Объектно-ориентированное программирование

Object-oriented programming

X. Шаблоны проектирования

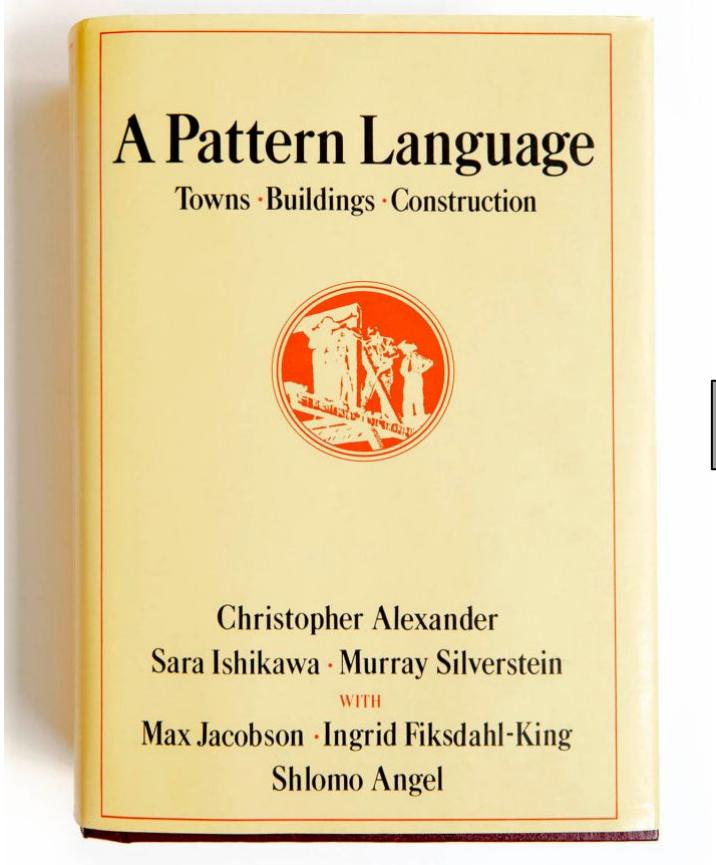
Design patterns

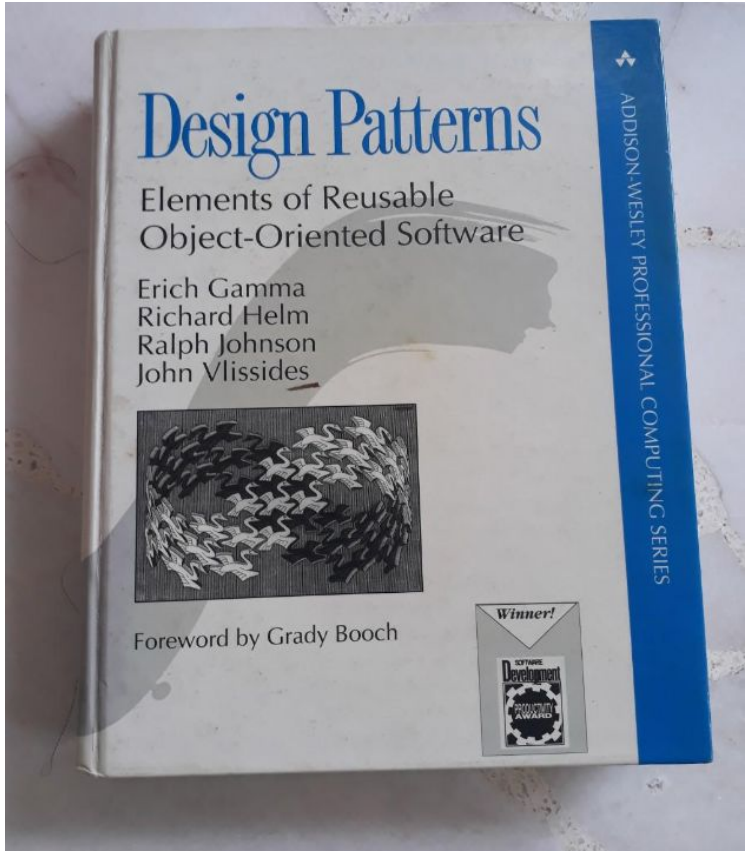
“Десять заповедей” ОО дизайна

1. Компоненты (классы, модули) должны быть открыты для расширения, но закрыты для изменений.
2. Потомки должны быть доступны посредством интерфейса базового класса без заметной разницы для клиента.
3. Реализации должны зависеть от абстракций, но не наоборот.
4. Критерий “повторного использования” – это версионность. Только компоненты, выпускаемые с контролем версий, могут быть эффективно **использованы многократно**.
5. Классы одного компонента должны иметь общее “замыкание”. Если один из классов необходимо изменить, изменять придется все.

1. Классы одного компонента следует **использовать** вместе. Невозможно отделить компоненты друг от друга, используя один из них. Используются всегда все.
2. Структура зависимостей должна быть ориентированным ациклическим графом.
3. Зависимости должны быть направлены в сторону наивысшей стабильности.
4. Чем стабильней компонент, тем больше абстрактных классов он должен содержать.
5. При переходе из одной парадигмы в другую в месте стыковки следует создавать абстрактный слой. Это необходимо для защиты обеих сторон от изменений с противоположной стороны.

https://groups.google.com/g/comp.object/c/WICPDcXAMG8?hl=en#adee7e5bd99ab111


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Winner!

"Each pattern describes a **problem** which occurs over and over again in our environment, and then describes the core of the **solution** to that problem, in such a way that you can **use this solution** a million times over, without ever doing it the same way twice."

C. Alexander

"A Pattern Language: Towns, buildings, construction"

https://refactoring.guru/ru/design-patterns

"A **design pattern** *systematically* names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the **problem**, the **solution**, **when** to apply the solution, and its **consequences**. It also gives implementation hints and examples. The **solution** is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context."

E. Gamma, J. Vlissides, R. Helm, R. Johnson

"Design Patterns: Elements of Reusable Object-Oriented Software"

Категории шаблонов

1. **Порождающие** (creational) – описывают **создание** объектов
2. **Структурные** (structural) – описывают **отношения** между объектами
3. **Поведенческие** (behavioral) – описывают **коммуникацию** между объектами

Из чего состоит шаблон?

1. **Имя** (краткое описание шаблона, например, в документации)
2. **Проблема** (причина – контекст – применения шаблона)
3. **Решение** (элементы, составляющие дизайн, отношения между ними, ответственность и взаимодействие – абстракция)
4. **Последствия** (результаты, положительные и отрицательные, от применения шаблона – время/память/ресурсы)

Model-View-Controller (MVC) в Smalltalk*

1. **Model** (application object) – описывает объект-приложение
2. **View** (screen presentation) – описывает вывод на экран
3. **Controller** (user interface) – описывают реакцию интерфейса приложения на ввод пользователя

* Используется для разделения логики приложения и вывода на экран. Часть **view** отвечает за правильность отображения текущего состояния. Часть **model** отвечает за правильность изменения состояния. Когда изменение происходит, model *оповещает* view об этом, используя специальный *протокол*.

Принципы OO дизайна

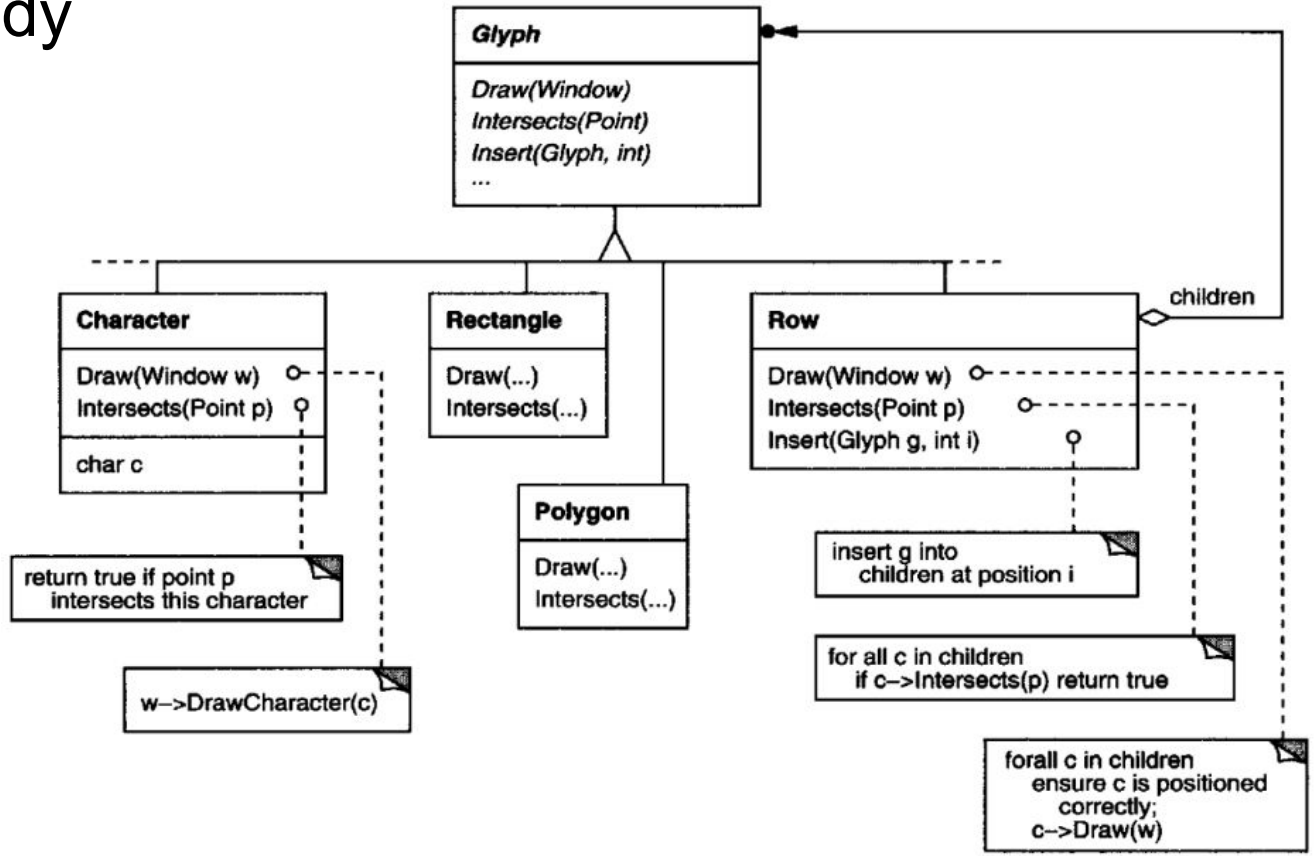
1. Program to an **interface**, not an implementation
2. Favor **object composition*** over class **inheritance**
3. Even though there are run-time inefficiencies, human inefficiencies are more important in the long run
4. Design for change (create objects indirectly, don't hard-code requests, limit platform dependencies, hide implementation details from clients, isolate unstable algorithms, use loose coupling, use composition and delegation)

* Черный ящик/белый ящик, динамическое/статическое изменение.

Toolsets, frameworks, design patterns

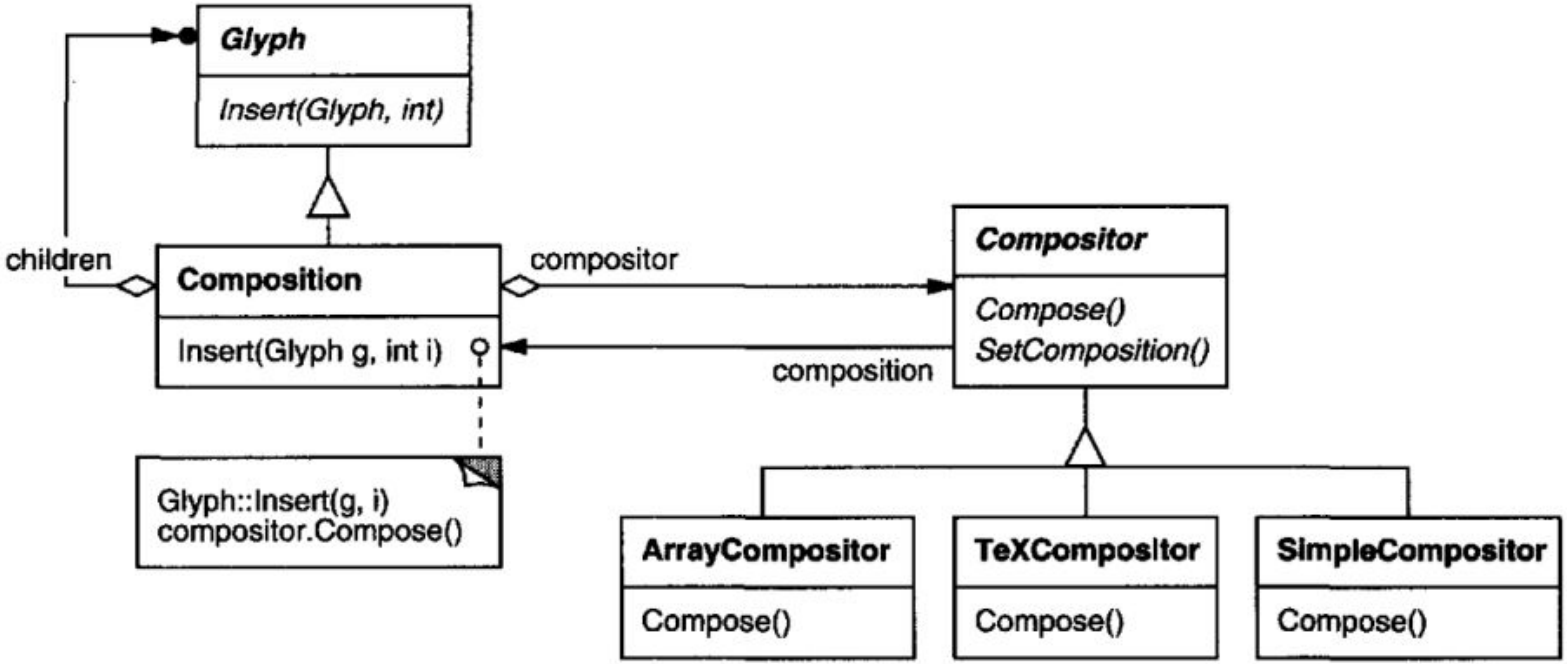
1. Design patterns are more abstract than frameworks (шаблоны не могут быть выражены в коде, только конкретные примеры их использования)
2. Design patterns are smaller architectural elements than frameworks (шаблоны часто содержатся в последних, но не наоборот)
3. Design patterns are less specialized than frameworks (шаблоны не направлены на решение задач в конкретной области)

Case study


Glyph
Draw(Window)
Intersects(Point)
Insert(Glyph, int)
...
children
Character
Draw(Window w)
Intersects(Point p)
char c
Rectangle
Draw(...)
Intersects(...)
Polygon
Draw(...)
Intersects(...)
Row
Draw(Window w)
Intersects(Point p)
Insert(Glyph g, int i)
return true if point p
intersects this character
w->DrawCharacter(c)
insert g into
children at position i
for all c in children
if c->Intersects(p) return true
forall c in children
ensure c is positioned
correctly;
c->Draw(w)

“Стратегия” (strategy)

Инкапсулирует в себе какой-то **алгоритм**



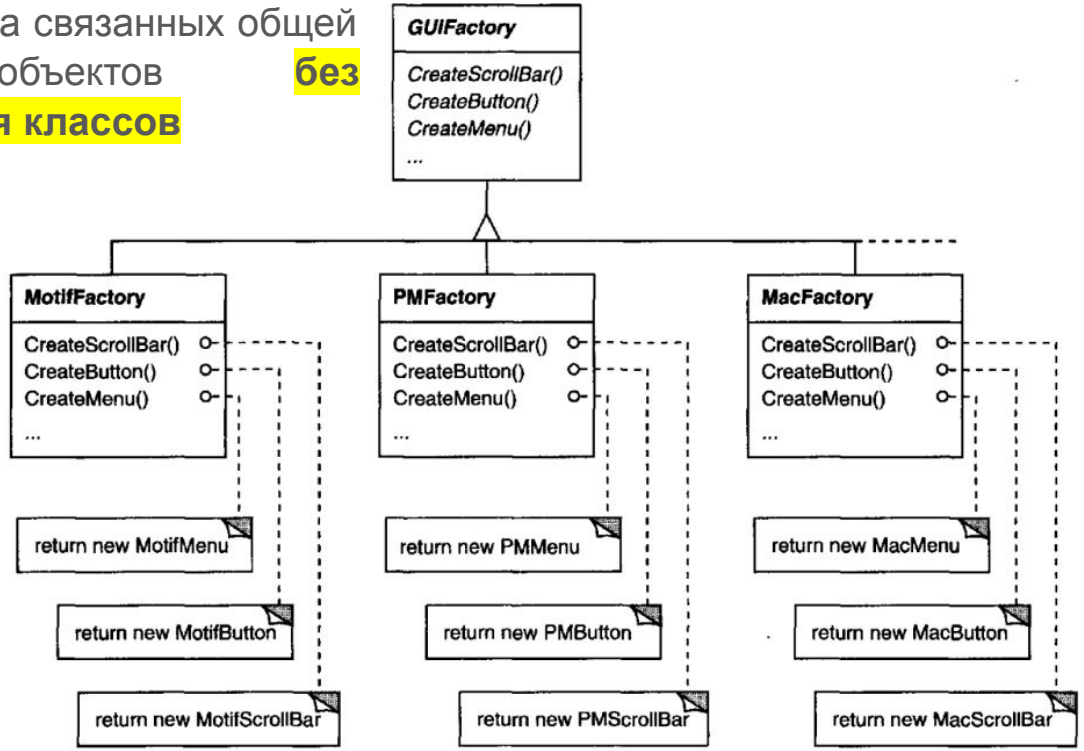
Пример “стратегии”

std::sort

```
template <typename Iter, typename Comp = less> void sort(Iter a, Iter b, Comp comp) {
    auto size = distance(a, b);
    if (size < 2) return;
    if (size < 6) {
        ...
        if (comp(*a, *b)) {
            ...
            swap(*a, *b);
            return;
        }
    }
    heapsort(a, b);
}
```

“Абстрактная фабрика” (abstract factory)

Создает семейства связанных общей структурой объектов **без инстанцирования классов**



“Одиночка” (singleton)

Гарантирует, что класс можно **инстанцировать только один раз**, и предоставляет глобальный доступ к своему объекту

```
class Settings {
    static Settings &instance;
    Settings( ) { }
    ...
public:
    static Settings& getInstance( ) {
        if (!&instance) {
            instance = *(new Settings);
        }
        return instance;
    }
    const char *mode = "dark";
}
```

“Прототип” (prototype)

Указывает на тип объектов, которые необходимо создать, используя готовый экземпляр, копируя его при необходимости.

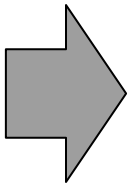
```
const zombie = {
    eatBrains() {
        return 'munch';
    }
}

const clone = Object.create(zombie, { name: { value: 'Zombie' } })
// не реализует eatBrains но может его вызывать в цепочке прототипа

Object.getPrototypeOf(clone)
```

https://fireship.io/lessons/typescript-design-patterns/

“Строитель” (builder)

Отделяет конструирование сложносоставного объекта от его представления, чтобы один и тот же процесс конструирования мог создавать разные представления.

```
class HotDog {
    char *bun;
    bool ketchup;
    bool mustard;
public:
    HotDog(
        const char *bun, bool ketchup, bool mustard);
    HotDog &addKetchup( ) {
        this.ketchup = true;
        return this;
    }
    HotDog &addMustard( ) {
        this.mustard = true;
        return this;
    }
}
```

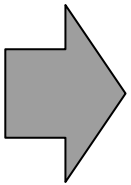


```
breakfast = HotDog(“wheat”, true, true);

lunch = HotDog(“Gluten-free”);

lunch.addKetchup().addMustard();
```

“Фабричный метод” (factory method)

Виртуальный конструктор – определяет интерфейс для создания объекта, но позволяет подклассам выбирать класс для инстанцирования.

```
class Button { … };
class IOSButton : public Button { … };
class AndroidButton : public Button { … };
class ButtonFactory {
public:
    Button &createButton(const char *os) {
        if ("ios" == os) {
            return *( new IOSButton( ) );
        } else {
            return *( new AndroidButton( ) );
        }
    }
}
```



```
auto factory = ButtonFactory();

auto button1 =
factory.createButton(“ios”);
```

Ну или чтобы не мучать птичку

```
auto button1 = (os == “ios”) ? IOSButton() : AndroidButton();
```

“Заместитель” (proxy)

Предоставляет подмену другого объекта для контроля доступа к нему.

```
const original = { name: 'Alice' }

const reactive = new Proxy(original, {
    get(target, key) {
        return target[key];
    },
    set(target, key, value) {
        return Reflect.set(target, key, value);
    },
});

reactive.name = 'Bob';
```

“Наблюдатель” (observer)

Определяет зависимость вида “**один-ко-многим**” между объектами для того, чтобы при изменении состояния одного объекта все зависимые объекты об этом оповещались.

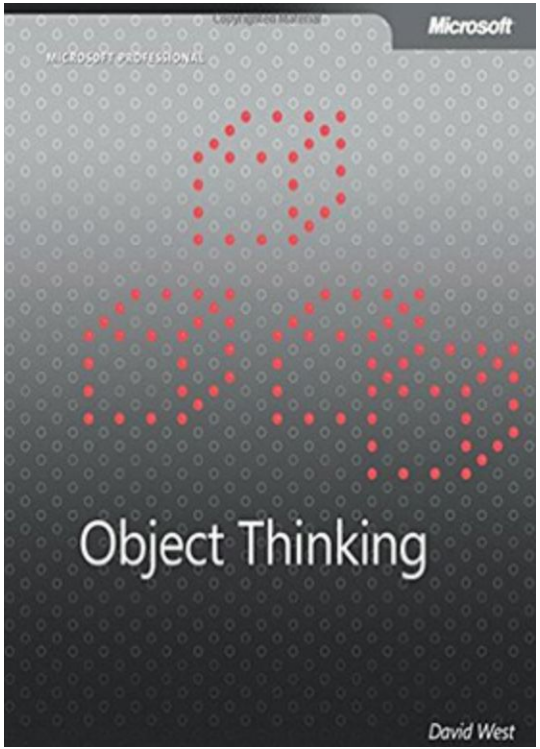
```
class Subject {
protected:
    Subject();
private:
    list<Observer*> observers;
public:
    virtual ~Subject();
    virtual void Attach(Observer &);
    virtual void Detach(Observer &);
    virtual void Notify() {
        list<Observer*>::iterator it( observers.begin() );
        for (; it != observers.end(); it.next()) {
            it.item()->Update(this);
        }
    }
};
```

“Посредник” (mediator)

Определяет объект (“брокер”), который инкапсулирует правила взаимодействия группы объектов. Эти объекты не могут обращаться друг-к-другу напрямую, чтобы между ними не было зависимостей.

```
import express from 'express';

const app = express();

function logger(req, res, next) {
    …
    next()
}

app.use(logger);
app.get('/', (req, res) => {
    res.send('Hello!);
});
app.get('/about', (req, res) => {
    res.send('Goodbye!');
});
```

D. West, “Object Thinking”



Yegor Bugayenko

★★★★★ **I'd like to give six stars to this book**

Reviewed in the United Kingdom on February 2, 2013

Verified Purchase

Great book about object oriented design, programming, analysis, and thinking. The book doesn't contain primers of Java code or any other practical examples. Instead, it gives a high-level overview of what is an object, how it differs from a data structure, how to think like an object, and why object thinking is a more effective approach than a procedural one. The book has a lot of references to other books, researches, ideas, and ideals. It is more a philosophical prophecy than a programmers guide.

rifraf

★☆☆☆☆ **I tried but...**

Reviewed in the United Kingdom on October 17, 2015

Verified Purchase

Recommended by Avdi Grimm, but sorry I didn't get it. Still no idea what Object Thinking is. Maybe I am spoiled by reading Uncle Bob - a great communicator.

http://davewest.us/product/object-thinking/

"Object-oriented programming aficionados think that everything is an object.... this [isn't] so. There are things that are objects. Things that have *state* and *change their state* are **objects**. And then there are things that are not objects. A binary search is not an object. It is an **algorithm**."

A. Stepanov

leon.brinzan@iis.utm.md