

CSSE2003

Software Engineering Studio

Semester 2, 2009

18: Mapping Design to Code

Lecture Summary

- ◆ Design class diagrams and code
- ◆ Sequence diagrams and code

Design and Code - 1

- ◆ OOA/D and OO programming provides an end-to-end roadmap from requirements analysis through to code (though the road may not be smooth).
- ◆ Implementation is about transforming a design model into code.
- ◆ Design class diagrams and sequence diagrams are the inputs into the coding phase.

Design and Code -2

- ◆ Design is a creative, decision-making activity.
- ◆ Typically, implementation is a more routine activity (the decisions involved are smaller and more localised).
- ◆ A design may need to change during implementation and afterward.
 - plan for change during design
 - use refactoring techniques when change is needed
- ◆ It is important to maintain traceability between design and code.

A code development process

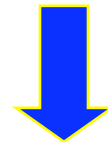
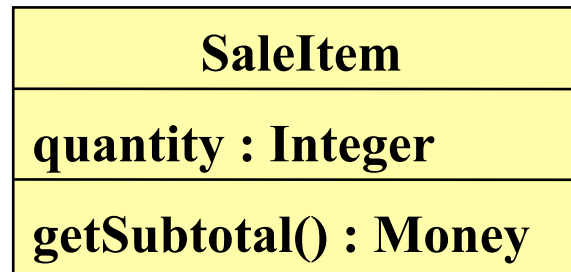
- ◆ Use design class diagrams to partially define classes and their interfaces. These will identify class attributes and method prototypes.
- ◆ Use sequence diagrams to partially define methods.
- ◆ Complete class definitions.

From design classes to code classes

- ◆ A design class diagram gives a class name, superclasses, method signatures and class attributes. This is enough to define a basic Java class.
- ◆ Design class attributes and methods become Java class members.

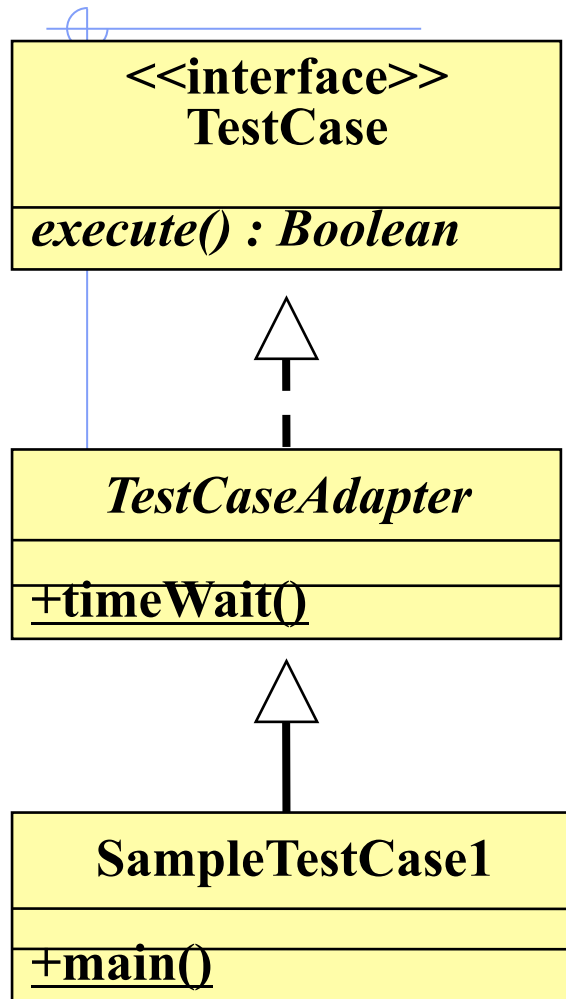
Example -1

- ♦ Make a constructor explicit



```
class SaleItem {...  
    int quantity;  
    SaleItem( ...) {... }  
    Money getSubtotal () {...}  
    ... }
```

Example -2



```
public interface TestCase {...
    public abstract boolean execute ();
    ... }
```

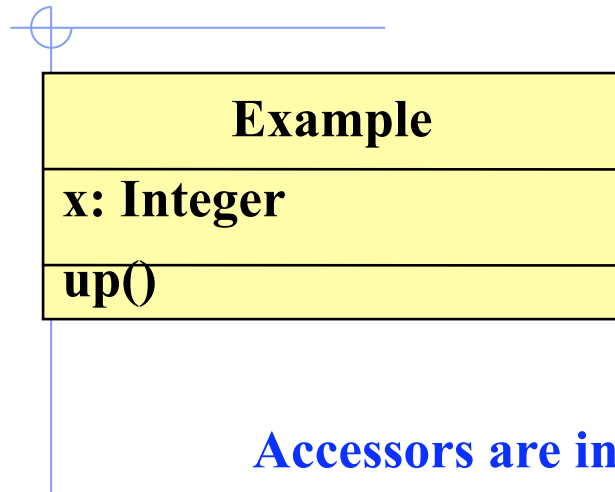
```
public abstract class TestCaseAdapter
    implements TestCase {...
    public static void timeWait ()
    {...} ...
}
```

```
public class SampleTestCase1 extends
    TestCaseAdapter {...
    public static void main () {...}
    ...}
```

Code level attributes

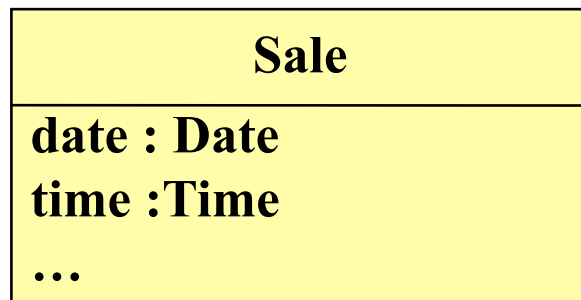
- ◆ Design class attributes typically become instance variables.
- ◆ Get and set methods and constructors, left implicit in the design class model, need to be coded.
- ◆ Attributes and methods introduced during coding (i.e. not in the design model) should be private (excepting getter/setter/constructor).

Example



```
class Example {...  
    private int x;  
    int getX() { ...};  
    void setX(int x) {...};  
    void up() { ... }; ...}
```

Accessors are introduced



```
class Sale {  
    private Date dateTime =  
        new Date(); ...  
}
```

java.util.Date combines date and time
(overloading of type in design and code models)

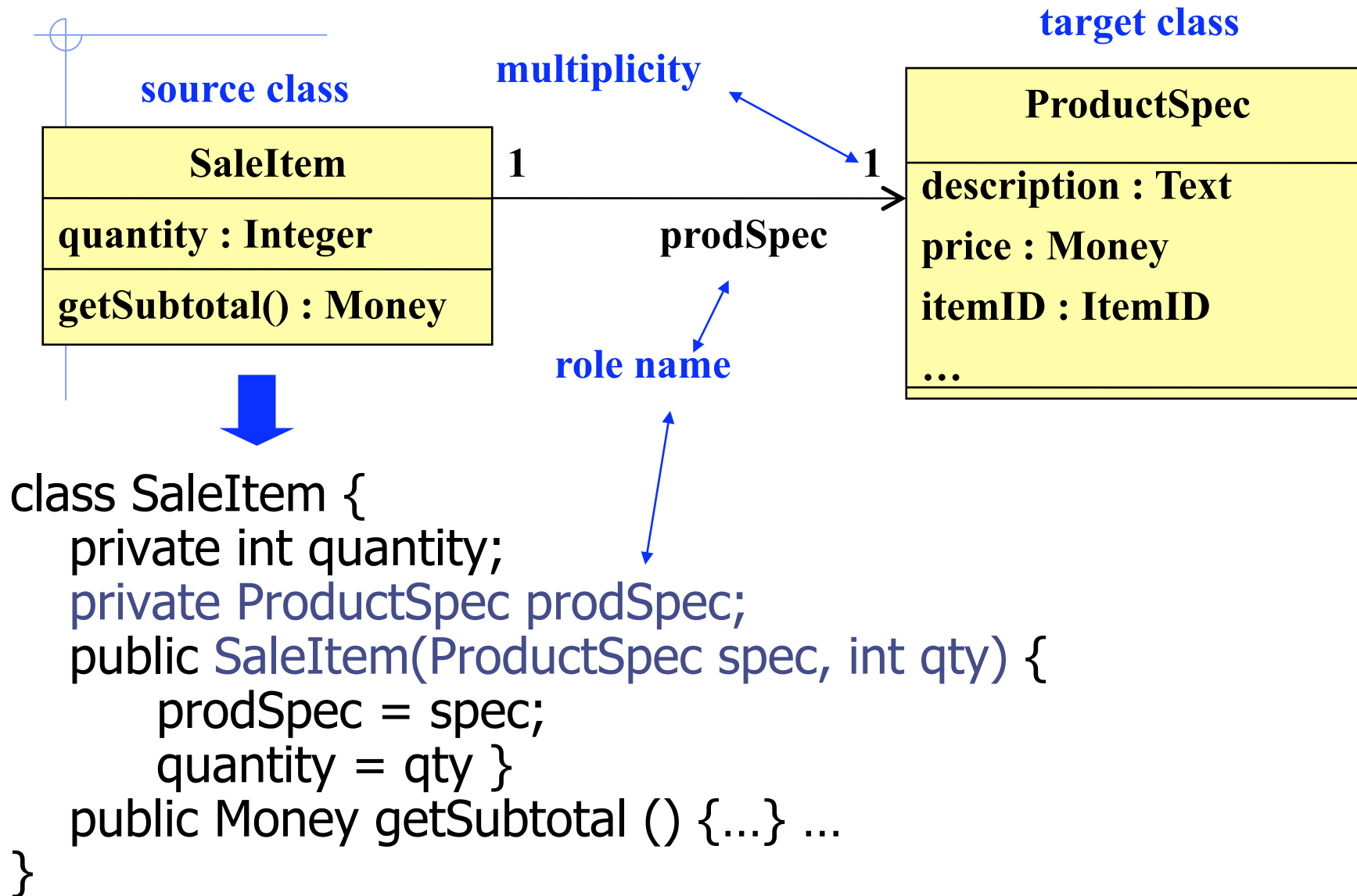
Coding associations - 1

- ◆ An association is coded as an attribute of reference type (i.e., an attribute that references an object).
- ◆ The navigability of a unidirectional association determines the 'ownership' of the attribute.
- ◆ Identify a role name in an association with an attribute name in the class code.
- ◆ An association relating a multi-object requires a choice of a container class.

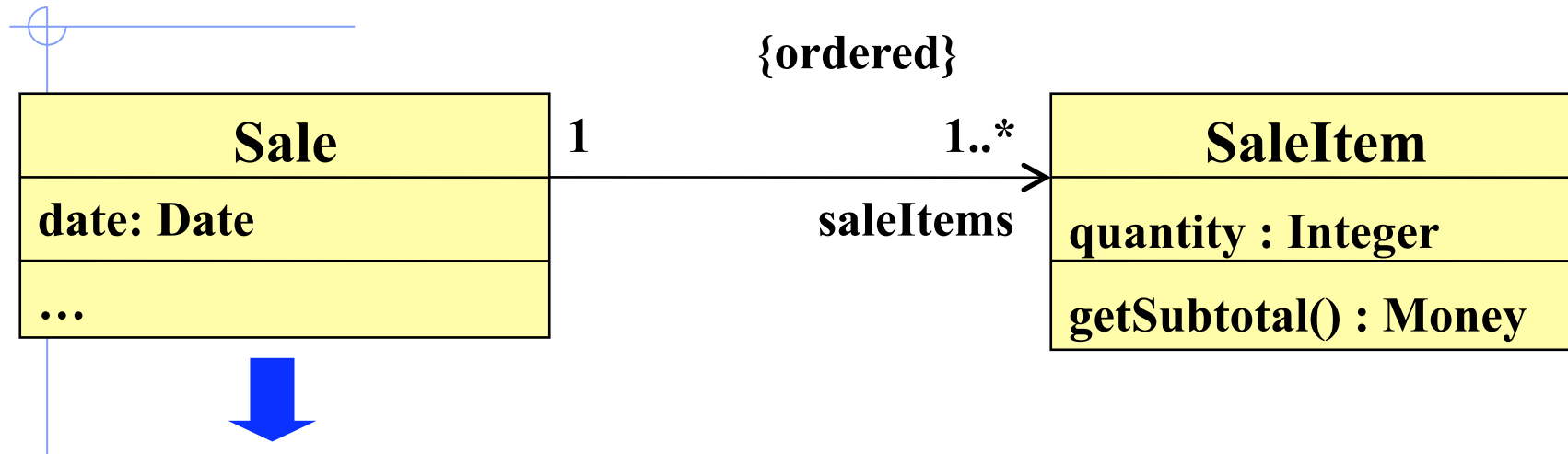
Coding associations - 2

- ◆ For a unidirectional association with target multiplicity 1, to the source class add an attribute that references the target class.
- ◆ For a unidirectional association with target multiplicity > 1 , to the source class add an attribute that references a **container class**, containing instances of the target class.

Example - 1



Example - 2



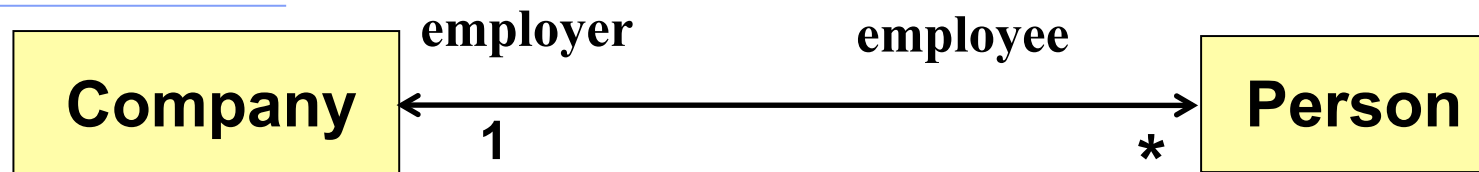
public class Sale { ...
private Date dateTime = new Date();
private List<SaleItem> saleItems =
new LinkedList<SaleItem>(); ... }

- ◆ Class Sale defines an attribute saleItems that is an ordered list of SaleItem objects

Bidirectional association

- ◆ Add an attribute to both sides of the association.
- ◆ Add an attribute to one source class and define a **search method** in the other, to be used when the relationship is to be navigated in the opposite direction.

Example



```
public class Company {...
    private Set<Person> employee = new HashSet<Person>();
    public void addEmployee(Person p) { ...employee.add(p);}
    public void removeEmployee(Person p) {...employee.remove(p);}
}
```

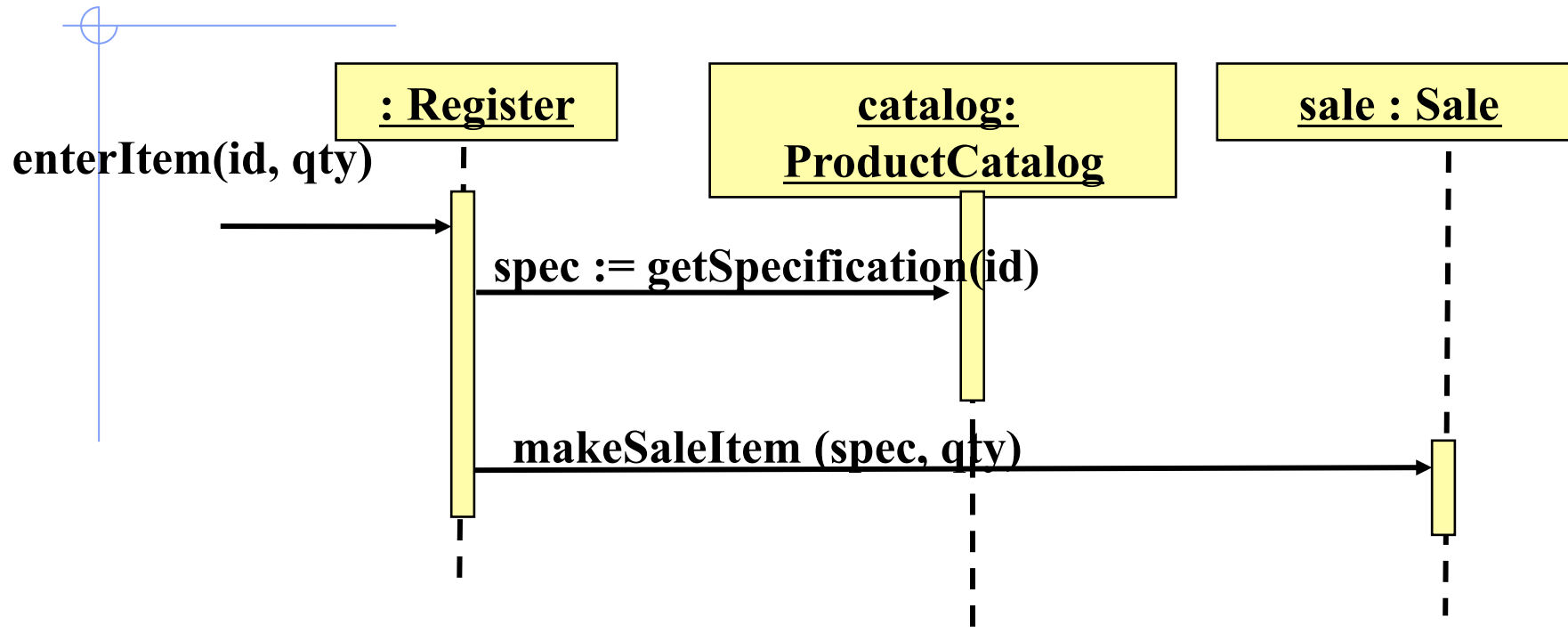
```
public class Person {...
    private Company employer;
    public Person(Company c) {c.addEmployee(this); employer = c; }
    public void setCompany(Company c) {
        employer.removeEmployee(this);
        c.addEmployee(this); employer = c; }
}
```

(Maintenance of an invariant involved here: all employees of company c have employer attribute equal to c.)

Obtaining methods from sequence diagrams

- ◆ Sequence diagram messages represent method calls.
- ◆ The temporal order (sequence) of messages in an activation record represents a sequence of method calls initiated by the corresponding method.
- ◆ A sequence diagram is underspecified with regard to choice, repetition and multi-objects. Resolving this choice involves making coding decisions.
- ◆ A sequence diagram can be used to model error handling using exceptions.

Example (temporal order)

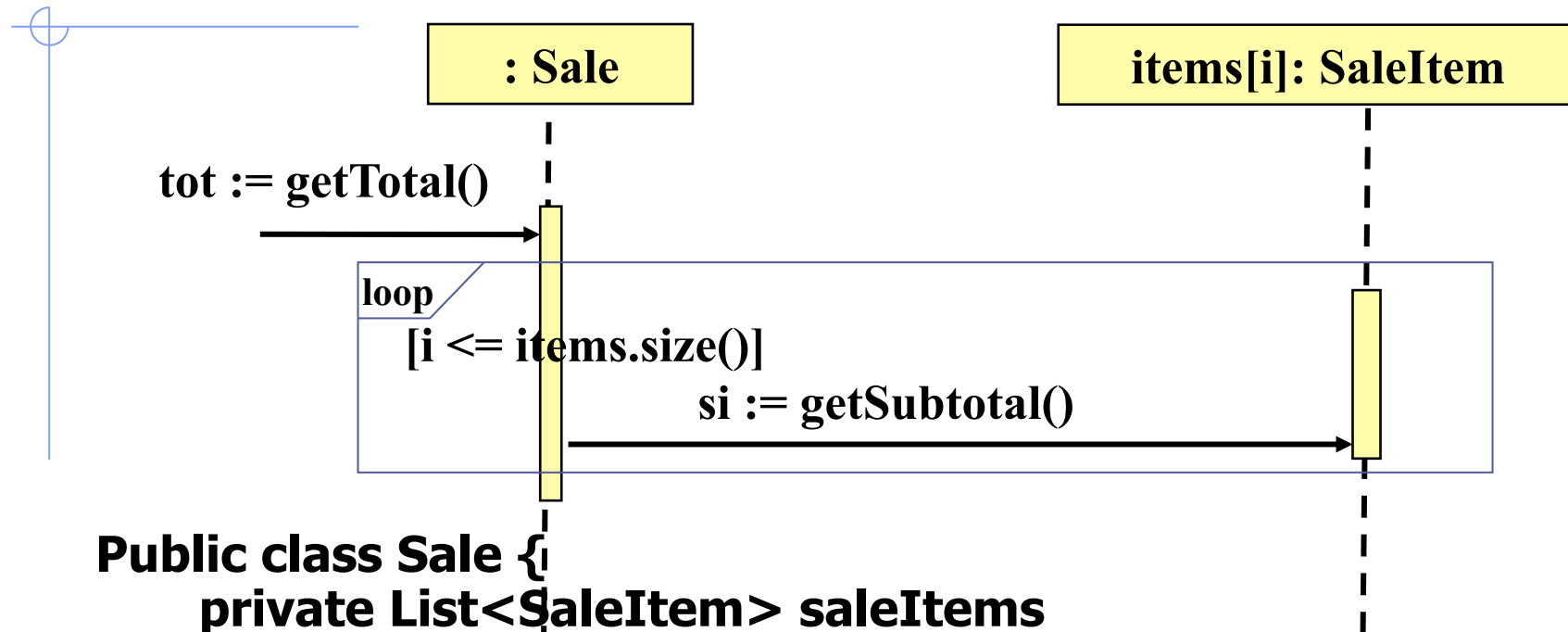


```
public class Register {
    private ProductCatalog catalog;
    private Sale sale;
    public void enterItem(ItemID id, int qty) {
        ProductSpec spec = catalog.getSpecification(id);
        sale.makeSaleItem(spec, qty);
    }
}
```

attribute visibility

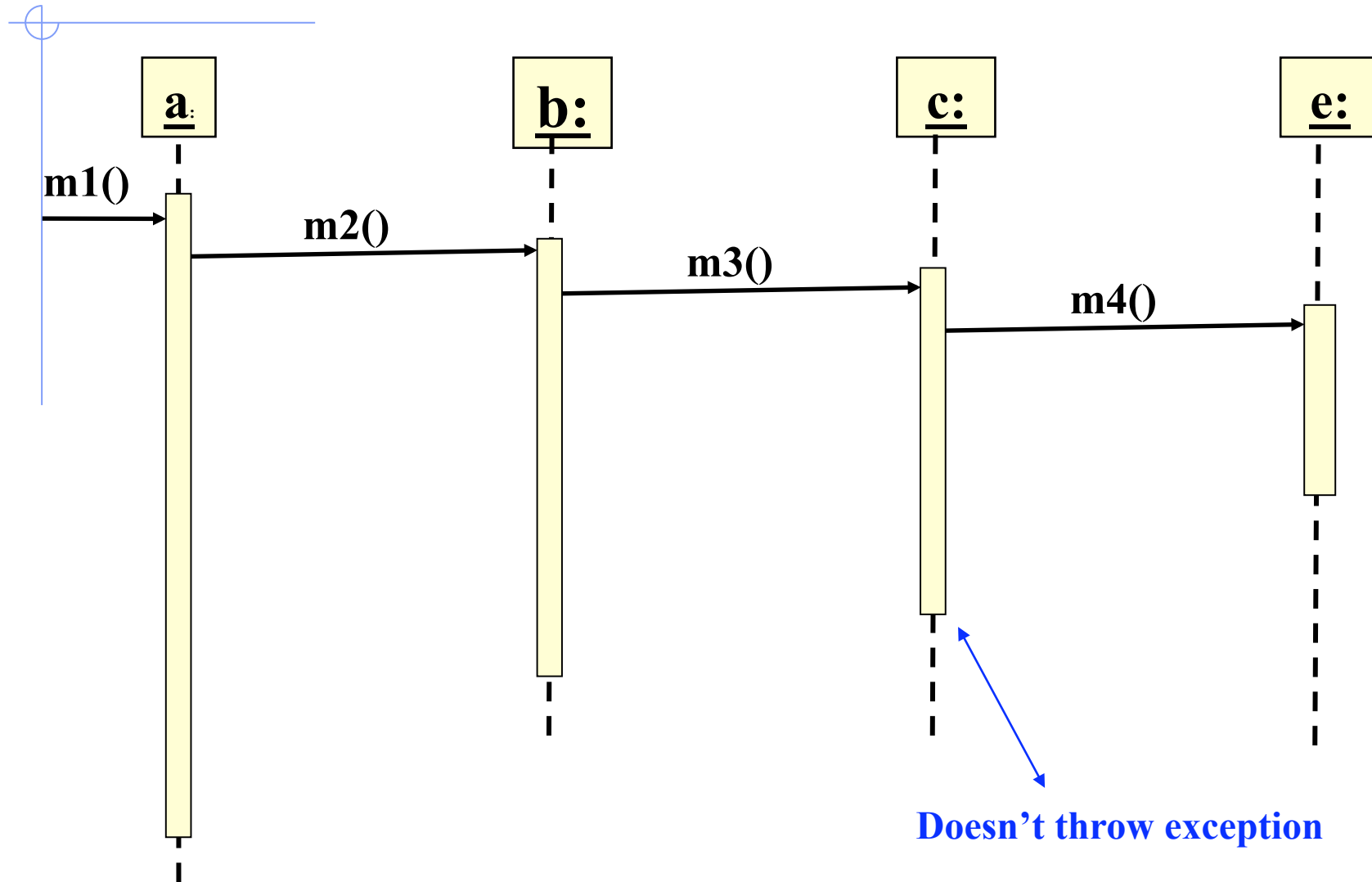
local visibility

Example (under specification)

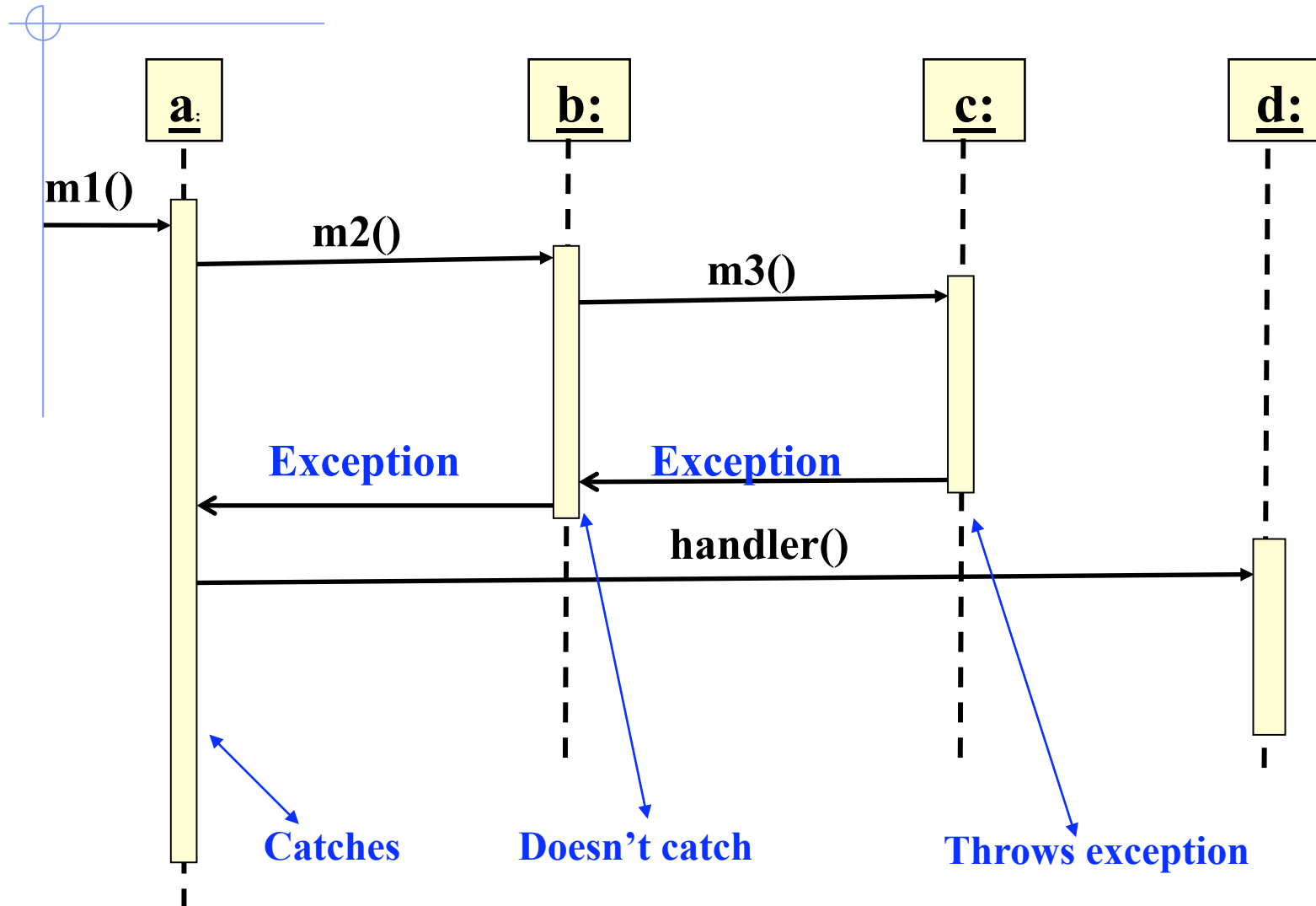


```
Public class Sale {  
    private List<SaleItem> saleItems  
        = new LinkedList<SaleItem>();  
    public Money getTotal() {  
        Money total = new Money();  
        for( si : saleItems ) {  
            total.add( si.getSubtotal() );  
        }  
        return total;  
    }  
}
```

Example (error handling)



Example (error handling)



Follow-up Reading

- ◆ Larman Chap. 20: Mapping designs to code.