GeekBrains

Как не нужно писать код

Принципы и антипаттерны

Оглавление

[00:01:36]

Приветствие

Друзья, рад вас приветствовать на очередной лекции из серии «Как не нужно писать код». Сегодня мы продолжим погружаться в эту непростую, но интересную тему.

У меня к вам есть несколько вопросов:

1. **Насколько сложно программировать?** Вы уже пишете код и знаете, как сделать его красивым. Но насколько это сложно?
2. **Что значит «программировать»?** Начинающие программисты часто путают синтаксис языка с процессом программирования. Например, идея циклов относится к программированию, а вот то, что нужно поставить точку с запятой или написать счётчик — это особенности языка. Наша задача — связать всё в кучу: в рамках отдельного языка научиться программировать и писать код, который будет не стыдно показать.
3. **Что такое красивый код?** Отвечайте именно про красоту, а не синтетические особенности. Какой код, как вам кажется, не стыдно показать?

[00:06:52]

Ложные признаки красивого кода

Начинающие много времени уделяют тому, что абсолютно не нужно — бантикам и рюшечкам, красивому читаемому шрифту и другим украшениям.

Рассмотрим пример: вы начинаете писать код, и вам хочется, чтобы пользователь вводил числа с клавиатуры. Вы ставите себе задачу, но как её решить — непонятно. Вы погуглили и узнали о конвертации строк в числа — всё хорошо работает. Казалось бы, вы решили одну небольшую проблему считывания данных, но создали себе целый вагон других. Например, что будет, если пользователь начнёт вводить не число? Вы знаете: если ваша программа запрашивает число, нужно вводить число. А классический пользователь станет вводить всё, что угодно, кроме числа, но ваша программа должна это корректно обрабатывать. 80% времени будет уходить на то, чтобы корректно обработать ошибки, возникающие не по вашей вине, а по вине пользователя.

Помните, не нужно ставить себе задачи, которые могут вызвать ещё больше проблем. Классический пример такой задачи — попытка ввода данных от пользователя. Вам может показаться, что решение этой проблемы сделает код лучше, но по факту будет только хуже.

[00:08:48]

По-настоящему красивый код

Хороший код:

- гибкий,
- расширяемый,
- модульный,
- поддерживаемый,
- документируемый.

Возможно, эти признаки покажутся вам абстрактными, потому что вы ещё не участвовали в написании больших систем. Но напомню, задача — не показать, что вы чего-то не знаете, а получить насмотренность кода. Чтобы у вас отложились какие-то принципы, и вы доставали их, чтобы применить на практике.

[00:09:28]

Гибкость

Здорово, если функционал одного модуля можно использовать несколько раз.

Допустим, вы написали метод заполнения массива числами. Если это массив целых чисел, туда можно положить только целые числа. Если массив double, вещественные числа. А что, если в качестве данных, которые нужно будет получить, у вас будут строки? Или котики, и нужно будет вернуть массив котиков? Хорошо, если система заранее построена так, что код можно переиспользовать.

При этом важно не пытаться всё обобщать. Писать код, который можно переиспользовать — хорошо. Но писать код, который будет исключительно общим, — так себе правило.

Может возникнуть когнитивный диссонанс: как писать код, который, с одной стороны, можно переиспользовать, и который, с другой стороны, должен решать конкретную задачу? Всё приходит с опытом. Со временем вы начнёте понимать, в какой задаче можно выделить общую логику, а в какой лучше использовать что-то конкретное.

[00:11:09]

Расширяемость

Хорошо, если вы спроектировали систему так, что в неё можно будет добавить новые модули.

Представим, что мы пишем мессенджер. Сегодня в нём можно принимать и отправлять сообщения только одному пользователю. Завтра мы захотим добавить групповые чаты. Сложно ли это? Если система изначально задумана так, что в чате могут находиться только две условные единицы (два пользователя), придётся писать её заново. Но ведь мы могли изначально продумать систему так, что в чате может быть сколько угодно пользователей. Просто в первой версии в ней могло бы находиться два человека, а дальше мы бы доделывали функционал.

Другой пример: изначально в нашем мессенджере можно было отправлять только текстовые сообщения, но мы решили добавить возможность отправлять картинки. Сильно ли они отличаются от песни или голосового сообщения? И то, и то — файл. Поэтому нам нужно было бы описать метод, который отправляет файл. А в качестве файла могут быть картинки, песни, звуки, видео, анимации и так далее.

[00:12:41]

Модульность

Представим, что мы написали мессенджер, который может отправлять картинки, но система отправки работает очень медленно. Как сделать лучше?

Плохо, если придётся переписывать весь мессенджер. Мы изначально могли спроектировать систему так, чтобы можно было вытащить один модуль, который отвечает за отправку картинок, и переписать

его, возможно, привлекая более опытных разработчиков. Как следствие, заменили бы модуль отправки картинок на новый, который работает лучше. Но это возможно только в том случае, если при написании системы мы заранее помнили об этом.

[00:13:25]

Поддерживаемость

После того как вы выкатили первую релизную версию продукта, скорее всего, какое-то время оно будет работать. Но после выхода очередного обновления что-то может сломаться, даже если раньше работало хорошо. И здорово, если кроме человека, который занимался этим модулем, в команде есть кто-то, кто может его починить.

Может быть веселее: вы, как заказчик, заказали продукт у компании-исполнителя. И после окончания разработки, исполнители передали вам исходный код, а вы наняли команду поддержки, которая сможет в этом коде разобраться и фиксить возникающие баги.

[00:14:18]

Документируемость

Я считаю, что документация должна быть. Но есть те, кто говорит, что код должен быть самодокументируемым (и если это не так, то код плохой). Могу и согласиться с этим мнением, и не согласиться. Почему — обсудим позже.

[00:14:44]

Пример: улучшение кода

В предыдущей лекции я смотрел ваш код и говорил, что в нём можно улучшить. Теперь предлагаю обратную ситуацию: я показываю вам свой код, а вы в комментариях пишете, что бы вы в нём улучшили. Постарайтесь за 3 минуты максимально откомментировать код.

```
const Double пи = 3.1415;
int k__BackingField;
void set_MyProperty(int value)
{
    k__BackingField = value;
}
int get_MyProperty()
{
    return k__BackingField;
}
int MyProperty { get; set; }
```

Если кого-то смущают слова **const**, **get**, **set**, не нужно на них ругаться. Это просто особенности языка. Здесь и в последующих примерах обращайте внимание на стиль написания кода, а не на то, какие слова используются.

Итак, что здесь точно нужно улучшать?

1. **Именование кириллицей.** Такое допускать нельзя, поэтому сразу убираем саму идею использования кириллицы в идентификаторах.
2. **Тип данных Double** является типом платформы .Net. В языке C# принято описывать тип с маленькой буквы. Мы описываем константу вещественного типа, для которой определяем число Пи. Дальше, используя встроенные механизмы округления или отсечения нужного количества знаков, мы можем превратить наше число Пи в число с 4 знаками после запятой.
3. **Использование имени переменной k_BackingField**. Из предыдущей лекции мы помним о том, что использование нижних подчёркиваний в языке C# не приветствуется, поэтому убираем. Как следствие, если у нас есть проблема в текущей переменной, то и в следующих 2-х методах её использовать тоже не очень хорошо, поэтому нужно придумывать новое название.
4. **Наименование метода с маленькой буквы, нижнее подчёркивание**. Помним, что нижнее подчеркивание не нужно использовать в принципе. А наименование метода с маленькой буквы тоже не считается правильным кейсом. Значит переписываем или придумываем новые наименования методов.
5. **Нейминг свойства int MyProperty.** Непонятно, о чём речь. Лучше заменить.

Но что главное? Прежде чем начинать рассматривать, хорошо ли написан код, нужно проверить, работает ли он. В данном случае код не запустится.

Об именовании переменных, классов и других сущностей в языке C# можно почитать в документации:

- Правила и соглашения об именовании идентификаторов C#
- Соглашения о написании кода на C#

[00:21:12]

Пример: магические числа

Следующий пример. Я показываю вам метод, который называется DrawText. Он принимает что-то в качестве аргумента, и в теле метода что-то происходит.

Снова даю вам 3 минуты на размышление. Напишите в комментариях, что плохо в DrawText. В методе ли? Или в его вызове? Или здесь всё хорошо? Или плохо? Укажите, что именно вам не понравилось.

```
void DrawText(string text, int left, int top)
{
    Console.SetCursorPosition(left, top);
    Console.WriteLine(text);
}
```

```
DrawText("Intensive C# Demo text", 629, 360);
```

С точки зрения метода и наименование аргументов всё хорошо. У нас есть метод, который отражает то, что он делает (он куда-то рисует текст). В качестве аргументов к нам приходит текст, позиция по левому краю и позиция по верхнему краю (или отступ от левого края и отступ от верхнего края). В разработке принято считать началом отсчёта или началом системы координат верхний левый угол, поэтому, скорее всего, все вас поймут.

Что дальше? Вызов метода **DrawText ("Intensive C# Demo text", 629, 360)**. Мне, как человеку, который читает этот код, непонятно, что это за 629 и 360. Мы можем спуститься в комментарии и, может быть, найдём там ответ.

629, 360 – что это?

629 - пусть ширина экрана будет 1280px, тогда середина 1280 / 2 - 640
в тексте "Intensive C# Demo text" 22 символа, чтобы отцентровать по ширине, нужно от центральной точки отступить половину выводимых символов
т.к. ширина символа одинакова, получаем 1280 / 2 - 11 = 629
пусть высота экрана будет 720px
720 / 2 = 360

Видим проблему: человеку, который писал код и делал вызов, всё понятно. А тот, кто будет делать код-ревью, не станет читать эти комментарии.

В программировании при написании кода есть определённые принципы, которых хорошо бы придерживаться. Кроме принципов есть паттерны — устоявшиеся модели описания кода. И есть антипаттерны — то, как не нужно писать код.

В примере я показываю использование **антипаттерна «магические числа»** или **«магические константы»**. 629 и 360 — это магические числа. Когда речь идёт о каких-то константах, в том числе о строковых, это уже магические константы.

В данном случае нам абсолютно непонятно, почему 360 (без чтения дополнительных комментариев). Это плохо.

Как сделать лучше? Для начала — описать некоторые переменные, которые отражают то, что они делают. Если мы хотим указать середину экрана, например, по высоте, мы так и напишем

screenHeightPosition. Если хотим указать середину по ширине, делаем так же: явно описываем ширину — **screenWightPosition**.

```
string caption = "Intensive C# Demo text";
int screenWidthPosition = (Console.WindowWidth - caption.Length) / 2;
int screenHeightPosition = Console.WindowHeight / 2;

DrawText(caption, screenWidthPosition, screenHeightPosition);
```

```
DrawText(
    text: caption,
    left: screenWidthPosition,
    top: screenHeightPosition
    );
```

В примере мы указываем, что нужно взять ширину экрана или ширину терминала, отнять от неё длину строки, которую мы пытаемся распечатать, и поделить пополам. Или можно взять экран, разделить его пополам, взять текст, разделить пополам и, соответственно, из координаты, которая указывает центр экрана, отнять половину нашей строки. Аналогично и с высотой. А дальше уже вызывать метод, передавая именованные переменные DrawText (caption, screenWightPosition, screenHeightPosition). Так код становится гораздо читаемее. При этом нам не нужны тонны комментариев.

От себя отмечу, что вызовы, у которых больше 2–3 аргументов, я стараюсь описывать таким образом: изначально указываю имя аргумента, текст и через двоеточие пишу значение.

```
DrawText(
    text: caption,
    left: screenWidthPosition,
    top: screenHeightPosition
    );
```

Почему я так делаю? Если мне потом нужно будет что-то поменять, мне не придётся вычленять какую-то строчку. Нужно будет просто убрать всю строку и на её место поставить новую.

[00:28:43]

Пример: 111 и 222

Следующий пример:

```
string label = "";              // 111
string address = String.Empty;  // 222
```

Здесь даже не буду делать перерыв. Просто напишите в чат:

- 111, если вы инициализируете строковые переменные так, как показано в первом примере на скриншоте,
- 222, если инициализируете так, как описано во втором,
- 333, если вы делаете и так, и так.

Интересно посмотреть, кто какие способы использует. Какой из них более правильный?

С точки зрения антипаттернов опять же работают магические константы или строки. Вариант 111 непонятен человеку, который не знаком с языком C#. Для него две закрывающие кавычки не ясны, непонятно, что это такое. В свою очередь, чёткий нейминг **String.Empty** показывает, что строка пустая.

Поэтому лучше использовать вариант 222. Причём это справедливо не только для языка C#, но и для других тоже. Скорее всего, будут свои именованные константы, обозначающие максимальное знание числа или пустую строку, или минимальное значение числа и так далее.

[00:29:57]

Принцип don't repeat yourself

Следующий пример. Предлагаю вам посмотреть на этот код. Попробуем понять, что он делает, и что в нём плохо.

```
// Пример 1
double a = 1, b = -26, c = 120;
var d = b * b - 4 * a * c;
double x1 = (-b + Math.Sqrt(d)) / (2 * a);
double x2 = (-b - Math.Sqrt(d)) / (2 * a);
Console.WriteLine($"x1 = {x1} x2 = {x2}");

// Пример 2
a = 2; b = 1; c = -3;
d = b * b - 4 * a * c;
x1 = (-b + Math.Sqrt(d)) / (2 * a);
x2 = (-b - Math.Sqrt(d)) / (2 * a);
Console.WriteLine($"x1 = {x1} x2 = {x2}");

// Пример 3
a = 1; b = 1; c = -6;
d = b * b - 4 * a * c;
x1 = (-b + Math.Sqrt(d)) / (2 * a);
x2 = (-b - Math.Sqrt(d)) / (2 * a);
Console.WriteLine($"x1 = {x1} x2 = {x2}");
```

Как сделать лучше? В первую очередь явно видно, что мы дважды сделали Ctrl+c и Ctrl+v, то есть просто скопипастили код. Насколько это хорошо? С точки зрения принципов разработки — плохо. Нам нужно было выделить отдельный метод, который решал бы конкретную задачу, в частности, принимал три коэффициента. Если вы заметили, в этом коде решается три квадратных уравнения. Как следствие, мы описываем метод, который в качестве аргументов принимает три коэффициента и возвращает корни уравнения.

В решении ниже я писал абстрактную структуру Roots, но можно было вернуть пару чисел или указать массив, где первый элемент указывал бы одно значение или один корень, а второй массив — второй корень. Это уже не принципиально. Идея в том, что описать метод один раз и использовать трижды — это лучше, чем копипастить код.

```
public Roots Solve(double a, double b, double c)
{
    var d = b * b - 4 * a * c;
    double x1 = (-b + Math.Sqrt(d)) / (2 * a);
    double x2 = (-b - Math.Sqrt(d)) / (2 * a);
    return new Roots { X1 = x1, X2 = x2 };
}
```

Когда вы копипастите, вы нарушаете один из принципов разработки — **don't repeat yourself**. Его вы должны придерживаться всегда и везде. Если вы копипастите, вы делаете неправильно. Нужно

остановиться и подумать, как этого избежать, вынести в отдельный модуль или создать какую-то специальную сущность.

[00:34:45]

Принцип YAGNI

Следующий пример — вам дали задачу: случайным образом на экране показать случайные цифры или символы. Как вы станете её решать? Не надо ничего описывать, просто подумайте, как бы вы это делали.

В качестве демонстрации привожу такой код:

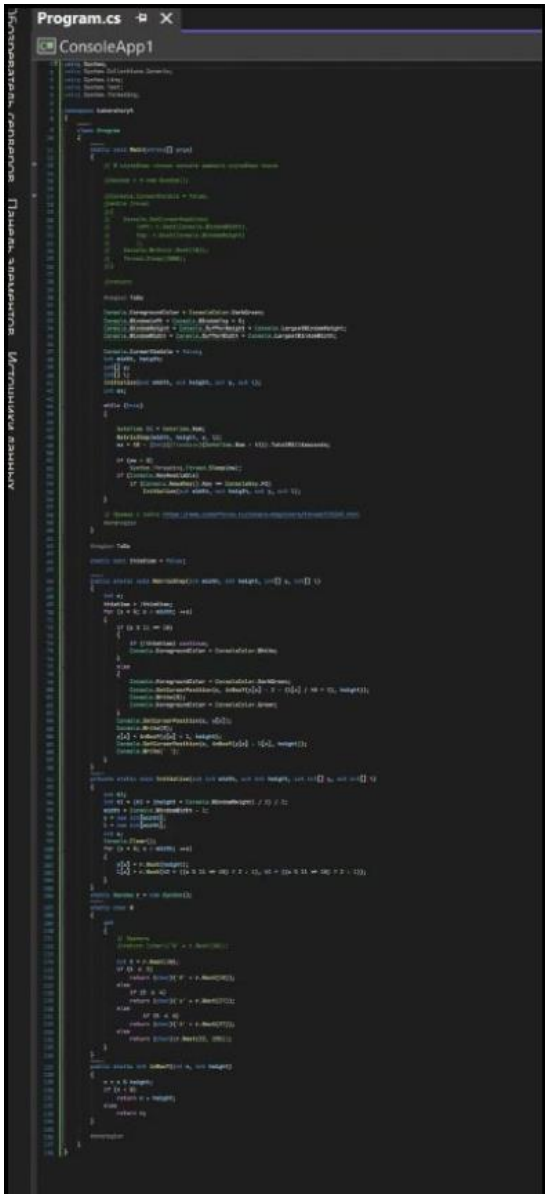
```
// В случайных точках консоли вывести случайные числа

Random r = new Random();

Console.CursorVisible = false;
while (true)
{
    Console.SetCursorPosition(
        left: r.Next(Console.WindowWidth),
        top: r.Next(Console.WindowHeight)
        );
    Console.Write(r.Next(10));
    Thread.Sleep(1000);
}
```

Технически он может показаться вам простым. Да, есть бесконечный цикл, который устанавливает позицию курсора в нужные координаты, и просто печатает какой-то символ.

Насколько это решение интересное? Вы посидели, подумали, вам этот код показался скучным, и вы решили немного потратить времени и написать какой-то другой код, который делает что-то иначе:



Здесь 130 строк, их необязательно рассматривать. Но если мы запустим код, вот во что он превратится:

Насколько правильно было решать задачу так? Если вы решили её за 130 строк кода вместо условных 10, вы потратили время. Если вы работаете над своим проектом или делаете тестовое задание, это нормально, тратьте, пожалуйста. Но когда вы работаете в компании, вы должны думать, что время, которое вы тратите, кем-то оплачивается. И когда вы начинаете проявлять такую инициативу, вы впустую тратите деньги.

Если вам была поставлена задача случайно вывести символы в каких-то координатах экрана, то, когда вы попытались показать пример использования такого экрана-матрицы, не исключено, что заказчик вообще не оценит эту инициативу. Он скажет: «Зачем мне нужны эти зелёные цифры, что вы натворили? Сделайте ровно то, что я прошу».

Вы потратили время (скорее всего, не две секунды), и в дальнейшем заказчик может сказать, что это полная чепуха и нужно сделать попроще. Так что помните, что не нужно делать то, о чём вас не просили. Это ещё один принцип разработки — **YAGNI** (you aren't gonna need it — вам это не понадобится): не делайте то, о чём вас не просят, не пытайтесь проявлять инициативу, потому что, скорее всего, окажется только хуже.

[00:37:24]

Принцип KISS

Кстати, этот пример — хорошая иллюстрация следующего принципа — **KISS** (keep it simple, stupid). Его суть в том, что не нужно делать что-то сложное, если можно сделать в разы проще.

Чем проще ваша система, тем легче её поддерживать и расширять. Тем легче в ней находить баги, писать тесты и так далее. Помните о том, что нужно описывать методы максимально просто, но в общем случае методы дальше компонуются в какие-то отдельные классы. Будь то структуры или какие-то иные системы, единицы.

[00:38:04]

Пишем парсинг входной строки

Основные принципы мы узнали, теперь напишем парсинг входной строки.

Есть строка, в которой находятся парные координаты точек фигуры.

```
string text = "(1,2) (2,3) (4,5) (6,7)";
```

Наша задача — увеличить каждую из координат в два раза и показать пользователю ответ.

Можно описать метод, который сначала будет разбивать пары в скобках. Затем, описав отдельный метод, взять каждую пару (то есть точку с координатой X и координатой Y), и увеличить значение каждой координаты с помощью ещё одного метода.

В целом, это хорошее решение. Если конкретная маленькая подзадача выделяется в отдельный метод, наверное, это хорошо. Есть свои тонкости, но я их сейчас опущу. При этом, когда вы описываете такой код, нужно помнить о том, что было на предыдущей лекции: код вы пишете не для себя, а для того, кто будет на него смотреть. Можно ли с первого раза понять, что ваши методы делают и сколько их там?

Давайте попробуем решить эту задачу, используя встроенный функционал языка C#. Ещё раз отмечаю, не думайте о том, что это попытка показать вам: смотрите, вот как может C#, и как я могу. Идея в том, что вы сможете понять, что происходит в коде, даже не зная, как его написать и прочитать.

Поехали. Что я здесь буду использовать? Как и в предыдущей лекции, я возьму дополнительный модуль **System.Linq**. Скорее всего, он мне понадобится. Далее у меня будет неявная типизация. И сделаем так: у строки будет метод **Split**, позволяющий сделать разбивку с учётом символа-

разделителя. Для меня сейчас символом разделителем будет пробел. И сразу же я буду превращать полученный набор данных в массив.

```
1  using System.Linq;
2
3  string text = "(1,2) (2,3) (4,5) (6,7)";
4
5  var data = text.Split(" ")
6
7              .ToArray();
8
9  for (int i = 0; i < data.Length; i++)
10 {
11     Console.WriteLine(data[i]);
12 }
```

Итак, давайте посмотрим, что же у нас лежит в data[i]. Каждый из примеров я буду запускать, и мы будем смотреть, что получится.

Итак, у нас была строка, которая содержала в себе сразу все точки. Мы разбили её и получили 4 точки.

```
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
(1,2)
(2,3)
(4,5)
(6,7)
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _
```

Пока что это строки. Технически вместо var мы можем прописать массив строк. Чтобы постоянно типы не менять, я оставлю var.

Дальше моя задача — взять пару и на её основе получить точку с числами, а не со строками. Технически скобки для нас ничего не значат, поэтому я могу избавиться от них на первом этапе. Для этого я напишу метод **Replace**, где в качестве первого аргумента мы укажем открывающую скобку, а в качестве второго — на что её нужно заменить. В данном случае — на пробел. Затем делаем то же с закрывающей строкой. На выходе у нас будет строка без скобок.

```
3  string text = "(1,2) (2,3) (4,5) (6,7)"
4              .Replace("(", "")
5              .Replace(")", "")
6              ;
7  Console.WriteLine(text);
8  var data = text.Split(" ")
9
10             .ToArray();
11
12 for (int i = 0; i < data.Length; i++)
```

Мы получили входную строку и разбили её на пары. Что делаем дальше? В каждом элементе можем сделать выборку, то есть разбить каждый элемент и получить на его основе массив из двух чисел. Первое число будет координатой X, второе — Y.

Я укажу **Select(item)**, и хочу его превратить, используя Split. Только теперь в качестве символа-разделителя у меня запятая.

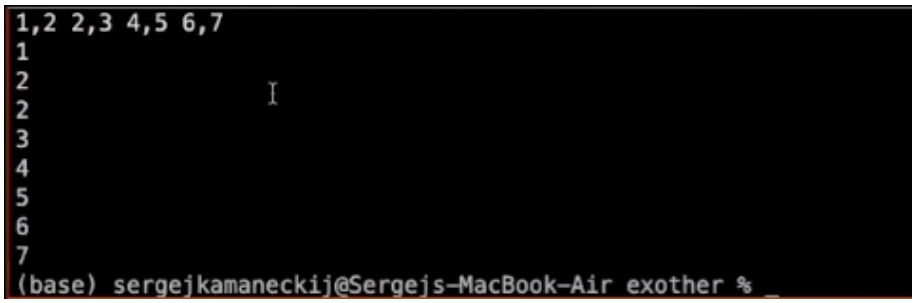
```
 9 var data = text.Split(" ")
10             .Select(item => item.Split(','))
11             .ToArray();
12
```

Когда выполним эту конструкцию, получим массив массивов. Каждым его элементом будет массив из 2-х строк. Давайте проверим.

Сейчас мы наблюдаем просто массив data[i]. Проведу ещё раз внутренний цикл.

```
15     //Console.WriteLine(data[i]);
16     for (int k = 0; k < data[i].Length; k++)
17     {
18         Console.WriteLine(data[i][k]);
19     }
20 }
21
```

Запускаем и смотрим, что получится. В идеале мы ожидаем просто набор чисел, которые показывают наши координаты.



После того как мы получили массив координат, может сделать ещё одну выборку. Сказать: «Давайте мы текущий массив координат превратим в кортеж чисел». При этом мы будем сами делать разбор строки (мы помним, что у нас массив строк). Первый элемент массива нулевой, в качестве второго я буду передавать **int.Parse(e1)**.

```
 9 var data = text.Split(" ")
10                .Select(item => item.Split(','))
11                .Select(e => (int.Parse(e[0]),int.Parse(e[1])))
12                .ToArray();
```

Теперь результатом уже будет не массив массивов, а массив кортежей. Поэтому внутренний цикл нам не нужен, и мы возвращаем всё как было.

```
15 {
16     //Console.WriteLine(data[i]);
17     for (int k = 0; k < data[i].Length; k++)
18     {
19         Console.WriteLine(data[i][k]);
20     }
21     Console.WriteLine();
22 }
```

```
15 {
16     Console.WriteLine(data[i]);
17     Console.WriteLine();
18 }
```

Очищаем экран и запускаем.

```
(1, 2)
(2, 3)
(4, 5)
(6, 7)
```

Круглые скобки остались, но теперь это уже числа. То есть в момент вывода я могу написать Console.WriteLine(data[i].Item1), что даёт нам первую координату, и, соответственно, умножить её, например, на 10.

```
15 {
16     Console.WriteLine(data[i].Item1*10);
17
18 }
```

То есть сейчас мы ожидаем на выходе увидеть 4 числа первых координат, увеличенных в 10 раз: вместо 1, 2, 4, 6 — 10, 20, 40, 60.

```
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
1,2 2,3 4,5 6,7
10
20
40
60
```

Обратите внимание: в коде у нас сейчас какой-то Item1. Что это — непонятно. Поэтому мы можем сказать, пусть int.Parse(e[0]) будет координатой X, а int.Parse(e[1]) — координатой Y:

```
 9 var data = text.Split(" ")
10                .Select(item => item.Split(','))
11                .Select(e => (x: int.Parse(e[0]), y: int.Parse(e[1])))
12                .ToArray();
```

Теперь Item1 можно заменить на X:

```
15 {
16     Console.WriteLine(data[i].x * 10);
17
18 }
```

Data[i] — конкретная точка массива, x — конкретная координата элемента массива, и мы её умножаем на 10. И в консоли наблюдаем то, что должны:

```
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
1,2 2,3 4,5 6,7
10
20
40
60
```

Можно ли как-то сделать по-другому? Технически, если нам нужно один раз и навсегда сделать увеличение этих координат, мы снова можем сделать выборку и дальше сказать: «У нас есть точка (point), и мы хотим превратить её во что-то новое». В данном случае это будет **point.x**, которая умножается на 10. А в качестве второго элемента кортежа будет **point.y**. То есть мы превращаем нашу точку с учётом этих вновь появившихся дополнений.

```
 9 var data = text.Split(" ")
10                .Select(item => item.Split(','))
11                .Select(e => (x: int.Parse(e[0]), y: int.Parse(e[1])))
12                .Select(point => (point.x * 10, point.y))
13                .ToArray();
```

Обновляем и смотрим:

```
1,2 2,3 4,5 6,7
(10, 2)
(20, 3)
(40, 5)
(60, 7)
```

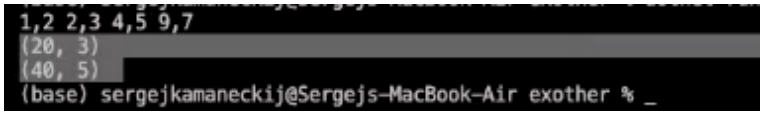
Дальше — больше. Если вам нужно произвести какую-то выборку, прежде чем делать домножение, вы можете сказать: «А давайте мы соберём или получим только те точки, для которых первая координата делится на 2 (то есть координата чётная). Хочу, чтобы выполнялось такое условие.

```
 9 var data = text.Split(" ")
10                .Select(item => item.Split(','))
11                .Select(e => (x: int.Parse(e[0]), y: int.Parse(e[1])))
12                .Where(e=> e.x % 2 == 0)
13                .Select(point => (point.x * 10, point.y))
14                .ToArray();
15
```

Когда будет выполнена 12-я строчка, мы получим набор точек, у которых первая координата чётная, после чего будет производиться домножение полученного набора на 10. Запустим и посмотрим, что происходит.

```
1,2 2,3 4,5 6,7
(20, 3)
(40, 5)
(60, 7)
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _
```

Итак, что мы наблюдаем? У нас есть чётные координаты 2, 4, 6. В итоге мы на выводе видим набор из трёх точек — 20, 40, 60. Если в исходной (3-й) строке вместо 6 напишем 9, в выводе увидим только две координаты:



Друзья, смотрите не на то, насколько сложно написано, а на то, что написано, насколько читаем код.

```
string text = "(1,2) (2,3) (4,5) (9,7)"
```
возьмите текст

```
.Replace("(", "")
.Replace(")", "")
;
```
замените в нём скобки

```
var data = text.Split(" ")
```
разбейте текст с учётом разделителя (в нашем случае — пробела)

```
.Select(item => item.Split(','))
```
сделайте выборку item, для которых нужно взять отдельную подстроку и разбить её на несколько элементов с учётом разделителя-запятой

```
.Select(e => (x: int.Parse(e[0]),
y: int.Parse(e[1])))
```
сделайте выборку из текущего массива, чтобы первой координатой был первый элемент массива (сразу конвертированный в число). То же — со второй

```
.Where(e => e.x % 2 == 0)
```
дайте такие пары, для которых первая координата — чётная

```
.Select(point => (point.x * 10, point.y))
```
дайте набор, который мы получили на предыдущем этапе, и увеличьте первую координату

```
.ToArray();
```
превратите в явный массив

```
16 for (int i = 0; i < data.Length; i++)
17 {
18     Console.WriteLine(data[i]);
19
20 }
```
используйте в цикле с известными вам свойствами и функционалом

Напоминаю, задача не в том, чтобы вы сами писали код. Я лишь даю пример кода, который вы можете наблюдать на Stack Overflow или Хабре. Не пугайтесь: идея принципиально простая — главное уметь читать код.

[00:53:00]

Нотации

Осталось напомнить, что есть разные нотации, которые хорошо бы использовать, когда вы описываете свой код. Среди них: венгерская, Pascal, CamelCase и другие, о которых обычно договаривается команда перед разработкой очередного продукта.

Все правила давно написаны. Осталось найти их, вычитать и использовать на практике. Почему я говорю «найти»? Потому что не все из вас выберут C#, так что нужно будет искать наборы правил для того языка, который вы выберите в качестве основного.

[00:53:41]

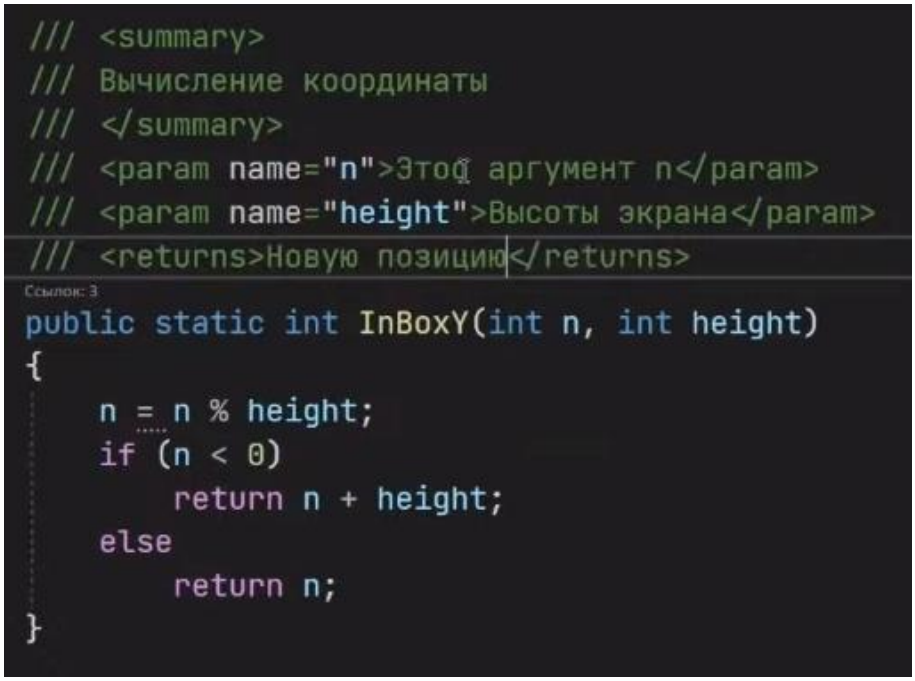
Принципы разработки

Старайтесь придерживаться принципов (Code Convention), которые оговорили с командой. Если большая команда (от 5 человек) будет писать вразнобой, конечный код не соберёт ни один лид. Да и сами вы вряд ли сможете легко переключаться между модулями, если у вас один стиль, а у коллеги другой. Если вы, как джун, приходите на проект, который пишется давно, идите к лиду и спрашивайте, какие есть правила написания продукта. Обычно всё это проговаривается на онбординге в начале первого рабочего дня, но всё же.

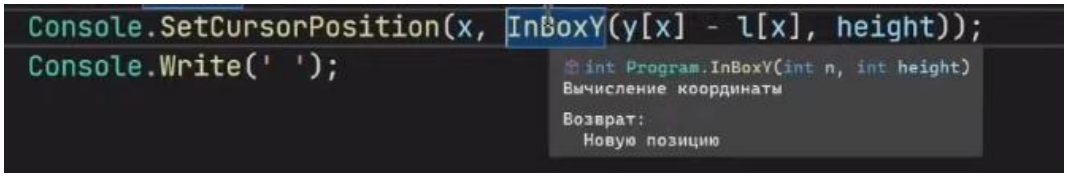
Пишите комментарии. Что я имею в виду? Допустим, в Visual Studio описан какой-то метод. Мы его писали и представляем, что он делает. Если я напишу в нём **///**, Visual Studio автоматически сгенерирует часть xml-документа. Именно здесь мы будем описывать, что делает метод.

```
/// <summary>
/// Вычисление координаты
/// </summary>
/// <param name="n"></param>
/// <param name="height"></param>
/// <returns></returns>
Ссылок: 3
public static int InBoxY(int n, int height)
{
    n = n % height;
    if (n < 0)
        return n + height;
    else
        return n;
}
```

Здесь же можно пояснить аргументы:



В итоге у нас есть полная документация по методу. Если я увижу в коде этот метод, при наведении курсором увижу и те комментарии, которые только что написал.



Дальше — больше. В Visual Studio вы можете зайти в «Свойства» → «Выходные данные» и поставить галочку в чекбоксе «Создание файла, содержащего документацию по API». Затем — запустить сборку проекта или пересобрать его и открыть в папке. Где-то рядом с другими файлами будет файл с текстом, который вы документировали. Это удобно для технических писателей, или когда по вашему коду нужно составлять отдельную документацию. Но это справедливо для .NET-платформы и языка C# в частности. Если вы используете другой язык или другие средства для разработки, всё может быть иначе.

Вернёмся к проблеме комментариев. Я сторонник комментариев, отражающих суть кода. Не нужно писать о том, что здесь вы объявили переменную целого типа (их можно писать для себя, но в конечном коде такого быть не должно).

У меня есть товарищ, опытный разработчик. Он говорит так: «Если нужен комментарий, объясняющий работу кода, скорее всего, код нужно переделывать из-за его непонятности и сложности». Я частично с ним согласен. Если вы делаете внутренние сервисы, а не продукты на сторону, возможно, комментарии не нужны. Но если вы делаете публичное API, которым будут пользоваться сотни тысяч людей, документация нужна. За неё вам скажут спасибо сторонние разработчики или простые фрилансеры, которые сервис может показаться полезным.

Производите внутреннюю декомпозицию. Это справедливо и для отдельных методов, и для больших сущностей: классов, структур, записей и так далее. Чем меньше единица, которую вы используете, тем лучше. Тем проще её понять, переписать, заменить и так далее.

Чем больше у вас методов, тем лучше, потому что вы уже знаете, для чего нужна декомпозиция.

Чем больше вы используете циклов, тем хуже ваш код. Когда вы начинаете писать цикл внутри цикла, внутри цикла, внутри цикла, — это маркер того, что ваш код будет работать очень медленно.

Идёт отсылка в сложность алгоритмов, это отдельная тема, где математику нужно знать больше, чем программирование. Но помните о том, что не нужно стараться бесконечно вкладывать циклы в циклы и ветвления в ветвления, потому что есть определённые анализаторы, которые смотрят на ваш код и говорят: «Друг, что-то у тебя с кодом не очень хорошо».

Если вы пишете, какой-то метод, для него сразу должен быть тест. Тесты — тема достаточно интересная, на семинарах можно будет заняться ими более плотно. Для чего нужно описывать тесты? Дело в том, что большие продукты, которые пилятся не одной командой разработки, невозможно собрать в кучу без предварительной проверки того, что действительно код в связке с предыдущими модулями хорошо себя показывает и работает. Есть определённые механизмы, которые в узких кругах зовутся как CI/CD, но сейчас мы не будем их обсуждать.

Но помните, что писать тесты — хорошо и правильно. В тестовых заданиях хорошо демонстрировать, что вы знаете, что такое тесты и примерно представляете, как они пишутся.

Пользовательские данные — на каждом этапе важно следить за тем, что вам подсовывает пользователь, как-то это обрабатывать. Много времени тратится именно на организацию пользовательского ввода. Часто это перекладывается на фронтендеров, которые должны делать валидацию всего и вся. Но они могут что-то пропустить и, когда бэкендер начинает работать с вроде как валидными данными, получается чепуха. С одной стороны, казалось бы, просто организовать ввод данных от пользователя. Но, с другой стороны, в каком формате эти данные придут? Как разобрать их правильно?

И главный вопрос, который я рекомендую задавать на каждом этапе разработки, после каждого написанного метода, — **можно ли сделать лучше и проще?** Пожалуйста, почаще задавайте себе этот вопрос, тогда ваш код станет более читаемым, более простым.

[01:02:00]

Итоги

Есть принципы, которых нужно придерживаться при разработке:

- Придерживаться SOLID
- Не использовать антипаттены
- Использовать паттерны
- Декомпозировать
- Писать тесты
- Придерживаться Code Convention

В какой-то степени это философские темы, которые уходят в архитектуру построения приложений, но лучше их знать, чем не знать.

Когда вы начинаете писать код (пусть даже простой), хорошо, если вы знаете, к чему нужно стремиться. В этом случае я вам рекомендую посмотреть, что такое SOLID-принципы, что такое антипаттерны, какие антипаттерны бывают, какие бывают паттерны. Можете прямо погуглить: «Банда четырёх, каталог GoF». Это Библия того, как нужно писать красивый код и строить системы, которые будут гибкими, расширяемыми и поддерживаемыми.

Всегда помните о декомпозиции практически всего. Если в методе больше 20 строчек, скорее всего, нужно его переписать или разделить на несколько. С классами и другим также — можно улучшить, упростить, сделать меньше и так далее.

И всегда пишите тесты. Помним о том, что, когда вы начинаете командную разработку, неважно, будь это реальный продакшн или учебные задачи, договаривайтесь, как вы будете описывать методы, как

вы будете называть переменные, чтобы результаты работы нескольких людей можно было объединить.

[01:04:12]

Заключение

Друзья, я искренне надеюсь, что две лекции из серии «Как не нужно писать код» помогли вам узнать что-то новое и получить представление о том, как писать код. Надеюсь, вы будете использовать эти практики пусть даже в учебных проектах. А когда попадёте в большой продакшн, будете использовать точно, потому что без них вас, скорее всего, не допустят к идеальному продукту. К тому продукту, который будет создавать компания.

На этом я с вами прощаюсь. Всем счастливого, всем пока-пока.