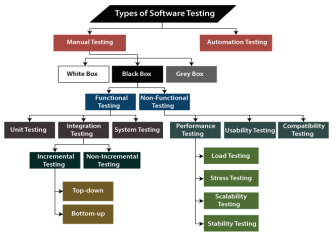
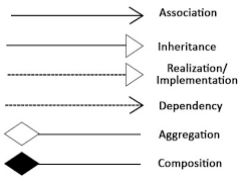
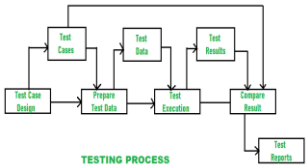
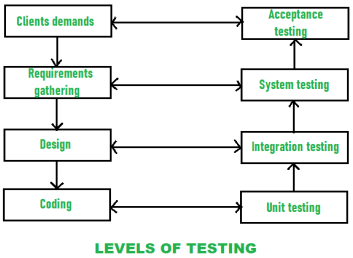






Software Design and Testing

Structural Modeling



Outline

- Classes,
- Relationships and Other Common Mechanisms,
- Types of Diagrams,
- Class Diagrams,
- Interfaces: Types and Roles,
- Object Diagrams

What is an Object?

- The term object was first formally utilized in the Simula language to simulate some aspect of reality.
- An object is an entity.
  - It knows things (has attributes)
  - It does things (provides services or has methods)
- Attributes or properties describe object's state (data) and methods define its behavior.
- An Object is anything, real or abstract, about which we store data and those methods that manipulate the data.
- **What's object: Identity, State, Behavior : Sequence Diagram,** Statechart Diagram
- Messages and methods

3

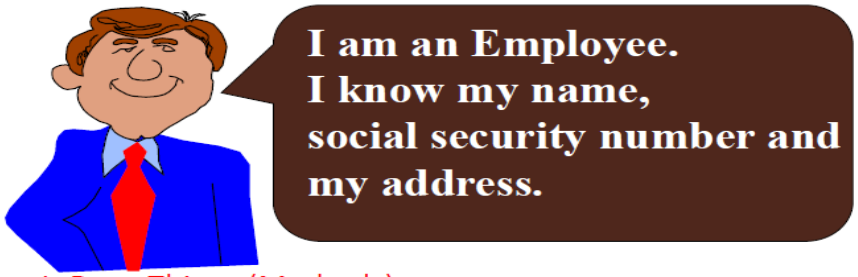
What is an Object?

- Conceptually, there are many ways to think of an object
  - something that can be seen or touched
  - a thing to which some action is directed
  - something that performs an action
- The structure and behaviour of similar objects are defined in their common class
- Objects have three properties: identity , state, and behaviour

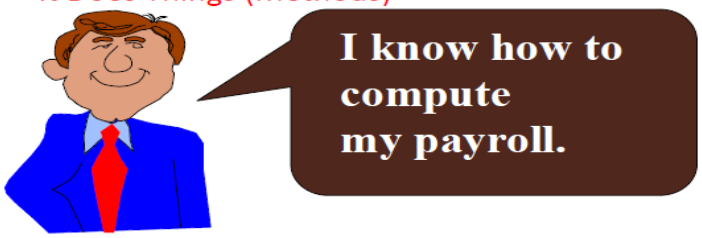
4

What is an Object?

- It Knows Things (Attributes)



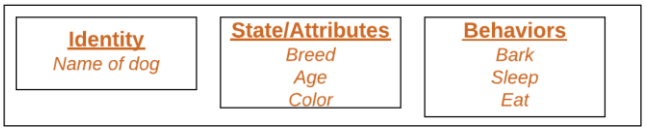
- It Does Things (Methods)



5

Object Property 1: Identity

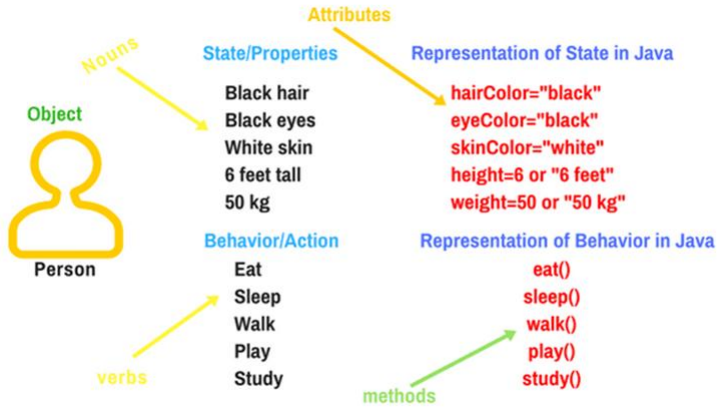
- Identity is that property of an object which distinguishes it from all other objects
- Most programming languages use variable names to refer to objects
- Keep in mind, however, that an object may not have a name; Similarly, an object might have multiple names (aliases)
- For this reason, there is a subtle distinction made between the concepts of "name" and "identity"



6

Object Property 2: State

- The state of an object encompasses all of the descriptions of the object plus the current values of each of these descriptions



7

Object Property 3: Behavior

- Behaviour is how an object acts and reacts, in terms of its state changes and message passing
- The state of an object represents the cumulative results of its behaviour
- In object-oriented programming, a behaviour is invoked by sending a message to an object

8

Examples of Object

- There are many physical objects we can examine right in this room
  - each person is an object
  - any chair is not an object
  - each light bulb is an object
  - Any book is not an object
  - this room itself is an object (full or not)

9

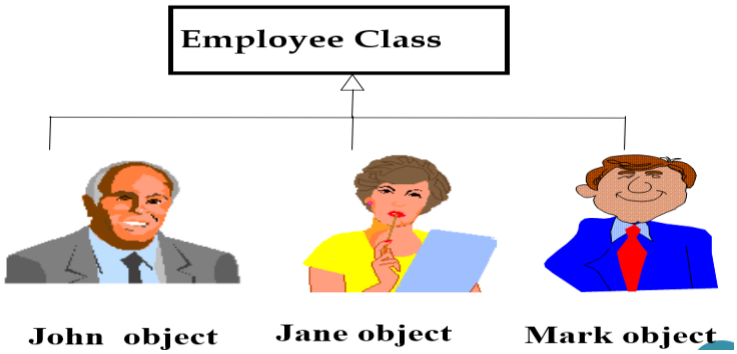
Examples of Object

- See if the followings are objects or not:
  - Desk
  - Person
  - Log
  - The Earth
  - Clock
  - Machine
  - Computer
  - Saving account
- Answer: N, Y, N, Y, Y, Y, Y, Y

10

Objects Are Grouped In Classes

- The role of a class is to define the attributes and methods (the state and behavior) of its instances
- We distinguish classes from instances.(e.g. eagle or airplane)
- The class car, for example, defines the property color
- Each individual car (object) will have a value for this property, such as "maroon," "yellow" or "white"



11

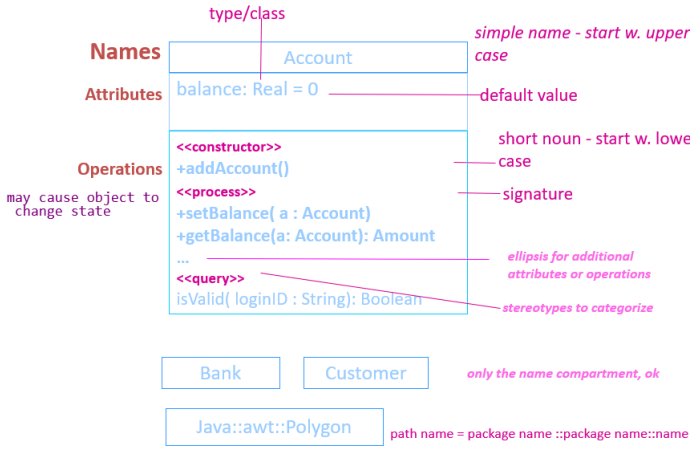
Objects Are Grouped In Classes

- Classes are the most important building block of any object-oriented system.
- The role of a class is to define the attributes and methods (the state and behavior) of its instances
- We distinguish classes from instances.(e.g. eagle or airplane)
- The class car, for example, defines the property color
- Each individual car (object) will have a value for this property, such as "maroon," "yellow" or "white“
- A class implements one or more interfaces.
- These classes may include abstractions that are part of the problem domain, as well as classes that make up an implementation.
- You can use classes to represent software things, hardware things, and even things that are purely conceptual.

12

Classes

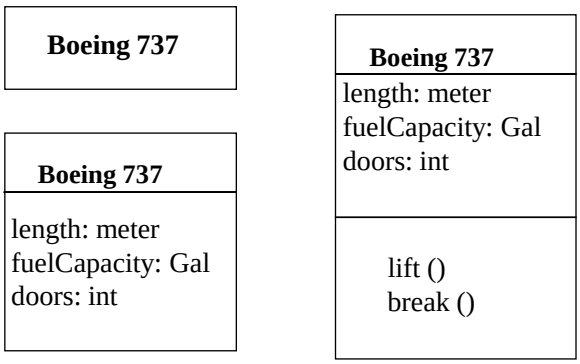
- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- Graphically, a class is rendered as a rectangle.



13

Classes

- In class notation, either one or both the attributes and operation compartments may be suppressed



14

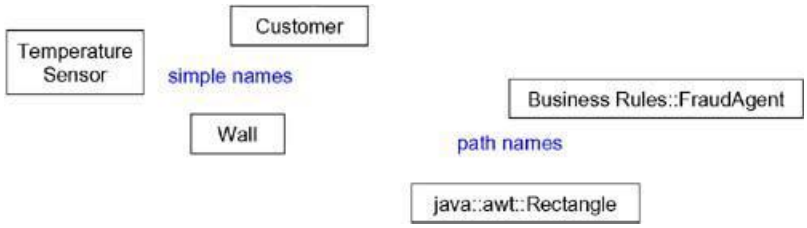
Classes

- **Names :**
  - A class name must be unique within its enclosing package.
  - Every class must have a name that distinguishes it from other classes. A name is a textual string. That name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives.
  - A class may be drawn showing only its name as well.
  - A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines.
  - In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling.

15

Classes

- **Names :**
  - Typically, you capitalize the first letter of every word in a class name, as in Customer or TemperatureSensor.

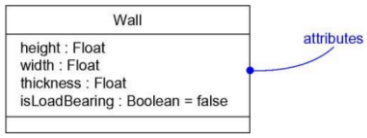


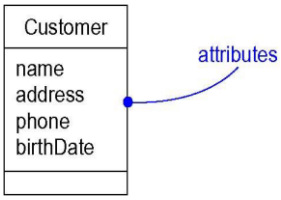
- **Attributes**:
  - An attribute is a named property of a class. A class may have any number of attributes or no attributes at all.
  - An attribute represents some property of the thing you are modeling that is shared by all objects of that class.

16

Classes

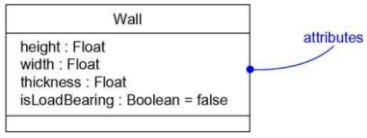
- **Attributes**:
  - For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth.
  - An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass.
  - Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names.

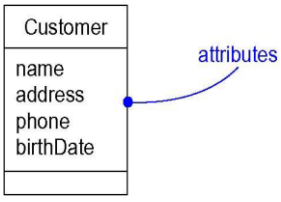




Classes

- **Attributes**:
  - An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class.
  - Typically, you capitalize the first letter
    - of every word in an attribute name
    - except the first letter, as in name or
    - loadBearing, birthDate.





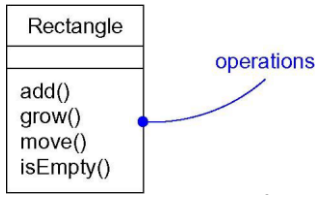
Classes

- **Operations**:
  - An operation is the implementation of a service that can be requested from any object of the class.
  - In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class.
  - A class may have any number of operations or no operations at all.
  - For example, in a windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties.
  - Often (but not always), invoking an operation on an object changes the object's data or state.

19

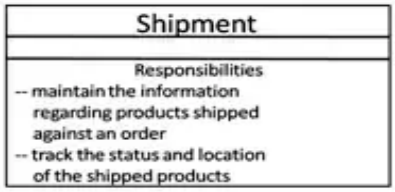
Classes

- **Operations**:
  - Graphically, operations are listed in a compartment just below the class attributes.
  - Operations may be drawn showing only their names.
  - **Note**: An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behaviour of its enclosing class.
  - Typically, you capitalize the first letter of every word in an operation name except the first letter, as in move or isEmpty.



Classes

- **Responsibilities**:
  - The responsibilities of a class specify the contract between the class and its objects. The responsibilities of a class represent its features (attributes and operations).
  - For example, the responsibility of a wall class is to represent the height, width and thickness of the wall. Responsibilities are represented as text in a compartment at the bottom of the class.

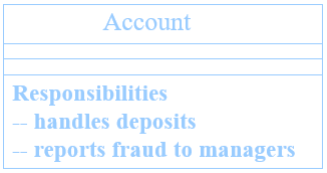


  - When you model a class, specify the things responsibilities.
  - You may use techniques like CRC cards and use-case based analysis.

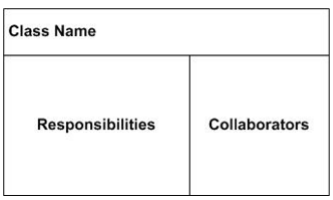
21

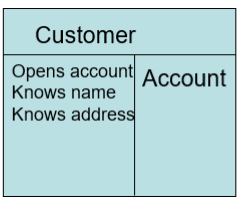
Classes

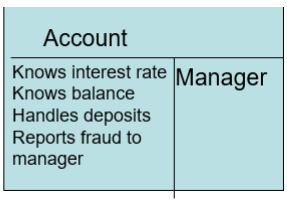
- **Responsibilities**:



  - You may use techniques like CRC cards and use-case based analysis.







22

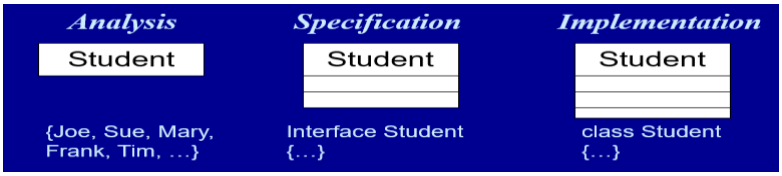
Classes

- Three common perspectives:
  - **Analysis** - description of the problem domain
  - **Specification** - logical description of software system
  - **Implementation** - description of software components and their deployment
- Meaning from three perspectives
  - Analysis: sets of objects
  - Specifications: interfaces to encapsulated software representations of objects
  - Implementations: abstract data types

23

Classes

- Three common perspectives:
  - **Analysis** - description of the problem domain
  - **Specification** - logical description of software system
  - **Implementation** - description of software components and their deployment
- Meaning from three perspectives
  - Analysis: sets of objects
  - Specifications: interfaces to encapsulated software representations of objects
  - Implementations: abstract data types

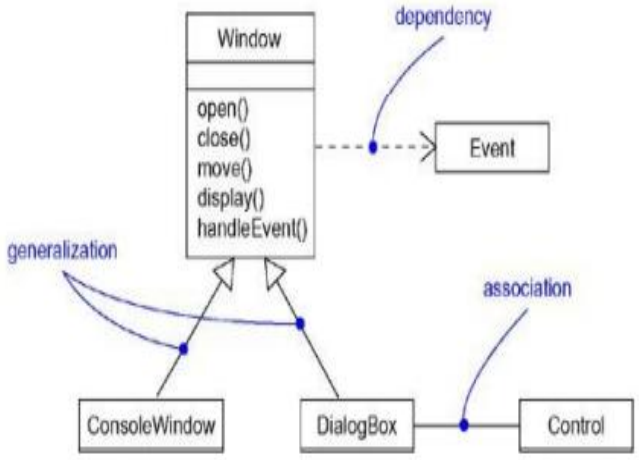


24

Relationships

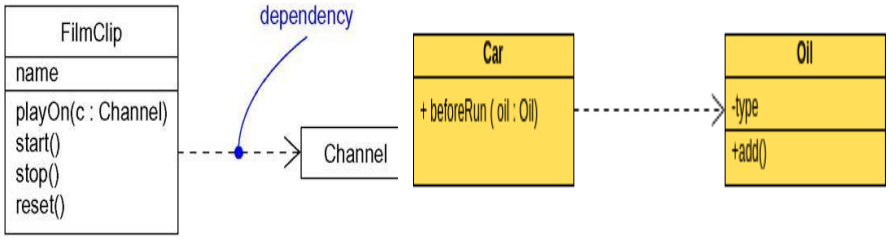
- When you build abstractions, you'll discover that very few of your classes stand alone.
- Instead, most of them collaborate with others in a number of ways.
- In object-oriented modeling, there are three kinds of relationships that are especially important:
  - Dependencies, which represent using relationships among classes (including refinement, trace, and bind relationships);
  - Generalizations, which link generalized classes to their specializations; and
  - Associations, which represent structural relationships among objects. Each of these relationships provides a different way of combining your abstractions.
- Other kinds of relationships are realization and refinement.

25



Relationships - Dependency

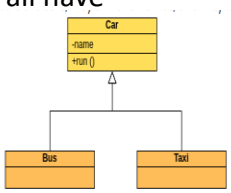
- Dependency : A dependency is a using relationship that states that a change in specification of one thing (for example, class Event) may affect another thing that uses it (for example, class Window), but not necessarily the reverse.
- The dependency relationship is “use” relationship
- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.



27

Relationships - Generalization

- Generalization/Inheritance : A generalization is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing (called the subclass or child).
- Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window).
- Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent.
- For example: buses, taxis, and cars are cars, they all have names, and they can all be on the road.



Relationships- Generalization

- A child inherits the properties of its parents, especially their attributes and operations.
- Often-but not always-the child has attributes and operations in addition to those found in its parents.
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent.
- A class may have zero, one, or more parents.
- A class that has no parents and one or more children is called a root class or a base class.
- A class that has no children is called a leaf class.

29

Relationships- Generalization

- A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

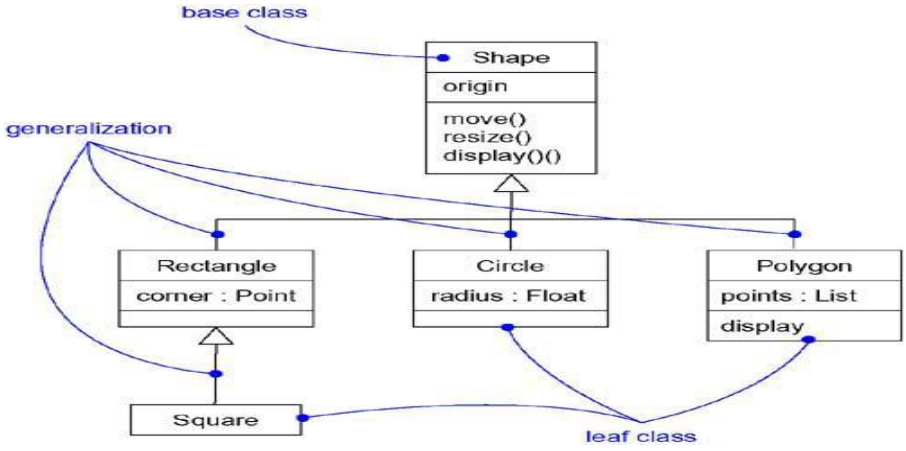
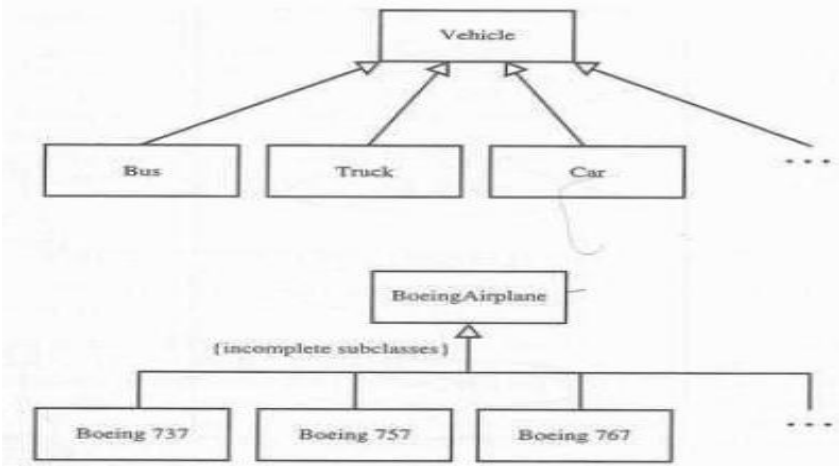


Figure: Generalization

30

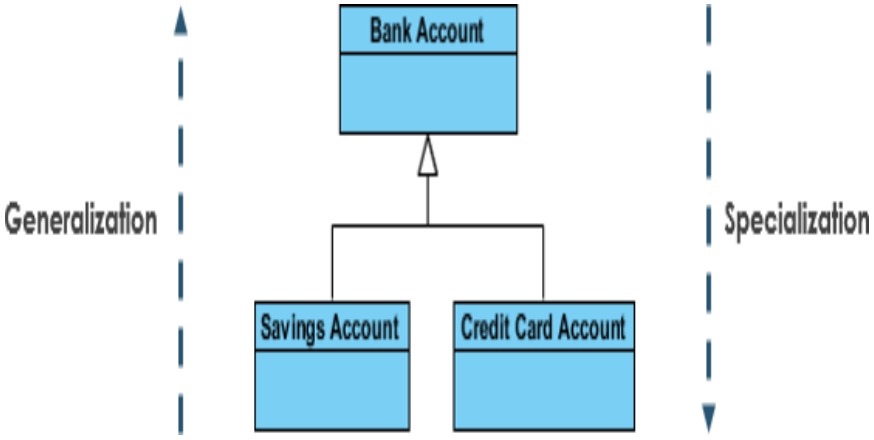
Relationships- Generalization

- Ellipses indicate that the generalization is incomplete and more subclasses exist that are not shown.
- If the text label is placed on hollow triangle, it applies to all the paths.



Relationships - Specialization

- Specialization is the reverse process of Generalization means creating new sub-classes from an existing class.



32

Relationships- Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.
- Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa.
- It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class.
- An association that connects exactly two classes is called a binary association.
- Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations.

33

Relationships- Association

- Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships.
- An association can have a name, and you use that name to describe the nature of the relationship.
- So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name employs

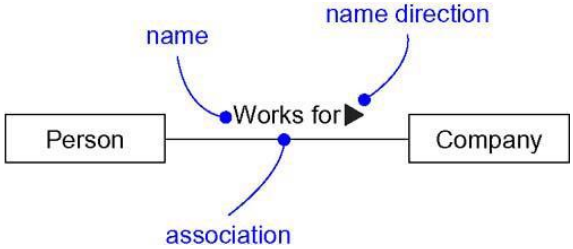


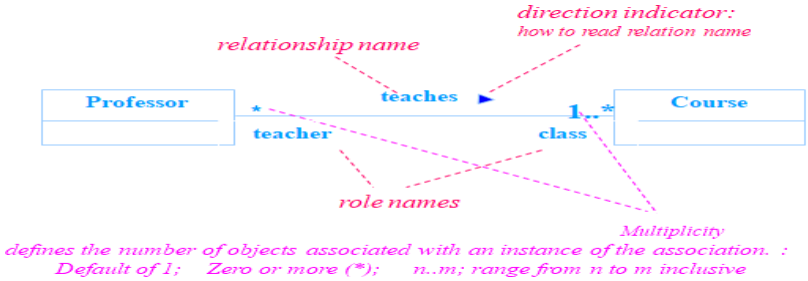
Figure: Association Names

34

Relationships- Association

- UML uses term association navigation or navigability to specify role.

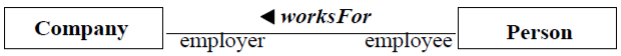
- Person class can't know anything about BankAccount class, but BankAccount class can know about Person class

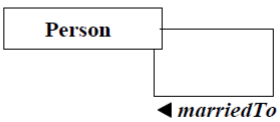


35

Relationships- Association (Binary)

- A binary association is drawn as a solid path connecting two classes or both ends may be connected to the same class.

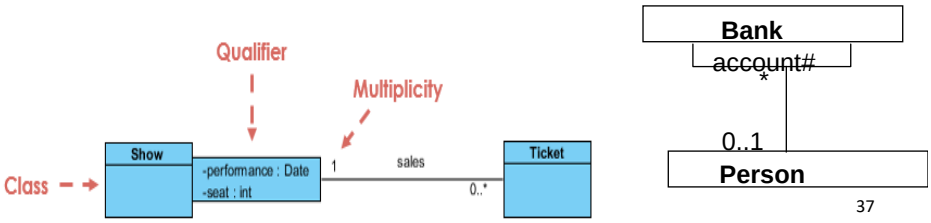




36

Relationships- Association

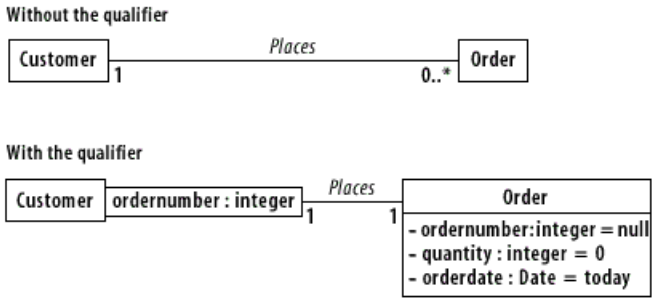
- Role : The end of an association, where it connects to a class, shows the association role
  - Role is a part of association, not class
  - Each association has two or more roles to which is connected
- Qualifier : A qualifier is an association attribute. For example, a person object may be associated to a Bank object
  - An attribute of this association is the account#
  - The account# is the qualifier of this association



37

Relationships- Association

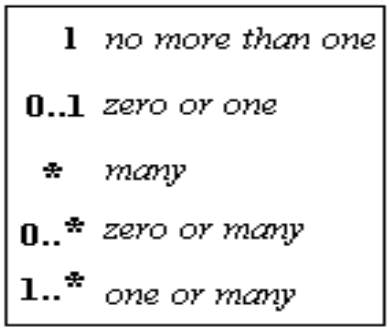
- Qualifier :

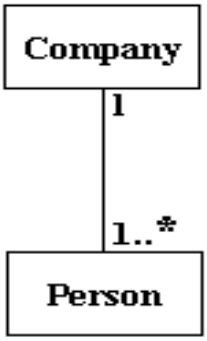


38

Relationships- Association

- Multiplicity : Multiplicity specifies the range of allowable associated classes.
- It is given for roles within associations, parts within compositions, repetitions, and other purposes.
- lower bound .. upper bound

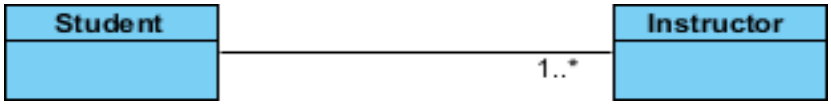




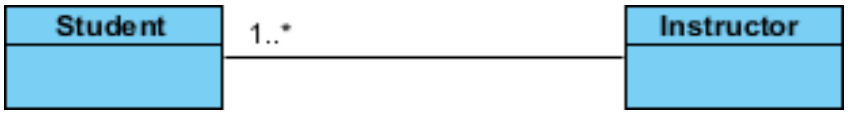
39

Relationships- Association

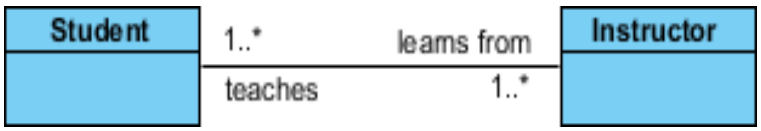
- A single student can associate with multiple teachers:



- The example indicates that every Instructor has one or more Students:



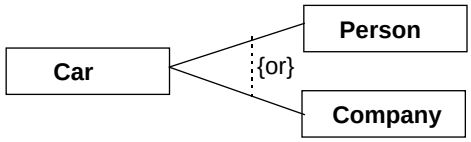
- We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



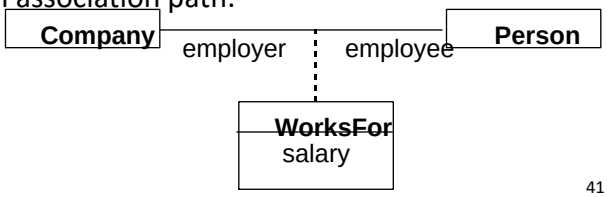
40

Relationships- Association

- An OR association indicates a situation in which only one of several potential associations may be substantiated at one time for any single object.



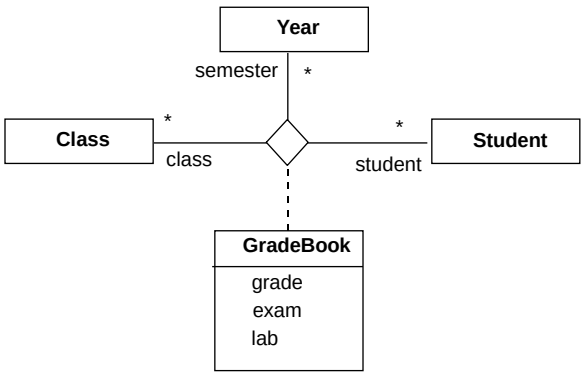
- An association class is an association that also has class properties.
- An association class is shown as a class symbol attached by a dashed line to an association path.



41

Relationships- Association

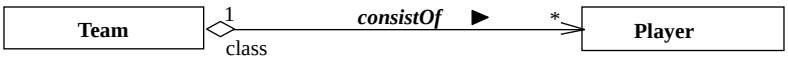
- An n-ary association is an association among more than two classes.
- Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association



42

Relationships- Aggregation

- Aggregation is a form of association
- A hollow diamond is attached to the end of the path to indicate aggregation
- Aggregation is used to represent ownership or a whole/part relationship

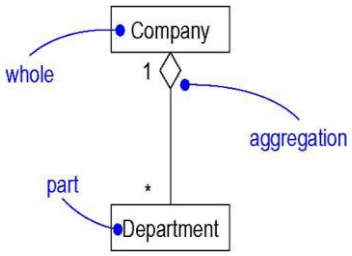


- Aggregation is a special form of association that represents an ownership relationship between two objects.
- Aggregation models has-a relationships.
- The owner object is called an aggregating object, and its class is called an aggregating class. The subject object is called an aggregated object, and its class is called an aggregated class.

43

Relationships- Association

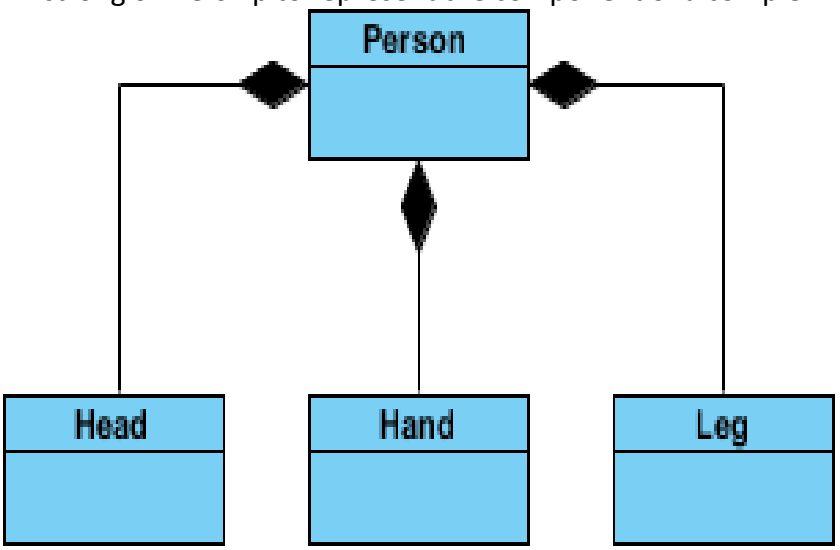
- Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").



44

Relationships- Composition

- Also known as the *a-part-of*, is a form of aggregation with strong ownership to represent the component of a complex



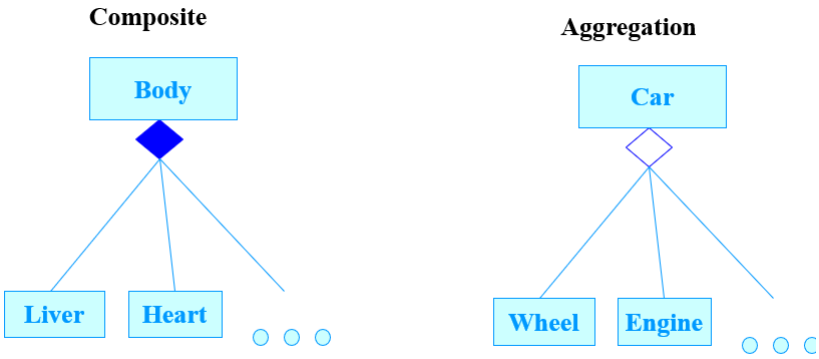
45

Relationships- Composition

- The UML notation for composition is a solid diamond at the end of a path , and is used to represent an even stronger form of ownership.
- The composite object has sole responsibility for the disposition of its parts in terms of creation and destruction
- In implementation terms, the composite is responsible for memory allocation and deallocation
- The multiplicity of the aggregate end may not exceed one; i.e., it is unshared. An object may be part of only one composite at a time
- If the composite is destroyed, it must either destroy all its parts or else give responsibility for them to some other object
- A composite object can be designed with the knowledge that no other object will destroy its parts

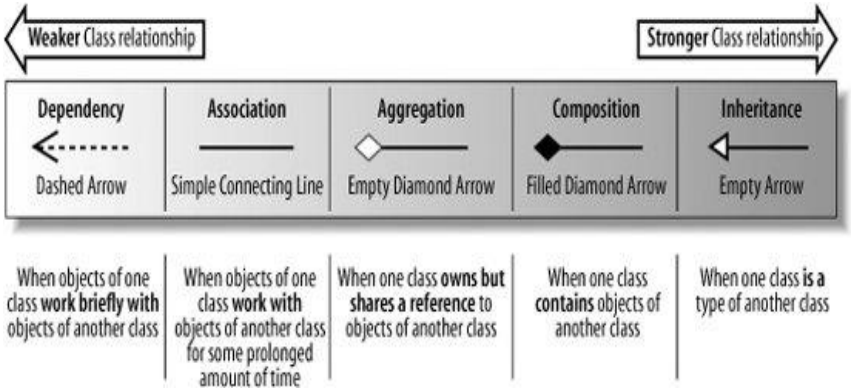
46

Relationships



Composite is a stronger form of aggregation.
Composite parts live and die with the whole.

47



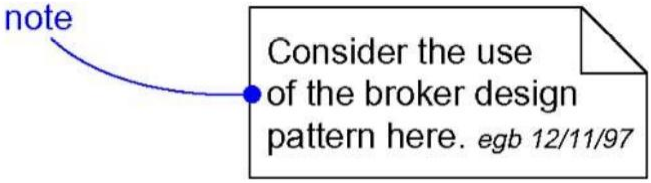
Common Mechanism

- Some common mechanisms used in UML are:
  - Notes
  - Stereotypes, tagged values, and constraints
  - Modeling comments
  - Modeling new building blocks
  - Modeling new properties
  - Modeling new semantics
  - Extending the UML
- The UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language: **specifications**, **adornments**, **common divisions**, and **extensibility mechanisms**.

49

Common Mechanism

- **Notes** are the most important kind of ***adornment*** that stands alone.
- A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements.
- You use notes to attach information to a model, such as requirements, observations, reviews, and explanations.



- Notes may represent artifacts that play an important role in the software development life cycle, such as requirements, or they may simply represent free-form observations, reviews, or explanations.

50

Common Mechanism

- The UML's *extensibility mechanisms* permit you to extend the language in controlled ways. These mechanisms include stereotypes, tagged values, and constraints.
- A **stereotype** extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem.
  - Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element.
  - As an option, the stereotyped element may be rendered by using a new icon associated with that stereotype.

51

Common Mechanism

- A **tagged value** extends the properties of a UML building block, allowing you to create new information in that element's specification.
  - Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- A **constraint** extends the semantics of a UML building block, allowing you to add new rules or modify existing ones.
  - Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.
  - As an alternative, you can render a constraint in a note.

52

Common Mechanism

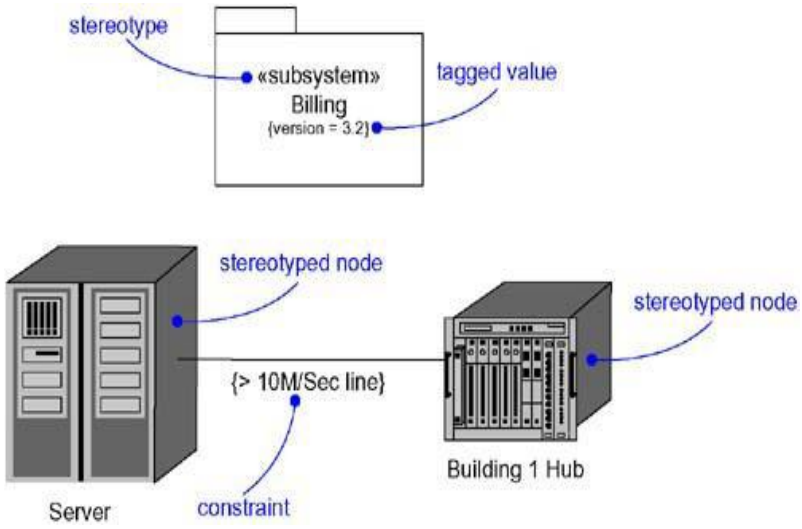
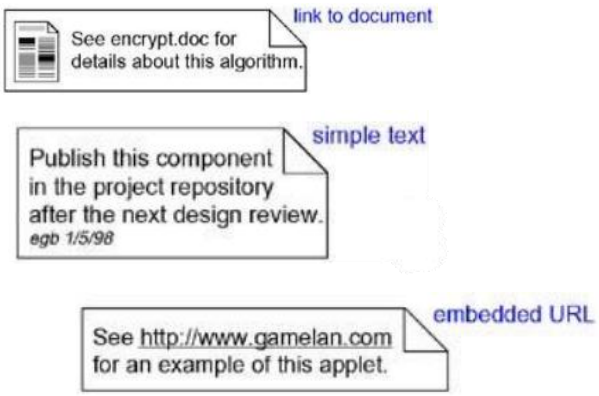


Figure: Stereotypes, Tagged Values, and Constraints

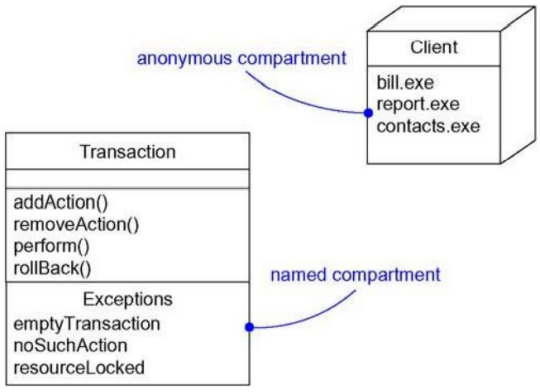
53

Common Mechanism



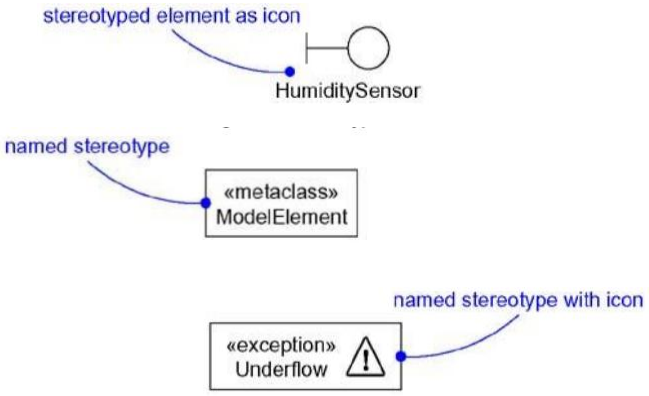
54

Common Mechanism - Adornments



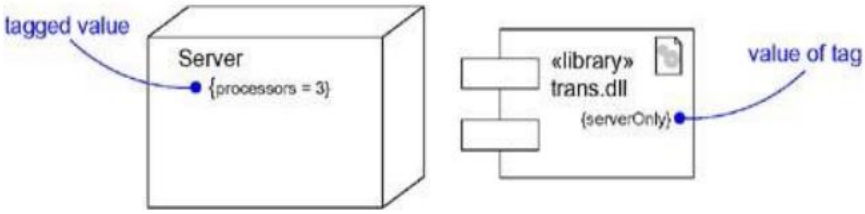
55

Common Mechanism -



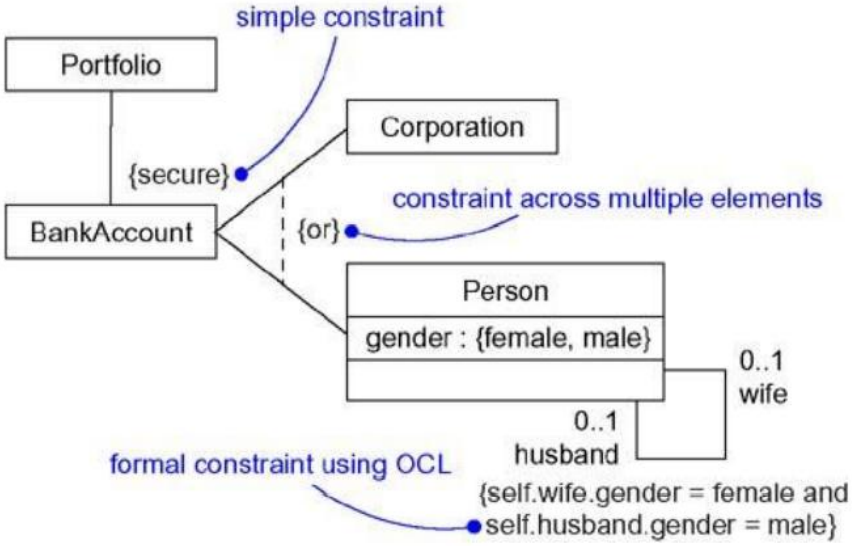
56

Common Mechanism -



57

Common Mechanism - Constraint



58

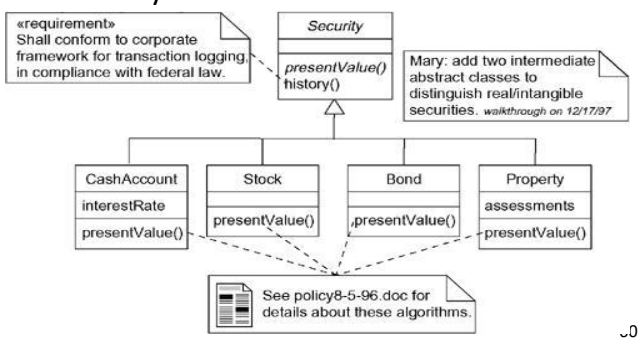
Common Mechanism

- The most common purpose of using notes is to write down free-form observations, reviews, or explanations.
- By putting these comments directly in your models, your models can become a common repository.
- You can even use notes to visualize requirements and show how they tie explicitly to the parts of your model.
- To model a comment,
  - Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
  - You don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.

59

Common Mechanism

- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and - unless they are of historic interest - discard the others.



60

Modeling New Building Blocks

- The UML's building blocks- classes, interfaces, collaborations, components, nodes, associations, and so on- are generic enough to address most of the things you'll want to model.
- If you want to extend your modeling vocabulary or give distinctive visual cues to certain kinds of abstractions that often appear in your domain, you need to use stereotypes.
- To model new building blocks,
  - Make sure there's not already a way to express what you want by using basic UML as chances are there's already some standard stereotype that will do what you want.
  - If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing.

61

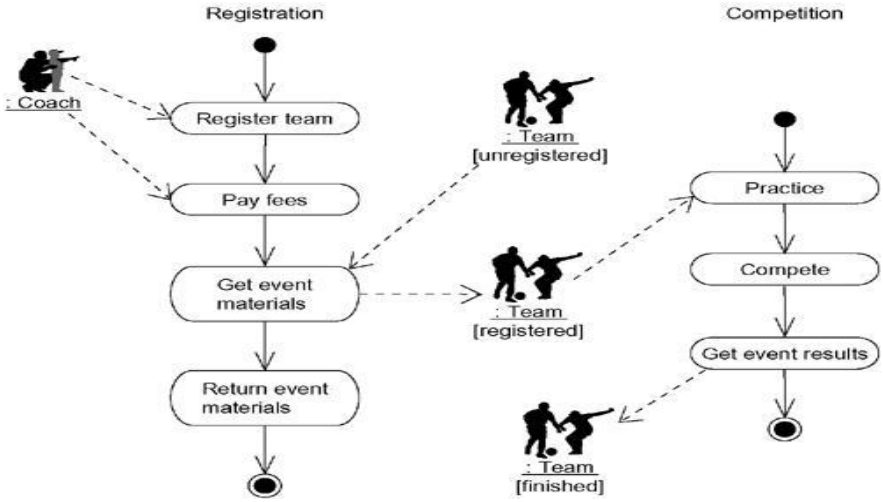
Modeling New Building Blocks

  - Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
  - If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.

62

Modeling New Building Blocks

- For example, suppose you are using activity diagrams to model a business process involving the flow of coaches and teams through a sporting event



Modeling New Building Blocks

- In this context, it would make sense to visually distinguish coaches and teams from one another and from the other things in this domain, such as events and divisions.
- In example there are two things that stand out - Coach objects and Team objects.
- These are not just plain kinds of classes.
- Rather, they are now primitive building blocks that you can use in this context. You can create these new building blocks by defining a coach and team stereotype and applying them to UML's classes.
- In this figure, the anonymous instances called :Coach and :Team (the latter shown in various states - namely, unregistered, registered, and finished) appear using the icons associated with these stereotypes.

64

Modeling New Properties

- However, if you want to extend the properties of these basic building blocks (or the new building blocks you create using stereotypes), you need to use tagged values.
- To model new properties,
  - First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
  - If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply- tagged values defined for one kind of element apply to its children.
  - For example, suppose you want to tie the models you create to your project's configuration management system.

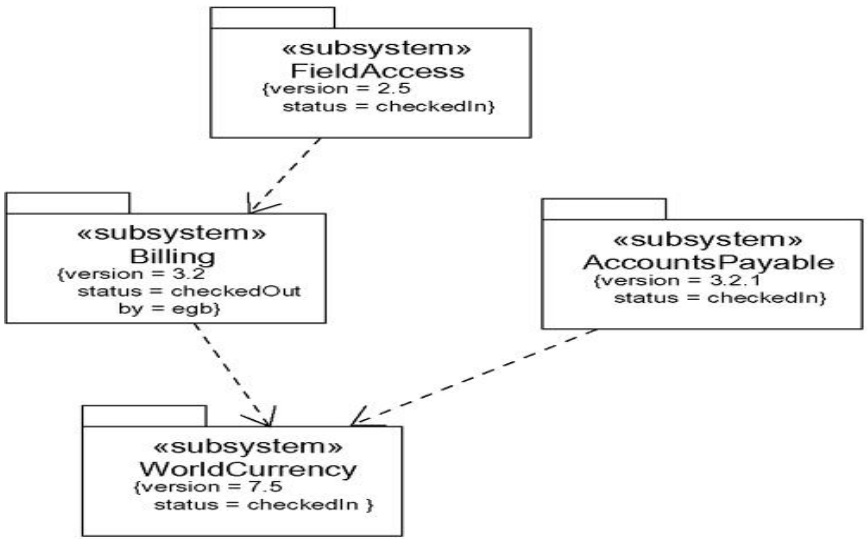
65

Modeling New Properties

- It means you also need to keep track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem.
- Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values.
- Furthermore, this information is not just a class attribute either.
- A subsystem's version number is part of its metadata, not part of the model.

66

Modeling New Properties



67

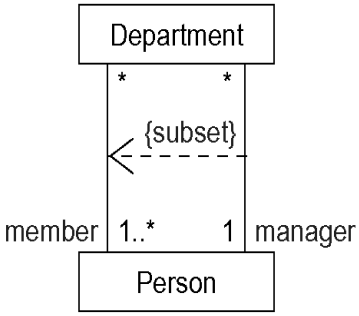
Modeling New Semantics

- If you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.
- To model new semantics,
  - First, make sure there's not already a way to express what you want by using basic UML.
  - If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.
  - If there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers.
  - You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.

68

Modeling New Semantics

- This diagram shows that each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member.
- This diagram goes on to indicate that each Department must have exactly one Person as a manager and every Person may be the manager of zero or more Departments.
- All of these semantics can be expressed using simple UML.



Types of diagram

- Covered in first chapter.
- You'll view the static parts of a system using one of the four following diagrams.
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram
- Five additional diagrams to view the dynamic parts of a system.
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

70

Class Diagram

- Class diagrams are the most common diagram found in modeling object- oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- You use class diagrams to model the static design view of a system.
- For the most part, this involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas.
- Class diagrams are also the foundation for a couple of related diagrams: component diagrams and deployment diagrams.
- Apart from visualizing, specifying, and documenting structural models, class diagrams are also used for constructing executable systems through forward and reverse `engineering.`

71

Class Diagram

- Class-based elements: Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system.
- These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.
- For example, a UML class diagram can be used to depict
- The **purpose** of class modeling is to describe **objects in systems** and **different types of relationships between them.**
- Class diagrams represent an overview of the system like classes, attributes, operations, and relationships.

72

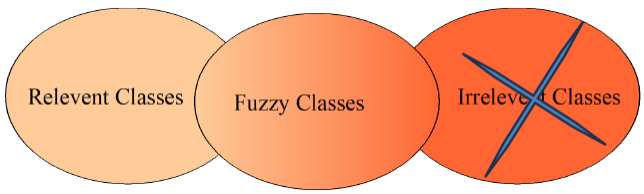
Approaches for identifying Class

- Following are the alternative approaches to identifying classes
  - The noun phrase approach
  - The common class patterns approach
  - The use-case driven approach
  - The class responsibilities collaboration (CRC) approach

73

Noun Phrase Approach

- Using this method, you have to read through the Use cases, interviews, and requirements specification carefully, looking for noun phrases
- Nouns in the textual description are classes and verbs are methods of the classes.
- Change all plurals to singular and make a list, which can then be divided into three categories



74

Guidelines For Identifying Classes

- Look for nouns and noun phrases in the problem statement
- Some classes are implicit or taken from general knowledge
  - E.g if library system : Book, member etc.
  - If Bank system : customer, Bank etc.
- A noun means a content word that can be used to refer to a person, place, thing, quality, or action
- All classes must make sense in the application domain.
- **Redundant Classes:**
  - Do not keep two classes that express the same information.
  - If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system.
  - While choosing vocabulary always select the word that is used by the user of the system.

75

Guidelines For Identifying Classes

- **Adjective Classes:**
  - An adjective is a word that expresses an attribute of something or the word class that qualifies nouns.
  - Adjectives can be used in many ways. It modifies a noun. It describes quality, state or action that noun refers to.
  - Adjective comes before nouns e.g. a new car
  - Size : big, tiny
  - Age : new, young, old
  - Shape : square, color, material etc.
  - Adjective can come after verbs e.g. your friend looks nice, dinner smells good
  - Adjective can suggest different kind of object, different use of same object or can be irrelevant.

76

Guidelines For Identifying Classes

- **Adjective Classes:**
  - Does the object represented by the noun behave differently when the adjective is applied to it?
  - N.B. An adjective is a word that expresses an attribute of something or the word class that qualifies nouns
  - If the use of the adjective signals that the behavior of the object is different, then make a new class
  - For example, If Adult Membership and Youth Membership behave differently, than they should be classified as different classes
- **Attribute Classes:**
  - Tentative objects which are used only as values should be defined or restated as attributes and not as a class
  - For example the demographics of Membership are not classes but attributes of the Membership class
  - E.g. client status is attribute of client class.

77

Guidelines For Identifying Classes

- **Adjective Classes:**
  - Does the object represented by the noun behave differently when the adjective is applied to it?
  - N.B. An adjective is a word that expresses an attribute of something or the word class that qualifies nouns
  - If the use of the adjective signals that the behavior of the object is different, then make a new class
  - For example, If Adult Membership and Youth Membership behave differently, than they should be classified as different classes
- **Attribute Classes:**
  - Tentative objects which are used only as values should be defined or restated as attributes and not as a class
  - For example the demographics of Membership are not classes but attributes of the Membership class
  - E.g. client status is attribute of client class.

78

Guidelines For Identifying Classes-Noun Phrase

1. **Describe the software product in paragraph.**
2. **Identifying Tentative classes by identify the noun (exclude those that lie outside the problem boundary).**
3. **Selecting classes from the Relevant and fuzzy categories.**
4. **Initial List of Noun phrases: candidate classes.**
5. **Reviewing the Redundant classes and Building a common vocabulary.**
6. **Reviewing the classes containing Adjectives.**
7. **Reviewing the possible Attributes.**
8. **Reviewing the class purpose.**

79

Case Study-ViaNet Bank ATM

- **Description of the ViaNet bank ATM system's requirements**
- The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the touch screen at the ViaNet bank ATM kiosk. Each transaction must be recorded, and the client must be able to review all transactions performed against a given account. Recorded transactions must include the date, time transaction type, amount and account balance after the transaction.
- A ViaNet bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist.
- Access to the ViaNet bank accounts is provided by a PIN code consisting of four integer digits between 0 and 9.
- One PIN code allows access to all accounts held by a bank client.
- No receipts will be provided for any account transactions.

80

Case Study-ViaNet Bank ATM

- The bank application operates for a single banking institution only.
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

81

ViaNet Bank ATM- Identifying classes-Noun Phrase Approach

- **Candidate classes**
- Account
- Account balance
- Amount
- Approval process
- ATM card
- ATM machine
- Bank
- Bank client
- Card
- Cash
- Check
- Current
- Current account
- Client
- Client's account
- Currency
- Envelope
- Four digits
- Fund
- Invalid PIN
- Message
- Money
- Password
- PIN
- PIN code
- Record
- Savings
- Savings account
- Step
- System
- Transaction
- Transaction history

82

ViaNet Bank ATM- Identifying classes-Noun Phrase Approach

- Remove **irrelevant** classes as they do not belong to problem statement.
  - Envelope
  - four digits
  - step
- **Redundant Classes**
- Client, bank client – Bank client
- Account, client's account – Account
- PIN, PIN code – PIN
- Current, Current account – Current account
- Savings, savings account - Savings account
- Fund, Money – Fund
- ATM card, Card – ATM card

83

After eliminating redundant class

- Account
- Account balance
- Amount
- Approval process
- ATM card
- ATM machine
- Bank
- Bank client
- Cash
- Check
- Current account
- Currency
- Fund
- Invalid PIN
- Message
- Password
- PIN
- Record
- Savings Account
- System
- Transaction
- Transaction history

84

Reviewing classes containing adjectives

- Does the object represented by the noun behave differently when the adjective is applied to it?
- If yes, make a new class
- Else class is irrelevant, we must eliminate it
- We have no classes containing adjectives that we can eliminate.

85

Reviewing the Possible Attributes

- Identifying the noun phrases that are attributes, not classes.
- **Amount** : a value, not a class
- **Account Balance** : An attribute of Account class.
- **Invalid PIN** : It is only a value, not a class.
- **Password** : An attribute of BankClient class.
- **Transaction history** : An attribute of Transaction class
- **PIN** : An attribute possibly of BankClient class.

86

Candidate class – Final

- Account
- Approval process
- ATM card
- ATM machine
- Bank
- Bank client
- Cash
- Check
- Current account
- Currency
- Fund
- Message
- Record
- Savings Account
- System
- Transaction

87

Reviewing the class purpose

- ATM Machine class: Provides an interface to the ViaNet bank.
- ATM Card class: Provides a client with key to an account.
- BankClient class: A client is an individual that has checking account, and possibly, a savings account.
- Bank class: Bank clients belong to the Bank. It is repository of accounts and processes the accounts transactions.
- Account class: An account class is a formal (or abstract) class , it defines the common behaviors that can be inherited by more specific classes such as Checking account and Savings Account.
- CheckingAccount class: It models a client's checking account and provides more specialized withdrawal service.
- SavingsAccount class: It models a client's savings account.
- Transaction class: Keeps track of transaction, time, date, type, amount, and balance.

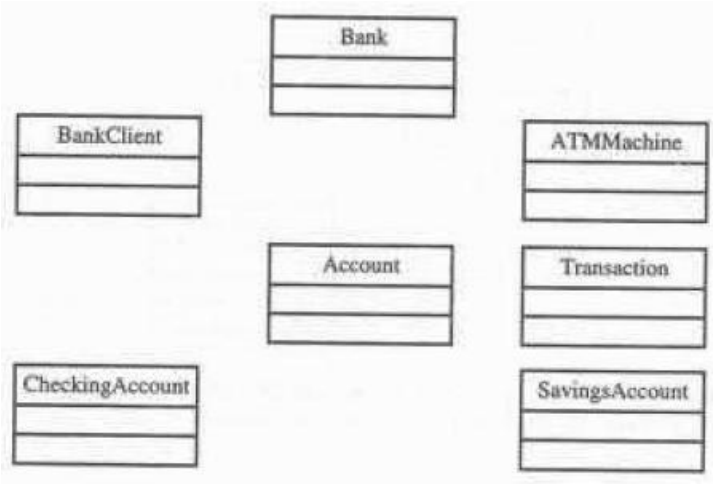
88

How to draw class diagram

- **Step 1: Determine the class names.** The first step is to determine the system's primary objects.
- **Step 2: Separate relationships** The next step is to figure out how each of the classes or objects is connected to the others. Look for commonalities and abstractions between them to aid in grouping them while building the class diagram.
- **Step 3: Build the Structure** Add the class names first, then connect them with the required connectors. Attributes and functions/methods/operations can be added afterward

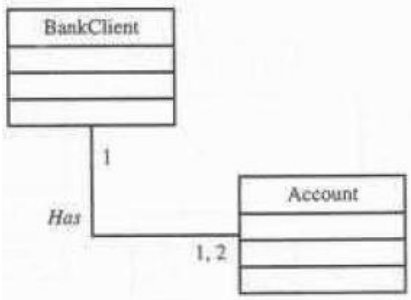
89

Class diagram for ViaNet ATM banking system



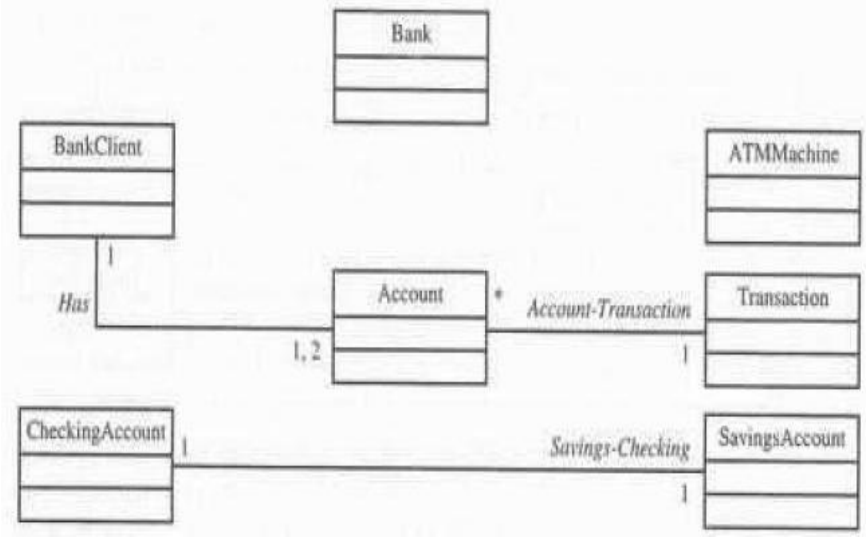
90

Relationship



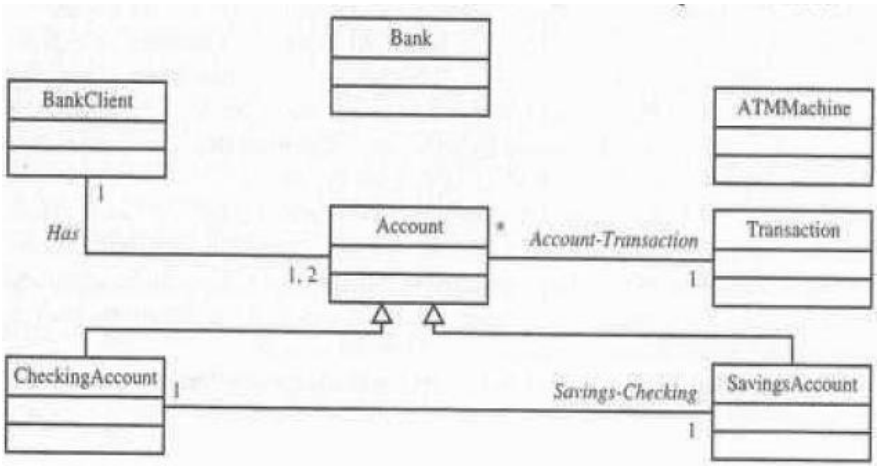
91

Associations



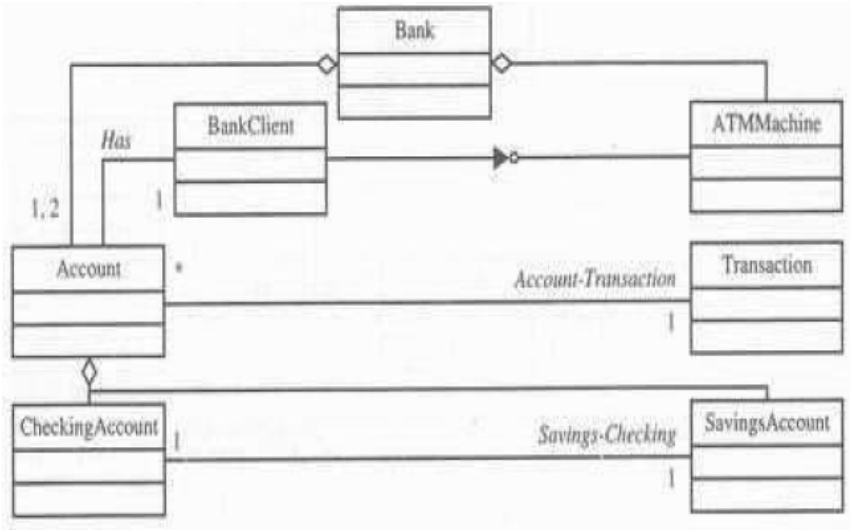
92

Super-sub relationships



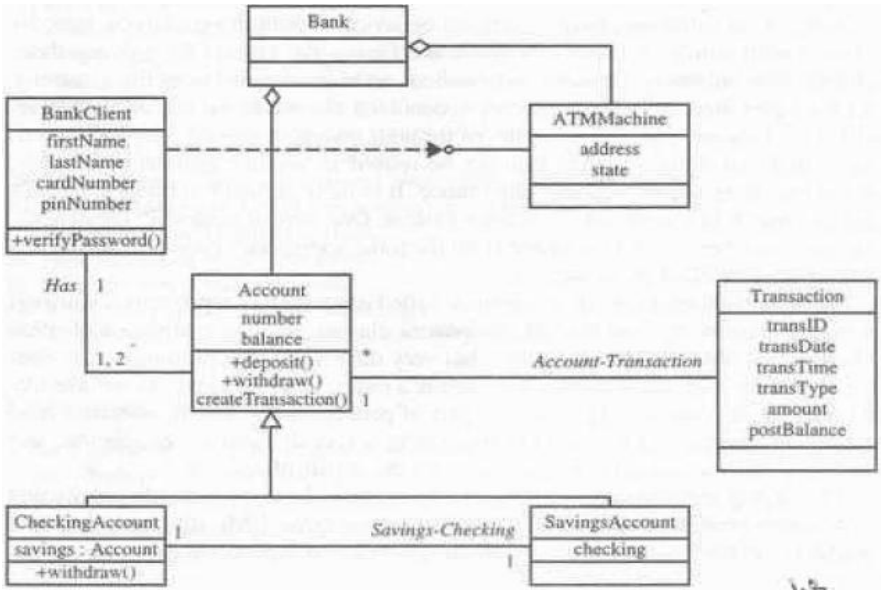
93

Association, generalization, aggregation and interface



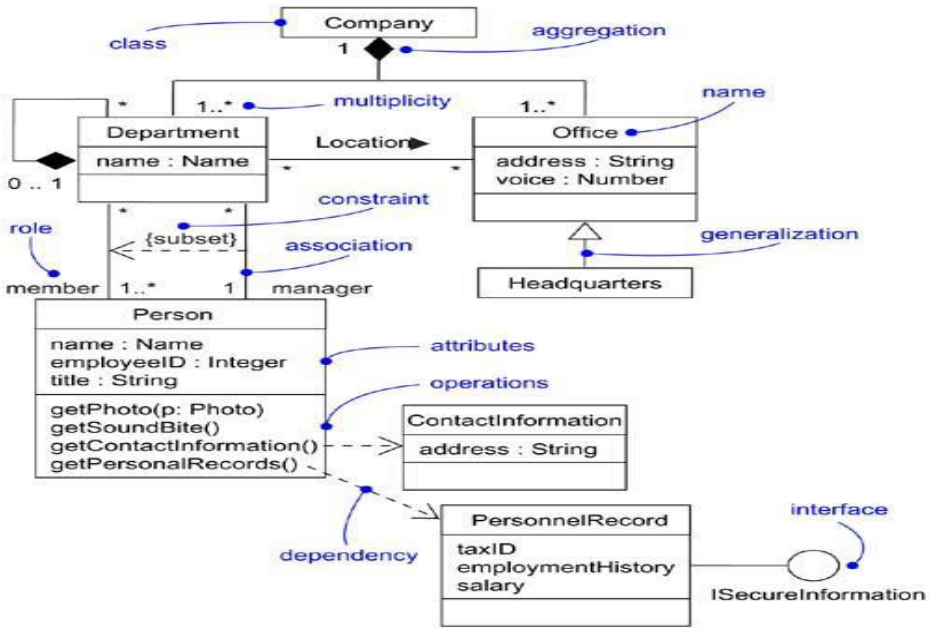
94

ATM- with attributes and operations



95

Class Diagram

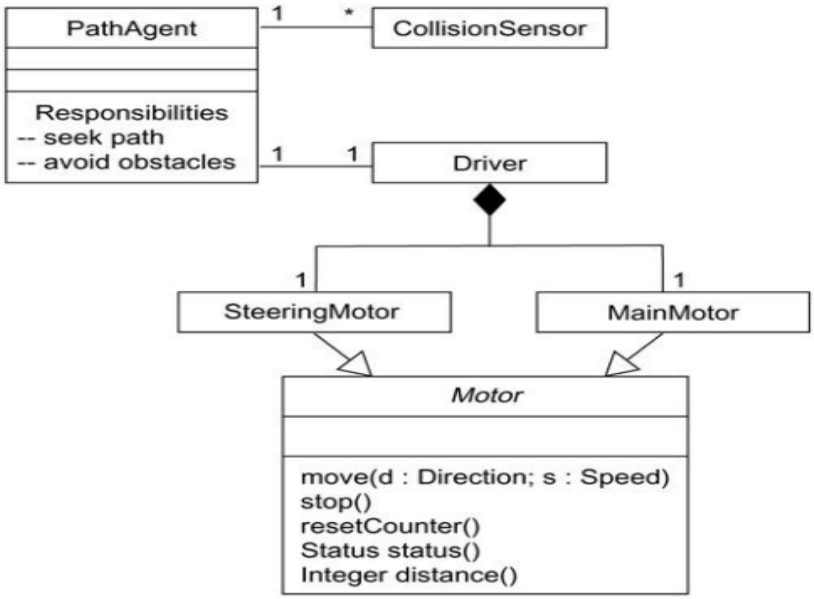


Modeling Simple Collaborations

- To model simple collaborations,
  1. Identify the function or behavior of the part of a system you would like to model.
  2. For each function or mechanism identify the classes, interfaces, collaborations and relationships between them.
  3. Use scenarios (sequence of activities) to walk through these things. You may find new things or find that existing things are semantically wrong.
  4. Populate the things found in the above steps. For example, take a class and fill its responsibilities. Now, convert these responsibilities into attributes and operations.

97

Modeling Simple Collaborations



Modeling Simple Collaborations

- Figure shows a set of classes drawn from the implementation of an autonomous robot.
- The figure focuses on the classes involved in the mechanism for moving the robot along a path.
- You'll find one abstract class (*Motor*) with two concrete children, *SteeringMotor* and *MainMotor*. Both of these classes inherit the five operations of their parent, Motor.
- The two classes are, in turn, shown as parts of another class, *Driver*.
- The class *PathAgent* has a one-to-one association to Driver and a one-to-many association to *CollisionSensor*. No attributes or operations are shown for PathAgent, although its responsibilities are given.
- There are many more classes involved in this system, but this diagram focuses only on those abstractions that are directly involved in moving the robot.

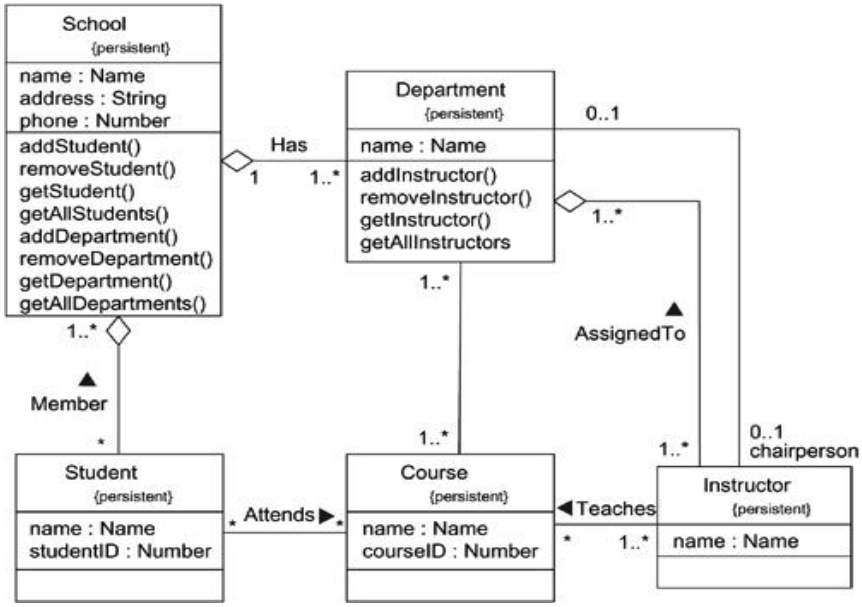
99

Modeling Logical Database Schema

- To model a schema,
  1. Identify the classes whose state must be saved over the lifetime of the application.
  2. Create a class diagram and mark these classes as persistent by using tagged values.
  3. Provide the structural details for these classes like the attributes, associations with other classes and the multiplicity.
  4. Minimize the common patterns which complicate the physical database design like cyclic associations, one-to-one associations and n-ary associations.
  5. Provide the behavior for these classes by listing out the operations that are important for data access and integrity.
  6. Wherever possible, use tools to convert the logical design to physical design.

100

Modeling Logical Database Schema



Modeling Logical Database Schema

- Figure shows a set of classes drawn from an information system for a school.
- This figure expands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database.
- Starting at the bottom-left of this diagram, you will find the classes named *Student*, *Course*, and *Instructor*.
- There's an association between *Student* and *Course*, specifying that students attend courses.
- Every student may attend any number of courses and every course may have any number of students.
- All six of these classes are marked as persistent, indicating that their instances are intended to live in a database or some other form of persistent store.

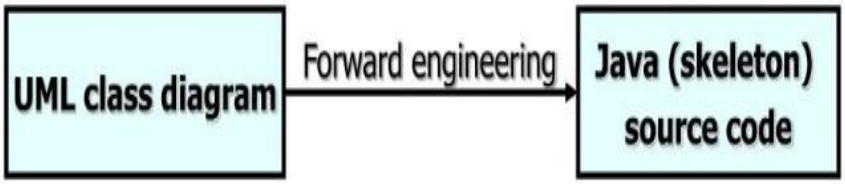
Modeling Logical Database Schema

- This diagram also exposes the attributes of all six of these classes. Notice that all the attributes are primitive types.
- When you are modeling a schema, you'll generally want to model the relationship to any nonprimitive types using an explicit aggregation rather than an attribute.
- Two of these classes (*School* and *Department*) expose several operations for manipulating their parts.
- These operations are included because they are important to maintain data integrity (adding or removing a *Department*, for example, will have some rippling effects).
- There are many other operations that you might consider for these and the other classes, such as querying the prerequisites of a course before assigning a student.
- These are more business rules than they are operations for database integrity and so are best placed at a higher level of abstraction than this schema.

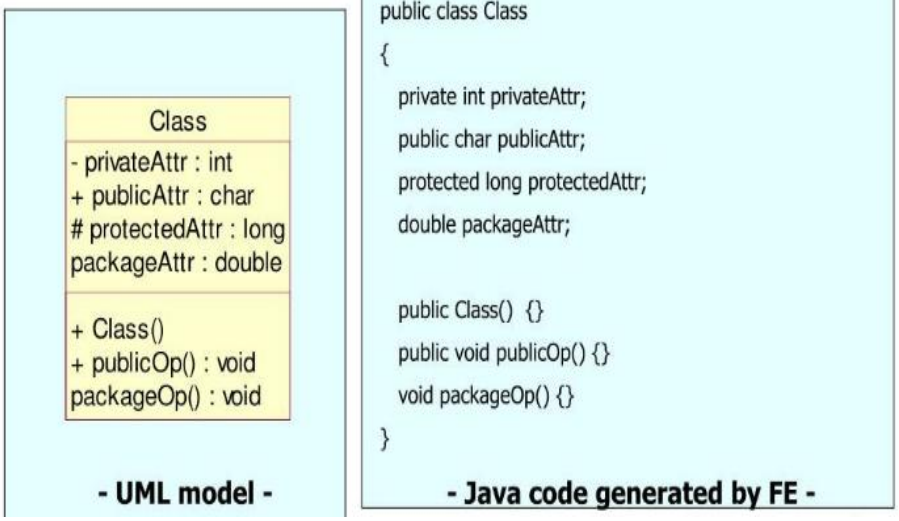
103

Forward and Reverse Engineering

- To forward engineer a class diagram,
  1. Identify the rules for mapping the models to the implementation language of your choice.
  2. Depending on the semantics of the language, you may want to restrict the information in your UML models. For example, UML supports multiple inheritance. But some programming languages might not allow this.
  3. Use tagged values to specify the target language.
  4. Use tools to convert your models to code.

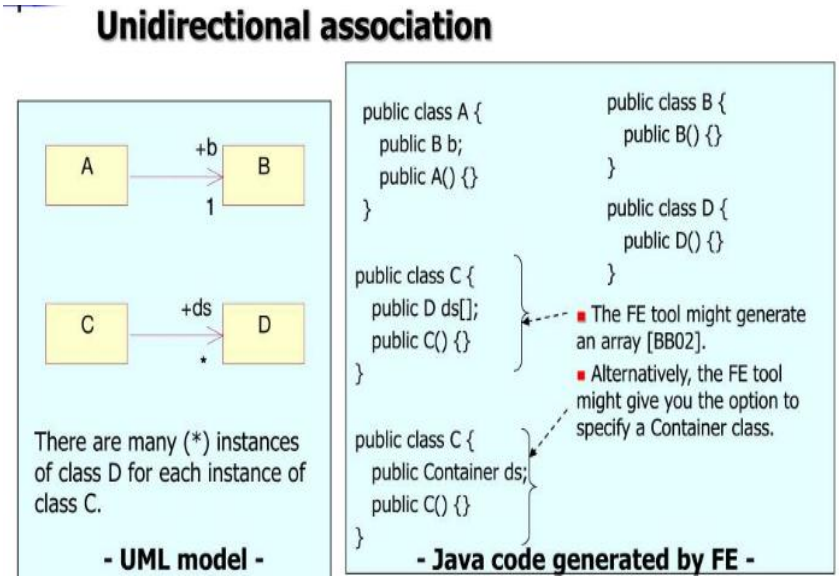


Forward Engineering –UML to Java



105

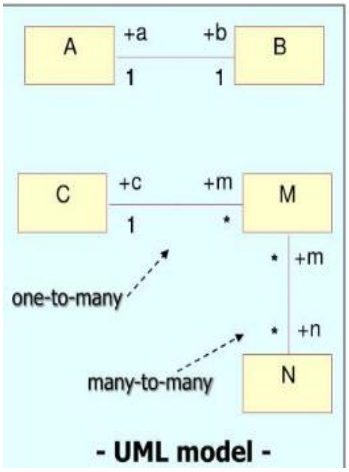
Forward Engineering –UML to Java



106

Forward Engineering –UML to Java

Bi-directional association



- UML model -

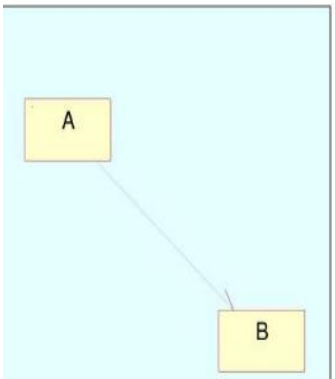
```
public class A {
  public B b;
  public A() {}
}

public class B {
  public A a;
  public B() {}
}

public class C {
  public M m[];
  public C() {}
}

public class M {
  public C c;
  public N n[];
  public M() {}
}
public class N {
  public M m[];
  public N() {}
}
```

- Java code generated by FE -

107

Forward Engineering –UML to Java

Dependency



- UML model -

```
public class A {
  public A() {}
}

public class B {
  public B() {}
}
```

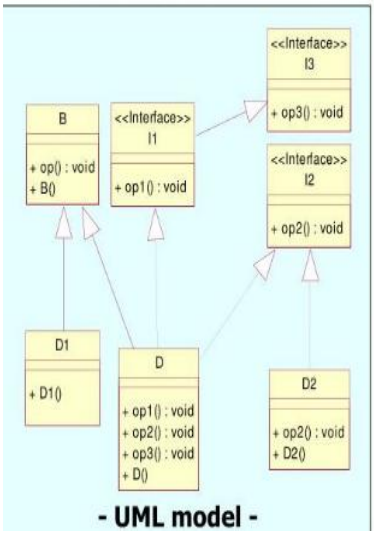
- The dependency is the most general relationship in UML diagrams.
- In this case the FE tool does not know what kind of dependency is assumed. Therefore, it generates no support for the relationship (this is the approach considered in [BB02]).

- Java code generated by FE -

108

Forward Engineering –UML to Java

Generalization and realization



- UML model -

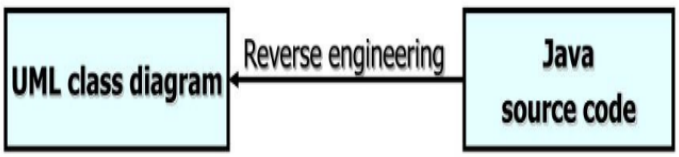
```
public interface I1 extends I3 {
  public void op1();
}
public class B {
  public B() {}
  public void op() {}
}
public class D1 extends B {
  public D1() {}
}
public class D extends B implements I1, I2 {
  public D() {}
  public void op1() {}
  public void op2() {}
  public void op3() {}
}
public interface I3 {
  public void op3();
}
public interface I2 {
  public void op2();
}
public class D2 implements I2 {
  public D2() {}
  public void op2() {}
}
```

- Java code generated by FE -

109

Forward and Reverse Engineering

- To reverse engineer the code,
  1. Identify the rules for mapping the implementation language to a model.
  2. Using a tool, navigate to the code that you want to reverse engineer. Use the tool to generate a new model.



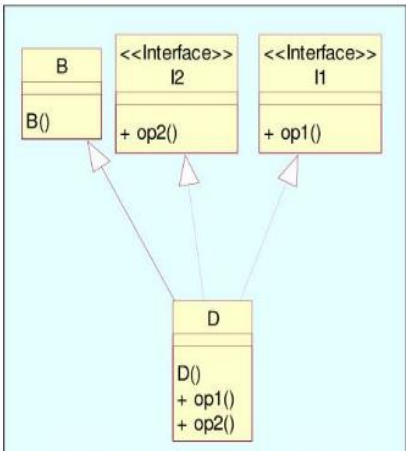
110

Reverse Engineering Java to UML

Generalization and realization

```
class B {
 B() {}
}

interface I1 {
  void op1();
}

interface I2 {
  void op2();
}

public class D extends B implements I1,I2 {
  D() {}
  public void op1() {}
  public void op2() {}
}
```

- Java source code -



- UML model generated by RE -

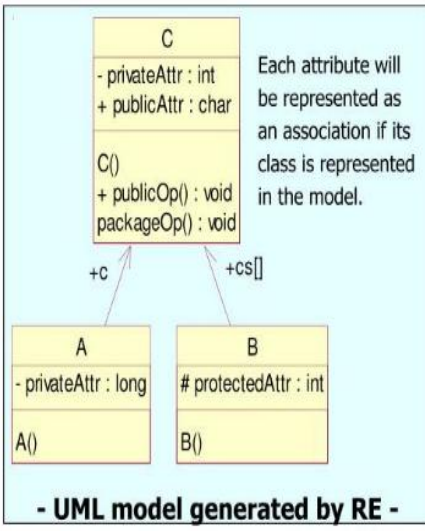
111

Reverse Engineering Java to UML

Classes and associations

```
public class C
{
   private int privateAttr;
   public char publicAttr;
   C() {}
   public void publicOp() {}
   void packageOp() {}
}

class A {
  private long privateAttr;
  public C c;
  A() {}
}

class B {
  protected int protectedAttr;
  public C[] cs;
  B() {}
}
```

- Java source code -



- UML model generated by RE -

112

Reverse Engineering Java to UML

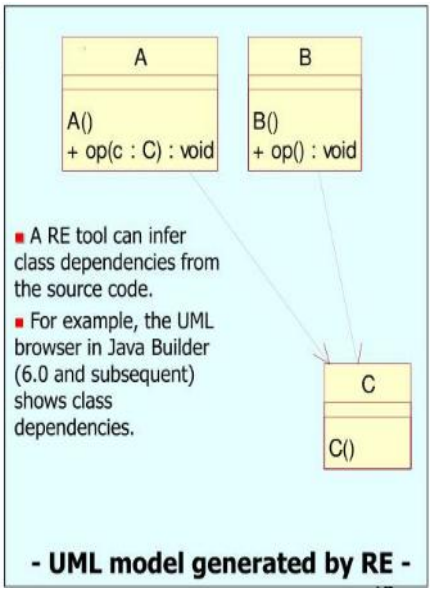
Dependency

```
class A {
 A() {}
 public void op(C c) {}
}
```

```
class B {
 B() {}
 public void op() {
  C c = new C();
 }
}
```

```
public class C {
  C() {}
 }
```

- Java source code -



- UML model generated by RE -

113

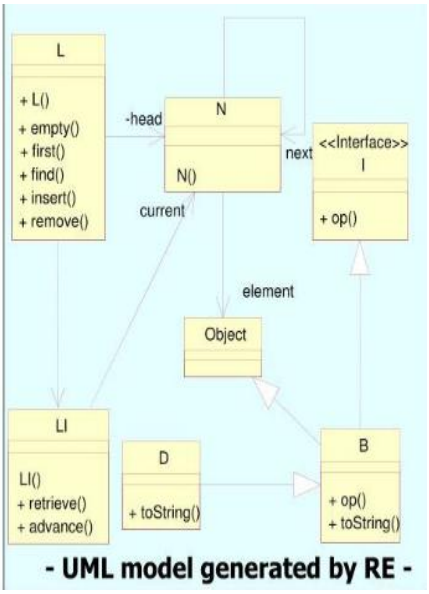
Reverse Engineering Java to UML

A larger example

```
class L {
  private N head;
  public L() {...}
  public void empty() {...}
  public LI first() {...}
  public LI find(Object x) {...}
  public void insert(Object x,LI p) {...}
  public void remove(Object x) {...}
}

class LI {
  N current;
  LI(N n) {}
  public Object retrieve() {...}
  public void advance() {...}
}
```

```
class N {
  Object element;
  N next;
  N(Object e,N n) {...}
 }

 interface I {
  void op();
 }

class B extends Object
          implements I {
  public void op() {}
  public String toString() {...}
 }

class D extends B {
  public String toString() {...}
 }
```

- Java source code -



- UML model generated by RE -

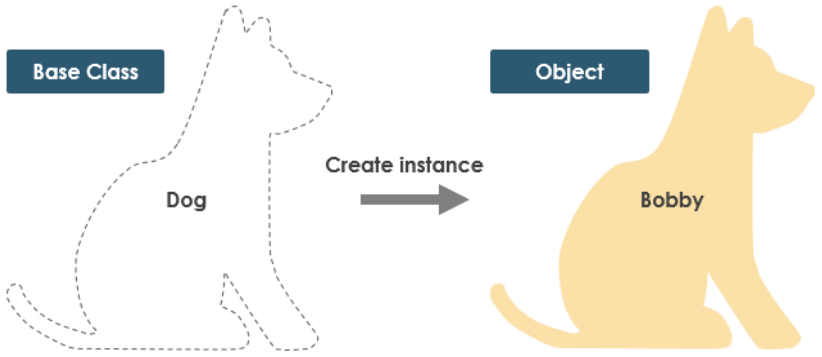
114

Object Diagram

- Object is an instance of a class in a particular moment in runtime that can have its own state and data values.
- Likewise a static UML object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time, thus an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram or a communication diagram

115

Object Diagram



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

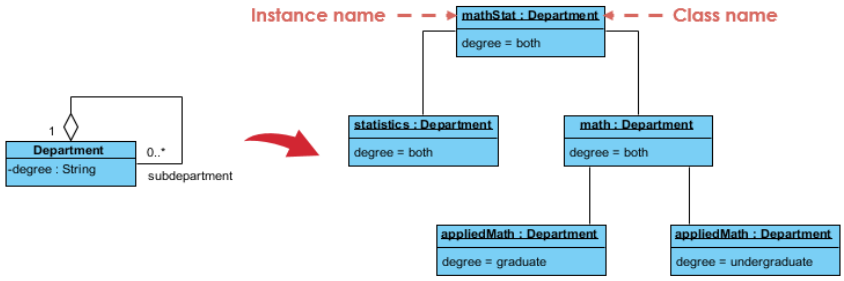
Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

Purpose of Object Diagram

- The use of object diagrams is fairly limited, mainly to show examples of data structures.
  - During the analysis phase of a project, you might create a class diagram to describe the structure of a system and then create a set of object diagrams as test cases to verify the accuracy and completeness of the class diagram.
  - Before you create a class diagram, you might create an object diagram to discover facts about specific model elements and their links, or to illustrate specific examples of the classifiers that are required.

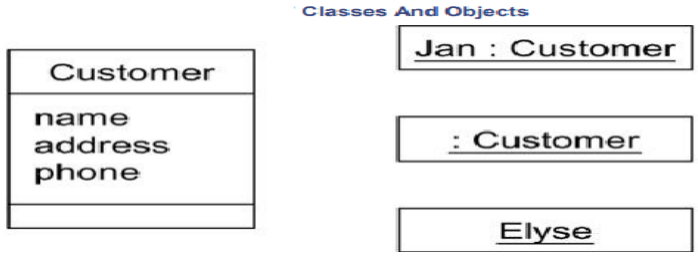
Object Diagram

- This small class diagram shows that a university Department can contain lots of other Departments and the object diagram below instantiates the class diagram, replacing it by a concrete example.



Object Diagram

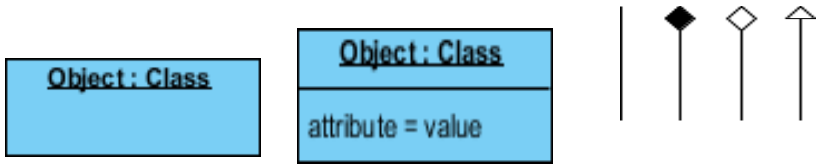
- In this figure, there is one class, named Customer, together with three objects:
  - Jan (which is marked explicitly as being a Customer object),
  - :Customer (an anonymous Customer object), and
  - Elyse (which in its specification is marked as being a kind of Customer object, although it's not shown explicitly here).



119

Basic Object Diagram Symbols and Notations

- **Object Names**: Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.
- **Object Attributes**: Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.
- **Links:** Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



120

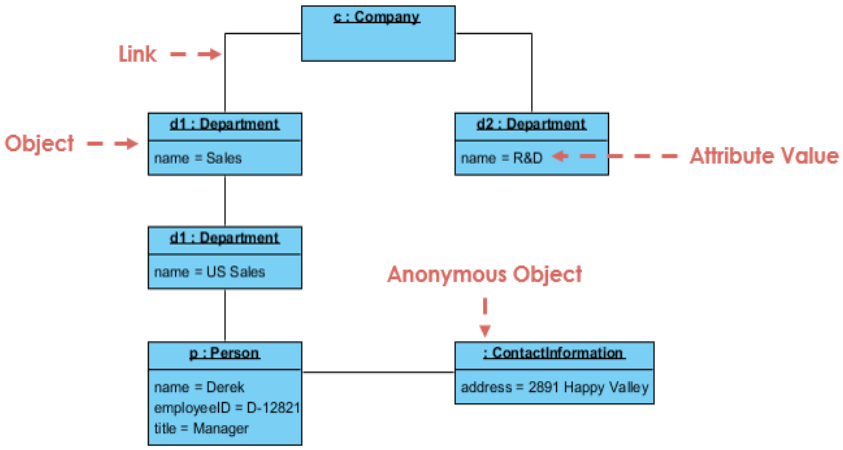
Class Diagram vs. Object Diagram

- In UML, object diagrams provide a snapshot of the instances in a system and the relationships between the instances.
  - An object diagram is a UML structural diagram that shows the instances of the classifiers in models.
  - Object diagrams use notation that is similar to that used in class diagrams.
  - Class diagrams show the actual classifiers and their relationships in a system
  - Object diagrams show specific instances of those classifiers and the links between those instances at a point in time.
  - You can create object diagrams by instantiating the classifiers in class, deployment, component, and use-case diagrams.

121

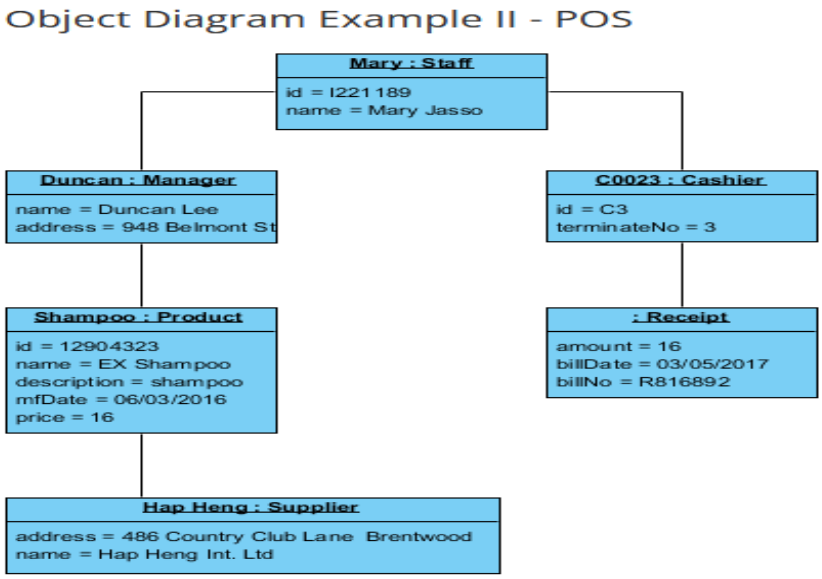
Object Diagram

Object Diagram Example I - Company Structure



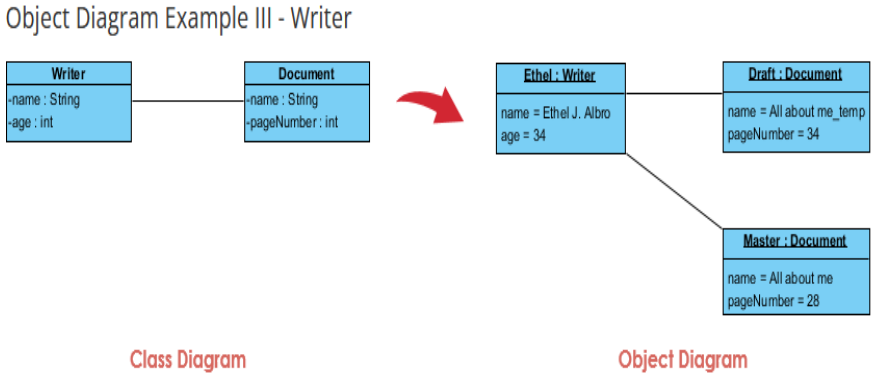
122

Object Diagram



123

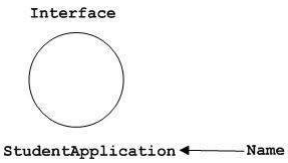
Object Diagram



124

Interface, Types and Role

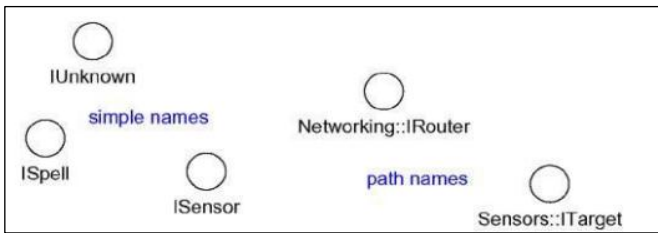
- **Interfaces** define a line between the specification of what an abstraction does and the implementation of how that abstraction does it.
- An **interface** is a collection of operations that are used to **specify a service** of a class or a component.
- A **type** is a stereotype of a class used **to specify a domain of objects** , together with the operations (but not the methods) applicable to the object.
- A **role** is the **behavior of an entity** participating in a particular context.
- Interface Notation :



125

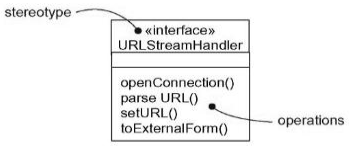
Interface, Types and Role

- Graphically, an interface is rendered as a circle; in its expanded form, an interface may be rendered as a **stereotyped class in order to expose its operations and other properties**.
- Interfaces may also be used to specify a contract for a use case or subsystem.
- **Names** : Every interface must have a name that distinguishes it from other interfaces.
  - A name is a textual string. That name alone is known as a simple name;
  - A path name is the interface name prefixed by the name of the package.



Interface, Types and Role

- **Operations :** An interface is a named collection of operations used to specify a service of a class or of a component.
  - Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation.
  - These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
  - You can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties.



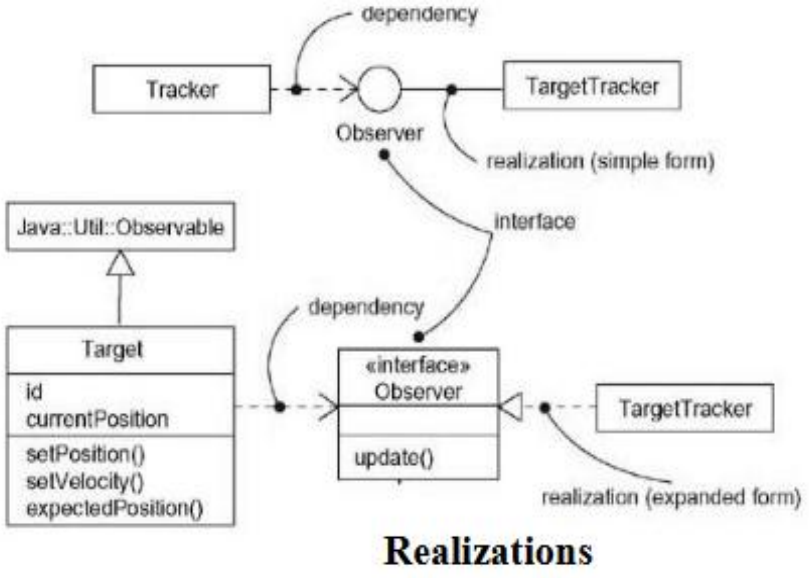
127

Interface, Types and Role

- **Relationships**: Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces.
- We can show that an element realizes an interface in two ways.
  - First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.
  - Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.

128

Interface, Types and Role

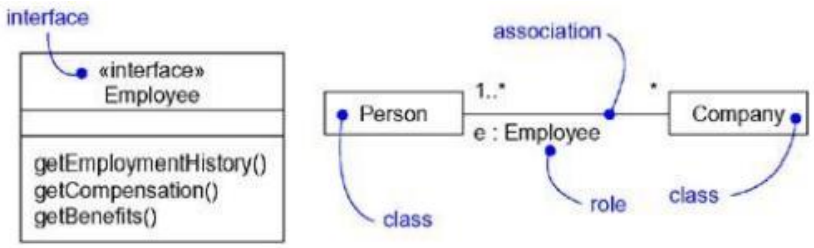


Realizations

129

Interface, Types and Role

- A class may realize many interfaces. An instance of that class must therefore support all those interfaces, because an interface defines a contract, and any abstraction that conforms to that interface must, by definition, faithfully carry out that contract.
- Nonetheless, in a given context, an instance may present only one or more of its interfaces as relevant. In that case, each interface represents a role that the object plays.



0

Types and Role

- A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.
- For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.
- An instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

131

Types

- If you want to formally model the **semantics of an abstraction** and its conformance to a specific interface, you'll want to **use the defined stereotype type**.
- Type is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type.
- The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not.
- To distinguish an interface from a class, consider prefix an I to every interface name, such as IUnknown and ISpelling.
- To distinguish a type from an interface or a class, consider prefix a T to every type, such as TNatural and TCharacter.

132

133