Principles of Software Construction: Objects, Design and Concurrency

Object-Oriented Design: Assigning Responsibilities

15-214

toad

Christian Kästner Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

With slides from Klaus Ostermann

Object-Oriented Design

- "After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements."
- But how?
  - How should concepts be implemented by classes?
  - What method belongs where?
  - How should the objects interact?
  - This is a critical, important, and non-trivial task

Responsibilities

- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - knowing
  - doing
- Doing responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

Design Goals - Summary

- 5 design goals
  - Design for division of labor
  - Design for understandability and maintenance
  - Design for change
  - Design for reuse
  - Design for robustness
- 5 design strategies (for now)
  - Explicit interfaces (clear boundaries)
  - Information hiding(hide likely changes)
  - Low coupling (reduce dependencies)
  - High cohesion (one purpose per class)
  - Low repr. gap (align requirements and impl.)

GRASP Patterns

- GRASP = General Responsibility Assignment Software Patterns (introduced by Craig Larman)
- Patterns of assigning **responsibilities**
  - reason about design trade-offs when assigning methods and fields to classes
- The GRASP patterns are a *learning aid* to
  - help one understand essential object design
  - apply design reasoning in a methodical, rational, explainable way
  - lower level and more local reasoning than most *design patterns*

Patterns/Principles aid Communication

Fred: "Where do you think we should place the responsibility for creating a SalesLineItem? I think a Factory."

Wilma: "By Creator, I think Sale will be suitable."

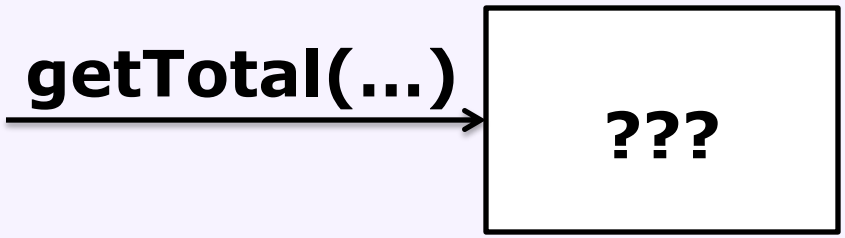
Fred: "Oh, right - I agree."

Nine GRASP Pattern:

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

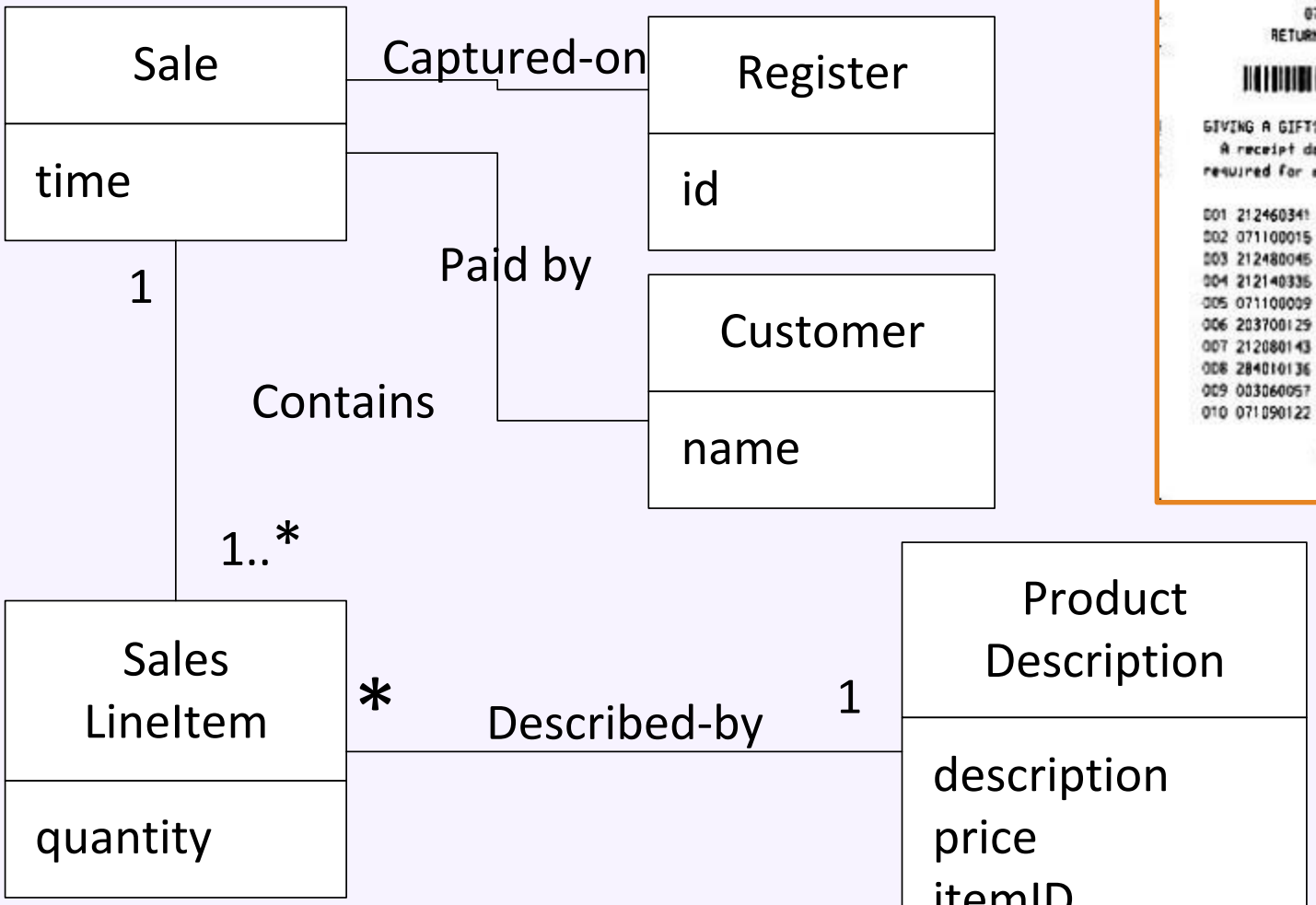
Information Expert (GRASP Pattern 1)

- Who should be responsible for **knowing** the grand total of a sale?



Information Expert (GRASP Pattern 1)

- Who should be responsible for **knowing** the grand total of a sale?

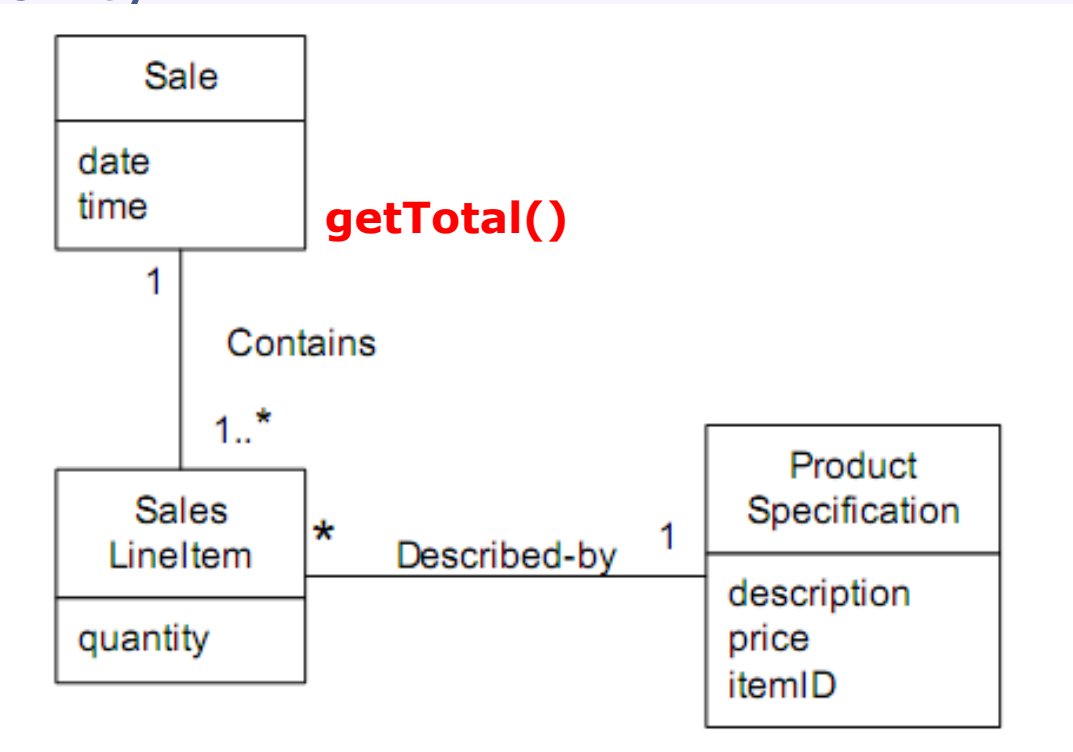


Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: **Assign a responsibility to the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Design Classes (Software Classes) instead of Conceptual Classes
  - If Design Classes do not yet exist, look in Domain Model for fitting abstractions (-> low representational gap)

Information Expert

- What information is needed to determine the grand total?
  - Line items and the sum of their subtotals
- *Sale* is the information expert for this responsibility.

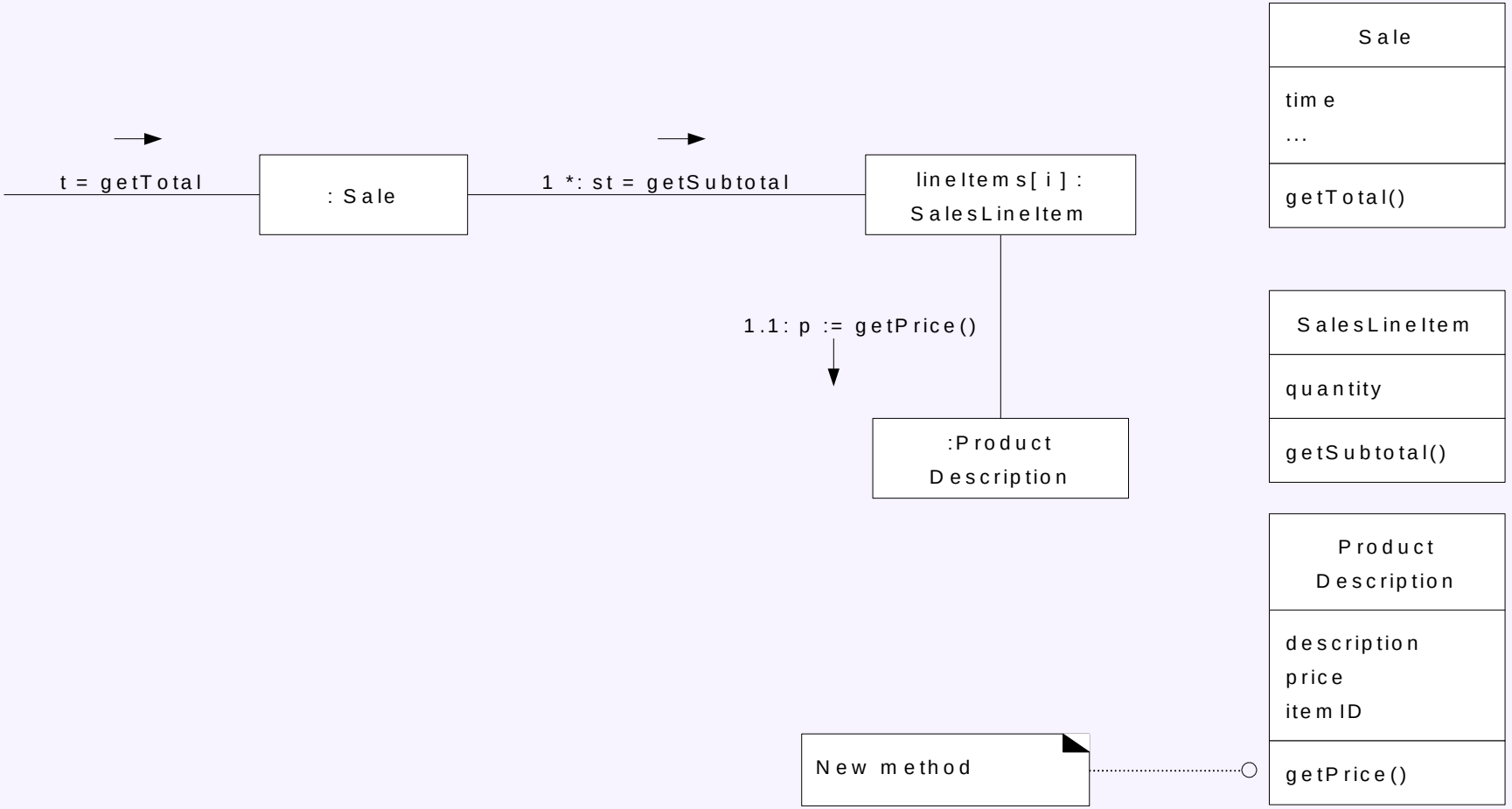


Information Expert

- To fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Information Expert



Information Expert -> "Do It Myself Strategy"

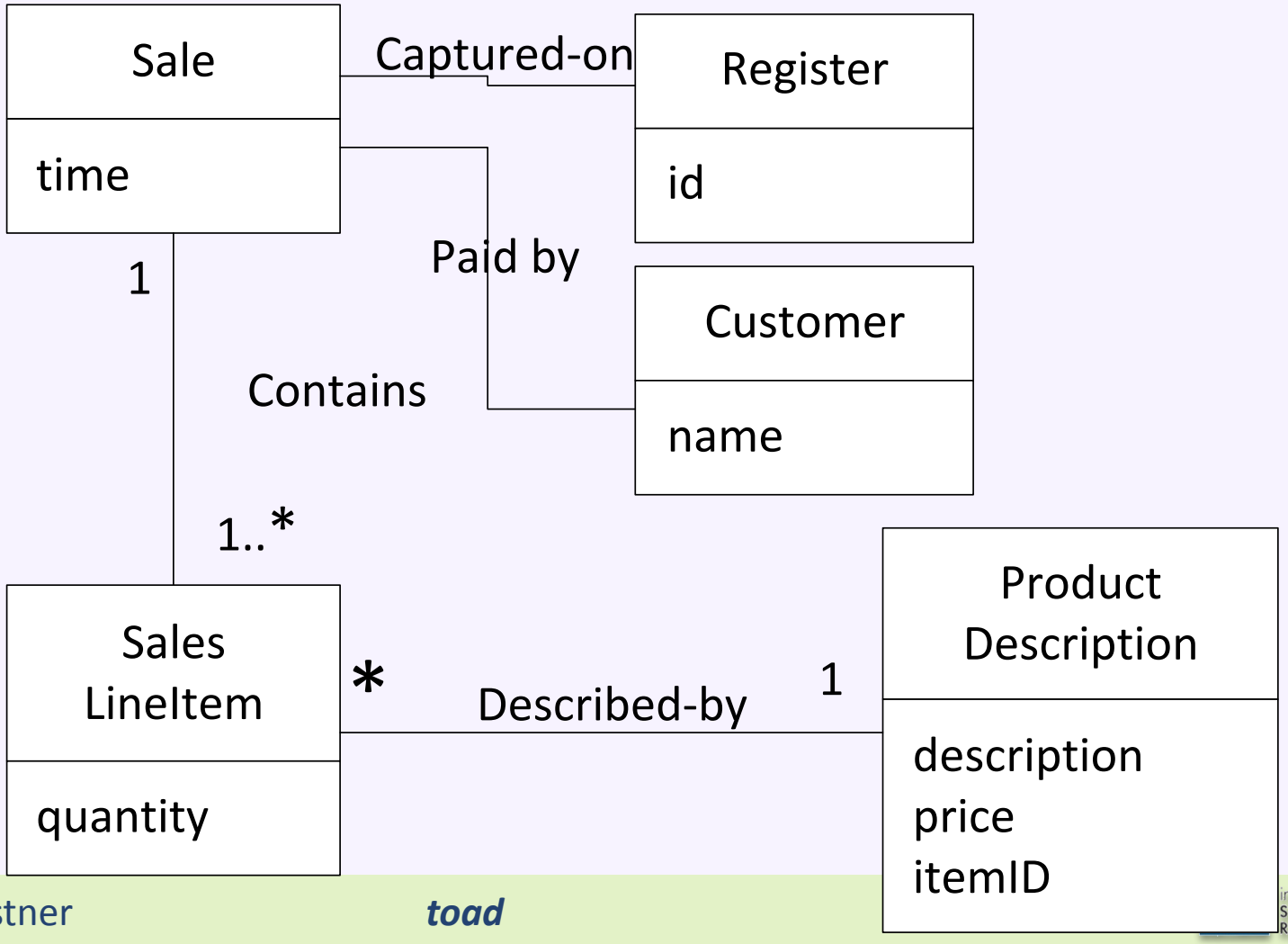
- Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
  - a sale does not tell you its total; it is an inanimate thing
- In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
- They do things related to the information they know.

Information Expert: Discussion of Design Goals/Strategies

- Explicit and small interfaces, information hiding
  - Does not expose how sale is computed
- Low coupling
  - Client does not need to know about LineItem and ProductDescription
- Low representational gap
  - Assign responsibilities similar to responsibilities of real-world abstractions
- May conflict with cohesion
  - Example: Who is responsible for saving a sale in the database?
  - Adding this responsibility to Sale would distribute database logic over many classes → low cohesion
- -> Design for Reuse, Understanding, Change, …

Creator (GRASP Pattern 2)

- Who is responsible for **creating** SalesLineItem objects?



Creator Pattern (GRASP Pattern 2)

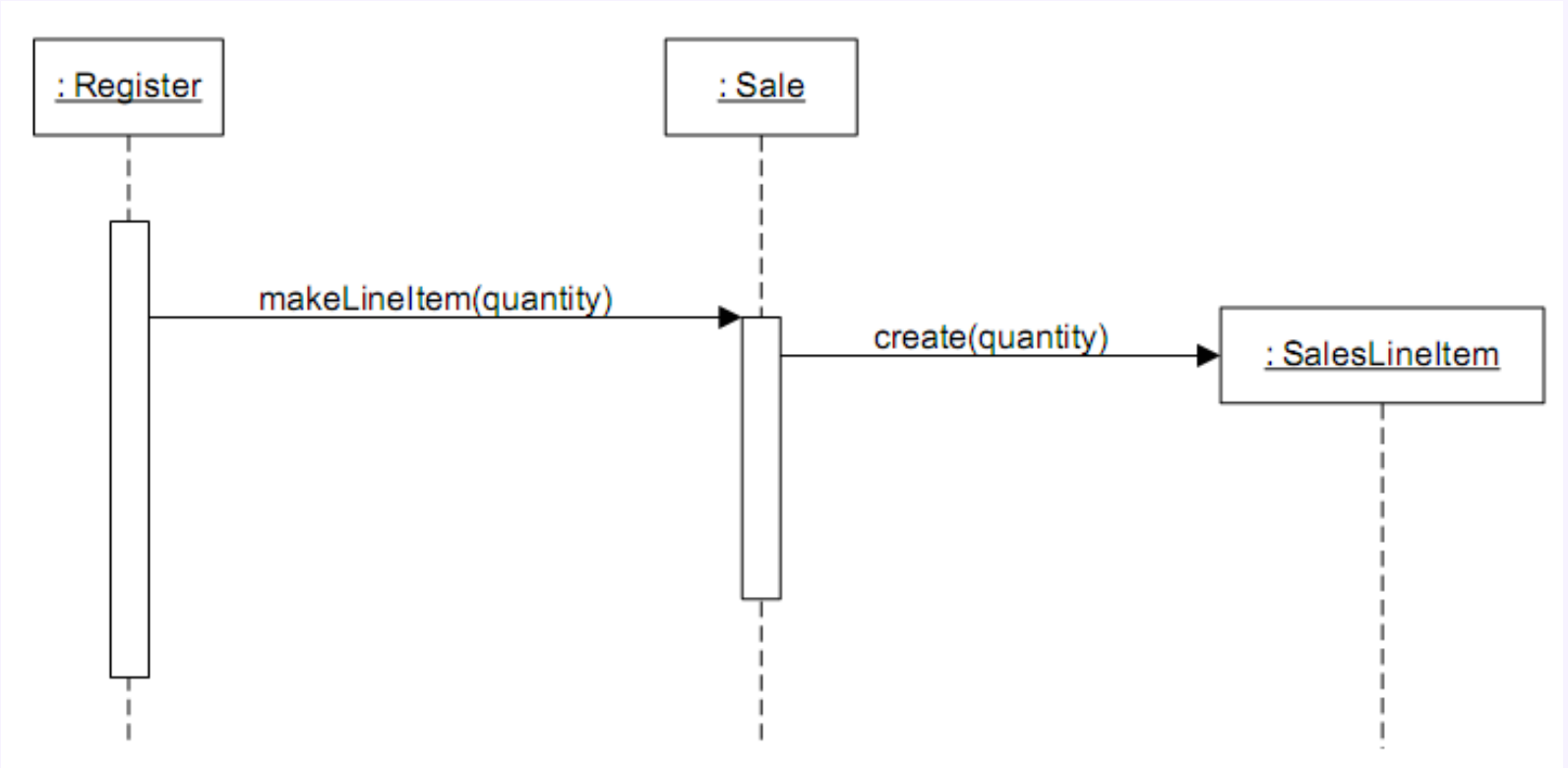
- Problem: Assigning responsibilities for creating objects
  - Who creates Nodes in a Graph?
  - Who creates instances of SalesItem?
  - Who creates Rabbit-Actors in a Game?
  - Who creates children if rabbits breed?
  - Who creates Tiles in a Monopoly game?
  - AI? Player? Main class? Board? Meeple (Dog)?

Creator Pattern

- Problem: Who creates an A?
- Solution: **Assign class responsibility of creating instance of class A to B if**
  - B aggregates A objects
  - B contains A objects
  - B records instances of A objects
  - B closely uses A objects
  - B has the initializing data for creating A objects
- the more the better; where there is a choice, prefer
  - B aggregates or contains A objects
- Key idea: Creator needs to keep reference anyway and will frequently use the created object

Creator : Example

- Who is responsible for creating SalesLineItem objects?
  - Creator pattern suggests Sale.
- Interaction diagram:

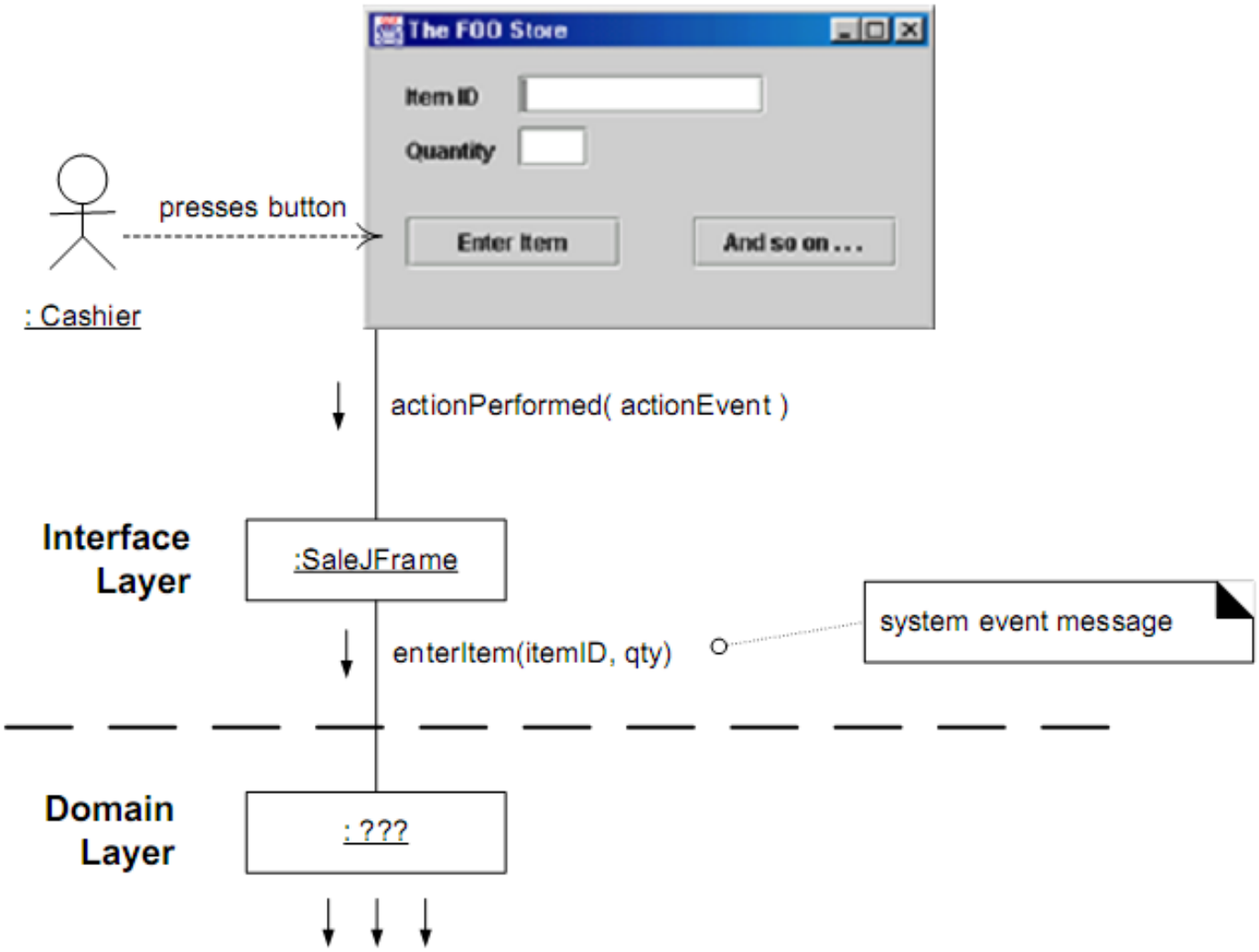


Creator: Discussion of Design Goals/Strategy

- Promotes **low coupling**, design for reuse
  - class responsible for creating objects it needs to reference
  - creating the objects themselves avoids depending on another class to create the object
- **Information hiding**, design for change
  - Object creation is hidden, can be replaced locally
- But creation may require significant complexity
  - using recycled instances for performance reasons
  - conditionally creating an instance from one of a family of similar classes based upon some external property value
  - Sometimes desired to outsource object wiring ("dependency injection")
- -> Several more complex creation strategies; many design patterns

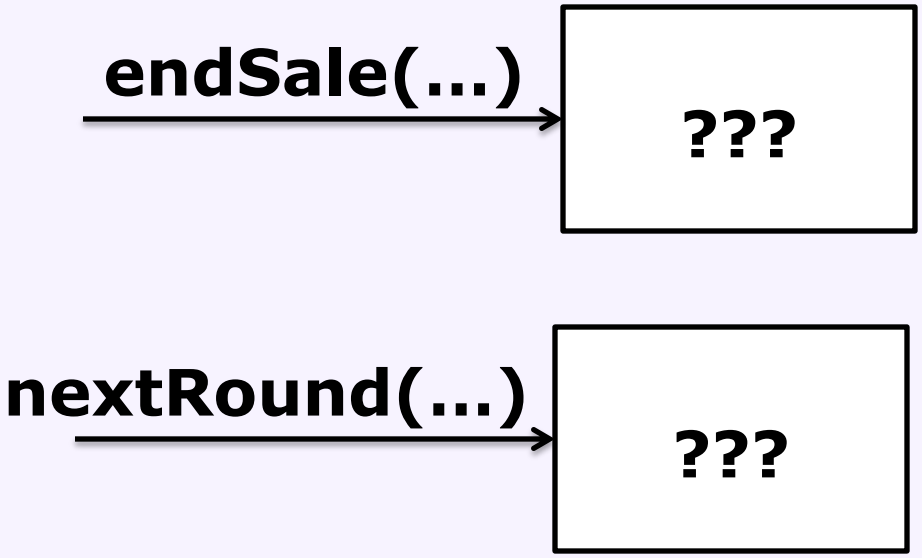
Controller (GRASP Pattern 3)

- What first object receives and coordinates a system operation (events)?



Controller (GRASP Pattern 3)

- What first object receives and coordinates a system operation (events)?
  - a user clicking on a button
  - a network request arriving
  - a database connection dropped



Controller (GRASP Pattern 3)

- Problem: What first object receives and coordinates a system operation (events)?
- Solution: Assign the responsibility to an object representing
  - the overall system, device, or subsystem (facade controller) or
  - a use case scenario within which the system event occurs (use case controller)

Controller: Example

- By the Controller pattern, here are some choices:
- *Register, POSSystem*: represents the overall "system," device, or subsystem
- *ProcessSaleSession, ProcessSaleHandler*: represents a receiver or handler of all system events of a use case scenario

Controller: Discussion

- Controller delegates to other objects; coordinates or controls the activity; does not much work itself
- Facade controllers suitable when not "too many" system events
  - -> one overall controller for the system
- Use case controller suitable when façade controller "bloated" with excessive responsibilities (low cohesion, high coupling)
  - -> several smaller controllers for specific tasks
- Closely related to Façade design pattern

Controller: Discussion of Design Goals/Strategies

- Design for Reuse
  - Reuse entire subsystem through small **explicit interface**
  - **Information hiding** for entire subsystem, e.g. hide that operations must be performed in specific sequence
  - Separation of application logic from GUI
- Design for Change
  - Application logic changeable without changing GUI
  - GUI changeable without changing application logic
- Design for Understandability
  - Dedicated place to understand interaction with environment events
- But, bloated controllers increase **coupling** and decrease **cohesion**; split if applicable

Other GRASP Patterns

- Low Coupling
  - Decide between two designs for the one with lower coupling
- High Cohesion
  - Decide between two designs for the one with higher cohesion
- Polymorphism
  - Support alternatives with dynamic dispatch (multiple implementations of an interface) instead of case analysis
  - See also Strategy Design Pattern
- Pure Fabrication
  - If domain model provides no reasonable concept to assign responsibility without violating cohesion/coupling -> create a new abstraction (e.g., PersistantStorage)

Summary

- Assigning Responsibilities to Classes
- GRASP Patterns for first design considerations
  - Information Expert
  - Creator
  - Controller
- Reason with Design Goals
- Patterns facilitate communication

Literature

- Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  - Chapter 16+17+22 introduce GRASP