Analysis and Design

- To develop an application, it is necessary
  - ◇ a description of the problem and requirements: what the problem is about and what a system must do
  - ◇ high-level and detailed descriptions of the logical solution and how it fulfills requirements and constraints
- **Analysis** emphasizes an investigation of the problem rather than how a solution is defined
- **Design** emphasizes a logical solution: how the system fulfills the requirements
- Ultimately, designs can be implemented in software and hardware

1

Object-Oriented Analysis and Design

- Essence of OAD: consider a problem domain and logical solution from the perspective of objects (things, concepts, entities)
- **Object Analysis**: find and describe the objects (concepts) in the problem domain
  - ◇ Some concepts in a library information system: Book, Library, Client
- **Object Design**: define logical software objects that will be implemented in an object programming language (OPL)
- These software objects have attribute and methods
  - ◇ Book may have a title attribute and a print method
- **Construction or Object Programming**: design components are implemented
  - ◇ Book class in C++, Java, Smalltalk, or Visual Basic

2

Requirements Analysis

- Understanding the requirements includes
  - ◇ domain processes, and
  - ◇ external environment: external actors participating in the processes
- Domain processes can be discovered and expressed in use cases
- **Use cases**: textual narrative descriptions of the processes in an enterprise or system
- There is nothing OO in use cases
- Simply describe processes, can be equally effective in a non-object technology project
- However, it is an important and widely-practiced early step in OO analysis and design methods
- Use cases are part of the UML

3

Use Cases: Examples

- Order management

Use case :	Place an order
Actors :	Customer
Description :	This use case begins when a customer phones a sales representative to request a purchase of one or several products. The sales representative records the customer and product information in a new order.

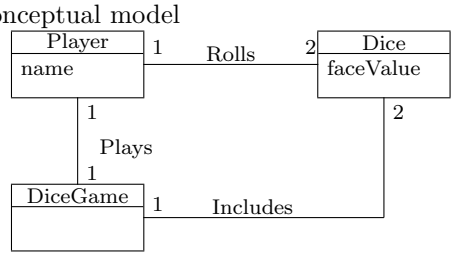
- Dice game

Use case :	Place a game
Actors:	Player
Description :	This use case begins when the player picks up and rolls the dice. If the dice total is seven he win, otherwise he lose

4

Domain Analysis

- Create a specification of the problem domain and the requirements from the perspective of
  - ◇ classification by objects, and
  - ◇ understanding the terms used in the problem domain
- Involves an identification of the concepts, attributes, and associations considered important in the domain
- Expressed with a conceptual model



- Conceptual model does not describe software components, it represents concepts in real-world problem domain

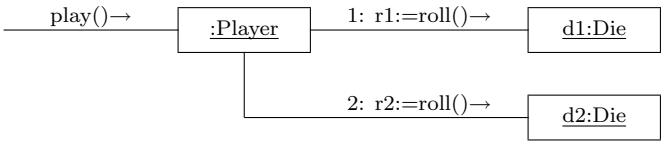
5

Design

- Define logical software specifications that fulfill the functional requirements based on decomposition by classes of objects
- Solution-oriented activity
- Must emphasize responsibility assignment: allocate tasks and responsibilities to the various objects in the application
- Software objects usually collaborate or interact in order to fulfill their responsibilities
- Often expressed with
  - ◇ design class diagrams: show definition of classes
  - ◇ collaboration diagrams: show flow of messages between software objects

6

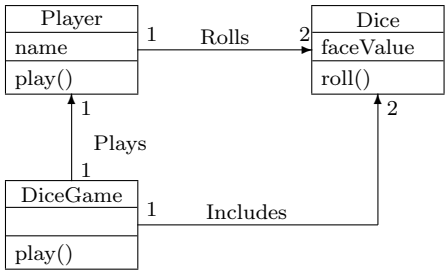
Collaboration Diagrams: Example



7

Design Class Diagrams

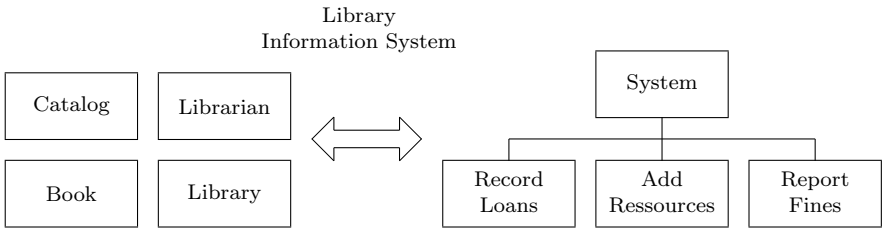
- Questions to be answered to design a class
  - ◇ How do objects connect to other objects ?
  - ◇ What are the methods of a class ?
- Collaborations diagrams used to answer these questions
- Design class diagrams express these design details
- Illustrate class definitions that are to be implemented in software



- Does not illustrate real-world concepts, describes software components
- A line with an arrow at the end may suggest an attribute

8

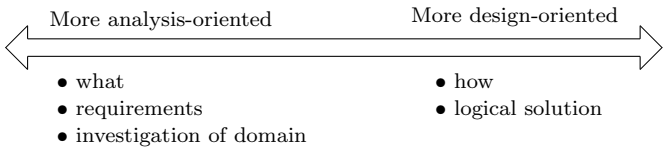
Object- vs Function-Oriented Analysis and Design



- Software projects are complex
- Decomposition (divide-and-conquer): primary strategy to deal with complexity
- Structured analysis and design
  - ◇ popular approach in the 70's
  - ◇ decomposition by function or process
  - ◇ result: hierarchical breakdown of processes composed of subprocesses
- OO analysis and design: decompose a problem space by objects rather than by functions

9

Analysis and Design: Terminology Wars



- Division between analysis and design is fuzzy
- They exists in a continuum
- Different practionners classify an activity at varying points of the continuum
- Distinction between investigation (analysis) and solution (desing) is important
  - ◇ Emphasizes what the problem is before diving in to how to create a solution
  - ◇ Understanding the problem during analysis, deferring to the design issues related to the solution, performance, ...

10

Unified Modeling Language (UML)

- Language for specifying, visualizing, and constructing the artifacts of software systems
- Notational system (with limited associated semantics) aimed at modeling systems using object concepts
- Join of methods by Grady Booch, James Rumbaugh (OMT), and Ivar Jacobson (OOSE)
- Accepted by Object Management Group (OMG) as a standard modeling language and notation
- De facto approval in industry
- Language for modeling, does not prescribe a specific development process
- Process standardization was outside the scope of the UML definition
- But the UML has an associated "Unified Software Development Process"

11

Software Development Process

- Method to organize activities related to creation, delivery, and maintenance of software systems
- More important than following an official process or method is that
  - ◇ developer acquire skills in how to create a good design
  - ◇ organization foster this kind of skill development
- This comes from mastering a set of principles and heuristics for
  - ◇ identifying and abstracting suitable objects
  - ◇ assigning responsibilites to them
- A process includes the activities from requirements through to delivery
- A complete process addresses broader issues related to the industrialization of software development : long-term life cycle, documentation, support and training, parallel work, coordination between parties
- Essential steps not covered in this course: conception, planning, parallel team interaction, project management, documentation, testing

12

Development Processes

- Reasons for not standardizing a process in the UML
  - ◇ Increase widespread acceptance of a standard modeling notation without commiting to a standard process
  - ◇ Significant variability in what constitutes an appropriate process, depending on staff skills, research-development ratio, nature of the problem, tools, ...
- But, general principles and typical steps that guide a successful process can be explained
- Macro-level steps for delivering an application
  - ◇ **Plan and elaborate**: planning, define requirements, build prototypes, ...
  - ◇ **Build**: construction of the system
  - ◇ **Deploy**: implementation of the system into use

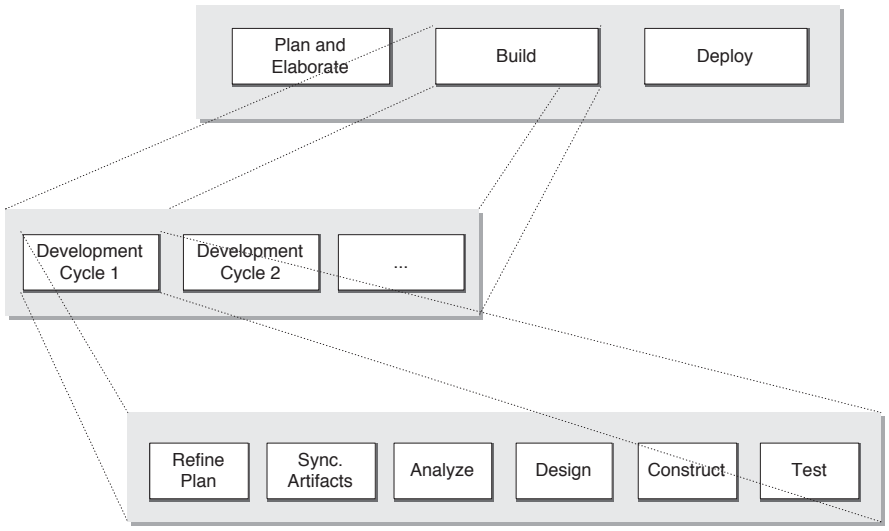
13

Iterative Development

- Iterative life cycle: successive enlargement and refinement of a system through multiple development cycles of
  - ◇ analysis, design, implementation, and testing
- System grows incrementally by adding new functions within each development cycle
- Build phase composed of a series of development cycles
- Each cycle tackles a relatively small set of requirements
- $\neq$ classic waterfall lifecycle: each activity (analysis, design, ...) done once for the entire set of system requirements
- Advantages of iterative development
  - ◇ complexity is never overwhelming
  - ◇ early feedback generated: implementation occurs rapidly for a small subset of the system

14

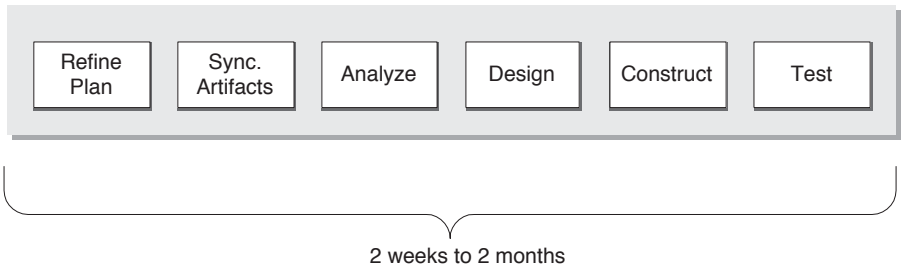
Iterative Development Cycles



15

Time-Boxing a Development Cycle

- Useful strategy: bound each development cycle within a time-box (a rigidly-fixed time), e.g., 4 weeks
- All work must be accomplished in that time frame
- A range between two weeks and two months is suitable
  - ◇ any less: difficult to complete tasks
  - ◇ any more: complexity becomes overwhelming, feedback is delayed



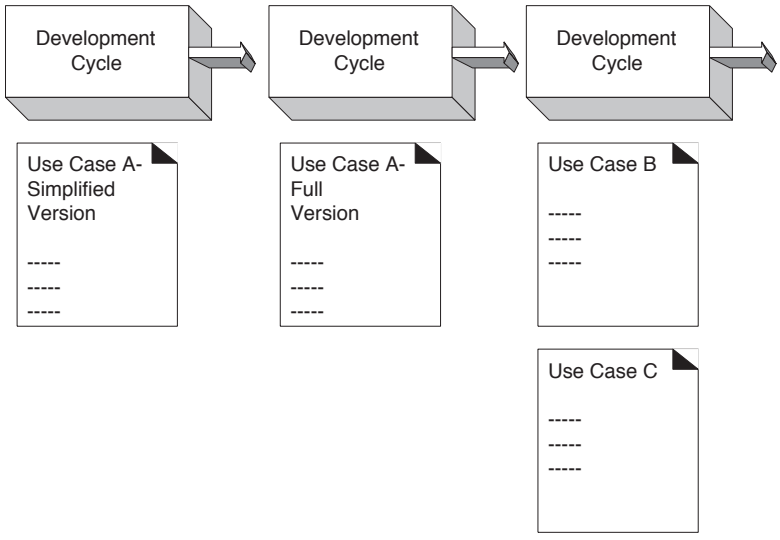
16

Use Cases and Iterative Development Cycles

- Use case: narrative description of a domain process
  - ◇ e.g., borrow books from a library
- Iterative development cycles organized by use case requirements
- One cycle implements one or more use cases
  - ◇ Simplified versions of use case when complete use case is too complex to tackle in one cycle
- Further, some development cycles (esp. early ones) must focus on supporting services: persistence, security, ...
- Use cases should be ranked: high-ranking use cases should be tackled in early development cycles
- Usual strategy: pick first
  - ◇ use cases influencing the core architecture by fleshing out the domain and high-level services layers, or
  - ◇ critical high-risk use cases

17

Development Cycles Driven by Uses Cases



18

Plan and Elaborate Phase: Sample activities

(1) Define draft plan
(2) Create preliminary investigation report
(3) Define requirements
(4) Record terms in glossary [a]
(5) Implement prototype [b,d]
(6) Define use cases (high-level and essential)
(7) Define draft conceptual model [c]
(8) Define draft system architecture [a,c,d]
(9) Refine Plan

Notes: a = ongoing, b = optional, c = may defer, d = varied order

19

Plan and Elaborate Phase: Artifacts

- **Plan**: schedule, ressources, budget, ...
- **Preliminary Investigation Report**: motivation, alternatives, business needs
- **Requirements Specification**: declarative statement of requirements
- **Glossary**: dictionary of terms (concept names, and so on) and any associated information, such as constraints and rules
- **Prototype**: system created to aid understanding of the problem, high-risks problems and requirements
- **Use cases**: prose descriptions of domain processes
- **Use case diagrams**: illustrates uses cases and their relationships
- **Draft conceptual model**: preliminary model as an aid in understanding the vocabulary of the domain, esp. as it relates to the use cases and requirements specification

20

Order of Artifact Creation

- Artifact creation is not done in a strictly linear order
- Some artifacts may be made in parallel
  - ◇ especially true of the conceptual model, glossary, use cases, use case diagram
- While the use cases are explored, other artifacs are developed to reflect the information arising from the use cases
- In the following, artifacts are introduced in a linear order to keep presentation straightforward
- In practice there is much more interplay

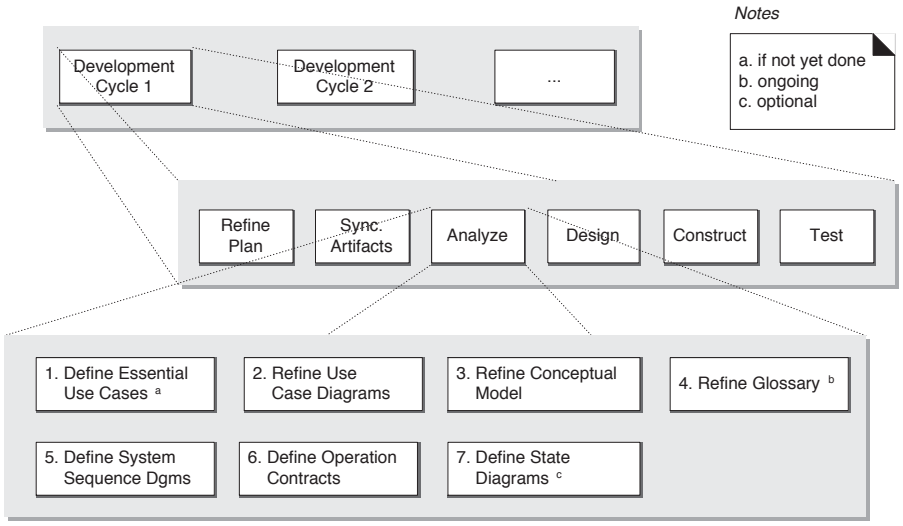
21

Build Phase: Development Cycles

- Final objective: working software system correctly meeting the requirements
- Within a single development cycle the major steps are to analyze and design
- As for requirements, there is no strict linear order in the artifacts produced
  - ◇ create conceptual model and glossary in parallel
  - ◇ create interaction diagrams and design class diagrams in parallel

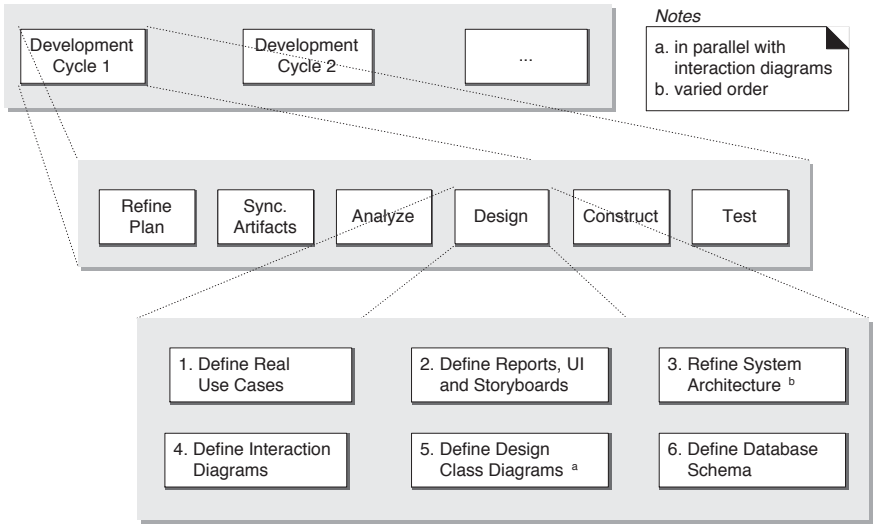
22

Sample Analysis Phase Activities



23

Sample Design Phase Activities



24

When to Create the Conceptual Model

- Conceptual model (CM): representation of concepts or objects in problem domain
- A draft CM should be done in the Plan and Elaborate Phase
- Goal: basic understanding of vocabulary and concepts used in the requirements
- Risk of fine-grained investigation: complexity overload
  - ◇ a thorough CM is overwhelmingly complex in large domains
- Recommended strategy: create a rough CM for finding obvious concepts expressed in the requirements
  - ◇ Later, within each development cycle, CM is refined and extended for the requirements under consideration within that cycle
- Another strategy: defer creation of CM until each development cycle
  - ◇ Advantage: deferring complexity
  - ◇ Disadvantage: less up-front information useful for general comprehension, for creating the glossary, for scoping and estimating

25

When to Create Expanded Use Cases

- **High-level** use cases: very brief, 2 or 3 sentence descriptions
- **Expanded** use cases: long narratives, may contain hundreds of sentences
- During the Plan and Elaborate phase
  - ◇ create all high-level use cases
  - ◇ rewrite most critical and important use cases in an expanded format, deferring the rest until the development cycle in which they are tackled
- Trade-off: benefit of early acquisition of information vs tackling too much complexity
- Advantage of early writing of all detailed use cases: more information
  - ◇ help with comprehension, risk management, scope, estimating
- Disadvantages
  - ◇ early complexity overload
  - ◇ may not be very reliable because of incomplete or misinformation, requirements may change

26

Defining Models and Artifacts

- Real-world or software systems are usually overwhelmingly complex
- System is decomposed into understandable chunks to comprehend and manage complexity
- These chunks may be represented as **models** describing and abstracting essential aspects of the system
- In a software system, models are used to organize and communicate the details of the real-world problem it is related and the system to be built
- Models should contain cohesive, strongly related elements
- Models are composed of **artifacts**: diagrams and documents describing things
- Models are visualized with **views**: visual projections of the model
- Models can be characterized emphasizing **static** or **dynamic** information about a system
  - ◇ Static model describes structural properties
  - ◇ Dynamic model describes behavioural properties of a system

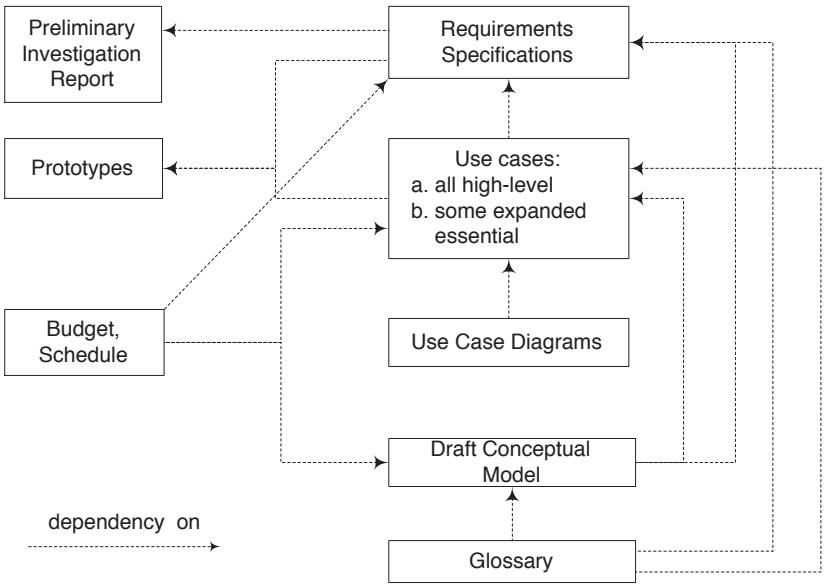
27

Relationships between Artifacts

- Independent of how artifacts are organized into models, there are influencial dependencies between artifacts
- Example: use case diagram (illustrates all uses cases) is dependent on the use case definitions themselves
- Dependency and influence between artifacts used for consistency checks and traceability
- Dependent artifacts are effectively used as input to creating later artifacts

28

Plan and Elaborate Phase Artifact Influence



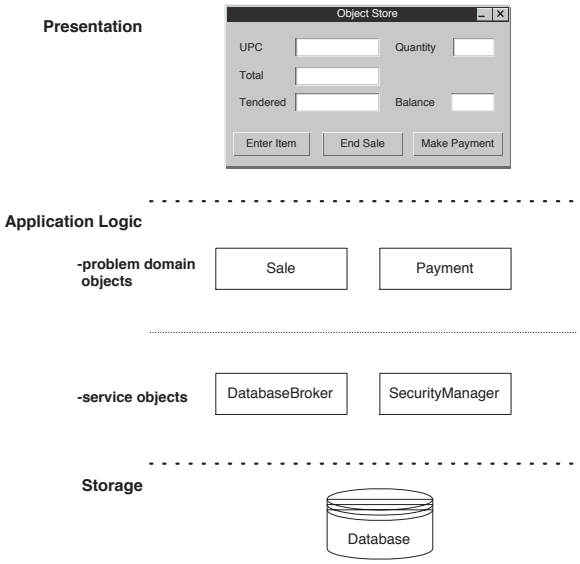
29

Case Study: Point-of-Sale Terminal

- Point-of-sale terminal: computerized system used to record sales and handle payments
- Typically used in a retail store
- Includes hardware components (computer, bar code scanner) and software running the sytem
- Typical information system with graphical user interface (GUI) and database access usually designed in terms of several layers
  - ◇ **Presentation**: GUI, windows
  - ◇ **Application Logic, Problem Domain Objects**: objects representing domain concepts that fulfill application requirements
  - ◇ **Application Logic, Service Objects**: non-problem domain objects that provide supporting services, e.g., interfacing with a database
  - ◇ **Storage**: persitent storage mechanism, e.g., object or relational database

30

Layers in Typical Object Information System



31

Understanding Requirements

- Requirements: description of needs or desires for a product
- Correct and thorough requirements specification is essential to a successful project
- Goal of the requirements phase: identify and document what is really needed, in a form that clearly communicates to the client and to the development team
- Challenge: define the requirements unambiguously, identify the risks
- Some artifacts in the requirements phase (none of them are UML-specific)
  - ◇ overview statement
  - ◇ customers
  - ◇ goals
  - ◇ system functions
  - ◇ system attributes
- Definition of requirements typically involves gathering and digesting varied paper and electronic documents, interview results, group requirements definition meetings, ...

32

Point-of-Sale System: Requirements

- **Overview Statement**: The purpose is to create a point-of-sale terminal sytem to be used in retail sales
- **Customers**: ObjectStore, Inc. a multinational object retailer
- **Goals**: Increased checkout automation, to support faster, better, and cheaper services and business process. More specifically these include
  - ◇ Quick checkout for the customer
  - ◇ Fast and accurate sales analysis
  - ◇ Automatic inventory control

33

System Functions

- What a system is supposed to do
- Should be identified and listed in logical cohesive groupings
- Verification: a system function X should make sense in the sentence

  The system should do X
- Example: The system should do credit payment authorization
- System attributes: nonfunctional system qualities often confused with functions
  - ◇ E.g., ease-of-use does not fit in the verification sentence
- System attributes should not be part of the functional specification document but in another document devoted to that purpose

34

Function Categories

- Functions should be categorized in order to prioritize them and identify those that might otherwise be taken for granted
- Categories include evident, hidden, and frill
- Evident: should perform and user should be congizant that it is performed
- Hidden:
  - ◇ should perform but not be visible to users
  - ◇ includes many underlying technical services, e.g., save information in a persistent storage mechanism
  - ◇ often missed during the requirements gathering process
- Frill: optional, adding it does not significantly affect cost or other functions

35

Point-of-Sale System: Basic Functions

Ref #	Function	Category
R1.1	Record the current sale, the items purchased	evident
R1.2	Calculate current sale total, including tax and cupon calculations	evident
R1.3	Capture purchase item information from a bar code (either with a scanner or manually)	evident
R1.4	Reduce inventory quantities when a sale is comitted	hidden
R1.5	Log completed sales	hidden
R1.6	Cashier must log in with an ID and password in order to use the system	evident
R1.7	Provide a persistent storage mechanism	hidden
R1.8	Provide inter-process and inter-system communication mechanisms	hidden
R1.9	Display description and price of item recorded	evident

36

Point-of-Sale System: Payment Functions

Ref #	Function	Category
R2.1	Handle cash payments, capturing amoung tendered and calculating balance due	evident
R2.2	Handle credit payments, capturing credit information (from a card reader or manually) and authorizing payment with the store's (external) credit authorization service via a modem connection	evident
R2.3	Handle check payments, capturing drivers license manually, and authorizing payment with the store's (external) credit authorization service via a modem connection	evident
R2.4	Log credit payments to the accounts receivable system, since the credit authorization service owes the store the payment amount	hidden

37

System Attributes

- Characteristics or dimensions of the system, they are not functions
- Examples: ease of use, fault tolerance, response time, interface metaphor, retail cost, platforms
- May cut across all functions (e.g., OS platform) or be specific to a particular function or group of functions
- Values of attribute tend to be discrete, fuzzy symbolic
  - ◇ response time = psychologically appropriate
  - ◇ interface metaphor = graphical, colorful, forms-based
- System attributes may also have attribute boundary constraints,
  - ◇ e.g., response time = 5 seconds maximum

38

Point-of-Sale System: System Attributes

Attribute	Details and Boundary Constraints
response time	When recording a sold item, the description and price will appear within 5 seconds
interface metaphor	Forms-metaphor windows and dialog boxes Maximize for easy keyboard navigation rather than pointer navigation
fault tolerance	Must log authorized credit payments to accounts receivable within 24 hours, even if power or device failure
OS platform	Microsoft Windows 95 and NT

39

Use Cases: Describing Processes

- Technique to improve understanding of requirements
- Use case: narrative document describing the sequence of events of an actor (external agent) using a system to complete a process
- Dependent on having at least partial understanding of the requirements of the system
- UML icon for a use case

40

High-level vs Expanded Use Cases

- Use cases may be expressed with varying degrees of detail and commitment to design decisions
- High-level
  - ◇ describes a process very briefly, usually 2-3 sentences
  - ◇ very terse, and vague on design decisions
  - ◇ useful to quickly obtain some understanding of the overall major processes
- Expanded use cases
  - ◇ Show more detail than high-level use cases
  - ◇ Useful to obtain a deeper understanding of the processes and requirements
  - ◇ Often done in "conversational" style between the actors and the system

41

High-level Use Case: Examples

Use case:	Buy Items
Actors:	Customer (initiator), Cashier
Type:	Primary
Description:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects a payment. On completion the Customer leaves with the items.

Use case:	Start Up
Actors:	Manager
Type:	Primary
Description:	A Manager powers on a POST in order to prepare it for use by Cashiers. The Manager validates that the date and time are correct, after which the system is ready for Cashier use.

42

Expanded Use Case: Example

Use case:	Buy Items with Cash
Actors:	Customer (initiator), Cashier
Purpose:	Capture a sale and its cash payment
Overview:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collect a cash payment. On completion the Customer leaves with the items
Type:	Primary and essential
Cross references :	Functions R1.1, R1.2, R1.3, R1.7, R1.9, R2.1

43

Typical Course of Event

Actor Action	System Response
1. This use case begins when a Customer arrives at a POST checkout with items to purchase	
2. The Cashier records the identifier from each item. If there is more than one of the same item, the Cashier can enter the quantity as well.	3. Determines the item price and adds the item information to the running sales transaction. The description and price of the current item are presented.
4. On completion of item entry, the Cashier indicates to the POST that the item entry is complete.	5. Calculates and presents the sale total.
6. The Cashier tells the Customer the total.	
7. The Customer gives a cash payment, possibly greater than the sale total.	
8. The Cashier records the cash received amount.	9. Shows the balance due back to the Customer. Generates a receipt.
10. The Cashier deposits the cash received and extracts the balance owing. The Cashier gives the balance owing and the printed receipt to the Customer.	11. Logs the completed sale.
12. The Customer leaves with the items purchased.	

Alternative Courses

- Line 2: Invalid indentifier entered. Indicate error.
- Line 7: Customer didn't have enough cash. Cancel sales transaction.

44

Expanded Use Case: Format

Use case:	Name of the use case
Actors:	List of actors (external agents), indicating who initiates the use case.
Purpose:	Intention of the use case.
Overview:	Repetition of the high-level use case of some similar summary.
Type:	(1) Primary, secondary, or optional (2) Essential or real
Cross references :	Related use cases and system functions

45

Expanded Use Case: Format, cont.

- Typical course of event
  - ◇ heart of the expanded format
  - ◇ conversation between actors and the system
  - ◇ most common (typical) sequence of events: average story of activities and successful completion of process
- Alternative course of events
  - ◇ important alternatives or exceptions that may arise
  - ◇ if complex, may themselves be expanded into their own use cases

46

Actors

- **Actor**: entity external to the system who participates in a use case
- Typically stimulates the system with input events, or receives something from it
- Usually the roles human play: Customer, Cashier, ...
- May be any kind of system: computer systems, electrical or mechanical devices
- Actors capitalized in the use case prose for ease of identification
- In a use case
  - ◇ one **initiator actor** generates the starting stimulus
  - ◇ several other **participating actors**
- UML icon for an actor

Customer

47

Identifying Use Cases

- Common error: represent individual steps, operations, or transactions
  - ◇ e.g., Printing the Receipt is a step in the use case Buy Items
- A use case is a relatively large end-to-end process description, typically includes may steps or transactions
- Actor-based identification
  - ◇ Identify the actors related to a system or organization
  - ◇ For each actor, identify the processes they initiate or participate in
- Event-based identification
  - ◇ Identify the external events that a system must respond to
  - ◇ Relate the events to actors and use cases

48

Point-of-Sale System: Identifying Use Cases

Actor	Process Initiated
Cashier	Login Cash Out
Customer	Buy Items Refund Items
Manager	Start Up Shut Down
System Administrator	Add New Users

49

Uses Cases and Domain Processes

- **Process**: describes a sequence of events, actions, and transactions required to produce or complete something of value to an organization or actor
  - ◇ e.g., withdraw cash from ATM, order a product, register for courses, ...
- System functions identified during requirements specifications, should be all allocated to use cases
- With Cross References section, verification that all functions have been allocated
  - ◇ Provides traceability between artifacts
- Ultimately, all system functions and use cases should be traceable through to implementation and testing

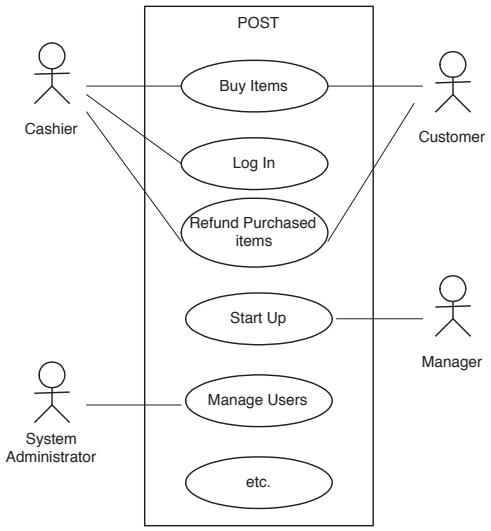
50

Use Case Diagrams

- Illustrates a set of use cases for a system, the actors and the relation between the actors and the use cases
- Specifies lines of communication between uses cases and actors
- Arrows can indicate flow of information or stimulus
- Purpose: represent a kind of context diagram to quickly understand
  - ◇ the external actors of a system
  - ◇ the key ways in which they use it

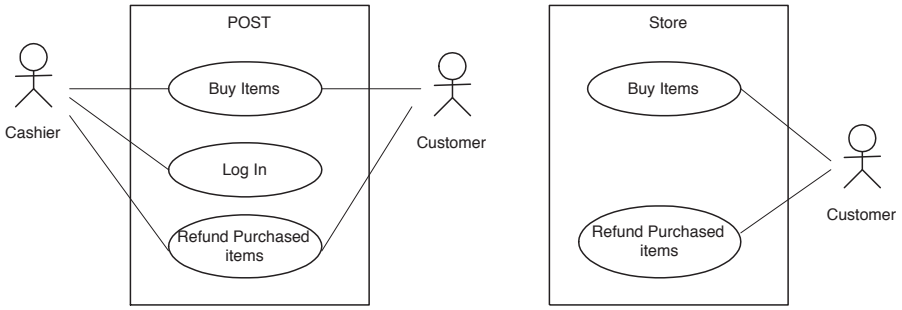
51

Point-of-Sale System: Partial Use Case Diagram



52

Systems and their Boundaries



- Typical system boundaries
  - ◇ hardware/software boundary of a device or computer system
  - ◇ department of an organization
  - ◇ entire organization
- Defining the system boundary allows to identify what is external vs internal, the responsibilities of the system

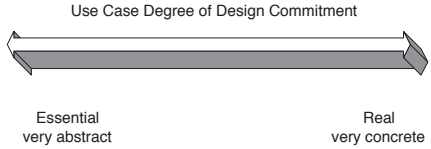
53

Influence of choosing the system boundary

- POST terminal harware and software as the system: both the customer and the cashier may be treated as actors
- Entire store or business as the system: only the custumer is an actor because the cashier is a ressource within the business system that carries out the tasks

Classification of Use Cases

- Importance
  - ◇ Primary: represent major common processes, e.g., Buy Items
  - ◇ Secondary: represent minor or rare processes, e.g., Stock New Product
  - ◇ Optional: represent processes that may not be tackled
- Essential vs Real



54

Essential Use Cases

- Expanded use cases expressed in an ideal form relatively free of technology or implementation details
- Design decisions deferred and abstracted, esp. those related to the UI
- High-level use cases are always essential in nature
- Important during early requirements elicitation for understanding the scope of the problem and the functions required
- Allow to see the essence of the process and its fundamental motivation without being overwhelmed with design details
- Tend to be correct for a long period of time
- Allow to understand and record the fundamental forces behind business processes

55

Real Use Cases

- Concretely describe the process in terms of its real current design, commited to specific input/output technologies
- When a user interface is involved, they show screen shots and discuss interaction with the widgets
- Ideally created during the design phase of a development cycle
- When in a project early design decisions regarding the UI are expected ⇒ created during the early elaboration phase
- Undesirable in the Plan and Elaborate phase
  - ◇ premature commitment to a design
  - ◇ overwhelming complexity involved
- Some organizations commit to a development contract on the basis of UI specifications

56

Essential vs Real Use Cases

- Essential use case

Actor Action	System Response
1. . . .	
2. The Cashier records the identifier from each item. If there is more than one of the same item, the Cashier can enter the quantity as well.	3. Determines the item price and adds the item information the the running sales transaction. The description and price of the current item are presented.
4. etc.	5. etc.

- Real use case

Actor Action	System Response
1. . . .	
2. For each item the Cashier types in the Universal Product Code (UPC) in the UPC input file of Window1. They then press the "Enter Item" button with the mouse or by pressing the Enter Key	3. Determines the item price and adds the item information the the running sales transaction. The description and price of the current item are presented in Textbox2 of Window1.
4. etc.	5. etc.

57

Decision Points and Branching

- Uses cases may contain decision points
- In Buy Items the customer may choose to pay via cash, credit or check
- If one of the decision paths represents the overwhelming typical case, it should be the only one written in the *Typical Course of Events*, the alternatives in the *Alternatives* section
- When alternatives are all relatively equal and normal use the structure
  (1) Within *Typical Course of Events* of main section, indicate branches to subsections
  (2) One subsection for each branch using the same structure *Typical Course of Events*
  (3) If subsection has alternatives, write them in *Alternatives* section

58

Point-of-Sale System: Section Main

Typical Course of Events

Actor Action	**System Response**
2. . . .	
3. Customer chooses payment type: a. If cash payment, see section *Pay by Cash* b. If credit payment, see section *Pay by Credit* c. If check payment, see section *Pay by Check*	
	4. Logs the completed sale.
	5. Prints a receipt.
6. The Cashier gives the receipt to the Customer	
7. The Customer leaves with the items purchased.	

59

Point-of-Sale System: Section Pay by Cash

Typical Course of Events

Actor Action	System Response
1. The Customer gives a cash payment (cash tendered) possibly greater than the sale total.	
2. The Cashier records the cash tendered.	3. Shows the balance due back to the Customer.
4. The Cashier deposits the cash received and extracts the balance owing. The Cashier gives the balance to the Customer.	5. Prints a receipt.

Alternative Courses

- Line 1: Customer does not have sufficient cash. May cancel sale or initiate another payment method.
- Line 4: Insufficient cash in drawer to pay balance. Ask for cash from supervisor, or ask Customer for a payment closer to sale total.

60

Point-of-Sale System: Section Pay by Credit

Typical Course of Events

Actor Action	System Response
1. The Customer communicates their credit information for the credit payment.	2. Generates a credit payment request and sends it to an external Credit Authorization Service (CAS).
3. The CAS authorizes the payment.	4. Receives a credit approval reply from the CAS.
	5. Records the credit payment and approval reply information to the Accounts Receivable System (ARS). (The CAS owes money to the store, hence the ARS must track it).
	6. Displays authorization success message.

Alternative Courses

- Line 3: Credit request denied by CAS. Suggest different payment method.

61

Point-of-Sale System: Section Pay by Check

Typical Course of Events

Actor Action	System Response
1. The Customer writes a check and identifies self.	
2. Cashier records identification information and requests check payment authorization.	3. Generates a check payment request and sends it to an external Check Autorization Service (CAS).
4. The CAS authorizes the payment.	5. Receives a check approval reply from the CAS.
	6. Displays authorization success message.

Alternative Courses

- Line 3: Check request denied by CAS. Suggest different payment method.

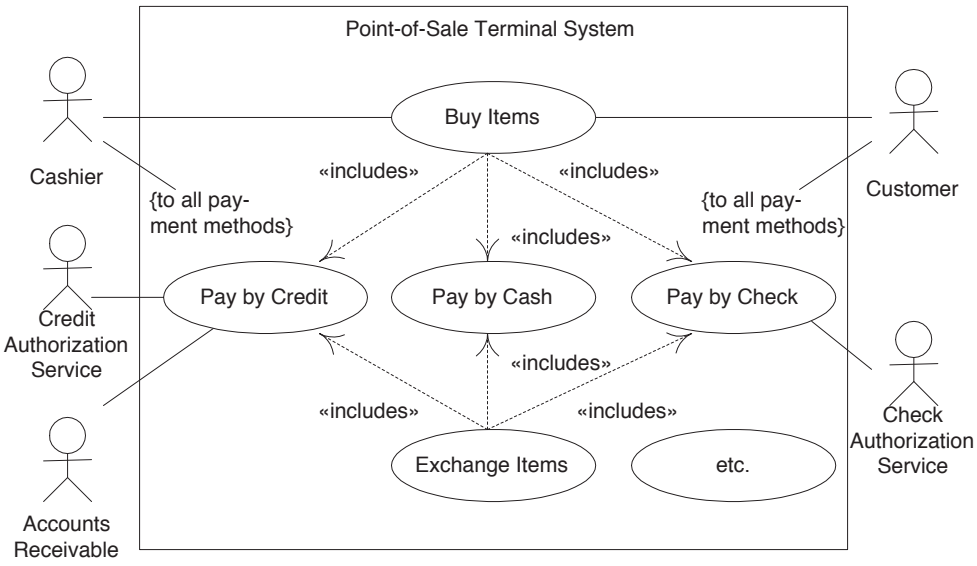
62

Relating Multiple Use Cases

- UML has special notation for illustrating use case relationships
- **Includes relationship**: One use case initiates or includes the behaviour of another use case
- **Extends relationship**: A second use case story follows a prior use case story
- These relationships are shown in the use case diagram
- Previously, different payment processes were written in subsections of Buy Items use case
- Alternatively they could have been split into separate use cases
- Heuristics: Write major steps or branching activities of a use case as separate use cases when
  - ◇ they are duplicated in other use cases
  - ◇ they are complex and long; separating them helps factor the use cases into manageable comprehensible units

63

Point-of-Sale Application: Relating Use Cases



Use Case Documents with Includes Relationship

- Includes relationships should be made explicit in use case documents

Use Case: Buy Items

. . .

Typical Course of Events

. . .

7. Customer chooses payment method:

a. If cash payment, **initiate** Pay by Cash

b. If credit payment, **initiate** Pay by Credit

c. If check payment, **initiate** Pay by Check

. . .

Alternative Courses of Events

. . .

Related Use Cases

◇ includes Pay by Cash

◇ includes Pay by Credit

◇ includes Pay by Check

Point-of-Sale System: Pay by Cash Use Case

Use case: **Pay by Cash**
Actors: Customer (initiator), Cashier
Overview: A customer pays for a sale by cash at a point-of-sale terminal.

66

Typical Course of Events

Actor Action	System Response
1. This use case begins when a Customer chooses to pay by cash, after being informed of the sale total.	
2. The Customer gives a cash payment (cash tendered) possibly greater than the sale total.	
3. The Cashier records the cash tendered.	4. Shows the balance due back to the Customer.
5. The Cashier deposits the cash received and extracts the balance owing. The Cashier gives the balance to the Customer.	

Alternative Courses

- Line 2: Customer does not have sufficient cash. May cancel sale or initiate another payment method.
- Line 4: Insufficient cash in drawer to pay balance. Asks for cash from supervisor, or asks Customer for a payment closer to sale total.

67

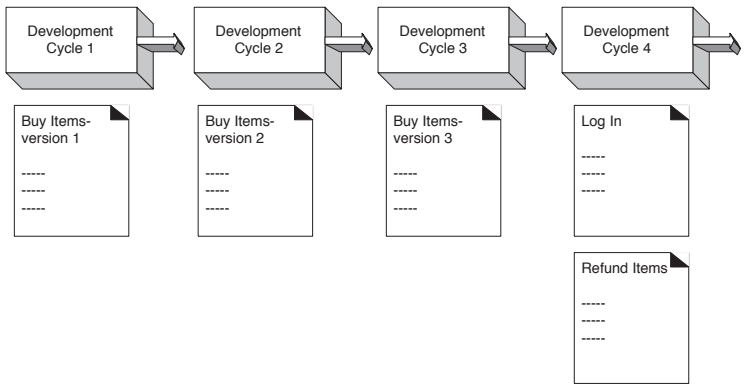
Use Cases in Plan and Elaborate Phase

(1) After system functions have been listed, define the system boundary and identify actors and use cases
(2) Write all use cases in high-level format. Categorize them as primary, secondary or optional
(3) Draw a use case diagram
(4) Relate use cases and illustrate relationships in the use case diagram (see later)
(5) Write the most critical, influential and risky use cases in the expanded essential format to better understand and estimate the nature and size of the problem. For the other use cases defer writing the expanded essential until the development cycles in which they will be tackled
(6) Ideally, real use cases sould be deferred until the design phase of a development cycle unless
   - Concrete descriptions significantly aid comprehension
   - Clients demand specifying their processes in this fashion
(7) Rank use cases (see later)

68

Scheduling Use Cases to Development Cycles

- Development cycles organized around use case requirements
- A cycle implement one or more use cases, or simplified versions of use cases when the complete use case is too complex to tackle in one cycle



69

Ranking Use Cases

- Use cases must be ranked
- High-ranking use cases need to be tackled in early development cycles
- Strategy: first pick use cases that significantly influence the core architecture
- Qualities that increase the ranking of a use case
  (1) significant impact of architectural design: adding many classes, persistence services
  (2) significant information and insight wrt design obtained with little effort
  (3) include risky, time-critical, or complex functions
  (4) involve significant research, new and risky technology
  (5) represent primary line-of-business processes
  (6) directly support increased revenue or decreased costs
- Ranking may be fuzzy (high, medium, low) or numerical (with weighting)

70

Point-of-Sale System: Ranking Use Cases

Rank	Use Case	Justification
High	Buy Items	Scores on most increased ranking criteria.
Medium	Add New Users	Affects security subdomain.
	Log In	Affects security subdomain.
	Refund Items	Important process; affects accounting.
Low	Cash Out	Minimal effect on architecture.
	Start Up	Definition is dependent on other use cases.
	Shut Down	Minimal effect on architecture.

71

"Start Up" Use Case

- Virtually all systems have a Start Up use case
- Necessary to tackle at least some simplified version of it in first development cycle
- Incrementally developed within each development cycle to satisfy start up needs of other use cases

72

Point-of-Sale System: Scheduling Use Cases

- Necessary to estimate if an entire use case can be tackled within the limited time-box of a cycle or if the use case must be distributed across multiple cycles
- Buy Items: complex use case requiring several development cycles
- Use case is redefined in terms of several use case versions
- Each encompasses more use case requirements, each limited to what is a reasonable amount of work in a cycle time-box (e.g. 4 weeks)
  - ◇ Buy Items version 1: cash payments, no inventory updates, ...
  - ◇ Buy Items version 2: allow all payment types
  - ◇ Buy Items version 3: complete version
- Simplifications, goals, and assumptions of each version must be stated
- Versions distributed over a series of development cycles along with other use cases

73

Use Case Buy Items: Version 1

- Cash payments only
- No inventory maintenance
- It is a stand-alone store, not part of a larger organization
- Manual entry of UPCs, no bar code reader
- No tax calculations
- No coupons
- Cashier does not have to log in; no access control
- No record of individual customers and their buying habits
- No control of the cash drawer
- Name and address of store, date and time of sale shown on the receipt
- Cashier ID and POST ID not shown on receipt
- Completed sales recorded in an historical log

74

Buy Items version 1

Use case:	Buy Items version 1
Actors:	Customer (initiator), Cashier
Purpose:	Capture a sale and its cash payment
Overview:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collect a cash payment. On completion the Customer leaves with the items
Type:	Primary and essential
Cross references :	Functions R1.1, R1.2, R1.3, R1.5, R1.7, R1.9, R2.1

75

Typical Course of Event

Actor Action	System Response
1. This use case begins when a Customer arrives at a POST checkout with items to purchase	
2. The Cashier records the universal product code (UPC) from each item. If there is more than one of the same item, the Cashier can enter the quantity as well.	3. Determines the item price and adds the item information to the running sales transaction. The description and price of the current item are presented.
4. On completion of item entry, the Cashier indicates to the POST that the item entry is complete.	5. Calculates and presents the sale total.
6. The Cashier tells the Customer the total.	
7. The Customer gives a cash payment (cash tendered) possibly greater than the sale total.	
8. The Cashier records the cash received amount.	9. Shows the balance due back to the Customer. Generate a receipt.
10. The Cashier deposits the cash received and extracts the balance owing. The Cashier gives the balance owing and the printed receipt to the Customer.	11. Logs the completed sale.
12. The Customer leaves with the items purchased.	

76

Use Case Buy Items: Version 2

- No inventory maintenace
- Stand-alone store, not part of a larger organization
- Manual entry of UPCs; no bar code reader
- No tax calculation
- No special pricing policies
- Cashier does not have to log in
- No record maintained of individual customers and their buying habits
- No control of the cash drawer
- Name and address of store and date ane time of sale are shown on the receipt
- Cashier ID and POST ID are not shown on receipt
- All completed sales are recorded in an historical log
- Only one payment, of one type, is used for a sale
- All payments are made in full, no partial or installment payments
- Check and credit payments are authorized

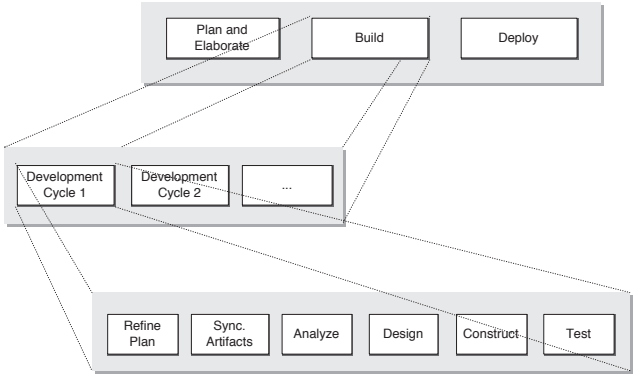
77

Use Case Buy Items: Version 2, cont.

- A different credit authorization service is used for each credit type (Visa, MasterCard, ...)
- The same authorization service is used for all checks
- The POST is responsible for communicating with the credit authorization service; the credit card reader is a dumb device that only sends the card information to the terminal
- Communication with an external service is via a modem. A phone number must be dialed each time
- Credit authorization services are usually provided by a bank
- Check and credit payments are for the exact amount of the sale total

78

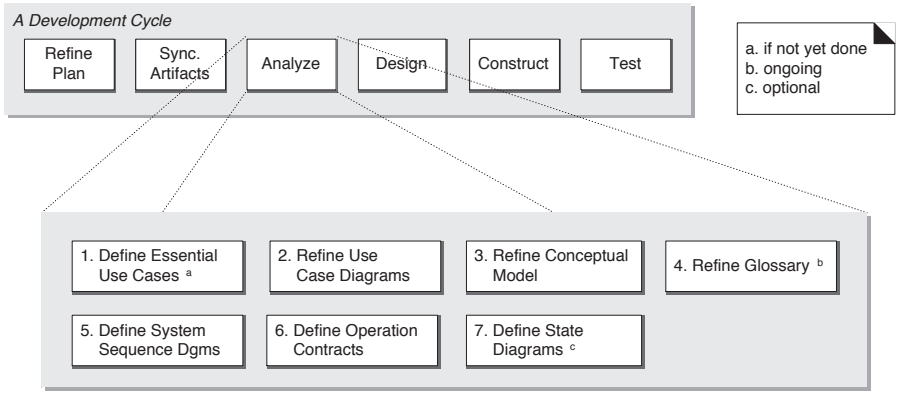
Starting a Development Cycle



- Build Phase: where iterative development cycles occur
- Initial activities within the cycle related to project management
- In parallel with a synchronization of documentation (e.g., diagrams) from the last cycle with the actual state of the code
- During the coding phase the design artifacts and the code invariably diverge

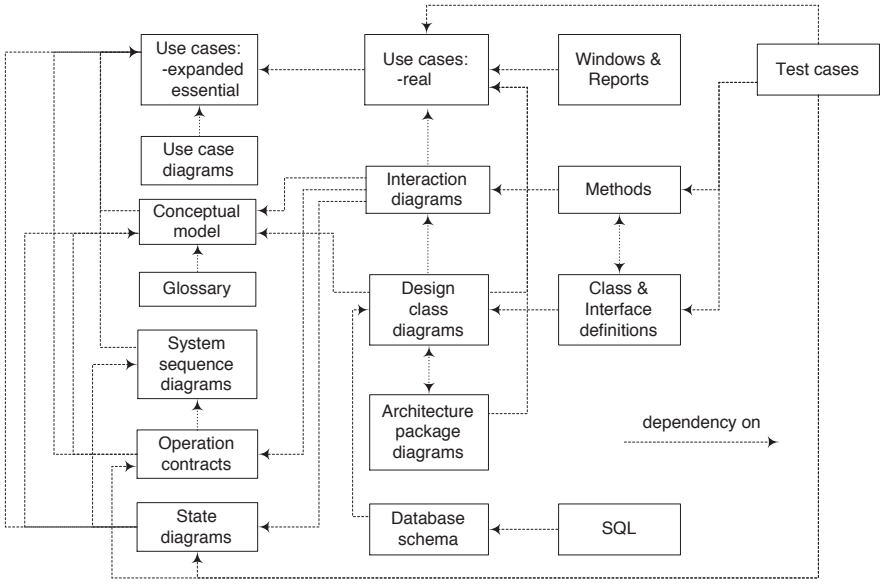
79

Analyze Phase Activities within a Development Cycle



80

Build Phase Activities within a Development Cycle



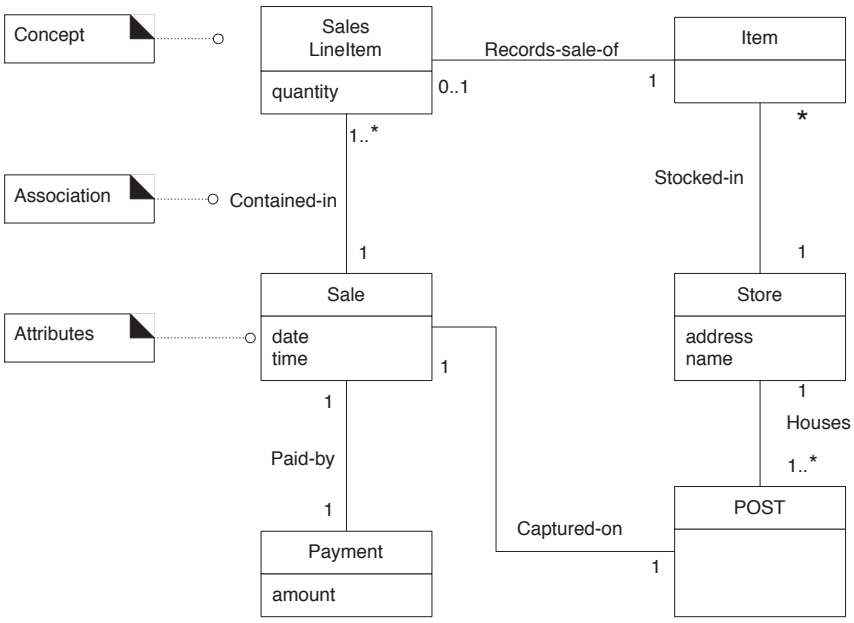
81

Conceptual Model

- Illustrates meaningful concepts (to the modelers) in a problem domain
- Most important artifact during OO analysis
- Objective: Identify a rich set of objects or concepts
- Aids in clarifying the terminology or vocabulary of the domain
- Critical quality of a conceptual model: it must be a representation of real-world things, not of software components
- Conceptual model must cover the use cases of the development cycle
- Creation depends on having use cases and other documents from which concepts can be identified
- In UML, a conceptual model is illustrated with a set of static structure diagrams in which no operations are defined
- Shows concepts, associations between concepts, attributes of concepts

82

Point-of-Sale System: Partial Conceptual Model



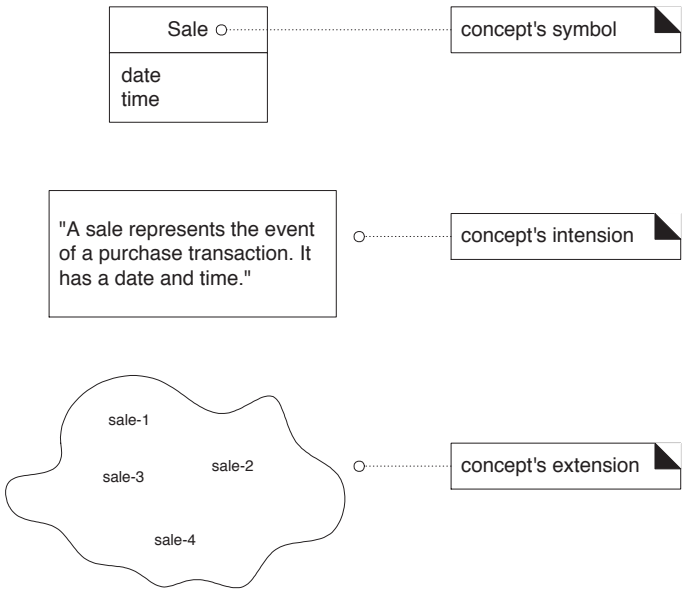
83

Concepts

- Informally, a concept is an idea, thing or object
- More formally, a concept may be considered in terms of
  - ◇ symbol: words or images representing a concept
  - ◇ intension: definition of a concept
  - ◇ extension: set of examples to which the concept applies
- Software problems can be complex
- Common strategy : Decomposition (divide and conquer) of the problem space into comprehensible units
- Dimension of decomposition
  - ◇ Structured analysis: processes or functions
  - ◇ OO analysis: concepts

84

Concepts



85

Point-of-Sale System: Concepts

86

Identifying Concepts

- Better to overspecify a CM with fine-grained concepts than to underspecify it
- Useful technique: candidate concepts or attributes are noun and noun phrases in textual descriptions of a problem domain
- Expanded use cases can be used for this purpose
- But, mechanical mapping is not possible, natural language is ambiguous

Typical Course of Event

Actor Action	System Response
1. This use case begins when a **Customer** arrives at a **POST checkout** with **items** to purchase	
2. The **Cashier** records the **universal product code** (UPC) from each **item**. If there is more than one of the same **item**, the **Cashier** can enter the **quantity** as well.	
	3. Determines the **item price** and adds the item information to the **running sales transaction**. The **description** and **price** of the current **item** are presented.

87

Report Objects

- Include Receipt in conceptual model ?
- Receipt: record of a sale
- Showing a report in a conceptual model is not useful: its information derived from other sources
  ⇒ one reason to exclude it
- Receipt has also special role in business rules: confers the right to the bearer to return bought items
  ⇒ one reason to show it
- Since item returns are not considered in this development cycle, it is excluded
- Should be included in the development cycle tackling Return Items use case

88

Different Categories of Concepts

Concept Category	Examples
physical or tangible objects	POST, Airplane
specifications, or descriptions of things	ProductSpecification, FlightSpecification
places	Store, Airport
transactions	Sale, Payment, Reservation
transaction line items	SalesLineItem
roles of people	Cashier, Pilot
containers of other things	Store, Bin, Airplane
things in a container	Item, Passenger
computer or mechanical external systems	CreditCardAuthorizationSystem
abstract concepts	Illness, Failure
organizations	SalesDepartment, Airline
events	Sale, Meeting, Flight, Crash, Landing
processes	SellingAProduct, BookingASeat
rules and policies	RefundPolicy, CancellationPolicy
catalogs	ProductCatalog, PartsCatalog
records of finance, contracts, legal matters	Receipt, Ledger, EmployementContract
financial instruments and services	LineOfCredit, Stock
manuals, book	EmployeeManual, RepairManual

89

Guidelines for Making a Conceptual Model

- How to make a CM
  (1) List the candidate concepts
  (2) Draw them in a conceptual model
  (3) Add the associations necessary to record relationships for which there is a need to preserve some memory
  (4) Add the attributes necessary to fulfill the information requirements
- Naming and modeling things: make a conceptual model in the spirit of how a cartographer works
  (1) Use the vocabulary of the domain when naming concepts and attributes
  (2) Exclude concepts in the problem domain not pertinent to the requirements
  (3) Exclude things not in the problem domain in consideration

90

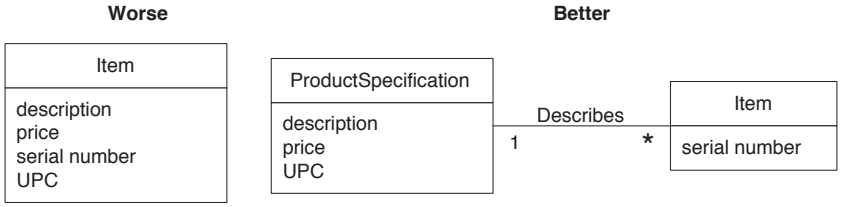
Guidelines for Making a Conceptual Model, cont.

- Rule of thumb: A conceptual model is not absolutely correct or wrong, but more or less useful; it is a tool of communication
- Common mistake: represent something as an attribute when it should be a concept
- Resolving similar concepts, e.g., POST vs Register
  - ◇ the same concept may have different names
  - ◇ sometimes called differently by different groups of users
  - ◇ sometimes subtle differences between these concepts

91

Description Concepts

- A concept of objects that are specifications or description of other things



- Disadvantages of first solution
  - ◇ when last item sold, loss of information that needs to be maintained
  - ◇ attributes repeated for each item of the same type
- Typical of sales and products domains, also in manufacturing

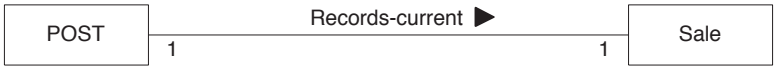
92

Defining Terms in UML

- Terms "class" and "type" are used in UML, but not "concept"
  - ◇ no universal agreement on the meaning of class and type
  - ◇ to avoid ambiguity UML defines these terms in its metamodel
- Bottom line: distinguish between
  - ◇ perspective of domain analysts looking at real-world concepts and
  - ◇ software engineers specifying software entities as classes in an OPL
- UML can be used for both perspectives with similar notation and terminology
- **Class**: description of a set of objects that share the same attributes, operations, methods, relationships, and semantics
  - ◇ distinguish implementation class
- **Type**: similar to a class but may not include any method
  ⇒ specification of a software entity, rather than an implementation
- **Interface**: set of externally visible operations
  - ◇ may be associated with types, classes, and packages
  - ◇ typically used for software entities

93

UML Associations

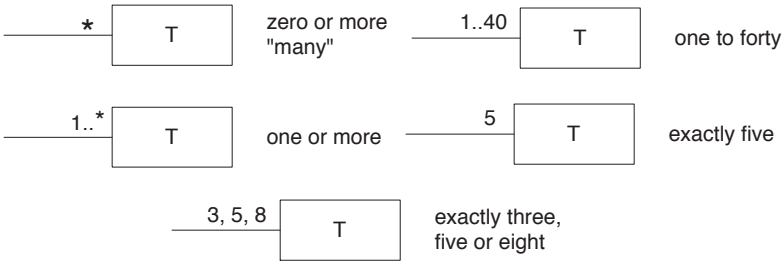


- Relationship between concepts that indicates some meaningful connection
- Definition: structural relationships between objects of different types
- Represented as a line between concepts with association name
- Inherently bidirectional
- Abstract traversal, not a statement about connections between software entities
- Optional reading arrow without semantic meaning
  - ◇ indicates the direction to read the association name
  - ◇ does not indicate direction of visibility or navigation
  - ◇ if not present, convention to read association left to right, or top to bottom

94

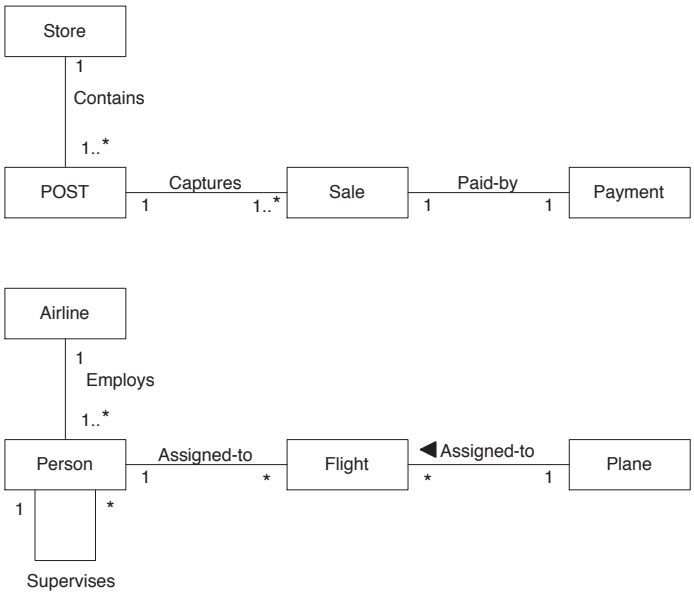
Multiplicity

- **Role**: and end of an association
- In addition to name and navigability, may contain a multiplicity expression
- Defines how many instances of a type A can be associated with one instance of a type B, at a particular moment in time
- UML notation for multiplicity



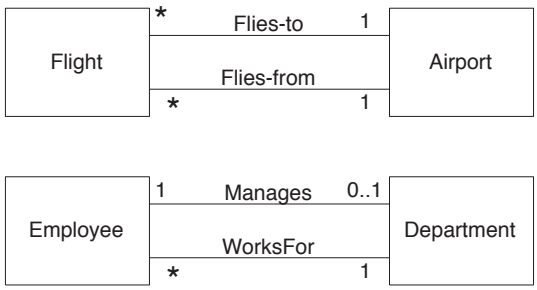
95

Example Associations



96

Multiple Associations between Two Types



- Two types may have multiple associations between them
- Sometimes there is a constraint between both associations
- E.g., the manager of a department must work on that department

97

Associations and Implementation

- During analysis, an association is not a statement about data flows, instance variables, object connections
- It is a statement that a relationship is meaningful in the real-world
- Many relationships will typically be implemented as paths of navigation or visibility, but their implementation is not required in a CM
- Deferring design considerations frees from extraneous information and decisions in the analysis model, maximizes options later on
- A CM may contain associations that are necessary during construction
- Also, associations needed to be implemented may be missed during analysis
  $\Rightarrow$ CM should be updated to reflect this

98

Association Guidelines

- Focus on associations for which knowledge needs to be preserved for some duration ("need-to-know" associations)
- It is more important to identify concepts than to identify associations
- Too many associations tend to confuse a CM, their discovery may be time-consuming with marginal benefit
- Avoid showing redundant or derivable associations
- Name associations based on a TypeName–VerbPhrase–TypeName format where this creates a readable and meaningful sequence

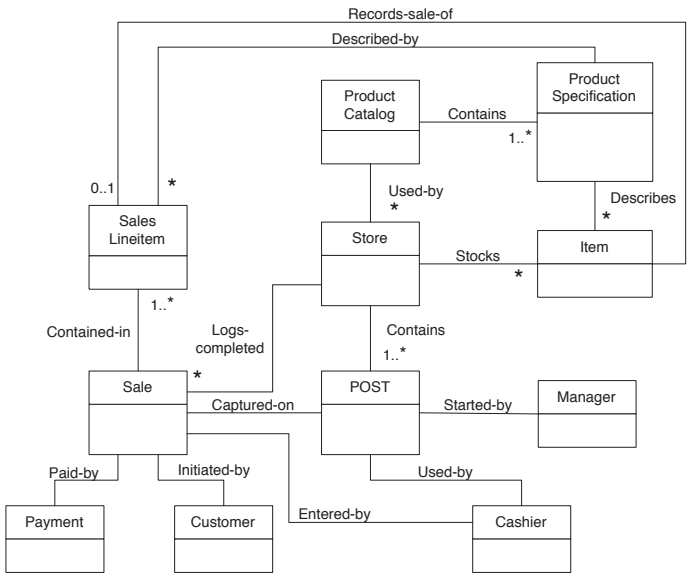
99

Different Categories of Associations

Category	Examples
A is physical part of B	Drawer—Post, Wing—Airplane
A is logical part of B	SalesLineItem—Sale, FlightLeg—FlightRoute
A is physically contained in/on B	POST—Store, Passenger—Airplane
A is logically contained in/on B	ItemDescription—Catalog, Flight—FlightSchedule
A is description for B	ItemDescription—Item, FlightDescription—Flight
A is a line item of a transaction B	SalesLineItem—Sale, MaintenanceJob—MaintenanceLog
A is known/logged/recorded/reported/ captured in B	Sale—POST, Reservation—FlightManifest
A is member of B	Cashier—Store, Pilot—Airline
A is an organization subunit of B	Department—Store, Maintenance—Airline
A uses or manages B	Cashier—POST, Pilot—Airplane
A communicates with B	Customer—Cashier, ReservationAgent—Passenger
A is related to a transaction B	Customer—Payment, Passenger—Ticket
A is a transaction related to another transaction B	Payment—Sale, Reservation—Cancellation
A is next to B	POST—POST, City—City, Room—Room
A is owned by B	POST—Store, Plane—Airline

100

Point-of-Sale System: Conceptual Model



101

Not every association shown is compelling.

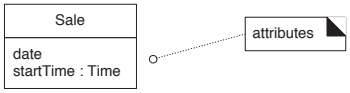
- Sale EnteredBy Cashier: Requirements do not indicate the need to record the current cashier. Also it is derivable from association POST UsedBy Cashier.
- Post UsedBy Cashier: Requirements do not indicate the need to record the current cashier.
- POST StartedBy Manager: Requirements do not indicate the need to record the manager that starts up a POST.
- Sale InitiatedBy Customer: Requirements do not indicate the need to record the current customer who initiates a sale.
- Store Stocks Item: Requirements do not indicate the need to record inventory information.
- SalesLineItem RecordsSaleOf Item: Requirements do not indicate the need to record inventory information.

Need-to-Know vs Comprehension Associations

- Strict need-to-know criterion for maintaining associations generates a minimal information model, bounded by requirements
- But, may create a model which does not convey a full understanding of the domain
- A CM is also seen as a tool of communication for understanding the important concepts and their relationships
- From this viewpoint, deleting some not-required associations can create a model which misses the point
  - ◇ E.g., Sale InitiatedBy Customer is not needed in a strict need-to-know basis
  - ◇ But its absence leaves out an important aspect in understanding the domain
- Good model: in the middle between a minimal need-to-know model and one which illustrates every conceivable relationship
- Approach: emphasize need-to-know associations, add comprehension-only associations to enrich critical understanding of the domain

Attributes

- Logical data value of an object
- Approach: in a CM include attributes for which the requirements (use cases) suggest or imply a need to remember information
- UML notation: attribute's type may optionally be shown

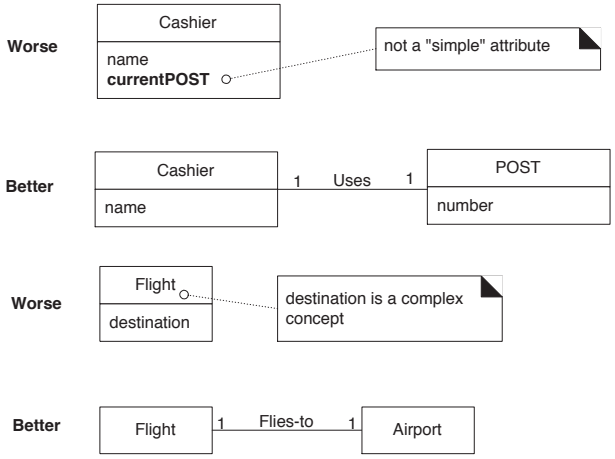


- Attributes in a CM should be simple attributes or pure data values
- Common simple attribute types: Boolean, Date, Number, String, Text, Time
- Other common types: Address, Color, Geometrics, Phone Number, Social Security Number, ZIP or Postal Codes, enumerated types

103

Attributes vs Associations

- Type of an attribute should not be a complex domain concept, e.g. Sale or Airport
- Concepts should be related with associations, not with an attribute



104

Pure Data Values

- Known as **data types** in UML
- Those for which unique identity is not meaningful
- E.g., not meaningful to separate
  - ◇ instances of Number 5
  - ◇ instances of String 'dog'
  - ◇ instances of PhoneNumber containing the same number
  - ◇ instances of Address containing the same address
- In contrast, meaningful to distinguish two instances of Person having the same name
- Identity vs Equality
- Element of a pure data value may be illustrated as an attribute
- But, it is also acceptable to model it as a distinct concept

105

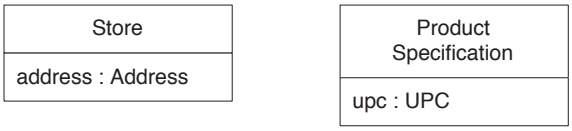
Non-primitive Attribute Types

- A primitive data type (string, number) may be represented as a non-primitive data type if
  - ◇ it is composed of separate sections: phone number, name of person
  - ◇ has associated operations, e.g., for parsing, validations: SSN, bank acount
  - ◇ has other attributes: promotional price has a start and end date
  - ◇ is a quantity with a unit: payment amount has unit of currency
- Non-primitive attribute types in the point-of-sale application
  - ◇ Universal Product Code (UPC): check-sum to validate, have attributes (manufacturer who assigned it)
  - ◇ Price and amount: non-primitive Quantity types because of unit of currency
  - ◇ Address attribute: separate sections

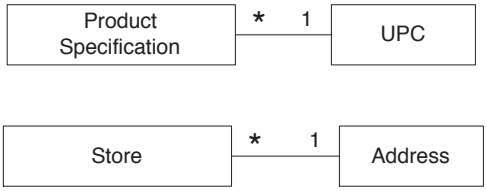
106

Non-primitive Attribute Types and Pure Data Values

- If the attribute type is a pure data value, it may be shown as attribute



- Non-primitive types, with attributes and associations, may be be shown as concepts

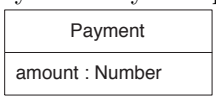


- Both approaches are valid
- Depend on how the CM is being used as tool of communication, and the significance of the concept in the domain

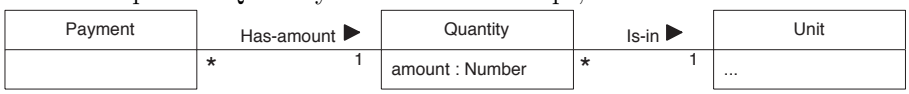
107

Attribute Quantities and Units

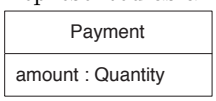
- Attribute such as amount of Payment may be represented as a Number



- In the general case this is not robust or flexible: units of a number are important
- Similarly, for velocity, currency
- Common requirement: units must be converted, e.g., imperial to metric
- Solution: represent Quantity as a distinct concept, with associated unit



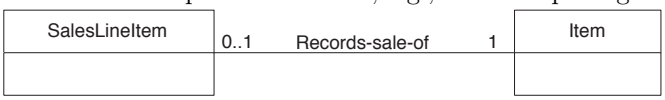
- It is also a pure value: may be represented as an attribute



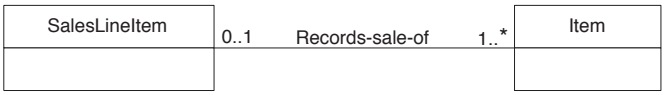
108

Multiplicity from SalesLineItem to Item

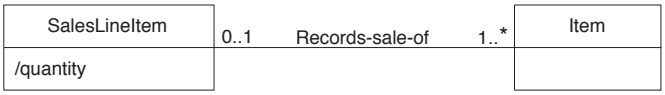
- Each line item records a separate item sale, e.g., 1 biscuit package



- Each line item can record a group of the same kind of items, e.g., 6 biscuit packages

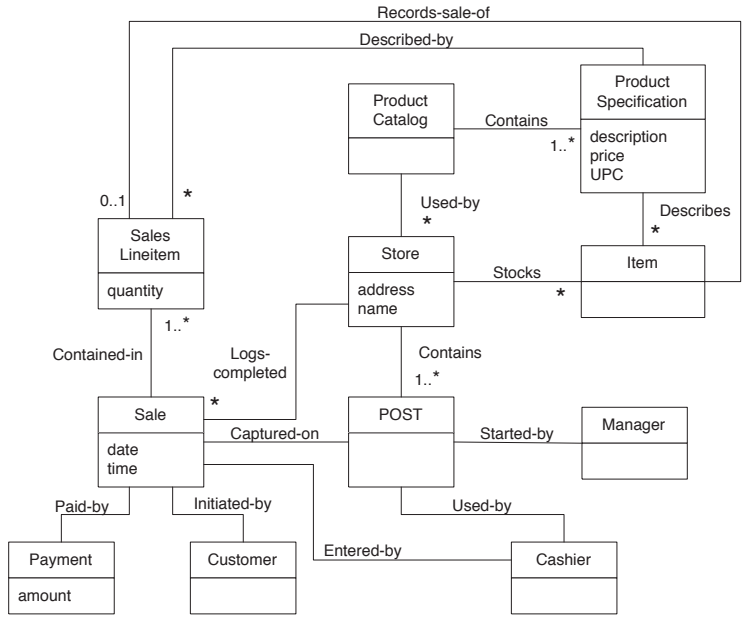


- Derived attribute from the multiplicity value



109

Point-of-Sale System: Conceptual Model



110

Cycle 2: Extending the Conceptual Model

- CM incrementally developed by considering current use cases for this cycle
- New concepts POST system:

Category	Examples
Physical or tangible objects	CreditCard, Check
Transactions	CashPayment, CreditPayment, CheckPayment
Organizations	CreditAuthorizationService, CheckAuthorizationService
Records of finance, work, contracts, legal matters	AccountsReceivable

111

Point-of-Sale System: Pay by Credit Use Case

Typical Course of Events

Actor Action	System Response
1. This use case begins when a Customer chooses to pay by credit, after being informed of the sale total.	
2. The Customer swipes their **credit card** through a card reader in order to complete the **credit payment**.	3. Generates a **credit payment request** and sends it to an external **Credit Authorization Service** (CAS) via a modem attached to the POST. Requires dialing the service, sending out a request record, and waiting for a reply record.
	4. Receives a **credit approval reply** from the CAS. The reply is encoded in a reply record and received via the modem.
	5. Posts (records) the credit payment and approval reply information to the **Accounts Receivable System** (ARS). (The CAS owes money to the store, hence the ARS must track it).
	6. Displays authorization success message.

Alternative Courses

- Line 4: Credit request denied by CAS. Suggest different payment method.

112

Point-of-Sale System: Section Pay by Check

Typical Course of Events

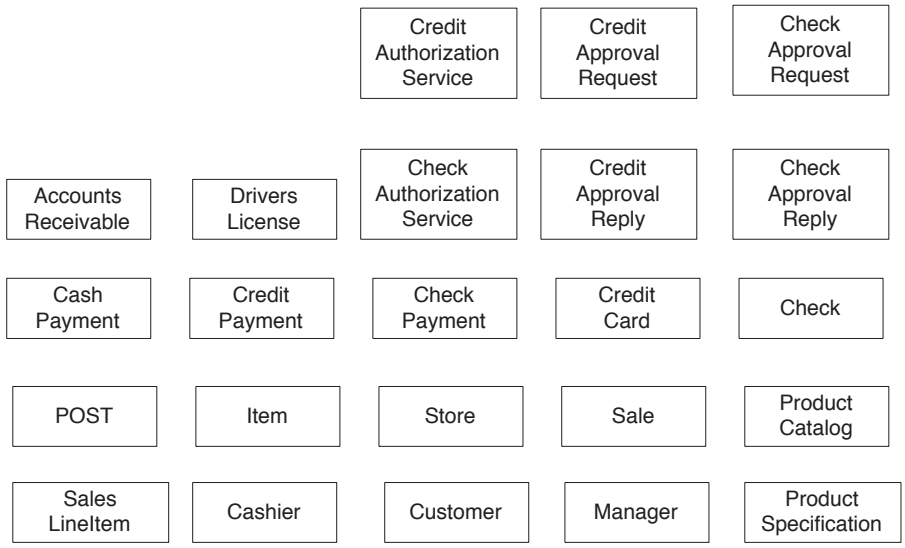
Actor Action	System Response
1. This use case begins when a Customer chooses to pay by check, after being informed of the sale total.	
2. The Customer writes a **check** and gives it and their **drivers license** to the Cashier.	
3. Cashier writes the **drivers license number** on the check, types it into the DL Number text field on the window and presses the **Check Authorization** button to requests check payment authorization.	
	4. Generates a **check payment request** and sends it to an external **Check Autorization Service** (CAS) via a **modem** attached to the POST. Requires dialing the service, sending out a request record, and waiting for a reply record.
	5. Receives a **check approval reply** from the Check Authorization Service. The reply is encoded in a reply record and received via the modem.
	6. Displays authorization success message.

Alternative Courses

- Line 5: Check request denied by CAS. Suggest different payment method.

113

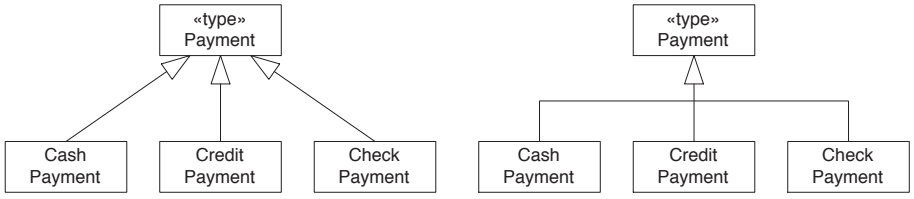
Point-of-Sale Application: Extending the Conceptual Model



114

Generalization

- Concepts CashPayment, CreditPayment, and CheckPayment are very similar
- Should be organized in to a **generalization-specialization type hierarchy**
- **Supertype**: more general concept, **subtype**: more specialized concept
- UML notations

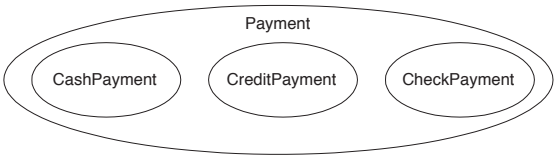


- **Generalization** allows to construct taxonomic classifications among concepts
- Allows us to understand concepts in more general, refined and abstract terms
- Leads to economy of expression, improved comprehension, reduction in repeated information

115

Generalization Characteristics

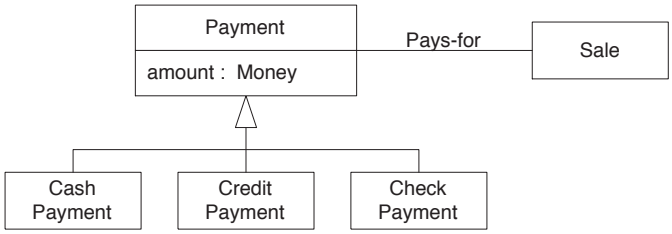
- **Type definition**: A supertype definition is more general or encompassing than a subtype definition
- **Membership inclusion**: All the members of a subtype set are members of their supertype set
  - ◇ Is-a rule: the subtype *is a kind of* the supertype



116

Generalization Characteristics, cont.

- **Subtype conformance**: 100% of the supertype's definition should be applicable to the subtype



117

Motivation to Partition a Type into Subtypes

- Subtype has additional attributes of interest
  - ◇ Library: Book, subtype of LoanableRessource, has an ISBN attribute
- Subtype has additional associations of interest
  - ◇ Library: Video, subtype of LoanableRessource, is associated with Director
- Subtype concept is operated upon, handled, reacted to or manipulated differently than the supertype or other subtypes, in ways that are of interest
  - ◇ Library: Software, subtype of LoanableRessource, requires a deposit before it may be loaned
- Subtype concept represents an animate thing that behaves differently than the supertype or other subtypes in ways that are of interest
  - ◇ Market Research: MaleCustomer and FemaleCustomer behaves differently wrt shopping habits

118

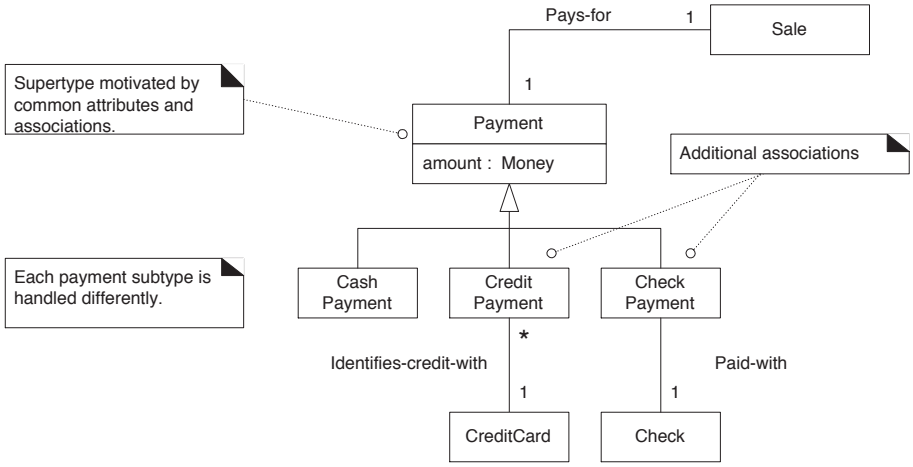
When to Define a Supertype

Motivated when commonality is identified among potential subtypes

- Potential subtypes represent variations on a similar concept
- Subtypes will conform to the 100% and Is-a rules
- All subtypes have the same attribute which can be factored out and expressed in the supertype
- All subtypes have the same association which can be factored out and related to the supertype

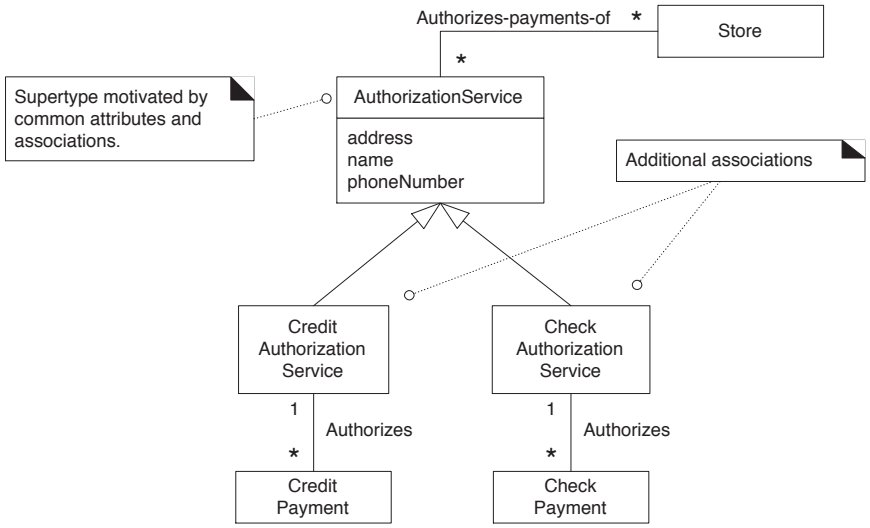
119

Point-of-Sale Application: Payment Subtypes



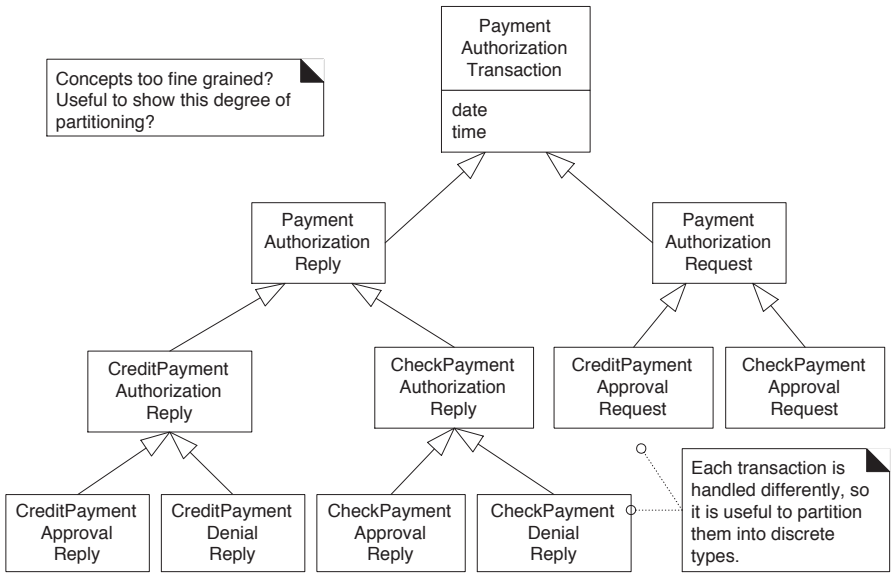
120

Point-of-Sale Application: AuthorizationService Hierarchy



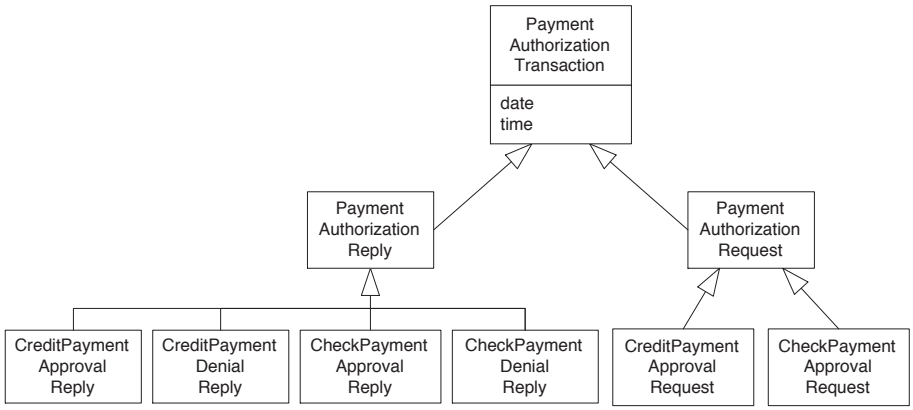
121

Point-of-Sale Application: External Service Transactions



122

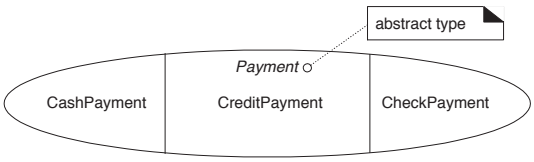
Point-of-Sale Application: External Service Transactions



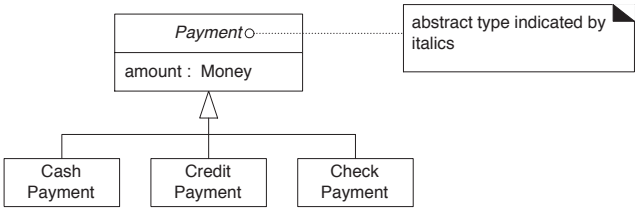
123

Abstract Types

- Every member of a type T must also be a member of a subtype



- UML notation

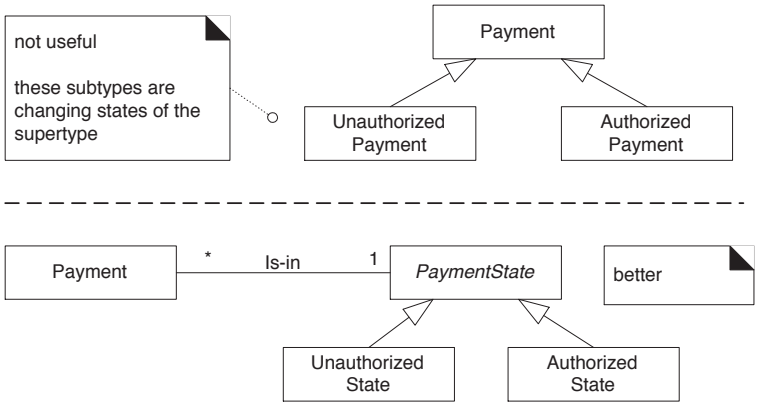


- Implemented as **abstract classes** (without instances) during design
- **Abstract method**: declared in abstract class, but not implemented

124

Modeling Changing States

- Do not model states of a concept X as subtypes of X, rather
  - ◇ Define a state hierarchy and associate the states with X, or
  - ◇ Ignore showing the states on a concept in the conceptual model; show them in state diagrams



125

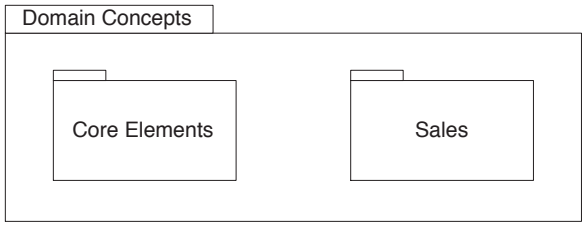
Class Hierarchies and Inheritance

- We have not mentioned inheritance because CM focus on real-world things not software artifacts
- A class is a software implementation of a concept or type
- In OPL, a subclass **inherits** the attribute and operation definitions of its superclasses
- **Inheritance**: software mechanism to implement subtype conformance to supertype definitions
- Inheritance has no real part to play in the conceptual model
- It most definitely does when we transition to the design phase
- Type hierarchies of the CM may or may not be reflected in solution
  - ◇ e.g., hierarchy of authorization service transaction types may be collapsed or expanded into alternate software class hierarchies
  - ◇ depends upon language features and other factors

126

Packages

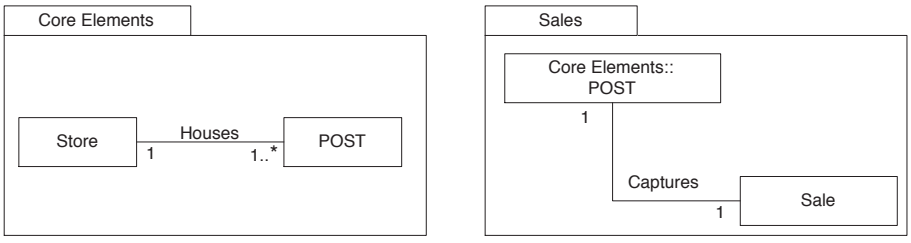
- Mechanism allowing to partition model elements into smaller subsets
- Support a higher-level view
- Overall system architecture composed of vertical layers and horizontal partitions
- Packages of the CM, if carried through design, may be considered partitions of the domain objects layer
- UML Notation



127

Ownership and References

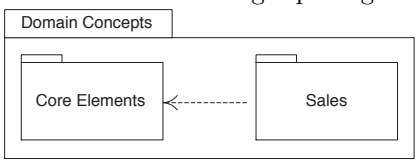
- An element (e.g., type) is **owned** by the package within which it is defined
- May be **referenced** in other packages ⇒ element name qualified using the format PackageName::ElementName
- A type or class in a foreign package may be modified with new associations, but must otherwise remain unchanged



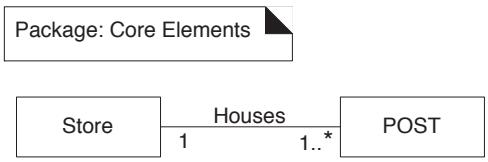
128

Dependencies

- **Dependency relationship**: A model element is in some way dependent on another
- **Package dependency**: Elements of the dependent package in some way know about or are coupled to elements in the target package



- Package ownership with a constraint note on the diagram



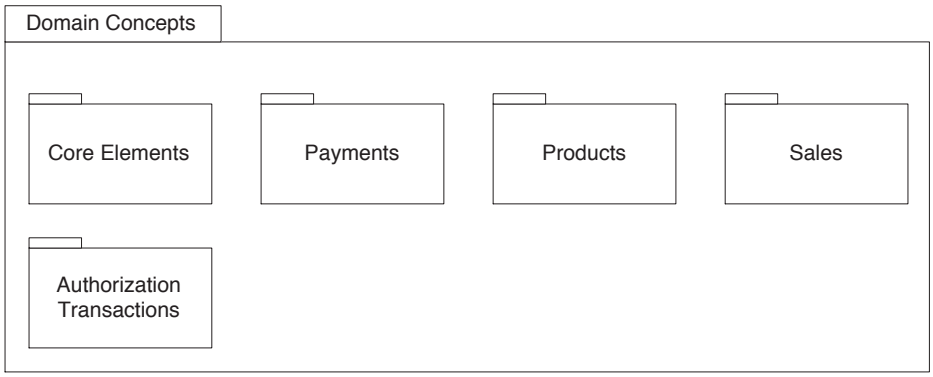
129

Guidelines for Partitioning the Conceptual Model

- Place together elements that
  - ◇ are in the same subject, are closely related by concept or purpose
  - ◇ are in a type hierarchy together
  - ◇ participate in the same use case
  - ◇ are strongly associated
- All elements related to the CM should be rooted in a Domain Concepts package
- Widely shared, core concepts should be defined in a Core Elements or Common Concepts package

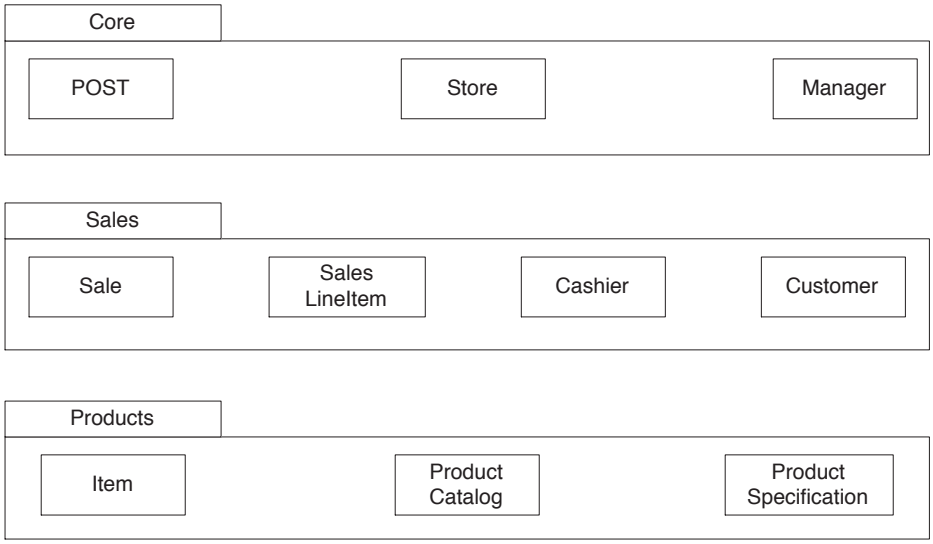
130

Point-of-Sale Application: Domain Concept Packages



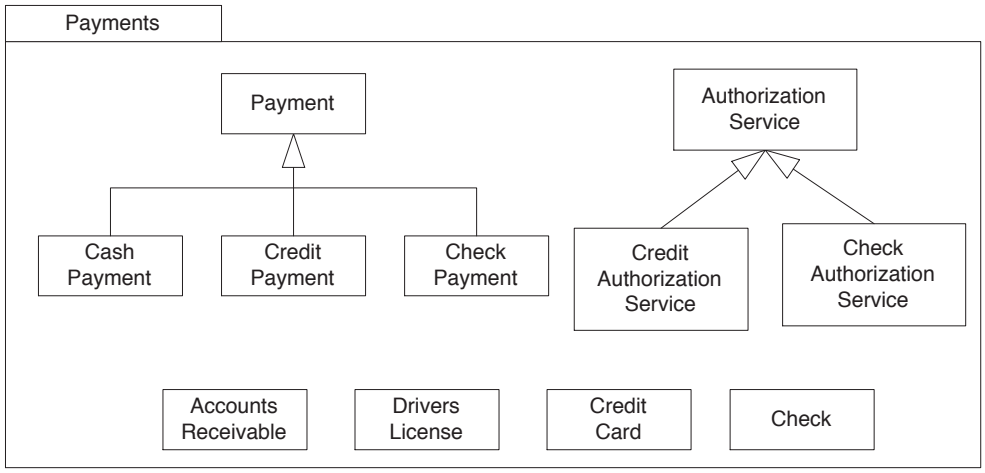
131

Point-of-Sale Application: Domain Packages



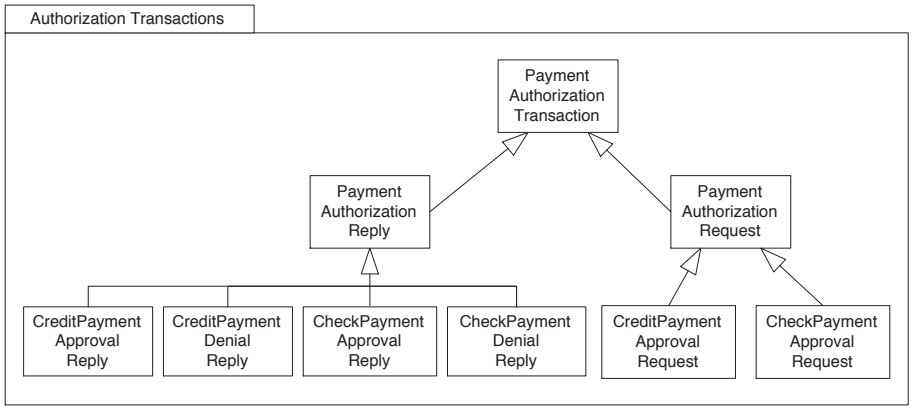
132

Point-of-Sale Application: Domain Packages, cont.



133

Point-of-Sale Application: Domain Packages, cont.



134

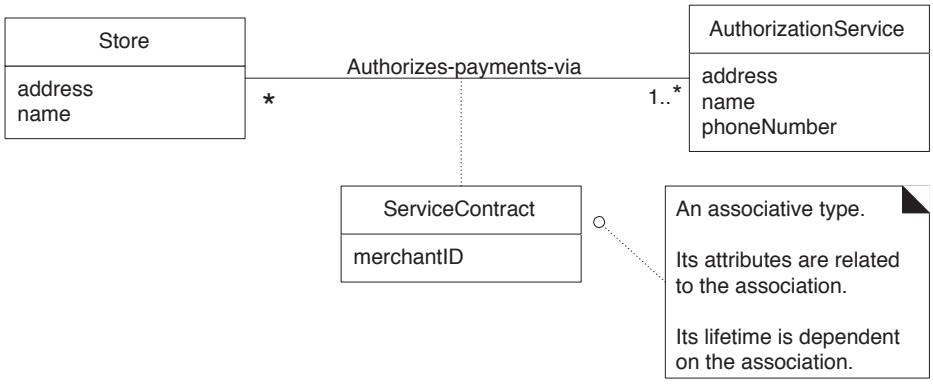
Associative Types

- Relationships may also have attributes
- These attributes are related to the association and cannot be placed to the participating classes

135

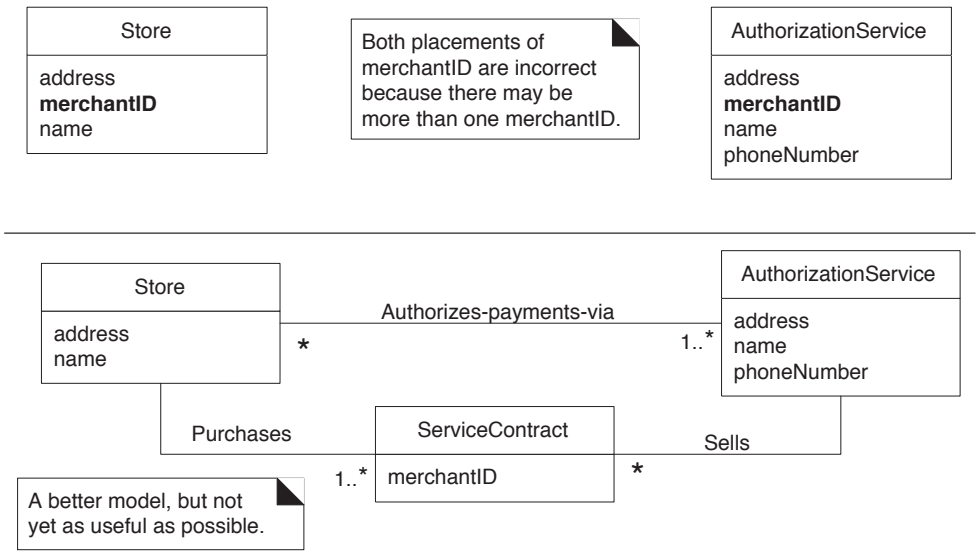
Associative Types: Example

- Authorization services assign a merchant ID to each store for identification during communications
- A payment authorization request requires the inclusion of the merchant ID that identifies the store to the authorization service
- A store has different merchant ID for each service



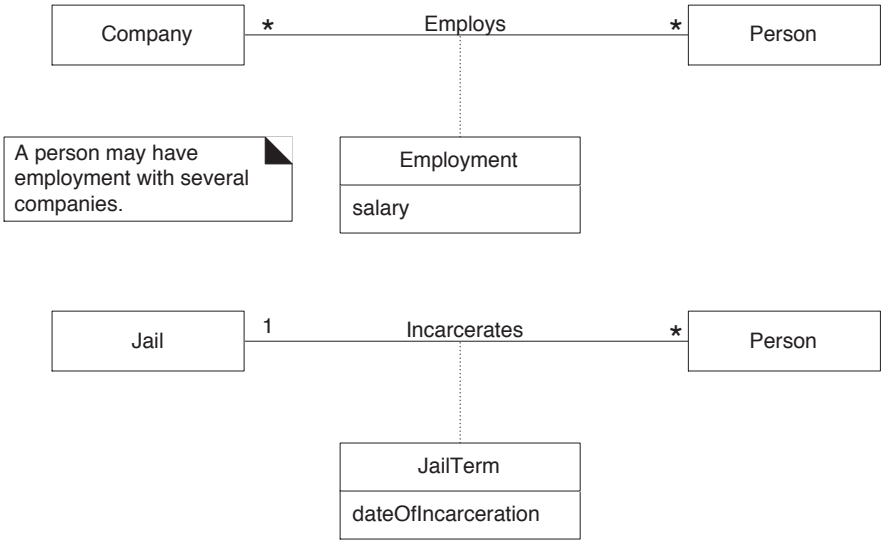
136

Merchant ID: Inadequate Alternatives



137

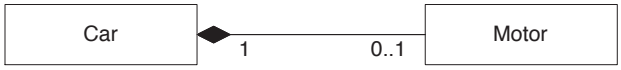
Associative Types: Other Examples



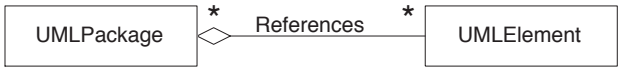
138

Aggregation

- Particular association used to model whole-part relationships between things
- Whole is called **the composite**, parts called **component**
- UML notation: hollow or filled diamond and the composite end
- **Composite aggregation**: multiplicity at the composite end is at most one, i.e., composite solely owns the part
  - ◇ Typical of physical aggregations



- **Shared aggregation**: multiplicity at the composite end may be more than one, i.e., the part may be in many composite instances
  - ◇ Involves nonphysical concepts



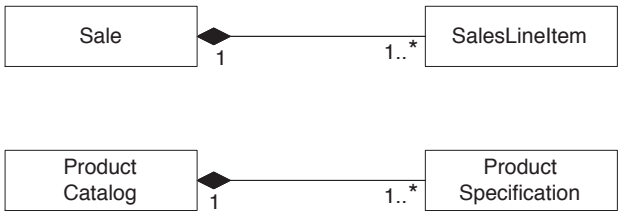
139

Guidelines for Identifying Aggregation

- Consider aggregation when
  - ◇ Lifetime of the part is bound within the lifetime of the composite — there is a create-delete dependency of the part on the whole
  - ◇ There is an obvious whole-part physical or logical assembly
  - ◇ Some properties of the composite propagate to the parts, such as its location
  - ◇ Operations applied to the composite propagate to the parts, such as destruction, movement, recording

140

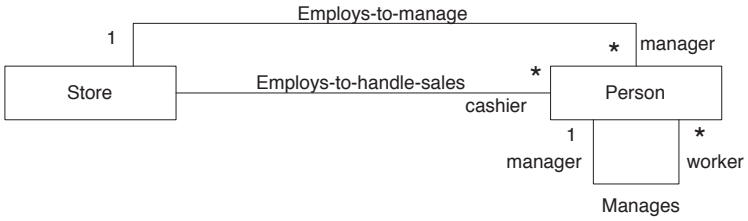
Point-of-Sale Application: Aggregations



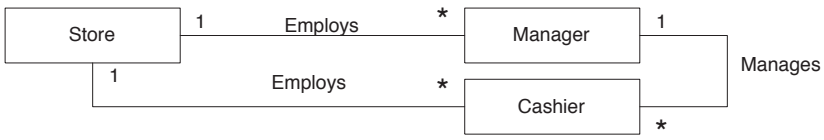
141

Roles as Concepts vs Roles in Associations

- Roles in associations



- Roles as concepts

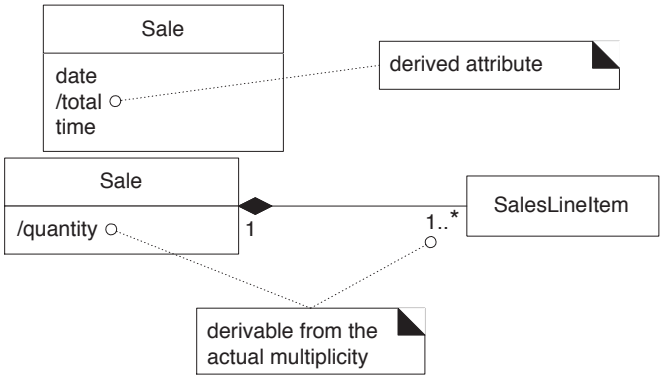


142

- Roles in associations
  - ◇ allow to express that the same instance takes on multiple (dynamically changing) roles in various associations
- Roles as concepts
  - ◇ Allows to add additional semantics: attributes, associations
  - ◇ better support for mutate an instance of one class into another class, or adding additional behaviour and attributes as the role of a person changes

Derived Elements

- Can be derived from other elements
- Attributes and associations are the most common derived elements

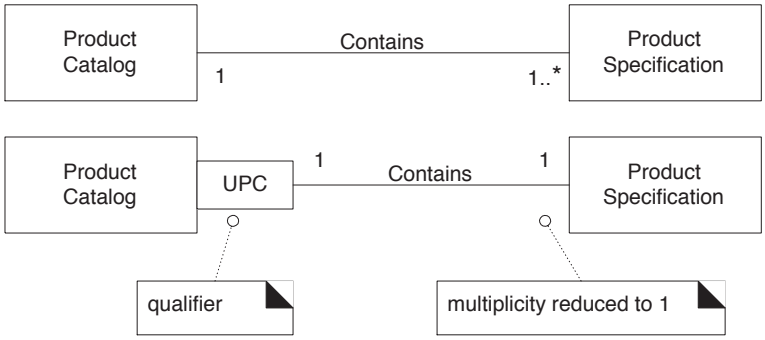


- Showing derived elements in a diagram
  - ◇ add complexity without new information
  - ◇ helps comprehension, may be prominent in the terminology of the domain

143

Qualified Associations

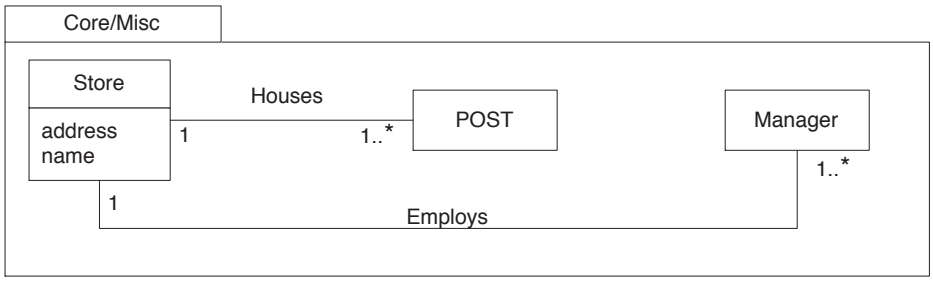
- **Qualifier**: distinguishes the set of objects at the far end of an association based upon the qualifier value
- **Qualified association**: has a qualifier



- Depicts how in the domain, things of one type are distinguished in relation to another type
- May be implemented during design with lookup keys

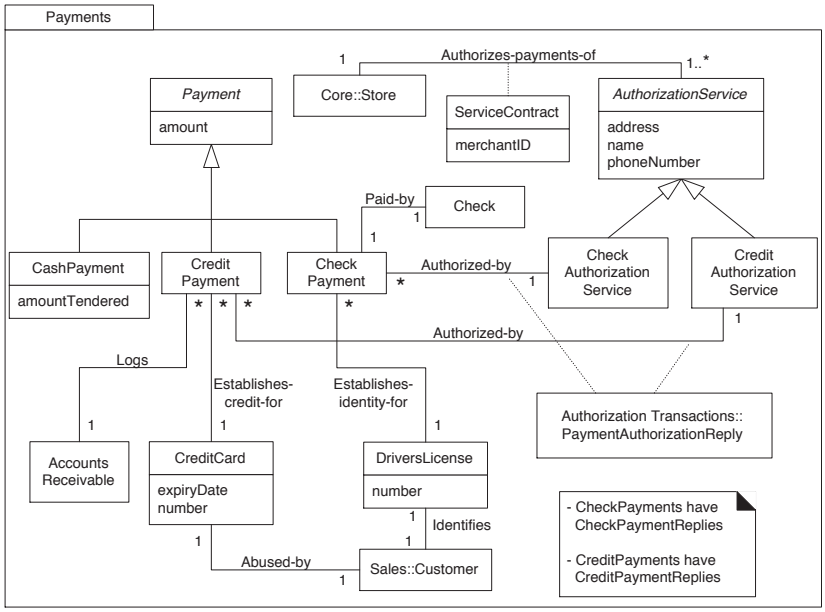
144

Point-of-Sale Application: Core/Misc Package



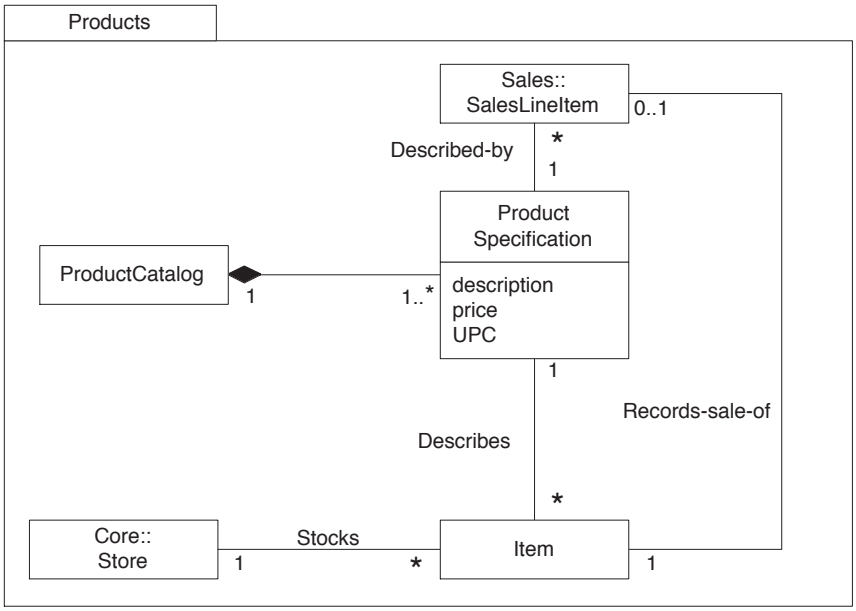
145

Point-of-Sale Application: Payments



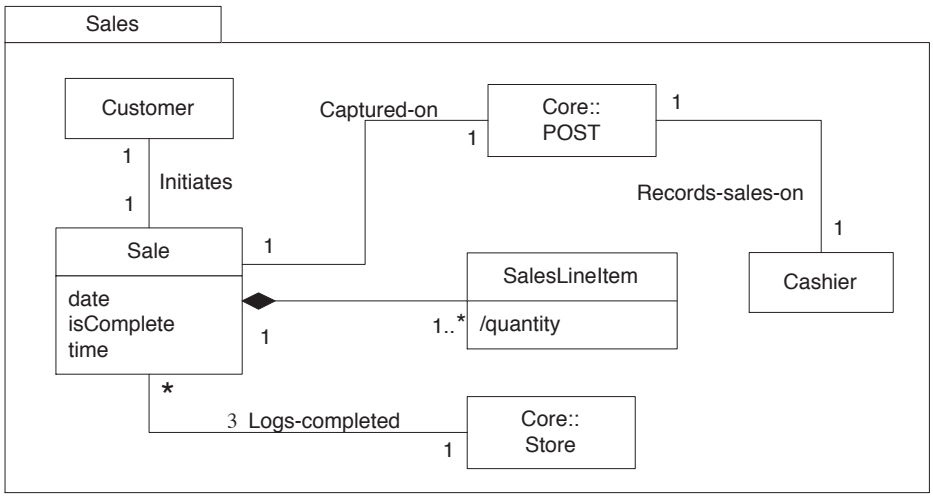
146

Point-of-Sale Application: Products



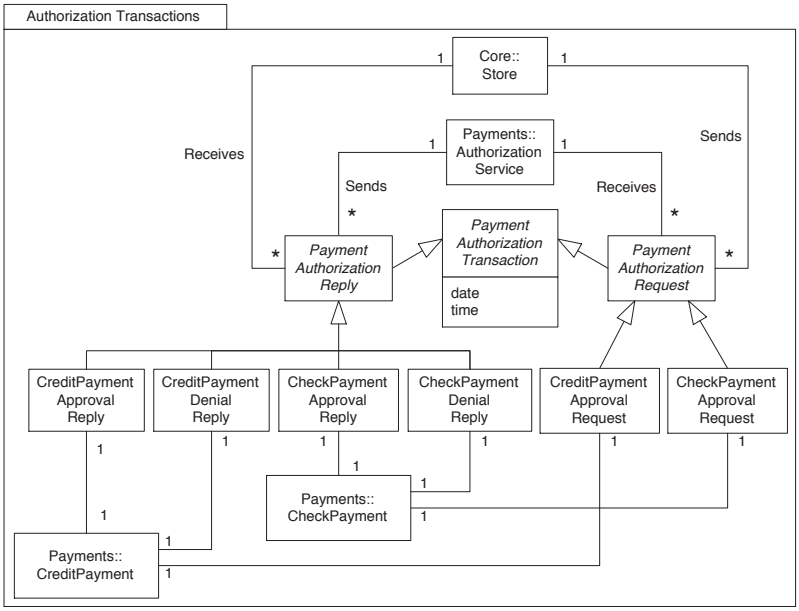
147

Point-of-Sale Application: Sales



148

Point-of-Sale Application: Authorization Transactions



149

Conclusion

- There is no such thing as a correct conceptual model
- All models are approximations of the domain we are attempting to understand
- A good conceptual model
  - ◇ captures the essential abstractions and information required to understand the domain in the context of current requirements
  - ◇ aids people in understanding the domain: its concepts, terminology, and relationships

150

Glossary or Model Dictionary

- Lists and defines all terms
- Improve communication and reduce risk of misunderstanding
- Consistent meaning and shared understanding of terms is extremely important during application development
- Especially when many team members are involved
- Originally created during the Plan and Elaborate Phase as terms are generated
- Continually refined within each development cycle as news terms are encountered
- Usually made in parallel with the requirements specifications, use cases, and conceptual model
- Maintaining the glossary: ongoing activity throughout the project
- Useful document within which record domain or business rules, constraints, ...
- But, other artifacts may record this kind of information

151

Point-of-Sale System: Sample Glossary

Term	Category	Comments
Buy Items	use case	Description of the process of a customer buying items in a store
Item	type	An item for sale in a Store
Payment	type	A cash payment
ProductSpecification.price : Quantity	attribute	The price of an item in a sale and its associated ProductSpecification
ProductSpecification.description : Text	attribute	A short description of an item in a sale and its associated ProductSpecification
ProductSpecification.upc : UPC	attribute	The universal product code of the item and its associated ProductSpecification
SalesLineItem.quantity : Integer	attribute	The quantity of one kind of Item bought
Sale	type	A sales transaction
SalesLineItem	type	A line item for a particular item bought within a Sale
Store	type	The place where sales of items occur
Sale.total : Quantity	attribute	The grand total of a Sale
Payment.amount : Quantity	attribute	The amount of cash tendered, or presented from the customer for payment

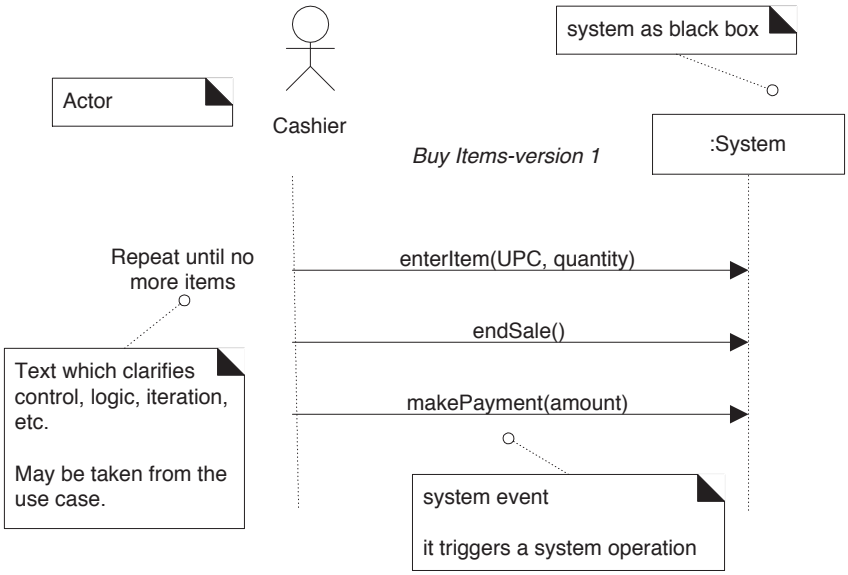
152

System Sequence Diagrams

- **System behavior**: description of what a system does, without explaining how it does it
- Investigate and define its behaviour as a "black box"
- Use cases suggest how actors interact with software system
  - ◇ actors generate events to a system, requesting some operation in response
- **System sequence diagrams**: show, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events
- **Scenario of a use case**: particular instance or realized path through its use, a real example of its enactment
- Part of the investigation into what system to build ⇒ included in analysis model
- Dependent of the prior development of the use cases
- Should be done for the typical course of events, for the most interesting alternative courses

153

Buy Items Use Case: System Sequence Diagrams



154

System Events and System Operations

- **System event**: external input event generated by an actor to a system
- **System operation**: operation that system executes in response to a system event
  - ◇ e.g., system event enterItem triggers a system operation of the same name
- Event vs operation: similar distinction as between messages and methods
- Events and operations should be expressed at the level of intent rather than in terms of physical input medium or interface widget
  - ◇ e.g., enterItem(UPC,quantity) vs enterKeyPressed(UPC,quantity)
- Usually, name start with a verb (add, enter, ...) for emphasizing command orientation of these events
- Sometimes desirable to show fragments of use case text

155

System Events and System Boundary

- Set of required system operations determined by identifying system events
- System operations can be grouped as operations of a type named System

System
endSale() enterItem() makePayment()

- Also works for multiple systems or processes in a distributed application
  - ◇ each system has unique name (System1, System2, ...) with its own operations
- System type very different of what is in the conceptual model
- To identify system events it is necessary to be clear on the choice of system boundary
  - ◇ for software development: software (and possibly hardware) system itself
  - ◇ for business re-engineering: may include manual processes

156

System Behaviour: Contracts

- Contracts describe the effect of operations upon the system
- UML allows the definition of pre- and post-conditions of operations
- Realized during analysis phase, within a development cycle
- Their creation depends on conceptual model, system sequence diagrams, and identification of system operations
- System sequence diagrams does not show the functionality associated with the system opearations invoked
- Contract: document describing what an operation commits to achieve
- Usually declarative in style
- System operation contract: describes changes in the state of the overall system when a system operation is invoked

157

Contract for enterItem

Name	enterItem(upc : number, quantity : integer)
Responsibilites:	Enter (record) sale of an item and add it to the sale. Display the item description and price.
Type:	System
Cross References:	System Functions: R1.1, R1.3, R1.9 Use cases: Buy Items
Notes:	Use superfast database access
Exceptions:	If the UPC is not valid, indicate that it was an error.
Output:	
Pre-conditions:	UPC is known to the system.
Post-conditions:	

- If a new sale, a Sale was created (instance creation).
- If a new sale, the new Sale was associated with the POST (association formed).
- A SalesLineItem was created (instance creation).
- The SalesLineItem was associated with the Sale (association formed).
- SalesLineItem.quantity was set to quantity (attribute modification).
- The SalesLineItem was associated with a ProductSpecification, based on UPC match (association formed).

158

Contract Sections

Name	Name of operation and parameters.
Responsibilites:	Informal desription of responsibilites this operation must fulfill.
Type:	Name of type (concept, software class, interface).
Cross References:	System functions reference numbers, use cases ...
Notes:	Design notes, algorithms ...
Exceptions:	Reaction to exceptional situations.
Output:	Non-UI outputs, such as messages or records sent outside of the system.
Pre-conditions:	Assumptions about the state of the system before execution of operation.
Post-conditions:	State of the system after completion of the operation.

159

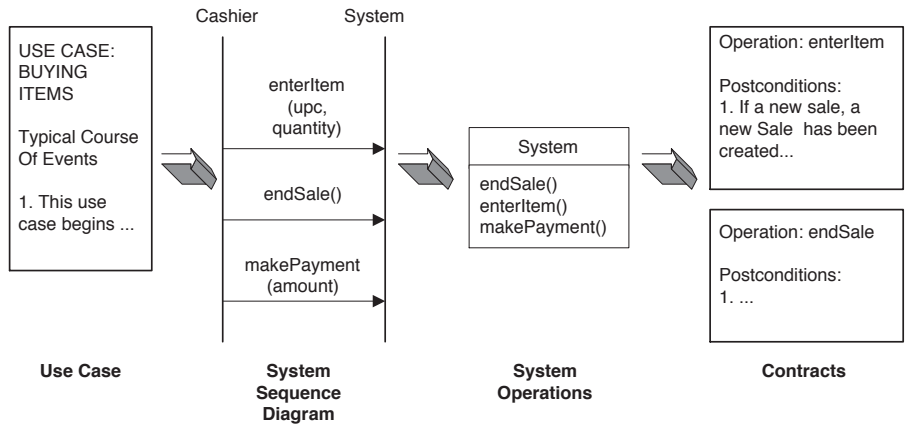
How to Create Contracts

For each use case

(1) Identify system operations from the system sequence diagrams
(2) For each system operation construct a contract
(3) Start by writing the Responsibilites section, informally describing the purpose of the operations
(4) Complete the Post-conditions section, declaratively describing state changes that occur to objects in the conceptual model
(5) Describe the post-conditions using the following categories
    - instance creation an deletion
    - attribute modification
    - associations formed and broken
(6) Describe Pre-conditions, Notes, and Exceptions sections

160

Contracts and Other Artifacts



161

Post-conditions

- Declarations about the system state that are true when the operation has finished
- They are not actions performed during the operation
- Often expressed in the past tense
  - ◇ A SalesLineItem was created vs Create a SaleLinesItem
- UML does not constrain how post-conditions should be expressed
- Important factor: to be declarative and state-change oriented rather than action-oriented
- Advantage: software design and solution deferred, allow to focus on what must happen, rather than how to be accomplished
- Expressed in the context of conceptual model
- Common during creation of contracts: discover the need of new concepts, attributes, or associations in the conceptual model

162

Pre-conditions

- Define assumptions about the state of the system at the beginning of the operation
- Many possible pre-conditions can be declared for an operation
- Some important pre-conditions
  - ◇ Things important to test at some point during execution of operation
  - ◇ Things that will not be tested, but upon which the success of the operation hinges

163

Contract for endSale

Name	endSale()
Responsibilites:	Record the end of entry of sale items and display sale total
Type:	System
Cross References:	System Functions: R1.2 Use cases: Buy Items
Notes:	
Exceptions:	If sale is not underway, indicate that it was an error
Output:	
Pre-conditions:	
Post-conditions:	

- Sales.isComplete was set to true (attribute modification).

164

Contract for makePayment

Name	makePayment(amount : number or Quantity)
Responsibilites:	Record the payment, calculate balance and print receipt.
Type:	System
Cross References:	System Functions: R2.1 Use cases: Buy Items
Notes:	
Exceptions:	If the sale is not completed, indicate an error. If the amount is less than the total sale, indicate an error.
Output:	
Pre-conditions:	
Post-conditions:	

- A Payment was created (instance creation).
- Payment.amountTendered was set to amount (attribute modification).
- The Payment was associated with the Sale (association formed).
- The Sale was associated with the Store, to add it to the historical log of completed sales (association formed).

165

Contract for startUp

Name startUp()
Responsibilites: Initialize the system
Type: System
Cross References:
Notes:
Exceptions:
Output:
Pre-conditions:
Post-conditions:

- A Store, POST, ProductCatalog and ProductSpecifications was created (instance creation).
- ProductCatalog was associated with ProductSpecifications (association formed).
- Store was associated with ProductCatalog (association formed).
- Store was associated with POST (association formed).
- POST was associated with ProductCatalog (association formed).

166

Changes to the Conceptual Model

- One datum suggested by these contracts is not yet represented in the conceptual model: completion of item entry to the sale
  - ◇ endSale specification modifies it
  - ◇ makePayment specification tests it as a precondition
- One way to represent this information: an isComplete attribute in Sale, as a Boolean value

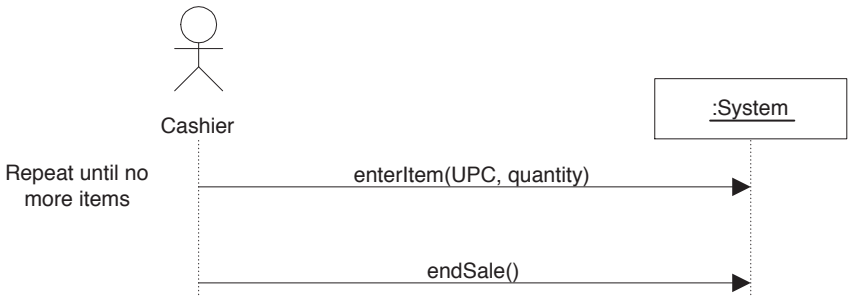
Sales
isComplete : Boolean() date time

- Other alternative to represent the changing state of the system: State pattern (see later)

167

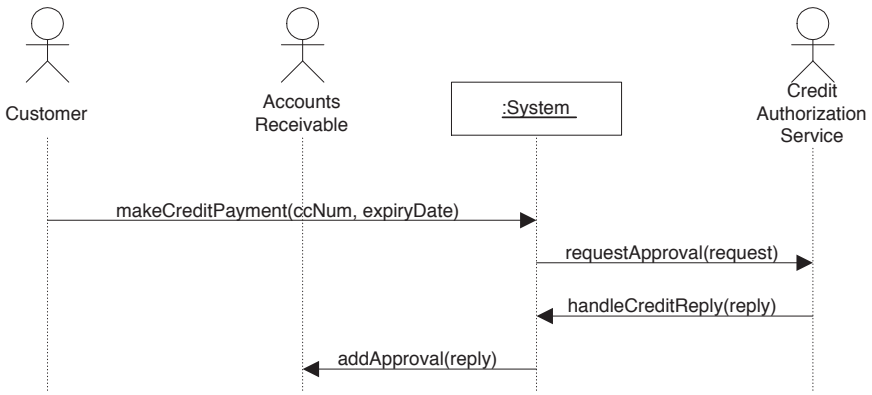
Cycle 2: System Sequence Diagrams

- Common beginning of Buying Items



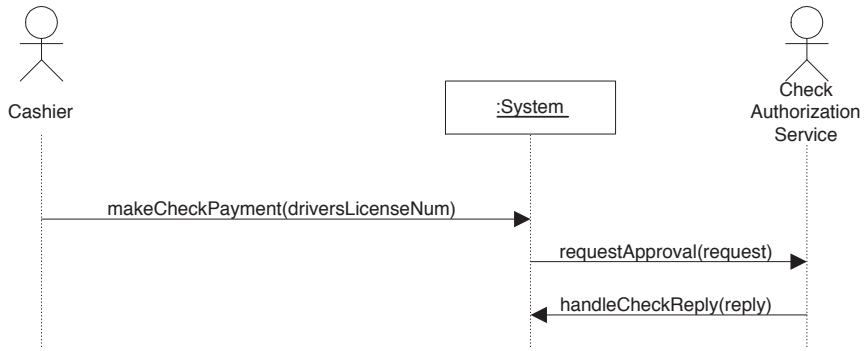
168

Point-of-Sale Application: Credit Payment SSD



169

Point-of-Sale Application: Check Payment SSD



170

Point-of-Sale Application: Changes

- New system events
  - ◇ makeCreditPayement, makeCreditReply, makeCheckPayement, makeCheckReply
- Rename MakePayment to MakeCashPayment
- A system operation is defined for each system event

System
endSale() enterItem() handleCheckReply() handleCreditReply() makeCashPayment() makeCheckPayment() makeCreditPayment()

- Contracts need to be written for new system operations

171

Point-of-Sale Application: Contract for makeCreditPayment

Name	makeCreditPayment(ccNumber : number, expiryDate : date)
Responsibilites:	Create and request authorization for a credit payment.
Type:	System (type)
Cross References:	
Notes:	The request has to be transformed into a flat record.
Output:	A credit payment request was sent to a credit authorization service.
Pre-conditions:	The current sale is complete.
Post-conditions:	

- A CreditPayment was created.
- pmt was associated with the current Sale.
- A CreditCard cc was created; cc.number = ccNum, cc.expiryDate = expiryDate.
- cc was associated with pmt.
- A CreditPaymentRequest cpr was set created.
- pmt was associated with cpr.
- cpr was associated with the CreditAuthorizationService.

172

Point-of-Sale Application: Contract for handleCreditReply

Name	handleCreditReply(reply : CreditPaymentReply)
Responsibilites:	Respond to authorization reply from the credit authorization service. If approved complete the sale, and record the payment in accounts receivable.
Type:	System (type)
Cross References:	
Notes:	reply is actually a record that needs to be transformed into a CreditPaymentApprovalReply or CreditPaymentDenialReply .
Output:	If approved, credit payment reply was sent to accounts receivable.
Pre-conditions:	The credit payment request was sent to a credit authorization service.
Post-conditions:	

- If reply represented approval:
  - ◇ A CreditPaymentApprovalReply was created.
  - ◇ approval was associated with AccountsReceivable.
  - ◇ Sale was associated with Store, to add it to the historical log of completed sales.
- Else if reply represented denial:
  - ◇ A CreditPaymentDenialReply denial was created.

173

From Analysis to Design

- **Analysis**: understanding of requirements, concepts, operations of a system
- Focuses on the what

Analysis Artifact	Questions answered
Use cases	What are the domain processes?
Conceptual model	What are the concepts, terms?
System sequence diagrams	What are the system events and operations?
Contracts	What do the system operations do?

- **Design**: development of a logical solution
- Artifacts
  - ◇ **interaction diagrams**: illustrate how objects communicate to fulfill requirements
  - ◇ **design class diagrams**: summarize the definition of classes (and interfaces) to be implemented in software
- Interaction diagrams: requires assigning **responsibilities** and the use of **design patterns**

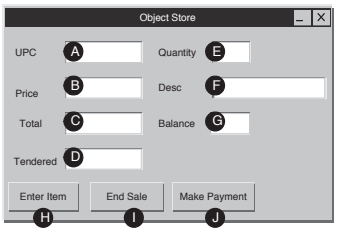
174

Real Use Cases

- Describes a design of the use case in terms of concrete input/output technology and its overall implementation
- E.g., for a GUI, include diagrams of windows involved, discussing low-level interaction with interface widgets
- Their definition is one of the first design phase activities within a development cycle
- Their creation is dependent upon the creation of associated essential use cases
- They may not be necessary to create
- Alternative: designer may create rough user interface storyboards, deferring the details to implementation
- Useful if developers or client require detailed interface descriptions prior to implementation

175

Real Use Cases: Buy Items, Version 1



Use case:	Buy Items with Cash
Actors:	Customer (initiator), Cashier
Purpose:	Capture a sale and its cash payment
Overview:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collect a cash payment. On completion the Customer leaves with the items
Type:	Primary and essential
Cross references :	Functions R1.1, R1.2, R1.3, R1.7, R1.9, R2.1

176

Real Use Cases: Buy Items, Version 1

Typical Course of Event

Actor Action	System Response
1. This use case begins when a Customer arrives at a POST checkout with items to purchase	
2. For each item the Cashier types in the Universal Product Code (UPC) in A of Window-1. If there is more than one of the same item, the quantity may optionally be entered in E. They press H after each item entry.	3. Determines the item price and adds the item information to the running sales transaction. The description and price of the current item are displayed in B and F of Window-1.
4. On completion of item entry, the Cashier indicates to the POST that the item entry is complete by pressing widget I.	5. Calculates and presents the sale total in C.
6. . . .	

177

Interaction Diagrams

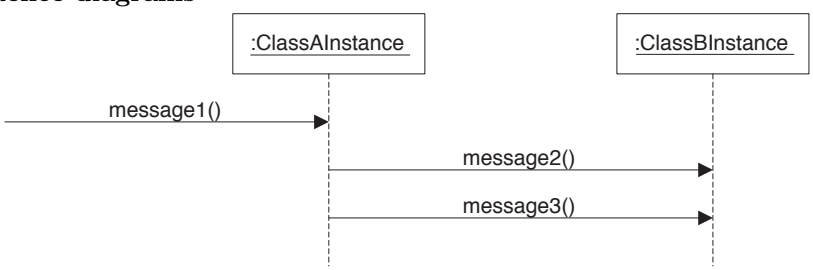
- Illustrates message interactions between instances and classes in class model
- Starting point: fulfillment of post-conditions of the operation contracts
- Realized within the design phase of a development cycle
- Their creation dependent upon the creation of
  - ◇ **conceptual model**: for defining software classes corresponding to concepts. Objects of these classes participate in interaction diagrams
  - ◇ **system operation contracts**: for identifying the responsibilities and post-conditions that the interaction diagrams must fulfill
  - ◇ **real (or essential) use cases**: for obtaining information about what tasks the interaction diagrams fulfill

178

UML Interaction Diagrams

- Two kinds of interaction diagrams
- Either can be used to express similar or identical message interactions
- **Collaboration diagrams**

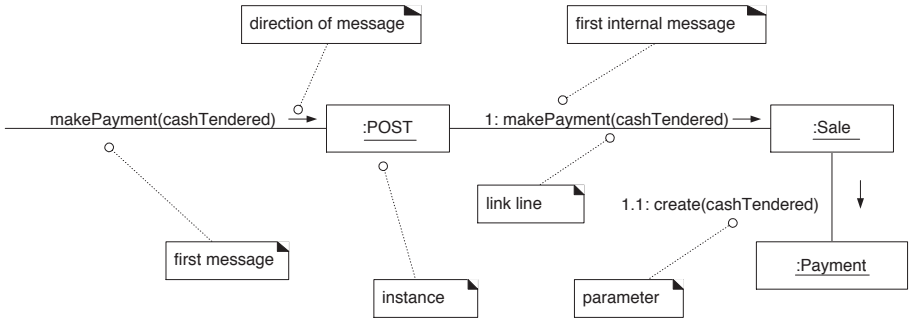
- **Sequence diagrams**



- Collaboration diagrams are more expressive, convey more contextual information, relative spatial economy

179

Collaboration Diagram: makePayment



- Message makePayment is sent to an instance of a POST
  - ◇ It corresponds to the makePayment system operation message
- The POST object sends the makePayment message to a Sale instance
- The Sale object creates an instance of a Payment

180

Importance of Interaction Diagrams

- One of the most important artifacts created in object analysis and design
- The amount of time and effort spent on their generation should absorb a significant percentage of the overall project effort
- Codified patterns, principles, and idioms can be applied to improve the quality of their design
- Relatively, creation of use case, conceptual models, and other artifacts is easier than the assignment of responsibilities and creation of well-designed interaction diagrams

181

How to Make Collaboration Diagrams

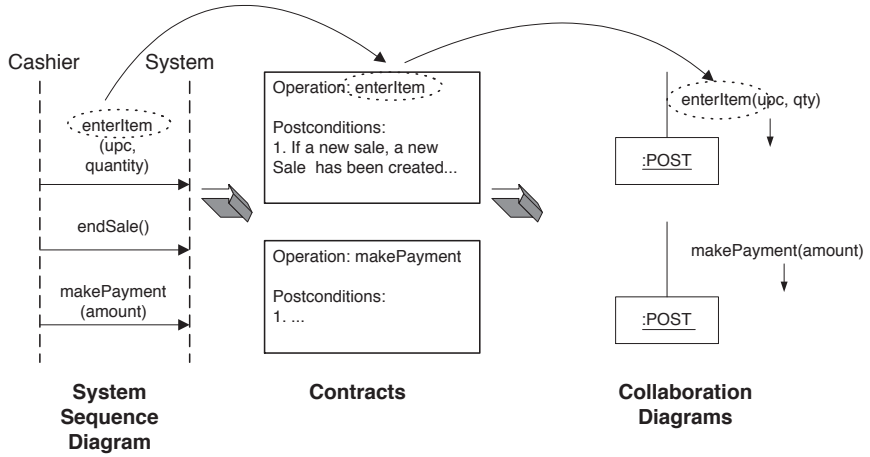
(1) Create a separate diagram for each system operation under development in current development cycle
   - For each system operation message, make a diagram with it as the starting message

(2) Complex diagrams (e.g., does not fit in one page) can be split in smaller diagrams

(3) Design a system of interacting objects to fulfill the tasks
   - Using operation contract responsibilities and post-conditions, use case description
   - Apply patterns to develop a good design

182

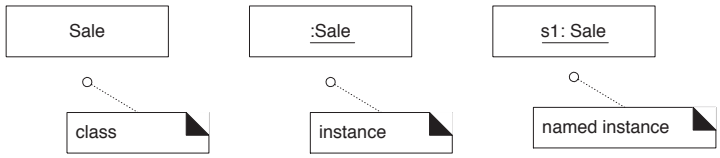
Collaboration Diagrams and Other Artifacts



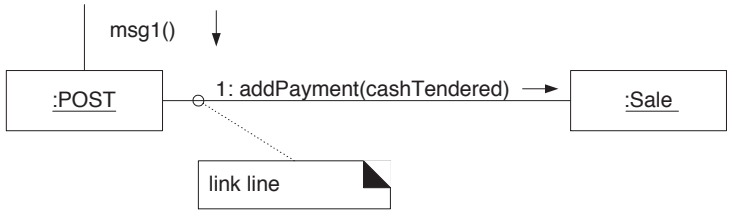
183

Classes, Instances, Links

- UML notation for classes and instances

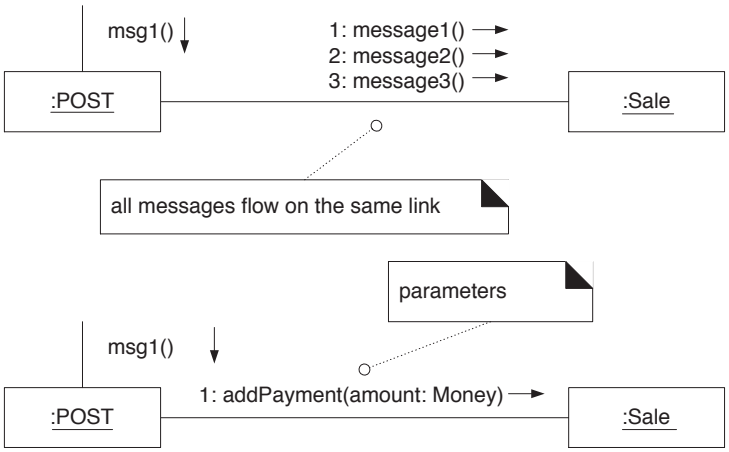


- Link: instance of an association, connection path between two instances
- Indicates some form of navigation and visibility
- Client/server view: messages may be sent from the client to the server



184

Messages, Parameters



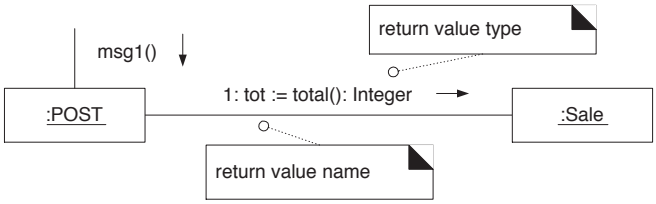
185

Return value, Messages to Self

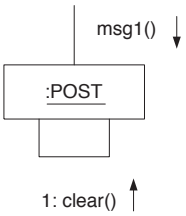
- UML syntax for messages

  return := message(parameter : parameterType): returnType

- Return value



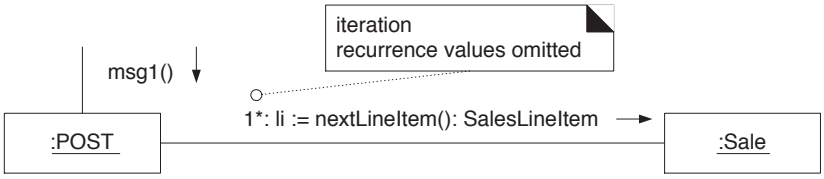
- Messages to self



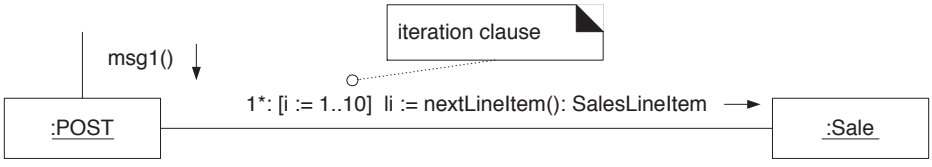
186

Iterations

- Indicated by following the sequence number with a '$*$'



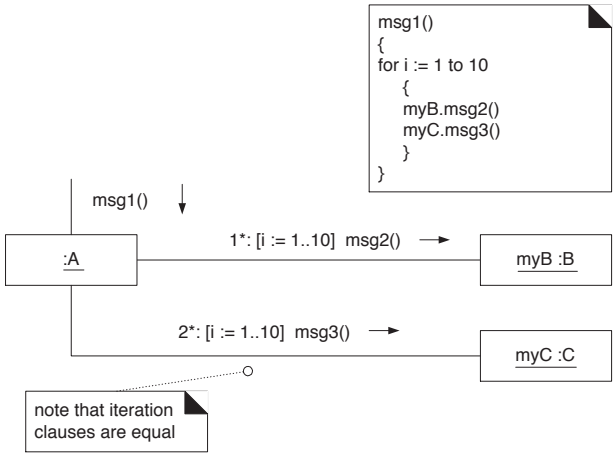
- Iteration clause indicates the recurrence values



187

Iterations, cont.

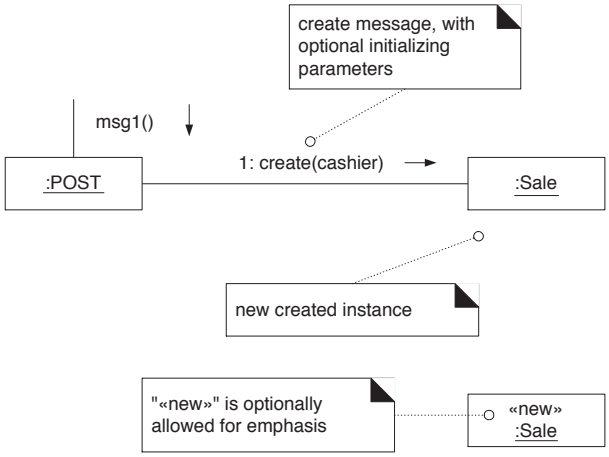
- To express more than one message happening within the same iteration clause: repeat iteration clause on each message



188

Creation of Instances

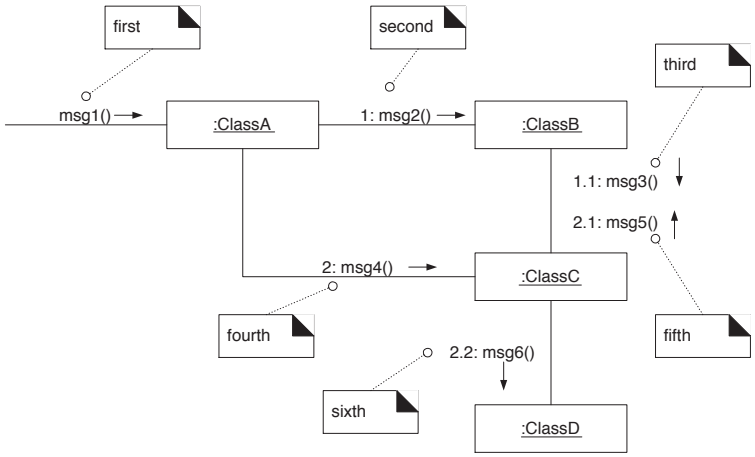
- Language-independent creation message: `create`



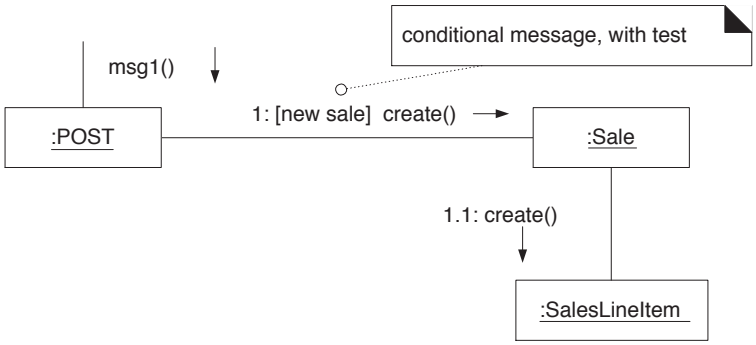
189

Message Number Sequencing

- Order of messages determined by **sequence numbers**
- First message is not numbered
- Nesting denoted by prepending incoming message number to the outgoing message number

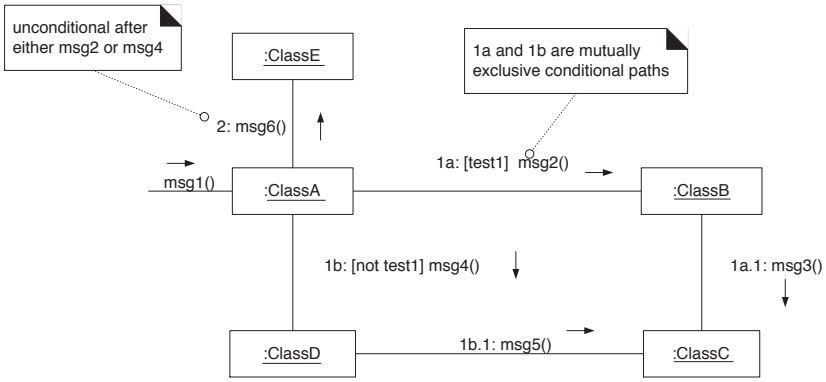


Conditional Message



- Shown by following sequence number with a conditional clause in square brackets
- Message only sent if clause evaluates to true

Mutually Exclusive Conditional Paths

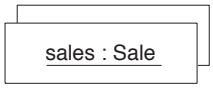


- Sequence expression have a conditional path letter
- First letter is a by convention
- Either 1a or 1b could execute after msg1()
- Subsequent nested messages still prepended with outer message sequence, e.g., 1b.1

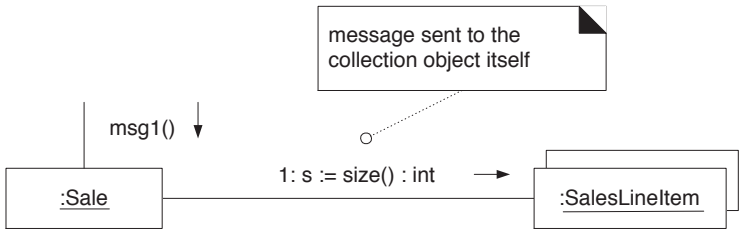
192

Messages to Collections

- UML notation for multiobjects



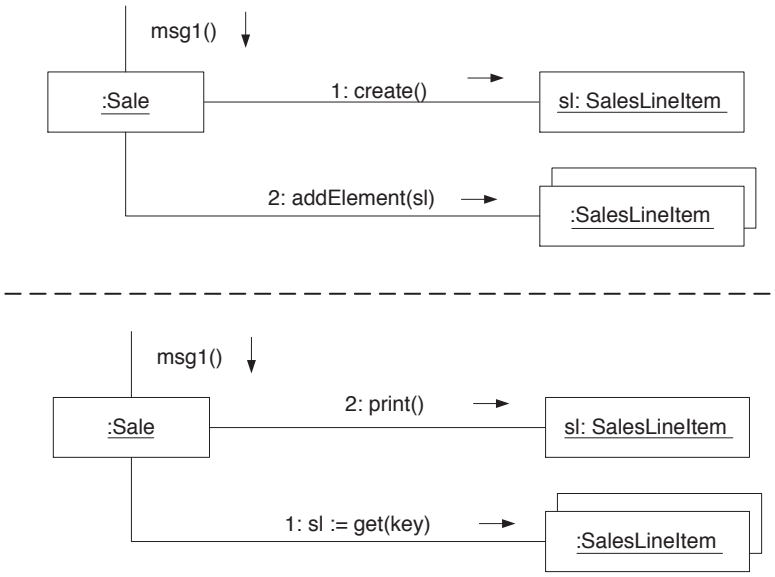
- Usually implemented as a group of instances stored in a **container** or **collection** object, but not necessarily so



- Messages to a multiobject are sent to the collection object itself
- Messages to a multiobject are not broadcast to each element

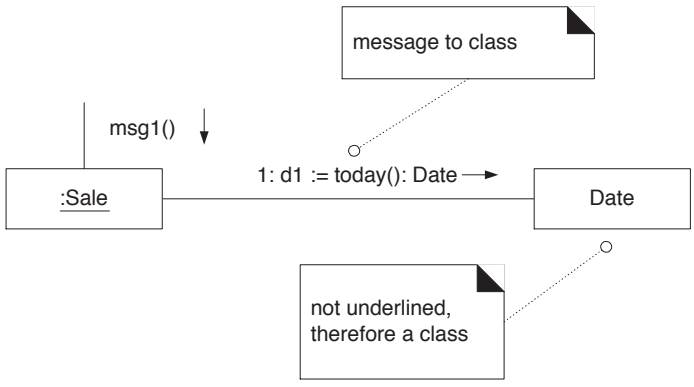
193

Messages to a Multiobject and to an Element



194

Messages to a Class Object



- Messages may be sent to a class to invoke class methods
- E.g., in Java = static methods, in Smalltalk = class methods

195

Patterns for Assigning Responsibilities

- Object systems: composed of objects sending message to other objects to complete operations
- Preliminary identification of postconditions and responsabilities in contracts
- Interaction diagrams show the solution satisfying these postconditions and responsibilities
- There is great variability in responsibility assignment
- Poor choices lead to systems and components which are fragile, hard to maintain, understand, reuse or extend
- Skillful implementation is founded in good principles of object design
- Some of these principles, applied during creation of interaction diagrams and/or responsibility assignement codified in some patterns

196

Responsibilities

- **Responsibility**: contract or obligation of a type or a class
- Related to the obligation of an object in terms of its behaviour
- Responsibilites are of two types
- **Doing** responsibilities of an object include
  - ◇ doing something itself
  - ◇ 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

197

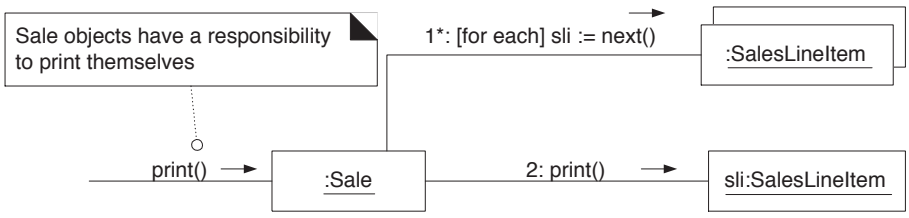
Responsibilities and Methods

- Assigned to objects during object design, e.g.,
  - ◇ a Sale is responsible for printing itself (a doing)
  - ◇ a Sale is responsible for knowing its date (a knowing)
- Knowing responsibilities inferable from the conceptual model
- Translation of responsibilities into classes and methods influenced by granularity
  - ◇ "provide access to relational DB" may involve many classes and methods
  - ◇ "print a sale" may involve one or a few methods
- Responsibilities implemented using methods which either act alone or collaborate with other methods and objects

198

Responsibilities Assignment

- Assignment of responsibilities usually occurs during creation of interaction diagrams
- Decisions in responsibility assignment reflected in what messages are sent to different classes of objects



199

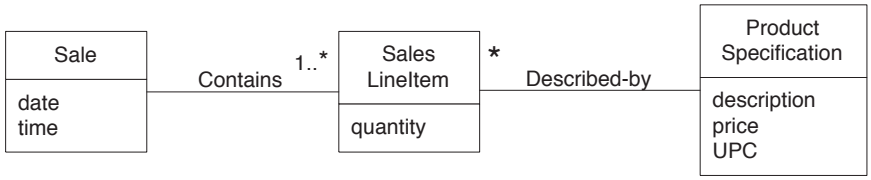
Patterns

- Experienced developers build up a repertoire of general principles and idiomatic solutions that guide them in the creation of software
- May be codified in a structured format describing the solution and given a name
- **Pattern**: named description of a problem and solution that can be applied in new contexts, with advice on how to apply it in novel situations
- Notion originated with architectural patterns of Christopher Alexander
- Their application to software originated in the 1980s
- Skillful assignment of responsibilities is extremely important in object design
- Some patterns describing principles of assigning responsibilities to objects
  - ◇ Expert, Creator, High Cohesion, Low Coupling, Controller

200

Expert Pattern

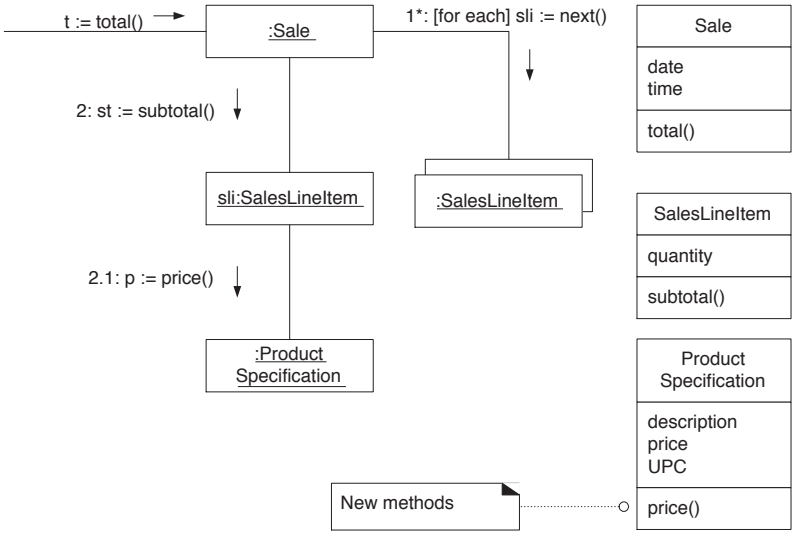
- **Problem**: Most basic principle for assigning responsibilities to objects
- **Solution**: Assign a responsibility to the information expert, the class that has the information needed to fulfill it.
- **Example**: in POST application some class need to know grand total of a sale



- Determined by all SalesLineItem instances of a sale, the sum of their subtotals
  - ◇ only Sale knows this ⇒ correct class for this responsibility
- Line item subtotal determined by SalesLineItem.quantity and ProductSpecification.price ⇒ SalesLineItem is correct class for this responsibility
- SalesLineItem need to know product price ⇒ ProductSpecification is assigned this responsibility

201

Expert Pattern: Calculating the Sale Total



Expert Pattern: Discussion

- More used than any pattern in assignment of responsibilities
- Intuition: objects do things related to the information they have
- Fulfillment of a responsability requires information spread across different classes
- Real-world analogy: responsibility usually given to individuals who have the information necessary to fulfill the task
- Maintains encapsulation: objects use their own information to fulfill tasks
- Supports **low coupling** $\Rightarrow$ more robust and maintainable systems
- Behaviour distributed across classes that have required information
  $\Rightarrow$ encourages "lightweight" classes easier to understand and maintain
- **High cohesion** is supported

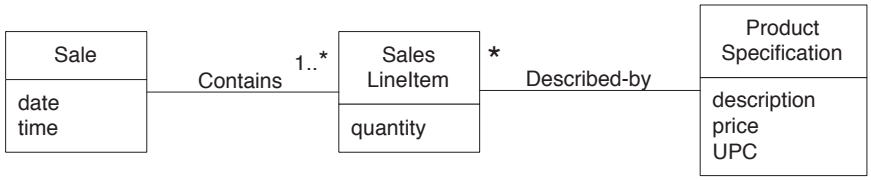
Creator Pattern

- **Problem**: Responsibility for creating a new instance of some class
- **Solution**: Assign class B the responsibility to create an instance of class A 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 that will be passed to A when it is created (i.e., B is an expert with respect to creating A)

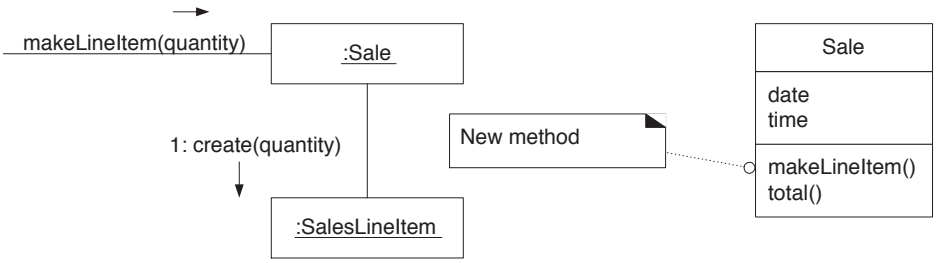
204

Creator Pattern: Example

- Who should be responsible for creating a SalesLineItem instance ?



- Sale aggregates many SalesLineItem objects ⇒ Sale should create these instances



205

Creator Pattern: Discussion

- Creation of objects: very common task in object systems
- Objective: find a creator needing to be connected to the new object in any event
- Aggregate *aggregates* Part, Container *contains* Content, Recorder *records* Record are very common relationships
- Creator suggests the enclosing container or recorder is a good candidate for creating the thing contained or recorded
- Sometimes, creator found by looking class having initializing data passed in during creation
- E.g., if Payment instance needs to be initialized with Sale total ⇒ Sale is candidate to be the creator
- **Low coupling** supported ⇒ lower maintenance dependencies, higher reuse
- Related patterns: Low Coupling, Whole-Part

206

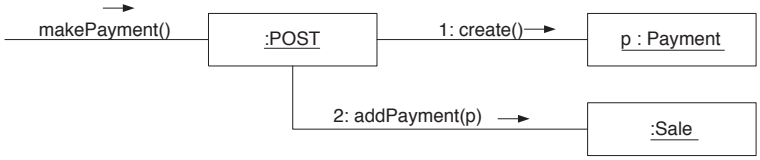
Low Coupling Pattern

- **Problem**: How to support low dependency and increased reuse
- **Solution**: Assign a responsibility so that coupling remains low
- **Coupling**: measure of how strongly one class is conected, has knowledge of, or relies upon other classes
- A class with low coupling is not dependent on too many other classes
- Problems with high coupling
  - ◇ changes in related classes force local changes
  - ◇ harder to understand in isolation
  - ◇ harder to reuse (because requires additional presence of classes it is dependent upon)

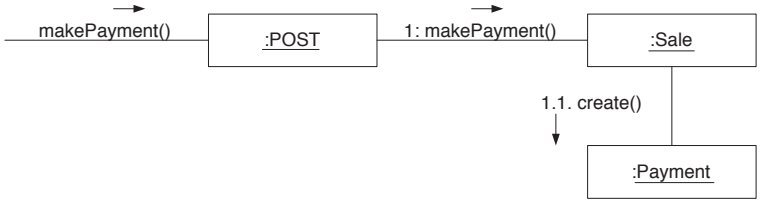
207

Low Coupling Pattern: Example

- Responsibility of create Payment instance and associate it to Sale
- Post records a Payment ⇒ Creator pattern suggests Sale as a candidate
  - ◇ couples POST class to knowledge of Payment class



- Alternative solution with lower coupling: Sale eventually coupled to Payment



- Example of two patterns suggesting different solutions

Low Coupling Pattern: Discussion

- Principle to keep in mind during all design decisions
- Common forms of coupling from TypeX to TypeY in object languages
  - ◇ TypeX has an attribute that refers to a TypeY instance, or TypeY itself
  - ◇ TypeX has a method referencing an instance of TypeY, or TypeY itself: parameter, local variable, object returned from a message
  - ◇ TypeX is a direct or indirect subclass of TypeY
  - ◇ TypeY is an interface implemented by TypeX
- Supports the design of more independent classes, reduces the impact of changes, improves reusability and higher productivity
- Cannot be considered in isolation from other patterns
- May not be that important if reuse is not a goal: cost-benefit consideration
- A subclass is strongly coupled to its superclass
- No absolute measure of when coupling is too high
- Low coupling taken to excess yields a poor design

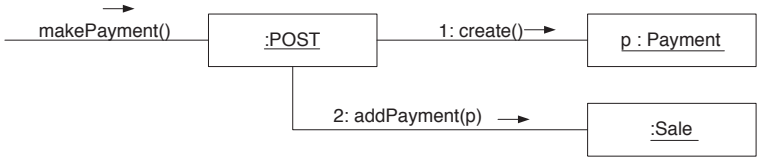
High Cohesion Pattern

- **Problem**: How to keep complexity manageable ?
- **Solution**: Assign a responsibility so that cohesion remains high
- **Cohesion**: measure of how strongly related and focused are responsibilities of a class
- Class with high cohesion: highly related responsibilites, does not do tremendous amount of work
- Problems with low cohesion
  - ◇ hard to comprehend
  - ◇ hard to reuse
  - ◇ hard to maintain
  - ◇ delicate, constantly affected by change

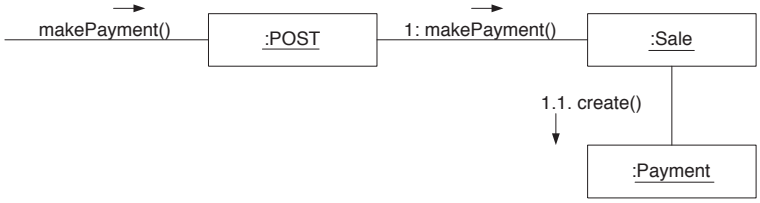
210

High Cohesion Pattern: Example

- POST takes the responsibility of makePayment system operation
  - ◇ If it is responsible for many other operations, it will become incohesive



- Delegates payment creation responsibility to Sale ⇒ higher cohesion in POST



- This design supports both high cohesion and low coupling ⇒ desirable

211

High Cohesion Pattern: Discussion

- Like Low Coupling, principle to keep in mind during all design decisions
- High cohesion: elements of a component all work together to provide some well-bounded behaviour
- Class with high cohesion: relatively small number of methods, with highly-related functionality, does not do too much work
- Analogy: a person with too many unrelated responsibilites is not effective
- Benefits
  - ◇ clarity and ease of comprehension of design is increased
  - ◇ maintenance and enhancements are simplified
  - ◇ low coupling is supported
  - ◇ fine grain of highly related functionality supports increased reuse potential since class can be used for a very specific purpose

212

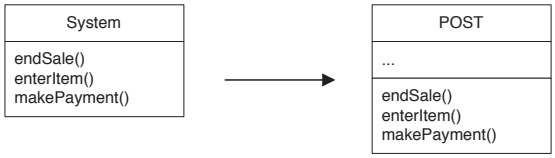
Controller Pattern

- **Problem**: Who should be responsible for handling a system event ?
- **Solution**: Assing the responsibility to a class
  - ◇ representing the overall system: façade controller
  - ◇ representing the overall business or organization: façade controller
  - ◇ representing something that is active in the real-world (e.g., role of person) that might be involved in the task: role controller
  - ◇ represents an artificial handler of all system events of a use case: use-case controller
- **Controller**: non-user interface object responsible for handling a system

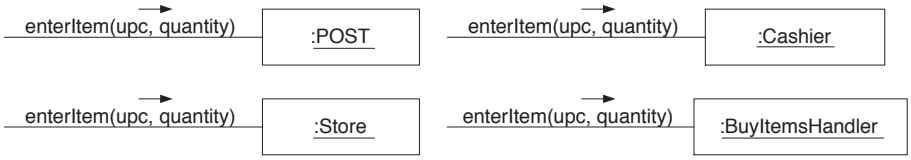
213

Controller Pattern: Example

- During design, system operations are assigned to one or more controller classes



- Controller for enterItem ?
  - ◇ POST: represents the overall system
  - ◇ Store: represents the overal business or organization
  - ◇ Cashier: represents the role of a person involved in the task
  - ◇ BuyItemsHandler: artificial handler of operations of a use case



- Choice influenced by other factors, e.g., cohesion, coupling

214

Controller Pattern: Discussion

- Controllers should not have too much responsibility, should delegate work to other objects while coordinating the activity
- **Façade controllers**
  - ◇ suitable when there are only a few system events, or
  - ◇ it is not possible to redirect system event messages to alternating controllers
- **Use case controllers**
  - ◇ used when using other approaches leads to low cohesion or high coupling, or there are many system events across different processes
  - ◇ same controller class should be used for all system events of a use case
  - ◇ allows to maintain state of the use case: useful, e.g., for identifying out-of-sequence events
  - ◇ different controllers may be used for different use cases

215

Controller Pattern: Discussion, cont.

- **Human-role controllers**
  - ◇ danger that mimic what role does in the real world
  - ◇ may create an incohesive role controller that does not delegate
  - ◇ should be used sparingly
- **Benefits of using controllers**
  - ◇ increased potential for reusable components: business or domain processes handled by domain objects rather than interface layer
  - ◇ reason about the state of the use case: ensures that system operations occur in a legal sequence

216

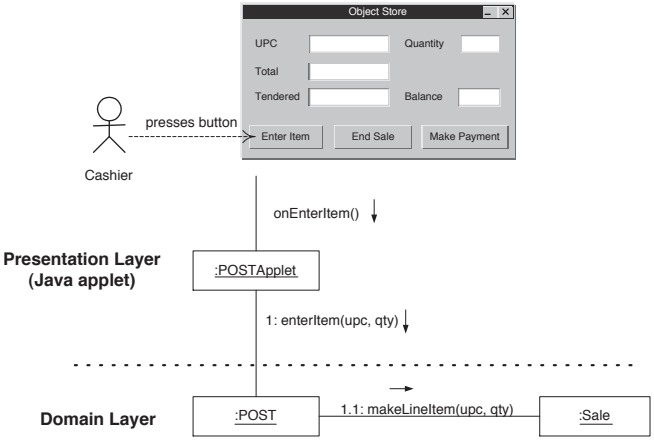
Bloated controllers

- Unfocused, handle too many areas of responsibilities ⇒ low cohesion
- Identifying signs
  - ◇ if there is a single controller receiving all system events
    - ∗ happens if role or façade controller is chosen
  - ◇ controller performs many tasks without delegating the work
    - ∗ usually involves a violation of Expert and High Cohesion patterns
  - ◇ controller has many attributes and maintains significant information about the system
    - ∗ information should be distributed

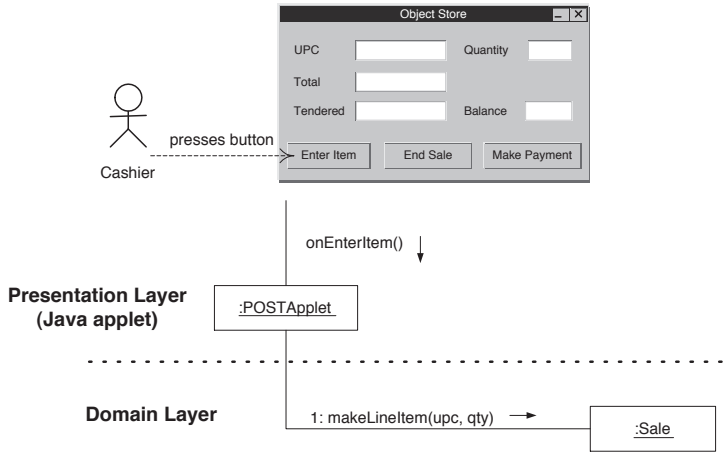
217

Presentation vs Domain Layer

- External interface objects (widgets, applets) and presentation layer should not have responsibility of system events
- System operations should be handled in domain layer



Presentation vs Domain Layer, cont.



- Undesirable coupling of presentation to domain layer
- Business logic embedded in presentation layer

Point-of-Sale Application: Collaboration Diagrams

- In current iteration we consider two use cases and their associated system events
  - ◇ Buy Items
    - ∗ enterItem
    - ∗ endSale
    - ∗ makePayment
  - ◇ StartUp
    - ∗ startUp

220

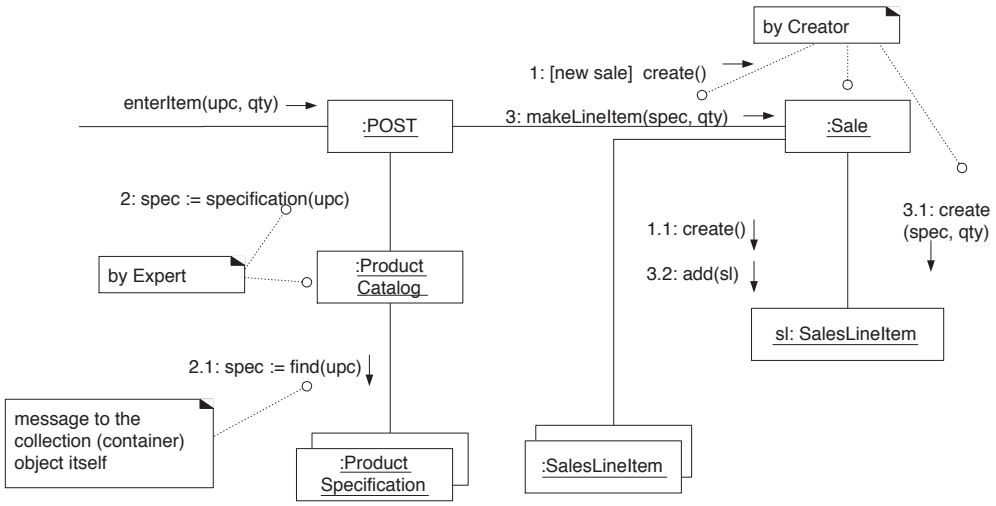
Contract for enterItem

Name	enterItem(upc : number, quantity : integer)
Responsibilites:	Enter (record) sale of an item and add it to the sale. Display the item description and price.
Type:	System
Cross References:	System Functions: R1.1, R1.3, R1.9 Use cases: Buy Items
Notes:	Use superfast database access
Exceptions:	If the UPC is not valid, indicate that it was an error.
Output:	
Pre-conditions:	UPC is known to the system.
Post-conditions:	

- If a new sale, a Sale was created (instance creation).
- If a new sale, the new Sale was associated with the POST (association formed).
- A SalesLineItem was created (instance creation).
- The SalesLineItem was associated with the Sale (association formed).
- SalesLineItem.quantity was set to quantity (attribute modification).
- The SalesLineItem was associated with a ProductSpecification, based on UPC match (association formed).

221

Collaboration Diagram: enterItem

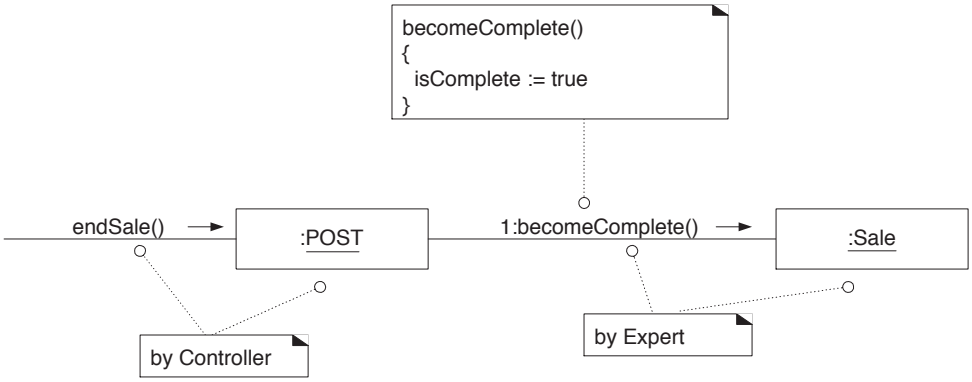


Contract for endSale

Name endSale()
Responsibilites: Record the end of entry of sale items and display sale total.
Type: System
Cross References: System Functions: R1.2
Use cases: Buy Items
Notes:
Exceptions: If sale is not underway, indicate that it was an error.
Output:
Pre-conditions:
Post-conditions:

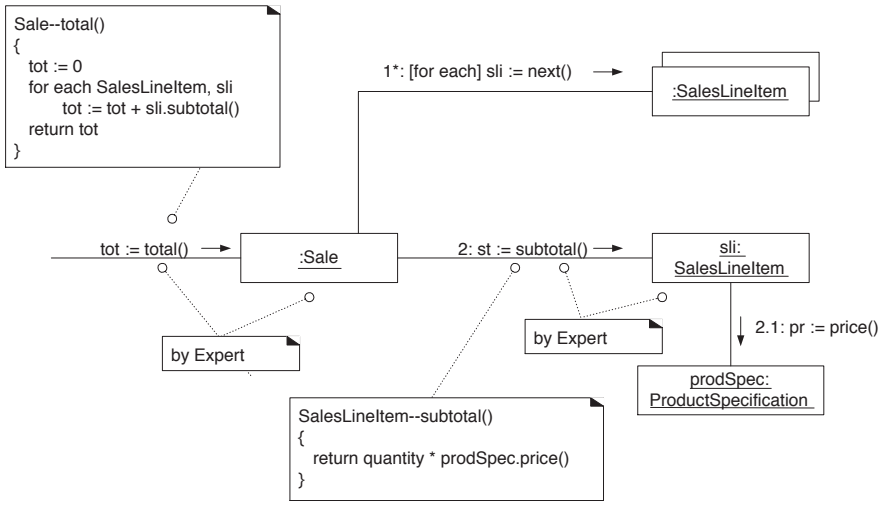
- Sales.isComplete was set to true (attribute modification).

Collaboration Diagram: endSale



224

Collaboration Diagram: saleTotal



225

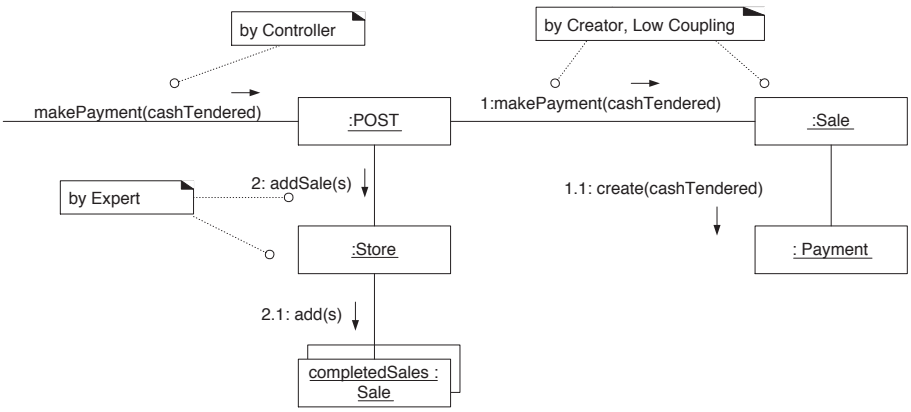
Contract for makePayment

Name	makePayment(amount : number or Quantity)
Responsibilites:	Record the payment, calculate balance and print receipt.
Type:	System
Cross References:	System Functions: R2.1 Use cases: Buy Items
Notes:	
Exceptions:	If the sale is not completed, indicate an error. If the amount is less than the total sale, indicate an error.
Output:	
Pre-conditions:	
Post-conditions:	

- A Payment was created (instance creation).
- Payment.amountTendered was set to amount (attribute modification).
- The Payment was associated with the Sale (association formed).
- The Sale was associated with the Store, to add it to the historical log of completed sales (association formed).

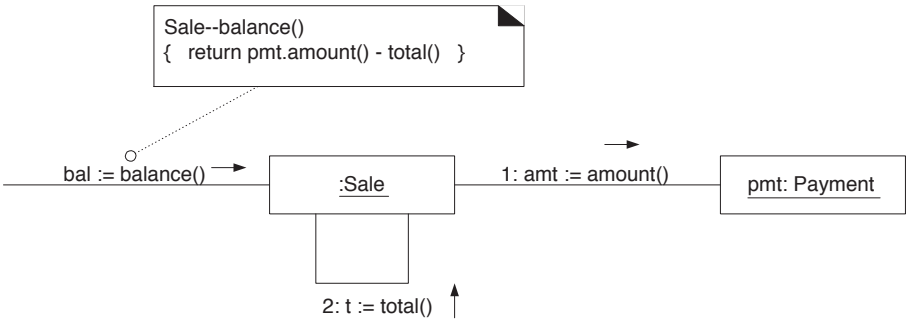
226

Collaboration Diagram: makePayment



227

Collaboration Diagram: saleBalance

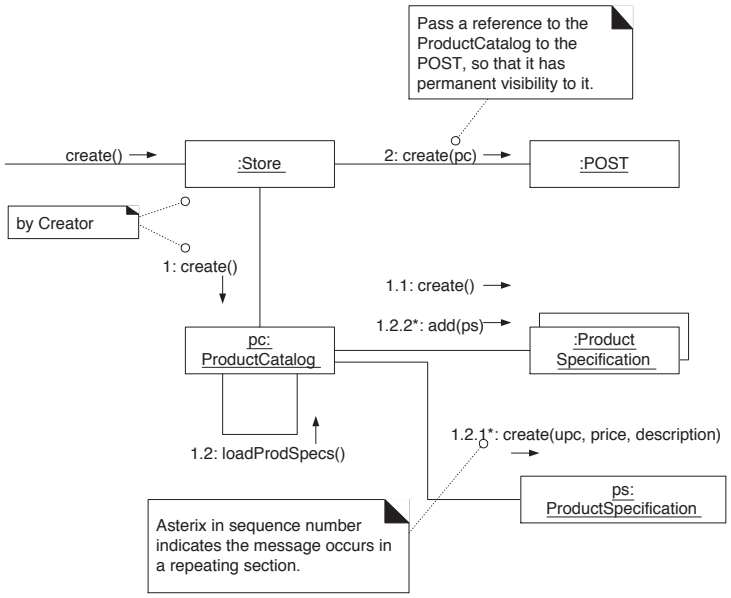


Contract for startUp

Name startUp()
Responsibilites: Initialize the system.
Type: System
Cross References:
Notes:
Exceptions:
Output:
Pre-conditions:
Post-conditions:

- A Store, POST, ProductCatalog and ProductSpecifications was created (instance creation).
- ProductCatalog was associated with ProductSpecifications (association formed).
- Store was associated with ProductCatalog (association formed).
- Store was associated with POST (association formed).
- POST was associated with ProductCatalog (association formed).

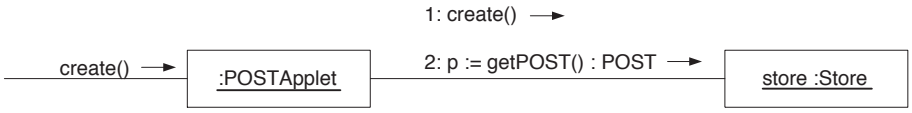
Collaboration Diagram: create



230

Connecting Presentation Layer to Domain Layer

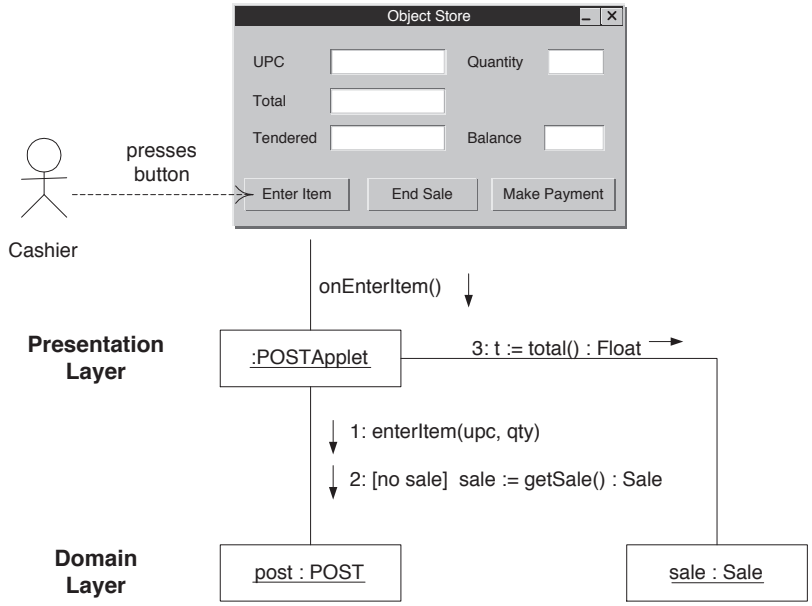
- With a GUI, it will be responsible for initiating creation of initial domain object



- Once applet has a connection to POST instance it can forward system event message to it
- For enterItem, window needs to show running total after each entry
- After the POSTApplet forwards enterItem to POST object
  (1) it gets a reference to the Sale (if it did not already have one)
  (2) it stores the Sale reference in the attribute
  (3) it sends a total message to Sale to get the information needed to display the runing total on the window

231

Connecting Presentation Layer to Domain Layer



232

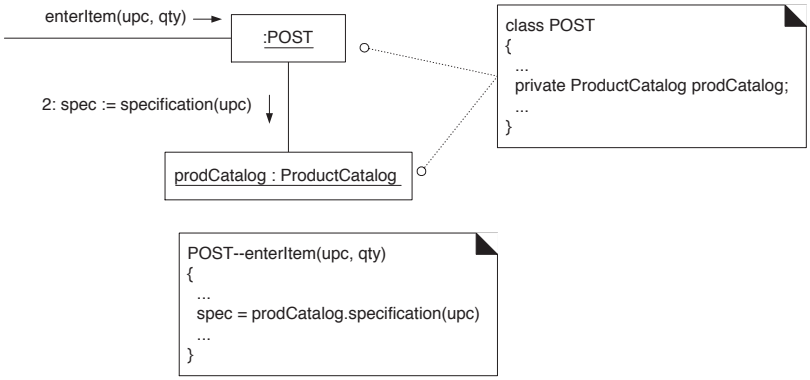
Determining Visibility

- Collaboration diagrams illustrate messages between objects
- For this, sender object must be visible to the receiver object
- **Visibility**: ability of one object to see or have reference to another
- Related to the issue of scope: is one ressource within the scope of another
- Four common ways to achieve visibility from object A to object B
  (1) Attribute visibility
  (2) Parameter visibility
  (3) Locally-declared visibility
  (4) Global visibility

233

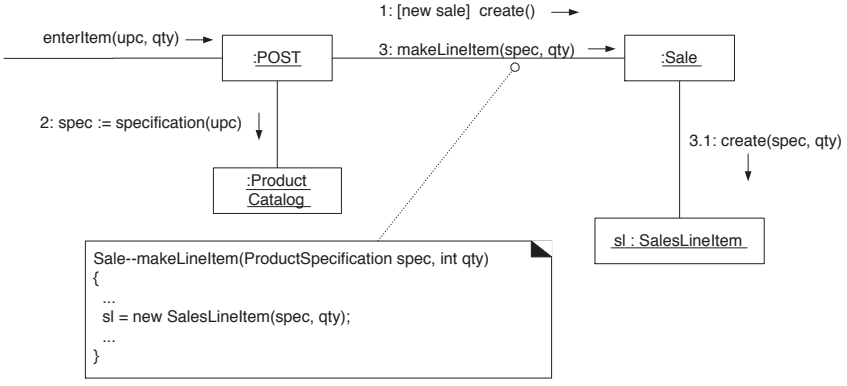
Attribute Visibility

- Attribute visibility from A to B: B is an attribute of A
- Relatively permanent visibility: persists as long as A and B exists
- Very common form of visibility in object systems



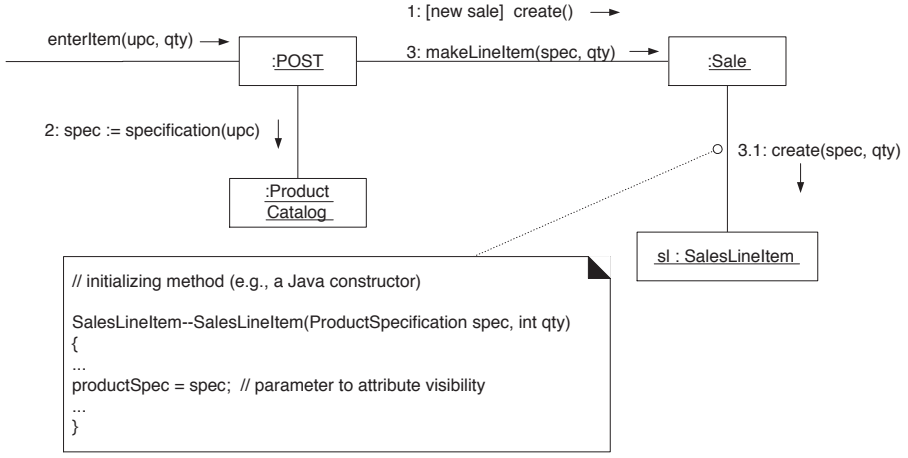
Parameter Visibility

- Parameter visibility from A to B: B is passed as a paremeter to a method of A
- Relatively temporary visibility: persists only within the scope of the method



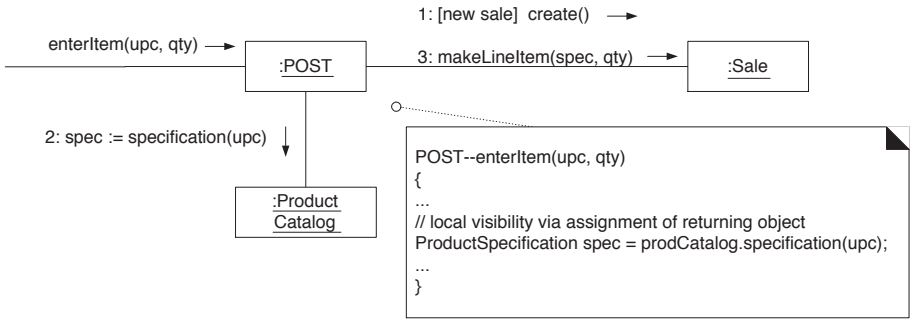
Parameter to Attribute Visibility

- It is common to transform parameter visibility into attribute visibility



Locally Declared Visibility

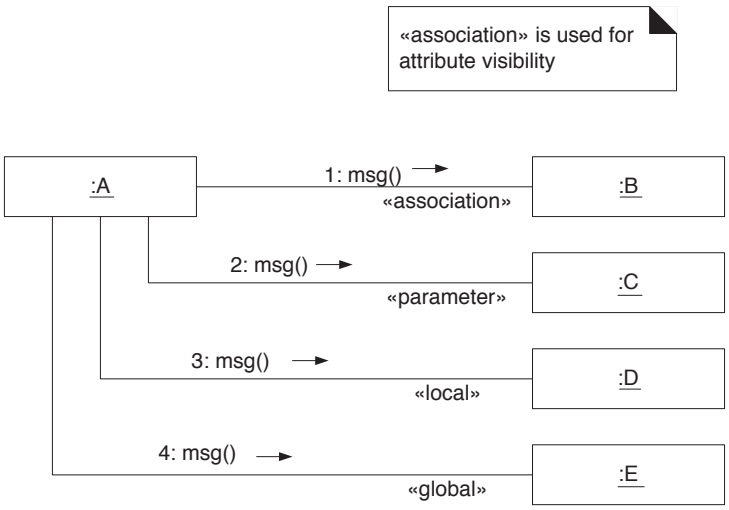
- Locally declared visibility from A to B: B is declared as a local object within a method of A
- Relatively temporary visibility: persists only within the scope of the method
- Common means by which this form of visibility is achieved
  - ◇ create a new local instance and assign it to a local variable
  - ◇ assign the return object from a method invocation to a local variable



Global Visibility

- Global visibility from A to B: B is global to A
- Relatively permanent visibility: persists as long as A and B exists
- Least common form of visibility in object systems
- Most obvious, but least desirable, way to achive it: assign an instance to a global variable
- Preferred method to achieve global visibility is to use the Singleton pattern

UML Stereotypes for Visibility



Design Class Diagrams

- Illustrates the specifications for software classes and interfaces in an application
- Show definitions for software entities rather than real-world concepts
- Typical information it includes
  - ◇ classes, associations, and attributes
  - ◇ interfaces with their operations and constants
  - ◇ methods
  - ◇ attribute type information
  - ◇ navigability
  - ◇ dependencies
- UML does not specifically define an element called "dessign class diagrams" but uses the more generic term "class diagrams"
- Their creation depends upon the prior creation of
  - ◇ interaction diagrams
  - ◇ conceptual model

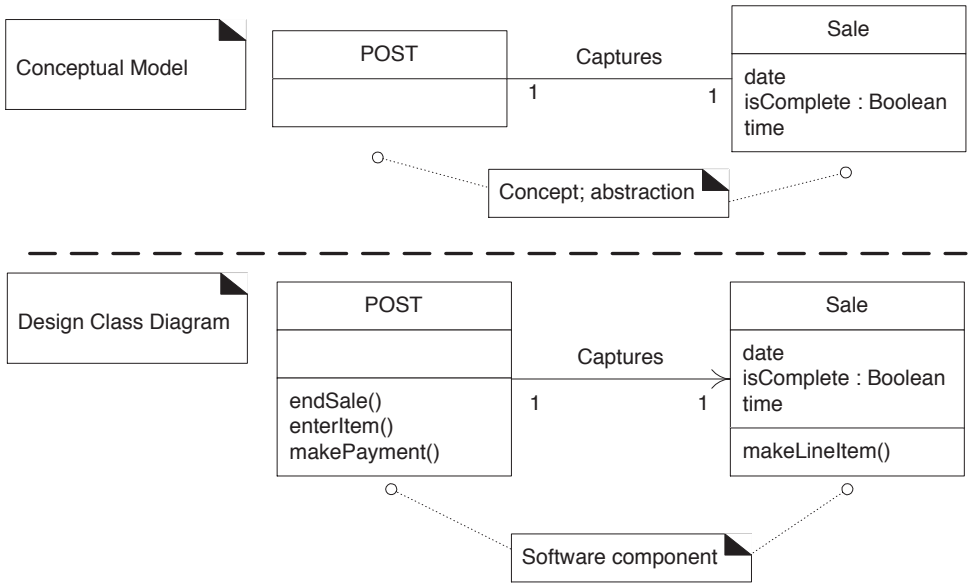
240

Making a Design Class Diagram

(1) Identify all classes participating in the software solution by analyzing interaction diagrams
(2) Draw them in a class diagram
(3) Duplicate the attributes from the associated concepts in the conceptual model
(4) Add method names by analyzing interaction diagrams
(5) Add type information to the attributes and methods
(6) Add the associations necessary to support the required attribute visibility
(7) Add navigability arrows to the associations to indicate the direction of attribute visibility
(8) Add dependency relationship lines to indicate non-attribute visibility

241

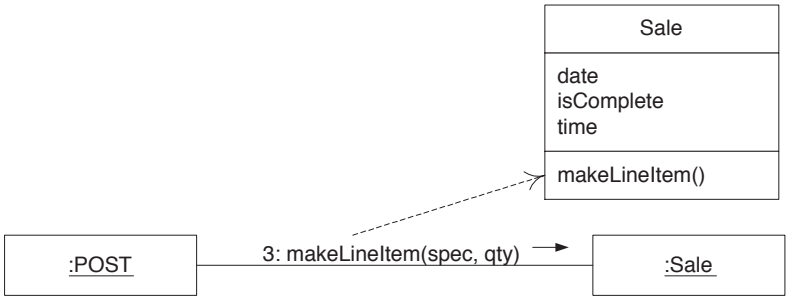
Conceptual Model vs Design Class Diagrams



242

Point-of-Sale Application: Design Class Diagrams

- Classes participating in the software solution found in interaction diagrams: POST, ProductCatalog, Store, Payment, Sale, ProductSpecification, SalesLineItem
  - ◇ attributes come from conceptual model
  - ◇ method names come from collaboration diagrams



- Many concepts in conceptual model (Cashier, Manager, . . .) not present in design
  - ◇ no need for the current development cycle to represent them in software
  - ◇ will be tackled in later cycles

243

Methods: Issues

Create message

- In UML, language independent form of instantiation and intialization
- Must be translated in terms of specific idioms of an object programing language
- E.g., in Java: invocation of new operator followed by a constructor call
- Because of its multiple interpretations it is common to omit creation-related methods and constructors from design class diagrams

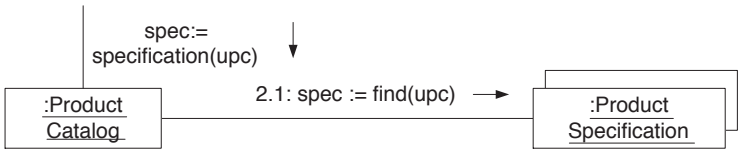
Accessing methods

- Those which retrieve (accessor) or set (mutator) attributes
- Common idiom to have an accessor and mutator for each attribute and declare all attributes private (encapsulation)
- Those methods are usually excluded from class diagrams

244

Methods: Issues, cont.

Multiobjects



- Message to a multiobject interpreted as a message to container/collection object
- find is not part of ProductSpecification but of container object

Language-Dependent Syntax

- Recommended to use basic UML format, even if implementation language has different syntax
- Translation sould take place during code generation

245

Methods: Issues, cont.

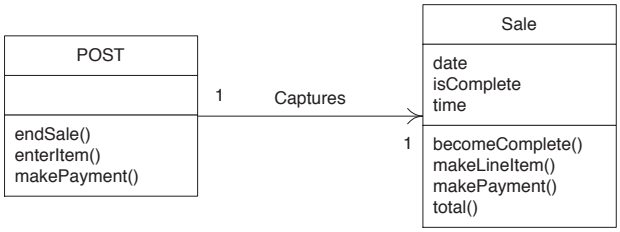
Type Information

- Design class diagrams should be created by considering the audience
- If created in CASE tool with automatic code generation: full details necessary
- If to be read by software developers, exhaustive detail may be inadequate

246

Adding Associations and Navigability

- In design class diagrams, roles may be decorated with navigability arrow
- **Navigability**: property of the role indicating that it is possible to navigate uni-directionally from objects of the source to the target class
- Implies visibility, usually attribute visibility

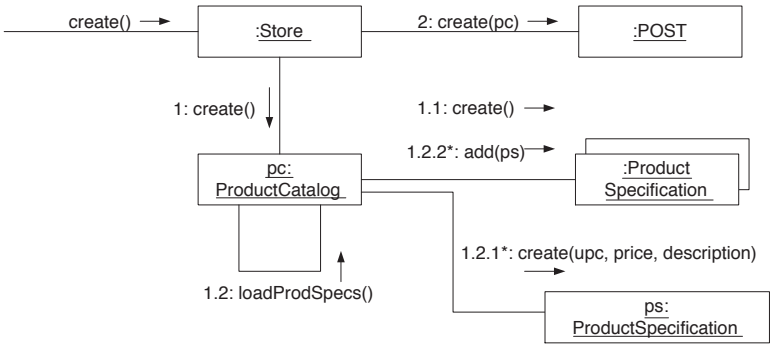


  - ◇ POST class will probably have an attribute pointing to a Sale object
  - ◇ navigability arrow: POST objects connected unidirectionally to Sale objects
  - ◇ no connection from Sale to POST
- Most associations in design class diagrams should be adorned with necessary navigability arrows

247

Adding Associations and Navigability, cont.

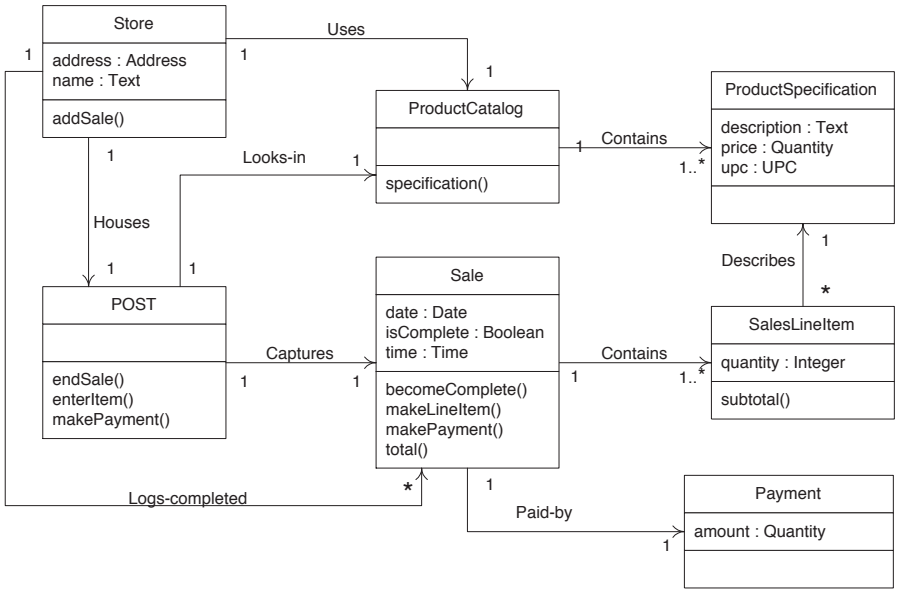
- Required visibility and associations indicated by interaction diagrams



- Common situations to add a navigability adornment from A to B
  - ◇ A send a message to B
  - ◇ A creates an instance B
  - ◇ A needs to maintain a connection to B

248

Point-of-Sale Application: Navigability Adornments



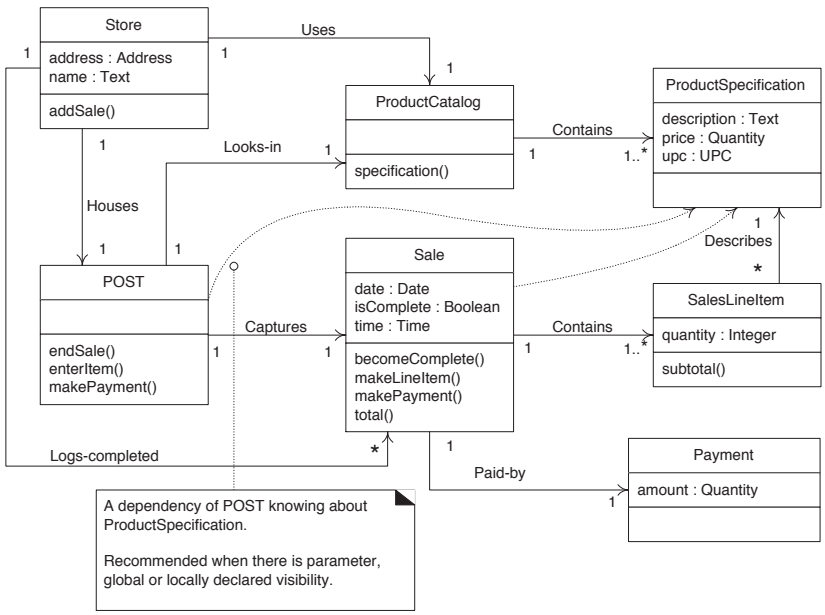
249

Adding Dependency Relationships

- UML has a general dependency relationship indicating that one element (of any kind) has knowledge of another element
- Indicated by a dashed arrowed line
- In class diagrams it is useful to depict non-attribute visibility between classes, i.e., parameter, global, or locally declared visibility

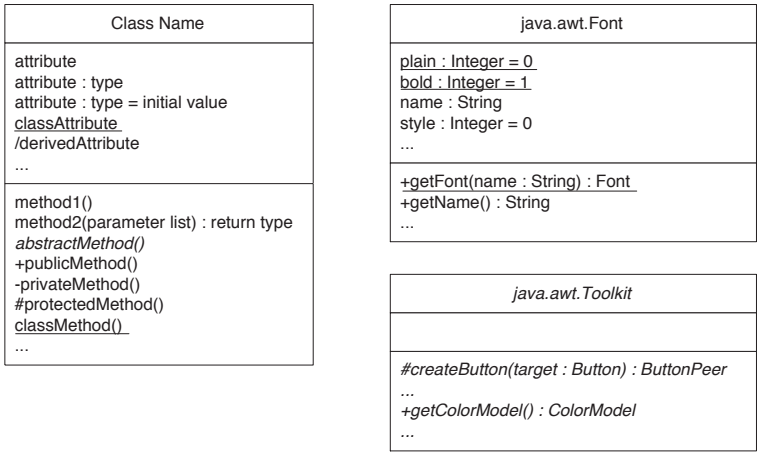
250

Dependency Relationships Indicating non-attribute Visibility



251

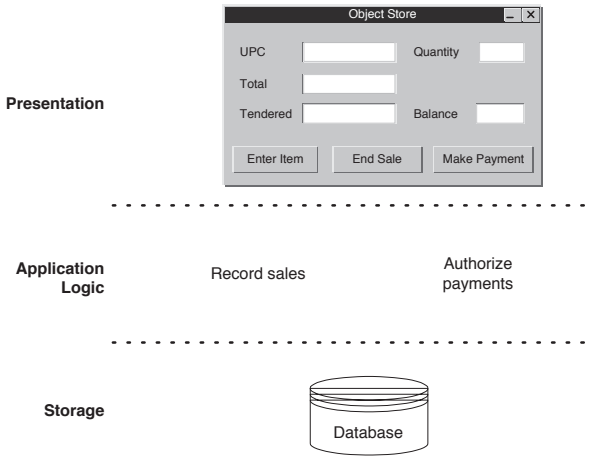
UML Notation for Member Details



- Describe features of class and interface members: visibility, initial values, ...
- Attributes are assumed to be private by default

252

Three-Tier Architecture



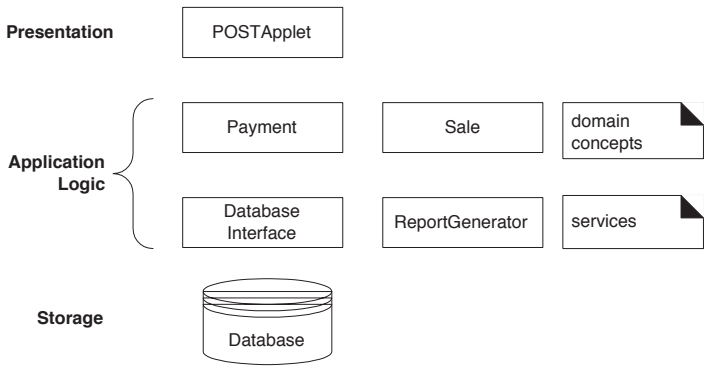
- A system is composed of multiple subsystems, domain objects are but one
- Typical information system has to connect to a UI and a persistent storage

253

Three-Tier Architecture, cont.

- Common architecture for information systems
  - ◇ **Presentation**: windows, reports, ...
  - ◇ **Application logic**: tasks and rules that govern the process
  - ◇ **Storage**: persistent storage mechanism
- Advantage: separation of application logic into a distinct tier
- Presentation tier is relatively free of application processing
- Two-tier design: application logic place within windows definition, which read and write directly to a database
- Disadvantage: inhibits software reuse

Decomposing the Application-Logic Tier



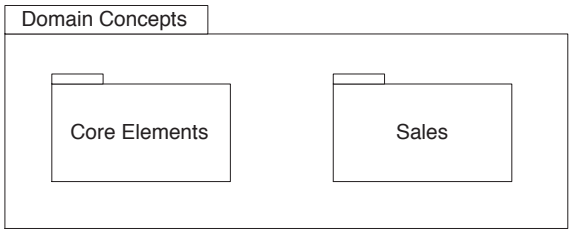
- Application logic layer decomposed in
  - ◇ **domain objects**: classes representing domain concepts
  - ◇ **services**: service objects for functions such as DB interaction, reporting, communications, security, ...
- This is called **multi-tiered architectures**

Multi-Tiered Architectures

- A **logical** 3-tier architecture may be **physically** deployed in various configurations
  - ◇ Presentation and application logic tiers on client computer, storage on server
  - ◇ Presentation on client computer, application logic on application server, storage on separate data server
- Motivations for multi-tiered architectures
  - ◇ Isolation of application logic into separate components
    - ∗ can be reused in other systems
  - ◇ Distribution of tiers on different physical computing nodes and/or processes
    - ∗ can improve performance and increase coordination and shared information in a client/server system
  - ◇ Allocation of developers to construct specific tiers
    - ∗ supports specialized expertise in terms of development skills, allows parallel development

256

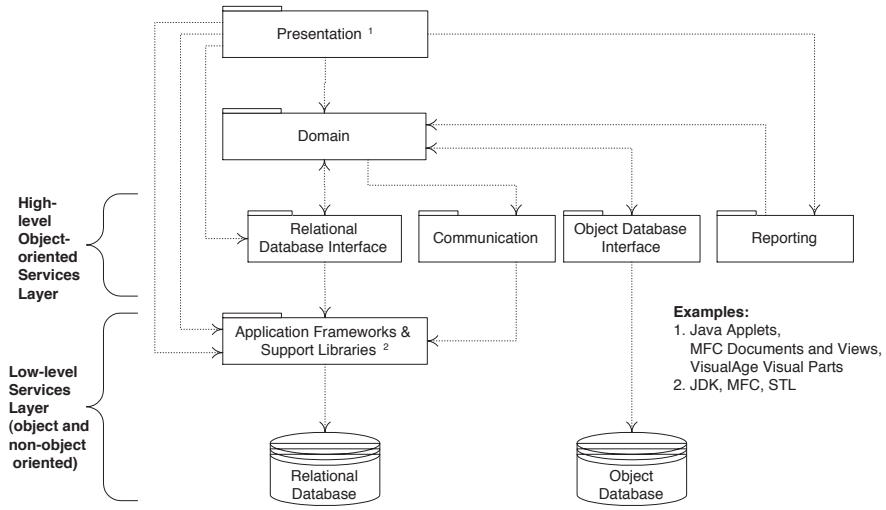
UML Package Notation



- Package mechanism: groups of elements or subsystems
- Set of model elements of any kind
  - ◇ e.g., classes, use cases, collaboration diagrams, or other (nested) packages
- Entire system may be considered within a top level package System
- Defines a nested name space, elements with the same name may be duplicated within different packages

257

Detailed Architecture Example



258

Package comments

- Relational and object DB interface packages provide mechanisms for communicating with databases. An ODB interface provided by the ODB vendor. A RDB interface must be custom developped or buy a third-party product
- High-level object service packages: reporting, DB interfaces, security, inter-process communications. Usually written by application developpers. Low-level services provide basic functions such as window and file manipulation, usually provided as standard language libraries or purchased from a third-party vendor
- Application Frameworks and Support Libraries typically include support for creating windows, defining application coordinators, accessing DB and files, inter-process communication, ...

Dependency comments

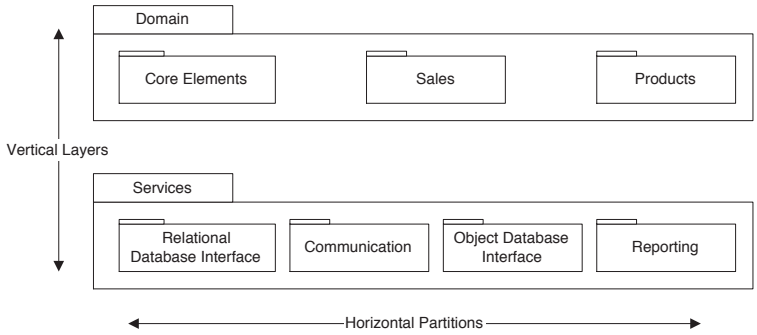
- Dependency relationships (dashed arrow line) indicates if a package has knowledge of (coupling) to another package. Lack of dependency from package A to B implies the components of package A have no references to any classes, components, interfaces, methods or services of package B.
- Domain package has no dependency (coupling) to the presentation layers: principle of Model-View Separation (see after)

Identifying Packages

- Group elements that provide a common service (or family of related services) with relatively high coupling and collaboration into a package
- At some level of abstraction the package will be viewed as highly cohesive: it has strongly related responsibilities
- In constrast, the coupling and collaboration between elements of different packages will be relatively low

259

Layers and Partitions



- Layers of an architecture represent the vertical tiers
- Partitions represent a horizontal division of relatively parallel subsystems of a layer

260

Layers and Partitions

- Layers are not coupled in the limited sense of network protocols (e.g., OSI 7-Layer model)
- Protocol models: elements of layer n only access services of immediate lower layer $n-1$
- Object systems: "relaxed layered" or "transparent layered" architecture
  - ◇ elements of a layer communicate with several other layers

261

Visibility Between Package Classes

- Access into the domain packages
  - ◇ other packages (e.g., presentation) have visibility into many of the classes representing domain concepts
- Access into the service packages
  - ◇ other packages (e.g., presentation, domain) have visibility into only one or a very few classes in each service package (usually a Façade class, see after)
- Access into the presentation packages
  - ◇ no other packages have direct visibility to the presentation layer
  - ◇ communication is indirect, if at all (using the Observer pattern, see after)

262

Service Packages Interface: The Façade Pattern

- **Façade class**: provides a common interface to a group of other components or a disparate set of interfaces
  - ◇ Disparate elements: classes in a package, set of functions, framework, subsystem (local or remote)
- Often used to provide a public interface to a service package
- Example: RelationalDatabaseInterface package with many internal classes
- One class (e.g., DBFacade) provides public interface into services of the package
- Classes in other packages send messages only to an instance of DBFacade, have no coupling to other classes in the package
- DBFacade collaborates wih other private classes of the package to provide services
- This design supports low coupling

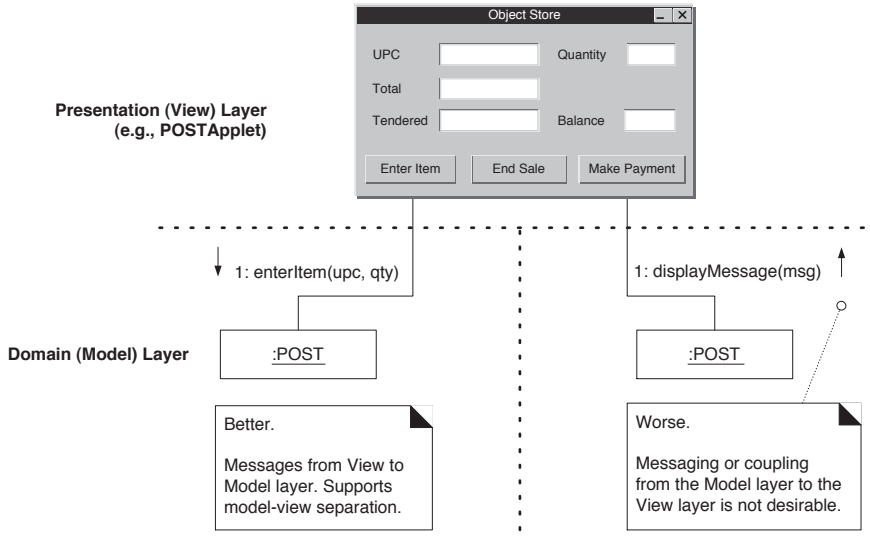
263

Model-View Separation Pattern

- Also known as **Domain-Presentation** pattern
  - ◇ formerly **Model-View-Controller**, but controller portion is an anachronism
- **Model**: synonym for the domain layer of objects
- **View**: synonym for presentation objects such as windows, applets, reports
- Model (domain) objects should not have direct knowledge of or be directly coupled to view (presentation) objects
- Model components may be reused in new applications or attached to a new interface
- **Corollary**: domain classes encapsulate information and behaviour related to application logic
- View classes are relatively thin, responsible for input/output, do not maintain data or directly provide application functionality

264

Conformance vs Violation of Model-View Separation



265

Motivation for Model-View Separation

- Support cohesive model definitions focusing on domain processes rather than on computer interfaces
- Separate development of the model and user interface layers
- Minimize impact of requirement changes in the interface upon the domain layer
- New views to be easily connected to an existing domain layer, without affecting the domain layer
- Multiple simultaneous views on the same model object
- Execution of model layer independent of the user interface layer, e.g., in a message-processing or batch-mode system
- Easy porting of the model layer to another user interface framework

266

Model-View Separation and Indirect Communication

- How can windows obtain information to display ?
- Usually it is sufficient for them to send message to domain objets, querying for information which they then display in widgets
- Called **polling** or **pull-from-above** model of display updates
- Sometimes insufficient: domain objects need to (indirectly) communicate with windows to cause a real-time update as domain objects changes, e.g.,
  - ◇ monitoring applications: network management
  - ◇ simulation applications requiring visualizations: aerodynamics modeling
- A **push-from-below** model of display update is required
- Model-view separation ⇒ indirect communication from other objects to windows

267

Indirect Communication in a System

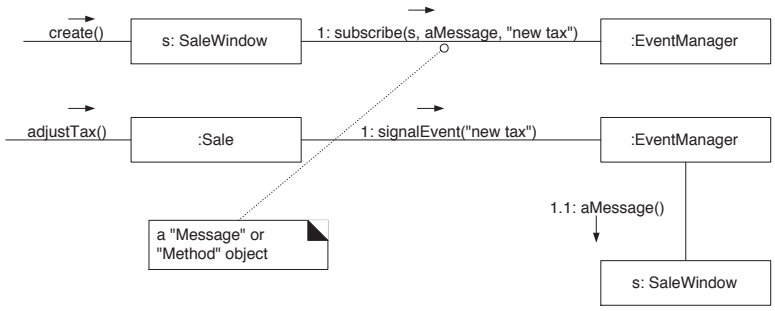
- Model-view separation: one of many examples where indirect communication between elements is needed
- A message between a sender and receiver objects requires that the sender has direct visibility to the receiver
- Other mechanisms besides direct messaging are required to de-couple the sender and receiver, or for supporting brodcast or multicast
- Some variants of indirect communication
  - ◇ Publish-subscribe pattern
  - ◇ Callbacks
  - ◇ Event notification systems

268

Publish-Subscribe Pattern

- Also known as Observer
- An event occurs within a Publisher, other objects (Subscribers) are interested in this event. Publisher should not have direct knowledge or its Subscribers.
- Solution: event modification system where Publisher indirectly notify Subscribers
- E.g., EventManager class maintains mappings between events and subscribers
- An event is published by publisher sending a signalEvent message to the EventManager
- When published, EventManager notifies all subscribers interested in the event
- Event is represented by a simple string, may also be an instance of Event class
- Typically, a single instance of EventManager class, globally accessed using the Singleton pattern (see later)
- This is a simple language-independent illustration, language-specific facilities should be used when available, e.g., in Java

Publish-Subscribe Pattern, cont.



- Classic design: Publisher object maintain a direct visibility to its collection of Subscribers interested in notification of Publisher changes
- Disadvantage: impacting the coupling and implementation of Publishers
- Alternative: use of indirect EventManager class managing subscribers minimizing publisher coupling and responsibilities
- But, may suffer from performance problems: bottleneck for all events

Event Notification Systems

- Publish-Subscriber architecture: general-purpose mechanism for event notification and indirect communication in a system
- New approach to the design of object systems
- Distinguishing qualities
  - ◇ Direct coupling between senders and receivers is not required
  - ◇ A single event can be broadcast to any number of subscribers
  - ◇ Reaction to an event can be generalized in Callback objects
  - ◇ Relatively easy to provide concurrency by executing each Callback on its own thread
- A system design relying on asynchronous event notification and broadcasting to subscribers ⇒ state-machine design (see later)

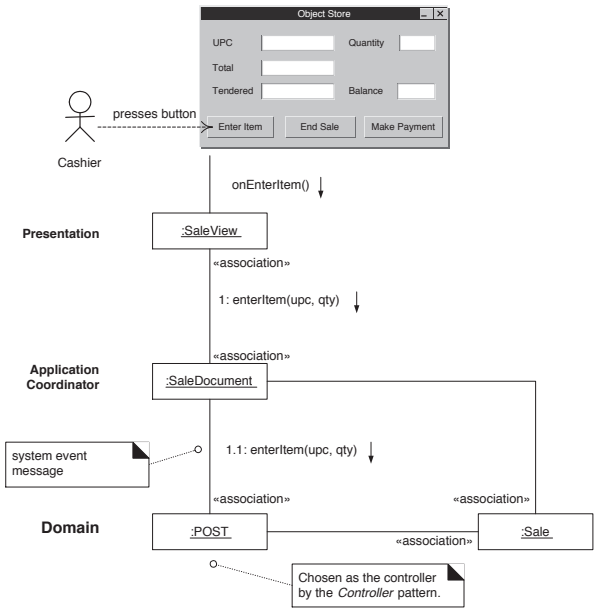
271

Application Coordinators

- Class responsible for mediating between interface and domain layer
- Basic responsibilities
  - ◇ Map information between domain objects and interface
  - ◇ Respond to events from interface
  - ◇ Open windows that display information from domain objects
  - ◇ Manage transactions, e.g., performing commit and rollback
- For some applications, coordinator has also multi-view responsibilities
  - ◇ Support the ability to have multiple windows simultaneous display information from one application coordinator instance
  - ◇ Notify dependent windows when information changes and windows need to be refreshed
- Some frameworks include support for some form of application coordinator
- E.g., MFC application coordinators are Documents in the Document-View architecture, they are subclasses of CDocument

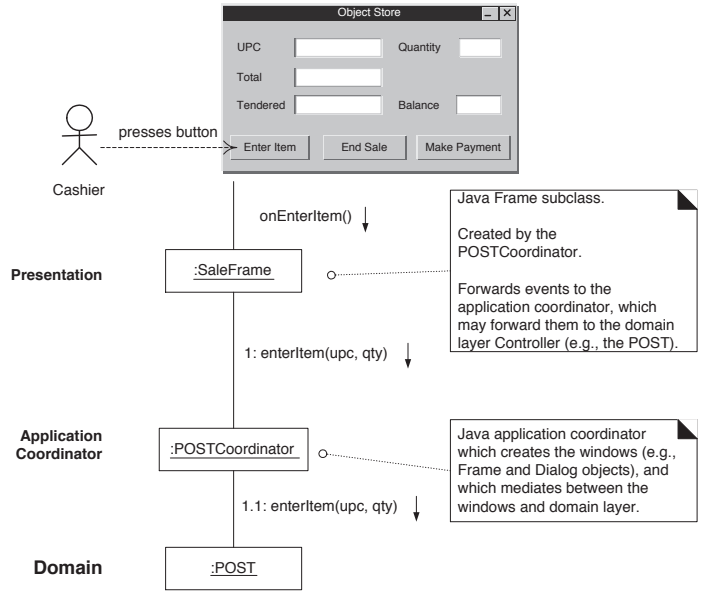
272

MFC Document-View Architecture Example



273

Application Coordinator in Java Application



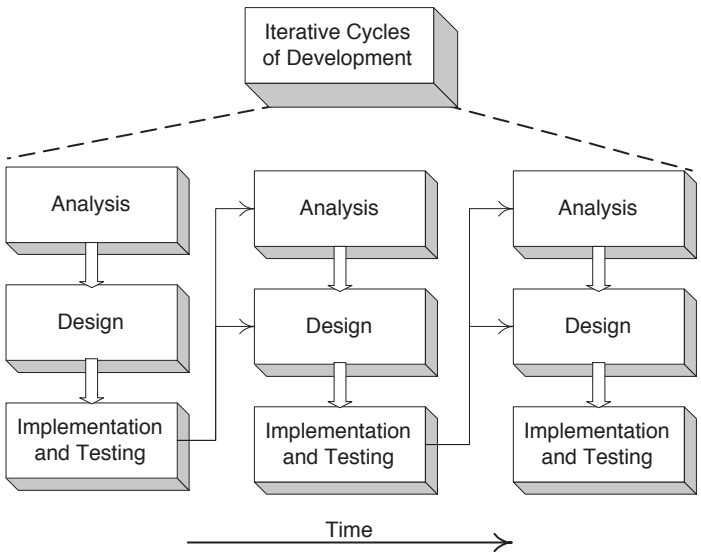
274

Mapping Designs to Code

- Strength of OAD and OPL used with a development process: provides a complete end-to-end road map from requirements through code
- Various artifacts feed into later artifacts in a traceable and useful manner
- Artifacts created during desing phase used as input to code generation process
- Core of the application: basic conceptual model, architectural layers, major allocations of responsibilities, major object interactions
- They are best determined in a formal investigation and design process
- Decision making and creative work done during the analysis and design phases
- Code generation may be a relatively mechanical translation process
- But, during programming and testing many changes will be made and detailed problems will be uncovered and resolved
- Design artifacts provide resilient core that scales up to meet the new problems encountered during programming

275

Code Changes and Iterative Process



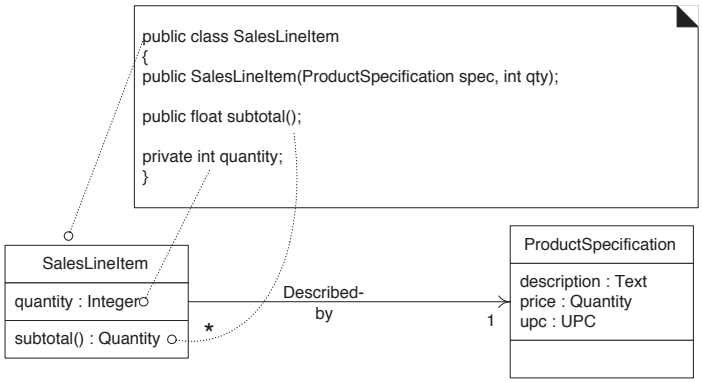
276

Code Changes and Iterative Process, cont.

- In iterative and incremental development process results of a prior cycle feed into the beginning of the next cycle
- Subsequent analysis and design results are continually being refined and informed from prior implementation work
- Diagrams generated during design must be updated to reflect changes in subsequent coding phase
- Should be done semi-automatically with a CASE tool that can read source code and automatically generate, e.g., class and collaboration diagrams
- This is an aspect of **reverse engineering**: generate logical models from executable source code

277

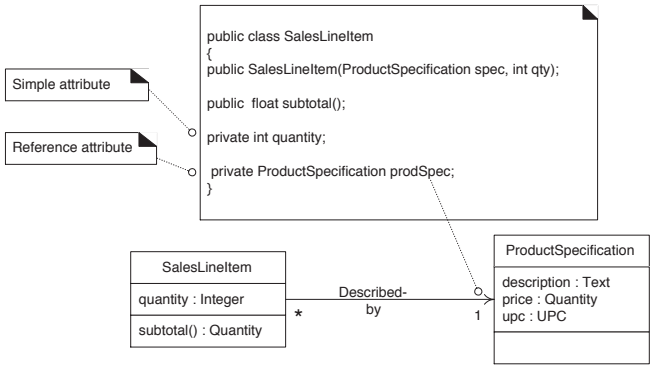
Creating Class Definitions from Design Class Diagrams



- Mapping basic attribute definitions and method signatures is straightforward
- Constructor derived from message create(spec,qty) sent to a SalesLineItem in the enterItem collaboration diagram
- Return type for subtotal method changed from Quantity to float

278

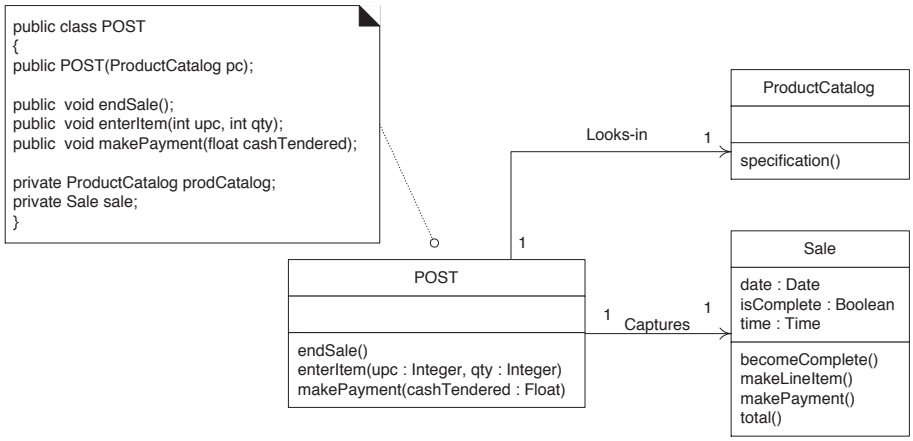
Adding Reference Attributes



- Reference attribute: attribute that refers to another complex object not to a primitive type
- They are suggested by associations and navigability in a class diagram, but they are not explicit
- Role names may be used to generate instance variable names

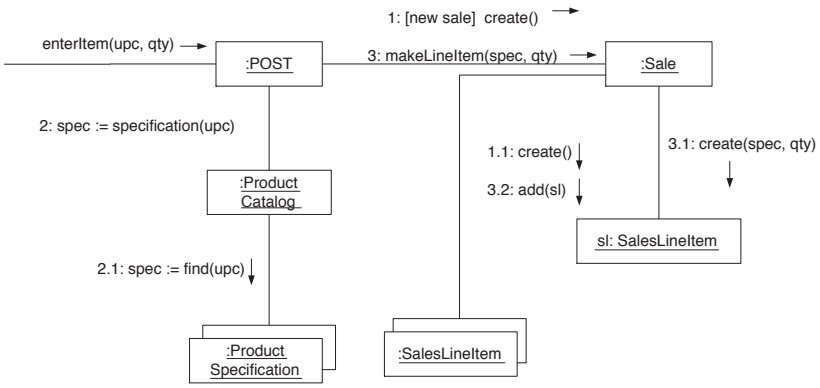
279

Generating Class Definitions: POST Class



280

Creating Methods from Collaboration Diagrams




```
public void enterItem(int upc, int qty) {
if ( isNewSale() ) { sale = new Sale; }
ProductSpecification spec = prodCatalog.specification(upc);
sale.makeLineItem(spec,qty);
}
```

281

Updating Class Definitions

- isNewSale method introduced in POST class while defining method enterItem
- Should test whether sale attribute is null

```
private boolean isNewSale() {
return ( sale == null )
}
```

- Not a good idea to hard-code this test into the enterItem method

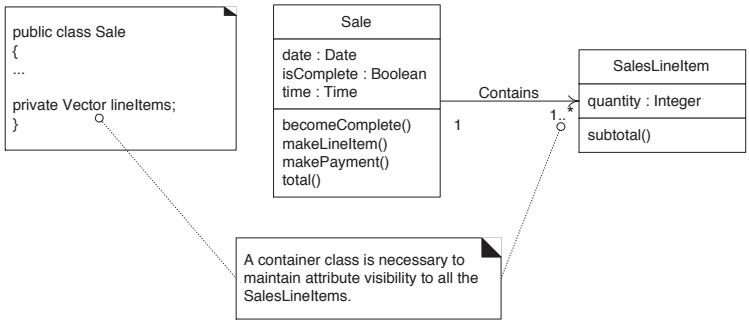
```
if ( isNewSale() )
  // versus
if ( sale == null )
```

- But, test in inadequate in the general case
- If one sale has completed and a second is about to begin, sale attribute will point to the last sale

```
private boolean isNewSale() {
return ( sale == null ) || ( sale.isComplete() );
}
```

282

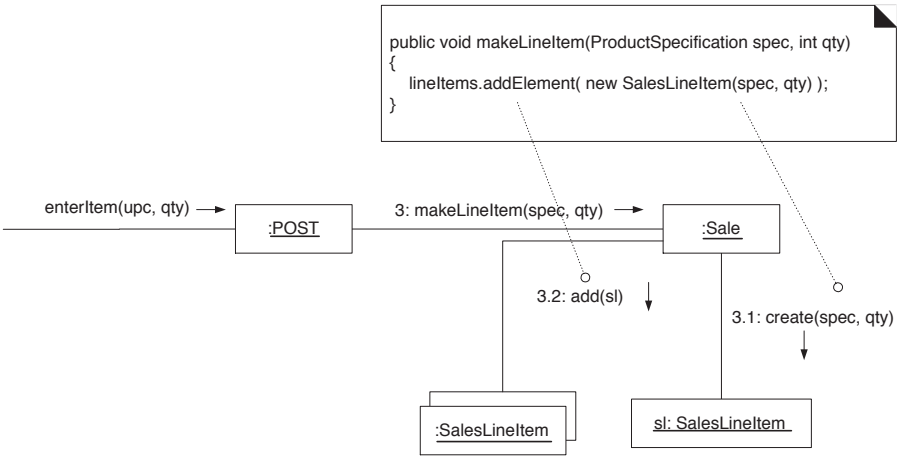
Container/Collection Classes in Code



- "1-Many" relationships relationships often implemented with the introduction of a container or collection
- 1-side class defines a reference attribute pointing to a container/collection instance, which contains instances of the many-side class
- Choice of container influenced by requirements: e.g., key-based lookup requires a Hashtable, growing ordered list requires a Vector

283

Defining the Sale–makeLineItem Method



284

Mapping Designs to Code

- Translation process relatively straightforward
  - ◇ from design-oriented class diagrams to class definitions
  - ◇ from collaboration diagrams to methods
- Still lots of room for decision-making, desing changes and exploration during programming phase
- But overall architecture and major decisions have ideally been completed prior to the coding phase

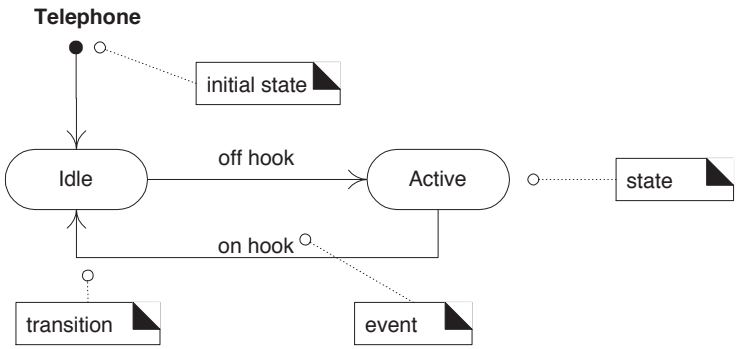
285

Modeling Behaviour with State Diagrams

- In UML used mainly for showing system events in use cases
- May be additionally applied to any type
- **Event**: significant or noteworthy occurrence
  - ◇ a telephone receiver is taken of the hook
- **State**: condition of an object at a moment in time — the time between events
  - ◇ a telephone is in state "idle" after the receiver is placed on the hook and until it is taken off the hook
- **Transition**: relationship between two states indicating that when an event occurs, the object moves from the prior state to the subsequent state
  - ◇ when the event "off hook" occurs, transition the telephone from state "idle" to state "active"

286

State Diagrams



- Shows the lifecyle of an object: interesting events and states of an object, and the behaviour of an object in reaction to an event
- Initial pseudo-state automatically transitions to another state on instance creation
- Need not to illustrate every possible event
- If an event arises that is not represented in the diagram, the event is ignored

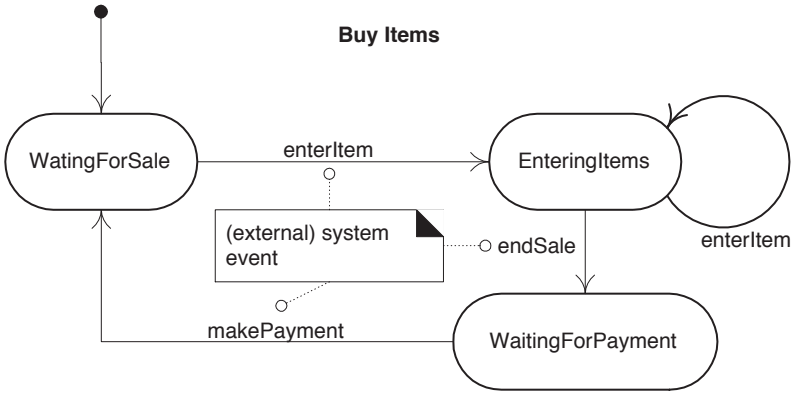
287

Subjects of a State Diagram

- May be applied to a variety of UML elements including
  - ◇ software classes
  - ◇ types (concepts)
  - ◇ use cases
- The entire system may be represented as a type, concept of a set of systems in a problem domain including distributed systems
- ⇒ The system may have its own state diagram

288

Use Case State Diagrams



- State diagrams describe legal sequence of external system events that are recognized and handled by a system in the context of a use case
- **Use case state diagram**: depicts the overall system events and their sequence within a use case

289

Utility of Use Case State Diagrams

- Complex use cases may have many system events
- A state diagram illustrating the legal order of external events is useful
- Design and implementation must ensure that no out-of-sequence events occur, otherwise an error condition is possible
  - ◇ POST should not be allowed to recieve a payment unless a sale is complete
- Posible design solutions include
  - ◇ hard-coded conditional tests for out-of-order events
  - ◇ use the State pattern (see later)
  - ◇ disabling widgets in active windows to disallow illegal events
  - ◇ a state machine interpreter that runs a state table representing use case state diagrams

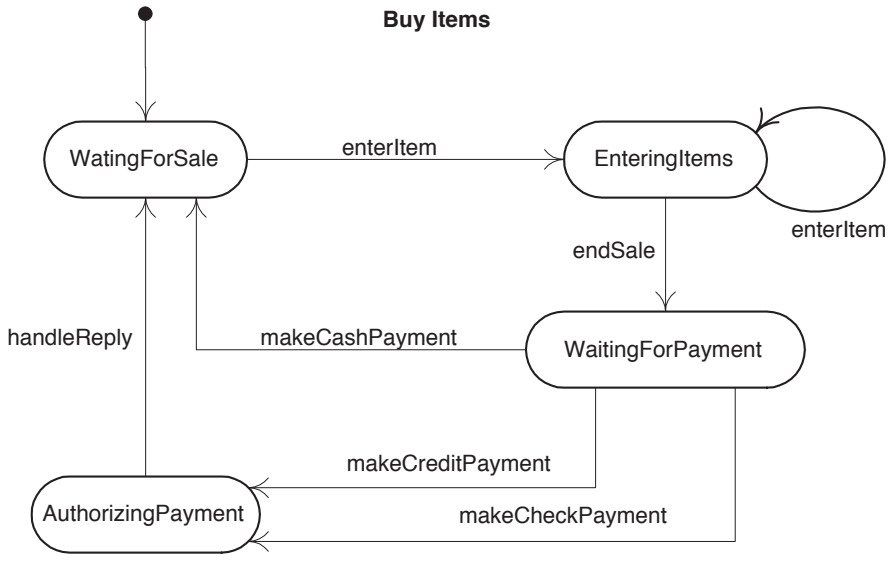
290

System State Diagrams

- Illustrates for one system all the transitions for system events across all the use cases
- It is a union of all use case state diagrams
- Only useful if total number of system events is small enough to keep it comprehensible

291

Point-of-Sale Application: Use Case State Diagram



292

Types with State Diagrams

- If an object always responds the same way to an event ⇒ **state-independent** (or modeless) wrt to that event
- **State-independent type**: always reacts the same way for all events of interest
- **State-dependent type**: reacts differently to events depending on their state interest
- State diagrams must be created for state-dependent types with complex behaviour
- Business information systems: minority of interesting state-dependent types
- Process control and telecommunication domains: many state-dependent objects

293

Common State-dependent Types and Classes

- Use cases (processes)
  - ◇ BuyItems reacts differently to endSale event if a sale is underway or not
- Systems
  - ◇ Point-of-sale system
- Windows
  - ◇ Edit-Paste action valid only if there is something to paste in clipboard
- Application coordinators
  - ◇ Applets in Java, Documents in MFC C++ Document-View framework
- Controllers
  - ◇ POST class, which handles the enterItem and endSale events
- Transactions
  - ◇ Sale receiving a makeLineItem message after the endSale event
- Devices
  - ◇ TV, VCR, modem
- Mutators: types that change their type or role
  - ◇ A person changing roles from student to employee

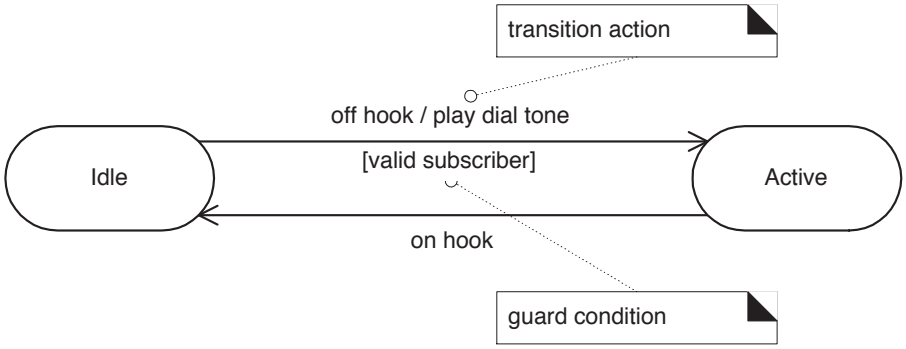
294

Event Types

- **External (System) Events**: caused by something outside system boundary
  - ◇ System sequence diagrams illustrate external events
  - ◇ External events causes invocation of system operations to respond to them
- **Internal Events**: caused by something inside system boundary
  - ◇ Arises when an operation is invoked via a message or signal sent by another internal object
  - ◇ Messages in collaboration diagrams suggest internal events
- **Temporal Events**: caused by the occurrence of a specific date and time or passage of time
  - ◇ Driven by a real-time or simulated-time clock

295

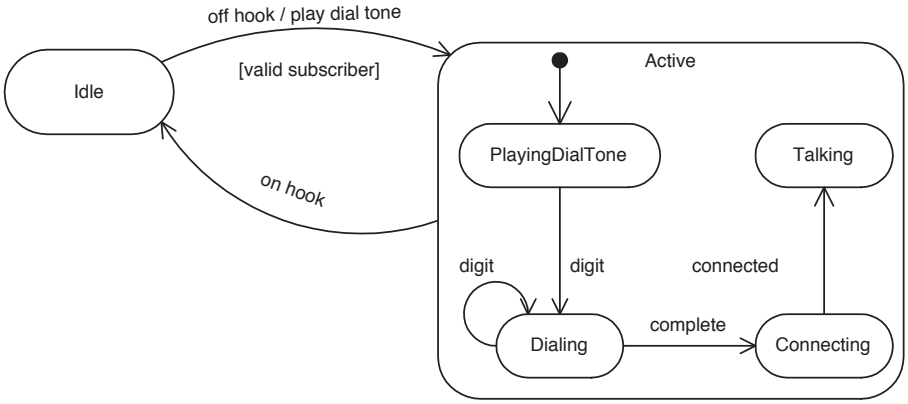
Transition Actions, Conditions



- A transition can cause an **action** to fire
  - ◇ may represent the invocation of a method of the class of the state diagram
- A transition may have a **conditional guard** (a boolean test)
  - ◇ transition is only taken if the test passes

296

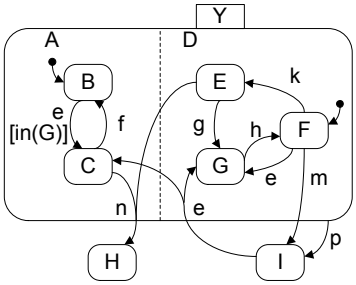
Nested States



- A state allows nesting to contain substates
- A substate inherits the transitions of its superstate (the enclosing state)
- This allows succint state diagrams

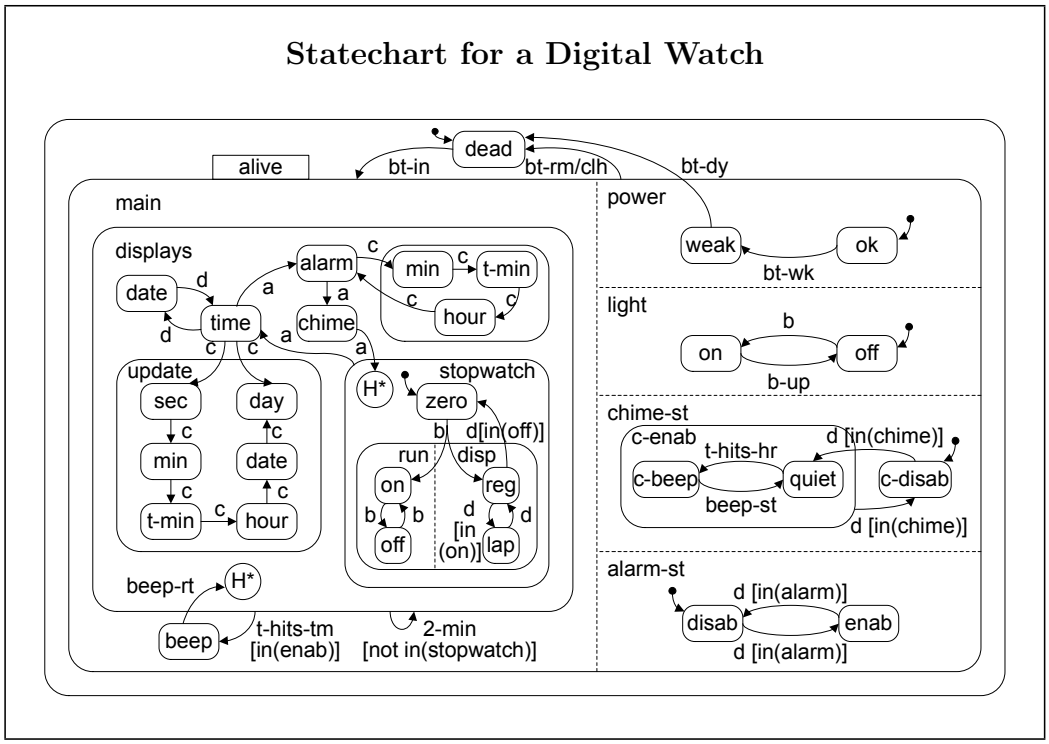
297

Concurrency in State Diagrams



- A state is decomposed into **orthogonal components**
- State Y consists of two orthogonal states A and D, each with its default state
  - ◇ To be in Y is tantamount to being in both A and D
- **Merging transitions**: From states C and E to H
- **Splitting transitions**: From I to C and G
- **Conditional transitions**: from C to B provided the system is in substate G in the other component

298



299

A simplified version of a digital watch.

David Harel, On Visual Formalisms. *Communications of the ACM*, 31(5), pp. 514-530 (May 1988).

The watch has four external control buttons, as well as a main display that can be used to show the time (hour, minutes, and seconds) or the date (weekday, day of month, and month). It has a chime that can be enabled or disabled, beeping on the hour if enabled. It has an alarm that can also be enabled, and beeps for 2 minutes when the time in the alarm setting is reached unless any one of the buttons is pressed earlier. It has a stopwatch with two display modes (regular and lap), a ligth for illumination, and a weak-battery blinking indication.

Some of the external events relevant to the watch are `a`, `b`, `c`, and `d`, which signify the pressing of the four buttons, respectively, and `b-up`, for example, which signifies the release of button `b`. Another event is `2-min`, which signifies that 2 minutes have elapsed since the last time a button was pressed.

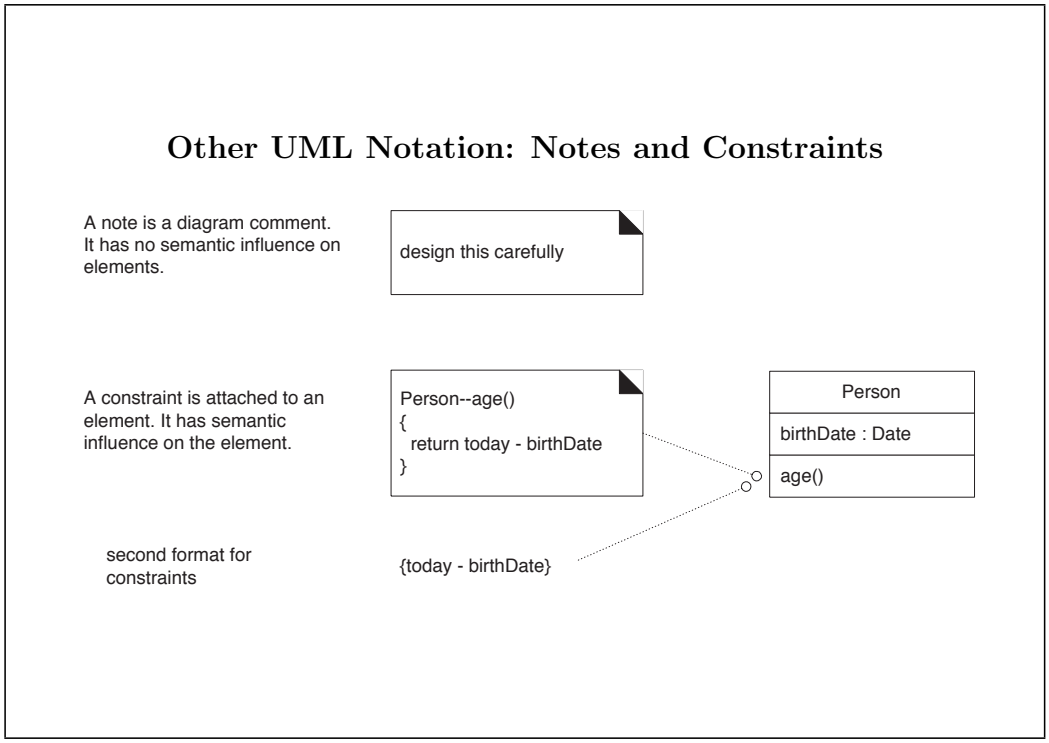
The specification of the watch contains examples of orthogonal states on various levels. The `stopwatch` state has two substates, `zero` and {`disp`,`run`}, the first being the default. Pressing `b` takes the stopwatch from the former to the latter causing it to start running with a regular display. Repeatedly pressing `b` causes it to stop and start alternately. Pressing `d` can be seen to cause the display to switch to `lap` and back to `reg`, or to leave the orthogonal state and return to `zero` depending on the present state configuration. The encircled and starred `H` prescribes that upon entering `stopwatch` from `chime` by pressing `a`, the state actually entered will be the one in which the system was in most recently. Thus, we are entering the stopwatch state by "history". The default will be used if this is the first time stopwatch is entered, or if the history has been cleared.

The description of the high levels of the watch also uses orthogonality. The watch is either `dead` or `alive`, with the latter consisting of five orthogonal components. The events `bt-in`, `bt-rm`, `bt-dy`, and `bt-wk` signify, respectively, the insertion, removal, expiration, and weakening (below a certain level) of the battery. We use the event `t-hits-hr` to signify

that the internal time of the watch has reached the internal time setting of the alarm, and `t-hits-hr` to signify that it has reached a whole hour. Also, `beep-rt` occurs when either any button is pressed or 2 minutes have elapsed since entering `beep`, and `beep-st` occurs 2 seconds after entering `c-beep`.

State `main` specifies the transitions between displaying and beeping. The `alarm-st` component describes the status of the alarm, specifiying that it can be changed using `d` when control is in the `alarm` display state. The `chime-st` state is similar, with the additional provision for beeping on the hour given within. The `power` state is self-explanatory, where the activity that would take place in the `weak` state would involve the displays blinking frantically.

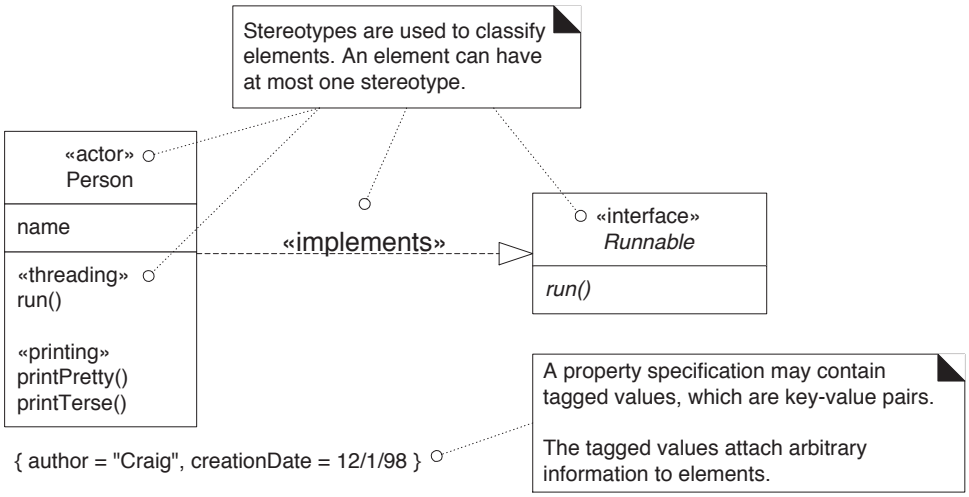
In considering the innocent-looking `light` state, the default is `off`, and depressing and releasing `b` cause the light to switch alternatively between `on` and `off`. What is interesting is the effect these actions might have elsewhere. If the entire statechart is contemplated, pressing `b` for illumination has significant side effects: It will cause a return from an `update` state if we happen to be in one, the stopping of the `alarm` if it happens to be beeping, and a change in the `stopwatch`'s behavior if we happen to be working with it. Conversely, if we use `b` in displays for any one of these things the light will go on, whether we like it or not. These seeming anomalies are all a result of the fact that the `light` component is orthogonal to the `main` component, meaning that its scope is very broad. One can imagine a far more humble `light` component, applicable only in the `time` and `date` states, which would not cause any of these problems. Its specification could be carried out by attaching it orthogonally, not to `main`, but to a new state surrounding `time` and `date`.



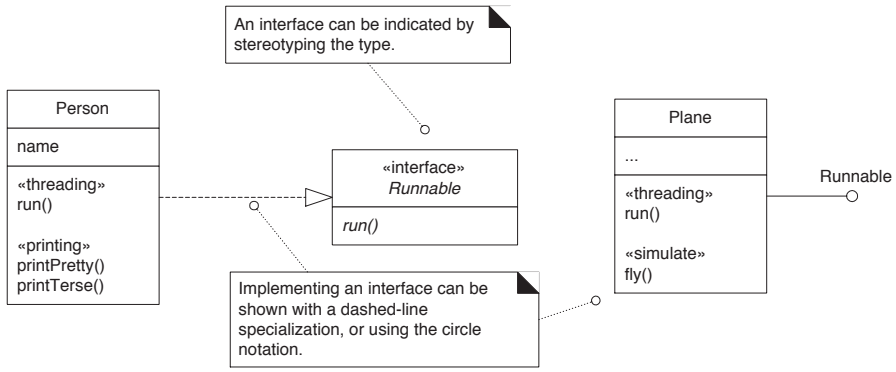
Other UML Notation: Dependency

- Dependencies can exist between any elements

Other UML Notation: Stereotypes and Property Specifications

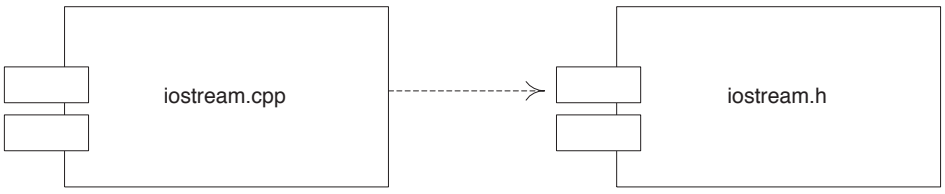


Other UML Notation: Interfaces



303

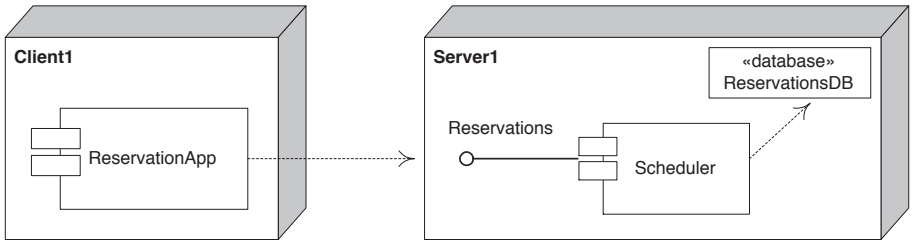
Other UML Notation: Component Diagrams



- Show compiler and runtime dependencies between software components, such as source code files and DLLs

304

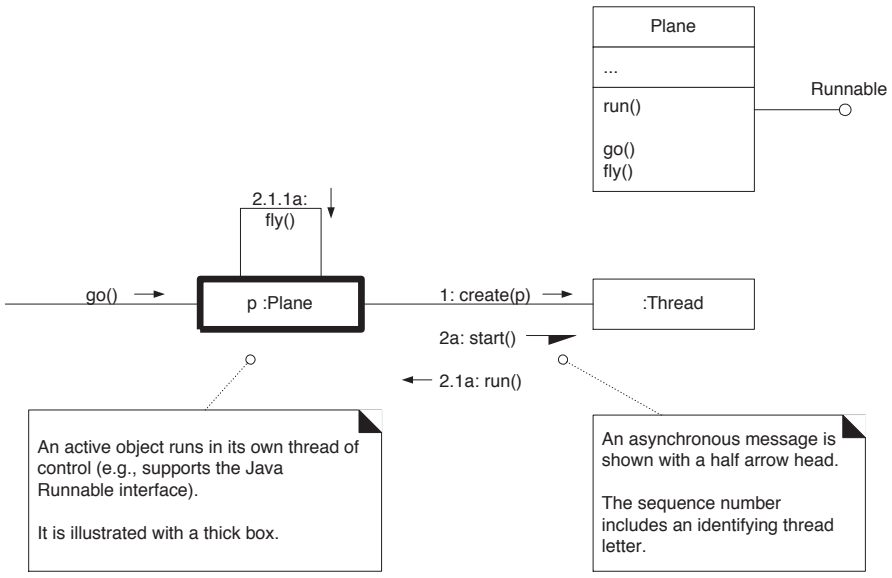
Other UML Notation: Deployment Diagrams



- Show the distribution of processes and components to processing nodes

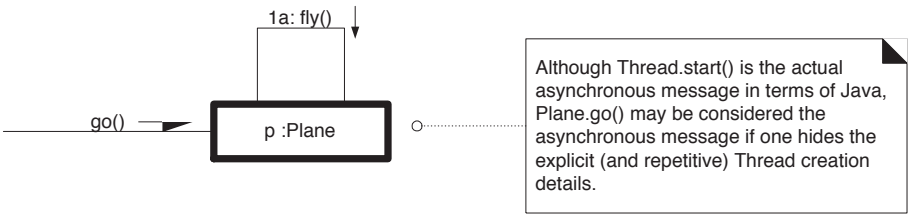
305

Other UML Notation: Asynchronous Messages



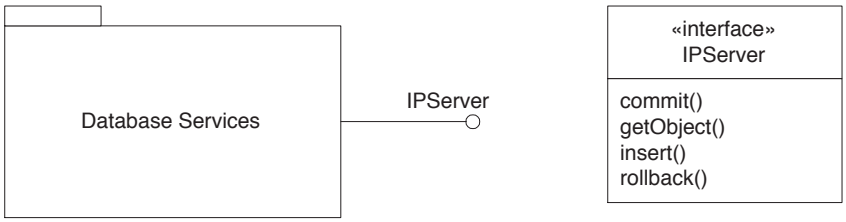
306

Other UML Notation: Asynchronous Messages, cont



307

Other UML Notation: Package Interfaces



- Packages may also indicate implementation of an interface which they expose to clients

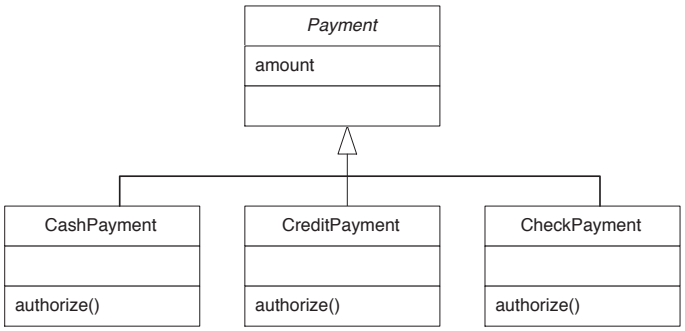
308

More Patterns: Polymorphism

- **Problem**:
  - ◇ How to handle alternatives based on type ?
    - ∗ Using if-then-else or case statement conditional logic, when a new variation arise this implies modification of program
    - ∗ Modifications tend to be required in several places
  - ◇ How to create pluggable software components ?
    - ∗ How can one replace one server component with another, without affecting the client ?
- **Solution**: When related alternatives or behaviours vary by type (class) assign responsibility for the behaviour (using polymorphic operations) to the types
- Avoid testing for the type of an object and using conditional logic

309

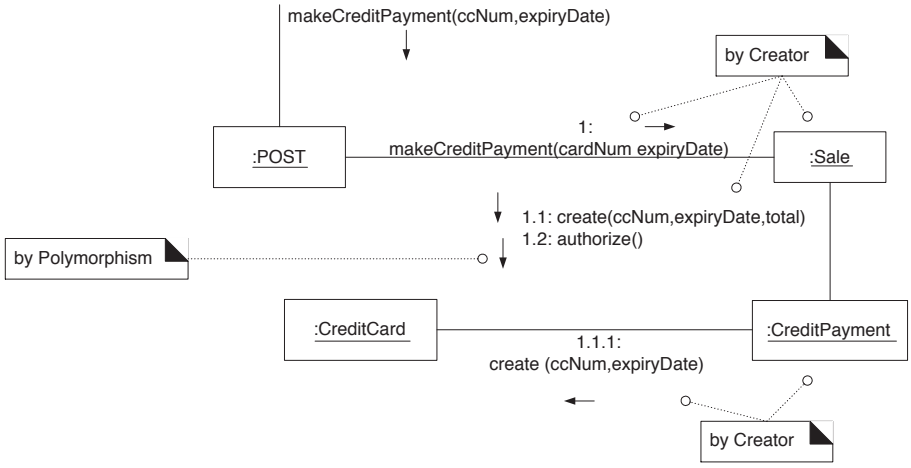
Polymorphism Pattern: Example



- Who should be responsible for authorizing different kinds of payments ?
- The behaviour of authorizing varies with the kind of payment
- By Polymorphism, the responsibility for authorizing should be assigned to each payment type
- Implemented with a polymorphic authorize operation
- Each implementation will communicate with a different authorization service

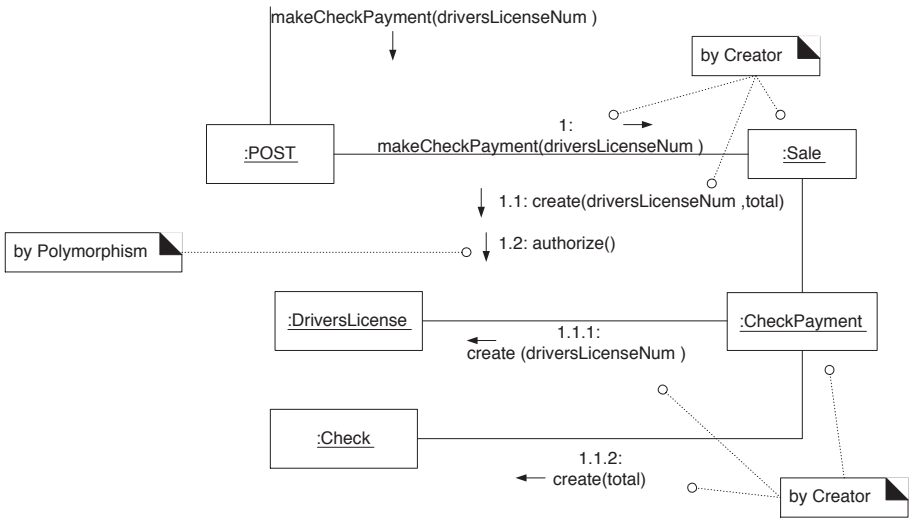
310

Polymorphism Example: Credit Payment



311

Polymorphism Example: Check Payment



312

Polymorphism Pattern: Discussion

- Expert: most important basic **tactical** pattern
- Polymorphism: most important basic **strategic** pattern
- Viewing objects in client-server relationships, client objects need little or no modification when a new server object is introduced
- Provided it supports the polymorphic operations that the client expects
- **Benefits**: Future extensions required for unanticipated new variations are easy to add
- **Also known as**: Do it Myself, Choosing Message, Don't Ask 'What Kind'

313

Pure Fabrication Pattern

- **Problem**: Who should have the responsibility, but do not want to violate High Cohesion and Low Coupling
  - ◇ In many situations, assigning responsibilities to domain classes leads to poor cohesion or coupling, low reuse potential
- **Solution**: Assign a highly cohesive set of responsibilities to an artificial class, does not represent anything in the problem domain
- Such a class is a **fabrication** of the imagination
- Its responsibilities must support high cohesion and low coupling ⇒ pure fabrication

314

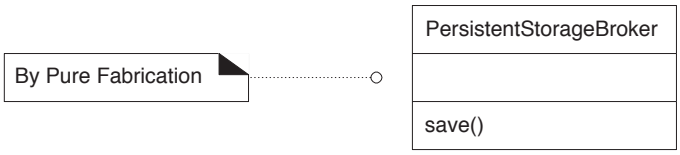
Pure Fabrication Pattern: Example

- Suppose Sale instances must be saved in a relational database
- By Expert, this responsibilities must be assigned to the Sale class
- Implications
  - ◇ Relatively large number of database-oriented operations, none of them related to the concept of Sale ⇒ class become incohesive
  - ◇ Sale class has to be coupled to the RDB interface (usually provided by the development tool vendor) ⇒ coupling increases
  - ◇ Saving objects in a RDB is a very general task needed by many classes ⇒ placing this reponsibilities in Sale implies poor reuse and code duplication

315

Pure Fabrication Pattern: Example, cont.

- Solution: create a class solely responsible for saving objects in some kind of persistent storage medium



- Sale remains well-designed with high cohesion and low coupling
- PersistentStorageBroker is
  - ◇ relatively cohesive
  - ◇ very generic and reusable

316

Pure Fabrication Pattern: Discussion

- A Pure Fabrication should have a high potential for reuse
- Their responsabilities are small and cohesive ⇒ fine-grained responsibilities
- Usually a function-centric object
- Usually considered part of the High-level Service Layer in an architecure
- Many design patterns are examples of Pure Fabrication
  - ◇ Adapter, Observer, Visitor, ...
- **Benefits**
  - ◇ High cohesion supported: responsibilities factored in fine-grain class focusing on a specific set of related tasks
  - ◇ Reuse potential may increase with several Pure Fabrications classes that can be used in other applications
- **Potential Problem**: Object-centric design may be lost, since in Pure Fabrications a class is made for a set of functions
  - ◇ May lead to a function or process-oriented design implemented in an OPL

317

Indirection Pattern

- **Problem**: To whom assign responsibility, for avoiding direct coupling ? How to de-couple objects so that Low Coupling is supported, reuse potential remains high ?
- **Solution**: Assign responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled
- Intermediary creates and **indirection** between the other components or services
- **Examples**
  - ◇ De-coupling the Sale from the RDB services through the introduction of PersistentStorageBroker: it is the intermediary between the Sale and the database
  - ◇ Publish-Subscribe or Observer pattern: through the indirection of an EventManager publishers and subscribers are de-coupled
  - ◇ Adapter, Façade are also examples
- Motivation for Indirection: low coupling

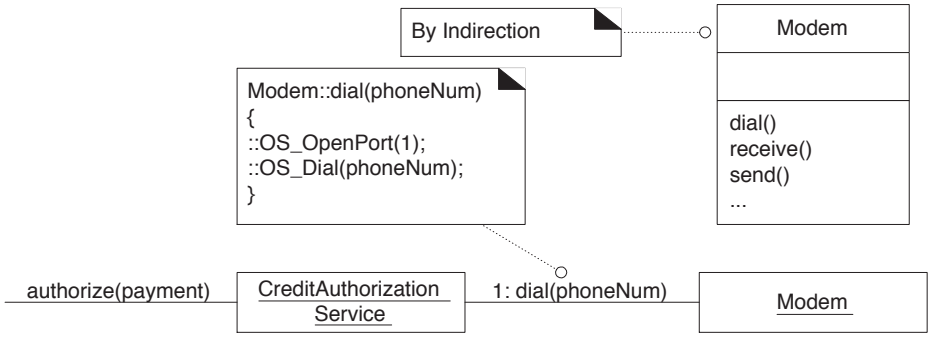
318

Indirection Pattern: Example

- POST application need to manipulate a modem for transmitting credit payment requests
- OS provides a low-level function call API for doing this
- Class CreditAuthorizationService is responsible for talking to the modem
- If this class invokes the low-level API function calls directly, highly coupled to the particular API
- If class must be ported to another OS it will require modification

319

Indirection Pattern: Example



- **Solution**: intermediate Modem class responsible for translating abstract modem resquests to the API
- **Device proxy**: class representing and interfacing with an electro-mechanical device

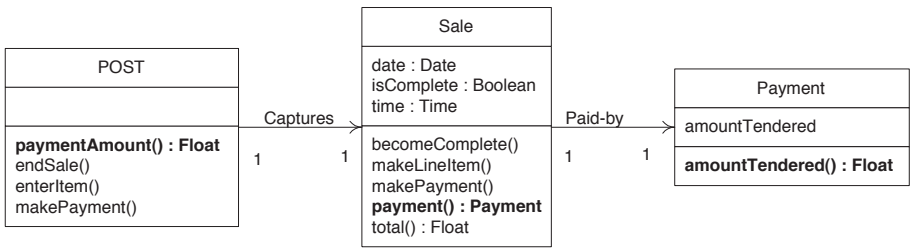
320

Don't Talk to Strangers Pattern

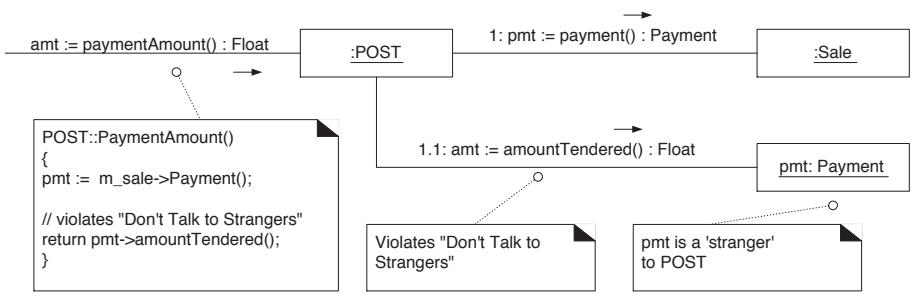
- **Problem**: Who should have the responsibility, to avoid knowing the structure of indirect objects ?
  - ◇ If an object has knowledge of internal connections and structure of other objects ⇒ high coupling
  - ◇ When a client has to use a service from an indirect object, how to do it without being coupled to the internals of the server or indirect objects
- **Solution**: A client's direct object should have the responsibility of collaborating with an indirect object
- A.k.a **Law of Demeter**: within a method, messages should only be sent to
  - ◇ the this or self object
  - ◇ a parameter of the method
  - ◇ an attribute of self
  - ◇ an element of a collection which is an attribute of self
  - ◇ an object created within the method
- Direct objects are a client's "familiars", indirect objects are "strangers"
- A client should only talk to familiars not to strangers

321

Don't Talk to Strangers Pattern: Example

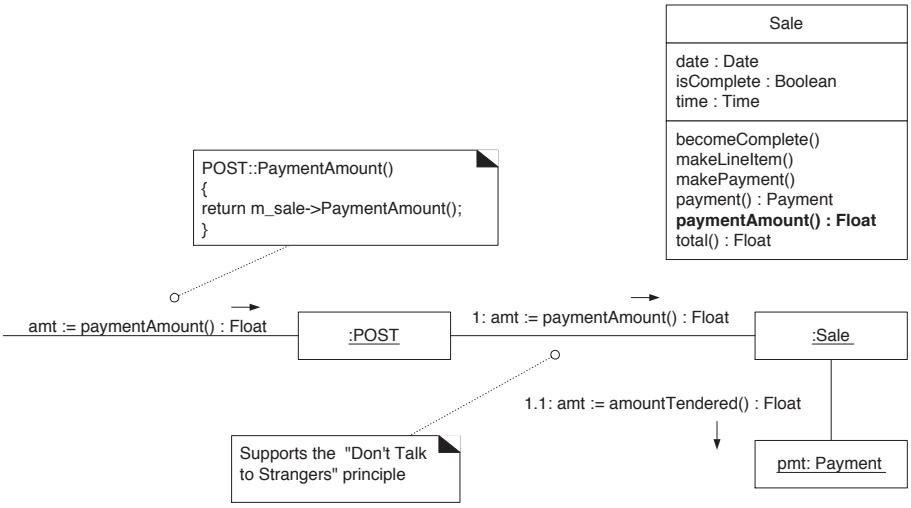


- One approach to return the payment amount



322

Don't Talk to Strangers Pattern: Example, cont.



Don't Talk to Strangers Pattern: Discussion

- Avoids gaining temporary visibility to indirect objects
- Disadvantage of gaining visibility to strangers: solution coupled to knowledge of internal structure of other objects
- This leads to high coupling, makes the design less robust, more likely to require a change if indirect structural relationships change
- But software laws are meant to be broken
- Sometimes it is reasonable to ignore the law of Demeter
- Example: a broker or object server responsible for returning other objects based upon lookup by a key value
- **Benefits**: Low coupling
- **Related Patterns**: Low coupling, Indirection, Chain of Responsibility

Designing with More Patterns

- Essence of object design: assignment of responsibilities to objects, object collaboration design
- Object design and assignment of responsibilities can be explained and learned based on the application of patterns: a vocabulary of principles and idioms that can be combined to design objects
- *Desing Patterns* by Gamma et al. (1995)
  - ◇ seminal work, presents 33 patterns (Gang of Four patterns)

325

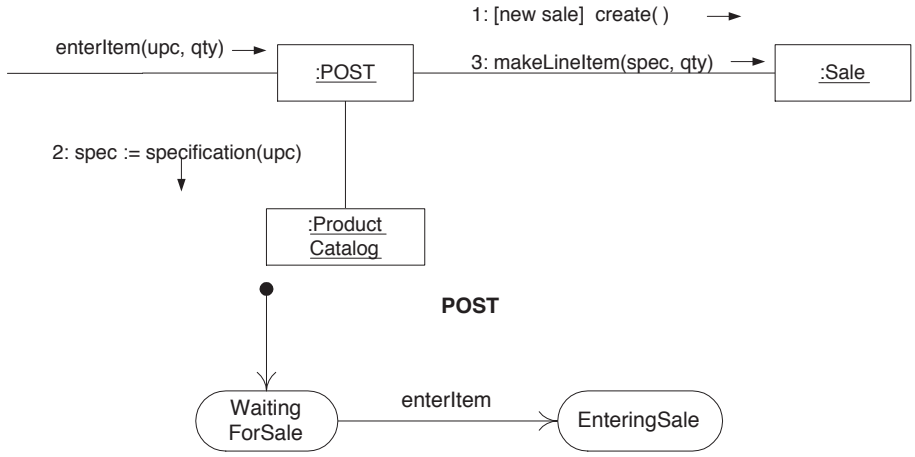
State Pattern

- **Problem**: An object's behaviour is dependent on its state
  - ◇ Conditional logic is undesirable because of complexity, scalability, duplication
- **Solution**
  - ◇ Create a class for each state that influences the behaviour of the state-dependent object (the "context" object
  - ◇ Based on Polymorphism, assign methods to each state class to handle the behaviour of the context calss
  - ◇ When a state-dependent message is received by the context object, forward it to the state object
- State pattern used to eliminate conditional tests caused by state dependencies

326

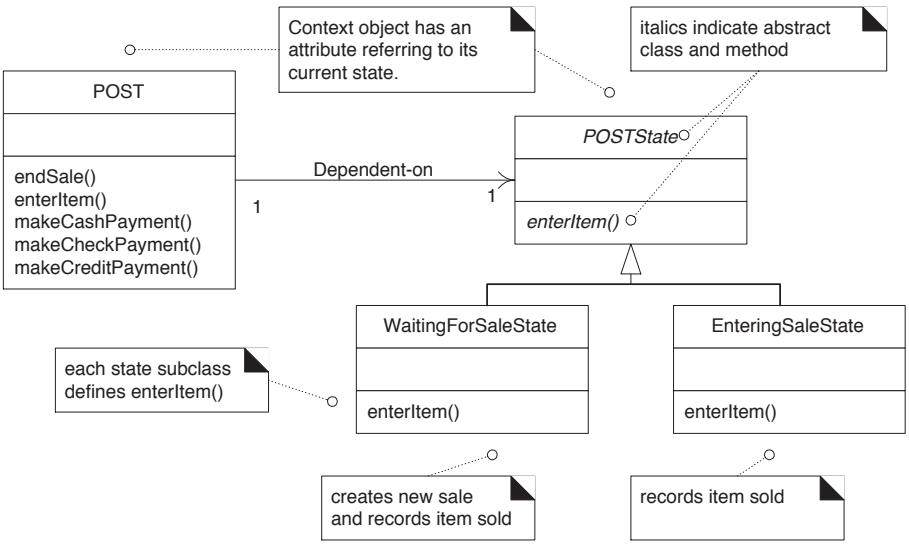
State Pattern: Example

- POST class reacts differently to EnterItem message depending on its state



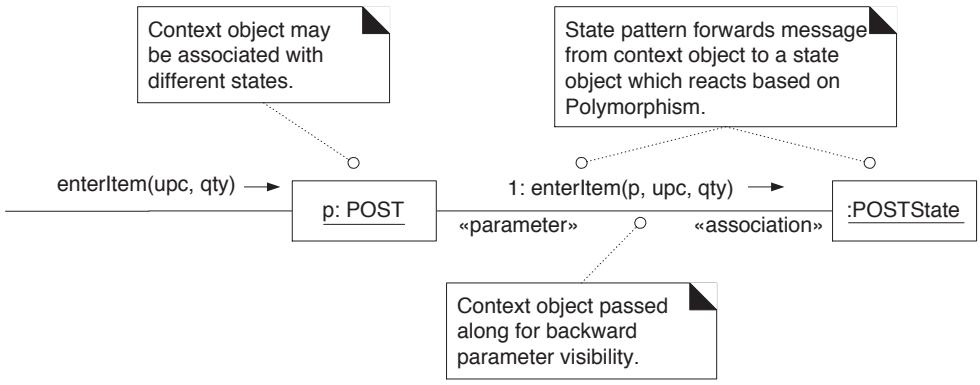
327

State Pattern Example: State Classes



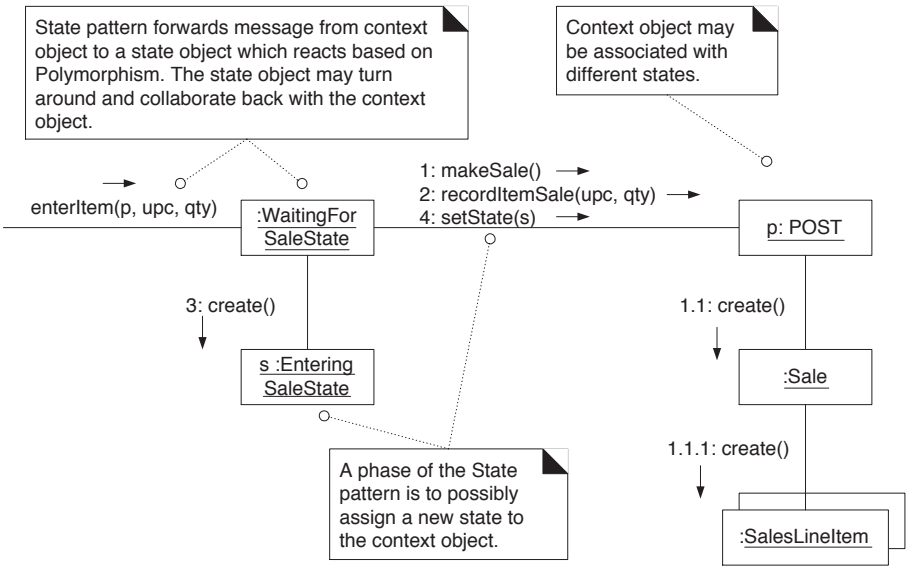
328

State Pattern Example: Forwarding Message



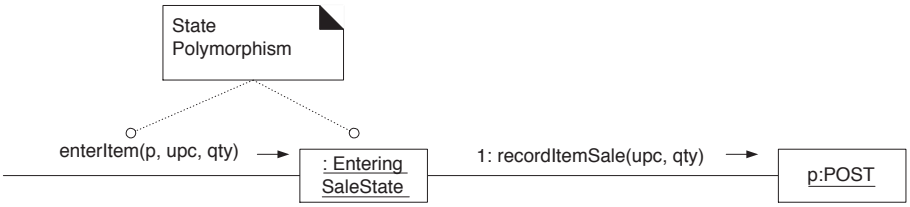
329

State Pattern Example: WaitingForSale State



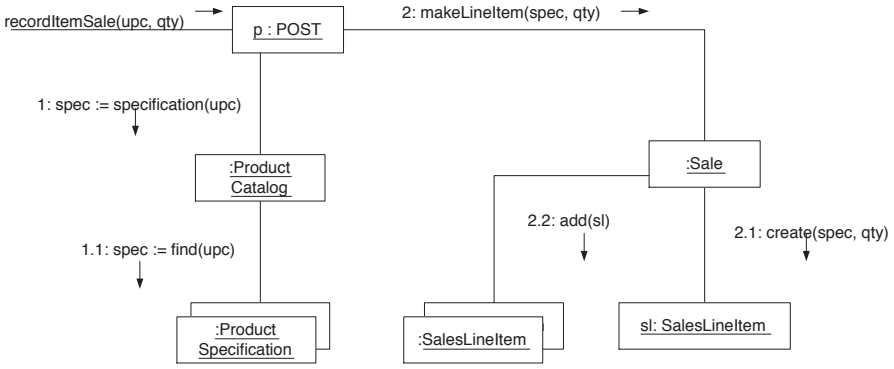
330

State Pattern Example: EnteringSale State



331

State Pattern Example: RecordItemSale Method



332

State Pattern: Conclusion

- Useful when an object's behaviour is dependent on its state
- Eliminates conditional logic in the methods of context object
- Provides elegant mechanism for extending behaviour of context object without modifying it
- Not suitable if many states in a system: class explosion
- Alternative: define a state machine interpreter that runs against a set of transition rules

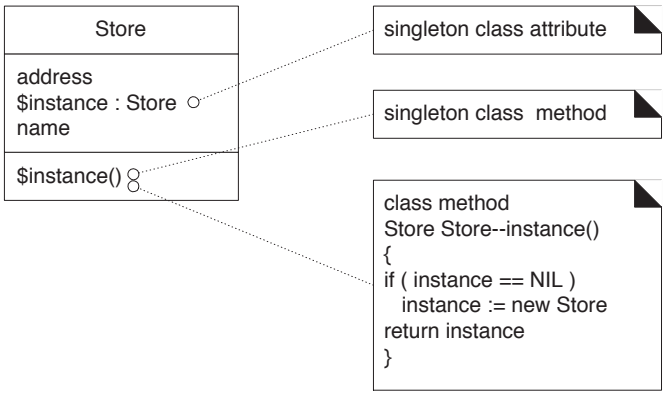
333

Singleton Pattern

- **Problem**: Exactly one instance of a class is allowed — it is a "singleton"
  - ◇ Objects need a single point of access
- **Solution**: Define a class method or non-member function (in C++) that returns the singleton
- **Example**: CreditPayment receives an Authorize message
- It needs to send a message to Store to find which authorization service to communicate with
- Visibility problem: CreditPayment does not have access to Store
- Alternatives
  - ◇ pass the Store down as a parameter
  - ◇ Singleton pattern

334

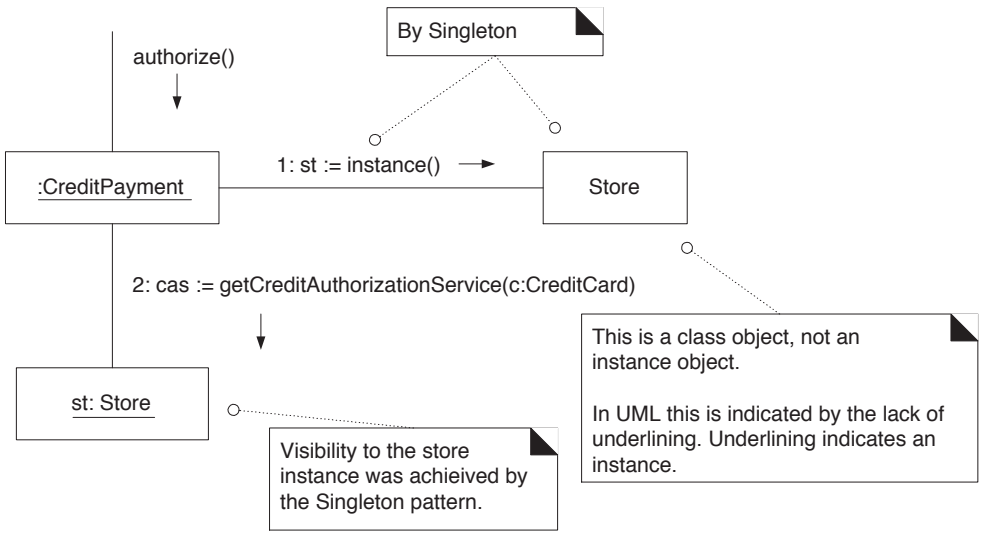
Singleton Pattern: Example



- Attribute instance and method instance(): implemented as static data and methods in Java

335

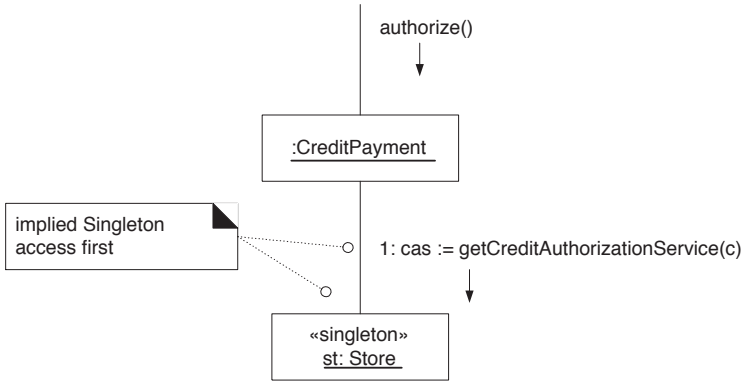
Singleton Pattern: Example, cont.



336

Singleton Pattern: Example, cont.

- UML Shorthand for a Singleton



- In an OPL, use of Singleton require sending a message to the class to get visibility to the instance, which can then be sent messages
  - ◇ C++: Store::getInstance()->getCAS(aCard);
  - ◇ Java: Store.getInstance().getCAS(aCard);

337

Remote Proxy and Proxy Patterns

Remote Proxy

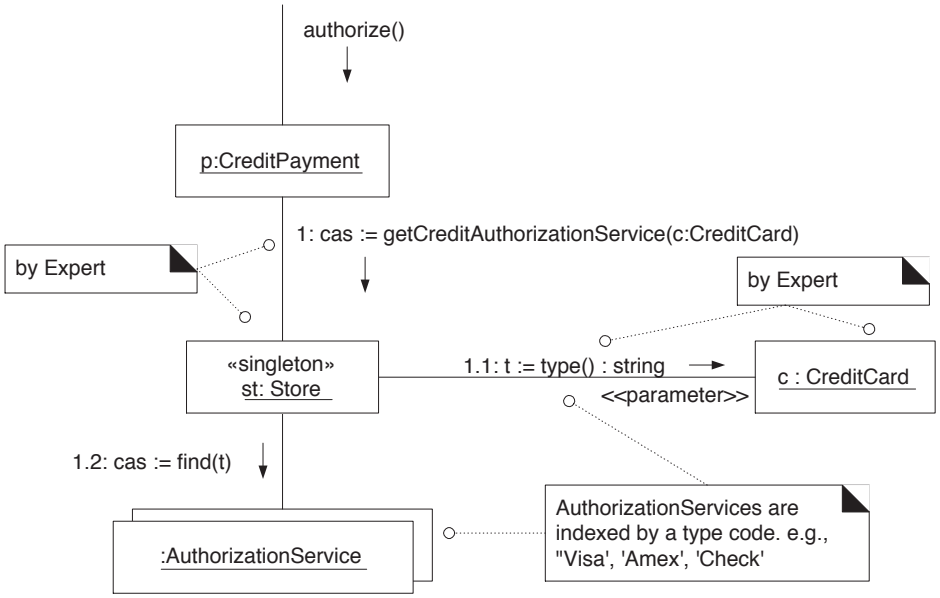
- **Problem**: The system must communicate with a component in another address space. Who should be responsible ?
- **Solution**: Make a local software class that represents the external component and give it the responsibility to communicating with the real component
- It is a special case of the general Proxy pattern

Proxy

- **Problem**: Direct access to a component is not desirable or possible.
- **Solution**: Define a surrogate software class that represents the component and give it responsibility for communicating with the real component

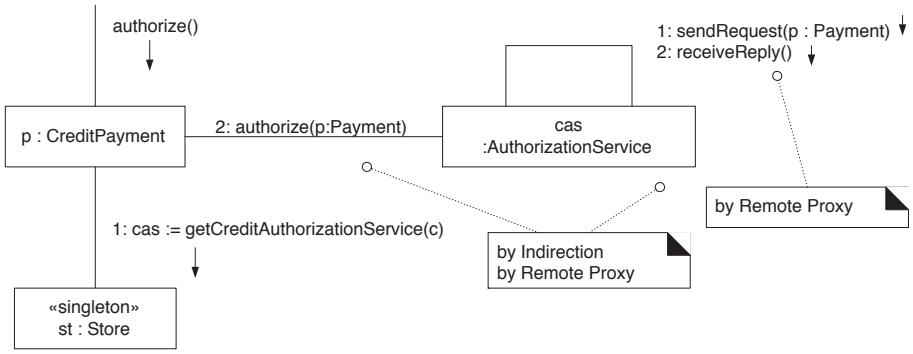
338

Remote Proxy Pattern: Example



339

Remote Proxy Pattern: Example, cont.



340

Wrapping and Façade

Wrapping

- POST system must use a modem to dial into the external service
- Underlying OS provides function-based interface for using a modem
- These non-object functions can be wrapped within a class that groups them together
- Applied to create an object interface to any non-object one

Façade

- **Problem**: Common, unified interface to a disparate set of interfaces (e.g., to a subsystem) is required
- **Solution**: Define a single class that unifies the interface and give it responsibility for communicating with the subsystem

341

Device Proxy

Device Proxy

- **Problem**: Interaction with an electro-mechanical device is required
- **Solution**: Define a single class that represents the device and give it responsibility for interacting with it

Indirection

- Façade, Remote Proxy, and Device Proxy, like many patterns, are a variation of the basic Indirection pattern

342

Marshaling and Serialization

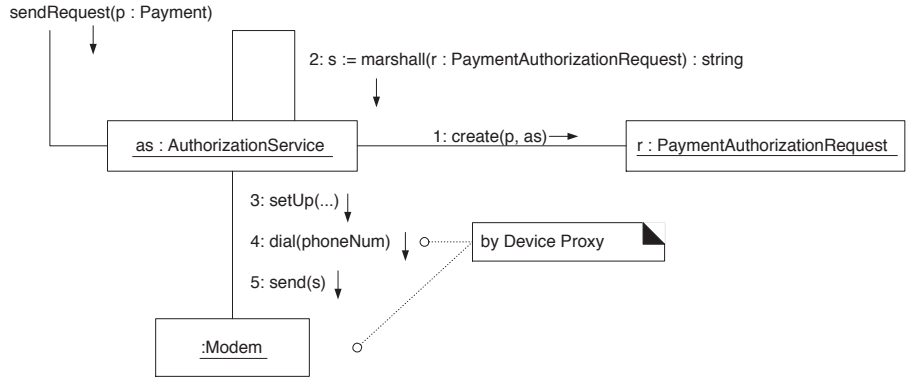
- **Serialization**: transformation of an object into a string representation
- Some languages (e.g., Java) provides built-in support
- **Marshalling**: sending messages and parameters to an object over a non-object comunication message (e.g., sockets, queues)
- Usually requires transforming the message and parameters (via serialization) into a stream of bytes suitable for transmission and for the receiving server
- **Unmarshalling**: transforming returning strings into commands and objects
- Responsibility for marshalling and serialization depends on the language and communication mechanism
- If using Java and its Remote Method Invocation (RMI) mechanism, only necessary to ensure that the serialization produces a suitable string layout for parameters
- Otherwise, usually the Remote Proxy is responsible for marshalling and unmarshalling

343

Marshalling The process of packing one or more items of data into a message buffer, prior to transmitting that message buffer over a communication channel. The packing process not only collects together values which may be stored in non-consecutive memory locations but also converts data of different types into a standard representation agreed with the recipient of the message.
Definition from "The Free On-line Dictionary of Computing" `http://foldoc.doc.ic.ac.uk/`

Serialization Object Serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI). From documentation of JDK.
Serialization is often thought of, in a limited sense, as a means to preserve objects in disk files. Actually, serialization abstracts the save/load mechanism away from any specific storage device, objects can be stored on disk or memory in various formats. Though developers often use the term serialization for both the store and load steps, the correct term for rebuilding objects is de-serialization.
From *Object Persistence and Versioning: Serialization in MFC*, by John Stout, Visual C++ Developpers Magazine, November 1997.

Using the Remote and Device Proxies



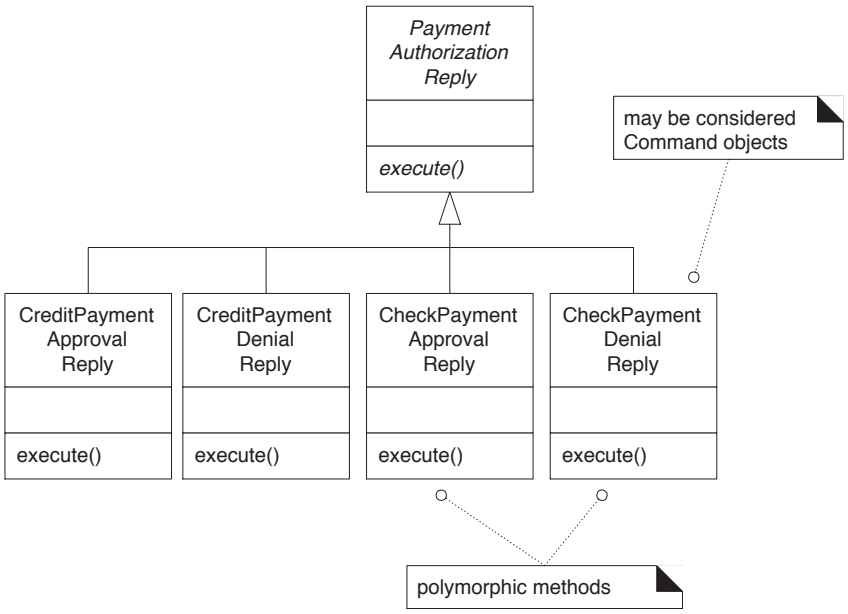
344

Command Pattern

- **Problem**
  - ◇ A variety of requests, or commands can be received by an object or system
  - ◇ Reduce the receiver's responsibility in handling the commands, increase the use with which new commands can be added, provide a foundation for logging, queuing, and undoing commands
- **Solution**: For each command, define a class that represents it and give it responsibility for executing it

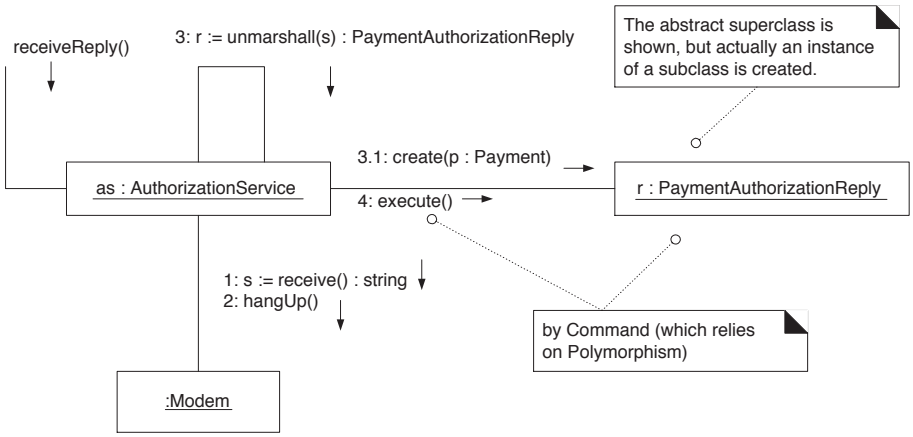
345

Command Classes: Example



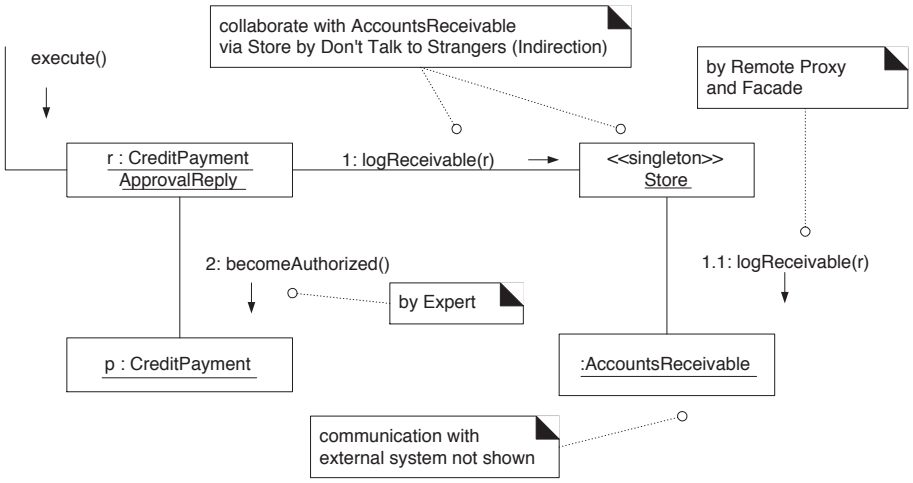
346

Command Pattern: Example



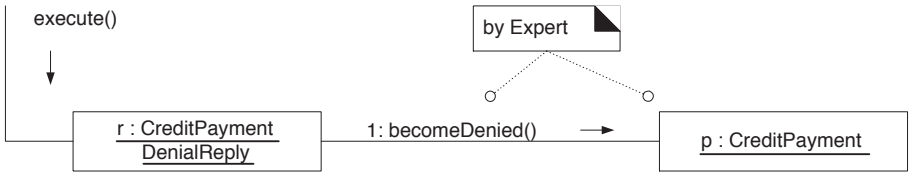
347

Command Pattern: CreditPaymentApprovalReply



348

Command Pattern: CreditPaymentDenialReply



349

UML: A Very Short Conclusion

- UML is unavoidable: became a standard for the modeling of applications
- Missing semanctics: it is hard to find a jugement about UML
- Ignores almost every theory that is known in the fields of conceptual modeling, software engineering, ...
- It is a "modern dinosaur"
- But, many work is being done around UML
  - ◇ Extensions, e.g., Real-Time UML, Agent UML, ...
  - ◇ Conferences, e.g., ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences) (11th edition in 2008)
- Little by little serious and formal work are finding its way into the standard