Goals of object-oriented modeling

- Integrated description of structure and behavior of objects
- Representation for properties of objects and relationships between objects
- Development of object descriptions capturing state transitions and object lifecycles
- Options to describe type level and instance level aspects
- Modeling language used here: Unified Modeling Language (UML)
Good reference book on UML (Unified Modeling Language)

James Rumbaugh, Ivar Jacobson, Grady Booch
ISBN 0321245628
Pearson Higher Education

Excerpts (tables, quotations, diagrams, ...) in the course slides
<table>
<thead>
<tr>
<th>Major Area</th>
<th>View</th>
<th>Diagram</th>
<th>Main Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>structural</td>
<td>static view</td>
<td>class diagram</td>
<td>association, class, dependency, generalization, interface, realization</td>
</tr>
<tr>
<td></td>
<td>design view</td>
<td>internal structure</td>
<td>connector, interface, part, port, provided interface, role, required interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>collaboration diagram</td>
<td>connector, collaboration, collaboration use, role</td>
</tr>
<tr>
<td></td>
<td></td>
<td>component diagram</td>
<td>component, dependency, port, provided interface, realization, required interface, subsystem</td>
</tr>
<tr>
<td>use case view</td>
<td>use case diagram</td>
<td></td>
<td>actor, association, extend, include, use case, use case generalization</td>
</tr>
<tr>
<td>Major Area</td>
<td>View</td>
<td>Diagram</td>
<td>Main Concepts</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------</td>
<td>--------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>dynamic</td>
<td>state machine</td>
<td>state machine</td>
<td>completion transition, do activity, effect, event,</td>
</tr>
<tr>
<td></td>
<td>view</td>
<td>diagram</td>
<td>region, state, transition, trigger</td>
</tr>
<tr>
<td></td>
<td>activity view</td>
<td>activity diagram</td>
<td>action, activity, control flow, control node,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data flow, exception, expansion region, fork,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>join, object node, pin</td>
</tr>
<tr>
<td></td>
<td>interaction</td>
<td>sequence</td>
<td>occurrence specification, execution specification,</td>
</tr>
<tr>
<td></td>
<td>view</td>
<td>diagram</td>
<td>interaction, interaction fragment, interaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operand, lifeline, message, signal</td>
</tr>
<tr>
<td></td>
<td>communication</td>
<td>collaboration</td>
<td>collaboration, guard condition, message, role,</td>
</tr>
<tr>
<td></td>
<td>diagram</td>
<td></td>
<td>sequence number</td>
</tr>
</tbody>
</table>
Basic diagrams explained by example model for a social network

- Class diagram
- Use case diagram
- Object diagram
- State chart diagrams
- Sequence diagram and communication diagram (Interaction)
- Activity diagram
- Diagrams enriched and made precise by expressions written in OCL (Object Constraint Language) being part of UML
- OCL expression itself without side-effect: no system state change
- Operation definitions, class invariants, operation pre- and postconditions (operation contracts), ...
- Explained with UML-based Specification Environment (USE)
UML-based Specification Environment (USE)

- steps to install USE under Windows XY
- google: use ocl bremen
- -> https://sourceforge.net/projects/useocl/
- download zip file "use-X.Y.Z.zip"; save file on Desktop
- unzip file to Desktop directory use-X.Y.Z
- on Desktop context menu "new link" (e.g. "use-4.2.0"); let the link point to:
  - use-X.Y.Z/bin/start_use.bat
- double click your link to start USE with an empty model
- USE offers CLI and GUI
- optional: adjust CLI via properties (font, colors, ...)


USE and OCL for the impatient: 9 OCL expressions on the CLI

• ?21+21
• ?20.9+21.1
• ?20.9+21.1=42
• '?for'+'tytwo'
• ?Set{7,9,5}
• ?Bag{7,9,5,7}
• ?Bag{7,9,5,7}=Bag{5,7,9}
• ?Set{7,9,5,7}=Set{5,7,9}
• ?Set{7,9}->union(Set{9,5})->select(i|i<9)
• OCL: datatypes, collections, operations
Profile
- firstN: String
- lastN: String
- userN: String
- initials: String

init(aFirstN: String, aLastN: String, aUserN: String)
invite(anInvitee: Profile)
accept(anInviter: Profile)
decline(anInviter: Profile)
publish(aPostText: String): Posting
comment(aPosting: Posting, aComment: String)
friends(): Set(Profile)
friendship(anInviter: Profile): Friendship

Subject
- subject: String

Interest
- * profile
- * subject

Friendship
- status: String
- acceptF()
declineF()

Posting
- posting: String

Commenting
- comment: String

PosterPosting
- * posting

* commenter

1 poster

* invitee

* inviter
Class diagram concepts (used in the example)

- USE diagrams customizable via context menu; layout storable
- Class, attribute, operation, parameter, (return) type
- OCL collection kind Set(T),
  more collection kinds: Bag(T), Sequence(T), ...
  superclass Collection(T) > Set(T), Collection(T) > Bag(T), ...
- Binary association, association class, reflexive association,
  composition, ..., generalization, aggregation
- composition, aggregation: part-whole relationships, acyclic on objects, composition strong binding (0..1), aggregation weak binding (0..*)
- Role, association name, multiplicities
- Roles used for navigation from one object to other objects
Example for class diagram concepts (generalization, multiplicities)
class Profile

operations
  friends():Set(Profile)=
      friendship[inviter]->select(oclInState(accepted)).invitee->union(
          friendship[invitee]->select(oclInState(accepted)).inviter)->asSet()
  friendship(anInviter:Profile):Friendship=
      self.friendship[invitee]->any(fs|fs.inviter=anInviter)

constraints
  inv asymmetricFriendship: invitee->intersection(inviter)->isEmpty()
  inv uniqueUserName: Profile.allInstances->isUnique(userN)

• class invariant with name and boolean OCL expression
• query operation definiton; class-valued or collection-valued
• roles
• collection operations allInstances, intersection, isEmpty, isUnique, ...
Use case diagram and concepts

- Actor, use case ("specification of an action sequence"), attributes
- Use case relationships: include (mandatory, once), extend (optional, repeatable), **generalization**
- not supported in USE
- Object (class), link (association), attribute value (attribute)
- Instance level in object diagram, type level in class diagram
State chart diagram and concepts

- State, state invariant
- Initial, final, normal state
- State transition: [ guard ] event / [ postcondition ]
- determine object life cycles
- determined here: protocol state machines; operation call sequences
State chart diagram and concepts: guard

- State transition: [ guard ] event / [ postcondition ]
- guard, also called precondition: [ not aSpouse.oclInState(married) ] marry(aSpouse)
Object diagram with roles, association names, state chart status
Sequence diagram:

- `merkel:Profile`
- `putin:Profile`
- `trump:Profile`
- `:Friendship`

**Command list:**

1. `!new Profile('merkel')`
2. `!new Profile('putin')`
3. `!new Profile('trump')`
4. `!merkel.init('Angela', 'Merkel', 'muddi')`
5. `!putin.init('Vladimir', 'Putin', 'crab')`
6. `!trump.init('Donald', 'Trump', 'theDonald')`
7. `!putin.invite(merkel)`
8. `!putin.invite(putin)`
9. `!putin.decline(trump)`
10. `declineF()`
Sequence diagram concepts

- **Object** (or object role)
- **Lifeline**
- **Activation**
- **Message representation:** solid arrow from caller to callee indicating message call, dashed arrow from callee to caller indicating message completion, optional with return value
- **Link representation:** link shown as link object with lifeline
Communication diagram and concepts

- Object, message, message number, link representation
Both sequence diagrams and communication diagrams show interactions, but they emphasize different aspects. A sequence diagram shows time sequence as a geometric dimension, but the relationships among roles are implicit. A communication diagram shows the relationships among roles geometrically and relates messages to the connectors, but time sequences are less clear because they are implied by the sequence numbers. Each diagram should be used when its main aspect is the focus of attention.

UML Reference Manual, p. 40
Activity diagram and concepts

```
publishSeq(aPostTextSeq:Sequence(String))
begin
    declare p:Posting, s:String;
    for s in aPostTextSeq do
        p := new Posting();
        p.posting := s;
        insert(self, p) into PosterPosting
    end
end
```

- Initial, final, decision, action node; not supported in USE
class Profile
attributes
    firstN: String init: ''
    ...
    initials: String derived:
        firstN.substring(1, 1).concat(lastN.substring(1, 1))
operations
    init(aFirstN: String, aLastN: String, aUserN: String)
    begin
        self.firstN := aFirstN; self.lastN := aLastN; self.userN := aUserN
    end
    pre aUserNNonEmpty: aUserN<>''
    post userNAssigned: aUserN=userN
    ...
constraints
    inv uniqueUserName: Profile.allInstances->isUnique(userN)
    ...
statemachines
    psm ProfileLife
    states
        born [userN='']
        ...
    transitions
        born -> living { init() }
        ...
end
Textual model definition in USE (part B)

associationclass Friendship between
    Profile [*] role inviter
    Profile [*] role invitee
attributes
    status:String init:'pending'
...
end

composition PosterPosting between
    Profile [1] role poster
    Posting [*] role posting
end

associationclass Commenting between
    Profile [*] role commenter
    Posting [*] role commented
attributes
    comment:String
...
end

association Interest between
    Profile [*]
    Subject [*]
end
model SocialNetwork

class Profile
attributes
    firstN: String init: ''
    lastN: String init: ''
    userN: String init: ''
    initials: String derived:
        firstN.substring(1,1).concat(lastN.substring(1,1))

operations
    init(aFirstN: String, aLastN: String, aUserN: String)
        begin
            self.firstN:=aFirstN; self.lastN:=aLastN; self.userN:=aUserN end
        pre  aUserNNonEmpty: aUserN<>''
        post userNAssigned: aUserN=userN
    invite(anInvitee: Profile)
        begin
            new Friendship between (self, anInvitee) end
        pre  notAlreadyTried: invitee->union(inviter)->excludes(anInvitee)
        post madeFS: friendship[inviter]->
            select(oclInState(pending)).invitee->includes(anInvitee)
    accept(anInviter: Profile)
        begin
            self.friendship(anInviter).acceptF() end
        pre  pendingFS: friendship[invitee]->
            select(oclInState(pending)).inviter->includes(anInviter)
        post acceptedFS: friendship[invitee]->
            select(oclInState(accepted)).inviter->includes(anInviter)
Textual model definition in USE (complete model part B)

decline(anInviter:Profile)
  begin self.friendship(anInviter).declineF() end
pre pendingFS: friendship[invitee]->
  select(oclInState(pending)).inviter->includes(anInviter)
post declinedFS: friendship[invitee]->
  select(oclInState(declined)).inviter->includes(anInviter)
publish(aPostText:String):Posting
  begin declare p:Posting;
  p:=new Posting(); p.posting:=aPostText;
  insert(self,p) into PosterPosting; result:=p
  end
pre nonEmpty: aPostText<>''
post newPosting: Posting.allInstances->exists(p |
  p.posting=aPostText and result=p)
comment(aPosting:Posting,aComment:String)
  begin declare c:Commenting;
  c:=new Commenting between (self,aPosting); c.comment:=aComment
  end
pre aPostingNonNullACommentNonEmpty: aPosting<>null and aComment<>''
post commentingExists: Commenting.allInstances->exists(c |}
  c.comment=aComment and aPosting.commenting->includes(c) and
  self.commenting->includes(c))
Textual model definition in USE (complete model part C)

friends():Set(Profile)=
   friendship[inviter]->select(oclInState(accepted)) . invitee->union(
   friendship[invitee]->select(oclInState(accepted)) . inviter)->asSet()

friendship(anInviter:Profile):Friendship=
   self.friendship[invitee]->any(fs|fs.inviter=anInviter)

constraints
   inv asymmetricFriendship: invitee->intersection(inviter)->isEmpty()
   inv uniqueUserName: Profile.allInstances->isUnique(userN)

statemachines
   psm ProfileLife
   states
      prenatal: initial
      born [userN='']
      living [userN<>'']
   transitions
      prenatal -> born  { create }
      born  -> living  { init() }
      living -> living  { invite() }
      living -> living  { accept() }
      living -> living  { decline() }
      living -> living  { publish() }
      living -> living  { comment() }
   end
end
Textual model definition in USE (complete model part D)

association class Friendship between
  Profile [*] role inviter
  Profile [*] role invitee
attributes
  status: String init: 'pending'
operations
  acceptF()
    begin self.status := 'accepted' end
  declineF()
    begin self.status := 'declined' end
statemachines
  psm FriendshipLife
  states
    prenatal: initial
    pending
    accepted: final
    declined: final
  transitions
    prenatal -> pending { create }
    pending -> accepted { acceptF() }
    pending -> declined { declineF() }
end
end
Textual model definition in USE (complete model part E)

composition PosterPosting between
   Profile [1] role poster
   Posting [*] role posting
end

class Posting
attributes
   posting: String
end

associationclass Commenting between
   Profile [*] role commenter
   Posting [*] role commented
attributes
   comment: String
end

constraints

context Commenting inv commentOnlyByFriends:
   commented.poster.friends() -> includes(commenter)
class Subject
attributes
  subject: String
constraints
  inv noDuplicates:
    Subject.allInstances->size=Subject.allInstances.subject->asSet->size
end

association Interest between
  Profile [*]
  Subject [*]
end
Scenario definition on the USE shell with SOIL statements

```
!create merkel,putin,trump:Profile
!merkel.init('Angela','Merkel','muddi')
!putin.init('Vladimir','Putin','crab')
!trump.init('Donald','Trump','theDonald')
!putin.invite(merkel)
!trump.invite(putin)
!putin.decline(trump)
!merkel.accept(putin)
!p:=merkel.publish('BMW, we have a problem')
!create may:Profile
!may.init('Theresa','May','motherTheresa')
!putin.comment(p,'May the Donald be with you')
!may.invite(merkel)
```

• SOIL: Simple Ocl-like Imperative Language
• Object creation and destruction; link creation and destruction: 'create' / 'new', 'destroy', 'insert', 'delete'
• Variable declaration 'declare'; assignment ':=' with OCL expressions
• Loop/If: 'for' var 'in' collection 'do' ... 'end'; 'if' ... 'then' ... ['else' ...] end
• (Recursive) calls object.operation(parameter)
• SOIL statements on USE shell starting with '!'
Communication diagram

1: create
4: init('Angela','Merkel','muddi')
10: accept(putin)
11: publish('BMW, we have a problem')
12: create
13: init('Theresa','May','motherTheresa')
15: invite(merkel)

3: create
6: init('Donald','Trump','theDonald')
8: invite(putin)
2: create
5: init('Vladimir','Putin','crab')
7: invite(merkel)
9: decline(trump)
14: comment(Posting1,'May the Donald be with you')

7.1: insert(@putin,@merkel)
8.1: insert(@trump,@putin)
10.1: acceptF()
9.1: declineF()
Evaluation of OCL expressions (Part A)

-- allInstances -----------------------------------------------

?Profile.allInstances
Set{may,merkel,putin, trump}:Set(Profile)

-- includes, excludes -----------------------------------------

?Set{may,merkel, trump}->includes(putin)
false

?Set{putin}->excludes(trump)
true
Evaluation of OCL expressions (Part B)

-- select, reject -----------------------------------------------

?Profile.allInstances->select(p | p.subject->includes(opera))
Set{may,merkel}:Set(Profile)

?Profile.allInstances->reject(p | p.subject->includes(hairstyle))
Set{putin}:Set(Profile)

-- size, isEmpty, notEmpty -------------------------------------

?Profile.allInstances->select(p | p.subject->size=3)
Set{may,merkel}:Set(Profile)

?Subject.allInstances->select(s | s.profile->size=0)
Set{ocl}:Set(Subject)

?Subject.allInstances->select(s | s.profile->isEmpty)
Set{ocl}:Set(Subject)

?Subject.allInstances->select(s | s.profile->notEmpty)
Set{hairstyle,horses,opera,shoes}:Set(Subject)
Evaluation of OCL expressions (Part C)

-- forAll, exists ---------------------------------------------------------------

?Subject.allInstances->forAll(s | s.profile->notEmpty)
false:Boolean

?Profile.allInstances->select(p | Subject.allInstances->exists(s1,s2 |
   s1<>s2 and s1.profile->includes(p) and s2.profile->includes(p)))
Set{may,merkel}:Set(Profile)

?Profile.allInstances->select(p | Subject.allInstances->exists(s1,s2 |
   s1<>s2 and p.subject->includes(s1) and p.subject->includes(s2)))
Set{may,merkel}:Set(Profile)

-- collectNested, collect, asSet -----------------------------------------------

?Profile.allInstances->collectNested(p | p.subject)
Bag{Set{hairstyle},
   Set{horses},
   ?Set{-2,0,2}->collect(i | i*i)
   Set{hairstyle,opera,shoes},
   Set{hairstyle,opera,shoes}}:Bag(Set(Subject))

?Profile.allInstances->collect(p | p.subject)
Bag{hairstyle,hairstyle,hairstyle,horses,opera,opera,shoes,shoes}:
   Bag(Subject)

?Profile.allInstances->collect(p | p.subject)->asSet -- SQL distinct
Set{hairstyle,horses,opera,shoes} : Set(Subject)
Evaluation of OCL expressions (Part D)

--- including, excluding -----------------------------------------------

?Set{putin, merkel} -> including(may)
Set{may, merkel, putin}: Set(Profile)

?Set{putin, merkel} -> excluding(putin)
Set{merkel}: Set(Profile)

?Set{putin, merkel} -> excluding(may)
Set{merkel, putin}: Set(Profile)

?Bag{opera, shoes, opera} -> including(shoes)
Bag{opera, opera, shoes, shoes}: Bag(Subject)

?Bag{opera, shoes, opera} -> excluding(opera) -- excluding radical on Bag(T)
Bag{shoes}: Bag(Subject)

-- ( includesAll, excludesAll ) = ( 'supersetOf', 'disjointFrom' ) ------

?Set{opera, shoes, hairstyle} -> includesAll(Set{opera, hairstyle}) = true

?Set{opera, shoes, hairstyle} -> includesAll(Set{opera, horses}) = false

?Set{opera, shoes} -> excludesAll(Set{horses, hairstyle}) = true

?Set{opera, shoes} -> excludesAll(Set{horses, opera}) = false
Evaluation of OCL expressions (Part E)

-- let, Tuple, product -----------------------------------------------

-- Profile objects having interest in a given set of subjects
?let INTEREST=Set{hairstyle,opera} in Profile.allInstances->select(p |
    INTEREST->forAll(s | p.subject->includes(s)))
Set{may,merkel}:Set(Profile)

-- Profile objects with a maximum number of interests
?let MAX=Profile.allInstances->collect(p | p.subject->size)->max() in
  Profile.allInstances->select(p | p.subject->size=MAX)
Set{may,merkel}:Set(Profile)

-- Profile object pairs with the same set of interests
?Profile.allInstances->product(Profile.allInstances)
Set{Tuple{first=may,second=may},
    Tuple{first=may,second=merkel},
    ...
    Tuple{first=trump,second=trump}} : Set(Tuple(first:Profile,second:Profile))  -- 16 tuples

?Profile.allInstances->product(Profile.allInstances)->
  select(t | t.first.subject=t.second.subject and t.first<>t.second)
Set{Tuple{first=may,second=merkel},Tuple{first=merkel,second=may}}:
  Set(Tuple(first:Profile,second:Profile))
Evaluation of OCL expressions (Part F)

-- navigation in plain association class --
Evaluation of OCL expressions (Part G)

-- navigation in reflexive association class ----------------------------------------

Profile::inviter : Set(Profile)
Profile::invitee : Set(Profile)
Profile::friendship[inviter] : Set(Friendship)
Profile::friendship[invitee] : Set(Friendship)
Friendship::inviter : Profile
Friendship::invitee : Profile

merkel.inviter = Set{may, putin}
merkel.invitee = Set{} : Set(Profile)
merkel.frienship[inviter] = Set{} : Set(Friendship)
merkel.frienship[invitee] = Set{Friendship1, Friendship3}
Friendship1.inviter = putin
Friendship1.invitee = merkel
Evaluation of OCL expressions (Part H)
Evaluation of OCL expressions (Part I)

-- dot shortcut --------------------------------------------------------

?merkel.inviter                 ?merkel.inviter->collect(p | p.userN)
Set{may,putin}: Set(Profile)    Bag{'crab','motherTheresa'}: Bag(String)

?merkel.inviter.userN -- dot shortcut on single object
Bag{'crab','motherTheresa'}: Bag(String)

?Set{merkel}.inviter.userN -- dot shortcut on object collection
Bag{'crab','motherTheresa'}: Bag(String)

?Profile.allInstances.inviter.userN -- dot shortcut on object collection
Bag{'crab','motherTheresa','theDonald'}: Bag(String) -- excludes 'muddi'

?trump.invitee.invitee.inviter -- long path; object-valued result
Bag{may,putin}: Bag(Profile)

?trump.invitee.invitee.posting.posting -- long path; data-valued result
Bag{'BMW, we have a problem'}: Bag(String)

?Posting1.poster.initials -- respect multiplicities; gives single value
'AM' : String

?Posting1.poster.inviter -- respect multiplicities; gives Set(T)
Set{may,putin}: Set(Profile)
Derived attributes, query operations, invariants: Applying OCL

Derived attributes

Profile::initials:String derived:
   firstN.substring(1,1).concat(lastN.substring(1,1))

Query operations

Profile::friends():Set(Profile)=
   friendship[inviter]->select(oclInState(accepted)).invitee->union(
      friendship[invitee]->select(oclInState(accepted)).inviter)->asSet
Profile::friendship(anInviter:Profile):Friendship=
   friendship[invitee]->any(fs|fs.inviter=anInviter)

Invariants

context Profile
   inv asymmetricFriendship: invitee->intersection(inviter)->isEmpty()
   inv uniqueUserName: Profile.allInstances->isUnique(userN)

context Commenting inv commentOnlyByFriends:
   commented.poster.friends()-->includes(commenter)

context Subject inv noDuplicates:
   Subject.allInstances->size=Subject.allInstances.subject->asSet->size
Object diagram with violated invariants

Class invariants

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commenting::commentOnlyByFriends</td>
<td>false</td>
</tr>
<tr>
<td>Profile::asymmetricFriendship</td>
<td>false</td>
</tr>
<tr>
<td>Profile::uniqueUserName</td>
<td>false</td>
</tr>
<tr>
<td>Subject::noDuplicates</td>
<td>false</td>
</tr>
</tbody>
</table>
Thanks for your attention!