Software Engineering Project

Morgan Ericsson

morgan@cse.gu.se

@morganericsson

morganericsson

UNIVERSITY OF GOTHENBURG

CHALMERS

Elements of the Object Model

- Abstraction
  - denote the essential characteristics
  - provide (conceptual) boundaries, relative to the perspective of the viewer
- Encapsulation
  - compartmentalising the structure and behavior
  - separate the interface and its implementation

Elements of the Object Model (cont'd)

- Modularity
  - a system that has been decomposed into a set of cohesive and loosely coupled modules.
- Hierarchy
  - ranking or ordering of abstractions.

Object-Oriented Software Development

- Object-Oriented Programming (OOP)
  - method of implementation
  - programs are organized as cooperative collections of objects
  - objects are instances of classes
  - classes are members of a hierarchy

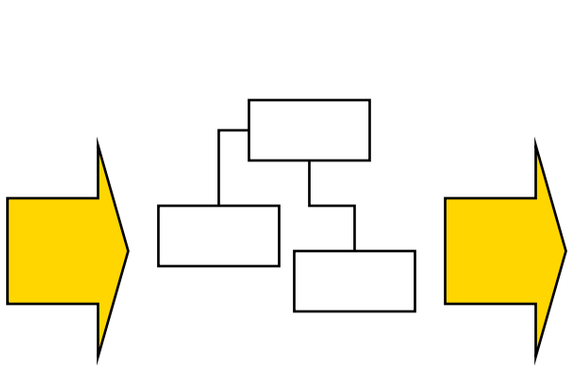
Object-Oriented Software Development

- Object-Oriented Analysis (OOA)
  - emphasises the building of real-world models, using an object-oriented view of the world
- Object-Oriented Design (OOD)
  - object-oriented decomposition
  - a notation for depicting both logical and physical as well as static and dynamic models of the system

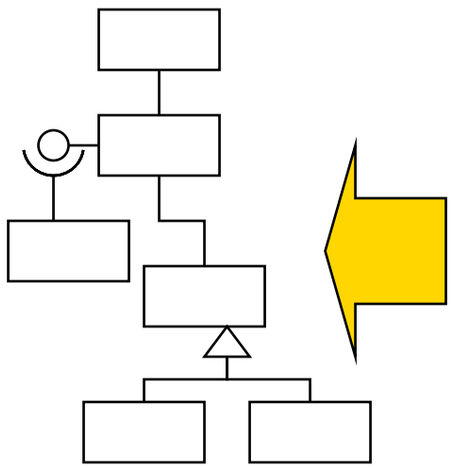
So…

Problem domain

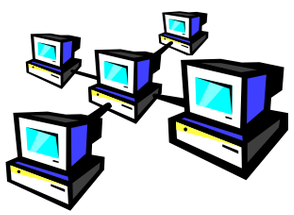
Domain model

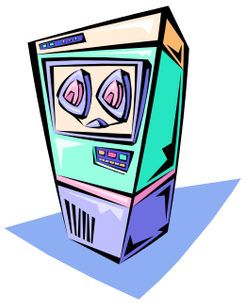


Design classes



Solution domain



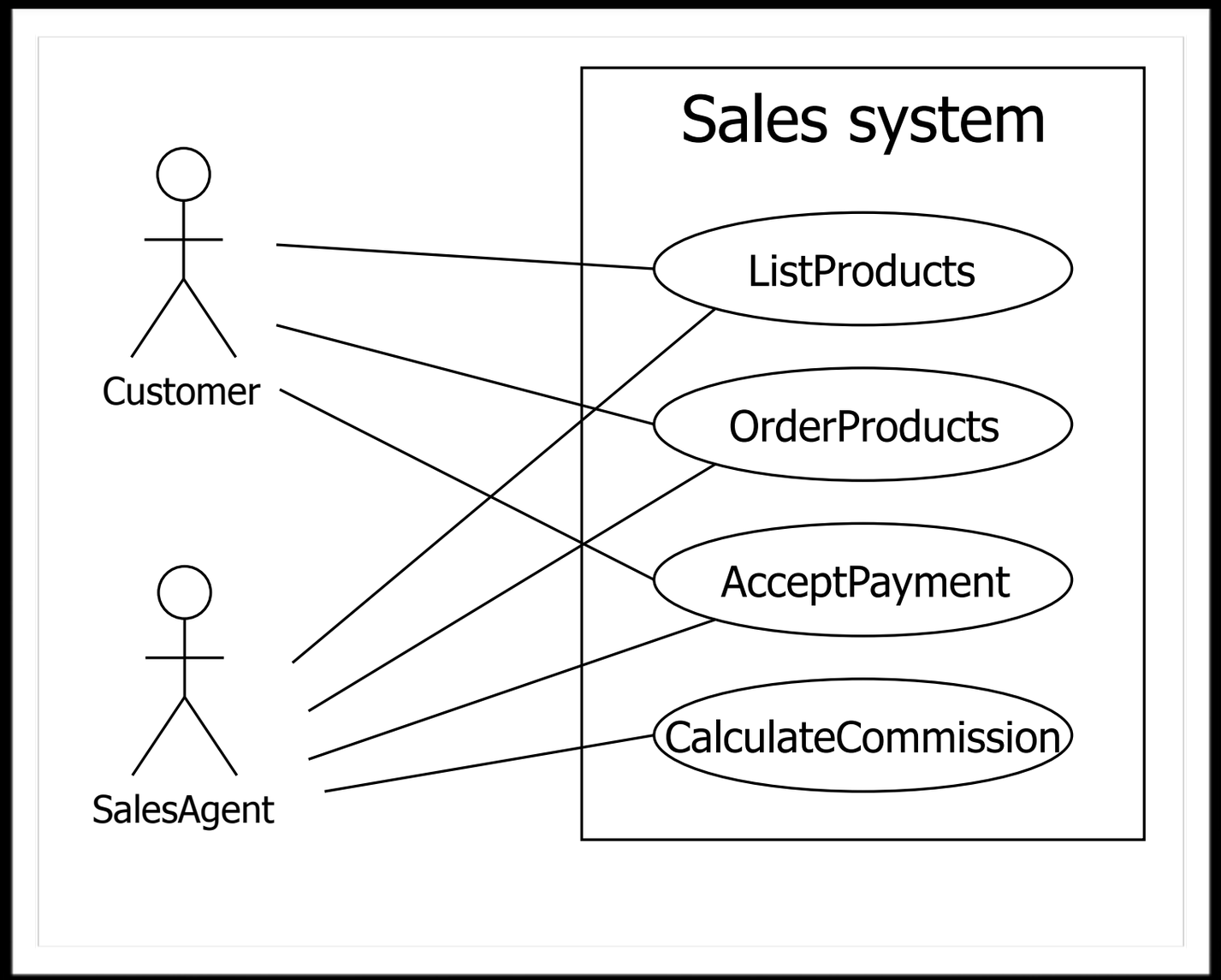


java.util

Functional Decomposition

- Decompose according to the functionality the system should provide
  - works well in "stable" systems
- But,
  - naive, modern system provide many functions
  - systems and functionality evolve
  - interoperability can be difficult

Functional Decomposition



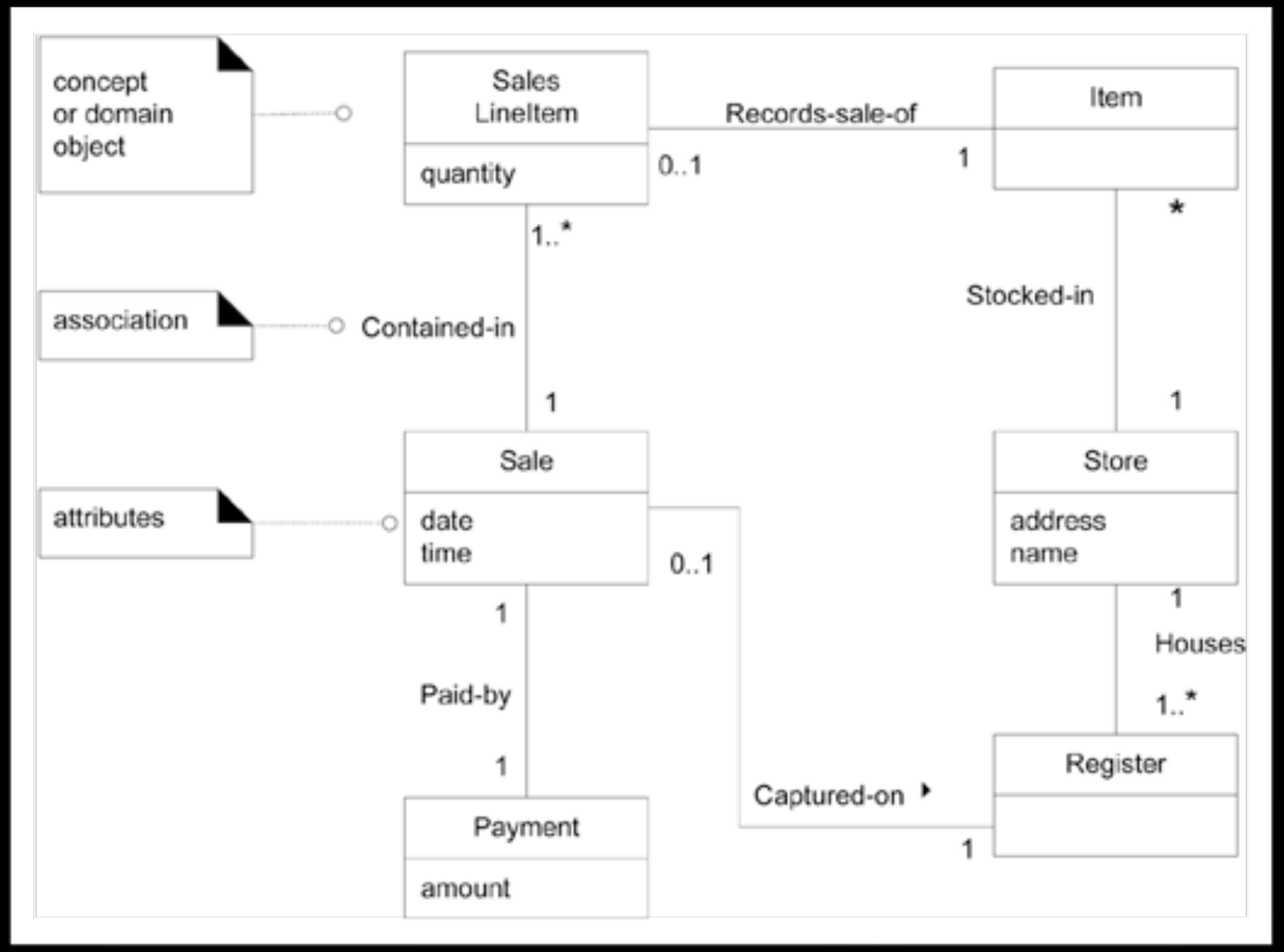
Object-Oriented Decomposition

- Decompose into the objects that the system should manipulate
- A better choice for complex and evolving systems
- But, how do we identify the objects?
  - often an iterative process
  - and seldom algorithmic

Object-Oriented Decomposition

- Create an object-oriented model of the "real world" (domain)
  - decompose into constituents
  - abstract these to objects
  - create relations between objects
- Domain model

Example



Domain models are often expressed using graphical notations

Perspectives

- Informal set of classes
  - the objects can be considered a super set of the classes the final design will contain
    - focus on objects and relations, not implementation details
- A dictionary of important concepts/terms
  - attributes can help define

Example

Main Success Scenario (or Basic Flow):

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs the completed sale and sends sale and payment information to the external Accounting (for accounting and commissions) and Inventory systems (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Example

Extensions (or Alternative Flows):

. . .

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

Example

- Sale
- Cashier
- CashPayment
- Customer
- SalesLineItem
- Store
- Item
- ProductDescription
- Register
- ProductCatalog
- Ledger

Example


Register
Item
Store
Sale
Sales LineItem
Cashier
Customer
Ledger
Cash Payment
Product Catalog
Product Description

Complete?

- It is ok to not include all classes
- Focus on what is important at the moment / iteration / sprint
- You will get to the rest later...

Example

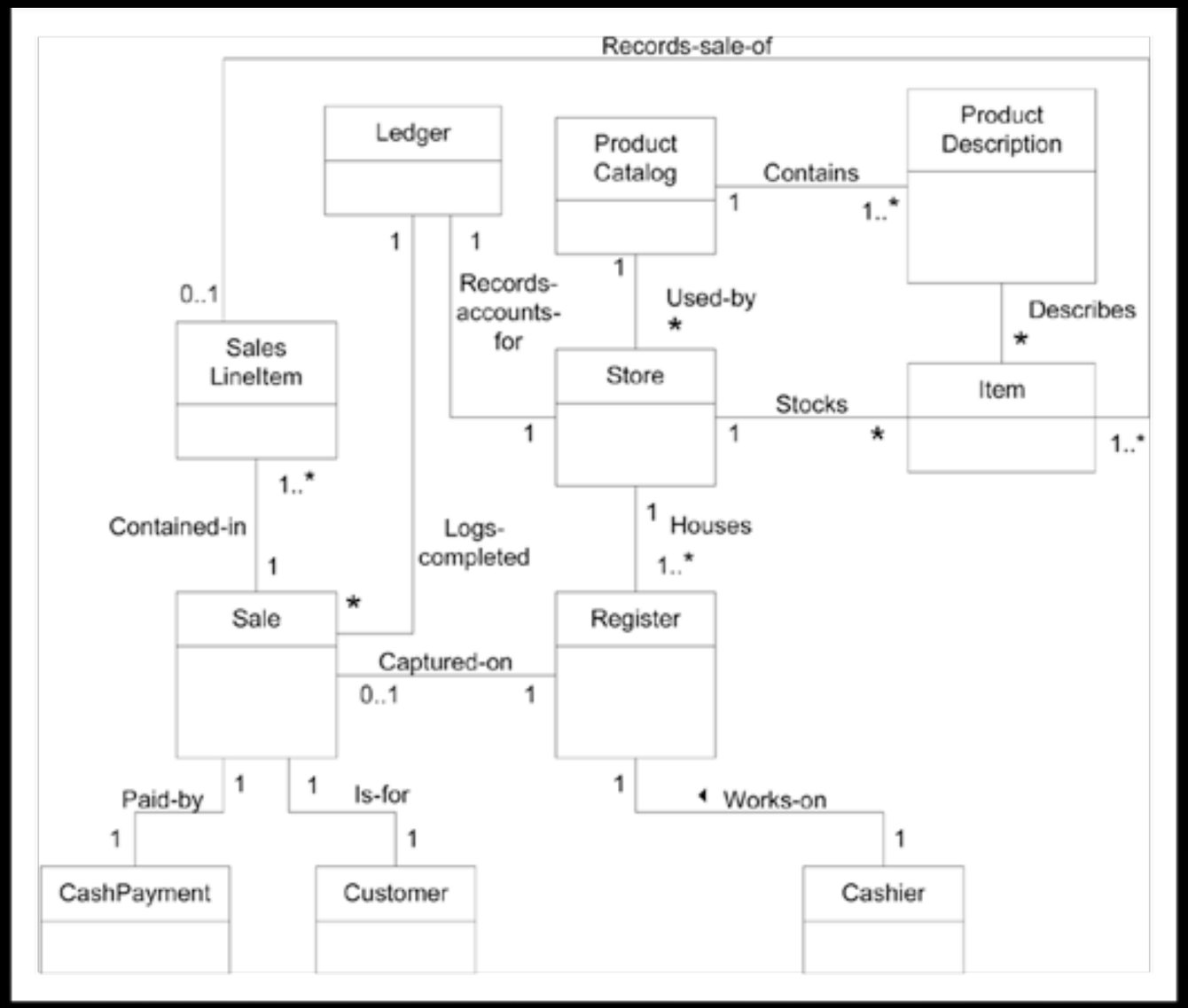
- What about receipt?
  - an important part of the domain
    - but the same information is contained in other classes (e.g., Sale)
    - receipt not important during sale (!!!)
- Not right or wrong – use your judgement!

Attributes vs Classes

Associations

- A relation between objects that indicate a meaningful and interesting relation
- Denotes relations that should be saved (for a specific amount of time)

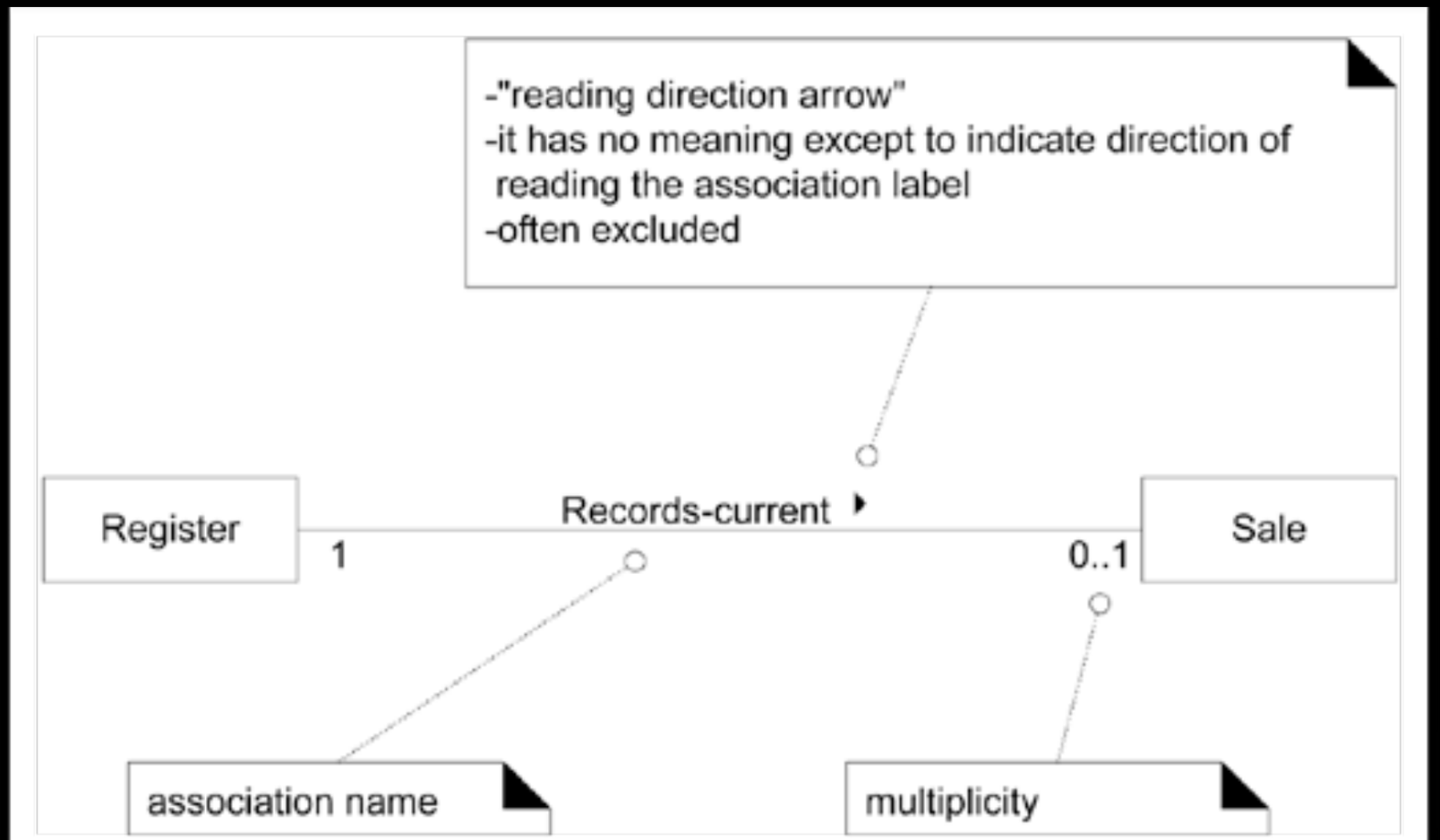
Example


Records-sale-of
Ledger
Product Catalog
Contains
Product Description
1
1..*
1
1
1
Records-accounts-for
Used-by
*
Describes
*
0..1
Sales LineItem
Store
Item
Stocks
1
1
*
1..*
1..*
Contained-in
1
Logs-completed
1
Houses
1..*
Sale
*
Register
Captured-on
0..1
1
1
1
Paid-by
1
Is-for
1
Works-on
1
1
CashPayment
Customer
Cashier

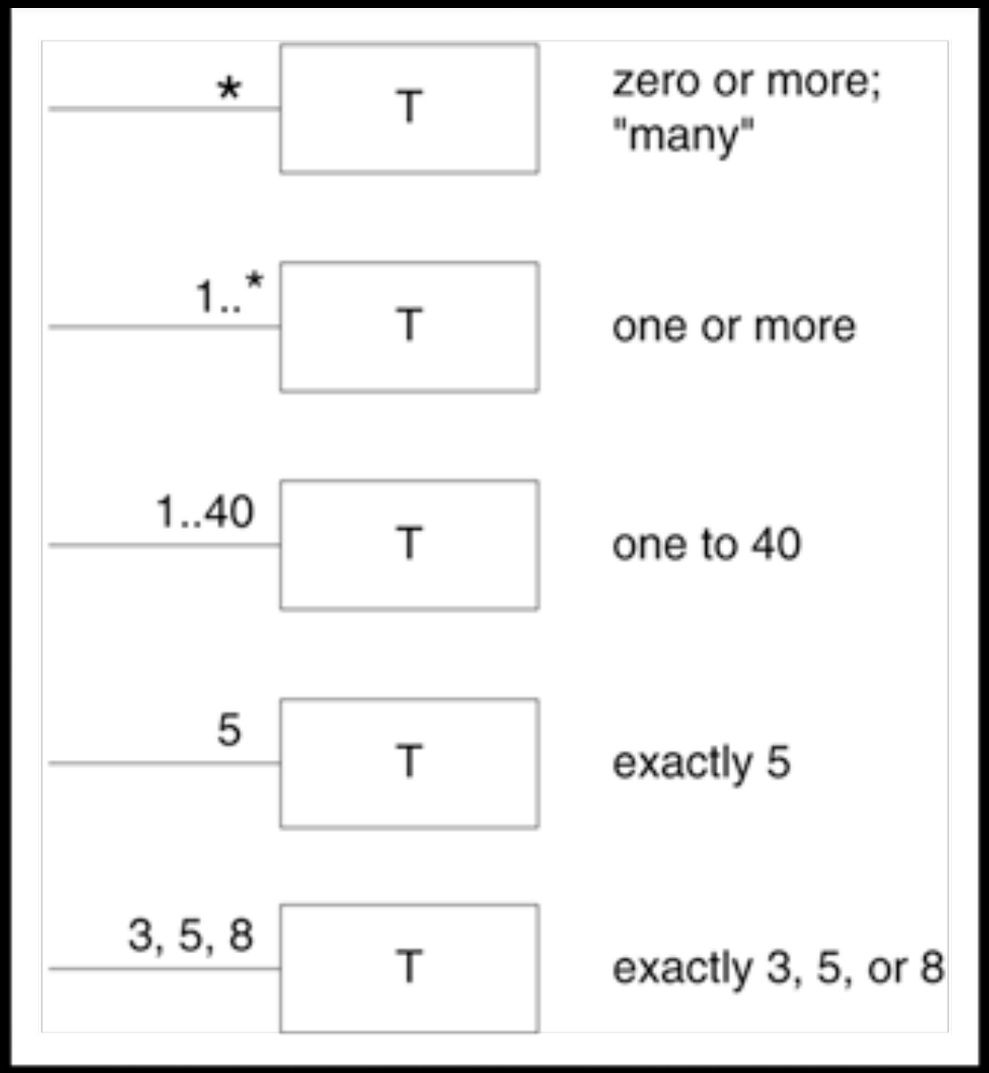
Example

- Is a Sale – SalesLineItem relation important?
  - yes, exchanges and returns, for example
- Is a Cashier – ProductDescription relation important?
  - no

Directions, Names and Multiplicity



Multiplicity



Common Associations

Category	Examples
A is a physical part of B	*Drawer — Register (or more specifically, a POST)* *Wing — Airplane*
A is a logical part of B	*SalesLineItem — Sale* *FlightLeg—FlightRoute*
A is physically contained in/on B	*Register — Store, Item — Shelf* *Passenger — Airplane*
A is logically contained in B	*ItemDescription — Catalog* *Flight— FlightSchedule*
A is a description for B	*ItemDescription — Item* *FlightDescription — Flight*
A is a line item of a transaction or report B	*SalesLineItem — Sale* *Maintenance Job — MaintenanceLog*
A is known/logged/recorded/reported/captured in B	*Sale — Register* *Reservation — FlightManifest*

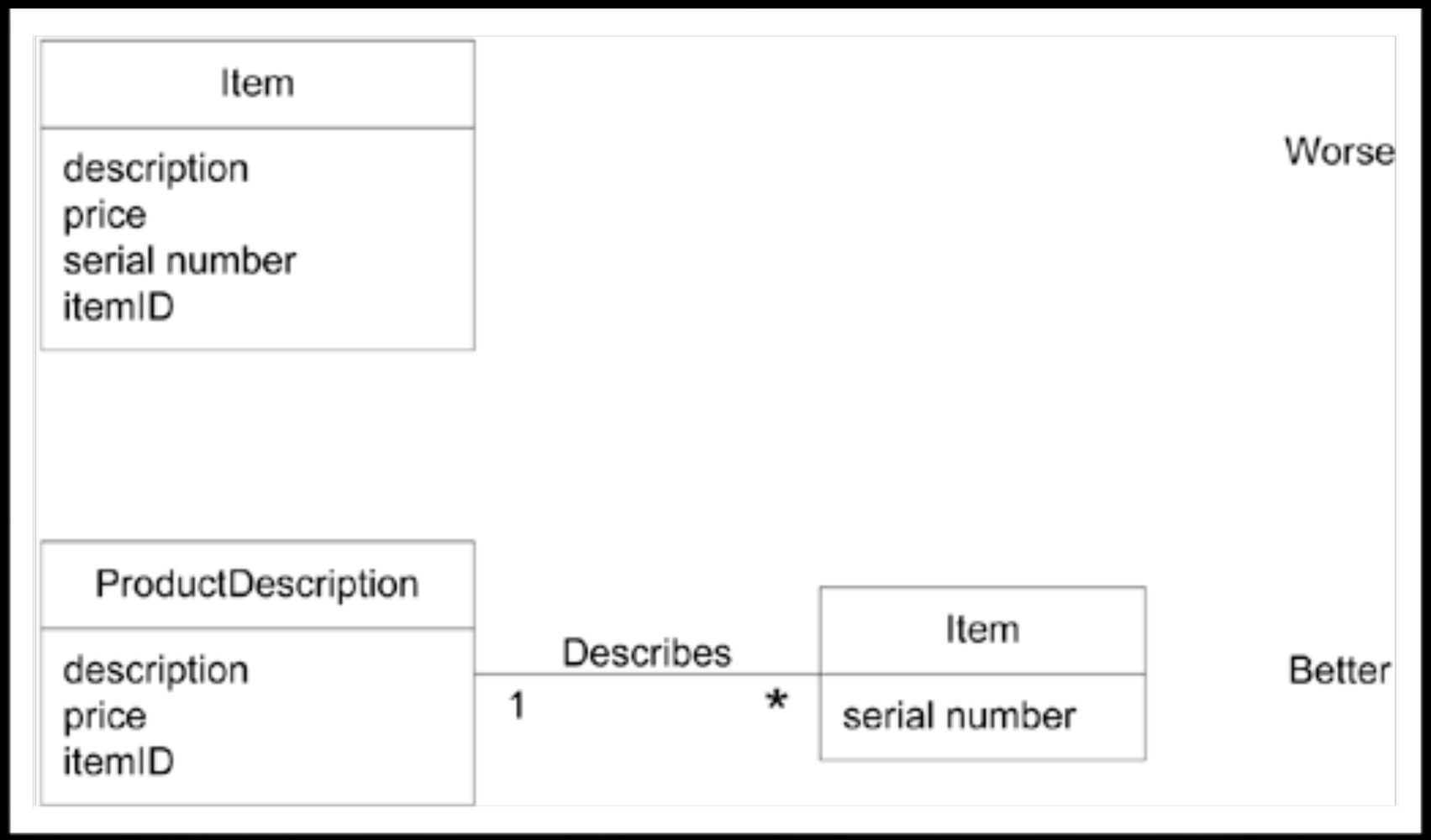
Common Associations

A is a member of B	*Cashier* — *Store* *Pilot* — *Airline*
A is an organizational subunit of B	*Department* — *Store* *Maintenance* — *Airline*
A uses or manages B	*Cashier* — *Register* *Pilot* — *Airplane*
A communicates with B	*Customer* — *Cashier* *Reservation Agent* — *Passenger*
A is related to a transaction B	*Customer* — *Payment* *Passenger* — *Ticket*
A is a transaction related to another transaction B	*Payment* — *Sale* *Reservation* — *Cancellation*
A is next to B	*SalesLineItem* — *SalesLineItem* *City*— *City*

Common Associations

Category	Examples
A is owned by B	*Register — Store* *Plane — Airline*
A is an event related to B	*Sale — Customer, Sale — Store* *Departure — Flight*

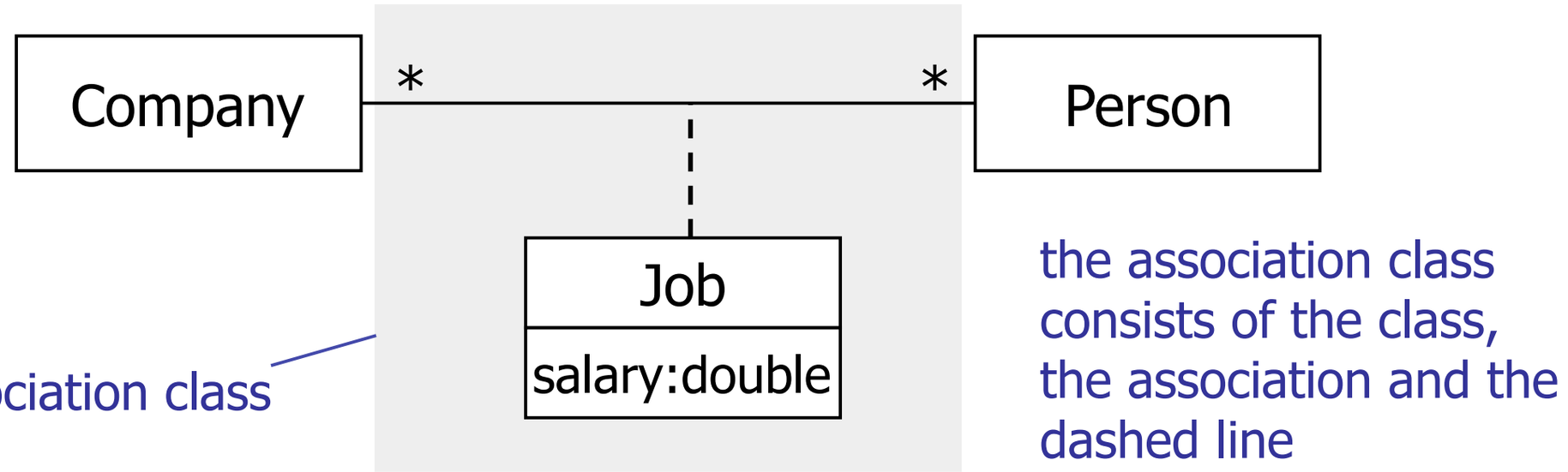
Description Classes



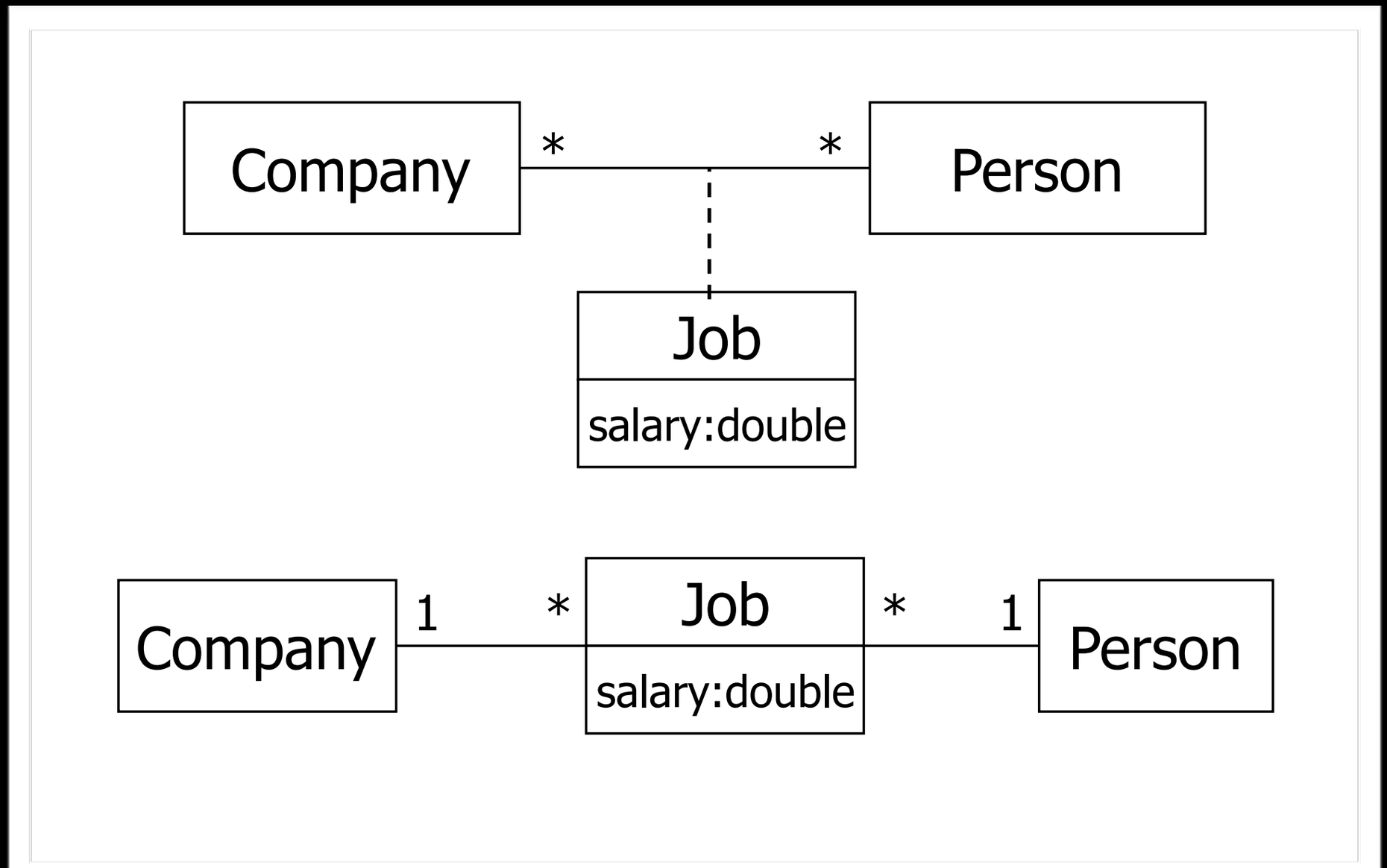
Association Classes

- A person can be employed many companies
- A company can employ many persons
- Every employee has a salary
- Where?

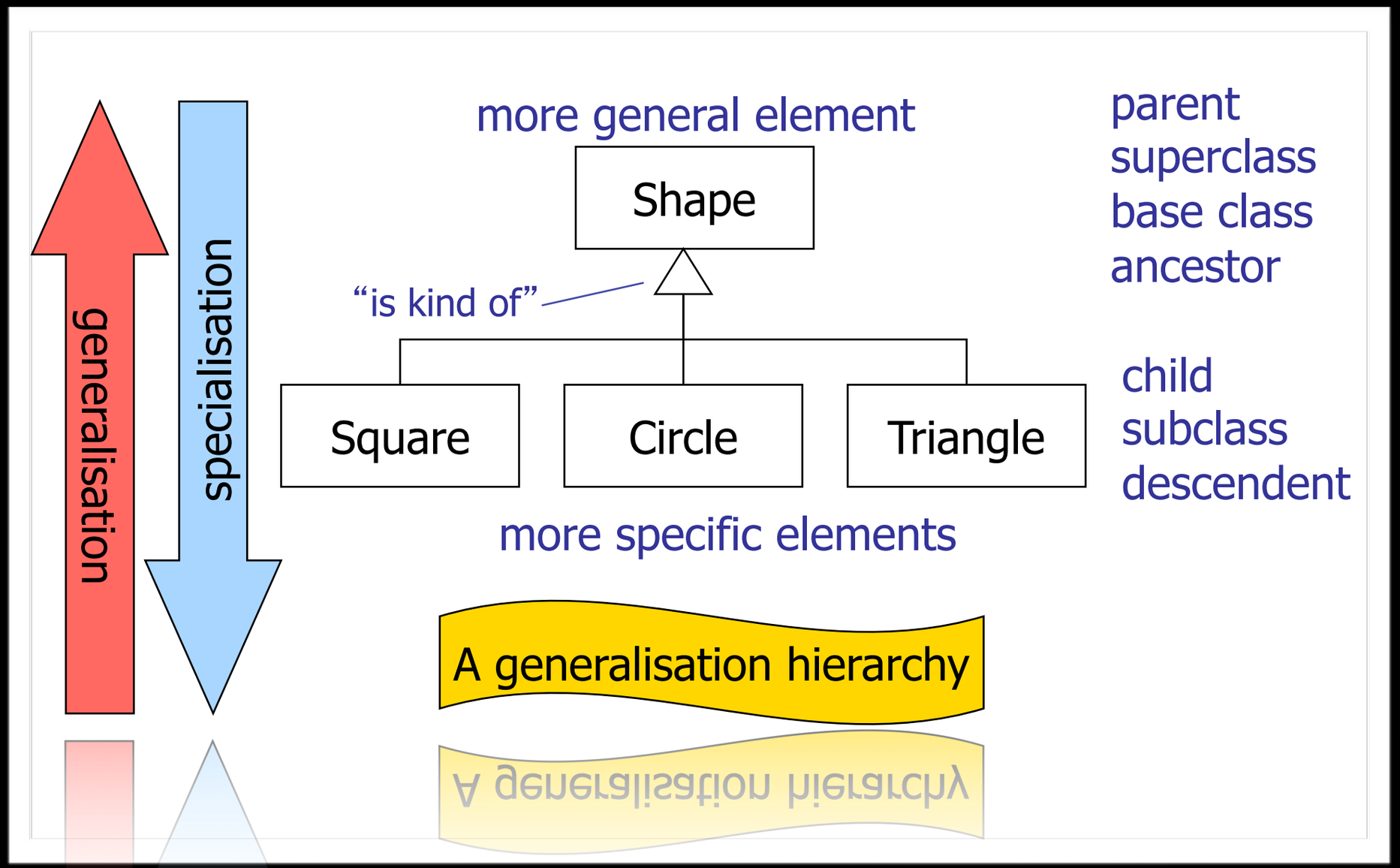
Association Classes



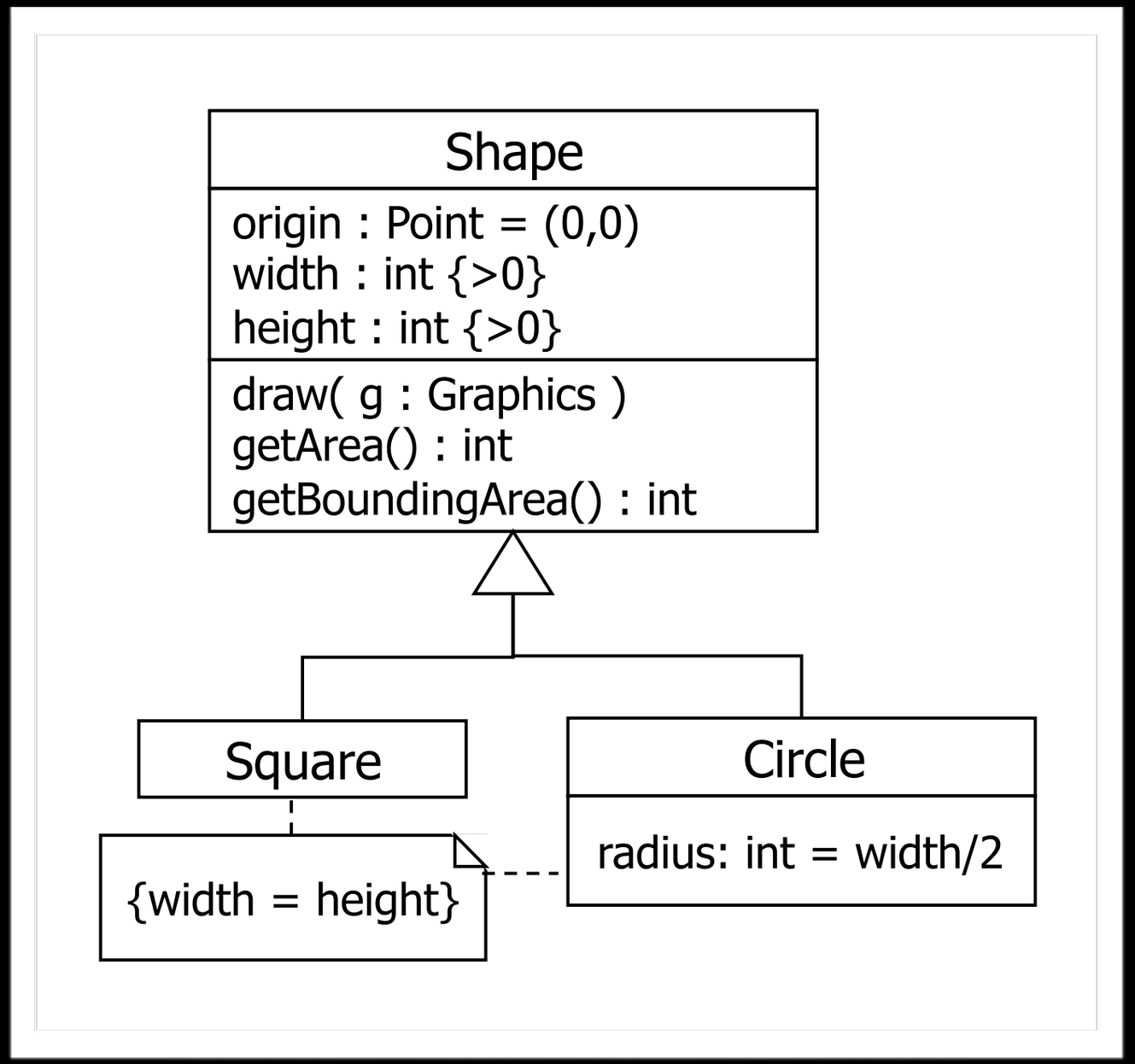
Association Classes



Generalization/ Specialization



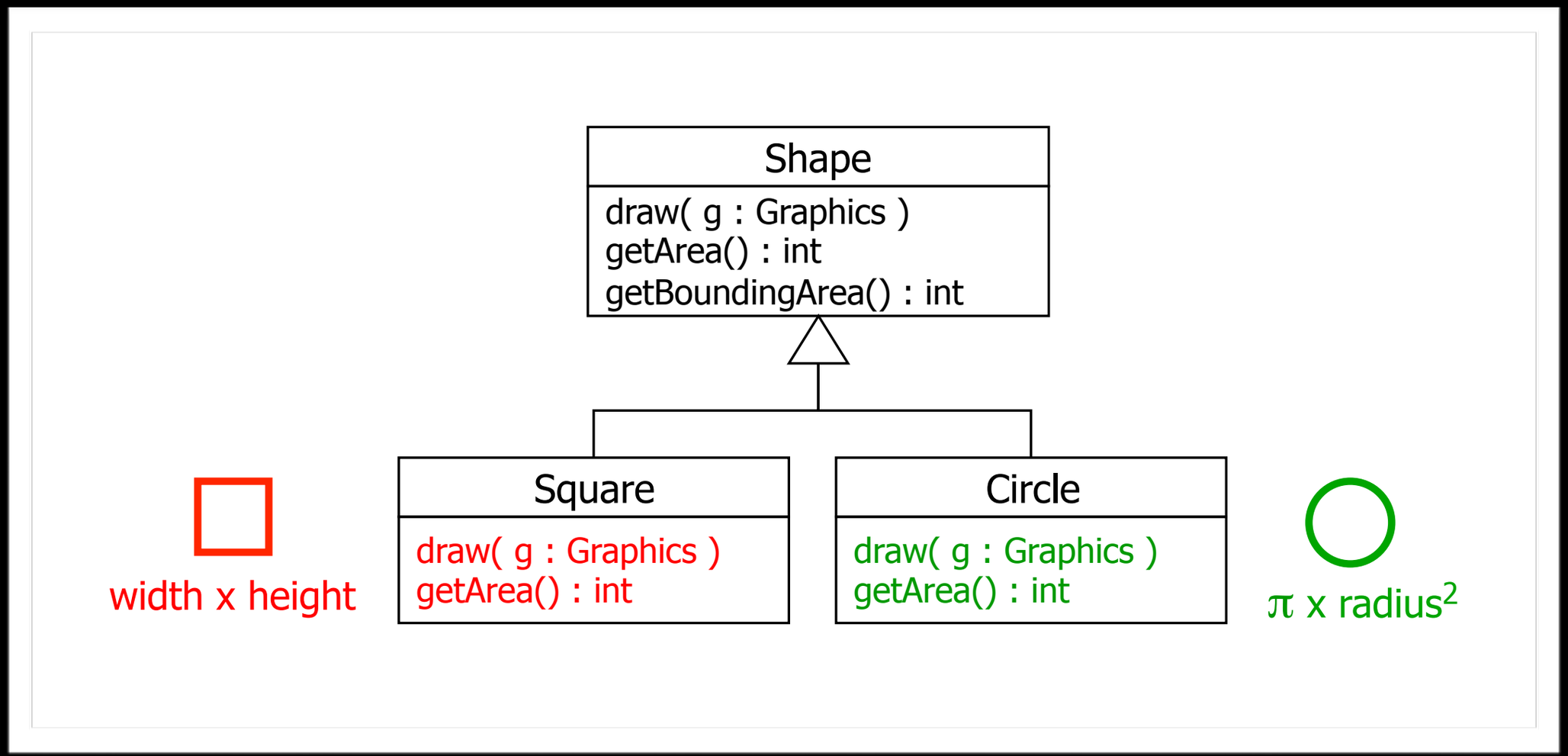
Inheritance



Inheritance

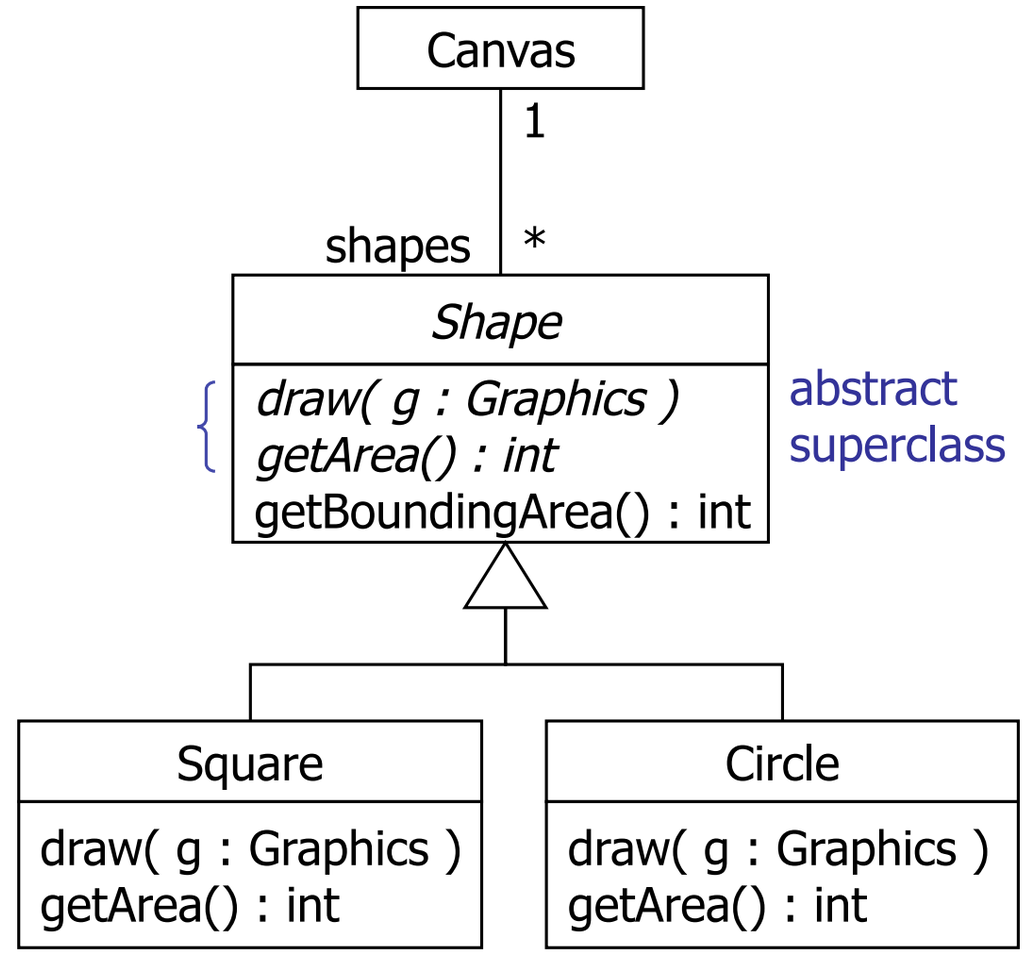
- A sub class inherits everything from a super class
  - even design elements
- Sub classes can be extended and changed
- A sub class can be used everywhere a super class is expected (LSP)

Change

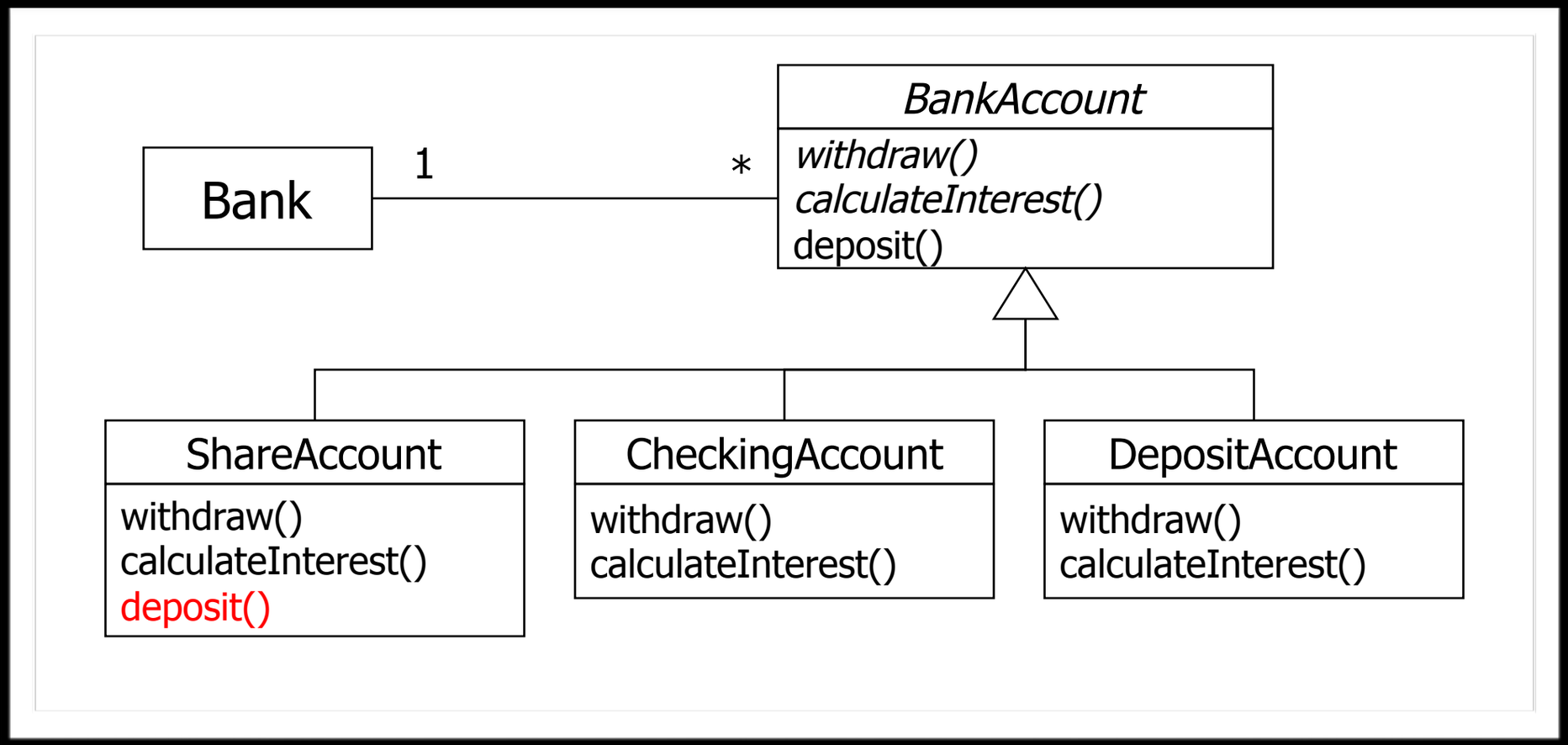

Shape
draw(g : Graphics)
getArea() : int
getBoundingArea() : int
Square
draw(g : Graphics)
getArea() : int
Circle
draw(g : Graphics)
getArea() : int
width x height
π x radius2

Abstract

A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle



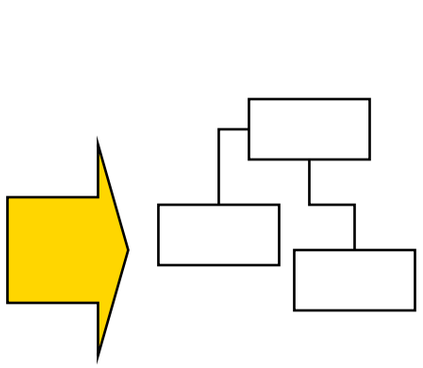
Example


BankAccount
withdraw()
calculateInterest()
deposit()
Bank
1
*
ShareAccount
withdraw()
calculateInterest()
deposit()
CheckingAccount
withdraw()
calculateInterest()
DepositAccount
withdraw()
calculateInterest()

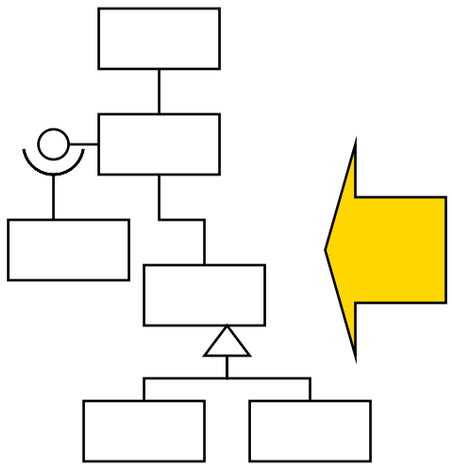
Now What?

Problem domain

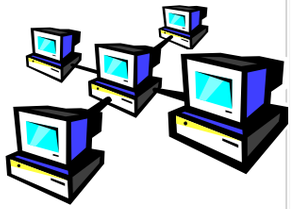
Domain model

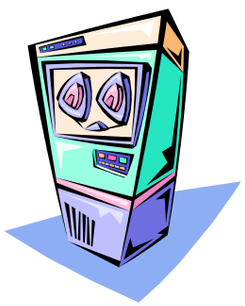


Design classes



Solution domain





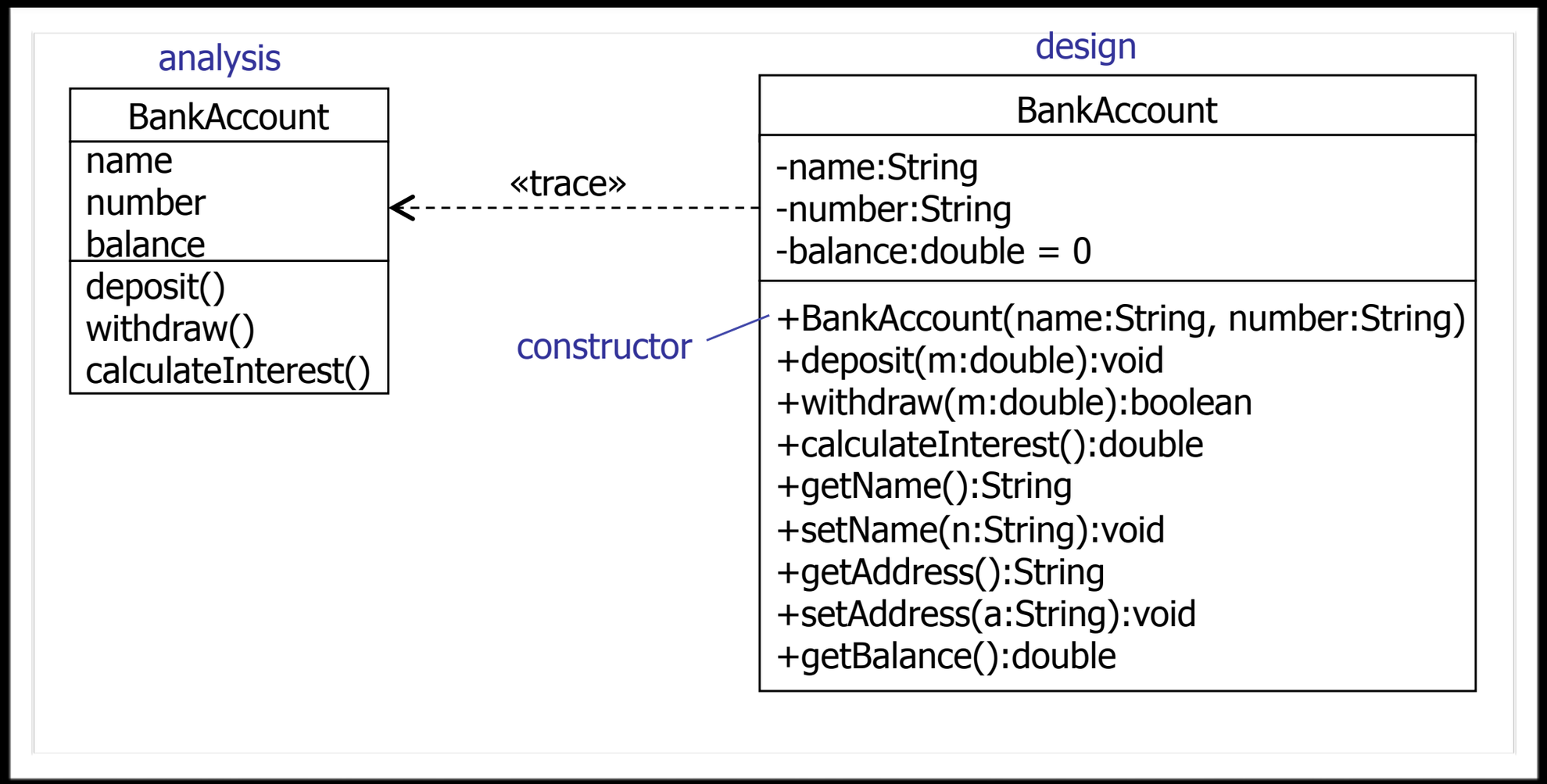
java.util

OO Design

"After identifying your requirements and creating a domain model, then add methods to the appropriate classes, and define the messaging between the objects to fulfill the requirements."

Larman

From Conceptual to Design

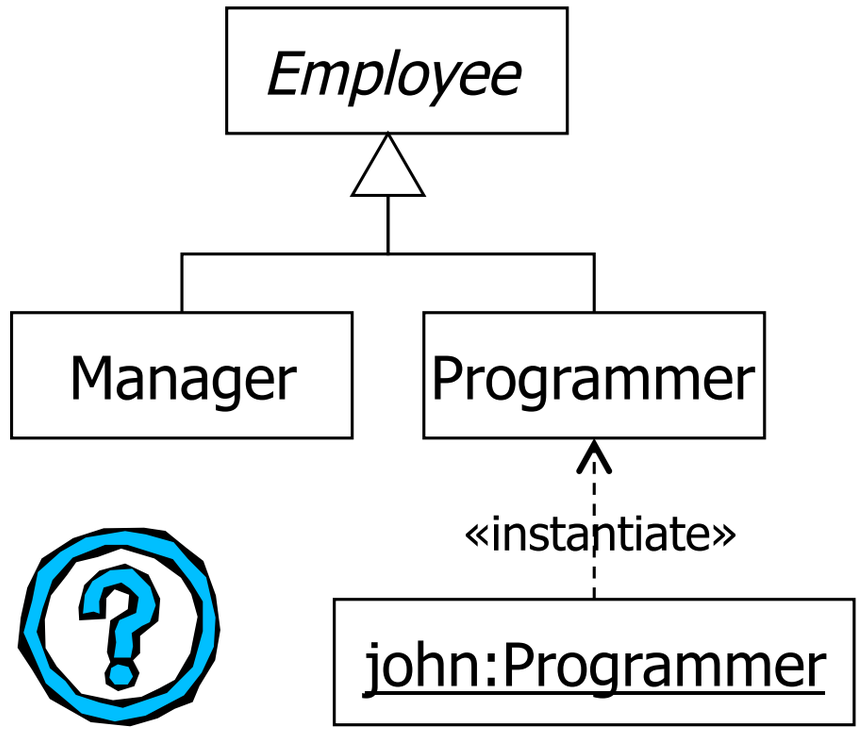


OO Design

"Ouch! Such vague advice doesn't help us, because deep principles and issues are involved. Deciding what methods belong where and how objects should interact carries consequences and should be undertaken seriously. Mastering OOD – and this is its intricate charm – involves a large set of soft principles, with many degrees of freedom. It isn't magic – the patterns can be named (important!), explained, and applied. Examples help. Practice helps..."

Larman

Inheritance, Composition, and Aggregation

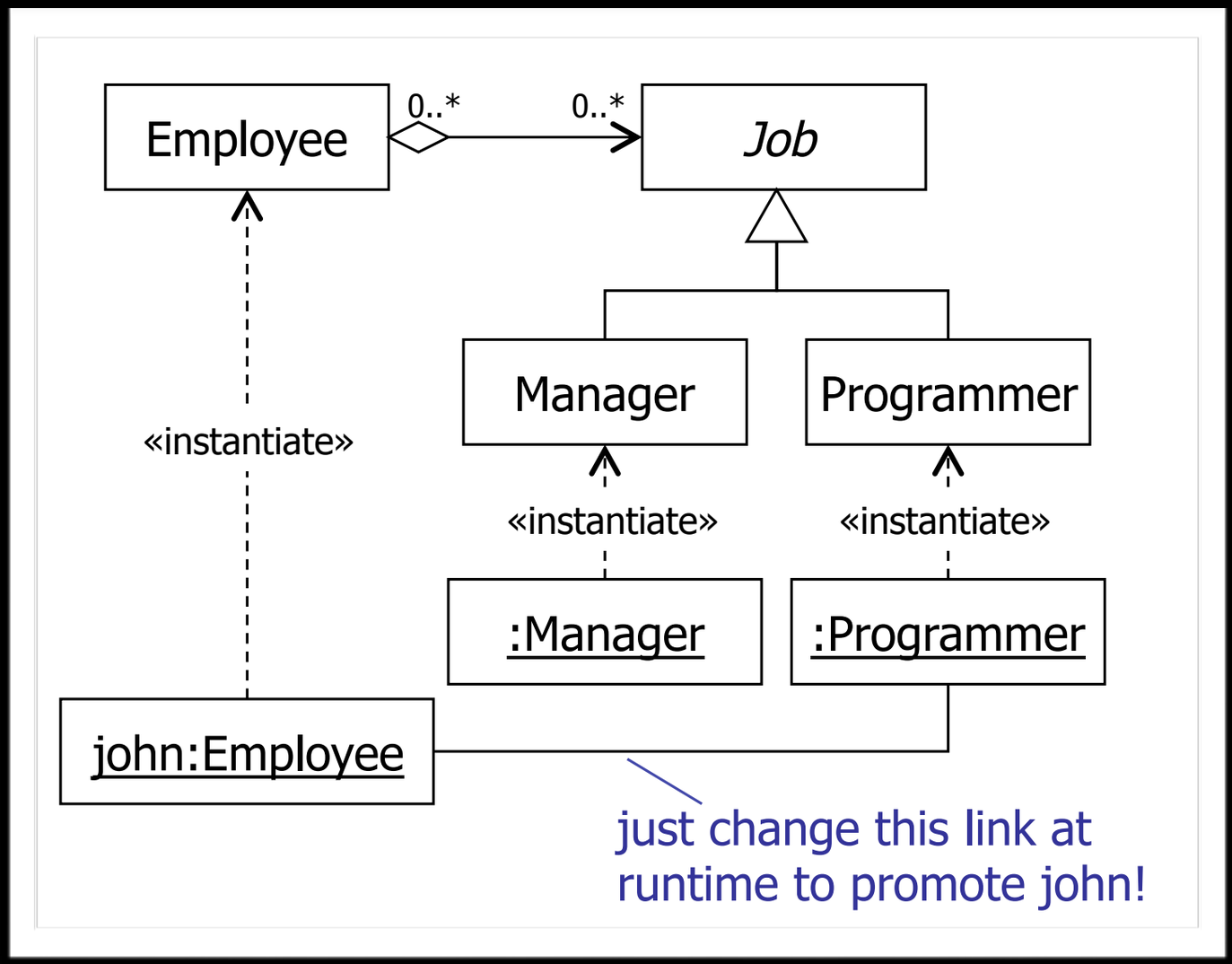


1. How can we promote john?
2. Can john have more than one job?

Inheritance, Composition, and Aggregation

- Inheritance is a predetermined, static association between instances of classes
  - and the class cannot be changed dynamically
- A semantic issue. Is an employee the job or does he or she have a job?

A Better Solution



OO Design

"Ouch! Such vague advice doesn't help us, because deep principles and issues are involved. Deciding what methods belong where and how objects should interact carries consequences and should be undertaken seriously. Mastering OOD – and this is its intricate charm – involves a large set of soft principles, with many degrees of freedom. It isn't magic – the patterns can be named (important!), explained, and applied. Examples help. Practice helps..."

Larman

Goals

- Maximize abstraction
  - hide difference between data and behavior
  - think in terms of responsibility to know, do, and decide
- Distribute behavior
  - smart objects, not data containers
  - find the right size / granularity

Goals

- Maintain flexibility
  - design objects and collaborations so that they can easily be changed and/or extended

Responsibility-Driven Design

- Two pillars
  - objects have responsibility
  - objects collaborate
- Responsibility – abstractions on different levels

Responsibility-Driven Design

- Responsibilities
  - doing
  - knowing
  - controlling

Doing and Controlling

- Do something
  - create new objects
  - compute
- Initiate actions in other objects
- Control and coordinate activities in other objects

Knowing

- Know something about private data and variables
- Know something about other, connected objects
- Know something about things that can be computed or deduced

Guideline

- Interactions helps to find doing and controlling
- Domain models help to find knowing

Concepts

- An application is a collection of collaborating objects
- An object is an implementation of one or more roles
- Roles are connected responsibilities
- A responsibility is a requirement to carry out a specific task or know a specific thing
- Collaboration happens between objects and/or roles

Process

1. Find classes/objects in the system
2. Determine the responsibility of each class/object
3. Decide how the classes/objects collaborate to fulfill their responsibility

GRASP

- General Responsibility Assignment Software Patterns
- Describes basic principles for object and responsibility design
  - as a collection of nine patterns

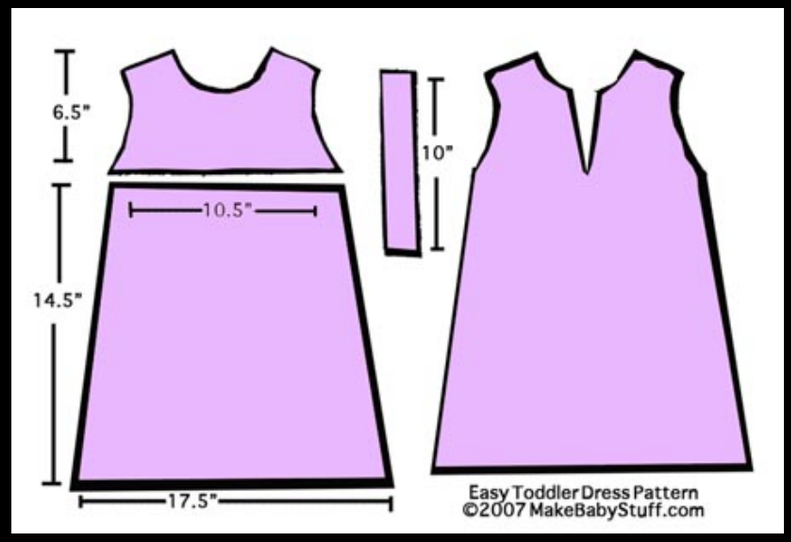
Patterns

"A pattern describes a problem which occurs over and over again in our environment, and then 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 the same way twice."

Christopher Alexander

Patterns

"I like to relate this definition to dress patterns... I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself."



Jim Coplein

Patterns and Engineering

- How do "other" engineers use and find patterns?
  - car manufacturer do not start from the laws of physics when they design a car
    - they have manuals that describe good solutions to known problems
    - and apply standard solutions that are known to work and learn from experience
- So, patterns should be important to software engineering

Patterns

- Alexander: *"A pattern is a recurring solution to a standard problem, in a context."*
- Larman: *"In OO design, a pattern is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs."*

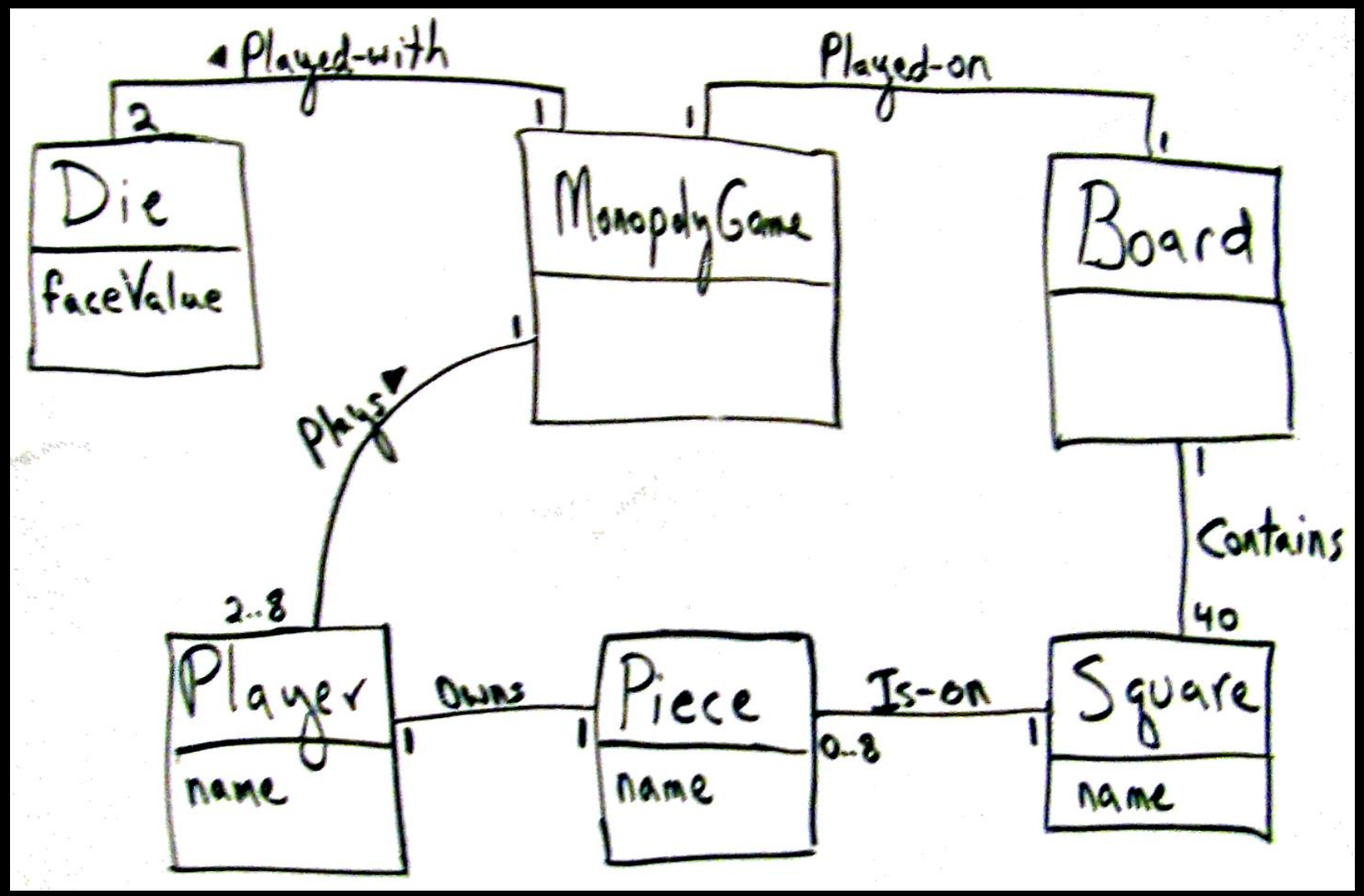
Naming is important

- Patterns have suggestive/expressive names
  - Arched Columns Pattern, Easy Toddler Dress Pattern, etc.
- Why is the name important?
  - easier to remember
  - easier to communicate
- ASP.NET MVC

GRASP

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Law of Demeter (Don’t talk to strangers)

Example

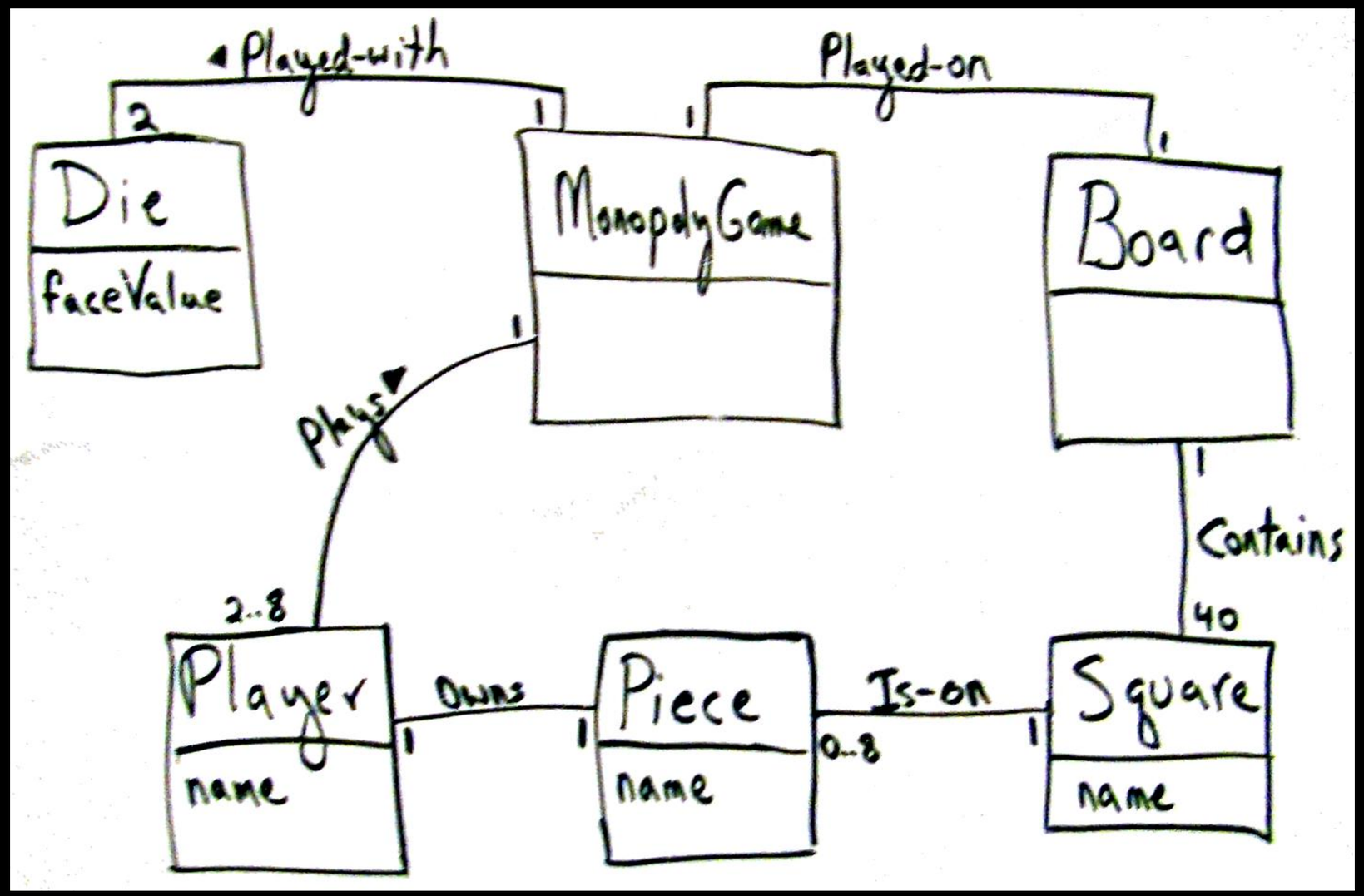


Where do *Squares* come from?

Creator

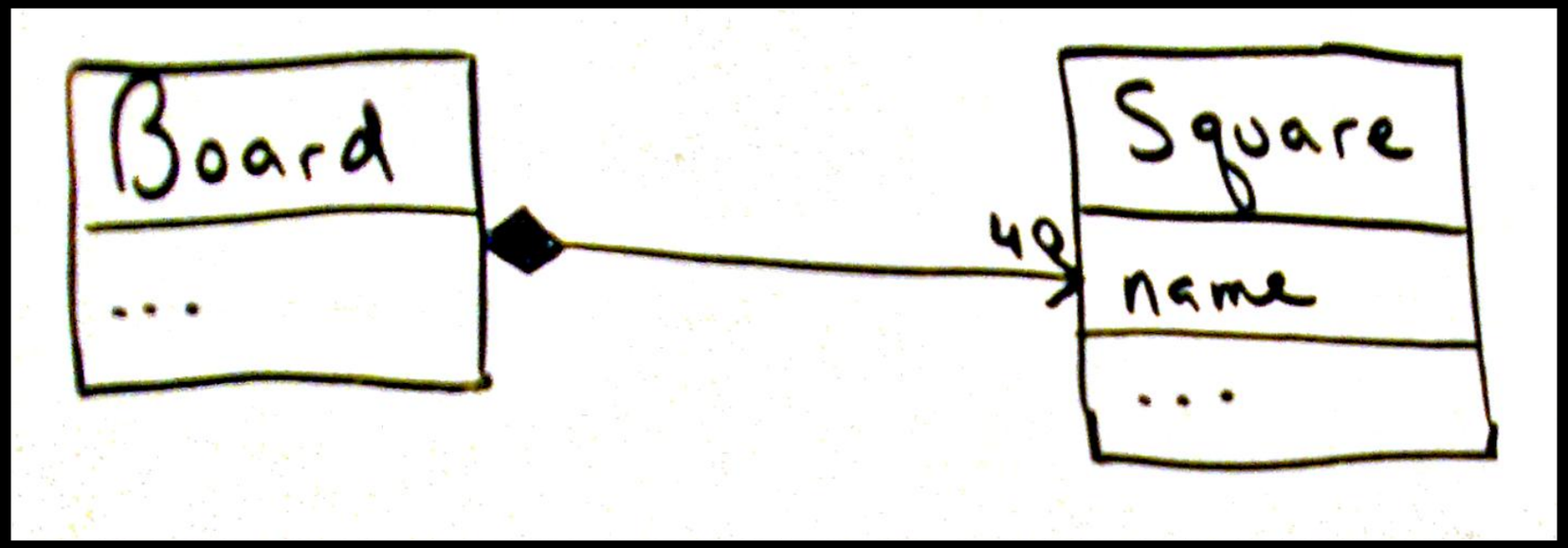
- Pattern to determine: who creates instances of class A
- B is responsible for it if:
  - B contains or aggregates A
  - B saves A
  - B uses A (close)
  - B has data to initialise A

Example



Who creates *Squares*?

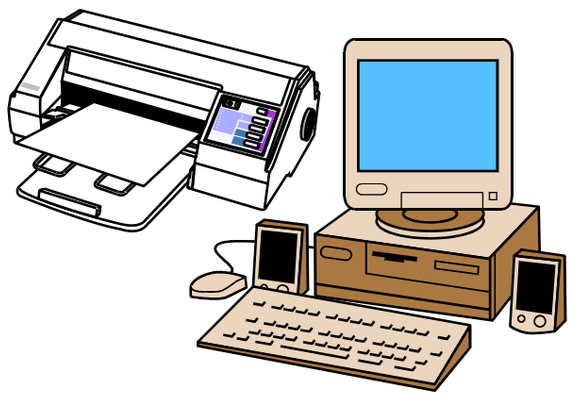
Example



- We need to translate associations to
  - aggregations, or
  - compositions

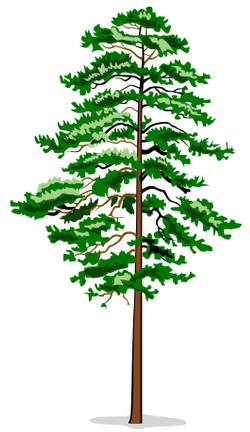
Difference

Aggregation



Certain objects have loose connections, e.g. computer and printer

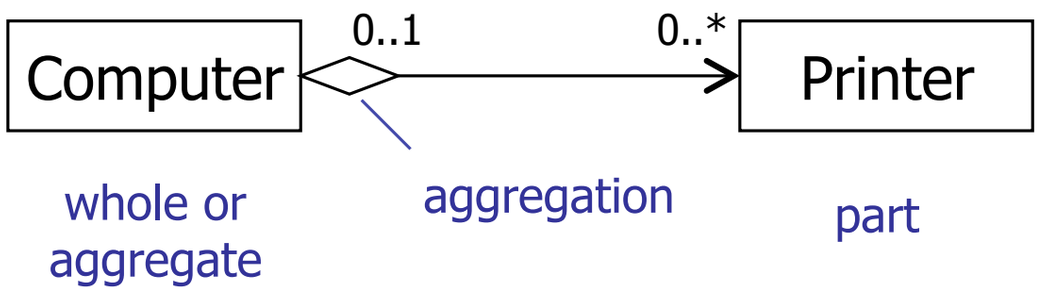
Composition



Other objects have strong connections, e.g. a tree and its leaves.

Aggregation

aggregation is a *whole–part* relationship



A Computer may be attached to 0 or more Printers

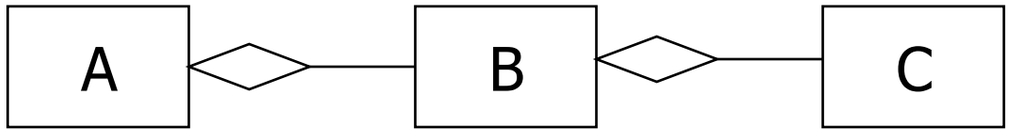
At any one point in time a Printer is connected to 0 or 1 Computer

Over time, many Computers may use a given Printer

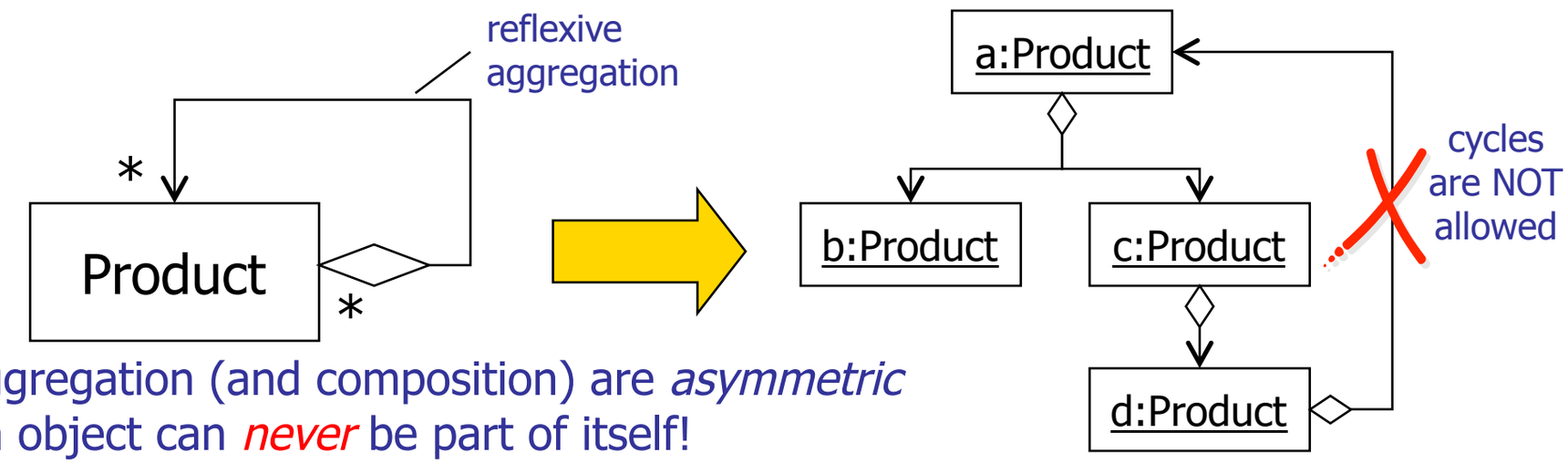
The Printer exists even if there are no Computers

The Printer is independent of the Computer

Aggregation



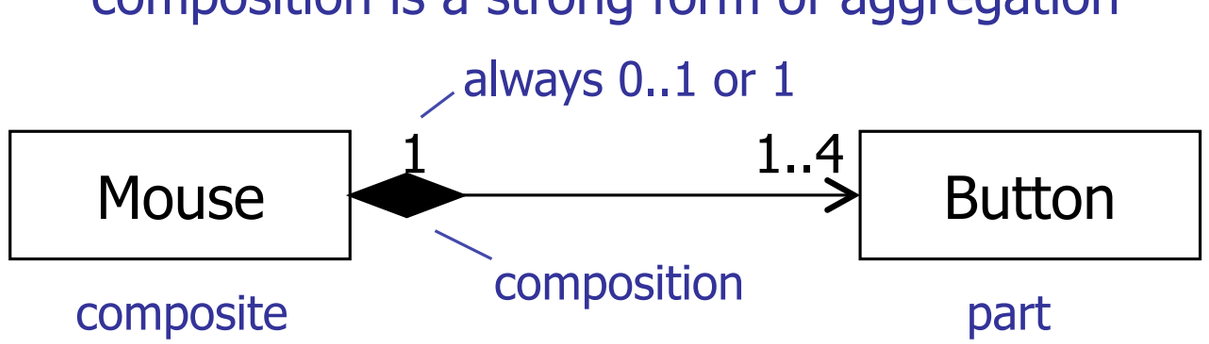
Aggregation (and composition) are *transitive*
If C is a part of B and B is a part of A, then C is a part of A



Aggregation (and composition) are *asymmetric*
An object can *never* be part of itself!

Composition

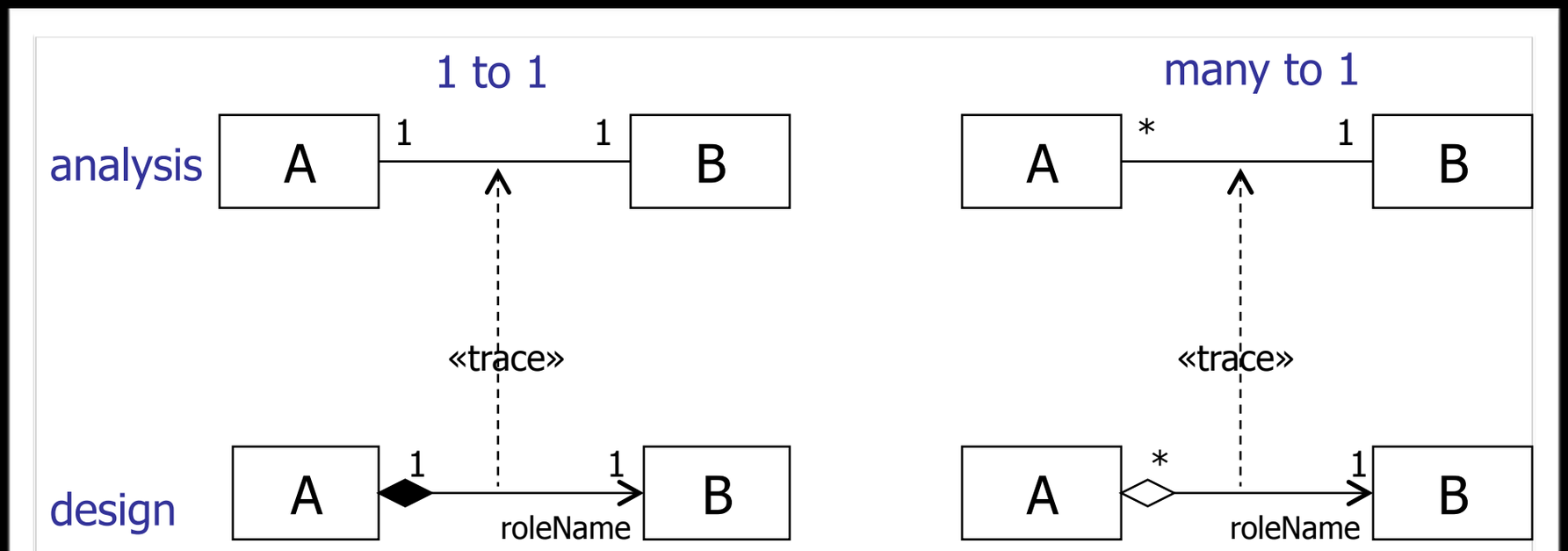
composition is a strong form of aggregation



The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

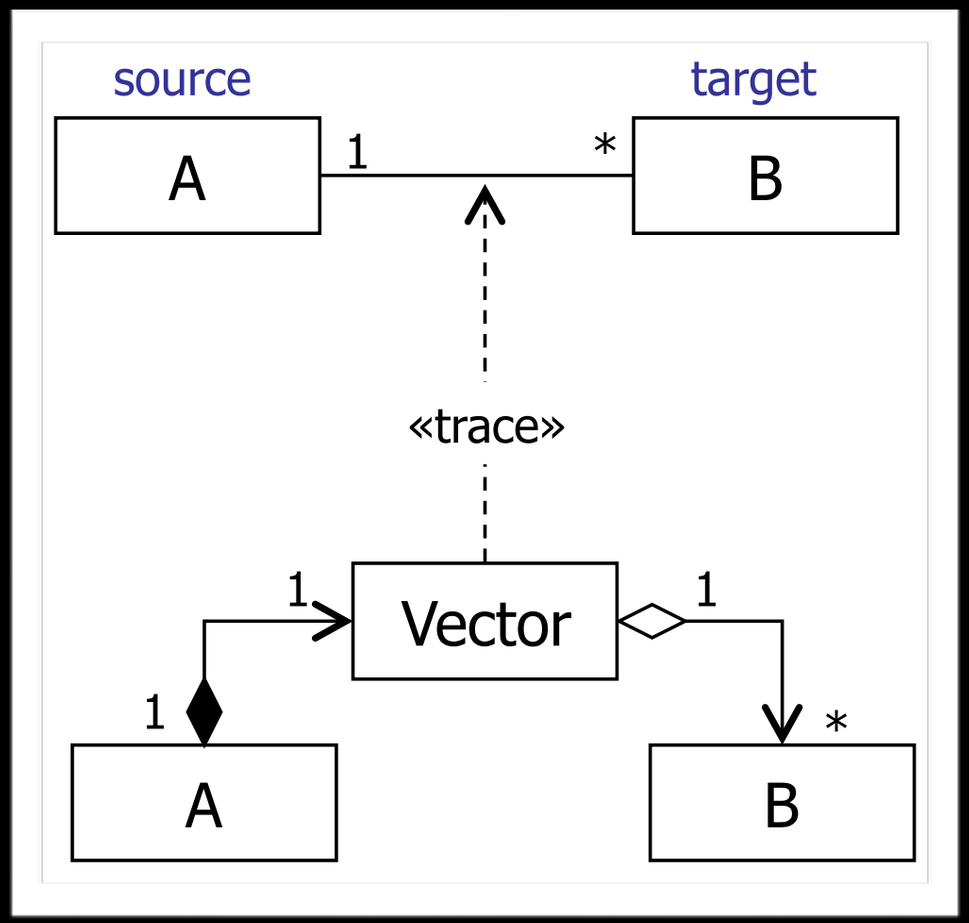
Each button can belong to exactly 1 mouse

Guidelines

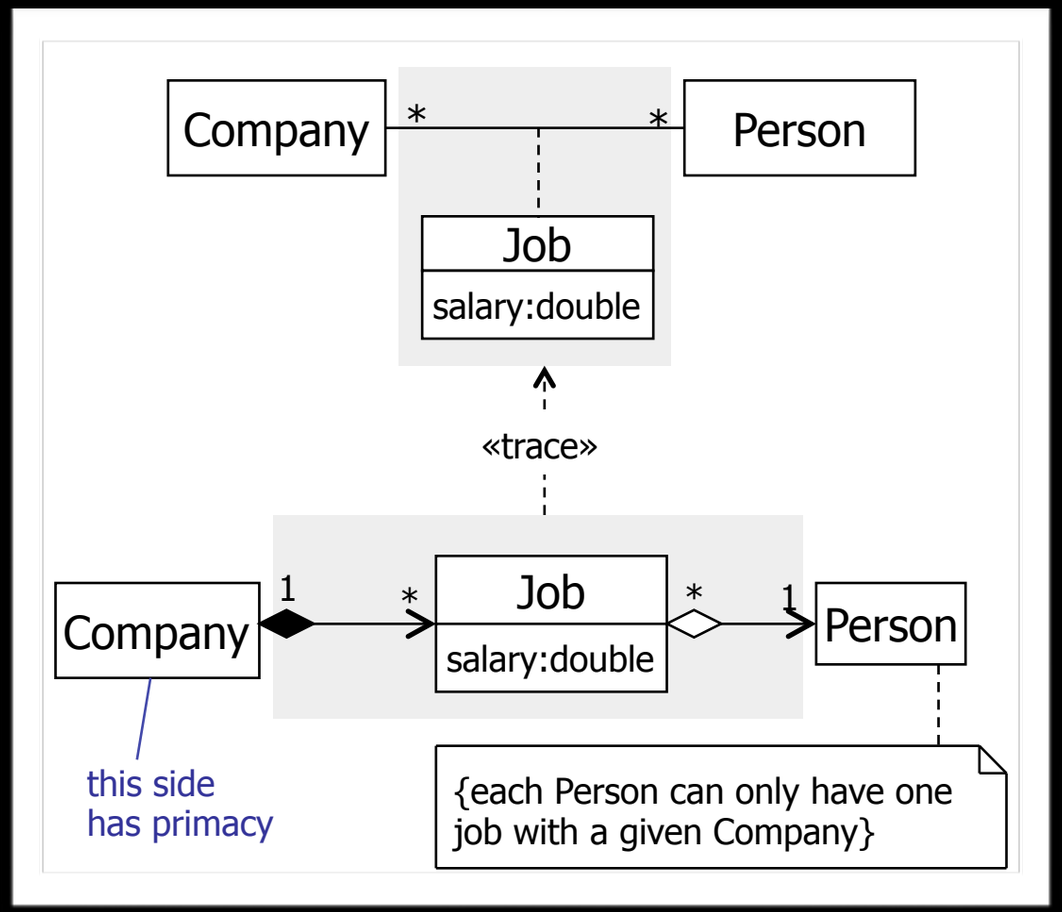


- 1:1 associations are often compositions
- N:1 associations are often aggregations

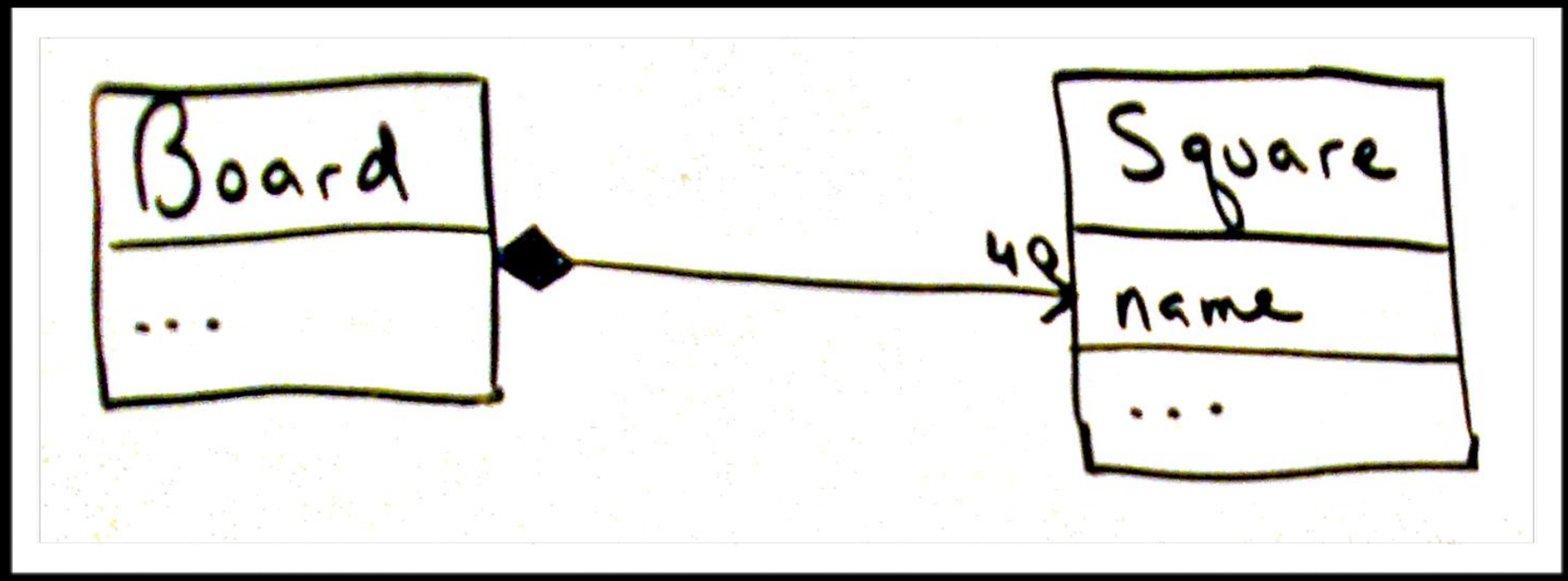
Guidelines



Guidelines

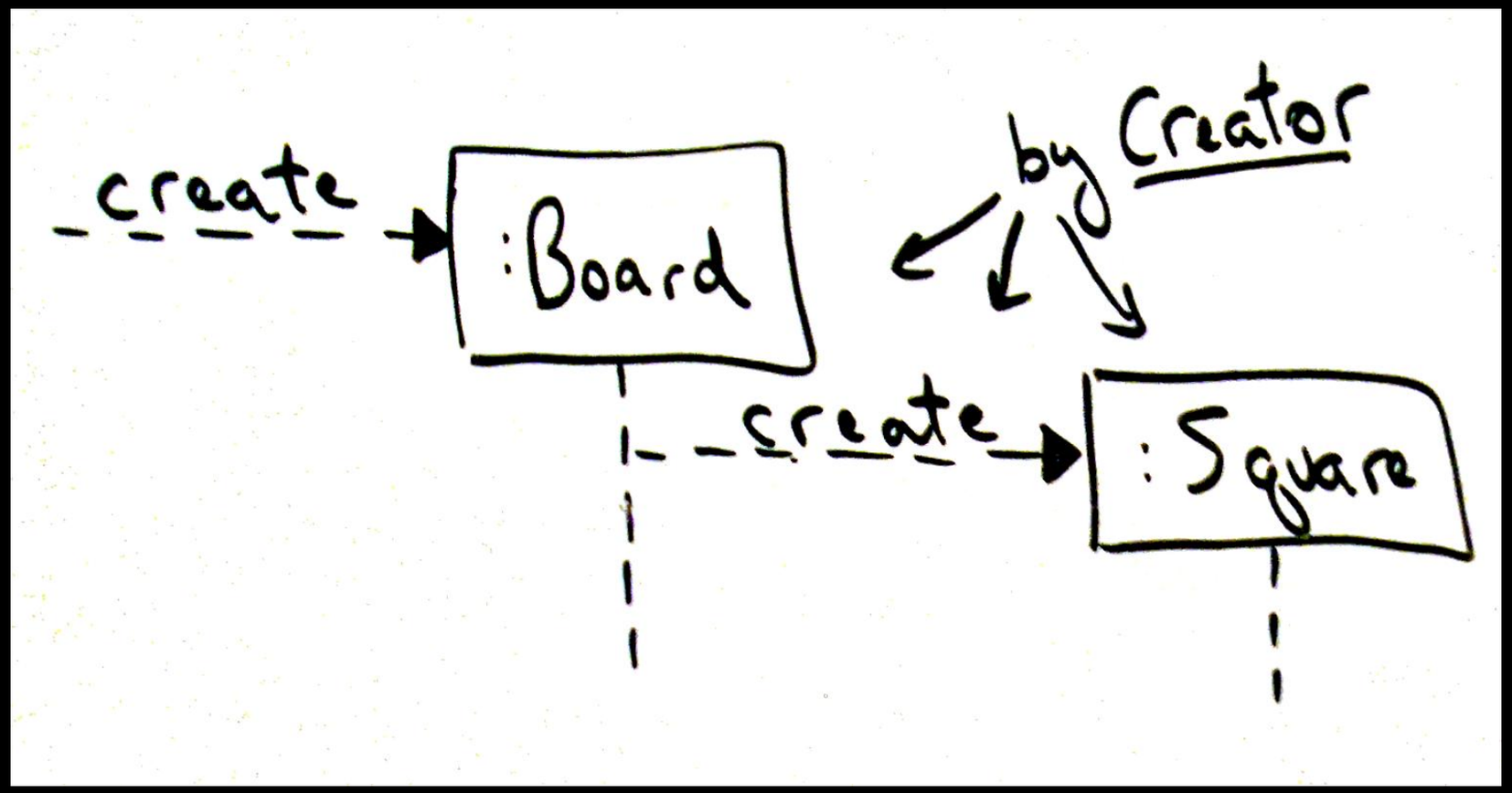

Company
*
*
Person
Job
salary:double
«trace»
Company
1
*
Job
salary:double
*
1
Person
this side
has primacy
{each Person can only have one
job with a given Company}

Example



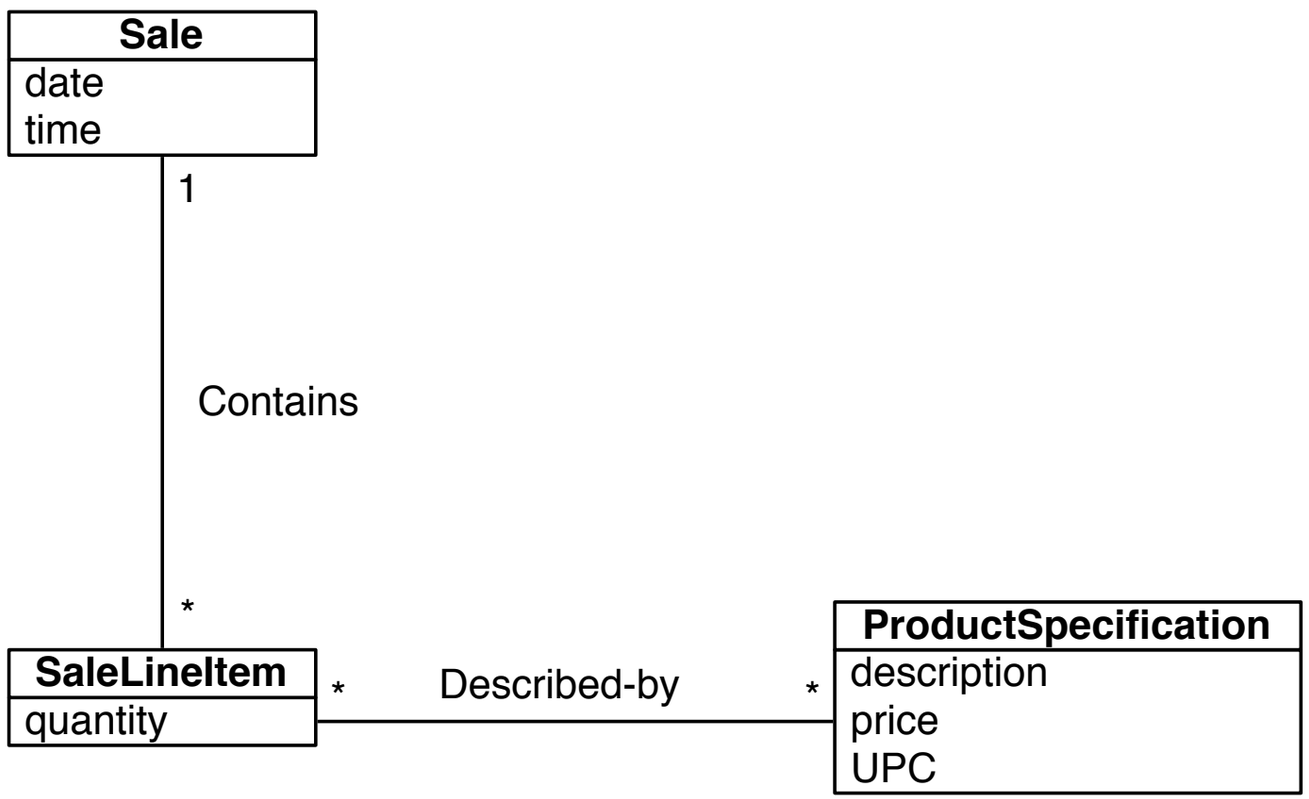
Composition, a Board contains a number (40) of Squares

Example



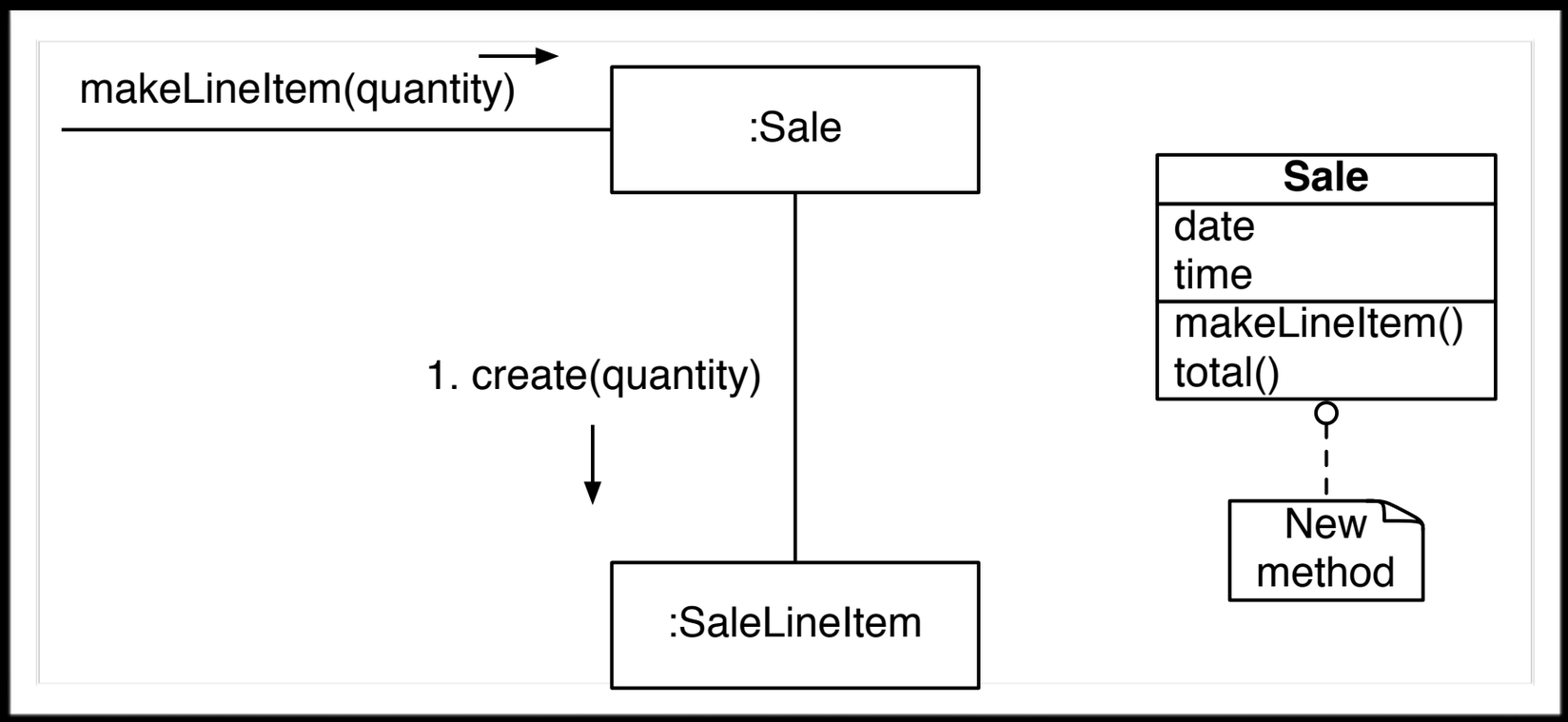
"B contains or aggregates A"
(A is Square, B is Board)

Example



Who creates *SalesLineItems*?

Example

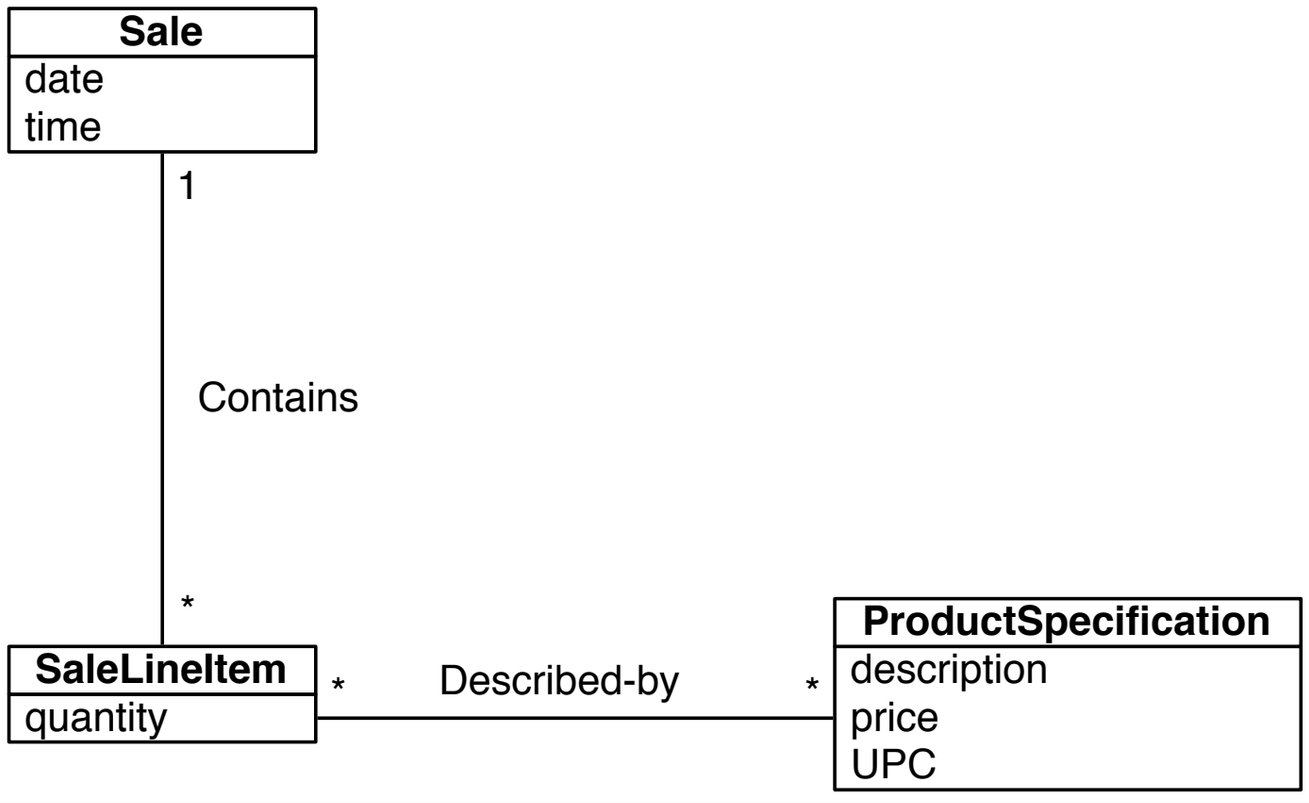


Creator suggests *Sale* since there is a composition relation between *Sale* and *SalesLineItems*

Information Expert

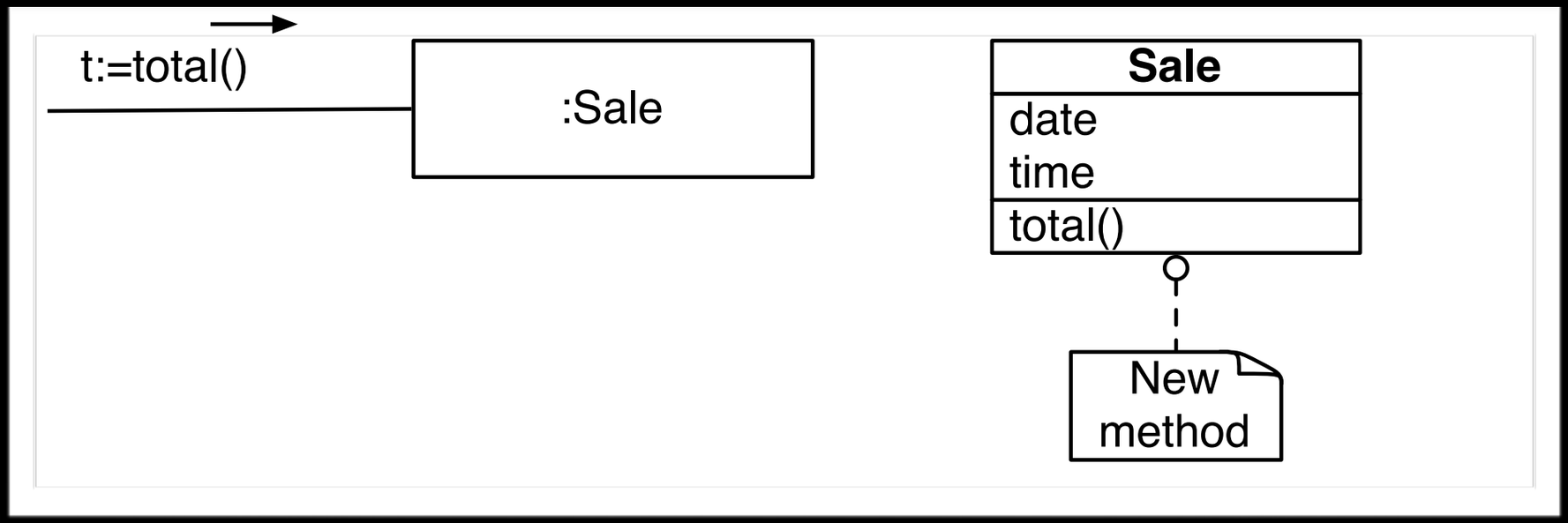
- How should responsibility be distributed among objects?
- The class/object with enough information should be responsible!

Example

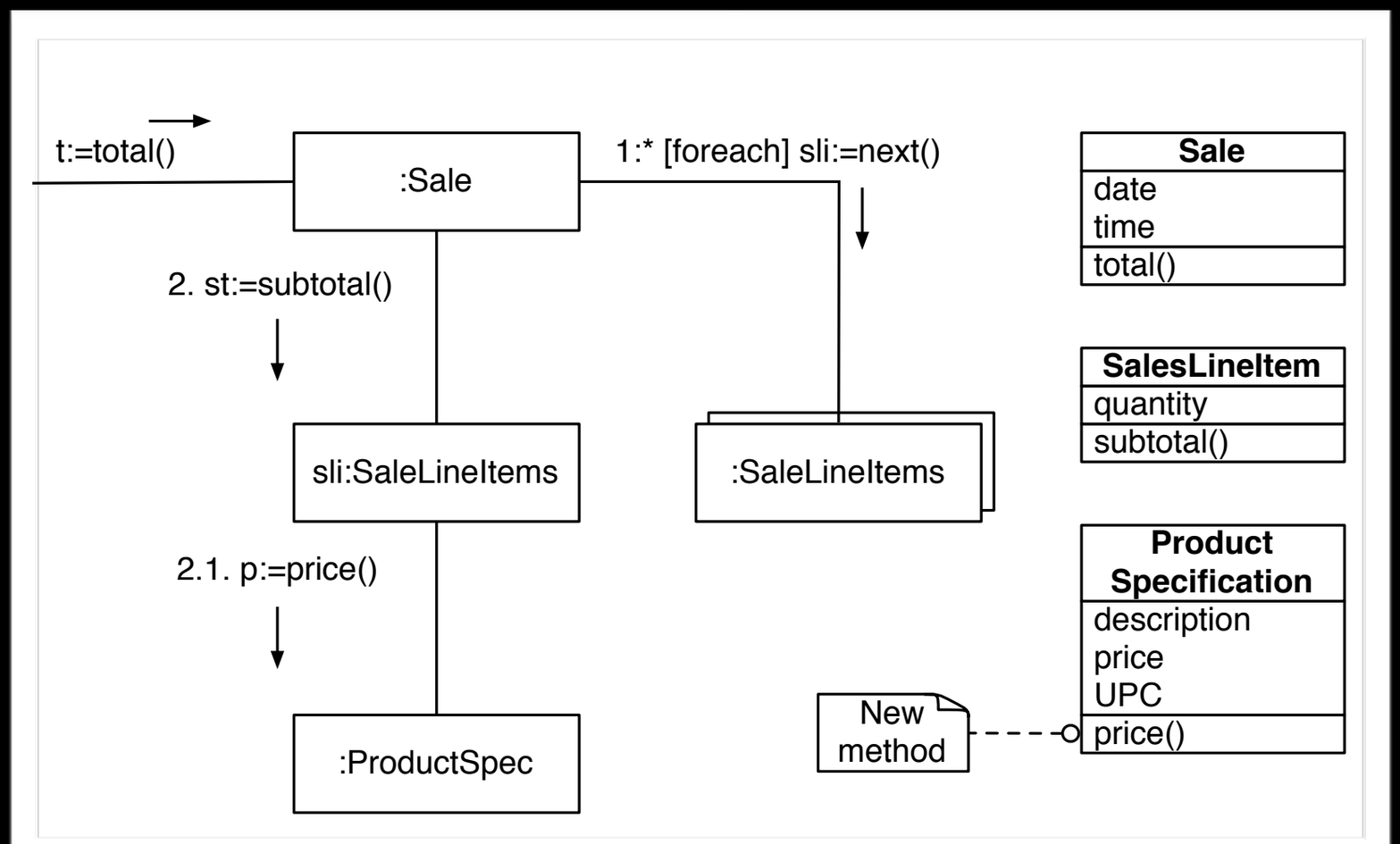

Sale
date
time
1
Contains
*
SaleLineItem
quantity
*
Described-by
*
ProductSpecification
description
price
UPC

Example

- Needs all instances of *SalesLineItem* and their sums
- *Sale* is Information Expert for the total sum



Example



Sum is required for each *SalesLineItem*. Computed from price and quantity, so salesLineItem is IE

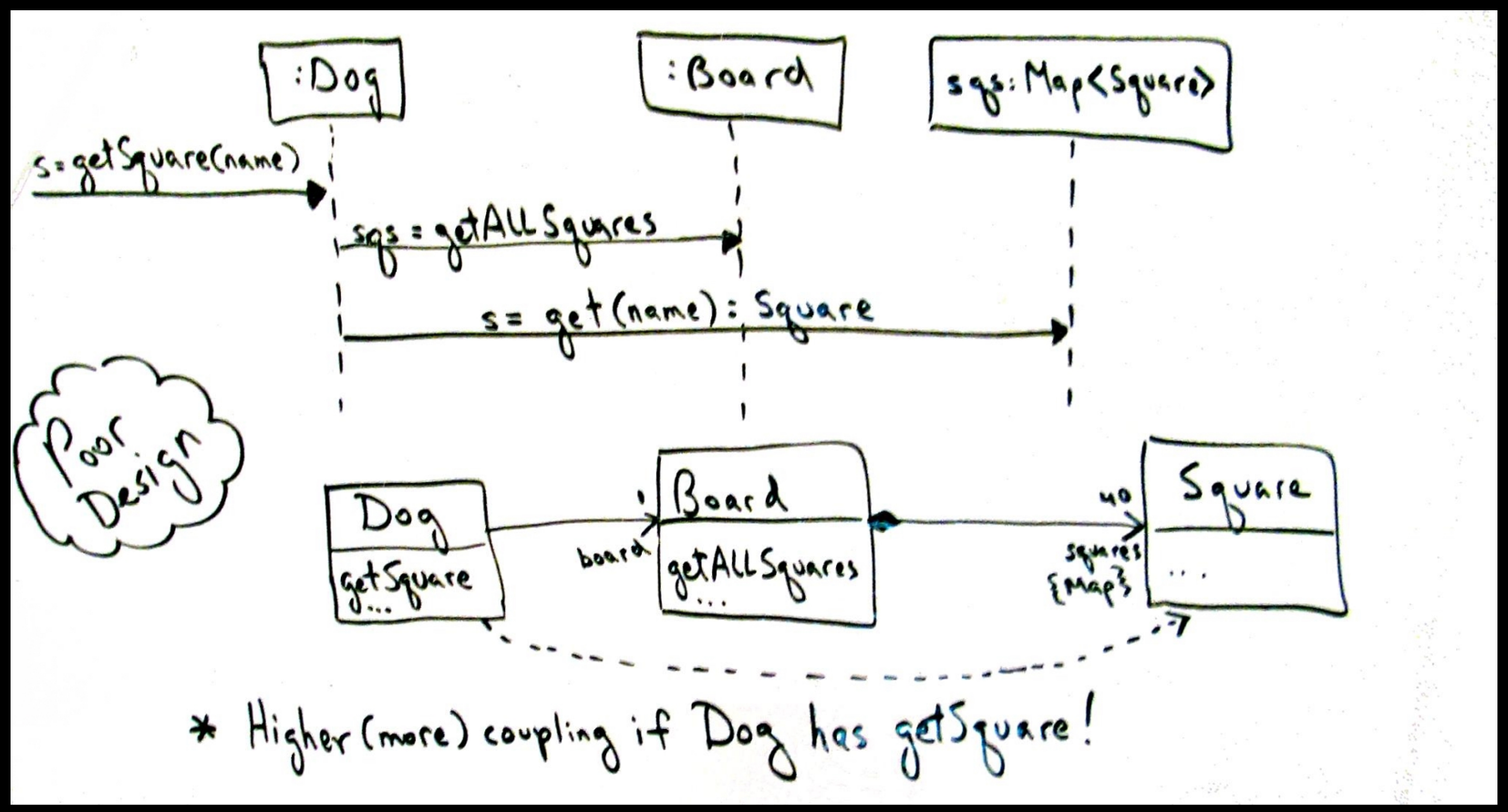
Example

Class	Responsibility
Sales	Total sum
SalesLineItem	Sum
ProductSpecification	Price

Low Coupling

- How can we minimise the effects of change and support reuse?
- Distribute responsibility to minimise dependencies between classes

Example

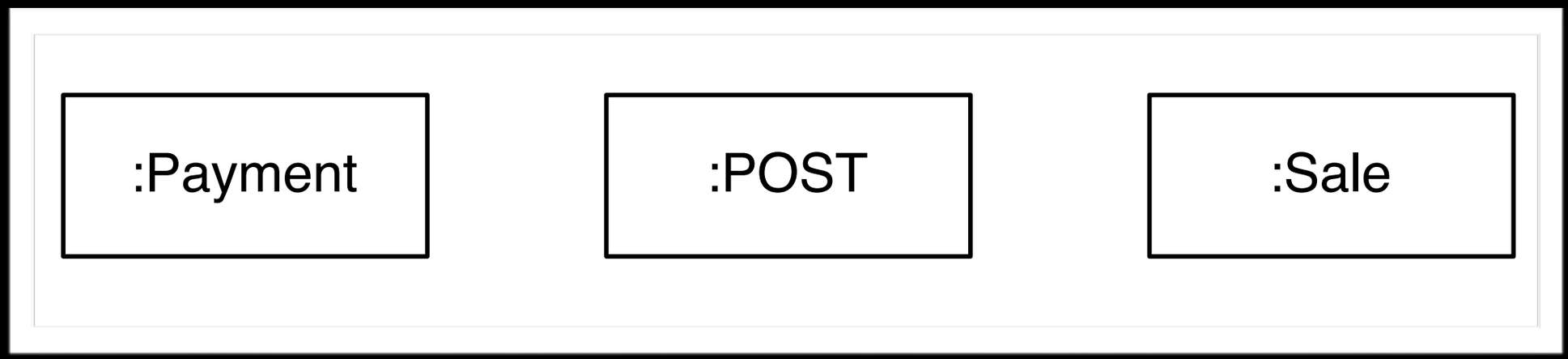


Many dependencies – Poor design

Common Dependencies

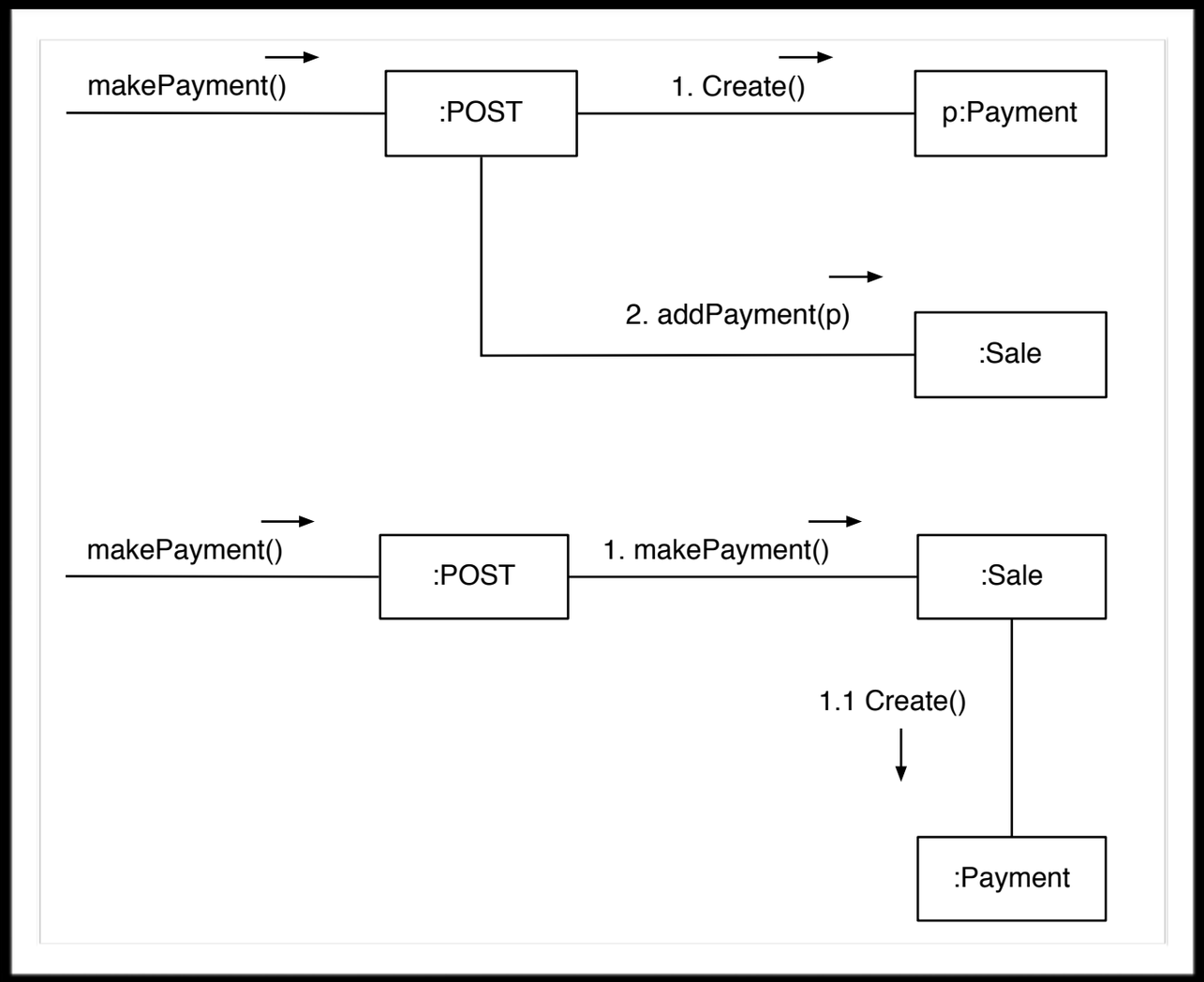
- *TypeX* contains an attribute of *TypeY*
- *TypeX* has a method that uses *TypeY*
- *TypeX* is a direct or indirect sub class of *TypeY*

Example



Who creates a *Payment*?

Example

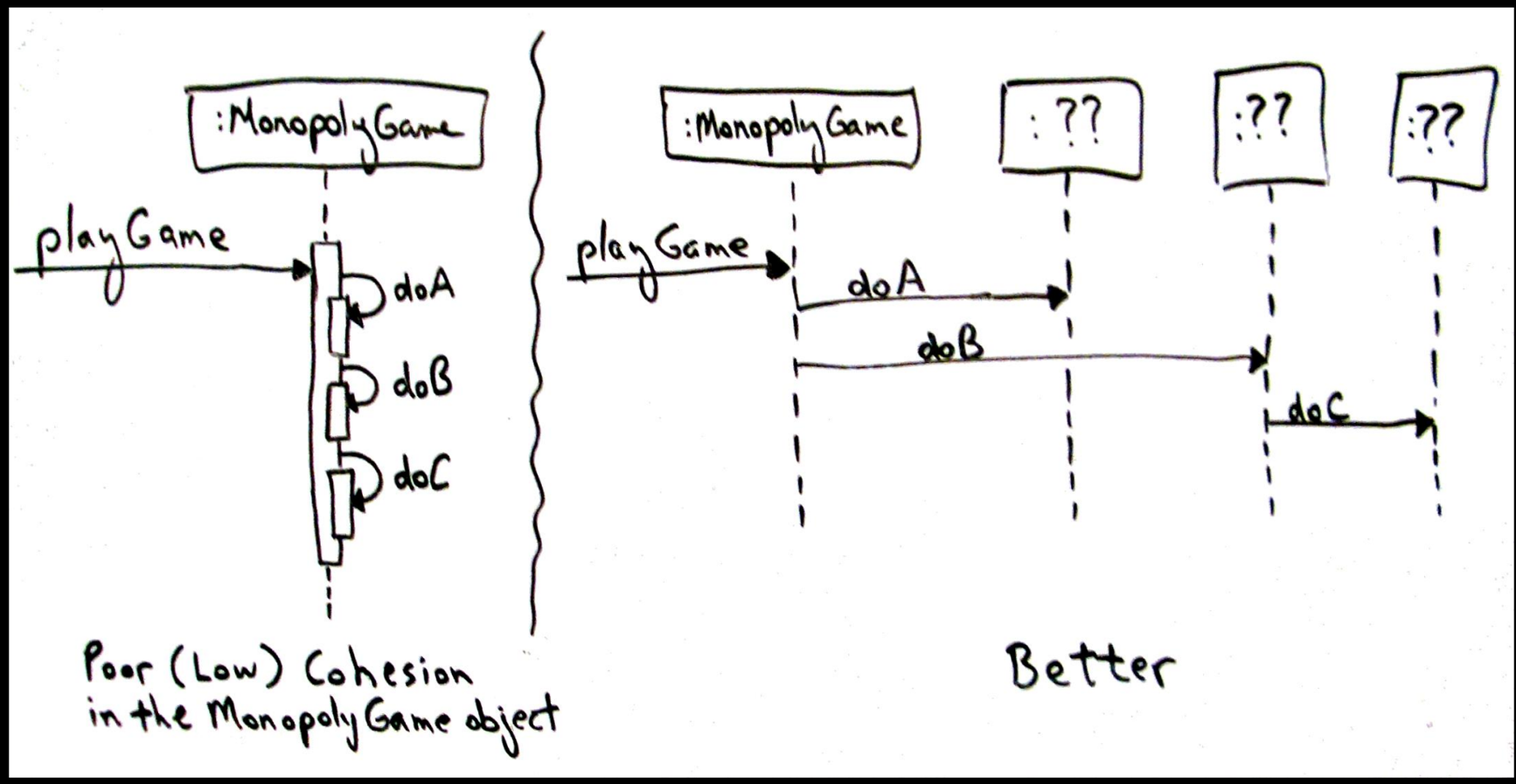


Two possibilities: *POST* or *Sale*. *Sale* is a better choice since it already has a dependency to *Payment*.

High Cohesion

- How can complexity be reduced/ managed?
- Distribute responsibility to create cohesive classes/objects
  - i.e., focused

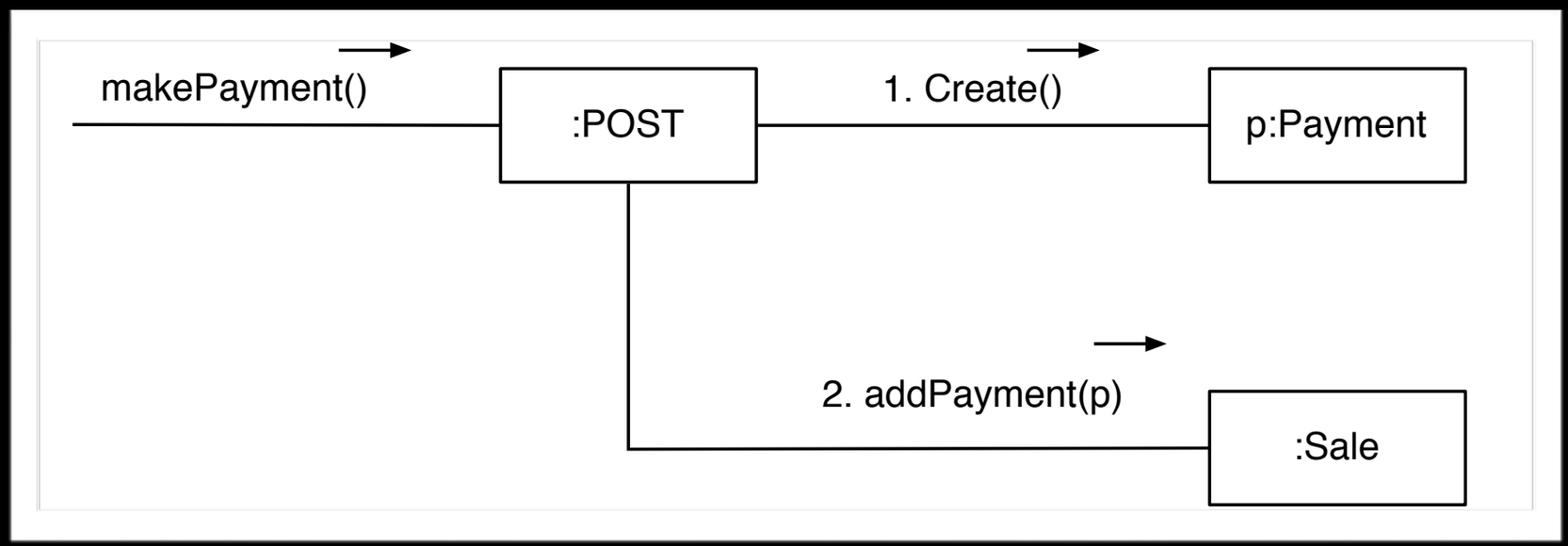
Example



Low vs. high cohesion

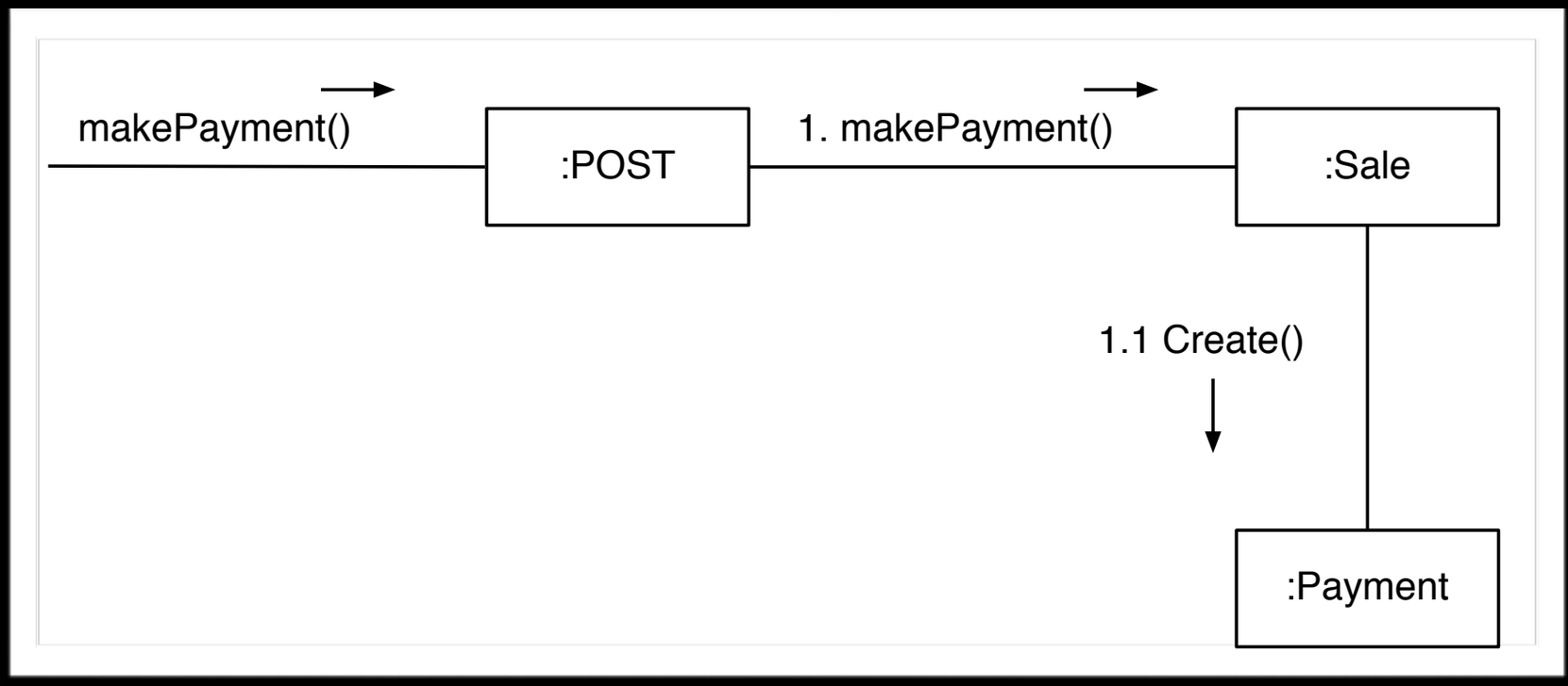
Example

Who should create instances of Payment?



Seems reasonable if we only consider *makePayment* but the more operations we add, the more responsibility (and less focused) *POST* gets.

Example

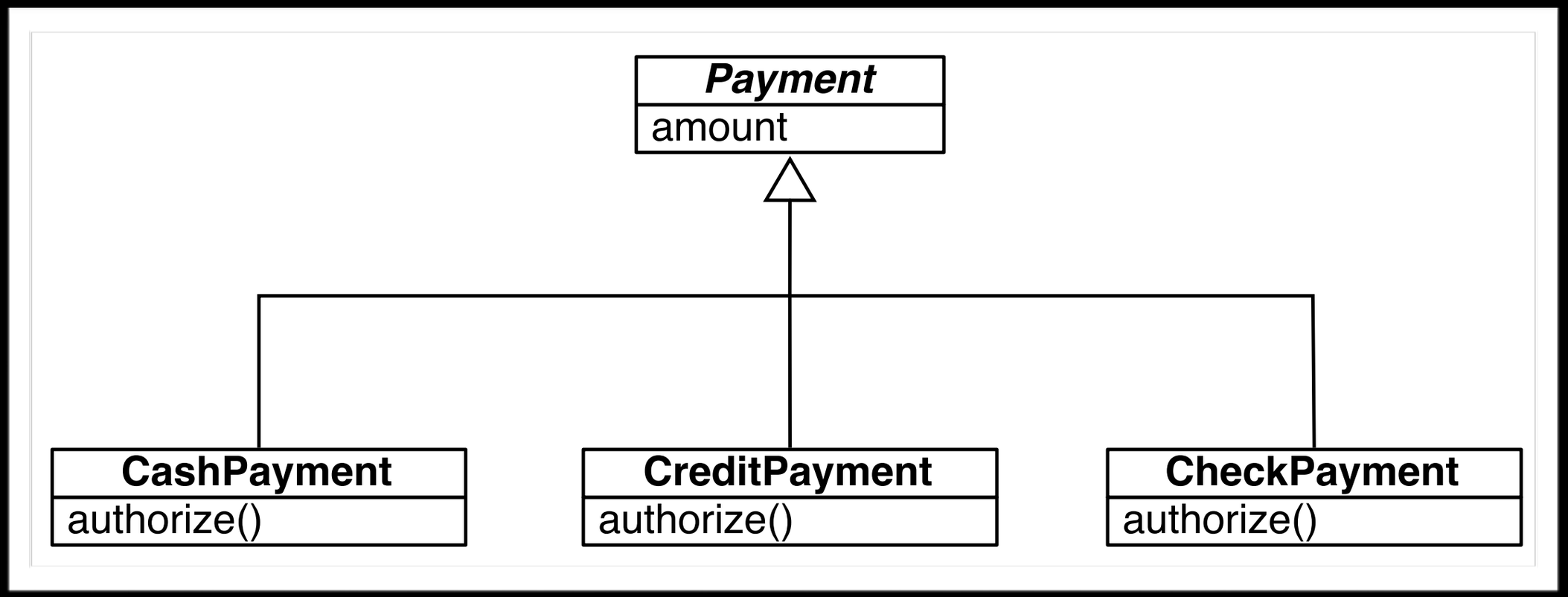


By moving the responsibility from *POST* to sale *POST* we increase cohesion (and reduce coupling!)

Polymorphism

- How can alternatives be managed based on class (type)
- Move responsibility to sub classes (types) and call the correct implementation

Example



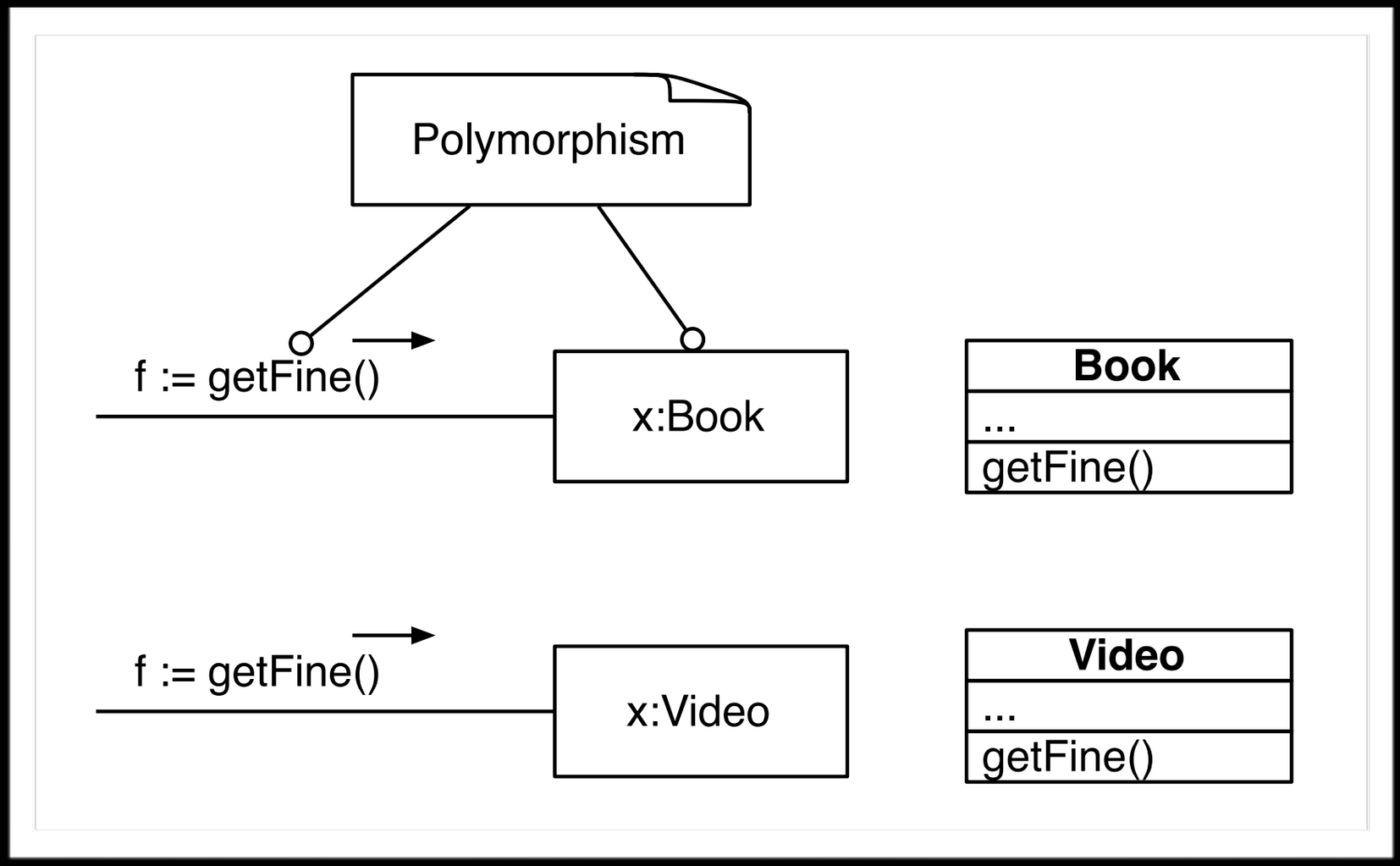
Each type of payment manage the authorization

Example

```
if x.type() == “Book” then
    fine := 2
if x.type() = “Video” then
fine := x.daysLate() * 10
```

Bad!

Example



Polymorphism is a better solution

Example

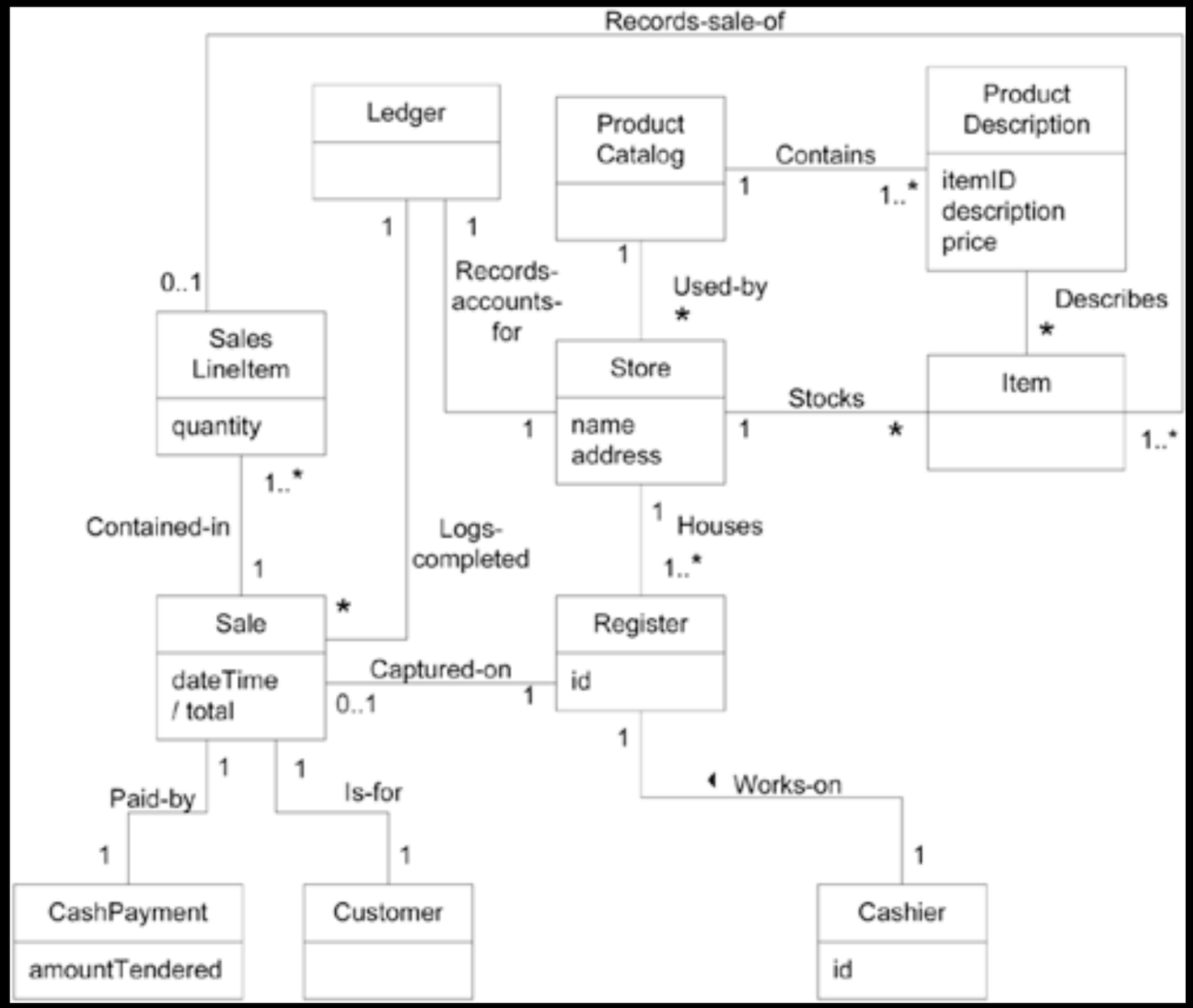
```
if (entity.type == "UseCase") then
        drawEllipse(...);
else if (entity.type == "Class") then
        drawRectangle(...);
...
end if
```

Cohesion?

Pure Fabrication

- Conceptual classes from the domain model can result in classes with low cohesion and high coupling.
- Invent new classes that are not part of the domain and assign these specific responsibilities.

Example

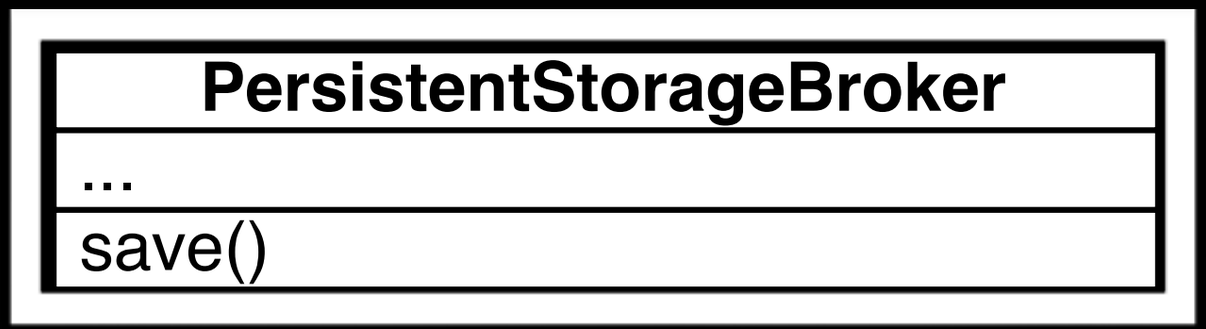


Assume that all sales should be saved in a database.

Example

- Information Expert suggests *Sale* should store these
  - *Sale* is now responsible => low cohesion
    - *Sale* depends on a set of other classes to interact with the database => high coupling
- To store information in the database might be a common task that several classes do => duplicated code and reduced reusability

Example



- Introduce a class to manage persistent storage
  - *Sale* remains of high cohesion/low coupling
  - PSB has reasonable cohesion
  - PSB is general and reusable

OK?

- Software is a not a simulation of the domain, but something that works within it
- Does not have to be identical to it
- Pure fabrication is a compromise between an exact model of the domain and maintainability/reusability
  - and we should always prefer the latter

When?

- Used when there are no suitable classes in the domain
  - common problem when classes depend on non-trivial technical artifacts
    - e.g., databases, graphics, etc
- Always try to find suitable classes in the domain, and use PF if there are none

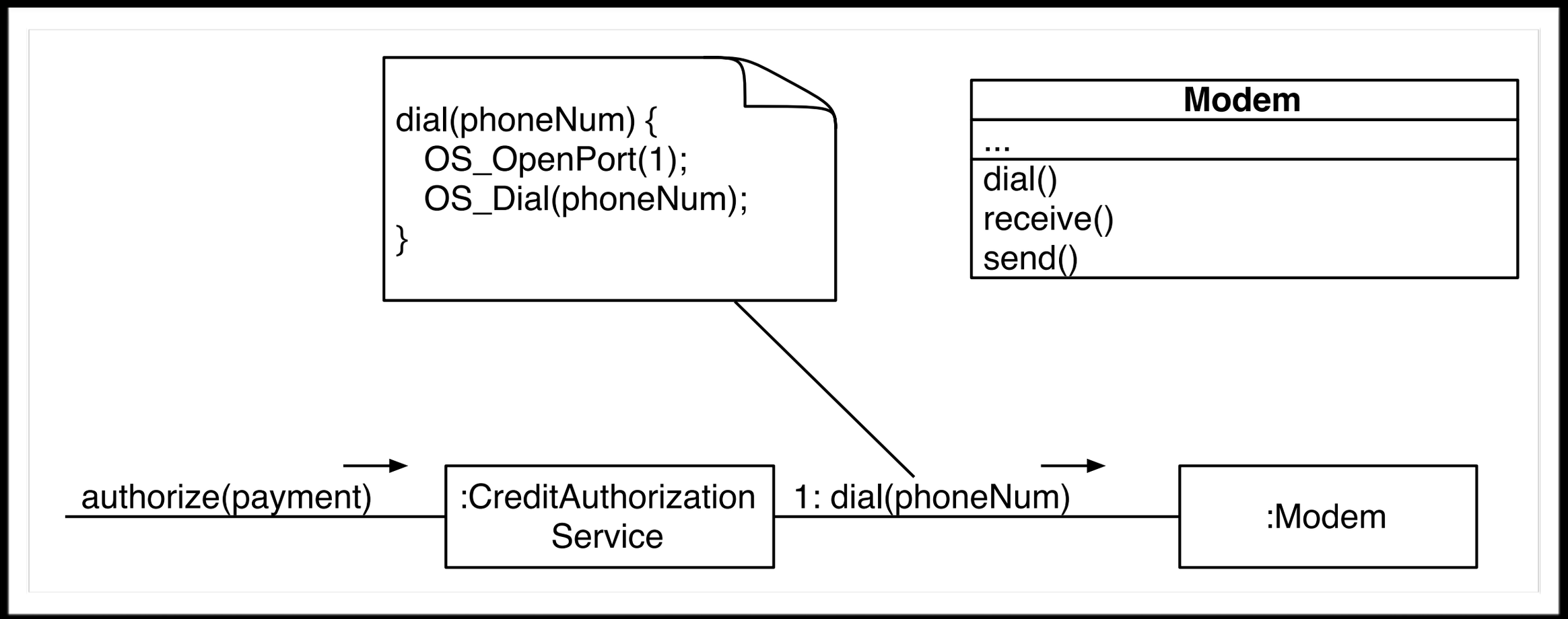
Indirection

- Remove direct coupling?
- Distribute responsibility to intermediary objects that manage the connection between two or more objects

Example

- Assume that the POST system authorise credit cards via a modem / phone line
- The operating system provides an API to communicate with the modem

Example


dial(phoneNum) {
OS_OpenPort(1);
OS_Dial(phoneNum);
}
Modem
...
dial()
receive()
send()
authorize(payment)
:CreditAuthorization Service
1: dial(phoneNum)
:Modem

Example

- If indirection is used to remove the direct coupling between *Sale* and the modem API:
  - the application is easier to port to other operating systems
  - the application is easier to maintain of the operating system/API/technology changes

Example

- *PersitentStorageBroker* (Pure fabrication)
  - PSB is an artificial class that maintains the connection between *Sale* and a database
  - PBS provides an indirection, which reduces coupling

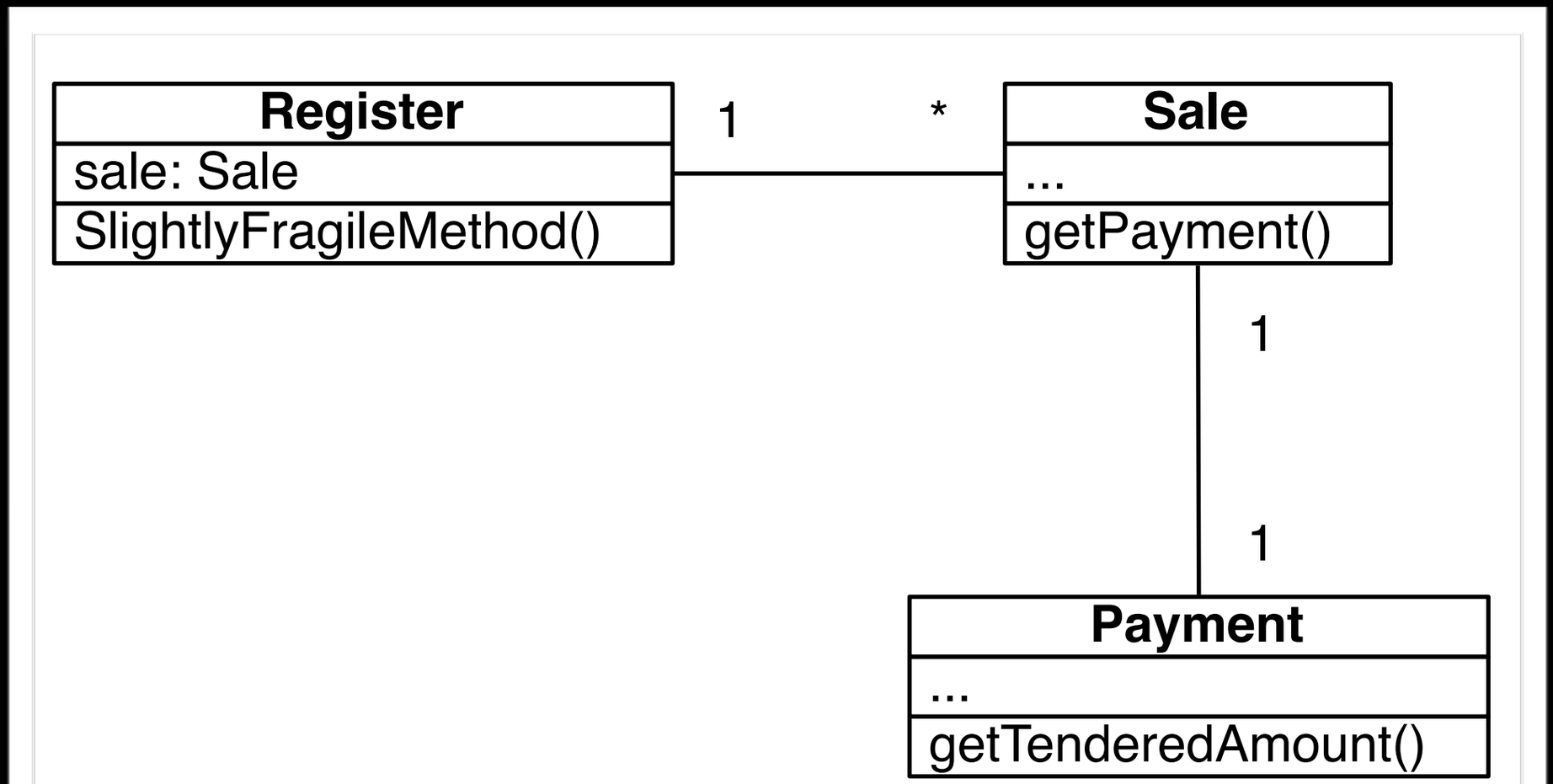
Don’t Talk to Strangers / Law of Demeter

- How to avoid knowledge of the structure (variable/operations) of indirect objects?
- Only model relevant associations
  - if two classes/objects have no reason to know of each other, they should not

Law of Demeter

- A method *M* in an object *O* should only call methods in the following objects:
  - self
  - objects used for parameters
  - objects created by *M*
  - objects contained in *O/M* (composition)

Example


Register
sale: Sale
SlightlyFragileMethod()
1
*
Sale
...
getPayment()
1
1
Payment
...
getTenderedAmount()

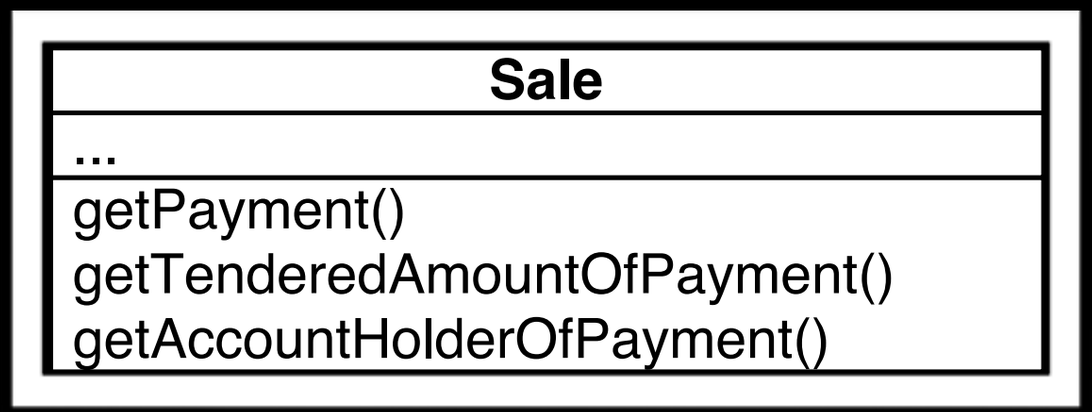
Example

```
class Register {
   private Sale sale;

   public void slightlyFragileMethod() {
      // sale.getPayment() sends a message to a "familiar" (passes #3)
      // but in sale.getPayment().getTenderedAmount()
      // the getTenderedAmount() message is to a "stranger" Payment

      Money amount = sale.getPayment().getTenderedAmount();

      // ...
   }

   // ...
}
```

Example

```
public void moreFragileMethod() {
   AccountHolder holder =
      sale.getPayment().getAccount().getAccountHolder();
   // ...
}


   public void doX() {
      F someF =
         foo.getA().getB().getC().getD().getE().getF();
      // ...
   }
```

Fix




```
// case 1
Money amount = sale.getTenderedAmountOfPayment();
// case 2
AccountHolder holder = sale.getAccountHolderOfPayment();
```

Protected Variations

- How can classes, subsystems and systems be constructed to minimise the effect of instability and change on other parts?
- Identify points where variation can happen or points that could be unstable
- Distribute responsibility to create stable interfaces

Protected Variations

- We already know variations of this!
  - indirection
  - polymorphism
  - encapsulation
- New
  - interfaces

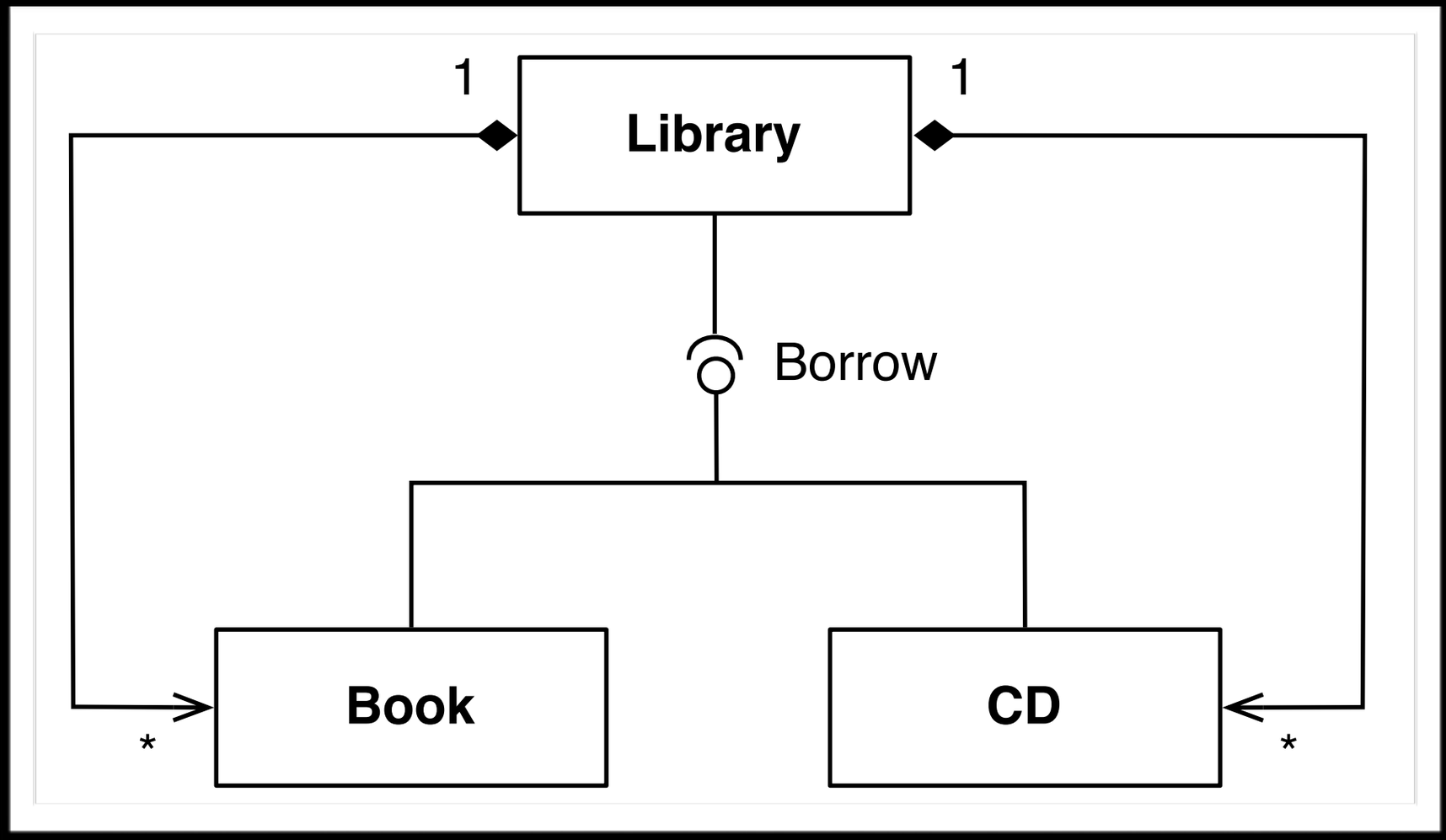
Interface

- Separates specification of functionality from implementation
- An interface cannot be instantiated
- Specifies a contract that can be realized by several different classes
- A class that realises an interface promises to uphold/follow the contract

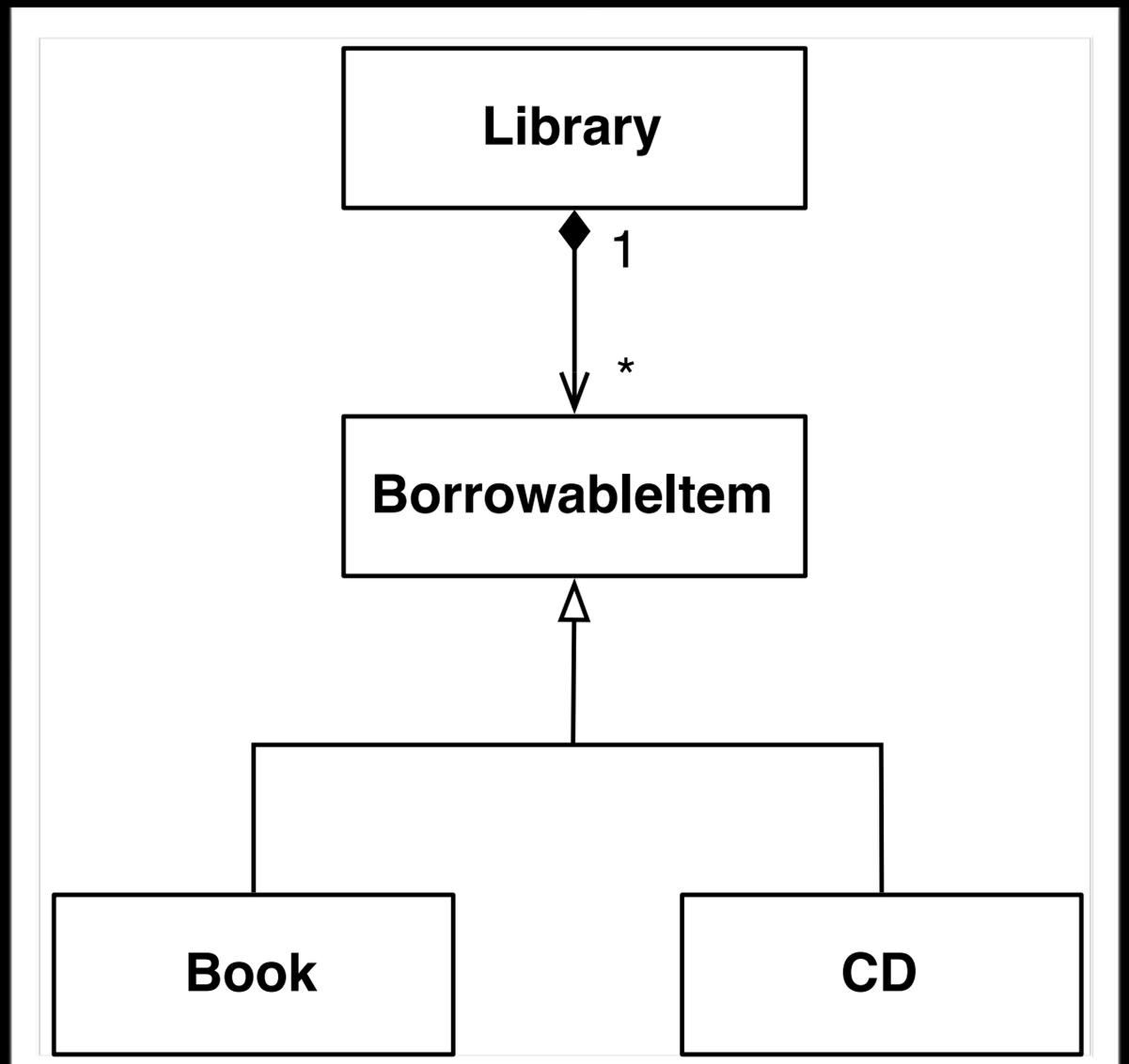
Example

```
interface Borrowable {
   void borrowIt();
   void returnIt();
   bool isOverdue();
}

class Book : Borrowable {
   public void borrowIt() { // do something }
   public void returnIt() { // do something }
   public bool isOverdue() { return false; }
   public int somethingDifferent(int x) { return x*x; }
}
```

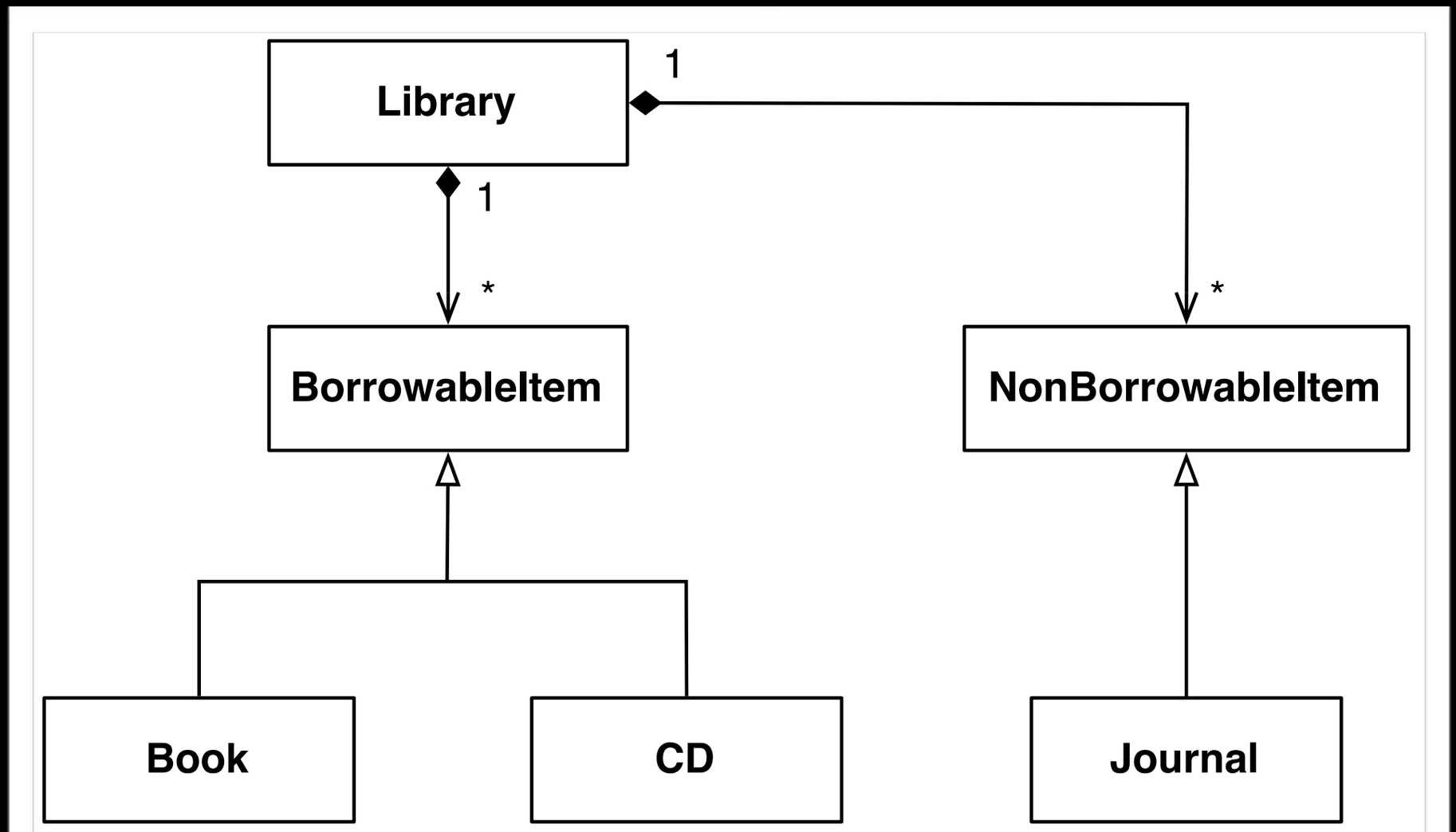
Example



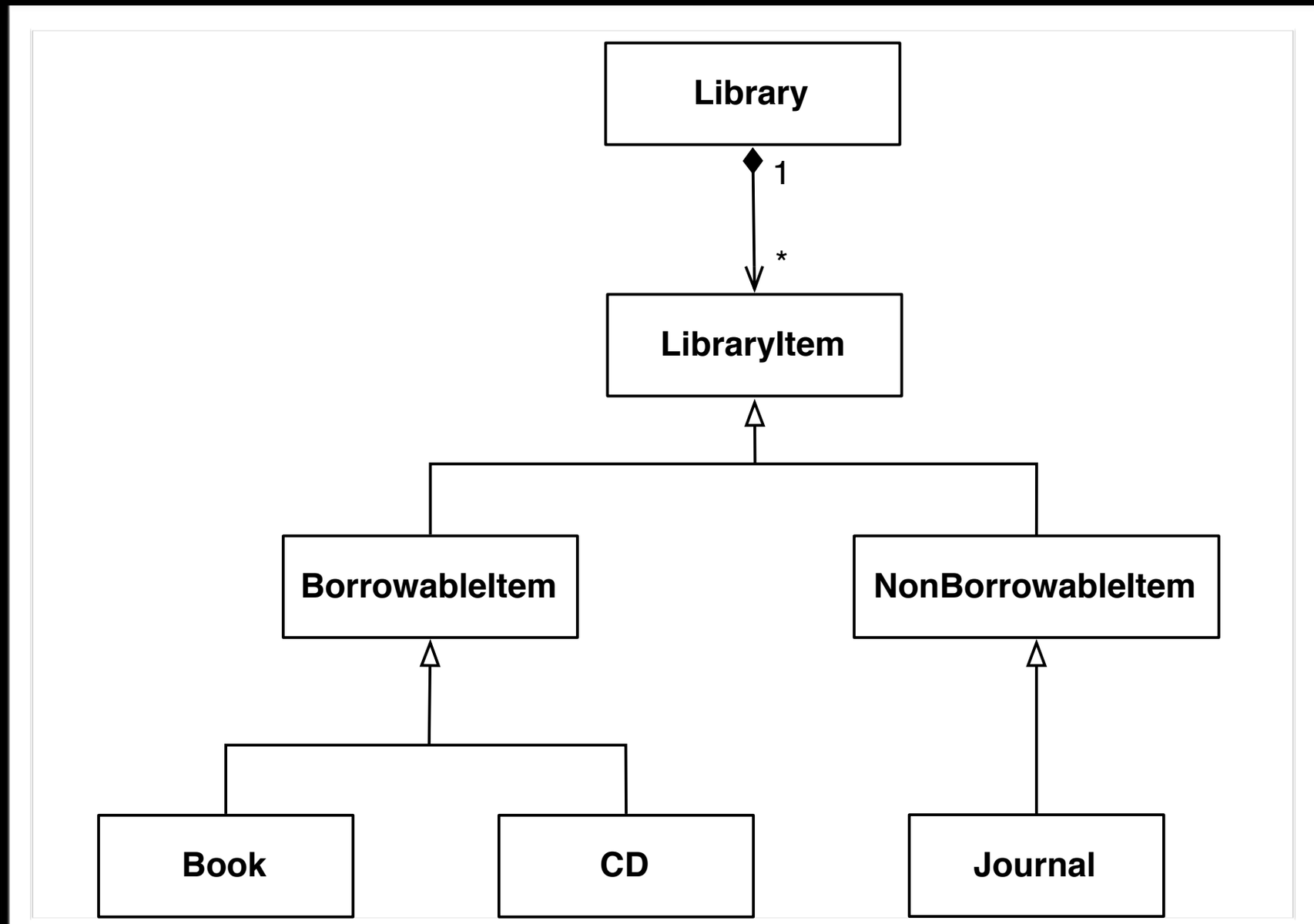
Example



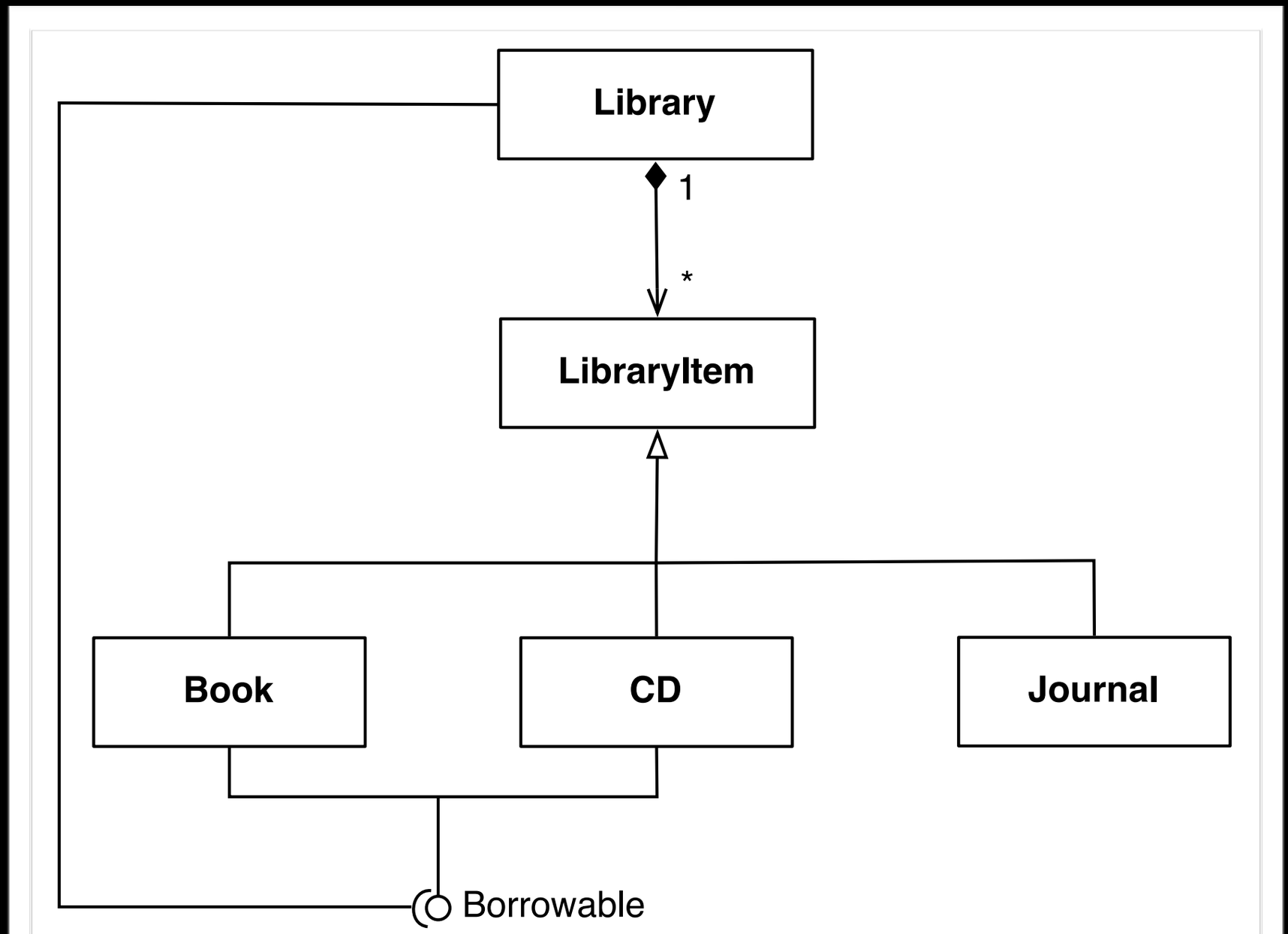
Example


Library
1
1
*
*
BorrowableItem
NonBorrowableItem
Book
CD
Journal

Example


Library
1
*
LibraryItem
BorrowableItem
NonBorrowableItem
Book
CD
Journal

Example



Protected Variations

- Use an artificial, shared abstraction to represent a base case
- Hide details, changes, and variations behind this base case
  - shared base class
  - interface
  - ...

Example

```
$link = mysql_connect('localhost', 'mysql_user', 'mysql_password');
$result = mysql_query("some query");

while ($row = mysql_fetch_assoc($result)) {
   echo "output something"
}

mysql_free_result($result);
```

Example

```
$db = new PDO("mysql:host=$hostname;dbname=articles",
   $username, $password);
$sql = "Select something;";
$result = $db->query($sql);
foreach ($result as $row) {
   echo "Output something;";
}
$db = null;
```

PV and LoD

- Law of Demeter can be considered a part of Protected Variations
- Variations happens in structure, and this can be hidden by a stable interface

But...

- Separate
  - variation points
  - evolution points
    - The first one should always be handled
    - The second one relates to future changes that may happen
      - can cost a lot of time and effort, and the evolution might never happen

Example

```
public void Process() {
    string connectionString = getConnectionString();
    SqlConnection connection = new SqlConnection(connectionString);
    DataServer1 server = new DataServer1(connection);
    int daysOld = 5;
    using (SqlDataReader reader = server.GetWorkItemData(daysOld)) {
        while (reader.Read()) {
            string name = reader.GetString(0);
            string location = reader.GetString(1);
            processItem(name, location);
        }
    }
}
```

Problem?

From MSDN

Example

```
public void Process() {
   DataServer2 server = new DataServer2();
   foreach (DataItem item in server.GetWorkItemData(5)) {
      processItem(item);
   }
}
```

Fix!

From
MSDN

Example

```
public interface DataService {
    InsuranceClaim[] FindClaims(Customer customer);
}

public class Repository {
    public DataService InnerService { get; set; }
}
```

```
public class ClassThatNeedsInsuranceClaim {
    private Repository _repository;

    public ClassThatNeedsInsuranceClaim(Repository repository) {
        _repository = repository;
    }

    public void TallyAllTheOutstandingClaims(Customer customer) {
        InsuranceClaim[] claims =
          _repository.InnerService.FindClaims(customer);
    }
}
```

Problem?

From MSDN

Example

```
public class Repository2 {
    private DataService _service;

    public Repository2(DataService service) {
        _service = service;
    }

    public InsuranceClaim[] FindClaims(Customer customer) {
        return _service.FindClaims(customer);
    }
}

public class ClassThatNeedsInsuranceClaim2 {
    private Repository2 _repository;

    public ClassThatNeedsInsuranceClaim2(Repository2 repository) {
        _repository = repository;
    }

    public void TallyAllTheOutstandingClaims(Customer customer) {
        InsuranceClaim[] claims = _repository.FindClaims(customer);
    }
}
```

Fix!

From MSDN

Example

```
public class DumbPurchase {
    public double SubTotal { get; set; }
    public double Discount { get; set; }
    public double Total { get; set; }
}

public class DumbAccount {
    public double Balance { get; set;}
}

public class ClassThatUsesDumbEntities {
    public void MakePurchase(DumbPurchase purchase, DumbAccount account) {
        purchase.Discount = purchase.SubTotal > 10000 ? .10 : 0;
        purchase.Total = purchase.SubTotal*(1 - purchase.Discount);

        if (purchase.Total < account.Balance)
            account.Balance -= purchase.Total;
        else
            rejectPurchase(purchase, "You don't have enough money.");
    }
}
```

Problem?

From MSDN

Example

```
public class Purchase {
    private readonly double _subTotal;

    public Purchase(double subTotal) {
        _subTotal = subTotal;
    }

    public double Total {
        get { double discount = _subTotal > 10000 ? .10 : 0; return _subTotal*(1 - discount); }
    }
}

public class Account {
    private double _balance;

    public void Deduct(Purchase purchase, PurchaseMessenger messenger) {
        if (purchase.Total < _balance)
            _balance -= purchase.Total;
        else
            messenger.RejectPurchase(purchase, this);
    }
}
```

Fix!

From MSDN

Example

```
public class ClassThatObeysTellDontAsk {
    public void MakePurchase(Purchase purchase, Account account) {
        PurchaseMessenger messenger = new PurchaseMessenger();
        account.Deduct(purchase, messenger);
    }
}
```

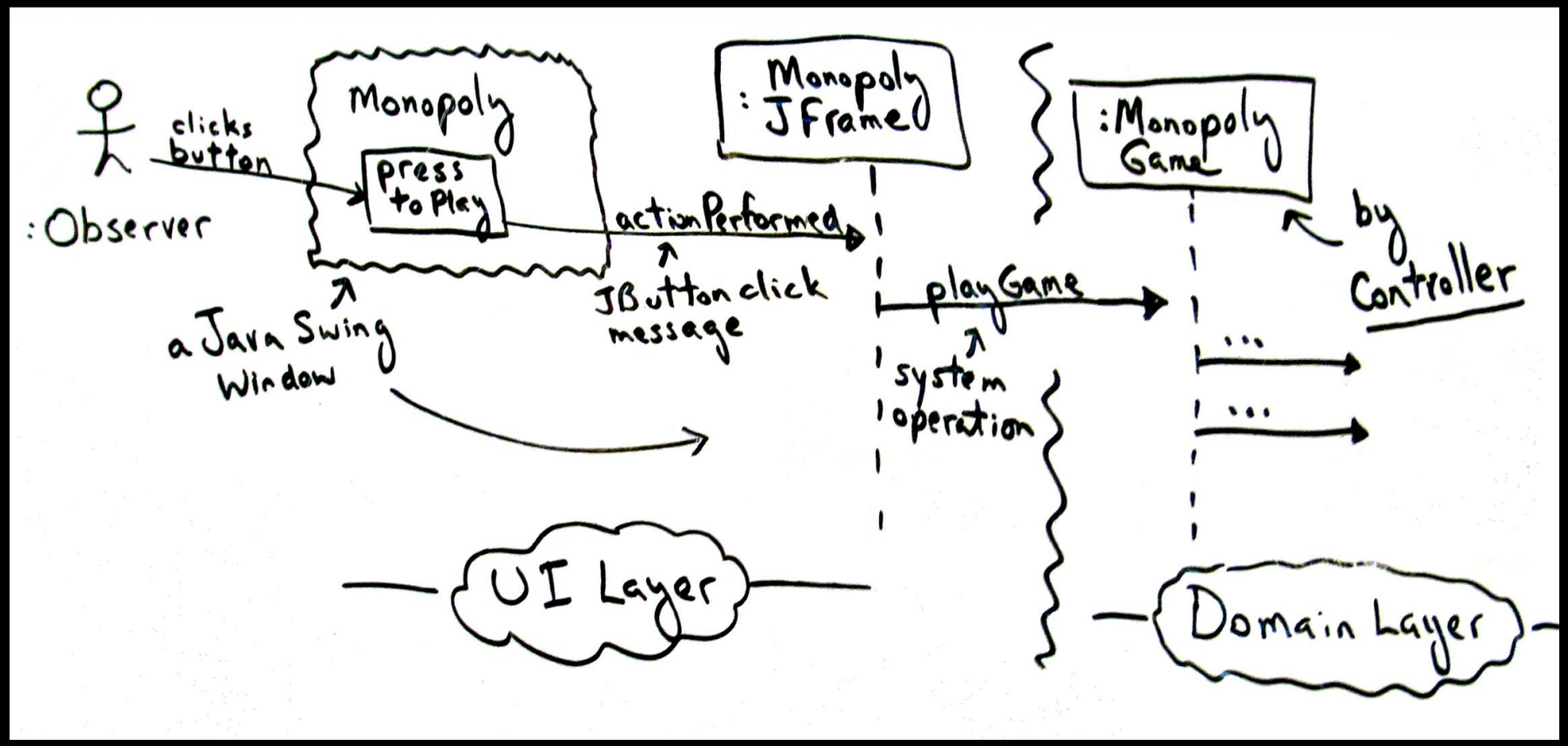
Fix!

From MSDN

Controller

- Who handles a system or UI event?
- Distributes responsibility in two ways:
  1. the entire system (façade)
  2. the case where the event occurred (session controller)

Example



Who is the controller?

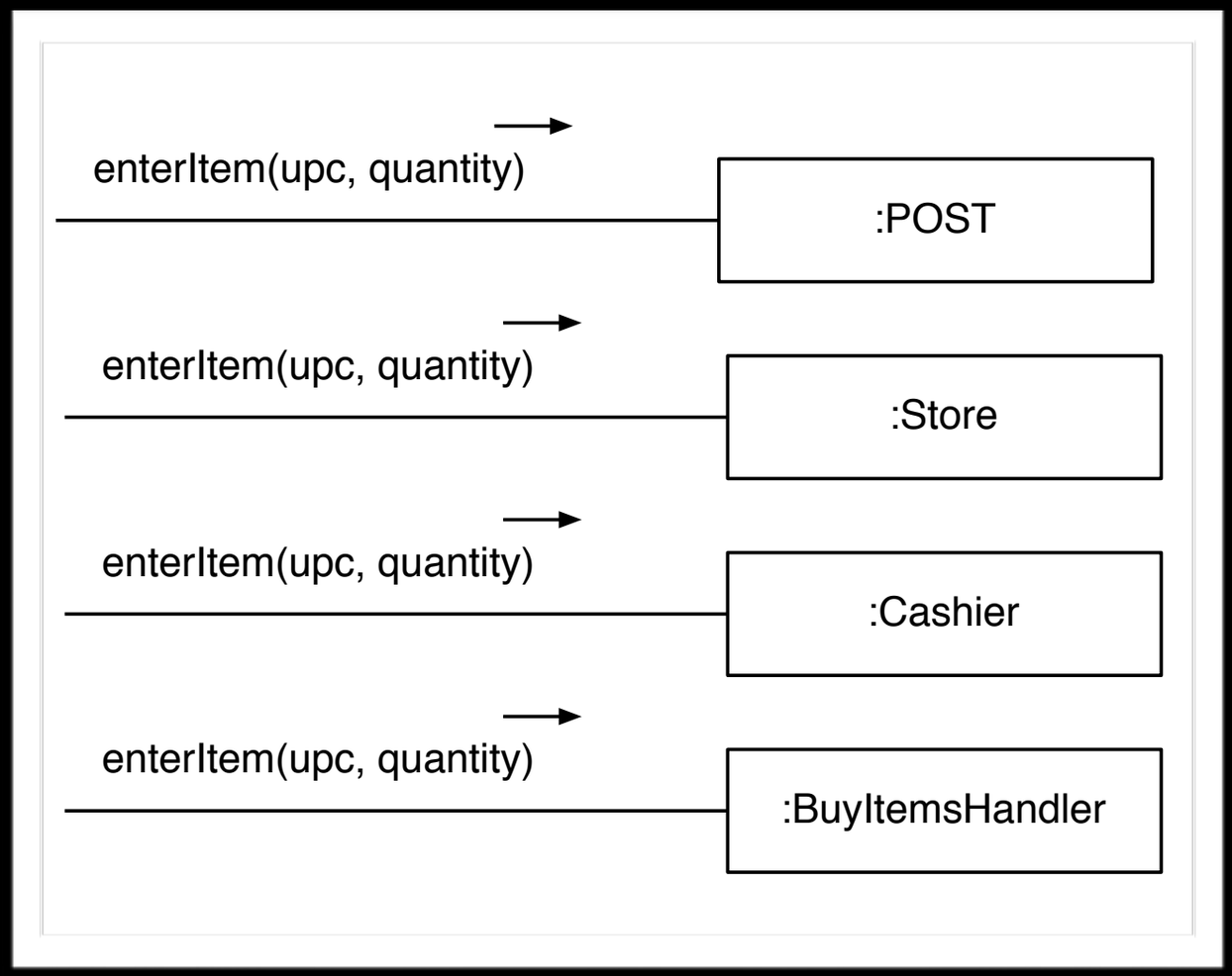
Example

- Assume a "Buy Items" case with the following events
  - *enterItem()*
  - *endSale()*
  - *makePayment()*
- Who is responsible for *enterItem()*?

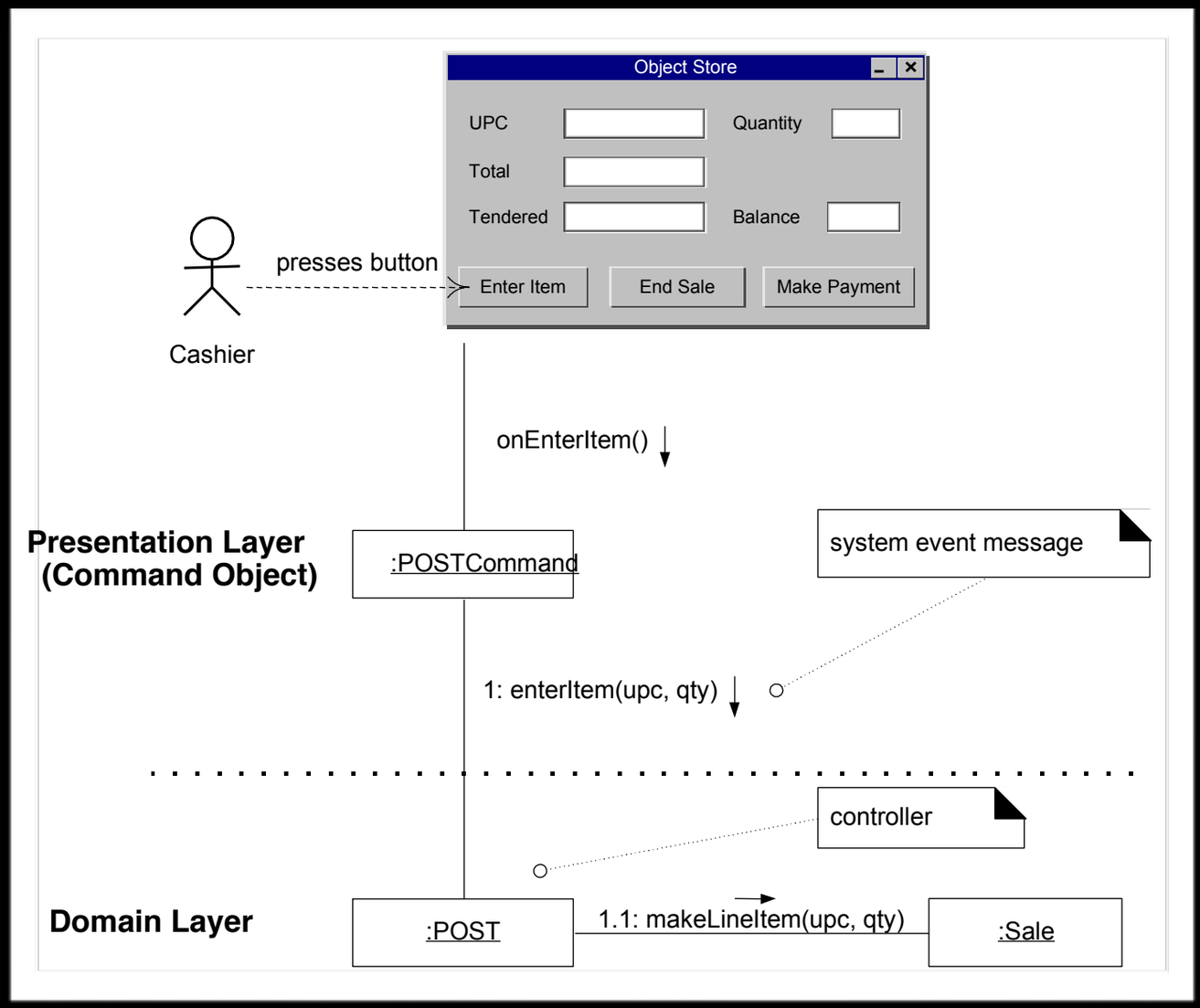
Example

Controller gives us four possibilities

1. The system
2. The store
3. An actor that is part of the context
4. A "session controller"

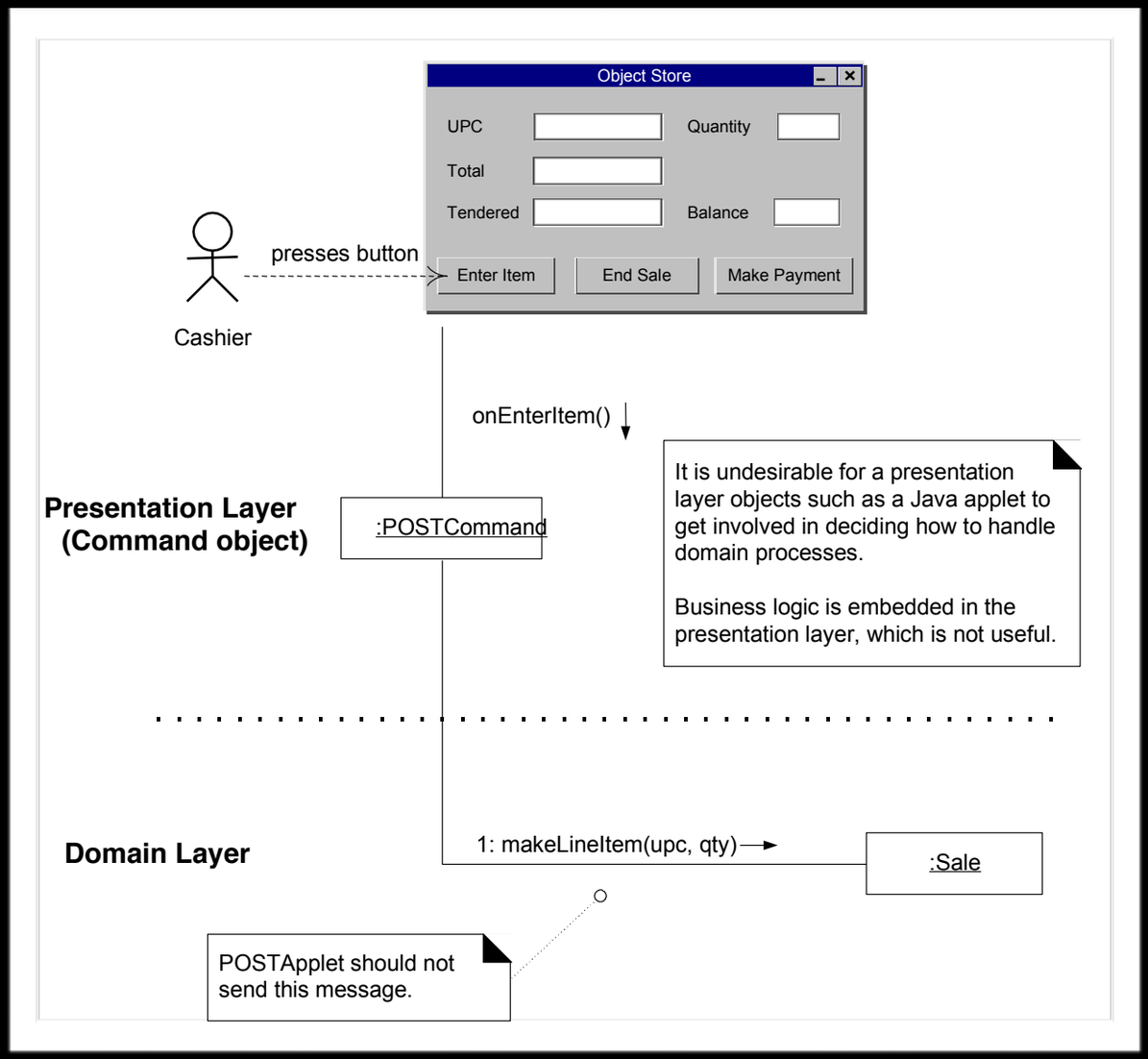


Example



A presentation layer separates the domain

Example



A mix between presentation and domain

Benefits

- Controller objects disconnects internal and external events from each other
- Controller objects can have low cohesion and high coupling