ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

SYSTEMS ANALYSIS AND DESIGN

Nguyen Thanh Binh, Nguyen Quang Vu, Le Viet Truong, Nguyen Thi Hanh, Vo Van Luong, Le Thi Bich Tra

Faculty of Computer Science

Vietnam - Korea University of Information and Communication Technology (VKU)

Design principles

- General Responsibility Assignment Software Principles/Patterns - GRASP

Understanding responsibilities is key to object-oriented design.

Martin Fowler

Responsibilities-Driven Design

- RDD is a metaphor for thinking about object-oriented design.
- Think of software objects similar to people with responsibilities who collaborate with other people to get work done.
- RDD leads to viewing an OO design as a community of collaborating responsible objects.

GRASP

- General Responsibility Assignment Software Patterns or Principles (GRASP)
  - Pattern is a solution which can be applied to a problem in a new context
- A learning aid for OO Design with responsibilities.
- A collection of patterns/principles for achieving good design - patterns of assigning responsibility.

Responsibility

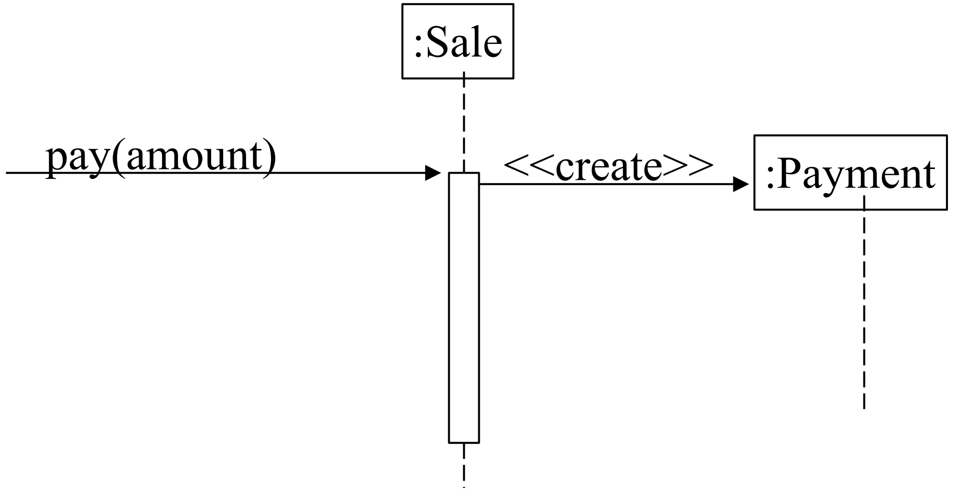
- A **responsibility** is an duty or a contract of a class
- The determination of the attributes and operations of a class is essentially based on its responsibilities
- The responsibilities of an object relate to the behaviour of an object
- Two main types of responsibility
  - **Do**
    - The object accomplishes something itself
    - The object initiates an action of another object
    - The object controls or coordinates activities of other objects
  - **Know**
    - The object knows private encapsulated data
    - The object knows the objects to which it is linked
    - The object has data that it can calculate or derive

Responsibility

- **The responsibilities are assigned to classes during the design phase**
  - Example
    - An object of *Sale* class is responsible for creating an object of *Payment* class (do)
    - An object of *Sale* class is responsible for knowing its total (know).
- The translation of responsibilities into methods of classes depends on the granularity of the responsibilities
  - A responsibility can be translated by several methods of several classes
    - Responsibility “offer access to the database” can be translated to several methods of several classes
  - A responsibility can be translated by one method
    - Responsibility “create a *Sale*” can be translated by only one method.

Assignment and discovery of responsibilities

- The assignment of responsibilities to objects is very important in object-oriented design.
- **The discovery of responsibilities is achieved when building interaction diagrams**

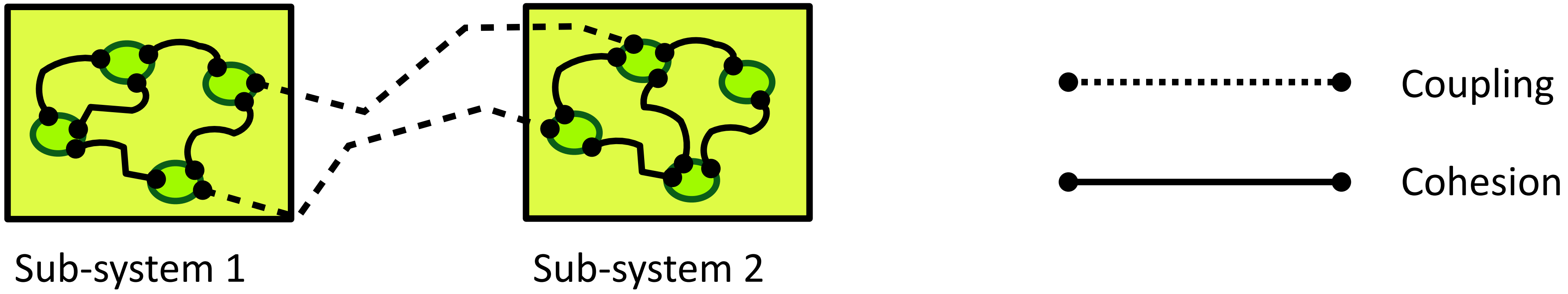


GRASP patterns

- We consider 5 among 9 GRASP patterns/principles
  - **Low Coupling**: assigning responsibilities in a low coupling way
  - **High Cohesion**: assigning the responsibilities to ensure that cohesion remains high
  - **Creator**: assigning the creation responsibility of an object to another object
  - **Information Expert**: the common principle when assigning responsibilities to classes
  - **Controller**: assigning the responsibility for management of the system event messages
  - **Polymorphism**
  - **Indirection**
  - **Pure fabrication**
  - **Protected variations**

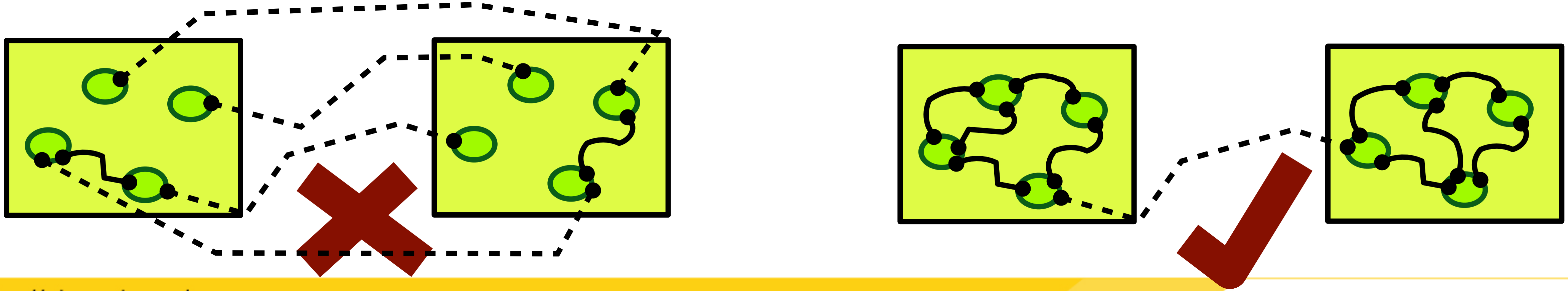
Coupling and Cohesion

- **Coupling**: Amount of relations between objects/sub-systems
- **Cohesion**: Amount of relations within sub-system



Properties of a good architecture

- Minimises coupling between modules
  - Goal: modules don’t need to know much about one another to interact
  - Low coupling makes future change easier
- Maximises cohesion within modules
  - Goal: the content of each module are strongly inter-related
  - High cohesion makes a module easier to understand



Low coupling

- Problem: How to support low dependency, low change impact, and increase reuse?
- Coupling:
  - Measure how strongly one element is connected to, has knowledge of or relies on other elements
  - An element with low (or weak) coupling is not dependent on two many other elements

When are two classes coupled?

- Common forms of coupling from TypeX to TypeY
  - TypeX has an attribute that refers to a TypeY instance
  - A TypeX object calls on services of TypeY object
  - TypeX has a method that references an instance of TypeY (parameter, local variable, return type)
  - TypeX is a direct or indirect subclass of TypeY
  - TypeX is an interface and TypeY implements that interface

High coupling (Bad)

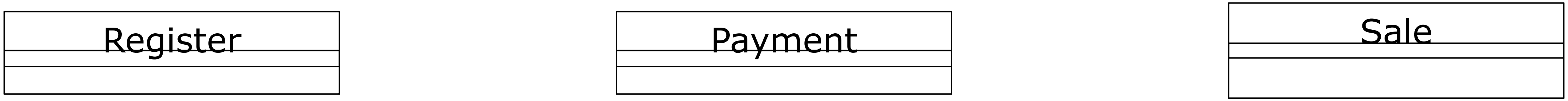
- A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable and suffer from the following problems:
  - Force local changes because of changes in related classes
  - Harder to understand in isolation
  - Harder to reuse because its use requires the additional presence of the classes on which it is dependent

Solution

- Assign responsibility so that coupling remain low
- Use this principle to evaluate alternatives

Example

- We have three following classes in the Cash Register system



Register

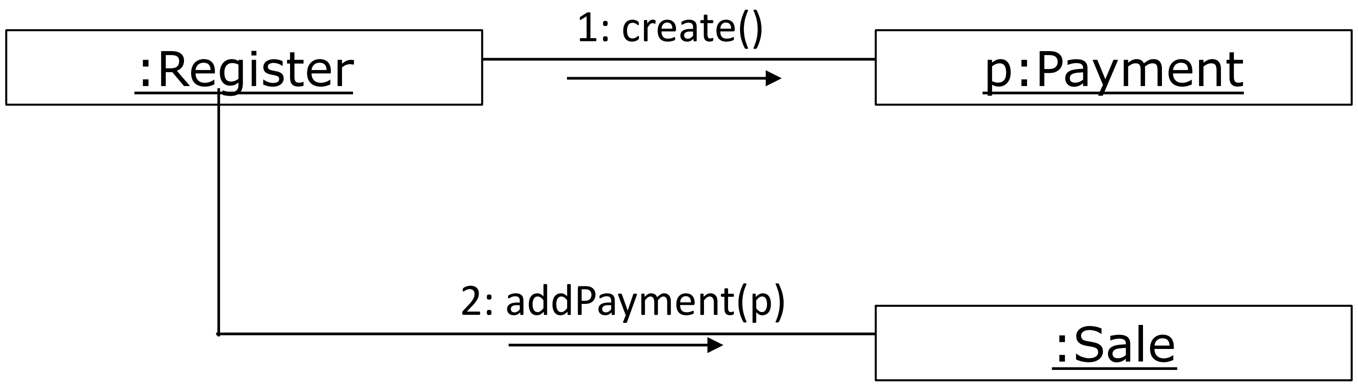
Payment

Sale

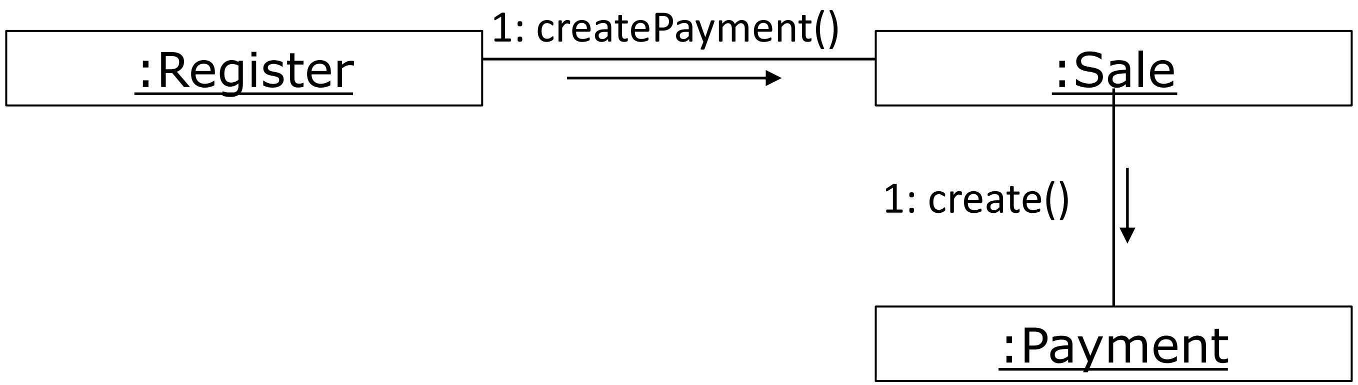
- Supposing that we would like to create an instance of Payment and associate it with Sale.
- How can we assign responsibilities to adhere to Low Coupling pattern?

Solutions

- Solution 1

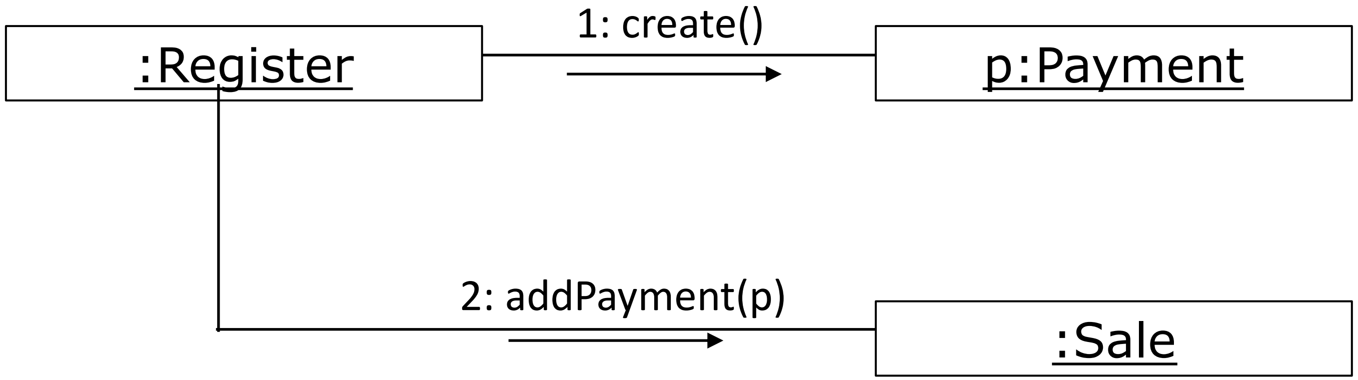


- Solution 2

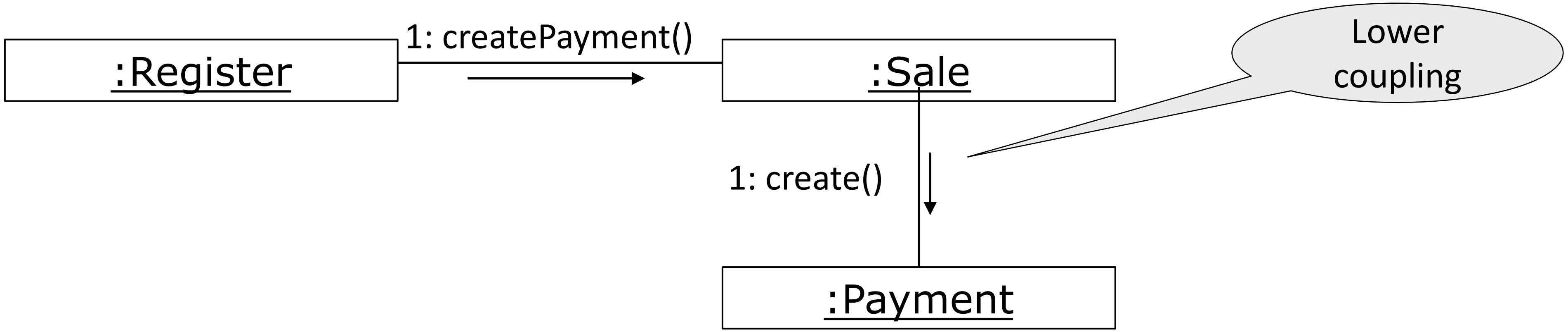


Solutions

- Solution 1: *Register* knows both *Payment* and *Sale*. *Register* depends on both *Payment* and *Sale*.



- Solution 2: *Register* and *Sale* are coupled, *Sale* and *Payment* are coupled.



High Cohesion pattern

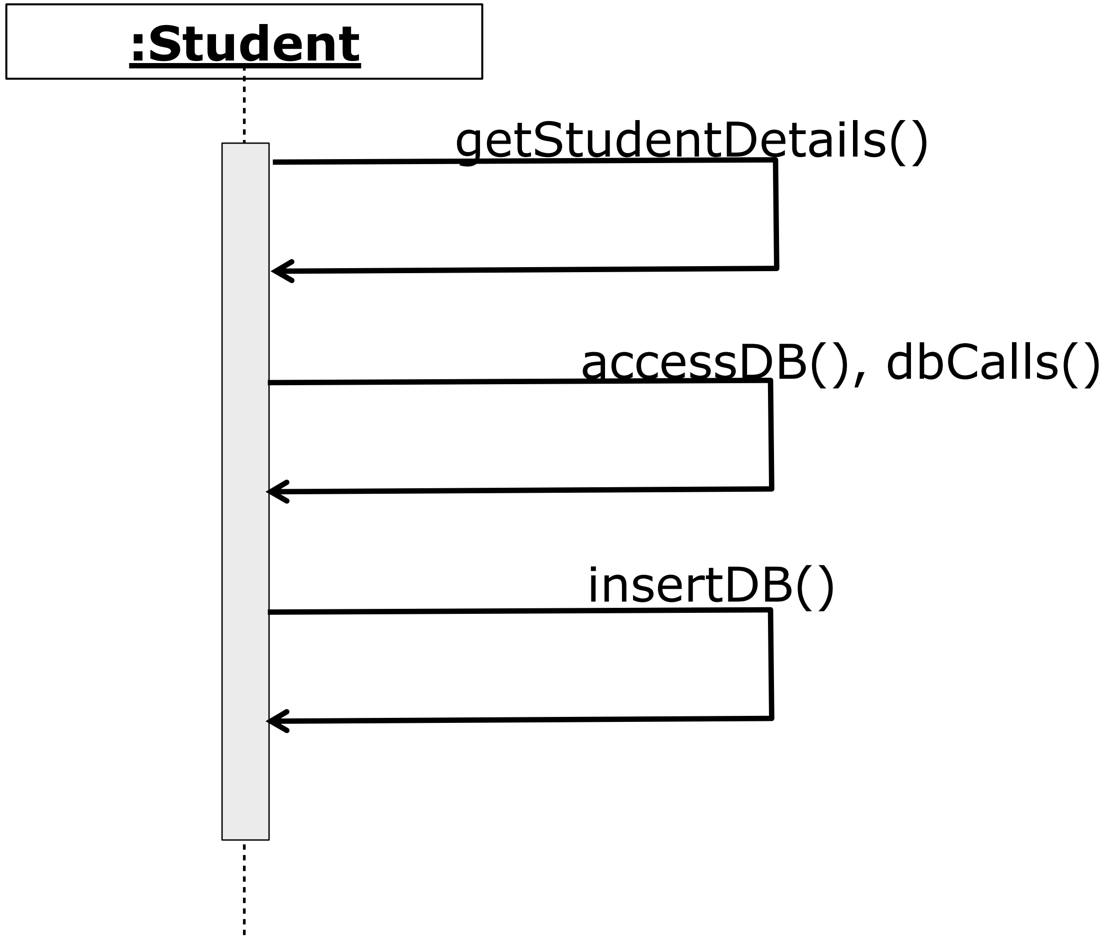
- Problem
  - How to ensure that the operations of any element are functionally related?
- Solution
  - Clearly define the purpose of the element
  - Gather related responsibilities into an element
- Benefit
  - Easily to understand and maintain

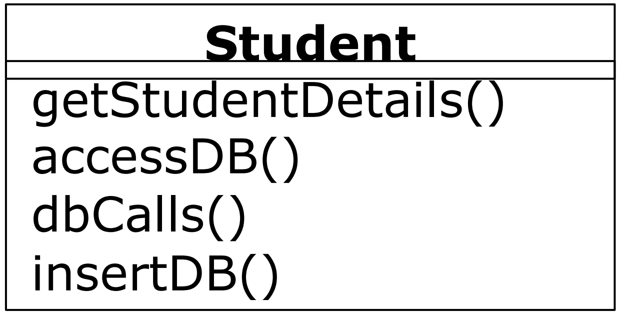
Low cohesion

- A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - constantly affected by change

High Cohesion pattern

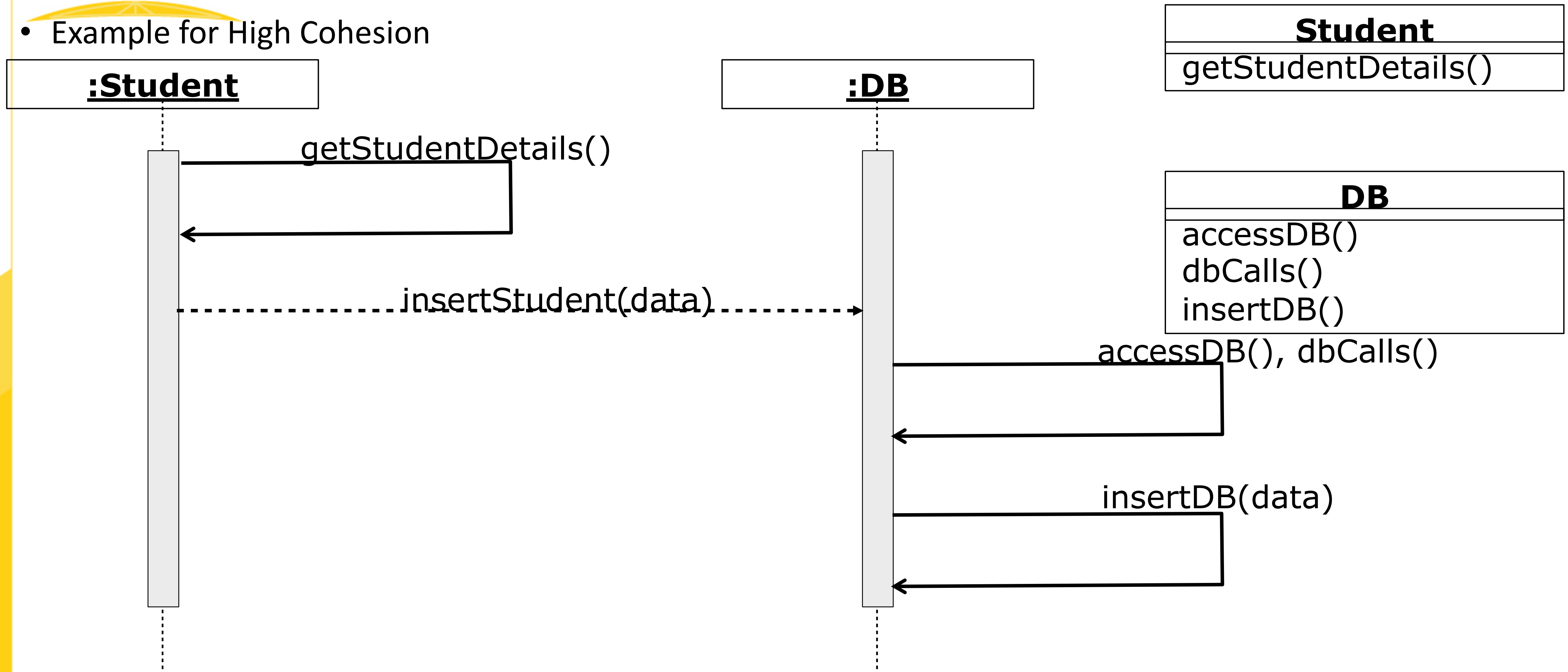
- Example for Low Cohesion





High Cohesion pattern

- Example for High Cohesion

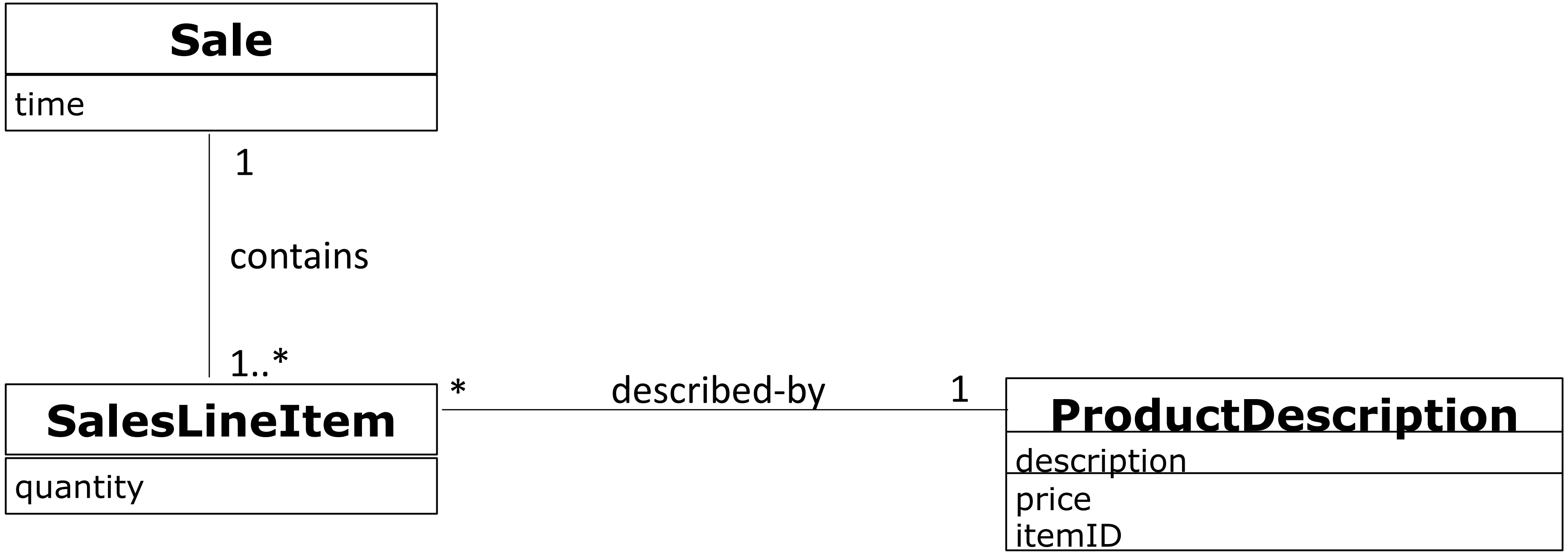


Rules of thumb

- For high cohesion, a class must
  - have few methods
  - have a small number of lines of code
  - not do too much work
  - have high relatedness of code

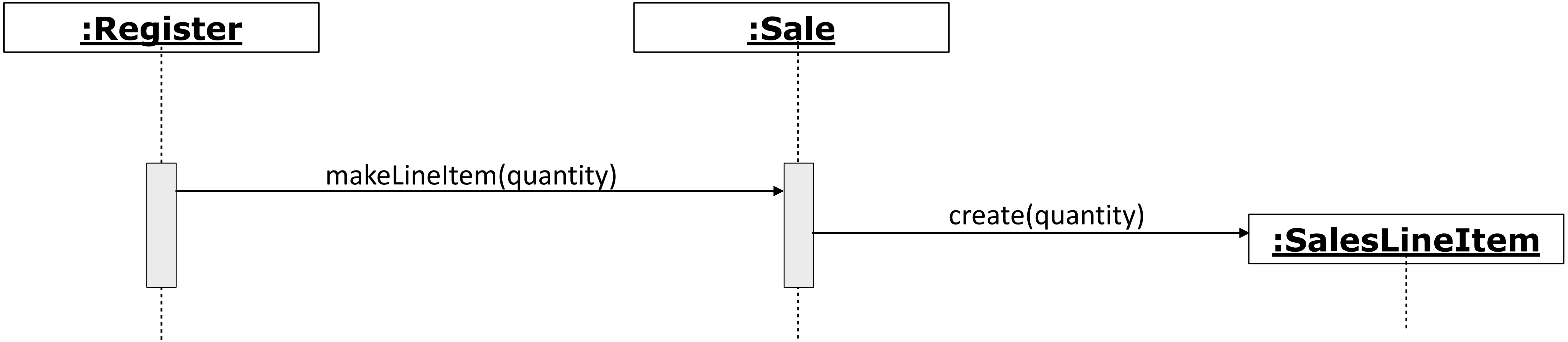
“creator” pattern

- Problem
  - Who is responsible for creating objects/instances of a class?
- Example
  - Who should be responsible for creating a SalesLineItem instance?



“creator” pattern

- Example (continue)
  - *Sale* contains *SalesLineItem*, so *Sale* should be responsible for creating objects of *SalesLineItem*



  - *“makeLineItem(quantity)”* method will be introduced to *Sale* class

“creator” pattern

- Discussion
  - Basic idea is to find a creator that needs to be connected to the created object in any event
  - Also need initialisation data to be nearby - sometimes requires that it is passed into client. e.g., *ProductionDescription* needs to be passed in.

  - Assign class B the responsibility to create an instance of class A if one of these is true
    - B contains A
    - B aggregates A
    - B has data for initialising A
    - B closely uses A

“creator” pattern

- Application
  - Guide in the assigning responsibility for creating objects
  - Help to find the class who is responsible for creating objects

- Advantages
  - The “creator” pattern supports the low coupling between classes
    - Fewer dependencies and more reusability
    - The coupling is not increased because the created class is visible to the “creator” class

Information Expert pattern

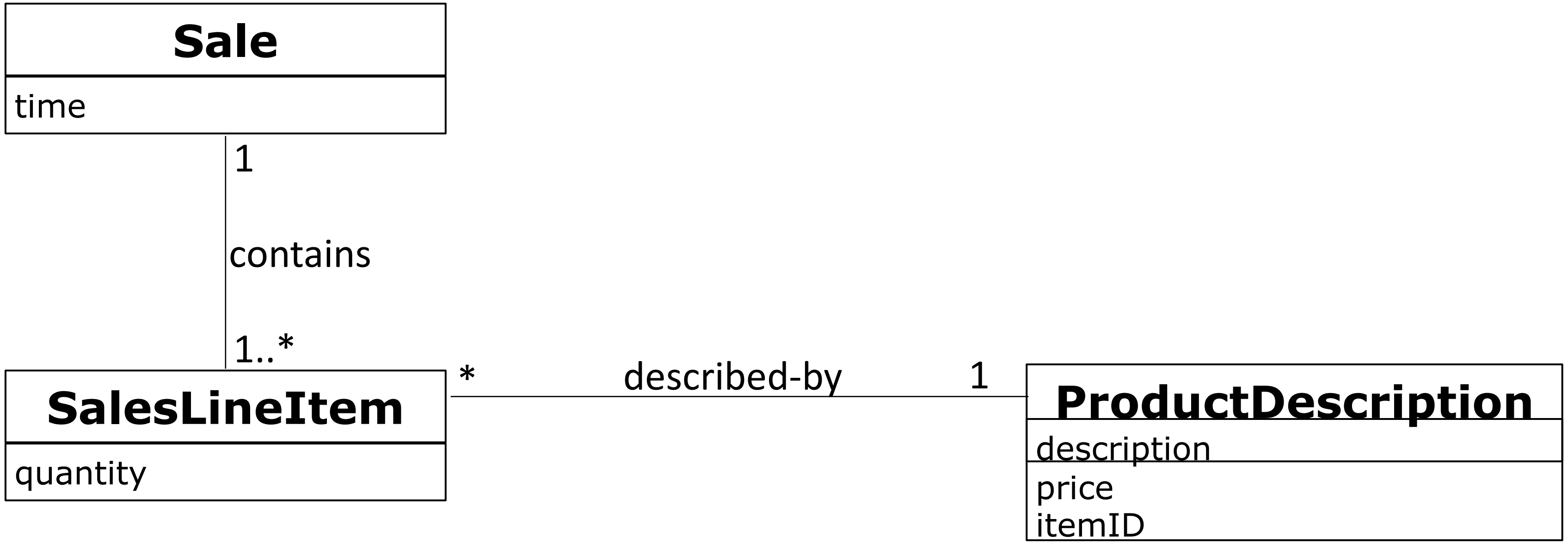
- Problem
  - What is the general principle of assigning responsibilities to objects?
    - Consider that there may be 100s or 1000s of classes
    - To which ones do we assign a particular functionality?
    - Assigning well makes our design easier to understand, maintain, extend and reuse.

- Solution
  - Assign responsibility to the information expert - the class that has the information to fulfil the responsibility

- Application
  - One of the most used patterns in object-oriented design
  - Accomplishing of a responsibility can request information distributed among several objects or classes, this implies several “partial experts” working together to fulfil the responsibility

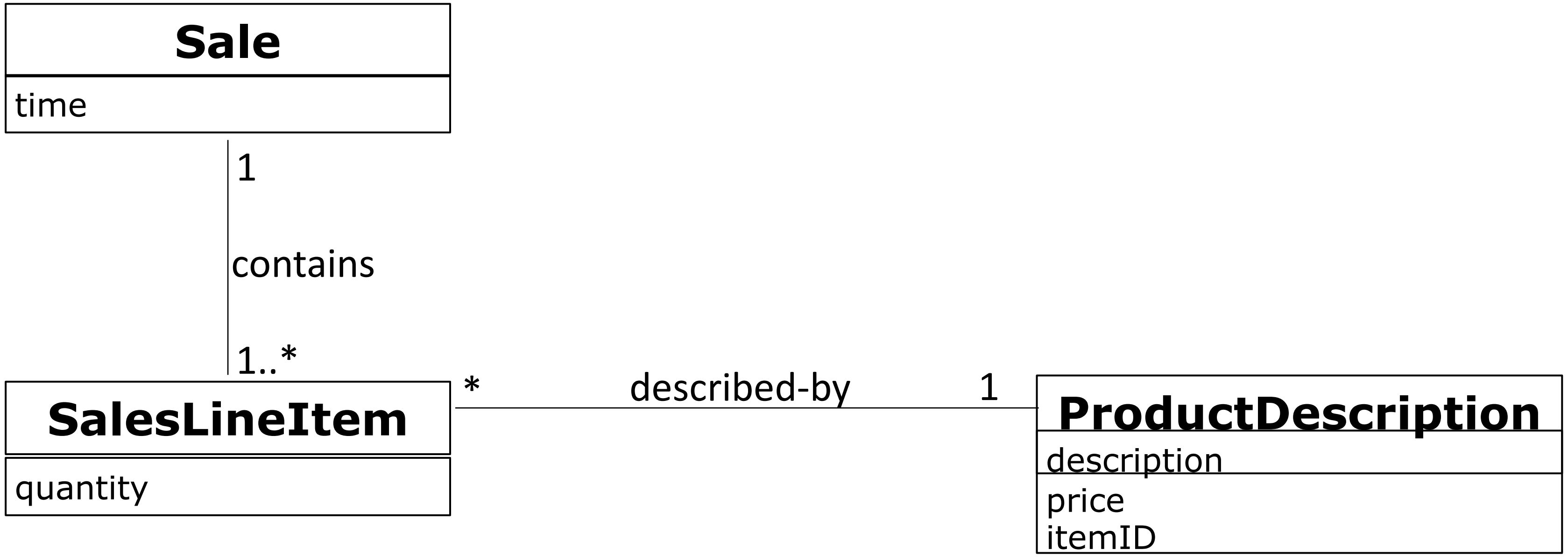
Information Expert pattern

- Example
  - In the *CashRegister* system, who is responsible for knowing the grand total of a *Sale*?



Information Expert pattern

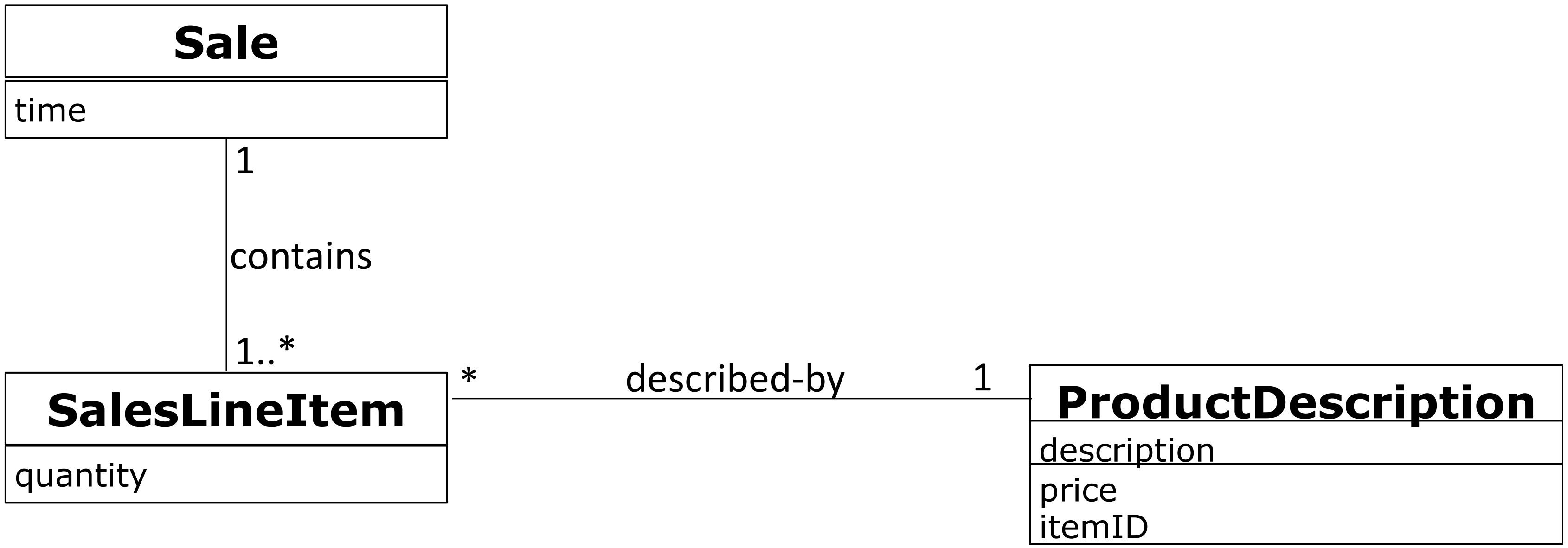
- Example: Responsibilities



Class	Responsibility
Sale	knows sale total
SaleLineItem	knows line items subtotal
ProductDescription	knows product price

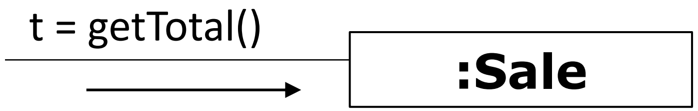
Information Expert pattern

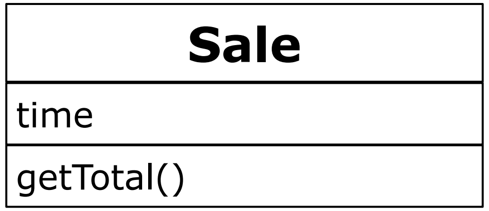
- Example (continue)
  - To calculate **grand total** of a *Sale*, it is necessary to know the instances of *SalesLineItem* and the sub-total of each instance.
  - According to the pattern, *Sale* knows the information



Information Expert pattern

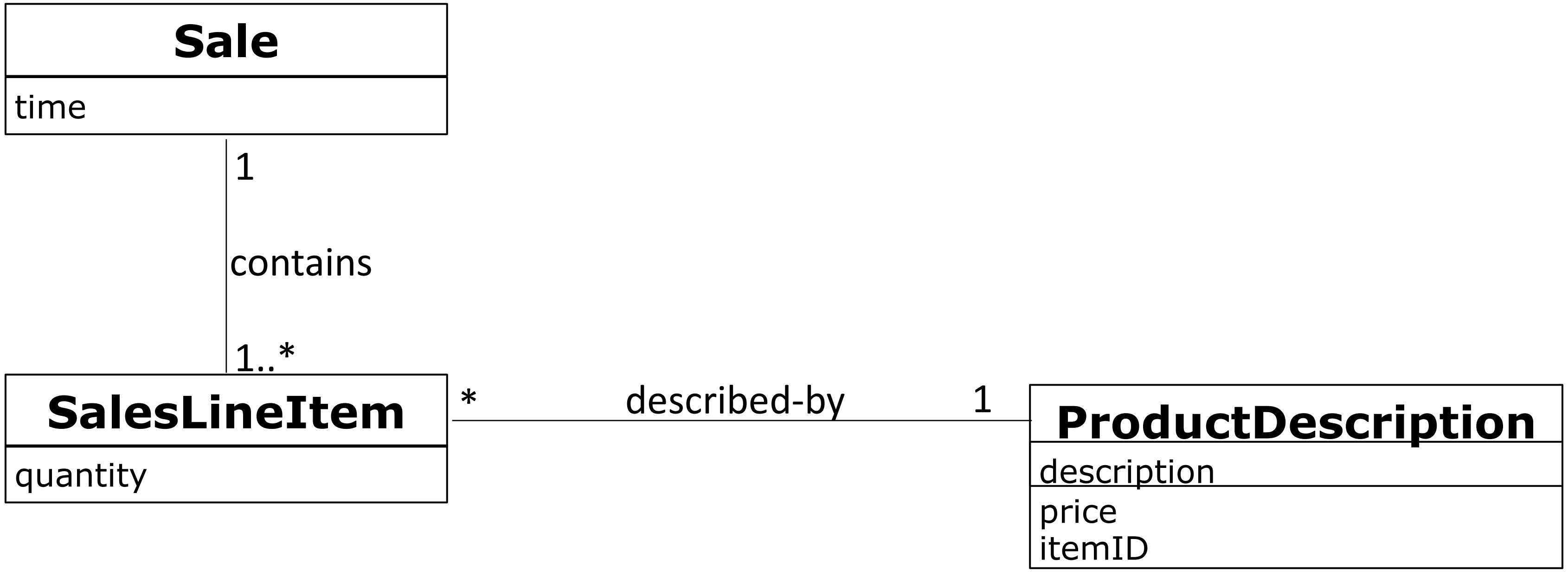
- Example (continue)
  - Introduce “*getTotal()*” method to *Sale* class





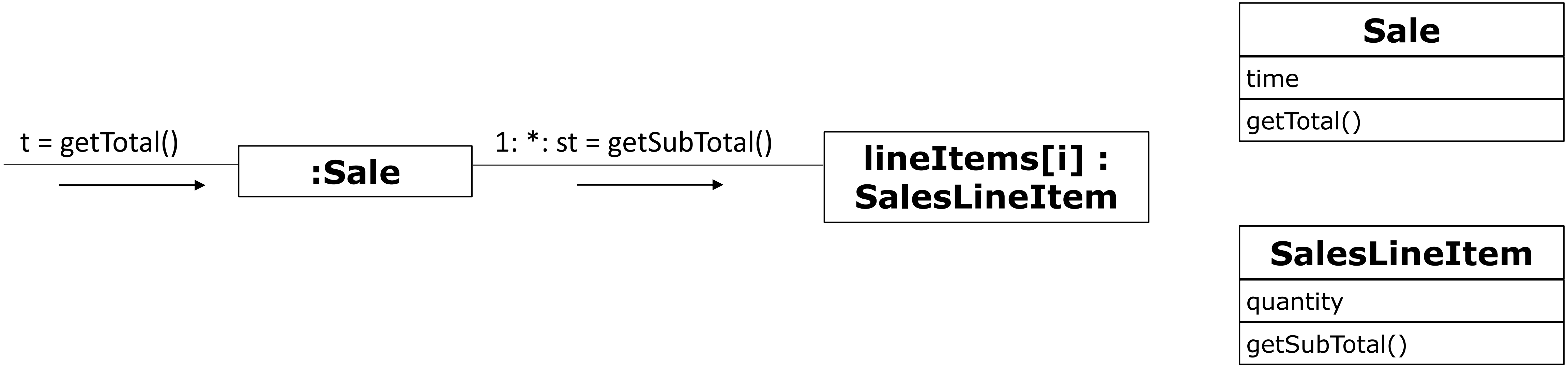
Information Expert pattern

- Example
  - Then, we need to determine the sub-total of each *SalesLineItems*. To do so, we need to know the number of *ProductDescription*
  - According to the pattern, *SalesLineItem* is the expert.



Information Expert pattern

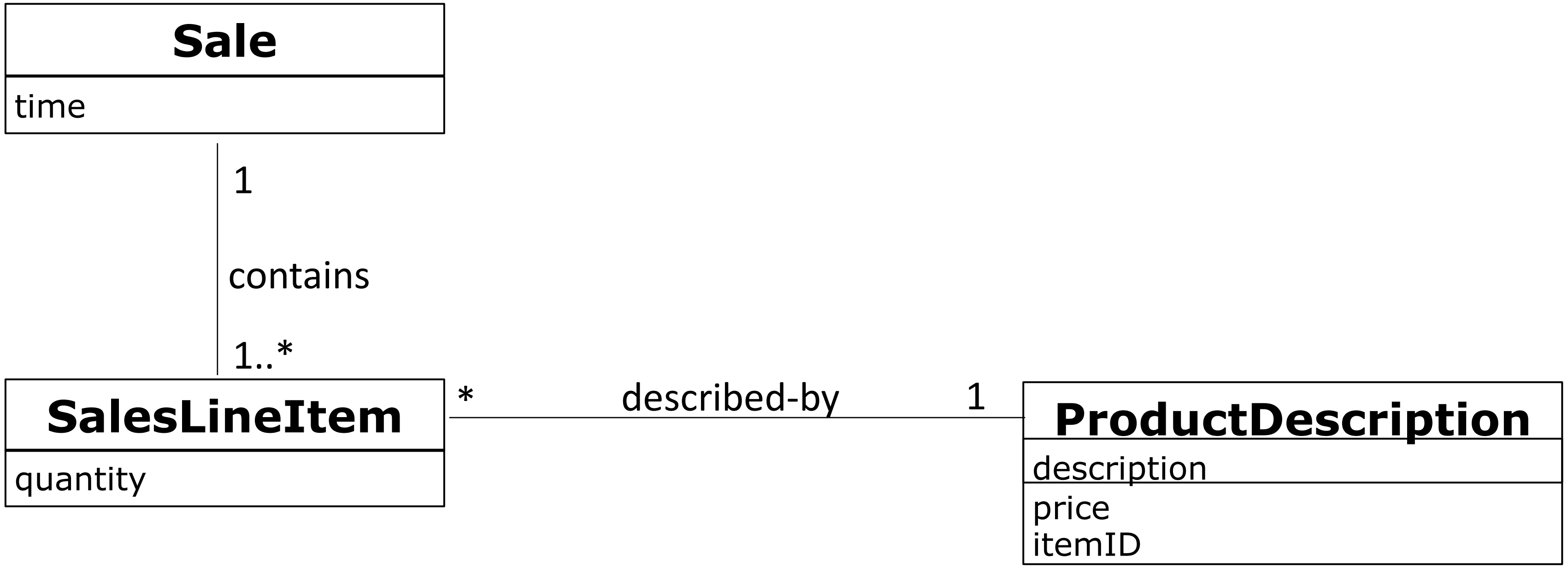
- Example
  - Introduce the “get*SubTotal()*” method to *SalesLineItem* class



]

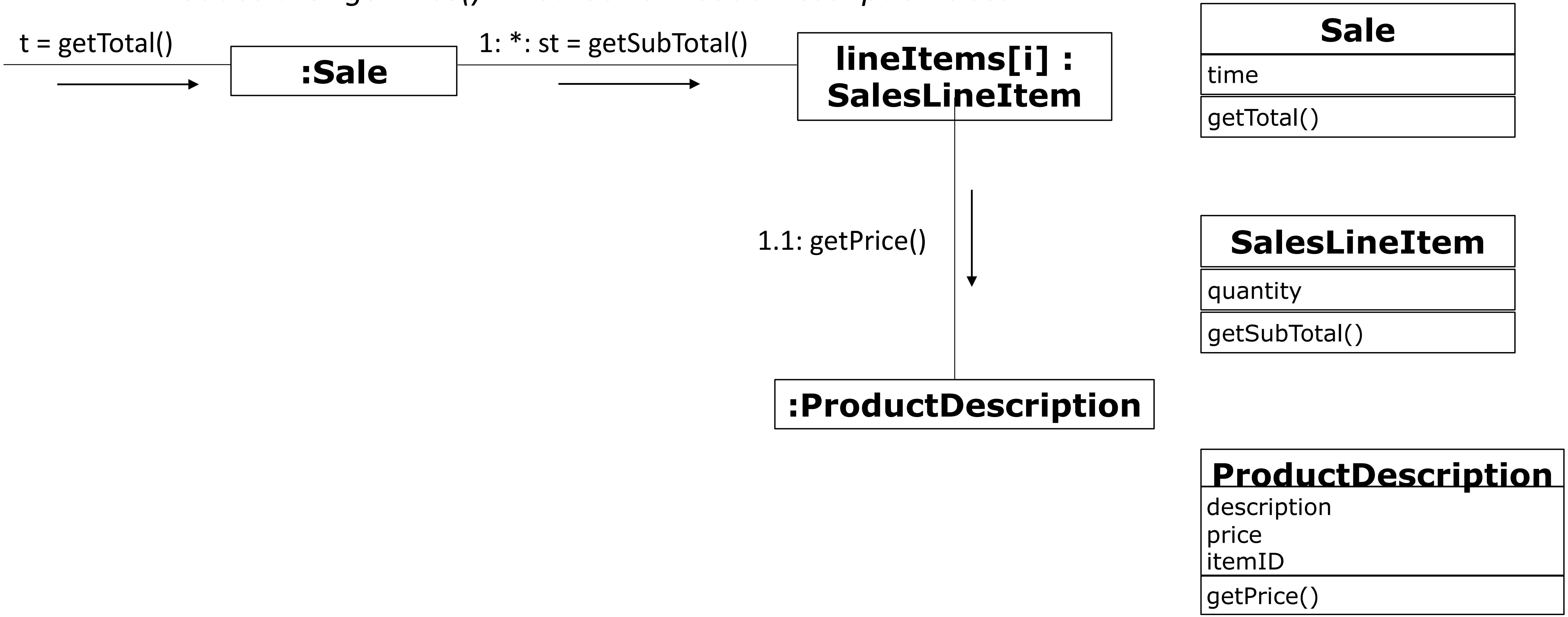
Information Expert pattern

- Example
  - To calculate the sub-total, *SalesLineItem* needs to know the price of each product.
  - *ProductionDescription* est expert.



Information Expert pattern

- Example
  - Introduce the “*getPrice()*” method to *ProductDescription* class

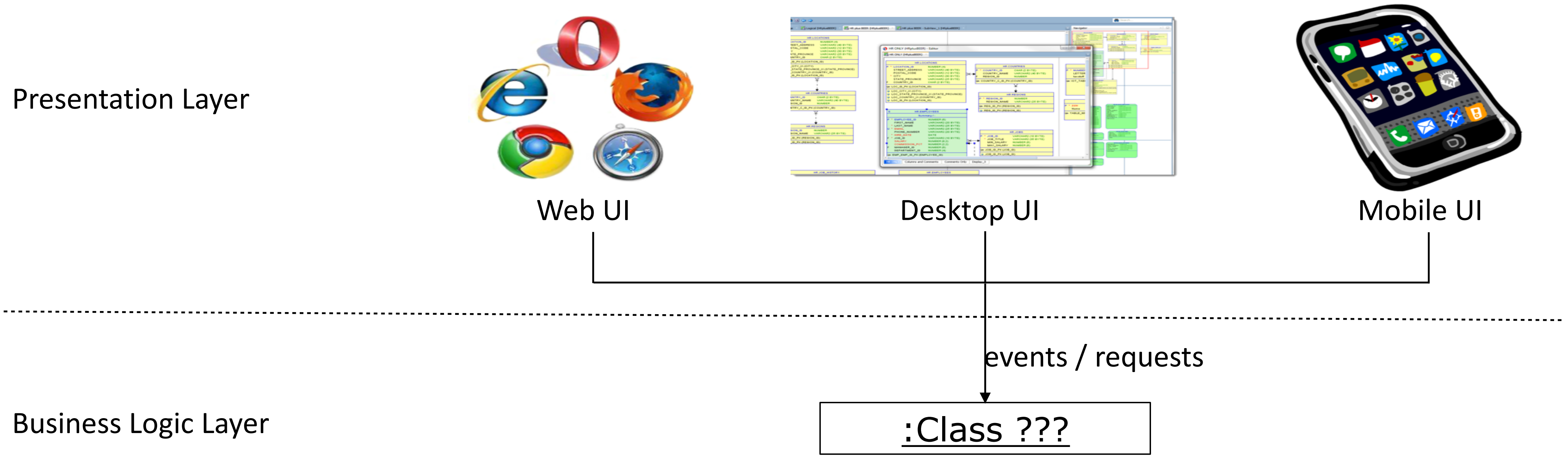


Information Expert pattern

- Advantages
  - The encapsulation is maintained since objects use their own information to satisfy responsibility
  - This pattern supports loose coupling, this allows the system to be more robust and easier to maintain
  - The behaviour is distributed among the classes that possess the necessary information, it encourages more coherent and smaller definitions are easier to understand and maintain

Controller pattern

- Problem
  - Which first object beyond the User Interface (UI) layer receives and coordinates ("controls") a system operation?



Presentation Layer

Web UI

Desktop UI

Mobile UI

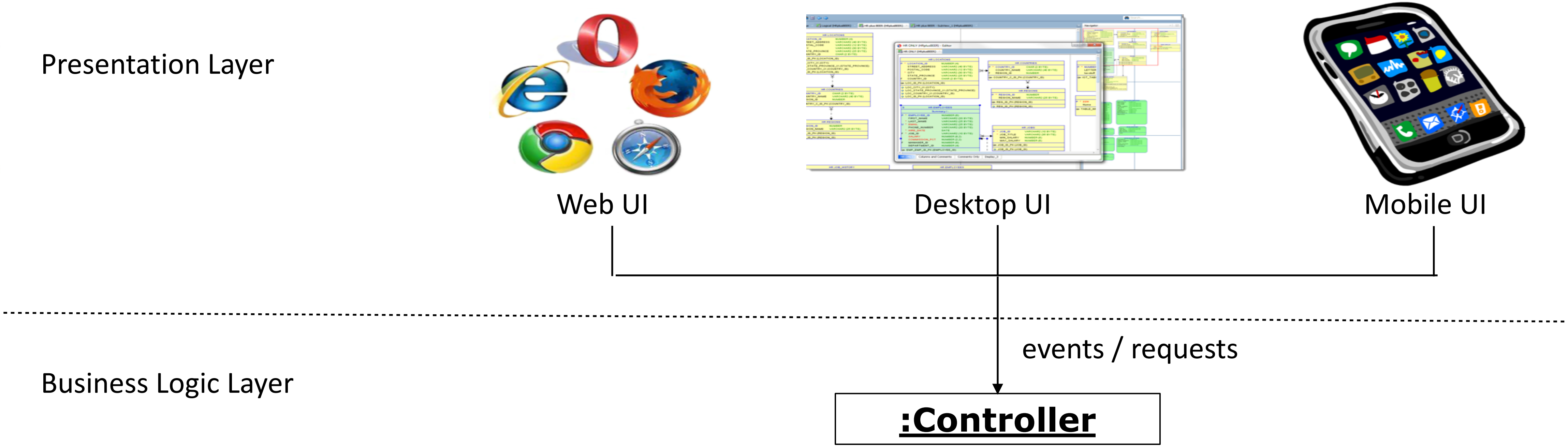
events / requests

Business Logic Layer

:Class ???

Controller pattern

- Solution
  - A **Controller** is the first object beyond the UI layer that is responsible for receiving and handling a system operation.
  - A controller should delegate the work to other objects. The controller only receives the requests but doesn’t not actually solve them.



Presentation Layer

Web UI

Desktop UI

Mobile UI

events / requests

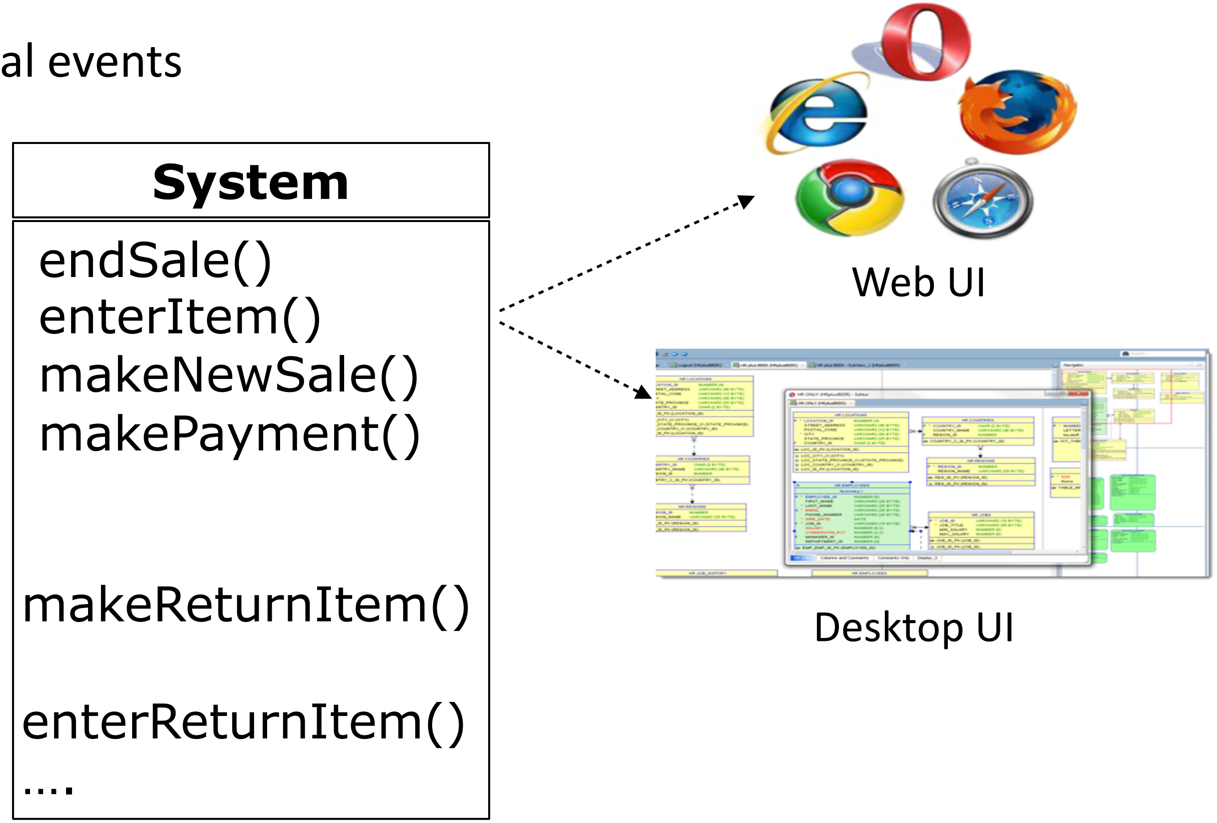
Business Logic Layer

:Controller

Controller pattern

- Application
  - The Controller pattern can be applied to all the systems that need to process external events
  - A controller class is selected to process the events
- Example
  - The Cash Register system has several events

Presentation Layer



System

endSale()
enterItem()
makeNewSale()
makePayment()

makeReturnItem()

enterReturnItem()
....

Web UI

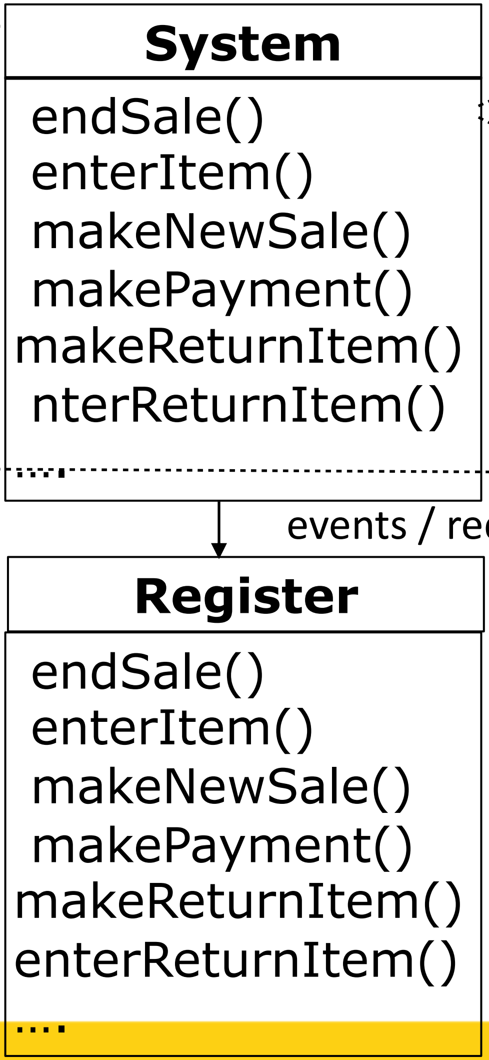
Desktop UI

  - What class can be the controller (i.e., what class processes the events)?

Controller pattern

- Example: Cash Register system
  - Solution 1: use one controller

Presentation Layer



System

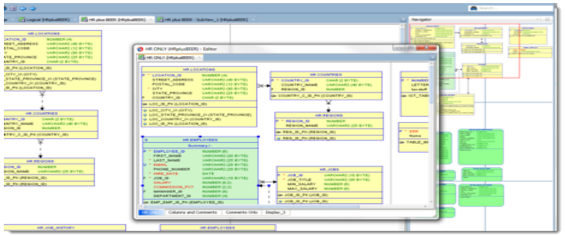
endSale()
enterItem()
makeNewSale()
makePayment()
makeReturnItem()
nterReturnItem()
....

events / requests

Register

endSale()
enterItem()
makeNewSale()
makePayment()
makeReturnItem()
enterReturnItem()
....

Web UI

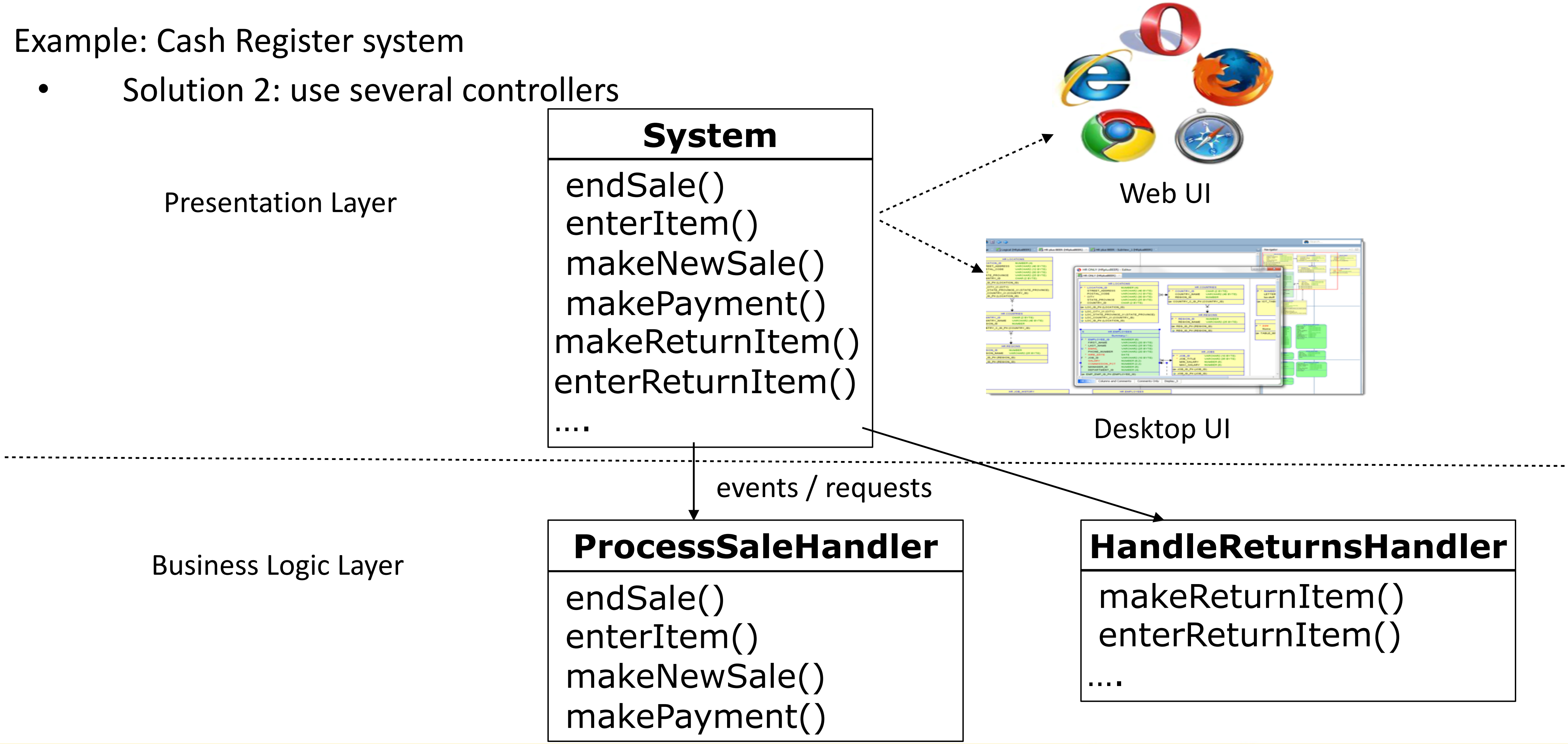


Desktop UI

Business Logic Layer

Controller pattern

- Example: Cash Register system
    - Solution 2: use several controllers



Presentation Layer

System

endSale()
enterItem()
makeNewSale()
makePayment()
makeReturnItem()
enterReturnItem()
....

Web UI

Desktop UI

events / requests

Business Logic Layer

ProcessSaleHandler

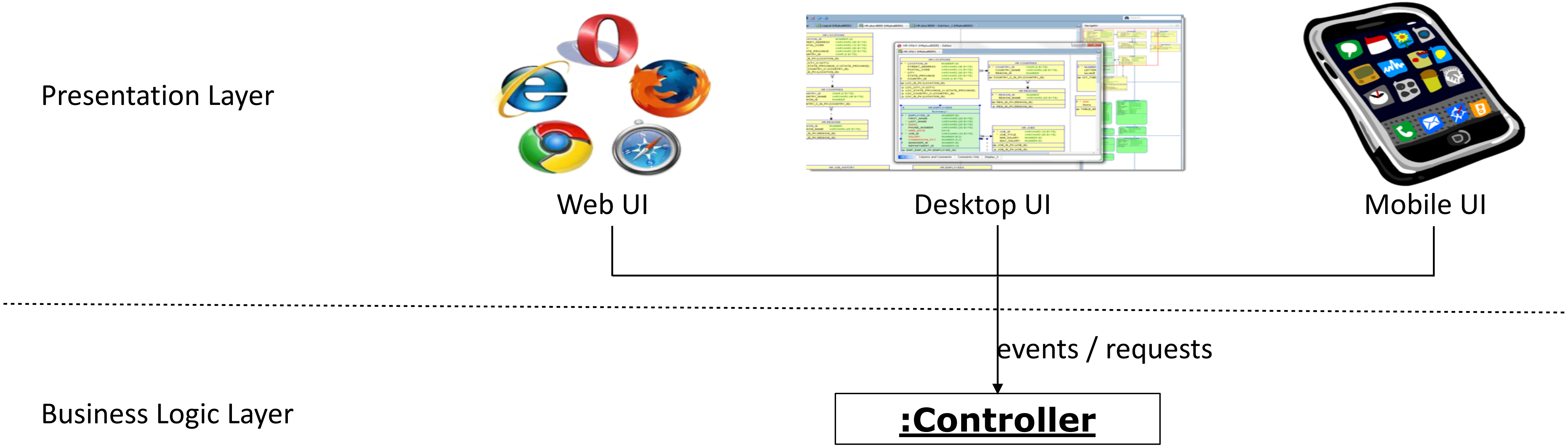
endSale()
enterItem()
makeNewSale()
makePayment()

HandleReturnsHandler

makeReturnItem()
enterReturnItem()
....

Controller pattern

- Discussion
  - Advantages
    - This is simply a delegation pattern - the UI should not contain application logic
    - Increase potential for reuse and pluggable interfaces
    - Creates opportunity to reason about state of a use-case, for example, to ensure that operations occur in a legal sequence.



Presentation Layer

Web UI

Desktop UI

Mobile UI

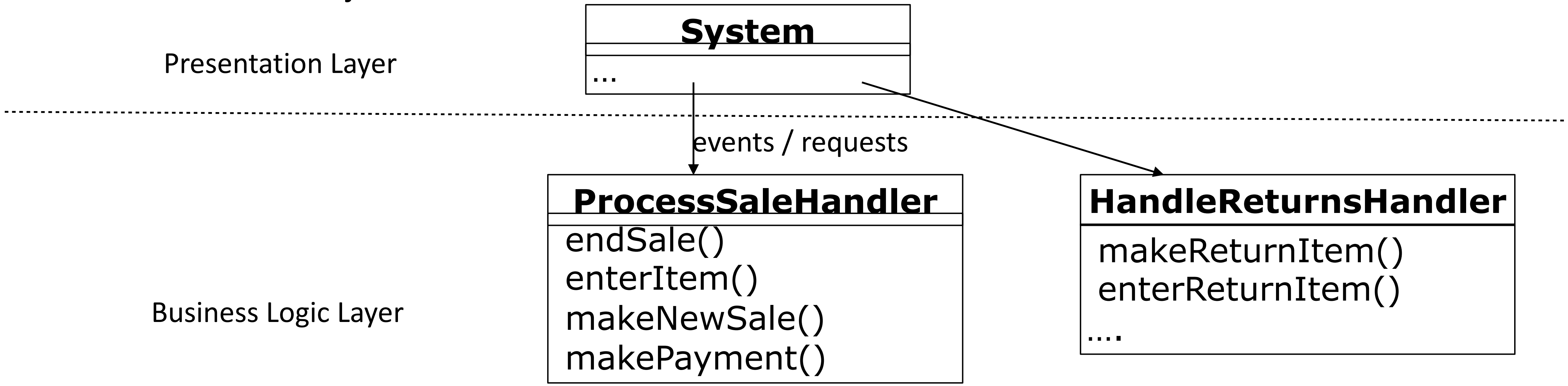
events / requests

Business Logic Layer

:Controller

Controller pattern

- Discussion
  - Difficulty: **Bloated controllers**
    - a single controller that receives all system events, does too much of the work handling events, has to many attributes (duplicating information found elsewhere), etc.
    - Remedies
      - Add more controllers
      - Design controller so that it primarily delegates the fulfilment of each system operation to other objects.



Conclusions

Conclusions

- Distinction between functional approach and object-oriented approach
- Master the basic object-oriented concepts

- UML: a modelling language
  - Need a development process
  - Different views
  - Different models
  - Use of the models in different development activities

- Master the main diagrams
  - Use-case diagram
  - Class diagram
  - Interaction diagram

Conclusions

- The UML concepts can be extended
  - The extensions
- Transformation of models to code
  - Models independent of programming language
- The automatic code generation is only a supplement
  - The models guide the coding process
- Master design principles
  - GRAPS principles/patterns
  - Some design patterns