

Responsibility Driven Object-Oriented Design

Object Oriented Analysis and Design

Aron Trauring

T++ Technical Skills Training Program

CUNY Institute for Software Design & Development (CISDD)

New York Software Industry Association (NYSIA)

December 13th, 2004

OOD Modeling — How?

- Code
- Draw then Code
- Draw

OOD Modeling — Why?

- To communicate
- To understand

Four Options

- Straight to code (XP: test-first development)
- UML modeling then code
- Other modeling techniques — e.g. CRC cards
- Draw only

Dynamic Models

- Design of logic/behavior
- Method bodies
- Helps identify objects and methods
- Interaction Diagrams

Static models

- Definition of packages
- Class names
- Attributes
- Method signatures
- Class Diagrams

Modeling activities

- Focus on design principles not diagrams
- Use output of OOA as input
- Dynamic models are more important
- Responsibility-driven design — assigning responsibility to collaborating objects
- Model to understand and communicate *not* document

Responsibility Driven Design

- Responsibility (System operations)
- Roles (Classes)
- Collaboration (Messages)
- OOD: Community of collaborating responsible objects
- OOD Modeling: thinking about assigning responsibilities to objects

Doing Responsibilities

- Doing something — Make Pizza
- Initiate action in other objects — Order Pizza
- Control and Co-ordinate other objects — PizzaShop

Knowing Responsibilities

- Knowing about my data — Oven knows how to Bake
- Knowing about related objects — Cashier takes money from Customer
- Knowing about things that can be derived or calculated — Cashier can calculate Order Total

Responsibilities Come from the Domain Model

- Low representational gap
- Not necessarily one to one
- One responsibility in Domain may be handled by many software classes
- One responsibility in Domain may be handled by one method
- Responsibilities are an abstraction
- Methods *fulfill* responsibilities

Collaboration

- Methods may collaborate with other methods or objects
- Server collaborates with Customer and RoboChef

Modeling — Tools for How-To

- *Principles* — basic guidelines for assigning responsibilities
- *Patterns* — named, well-known problem/solution which is codified
- Useful rules of thumb — *not* fixed laws
- Tried and true — not innovative

Patterns

- Name
- Detailed description of problem
- Detailed solution that can be applied
- Advice on how to apply in different contexts
- List of trade-offs, implementations, variations

GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Most basic principles of assigning responsibility
- Expressed as patterns

Nine GRASP Patterns

1. Creator
2. Information Expert
3. Low-coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication

8. Indirection

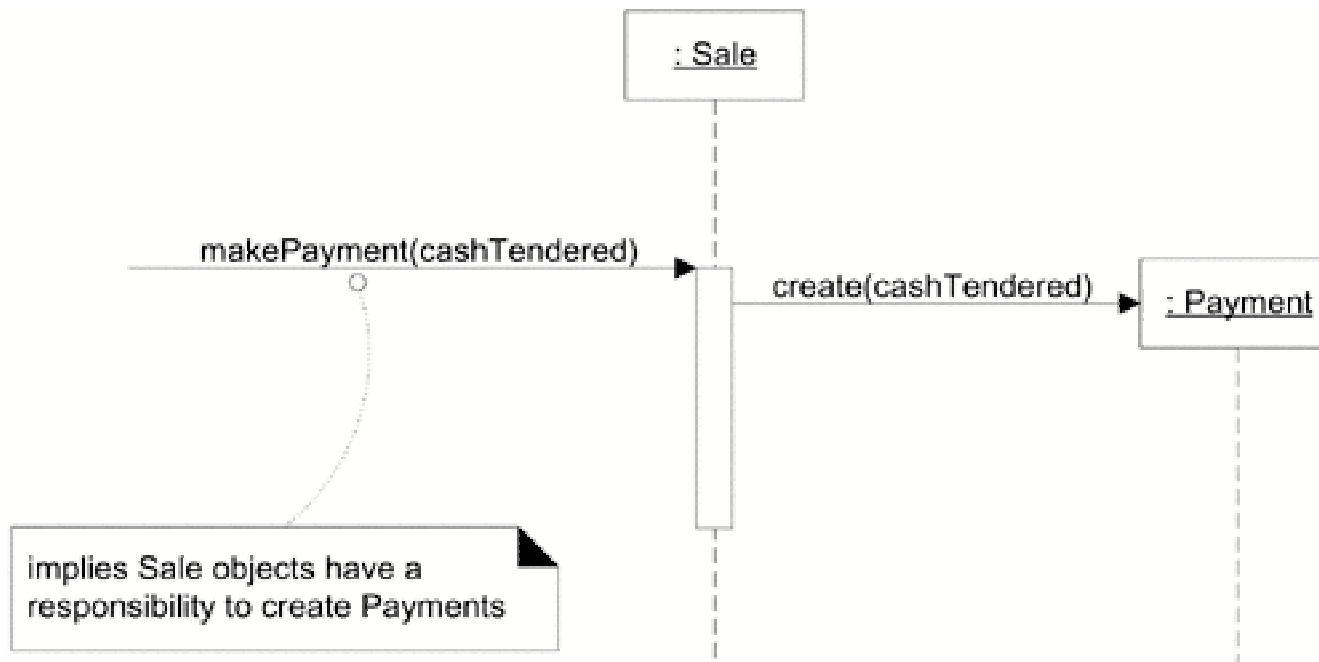
9. Protected Variation

Other Patterns

- GOF — Gang of Four — *Design Patterns* by Gamma, Helm, Johnson and Vlissides
- Mostly C++
- Lot's of new books

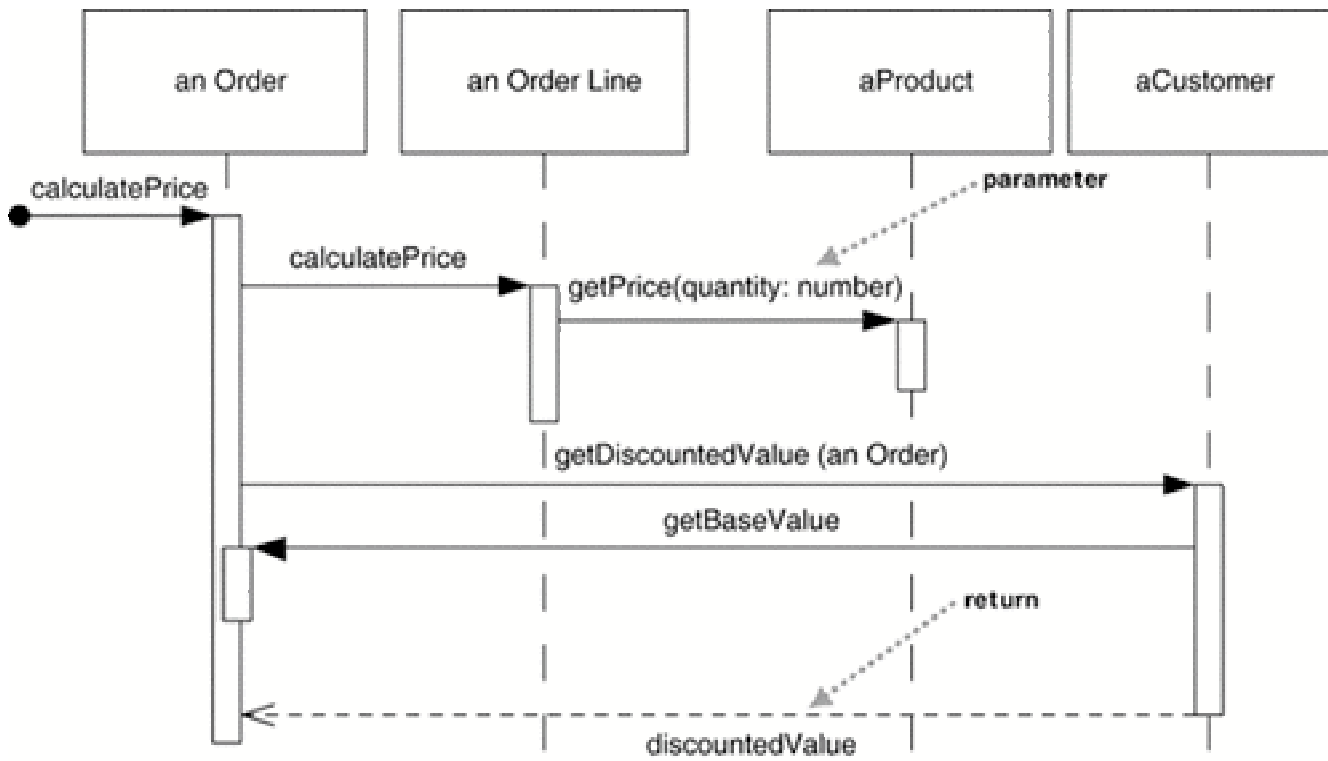
GRASP, Patterns and UML

- Consider responsibilities while drawing interaction diagrams



- Sale objects have responsibility to make payments
- Collaborates with Payment objects (creates them)

Sequence Diagram Syntax



Participants (Objects)

- Informal: an Order
- Formal: instance name : class
- 01 : Order or : Order

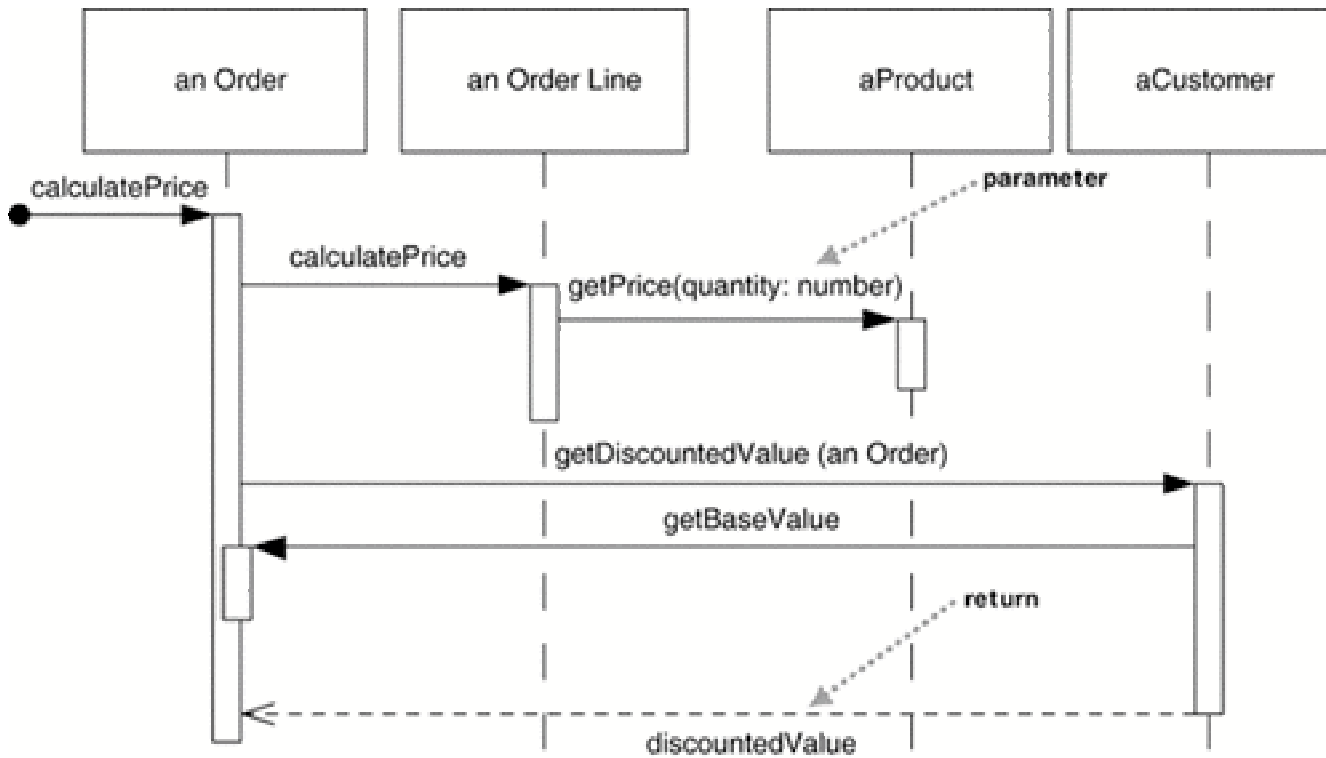
Messages (Methods)

- Found Message — “from the outside” — sender not specified
- Returns — broken arrows with stick arrowheads at end of activations
- List returns only if add information
- Message usually associated with method of receiving object — `Product.getPricingDetails`
- Messages can have parameters
- Filled arrowheads are synchronous
- Stick arrowheads are asynchronous

Lifelines

- Clearly shows order of operations
- Doesn't show how many times repeated (logic)
- Interaction frames — notational clutter to show looping and conditionals (Larman p.231)
- Activation — when message is active

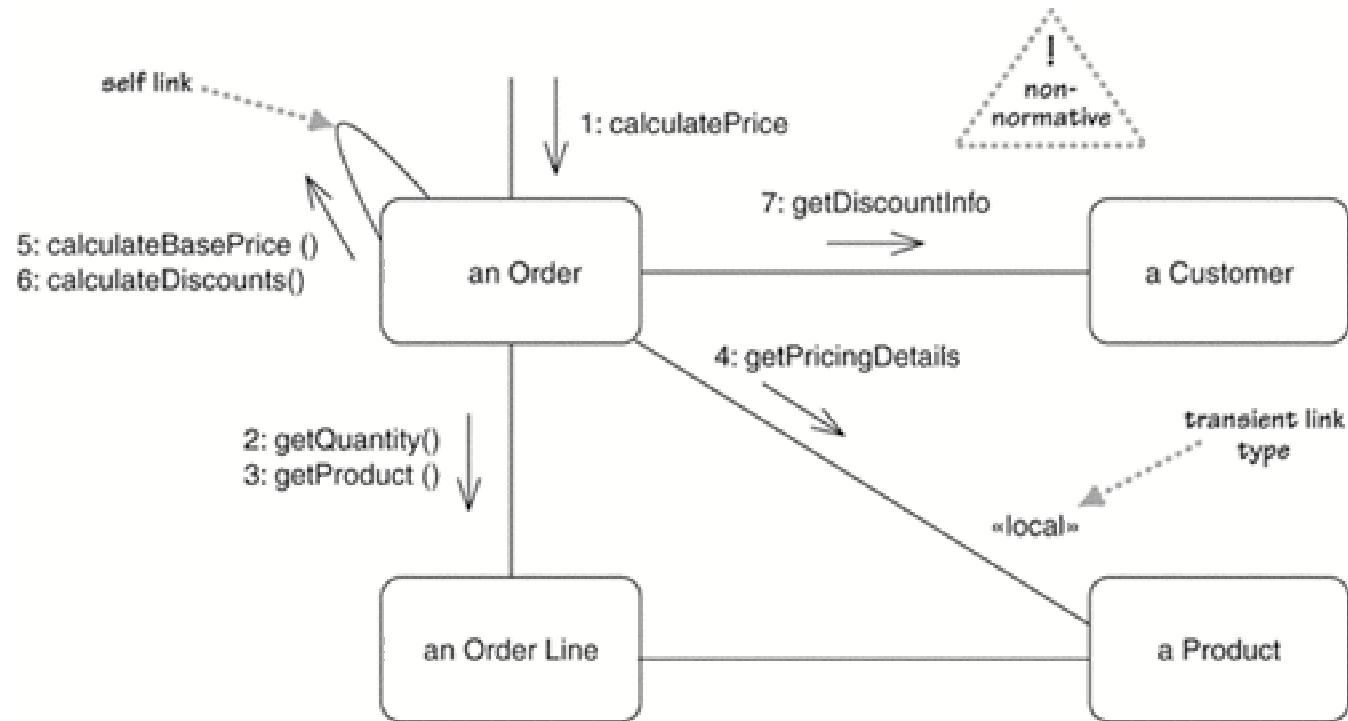
Distributed Control — Responsibility Driven Design



Creating Participants

- See diagram above with Sale and Payment
- Show deleting participants with big X

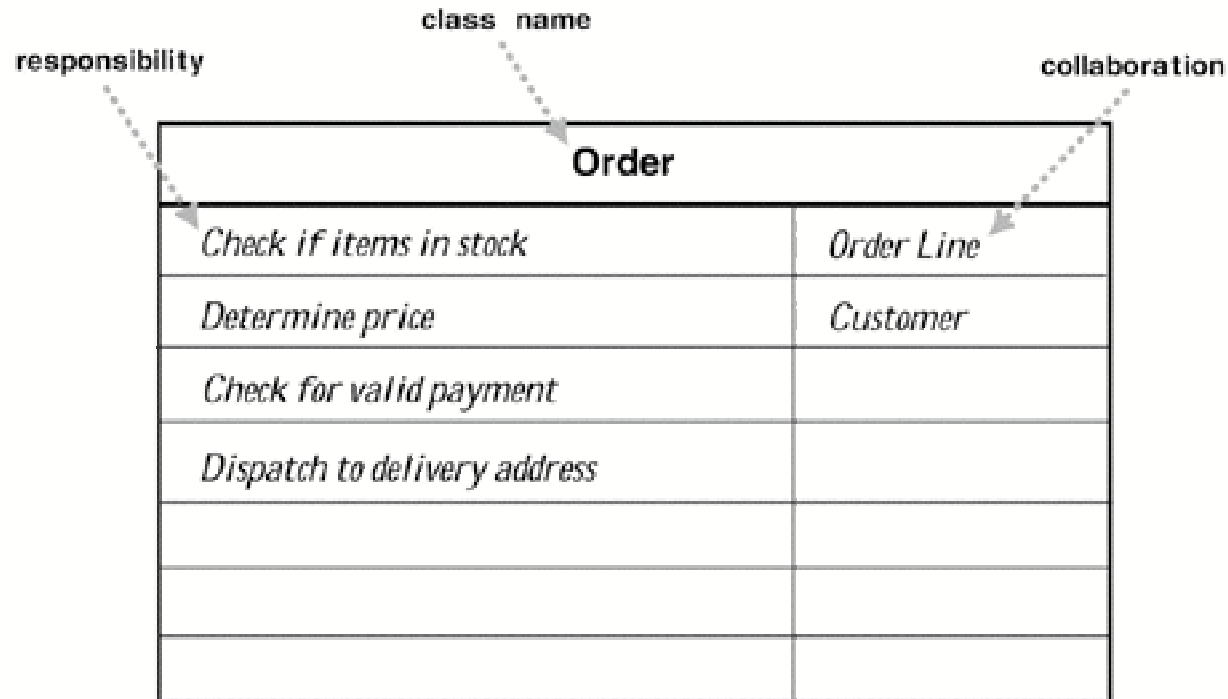
Communication Diagrams



- Normative version need to nest sequence number to avoid ambiguities
- 1.5 CalculateDiscounts calls 1.5.1 getDiscountInfo

- Flat numbering easier
- Easier to sketch than Sequence Diagrams

Class Responsibility Collaboration (CRC) Cards



CRC Cards — How?

- Gather around a table
- Take various scenarios and act them out with the cards
- Pick cards up in the air and move around to suggest how they send messages to each other
- Point is to identify and record responsibilities
- Take any class and summarize it with a *handful* of responsibilities
- Collaborators: the other classes that this class needs to work with

CRC Cards — Guidelines?

- Responsibilities should easily fit on one card
- If not, ask yourself whether the class should be split
- Alternative: responsibilities rolled up into higher-level statements

CRC Cards — Why?

- CRC cards encourage animated discussion among the developers
- Alternatives can take too long to draw and rub out with diagrams

Modeling Rules of Thumb (after Fowler)

- Use sequence diagrams to look at the behavior of several objects within a single use case
- Use CRC cards to explore multiple alternatives
- Use sequence diagrams to capture CRC sessions you want to refer to later
- Use code to model behavior within the scenario (Python as pseudo-code)
- For more complex behaviors across use cases use state or activity diagrams