Interfaces

BRIAN LAMARCHE

COMPUTER SCIENCE 323 – SOFTWARE DESIGN

Polymorphism

Polymorphism when broken down means:

- Poly = multiple
- Morphs = forms

In CS, this means objects of different types can have the same interface.

This is key

Language *Interfaces*

In C# and Java we have things called *Interfaces*

- Interfaces only specify what public methods an object implements

Interfaces only contains signatures of methods, delegates, and events

Interfaces **PROVIDE NO** implementation.

Language *Interfaces*

An object **realizes** an interface if it:

- Defines implementation for all method signatures defined by said interface

Interfaces are extremely useful, and allow for great extensibility

In C# and this class, prefix your interfaces with capital “i”

- ICamera
- IMissileLauncher
- IImageProcessor

Prefixing allows developers to distinguish between and interface and an abstract class.

Example: Interfaces

```
public interface IMissileLauncher
{
        void Start();
        void Stop();
        void Move(double x, double y, double speedX, double speedY);
        void Fire();
        void Fire(double x, double y);
        CalibrationData Calibrate(double x, double y);
}
```

The above defines only what the object that realizes it should look like. Public is implied, so no scope identifiers are listed on the method signature. No implementation is given.

When to use?

Use interfaces when you want an object to exert an implied behavior

- ICamera
  - Acquire Image
  - Turn On
  - Turn Off
- IMissileLauncher

Defining API's

- An application programming interface (API) often define what an objects interface (e.g. method signature) should look like. API's often extensibility to your application, or to another, where implementation specifics are not understood.

Creating Plug-in architectures

- Plug-in architectures and API's go hand in hand. You may know what an object needs to look like, but allow the 3rd party developer to define the implementation.

Interfaces vs. Abstract Class?

	Interfaces	**Abstract Classes**
Implementation?	Provides None	Can provide, but does not have to
Is a?	N/A	x
Multiple …	Can realize multiple interfaces	Can only inherit from one base class
Changes to method signatures	All objects that realize the interface have to update their interfaces	Sub-classes automatically inherit changes
Extensibility (subjective)	Allows for objects to add	

An interface is not better than an abstract class, or vice versa. It all depends on intent of design…

Talking Points – My $1.05

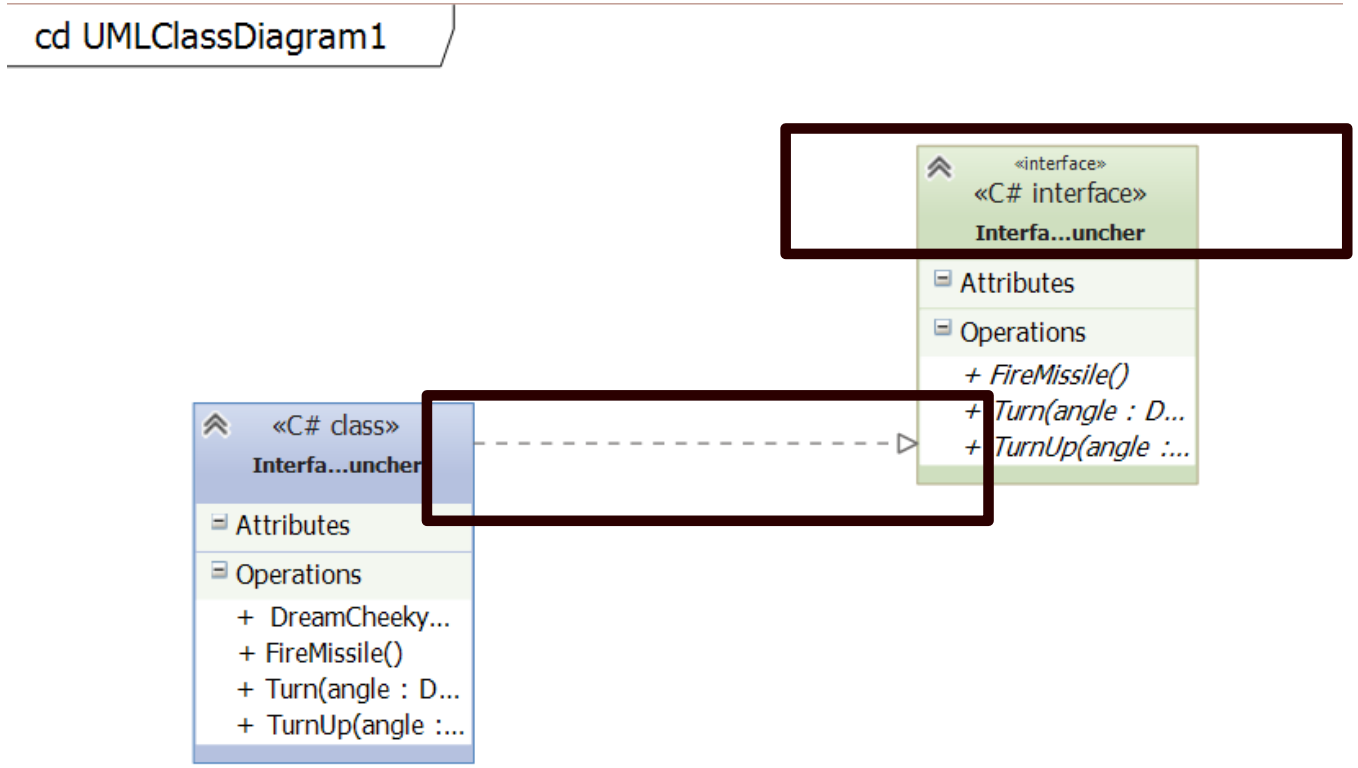
A **purely abstract base class** *(all method signatures are marked abstract)* is essentially an interface, however, you now lock all sub-classes to only being able to inherit from that one object. This is bad if you only need the interface to comply with a set of method signatures

Abstract class benefits outweigh interfaces when you can provide a common functionality

Interfaces otherwise are generally more flexible, until you need to modify the interface, then you are have to modify everything that realizes it ☹

Interfaces play a big role in later parts of this semester

Example:


cd UMLClassDiagram1
«interface»
«C# interface»
Interfa...uncher
Attributes
Operations
+ FireMissile()
+ Turn(angle : D...
+ TurnUp(angle :...
«C# class»
Interfa...uncher
Attributes
Operations
+ DreamCheeky...
+ FireMissile()
+ Turn(angle : D...
+ TurnUp(angle :...