Software Engineering
G22.2440-001

Session 7 – Sub-Topic 2
UML Review

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Diagrams Reviewed ...

- A review of the various notations (use-case, activity, class, sequence, collab, component, etc…)
- And a close look at more "exotic" notations

Use Cases: Scenario based requirements modeling

- Recommended: UML distilled...

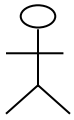
Use Cases

Use case

- specifies the behavior of a system
- sequence of *actions* to yield an observable result of value to an *actor*
- Capture the *intended* behavior (the what) of the system omitting the implementation of the behavior (the how)
- customer requirements/ early analysis

What is a use case?

- Description of a sequence of *actions*, including variants (specifies desired behavior)
- Represents a functional requirement on the system
- Use case involves interaction of actors and the system



Financial Officer

Use cases: terms and concepts

- Unique name
- Sequence of actions (event flows)
  - textual (informal, formal, semi fomal)

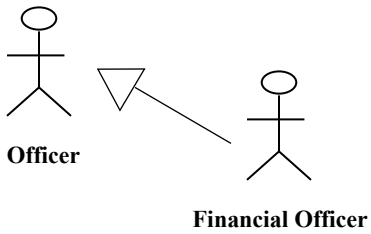
    Main flow of events: The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a pin number...
  - interaction diagrams

Actors

- *Role* that a user plays with respect to the system
- Actors carry out use cases
  - look for actors, then their use cases
- Actors do not need to be humans!
- Actors can get value from the use case or participate in it

Actors

- Actors can be specialized



- connected to use cases only by association
- association = communication relationship (each one sending, or receiving messages)

Use case description

- Generic, step-by-step written description of a use case's event flow
- Includes interactions between the actor(s) and a use case
- May contain extension points
- Clear, precise, short descriptions

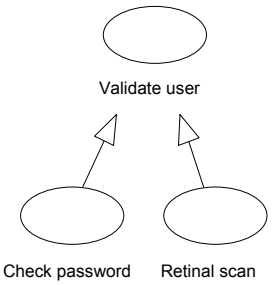
Example use case description

- Capture deal
  1. Enter the user name & bank account
  2. Check that they are valid
  3. Enter number of shares to buy & share ID
  4. Determine price
  5. Check limit
  6. Send order to NYSE
  7. Store confirmation number

Organizing Use Cases

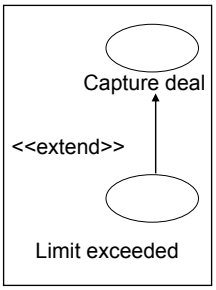
- Generalization
- Use/Include
- Extend

Generalization relationship



- child use case inherits behavior and meaning of the parent use case
- child may add or override the parent's behavior
- child may substitute any place the parent appears

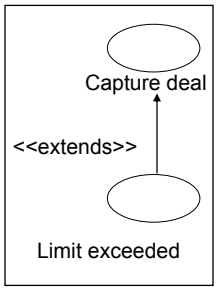
Extends relationship



- Allows to model the part of a use case the user may see as optional
- Allows to model conditional subflows
- Allows to insert subflows at a certain point, governed by actor interaction

- represented by an *extend* dependency
- extension points (in textual event flows)

Extends relationship



- Allows to model the part of a use case the user may see as optional
- Allows to model conditional subflows
- Allows to insert subflows at a certain point, governed by actor interaction

⇨ Capture the base use case

⇨ For every step ask

what could go wrong

how might this work out differently

⇨ Plot every variation as an extension of the use case

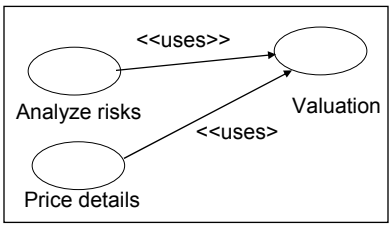
Example: extension points

- Capture deal
  1. Enter the user name & bank account
  2. Check that they are valid

  extension point: reenter data in case they are invalid
  3. Enter number of shares to buy & share ID
  4. Determine price
  5. Check limit
  6. Send order to NYSE
  7. Store confirmation number

Uses/Includes relationship

- Used to avoid describing
  the same flow of events several times, by putting the common behavior in a use case of its own

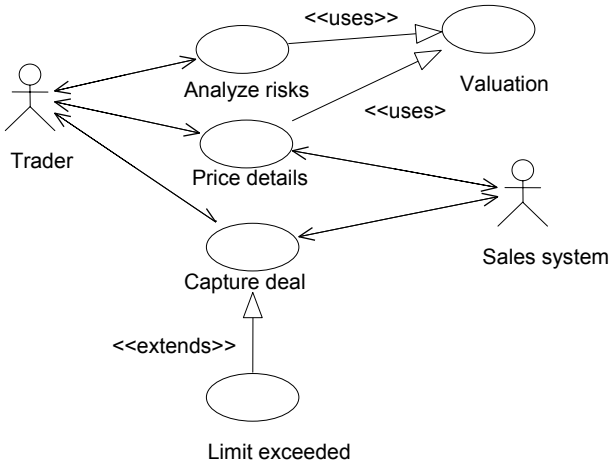


- Avoids copy-and-paste of parts of use case descriptions

Comparing extends/uses

- Different intent
  - extends
    - to distinguish variants
    - set of actors perform use case and all extensions
    - actor is linked to “base” case
  - uses/includes
    - to extract common behavior
    - often no actor associated with the common use case
    - different actors for “caller” cases possible

A use case diagram



Use Case Diagrams (Functional)

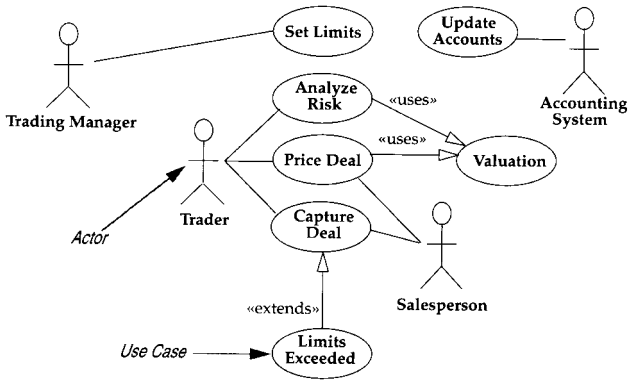


Figure 3-1: *Use Case Diagram*

Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Properties of use cases

- Granularity: fine or course
- Achieve a discrete goal
- Use cases describe externally required functionality
- Often: Capture user-visible function

When and how

- Requirements capture - first thing to do
- Use case: Every discrete thing your customer wants to do with the system
  - give it a name
  - describe it shortly (some paragraphs)
  - add details later

Class diagrams

- *Overview*
- Class diagram essentials
- Generalization

Class diagram

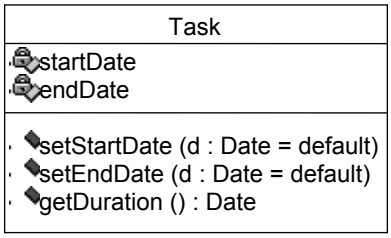
- Central for OO modeling
- Shows static structure of the system
  - Types of objects
  - Relationships
    - Association
    - Subtypes

Perspectives

- Conceptual
  - Shows concepts of the domain
  - Independent of implementation
- Specification
  - Interfaces of software (types)
  - Often: Best perspective
- Implementation
  - Structure of the implementation
  - Most often used

Class

- Set of objects
- Defines
  - name
  - attributes
  - operations



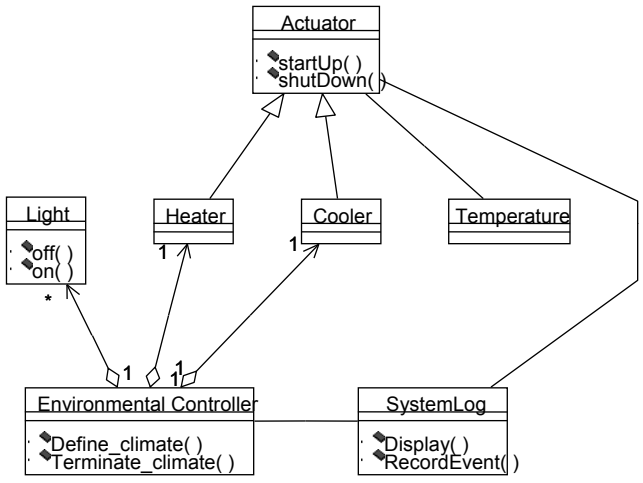
Class versus type

- OO type
  = protocol understood by an object
  = set of methods that are implemented
- Class =
  implementation oriented construct
  - implements one or more types
- Type: Used for specification

Association

- Relationship between instances of classes
  - A student is registered for a course
  - A professor is teaching the course

Class diagram example



Rules of thumb

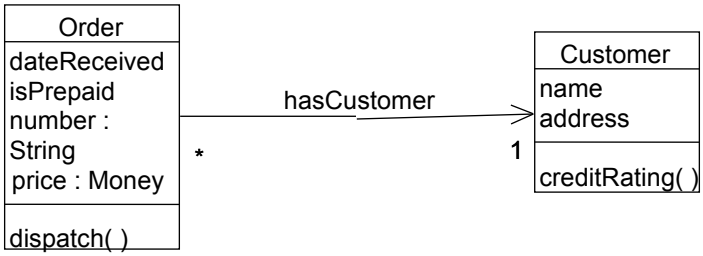
- One class can be part of several diagrams
- Diagrams shall illustrate specific aspects
  - Not too many classes
  - Not too many associations
  - Hide irrelevant attributes/operations
- Several iterations needed to create diagram

Class diagrams

- Overview
- *Class diagram essentials*
- Generalization

Association

- Relationship between classes

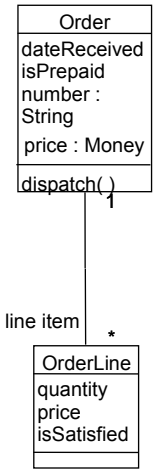


- Order comes from one customer
  Customer may make several orders

Naming associations

- Avoid meaningless names
  - associated_with
  - has
  - is_related_to
- Name is often a verb phrase
  - has_part
  - is_contained_in

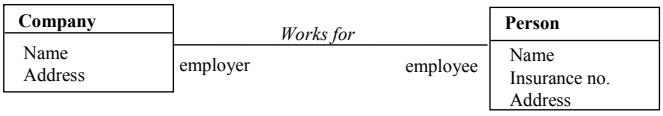
Roles



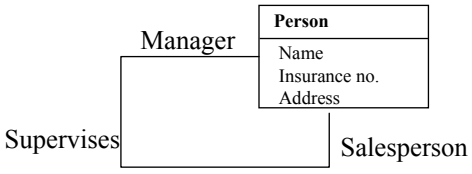
- Association has two roles
- Role is a direction on the association
- Role
  - Explicit labeled
  - Implicitly named after the target class

Role names

- Role = identifies one end of an association

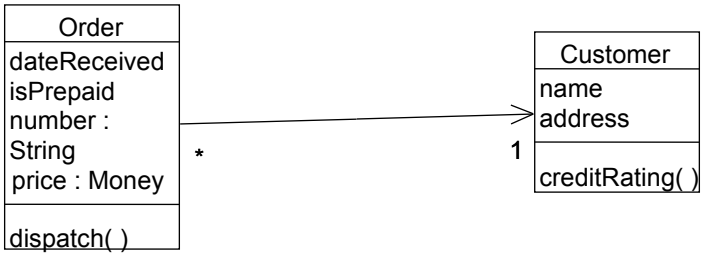


- Role name is obligatory for associations between objects of the same class



Multiplicity

- Indicates how many object can participate in the relationship

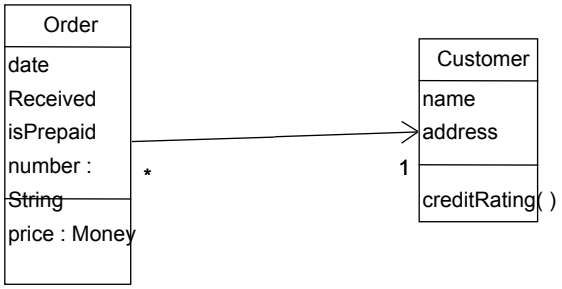


Multiplicity (2)

- *: 0..infinity
- 1: 1..1
- 0..1
- 1..100
- 2,4,5

Specification perspective

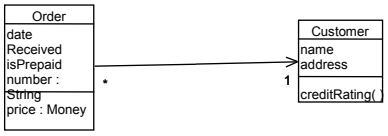
- Association represents responsibilities



- Method in Customer returning Orders
  Method in Order that returns the Customer that made the order

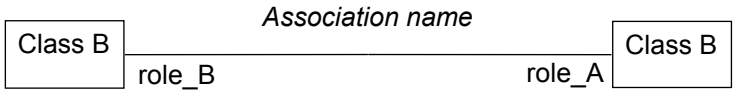
Navigability

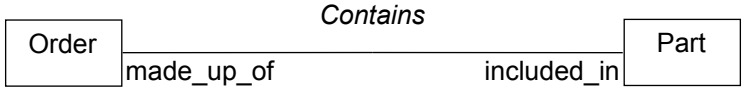
- Arrows indicate navigability



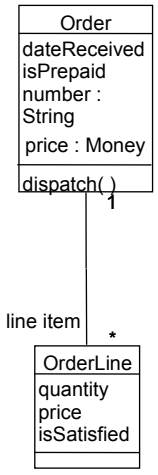
- Order has to be able to determine the Customer
- Customer does not know Orders
- Bi-directional association: Navigability in both directions (inverse roles)

Summary: Basic notation for associations





Naming conventions



- Naming conventions allow often to infer the names of messages from the diagram

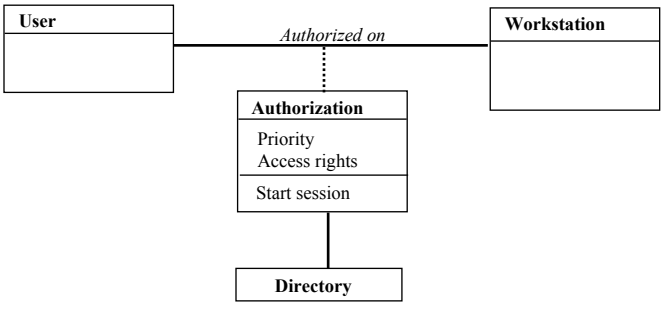
```
class Order {
   public Enumeration
   orderLines();
   public Customer customer();
   }
```

Example: Hockey statistics

Class

Association classes

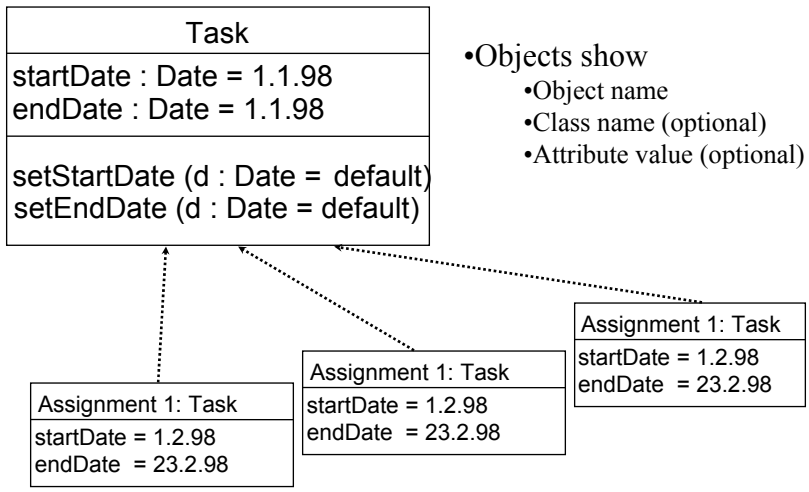
- Useful if
  - attributes don't belong to any one class but to the association



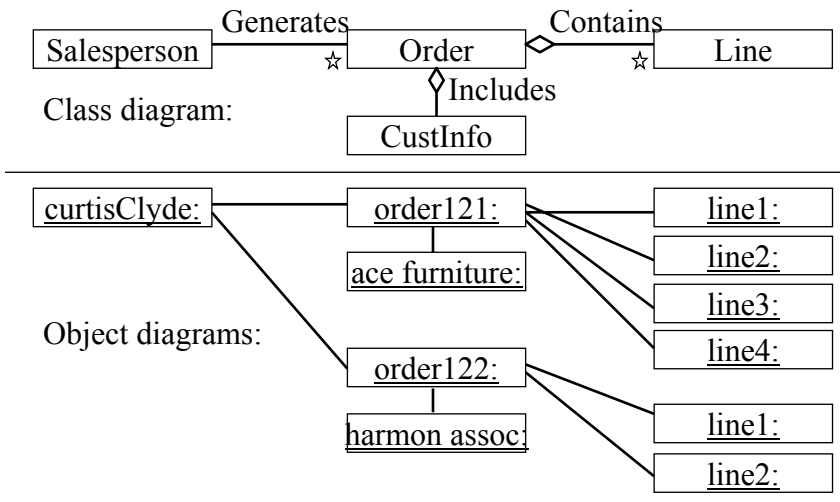
Contents

- *Attributes and operations*
- Aggregation
- Inheritance
- Interfaces and abstract classes
- Advanced association concepts
- When and how

Classes and objects



Example



Attributes

Customer
name address
creditRating

- Conceptual: Indicates that customer have names
- Specification: Customer can tell you his/her name and set it
- Implementation: An instance variable is available
- UML syntax:

  <attribute name>: <Data type>

Difference between attribute and association

- Conceptual perspective
  - not much of a difference!
- Specification/implementation perspective
  - Attribute stores values NOT references
    - no sharing of attribute values between instances!
- Often: Stores simple objects
  - Numbers, Strings, Dates, Money objects

Operations

- Processes that can be carried out on instances
- Correspond to messages of the class
- Conceptual perspective
  - principal responsibilities
- Specification perspective
  - public messages = interface of the class
- Normally: Don't show operations that manipulate attributes

UML syntax for operations

<visibility> <name> (<parameter list>) : <return-type-expression>

+ assignAgent (a : Agent) : Boolean

- visibility: public (+), protected (#), private (-)
  - Interpretation is language dependent
  - Not needed on conceptual level
- name: string
- parameter list: arguments (syntax as in attributes)
- return-type-expression: language-dependent specification

Types of operations

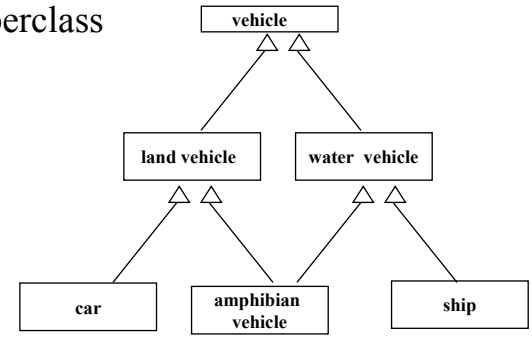
- *Query* = returns some value without modifying the class' internal state
- *Modifier* = changes the internal state
- Queries can be executed in any order
- Getting & setting messages
  - getting: query
  - setting: modifier

Contents

- Attributes and operations
- *Inheritance*
- Aggregation
- Interfaces and abstract classes
- Advanced association concepts
- When and how

Subclassing

- Class inherits features from (more than) one superclass



Subclassing

- Attributes & operations of an ancestor class are inherited to the subclass
- Extension: adding of new attributes or operations
- Restriction: additional restrictions on ancestor attributes

Perspectives

- Conceptual: Subset relationship
- Specification: Subtype conforms to supertype interface
- Implementation: Implementation inheritance, subclassing

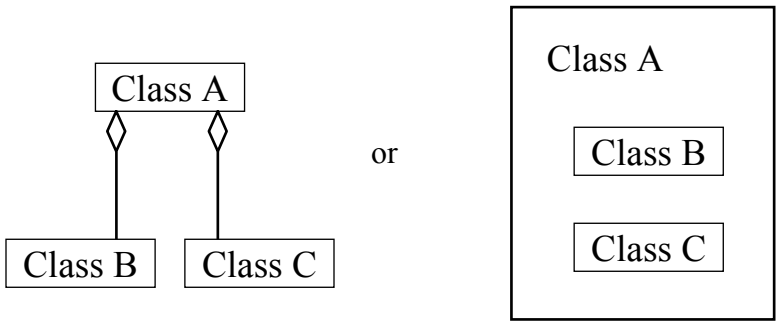
Contents

- Attributes and operations
- Inheritance
- *Aggregation*
- Interfaces and abstract classes
- Advanced association concepts
- When and how

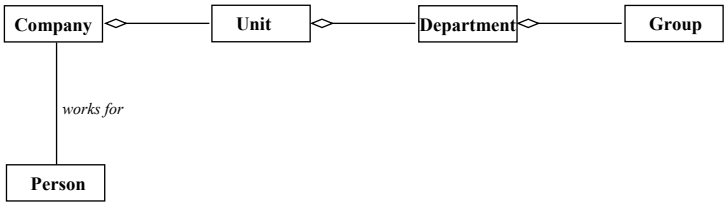
Aggregation

- Special form of association
- Components are *parts of* aggregated object
  - Car has an engine and wheels as its part
- Typical example:
  - parts explosion
  - organizational structure of a company

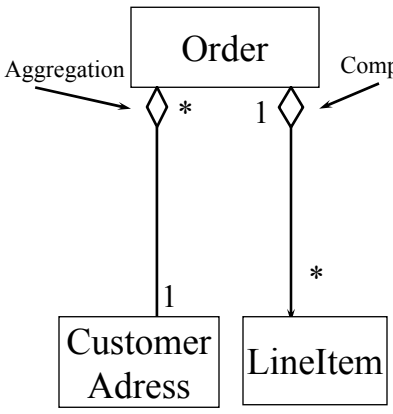
Notation for aggregation



Example: Aggregation



Aggregation and composition



- Composition
  - Components belong only to one whole
  - Parts live and die with the whole
    - cascading delete
    - also needed for 1..1 associations
  - The players can be aggregated to the Flames
    BUT
    they are not killed when the Flames disappear

Aggregation association

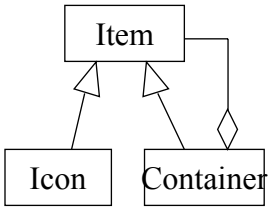
- Transitive
- Antisymmetric: Object may not be directly or indirectly part of itself

Recursion

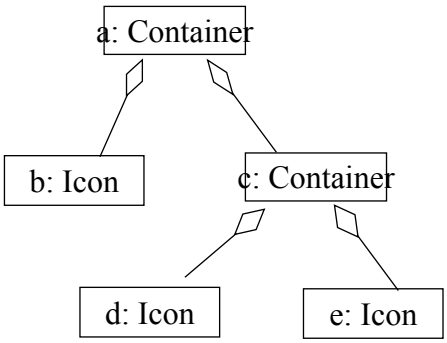
- Directed path of aggregation associations from a class to itself
- Variable aggregation: finite number of levels, number of parts variable (example: company)

Example: recursive aggregation

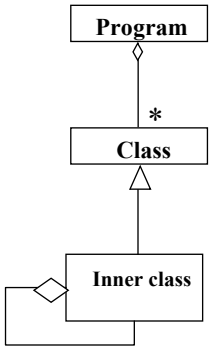
Class diagram:



Object diagram:



Example: Recursive aggregation

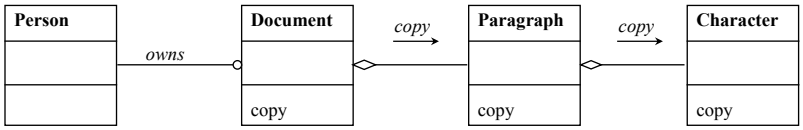


Rules for using aggregation

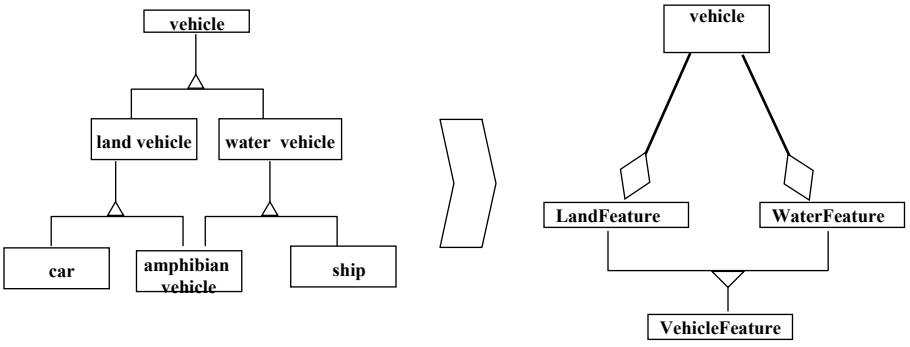
- Distinction between association and aggregation often rather matter of taste than difference in semantics
- Aggregation IS association
- Aggregate is inherently sum of its parts
- Chains of aggregate links may not form cycles
- Composition is appropriate when each part is owned by one object, part has not have an independent life from its owner

Chaining of operations

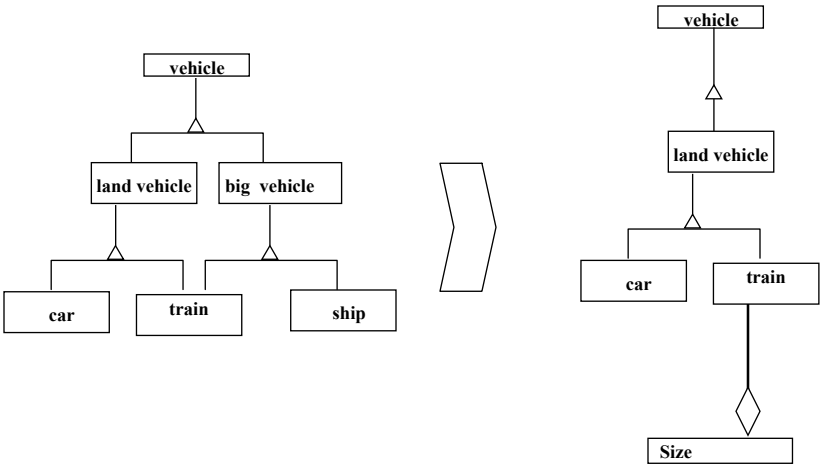
- Chaining: Applying an operation to a net of objects
- Often for: copy, save, redo, delete, print



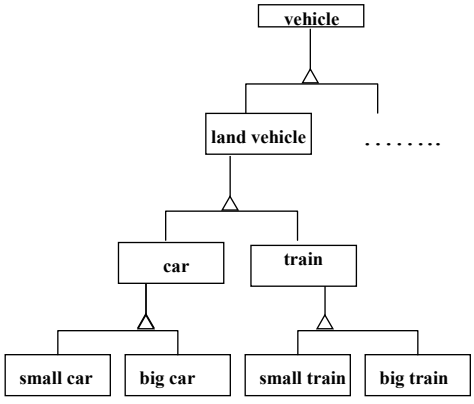
Delegation & aggregation



Most important feature & aggregation



Generalization based on different dimensions

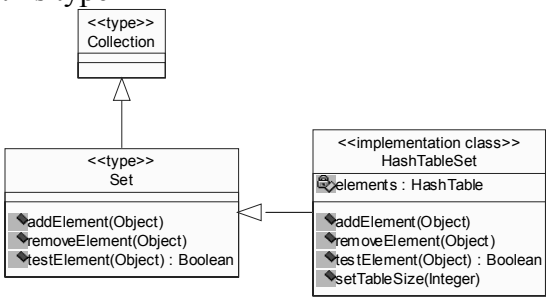


Contents

- Attributes and operations
- Inheritance
- Aggregation
- *Types, interfaces and abstract classes*
- Advanced association concepts
- When and how

OO types

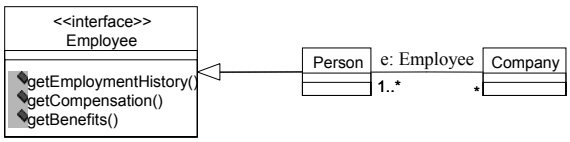
- Stereotype <<type>> specifies
  - domain of objects
  - operations (not their implementation) applicable to the objects of this type



- **Stereotype <<implementation class>>**
  physical data structures and methods of an object

Types and Roles

- interfaces that belong to a class represent different roles
- You can explicitly state the role a class presents to another class:



Static and dynamic types

- Static types: the type of an object doesn't change over time, e.g. classes
- Dynamic types: object can gain and lose types during lifetime

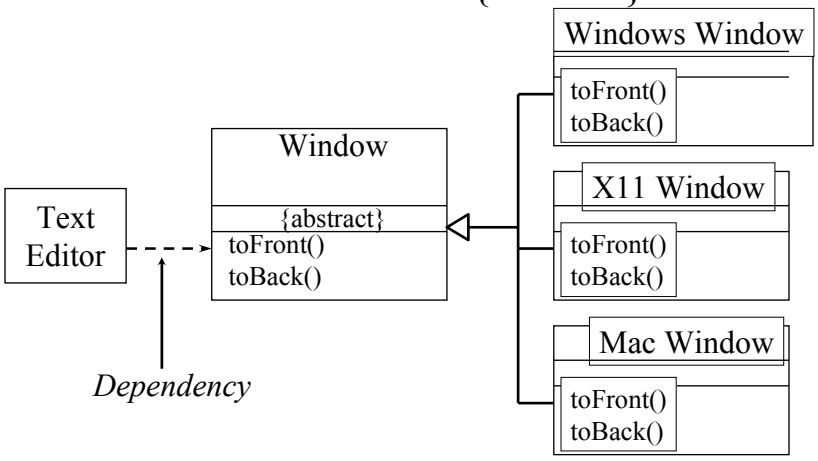
- Example: Candidate, Employee, Retiree

Abstract class

- has no instances
- organizes attributes & operations
- often: facilitates code reuse
- abstract operation: implementation in concrete subclasses
- can contain concrete implementations

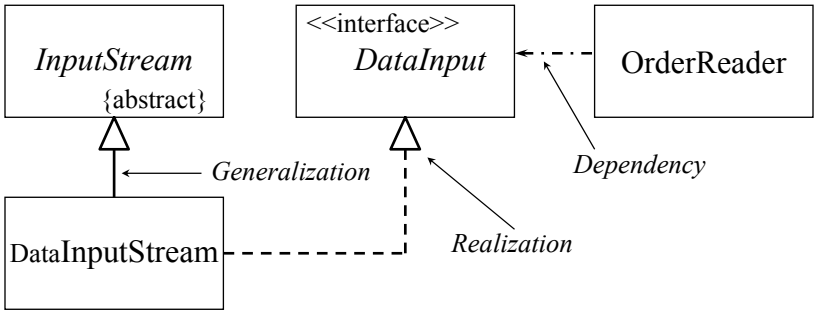
Abstract class in UML

- Name in italic and/or {abstract} constraint

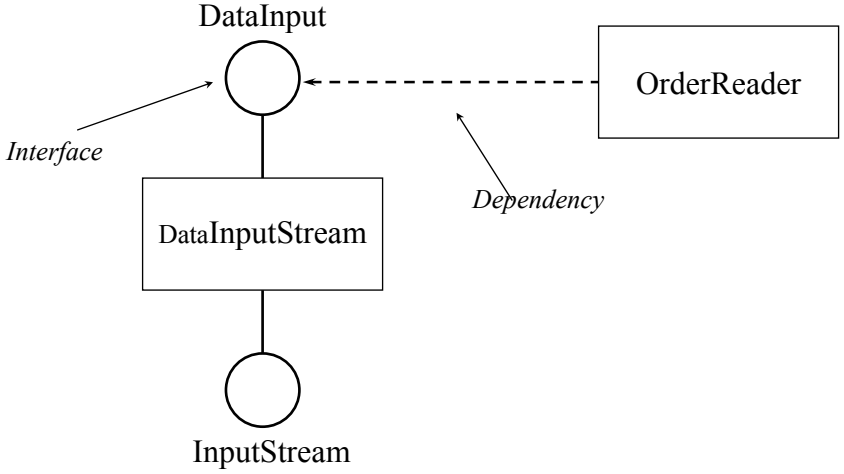


Interfaces in UML (1)

- Stereotype <<interface>>
- Lollipops

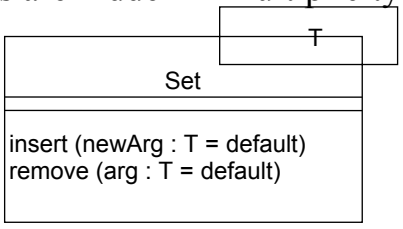


Interfaces in UML (2)



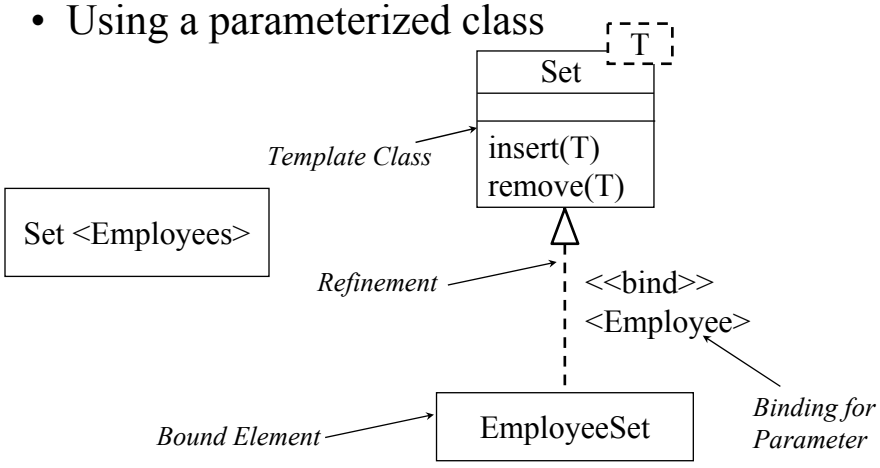
Parameterized classes

- Parameterized class = template
- Often used for collections in typed languages
- Not needed in conceptual modeling
  - Collections are hidden in multiplicity



Bound element

- Using a parameterized class



Contents

- Attributes and operations
- Inheritance
- Aggregation
- Interfaces and abstract classes
- *Advanced association concepts*
- When and how

Constraints

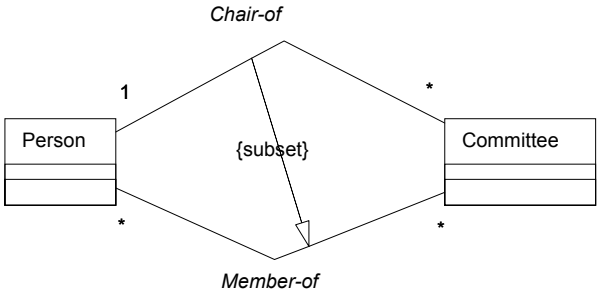
- Basic constructs specify important constraints
  - but: can not capture everything
- Additional constraints: in braces { }

  {UofC has always to be better than UofA}

  {immutable}

  {read only}

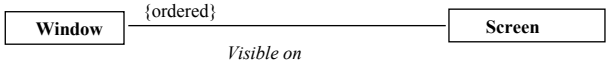
Example



Collections for multi-valued roles

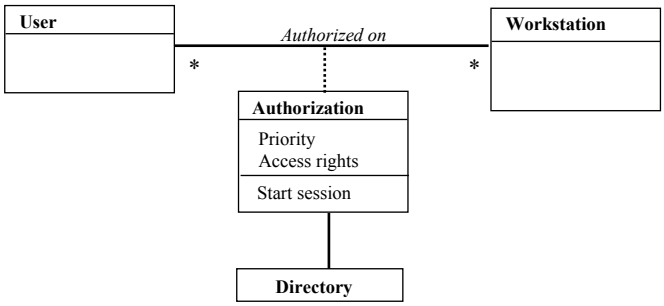
- Multiplicity > 1
  - Set
    - no target object appears more than once
    - not ordered
- Add constraint to change that

{ordered} {bag}
{ordered bag} {hierarchy}
{dag}

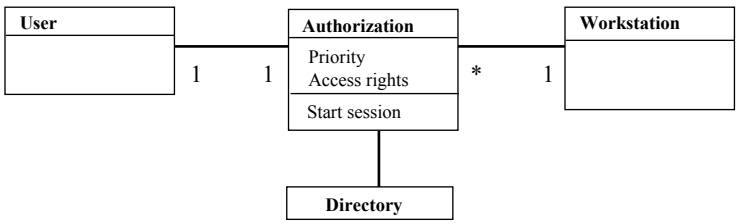


Association classes

- Useful if
  - attributes don't belong to any one class but to the association



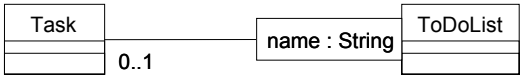
Remodeling: association classes



Qualified associations (1)

- UML equivalent for Hashtable

o1	44
o2	56
o3	87
o4	99



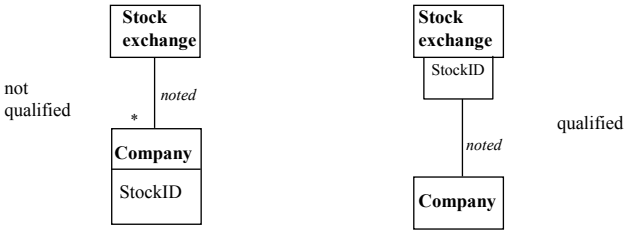
  - Within a ToDoList, you mustn't have two tasks with the same name

```
class ToDoList {
  public Task getTask(String name);
  public void addTask(String name, Task aTask);
  …}
```

  - Multiplicity *: Multiple task with one name

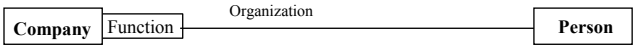
Qualified association (2)

- Improves semantic accuracy
- Makes navigation paths understandable



Qualified association (3)

- Qualification splits a set of objects in disjunctive parts

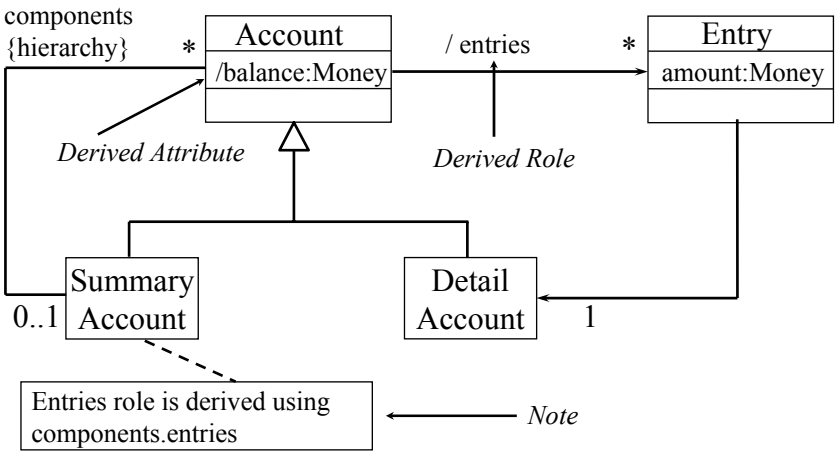


ABC Inc.	President	Roger Rabbit
ABC Inc.	Vice President Finances	Joe Savemoney
ABC Inc.	Member of board	John Walker
ABC Inc.	Member of board	Susi Sanssouci
ABC Inc.	Member of board	Karl Eichbaum
XYZ Inc.	President	Donald Duck

Derived associations and attributes (1)

- Calculated based on other attributes and associations
- Specification: Shows constraint not what is stored and what is calculated

Derived associations and attributes (2)



Class Diagram (Structural)

- Use: Describe the static structure of a system
  - Hierarchy
  - Containment
  - Inheritance
  - Calling
  - Object Types

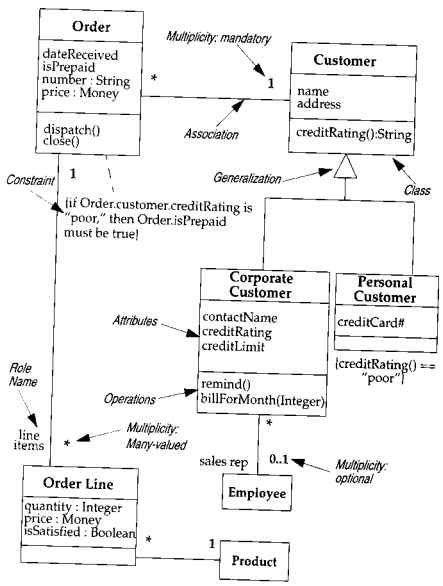


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Contents

- Attributes and operations
- Inheritance
- Aggregation
- Interfaces and abstract classes
- Advanced association concepts
- *When and how*

When to use class diagrams

- Class diagrams are the backbone of OO development approaches
- Don't use all the notations
  - start with simple stuff
- Take the perspective into account
  - not to many details in analysis
  - specification often better than implementation
- Concentrate on key areas
  - better few up-to-date diagrams than many obsolete models

Creating a class diagram

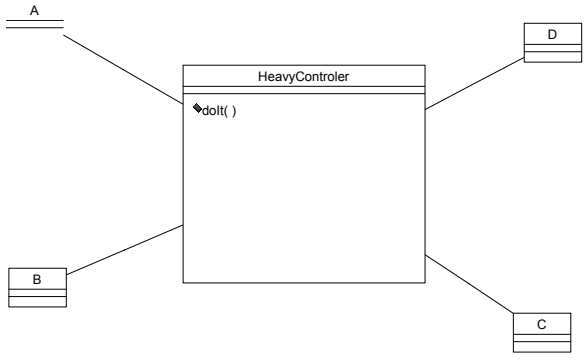
- Start simple
  - major classes & obvious associations
- Then add
  - Attributes
  - Multiplicity
  - Operations

Rules of thumb

- One class can be part of several diagrams
- Diagrams shall illustrate specific aspects
  - Not too many classes
  - Not too many associations
  - Hide irrelevant attributes/operations
- Several iterations needed to create diagram

Avoid “Heavy” classes

- Controller does everything
- Other classes: Data encapsulation only



Contents

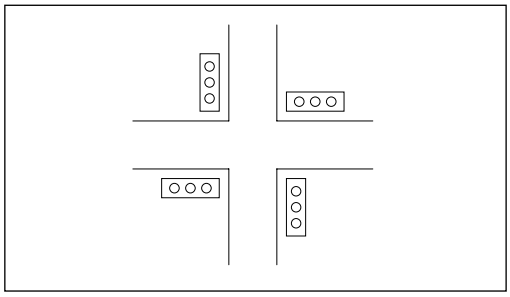
- *State diagrams: an example*
- Interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams

Example

- A zoo consists of a set of cages.
- Every cage is the home of at least 2 animals.
- Cages are located besides each other.
- Every cage has at most one left neighbor and at most one right neighbor.
- Animals can be reptiles, insects, and mammals.
- Mammals are elephants, monkeys, and tigers.
- Monkeys eat bananas.
- Tigers prefer meat.

Traffic lights

- Develop a state transition diagram for the 4 traffic lights at a crossing. Make sure that the lights never allow traffic to move east to west (or west to east) at the same time as they allow traffic to move north to south (or south to north). Give meaningful names to all state transitions.



Contents

- State diagrams: an example
- *Interaction diagrams*
  - *Sequence diagrams*
  - Collaboration diagrams

Interaction diagrams

- describe how groups of objects interact
- typically describe the scenario of a single use case
- show
  - example objects
  - messages between them
  - timeline

Sequence diagrams

- shows object interactions arranged in time sequence
  - objects (and classes)
  - message exchange to carry out the scenarios functionality
- time line

Objects in UML

- Rectangle
- Name (specific or general) of object is underlined
  - name
  - name & class
  - class (anonymous object)

Object Name

History 101-Section 2

Object Name and Class

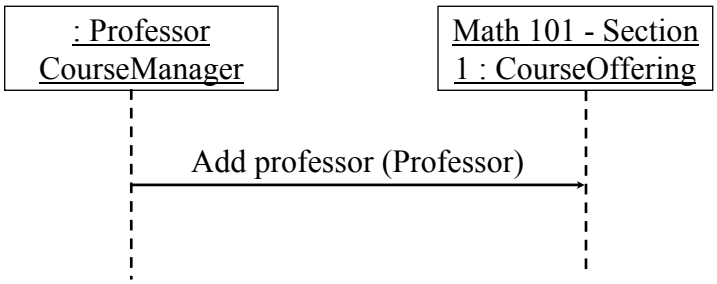
History 101-Section 7: CourseOffering

Class Name

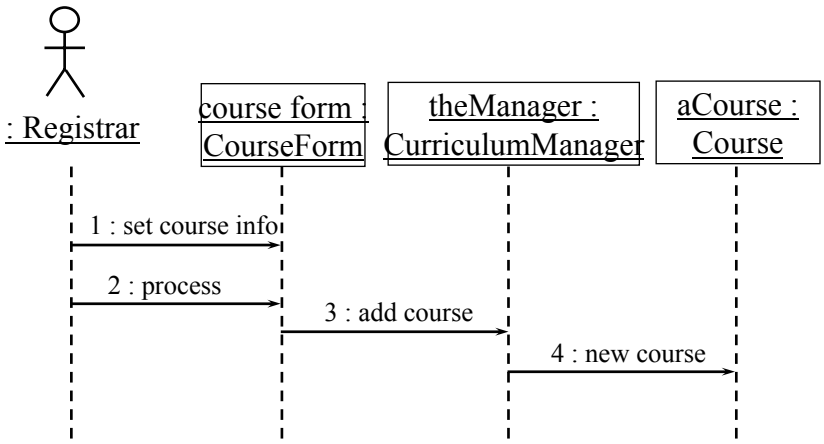
: CourseOffering

Timelines

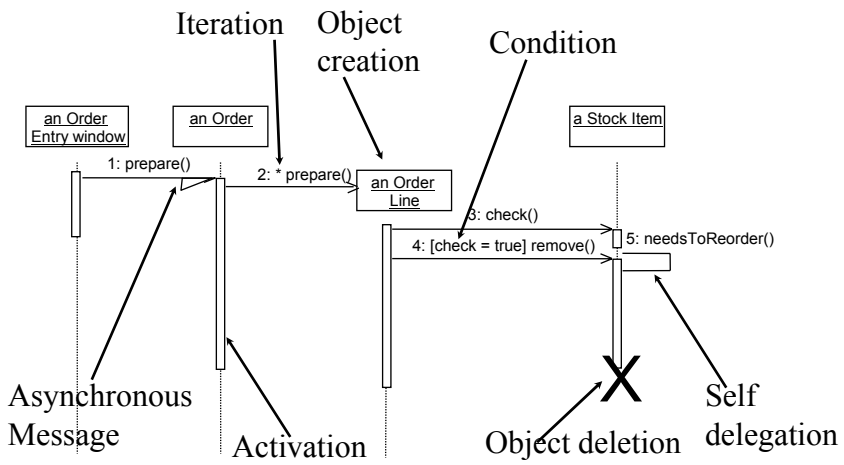
- Messages point from client to supplier



Example: Sequence diagram



Sequence diagrams: More details



Asynchronous messages

- Do not block the caller
- Can do 3 things:
  - Create a new thread
  - Create a new object
  - Communicate with a thread that is already running

Boundary classes

- Handle communication between system and outside world
  - e.g. user interface or other system
- Boundary classes in interaction diagrams:
  - capture interface requirements
  - do NOT show how the interface will be implemented

Complexity and sequence diagrams

- KISS

  = keep it small and simple
- Diagrams are meant to make things clear
- Conditional logic
  - simple: add it to the diagram
  - complex: draw separate diagrams

Contents

- State diagrams: an example
- *Interaction diagrams*
  - Sequence diagrams
  - *Collaboration diagrams*

Sequence Diagram (Behavioral)

- Use: Describing behavior across several objects of a use-case or scenario

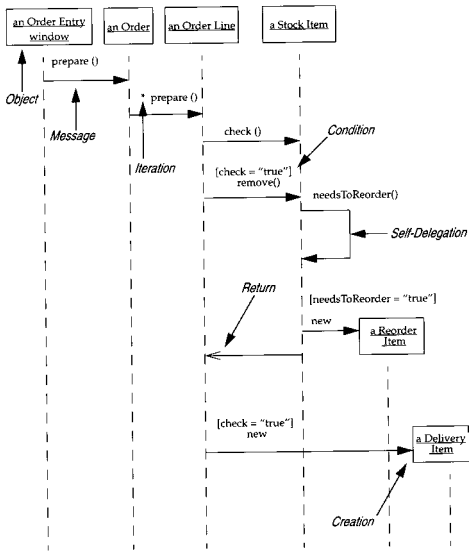


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Sequence Diagram with Concurrency

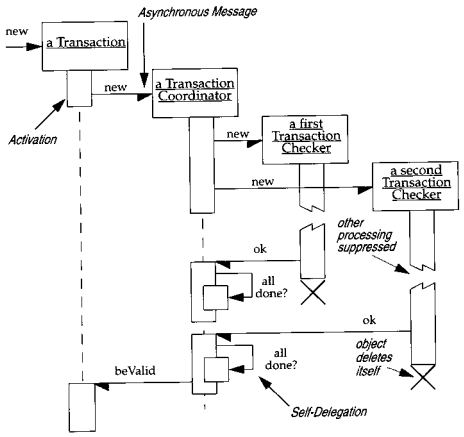
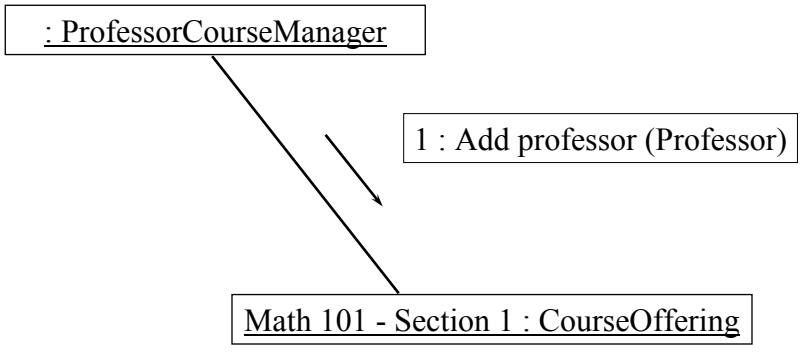


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

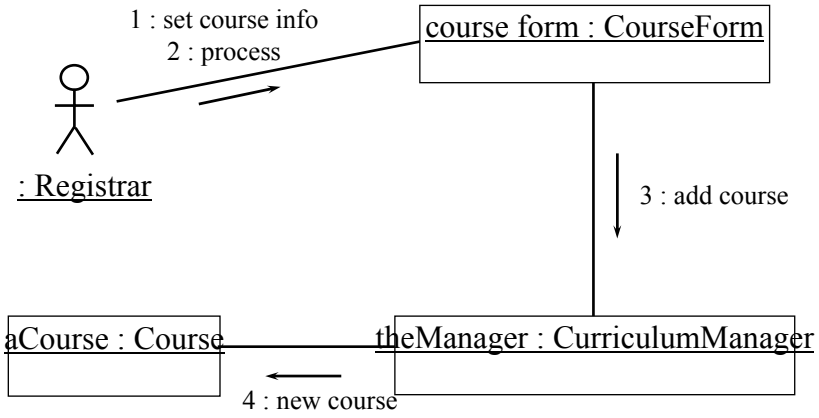
Collaboration diagrams

- Show objects and messages
- Sequence of messages determined by numbering
  - 1, 2, 3, 4, …..
  - 1, 1.1, 1.2, 1.3, 2, 2.1, 2.1.1, 2.2, 3
    (shows which operation calls which other operation)

Collaboration diagram basics



Collaboration diagram example



Collaboration Diagram (Behavioral)

- Use: Describing behavior across several objects of a use-case or scenario

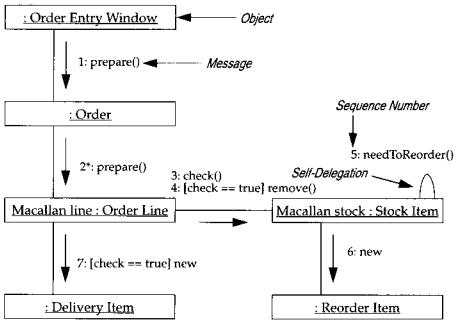


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

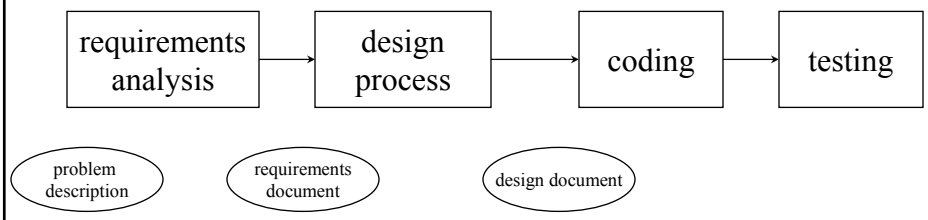
Comparing sequence & collaboration diagrams

- Sequence of messages more difficult to understand in collaboration diagrams
- Layout of collaboration diagrams may show static connections of objects
- Complex control is difficult to express

State diagrams

- *Design document*
- State diagrams

Main processes of the team assignment

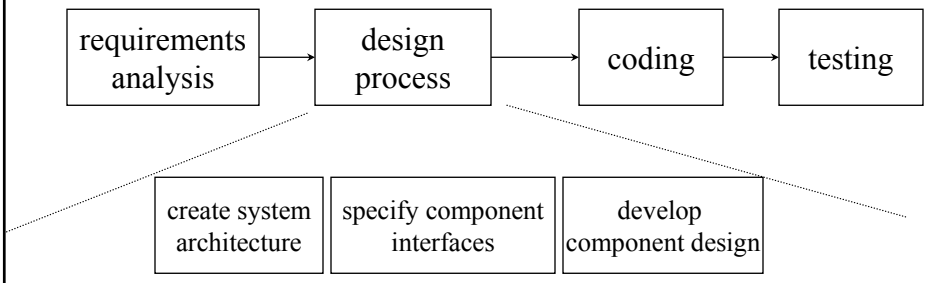


Used technique:
Use cases

Used technique:
UML class diagrams,
UML sequence diagrams,
UML activity diagrams
UML state diagrams

Used language:
Java

Refined design processes of the team assignment



Techniques:
UML class diagrams,
UML sequence diagrams,
UML state diagrams

Design document

- System architecture: class diagrams
- Component interfaces
  - class diagrams (interfaces, types)
  - sequence diagrams
- Component design
  - class diagrams
  - state diagrams
  - sequence diagrams

Design document - aim

- Basis for implementation
- provides different views
  - other developers: architecture, component interfaces
  - implementation: straightforward
- Allows quick overview over the system structure and main design decisions
- Allows developers to work in parallel

Diagram notations

- State diagrams
  - describe the behavior of objects
- Activity diagrams
  - describe the flow of work
  - parallel processing
- Sequence diagrams
  - describe time ordering of messages
- Deployment diagrams
  - physical relationship of software and hardware

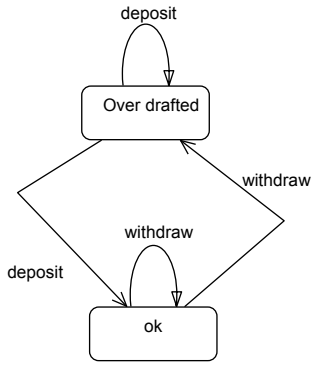
State diagrams

- Design document
- *State diagrams*

State diagrams

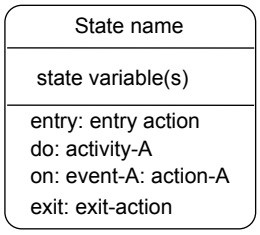
- State diagram: Shows the behavior of *one* object
  - how does it change its state based on the messages it receives
  - narrowly focused, fine-grained
- Other names
  - State transition diagram
  - Harel diagram (statecharts)

State diagrams (2)



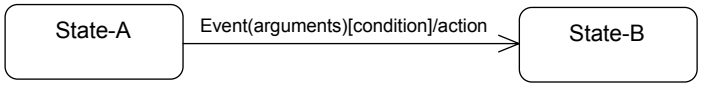
- State: condition/situation during lifetime of an object
- State transition: relationship indicating a state change
  - atomic & non-interruptible
- Action:
  - atomic & non-interruptible

State notation (1)



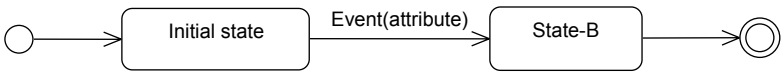
- Substates: disjoint/concurrent
- Entry/exit actions
  - entry: an action that is performed on entry to the state
  - exit: an action performed on exiting the state
- do: an ongoing activity performed while in the state (example: display window)
  - interruptible
- on: an action performed as a result of a specific event

Transition notation (2)



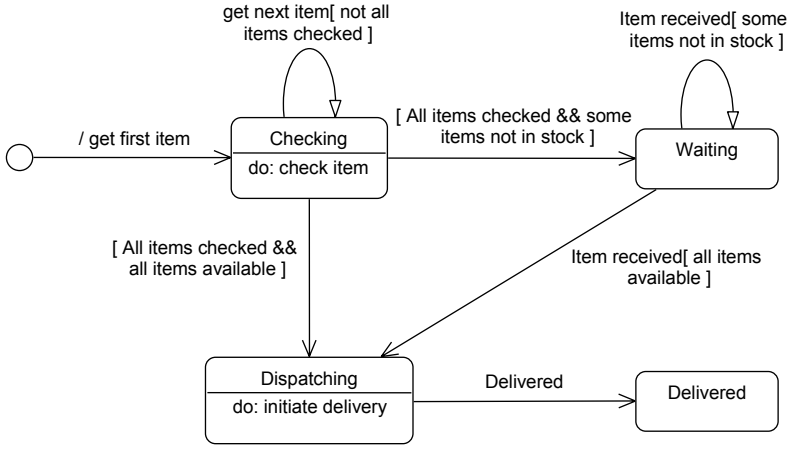
- Event: significant occurrence that has a location in time and space
  - triggers the transition
  - signals, calls, passing of time, change in state
- Guard condition:
  - Transition only eligible to fire when guard evaluates to true
  - Guards of transition exiting one state are mutually exclusive
- Action: executable atomic computation

State diagram notation (3)



- Start state
  - No event triggers allowed
  - branch conditions allowed
  - may not remain in start states
- End state
  - Top level end state terminates a state machine

State transitions for an order



State Diagram (Behavioral)

- Use: Describing behavior of a single object
- Hint: the entire system is a single top-level object

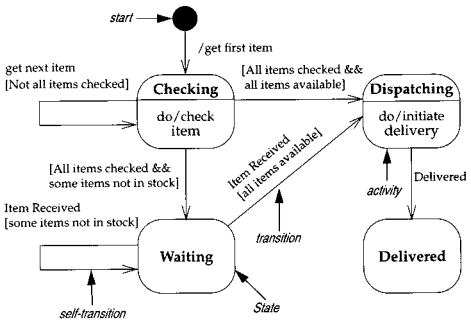
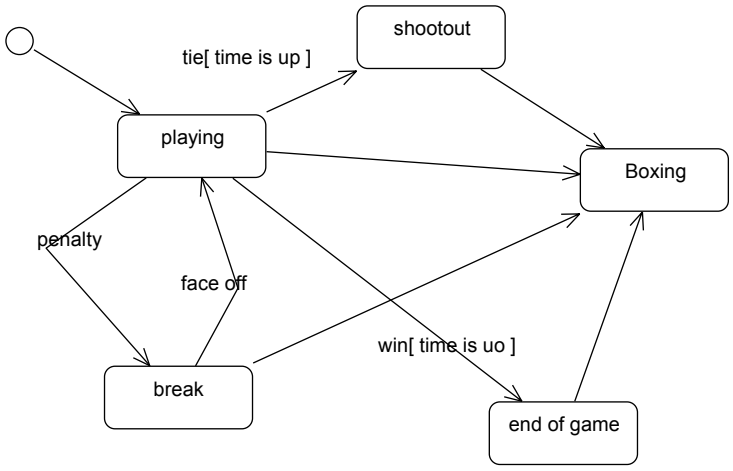


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

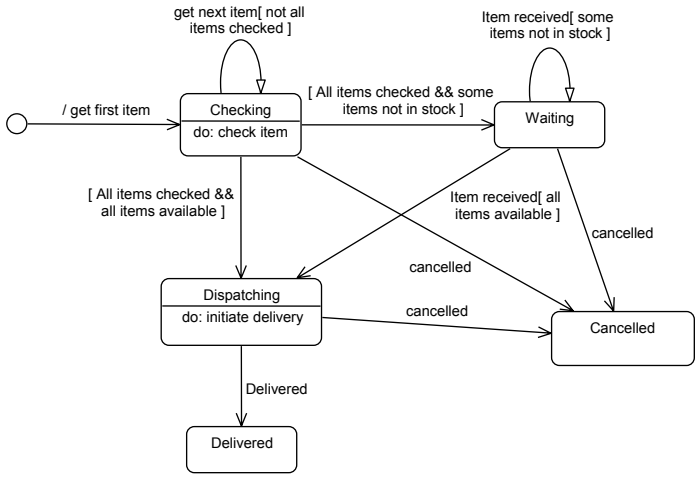
States of a hockey game



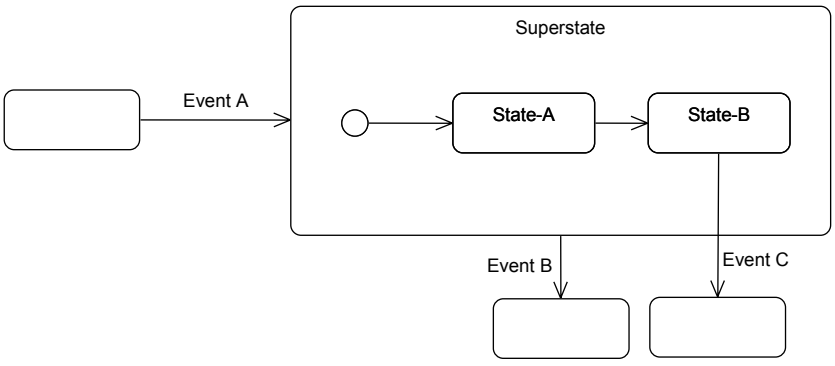
Problem: Cancel the order

- Want to be able to cancel an order at any time
- Solutions
  - Transitions from every state to state “cancelled”
  - Superstate and single transition

Transitions to “cancelled”



State diagram notation (4)



State Diagram with Substates

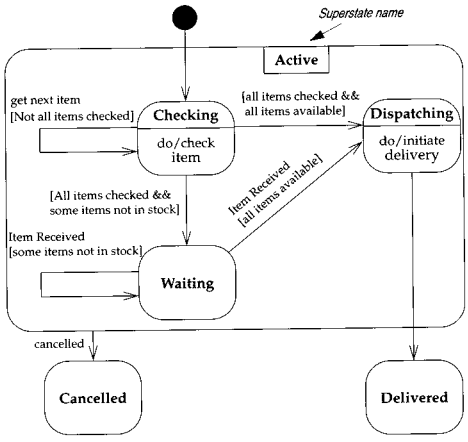
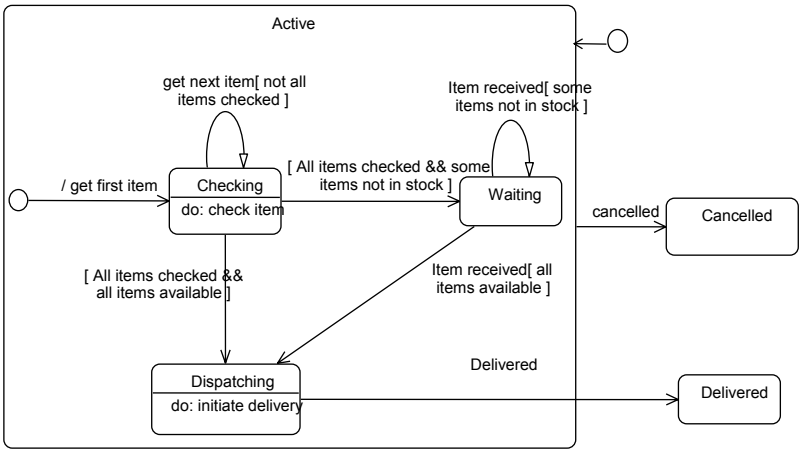
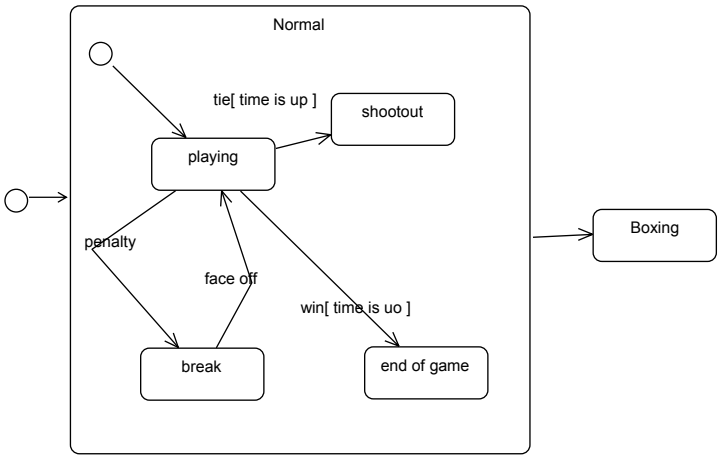


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Superstate



Hockey example with superstate



Some remarks

- Only one initial state may occur (directly) within a composite state
- End state represents completion of a composite
- End state triggers transition with composite as source

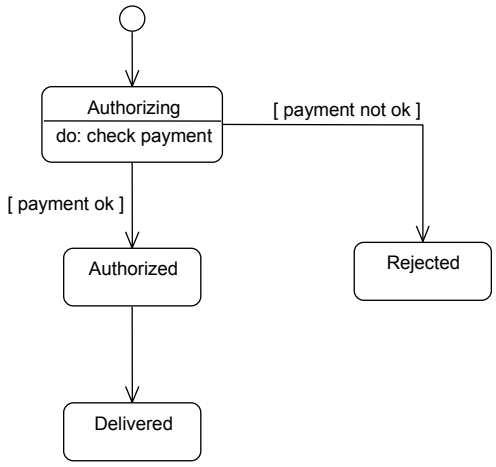
Orthogonal components and concurrency

- Unrelated components of objects
  - combinatorial number of states
- Example: Car states
  - engine (started, stopped)
  - doors (open, closed)
- What happens when we add one component?
  - seat belt (fastened, open)

4 car states:
started_open
started_closed
stopped_open
stopped_closed

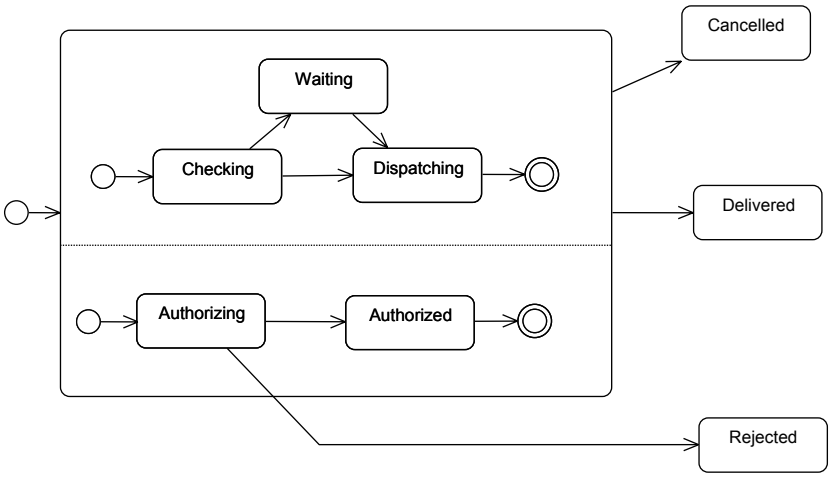
8 car states:
started_open_open
started_closed_open
stopped_open_open
stopped_closed_open
started_open_fastened
started_closed_fastened
stopped_open_fastened
stopped_closed_fastened

Example: Payment authorization in class Order



2 parallel processes:
- authorization
- order handling

Concurrent state diagram for the class Order



State Diagram with Concurrency

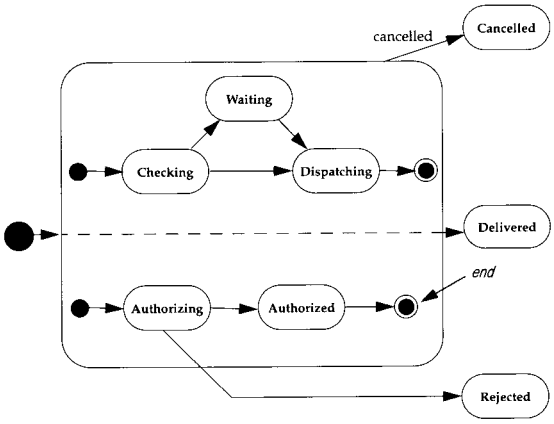


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Rules of thumb

- Not every class needs a state diagram
- Often: State diagram not very complex
- State diagrams are often used for UI and control objects
- Not to many concurrent sets of behavior occurring in a single object (in that case: split into separate objects)

Activity Diagrams (Behavioral)

- Use: Understanding Work-Flow
- Use: Analyzing Use-Case
- Use: Dealing with Multi-Threading
- No: Analyzing Object Collaboration
  - Use Sequence or Collaboration Diagrams
- No: Analyzing Object Behavior
  - Use State Diagram

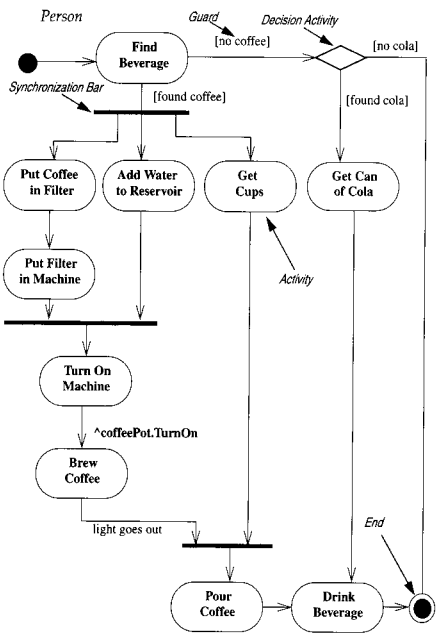


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Activity Diagrams with Swim Lanes

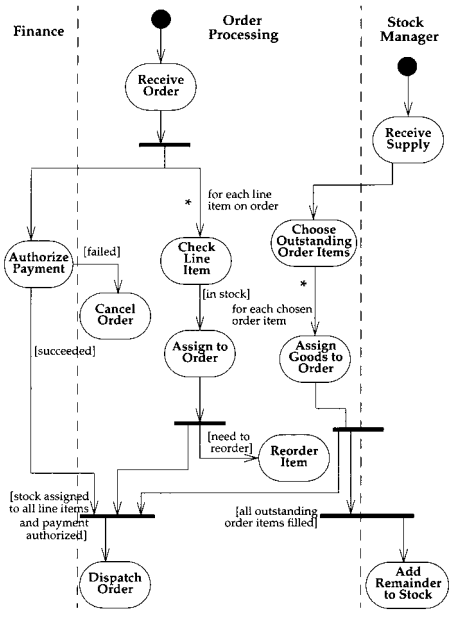


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Package Diagram (Structural)

- Use: Large-Project Structures

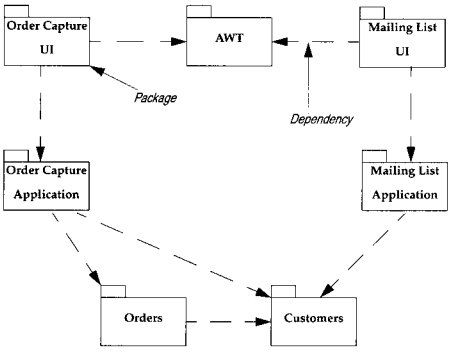


Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley

Deployment Diagram

- Use: Describing System/Hardware/ Software Relationships