OO Analysis and Design with UML 2 and UP

Dr. Jim Arlow,

Zuhlke Engineering Limited

Analysis - dependencies

What is a dependency?

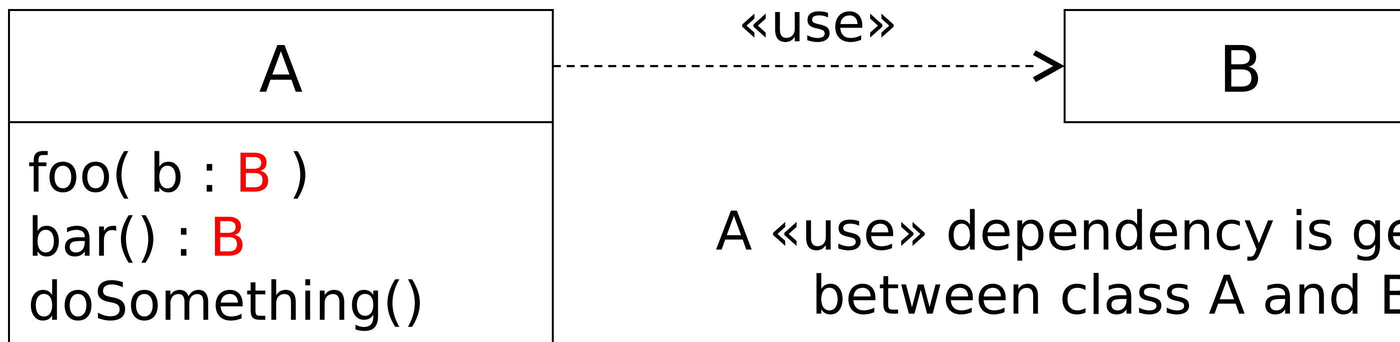
- "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)". In other words, the client *depends* in some way on the supplier
  - Dependency is really a catch-all that is used to model several different types of relationship. We've already seen one type of dependency, the «instantiate» relationship
- Three types of dependency:
  - Usage - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
  - Abstraction - a shift in the level of abstraction. The supplier is more abstract than the client
  - Permission - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

© Clear View Training 2008 v2.5

Usage dependencies

- «use» - the client makes use of the supplier to implement its behaviour
- «call» - the client operation invokes the supplier operation
- «parameter» - the supplier is a parameter of the client operation
- «send» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «instantiate» - the client is an instance of the supplier

«use» - example



A «use» dependency is generated between class A and B when:

1) An operation of class A needs a parameter of class B
2) An operation of class A returns a value of class B
3) An operation of class A uses an object of class B somewhere in its implementation

```
A :: doSomething()
{
  B myB = new B();
  ...
}
```

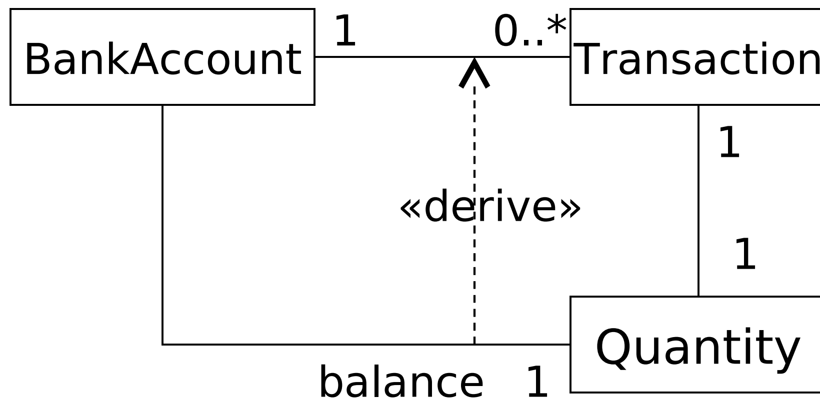
© Clear View Training 2008 v2.5

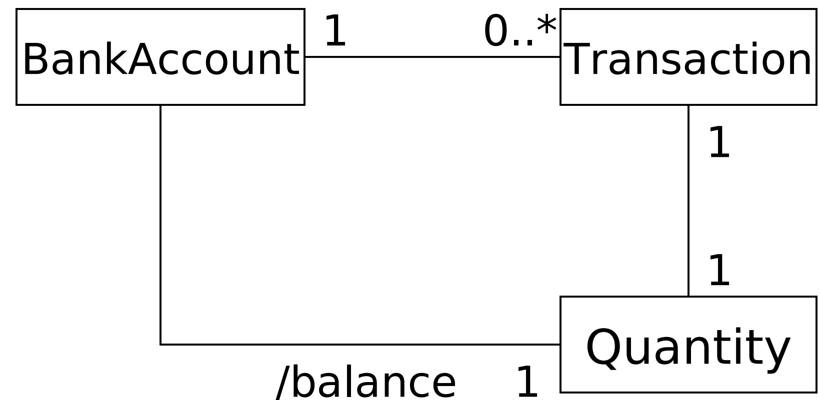
Abstraction dependencies

- «trace» - the client and the supplier represent the same concept but at different points in development
- «substitute» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
- «refine» - the client represents a fuller specification of the supplier
- «derive» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

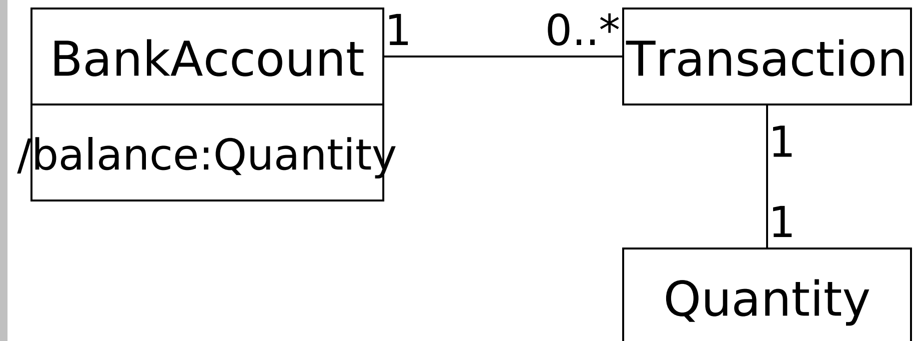
© Clear View Training 2008 v2.5

«derive» - example





This example shows three possible ways to express a «derive» dependency



Permission dependencies

- «access»
  - The public contents of the supplier package are added as private elements to the namespace of the client package
- «import»
  - The public contents of the supplier package are added as public elements to the namespace of the client package
- «permit»
  - The client element has access to the supplier element despite the declared visibility of the supplier

Summary

- Dependency
  - The weakest type of association
  - A catch-all
- There are three types of dependency:
  - Usage
  - Abstraction
  - Permission

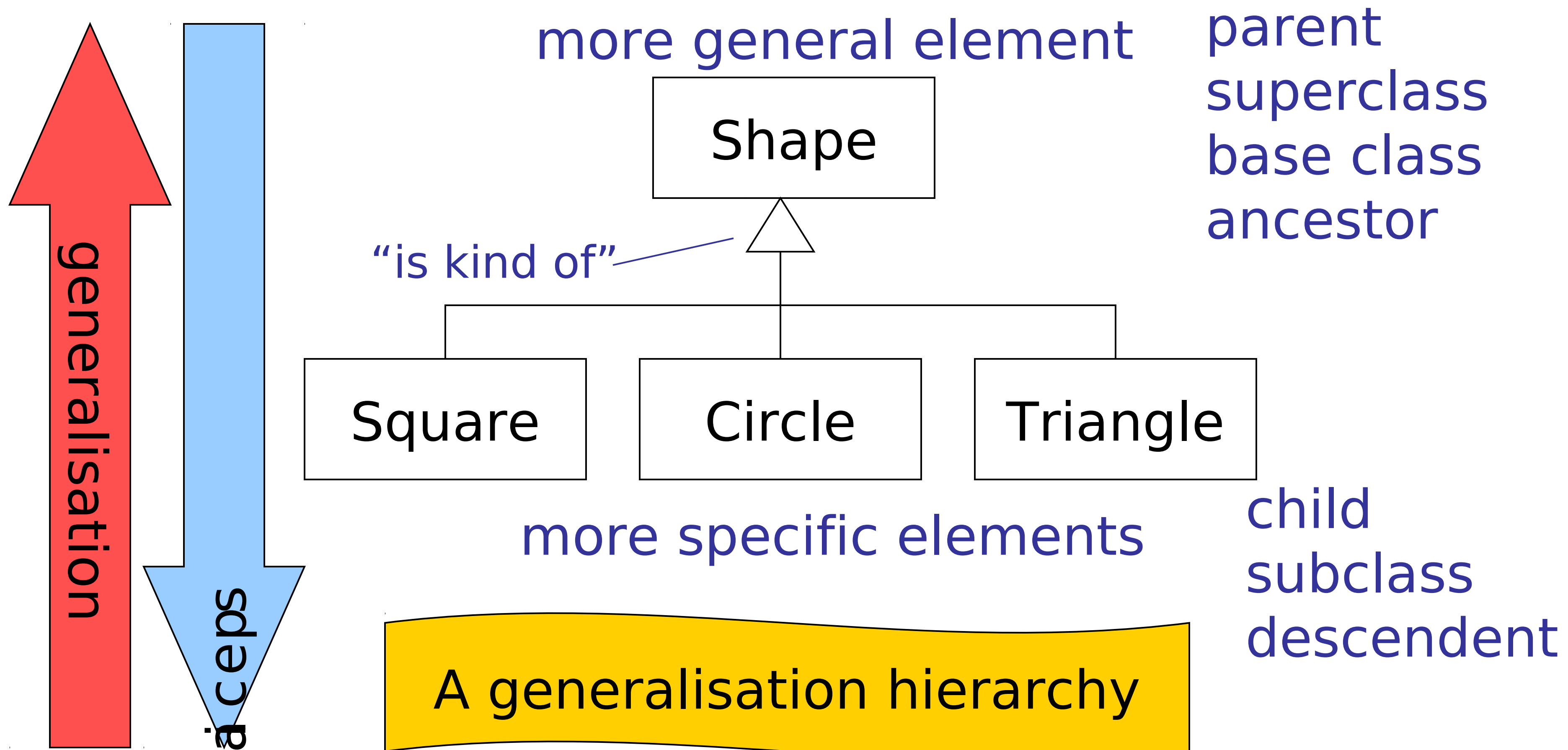
Analysis – inheritance and polymorphism

Generalisation

- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information
- An instance of the more specific element may be used where an instance of the more general element is expected

© Clear View Training 2008 v2.5

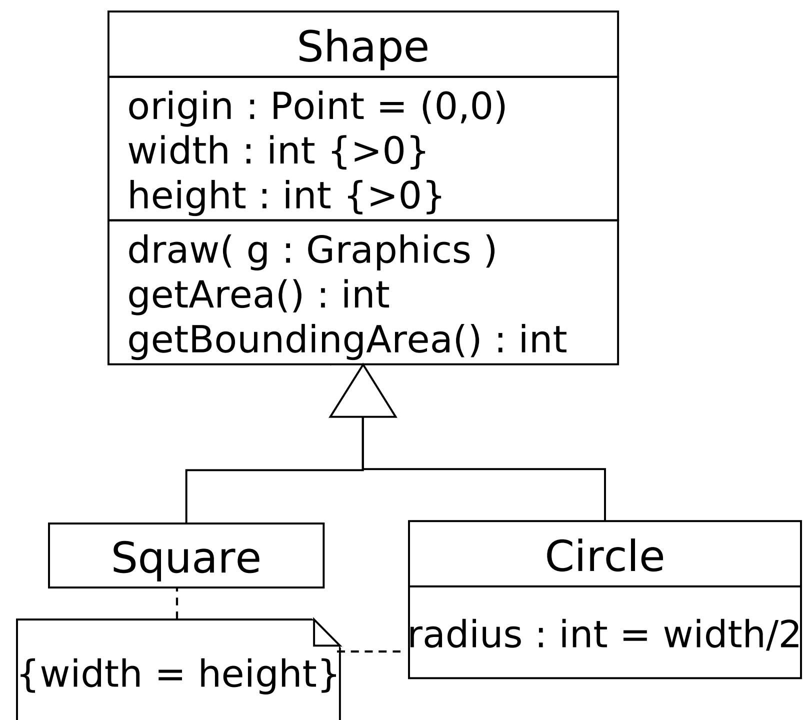
Example: class generalisation

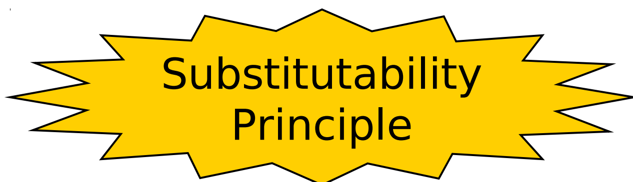


© Clear View Training 2008 v2.5

Class inheritance

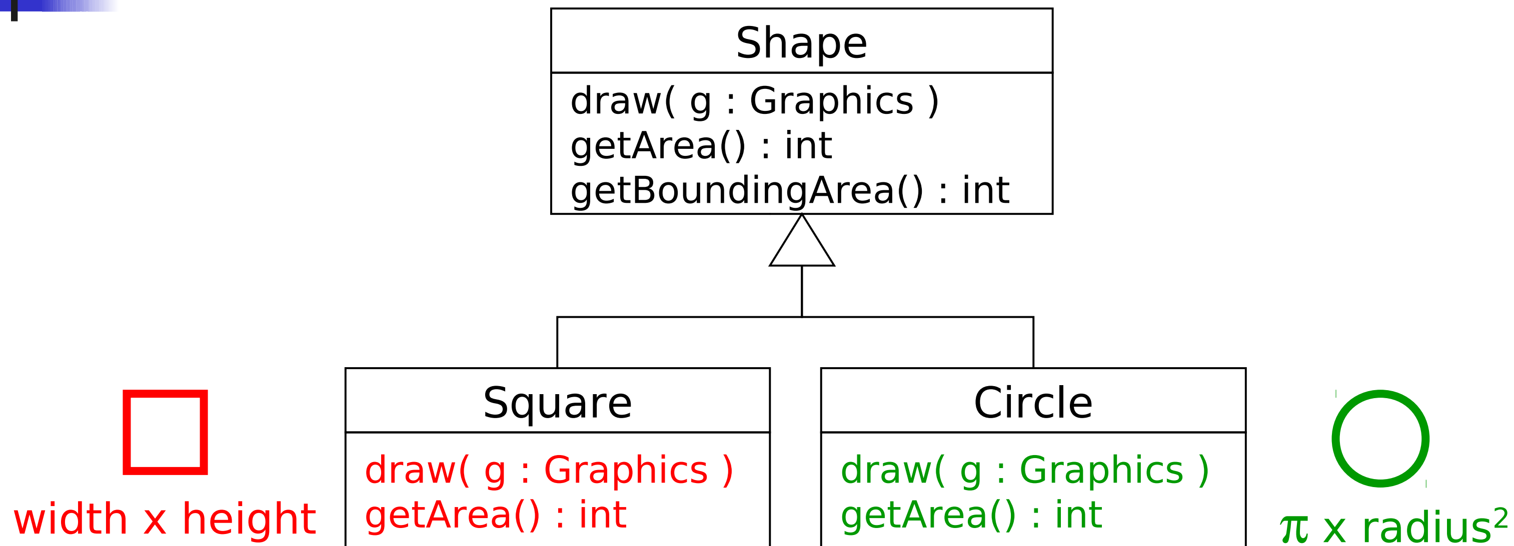
- Subclasses inherit *all* features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- Subclasses can add new features
- Subclasses can override superclass operations
- We can use a subclass instance *anywhere* a superclass instance is expected





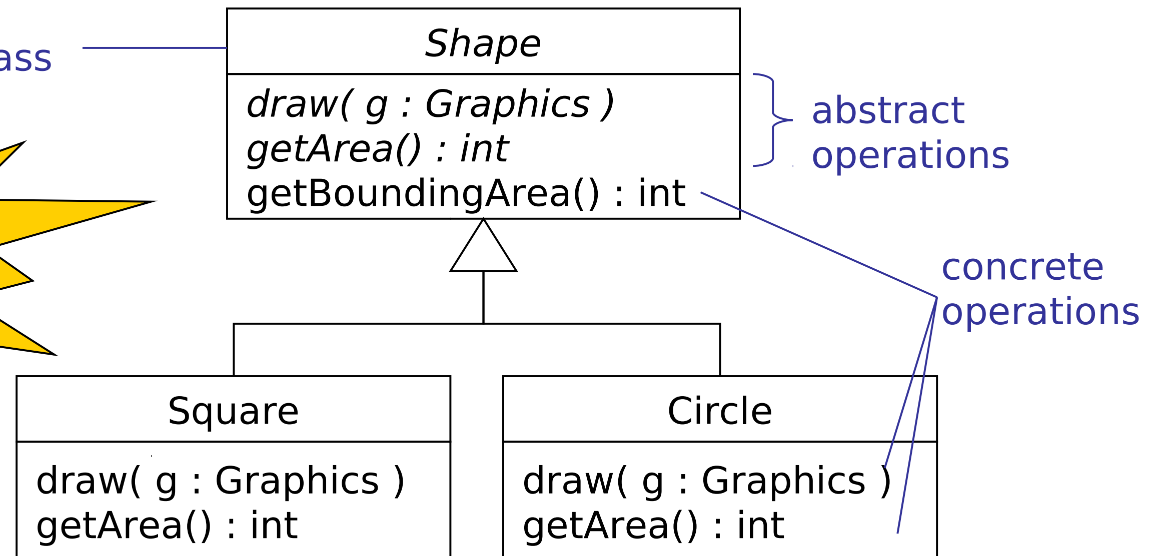
But what's wrong with these subclasses

Overriding



- Subclasses often need to *override* superclass behaviour
- To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters
  - The names of the parameters don’t count as part of the signature

Abstract operations & classes

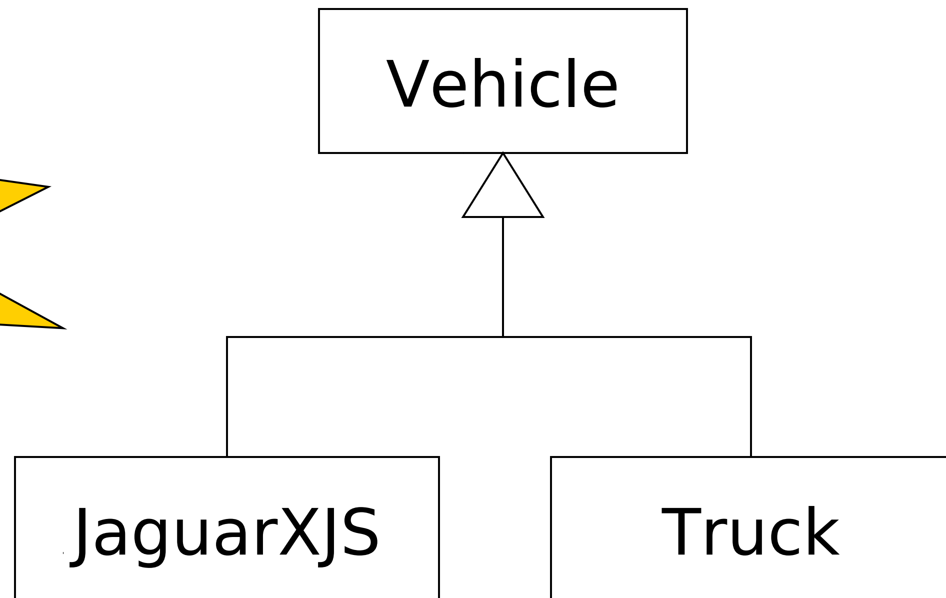


- We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : int* because we don't know how to draw or calculate the area for a "shape"!
- Operations that lack an implementation are *abstract operations*
- A class with any abstract operations *can't* be instantiated and is therefore an *abstract class*

© Clear View Training 2008 v2.5

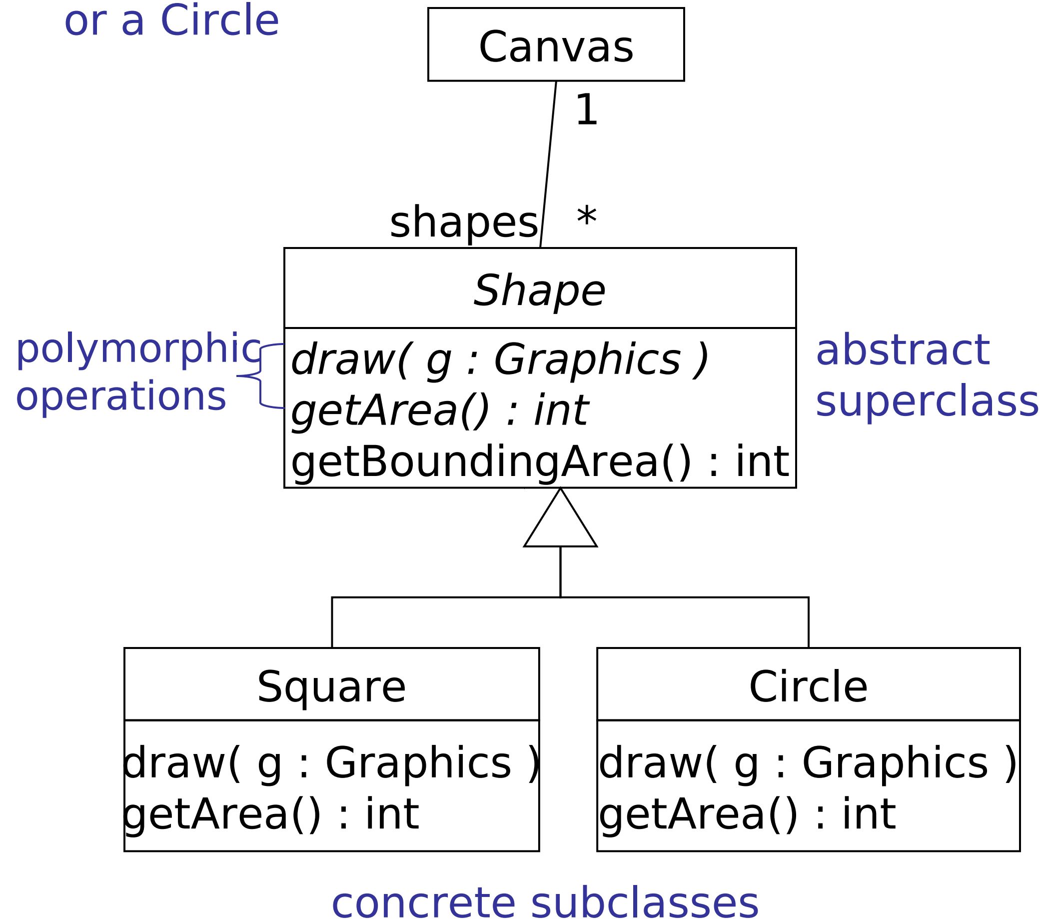
Exercise

what's wrong with this model?



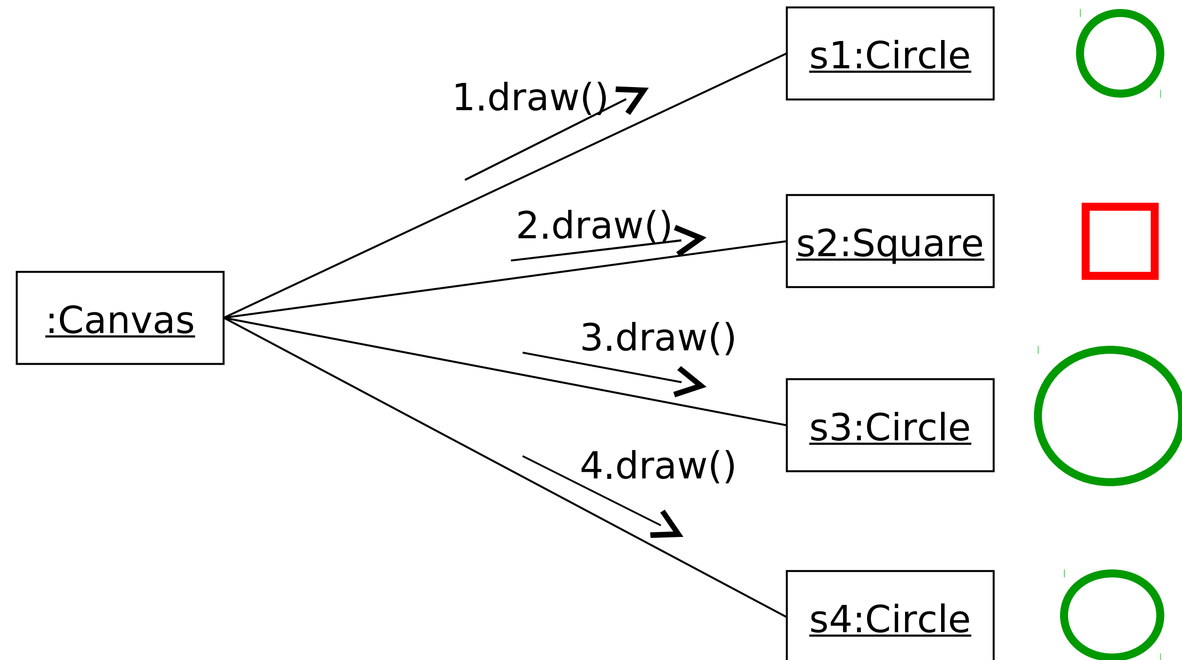
Polymorphism

- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*
- All concrete subclasses of Shape *must* provide concrete draw() and getArea() operations because they are abstract in the superclass
  - For draw() and getArea() we can treat all subclasses of Shape in a similar way - we have defined a contract for Shape subclasses

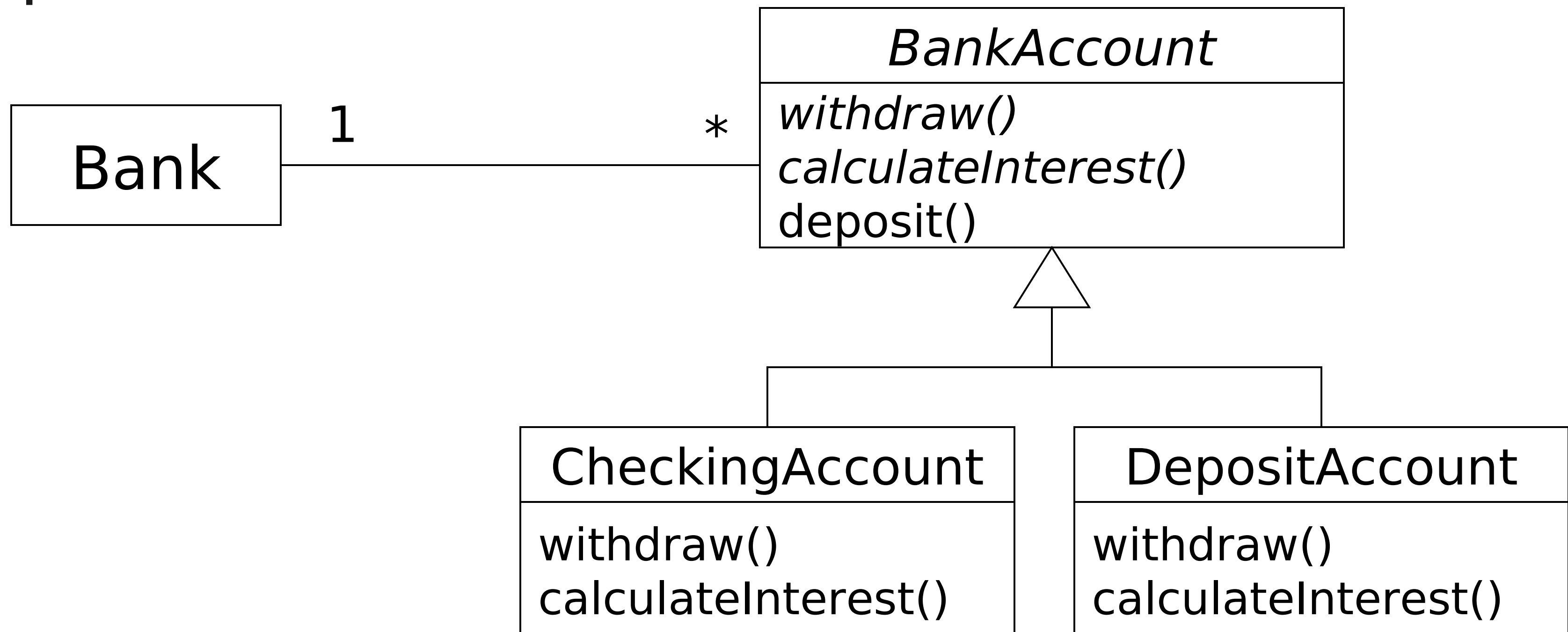


What happens?

- Each class of object has its own implementation of the draw() operation
- On receipt of the draw() message, each object invokes the draw() operation specified by its class
- We can say that each object "decides" how to interpret the draw() message based on its class

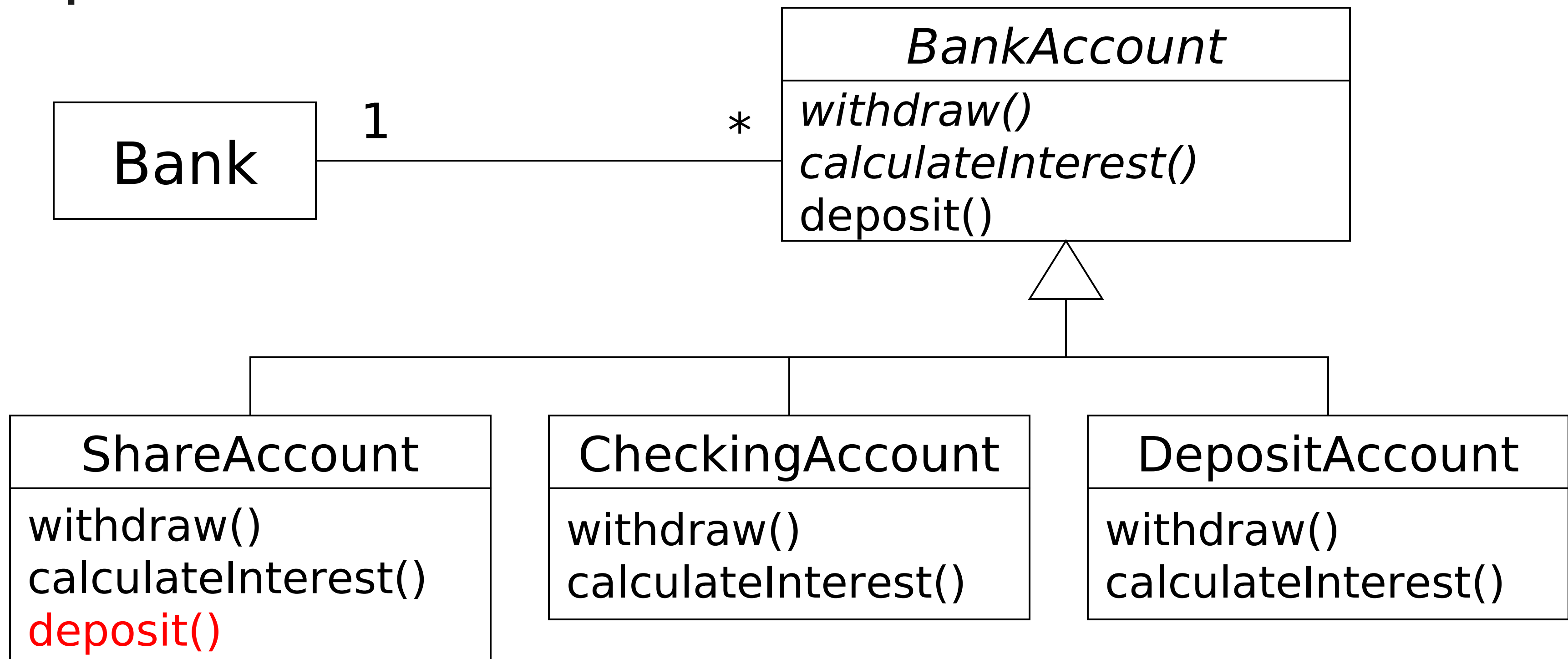


BankAccount example



© Clear View Training 2008 v2.5

BankAccount example



- We have overridden the deposit() operation even though it is *not* abstract. This is perfectly legal, and quite common, although it is generally considered to be bad style and should be avoided if possible

© Clear View Training 2008 v2.5

Summary

- Subclasses:
  - inherit *all* features from their parents including constraints and relationships
  - may add *new* features, constraints and relationships
  - may *override* superclass operations
- A class that can’t be instantiated is an abstract class