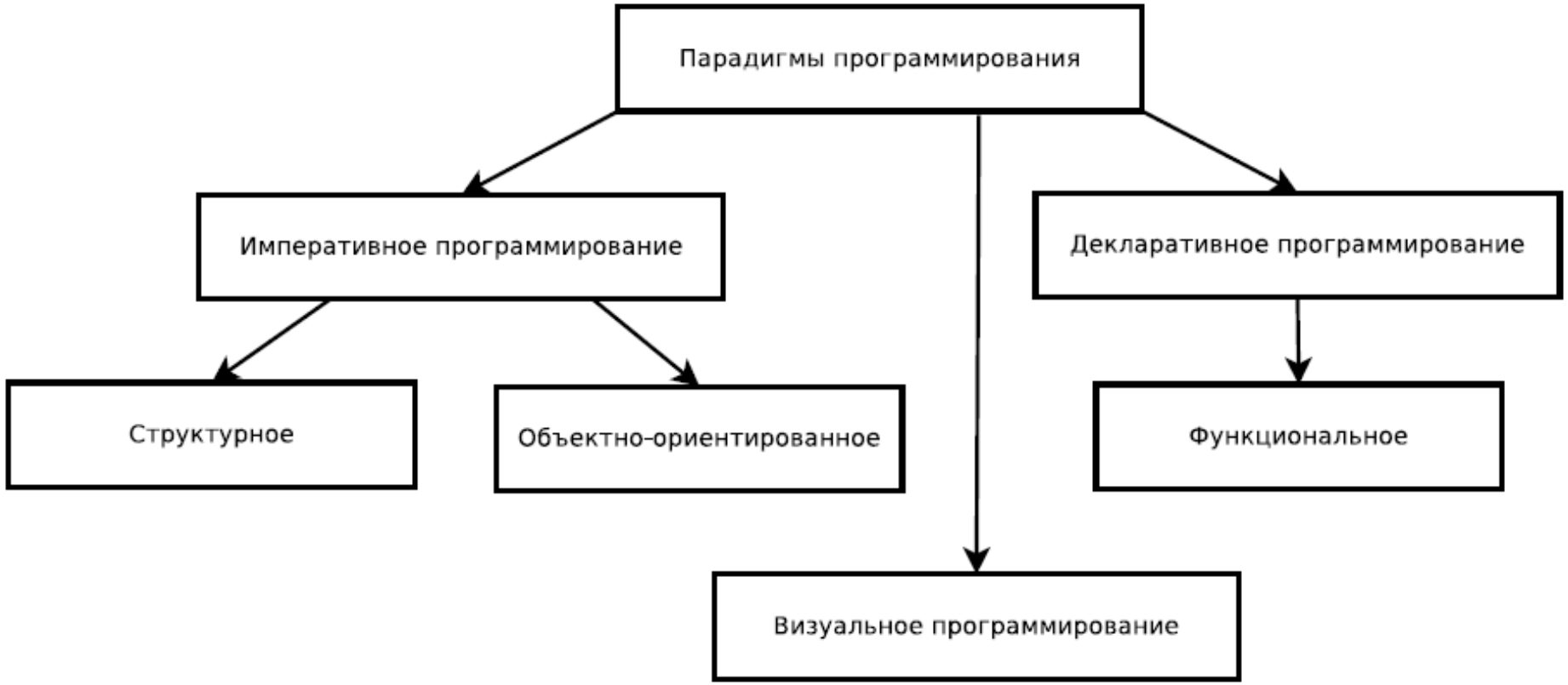
Парадигмы программирования

Определение Роберта Мартина

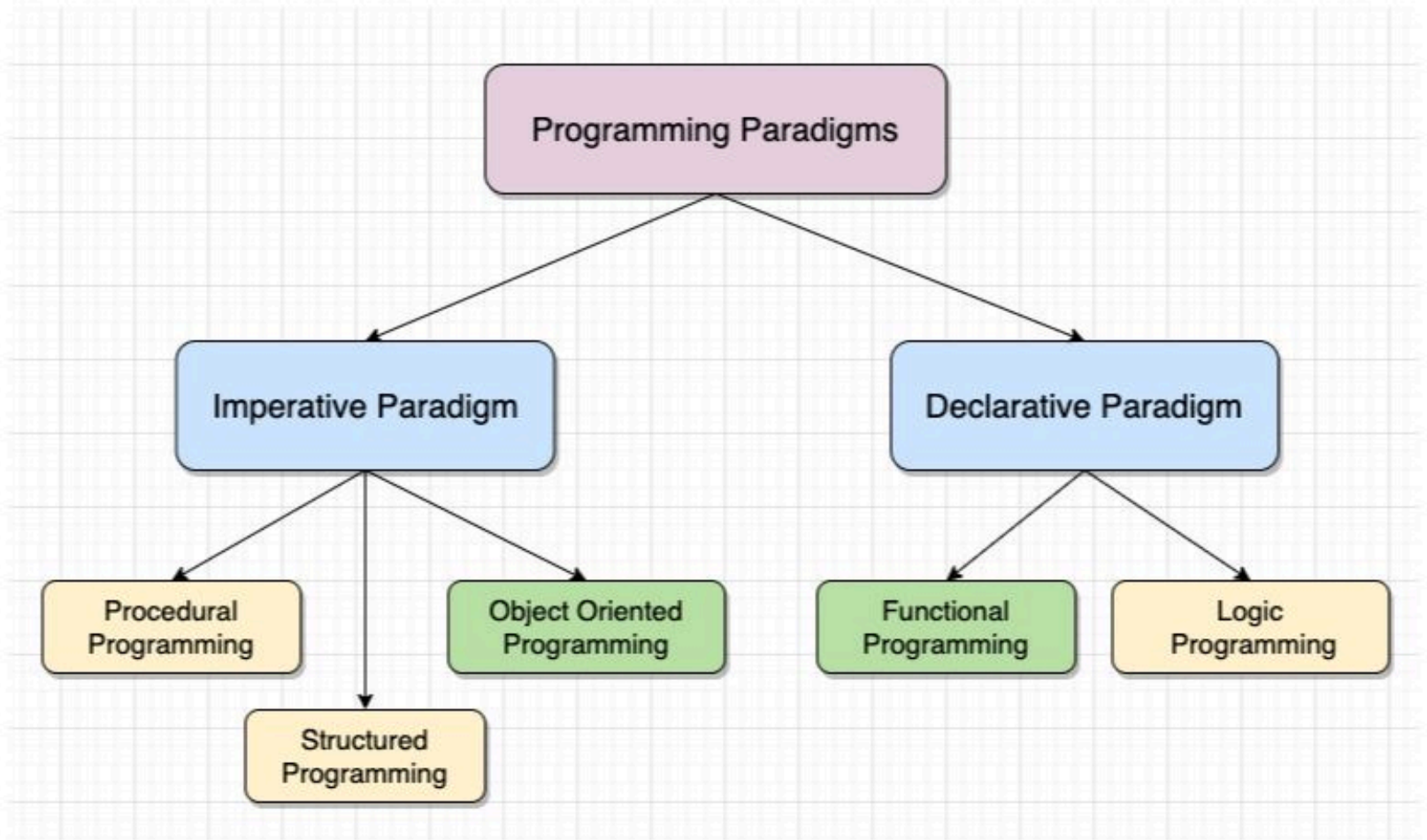
Роберт Мартин — международный консультант в области разработки, известный среди разработчиков как дядя Боб.

С его точки зрения, парадигмы — это ограничения на определённые языковые конструкции, которые вынуждают использовать определённый стиль.

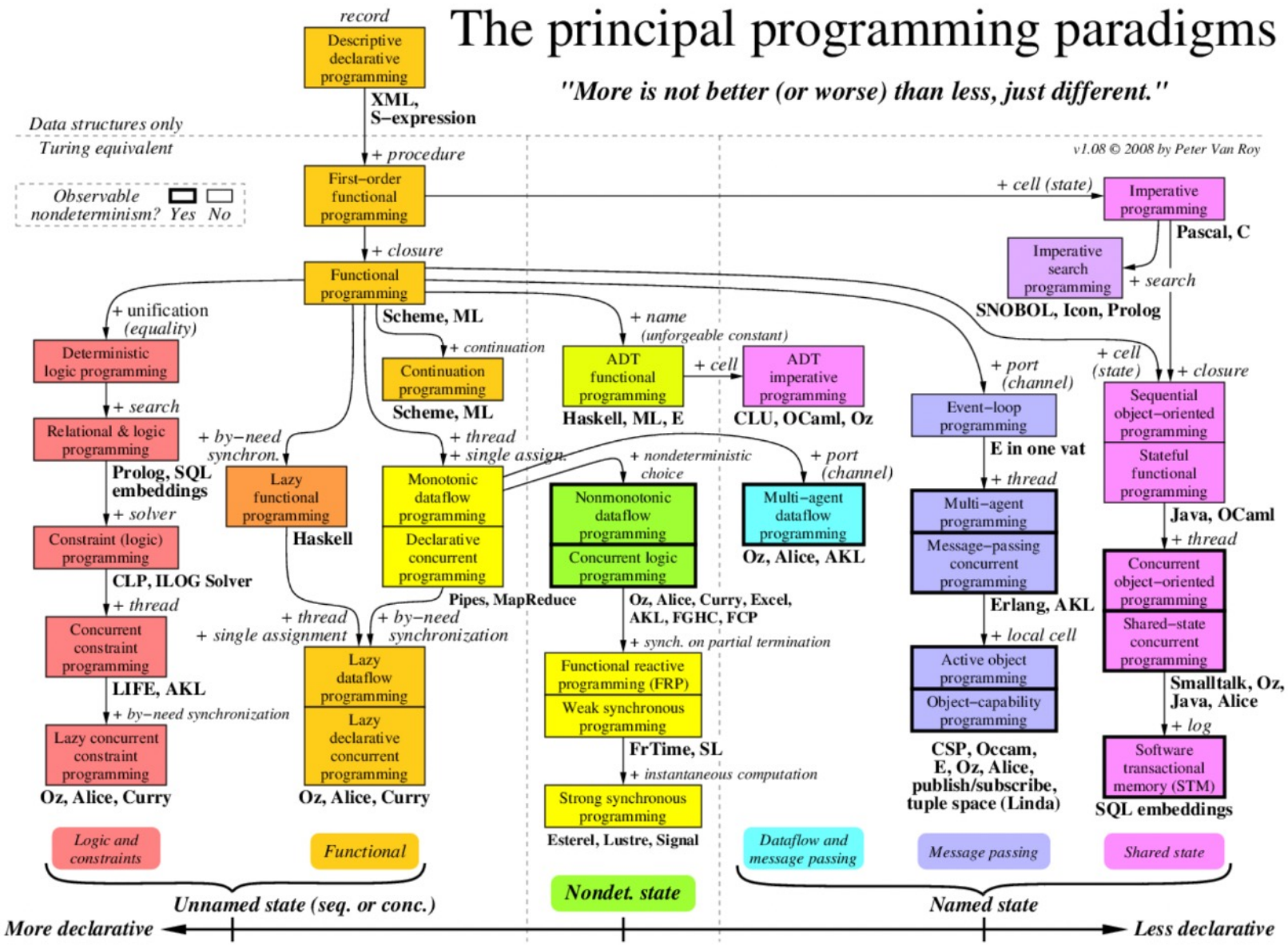
Классификация (первый вариант)



Классификация (очередной вариант)



Классификация (для тех, кто хочет 5)



Парадигмы с точки зрения истории

1. Императивная
2. Структурная
3. Модульная
4. Объектно-ориентированная
5. Декларативная
6. Функциональная
7. Реактивная
8. и многие многие другие менее известные...

Появление определенной парадигмы тесно связано с появлением определенного языка программирования.

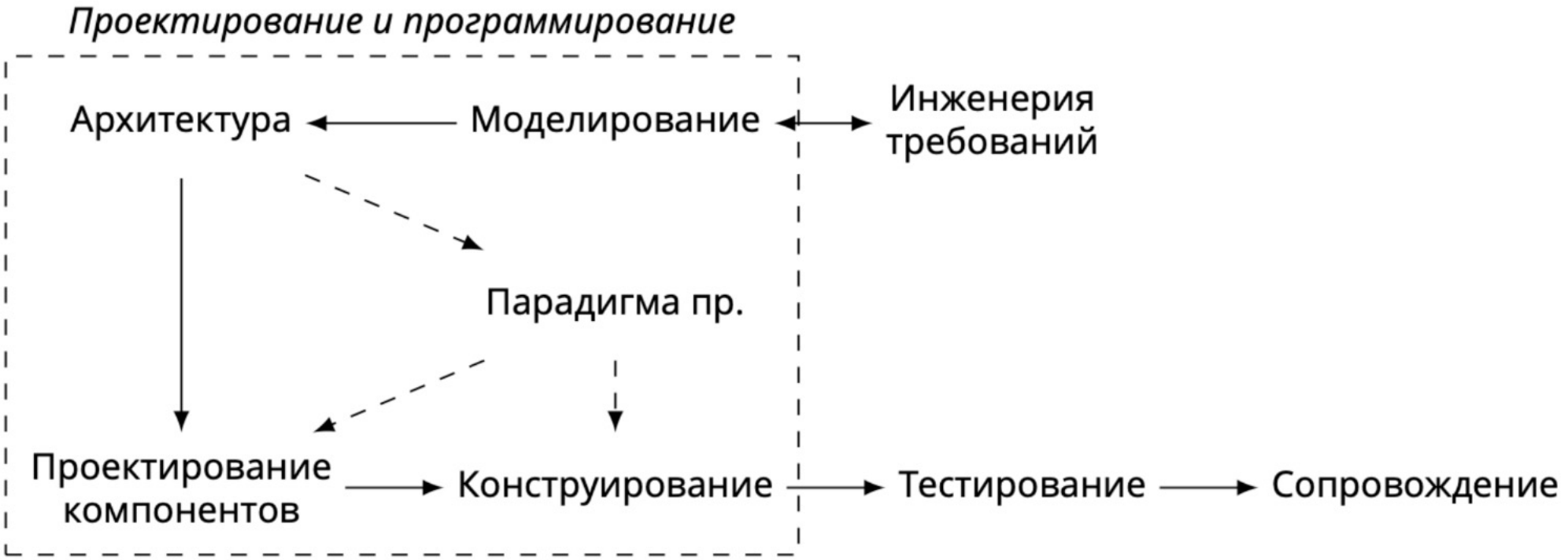
Некоторые критерии качества программ:

– результативность;
– надежность;
– устойчивость;
– автоматизируемость;
– эффективное использование ресурсов (время, память, устройства, информация, люди);
– удобство разработки и применения;
– наглядность текста программы;
– наблюдаемость процесса работы программы;
– диагностика происходящего.

Непрерывное развитие пространства, в котором решается задача, вводит дополнительные требования к **стилю программирования** информационных систем:

– гибкость;
– модифицируемость;
– верифицируемость;
– безопасность;
– мобильность/переносимость;
– адаптируемость;
– конструктивность;
– измеримость характеристик и качества;
– улучшаемость.

Место парадигмы в разработке



Парадигма выбирается исходя из архитектуры системы и влияет на построение и реализацию ее компонентов.

Императивная парадигма

Императивная парадигма программирования - это стиль написания кода, подразумевающий *подробное описание алгоритма* решения задачи и *получения искомых данных (ответ на вопрос КАК?)*.

В императивном стиле:

1. **Описывают** алгоритм решения поставленной задачи.

2. **Сохраняют** состояния в переменных.

3. Зачастую **снабжают** код комментариями, поскольку по самому коду бывает сложно понять его итоговую цель.

Эта парадигма популярна потому, что она в точности соответствует тому, как работает компьютер: **последовательно выполняет инструкции** и **использует память для хранения промежуточных результатов**.

Процедурная парадигма

Процедурная парадигма подразумевает **написание алгоритмов** программ пошагово и **частое использование подпрограмм**, называемых процедурами

Процедуры – практически синонимы функций. Единственная разница их в том, что процедуры просто выполняют какие-то действия, ничего не возвращая, а функции могут и должны возвращать значения.

Структурная парадигма

Структурная парадигма программирования запрещает использование оператора goto, настоятельно рекомендуя использовать другие подходы:

1. **последовательность** (выполнение программы *сверху вниз*),
2. **ветвление** (при помощи *условий*);
3. **цикличность** (при помощи операторов *циклов*).

В то же самое время развивалась **модульная парадигма**.

Модульная парадигма

- **Модульное парадигма** — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам.
- По модульной парадигме, необходимо размещать самодостаточный код в отдельные модули (чаще всего, отдельные файлы), которые могут взаимодействовать между собой.

Декларативная парадигма

Декларативная парадигма программирования представляет собой стиль описания того, **что** вы хотите увидеть в итоге (ответ на вопрос ЧТО).

В декларативном стиле:

1.не пишут, как решить задачу, но пишут, что требуется получить;

2.не сохраняют промежуточные состояния;

3.пишут так, чтобы по виду кода можно было понять его цель.

К декларативным парадигмам часто относят:

- **функциональную**;
- **логическую**.

Функциональная парадигма

- **Функциональное парадигма** — это парадигма декларативного программирования, в которой программы создаются путем последовательного применения функций, а не инструкций.

- Каждая из этих функций принимает **входное** значение и возвращает согласующееся с ним **выходное** значение, **не изменяясь** и **не подвергаясь** воздействию со стороны состояния программы.

Чистые функции

Чистые функции не производят побочных эффектов и не зависят от глобальных переменных или состояний.

```
int sqr(int x)
{
    return x * x;
}
```

Если функция изменяет любую память, за исключением собственного стека, она уже не может считаться чистой.

```
int dirty5(int *x)
{
    return *x;
}
```

Функциональная парадигма

Есть ряд определений, которые делают функциональное программирование отдельной парадигмой и стилем создания алгоритмов:

•Функция или идентифицированный блок кода;
•Чистые функции;
•Лямбда функции;
•Функции высшего порядка;
•Композитные функции;
•Константы;
•Замыкания;
•Рекурсия.
Боле подробно здесь

Преимущества ФП

- Простота в отладке
- Ленивые вычисления
- Деление программы на модули
- Облегченная читаемость кода
- Многопоточность

Логическая парадигма

Логическое программирование – это парадигма программирования, которая в значительной степени основана на формальной логике.

Любая программа, написанная на языке логического программирования, представляет собой **набор предложений** в логической форме, выражающих **факты** и **правила** о некоторой проблемной области.

Отличие декларативного от императивного подхода

Императивный подход использует последовательность утверждений, меняющих состояние программы для достижения результата. При декларативном же подходе описывается результат, который необходимо достичь. При этом без указания шагов, необходимых для его получения.

Переход к ООП

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленного программного продукта. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Критерии качества декомпозиции проекта

Для оценки качества программного проекта нужно учитывать, кроме всех прочих, следующие два показателя:

- **Сцепление** (*cohesion*) внутри компонента — показатель, характеризующий степень взаимосвязи отдельных его частей.
- **Связанность** (*coupling*) между компонентами — показатель, описывающий интерфейс между компонентом-клиентом и компонентом-сервером. Общее число входов и выходов сервера есть мера связанности. Чем меньше связанность между двумя компонентами, тем проще понять и отслеживать в будущем их взаимодействие.

Класс

Класс — это тип, определяемый программистом, в котором объединяются структуры данных и функции их обработки. Конкретные переменные типа данных "класс" называются экземплярами класса, или объектами. Программы, разрабатываемые на основе концепций ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Класс содержит константы и переменные, называемые полями, а также выполняемые над ними операции и функции. Функции класса называются методами. Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Поля и методы являются элементами, или компонентами класса.

Основные принципы ООП

Четыре основных принципа объектно-ориентированного программирования следующие:

- **Абстракция**. Моделирование требуемых атрибутов и взаимодействий сущностей в виде классов для определения общего представления системы.
- **Инкапсуляция**. объединении данных и методов, работающих с этими данными, в единый компонент (класс) и ограничении прямого доступа к ним извне
- **Наследование**. Возможность создания новых классов на основе существующих.
- **Полиморфизм**. Способность объектов разных наследуемых классов по-разному реагировать на одноимённые вызовы методов.

POD (Plain Old Data)

- **Отсутствие сложной логики:** Нет пользовательских конструкторов, деструкторов или операторов присваивания.
- **Структура данных:** Нет виртуальных методов, виртуальных базовых классов и закрытых/защищенных нестатических полей.
- **Совместимость с C:** Объекты POD-типа имеют точно такое же представление в памяти, как и соответствующие структуры в языке C.

Если класс или структура является тривиальным типом и типом стандартной структуры — это тип POD (обычные старые данные). Таким образом, распределение памяти для типов POD является непрерывным, и адрес каждого члена выше, чем адрес члена, объявленного до него, что дает возможность выполнять побайтовое копирование и двоичный ввод-вывод для этих типов. Скалярные типы, такие как int, также являются типами POD. Типы POD, которые являются классами, могут содержать только типы POD в качестве нестатических членов данных.

Пример

```
// Trivial but not standard-layout
struct C {
   int a;
private:
   int b;   // Different access control
};


// Standard-layout but not trivial
struct D {
   int a;
   int b;
   D() {} //User-defined constructor
};


struct POD {
   int a;
   int b;
};
```

Классы и структуры

```
struct S {
    int x = 1;
    double d = 3.14;
    void sum (int y) {
        std::cout << x + y << ' ';
        diff();
    }
    void diff () {
        std::cout << d - x;
    }
    int mult (int x);
};
int S::mult (int x) {
    return this->x * x;
}
int main () {
    S s;
    std::cout << s.x;              // 1
    std::cout << sizeof(s);        // 16 (4 int + выравнивание + 8 double)
    S s1{2, 3.12};                 // ok
    s1.sum (5);                    // 7 1.12
}
```

```
struct S {
    int x = 1;
    double d = 3.14;
    struct A {
        char ch;
    };
    char result () {
        A* pa = new A;
        pa->ch = 'A';
        return pa->ch;
    }
};


int main () {
    S s;
    std::cout << s.x << "\n";          // 1
    std::cout << s.result();           // A
}
```

Указатель this

```
struct Point {
    int x = 5;
    int y = 4;

    void showCoords() {
        std::cout << "Coords x: " << x << "\t y: " << y << std::endl;
    }
    void move(int x, int y) {
        this->x += x;
        this->y += y;
    }

};

int main () {
    Point A;
    A.showCoords();    //Coords x: 5     y: 4
    A.move(6, 7);
    A.showCoords();    //Coords x: 11     y: 11
}
```

Модификаторы доступа

private – нельзя обращаться к полям и методам класса извне

public – можно обращаться

```
class S {
    int x = 1;
    double d = 3.14;
};

int main () {
    S s;
    std::cout << s.x <<"\n";

}
```

'x' is a private member of 'S'

К закрытым полям можно обращаться из открытых методов.

Модификаторы доступа у структуры

```
struct Point {
    void showCoords() {
        std::cout << "Coords x: " << x << "\t y: " << y << std::endl;
    }
    Point &move(int x, int y) {
        this->x += x;
        this->y += y;
        return *this;
    }
  private:
    int x = 5;
    int y = 4;
};

int main () {
    Point A;
    Point B;
    A.x = 9;                                   'x' is a private member of 'Point'
    A.showCoords();     //Coords x: 5     y: 4
    B = A.move(6, 7);
    A.showCoords();     //Coords x: 11     y: 11
    B.showCoords();     //Coords x: 11     y: 11
}
```

Модификаторы доступа

```
#include <iostream>

class S {
private:
    class I {
    public:
        int x = 1;
    private:
        int y = 2;
    };
public:
    I f() {
        return I();
    }
};

int main () {
    S str;
    std::cout << str.f().x;  //Ошибка – нет
}
```

Модификаторы доступа

```
#include <iostream>

class S {
private:
    void f(int) {
        std::cout << 1;
    };
public:
    void f(float) {
        std::cout << 2;
    }
};


int main () {
    S str;
    str.f(0);       //???
    str.f(3.14);    //???
}
```

Компиляция идёт перед линковкой.

Сначала происходит выбор версии перегрузки, а потом проверка доступа, а уже после компиляции линковка.

Таким образом, проверка приватности происходит после выбора функции.

```
class S {
private:
    void f(int) {
        std::cout << 1;
    };
public:
    void f(float) {
        std::cout << 2;
    }
};

int main () {
    S str;
    str.f(0);       //???
    str.f(3.14);    //???
}
```

'f' is a private member of 'S'

Call to member function 'f' is ambiguous