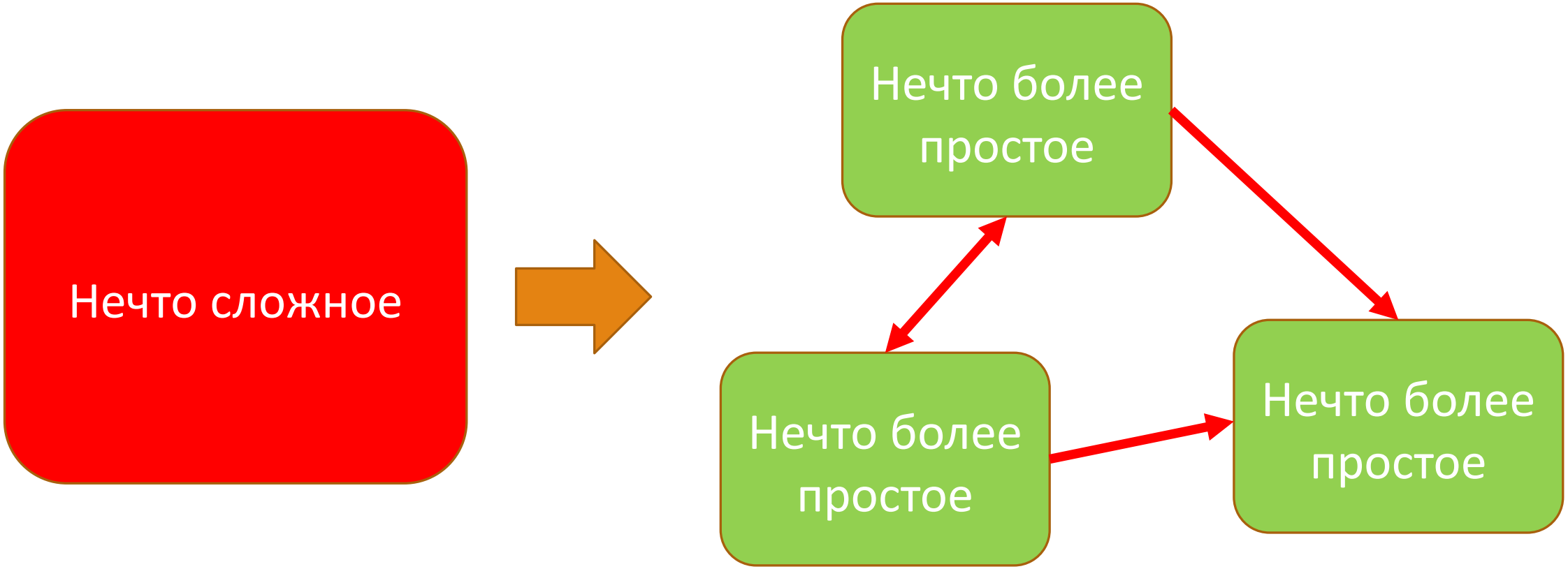
Основы программного конструирования

ЛЕКЦИЯ №5

13 МАРТА 2022

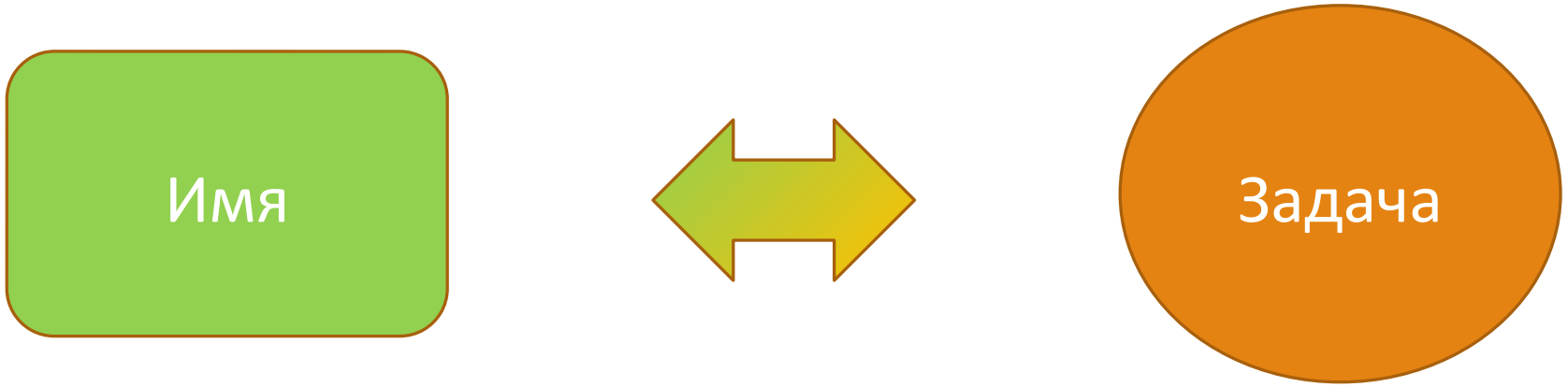
Декомпозиция



Простейшие способы декомпозиции

- Взаимодействие с пользователем (**U**ser **I**nterface) – отдельный модуль.
- Обработка разных структур данных – в разных модулях (**list**, **stack**, **queue**, …).
- Элементы функциональности, близкие по смыслу – в один модуль (**io**, **parse**, **errors**).
- Если процесс естественным образом разбивается на отдельные шаги, то каждый шаг – в свой модуль (**prepare**, **get_data**, **process**, **output**).

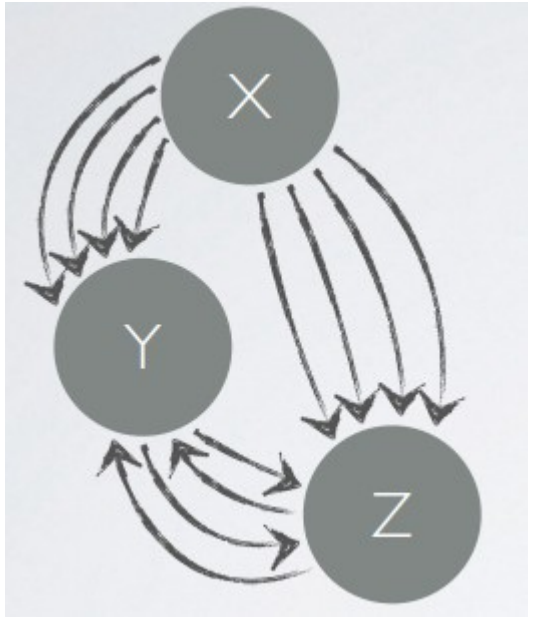
Первое представление модуля

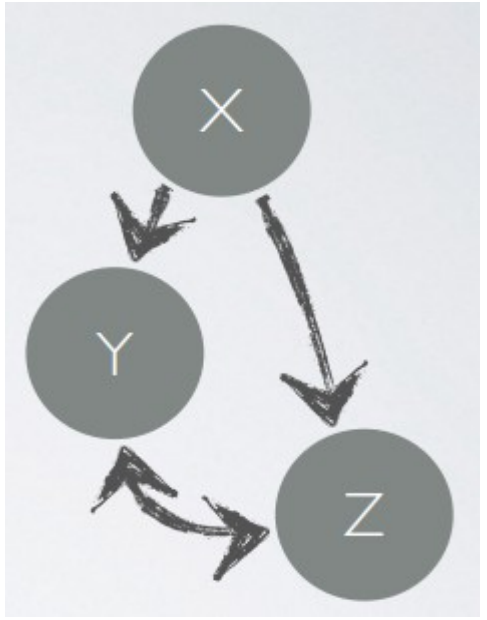


Уточнение представления

- Модули состоят из функций.
- Функции вызывают другие функции, из своего модуля и из других модулей.
- Вызовы функций – связи между модулями.

Картинки связей

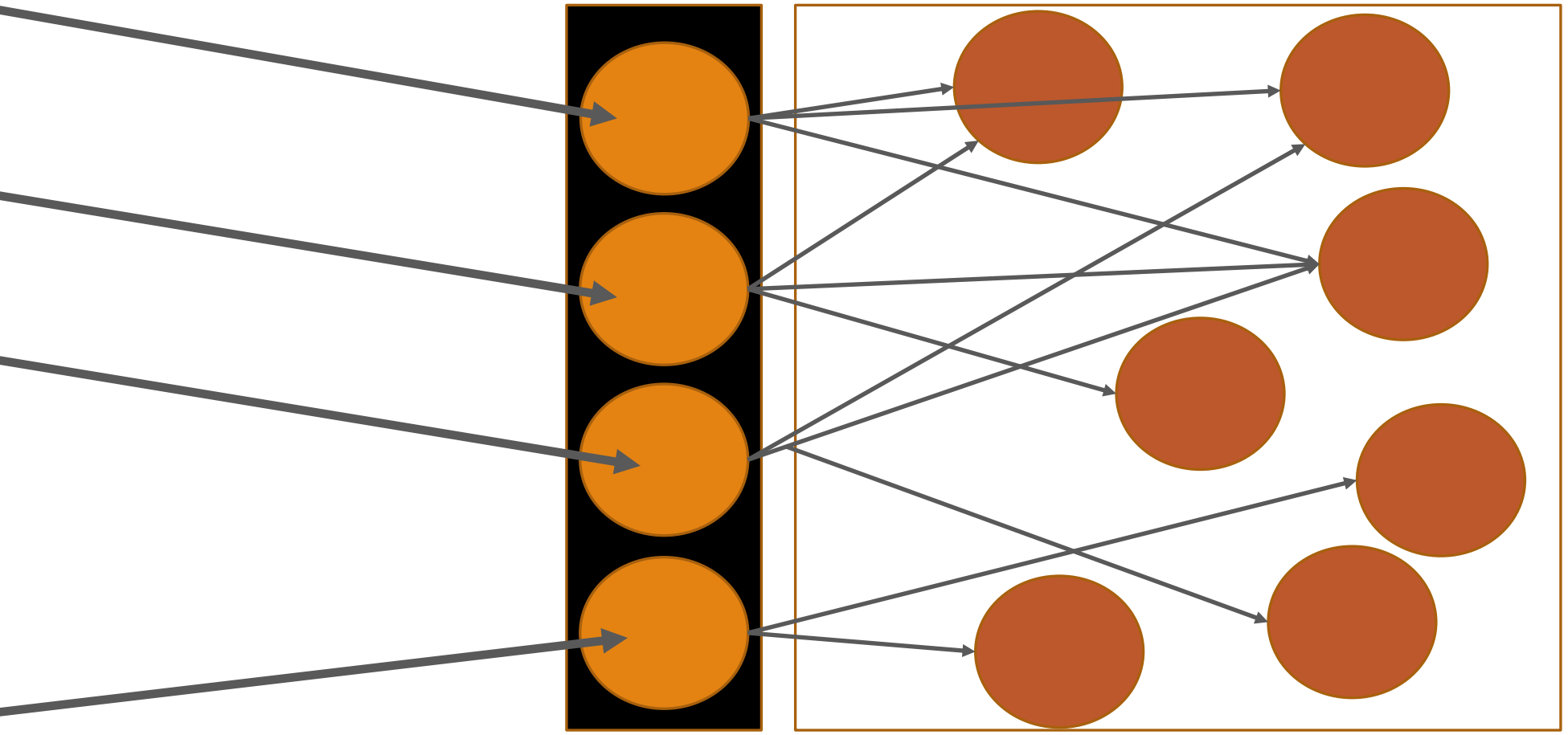




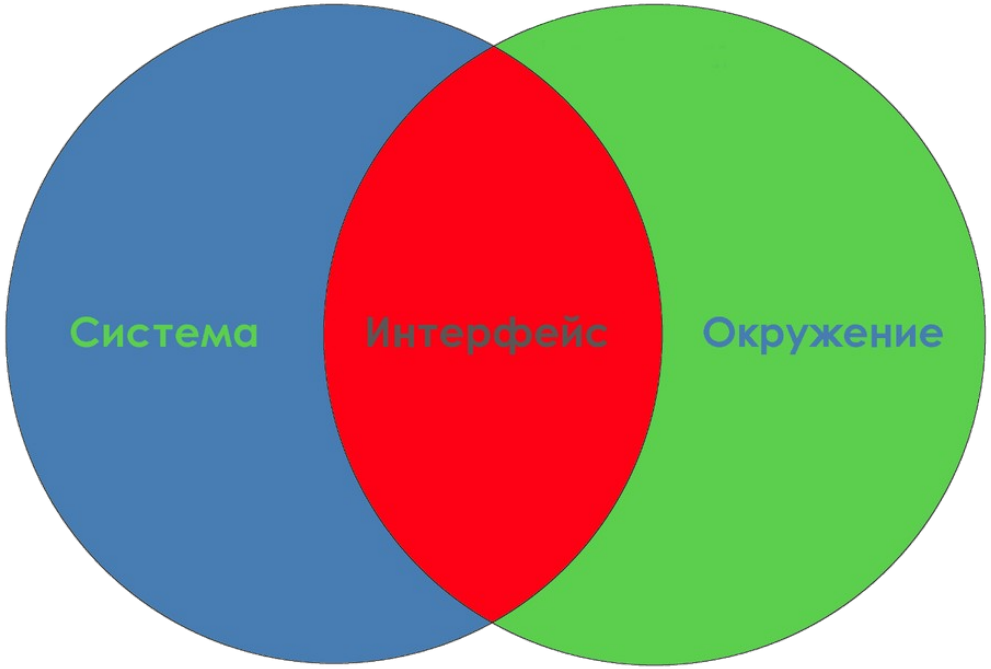
Что проще?

Мера сложности системы – количество связей.

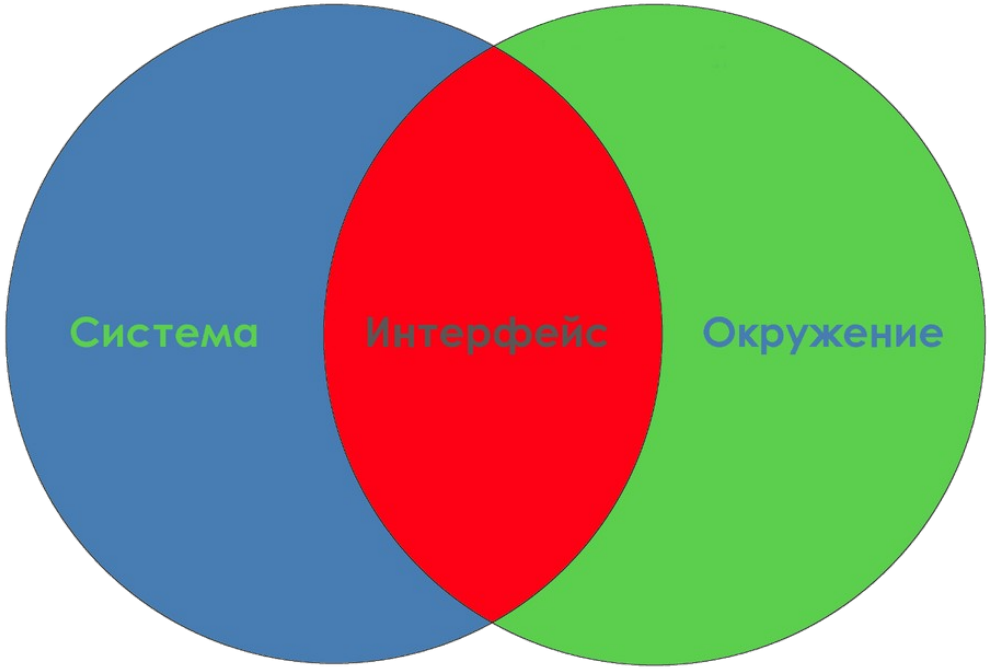
Меры против сложности



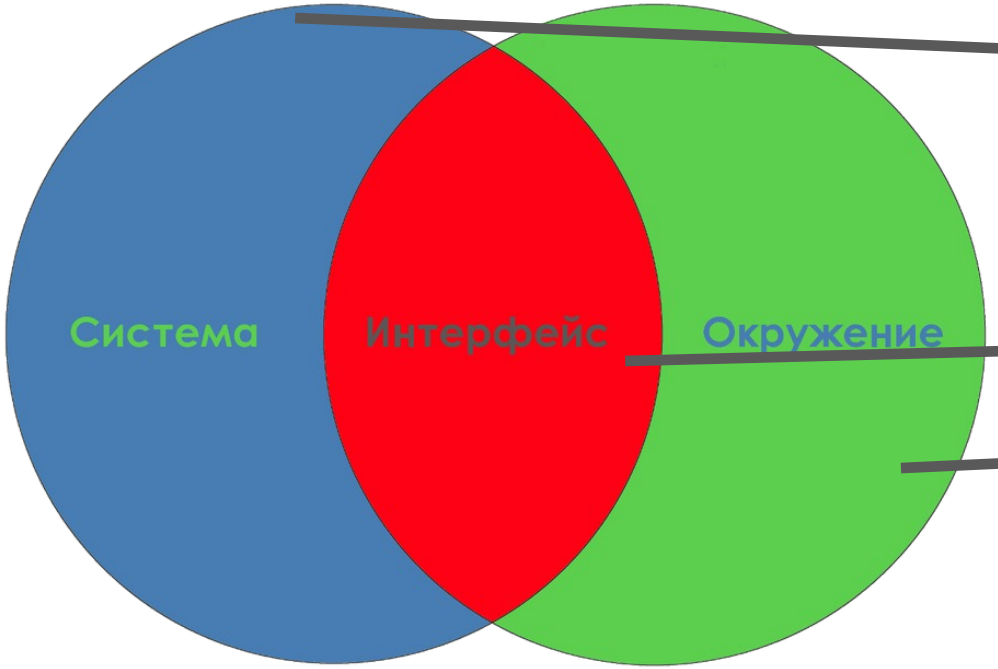
Интерфейс



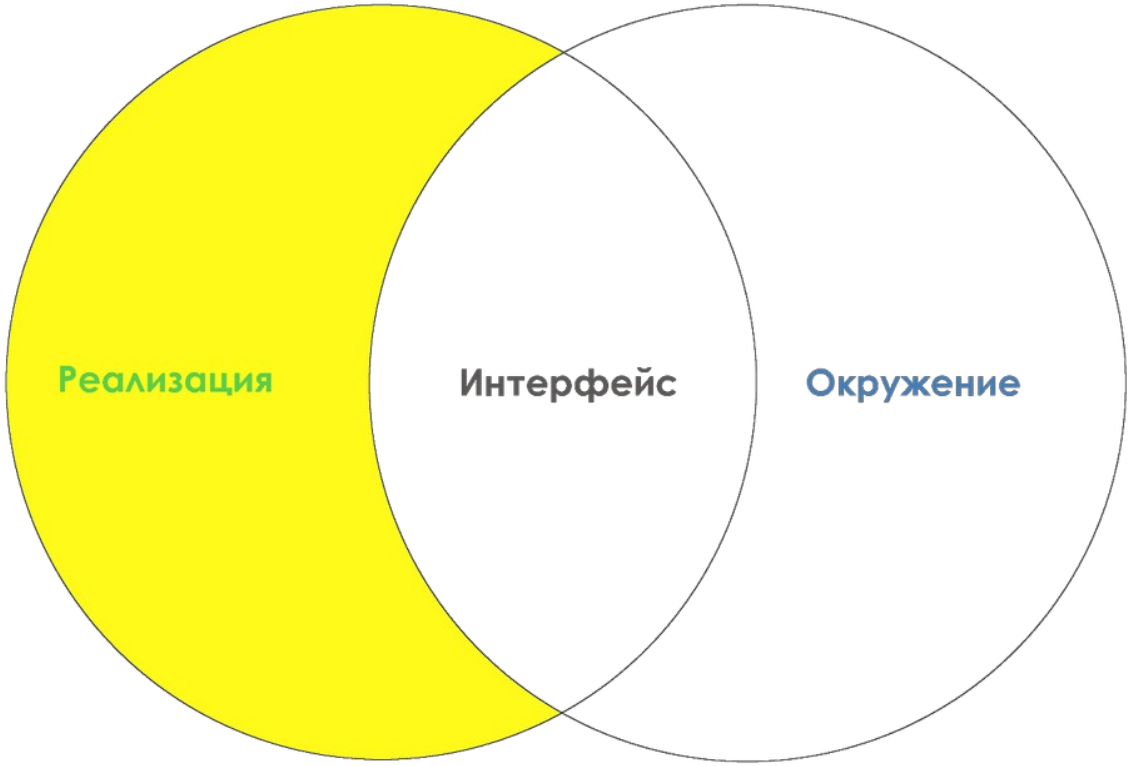
Интерфейс



Интерфейс



Реализация



Интерфейс и реализация

Интерфейс и реализация

Адресная книга:

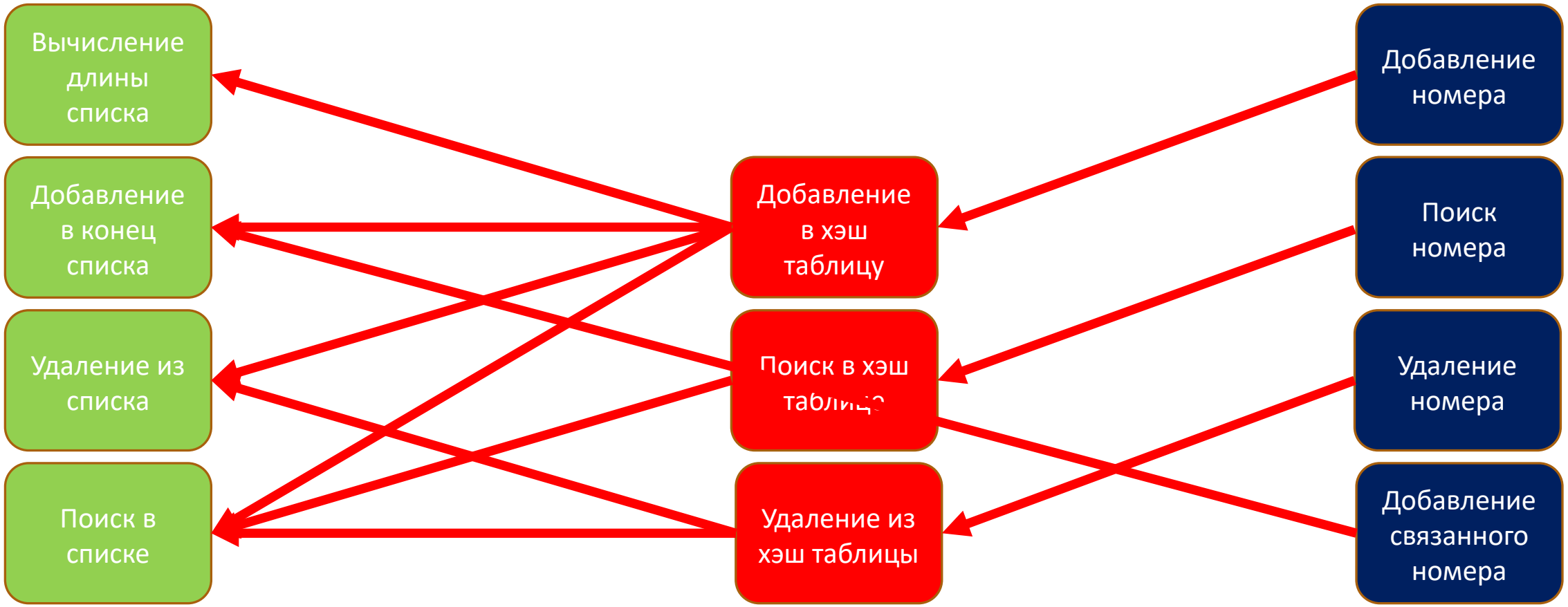
- **Интерфейс**:
- **Реализация:**

Интерфейс и реализация

Адресная книга:

- **Интерфейс**: Операции (добавить\удалить запись, найти по имени).
- **Реализация** операций: код функций, основанный на конкретном представлении хранилища и выбранных алгоритмах, а также вспомогательные функции.

Картинки связей



Устройство модулей в Python

- Каждый python-файл можно использовать как модуль.
- Вспомогательные сущности именуются с подстрочника.
- Использование модуля:

```
import mymodule
mymodule.myfunc()


from mymodule import myfunc, myconst
myfunc()
```

Equsolver.py

```
from math import sqrt

def solve2(a, b, c):
    if a != 0:
        return _solve2_real(a, b, c)
    else:
        return solve1(b, c)
def _solve_real(a, b, c):
    d = b*b - 4*a*c
    if d > 0:
        x_1 = (-b - sqrt(d) / (2 * a))
        x_1 = (-b + sqrt(d) / (2 * a))
        return [x_1, x_2]
    else:
        ...

def solve1(a, b):
    ...
```

Main.py

```
import equsolver

roots = equsolver.solve2(1, -5, 6)
print(roots)
```

Стандартная библиотека Python

- **math** – математические функции.
- **time** – функции работы со временем и датой.
- **random** – функции для генерации случайных чисел.
- **sys** – функции для взаимодействия со средой исполнения.
- **os** – функции для работы с операционной системой.

Что важнее интерфейс или реализация?

Почему интерфейс важнее реализации?

- На один интерфейс может быть несколько реализаций (но не наоборот!).
- Если интерфейс удачный, модуль легче повторно использовать.
- Если интерфейс выпущен в широкую публику, его гораздо сложнее изменить.
- Интерфейс – это то, что увидят другие, реализацию могут и не увидеть.

Что тут происходит?!?!

```
from random import randint
a = int(input())
b = [None]*a
bb = [None] * a
for c in range(0, a):
    b[c] = randint(0,100)
for d in range(0, a):
    print(str(d)+"->"+str(b[d]))
e = 0
for q in range(0, len(b)):
    e += b[q]
e = e / a
O = 0
print(e)
for w in range(0, a):
    bb[w] = int(input())
while (O < a):
    l = 0
    while (l < a - O - 1):
        if (b[l]*bb[l]>b[l+1]*bb[l+1]):
           b[l], b[l+1] = b[l+1], b[l]
           bb[l], bb[l+1] = bb[l+1], bb[l]
        l += 1
    O += 1
for d in range(0, a):
    print(str(d)+"->"+str(b[d]))
```

- Назначение переменных.
- Назначение циклов.
- Что в целом делает код (по пунктам 1, 2, 3…).
- Ваше впечатление от увиденного?

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

```
from random import randint
n = int(input())
arr = [None] * n
weights = [None] * n
for i in range(0, n):
    arr[i] = randint(0, 100)
for i in range(0, n):
    print(str(i)+"->"+str(arr[i]))
average = 0
for i in range(0, len(arr)):
    average += arr[i]
average = average / n
print(average)

for i in range(0, n):
    weights[i] = int(input())
i = 0
while (i < n):
    j = 0
    while (j < n - i - 1):
        if (arr[j]*weights[j]>arr[j+1]*weights[j+1]):
            arr[j], arr[j+1] = arr[j+1], arr[j]
            weights[j], weights[j+1] = weights[j+1], weights[j]
        j += 1
    i += 1
for i in range(0, n):
    print(str(i)+"->"+str(arr[i]))
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) ~~Разделили код на функциональные части.~~ Сделали хуже.

```
n, arr, weights = f1()
f2(arr)
f3(arr)
f4(n, weights)
f5(arr, weights)
f2(arr)
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) Разделили код на функциональные части.

```
n, arr, weights = init()
print_array(arr)
print_average(arr)
read_weights(n, weights)
bubble_sort_modified(arr, weights)
print_array(arr)
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) Разделили код на функциональные части.

```
def print_array(arr):
    for i in range(0, n):
        print(str(i) + "->" + str(arr[i]))
```

```
def print_average(arr):
    average = 0
    for i in range(0, len(arr)):
        average += arr[i]
    average = average / n
    print(average)
```

```
def read_weights(n, weights):
    for i in range(0, n):
        weights[i] = int(input())
```

```
def init():
    n = int(input())
    arr = [None] * n
    weights = [None] * n
    rand_array(arr, n)
    return n, arr, weights
```

```
def rand_array(arr, n):
    for i in range(0, n):
        arr[i] = randint(0, 100)
```

```
n, arr, weights = init()
print_array(arr)
print_average(arr)
read_weights(n, weights)
bubble_sort_modified(arr, weights)
print_array(arr)
```

```
def bubble_sort_modified(arr, weights):
    i = 0
    while (i < n):
        j = 0
        while (j < n - i - 1):
            if (arr[j] * weights[j] > arr[j + 1] * weights[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                weights[j], weights[j + 1] = weights[j + 1], weights[j]
            j += 1
        i += 1
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) Разделили код на функциональные части.

```
def print_array(arr):
    for i in range(0, n):
        print(str(i) + "->" + str(arr[i]))
```

```
def print_average(arr):
    average = 0
    for i in range(0, len(arr)):
        average += arr[i]
    average = average / n
    print(average)
```

```
def read_weights(n, weights):
    for i in range(0, n):
        weights[i] = int(input())
```

```
def init():
    n = int(input())
    arr = [None] * n
    weights = [None] * n
    rand_array(arr, n)
    return n, arr, weights
```

```
def rand_array(arr, n):
    for i in range(0, n):
        arr[i] = randint(0, 100)
```

```
n, arr, weights = init()
print_array(arr)
print_average(arr)
read_weights(n, weights)
bubble_sort_modified(arr, weights)
print_array(arr)
```

```
def bubble_sort_modified(arr, weights):
    i = 0
    while (i < n):
        j = 0
        while (j < n - i - 1):
            if (arr[j] * weights[j] > arr[j + 1] * weights[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                weights[j], weights[j + 1] = weights[j + 1], weights[j]
            j += 1
        i += 1
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) Разделили код на функциональные части.

```
def print_array(arr):
    for i in range(0, n):
        print(str(i) + "->" + str(arr[i]))
```

```
def print_average(arr):
    average = 0
    for i in range(0, len(arr)):
        average += arr[i]
    average = average / n
    print(average)
```

```
def read_weights(n, weights):
    for i in range(0, n):
        weights[i] = int(input())
```

```
def init():
    n = int(input())
    arr = [None] * n
    weights = [None] * n
    rand_array(arr, n)
    return n, arr, weights
```

```
def rand_array(arr, n):
    for i in range(0, n):
        arr[i] = randint(0, 100)
```

```
def bubble_sort_modified(arr, weights):
    i = 0
    while (i < n):
        j = 0
        while (j < n - i - 1):
            if (arr[j] * weights[j] > arr[j + 1] * weights[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                weights[j], weights[j + 1] = weights[j + 1], weights[j]
            j += 1
        i += 1
```

```
n, arr, weights = init()
print_array(arr)
print_average(arr)
read_weights(n, weights)
bubble_sort_modified(arr, weights)
print_array(arr)
```

Как сделать лучше?

1) Дали переменным осмысленные названия, из которых сильно проще понять что происходит.

2) Разделили код на функциональные части.

```
def print_array(arr):
    for i in range(0, n):
        print(str(i) + "->" + str(arr[i]))
```

```
def print_average(arr):
    average = 0
    for i in range(0, len(arr)):
        average += arr[i]
    average = average / n
    print(average)
```

```
def init():
    n = int(input())
    arr = [None] * n
    weights = [None] * n
    rand_array(arr, n)
    return n, arr, weights
```

```
def rand_array(arr, n):
    for i in range(0, n):
        arr[i] = randint(0, 100)
```

```
def read_weights(n, weights):
    for i in range(0, n):
        weights[i] = int(input())
```

```
def bubble_sort_modified(arr, weights):
    i = 0
    while (i < n):
        j = 0
        while (j < n - i - 1):
            if (arr[j] * weights[j] > arr[j + 1] * weights[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                weights[j], weights[j + 1] = weights[j + 1], weights[j]
            j += 1
        i += 1
```

```
n, arr, weights = init()
print_array(arr)
print_average(arr)
read_weights(n, weights)
bubble_sort_modified(arr, weights)
print_array(arr)
```

Хороший интерфейс

- Легко изучить.
- Легко использовать, даже без документации.
- Сложно использовать неправильно.
- Код, написанный с помощью этого интерфейса легко читать и поддерживать.
- Достаточная сила (количество возможностей).
- Легкая расширяемость.
- Подходит пользователям, которые будут его использовать.

Записи

Объединяют разнотипные данные (элементы, поля), относящиеся к одному объекту из предметной области.

Классы (именованные изменяемые элементы):

```
class Planet:
    pass

e = Planet()
e.name = "Earth"
e.radius = 6.371e6
e.mass = 5.9726e24
```

```
class Satellite:
    pass

m = Satellite()
m.name = "Moon"
m.planet = e
```

Кортежи (tuple) (индексируемые неизменяемые элементы)

```
student = ("Иван Петров", 21305)
```

Массивы

Последовательность однотипных данных.

Модуль **array** для примитивных типов:

```
import array
xs = array.array('i', [37, 42, 99])
```

Тип **list**:

```
ys = ["thirty seven", "42", "0x63"]
```

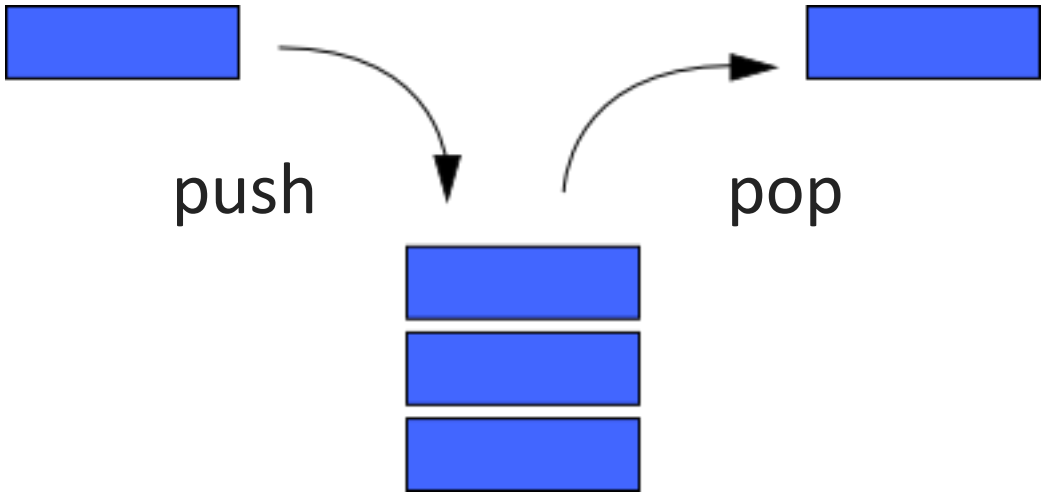
Как это работает?!

```
ys = ["thirty seven", "42", "0x63"]
print(ys) #["thirty seven", "42", "0x63"]
ys.append("kek")
print(ys.pop(0)) #"thirty seven"
print(ys) #["42", "0x63", "kek"]

print(e.__dict__)

#{'name': 'Earth', 'radius': 6371000.0, 'mass': 5.9726e+24}
```

Стек

- Абстрактный тип данных.
- Базовые операции:
  - Вставка элемента (push).
  - Извлечение элемента (pop).

Принцип LIFO (last-in, first-out).



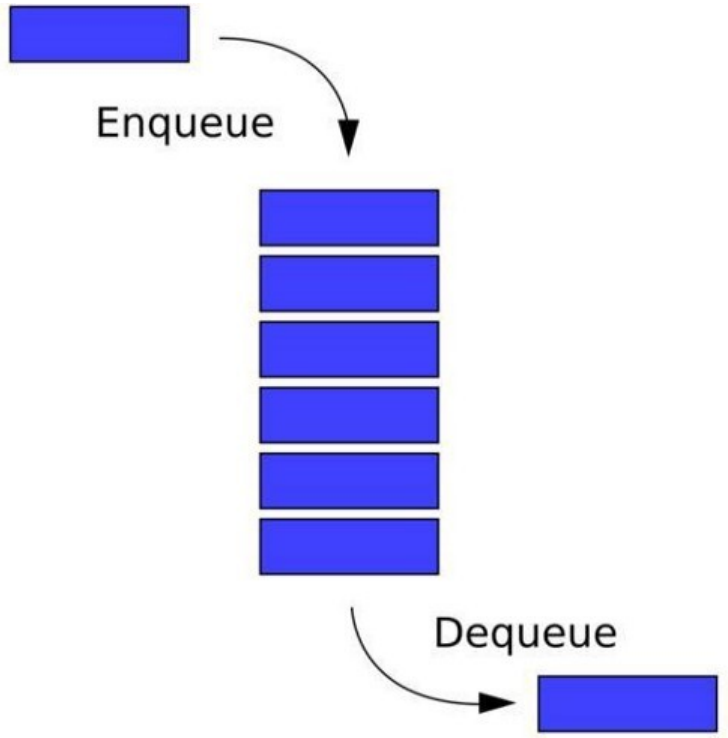
Применения стека

- Машинный стек.
- Вычисление выражений в обратной польской записи:
- (1+2*4+3) -> 1 2 4 * + 3 +
- Поиск пути в лабиринте.
- Задачи с иерархией элементов (обход дерева).

Очередь

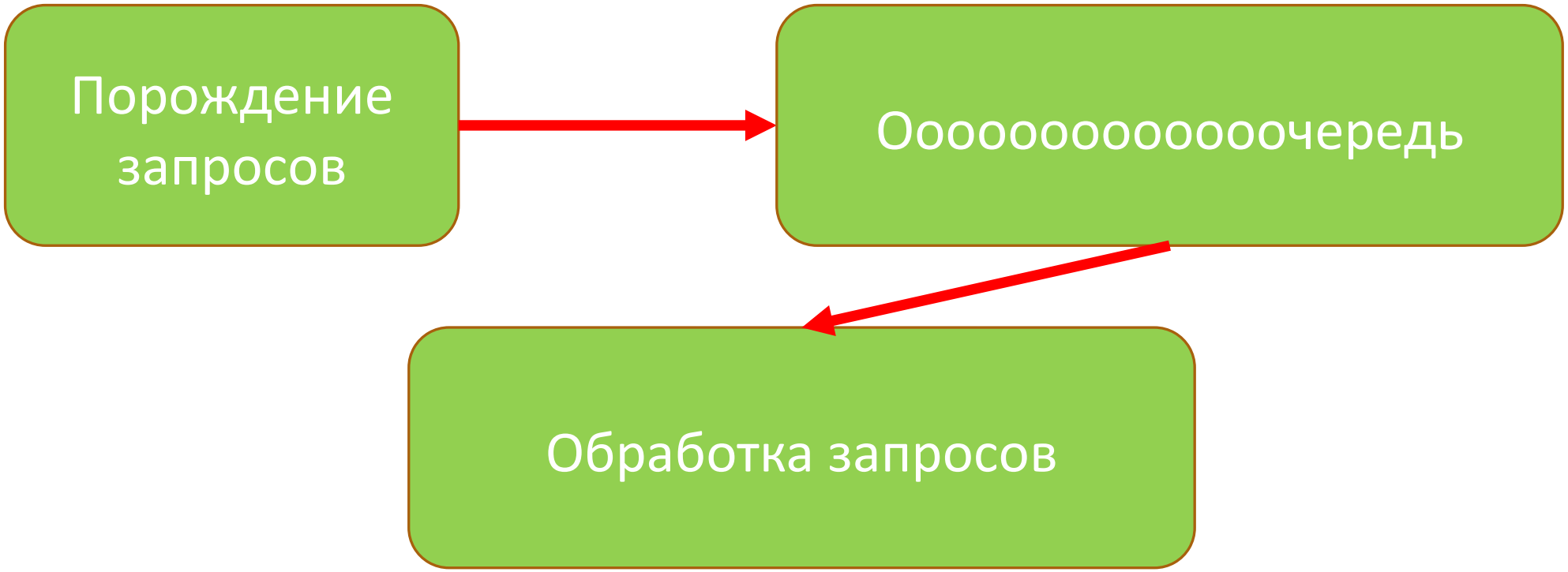
- Абстрактный тип данных.
- Базовые операции:
  - Вставка элемента (enqueue).
  - Извлечение элемента (dequeue).

Принцип FIFO (first-in, first-out).



Применение очередей

Обработка отложенных запросов.



Стек через статический массив

Статический массив достаточно большого размера.

```
stack = [None]*99999
sp = 0
stack[sp] = 10
sp += 1
stack[sp] = 20
sp += 1


sp -= 1
print(stack[sp])
```

Стек через динамический массив

- Динамически расширяющийся массив.
- Храним **N** элементов в массиве размера **M** (**N≤M**).

Стек через динамический массив

- Динамически расширяющийся массив.
- Храним **N** элементов в массиве размера **M** (**N≤M**).
- Если при добавлении **N** становится больше **M**, то увеличиваем массив до размера **2M** копируя содержимое.
- Если при удалении **N** становится меньше **M/4**, то уменьшаем массив до размера **M/2**, копируя содержимое.

Стек через динамический массив

- Временная сложность операции в стеке:

Стек через динамический массив

- Временная сложность операции в стеке:
- В лучшем случае потребуется просто изменить элемент массива **O(1)**.
- В худшем случае потребуется скопировать все **N** элементов и затем изменить один элемент: **O(N)**.

Анализ

Тип list в Python реализован как динамически расширяющийся массив.

- Вставка\удаление в конец
- Доступ по индексу
- Вставка\удаление в середину\начало

Анализ

Тип list в Python реализован как динамически расширяющийся массив.

- Вставка\удаление в конец – **O(1)**
- Доступ по индексу – **O(1)**
- Вставка\удаление в середину\начало – **O(N)**

list как стек – хорошо, а **list** как очередь – плохо.

На сегодня все