Software Design
SOLID and GRASP

What is a good software design ?

TINHIWHDI principle

- Not just one solution to a design problem

- How do you decide for the most appropriate ?

Principles and patterns

OOP and SOLID Principles

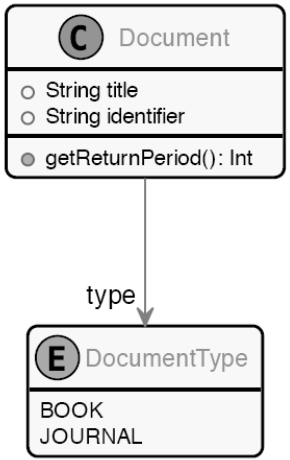
- Single Responsibility
- Open/Closed Principle
- Liskov Substitution
- Interface Segregation
- Dependency Injection

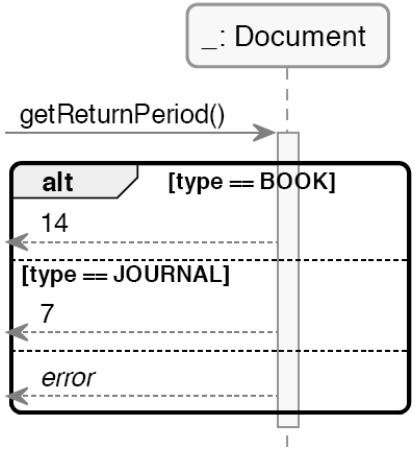
Single Responsibility

- A class should have one and only one reason to change
- Or the class should implement only one functionality

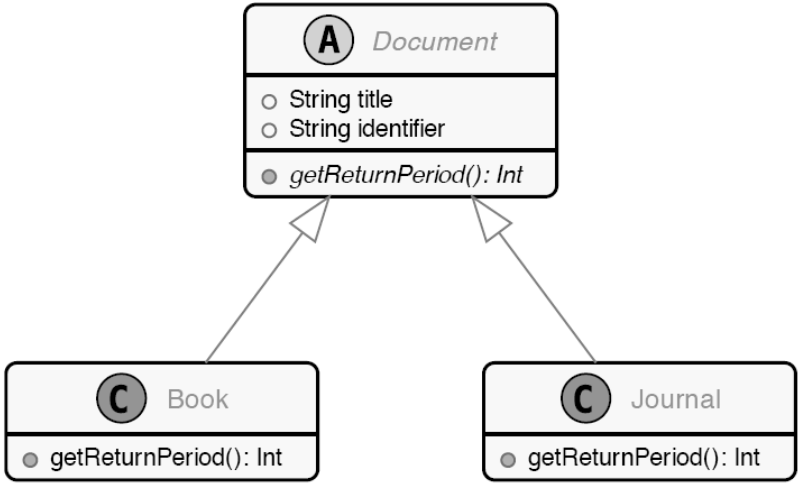
Open/Closed Principle

- Software entities should be Open for extension but Closed for modification





Switch statement

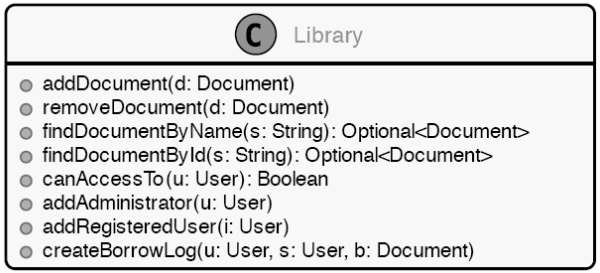


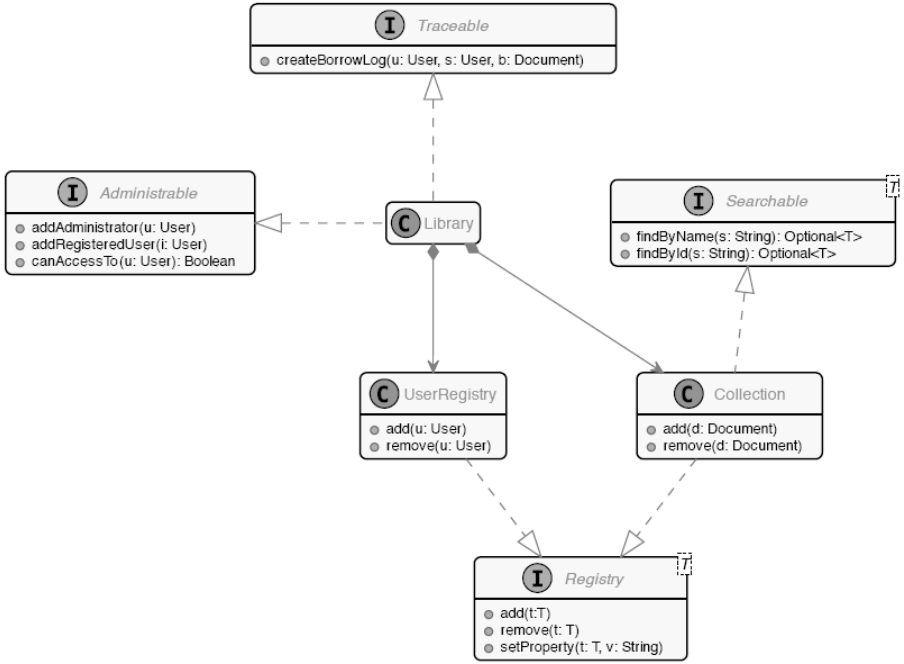
Liskov Substitution Principle

- *Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T. (Data Abstraction – 1987 – Barbara Liskov)*
- *Instances of a superclass should be replaceable by instances of a subclass*
- Example :
  - Shape, Rectangle and Square
  - Bird, Dove and Penguin
- Be careful with input and output parameters

Interface Segregation Principle

- *Clients should not be forced to depend upon interfaces that they do not use.*



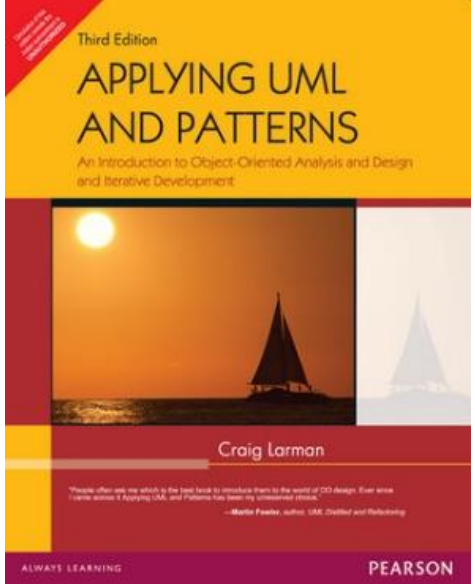


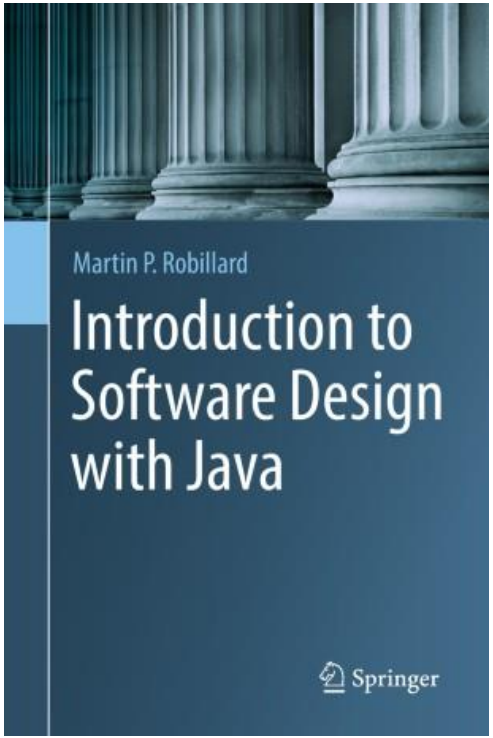
Dependency Inversion

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Grasp Patterns

- Recognize that according to Craig Larman:
- “The skillful assignment of responsibilities is extremely important in object design,
- Determining the assignment of responsibilities often occurs during the creation of interaction diagrams and certainly during programming.”





GRASP

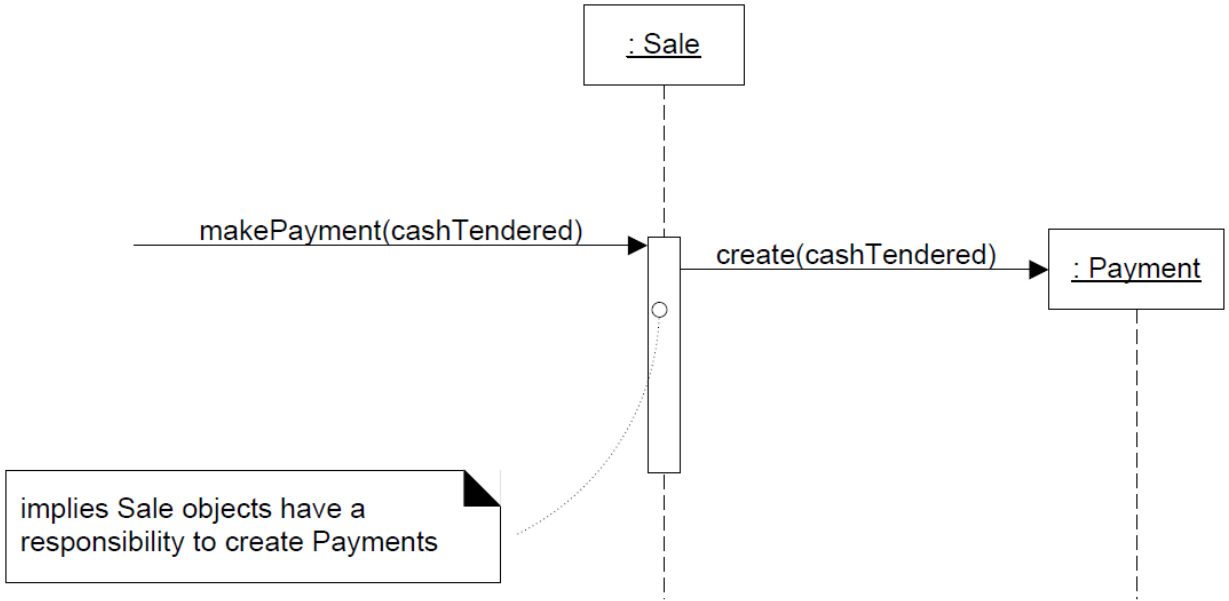
- During Object Design
  - Make choice about the assignment of responsibilities to software classes

What is a responsibility

- It’s not a method
- It’s the abstraction of a behavior

- Responsibility of knowing things
  - A value, a collection of values, a derivation based on value (age from date)
- Responsibility of doing things
  - Orchestrate actions
  - Create new objects
  - Delegate to other objects

Responsibility



Information Expert

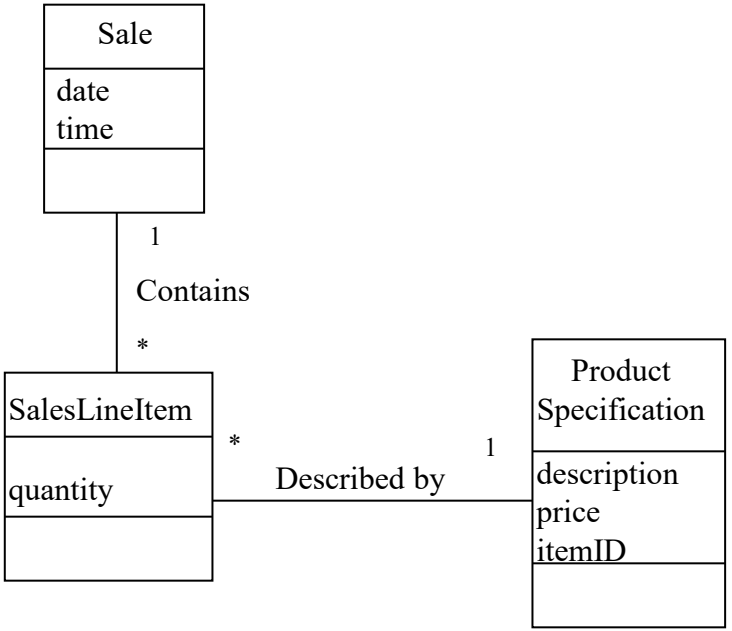
- Sale example
- Who is responsible for knowing the total of the sale ?
- Who has the information to determine the total ?

Information Expert

- 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

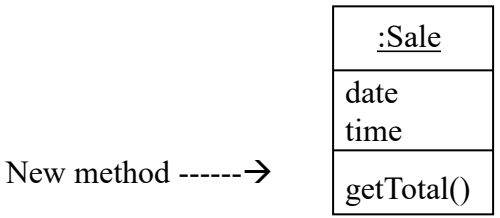
Expert– Using Domain Model

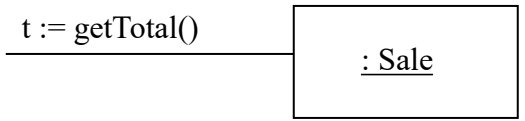
- There is a Sale class in the domain model



Add Sale Class to the design model

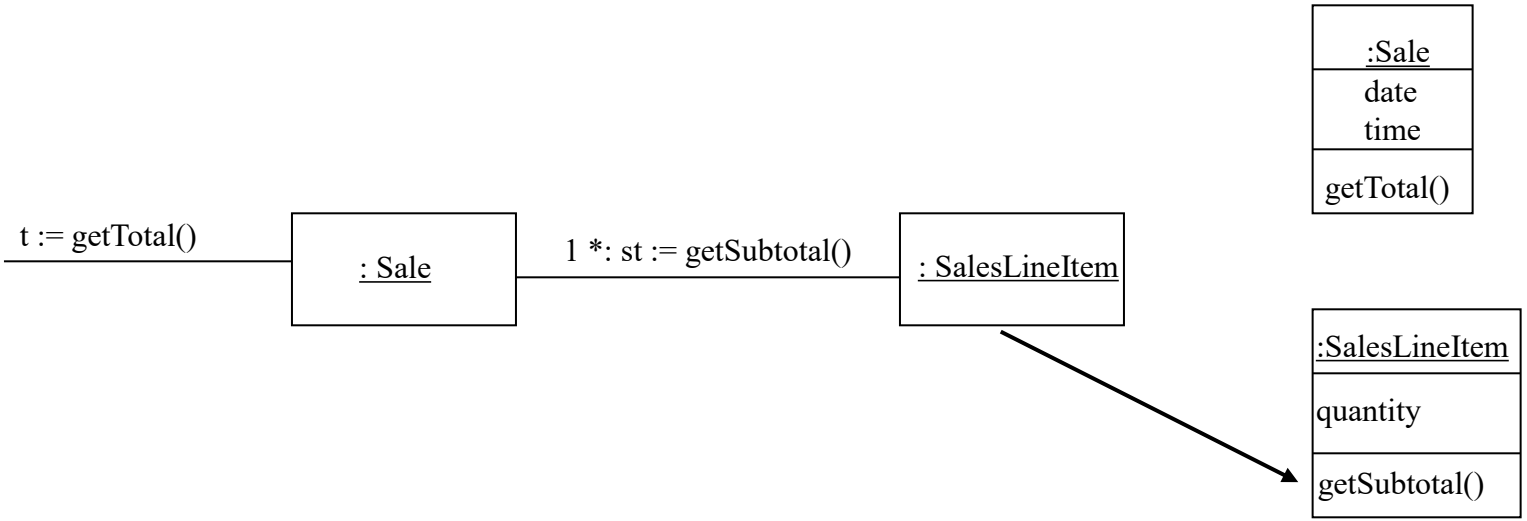
- Add the responsibility of knowing its total
  - Method getTotal()





And then

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

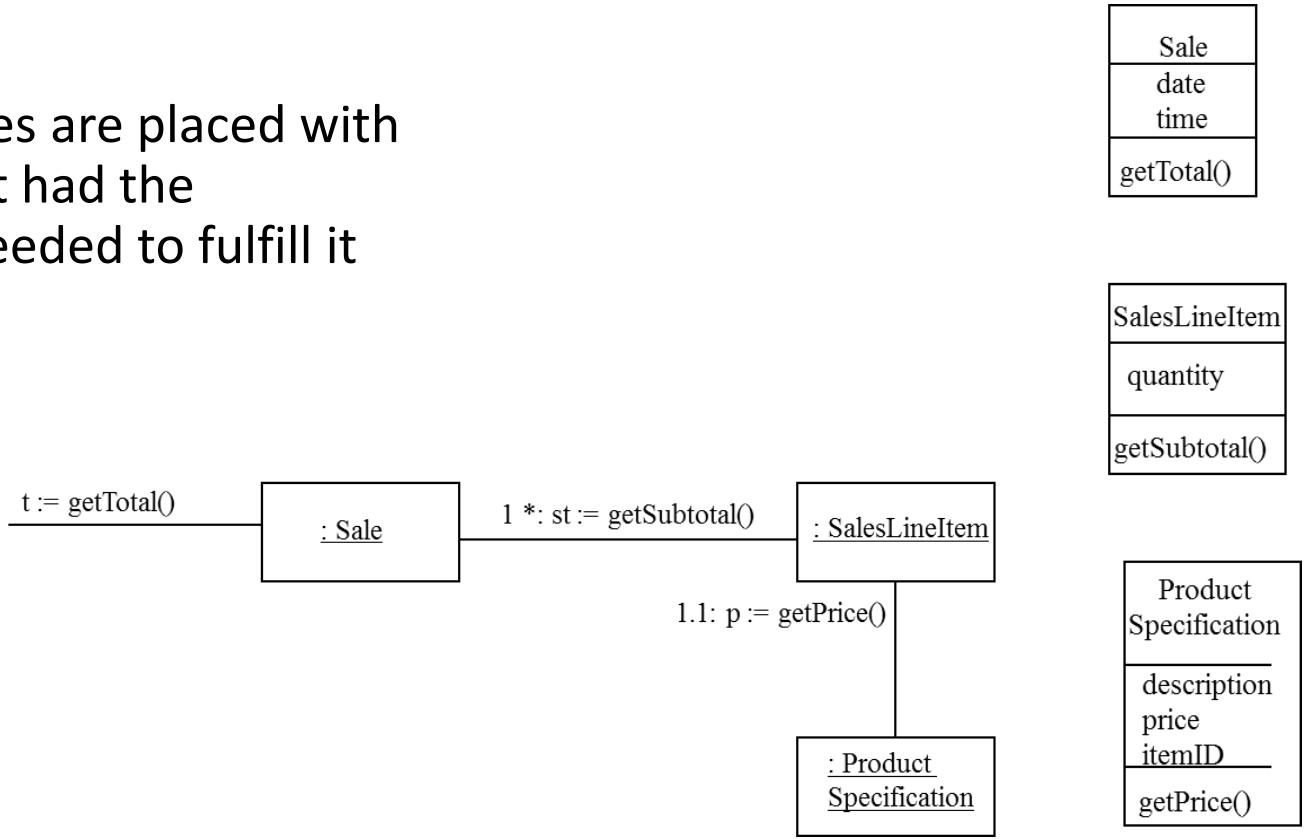


How the domain model is used

- 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
  - Break several SOLID principles

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 information expert

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

Creator

- Who is responsible for creating new instances of some classes

Solution

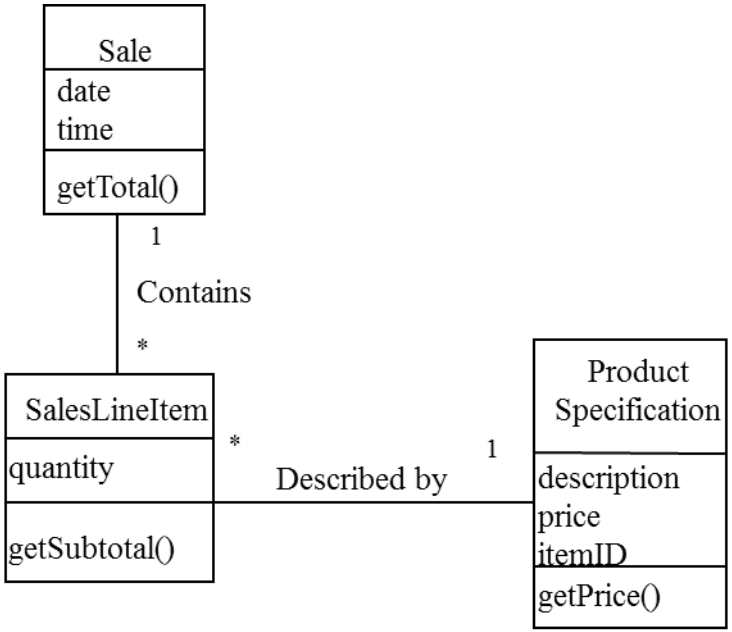
- Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - B aggregates A (simple aggregate; shared attributes)
  - B contains A (composition; non-shared attributes)
  - B records instances of A objects
  - B closely uses A objects
  - B has the 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

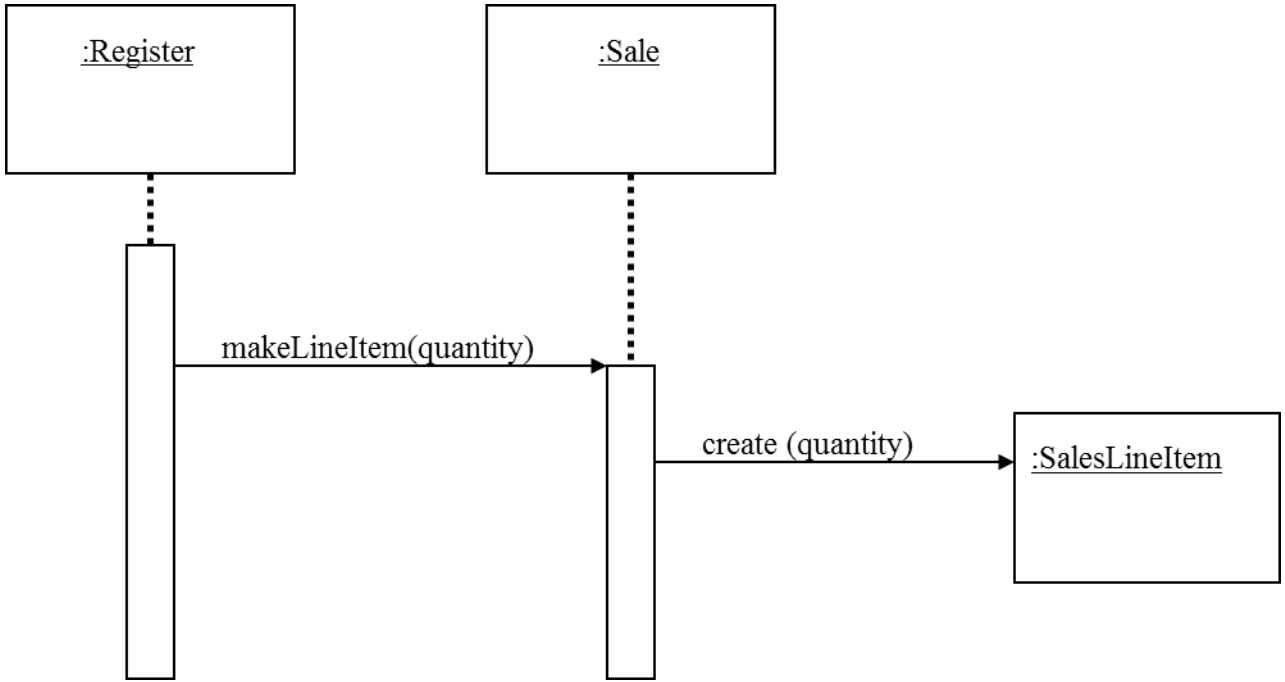
- Who is responsible for creating SalesLineItem



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

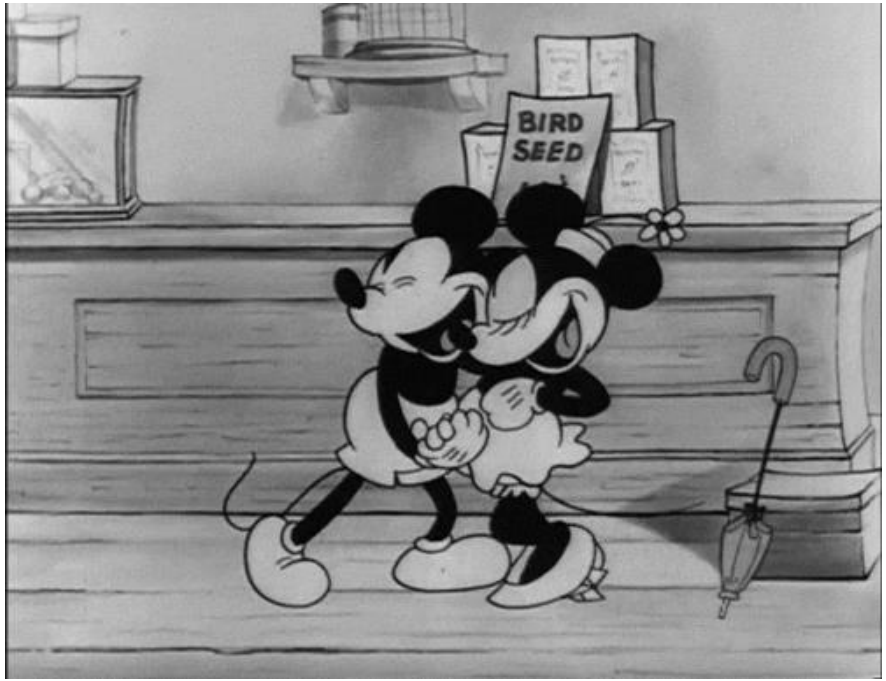
- Creator is a kind of expert

Creator

- Sometimes it is better to delegate creation to a helper Class
- The Factory design pattern

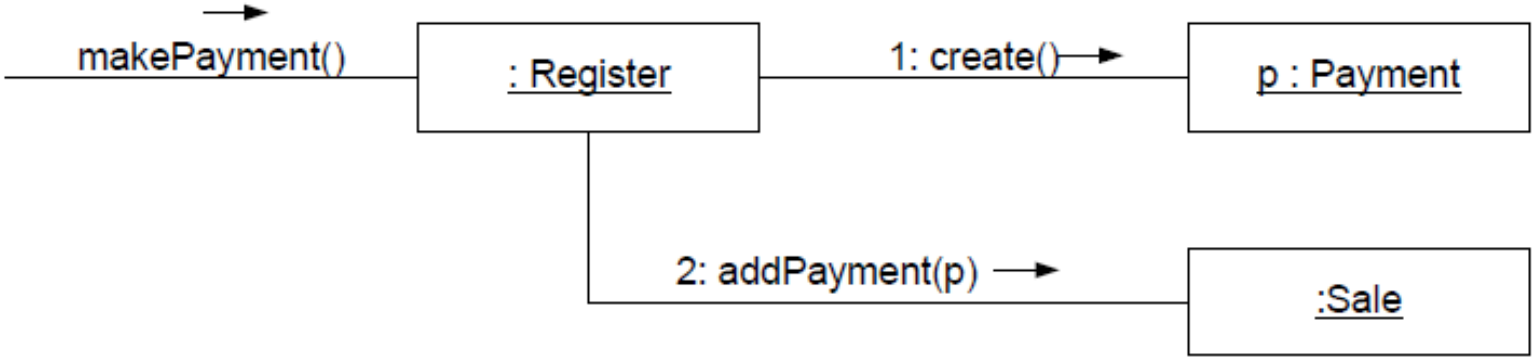
Low Coupling

- 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



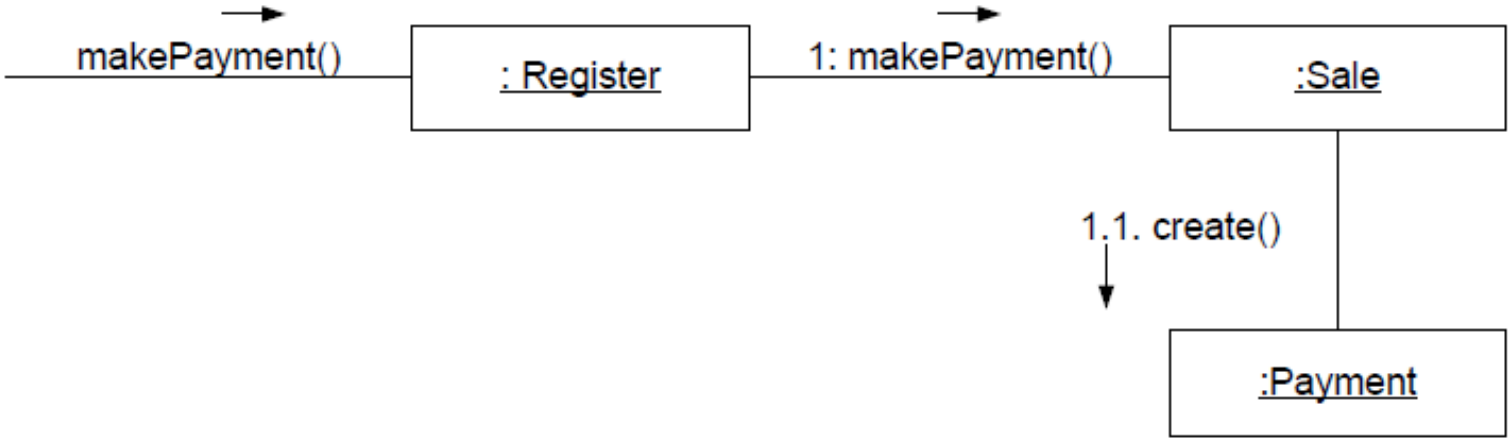
Example

- Register is coupled to payment



Alternative

- Payment known from Sale. Sale has to know Payment



Common form of coupling

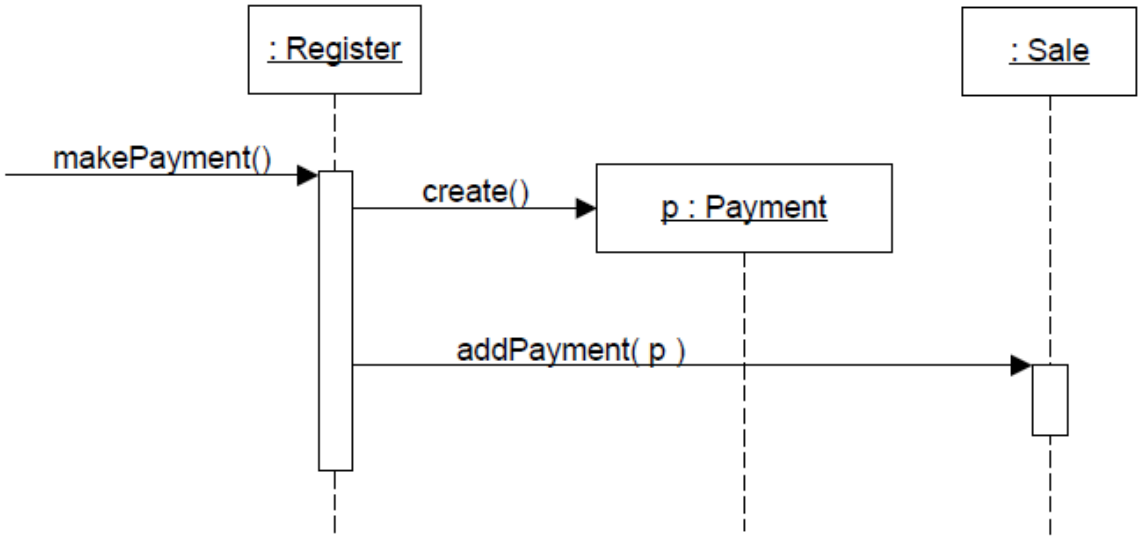
- TypeX has an attribute that refers to TypeY
- TypeX instance call a service on a TypeY instance
- TypeX has a method that references an instance of TypeY (parameter, local variable)
- TypeX is a subclass of TypeY
- TypeY is an interface and TypeX implements it

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

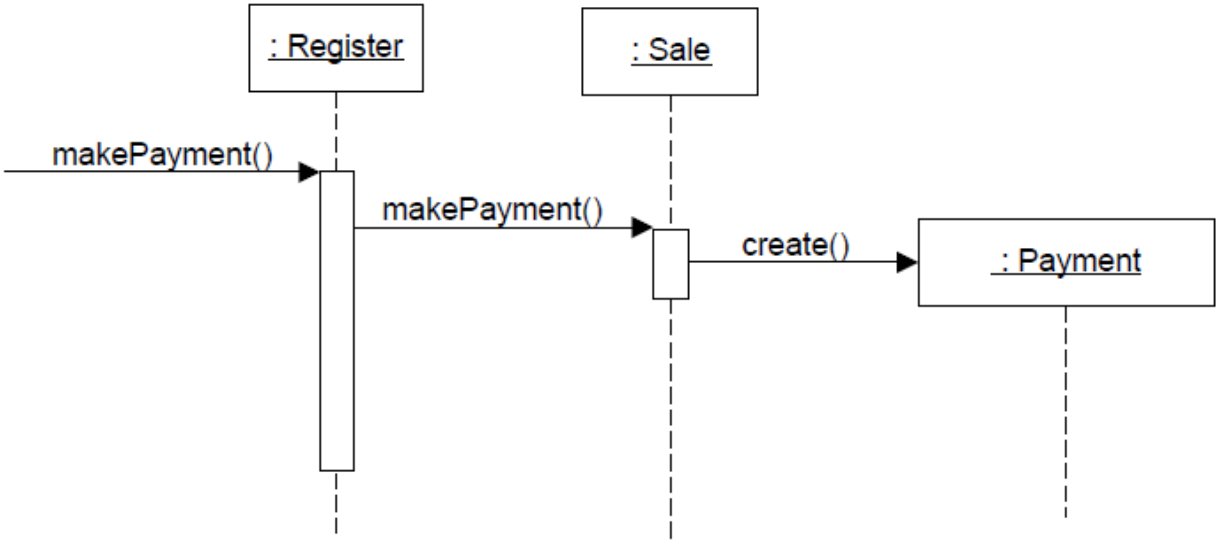
Example

- Register creates payment



Same alternative

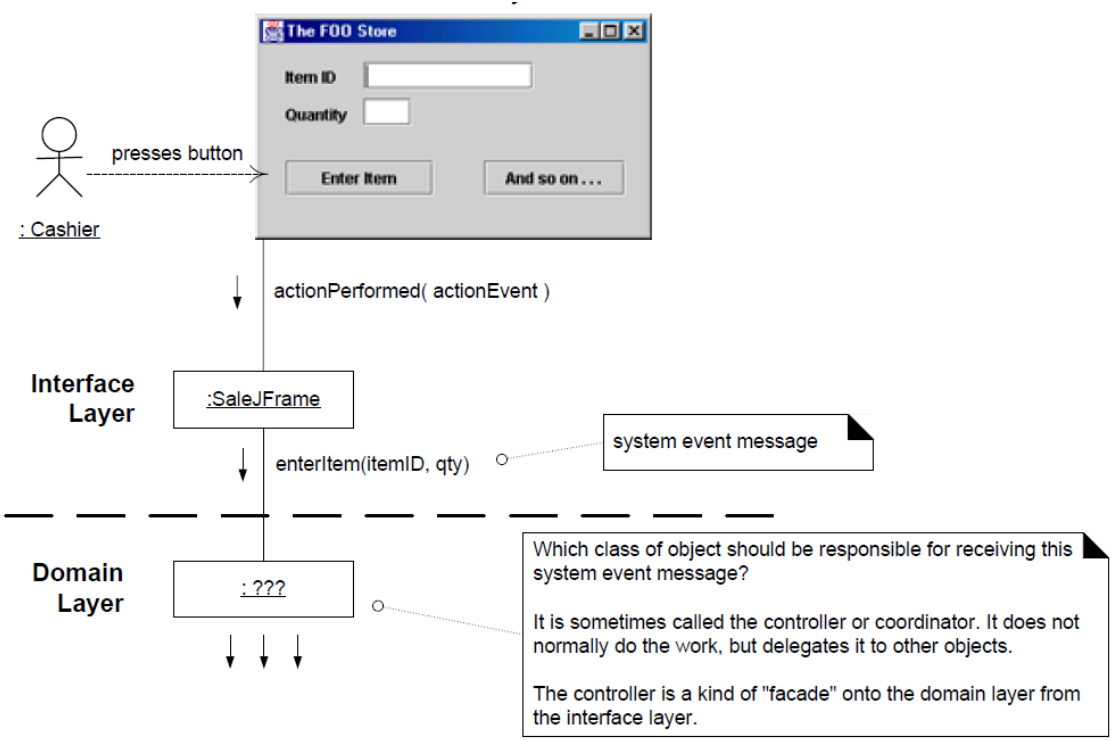
- Register has less responsibilities – Higher cohesion



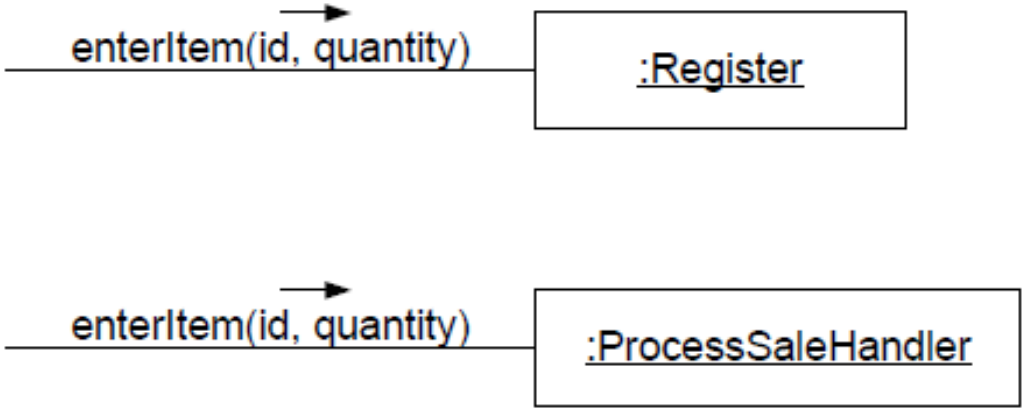
Controller

- 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

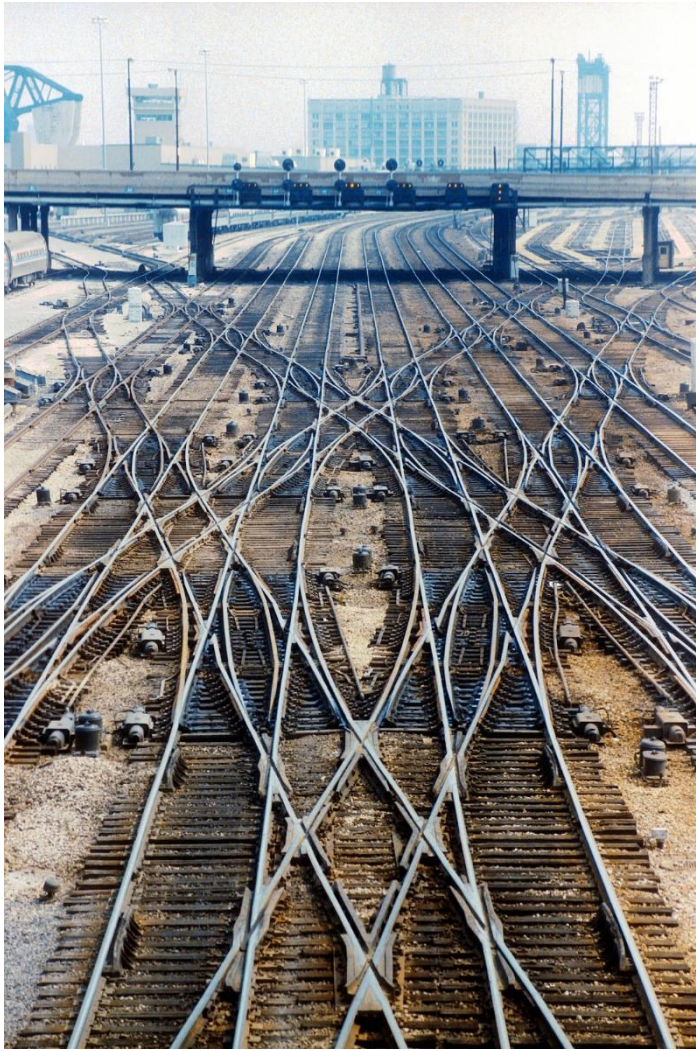

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
enterItem(itemID, qty)
system event message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes 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

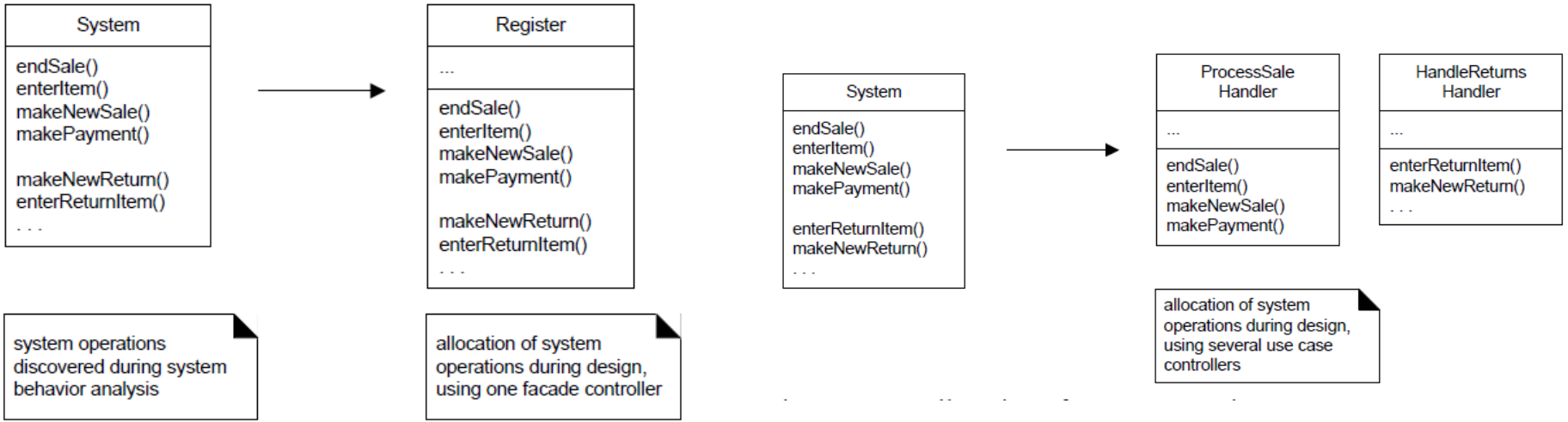


The controller delegates

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



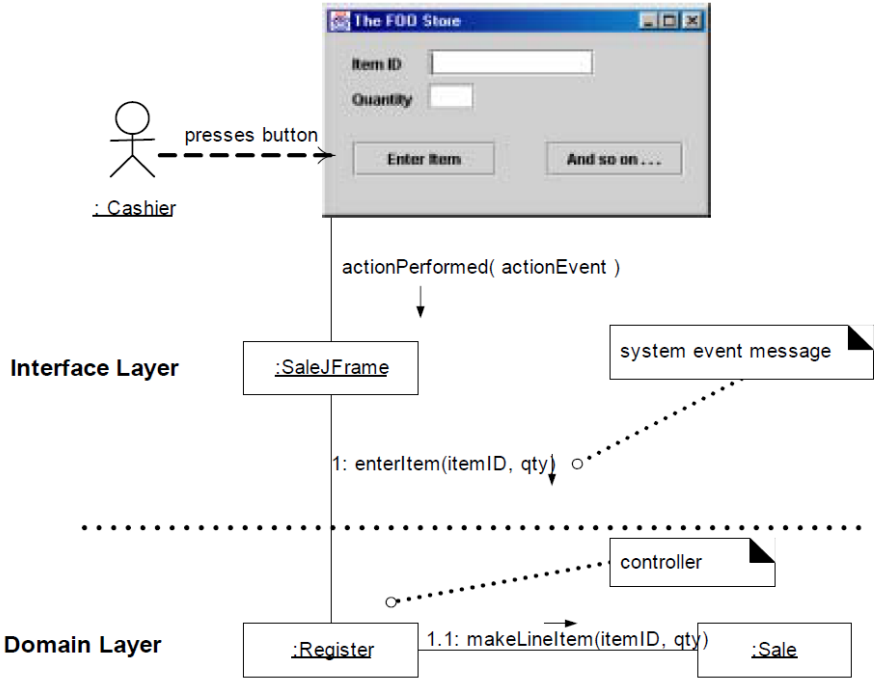
Allocation of operations

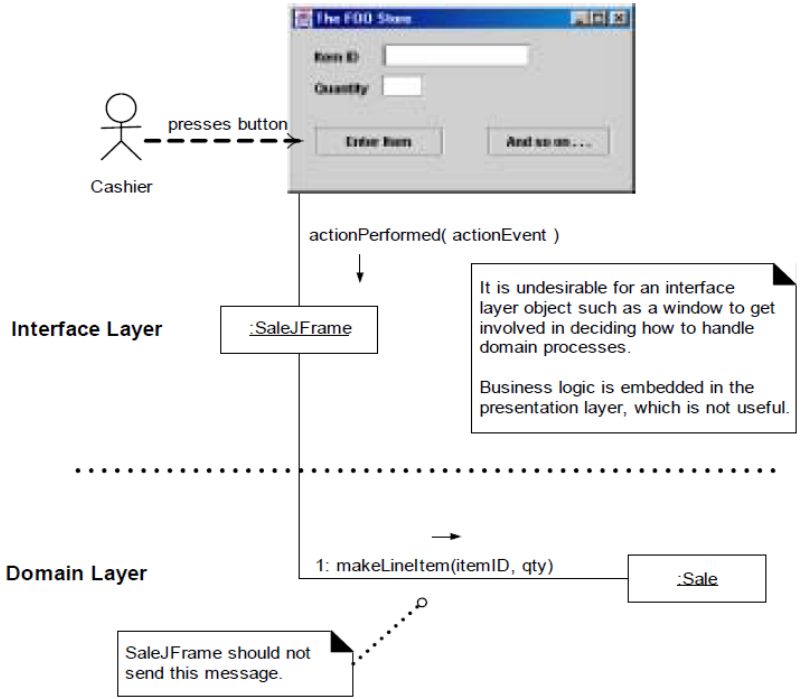


Issues

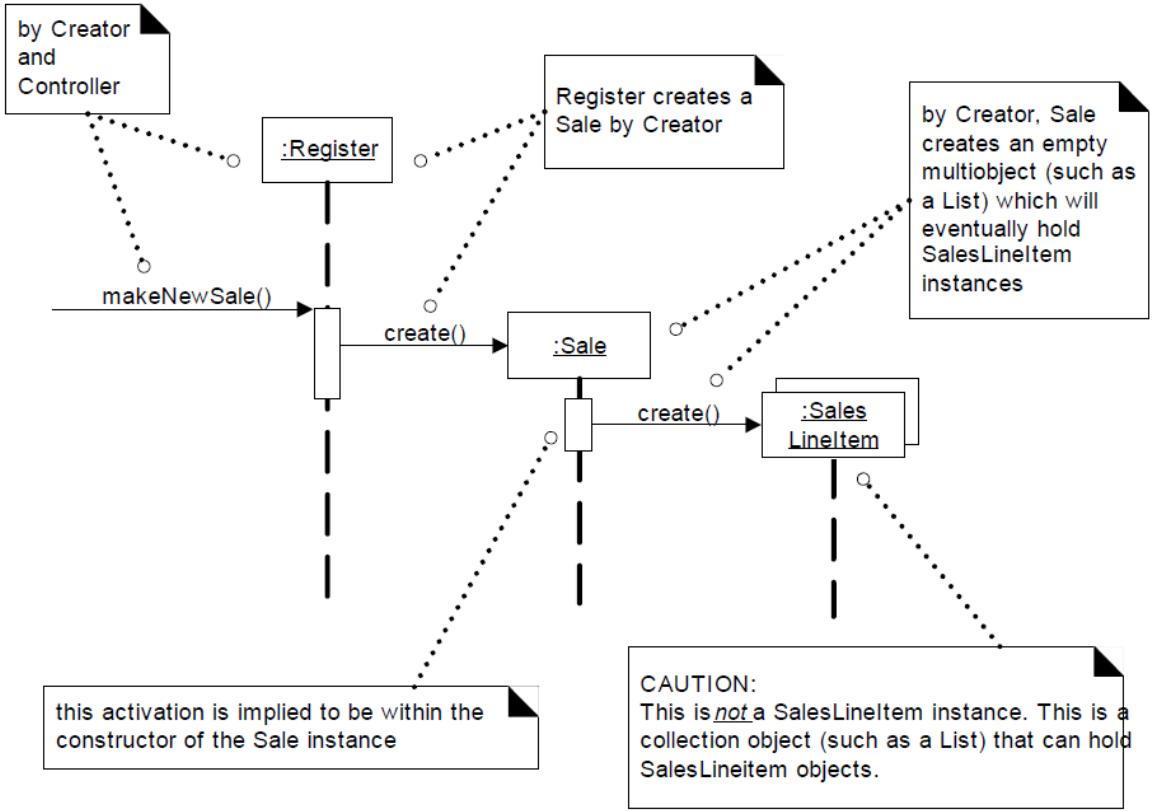
- Avoid bloated controllers (low cohesion)
  - Add more controllers
  - The controller delegates the responsibility to fulfill operation on to other objects.

Two couples

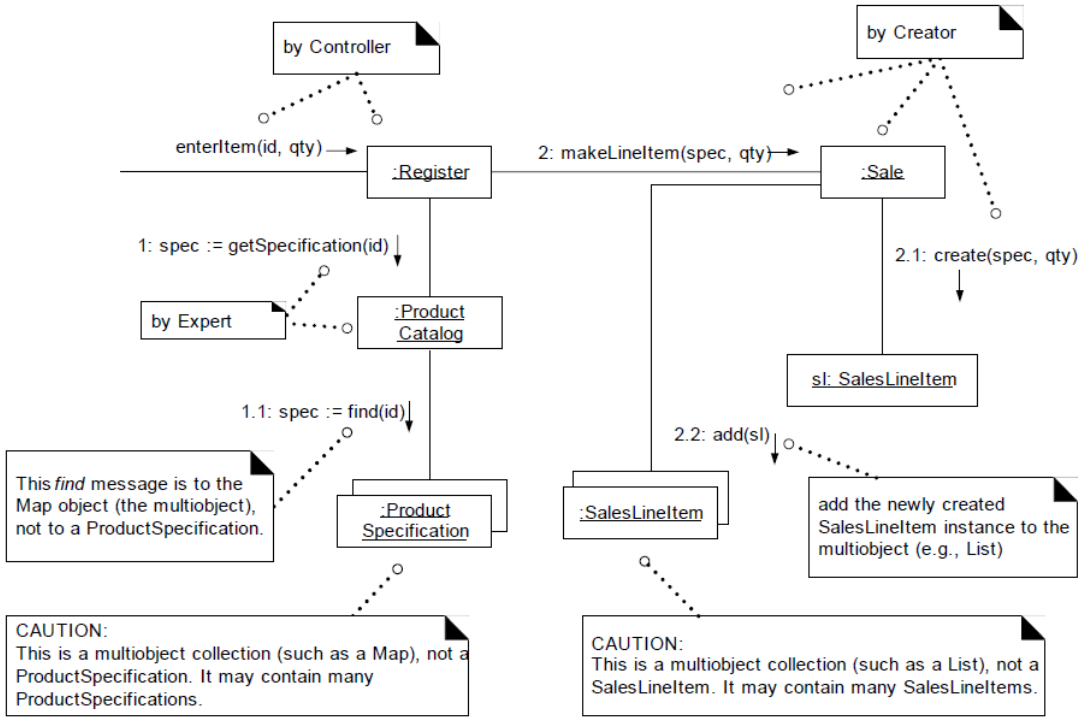

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
system event message
1: enterItem(itemID, qty)
controller
Domain Layer
:Register
1.1: makeLineItem(itemID, qty)
:Sale


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
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.
Domain Layer
1: makeLineItem(itemID, qty)
:Sale
SaleJFrame should not send this message.

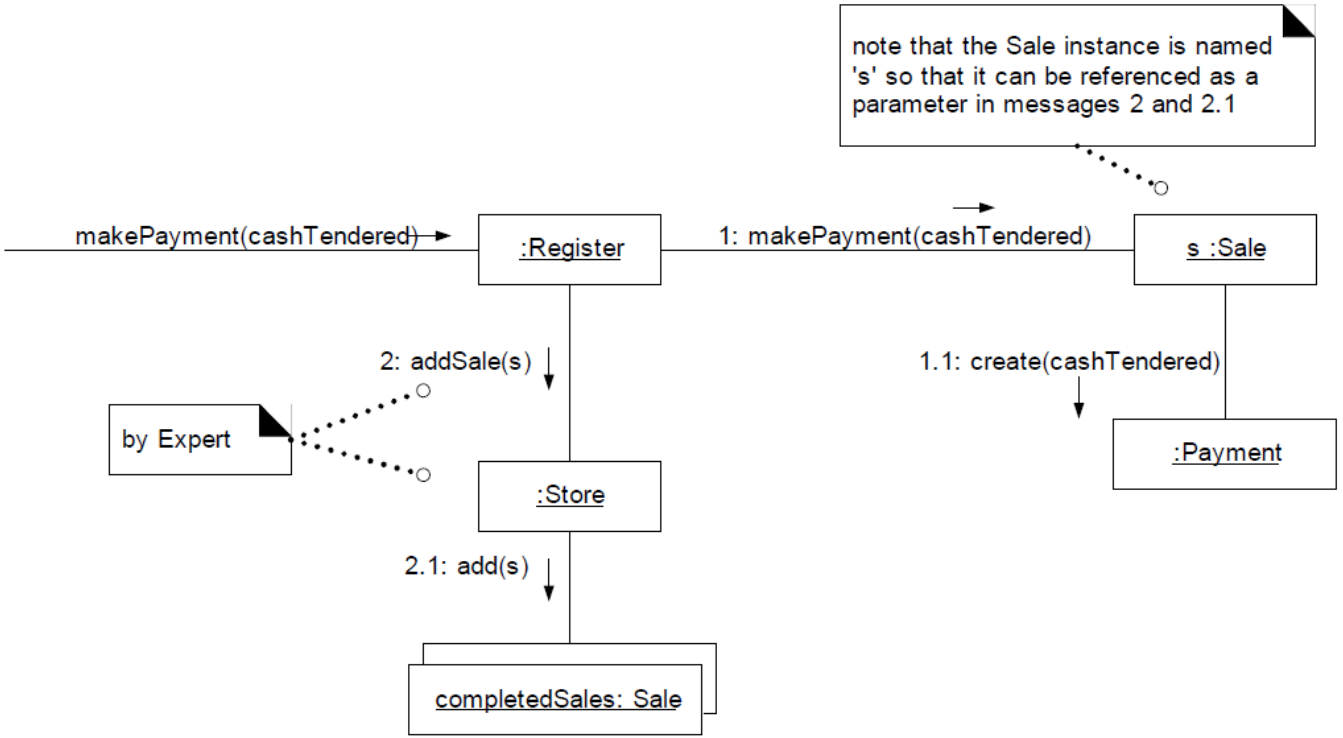
Creating a Sale



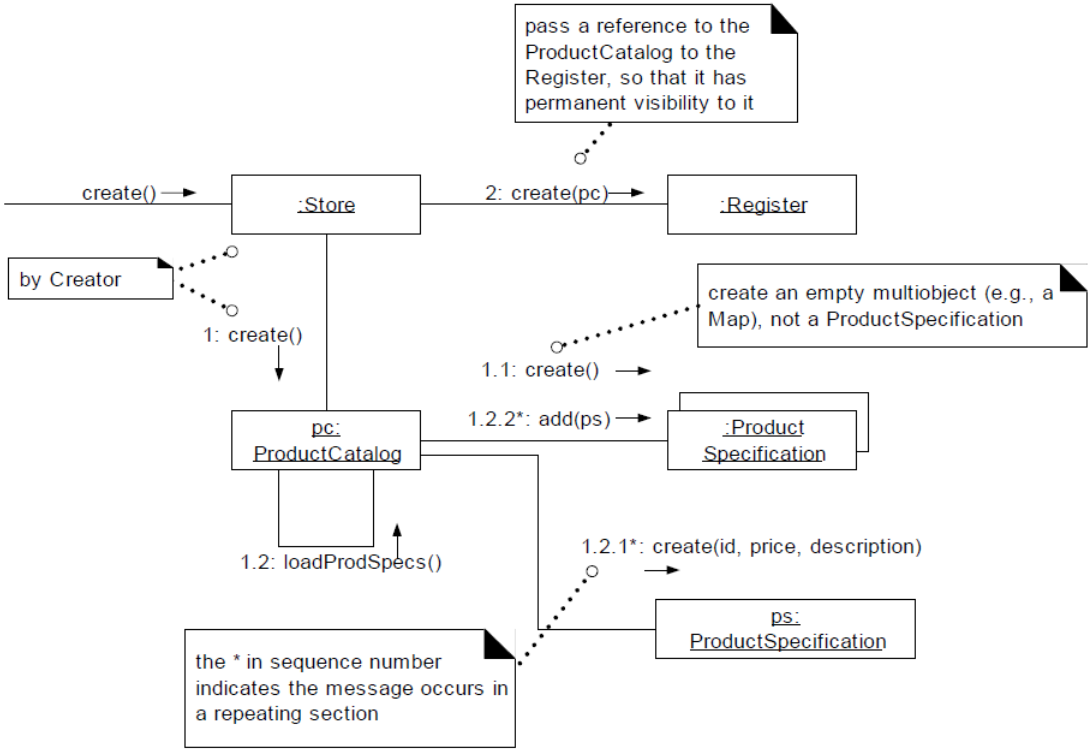
Enter an Item to the Sale



Making payment



Initialisation



Remember

- Low Coupling/High Cohesion
- Expert
- Creator
- Controller
- Guide for design decision.

Resources

- www.unf.edu/~broggio/cen6017/38.DesignPatters-Part2.ppt
- www.academic.marist.edu/~jzbv/.../DesignPatterns/GRASP.pp
- …