# Жизненный цикл ПО

*Жизненный Цикл По* – это период времени, начинающийся с момента принятия решения о необходимости создания ПО и заканчивающийся в момент его полного изъятия из эксплуатации.

Этапы жизненного цикла ПО в соответствии с каскадной моделью разработки ПО (Waterfall):

1. Формирование требований

Включает в себя:

* сбор и анализ требований, их документирование,
* планирования,
* бизнес-моделирование.

В результате оформляются в виде Т3 (Технического задания).

1. *Проектирование* – это процесс описания схем алгоритма и архитектуры приложения.

* ERD
* UHL

1. Дизайн

Способы представления интерфейса:

* Wireframe (каркас)
* Mockup

Прототип: figma

1. Разработка (написание кода)

На выходе исходный код, исполняемый файл или библиотека (для компилируемых языков)

C/C#/C++, Pascal – компилируемые

Python, JS, PHP – интротеритируемые

1. Тестирование и отладка

*Тестирование* – это проверка работы программы на наборе тестов.

*Отладка* – это тестирование, локализация ошибок и их исправление.

1. *Внедрение* – этодокументирование, сертификация, настройка ПО
2. Эксплантация и сопровождение

# Критерии качества ПО

*Качество ПО* – это способность ПО отвечать установленным или предложенным требованиям или потребностям.

*Критерий качества ПО* – это требования к программе, которые обычно не описываются в договоре с заказчиком, но являются желательными, так как повышают качество ПО.

Критерии:

1. *Функциональность* – ПО решает поставленную задачу согласно заключенным или подразумеваем потребностям.
2. *Usability* (удобство использования) – возможность легкого понимания, изучения и использования ПО, а также привлекательность ПО для пользователя.

(UI/UX)

*User Interface* (тренды расположения и отображения интерфейса)

*User eXperience* (опыт пользователя)

1. *Производительность* – способность ПО обеспечивать требуемую производительность относительно качества минимально заявленных ресурсов.
2. *Надёжность* – способность По восстанавливаться после сбоев и его отказоустойчивости.

try-catch

if-else

1. *Защищенность* – способность ПО прекращать несанкционированный доступ к данным.
2. *Мобильность и переносимость* – способность ПО позволять перенести его из одной среды в другую.
3. *Совместимость* – способность ПО взаимодействовать уже с установленным.
4. *Сопровождаемость* – обеспечивает легкость внесения изменений ПО для его модификации и улучшения.

# РЕФАКТОРИНГ

*Рефакторинг* – это процесс изменения структуры программы, не изменяющий её поведения, но облегчающий процесс понимания кода.

Книги полезные для рефакторинга:

* С. Макконалл – «Совершенный код»,
* Р. Мартин (Java) – «Чистый код»,
* М. Фаулер – «Рефакторинг».

Сайт refactorinr.guru

Признаки плохого кода:

1. Неинформативные названия,
2. Дублирование кода,
3. Длинные методы,
4. Денный список параметров: (не более 3х),
5. Избыточное количество временных переменных,
6. Большой класс,
7. Не сгруппированные данные.

Методы рефакторинга:

1. Переименовать CTRL+R+R
2. Выделение метода CTRL+R+M
3. Изменение сигнатуры метода (изменить подпись)
4. *Сигнатура метода* – это имя метода и список его параметров
5. Выделение класса
6. Выделение интерфейса
7. Перемещение метода
8. *Инкапсуляция кода* – это замена открытого поля закрытыми с добавлением свойств или методов доступных к полю

## Стандарты оформления кода

*CodeStyle* – набор правил и соглашений использующийся при написании кода на определенном языке программирования.

Стили наименования:

* **PascalCase** – каждое слово в заглавной буквы.

В С# используется для типов данных и методов

Console.WriteLine

BackColor

UserRole

* **camelCase** – первое слово с маленькой буквы остальные с заглавной.

У переменных, параметров метода.

a, i

minWidth

LoginTextBox

* **Венгерская нотация** – в названии есть указание на табличные данные.

Элементы управления,

ProductsСontroller

I Printer - IntertacePrinter.

* **snake\_case** – слова отделяются друг от друга без учета регистра.

SaveButton\_Click

элемент\_cобытия.

\_login

Закрытые поля

\_userRole

* **SCREAMING\_SNAKE\_CASE** – все буквы заглавные с разделением.

константы

PI

MAX\_HELGT

## Форматирование кода

Форматирование кода – это расстановка скобок, отступов, пустых строк между методами полями и др. компонентами кода.

**Способы форматирования кода с VS2022:**

* Настроить метёлочку и нажимать на неё (располагается внизу экрана, где пишет ошибки и предупреждения)
* CTRL+K+Е
* Ручной метод (удаляем и сами заново ставим скобку)

## Комментирование кода

*Само документируемый код* – код, не требующий наличия комментариев.

Недостаток комментариев: после изменения кода комментарии могут противоречить коду.

//

/\*….\*/

/// XML-комментарий (документация) – позволяет воздать всплывающие подсказки и выполнить документирование кода.

**<summary>** (описание элемента) **</summary> обязательный**

**<param** (атрибут)**>** (описание параметра)**</param>**

**<returns>** (описание возвращающегося значения)**</returns>**

# Тестирование и отладка

Назначение тестирования:

1. Повышает вероятность корректности работы программы,
2. Повышает вероятность соответствия приложения определённым требованиям,
3. Предоставляет актуальную информацию текущем состояния приложения.

Примеры видов тестирования:

1. Ручное тестирование
2. Автоматизированное тестирование – тестирование с использованием специального ПО позволяющее автоматизировать рутинные действия.
3. Примеры: Unit-тестирование (модульное),
4. Интеграционное-тестирование.
5. Тестирование методом белого (или стеклянного ящика) – это структурное тестирование, требующее знания структуры программы и доступ к коду.
6. Тестирование методом черного ящика – это тестирование функциональности ПО, выполняется без доступа к коду.
7. Пример: Betatest

*Набор тестов* – набор данных или последовательности действий, для которых заранее известен результат применения. Обычно представляется в виде таблицы.

1-й столбец входные данные (действия)

2-й столбец ожидаемый результат

3-й столбец полученный результат

До создания программы заполняется первые два столбика.

*test-case* – алгоритм действий при проверке

*bug report* – описание ошибки

Таблица 1 – Набор тестов метода вычисления факториала

|  |  |  |
| --- | --- | --- |
| Входные данные | Ожидаемый результат | Полученный результат |
| -1 | 0 |  |
| 0 | 1 |  |
| 1 | 1 |  |
| 5 | 120 |  |

Для некорректных данных можно определить:

1. Заведомо некорректные значения,
2. Код ошибки,
3. Генерация исключений.

throw new Exception();

Желательно выбирать входные данные и результат так чтоб их значения не совпадали.

Если входных данных несколько:

|  |  |  |
| --- | --- | --- |
| Входные данные | Ожидаемый результат | Полученный результат |
| a=3, b=2 | 5 |  |

или

|  |  |  |  |
| --- | --- | --- | --- |
| Входные данные | | Ожидаемый результат | Полученный результат |
| A | B |
| 3 | 2 | 5 |  |

Если в первом столбце указывается действие, то оно должно полностью описывать всю последовательность действий в одном стиле использовать глагол или отглагольное существительное.

Пример: тестирование страницы авторизации

1. Ввод номера телефона (корректно) +71234567890
2. Нажатие на кнопку «Войти»

Результат: перенаправление на экран ввода пароля

или же

1. Ввести номер телефона +71234567890
2. Нажать на кнопку «Войти»

Результат: перенаправление на экран ввода пароля

Существует генерация модульных тестов

ПКМ Создание модульных тестов.

AAA

* Arrange – подготовка (ожидаемый результат),
* Act – выполнение действия,
* Assert – проверка результата действия.

Методы класса Accept:

*AreEqual* – проверка isTrue/isFalse/isNull

*ThrowsException* – генерируется ли исключение.

Пример №1:

//Arrange

int a = 2;

int b = 3

int expected = 5;

//Act

int actual = Math.Sum(a,b);

//Assert

Assert.AreEqual(expected, actual);

Пример №2:

[TestMethod]

[DataTestMethod]

[DataRow(0,1)]

[DataRow(1,1)]

[DataRow(5,120)]

*Trace.Console.WriteLine()* – можно вывести не только в отладке.

*Debug.Console.WriteLine()* – можно вывести только в отладке.

# Технологии структурного программирования

*Структурное программирование* – это парадигма программирования в основе которой лежит представление программа в виде иерархической структуры блоков.

Структура программа может быть приставлена в виде схемы алгоритма.

goto не рекомендуется использовать!

Управляющую структуру или конструкцию, из которой состоит программа:

1. Последовательность,
2. Ветвление,
3. Циклический,
4. *Подпрограмма* (или вложенный алгоритм) – позволяет использовать алгоритм в разных программах.

Для проектирования схем алгоритмов можно использовать следующее ПО:

1. Visio
2. LibreOfficeDraw
3. Draw.IO

Для оформления схемы алгоритмов используется стандарт ГОСТ 19.701-90

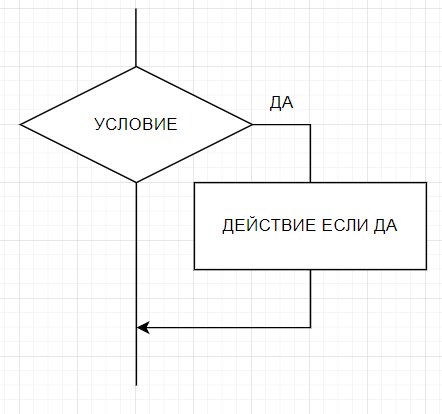
* Если направление в схеме естественное ( ) ,то стрелки не нужны,
* У большинства элементов вход сверху, а выход снизу,
* У элементов должно быть соотношение сторон 1:1,5 или 1:2,
* У начала и конца высота в 2 раза меньше, чем у обычных элементов.

Примеры элементов:

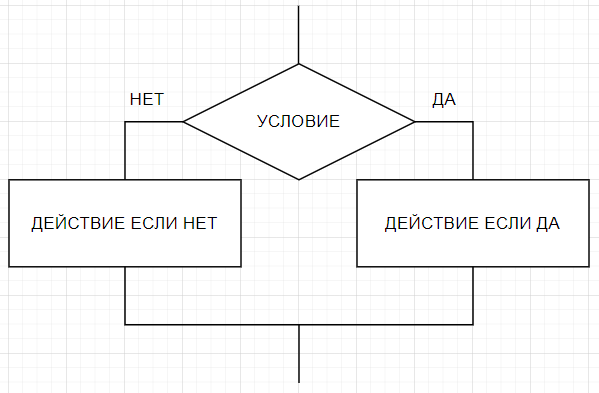
|  |  |
| --- | --- |
|  | Начало/Конец |
|  | Терминатор (для продолжения на другом листе) |
|  | Действие |
|  | Подпрограмма |
|  | Программный ввод |
|  | Ввод с клавиатуры |
|  | Вывод на экран |
|  | Проверка условия |
|  | Блок счётчика |

## Условные алгоритмы

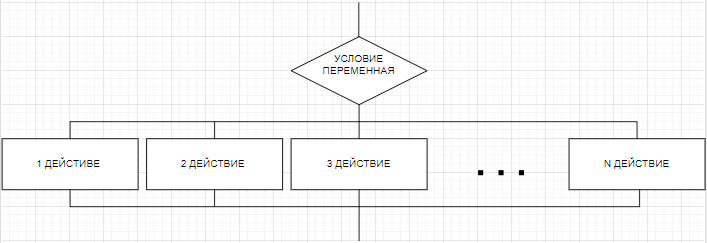
if:



if-else:

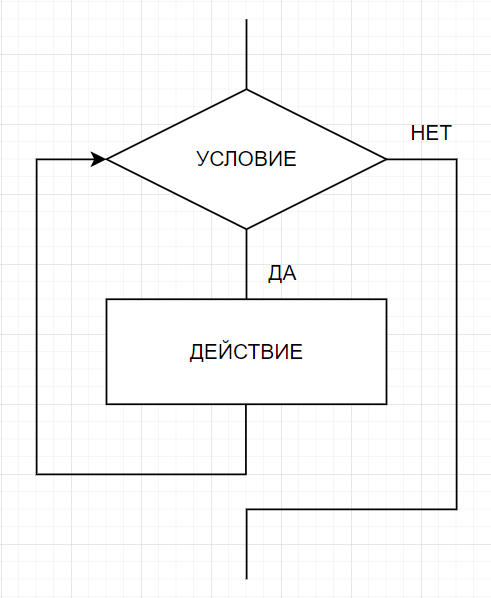


switch-case:

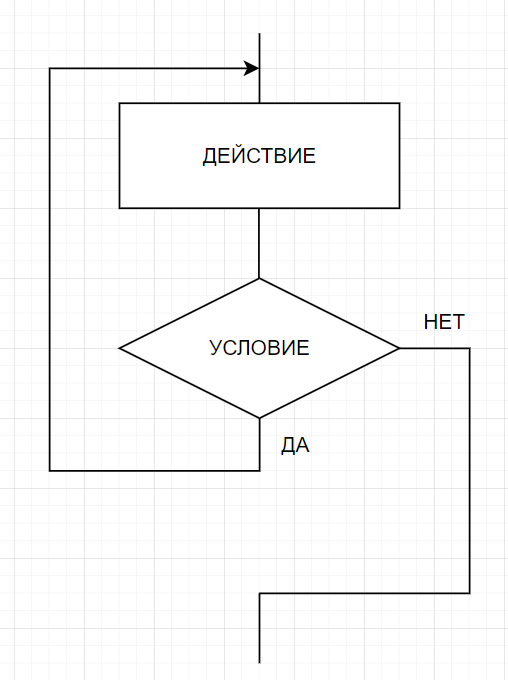


Циклические алгоритмы

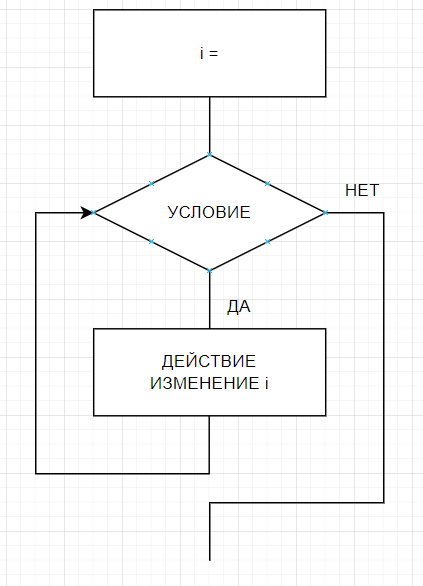
while:



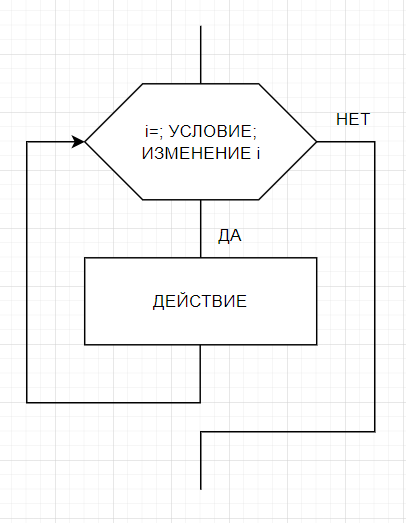
do-while:



for (вариант 1):



for (вариант 2):



## Снипеты в C#

*Снипет* (snippet) – это заготовка или шаблон кода.

Для вызова нужно написать часть названия snippeta и нажать Tab.

Примеры снипетов:

1. CW + Tab (консольное)
2. mbox + Tab (оконное)
3. if + Tab
4. else + Tab
5. swich + Tab
6. while + Tab
7. do + Tab
8. for + Tab (счетчик по возрастанию)
9. forr + Tab (счётчик по убыванию)
10. foreach + Tab
11. try + Tab
12. tryf + Tab

Тернарный оператор:

(условие) ? результат1 : резульатат2 (if-else)

swich возвращающий результат:

переменная swich

{

значение1 => результат1,

значение2 => результат2,

\_ => результатПоУмолчанию,

}

# Оценка сложности алгоритмов

Основные свойства алгоритма:

1. Массовость. Алгоритм решает задачи на некотором множестве доступных данных.
2. Конечность. Алгоритм должен завершаться за конечное число шагов.

Примечание: не для всех задач существует алгоритма решения. Даже если алгоритма существует, то он может быть не применим для успешного решения задачи из-за высокой сложности.

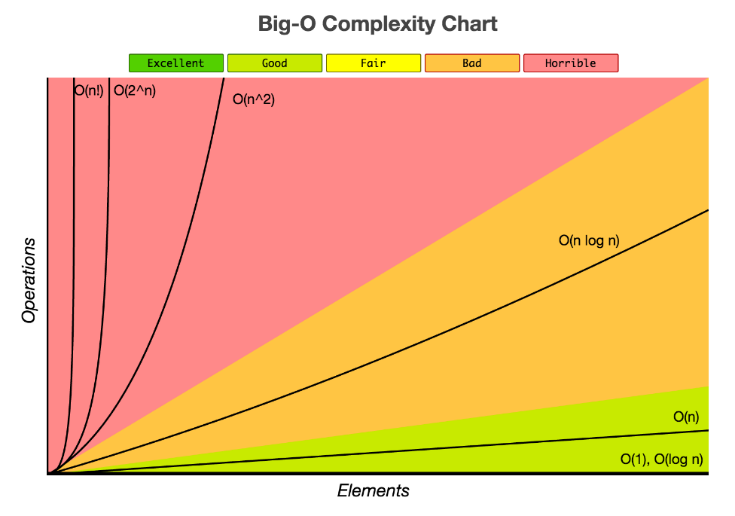
*Сложность алгоритма* – это количественная характеристика ресурсов необходимых алгоритму для успешного решения задачи.

Основные ресурсы:

1. Время – временная сложность
2. Память – емкостная сложность

Сложность задачи может быть разной для разных входных данных. Обычно оценивают порядок роста сложности при N стремящейся к бесконечности. Обычно учитывается сложность для худшего случая.

*BigO* – это мера фиктивности для описания сложности обозначающая сколько времени памяти необходимо для решения задачи. (Обычно для худшего случая)



Виды сложности:

1. Константное О(c) или O(1)

Например: получение элемента массива по индексу

1. Логарифмическое О(log n)

Например: бинарный поиск (работает только с отсортированными данными)

1. Линейная О(n)

Например: поиск перебором

1. Линейно-логарифмическая О(log 2n)

Например: быстрая сортировка

1. Квадратическая О(n2)

Например: сортировка пузырьком

1. Экспоненциальная О(2n)

Например: получение списка подмассивов

1. Факториальная О(n!)

Например: список простановок

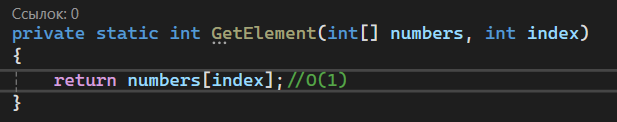
* Количество вводных данных не учитывается в сложности,
* Множитель можно пропускать если он не влияет на сложность.
* При вычислении сложности берётся только максимальная.

O(n!) + O(n) = O(n!)

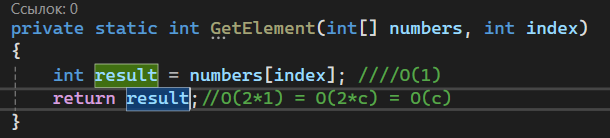
O(n2) + O(2n) = O(2n)

Примеры оценки сложности алгоритма:

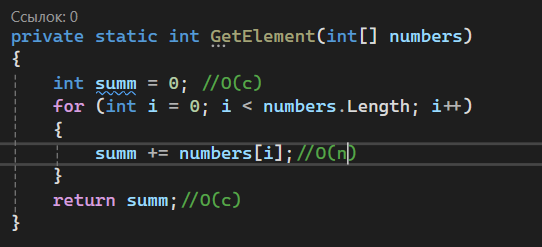
Сложность О(1)



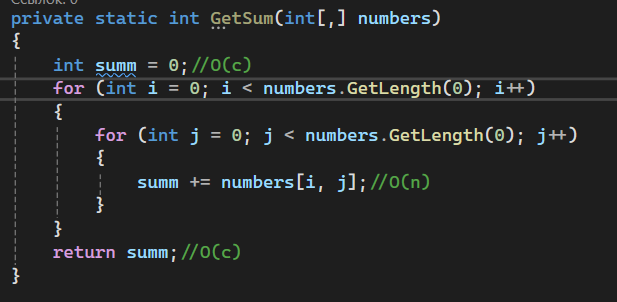
Сложность О(с)



Сложность О(с)



Сложность О(n2)

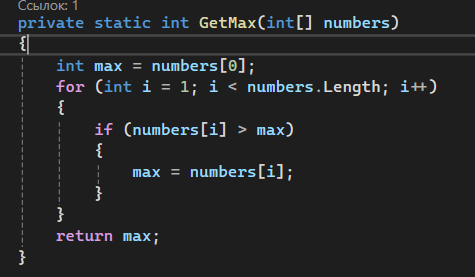


Вложенные сложности перемешиваются.

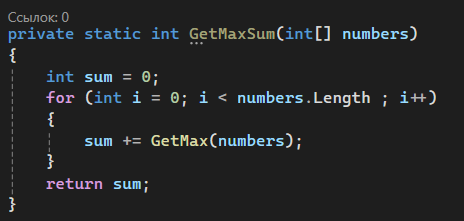
Например: цикл в цикле, вызов метода, вызов метода в другом методе.

Пример вызова метода в методе:

Метод GetMax



Использование метода GetMax в методе GetMaxSum



*Многократная рекурсия* – это рекурсивный алгоритм, который вызывается в себе несколько раз.

*Класс сложности* – это множество задач, для решения которых существует алгоритм сложности схожий по высшей сложности.