# Software modeling with UML



## Agenda

- Approach and motivations
  - What is a (software) model
  - Why do we model? What is UML? Why UML?
  - UML: not a programming language
- Modeling functional requirements
- Modeling the system structure
- Modeling the system dynamics
- Putting all together – a simple case study
- Conclusions
- Bibliography

## Approach and motivations

- Practical approach, focusing on technologies and tools widely accepted and used in industry
- Construction of language-independent models
- Model-Driven Development paradigm
  - Strong impact in analysis, design, documentation
  - Growing impact in development (forward-reverse engineering) and testing



## What is a (software) model?

- A simplification of reality
- An accurate and possibly partial description of a system under study at some level of abstraction
  - A model consists of several submodels describing a certain view of a system
  - A model needs not to be complete
  - A model is expressed in some language at some level of language abstraction
  - A model is more than a description: it is an analogical representation of the things it models.

# Why do we model?

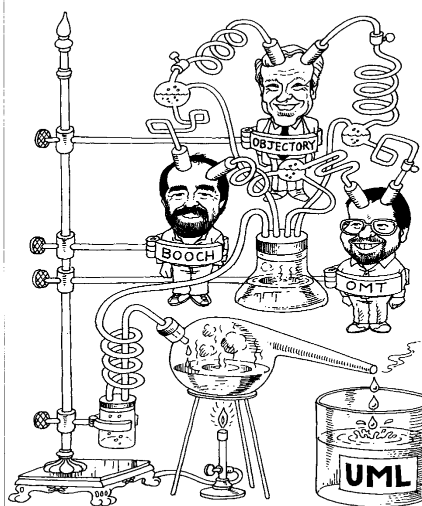
- Models help us to visualize a system as it is (or as we want it to be)
- Models permit us to specify both the structure and the behavior of a system
- Models give us a template that guides the entire system construction
- Models document the decisions we have made



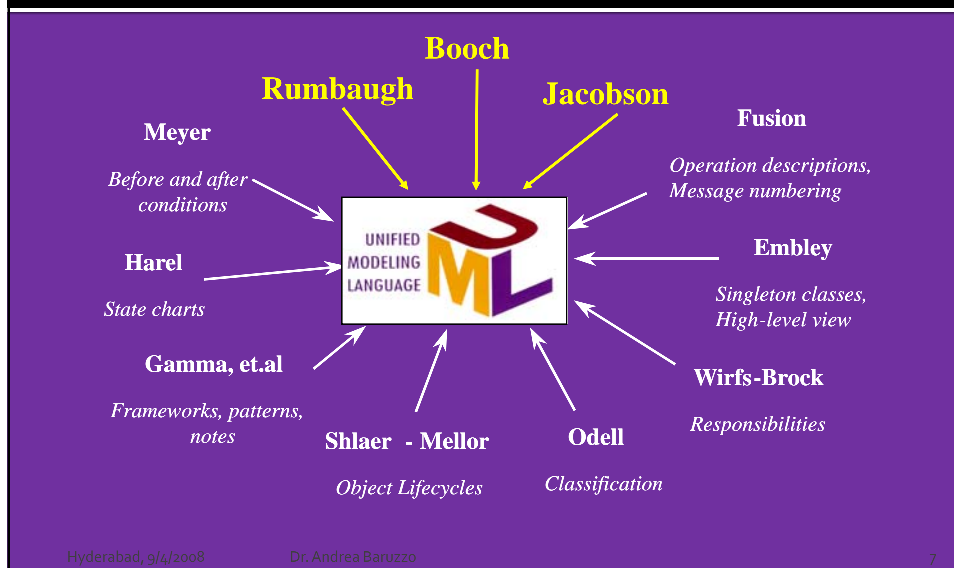
# What is UML?

- UML is a **modeling language** which unifies three methods: Booch, Objectory, OMT
- UML is a language for
  - Visualize,
  - Specify,
  - Build,
  - Document…

  … software artifacts



# What is UML? (cont'd)






# Why UML?

- Software quality!
- The impact of globalization is changing the ways in which software is designed.
- The one word which best describes the benefits of UML is communication
  - A failure to communicate during the development process can lead to disaster, and a great deal of money and time may be wasted
- UML is independent of methods and programming languages

## UML: not a programming language

- One of the main goal of UML is to abstract from the physical machine!
- UML speaks about the problem and the design, conveying only the essential information for the purpose of the current diagram
- UML provides multiple views of the same artifact, adapting the level of detail to the task handles by the modeler



## Agenda

- Approach and motivations
- Modeling functional requirements
  - Use case diagrams
- Modeling the system structure
- Modeling the system dynamics
- Putting all together – a simple case study
- Conclusions
- Bibliography

## Use case diagrams

- A use case diagram is an excellent way to communicate to management, customers, and other non-development people what a system will do when it is completed
- Use case diagrams are used to …
  - Model the context of a system
  - Model the requirements of a system
- They provide a user's perspective of the system



## Use cases and actors

- A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
- A use case describes what a system does but it does not specify how it does it
- A use case typically represents a major piece of functionality which provides some value to the user

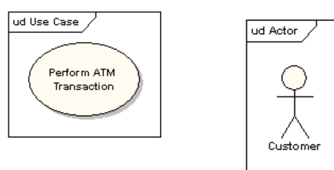
# Use cases and actors *(cont'd)*

- A use case is a description of a scenario (or closely related set of scenarios) in which the system interacts with its users
- Use cases are described as both *narrative scenarios* and *graphical models*
- They can also be refined by *class diagrams* and *interaction diagrams* (to be discuss later)



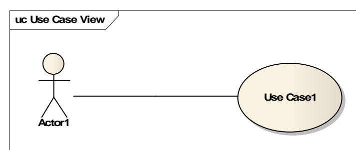
# Use cases and actors *(cont'd)*

- An ***actor*** is anyone or anything that must interact with the system
- Actors are NOT part of the system
- In the UML, a use case is represented as an oval, whereas an actor is represented as a stickman



# Associations between actors and use cases

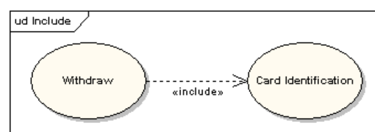
- An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.






# Dependency relations between use cases

- include
  - Specifies that the source use case explicitly incorporates the behavior of another use case at a location specifies by the source.



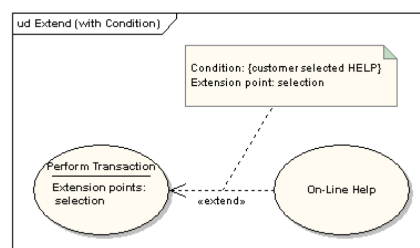
# Dependency relations between use cases (*Cont'd*)

- extend
  - Specifies that the target use case extends the behavior of the source use case, adding an exceptional custom logic at a location specifies by the source.



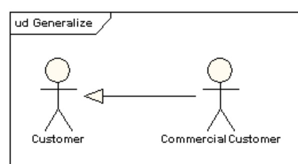
# Dependency relations between use cases (*Cont'd*)

- Extend (cont'd)
  - Extension points and conditions



# Dependency relations between use cases (*Cont'd*)

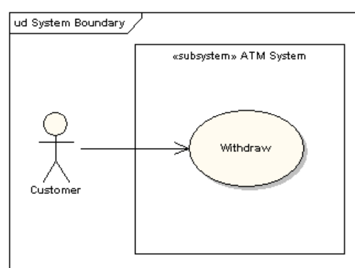
- Generalization
  - Specifies hierarchies of actors (the classical inheritance relation)



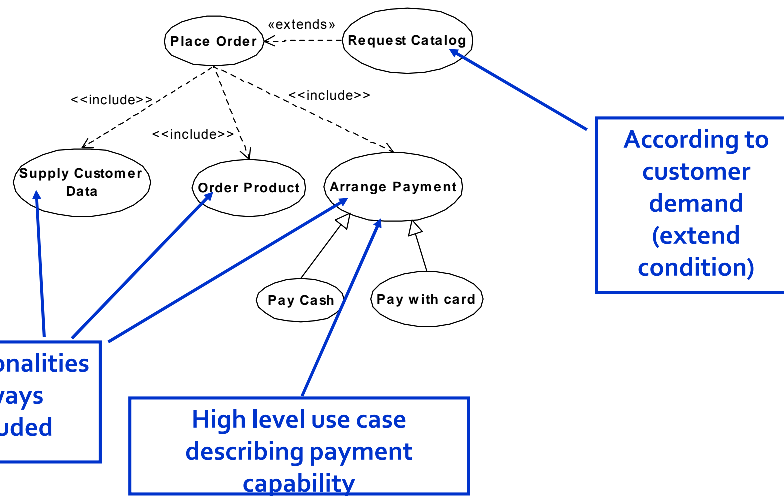



# System boundary

- It is usual to display use cases as being inside the system and actors as being outside the system



# Use case diagram example






# Agenda

- Approach and motivations
- Modeling functional requirements
- Modeling the system structure
  - Class diagrams basics
- Modeling the system dynamics
- Putting all together – a simple case study
- Conclusions
- Bibliography

## What is a class diagram?

- A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them
  - A graphical representation of a static view on static elements
- A central modeling technique that is based on object-oriented principles
- The richest notation in UML

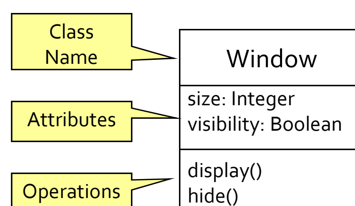


## Essential elements of a class diagram

- Classes (obviously!)
- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
  - Dependency
  - Realization
- Constraint Rules and Notes

# Classes

- A class is the description of a set of objects having similar attributes, operations, relationships and behavior






# Associations

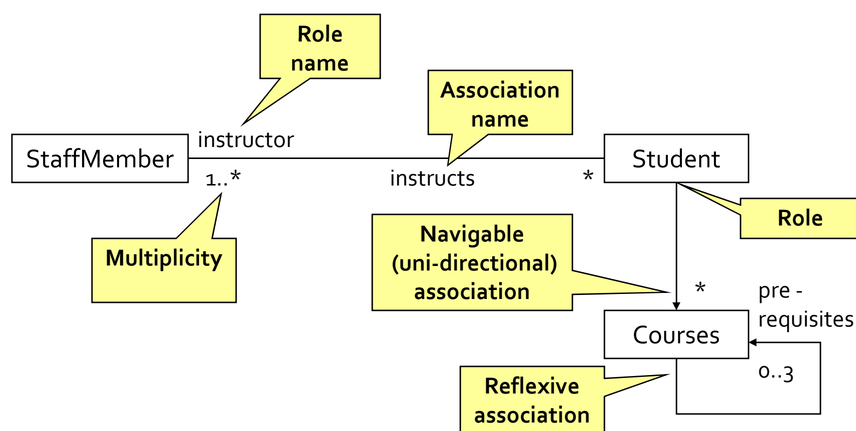
- A semantic relationship between two or more classes that specifies connections among their instances
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class
- Example: "An Employee works in a department of a Company"

# Associations (*cont'd*)

- An association between two classes indicates that objects at one end of an association "recognize" objects at the other end and may send messages to them
  - This property will help us discover less trivial associations using interaction diagrams



# Associations (*cont'd*)



## Associations (*cont'd*)

- To clarify its meaning, an association may be named
  - The name is represented as a label placed midway along the association line
  - Usually a verb or a verb phrase
- A **role** is an end of an association where it connects to a class.
  - May be named to indicate the role played by the class attached to the end of the association path
    - Usually a noun or noun phrase
    - Mandatory for reflexive associations



## Associations (cont'd)

- Multiplicity
  - The number of instances of the class, next to which the multiplicity expression appears, that are referenced by a ***single*** instance of the class that is at the other end of the association path.
  - Indicates whether or not an association is mandatory.
  - Provides a lower and upper bound on the number of instances.

# Associations (cont'd)

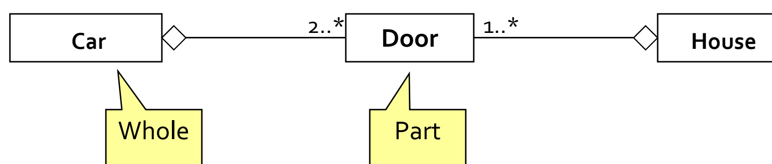
- Multiplicity Indicators

| Exactly one | 1 |
|---|---|
| Zero or more (unlimited) | * (0..*) |
| One or more | 1..* |
| Zero or one (optional association) | 0..1 |
| Specified range | 2..4 |
| Multiple, disjoint ranges | 2, 4..6, 8 |



# Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a "is a part-part of" and "holds/contains" relationships
  - Examples: car-door; house-door; hangar-airplane



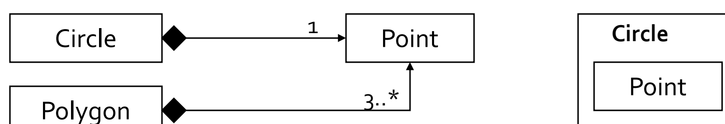
# Aggregation (cont'd)

- Aggregation tests:
  - Is the phrase "part of" used to describe the relationship?
    - A door is "part of" a car
  - Are some operations on the whole automatically applied to its parts?
    - Move the car, move the door.
  - Are some attribute values propagated from the whole to all or some of its parts?
    - The car is blue, therefore the door is blue.
  - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
    - A door **is** part of a car. A car **is not** part of a door.



# Composition

- A strong form of aggregation
  - The whole is the sole owner of its part
    - The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one
  - The life time of the part is dependent upon the whole
    - The composite must manage the creation and destruction of its parts



## Composition vs. Aggregation

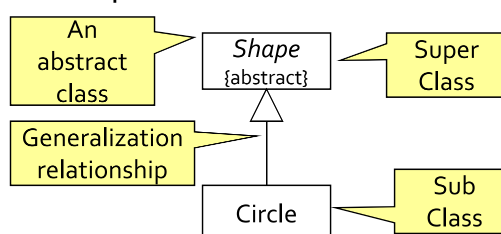
- Quite often, “part of” (“is composed of/by”) is best suited for composition, whereas aggregations are best described by “contains”, “holds”, “has a”
- Not always simple to discriminate
- A car “is composed” of doors or it “contains” doors?
- Don’t spend a lot of time struggling with these details…
- Aggregation and composition describe forms of containment: recognize it and distinguish it from other types of relations!



## Generalization

- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).
  - “is kind of” relationship

`{abstract}` is a tagged value that indicates that the class is abstract. The name of an abstract class should be italicized



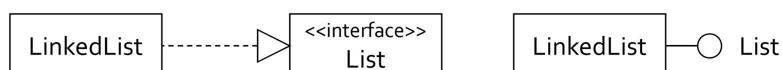
# Generalization (*cont'd*)

- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operations
- A generalization relationship **may not** be used to model interface implementation



# Realization

- A realization relationship indicates that one class implements a behavior specified by some *interface*
- An interface can be realized by many classes
- A class may realize many interfaces

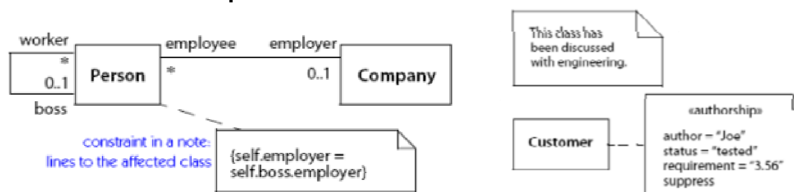


# Dependency

- A dependency indicates a semantic relation between two classes although there is no explicit association between them
- A stereotype may be used to denote the type of the dependency

# Constraint rules and notes

- **Constraints** and **notes** annotate among other things associations, attributes, operations and classes.
- Constraints are *semantic restrictions* noted as Boolean expressions.



## Constraints are used for…

- Document **assumptions** (concerning analysis, design or implementation aspects)
- Describe **invariants**
- Design by contract :
  - Invariant : is always true for an object
  - Pre-condition : is true when method is called
  - Post-condition : is true after method is called

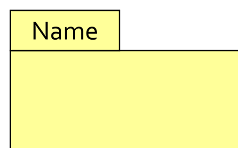


## Stereotypes

- Extend the UML using the << … >> notation
- e.g. Classes can be
  - <<Interface objects>>
  - <<Control objects>>
  - <<Entity objects>>
- Patterns
  e.g. <<singleton>>

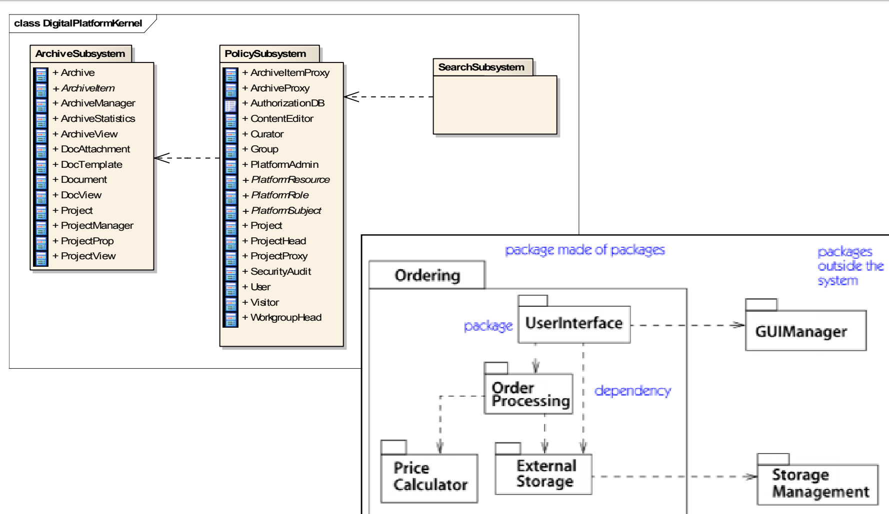
# Packages

- A package is a general purpose grouping mechanism.
- Commonly used for specifying the logical architecture of the system.
- A package does not necessarily translate into a physical sub-system.

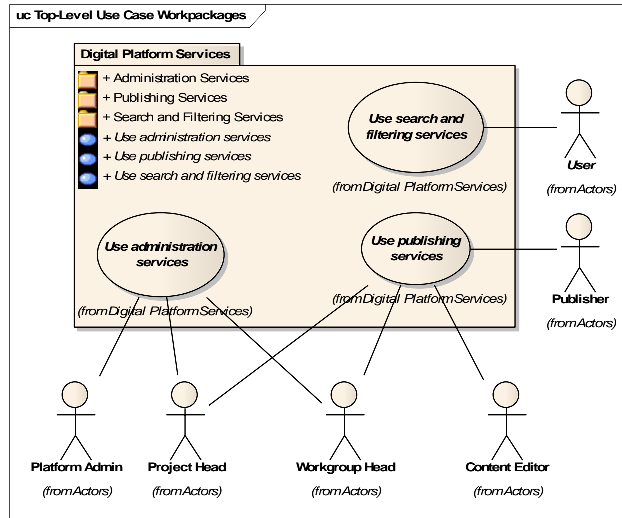





# Package diagrams

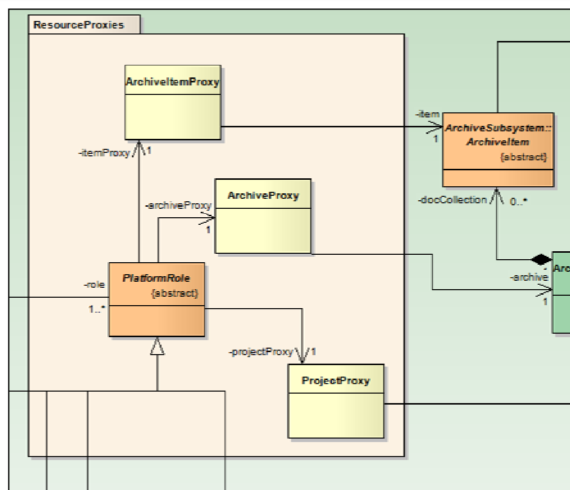


# Packages in use case diagrams






# Packages in class diagrams



## Modeling tips

- Don't try to use all the various notations
- Don't draw models for everything, concentrate on the key areas
- Draw implementation models only when illustrating a particular implementation technique



## Agenda

- Approach and motivations
- Modeling functional requirements
- Modeling the system structure
- Modeling the system dynamics
  - Sequence diagrams
  - Other interaction diagrams
- Putting all together – a simple case study
- Conclusions
- Bibliography

# Interaction diagrams

An **interaction diagram** models communication behavior of individuals exchanging information to accomplish some task

- *Sequence diagram*—shows interacting individuals along the top and message exchange down the page
- *Communication diagram*—shows messages exchanged on a form of object diagram
- *Interaction overview diagram*—a kind of activity diagram whose nodes are sequence diagram fragments
- *Timing diagram*—shows individual state changes over time

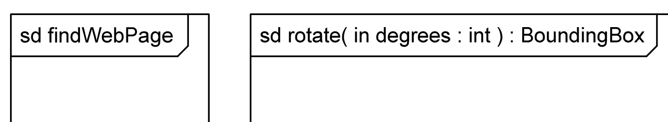


# Sequence diagrams

- Sequence diagrams are useful for modeling
  - Interactions in mid-level design;
  - The interaction between a product and its environment (called *system sequence diagrams*);
  - Interactions between system components in architectural design
- Sequence diagrams can be used as (partial) use case descriptions

# Sequence diagram frame

- Frame—a rectangle with a pentagon in the upper left-hand corner called the name compartment.
- <*sd interactionIdentifier*>
- <*interactionIdentifier*> is either a simple name or an operation specification as in a class diagram



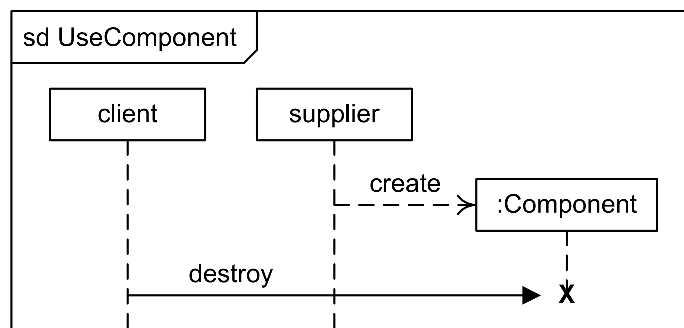



# Lifelines

- Participating individuals are arrayed across the diagram as *lifelines*:
  - Rectangle containing an identifier
  - Dashed line extending down the page
- The vertical dimension represents time; the dashed line shows the period when an individual exists

# Lifeline creation and destruction

- An new object appears at the point it is created.
- A destroyed object has a truncated lifeline ending in an **X.**
- Persisting objects have lifelines that run the length of the diagram





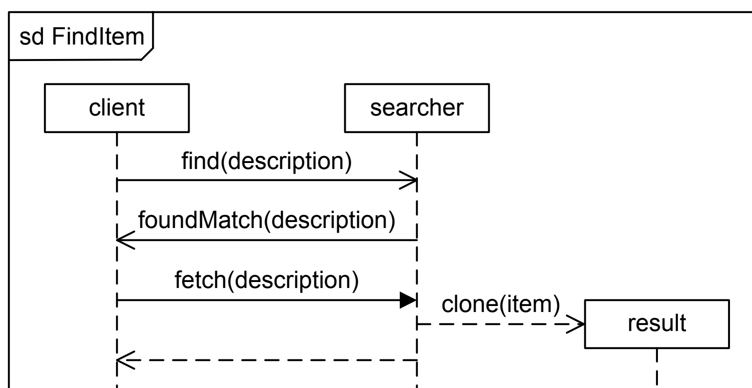

# Messages and arrows

- *Synchronous*—The sender suspends execution until the message is complete

- *Asynchronous*—The sender continues execution after sending the message

- *Synchronous message return or instance creation*

# Message arrow example

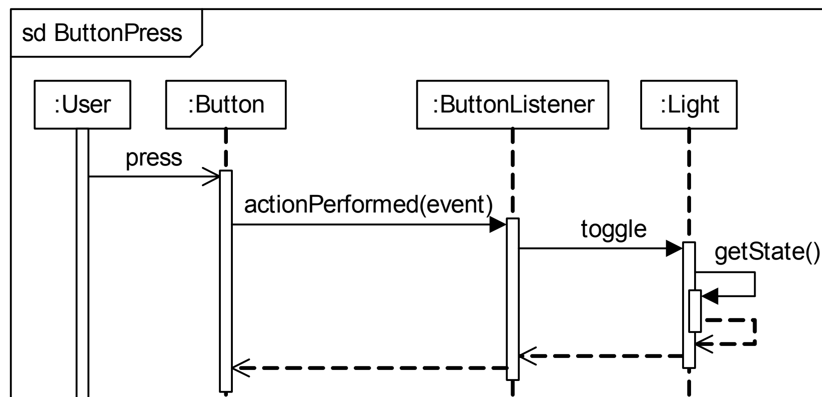





# Execution occurrences

- An operation is **executing** when some process is running its code.
- An operation is **suspended** when it sends a synchronous message and is waiting for it to return.
- An operation is **active** when it is executing or suspended.
- The period when an object is active can be shown using an *execution occurrence*.
  - Thin white or grey rectangle over lifeline dashed line

# Execution occurrence example

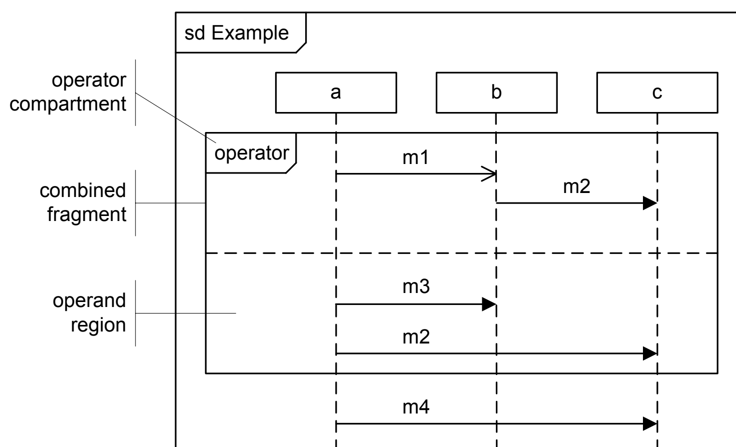





# Combined fragments

- A **combined fragment** is a marked part of an interaction specification that shows
  - Branching,
  - Loops,
  - Concurrent execution,
  - And so forth
- It is surrounded by a rectangular frame.
  - Pentagonal operation compartment
  - Dashed horizontal line forming regions holding operands

# Combined fragment layout

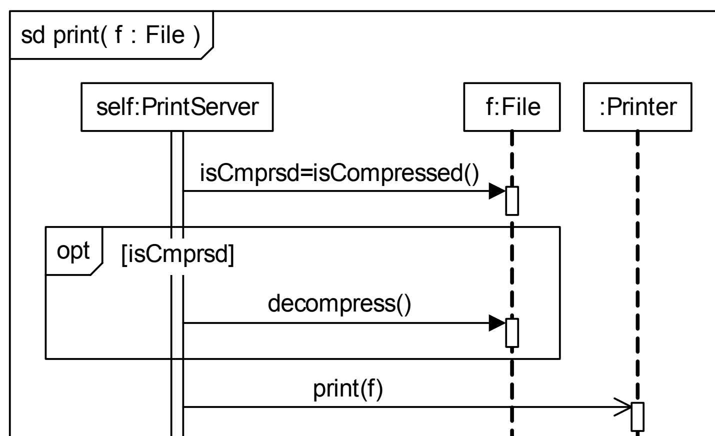





# Optional fragment

- A portion of an interaction that may be done
  - Equivalent to a conditional statement
  - Operator is the keyword opt
  - Only a single operand with a guard
- A *guard* is a Boolean expression in square brackets in a format not specified by UML.
  - [else] is a special guard true if every guard in a fragment is false.

# Optional fragment example



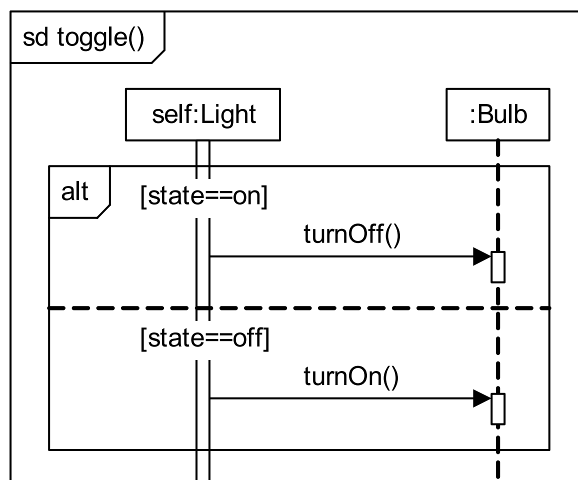



# Alternative fragment

A combined fragment with one or more guarded operands whose guards are mutually exclusive

- Equivalent to a case or switch statement
- Operator is the keyword alt

# Alternative fragment example



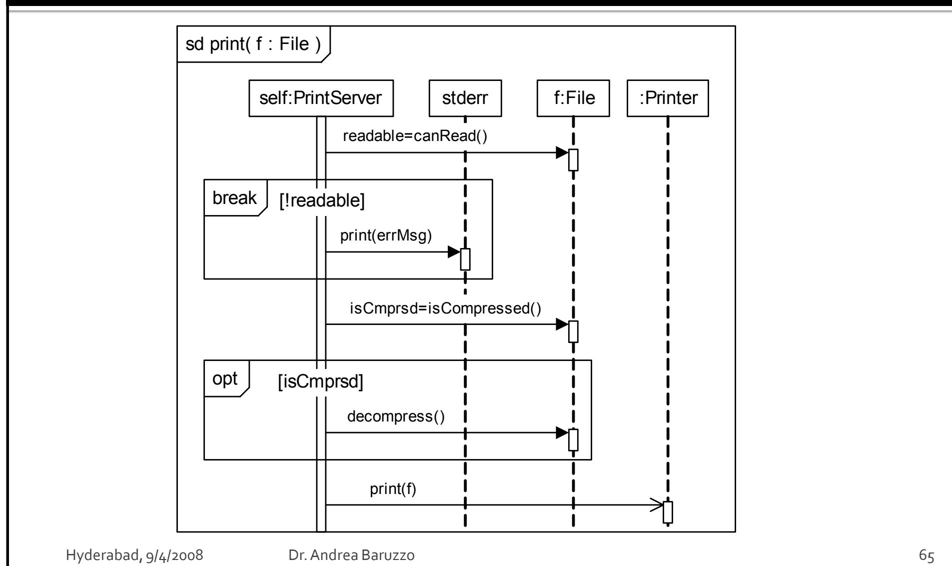



# Break fragment

A combined fragment in which an operand performed in place of the remainder of an enclosing operand or diagram if the guard is true

- Similar to a break statement
- Operator is the keyword break

# Break fragment example






# Loop fragment

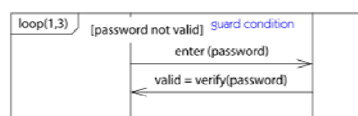
- Single loop body operand that may have a guard
- Operator has the form loop( *min*, *max* ) where
  - Parameters are optional or omitted, so are the parentheses
  - *min* is a non-negative integer
  - *max* is a non-negative integer at least as large as *min* or *; max is optional; if omitted, so is the comma

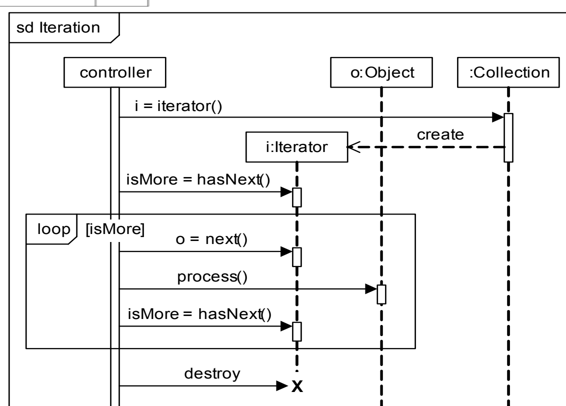
# Loop fragment execution rules

- The loop body is performed at least *min* times and at most *max* times.
- If the loop body has been performed at least *min* times but less than *max* times, it is performed only if the guard is true.
- If *max* is *, the upper iteration bound is unlimited.
- If *min* is specified but *max* is not, then *min*=*max*.
- If the loop has no parameters, then *min*=0 and *max* is unlimited.
- The default value of the guard is true.



# Loop fragment example



# Modeling tips

- Put the sender of the first message leftmost.
- Put pairs of individuals that interact heavily next to one another.
- Position individuals to make message arrows as short as possible.
- Position individuals to make message arrows go from left to right.



# Modeling tips (*Cont'd*)

- Put the self lifeline leftmost.
- In a sequence diagram modeling an operation interaction, draw the self execution occurrence from the top to the bottom of the diagram.
- Name individuals only if they are message arguments or are used in expressions.

# Modeling tips (*Cont'd*)

- Choose a level of abstraction for the sequence diagram.
- Suppress messages individuals send to themselves unless they generate messages to other individuals.
- Suppress return arrows when using execution occurrences.
- Don't assign values to message parameters by name.

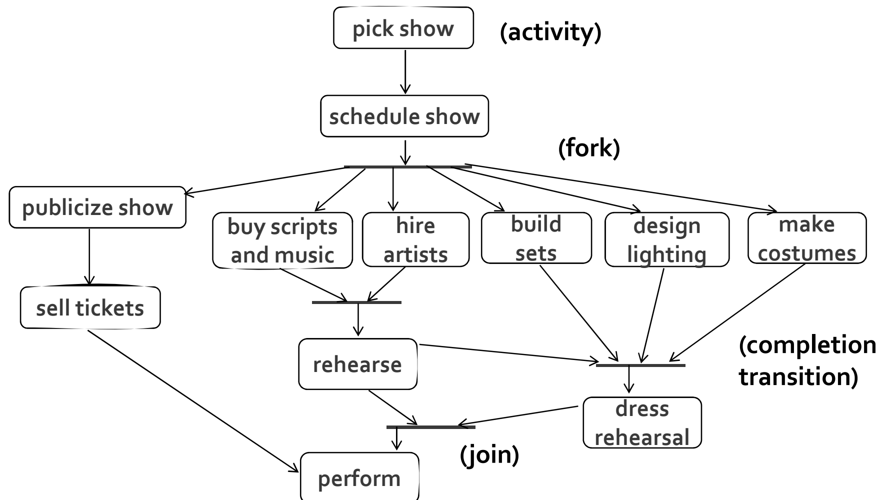


# Other UML diagrams

- Different nature
  - Behavioral: describes behavior of things
  - Structural: describes organization of things
  - Dynamic nature: describe flow of time
  - Static: the time notion is frozen

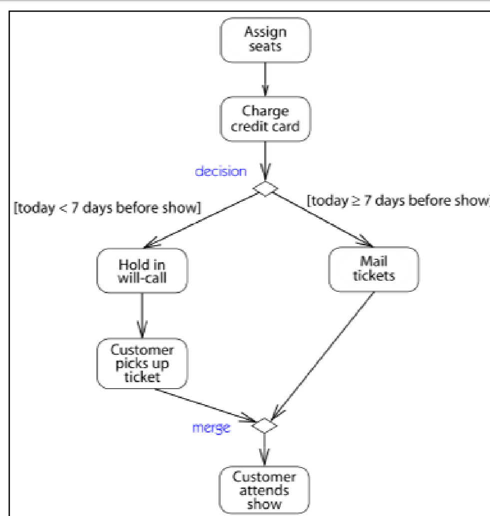
| | Behavioral nature | Structural nature |
|---|---|---|
| Static nature | Use Case Diagram | Package Diagram |
| | Activity Diagram | Class Diagram |
| | Interaction Overview Diagram | Deployment Diagram |
| | | Component Diagram |
| Dynamic nature | State Machine Diagram | Object Diagram |
| | Sequence Diagram | Composite Structure Diagram |
| | Communication Diagram | |
| | Timing Diagram | |

# Activity diagrams



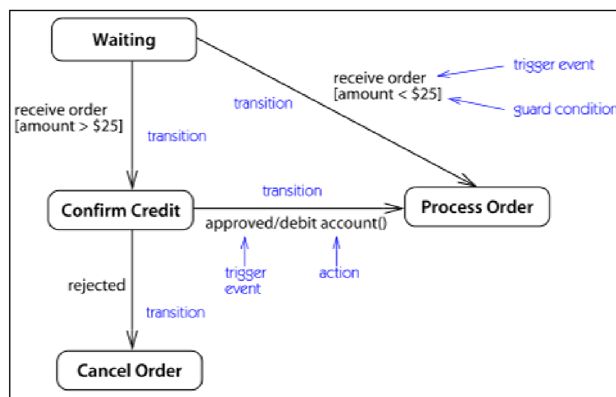

Hyderabad, 9/4/2008 Dr. Andrea Baruzzo 73

# Activity diagrams (*Cont'd*)




Hyderabad, 9/4/2008 Dr. Andrea Baruzzo 74

# State-chart diagram






# Agenda

- Approach and motivations
- Modeling functional requirements
- Modeling the system structure
- Modeling the system dynamics
- Putting all together – a simple case study
- Conclusions
- Bibliography

# Videogame case study

- Live demo with Enterprise Architect modeling tool
  - Use case diagrams for requirements
  - Class diagrams for system structure
  - Sequence diagrams, state diagrams and activity diagrams for system dynamics



# Agenda

- Approach and motivations
- Modeling functional requirements
- Modeling the system structure
- Modeling the system dynamics
- Putting all together – a simple case study
- Conclusions
- Bibliography

## Conclusions: tenets of Model-Based development

- Start quick vs. start right!
- UML modeling as a knowledge crunching process
- What knowledge?
  - The knowledge of the problem domain
  - The knowledge recognizable in user requirements (explicit, but especially implicit ones!)
  - How can I hope to build a useful system if I do not know what I have to build and why?



## Bibliography


- **The Unified Modeling Language User Guide 2/E**
  G. Booch, J. Rumbaugh, I. Jacobson - Addison-Wesley, 2005
- **The Unified Modeling Language Reference Manual 2/E**
  J. Rumbaugh, I. Jacobson, G. Booch - Addison-Wesley, 2004
- **UML Distilled 3/E**
  M. Fowler - Addison-Wesley, 2003
- **Applying UML and Patterns**
  C. Larman - Addison-Wesley, 2004
- **UML Bible**
  T. Pender – Wiley&Sons, 2003

## Thank you!


- Dr. Andrea Baruzzo
  - University of Udine, Computer Science Dept.
    Artificial Intelligence Laboratory
    Office: 2nd floor, room SSSH
    Via delle Scienze 206, Loc. Rizzi
    33100 Udine, ITALY
  - Phone: +39 0432 55.84.35
    Fax: +39 0432 55.84.99
    E-mail: andrea.baruzzo(at)dimi.uniud.it
    http://users.dimi.uniud.it/~andrea.baruzzo/