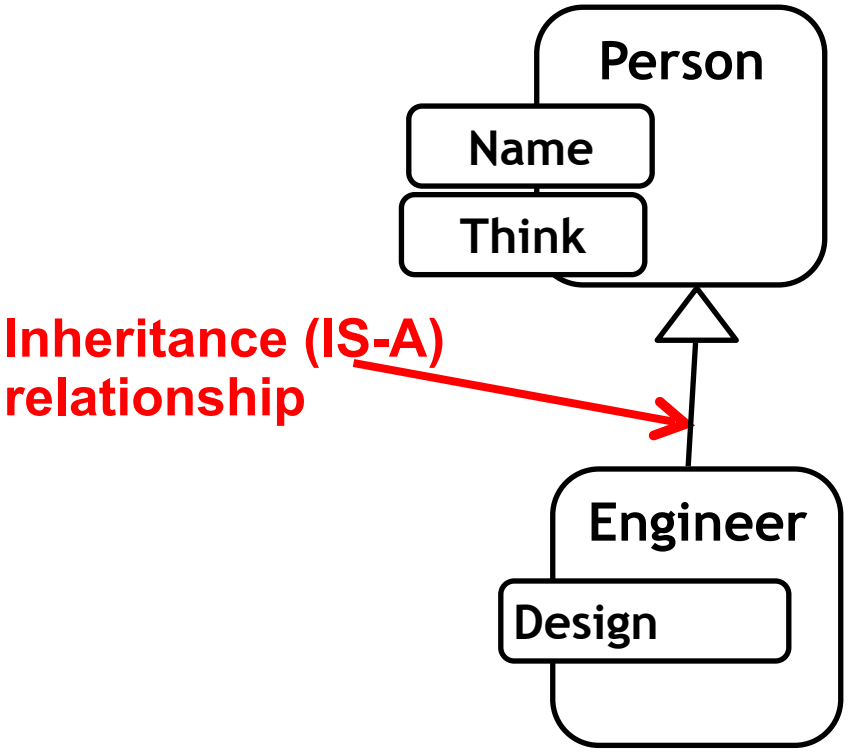
Polymorphism (Chapter 15)

Topics

Object-Oriented Design Technique:

- Review object inheritance
- What is polymorphism?
- Why polymorphism?

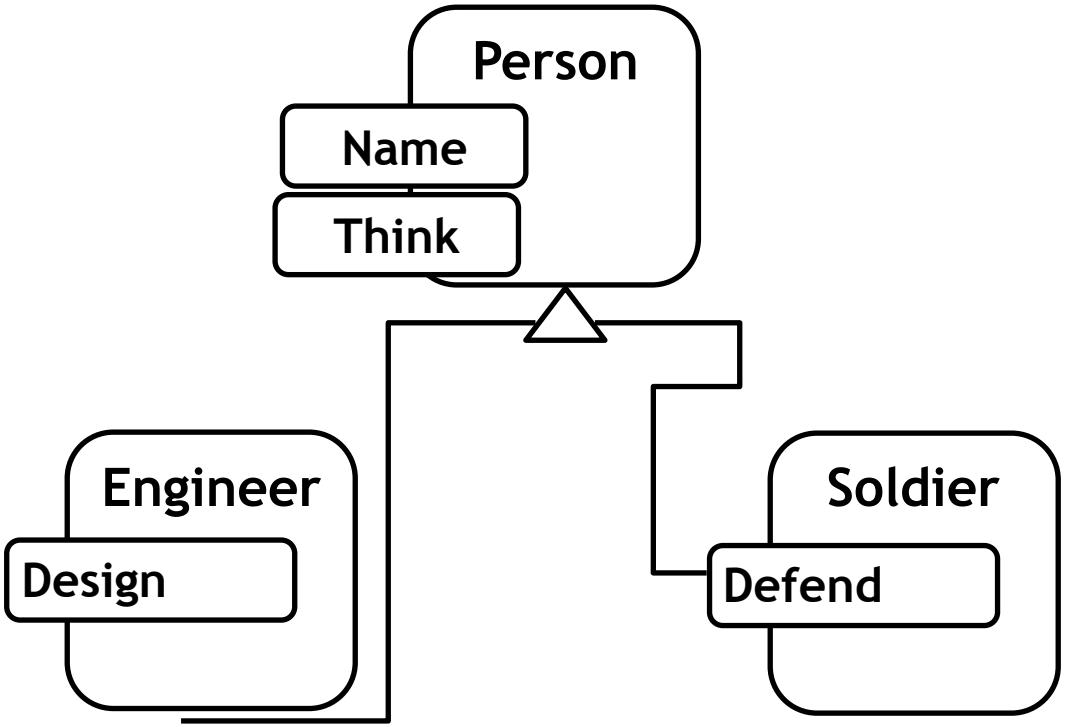
Objects can **"Inherit" properties and behaviors**



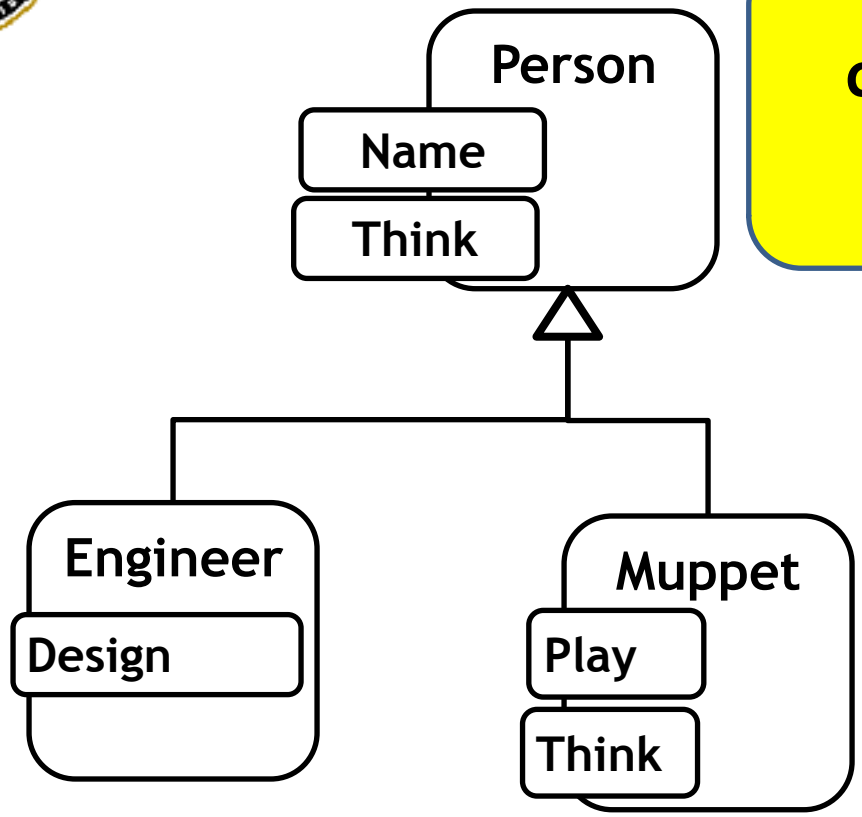
The base class (superclass) defines properties and behaviors that are common to many classes

The subclass (derived class) can inherit and extend behaviors

Exercise: Implement a **Soldier**, who is also a type of **Person**! Create a Soldier object.



Objects can "**override**" behaviors



The base class defines common properties and behaviors

The subclass can inherit and extend behaviors

The derived class can also override behaviors

Overriding Base Class Methods

- **Overriding**: create a method in a derived class that has the ***same name and parameters*** as a base class method

- **Replaces** a method in base class with specialized logic for the derived class

Access to Base class method

- When a method is overridden, all objects of derived class use the overriding function.

- If you want to call the base class version, use the **scope resolution operator (::)** , e.g.

```
Person::Think();
```

Let's see this in our example …

In-class Exercise

- Use your Persons solution, with Engineers, Bellhops, Secretaries, Executives, and Soldiers
  - Go through each of the classes, and decide whether overriding think makes sense for other subclasses
  - Maybe it also makes sense to override getName()
    - E.g. you might add “Mr./Mrs.” to the name of a business executive

Assignment rule: a **derived class object** can be assigned to a **base class pointer**

- Because a **Muppet IS-A Person**, we can assign a Muppet to a **pointer** of a Person class:

```
Person* person = new Muppet("Elmo");
```

- An **Engineer IS-A Person**, we can assign an Engineer to a **pointer** of a Person class:

```
Person* person = new
Engineer("Mike");
```

I can now have **ONE** vector of "**Person** *", and store any object derived from the Person class!

```
vector <Person*> population;

population.push_back(new Engineer("Mike"));
population.push_back(new Soldier("Sue"));
population.push_back(new Muppet("Fozzy"));


for (int i=0; i < population.size(); ++i)
    population[i]->think();
```

Bad news, though. The think function that was called was for Person, not Soldier or Muppet. Uh oh …

Virtual Methods

- To allow a base class pointer to call an overridden method, you must declare the base class method as **virtual**

```
class Person {
    virtual string think() { … }
};
```

- Call the overridden method. E.g.

```
Person *p1 = new Muppet("Elmo");
p1->Think(); // calls the overridden method
```

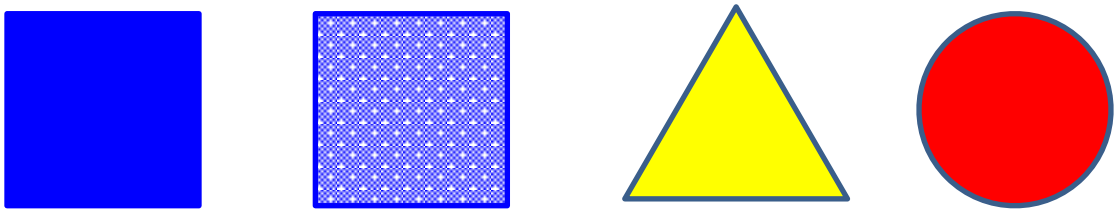
Polymorphism in object-oriented design is the ability for an object to take different forms.

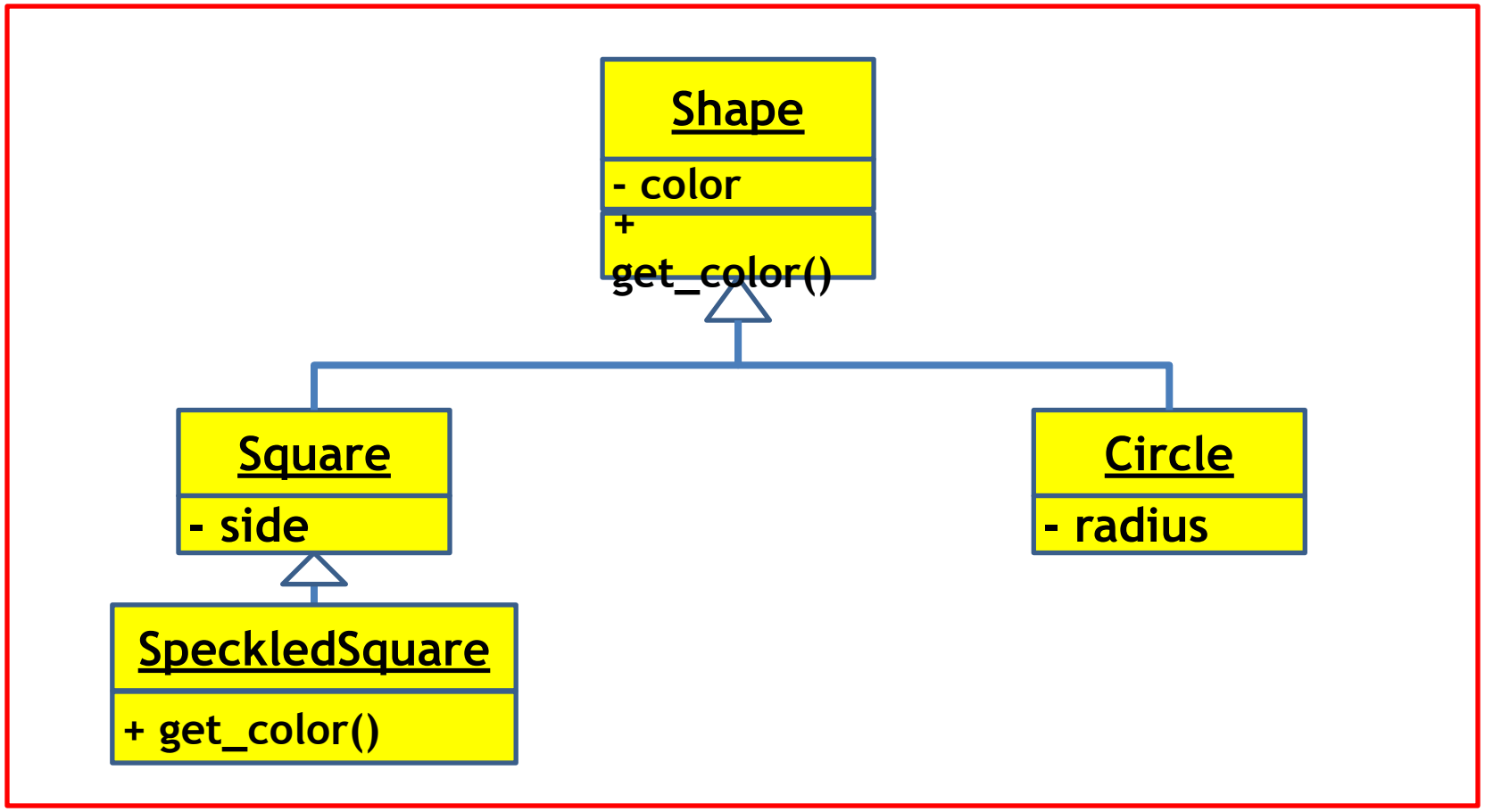
Like transformers!
I'm a Vehicle!
Now I'm a CAR!…
Now I'm a Truck!…
Now I'm a Jet!

In-class Exercise

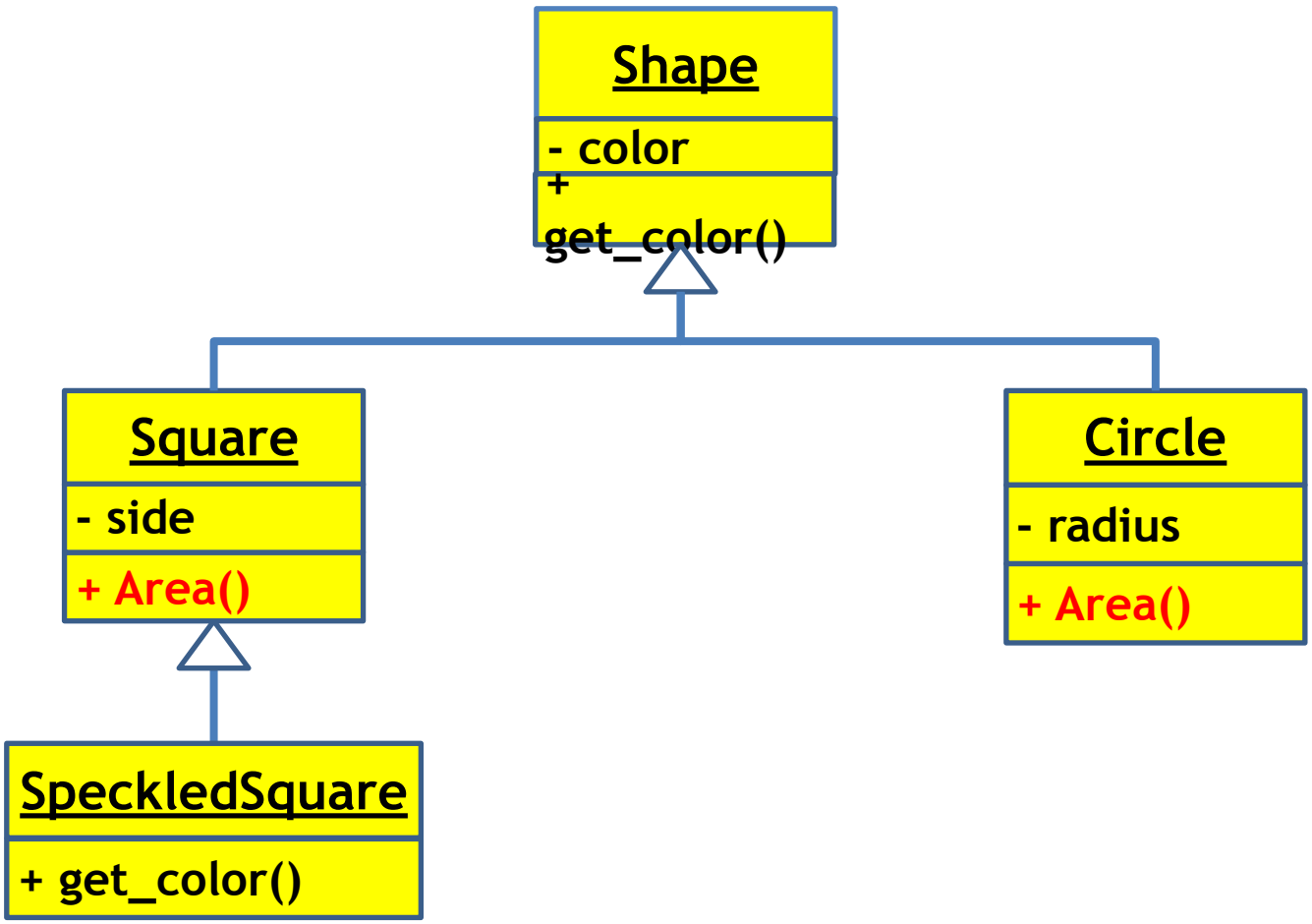
- Go back to your Persons solution
- Make a vector of Person* objects, and assign new pointers to them
- Loop through your vector, and have each Person object “think”
- Make sure you delete your pointers!

Lets model colored shapes!

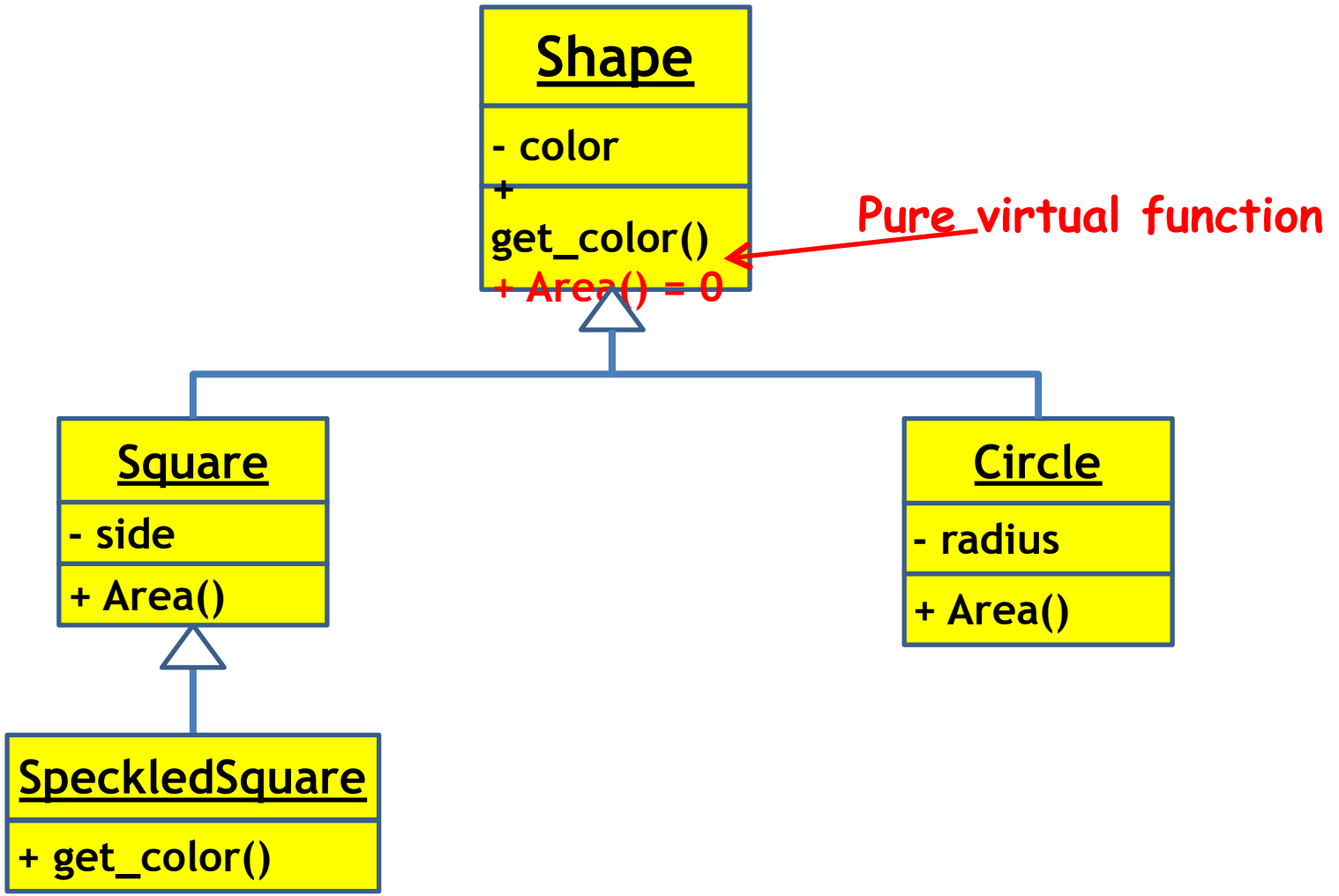




Area for **Shapes**.



Abstract Base Class.



An Abstract Base Class has at least ONE pure virtual function

Abstract Base Class versus Non-Abstract Class.

You cannot create an object of a Abstract Base Class E.g.

```
Shape *obj = new Shape();
```

But you can assigned derived class objects to a pointer to a abstract base class. E.g.

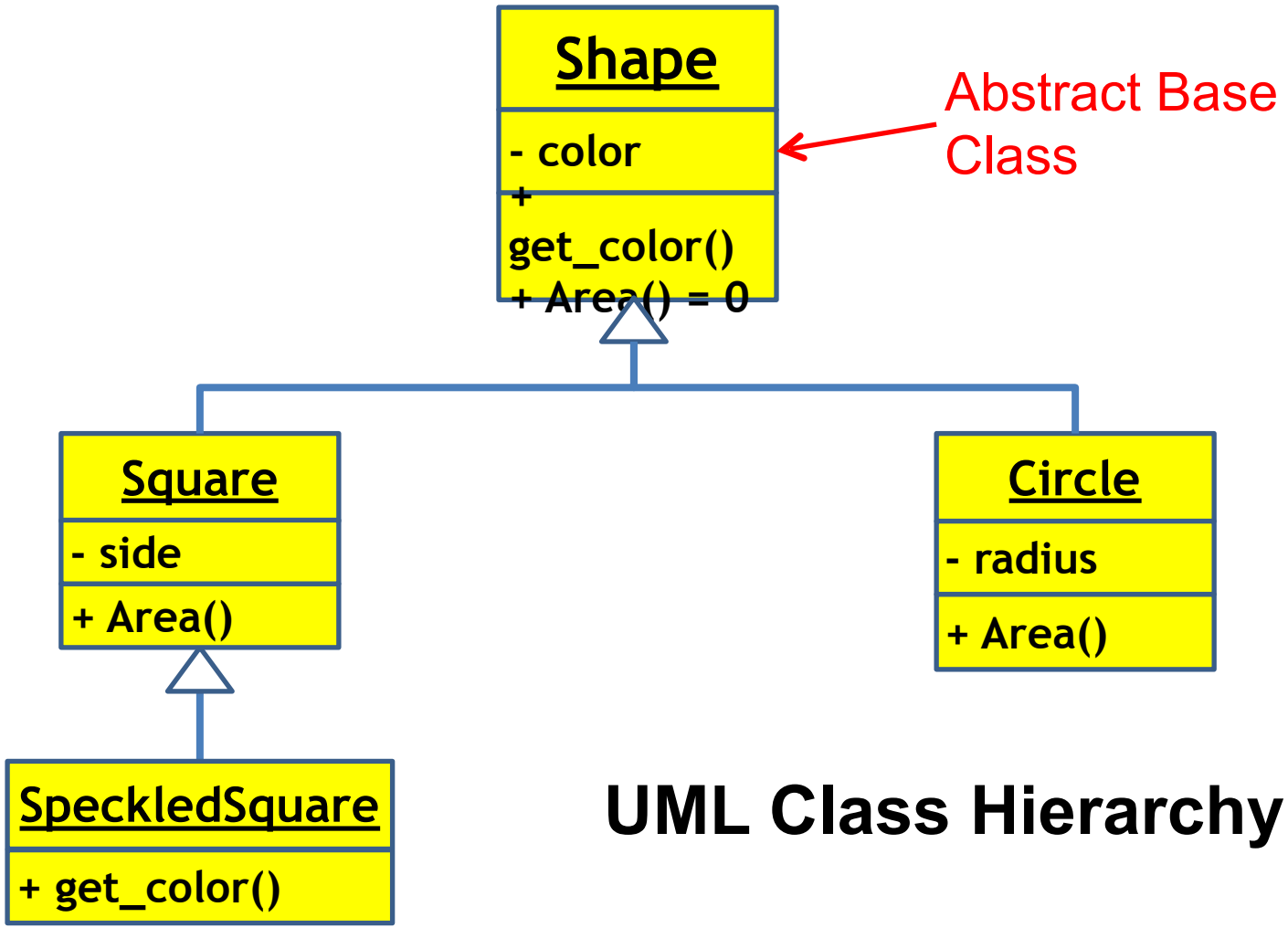
```
Shape *obj = new Square();

vector<Shape *> shape_collection;
shape_collection.push_back(new Square());
shape_collection.push_back(new Circle());
```

Let’s see this

- We can make Person::think() abstract

Review: Shapes UML Diagram



UML Class Hierarchy

Review assignment rule: a derived class object can be assigned to a base class pointer

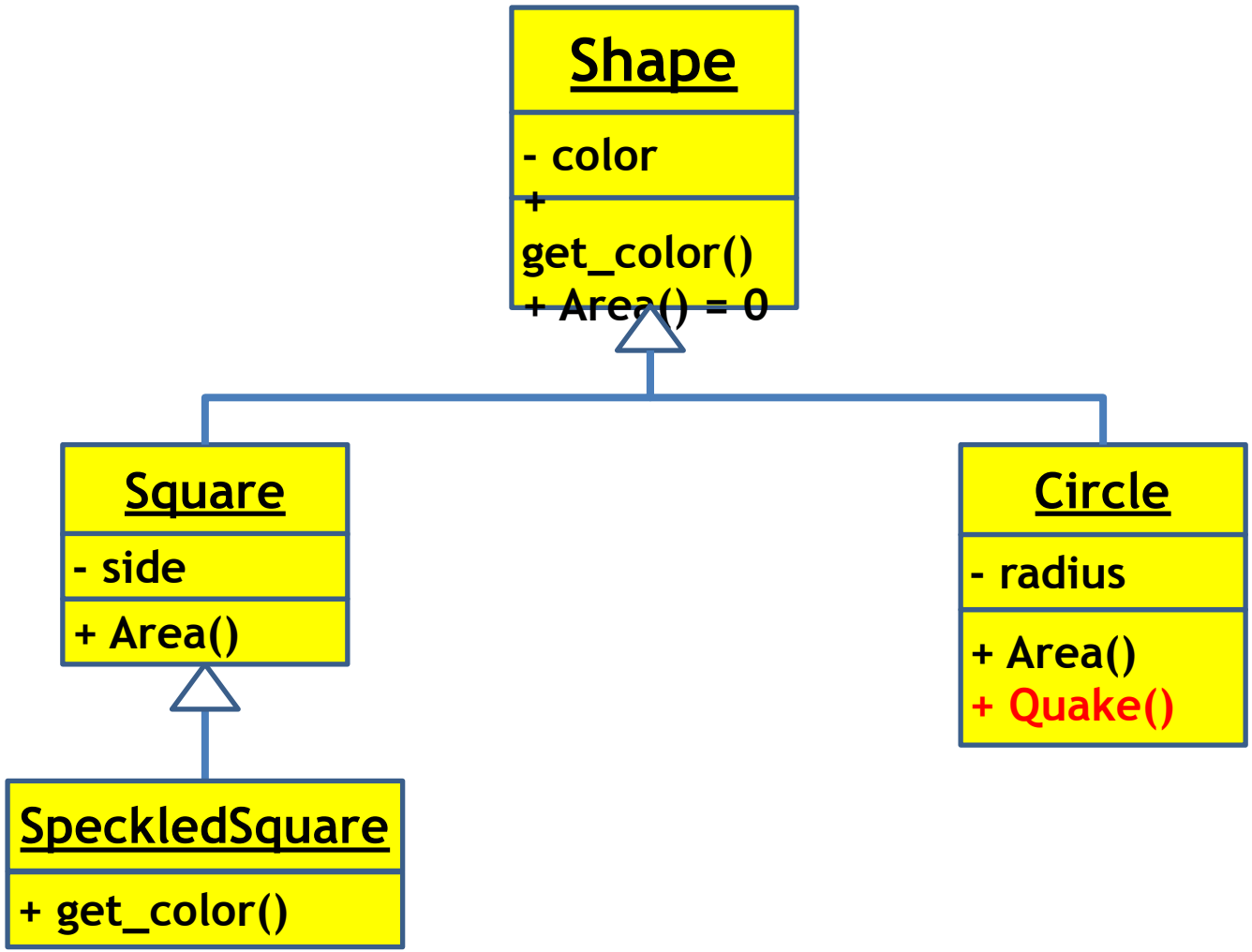
- Because a **Square IS-A Shape**, we can assign a **Square** to a **Shape pointer** :

```
Shape* shapes = new Square("pink",
10);
```

- A **Circle IS-A Shape**, we can assign an **Circle** to a **Shape pointer** :

```
Shape* shapes = new Circle("blue",
3.2);
```

Dynamic Casting Example



Checking Types and Dynamic Casting …

- You can check the real type of "**Shape**", by using the **typeid()** operator:

```
cout << typeid(*shape).name() << endl;
```

- You can get your **subclass** back by using the **dynamic_cast()** operator:

```
if (typeid(*shape) == typeid(Circle))
 dynamic_cast<Circle *>(shape)->Quake();
```

Again, let's try this out

Polymorphism Example: iPhone App