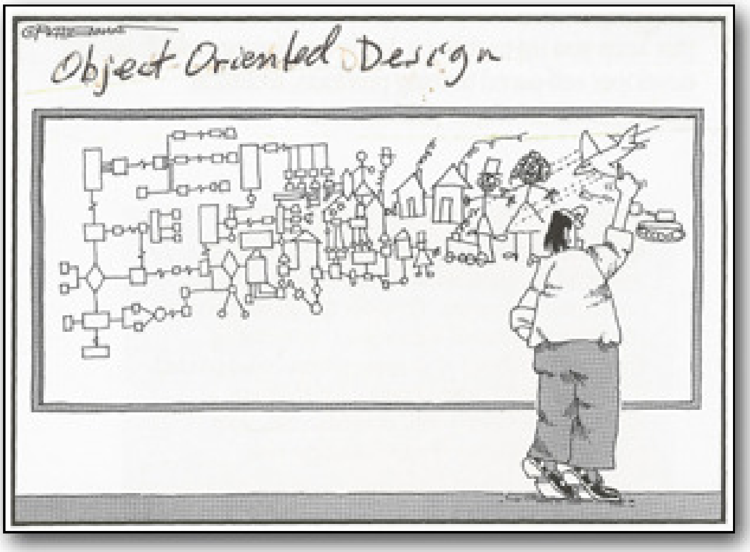
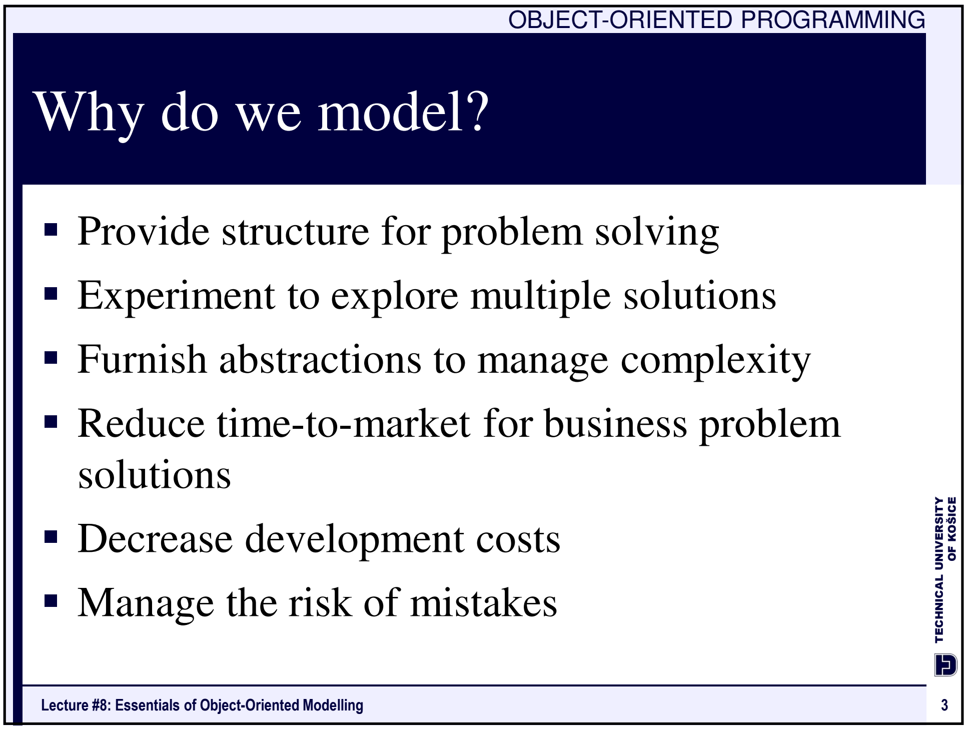


OBJECT-ORIENTED PROGRAMMING

Essentials of Object-Oriented Modelling

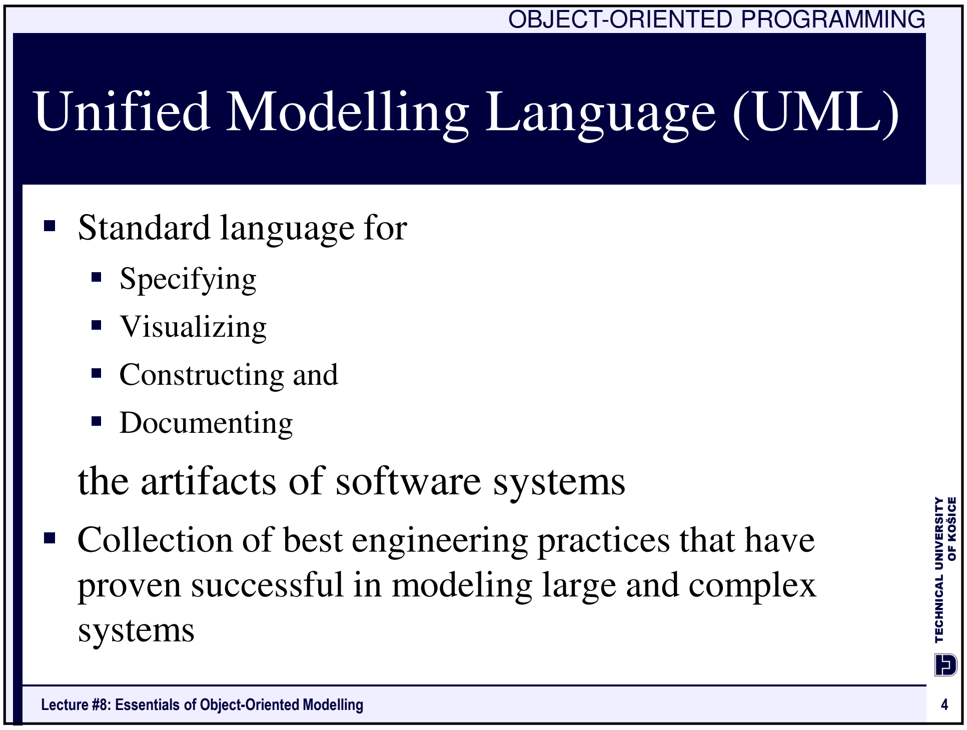
Lecture #8

TECHNICAL UNIVERSITY OF KOSICE



Why do we model?

- Provide structure for problem solving
- Experiment to explore multiple solutions
- Furnish abstractions to manage complexity
- Reduce time-to-market for business problem solutions
- Decrease development costs
- Manage the risk of mistakes

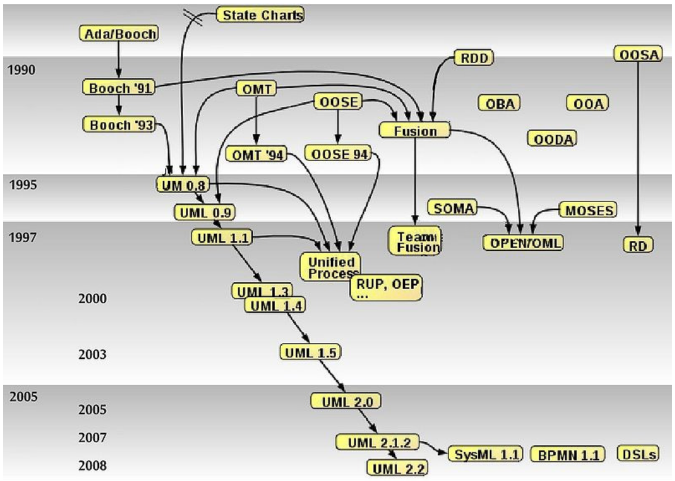


Unified Modelling Language (UML)

- Standard language for
  - Specifying
  - Visualizing
  - Constructing and
  - Documenting

  the artifacts of software systems
- Collection of best engineering practices that have proven successful in modeling large and complex systems

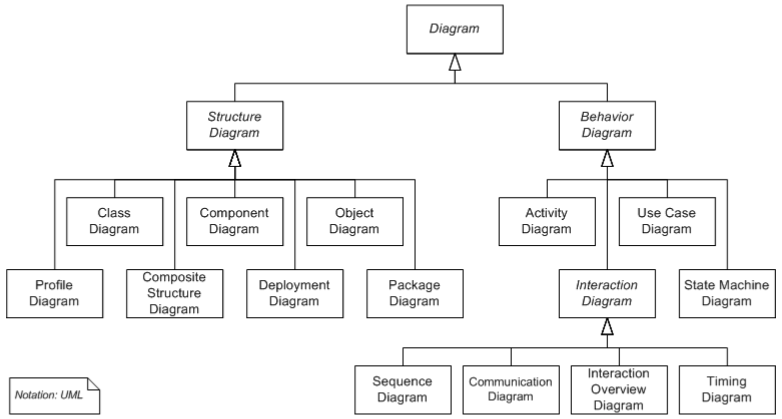
UML history



UML diagrams

- **Structural diagrams**
  - Used to describe the building blocks of the system
    - Features that do not change with time
  - These diagrams answer the question: What is there?
- **Behavioral diagrams**
  - Used to show how the system evolves over time
    - Responds to requests, events, etc.

UML diagrams



Class diagrams

- Classes are represented by a rectangle divided to three parts
  - **Class name**
  - **Attributes**
  - **Operations**
- Attributes are written as

  `visibility name[multiplicity]: type_expression = initial_value`
- Operations are written as

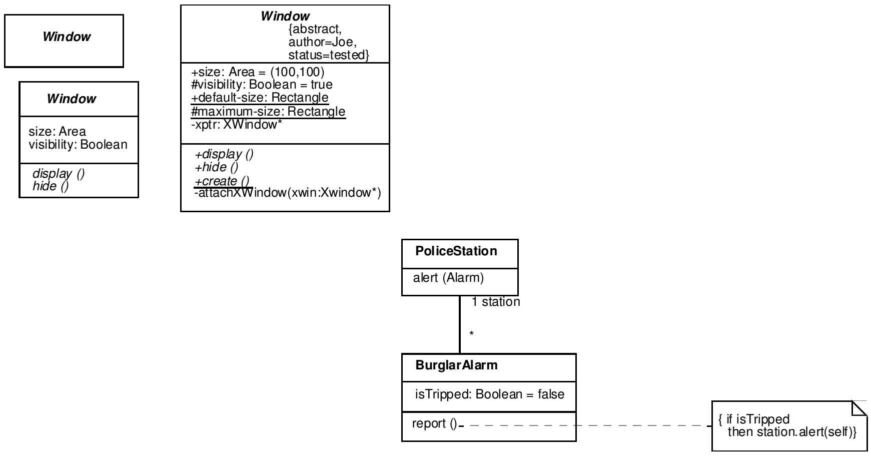
  `visibility name(parameter_list): return_type_expression`
- Visibility is written as

  + public

  # protected

  – private

Class diagram – Examples



Window

Window
size: Area
visibility: Boolean
display ()
hide ()

Window
{abstract,
author=Joe,
status=tested}
+size: Area = (100,100)
#visibility: Boolean = true
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*
+display ()
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

PoliceStation
alert (Alarm)

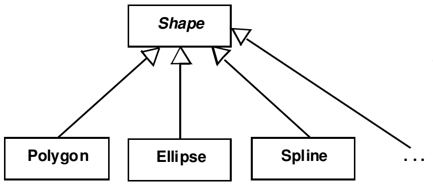
BurglarAlarm
isTripped: Boolean = false
report ()

{ if isTripped
then station.alert(self)}

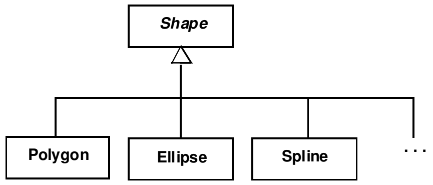
Relationships in class diagram

- **Association**
  - Two classes are associated if one class has to know about the other
- **Aggregation**
  - An association in which one class belongs to a collection in the other
- **Generalization**
  - An inheritance link indicating one class is a base class of the other
- **Dependency**
  - A labeled dependency between classes (friend classes, instantiation, etc.)
- **Realization**
  - A relationship between specification and its implementation

Generalization – Examples

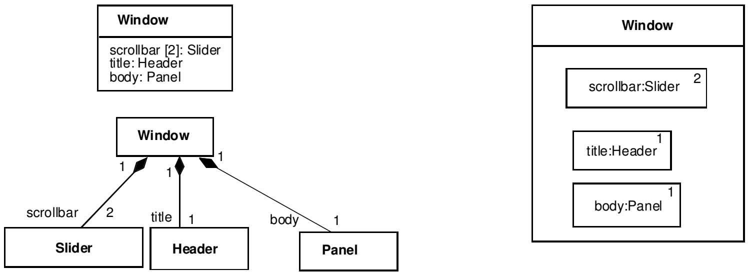


Separate Target Style

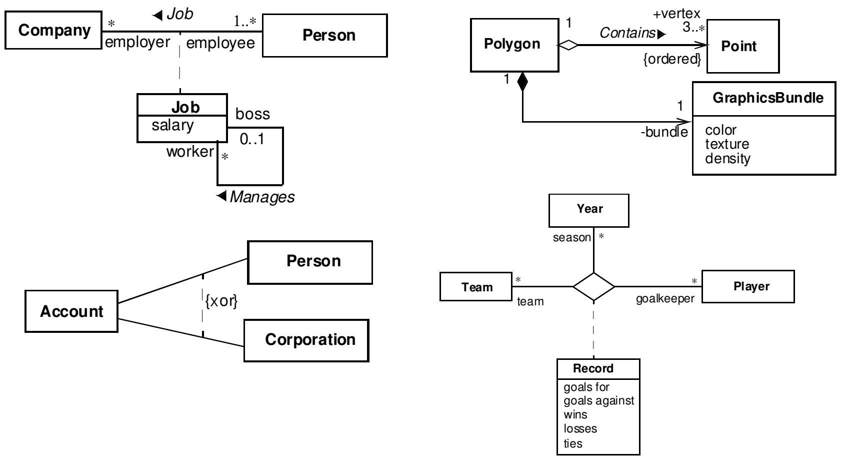


Shared Target Style

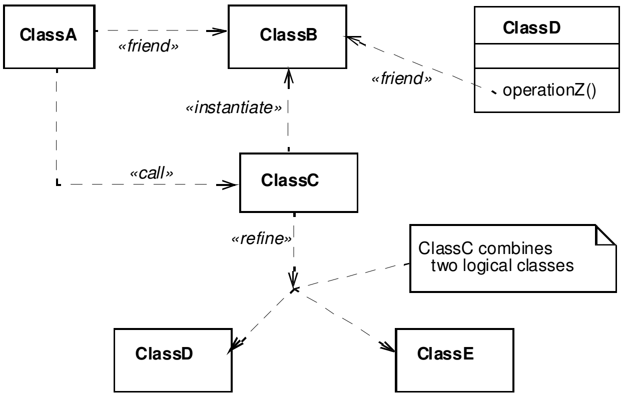
Aggregation – Examples



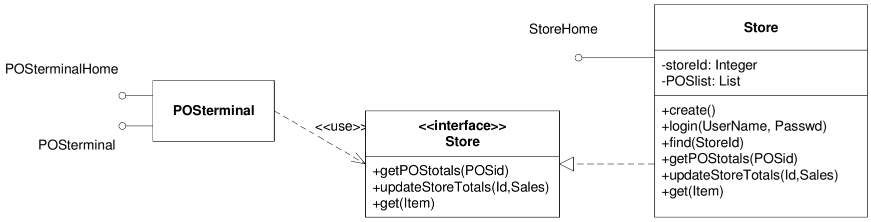
Association – Examples



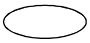
Dependencies – Examples



Realization of interfaces – Examples



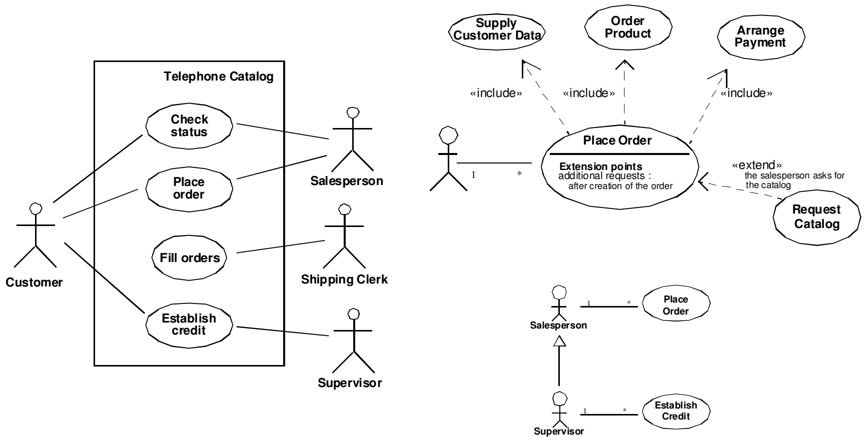
Use case diagram

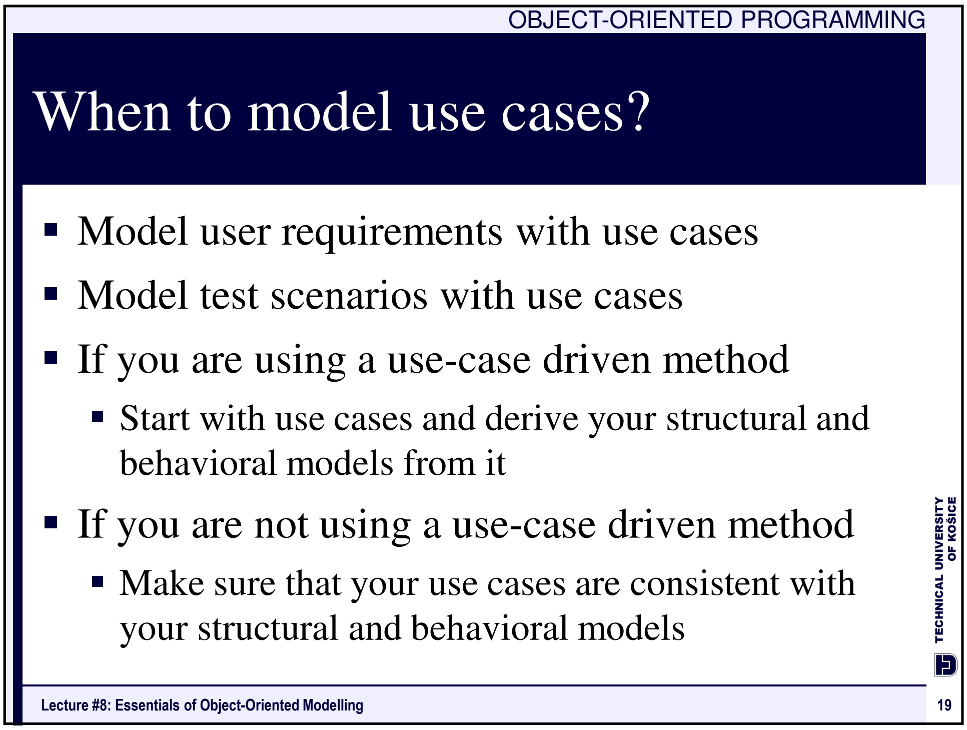
- Describes what a system does from the standpoint of an external observer
  - Emphasis on **what** a system does rather then **how**
- **Scenario** 
  - An example of what happens when someone interacts with the system
- **Actor**
  - A user or another system that interacts with the modeled system
- **System boundary**
  - Represents the boundary between the physical system and the actors who interact with the physical system
- A use case diagram describes the relationships between actors and scenarios
- Provides system requirements from the user's point of view

Relationships in use case diagram

- **Association** ————————
  - The participation of an actor in a use case i.e., instance of an actor and instances of a use case communicate with each other
- **Generalization** ————————▷
  - A taxonomic relationship between a more general use case and a more specific use case
- **Extend** - - - - «extend» - - - ->
  - A relationship from an extension use case to a base use case, specifying how the behavior for the extension use case can be inserted into the behavior defined for the base use case
- **Include** - - - - «include» - - - ->
  - An relationship from a base use case to an inclusion use case, specifying how the behavior for the inclusion use case is inserted into the behavior defined for the base use case

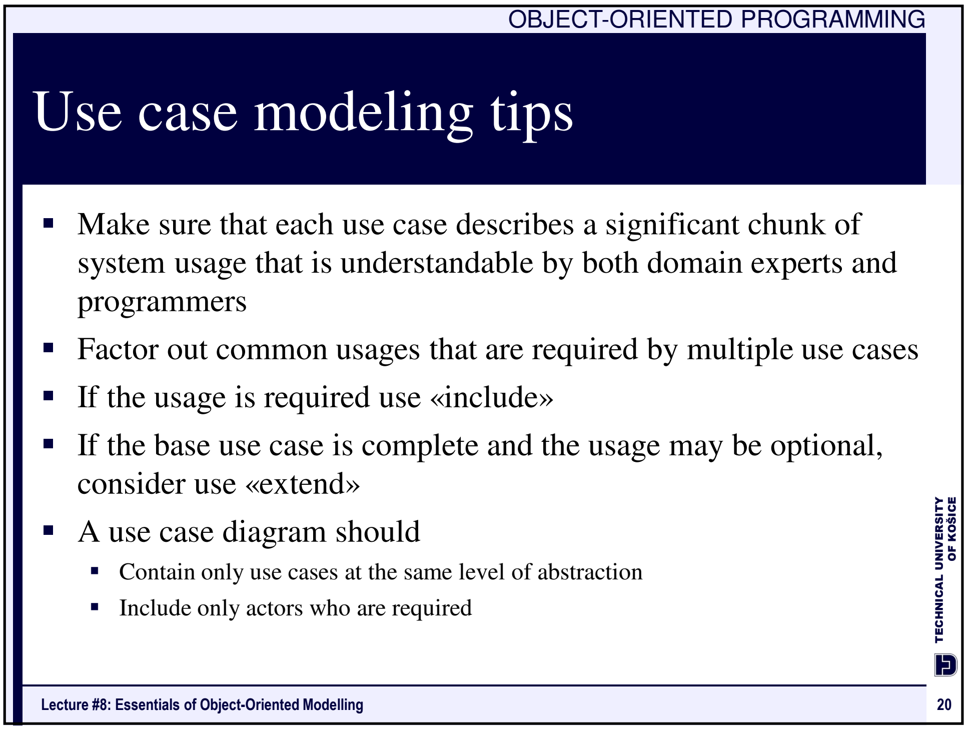
Use case – Examples





When to model use cases?

- Model user requirements with use cases
- Model test scenarios with use cases
- If you are using a use-case driven method
  - Start with use cases and derive your structural and behavioral models from it
- If you are not using a use-case driven method
  - Make sure that your use cases are consistent with your structural and behavioral models



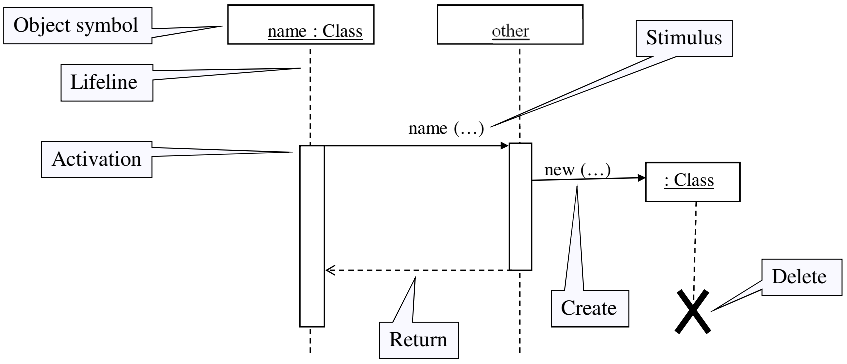
Use case modeling tips

- Make sure that each use case describes a significant chunk of system usage that is understandable by both domain experts and programmers
- Factor out common usages that are required by multiple use cases
- If the usage is required use «include»
- If the base use case is complete and the usage may be optional, consider use «extend»
- A use case diagram should
  - Contain only use cases at the same level of abstraction
  - Include only actors who are required

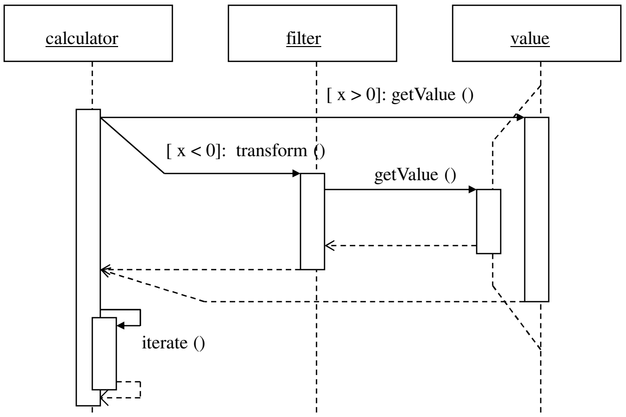
Interactions modelling

- Show interactions between instances in the model
  - Graph of instances (possibly including links) and stimuli
  - Existing instances
  - Creation and deletion of instances
- Kinds
  - **Sequence diagram** (temporal focus)
  - **Collaboration diagram** (structural focus)

Sequence diagram – Overview



Sequence diagram – Recursion, condition, etc.



calculator

filter

value

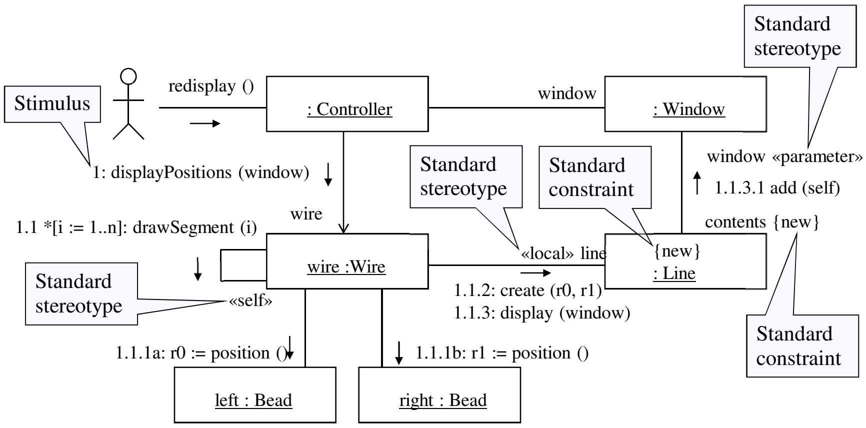
[x > 0]: getValue ()

[x < 0]: transform ()

getValue ()

iterate ()

Collaboration diagram – Overview



Standard stereotype

Stimulus

redisplay ()

: Controller

window

: Window

1: displayPositions (window)

Standard stereotype

Standard constraint

window «parameter»

1.1.3.1 add (self)

1.1 *[i := 1..n]: drawSegment (i)

wire

contents {new}

wire :Wire

«local» line

{new}
: Line

Standard stereotype

«self»

1.1.2: create (r0, r1)
1.1.3: display (window)

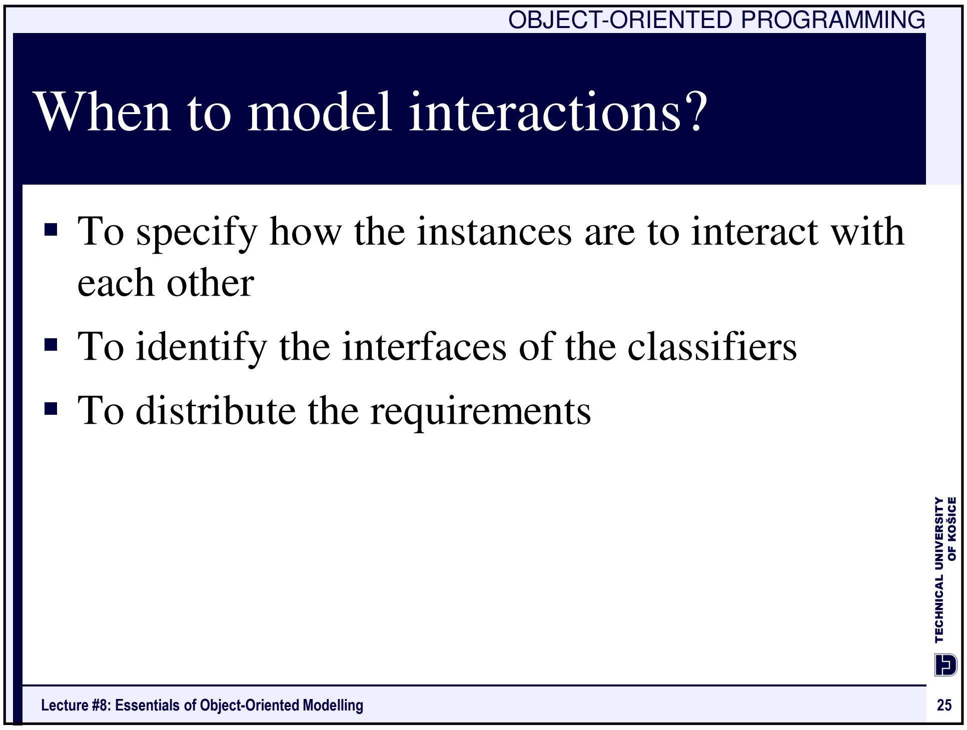
1.1.1a: r0 := position ()

1.1.1b: r1 := position ()

Standard constraint

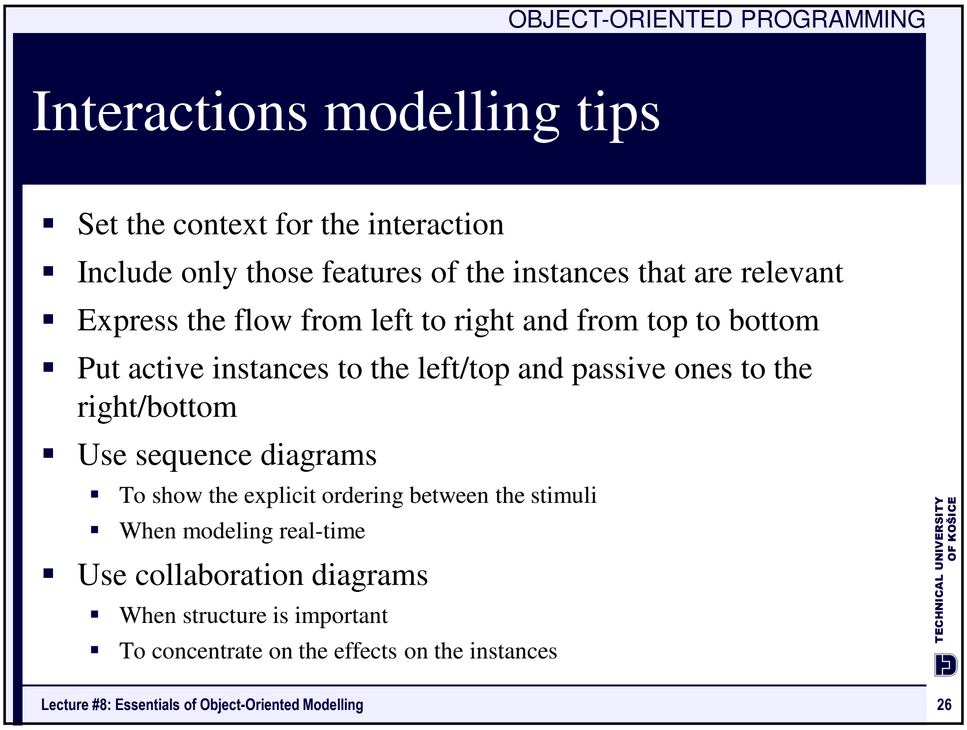
left : Bead

right : Bead



When to model interactions?

- To specify how the instances are to interact with each other
- To identify the interfaces of the classifiers
- To distribute the requirements



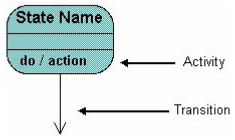
Interactions modelling tips

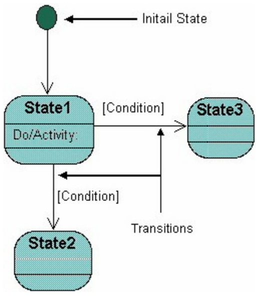
- Set the context for the interaction
- Include only those features of the instances that are relevant
- Express the flow from left to right and from top to bottom
- Put active instances to the left/top and passive ones to the right/bottom
- Use sequence diagrams
  - To show the explicit ordering between the stimuli
  - When modeling real-time
- Use collaboration diagrams
  - When structure is important
  - To concentrate on the effects on the instances

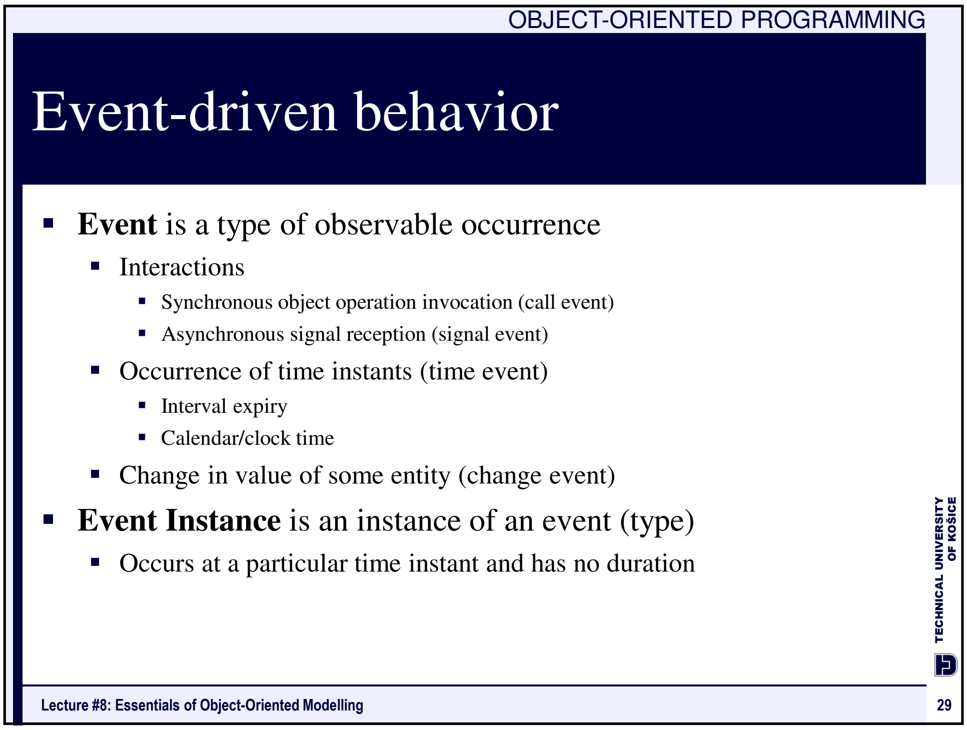
State machine diagrams

- A machine whose output behavior is not only a direct consequence of the current input, but of some past history of its inputs
- Characterized by an internal state which represents this past experience
- Theoretical background: **Finite-State Automata**
  - Sequential systems: **Transducers** by Mealy and Moore
  - Formal languages processing: **Acceptors**

State machine diagram – Overview

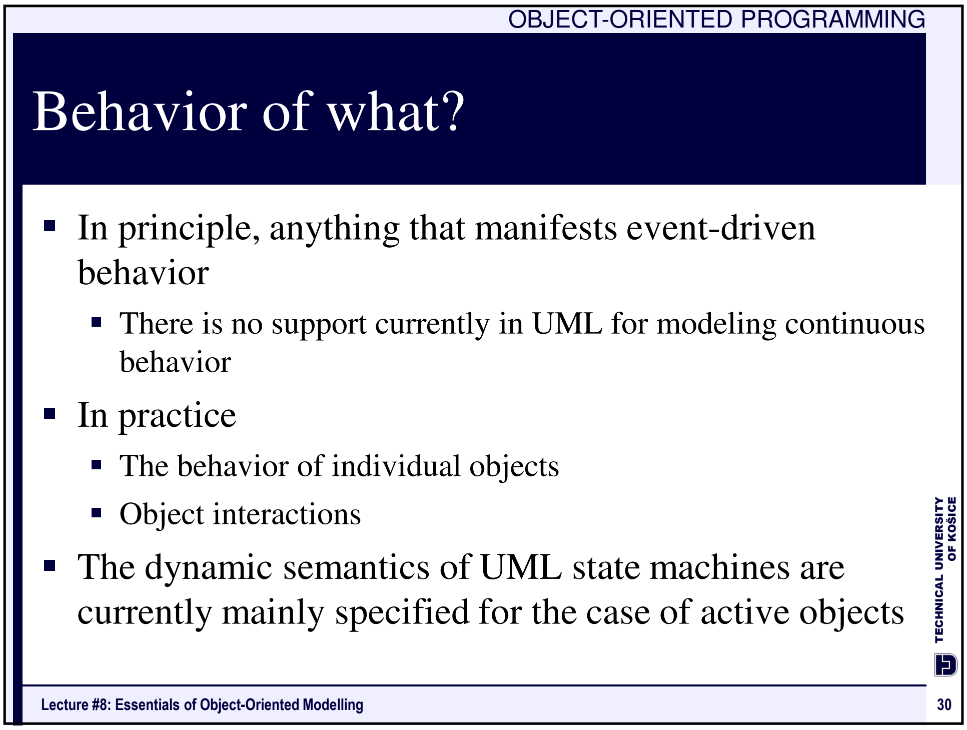






Event-driven behavior

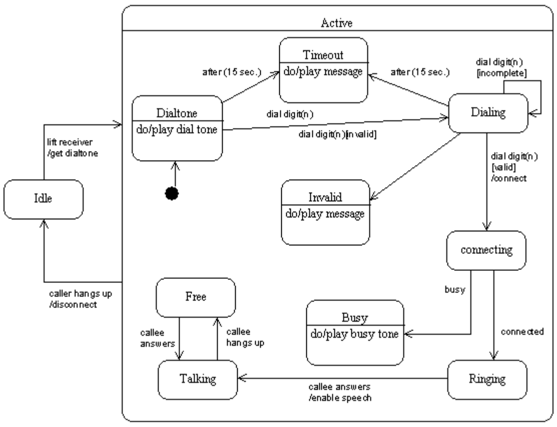
- **Event** is a type of observable occurrence
  - Interactions
    - Synchronous object operation invocation (call event)
    - Asynchronous signal reception (signal event)
  - Occurrence of time instants (time event)
    - Interval expiry
    - Calendar/clock time
  - Change in value of some entity (change event)
- **Event Instance** is an instance of an event (type)
  - Occurs at a particular time instant and has no duration



Behavior of what?

- In principle, anything that manifests event-driven behavior
  - There is no support currently in UML for modeling continuous behavior
- In practice
  - The behavior of individual objects
  - Object interactions
- The dynamic semantics of UML state machines are currently mainly specified for the case of active objects

State machine diagram – Examples



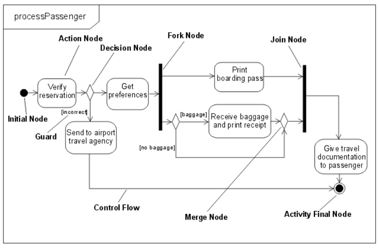
Activity modelling

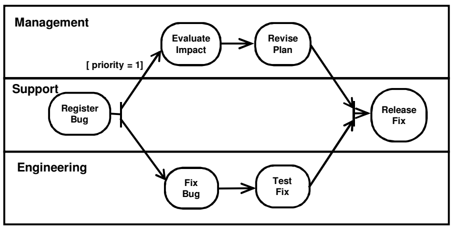
- Intended for applications that need control flow or object/data flow models rather than event-driven models like state machines
- For example
  - Business process modeling and workflows
- How step in a process is initiated, especially with respect to how the step gets its inputs

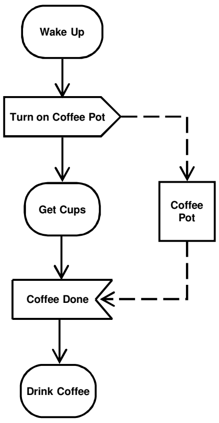
Activity diagram

- **Swimlane**
  - Used to organize responsibility for actions and subactivities. Often corresponds to organizational units in a business model
- **Fork**
  - Splits an incoming transition into several concurrent outgoing transitions. All of the transitions fire together
- **Join**
  - Merges transitions from concurrent regions into a single outgoing transition. All the transitions fire together
- **Decision**
  - A state node that represents a decision. Each transition from this node depends on a Boolean condition

Activity diagram – Examples





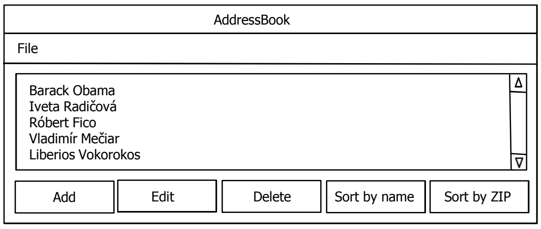


When to use activity diagrams?

- Use activity diagrams when the behavior you are modeling
  - Does not depend much on external events
  - Mostly has steps that run to completion, rather than being interrupted by events
  - Requires object/data flow between steps
  - Is being constructed at a stage when you are more concerned with which activities happen, rather than which objects are responsible for them (except partitions possibly)

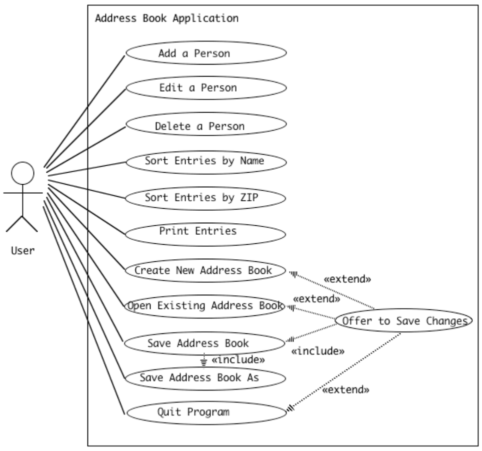
Simple Address Book example

- Basic requirements
  - Add a new person
  - Edit personal information
  - Remove a person from the address book
  - Sort persons by name and ZIP code
  - Manage more address books (manage various address book files)
  - Keep track of all changes



Barack Obama
Iveta Radičová
Róbert Fico
Vladimír Mečiar
Liberios Vokorokos

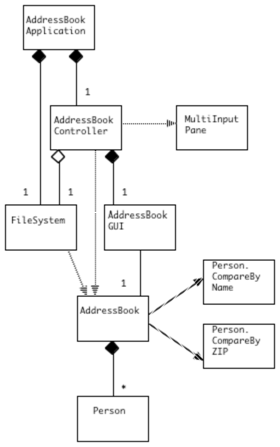
Use case model



How to identify classes and their responsibilities?

- Responsibilities are assigned to the various classes based on the use of the model-view-controller design pattern
  - The two entity classes (`AddressBook` and `Person`) serve as the model
  - The GUI class (`AddressBookGUI`) serves as the view
  - The controller class (`AddressBookController`) serves, of course, as the controller
- The view (`AddressBookGUI`) needs to be made an observer of the model (specifically, `AddressBook`) so that it always reflects the current state of the model – specifically, the list of names, the title, and its saved/needs to be saved status
- Assigning responsibilities to various classes for the tasks required by the various use cases leads to the creation of the following cards
  - Class `AddressBook`
  - Class `AddressBookController`
  - Class `AddressBookGUI`
  - Class `FileSystem`
  - Class `Person`

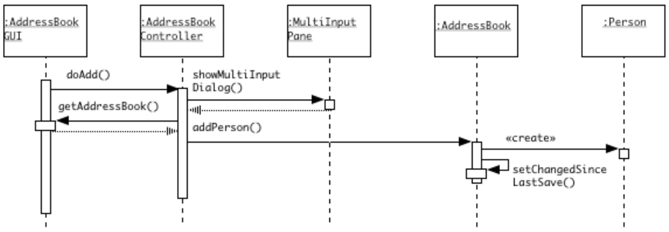
Class diagram



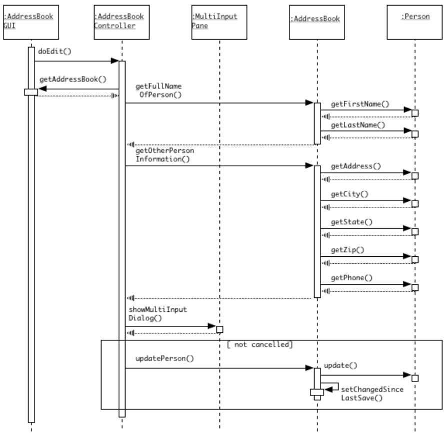
Modelling interactions

- Each of the use cases discovered in the analysis of the system will be realized by a sequence of operations involving the various objects comprising the system

Add a person

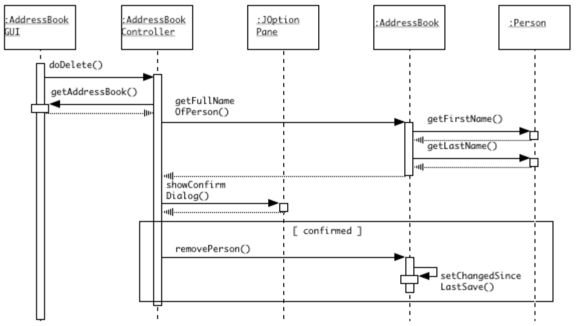


Edit a person



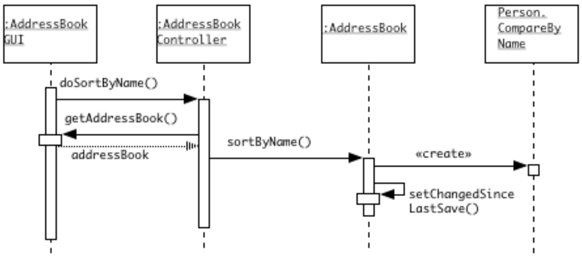
If there is no selected name, none of the above is done; instead, an error is reported

Delete a person

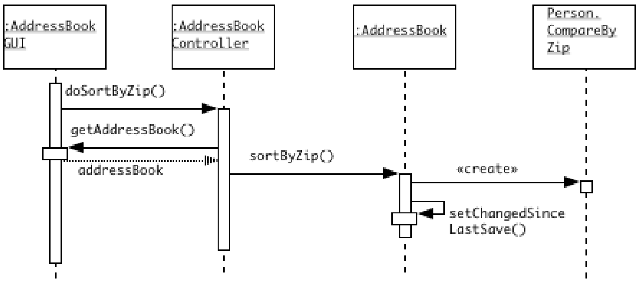


If there is no selected name, none of the above is done;
instead, an error is reported

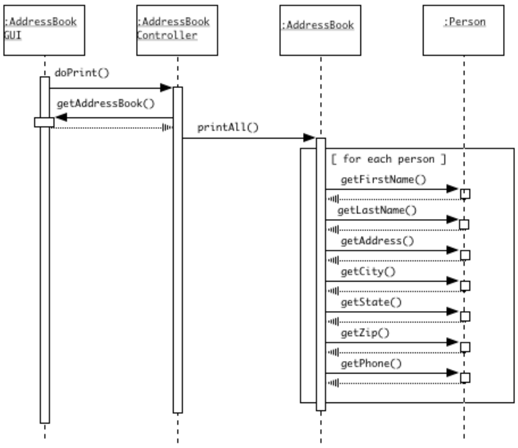
Sort enries by name



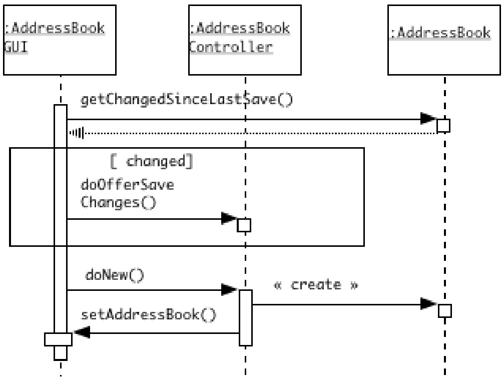
Sort entries by ZIP



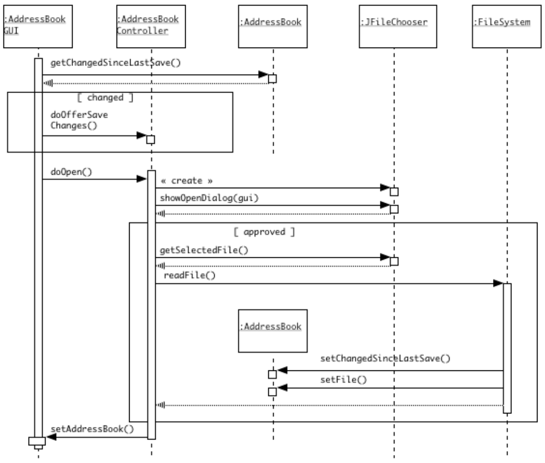
Print entries



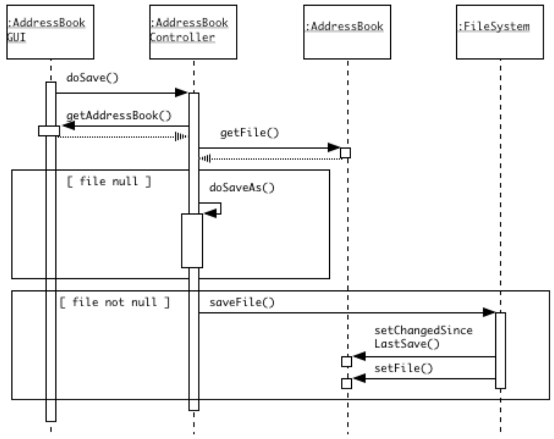
Crerate new address book



Open existing address book



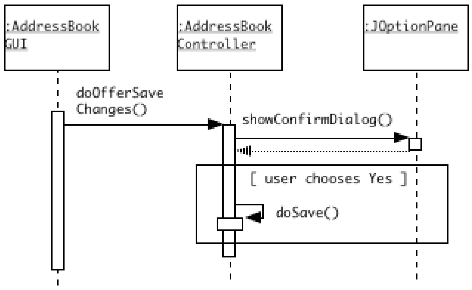
Save address book



Save address book as

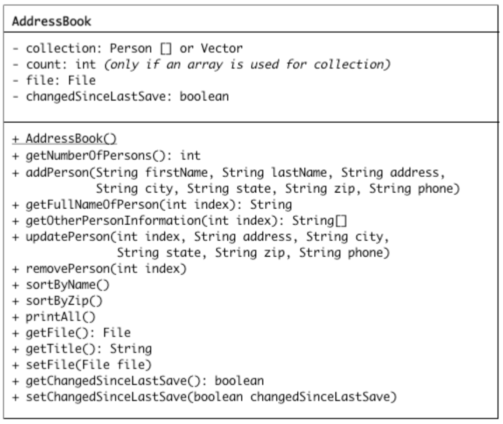
- Homework

Offer to save changes



If the user chooses "Cancel" in the confirm dialog,
an InterruptedException is thrown

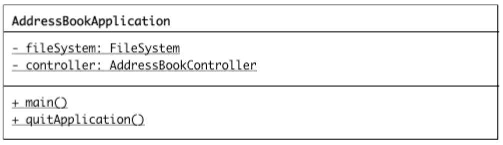
Class AddressBook



```
AddressBook
- collection: Person [] or Vector
- count: int (only if an array is used for collection)
- file: File
- changedSinceLastSave: boolean

+ AddressBook()
+ getNumberOfPersons(): int
+ addPerson(String firstName, String lastName, String address,
            String city, String state, String zip, String phone)
+ getFullNameOfPerson(int index): String
+ getOtherPersonInformation(int index): String[]
+ updatePerson(int index, String address, String city,
               String state, String zip, String phone)
+ removePerson(int index)
+ sortByName()
+ sortByZip()
+ printAll()
+ getFile(): File
+ getTitle(): String
+ setFile(File file)
+ getChangedSinceLastSave(): boolean
+ setChangedSinceLastSave(boolean changedSinceLastSave)
```

Class AddressBookApplication



AddressBookApplication

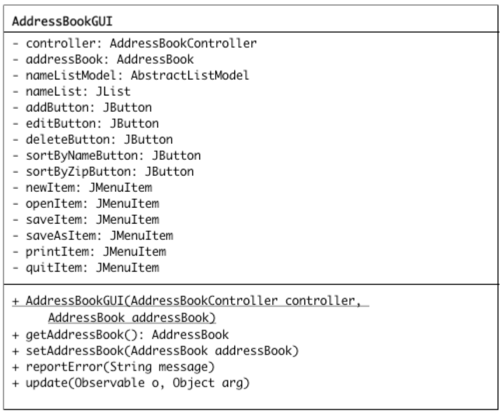
- fileSystem: FileSystem
- controller: AddressBookController

+ main()
+ quitApplication()

Class AddressBookController

- Homework

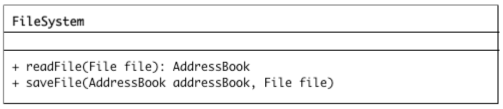
Class AddressBookGUI



```
AddressBookGUI
- controller: AddressBookController
- addressBook: AddressBook
- nameListModel: AbstractListModel
- nameList: JList
- addButton: JButton
- editButton: JButton
- deleteButton: JButton
- sortByNameButton: JButton
- sortByZipButton: JButton
- newItem: JMenuItem
- openItem: JMenuItem
- saveItem: JMenuItem
- saveAsItem: JMenuItem
- printItem: JMenuItem
- quitItem: JMenuItem

+ AddressBookGUI(AddressBookController controller,
     AddressBook addressBook)
+ getAddressBook(): AddressBook
+ setAddressBook(AddressBook addressBook)
+ reportError(String message)
+ update(Observable o, Object arg)
```

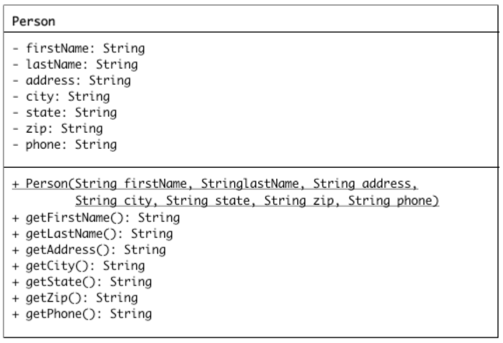
Class FileSystem



```
FileSystem

+ readFile(File file): AddressBook
+ saveFile(AddressBook addressBook, File file)
```

Class Person



```
Person
- firstName: String
- lastName: String
- address: String
- city: String
- state: String
- zip: String
- phone: String
+ Person(String firstName, StringlastName, String address,
         String city, String state, String zip, String phone)
+ getFirstName(): String
+ getLastName(): String
+ getAddress(): String
+ getCity(): String
+ getState(): String
+ getZip(): String
+ getPhone(): String
```

Readings

- RUMBAUGH, J. – JACOBSON, I. – BOOCH, G.: *The Unified Modeling Language Reference Manual*. 2nd Edition, Addison-Wesley Professional, 2004
- ARLOW, J. – NEUSTADT, I.: *UML 2 a unifikovaný proces vývoje aplikací*. Computer Press, 2007