System Analysis and Design

L20. Mapping Designs to Code

Topics

- Mapping Designs to Code
  - Creating Classes
  - Creating Methods
  - Collection Classes in Code
  - Exception and Error Handling
  - Order of Implementation
- Test-Driven Development
- Refactoring

(UP) Implementation Model

- This is the actual program: source code, database definitions, JSP, ASP, etc.
- Code is not part of OOAD, it is the goal
- Modern tools permit much design while programming.

We do implementation "modeling". (実装モデル)

Inputs for Code Generation

- The UML artifacts created during the design work,

  the interaction diagrams and DCDs, 交互图和DCD

  will be used as input to the code generation process.

Design-while-programming

- Modern development tools provide an excellent environment to
  - quickly explore alternate approaches
  - quickly refactor alternate approaches

  } 设计耦关
- Some (often lots) design-while-programming is worthwhile

实现时仍需做很多设计与重构的工作.

Creativity and Change During Implementation

- Your design is a necessary first step, but design models and class diagrams are incomplete 设计是第一步，但它不完整.
- During programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved. 实现与测试时，大量的变化会出现，细节的问题被发现并解决
- Your understanding of the domain generated during the process provide the basis for writing code 你对领域的理解是代码基础
- But, **expect and plan for lots of change and deviation from the design during programming**. That's a key, and pragmatic attitude in iterative and evolutionary methods 要考虑到在实现时会出现大量的设计变化和设计偏差.

Mapping Designs to Code

- Implementation in an object-oriented language requires writing source code for:
  - class and interface definitions
  - method definitions

类、接口定义

方法定义

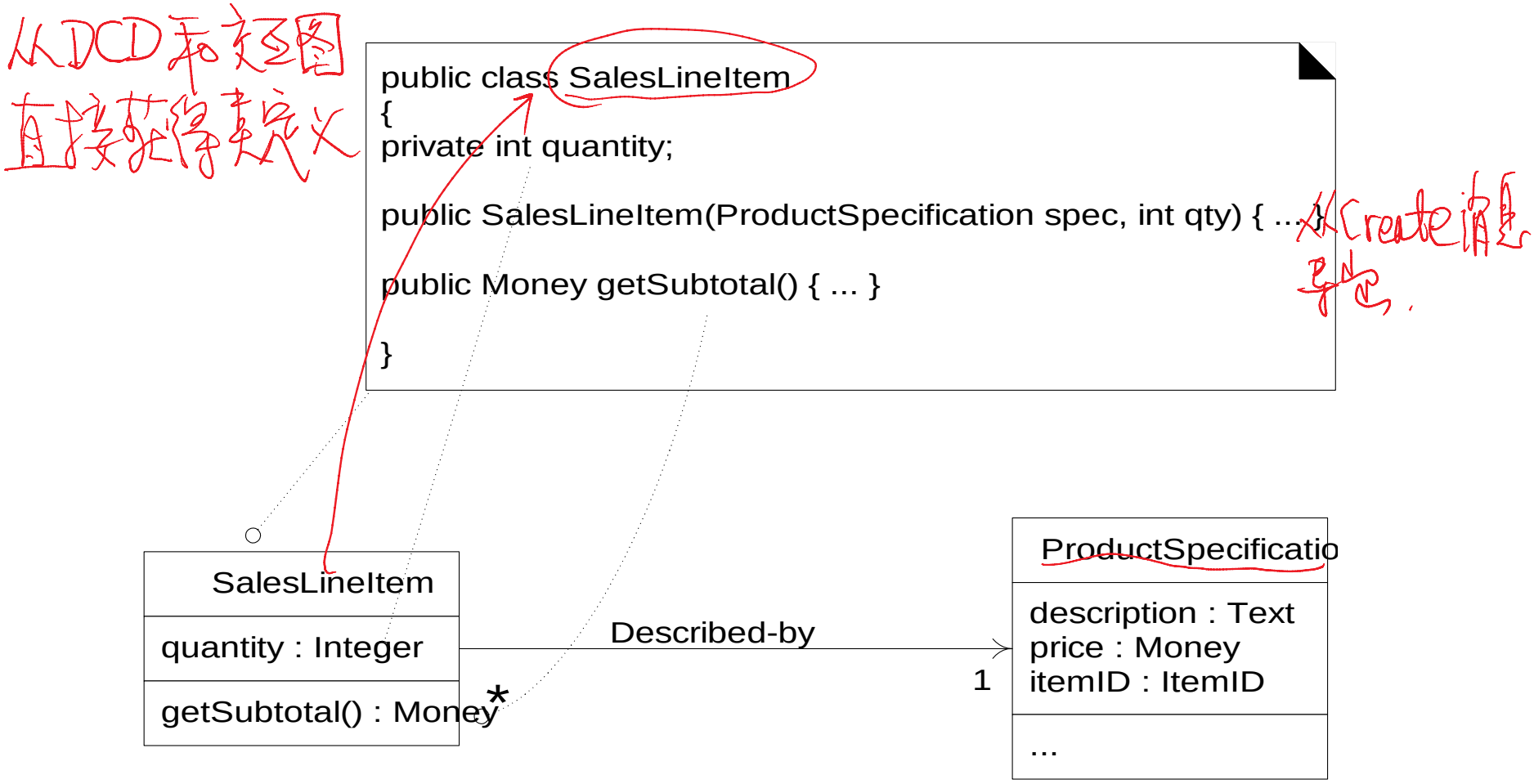
实现的主要任务

Creating Classes from Design Class Diagrams

从DCD创建类

- At the very least, DCDs depict the class or interface name, superclasses, operation signatures, and attributes of a class.
- This is sufficient to create a basic class definition in an OO language.
- Some UML tools will generate the basic class structure from your class diagrams
- You can easily derive your class in Java, C#, etc from a class diagram

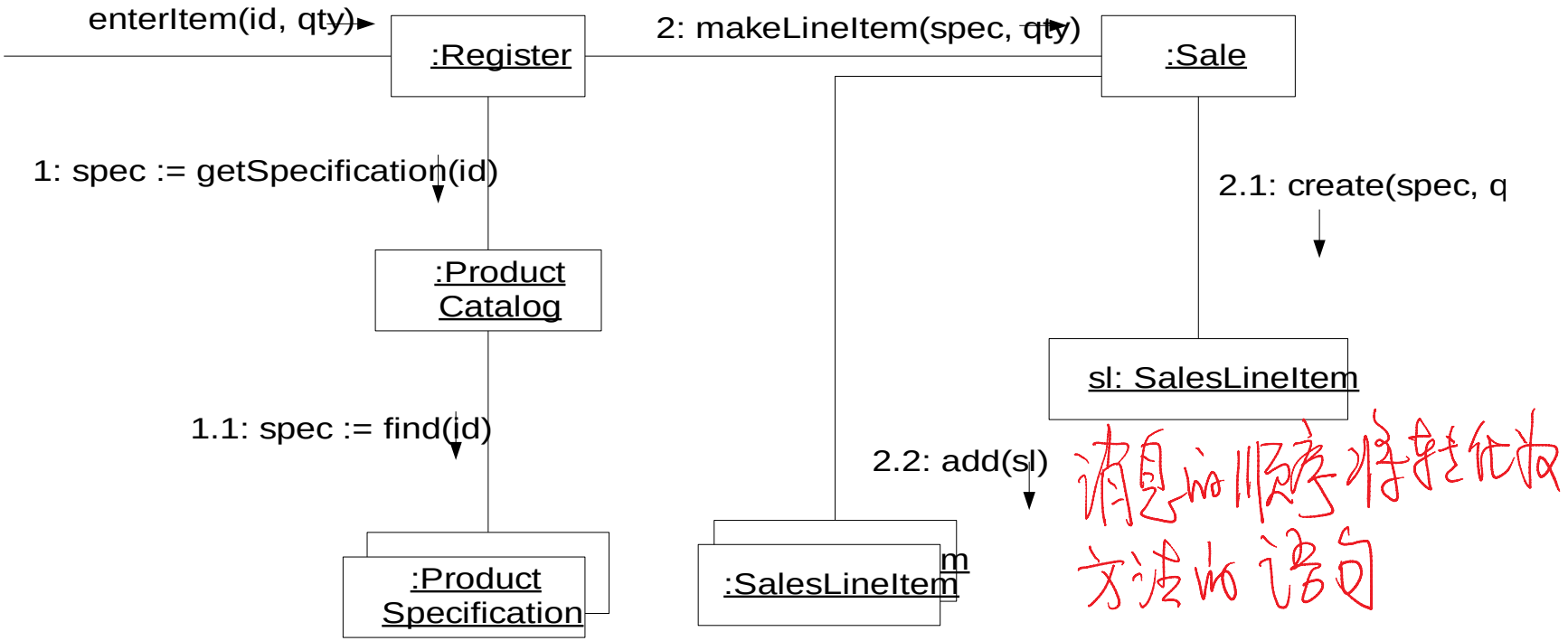
SalesLineItem Example



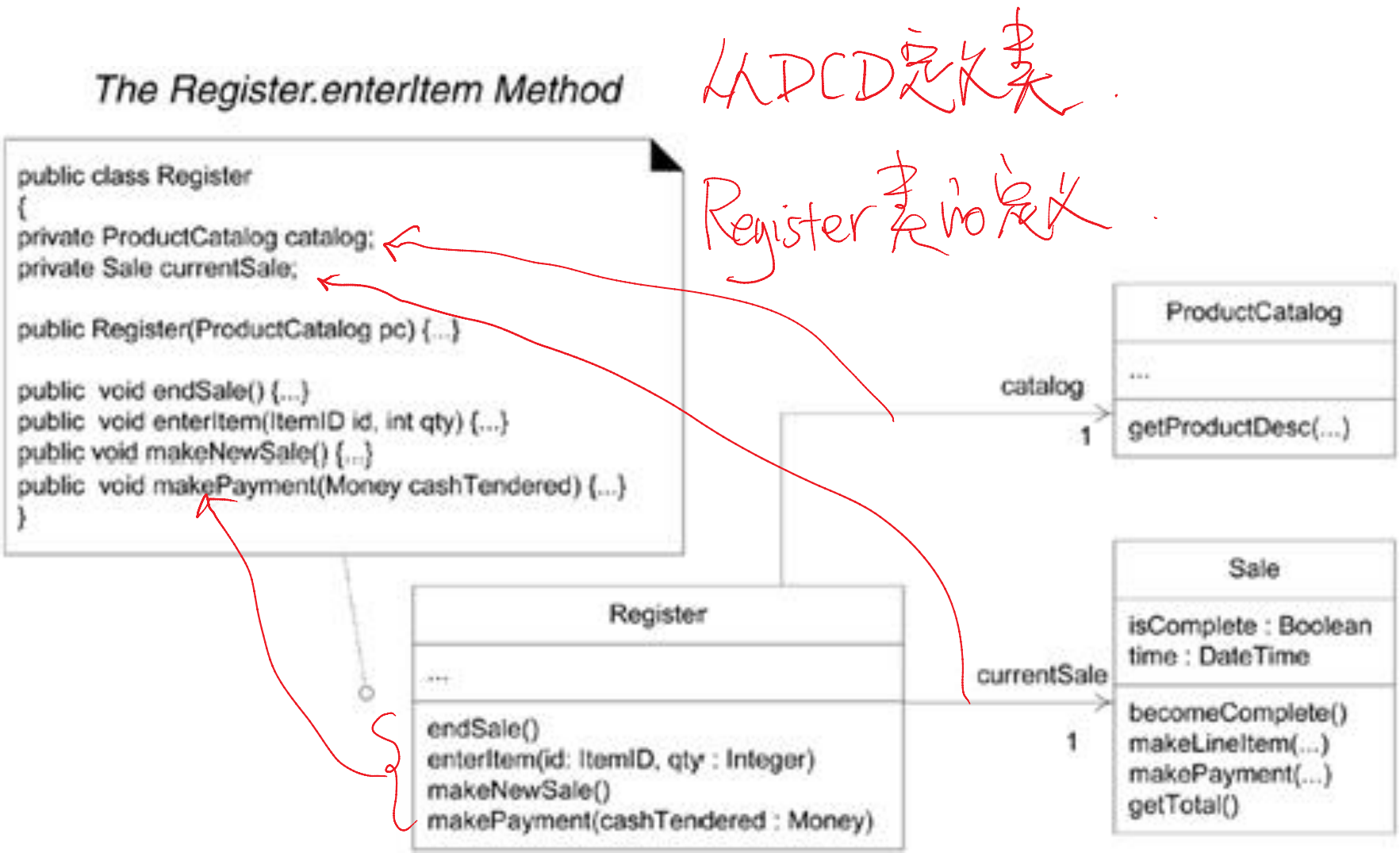
Note the addition in the source code of the Java constructor *SalesLineItem(…)*. It is derived from the *create(desc, qty)* message sent to a *SalesLineItem* in the *enterItem* interaction diagram.

Creating Methods from Interaction Diagrams

- The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions.
- We will explore the implementation of the Register and its enterItem method

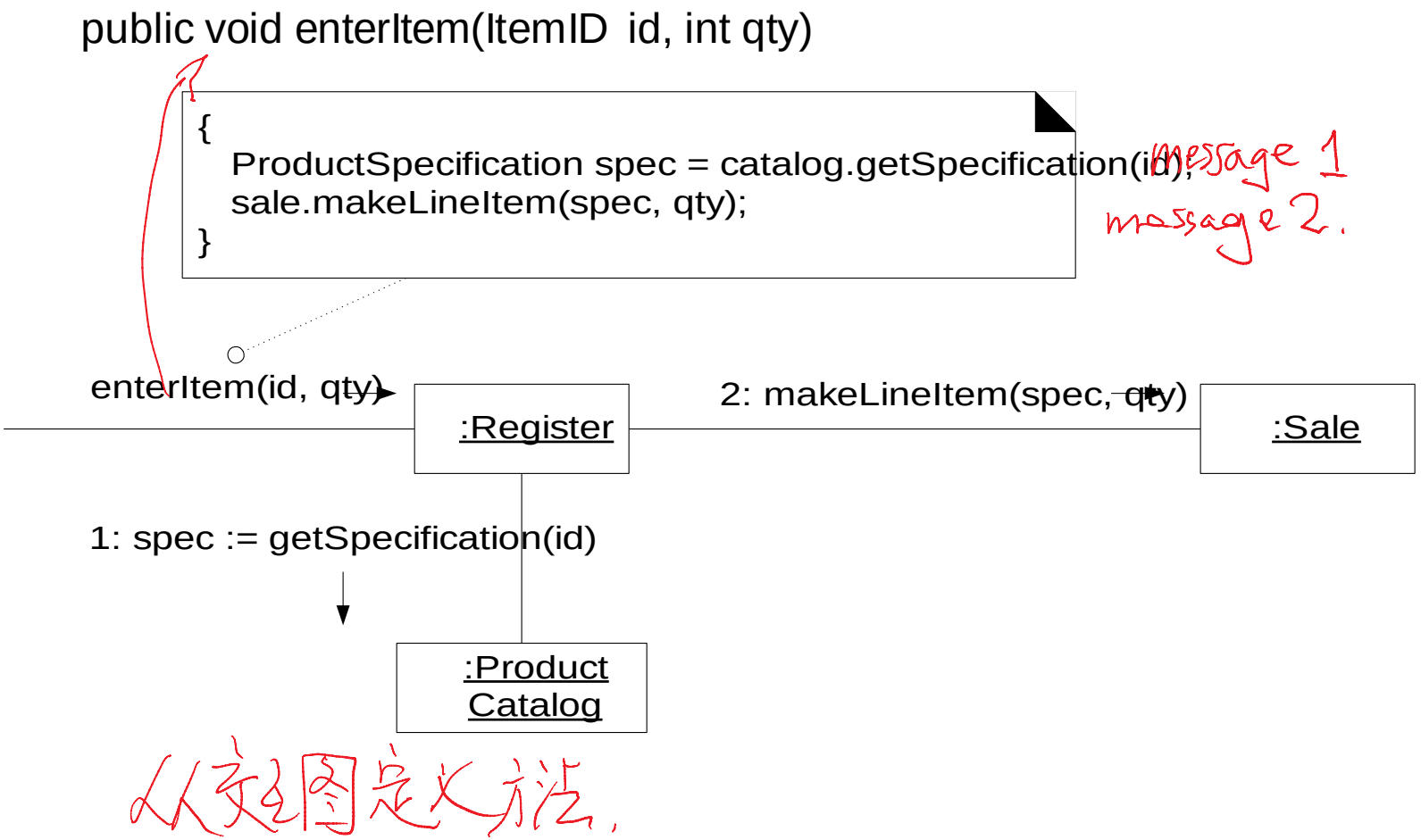


A Java definition of the *Register* class



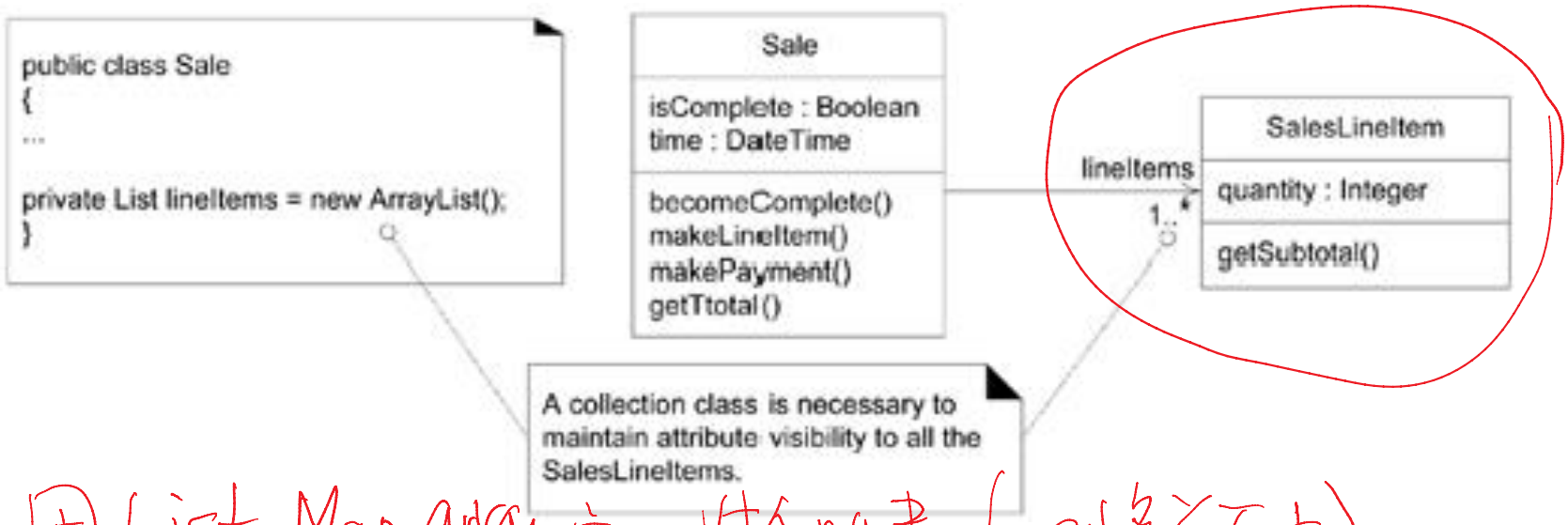
Methods from Interaction Diagrams

The *enterItem* interaction diagram illustrates the Java definition of the *enterItem* method.



Collection Classes in Code

- One-to-many relationships may are usually implemented with the introduction of a **collection** object, such as a *List* or *Map*, or even a simple array.



Guideline: Code to Interface

- If an object implements an interface, declare the variable in terms of the interface, not the concrete class.

```
private List lineItems = new ArrayList();
```

接口　　具体类

使用接口声明，而不要用具体类声明

Exception and Error Handling

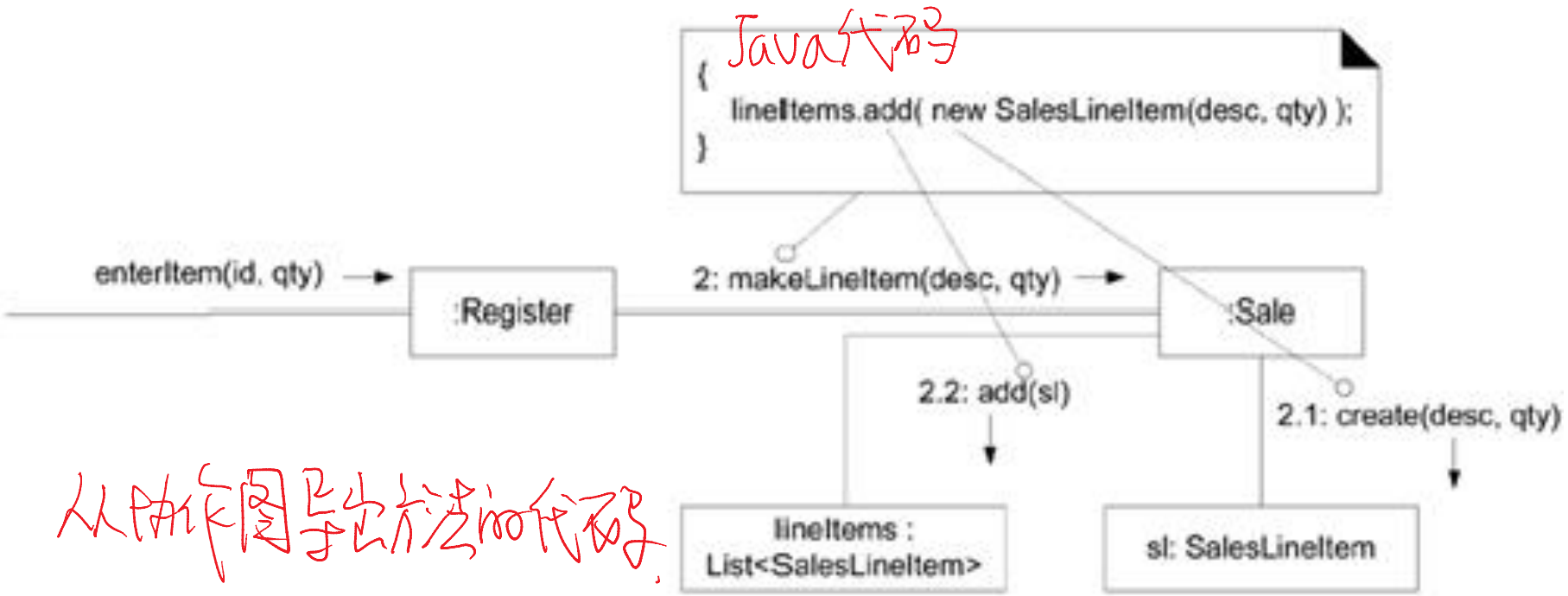
异常和错误处理.

- This is too often ignored in the design
- Consider large-scale strategy as they have a large-scale architectural impact 大规模的考虑
  - Communication failure
  - "hard" errors
- Consider case-by-case strategy 按情况各自处理.
  - What if the record we "know" is there not found
  - What if there is bad data
- In UML, errors can be property strings of messages 在UML中，可以用特性串表示错误.

Defining the Sale.makeLineItem Method

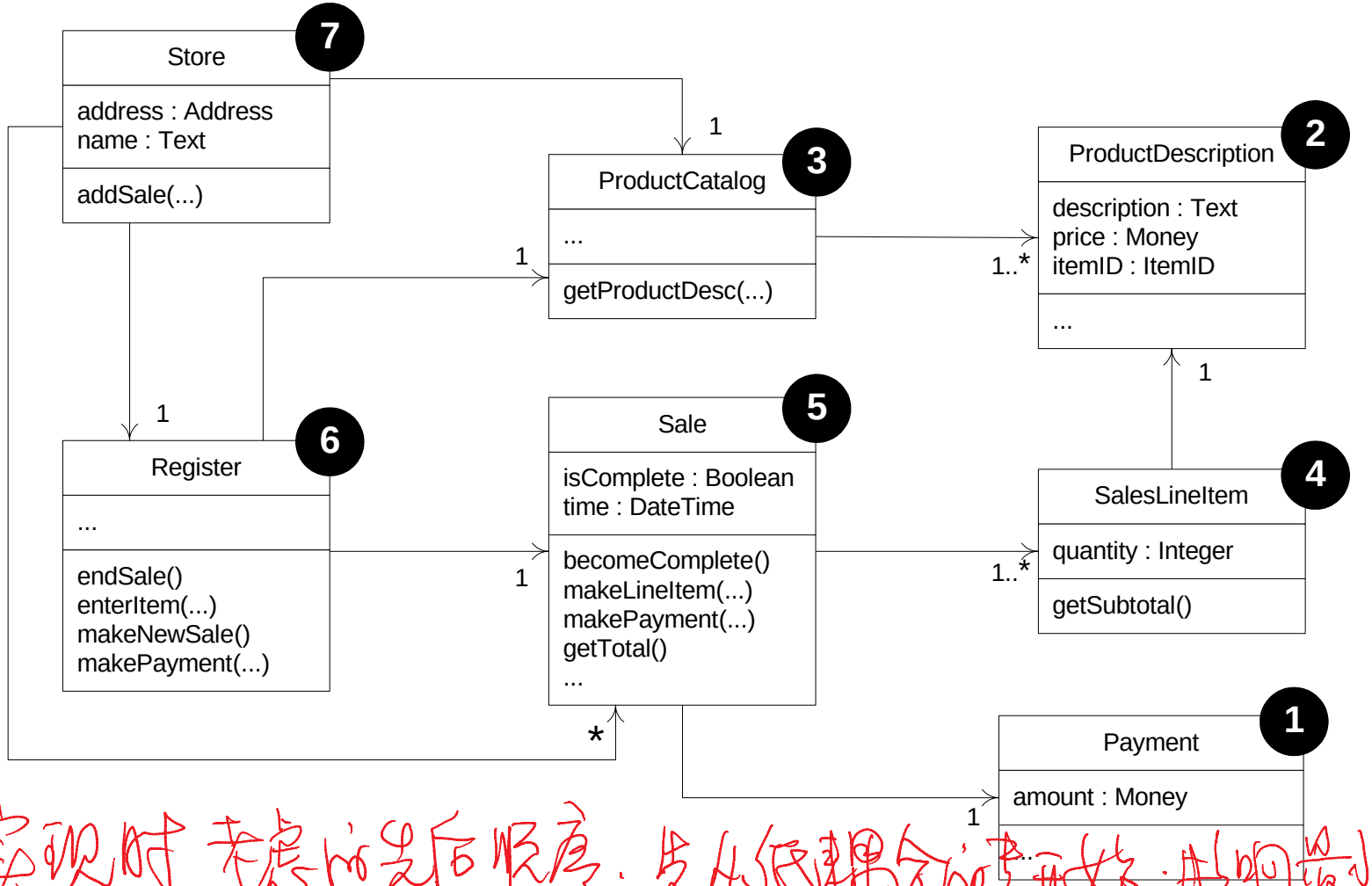
Case study

- The *makeLineItem* method of class *Sale* can also be written by inspecting the *enterItem* collaboration diagram with the accompanying Java method



Order of Implementation

- From least-coupled to most-coupled (why?)



在实现时，考虑的先后顺序：先从依赖别的类开始，然后是...

Summary of Mapping Designs to Code

- As demonstrated, there is a translation process
  - from UML class diagrams to class definitions, 类定义
  - and from interaction diagrams to method bodies. 方法定义
- There is still lots of room for creativity, evolution, and exploration during programming work. 在程序设计阶段仍有大量创造性工作。(设计，甚至分析工作)

Test-Driven Development (TDD)

- An excellent practice promoted by the iterative and agile XP method, and applicable to the UP (as most XP practices are), is **test-driven development** (TDD).
- It is also known as **test-first development**.

这个敏捷方法提倡测试驱动开发(测试先开发)

Test-Driven Development

- Create your unit tests before you write the code 在写代码之前先建立单元测试
- Writes unit testing code for *all* production code. 为所有的代码书写单元测试代码.
- The basic rhythm is to write a little test code, then write a little production code, make it pass the test, then write some more test code, and so forth.

基本规则是：小步测试，小步代码推进.

Advantages of Write the Tests First

- The tests actually get written
- Programmer satisfaction leads to more consistent test writing
- Clarification of detailed interface and behavior
- Provable, repeatable, automated verification
- Confidence to change things

Example

- To test the **Sale** class, create a **SaleTest** class that:
  - Creates a Sale (the thing to be tested; a fixture)
  - Adds some line items to it with the makeLineItem method
  - Asks for the total and verifies that it is the expected value

测试方法的模板：

Pattern of Testing Methods

- Create the fixture 创建被测件.
- Do some operation you want to test 执行要测试的操作
- Evaluate the results 确认结果.

- **Point:** Don't write all of the tests for Sale first; write one test method, implement the solution in Sale to make it pass, then repeat

(小步迭代)

Using xUnit

- Create a class that extends the TestCase class
- Create a separate testing method for each Sale method you want to test. This will generally be every public method.

```
public class SaleTest extends TestCase
{
    // …
    // test the Sale.makeLineItem method
    public void testMakeLineItem()
    {
        // STEP 1: CREATE THE FIXTURE
        Sale fixture = new Sale();
        // set up supporting objects for the test
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
        ItemID id = new ItemID( 1 );
        ProductDescription desc =
                new ProductDescription( id, price, "product 1" );

        // STEP 2: EXECUTE THE METHOD TO TEST
        sale.makeLineItem( desc, 1 );    // test makeLineItem
        sale.makeLineItem( desc, 2 );

        // STEP 3: EVALUATE THE RESULTS
        // verify the total is 7.5
        assertTrue( sale.getTotal().equals( total ));
    }
}
```

从TestCase继承.

测试方法定义(要测试MakeLineItem())

①建立被测件

②执行被测试方法

③确认结果

Using xUnit

Refactoring 重构

- A structured, disciplined method to 重整代码的方法 rewrite or restructure existing code
  - without changing its external behavior, 不改变行为
  - applying small transformations and re-testing each step. 小步推进

Relationship between Refactoring and TDD

- After each transformation, the unit tests are re-executed to prove that the refactoring did not cause a regression (failure). 每次重构需用测试验证
- Therefore, all those unit tests support the refactoring process. 所以单元测试支持了重构.
- Each refactoring is small 每一个重构很小.
- But a series of transformations, each followed by executing the unit tests again, can produce a major restructuring of the code and design (for the better) 一系列的重构导致更好的代码.
- All the while ensuring the behavior remains the same. 保证行为不变

Goals of Refactoring

- Remove duplicate code
- Improve clarity
- Make long methods shorter
- Remove the use of hard-coded literal constants
- and more…

Good Code Qualities

- Code that's been well-refactored is
  - short
  - tight
  - clear
  - without duplication
- Code that doesn't have these qualities *smells bad*. (there is a poor design)

Code Smells 代码的臭味

- Duplicated code 重复的代码
- Big methods 方法太大
- Class with many instance variables 类内太多实例变量
- Class with lots of code 类内太多代码
- Strikingly similar subclasses 明显的类同子类
- Little or no use of interfaces 没有使用接口
- High coupling 高耦合

改正坏味代码正是重构

The remedy to smelly code are the refactorings.

Basic Refactorings 基本的重构方法

Refactoring	Description
Extract Method	Transform a long method into a shorter one by factoring out a portion into a private helper method.
Extract Constant	Replace a literal constant with a constant variable.
Introduce Explaining Variable (specialization of Extract Local Variable)	Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.
Replace Constructor Call with Factory Method	In Java, for example, replace using the new operator and constructor call with invoking a helper method that creates the object (hiding the details).

More Refactorings

- Change method parameters (changes them in the definition, then finds all references)
- Replace using *new* with a call to a method that returns an object. (*Factory* pattern)

取代使用 new，而是用一个方法返回对象。

The takeTurn method before refactoring

```
public class Player
{
  private Piece piece;
  private Board board;
  private Die[] dice;
  // …
  public void takeTurn()
  {
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
      dice[i].roll();
      rollTotal +=
            dice[i].getFaceValue();
    }
    Square newLoc =
      board.getSquare(
        piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
  }
} // end of class
```

Extract Method.

Code after Applying the Extract Method Refactoring

```
public class Player
{
private Piece piece;
private Board board;
private Die[] dice;
// …
public void takeTurn()
{
  // the refactored helper method
  int rollTotal = rollDice();
  Square newLoc =
board.getSquare(piece.getLocation(),
rollTotal);
  piece.setLocation(newLoc);
}
```

```
  private int rollDice()
  {
  int rollTotal = 0;
  for (int i = 0; i < dice.length; i++)
  {
  dice[i].roll();
  rollTotal += dice[i].getFaceValue();
  }
  return rollTotal;
  }
  } // end of class
```

私有的方法

Extract Method Example

```
Public class Die {
  public static final int MAX=6;
  public int faceValue;
  public Die(){
    roll();
  }
  Public void roll() {
    faceValue = (int)((Math.Random() * MAX) + 1;
  }
  Public int getFaceValue() {
    return faceValue;
  }
}
```

抽出的方法

Before introducing an explaining variable

```
// good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
  return (((year%400)==0)||(((year%4)==0)&&((year%100)!=0)));
}
```

还算不清晰，需重构

simple refactorings is *Introduce Explaining Variable* because it clarifies, simplifies, and reduces the need for comments.

引入有含义的变量

After introducing an explaining variable

```
// that's better!
boolean isLeapYear( int year )
{
    boolean isFourthYear = ( ( year % 4 ) == 0 );
    boolean isHundrethYear = ( ( year % 100 ) == 0);
    boolean is4HundrethYear = ( ( year % 400 ) == 0);
    return ( is4HundrethYear
    || ( isFourthYear && ! isHundrethYear ) );
}
```

引入有含义的变量.

变得清晰.

IDE Support for Refactoring

- Most of the dominant IDEs include automated refactoring support.

很多IDE支持自动重构功能。