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

Market Analysis

Financial Officer
Use cases: terms and concepts

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

Financial Officer

• 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 deal

Limit exceeded

<<extends>>

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

Trader

Analyze risks

Price details

Capture deal

Valuation

Sales system

<<uses>>

<<extends>>

Limit exceeded
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

<table>
<thead>
<tr>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDate</td>
</tr>
<tr>
<td>endDate</td>
</tr>
<tr>
<td>setStartDate (d : Date = default)</td>
</tr>
<tr>
<td>setEndDate (d : Date = default)</td>
</tr>
<tr>
<td>getDuration () : Date</td>
</tr>
</tbody>
</table>
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

- **Light**
  - `off()`
  - `on()`

- **Heater**

- **Cooler**

- **Environmental Controller**
  - `Define_climate()`
  - `Terminate_climate()`

- **SystemLog**
  - `Display()`
  - `RecordEvent()`

- **Actuator**
  - `startUp()`
  - `shutDown()`
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

<table>
<thead>
<tr>
<th>Order</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateReceived</td>
<td>name</td>
</tr>
<tr>
<td>isPrepaid</td>
<td>address</td>
</tr>
<tr>
<td>number : String</td>
<td>creditRating( )</td>
</tr>
<tr>
<td>price : Money</td>
<td></td>
</tr>
<tr>
<td>dispatch( )</td>
<td></td>
</tr>
</tbody>
</table>

• 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

<table>
<thead>
<tr>
<th>Order</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateReceived</td>
<td>name</td>
</tr>
<tr>
<td>isPrepaid</td>
<td>address</td>
</tr>
<tr>
<td>number: String</td>
<td>creditRating()</td>
</tr>
<tr>
<td>price: Money</td>
<td></td>
</tr>
</tbody>
</table>

dispatch()
Multiplicity (2)

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

Class B

Association name

role_B

role_A

Class B

Order

Contains

made_up_of

included_in

Part
Naming conventions

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

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

• Useful if
  – attributes don’t belong to any one class but to the association

![Diagram showing association between User, Authorized on, Authorization, Priority, Access rights, Start session, and Directory to Workstation]
Contents

• Attributes and operations
• Aggregation
• Inheritance
• Interfaces and abstract classes
• Advanced association concepts
• When and how
Classes and objects

<table>
<thead>
<tr>
<th>Task</th>
<th>Assignment 1: Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDate : Date = 1.1.98</td>
<td>startDate = 1.2.98</td>
</tr>
<tr>
<td>endDate : Date = 1.1.98</td>
<td>endDate = 23.2.98</td>
</tr>
</tbody>
</table>

• Objects show
  • Object name
  • Class name (optional)
  • Attribute value (optional)
Example

Class diagram:

Object diagrams:
Attributes

- 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

Class A

Class B
Class C

or

Class A

Class B
Class C
Example: Aggregation

Person works for Company

Company <-> Unit <-> Department <-> Group
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

Program

Class

Inner class
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

- Vehicle
  - Land vehicle
    - Car
    - Train
  - Big vehicle
    - Ship

- Vehicle
  - Land vehicle
    - Car
    - Train
  - Size
Generalization based on different dimensions

vehicle

land vehicle

- car
  - small car
  - big car
- train
  - small train
  - big train
Contents

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

![Diagram showing the relationship between Text Editor, Window, X11 Window, and Mac Window with toFront() and toBack() methods.]
Interfaces in UML (1)

- Stereotype <<interface>>
- Lollipops

![Diagram showing generalization and realization relationships between classes](image-url)
Interfaces in UML (2)

Diagram showing relationships between interfaces and classes:
- **DataInputStream**
- **InputStream**
- **DataInput**
- **OrderReader**

Relationships:
- Interface: DataInput
- Dependency: OrderReader
Parameterized classes

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

```
Set

T

insert (newArg : T = default)
remove (arg : T = default)
```
Bound element

• Using a parameterized class

Set <Employees>

Template Class

Set

insert(T)
remove(T)

Refinement

<<bind>>
<Employee>

Binding for Parameter

EmployeeSet

Bound Element
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

Chair-of

1

{subset}

Member-of

Person

Committee
Collections for multi-valued roles

- Multiplicity > 1
  - Set
    - no target object appears more than once
    - not ordered

- Add constraint to change that

```plaintext
{ordered}       {bag}
{ordered bag}   {hierarchy}
{dag}           {ordered}
```

Window → Screen

Visible on
Association classes

- Useful if
  - attributes don’t belong to any one class but to the association
Remodeling: association classes

- User
  - Authorization
    - Priority
    - Access rights
    - Start session
  - Directory
- Workstation
Qualified associations (1)

- UML equivalent for Hashtable

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>44</td>
</tr>
<tr>
<td>o2</td>
<td>56</td>
</tr>
<tr>
<td>o3</td>
<td>87</td>
</tr>
<tr>
<td>o4</td>
<td>99</td>
</tr>
</tbody>
</table>

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

```java
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

Diagram:

- Stock exchange
- Company
- StockID

Not qualified

Qualified

Stock exchange

Company

StockID

Noted

Noted
Qualified association (3)

- Qualification splits a set of objects in disjunctive parts

<table>
<thead>
<tr>
<th>Company</th>
<th>Function</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC Inc.</td>
<td>President</td>
<td>Roger Rabbit</td>
</tr>
<tr>
<td>ABC Inc.</td>
<td>Vice President Finances</td>
<td>Joe Savemoney</td>
</tr>
<tr>
<td>ABC Inc.</td>
<td>Member of board</td>
<td>John Walker</td>
</tr>
<tr>
<td>ABC Inc.</td>
<td>Member of board</td>
<td>Susi Sanssouci</td>
</tr>
<tr>
<td>ABC Inc.</td>
<td>Member of board</td>
<td>Karl Eichbaum</td>
</tr>
<tr>
<td>XYZ Inc.</td>
<td>President</td>
<td>Donald Duck</td>
</tr>
</tbody>
</table>
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)

Entries role is derived using components.entries
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
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
Timelines

• Messages point from client to supplier

: Professor CourseManager

Math 101 - Section 1 : CourseOffering

Add professor (Professor)
Example: Sequence diagram

: Registrar

1 : set course info
2 : process

theManager : CurriculumManager
3 : add course

aCourse : Course
4 : new course
Sequence diagrams: More details

- **Iteration**
  - 1: prepare()
  - 2: *prepare()
- **Object creation**
  - an Order Line
  - 3: check()
  - [check = true] remove()
- **Condition**
  - a Stock Item
  - 5: needsToReorder()
- **Asynchronous Message**
  - Activation
  - Self delegation

- **Object deletion**

Additional notes:
- Self delegation
- Object deletion
- Sequence diagrams: More details
- Activation
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
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

: ProfessorCourseManager

1 : Add professor (Professor)

Math 101 - Section 1 : CourseOffering
Collaboration diagram example

1: set course info
2: process

Registrar

course form : CourseForm

3: add course

aCourse : Course

theManager : CurriculumManager

4: new course
Collaboration Diagram (Behavioral)

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

![Collaboration Diagram](image)

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

• 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

<table>
<thead>
<tr>
<th>State name</th>
<th>State variable(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry: entry action</td>
<td>state variable(s)</td>
</tr>
<tr>
<td>do: activity-A</td>
<td>entry: entry action</td>
</tr>
<tr>
<td>on: event-A: action-A</td>
<td>do: activity-A</td>
</tr>
<tr>
<td>exit: exit-action</td>
<td>on: event-A: action-A</td>
</tr>
</tbody>
</table>
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

/ get first item

Checking
  do: check item

[ All items checked && all items available ]

Waiting
  Item received[ some items not in stock ]

Dispatching
  do: initiate delivery

[ All items checked && some items not in stock ]

Delivered
  [ All items checked && all items available ]

Delivered
  Item received[ all items available ]

get next item[ not all items checked ]
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

- Playing
  - Face off
  - Penalty
  - Tie [time is up]
- Break
- Boxing
  - Shootout
  - Win [time is up]
  - End of 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”

- **Checking**
  - do: check item
  - / get first item
  - [ All items checked && all items available ]
  - [ All items checked && some items not in stock ]

- **Dispatching**
  - do: initiate delivery
  - Delivered
  - [ All items checked && some items not in stock ]
  - [ All items checked && all items available ]
  - cancelled

- **Waiting**
  - Item received[ some items not in stock ]
  - Item received[ all items available ]
  - cancelled

- **Cancelled**
  - cancelled

Delivered
State diagram notation (4)
State Diagram with Substates

Diagram: *UML Distilled*, Martin Fowler, Kendall Scott, 1997, Addison-Wesley, Copyright 1997, Addison-Wesley
Hockey example with superstate

- Normal
  - tie[ time is up ]
  - playing
    - penalty
    - face off
    - win[ time is up ]
  - end of game
  - shoot out

- Boxing

- Break
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)

  8 car states:
  started_open_open
  started_closed_open
  stopped_open_open
  stopped_closed_open

  4 car states:
  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

- Checking
- Dispatching
- Authorizing
- Authorized
- Waiting
- Cancelled
- Delivered
- Rejected
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 too 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
Package Diagram (Structural)

• Use: Large-Project Structures
Deployment Diagram

- Use: Describing System/Hardware/Software Relationships