных методов. Заполнение этого поля поручено

конструктору и производится при создании объекта. Все объекты-экземпляры одного класса связаны с **одной и той же** таблицей.

Например, пусть имеются два класса A и B, в каждом из которых объявлено по два виртуальных метода. Пусть при выполнении программы создано по два объекта этих классов. Тогда в памяти будут динамически выделены области для самих объектов, их таблиц виртуальных методов и кода этих методов. Схематично связи между этими областями можно представить следующим образом.



Эта схема помогает понять, что происходит при выполнении программы, когда объекты вызывают виртуальные методы. Пусть выполнение программы дошло до точки, где объект 1 класса A вызывает свой виртуальный метод 1. Тогда с помощью соответствующей таблицы определяется размещение в памяти кода этого метода и вместо заглушки происходит передача управления найденному фрагменту. Если же к виртуальному методу 1 обращается объект класса B, то происходит обращение к другой таблице и вызывается другой (альтернативный) программный код. В следующем разделе эти общие концепции иллюстрируются на примере иерархии графических объектов.

В завершение данного раздела опишем **синтаксические** приемы, используемые для объявления виртуальных методов в разных языках. Здесь, как ни странно, **особняком** стоит язык Java. Дело в том, что в языках Delphi/FP, C#, и C++ **по умолчанию** методы считаются **обычными**, не переопределяемыми, т.е. обрабатываемыми по схеме раннего связывания. А вот в языке Java **по умолчанию все методы считаются виртуальными**, и для них используется механизм динамической компоновки. Это необходимо учитывать при объявлении методов в этих языках.

В языках C#, C++ и Delphi для объявления метода переопределяемым в родительском классе в его заголовке надо поставить директиву **virtual**, а в дочерних классах использовать либо директиву **override** (языки C# и Delphi), либо опять же директиву **virtual** (язык C++).

```
TParentClass = class
   procedure VirtMethod (параметры); virtual; // можно переопределять
   procedure Method (параметры);  // обычный (не переопределяемый)
end;

TChildClass = class (TParentClass)
   procedure  VirtMethod (параметры); override; // переопределяем
   //  метод  Method  просто наследуется и используется без изменений
   procedure NewMethod (параметры);  // новый обычный метод
   procedure NewVirtMet (параметры); virtual; // новый виртуальный
end;
```

```
class  ParentClass
{  virtual  void  VirtMethod (параметры);  // можно переопределять
   void  Method (параметры);  // обычный (не переопределяемый)
};
```

```
class   ChildClass :  ParentClass
{   override  void   VirtMethod (параметры);   // переопределяем
   //  метод  Method   просто наследуется и используется без изменений
   void   NewMethod (параметры);   // новый обычный метод
   virtual   void   NewVirtMet (параметры);   // новый виртуальный
};
```

В отличие от этих языков, в Java **отключается** механизм переопределения, для чего используется директива **final** :

```
class   ParentClass
{  void   VirtMethod (параметры);   // оставляем как переопределяемый
   final   void   Method (параметры);   // делаем не переопределяемым
};

class   ChildClass  extends  ParentClass
{   void   VirtMethod (параметры);   // переопределяем
   //  метод  Method   просто наследуется и используется без изменений
   final   void   NewMethod (параметры);   // новый обычный метод
   void   NewVirtMet (параметры);   // новый виртуальный
};
```

В качестве **примера** рассмотрим задачу разработки библиотеки классов для графических объектов с использованием виртуальных методов**.** В разделе 7 была разработана небольшая иерархия классов для графических объектов (см. схему). В этой иерархии по вполне понятным причинам не использовался механизм переопределения методов. Теперь же, после изучения общих принципов переопределения, можно пересмотреть данную иерархию с точки зрения использования в ней виртуальных методов.

Оставляя «за скобками» конструкторы и методы доступа, в числе претендентов на виртуализацию остаются методы **отображения** и **перемещения**, и здесь возникают достаточно интересные моменты. Метод отображения **уникален** для каждого класса и поэтому в каждом классе должен быть реализован **своим** программным кодом. А вот метод перемещения **алгоритмически одинаков**: перемещение **любой** фигуры можно описать тремя основными шагами:

- стираем старое изображение за счет вызова метода прорисовки
- изменяем координаты базовой точки фигуры
- снова вызываем метод прорисовки

Ну а поскольку алгоритм перемещения одинаков, то зачем в каждом классе **повторять** программный код метода перемещения? Разумно выполнить реализацию этого метода **один раз** – в **абстрактном классе фигур** и далее **наследовать** его во всех производных классах. Напомним, что понятие абстрактного класса не запрещает включать в него программный код некоторых методов. Следовательно, метод перемещения не имеет смысла делать переопределяемым.

Идем дальше. Мы решили реализовать метод перемещения на «самом верху» - в классе фигур. Но для программной реализации этого метода необходимо вызывать метод отображения, а вот его в классе фигур **нельзя** реализовать в принципе, можно включить в класс лишь его **заголовок**! Раз нельзя – так нельзя, но заголовок-то метода **есть**, вот и будем в методе перемещения «как бы вызывать» метод отображения, используя только его заголовок! Тем самым метод перемещения оформляется как некая **заготовка** настоящего полноценного метода, а для того чтобы все это заработало, метод прорисовки надо объявить **виртуальным** и переопределять в каждом производном классе.

В этом случае при обработке класса фигур компилятор/компоновщик для метода перемещения создаст лишь **часть** программного кода, поскольку в точках вызова метода отображения вместо реальных команд будут

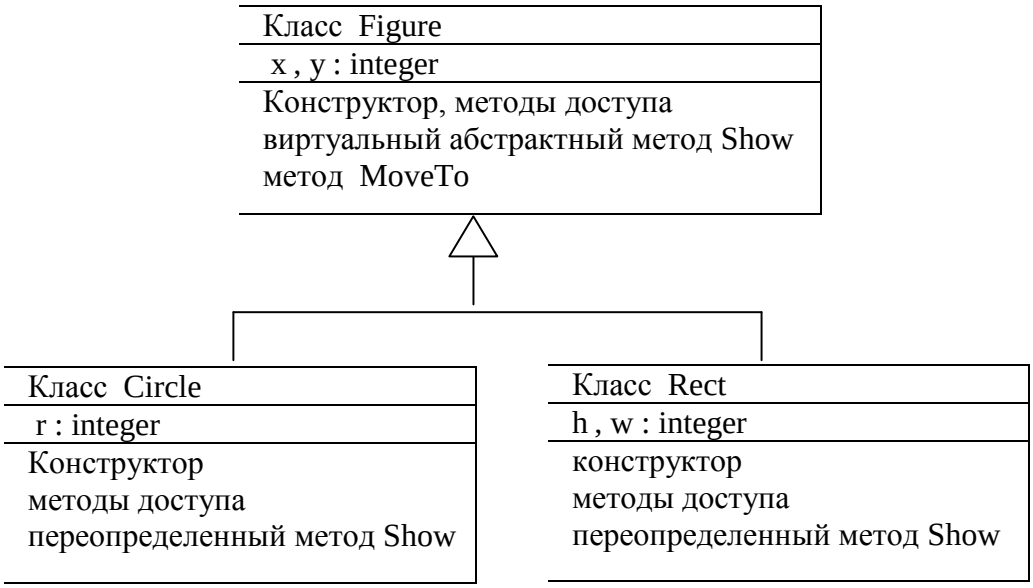
вставлены **заглушки**. При обработке дочерних классов, в которых переопределены методы прорисовки, компилятор/компоновщик для каждого класса создаст **таблицу виртуальных методов**, которая во время выполнения программы способна хранить **адрес размещения** в памяти программного кода **конкретной** версии метода отображения. Это позволит во время работы программы **динамически настраивать** метод перемещения на работу с конкретным объектом, подставляя вместо заглушек **необходимый** код отображения. Это и есть позднее связывание: имя метода (в данном случае – метода отображения) заменяется программным кодом лишь во время работы программы!

Теперь становится понятным, почему во **всех** классах методы отображения должны иметь **одинаковые** имена (неважно – Show, Draw или как-то еще, главное – одинаково!). Напомним еще раз, что одноименность методов является обязательным условием использования механизма переопределения. Это и есть полиморфизм методов – имя одно, а вариантов программной реализации много (ну хотя бы два :)).

Итак, в иерархию классов для графических объектов надо внести **следующие изменения:**

- в базовом классе фигур метод **отображения** объявляем **виртуальным**, оставляя его **абстрактным**
- здесь же метод **перемещения** объявляем уже **НЕ абстрактным**, и следовательно даем ему программную реализацию, в которую вставляем вызовы абстрактного виртуального метода отображения
- во всех производных классах **убираем методы перемещения,** поскольку этот метод теперь может **наследоваться** из корневого класса иерархии и с помощью механизма позднего связывания **динамически настраиваться** на перемещение объекта **любого** класса
- во всех производных классах **переопределяем** метод отображения и даем ему **конкретную** программную реализацию с учетом специфики объектов данного класса

Фрагмент UML-диаграммы классов для улучшенной иерархии графических фигур теперь можно представить следующим образом.



Этот простой пример иллюстрирует возможности мощного механизма динамической настройки, т.к. однажды созданный метод перемещения будет в дальнейшем успешно работать с объектами **любых классов,** лишь бы они входили в общую иерархию классов и реализовывали метод отображения с общим именем Show.

Приведем фрагменты описания классов для графических фигур с использованием виртуальных методов (для наглядности убраны директивы ограничения доступа).

```
TFigure = class
. . . . .
procedure  Show;  abstract; virtual;   // реализации нет!
procedure  MoveTo( . . . );  // обязательно дать реализацию метода!
end;
```

```
TCircle = class (TFigure)
procedure  Show;  override;  //  конкретная реализация для окружности!
. . . . .                          //   метод перемещения НЕ нужен!
end;

TRect = class (TFigure)
procedure  Show;  override;  //  конкретная реализация для прямоугольника
. . . . .                          //   метод перемещения НЕ нужен!
end;
```

C#

```
class    Figure
  { . . . . .
virtual   abstract   void  Show( );  // реализации нет
void    MoveTo(int  dx,  int  dy) { Show( );  x = x + dx;  y = y + dy;  Show( ); }
  };

class   Circle : Figure  {
override   void   Show( ) {…}  //  конкретная реализация для окружности!
. . . . .                          //   метод перемещения НЕ нужен!
 };

class   Rect : Figure  {
override  void   Show( ) {…}  // конкретная реализация для прямоугольника
. . . . .                          //   метод перемещения НЕ нужен!
 };
```

Java

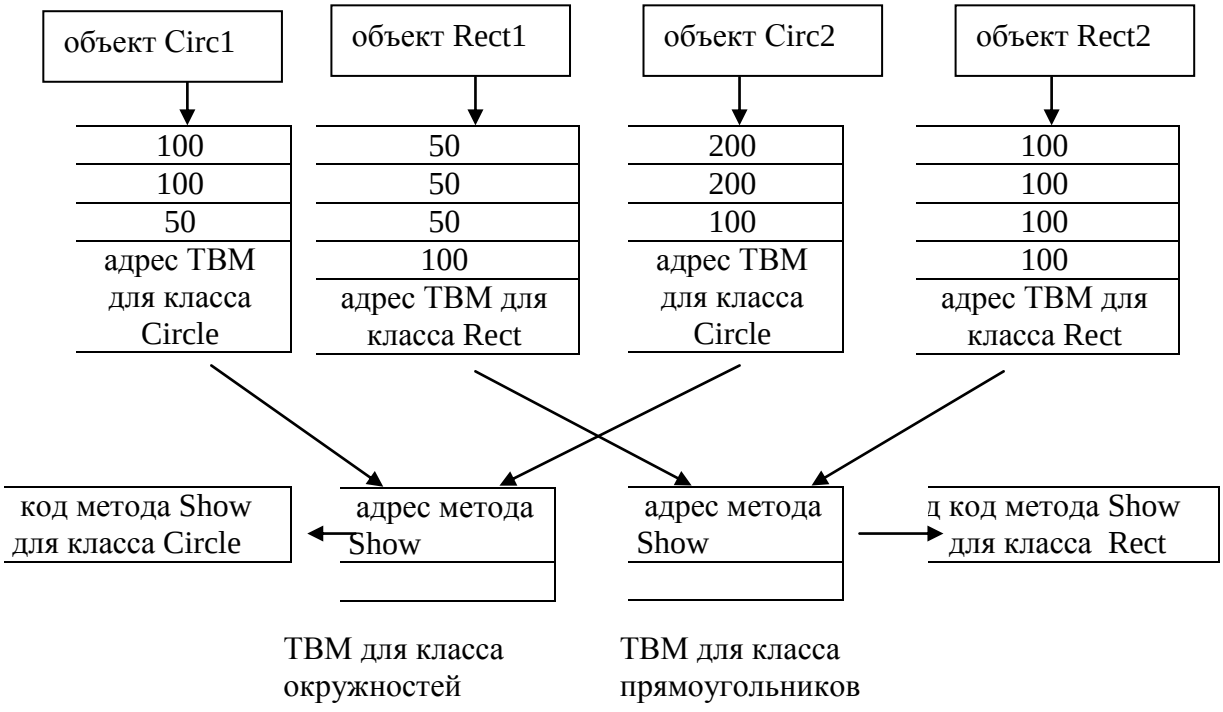
```
class   Figure   {
```

```
. . . . . .
abstract void Show ( );  // виртуальный абстрактный метод
final void MoveTo( . . . ) { Show( );  x = x + dx;  y = y + dy;  Show( ); }
 };

class Circle extends Figure {
void Show ( ) { реализация переопределенного метода для окружности}
. . . . .                    //  а вот метод перемещения здесь уже НЕ нужен!
};

class Rect extends Figure {
void Show( ) {…}  // конкретная реализация для прямоугольника
. . . . .                          //  метод перемещения НЕ нужен!
};
```

Предположим, что в процессе выполнения программы созданы два объекта-окружности Circ1 и Circ2 и два объекта-прямоугольника Rect1 и Rect2. Тогда в основной памяти будут выстроены связи, которые условно можно показать с помощью следующей схемы:



Пусть в программе реализовано несколько обращений к методу перемещения MoveTo со стороны разных объектов:

```
Circ1.MoveTo( ); // операцию перемещения вызывает объект-окружность
Rect2.MoveTo ( );  //  а теперь – объект-прямоугольник
```

При отработке этих вызовов включается механизм динамической компоновки, который прежде всего определяет, объект **какого класса** обращается к методу перемещения. После этого с помощью Таблицы Виртуальных Методов для данного класса в оперативной памяти находится код, реализующий метод Show именно для этого класса. Найденный программный код подключается к методу перемещения MoveTo. Например, для перемещения окружности Circ1 происходит обращение к ТВМ класса окружностей, и именно этот программный код подключается к MoveTo, а для перемещения прямоугольника Rect2 обращение идет уже к другой ТВМ и с ее помощью активизируется другой фрагмент программного кода.

Отметим, что это еще не все преимущества использования виртуальных методов. После рассмотрения в следующем разделе использования полиморфизма применительно к объектным переменным мы вернемся к примеру с иерархией графических классов и увидим какие новые возможности дают все эти механизмы вместе.