So Far …

Part 1: OOAD Intro

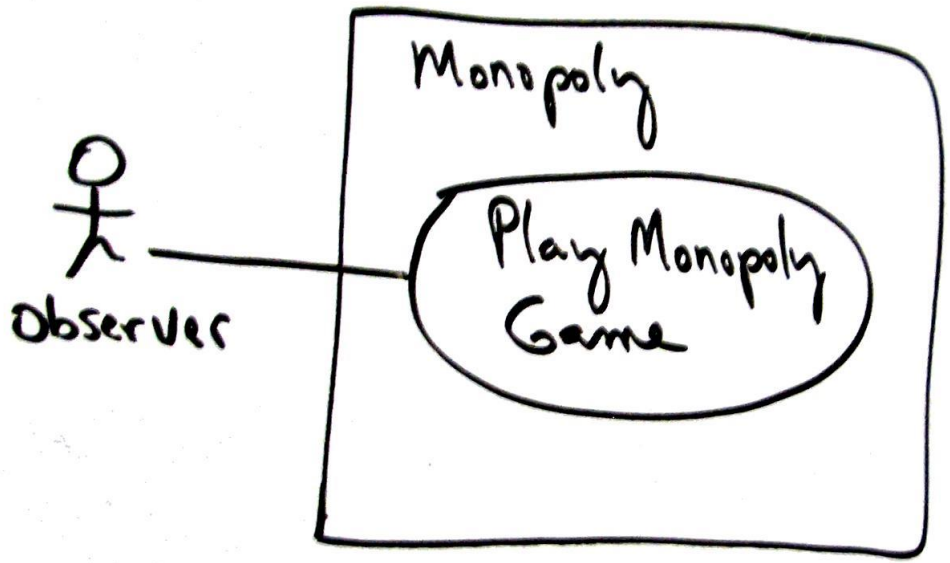
Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code

Use-Case Realizations
Monopoly

Monopoly Use Case Diagram



Monopoly Use Case Text:

- **Scope**: Monopoly Application
- **Level**: User Goal
- **Primary Actor**: Observer
- **Stakeholders**:
  Observer: easily observe game simulation output
- **Main Scenario**:
  1. Observer requests new simulation, enters num players
  2. Observer starts play.
  3. System displays game trace after each play
  4. *Repeat 3. until game over or Observer cancels*

Monopoly Use Case Text:

- Extensions:
  - **a*: At any time, system fails

    (System logs each move)

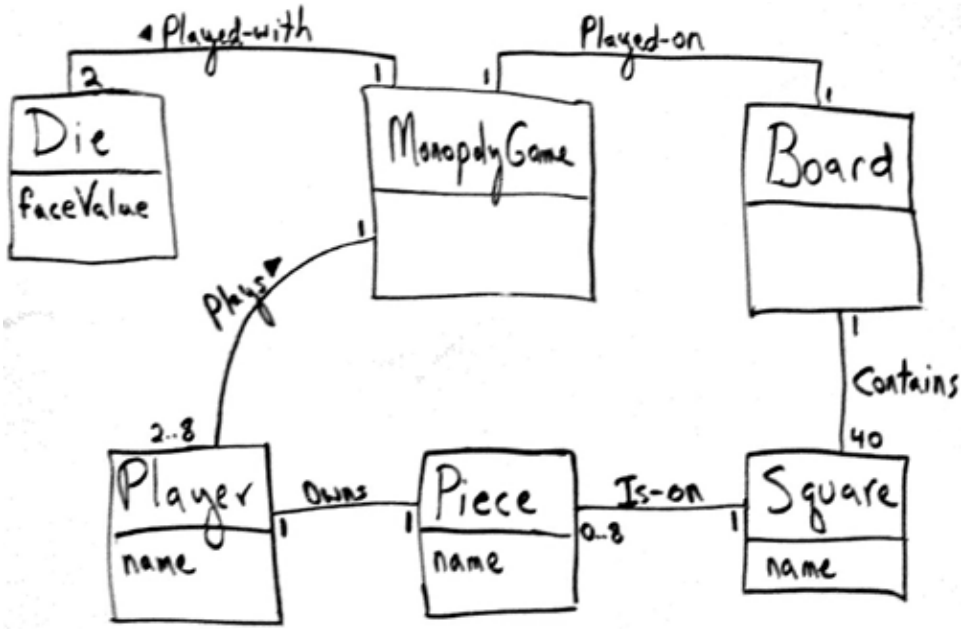
    1. Observer restarts system
    2. System detects failure and reconstructs correct state, continues
    3. Observer chooses to continue
- Special Requirements:
  - Provide graphical and text trace modes

Monopoly Game .

Domain rules or legal rules or business rules part of the Supplementary Specification (SS) more than scenarios

Trying to capture all the game rules in the **use case format is unnatural**

Monopoly Domain Model



Choosing Controller Object

MonopolyGame is a **root** object that represents the overall system

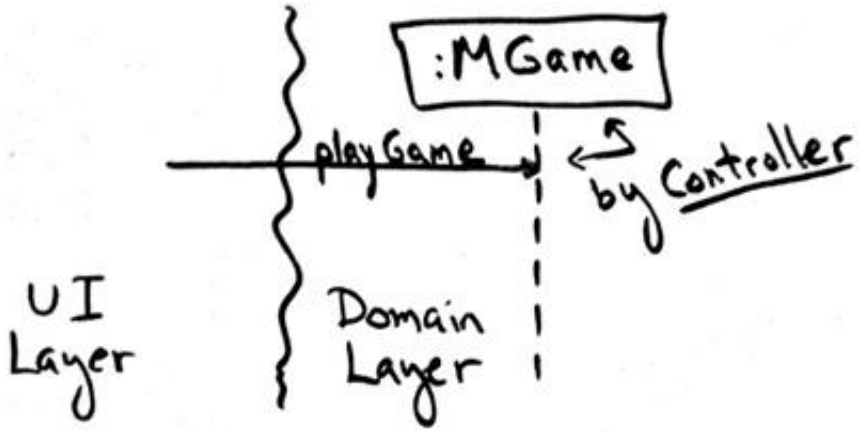
A system **controller** could also be called **MonopolyGameSystem**

An event **handler controller** could be **PlayMonopolyGameHandler**

Choosing a **root-object facade controller MonopolyGame** is satisfactory

- If few system operations and the **facade** controller is not taking on too many responsibilities (not becoming incohesive)

Applying the Controller Pattern



The Game-Loop Algorithm

Round— all the players taking one turn

Turn— a player rolling the dice and moving the piece

Game loop:

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

Who is Responsible for Controlling the Game Loop?

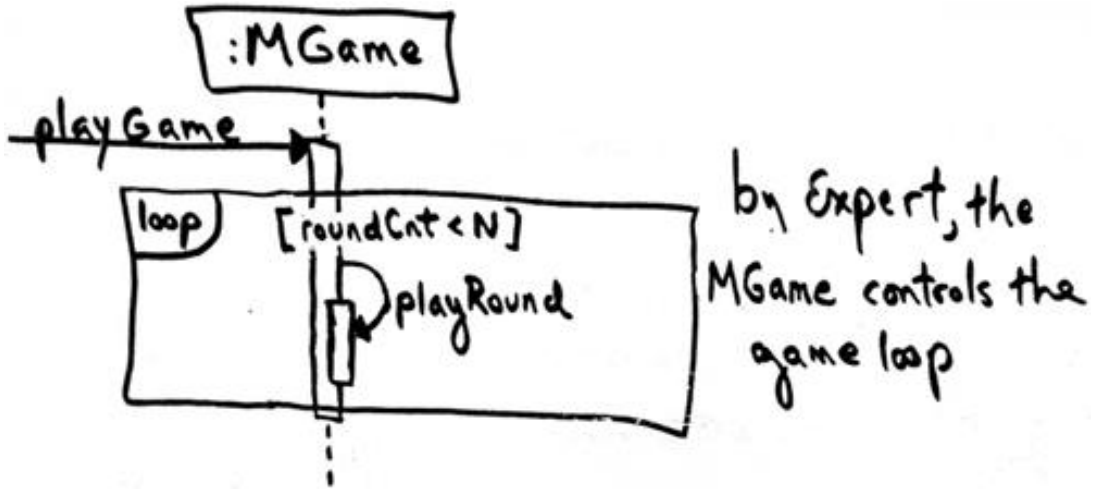
Information Needed	Who Has the Information?
The current round count	No object has it yet, but by LRG, assigning this to the **MonopolyGame** object is justifiable.
All the players (so that each can be used in taking a turn)	Taking inspiration from the domain model, **MonopolyGame** is a good candidate.

Who is Responsible for Controlling the Game Loop?

What object should control the game loop to manage the rounds?

By the Expert Pattern, the ***playRound*** method can be managed by the **MonopolyGame**

Who is Responsible for Controlling the Game Loop?



Who Takes a Turn?

Logical controller for a turn is a **Player**

- In real world, a human player makes all decisions for the game

OO designs are not one-to-one simulations of a real domain

Same person can perform many roles, **Cashier** in a store object would do almost everything!

- A violation of High Cohesion and Low Coupling. Big fat objects.

OO designs **distribute responsibilities** among many objects

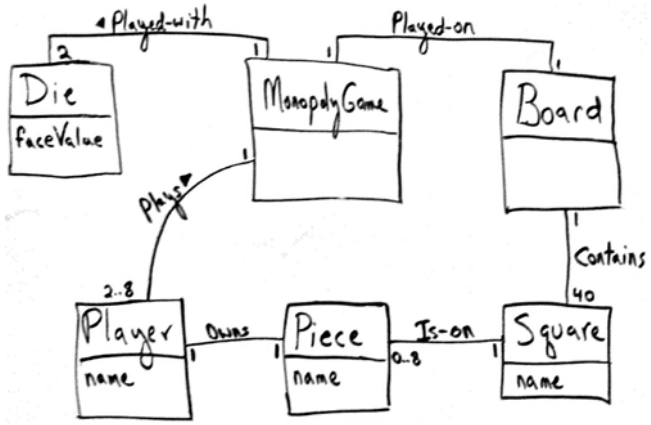
Single Purpose Principle

Organize **behavior** into **chunks** for high cohesion and low coupling

Good OO method design encourages small methods with a single purpose

“Take a Turn” Responsibility

Information Needed	Who Has the Information?
Current location of the player (starting point of a move)	Taking inspiration from the domain model, a **Piece** knows its **Square** and a **Player** knows its **Piece**. Therefore, a **Player** could know its location by LRG
The two **Die** objects (to roll them and calculate their total)	Taking inspiration from the domain model, **MonopolyGame** is a candidate since we think of the dice as being part of the game
All the squares—the square organization (to be able to move to the correct new square)	By LRG, **Board** is a good candidate



“Take a Turn” Responsibility

Three partial information experts for the “**take a turn**” responsibility: **Player**, **MonopolyGame**, and **Board**.

Who gets to start “Take a Turn”?

Evaluations and trade-offs an OO developer may consider:

1. Multiple experts? Place responsibility for dominant (majority of information), tends to best support Low Coupling

Unfortunately, all equal, no dominant expert

“Take a Turn” Responsibility

2. Coupling and cohesion impact, choose best

MonopolyGame is doing some work, more work impacts its cohesion

Player and **Board** are not doing anything yet, a tie.

“Take a Turn” Responsibility

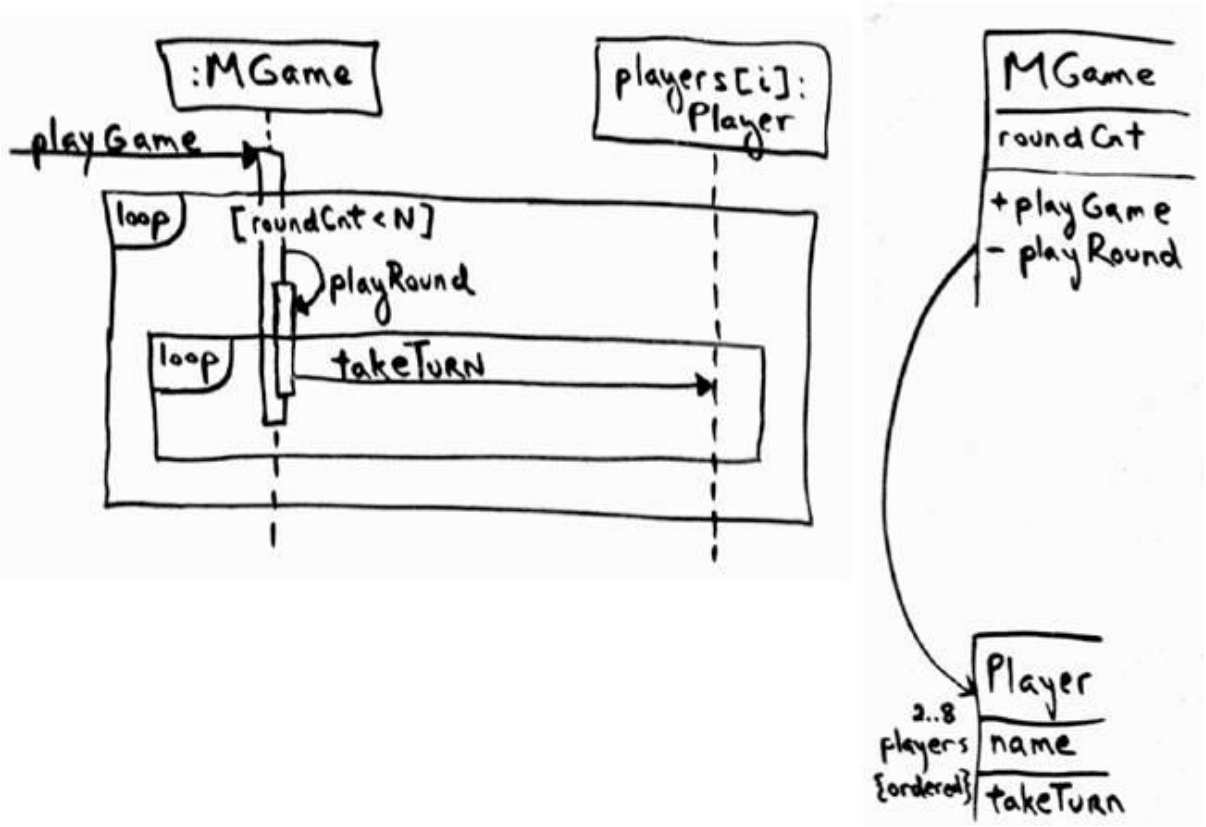
3. Probable future evolution of the software + **1** and **2**

Consider more game **rules**, taking a turn can involve:

- Buying a property that the player lands on, if the **Player** has enough money, what object would be expected to know a player's cash total?
  - Answer: a **Player** (by LRG)
- If its color fits in with the player's “color strategy.” What object would be expected to know a player's color strategy?
  - Answer: a **Player** (by LRG, as it involves a player's current holdings of properties)

Player turns out to be a good candidate, justified by Expert when considering few game rules

“Take a Turn” Responsibility



“Taking a Turn” Coordinator?

Calculating a random number total between 2 and 12

Who rolls the dice?

By LRG:

- Create a **Die** object with a faceValue attribute
- Changing information in the **Die**, so by Expert **Die** should be able to ***roll*** itself (domain vocabulary)
- And answer its faceValue.

“Taking a Turn” Coordinator?

Moving the player's piece from an old location to a new square location.

Who knows any given new location?

By LRG:

- **Board** knows all its **Squares**
- By Expert a **Board** knows new square location, given old square location + the dice total

“Taking a Turn” Coordinator?

Who need to know the new square location?

By LRG,

- **Player** knows its **Piece**, **Piece** knows its **Square** location (**Player** know its **Square** location)
- By Expert a **Piece** receives the new location from its owner, **Player**

“Taking a Turn” Coordinator?

Who Calculates total dice?

By LRG,

- **Player** or **MonoplyGame**
- **Player** knows its **Piece**, **Piece** knows its **Square** location (**Player** know its **Square** location)
- By Expert a **Player** calculates the total

“Taking a Turn” Coordinator?

Since the **Player** is responsible for taking a turn,

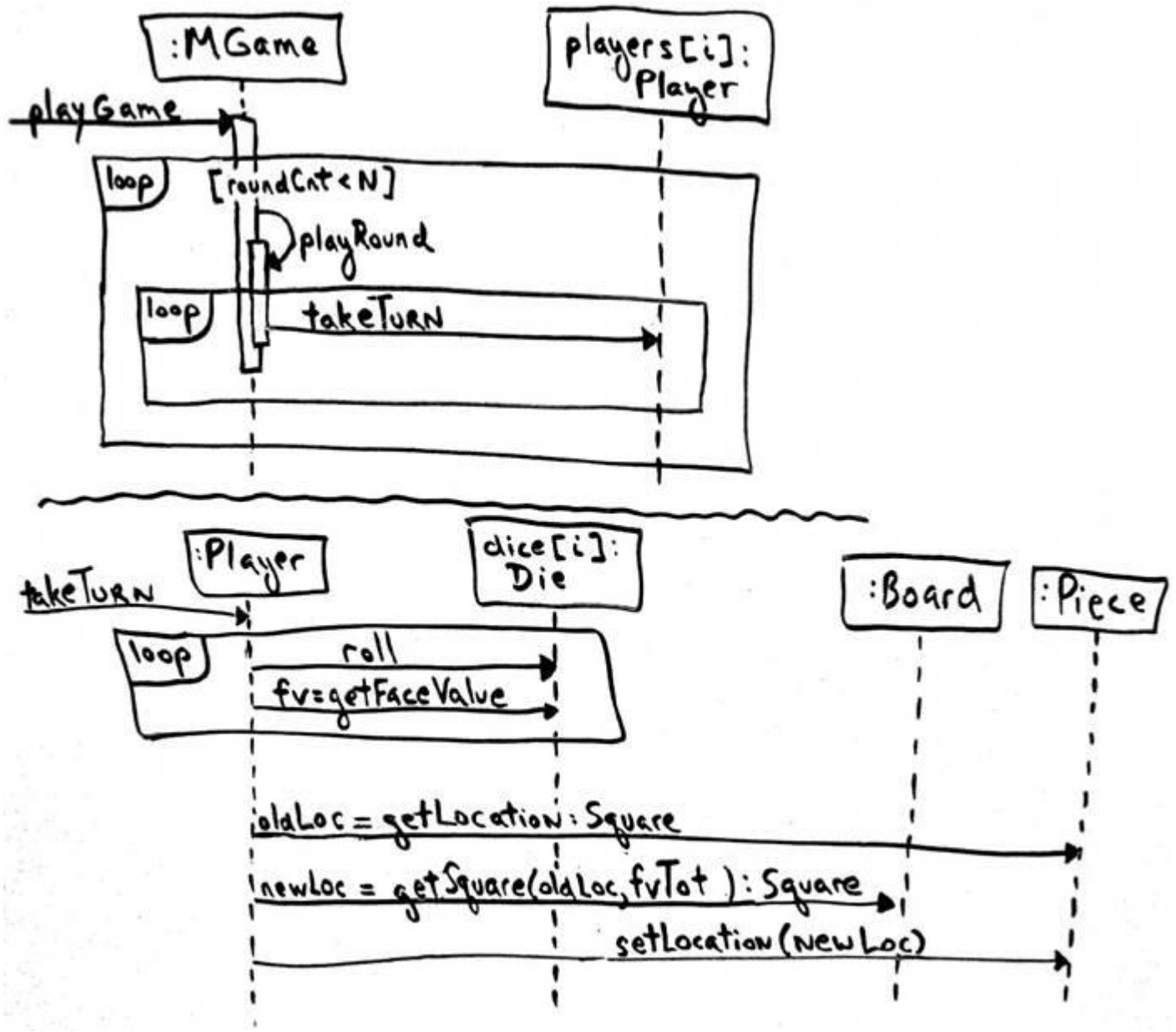
the **Player** should coordinate

Implies collaboration with the **Die**, **Board**, and **Piece** objects

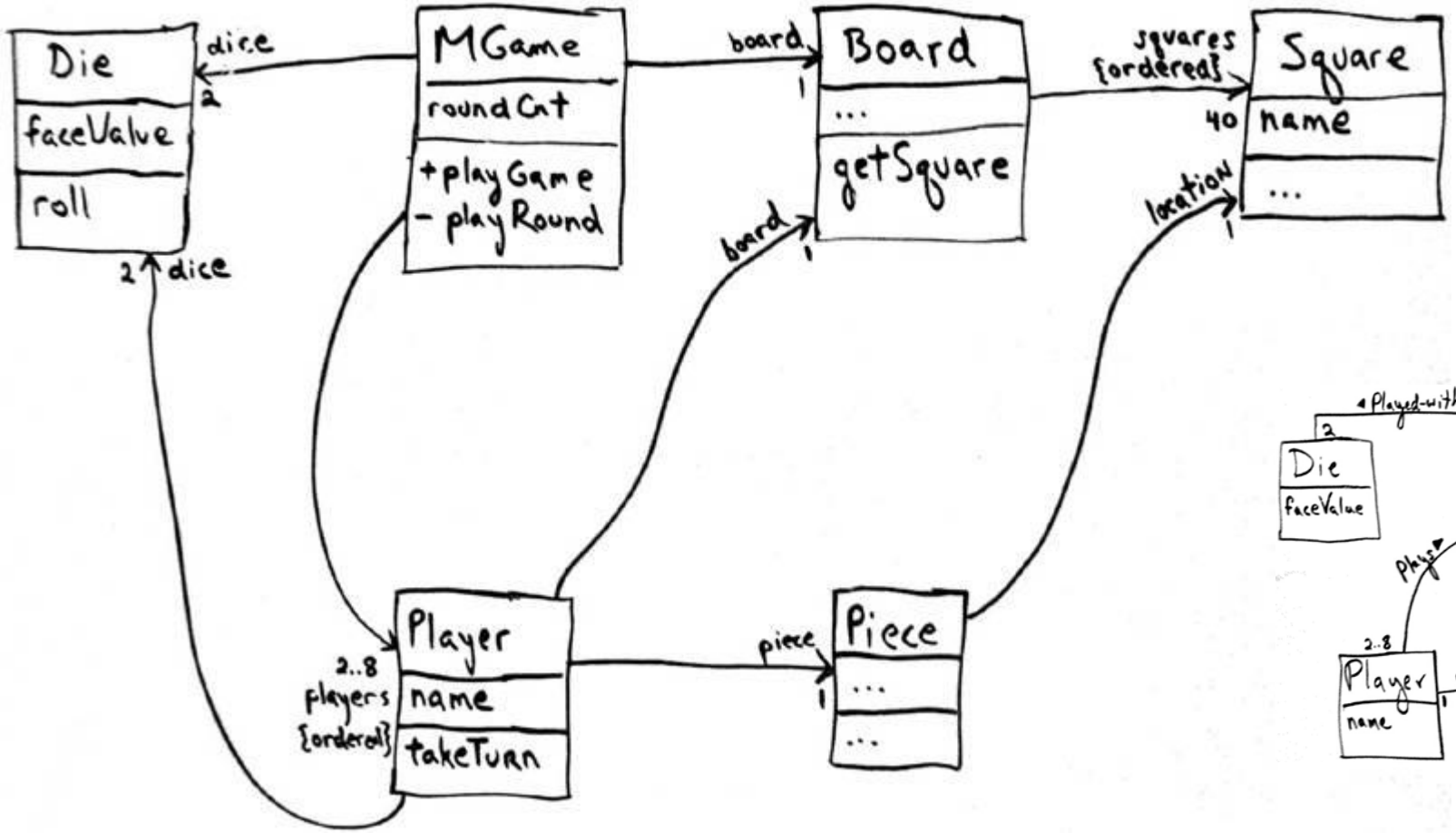
Implies a visibility to all

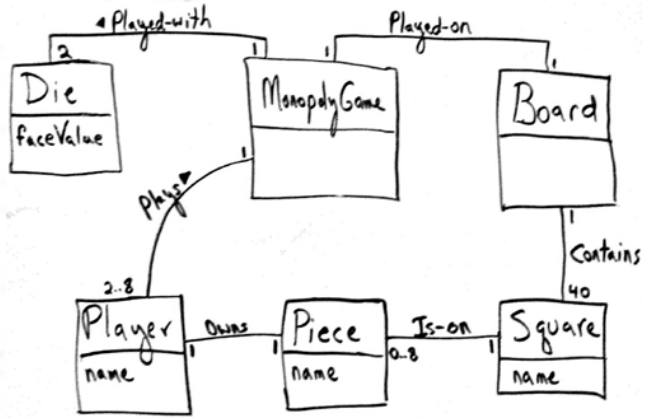
Initialize **Player** during ‘startup’ with permanent references to those objects

Design of playGame



Design of playGame





Command-Query Separation Principle (CQS or CQRS)

```
// style #1; used in the official solution
public void roll() {
	faceValue = // random num generation
}

public int getFaceValue() {
	return faceValue;
}
```

Command-Query Separation Principle (CQS or CQRS)

```
// style #2; why is this poor?


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

Command-Query Separation Principle (CQRS)

Style #2 though widely used; it violates CQRS classic OO design principle for methods

CQRS:

- A **Command method** performs an action with side effects (updating, coordinating, …) **changes** the **state** of objects, and is void

- A **Query method** returns data to the caller and has no side effects—no change to the state of any objects

BUT a method should not be both!

roll method is a command—it has the side effect of changing the state of the **Die**'s faceValue, no return of the new faceValue

Command-Query Separation Principle (CQRS), Why?

CQRS makes designs simpler to understand and anticipate

Consistency!

For example, if an application consistently follows CQRS, you know:

- A **query** or **getter** method isn't going to **modify** anything and
- A command isn't **going** to **return anything**

Principle of Least Surprise

Nice to rely on, as the alternative can be a nasty surprise—violating the **Principle of Least Surprise** in software development

“Start Up” Use Case

The initialize system operation occurs

Root object that will be the creator of some other objects

MonopolyGame is a root object

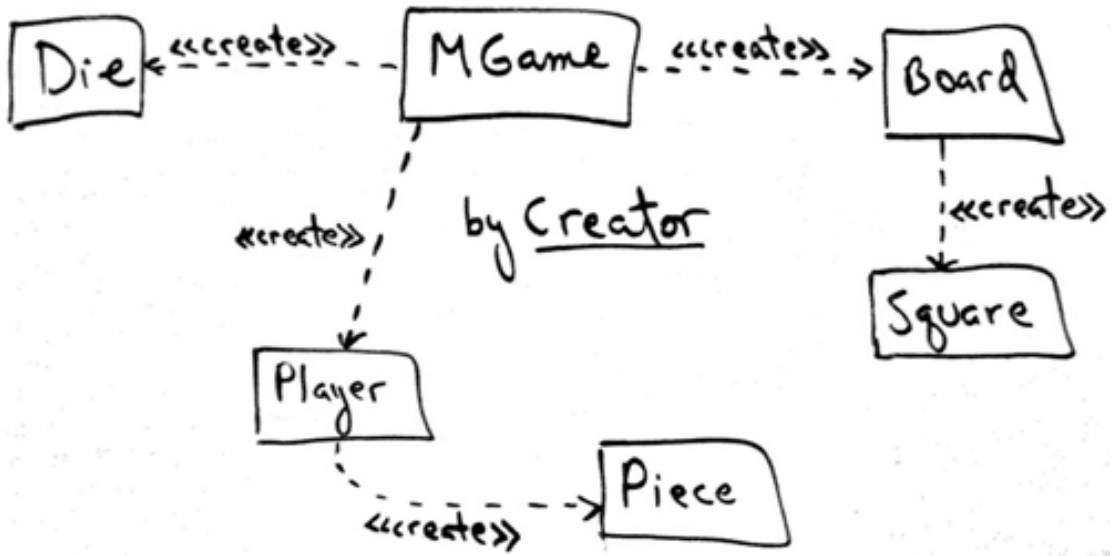
By Creator,

MonopolyGame creates **Board**, **Player**s, **Die** (Dice)

Board creates **Square**s

Player creates **Piece**

“Start Up” Use Case



UP Design Model Use case realizations

Inception—

Design Model and Use Case Realizations will not usually be started until elaboration because they involve detailed design decisions, which are premature during inception

Elaboration—

Use case realizations for most architecturally significant or risky scenarios of the design

UML diagramming some scenarios, and not necessarily in complete and fine-grained detail

Interaction diagrams for the **key use case realizations**

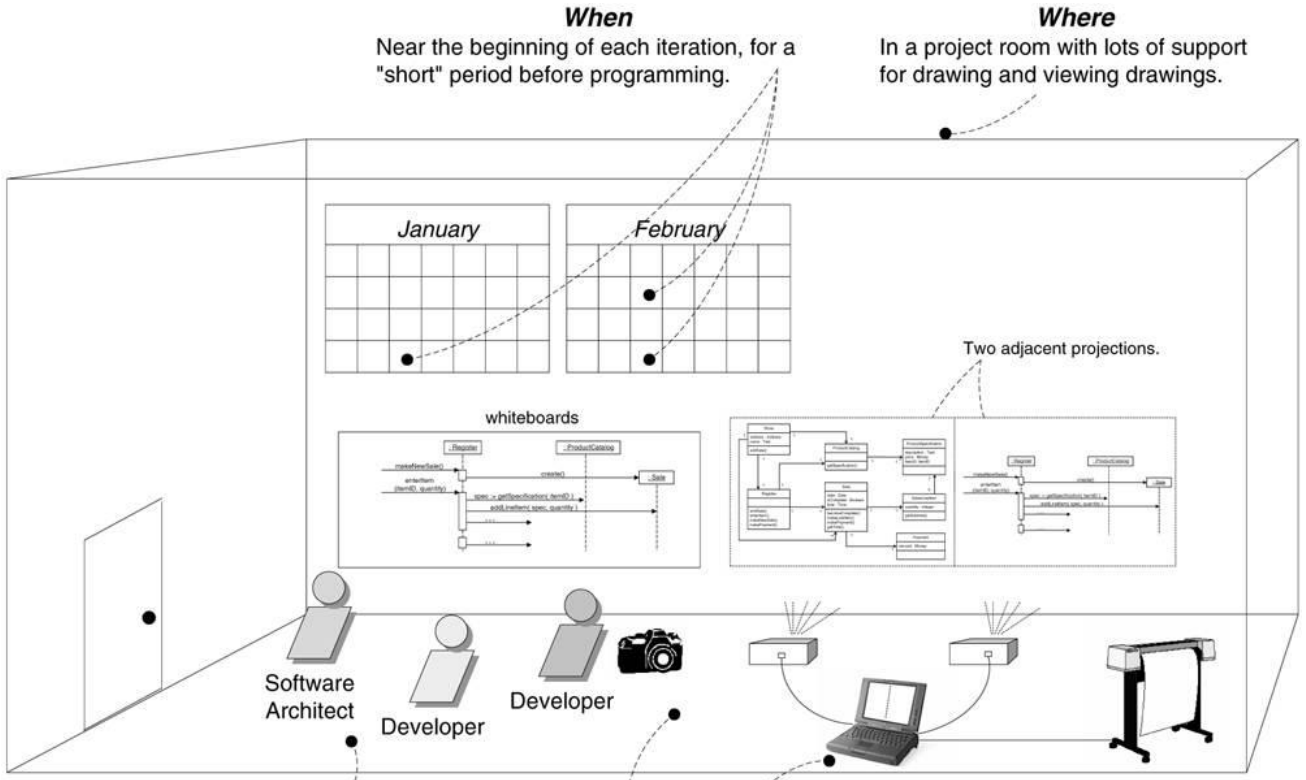
Construction—

Use case realizations are created for remaining design problems

Sample UP Artifacts and timing

Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration→	I1	E1..En	C1..Cn	T1..T2
Business Modeling	Domain Model		s		
Requirements	Use Case Model (SSDs)	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	

Iterative, Evolutionary OOD Process—Setting Context



So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code
- Test-Driven Development and Refactoring

Designing for Visibility

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

A mathematician is a device for turning coffee into theorems.

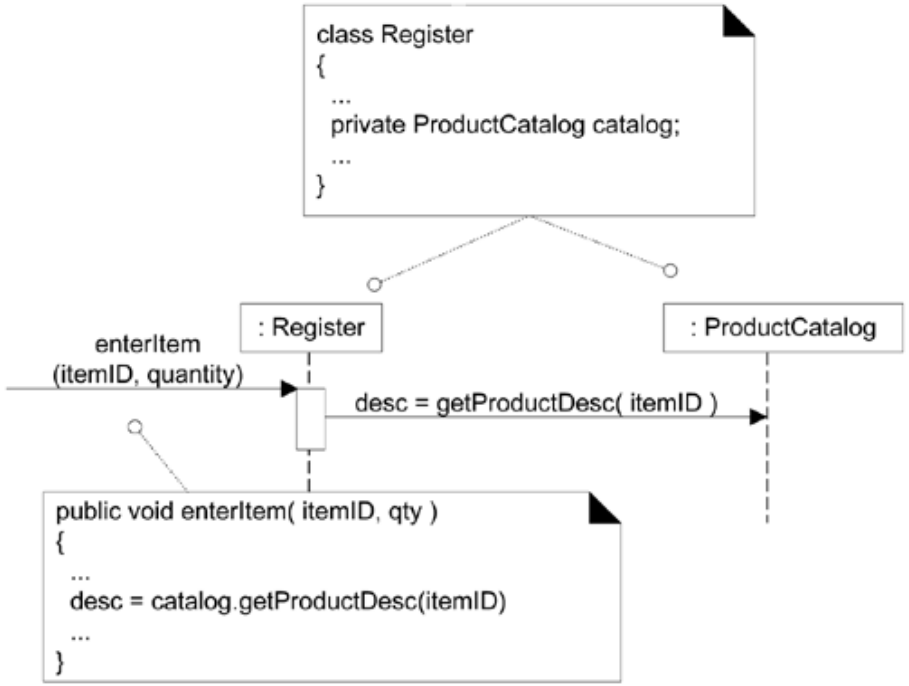
—Paul Erdös

Visibility Between Objects

getProductDescription message sent from a **Register** to a **ProductCatalog**, Implies

ProductCatalog instance is visible to the **Register** instance

Visibility Between Objects



What is Visibility?

Ability of an object to “see” or have a reference to another object

It is **Scope**, Is one resource within the scope of another?

Visibility, Object A To Object B

1. **Attribute visibility**—B an attribute of A

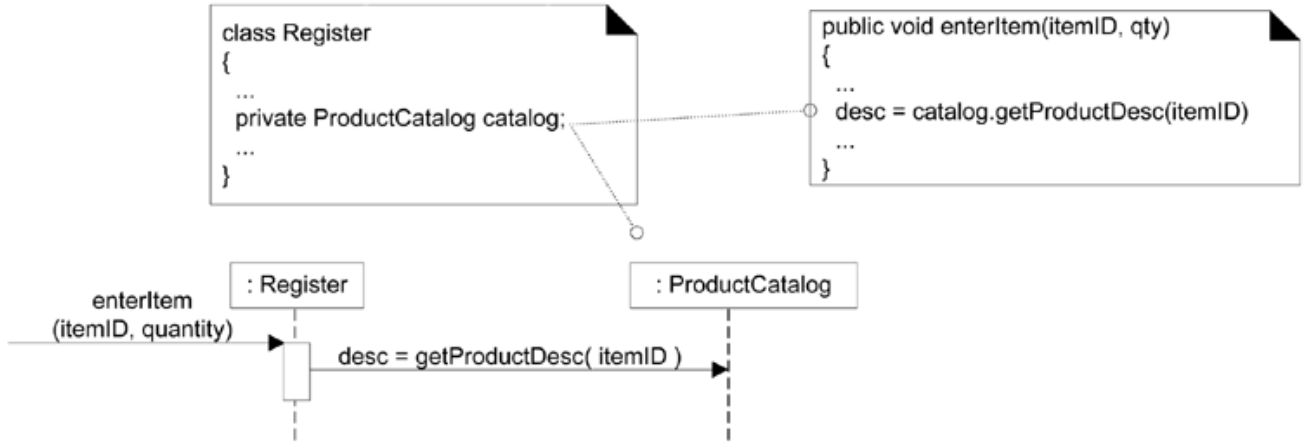
2. **Parameter visibility**—B a parameter of a method of A

3. **Local visibility**—B a local object in a method of A

4. **Global visibility**—B globally visible

Attribute Visibility —Permanent

```
public class Register{
        private ProductCatalog catalog;
}
```



A To Send A Message To B, B Must Be Visible To A

A message is sent from a **Register** instance to a **ProductCatalog** instance

Register must have visibility to the **ProductCatalog**

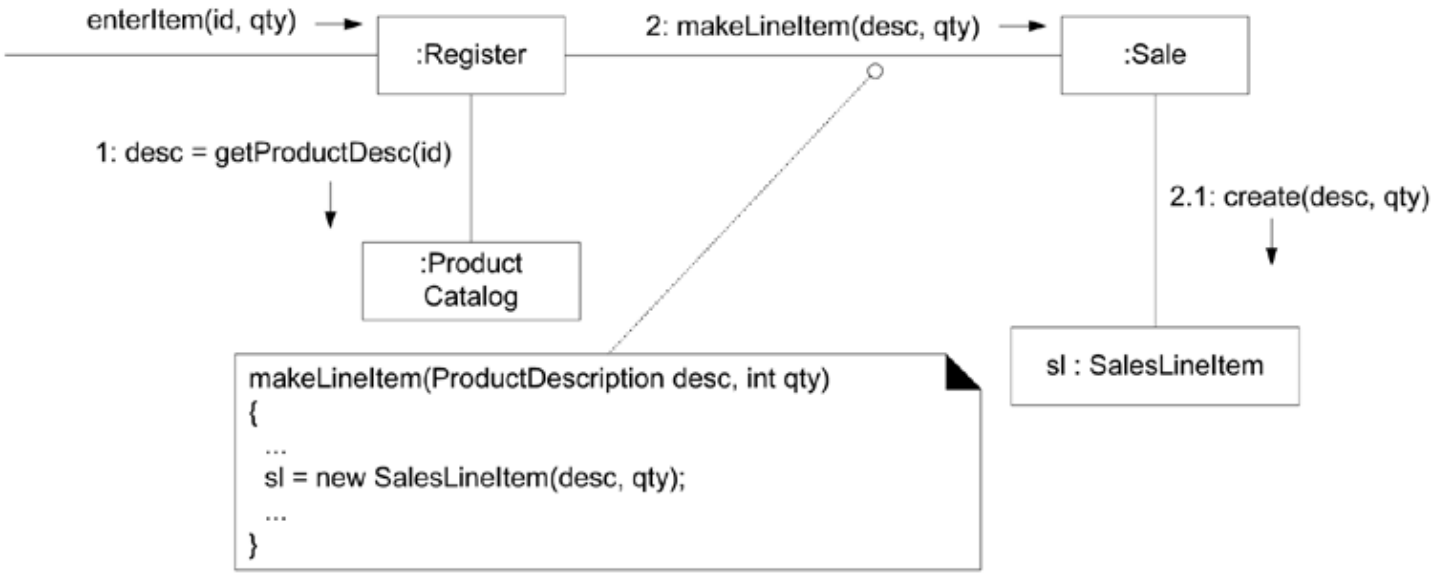
ProductCatalog instance is maintained as an attribute in **Register**

Parameter Visibility —Temporary

makeLineItem message sent to a **Sale**, **ProductDescription** is passed as a parameter

Within the scope of the ***makeLineItem*** method, the **Sale** has parameter visibility to a **ProductDescription**

Parameter Visibility (Temporary)



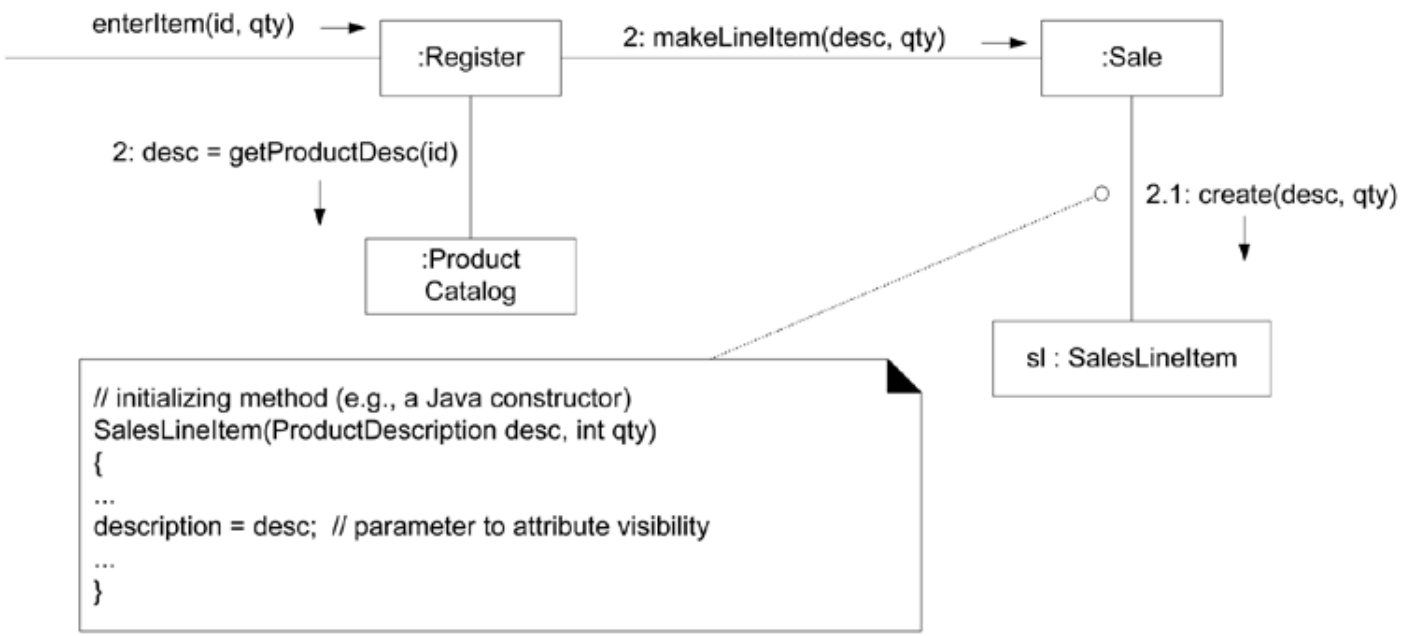
Convert Parameter Visibility Into Attribute Visibility

Common to transform parameter visibility into attribute visibility

Sale creates a new **SalesLineItem**, it passes the **ProductDescription** in to its initializing method (in C++ or Java, this would be its constructor)

Within the initializing method, the parameter is assigned to an attribute, thus establishing attribute visibility

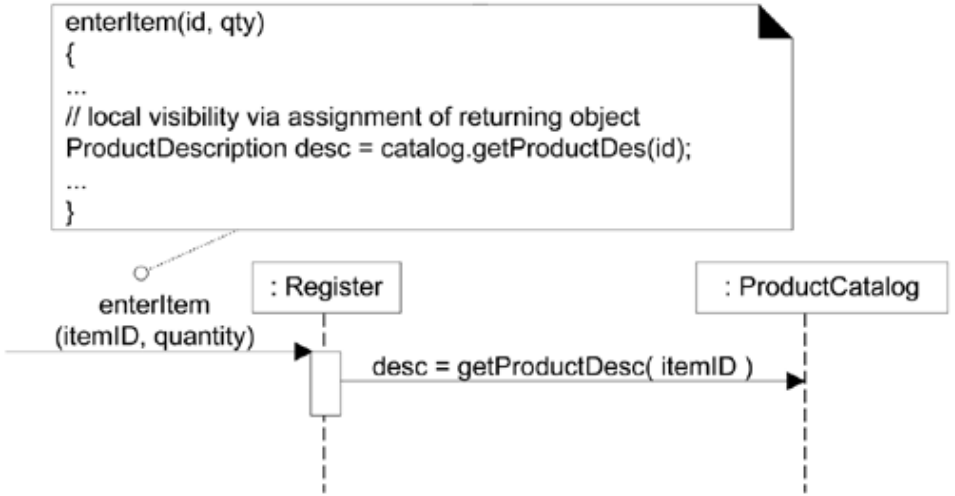
Parameter Visibility Into Attribute Visibility



Local Visibility —Temporary

1. Create a new local instance and assign it to a local variable

2. Assign the returning object from a method invocation to a local variable



Global Visibility —Permanent

One way to achieve global visibility is to assign an instance to a global variable, which is possible in some languages, such as C++, but not others, such as Java

Singleton pattern

Mapping Designs to Code

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Beware of bugs in the above code; I have only proved it correct, not tried it.

—Donald Knuth

Implementation Model

The **UML artifacts** created during the design work:

- The interaction diagrams, and
- Design class diagrams (DCDs)

Will be used as **input** to the **code generation** process

Programming and Iterative, Evolutionary Development

Requirements (UCs) + OOA + OOD + OO programming power is in providing an end-to-end roadmap, **requirements-to-code**

Doesn't mean no prototyping or

design-while-programming,

Doesn't mean road will be smooth,
or can simply be mechanically followed—plenty of variables

BUT having a **roadmap** provides a starting point for experimentation and discussion

Creativity and Change During Implementation

Decision-making and creativity showed up during design

At small scale, code generation (translation process) can be almost mechanical

Larger scale, programming is not a trivial code generation step—it's quite the opposite!

Realistically, results generated during design modeling are an incomplete first step

Creativity and Change During Implementation

Programming + testing + endless changes will be made, and detailed problems will be uncovered and resolved, reality check!

Ideas and understanding (not the diagrams or documents!) generated during OO design modeling will provide a great base that scales up, if Done well!

Creativity and Change During Implementation

BUT expect and plan for lots of change and deviation from the design during programming

"Pragmatic Attitude" in iterative and evolutionary methods

Mapping Designs to Code

Implementation in an OO language is coding:

- Interface definitions
- Class definitions
- Method definitions
- Attribute definitions

Creating Class Definitions from DCDs

DCDs contains class or interface name, superclasses, operation signatures, and attributes of a class

Sufficient to create a basic class definition in OOP

DCD was drawn in a UML tool; it can auto generate the basic skeleton for class definitions from diagrams

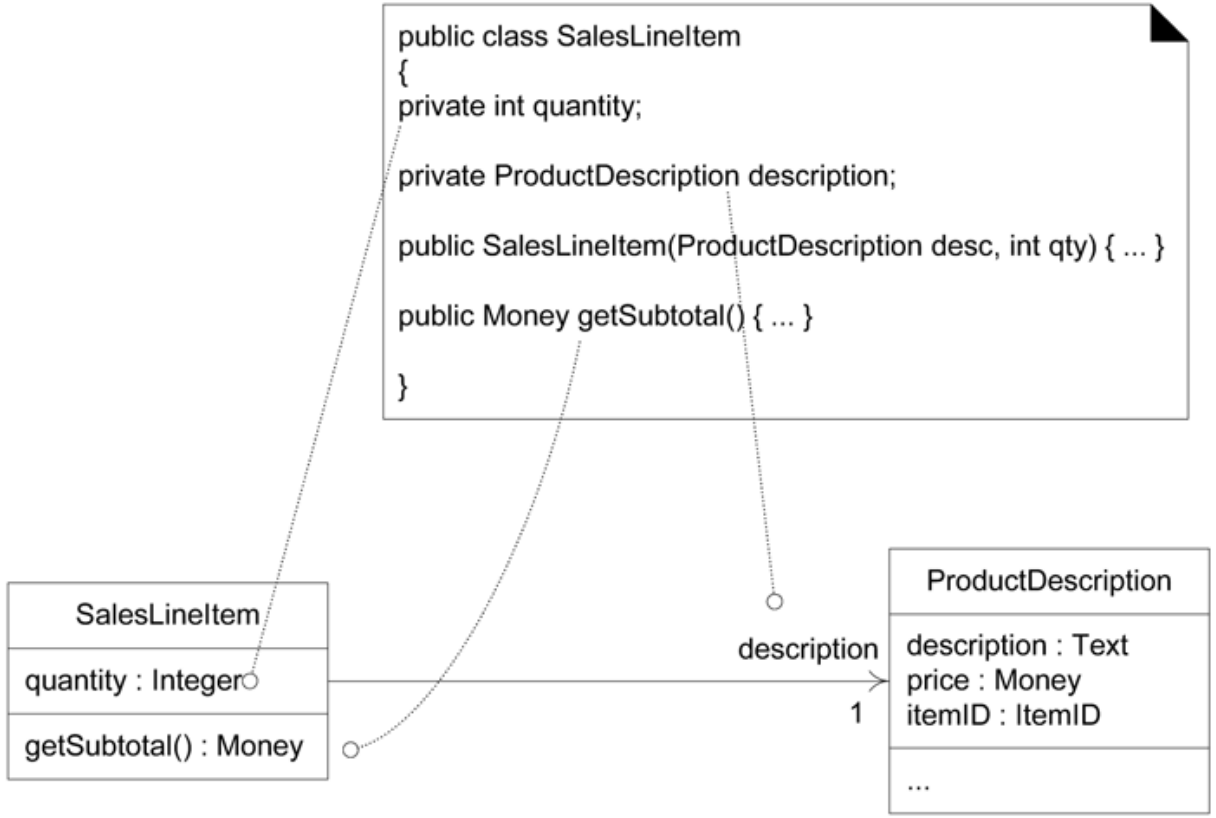
Defining a Class with Method Signatures and Attributes

DCD, a **mapping** to the attribute definitions and method signatures for a code definition of **SalesLineItem** is straightforward

Except the constructor ***SalesLineItem***(...), it is derived from the ***create(****desc*, *qty****)*** message sent to a **SalesLineItem** in the ***enterItem*** interaction diagram

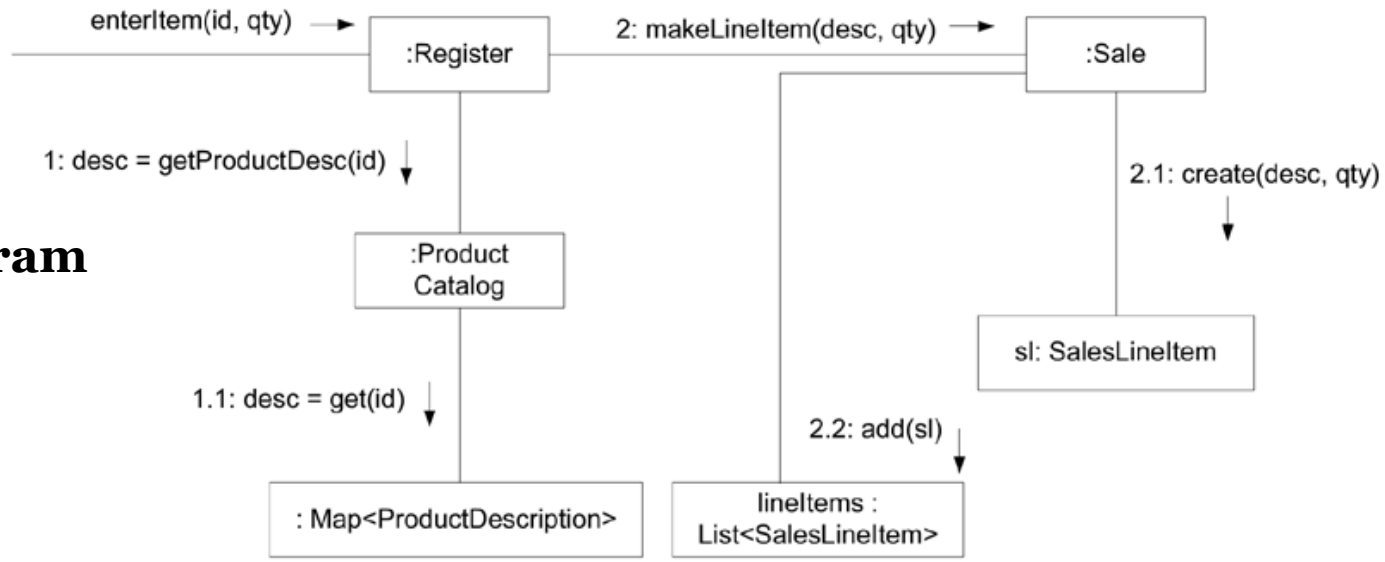
Create method is often excluded from the class diagrams, language specific

Defining a Class (DCD) with Method Signatures and Attributes



Creating Methods from Interaction Diagrams

***enterItem* interaction diagram**

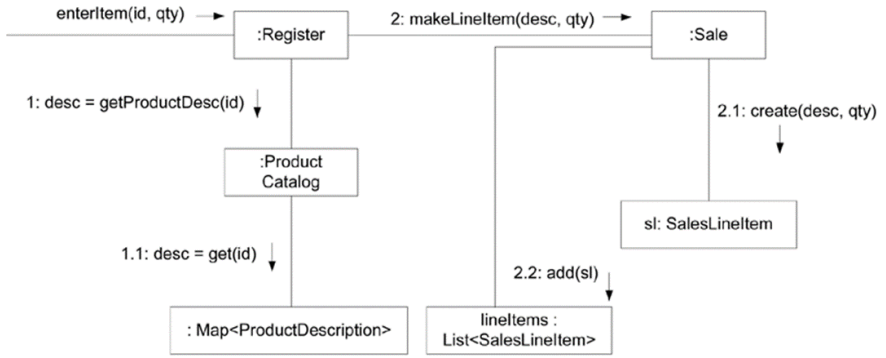


Register.*enterItem* Method

The ***enterItem*** message is sent to a **Register** instance; therefore, the ***enterItem*** method is defined in class **Register**

Each **sequenced message** within a method, as shown on the interaction diagram, is mapped to a statement in a **method** (e.g. ***enterItem***)

Register.*enterItem* Method



Message 1:

A ***getProductDescription*** message is sent to the **ProductCatalog** to retrieve a **ProductDescription**.

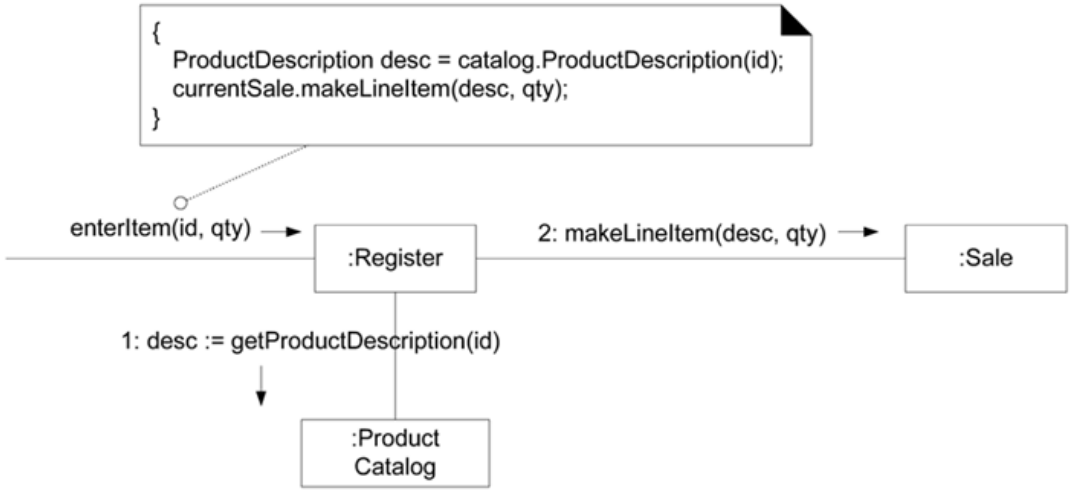
```
ProductDescription desc =
catalog.getProductDescription(itemID);
```

Message 2:

The ***makeLineItem*** message is sent to the **Sale**.

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

Register.*enterItem* Method



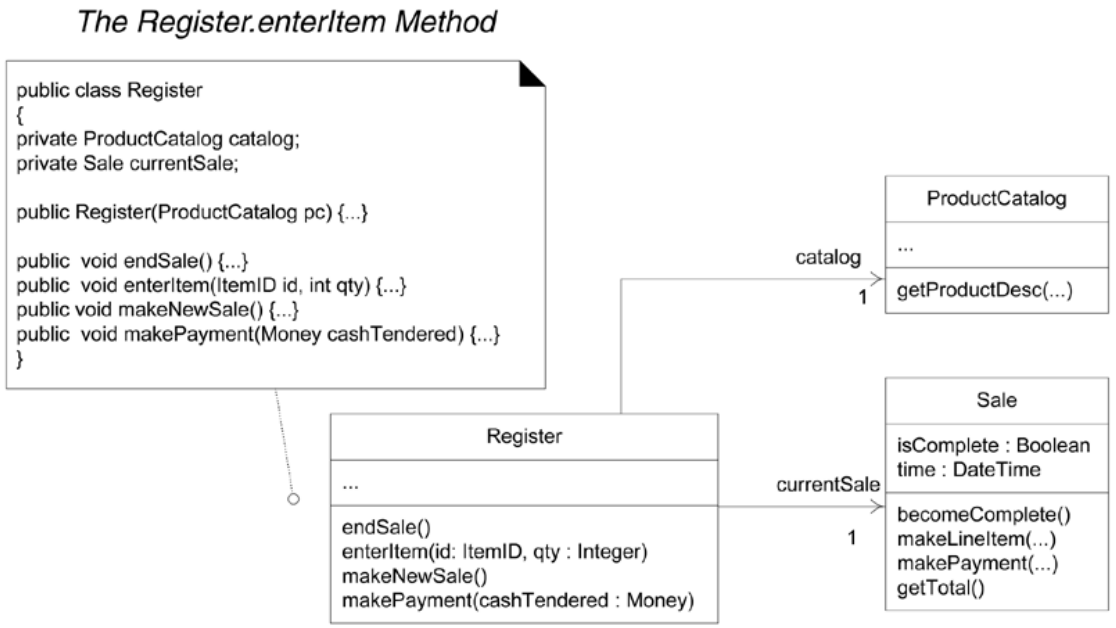
Register.*enterItem* Method

```
public void enterItem(ItemID itemID,int qty){
    /** A getProductDescription message is sent to the ProductCatalog to
        retrieve a ProductDescription */
    ProductDescription desc=catalog.getProductDescription(itemID);

    /** The makeLineItem message is sent to the Sale.*/
    currentSale.makeLineItem(desc,qty);
}
```

Creating Methods from Static Diagrams

Register Class



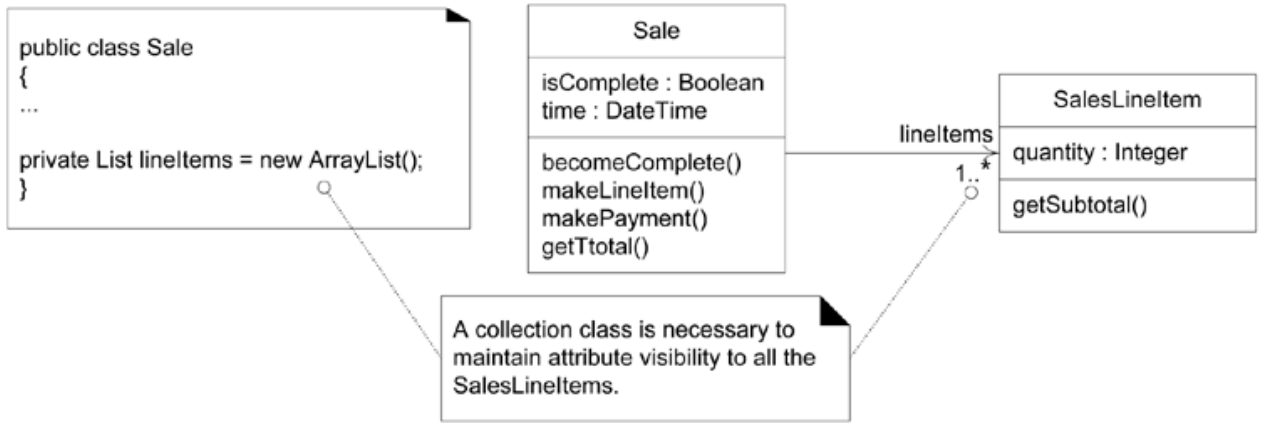
Collection Classes in Code

One-to-many relationships are common. e.g., a **Sale** must maintain visibility to a group of many **SalesLineItem** instances

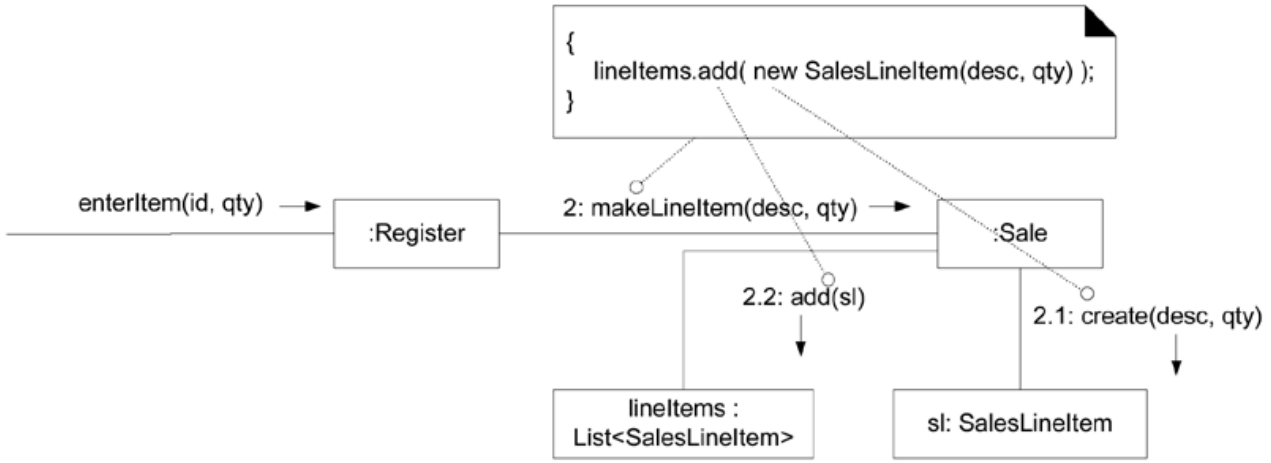
In OOP, a collection object, such as a List or Map, a simple array

Collection Classes in Code

lineItems attribute is declared in terms of its interface, *why*?



Defining the Sale.*makeLineItem* Method



Defining the Sale.*makeLineItem* Method

```
public void makeLineItem(ProductDescription desc,int quantity){
        lineItems.add(new SalesLineItem(desc,quantity));
}
```

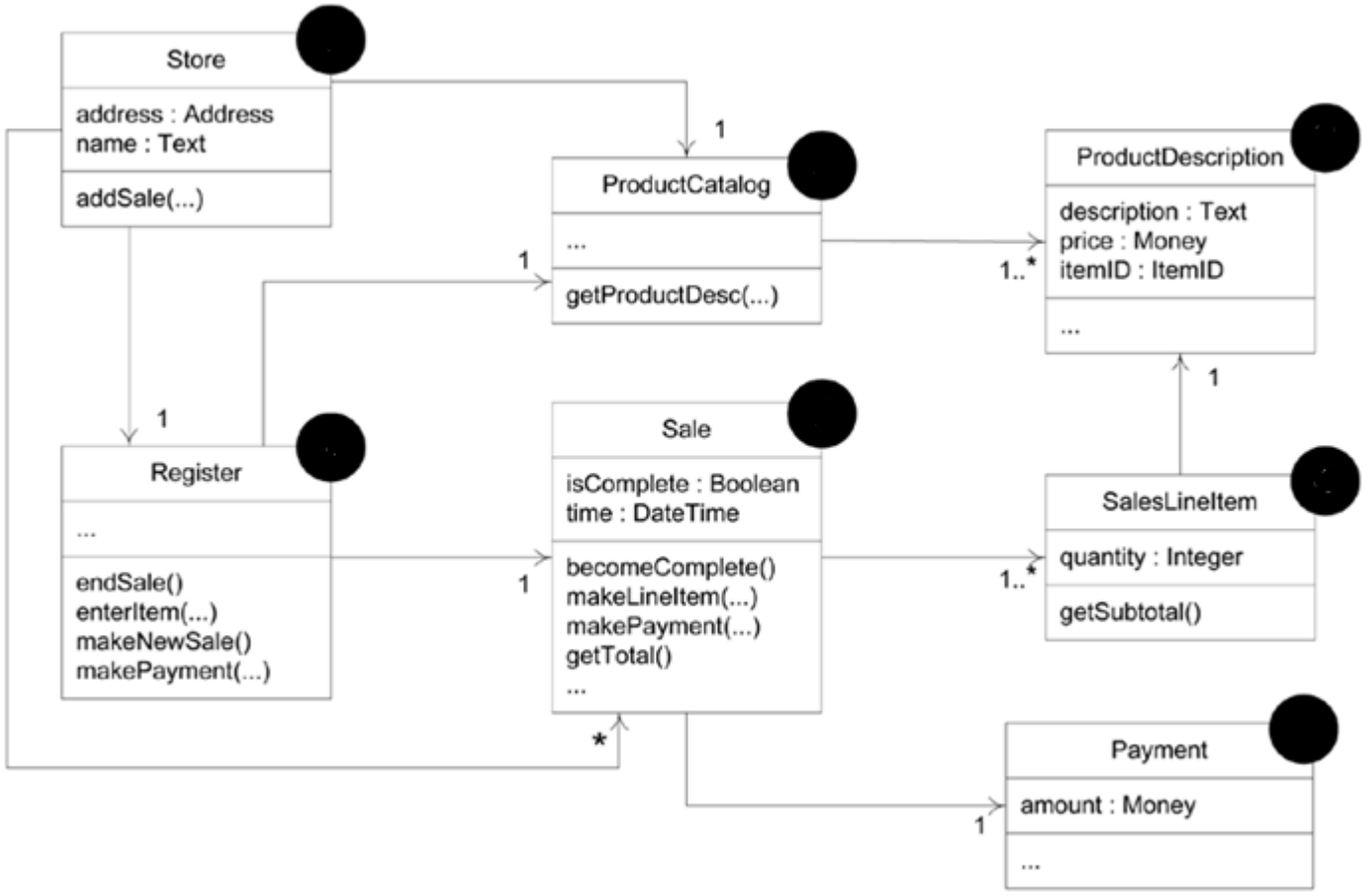
Order of Implementation

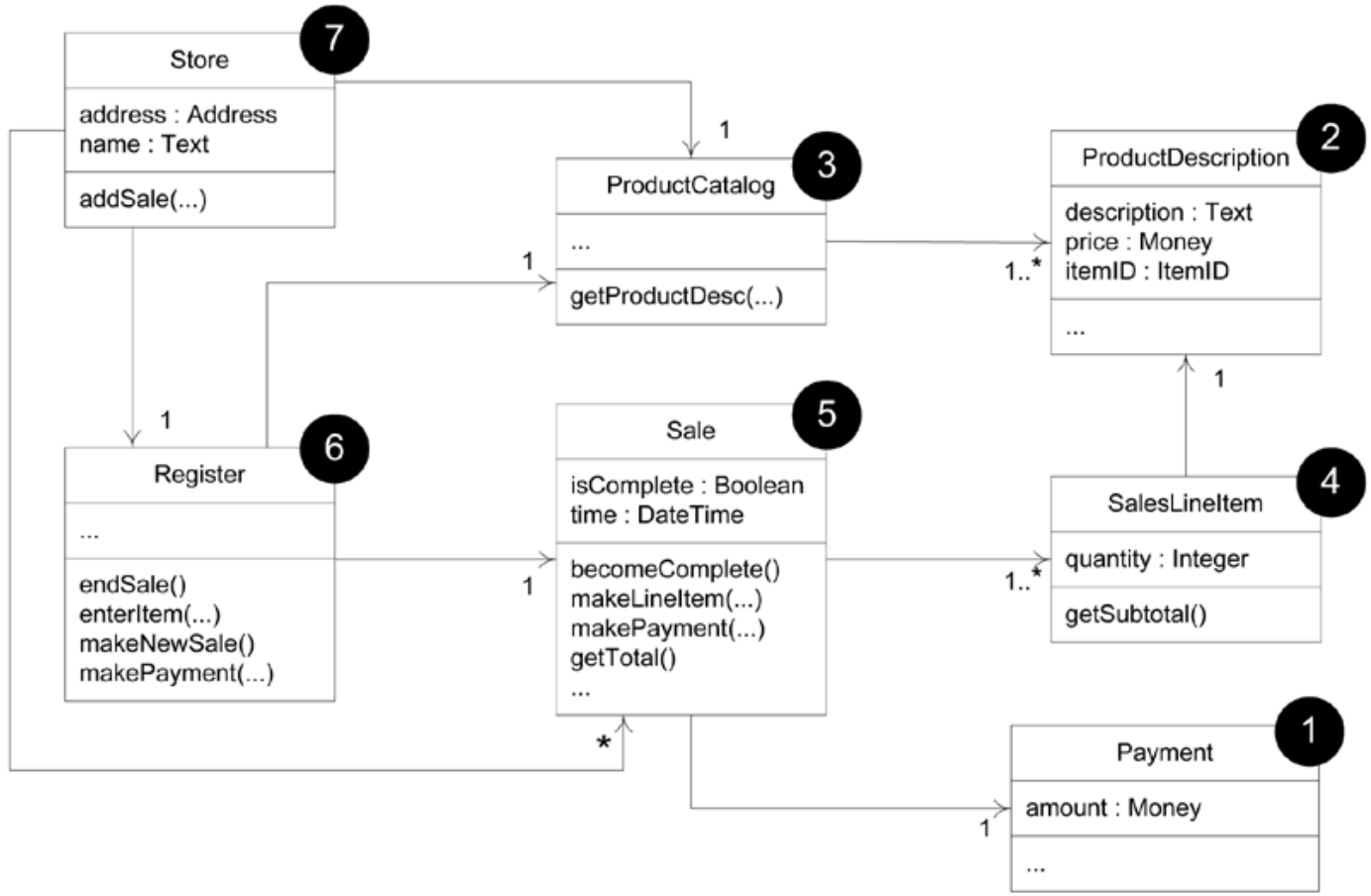
Classes need to be implemented (ideally, fully unit tested) from **least-coupled to most-coupled**

First, **Payment** or **ProductDescription**;

Next are classes only **dependent** of the prior implementations—**ProductCatalog** or **SalesLineItem**.

Order of Implementation




7
Store
address : Address
name : Text
addSale(...)
1
3
ProductCatalog
...
getProductDesc(...)
1
1..*
2
ProductDescription
description : Text
price : Money
itemID : ItemID
...
1
1
6
Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
1
5
Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
...
1..*
4
SalesLineItem
quantity : Integer
getSubtotal()
*
1
1
Payment
amount : Money
...

POS &

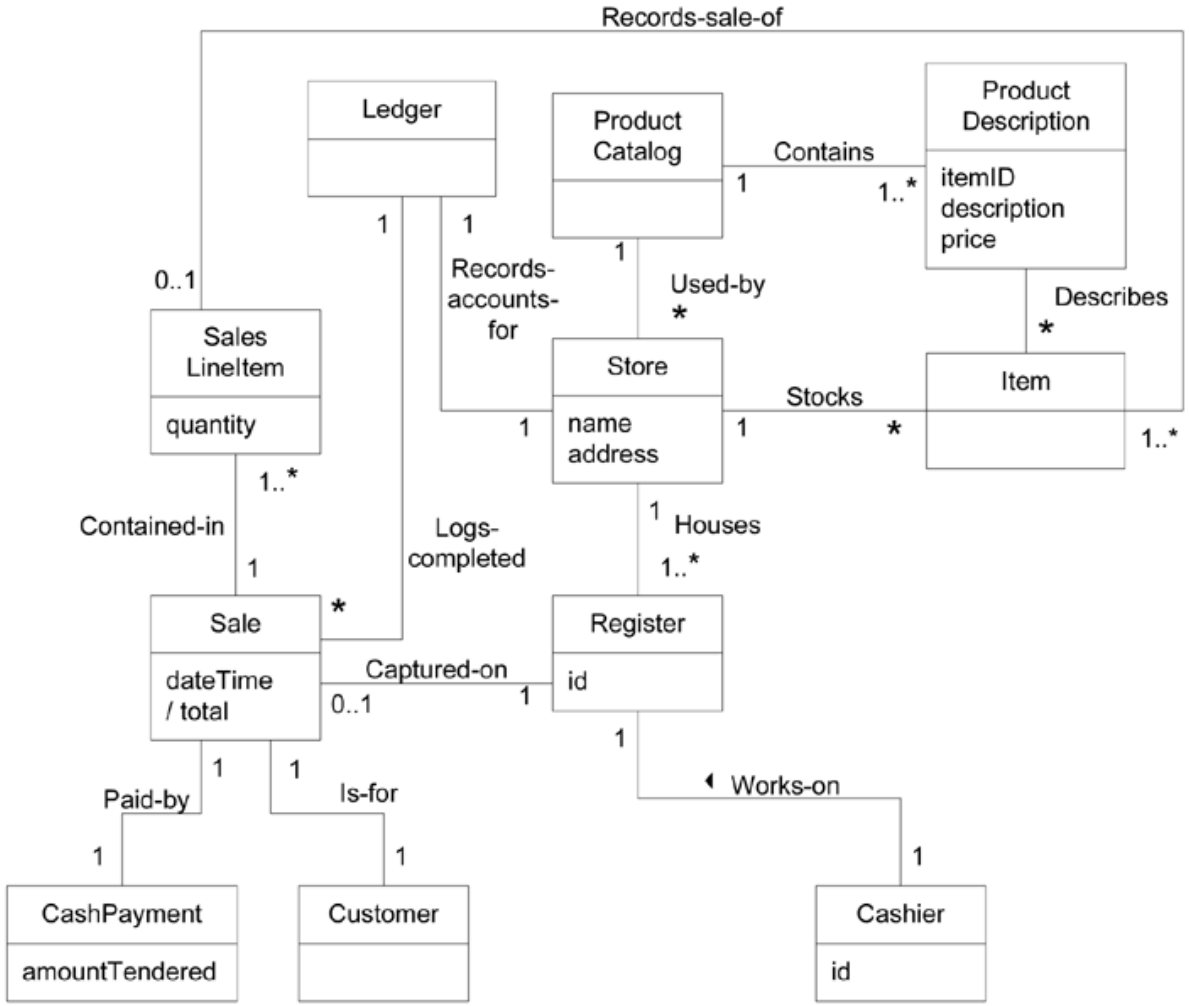
Monopoly

CODE . ☺

If reading clean modular code makes you happy, then coding would make you a creative writer, a novelist, a software engineer, a computer programist, a codeist!

—Abdulkareem Alali

POS Partial Domain Model



Class Payment

```
// all classes are probably in a package named
// something like:
package com.foo.nextgen.domain;


public class Payment
{
   private Money amount;

   public Payment( Money cashTendered ){ amount = cashTendered; }
   public Money getAmount() { return amount; }
}
```

Class ProductDescription

```
public class ProductDescription
{
   private ItemID id;
   private Money price;
   private String description;

   public ProductDescription
      ( ItemID id, Money price, String description )
   {
      this.id = id;
      this.price = price;
      this.description = description;
   }

   public ItemID getItemID() { return id;   }

   public Money getPrice() { return price; }
   public String getDescription() { return description; }
}
```

Class ProductCatalog

```
public class ProductCatalog
{

   private Map<ItemID, ProductDescription>
         descriptions = new HashMap()<ItemID, ProductDescription>;

   public ProductCatalog()
   {
      // sample data
      ItemID id1 = new ItemID( 100 );
      ItemID id2 = new ItemID( 200 );
      Money price = new Money( 3 );

      ProductDescription desc;
      desc = new ProductDescription( id1, price, "product 1" );
      descriptions.put( id1, desc );
      desc = new ProductDescription( id2, price, "product 2" );
      descriptions.put( id2, desc );
   }

   public ProductDescription getProductDescription( ItemID id )
   {
      return descriptions.get( id );
   }
}
```

Class SalesLineItem

```
public class SalesLineItem
{
   private int    quantity;
   private    ProductDescription    description;

   public SalesLineItem (ProductDescription desc, int quantity )
   {
      this.description = desc;
      this.quantity = quantity;
   }
   public Money getSubtotal()
   {
      return description.getPrice().times( quantity );
   }
}
```

Class Sale

```
public class Sale
{
   private List<SalesLineItem> lineItems =
                          new ArrayList()<SalesLineItem>;
   private Date date = new Date();
   private boolean isComplete = false;
   private Payment payment;

   public Money getBalance()
   {
      return payment.getAmount().minus( getTotal() );
   }

   public void becomeComplete() { isComplete = true; }

   public boolean isComplete() { return isComplete; }

   public void makeLineItem
      ( ProductDescription desc, int quantity )
   {
      lineItems.add( new SalesLineItem( desc, quantity ) );
   }

   public Money getTotal()
   {
      Money total = new Money();
      Money subtotal = null;

      for ( SalesLineItem lineItem : lineItems )
      {
         subtotal = lineItem.getSubtotal();
         total.add( subtotal );
      }
   return total;
   }


   public void makePayment( Money cashTendered )
   {
      payment = new Payment( cashTendered );
   }
}
```

Class Register

```
public class Register
{
   private ProductCatalog catalog;
   private Sale currentSale;

   public Register( ProductCatalog catalog )
   {
      this.catalog = catalog;
   }

   public void endSale()
   {
      currentSale.becomeComplete();
   }

   public void enterItem( ItemID id, int quantity )
   {
      ProductDescription desc = catalog.getProductDescription( id );

      currentSale.makeLineItem( desc, quantity );
   }

   public void makeNewSale()
   {
      currentSale = new Sale();
   }

   public void makePayment( Money cashTendered )
   {
      currentSale.makePayment( cashTendered );
   }


}
```

Class Store

```
public class Store
{
   private ProductCatalog catalog = new ProductCatalog();
   private Register register = new Register( catalog );

   public Register getRegister() { return register; }
}
```

POS &
Monopoly
CODE . ☺

Class Square

```
// all classes are probably in a package named
// something like:
package com.foo.monopoly.domain;

public class Square
{
   private String name;
   private Square nextSquare;
   private int index;

   public Square( String name, int index )
   {
      this.name = name;
      this.index = index;
   }

   public void setNextSquare( Square s )
   {
       nextSquare = s;
    }

   public Square getNextSquare(  )
   {
      return nextSquare;
    }

   public String getName(  )
   {
      return name;
    }


  public int getIndex()
  {
    return index;
  }
}
```

Class Piece

```
public class Piece
{

 private Square location;

 public Piece(Square location)
  {
    this.location = location;
  }

 public Square getLocation()
  {
    return location;
  }

  public void setLocation(Square location)
  {
    this.location = location;
  }
}
```

Class Die

```
public class Die
{
   public static final int MAX  = 6;
   private int           faceValue;

   public Die(  )
   {
      roll(  );
   }

   public void roll(  )
   {
      faceValue = (int) ( ( Math.random(  ) * MAX ) + 1 );
   }

   public int getFaceValue(  )
   {
      return faceValue;
   }
}
```

Class Board

```
public class Board
{

  private static final int SIZE    = 40;
  private List             squares = new ArrayList(SIZE);

  public Board()
  {
    buildSquares();
    linkSquares();
  }

  public Square getSquare(Square start, int distance)
  {
    int endIndex = (start.getIndex() + distance) % SIZE;
    return (Square) squares.get(endIndex);
  }

  public Square getStartSquare()
  {
    return (Square) squares.get(0);
  }

  private void buildSquares()
  {
    for (int i = 1; i <= SIZE; i++)
    {
      build(i);
    }
  }


  private void build(int i)
  {
    Square s = new Square("Square " + i, i - 1);
    squares.add(s);
  }

  private void linkSquares()
  {
    for (int i = 0; i < (SIZE - 1); i++)
    {
      link(i);
    }

    Square first = (Square) squares.get(0);
    Square last = (Square) squares.get(SIZE - 1);
    last.setNextSquare(first);
  }

  private void link(int i)
  {
    Square current = (Square) squares.get(i);
    Square next = (Square) squares.get(i + 1);
    current.setNextSquare(next);
  }
}
```

Class Player

```
public class Player
{
  private String name;
  private Piece  piece;
  private Board  board;
  private Die[]  dice;

  public Player(String name, Die[] dice, Board board)
  {
    this.name = name;
    this.dice = dice;
    this.board = board;
    piece = new Piece(board.getStartSquare());
  }
```

```
  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);

  }


  public Square getLocation()
  {
    return piece.getLocation();
  }

  public String getName()
  {
    return name;
  }

}
```

Class MonopolyGame

```
public class MonopolyGame
{
   private static final int ROUNDS_TOTAL  = 20;
   private static final int PLAYERS_TOTAL = 2;
   private List players = new ArrayList( PLAYERS_TOTAL )
   private Board   board  = new Board(  );
   private Die[]   dice   = { new Die(), new Die() };

   public MonopolyGame(  )
   {
      Player p;
      p = new Player( "Horse", dice, board );
      players.add( p );
      p = new Player( "Car", dice, board  );
      players.add( p );
   }

   public void playGame(  )
   {
      for ( int i = 0; i < ROUNDS_TOTAL; i++ )
      {
         playRound();
      }
   }

   public List getPlayers(  )
   {
      return players;
   }


   private void playRound(  )
   {
      for ( Iterator iter = players.iterator(  ); iter.hasNext(  ); )
      {
         Player player = (Player) iter.next();
         player.takeTurn();
      }
   }
}
```