**schUML Documentation**

**Class Definition**

A Class Definition can be represented by up to 4 boxes:

|  |
| --- |
| **A**  Class Name and Accessibility |
| **B**  Field members |
| **C**  Methods and Properties |
| **D**  Constructors |

1. The Class Name and Class Accessibility Level

|  |  |
| --- | --- |
| **C# Code** | **schUML** |
| public class Shape | +Shape |
| internal class Shape | -Shape |
| public abstract class Shape | +A:Shape |
| public sealed class Shape | +S:Shape |
| public static class Shape | +Shape:s |

* + public: the class is accessible to all outside assemblies (modules)
  + internal: the class is only accessible in the current assembly (default)
  + abstract: the class cannot be instantiated
  + sealed: the class cannot be inherited
  + static: the class cannot be inherited or instantiated

1. Field Members

|  |  |
| --- | --- |
| **C# code** | **schUML** |
| public int nAge; | +nAge:int |
| private int license; | -license:int |
| protected int nAge; | =nAge:int |
| internal int nAge; | $nAge:int |
| public static int nAge | +nAge:int:s |

* public: the field is accessible via a class object outside of the class
* private: the field is accessible only within the class (default)
* protected: the field is accessible only within the class and any derived class
* internal: the field is private to all classes in the current assembly
* static: the field is attached to the class, not to the created object (static fields can be accessed without creating an object of the class)

1. Methods and Properties

|  |  |
| --- | --- |
| **C# code** | **schUML** |
| public int License {get; set;} | +License:int |
| public int License {get;} | +License:int:r |
| public int License {set;} | +License:int:w |
| public Person this[string email] {get; set;} | +this[email:string]:Person |
| public Person this[string email] {get;} | +this[email:string]:Person:r |
| public Person this[string email] {set;} | +this[email:string]:Person:w |
| private string MyMethod(int a) | -MyMethod(a:int):string |
| protected int MyMethod(int a) | =MyMethod(a:int):int |
| internal bool MyMethod() | $MyMethod():bool |
| public abstract int MyMethod() | +MyMethod():int:a |
| public virtual int MyMethod() | +MyMethod():v |
| public static int MyMethod() | +MyMethod():s |

* Age is a property which can have get{} and set{} code blocks. If only the get{} block is included, then it is a read-only property (:r suffix). If only the set{} block is included, then it is a write-only property. The convention is to name property fields using capital case.
* A convention is for a property to be used to provide access to a private class field. For example, above there is a class field defined as:

private int license;

The License property could contain a set{} code block that sets the license field based on the age of the person (refer to the License property in PersonLib)

* this[string email] is an indexer property for a SortedList<string,Person>. Index properties can also be defined as read-write (the default), read-only and write-only
* public: the method/property is accessible via a class object outside of the class
* private: the method/property is accessible only within the class (default)
* protected: the method/property is accessible only within the class and any derived class
* internal: the method/property is private to all classes in the current assembly
* abstract: only the method signature is defined, not any of the implementing code. Any child class that derives from this class must implement the block of code for the method.
* virtual: the default code for the method is defined in the parent class and can be overriden with child-specific code using the "override" keyword (refer to the Work() method in PeopleLib)
* static: the method/property is attached to the class, not to the created object (static members can be accessed without creating an object of the class)

1. Constructors

If more than the default constructor (eg. MyClass()) is required, then they should be documented in the 4th box of the class model.

|  |  |
| --- | --- |
| **C# code** | **schUML** |
| public MyClass(string sName, int nAge) | +(sName:string, nAge:int) |

* in this example, an additional constructor is required which accepts string and int parameters

**Interface Definition**An Interface Definition is represented by 2 boxes:

|  |
| --- |
| **A**  Interface Name and Accessibility |
| **B**  Methods and Properties |

1. Interface Name and Accessibility

|  |  |
| --- | --- |
| **C# Code** | **schUML** |
| public interface iShape | +iShape |
| internal interface iShape | -iShape |

1. Methods and Properties

|  |  |
| --- | --- |
| **C# code** | **schUML** |
| int Age {get; set;} | Age:int |
| int Age {get;} | Age:int:r |
| int Age {set;} | Age:int:w |
| Person this[string email] {get; set;} | this[email:string]:Person |
| Person this[string email] {get;} | this[email:string]:Person:r |
| Person this[string email] {set;} | this[email:string]:Person:w |
| string MyMethod(int a) | MyMethod(a:int):string |

* interfaces can only contain method and property signatures, they cannot contain blocks of code. The classes that inherit an interface must implement the code for any methods or properties defined in the interface.
* this[string email] is an indexer property for a SortedList<string,Person>. Index properties can also be defined as read-write (the default), read-only and write-only

**Inheritance and Base Constructors**

A class can only inherit from one parent class, but can inherit any number of interfaces

For example the Dog class may inherit the Pet class, and the IPet and IDog interfaces:

public class Dog : Pet, IDog, IPet

Using the base keyword, a class constructor can call a constructor in the parent class:

public Dog(string szLicense, string szName, int nAge) : base(szName, nAge)

{

this.license = szLicense;

}

* the constructor Pet(string name, int age) sets the name and age fields
* the constructor Dog(string szLicense, string szName, int nAge) calls the Pet constructor to set the name and age fields then sets the license field

**Using YUML to Create schUML Diagrams**

Using Visual Studio Code with the yUML v3.5.1 extension by Jaime Olivares, create files with a .yuml extension and the following contents:

1. The first 3 lines of the yuml file must be the following:

// {type:class}

// {generate:true}

// {direction:topDown}

* these define the type of UML to generate, whether to actually generate it, and the direction to render it
* direction can be either topDown or leftRight

1. Each class and interface is defined using square brackets
2. The vertical pipe character (|) delimits between the boxes in the class or interface definition
3. The class and interface can be defined separately from their inheritance.

For example, class Shape may inherit from IDrawObject:

* Class Shape may be defined as:

[+Shape|+PI:double;=x:double;=y:double|+Area():double:v;+DrawMe():v|();(x:double,y:double)]

* Interface IDrawObject may be defined as: [+I:IDrawObject|DrawMe()]
* And the inheritance can be defined using only the Interface and Shape names.

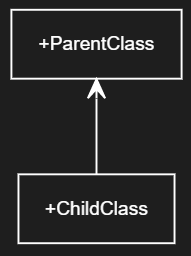
For example: [+I:IDrawObject]^[+Shape]

* The order of inheritance in the yuml is [parent]^[child]

1. Different symbols control the line and arrow styles based on the type of inheritance:

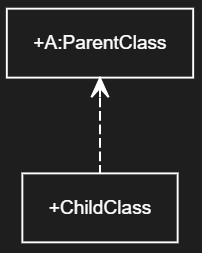
* Class inheritance illustrated by solid line and solid arrow:

[+ParentClass] <- [+ChildClass]



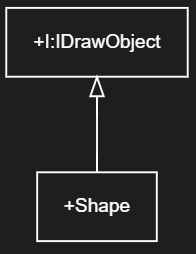
* Abstract class inheritance illustrated by dotted line and solid arrow:

[+A:ParentClass] <-.- [+ChildClass]



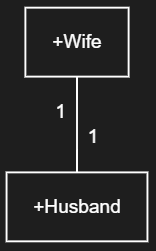
* Interface inheritance illustrated by solid line and outlined arrow:

[+I:IDrawObject] ^ [+Shape]

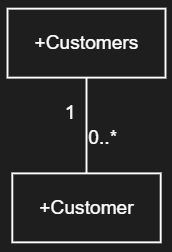


* 1-to-1 Class Relationship (no inheritance but useful to indicate related class objects):

[+Wife] 1-1 [+Husband]



* 1-to-Many Class Relationship:



Here is the Shape class represented in C# and schUML:

|  |  |
| --- | --- |
| **C# code** | **schUML** |
|  | // {type:class}  // {generate:true}  // {direction:topDown} |
| public interface IDrawObject  {  void DrawMe();  } | [+I:IDrawObject|DrawMe()] |
| public class Shape : IDrawObject  {  public const double PI = Math.PI;  protected double x, y;  public Shape()  {  }  public Shape(double x, double y)  {  this.x = x;  this.y = y;  }  public virtual double Area()  {  return this.x \* this.y;  }  public virtual void DrawMe()  {  }  } | [+Shape|+PI:double;=x:double;=y:double| +Area():double:v;+DrawMe():v|();(x:double,y:double)]  [+I:IDrawObject]^[+Shape] |
| public class Circle : Shape  {  public Circle(double r) : base(r, 0)  {  }  public override double Area()  {  return PI \* x \* x;  }  } | [+Circle||+Area():double:o|(r:double):base(r,0]  [+Shape]<-[+Circle] |
| class Sphere : Shape  {  public Sphere(double r) : base(r, 0)  {  }  public override double Area()  {  return 4 \* PI \* x \* x;  }  } | [-Sphere||+Area():double:o| (r:double):base(r,0)]  [+Shape]<-[-Sphere] |

|  |  |
| --- | --- |
| class Cylinder : Shape  {  public Cylinder(double r, double h) : base(r, h)  {  }  public override double Area()  {  return 2 \* PI \* x \* x + 2 \* PI \* x \* y;  }  } | [-Cylinder| |+Area():double:o |(r:double, h:double):base(r,h)]  [+Shape]<-[-Cylinder] |
| class Rectangle : Shape  {  public bool IsSquare  {  get  {  if( x == y )  {  return true;  }  else  {  return false;  }  }  }  public Rectangle(double w, double h) : base(w,h)  {  }  } | [-Rectangle| |+IsSquare:bool:r| (w:double,h:double):base(w,h)]  [+Shape]<-[-Rectangle] |
| public class Blood : IDrawObject  {  public virtual void DrawMe()  {  }  } | [+Blood||DrawMe():v]  [+I:IDrawObject]^[+Blood] |

