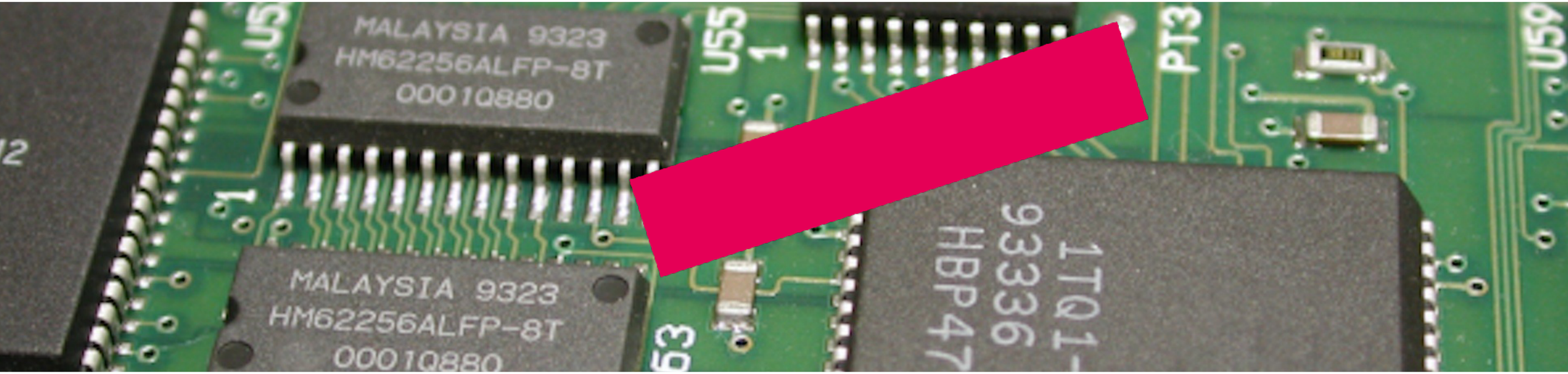
Prog 5 - 4. Interfaces and abstract classes



Electrical Engineering / Embedded Systems
Faculty of Engineering

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

To begin with…

Attendance

Schedule (exact info see #OO and roster at insite.han.nl)

		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	Class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	Liskov Substitution Principle	lists, inline functions, default params	Activities
Step 4	Interface Segregation Principle	**interfaces and abstract classes**	Depencencies
Step 5	Dependency Inversion Principle	threads, callback	Sequence diagrams
Step 6	Coupling and cohesion	polymorphism	
Step 7	n/a	n/a	n/a

Note: subject to changes as we go…

Interfaces / Abstract classes. To start with: some terms

A **virtual function in C++** is a member function in a class that we declare within the base class and redefine in a derived class.

A **pure virtual function in C++** is nothing but a virtual function that we know exists but cannot be implemented. It is simply declared, not implemented.

Implementing a **pure virtual function in C++**

/* 0 does not indicate that we initialize the function to a null or zero value */

```
virtual void function() = 0;
```

https://data-flair.training/blogs/interfaces-in-cpp/

Abstract classes and Interfaces

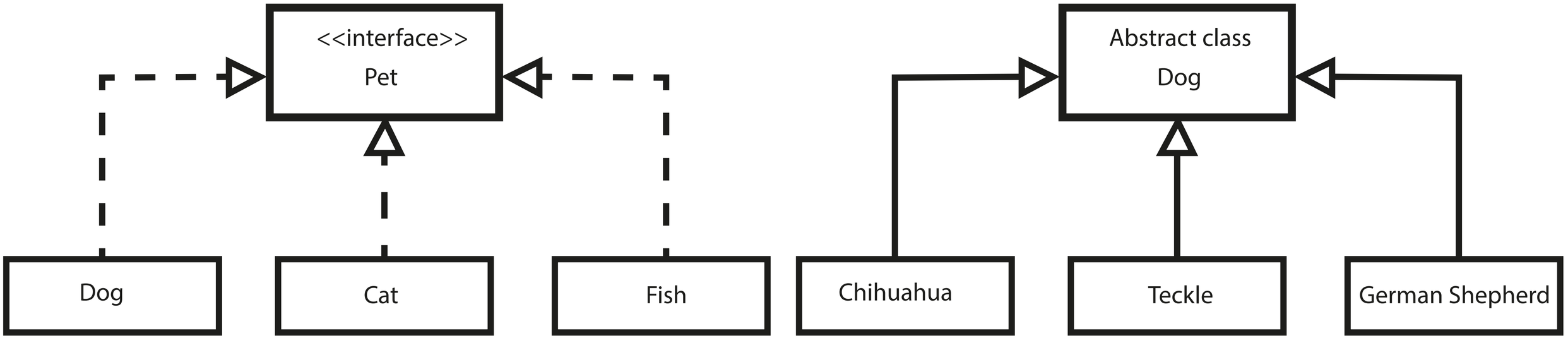
```
interface Animal {
  public void animalSound();
  public void sleep();
}

class Pig implements Animal {
      // implement animalSound and sleep
}
```

```
abstract class Animal {
  public abstract void animalSound();

  public void sleep() {
    System.out.println("Zzz");
  }
}

class Pig extends Animal
      // implement animalSound here
}
```

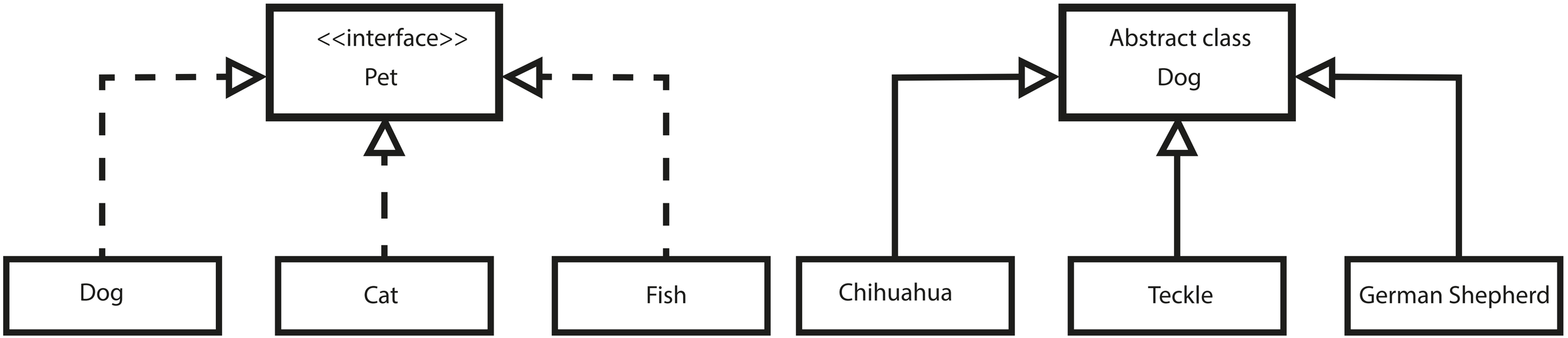
Main difference: Interfaces can not have instance variables (and therefor no constructors).

Note: this is Java, not C++!

Abstract classes vs. Interfaces



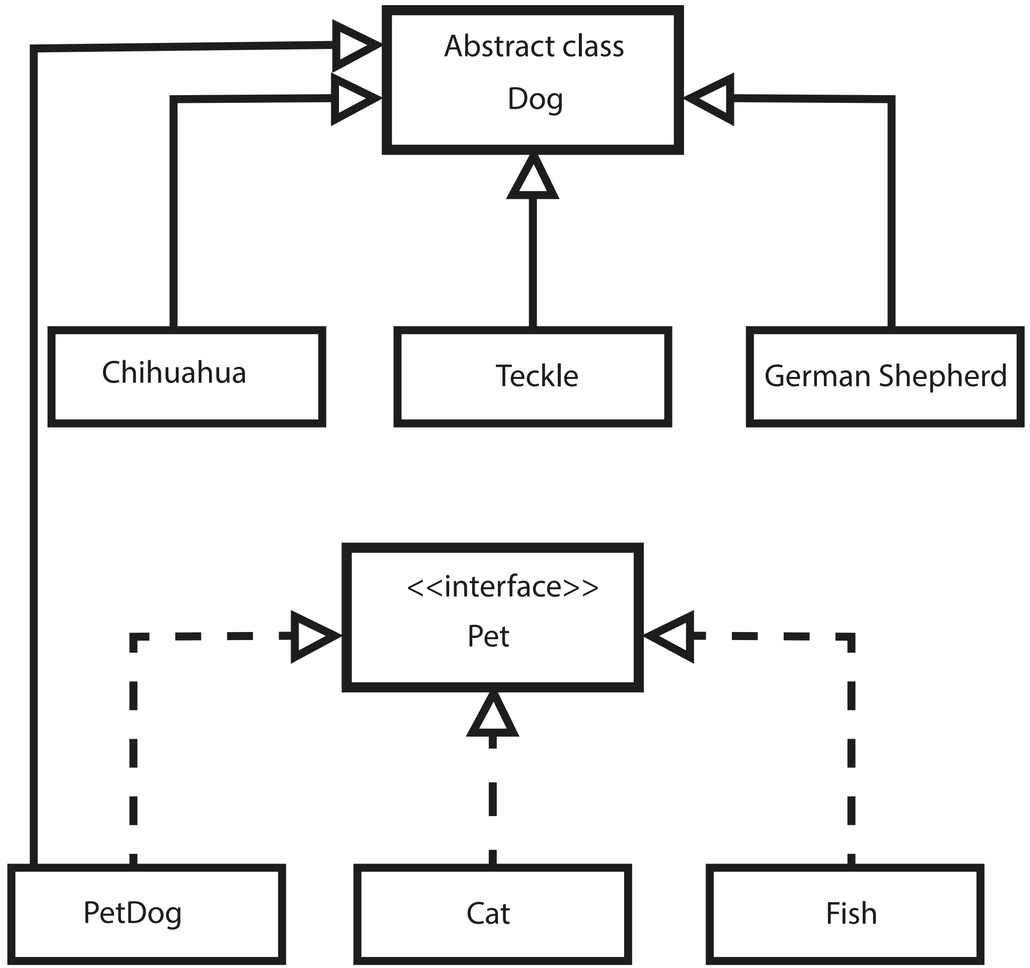
Abstract classes vs. Interfaces: UML



Interface: Fish implements Pet (Generalisation)

Inheritance: Dog is-a Mammal (Realization)

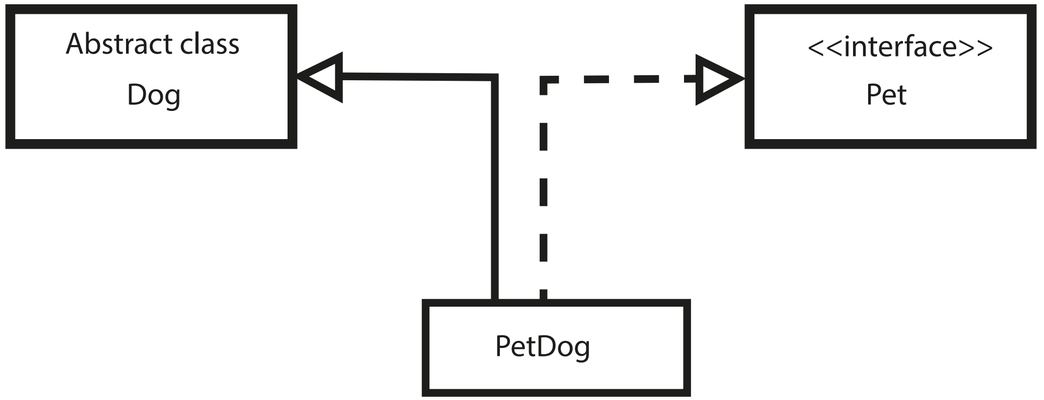
Abstract classes vs. Interfaces: UML



Inheritance: PetDog is-a Dog (Realization)

Interface: PetDog implements Pet (Generalisation)

Abstract classes vs. Interfaces: UML



Inheritance: PetDog is-a Dog (Realization)

Interface: PetDog implements Pet (Generalisation)

Implementing an Interface in C++

- See Demo Prog5_DogDemo_Interfaces.pdf

So… does a Dog always have a name?

What about a stray dog?

```
class StrayDog : public Dog {
    public:
        void barks();
    private:
};

void StrayDog::barks() {
    cout << "Some stray dog: I always bark in fear of the dog catcher!" << endl;
}
```

Summary Implementing an Interface in C++

- A pure virtual function can *only be declared*, it *cannot be defined*.
- You cannot assign any value other than 0 to the pure virtual function.
- It is not possible to create an instance of a class with virtual functions: It would result in a compilation error.

Interfaces, what's all the fuzz about…

"No client should be forced to depend on methods it does not use."

- Uncle Bob

https://claudiorivera.net/2018/02/04/isp-interface-segregation-principle/

Three ways to implement Interfaces in C++

Option 1: Multiple Inheritance

Option 2: Composition

Option 3: Interface Inheritance + Concrete Class

Option 4: (C++ ≥ 20) Templates and/or Curiously Recurring Template Pattern

Three ways to implement Interfaces in C++

Option 1: Multiple Inheritance

Option 2: Composition

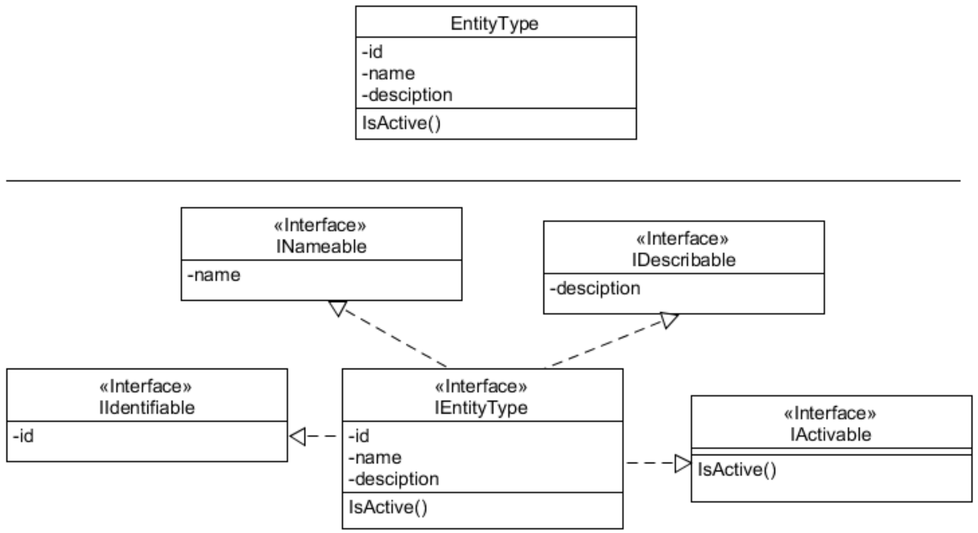
Option 3: Interface Inheritance + Concrete Class

When considering SOLID principles, Option 3: Interface Inheritance + Concrete Class aligns best, especially with the Interface Segregation Principle (ISP) and the Dependency Inversion Principle (DIP).

- Interface Segregation Principle (ISP)

- Dependency Inversion Principle (DIP) - Next week

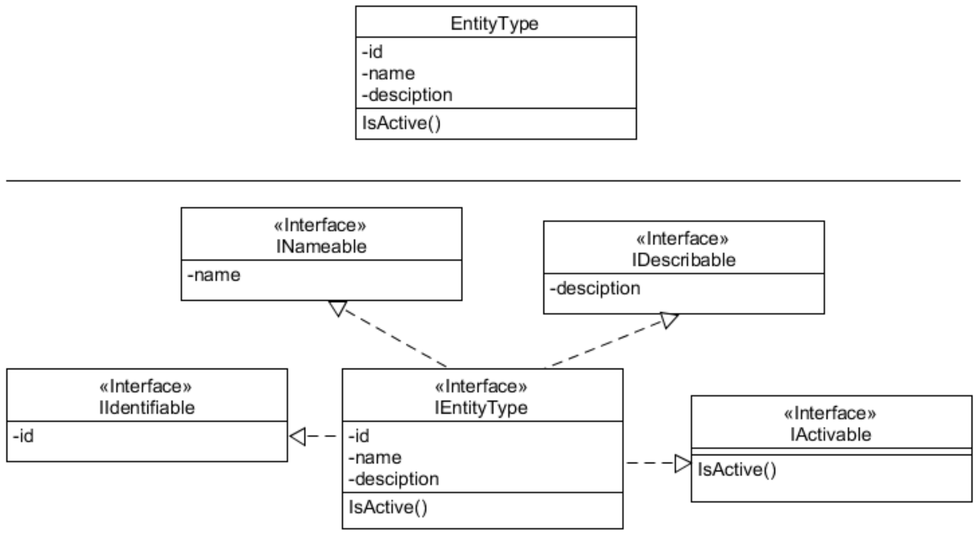
SOLID: Interface Segregation



https://claudiorivera.net/2018/02/04/isp-interface-segregation-principle/

SOLID: Interface Segregation

Make interfaces as slender as possible…



https://claudiorivera.net/2018/02/04/isp-interface-segregation-principle/

SOLID: Interface Segregation: summary

Clients should not be compelled to implement an interface that contains declarations of members or operations that they would not need or never use.

Interfaces that violate this principle are known as 'fat' or polluted interfaces as they contain too many operations (this violation is also known as 'interface bloating').

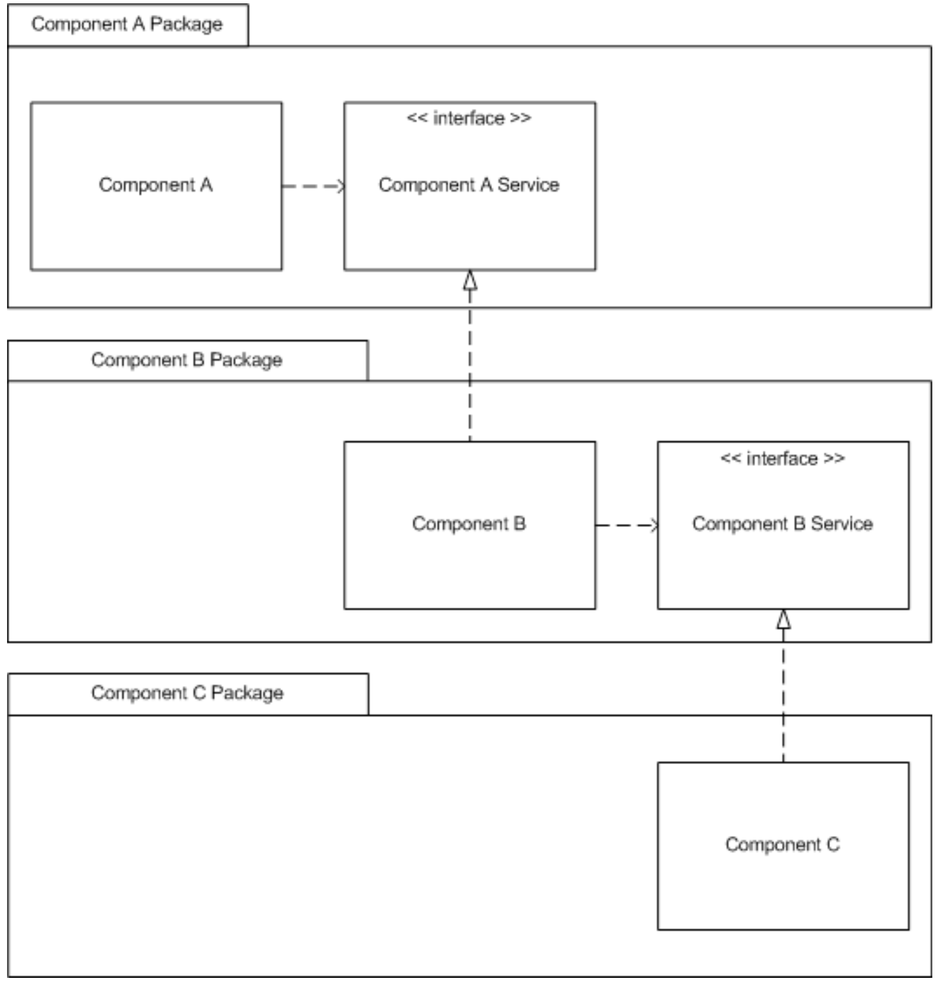
http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

SOLID: Integration Segregation: summary

Helps to sustain low-coupling between the components comprising an application.

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.



https://dzone.com/articles/the-dependency-inversion-principle-for-beginners

What to remember for CPP on Interfaces

- Define them as abstract classes
- How?

What to remember for CPP on Interfaces

- Define them as abstract classes
- Class is made abstract by declaring at least one of its functions as pure virtual function

What to remember for CPP on Interfaces

- Define them as abstract classes
- Class is made abstract by declaring at least one of its functions as pure virtual function
- A pure virtual function is specified by placing "= 0" in its declaration

Demo for Dogs…

Demo for Dogs…

Explanation:

- We maintain the Dog interface with the pure virtual barks() function.
- PetDog and GermanShepherd both inherit from Dog and provide their specific implementations of barks().
- Both classes also have a pet() function to demonstrate their unique responses to petting.
- In main(), we create instances of PetDog (Pluto) and GermanShepherd (Rex) and interact with them to observe their distinct behaviors.

Key points:

- This design adheres to SOLID principles, especially ISP and DIP, promoting loose coupling and flexibility.
- You can easily add more dog breeds by creating new classes that inherit from Dog and implement the required functions.
- Consider using smart pointers if you need to manage the lifetime of Dog objects dynamically.

Links

- https://www.codeproject.com/Articles/10553/Using-Interfaces-in-C
- https://data-flair.training/blogs/interfaces-in-cpp/
- https://www.tutorialcup.com/cplusplus/interfaces.htm
- https://github.com/ksvbka/design_pattern_for_embedded_system/blob/master/design-patterns-for-embedded-systems-in-c-an-embedded-software-engineering-toolkit.pdf
- https://martinfowler.com/articles/injection.html

Any questions?

Try it yourself…

Cue: “What is the best way to use interfaces in C++?”

https://copilot.microsoft.com/
https://gemini.google.com/
https://claude.ai/
https://chatgpt.com/

Which one do you prefer?!