Introduction to Software Technique
3. Just Enough UML

Klaus Ostermann

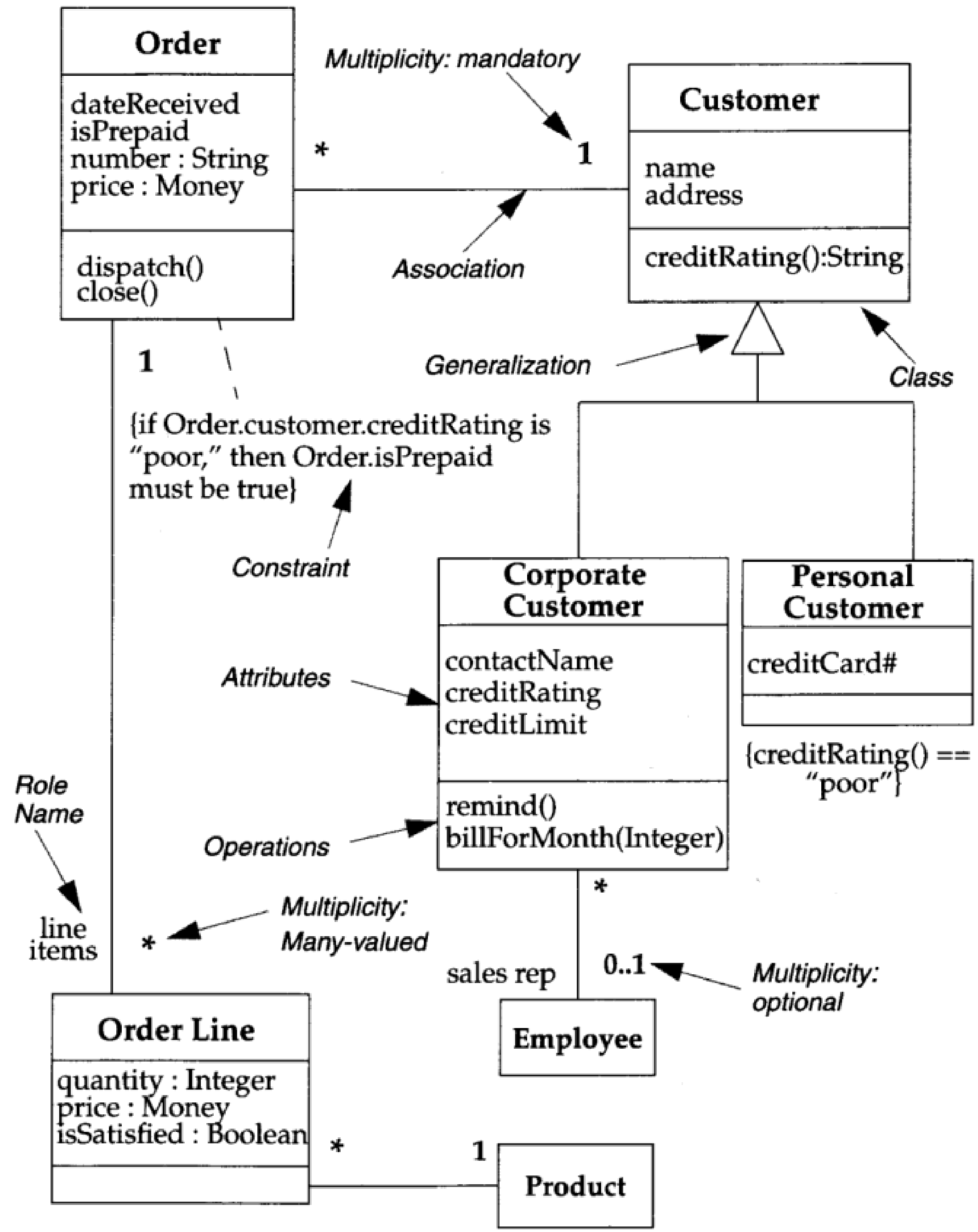
Just Enough UML…

- The UML is the Unified Modeling Language
  - Successor to a wave of OO analysis & design methods that appeared in the 1980s and 1990s
- It is a modeling language to express high-level design
  - Defines several diagram types
- Implicitly associated with the UML is also a method or process
  - Method: advice on what steps to take in doing a design
- There are different ways to use UML. We will mainly use it as a notation to communicate high-level OO design ideas.
- But keep in mind: No user is going to thank you for pretty pictures; what a user wants is software that executes

Class Diagrams

- A class diagram describes the types of objects in a system and the various kinds of static relationships between them
  - Associations
  - Subtypes
- Class diagrams also show the attributes, names/types of operations, and constraints that restrict how objects are connected

Class Diagrams Example



Three ways to use class diagrams

- **Conceptual**: Draw a diagram that represents the concepts in the domain under study
  - Little or no regard for the software that might implement it
- **Specification**: Describing the interfaces of the software, not the implementation
  - Often confused in OO since classes combine both interfaces and implementation
- **Implementation**: Diagram describes actual implementation classes
- Understanding the intended perspective is crucial to drawing and reading class diagrams
  - Even though the lines between them are not sharp

Associations

- Associations represent relationships between instances of classes
- Conceptual perspective: Associations represent conceptual relationships
- Specification perspective: Associations represent responsibilities
- Implementation perspective: Associations represent pointers/fields between related classes

Associations

- Each association has two ends
  - Each end can be named with a label called role name
  - An end also has a multiplicity: How many objects participate in the given relationship
    - General case: give upper and lower bound in lower..upper notation
    - Abbreviations: * = 0..infinity, 1 = 1..1
    - Most common multiplicities: 1, *, 0..1
- In the specification perspective, one can infer existence and names (if naming conventions exist) of methods to navigate the associations, for example:

```
Class Order {
  public Customer getCustomer();
  public Set<OrderLine> getOrderLines();
 …
}
```

Associations

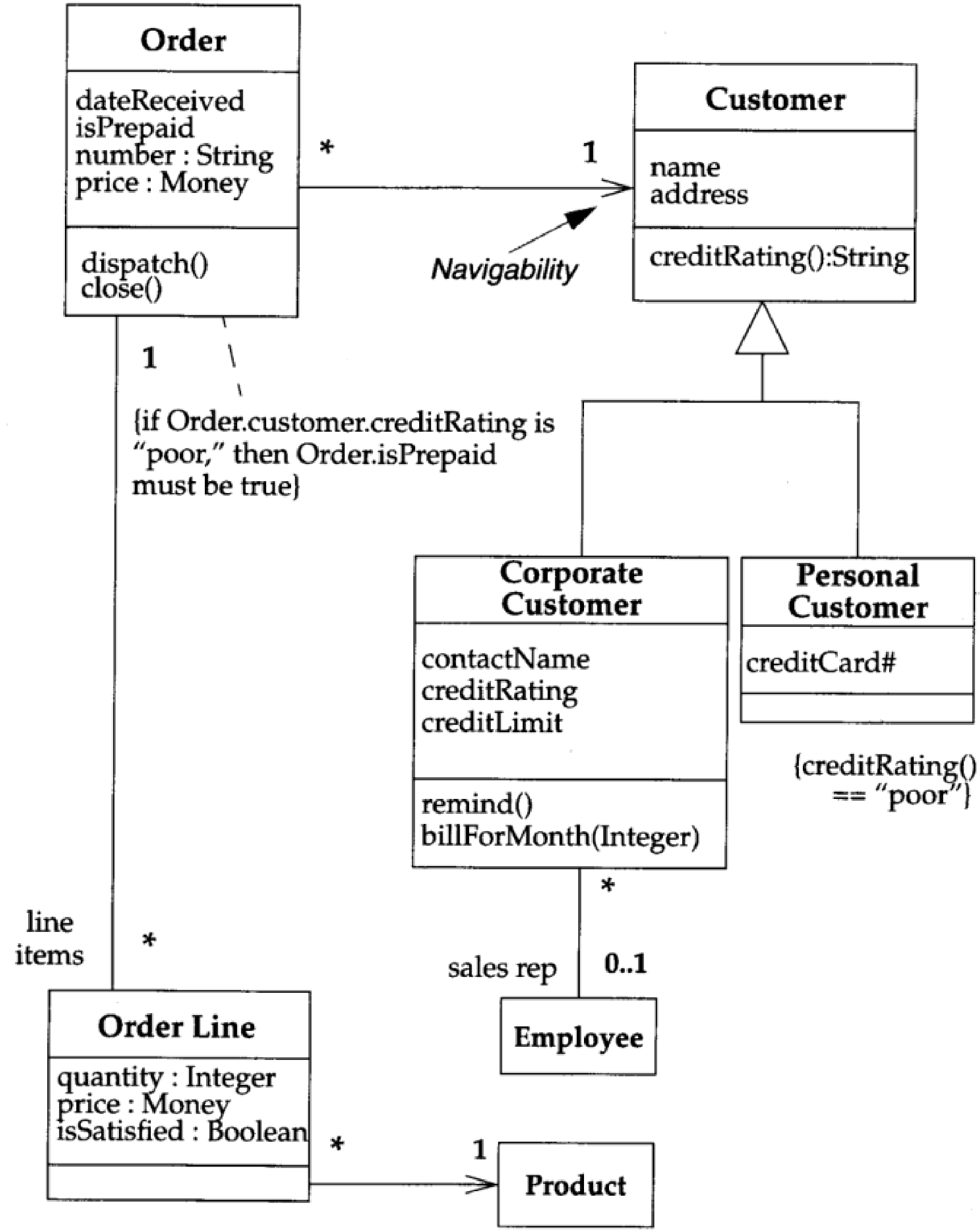
- In the implementation perspective we can conclude existence of pointers in both directions between related classes

```
class Order {
  private Customer _ customer;
  private Set<OrderLine> _orderLines;
  …
}
class Customer {
  private Set<Order> orders;
  …
}
```

Associations
Unidirectional vs bidirectional

- Arrows in association lines indicate navigability
  - Only one arrow: unidirectional association
  - No or two arrows: bidirectional association
- Specification perspective: Indicates navigation operations in interfaces
- Implementation perspective: Indicates which objects contain the pointers to the other objects
- Arrows serve no useful purpose in conceptual perspective
- For bidirectional associations, the two navigations must be inverses of each other

Unidirectional Associations



Class Diagrams: Attributes

- Attributes are very similar to associations
  - Conceptual level: A customer's name attribute indicates that customers have names
  - Specification level: Attribute indicates that a customer object can tell you its name
  - Implementation level: customer has a field (aka instance variable) for its name
  - UML syntax for attributes:
    visibility name : type = defaultValue
    - Details may be omitted

Class Diagrams: Attributes vs Associations

- Attributes can describe non-object-oriented data
  - Integers, strings, booleans, …
- From conceptual perspective this is the only difference
- Specification and implementation perspective:
  - Attributes imply navigability from type to attribute only
  - Implied that type contains solely its own copy of the attribute objects

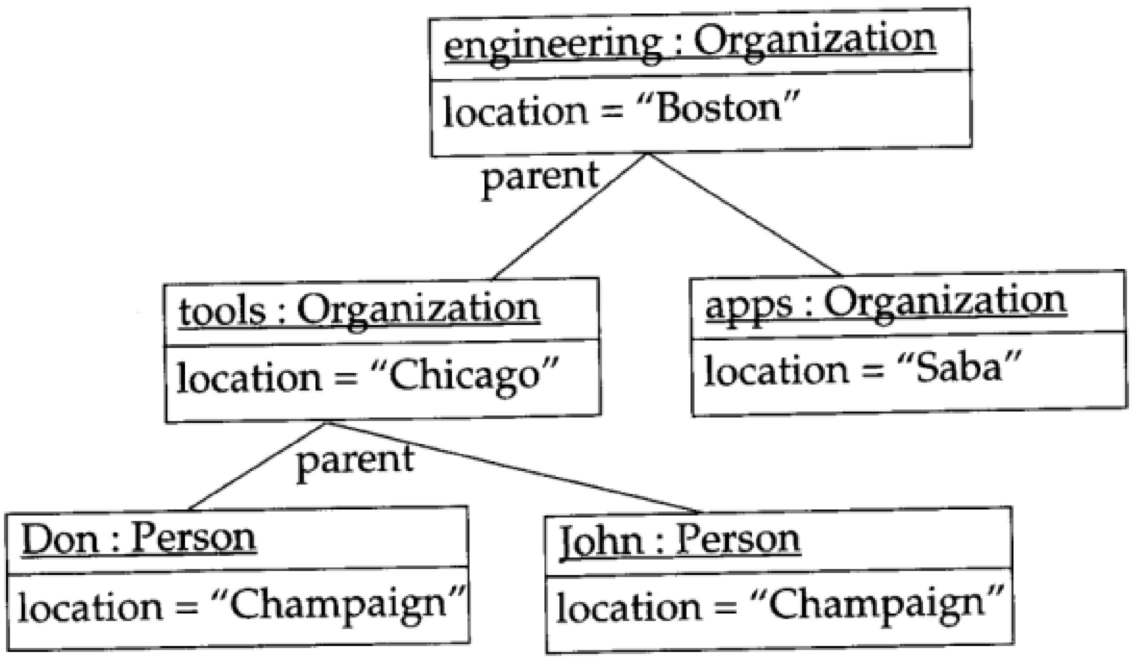
Class Diagrams: Operations

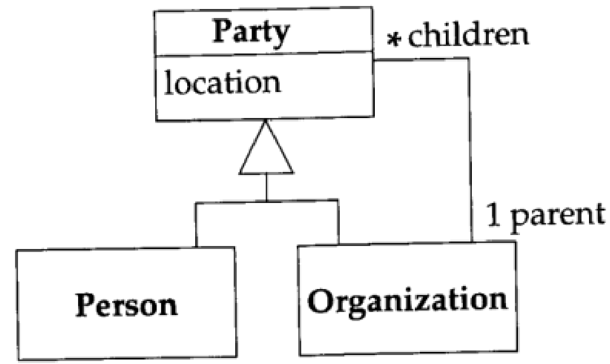
- Operations are the processes that a class knows to carry out
- Most obviously correspond to methods on a class
- Full syntax:
  visibility name(parameter-list) : return-type
  - *visibility* is + (public), # (protected), or - (private)
  - *name* is a string
  - *parameter-list* contains comma-separated parameters whose syntax is similar to that for attributes
    - Can also specificy direction: input (in), output(out), or both (inout)
    - Default: in
  - *return-type* is comma-separated list of return types (usually only one)

Class Diagrams: Constraint Rules

- Arbitrary constraints can be added by putting them inside braces({})
- Mostly formulated in informal natural language
- UML also provides a formal Object Constraint Language (OCL)
- Constraints should be implemented as assertions in your programming language

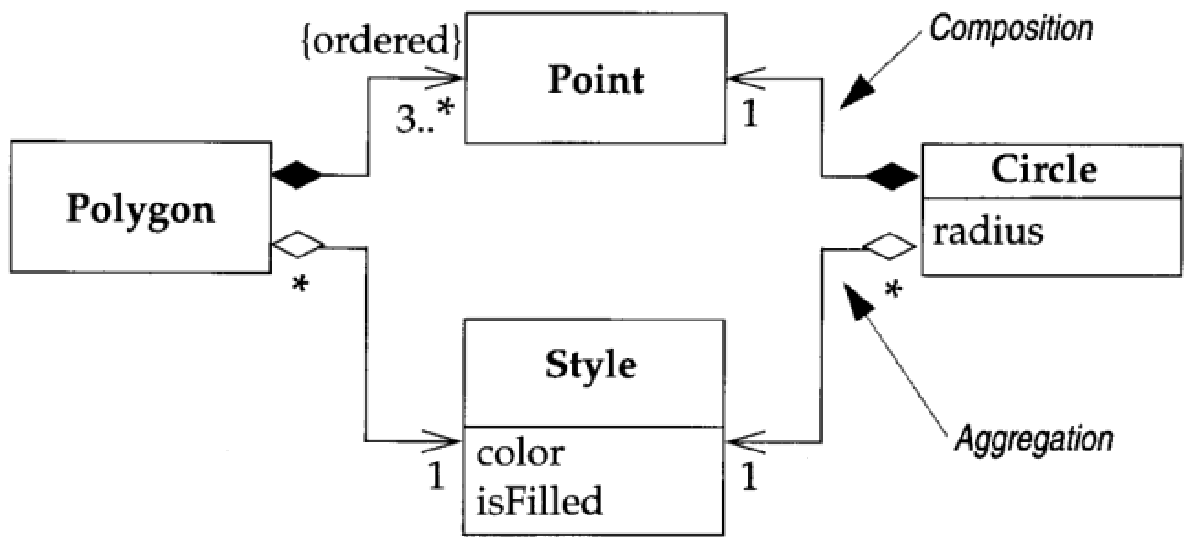
Object Diagrams





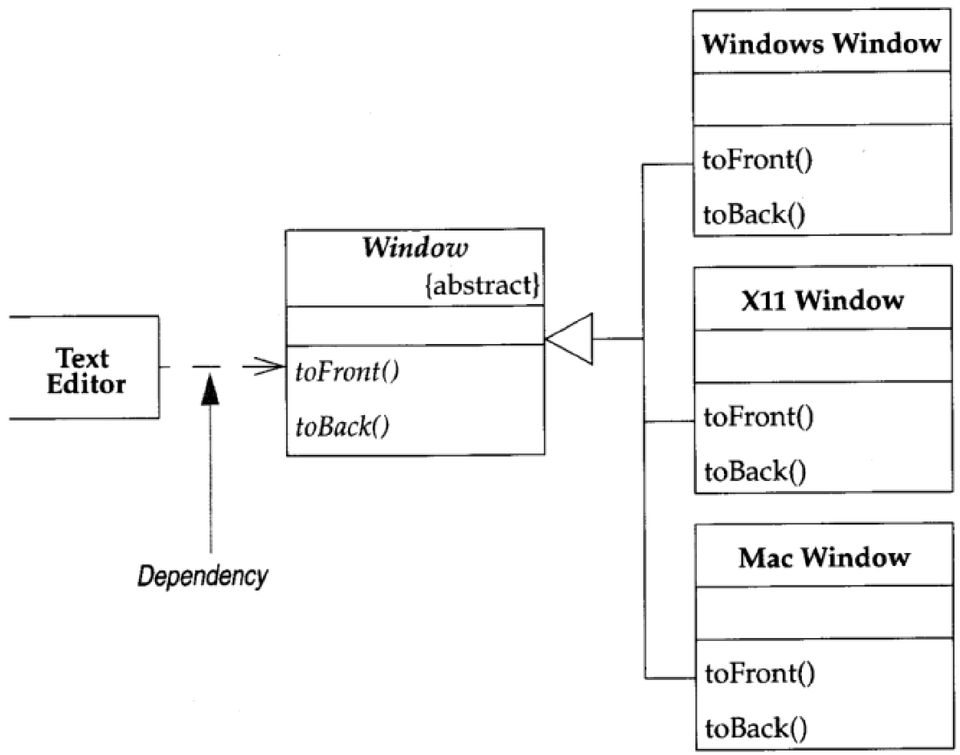
(Class diagram that belongs to the object diagram)

Aggregation vs Composition



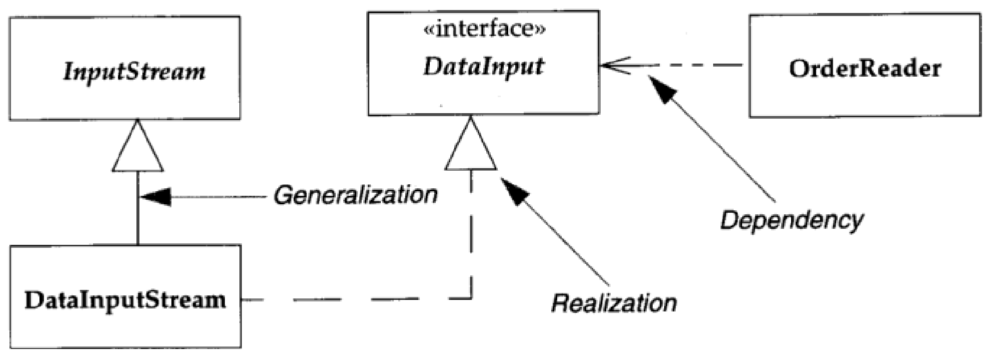
- Aggregation expresses “part-of” relationships, but rather vague semantics
- Composition is stronger: Part object live and die with the whole

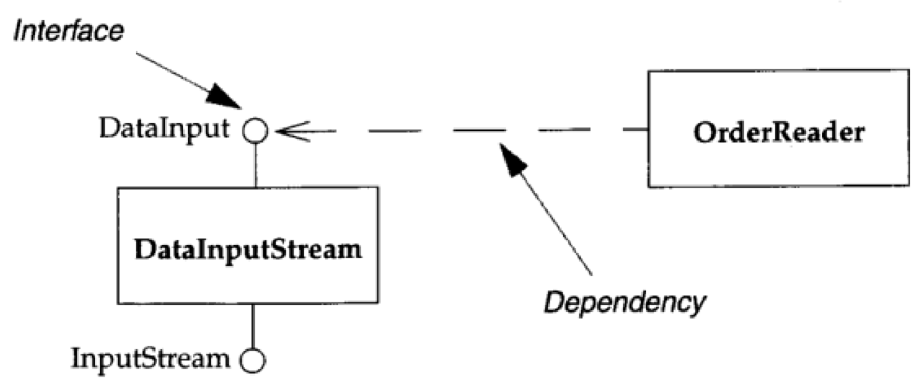
Abstract classes and methods



- UML convention for abstract classes/methods: Italicize name of abstract item or use {abstract} constraint

Interfaces and Lollipop notation

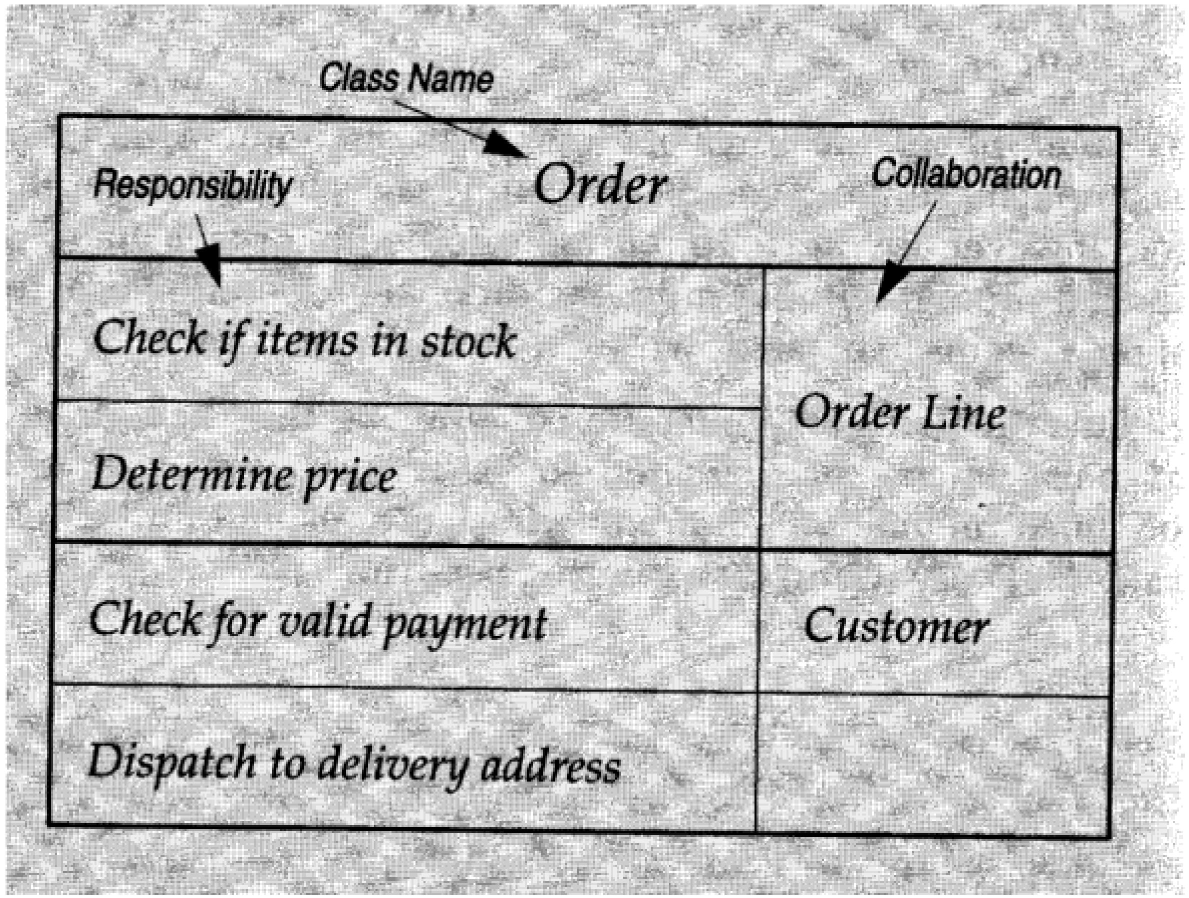




CRC cards

- CRC = Class-Responsibility-Collaboration
- Invented by Ward Cunningham and Kent Beck in the 1980s to ease the development of a class model from the requirements
- Not part of UML, but have proven to be quite useful
- More information: http://c2.com/doc/oopsla89/paper.html

Sample CRC card



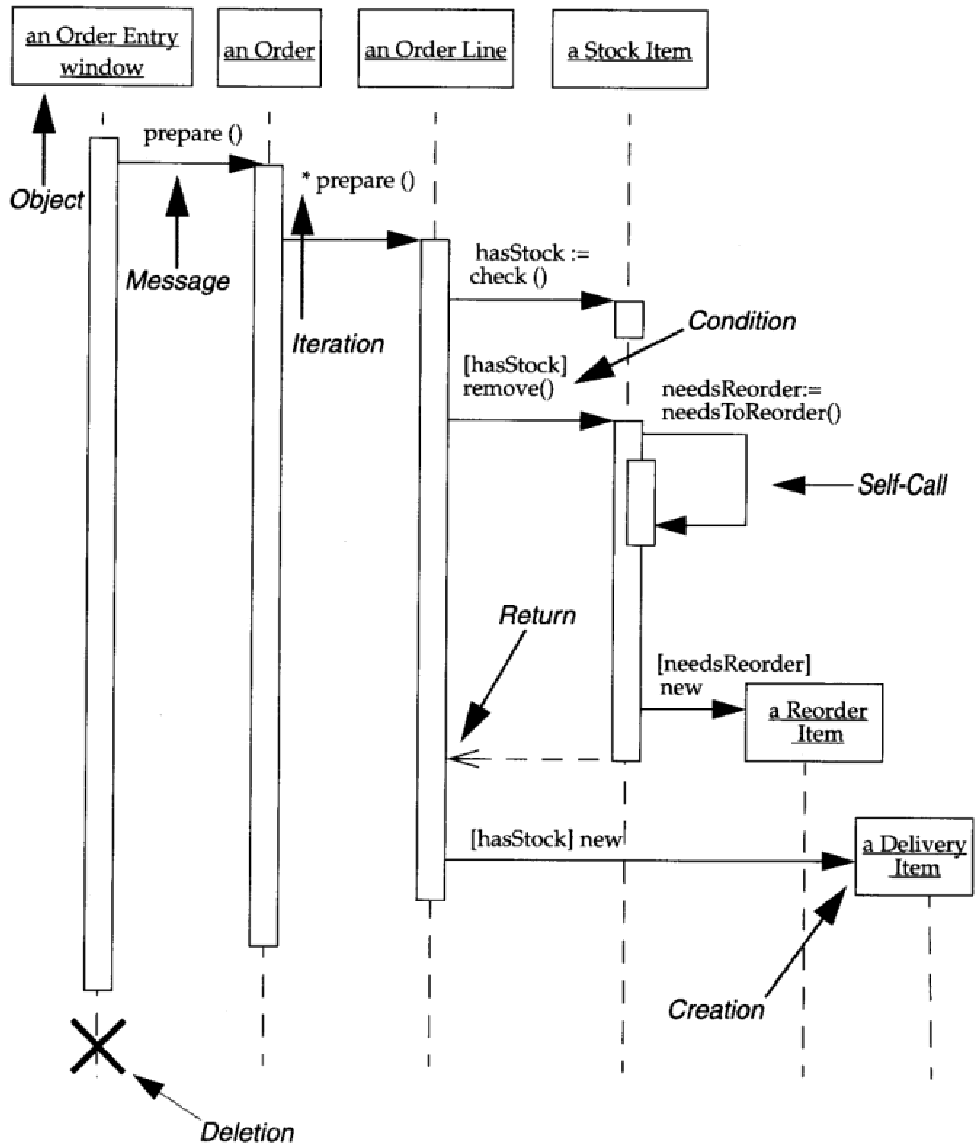
CRC Cards

- Idea: Describe responsibilities and collaboration of each class on an index card ("Karteikarte")
- Motivation: Capture purpose of class in a few sentences without thinking about data, processes, and other implementation details
- Chief benefit of CRC cards: They encourage discussion among developers
- Common mistake: Long lists of low-level responsibilities
  - Responsibilities should fit conveniently on an index card
  - Otherwise consider to split the class or summarize low-level responsibilities in higher-level responsibilities

Interaction Diagrams

- Interaction diagrams describe how groups of objects collaborate in some behavior
- Two kinds of interaction diagrams: **sequence diagrams** and **collaboration diagrams**

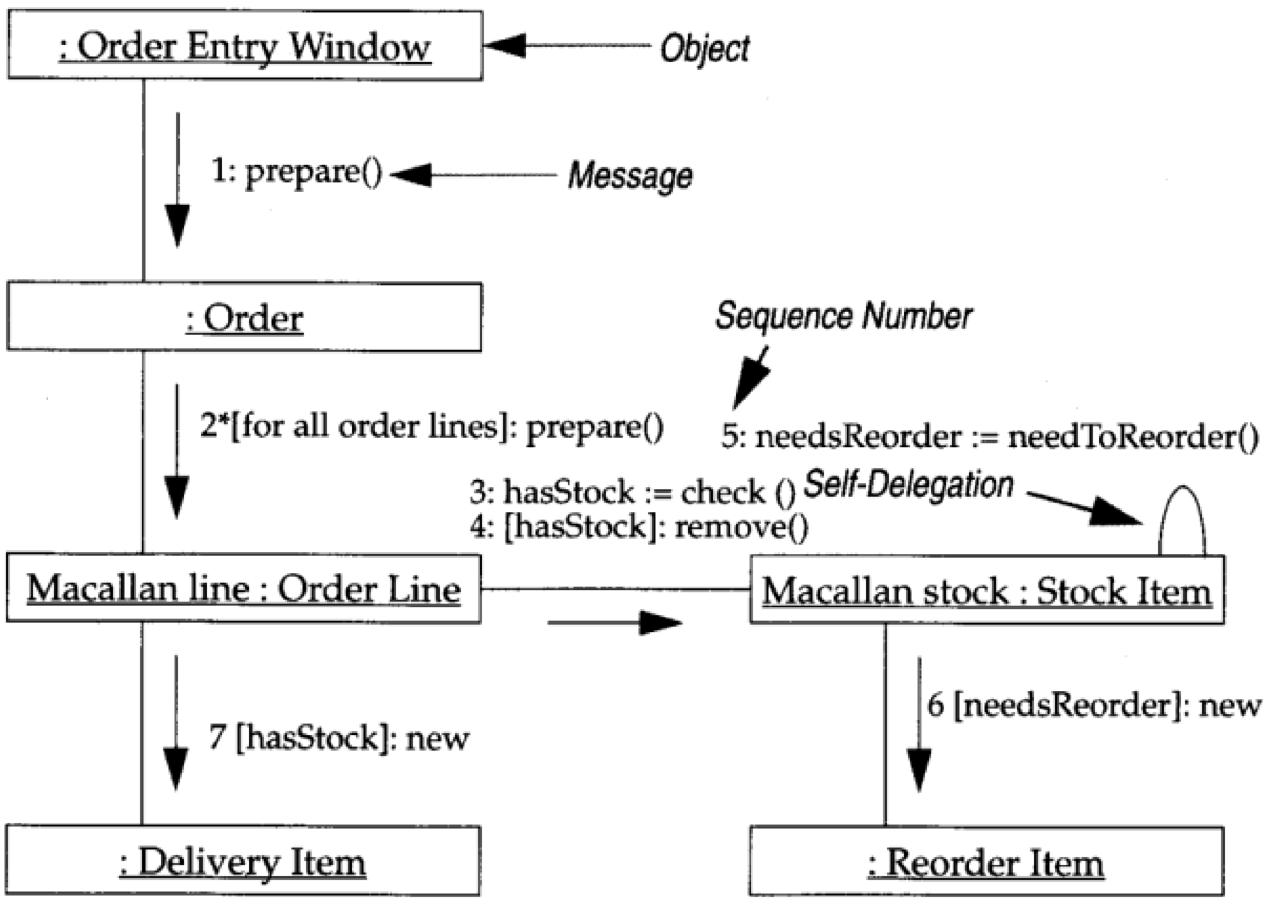
Sequence Diagram Example



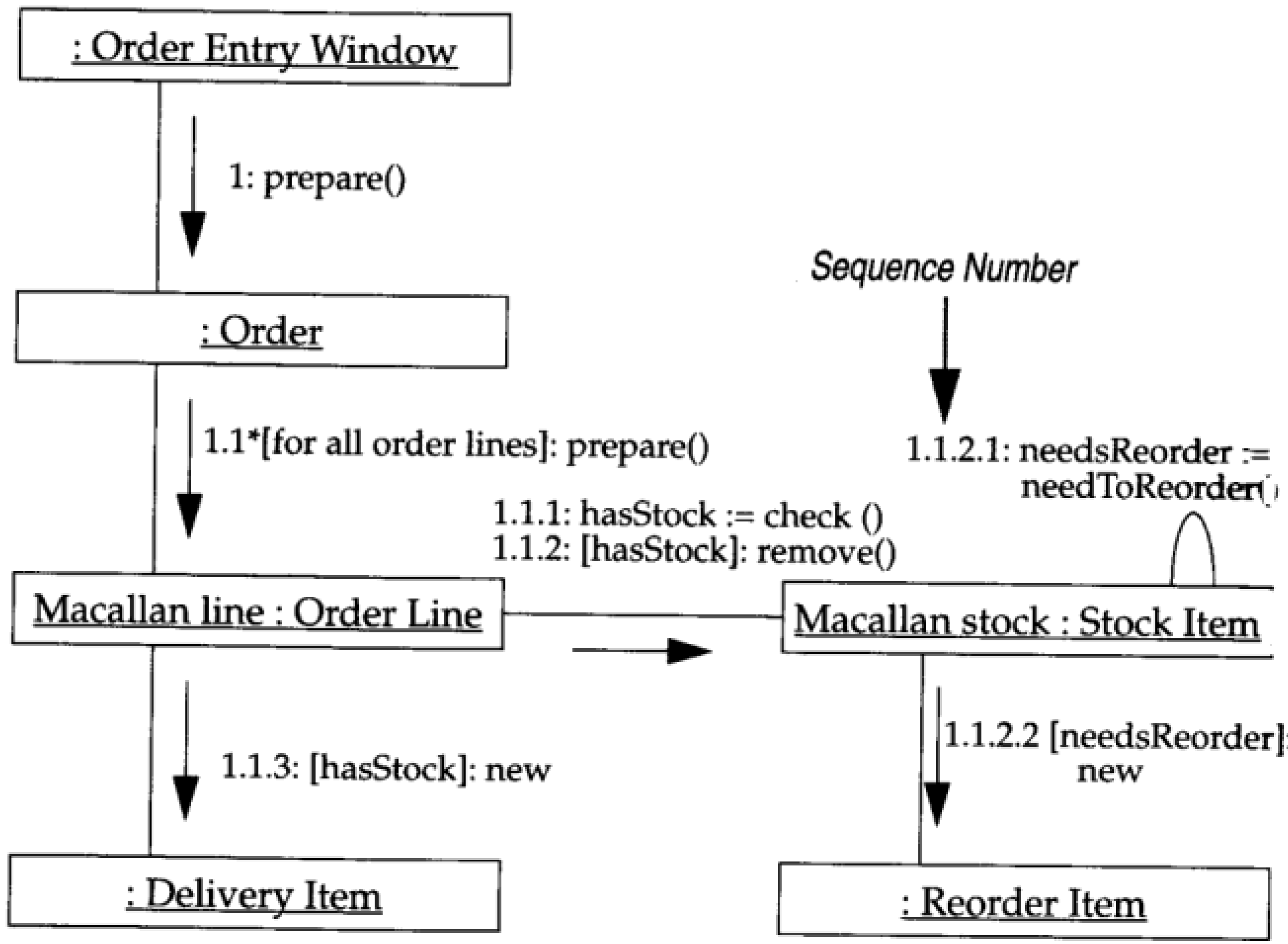
Sequence Diagrams

- Vertical line is called lifeline
- Each message represented by an arrow between lifelines
  - Labeled at minimum with message name
  - Can also include arguments and control information
  - Can show self-call by sending the message arrow back to the same lifeline
- Can add condition which indicates when message is sent, such as [needsReorder]
- Can add iteration marker which shows that a message is sent many times to multiple receiver objects

Collaboration Diagram Example



Collaboration Diagram Example
Decimal Numbering System



Sequence vs Collaboration Diagrams

- Sequence diagrams are better to visualize the order in which things occur
- Collaboration diagrams also illustrate how objects are statically connected
- You should generally use interaction diagrams when you want to look at the behavior of several objects within a single use case.

The UML universe

- There is a lot more to the UML than what we have shown here
  - More diagram types
    - State diagrams, activity diagrams, use cases, deployment diagrams, …
  - More notational features in all diagram types
    - Stereotypes, parameterized classes, …
- We will touch some UML features not shown here during the course and will explain them as needed

UML Misconceptions and Limitations

- UML is not language-independent. It *is* a language, as the L in UML suggests.
- This language is something like a high-level “best-of” of common OO programming language features
  - It contains notation for features that are only available in some (or even no) programming language (such as: dynamic classification)
  - Every OO language has features that have no corresponding notation in the UML (e.g. wildcards in Java)
  - The same UML notation may have a different meaning in different OO languages (e.g. visibility)
- The UML has no clearly defined semantics. This is both a limitation and a feature
  - Good for informal diagrams, bad for formal specifications
- No consensus in the community about the scenarios where UML is useful

Literature

- Martin Fowler. *UML Distilled*. Addison-Wesley.
- Beck, Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*. OOPSLA’ 89
  available online at c2.com/doc/oopsla89/paper.html