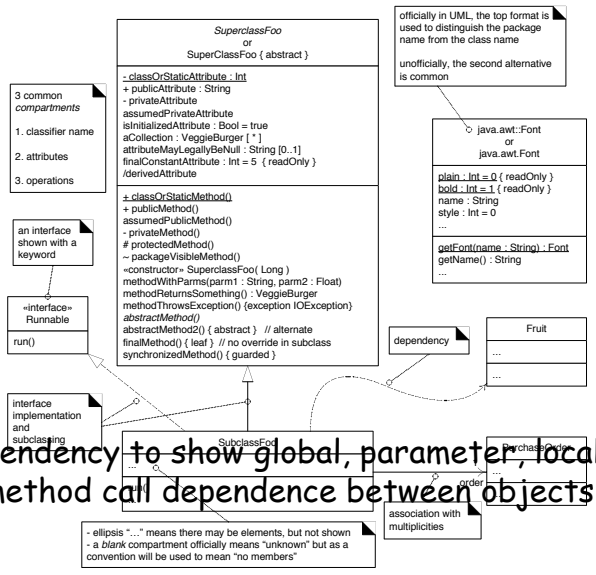
OO Design

- UML Class diagram notation (LAR,Ch 16)
- OO Design Principles
  - Responsibility-driven design
- Useful patterns of design
  - General Responsibility Assignment Software Patterns (GRASP)
  - Expert, Creator, Low Coupling, Controller, High Cohesion

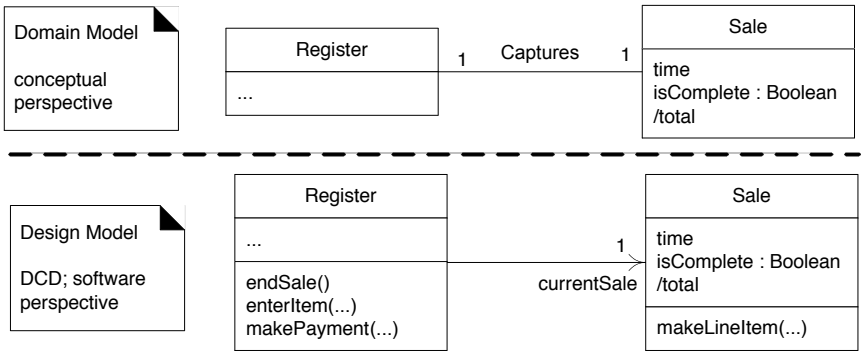
Example (LAR, Fig 16.1 p 250)



Use dependency to show global, parameter, local and/or static-method call dependence between objects

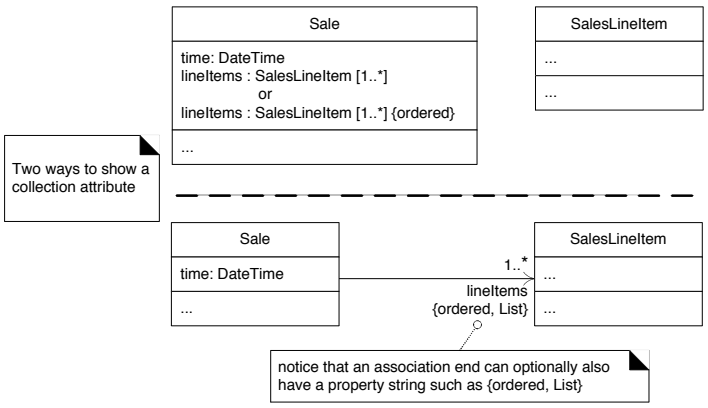
UML Associations (LAR, Fig 16.4, p 253)

visibility:type multiplicity=default{property-string}



UML Attributes with Properties

visibility:type multiplicity=default{property-string}



Goal of Object-Oriented Design

- Produce a design model
  - Two major categories of artifacts
- Interaction diagrams
  - Sequence diagrams
  - Communication diagrams
- Design classes and the corresponding class diagrams
  - Often based on conceptual classes from the domain model

Basics of OO Design

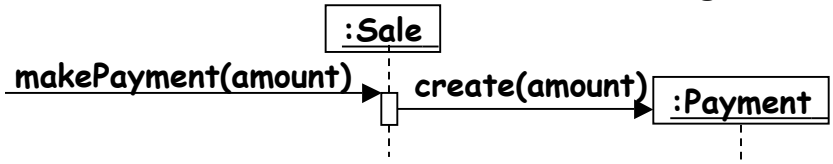
- Core idea: identify responsibilities and assign them to classes and objects
- Responsibilities for doing
  - Create an object, perform calculations, invoke operations on other objects, …
- Responsibilities for knowing
  - Private encapsulated data, related objects, things that can be derived or calculated, …
- Methods fulfill responsibilities

Responsibilities

- **Doing:** "a Sale **object is responsible for creating its** SalesLineItem **objects"**
- **Knowing:** "a Sale **object is responsible for knowing its *total*"**
  - **Often inferable from the domain model**
- **Implemented by operations: act alone or in collaboration with other objects**
  - Sale: **operation** getTotal, **which collaborates with all** SalesLineItem **objects**

Responsibilities & Interaction Diagrams

- **Assignment of responsibilities occurs during the creation of interaction diagrams**
  - **Decisions are encoded in the diagrams**



Sale has the responsibility to handle message **makePayment**; to fulfill this responsibility, **Sale** is responsible for creating a new object **Payment** and for collaborating with it

A Sample Design Pattern

A pattern is a named, well-known problem/solution pair that can be applied in new contexts (LAR, p.279)

Name: *Information Expert*

Solution: Assign a responsibility to the class that has the information needed to fulfill that responsibility

Problem it solves: How do we assign responsibilities to classes and objects?

Example: …

Discussion: when to use it, how to use it, …

Contraindications: when not to use it, …

Some Basic Patterns

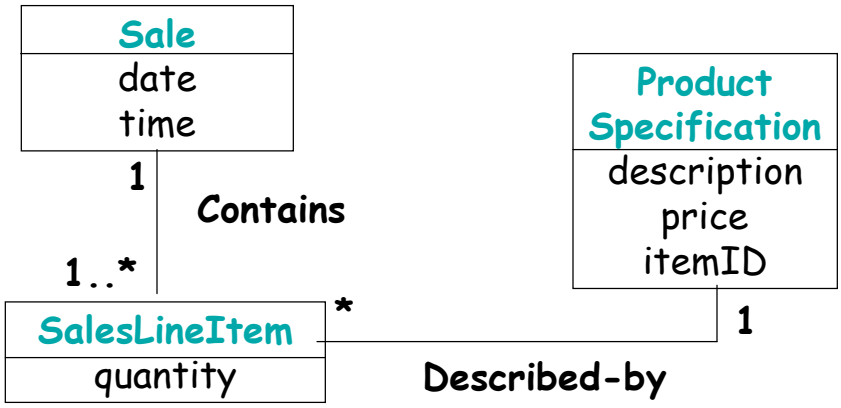
- Focus now on five basic patterns for assigning responsibilities
- *Information Expert*
- *Creator*
- *High Cohesion*
- *Low Coupling*
- *Controller*

Pattern 1: Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill the responsibility
- Example in POS system: *Who should be responsible for knowing the total of a sale?*
- Look for candidates among the existing design classes in the design model
  - If nothing applicable, look at the domain model

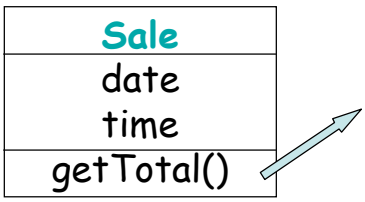
Domain Model

- Suppose we haven't created any design classes yet; look at the domain model



Design Class Sale

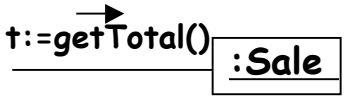
- In the domain model, Sale looks like an expert: it knows about all lineItems, and can compute the sum of their subtotals
- So we create a design class Sale in the design class diagram



Expresses the fact that we gave Sale the responsibility to know its total

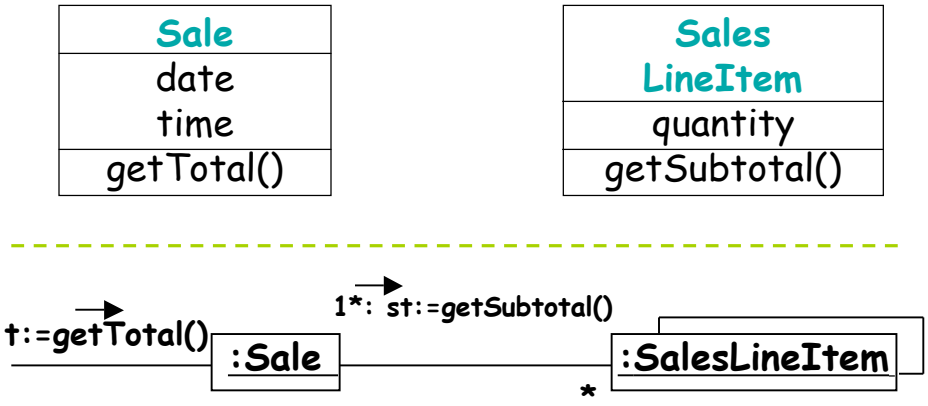
Interaction Diagram

- Now need to create an interaction diagram for the responsibility



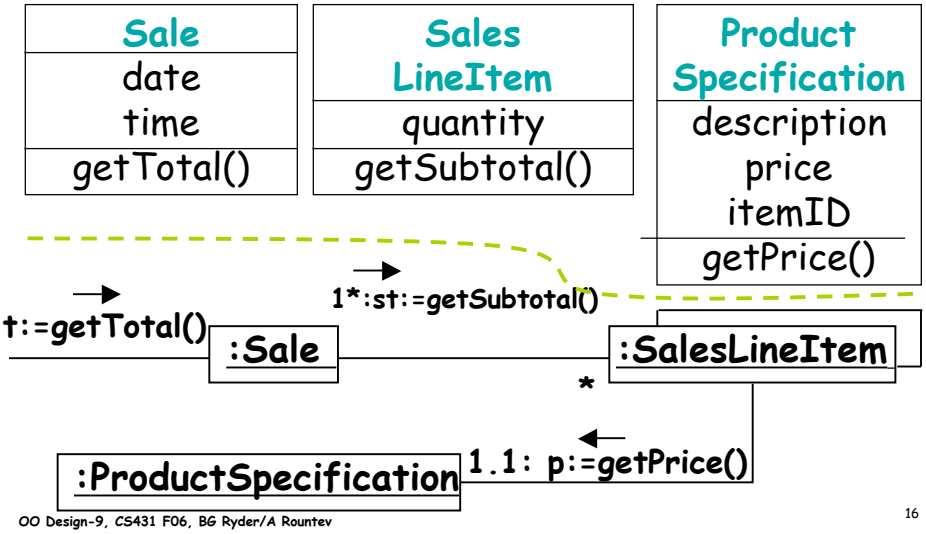
- Need subtotal for each item (i.e. quantity * price)
- *Who is responsible for knowing the subtotal for a line item?*
- SalesLineItem is the expert, so we create a corresponding design class

SalesLineItem in the Design Model



ProductSpec in the Design Model

- ***Who is responsible for knowing item price?***



Assigned Responsibilities

- To know a Sale's total, three responsibilities were assigned
  - Sale: knows sale total
  - SalesLineItem: knows subtotal for line item
  - ProductSpecification: knows product price
- Interaction diagram: shows the dynamic behavior
- Design classes were created as necessary
  - Class diagram shows the static structure of the software

Wasn't That Solution Obvious?

- Consider an alternative (Class X is responsible)
  - Some object X asks the Sale for all of its SalesLineItem objects
  - X asks each line item for the quantity and the ProductSpecification
  - X asks each specification for the price
  - X computes sum(quantity*price)
- X has several responsibilities related to data that lives in other objects
- Not uncommon for OO novices to do this

Summary

- *Information Expert:* objects do things related to the information they have, often requiring collaboration among objects
- Information hiding: objects use their own info to fulfill tasks
  - Low coupling, more robust and maintainable system, better opportunities for reuse

Summary, cont.

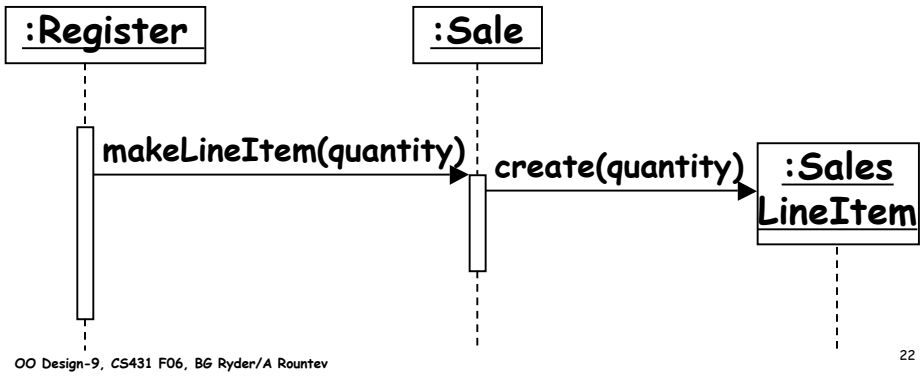
- Contraindications: may create problems with coupling and cohesion
  - E.g., should a Sale be responsible for saving itself to a database?
    - Increased coupling: then the code in Sale depends on DB services (SQL, JDBC, etc)
    - Duplicated code: similar DB logic will be duplicated in many persistent classes (bad for maintenance)

Pattern 2: Creator

- Assign class B the responsibility of creating an instance of class A if:
  - B aggregates A objects
    - Whole-Part; Assembly-Part (e.g. Body-Leg)
  - B contains A objects
  - B records A objects
  - B closely uses A objects
  - B has initializing data that will be passed to a new A object
    - B is an expert w.r.t. creating A objects

Example

- Who should be responsible for creating a SalesLineItem?
  - Sale aggregates SalesLineItem objects



Summary

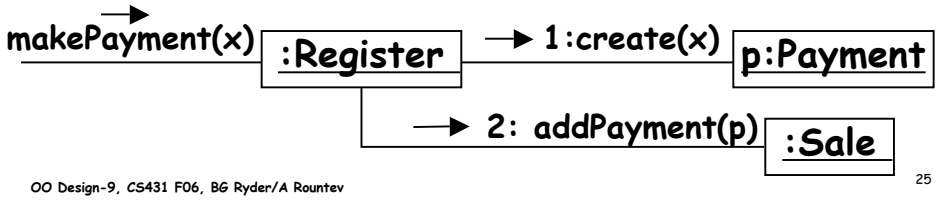
- The creating object will have to be connected with the new object anyway, so we just add some extra work there
  - If some other object were to do it, increases coupling, at cost of reduced maintainability/reuse
- E.g.: the enclosing container or recorder is a natural candidate for a creator
- Contraindications: complex creation
  - e.g. using recycled objects for performance
    - better to use the *Factory* pattern (more later)

Pattern 3: Low Coupling

- Assign responsibilities so that coupling remains low
  - Goal: few dependences, low change impact, increased possibilities for reuse
- Coupling: measure of how strongly one class is connected to, has knowledge of, or relies on other classes
  - Changes in related classes force local changes
  - Harder to understand classes in isolation
  - Harder to reuse because a class requires the presence of other classes it depends on

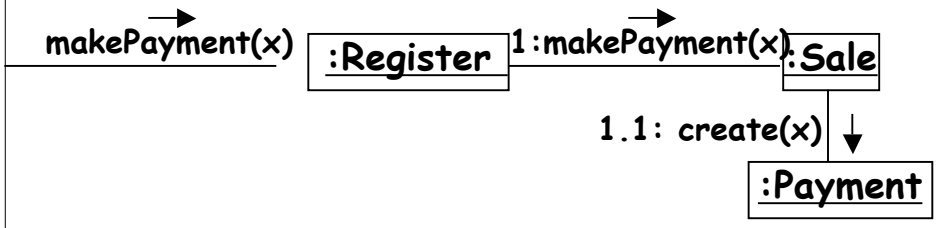
Example

- Classes Payment, Register, Sale
- Need to create a Payment instance and associate it with the Sale
  - *Which class creates Payment instances?*
- Register has the info necessary to create a payment, so we can use *Creator*



Example, cont.

- But this couples class Register to knowledge of the Payment class
- Alternative



- Basic idea: Sale will need to know about Payment, so this coupling is already there, but Register does not need to know

Examples of Coupling

- Class A has an attribute (field) of class B
- An instance of A calls an instance of B
- A has a method that references instances of B
  - local variable/parameter/return value is a reference (i.e., pointer) to a B object
- A is a direct or indirect subclass of B

Summary

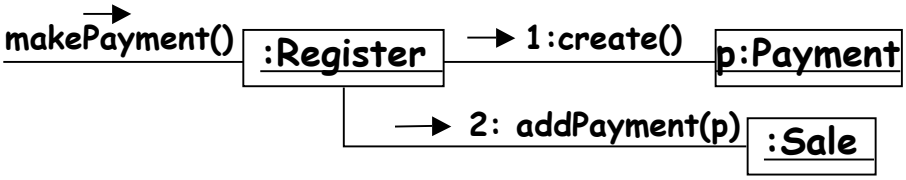
- Low coupling: general principle for design
  - Should be considered together with other patterns, and some trade-offs may be needed
- Classes that are inherently generic in nature and have high probability of reuse should have especially low coupling
- Some degree of coupling is necessary, the goal is to avoid unnecessary coupling

Pattern 4: High Cohesion

- Cohesion: how strongly related and focused are the responsibilities of a class
- A low-cohesion class does unrelated things, or just does too many things
- Problem: responsibilities should have been delegated to other classes

Example

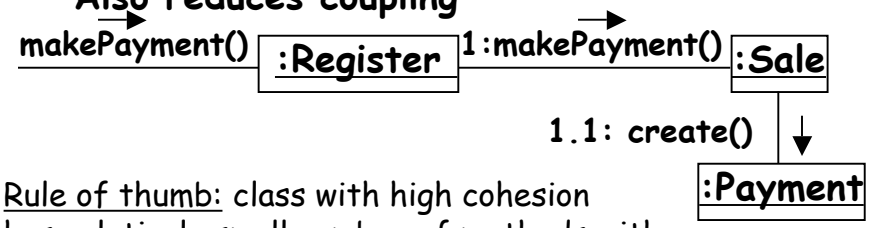
- *Who creates Payment objects?*



- If Register does the work for all system events, it will become bloated and not cohesive

Example

- Our better solution: delegate Payment creation to Sale
  - Higher cohesion for Register
  - Also reduces coupling



Rule of thumb: class with high cohesion has relatively small number of methods with highly related functionality, and does not do too much work (LAR, p 317)

Degrees of Cohesion

- Very low: a class is solely responsible for many tasks in different areas
  - E.g. RDB-RPC-Interface for interacting with relational databases (RDB) and handling of remote procedure calls (RPC)
- Low: sole responsibility for a complex task in one area
  - E.g. RDBInterface for interacting with relational databases: still too much code

Degrees of Cohesion

- Moderate: lightweight and sole responsibilities in a few different areas
  - Areas logically related to the class but not to each other
    - Company class that is responsible for employee and financial info
- High: moderate responsibilities in one area
  - Collaborates with other classes
  - E.g., RDBInterface, but partially responsible
    - Collaborates with a dozen other classes related to RDB access

Benefits

- Clarity and ease of comprehension
- Maintenance and enhancements are simplified
- Often results in low coupling
- Cohesive classes are easier to reuse
- Contraindications:
  - Distributed server objects need to be larger, w/ coarse-grain operations
    - Reduces the number of remote calls
  - To simplify maintenance by an expert developer

System Events

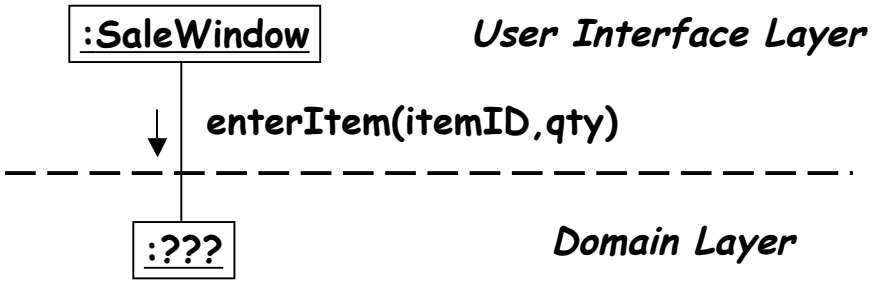
- *Who should be responsible for handling an input system event?*
  - An event generated by an external actor
  - E.g., word processor: "spell check" button triggers event "perform spell check"
- System sequence diagrams from analysis: conceptual class System handles events
- In design: handling by instances of controller classes

Pattern 5: Controller

- Facade controller: a class representing the entire system or device
- Use case controller: a class representing a use case within which the event occurs
  - e.g. XyzHandler, XyzCoordinator, XyzSession
    - Xyz=name of the use case
  - Handles all system events in the use case

Example

- System events in POS system
  - endSale(), enterItem(), makeNewSale(), makePayment(), …
- Typically generated by the GUI



Controller Classes

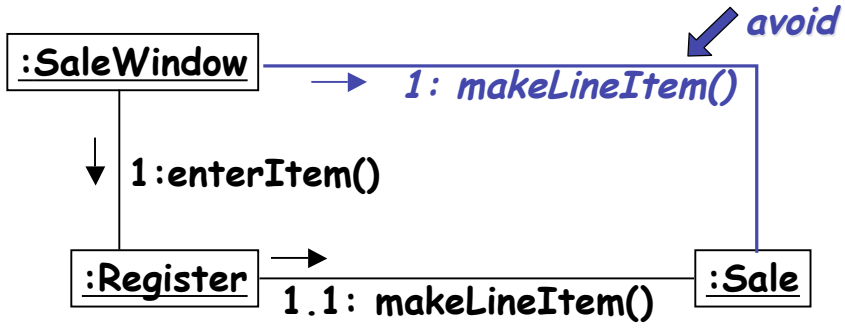
- Entry points into the domain layer
  - Isolate the internals of the domain layer
- Facade controller: entire system/device
  - POS_System, Register
- Use case controller: handler for all events in a use case
  - ProcessSaleHandler, ProcessSaleSession
- Can track the state of interactions (e.g., order of events)

Using Controller Classes

- Facade controller: used when there are not "too many" system events
  - Avoid "bloated" controllers (e.g., too many responsibilities, has too much data)
- Use-case controllers
  - Good when there are many system events
  - Several manageable controller classes
  - Tracking of the state of the current use-case scenario: e.g. to enforce sequencing constraints

Interface Layer

- Interface objects (windows, etc.) should not handle system events
  - The domain layer has the application logic
  - Good for reuse of application logic and UI



Client/Server Applications

- GUI + controllers on the client side
  - GUI sends requests to the controller
    - In the same OS process
  - Controller forwards the request to a remote server
- Systems with Web interface
  - Server-side use-case controllers
    - e.g., for Enterprise Java Beans: often there is a session bean per use case