Лекция

Понятие парадигмы

✓ *ПАРАДИГМА (от греч. paradeigma — пример, образец):*

а) Строго научная теория, воплощенная в системе понятий, выражающих существенные черты действительности.

б) Исходная концептуальная схема, модель постановки проблем и их решения, методов исследования.

в) Набор теорий, стандартов и методов, которые совместно представляют собой способ организации научного знания, иными словами, способ видения мира.

✓ *ПАРАДИГМА В ПРОГРАММИРОВАНИИ* — это совокупность идей и понятий, определяющая *стиль написания программ*[1].

Программирование — процесс создания компьютерных программ с помощью языков программирования. Программирование сочетает в себе **элементы искусства, науки, математики** *и* **инженерии**. Большая часть работы программиста связана с написанием исходного кода на одном из языков программирования.

Различные языки программирования поддерживают различные парадигмы программирования (так называемые стили программирования). Отчасти, **искусство программирования** состоит в том, чтобы выбрать один из языков, наиболее полно подходящий для решения имеющейся задачи. Разные языки требуют от программиста различного уровня внимания к деталям при реализации алгоритма, результатом чего часто бывает компромисс между простотой и производительностью (или между временем, затрачиваемым программиста и временем, затрачиваемым пользователя).

Основные парадигмы современного программирования

В настоящее время известно множество парадигм программирования, каждая из которых имеет свою собственную модель вычислений. Причем, количество и разнообразие их непрерывно растет.

Наиболее распространенные парадигмы программирования:

1. Нисходящее программирование и Восходящее программирование.

[1] Здесь *стиль* надо понимать в самом широком смысле — как набор концепций, используемых при проектировании и разработке программ.

2. Императивное (или процедурное) программирование.
3. Модульное программирование.
4. Структурное программирование.
5. Событийное программирование.
6. Визуальное программирование.
7. Объектно-ориентированное программирование.

8. Декларативное программирование.
9. Автоматное программирование.
10. Функциональное (аппликативное) программирование.
11. Логическое программирование.
12. Аспектно-ориентированное программирование.
13. Доказательное программирование.
14. Динамическое программирование.
15. Экстремальное программирование.
16. Эвристическое программирование.
17. Параллельное программирование.
18. Нейронные сети.
19. ~~Линейное программирование.~~

http://comp.vslovar.org.ru/635.html

Нисходящее и Восходящее программирование (проектирование)

Программирование «сверху вниз». Нисходящее программирование — методика разработки программ, при которой разработка начинается с определения целей решения проблемы/задачи, после чего идет последовательная детализация. А именно, чтобы решить задачу, она делится на подзадачи, подзадачи делятся на подзадачи и т.д. до тех пор, пока алгоритм подзадачи не станет очевиден. Методы, на которые можно разделить первоначальную задачу, зависят непосредственно от методов, которыми можно склеивать/объединять между собой решения отдельных подзадач.

Пример. Решение задач на основе СЛАУ.

При Нисходящем проектировании разработка программного комплекса идет «сверху вниз».

На первом этапе разработки кодируется, тестируется и отлаживается головной модуль, который отвечает за логику работы всего программного комплекса. Остальные модули заменяются заглушками, имитирующими работу этих модулей. Применение заглушек необходимо для того, чтобы на самом раннем этапе проектирования

можно было проверить работоспособность головного модуля. На последних этапах проектирования все заглушки постепенно заменяются рабочими модулями.

При *Восходящем* проектировании разработка идет «снизу-вверх». На первом этапе разрабатываются модули самого низкого уровня. На следующем этапе к ним подключаются модули более высокого уровня и проверяется их работоспособность. На завершающем этапе проектирования разрабатывается головной модуль, отвечающий за логику работы всего программного комплекса. Методы нисходящего и восходящего программирования имеют свои преимущества и недостатки.

Императивное (процедурное) программирование

Императивный (от лат. Imperativus) — это безусловный, повелительный, не допускающий выбора. (Из словаря).

Императивная парадигма программирования описывает процесс вычислений посредством описания управляющей логики программы, т.е. в виде последовательности отдельных команд, которые должен выполнить компьютер. Каждая команда является инструкцией, изменяющей состояние программы. Программа, написанная в императивном стиле, похожа на набор приказов, выражаемых повелительным наклонением в естественных языках.

Машинный код является наиболее низкоуровневым примером реализации этой парадигмы: состояние программы определяется содержимым памяти, а команды — инструкциями машинного кода. Поскольку эта парадигма естественна для человеческого понимания и непосредственно реализована на аппаратном уровне, большинство языков программирования придерживаются именно ее.

Императивное программирование является противоположностью *декларативного* программирования (HTML); второе описывает, *что* необходимо сделать, а первое — *как* именно это сделать.

Чем **императивное программирование** отличается от **процедурного**?

Ничем. Когда говорят о процедурном программировании, хотят подчеркнуть метасистемный переход от элементарных действий к более высокоуровневым действиям, представленных процедурами и функциями.

Использование процедур является естественным способом борьбы со сложностью любой задачи. В *C++* задача может быть разделена на более простые и обозримые части с помощью функций, после чего программу можно рассматривать в более укрупненном виде — на уровне взаимодействия функций. Это важно, поскольку человек способен помнить ограниченное количество фактов.

Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры. Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

https://msdn.microsoft.com/ru-ru/library/bb669144.aspx

Обычно процедурно-ориентированные языки задают программы как совокупности процедур или подпрограмм.

Основными чертами процедурных языков программирования являются:

- значительная сложность;
- отсутствие строгой математической основы;
- необходимость явного управления памятью (которым вынужден заниматься программист), в частности, необходимость описания переменных;
- малая пригодность для символьных вычислений;
- высокая эффективность реализации на традиционных ЭВМ (ЭВМ, имеющих фоннеймовскую структуру).

Так как при выполнении программы в ЭВМ могут возникать побочные эффекты (взаимное влияние программных модулей через общую память), то программы на процедурных языках в этом отношении трудно отлаживать, т.е. они ненадежны. Поэтому предпочтительно только последовательное выполнение.

Модульное программирование

Развитие модульного программирования было обусловлено:

- возрастающими объемами программ;
- увеличивающейся внутренней сложностью;

- коллективным характером разработок.

В общем случае модуль представляет собой совокупность программных ресурсов, предназначенных для использования другими модулями и программами.

Интерфейсом модуля являются заголовки всех функций и описания, доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью: директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строго говоря, раздельная компиляция не является элементом языка, а относится к его реализации). Модульность ведет к повышению производительности при создании проекта. *Прежде всего*, маленькие модули могут быть закодированы быстро и легко. *Во-вторых*, универсальные модули могут многократно использоваться, что приводит к более быстрому построению последующих программ. *В-третьих*, модули программы могут быть проверены независимо, что помогает уменьшить время, потраченное на отладку. (разделяй и властвуй)

Структурное программирование

Предполагает создание удобочитаемых программ, характерными особенностями которых являются модульность, отказ от оператора безусловного перехода, ограниченное использование глобальных переменных. Основная идея структурного программирования соответствует принципу «Разделяй и властвуй» т.е. программа делится на ряд простых подзадач. Структурное программирование — методология и технология разработки программных комплексов, основанная на принципах:

- нисходящего программирования или программирования «сверху вниз»;
- модульного программирования.

При этом логика алгоритма и программы должны использовать три основные структуры: последовательное выполнение, ветвление и повторение. Блоки в структурной программе не имеют несколько входов или выходов. Структурные программы лучше поддаются математической обработке, чем их неструктурные аналоги.

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление

программы в виде иерархической структуры блоков. Предложена в 70-х годах XX века Э. Дейкстрой, разработана и дополнена Н. Виртом. В соответствии с данной методологией любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:

- **Линейная** — однократное выполнение операторов в том порядке, в котором они записаны в тексте программы;
- **Разветвляющая** — выбор одного из альтернативных путей работы алгоритма в зависимости от выполнения некоторого заданного условия;
- **Циклическая** — многократное исполнение операторов до тех пор, пока выполняется некоторое заданное условие.

Структурное программирование облегчает написание больших программ, упрощает их отладку, помогает разделению работы между исполнителями и облегчает дальнейшее изменение программы с целью модернизации.

Чтобы написать структурированную программу надо разбить задачу на отдельные осмысленные части, и начать писать ее с головной программы.

Все что не ясно, надо исключать из головной программы и относить к блокам программ, а в головном модуле выяснять только, когда к какому блоку следует обратиться.

Структурное программирование обычно связывают с правилами, которыми следует придерживаться при написании текстов программ:

- сопровождение текста грамотными комментариями;
- подчеркиванием вложенности операторов: если оператор принадлежит другому оператору, то принадлежность следует показывать смещением[2] этого оператора относительно того оператора, которому он принадлежит;
- при написании программы на C++ следует фигурные скобки располагать на отдельных строчках и друг под другом.

Следует отметить, что преимущества структурированного программирования убедительно сказываются только на больших программах.

[2] Обычно сдвиг оператора производится на две-четыре позиции

Объектно-ориентированное программирование

Если приемы процедурного программирования концентрируются на алгоритмах, используемых для решения задачи, и не обращается внимание на структуры данных, то ООП концентрируется на сути задачи. Элементы программы разрабатываются в соответствии с объектами, присутствующими в описании задачи. При этом первичными считаются объекты (данные), которые могут активно взаимодействовать друг с другом с помощью механизма передачи сообщений (называемого также и механизмом вызова методов). Функция программиста — придумать и реализовать такие объекты, взаимодействие которых после старта программы приведет к достижению необходимого конечного результата.

ООП включает в себя следующие концепции:

1. Наличие типов, определенных пользователем.
2. Скрытие деталей реализации (инкапсуляция).
3. Использование кода через наследование.
4. Разрешение интерпретации вызова функции во время выполнения программы (полиморфизм).

На основании выделенных объектов разрабатываются новые типы данных, которые называются *классами*.

Событийно-ориентированное программирование

Событийно-ориентированное программирование (СОП)**, создание событийно-управляемых программ (event-driven programming)** — парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета)[3].

СОП можно также определить, как способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события.

Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. В связи с этим

[3] Хорошо использовать UML.

при написании событийно-ориентированных программ часто применяют *автоматное программирование*.

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

- построение пользовательских интерфейсов (в том числе графических);
- создание серверных приложений;
- моделирование сложных систем;
- параллельные вычисления;
- автоматические системы управления, SCADA;
- программирование игр, в которых осуществляется управление множеством объектов.

Разные языки программирования поддерживают СОП в разной степени. Наиболее полной поддержкой событий обладают следующие языки (неполный список):

- Perl (события и демоны DAEMON, и их приоритеты PRIO),
- PHP,
- Java,
- Delphi (язык программирования),
- ActionScript 3.0,
- C# (события event),
- JavaScript (действия пользователя).

Остальные языки, в большей их части, поддерживают события как обработку исключительных ситуаций.

Визуальное программирование

Визуальное программирование является в настоящее. время одной из наиболее популярных парадигм программирования. Визуальное программирование состоит в автоматизированной разработке программ с использованием особой диалоговой оболочки. Рассматривая системы визуального программирования, легко увидеть, что все они базируются на объектно-ориентированном программировании и являются его логическим продолжением. Наиболее часто визуальное программирование используется для создания интерфейса программ и систем управления базами данных.

С объектно-ориентированными системами ассоциируется программа *Browser*. Это средство вместе с системой экранных подсказок позволяет программисту по желанию просматривать некоторые части программного окружения и видеть весь проект уже созданной

программы. Под проектом программы здесь понимается структура программы — состав файлов, объектов и их порождающих классов, которые слагают программу в целом.

Одну из ключевых возможностей программы *Browser* предоставляет окно, в котором находится список всех классов системы. При выборе одного из классов в специальных окнах отображаются его локальные функции и переменные. Затем при выборе одного из методов на отдельной панели высвечивается его код. Обычно в системе присутствуют средства для добавления и удаления классов из проекта. Программа *Browser* — это не просто визуализатор. Это основной, интегрирующий инструмент, который помогает одновременно рассматривать существующую систему и разрабатывать документацию программного проекта.

Структурной единицей визуального программирования в *Delphi* и *C++ Builder* является компонента. Компонента представляет собой разновидность объекта, который можно перенести (агрегировать) в приложение из специальной *Палитры компонент*. Компонента имеет набор свойств, которые можно изменять, не изменяя исходный код программы.

Компоненты бывают *визуальными* и *невизуальными*. Первые предназначены для организации интерфейса с пользователем. Это различные кнопки, списки, статический и редактируемый текст, изображения и многое другое. Эти компоненты отображаются при выполнении разрабатываемого приложения. *Невизуальные* компоненты отвечают за доступ к системным ресурсам: драйверам баз данных, таймерам и т.д. Во время разработки они отображаются своей пиктограммой, но при выполнении приложения, как правило, невидимы. Компонента может принадлежать либо другой компоненте, либо форме.

Можно отметить следующее:

- Визуальное программирование во многом автоматизирует труд программиста по написанию программ.
- Визуальное программирование — одна из самых популярных парадигм программирования на данный момент. Оно базируется на технологии ООП.
- Среда визуального программирования поддерживает работу браузеров (*Browser*), при помощи которых можно автоматически получить документацию по структуре программы.

- Основным элементом в средствах визуального программирования является компонент. Компоненты бывают визуальными и невизуальными.
- Технология визуального программирования состоит в следующем: создание экранных форм, нанесение визуальных и невизуальных компонент, программирование событий и методов оконных форм.

Элементы модульного программирования

Развитие модульного программирования было обусловлено:

- возрастающими объемами программ;
- увеличивающейся внутренней сложностью;
- коллективным характером разработок.

В общем случае МОДУЛЬ представляет собой совокупность программных ресурсов, предназначенных для использования другими модулями и программами. Программные ресурсы — это: константы, типы, подпрограммы, переменные.

Под МОДУЛЬНЫМ программированием понимается процесс разработки программы из нескольких логически завершенных единиц — модулей.

Будем считать, что модуль — файл Си-программы, транслируемый независимо от других файлов (модулей).

Принцип МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ — представление текста программы в виде нескольких файлов, каждый из которых транслируется отдельно.

С модульным программированием мы сталкиваемся в двух случаях:

- когда сами пишем модульную программу;
- когда используем стандартные библиотечные функции.

Модуль сам по себе не является выполнимой программой. Чтобы можно было использовать модуль, его необходимо подключить.

1. **Подключение**

Включение файлов/модулей в Borland C++ производится с помощью команды препроцессора #include :

```
#include <stdlib.h>
#include "matr.h"
```

В первом случае поиск файла начинается с папки/директории, указанной в командной строке компиляции, а во втором случае — поиск в директории, содержащей текущий файл.

2. **Условная компиляции**

В С++ предусмотрена возможность избирательно копировать части файла в зависимости значения некоторого константного выражения:

```
#if B1
// компилируется, если B1— истина
#elif B2
//компилируется, если B2— истина, B1— ложь .
#elif B3
// компилируется, если B3— истина, B1и B2— ложь
#else
//компилируется, если B1, B2, B3 — ложь
#endif
```

Здесь:

- ✓ для каждого #if д.б. обязательно #endif ;
- ✓ #else д.б. единственным и находиться перед #endif;
- ✓ значения B1, B2 , B3 д.б. целой константой или константным выражением, которое может содержать препроцессорную операцию с именем defined;
- ✓ операция defined позволяет проверить, определен или нет идентификатор в данный момент;
- ✓ процессор выбирает один из участков текста для обработки;
- ✓ участок текста, не выбранный препроцессором для обработки, игнорируется и не компилируется.

3. Реализация модульного программирования в Borland C++

В С++ с каждым модулем связаны, по крайней мере, два файла. Это - *.h - хэдер-файл, содержащий интерфейс представляемых ресурсов, и *.cpp - файл, содержащий описания или реализации функций представленных ресурсов.

Имена *.h и *.cpp — одинаковые и, как правило, совпадают с именем класса, определение которого они содержат. При формировании хэдер-файла обычно создают именную константу — псевдопеременную, которая становится известной или определенной при компиляции.

Псевдопеременные - представляют собой зарезервированные именные константы, которые можно использовать в любом исходном файле. Каждая из них начинается и оканчивается двумя символами подчеркивания: __stdio_h. __stdlib_h, __iostream_h. Псевдопеременная определяется с помощью команды препроцессора #define.

Пример файла описаний в С++ с именем Matr.h

```
#ifndef __Matr_H
#define __Matr_H
class Matr
{
} ;
#endif
```

Здесь операторы препроцессора #ifndef __Matr_H и #endif заставляют транслятор обходить данный файл в том случае, если он уже транслировался и известен программе. Оператор препроцессора # define __Matr_H определяет уникальный макрос (псевдопеременную) для данного *.h - файла, который станет определенным при компиляции. Теперь, сколько бы раз не появлялся #include "Matr.h", компиляция будет выполняться один раз.

Файл реализаций в С++ с именем Matr.cpp

Листинг

```
#if !defined ( __STDLIB_H)
```

```
#include <stdlib.h>
#endif
#if !defined ( __IOSTREAM_H)
#include <iostream.h>
#endif
#include "Matr.h"
// описание методов класса
Matr::Matr()
{
}
```

и и т.д.

Здесь операторы препроцессора, например, #ifndef __STDIO_H и #endif заставляют транслятор обходить определение данного файла в том случае, если он уже известен программе. Если теперь вы захотите использовать класс из модуля Matr, то вам достаточно будет в головной программе подключить файл Matr.h при помощи #include "matr.h"
Предположим, что вы написали несколько модулей и хотите собрать их в единую программу. Здесь у вас есть выбор:

- воспользоваться файлом проекта;
- включить файл реализаций в головной файл, используя #include;
- поместить модули в библиотеку,

В первом случае вы средствами IDE формируете отдельный файл проекта, в котором перечисляете модули, входящие в проект, и их местоположение. Интегрированная среда Borland C++ будет автоматически транслировать модули проекта в объектные файлы и передавать их компоновщику.
Во втором случае все файлы перечисляются в инструкциях препроцессора #include.
В третьем случае надо создать вашу личную библиотеку, ее хедер-файл и включить их в файл проекта.
Первый и второй варианты предпочтительнее на этапе отладки, третий - когда модуль отлажен.

```
#ifndef Unit1H
#define Unit1H
#endif
```

Файл 3.

Технология создания модуля для хранения класса в C++ Builder

1) *Создать заготовку модуля:*
   File→New→UnitC++ Builder
   В результате появится шаблон-заготовки для двух файлов с именами «по умолчанию»:
   Unit.h - файла*
   Unit.cpp - файла*
2) *Сохранить новый модуль в папке проекта: по заданным именем (обязательно изменить!)*
   File→Save as…
   Как правило, имя модуля совпадает с именем класса, который в нем содержится.

Если посмотреть на заготовку, можно увидеть команду условной компиляции
#ifndef (if !defined - если не определено)
и команду определения

#define (то определить)
У этих команд в качестве параметра используется имя уникальной для данного модуля переменной, которое складывается из названия модуля и символа “H”.
*Оператор #ifndef позволяет проверить, определен ли идентификатор Unit*H в данный момент.*
*Оператор #define Unit*H определяет эту уникальную переменную для данного модуля, и она станет известной при компиляции.*
*#ifndef и #define заставляют транслятор обходить данный файл в том случае, если он транслировался и известен программе. Теперь, сколько бы раз не появлялся #include Unit*H, компилятор будет выполняться один раз. Оператор #endif ограничивает область действия #if и является необходимым для него.*