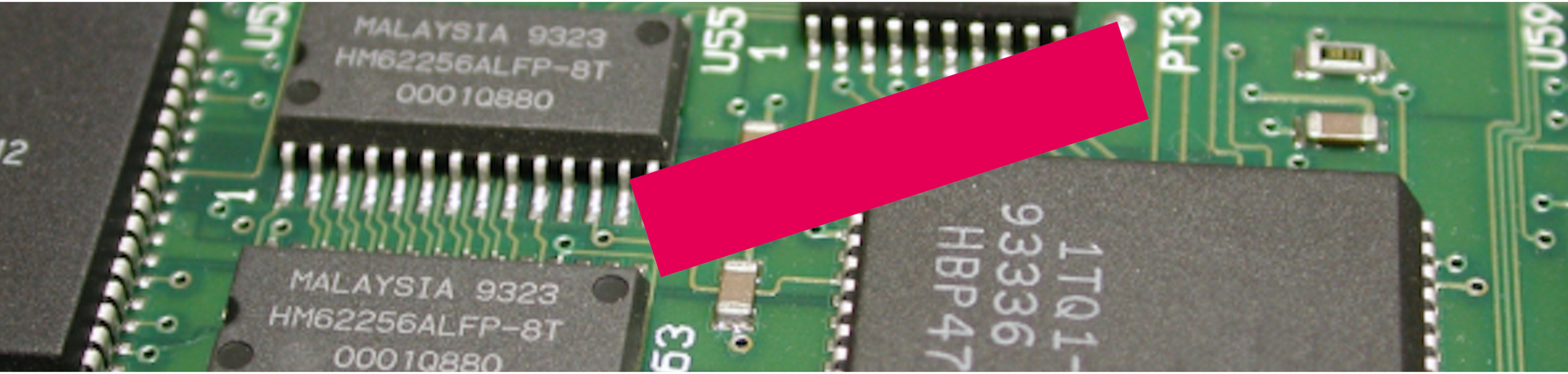
Embedded Systems Development - 4. Interfaces and abstract classes



Electrical Engineering / Embedded Systems
School of Engineering and Automotive

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

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

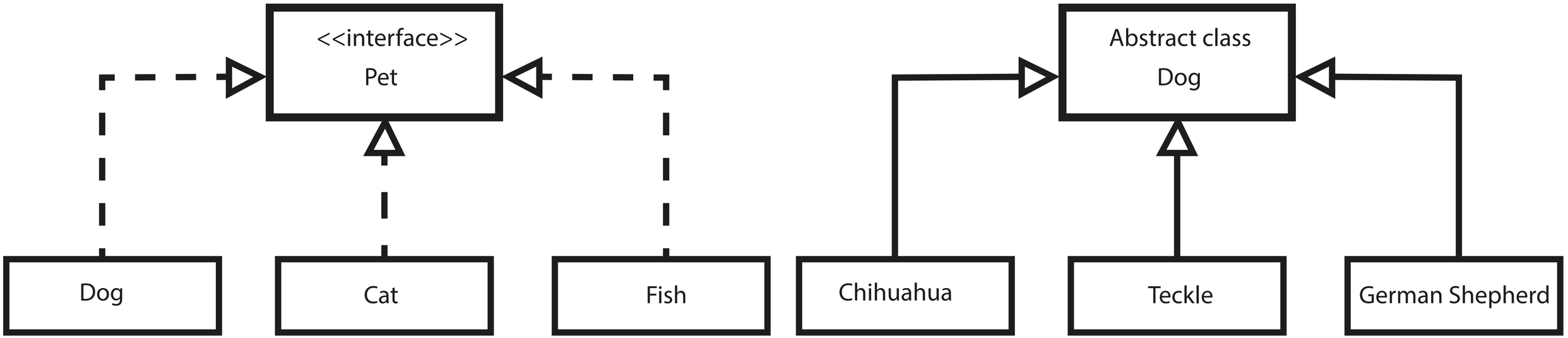
```
interface Animal {
  public void animalSound();
  public void sleep();
}

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
}
```

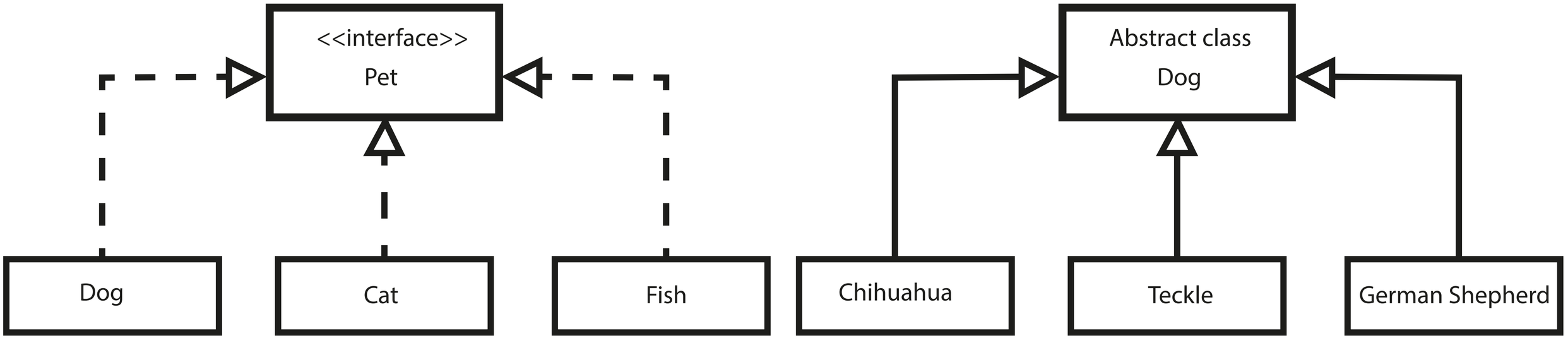
Main difference: Interfaces can not have instance variables (and therefor no constructors).

Note: this is Java, not C++!

Abstract classes vs. Interfaces



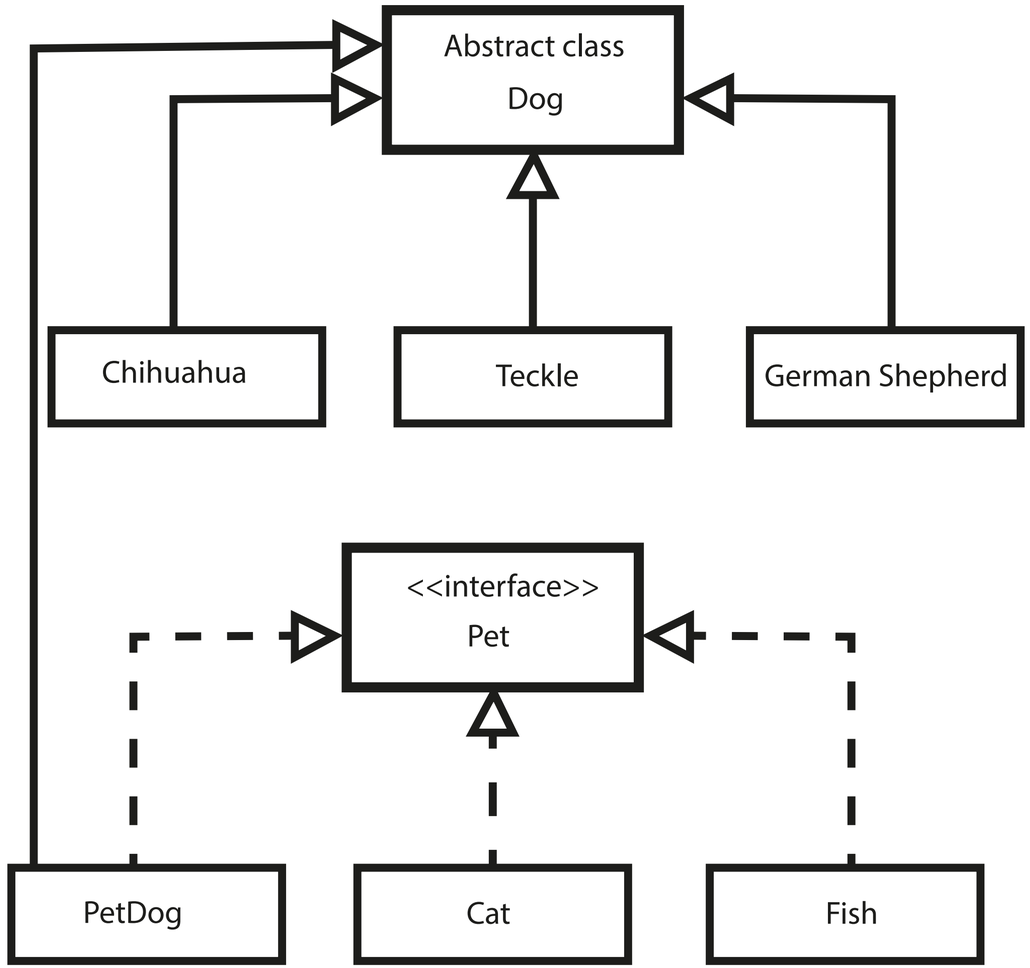
Abstract classes vs. Interfaces: UML



Interface: Fish implements Pet (Generalisation)

Inheritance: Dog is-a Mammal (Realization)

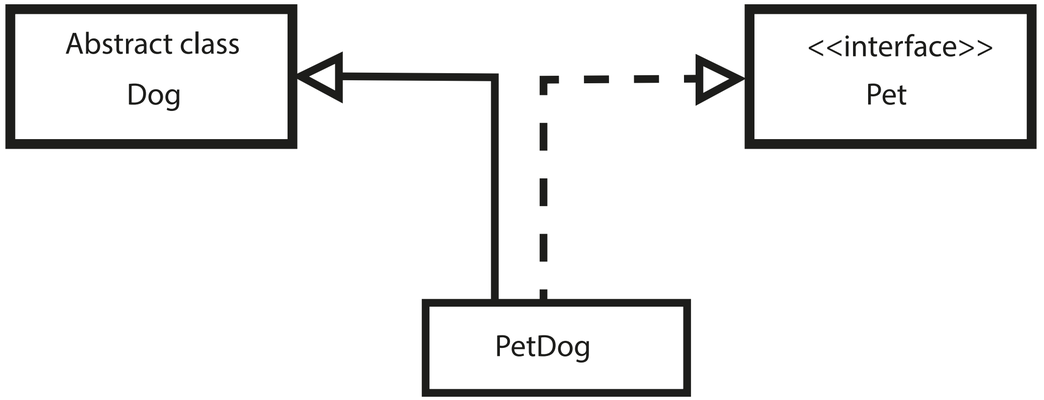
Abstract classes vs. Interfaces: UML



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++

```
class Dog {
   public:
       virtual void barks() = 0;
       ~ Dog();
   private:

};
```

Trick question: can a PetDog bark?

Implementing an Interface in C++

```
class PetDog : public Dog, implements IPet {
    public:
        void gotPetted();
        PetDog(string name);
       void barks();
    private:
        string getName();
        string _name;
};
```

Implementing an Interface in C++

```
using namespace std;

#include "CppInterfaces.h"

DeclareInterface(IPet)
   virtual void gotPetted() = 0;
   virtual string getName() = 0;
EndInterface
```

Implementing an Interface in C++

```
//
// CppInterfaces.h
// https://www.codeproject.com/Articles/10553/Using-Interfaces-in-C
//

#ifndef InterfaceClass
#define InterfaceClass

#define Interface class

#define DeclareInterface(name) Interface name { \
          public: \
          virtual ~name() {}

#define DeclareBasedInterface(name, base) class name : public base { \
          public: \
          virtual ~name() {}

#define EndInterface };

#define implements public

#endif
```

Interface IPet

```
using namespace std;

#include "CppInterfaces.h"

DeclareInterface(IPet)
   virtual void gotPetted() = 0;
   virtual string getName() = 0;
EndInterface
```

Apparently a pet is something that can get petted and has a name.

PetDog: fusion of abstract class and interface implementation

```
class PetDog : public Dog, implements IPet {
    public:
        void gotPetted();
        PetDog(string name);
        void barks();
    private:
        string getName();
        string _name;
};

void PetDog::barks() {
    cout << "Woof, woof, woof (meaning: I don't bite!)" << endl;
}

void PetDog::gotPetted() {
    cout << _name << " just wagged it's tail to you..." << endl;
}

PetDog::PetDog(string name) {
    _name = name;
}

string PetDog::getName() {
    return _name;
}
```

GermanShepherd: only abstract class here…

```
class GermanShepherd : public Dog {
    public:
        GermanShepherd(string name);
        void barks();
    private:
        string _name;
};

GermanShepherd::GermanShepherd(string name) {
    _name = name;
}

void GermanShepherd::barks() {
    cout << _name << " barks loudly!" << endl;
}
```

Implementing an Interface in C++

```
int main() {

    PetDog pluto("Pluto");
    pluto.gotPetted();
    pluto.barks(); // of course, once a dog, always a dog...

    GermanShepherd rex("Rex");
    rex.barks();
    //rex.gotPetted(); you can't pet Rex, he is not a PetDog after all...

}
```

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.

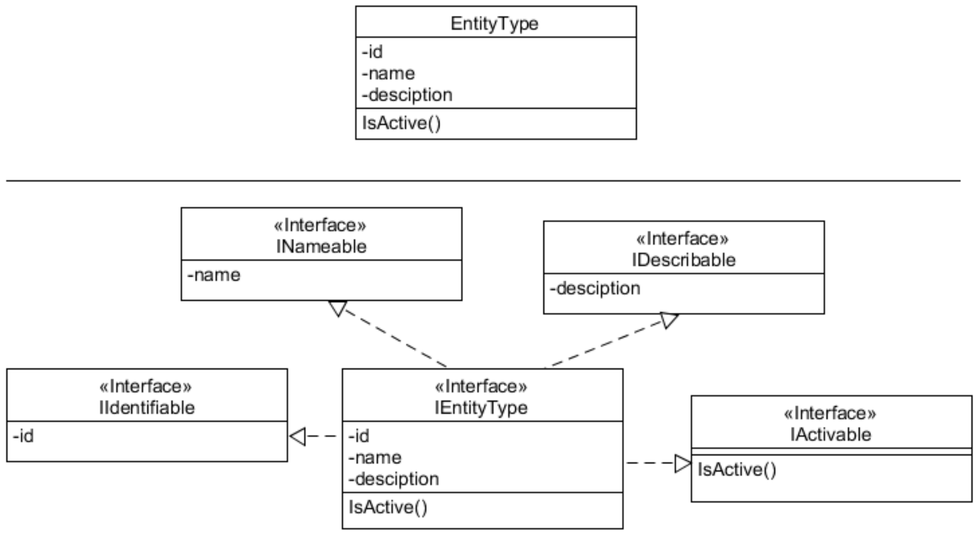
Interfaces, what's all the fuzz about…

"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/

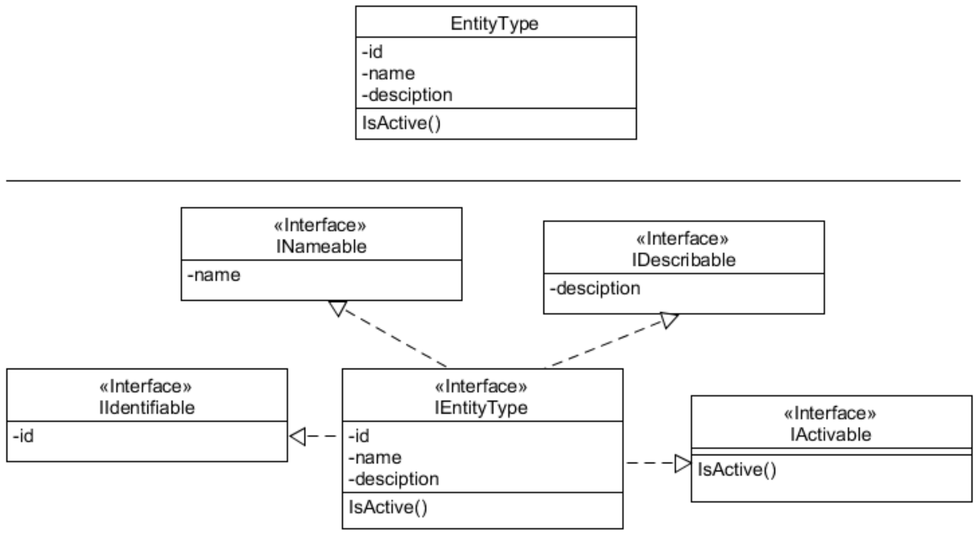
SOLID: Interface Segregation



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').

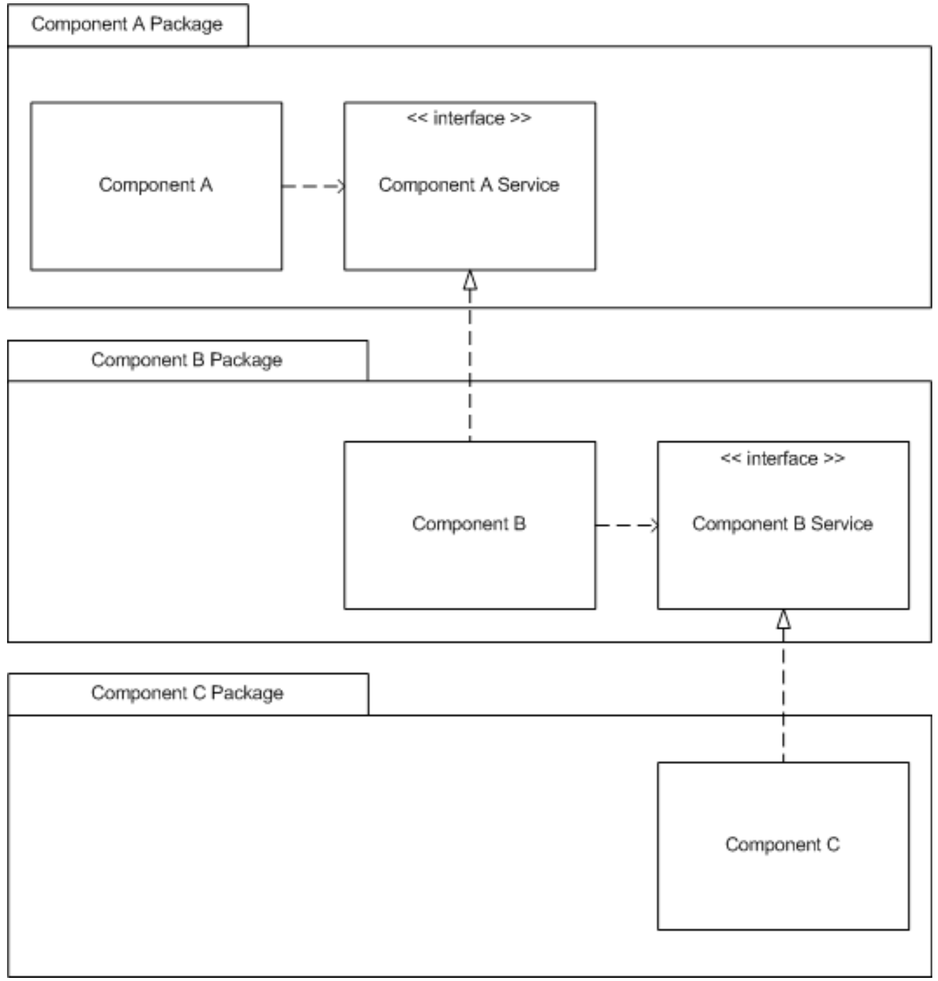
http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

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

Casus: refining an Arduino library

https://www.adafruit.com/?q=i2c

E.g. https://github.com/adafruit/Adafruit_MMA8451_Library

- Open and read the library.
- Check for *dependencies*.
- What can you say about *coupling*?
- Try to improve the library.

Hint: the library for MPL3115A2 is already a big improvement…

Typical Sensor Library

```
#include <Wire.h>
#include <Adafruit_MMA8451.h>

static inline uint8_t i2cread(void) {
  #if ARDUINO >= 100
  return Wire.read();
  #else
  return Wire.receive();
  #endif
}

static inline void i2cwrite(uint8_t x) {
  #if ARDUINO >= 100
  Wire.write((uint8_t)x);
  #else
  Wire.send(x);
  #endif
}

bool Adafruit_MMA8451::begin(uint8_t i2caddr) {
 Wire.begin();
  ...
}
```

Typical Sensor Library

```
#include <Wire.h>
#include <Adafruit_MMA8451.h>

static inline uint8_t i2cread(void) {
  #if ARDUINO >= 100
  return Wire.read(); // some global object...
  #else
  return Wire.receive();
  #endif
}

static inline void i2cwrite(uint8_t x) {
  #if ARDUINO >= 100
  Wire.write((uint8_t)x);
  #else
  Wire.send(x);
  #endif
}

bool Adafruit_MMA8451::begin(uint8_t i2caddr) {
 Wire.begin(); // separation of concerns?!
  ...
}
```

Patterns: Dependency Injection

Dependency Injection:

Three ways:

- component can create the dependency (at construction)
- component can look up the dependency, by referring to a global variable
- component can have the dependency passed to it where it is needed

Patterns: Dependency Injection

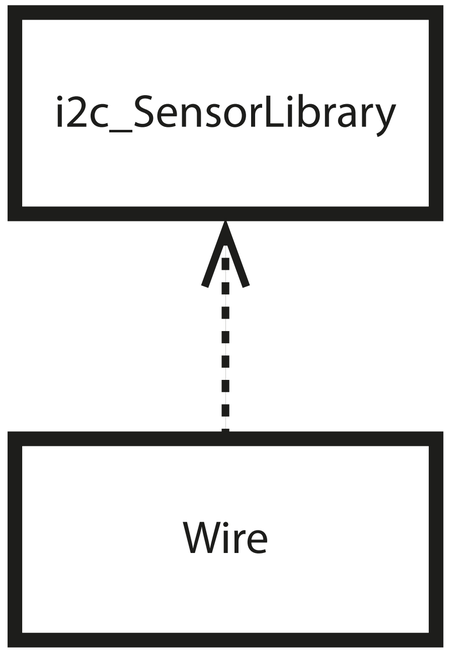
Making a class's dependencies explicit and requiring that they be injected into it is a good way of making a class more *reusable, testable and decoupled* from others. There are several ways that the dependencies can be injected:

- Setter Injection
- Constructor Injection
- Property Injection

Using Dependency Injection in an Arduino Library

```
#include <Wire.h>
#include <Adafruit_ADS1015_Flex.h>


Adafruit_ADS1015_Flex ads = Adafruit_ADS1015_Flex(&Wire);

void setup(void)
{
   Wire.begin(); // leave this to the 'master'

   ...

   ads.doSomethingUseful();
}
```

UML Dependency Injection



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?