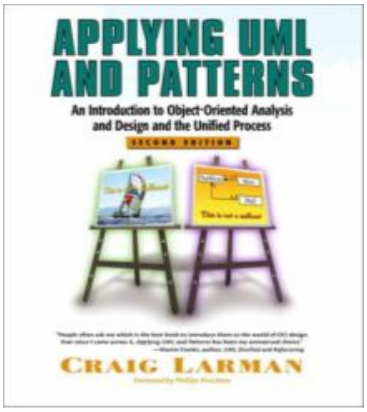
1



- **Disclaimer**: These slides are based on the 2nd edition of *"Applying UML and Patterns; An introduction to OOAD and the Unified process"* by Craig Larman (2002). I take responsibility for any errors.

Constantinos Constantinides
Computer Science and Software Engineering
Concordia University
Montreal, Canada
cc@cs.concordia.ca

2

The Need for Software Blueprints

- Knowing an object-oriented language and having access to a library is necessary but not sufficient in order to create object software.
- In between a nice idea and a working software, there is much more than programming.
- Analysis and design provide software “blueprints”, illustrated by a modeling language, like the Unified Modeling Language (UML).
- Blueprints serve as a tool for thought and as a form of communication with others.

3

Object-Oriented Analysis

- An investigation of the problem (rather than how a solution is defined)
- During OO analysis, there is an emphasis on finding and describing the objects (or concepts) in the problem domain.
  - For example, concepts in a Library Information System include *Book*, and *Library*.

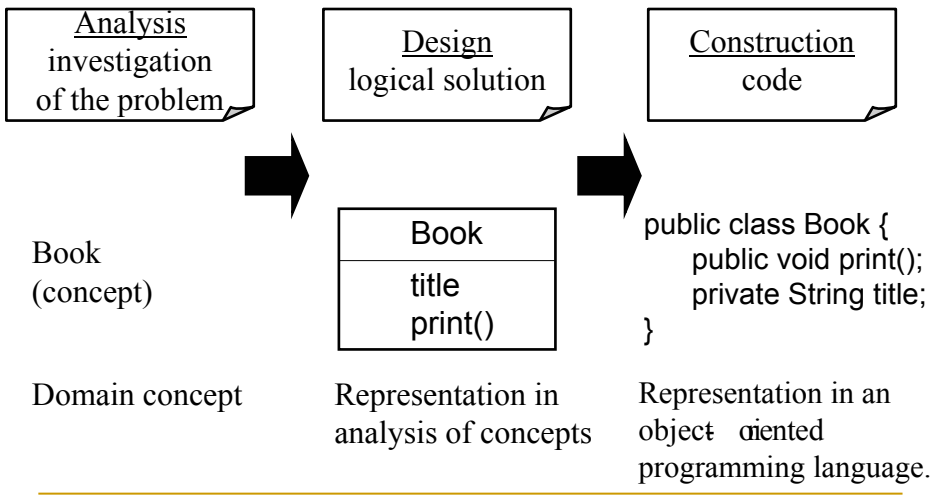
4

Object-Oriented Design

- Emphasizes a conceptual solution that fulfils he requirements.
- Need to define software objects and how they collaborate to fulfil the requirements.
  - For example, in the Library Information System, a *Book* software object may have a *title* attribute and a *getChapter* method.
- Designs are implemented in a programming language.
  - In the example, we will have a *Book* class in Java.

5

From Design to Implementation



6

Iterative Development and the Unified Process

7

The Unified Process (UP)

- A process is a set of partially ordered steps intended to reach a goal.
- In Software Engineering, the goal is to efficiently and predictably deliver a software product that meets the needs of your business.
- A **software development process** is an approach to building, deploying and maintaining software.
- The **Unified Process (UP)** is a process for building object-oriented systems.
- The goal of the UP is to enable the production of high quality software that meets users needs within predictable schedules and budgets.

8

The Unified Process (UP)

- For simple systems, it might be feasible to sequentially define the whole problem, design the entire solution, build the software, and then test the product.
- For complex and sophisticated systems, this linear approach is not realistic.
- The UP promotes **iterative development**:The life of a system stretches over a series of cycles, each resulting in a product release.

9

Iterative Development

- Development is organized into a series of short fixed-length mini-projects called **iterations**.
- The outcome of each iteration is a tested, integrated and executable system.
- An iteration represents a complete development cycle: it includes its own treatment of requirements, analysis, design, implementation and testing activities.

10

Iterative Development

- The iterative lifecycle is based on the successive enlargement and refinement of a system though multiple iterations with feedback and adaptation.
- The system grows incrementally over time, iteration by iteration.
- The system may not be eligible for production deployment until after many iterations.

11

Iterative Development

[iteration N]
Requirements – Analysis - Design- Implementation - Testing

[iteration N+1]
Requirements – Analysis - Design- Implementation - Testing

Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.

The system grows incrementally.

12

Iterative Development

- The output of an iteration is not an experimental prototype but a production subset of the final system.
- Each iteration tackles new requirements and incrementally extends the system.
- An iteration may occasionally revisit existing software and improve it.

13

Embracing Change

- Stakeholders usually have changing requirements.
- Each iteration involves choosing a small subset of the requirements and quickly design, implement and testing them.
- This leads to rapid feedback, and an opportunity to modify or adapt understanding of the requirements or design.

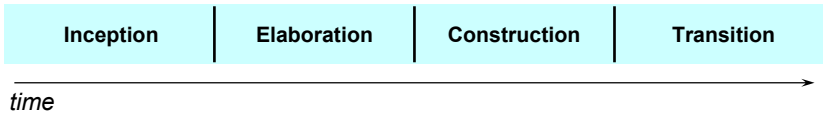
14

Iteration Length and Timeboxing

- The UP recommends short iteration lengths to allow for rapid feedback and adaptation.
- Long iterations increase project risk.
- Iterations are fixed in length (**timeboxed**).If meeting deadline seems to be difficult, then remove tasks or requirements from the iteration and include them in a future iteration.
- The UP recommends that an iteration should be between two and six weeks in duration.

15

Phases of the Unified Process

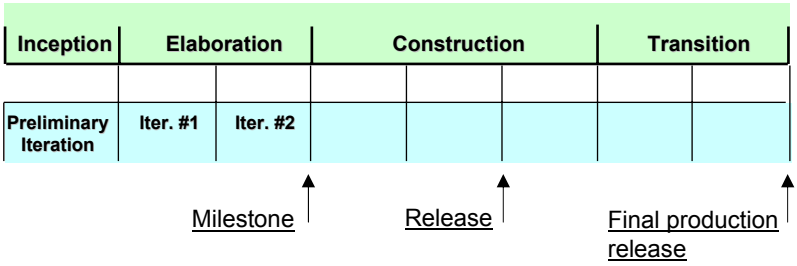


A UP project organizes the work and iterations across four major phases:

- Inception - Define the scope of project.
- Elaboration - Plan project, specify features, baseline architecture.
- Construction - Build the product
- Transition - Transition the product into end user community

16

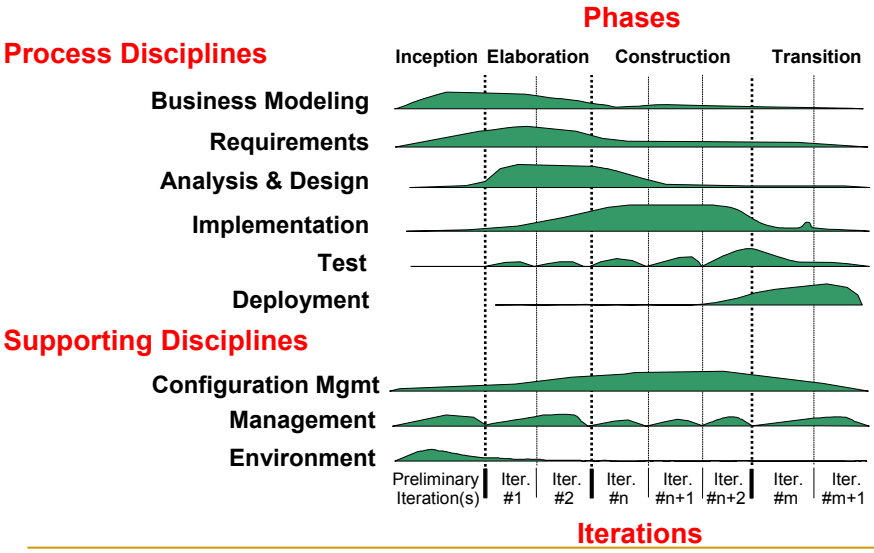
Iterations and Milestones



- Each phase and iteration has some risk mitigation focus, and concludes with a well-defined milestone.
- The milestone review provides a point in time to assess how well key goals have been met and whether the project needs to be restructured in any way to proceed.
- The end of each iteration is a minor release, a stable executable subset of the final product.

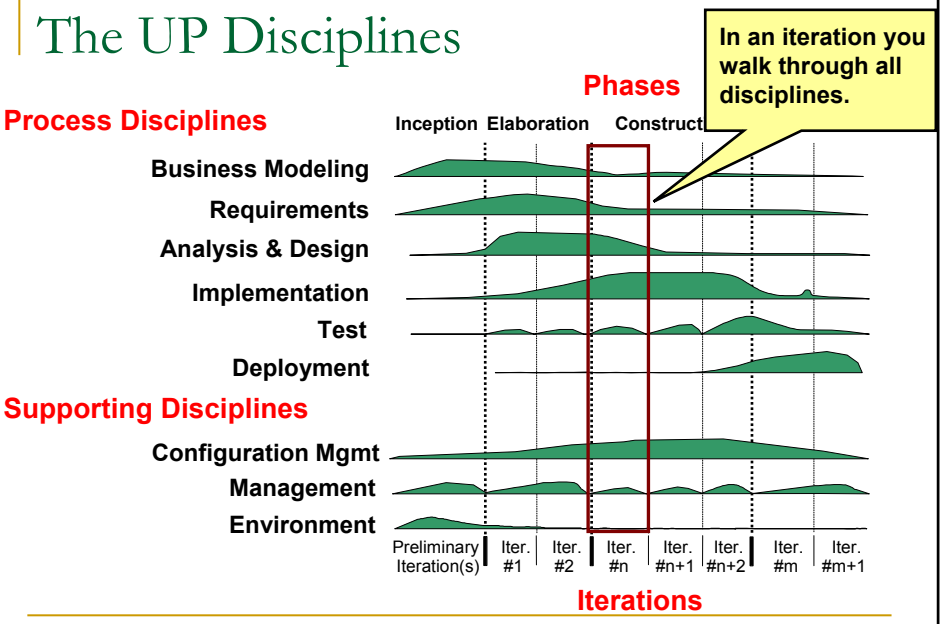
17

The UP Disciplines



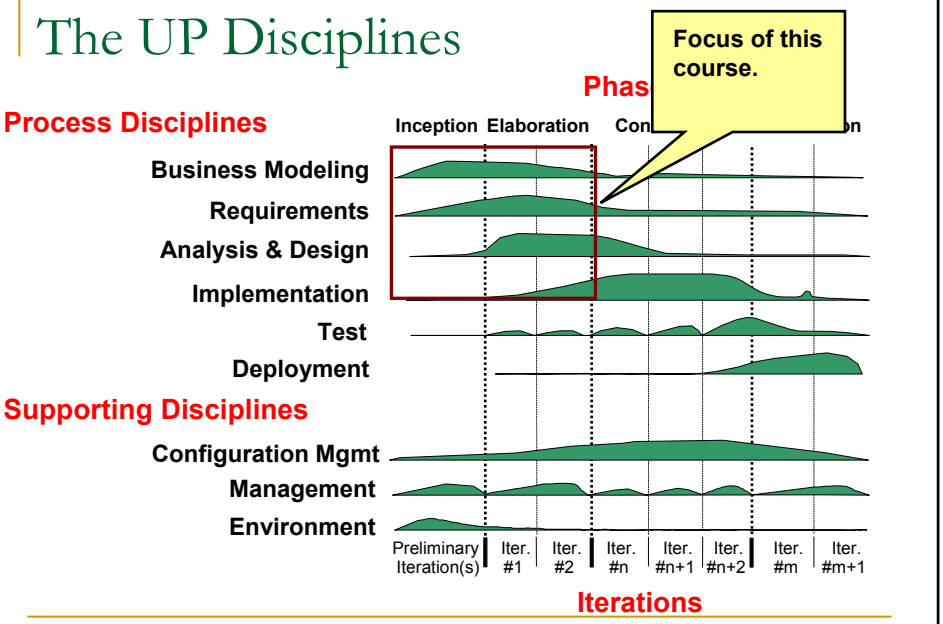
18

The UP Disciplines



19

The UP Disciplines



20

Disciplines and Phases

- Although an iteration includes work in most disciplines, the relative effort and emphasis change over time.
  - Early iterations tend to apply greater emphasis to requirements and design, and later ones less so.
  - Figure illustrations are suggestive, not literal.
- Note that activities and artifacts are *optional* (except code!)
  - Developers select those artifacts that address their particular needs.

21

Advantages of an Iterative Process

- Reduce risks
  - Risks are identified early, progress is easier to see.
- Get a robust architecture
  - Architecture can be assessed and improve early.
- Handle evolving requirements
  - Users provide feedback to operational systems.
  - Responding to feedback is an incremental change.
- Allow for changes
  - System can adapt to problems
- Attain early learning
  - Everyone obtains an understanding of the different workflows early on.

22

Inception

23

Inception

- A number of questions need to be explored:
  - What is the vision and business case for this project?
  - Is it feasible?
  - Buy and/or build?
  - Rough estimate of cost.
  - Should we proceed or stop?
- The intent is to establish some initial common vision for the objectives of the project, determine if it is feasible and decide if it is worth some serious investigation in elaboration.

24

What artifacts may start in inception?

- Vision and business case
  - Describes high-level goals and constraints.
- Use Case model
  - Describes functional requirements and related non-functional requirements.
- Supplementary specification
  - Describes other requirements
- Glossary
  - Key domain terminology
- Risk list and Risk Management Plan
  - Describes business, technical, resource and schedule risks and ideas for their mitigation or response.

25

What artifacts may start in inception?

- Prototypes and proof-of-concepts
- Iteration plan
  - Describes what to do in the first elaboration iteration
- Phase Plan & Software development Plan
  - Guess for elaboration phase duration. Tools, people, education and other resources.
- Development Case
  - Description of the customized UP steps and artifacts for this project.
- Artifacts will be partially completed in this phase and will be refined in later iterations.

26

Understanding Requirements

27

Introduction to Requirements

- Requirements are system capabilities and conditions to which the system must conform.
- Functional requirements
  - Features and capabilities.
  - Recorded in the Use Case model (see next), and in the systems features list of the Vision artifact.
- Non-functional (or quality requirements)
  - Usability (Help, documentation, …), Reliability (Frequency of failure, recoverability, …), Performance (Response times, availability, …), Supportability (Adaptability, maintainability, …)
  - Recorded in the Use Case model or in the Supplementary Specifications artifact.
- The nature of UP supports changing requirements.

28

Use-Case Model: Writing Requirements in Context

29

Use cases and adding value

- Actor: something with behavior, such as a person, computer system, or organization, e.g. a cashier.
- Scenario: specific sequence of actions and interactions between actors and the system under discussion, e.g. the scenario of successfully purchasing items with cash.
- Use case: a collection of related success and failure scenarios that describe actors using a system to support a goal.

30

Use cases and adding value

Handle returns

Main success scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item…

Alternate scenarios:

If the credit authorization is reject, inform customer and ask for an alternative payment method.

If item identifier not found in the system, notify the Cashier and suggest manual entry of the identifier code.

…

31

Use cases and adding value

- A key point is to focus on the question “how can using the system provide observable value to the user, or fulfill their goals?”
- Use cases mainly constitute functional requirements.

32

Use case types and formats

- Black-box use cases describe system responsibilities, i.e. define what the system must do.
- Uses cases may be written in three formality types
  - Brief: one-paragraph summary, usually of the main success scenario.
  - Casual: Informal paragraph format (e.g. Handle returns)
  - Fully dressed: elaborate. All steps and variations are written in detail.

33

Fully-dressed example: Process Sale

Use case UC1: Process Sale
Primary Actor: Cashier
Stakeholders and Interests:
-Cashier: Wants accurate and fast entry, no payment errors, …
-Salesperson: Wants sales commissions updated.
…
Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions):
-Sale is saved. Tax correctly calculated.
…
Main success scenario (or basic flow): [see next slide]
Extensions (or alternative flows): [see next slide]
Special requirements: Touch screen UI, …
Technology and Data Variations List:
-Identifier entered by bar code scanner,…
Open issues: What are the tax law variations? …

34

Fully dressed example: Process Sale (cont.)

Main success scenario (or basic flow):

The Customer arrives at a POS checkout with items to purchase.
The cashier records the identifier for each item. If there is more than one of the same item, the Cashier can enter the quantity as well.
The system determines the item price and adds the item information to the running sales transaction. The description and the price of the current item are presented.
On completion of item entry, the Cashier indicates to the POS system that item entry is complete.
The System calculates and presents the sale total.
The Cashier tells the customer the total.
The Customer gives a cash payment ("cash tendered") possibly greater than the sale total.

Extensions (or alternative flows):

If invalid identifier entered. Indicate error.
If customer didn't have enough cash, cancel sales transaction.

35

Goals and Scope of a Use Case

- At what level and scope should use cases be expressed?
- A: Focus on uses cases at the level of **elementary business process** (EBP).
- EBP: a task performed by one person in one place at one time which adds measurable business value and leaves the data in a consistent state.
  - Approve credit order - OK.
  - Negotiate a supplier contract - not OK.
- It is usually useful to create separate "sub" use cases representing subtasks within a base use case.
  - e.g. Paying by credit

36

Finding primary actors, goals and use cases

- Choose the system boundary.
- Identify primary actors.
  - Those that have user goals fulfilled through using services of the system
- For each actor identify their user goals.
  - Tabulate findings in the Vision artifact.
- Define use cases that satisfy user goals; name them according to their goal.

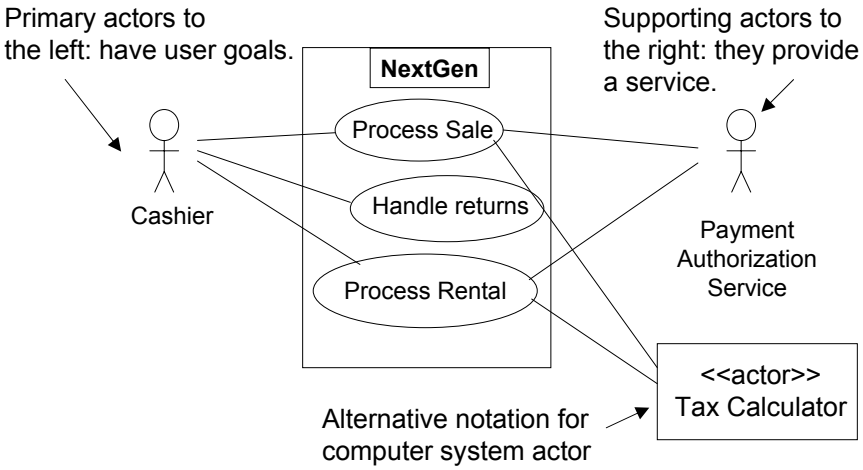
37

Essential vs. Concrete style

- Essential: Focus is on intend.
  - Avoid making UI decisions
- Concrete: UI decisions are embedded in the use case text.
  - e.g. “Admin enters ID and password in the dialog box (see picture X)”
  - Concrete style not suitable during early requirements analysis work.

38

Use Case Diagrams



39

From Inception to Elaboration

40

Iteration 1 Requirements

- Implement a basic key scenario of the Process Sale use case: entering items and receiving a cash payment.
- No collaboration with external devices (such as tax calculator or product database)
- No complex pricing rules are applied.
- Subsequent iterations will grow on this foundation.

41

Incremental Development for the Same Use Case Across Iterations

- Not all requirements in the Process Sale use case are being handled in iteration 1.
- It is common to work on varying scenarios or features of the same use case over several scenarios and gradually extend the system to ultimately handle all the functionality required.
- On the other hand, short, simple use cases may be completed within one iteration.

42

Use-Case Model: Drawing System Sequence Diagrams

43

System Behavior and UML Sequence Diagrams

- It is useful to investigate and define the behavior of the software as a “black box”.
- System behavior is a description of *what the system does* (without an explanation of how it does it).
- Use cases describe how external actors interact with the software system. During this interaction, an actor generates events.
- A request event initiates an operation upon the system.

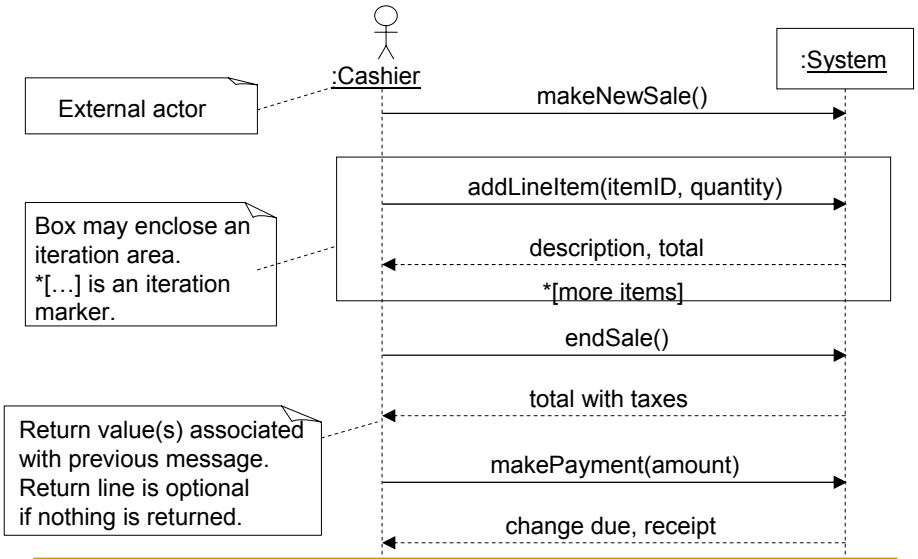
44

System Behavior and System Sequence Diagrams (SSDs)

- A sequence diagram is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.
- All systems are treated as a black box; the diagram places emphasis on events that cross the system boundary from actors to systems.

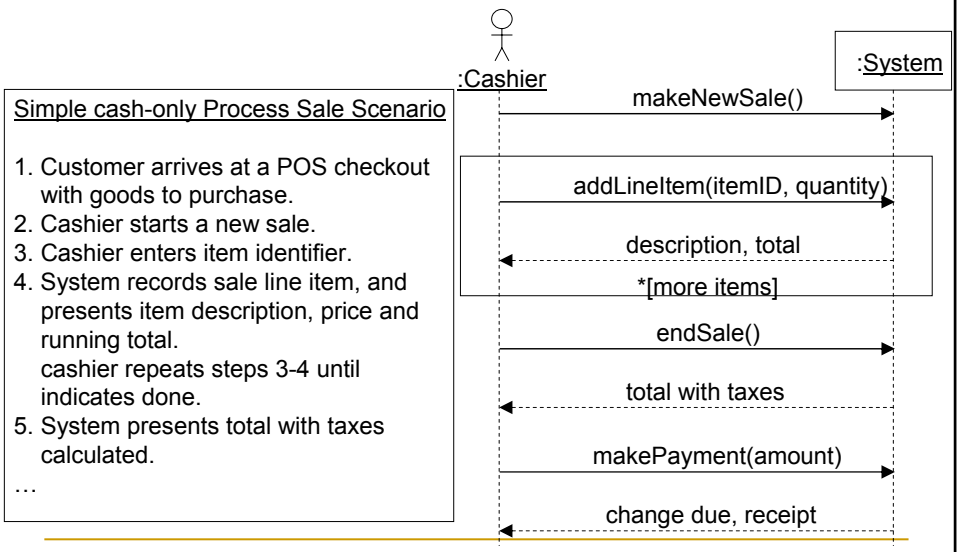
45

SSDs for Process Sale Scenario



46

SSD and Use Cases



47

Naming System Events and Operations

- The set of all required system operations is determined by identifying the system events.
  - makeNewSale()
  - addLineItem(itemID, quantity)
  - endSale()
  - makePayment(amount)

48

Domain Model: Visualizing Concepts

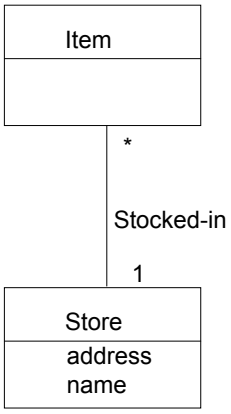
49

Domain Models

- A Domain Model illustrates meaningful concepts in a problem domain.
- It is a representation of real-world things, not software components.
- It is a set of static structure diagrams; no operations are defined.
- It may show:
  - concepts
  - associations between concepts
  - attributes of concepts

50

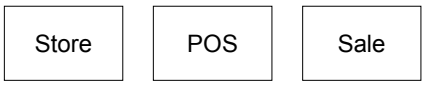
Domain Models



- A Domain Model is a description of things in the real world.
- A Domain Model is not a description of the software design.
- A concept is an idea, thing, or object.

51

Conceptual Classes in the Sale Domain



Partial Domain Model.

- A central distinction between object-oriented and structured analysis: division by concepts (objects) rather than division by functions.

52

Strategies to Identify Conceptual Classes

- Use a conceptual class category list.
  - Make a list of candidate concepts.
- Use noun phrase identification.
  - Identify noun (and noun phrases) in textual descriptions of the problem domain, and consider them as concepts or attributes.
  - Use Cases are excellent description to draw for this analysis.

53

Use a conceptual class category list

Concept Category	Example
Physical or tangible objects	POS
Specifications, designs, or descriptions of things	ProductSpecification
Places	Store
Transactions	Sale, Payment
Transaction line items	SalesLineItem
Roles of people	Cashier
Containers of other things	Store, Bin
(See complete list in Larman 2nd. ed., pp. 134-135)	

54

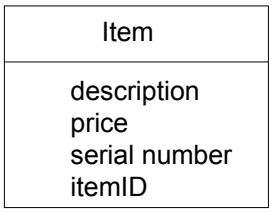
Finding Conceptual Classes with Noun Phrase Identification

1. This use case begins when a **Customer** arrives at a **POS checkout** with items to purchase.
2. The **Cashier** starts a new sale.
3. **Cashier** enters **item identifier**.

…

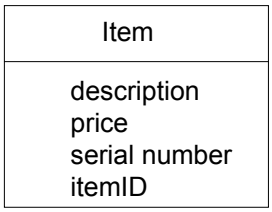
- The fully addressed Use Cases are an excellent description to draw for this analysis.
- Some of these noun phrases are candidate concepts; some may be attributes of concepts.
- A mechanical noun-to-concept mapping is not possible, as words in a natural language are (sometimes) ambiguous.

The Need for Specification or Description Conceptual Classes



- What is wrong with this picture?
- Consider the case where all items are sold, and thus deleted from the computer memory.
- How much does an item cost?

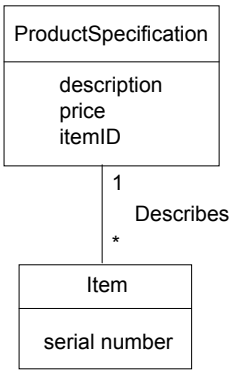
The Need for Specification or Description Conceptual Classes



- The memory of the item's price was attached to inventoried instances, which were deleted.
- Notice also that in this model there is duplicated data (description, price, itemID).

57

The Need for Specification or Description Conceptual Classes



- Add a specification or description concept when:
  - Deleting instances of things they describe results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
  - It reduces redundant or duplicated information.

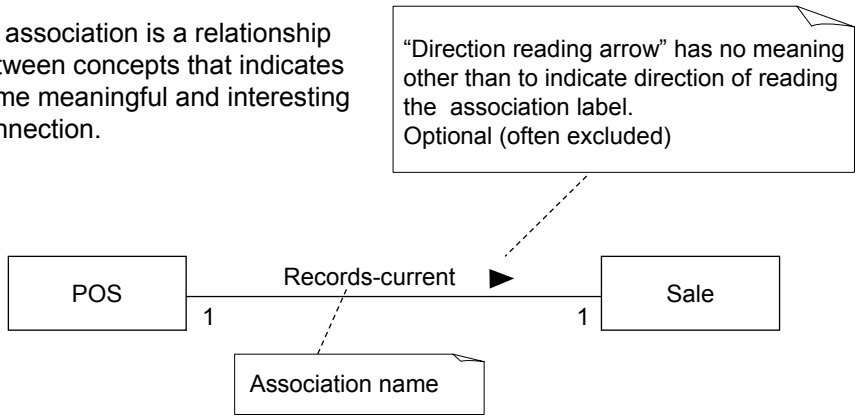
58

The NextGen POS (partial) Domain Model

59

Adding Associations

An association is a relationship between concepts that indicates some meaningful and interesting connection.



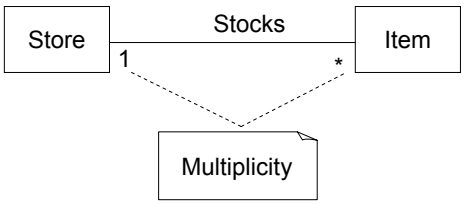
60

Finding Associations –Common Associations List

Category	Examples
A is a physical part of B*	Drawer - POS
A is a logical part of B	SalesLineItem - Sale
A is physically contained in/on B	POS - Store
A is logically contained in B	ItemDescription - Catalog
A is a description of B	ItemDescription - Item
A is a line item of a transaction or report B	SalesLineItem - Sale
A is known/logged/recorded/ captured in B	Sale - POS
A is a member of B	Cashier - Store
...	
(See complete list in Larman 2nd. ed., pp. 156-157)	**** High-priority associations***

61

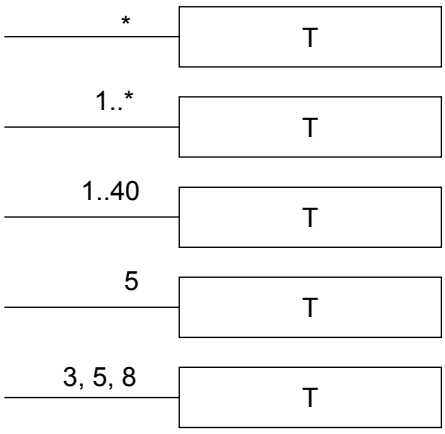
Multiplicity



- Multiplicity defines how many instances of a type A can be associated with one instance of a type B, at a particular moment in time.
- For example, a single instance of a Store can be associated with "many" (zero or more) Item instances.

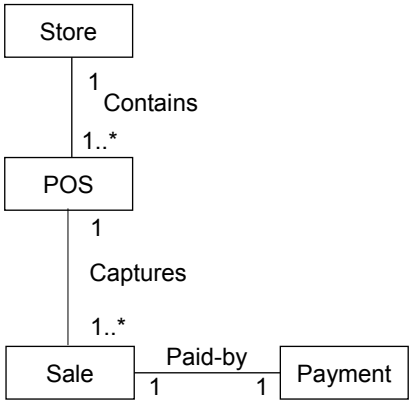
62

Multiplicity



63

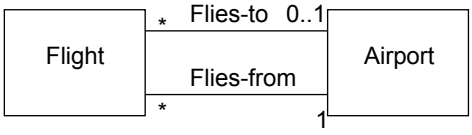
Naming Associations



- Name an association based on a `TypeName-VerbPhrase-TypeName` format.
- Association names should start with a capital letter.
- A verb phrase should be constructed with hyphens.
- The default direction to read an association name is left to right, or top to bottom.

64

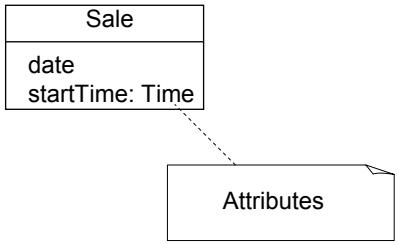
Multiple Associations Between Two Types



- It is not uncommon to have multiple associations between two types.
- In the example, not every flight is guaranteed to land at an airport.

65

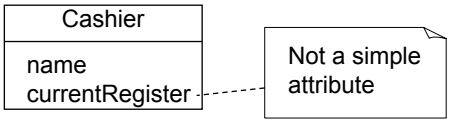
Adding Attributes

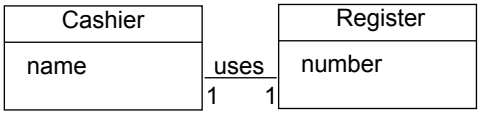


- An attribute is a logical data value of an object.
- Include the following attributes: those for which the requirements suggest or imply a need to remember information.
- For example, a Sales receipt normally includes a date and time.
- The Sale concept would need a date and time attribute.

66

Valid Attribute Types

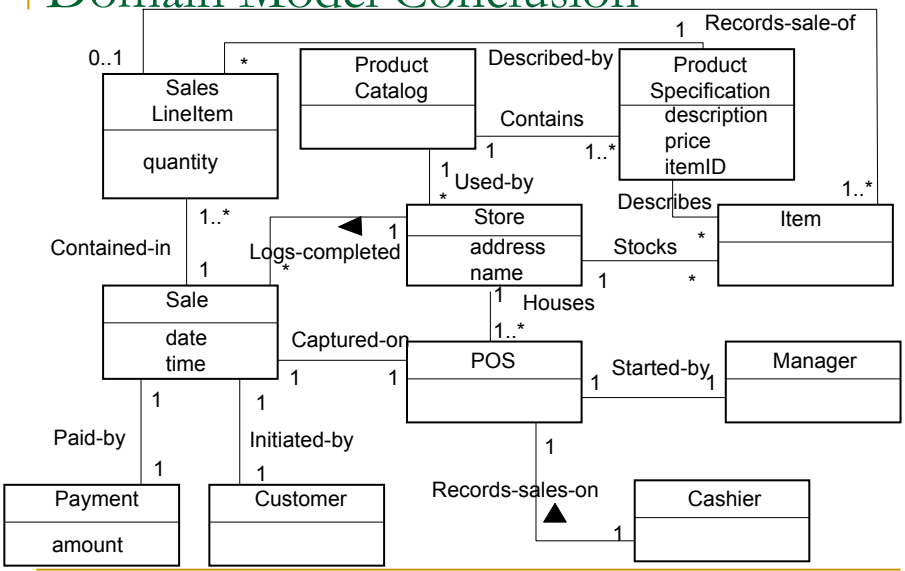




- Keep attributes simple.
- The type of an attribute should not normally be a complex domain concept, such as Sale or Airport.
- Attributes in a Domain Model should preferably be
  - Pure data values: Boolean, Date, Number, String, …
  - Simple attributes: color, phone number, zip code, universal product code (UPC), ...

67

Domain Model Conclusion



68

Use-Case Model: Adding Detail with Operation Contracts

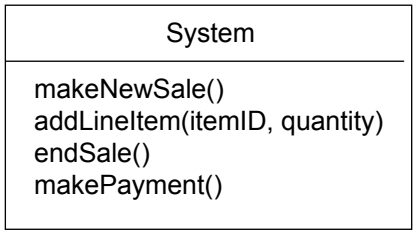
69

Contracts

- Contracts are documents that describe system behavior.
- Contracts may be defined for **system operations**.
  - Operations that the system (as a black box) offers in its public interface to handle incoming system events.
- The entire set of system operations across all use cases, defines the public system interface.

70

System Operations and the System Interface



- In the UML the system as a whole can be represented as a class.
- Contracts are written for each system operation to describe its behavior.

71

Example Contract: addLineItem

Contract CO2: addLineItem

Operation: addLineItem (itemID: ItemID, quantity: integer)
Cross References: Use Cases: Process Sale.
Pre-conditions: There is a sale underway.
Post-conditions:

- A SalesLineItem instance *sli* was created. (instance creation)
- *sli* was associated with the Sale. (association formed)
- *sli.quantity* was set to quantity. (attribute modification)
- *sli* was associated with a ProductSpecification, based on itemID match (association formed)

72

Pre- and Postconditions

- Preconditions are assumptions about the state of the system before execution of the operation.
- A postcondition is an assumption that refers to the state of the system after completion of the operation.
  - The postconditions are not actions to be performed during the operation.
  - Describe changes in the state of the objects in the Domain Model (instances created, associations are being formed or broken, and attributes are changed)

73

addLineItem postconditions

- Instance Creation and Deletion
- After the itemID and quantity of an item have been entered by the cashier, what new objects should have been created?
  - A SalesLineItem instance *sli* was created.

74

addLineItem postconditions

- Attribute Modification
- After the itemID and quantity of an item have been entered by the cashier, what attributes of new or existing objects should have been modified?
- sli.quantity was set to quantity (attribute modification).

75

addLineItem postconditions

- Associations Formed and Broken
- After the itemID and quantity of an item have been entered by the cashier, what associations between new or existing objects should have been formed or broken?
  - sli was associated with the current Sale (association formed).
  - sli was associated with a ProductSpecification, based on itemID match (association formed).

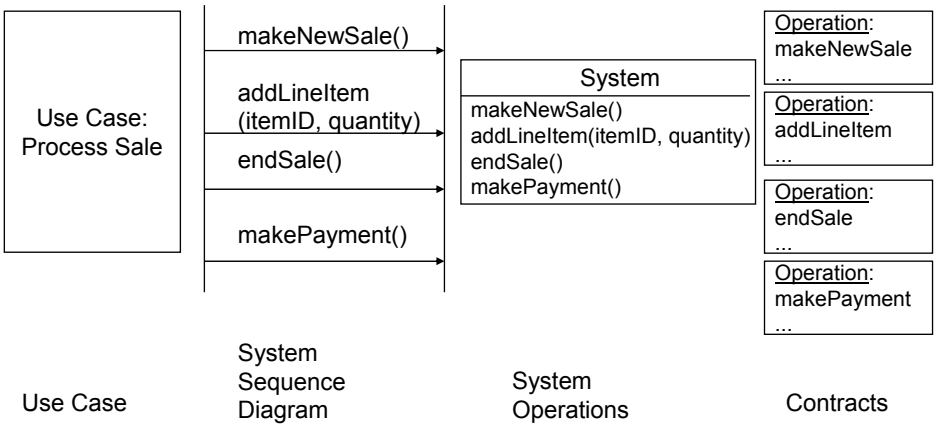
76

Writing Contracts leads to Domain Model Updates

- It is also common to discover the need to record new concepts, attributes or associations in the Domain Model.

77

Guidelines for Contracts

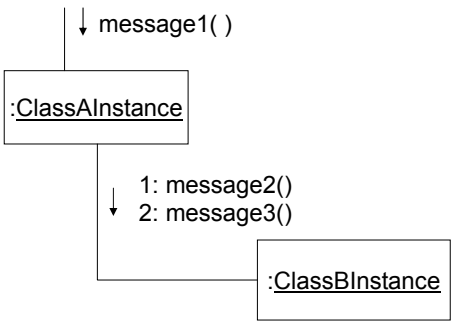


78

Interaction Diagram Notation

79

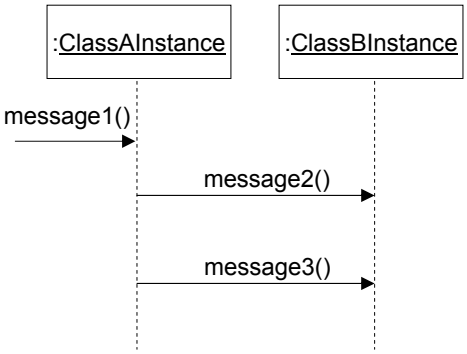
Introduction



- *Interaction diagrams* illustrate how objects interact via messages.
- *Collaboration diagrams* illustrate object interactions in a graph or network format.

80

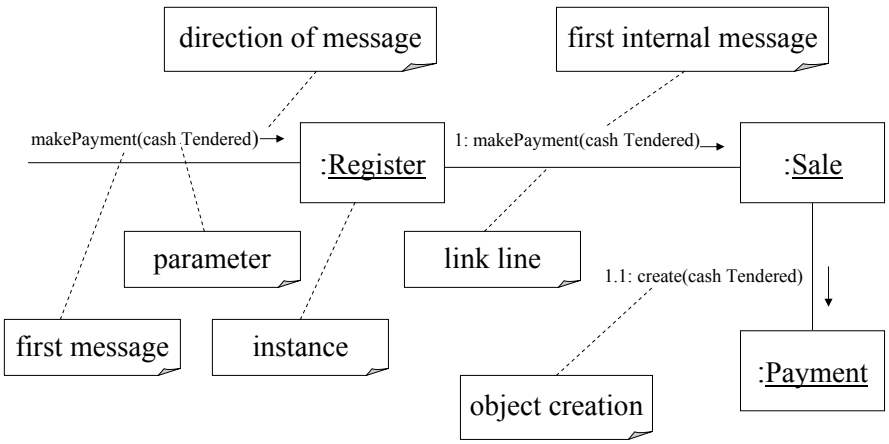
Introduction



- *Sequence diagrams* illustrate interactions in a kind of fence format.
- Set of all operation contracts defines system behavior.
- We will create an interaction diagram for each operation contract.

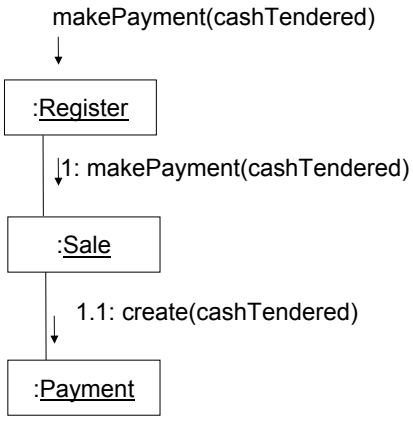
81

Example Collaboration Diagram: makePayment



82

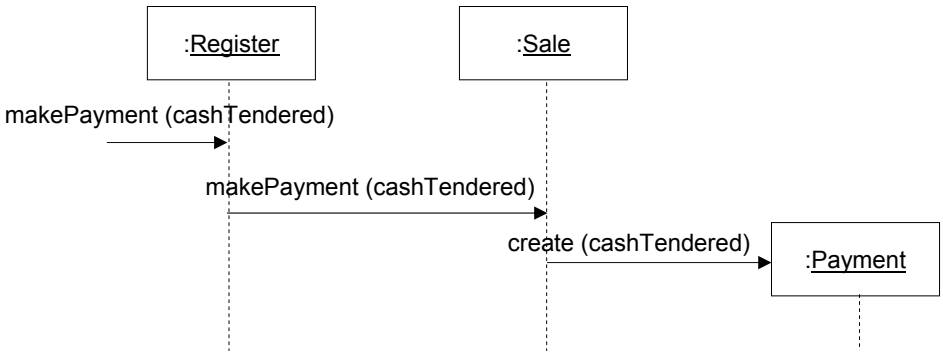
How to Read the makePayment Collaboration Diagram



1. The message makePayment is sent to an instance of Register. The sender is not identified.
2. The Register instance sends the makePayment message to a Sale instance.
3. The Sale instance creates an instance of a Payment.

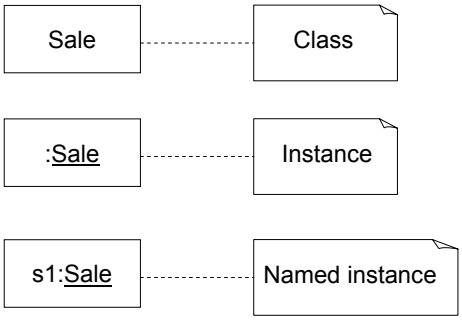
83

Example Sequence Diagram: makePayment



84

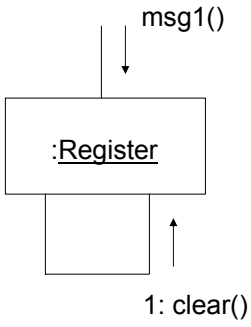
Illustrating Classes and Instances



- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined. Additionally a class name should be preceded by a colon.
- An instance name can be used to uniquely identify the instance.

85

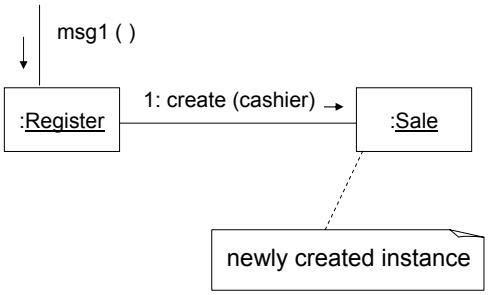
Messages to "self" or "this"



- A message can be sent from an object to itself.
- This is illustrated by a link to itself, with messages flowing along the link.

86

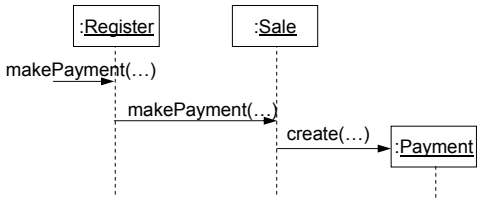
Creation of Instances



- The language independent creation message is create, being sent to the instance being created.
- The create message may include parameters, indicating passing of initial values.

87

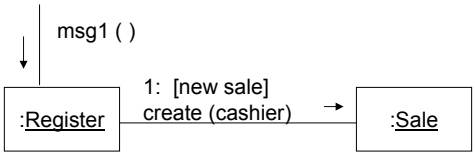
Creation of Instances



- An object lifeline shows the extend of the life of an object in the diagram.
- Note that newly created objects are placed at their creation height.

88

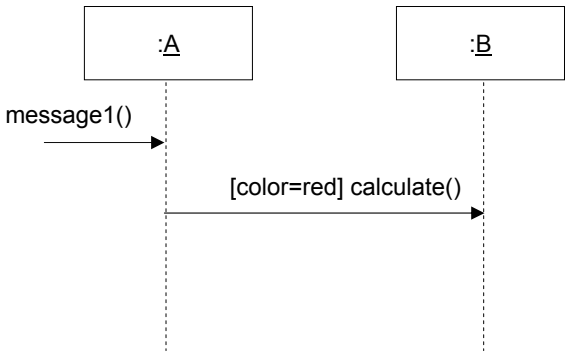
Conditional Messages



- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.
- The message is sent only if the clause evaluates to true.

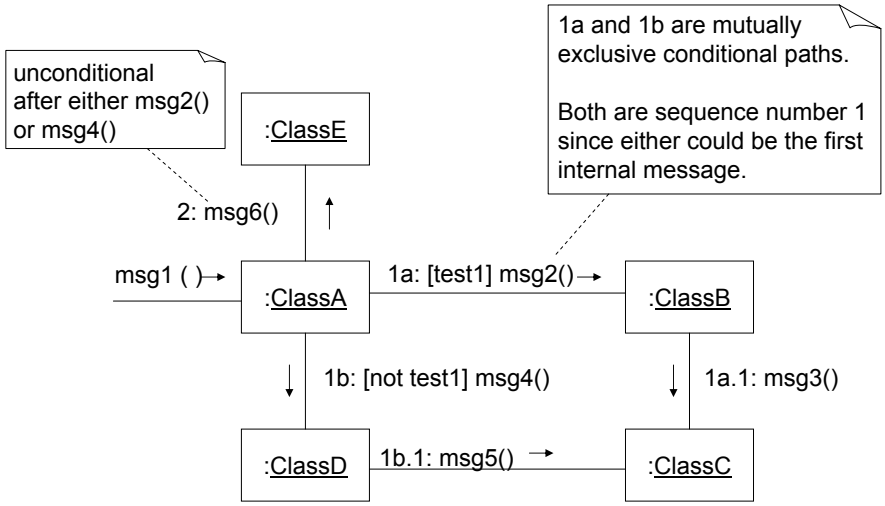
89

Conditional Messages



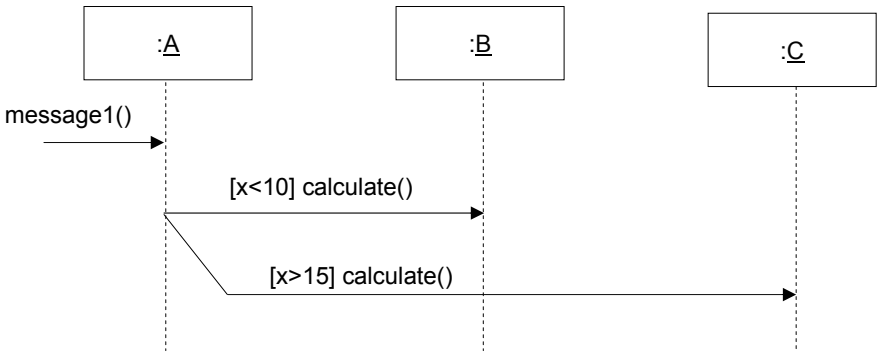
90

Mutually Exclusive Conditional Paths



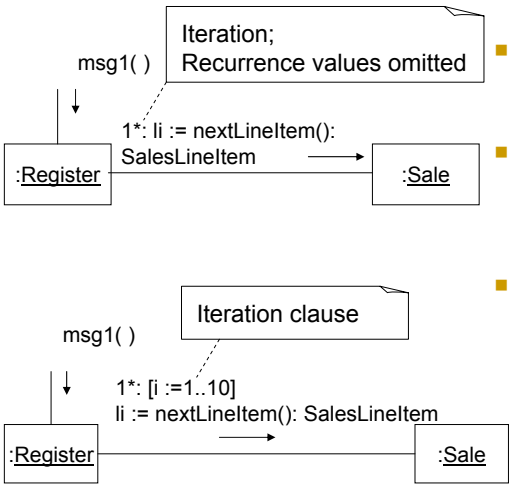
91

Mutually Exclusive Conditional Messages



92

Iteration or Looping



- Iteration is indicated by following the sequence number with a star *
- This expresses that the message is being sent repeatedly, in a loop, to the receiver.
- It is also possible to include an iteration clause indicating the recurrence values.

93

GRASP*: Designing Objects with Responsibilities

* Genenal Responsibility Assignment Software Patterns

94

Responsibilities and Methods

- The focus of object design is to identify classes and objects, decide what methods belong where and how these objects should interact.
- Responsibilities are related to the obligations of an object in terms of its behavior.
- Two types of responsibilities:
  - Doing:
    - Doing something itself (e.g. creating an object, doing a calculation)
    - Initiating action in other objects.
    - Controlling and coordinating activities in other objects.
  - Knowing:
    - Knowing about private encapsulated data.
    - Knowing about related objects.
    - Knowing about things it can derive or calculate.

Responsibilities and Methods

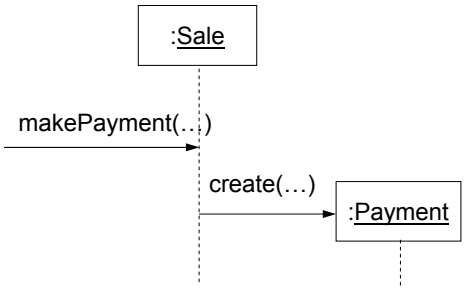
- Responsibilities are assigned to classes during object design. For example, we may declare the following:
  - *"a Sale is responsible for creating SalesLineItems"* (doing)
  - *"a Sale is responsible for knowing its total"* (knowing)
- Responsibilities related to "knowing" are often inferable from the Domain Model (because of the attributes and associations it illustrates)

Responsibilities and Methods

- The translation of responsibilities into classes and methods is influenced by the granularity of responsibility.
  - For example, *"provide access to relational databases"* may involve dozens of classes and hundreds of methods, whereas *"create a Sale"* may involve only one or few methods.
- A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- Methods either act alone, or collaborate with other methods and objects.

97

Responsibilities and Interaction Diagrams



- Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams.
- Sale objects have been given the responsibility to create Payments, handled with the makePayment method.

98

Patterns

- We will emphasize principles (expressed in patterns) to guide choices in where to assign responsibilities.
- A pattern is a named description of a problem and a solution that can be applied to new contexts; it provides advice in how to apply it in varying circumstances. For example,
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfil it.

99

Information Expert (or Expert)

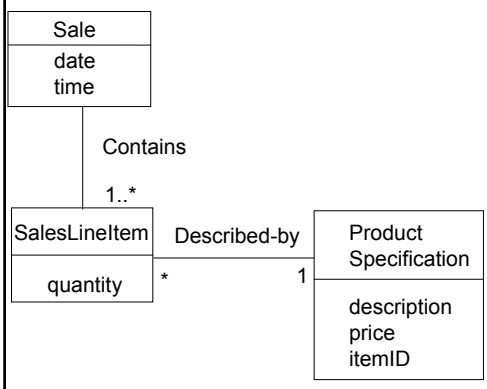
- Problem: what is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.
- In the NextGen POS application, who should be responsible for knowing the grand total of a sale?
- By Information Expert we should look for that class that has the information needed to determine the total.

100

Information Expert (or Expert)

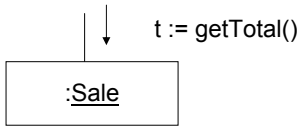
- Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?
- A: Both. Assume there is no or minimal Design Model. Look to the Domain Model for information experts.

Information Expert (or Expert)



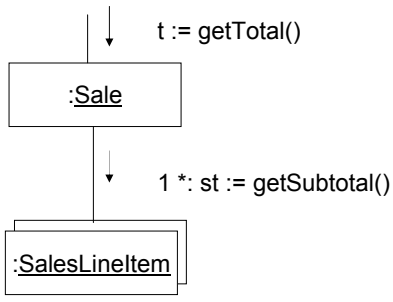
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.

Information Expert (or Expert)



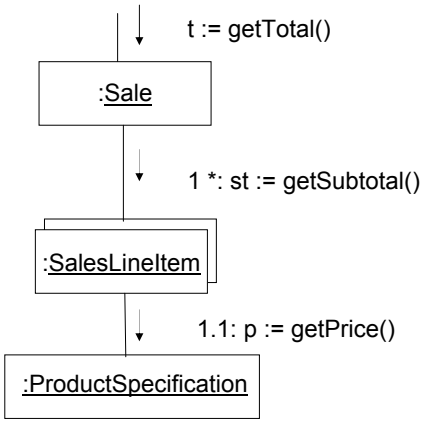
- This is a partial interaction diagram.

Information Expert (or Expert)



- What information is needed to determine the line item subtotal?
  - quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.

Information Expert (or Expert)



- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.

105

Information Expert (or Expert)

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

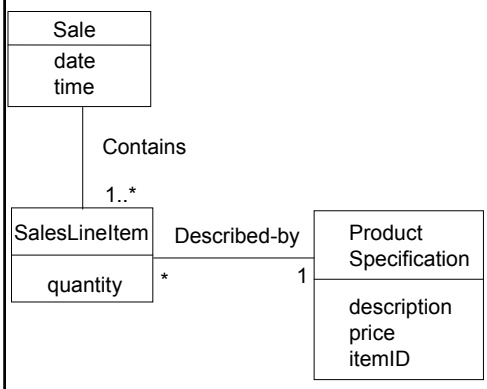
- To fulfil the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial experts" who will collaborate in the task.

106

Creator

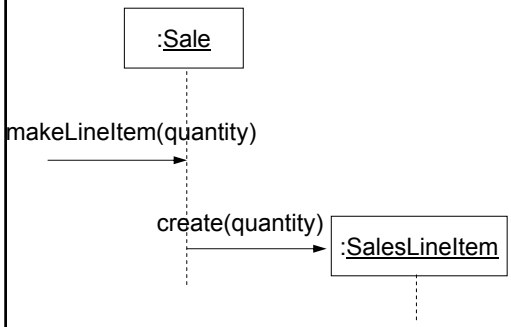
- Problem: Who should be responsible for creating a new instance of some class?
- Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  1. B *aggregates* A objects.
  2. B *contains* A objects.
  3. B *records* instances of A objects.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

Creator



- In the POS application, who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate.

Creator



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

109

Low Coupling

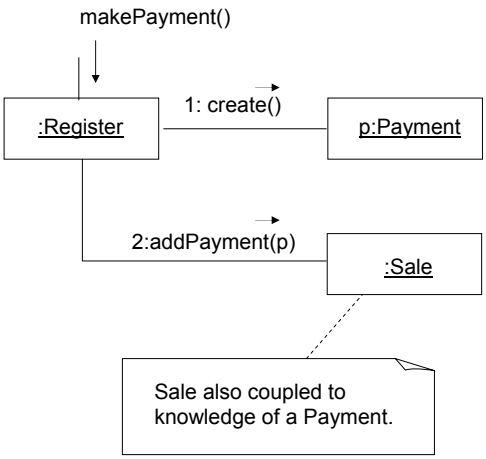
- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

110

Low Coupling

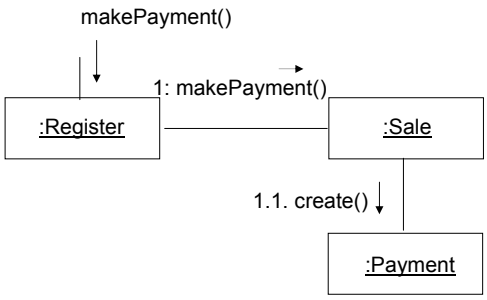
- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.

Low Coupling



- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling



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

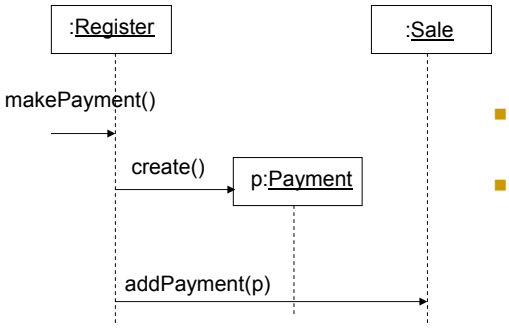
Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.

High Cohesion

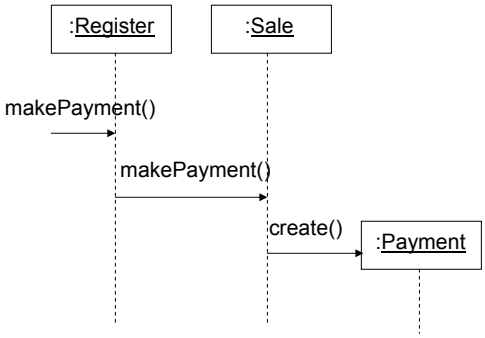
- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

High Cohesion



- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.

High Cohesion



- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.

117

High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion

1. Very low cohesion: class responsible for many things in many different areas.
   - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.
2. Low cohesion: class responsible for complex task in a functional area.
   - e.g.: a class responsible for interacting with a relational database.

118

High Cohesion

3. High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.
   - e.g.: a class responsible for one section of interfacing with a data base.
- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.

Controller

- Problem: Who should be responsible for handling an input system event?
- Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of he following choices:
  - Represents the overall system.
  - Represents a use case scenario.
  - A Controller is a non-user interface object that defines the method for the system operation. Note that windows, applets, etc. typically receive events and delegate them to a controller.

Use Case Realizations

Use Case Realizations

System
makeNewSale() addLineItem(itemID, quantity) endSale() makePayment()

Contract CO1: makeNewSale

Operation: makeNewSale ()

Cross References: Use Cases: Process Sale.

Pre-conditions: none.

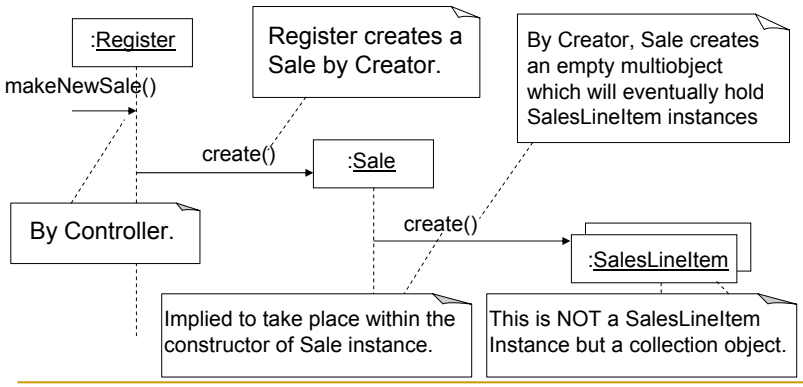
Post-conditions:

- A Sale instance *s* was created. (instance creation)
- s was associated with the Register (association formed)
- Attributes of s were initialized

- A use-case realization describes how a use case is realized in terms of collaborating objects.
- UML interaction diagrams are used to illustrate use case realizations.
- Recall Process Sale: from main scenario we identified a number of system events (operations)
- Each system event was then described by a contract.

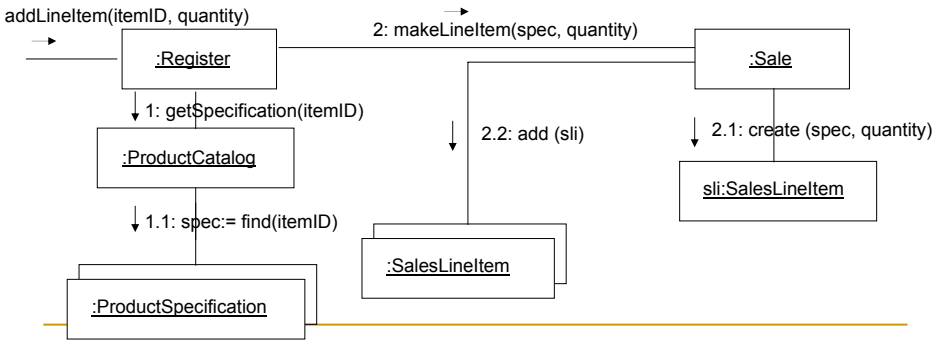
Object Design: makeNewSale

- We work through the postcondition state changes and design message interactions to satisfy the requirements.

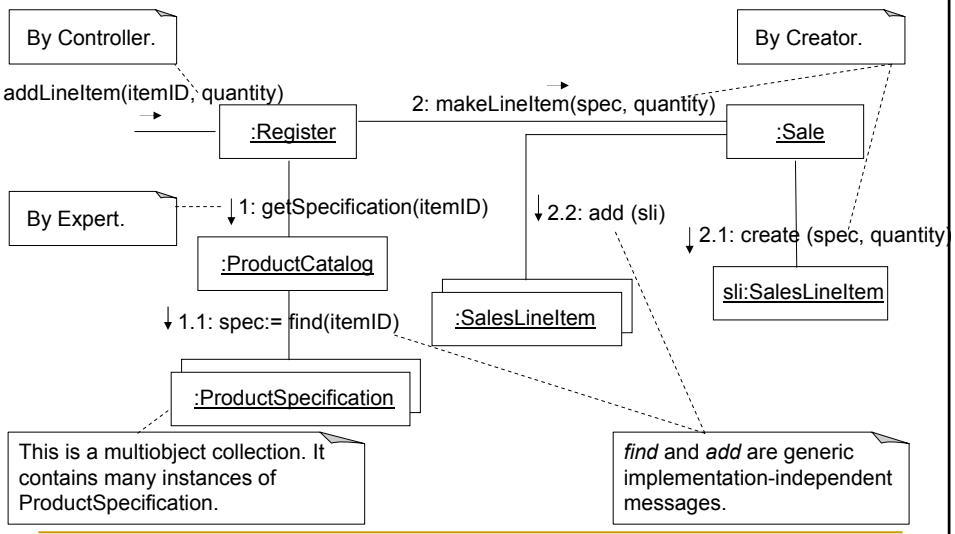


Object Design: addLineItem

- **Contract CO2: addLineItem**
- **…**
- **Post-conditions:**
  - A SalesLineItem instance *sli* was created. (instance creation)
  - *sli* was associated with the Sale. (association formed)
  - *sli.quantity* was set to quantity. (attribute modification)
  - *sli* was associated with a ProductSpecification, based on itemID match (association formed)



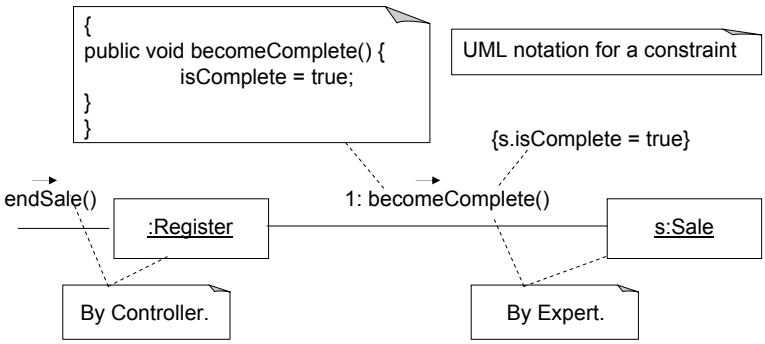
Object Design: addLineItem



125

Object Design: endSale

- **Contract CO3: endSale**
- **…**
- **Post-conditions:**
  - Sale.isComplete became true (attribute modification)



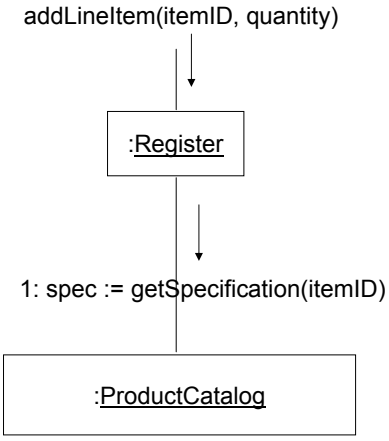
126

Design Model: Determining Visibility

Introduction

- Visibility: the ability of an object to “see” or have reference to another object.
- For a sender object to send a message to a receiver object, the receiver must be visible to the sender – the sender must have some kind of reference (or pointer) to the receiver object.

Visibility Between Objects



- The getSpecification message sent from a Register to a ProductCatalog, implies that the ProductCatalog instance is visible to the Register instance.

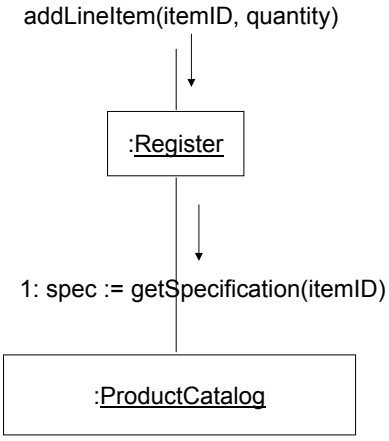
129

Visibility

- How do we determine whether one resource (such as an instance) is within the scope of another?
- Visibility can be achieved from object A to object B in four common ways:
  - Attribute visibility: B is an attribute of A.
  - Parameter visibility: B is a parameter of a method of A.
  - Local visibility: B is a (non-parameter) local object in a method of A.
  - Global visibility: B is in some way globally visible.

130

Visibility



- The Regiser must have visibility to the ProductCatalog.
- A typical visibility solution is that a reference to the ProductCatalog instance is maintained as an attribute of the Register.

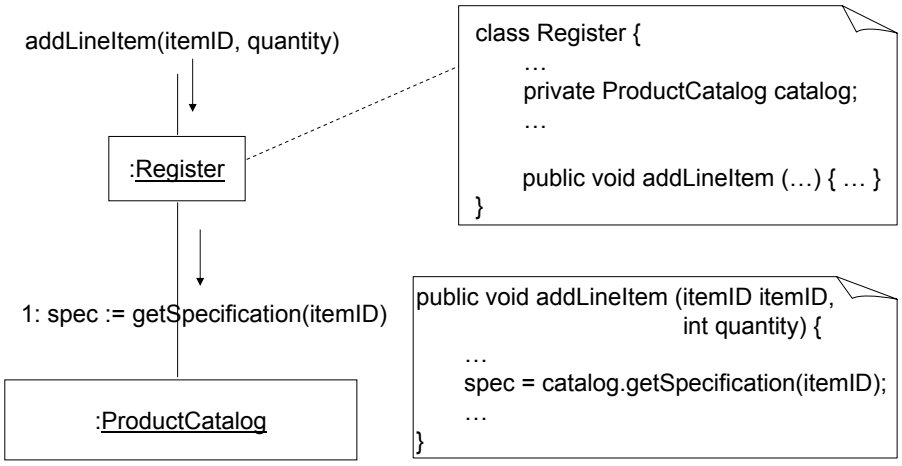
131

Attribute Visibility

- Attribute visibility from A to B exists when B is an attribute of A.
- It is a relatively permanent visibility, because it persists as long as A and B exist.
- In the addLineItem collaboration diagram, Register needs to send message getSpecification message to a ProductCatalog. Thus, visibility from Register to ProductCatalog is required.

132

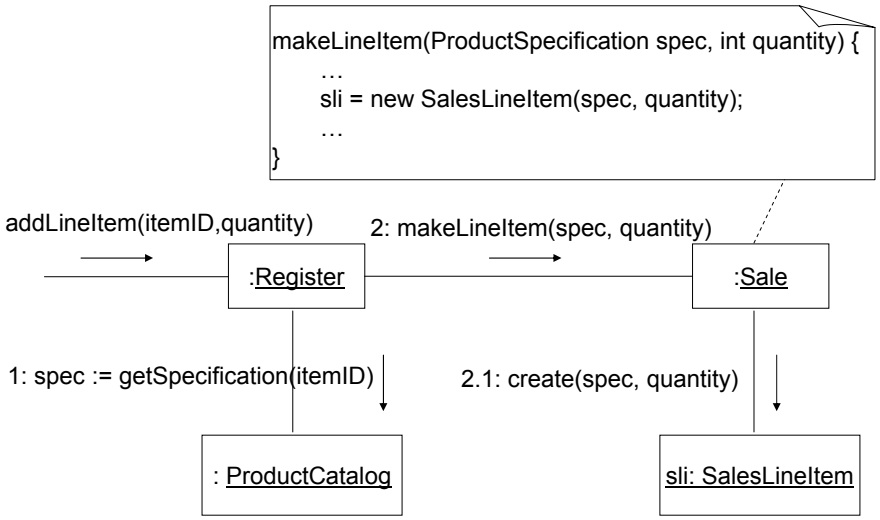
Attribute Visibility



Parameter Visibility

- Parameter visibility from A to B exists when B is passed as a parameter to a method of A.
- It is a relatively temporary visibility, because it persists only within the scope of the method.
- When the makeLineItem message is sent to a Sale instance, a ProductSpecification instance is passed as a parameter.

Parameter Visibility



135

Parameter Visibility

```
// constructor
SalesLineItem(ProductSpecification spec,
                    int quantity) {
    …
    // parameter to attribute visibility
    productSpec = spec;
    …
}
```

- When Sale crates a new SalesLineItem, it passes a ProductSpecification to its constructor.
- We can assign ProductSpecification to an attribute in the constructor, thus transforming parameter visibility to attribute visibility.

136

Local Visibility

- Locally declared visibility from A to B exists when B is declared as a local object within a method of A.
- It is relatively temporary visibility because it persists only within the scope of the method. Can be achieved as follows:
  1. Create a new local instance and assign it to a local variable.
  2. Assign the return object from a method invocation to a local variable.

```
addLineItem(itemID, quantity) {
    …
    ProductSpecification spec = catalog.getSpecification(itemID);
    ...
}
```

137

Global Visibility

- Global visibility from A to B exists when B is global to A.
- It is a relatively permanent visibility because it persists as long as A and B exist.
- One way to achieve this is to assign an instance to a global variable (possible in C++ but not in Java).

138

Design Model: Creating Design Class Diagrams

139

When to create DCDs

- Once the interaction diagrams have been completed it is possible to identify the specification for the software classes and interfaces.
- A class diagram differs from a Domain Model by showing software entities rather than real-world concepts.
- The UML has notation to define design details in static structure, or class diagrams.

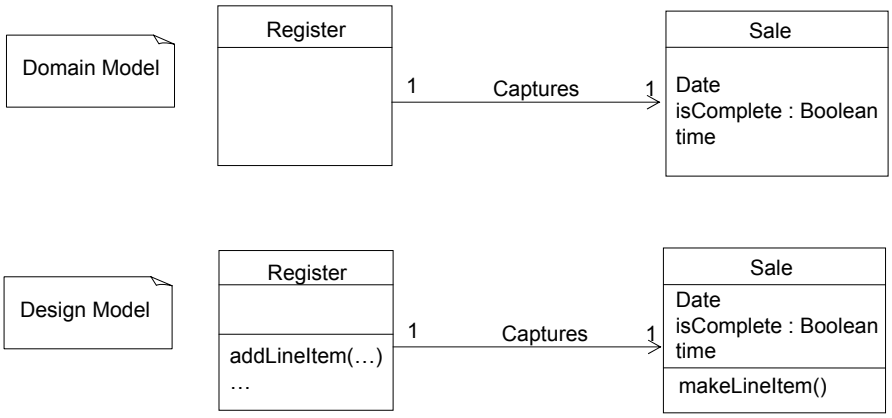
140

DCD and UP Terminology

- Typical information in a DCD includes:
  - Classes, associations and attributes
  - Interfaces (with operations and constants)
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies
- The DCD depends upon the Domain Model and interaction diagrams.
- The UP defines a Design Model which includes interaction and class diagrams.

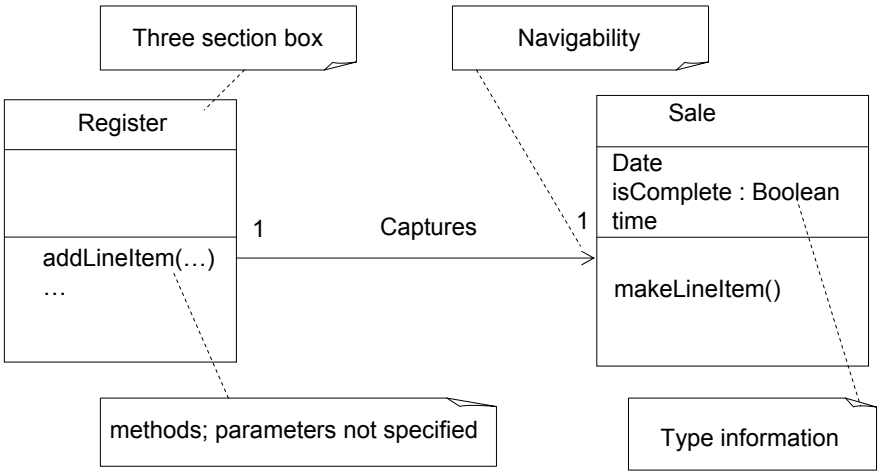
141

Domain Model vs. Design Model Classes



142

An Example DCD



143

Creating a NextGen POS DCD

- Identify all the classes participating in the software solution. Do this by analyzing the interaction diagrams. Draw them in a class diagram.
- Duplicate the attributes from the associated concepts in the Domain Model.

Register

ProductCatalog
quantity

ProductSpecification
description price itemID

Payment
amount

Store
address name

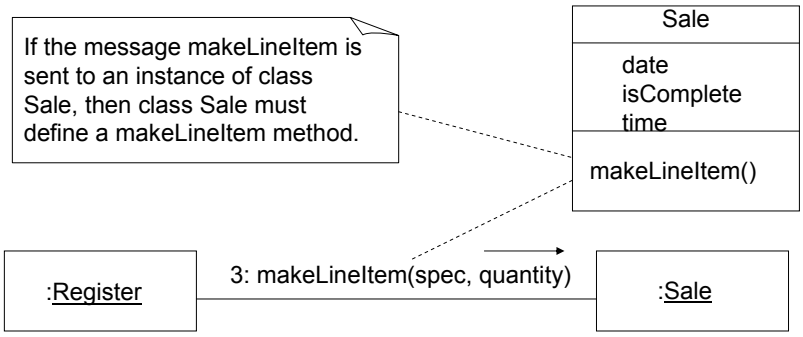
Sale
date isComplete time

SalesLineItem
quantity

144

Creating a NextGen POS DCD

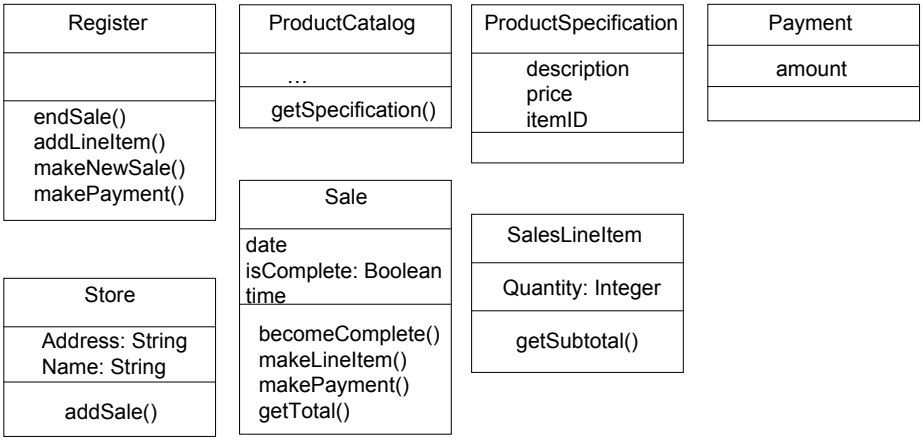
- Add method names by analyzing the interaction diagrams.
  - The methods for each class can be identified by analyzing the interaction diagrams.



145

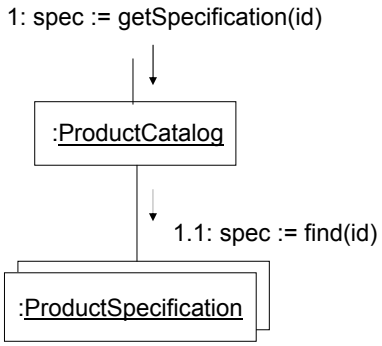
Creating a NextGen POS DCD

- Add type information to the attributes and methods.



146

Method Names -Multiobjects

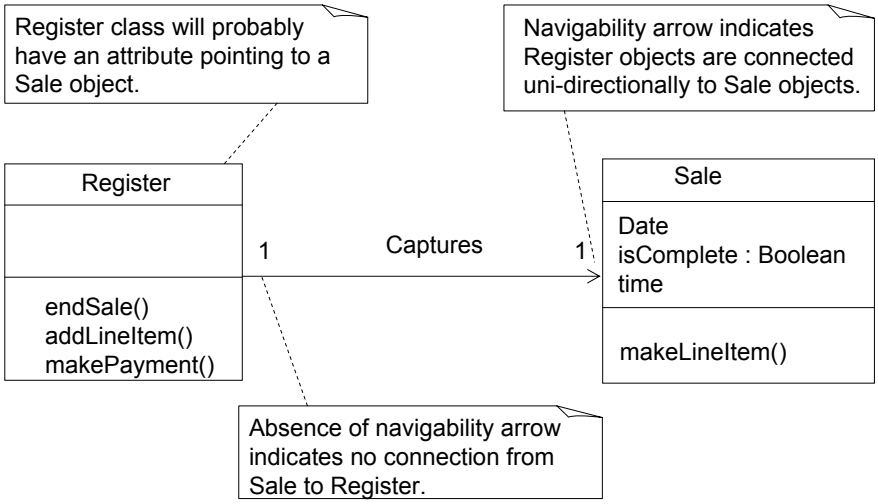


- The find message to the multiobject should be interpreted as a message to the container/ collection object.
- The find method is not part of he ProductSpecification class.

Associations, Navigability, and Dependency Relationships

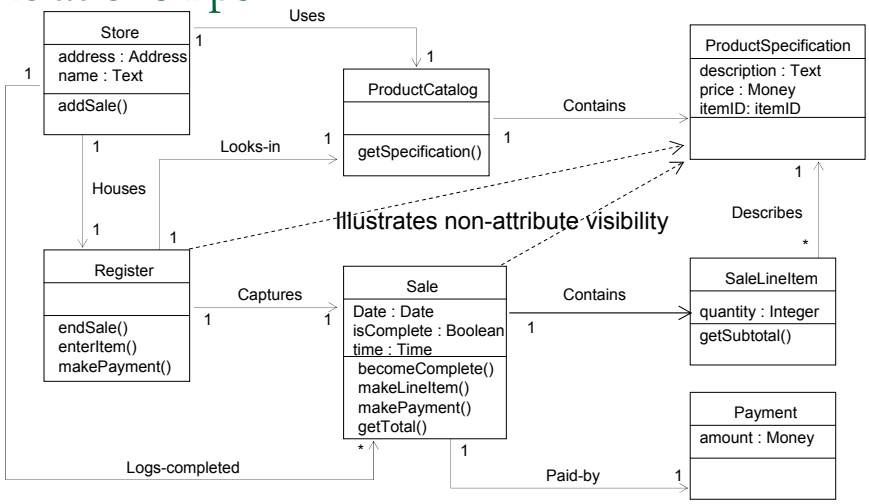
- Add the associations necessary to support the required attribute visibility.
  - Each end of an association is called a role.
- Navigability is a property of the role implying visibility of the source to target class.
  - Attribute visibility is implied.
  - Add navigability arrows to the associations to indicate the direction of attribute visibility where applicable.
  - Common situations suggesting a need to define an association with navigability from A to B:
    - A sends a message to B.
    - A creates an instance of B.
    - A needs to maintain a connection to B
- Add dependency relationship lines to indicate non-attribute visibility.

Creating a NextGen POS DCD



149

Adding Navigability and Dependency Relationships



150

Implementation Model: Mapping Designs to Code

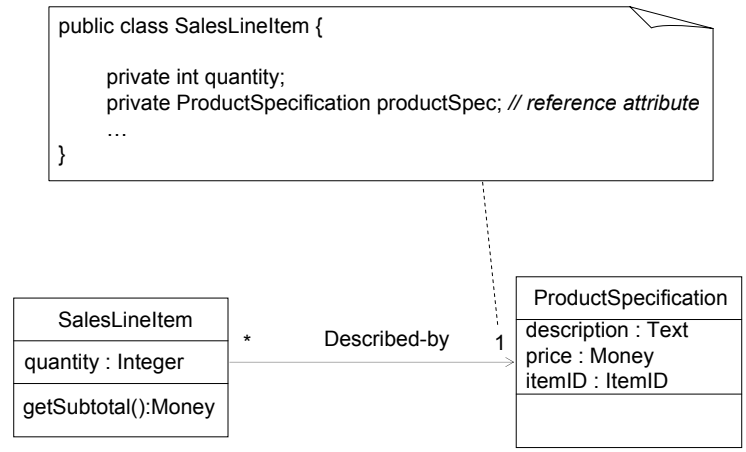
151

Defining a Class with Methods and Simple Attributes

```
public class SalesLineItem {

    private int quantity;

    public SalesLineItem(ProductSpecification spec, int, qty) {...}
    public Money getSubtotal() {…}
    …
}
```

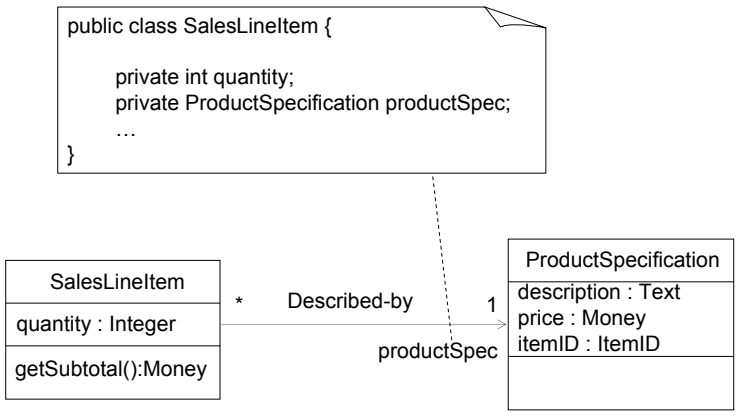
152

Adding Reference Attributes



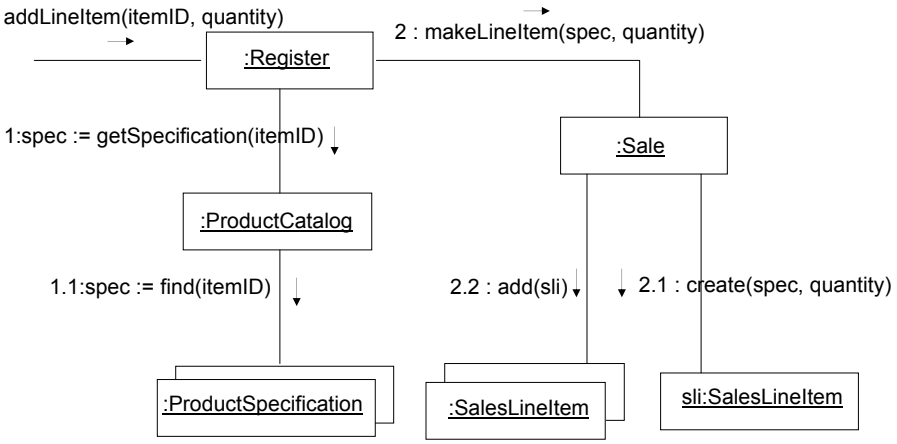
153

Reference Attributes and Role Names



154

Creating Methods from Interaction Diagrams



155

The Register – addLineItem method

- The addLineItem collaboration diagram will be used to illustrate the Java definition of the addLineItem method.
- In which class does addLineItem belong to?
- The addLineItem message is sent to a Register instance, therefore the addLineItem method is defined in class Register.

  public void addLineItem(itemID itemID, int quantity);
- Message 1. A getSpecification message is sent to the productCatalog to retrieve a productSpecification

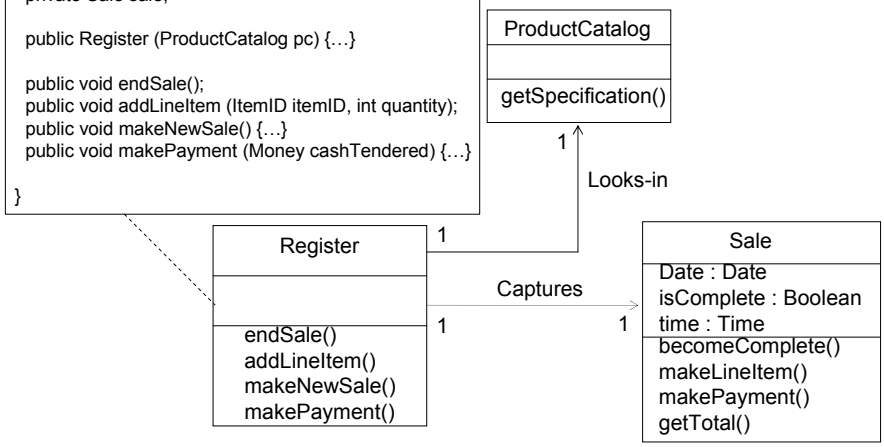
  productSpecification spec = catalog.getSpecification(itemID);
- Message 2: The makeLineItem message is sent to the Sale.

  sale.makeLineItem(spec, quantity);

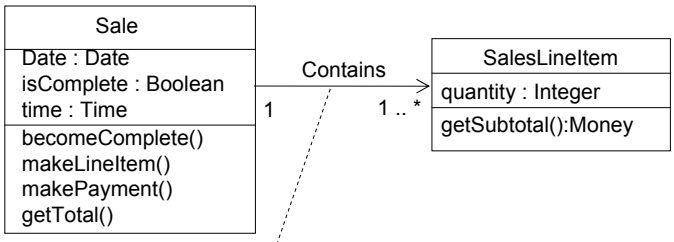
156

A Skeletal definition of Register Class

```
public class Register {

  private productCatalog catalog;
  private Sale sale;

  public Register (ProductCatalog pc) {…}

  public void endSale();
  public void addLineItem (ItemID itemID, int quantity);
  public void makeNewSale() {…}
  public void makePayment (Money cashTendered) {…}

}
```



Container/Collection Classes in Code




```
public class Sale {
  . . .

  private Vector lineItems;
}
```

A container class is necessary to maintain attribute visibility to all the SalesLineItem instances.

Vector is an example of a dynamic data structure.

Iteration 2 and its Requirements

From Iteration 1 to 2

- By the end of Elaboration 1 the software has been built and tested.
- The idea of UP is to do early, and continuous verification of quality and correctness.
  - Early feedback guides the developers to adapt and improve the system.
  - Stakeholders get to see early visible progress with the system.
- Requirements for the next iteration are chosen.
  - Most of them will have been identified during Inception.

From Iteration 1 to 2

- Use cases will be revisited, and more scenarios will be implemented.
- It is common to work on varying scenarios or features of the same use case over several iterations and gradually extend the system.
- Short, simple use cases may be completely implemented within one iteration.

161

Iteration 3 and its Requirements

162

Iteration 3 Emphasis

- Inception and Iteration 1 explored a variety of fundamental issues in requirements analysis and OOA/D.
- Iteration 2 emphasizes object design.
- Iteration 3 explores a wide variety of analysis and design.

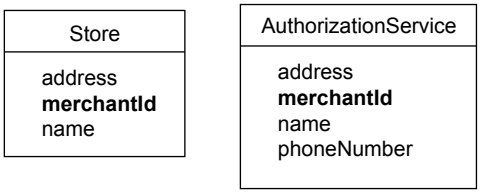
Refining the Domain Model

Association Classes

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service requires the inclusion of the merchant ID that identifies the store to the service.
- Consider a store that has a different merchant ID for each service. (e.g. Id for Visa is XXX, Id for MC is YYY, etc.)

165

Association Classes

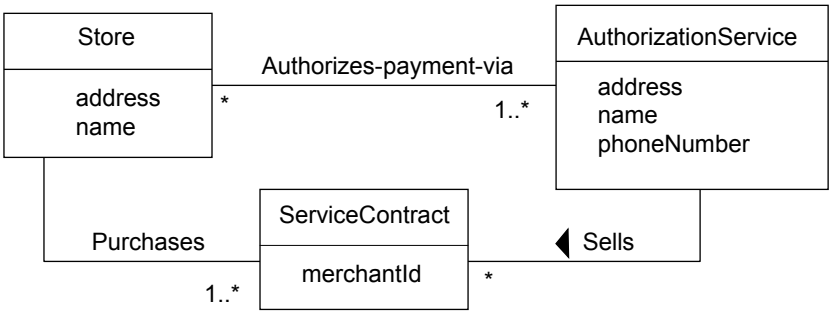


- Where in the conceptual model should the merchant ID attribute reside?
- Placing the merchantId in the Store is incorrect, because a Store may have more than one value for merchantId.
- The same is true with placing it in the AuthorizationService

166

Association Classes

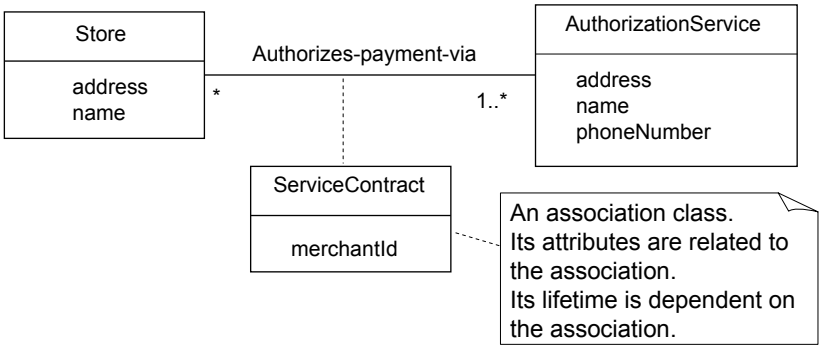
- In a conceptual model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another type that is associated with C.



167

Association Classes

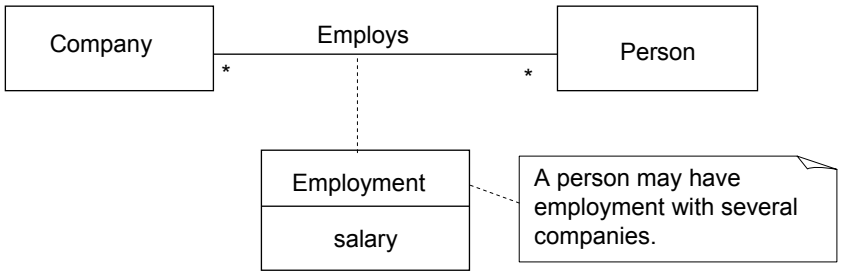
- The merchantId is an attribute related to the association between the Store and AuthorizationService; it depends on their relationship.
- ServiceContract may then be modeled as an association class.



168

Guidelines for Association Classes

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- There is a many-to-many association between two concepts.
- The presence of a many-to-many association between two concepts is a clue that a useful associative type should exist in the background somewhere.



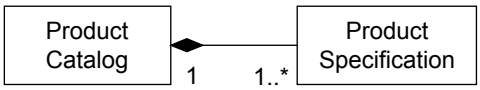
169

Aggregation and Composition

- Aggregation is a kind of association used to model whole-part relationships.
- The whole is generally called the composite; parts have no standard name (part or component is common).

170

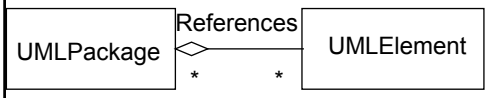
Composite Aggregation - Filled diamond



- Composite aggregation or composition means that the multiplicity at the composite end may be at most one signified with a filled diamond).
- ProductCatalog is an aggregate of ProductSpecification.

171

Shared Aggregation - Hollow diamond



- Shared aggregation means that the multiplicity at the composite end may be more than one (signified with a hollow diamond).
- It implies that the part may be in many composite instances.

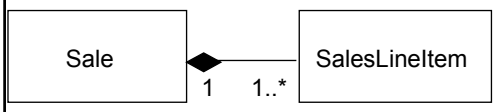
172

How to identify Aggregation

- The 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.

173

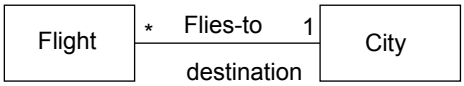
Aggregation in the POS Domain Model



- In the POS domain, the SalesLineItem may be considered a part of a composite Sale; in general, transaction line items are viewed as part of an aggregate transaction.
- In addition, there is a create-delete dependency of the line items on the Sale - their lifetime is bound within the lifetime of the Sale.

174

Association Role Names



- Each end of an association is a role, which has various properties such as name, and multiplicity.
- The role name identifies an end of an association and ideally describes the role played by objects in the association.
- An explicit role name is not required - it is useful only when the role of the object is not clear.

175

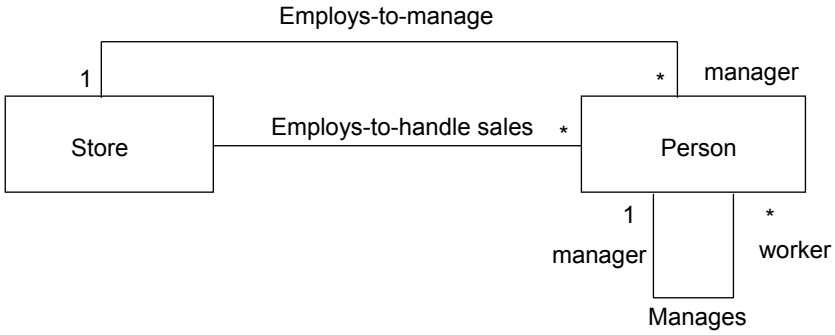
Roles as concepts versus roles in associations

- In a conceptual model, a real-world role may be modeled in a number of ways, such as
  - a discrete concept, or
  - expressed as a role in an association.

176

"Roles in associations"

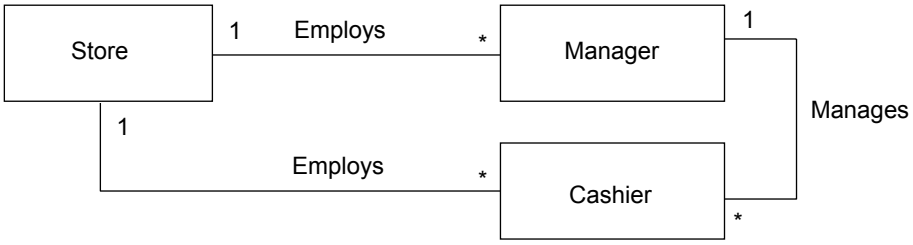
- A relatively accurate way to express the notion that the same instance of a person takes on multiple (and dynamically changing) roles in various environments.



177

"Roles as concepts"

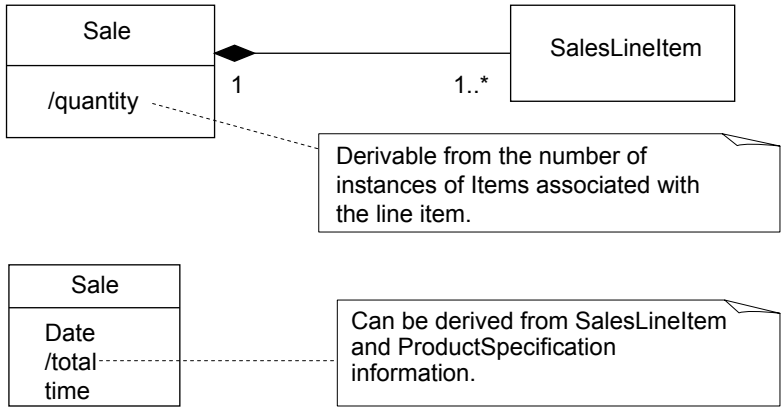
- Modeling roles as concepts provides ease and flexibility in adding unique attributes, associations, and additional semantics.



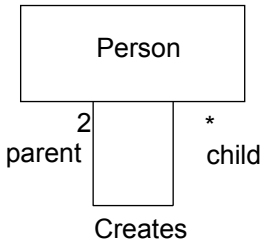
178

Derived Elements

- A derived element can be determined from others.



Recursive or Reflexive Associations



- A concept may have an association to itself; this is known as a recursive association or reflective association.

Modeling Behavior in Statechart Diagrams

181

Introduction

- A state diagram (also state transition diagram) illustrates the events and the states of things: use cases, people, transactions, objects, etc.

182

Events, States and Transitions

- An event is a trigger, or occurrence.
  - e.g. a telephone receiver is taken off the hook.
- A state is the condition of an entity (object) at a moment in time - the time between events.
  - e.g. a telephone is in the state of being idle after the receiver is placed on the hook and until it is taken off the hook.

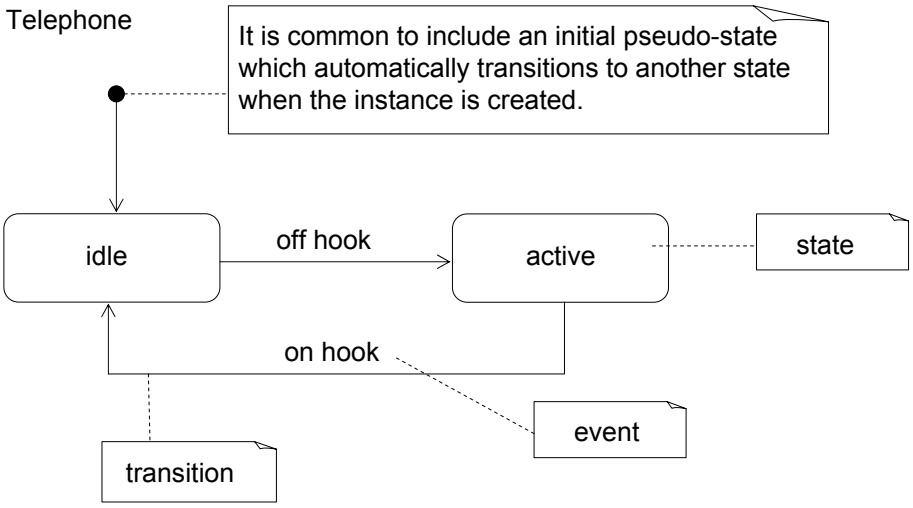
183

Events, States and Transitions

- A transition is a relationship between two states; It indicates that when an event occurs, the object moves from the prior state to the subsequent state.
  - e.g. when an event off the hook occurs, transition the telephone from the idle state to active state.

184

Statechart Diagrams



185

Statechart Diagrams

- A statechart diagram shows the life-cycle of an object; what events it experiences, its transitions and the states it is in between events.
- A state diagram need not illustrate every possible event; if an event arises that is not represented in the diagram, the event is ignored as far as the state diagram is concerned.
- Thus, we can create a state diagram which describes the life-cycle of an object at any simple or complex level of detail, depending on our needs.

186

Statechart Diagrams

- A statechart diagram may be applied to a variety of UML elements, including:
  - classes (conceptual or software)
  - use cases
- Since an entire system can be represented by a class, a statechart diagram may be used to represent it.
- Any UP element (Domain Model, Design model, etc.) may have deploy statecharts to model its dynamic behavior in response to events.

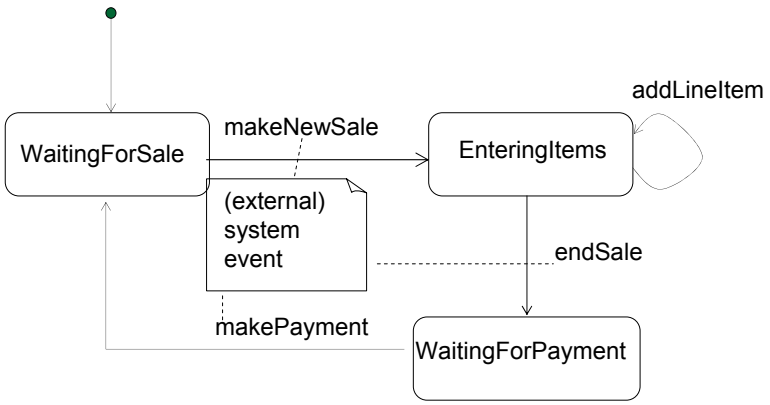
187

Use Case Statechart Diagrams

- A useful application of state diagrams is to describe the legal sequence of external system events that are recognized and handled by a system in the context of a use case.
- For example, During the *Process Sale* use case in the NextGen POS application, it is not legal to perform the *makeCreditPayment* operation until the *endSale* event has taken place.

188

Use Case Statechart Diagrams



189

Utility of Use Case State Diagrams

- A statechart diagram that illustrates the legal order of external events is particularly helpful for complex use cases.
- It is necessary to implement designs that ensure that no out-of-sequence events occur.
- Possible design solutions include:
  - Hard-coded conditional tests for out-of-order events.
  - Disabling widgets in active windows to disallow illegal events.
  - A state machine interpreter that runs a state table representing use case state diagram.

190

Classes that Benefit from Statechart Diagrams

1. State-independent and State Dependent Objects

- An entity is considered to be *state-independent* if it responds in the same manner to all events.
  - An object receives a message and the corresponding method always does the same thing. The object is state-independent with respect to that message.
  - If, for all events of interest, an object always reacts the same way, it is a state-independent object.
- An entity is considered to be state-dependent if it responds differently to events.
  - State diagrams are created for state-dependent entities with complex behavior.

191

Classes that Benefit from Statechart Diagrams

2. Common State-dependent Classes

- Use cases:
  - *Process Sale* use case reacts differently to the *endSale* event, depending on whether a sale is underway or not.
- Systems:
  - A type representing the overall behavior of the system.
- Windows:
  - Edit-paste action is only valid if there is something in the "clipboard" to paste.
- Transaction. Dependent on its current state within the overall life-cycle.
  - If a *Sale* received a *makeLineItem* message after the *endSale* event, it should raise an exception.

192

Illustrating External and Internal Events

- External event: caused by something outside a system boundary.
  - SSDs illustrate external events.
- Internal event: caused by something inside the system boundary.
  - Messages in interaction diagrams suggest internal events.

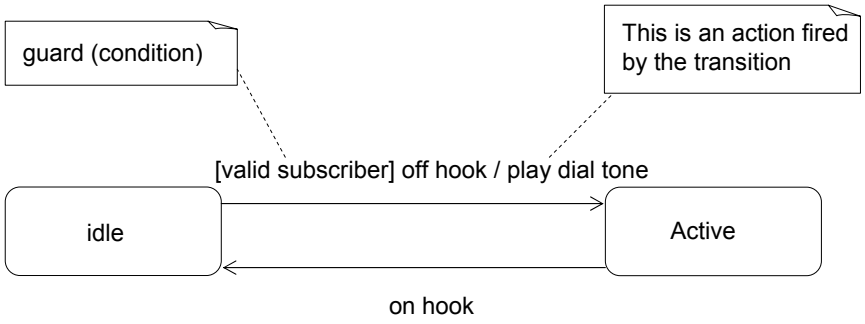
193

Illustrating External and Internal Events

- Temporal event: caused by the occurrence of a specific date and time or passage of time.
  - Driven by a real-time or simulated-time clock.
  - Suppose that after an *endSale*, a *makePayment* operation must occur within 5 minutes.
- Prefer using state diagrams to illustrate external and temporal events, and the reaction to them.

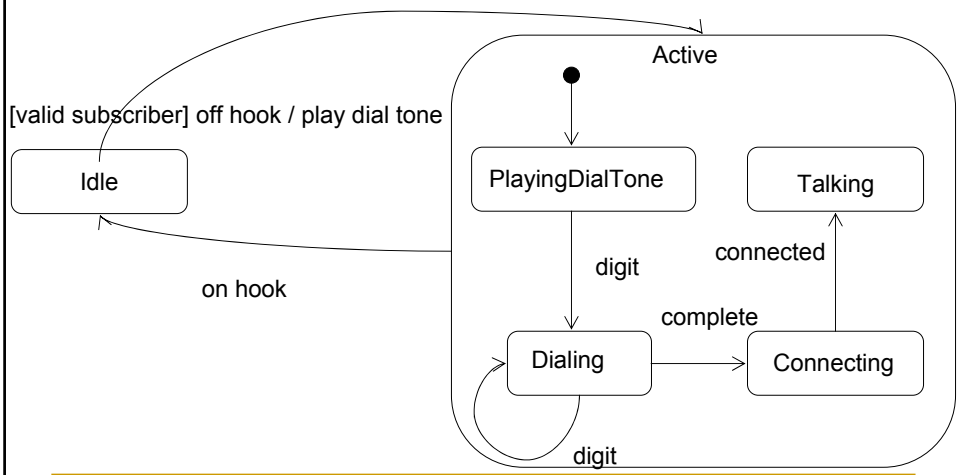
194

Additional Statechart Diagram Notation



195

Additional Statechart Diagram Notation



196

Additional Statechart Diagram Notation

- A state allows nesting to contain substates. A substate inherits the transitions of its superstate (the enclosing state).
  - Within the *Active* state, and no matter what substate the object is in, if the *on hook* event occurs, a transition to the *idle* state occurs.

197

Comments on Iterative Development and the UP

198

Additional UP Best Practices and Concepts

- The central idea behind the UP is short timeboxed iterative, adaptive development.
- Tackle high-risk and high-value issues in early iterations.
  - Leave easier work in later iterations.
  - This way the project does not “fail late”
  - Better to “fail early” if at all.

199

Additional UP Best Practices and Concepts

- Continuously engage users.
  - Development involves taking small steps and getting feedback.
  - “The majority of failed projects are correlated with lack of user engagement” [Standish, 1994]
- Early iterations focus on core architecture.
- Continuously verify quality.
  - No big surprises near the end of the project.

200

Additional UP Best Practices and Concepts

- Apply Use Cases.
  - Use Cases explore and record functional requirements.
  - Capture requirements, used for design, testing and writing end-user documentation.
- UML supports abstract (and visual) models of software.
- Carefully manage requirements.
  - "Poor requirements management is a common factor on troubled projects" [Standish, 1994]
- Control changes.
  - UP embraces change, not chaos.

The Construction and Transition Phases

- During Elaboration "most" requirements are understood.
  - Elaboration builds the risky and architecturally significant core of the system.
- Construction builds the remainder.
  - Development proceeds through a series of timeboxed iterations.
  - Testing during Construction is referred to as "alpha testing".

The Construction and Transition Phases

- Construction ends when the system is ready for operational deployment and all supporting materials are complete (user guides, training material etc.)
- The purpose of Transition phase is to put the system into production use.
  - Testing during Transition is referred to as “beta testing”

Other Practices

- Extreme Programming
  - Write a unit test before the code to be tested, and write tests for all classes.
  - http://www.extremeprogramming.org/
- SCRUM process pattern
  - Focuses on team behavior.
  - http://www.controlchaos.com

Motivations for Timeboxing an Iteration

- Development team has to focus to produce a tested executable partial system on-time.
- Timeboxing forces developers to prioritize work and risks, as well as identify tasks of high business and high technical value.
- Complete mini projects build team confidence and stakeholder confidence.

205

The Sequential "Waterfall" Lifecycle

- Clarify, record and commit to a set of final requirements.
- Design a system based on these requirements.
- Implement based on design.
- Integrate different modules.
- Evaluate and test for correctness and quality.

206

Some Problems with the Waterfall Lifecycle

- Tackling high-risk or difficult problems late.
  - In a waterfall lifecycle there is no attempt to identify and tackle riskiest issues first.
  - In an iterative development (like the UP) early iterations focus on driving down the risk.
- Requirements speculation and inflexibility.
  - The waterfall process assumes that requirements can be fully specified and then frozen in the first phase of the project.
    - Stakeholders want to see something concrete very early.
    - Market changes.
  - In iterative development requirements tend to stabilize after several iterations.

Some Problems with the Waterfall Lifecycle

- Design speculation and inflexibility.
- The waterfall lifecycle dictates that the architecture should be fully specified once the requirements are clarified and before implementation begins.
  - Since requirements usually change, the original design will not be reliable.
  - Lack of feedback on design until long after design decisions are made.