272: Software Engineering

Instructor: Tevfik Bultan

Lecture 1: Software Specification Problem, Unified Modeling Language (UML), Object Constraint Language (OCL)

Software Specification Problem

- In different phases of the software process we need ways to specify the deliverable for that phase
  - Need to specify the requirements
    - What is the software system expected to do, what is its functionality?
  - Need to specify the design
    - We need to document and communicate the design
  - Need to specify the implementation
    - Comments
    - Assertions

Specification Languages

- Main issue: When you write code you write it in a programming language
  - How do you *write* the requirements?
  - How do you *write* the design?
- Specification languages
  - Used to specify the requirements or the design
  - Parts of requirements are necessarily in English (customer has to understand). To bring some structure to requirements one can use semi-formal techniques such as UML use-case diagrams. Depending on the application you maybe able to use formal techniques too
  - For design you can use UML class diagrams, sequence diagrams, state diagrams, activity diagrams
  - Some specification languages (such as UML class diagrams are supported with code generation tools)

Specification

- Specifications can be
  - Informal
    - No formal syntax or semantics
      - for example in English
    - Informal specifications can be ambiguous and imprecise
  - Semiformal
    - Syntax is precise but does not have formal semantics
    - UML (Universal Modeling Language) class diagrams, sequence diagrams
  - Formal
    - Both syntax and semantics are formal
    - Alloy, Z, Statecharts, SDL (Specification and Design Language), Message Sequence Charts (MSC), Petri nets, CSP (Communicating Sequential Processes), Process Algebras, SCR (Software Cost Reduction), RSML (Requirements State Machine Language), …

Ambiguities in Informal Specifications

- "The input can be typed or selected from the menu"
  - The input can be typed or selected from the menu or both
  - The input can be typed or selected from the menu but not both

- "The number of songs selected should be less than 10"
  - Is it strictly less than?
  - Or, is it less than or equal?

- "The user has to select the options A and B or C"
  - Is it "(A and B) or C"
  - Or, is it "A and (B or C)"

A Success Story: RSML and TCAS

- Requirements State Machine Language (RSML)
  - A formal specification language based on hierarchical state machines (statecharts)
- The developers of RSML applied it to the specification of Traffic Collision Avoidance System (TCAS) to demonstrate benefits of using RSML [Leveson et al. 1994]
  - TCAS: the specification of a software system which is required on all aircraft in USA carrying more than 30 passengers During the specification of TCAS in RSML ambiguities were discovered in the original English specification of TCAS
  - Eventually FAA decided to use the RSML versions of the TCAS specification

Another Example Formal Specification

- Formal specifications avoid ambiguity
  - However, they could be hard to understand
  - And it is not easy to write formal specifications

- Let's try to specify a sorting procedure formally (mathematically)
- I will just use basic Math concepts: functions, integers, arithmetic
  - Input: I : An array of size n of integers
    - How do we formally specify what an *array* is?
    - $I : \mathbf{Z} \rightarrow \mathbf{Z}$ (a function from integers to integers)
    - $I : 1 \ldots n \rightarrow \mathbf{Z}$
    - $n \geq 1$

Example: Sorting

– Output: $O : 1 \ldots n \rightarrow \mathbf{Z}$

- $\forall\, i\,,\ O(i) \leq O(i{+}1)$
- $\forall\, i\,,\ 1 \leq i \leq n \Rightarrow O(i) \leq O(i{+}1)$
- $\forall\, i\,,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1)$
- $(\forall\, i\,,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

  $\wedge\ (\forall\, i\,,\ 1 \leq i \leq n \Rightarrow (\exists\, j\,,\ 1 \leq j \leq n\ \wedge O(i) = I(j)))$
- $(\forall\, i\,,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

  $\wedge\ (\forall\, i\,,\ 1 \leq i \leq n \Rightarrow (\exists\, j\,,\ 1 \leq j \leq n\ \wedge O(i) = I(j)))$

  $\wedge\ (\forall\, i\,,\ 1 \leq i \leq n \Rightarrow (\exists\, j\,,\ 1 \leq j \leq n\ \wedge I(i) = O(j)))$
- $(\forall\, i\,,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

  $\wedge\ (\exists\, f\ : 1 \ldots n \rightarrow 1 \ldots n\ ,$

  $\quad (\forall\, i\,, j\,,\ (1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j) \Rightarrow f(i) \neq f(j))$

  $\quad \wedge\ (\forall\, i,\ 1 \leq i \leq n \Rightarrow O(i) = I(f(i))))$

UML (Unified Modeling Language)

- Combines several visual specification techniques
  - use case diagrams, component diagrams, package diagrams, deployment diagrams, class diagrams, sequence diagrams, collaboration diagrams, state diagrams, activity diagrams
- Based on object oriented principles and concepts
  - encapsulation, abstraction
  - classes, objects
- Semi-formal
  - Precise syntax but no formal semantics
  - There have been efforts in formalizing UML semantics
- The Object Management Group (OMG, a computer industry consortium) defines the UML standard
  - The current UML language specification is available at:

    http://www.uml.org/

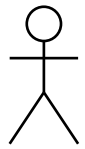
UML

- UML can be used in all phases of software development
  - specification of requirements, architectural design, detailed design and implementation

- There are different types of UML diagrams for specifying different aspects of software:
  - Functionality, requirements
    - Use-case diagrams
  - Architecture, modularization, decomposition
    - Class diagrams (class structure)
    - Component diagrams, Package diagrams, Deployment diagrams (architecture)
  - Behavior
    - State diagrams, Activity diagrams
  - Communication, interaction
    - Sequence diagrams, Collaboration diagrams

Use cases for Requirements Specification

- Use cases document the behavior of the system from the users' point of view.
  - By user we mean anything external to the system
- An **actor** is a role played by an outside entity that interacts directly with the system
  - An actor can be a human, or a machine or program
  - Actors are shown as stick figures in use case diagrams



Customer

Use cases

- A **use case** describes the possible sequences of interactions among the system and one or more actors in response to some initial stimulus by one of the actors
  - Each way of using the system is called a use case
  - A use case is not a single scenario but rather a description of *a set of scenarios*
  - For example: *Creating an account*
  - Individual use cases are shown as named ovals in use case diagrams

- A use case involves a sequence of interactions between the initiator and the system, possibly involving other actors.
- In a use case, the system is considered as a black-box. We are only interested in externally visible behavior

Documenting use cases: Online Shopping

Use case: Place Order **Actors:** Costumer

Precondition: A valid user has logged into the system

Flow of Events:

1. The use case begins when the customer selects Place Order
2. The customer enters his or her name and address
3. If the customer enters only the zip code, the system supplies the city and state
4. The customer enters product codes for products to be ordered
5. For each product code entered
    a) the system supplies a product description and price
    b) the system adds the price of the item to the total

end loop

6. The customer enters credit card payment information
7. The customer selects Submit
8. The system verifies the information [Exception: Information Incorrect], saves the order as pending, and forwards payment information to the accounting system.
9. When payment is confirmed [Exception: Payment not Confirmed], the order is marked confirmed, an order ID is returned to the customer, and the use case terminates

Exceptions:

Payment not Confirmed: the system will prompt the customer to correct payment information or cancel. If the customer chooses to correct the information, go back to step 6 in the Basic Path. If the customer chooses to cancel, the use case terminates.

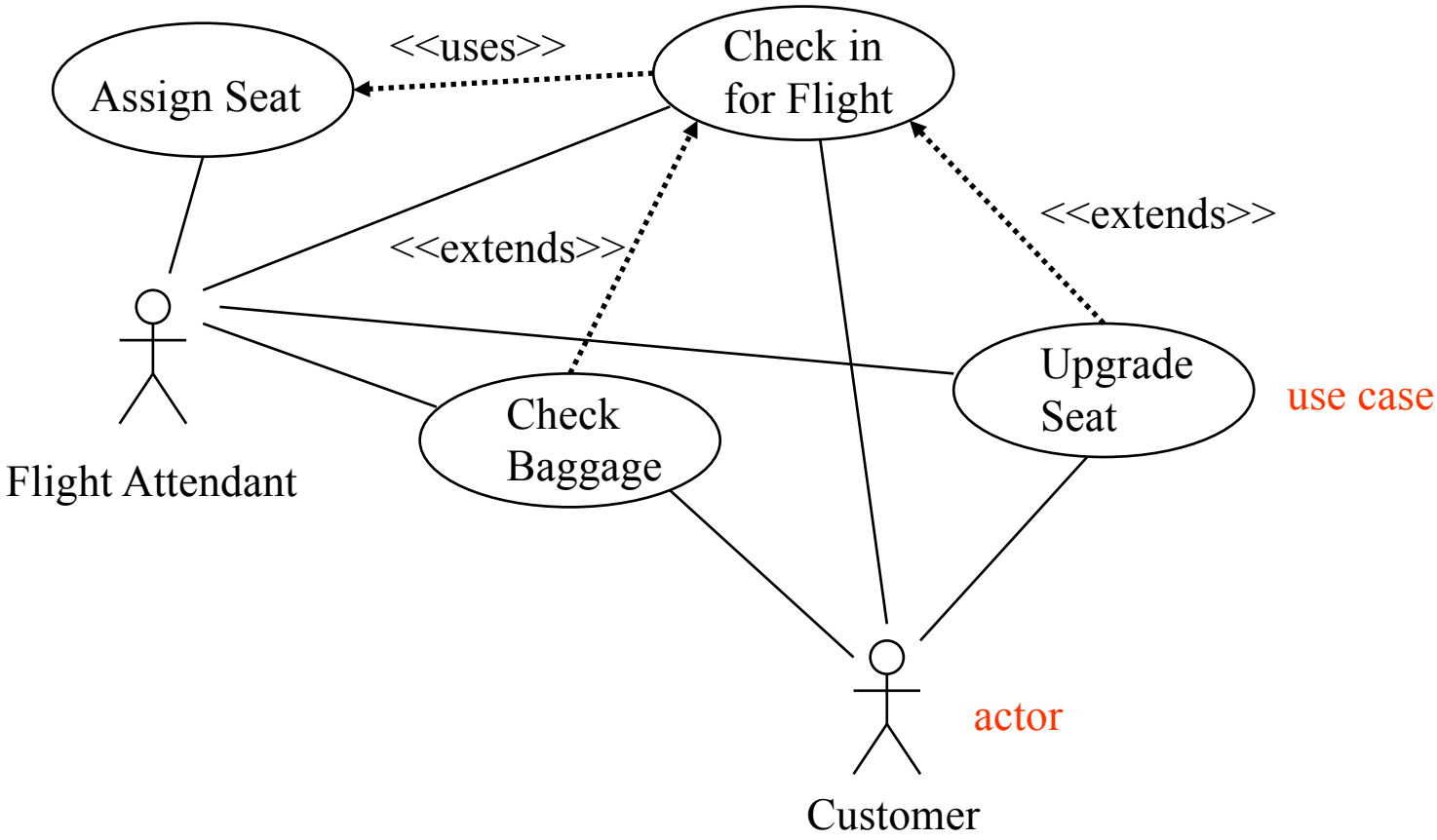
Information Incorrect: If any information is incorrect, the system prompts the customer to correct it.

Postcondition: If the order was not canceled, it is saved in the system and marked confirmed

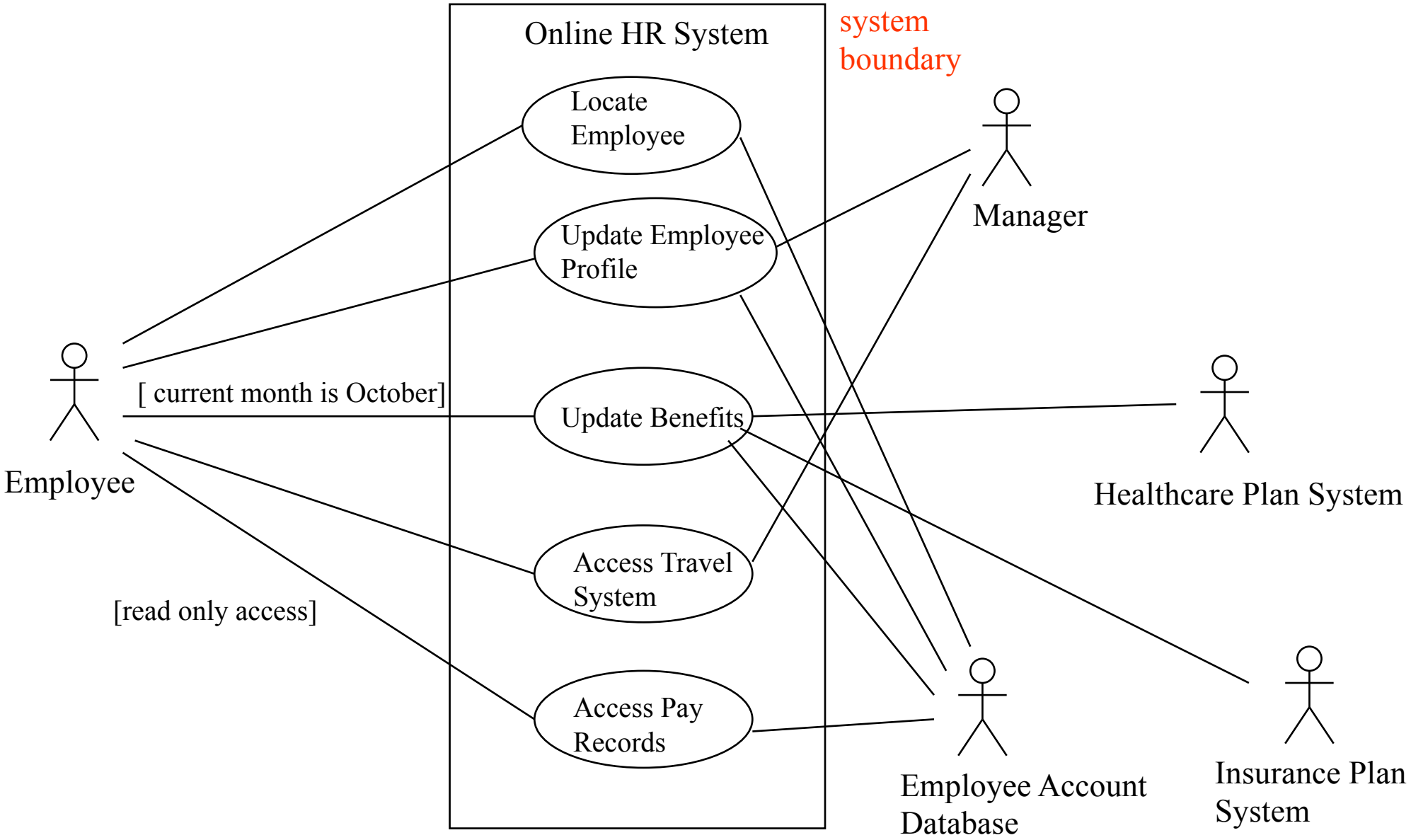
Combining use cases

- A use case **extends** another use case when it embeds new behavior into a complete base case
  - *Check Baggage* **extends** the base case *Check in for Flight*
  - You do not have to check baggage to check in for flight.
- A use case **uses** another use case when it embeds a subsequence as a necessary part of a larger case (In some texts this relationship is called **includes** instead of **uses**)
  - **uses** relationship permits the same behavior to be embedded in many otherwise unrelated use cases
  - For example C*heck in for Flight* use case **uses** *Assign Seat* use case
- The difference is
  - In the **extends** the extended use case is a valid use case by itself
  - In the **uses** the use case which is using the other use case is not complete without it

UML use case diagrams



Online Human Resources (HR) System



UML Class Diagrams

- Class diagram describes
  - Types of objects in the system
  - Static relationships among them

- Two principal kinds of static relationships
  - Associations between classes
  - Subtype relationships between classes

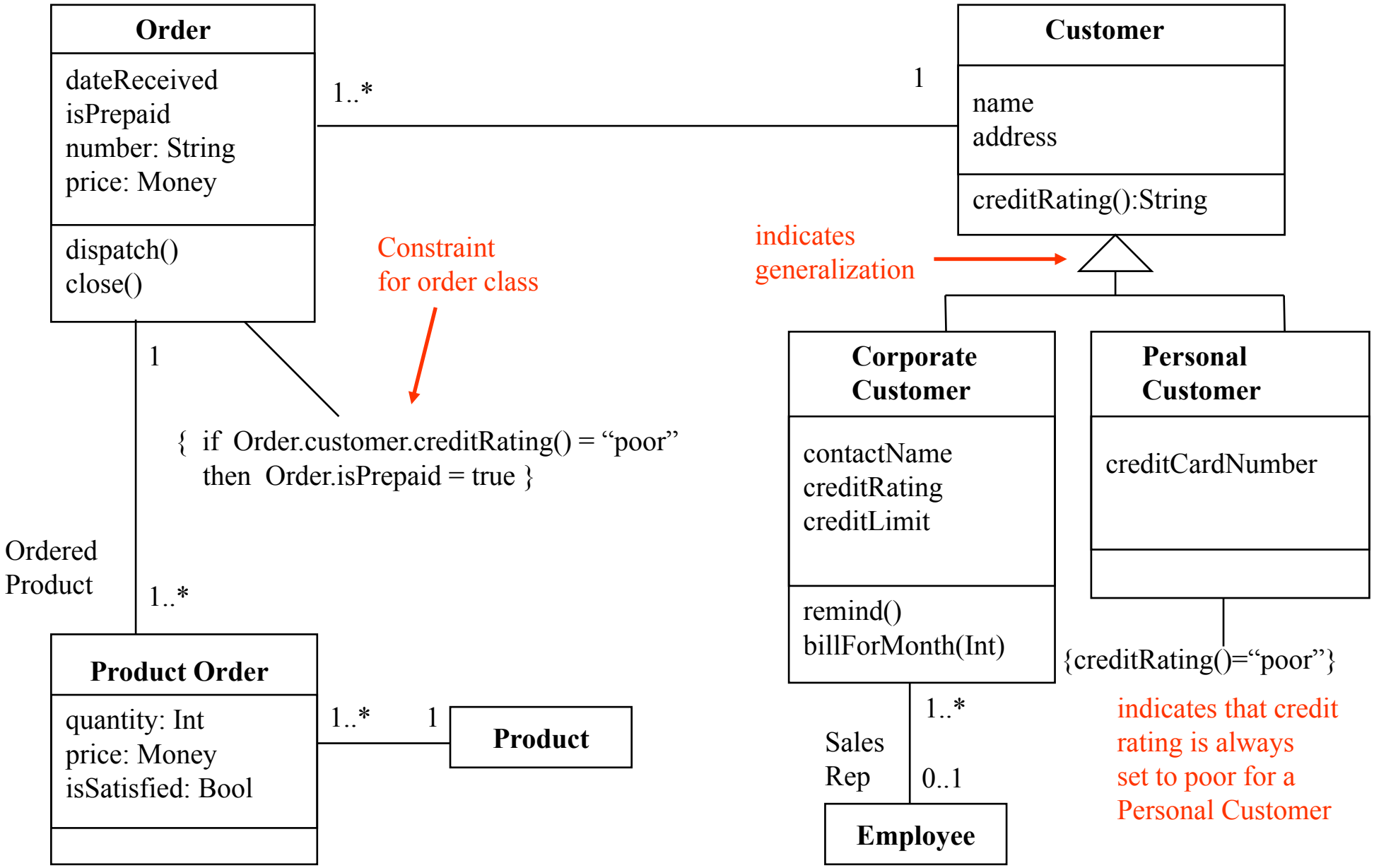
- Class descriptions show
  - Attributes
  - Operations

- Class diagrams can also show constraints on associations

Sequence Diagrams

- A sequence diagram shows a particular sequence of messages exchanged between a number of objects
- Sequence diagrams also show behavior by showing the ordering of message exchange
- A sequence diagram shows some particular communication sequences in some run of the system
  - it is not characterizing all possible runs

Example Class Diagram



Sequence Diagrams

- A sequence diagram shows a particular sequence of messages exchanged between a number of objects
- Sequence diagrams also show behavior by showing the ordering of message exchange
- A sequence diagram shows some particular communication sequences in some run of the system
  - it is not characterizing all possible runs

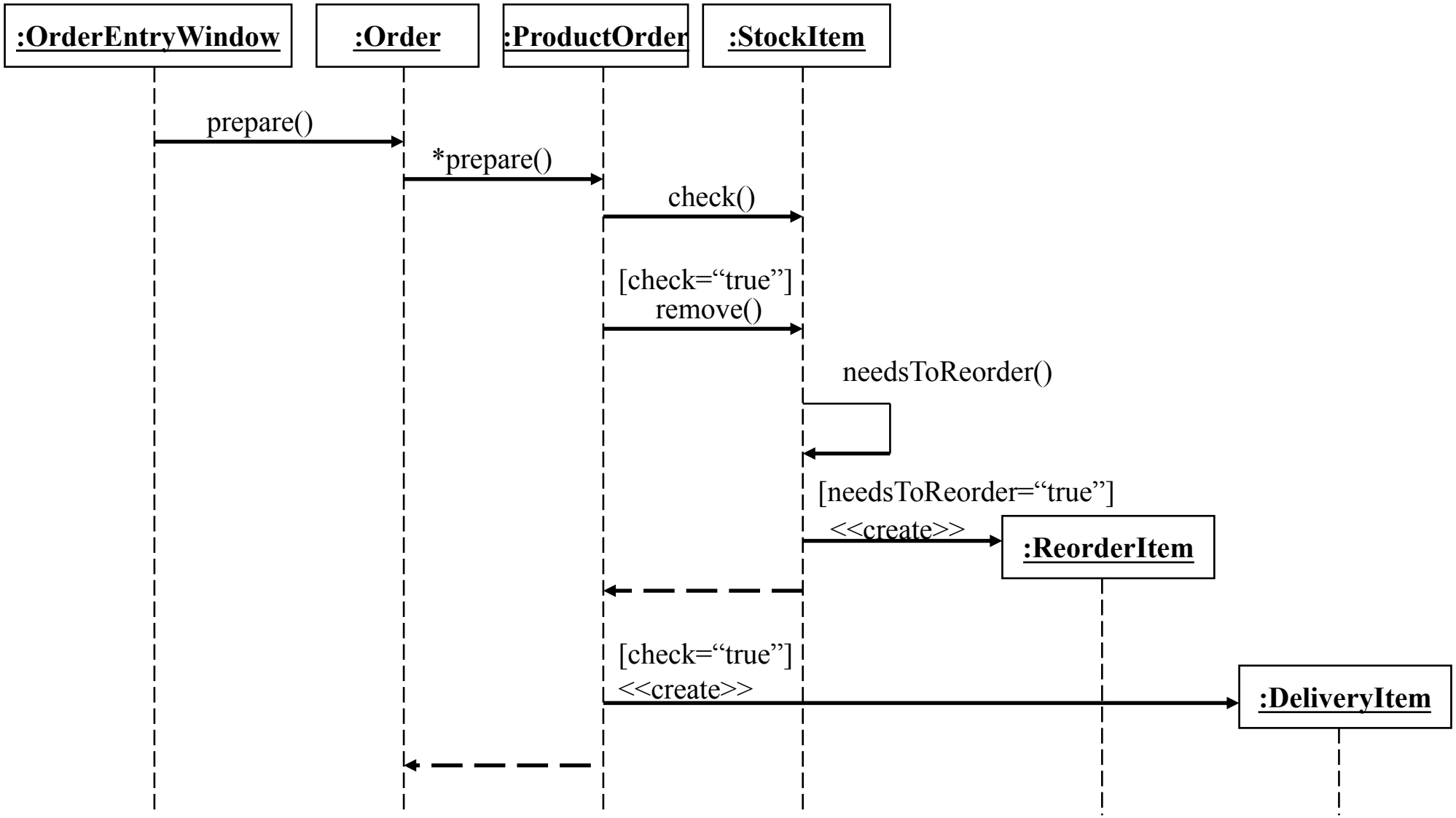
Sequence Diagrams

- Sequence diagrams can be used in conjunction with use-cases
  - At the requirements phase they can be used to visually represent the use cases
  - At the design phase they can be used to show the system's behavior that corresponds to a use-case

- During the testing phase sequence diagrams from the requirements or design phases can be used to generate test cases for the software product

- Sequence diagrams are similar to MSCs (Message Sequence Charts) which are a part of SDL(Specification and Description Language) and have formal semantics

Example Sequence Diagram



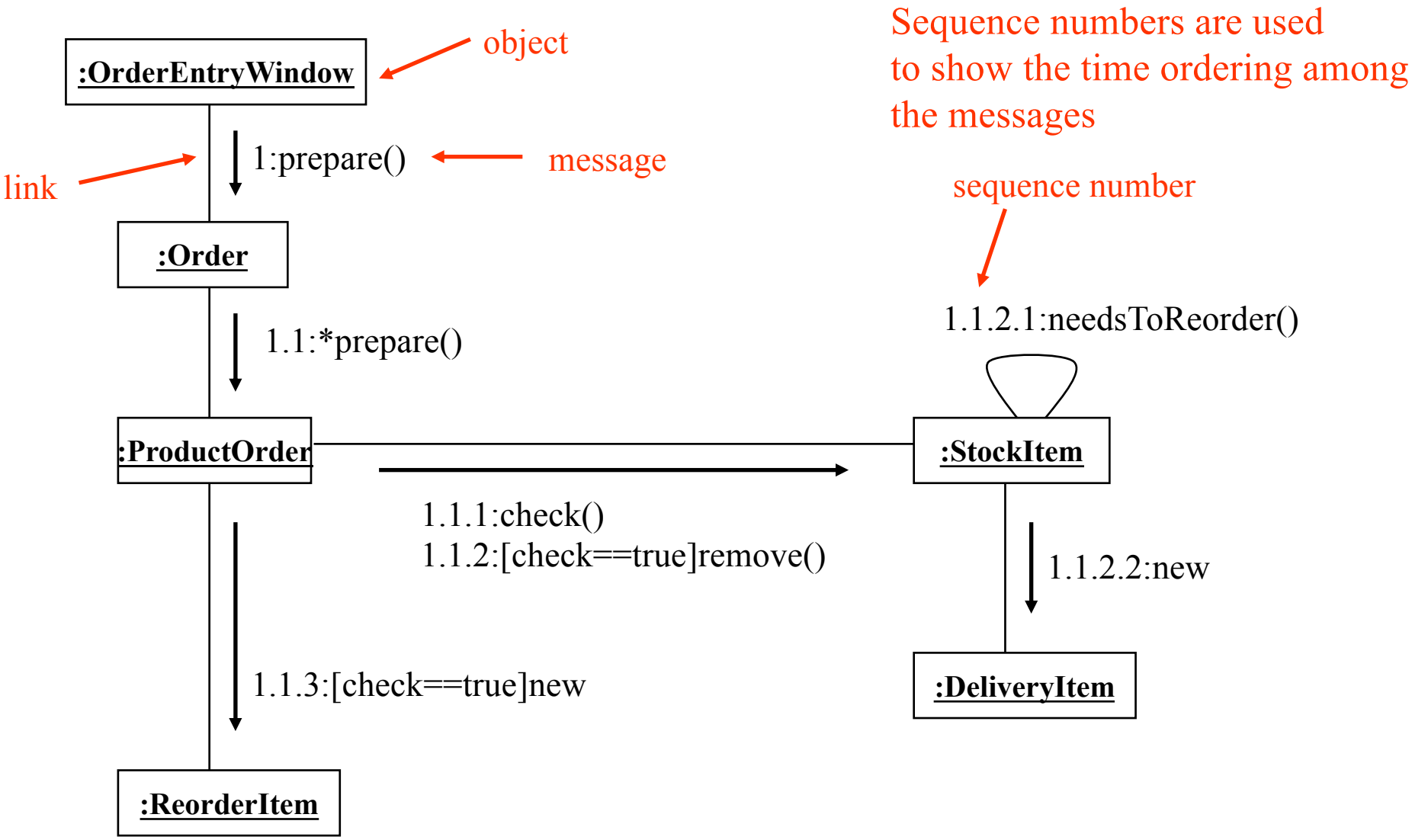
Collaboration (Communication) Diagrams

- Collaboration diagrams (aka Communication diagrams) show a particular sequence of messages exchanged between a number of objects
  - this is what sequence diagrams do too!

- Use sequence diagrams to model flows of control by time ordering
  - sequence diagrams can be better for demonstrating the ordering of the messages
  - sequence diagrams are not suitable for complex iteration and branching

- Use collaboration diagrams to model flows of control by organization
  - collaboration diagrams are good at showing the static connections among the objects while demonstrating a particular sequence of messages at the same time

Corresponding Collaboration Diagram



State Diagrams

- State diagrams are used to show possible states a single object can get into
  - shows states of an object
- How object changes state in response to events
  - shows transitions between states
- Uses the same basic ideas from statecharts and adds some extra concepts such as internal transitions, deferred events etc.
  - “A Visual Formalism for Complex Systems,” David Harel, Science of Computer Programming, 1987
  - Statecharts are basically hierarchical state machines
  - Statecharts have formal semantics

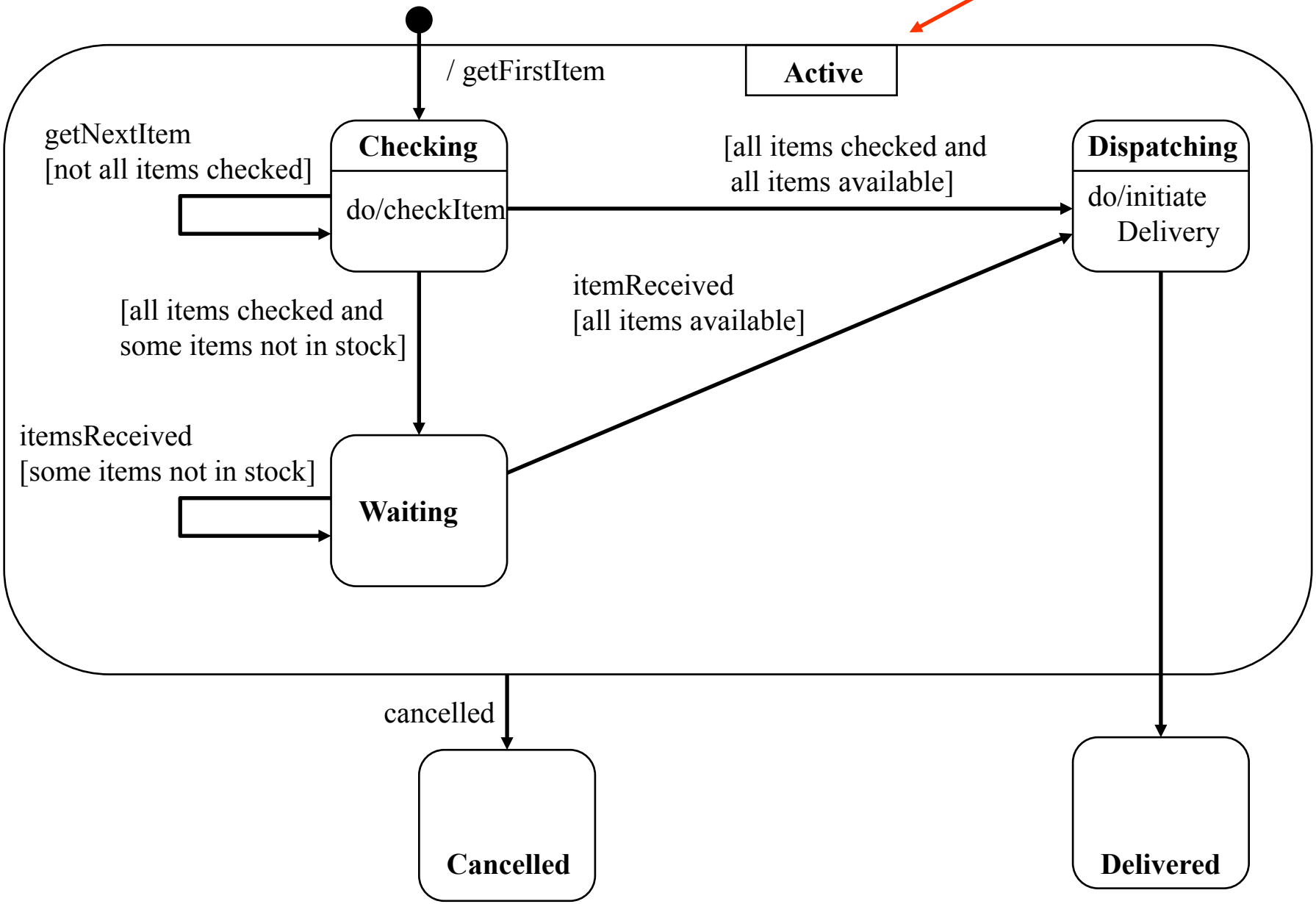
State Diagrams

- Hierarchical grouping of states
  - composite states are formed by grouping other states
  - A composite state has a set of sub-states

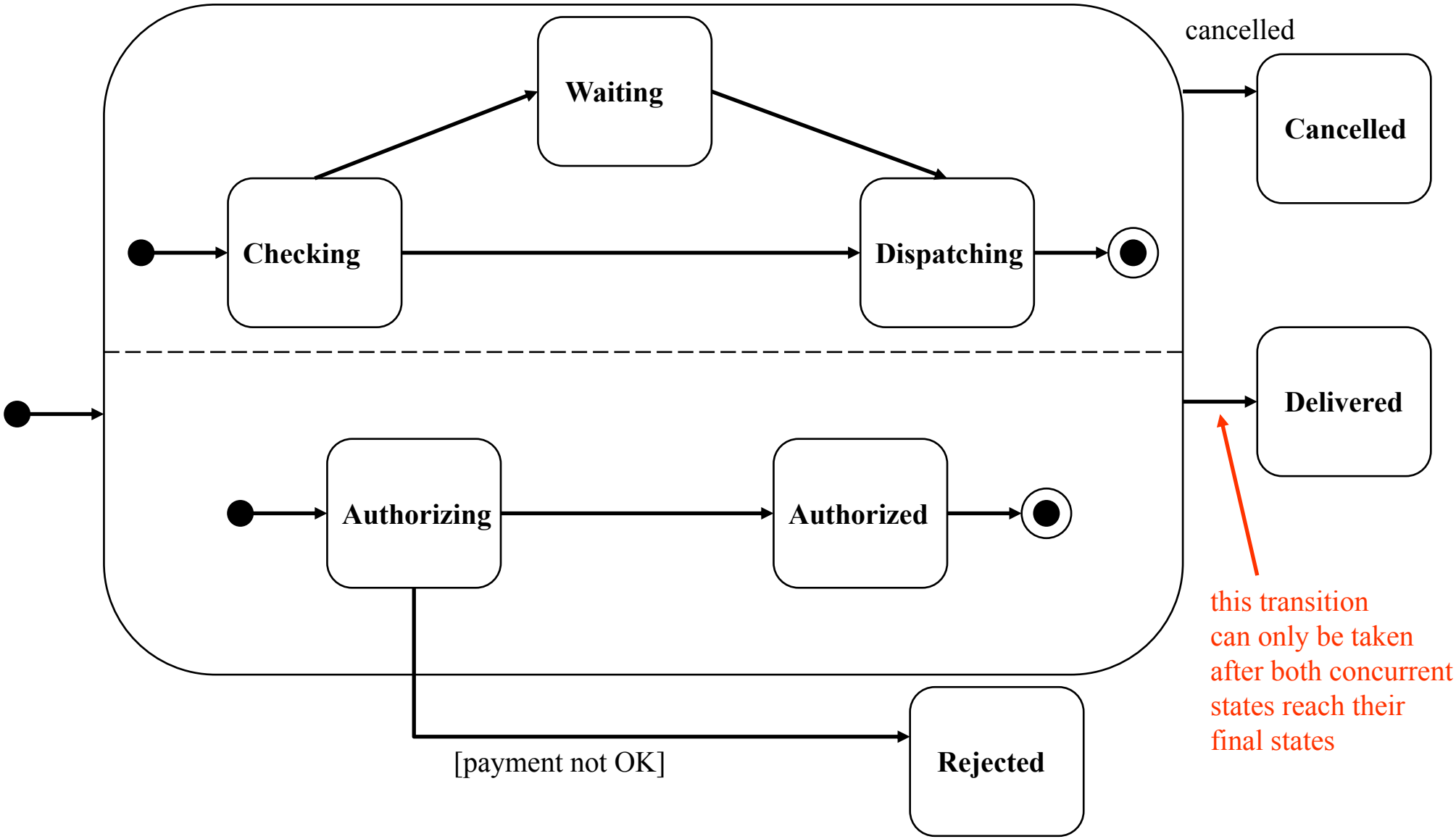
- Concurrent composite states can be used to express concurrency
  - When the system is in a concurrent composite state, it is in all of its substates at the same time
  - When the system is in a normal (non-concurrrent) composite state, it is in only one of its substates
  - If a state has no substates it is an atomic state

- Synchronization and communication between different parts of the system is achieved using events

State Diagrams: Superstates



State Diagrams: Concurrent States

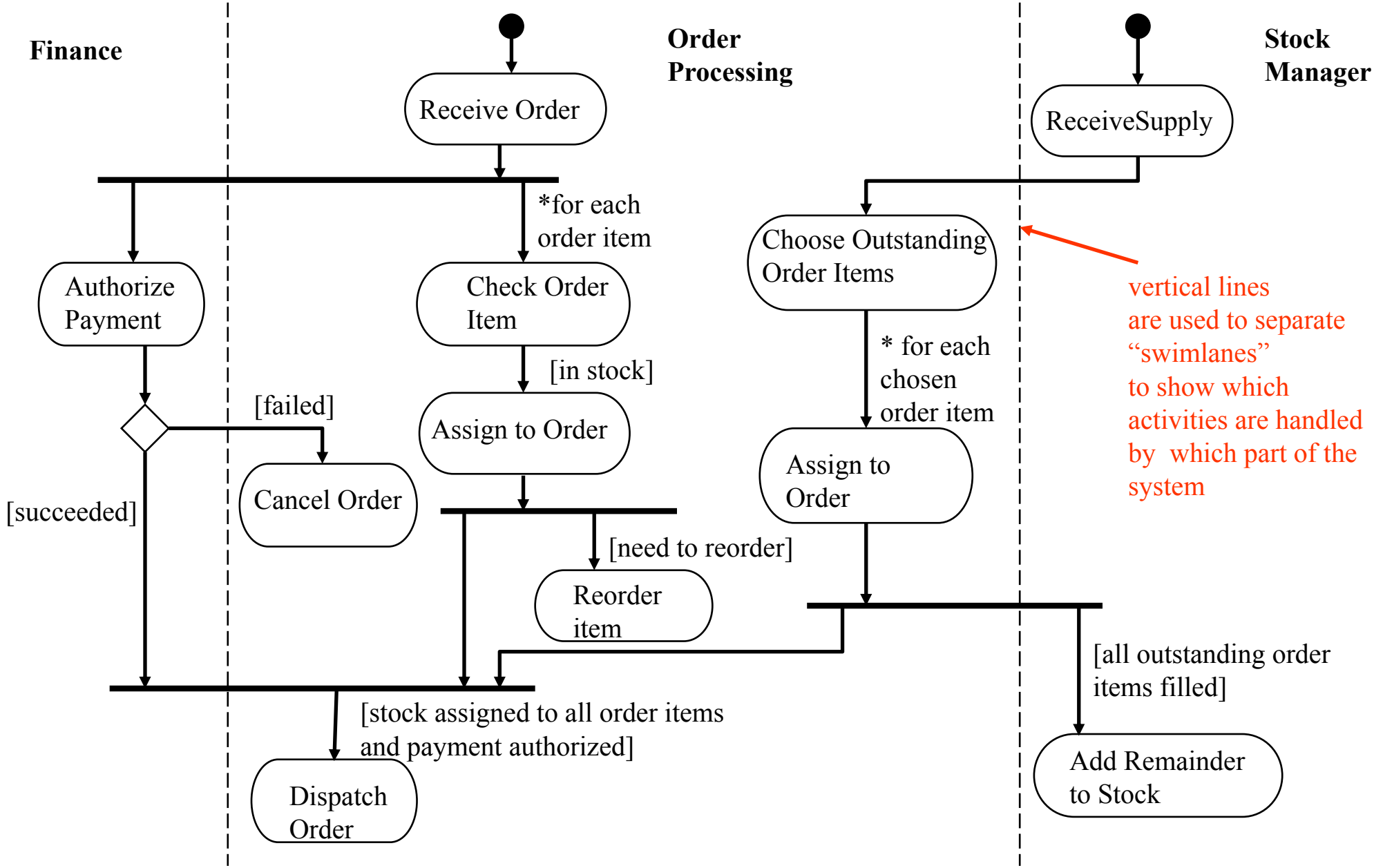


Activity Diagrams

- Activity diagrams show the flow among activities and actions associated with a given object using:
  - activity and actions
  - transitions
  - branches
  - merges
  - forks
  - joins

- Activity diagrams are similar to SDL state diagrams, SDL state diagrams have formal semantics

- Activity diagrams are basically an advanced version of flowcharts


Finance
Order Processing
Stock Manager
Receive Order
ReceiveSupply
*for each order item
Choose Outstanding Order Items
Authorize Payment
Check Order Item
[in stock]
* for each chosen order item
vertical lines are used to separate "swimlanes" to show which activities are handled by which part of the system
[failed]
Assign to Order
Assign to Order
Cancel Order
[succeeded]
[need to reorder]
Reorder item
[all outstanding order items filled]
[stock assigned to all order items and payment authorized]
Add Remainder to Stock
Dispatch Order

UML Diagrams

- Functionality, requirements
  - use case diagrams
- Architecture, modularization, decomposition
  - class diagrams (class structure)
  - component diagrams, package diagrams, deployment diagrams (architecture)
- Behavior
  - state diagrams, activity diagrams
- Communication, interaction
  - sequence diagrams, collaboration diagrams

How do they all fit together?

- Requirements analysis and specification
  - use-cases, use-case diagrams, sequence diagrams
- Design and Implementation
  - Class diagrams can be used for showing the decomposition of the design
  - Activity diagrams can be used to specify behaviors described in use cases
  - State diagrams are used to specify behavior of individual objects
  - Sequence and collaboration diagrams are used to show interaction among different objects
  - Component diagrams, package diagrams and deployment diagrams can be used to show the high level architecture
  - Use cases and sequence diagrams can be used to derive test cases

Object Constraint Language

- Object Constraint Language (OCL) is part of UML

- OCL was developed at IBM by Jos Warmer as a language for business modeling within IBM

- OCL specification is available here:

https://www.omg.org/spec/OCL

- My slides are based on the following text:
  - "The Objection Constraint Language: Precise Modeling with UML", by Jos Warmer and Anneke Kleppe

Object Constraint Language (OCL)

- OCL provides a way to develop more precise models using UML

- What is a constraint in Object Constraint Language?
  - A constraint is a restriction on one or more values of (part of) an object-oriented model or system

Advantages of Constraints

- Better documentation
  - Constraints add information about the model elements and their relationships to the visual models used in UML
  - It is a way of documenting the model
- More precision
  - OCL constraints have formal semantics, hence, can be used to reduce the ambiguity in the UML models
- Communication without misunderstanding
  - UML models are used to communicate between developers
  - Using OCL constraints modelers can communicate unambiguously

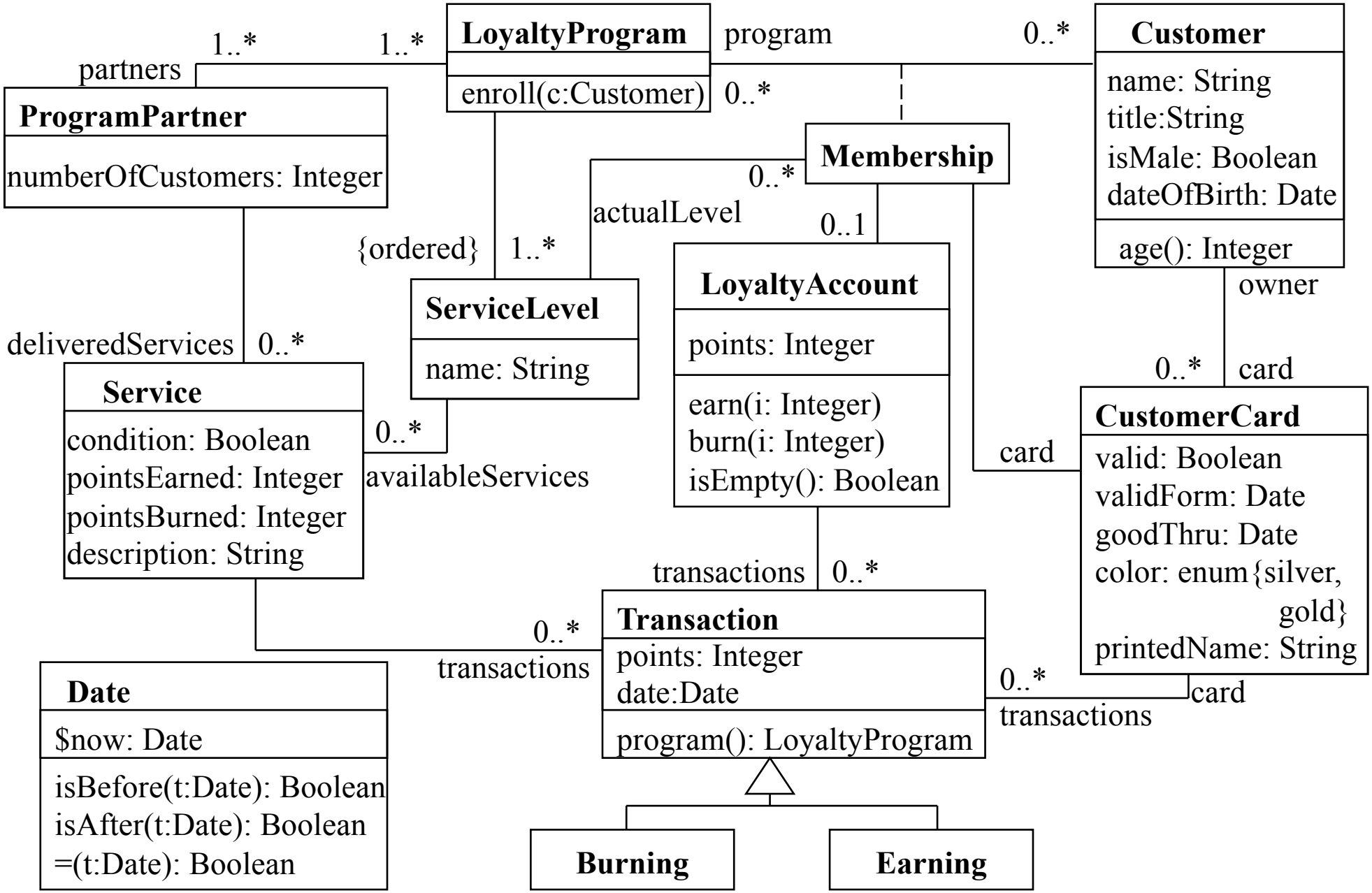
OCL Constraints

- OCL constraints are declarative
  - They specify what must be true not what must be done
- OCL constraints have no side effects
  - Evaluating an OCL expression does not change the state of the system
- OCL constraints have formal syntax and semantics
  - their interpretation is unambiguous

An Example

- Loyalty programs are used by companies to offer their customers bonuses (for example frequent flier miles programs)
- There may be more than one company participating in a loyalty program (each participating company is called a program partner)
- A customer who enters a loyalty program gets a membership card.
- Program partners provide services to customers in their loyalty programs.
- A loyalty program account can be used to save the points accumulated by a customer. Each transaction on a loyalty program account either earns or burns some points.
- Loyalty programs can have multiple service levels

An Example


LoyaltyProgram
enroll(c:Customer)
program
1..*
1..*
partners
0..*
Customer
name: String
title:String
isMale: Boolean
dateOfBirth: Date
age(): Integer
ProgramPartner
numberOfCustomers: Integer
0..*
Membership
actualLevel
0..1
{ordered}
1..*
ServiceLevel
name: String
LoyaltyAccount
points: Integer
earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean
owner
card
0..*
deliveredServices
0..*
Service
condition: Boolean
pointsEarned: Integer
pointsBurned: Integer
description: String
0..*
availableServices
card
CustomerCard
valid: Boolean
validForm: Date
goodThru: Date
color: enum{silver, gold}
printedName: String
transactions
0..*
0..*
transactions
Transaction
points: Integer
date:Date
program(): LoyaltyProgram
0..*
transactions
card
Date
$now: Date
isBefore(t:Date): Boolean
isAfter(t:Date): Boolean
=(t:Date): Boolean
Burning
Earning

Types and Instances

- OCL types are divided into following groups
  - Predefined types
    - Basic types: String, Integer, Real, Boolean
    - Collection types: Collection, Set, Bag, Sequence
  - User-defined model types
    - User defined classes such as Customer, Date, LoyaltyProgram

Operations on Boolean Type

- Boolean operators that result in boolean values

  `a or b, a and b, a xor b, not a, a = b,`
  `a <> b` (not equal), `a implies b`

- Another operator that takes a boolean argument is

  `if b then e1 else e2 endif`

  where `b` is Boolean and the result type of the expression is the type of `e1` if `b` is true or the type of `e2` if `b` is false

```
Customer
title = (if isMale = true
            then 'Mr.'
            else 'Ms.'
         endif)
```

Operations on Integer and Real Types

- Operation on Real and Integer with Boolean result type

```
a = b, a <> b , a < b, a > b, a <= b, a >= b
```

- Operations on Real and Integer types with result type Real or Integer

```
a + b, a – b, a * b, a / b, a.abs, a.max(b),
  a.min(b)
```

- Operations on Real and Integer types with result type Integer

```
a.mod(b), a.div(b), a.round, a.floor
```

Operations on String Type

- Operations on String type with result type Boolean

  `s1 = s2, s1 <> s2`

- Operations on String type with result type String

  `s1.concat(s2), s1.toLower, s1.toUpper,`

  `s1.substring(int1, int2)` returns the substring that starts at character int1 up to and including character int2

- Operations on String type with result type Integer

  `s1.size`

Model Types

- Model types are classes, subclasses, association classes, interfaces, etc. defined in the model

- Properties of a model type are
  - attributes
  - operations and methods
  - navigations that are derived from the associations
  - enumerations defined as attribute types

- Properties of a model type can be referenced in OCL expressions

OCL expressions and constraints

- Each OCL expression has a result
  - the value that results by evaluating the expression
- The type of an OCL expression is the type of the result value
  - either a predefined type or a model type
- An OCL constraint is an OCL expression of type Boolean

Invariants

- Using OCL we can specify class invariants such as

```
Customer
age >= 18
```

- As a convention we will write the OCL expressions in the following form:

```
OCLcontext
OCLexpression
```

- The class on which the invariant must hold is the invariant context
  - Invariant has to hold for all instances of the class
- For the above example, the expression `age >= 18` is an invariant of the `Customer` class, i.e. it holds for every instance of that class

Invariants

- We can also write invariants on attributes of associated classes
- In OCL you can use the rolename to refer to the object on the other end of an association.
  - If the rolename is not present, you can use the classname starting with a lowercase letter
- Examples:

```
Membership
card.owner = customer

CustomerCard
printedName = owner.title.concat( owner.name )
```

Choosing a Context

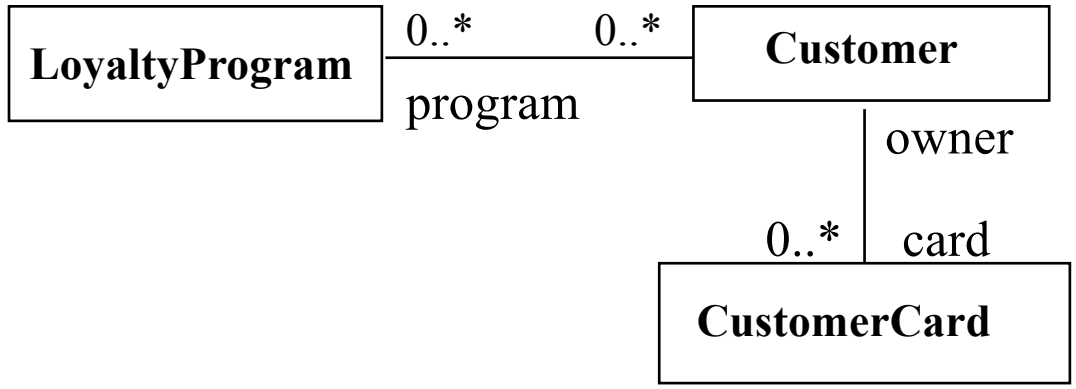
- The class on which the invariant must be put is the invariant context
- One can write the same invariant property in different contexts
- For example

```
Customer
age >= 18

LoyaltyProgram
customer.forAll( age >= 18 )
```

Navigating Associations

- Navigation starting from CustomerCard




```
CustomerCard
owner.program  ...
```

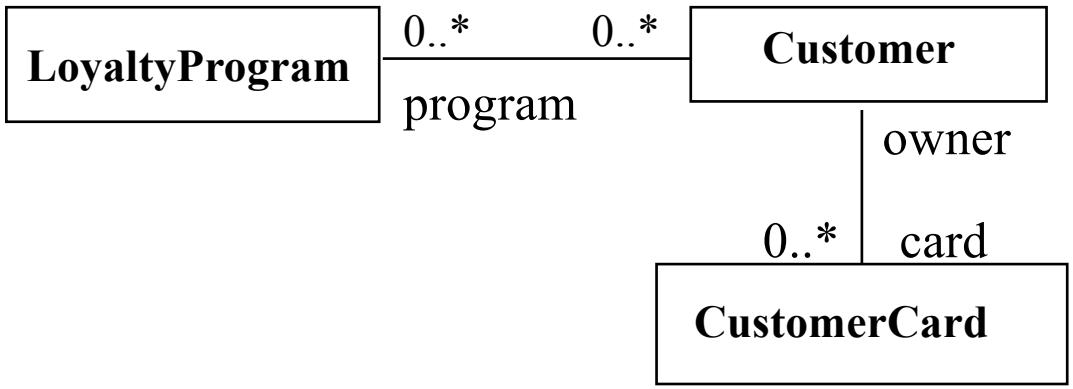
The `owner` attribute of the `CustomerCard` instance (which is an instance of the `Customer` class)

The `program` attribute of the `owner` attribute (which will be instances of the `LoyaltyProgram` class)

You can also say:

```
CustomerCard
self.owner.program  ...
```

Multiplicity and Navigation

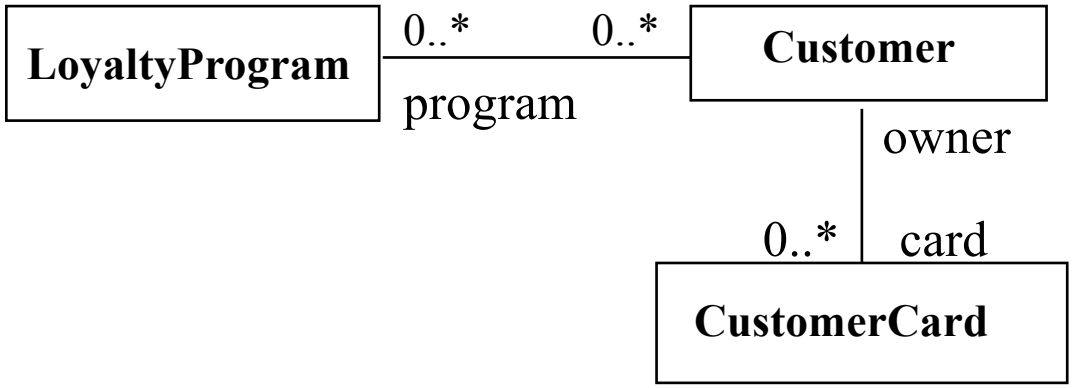


- Default multiplicity in UML is 1
  - Hence, each `CustomerCard` instance has exactly one `owner` and navigating from `CustomerCard` class to `Customer` class through the `owner` attribute results in a single instance of `Customer` class

```
CustomerCard
printedName = owner.title.concat( owner.name )
```

A single instance of `Customer`

Multiplicity and Navigation



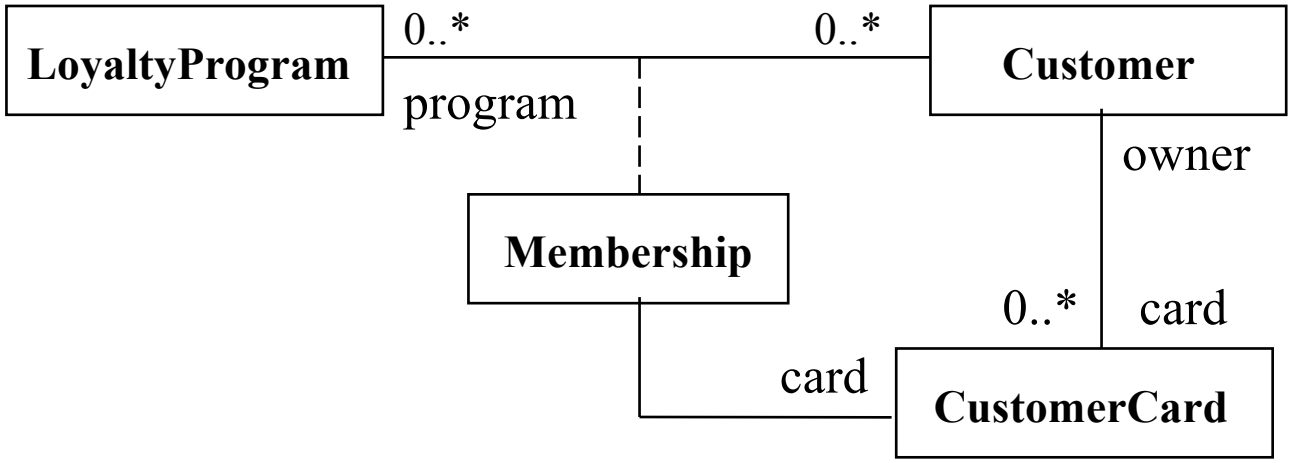
- If the multiplicity is greater than 1, navigation results in a collection of values
  - Navigating from `Customer` class to `LoyaltyProgram` class through the `program` attribute results in a set of instances of the `LoyaltyProgram` class

```
Customer
program->size <= 10
```

A set of instances of the `LoyaltyProgram` class

Equivalently, this constraint can be specified as a multiplicity constraint by changing the multiplicity of the association between `LoyaltyProgram` and `Customer` from `0..*` to `0..10` on the `LoyaltyProgram` side

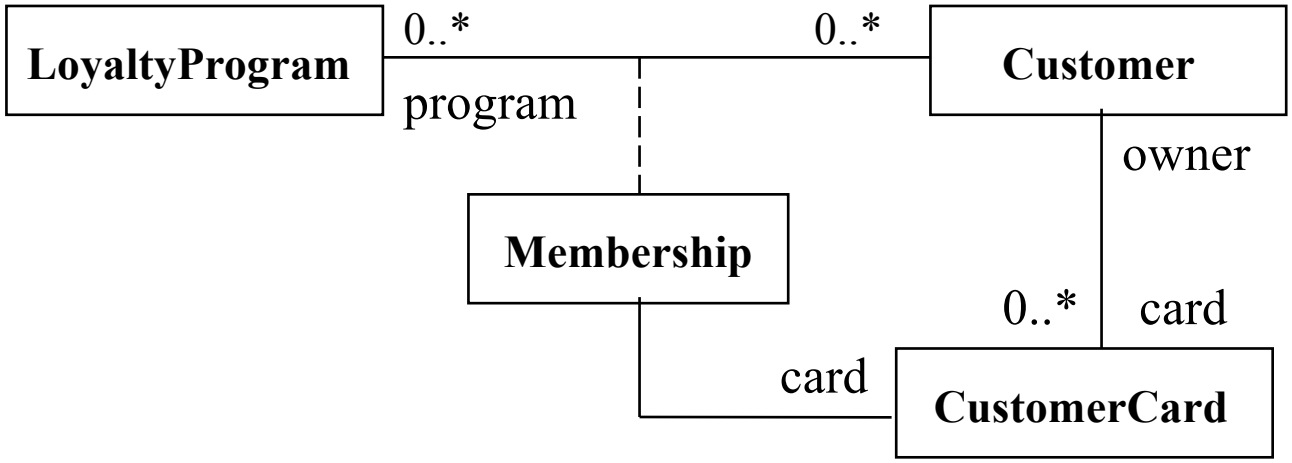
Multiplicity and Navigation



- According to UML semantics an instance of an association class is associated with one instance of the classes in each side of the association
  - Hence, each instance of `Membership` is associated with one instance of `Customer` and one instance of `LoyaltyProgram`
  - If we navigate from `Membership` class to `Customer` we would get a single instance of `Customer` class

A single instance of `CustomerCard` → `Membership` `card.owner = customer` ← A single instance of `Customer`

Multiplicity and Navigation



- However, one class can be associated with multiple instances of an association class
  - Hence, navigating from `Customer` to `Membership` results in a set of instances of the `Membership` class.

A set of instances of `Membership` class

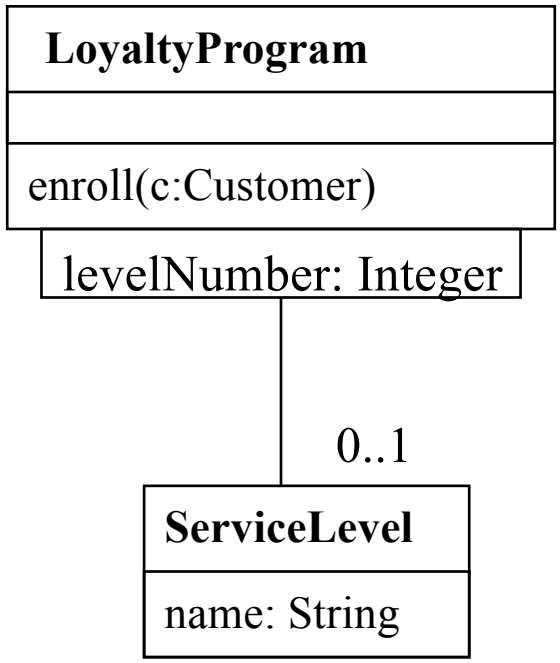
```
Customer
membership.program = program
```

Qualified Associations

- You can navigate qualified associations by providing an index to the qualified association using a qualifier

```
object.navigation[qualifierValue]
```

  - If there are multiple qualifiers their values are separated using commas



- Example

```
LoyaltyProgram
serviceLevel[1].name = 'basic'

LoyaltyProgram
serviceLevel-> exists(name = 'basic')
```

Collections

- Often the multiplicity of an association is greater than 1, linking one object to a set of objects of the associated class
  - In such a scenario navigating through the association results in a collection of objects from the association class

- OCL has a number of collection operations to write expressions in such situations

- Whenever the link in a constraint results in a set of objects, you can use one of the collection operations by putting an arrow between the rolename and the operation

Collections

- The size of a collection can be expressed as:

```
LoyaltyProgram
serviceLevel->size = 2
```

- You can also select a subset of a list by passing an OCL expression as an argument to the select operation:

```
Customer
program->size = cards->select( valid = true )->size
```

Flattening of Collections

- In OCL collection types are automatically flattened
  - Whenever a collection is inserted into another collection, the resulting collection is automatically flattened; the elements of the inserted collection become elements of the resulting collection
  - for example, there is no “set of sets of integers” it is always just a “set of integers” or a “bag of integers”

- Example: in OCL

we do not have:

```
Set{ Set{ 1, 2, 3 } Set{ 3,  4} }
```

instead we get:

```
Bag{ 1, 2, 3, 3, 4 }
```

Flattening of a set of sets results in a Bag.

Collections

- You can also use `forAll` operation to evaluate a condition on a collection
- `forAll` operation takes an OCL expression as an argument, and returns a boolean value
- If the argument of forAll operation evaluates to true for all members of the collection, then the result of the forAll operation is true, otherwise it is false
- Example

```
LoyaltyProgram
partners.deliveredServices->forAll(
            pointsEarned = 0 and pointsBurned = 0 )
    implies membership.loyaltyAccount->isEmpty
```

If there is no way to earn or burn points then there should be no loyalty accounts

Collections: Sets, Bags and Sequences

- There are different types of collections
  - Set
    - In a set, each element may occur only once
  - Bag
    - In a bag, elements may be present more than once
  - Sequence
    - Sequence is a bag in which elements are ordered
    - When you navigate an association marked `{ordered}` then the resulting collection is a sequence

Sets vs. Bags

- Consider the expression

```
ProgramPartner
numberOfCustomers = loyaltyProgram.customer->size
```

- This expressions is not correct since a customer can participate in more than one program
- In OCL, the rule is:
  - If you navigate through more than one associations with multiplicity greater than one, you get bags.
  - If you navigate through only one association with multiplicity greater than one you get a set.

- The correct constraint is:

```
ProgramPartner
numberOfCustomers = loyaltyProgram.customer->asSet->size
```

Operations on Collection Types

- The following operations have different meanings for Sets, Bags and Sequences: `=, union, intersection, including, excluding`

`including(object)` adds one element to the collection (for a set the element is added if it is not in the set)

`excluding(object)` removes one element from the collection (from a bag or sequence it removes all occurrences)

`size` number of elements in the collection

`count(object)` number of occurrences of the object in the collection

`includes(object)` True if the object is an element of the collection

`includesAll(collection)` True if all elements of the parameter collection are members of the collection

`isEmpty`

`notEmpty`

Operations on Collection Types

- Operations on collection types

`iterate(Expression)` The expression is evaluated for every element in the collection. The result type depends on the expression.

`sum()` The addition of all the elements in the collection. The elements in the collection must support addition operation

`exists(expression)` True is expression is true for at least one element in the collection (expression must be a boolean expression)

`forAll(expression)` True if for all elements expression is true (expression must be a boolean expression)

Operations on Collection Types

- Operations on Sets

```
s1->minus(s2)
```

- Operations on Sequences

```
s->first
s->last
s->at(i)
s1->append(s2)
s1->prepend(s2)
```

Example:

LoyaltyProgram

```
serviceLevel->first.name = 'Silver'
```

Select Operation

- The result of the select operation is the collection that contains all elements for which the boolean expression (that is the parameter of the select operation) evaluates to true
- The result of the select operation is a subset of the original collection

```
CustomerCard
self.transactions->select( points > 100 )
```

- General syntax is

```
collection->select( element: Type | expression )
```

- Example

```
Customer
membership.loyaltyAccount->select( a: LoyaltyAccount |
                                   a.points > 0 )
```

Reject Operation

- Reject operation is used to remove elements from a collection
- Returns the collection that contains all elements for which the boolean expression (that is the parameter of the select operation) evaluates to false

- Following two expressions are equivalent

```
CustomerCard
self.transactions->select( points > 100 )

CustomerCard
self.transactions->reject( not (points > 100) )
```

Collect Operation

- Collect operation operates over a collection, computes an expression for each member of the collection and gathers the results in a new collection

```
LoyaltyAccount
transactions->collect(points)
```

- The result of a collection operation on a Set or a Bag is a Bag and on a Sequence is a Sequence

- General syntax is

```
collection->collect( element: Type | expression )
```

Writing Pre and Postconditions

- OCL supports Design by Contract style constraints

- One can specify the pre and postcondition of an operation of a class using OCL expressions

```
Type1::operation(arg: Type2) : ReturnType
pre: arg.attr = true
post: result = arg.attr xor self.attribute2
```

- For example

```
LoyaltyAccount::isEmpty()
pre: -- none
post: result = (points = 0)
```

Constructs for Postconditions

- One can refer to the value of an attribute at the beginning operation in the postcondition using the `@pre` syntax

```
LoyaltyProgram::enroll(c: Customer)
pre: not customer->includes(c)
post: customer = customer@pre->including(c)
```

- You can refer to the return value of the method using the `result` keyword

```
LoyaltyAccount::isEmpty()
pre: -- none
post: result = (points = 0)
```

Iterate Operation

- The `select`, `reject`, `collect`, `forAll` and `exists` operations can all be described as a special case of iterate operation
- The syntax is

```
collection->iterate(
    element : Type1;
    result : Type2 = <expression>
    | <expression-with-element-and-result>)
```

- The `element` is the iterator, the resulting value is accumulated in the variable `result` (called accumulator). The accumulator is initialized to the `<expression>`
- The result of the iterate operation is a value accumulated by iterating over all elements in a collection.

Iterate Operation

- Iterate operation corresponds to the following pseudo code

```
result = <expression>;
while ( collection.notEmpty() do
  element = collection.nextElement();
  result = <expression-with-element-and-result>;
endwhile
return result;
```

A Class Invariant Using the Iterate Operation

```
ProgramPartner
self.services.transaction->iterate(
  t: Transaction;
  result : Integer = 0 |
  if t.oclType = Burning then
     result + t.points
  else
     result
  endif
)
<=
    self.services.transaction->iterate(
      t: Transaction;
      result : Integer = 0 |
      if t.oclType = Earning then
         result + t.points
      else
         result
      endif
    )
```

This constraint states that given a program partner, the total points for all burning transactions of all the delivered services should be less than or equal to total points of all earning transactions of all the services

Enumerations

- Values of an enumerated variable can be written as #valuename

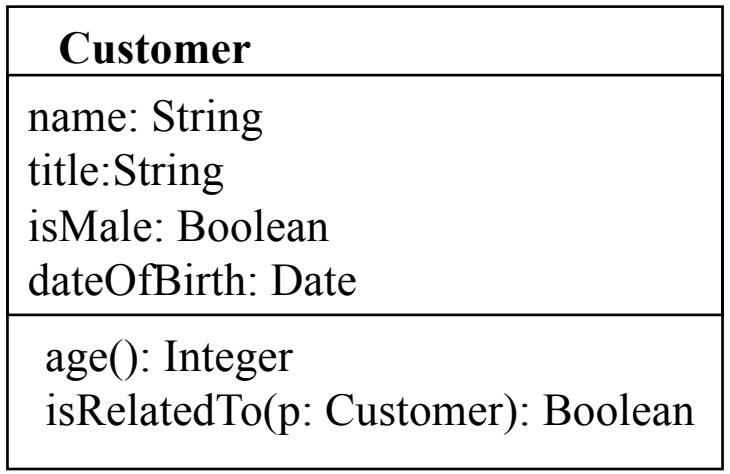
- For example

```
Membership
actualLevel.name = 'Silver' implies card.color = #silver
and actualLevel.name = 'Gold' implies card.color = #gold
```

Using Operations of a Model Type in OCL

- The operations that are defined on the UML model types can be used in OCL
  - Only query operations can be used in OCL
  - Query operations are operations which return a value but do not change the state of the object

- For example




```
Customer
age() >= 18

Customer
self.isRelatedTo(self) = true

Customer
self.name='Joe Senior' implies
                    self.age() > 21
```

Classes and Subclasses

- Consider the following constraint used to limit the number of points given by each partner

```
LoyaltyProgram
partners.deliveredServices.transactions.points->sum < 10,000
```

- However, this constraint is incorrect since it does not distinguish burning and earning transactions.

- To determine the subclass of an element that belongs to a collection, we can use the operation `oclType` and fix the above constraint as follows:

```
LoyaltyProgram
partners.deliveredServices.transactions
                ->select(oclType = Burning)
                  ->collect( points )->sum < 10,000
```

Operations Defined for every OCL Type

- Following operations return a Boolean value

```
o1 = o2, o1 <> o2
```

- Following operation returns true only if the type of the object is identical to the argument type

```
o.oclIsTypeOf(type: OclType)
```

- Following operation returns true only if the type of the object is identical to the argument type or identical to any of the subtypes of the argument type

```
o.oclIsKindOf(type: OclType)
```

- Following operation returns the type of the object

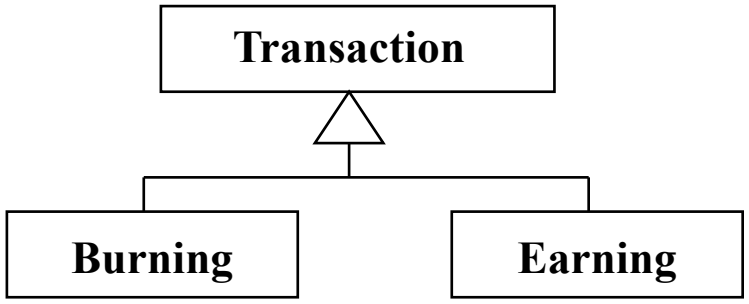
```
o.oclType
```

- Following operation returns the same object but changes its type to the argument type

```
o.oclAsType(type: OclType)
```

Example

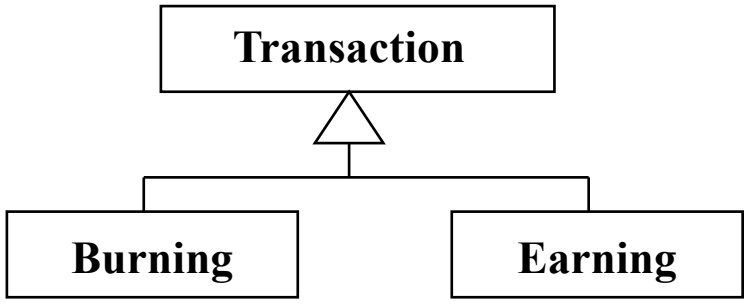
- Given the class structure shown in the diagram, following invariants for the Transaction class will hold




```
Transaction
self.oclType = Transaction
self.oclIsKindOf(Transaction) = true
self.oclIsTypeOf(Transaction) = true
self.oclIsTypeOf(Burning) = false
self.oclIsTypeOf(Earning) = false
self.oclIsKindOf(Burning) = false
self.oclIsKindOf(Earning) = false
```

Example

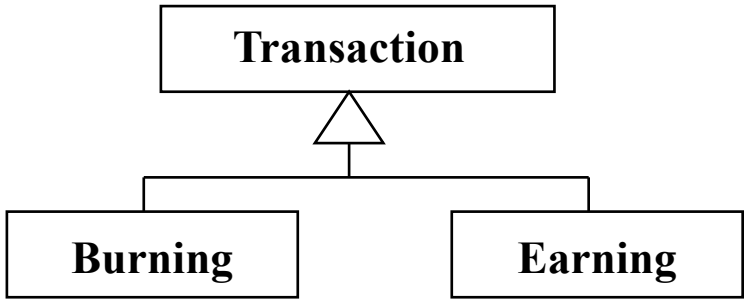
- Given the class structure shown in the diagram, following invariants for the Burning class will hold




```
Burning
self.oclType = Burning
self.oclIsKindOf(Transaction) = true
self.oclIsTypeOf(Transaction) = false
self.oclIsTypeOf(Burning) = true
self.oclIsKindOf(Burning) = true
self.oclIsTypeOf(Earning) = false
self.oclIsKindOf(Earning) = false
```

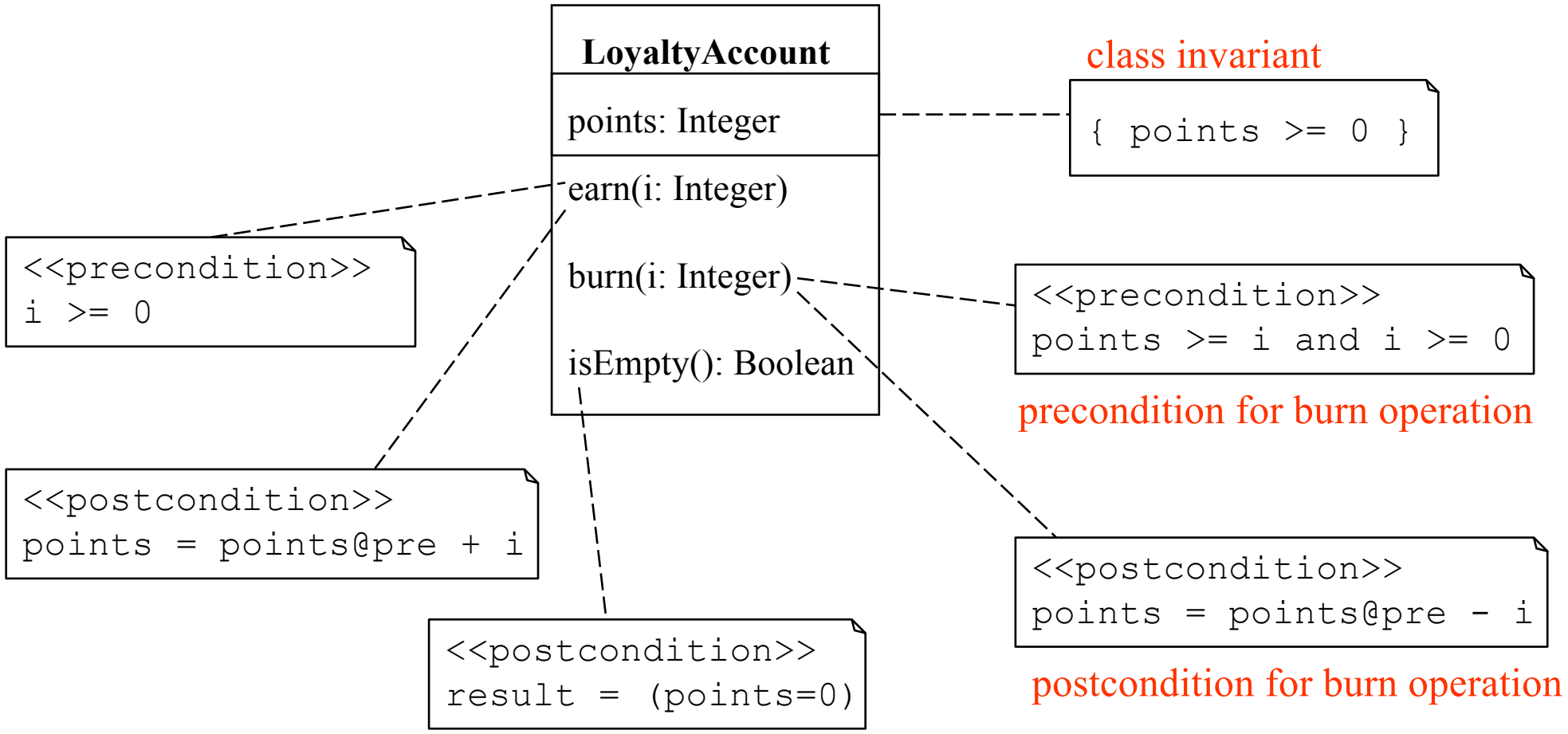
Example

- Given the class structure shown in the diagram, following invariants for the Earning class will hold



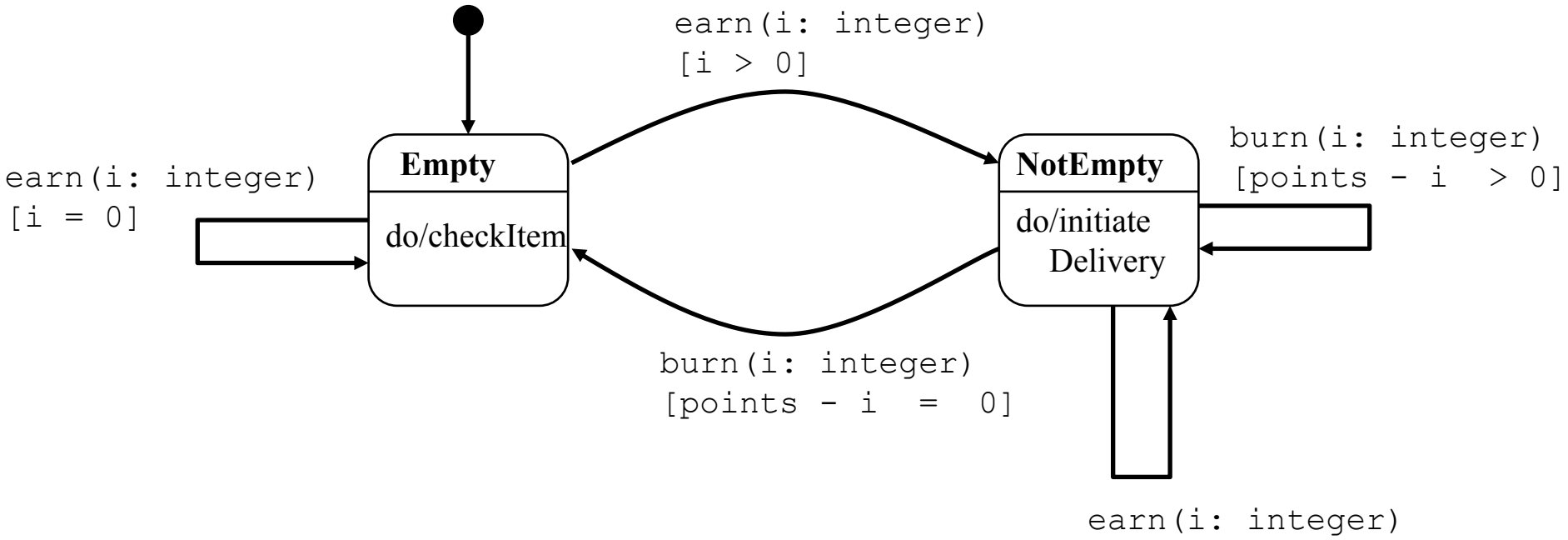

```
Earning
self.oclType = Earning
self.oclIsKindOf(Transaction) = true
self.oclIsTypeOf(Transaction) = false
self.oclIsTypeOf(Burning) = false
self.oclIsKindOf(Burning) = false
self.oclIsTypeOf(Earning) = true
self.oclIsKindOf(Earning) = true
```

Using OCL in Class Diagrams



Using OCL in State Diagrams

State Transition Diagram for LoyaltyAccount



USE: A Tool for Validating OCL Specifications

- USE (UML Specification Environment)

available at: https://sourceforge.net/projects/useocl/

- A tool for validating OCL specifications
  - The developer can create test cases and check if the specified constraints are satisfied for these test cases
  - USE checks the test cases with respect to invariants and pre-post-conditions
- There are special USE commands for creating and manipulating object diagrams that can be accumulated in command files
- There is some support for automated testing
  - USE has a snapshot sequence language and a snapshot generator
  - The snapshot sequence language enables the user to write high level implementations for the user defined operations, so that they can be tested