# SWEN30006
# Software Design and Modelling

## GRASP:
## OO Design with Responsibilities

**Textbook: Larman Chapter 17**

*"Understanding responsibilities is key to good object-oriented design"*

*—Martin Fowler*


# Learning Objectives

On completion of this topic you should be able to:

- Understand the concepts of responsibility and responsibility-driven design
- Understand the advantages of using patterns
- Apply five of the GRASP principles or patterns for Object-Oriented Design.

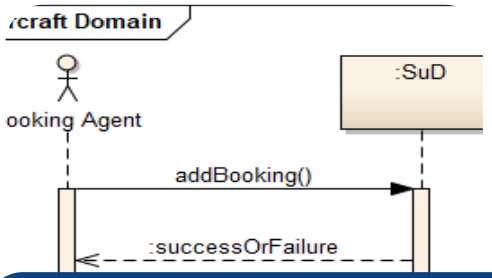
# Revisited: Object-Oriented Analysis, Design, and Development

**Problem domain**

- Concepts
- Relationships

**Use Cases**

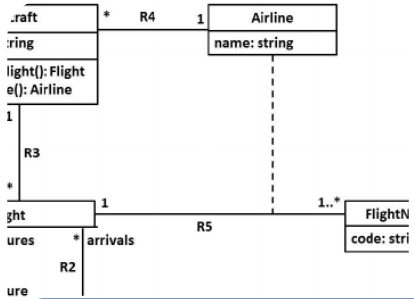
*Text stories*
Week 1




**OO Domain Models**

- Static: Domain Class Diagram
- Dynamic: System Sequence Diagram

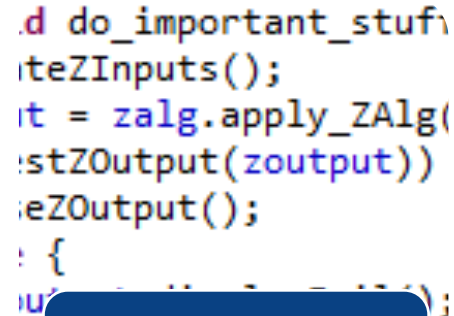
*Conceptual Classes*
Week 2




**OO Design Models**

- Static: Design Class Diagram
- Dynamic: Design Sequence Diagram

*Software Classes*
Week 3




**OO Implementation**

*Classes in Programming*
Week 3

# Revisited: Responsibility-Driven Design (RDD)

- RDD focuses on assigning *responsibility* to software objects
  - Responsibility: The obligations or behaviours of an object in terms of its role
- Two types of responsibility:

- **Knowing** responsibilities include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

- **Doing** responsibilities include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects

*How to assign responsibilities???*

# GRASP: General Responsibility Assignment Software Patterns/Principles

- **GRASP – Definition:** A set of patterns (or principles) of assigning responsibilities into an object.
- GRASP aids in applying design reasoning in a methodical, rational, and explainable way
  - Given a specific category of problem, GRASP guides the assignment of responsibilities to objects
- Useful to know and support Responsibility-Driven Design (RDD)
- **Pattern – Definition:**
  - A *named* and *well-known* problem/solution pair that can be applied in new contexts
  - A recurring successful application of expertise in a particular domain

# Some patterns in real life

- Q: What should a bathroom have?
- A: It should have a wash basin, a bathtub, a toilet, etc.

- Q: Where can I find a water fountain at the airport?
- A: It should be nearby toilets.


- Q: What's the area near the entrance of a building (like Stop1) for?
- A: It should be a reception

# Advantages of Patterns

*"To become a master, one must study the successful results of other masters!"*

- *Capture expertise* and make it accessible to non-experts in a standard form
- Facilitate successful applications of expertise is *easily re-used*
- *Improve understandability*
- Facilitate communication among practitioners by *providing a common language*
- A new solution can be generated based on a modified version of existing patterns

# GRASP: Learning Aid for RDD

- This subject will cover 9 GRASP patterns/principles
  - Creator
  - Information Expert
  - Low Coupling
  - High Cohesion
  - Controller
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations

# GRASP: Learning Aid for RDD

- This subject will cover 9 GRASP patterns/principles
  - Creator
  - Information Expert
  - Low Coupling
  - High Cohesion
  - Controller*
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations

*Controller will be covered in Workshop supplementary material

# GRASP: Creator

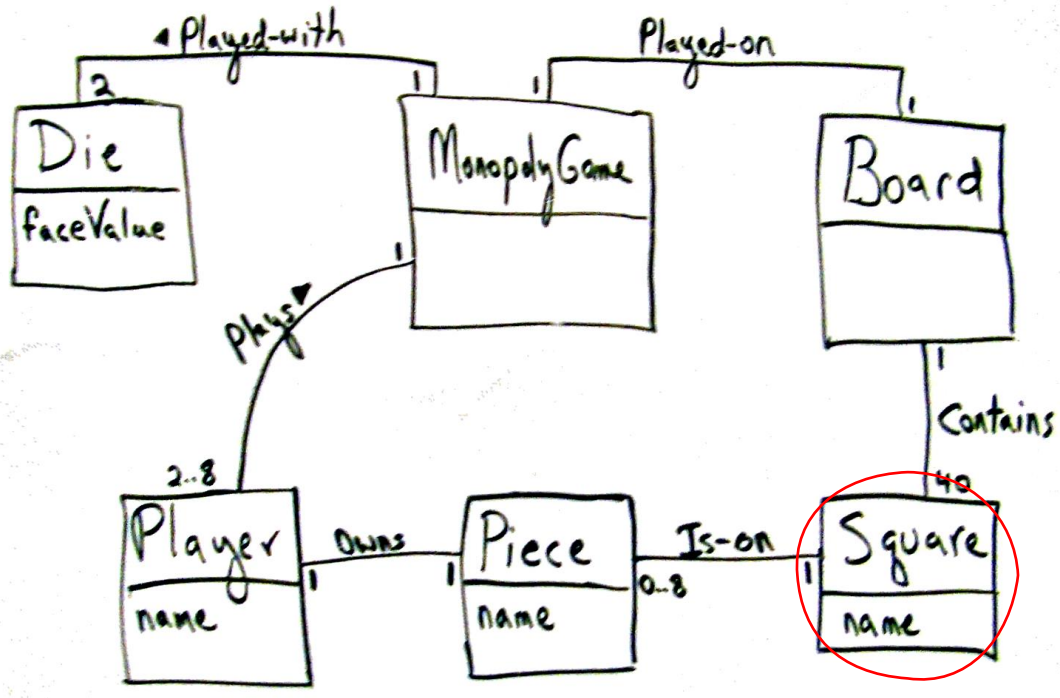
# Creator

- **Problem:** Who should be responsible for creating a new instance of class *A*? *(Doing Responsibility)*
  - Creation of objects is one of the most common OO activities
  - Useful for lower maintenance and higher opportunities for reuse (Low Coupling)
- **Solution:** Assign class *B* responsibility to create instances of class *A if* one of these is true (the more the better):
  - *B* "contains" or compositely aggregates *A*
  - *B* records *A*.
  - *B* closely uses *A*.
  - *B* has the initializing data for *A*.

*To achieve **low representational gap**, we use the domain model to inspire the design classes*

*If >1 classes applicable, choose the class which contains/aggregate *A*

# Example: Monopoly




Monopoly Game Simulation Domain Model

**Problem:** Who should be responsible for creating 'Square' object?

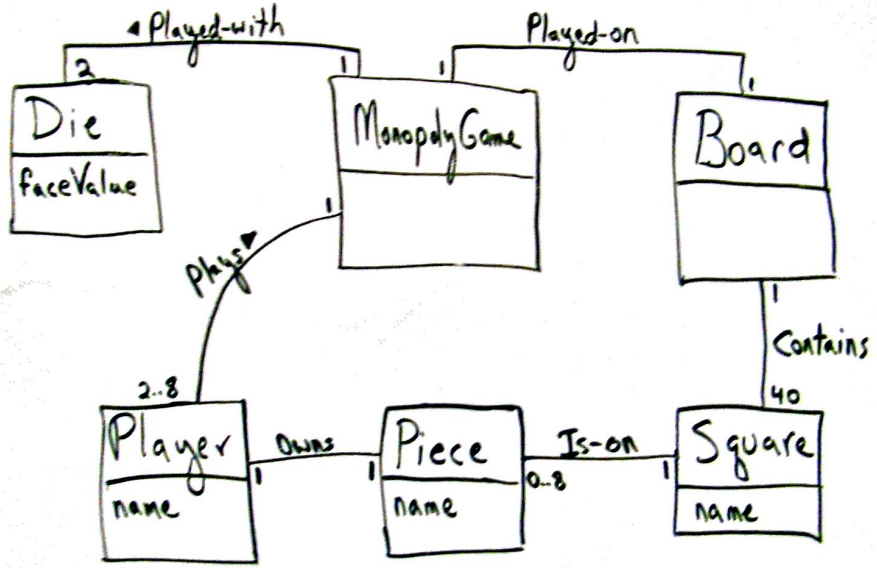
**Solution:**

- *Who* "contains" or compositely aggregates *Square objects* → ***Board***
- *Who* records *Square objects* → ***N/A***
- *Who* closely uses *Square objects* → ***Board & Piece***
- *Who* has the initializing data for *Square objects* → ***N/A***

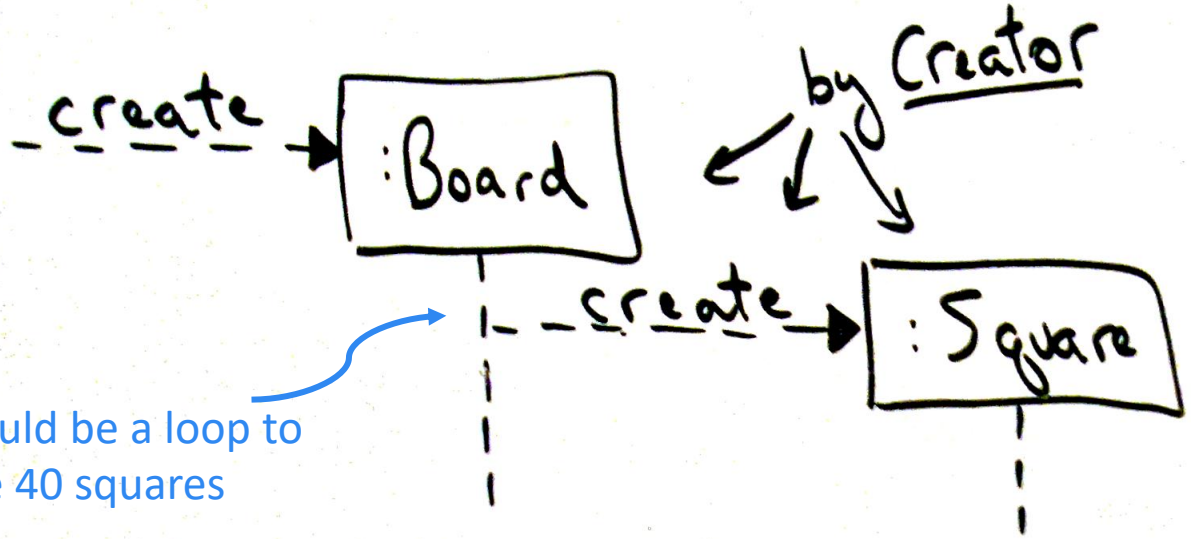
***Board* class should be responsible for creating *Square* object**

# Example: Monopoly

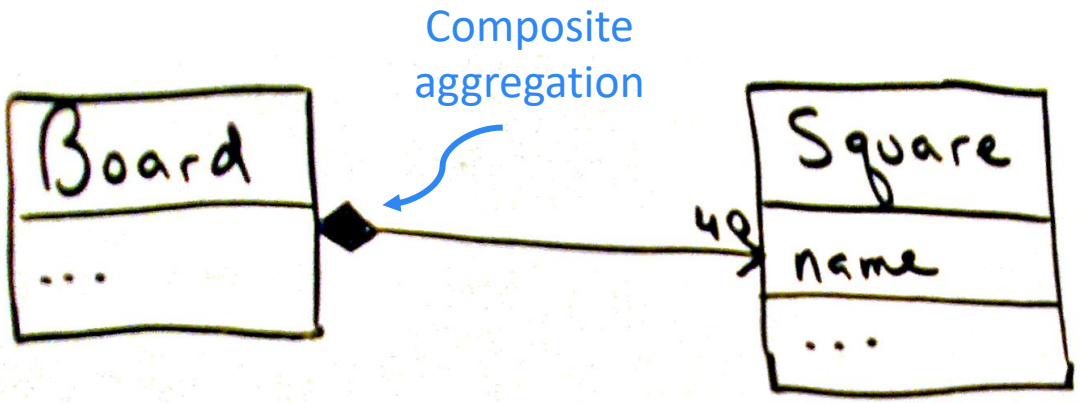
***Board* class should be responsible for creating *Square* object**




Monopoly Game Simulation Domain Model




Design Sequence Diagram

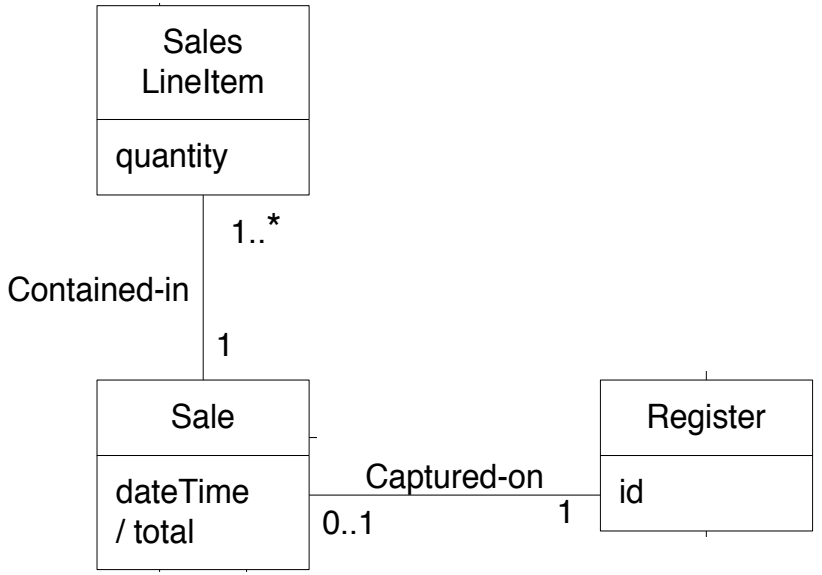



Design Class Diagram

# More Examples

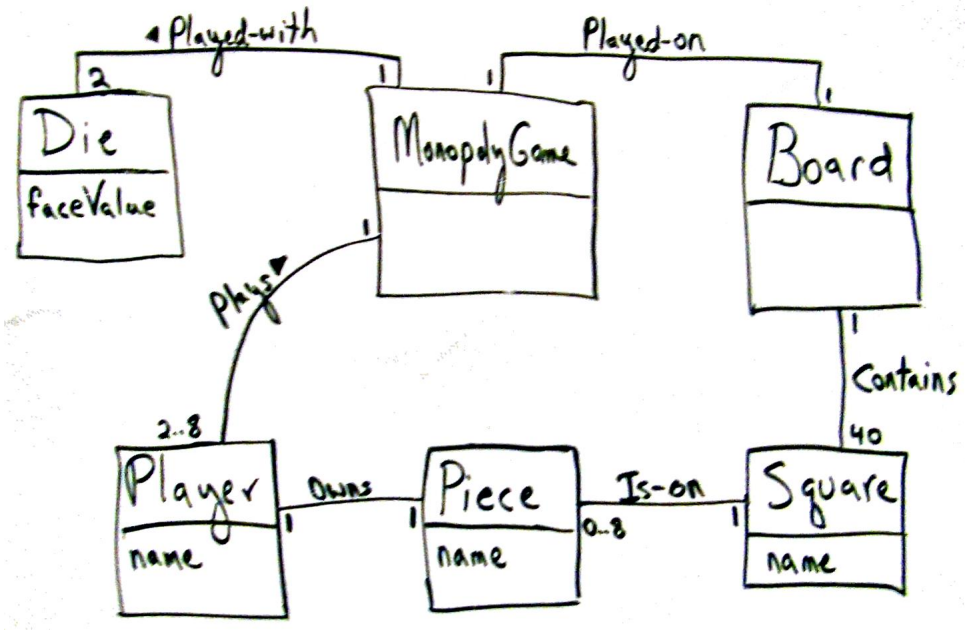
**Who is responsible for creating "SaleLineItem" object?**

→ Since a *Sale* class contains/aggregates many *SaleLineItem* objects, Sale should create *SaleLineItem* objects




**Who is responsible for creating "Player" object?**

→ Since a MonopolyGame class has the initializing data for "Player" object (i.e., the Player's name received from the user input), MonopolyGame should create 'Player' object



# Creator: Contradictions

- If creation of objects has significant complexity, e.g.,
  - Using recycled instances for performance
  - Conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth
- It is advisable to delegate creation to a helper class called a *Concrete Factory* or an *Abstract Factory* Rather than use the class suggested by Creator
- Concrete Factory will be taught in GoF Design Patterns lectures (Weeks 7-9)

# Information Expert (or Expert)

# Information Expert (or Expert)

- **Problem:** What is a general principle of assigning responsibilities to objects?
  - A design model may have 100s (or 1,000s) software classes with 100s (or 1,000s) of responsibilities to be fulfilled
  - Given object *X,* which responsibilities (e.g., doing something or knowing something) can be assigned to *X*?
- **Solution:** Assign the responsibility to object *X* if *X* has the information to fulfill that responsibility
  - Information Expert is useful for understandability, maintainability, and extendibility

*Where to look?:*

1. *If there is relevant class in the Design model, look there first*
2. *Otherwise, look in the Domain model to inspire creation of design classes*

# Example: Monopoly

Suppose there are objects that need to be able to reference a particular *Square*, given its name.

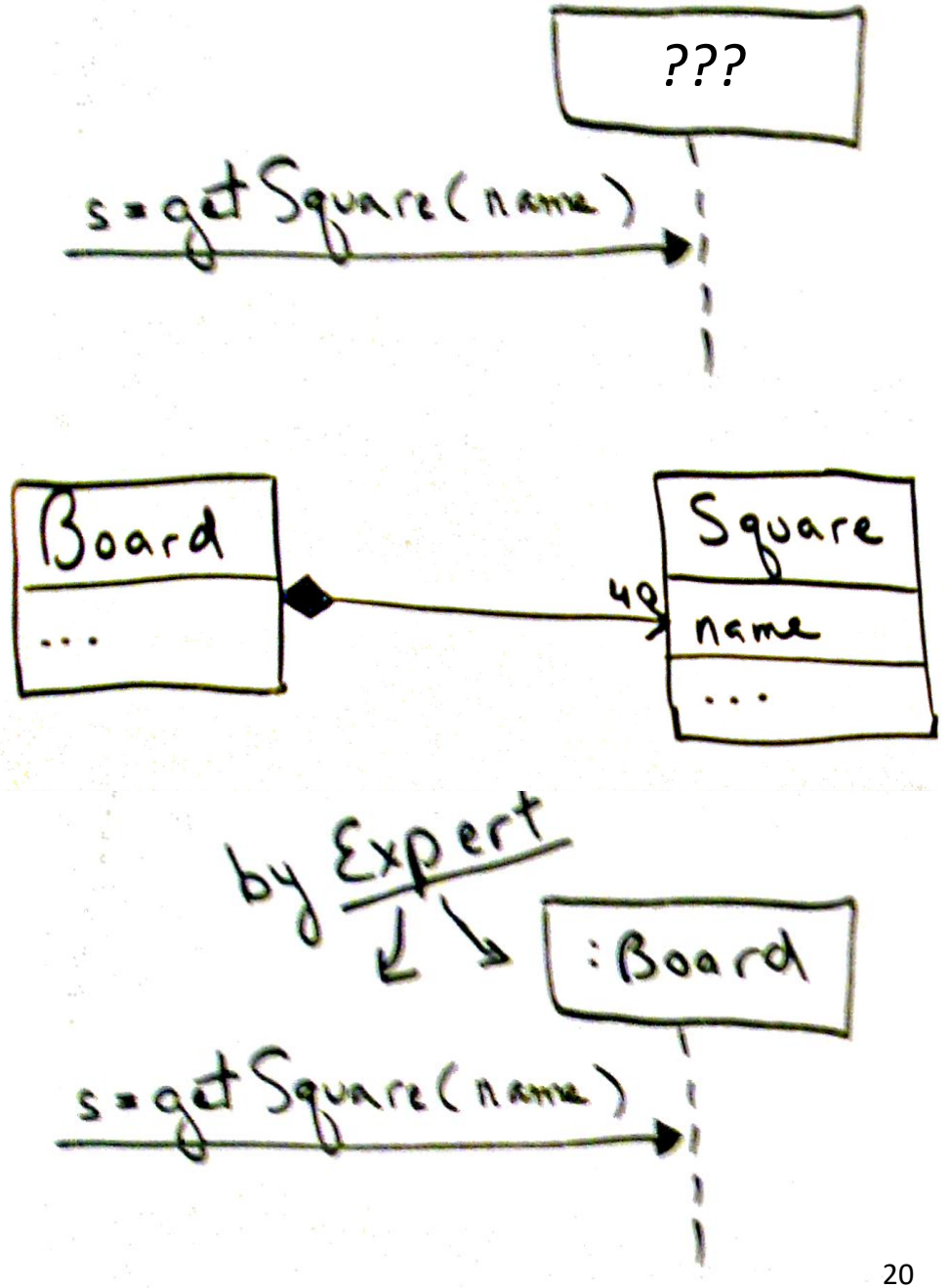
**Who should be responsible for knowing a *Square*, given a key?**

Approach:

→ Who has information about a Square? Who can access Square objects?

→ A software *Board* class aggregate all the *Square* objects (based on the previous example).

Solution: Board has information about 'Square' (i.e., an information expert). Then, Board should be responsible for knowing a Square, given a key.



# Another Example: POS

**Who should be responsible for calculating *Subtotal*?**

Approach:

→ What information do we need for 'Subtotal'?

→ Subtotal = quantity * price

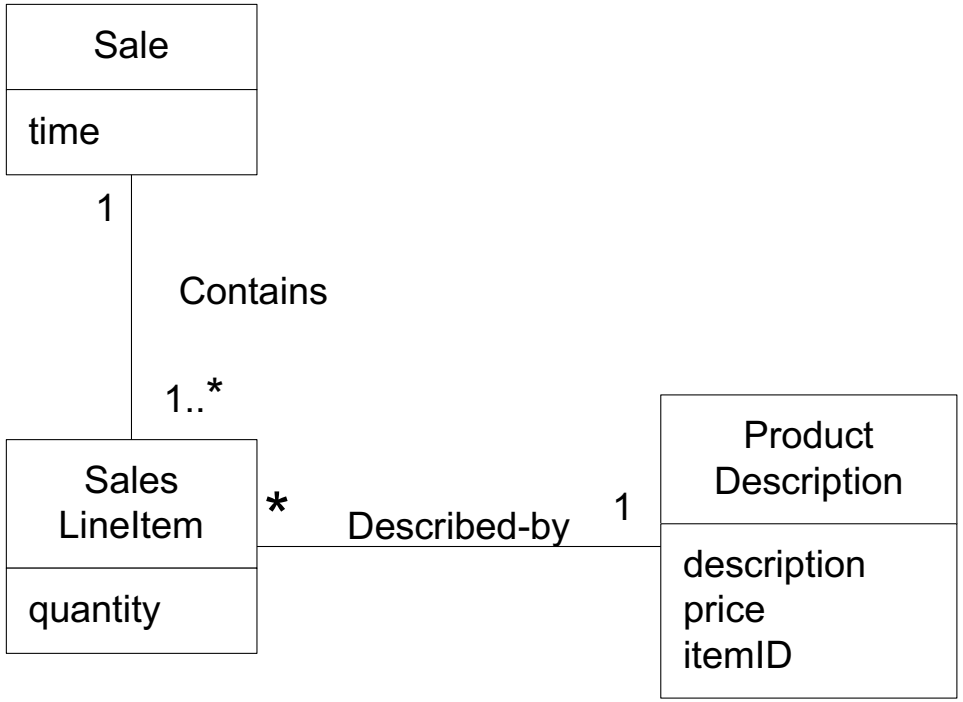
→ Who has (or can access) information about quantity and price?

→ SaleLineItem knows quantity, ProductDescription knows price

Also an Information Expert

→ SaleLineItem associates with ProductDescription, then SaleLineItem can also access price

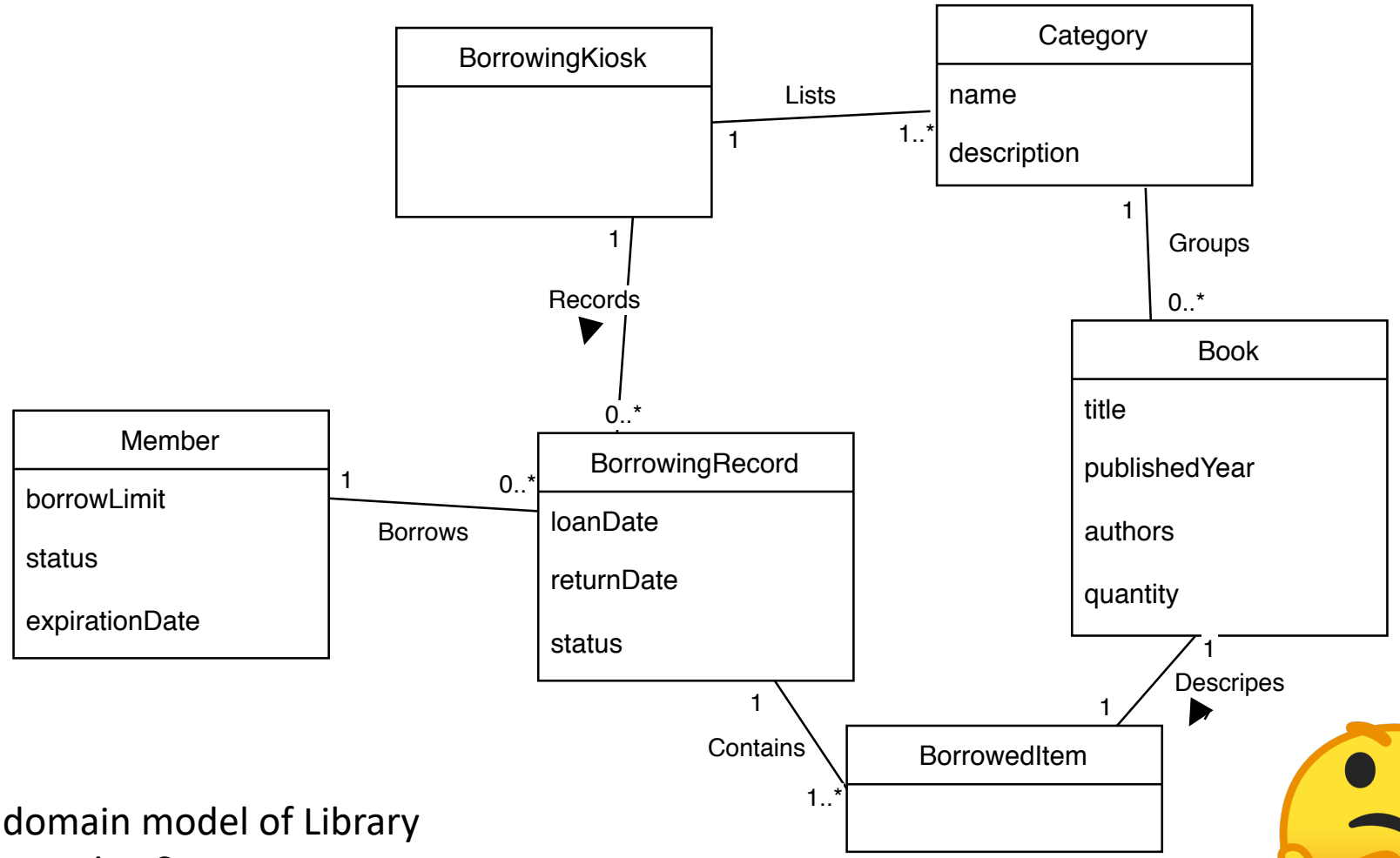
Solution: By Expert, SaleLineItem should be responsible for calculating Subtotal



# Information Expert: Contradictions

- In some situations, a solution suggested by Expert is undesirable, usually because of problems in high dependencies
- Example: Who should be responsible for storing Sale transaction into the database?
  - By Expert, Sale should be responsible
  - BUT, database handling (e.g., SQL) is not related to the logic of the application
  - Put the database logic and application logic in one class will cause high dependencies -> poor maintainability
  - Database logic should be done by other helper classes

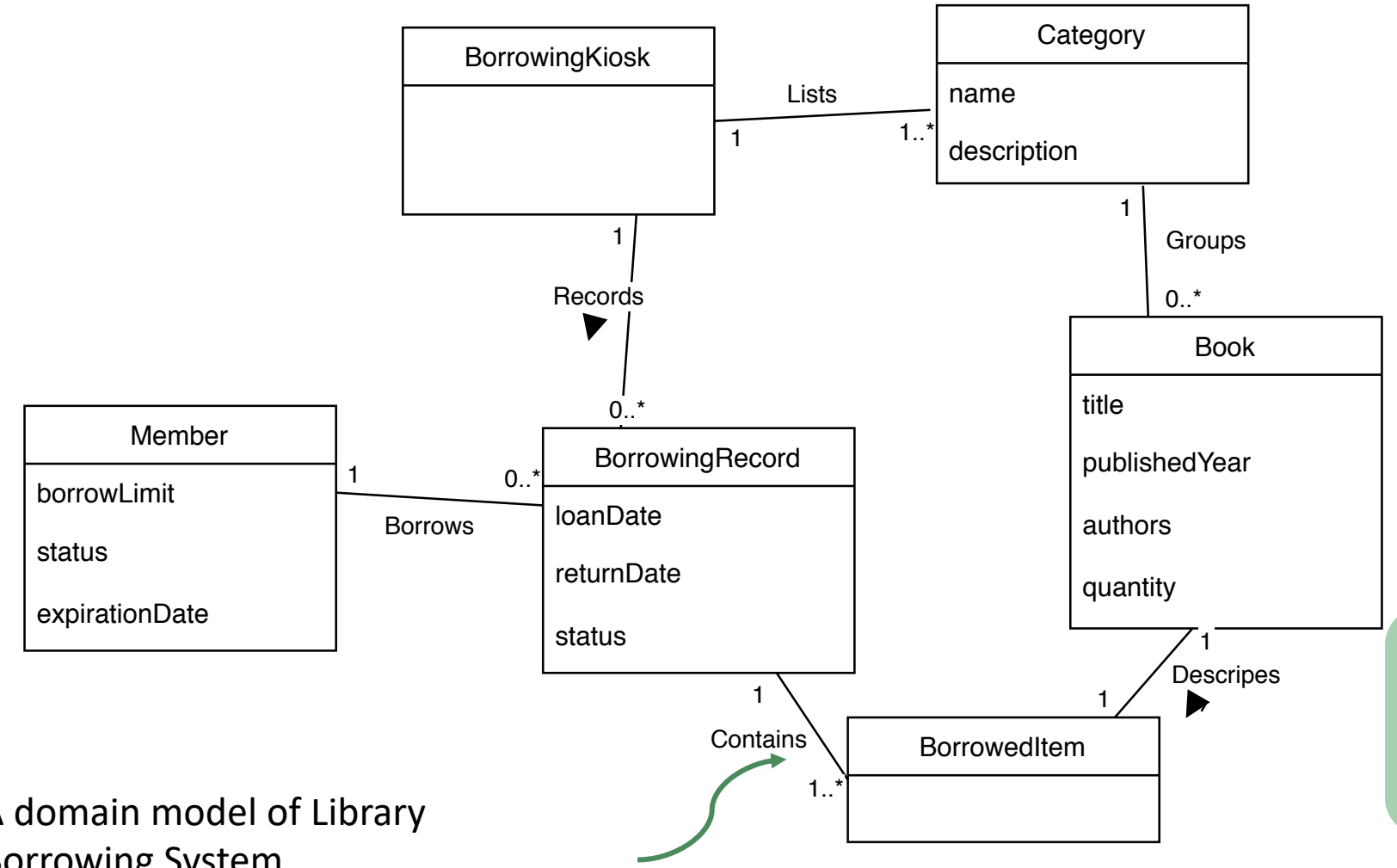
# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




A domain model of Library Borrowing System

Who should be responsible for creating “BorrowedItem” object?

# Exercise: Assigning responsibilities to software objects of the design model based on the domain model



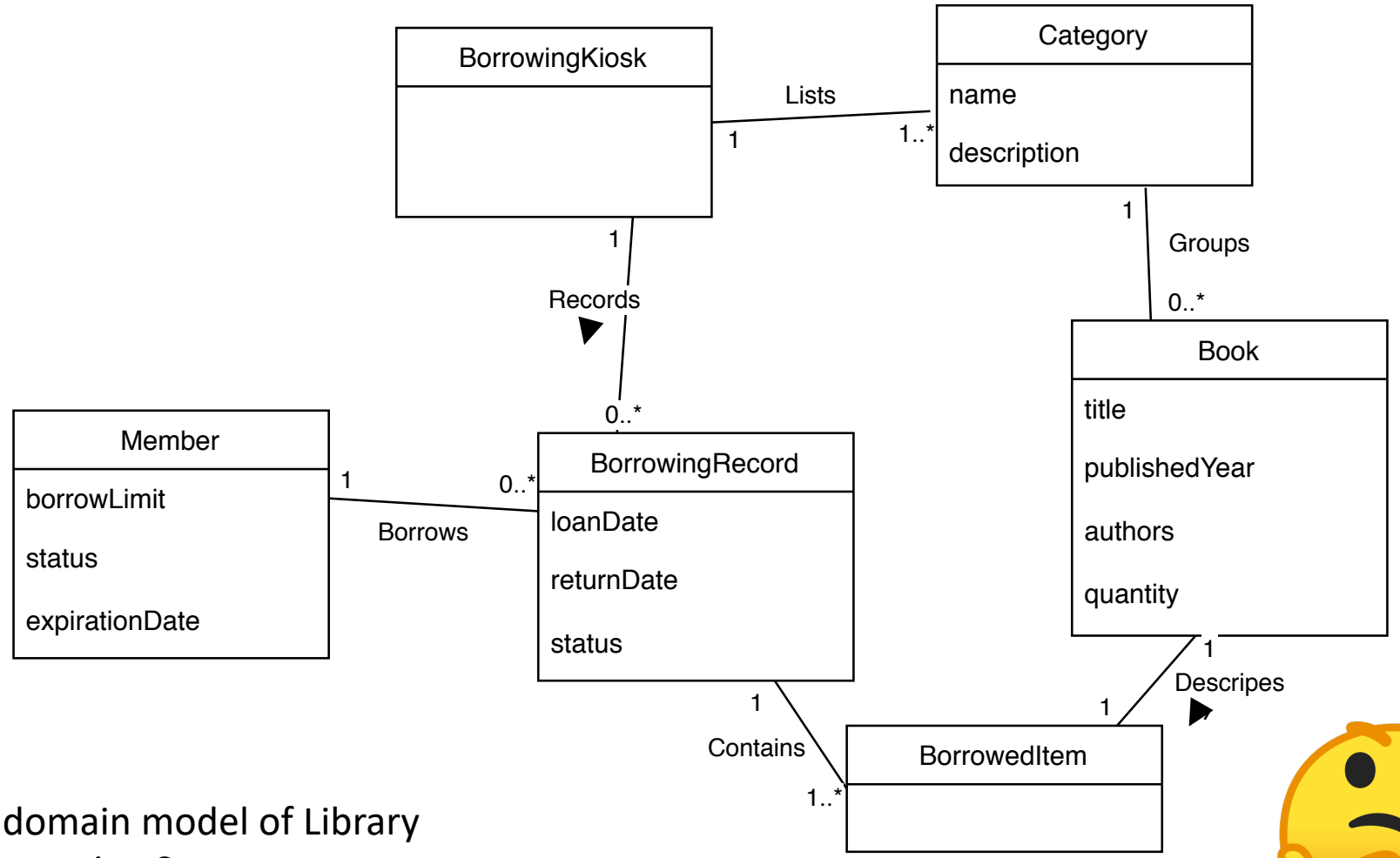

A domain model of Library Borrowing System

Who should be responsible for creating "BorrowedItem" object?

Suggested by Creator, BorrowingRecord should be responsible for creating BorrowedItem object.

Inspirerd by the domain model, BorrowingRecord contains/compositely aggregate BorrowedItem

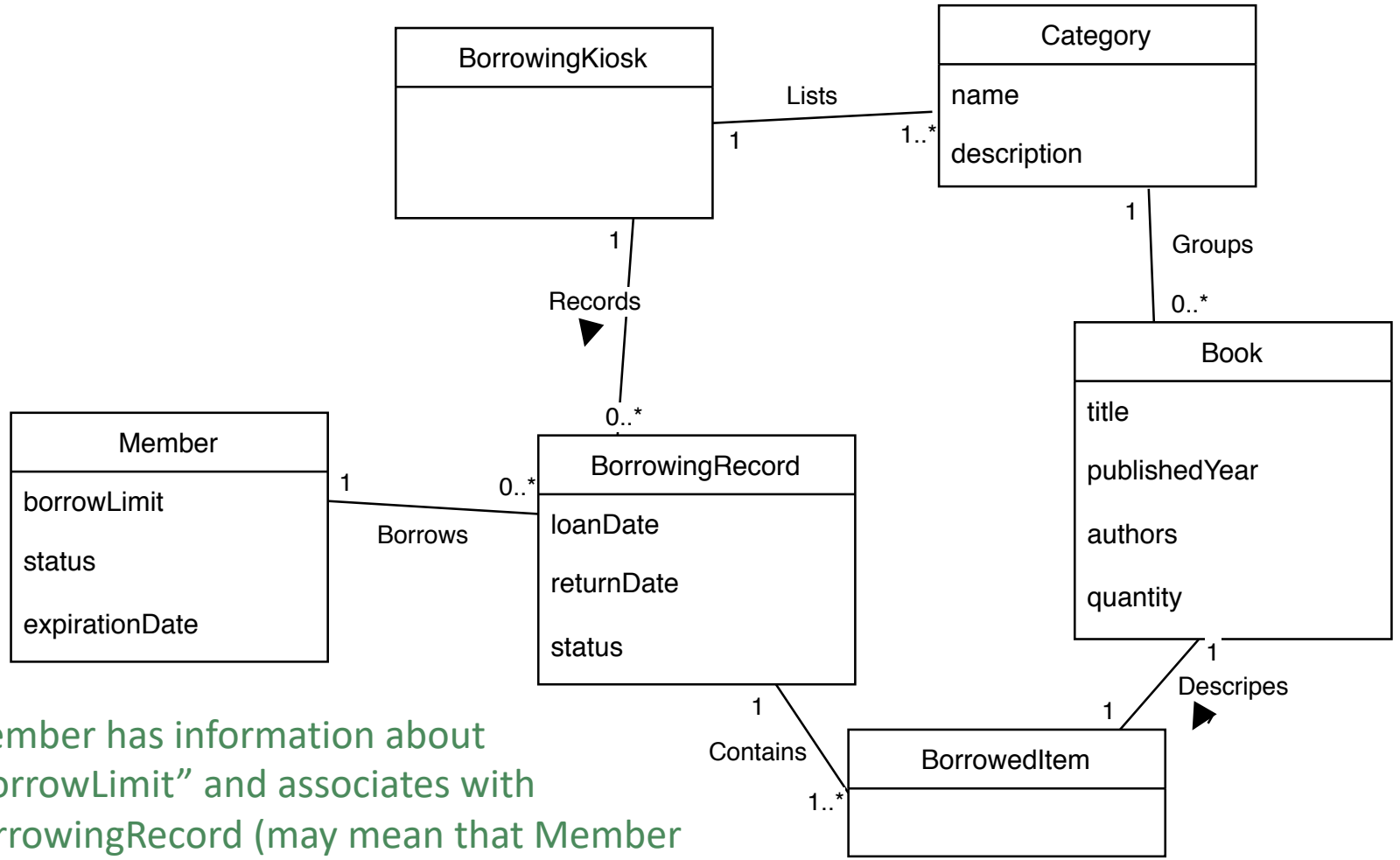
# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




A domain model of Library Borrowing System

Who should be responsible for checking if a member reaches the borrowing limit?

# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




Who should be responsible for checking if a member reaches the borrowing limit?

By Expert, Member should be responsible for checking if a member reaches the borrowing limit

Member has information about "borrowLimit" and associates with BorrowingRecord (may mean that Member can also own BorrowingRecord)



NOTE: Other designs might be applicable. Use other GRASP principles to evaluate the alternatives


# Low Coupling

# Low Coupling

- **Problem:** How to support low dependency, low change impact, and increased reuse?
  - Coupling = how strongly one element is connected to (has knowledge of, or relies on) other elements
  - Low Coupling = An element is not dependent on too many other elements
  - High Coupling = Hard to understand in isolation, hard to reuse, one change impacts many classes (high impact change)

- **Solution:** Assign the responsibility to object *X* that coupling remain low.
  - *Use this principle to evaluate alternatives*
  - Low Coupling minimize the dependency, hence making system maintainable, efficient, and code reusable

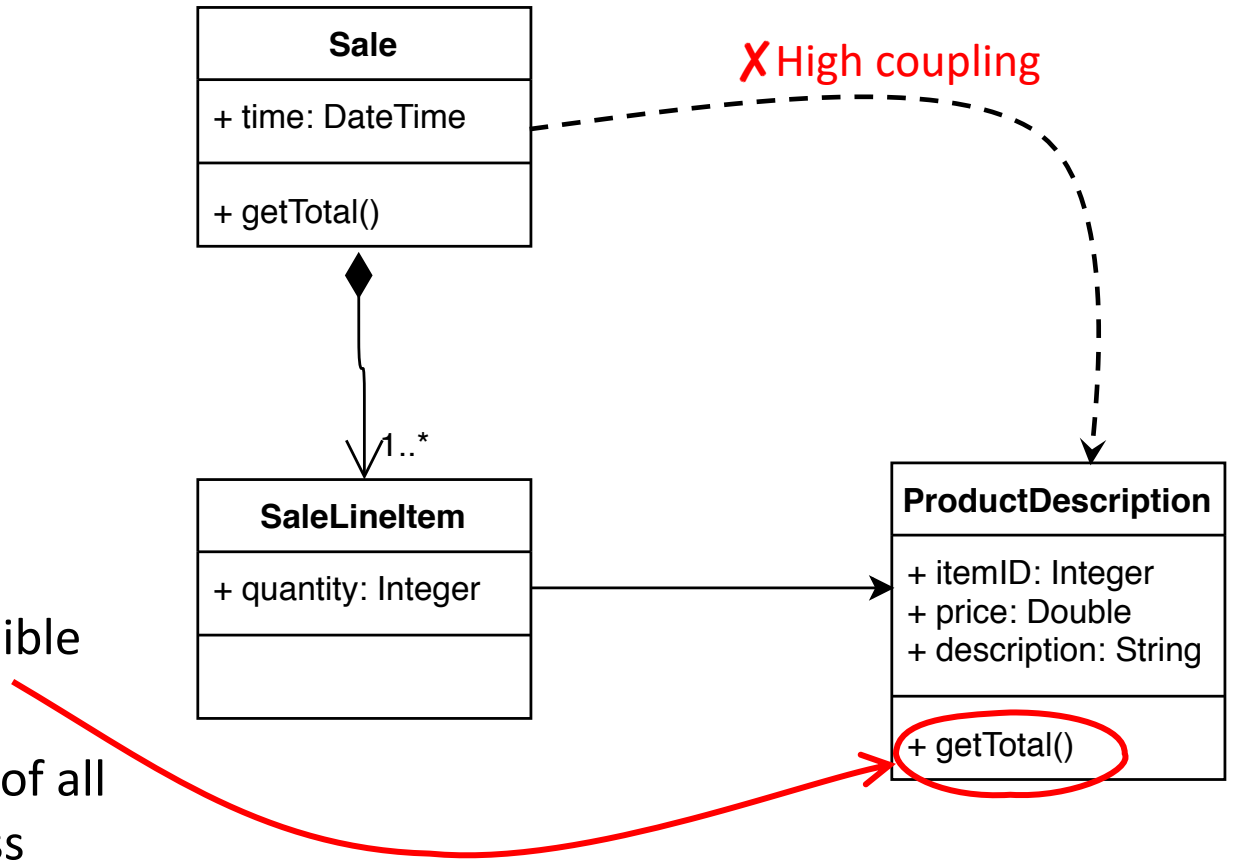
# Example: POS

**Who should be responsible for calculating *SubTotal*?**

- SaleLineItem knows quantity,
  ProductDescription knows price

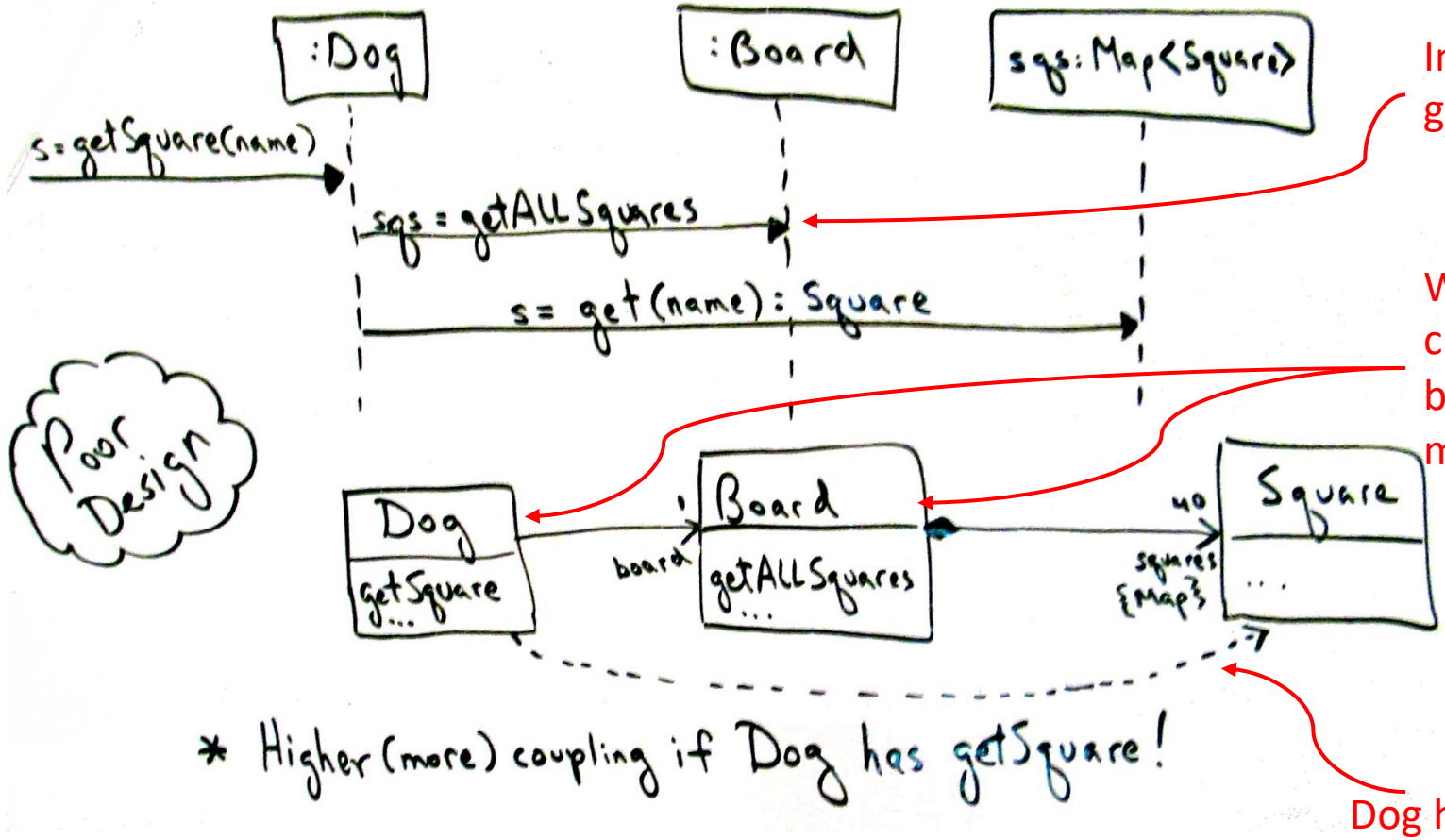
**An alternative solution:**

- By Expert, ProductDescription can also be responsible to calculate SubTotal
  - → Yes. But when Sale object will sum the SubTotal of all SaleLineItem objects, Sale object needs to access ProductDescription
  - → It is not suggested by Low Coupling principle



# Another Example: Monopoly

**Why can't we have other class (like Dog; an arbitrary class) to get a Square object instead of Board?**




Increase an additional step to get Square → Not efficient

When an interface in Square is changed, forcing a change in both Dog and Board → Hard to maintain

Dog has to connected with Square to obtain knowledge

# Low Coupling: Discussion

- Low Coupling is an ***evaluative principle*** that should be kept in mind during all design decisions
- Low Coupling supports the design of classes that are more independent, reducing the impact of change
- Low Coupling help you avoid increasing unnecessary dependencies/coupling that could lead to a negative result

**Contradictions**

- Highly coupled with stable elements (e.g., Standard Java Libraries) should not be a problem, since these stable elements are unlikely to be changed often


# High Cohesion

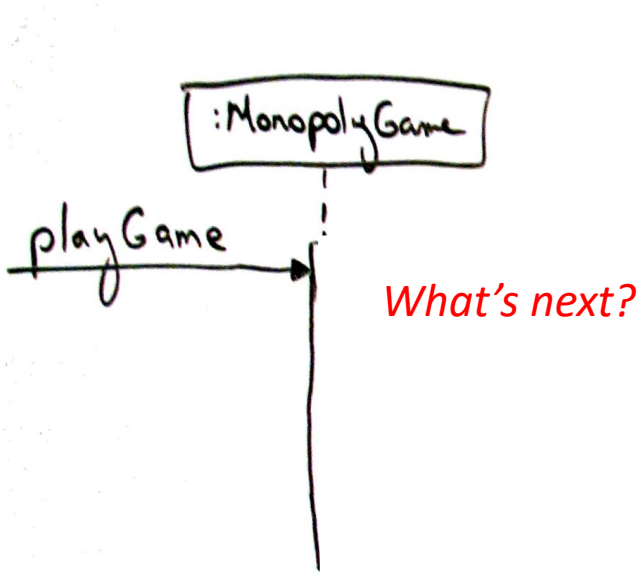
# High Cohesion

- **Problem:** How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - (Functional) Cohesion = How strongly related and focused the responsibilities of an element are
  - Low cohesion = a class has many unrelated things or does too much work, resulting code to be hard to comprehend, reuse, maintain

- **Solution:** Assign the responsibility to object *X* that make cohesion remains high.
  - *Use this principle to evaluate alternatives*
  - High cohesion will ease of comprehension of the design, simplify maintainability and enhancement

# Example: Monopoly

**When there is an input of playGame to MonopolyGame object, what MonopolyGame should do next?**



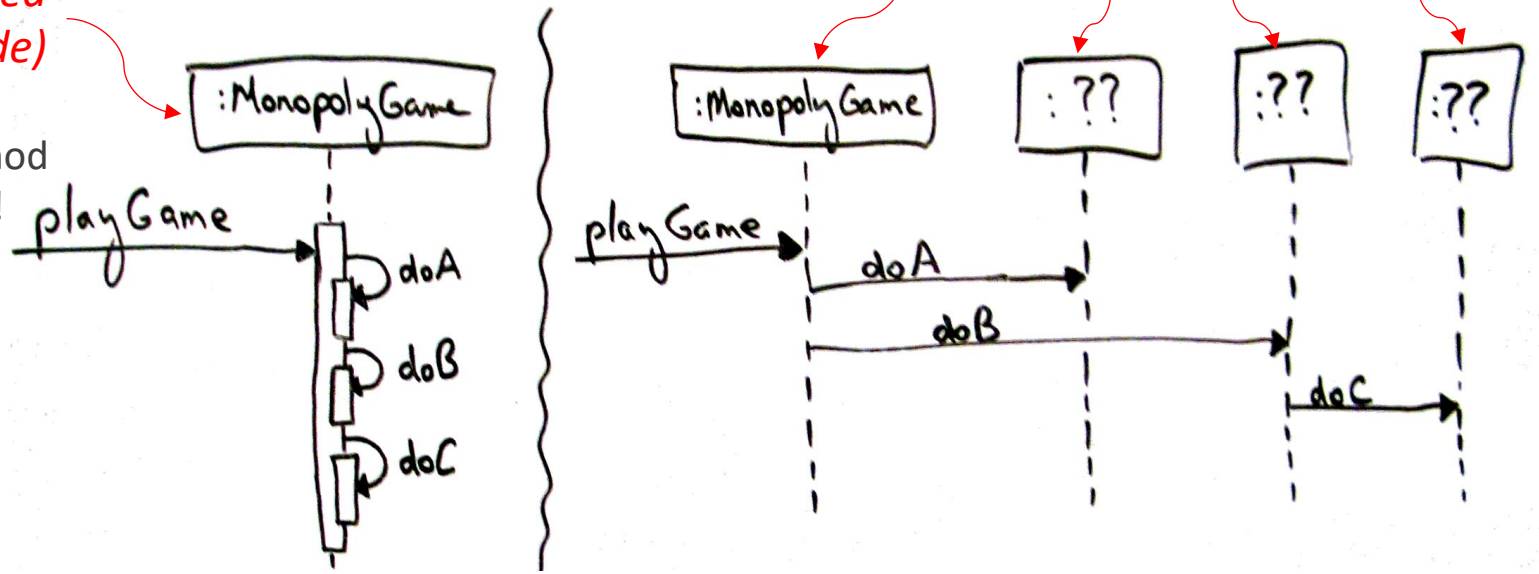
# Example: Monopoly

**When there is an input of playGame to MonopolyGame object, what MonopolyGame should do next?**
*-- Should MonopolyGame object be responsible for doing all the tasks, or delegate some tasks to other object?*

*The class can be bloated (too much code)*

Imagine a class has 100 methods, and each method has 200 lines = 20K lines!

*The code is more focused. Easy to understand, and maintainable*




*Do all the tasks?*

*Delegate some tasks to others?*

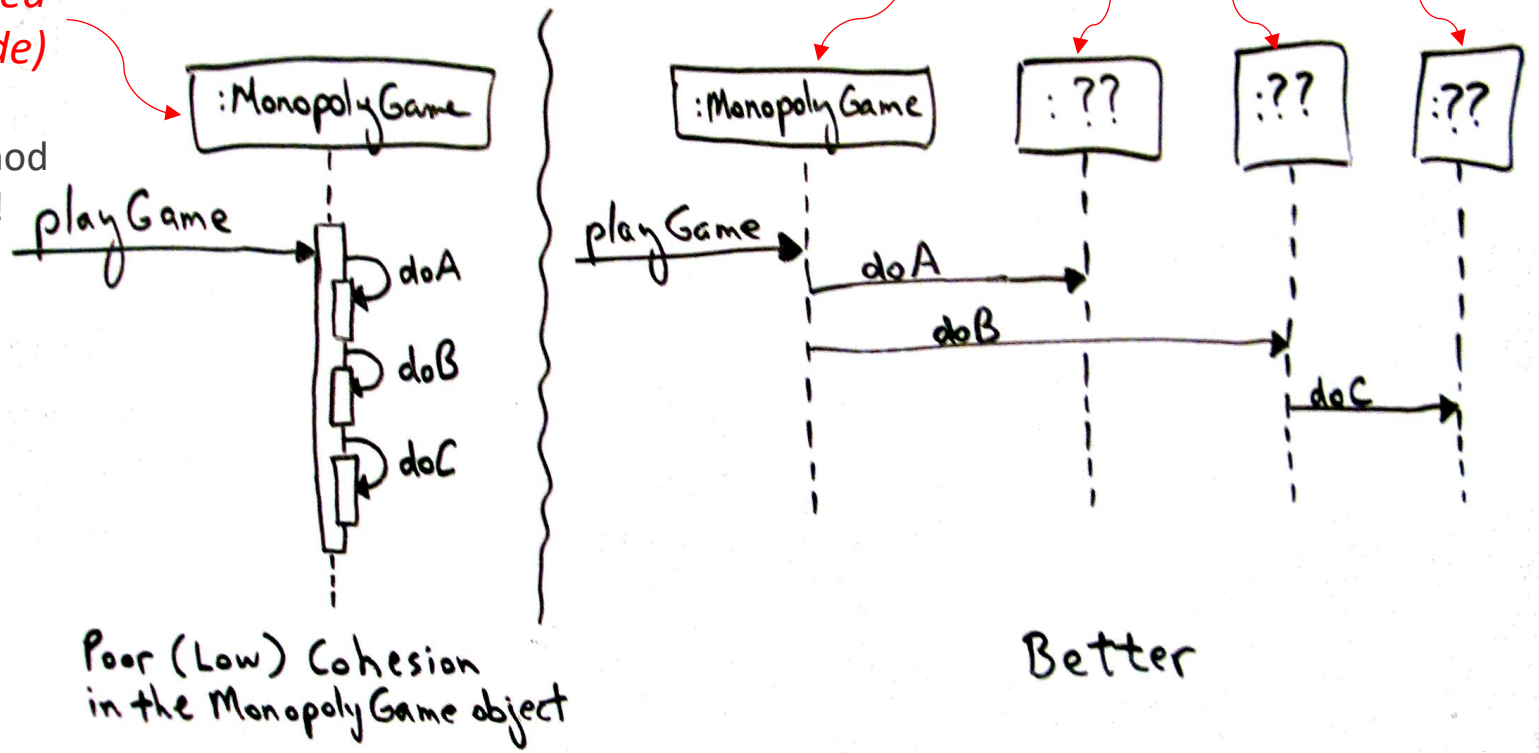
# Example: Monopoly

**When there is an input of playGame to MonopolyGame object, what MonopolyGame should do next?**
*-- Should MonopolyGame object be responsible for doing all the tasks, or delegate some tasks to other object?*

*The class can be bloated (too much code)*

Imagine a class has 100 methods, and each method has 200 lines = 20K lines!

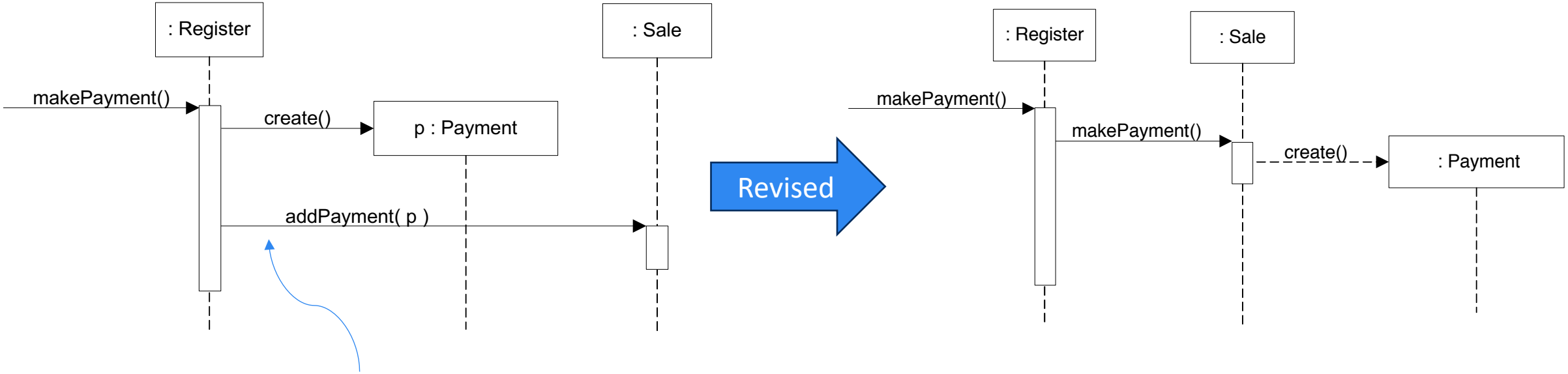
*The code is more focused. Easy to understand, and maintainable*



# Another Example: POS

**Why Sale is responsible fore creating Payment? Why not Register?**

In the real-world domain, Register *records* the Payment. By Expert, Register should be responsible for creating Payment.
→ *Yes.* It is acceptable.
→ BUT, in the real-world domain, Register involves most of the related system operations → *Register is potentially bloated*




Since Register has to addPayment to Sale, let's delegate the creation of Payment responsibility to Sale!

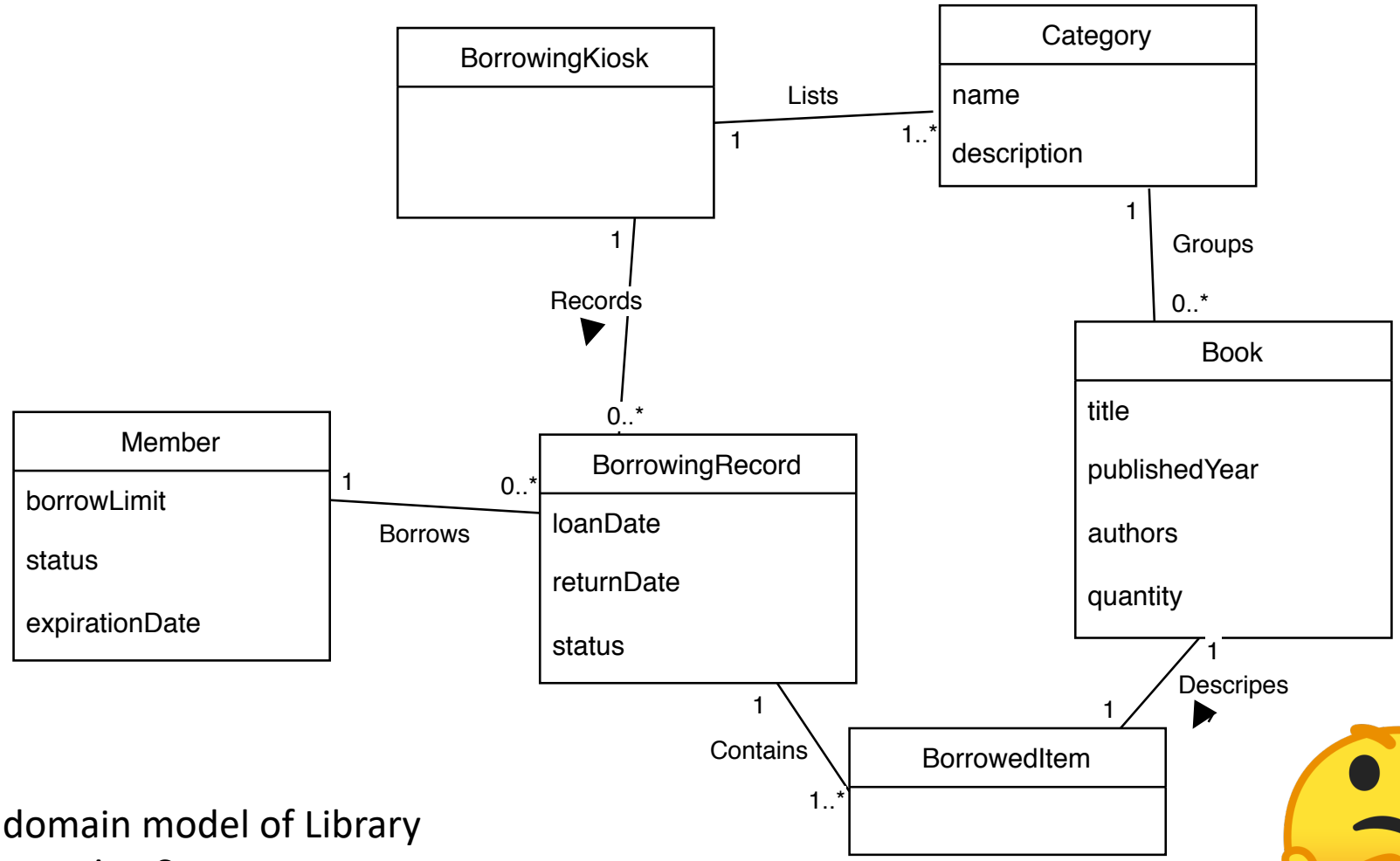
# High Cohesion: Discussion

- Similar to Low Coupling, High Cohesion is an ***evaluative principle*** that should be kept in mind during all design decisions
- Both Low Coupling and High Cohesion must be considered together when designing software objects

**Contradictions**

- Low cohesion (=many responsibilities in one class) sometimes is preferable if the design meets non-functional requirements
  - Ex: Larger and less cohesive classes are used to reduce processing overheads (e.g., transmission, storage) to meet performance requirement

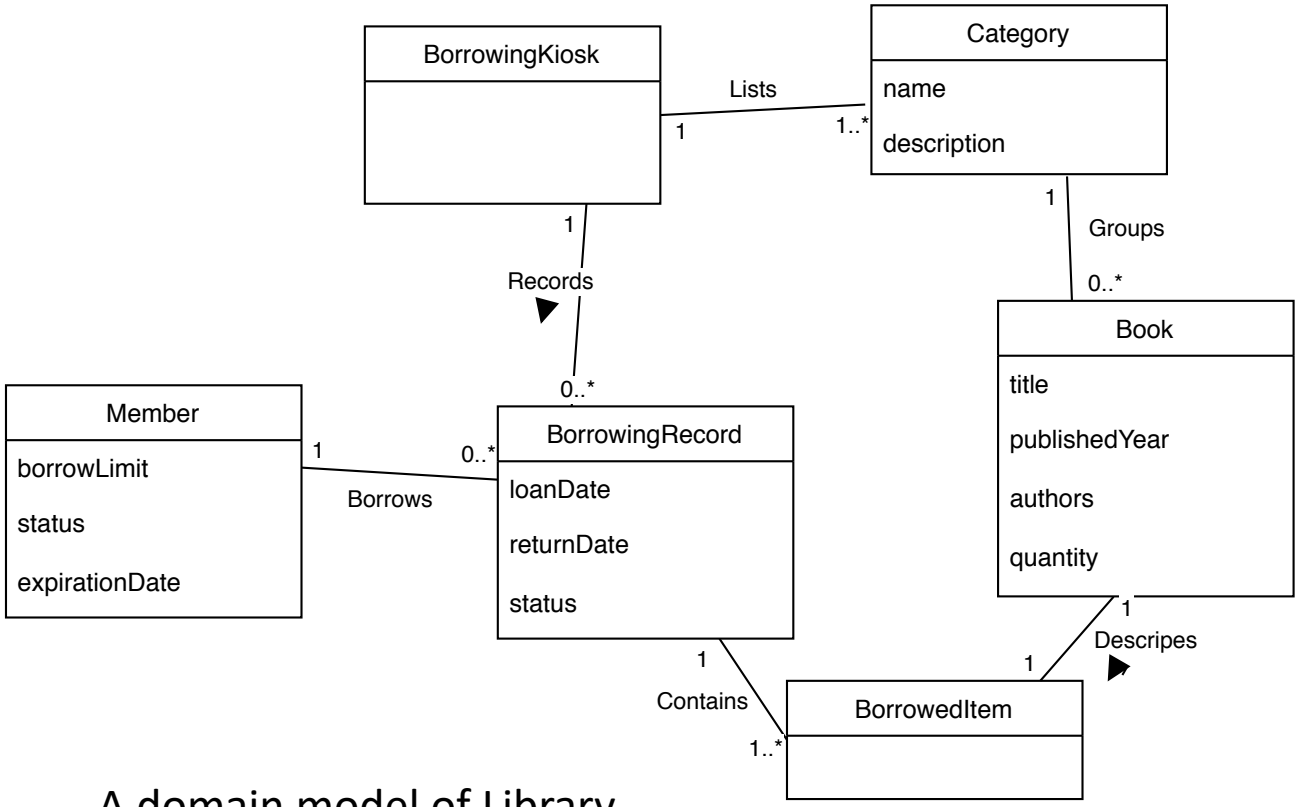
# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




A domain model of Library Borrowing System

BorrowingKiosk can also access BorrowingRecord. Why don't we assign the responsibility of "hasReachedLimit" to BorrowingKiosk?

# Exercise: Assigning responsibilities to software objects of the design model based on the domain model



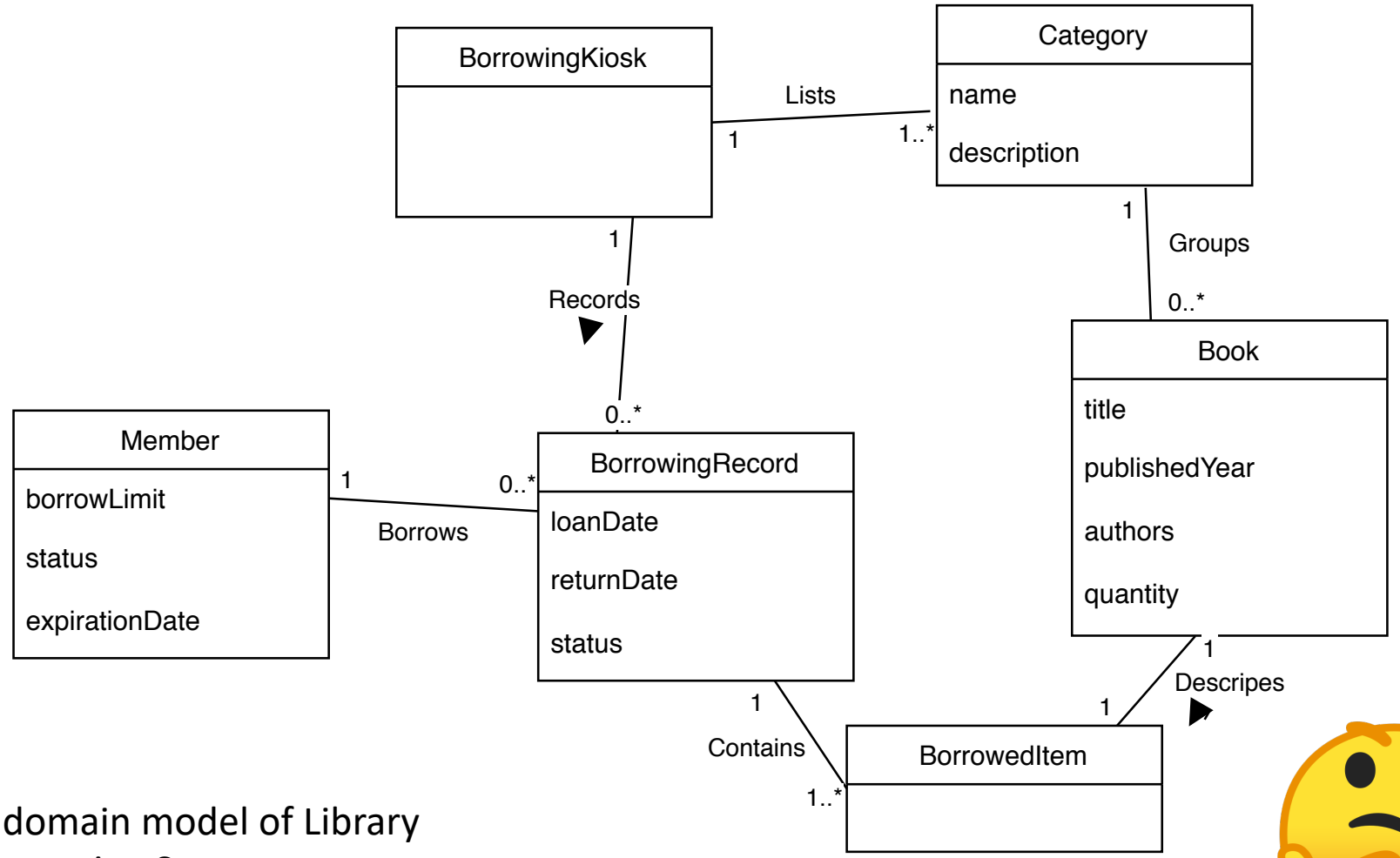

A domain model of Library Borrowing System

BorrowingKiosk can also access BorrowingRecord. Why don't we assign the responsibility of "hasReachedLimit" to BorrowingKiosk?

**Discussion**:

- Yes. It is acceptable.
- But it can increase coupling (i.e., an association between BorrowingKiosk and Member is needed in the design model)
- Also, BorrowingKiosk is likely to handle many operations -> increase a chance of low cohesion

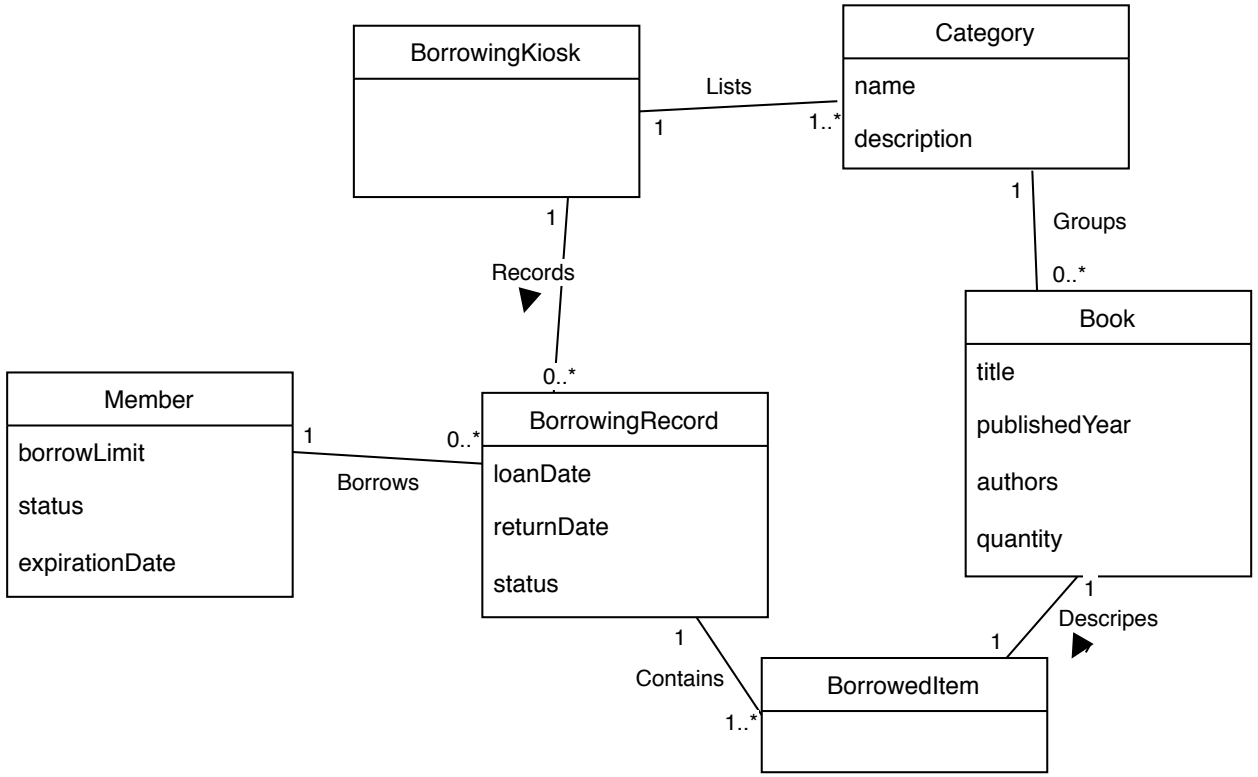
# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




A domain model of Library Borrowing System

Who should be responsible for creating "BorrowingRecord"?

# Exercise: Assigning responsibilities to software objects of the design model based on the domain model




A domain model of Library Borrowing System

Who should be responsible for creating "BorrowingRecord"?

**Discussion**:

- Depending on other design decision:
- One possible solution: If we assign Member to be responsible for checking the borrowing limit, then assigning a responsible for creating "BorrowingRecord" to Member will keep low coupling

NOTE: Other designs might be applicable. Use other GRASP principles to evaluate the alternatives

# Summary & Remarks

- GRASP is a set of patterns/principles that aid us to design software objects to be more maintainable
- GRASP guides us in making decision about ‘who should be responsible for doing/knowing something
- Given the same domain model/problem, there can be many possible design solutions that meets GRASP principles
- When there are many possible design solutions

  → Use Low Coupling and High Cohesion to evaluate the design solutions

# Lecture Identification

**Lecturer: Patanamon Thongtanunam**

**Semester 2, 2020**

**© University of Melbourne 2020**

**These slides include materials from:**

*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Third Edition, by Craig Larman, Pearson Education Inc., 2005.