CLASS DIAGRAMS

UML Diagrams Used

- **Requirements Analysis**
  - **Use cases**
    - **which describe how people interact with the system .**
  - **A class diagram**
    - **drawn from the conceptual perspective, which can be a good way of building up a rigorous vocabulary of the domain .**
  - **An activity diagram**
    - **which can show the work flow of the organization, showing how software and human activities interact . An activity diagram can show the context for use cases and also the details of how a complicated use case works .**
  - **A state diagram**
    - **which can be useful if a concept has an interesting life cycle, with various states and events that change that state .**

UML Diagrams Used

- **Design**
  - **Class diagrams**
    - **from a software perspective . These show the classes in the software and how they interrelate .**
  - **Sequence diagrams**
    - **for common scenarios . A valuable approach is to pick the most important and interesting scenarios from the use cases and use sequence diagrams to figure out what happens in the software .**
  - **Package diagrams**
    - **to show the large-scale organization of the software.**
  - **State diagrams**
    - **for classes with complex life histories .**
  - **Deployment diagrams**
    - **to show the physical layout of the software .**

Documentation

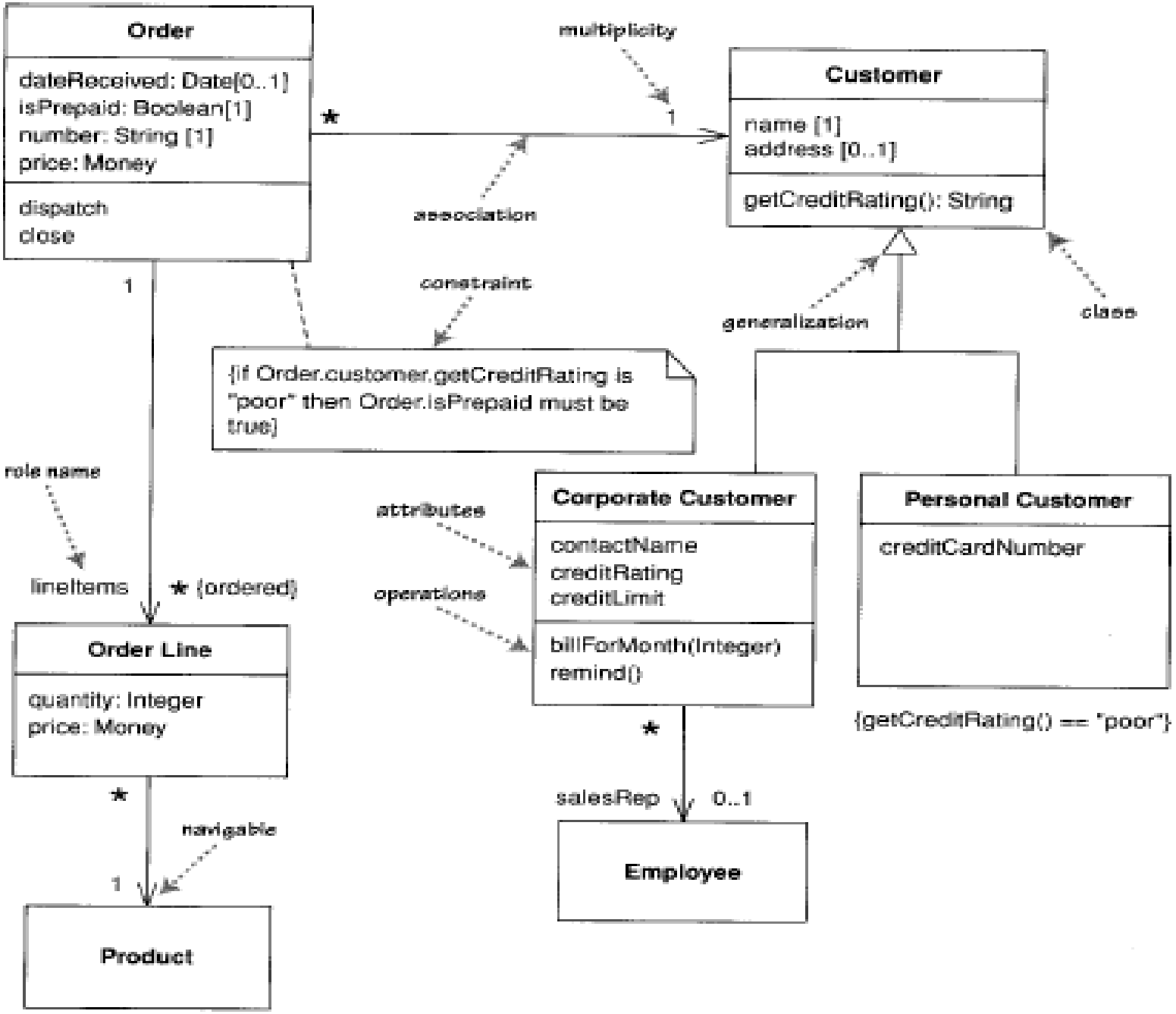
- **A package diagram**
  - makes a good logical road map of the system . This diagram helps understand the logical pieces of the system and see the dependencies and keep them under control .
- **A deployment diagram**
  - which shows the high-level physical picture, may also prove useful at this stage .
- **A class diagram**
  - within each package.
  - Don't show every operation on every class .
  - Show only the important features that help understand what is in there.
  - This class diagram acts as a graphical table of contents .
- The class diagram should be supported by a handful of **interaction diagrams**
  - that show the most important interactions in the system . Again, selectivity is important here.

Documentation

- If a class has complex life-cycle behavior, draw **a state machine diagram**
  - Do this only if the behavior is sufflciently complex.
- Include some important **code**, written in pseudecode.
- If a particularly complex algorithm is involved, consider using an **activity diagram**
  - but only if it gives more understanding than the code alone .
- One of the most important things to document is the **design alternatives you didn't take and why you didn't do them**
  - That's often the most forgotten but most useful piece of external documentation you can provide .

Class Diagrams

- Describe the types of objects in the system and the various kinds of static relationships that exist among them.
- Show the properties and operations of a class and the constraints that apply to the way objects are connected .
- The boxes in the diagram are classes, which are divided into three compartments:
  - the name of the class (in bold),
  - its attributes, and
  - its operations


Order
dateReceived: Date[0..1]
isPrepaid: Boolean[1]
number: String [1]
price: Money
dispatch
close
multiplicity
1
*
Customer
name [1]
address [0..1]
getCreditRating(): String
association
constraint
generalization
class
1
{if Order.customer.getCreditRating is "poor" then Order.isPrepaid must be true}
role name
lineItems
* {ordered}
Order Line
quantity: Integer
price: Money
*
navigable
1
Product
attributes
operations
Corporate Customer
contactName
creditRating
creditLimit
billForMonth(Integer)
remind()
Personal Customer
creditCardNumber
{getCreditRating() == "poor"}
*
salesRep
0..1
Employee

Attributes

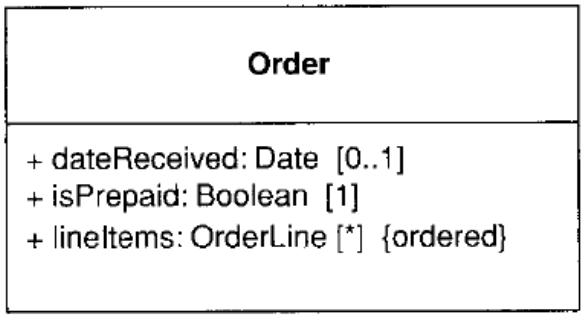
- The full form of an attribute is :
  - ***visibility name : type multiplicity = default {property-string}***
- An example of this is :
  - - name : String [1] = "Untitled" {readOnly}

- Only the name is necessary .
- Visibility marker indicates whether the attribute is public (+) or private (-)
- The type of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in a programming language.
- Multiplicity will be explained later.
- The default value is the value for a newly created object if the attribute isn't specified during creation.
- The {property-string} allows you to indicate additional properties for the attribute .
  - {readOnly} indicates that clients may not modify the property.

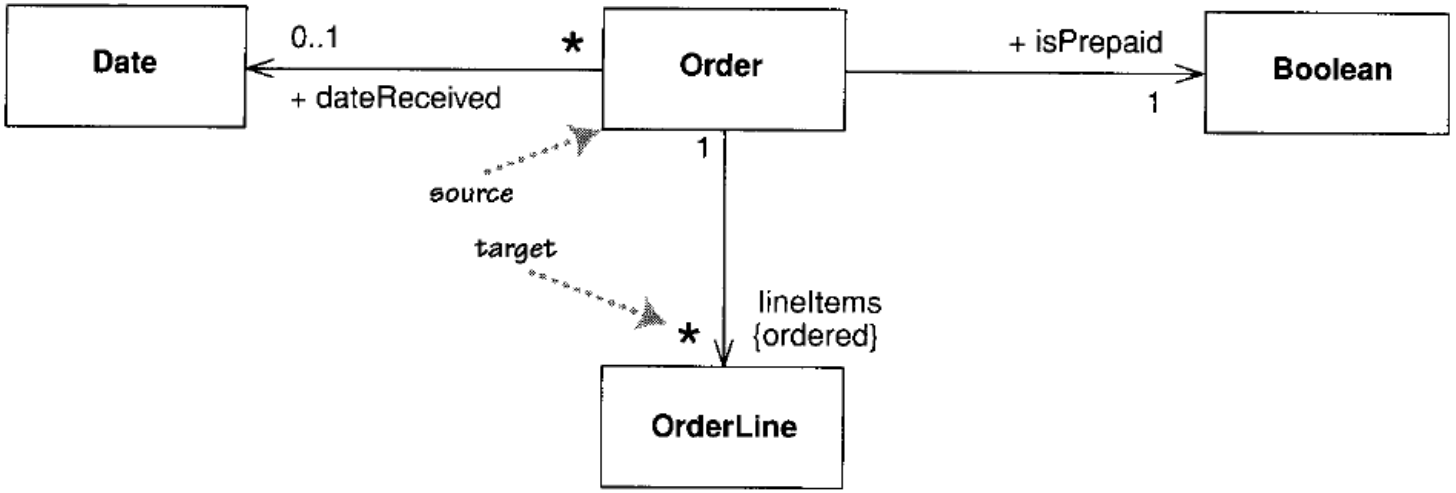
Associations

- The other way to notate a property is as an association .
- Much of the same information that you can show on an attribute appears on an association.
- An association is a solid line between two classes,
  - directed from the source class to the target class.
  - The name of the property goes at the target end of the association, together with its multiplicity .
  - The target end of the association links to the class that is the type of the property .

Attributes vs. Associations



Showing properties of an Order as attributes



Showing properties of an Order as associations

Attributes vs. Associations

- Although most of the same information appears in both notations, some items are different.
  - In particular, associations can show multiplicities at both ends of the line.

- Why should you use one or the other?
  - You can use attributes for small things, such as dates or Booleans- in general, value types.
  - You can use associations for more significant classes, such as customers and orders .
  - Also, you can use class boxes for classes that are significant for the diagram,
    - which leads to using associations, and attributes for things less important for that diagram.

Multiplicity

- The multiplicity of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are:
  - 1 (An order must have exactly one customer .)
  - 0..1 (A corporate customer may or may not have a single sales rep.)
  - * (A customer need not place an Order and there is no upper limit to the number of Orders, a Customer may place-zero or more orders .)

- More generally, multiplicities are defined with a lower bound and an upper bound,
  - such as 2..4 for players of a game.
- The lower bound may be any positive number or zero; the upper is any positive number or *(for unlimited).
- If the lower and upper bounds are the same, you can use one number;
  - 1 is equivalent to 1. .1
- Because it's a common case, * is short for 0..* .

Multiplicity

- In attributes, you come across various terms that refer to the multiplicity.
  - Optional implies a lower bound of 0 .
  - Mandatory implies a lower bound of 1 or possibly more .
  - Single-valued implies an upper bound of 1 .
  - Multivalued implies an upper bound of more than 1 : usually *
- By default, the elements in a multivalued multiplicity form a set, so if you ask a customer for its orders, they do not come back in any order .
  - If the ordering of the orders in association has meaning, you need to add **{ordered}** to the association end.
  - If you want to allow duplicates, add **{nonunique}**.
  - If you want to explicitly show the default, you can use **{unordered}** and **{unique}**.
  - You may also see collection-oriented names, such as **{bag}** for unordered, nonunique.
- The default multiplicity of an attribute is [1].

Programming Interpretation of Properties

```
public class OrderLine . . .
  private int quantity ;
  private Money price ;
  private Order order ;
  private Product product
```

Programming Interpretation of Properties

- For private attributes, you may see the fields exposed through accessor methods (getters and setters).
- A read-only attribute will have no setter method (with fields).
- We might see the OrderLine's attributes corresponding to the following methods :

Programming Interpretation of Properties

```
public class OrderLine . . .
   private int quantity ;
   private Product product ;
   public int getQuantity () {
       return quantity ;
   }
   public void setQuantity(int quantity) {
       this.quantity = quantity ;
   }
   public Money getPrice () {
       return product.getPrice().multiply(quantity);
   }
```

Programming Interpretation of Properties

- In this case, there is no data field for price;
  - instead, it's a computed value.
- But as far as clients of the OrderLine class are concerned, it looks the same as a field.
- Clients can't tell what is a field and what is computed.
- This information hiding is the essence of encapsulation.

Programming Interpretation of Properties

- If an attribute is multivalued,
  - this implies that the data concerned is a collection.
- So an Order class would refer to a collection of OrderLines.
- Because this multiplicity is ordered, that collection must be ordered, (such as a List in Java).
- If the collection is unordered, it should, strictly, have no meaningful order and thus be implemented with a set.
  - You may use arrays, but the UML implies an unlimited upper bound, so use a collection for data structure.

Programming Interpretation of Properties

```
class Order {
    private Set lineItems = new HashSet() ;
    public Set getLineItems() {
        return Collections.unmodifiableSet(lineItems) ;
    }
    public void addLineItem (OrderItem arg) {
        lineItems.add(arg) ;
    }
    public void removeLineItem (OrderItem arg) {
        lineItems.remove(arg) ;
    }
```

Bidirectional Associations

- A bidirectional association is a pair of properties that are linked together as inverses.
  - the Car class has property *owner :Person[0..1]*,
  - the Person class has a property *cars :Car[*]* .
- The inverse link between them implies that if you follow both properties, you should get back to a set that contains your starting point .
  - For example, if you begin with a particular Mercedes, find its owner, and then look at its owner's cars, that set should contain the Mercedes that you started from.

Operations

- Operations correspond to the methods on a class .
- Normally, you don't show those operations that simply manipulate properties, because they can usually be inferred .

- The full UML Syntax for operations is :
  - ***visibility name (parameter-list) : return-type {property-string}***

- Visibility marker is public (+) or private (-).
- The name is a string.
- The parameter-list is the list of parameters for the operation .
- The return-type is the type of the returned value, if there is one .
- The property-string indicates property values that apply to the given operation .

Operations

- The parameters in the parameter list are notated in a similar way to attributes .
- The form is :
  - ***direction name : type = default value***
- The **name, type**, and **default value** are the same as for attributes .
- The **direction**
  - indicates whether the parameter is *input (in), output (out)* or *both (inout)*.
  - If no direction is shown, it's assumed to be *(in)*.
- An example operation on account might be :
  - + balanceOn (date : Date) : Money

Generalization

- A typical example of generalization involves the personal and corporate customers of a business .
  - They have differences but also many similarities.
  - The similarities can be placed in a general Customer class (the Supertype), with Personal Customer and Corporate Customer as subtypes.
- This phenomenon is also subject to various interpretations at the various perspectives of modeling .
  - Conceptually, we can say that Corporate Customer is a subtype of Customer if all instances of Corporate Customer are also, by definition, instances of Customer .
  - A Corporate Customer is then a special kind of Customer. The key idea is that everything we say about a Customer -associations, attributes, operations- is true also for a Corporate Customer.

Generalization

- With a software perspective, the obvious interpretation is inheritance:
  - The Corporate Customer is a subclass of Customer .
  - In OO languages, the subclass inherits all the features of the superclass and may override any superclass methods.
- An important principle of using inheritance effectively is substitutability.
  - You should be able to substitute a Corporate Customer within any code that requires a Customer, and everything should work fine .
  - This means that if you write code assuming you have a Customer, you can freely use any subtype of Customer.

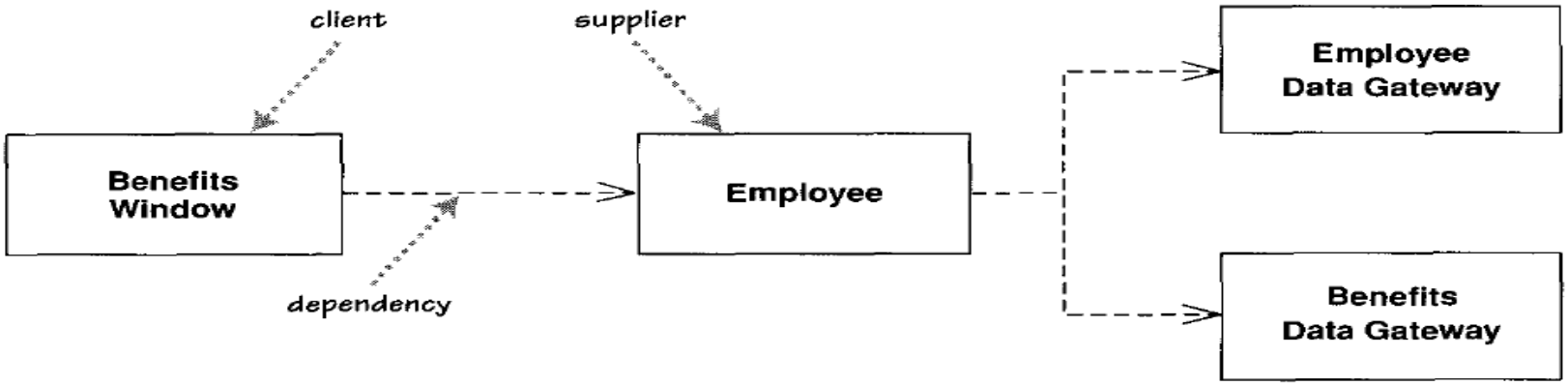
Notes and Comments

- Notes are comments in the diagrams .
- Notes can stand on their own, or they can be linked with a dashed line to the elements they are commenting
- Sometimes, it's useful to have an in-line comment on a diagram element .
  - You can do this by prefixing the text with two dashes : -- .

Dependency

- A dependency exists between two elements
  - if changes to the definition of one element (the supplier) may cause changes to the other (the client) .
- With classes, dependencies exist for various reasons:
  - one class sends a message to another ;
  - one class has another as part of its data ;
  - one class mentions another as a parameter to an operation .
- If a class changes its interface, any message sent to that class may no longer be valid .
- The UML allows you to depict dependencies between all sorts of elements.
- You use dependencies whenever you want to show how changes in one element might alter other elements .

Dependency



- The Benefits Window class

  is dependent on

- the Employee class: a domain object that captures the essential behavior of the system.
- This means that if the Employee class changes its interface, the Benefits Window may have to change .