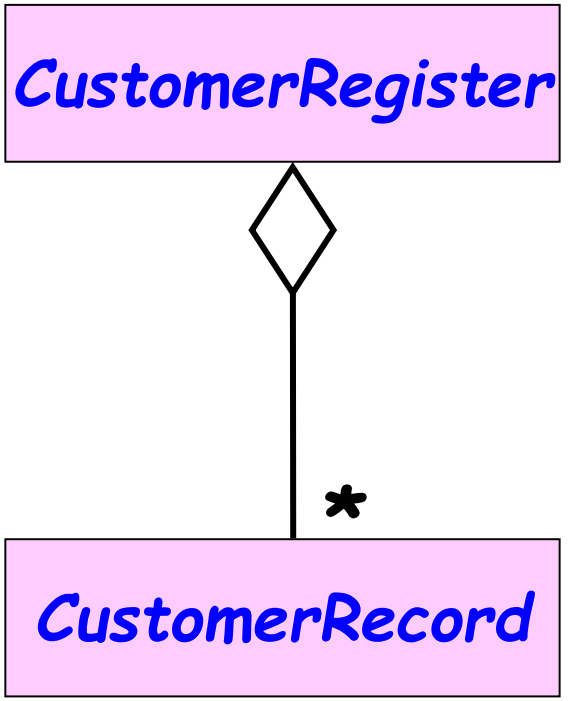
Review of Domain Modelling

Lect 18
3-10-2023

Identification of Entity Objects: Some Hints

- Usually:
  - Appear as data stores in DFD
  - Occur as aggregate objects
  - The aggregator corresponds to registers in physical world



7. Restaurant Automation Software

- A Resturant sells certain food items. The computer should maintain the prices of all the items:
  - Also support changing the prices by the manager.
- Whenever any item is sold:
  - The sales clerk would enter the item code and the quantity sold.
  - The computer should generate bills and register the payments.
- Whenever ingredients are issued for preparation of food items, the data is to be entered into the computer.
- Purchase orders are generated on a daily basis, for all ingredients whose stock falls below a threshold value.
  - The threshold value for each item is computed based on the average consumption for the past three days and assuming that two days stock must be maintained.
- Whenever the ordered ingredients arrive, the invoice data regarding the quantity and price is entered.
  - If sufficient cash balance is available, the computer should print cheques against invoice.
- Monthly sales receipt and expenses data should be generated whenever the manger would request to see them.

Design Patterns

Design Patterns

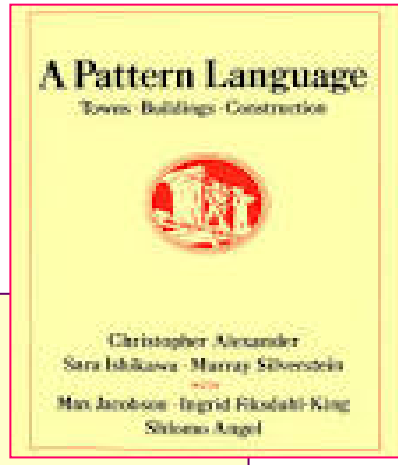
- **Patterns are used as "building blocks" in software design.**
  - Help designers to make important design decisions correctly.

- If you can master a few important patterns:
  - **You can easily spot them in your next application design problem and use pattern solutions.**

Origin of Patterns

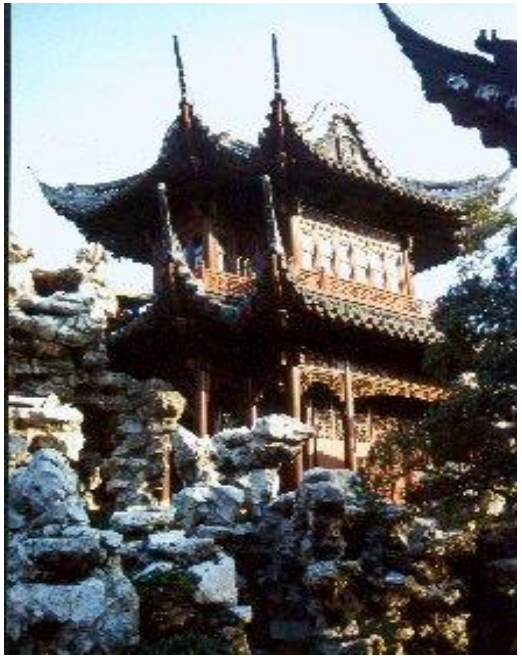
- Roots in the architecture field:
  - Made prominent by Christopher Alexander (1977)

Christopher Alexander, A Pattern Language, 1977



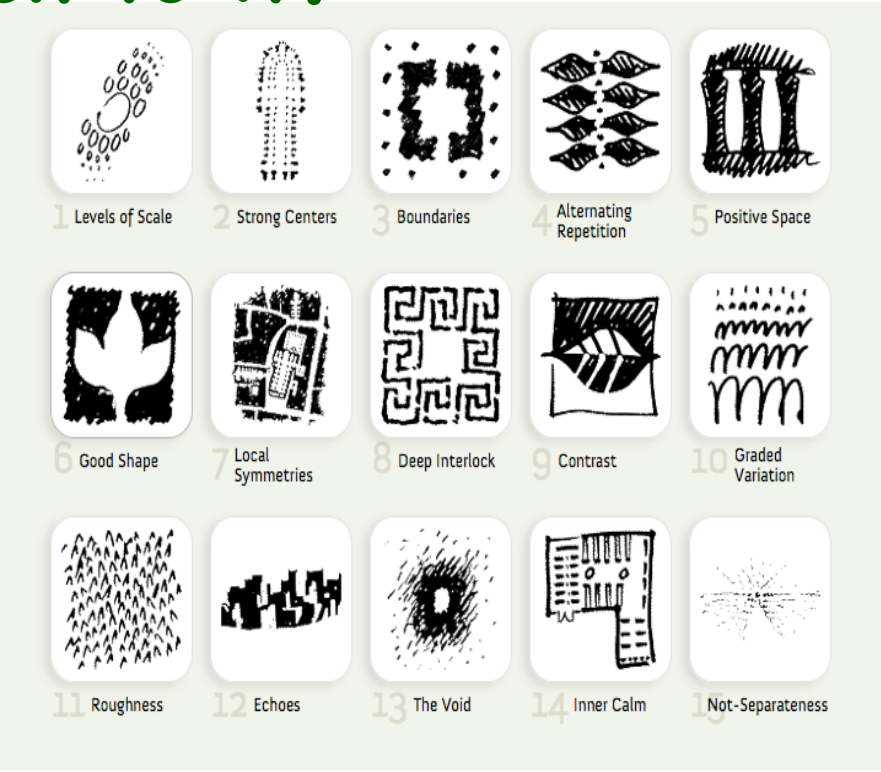
- Each pattern:
  - Describes a problem which occurs over and over again…
  - Describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.

… use this solution a million times over, without ever doing it the same way twice... Christopher Alexander



Christopher Alexander, A Pattern Language, 1977

- 253 patterns together formed the vocabulary of the language.
- **Each pattern describes a problem:**
  - **and then offers a solution to it.**
- They give ordinary people as well as professionals:
  - **A way to design a house, improve a town or neighborhood, etc…**



Patterns in Engineering

- How do other engineering fields find and use patterns?
  - **Mature engineering disciplines have handbooks describing successful solutions to known problems**
  - **Automobile designers don't design cars from scratch.**
  - **Instead, they reuse standard designs with successful track records.**
- Should software engineers make use of patterns?
  - **Developing software from scratch is expensive**
  - **Patterns support reuse of past knowledge mainly in the form of software architecture and design.**

Patterns in Software Design

- **Architectural Patterns:** MVC, Layers etc.
- **GoF Design Patterns:** Singleton, Observer etc
- **GUI Design Patterns:** Window per task, Disabled irrelevant things, Explorable interface etc
- **Database Patterns:** decoupling patterns, resource patterns, cache patterns etc.
- **Concurrency Patterns:** Double buffering, Lock object, Producer-consumer,Asynchronous processing etc.
- **Enterprise (J2EE) Patterns:** Data Access Object, Transfer Objects etc.
- **GRASP (General Responsibility Assignment Patterns):** Low coupling/high cohesion, Controller, Law of Demeter (don't talk to strangers), Expert, Creator etc.
- **Anti-patterns (**Bad solutions deceptively appear good): God class,Singletonitis, Basebean, Golden hammer, etc.

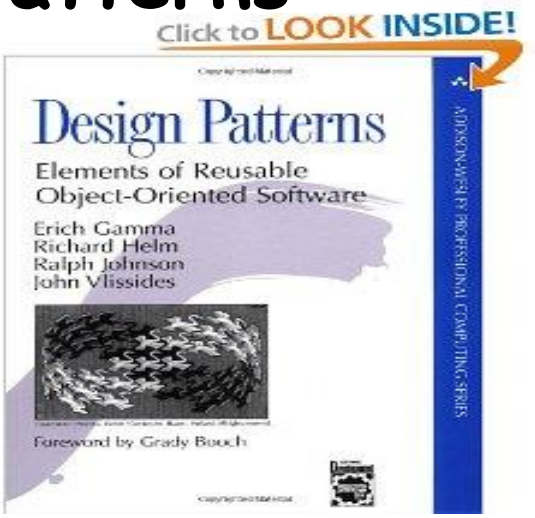
History of Design Patterns

- The concept of a "pattern":
  - First expressed in Christopher Alexander's work *A Pattern Language* in 1977
  - *A solution to a common design problem in a certain context.*

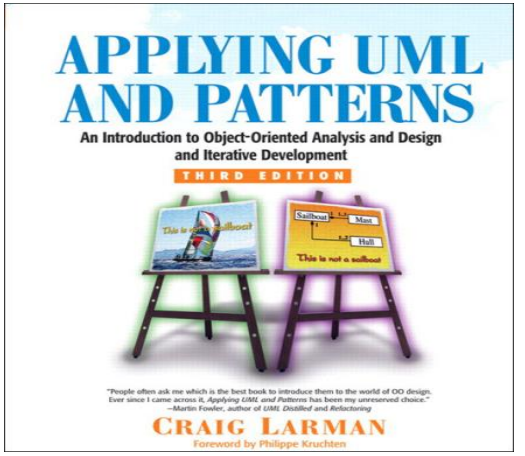



- In 1990:
  - Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compiled a catalog of design patterns
- Now assimilated into programming languages:
  - **Example:** Decorator, Composite, Iterator pattern **Defines an interface that declares methods for sequentially accessing the objects in a collection.**

"Gang of four" (GoF) and GRASP Patterns



- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - Design Patterns book catalogs 23 different patterns



- Larman
  - GRASP (General Responsibility Assignment Software Patterns) pattern
- Several other types of patterns are also popular.

Elements of Design Patterns

- Design patterns have 4 essential elements:
  - **Pattern name:** Forms designers' vocabulary
  - **Problem:** intent, context, when to apply
  - **Solution:** UML model, skeletal code
  - **Consequences:** results and tradeoffs

Goals of Design Patterns

- **Codify good design**
  - Distill & generalize experience
  - Aid to novices & experts alike
- **Give design structures explicit names**
  - Common vocabulary
- **Save design iterations**
  - Improve documentation
  - Improve understandability
  - Facilitate restructuring/refactoring

Types of Patterns

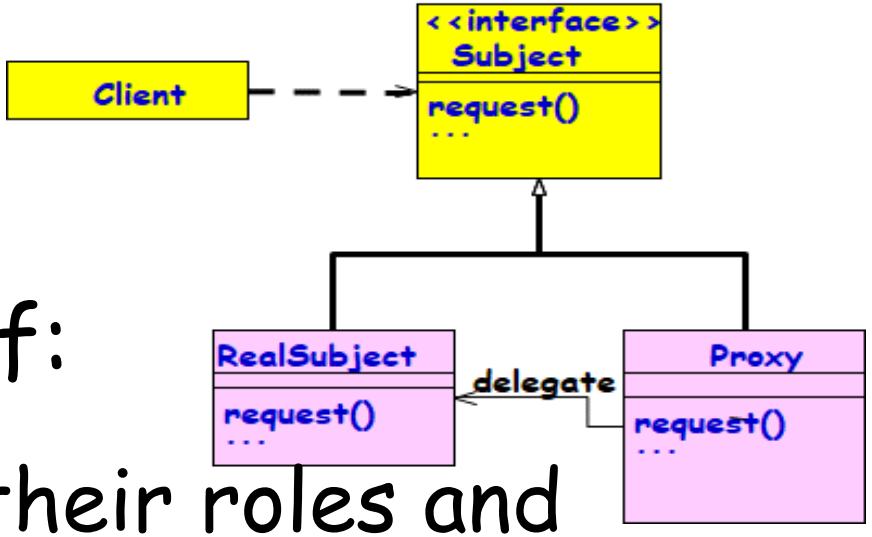
- Architectural patterns
- Design patterns
- Code patterns (Idioms)

Architectural Patterns

- Architectural designs concern the overall structure of software systems.
  - Architectural designs cannot directly be programmed.
  - Form a basis for more detailed design.
- **Architectural patterns:**
  - **Provide solutions to issues relevant to architectural design of large problems.**

Design Patterns

- A design pattern:
  - Suggests a scheme for structuring the classes in a design solution.
  - Also, defines the interactions required among those classes.



- Design pattern solutions are described in terms of:
  - Classes, their instances, their roles and collaborations.

Idioms

- Idioms are low-level patterns:
  - Programming language-specific.
  - **Describe how to implement a solution to a particular problem in a given programming language.**

Idioms

- What is an Idiom in English language?
  - *A group of words that has meaning different from a simple juxtaposition of the meanings of the individual words.*
  - Example: "Raining cats and dogs"
- *A C* idiom:

```
for(i=0;i<1000;i++){
                    }
```

Patterns versus Idioms

- A pattern:
  - Describes a recurring problem
  - Describes a core solution
  - Used for generating many distinct designs
- An Idiom though describes solution to a recurring problem, is rather restricted:
  - **Provides only a specific solution, with fewer variations.**
  - **Applies only in a narrow context**
    - **e.g., C++ or Java language**

Antipattern

- If each pattern represents a best practice:
  - **An antipattern represents lessons learned from a bad design.**
- Antipatterns help to recognise deceptive solutions:
  - **They appear attractive at first, but turn out to be a liability later…**
- Not enough to use patterns in a solution --- must consciously avoid antipatterns.

Patterns versus Algorithms

- **Are patterns and algorithms identical concepts?**
  - After all, both target to provide reusable solutions to problems!

- Algorithms primarily focus on solving problems with reduced space and/or time requirements:
  - **Patterns focus on understandability and maintainability of design and easier development.**

Pros of Design Patterns

- Help capture and disseminate expert knowledge.
  - **Promotes reuse and helps avoid mistakes.**
- Provide a common vocabulary:
  - Help improve communication among the developers.
  - **"The hardest part of programming is coming up with good variable and function names."**

Pros of Design Patterns cont…

- Reduces the number of design iterations:
  - **Helps improve the design quality and designer productivity**.

Patterns put you on the shoulders of Giants

- Isaac Newton famously remarked in a letter to Robert Hooke, in 1676:

  "If I have seen a little further, it is by standing on the shoulders of Giants."

- When Hamming received the Turing award in 1968. He remarked:

  "While Newton may have stood on the shoulders of Giants; in software development, sadly we stand on each other's feet."

Pros of Design Patterns

- Patterns exempify solutions to software problems making use of:
  - **Abstraction,**
  - **Encapsulation**
  - **SRP, OCP, LSP, ISP, DIP**
  - **Separation of concerns**
  - **Coupling and cohesion**
  - **Divide and conquer**

Cons of Design Patterns

- Design patterns do not directly lead to code reuse.
- How to select the right design pattern at the right point during a design exercise?
  - **At present no systematic methodology exists.**

Why Learn Design Patterns?

- **Your own designs will improve:**
  - Learn to borrow from well-tested ideas
  - Learn from pattern descriptions that usually contain some analysis of tradeoffs
- **You will be able to describe complex design ideas to others:**
  - Assuming that they also know the same patterns
- **You can use patterns to "refactor" existing code:**
  - Improve the structure of existing code without adding new functionality
- **You can understand why some aspects of Java language are that way.**

Thought for the day…

- **Why are there no design patterns for procedural designs? ….**

Why No Patterns for Procedural Development?

- For patterns to be defined, procedural languages need to support:
  - **Inheritance**
  - **Abstract classes and Interfaces**
  - **Polymorphism**
  - **Encapsulation**

Design Patterns

- These are standard solutions to commonly recurring problems.
  - Provide good solutions based on common sense
- Pattern has four important parts:
  - The problem
  - The context
  - The solution
  - The context in which it works or does not work

Pattern Problem

- The problem statement describes the problem and its context.
  - Also explainss when to apply the pattern.
  - It might describe situations where it works and does not work.
  - It can include a list of conditions that must be met before it makes sense to apply the pattern.

Pattern Solution

- Abstract description of design problem:
  - how the pattern solves it
- Describes the elements that make up:
  - Class design,
  - Class relationships,
  - Class responsibilities and
  - Class collaborations
- **Only skeletal implementation may be available.**

Design Patterns are NOT…

- **NOT** designs that can be plugged in and reused as it is:
  - Unlike code for linked lists, hash tables
- **NOT** complex domain-specific designs:
  - For an entire application or subsystem.
- Patterns are actually:
  - **"Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."**

GRASP Patterns

GRASP Patterns

- GRASP: **Generalized Responsibility Assignment Software Patterns:**
  - Larman, "Applying UML and Patterns"

- **GRASP patterns can more accurately be described as best practices:**
  - If used judiciously, will lead to maintainable, reusable, understandable, and easy to develop software

GRASP Patterns

- GRASP patterns essentially describe how to assign responsibilities to classes:
  - **Warning**: **Some Grasps tend to be vague and need to be seen as guidelines rather than solutions.**

- What is a responsibility?
  - ***A contract or obligation of a class***
  - Responsibilities can include behaviour, data storage, object creation, etc.
  - Usually fall into two categories:
    - **Doing**
    - **Knowing**

Responsibility-Driven Design (RDD)

- Advocates carrying out OOD by assigning:
  - **Responsibilities**
  - Roles
  - Collaborations
- Common responsibility categories:
  - **Doing:**
    - Creating an object or doing a calculation
    - Initiating action in other objects
  - **Knowing:**
    - Knowing about private data
    - Knowing about related objects

Responsibilities

- Responsibilities are implemented by methods:
  - Some methods act alone and do a job
  - Some collaborate with other objects

  - **Example:** SaleTransaction class has Total()
  - Total() collaborates with subtotal() method of SaleLineItem object

GRASP Patterns

- **Creator**
  - Who creates an object?
- **Information Expert**
  - Which class should be responsible?
- **Low Coupling**
  - Support low dependency and increase reuse
- **Controller**
  - Who handles a system event?
- **High Cohesion**
  - How to keep complexity manageable?
- **Polymorphism**
  - How to handle behavior that varies by type?
- **Pure Fabrication**
  - How to handle a situation, when you do not want to violate High Cohesion and Low Coupling?
- **Indirection**
  - How to avoid direct coupling?
- **Law of Demeter (Don't talk to strangers)**
  - How to avoid knowing about unassociated objects?

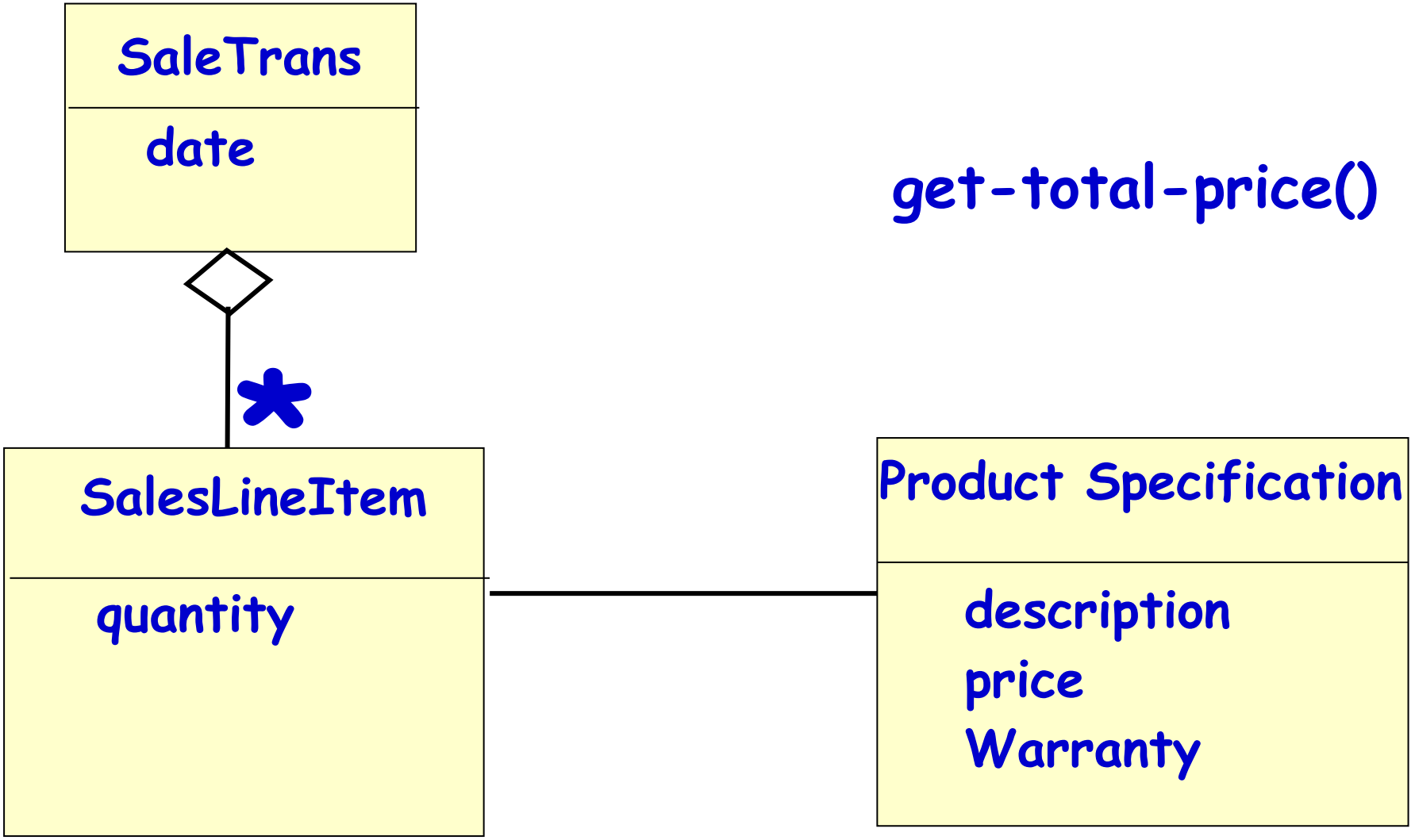
Grasp Patterns

Expert Pattern

- **Problem**: Which class should be responsible for doing a certain thing?

- **Solution**:
  - *Assign responsibility to the expert (class that has all/most information necessary to fulfil the required responsibility).*

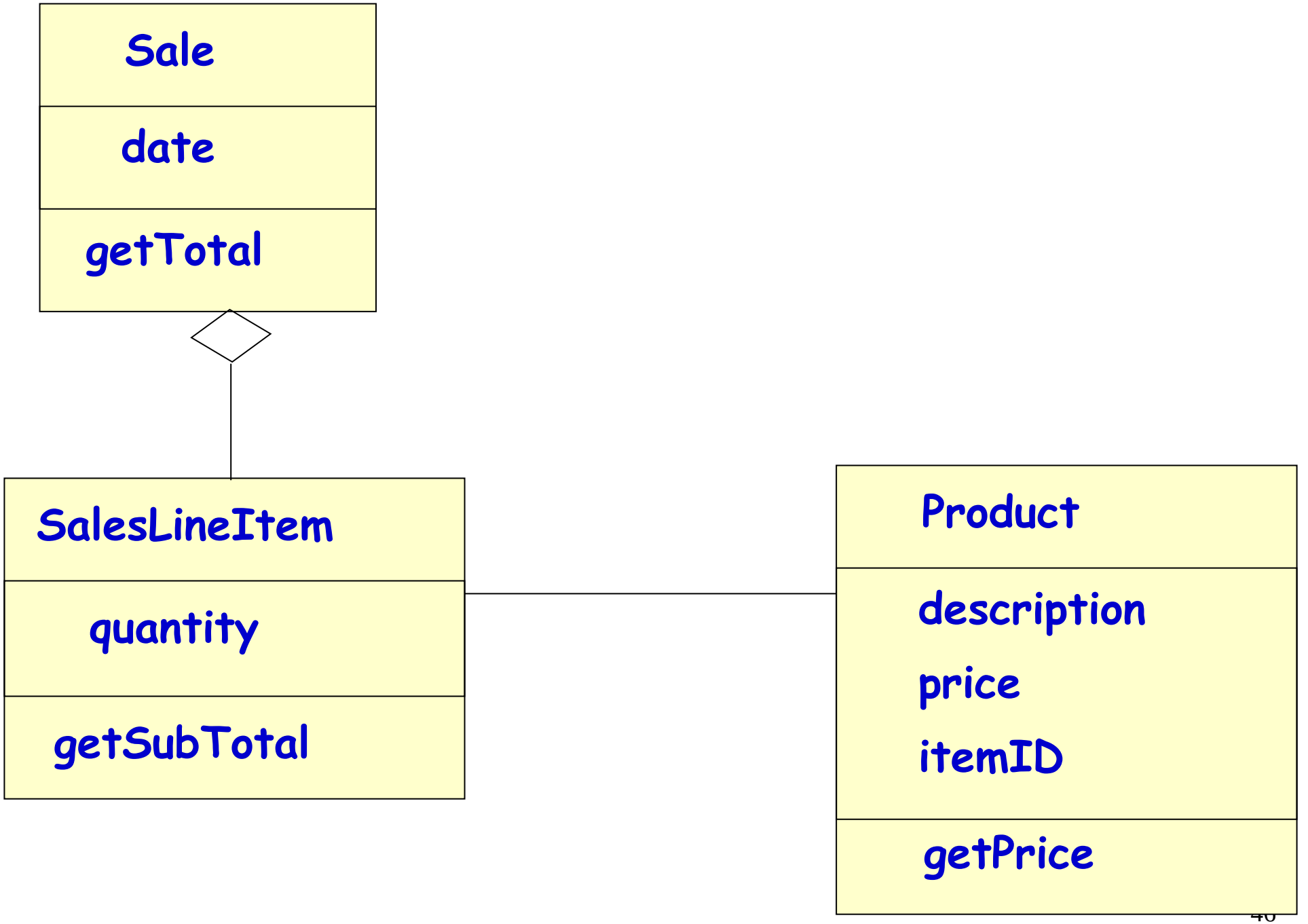
Which class is information expert for computing total price of a sales transaction?



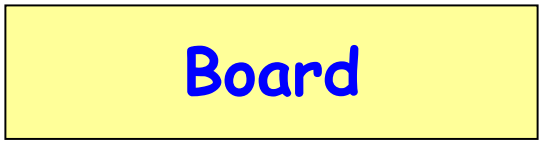
Example 1: Expert

Class Diagram

Collaboration Diagram


Sale
date
getTotal
SalesLineItem
quantity
getSubTotal
Product
description
price
itemID
getPrice

Example 2:Tic-Tac Toe

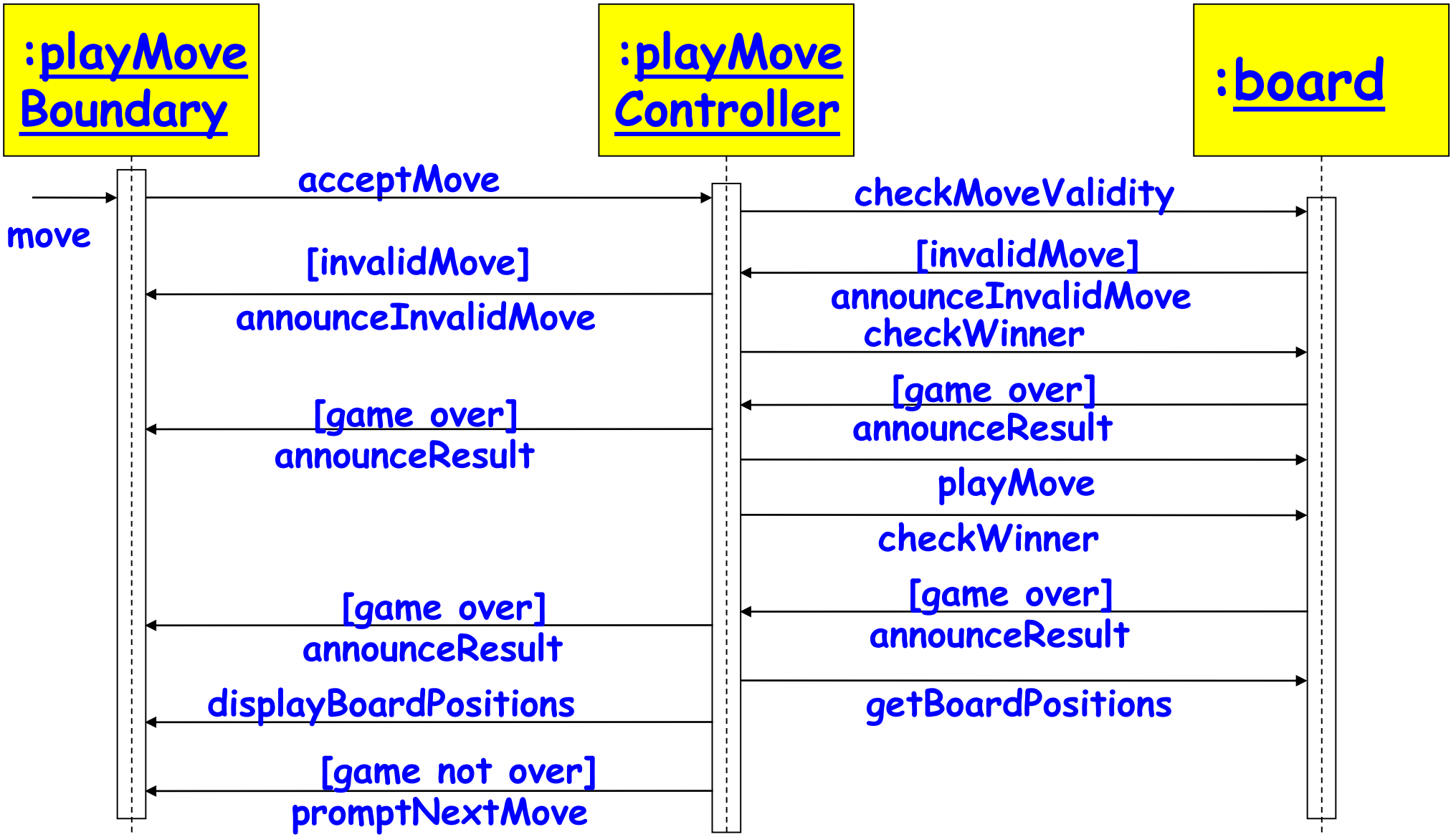


Initial domain model

Refined domain model

- Which class should check *game result* after a move?
- Which class should *play the computer's move*?

Example 2: Sequence Diagram



Sequence Diagram for the play move use case

Expert Pattern: An Analysis

- **Expert improves cohesion.**
  - Cohesion: the degree to which the information and responsibilities of a class are related to each other

- **How cohesion is improved?**
  - The information needed for a responsibility is in the same class as the responsibility itself.

Creator Pattern: Background

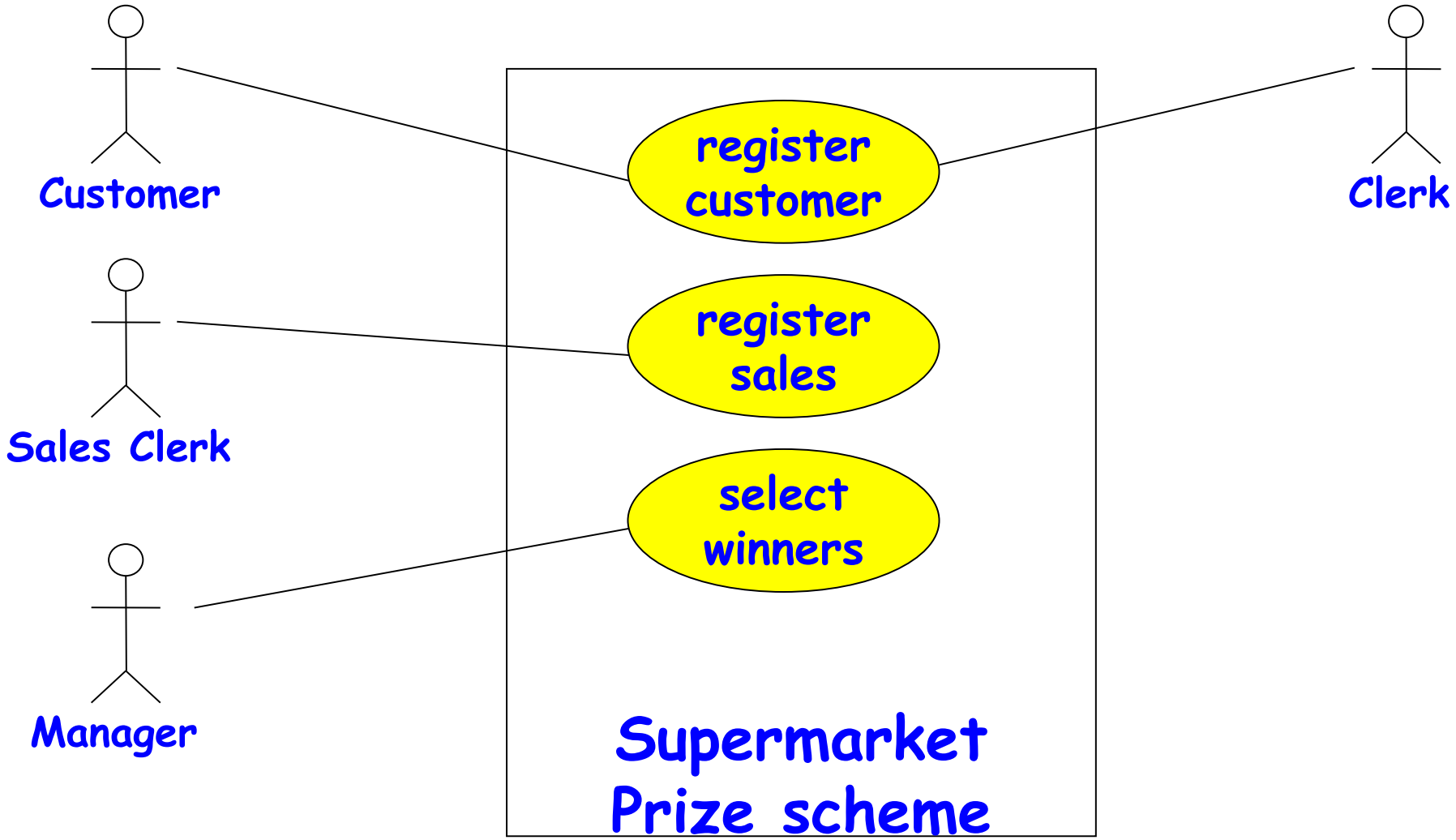
- Every object must be created somewhere.
- Consider making a class responsible for creating an object if:
  - **It is an inventory of objects of that type.**
  - **It has the information needed to initialize the object.**
  - **It will be the primary client of the object.**
- In a Library software, who creates a Member?

Creator Pattern

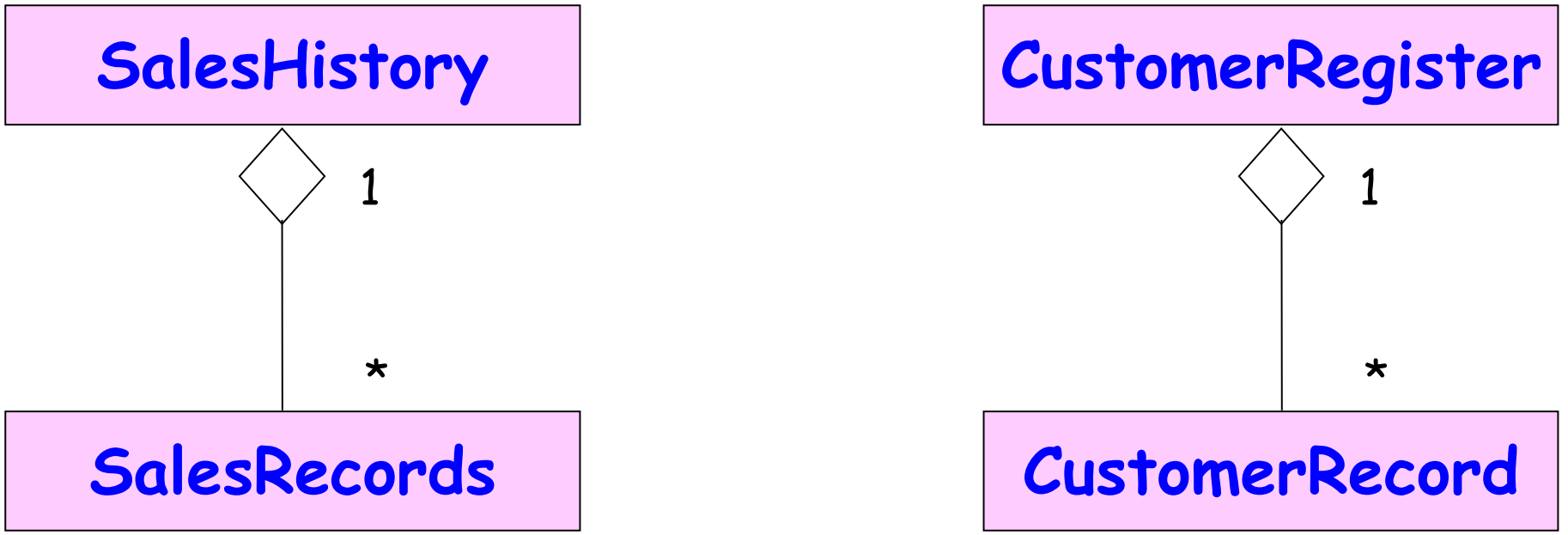
- **Problem:** Which class should be responsible for creating a new instance of some class?

- **Solution:** Assign a class C1 the responsibility to create objects of class C2 if
  - **C1 object aggregates objects of type C2**
  - **C1 object contains object of type C2**
  - **C1 objects closely use C2 objects**
  - **C1 object has the initializing data for C2 objects**

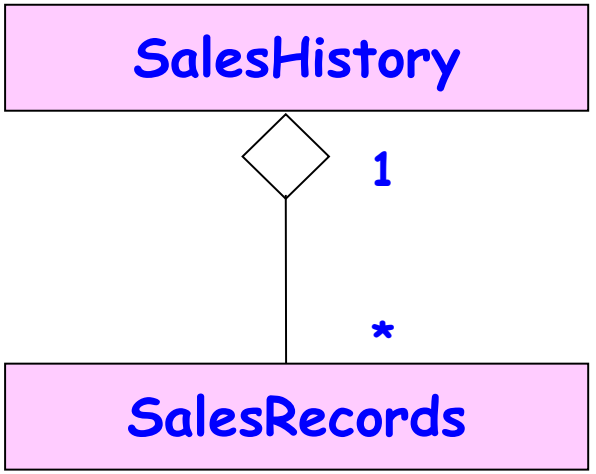
Example 2: Use Case Model

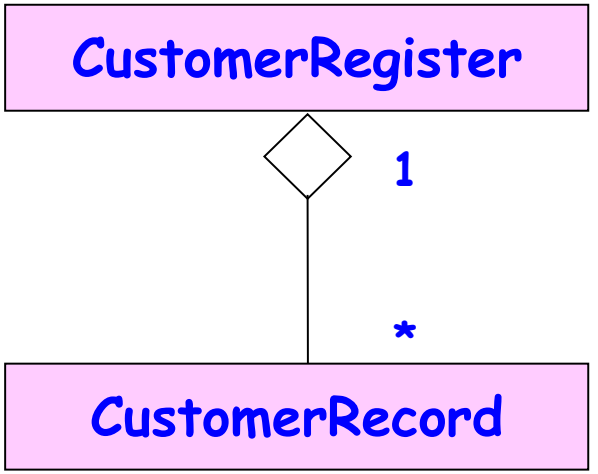


Example 2: Initial Domain Model



Example 2: Refined Domain Model





RegisterCustomerBoundary

RegisterCustomerController

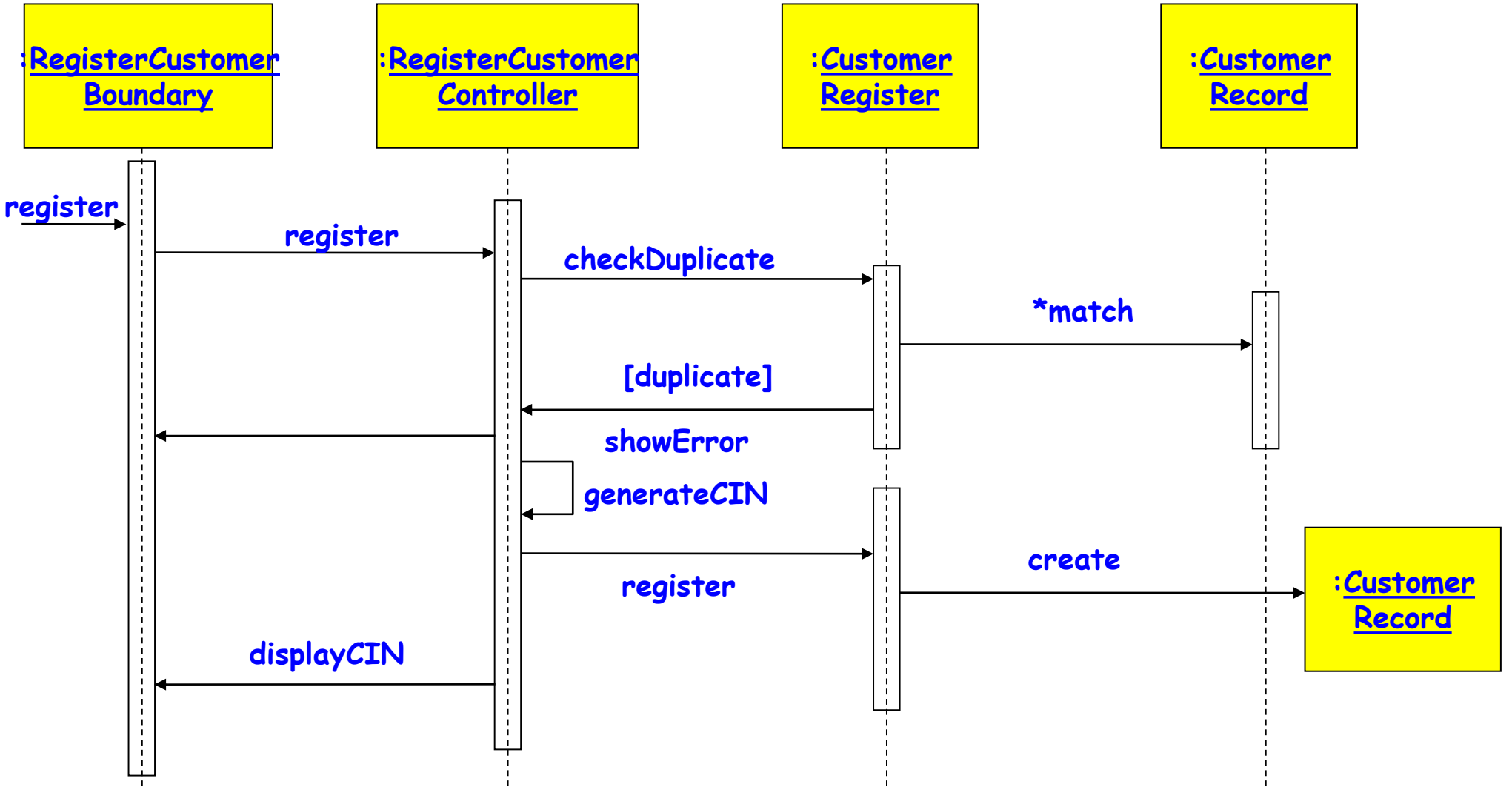
RegisterSalesBoundary

RegisterSalesController

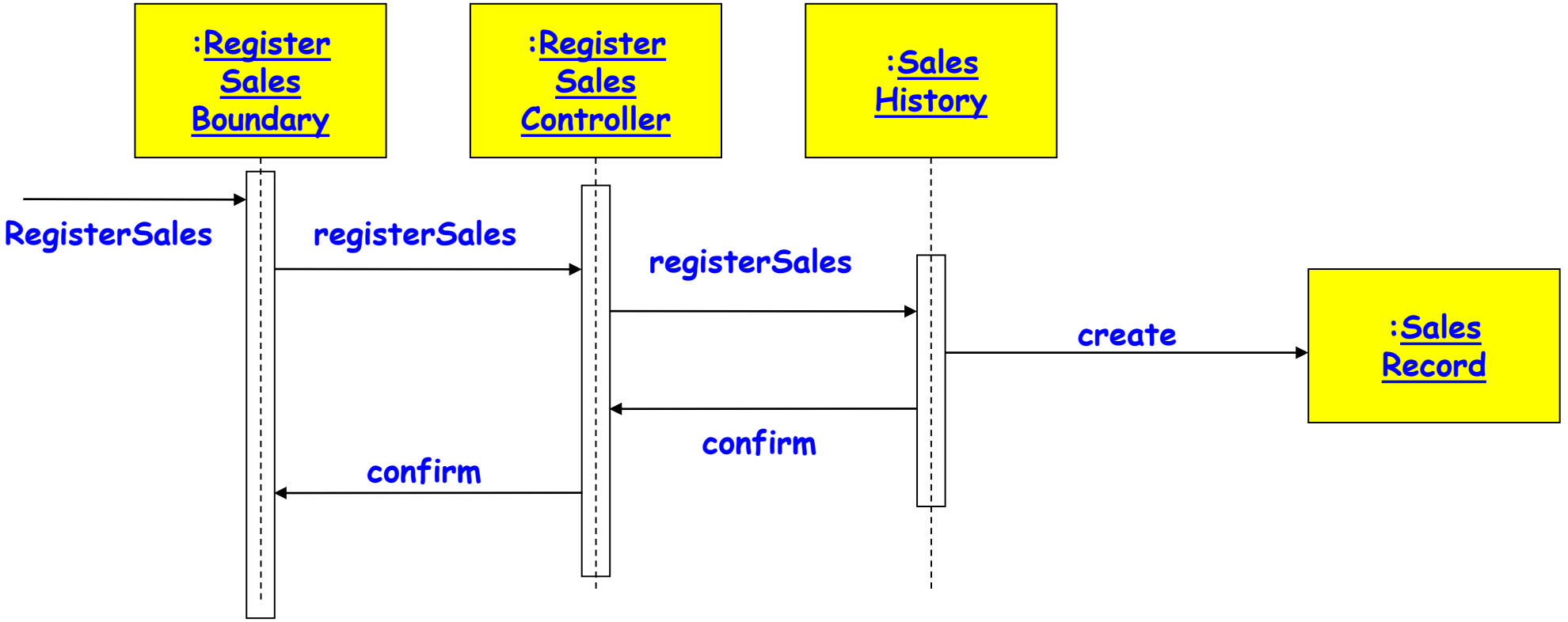
SelectWinnersBoundary

SelectWinnersControllers

Example 2: Sequence Diagram for the Register Customer Use Case

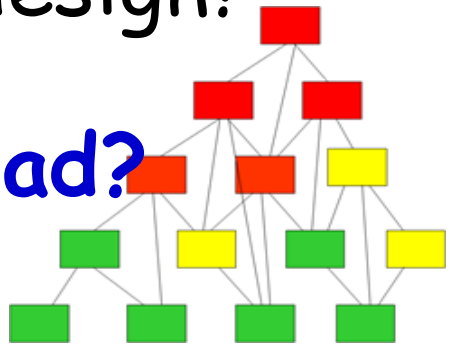


Example 2: Sequence Diagram for the Register Sales Use Case

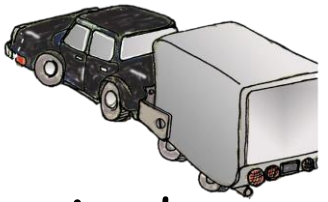


Low Coupling Pattern

- **Problem:** How to reduce coupling in a design?

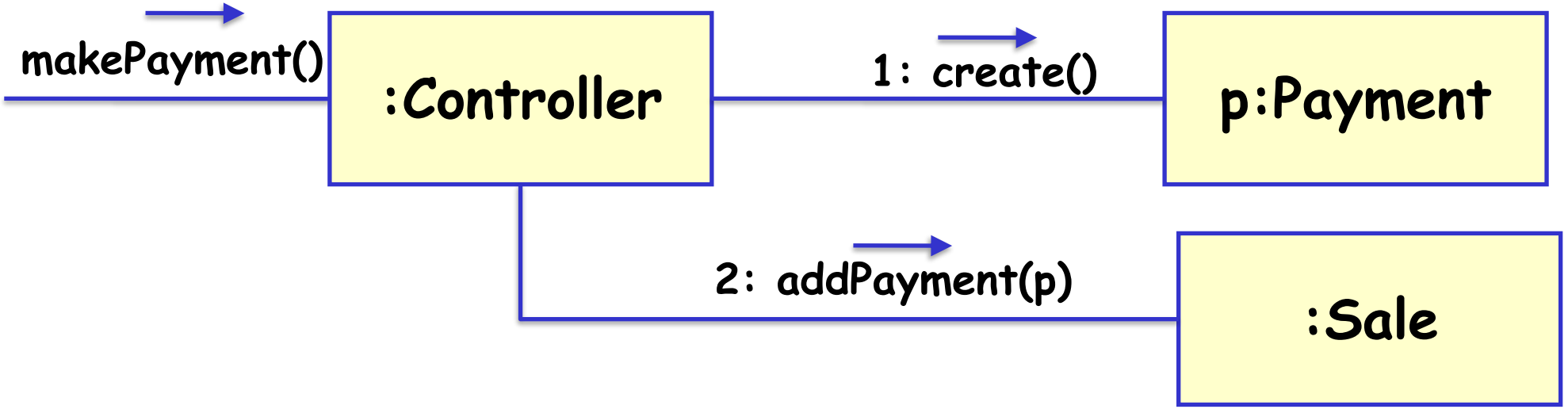


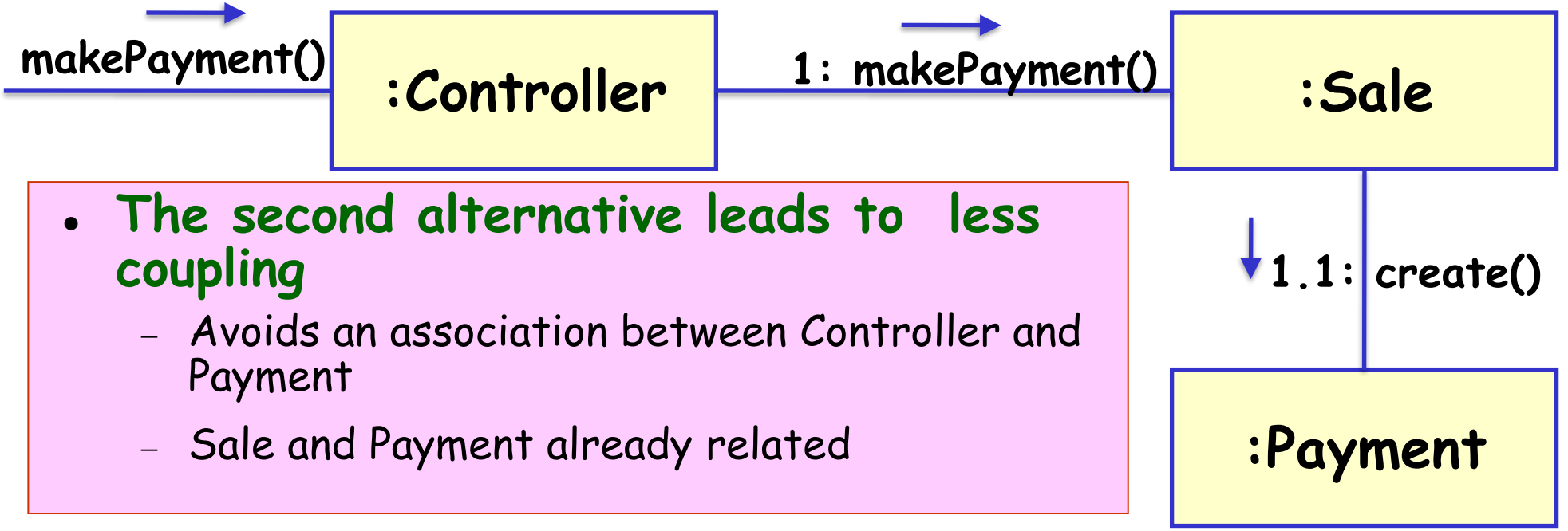
- **Why is classes having high coupling bad?**
  - Developer is forced to change all classes frequently because of changes to related classes…
  - Hard to understand in isolation, debug, test…
  - Hard to re-use
    - Because it requires presence of classes that it depends on



- **Solution:** ***Assign responsibilities so that coupling remains low.***
  - Use this principle to evaluate alternatives

Two alternative responses to "Who creates Payment?"



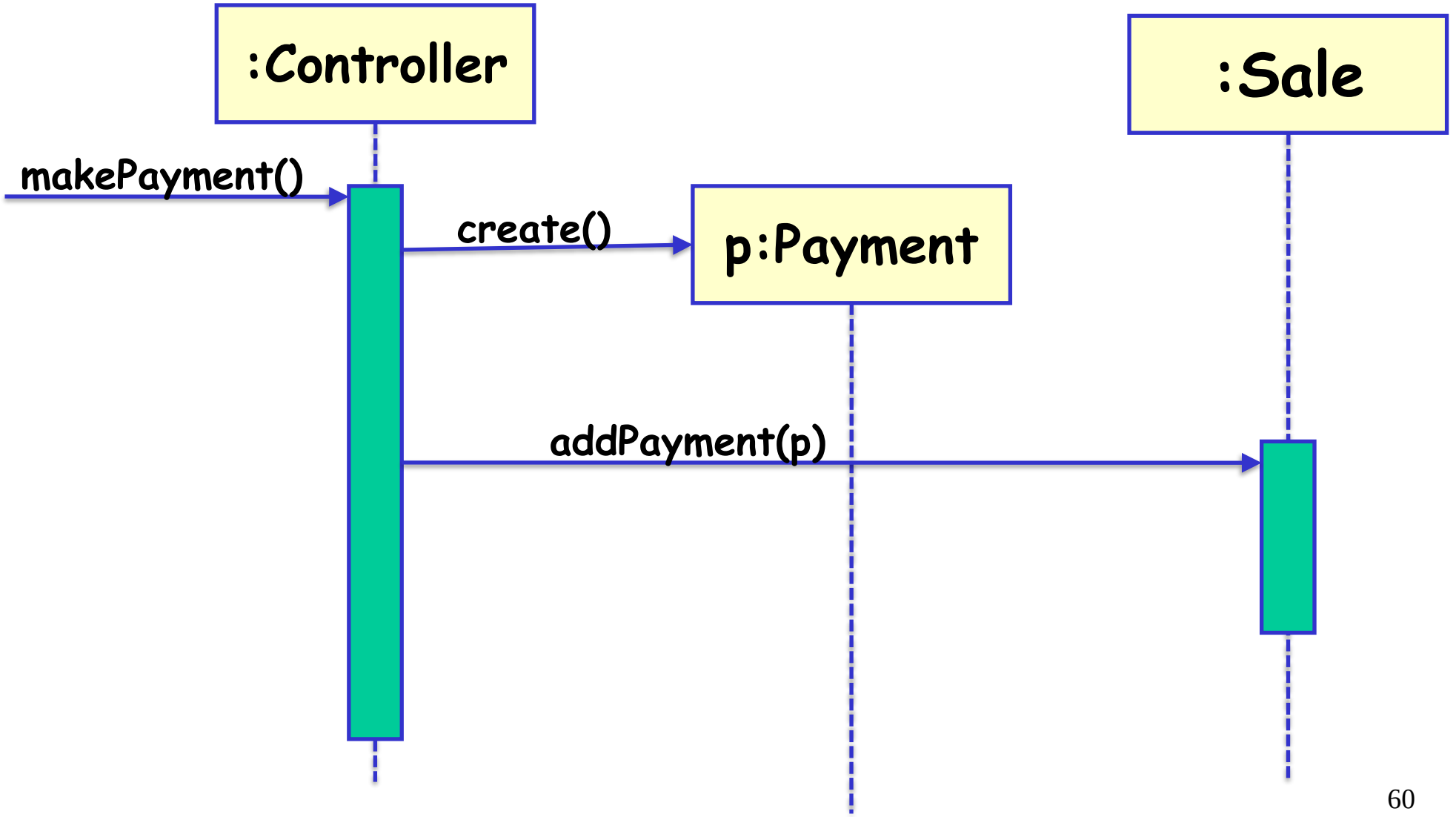


- **The second alternative leads to less coupling**
  - Avoids an association between Controller and Payment
  - Sale and Payment already related

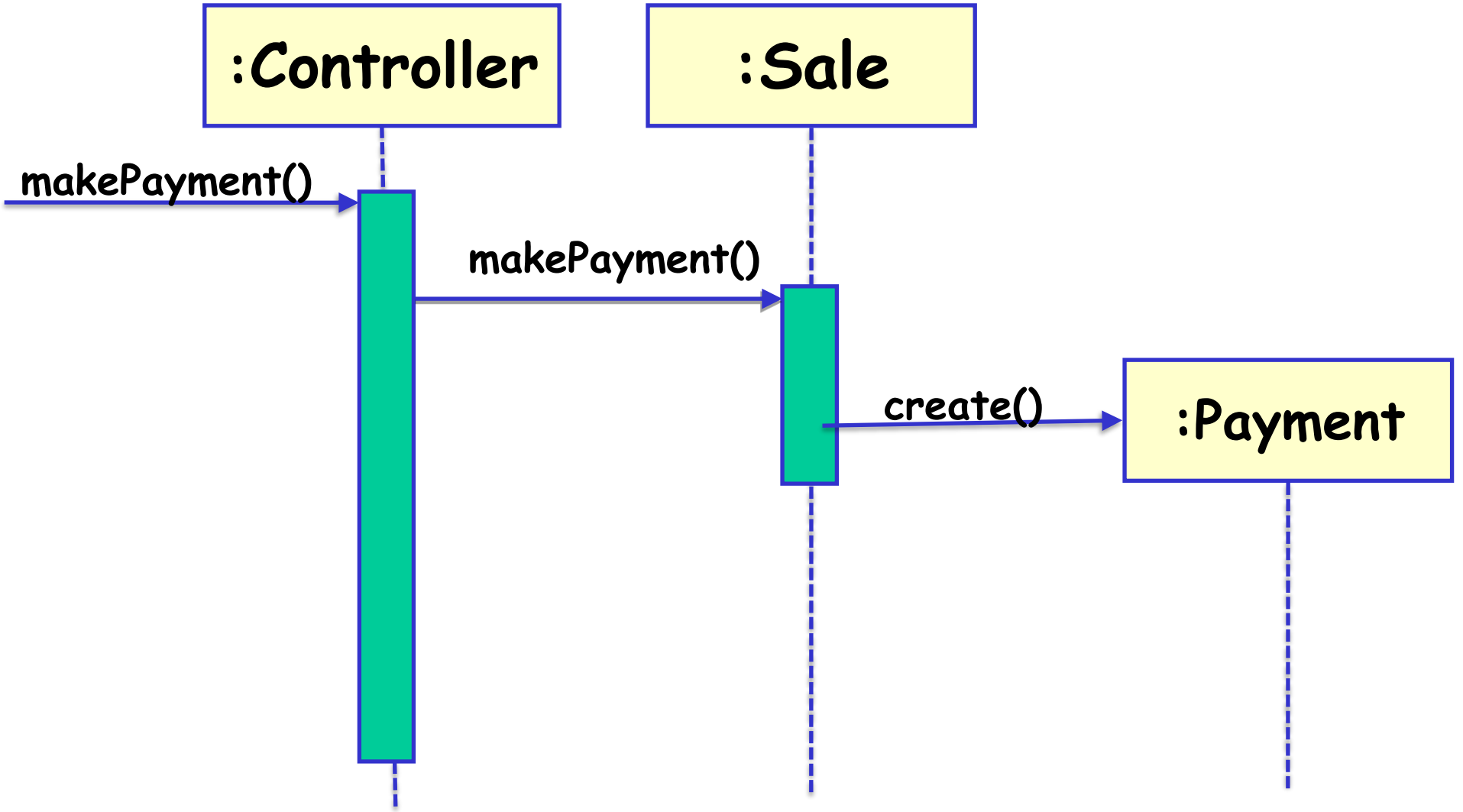
High Cohesion

- **Problem:** How to increase cohesion?
- Problems when a class has low cohesion:
  - **Hard to comprehend**
  - **Hard to reuse**
  - **Hard to maintain**
  - **Frequent changes required**

Example: Controller has Diverse Responsibilities



Better design: Controller delegates, has higher cohesion



Pure Fabrication: Background

- **High Cohesion** **But how???**
  - **Problem**: To keep complexity manageable. Classes implement a set of cohesive tasks.
  - **Solution**: Assign focused and related responsibilities to classes.

- **Low Coupling** **But how???**
  - **Problem**: To support low dependency and increased reuse.
  - **Solution**: Assign responsibilities so that coupling remains low.

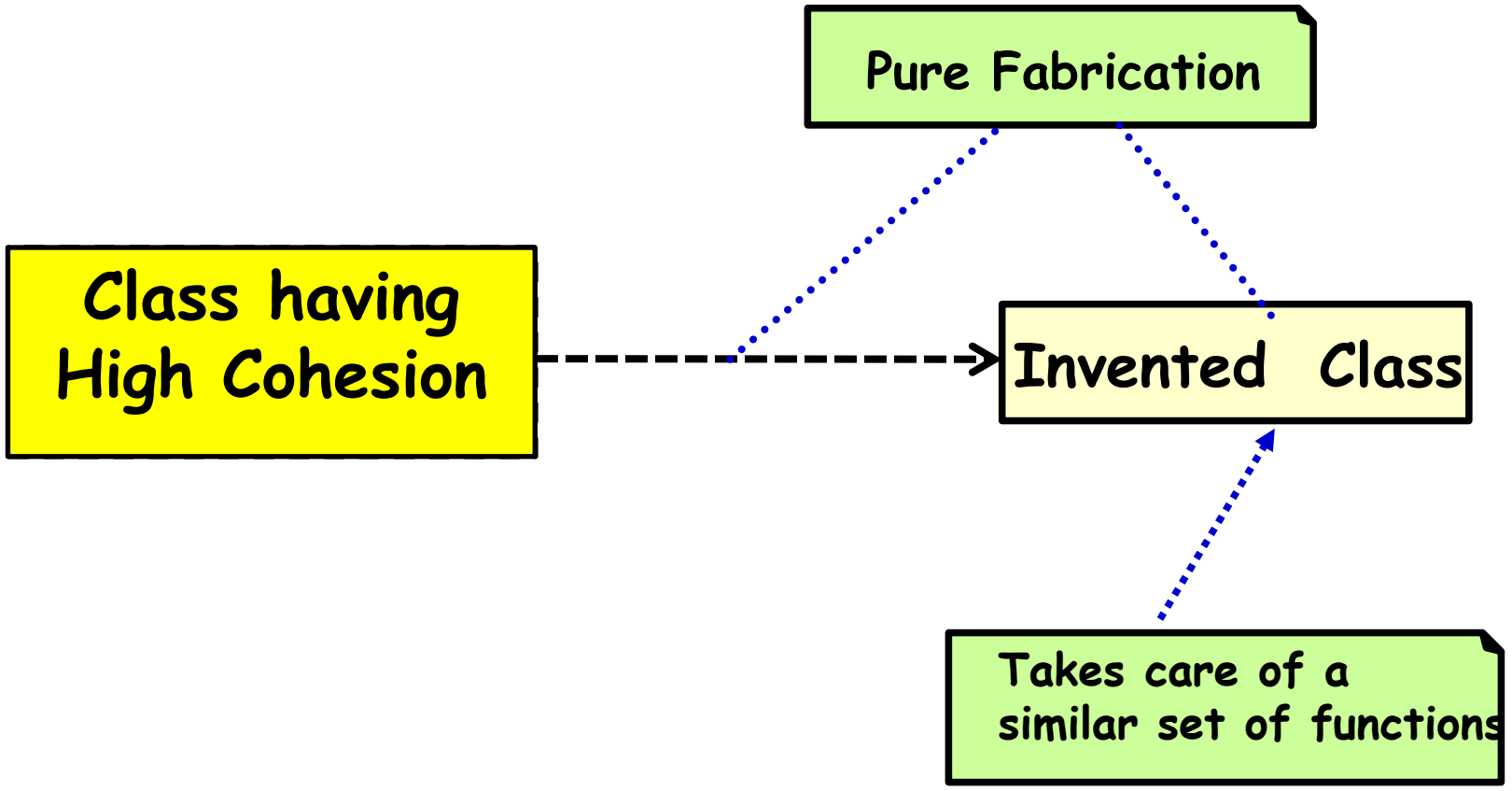
Pure Fabrication – Background

- Suppose a class has responsibilities unrelated to its main task.
  - **It is a bad design** --- low cohesion and high coupling.
  - **But, how to improve the design?**

Pure Fabrication

- **Problem:**
  - How to improve a design when a class has very high coupling and low cohesion?
- **Solution:**
  - **Assign a highly cohesive set of responsibilities to an artificial class**
  - May not represent anything in the problem domain...
  - Created only to support high cohesion, low coupling, and reuse...

Pure Fabrication: Explanation



Fabrication step: Assign a highly cohesive set of responsibilities to an artificial class.

Pure Fabrication Example

Suppose we need to save instances of **SaleTrans** in a relational database.

To which class would you assign this responsibility?

SaleTrans

SaleTrans has all the information, so SaleTrans is suggested by Expert pattern.

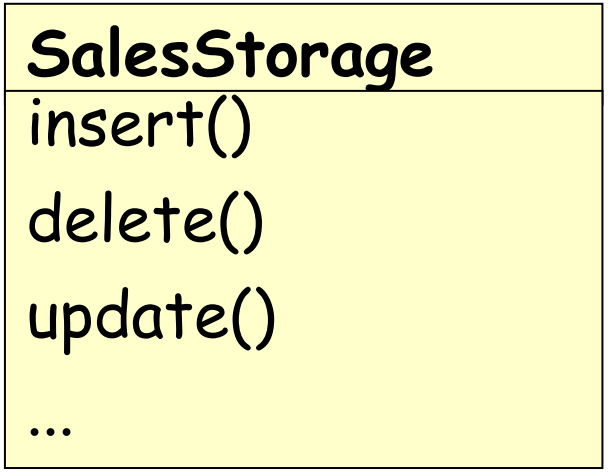
To manage data transfer to and from a relational database will require a large number of operations … insert, delete, update, select, rollback, commit, buffer management, …

Cohesion of SaleTrans class reduces…

Pure Fabrication

Create a new class for the database access related responsibilities:

- **SalesStorage class is made up from imagination; it is not present in the Domain Model**



SaleTrans is now well-designed - high cohesion, low coupling

SalesStorage class is cohesive: Its sole purpose is to store/retrieve objects to/from a relational database

Pure Fabrication: Java Snippet

```
class SaleTrans {
// Process and record the sale for each line item
// Use SaleStorage class for persistent storage
}
class SalesLineItem {
//Process items of similar type and return subtotal
// Use SaleStorage class for persistent storage
}
class SaleStorage{
// Form JDBCDatabase objects to store data to the
  database
}
```

Pure Fabrication: An Analysis

- Many GoF object-oriented patterns are examples of Pure Fabrication.:
  - **Examples: Facade, Adapter, Mediator, command, strategy,…**

- Pure Fabrication may contraindicate the Expert pattern.
  - **An informed decision is needed.**

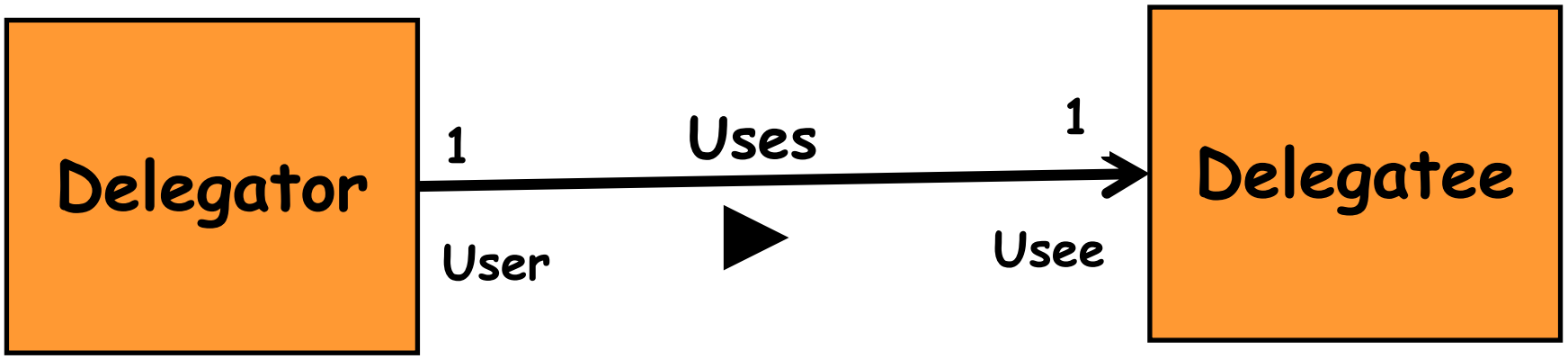
- Increases Cohesion and reusability

Pure Fabrication: Final Analysis

Objects can be divided into two groups:

- **Representational:**
  - Related to representation of entities and their components, such as *Sale, Vehicle, Document*, …
  - These are generally identifiable in the problem description.
- **Behavioural:**
  - Often concocted, also may represent an algorithm.
  - Examples includes *controller* classes for use cases, *Command* class, *Strategy* class, etc
  - **These are generally not in the problem domain model, and are mostly pure fabrication.**

Pure Fabrication Final Analysis: Delegate Unrelated Responsibility



Indirection Pattern

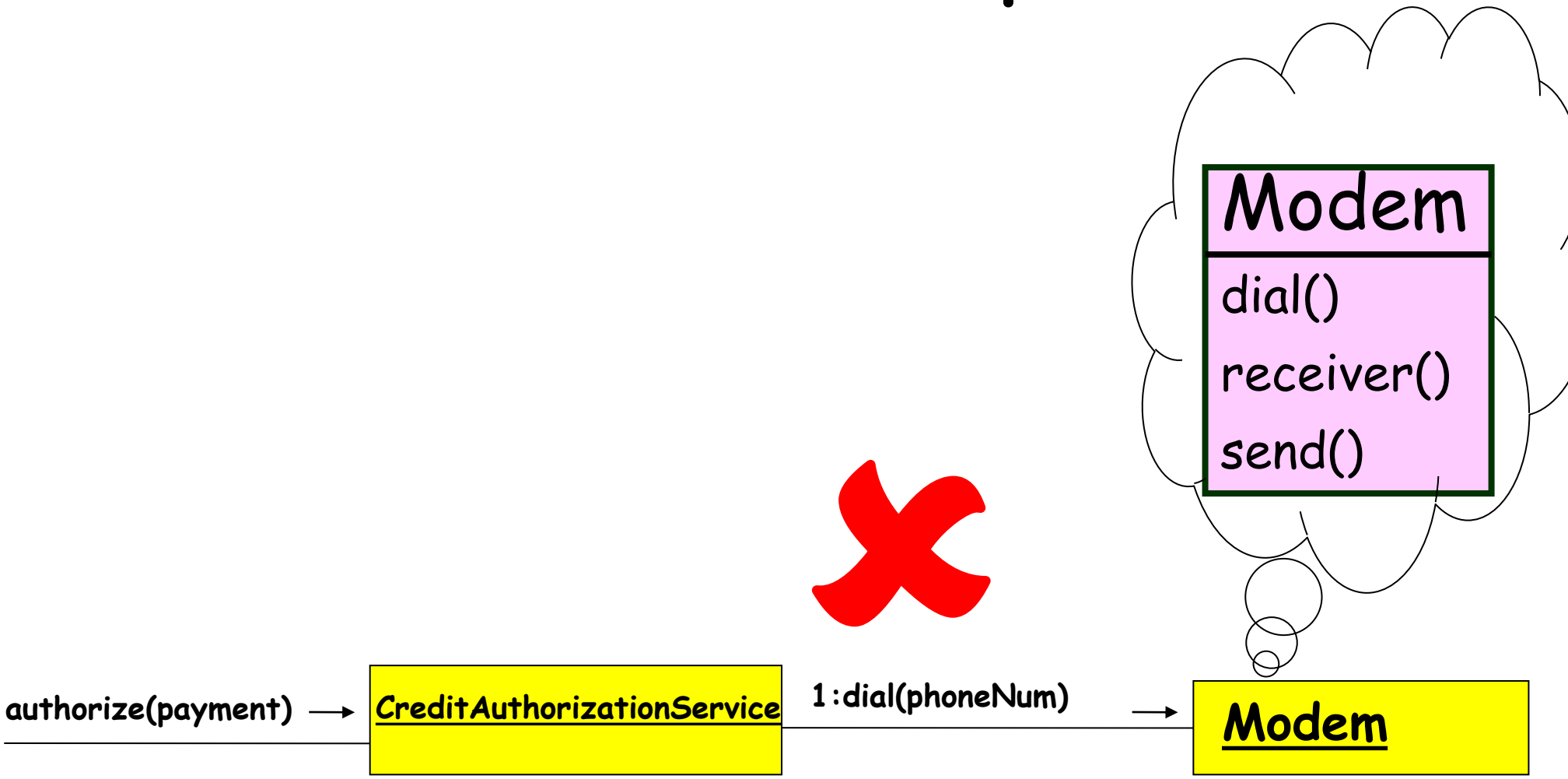
Problem:

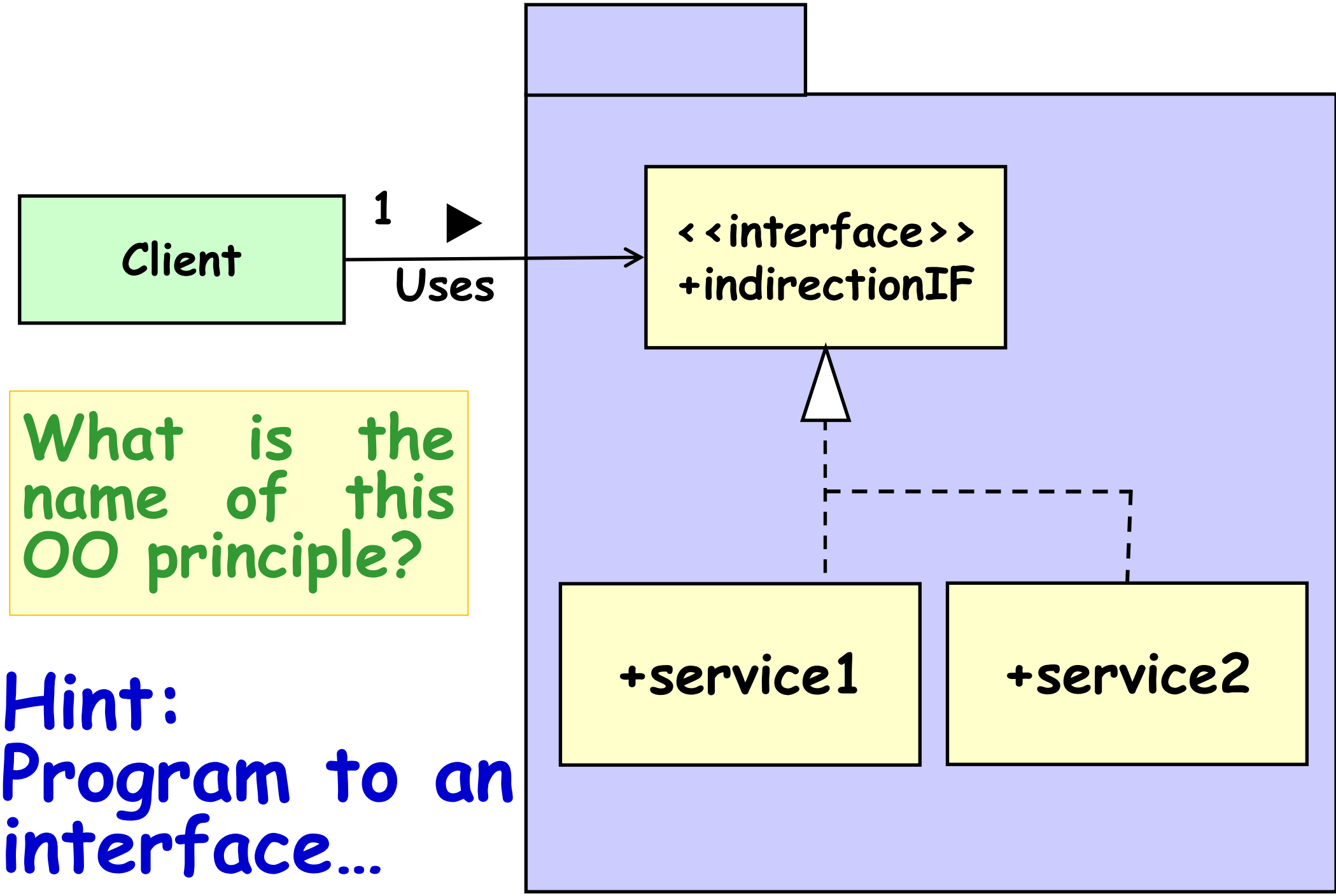
- **How to avoid direct coupling between classes?**
- How to decouple objects so that low coupling is achieved and changes become easy?

Solution:

- Depend on an interface class,
  - So that objects are not directly coupled.

Indirection: Example

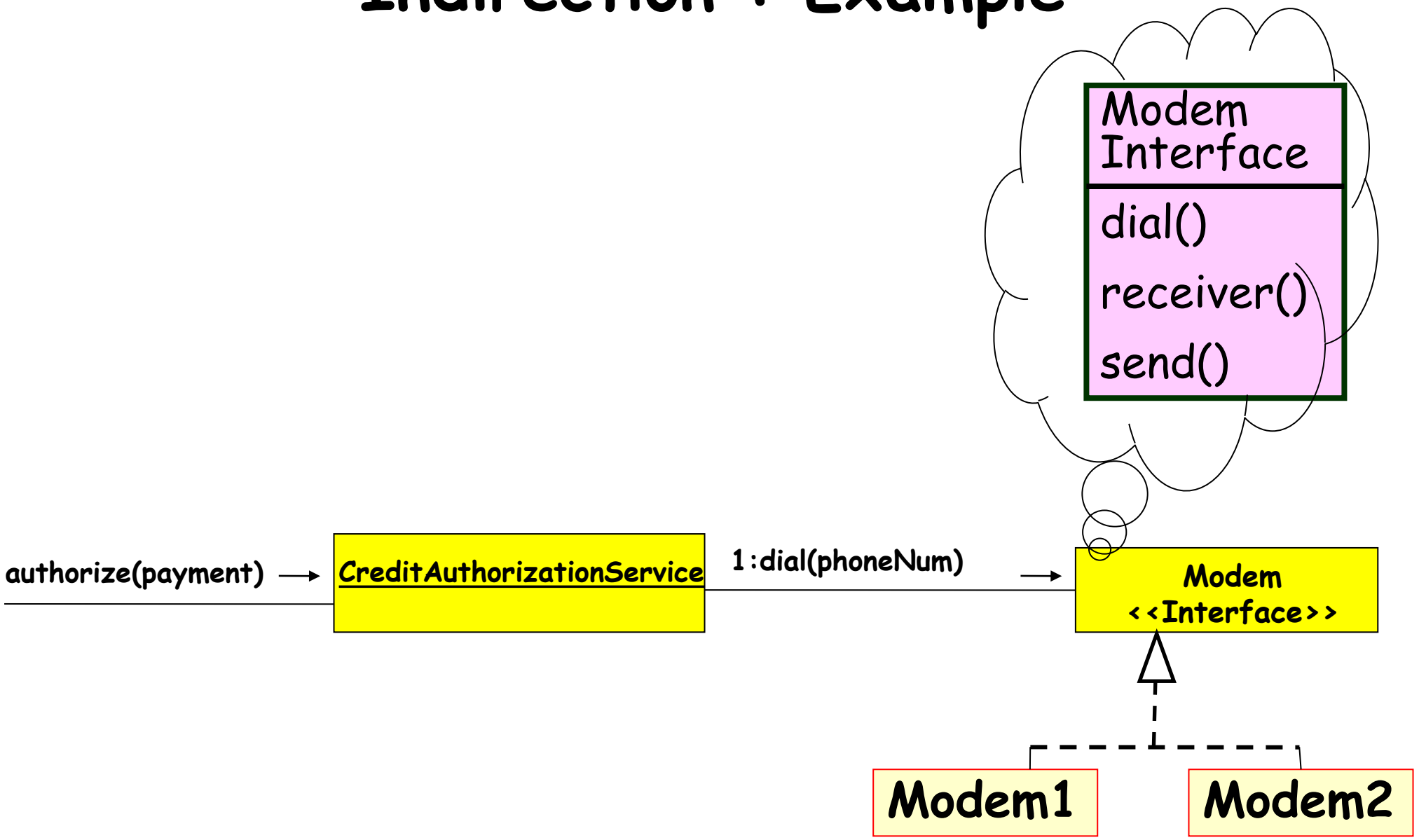




What is the name of this OO principle?

Hint:
Program to an interface…

Indirection : Example



Indirection Advantages

- Low coupling
- Promotes reusability