МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

**РЕФЕРАТ**

на тему:

«Отличие структурного программирования от объектно-ориентированного»

Выполнил:

студент I курса 6 группы

специальности ПОИТ Шимко А.А

Преподаватель:

Белодед Николай Иванович

**Оглавление**

Введение……………….…………….…….………………………………..…..3-5

Структурное программирование …………………….………………………..6-8

Объектно-ориентированное программирование……………………………9-12

Принципы, отличия объектно-ориентированного подхода от структурного…………………………………………………………………13-14

Заключение……………………………………………………………………….15

**Введение**

## Парадигмы программирования

Дисциплину, которая в дальнейшем стала называться программированием, зародил Алан Тьюринг в 1938 году. В 1945 он уже писал полноценные программы, которые работали на реальном железе.

Первый компилятор был придуман в 1951 году Грейс Хоппер (бабушка с татуировкой Кобола). Потом начали создаваться языки программирования.

Существует три основных парадигмы: **структурное**, **объектно-ориентированное** и **функциональное**. Интересно, что сначала было открыто функциональное, потом объектно-ориентированное, и только потом структурное программирование, но применяться повсеместно на практике они стали в обратном порядке.

Структурное программирование было открыто Дейкстрой в 1968 году. Он понял, что goto – это зло, и программы должны строиться из трёх базовых структур: последовательности, ветвления и цикла.

Объектно-ориентированное программирование было открыто в 1966 году.

Функциональное программирование открыто в 1936 году, когда Чёрч придумал лямбда-исчисление. Первый функциональный язык LISP был создан в 1958 году Джоном МакКарти.

Каждая из этих парадигм убирает возможности у программиста, а не добавляет. Они говорят нам скорее, что нам **не нужно делать**, чем то, что нам нужно делать.

Все эти парадигмы очень связаны с архитектурой. Полиморфизм в ООП нужен, чтобы наладить связь через границы модулей. Функциональное программирование диктует нам, где хранить данные и как к ним доступаться. Структурное программирование помогает в реализации алгоритмов внутри модулей.

## Структурное программирование

Дейкстра понял, что программирование – это сложно. Большие программы имеют слишком большую сложность, которую человеческий мозг не способен контролировать.

Чтобы решить эту проблему, Дейсктра решил сделать написание программ подобно математическим доказательствам, которые также организованы в иерархии. Он понял, что если в программах использовать только if, do, while, то тогда такие программы можно легко рекурсивно разделять на более мелкие единицы, которые в свою очередь уже легко доказуемы.

С тех пор оператора goto не стало практически ни в одном языке программирования.

Таким образом, структурное программирование позволяет делать функциональную декомпозицию.

Однако на практике мало кто реально применял аналогию с теоремами для доказательства корректности программ, потому что это слишком накладно. В реальном программировании стал популярным более «лёгкий» вариант: тесты. Тесты не могут доказать корректности программ, но могут доказать их некорректность. Однако на практике, если использовать достаточно большое количество тестов, этого может быть вполне достаточно.

## Объектно-ориентированное программирование

ООП – это парадигма, которая характеризуется наличием **инкапсуляции**, **наследования** и **полиморфизма**.

**Инкапсуляция** позволяет открыть только ту часть функций и данных, которая нужна для внешних пользователей, а остальное спрятать внутри класса.

Однако в современных языках инкапсуляция наоборот слабее, чем была даже в C. В Java, например, вообще нельзя разделить объявление класса и его определение. Поэтому сказать, что современные объектно-ориентированные языки предоставляют инкапсуляцию можно с очень большой натяжкой.

**Наследование** позволяет делать производные структуры на основе базовых, тем самым давая возможность осуществлять повторное использование этих структур. Наследование было реально сделать в языках до ООП, но в объектно-ориентированных языках оно стало значительно удобнее.

Наконец, **полиморфизм**позволяет программировать на основе интерфейсов, у которых могут быть множество реализаций. Полиморфизм осуществляется в ОО-языках путём использования виртуальных методов, что является очень удобным и безопасным.

Полиморфизм – это **ключевое свойство ООП** для построения грамотной архитектуры. Он позволяет сделать модуль независимым от конкретной реализации (реализаций) интерфейса. Этот принцип называется **инверсией зависимостей**, на котором основаны все плагинные системы.

Инверсия зависимостей так называется, что она позволяет изменить направление зависимостей. Сначала мы начинаем писать в простом стиле, когда высокоуровневые функции зависят от низкоуровневых. Однако, когда программа начинает становиться слишком сложной, мы инвертируем эти зависимости в противоположную сторону: высокоуровневые функции теперь зависят не от конкретных реализаций, а от интерфейсов, а реализации теперь лежат в своих модулях.

**Любая зависимость всегда может быть инвертирована**. В этом и есть мощь ООП.

Таким образом, между различными компонентами становится меньше точек соприкосновения, и их легче разрабатывать. Мы даже можем не перекомпилировать базовые модули, потому что мы меняем только свой компонент.

**Структурное программирование**

Структурное программирова­ние- это первая законченная методология программирования. Структурное программирование оказало огромное влияние на раз­витие программирования. Этот метод применялся очень широко в практическом программировании и по сей день не поте­рял своего значения для определенного класса задач.

Структурный подход базируется на двух основополагающих принципах:

1. использование процедурных языков про­граммирования (Алгол, Паскаль, СИ);
2. последовательная декомпозиция ал­горитма решения задачи сверху вниз.

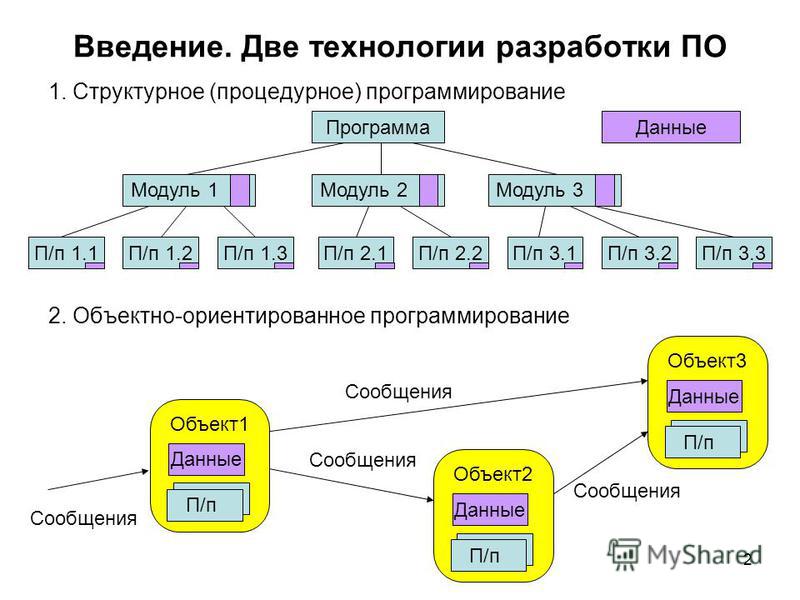
Задача решается применением последовательности действий. Первоначально задача формулируется в терминах входа-выхода. Это означает, что на вход программы подаются некоторые данные. Программа работает и выдает ответ.

После этого начинается последовательное разложение всей за­дачи на более простые действия.

Очень важно, что на любом этапе программу можно проверить. Для этого нужно написать заглушки - процедуры, имитирующие вход и выход процедур нижнего уровня. (В приве­денной выше программе можно использовать процедуру чтения адреса, которая вместо ввода с терминала просто подставляет ка­кой-нибудь фиксированный адрес, и процедуру сверки с базой данных, которая ничего не делает, а просто всегда возвращает истину.)

Программа компонуется с заглушками и может работать. Заглушки позволяют проверить логику верхнего уровня до реализации следующего уровня.

Структурное программирование ясно опре­делило *значение модульного построения программ*при разработке больших проектов, но языки программирования еще слабо поддержи­вали модульность. Единственным способом структуризации про­грамм являлось составление ее из подпрограмм или функций. Контроль за правильностью вызова функций, в том числе соответствия количества и типов фактических аргументов ожидаемым формальным параметрам, осуществлялся только на стадии выполнения (понятие прототипа функции появилось позже).



Выполнение оператора начинается с вычисления выражения (оно должно быть целочисленным), а затем управление передается первому оператору из списка, помеченного константным выражением, значение которого совпало с вычисленным. После этого, если выход из переключателя явно не указан, последовательно выполняются все остальные ветви.

Выход из переключателя обычно выполняется с помощью операторов **break** или **return**. Оператор break выполняет выход из самого внутреннего из объемлющих его операторов switch, for, while и do. Оператор return выполняет выход из функции, в теле которой он записан.

Все константные выражения должны иметь разные значения, но быть одного и того же целочисленного типа. Несколько меток могут следовать подряд. Если совпадения не произошло, выполняются операторы, расположенные после слова default (а при его отсутствии управление передается следующему за switch оператору).

Пример (программа реализует простейший калькулятор на 4 действия):

|  |  |
| --- | --- |
| Исходный код | Блок-схема |
| #include<iostream>  int main() {  using namespace std;  setlocale(LC\_ALL, "Russian");  int a, b, res;  char op;  cout << "\nВведите 1й операнд : "; cin >> a;  cout << "\nВведите знак операции : "; cin >> op;  cout << "\nВведите 2й операнд : "; cin >> b;  bool f = true;  switch (op) {  case '+': res = a + b; break;  case '-': res = a - b; break;  case '\*': res = a \* b; break;  case '/': res = a / b; break;  default: "nНеизвестная операция"; f = false;  }  if (f) cout << "\nРезультат: " << res;  return 0; |  |

**Объектно-ориентированное программирование**

*Объектно-ориентированное программирование* появилось и по­лучило широкое распространение ввиду осознания трех важнейших проблем программирования.

1 - раз­витие языков и методов программирования не успева­ло за растущими потребностями в про­граммах. Единственным реальным способом удовлетворить эти потребности был метод многократ­ного использования уже разработанного, протестированного и отлаженного программного обеспечения.

2 - необходимость упрощения сопровождения и модификации разра­ботанных систем. (Факт постоянного изменения требований к системе был осознан как нормальное условие развития системы, а не как неумение или недостаточно четкая организация работы разра­ботчиков.) Требовалось изменить способ построения программных систем так, чтобы локальные модификации не нарушали работоспособность всей системы и было легче производить изменения поведения системы.

3 - это облегчение проектирования систем. Далеко не все задачи поддаются алгоритмическому описанию и тем более алгоритмической декомпозиции, как того требует структур­ное программирование. Требовалось приблизить структуру программ к структуре решаемых задач и сократить так называемый *семантический разрыв*между структурой решаемой задачи и структурой программы. Семантический разрыв имеет место в том случае, когда понятия, лежащие в основе языка задачи и средств ее реше­ния, различны. Поэтому наряду с необходимостью записи самого решения требуется еще перевести одни понятия в другие. (Сравните это с переводом с одного естественного языка на другой. Именно потому, что таких понятий в русском языке раньше не было, появляются слова типа *брокер, оффшор*или *инвестор.*К сожалению, в программировании заимствование слов невозможно.)

Итак, упрощение проектирования, ускорение разработки за счет многократного использования готовых модулей и легкость модифика­ции - вот три основных достоинства объектно-ориентированного программирования, которые пропагандировались его сторонниками.

Объектно-ориентированное программирование предполагает единый подход к проектированию, построению и развитию системы.

Появление объектно-ориентированного метода произошло на основе всего предыдущего развития методов разработки программного обеспечения, а также многих других отраслей науки.

Возникновению объектно-ориентированного подхода к проектированию систем способствовали следующие достижения технологии:

1 - Развитие вычислительной техники, в частности аппаратная поддержка основных концепций операционных систем и построение функционально-ориентированных систем.

2 - Достижения в методологии программирования, в частности модульное построение систем и инкапсуляция информации.

3 - Теория построения и моделирования систем управления базами данных внесла в объектное программирование идеи построения отношений между объектами.

4 - Исследования в области искусственного интеллекта позволили лучше осознать механизмы абстракции. Теория фреймов, предложенная Минским для представления реальных объектов в системах распознавания образов, дала мощный импульс не только системам искусственного интеллекта, но и механизмам абстракции в языках программирования.

5 - Развитие философии и теории познания. Во многом объектно-ориентированное построение систем - это определенный взгляд на моделируемый реальный мир. Именно в этом аспекте философия и теория познания оказали сильное влияние на объектную модель. Еще древние греки рассматривали мир в виде объектов или процессов. Декарт выдвинул предположение, что для человека естественным представляется объектно-ориентированное рассмотрение окружающего мира. Минский предположил, что разум прояв­ляется как взаимодействие агентов, не умеющих по отдельности мыслить.

Объектно-ориентированный подход к программированию поддерживается языками программирования, такими, как Смолток, Си++, Java и т.д. Языки являются главными инструмента­ми объектно-ориентированного программирования, именно при их разработке появилось большинство тех идей, которые и со­ставляют основу объектно-ориентированного метода.

Первым шагом на пути создания объек­тной модели следует считать появление абстрактных типов дан­ных. Считается, что первой полной реализацией абстрактных ти­пов данных в языках программирования является язык Симула.

Затем появились (и продолжают появляться) другие объектно-ориентированные языки, которые определяют современное состоя­ние программирования. Наиболее распространенными из них стали Си++, CLOS, Эйффель, Java.

Рассмотрим реализацию комплексного числа, вещественная и мнимая части которого являются объектами встроенного типа double. Коллекция операций состоит из четырех операций комплексной арифметики (сложение, вычитание, умножение и деление), а также операций инициализации и визуализации комплексных чисел. Например, вычислим значение выражения : 

// C++ Абстрактный тип данных - комплексное число

#include <iostream>

using namespace std;

struct Complex

{

double re; // вещественная часть комплексного числа

double im; // мнимая часть комплексного числа

};

// Инициализация комплексного числа

void define(Complex& c, double r = 0.0, double i = 0.0)

{

c.re = r;

c.im = i;

}

// Сложение комплексных чисел

Complex add(Complex a, Complex b)

{

Complex temporary;

temporary.re = a.re + b.re;

temporary.im = a.im + b.im;

return temporary;

}

// Вычитание комплексных чисел

Complex subtract(Complex a, Complex b)

{

Complex temporary;

temporary.re = a.re - b.re;

temporary.im = a.im - b.im;

return temporary;

}

// Умножение комплексных чисел

Complex multiply(Complex a, Complex b)

{

Complex temporary;

temporary.re = a.re \* b.re - a.im \* b.im;

temporary.im = a.re \* b.im + b.re \* a.im;

return temporary;

}

// Деление комплексных чисел

Complex divide(Complex a, Complex b)

{

Complex temporary;

double divider = b.re \* b.re + b.im \* b.im;

temporary.re = (a.re \* b.re + a.im \* b.im) / divider;

temporary.im = (b.re \* a.im - a.re \* b.im) / divider;

return temporary;

}

// Визуализация комплексного числа

void print(Complex c)

{

cout << '(' << c.re << ", " << c.im << ')' << endl;

}

int main()

{

Complex x1, y1, z1, x2, z2;

// Инициализация операндов выражения - комплексных чисел

define(x1, -1, 5); // x1 = -1 + 5i

define(y1, 3, -4); // y1 = 3 - 4i

define(z1, 1, 3); // z1 = 1 + 3i

define(x2, 10, 7); // x2 = 10 + 7i

define(z2, 0, 5); // z2 = 5i

// Визуализация вычисленного значения выражения

print(add(divide(multiply(multiply(x1, x1), y1), z1),

divide(x2, z2)));

return 0;

}

**Принципы, отличия объектно-ориентированного подхода от структурного**

Объектно-ориентированныйподход - разрабатывает программное обеспечение, основанное на моделировании объектов реального мира и фокусирующее внимание на объекте, представляет собой целостную модель и естественную имитацию деятельности некоторого элемента реального мира.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. Объектно-ориентированный подход является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

Три основных принципа языков объективно-ориентированного программирования. Язык может называться объектно-ориентированным, если в нем реализованы три концепции: ***объекты, классы*** и ***наследование.*** Однако теперь принято считать, что такие языки должны держаться на других трех китах: ***инкапсуляции, наследовании***и ***полиморфизме****.*Этот философский сдвиг произошел из-за того, что со временем стали понимать, что построить объектно-ориентированные системы без инкапсуляции и полиморфизма так же невозможно, как без классов и наследования.

Отличия объектно-ориентированного подхода от структурного:

*Первое отличие* этих подходов друг от друга заключается в принципах декомпозиции и структурной организации элементов (компонентов, модулей) системы. Согласно этим принципам система представляет собой структуру, состоящую из четко выраженных модулей, связанных между собой определенными отношениями.

При использовании структурного подхода (первый вид декомпозиции), выполняется функциональная (процедурная, алгоритмическая) декомпозиция системы. Т. е. она представляется в виде иерархии (дерева) взаимосвязанных функций. На высшем уровне система представляется единым целым с наивысшей степенью абстракции и по мере детализации (добавления уровней) разбивается на функциональные компоненты с более конкретным содержанием.

Второй вид декомпозиции – объектно-ориентированный. В рамках этого подхода система разбивается на набор объектов, соответствующих объектам реального мира, взаимодействующих между собой путем посылки сообщений.

*Вторым отличием* является объединение в объекте как атрибутивных данных (характеристики, свойства), так и поведения (функции, методы). В функционально-ориентированных системах функции и данные хранятся (существуют) отдельно.

*Третье отличие* двух подходов заключается в структурной организации внутри модулей системы. В структурном подходе модуль состоит из функций, иерархически связанных между собой отношением композиции (англ. part-of – часть-целое), т. е. функция состоит из подфункций, подфункция из подподфункций и т.д. В объектно-ориентированном подходе иерархия выстраивается с использованием двух отношений: композиции и наследования (англ. is-a – это есть). При этом в объектно-ориентированном подходе «объект-часть» может включаться сразу в несколько «объектов-целое». Таким образом, модуль в структурном подходе представляется в виде дерева, а в объектно-ориентированном подходе – в виде ориентированного графа, т. е. с помощью более общей структуры.

**Заключение**

Применявшийся ранее структурный подход к программированию сейчас практически вытеснен объектно-ориентированным. Тем не менее, для простых программ не требующих работы с большим кол-вом разнообразных данных, он до сих про является эффективным благодаря простоте и малой нагруженности синтаксическими конструкциями, которыми изобилует объектно-ориентированный подход. Также для написания структурного кода необходимо гораздо меньше познаний в языке программирования, соответственно он лучше подходит для людей только начавших изучать кокой-либо язык программирования.

Объектно-ориентированный подход – это подход широко применяемый в масштабной разработке. Благодаря ему код одной программы могут независимо создавать несколько программистов. Он существенно упрощает разбор кода больших программ, выполняющих несколько разных задач.

Выбор подхода программирования существенно зависит от задачи, нет смысла применять объектно-ориентированный подход в простой программе, ведь это приведет лишь к увеличению количества кода. Так же бессмысленным является применение структурного программирования при создании фоторедактора(или аналогичной по сложности программы).