Advanced System Analysis and Design

GRASP*: Designing Objects with Responsibilities

* Genenal Responsibility Assignment Software Patterns

Responsibilities and Methods

- The focus of object design is to identify classes and objects, decide what methods belong where and how these objects should interact.
- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - Doing:
    - Doing something itself (e.g. creating an object, doing a calculation)
    - Initiating action in other objects.
    - Controlling and coordinating activities in other objects.
  - Knowing:
    - Knowing about private encapsulated data.
    - Knowing about related objects.
    - Knowing about things it can derive or calculate.

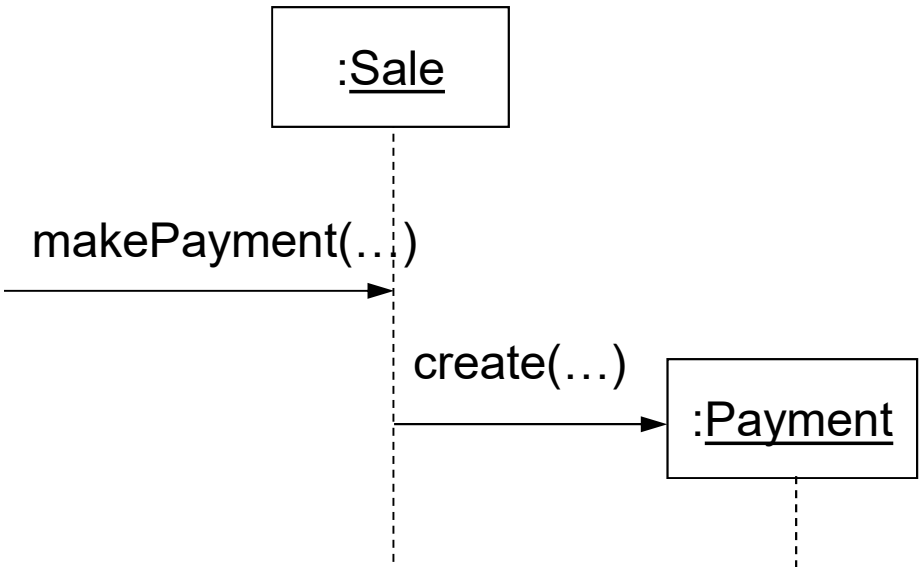
Responsibilities and Methods

- Responsibilities are assigned to classes during object design. For example, we may declare the following:
  - *“a Sale is responsible for creating SalesLineItems*” (doing)
  - *“a Sale is responsible for knowing its total”* (knowing)
- Responsibilities related to “knowing” are often inferable from the Domain Model (because of the attributes and associations it illustrates)

Responsibilities and Methods

- The translation of responsibilities into classes and methods is influenced by the granularity of responsibility.
  - For example, *"provide access to relational databases"* may involve dozens of classes and hundreds of methods, whereas *"create a Sale"* may involve only one or few methods.
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- Methods either act alone, or collaborate with other methods and objects.

Responsibilities and Interaction Diagrams



- Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.
- Sale objects have been given the responsibility to create Payments, handled with the makePayment method.

Patterns

- We will emphasize principles (expressed in patterns) to guide choices in where to assign responsibilities.
- A pattern is a named description of a problem and a solution that can be applied to new contexts; it provides advice in how to apply it in varying circumstances. For example,
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfil it.

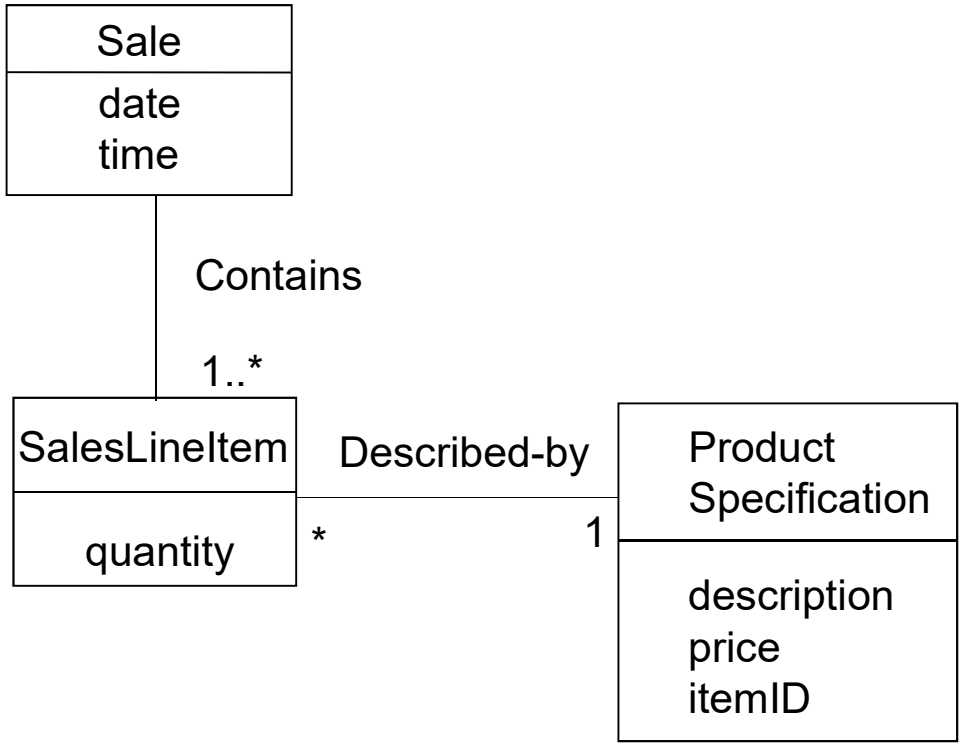
Information Expert (or Expert)

- Problem: what is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.
- In the NextGen POS application, who should be responsible for knowing the grand total of a sale?
- By Information Expert we should look for that class that has the information needed to determine the total.

Information Expert (or Expert)

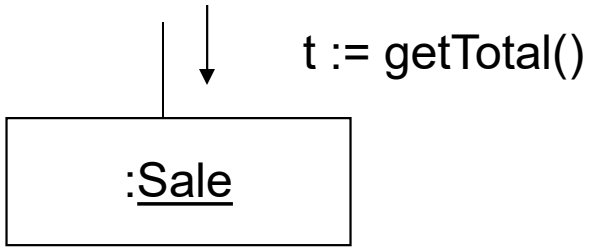
- Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?
- A: Both. Assume there is no or minimal Design Model. Look to the Domain Model for information experts.

Information Expert (or Expert)



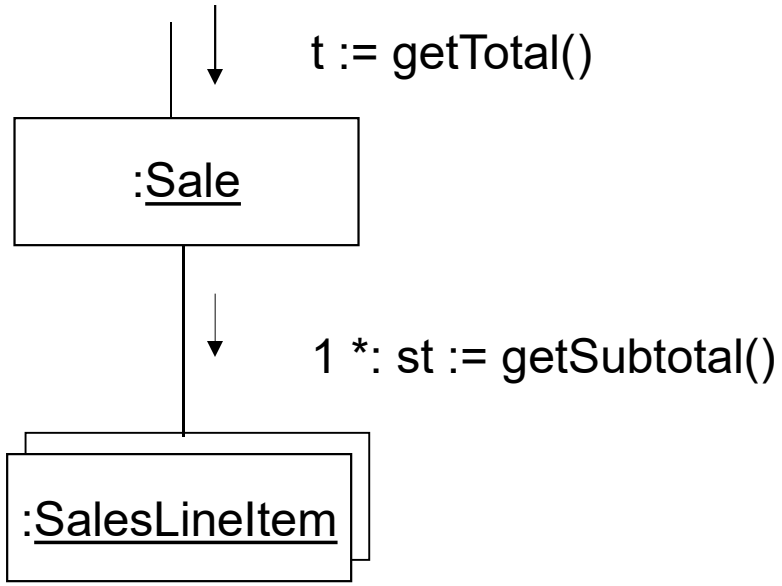
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.

Information Expert (or Expert)



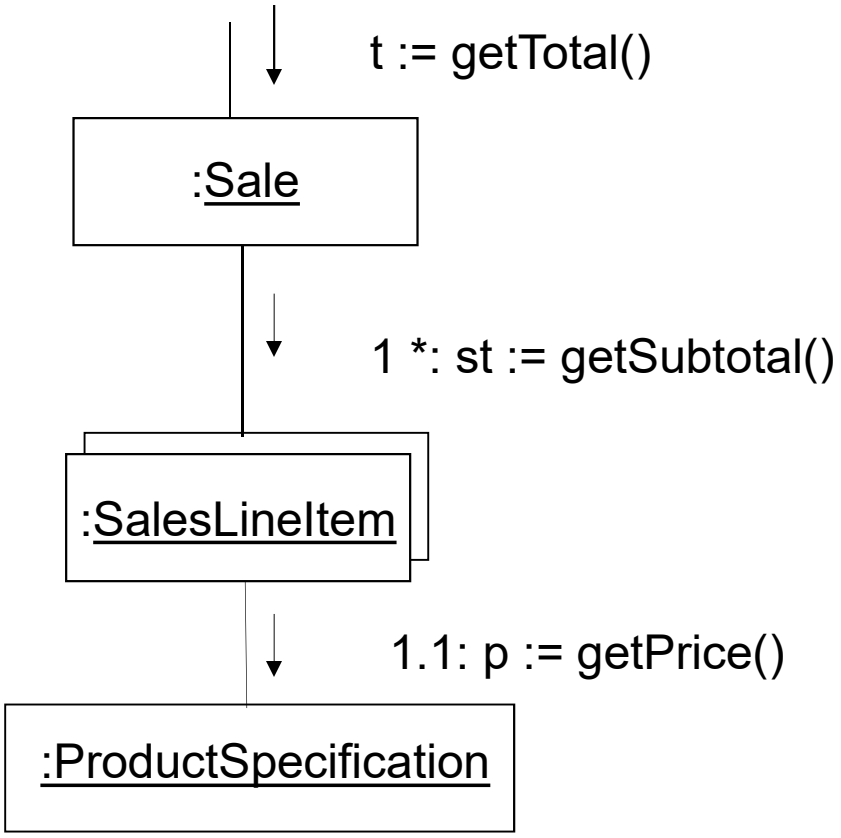
- This is a partial interaction diagram.

Information Expert (or Expert)



- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.

Information Expert (or Expert)



- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.

Information Expert (or Expert)

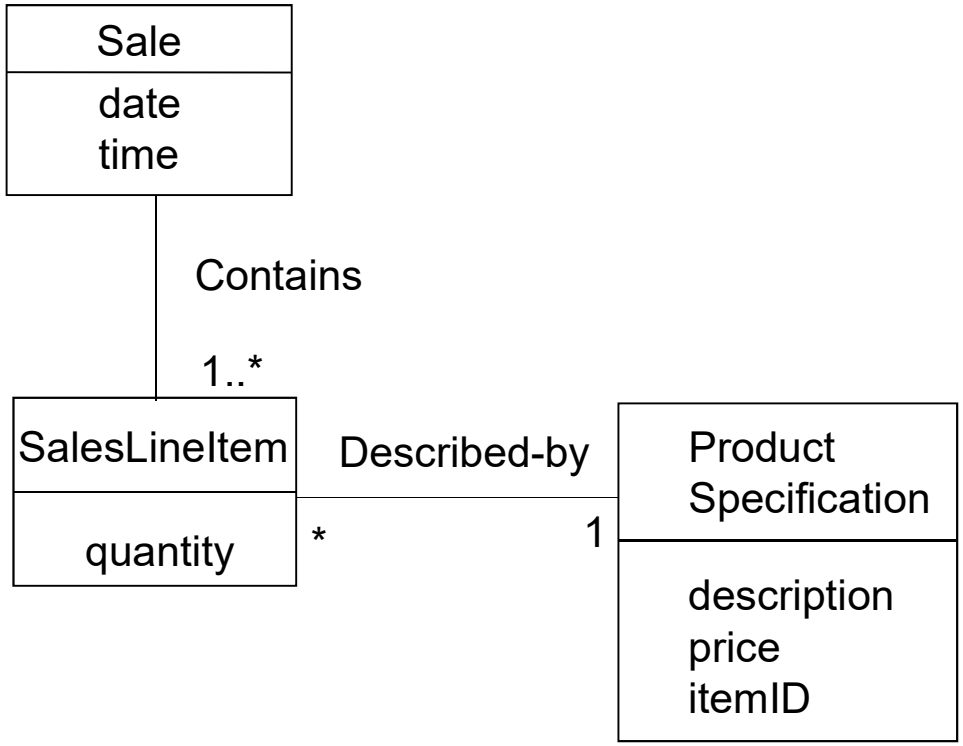
Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial experts" who will collaborate in the task.

Creator

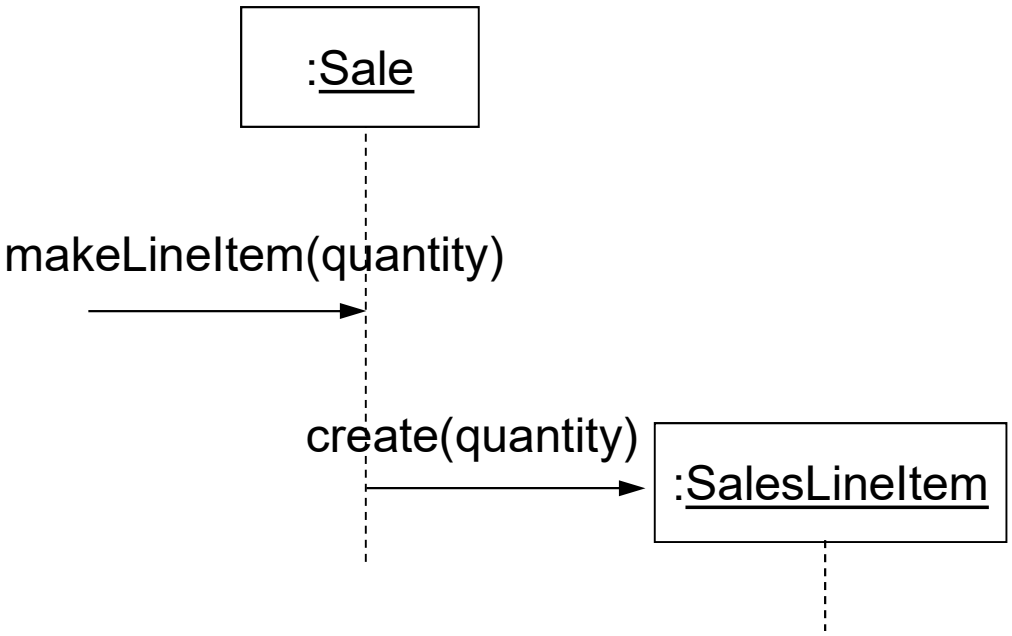
- Problem: Who should be responsible for creating a new instance of some class?
- Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  1. B *aggregates* A objects.
  2. B *contains* A objects.
  3. B *records* instances of A objects.
  4. 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).

Creator



- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.

Creator



- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.

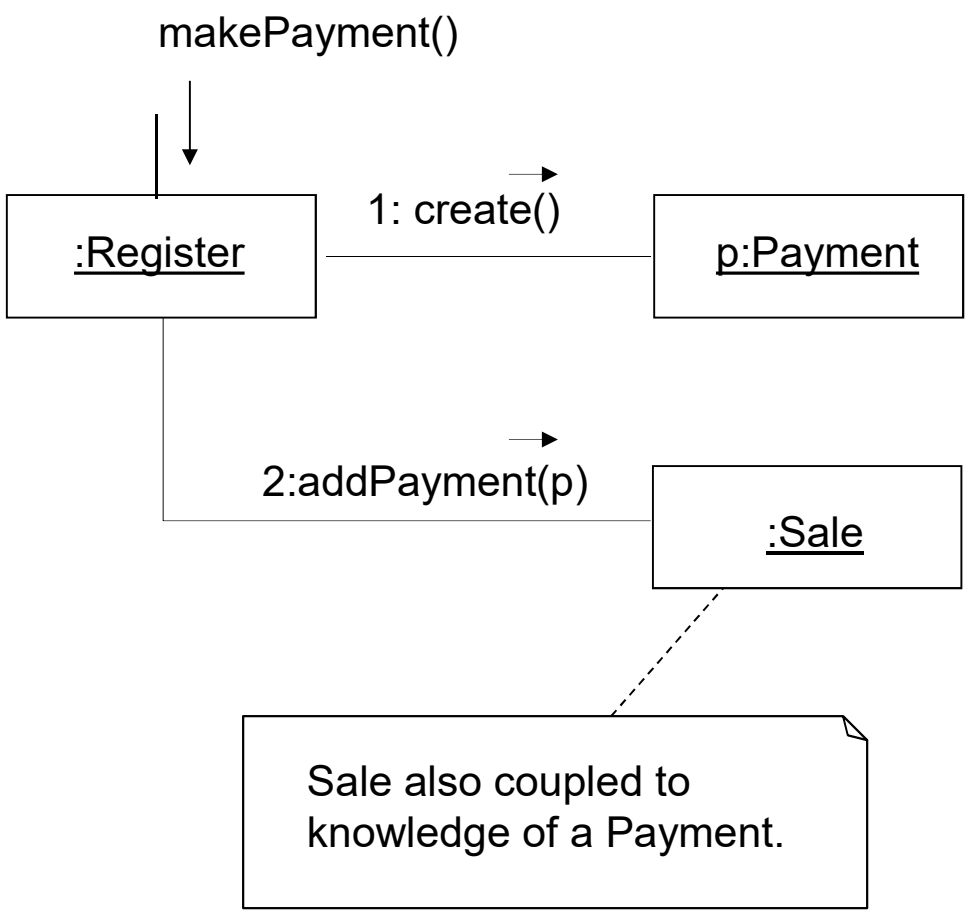
Low Coupling

- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

Low Coupling

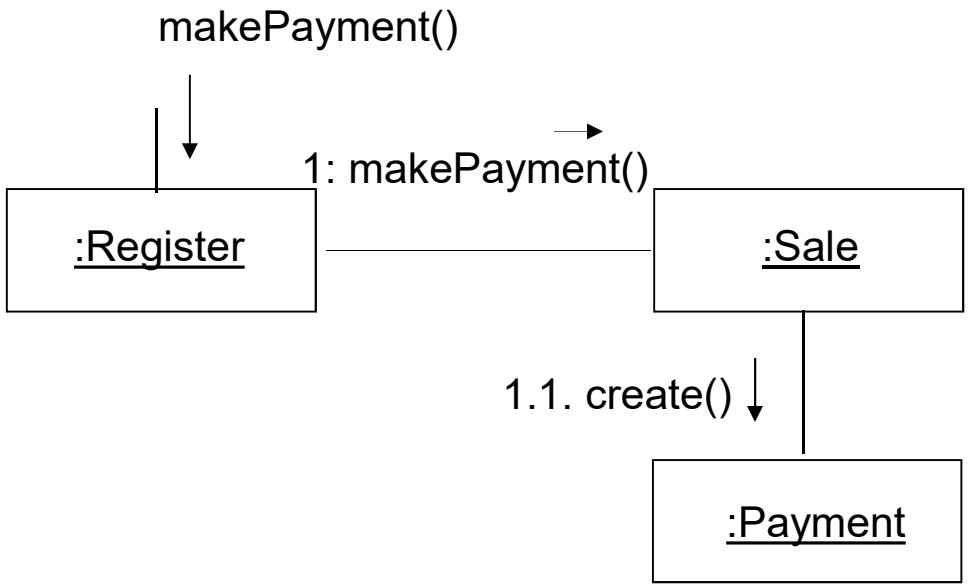
- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.

Low Coupling



- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling



- An alternative solution is to create Payment and associate it with the Sale.
- No coupling between Register and Payment.

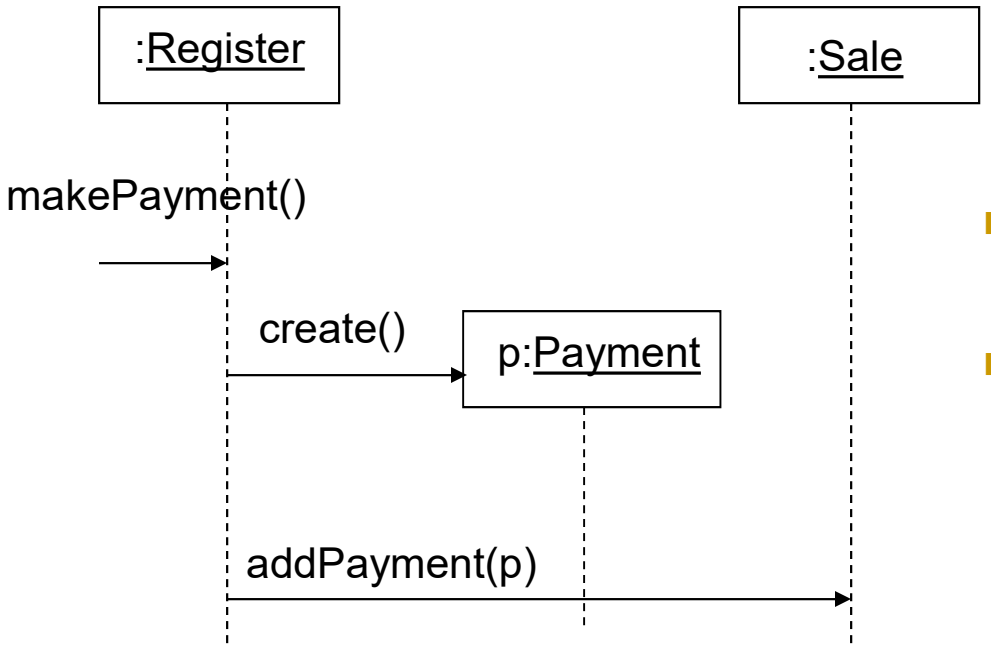
Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.

High Cohesion

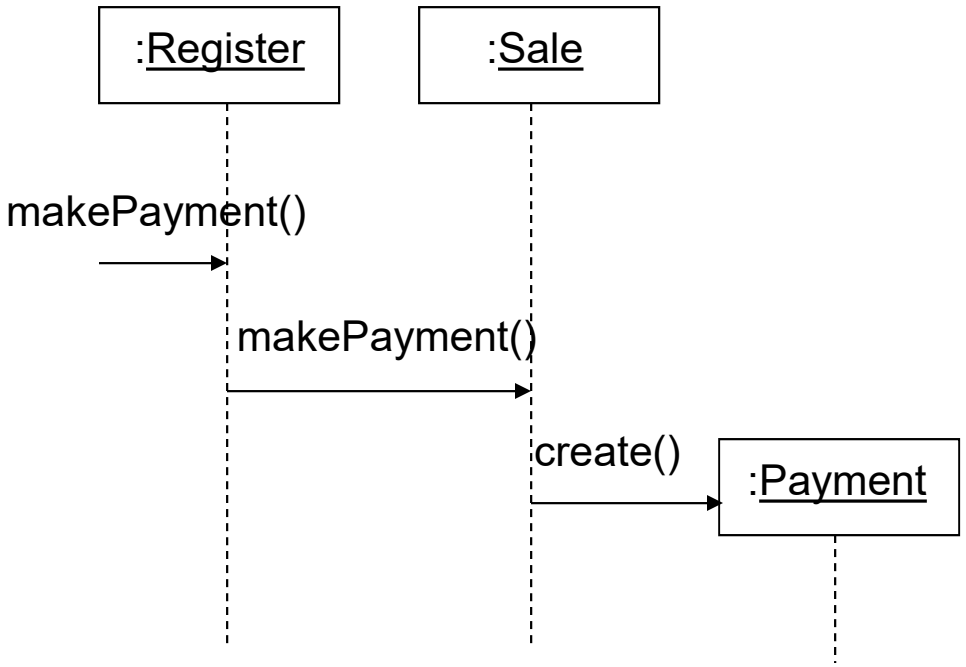
- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

High Cohesion



- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.

High Cohesion



- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.

High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion

1. Very low cohesion: class responsible for many things in many different areas.
   - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.
2. Low cohesion: class responsible for complex task in a functional area.
   - e.g.: a class responsible for interacting with a relational database.

High Cohesion

3. High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.
   - e.g.: a class responsible for one section of interfacing with a data base.

- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.

Controller

- Problem: Who should be responsible for handling an input system event?
- Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of he following choices:
  - Represents the overall system.
  - Represents a use case scenario.
  - A Controller is a non-user interface object that defines the method for the system operation. Note that windows, applets, etc. typically receive events and delegate them to a controller.