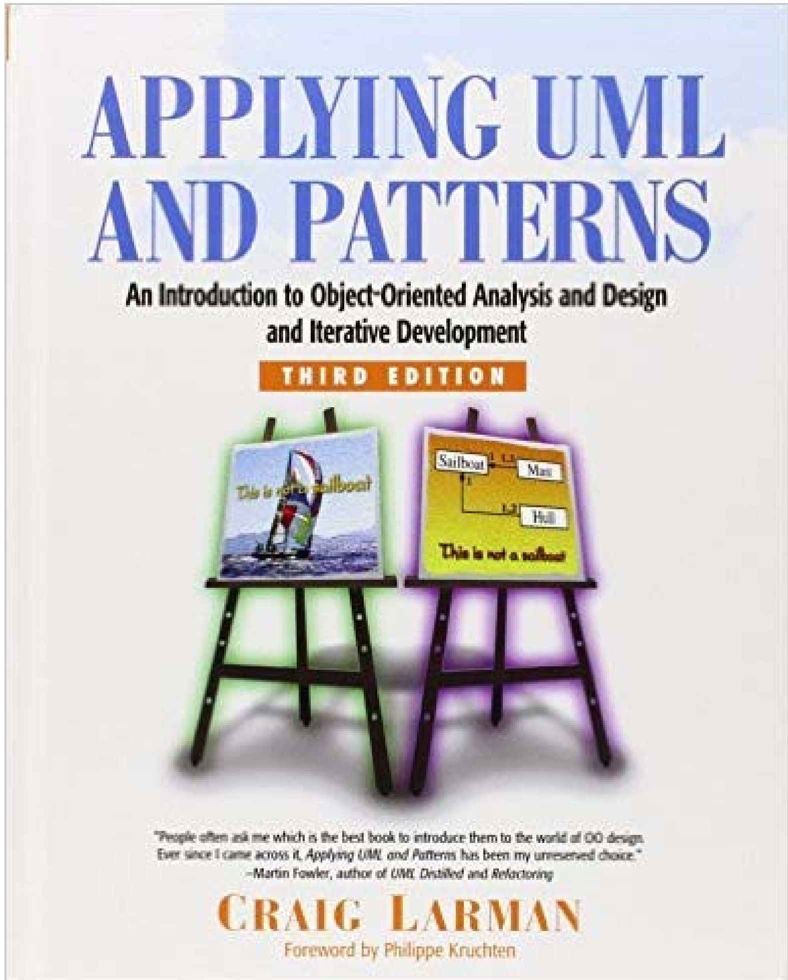
Object-Oriented Analysis and Design

JUNBEOM YOO

Dependable Software Lab.

Dependable Software Laboratory

Text and Contents



Contents at a Glance

OOAD

Design Patterns

Architecture Style

Architecture Description

Part 1: Introduction

- Chapter 1. Object-Oriented Analysis and Design
- Chapter 2. Iterative, Evolutionary, and Agile
- Chapter 3. Case Studies

Chapter 1.

Object-Oriented Analysis and Design

Object-Oriented Analysis and Design

- **Object-Oriented Analysis (OOA)**
  - Discover the domain concepts/objects (the objects of the problem domain)

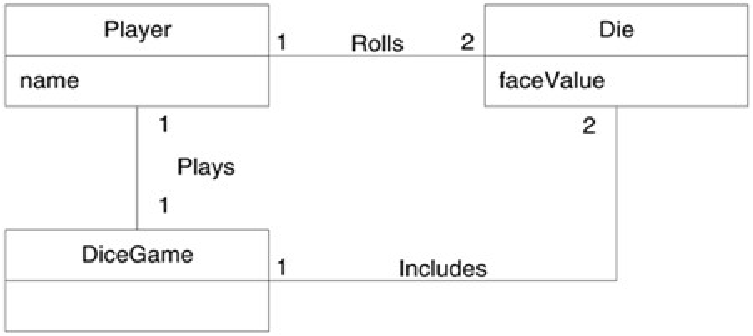
- **Object-Oriented Design (OOD)**
  - Define software objects (static)
  - Define how they collaborate to fulfill the requirements (dynamic)

An OOAD Example - Dice Game

Define use cases | Define domain model | Define interaction diagrams | Define design class diagrams

OOA

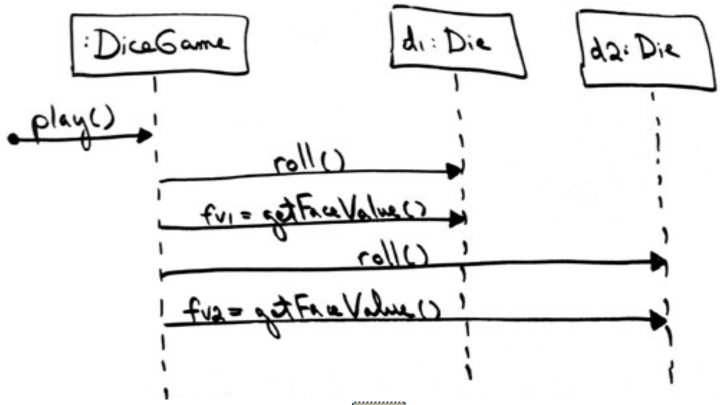
Use Case : Play a Dice Game
- Player requests to roll the dice.
- System presents results.
- If the dice's face value totals seven, player wins; otherwise, player loses.

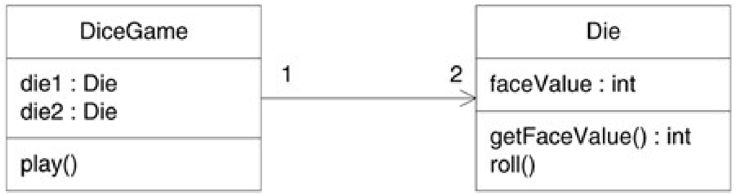


Domain Model

OOD

Interaction Diagram





Design Class Diagram

UML

- *"The Unified Modeling Language (UML) is a visual language for specifying, constructing and documenting the artifacts of systems."*

- **3 ways to apply (use) UML**
  - **Sketch**
    - Conceptual perspective
    - Informal and incomplete diagrams are created to explore difficult parts of the problem or solution space. → Intercommunication medium
  - **Blueprint**
    - Specification perspective
    - Relatively detailed design diagrams are used for code generation.
  - **Programming language**
    - Implementation perspective
    - Complete executable specification of a software system in UML
      - Executable code will be automatically generated.
      - Still under development in terms of theory, tool robustness and usability.

What the UML is Not?

- UML is **not** an Object-Oriented analysis and design process.
  - UML is not a systematic way to develop software systems.

- UML will **not** teach you an Object-Oriented way of thinking.
  - It will not tell you how to design object structures or behaviors.
  - It will not tell you whether your design is good or bad.

Chapter 2.
Iterative, Evolutionary, and Agile

Software Development Process and the UP

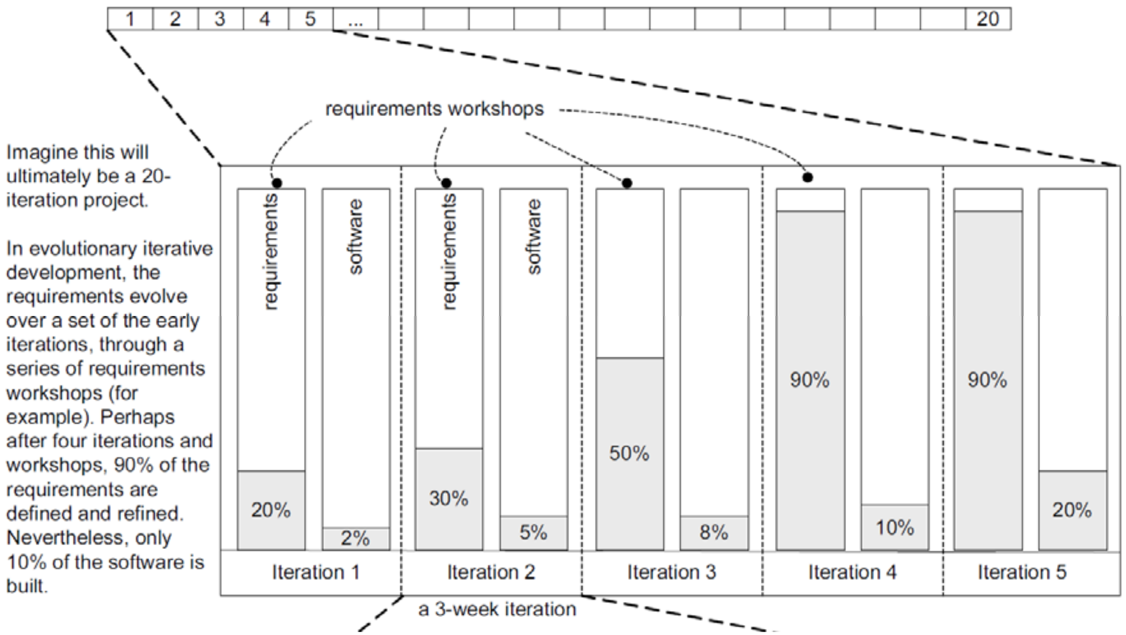
- **Software development process**
  - A **systematic approach** to building, deploying and possibly maintaining software

- **Unified Process (UP)**: a popular iterative software development process for building object-oriented systems
  - Iterative with fixed-length iterations (mini waterfalls of about **3 weeks**)
  - Inspired from Agile (*i.e.*, opposite from waterfall)
  - Flexible (can be combined with practices from other OO processes)
  - A de-facto industry standard for developing OO software

Risk-Driven and Client-Driven Iterative Planning

- The **UP** encourages a combination of **risk-driven** and **client-driven** **iterative planning**.
  - To identify and drive down the high risks, and
  - To build visible features that clients care most about.

- **Risk-driven** iterative development includes more specifically the practice of **architecture-centric** iterative development.
  - Early iterations focus on building, testing, and stabilizing the core architecture.



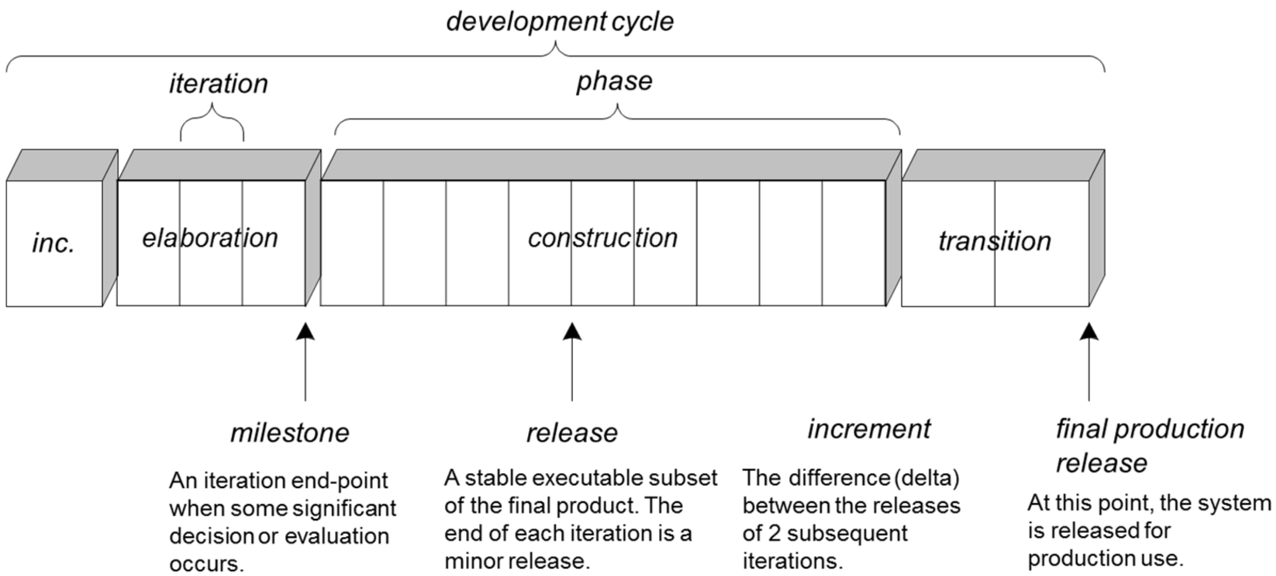
The UP Practices

- The central idea to **UP practices** :
  - A short timeboxed **iterative**, **evolutionary** and **adaptive** development

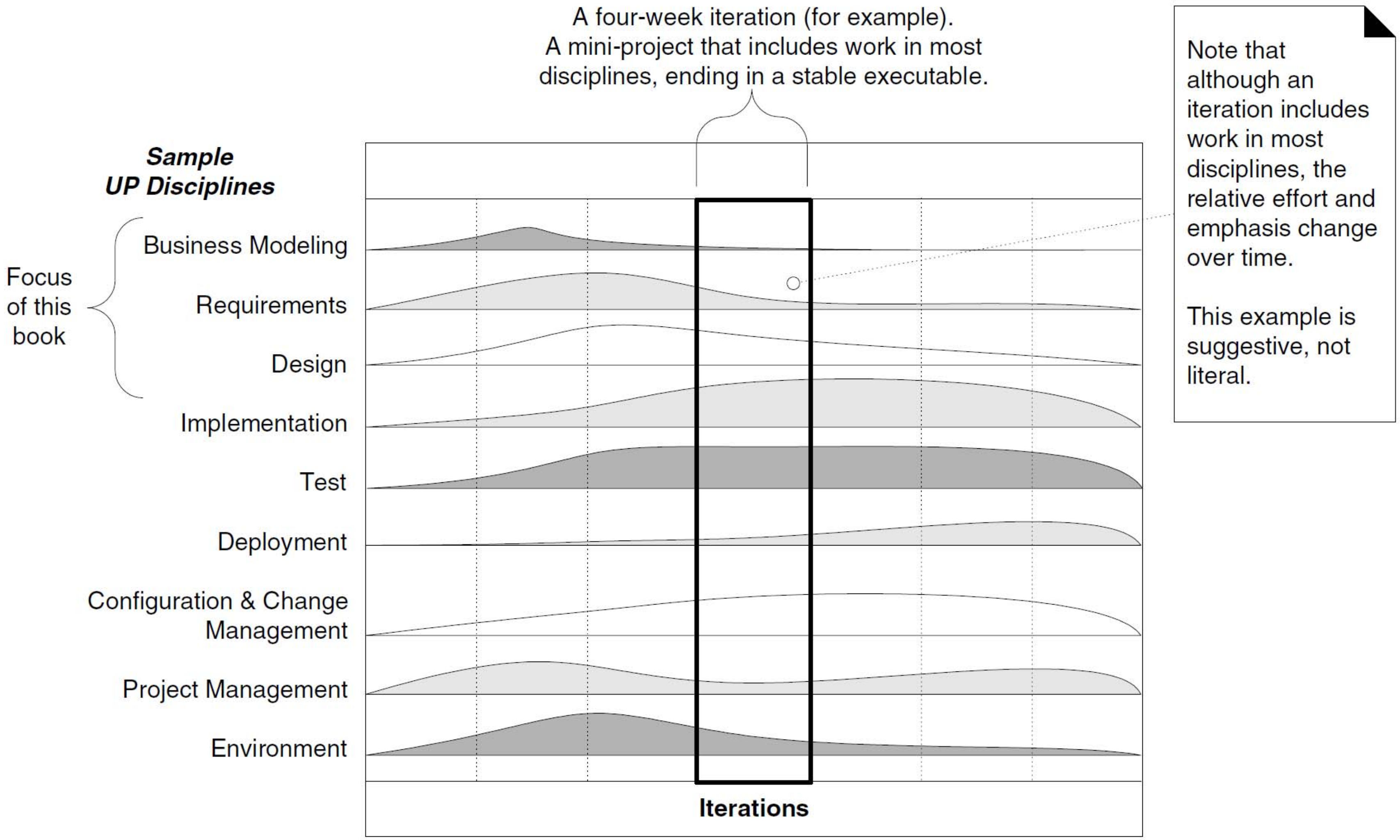
- Additional **best practices** and **key concepts**:
  - Tackle high-risk and high-value issues in early iterations **(→ Risk-driven, Client-driven)**
  - Continuously engage users for evaluation and feedback **(→ Client-driven)**
  - Build a cohesive, core architecture in early iterations **(→ Architecture-centric)**
  - Continuously verify quality; test early, often, and realistically
  - Apply use cases where appropriate
  - Do some visual modeling (with the UML)
  - Carefully manage requirements (configuration management)

The UP Phases

- A UP project organizes the work and iterations across **4 major phases**:
  1. **Inception** : approximate vision, business case, scope, vague cost estimates
  2. **Elaboration** : refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates
  3. **Construction** : iterative implementation of the remaining lower risk and easier elements, and preparation for deployment
  4. **Transition** : beta tests, deployment

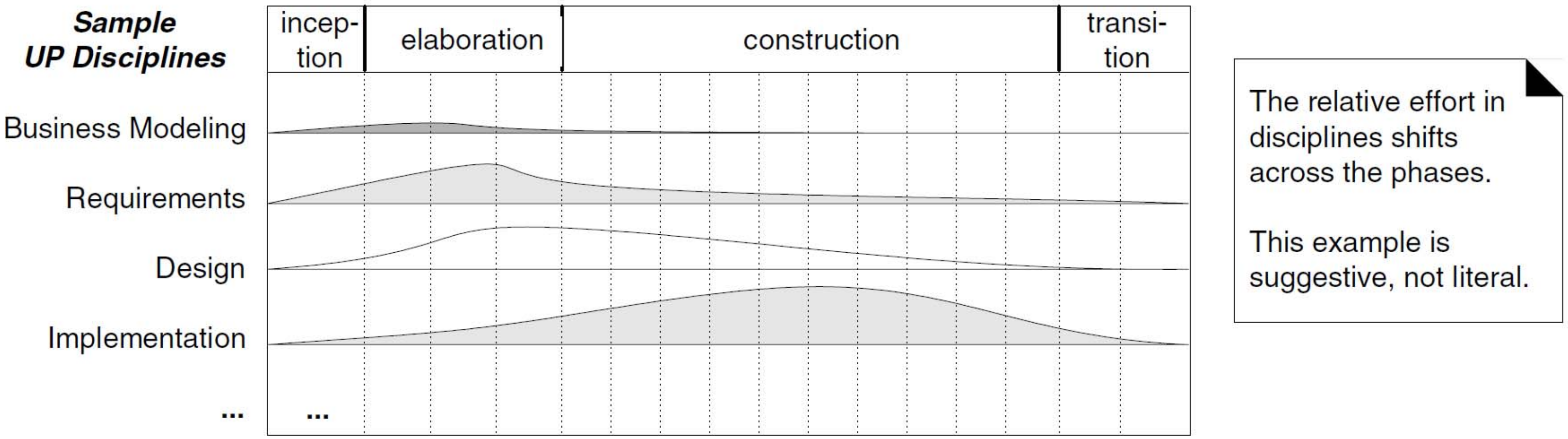


The UP Disciplines



Relationship Between the Disciplines and Phases

- The relative effort in disciplines **shifts** to across the phases.



- Artifact : A general term for any work product
    - Example: code, web graphics, database schema, text documents, diagrams, models and so on

- Discipline : A set of activities and related artifacts in one subject area
    - Example: the activities within requirements analysis

The UP Development Case

- **Development Case**:
  - An artifact in the Environment discipline
  - Documenting the choice of practices and UP artifacts for a project

  - For example, the development case for the NextGen POS case study :

Discipline	Practice	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	agile modeling req. workshop	Domain Model		s		
Requirements	req. workshop vision box exercise dot voting	Use-Case Model	s	r		
		Vision	s	r		
		Supplementary Specification	s	r		
		Glossary	s	r		
Design	agile modeling test-driven dev.	Design Model		s	r	
		SW Architecture Document		s		
		Data Model		s	r	
Implementation	test-driven dev. pair programming continuous integration coding standards	…				
Project Management	agile PM daily Scrum meeting	…				
…						

You Know You Didn't Understand Iterative Development or the UP When …

- Some signs that you have not understood what it means to adopt iterative development and the UP in a healthy agile spirit.

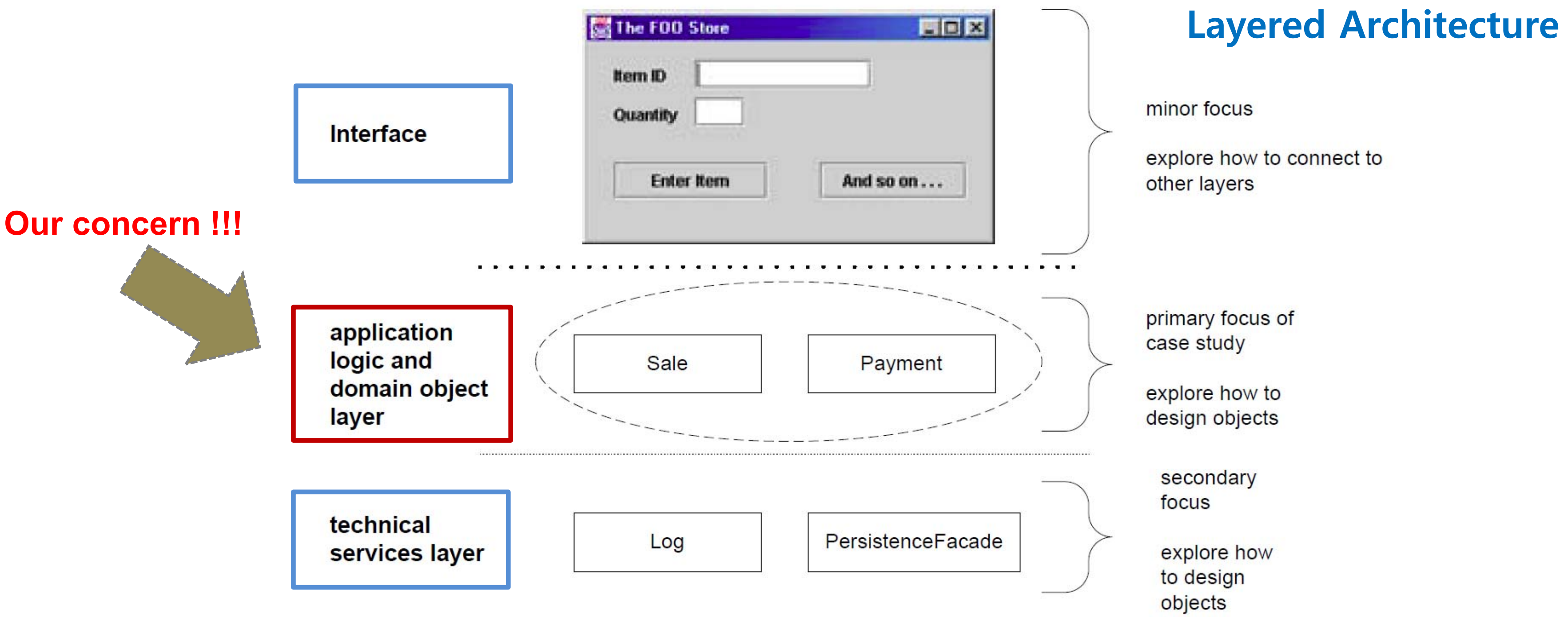
  - You try to define most of the requirements before starting design or implementation. Similarly, you try to define most of the design before starting implementation; you try to fully define and commit to an architecture before iterative programming and testing.
  - You think that inception = requirements, elaboration = design, and construction = implementation (that is, superimposing the waterfall on the UP).
  - You think that the purpose of elaboration is to fully and carefully define models, which are translated into code during construction.
  - You believe that a suitable iteration length is three months long, rather than three weeks long.
  - You try to plan a project in detail from start to finish; you try to speculatively predict all the iterations, and what should happen in each one.

Chapter 3.

Case Studies

What is Covered in the Case Studies?

- Generally, applications include
  - UI elements,
  - **Core application logic**,
  - OS, database access and collaboration with external SW/HW components.



Case One: The NextGen POS System

The first case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are interesting requirement and design problems to solve. In addition, it's a real problemgroups really do develop POS systems with object technologies.

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.

Part 2: Inception

- Chapter 4. Inception is Not the Requirements Phase
- Chapter 5. Evolutionary Requirements
- Chapter 6. Use Cases
- Chapter 7. Other Requirements

Chapter 4.

Inception is Not the Requirements Phase

What is Inception?

- Most projects require **a short initial step** to question about:
  - What is the vision and business case for this project?
  - Feasible?
  - Buy and/or build?
  - Rough unreliable range of cost: Is it $10K-100K or in the millions?
  - Should we proceed or stop?

- Inception should be short.
  - **One week** for most projects
  - Most requirements analysis occurs during the elaboration phase, not inception.

Artifacts Start in Inception

Artifact[ ]	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

[ ] – These artifacts are partially completed in this phase. They will be iteratively refined in subsequent iterations.
Name capitalization implies an officially named UP artifact.

How Much UML During Inception?

- The purpose of inception is to collect just enough information to
  - establish a common vision,
  - decide if moving forward is feasible, and
  - decide if the project is worth serious investigation in the elaboration phase.

- Much UML diagramming is not required.
  - Inception has more focus on understanding the basic scope and 10% of the requirements, expressed mostly in text forms.
  - In practice, most UML diagramming will occur in the next phase elaboration.

Chapter 5.
Evolutionary Requirements

Requirements

- **Requirements**
  - **Capabilities** and **conditions** to which **the system must conform**

- **Requirement analysis** is
  - to **find**, communicate and **organize** what is really needed, in a form that is clear both to clients and team members.

- In the UP, requirements are analyzed iteratively and skillfully.

- The UP encourages skillful elicitation (finding) via techniques such as
  - writing use cases with customers,
  - requirements workshops that include both developers and customers,
  - a demo of the results of each iteration to the customers, to solicit feedback.

Types and Categories of Requirements

- In the UP, requirements are categorized according to the **FURPS+ model**
  [R. Grady: "Practical Software Metrics for Project Management and Process Improvement", Prentice-Hall Inc, 1992.]
  - **Functional** : features, capabilities, security
  - **Usability** : human factors, help, documentation
  - **Reliability** : frequency of failure, recoverability, predictability
  - **Performance** : response times, throughput, accuracy, availability, resource usage
  - **Supportability** : adaptability, maintainability, internationalization, configurability

  - The "+" in FURPS+ indicates ancillary and sub-factors such as:
    - Implementation : resource limitations, languages and tools, hardware, …
    - Interface : constraints imposed by interfacing with external systems
    - Operations : system management in its operational setting
    - Packaging : for example a physical box
    - Legal : Licensing and so forth

- It is helpful to use FURPS+ categories as **a checklist for requirements coverage**.

Quality Attributes/Requirements

- **Quality attributes/requirements**:
  - **Usability + Reliability + Performance + Supportability**
  - Also called "**Non-functional requirements**"

- The quality attributes often have a strong influence on the architecture of a system.

How Requirements are Organized

- The UP offers several requirements artifacts. (But, they are all optional.)
  - Use-Case Model
    - A set of typical scenarios of using a system
    - These are primarily for **functional (behavioral) requirements**.
  - Supplementary Specification
    - Basically, everything is not in the use cases.
    - This artifact is primarily for **all non-functional requirements**, such as performance or licensing.
    - It is also the place to record functional features not expressed (or expressible) as use cases; for example, a report generation.
  - **Glossary**
    - It defines noteworthy terms.
  - **Vision**
    - A short executive overview document for quickly learning the project's big ideas.
  - **Business Rules**
    - It typically describe requirements or policies that transcend one software project.

Chapter 6.
Use Cases

Use Cases

- **Use cases** are **text stories** of some actors using a system to meet goals.
  - A mechanism to capture (analyzes) requirements
  - An example (Brief format):
    - **Process Sale**: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.
  - Use case is not a diagram, but a text.

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case Diagram

- **Use case diagram** illustrates the name of use cases and actors, and the relationships between them.
  - System context diagram
  - A summary of all use cases

Use case

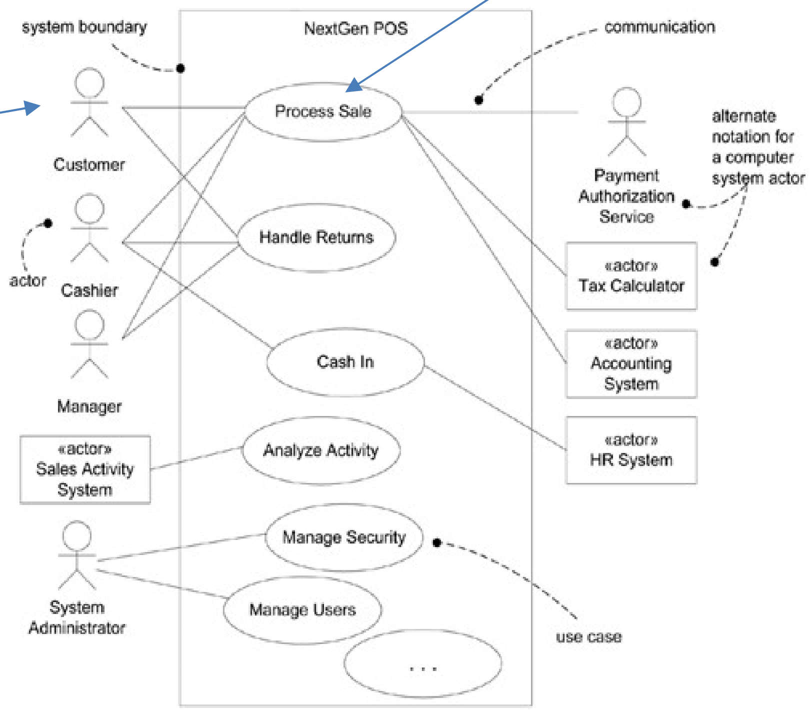
Actor

Something with behavior, such as a person, computer system, or organization

- Primary Actor : has user goals fulfilled through using services of the SuD (System Under Discussion) , *e.g.,* cashier

- Supporting Actor : provides a service to the SuD, *e.g.,* payment authorization service

- Offstage Actor : has an interest in the behavior of the use case, but is not primary or supporting, *e.g.,* tax agency



Are Use Cases Functional Requirements?

- Yes, **Use Cases are requirements**, primarily **functional (behavioral)** requirements.
  - "**F**" (functional or behavioral) in terms of **FURPS+** requirements types
  - Can also be used for other types.

Three Common Use Case Formats

- **Brief** :
  - Terse one paragraph summary, usually the main success scenario or a happy path

- **Casual** :
  - Informal paragraph format.
  - Multiple paragraphs that cover various scenarios.

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 customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting system, ...

- **Fully Dressed** :
  - Includes all steps, variations and supporting sections (e.g., preconditions)

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Example: Process Sale, Fully Dressed Style

Use Case UC1: Process Sale

Scope: NextGen POS application
Level: user goal
Primary Actor: Cashier
Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.
Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

*a. At any time, Manager requests an override operation:
  1. System enters Manager-authorized mode.
  2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
  3. System reverts to Cashier-authorized mode.

*b. At any time, System fails:
  To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
  1. Cashier restarts System, logs in, and requests recovery of prior state.
  2. System reconstructs prior state.
    2a. System detects anomalies preventing recovery:
      1. System signals error to the Cashier, records the error, and enters a clean state.
      2. Cashier starts a new sale.

1a. Customer or Manager indicate to resume a suspended sale.
  1. Cashier performs resume operation, and enters the ID to retrieve the sale.
  2. System displays the state of the resumed sale, with subtotal.
    2a. Sale not found.
      1. System signals error to the Cashier.
      2. Cashier probably starts new sale and re-enters all items.
  3. Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
  1. Cashier verifies, and then enters tax-exempt status code.
  2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):
  1. System signals error and rejects entry.
  2. Cashier responds to the error:
    2a. There is a human-readable item ID (e.g., a numeric UPC):
      1. Cashier manually enters the item ID.
      2. System displays description and price.
        2a. Invalid item ID: System signals error. Cashier tries alternate method.
    2b. There is no item ID, but there is a price on the tag:
      1. Cashier asks Manager to perform an override operation.
      2. Managers performs override.
      3. Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)
    2c. Cashier performs Find Product Help to obtain true item ID and price.
    2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
  1. Cashier can enter item category identifier and the quantity.

3c. Item requires manual category and price entry (such as flowers or cards with a price on them):
  1. Cashier enters special manual category code, plus the price.

3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:
  This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.
  1. Cashier enters item identifier for removal from sale.
  2. System removes item and displays updated running total.
    2a. Item price exceeds void limit for Cashiers:
      1. System signals error, and suggests Manager override.
      2. Cashier requests Manager override, gets it, and repeats operation.

3-6b. Customer tells Cashier to cancel sale:
  1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:
  1. System records sale so that it is available for retrieval on any POS register.
  2. System presents a "suspend receipt" that includes the line items, and a sale ID used to retrieve and resume the sale.

4a. The system supplied item price is not wanted (e.g., Customer complained about something and is offered a lower price):
  1. Cashier requests approval from Manager.
  2. Manager performs override operation.
  3. Cashier enters manual override price.
  4. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:
  1. System restarts the service on the POS node, and continues.
    1a. System detects that the service does not restart.
      1. System signals error.
      2. Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
  1. Cashier signals discount request.
  2. Cashier enters Customer identification.
  3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:
  1. Cashier signals credit request.
  2. Cashier enters Customer identification.
  3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don't have enough cash:
  1. Cashier asks for alternate payment method.
    1a. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:
1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:
1. Customer enters their credit account information.
2. System displays their payment for verification.
3. Cashier confirms.
   3a. Cashier cancels payment step:
      1. System reverts to "item entry" mode.
4. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
   4a. System detects failure to collaborate with external system:
      1. System signals error to Cashier.
      2. Cashier asks Customer for alternate payment.
5. System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).
   5a. System receives payment denial:
      1. System signals denial to Cashier.
      2. Cashier asks Customer for alternate payment.
   5b. Timeout waiting for response.
      1. System signals timeout to Cashier.
      2. Cashier may try again, or ask Customer for alternate payment.
6. System records the credit payment, which includes the payment approval.
7. System presents credit payment signature input mechanism.
8. Cashier asks Customer for a credit payment signature. Customer enters signature.
9. If signature on paper receipt, Cashier places receipt in cash drawer and closes it.

7c. Paying by check...

7d. Paying by debit...

7e. Cashier cancels payment step:
1. System reverts to "item entry" mode.

7f. Customer presents coupons:
1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
   1a. Coupon entered is not for any purchased item:
      1. System signals error to Cashier.

9a. There are product rebates:
1. System presents the rebate forms and rebate receipts for each item with a rebate.

9b. Customer requests gift receipt (no prices visible):
1. Cashier requests gift receipt and System presents it.

9c. Printer out of paper.
1. If System can detect the fault, will signal the problem.
2. Cashier replaces paper.
3. Cashier requests another receipt.

Special Requirements:
- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.
- . . .

Technology and Data Variations List:

*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:
- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

Guideline: Write in an Essential UI-Free Style

- **Essential writing style** is to express user intentions and system responsibilities, rather than concrete actions.
  - Concrete use cases are better avoided during early requirements analysis.
  - For example: ***Manage Users*** use case

Essential Style	Concrete Style
1. Administrator identities self. 2. System authenticates identity. 3. …	1. Administrator enters ID and PW in dialog box. 2. System authenticates Administrator. 3. System displays the “edit user” window. 4. …

Guideline: Write Black-Box Use Cases

- Don't describe **the internal working** of the system, its components or design.
  - Define what the system does (analysis), rather than how it does it (design).

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Process: Evolutionary Requirements in Iterative Methods

Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	***Use-Case Model***	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		

Case Study: Use Cases in the NextGen POS

- Use cases are developed and refined iteratively.

- Use Cases of the NextGen POS at the inception phase

Fully Dressed	Casual	Brief
Process Sale	Process Rental	Cash In
Handle Returns	Analyze Sales Activity	Cash Out
	Manage Security	Manage Users
	...	Start Up
		Shut Down
		Manage System Tables
		...

Chapter 7.

Other Requirements

Other Requirements Artifacts

- **Supplementary Specification**
  - Captures and identifies **other kinds of requirements**, such as
    - reports, documentation, packaging, supportability, licensing, and so forth

- **Glossary**
  - Captures terms and definitions; a data dictionary

- **Vision**
  - Summarizes the "vision" of the project; an executive summary

- **Business Rules**
  - Capture long-living and spanning rules or policies (such as tax laws), that transcend one particular application

Supplementary Specification

- Other requirements, information and constraints not easily captured in the use cases or Glossary, including system-wide "**URPS+**" **quality attributes**.

- Elements of the Supplementary Specification include:
  - FURPS+ requirementsfunctionality, usability, reliability, performance, and supportability
  - reports
  - hardware and software constraints (operating and networking systems, ...)
  - development constraints (for example, process or development tools)
  - other design and implementation constraints
  - internationalization concerns (units, languages)
  - documentation (user, installation, administration) and help
  - licensing and other legal concerns
  - packaging
  - standards (technical, safety, quality)
  - physical environment concerns (for example, heat or vibration)
  - operational concerns (for example, how do errors get handled, or how often should backups be done?)
  - application-specific domain rules
  - information in domains of interest (for example, what is the entire cycle of credit payment handling?)

Process: Evolutionary Requirements in Iterative Methods

Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration ⟶	I1	E1..En	C1..Cn	T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
	Business Rules	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	

Part 3: Elaboration - Iteration 1 Basics

- Chapter 8. Iteration 1 Basics
- Chapter 9. Domain Models
- Chapter 10. System Sequence Diagram
- Chapter 11. Operation Contracts

- Chapter 12. Requirements to Design Iteratively
- Chapter 13. Logical Architecture and UML Package Diagrams
- Chapter 14. On to Object Design
- Chapter 15. UML Interaction Diagram
- Chapter 16. UML Class Diagram

- Chapter 17. GRASP: Designing Objects with Responsibilities
- Chapter 18. Object Design Examples with GRASP

- Chapter 19. Designing for Visibility
- Chapter 20. Mapping Designs to Code
- Chapter 21. Test-Driven Development and Refactoring

Chapter 8.

Iteration 1 Basics

What Happened in Inception?

- **Inception is a short** **(only one week)** **step** to elaboration including:
  - A short requirements workshop
  - Most actors, goals, and use cases named
  - Most use cases written in brief format (10~20% are written in fully dressed detail)
  - Most influential and risky requirements identified
  - Version one of the Vision and Supplementary Specification written
  - Risk list
  - Technical proof-of-concept prototypes and other investigations to explore the technical feasibility of special requirements
  - User interface-oriented prototypes to clarify the vision of functional requirements
  - Recommendations on what components to buy/build/reuse, to be refined in elaboration
  - High-level candidate architecture and components proposed
  - Plan for the first iteration
  - Candidate tools list

On to Elaboration

- **Elaboration** is the initial series of iterations during which:
  - The core, risky software architecture is programmed and tested.
  - The majority of requirements are discovered and stabilized.
  - The major risks are mitigated or retired.

Iteration 1 Requirements and Emphasis

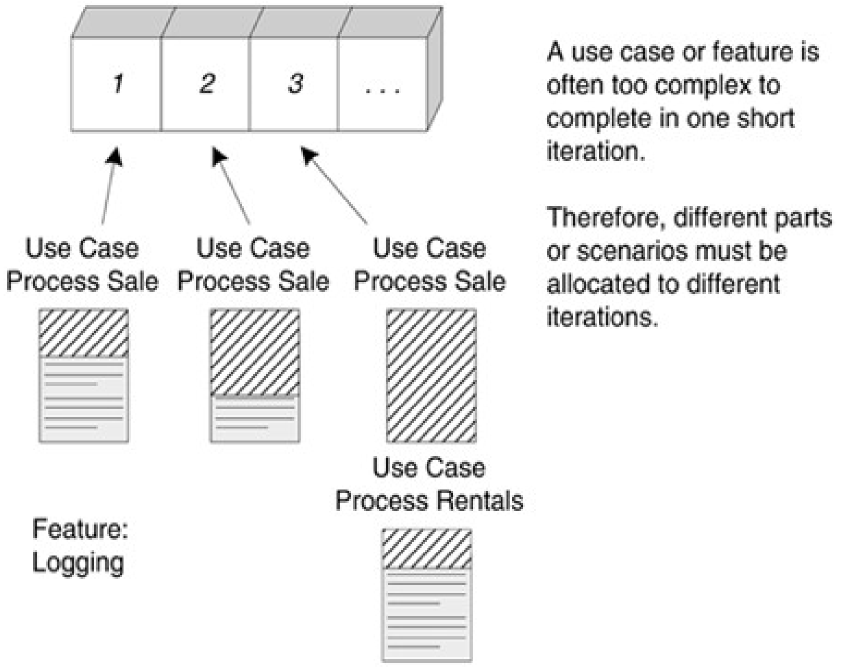
Book Iterations vs. Real Project Iterations

Iteration-1 of the case studies in this book is driven by learning goals rather than true project goals. Therefore, iteration-1 is not architecture-centric or risk-driven. On a UP project, we would tackle difficult, risky things first. But in the context of a book helping people learn fundamental OOA/D and UML, we want to start with easier topics.

- The NextGen POS example
  - The requirements for the 1st iteration follow:
    - Implement a basic, key scenario of the *Process Sale* use case: entering items and receiving a cash payment.
    - Implement a *Start Up* use case as necessary to support the initialization needs of the iteration.
    - Nothing fancy or complex is handled, just a simple happy path scenario, and the design and implementation to support it.
    - There is no collaboration with external services, such as a tax calculator or database.
    - No complex pricing rules are applied.

Implement Requirements Incrementally

- Incremental development for the same use case across iterations
  - The requirements for the iteration-1 are subsets of the complete requirements or use cases.



UP Artifacts Start in Elaboration

- These will not be completed in one iteration; rather will be refined over a series of iterations.

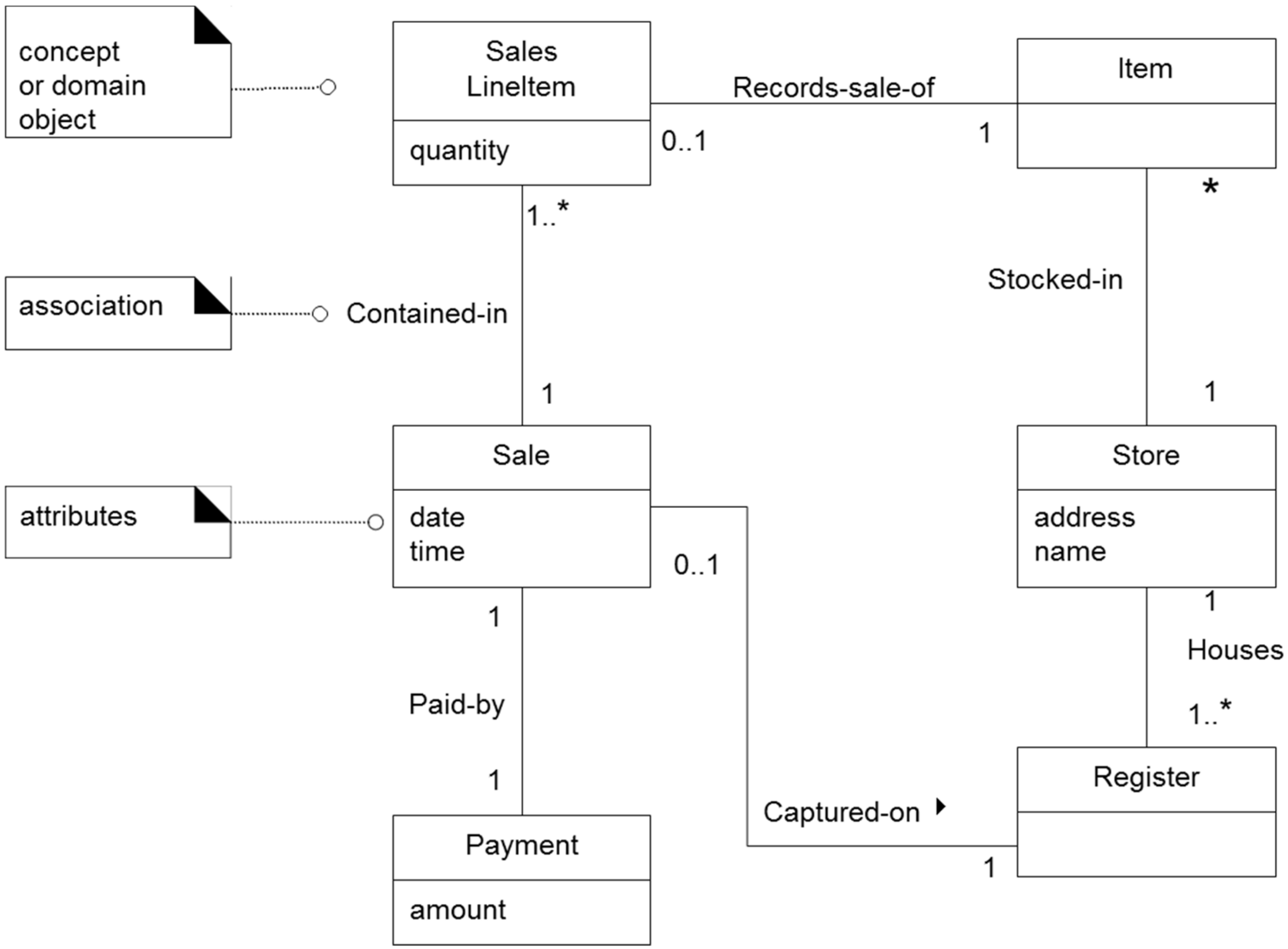
Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

Chapter 9.
Domain Models

Domain Model

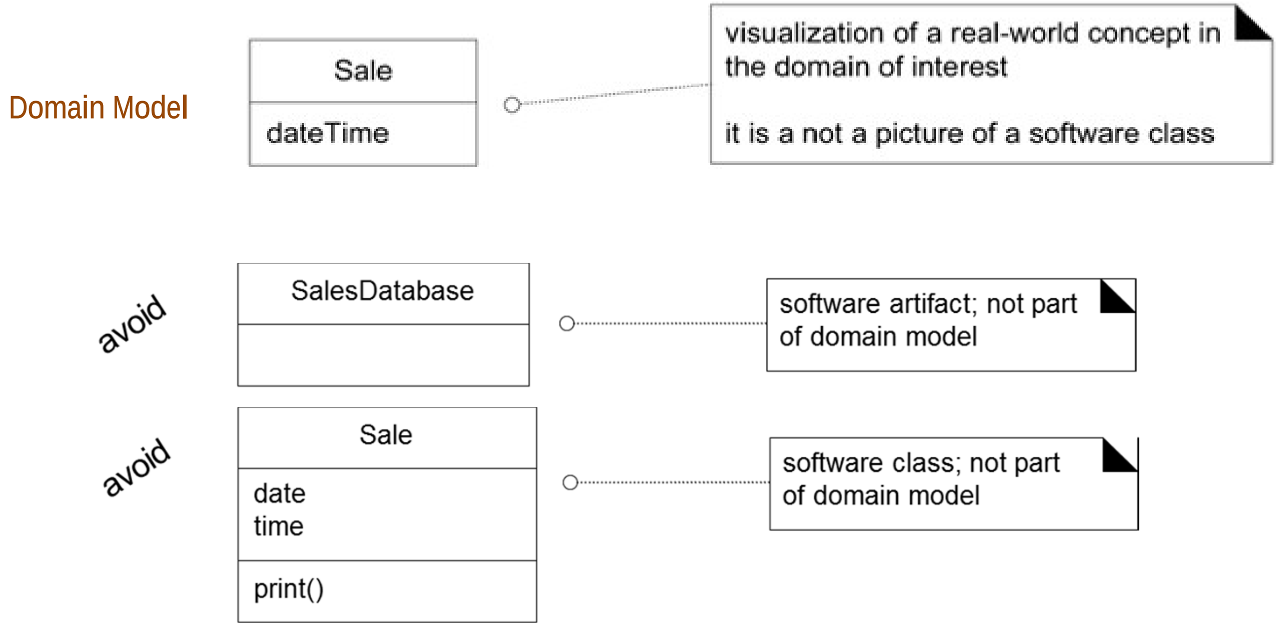
- **Domain model** is a *visual* representation of conceptual classes or real-situation objects in a domain.
  - The most important classic model in OO analysis
  - Can act as a source of inspiration for designing software objects and classes.
  - **Visual dictionary** of the noteworthy abstractions, domain vocabulary, and information contents of the domain
  - Not represents software objects
- Domain model is illustrated with class diagrams
  - **no operations**
  - **domain objects** (or conceptual classes)
  - **associations** between conceptual classes
  - **attributes** of conceptual classes
- Domain model is a kind of a preliminary version of class diagram, if we are well used to the application domain.

Partial Domain Model for NextGen POS



Domain Model is Not Software Objects

- A UP domain model is **<u>not</u>** <u>of software objects</u> such as:
  - Software classes (*i.e.*, C++ or Java classes)
  - Elements representing artifacts related to the implementation of the system (*e.g.*, a database or a window)
  - Methods (operations)



Why Create a Domain Model?

- Two reasons to create a domain model:

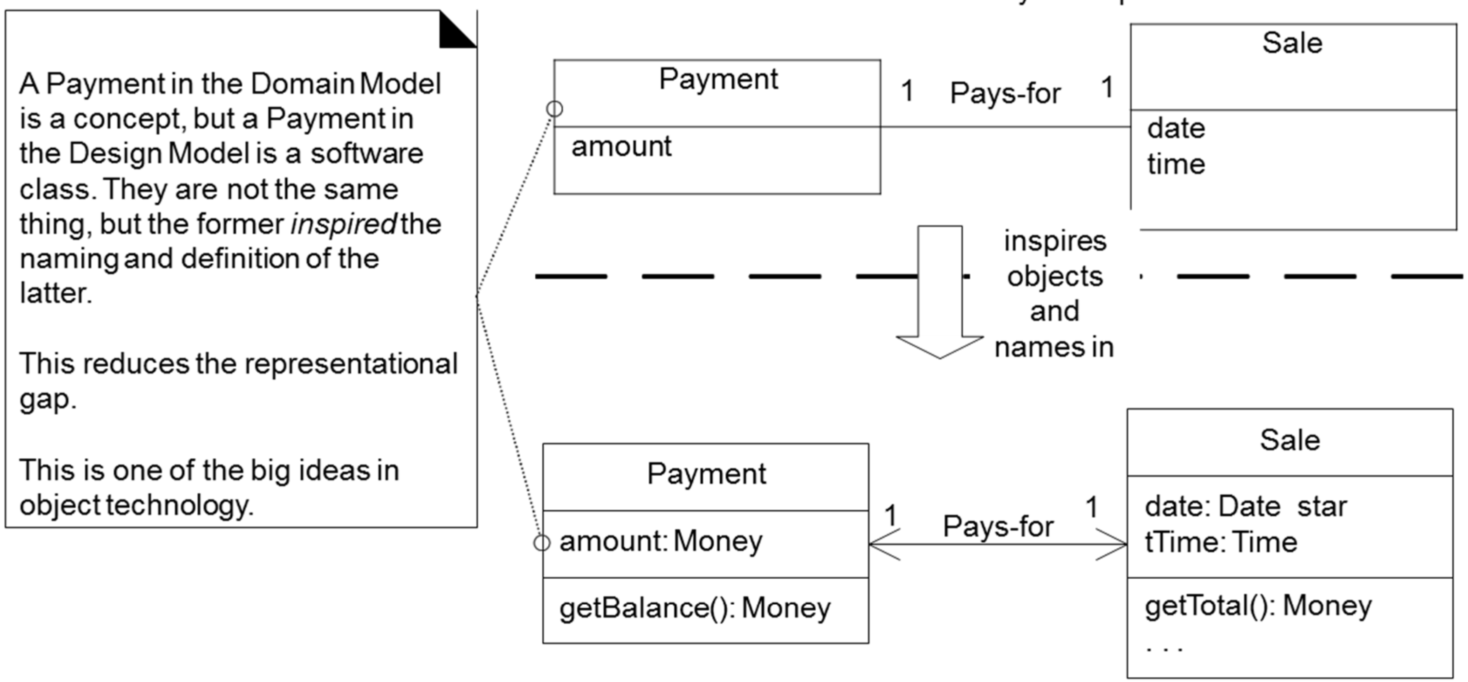
  1. **Getting to know the domain** during early elaboration iterations, understanding the concepts involved and their relationships

  2. **Inspiring the software classes** of the **domain layer** in the design model.
     - This prevents software from being far away from the reality of the domain.
     - **lower representation gap :** Use software class names in the domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities.

Lower Representation Gap

UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.



UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

How to Create a Domain Model

- Same as the way of creating class diagrams.
  1. Find **conceptual classes** and draw them in a UML class diagram
  2. Add **associations** and **attributes** to conceptual classes

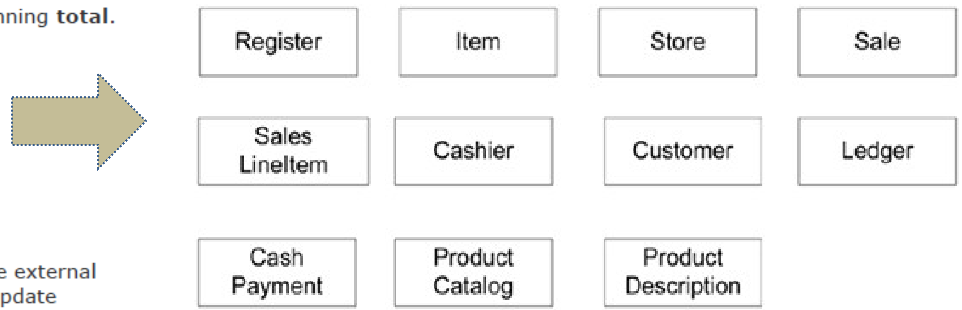
- Identification of Noun Phrases
  – Identify **the nouns and noun phrases** in a textual description of the domain, and consider them as candidate conceptual classes and attributes.

Process Sale use case

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters **item identifier**.
4. System records **sale line item** and presents **item description, price**, and running **total**. Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment**.
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions**) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).



Is the Domain Model Correct?

- There is **no** such thing as a single correct domain model.
  - All models are approximations of the domain we are attempting to understand.

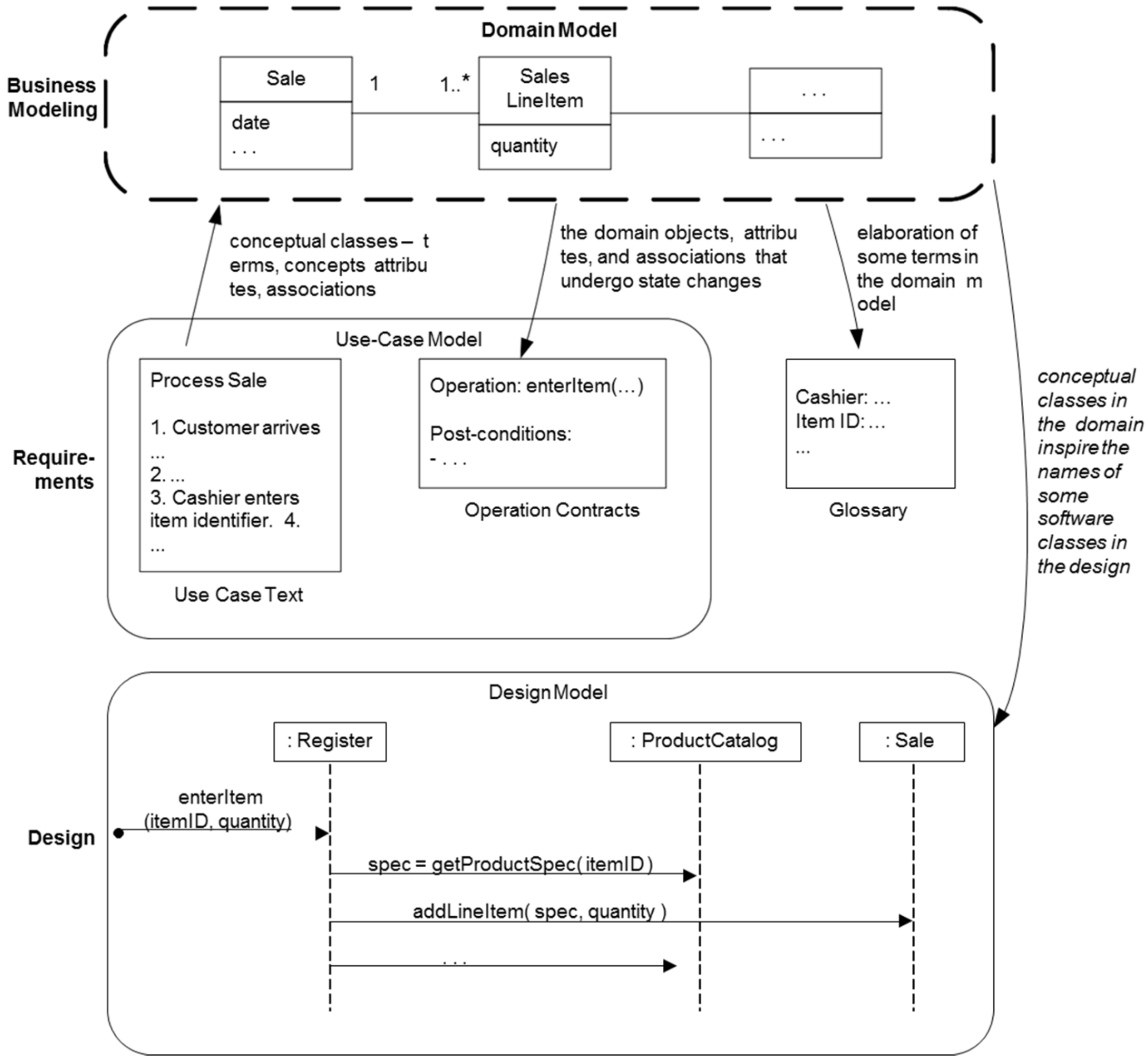
- The domain model is a primary tool of understanding and communication among a particular group.
  - Correct << Useful

Process: Iterative and Evolutionary Domain Modeling

- The UP Domain Model is usually both started and completed in the elaboration phase.

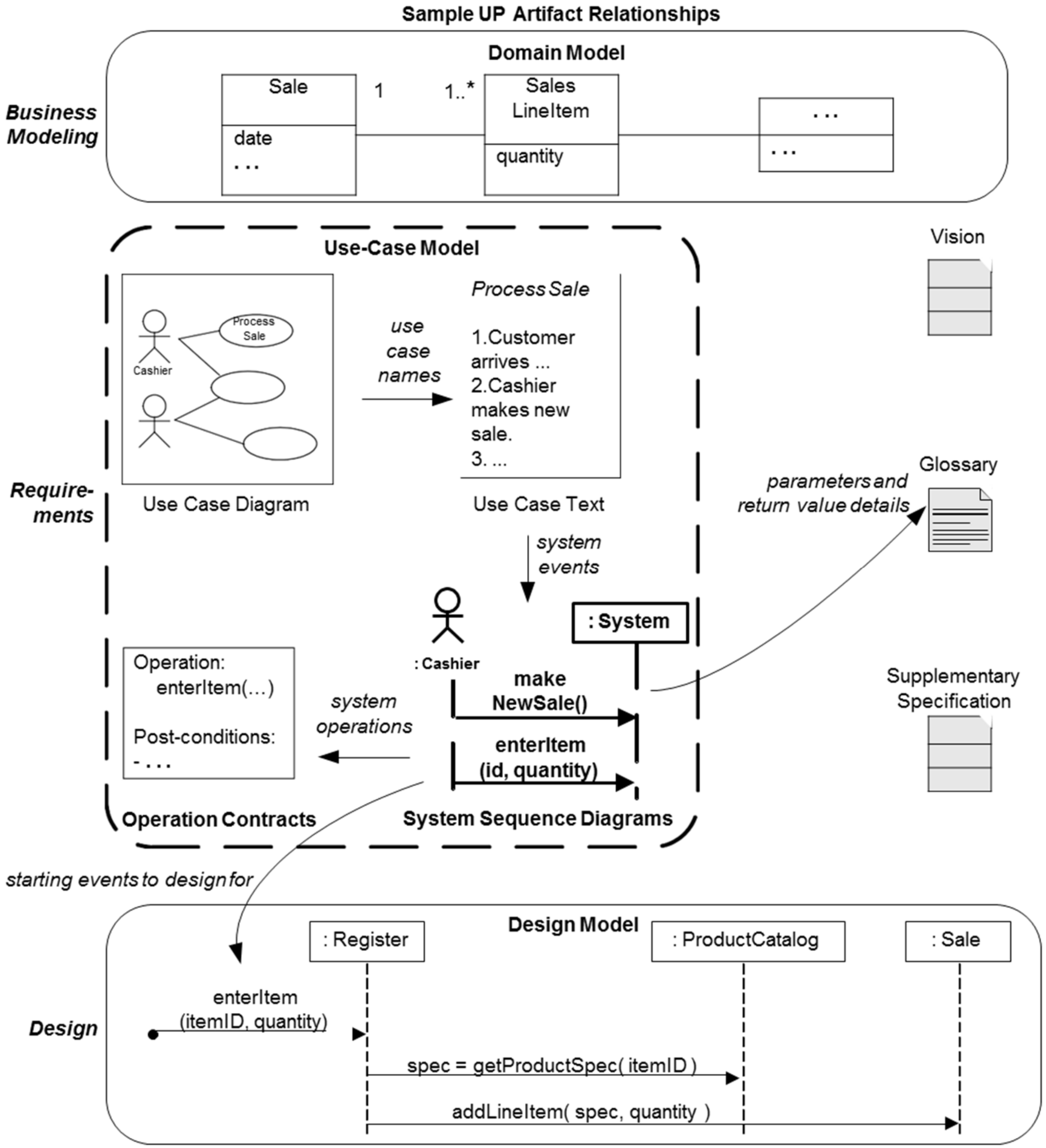
Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration →	I1	E1..En	C1..Cn	T1..T2
Business Modeling	***Domain Model***		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	

Sample UP Artifact Relationships



Chapter 10.

System Sequence Diagram



Sample UP Artifact Relationships
Business Modeling
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Use-Case Model
Process Sale
Cashier
Use Case Diagram
use case names
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. ...
Use Case Text
Vision
Require-ments
parameters and return value details
Glossary
system events
: Cashier
: System
make NewSale()
enterItem (id, quantity)
Operation: enterItem(…)
Post-conditions:
- . . .
system operations
Supplementary Specification
Operation Contracts
System Sequence Diagrams
starting events to design for
Design Model
: Register
: ProductCatalog
: Sale
Design
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)

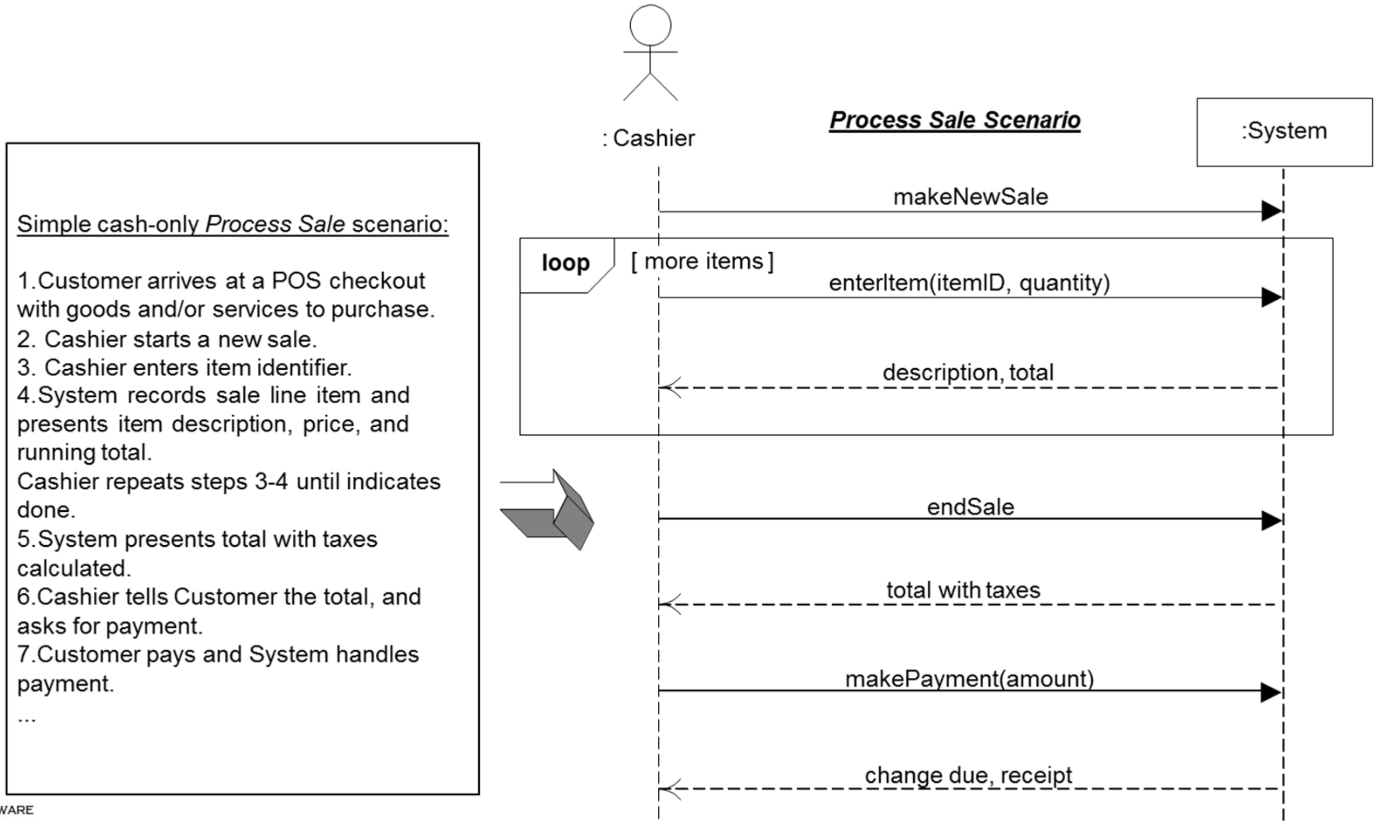
System Sequence Diagram

- **System sequence diagram (SSD)**
  - A picture that shows the events that external actors generate, their order, and inter-system events, for one particular scenario of a use case.
    - the external actors that interact directly with the system,
    - the system (as a black box), and
    - the system events that the actors generate
  - In the sequence diagram notation
  - Depict system behavior in terms of **what the system does**, not how it does it
  - Used as input to object design → **System operations**

- **Use cases** describe how external actors interact with the software system we are interested in creating.
  - During this interaction, an **actor** generates **system events** to a system, usually requesting some **system operation** to handle the event.

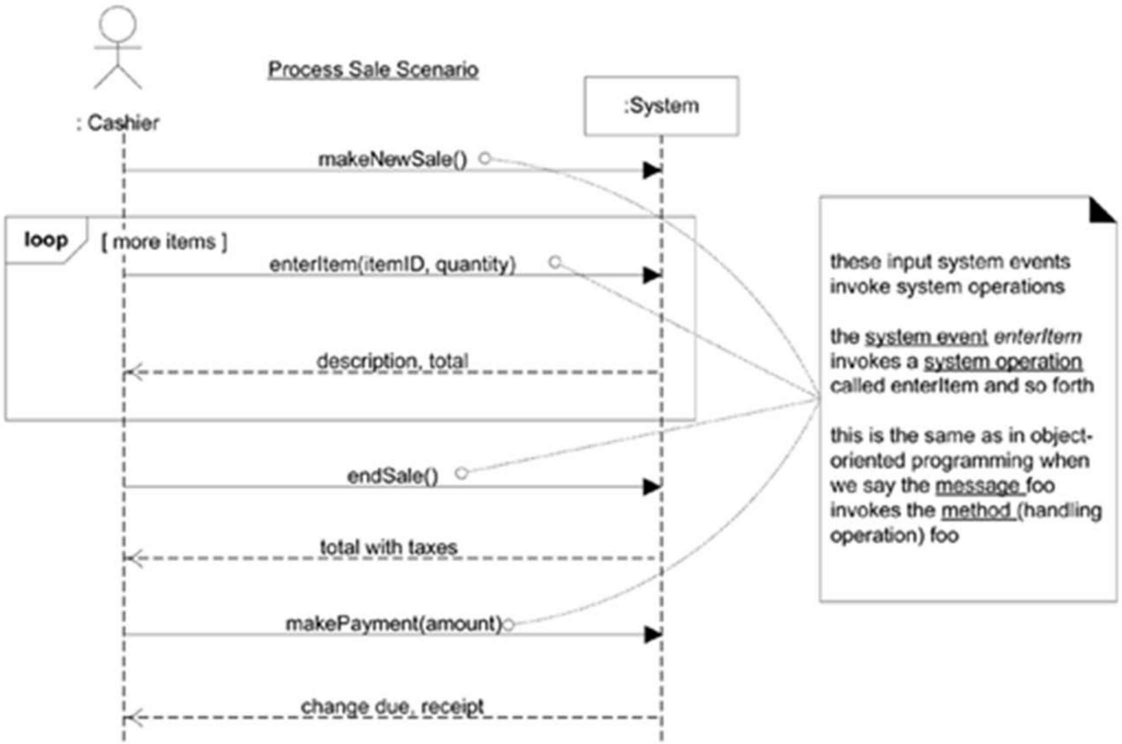
Applying UML: Sequence Diagrams

- The UML does not define something called ‘System Sequence Diagrams’.
  - We use the general UML sequence diagram notation.
  - The term ‘system’ in SSDs is used to emphasize the application of the UML sequence diagram to systems viewed as black boxes.
  - An SSD shows system events for one scenario of a use case.



System Operation

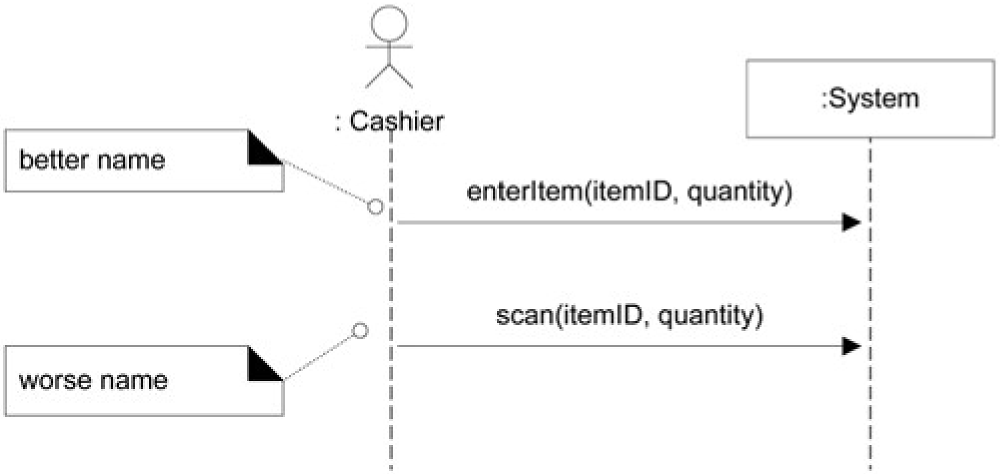
- **System operations**
  - Operations that the system as a black box component offers in its public interface
  - Show **system events**, which the SUD should have system operations to handle the system events.
  - **System Interfaces**: the entire set of system operations across all use cases



Guideline: How to Name System Events and Operations?

- System events should be expressed at the abstract level of intention rather than in terms of the physical input device.

- Example : *scan(itemID)* vs. *enterItem(itemID)*
  - The *enterItem* name is better, since it communicates intention rather than the input device.



Process: Iterative and Evolutionary SSDs

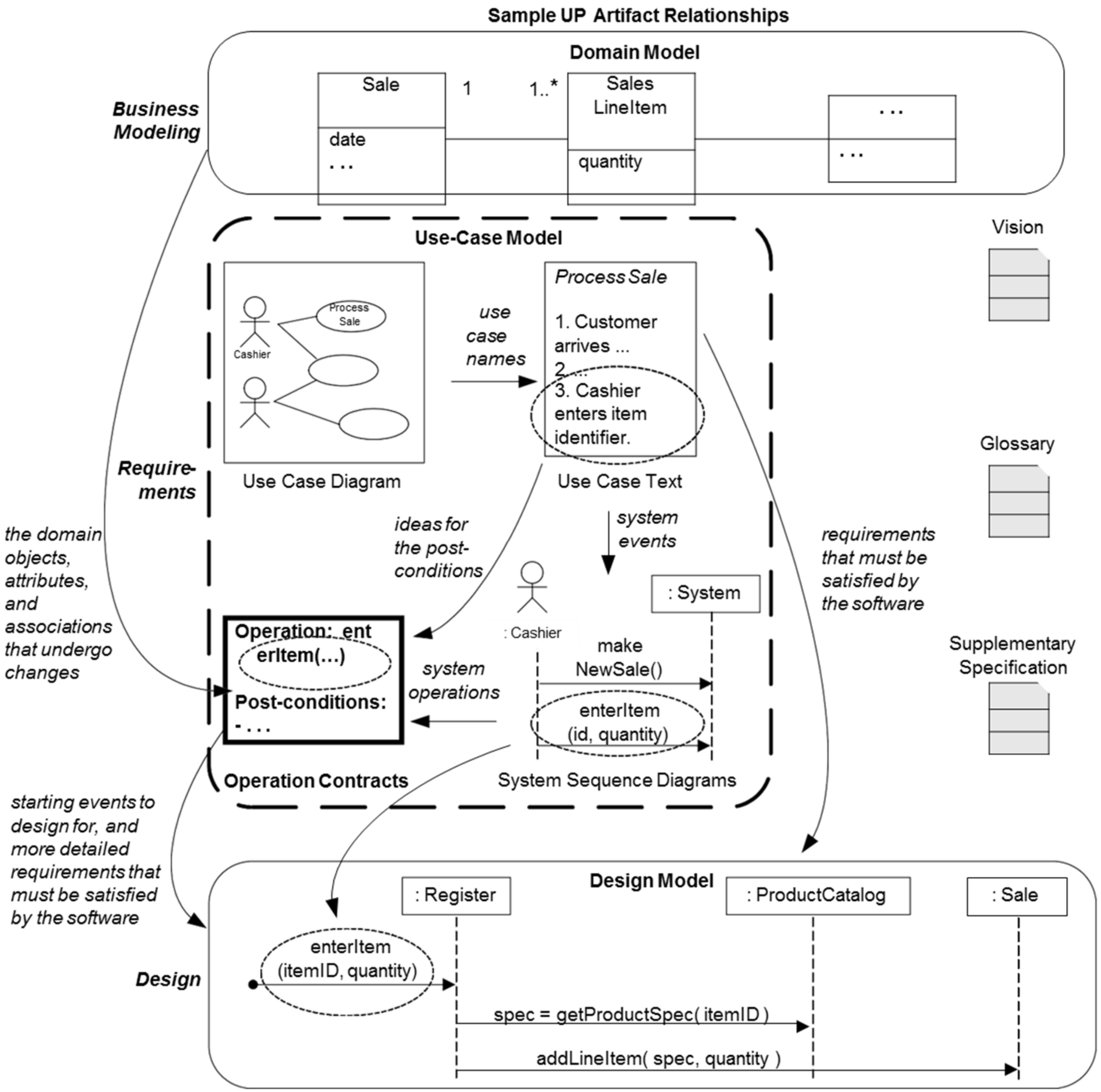
- The UP doesn't mention explicitly SSDs, but we can use them.
  - Since the UP is very flexible, allowing any useful technique to be applied in its context.

- **Most SSDs are created during elaboration**, when it is useful to
  - identify the details of the **system events** to clarify what major operations which the system must be designed to handle,
  - write **system operation contracts**, and possibly to support **estimation**.

Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration ⟶	I1	E1..En	C1..Cn	T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	

Chapter 11.

Operation Contracts



Sample UP Artifact Relationships
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Business Modeling
Use-Case Model
Cashier
Process Sale
use case names
Process Sale
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Use Case Diagram
Use Case Text
Require-ments
Vision
Glossary
Supplementary Specification
the domain objects, attributes, and associations that undergo changes
ideas for the post-conditions
system events
requirements that must be satisfied by the software
: System
: Cashier
make NewSale()
enterItem (id, quantity)
Operation: enterItem(…)
Post-conditions: - . . .
system operations
Operation Contracts
System Sequence Diagrams
starting events to design for, and more detailed requirements that must be satisfied by the software
Design Model
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)
Design

Operation Contracts

- **Operation contracts**
  - Use a pre- and post- condition form to describe detailed changes to objects in a domain model, as the result of a system operation.
  - Operation contracts are usually used in a Design Model for object methods,
  - But, can also be used in a domain model **as contracts of high-level system operations**.

Operation:	Name of operation, and parameters
Cross References:	Use cases this operation can occur within
Preconditions:	Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation. These are non-trivial assumptions the reader should be told.
Postconditions:	This is the most important section. The state of objects in the Domain Model after completion of the operation. Discussed in detail in a following section.

Example

- An operation contract for the *enterItem* system operation.

Contract CO2: enterItem

Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- A SalesLineItem instance sli was created *(instance creation)*. - sli was associated with the current Sale *(association formed)*. - sli.quantity became quantity *(attribute modification)*. - sli was associated with a ProductDescription, based on itemID match *(association formed)*.

The categorizations such as "*(instance creation)*" are a learning aid, not properly part of the contract.

Postconditions

- *Postconditions* describe changes in the state of objects in the domain model.
  - Not actions to be performed during the operation
  - Rather, Observations about the domain model objects that are true when the operation has finished. (→ past tense)
    - **Instance Creation and Deletion**
    - **Associations Formed and Broken**
    - **Attribute Modification**
  - Only necessary when the outcome of a system operation is not clear from the use case description.
    - It will be helpful when there are situations where the details and complexity of required state changes are awkward or too detailed to capture in use cases.

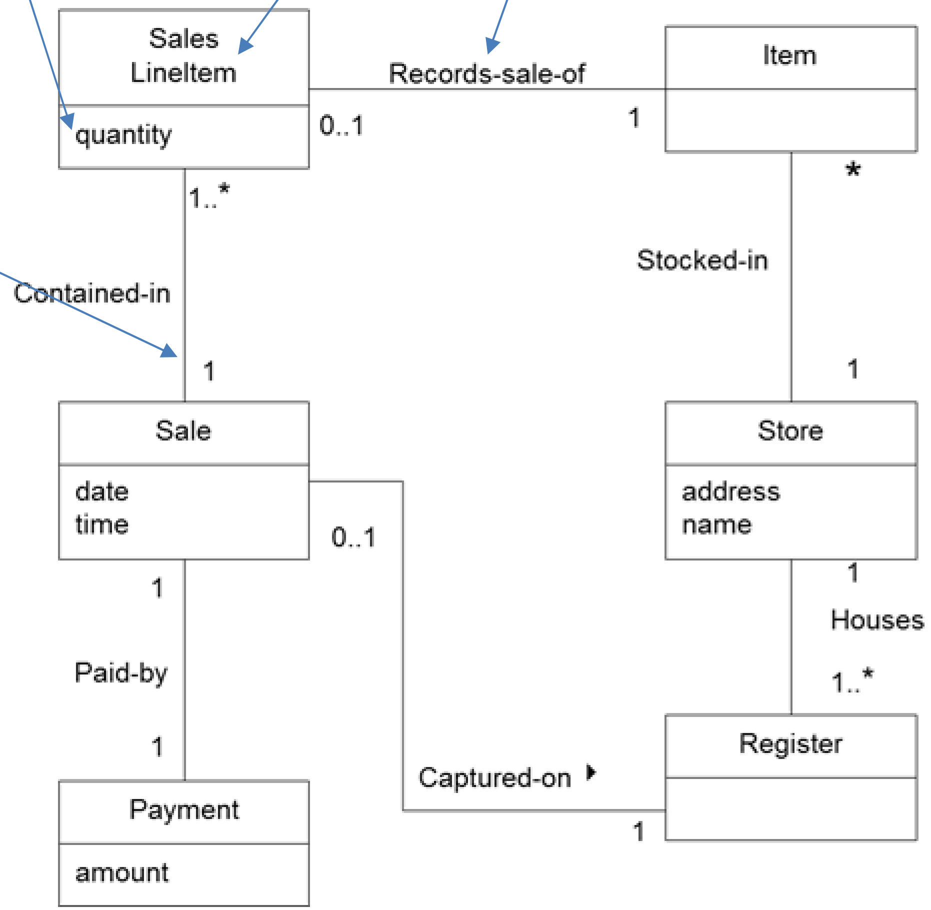
Example: *EnterItem* Postconditions

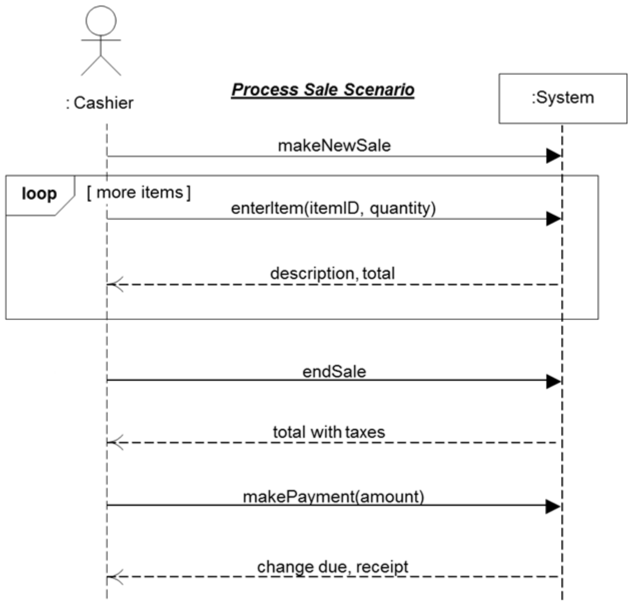
4 *sli* was associated with a *ProductDescription*, based on *itemID* match (association formed).

1 A *SalesLineItem* instance *sli* was created (instance creation).

3 *sli.quantity* became quantity (attribute modification).

2 *sli* was associated with the current *Sale* (association formed).





Applying UML: Operations, Contracts, and OCL

- In the UML,
  - **Operation** : a specification of a transformation or query that an object may be called to execute
  - **Method** : the implementation of an operation
    - Specifies the algorithm or procedure associated with an operation

- In the UML metamodel,
  - Operations have a **signature** **(name and parameters)** and are associated with **constraints** **(preconditions and postconditions).**
  - **OCL** **(Object Constraint Language)** is the formal language for expressing constraints in UML.

Chapter 12.

Requirements to Design Iteratively

Iteratively Analysis and Design

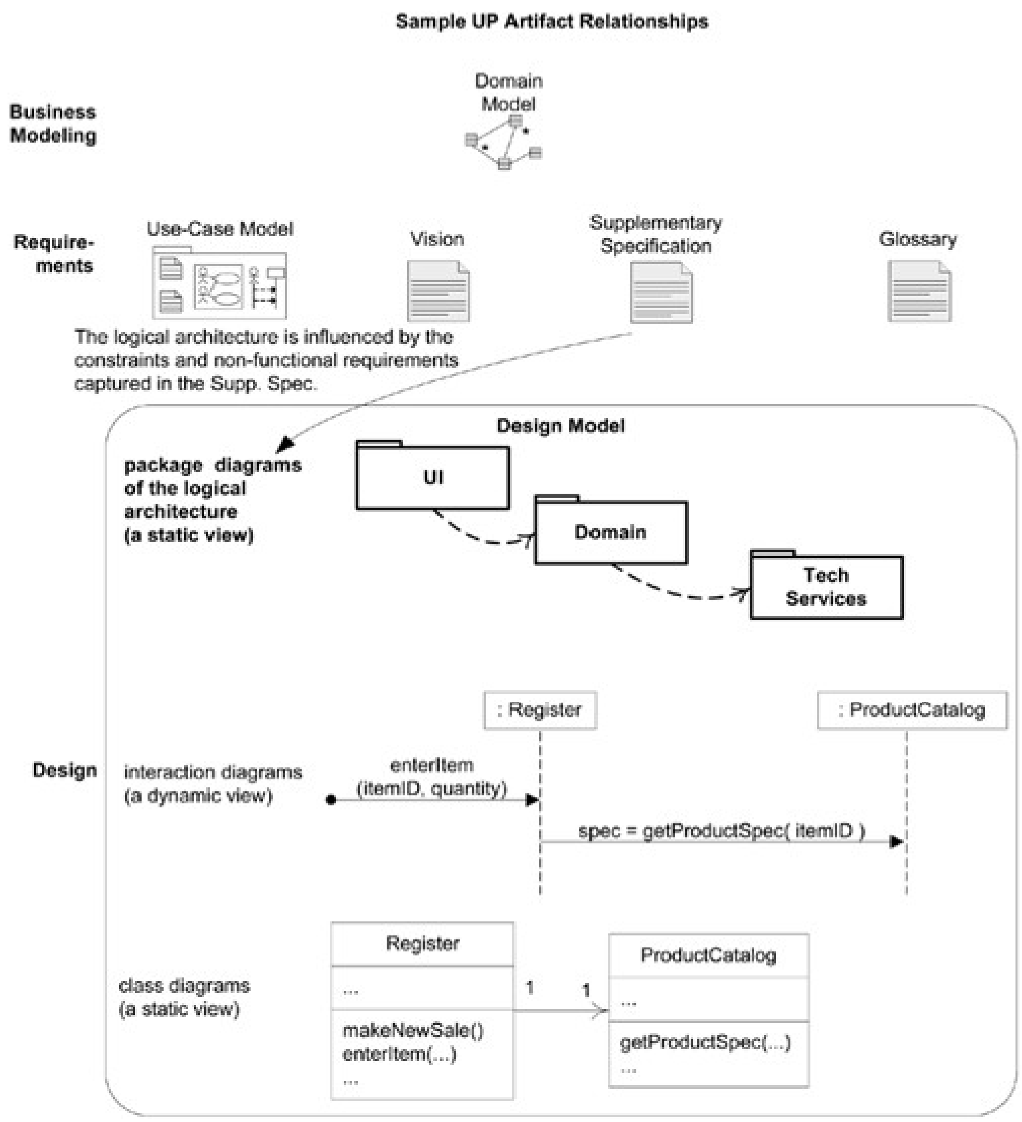
- **Analysis : Do the right thing**
  - The requirements and OOA have focused on learning to do the right thing.
  - Understanding some outstanding goals, related rules and constraints.

- **Design : Do the thing right**
  - Design work will stress do the thing right.
  - Skillful designing a solution to satisfy the requirements for its iteration.

- In iterative development, **a transition from requirements/OOA to design/implementation occur in each iteration.**

Chapter 13.

Logical Architecture and UML Package Diagrams


Sample UP Artifact Relationships
Business Modeling
Domain Model
Require-ments
Use-Case Model
Vision
Supplementary Specification
Glossary
The logical architecture is influenced by the constraints and non-functional requirements captured in the Supp. Spec.
Design Model
package diagrams of the logical architecture (a static view)
UI
Domain
Tech Services
Design
interaction diagrams (a dynamic view)
: Register
: ProductCatalog
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
class diagrams (a static view)
Register
...
makeNewSale()
enterItem(...)
...
1
1
ProductCatalog
...
getProductSpec(...)
...

Logical Architecture

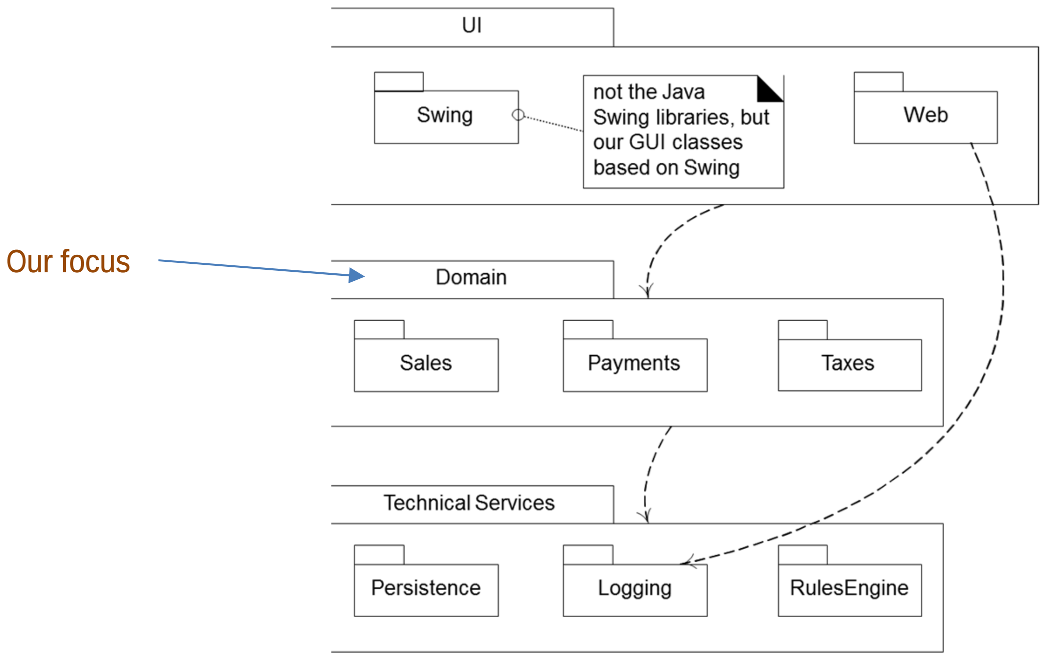
- The **logical architecture** is the large-scale organization of the **software classes** into **packages**, **subsystems**, and **layers**.
  - But, no decision about how these elements are deployed across different operating system processes or across physical computers in a network.
    → the deployment architecture (→ **UML Deployment Diagram)**

- **UML Package Diagrams** illustrate the logical architecture.
  - Can also be summarized as **Views** in a Software **Architecture Document (AD)**

- **Layer**
  - A very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system
  - Organized such that "higher" layers call upon services of "lower" layers
  - Can be depicted easily with UML package diagrams

Layered Architecture

- Typical layers in object-oriented systems:
  - **User Interface layer**
  - **Application Logic and Domain Objects layer**
    - Software objects representing domain concepts that fulfill application requirements
  - **Technical Services layer**
    - General purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging.
    - Usually application-independent and reusable across several systems



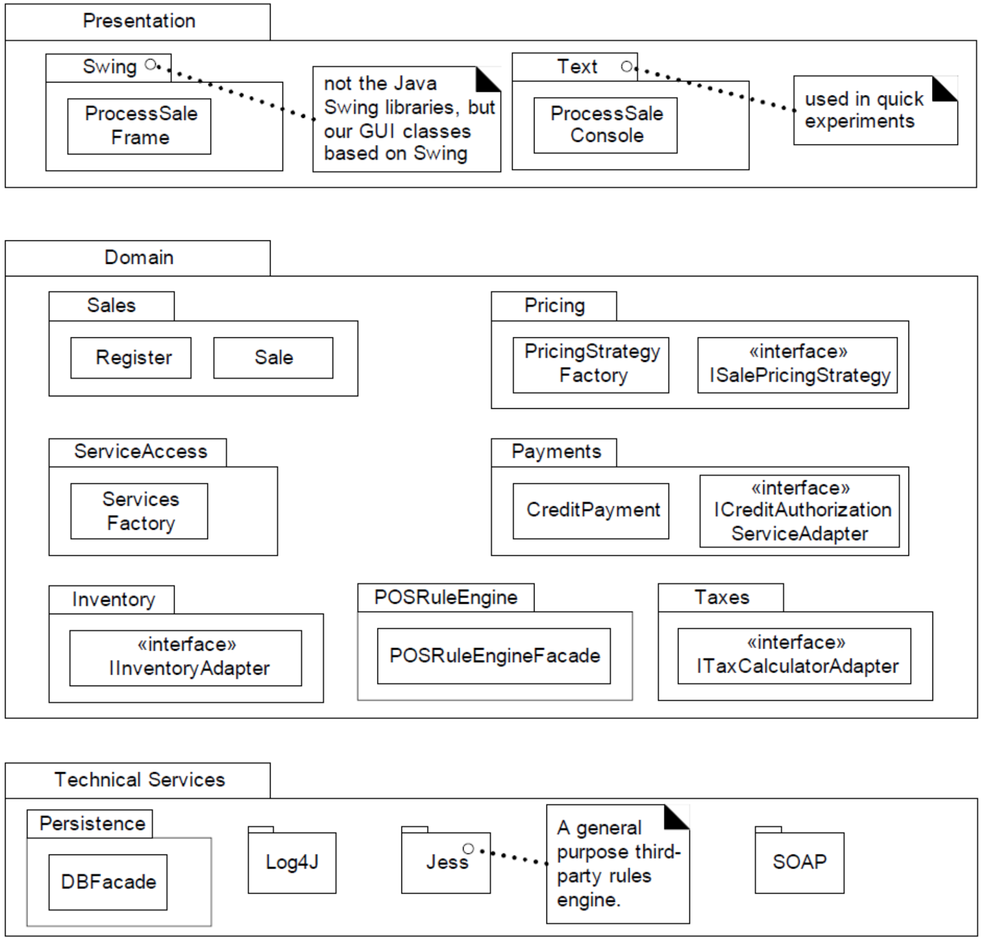
Software Architecture

- "A **software architecture** is the set of **significant decisions** about the organization of a software system,
  - the **selection of** the **structural** elements and their **interfaces** by which the system is composed, together with their **behavior** as specified in the collaborations among those elements,
  - the **composition of** these structural and behavioral elements into progressively larger subsystems,
  - and the **architectural style** that guides this organization - these elements and their interfaces, their collaborations, and their composition."

Booch, G., Rumbaugh, J, and Jacobson, I. 1999. The Unified Modeling Language User Guide.

Applying UML: Package Diagrams

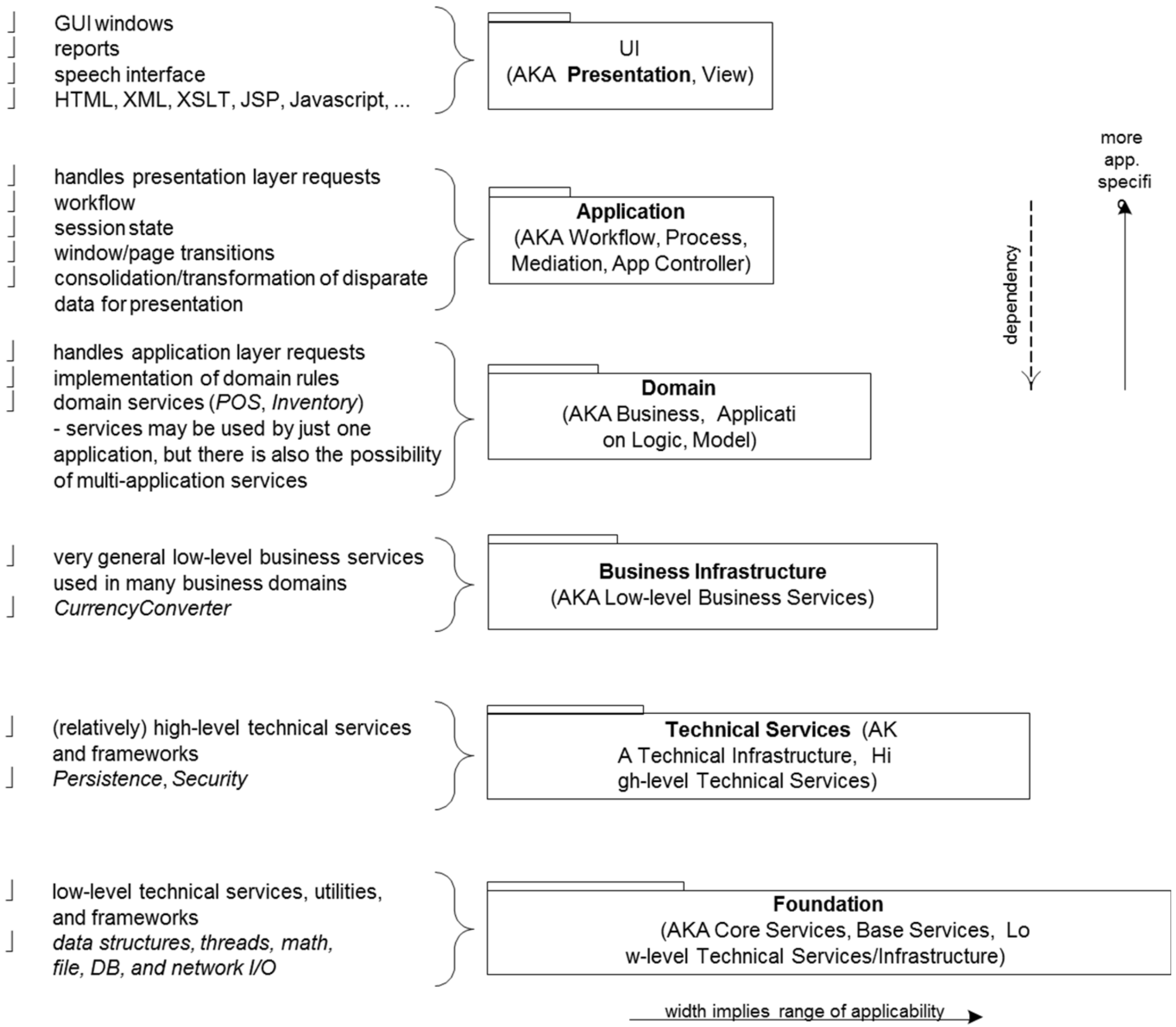
- **UML package diagrams** are often used to illustrate the logical architecture of a system.



A partial LA of NextGen POS

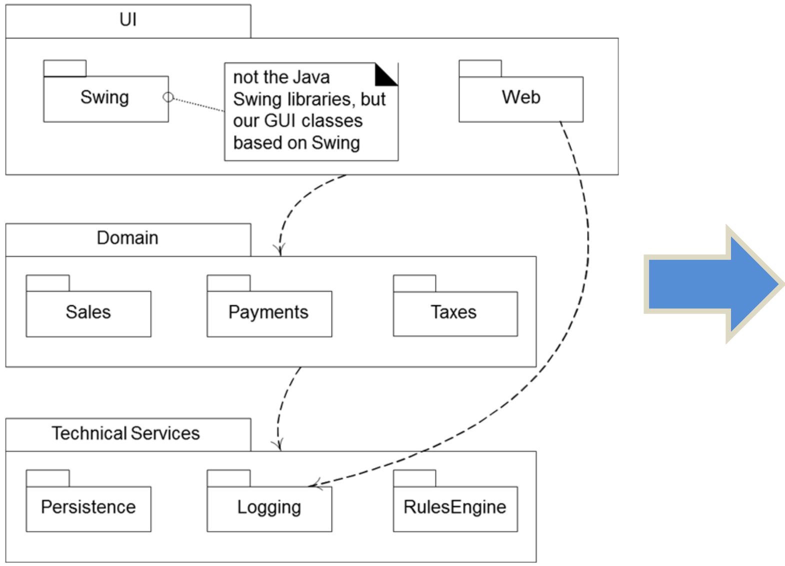
Design with Layers

- Example: Common Layers in an Information Systems Logical Architecture



Mapping Code Organization to Layers and UML Packages

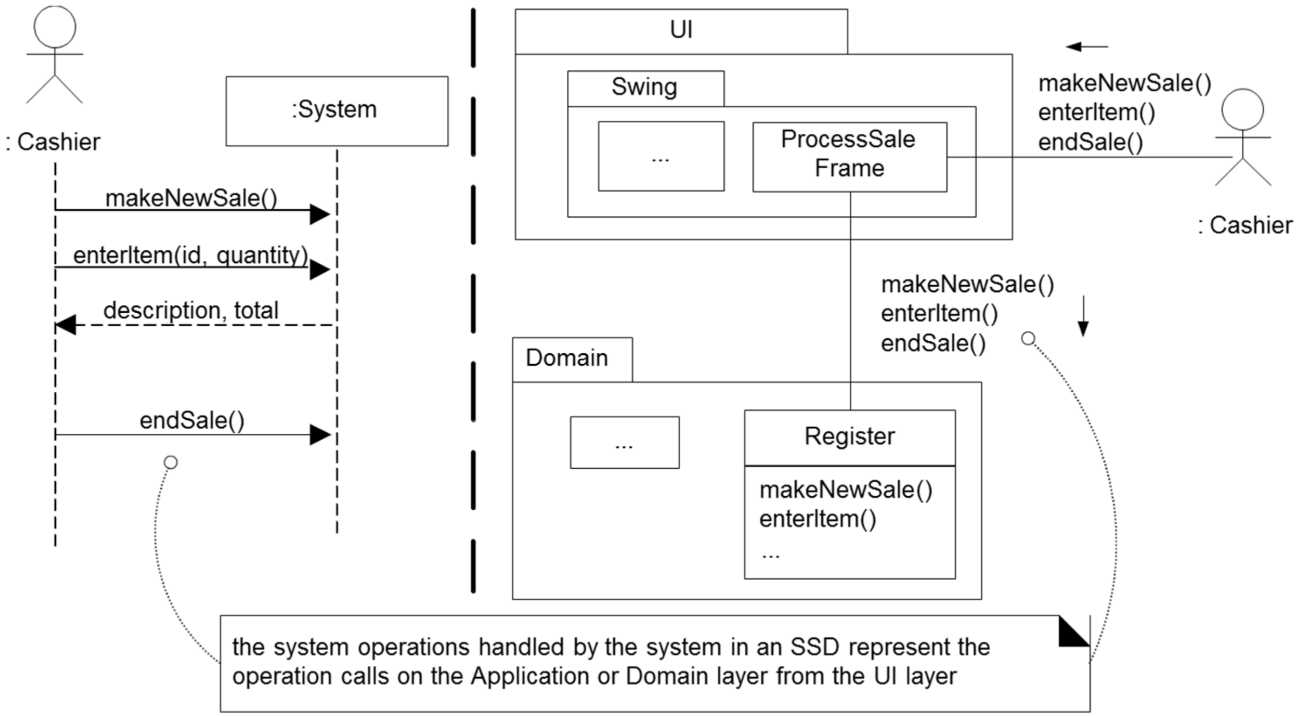
- Most popular OO languages provide support for packages.




```
// --- UI Layer
com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web

// --- DOMAIN Layer
    // packages specific to the NextGen project
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments

// --- TECHNICAL SERVICES Layer
    // our home-grown persistence (database) access layer
com.mycompany.service.persistence
    // third party
org.apache.log4j
org.apache.soap.rpc

// --- FOUNDATION Layer
    // foundation packages that our team creates
com.mycompany.util
```

Connections Between SSDs, System Operations and Layers

- In a well-designed layered architecture,
  – The UI layer objects will forward or delegate the requests from the UI layer (system operations) onto the domain layer for handling.
  – The messages sent from the UI layer to the domain layer will be the messages illustrated on the SSDs.

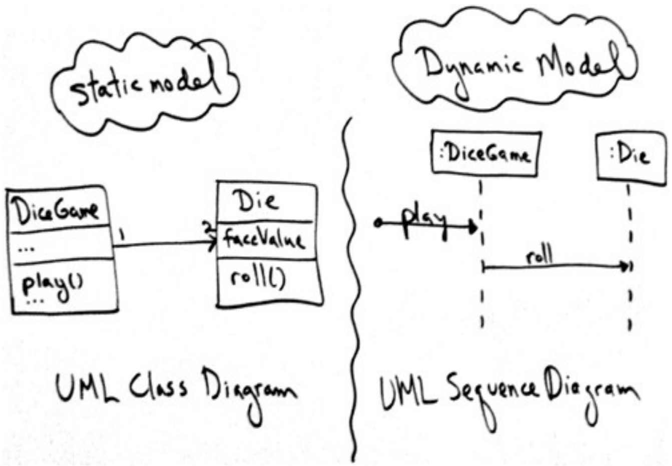


Chapter 14.
On to Object Design

Designing Objects: Static vs. Dynamic

- Two kinds of object models:
  - **Static models** help design the definition of packages, class names, attributes, and method signatures (but not method bodies).
    - Example: **UML class diagram**
    - Looks like the most important model.
  - **Dynamic models** help design the logic, the code, or the method bodies.
    - Example: **UML interaction diagrams** (sequence diagram, communication diagram)
    - Tend to be the more interesting, difficult, and important diagrams to create.

- Relationship between static and dynamic modeling:
  - Spend a short period of time on interaction diagrams, then switch to a wall of related class diagrams.



Dependable Software Laboratory

Static Object Modeling

- People new to UML tend to think that the important diagram is the static-view class diagram.
  - But, static and dynamic modelling are all important equivalently.
  - The most common static object modeling is with UML class diagrams.

- **Static UML Tools:**
  - Class diagram
  - Package diagram
  - Deployment diagram

Dynamic Object Modeling

- Most useful design work happens while drawing the UML dynamic-view interaction diagrams.
  - During dynamic object modeling (such as drawing sequence diagrams), we really think the exact details of what objects need to exist and how they collaborate via messages and methods.

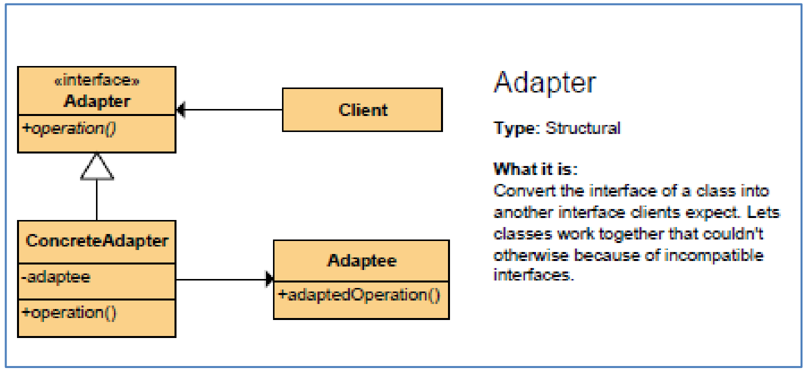
- Dynamic UML Tools:
  - Interaction diagrams (Sequence diagram)
  - Statechart diagram
  - Activity diagram

Object Design Skill over UML Notation Skill

- **The object design skills** are matter, not knowing how to draw UML.
  - Since, Drawing UML is a reflection of making decisions about the design.
- **Fundamental object design** requires knowledge of:
  - **Principles of responsibility assignment (GRASP)**
  - **Design patterns**

Pattern/ Principle	Description
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A

GRASP



Design Pattern of GoF

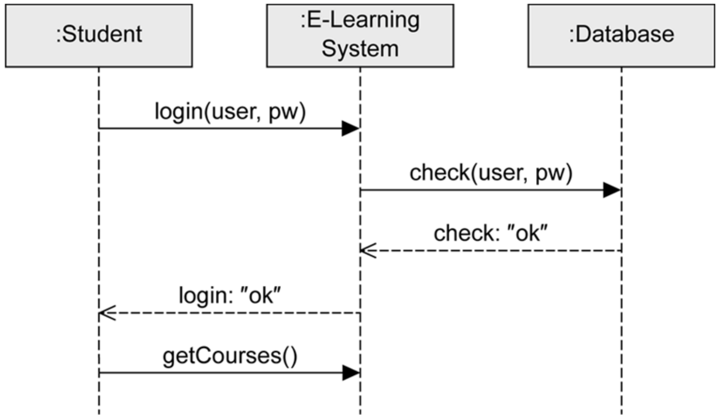
Chapter 15.
UML Interaction Diagrams

Interaction Diagrams

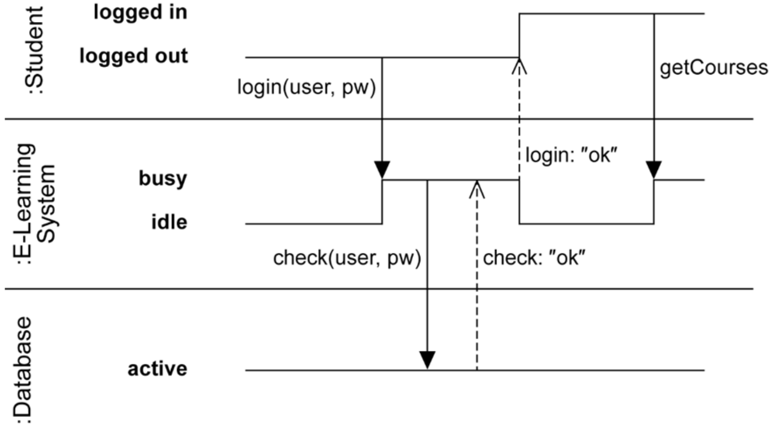
- **Interaction diagrams** illustrate how objects interact via messages.
  - Dynamic object modeling

  - **Sequence diagram**
  - (+) **Communication diagram**
  - (+) **Interaction overview diagram**
  - (+) **Timing diagram**

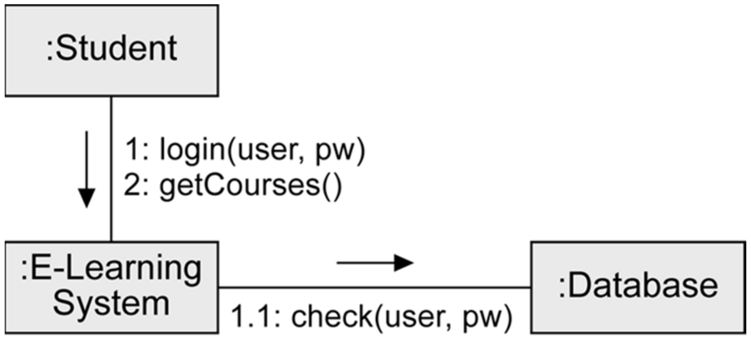
4 Interaction Diagrams



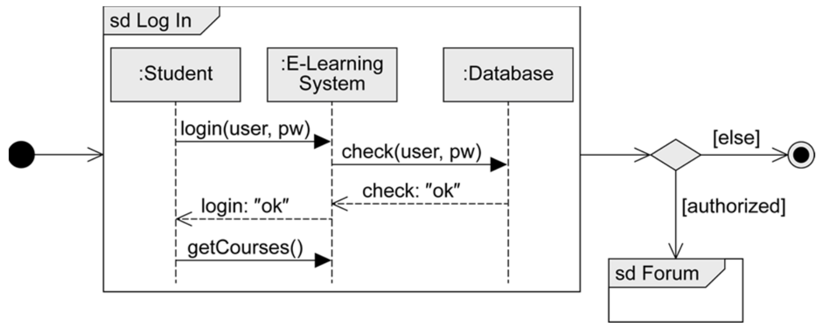
Sequence diagram



Timing diagram



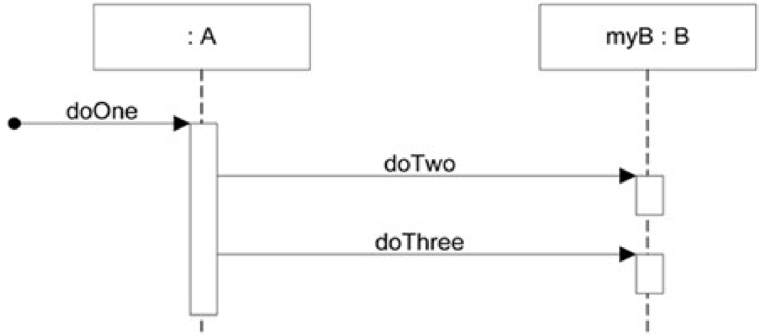
Communication diagram

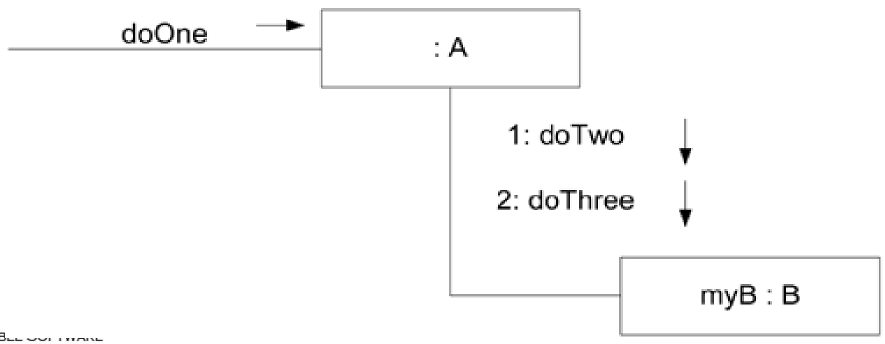


Interaction Overview diagram

Sequence and Communication Diagram

- **Sequence diagrams**
  - model the collaboration of objects based on a time sequence
- **Communication diagrams**
  - focus on showing the collaboration of objects rather than the time sequence

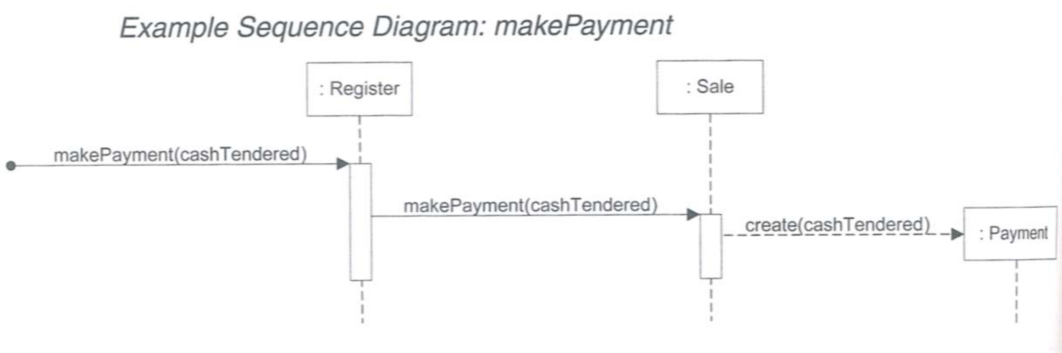


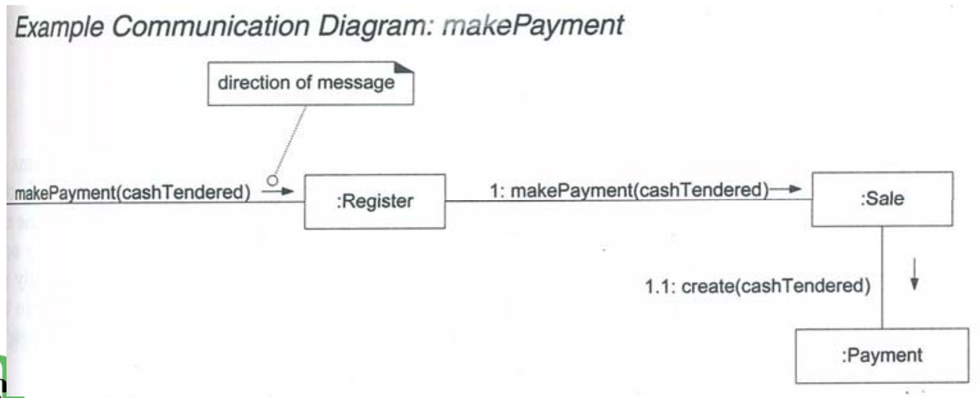



```
public class A
{
   private B myB = new B();

   public void doOne()
   {
      myB.doTwo();
      myB.doThree();
   }
// …

}
```

Example : Sequence/Communication Diagrams

- An example scenario:
  1. The message *makePayment* is sent to an instance of a *Register*.
  2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
  3. The *Sale* instance *creates* an instance of a *Payment*.

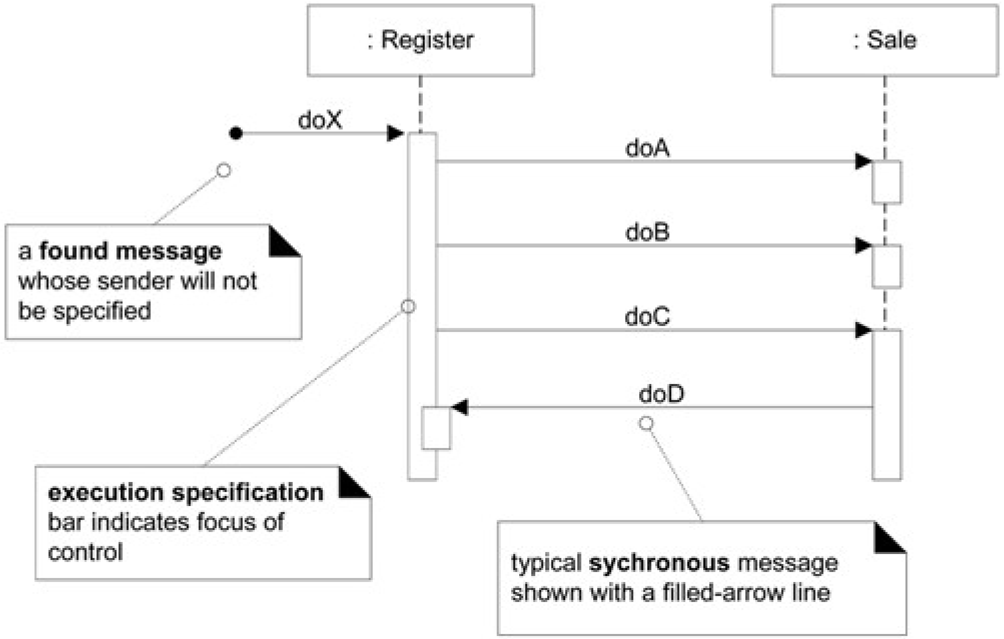




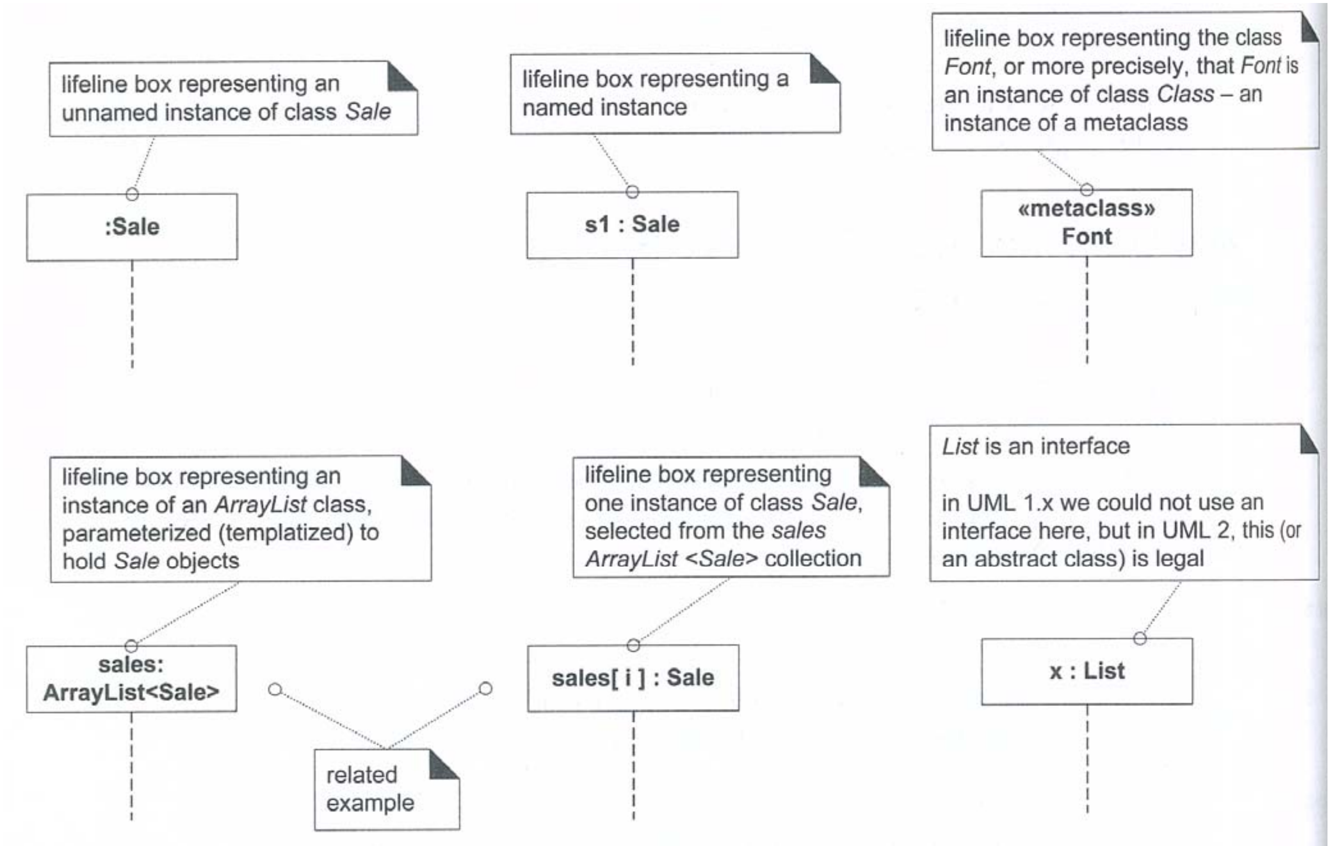

```
public class Sale
{
   private Payment payment;

   public void makePayment( Money cashTendered )
   {
      payment = new Payment( cashTendered );
      //…
   }
// …

}
```

Basic Sequence Diagram Notations

- Lifeline boxes and lifelines
- Messages

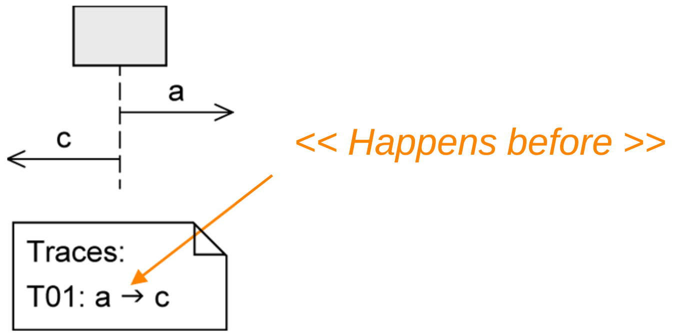


- Lifeline box
  - Represent the participants in the interaction, informally and practically
    - object(s), class, subsystem, component, etc.

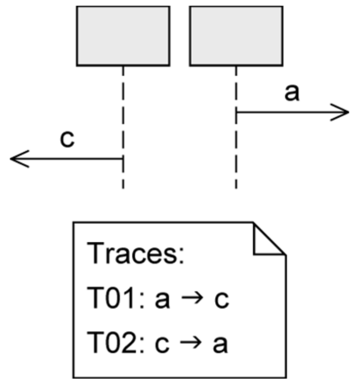


Order of Messages

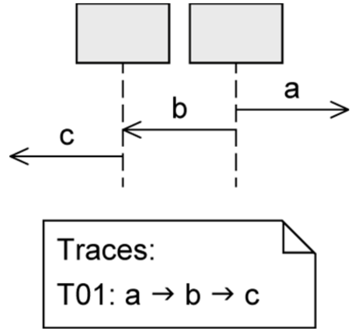
… on one lifeline



… on different lifelines

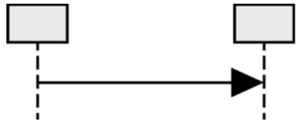


… on different lifelines which exchange messages

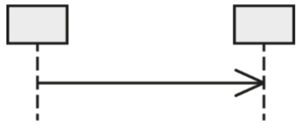


3 Types of Messages

- **Synchronous message**
  - Sender waits until it has received a response message before continuing.
  - An execution specification is inserted at target.



- **Asynchronous message**
  - Sender continues without waiting for a response message.

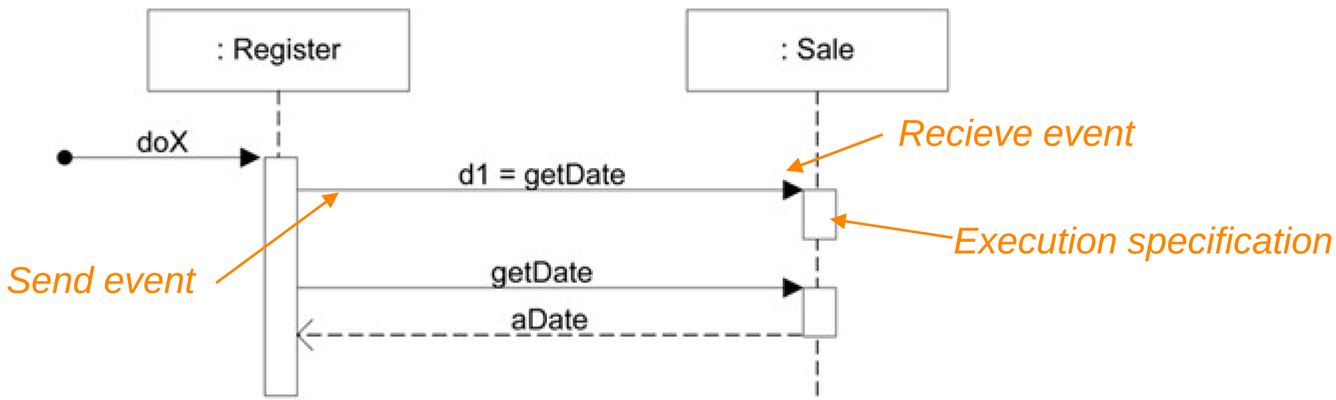


- **Response message**
  - May be omitted if content and location are obvious

Message Syntax

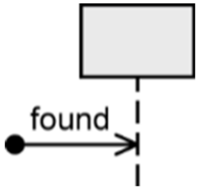
return = message (parameter: parameterType) : returnType

- For example:
  initialize(code)
  initialize
  d = getProductDescription(id)
  d = getProductDescription(id:ItemID)
  d = getProductDescription(id:ItemID) : ProductDescription

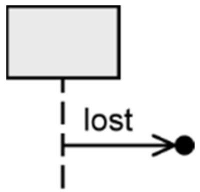


Other Types of Messages

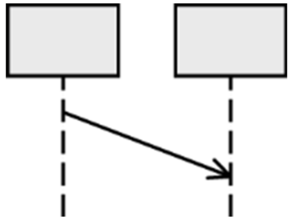
- Found message
  - Sender of a message is unknown or not relevant.

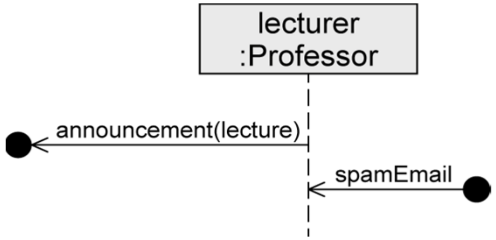


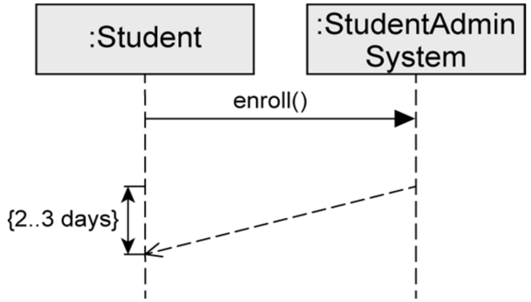
- Lost message
  - Receiver of a message is unknown or not relevant.



- Time-consuming message
  - Message with duration : Express that time elapses between the sending and the receipt of a message
  - Usually messages are assumed to be transmitted without any loss of time.

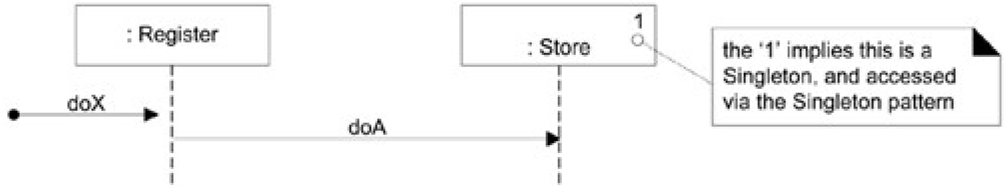






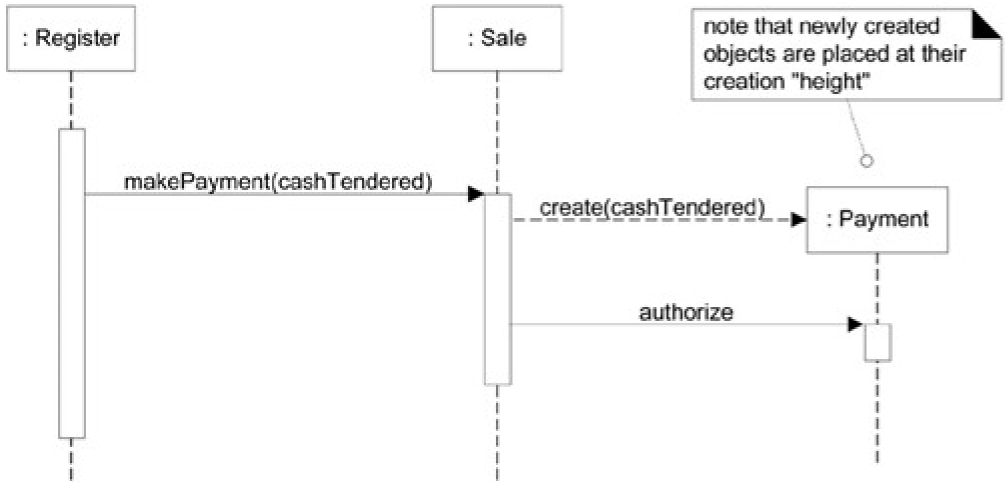
Singleton Objects

- There is only one instance of a class instantiated : **a singleton object**
  - Implying to **the Singleton design pattern**



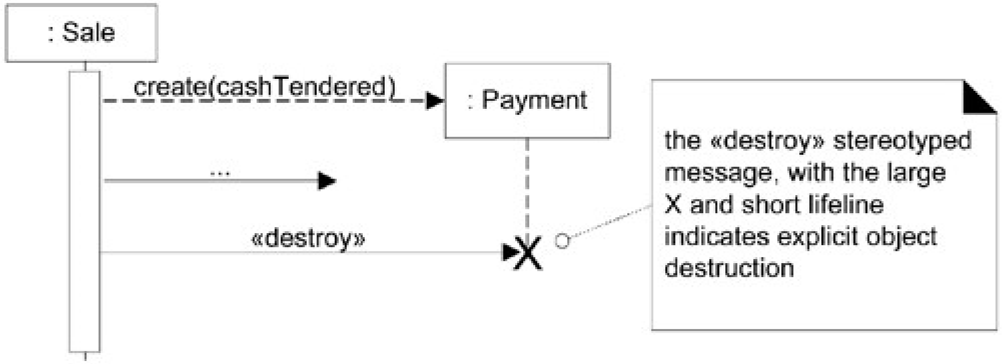
Instance Creation

- To create an instance of a class
  - The UML mandates dashed line.
  - The message name *create* is not required ; anything is legal.
    - But, it's a UML idiom.



Object Destruction

- To show explicit destruction of an object
  - The <<destroy>> stereotyped message, with the large X and short lifeline indicates explicit object destruction



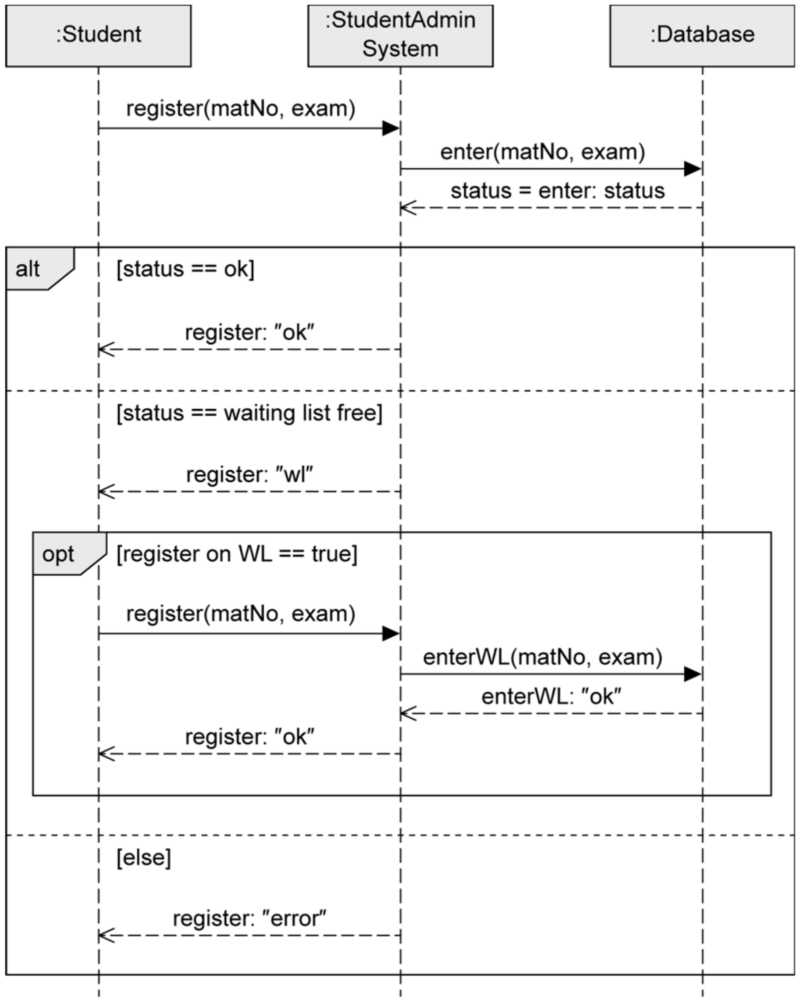
Combined Fragments and Operators

- 12 predefined types of **operators**
  - Model various control structures with frames
    - **Frames** : regions or fragments of the diagrams, which has an operator and a guard
  - Frames are nested.

	Operator	Purpose
Branches and loops	`alt`	Alternative interaction
	`opt`	Optional interaction
	`loop`	Repeated interaction
	`break`	Exception interaction
Concurrency and order	`seq`	Weak order
	`strict`	Strict order
	`par`	Concurrent interaction
	`critical`	Atomic interaction
Filters and assertions	`ignore`	Irrelevant interaction
	`consider`	Relevant interaction
	`assert`	Asserted interaction
	`neg`	Invalid interaction

`alt` Fragment

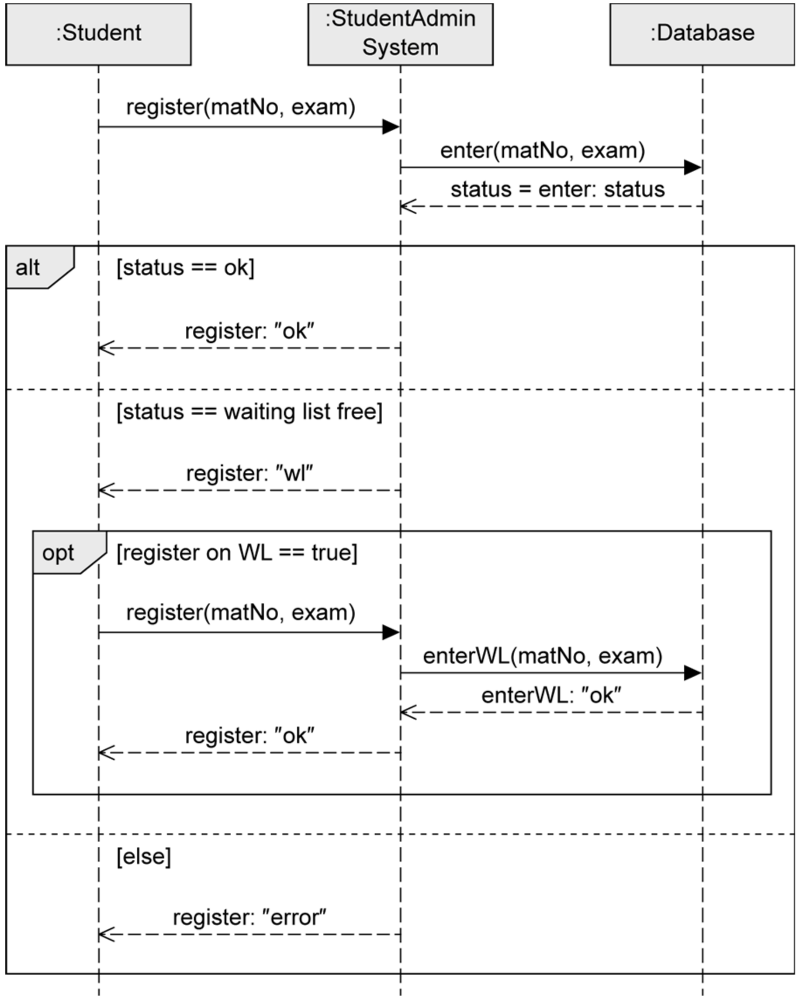
- To model alternative sequences
- Similar to switch statement in Java
  - Guards are used to select the one path to be executed.
  - Multiple operands
- Guards
  - Modeled in square brackets
  - default: true
  - predefined: [else]
- Guards have to be disjoint to avoid non-deterministic behavior.



Dependable Software Laboratory

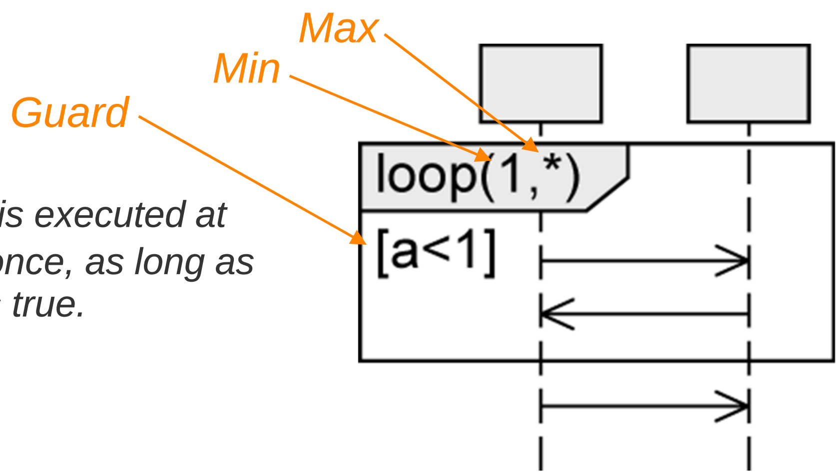
opt Fragment

- To model an optional sequence
- Similar to **if statement without else branch**
  - Exactly one operand
  - Actual execution at runtime is dependent on the guard.



`loop` Fragment

- To model <u>repeatedly-executed sequences</u>
  - Exactly one operand
- Keyword loop followed by the minimal/maximal number of iterations
  - (**`min..max`**) or (**`min,max`**)
  - default: `(*)` .. no upper limit
- Guard
  - Evaluated as soon as the minimum number of iterations has taken place
  - Checked for each iteration within the **`(min,max)`** limits
  - If the guard evaluates to false, the execution of the loop is terminated.



`loop` *is executed at least once, as long as* ***`a<1`*** *is true.*

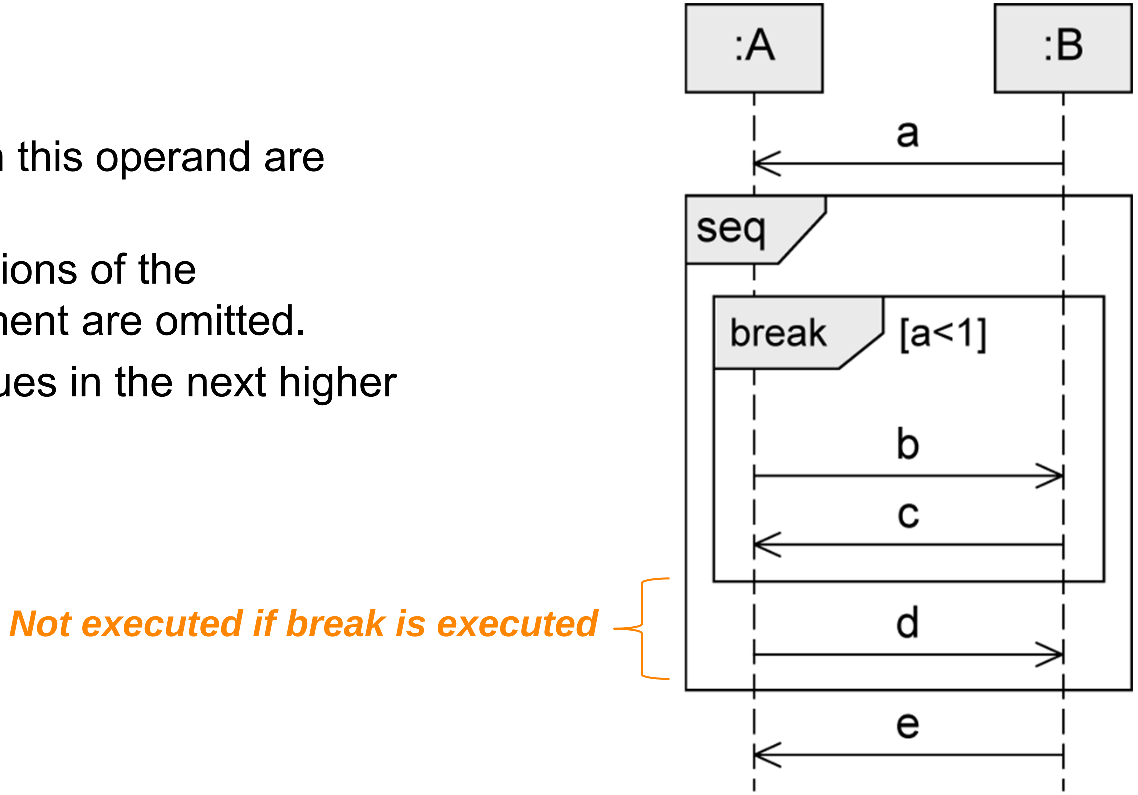
Notation alternatives:

```
loop(3,8) = loop(3..8)
loop(8,8) = loop (8)
loop = loop(*) = loop(0,*)
```

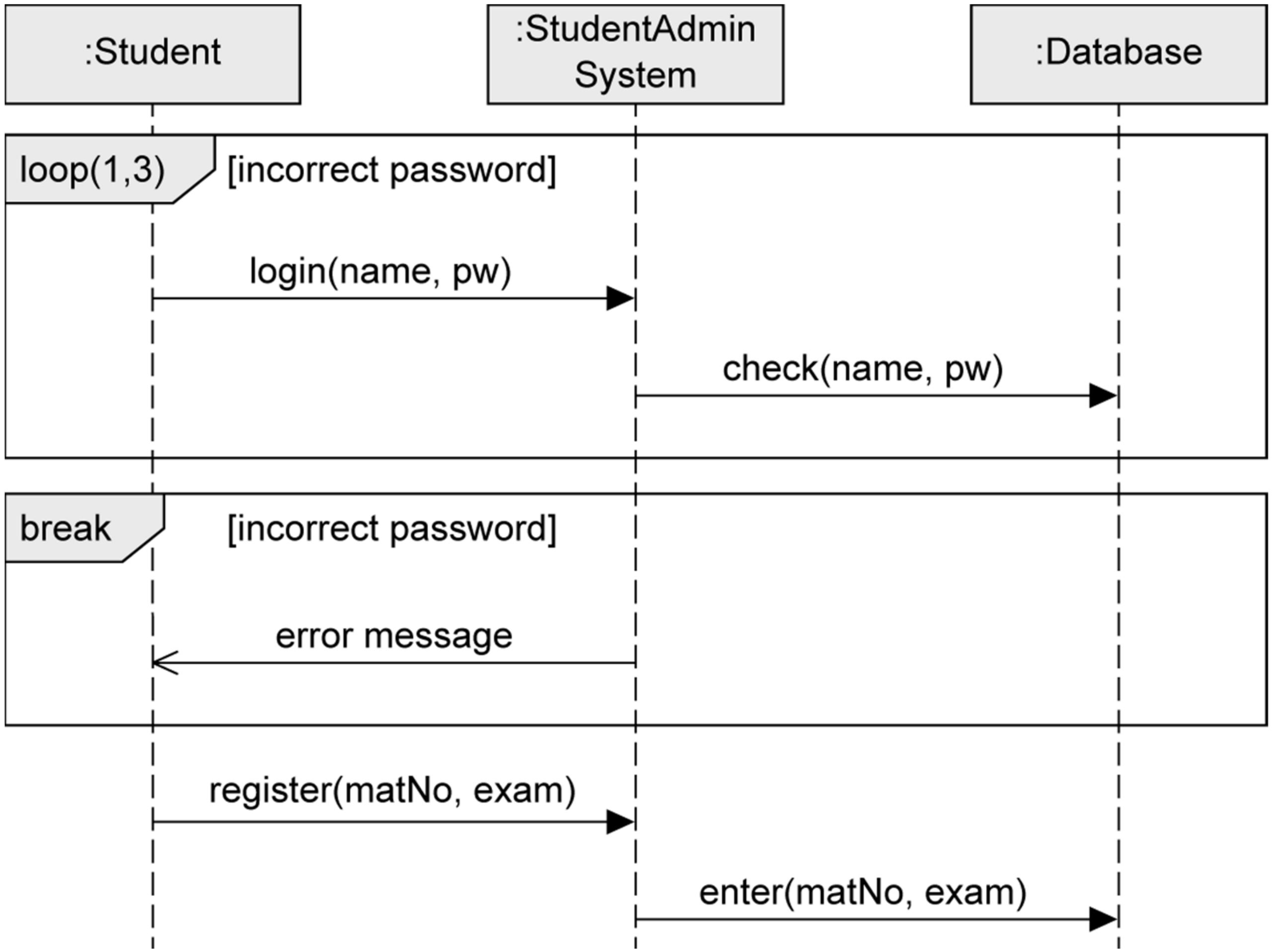
`break` Fragment

- Similar to exception handling
  - Exactly one operand with a guard

- If the guard is true:
  - Interactions within this operand are executed.
  - Remaining operations of the surrounding fragment are omitted.
  - Interaction continues in the next higher level fragment.

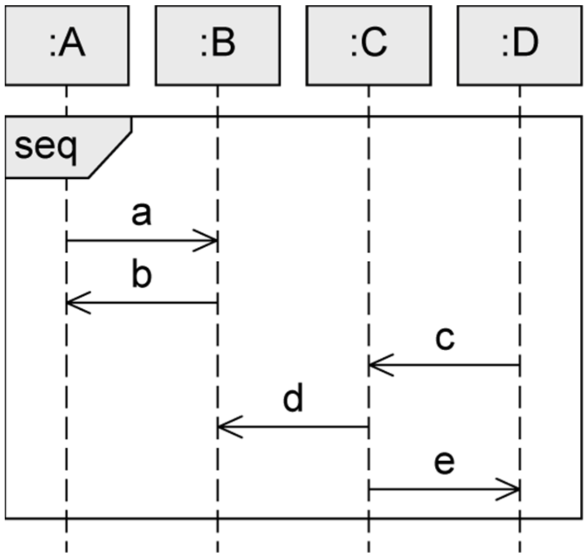


`loop` and `break` Fragment - Example



`seq` Fragment

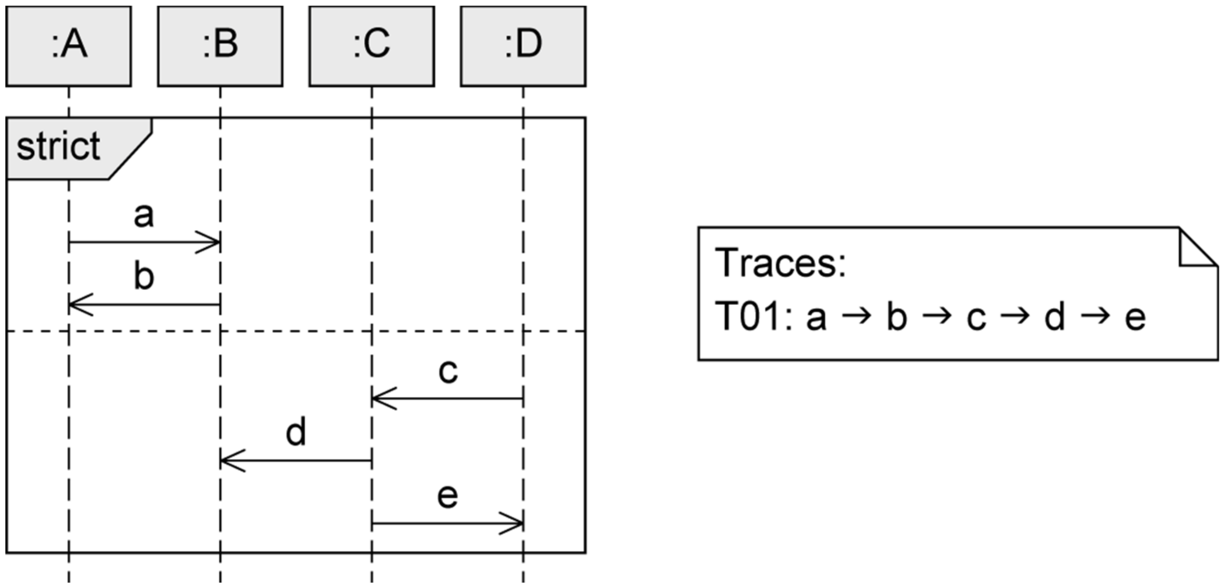
- <u>Default order of events</u>
- Weak sequencing:
    1. Events on different lifelines from different operands may come in any order.
    2. Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.



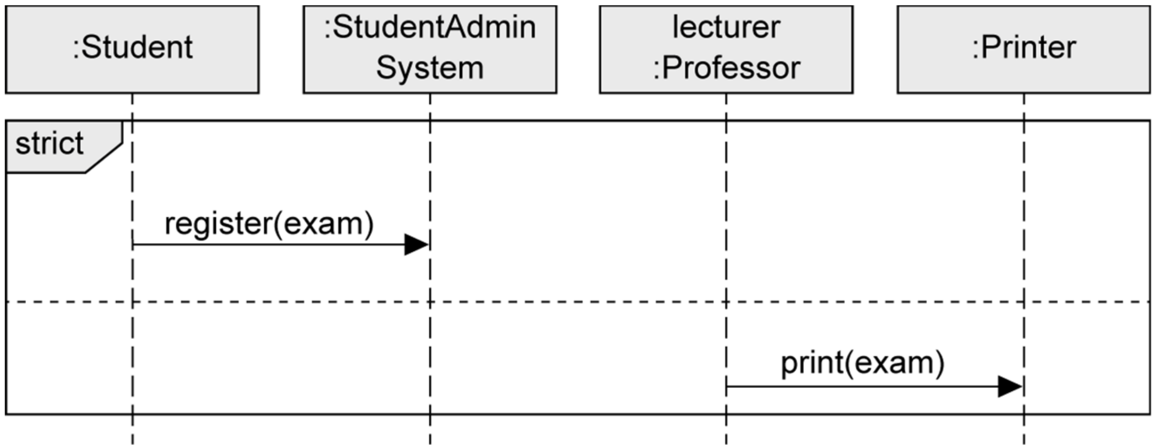
Traces:
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: c → a → b → d → e

`strict` Fragment

- <u>Sequential interaction</u> with order
  - Messages in an operand that is higher up on the vertical axis are always exchanged (executed) before the messages in an operand that is lower down on the vertical axis.

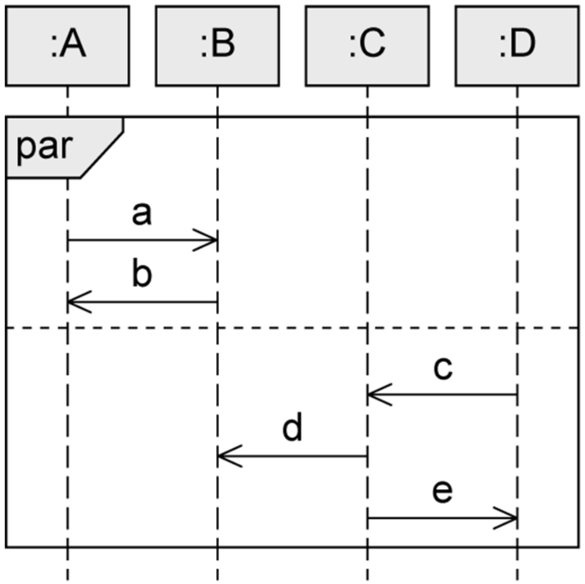


`strict` Fragment - Example



`par` Fragment

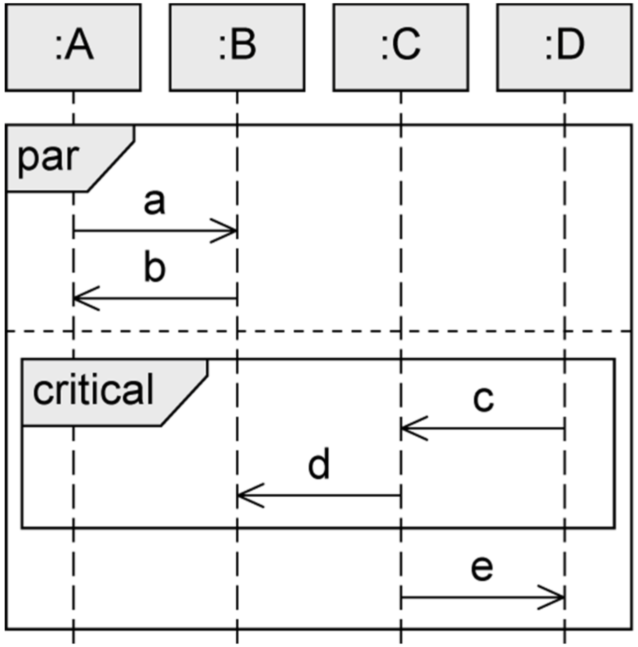
- To set aside chronological order between messages in different operands
  - Execution paths of different operands can be interleaved.
  - Restrictions of each operand are respected, but the order of the different operands is irrelevant
- Concurrency, no true parallelism



Traces:
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: a → c → d → b → e
T04: a → c → d → e → b
T05: c → a → b → d → e
T06: c → a → d → b → e
T07: c → a → d → e → b
T08: c → d → a → b → e
T09: c → d → a → e → b
T10: c → d → e → a → b

`critical` Fragment

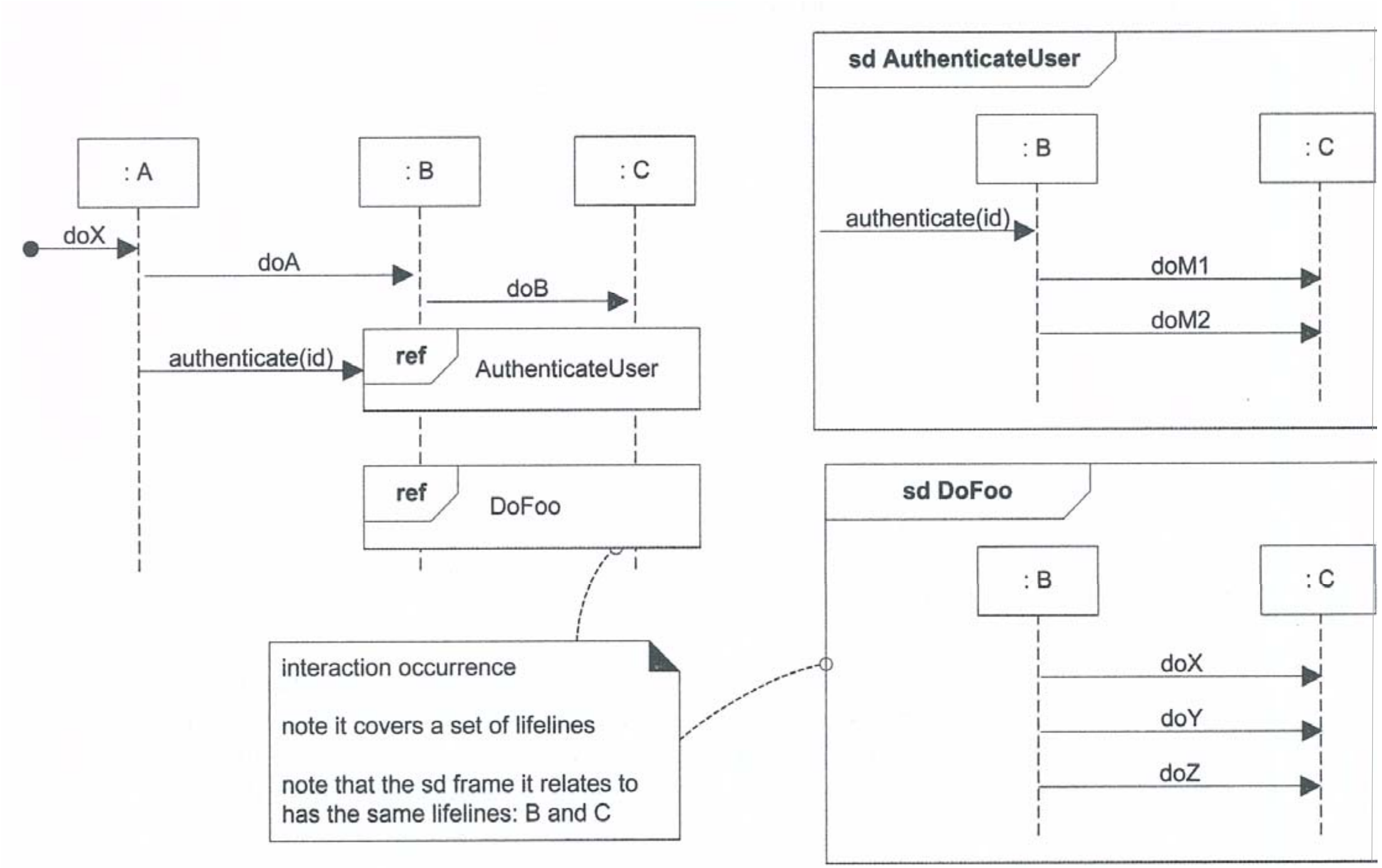
- <u>Atomic area</u> in the interaction
  - To make sure that certain parts of an interaction are not interrupted by unexpected events
  - Order within **`critical`** is the default order **`seq`**.



Traces:
T01: a → b → c → d → e
T02: a → c → d → b → e
T03: a → c → d → e → b
T04: c → d → a → b → e
T05: c → d → a → e → b
T06: c → d → e → a → b

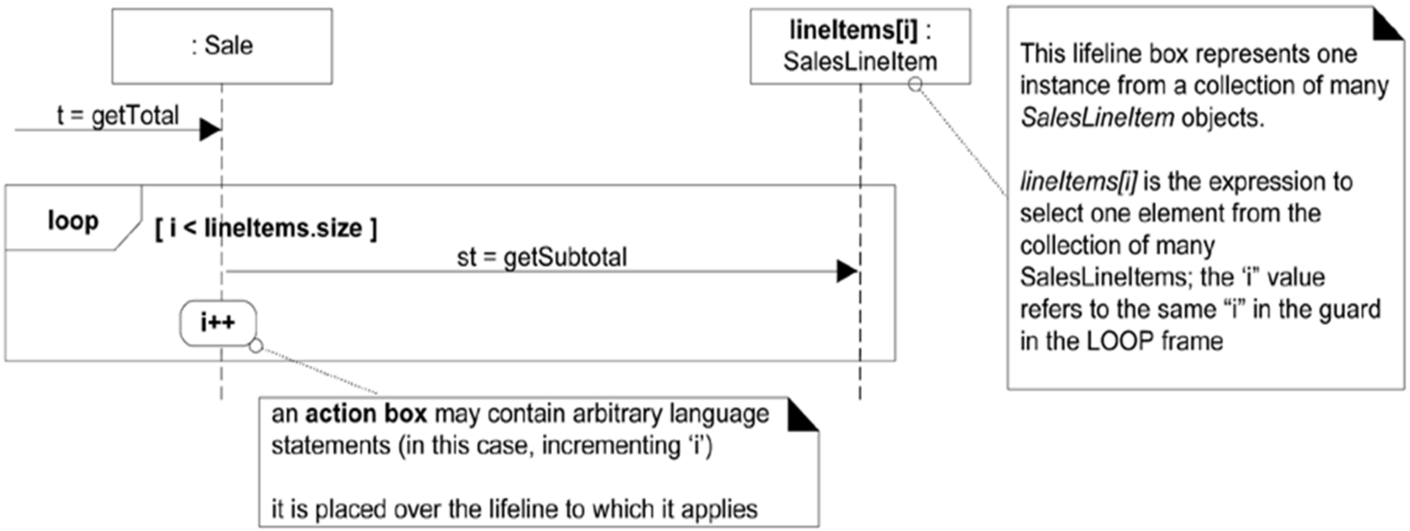
Interaction Reference

- Integrates one sequence diagram in another sequence diagram



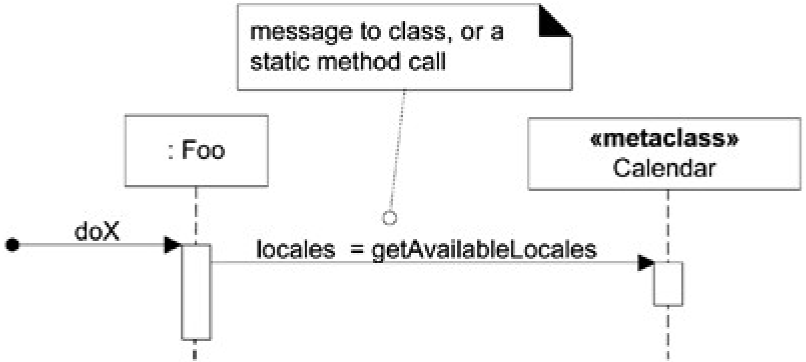
Iteration Over a Collection

- Sending the same message to each object to iterate over all members of a collection (such as a list or map).
  - The **selector** expression (as *lineItems[i]* in the lifeline) selects one object from a group.
  - Lifeline participants should represent one object, not a collection.



Messages to Classes to Invoke Static (or Class) Methods

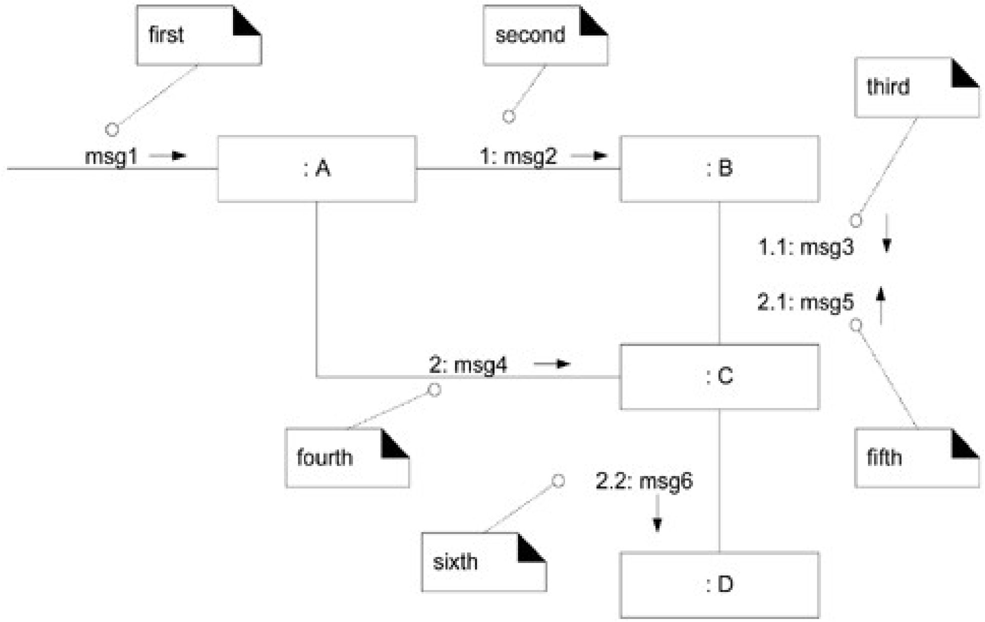
- You can show <u>class or static method calls</u> by
  - using a lifeline box label that indicates the receiving object is a class, or
  - more precisely, an instance of a **metaclass**




```
public class Foo
{
   public void doX()
   {
      // static method call on class Calendar
      Locale[] locales = Calendar.getAvailableLocales();
      // …
   }
// …
}
```

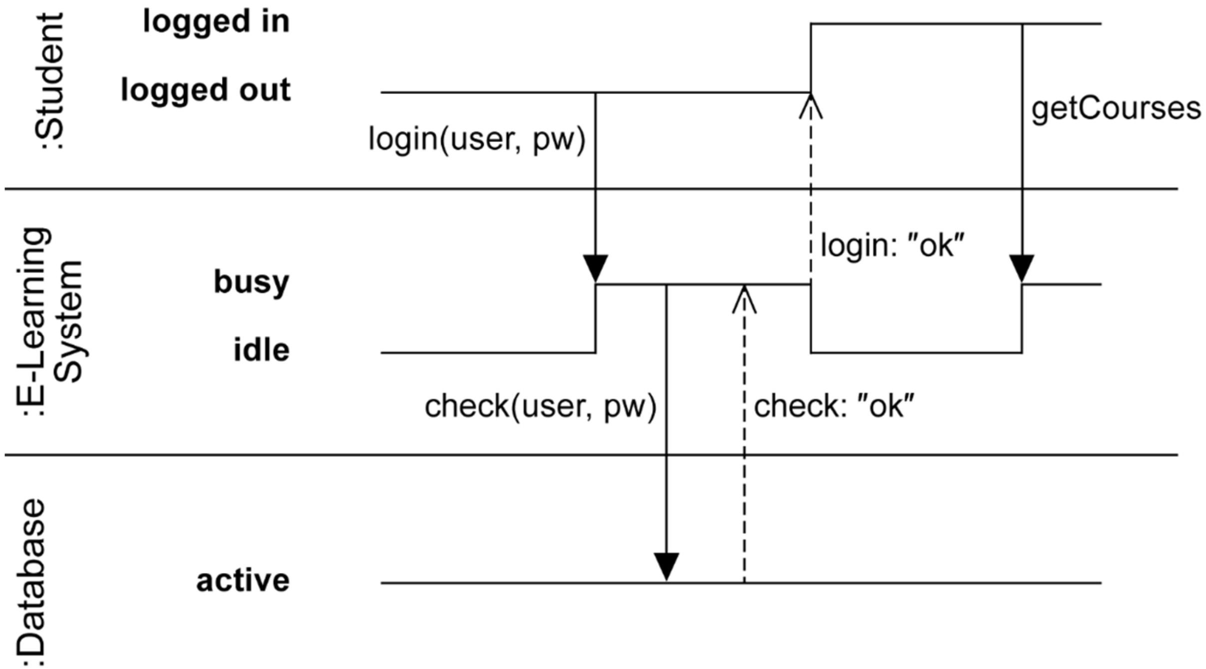
Basic Communication Diagram Notations

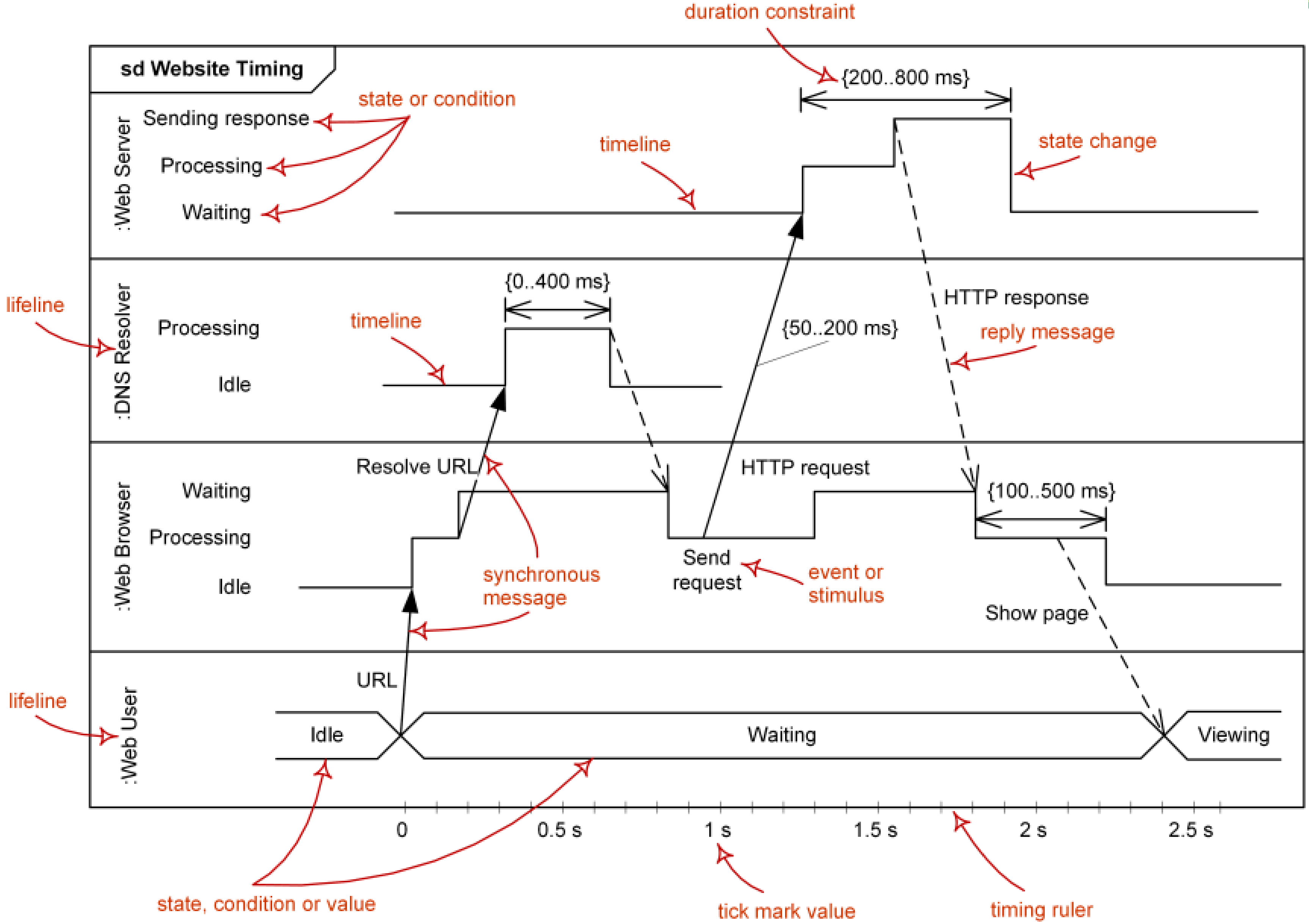
- **Link** and **Message**
  - A connection path between two objects indicating some form of possible navigation and visibility between the objects
  - All messages flow on the same line, and many messages may flow along a link.
    - Each message between objects is represented with a message expression and small arrow indicating the direction of the message.
    - A sequence number is added to show the sequential order of messages in the current thread of control.



Timing Diagram

- **Timing diagram**
  - Shows state changes of the interaction partners that result from the occurrence of events
    - Vertical axis: interaction partners
    - Horizontal axis: chronological order

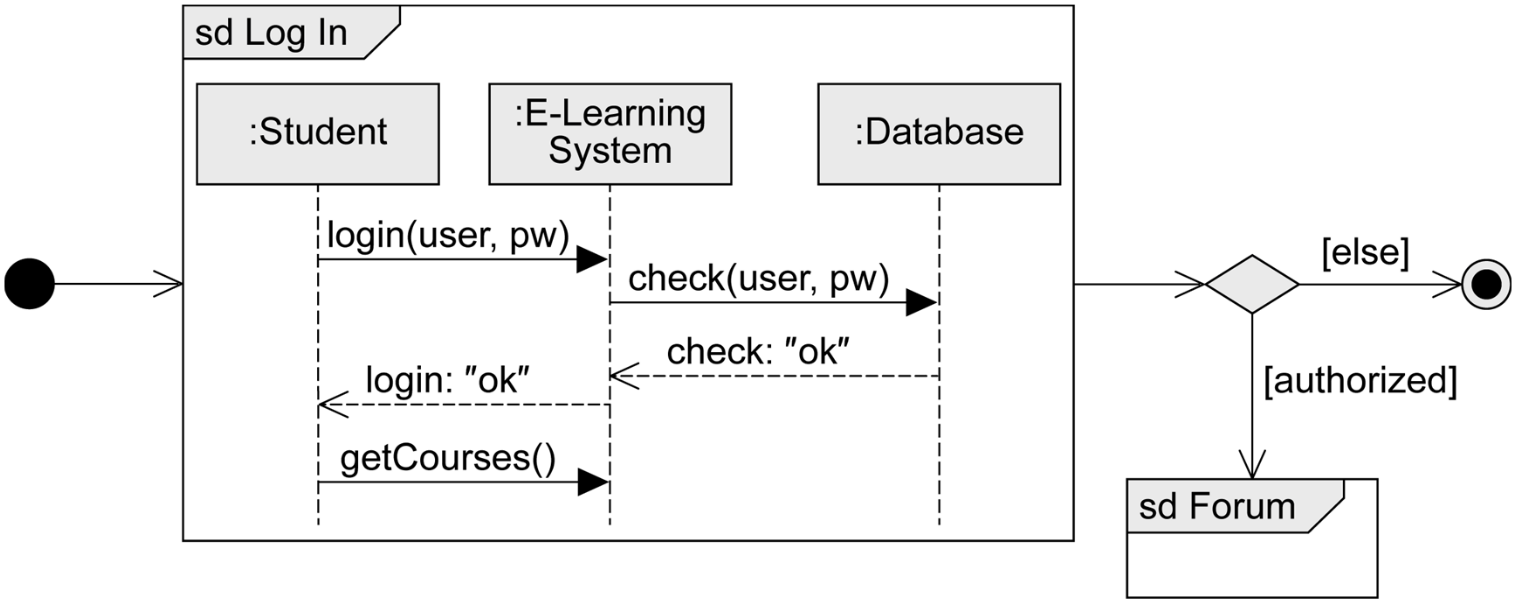




DEPENDABLE SOFTWARE LABORATORY

Interaction Overview Diagram

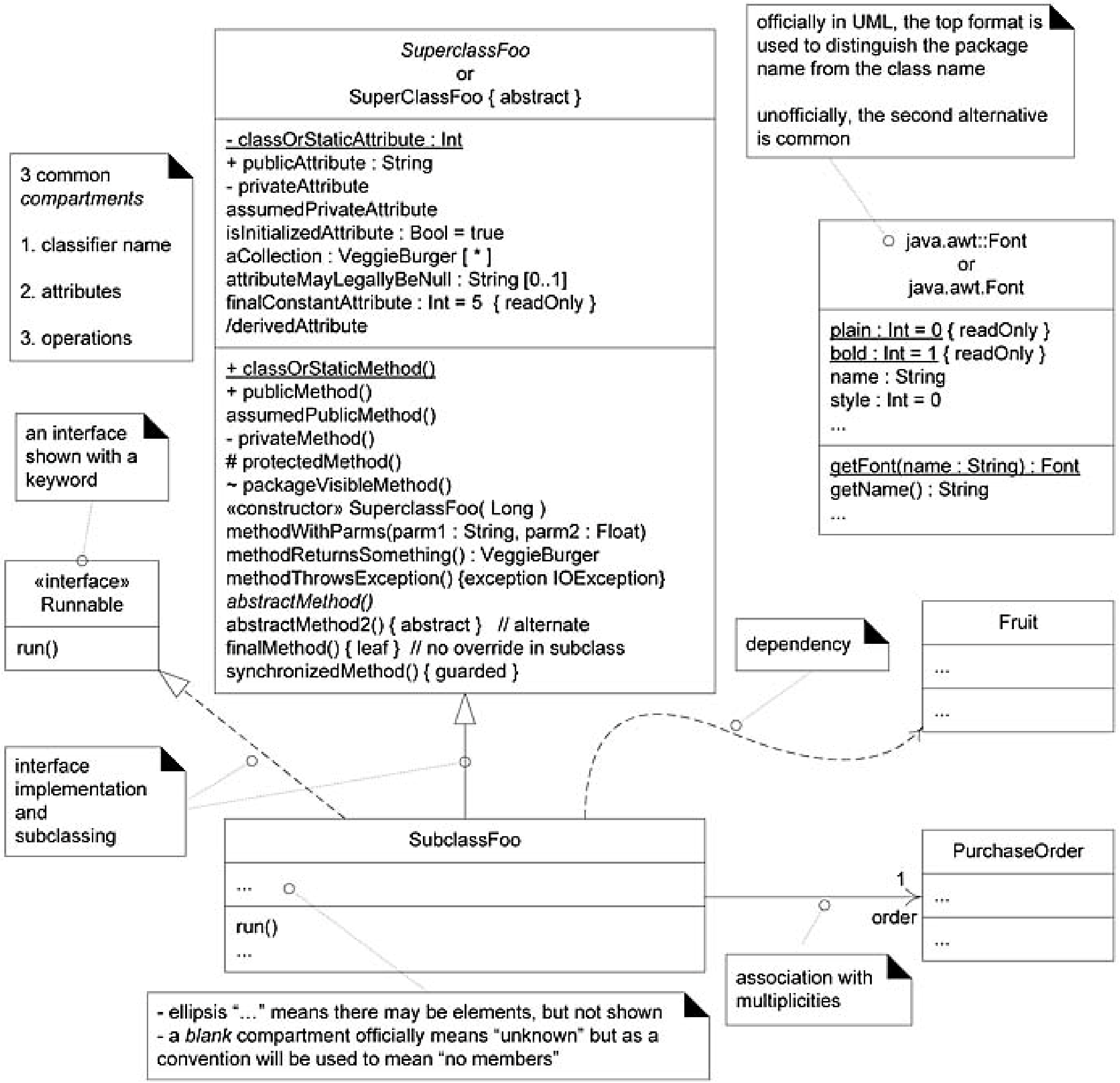
- **Interaction overview diagram**
  - Visualizes order of different interactions
  - Allows to place various interaction diagrams in a logical order
  - Basic notation concepts of activity diagram



Chapter 16.

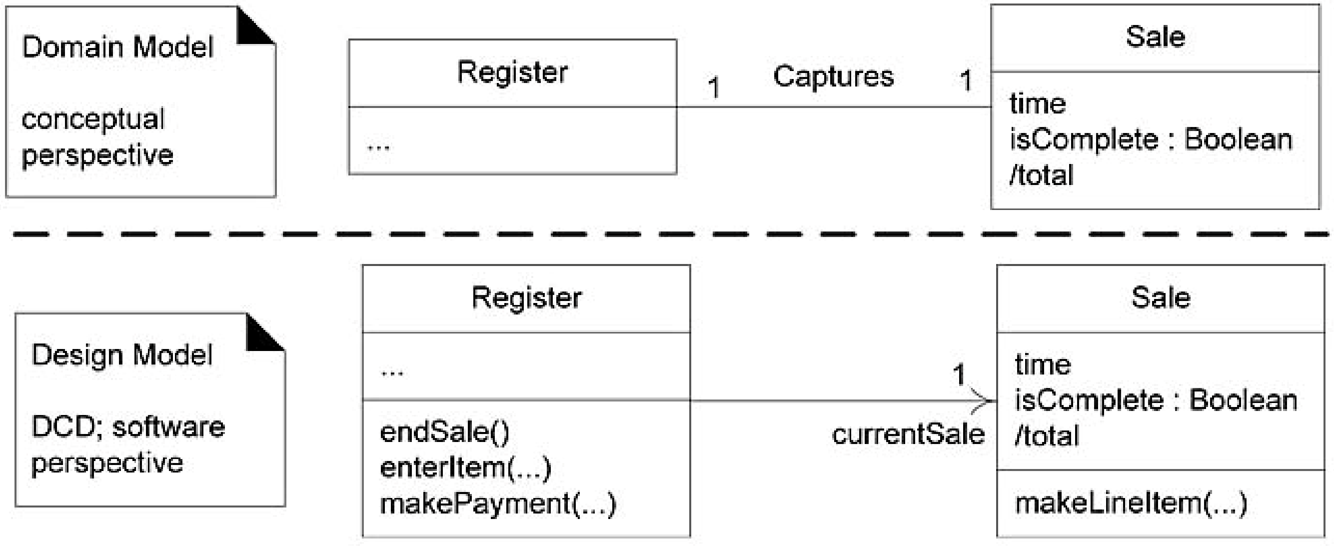
UML Class Diagram

Applying UML: Common Class Diagram Notation



Design Class Diagram

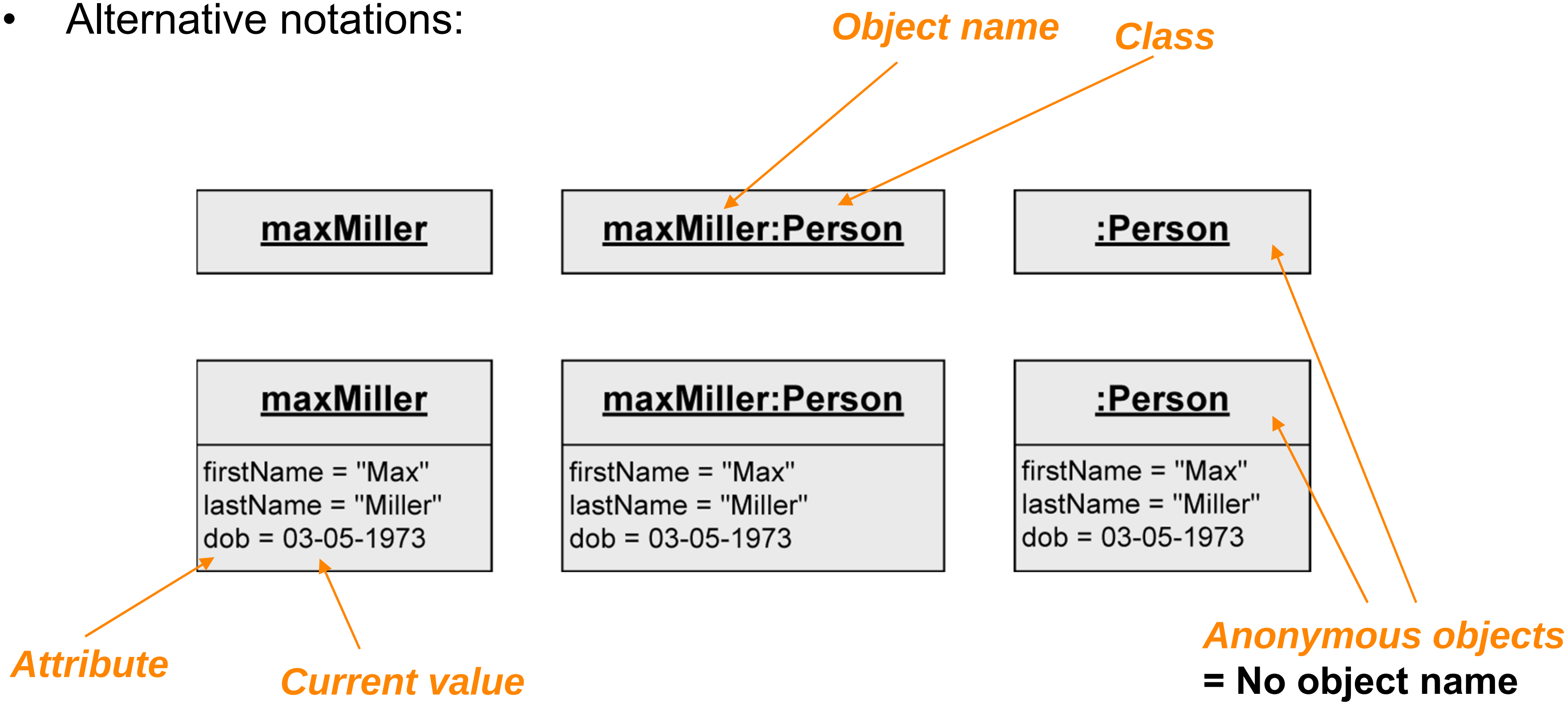
- The same UML class diagrams can be used in multiple perspectives.
  - In a conceptual perspective, **Domain model**
  - In a design perspective, **Design Class Diagram (DCD)**



Object

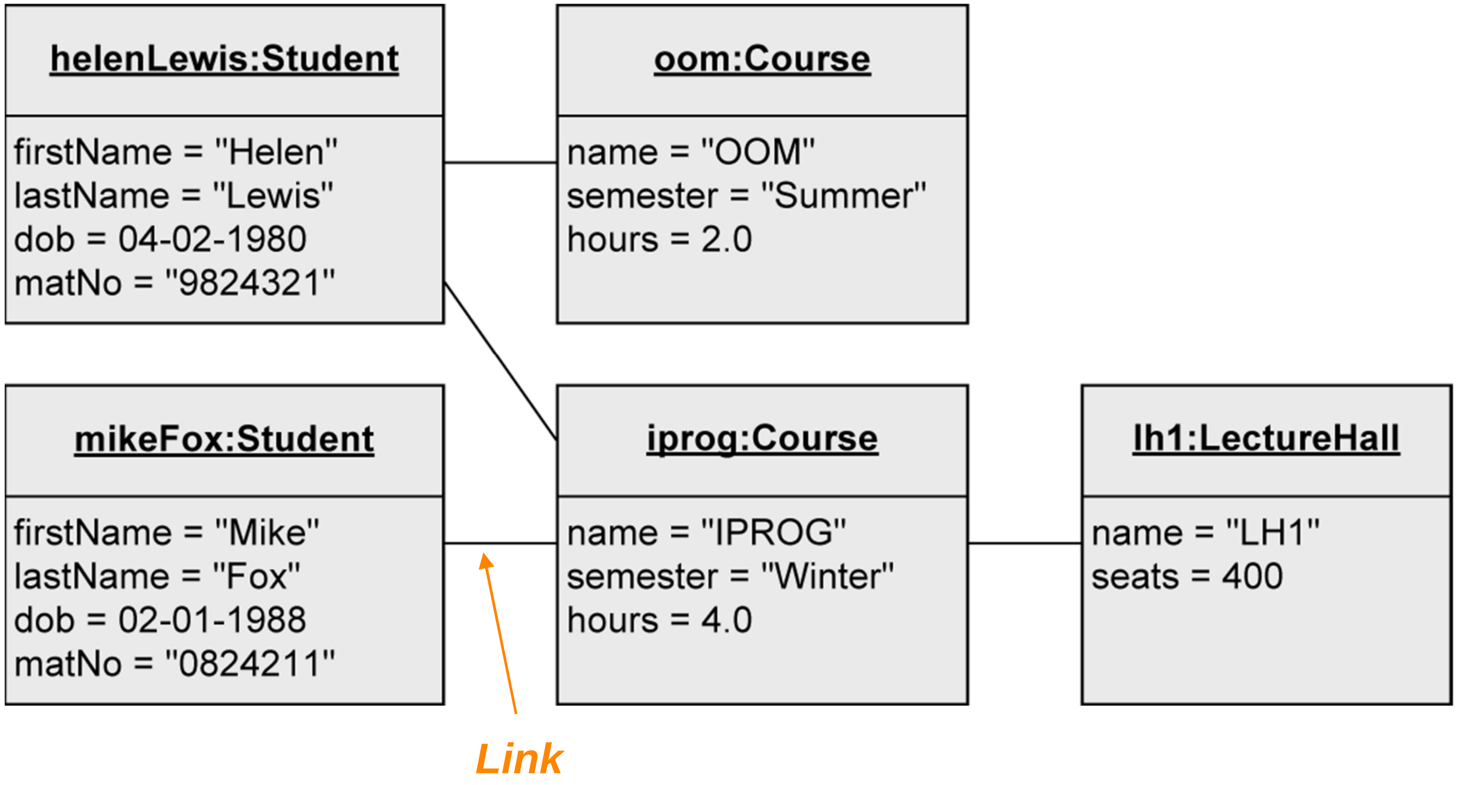
- Individuals of a system

- Alternative notations:



Object Diagram

- Depicts objects and their relationships at a specific moment in time

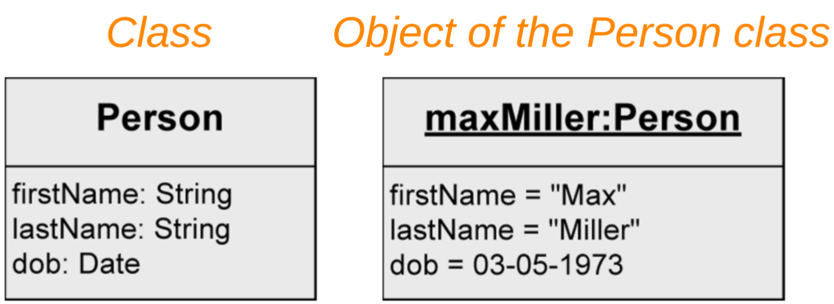


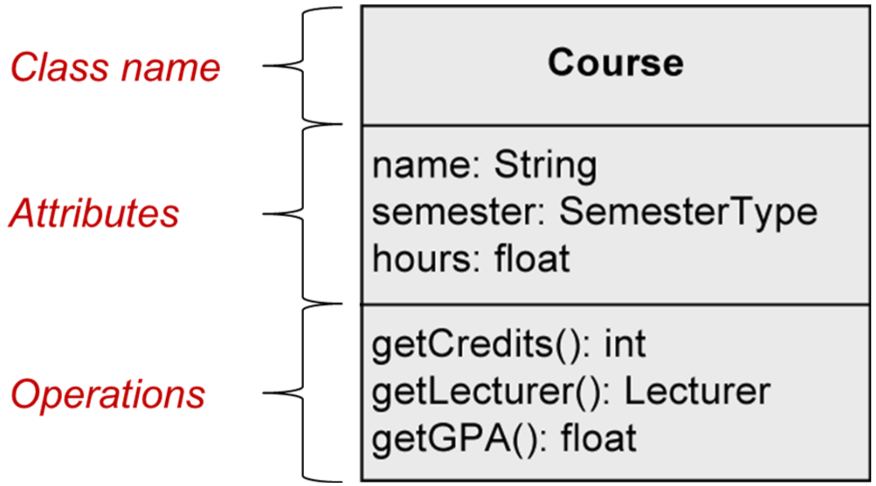
From Object to Class

- A class is a construction plan for a set of similar objects of a system.
  - Objects are instances of classes.

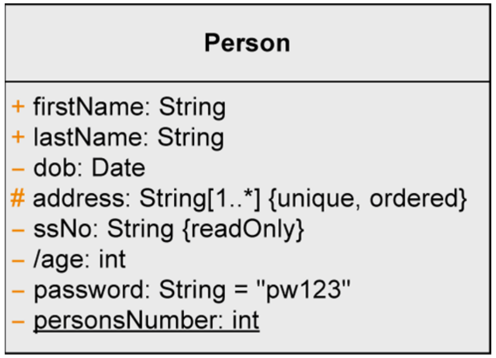
- **Attributes**: structural characteristics of a class
  - Different value for each instance (object)

- **Operations**: behavior of a class
  - Identical for all objects of a class
    → not depicted in object diagram



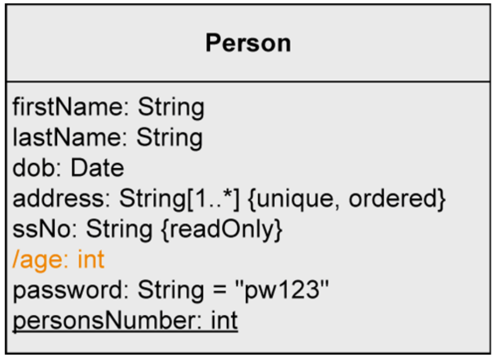


Attribute Syntax - Visibility



- Who is permitted to access the attribute.
  - + ... public: everybody
  - - ... private: only the object itself
  - # ... protected: class itself and subclasses
  - ~ ... package: classes that are in the same package

Attribute Syntax - Derived Attribute



- Attribute value is derived from other attributes or associations.
  - `age`: calculated from the date of birth

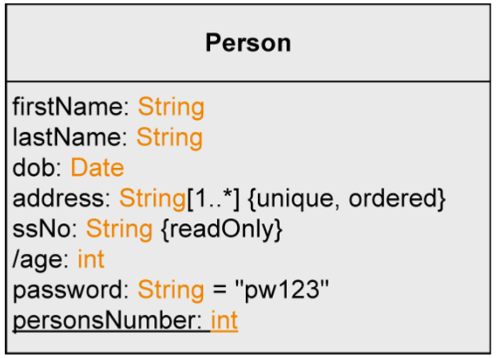
Attribute Syntax - Name

Person

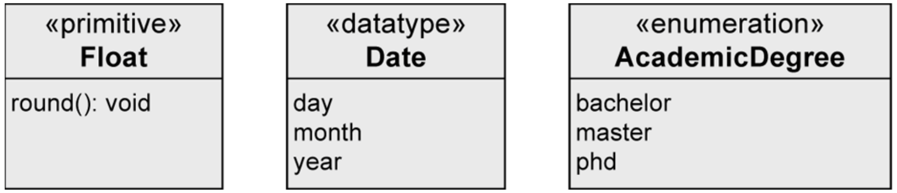
firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

- Name of the attribute

Attribute Syntax - Type



- Types of attributes
  - Data types
    - Primitive data type
      - Pre-defined: Boolean, Integer, Unlimited Natural, String
      - User-defined: «**primitive**»
      - Composite data type: «**datatype**»
    - Enumerations: «**enumeration**»



  - User-defined classes

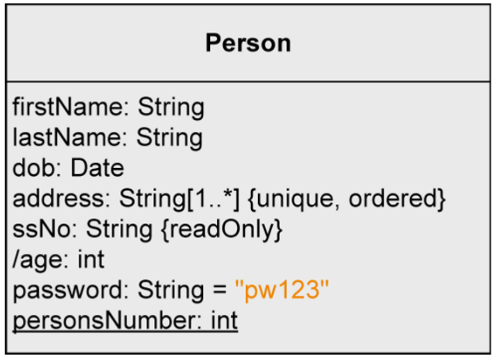
Attribute Syntax - Multiplicity

Person

firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
<u>personsNumber: int</u>

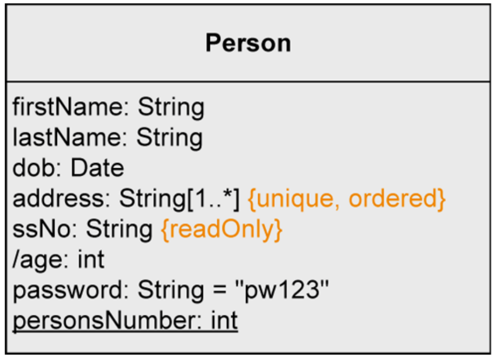
- Number of values which an attribute may contain
  - Default value: 1
- Notation: [min..max]
  - no upper limit: [*] or [0..*]

Attribute Syntax - Default Value



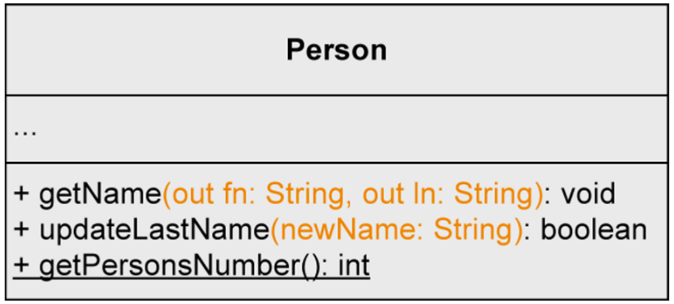
- Default value
  - Used if the attribute value is not set explicitly by the user

Attribute Syntax - Properties



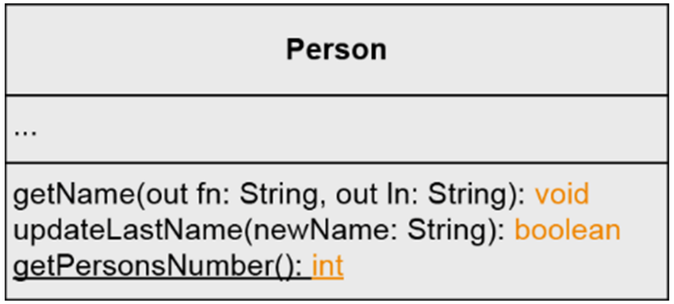
- Pre-defined properties
  - {readOnly} … value cannot be changed
  - {unique} … no duplicates permitted
  - {non-unique} … duplicates permitted
  - {ordered} … fixed order of the values
  - {unordered} … no fixed order of the values
- Attribute specification
  - Set: {unordered, unique}
  - Multi-set (Bag): {unordered, non-unique}
  - Ordered set: {ordered, unique}
  - List: {ordered, non-unique}

Operation Syntax - Parameters



- Notation similar to attributes
- Direction of the parameter
  - in … input parameter
    - When the operation is used, a value is expected from this parameter
  - out … output parameter
    - After the execution of the operation, the parameter has adopted a new value
  - inout : combined input/output parameter

Operation Syntax - Type



- Types of the return value

Operations and Methods

- **Operations**
  - The full official format of the operation syntax :
    - **visibility name (parameter-list) {property-string}**
    - **visibility name (parameter-list) : return-type {property-string}** ← **UML1.X**
  - Guidelines
    - Assume that the new version includes a return type.
    - Operations are usually assumed public if no visibility is shown.

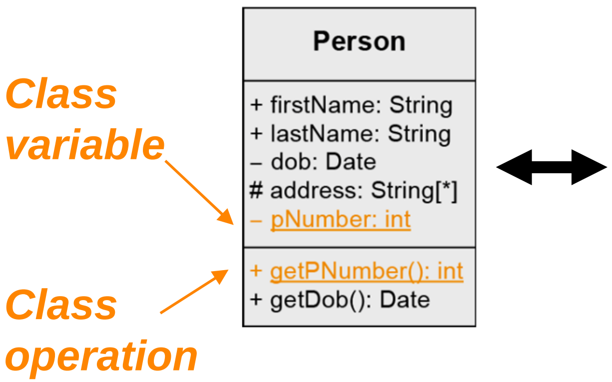
- An operation is not a method.
  - **A UML operation is a declaration**, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre-and post-conditions.
  - Not an implementation - rather, **methods are implementations**.

Note Symbols

- **A UML note symbol** may represent several things, such as:
  - UML **note** or **comment**, which by definition have no semantic impact
  - UML **constraint**, in which case it must be encased in braces '{…}'
  - **Method body** : the implementation of a UML operation

Class Variable and Operation

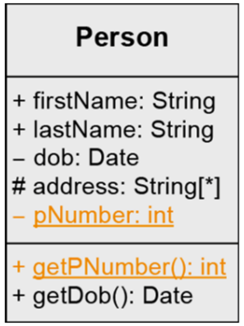
- **Instance variable** (= instance attribute) : attributes defined on instance level
- **Class variable** (= class attribute, static attribute)
  - Defined only once per class, *i.e.*, shared by all instances of the class
  - Example: counters for the number of instances of a class
- **Class operation** (= static operation)
  - Can be used, if no instance of the corresponding class was created
  - Example: constructors, counting operations, etc.




```
class Person {

        public String firstName;
        public String lastName;
        private Date dob;
        protected String[ ] address;
        private static int pNumber;

        public static int getPNumber() {…}
        public Date getDob() {…}
}
```

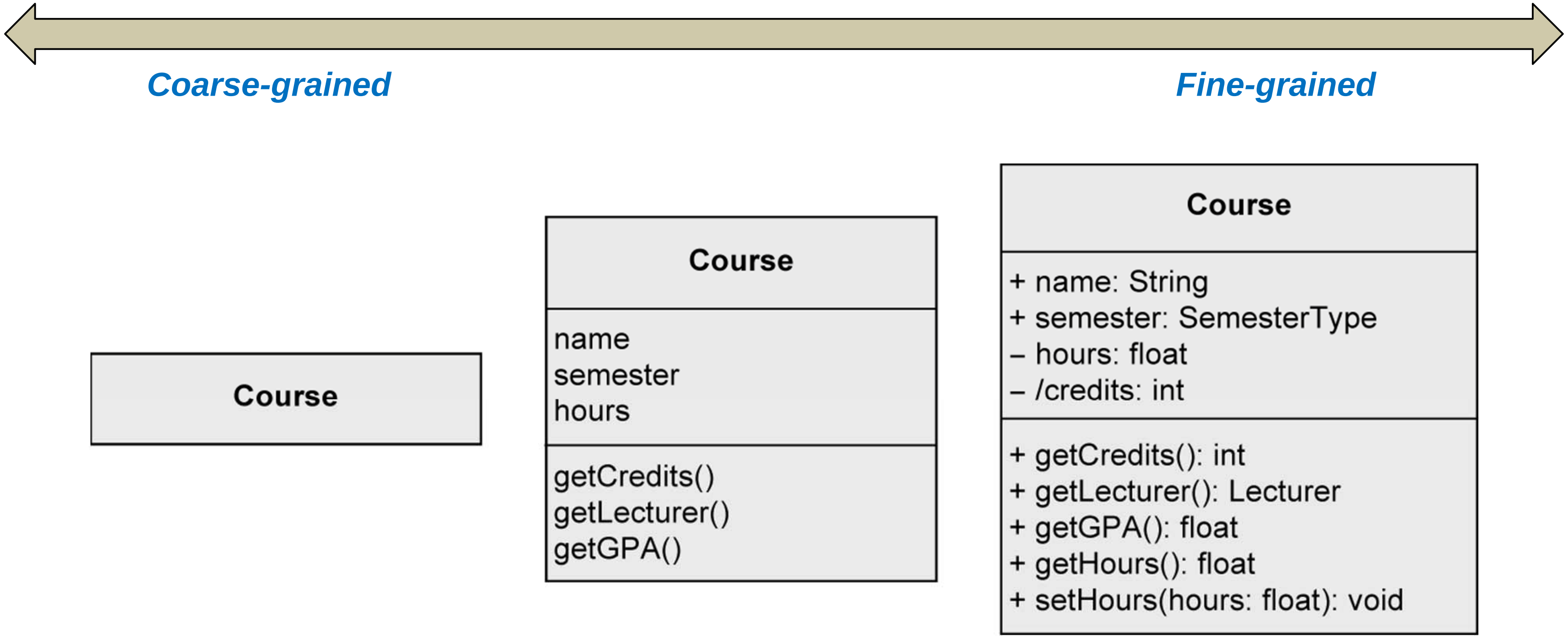
Operations to Access Attributes in DCDs

- **Accessing operations** to retrieve or set all (private) attributes
  - Example: *getPNumber()* and *setPNumber()*

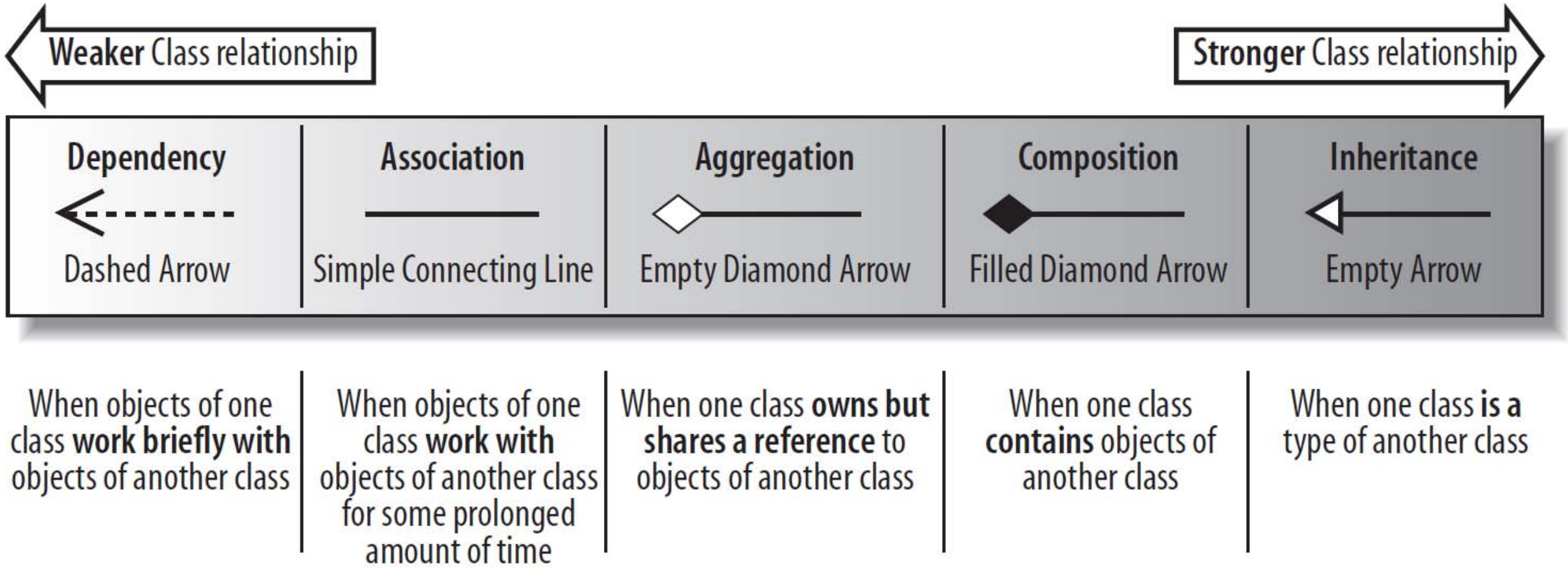


  - Often excluded (or filtered) from the class diagram, since they are too many.
    - For n attributes, there may be 2n uninteresting **getter** and **setter** operations.
  - Most UML tools support filtering their display.

Different Levels of Class Detail

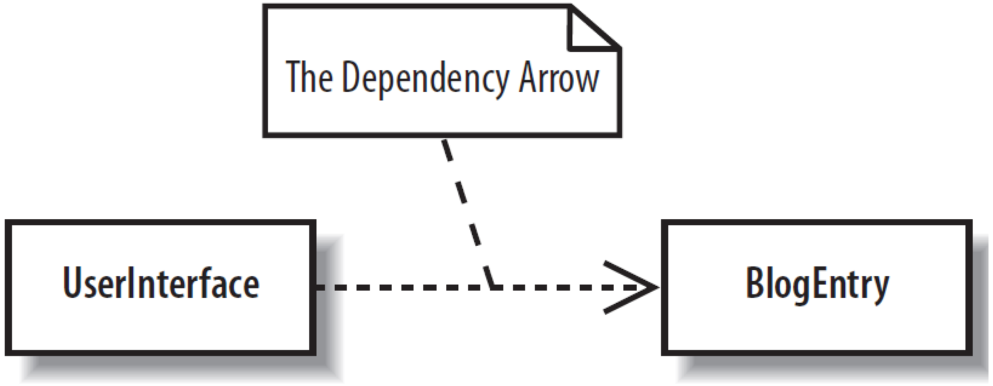


Types of Class Relationship



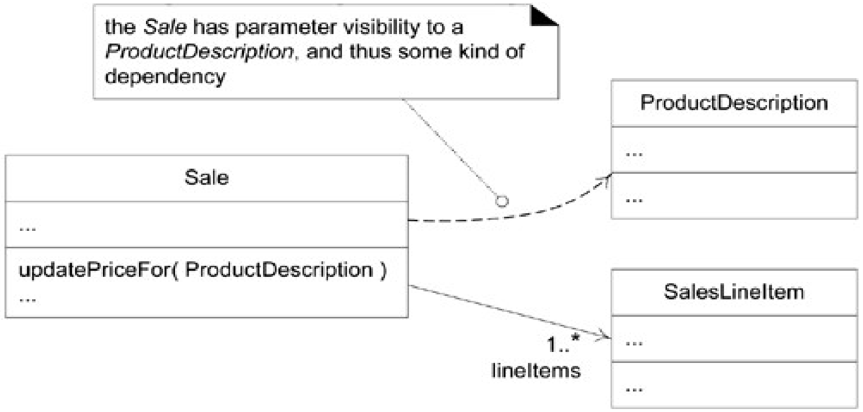
Dependency

- Models weakest possible relationships between classes
  - A class needs to know about another class to use objects of that class briefly.
  - Not used often in class diagram, but does in component diagram.



Dependency - Example

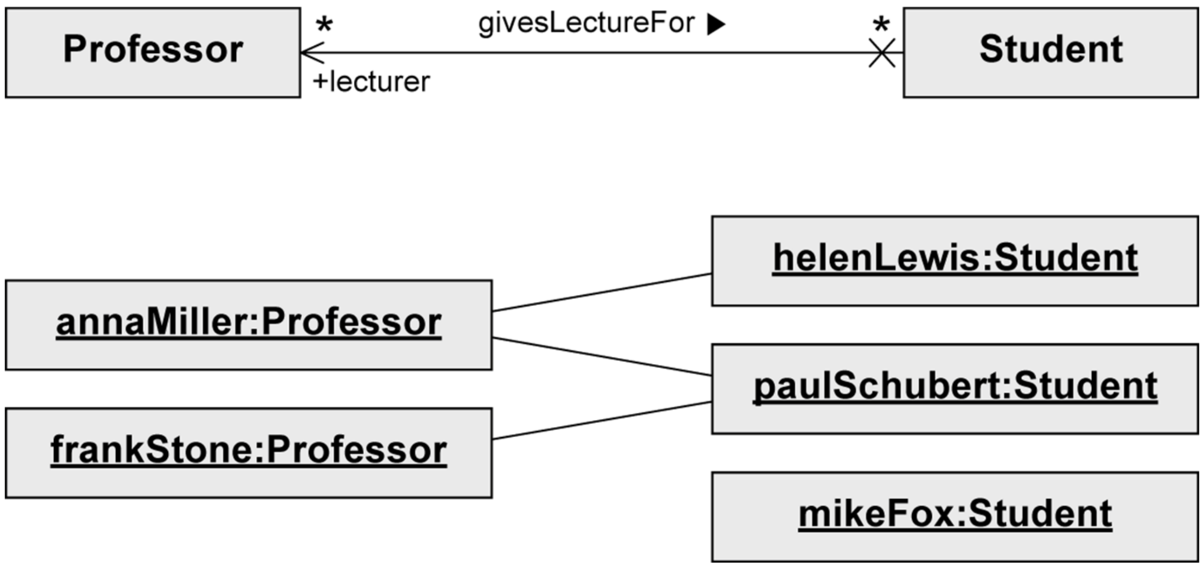
- Example:
  - The *updatePriceFor* method receives a *ProductDescription* parameter object and then sends it a *getPrice* message.
  - Therefore, the *Sale* object has parameter visibility to the *ProductDescription*, and message-sending coupling, and thus a dependency on the *ProductDescription*.
  - If the latter class changed, the *Sale* class could be affected.




```
public class Sale
{
   public void updatePriceFor( ProductDescription description )
   {
      Money basePrice = description.getPrice();
      //…
   }
// …
}
```

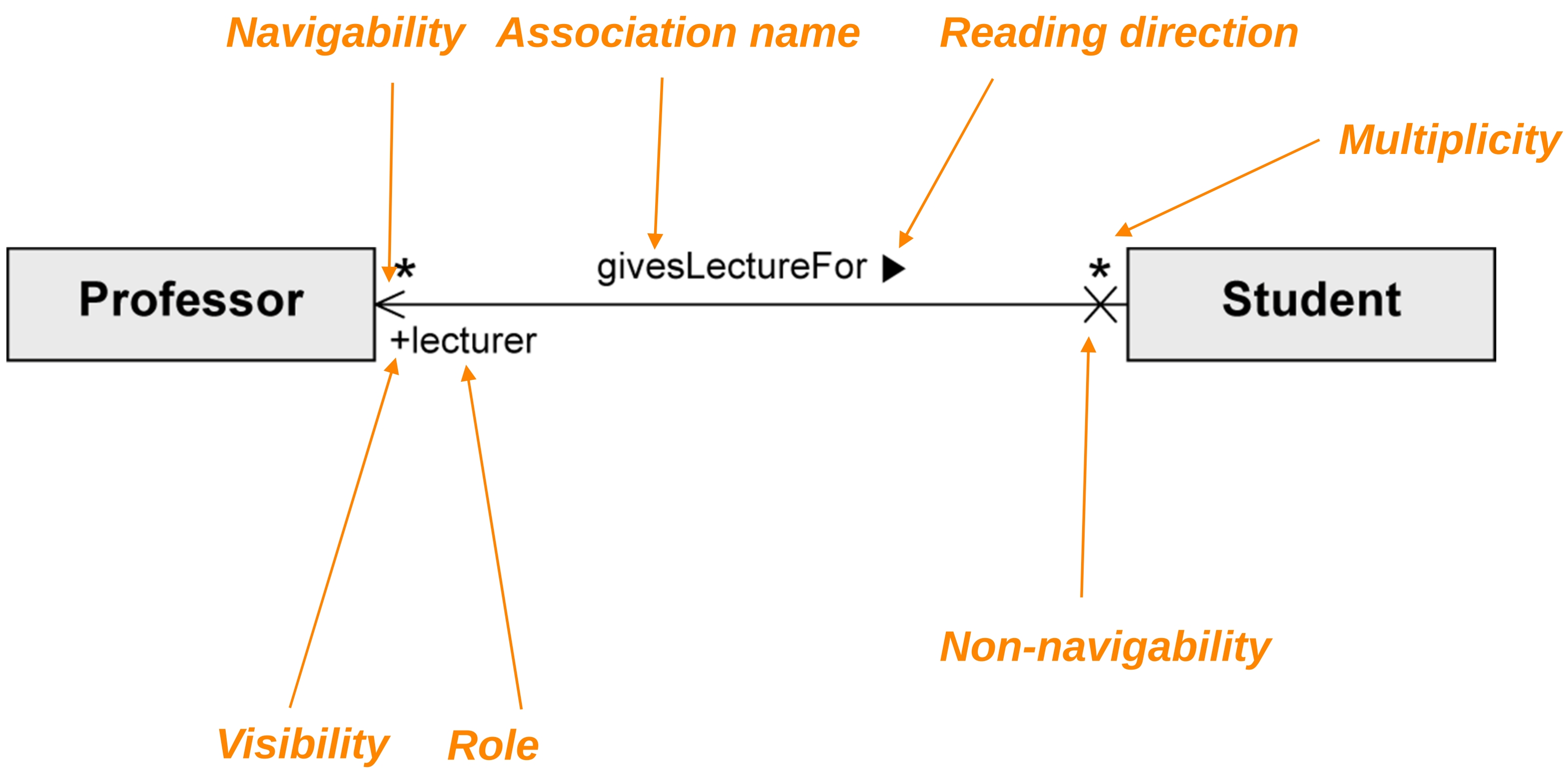
Association

- Models possible relationships between instances of classes
  - When objects of one class work with objects of another class for some prolonged amount of time.



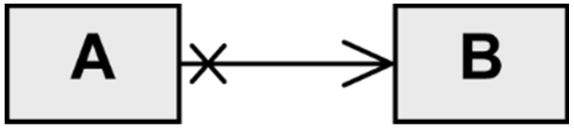
Binary Association

- Connects instances of two classes with one another



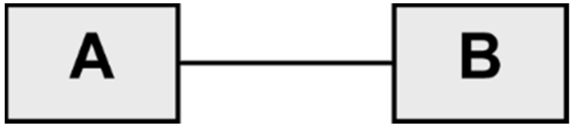
Binary Association - Navigability

- Navigability
  - An object knows its partner objects and can therefore access their visible attributes and operations.
  - Indicated by open arrow head or cross

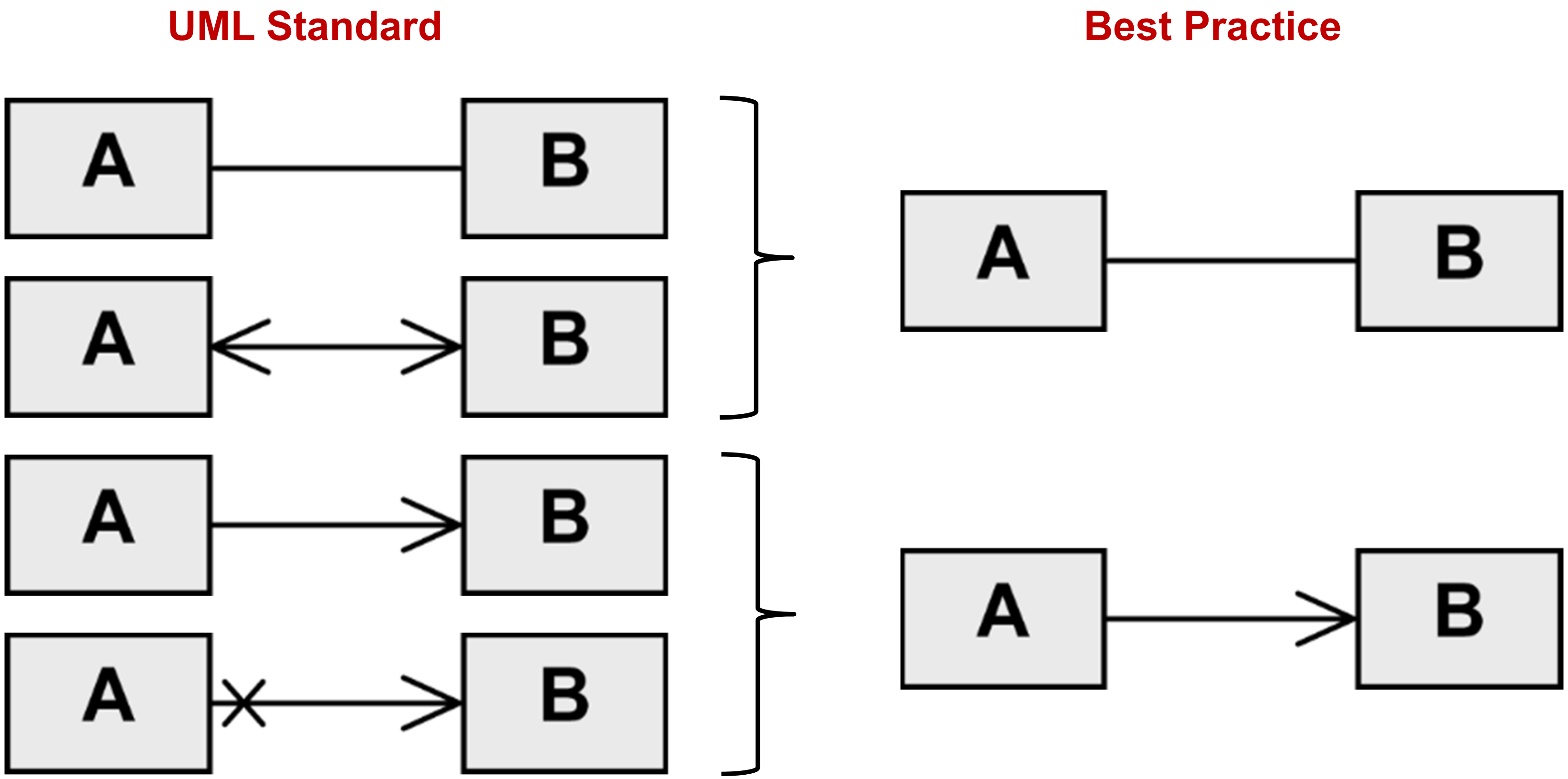


- Example:
  - "**A** can access the visible attributes and operations of **B**"
  - "**B** cannot access any attributes and operations of **A**"

- Navigability undefined
  - Bidirectional navigability is assumed.

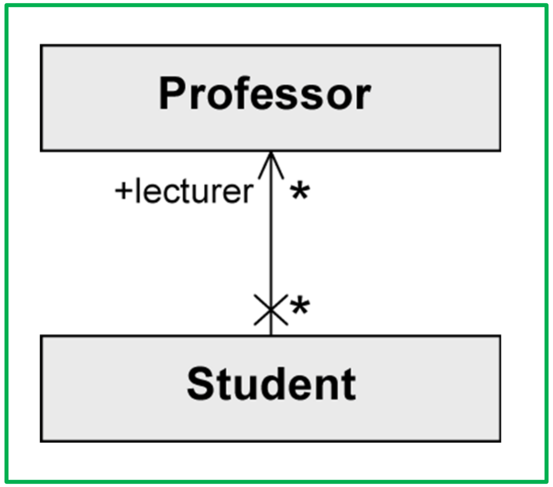


Navigability - UML Standard vs. Best Practice

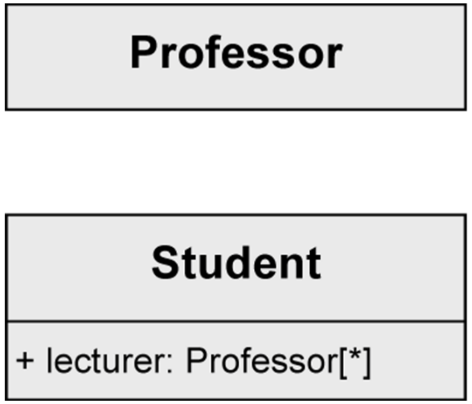


Binary Association as Attribute

Preferable



=



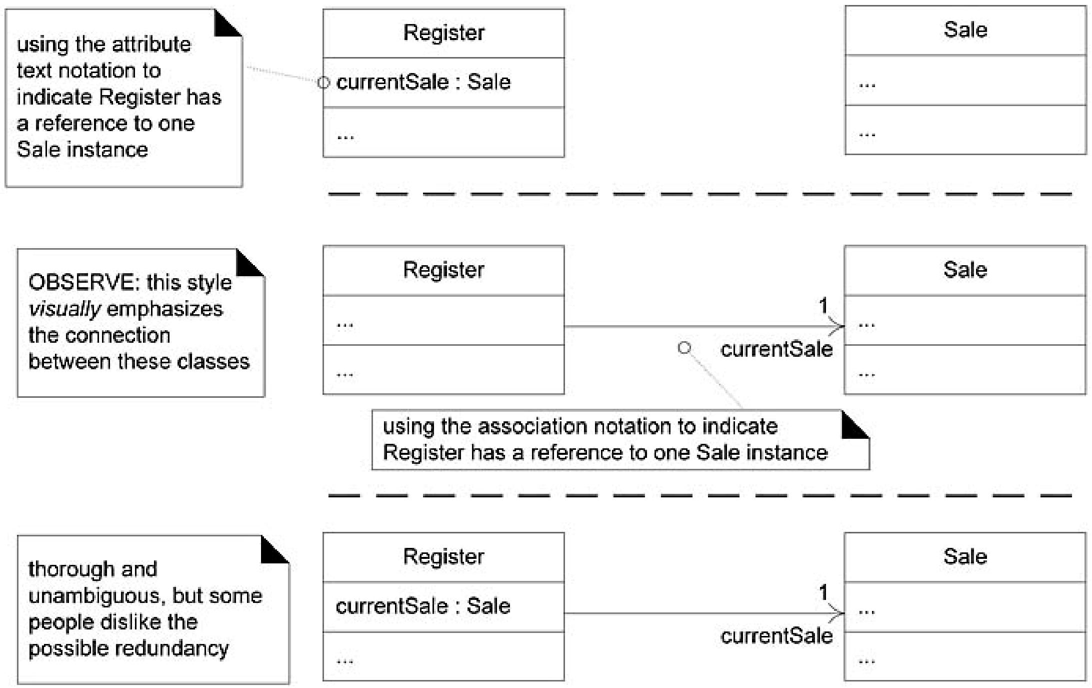
- Java-like notation:

```
class Professor {…}

class Student {
   public Professor[] lecturer;
   …
}
```

Ways to Show UML Attributes

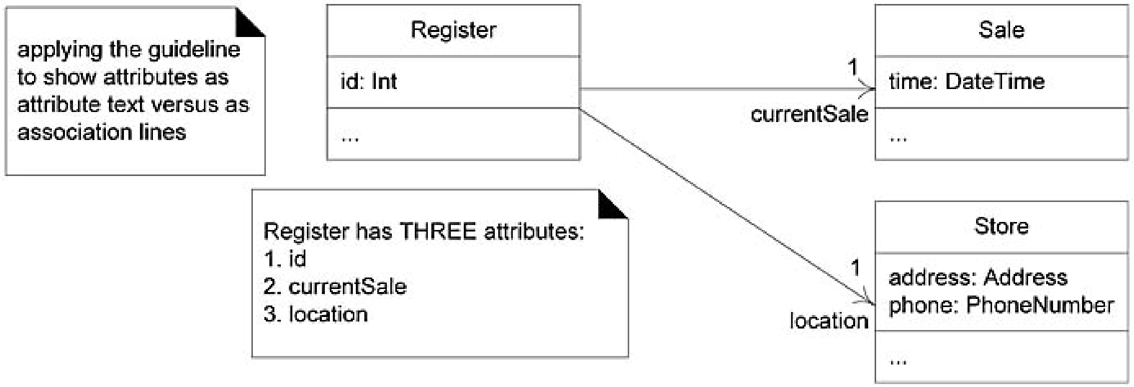
- Attributes can be shown in three ways:

  1. attribute text
     - ***visibility name : type multiplicity = default {property-string}***
  2. association line
     - **a navigability arrow**
     - **multiplicity**
     - **a role name**
  3. both together



Attribute Text vs. Association Lines for Attributes

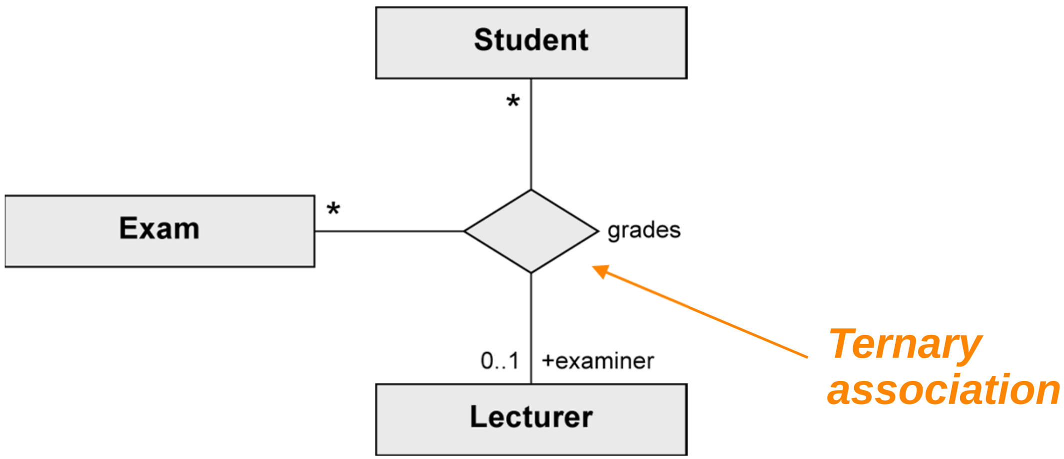
- Use the attribute text notation for data type objects, while the association line notation for others.
  - Both are semantically equal.
  - But, showing an association line to another class box in the diagram gives visual emphasis.




```
public class Register
{
   private int id;
   private Sale currentSale;
   private Store location;
   // …
}
```

n-ary Association

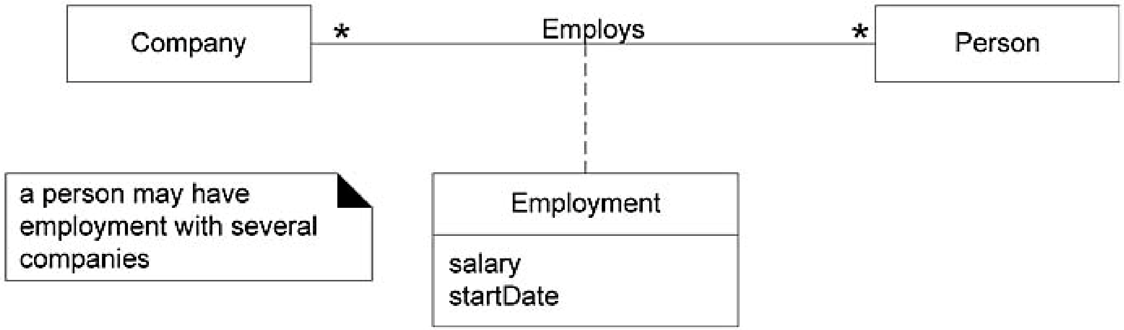
- More than two partner objects are involved in the relationship.
  - No navigation directions



Association Class

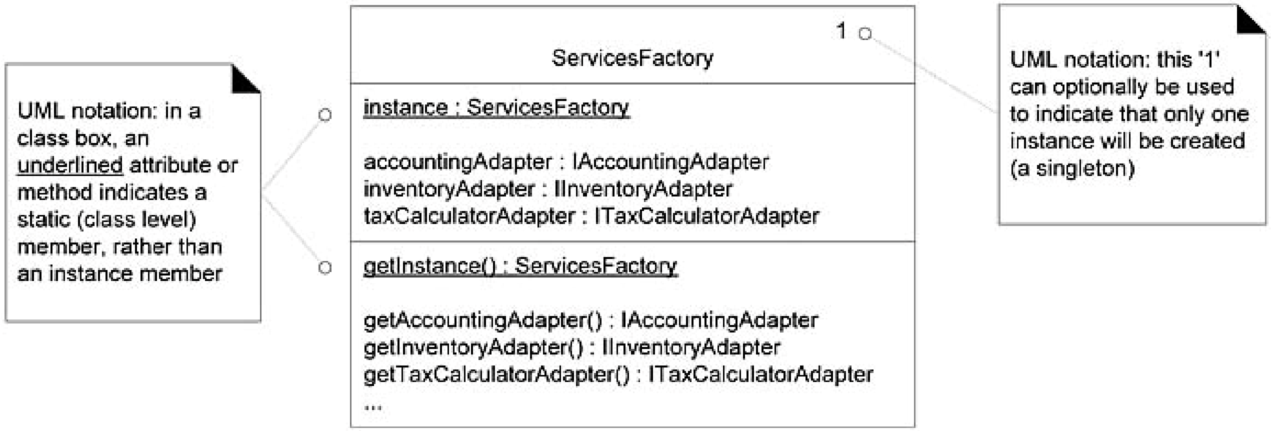
- **Association class**
  - Assign attributes to the relationship between classes rather than to a class itself.
  - Treat an association itself as a class, and model it with attributes, operations, and other features.
    - Illustrated with a dashed line from the association to the association class.
    - Necessary when modeling n:m Associations

  - Example : If a *Company* employs many *Persons*, modeled with an *Employs* association, you can model the association itself as the *Employment* class, with attributes such as *salary* and *startDate*.



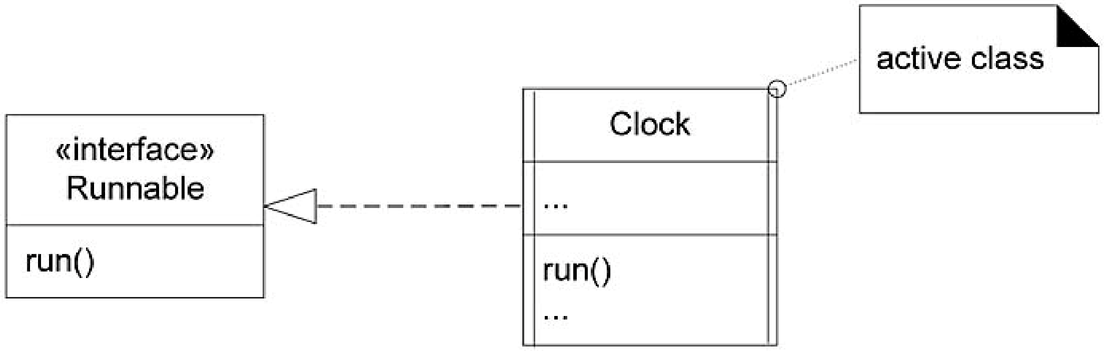
Singleton Classes

- **Singleton class** has only one instance of the class.
  - "singleton" instance
  - In a UML diagram, it is marked with a '1' in the upper right corner of the name compartment.
  - The Singleton design pattern



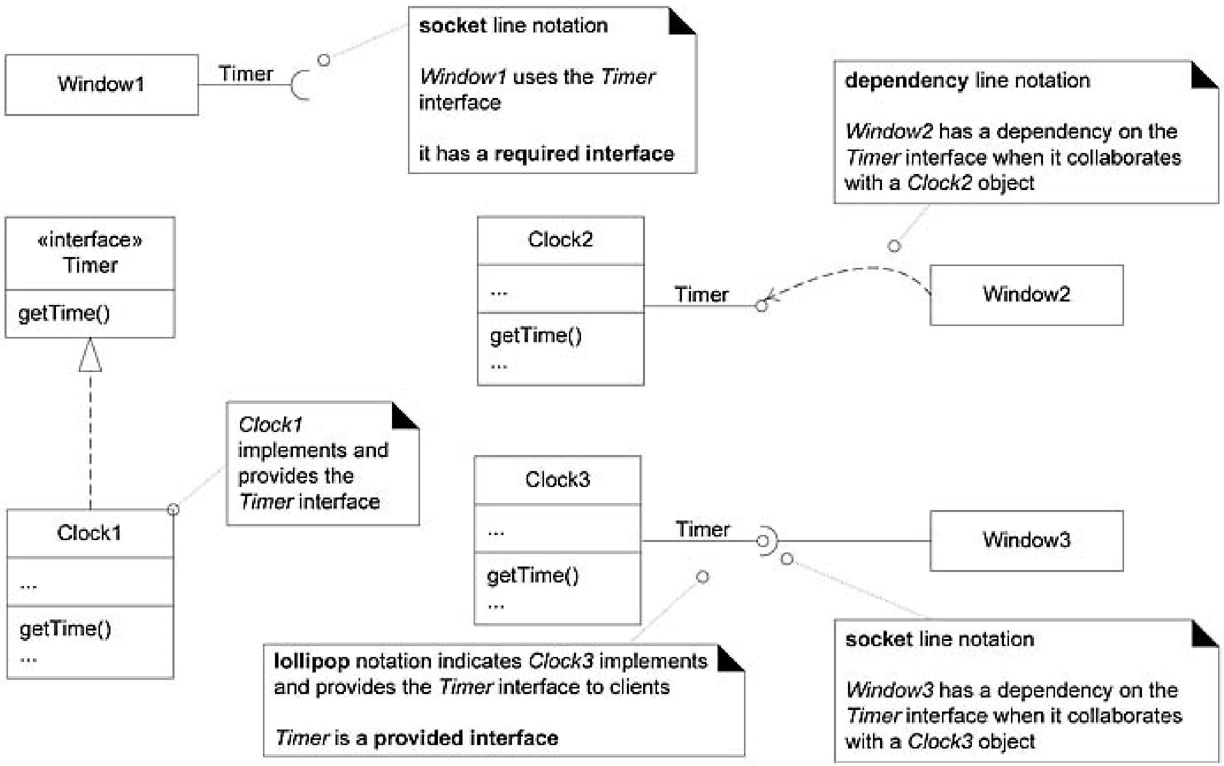
Active Class

- An **active object** runs and controls on its own thread of execution.
  - The class of an active object is an **active class**.
  - In the UML, it may be shown with double vertical lines on the left and right sides of the class box.



Interfaces

- The UML provides several ways to show interface implementation.
  - Formally called **interface realization**
  - 3 Notations:
    - Socket + lollipop notation
    - Dependency line notation
    - Interface implementation



DEPENDABLE SOFTWARE LABORATORY

Aggregation

- Special form of **association**
  - Used to express that **a class is part of another class**.

- Properties of the aggregation association:
  - **Transitive**: if **B** is part of **A** and **C** is part of **B**, **C** is also part of **A**
  - **Asymmetric**: it is not possible for **A** to be part of **B** and **B** to be part of **A** simultaneously.

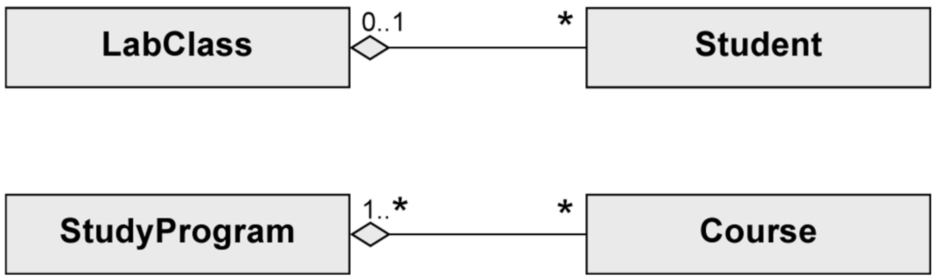
- Two types:
  - **Shared aggregation**
  - **Composition**

Shared Aggregation

- Expresses a **weak belonging** of the parts to a whole
  - Parts also exist independently of the whole.

- Multiplicity at the aggregating end may be >1.
  - One element can be part of multiple other elements simultaneously.
  - Spans a directed acyclic graph.
  - Syntax: Hollow diamond at the aggregating end

- Example:
  - `Student` is part of `LabClass`.
  - `Course` is part of `StudyProgram`.



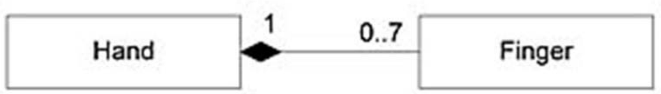
Composition

- **Existence dependency** between the composite object and its parts
  - One part can only be contained in at most one composite object at one specific point in time.
  - If the composite object is deleted, its parts are also deleted.
  - Multiplicity at the aggregating end is max. 1
    → The composite objects form a tree.
  - Syntax: Solid diamond at the aggregating end

- Example:
  - `Beamer` is part of `LectureHall` which is part of `Building`.

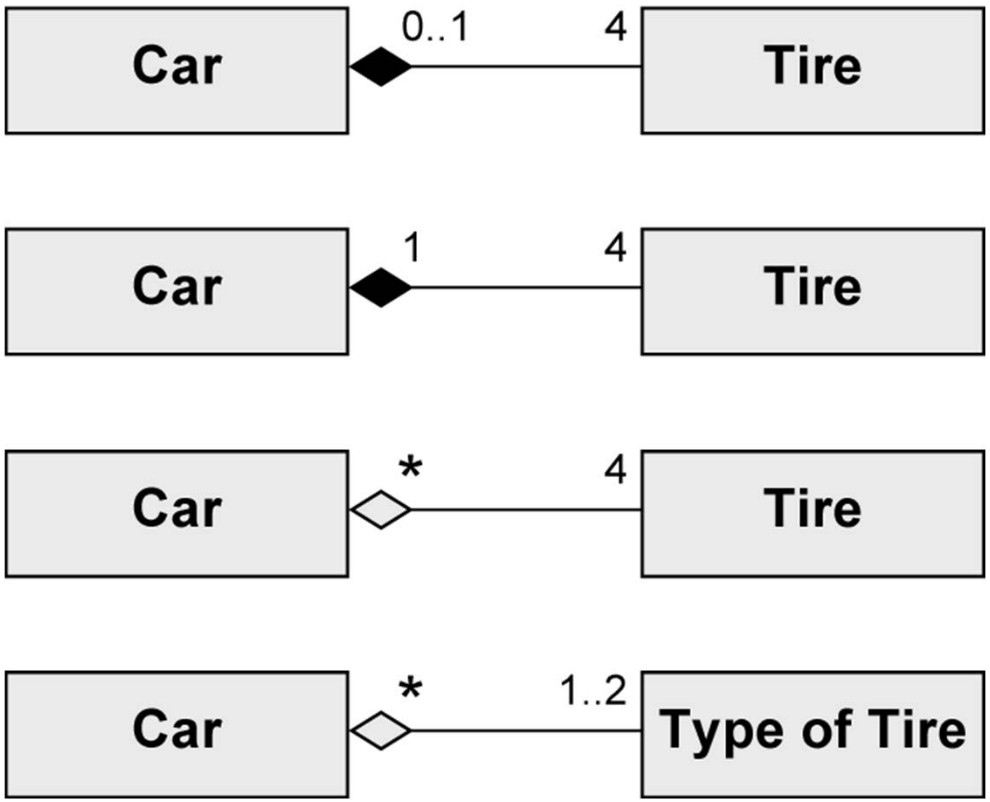
If the `Building` is deleted, the `LectureHall` is also deleted.

The `Beamer` can exist without the `LectureHall`, but if it is contained in the `LectureHall`, while it is deleted, the `Beamer` is also deleted.



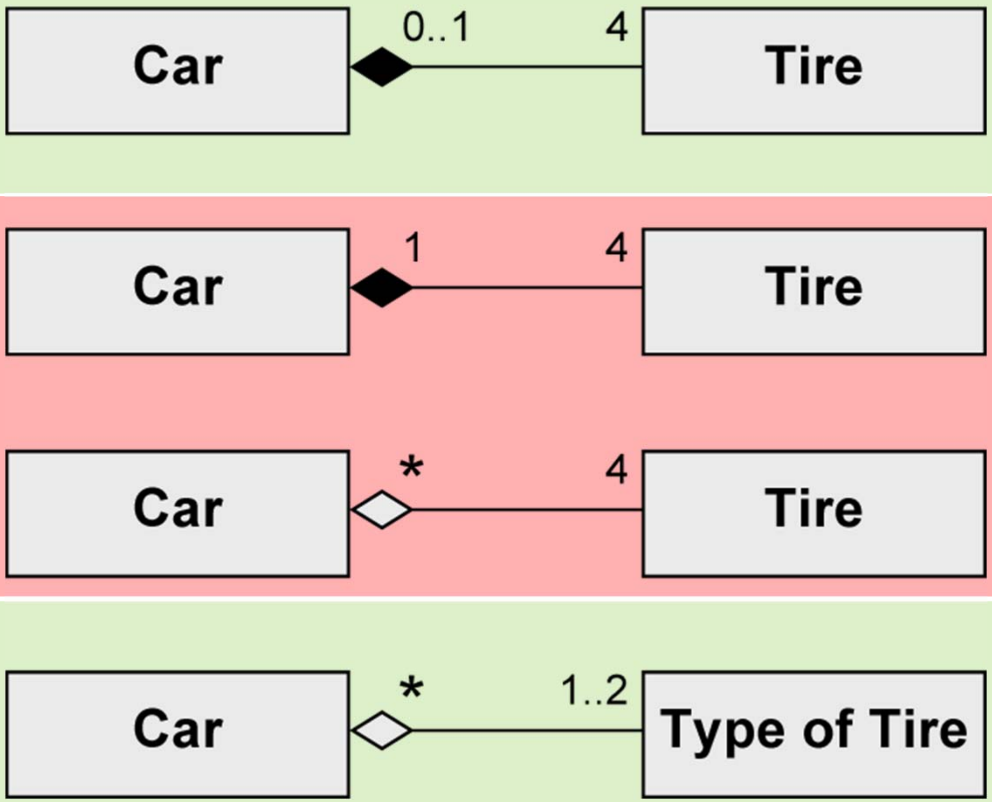
Shared Aggregation and Composition

- Which model applies?



Shared Aggregation and Composition

- Which model applies?



A Tire can exist without a Car. A Tire belongs to one Car at most. — Yes

A Tire cannot exist without a Car. — No

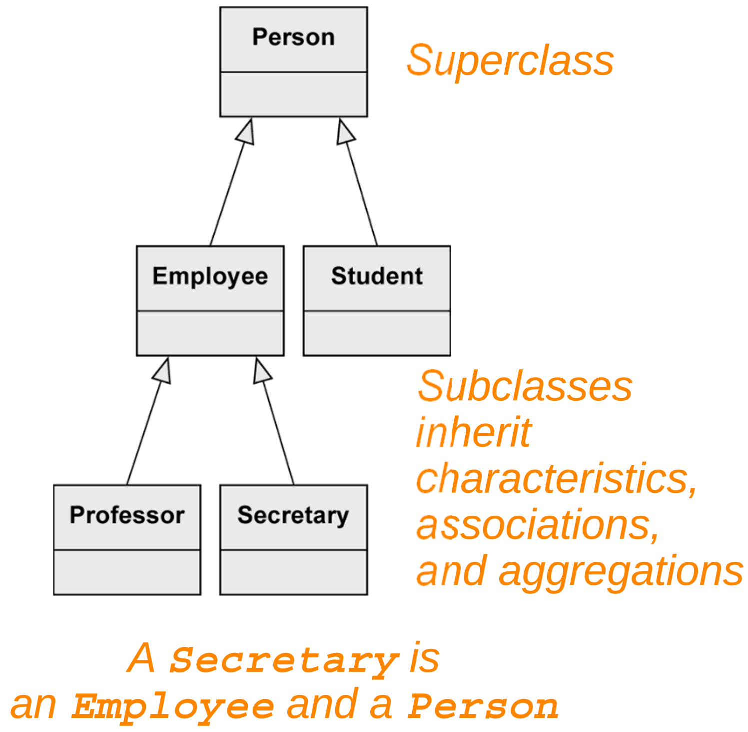
A Tire can belong to multiple Cars — No

A Car has one or two types of Tires. Several Cars may have the same Type of Tires. — Yes

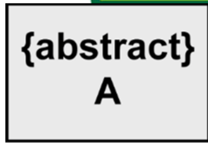
Generalization

- Everything of a general class are passed on to its subclasses.
  - Every instance of a subclass is simultaneously an indirect instance of the superclass.
  - Subclass inherits all characteristics (attributes and operations), associations, and aggregations of the superclass except private ones.
  - Subclass may have further characteristics, associations, and aggregations.

- Generalizations are transitive.

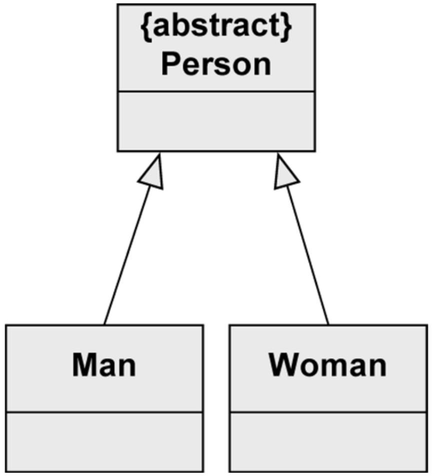


Generalization - Abstract Class

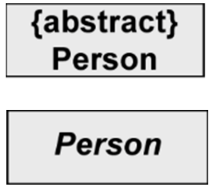


- Used to highlight common characteristics of their subclasses
- Used to ensure that there are no direct instances of the superclass
  - Only its non-abstract subclasses can be instantiated.

- Notation: keyword `{abstract}` or class name in italic font.



No `Person`-object possible

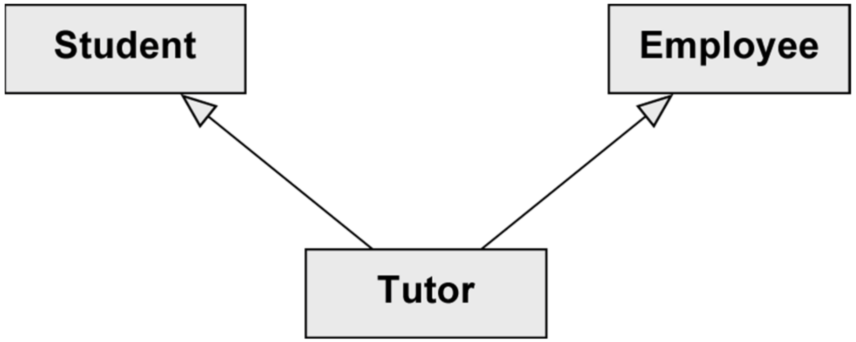


Two types of `Person`: `Man` and `Woman`

Generalization - Multiple Inheritance

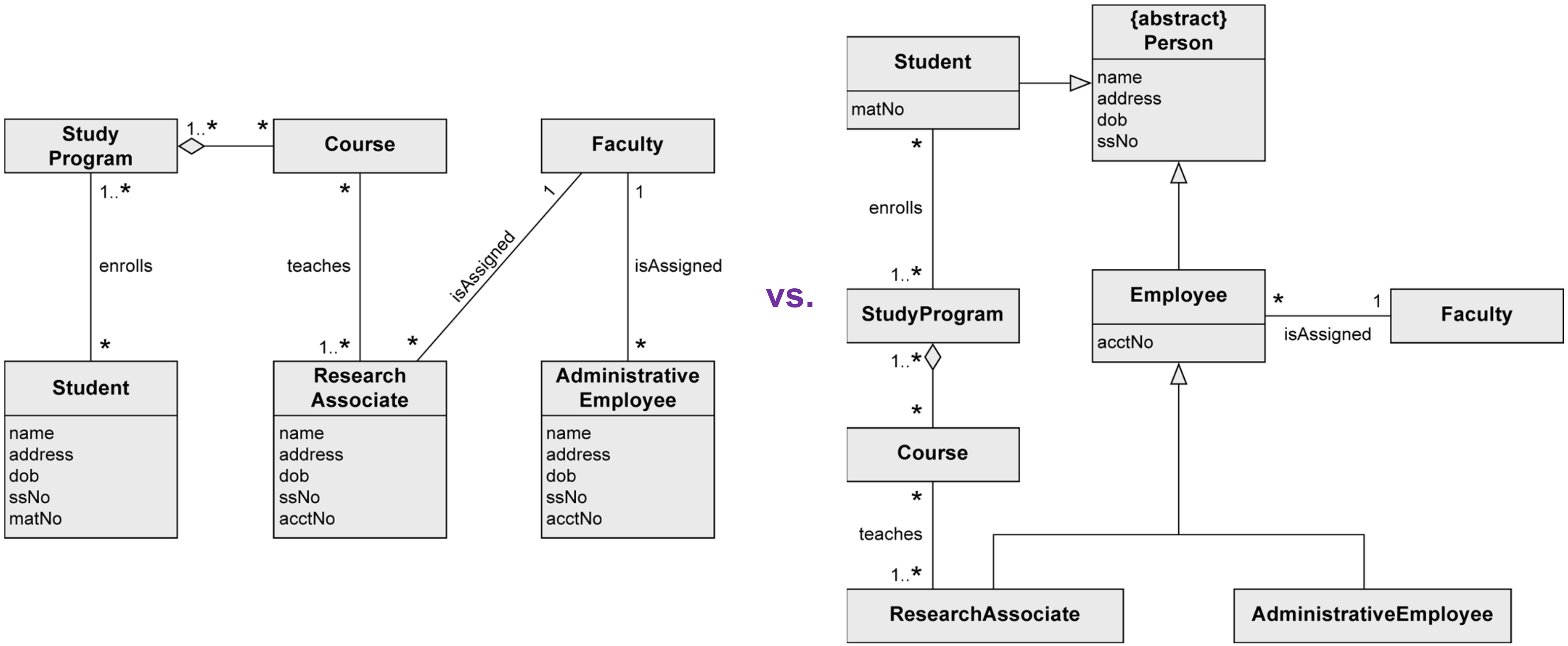
- UML allows multiple inheritance.
  - A class may have multiple superclasses.
  - Not allowed for JAVA programming language.

- Example:



A `Tutor` is both an `Employee` and a `Student`

With and Without Generalization

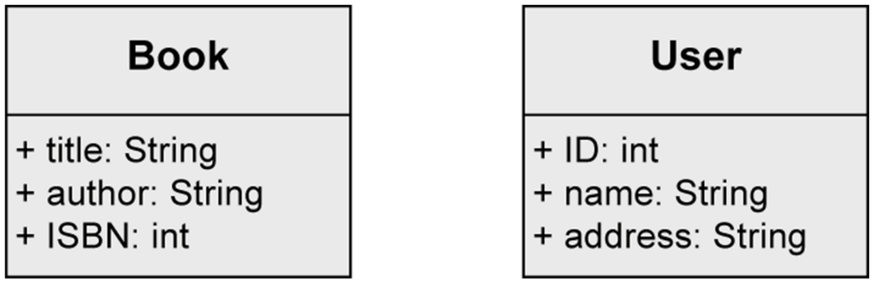


Creating a Class Diagram

- Not possible to completely extract classes, attributes and associations from a natural language text <u>automatically</u>.

- **Guidelines**
  - Nouns often indicate classes
  - Adjectives indicate attribute values
  - Verbs indicate operations

- Example: "The library management system stores users with their unique ID, name and address as well as books with their title, author and ISBN number. Ann Foster wants to use the library."



Example - University Information System

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.

- Each faculty is led by a dean, who is an employee of the university.

- The total number of employees is known. Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.

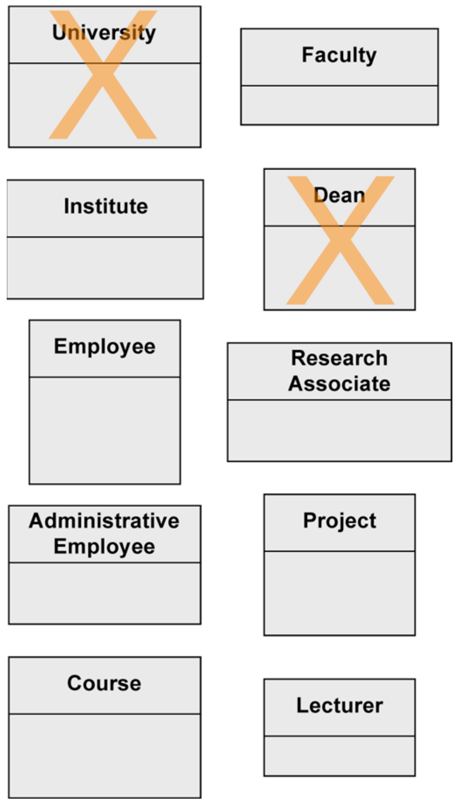
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.

- Courses have a unique number (ID), a name, and a weekly duration in hours.

Example - Step 1: Identifying Classes

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

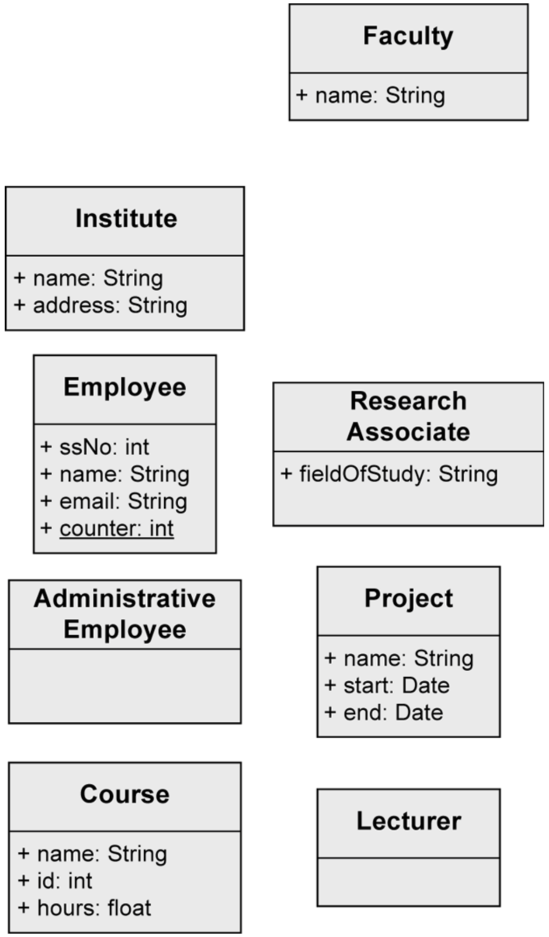
We model the system "University"



Dean has no further attributes than any other employee

Example - Step 2: Identifying the Attributes

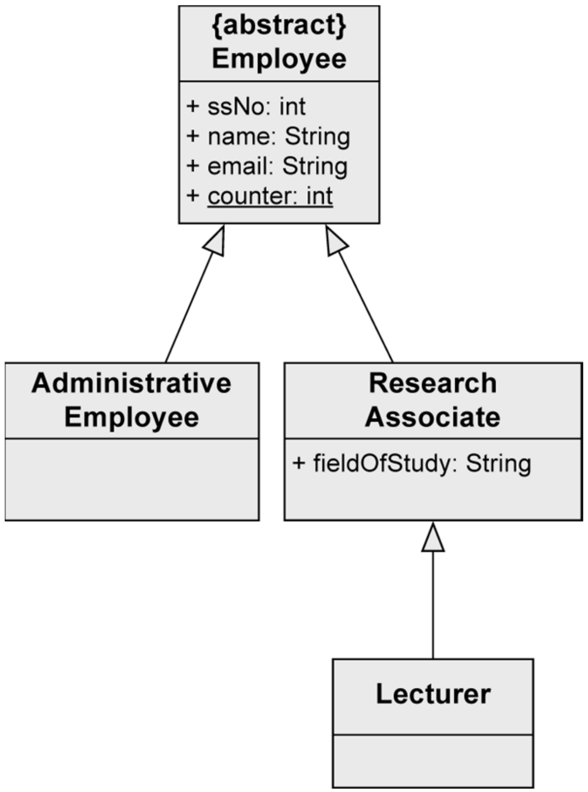
- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.



Example - Step 3: Identifying Relationships (1/6)

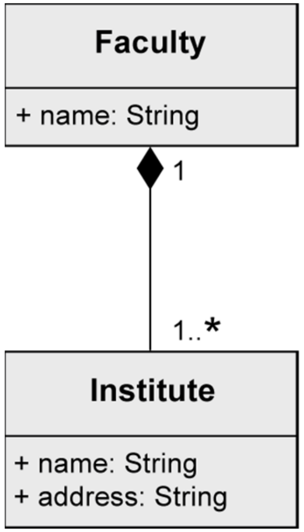
- Three kinds of relationships:
  - Association
  - Generalization
  - Aggregation

- Indication of a generalization
  - *“There is a distinction between research and administrative personnel.”*
  - *“Some research associates hold courses. Then they are called lecturers.”*



Example - Step 3: Identifying Relationships (2/6)

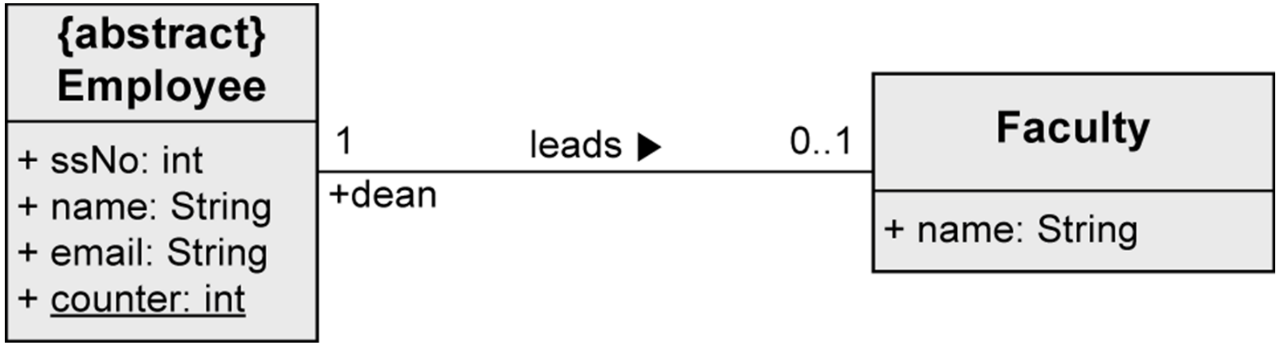
- *"A university consists of multiple faculties which are composed of various institutes."*



Composition to show existence dependency

Example - Step 3: Identifying Relationships (3/6)

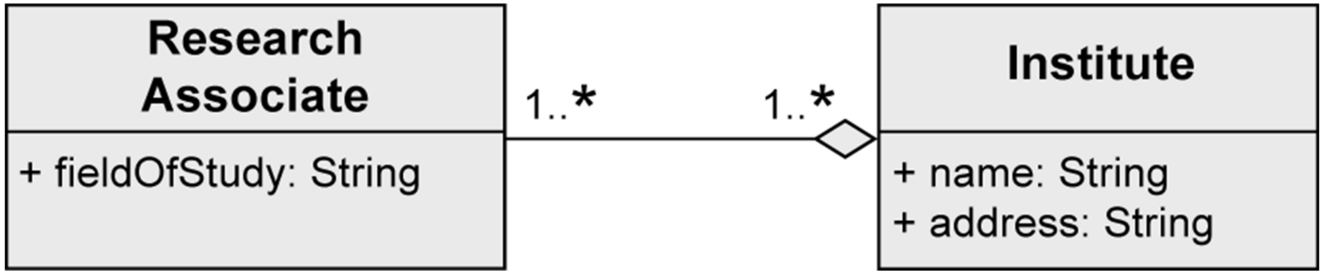
- *“Each faculty is led by a dean, who is an employee of the university”*



In the `leads`-relationship, the `Employee` takes the role of a `dean`.

Example - Step 3: Identifying Relationships (4/6)

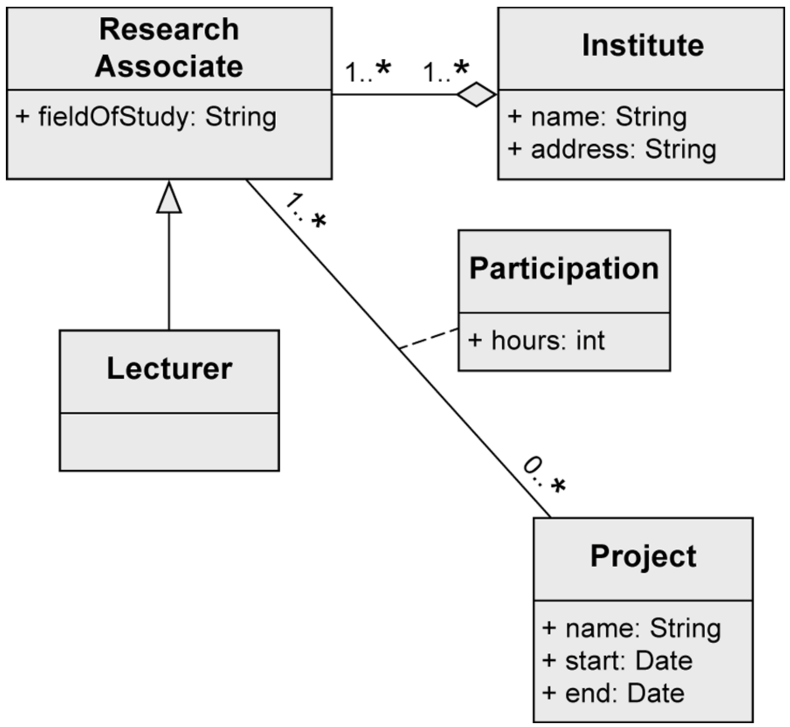
- *“Research associates are assigned to at least one institute.”*



Shared aggregation to show that `ResearchAssociates` are part of an `Institute`, but there is no existence dependency

Example - Step 3: Identifying Relationships (5/6)

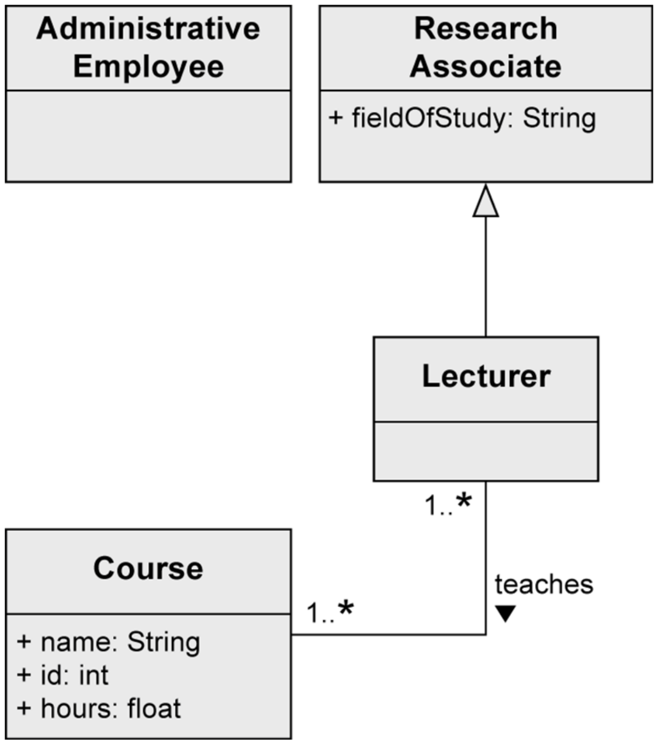
- *"Furthermore, research associates can be involved in projects for a certain number of hours."*



Association class enables to store the number of hours for every single `Project` of every single `ResearchAssociate`

Example - Step 3: Identifying Relationships (6/6)

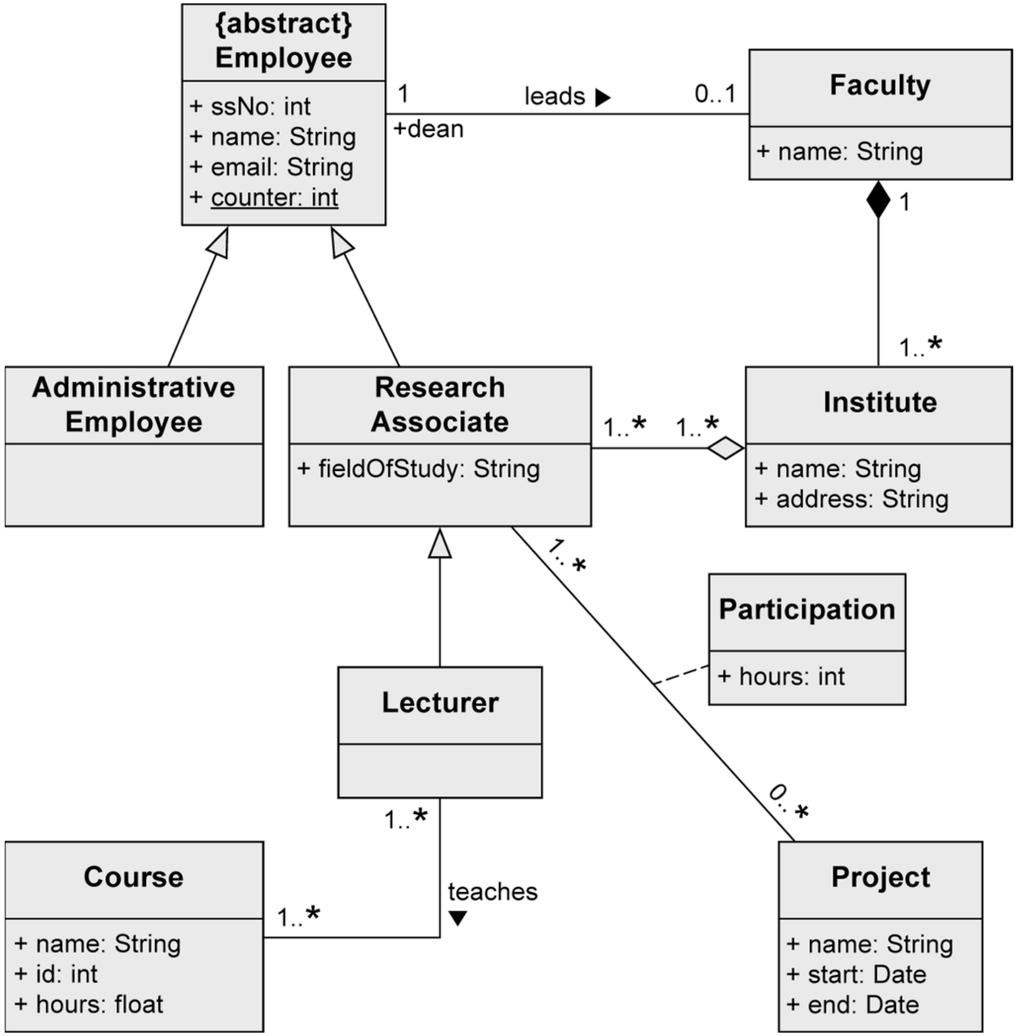
- *"Some research associates hold courses. Then they are called lecturers."*



`Lecturer` inherits all characteristics, associations, and aggregations from `ResearchAssociate`.

In addition, a `Lecturer` has an association `teaches` to `Course`.

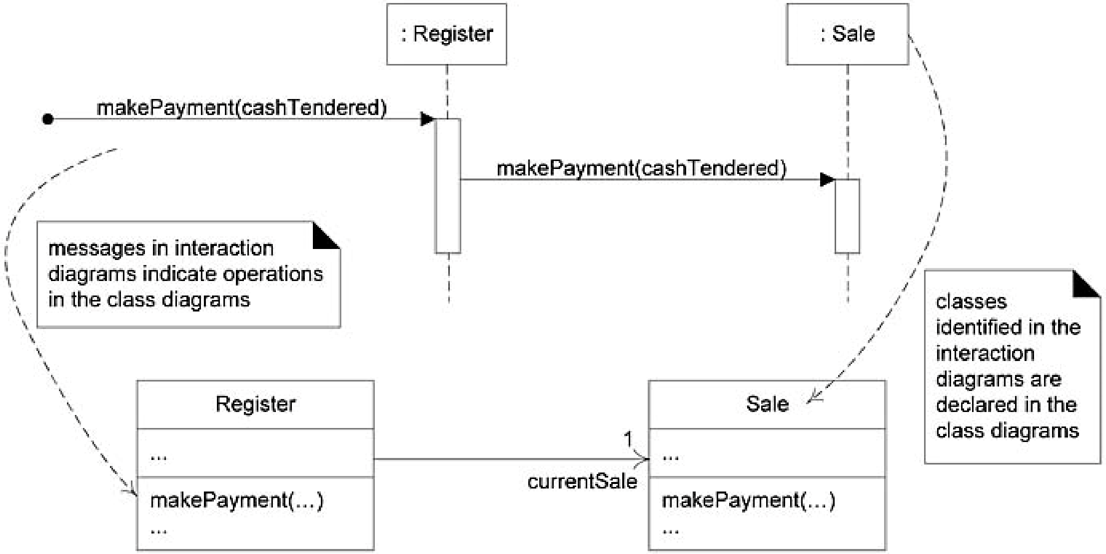
Example - A Complete Class Diagram



What's the Relationship between Interaction and Class Diagrams?

- From interaction diagrams, class diagrams can be generated iteratively.
  - When we draw interaction diagrams, a set of classes and their methods emerge.
  - Two complementary dynamic and static views are drawn concurrently and iteratively.

  - Example:
    - If we started with the *makePayment* sequence diagram, we see that a *Register* and *Sale* class definition in a class diagram can be obviously derived.



Chapter 17.

GRASP: Designing Objects with Responsibilities

OOD : Object-Oriented Design

- **OOD** is sometimes taught as some variation of the following:
  - "*After identifying your requirements and creating a domain model, then add methods to the appropriate classes, and define the messaging between the objects to fulfill the requirements.*"

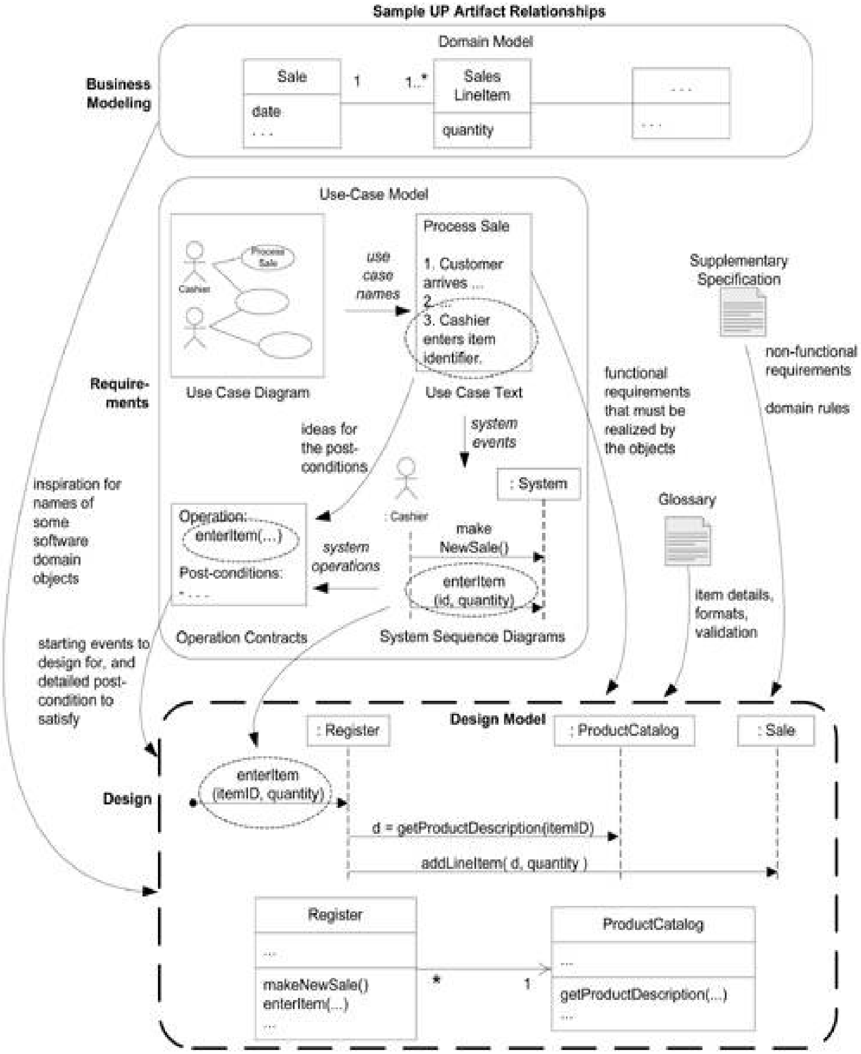
- But, it is not enough, because OOD involves **deep principles**.
  - Deciding what methods belong to where and how objects should interact carries consequences should be undertaken seriously.

- Mastering OOD is hard.
  - Involving a large set of soft principles, with many degrees of freedom.
  - A mind well educated in design principles is important.
  - **Patterns** can be applied.

Object Design with Patterns

- During the UML drawing activity, we can apply various **OO design principles**, such as
  - **GRASP** (General Responsibility Assignment Software Patterns)
  - Gang-of-Four (**GoF**) **design patterns**.

- Design outputs:
  - UML interaction diagrams
  - Class diagram
  - Package diagrams



GRASP: A Methodical Approach to Basic OO Design

- GRASP : **A Learning Aid for OO Design with Responsibilities**
  - **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns

- The GRASP principles or patterns are a learning aid to help you
  - Understand essential object design,
  - Apply design reasoning in a methodical, rational, and explainable way,
  - based on patterns of assigning responsibilities.

- We can apply the GRASP principles **while** drawing UML interaction diagrams.
  - Aid for naming, presenting, and remembering basic/classic design ideas

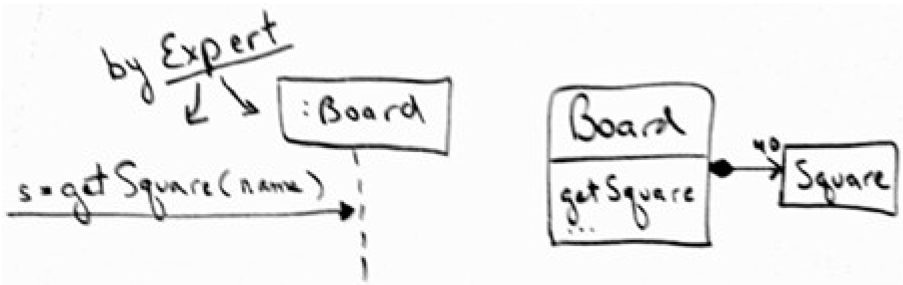
GRASP

- **9 basic OO design principles** or basic building blocks in design.
  - **Creator**
  - **Controller**
  - **Pure Fabrication**
  - **Information Expert**
  - **High Cohesion**
  - **Indirection**
  - **Low Coupling**
  - **Polymorphism**
  - **Protected Variations**

Pattern/ Principle	Description
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: 1. Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). 2. Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)
Low Coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
High Cohesion (evaluative)	How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
Polymorphism	Who is responsible when behavior varies by type? When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.
Pure Fabrication	Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling? Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept—something made up, in order to support high cohesion, low coupling, and reuse.
Indirection	How to assign responsibilities to avoid direct coupling? Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.
Protected Variations	How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements? Identify points of predicted variation or instability; assign responsibilities to create a stable "interface" around them.

Information Expert

Name	**Information Expert**
Problem	What is a basic principle by which to assign responsibilities to objects?
Solution	Assign a responsibility to the class that has the information needed to fulfill it.



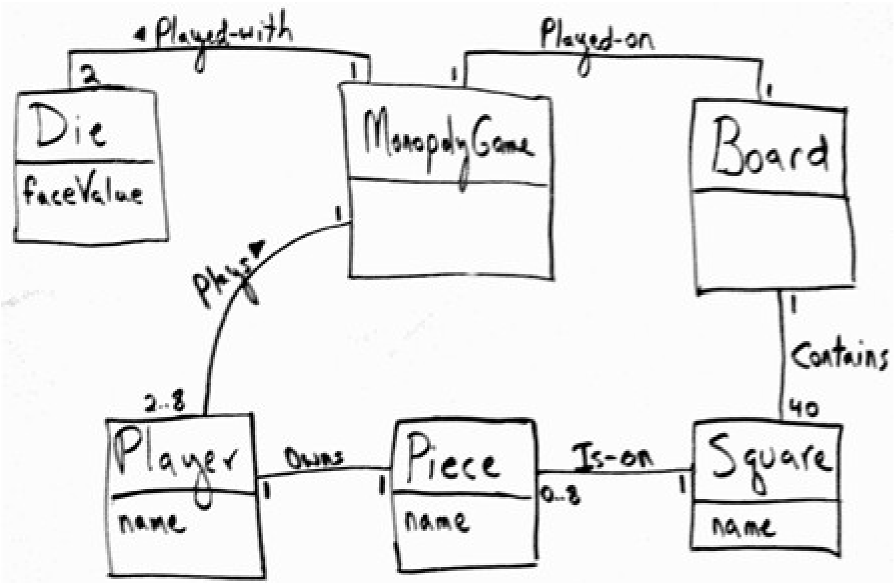
Applying Information Expert

A software *Board* will aggregate all the *Square* objects. Therefore, *Board* has the information necessary to fulfill this responsibility.

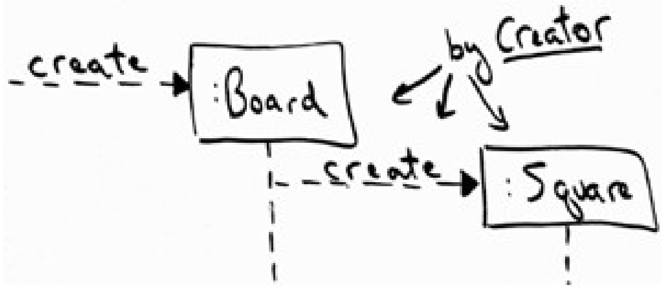
Creator

Name	**Creator**
Problem	Who creates an A?
Solution	Assign class B the responsibility to create an instance of class A, if one of these is true (the more the better): • B "contains" or compositely aggregates A. • B records A. • B closely uses A. • B has the initializing data for A.

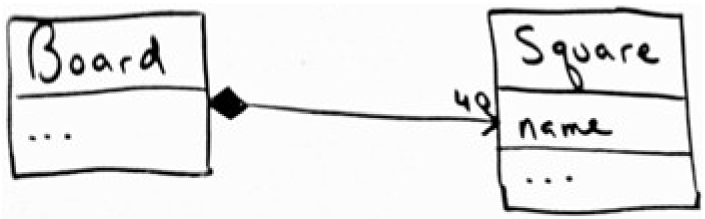
Example: Creator



Monopoly iteration-1 domain model



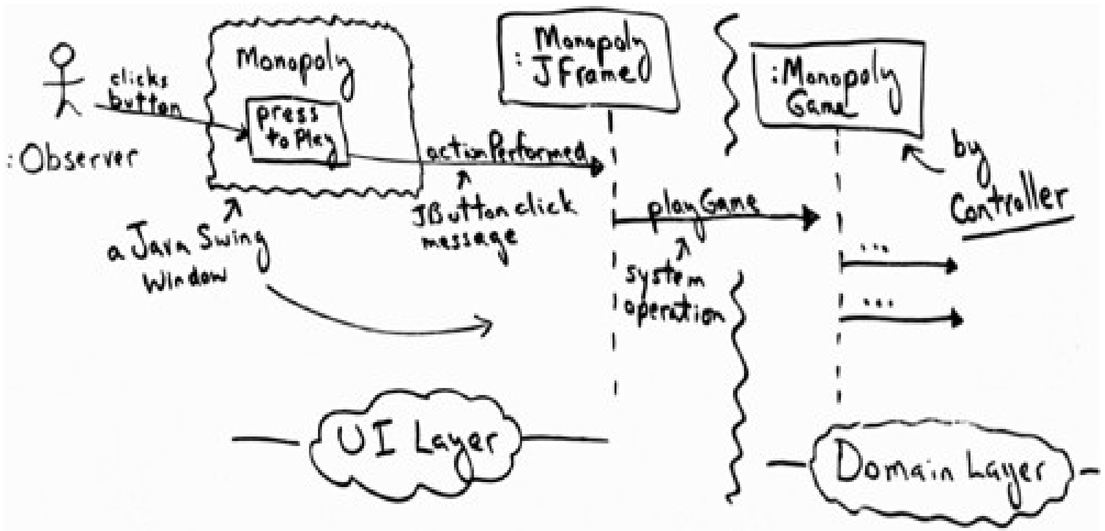
Applying the Creator pattern in a dynamic model



In a DCD of the Design Model, *Board* has a composite aggregation association with *Squares*.
We are applying *Creator* in a static model.

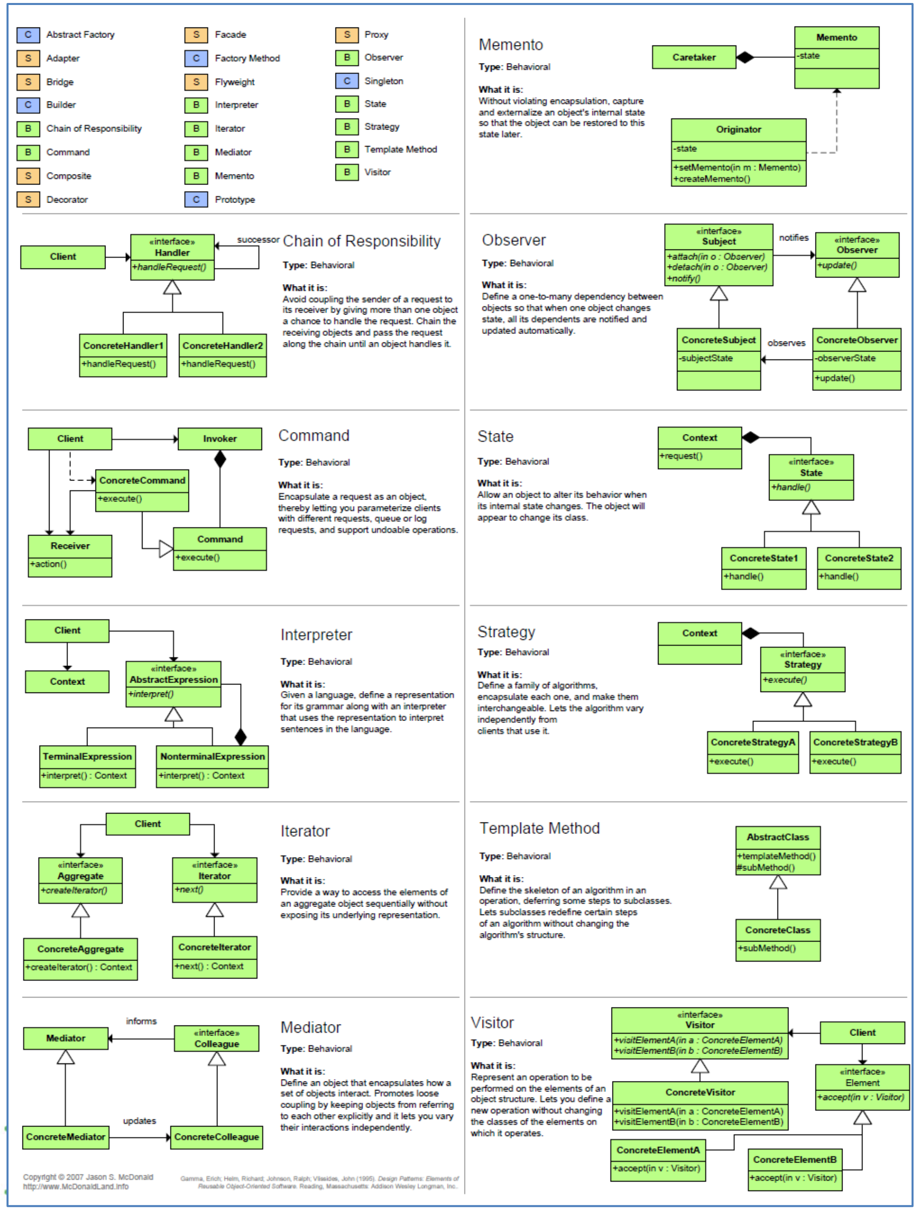
Controller

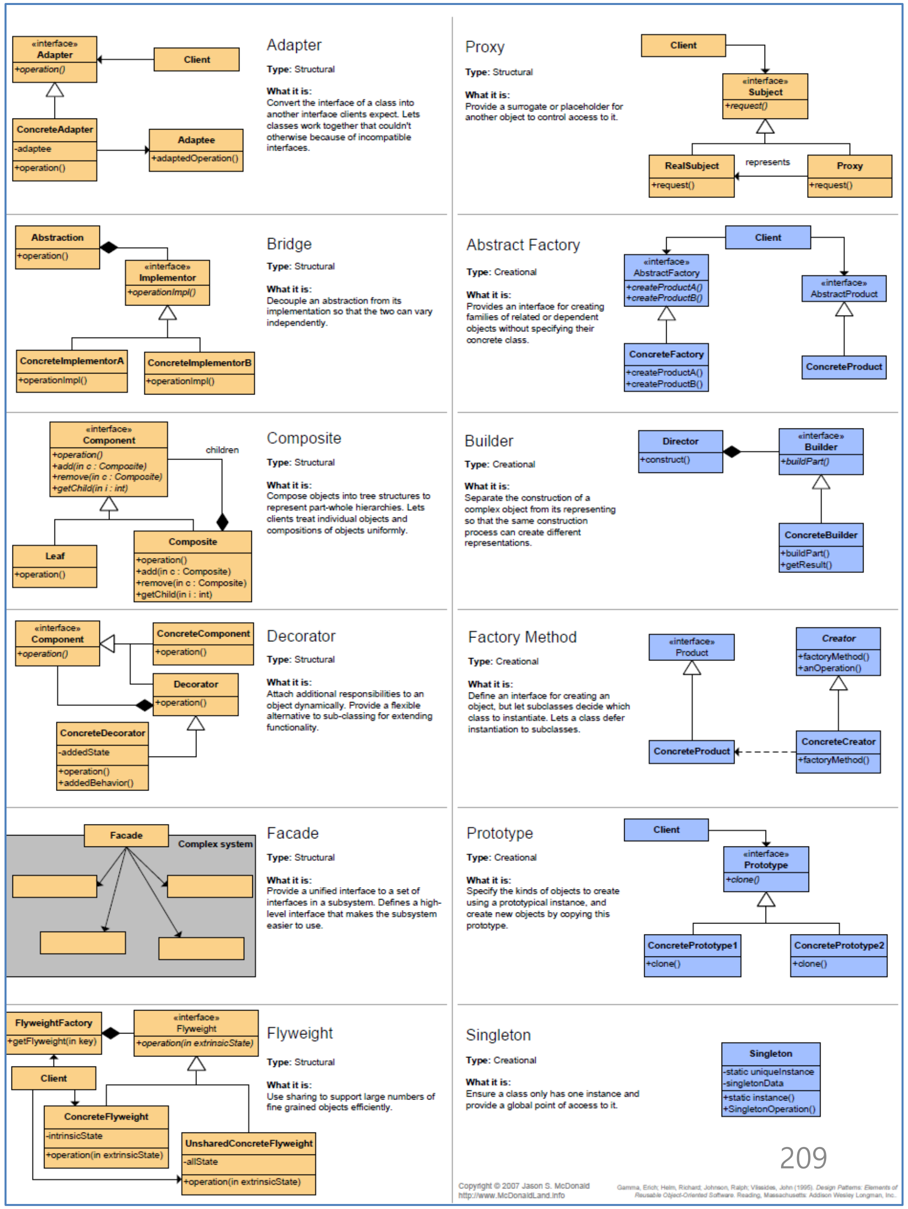
Name	**Controller**
Problem	What first object beyond the UI layer receives and coordinates ("controls") a system operation?
Solution	Assign the responsibility to an object representing one of these choices: • Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (all variations of a *facade controller*). • Represents a use case scenario within which the system operation occurs. (a use case or *session controller*)



Applying the Controller pattern using *MonopolyGame*.
Connecting the UI layer to the domain layer of software objects.

23 Design Patterns of GoF

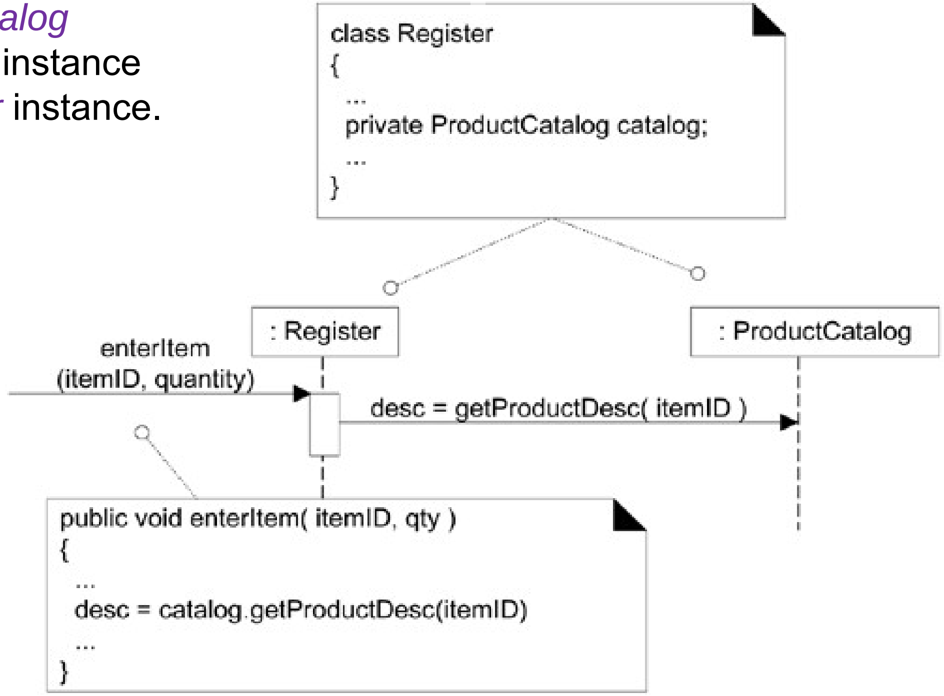




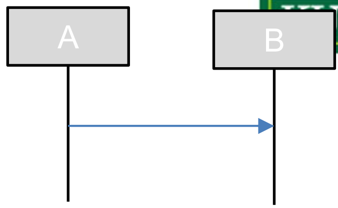
Chapter 19.
Designing for Visibility

Visibility Between Objects

- In message passing between objects,
  - 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.
  - Example,
    - The *getProductDesc* message sent from a *Register* to a *ProductCatalog* implies that the *ProductCatalog* instance should be visible to the *Register* instance.



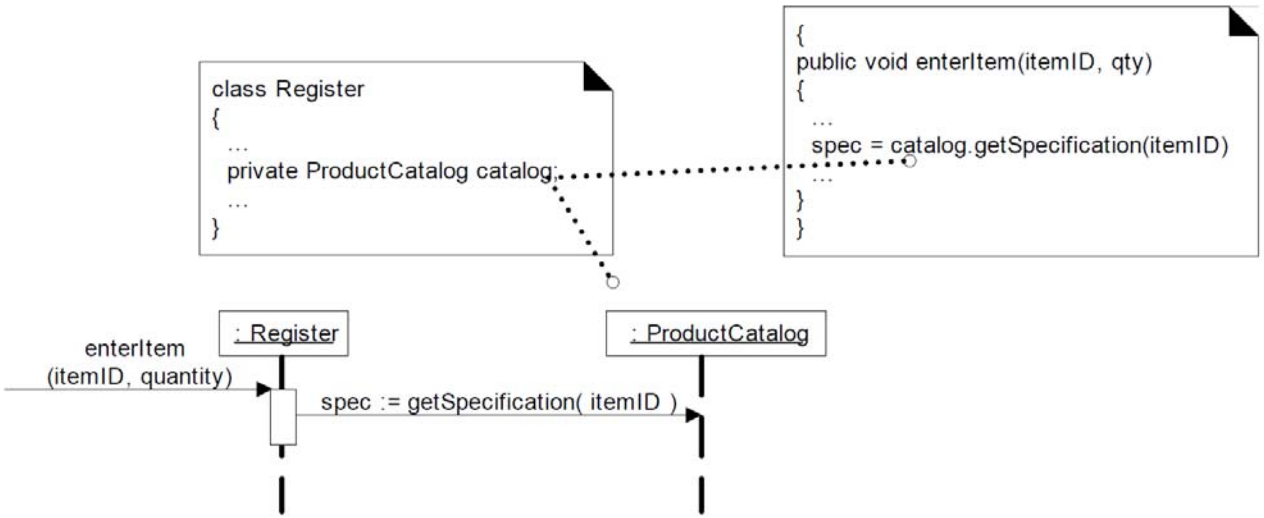
Visibility



- **Visibility** is the ability of an object to "**see**" or "**have a reference to"** another object.
  - When an object A sends a message to an object B, B must be visible to A.
  - The issue of scope: "Is one resource (such as an instance) within the scope of another?"
  - 4 common ways that visibility can be achieved from object A to object B:
    1. **Attribute visibility** : B is an attribute of A.
    2. **Parameter visibility** : B is a parameter of a method of A.
    3. **Local visibility** : B is a (non-parameter) local object in a method of A.
    4. **Global visibility** : B is in some way globally visible.

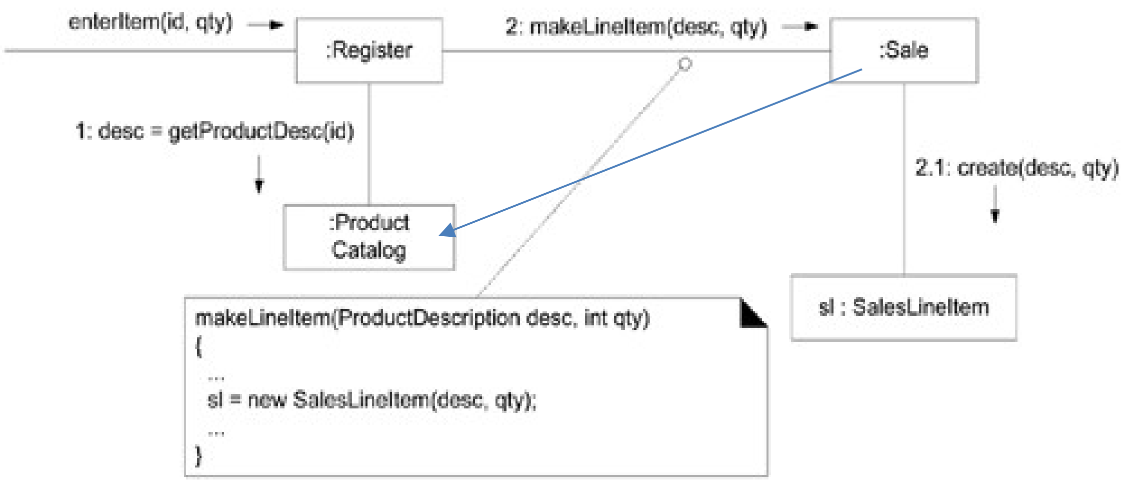
Attribute Visibility

- **Attribute visibility** from A to B exists, when B is an attribute of A.
  - Relatively permanent visibility, because it persists as long as A and B exist.
  - Very common form of visibility in object-oriented systems
  - For example,
    - For the class *Register*, a *Register* instance may have attribute visibility to a *ProductCatalog*, since it is an attribute of the *Register*.



Parameter Visibility

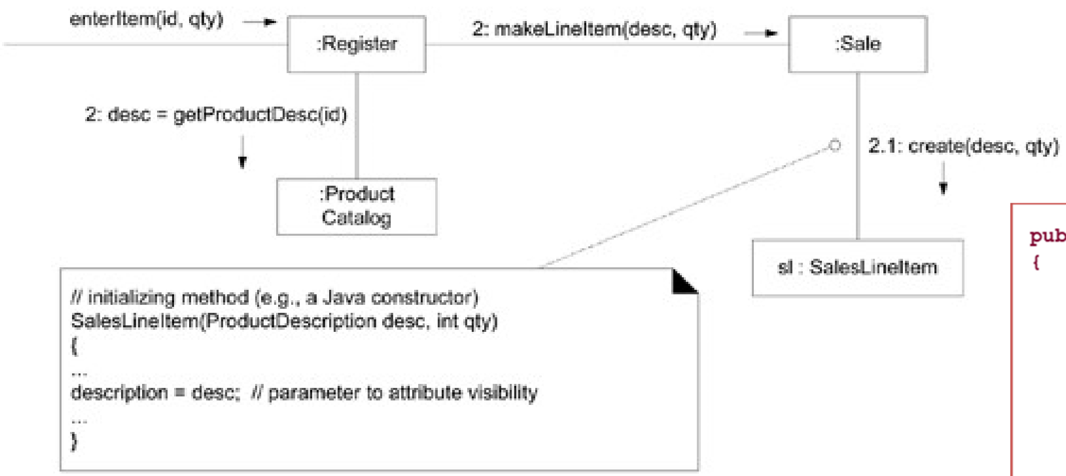
- **Parameter visibility** from A to B exists, when B is passed as a parameter to a method of A.
  - Relatively temporary visibility, because it persists only within the scope of the method.
  - The second most common form of visibility in object-oriented systems.
  - For example,
    - When the *makeLineItem* message is sent to a *Sale* instance, a *ProductDescription* instance is passed as a parameter. Within the scope of the *makeLineItem* method, the *Sale* has parameter visibility to a *ProductDescription*.



DEPENDABLE SOFTWARE LABORATORY

Parameter to Attribute Visibility

- It is common to transform parameter visibility into attribute visibility.
  - For example,
    - When the *Sale* creates a new *SalesLineItem*, it passes the *ProductDescription* in to its initializing method (in C++ or Java, this would be its constructor). Within the initializing method, the parameter is assigned to an attribute, thus establishing attribute visibility.

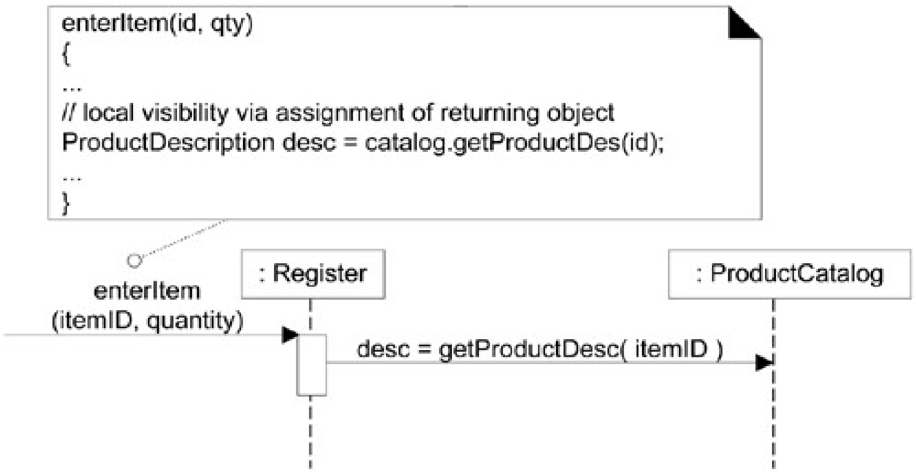



```
public class SalesLineItem
{
   private int quantity;
   private ProductDescription description;

   public SalesLineItem (ProductDescription desc, int quantity )
   {
      this.description = desc;  this.quantity = quantity;
   }

   public Money getSubtotal()
   {
      return description.getPrice().times( quantity );
   }
}
```

Local Visibility

- **Local visibility** from A to B exists, when B is declared as a local object within a method of A.
  - Relatively temporary visibility, because it persists only within the scope of the method.
  - As with parameter visibility, it is common to transform local visibility into attribute visibility.

- Two common ways for local visibility:
  1. Create a new local instance and assign it to a local variable.
  2. Assign the returning object from a method invocation to a local variable.



Global Visibility

- **Global visibility** from A to B exists, when B is global to A.
  - Relatively permanent visibility, because it persists as long as A and B exist.
  - The least common form of visibility in object-oriented systems

- One way to achieve global visibility is
  - Assign an instance to a global variable, which is possible in some languages, such as C++, but not others, such as Java.

- The preferred method to achieve global visibility is to use the **Singleton pattern**.

Chapter 20.
Mapping Designs to Code

Mapping Designs to Code

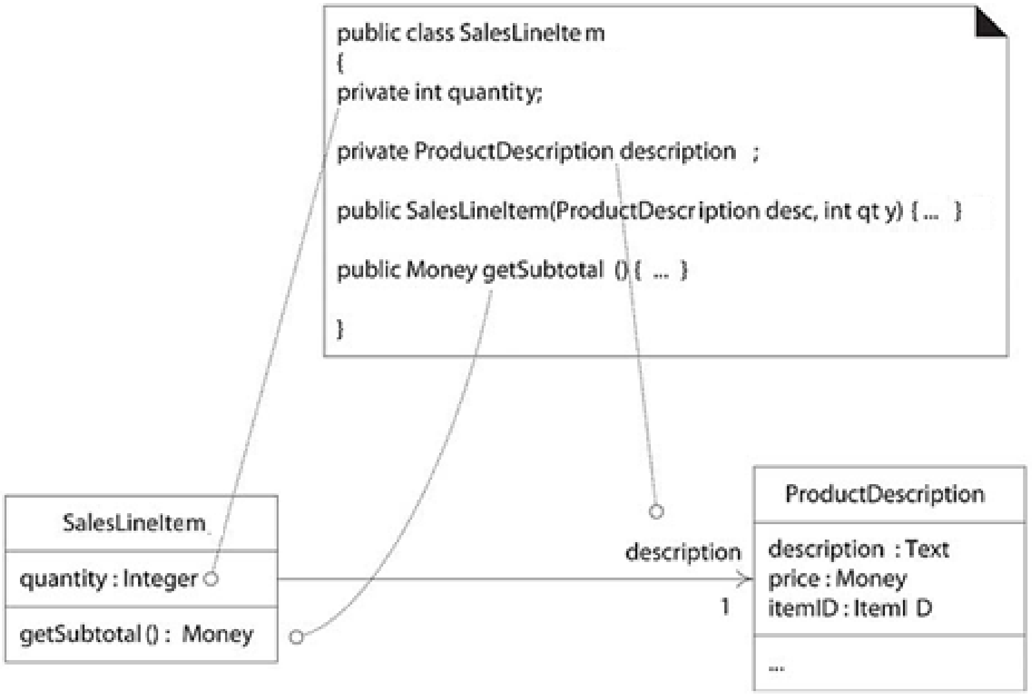
- The UML artifacts created during the design work (Interaction diagrams and DCDs) will be used as input to **the code generation** process.

- Implementation in an OO language requires writing source code for:
  - class and interface definitions
  - method definitions

- A translation from UML designs to code is required.
  - from **class diagrams** to **class definitions**,
  - from **interaction diagrams** to **method bodies**.

Creating Class Definitions from DCDs

- DCDs are sufficient to create a basic class definition in an OO language.
  - For example,
    - From the DCD, a mapping to the attribute definitions(Java fields) and method signatures for the Java definition (*SalesLineItem*) is straightforward.



Creating Methods from Interaction Diagrams

- The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions.
  - For example,
    - The *enterItem* interaction diagram illustrates the Java definition of the *enterItem* method.

(Method) The *enterItem* message is sent to a *Register* instance; therefore, the *enterItem* method is defined in class *Register*.

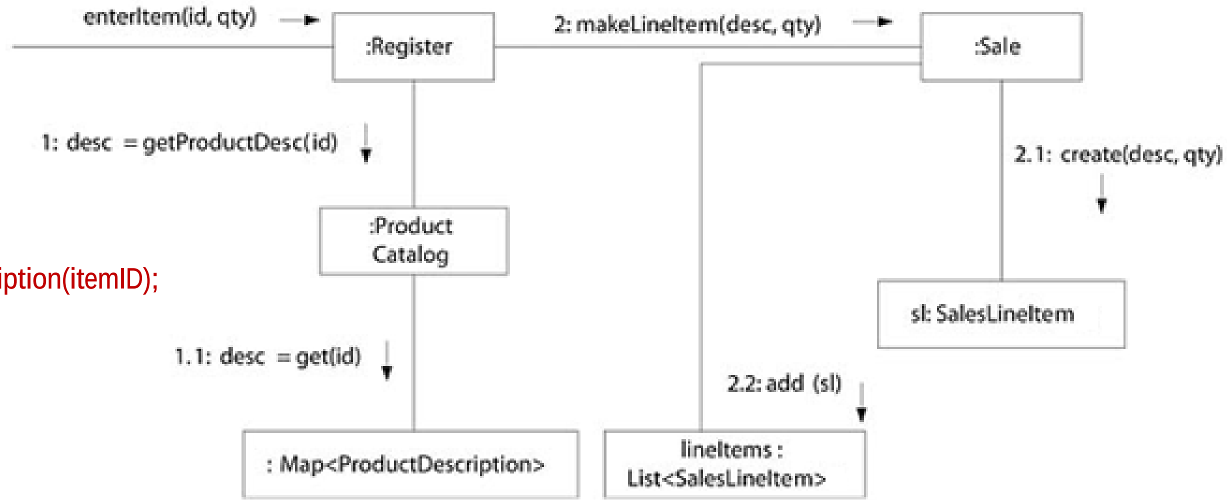
public void enterItem(ItemID itemID, int qty)

Message 2: The *makeLineItem* message is sent to the *Sale*.

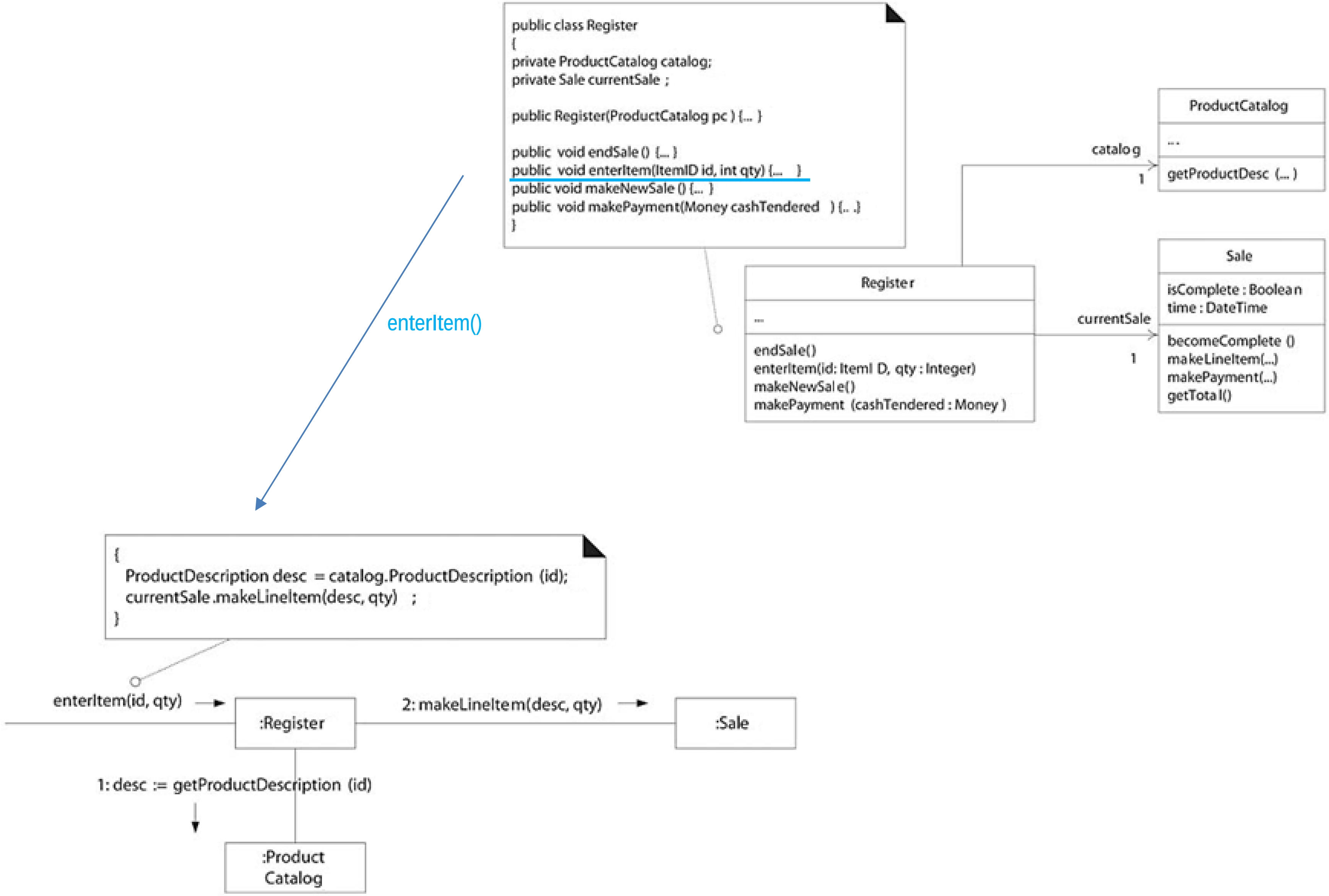
currentSale.makeLineItem(desc, qty);

Message 1: A *getProductDescription* message is sent to the *ProductCatalog* to retrieve a *ProductDescription*.

ProductDescription desc = catalog.getProductDescription(itemID);

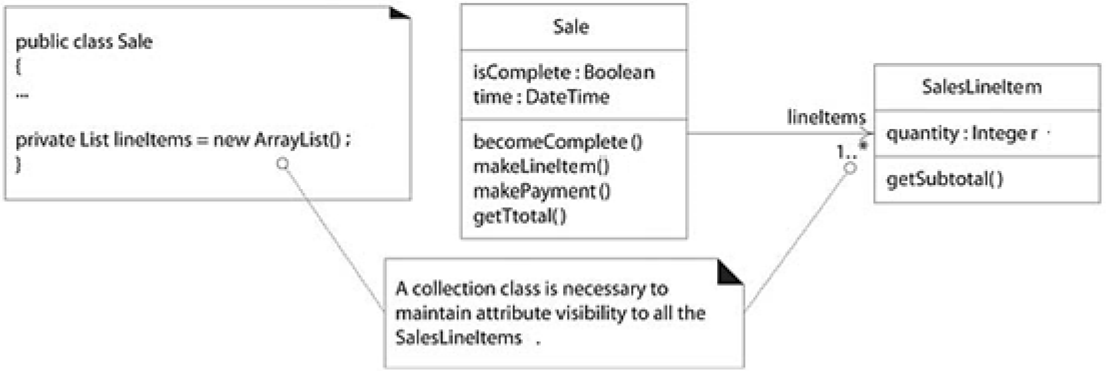


The Register.enterItem Method



Collection Classes in Code

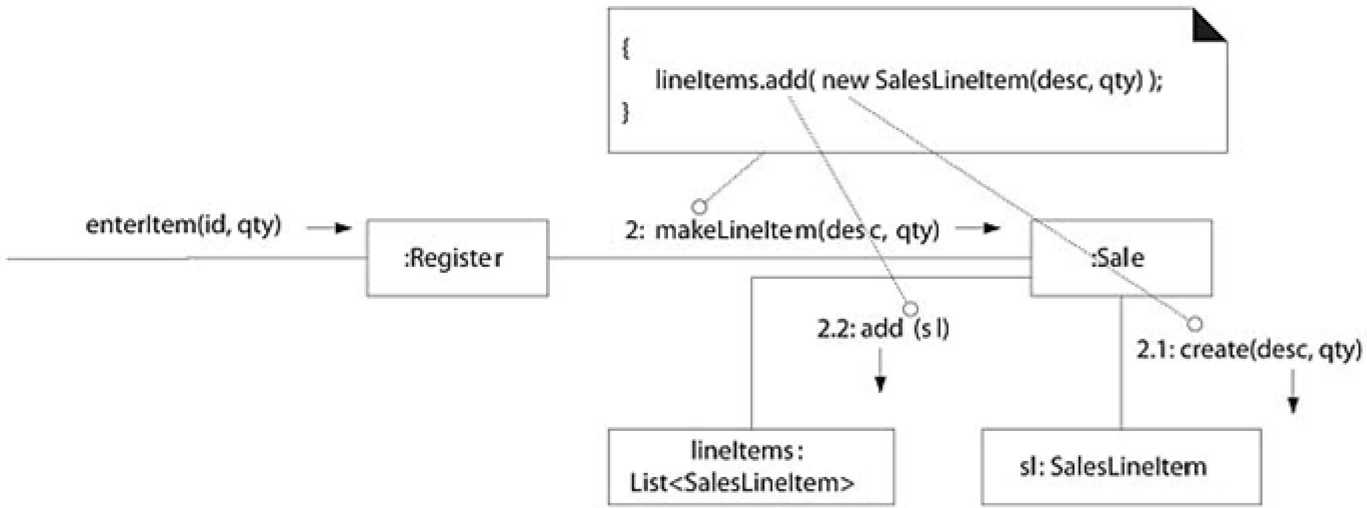
- One-to-many relationships are common.
  - For example, a *Sale* must maintain (attribute) visibility to a group of many *SalesLineItem* instances.



- In OO programming languages, they are usually implemented with the introduction of a **collection object** of collection classes.
  - List (*ArrayList – List interface*) : a growing ordered list
  - Map (*HashMap – Map interface*) : a key-based lookup
  - Simple array

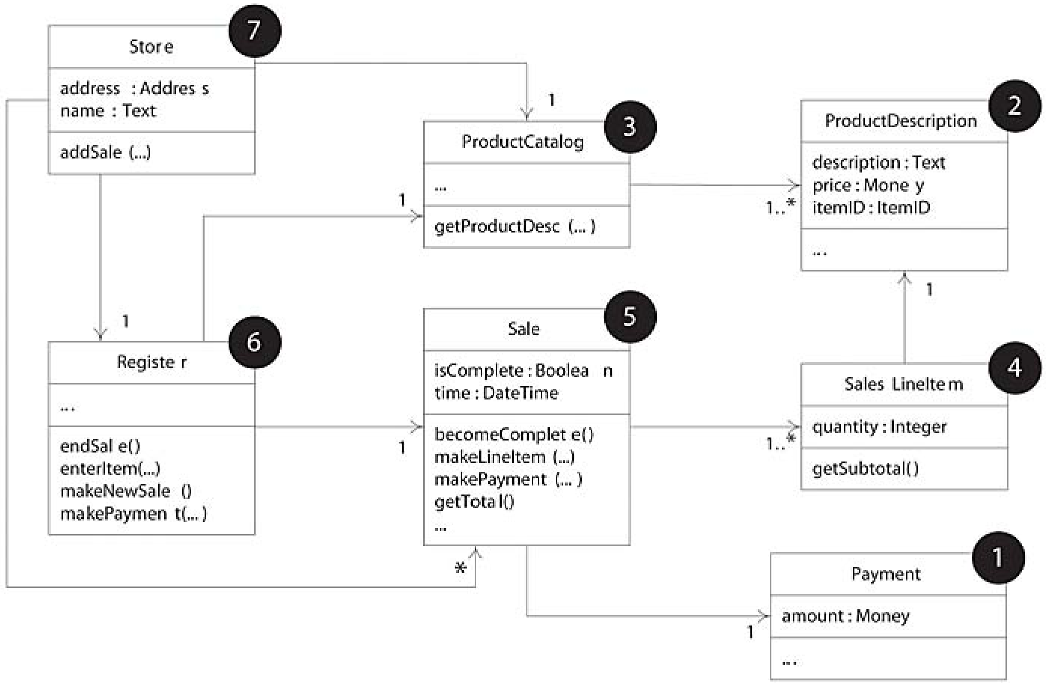
Example : Defining the *Sale.makeLineItem* Method

- The *makeLineItem* method of class *Sale* can be written by inspecting the *enterItem* communication diagram.



Order of Implementation

- Classes need to be implemented from least-coupled to most-coupled.
  - For example,
    - Possible first classes to implement are either *Payment* or *ProductDescription*.
    - Next are classes only dependent on the prior implementations; *ProductCatalog* or *SalesLineItem*.

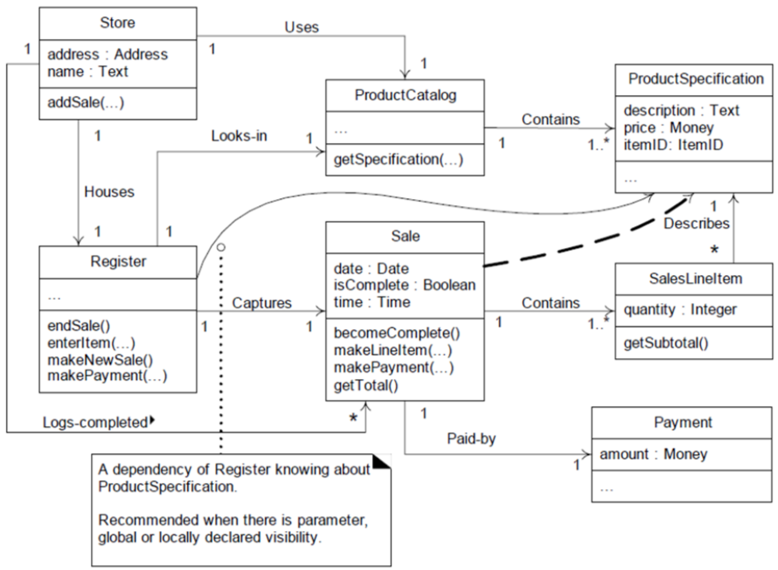


A possible order of class implementation and testing

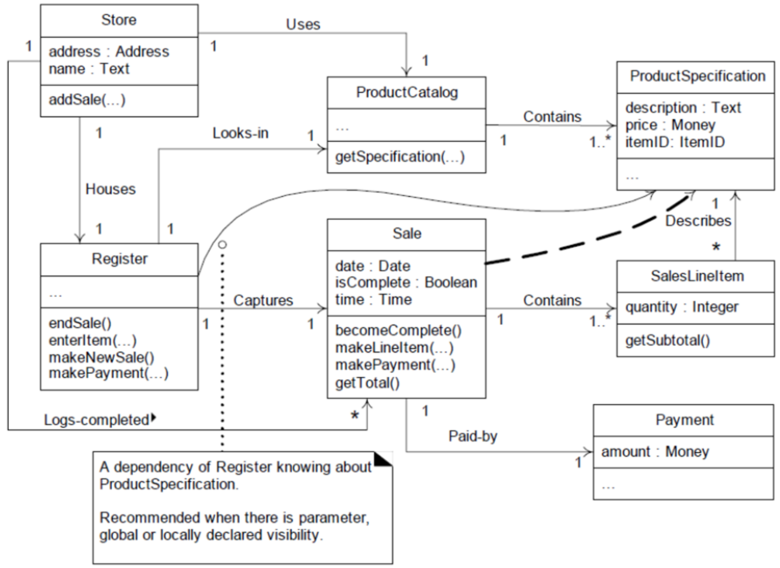
Example: the NextGen POS Program Solution

- Translation from design artifacts to a foundation of code.
  - This code defines a simple case; it is not meant to illustrate a robust, fully developed Java program with synchronization, exception handling, and so on.

```
// all classes are probably in a package named something like:
// package com.foo.nextgen.domain;


public class Payment
{
   private Money amount;

   public Payment( Money cashTendered ){ amount = cashTendered; }
   public Money getAmount() { return amount; }
}
```

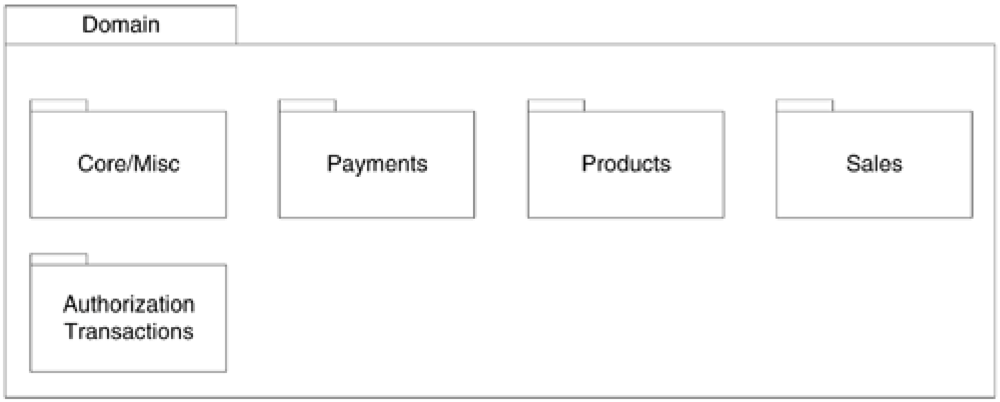


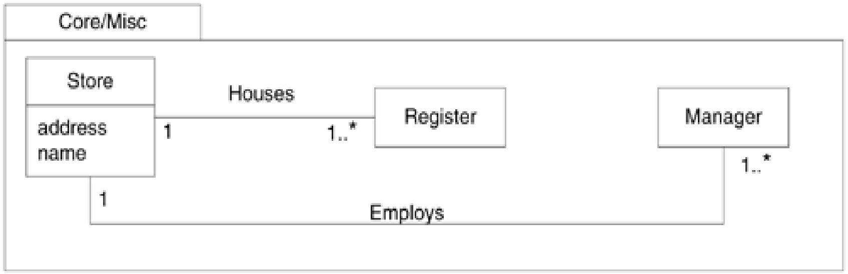
```java
public class Register
{
   private ProductCatalog catalog;
   private Sale currentSale;

   public Register ( ProductCatalog catalog )
   {
      this.catalog = catalog;
   }

   public void endSale()
   {
      currentSale.becomeComplete();
   }

   public voud enterItem ( ItemID id, int quantity )
   {
      ProductDescription desc = catalog.getProductDescription ( id );
      currentSale.makeLineItem ( desc, quantity );
   }

   public void makeNewSale()
   {
      currentSale = new Sale();
   }

   public void makePayment ( Money cashTendered )
   {
      currentSale.makePayment (cashTendered);
   }
}
```

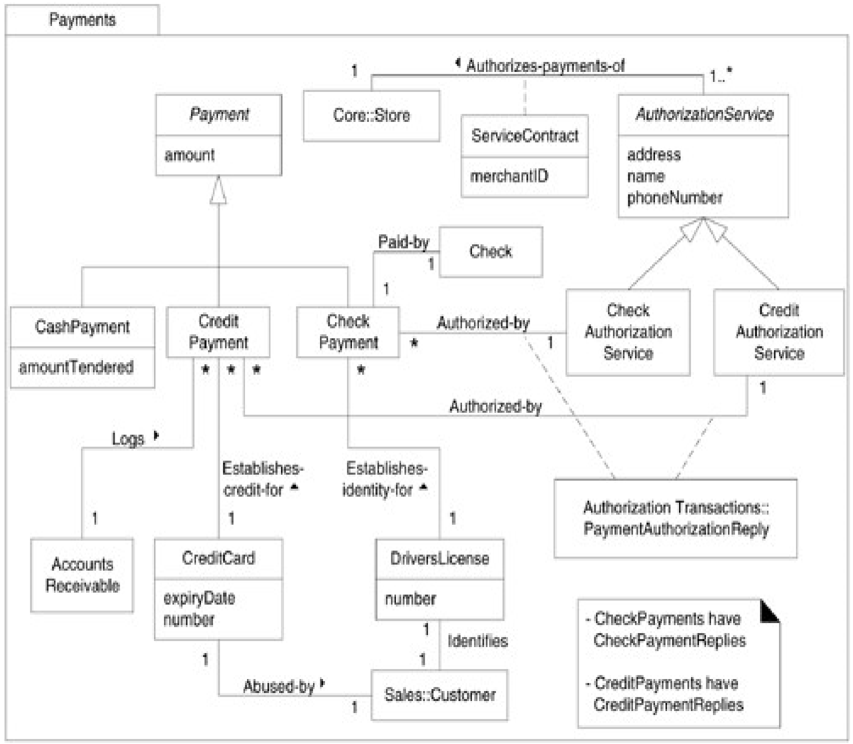


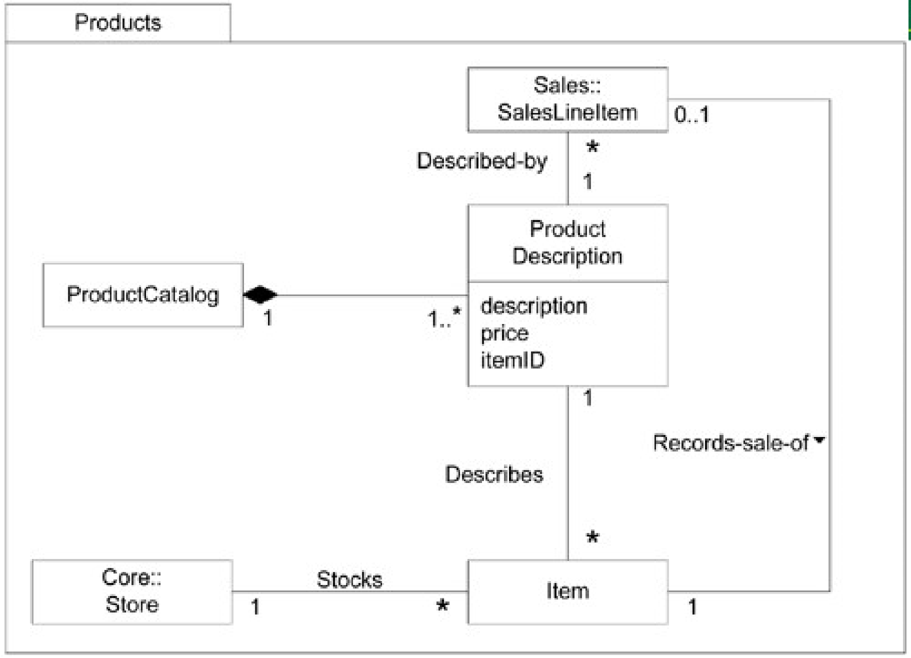
Example: POS Domain Model Packages

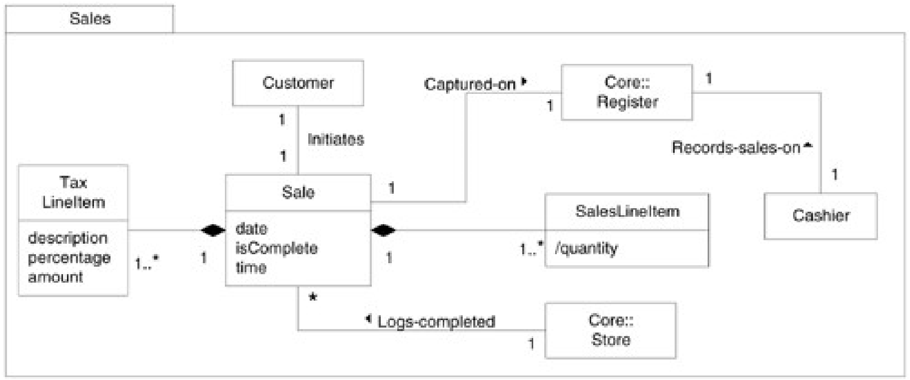
- After Elaboration - Iteration 3.



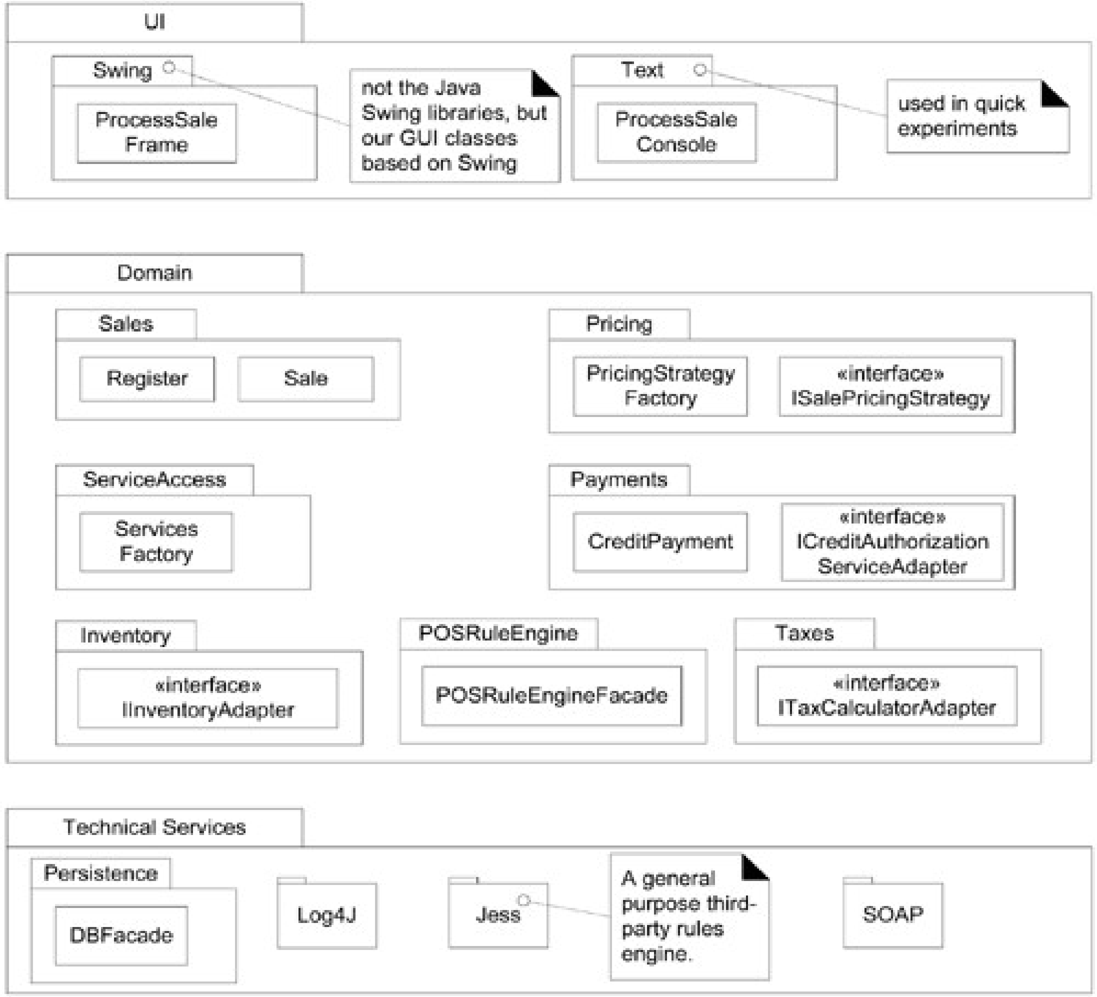

Core/Misc
Store
address
name
Houses
1
1..*
Register
Manager
1..*
1
Employs


Payments
1
Authorizes-payments-of
1..*
Payment
amount
Core::Store
ServiceContract
merchantID
AuthorizationService
address
name
phoneNumber
Paid-by
Check
1
1
CashPayment
amountTendered
Credit Payment
Check Payment
Authorized-by
*
1
Check Authorization Service
Credit Authorization Service
1
Authorized-by
Logs
Establishes-credit-for
Establishes-identity-for
1
1
1
Accounts Receivable
CreditCard
expiryDate
number
DriversLicense
number
Authorization Transactions::
PaymentAuthorizationReply
Identifies
1
1
Abused-by
1
Sales::Customer
- CheckPayments have CheckPaymentReplies
- CreditPayments have CreditPaymentReplies

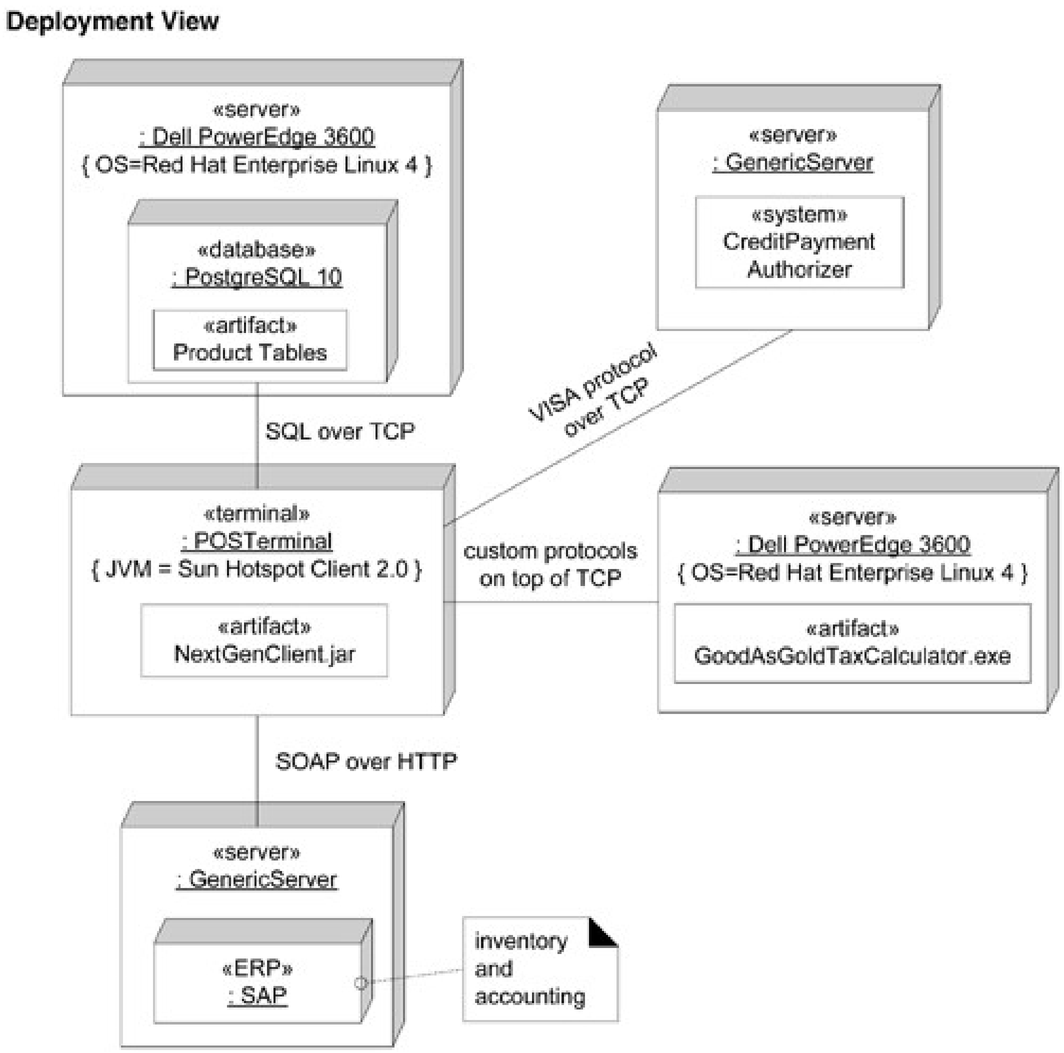

Products
Sales::
SalesLineItem
0..1
Described-by
*
1
Product
Description
description
price
itemID
ProductCatalog
1
1..*
1
Records-sale-of
Describes
*
Core::
Store
Stocks
1
*
Item
1


Sales
Customer
Captured-on
Core::
Register
1
1
1
Initiates
1
Records-sales-on
1
Tax
LineItem
description
percentage
amount
Sale
date
isComplete
time
1
SalesLineItem
/quantity
Cashier
1..*
1
1
1..*
*
Logs-completed
Core::
Store
1

Partial Layers of POS



Deployment View of POS



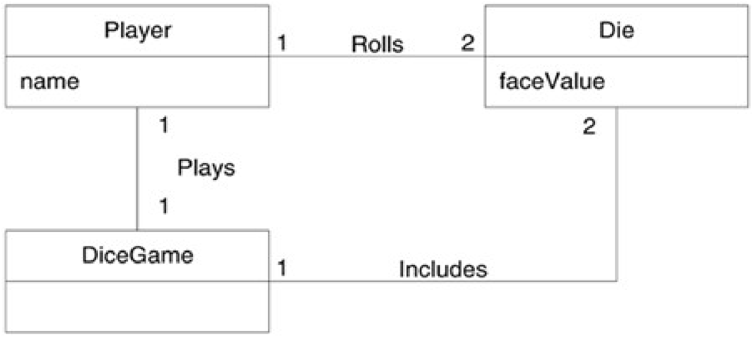
Object-Oriented Analysis and Design - Summary

An Short Example of OOAD - Dice Game

Define use cases → Define domain model → Define interaction diagrams → Define design class diagrams

OOA

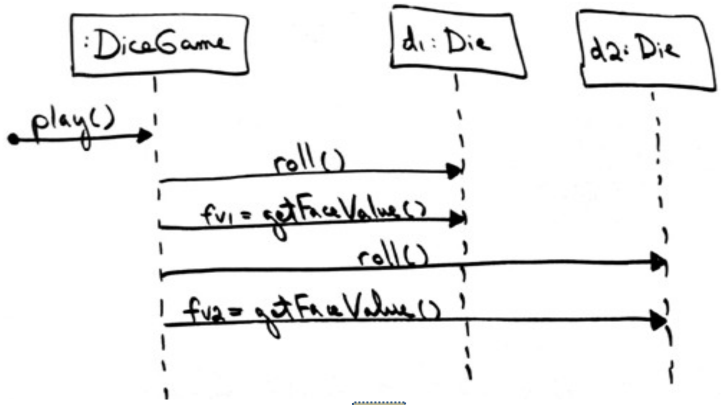
Use Case : Play a Dice Game
- Player requests to roll the dice.
- System presents results.
- If the dice's face value totals seven, player wins; otherwise, player loses.

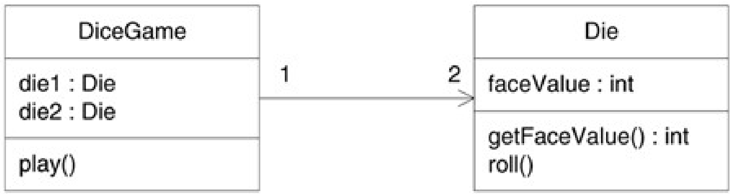


Domain Model

OOD

Interaction Diagram





Design Class Diagram

Software Development Process and the UP

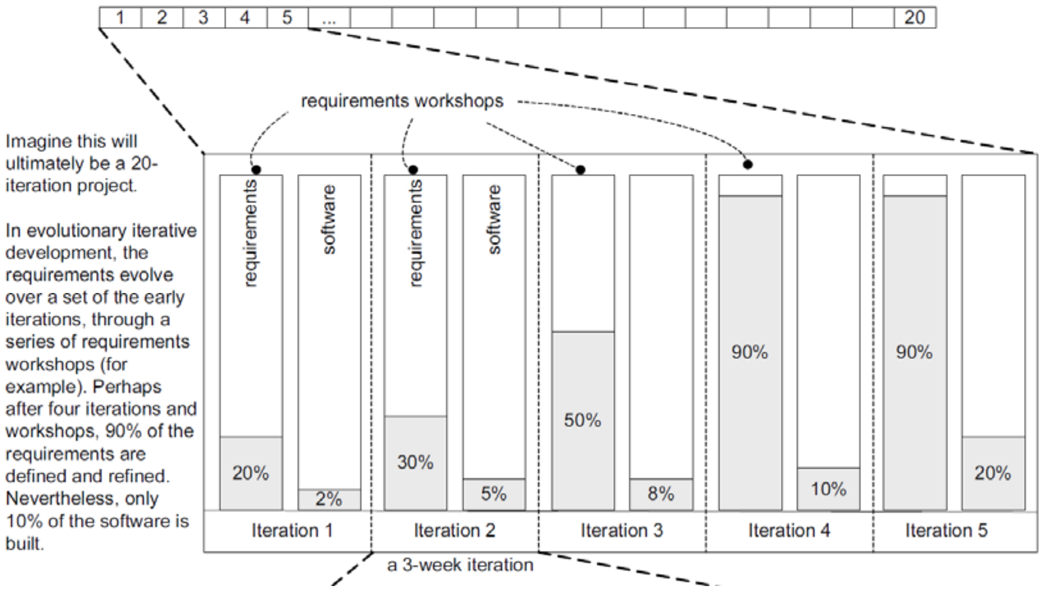
- **Software development process**
  - A **systematic approach** to building, deploying and possibly maintaining software

- **Unified Process (UP)**: a popular iterative software development process for building object-oriented systems
  - Inspired from Agile
  - Iterative
  - Provides an example structure for how to do OOA/D
  - Flexible (can be combined with practices from other OO processes)
  - A de-facto industry standard for developing OO software

Risk-Driven and Client-Driven Iterative Planning

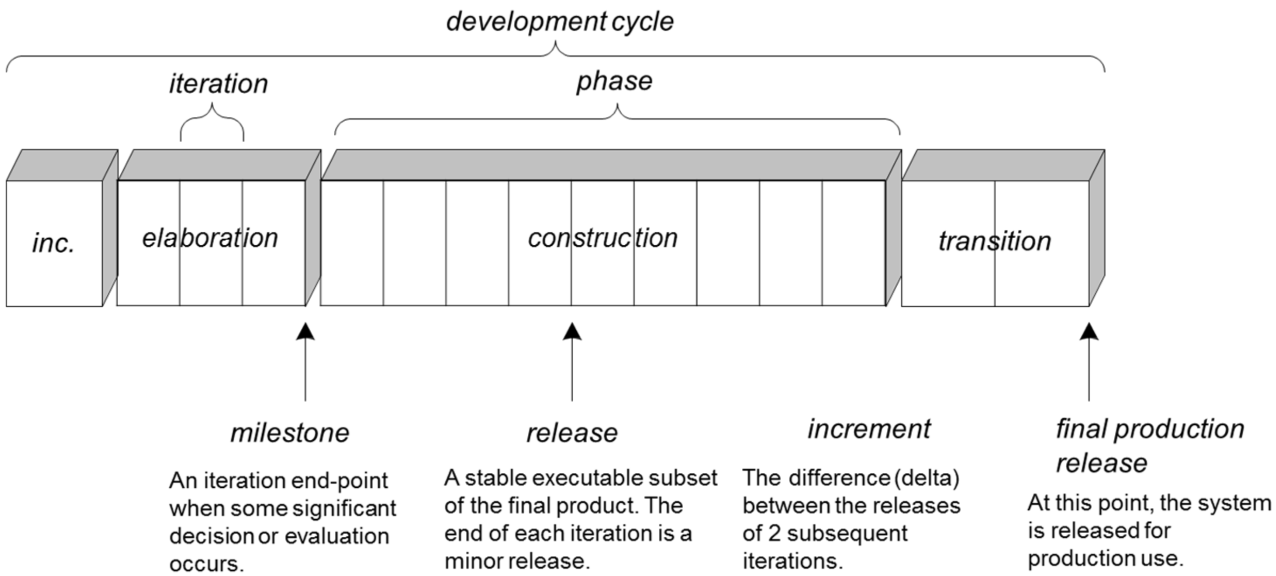
- The **UP** encourages a combination of **risk-driven** and **client-driven** **iterative planning**.
  - To identify and drive down the high risks, and
  - To build visible features that clients care most about.

- **Risk-driven** iterative development includes more specifically the practice of **architecture-centric** iterative development.
  - Early iterations focus on building, testing, and stabilizing the core architecture.

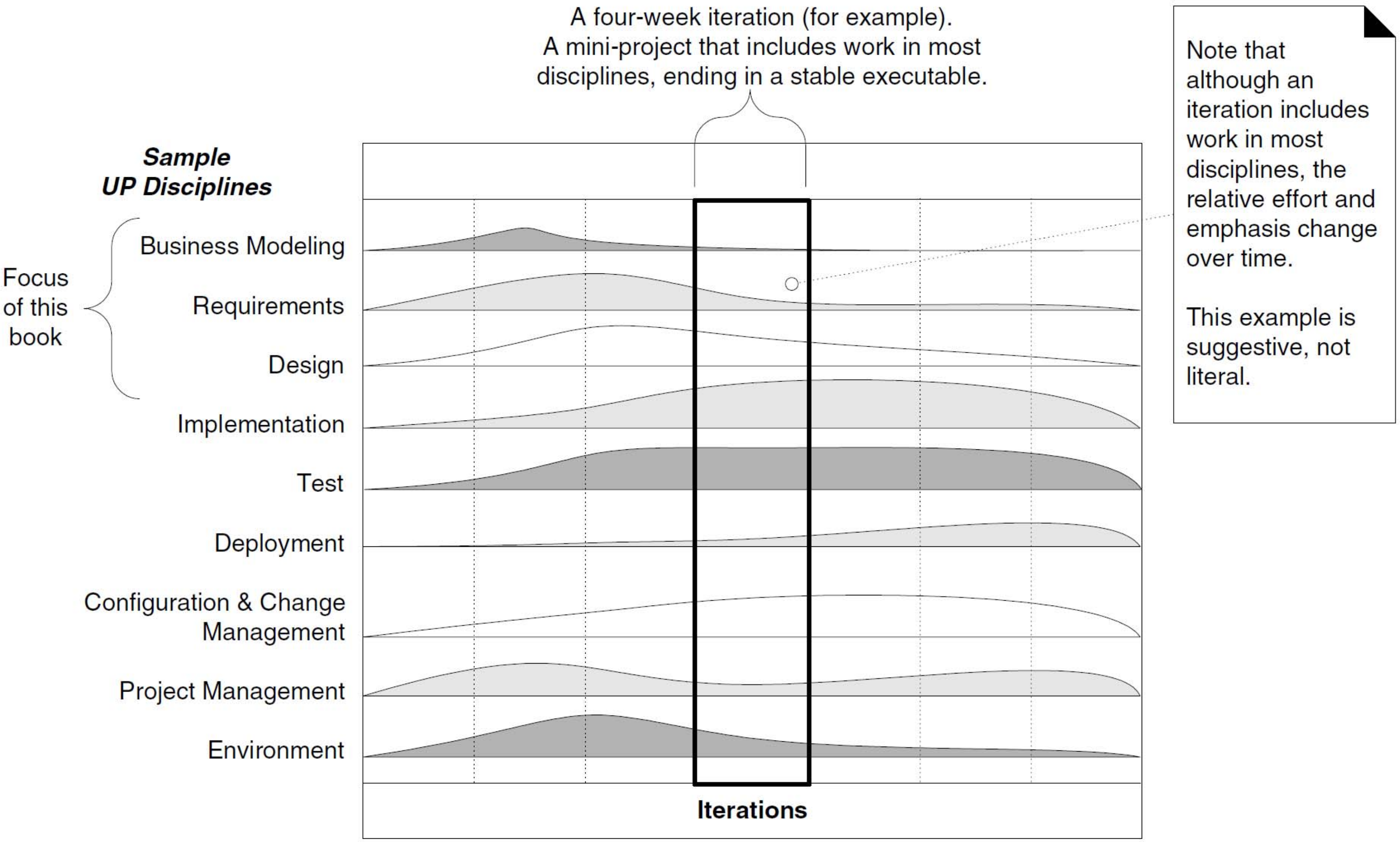


The UP Phases

- A UP project organizes the work and iterations across **4 major phases**:
  1. **Inception** : approximate vision, business case, scope, vague cost estimates
  2. **Elaboration** : refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates
  3. **Construction** : iterative implementation of the remaining lower risk and easier elements, and preparation for deployment
  4. **Transition** : beta tests, deployment

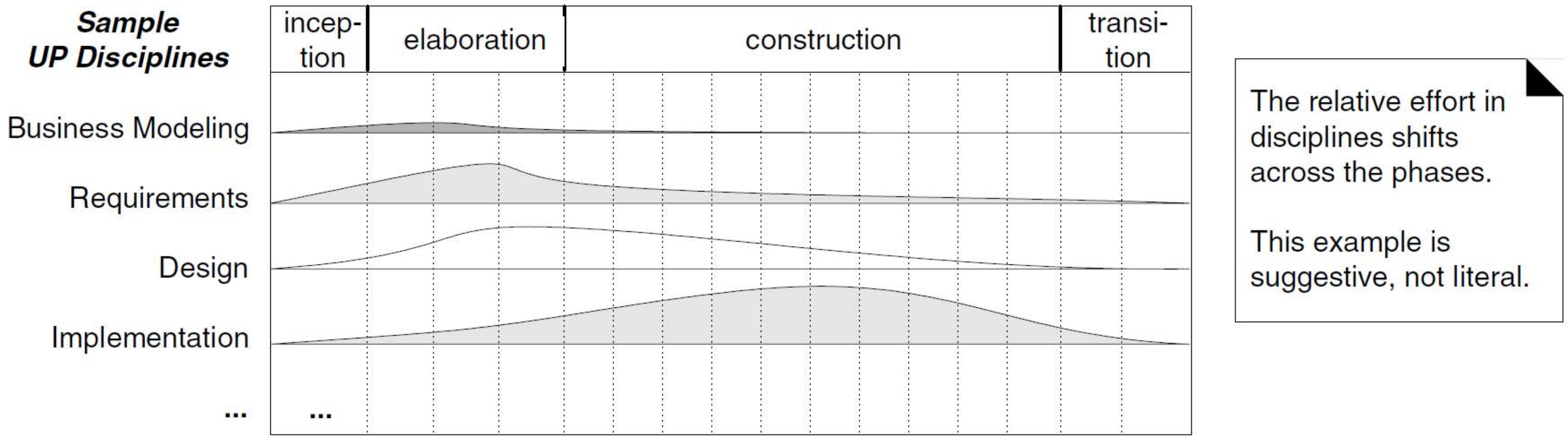


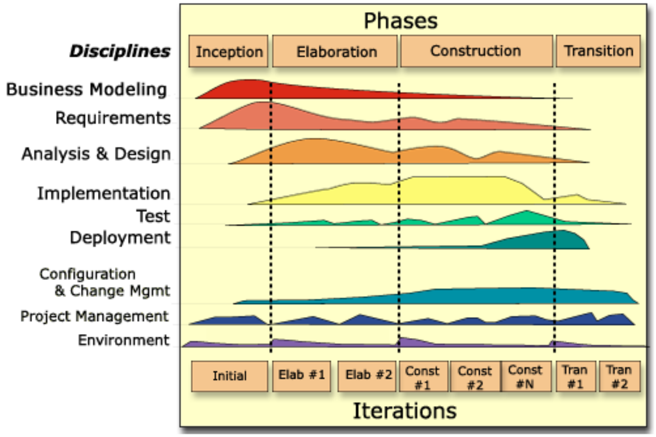
The UP Disciplines



Relationship Between the Disciplines and Phases

- The relative effort in disciplines shifts to across the phases.





The UP Artifacts and Timing

Sample Unified Process Artifacts and Timing (s-start; r-refine)

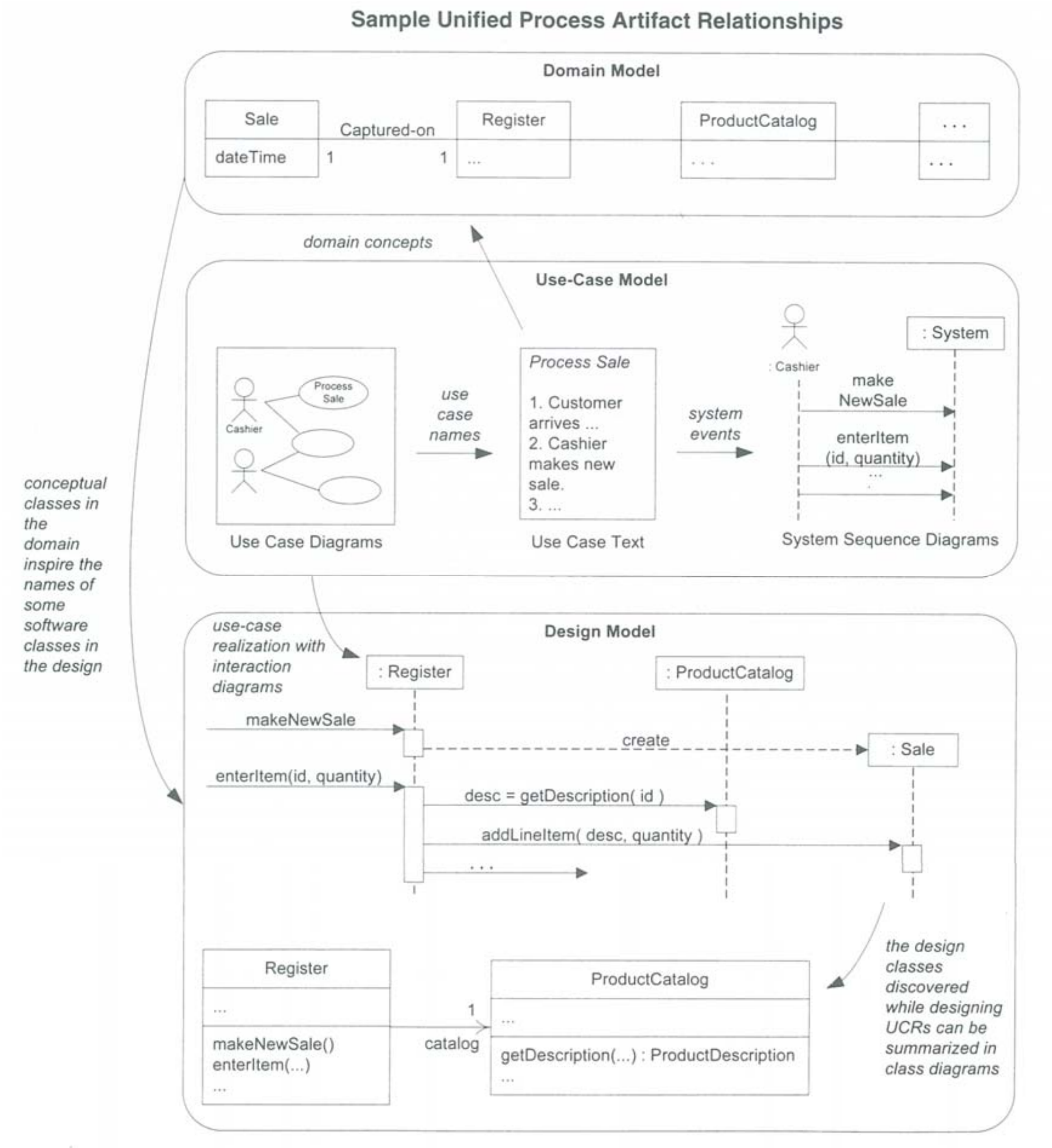
Discipline	Artifact Iteration➔	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model (code, html, ...)		s	r	r

+ System Sequence Diagram
+ Operation Contract

Design Model
  + Class Diagram
  + Interaction Diagram
  + Package Diagram

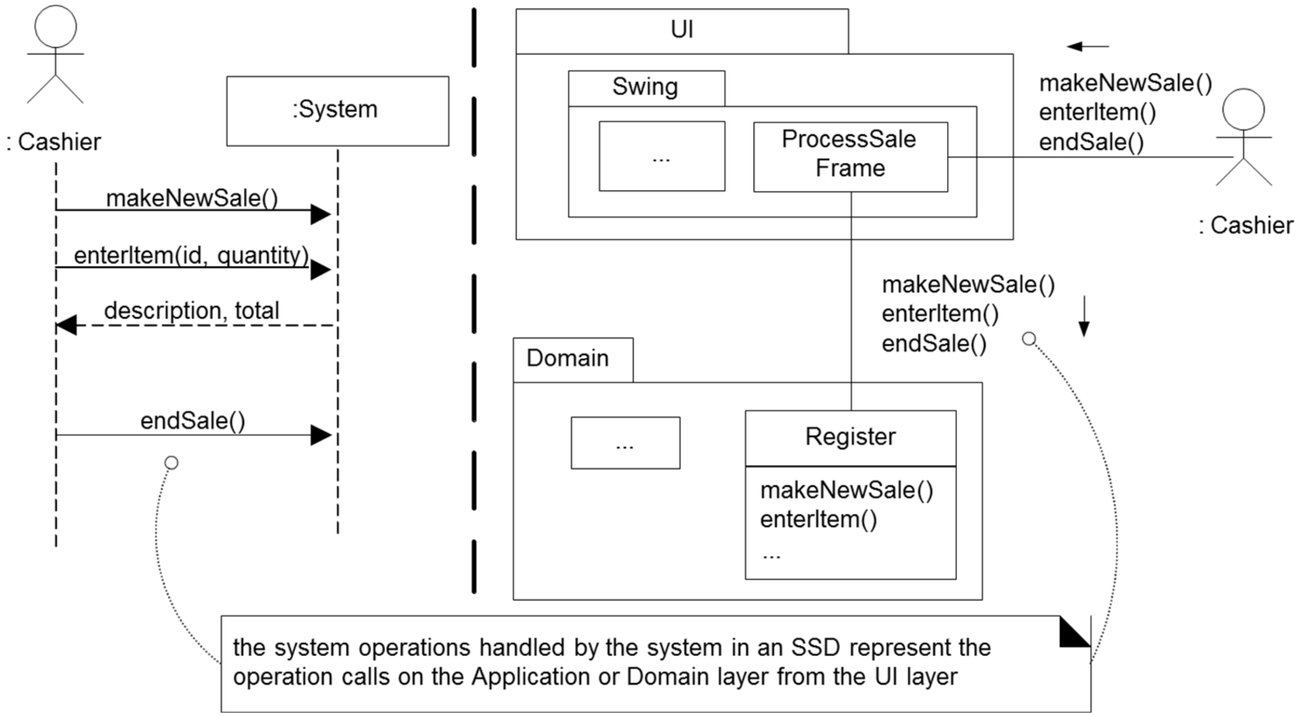
+ Statechart Diagram
+ Activity Diagram
+ Deployment Diagram

The UP Artifact Relationships



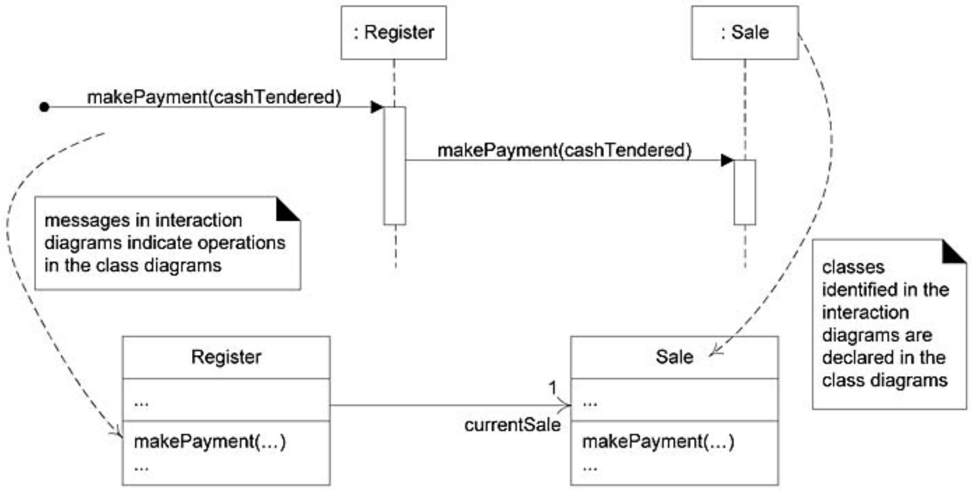
Connections Between SSDs, System Operations, and Layers

- In a well-designed layered architecture,
  - The UI layer objects will forward or delegate the requests from the UI layer (system operations) onto the domain layer for handling.
  - The messages sent from the UI layer to the domain layer will be the messages illustrated on the SSDs.



What's the Relationship between Interaction and Class Diagrams?

- From interaction diagrams, class diagrams can be generated iteratively.
  - When we draw interaction diagrams, a set of classes and their methods emerge.
  - Suggests a linear ordering of drawing interaction diagrams before class diagrams.
  - But in practice, these complementary dynamic and static views are drawn concurrently or iteratively.
  - Example:
    - if we started with the *makePayment* sequence diagram, we see that a *Register* and *Sale* class definition in a class diagram can be obviously derived.



OOD : Object-Oriented Design

- **OOD** is sometimes taught as some variation of the following:
  - "*After identifying your requirements and creating a domain model, then add methods to the appropriate classes, and define the messaging between the objects to fulfill the requirements.*"

- But, it is not enough, because OOD involves **deep principles**.
  - Deciding what methods belong to where and how objects should interact carries consequences should be undertaken seriously.

- Mastering OOD is hard.
  - Involving a large set of soft principles, with many degrees of freedom.
  - A mind well educated in design principles is important.
  - **Patterns** can be applied.

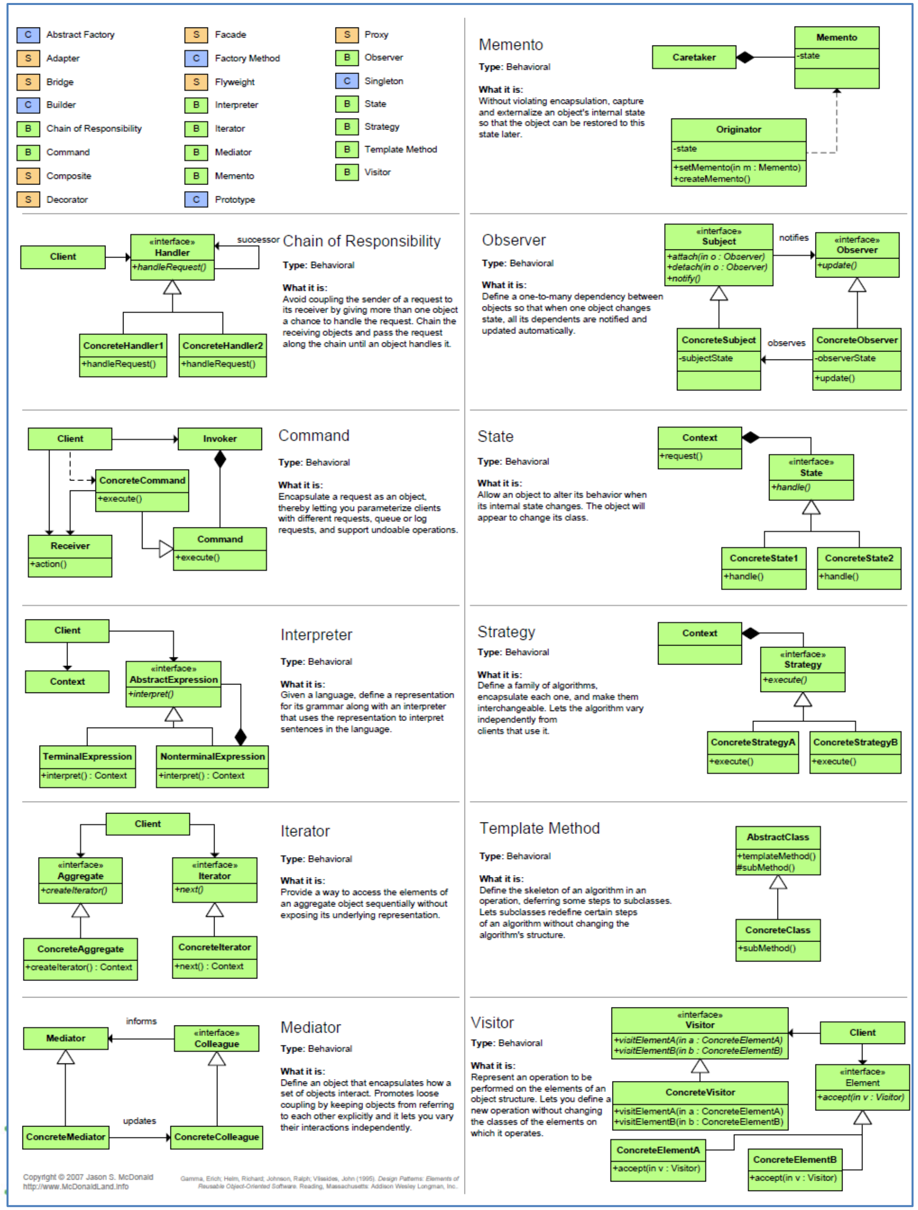
GRASP

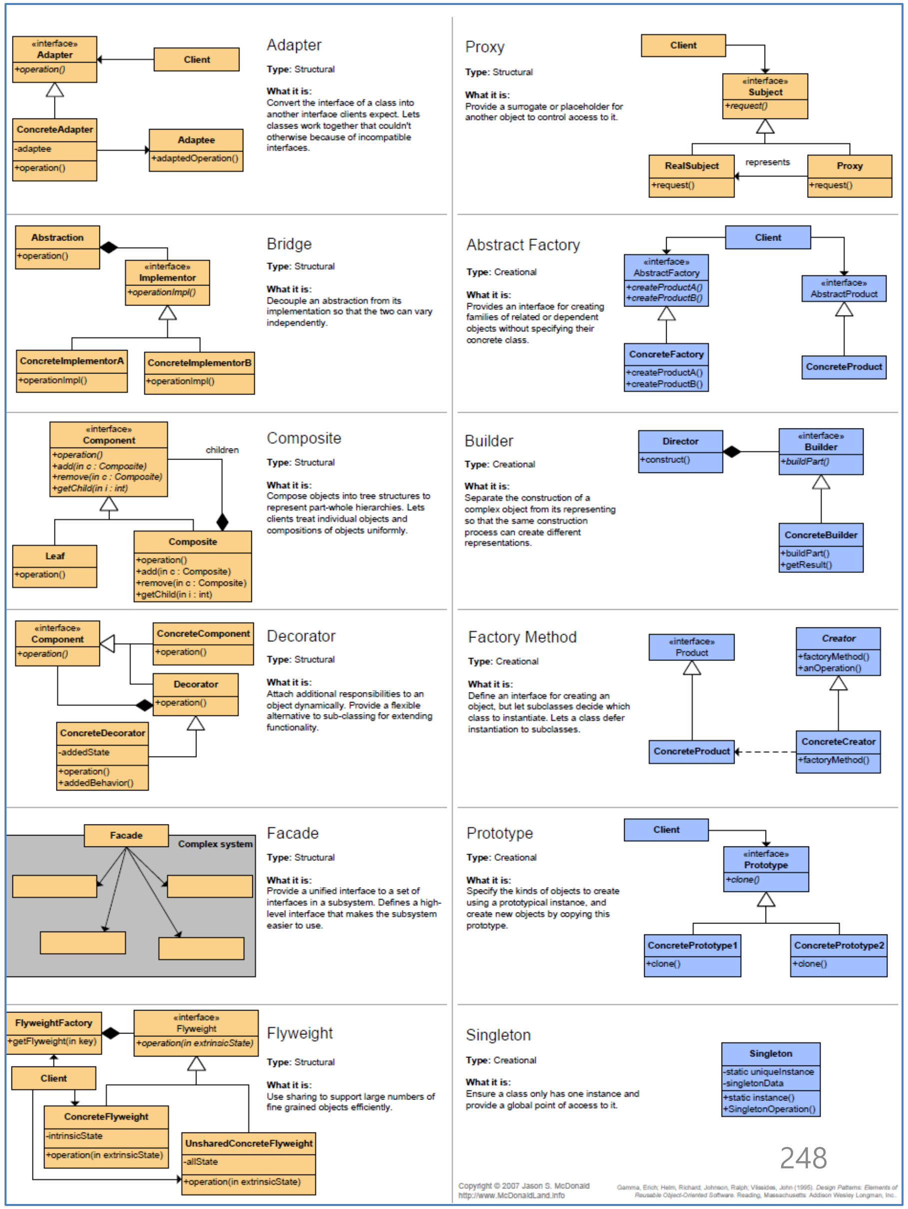
- **<u>9 basic OO design principles</u>** or basic building blocks in design.
  - Focusing on using the pattern style as an excellent learning aid for naming, presenting and remembering basic/classic design ideas

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

Pattern/ Principle	Description
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: 1. Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). 2. Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)
Low Coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

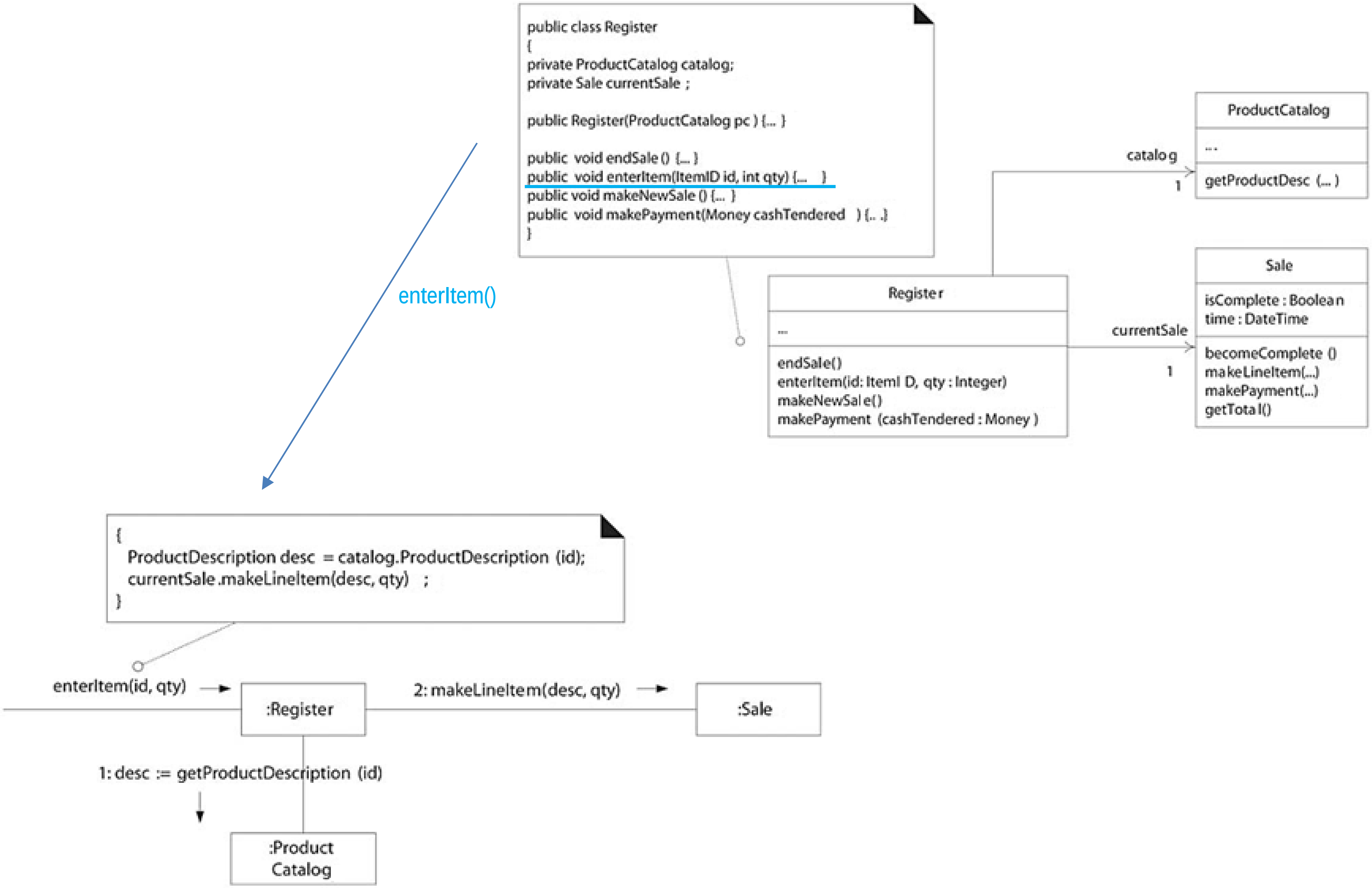
23 Design Patterns of GoF





Mapping Designs to Code

The Register.enterItem Method



An Overview of Object-Oriented Development - What We Covered?