Информатика

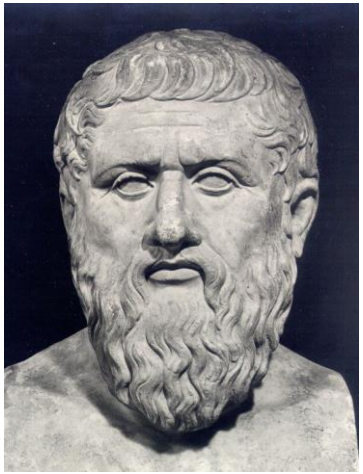
Объектно-ориентированное программирование

© Марченко Антон Александрович 2017 г.
Абрамский Михаил Михайлович

Принципы на примерах...

Философия: Платон

Во-первых, есть […] **идея**, […] незримая и никак иначе не ощущаемая, но отданная на попечение мысли. Во-вторых, есть *нечто* ***подобное этой идее*** *и носящее то же имя — ощутимое, рождённое, вечно движущееся, возникающее в некоем месте и вновь из него исчезающее* […].
В-третьих, есть […] **пространство**: оно вечно, не приемлет разрушения, дарует обитель всему роду, но само воспринимается вне ощущения, посредством некоего незаконного умозаключения, и поверить в него почти невозможно.



428-348 до н.э.

Kung Fu Panda

*- You can not win! You are just **a** big and fat panda!*

*- No! I am **the** big and fat panda!*

Два приложения

Заказ на разработку двух приложений:

- #1 Система управления договорами
- #2 Текстовая игра

Система управления договорами

Договор **заключается** с юр. лицами и физ. лицами.
У каждого договора есть **предмет, сумма, сроки**.
Сроки и суммы можно изменять.
У договора должен быть **статус** и **ответственный** за договор сотрудник.
У заказчиков ***физ. лиц*** есть ФИО, паспортные данные, прописка;
у ***юр. лиц*** – наименование, адрес, реквизиты, директор.
Договора сохраняются в хранилище с возможностью поиска по заказчикам.

Текстовая игра

Идея текстовой игры для **двух игроков**.
Игроки наносят друг другу **удары по очереди**.
Игроки **указывают силу удара** от 1 до 9, с увеличением силы возрастает вероятность промахнуться.
При успешном ударе у противника уменьшаются **очки здоровья** (hp).
Когда hp одного из игроков становится <=0, игрок проигрывает.

Система управления договорами

Решение понятно:

– типы данных – числа и строки

– хранить можно в файле

– набор договоров – массив

Но есть проблема: что такое договор?

Договор

Набор разнотипных переменных

```
string client; DateTime dueTo; double price;
```

Несколько переменных – несогласованные данные. Нужно хранить вместе

- Нужно хранилище разнотипных данных массив не подходит (однотипные данные)
  - так появились record в Pascal, struct в C
  - но это еще половина проблемы

Недостаток процедурного подхода

Как разработать систему в процедурном стиле?

- Нужно написать методы:
  - создать договор (физ.лицо, предмет)
  - ударить (игрок1, игрок2)

Что за тип физ.лицо или игрок1?

Функции

- *Функция (метод) – набор операторов, оператор – действие. Функция – тоже действие*
- В русском языке действие – **глагол/сказуемое**.
- **Описание системы на уровне функций –** описание мира неопределёнными глаголами – **несколько ограниченно!**

С другой стороны

- Понятны операции и типы данных в обоих примерах
- типы данных
  ```
  string client; DateTime dueTo; double price;
  ```
- операции
  #1 изменить сумму/строки – присвоить значение
  #2 уменьшение hp - вычитание

Всё вместе

- Нам не нужны новые способы обработки данных
- Нужен новый *подход* к разработке ***(новая парадигма)***

Анализ требований

- Договор заключается с физ.лицом/юр.лицом
- Сроки меняются
- У договора – статус, сотрудник
- У физ.лица – ФИО, паспорт, прописка
- У юр.лица – наименование, адрес, реквизиты, директор

Анализ требований

- Игроки наносят удары
- Игроки вводят силу удара
- У противника уменьшаются очки здоровья

Данные - **существительные**

- Договор, срок, сумма, предмет, хранилище, адрес, сотрудник
- Игрок, hp, сила удара

Все – **существительные**

1. Некоторые – примитивные
2. Некоторые – нет

Не примитивные

Договор, сотрудник, хранилище, игрок…

Не простые!
Вся система строится вокруг них!

- **Данные** хранятся у них
- **Действия** привязаны к ним

Существительное-сущность-объект

Объект

Основная единица (строительный блок) системы

- Содержит в себе данные (статика)
- совершает действия (динамика)

Данные

Статика, состояние объектов

- Какие данные хранятся в объекте?
  - примитивные
  - другие объекты
- Как хранить?
  - объявить переменную
- *Данные* (переменные) *внутри объектов* – **поля**

Динамика

Действия, которые совершаются объектами, их поведение

- Процедуры и функции
- Привязаны к объектам – поэтому **методы**
- Методы изменяют значения полей, значит им нужно иметь доступ

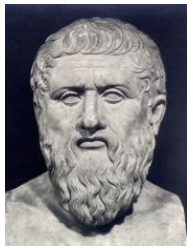
Сущности и объекты

Уникальные сущности отличаются от реального количества объектов

– Сущности **– договор, сотрудник, игрок**

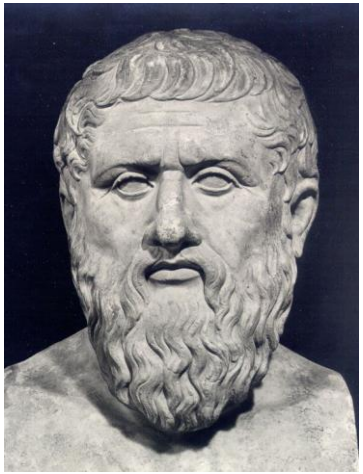
– Договором – много, а сотрудников и игроков?

Есть объекты, составляемые по единому подобию, шаблону, каркасу…



Философия: Платон

Во-первых, есть […] **идея**, […] незримая и никак иначе не ощущаемая, но отданная на попечение мысли. Во-вторых, есть *нечто* ***подобное этой идее*** *и носящее то же имя — ощутимое, рождённое, вечно движущееся, возникающее в некоем месте и вновь из него исчезающее* […].
В-третьих, есть […] **пространство**: оно вечно, не приемлет разрушения, дарует обитель всему роду, но само воспринимается вне ощущения, посредством некоего незаконного умозаключения, и поверить в него почти невозможно.



428-348 до н.э.

Kung Fu Panda

*- You can not win! You are just **a** big and fat panda!*

*- No! I am **the** big and fat panda!*

Класс

- Пользовательский **тип данных**
  - Каркас, шаблон, образец, чертёж объекта
- Объект – переменная этого типа
- **Объект – экземпляр класса**

Итоги и влияние на разработку...

Структурный и процедурный подход

- Структурных подход позволяет строить алгоритмы, но не способствует повторному использованию
- Процедурный подход хорошо описывает процессы/действия, но отстранён от данных
- Вся программа – единый алгоритм

Программирование в малом

- Программу может написать один человек
- Программу можно переписать для
  - исправления/расширения,
  - портирования на новую платформу
- ***Подходит для небольших систем***

Системная сложность

Не алгоритмическая сложность

- Большое количество независимых компонент
- Сложные связи между компонентами
- Одному человеку не разобраться
- Нужно бороться со сложностью,
  держать сложность «под контролем»

Программирование в большом

- Разрабатывается командой
- Никто не знает всех деталей реализации программы
- Продолжительный срок «жизни» программы
- Нельзя просто так переписать, расширить, исправить
- Требует организации процесса разработки
- Требует использования подходящих методов и инструментов разработки

Требования к системам

- **Эффективность** (надёжность, производительность)
- **Гибкость** (изменяемость)
- **Расширяемость** (добавление сущностей)
- **Масштабируемость процесса** (добавление людей)
- **Тестируемость** (возможность проверки)
- **Возможность повторного использования**
- **Сопровождаемость** (легко разобраться в программе)

Архитектура ПО

Правила, эвристики и паттерны для

- проектирования системы как нескольких частей
- создание интерфейсов взаимодействия этих частей
- контроля над общей структурой и потоком управления
- взаимодействия с окружением
- соответствующего использования подходов, техник и инструментов разработки

ООП

Объектно-ориентированное программирование

- развитие идей структурного, процедурного и модульного программирования
- поддержка «программирования в большом»
- *ориентировано* на архитектуру ПО

ОО разработка

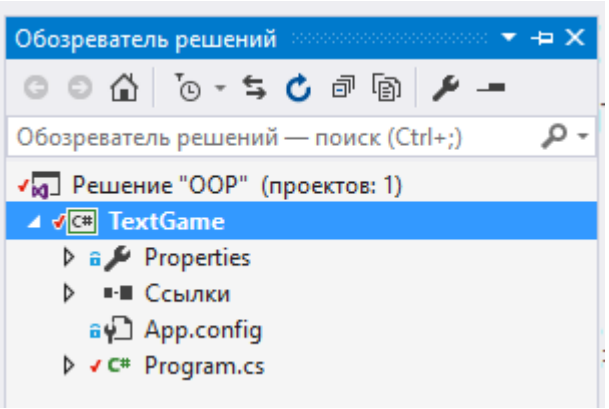
- Описание системы
- Выделение подсистем
- Разделение подсистем на классы
- Определение взаимодействия
- Проектирование классов
- Тестирование
- Внедрение

Let’s code…

Начнём

Создадим новый проект

консольного приложения C#



Увидим класс Program с точкой входа – Main

их трогать не будем, пока…

Добавим новые элементы в решение

Нам нужен класс Игрока…

Player

```
class Player {}
```

- Уже корректное описание класса:
- Название – **P**ascalCase
- В теле – члены класса, которых может не быть
- Рекомендуется классы описывать в отдельном файле с тем же именем, что и у класса

> в Java выделение класса в отдельный файл – обязательное требование, в C# - рекомендация

Player.cs

```
class Player
{
    int hp;
}
```

hp – поле класса (**c**amelCase)

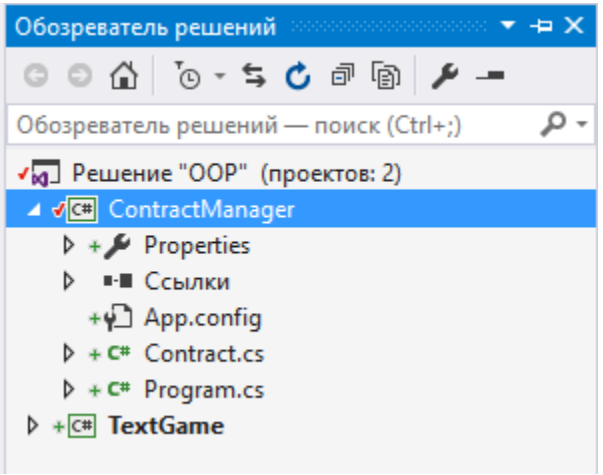
- в Java данные в классе – атрибуты
- в C# у понятия «атрибут» другое значение

будет доступно всем элементам Player

А что насчёт другого заказа?

Начинаем проектировать Contract.cs

```
class Contract
{
    string subject;
    DateTime dueTo;
    double cost;
    …
}
```



Позвонил заказчик, изменил требования

- Слушай, это, уточнили у генерального, ответственный должен быть только у договора с физиками. Учтите это у себя

Contract

Ок, значит у двух типов контрактов разные наборы полей, поэтому **пока** придётся написать два класса.
Пример для физ. лица:

```
class IndividualContract
{
    string subject;
    DateTime dueTo;
    double cost;
    …
}
```

Не хватает ответственного, самого физ. лица...
А чем они являются?

Нужно больше классов

```
class IndividualContract
{
    string subject;
    DateTime dueTo;
    double cost;
    Individual individual;
    Employee responsible;
}

class Individual
{
    string name;
    PassportInfo passportInfo;
    Address address;
}
```

```
class PassportInfo {…}

class Address {…}

class Employee {…}
```

Что писать в Employee?

- Заказчик ничего не сказал

- Аналитик сказал что Employee – человек
  - Какие поля у человека?

Человек

- Ф.И.О
- Год рождения
- Образование
- Семейное положение
- Любимая музыка
- Пол
- Группа крови
- Предпочтения в еде
- Страница в ВК
- Страница в Facebook

Их тысячи! Все ли нужны?

Принцип ООП: Абстракция

В класс добавляется только то, что действительно необходимо в рамках предметной области и разрабатываемой системы

есть и другие проявления абстракции

- Клиенту интернет-магазина «группа крови» не нужна
- Пациенту в медицинской информационной системе «группа крови» необходима

Агрегация

А что, если добавить в класс Employee (сотрудника) поле – ссылку на начальника, который сам является экземпляром Employee (сотрудником)?

```
class Employee
{
    string name;
    Department department;
    Employee chief;
}
```

Рекурсивные типы?
Никаких проблем!

Добавим метод

- Пусть игрок перед началом битвы произносит свой боевой клич!

  – Заказчик одобряет!

```
class Player
{
    int hp;
    string battleCry;
    void ShoutBattleCry()
    {
        Console.WriteLine(battleCry);
    }
}
```

Создадим объекты

Если не оговорено другого, считайте, что код пишется в Main

```
Player first = new Player();
```

Синтаксис нам знаком:

- Player – ссылочный тип
- first – ссылка на экземпляр класса Player
- объект – экземпляр класса Player создаётся вызовом оператора new (выделяется память в куче)
- **Что такое Player()?**

Конструктор

Метод, вызывающийся при создании экземпляра класса

- В нём можно инициализировать значения полей
  - иначе все они будут null, 0, false

Что можно инициализировать в конструкторе Игрока?

Конструктор по умолчанию

- Если нет других конструкторов (перегрузок), существует по умолчанию
- Не имеет параметров
- Ничего не делает, если его явно не определим
- hp не зависит от параметров, определим в конструкторе

Конструктор для Player

```
class Player
{
    int hp;
    string battleCry;

    Player()
    {
        hp = 100;
    }

    void ShoutBattleCry()
    {
        Console.WriteLine(battleCry);
    }
}
```

void, но не пишем

Герой немой и безымянный

```
Player first = new Player();
first.ShoutBattleCry();
```

. – точка доступа к членам класса

ShoutBatleCry выдаст пустую строку

Нужно определить боевой клич игрока!
И имя, нужно дать ему имя!

- Заказчик приходил, сказал, что думал, что мы сами до этого додумаемся

Конструктор с параметрами

Необходимо передать данные извне для инициализации полей

У нас уже есть средство для этого!

Generate From Usage

- Можно сперва определить поведение объекта, его ***интерфейс***
  – интерфейс – то, «**как объект себя ведёт**»
- Потом определить то, что за этим поведением стоит (реализуем метод в классе)

Visual Studio может помочь нам в этом

Хотим, чтобы выглядело так

```
Player first = new Player("Leonidas", "This is Sparta!!!11");
first.ShoutBattleCry();
```

Ок, сделаем

```
class Player
{
    int hp;
    string name;
    string battleCry;
    Player(string name, string battleCry)
    {
        hp = 100;
        name = name;
        battleCry = battleCry;
    }
    void ShoutBattleCry()
    {
        Console.WriteLine(name + ": " + battleCry);
    }
}
```

добавили имя

передали параметры

пытаемся присвоить...
и присваиваем значения параметров самим параметрам, а не полям

Какой выход?

- Переименовать параметры?
  - Не круто!
    - battleCry – он везде battleCry
- Нужно указать что обращаемся к полям класса!

this

```
Player(string name, string battleCry)
{
    hp = 100;
    this.name = name;
    this.battleCry = battleCry;
}
void ShoutBattleCry()
{
    Console.WriteLine(name + ": " + battleCry);
}
```

this – это ссылка на текущий объект

соответствует слову «МОЙ»

не требуется там, где очевидно

Дублирование кода

```
Player(string name)
{
    hp = 100;
    this.name = name;
}

Player(string name, string battleCry)
{
    hp = 100;
    this.name = name;
    this.battleCry = battleCry;
}
```

Или так

```
Player(string name)
{
    hp = 100;
    this.name = name;
    this.battleCry = "This is Sparta!!!11";
}

Player(string name, string battleCry)
{
    hp = 100;
    this.name = name;
    this.battleCry = battleCry;
}
```

Какой из конструкторов частный случай?

Ещё один смысл this

```
Player(string name)
    : this(name, "This is Sparta!!!11")
{
}

Player(string name, string battleCry)
{
    hp = 100;
    this.name = name;
    this.battleCry = battleCry;
}
```

Продолжаем убирать «лишнее»

Методы с опциональными параметрами – можно указать значения параметров по умолчанию и получить возможность «пропускать» параметр при вызове

- Опциональные параметры должны идти после обязательных

```
Player(string name, string battleCry = "This is Sparta!!!11")
{
    hp = 100;
    this.name = name;
    this.battleCry = battleCry;
}
```

Классы и объекты

- Классы проектируются от первого лица, экземпляры классов (объекты) используются от третьего
  - Сравните класс Player и объект first
- **Проектирование класса и использование экземпляров – два разных процесса разработки**
  - выполняется разными людьми, в разных частях системы, в разное время
    - Экземпляры Player – поля класса Game
    - Используем string, Array, Console, а реализацию даже не видели!

Доступ к полям

Что там с договорами?

У договоров «можно менять сроки и сумму»

Возьмём и поменяем, это же переменные

```
IndividualContract ic1 =
    new IndividualContract(
        "Development",
        new DateTime(2016,4,1),
        100000);
…
ic1.Cost = 200000;
```

Если просто переменные...

Значит можно написать так:

```
ic1.Subject = "Ерунда какая-то";
```

Но в требованиях не было указано что так можно!

Заказчик сообщил о проблеме, но прямо противоположной! Что случилось?

Заказчик пришёл, сказал что мы попадаем на деньги...

Модификаторы доступа

Можно ограничить возможности прямого доступа к членам класса и к самим классам

в нашем случае даже нужно!

- **public** – доступ всем и отовсюду
- **private** – доступ только внутри класса, в котором находится данное поле/метод
- …

Доступ

А что, если нам нужно иметь возможность получать значения полей отовсюду, но запретить их изменение всем, кроме самого класса – владельца полей?

- Как мы обычно это делаем?
  Как узнать какого цвета глаза?
- Посмотреть, Спросить – глаголы? Методы!

Инкапсуляция

Латинское ***in capsula*** — *размещение в оболочке*, изоляция, закрытие чего-либо инородного с целью исключения влияния на окружающее, *обеспечение доступности главного*, выделение основного содержания – *помещение* всего мешающего, *второстепенного в некую условную капсулу*

Инкапсуляция

- Связывание данных с методами обработки
- Механизм языка, позволяющий ограничить доступ одних программных компонент другим,
  разграничить публичный интерфейс от деталей реализации
  - не нужно уметь «вращать двигатель» чтобы ездить на автомобиле

IndividualContract

```
public class IndividualContract
{
    private string _subject;
    private DateTime _dueTo;
    private double _cost;
    private Individual _individual;
    private Employee _responsible;

    public IndividualContract(string subject, DateTime dueTo, double cost)
    {
        this._subject = subject;
        this._dueTo = dueTo;
        this._cost = cost;
    }
}
```

А как же доступ?

Можно описать Get, Set методы для получения/изменения значения полей

– подход Java

- Сами поля – закрытые (private)
- Методы доступа – открыты или закрыты по необходимости

get, set в Contract

```
private string _subject;
private DateTime _dueTo;
private double _cost;

public string GetSubject()
{
    return _subject;
}
public DateTime GetDueTo()
{
    return _dueTo;
}
```

```
public void SetDueTo(DateTime value)
{
    _dueTo = value;
}
public double GetCost()
{
    return _cost;
}
public void SetCost(double value)
{
    _cost = value;
}
```

Заказчик доволен, штрафовать не будет.

Писанины многовато

Expression body (C# 6.0)

Можно использовать лямбда-выражения вместо тел методов

```
private string _subject;
private DateTime _dueTo;
private double _cost;

public string GetSubject() => _subject;
public DateTime GetDueTo() => _dueTo;
public void SetDueTo(DateTime value) => _dueTo = value;
public double GetCost() => _cost;
public void SetCost(double value) => _cost = value;
```

Свойства

Развитие идей инкапсуляции

```
private string _subject;
private DateTime _dueTo;
private double _cost;

public string Subject
{
    get { return _subject; }
}
```

```
public DateTime DueTo
{
    get { return _dueTo; }
    set { _dueTo = value; }
}
public double Cost
{
    get { return _cost; }
    set { _cost = value; }
}
```

Автоматические свойства

Если для чтения/изменения «обычное», можно использовать автоматические свойства

```
public string Subject { get; private set; }
public DateTime DueTo { get; set; }
public double Cost { get; set; }
```

При этом закрытое поле тоже писать не нужно – создастся автоматически.

Статические поля и методы

- У каждого объекта – свой набор значений полей:
  - У каждого договора – свои поля
  - У каждого игрока – свои hp, клич
- Но иногда есть необходимость в общих полях и методах для всех
  - Заказчик: хочу сделать сквозную нумерацию договоров IndividualContract

static

Статические static данные для всех объектов (глобальны), они должны существовать, даже если ни одного объекта не создано

```
public class IndividualContract
{
    private static int _numberOfContracts = 0;
    …
```

Инициализировать можно при объявлении или в статическом конструкторе

```
static IndividualContract()
{
    _numberOfContracts = 0;
}
```

Статический конструктор

- Может существовать как отдельно, так и совместно с конструктором по умолчанию
- Не может иметь модификаторов доступа
- Вызывается неявно при первом обращении к классу как и инициализатор статических полей

Работа со static обычна

```
public IndividualContract(string subject, DateTime dueTo, double cost)
{
    this.Subject = subject;
    this.DueTo = dueTo;
    this.Cost = cost;
    _numberOfContracts++;
}
```

Предостережение

Статические поля могут приводить к некорректно обрабатываться в многопоточных приложениях. Нужно работать с ними специальным образом

```
public IndividualContract(string subject, DateTime dueTo, double cost)
{
    this.Subject = subject;
    this.DueTo = dueTo;
    this.Cost = cost;
    Interlocked.Increment(ref _numberOfContracts);
}
```

Статические методы

Если есть общие данные, привязанные к классам, должны быть и методы, привязанные к классам

Примеры:

Math.Sin, Math.Cos, Console.WriteLine

Какой метод должен (ОБЯЗАН) работать, даже когда ни одного объекта не создано?

Разгадка тайны Main!

```
static void Main()
```

Точка запуска программы, не принадлежит конкретному объекту. Метод должен запуститься когда ещё ни один объект не создан

Должна быть одна точка доступа в программе

CLR ищет точку доступа автоматически

В статических методах

Могут использоваться только другие статические методы/статические поля класса

- Опять, потому что должны работать когда ни одного объекта не создано
- Поэтому мы все методы рядом с Main писали статическими

Вопросы?

e-mail: marchenko@it.kfu.ru