**Структурное программирование**

Самое общеизвестное определение структурного программирования – подход к программированию, в котором для передачи управления в программе используется три конструкции: следование, выбора и цикл.

Классическая теорема Боэма и Джакопини о структурном программировании утверждает, что всякую правильную программу (т. е. программу с одним входом и одним выходом, без зацикливаний и недостижимых веток) можно записать с использованием следующих логических структур:

последовательности двух или более операторов;

дихотомического выбора;

повторения;

Дейкстра предложил отказаться от оператора безусловного перехода и ограничиться тремя конструкциями – последовательностью, выбором и циклом;

Дональд Кнут продемонстрировал случаи, в которых оператор безусловного перехода оказывался полезным (например, выход из нескольких вложенных циклов) и подверг критике утверждение Дейкстры.

В 1965 году академик Глушков обратил внимание на то, что структурированные программы можно рассматривать как формулы в некоторой алгебре. Зная правила преобразования выражений в такой алгебре, можно осуществлять глубокие формальные (и, следовательно, автоматизированные) преобразования программ.

*Структурное программирование* – не самоцель, его основное назначение – получение хорошей программы. Однако даже в самой хорошей программе операторы перехода требуются, например при выходе из множества вложенных циклов.

**Модульное программирование**

Модульное программирование – это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой. Единственная альтернатива модульности – монолитная программа, что, конечно, неудобно. Таким образом, наиболее интересный вопрос при изучении модульности – определение критерия разбиения на модули. В основе модульного программирования лежат три основные концепции.

*Принцип утаивания информации*. Всякий компонент утаивает единственное проектное решение, т. е. модуль служит для утаивания информации. Подход к разработке программ заключается в том, что сначала формируется список проектных решений, которые особенно трудно принять или которые, скорее всего, будут меняться. Затем определяются отдельные модули, каждый из которых реализует одно из указанных решений.

*Аксиома модульности*. Модуль – независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы. Программная единица должна удовлетворять следующим условиям:

– блочность организации, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;

– синтаксическая обособленность, т. е. выделение модуля в тексте синтаксическими элементами;

– семантическая независимость, т. е. независимость от места, где программная единица вызвана;

– общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;

– полнота определения, т. е. самостоятельность программной единицы.

*Сборочное программирование*. Модули – это программные кирпичи, из которых строится программа.

*Сцепление модулей* – мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

– независимые модули – это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого.

– сцепление по данным (параметрическое) - это сцепление, когда данные передаются модулю, как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам).

Рутинность модуля – это независимость модуля от предыдущих обращений к нему (от предыстории). Модуль является рутинным, если результат его работы зависит только от количества переданных параметров (а не от количества обращений). Модуль должен быть рутинным в большинстве случаев, но есть и случаи, когда модуль должен сохранять историю. В выборе степени рутинности модуля пользуются тремя рекомендациями:

– в большинстве случаев делаем модуль рутинным;

– зависящие от предыстории модули следует использовать только в тех случаях, когда это необходимо для сцепления по данным;

– в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы пользователи имели возможность прогнозировать поведение такого модуля.

**Объектно-ориентированное программирование**

*Методология объектно-ориентированного программирования* – подход, использующий объектную декомпозицию, при которой статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

В основе объектно-ориентированного языка программирования лежат понятия: объект, класс, инкапсуляция, наследование и полиморфизм.

В частном случае в Visual Basic *объектом* называется элемент пользовательского интерфейса, который создается на форме Visual Basic, а также элементы управления и отображения.

Каждый объект есть представитель некоторого *класса* однотипных объектов, т.е. объект является экземпляром класса. Класс определяет общие для всех его объектов методы и свойства. *Методы* – это программные процедуры, определяющие взаимодействие объектов класса с внешней средой. *Свойства* представляют собой характеристики (атрибуты), присущие объектам (например, размер шрифта, название и др.).

*Инкапсуляция* – это скрытие информации. При объектно-ори­ентированном программировании возможен доступ к объекту только через его методы и свойства. Внутренняя структура объекта скрыта от пользователя, т.е. объекты – это самостоятельные сущности, отделенные от внешнего мира. Инкапсуляция позволяет изменять реализацию объектов любого класса без опасений, что это вызовет нежелательные побочные эффекты в программной системе. Это мощное средство обеспечивает многократное использование одного и того же программного кода.

*Наследование* – это возможность выделить свойства, методы и события одного объекта и приписать их другому объекту, иногда с их модификацией. С точки зрения программиста, новый класс должен содержать только коды и данные для новых или изменяющихся методов.

*Полиморфизм* – это способность объектов выбирать операцию на основе данных, принимаемых в сообщении. Каждый объект может реагировать по-своему на одно и то же сообщение. Например, команда **Print** будет по-разному воспринята черно-белым или цветным принтером.

**Базовые понятия объектно-ориентированного программирования**

При использовании ООП основными действующими лицами являются не переменные, а *объекты*.

Объектам Visual Basic присуща функциональность. Иными словами, они действуют определенным образом и могут откликаться на определенные ситуации. При этом если свойства объекта определяют его внешний вид и поведение, то методы объекта – те задачи, которые может выполнить данный объект. Методы по сути дела представляют собой сегмент программного кода, внедрен­ный в объект.

Существует определенный формат программного кода, задающего установку свойства и использование метода:

Объект.Свойство = Значение

Объект.Метод [Параметр1 [...]]

Здесь *Объект* – имя настраиваемого объекта; *Свойство* – характеристика, которую нужно изменить; *Метод* – команда, которая используется для изменения объекта; *Значение* – новая установка свойства; *Параметр* – аргумент, используемый методом.

Пусть необходимо написать программу, которая будет рисовать на экране снеговика, состоящего и кругов какого-либо определенного цвета. Необходимо предусмотреть возможность перемещения снеговика по экрану в различных направлениях – процедура Move. Сделать возможным изменение цвета для всех кругов, из которых состоит снеговик.

Все крути можно представить в виде самостоятельных объектов с одинаковой структурой, которые отличаются друг от друга лишь значениями параметров. Действие процедур Draw (рисование круга), Move (перемещение круга) и ChangeColor (смена цвета круга), идентичны применительно к каждому из кругов. Все круги являются объектами, которые имеют одинаковую структуру (радиус, координаты центра, цвет) и ведут себя схожим образом при выполнении любого действия (рисование, перемещение, изменение цвета), т.е. все они принадлежат одному и тому же семейству.

С точки зрения объектно-ориентированного программирования все "действующие лица" программы представляют собой *объекты,* каждый из которых является элементом (экземпляром) какого-либо *класса.* Параметры объекта (радиус, цвет и т. п.) называются его *свойствами,* а процедуры или функции, которые он выполняет в ответ на какой-либо запрос (поменять цвет, переместиться и т. п.), называются *методами.* При этом метод, который должен вызываться в ответ на запрос, определяется классом, экземпляром которого является данный объект. И наоборот, если объекты принадлежат одному и тому же классу, то они должны вызывать одинаковые методы в ответ на один и тот же запрос.

Представление свойств и методов как неотъемлемых частей любого объекта носит название *инкапсуляции*, т.е. в этом случае объект можно представить как своеобразную оболочку (капсулу), которая "окружает" программный код (методы) и данные (свойства). На рис. 1 приведена иллюстрация этого важного понятия ООП.

|  |
| --- |
| ОБЪЕКТ |
|  | Свойства |  | Методы |  |
|  |  |

Рис.1. Иллюстрация понятия инкапсуляции

Пусть класс, которому принадлежат все объекты-круги, называется Round.

Свойствами класса Round являются следующие:

– *R* – радиус круга;

– *X*, *Y* – координаты центра круга;

– *Color* – цвет круга.

Методы класса Round:

– Draw – рисует круг с заданными параметрами;

– Move – перемещает круг на определенное расстояние в выбранном направлении;

– ChangeColor – изменяет цвет крута.

Для того чтобы нарисовать снеговика, потребуются три объекта-круга. Верхний из них можно назвать Head, средний – Body, a нижний – Foot. Все эти объекты принадлежат классу Round. Следовательно, все они имеют одинаковые свойства (*R*, *X*, *Y*, *Color*) и вызывают одинаковые методы (Draw, Move, ChangeColor) в ответ на одни и те же запросы.

Таким образом, в ООП принципы абстрагирования и сокрытия данных нашли свое максимальное отражение. Программа состоит из независимых блоков, которые могут использоваться и в других приложениях. ООП зачастую позволяет создавать конечный продукт просто путем соединения нескольких готовых блоков, написанных ранее.

**Наследование**

Пусть необходимо нарисовать снеговика с глазами, которые должны время от времени мигать. Все остальные свойства и методы у объектов-глаз должны быть такими же, как и у объектов-кругов, принадлежащих классу Round, т. е, глаза снеговика тоже нужно перемещать по экрану вместе с остальными кругами, а также должна оставаться возможность изменения их цвета. Для реализации этого дополнения можно создать еще один класс – Eyes (глаза), в котором будут те же свойства и методы, что и в Round, но еще будет добавлен метод Blink (мигание). Однако при этом придется для класса Eyes заново создавать все свойства и методы, которые уже были описаны для Round.

Чтобы не выполнять подобную двойную работу, в ООП существует механизм наследования. Этот механизм реализуется путем создания *иерархической структуры* классов. Вначале создается класс Round, для которого описываются все свойства и методы. Он располагается в основании иерархической структуры. Затем создается еще один класс под названием Eyes, в котором описывается новый метод Blink, а все остальные свойства и методы остаются без изменений, т.е. класс Eyes *наследует* все свойства и методы Round и, в то же время, включает в себя дополнительный метод Blink. В иерархической структуре Eyes следует за классом Round, и в этом случае говорят, что Round является *предком* (или *родительским классом*) для Eyes. В свою очередь, Eyes является *потомком* (или *дочерним классом* для Round.

*Наследованием* называется возможность доступа объектов, принадлежащих классу-потомку, к методам и свойствам класса-предка. Иллюстрация наследования приведена в табл. 1.

Таким образом, применение наследования в ООП ведет к значительному уменьшению объема программы и повышению ее функциональности.

**Полиморфизм**

Этот термин относится к таким переменным или параметрам процедур (функций), которые в процессе выполнения программы могут принимать значения разных типов. Процедуры (функции), имеющие полиморфные параметры, также называются *полиморфными*.

Пусть необходимо рисовать на экране не только круги, но и другие геометрические фигуры (например, прямоугольники), причем с ними нужно выполнять те же самые действия, которые были описаны ранее (рисовать, перемещать, изменять цвет фигуры). Но для того, чтобы нарисовать прямоугольник, необходимо знать координаты какой-либо его вершины (например, левой верхней), а также его длину и ширину.

Таблица 1. Наследование классом Eyes свойств и методов Round

|  |  |
| --- | --- |
| Round | *Свойства* |
| R |
| X |
| Y |
| Color |
| *Методы* |
| Draw |
| Move |
| ChangeColor |
| Eyes | *Свойства* |
| R |
| X |
| Y |
| Color |
| *Методы* |
| Draw |
| Move |
| ChangeColor |
| Blink |

Для реализации приведенных требований следует создать класс Rect, являющийся потомком Round, и применительно к нему *X* и *Y* – это координаты выбранной вершины прямоугольника, a *R* – его длина (или ширина). Но в этом случае придется описать еще одно свойство для обозначения ширины (или длины). Очевидно, что при работе с подобными классами легко может возникнуть путаница в использовании свойств, и причина этого – узкая направленность класса Round, который ориентирован на работу только с определенной геометрической фигурой.

Чтобы сделать Round более универсальным, т.е. пригодным для изображения большего числа геометрических фигур, следует изменить его свойства следующим образом. Вместо радиуса и координат центра нужно описать такие свойства как:

– Тор – координата верхний границы фигуры;

– Left – координата левой границы фигуры;

– Height – высота фигуры;

– Width – ширина фигуры.

По этим свойствам легко можно определить недостающие координаты для любой фигуры. Например, для получения всех координат прямоугольника (или квадрата) необходимо к координате левой границы прибавлять ширину, а из верхней вычитать высоту. Пусть

Тор = 80,

Left = 20,

Height = 40,

Width = 60.

Тогда

– правая граница равна

Left + Width = 80,

– нижняя равна

Тор – Height = 40.

Следовательно, данный прямоугольник имеет следующие координаты (первая координата – для горизонтальной оси, вторая – для вертикальной): А(20, 80), В(80, 80), С(20, 40), D(80, 40).

Теперь на основе предка Round с описанным выше новым набором свойств можно создать потомка Rect для изображения прямоугольников. Однако методика рисования в этом случае должна отличаться от той, которая использовалась при работе с кругами. Т.е. необходимо, чтобы для Rect содержимое унаследованного метода Draw отличалось от родительского, в противном случае, если вызывать Draw для объекта, принадлежащего классу Rect, то на экране все равно будет нарисован круг, а не прямоугольник.

Чтобы этого избежать, необходимо для Rect описать свой метод Draw. Тогда для объекта, принадлежащего классу Round, вызов Draw будет означать изображение на экране круга, а для объекта из Rect – прямоугольника. Т.е. методы с одним и тем же именем Draw отвечают за разные действия в зависимости от того, объекту какого класса необходимо выполнить рисование. Подобная ситуация в ООП носит название *полиморфизма.* Графическая интерпретация описанного примера полиморфизма в ООП приведена на рис. 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Классы |  | Метод |  | Действия |
| Rect |  | Draw |  | Рисует прямоугольник |
| Round |  |  | Рисует круг |

Рис 2. Полиморфизм при вызове метода Draw объемами классов Rect

и Round