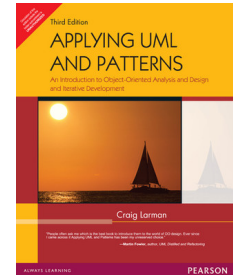


Conception OBJET GRASP

General Responsibility Assignment Software Patterns

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.”



Resources

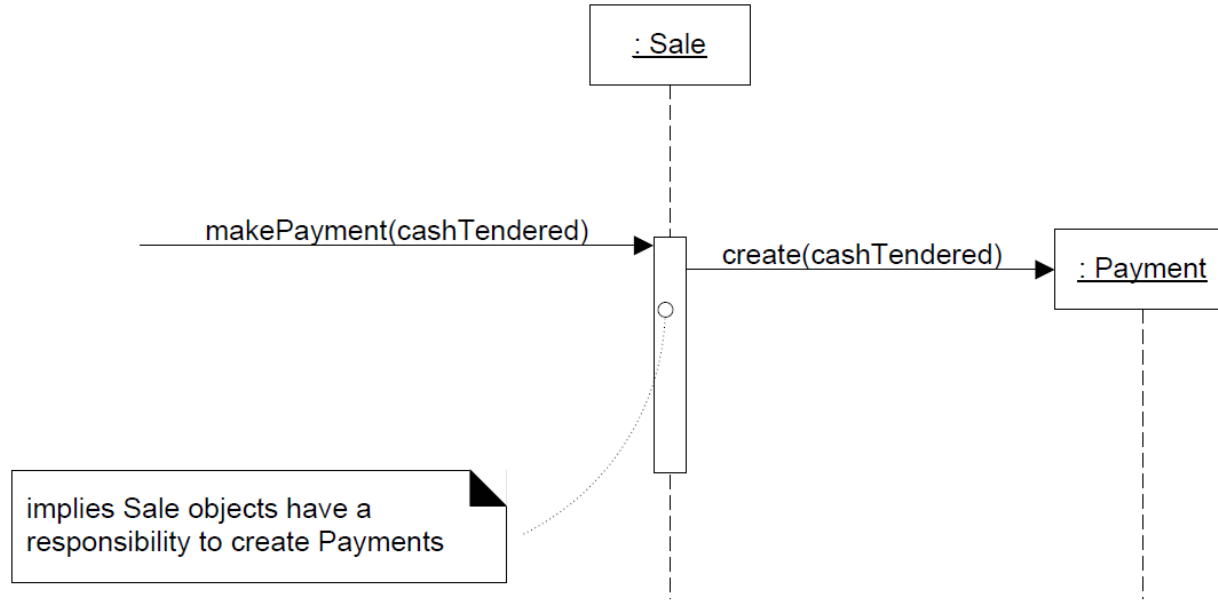
- www.unf.edu/~broggio/cen6017/38.DesignPatters-Part2.ppt
- www.academic.marist.edu/~jzbv/.../DesignPatterns/GRASP.pp
- ...

GRASP

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



Responsibility



Expert

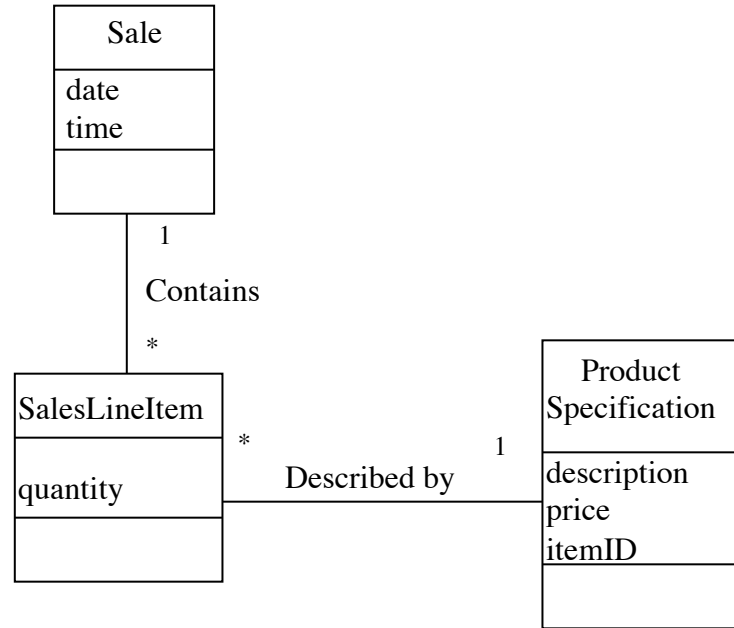
- Sale example
- Who is responsible for knowing the total of the sale ?
- Who has the information to determine the total

(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

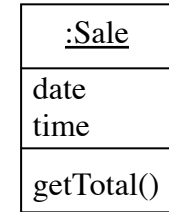
- There is a Sale class in the domain model



Add Sale Class to the design model

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

New method ----->

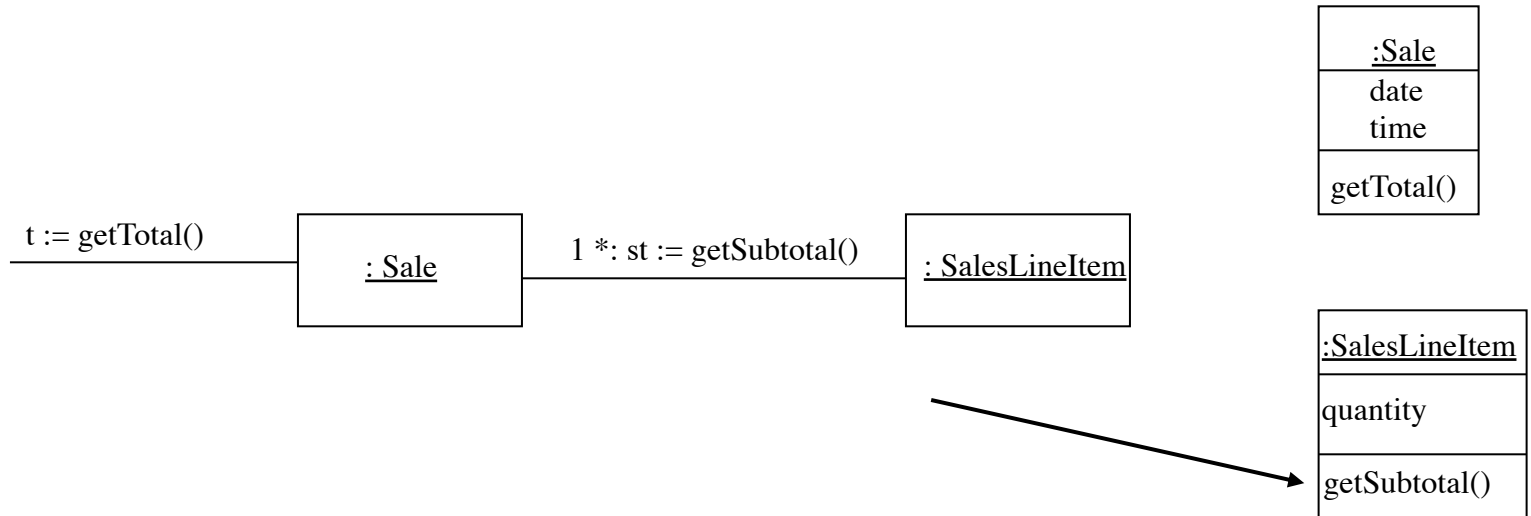


t := getTotal()



And then

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

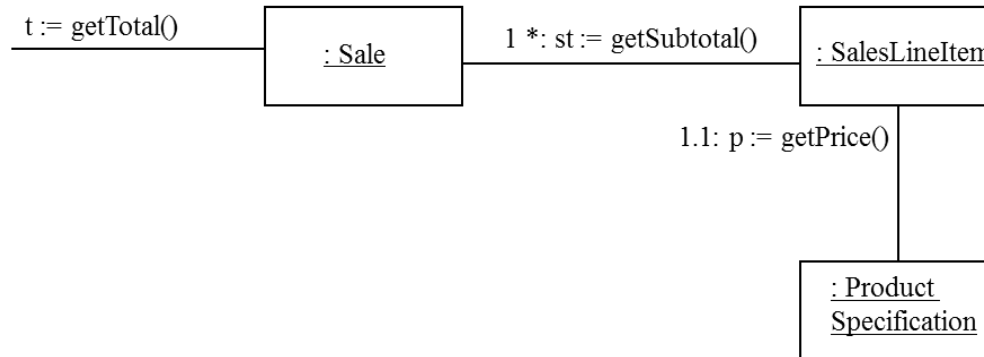


How the domain model is used

- 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

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



Sale
date
time
getTotal()

SalesLineItem
quantity
getSubtotal()

Product Specification
description
price
itemID
getPrice()

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

- Who is responsible for creating new instances of some classes



Solution

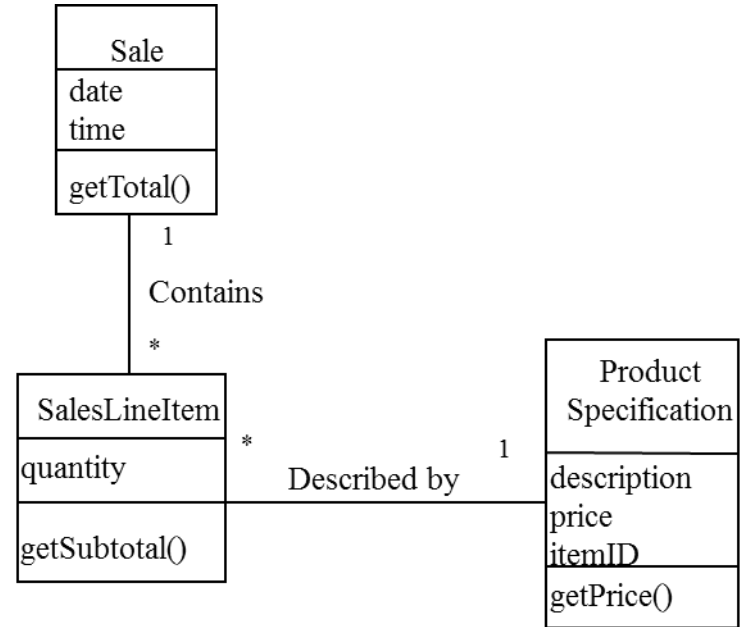
- Assign class B the responsibility to create an instance of class A if one or more of the following is true:
 - B aggregates A (simple aggregate; shared attributes)
 - B contains A (composition; non-shared attributes)
 - B records instances of A objects
 - B closely uses A objects
 - B has the initializing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

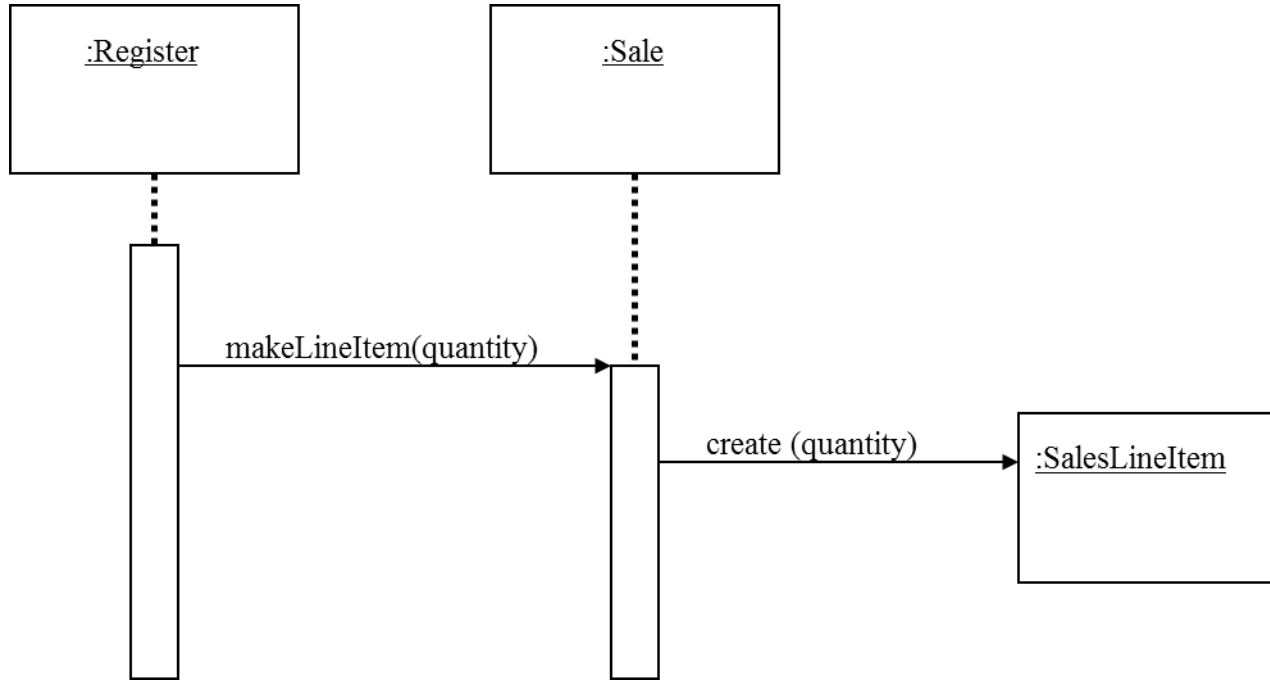
- Who is responsible for creating SalesLineItem



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

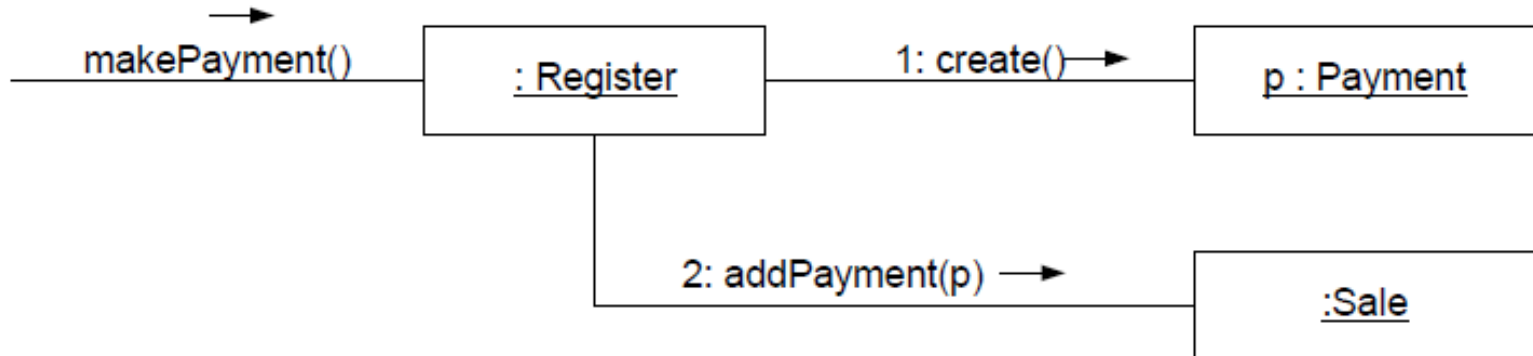
Low Coupling

- 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



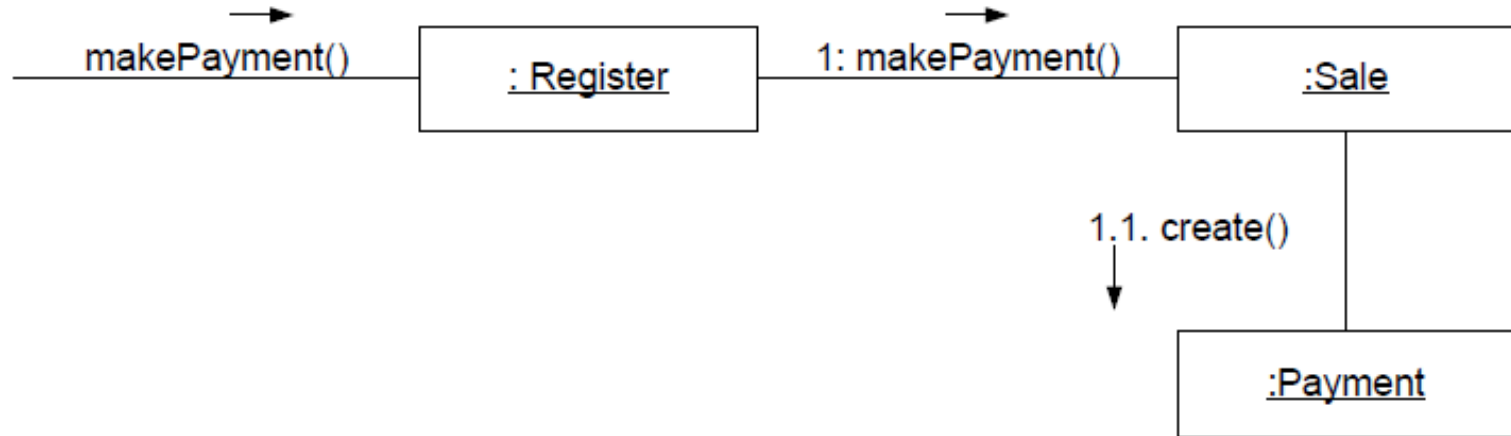
Example

- 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

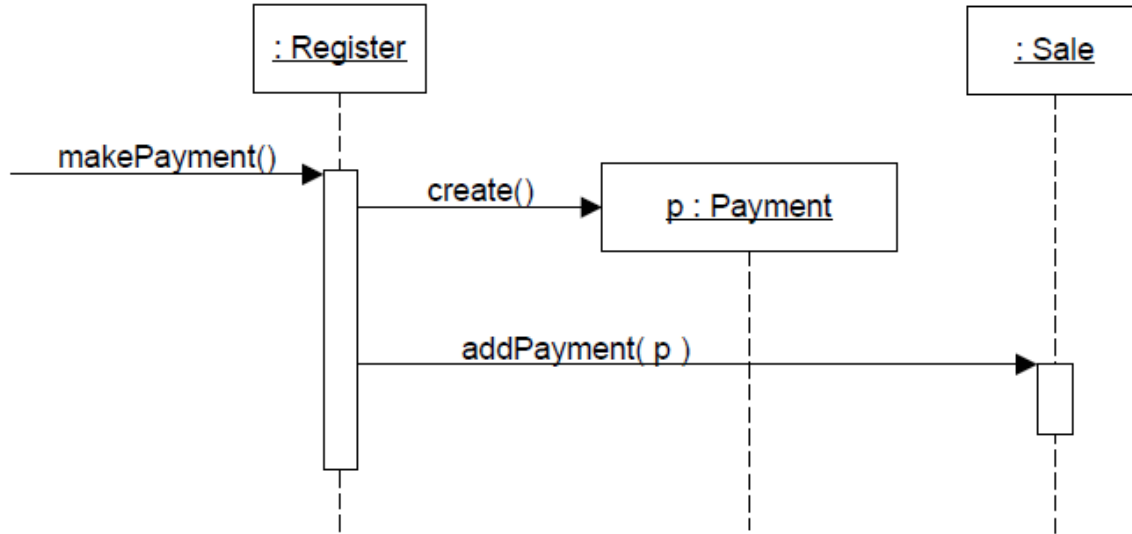
High Cohesion

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



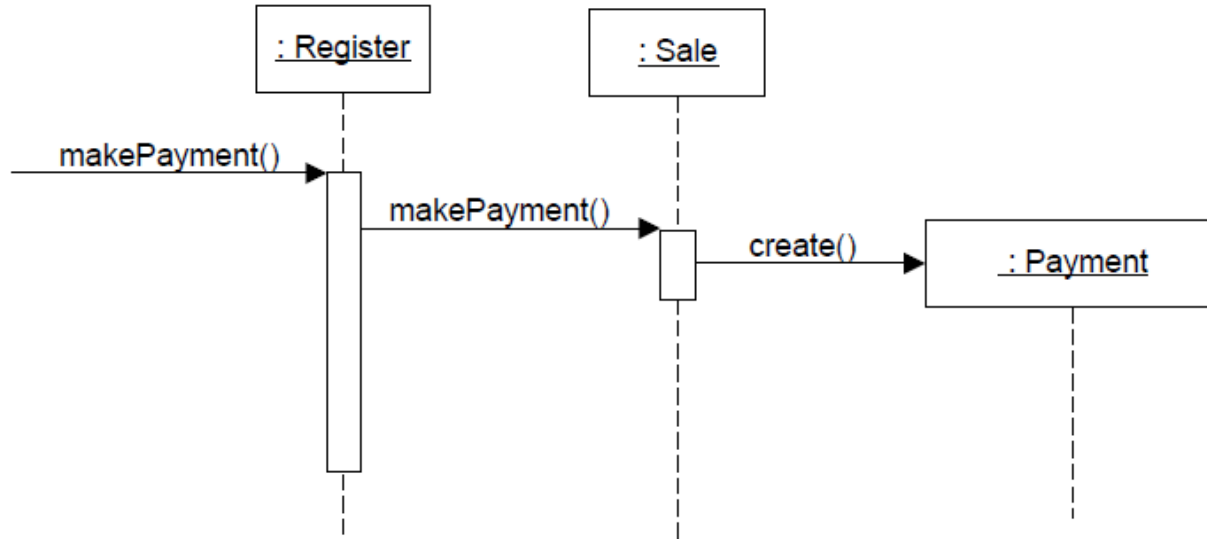
Example

- Register creates payment



Same alternative

- Register has less responsibilities – Higher cohesion



Scenarios (Booch94)

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

- o 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.

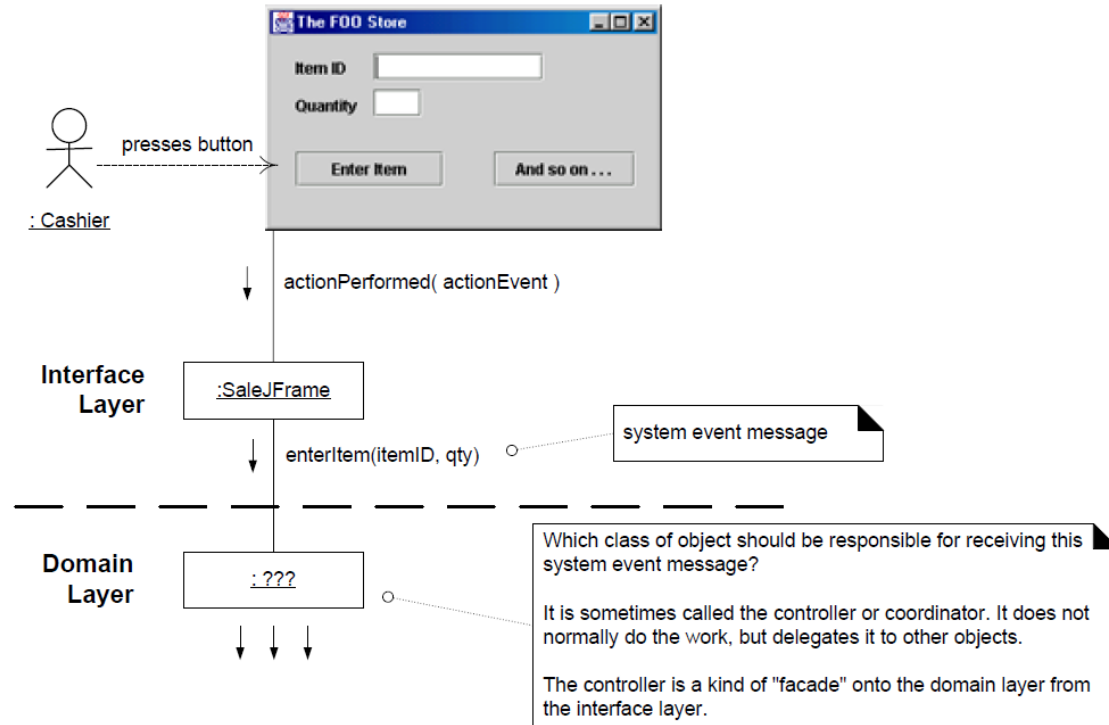
High cohesion—A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

- o 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.

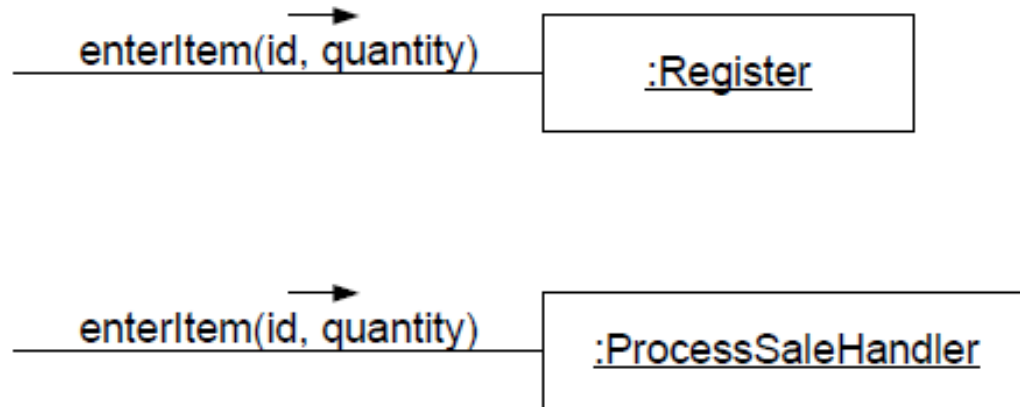
Controller

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

Example



Two possibilities

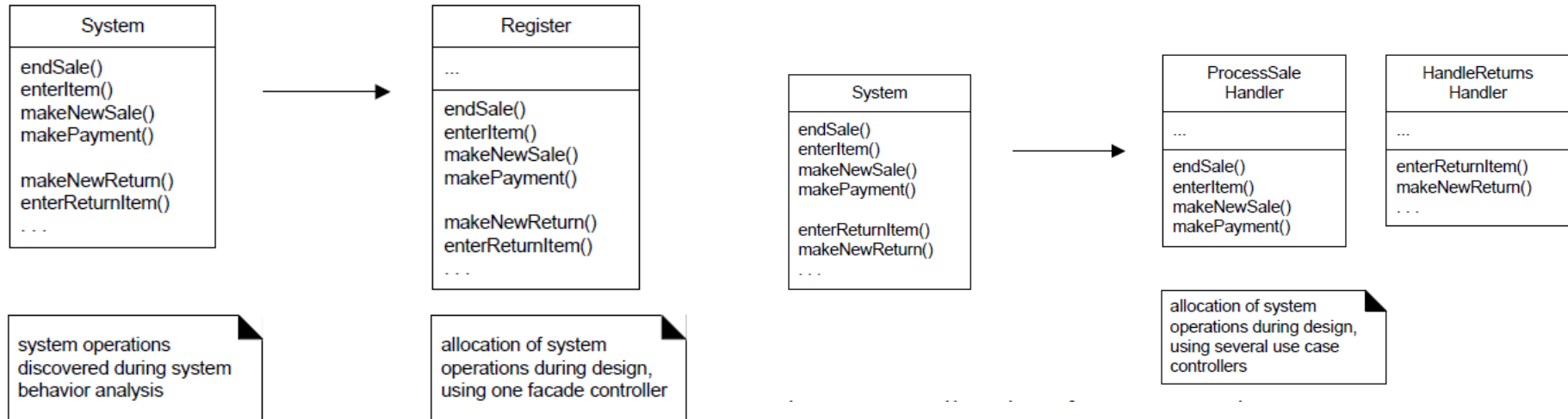


The controller delegates

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



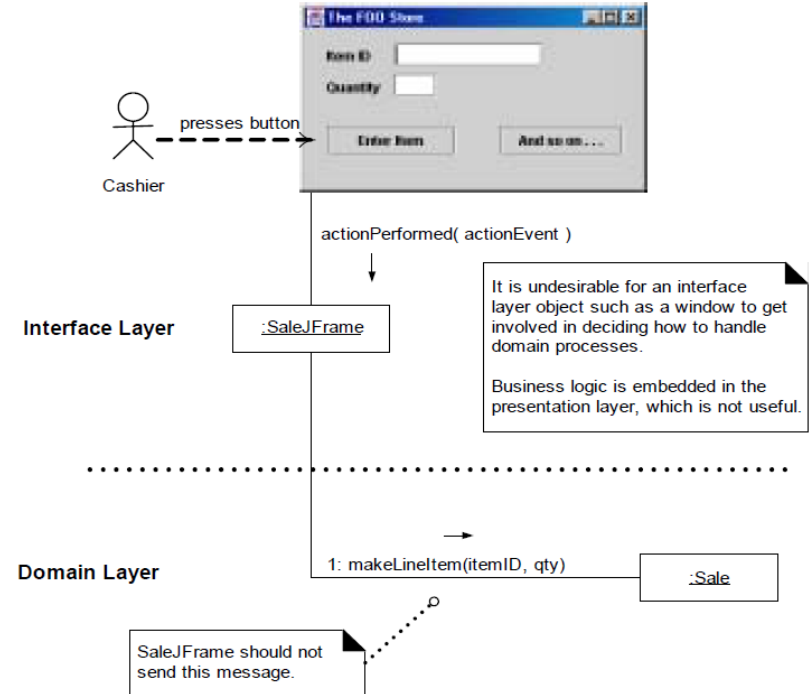
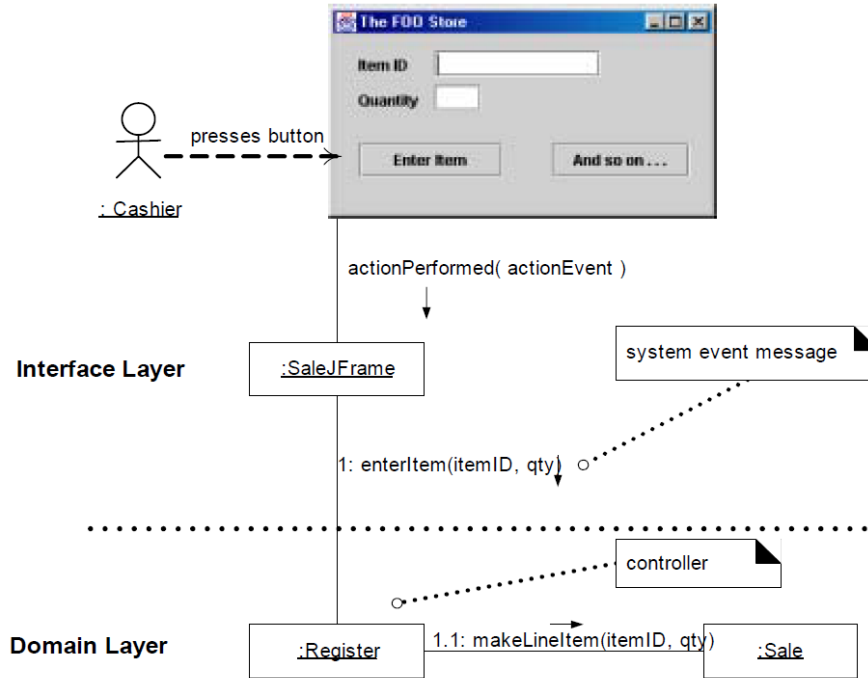
Allocation of operations



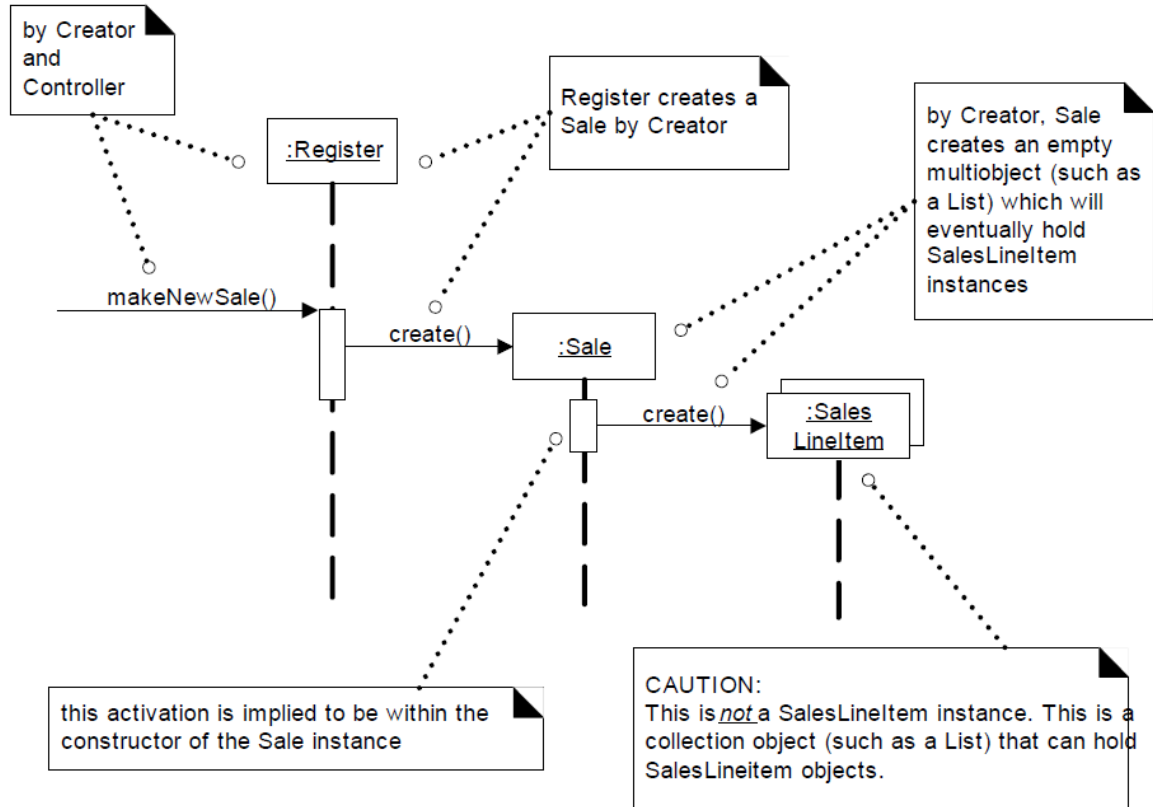
Issues

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

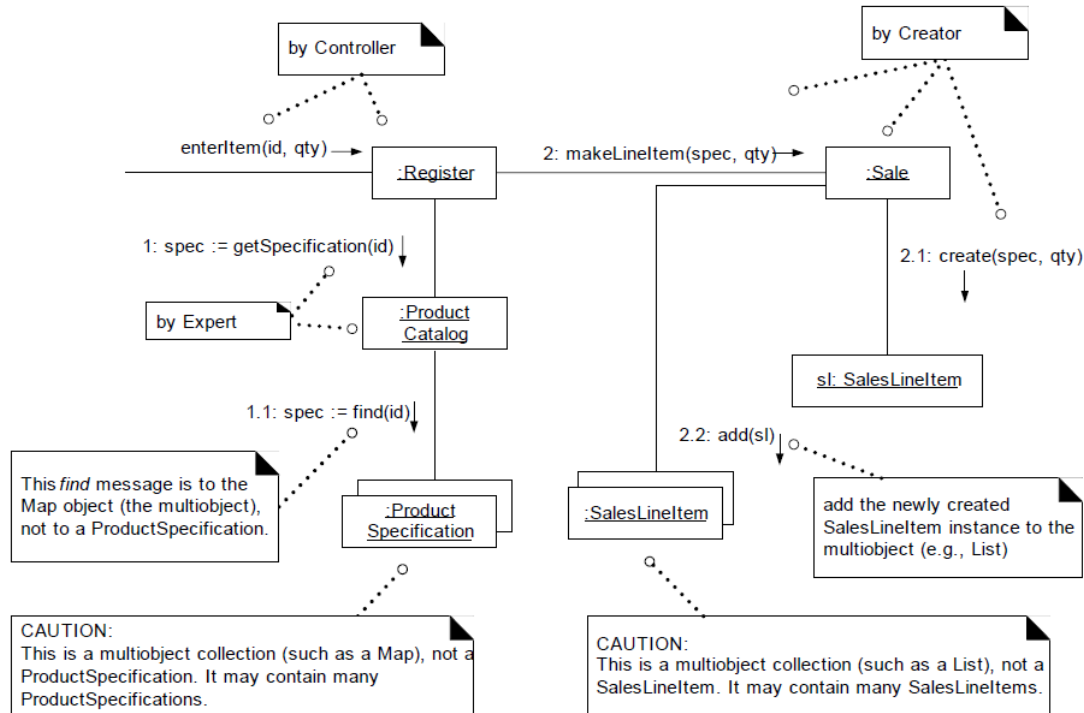
Two couples



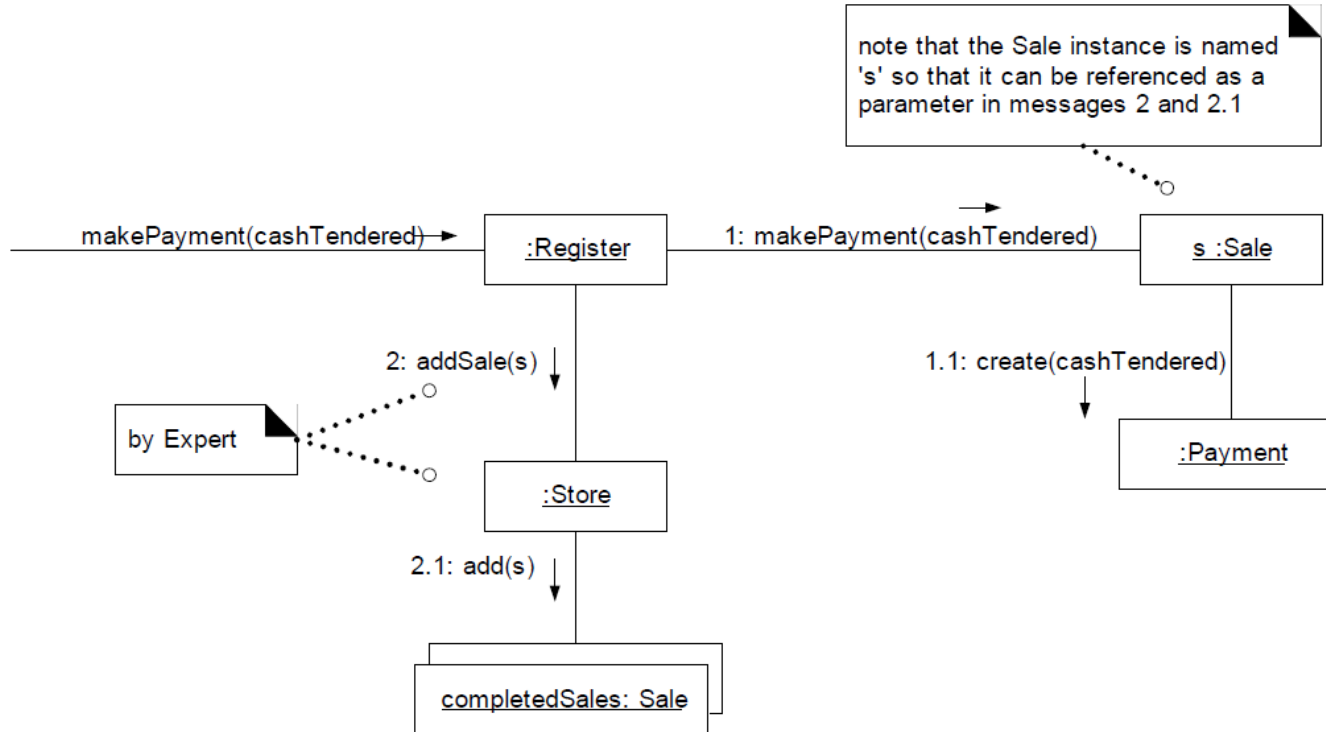
Creating a Sale



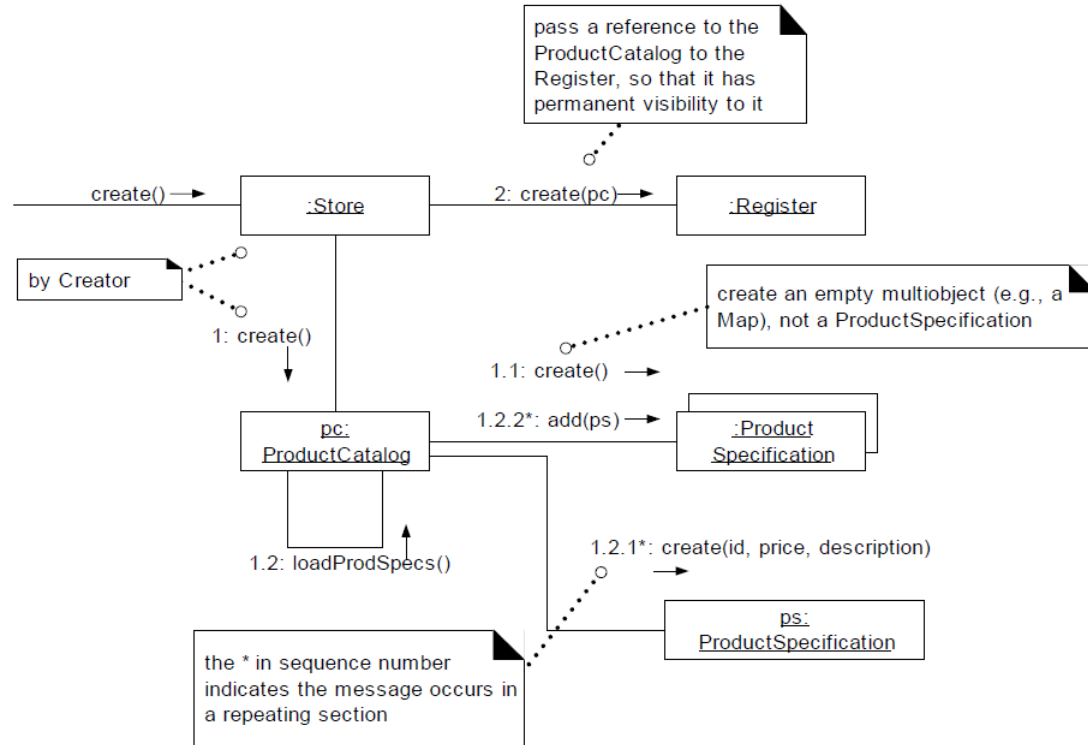
Enter an Item to the Sale



Making payment



Initialisation



Remember

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



More principles

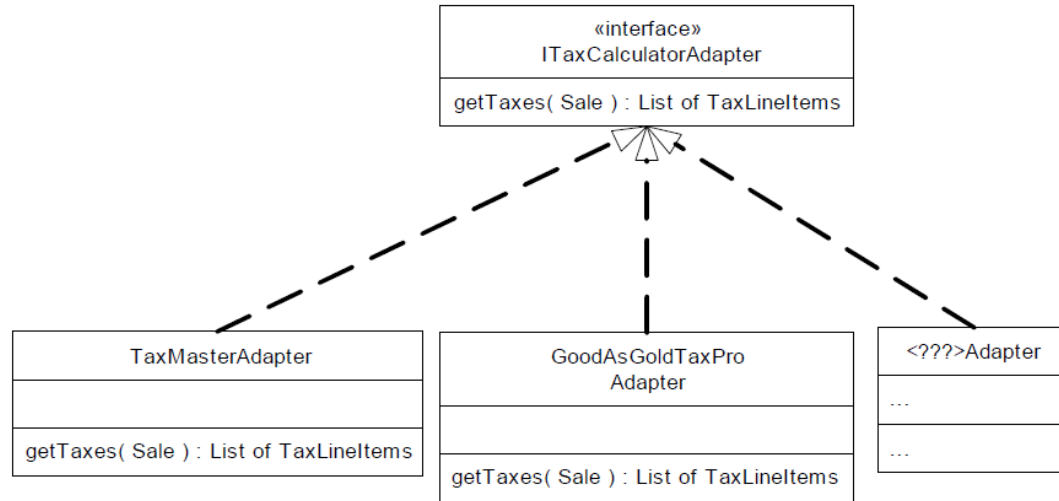
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variation



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



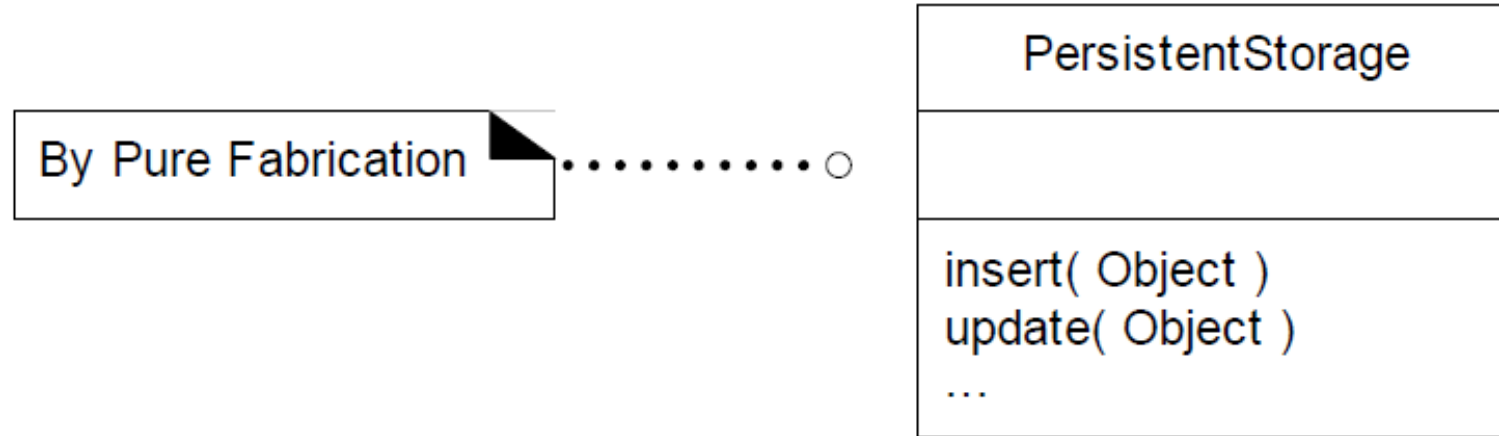
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Final

- Very easy to extend and add variations
- New implementations can be added without affecting the client
- Do it only if there are known variations (no future proofing)

Pure Fabrication

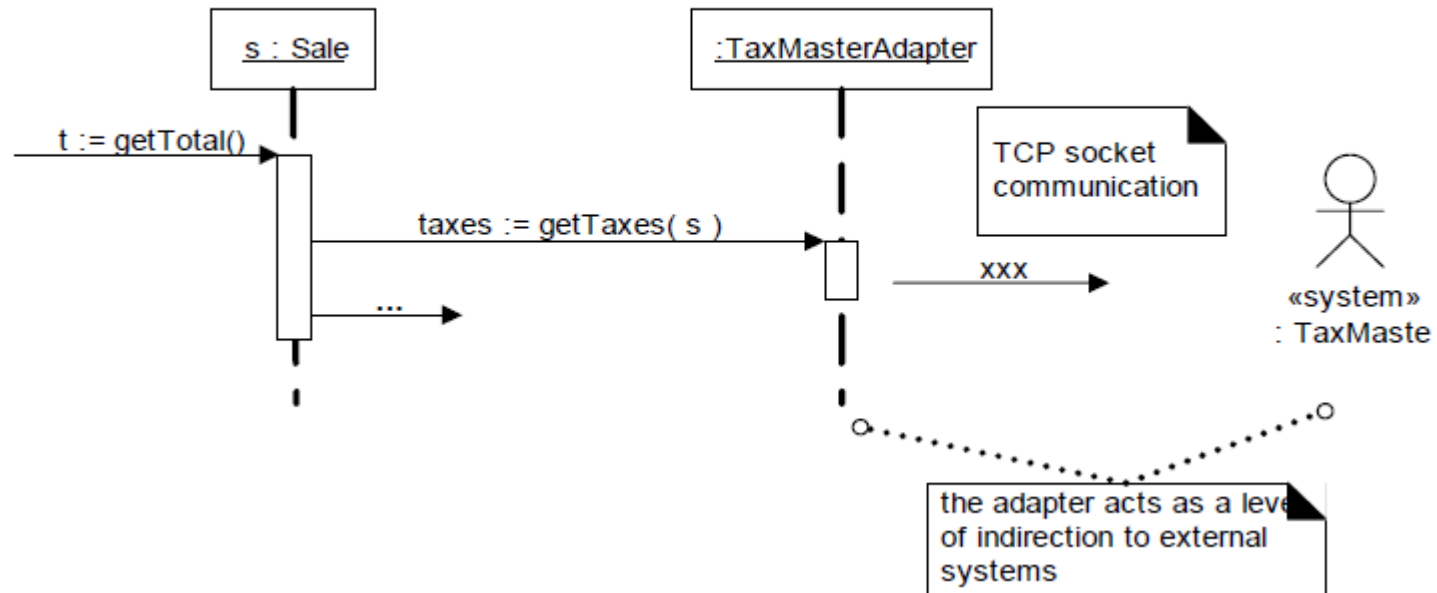
- A class to save cohesion and coupling – a creation of imagination



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



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