UML

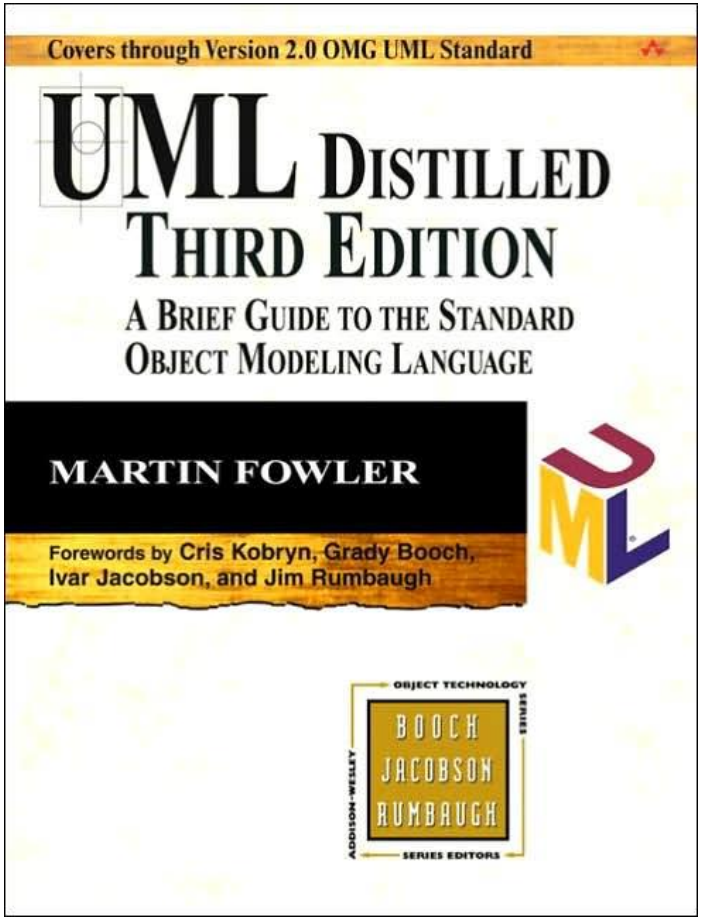
CS 345 Winter 2018

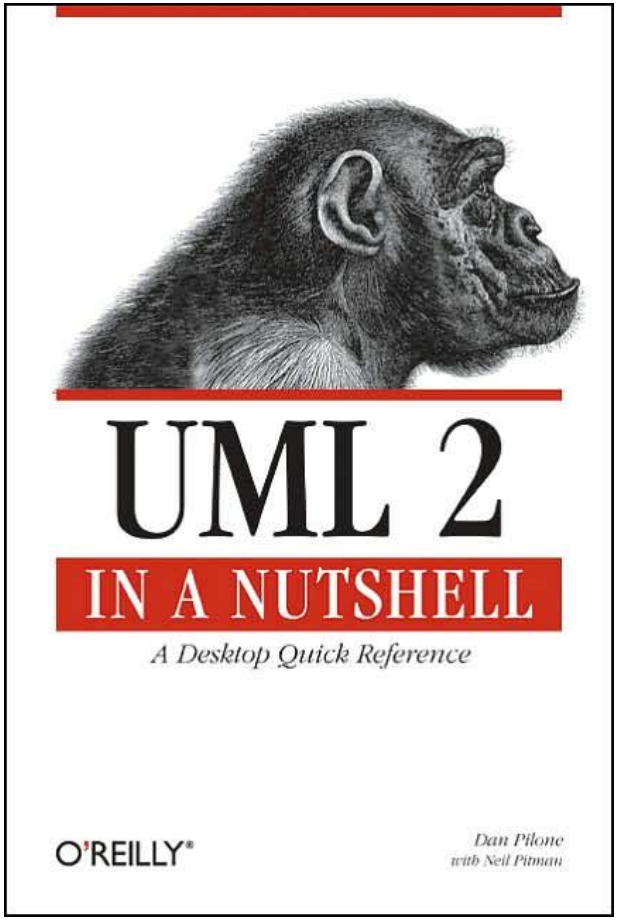
Chris Reedy

Outline

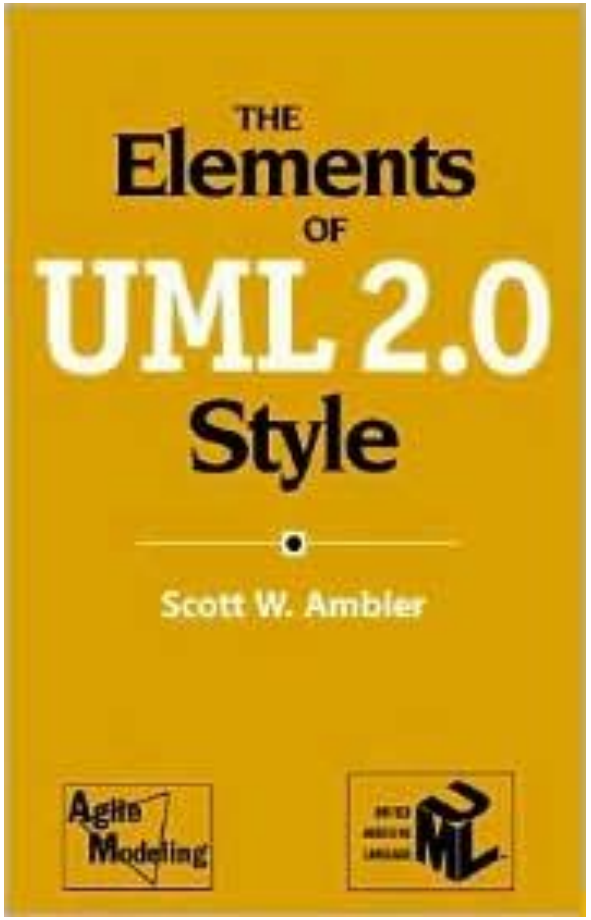
- UML
  - UML Background
  - UML Diagrams
  - Using UML
  - Advice on UML

References
UML



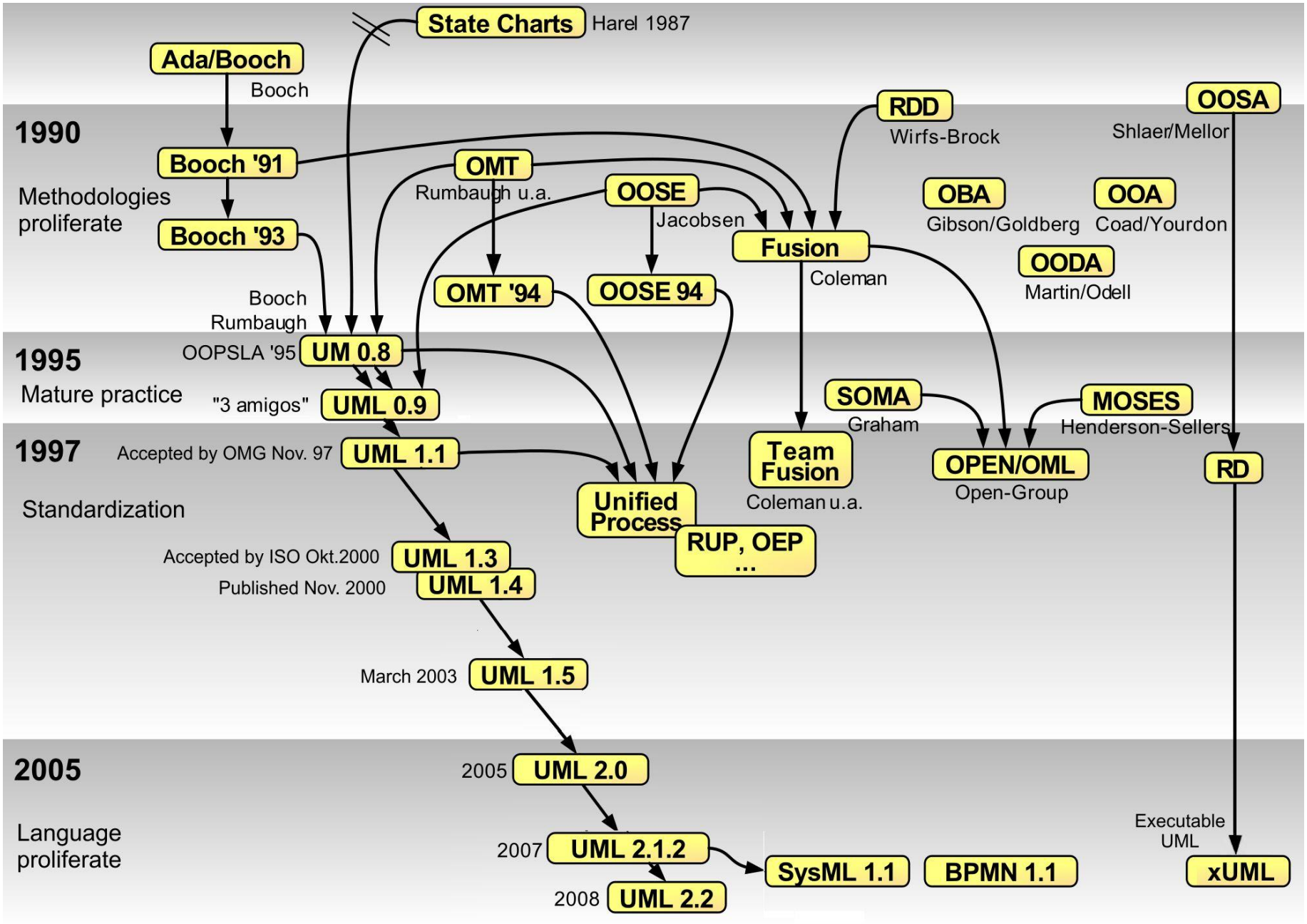


Additional References
UML



Unified Modeling Language (UML)

- What it is
  - A collection of diagrams for expressing the solution to common requirements analysis and software design problems
- What it is not
  - A software development process
    - A prescription for how to build software
  - A design methodology
    - A prescription for how to design software
  - However, UML can be an important part of either of the above


State Charts
Harel 1987
Ada/Booch
Booch
1990
Booch '91
Methodologies
proliferate
Booch '93
OMT
Rumbaugh u.a.
OOSE
Jacobsen
RDD
Wirfs-Brock
OOSA
Shlaer/Mellor
OBA
Gibson/Goldberg
OOA
Coad/Yourdon
Fusion
Coleman
OODA
Martin/Odell
OMT '94
OOSE 94
Booch
Rumbaugh
1995
OOPSLA '95
UM 0.8
Mature practice
"3 amigos"
UML 0.9
SOMA
Graham
MOSES
Henderson-Sellers
1997
Accepted by OMG Nov. 97
UML 1.1
Team
Fusion
Coleman u.a.
OPEN/OML
Open-Group
RD
Unified
Process
RUP, OEP
...
Standardization
Accepted by ISO Okt.2000
UML 1.3
Published Nov. 2000
UML 1.4
March 2003
UML 1.5
2005
2005
UML 2.0
Language
proliferate
2007
UML 2.1.2
SysML 1.1
BPMN 1.1
Executable
UML
xUML
2008
UML 2.2

A Little UML History

- Precursors – “the three amigos”
  - Grady Booch – OOAD – Object-Oriented Analysis and Design
  - Ivar Jacobson – OOSE – Object-Oriented Systems Engineering
  - James Rumbaugh – OMT – Object Modeling Technique
- UML 1.0
  - Original version standardized 1996-1997 by OMG (Object Management Group)
- UML 2.0
  - New version, standardized 2005
  - Improved precision of specification for communications between computers
  - Some reorganization and additional diagrams
  - Current version: 2.5, June 2015

Degrees of UML

- As a sketch
  - Illustrate key points
  - “Throwaway” diagrams
- As a blueprint
  - Detailed specification of one or more aspects of a system
- As a programming language
  - Every aspect of a system is modeled
  - In theory, this is combined with automated transformation to generate the code

Some UML “Rules of Thumb”

- Nearly everything in UML is optional
  - You should only use the parts that you need to convey your message.
- UML models are rarely complete
  - Since nearly everything is optional
  - Also depends on requirements of your UML tool
- UML is open to interpretation
  - You need to establish conventions and understandings for your organization and/or project
  - If you use automated translation, your conventions have to match those of your translation tool
- UML is intended to be extended
  - For example, stereotypes

UML Diagrams

Structural Diagrams

- These diagrams capture **static** aspects of the system
- Diagrams
  - Class*
  - Component
  - Composite Structure
  - Deployment
  - Package
  - Object

Behavioral Diagrams

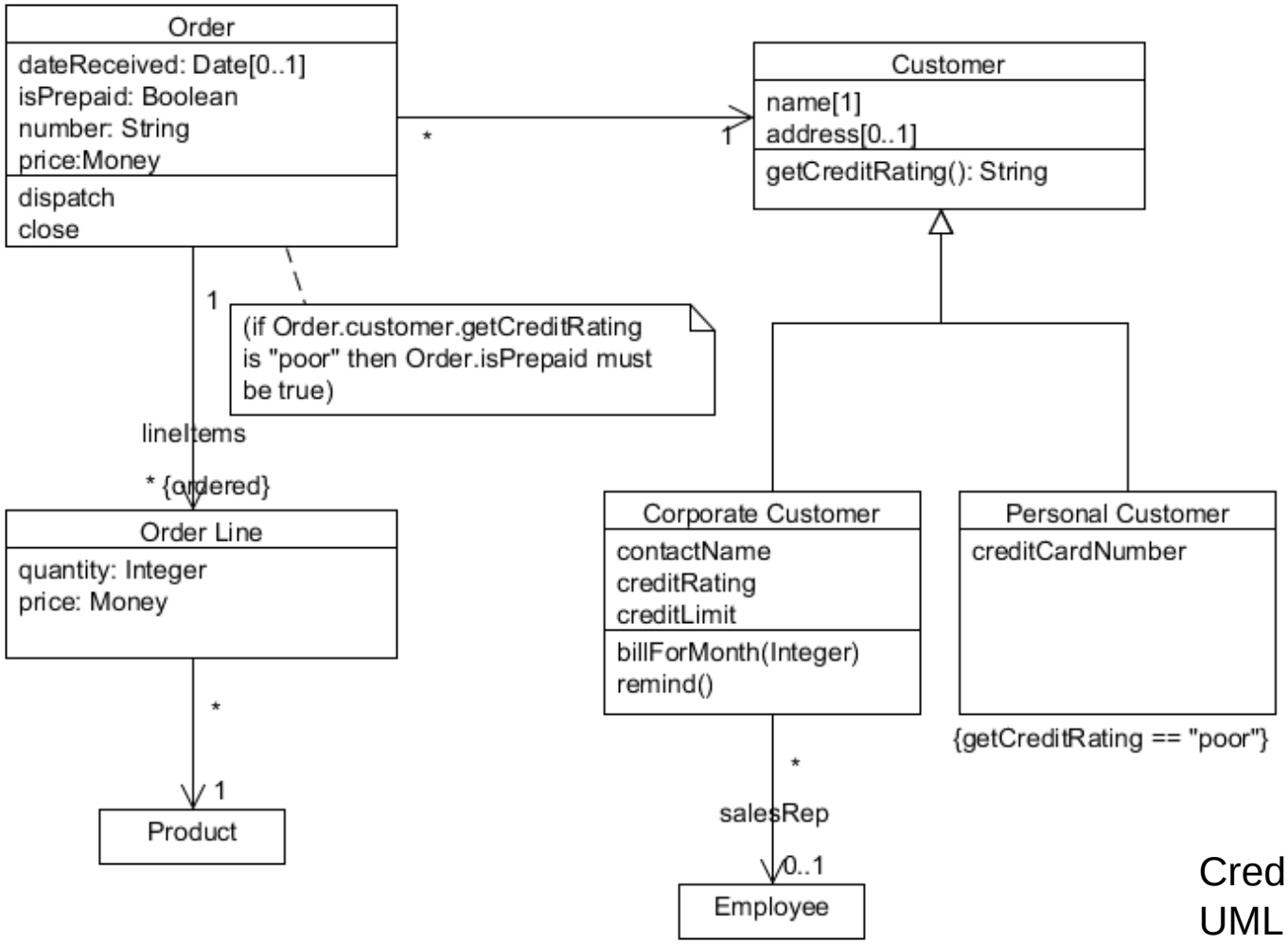
- These diagrams capture **dynamic** aspects of the system
- Diagrams
  - Activity*
  - Communication
  - Interaction Overview
  - Sequence*
  - State machine
  - Timing
  - Use Case

*Described in more detail in this lecture

Class Diagram

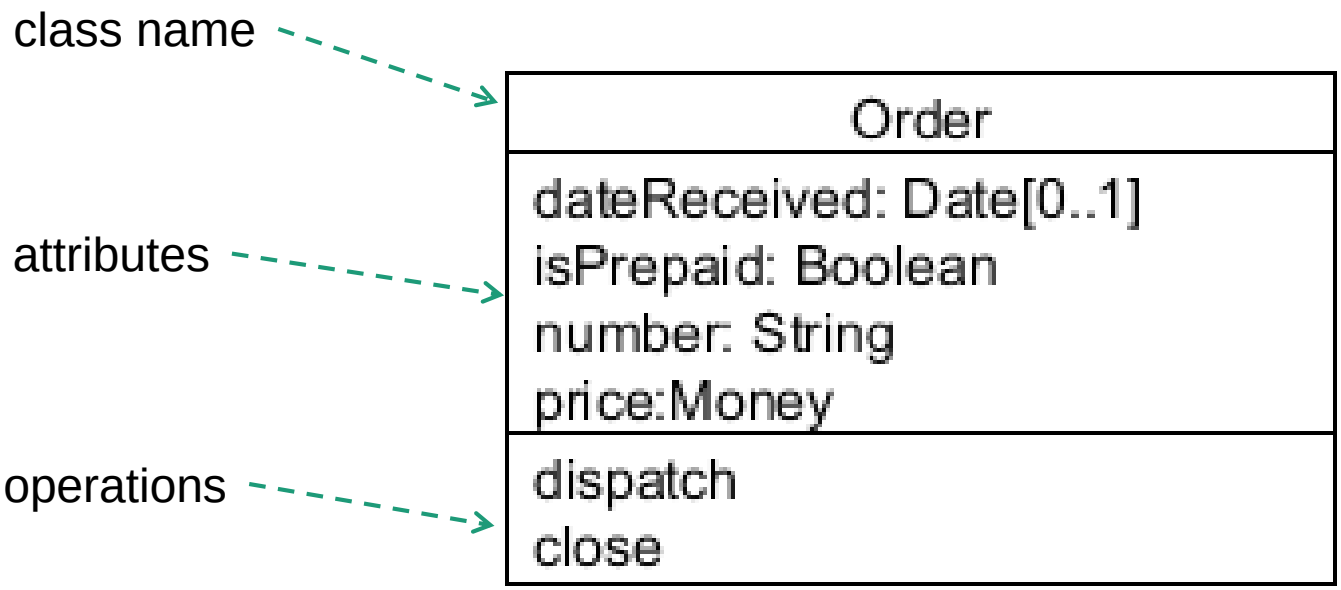
- Used to show the *static* structure of classes in your software
  - Classes
  - Properties of classes
    - Attributes, operations
  - Relationships between classes
- Most frequently used UML diagram

A Simple Class Diagram



Credit:
UML Distilled

A Simple Class Diagram
A Class



- You can omit attributes or operations or both. Leave a double line if you omit attributes but not operations.
- The class name can be annotated with stereotypes and properties.

Attributes

visibility name: type [multiplicity] = default_value {properties}

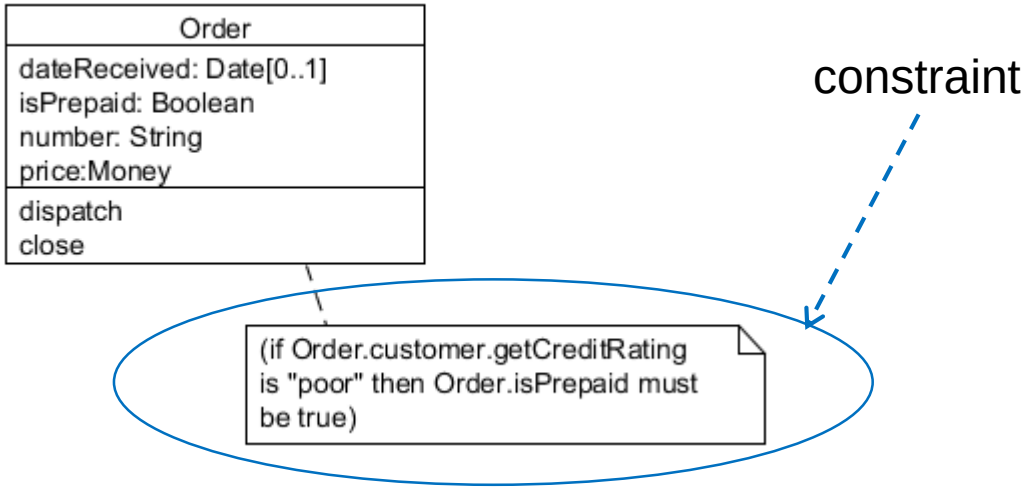
- *visibility*
  - \+ public
  - \- private
  - # protected
  - ~ package
- *name* – the name
- *type* – the type (e.g. int, String, Employee)
- *[multiplicity]*
  - 1 exactly one – assumed if not specified
  - 0..1 zero or one (at most one)
  - * any number, including zero
  - 1..* any number greater than zero
  - 3..10 at least 3, no more than 10 (inclusive bounds)
- *= default_value* – default value
- *{properties}*
  - A string, for example {readOnly}, {readOnly, range=0..999}
- Only the name is required—everything else is optional

Methods

visibility name(parameters): return-type {properties}

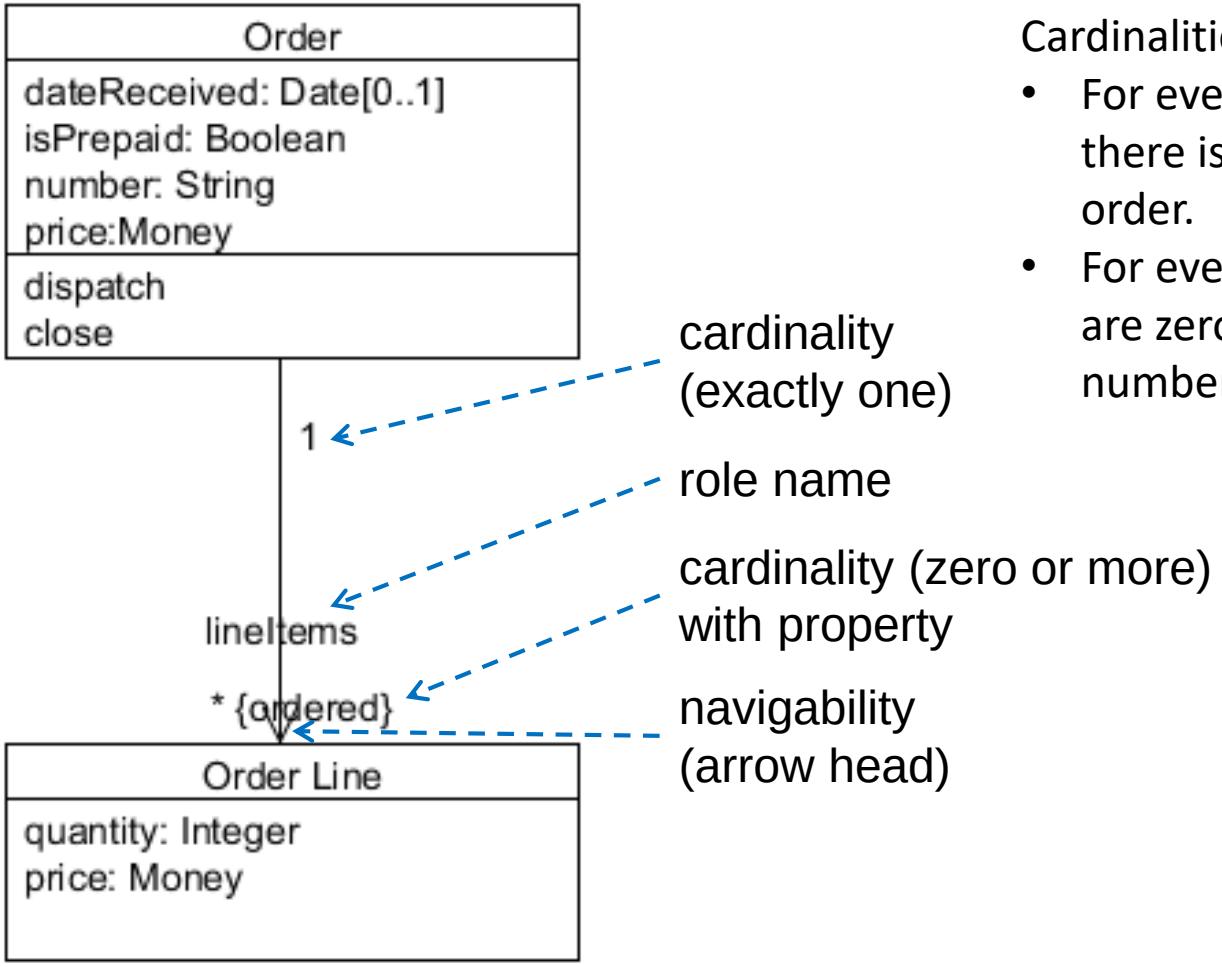
- *visibility, name* – same as attributes
- *return-type* – the type (e.g. int, Employee)
  - Return nothing is *void*
  - Missing means that *return-type* is unspecified
- *(parameters)*
  - list of
    direction name: type [multiplicity] = default_value {properties}
  - *direction* – one of *in*, *out*, *inout*, or *return*, *in* is default
  - Others are same as attributes
  - Missing parameters means unknown or unspecified
- *{properties}* – same as attribute
- Again, only the name is required—everything else is optional

A Simple Class Diagram Constraint



- There is an Object Constraint Language (OCL) for specifying constraints. Use of OCL can be mandated by tools or local convention.
- In the absence of OCL, any English language phrase describing the class is allowed.

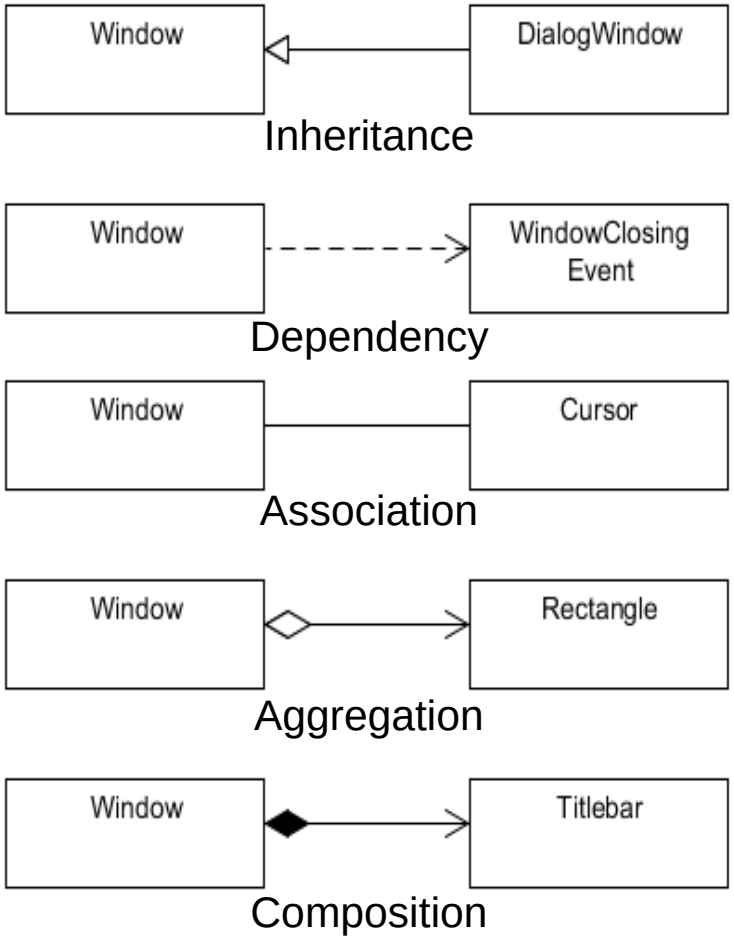
A Simple Class Diagram Association



Cardinalities (*look across*):

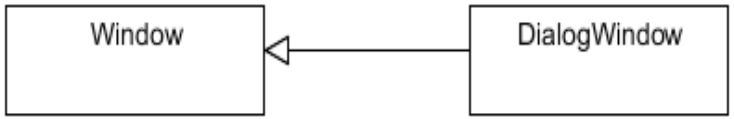
- For every Order Line there is exactly one order.
- For every Order there are zero or more (any number of Order Lines.

Class Diagram Relationships



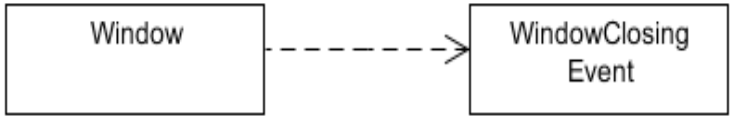
Credit: UML 2.0 In A Nutshell

Class Diagram Inheritance



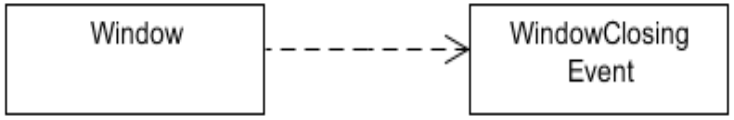
- One class inherits from (extends, specializes, is a subclass of, IS-A) the other.
  - DialogWindow IS-A Window
- **Do not** annotate inheritance relationships with names, cardinalities, or navigability

Class Diagram Relationship Dependency



- Indicates that one class depends on or knows of or makes use of another class.
- If the code for WindowClosingEvent changes then the code for Window may have to be change.
- Typically indicates that the code for Window "imports" or "withs" or "includes" WindowClosingEvent.

Class Diagram Relationship Dependency



- No structural relationship is implied.
  - Window does not have to have a WindowClosingEvent attribute.
- The arrow head shows the direction of the dependence.
  - No arrow head would imply co-dependency—each depends on the other.
- This relationship should not be annotated with cardinality.
  - A role name may be appropriate.
- Likely relationship: WindowClosingEvent is a parameter of some method of Window
  - Maybe the type of some local variable in a method or the return type of a method.

Class Diagram Relationship Association

- Indicates a structural relationship between classes. A connection exists between the classes at run-time.
- Window has a associated Cursor.
  - Likely, Window has an attribute of type Cursor.
    - In a class diagram, the attribute might or might not be listed among the attributes.
- Implies navigability, that is, one instance can be found from the other.
  - No arrows: Navigable in both directions
  - One arrow: Navigable in the indicated direction.
  - X: Not navigable in the indicated direction.
- Associations should be annotated with cardinalities.

Class Diagram Relationship Aggregation

- Indicates a whole-part relationship.
  - Implies Association
- The Rectangle is an essential part of the Window.
- Aggregation indicates that the Rectangle might be shared with other Windows.
- Arrow heads indicate navigability, same as Association.
  - It is always assumed that the relationship is navigable from whole to part (Window to Rectangle). So,
    - No arrow heads indicates that the relationship may be navigable from part to whole.
    - One arrow head (pointing from whole to part) indicates the the relationship is not (necessarily) navigable from part to whole.

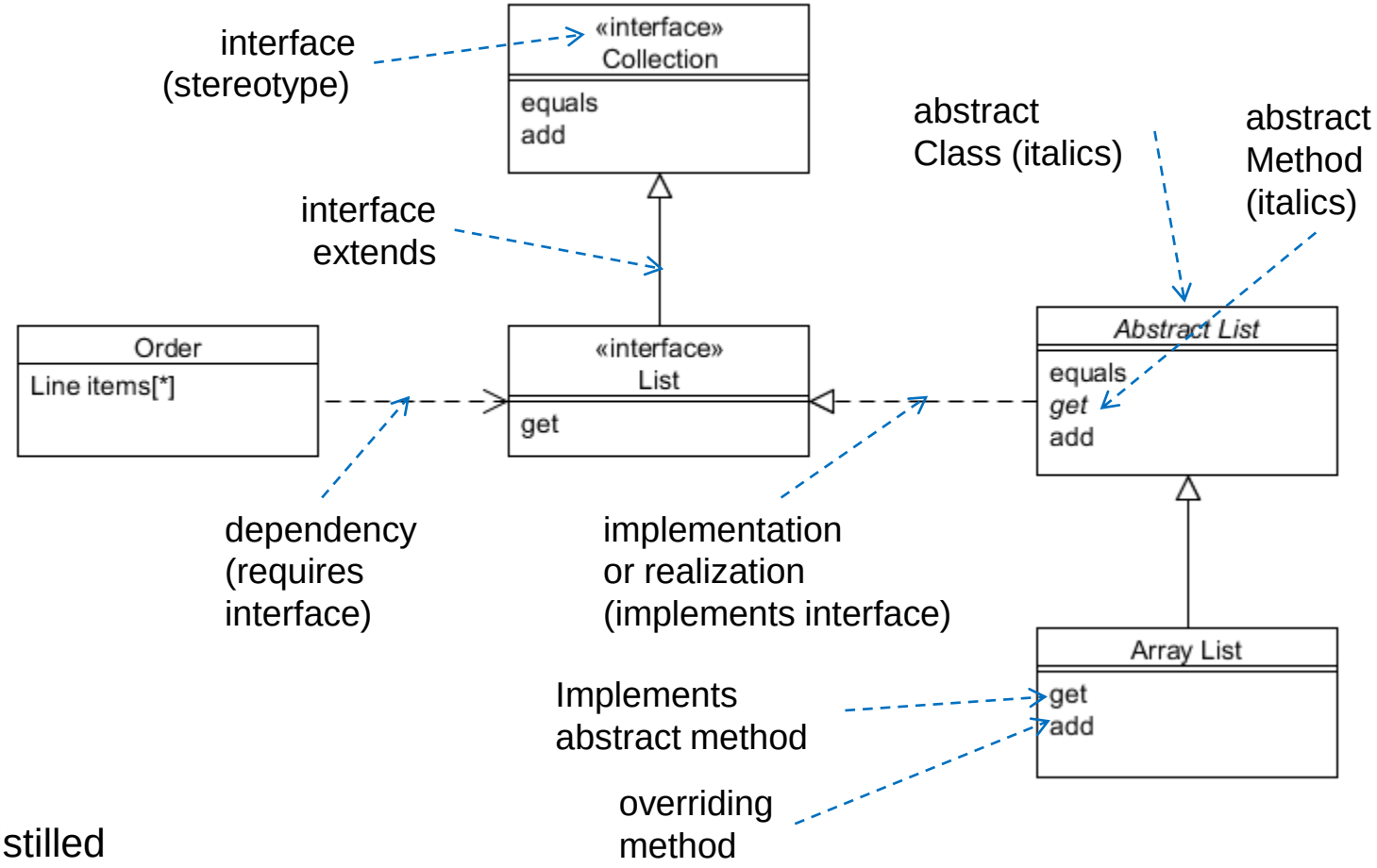
Class Diagram Relationship Composition

- Indicates a whole-part “HAS-A” relationship.
- TitleBar is part-of a Window. Window owns a TitleBar.
- Composition indicates that the TitleBar is not part of any other window.
  - In general, composition implies that when the Window is garbage, so is the TitleBar.
    - In C++, this would usually imply that when the Window is deleted, the TitleBar should also be deleted.

Aggregation vs. Composition

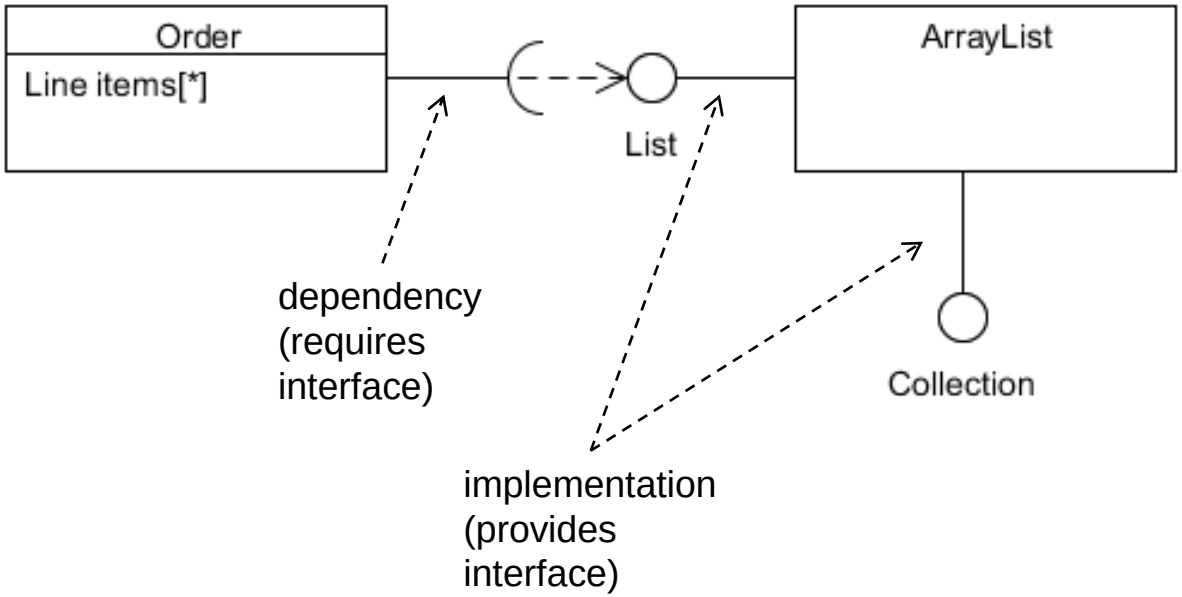
- “there is very little difference between association, aggregation and composition at the coding level.” – Elements of UML 2.0 Style
  - Distinction between the three is mostly capturing the modeler’s intent.
- Some style recommendations:
  - A frequent source of confusion – UML Distilled
  - Use composition when classes represent physical parts
  - Use composition when objects share a persistent lifecycle
    - That is they are created and destroyed together
  - Don’t worry about it. – Elements of UML 2.0 Style

Interfaces and Abstract Classes



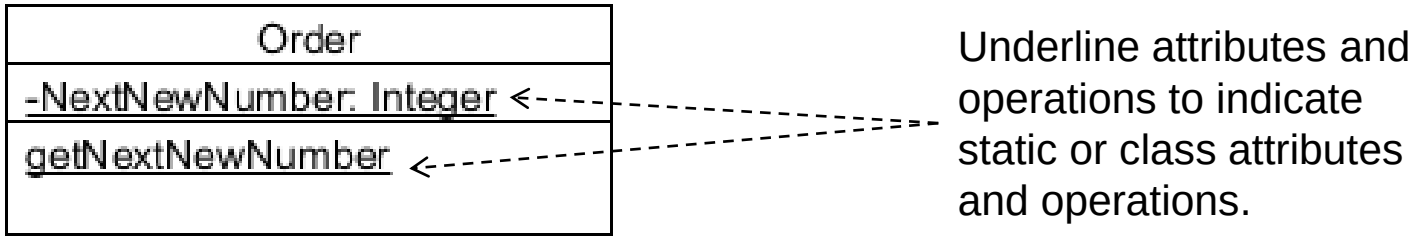
Credit:
UML Distilled

Ball and Socket Interfaces



Credit:
UML Distilled

Static Attributes and Operations



Activity Diagram

- Used to show the execution and flow of the behavior of a system
- This diagram is applicable to just about any behavioral modeling activity
- For software modeling an activity usually represents a behavior invoked via a method or subprogram call

Simple Activity Diagram

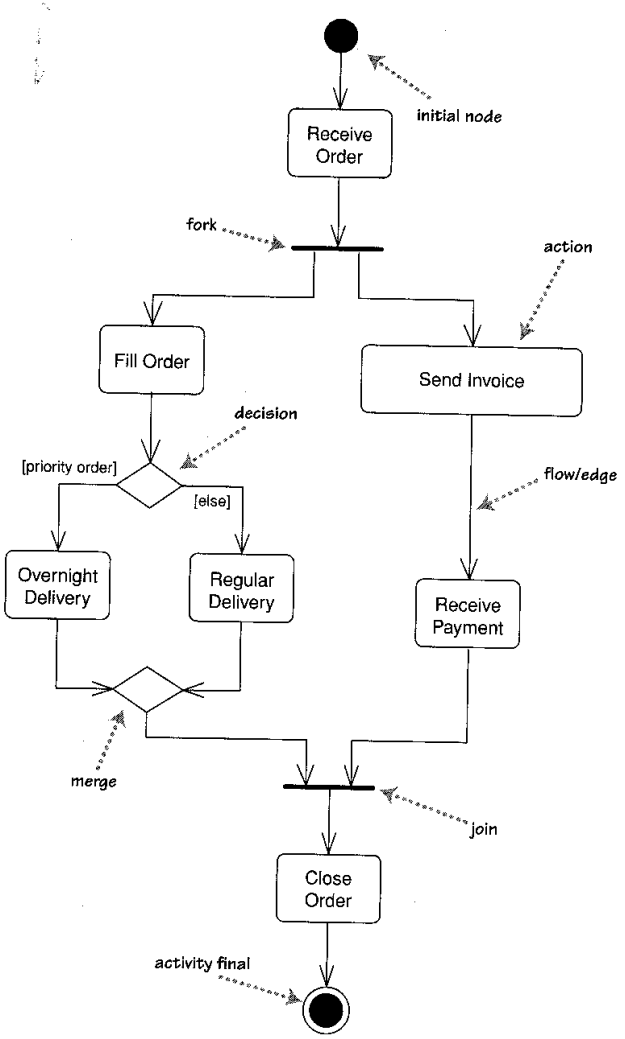


Figure 11.1 *A simple activity diagram*

Credit:
UML Distilled

Partitions or Swimlanes

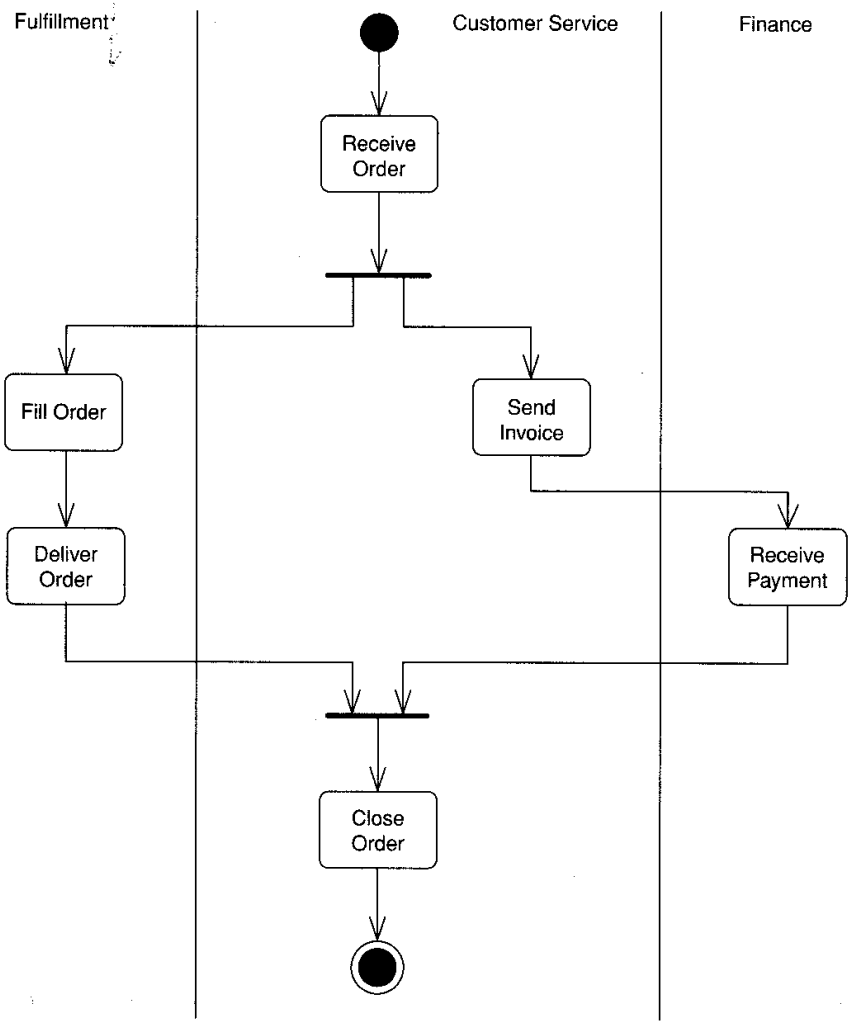


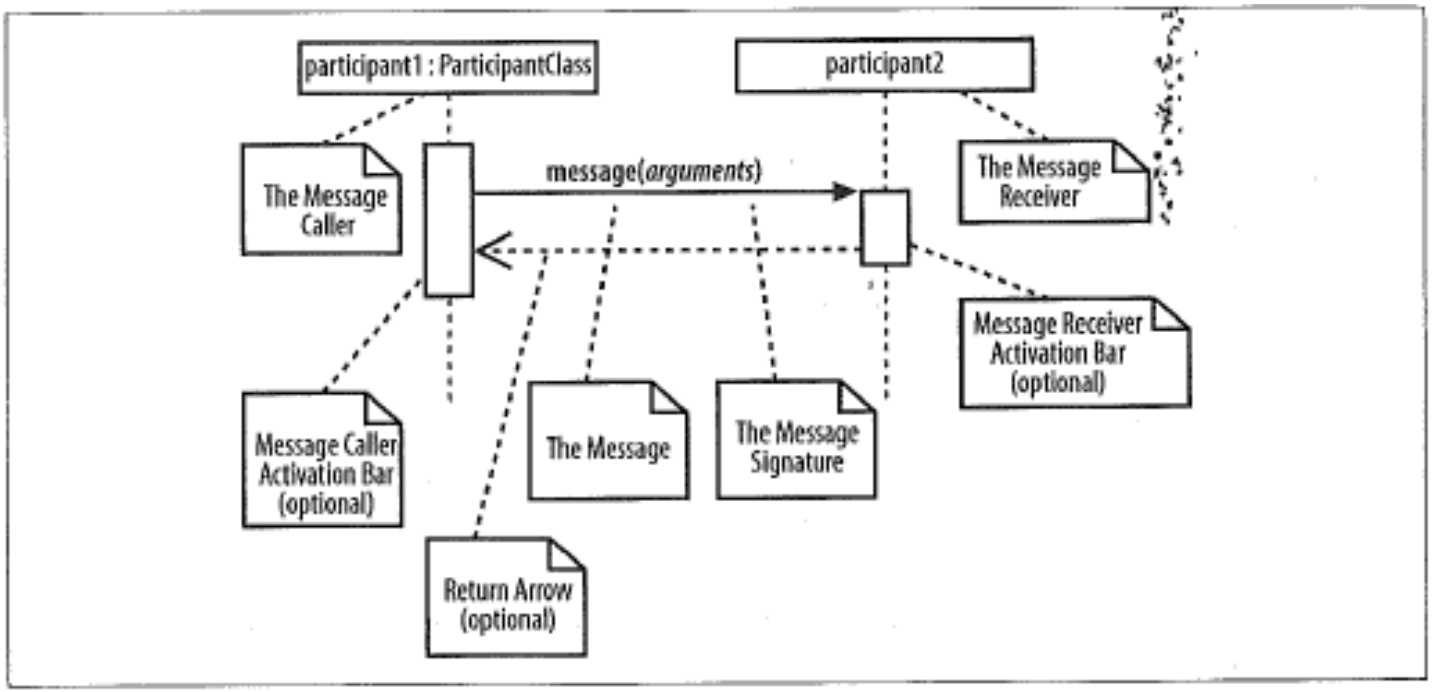
Figure 11.4 *Partitions on an activity diagram*

Credit:
UML Distilled

Sequence Diagram

- Shows the communication between multiple objects
- Questions answered by sequence diagrams
  - What messages are sent between objects?
    - What operations are invoked?
  - How do the objects cooperate to realize a specific result?

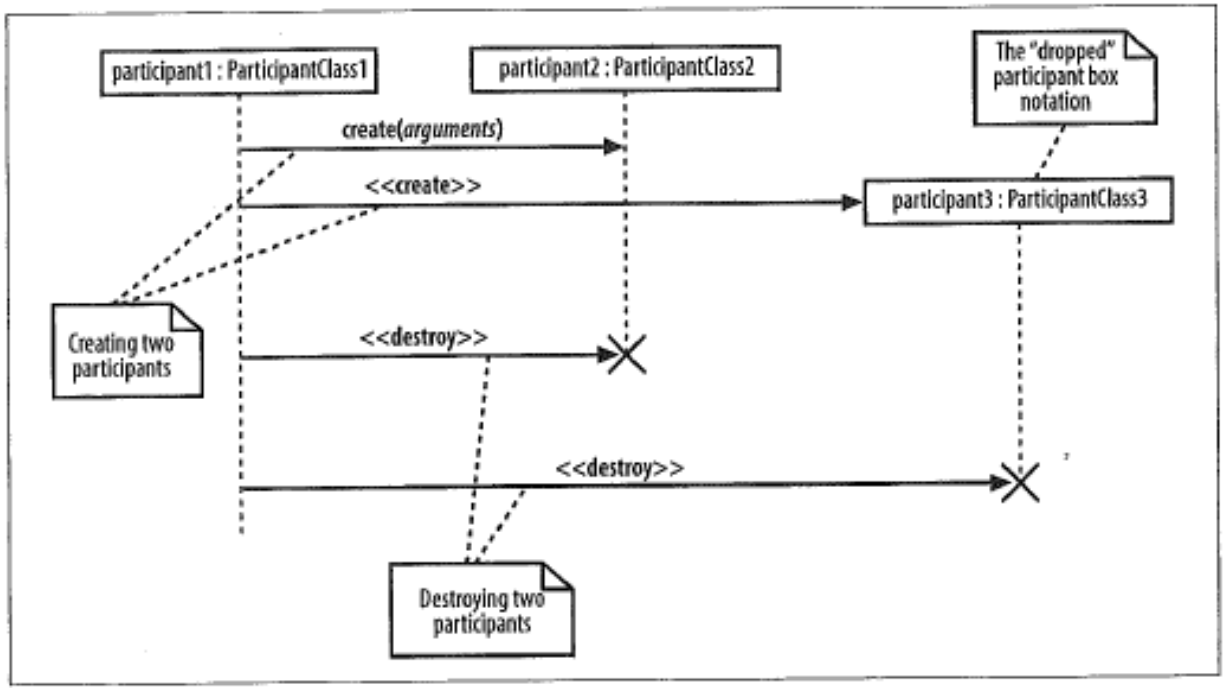
Sequence Diagram Basics



Credit: Learning UML 2.0

- Participants are Objects (*name:type* or *name*)
- Vertical dotted lines are “life lines”. Boxes show activity.
- Return is optional. Can be labeled with what is returned.

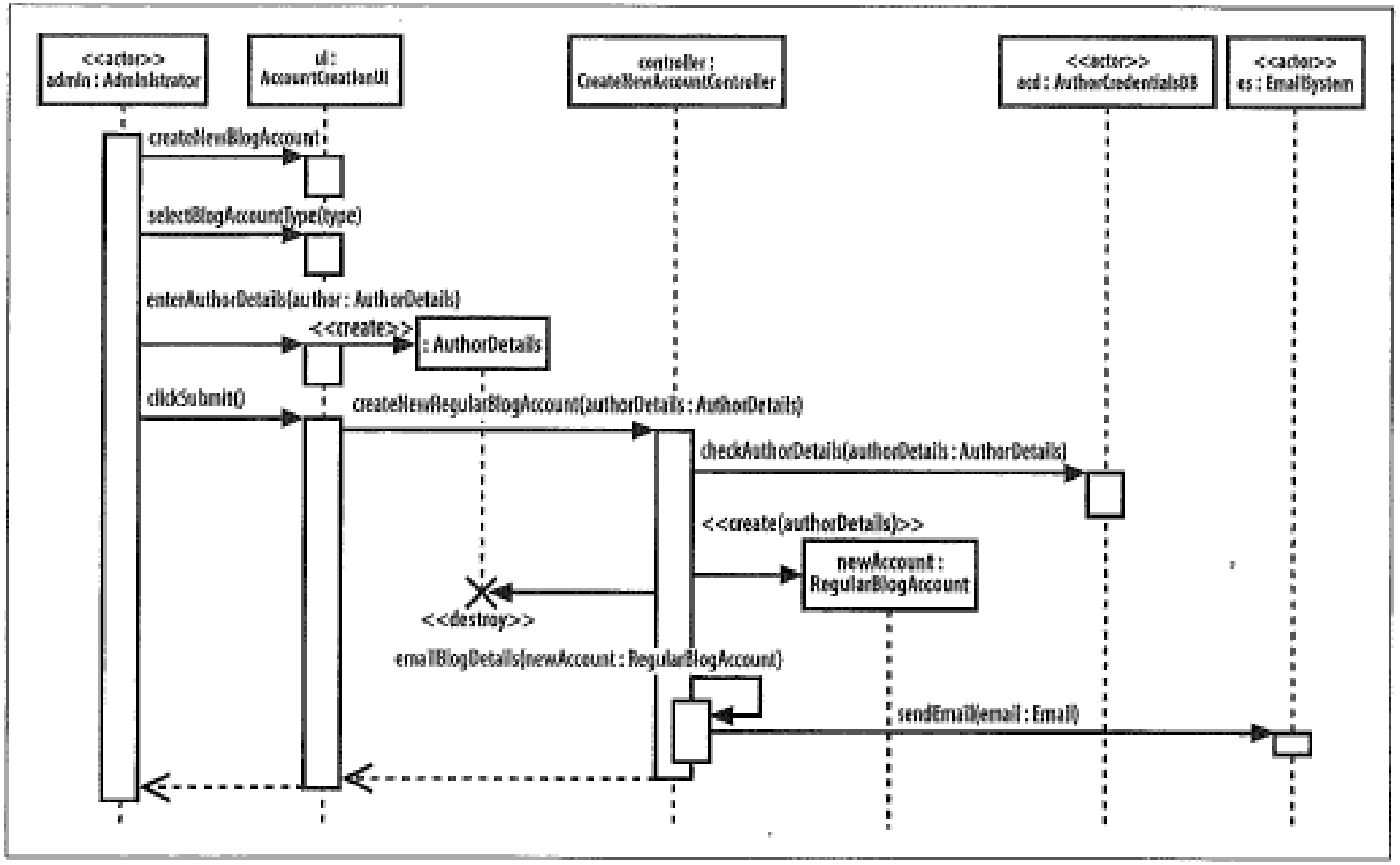
Sequence Diagram
Creation and Destruction



Credit: Learning UML 2.0

- Show creation and destruction of objects.
  - Messages don't have to be "<<create>>" and "<<destroy>>".
  - <<destroy>> message is optional. An X can be used to show that object is no longer used or useful.

An Example Sequence Diagram



Credit: Learning UML 2.0

Advice on UML

- Who is your audience?
  - Yourself—Do I know how to solve this problem?
  - Other designers—here's our proposed approach
  - Reviewers—Here is what we propose to implement.
- What's the time frame?
  - I need to get a start on writing the code
  - There's a group of us that need to agree on how to break up our subsystem
  - We're documenting the structure of the system for future maintainers
- What aspects of the system am I trying to capture?

Advice on UML
Avoid “Wallpaper” Diagrams

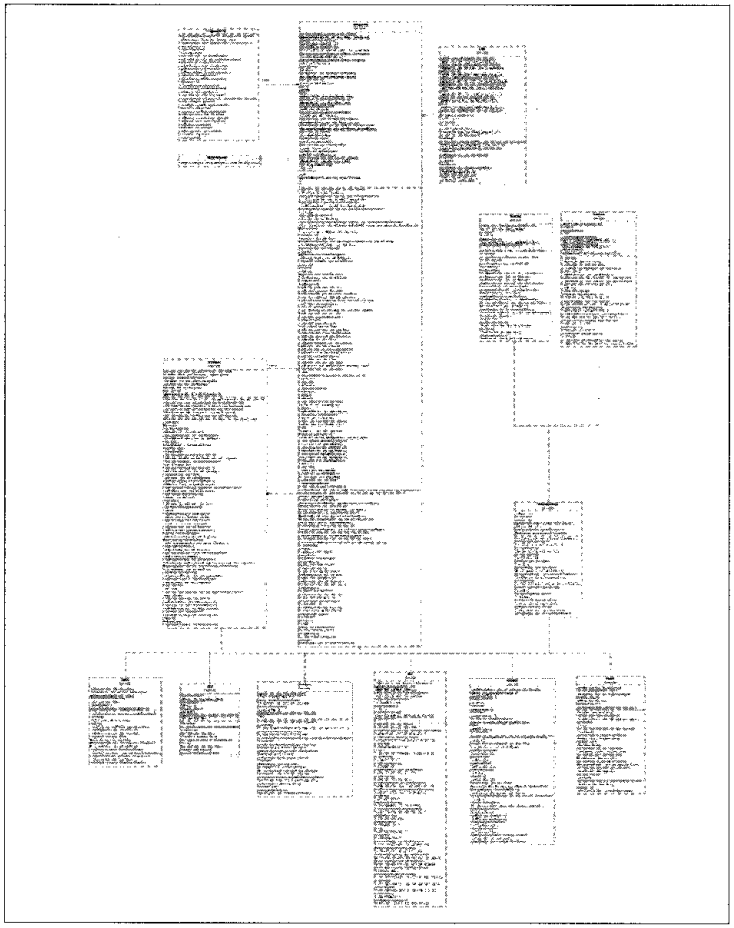


Figure 12-1. An overloaded and ineffective class diagram

Credit:
UML 2.0
In A Nutshell

Advice on UML – Separate Inheritance and Relationships

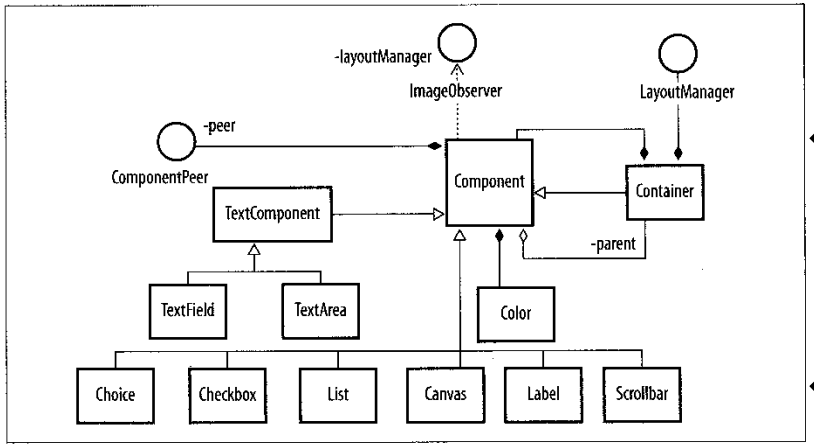


Figure 12-2. The structure of commonly used components within the Java AWT

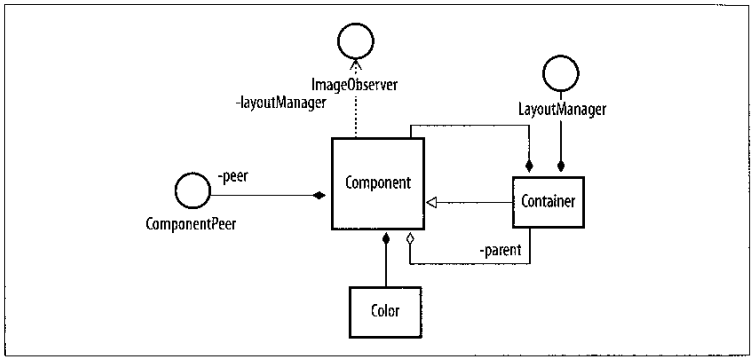


Figure 12-3. Structure of Components and Containers in the AWT

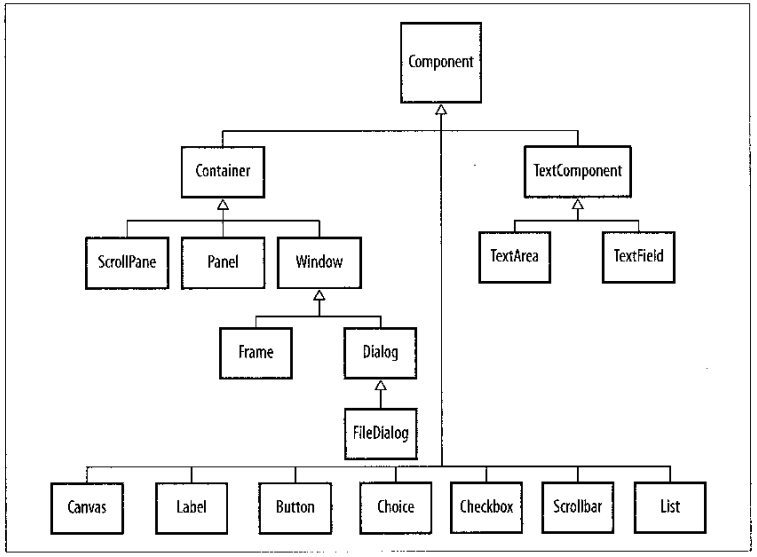


Figure 12-4. Inheritance tree of Component in the AWT

Credit:
UML 2.0
In A Nutshell

Advice on UML
Avoid Sprawling Scope

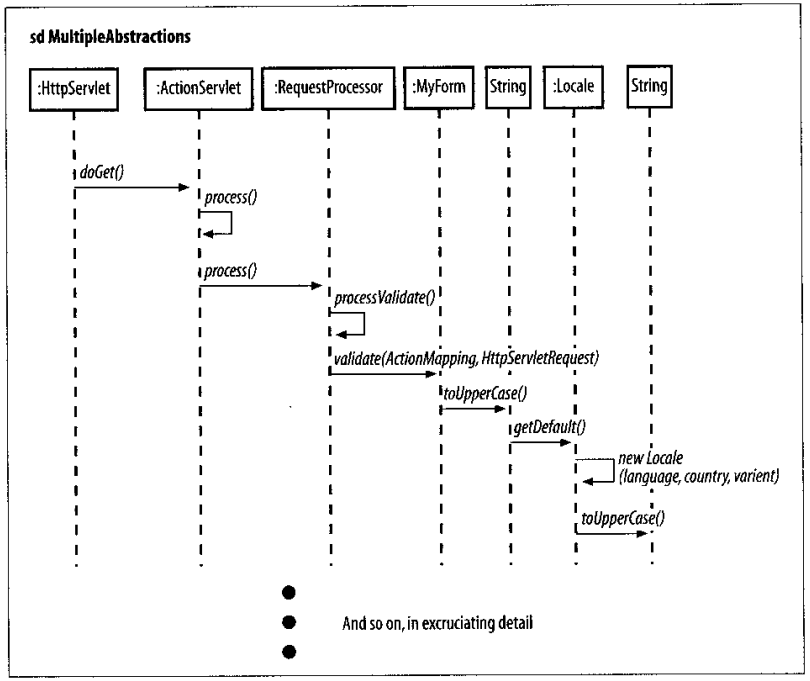


Figure 12-5. Sequence diagram of HTTP form processing, spanning Struts, application, and java.lang

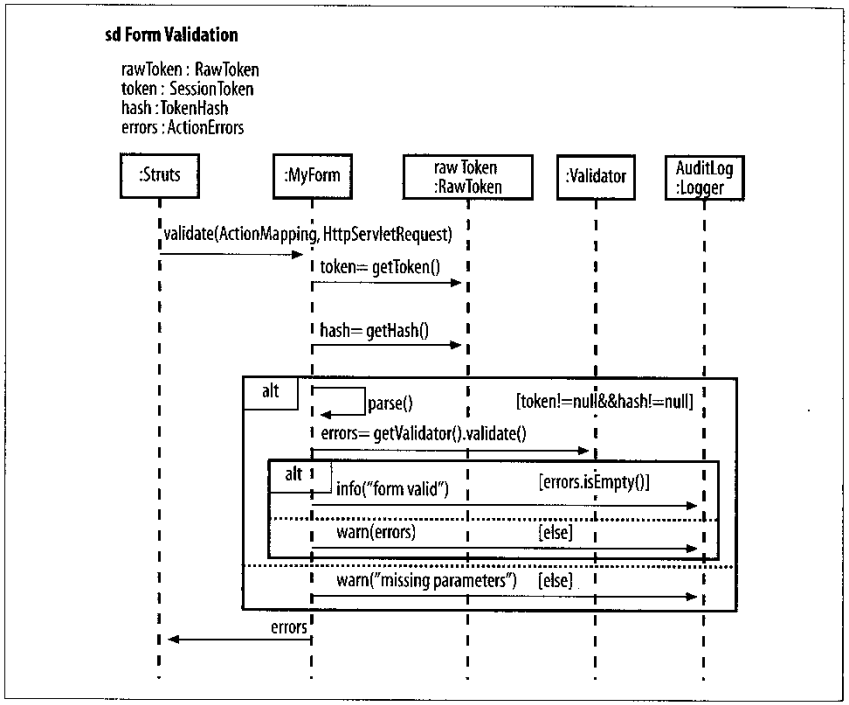


Figure 12-6. Sequence diagram with scope restricted to programmer-defined classes

Credit:
UML 2.0 In A Nutshell

Advice on UML
One Diagram/One Abstraction

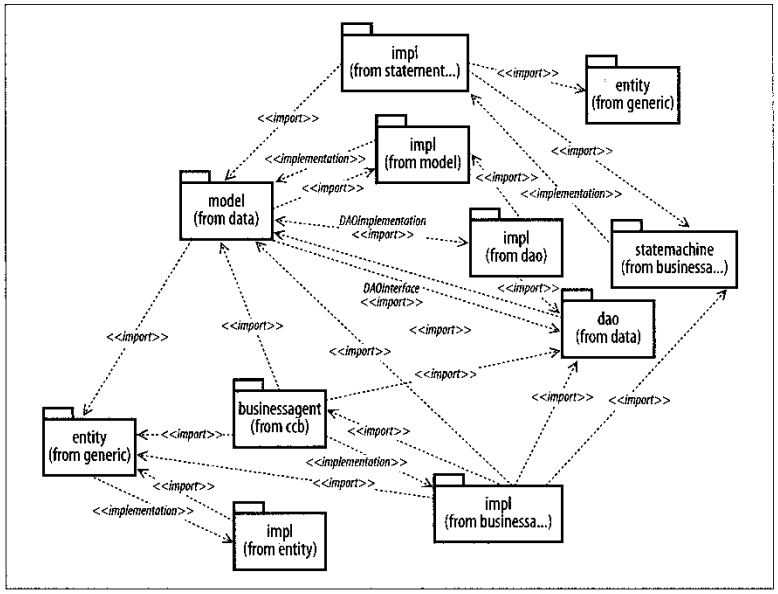


Figure 12-7. Package diagram controlling imports and code generation directives

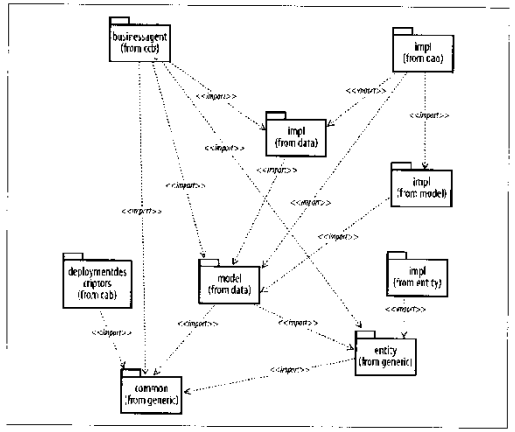


Figure 12-8. Package imports

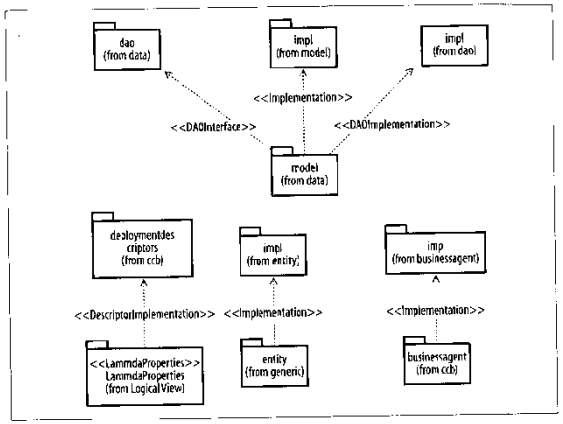


Figure 12-9. Code generation directives

Credit:
UML 2.0 In A Nutshell