Introduction to GRASP:
Assigning Responsibilities to Objects

CS 4354
Summer II 2015

Jill Seaman

Object Analysis & Design in the textbook

- Chapter 5 Analysis activities: from use cases to objects
  - ✦Gives good guidelines for identifying and assigning the following:
    - – objects (classes)
    - – attributes
    - – associations, aggregations, inheritance relationships
    - – Good start to a class diagram representing the domain model
  - ✦But what about operations?
    - – Sequence diagrams are good tools to explore interactions and operations
    - – But little advice is given on how to decide who does what.

The design of behavior

- What methods go in what classes? How should objects interact?
  - ✦These are critical questions in the design of behavior.
  - ✦Poor answers lead to abysmal, fragile systems with low reuse and high maintenance.

Responsibility-Driven Design

- Assigns responsibilities to classes
- Methods are implemented to fulfill responsibilities of the given class.
- Methods may act alone or in collaboration to fulfill their obligations.

- Responsibilities of classes:
  - ✦Knowing: about attributes, related classes, computed values
  - ✦Doing: Calculating, coordinating, creating, controlling
- Responsibilities come from the use cases: If “the system does X”, then what class is responsible for carrying out X?

GRASP Patterns

GRASP

- General Responsibility Assignment Software Patterns.
- These are well-known best principles for assigning responsibilities.
- Nine core principles that object-oriented designers apply when assigning responsibilities to classes and designing message interactions.
  - We will look at 5 of these 9 principles
- Can be applied during the creation of sequence diagrams, or even during implementation.
- After or in tandem with developing the domain model.

Patterns

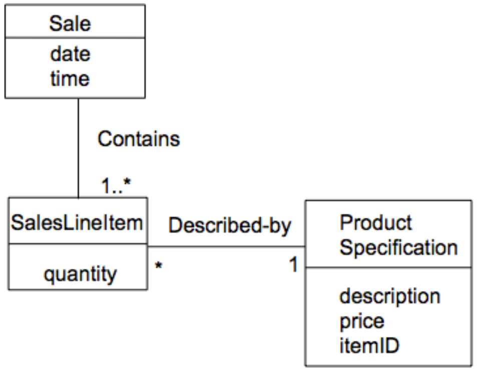
- Named description of a problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations, and discussion of its trade-offs.
- Notable benefits of patterns:
  - Simplifying: provides a named, generally understood building block
    - Facilitates communication
    - Aids thinking about the design
  - Accelerates learning to not have to develop concepts from scratch

Pattern: **Information Expert**

- Problem: What is most basic, general principle of responsibility assignment?
- Solution: Assign a responsibility to the object that has the information necessary to fulfill it.
  - "That which has the information, does the work."

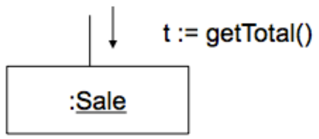
- In a "Point of Sale" (think: cash register) application, who should be responsible for knowing the grand total of a sale?
- By Information Expert we should look for that class that has the information needed to determine the total.

POS domain model



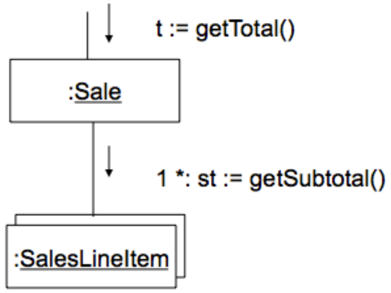
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.

POS Information Expert



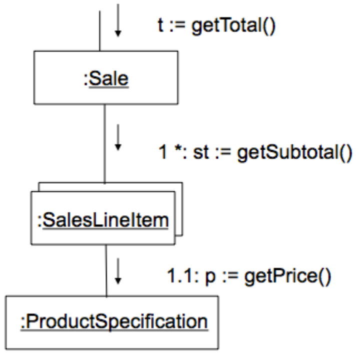
- This is a partial interaction diagram.
- It’s a variation of a sequence diagram.

POS Information 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.

POS Information Expert



- To fulfill 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.

POS Information Expert

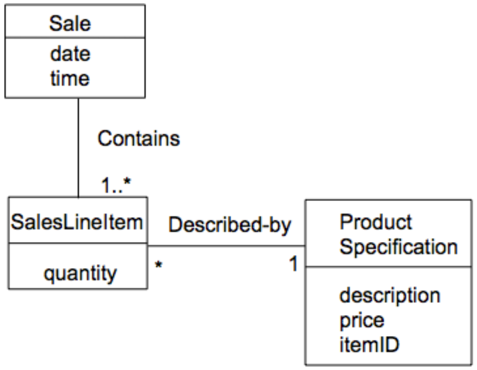
Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

- To fulfill 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.

Pattern: **Creator**

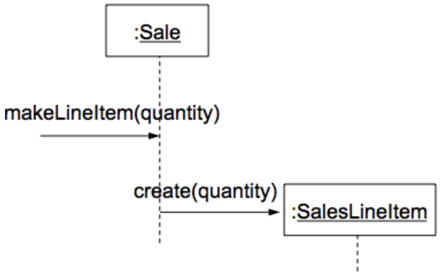
- Problem: Who should be responsible for creating a new instance of some class?
- Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - ✦B aggregates A objects.
  - ✦B contains A objects.
  - ✦B records instances of A objects.
  - ✦B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).
- The more, the better.

POS domain model



- 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.

POS Creator



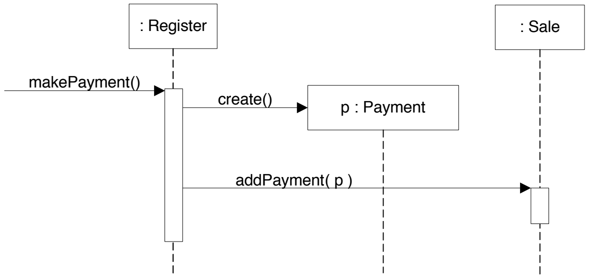
- This assignment of responsibilities requires that a makeLineItem method be defined in Sale.

Pattern: **Low Coupling**

- **Coupling** (in a class diagram) is a measure of how strongly one class is connected to, has knowledge of, or relies on other classes.
- 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.

POS: Low Coupling

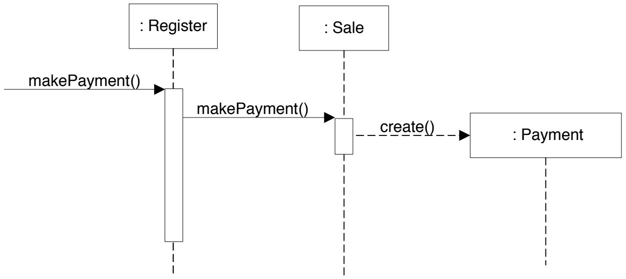
- Which class should be responsible for creating a Payment and associating it with a sale?



- Since Register records a payment (in real life), it could be Register, by the Creator pattern
- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- This assignment of responsibilities couples the Register class to knowledge of the Payment class.

POS: Low Coupling

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

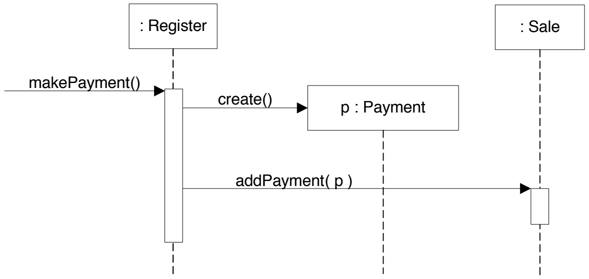


Pattern: **High Cohesion**

- **Cohesion** (in a class diagram) is a measure of how strongly related and focused the responsibilities of a class 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.

POS High Cohesion

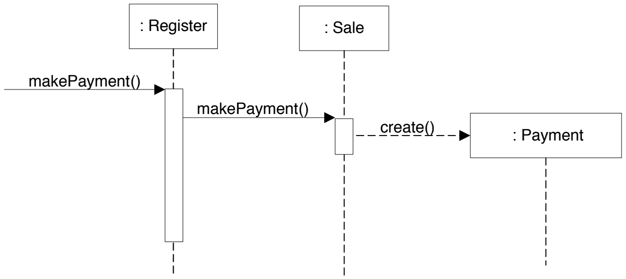
- Let’s compare the same two examples as before with respect to cohesion:



- Since Register records a payment (in real life), it could be Register, by the Creator pattern
- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- Register may become bloated if it is assigned more and more system operations.

POS: High Cohesion

- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- No class has too much work (good delegation).
- This design supports high cohesion and low coupling.



: Register

: Sale

: Payment

makePayment()

makePayment()

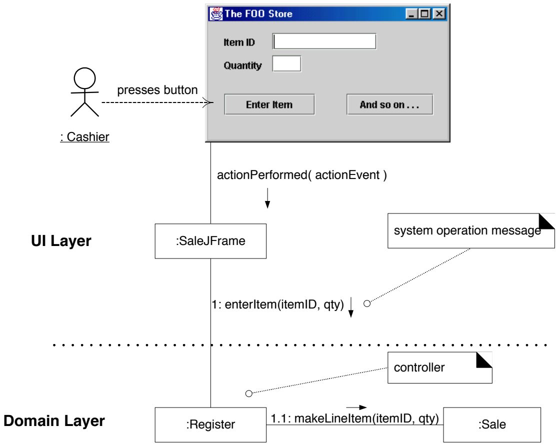
create()

Pattern: **Controller**

- What class should handle system event messages (such as input from the user/user interface)?
- Solution: Choose a class whose name/job suggests:
  - ✦The overall “system,” device, or subsystem
  - ✦OR, represents the use case scenario or session
- Recall: during analysis,we identified three types of objects:
  - ✦Entity Objects: persistent information tracked by system (domain objects)
  - ✦Boundary Objects: represent the interface between the actors and the system
  - ✦Control Objects: are in charge of realizing use cases
- Recall: MVC architectural pattern: the Controller component

POS: Controller

- In this example, the Register object (a controller) handles the input event.



: Cashier

presses button

actionPerformed(actionEvent)

UI Layer

:SaleJFrame

system operation message

1: enterItem(itemID, qty)

controller

Domain Layer

:Register

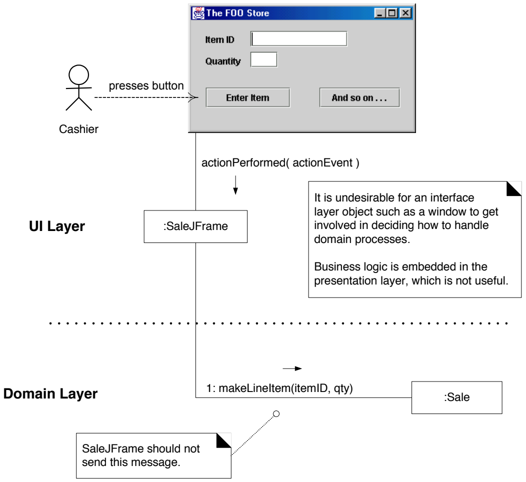
1.1: makeLineItem(itemID, qty)

:Sale

POS: Controller

- In this example, SaleJFrame, a UI (boundary) object handles the input event

Don’t want the UI objects tightly coupled with the entity objects (Sale)



Cashier

presses button

actionPerformed(actionEvent)

UI 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.

Summary of Introduction to GRASP

- 5 principles for deciding how to assign responsibility (behavior) to classes:
  - ✦Information Expert
  - ✦Creator
  - ✦Low Coupling
  - ✦High Cohesion
  - ✦Controller
- These decisions are made during analysis and/or object design.
- These decisions are made (initially) when designing the sequence diagrams from the use cases (deciding which messages are handled by which objects)

Example: Object-Oriented Analysis & GRASP

- This example is based on the Inventory system described in Assignment 2.
- I will treat “Process Sale” as a use case for the Fulfillment Specialist actor. (I’m not going to address any other use cases).
- Note: I am not going to consider “Boundary Objects”: I am going to ignore the User Interface, and assume that the actor interacts directly with the Controller Object.

Step 1: Use case for process sale

Use Case Name	Process Sale
Participating Actors	Initiated by Fulfillment Specialist
Flow of Events	1. The Fulfillment Specialist activates the “Process Sale” function. 2. The Fulfillment Specialist enters the following values: the sku of the sold item, the quantity that were sold, and the cost to ship all of the items. 3. The system finds the Product with the given sku in the Inventory. 4. The system decreases the quantity of the Product by the given quantity that were sold. 5. The system computes the total price, shipping credit, commission, and profit, and outputs these values to the FulfillmentSpecialist.
Exceptional Flow of Events	1. If there is no product in the inventory with the given sku, the System outputs an error message and aborts the operation. 2. If the quantity of the Product in the inventory is not greater than the quantity sold, the System outputs an error message and aborts the operation.
Entry Condition	The Fulfillment Specialist has started the system.
Exit Condition	The Fulfillment Specialist has received the computed statistics, and the quantity of the Product has been decreased by the quantity sold.

Step 2: Entity, boundary, and control objects

- Entity objects:

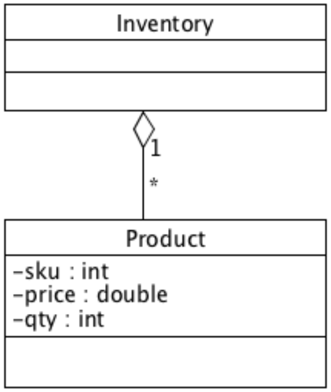
Product	The item that was sold
Inventory	The list of Products sold by the company

- Boundary objects: Ignoring these for this assignment.
- Control objects (Note we did not have this class in Assignment 2):

ProcessSaleControl	Manages the processSale reporting function. This object will coordinate the work done by the system.

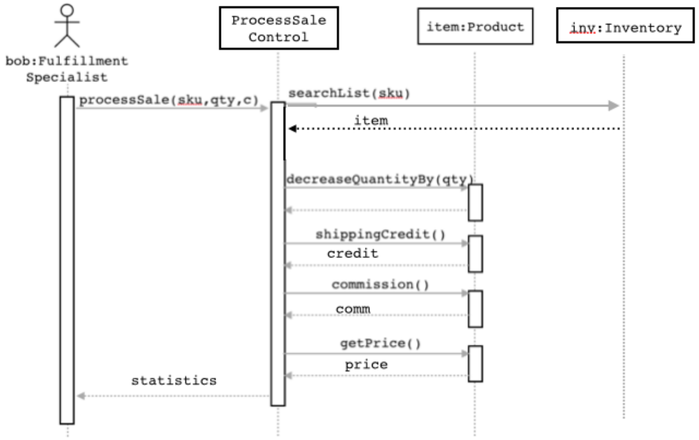
Step 3: Class diagram with attributes, associations

- The Product price is needed to compute the desired statistics
- The Product qty will be updated by the use case.



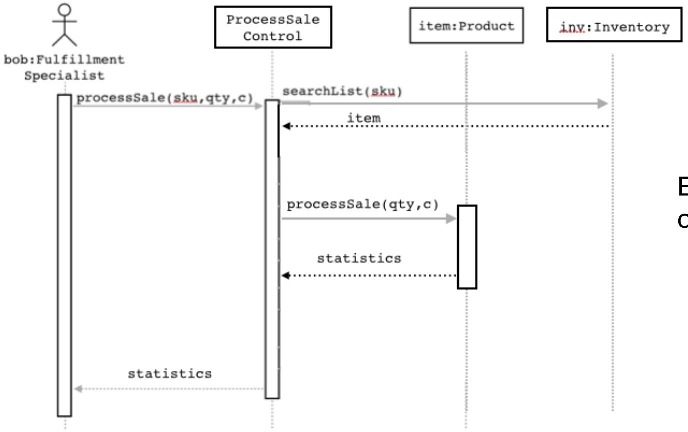
Step 4: Sequence diagram for process sale, v.1

- Forwards engineering (this is not like the code I wrote).
- Controller does all the work (low coherence, bad)



Step 4: Sequence diagram for process sale, v.2

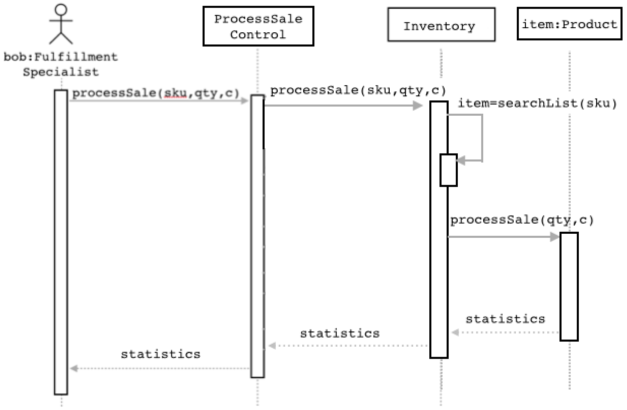
- Now the Controller dispatches the work to Inventory and Product.
- The Product has all the information to compute the statistics, so now it calculates the statistics (by Information Expert)



But what about coupling?

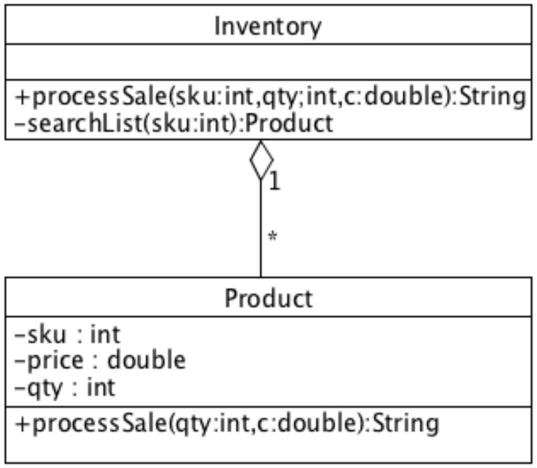
Step 4: Sequence diagram for process sale, v.3

- Now the Controller dispatches the work to Inventory, who delegates to the Product.
- Low coupling, high cohesion, and information expert is applied



Step 5: Add operations to class diagram

- searchList can be private now (and returns Product)


Inventory
+processSale(sku:int,qty;int,c:double):String
-searchList(sku:int):Product
1
*
Product
-sku : int
-price : double
-qty : int
+processSale(qty:int,c:double):String

Explain GRASPatterns used

- Inventory.processSale(sku,qty,c): By high cohesion, Inventory should be responsible to process the sale (so ProcessSaleControl doesn't have to do all of it).

- Product.processSale(qty,c): By information expert, Product has the information needed to compute the statistics (and update the quantity) so it will do that work.

- Inventory.processSale(sku,qty,c): By low coupling, Inventory should find the product AND send the processSale message to it (so ProcessSaleControl doesn't have to talk to the Product).