Mapping design to code

Week 6 lecture
August 31, 2005

Agenda

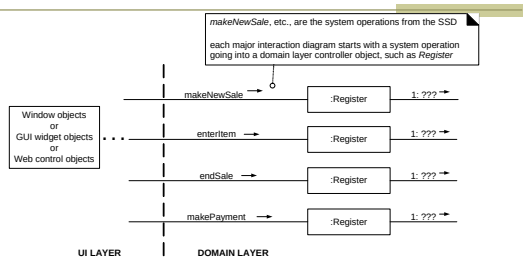
- Use Case Realization with GRASP
  - POS project
  - Monopoly Game project
- Designing for visibility
- Mapping design to code
- Test-driven development and refactoring

Use-Case Realization

- "…describes how a particular use case is realized within the design model, in terms of collaborating objects" [RUP]
- Individual scenarios are realized

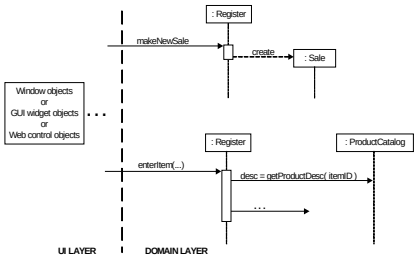
Use case -> System events -> System sequence diagram ->
System operation contracts -> Interaction diagrams -> Design classes

System operation



- The system operations in the SSD are used as the start messages into the domain layer
- If communication diagrams are used, one per system operation
- Same for sequence diagram

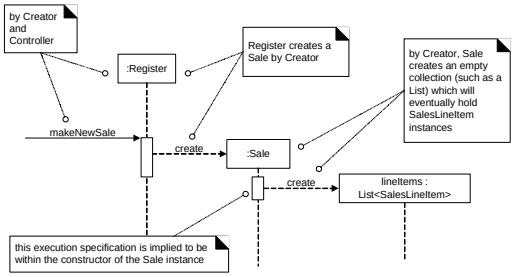
One diagram per system operation



Design `makeNewSale`

- Choosing the controller class
  - A façade controller is satisfactory if there are only a few system operations
  - Use `Register` here.
- Creating a new `Sale`
  - `Register` create `Sale`
  - `Sale` create a collection to store SalesLineItems

Design `makeNewSale`



Design: enterItem

Contract CO2: enterItem

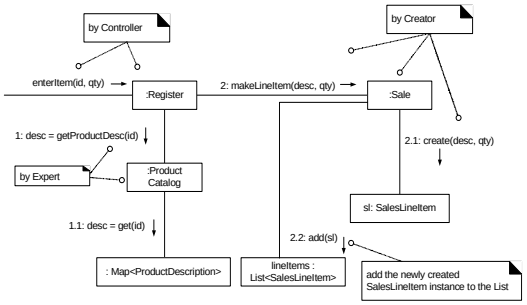
Operation:	enterItem(itemID : ItemID, quantity : integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	– A SalesLineItem instance sli was created (instance creation). – sli was associated with the current Sale (association formed). – sli.quantity became quantity (attribute modification). – sli was associated with a ProductSpecification, based on itemID match (association formed).

[Larman 2002]

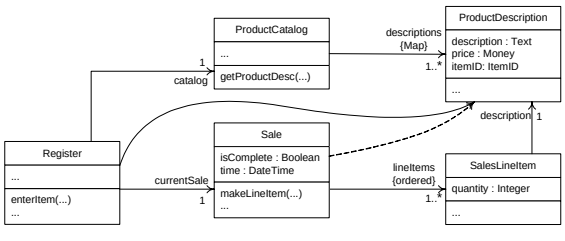
Design **`enterItem`**

- Choosing controller class
- Display item description and price (ignore at this stage)
- Create a new **`SalesLineItem`**
- Finding a **`ProductDescription`**
- Visibility to **`ProductCatalog`**
- Temporary and persistent storage
  - We can defer database design and use temporary memory object instead

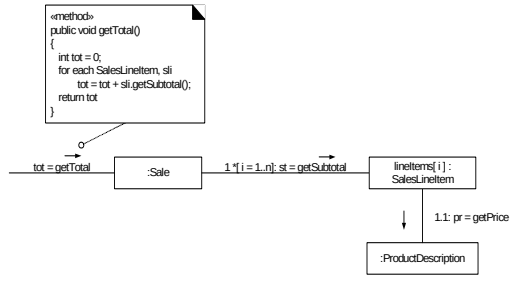
Design **`enterItem`**



Partial Design Class Diagram

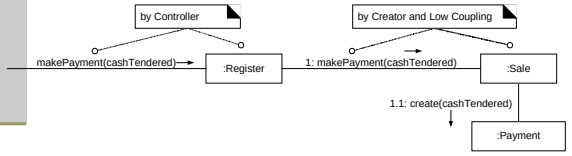


Design **`endSale`**



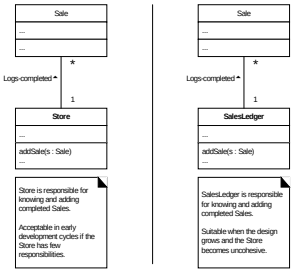
Design makePayment

- Creating payment

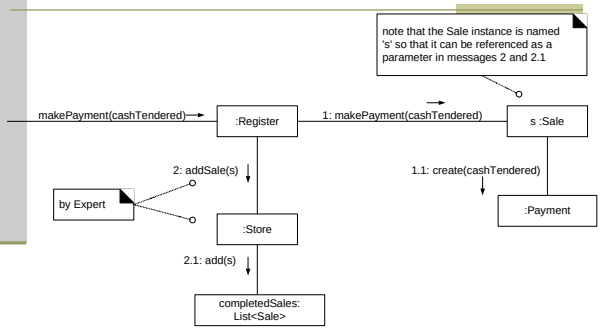


Design **makePayment**

- Logging a sale



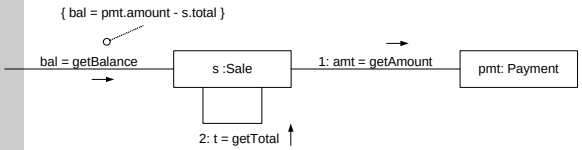
Design **makePayment**



Design **makePayment**

- Calculating the balance
  - Who is responsible for knowing the balance?
    - To calculate, sale total and payment tendered are required
    - *Sale* and *Payment* are partial Experts
  - Consider coupling…
    - *Sale* already has visibility into *Payment*
    - *Payment* does not have visibility into *Sale*
    - Giving *Sale* primary responsibility doesn't increase overall coupling

getBalance design



Object Design: startUp

- Initial system operation
- Delay until all other system operations have been considered

Do the initialization design last

Create Initial Domain Object

```
public class Main
{

public static void main( String[] args )
{
    // Store is the initial domain object.
    // The Store creates some other domain objects.

    Store store = new Store();

    Register register = store.getRegister();

    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    ...
}

}
```

[Larman 2003]

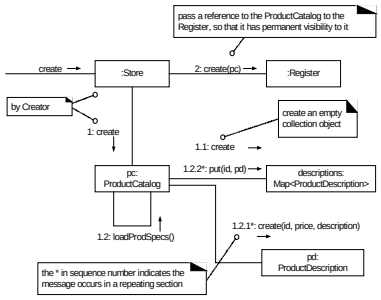
Designing startUp

1. In one interaction diagram, model a create() message to the initial domain object
2. (optional) If the initial object takes control of the process, model a run() message in a second diagram

Designing Store.create()

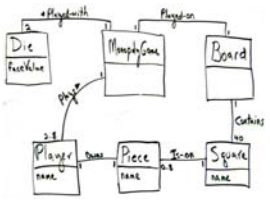
- **Create**: *Store, Register, ProductCatalog,ProductSpecifications*
- **Associate**: *ProductCatalog* with *ProductSpecifications*; *Store* with *ProductCatalog*; *Store* with *Register*; *Register* with *ProductCatalog*

Designing Store.create()



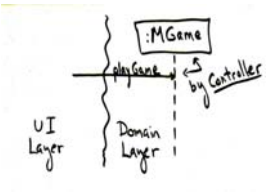
Monopoly game project

- Two system operations in the first iteration
  - Initialize
    - Ignore at this state
  - Play game
    - Main focus



Design **playGame**

- Choosing the Controller Class
  - Represents the overall "system", "root object"
    - MonopolyGame? *
  - Represents a receiver or handler of all system events of a use case scenario
    - PlayGameHandler?



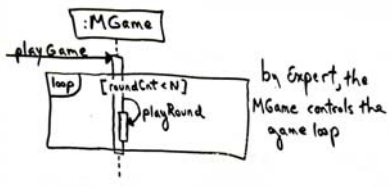
Game-Loop algorithm

- Terminology
  - *turn* – a player rolling the dice and moving the piece
  - *round* – all the players taking one turn
- Game loop

```
for N rounds
   for each player p
     p takes a turn
```


Who is responsible for controlling the game loop

Information needed	Who has this information
The current round count	No object has it yet, assigning this to the **`MonopolyGame`** object is justifiable
All players	**`MonopolyGame`** object

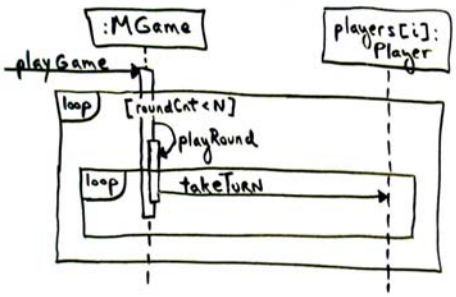


Who takes a turn?

- When there are multiple partial information experts to choose from
  - Place the responsibility in the dominant information expert
  - If the first guideline does not apply, consider the coupling and cohesion impact of each and choose the best
  - Consider possible future evolution
    - Taking a turn might involve buying property, deposit money in bank.. Etc. A player should have all information regarding those activities

Information Needed	Who has the information
Current location of the player	**`Player`**
Two die objects	**`MonopolyGame`**
All the squares	**`Board`**

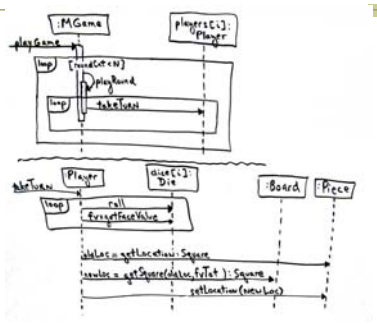
Partial design diagrams



Taking a turn refine

- Taking a turn involves
  - Calculating the face value of two die
  - Calculating the new square location
  - Moving the player's piece from an old location to a new location
- Who coordinate all this
  - **`Player`**
- Visibility problem
  - **`Player`** must have references to all those objects

Taking a turn



Command-Query Separation Principle

- Style 1#

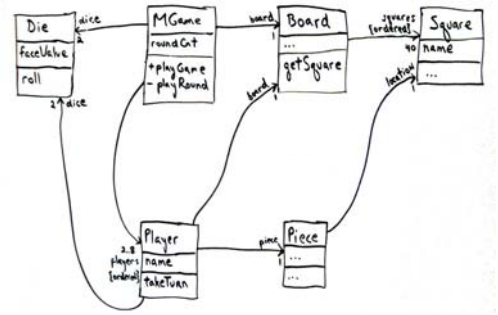
```
public void roll(){
   faceValue = //random num generation
}
public int getFaceValue(){
        return faceValue;
}
```

- Style 2#

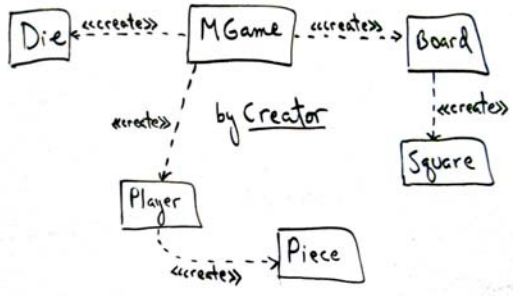
```
public int roll(){
   faceValue = //random num generation
   return faceValue;
}
```

- Every method should either be:
  - A command method that performs an action often has side effects such as changing the state of objects and is **`void`**.
  - A query that returns data to the caller and has no side effects – it should not permanently change the state of any objects.
- But a method should not be both

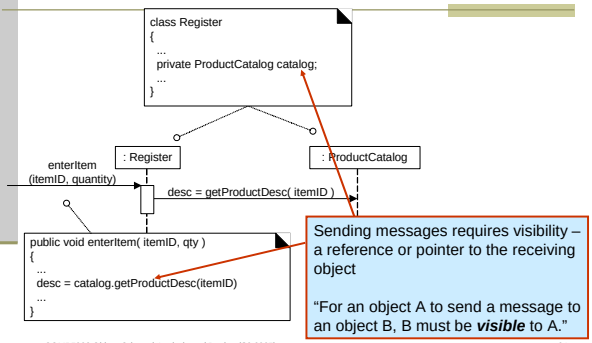
Static design for playGame



Initialization and the start up use case



Visibility between objects

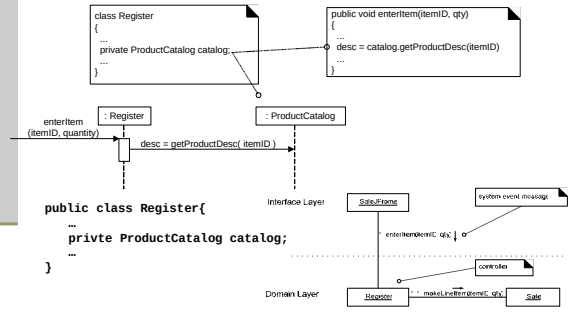


What is visibility

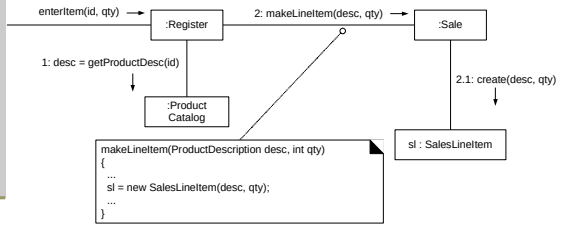
- Ways in which A can have visibility into B:
  - Attribute Visibility
    B is an attribute of A
  - Parameter Visibility
    B is a parameter of a method of A
  - Local Visibility
    B is a (non-parameter) local object in a method of A
  - Global visibility
    B is in some way globally visible

Listed in order of how commonly they appear in object-oriented systems (from most-common to least-common)

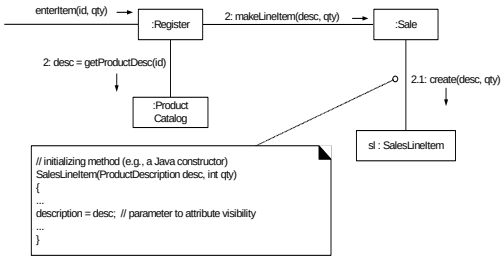
Attribute Visibility



Parameter Visibility

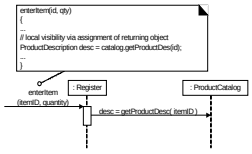


Parameter to attribute visibility



Local visibility

- Local visibility from A to B exists when B is declared as a local object with in a method of A
  - Create a new local instance and assign it to a local variable
  - Assign the returning object from a method invocation to a local variable
    - Variable **`desc`** of type **`ProductionDescription`** is a local variable of **`enterItem`** method.



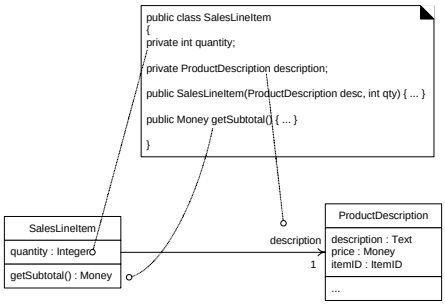
Global visibility

- Global visibility from A to B exists when B is global to A;
- Permanent visibility as it persists as long as A and B exist
  - Global variable
  - Singleton pattern

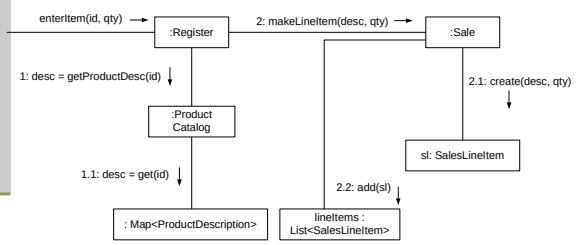
Mapping Designs to Code

- Class and interface definition
- Method definitions

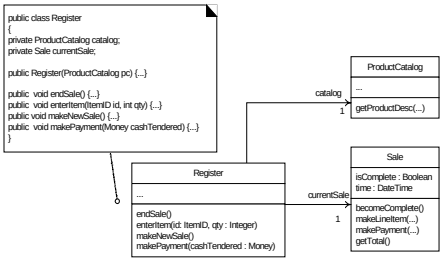
Creating Class Definition from DCD



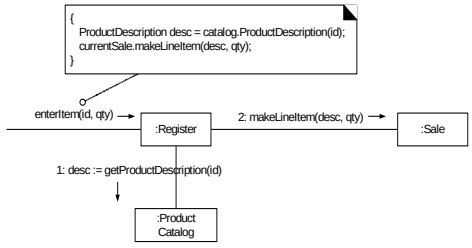
Creating methods from interaction diagram



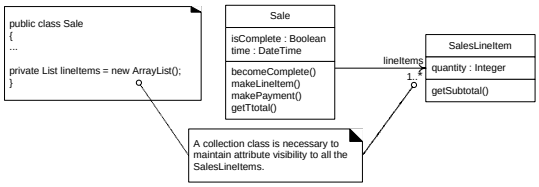
The **`Register`** class



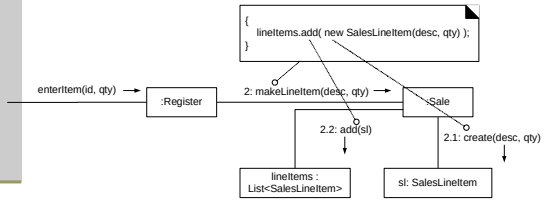
The `enterItem` method



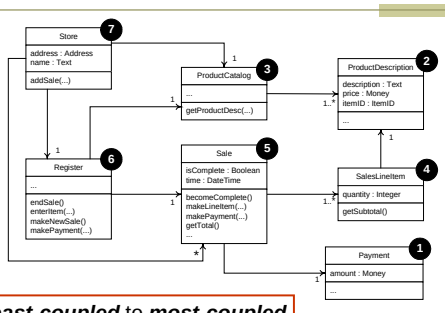
Collection classes in code



Defining the Sale.makeLineItem Method



Order of implementation



From ***least-coupled*** to ***most-coupled***

Test-Driven development

- One of the 12 XPXtudes
- The unit tests actually get written
- The programmer satisfaction leading to more consistent test writing
- Clarification of detailed interface and behavior
- Provable, repeatable automated verification
- The confidence to change things.
- See example in page 388

JUnit

- A regression testing framework written by Erich Gamma and Kent Beck
- Used by the developer who implements unit tests in Java
- Center of Test-Driven development
- Excellent case of OO Design and pattern application

A simple TDD example

- Problem
  - We want to write a Calculator
  - Functional requirement
    - Perform simple integer arithmetic operation: add, sub, multiplex and divide

TDD solution

- TDD solution
  - Write a unit test for class **`Calculator`**
    - A class **`CalculatorTest`**
  - What do we do in **`CalculatorTest`**
    - Create a Calculator
    - Use the calculator to `add` two integer, say 5 and 6
    - Get the total, and verify that it is the expected value (11)

Test Code

- Test Code

```
import junit.framework.TestCase;
public class CalculatorTest extends TestCase{
  public void testAdd(){
      Calculator calculator = new Calculator();
      int result = calculator.add(5,6);
      assertEquals(11, result);
  }
}
```


Product code

- Product Code

```
public class Calculator{
  public int add (int number1, int number2){
      return number1 + number2;
  }
```

- We do not write all the test first, we test one or a few at first
- Run the testing

```
D:\> java junit.textui.TestRunner
  TestCalculator
.
Time: 0.01
OK (1 test)
D:\>
```


JUNIT – GUI

- Run the testing with GUI

```
D:\>java junit.swingui.TestRunner TestCalculator
```

What does a failed test look like?

```
//test code
import junit.framework.TestCase;
public class TestCalculator extends TestCase{
  public void testAdd(){
      Calculator calculator = new Calculator();
      int result = calculator.add(5,6);
      assertEquals(11, result);
  }
}
// product code
public class Calculator{
  public int add (int number1, int number2){
      return number1 = number2;
  }
}
```

JUnit Textual interface – failed test

```
D:\>java junit.textui.TestRunner TestCalculator
.F
Time: 0.02
There was 1 failure:
1) testAdd(TestCalculator)junit.framework.AssertionFailedError:
    expected:<11> but was:<6>
        at TestCalculator.testAdd(TestCalculator.java:7)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at
   sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
   ava:39)
        at
   sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccess
   orImpl.java:25)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```


Proceed with more tests and methods

```
//test code
import junit.framework.TestCase;
public class TestCalculator extends TestCase{
    public void testAdd(){
           Calculator calculator = new Calculator();
           int result = calculator.add(5,6);
           assertEquals(11, result);
    }
    public void  testSub(){
           Calculator calculator = new Calculator();
           int result = calculator.sub(6,5);
           assertEquals(1, result);
    }
}
// product code
public class Calculator{
    public int add (int number1, int number2){
           return number1 + number2;
    }
    public int sub (int number1, int number 2){
           return number1-number2;
    }
}
```


Refactoring

- One of the 12 XPxtudes
- Structured, disciplined method to rewrite or restructure existing code without changing its external behavior
- Activities and goals of good programming
  - Reduce duplicate code
  - Improve clarity
  - Make long methods shorter
  - Remove the use of hard-coded literal constants

Code smells…

- Duplicated code
- Big method
- Class with many instance variables
- Class with lots of code
- Strikingly similar subclasses
- Little or no use of interfaces in the design
- High coupling between many objects
- So many other ways bad code is written…

Simple refactorings

- 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

Refactoring example -- Before

```
public class player{
   private Piece piece;
   private Board board;
   private Die[] dice;

   public void takeTure(){
       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);
   }
}
```


Refactoring example -- after

```
public class player{
   private Piece piece;
   private Board board;
   private Die[] dice;

   public void takeTure(){
       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;
   }
}
```