Introduction to UML

Nguyen Van Vu
nvu@fit.hcmus.edu.vn

Topics

- Introduction to UML
- Use case diagrams
- Class diagrams

Unified Modeling Language (UML)

- A general-purpose modeling language for software system analysis and design

- Unifying works on OO development methods by Grady Booch, James Rumbaugh, Ivar Jacobson of Rational Software (now part of IBM)

- Adopted by Object Management Group (OMG) as a standard in 1997

- OMG now responsible for development of UML

Objectives of UML

- A non-proprietary general-purpose modeling language for software development

- Supports good practices for software analysis and design

- Intended to handle all concepts in a system

- Supports existing development processes, but does not include a development process

  - UML ≠ process of using UML
  - UML can be used with any development process

Objectives of UML (cont'd)

- UML is the language for
  - Visualizing
  - Specifying
 - Constructing
  - Documenting

Model

- A simplified representation of a thing from a specific perspective
    - Capturing important aspects of a thing while ignoring others
- Different kinds of models
    - Process models, use-case models, class model, object model, relationship models, etc.
- Forms of software models
    - Text
    - Diagrams
    - Pictures

Why modeling?

- Generally, modeling helps make things more precise, simpler, easy to understand, more complete

- Specifically, models help
  - capture and state precisely requirements and domain knowledge
  - link requirements and design
  - capture design decisions
  - general usable work products (software artifacts)
  - organize systems, especially large ones
  - explore and contemplate multiple solutions

Conceptual model of UML
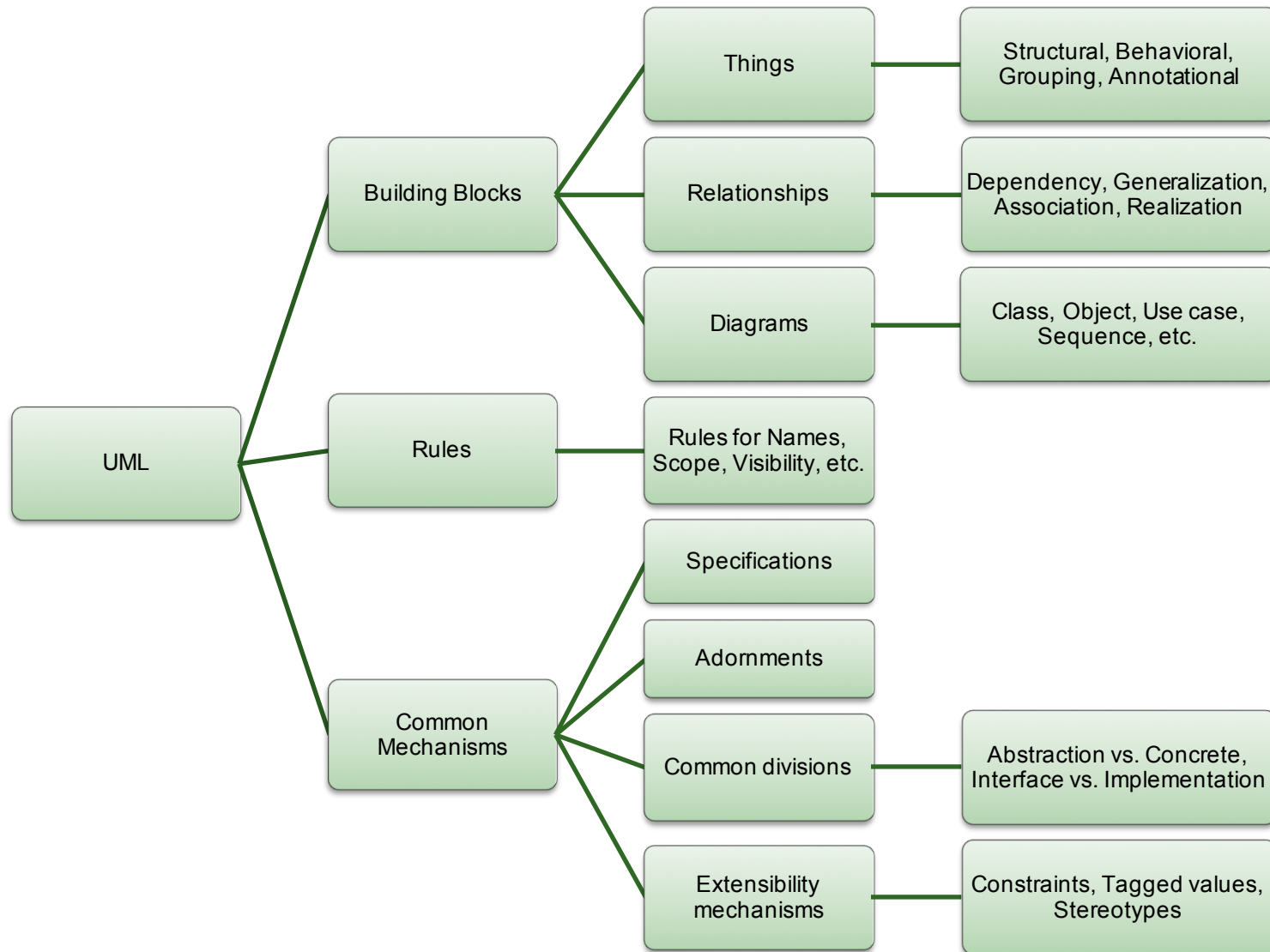
UML Major Areas and Views

Major Area	View	Diagrams	Main Concepts
structural	static view	class diagram	class, association, generalization, dependency, realization, interface
	use case view	use case diagram	use case, actor, association, extend, include, use case generalization
	implementation view	component diagram	component, interface, dependency, realization
	deployment view	deployment diagram	node, component, dependency, location

(Source: J. Rumbaugh, I. Jacobson, G. Booch, "The Unified Modeling Language Reference Manual", Addison Wesley, 2004.)

UML Major Areas and Views (cont'd)

dynamic	state machine view	statechart diagram	state, event, transition, action
	activity view	activity diagram	state, activity, completion transition, fork, join
	interaction view	sequence diagram	interaction, object, message, activation
		collaboration diagram	collaboration, interaction, collaboration role, message
model management	model management view	class diagram	package, subsystem, model

(Source: J. Rumbaugh, I. Jacobson, G. Booch, "The Unified Modeling Language Reference Manual", Addison Wesley, 2004.)

Essential views and diagrams

- Static view
  - Class diagram
- Use-case view
  - Use-case diagram
- Implementation view
  - Component diagram
- State machine view
  - Statechart diagram
- Activity view
  - Activity diagram
- Interaction view
  - Sequence diagram

- We will study and apply these in details

Overview of UML Modeling

- UML provides approaches to modeling both structural and behavioral aspects of a system

- Static modeling – structural diagrams
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram

- Dynamic modeling – behavioral diagrams
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

Topics

- Introduction to UML
- Use case diagrams
- Class diagrams

Use Case Diagrams

- Use case diagrams are used to model the use case view of a system as seen by end users, analysts, and testers

- Use case diagrams are usually a part of requirement specifications

    - Accompanying use case specifications

- Main elements

    - Actors
    - Use cases
    - Relationships

Doctor — Specify Prescription

Patient — Provide Health Conditions
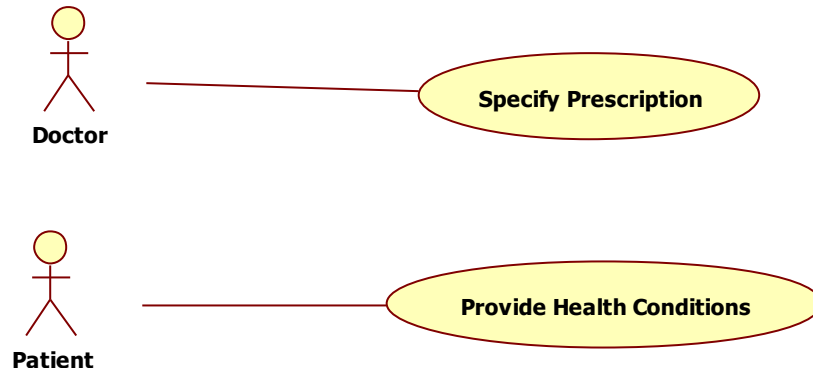
Benefits of Use Case Diagrams

- Capturing and visualizing precisely functional requirements

- Making connections between requirements, design, implementation, and testing

- Facilitating communication between users and analysts, designers, testers, implementers

- Supporting identifying functionality easily

- Supporting validation/testing
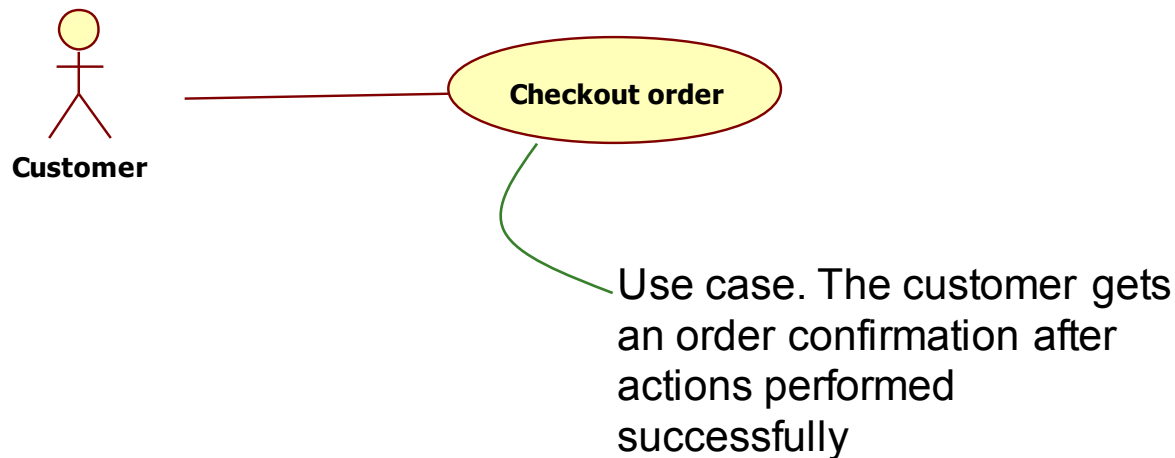
→ Second most commonly used diagrams

Actors

- An actor represents an external person, process, or **anything** that interacts with the system

- It represents a role that a person, process, or thing plays with the system

  - One physical user can play multiple roles. Thus, multiple actors can be bound to one physical user

  - Multiple users have the same role, hence, represented by one actor

- Between actors may have a generalization

Employee

Doctor
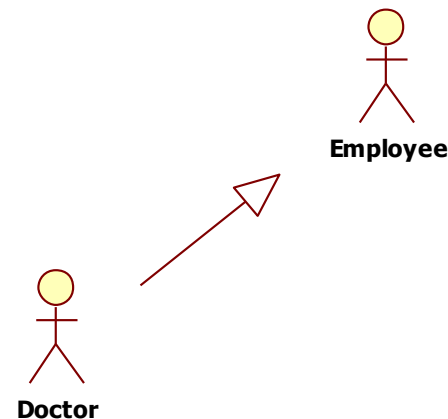
Use Cases

- A use case describes a set of sequences of actions performed by an actor to produce observable results value to the actor

Customer

Checkout order

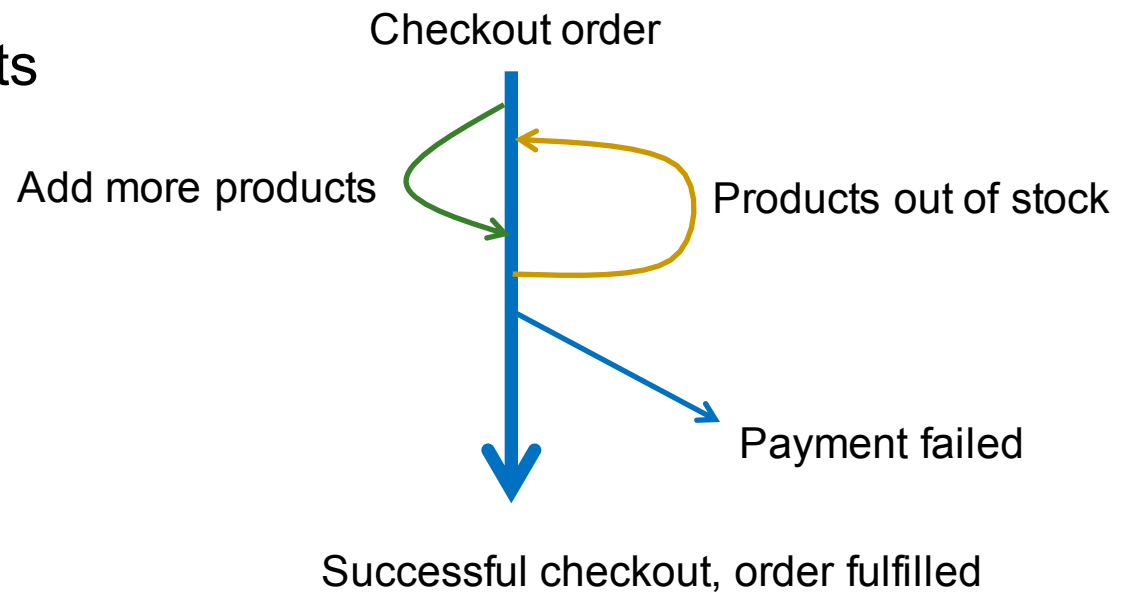Use case. The customer gets an order confirmation after actions performed successfully

Use Cases (cont'd)

- At the system level, use cases represent external behavior of a system that is visible to outside users

- A use case specifies WHAT the system does, but not HOW it does

- A use case hides internal structure and operations of a system. It usually represents system-level functions of a system

Description of Use Cases

- The behavior of a use case is described by a flow of events

- A use case has one main flow (basic flow) and alternative flows
    - Regular variants
    - Odd cases
    - Exceptions

Checkout order

Add more products

Products out of stock

Payment failed

Successful checkout, order fulfilled

Scenarios

- A use case describes a set of sequences which each sequence in the set represents a possible flow in the use case

- A scenario is a specific sequence of events happening

- A scenario is viewed as an instance of a use case

  - A scenario is concrete or *real*

  - Remember class vs. object?
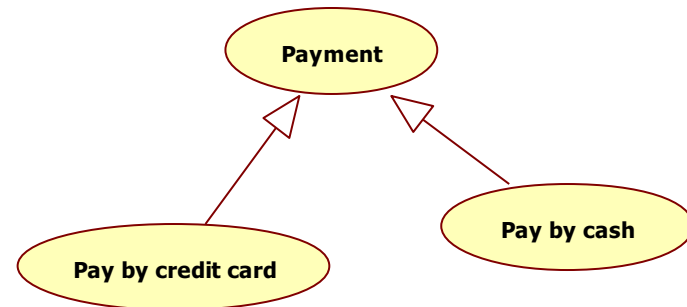
Types of Use Case Relationships

- ## Association
  - ❑ Between actors and use cases
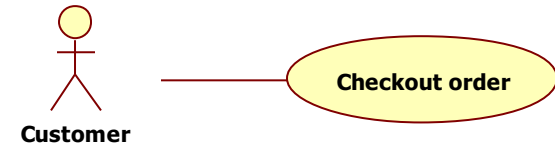
- ## Generalization
  - ❑ Between general use cases and more specific ones

- ## Include
  - ❑ Between use cases
- ## Extend
  - ❑ Between use cases
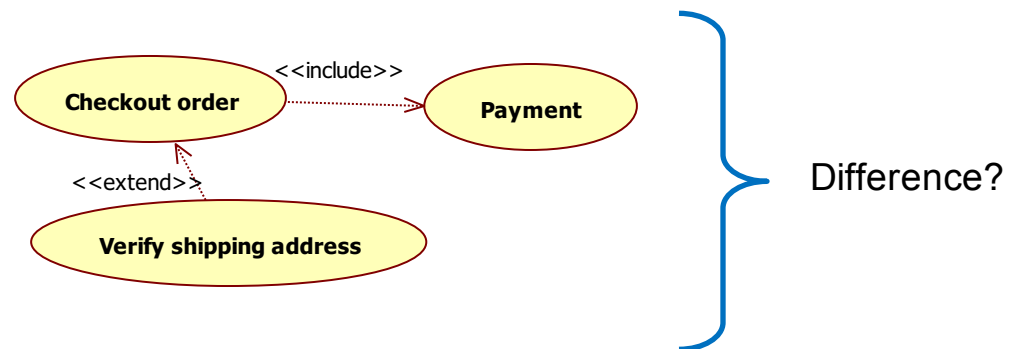
Include vs. Extend

- An include relationship between use cases specifies that the base use case **<u>explicitly</u>** incorporates the behavior of another use case
  - The included use case does not stand alone
  - The base use case is *aware* of its included use case
- An extend relationship between use cases specifies that the base use case **<u>implicitly</u>** incorporates the behavior of another use case at location specified indirectly by the extending use case
  - The extending use case may stand alone
  - The base use case is not aware of the extending use case

Include vs. Extend (cont'd)

Online shopping

Customer

Checkout order

<<include>>

Payment

<<extend>>

Verify shipping address

Firefox Web browser

Firefox User

Load and Display web page

<<include>>

Show images

<<extend>>

Read web page loud
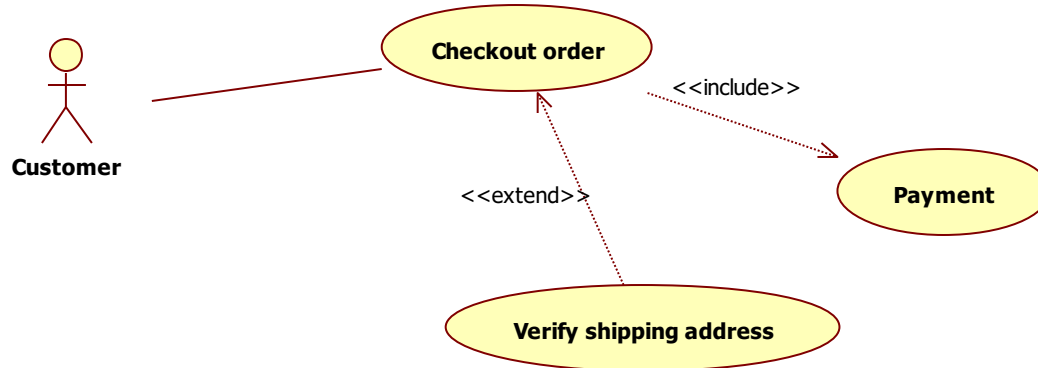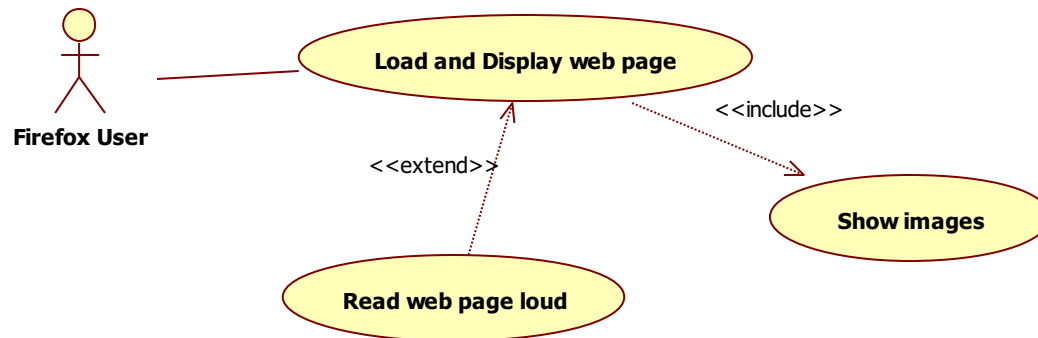
Notes on Use Case Diagrams

- A well-defined use case should
  - specify identifiable and reasonably atomic behavior of the system
  - include common behavior from included use cases
  - distribute variations into extending use cases
- Number of use cases depends on how to include or extend system behavior
- Only include use cases and actors essential to understanding an aspect
- Layout of elements is important to understanding
  - minimize line crosses
  - group related elements together

Topics

- Introduction to UML
- Use case diagrams
- Class diagrams

Class Diagram

- ## The most commonly used diagram in practice
- ## Main elements
  - ❑ Classes
  - ❑ Interfaces
  - ❑ Relationships
  - ❑ Common mechanisms

Person
+name: String
#age: int
#address: String

{age > 0}

Patient
+healthPrecondition: String

Employee
-title: String
-baseSalary: float
<<constructor>>+Employee()
<>+calcSalary(year: int): float

Hospital
+name: String
+address: String

Doctor
+specialty: String
+cases: int
+calcSalary(year: int): float

Nurse
+hourRate: float
+calcSalary(year: int): float
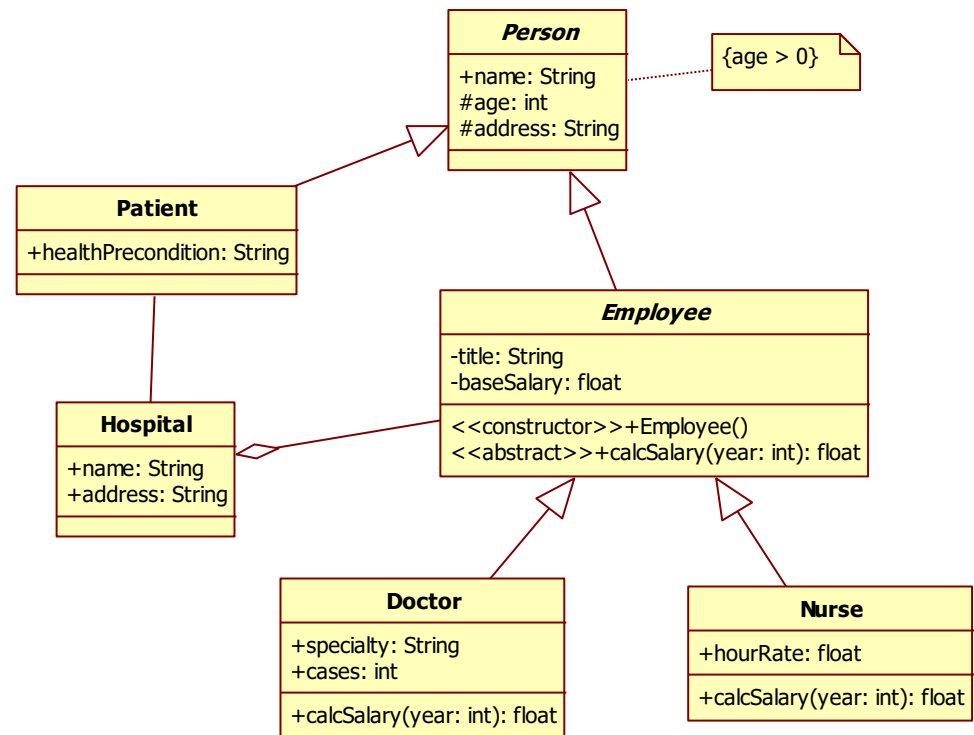
Class

- Defines the set of common objects that have same the same attributes, operations, relationships, and semantics
- Represents a thing
- Notation

Employee
-title: String -baseSalary: float
<<constructor>>+Employee() <>+calcSalary(year: int): float

Name: must be unique within its group

Attributes

Operations

Attribute

- Defines data that characterize a class
- An abstraction of the kind of data or object
  - *title* is an attribute of the kind of *String* object
- Data type is specified by a semicolon ":"

Employee
-title: String -baseSalary: float
<<constructor>>+Employee() <>+calcSalary(year: int): float

Title and *baseSalary* are two attributes of *String* and *float* data types, respectively

Operation

- An operation specifies a service that can be requested from objects of the class

- Attribute and operation visibility

protected

private

public

Doctor
#specialty: String -cases: int
+calcSalary(year: int): float
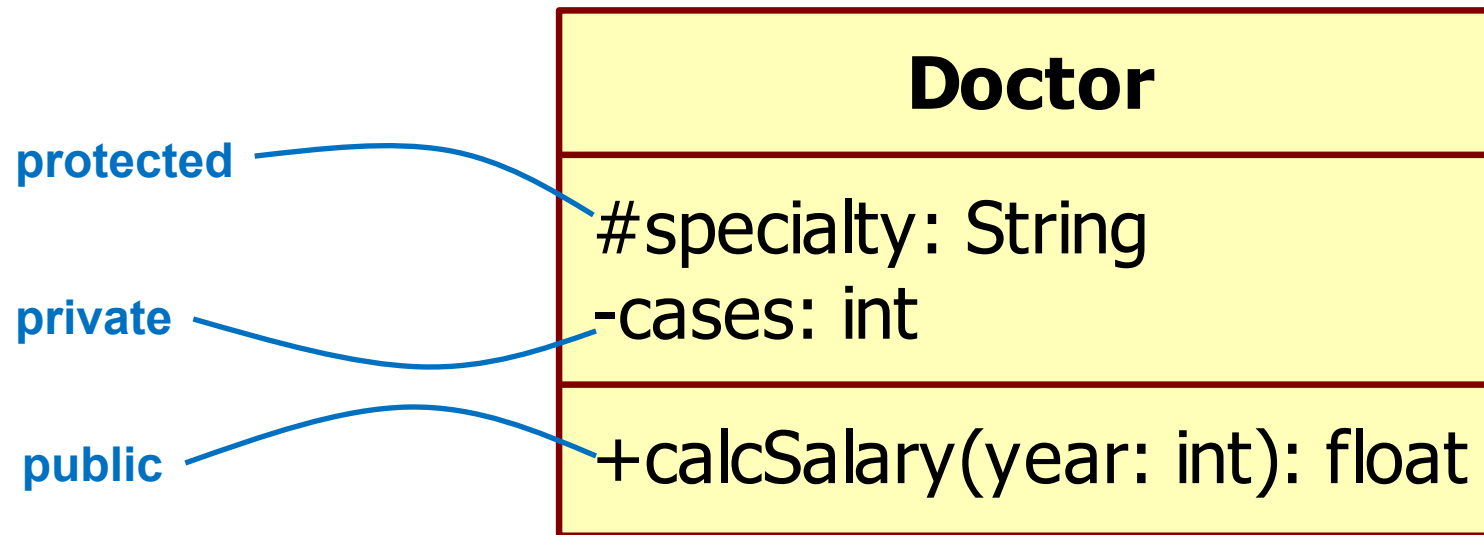
Interface

- A element that has a set of operations characterizing its behavior

- Notation
  - A circle with a name
  - OR a class with stereotype <<interface>>
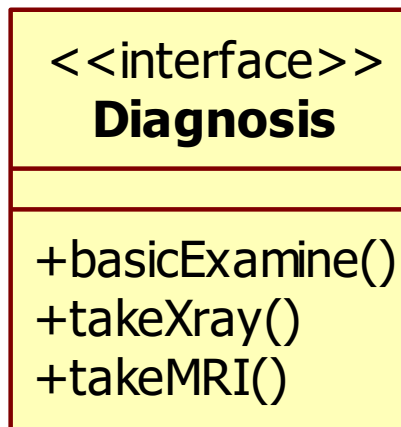
<<interface>>
Diagnosis

+basicExamine()
+takeXray()
+takeMRI()

Diagnosis

Class Diagram

- The most commonly used diagram in practice
- Main elements
  - Classes
  - Interfaces
  - Relationships
  - Common mechanisms

Class Relationship Types

- Most classes associate with others
- Classes have different types of relationship with each other
    - Dependency
    - Generalization
    - Association

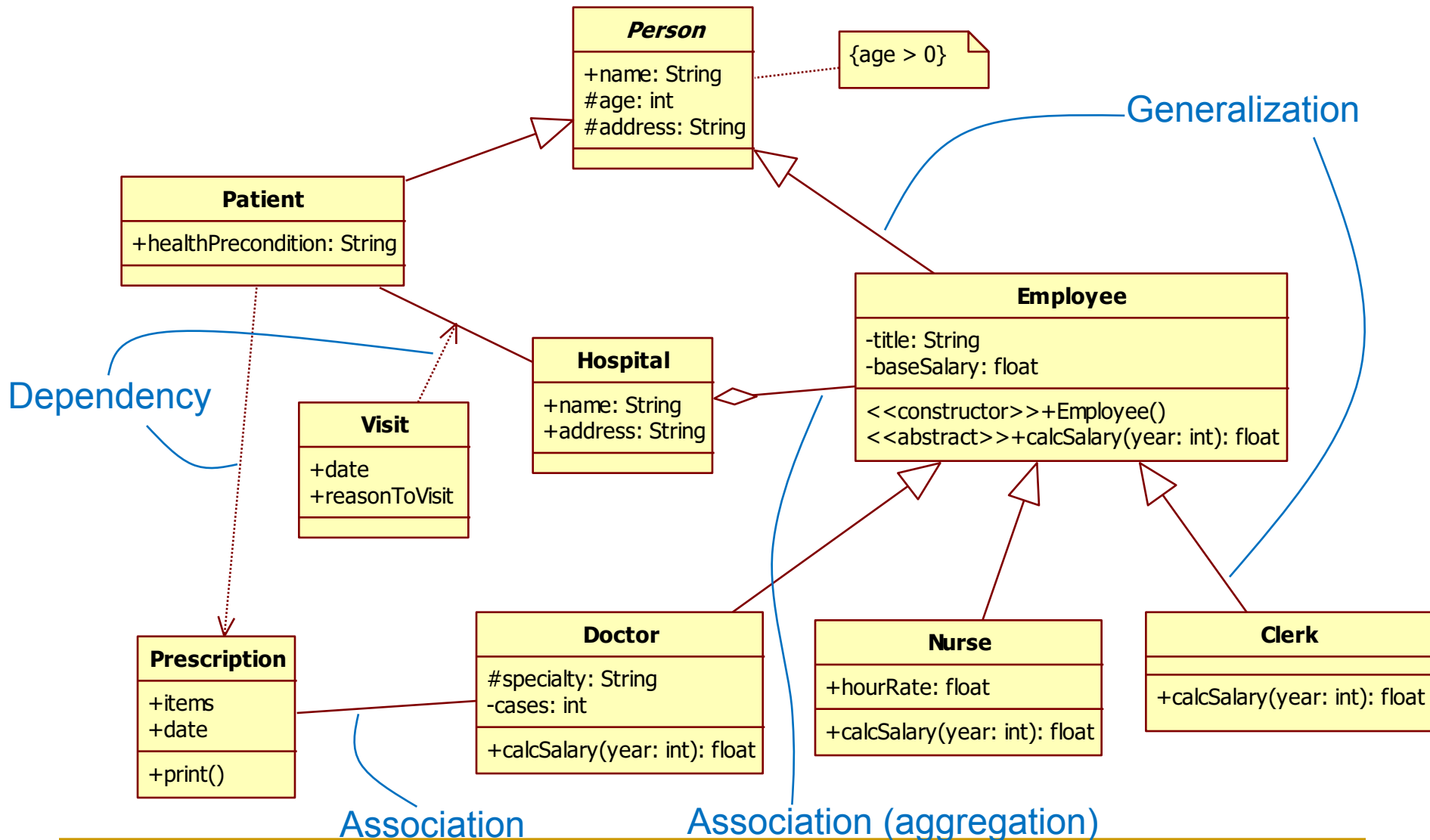# Class Relationship Types (cont'd)

Person
+name: String
#age: int
#address: String

{age > 0}

Generalization

Patient
+healthPrecondition: String

Employee
-title: String
-baseSalary: float

<<constructor>>+Employee()
<>+calcSalary(year: int): float

Dependency

Hospital
+name: String
+address: String

Visit
+date
+reasonToVisit

Prescription
+items
+date

+print()

Doctor
#specialty: String
-cases: int

+calcSalary(year: int): float

Nurse
+hourRate: float

+calcSalary(year: int): float

Clerk

+calcSalary(year: int): float

Association

Association (aggregation)

Dependency Relationship

- ## Represents one class uses another
  - Change in one class may affect the one that uses it
  - e.g., two classes have dependency relationship if
    - one operation calls another operation on another class
    - one class is used as a parameter in another class

- ## Dependency is less restrictive than other relationships

Patient
+healthPrecondition: String

Prescription
+items +date
+print()
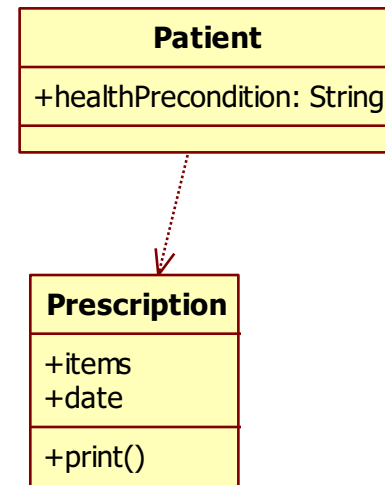
Patient uses *Prescription*

Generalization Relationship

- A relationship between a general class and more specific class (superclass and subclass)

Person
+name: String
#age: int
#address: String

Patient
+healthPrecondition: String

Employee
-title: String
-baseSalary: float

<<constructor>>+Employee()
<>+calcSalary(year: int): float

Doctor
#specialty: String
-cases: int

+calcSalary(year: int): float

Nurse
+hourRate: float

+calcSalary(year: int): float

Clerk

+calcSalary(year: int): float
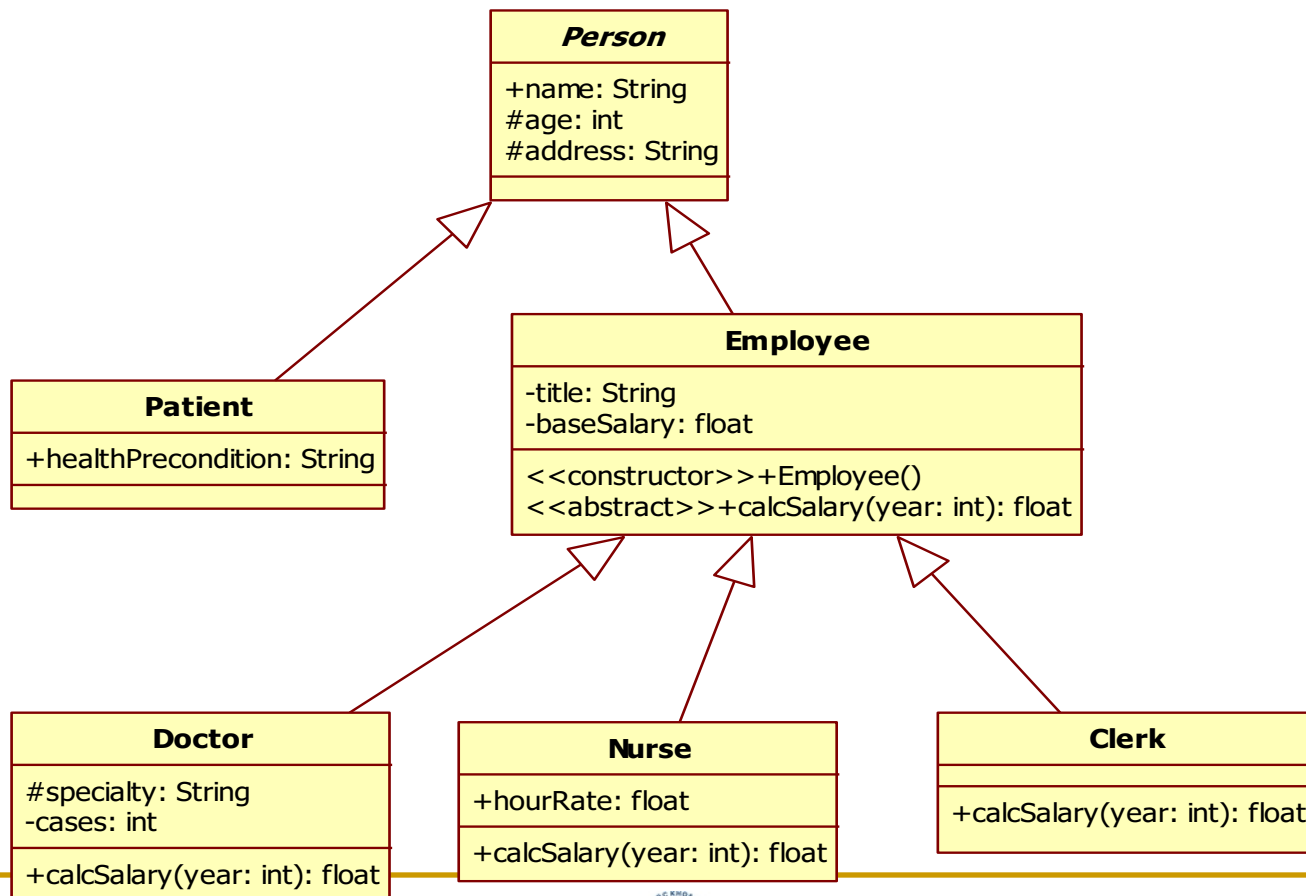
Association Relationship

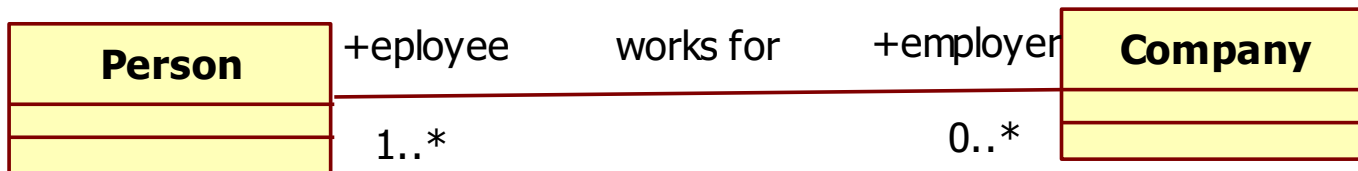- A relationship specifying objects of one class are connected to objects of another

- Typically, one object holds objects (instances) of the same class or another

- Association can have a name, roles at both ends
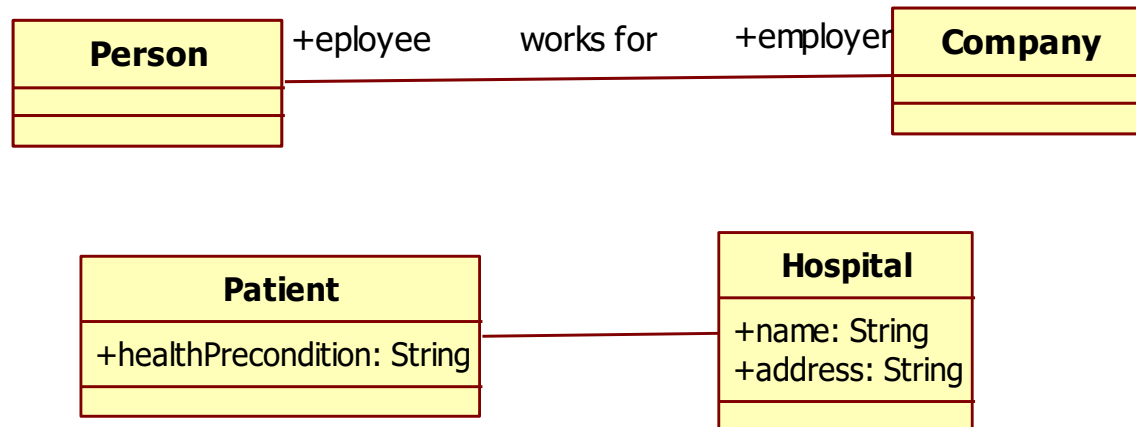
Multiplicity

- Refers to how many objects may be connected across an instance of an association

Person	+eployee works for +employer	Company
	1..* 0..*	

Aggregation and Composition

- Two special types of the association that one object of the whole has objects of the part

- Composition

  - The part may belong to only one whole class (composite)

- Difference between the relationships below?
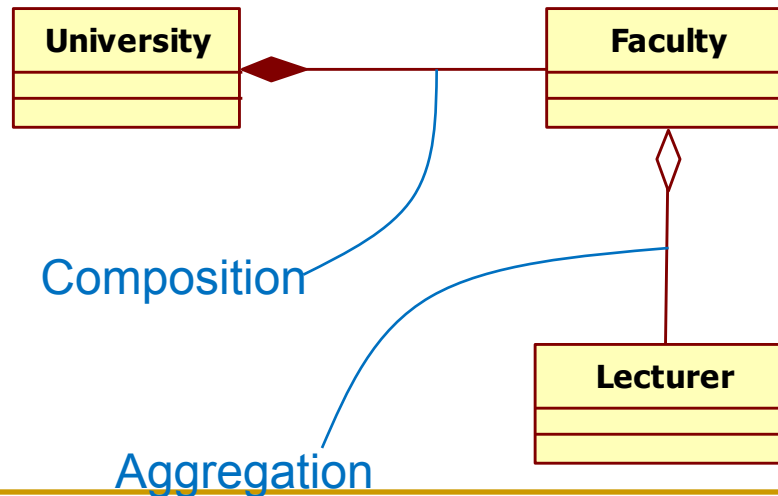
Class Diagram

- The most commonly used diagram in practice
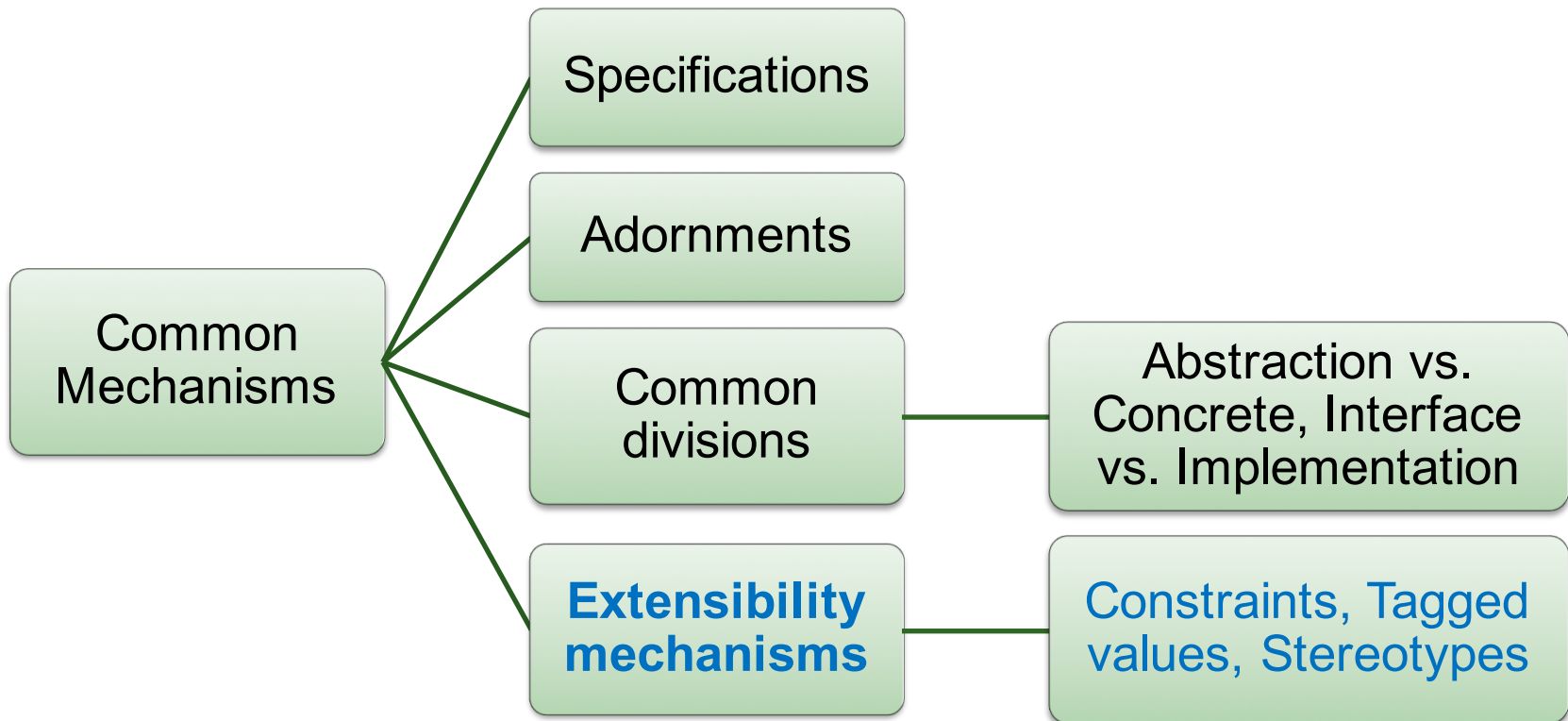- Main elements
  - Classes
  - Interfaces
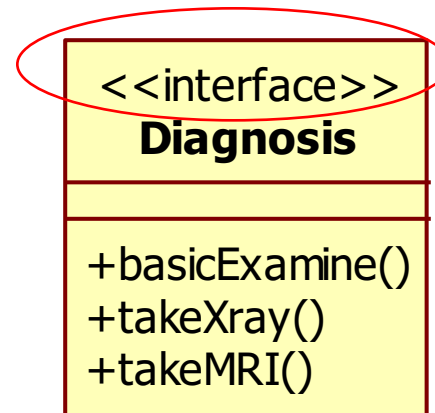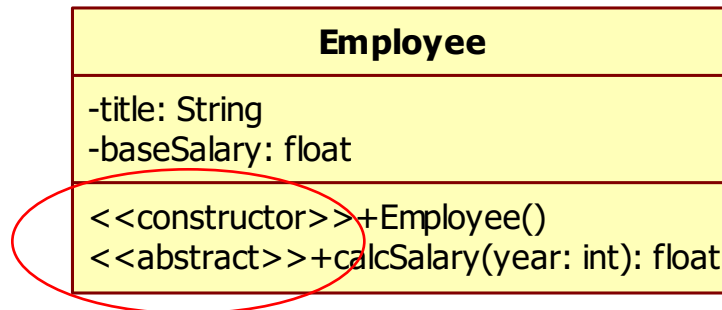 - Relationships
  - Common mechanisms

Common Mechanisms

```
                          ┌─────────────────┐
                          │  Specifications │
                          └─────────────────┘
                          ┌─────────────────┐
                          │   Adornments    │
                          └─────────────────┘
┌──────────────┐          ┌─────────────────┐        ┌──────────────────────┐
│   Common     │          │    Common       │        │  Abstraction vs.     │
│  Mechanisms  │          │    divisions    │────────│  Concrete, Interface │
└──────────────┘          └─────────────────┘        │  vs. Implementation  │
                          ┌─────────────────┐        └──────────────────────┘
                          │ Extensibility   │        ┌──────────────────────┐
                          │ mechanisms      │────────│ Constraints, Tagged  │
                          └─────────────────┘        │ values, Stereotypes  │
                                                     └──────────────────────┘
```

Stereotypes

- An extension of the UML vocabulary provides specifics to an existing UML element

- It can be specified for class, attribute, operation, relationship, etc.
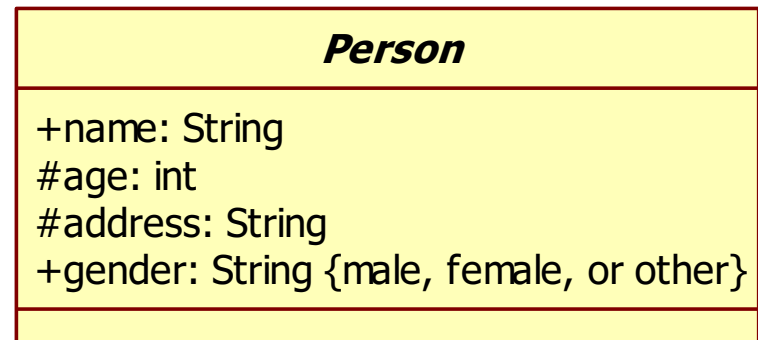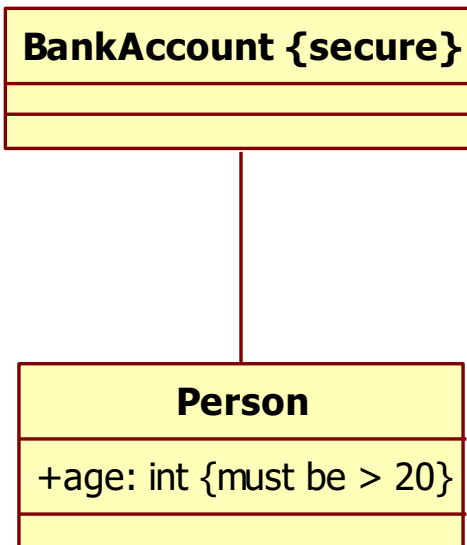
- A stereotype is specified within << >>

Employee
-title: String -baseSalary: float
<<constructor>>+Employee() <>+calcSalary(year: int): float

<<interface>> **Diagnosis**
+basicExamine() +takeXray() +takeMRI()

Tagged Values

- Allows to provide extensions to a property of a UML element, e.g., specifying default values, ranges, etc.
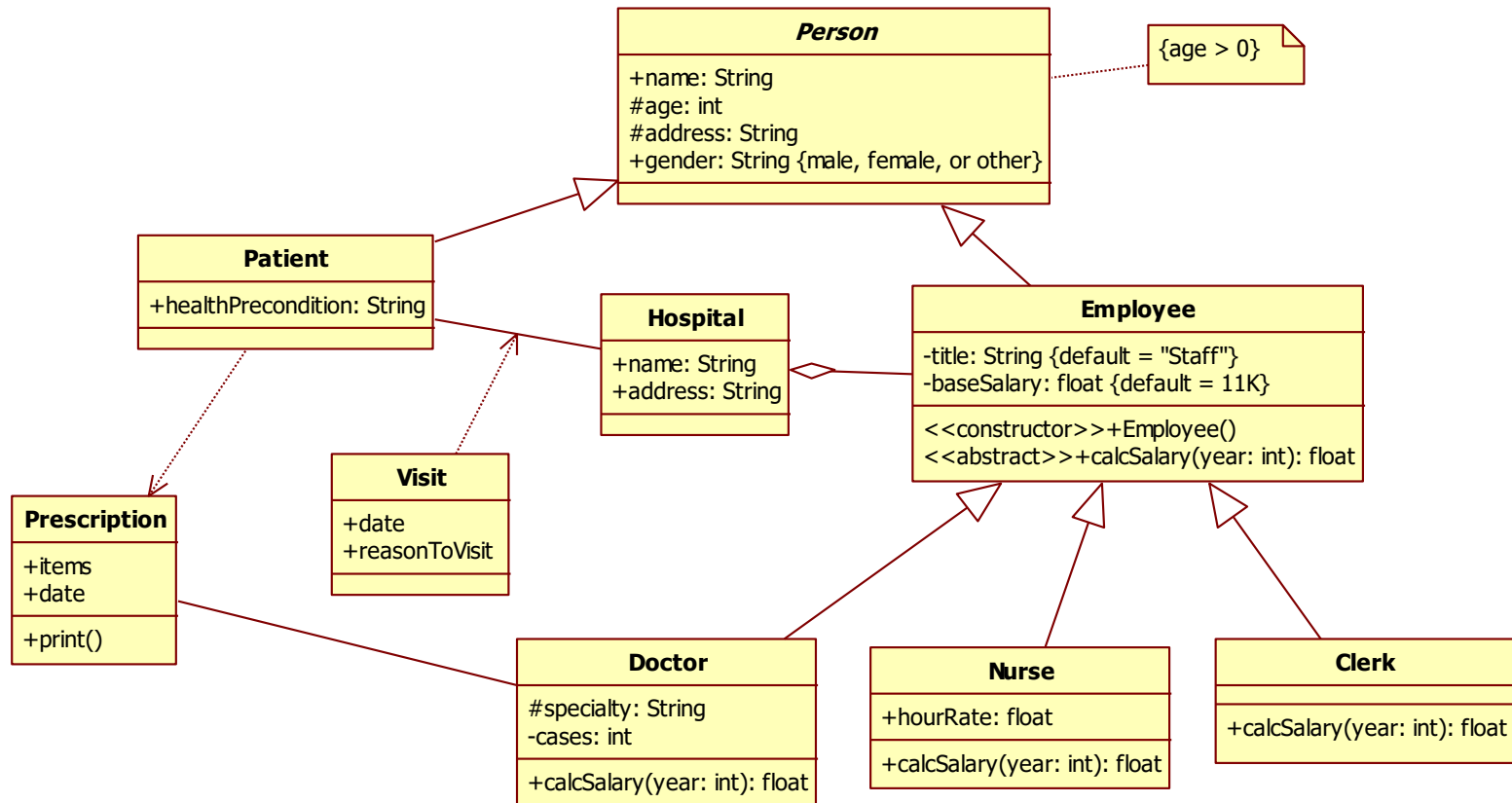
- Notation

  {tagname = value, tagname = value}

Employee
-title: String {default = "Staff"} -baseSalary: float {default = 11K}
<<constructor>>+Employee() <>+calcSalary(year: int): float

Constraints

- Allow to add new rules or modify existing ones for a UML element

- Notation: any text within curly braces {}

BankAccount {secure}

Person
+age: int {must be > 20}

Person
+name: String #age: int #address: String +gender: String {male, female, or other}

Putting Things Together

Some Tips

- Classes, relationships, etc. are abstractions of things (objects, links, etc.)

- To identify abstractions, specify things that users and implementers/developers use

- To identify attributes and operations
  - Identify **responsibilities** of each abstraction
  - Provide attributes and operations to perform these responsibilities

Some Tips (cont'd)

- **Example, identifying attributes and operations of doctor**
  - What are responsibilities of doctors?
    - Diagnose patients' problems
    - Consult patients
    - Write prescriptions
    - Get paid
    - Has title
    - Has salary
    - Has bonus
  - Specify attributes and operations based on the responsibilities identified above

Some Tips (cont'd)

- Group related classes into packages
- Show only related classes in the same diagrams
- Show only classes, operations, attributes that are important to understand
  - You can suppress operations and attributes from their classes
- Don't try to identify all possible classes, attributes, and operations at once
  - Important classes are identified first, and gradually identify other later