

Chapter 16

UML Class Diagrams

SuperclassFoo
or
SuperClassFoo { abstract }

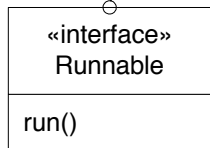
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute

+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

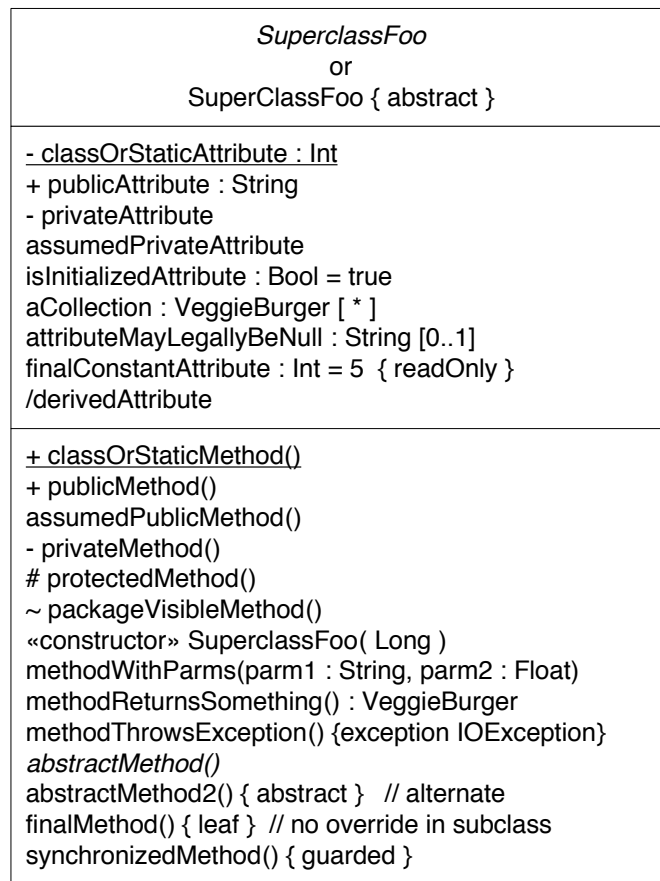
3 common compartments

1. classifier name
2. attributes
3. operations

an interface shown with a keyword

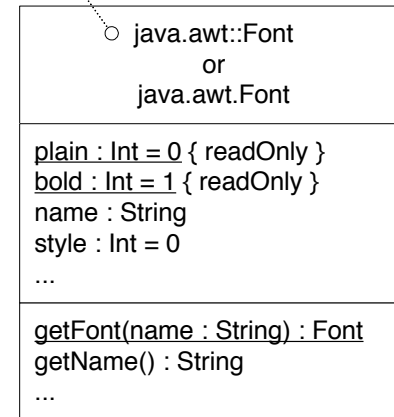


interface implementation and subclassing

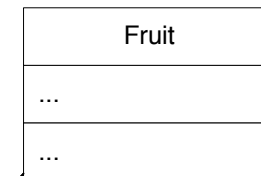


officially in UML, the top format is used to distinguish the package name from the class name

unofficially, the second alternative is common



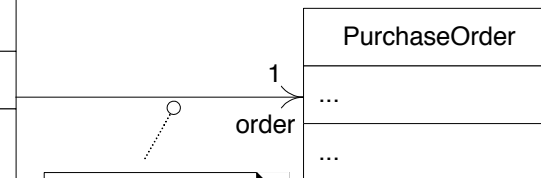
dependency



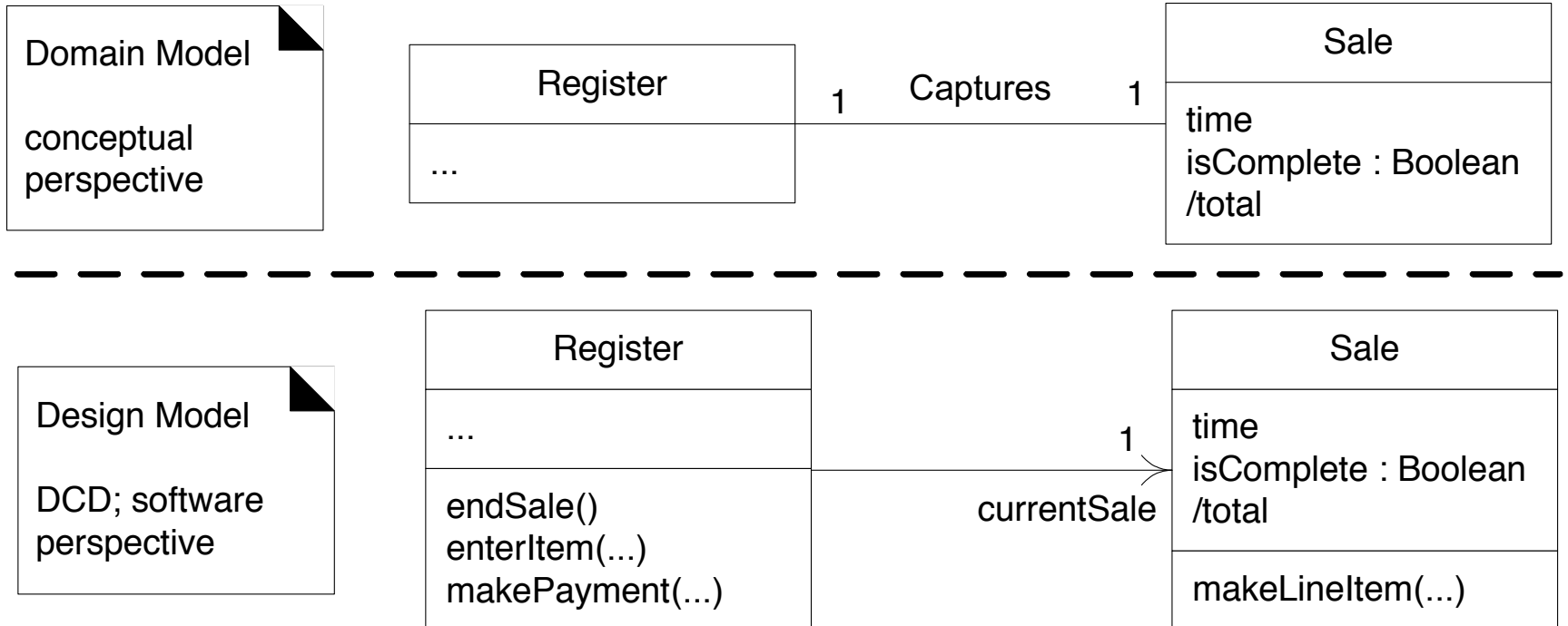
- ellipsis "... " means there may be elements, but not shown

- a *blank* compartment officially means "unknown" but as a convention will be used to mean "no members"

association with multiplicities



Domain Model vs. Design Class Diagram (DCD)

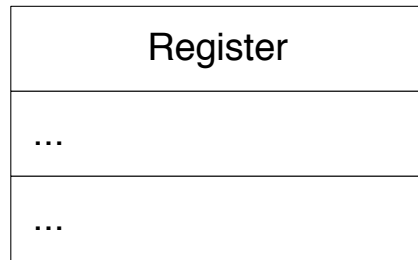


Attribute text vs. association lines

using the attribute text notation to indicate Register has a reference to one Sale instance

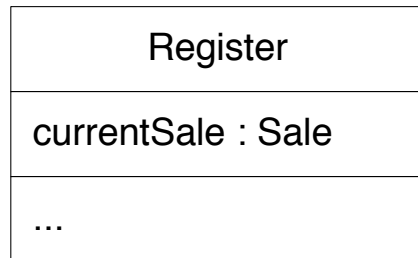


OBSERVE: this style *visually* emphasizes the connection between these classes



using the association notation to indicate Register has a reference to one Sale instance

thorough and unambiguous, but some people dislike the possible redundancy



Full format attribute text notation

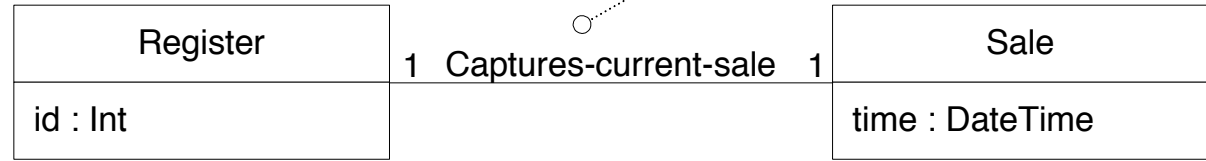
visibility name : type multiplicity = default {property string}

<i>SuperclassFoo</i> or SuperClassFoo { abstract }
<u>- classOrStaticAttribute</u> : Int + publicAttribute : String - privateAttribute assumedPrivateAttribute isInitializedAttribute : Bool = true aCollection : VeggieBurger [*] attributeMayLegallyBeNull : String [0..1] finalConstantAttribute : Int = 5 { readOnly } /derivedAttribute
+ <u>classOrStaticMethod()</u> + publicMethod() assumedPublicMethod() - privateMethod() # protectedMethod() ~ packageVisibleMethod() «constructor» SuperclassFoo(Long) methodWithParms(parm1 : String, parm2 : Float) methodReturnsSomething() : VeggieBurger methodThrowsException() {exception IOException} <i>abstractMethod()</i> abstractMethod2() { abstract } // alternate finalMethod() { leaf } // no override in subclass synchronizedMethod() { guarded }

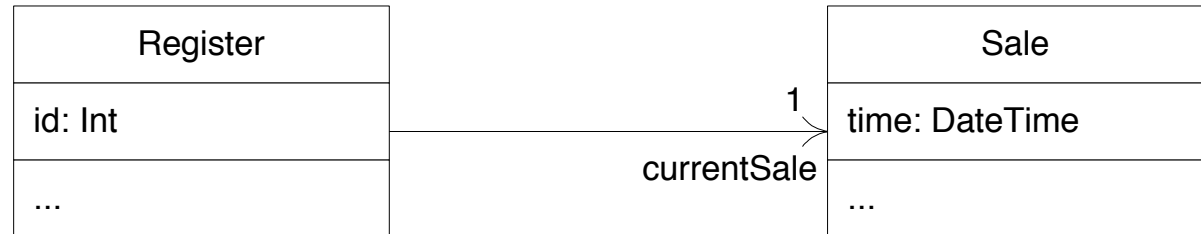
UML Class Diagram Conventions: Domain Models vs. DCDs

the association *name*, common when drawing a domain model, is often excluded (though still legal) when using class diagrams for a software perspective in a DCD

UP Domain Model
conceptual perspective

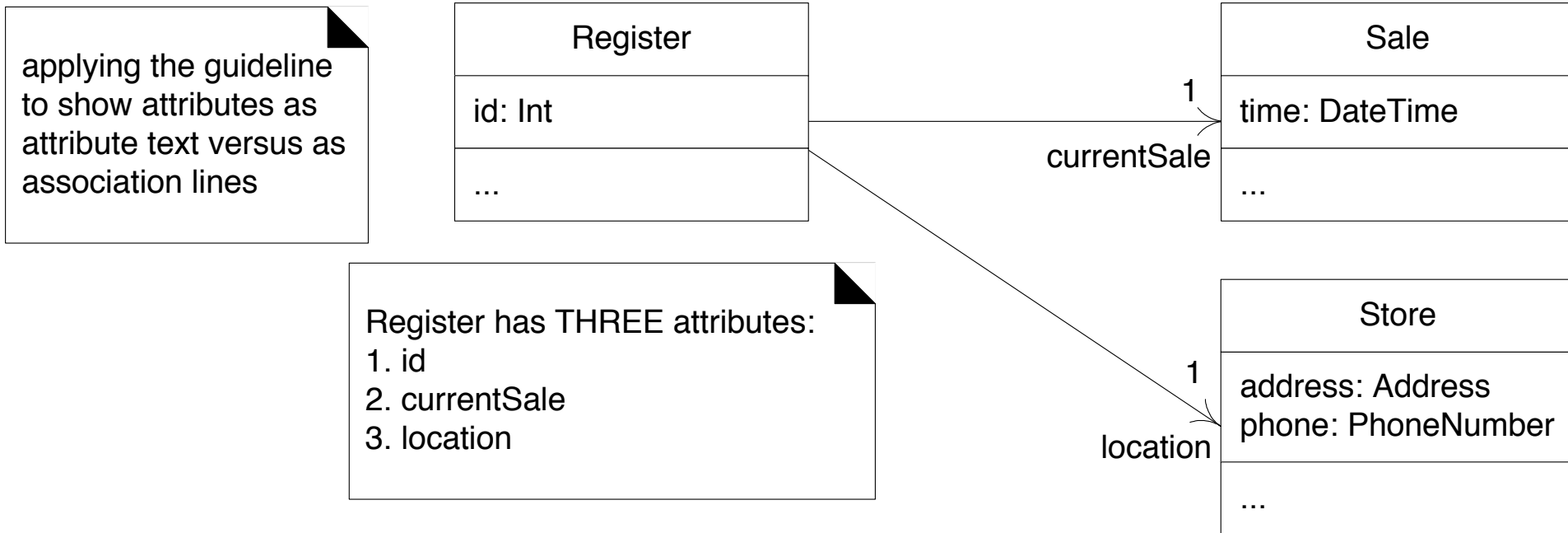


UP Design Model
DCD
software perspective



- Arrows:
 - Used in DCDs. Register has currentSale attribute
 - Not used in domain models.
- DCDs have multiplicities only at target end
- No role name for DCDs
- No association name for DCDs

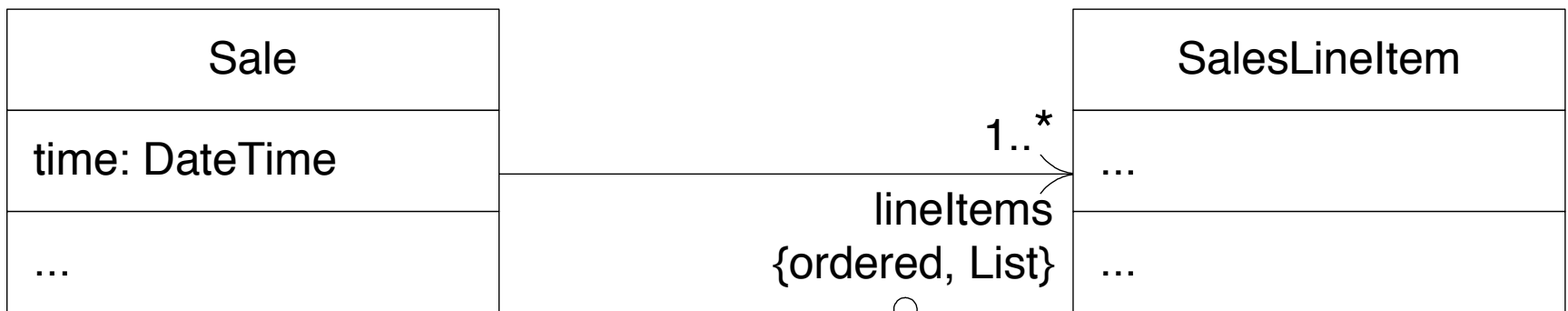
Attribute text vs. association lines



- Guideline: Attributes should be (simple, primitive) data types
 - Boolean, Date, Number (Integer, Real), String (Text), Time, Phone Number, ID Number, Postal Code, ...
- Non-data type relationships (i.e., class relationships) should be expressed using associations, not attributes.
- Notational difference only.
 - As far as coding is concerned, they mean the same thing.

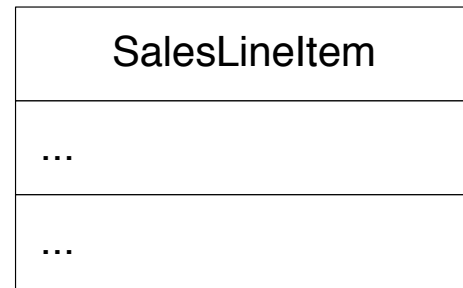
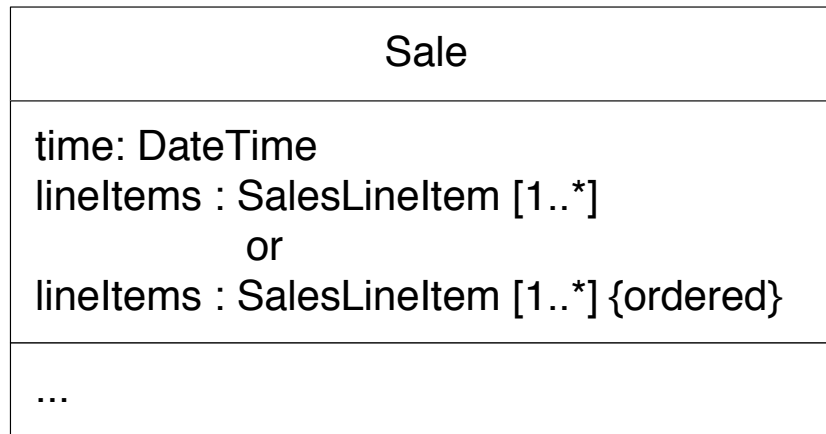
Notation at Association Ends

- Rolename: Association end name
- Multiplicity
- Property string

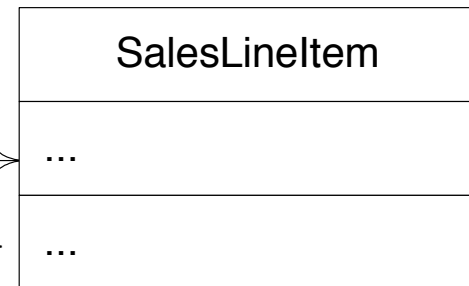
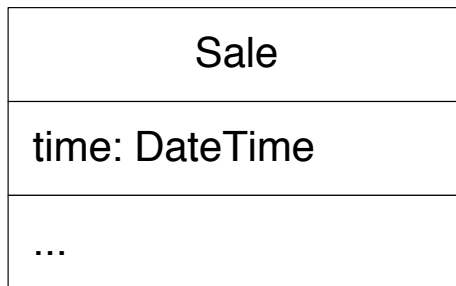


notice that an association end can optionally also have a property string such as `{ordered, List}`

Collection Attributes



Two ways to show a collection attribute



1..*
lineItems
{ordered, List}

notice that an association end can optionally also have a property string such as {ordered, List}

Note Symbols: Notes, Comments, Constraints and Method Bodies

«method»

// pseudo-code or a specific language is OK

```
public void enterItem( id, qty )
```

```
{
```

```
    ProductDescription desc = catalog.getProductDescription(id);
```

```
    sale.makeLineItem(desc, qty);
```

```
}
```

Register

...

endSale()

○ enterItem(id, qty)

makeNewSale()

makePayment(cashTendered)

Operations (Methods)

- Full official format
visibility name (parameter-list) : return-type {property-string}
- Can use programming language notation if more convenient and understandable
- Operations assumed public unless indicated otherwise
- Property string: Arbitrary info
 - Example: Exceptions that may be raised
- Operation vs. method
 - Operation: A declaration, the specification of a method
 - Method: Implementation of an operation
- Access operations (set & get methods) usually excluded from DCD diagrams

UML Keywords

- <<actor>>
- <<interface>>
- {abstract}
- {ordered}

Generalization, Abstract Classes, Abstract Operations

- final class shown by {leaf}

3 common compartments

1. classifier name

2. attributes

3. operations

an interface shown with a keyword

«interface» Runnable

run()

«interface» Runnable

run()

interface implementation and subclassing

SuperclassFoo
or
SuperClassFoo { abstract }

- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute

+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

SubclassFoo

...

run()

...

officially in UML, the top format is used to distinguish the package name from the class name

unofficially, the second alternative is common

java.awt::Font
or
java.awt.Font

plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...

getFont(name : String) : Font
getName() : String
...

dependency

Fruit

...

...

PurchaseOrder

...

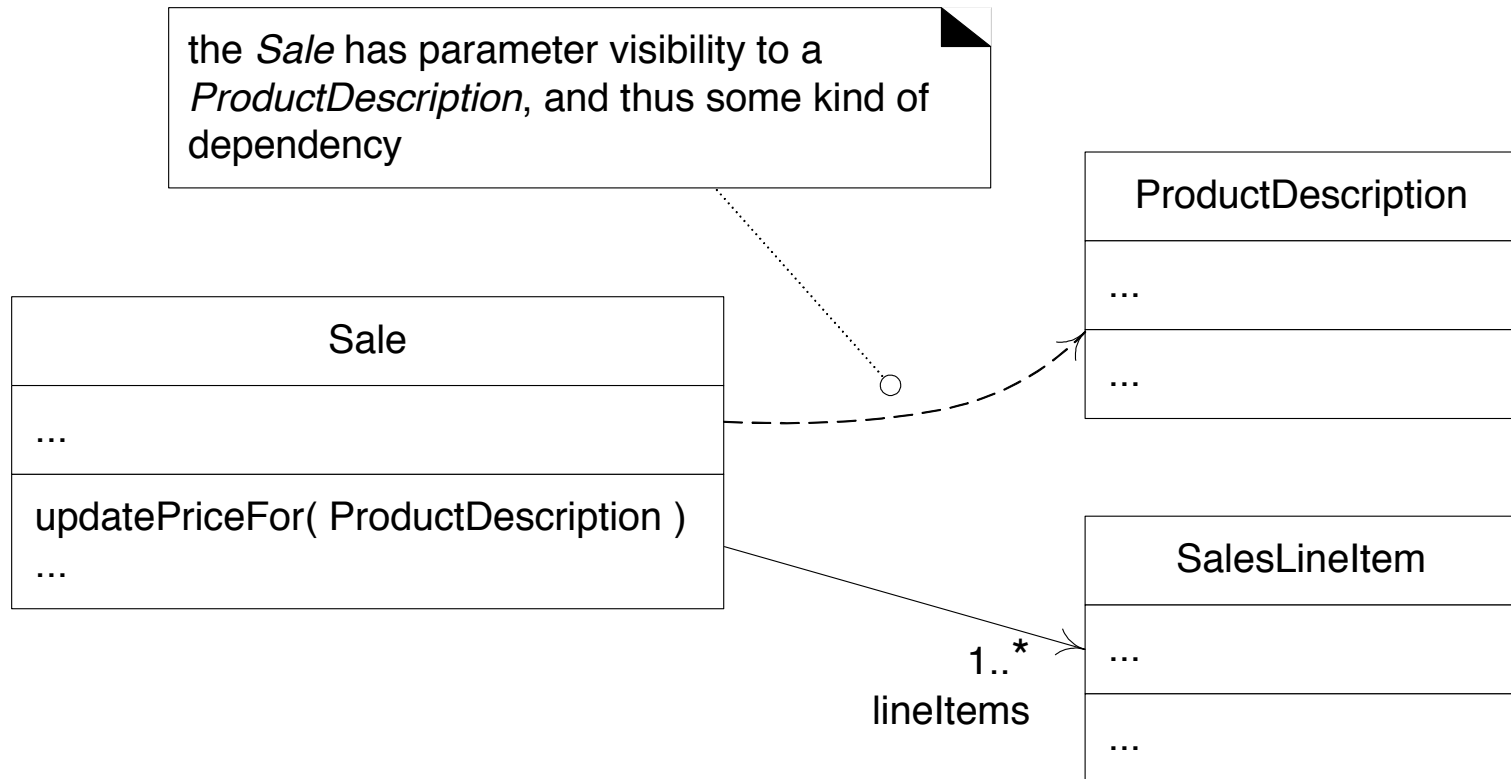
...

association with multiplicities

- ellipsis ... means there may be elements, but not shown

- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Dependency

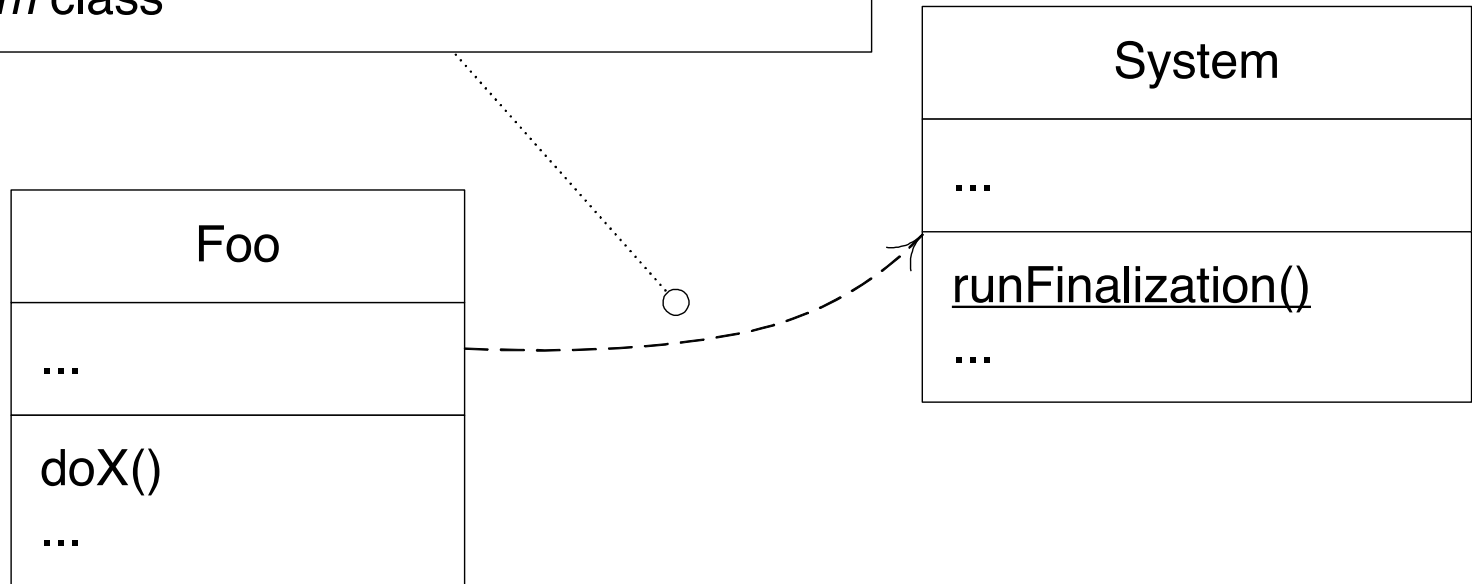


- Reserve its use for purposes other than associations/attributes
 - Examples: Global, parameter variable, local variable and static method dependencies

```
public class Sale {
    public void updatePriceFor(ProductDescription description) {
        Money basePrice = description.getPrice();
    }
}
```

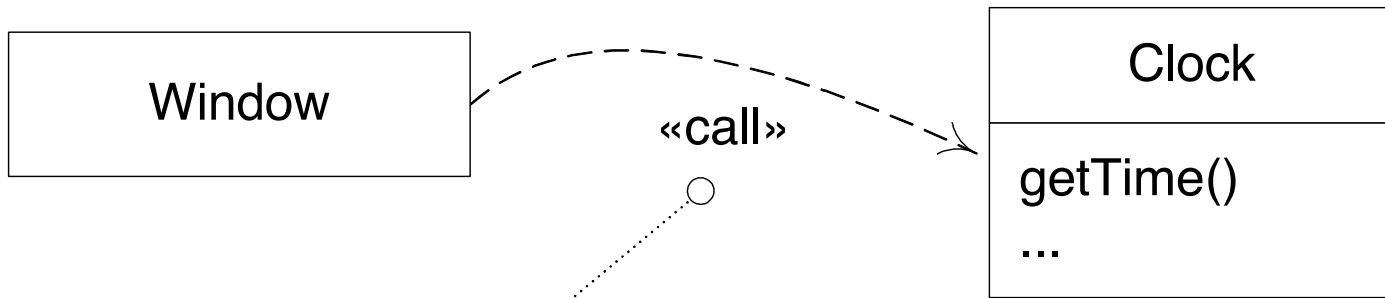
Dependency Example

the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class

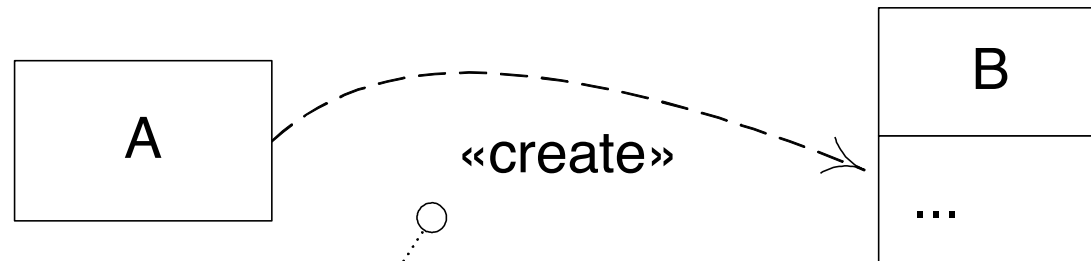


```
public class Foo {  
    public void doX() {  
        System.runFinalization();  
        // ...  
    }  
}
```


Dependency Labels

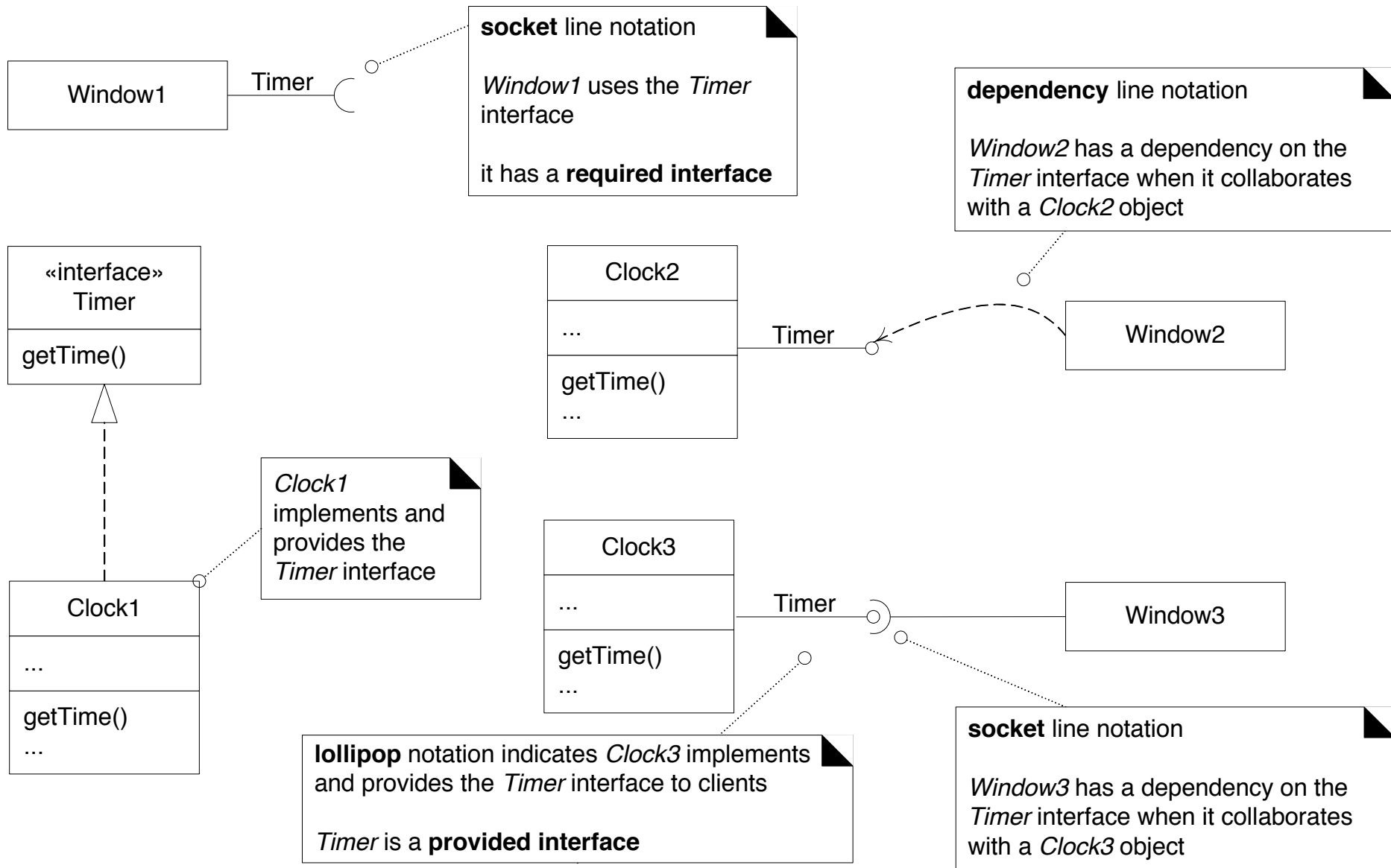


a dependency on calling on operations of the operations of a *Clock*

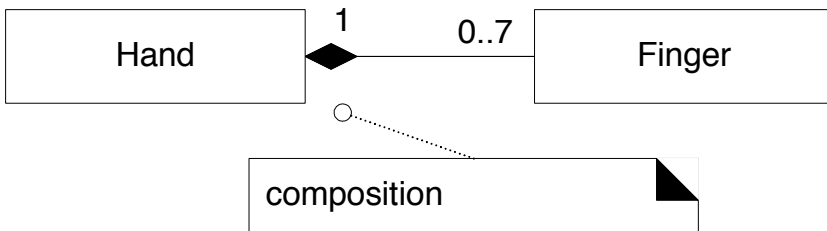


a dependency that A objects create B objects

Interface Notations



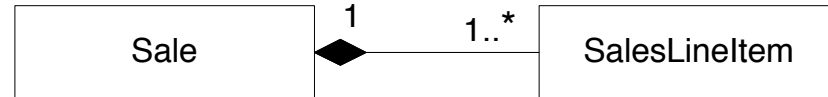
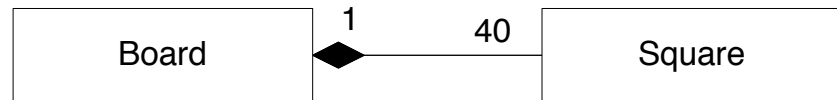
Composition



composition means

-a part instance (*Square*) can only be part of one composite (*Board*) at a time

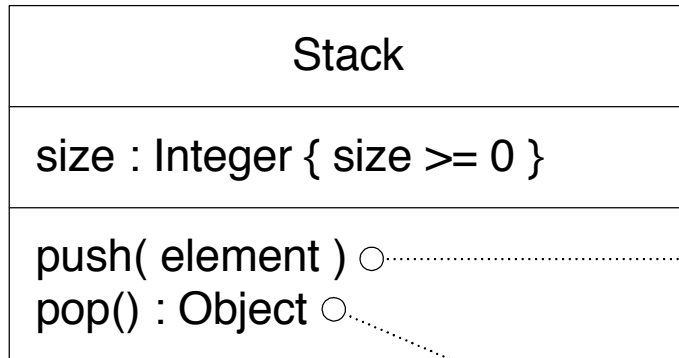
-the composite has sole responsibility for management of its parts, especially creation and deletion



- Composition: A strong kind of association
 - An instance of *Square* (“the part”) belongs to only one *Board* (“the composite instance”) at a given time
 - A *Square* object must always belong to a *Board*
 - No free-floating parts
 - The *Board* (composite) is responsible for creating and deleting the parts
 - If composite is destroyed, the parts must also be destroyed or attached to another composite.

Constraints Notation

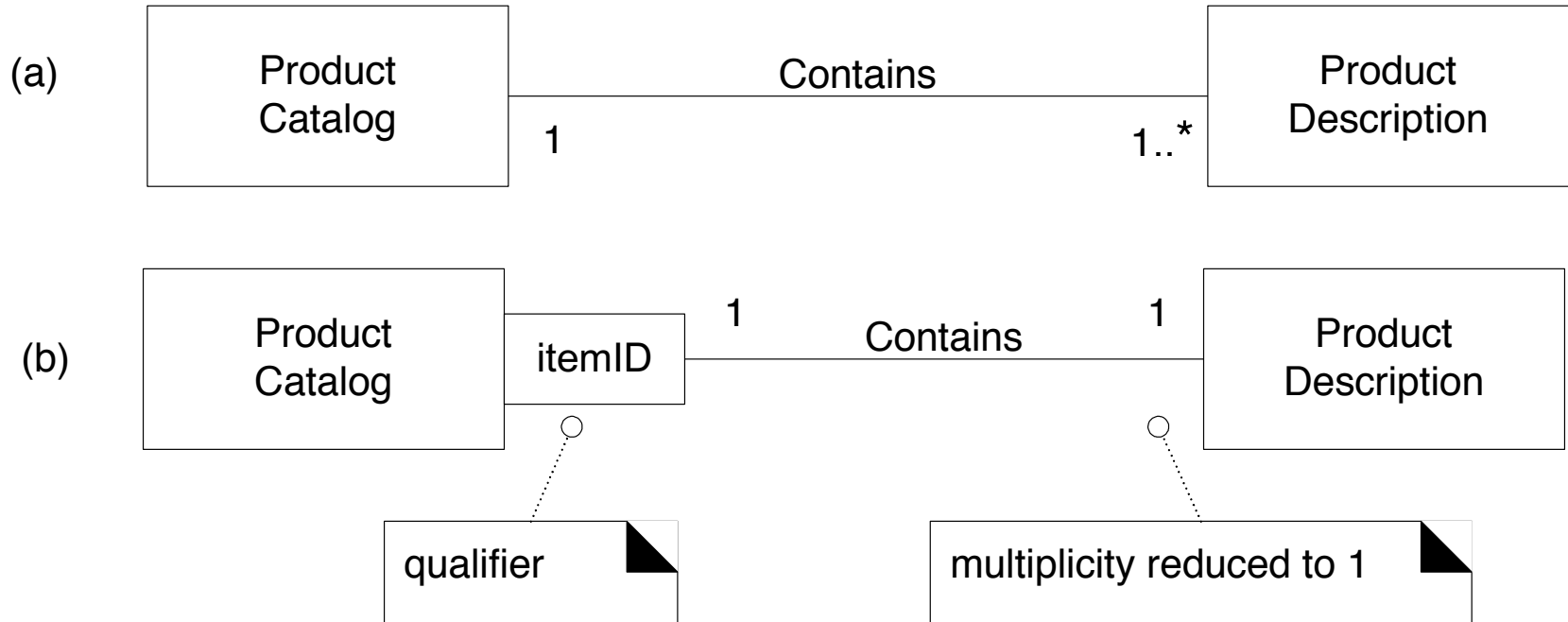
three ways to show UML constraints



{ post condition: new size = old size + 1 }

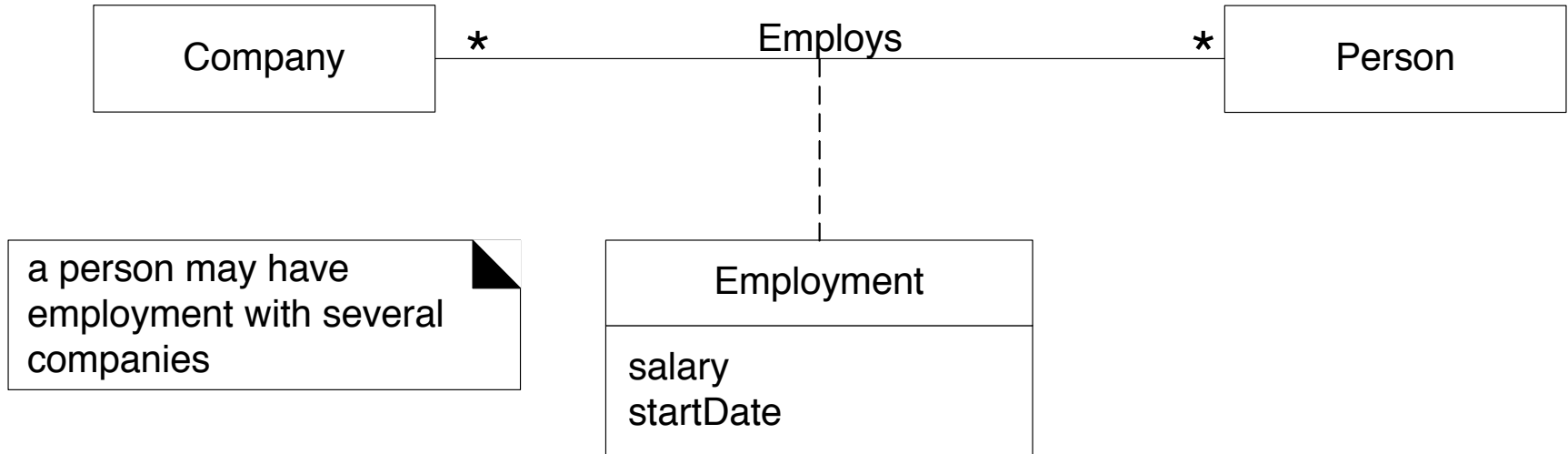
{
post condition: new size = old size - 1
}

Qualified Associations



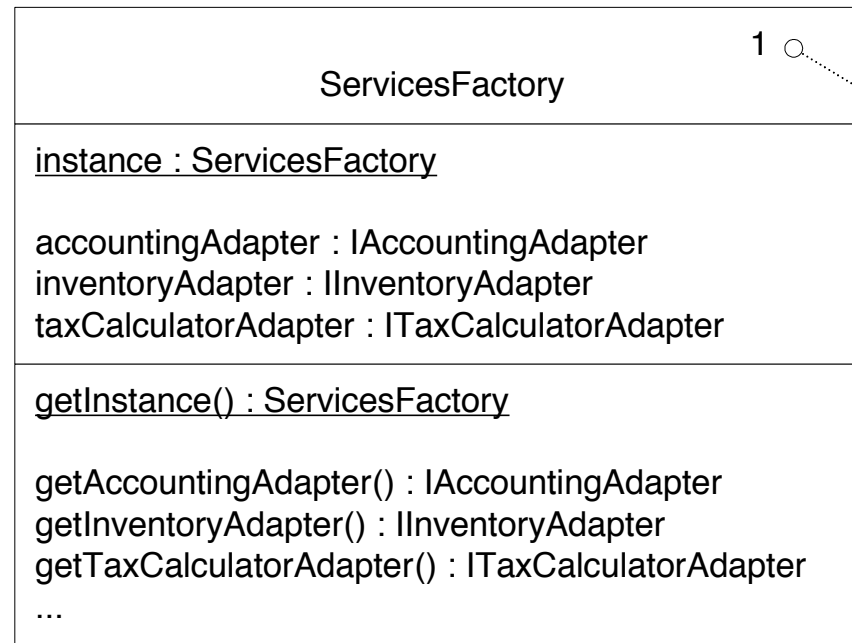
- Qualifier: Selects an object from a larger set of related objects
 - Based on the qualifier key
 - Suggests looking things up by key
 - e.g. in a hashtable
- Multiplicity on the association is changed from many to one.

Association Classes



- The association itself is a class
 - Why: A lot of information needs to be remembered/stored about the association
 - attributes
 - operations

Singleton Classes



UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member

UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

Template Classes and Interfaces

```
public class Board {  
    private List<Square> squares = new ArrayList<Square>();  
    // ...  
}
```

parameterized or template
interfaces and classes

K is a **template parameter**

«interface»
List

clear()
...

K



ArrayList

elements : T[*]
...

clear()
...

T

the attribute type may be expressed in
official UML, with the template binding
syntax requiring an arrow
or
in another language, such as Java

Board

squares : List<K>Square>
or
squares : List<Square>
...

anonymous class with
template binding complete

ArrayList<T>Square>

clear()
...

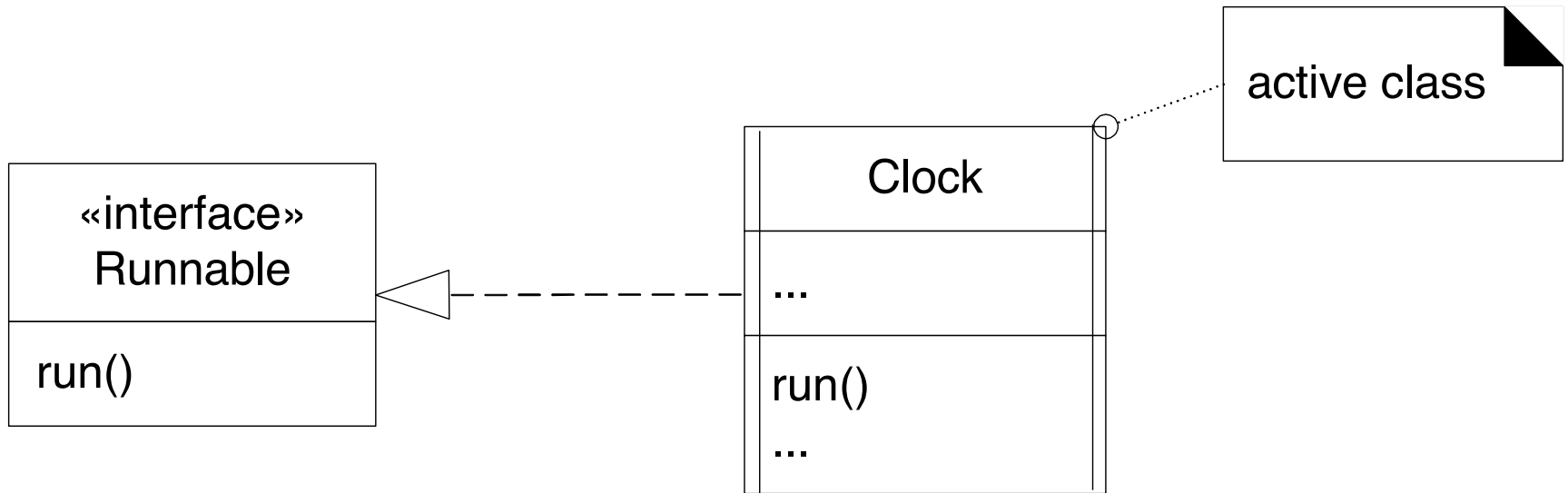
for example, the *elements* attribute is an
array of type T, parameterized and bound
before actual use.

there is a chance the UML 2 “arrow” symbol will
eventually be replaced with something else e.g., “=“

User-Defined Compartments

.DataAccessObject
id : Int ...
doX() ...
exceptions thrown DatabaseException IOException
responsibilities serialize and write objects read and deserialize objects ...

Active Classes



- Active object: Runs on and controls its own thread of execution

Interaction diagrams help build and debug class Diagrams

