Back To Basics
Object-Oriented Programming

AMIR KIRSH

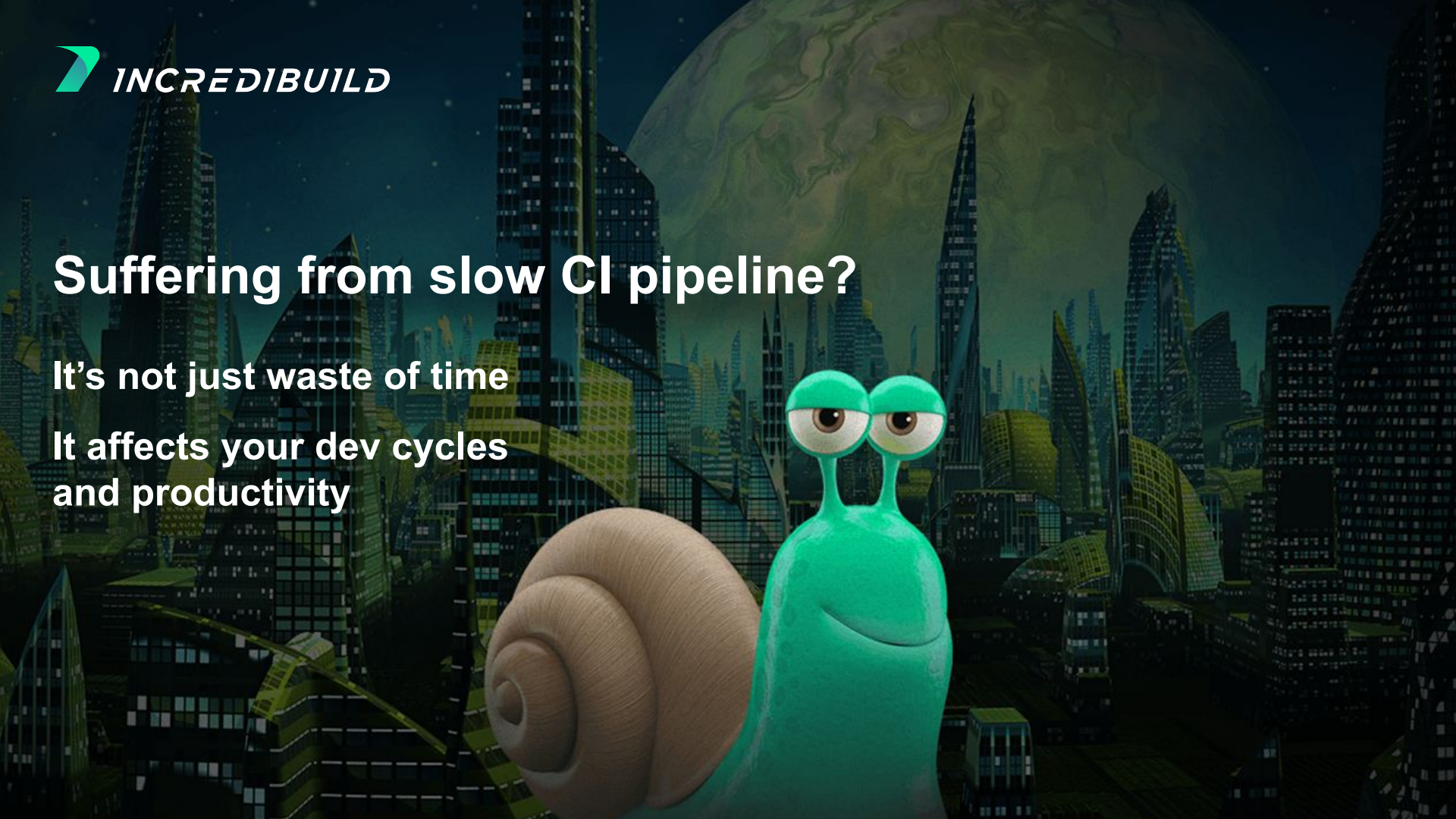
About me

Lecturer
Academic College of Tel-Aviv-Yaffo
and Tel-Aviv University
Visiting researcher at SBU, NY

Developer Advocate at

INCREDIBUILD

Co-Organizer of the **CoreCpp** conference and meetup group


INCREDIBUILD
Suffering from slow CI pipeline?
It’s not just waste of time
It affects your dev cycles
and productivity

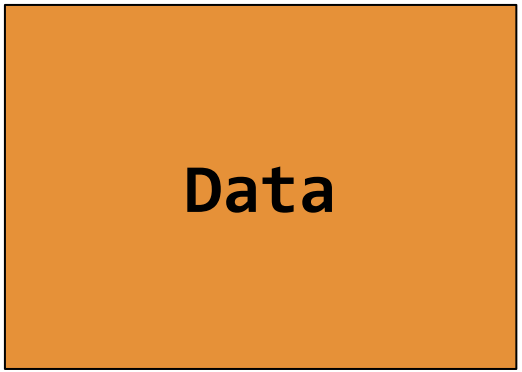
Goals

- Discuss the basics of Object Oriented Programming in C++
- Understand the alternatives and tradeoffs

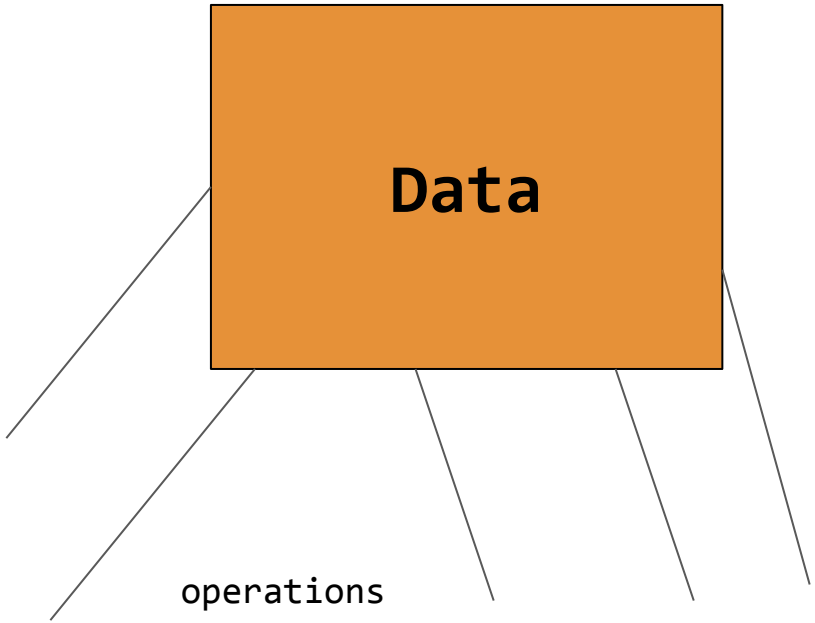
Part 1

Object Oriented Programming

Object Oriented Programming



Object Oriented Programming



Classes and Objects

Class = the code that describes an entity

Object = the actual instance of a class

```
class Widget { … }; // describes widget, nothing born yet

int main() {
    Widget w; // an actual object is created
}
```

Stick to what you do

Very misanthropic, but important:

Every class takes care of its own business

Single Responsibility

A class should only have a single responsibility

we all know such classes,
try to avoid it

A crash syntax course

class Point

```
class Point {
    int x, y;
public:
    Point(int x1 = 0, int y1 = 0): x(x1), y(y1) {}
    void set(int x1, int y1) {
        x = x1;
        y = y1;
    }
    void move(int diffX, int diffY);
    void print() const { std::cout << "x = " << x << ", y = " << y; }
};
```

don’t forget the semicolon!

class Point - usage

```
int main() {

    Point p1;
    p1.set(3, 7);
    p1.move(2, 2);
    p1.print();

    const Point p2(10, 5);
    // p2.set(10, 5);
    // p2.move(2, 2);
    p2.print();
}
```

Privileges (“access modifiers”)

public

accessed by anyone (with the proper context / caller)

protected

accessed by the class itself and derived classes

private

accessed by the class itself only
(Note: you can access privates of other objects of same class - the privilege is on the class not on the object)

Privileges - class and struct

class and struct are the same in C++ except for two differences:

[1]

- the default privilege in **class** is **private**
- the default privilege in **struct** is **public**

[2]

- the default inheritance mode in **class** is **private**
- the default inheritance mode in **struct** is **public**

  (related to inheritance which would be discussed later…)

Data members

The data that the class manages

- each object has its own copy of the data members
- usually (*almost always!*) - should be private
- there is no default initialization for primitive types if not initialized

Member functions (= “methods”)

The operations that can be performed on an object of this type

- might be public, protected or private (yes, should be private in some cases!)
- are called with an object (“the calling object”, “the caller”)
- can access the data members - of the calling object
- is not part of the object size

Object size

Object size

- includes the size of all its members
- doesn’t include functions
- in case of inheritance: includes the size of its parent(s)
- may include additional parts, e.g. pointer to vtable (discussed in another lesson)
- may include padding (see cppreference)

header and cpp

```
// .h file
class Point {
    int x, y;
public:
    void set(int, int); // declaration only
    void print() const { std::cout << "x = " << x << ", y = " << y; }
};

// .cpp file
#include "Point.h"
void Point::set(int x1, int y1) {
    x = x1;
    y = y1;
}
```

this

this

is a pointer to the calling object

```
struct A {
    void printAddress() { std::cout << (void*)this << std::endl; }
};

int main() {
    A a;
    std::cout << (void*)(&a) << std::endl;
    a.printAddress();
}
```

http://coliru.stacked-crooked.com/a/115ddc47da51892f

Constructors

Rules:

- **No constructors at all = there is empty ctor by default**
- **No empty constructor = must pass parameters!**
- **Can overload constructors**
- **C++11: Can call another constructor (“ctor delegation”)**
- **Can use default parameters - as any other method in C++**
  (a single ctor can get parameters and still be empty ctor)
- **Ctor Init list - as seen in first example**
  used for initialization of members as well as base class(es)

Constructors: init list

```
class Point {
    int x, y;
public:
    Point(int x1, int y1): x(x1), y(y1) {}
    void print() const { std::cout << "x = " << x << ", y = " << y; }
};
```

Init list

```
class Rectangle {
    Point TL, BR;
public:
    Rectangle(const Point& tl, const Point& br): TL(tl), BR(br) {}
    void print() const {
        std::cout << "TL: "; TL.print();
        std::cout << ", BR: "; BR.print();
    }
};
```

Init list

http://coliru.stacked-crooked.com/a/837957d32f3bd6f8

Ctor init list

Use cases: (a) efficiency (b) correctness (c) in some cases you MUST

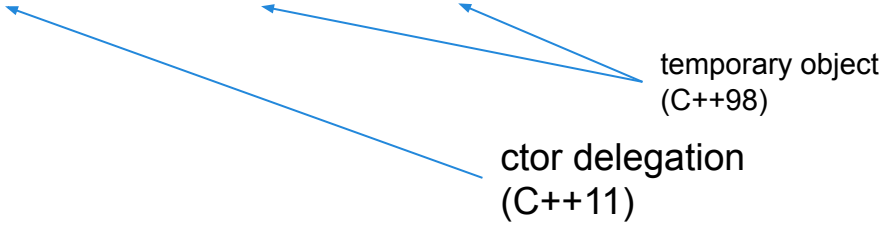
MUST:

1. **contained object with no default ctor and no initialization on declaration**
2. **contained const data member**
3. **contained reference data member**
4. **base class with no default ctor**

example for 2 and 3: http://coliru.stacked-crooked.com/a/697ef6e8d3a763ef

Constructor delegation (C++11)

```
class Rectangle {
    Point TL, BR;
public:
    Rectangle(const Point& tl, const Point& br): TL(tl), BR(br) {}
    Rectangle(int x1, int y1, int x2, int y2)
                    : Rectangle(Point(x1, y1), Point(x2, y2)) {}
};
```



* C++11 also added ctor inheritance

Copy C’tor

- **Signature:**

```
A::A(const A& a);
```

- **Used when creating a copy**
- **Called automatically when passing objects of this class By Value**
- **If you don’t implement your own - you get a default one by the compiler, which does memberwise-copy**

Copy C’tor

- **Signature:**

```
A::A(const A& a);
```

- **Used when creating a copy**
- **Called automatically when passing objects of this class By Value**
- **If you don’t implement your own - you get a default one by the compiler, which does memberwise-copy**

Copy C’tor

What happens if this is my copy c’tor signature:

```
A::A(A a);
```

Assignment Operator

- **Signature:**

```
A& A::operator=(const A& a);
```

- **Used when assigning an object of same type**
- **Don’t confuse with Copy C’tor! They are very similar but not the same**
- **If you don’t implement your own - you get a default one by the compiler, which does memberwise-copy**

Assignment Operator

- **Signature:**

```
A& A::operator=(const A& a);
```

- **Used when assigning an object of same type**
- **Don’t confuse with Copy C’tor! They are very similar but not the same**
- **If you don’t implement your own - you get a default one by the compiler, which does memberwise-assignment**

Assignment Operator

Can we implement assignment as aglobal function:

```
A& operator=(A& a1, const A& a2);
```

Assignment Operator

Can we get by value?

```
A& A::operator=(A a);
```

C’tor used for Casting

```
class A {
    int i;
public:
    A(int i1):i(i1){}
};

void f(const A& a);

// implicit casting works
// only for ‘const ref’
// or for byval
// but not for byref
```

```
int main() {
    A a1(1);
    A a2 = 2;
    f(A(1)); // works
    f((A)1); // works
    f(1);    // works!
    a1 = 3;  // works!
}
```

explicit

```
class A {
    int i;
public:
    explicit A(int i1):i(i1){}
};

void f(const A& a);
```

```
int main() {
    A a1(1);      // ok
    // A a2=2;    // can't...
    f(A(1));      // ok
    f((A)1);      // ok
    // f(1);      // can't...
    // a1 = 3;    // can't...
    a1 = A(3);    // ok
}
```

const + mutable members

```
class Array {
    int arr[SIZE]{};
    mutable int sum = 0;
    mutable bool isSumUpdated = true;
    void calcSum()const;
public:
    Array() {}
    // …
```

Destructor

Called automatically when object dies*

- **Takes no arguments, thus there is only one per class:**
  `~<ClassName>(); // e.g. for class A: ~A();`
- **Guaranteed to be called immediately when object dies**
  (if process is not terminated)
- **Usually used for resource de-allocations (but can actually do anything)**

*** When object dies?**

- **Stack object - at the matching closing curly brackets / end of block**
- **Heap object allocated with 'new' - when deleted with 'delete'**
- **Global or Static object - at the end of the process**
- **Temporary object - by the end of the statement**

```
message("hello", Point(10,10));
```

Rule of Zero

It is the best if your class doesn't need any resource management

- no need for dtor, copy ctor, assignment operator
- defaults do the job
- that includes defaults for move operations

To achieve that, use properly managed data members: std::string, std containers, std::unique_ptr, std::shared_ptr

Image Source:
https://www.fluentcpp.com/2019/04/23/the-rule-of-zero-zero-constructor-zero-calorie/

Rule of Three

If you need a destructor, first thing block the copy ctor and assignment operator

- No TODO, no let's check if we need to implement them, BLOCK NOW

```
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

- If you need them later => implement

Rule of Five

If you implement or block any one of the five, you lose the defaults for the move operations

- Make sure to ask back for the defaults if they are fine

```
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

We will talk later on RValue reference and Move semantics ^ ...

Inheritance

Inheritance - why?

Code reuse:

1. **We have a class that we like and we want to add functionality or change its behavior, without touching the original code**

2. **We want to use both the ‘old’ class and the ‘new’ class - so we can’t change the code of the old one**

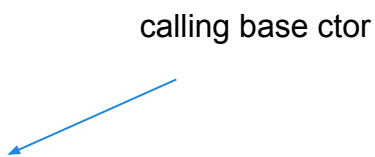
Polymorphism:

We want to hold and manage objects of either type without having to handle them differently (e.g. Person and Student, Dog and Cat)

Inheritance - ctor

```
class Person {
// ...
public:
    Person(const string& name);
// ...
};

class Student: public Person {
// ...
public:
    Student(const string& name): Person(name){}
// ...
};
```



Inheritance - dtor

```
struct A {
    ~A() { cout << "~A" << endl; }
};

// B is inherited from A for non-polymorphic usage
struct B: public A {
    ~B() { cout << "~B" << endl; }
};

int main() {
    B b;
}
```

would print:
~B
~A

Polymorphism

Polymorphism is the ability to treat different types similarly:

```
class Pet {
public:
    virtual void eat(const Food& food) = 0;
    // …
};
```

```
pet.eat(food);
```

any type of Pet

any (proper) type of Food

virtual functions

```
class Pet {
    //...
public:
    virtual void makeSound() const = 0;
    virtual ~Pet() {}
};
```

Note: if makeSound is const, it must be const in all the classes to preserve the same signature

```
class Dog: public Pet {
    //...
public:
    void makeSound() const override {
        cout << "Raf raf";
    }
    ~Dog() override {}
};
```

```
class Cat: public Pet {
    //...
public:
    void makeSound() const override {
        cout << "mewo";
    }
    ~Cat() override {}
};
```

abstract classes

```
class Pet {
    //...
public:
    virtual void makeSound() const = 0;
    virtual ~Pet() {}
};
```

Note: makeSound method is pure virtual at Pet, which makes Pet an abstract class

```
class Dog: public Pet {
    //...
public:
    void makeSound() const override {
        cout << "Raf raf";
    }
    ~Dog() override {}
};
```

Can't create an object of type Pet

```
int main() {
    // Pet pet;
    Dog d;
    Pet* p = &d
    p->makeSound();
}
```

Usage Example - Command Pattern

- Encapsulate the information needed to perform an action.
- Classical for implementing Undo/Redo stack.

Advantages

- Encapsulates and hides the action itself, easier to code and maintain.

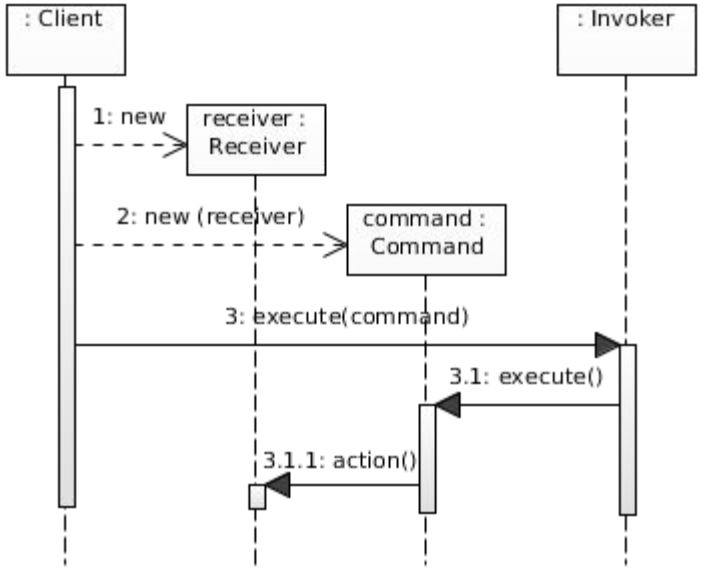


Image source:
https://javaobsession.wordpress.com/2010/07/25/command-pattern/

https://en.wikipedia.org/wiki/Command_pattern

OO Low-Level Design Principles

- A class shall represent a single thing
- Break a complicated entity into several smaller classes
- Use composition and inheritance properly
- Keep abstraction - implement your code for a “generic” interface

Also

- Hide your privates: data members and member functions
- Keep clear and simple API
- Try to keep your classes under the rule of zero

Part 2

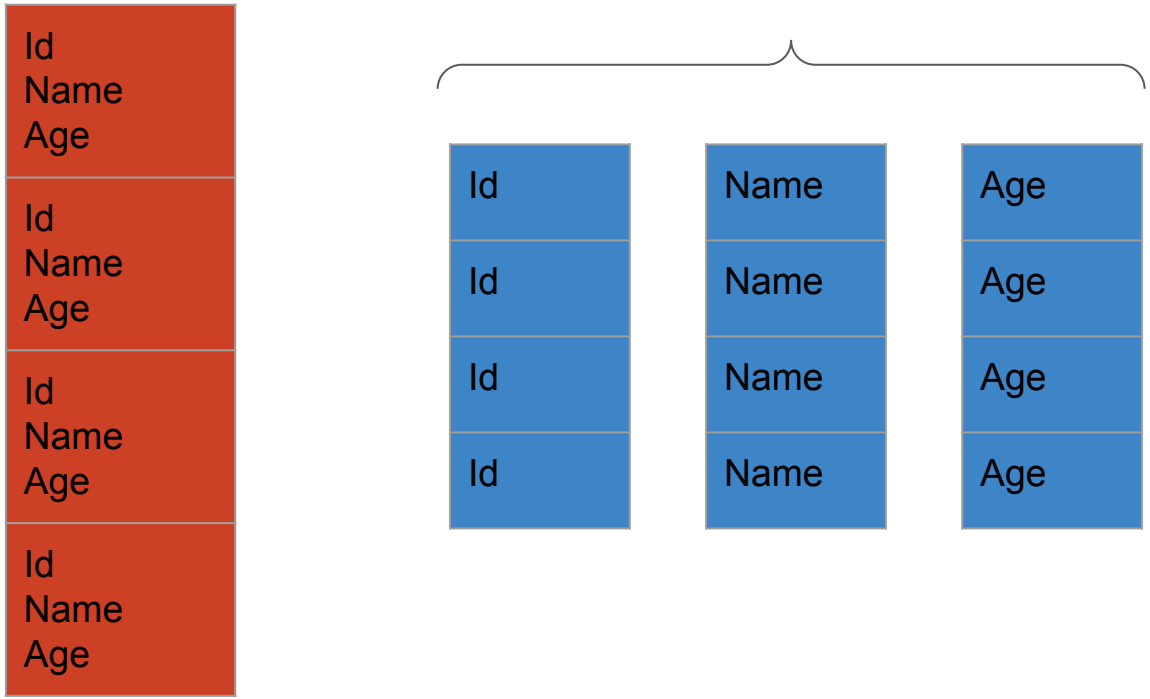
Beyond the “Classic” Model

C++ is not *Just* an Object Oriented Language (Bjarne Stroustrup)

Let’s discuss:

- When and way not to use the classic encapsulation
- When to avoid or delay inheritance

Array of Structs vs. Structs of Arrays

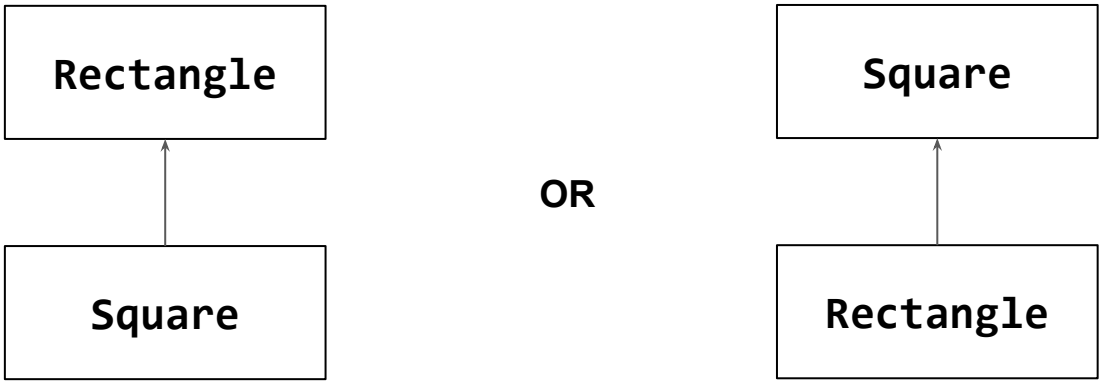


Inheritance

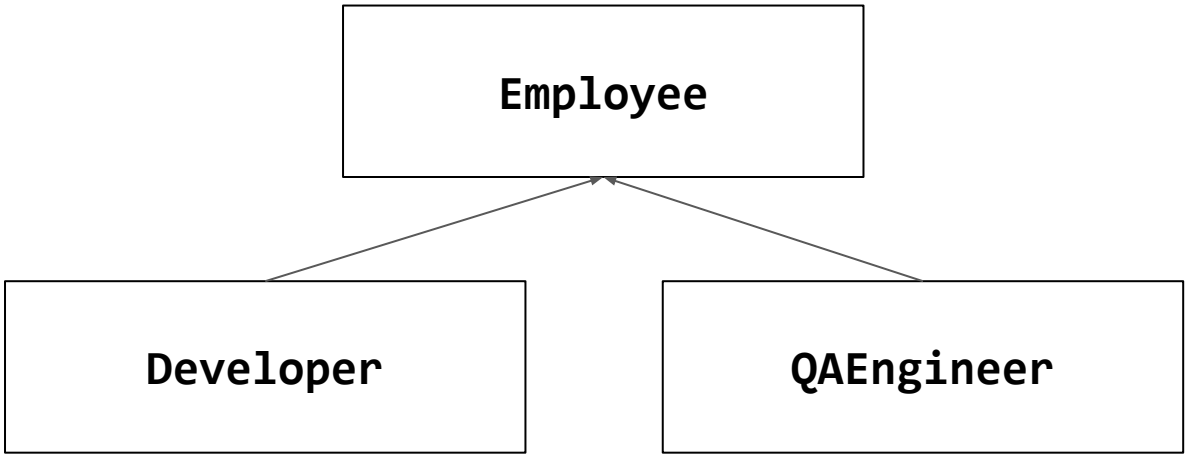
Inheritance is overrated
In some cases it’s tricky

Sean Parent, 2013:
Inheritance Is The Base Class of Evil

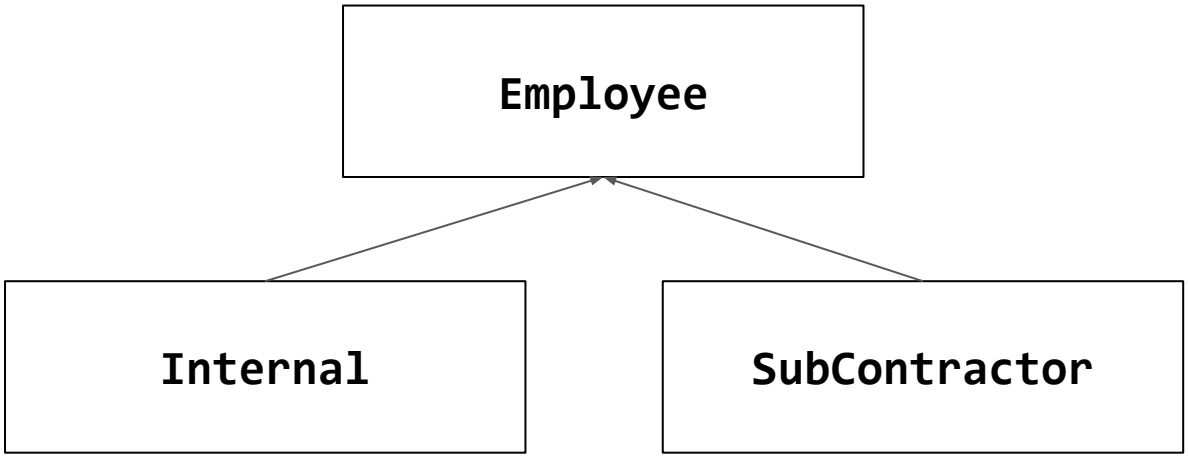
Inheritance and Liskov Substitution



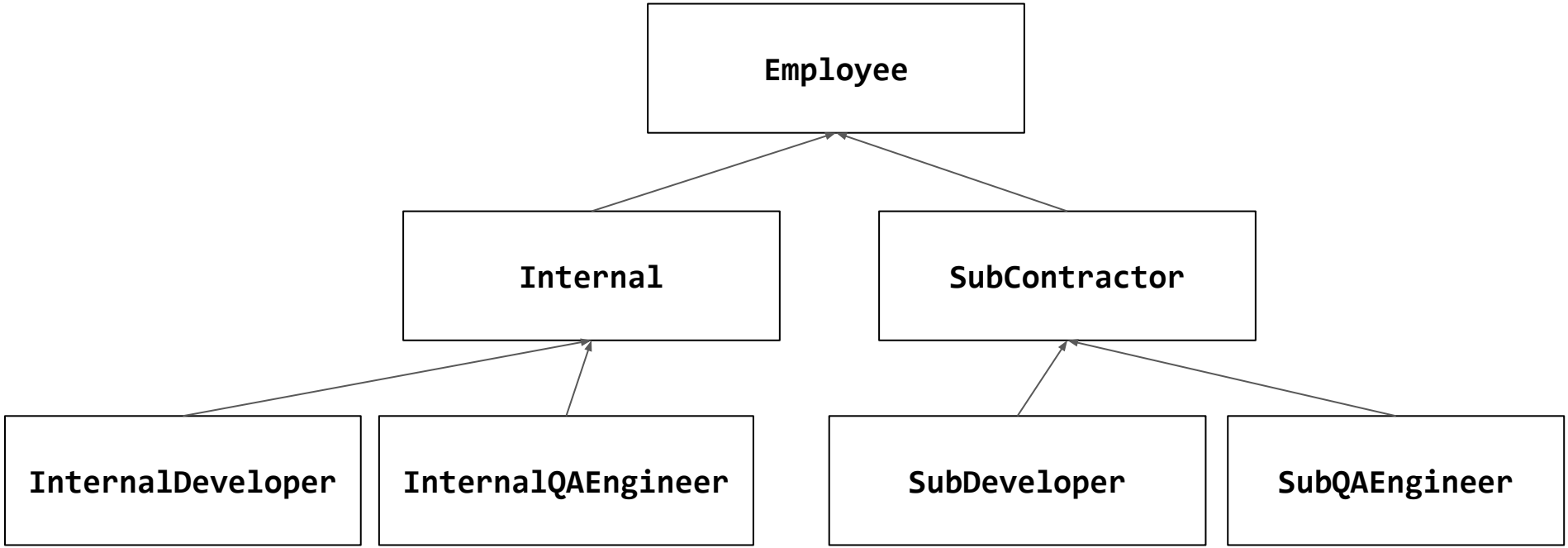
Employee Inheritance (?)



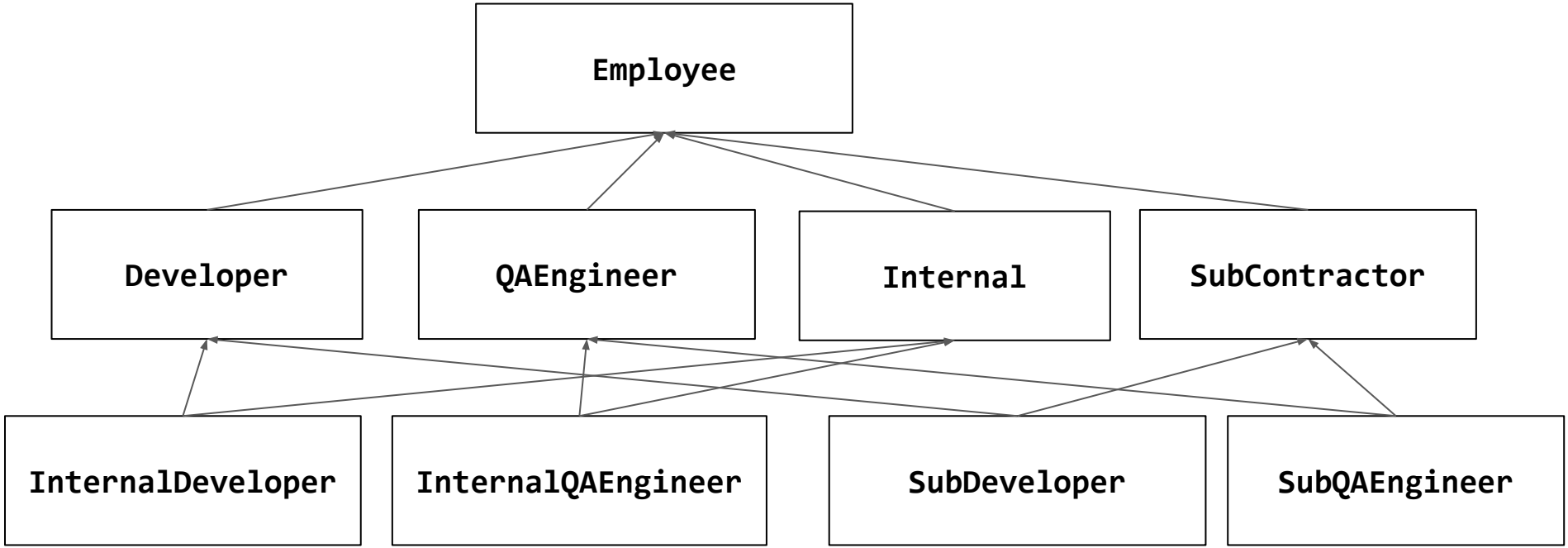
Employee Inheritance (?)



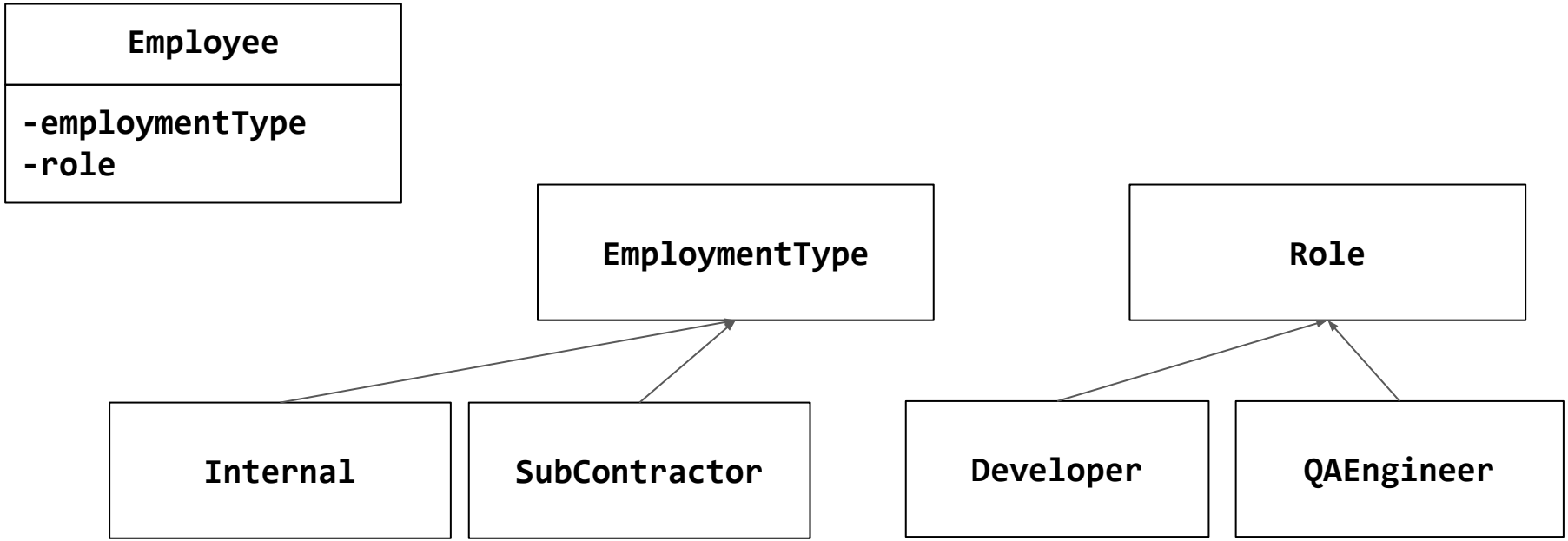
Employee Inheritance (?)



Employee Inheritance (?)

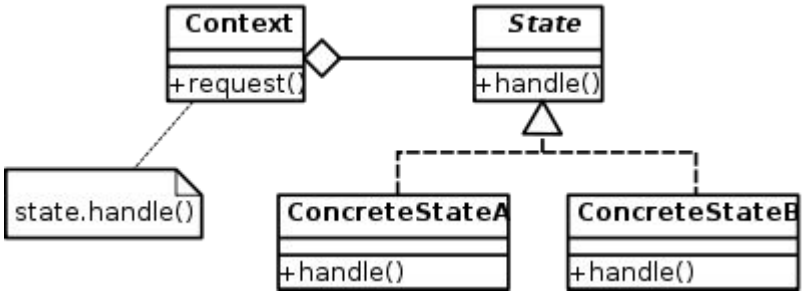


Employee Inheritance (!)



State Pattern

- Encapsulate varying behavior based on object's state.
- Allowing combination of behaviors per characteristic, with specific State hierarchy per each.
- Decoupling state from Object Structure.



https://en.wikipedia.org/wiki/State_pattern

Advantages

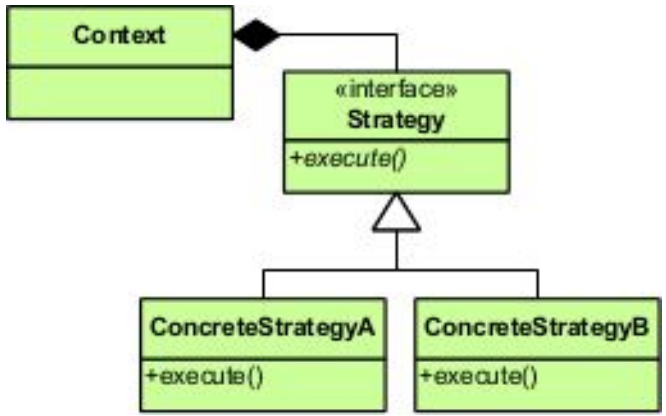
- Allowing objects to dynamically change state.
- Allowing objects to have more than one state.

Strategy Pattern

- Select algorithm (strategy) to be used at runtime.
- Defines a family of possible algorithms for same problem.

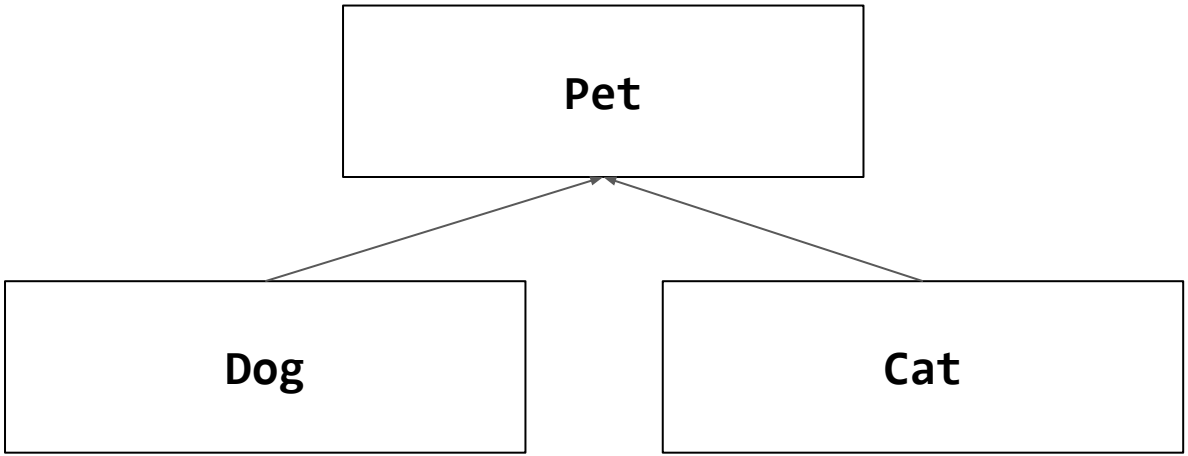
Advantages

- Can be used to pick the matching / best algorithm according to defined rules.
- Algorithm selection is encapsulated and can be cached

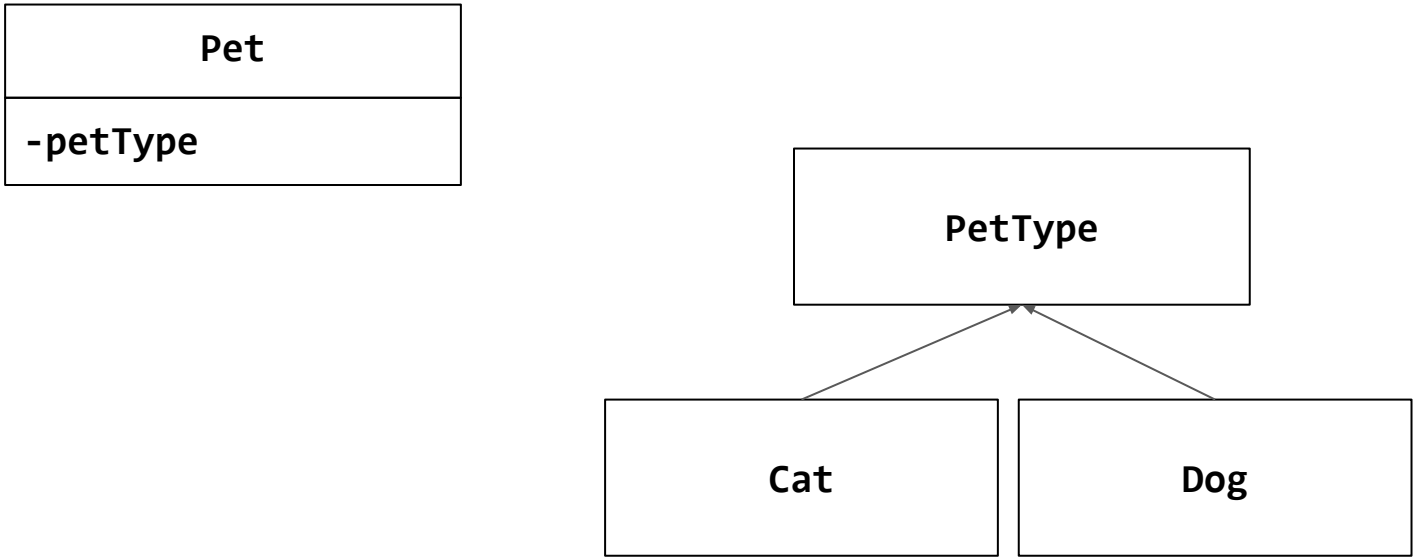


https://en.wikipedia.org/wiki/Strategy_pattern

Pet Inheritance (?)



Pet Inheritance (!)



Issues with Inheritance

- **Changing type in runtime (was a QAEngineer now a Developer)**
- **Inflation in derived classes**
  (need to think of ways to reduce number of classes!)

Solution: State or Strategy Patterns

- **Forces the user to be in the details of our internal design**
  (which exact type to create)

Solution: Factory Method / Abstract Factory Patterns

Inheritance Design Principles

Scott Meyers:

Make non-leaf classes abstract

Herb Sutter:

- *Don’t derive from concrete classes*
- *Make virtual functions private*

Inheritance Design Principles

I say:

- *User should work with a universal type - **same type represents all***
  - keep your inheritance for internal State/Strategy
- *Prefer to have **stateless** abstract classes (“pure interfaces”)*
- *If a base class does manage data, **keep it very-small and specific***

Polymorphism vs. Templates

Polymorphism vs. Templates

Implement a generic ‘volume’ function for any prism

- based on polymorphism
- based on templates

Solutions:

- Polymorphism
- Templates

Discuss: when and why

Substitutes for Inheritance (or how to delay it)

1. **Avoiding inheritance: using templates, composition, lambdas or just simple “duck type” with generic algorithms**

   list::iterator and vector::iterator do not (necessarily) share a base!
   (as a side note => may use C++20 concepts to set expectations on type)

2. **Inheritance of smaller things (state / strategy)**
   - for properties, behavior, policy

3. **Hiding your inheritance with a facade / Proxy of a one clear type**
   - user should preferably work with one universal type

To Summarize

To Summarize

Object Oriented Programming is good - this is why it's so widely used

But use it with care:

- **Different problems may need different tools**
- **Think of things that may change: additional future classes and usages**

Complex Code

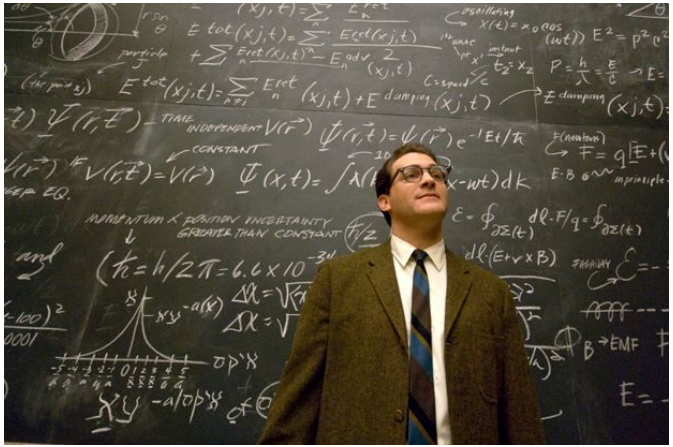


Image Source: http://iscoweb.iut.ac.ir/en/content/adjunct-professor

What makes code unnecessarily too complex?

- **classes that do more than one thing**
- methods that do more than one thing,
  or **do not use helper methods**
- **too much abstraction**
  (an interface for the interface)
- **exposing your internal design** (forcing the user to know too much, allowing abuse)

And in general: bad design :/

OO Low-Level Design Principles

- A class shall represent a single thing
- Break a complicated entity into several smaller classes
- Use composition and inheritance properly
- Keep abstraction - implement your code for a “generic” interface

Also

- Hide your privates: data members and member functions
  (design decisions such as inheritance can also be hidden in a universal holder)
- Keep clear and simple API
- Try to keep your classes under the rule of zero

Any questions before we conclude?

Photo by Howie R on Unsplash

Amir Kirsh
amir.kirsh@incredibuild.com
kirshamir@gmail.com