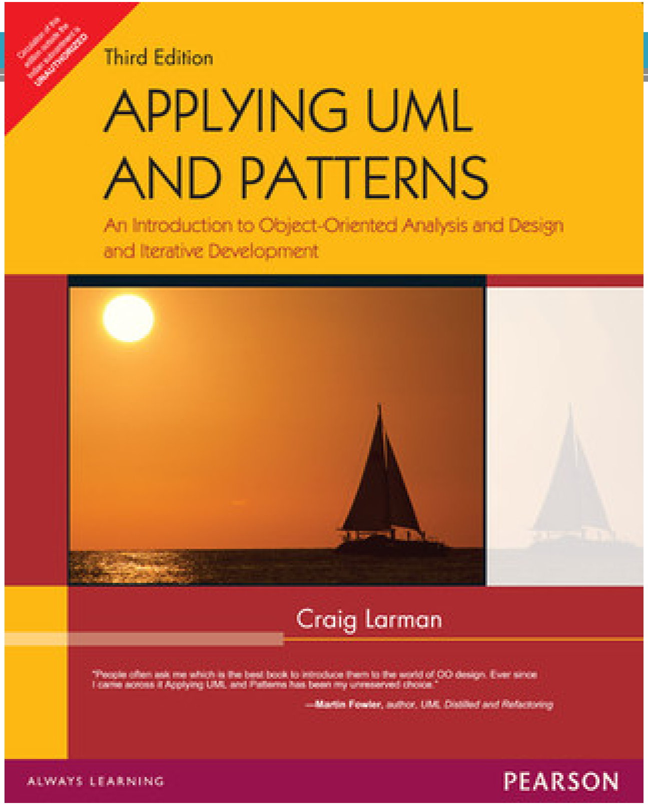
CONCEPTION OBJET GRASP PATTERNS

General Responsibility Assignment Software Patterns

Grasp Patterns

□ Recognize that according to Craig Larman:

1. “The skillful assignment of responsibilities is extremely important in object design,
2. Determining the assignment of responsibilities often occurs during the creation of interaction diagrams and certainly during programming.”


Third Edition
APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and Iterative Development
Craig Larman
"People often ask me which is the best book to introduce them to the world of OO design. Ever since I came across it Applying UML and Patterns has been my unreserved choice."
—Martin Fowler, author, UML Distilled and Refactoring
ALWAYS LEARNING
PEARSON

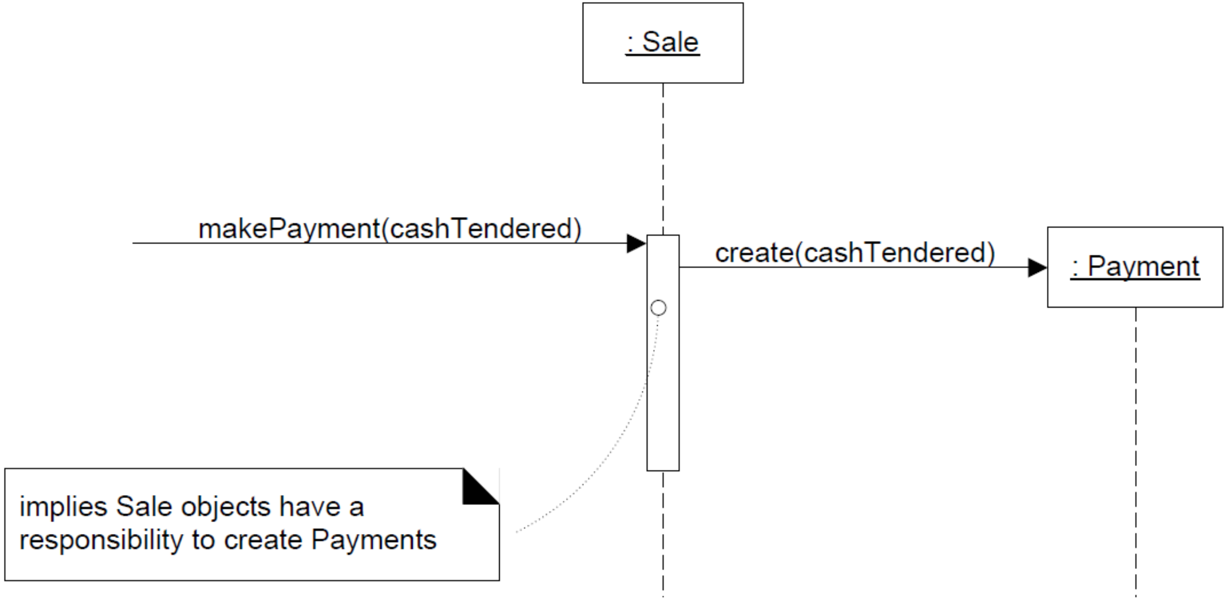
Resources

- www.unf.edu/~broggio/cen6017/38.DesignPatters-Part2.**ppt**
- www.academic.marist.edu/~jzbv/.../Design**Patterns/GRASP.pp**
- **...**

Grasp Patterns

- During Object Design
  - Make choice about the assignment of **responsibilities** to software classes

Responsibility



implies Sale objects have a responsibility to create Payments

Expert Pattern

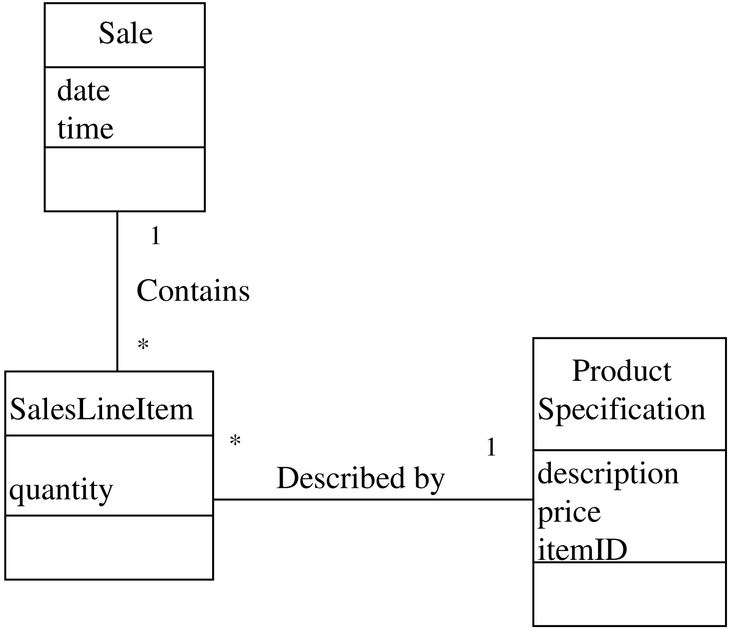
- *Sale* example
- Who is responsible for knowing the total of the sale ?
- Who has the information to determine the total

(Information) Expert Pattern

- Look in the Domain Model
- Domain Model : conceptual classes
- Design Model : software classes
- So
  - Choose a domain model class
  - Create a new class based on Domain Model class

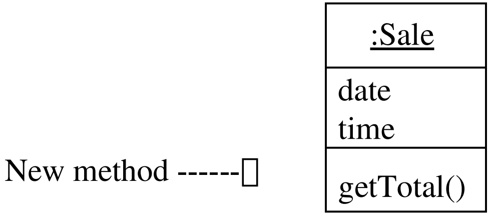
Expert Pattern – Using Domain Model

- There is a Sale class in the domain model



Add Sale Class to the design model

- Add the responsibility of knowing its total
  - Method *getTotal*()

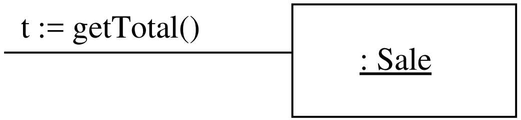


New method ------

:Sale

date
time

getTotal()

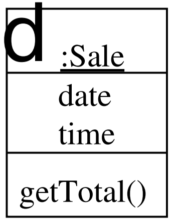


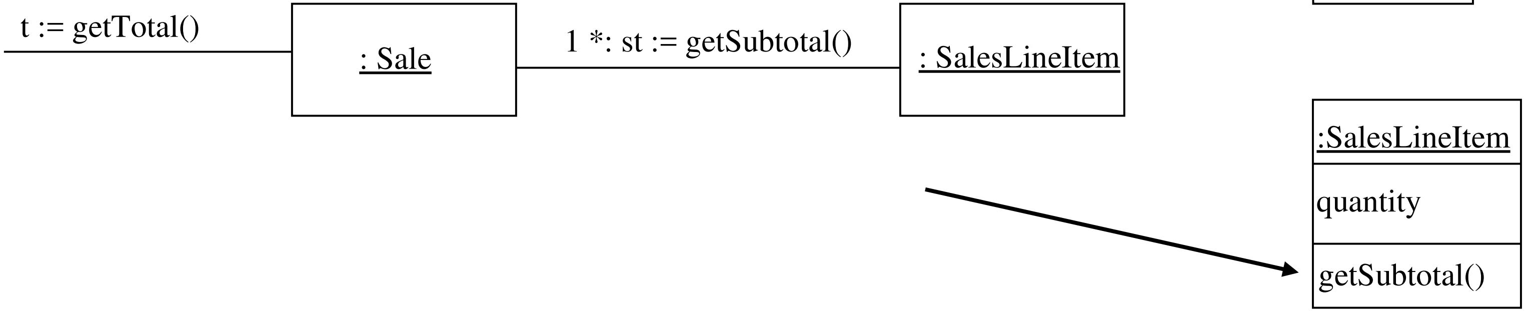
t := getTotal()

: Sale

And then

- What information is needed to determine the line item subtotals?
- We need: *SalesLineItem.quantity* and
- *ProductSpecification.price*



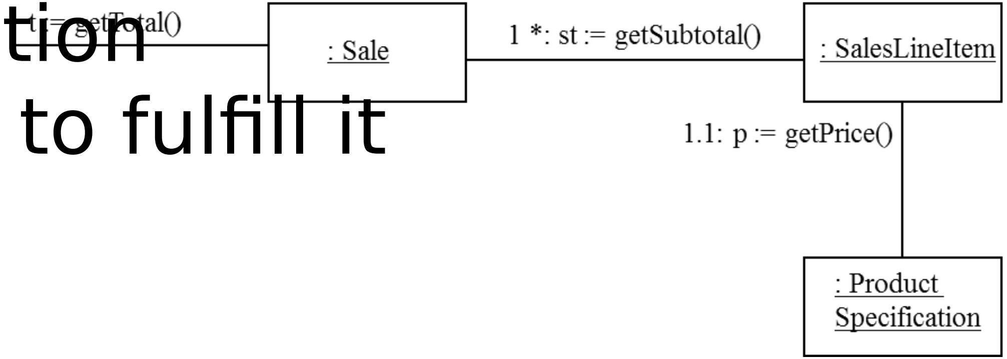


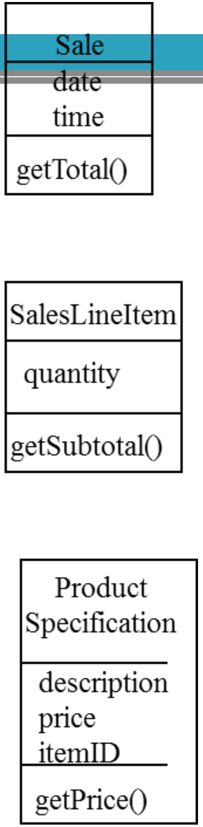
How the domain model is used

- And we need to know the product price
- The design class must include a method getPrice()
- The design classes show how entities are used

Finally

- □ Responsabilities are placed with the object that had the information needed to fulfill it





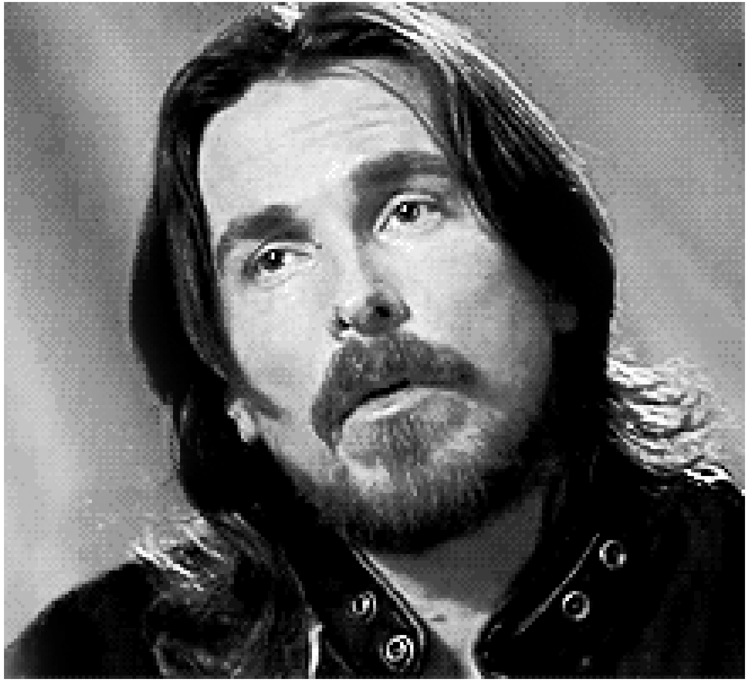
Design Model considerations

- Often requires spanning several classes
- Collaboration between partial information experts
- these “information experts” do things relative to the information they ‘know.’

Be careful

- Who should be responsible for saving Sale in the database ?
- Each entity cannot be responsible for that
- Problem of
  - Cohesion and coupling
  - Reuse and duplication

But why ???



Cohesion and Coupling

- SQL/JDBC Code in the *Sale* Class
- It is not anymore only a sale (**decreased cohesion**)
- This is a new responsibility (saving itself)
- (Separate I/O from data manipulation)

Cohesion and coupling

- Coupling Sale with the database service
- Sale belong to the domain layer
  - Coupled to other domain objects
- Difficult to change the storage service

Final : be careful

- Keep application logic in one place
- Keep database logic in another place
- Separation of concern is good for cohesion and coupling

Benefits of expert

- Maintain encapsulation
- Supports low coupling
- Behavior distributed accross classes that have the required information
- High cohesion, Better reuse

Creator Pattern

- Who is responsible for creating new instances of some classes

Solution

- Assign class B the responsibility to create an instance of class A <u>if one or more</u> of the following is true:
  - B *aggregates* A (simple aggregate; shared attributes)
  - B *contains* A (composition; non-shared attributes)
  - B *records* instances of A objects
  - B *closely uses* A objects
  - B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A)

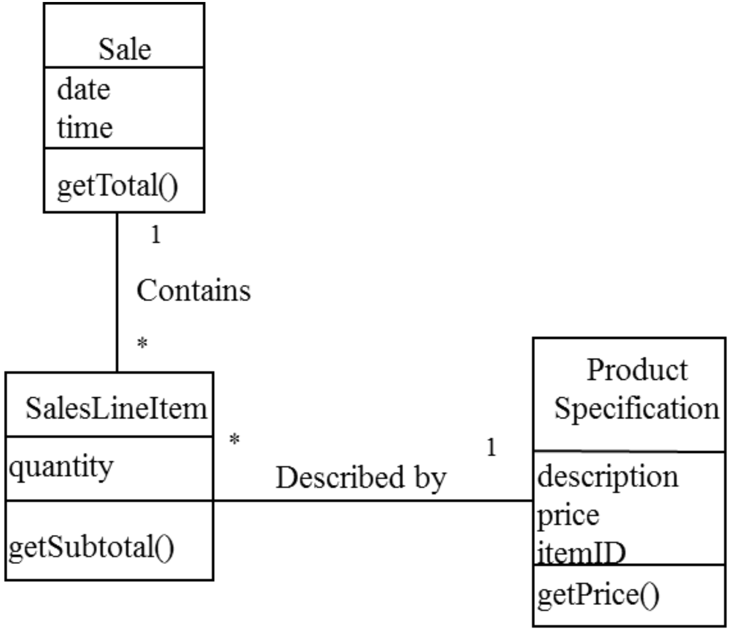
  e.g. queue collection class; queue driver class; stack ….
- If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

Creator

- Creation of objects is very common
  - We have a State class and we create instances of State objects, or
  - We have a CD class, and we create instances (an array?) of CD objects….
- Creator results in low coupling, increased clarity, encapsulation and reusability

Creator Example

- Who is responsible for creating SalesLineItem

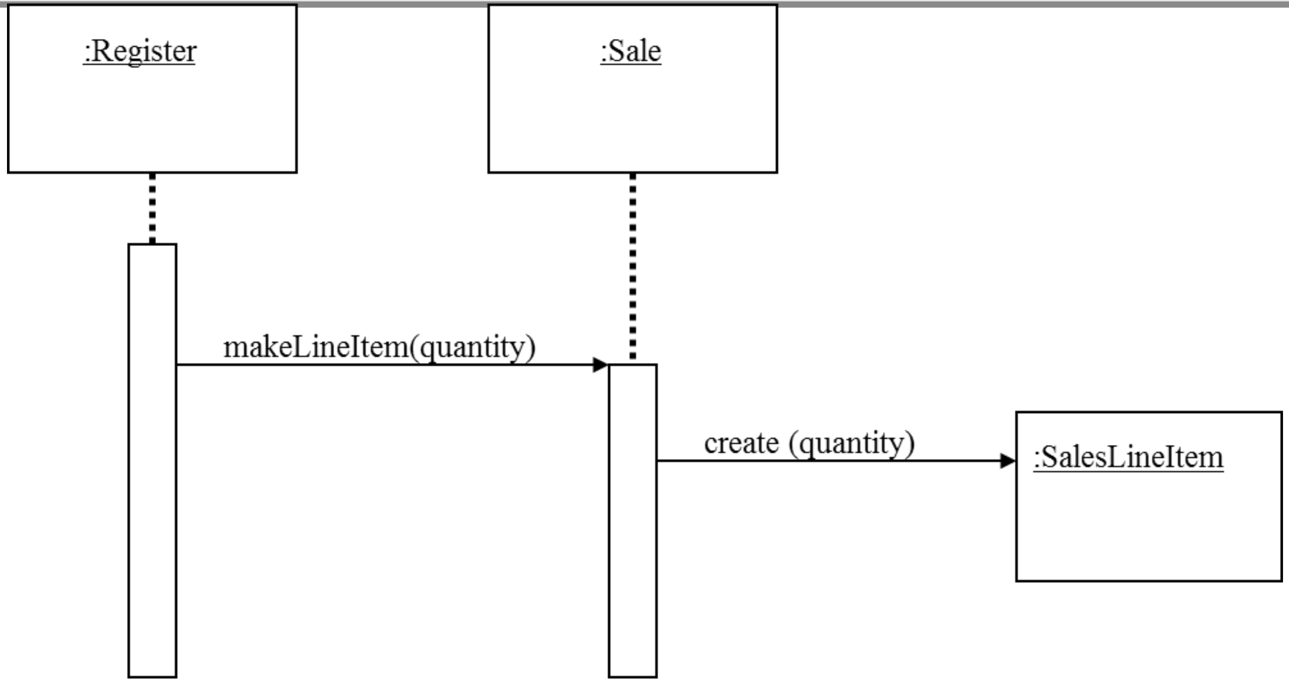


Sale aggregates SalesLineItems

- Sale is a good candidate to have the responsibility of creating SalesLineItems

- Seems very obvious

The sequence diagram helps



Benefits

- Creator connected to the created object
- Creator has the initializing data needed for the creation
- Cf Larman book

- Creator is a kind of expert

Creator Pattern

- Sometimes it is better to delegate creation to a helper Class

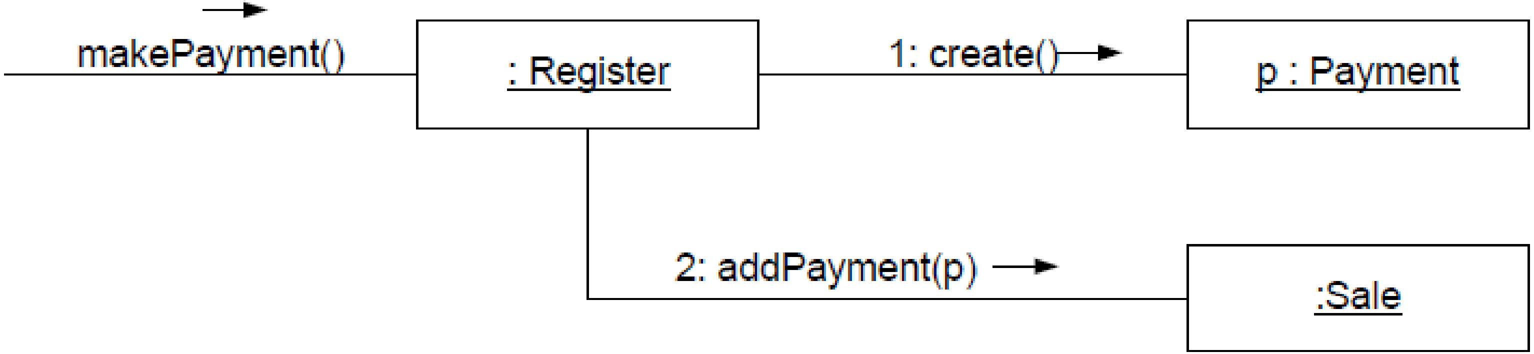
- The Factory pattern

Low Coupling

- Assign a responsibility to keep the coupling low
- Support low dependency, low change impact and increased use
- High coupling is not desirable
  - Hard to change, understand, reuse

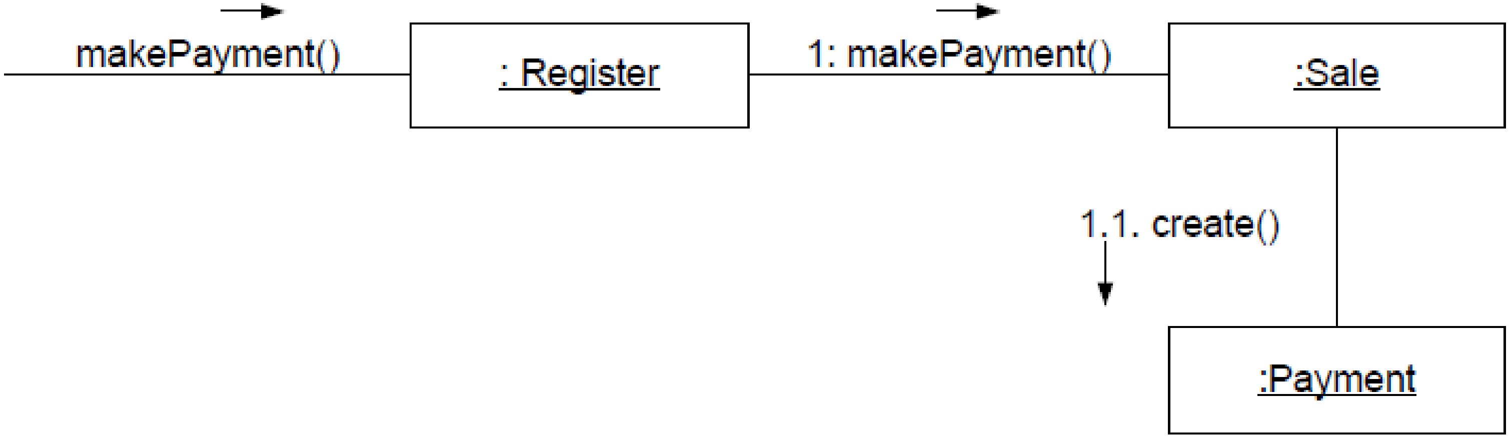
Example

- Register is coupled to payment



Alternative

- Payment known from Sale. Sale has to know Payment



Common form of coupling

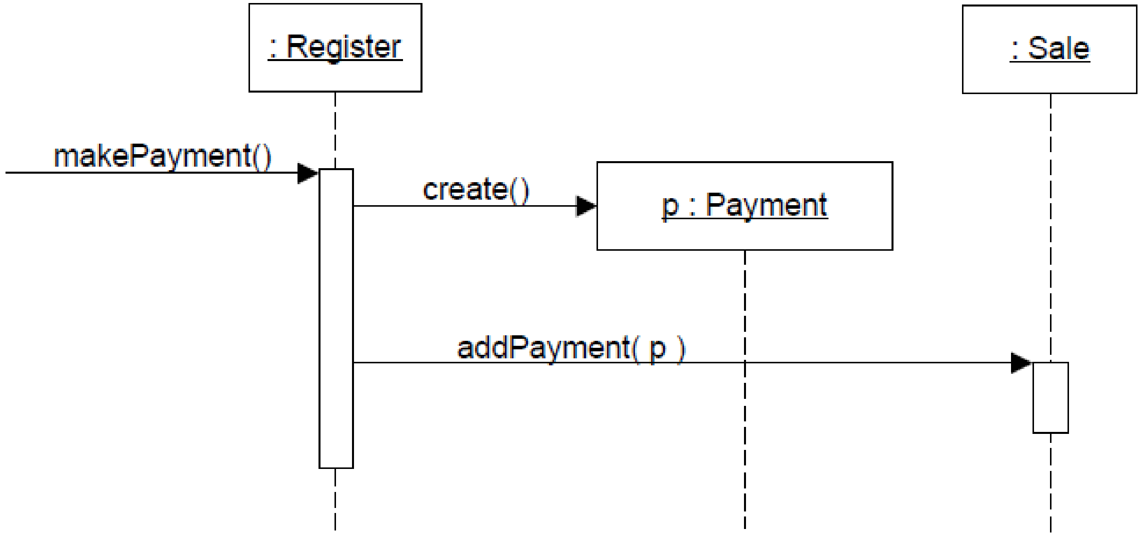
- TypeX has an attribute that refers to TypeY
- TypeX instance call a service on a TypeY instance
- TypeX has a method that references an instance of TypeY (parameter, local variable)
- TypeX is a subclass of TypeY

High Cohesion

- Assign responsibility to keep cohesion high
- Measure of the relation between an element responsibilities
- Low cohesion mean
  - Hard to comprehend, reuse and maintain

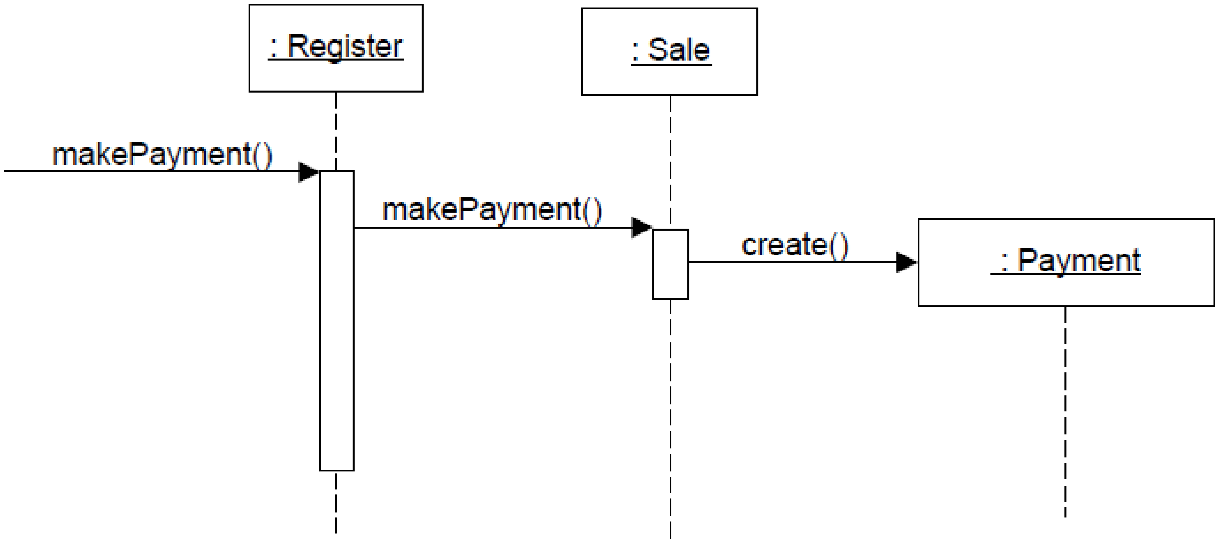
Example

- Register creates payment

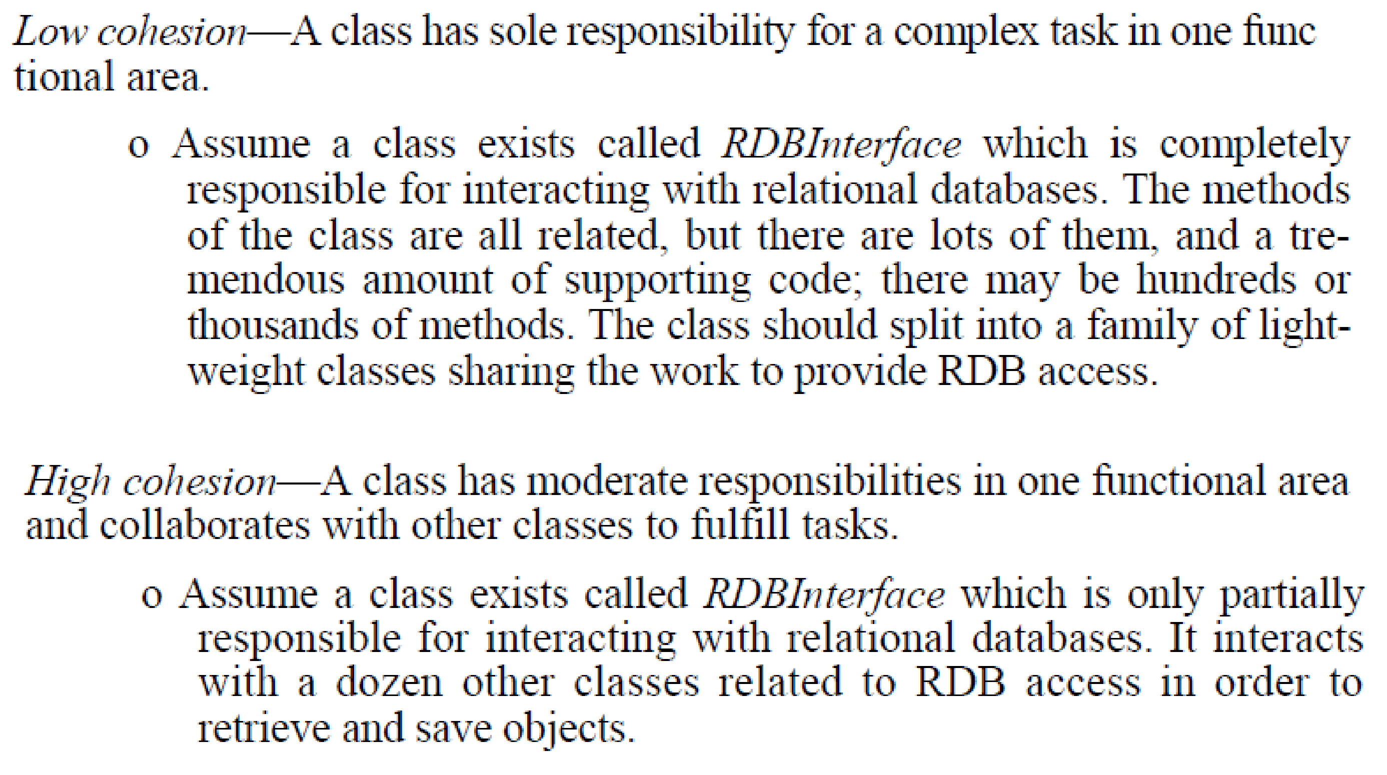


Same alternative

- Register has less responsibilities – Higher co



Scenarios (Booch94)



Low cohesion—A class has sole responsibility for a complex task in one functional area.

- Assume a class exists called *RDBInterface* which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods. The class should split into a family of lightweight classes sharing the work to provide RDB access.

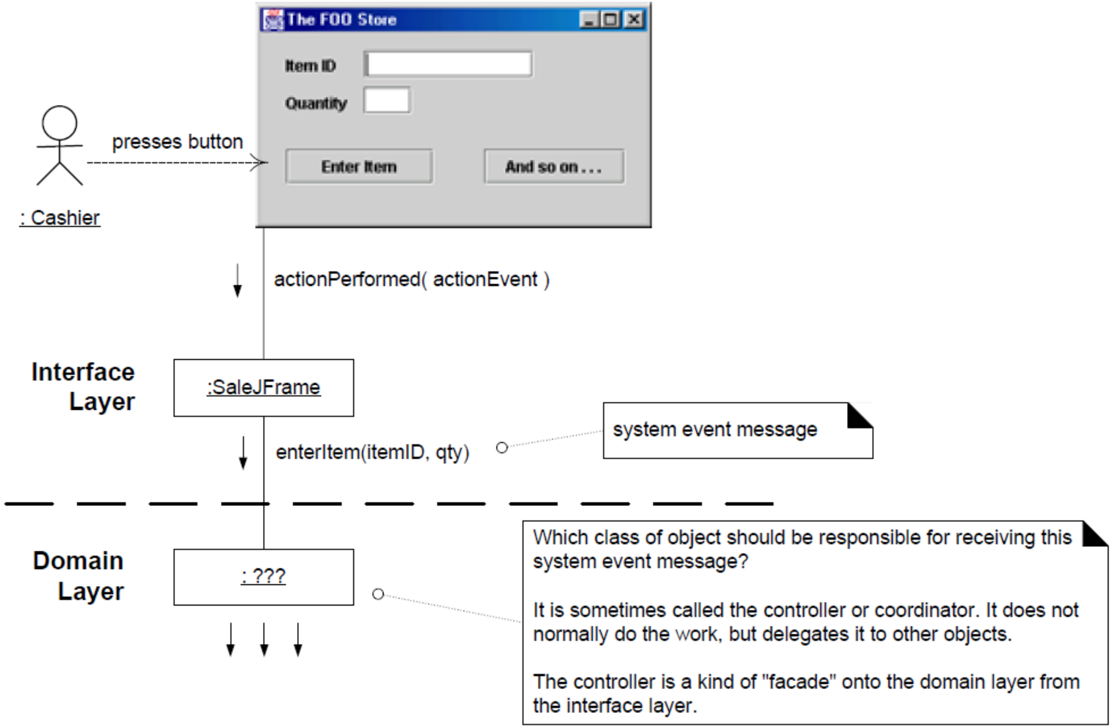
High cohesion—A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

- Assume a class exists called *RDBInterface* which is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.

Controller

- Assign the responsibility for handling event message
  - Facade Controller
  - Use Case or Session controler
- This is not a UI class
- Who is responsible for handling input system event

Example



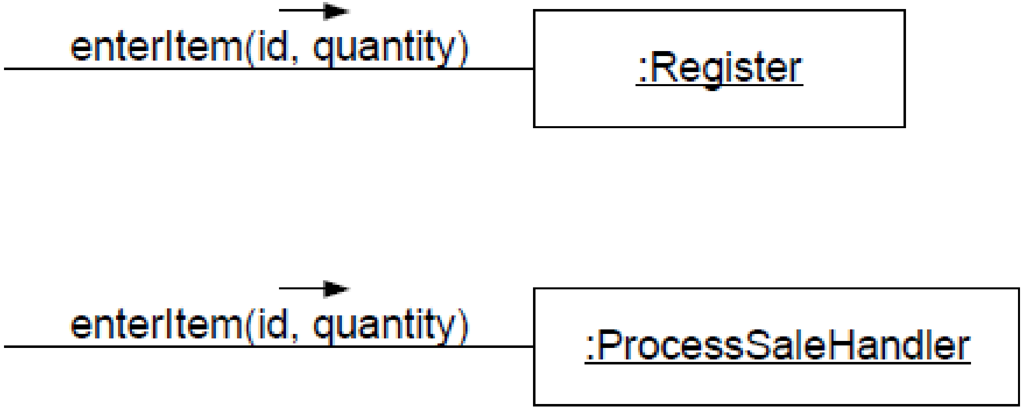
system event message

Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

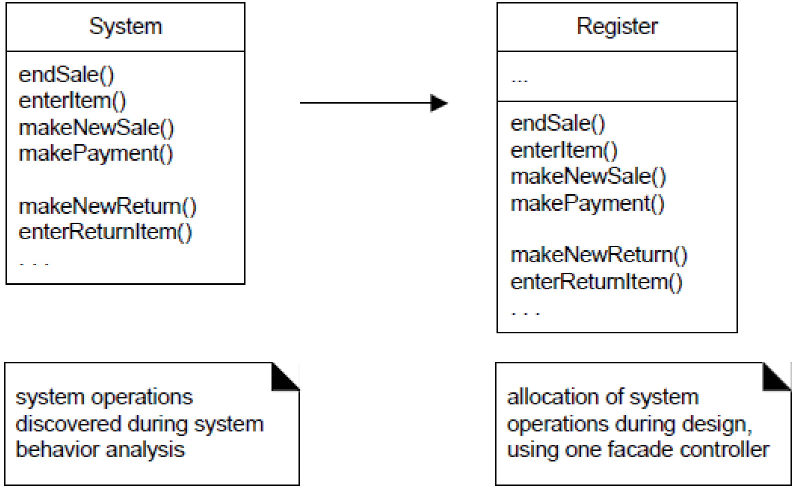
Two possibilities



The controller delegates

- It does not do the work by itself
- It coordinates/controls the activity

Allocation of operations



System

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

makeNewReturn()
enterReturnItem()
. . .

Register

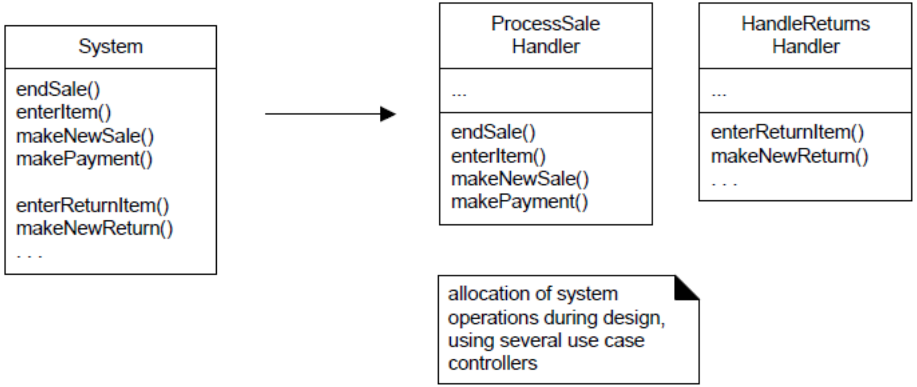
...

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

makeNewReturn()
enterReturnItem()
. . .

system operations discovered during system behavior analysis

allocation of system operations during design, using one facade controller



System

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

enterReturnItem()
makeNewReturn()
. . .

ProcessSale Handler

...

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

HandleReturns Handler

...

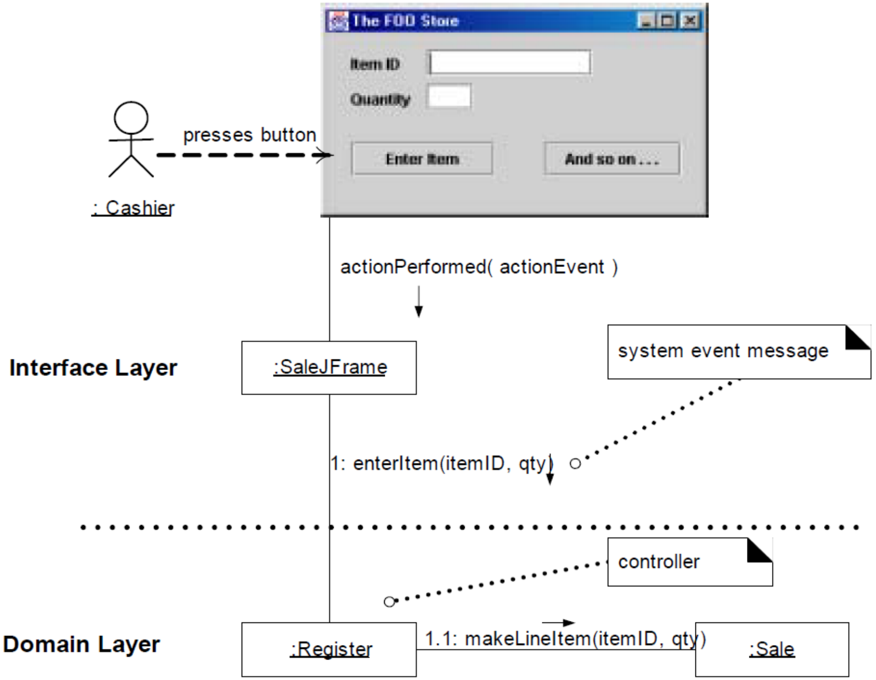
enterReturnItem()
makeNewReturn()
. . .

allocation of system operations during design, using several use case controllers

Issues

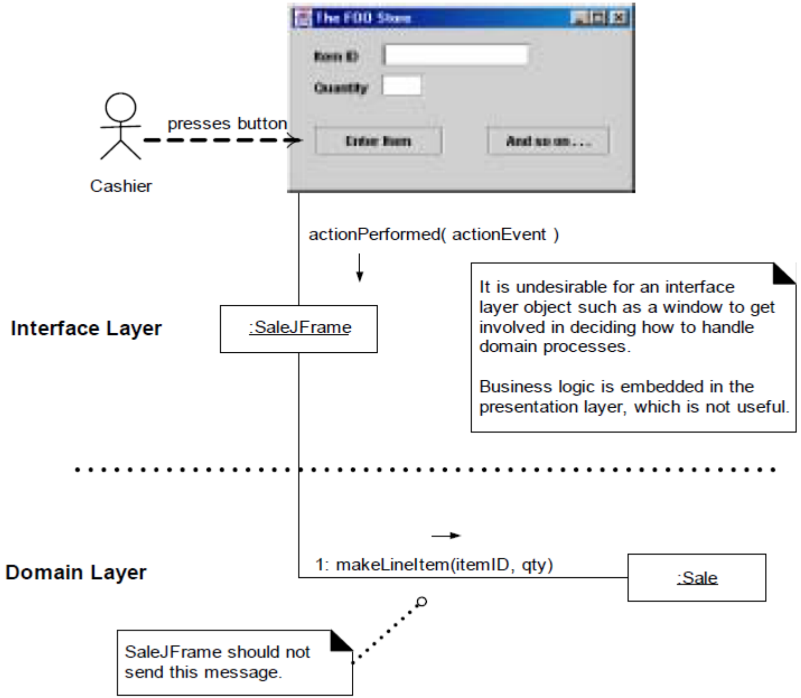
- Avoid bloated controllers (low cohesion)
  - Add more controllers
  - The controller delegates the responsibility to fulfill operation on to other objects.

Two couples



system event message

controller

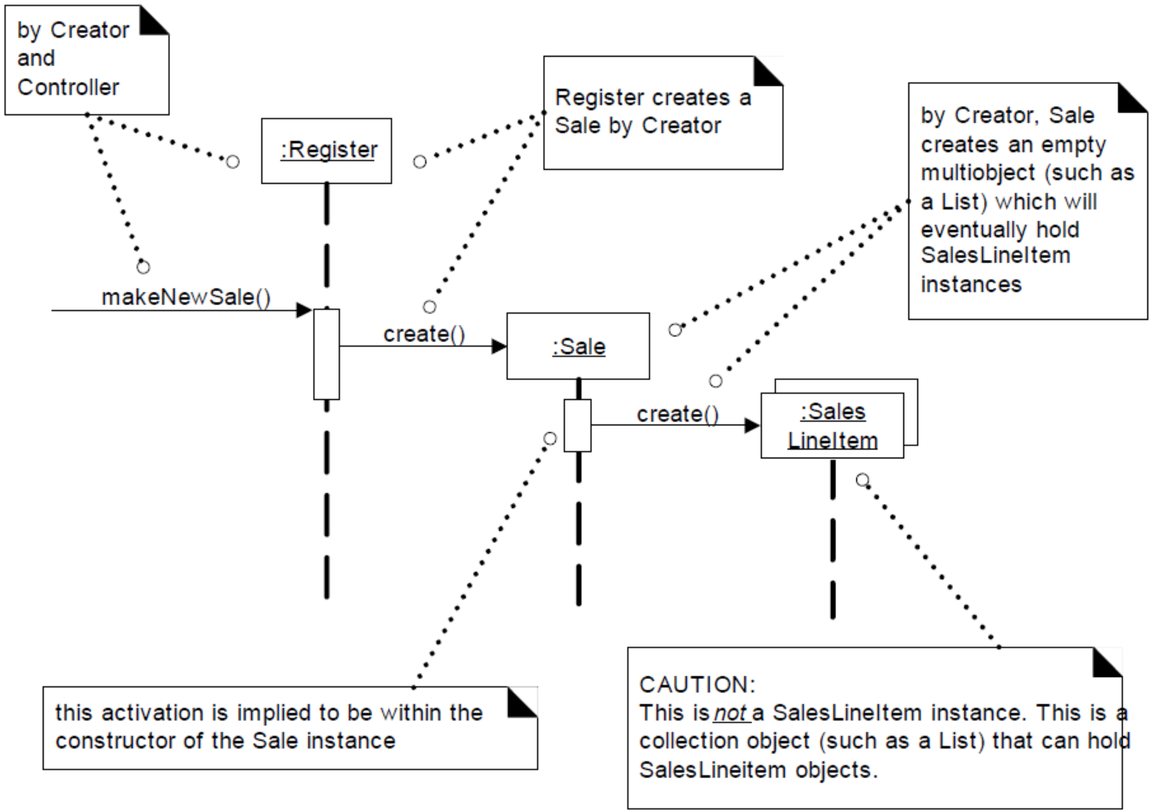


It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.

Business logic is embedded in the presentation layer, which is not useful.

SaleJFrame should not send this message.

Creating a Sale



by Creator and Controller

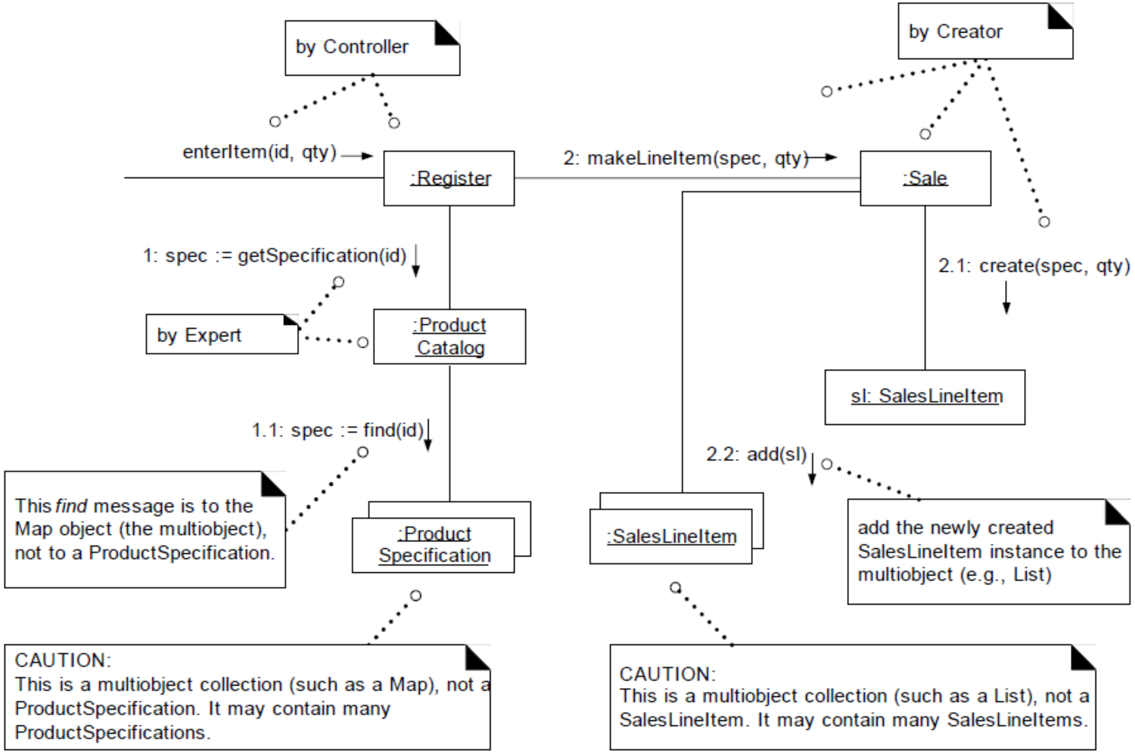
Register creates a Sale by Creator

by Creator, Sale creates an empty multiobject (such as a List) which will eventually hold SalesLineItem instances

this activation is implied to be within the constructor of the Sale instance

CAUTION:
This is *not* a SalesLineItem instance. This is a collection object (such as a List) that can hold SalesLineitem objects.

Enter an Item to the Sale



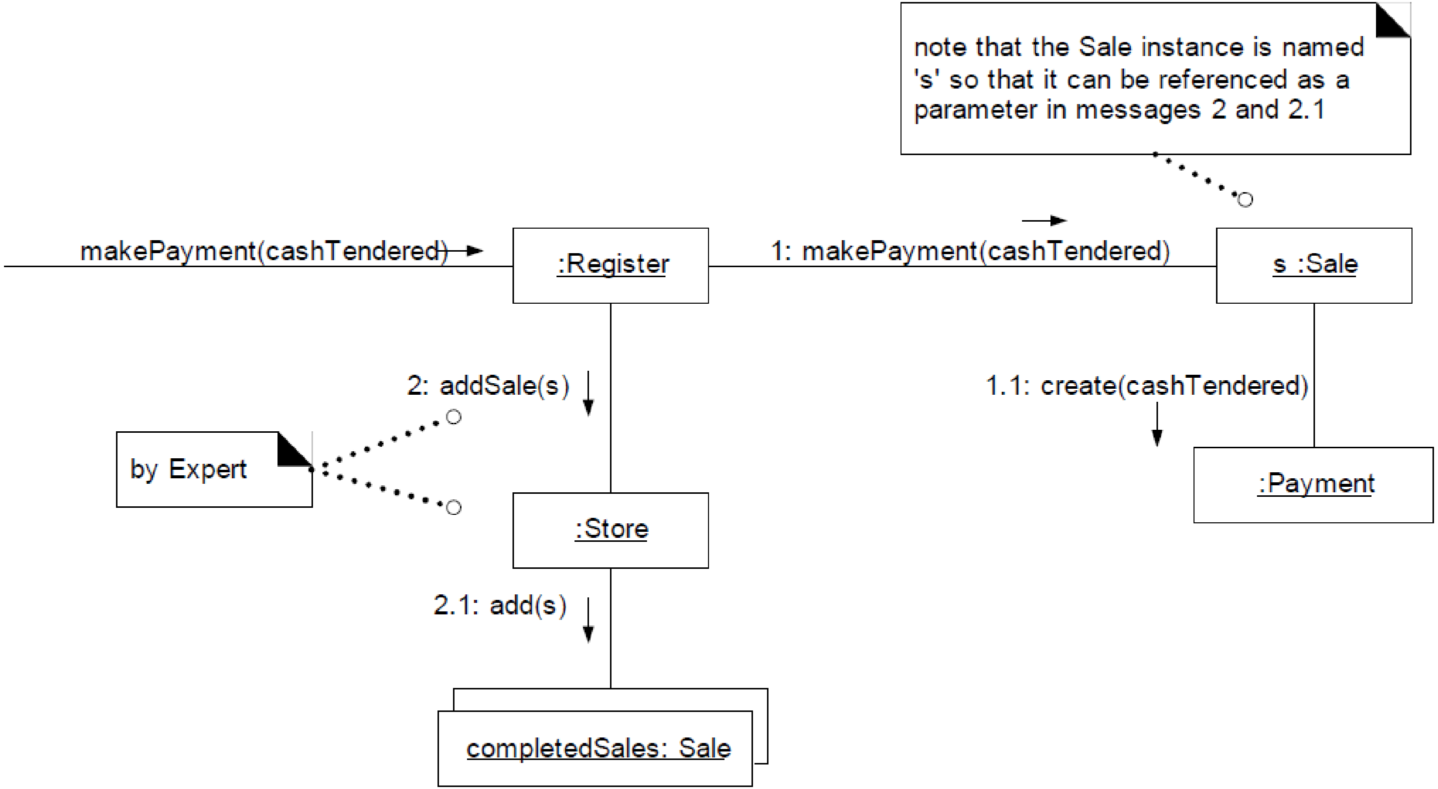
This *find* message is to the Map object (the multiobject), not to a ProductSpecification.

add the newly created SalesLineItem instance to the multiobject (e.g., List)

CAUTION:
This is a multiobject collection (such as a Map), not a ProductSpecification. It may contain many ProductSpecifications.

CAUTION:
This is a multiobject collection (such as a List), not a SalesLineItem. It may contain many SalesLineItems.

Making payment

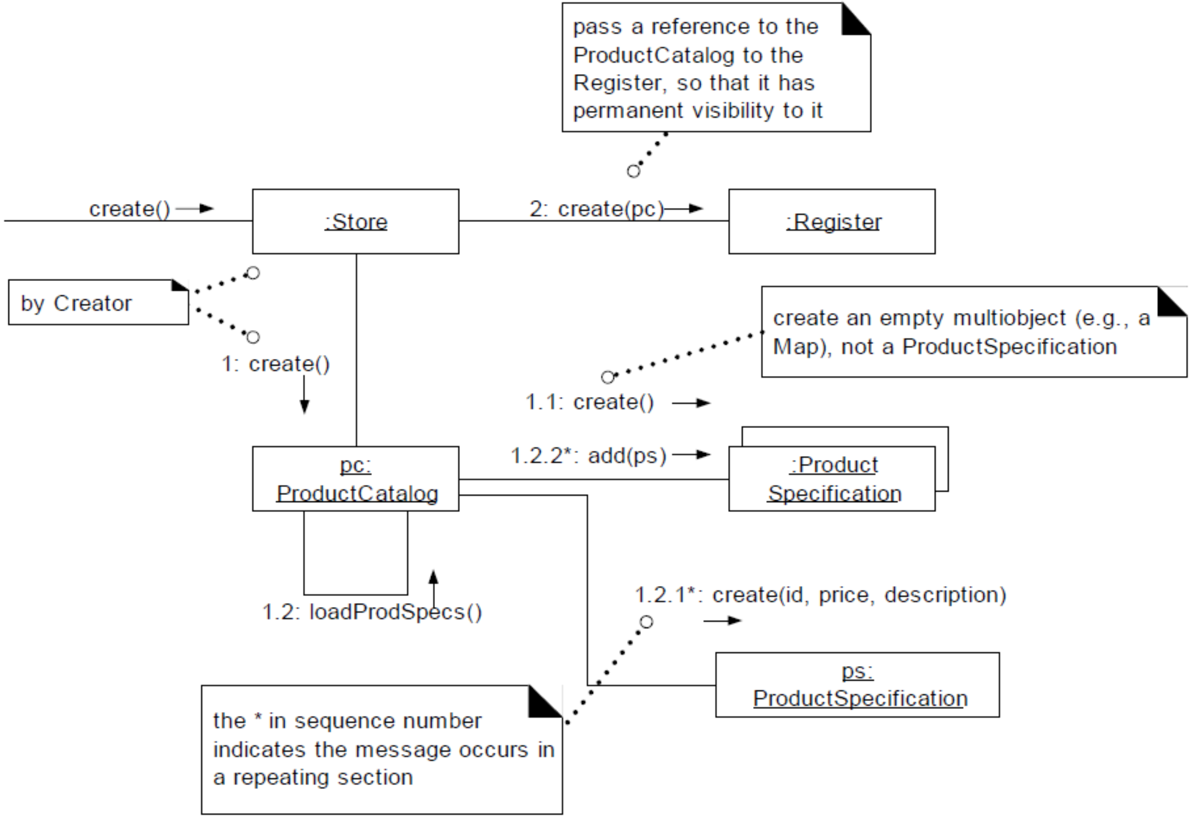


note that the Sale instance is named 's' so that it can be referenced as a parameter in messages 2 and 2.1

by Expert

makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
s :Sale
2: addSale(s)
1.1: create(cashTendered)
:Payment
:Store
2.1: add(s)
completedSales: Sale

Initialisation


pass a reference to the ProductCatalog to the Register, so that it has permanent visibility to it
create()
:Store
2: create(pc)
:Register
by Creator
1: create()
create an empty multiobject (e.g., a Map), not a ProductSpecification
1.1: create()
1.2.2*: add(ps)
pc: ProductCatalog
:Product Specification
1.2: loadProdSpecs()
1.2.1*: create(id, price, description)
ps: ProductSpecification
the * in sequence number indicates the message occurs in a repeating section

Remember

- Low Coupling/High Cohes
- Expert
- Creator
- Controller
- Not exactly patterns but strong guidelines.

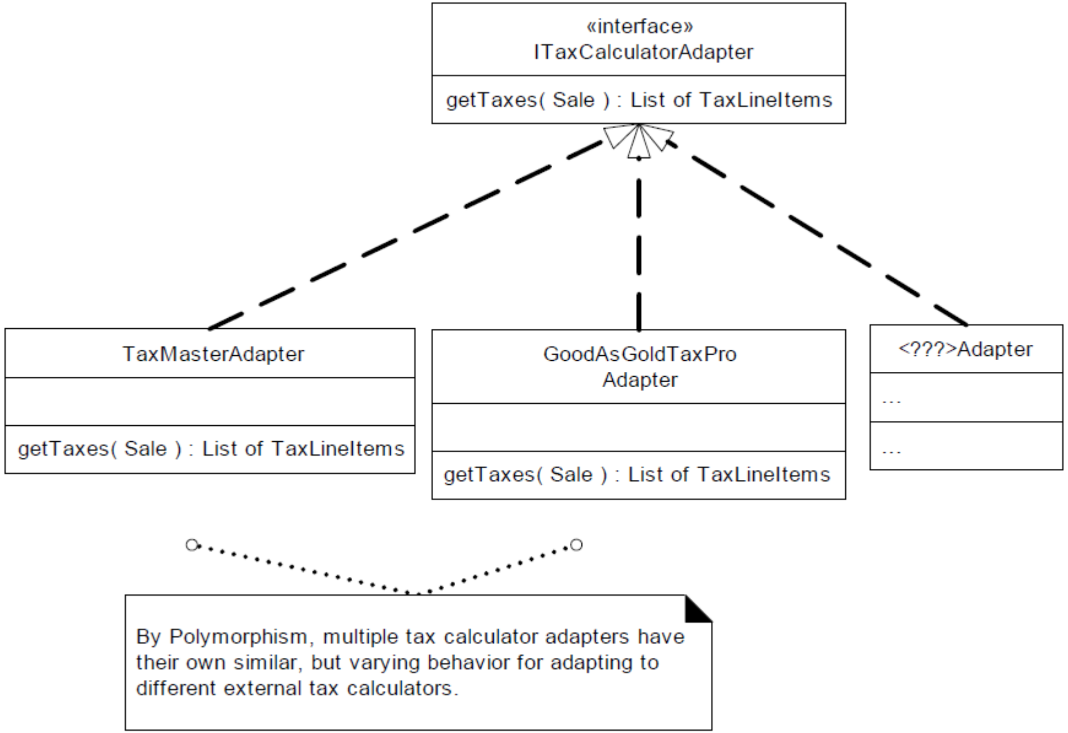
More patterns (or principles)

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variation

Polymorphism

- When behavior vary by type assign the responsibility to the type for which the type vary.
  - *Corollary:* Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.
- How to create pluggable component ? How to handle alternatives based on types ?

Example : Multiple tax calculator


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...

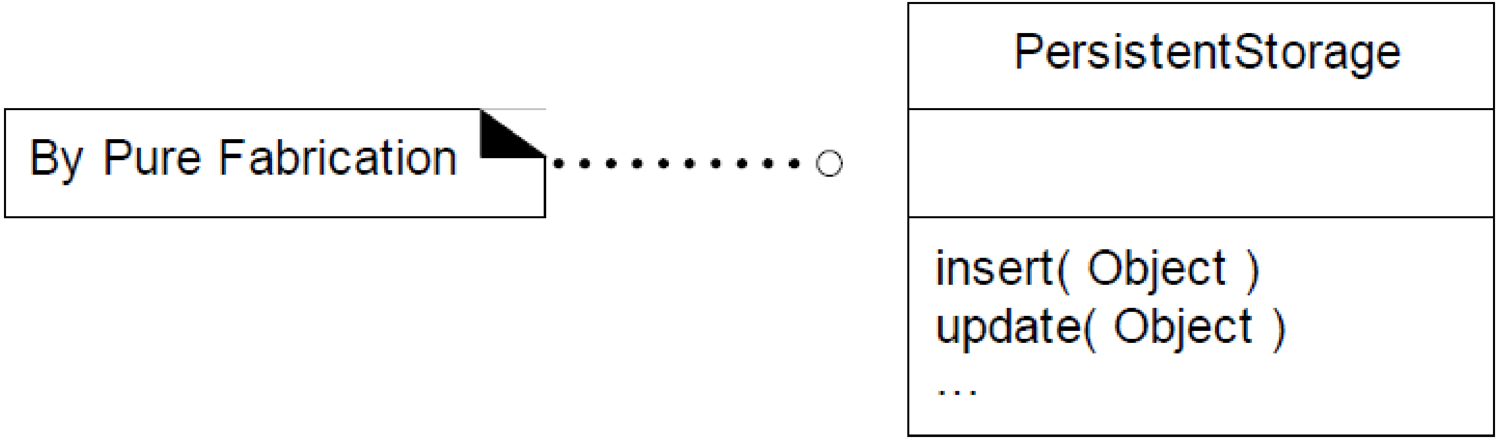
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Final

- Very easy to extend and add variations
- New implementations can be added with affecting the client
- Do it only if there are known variations (no future proofing)

Pure Fabrication

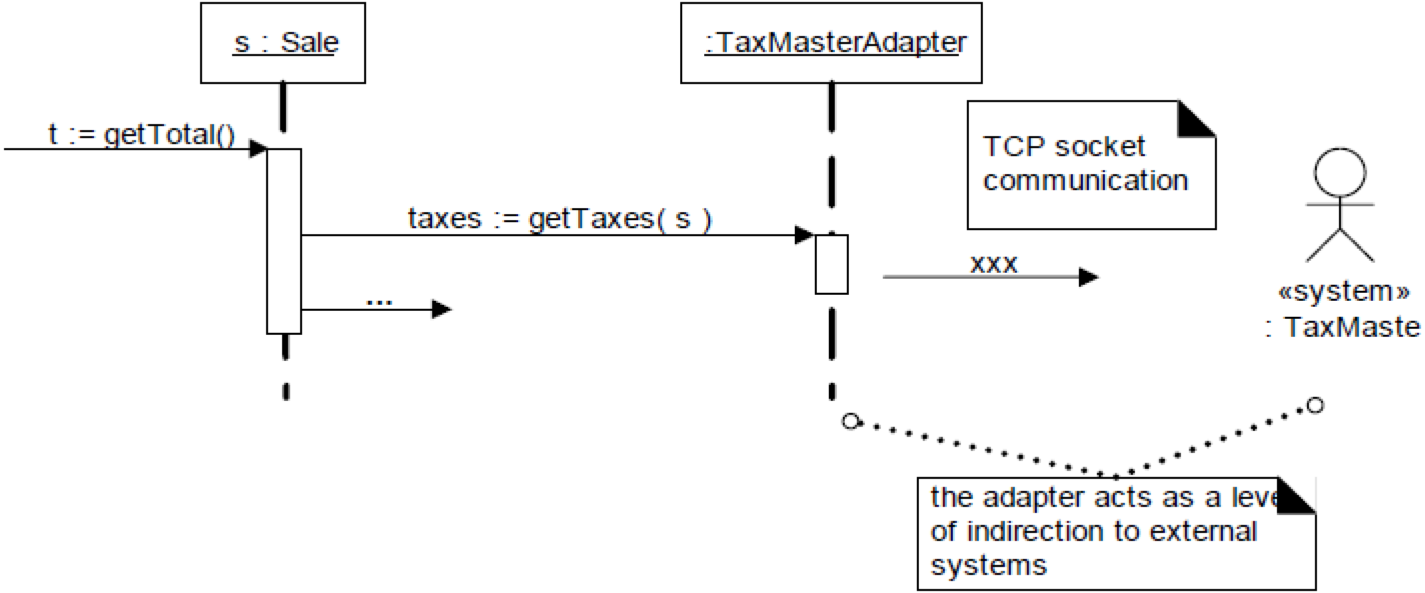
- A class to save cohesion and coupling – a creation of imagination



Indirection

- Assign the responsibility to an intermediate object to mediate between component or services so that they are not directly coupled
- How to decouple objects to increase reuse.

Example : an adapter



TCP socket communication

the adapter acts as a leve of indirection to external systems

Finally

- Reduce coupling
- Protect from variations
- Indirections are often Pure Fabrication
    - PersistenceStorage

Protected Variation

- Identify points of predicted variation and instability. Assign responsibilities to create a stable interface around them