# 09- Notation-Formal & UML Intro

Rick Adrion



## Concurrent & distributed systems

- FSA
- Petri nets
- Trace specifications
  - a trace is a sequence of procedure or function calls and return values from those calls
    - proposed by David Parnas, 1977
    - formalized by McLean, 1984
    - further developed by Dan Hoffman, Rick Snodgrass, etc



## Trace specifications

**NAME**
label
**SYNTAX**
name: __type ... __type ⇒ return_value_type
**SEMANTICS**
assertions of the form:
L(T) -- asserts that T is a legal trace
V(T) = *value* -- is the value returned if T ends in a function call

- operator precedence

≡ < “ = ≥ >

¬

& ~ |

⇒ ⇔



## Trace specifications

**T1 ≡ T2 ⇒**

(∀T) ((L(T1·T) ⇒ L(T2·T)) &

(T is not empty ⇒ (

($T_1$·T has a value ⇔ $T_2$·T has a value) &

($T_1$·T has a value ⇒V($T_1$·T)= V($T_2$·T))))

note (∀S,T) (L(S·T) ⇒ L(S))

## Example

**NAME**
stack
**SYNTAX**
push: integer;
pop: ;
top: ⇒ integer;
**SEMANTICS**
/*1*/ (∀T,i) (L(T) ⇒ L(T·push(i))
/*2*/ (∀T) (L(T·top) ⇔ L(T·pop)
/*3*/ (∀T,i) (T ≡T·push(i)·pop)
/*4*/ (∀T) (L(T·top) ⇒ T ≡ T·top)
/*5*/ ( ∀T,i) (L(T) ⇒V(T·push(i)·top)=i)



## Interpretation

**/*1*/ (∀T,i) (L(T) ⇒ L(T·push(i))**
/*1*/ unbounded stack
**/*2*/ (∀T) (L(T·top) ⇔ L(T·pop)**
/*2*/ top cause no error iff pop causes no error
**/*3*/ (∀T,i) (T ≡T·push(i)·pop)**
/*3*/ push followed by pop does not affect the future behavior
**/*4*/ (∀T) (L(T·top) ⇒ T ≡ T·top)**
/*4*/ top does not affect the behavior
**/*5*/ (∀T,i) (L(T) ⇒V(T·push(i)·top)=i)**
/*5*/ how to compute the value of top



## Example - using /*3*/ and /*5*/

*note: push(i)·push(j)·push(k)·pop·pop·top ⇒ top= i*

***By*** /*3*/ (∀T,i) (T ≡T·push(i)·pop)

***By*** /*5*/ (∀T,i) (L(T) ⇒V(T·push(i)·top)=i)



## Heuristics

- define normal forms
- structure semantics
- use predicates
- develop specs incrementally
- use macros

## Comparison

| trace specifications | algebraic specifications |
|---|---|
| based on call sequence | based on "type of interest," therefore maybe in terms of objects not visible to user |
| no "hidden functions" | requires "hidden functions" |
| natural application to inter-process communication | cannot handle concurrency |
| universal & existential quantifiers | no existential quantification |



## Property-oriented techniques

- Abstract-data-type specification languages
  - Axiomatic: Hoare, OBJ, Anna, Larch, and algebraic, e.g., Clear, ActOne, Aspeque
  - Concurrent and distributed systems specification languages: temporal logic, Lamport, LOTOS
- Semi-formal
  - ER diagrams



## Logic Specifications

- Expressed using formulas under a first order logic theory (usually with quantification), e.g.,
  - $\exists$ j [1 $\leq$ j $\leq$ s.top| t.data[j]=s.data[j]]
- Typically expressed as pre- and post-conditions, e.g.,
  - Let P be a sequential program
  - with inputs $(i_0, i_1, \ldots, i_n)$ and outputs $(o_0, o_1, \ldots, o_m)$
  - Pre $(i_0, i_1, \ldots, i_n)$ P Post $(o_0, o_1, \ldots, o_m, i_0, i_1, \ldots, i_n)$ is a property



## "Hoare" example

type stack =
    record top: integer
        data:array [1 ... 100] of integer
    end

t:= push(s, i)

true{t:= push(s, i)} $\exists$ j [1$\leq$ j$\leq$s.top| t.data[j]=s.data[j]
    $\wedge$ t.data[t.top] = i
    $\wedge$ t.top =s.top +1]

precondition

"program"

post condition

## "Hoare" example

Logic specification:

true {t:= push(s, i)} ∃ j [1 ≤ j ≤ s.top| t.data[j]=s.data[j]

∧ t.data[t.top] = I ∧ t.top =s.top +1]

Operational specification

{true} push ($S_0$, I) {∀ J, 1 < J ≤ $S_0$.top

$S_0$.data [J] = S.data [J] ∧

S.top = $S_0$.top + 1 ∧

S.Data [S.top] = I }



## Algebraic Specification

Stack (S) ∧ Integer (I) …

(1) Top (Push (S, I)) = I

(2) Top (Create) = Integer Error

(3) Pop (Push (S, I)) = S

(4) Pop (Create) = Stack Error

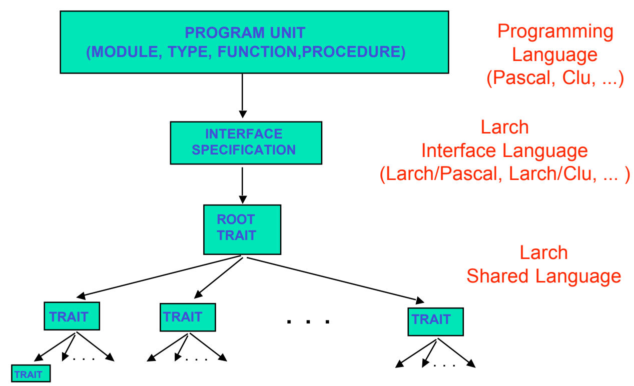


## Larch

- The Larch Family of Specification Languages
  - John Guttag, James Horning, Jeannette Wing IEEE Software, 1985
- Larch Shared Language
  - Common language for formally representing models
- Larch Interface Language
  - Interface between the shared language and the target programming language
    - Larch/Pascal
    - Larch/CLU
- Specific implementation language



## Larch



## Terminology

| SPECIFICATION TERM | PROGRAMMING LANGUAGE TERM |
|---|---|
| Operator | Function |
| Sort | Type |
| Term | Expression |
| Trait | Module (ADT), Function, Procedure type |



## Goals of Larch

- Composability
  - Common specifications from existing specifications
  - Library or handbook
- Readability
- Localize programming language dependence
  - General model is very complex so use different language specific models
- Automated Support
  - Construction tool
  - Syntactic checking
  - Semantic checking
  - Support incompleteness



## Trait

Introduces — signature of the operation (sort checking)

Constrains — constrains the operations & relations among the operators

theory - set of theorems that can be proved about the operator done by substitution, using rules of first order predicate calculus with equality



## Examples

Container: **trait**
**introduces**
new: → C
insert: C, E → **C**
**constrains** C **so that**
C **generated by** [ new, insert ]

## Examples

IsEmpty: **trait**
**assumes** Container
**introduces**
isEmpty: C → Bool
**constrains** isEmpty, new, insert
**so that for all** [ *c* :C, *e* :E ]
isEmpty(new) = true
isEmpty(insert(c,e)) = false
**implies converts** [ isEmpty ]

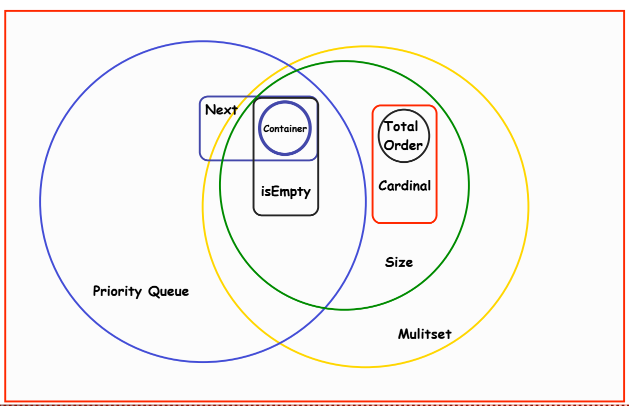


## Extended example

```
Container (E, C): trait
  % head and tail enumerate contents of a C
  includes InsertGenerated, Integer
  introduces
    isEmpty: C -> Bool
    count: E, C -> Int
    __ \in __: E, C -> Bool
    head: C -> E
    tail: C -> C
  asserts
    C partitioned by isEmpty, head, tail
    forall e, el: E, c: C
      isEmpty(empty);
      ~isEmpty(insert(e, c));
      count(e, empty) == 0;
      count(e, insert(el, c)) ==
        count(e, c) + (if e = el then 1 else 0);
      e \in c == count(e, c) > 0;
      ~isEmpty(c) =>
        count(e, insert(head(c), tail(c)))
          = count(e, c)
  implies
    forall c: C
      ~isEmpty(c) => count(head(c), c) > 0;
    converts isEmpty, count, \in
```

University of Massachusetts Amherst · Department of Computer Science

## Constructing traits




University of Massachusetts Amherst · Department of Computer Science

## Interface Languages

- "bridge" between shared language and implementation language
- "Two-tiered" specification approach: principal innovation of Larch w/r/t algebraic specification languages

University of Massachusetts Amherst · Department of Computer Science

## Interface Languages

- Larch/L incorporates “flavor” of L
  - semantics, keywords
  - makes it easier for those who know L to write provable specs
  - just need to adapt existing shared traits from Library (in theory...)
- Larch/L languages designed to support data abstraction, even if language L doesn’t directly support it (Pascal, C, C++)



## Larch/Pascal specification

```
type Bag exports bagInit, bagAdd, bagRemove, bagChoose
  based on sort Mset from MultiSet with [integer for E]
  procedure bagInit(var b:Bag)
        modifies at most [ b ]
        ensures bpost = { }
  procedure bagAdd(var b:Bag; e; integer)
        requires numElements(insert(b,e )) ≤ 100
        modifies at most [ b ]
        ensures bpost = insert(b,e )
  procedure bagRemove(var b:Bag; e; integer)
        modifies at most [ b ]
        ensures bpost = delete(b,e )
  procedure bagChoose(var b:Bag; e; integer): boolean
        modifies at most [ b ]
        ensures if ~ isEmpty (b )
         then bagChoose & count (b, epost)>0
         else ~ bagChoose & modifies nothing
End Bag
```



## Pascal implementation of BagAdd

```
prodedure bagAdd(var B:Bag;e:integer);
  var i, lastEmpty: 1...MaxBagSize
  begin
    i:= 1;
    while ((i < MaxBagSize) and (b.elems[i]<>e)) do
      begin
        if b.counts[i] = 0 then LastEmpty:=i;
        i:= i+1;
      end;
    if b.elems[i] = e
      then b.counts[i]:= b.counts[i]+1;
      else begin
        if b.counts[i]=0 then LastEmpty:=i;
        b.elems[LastEmpty]:=e;
        b.counts[LastEmpty]:=1;
      end;
end[bagAdd];
```



## Conclusions

- Interesting attempt to address:
  - readability/writability of formal specs
  - large, multi-lingual environment issues
- Relationship between shared and interface languages complex and unclear
- Relationship between interface and implementation languages not as strong as one would like
- “Software tool support needed” (syntax-directed editors, browsers, theorem-provers, etc.)

## Current Status of Formal Methods

- Strong theoretical foundation
- Some practical use, especially in Europe
- Current Languages trying to be more practical



## How effective are these methods?

- Wing's study of the Library Problem
  - a small library database
  - transactions
    - checkout/return book
    - add/remove book
    - get a list of books
      - author
      - subject
      - borrower
    - get date/borrower for book
  - users
    - staff
    - borrowers
  - restrictions
    - availability
    - no book available & checked out
    - # books borrowed ≤max



## Analysis

- Specification approaches
  - informal
  - AI
  - logic
  - executable/non-executable
- Comparisons
  - formality
  - life-cycle phase
  - operational vs. behavioral
  - modularity
  - readability
  - completeness
- Not considered
  - concurrency
  - reliability
  - fault-tolerance
  - security
- initialization
  - what's the initial state of the library?
- missing operations
  - need more transactions?
- error handling
  - what to do with errors?
  - checkout, return, add, remove, "type errors"
- missing constraints
  - more than one copy in library, checked out
- state
  - what to record, change?
- "non-functional" specification
  - human factors, liveness, time



## Conclusions

- methods do not differ radically
- style
  - most use pre- and post-conditions for specifying behavior
  - algebraic & set-theoretic most common for specifying data (operational)
  - model-oriented (operational) most common approach
- formal specs can
  - identify diff in informal specs
  - handle simple, small problems
  - specify sequential functional behavior
- Challenges
  - scaling
  - non-functional behavior
  - combining techniques
  - tools
  - integrating specification into the lifecycle

## Note - UML overheads are adapted from


- "Introduction to UML: Structural and Use Case Modeling," Cris Kobryn, Co-Chair UML Revision Task Force Object Modeling with OMG UML Tutorial Series © 1999-2001 OMG and Contributors: Crossmeta, EDS, IBM, Enea Data, Hewlett-Packard, IntelliCorp, Kabira Technologies, Klasse Objecten, Rational Software, Telelogic, Unisys
- "Behavioral Modeling," Gunnar Övergaard, Bran Selic, Conrad Bock and Morgan Björkande, UML Revision Task Force, Object Modeling with OMG UML Tutorial Series © 1999-2001 OMG and Contributors: Crossmeta, EDS, IBM, Enea Data, Hewlett-Packard, IntelliCorp, Kabira Technologies, Klasse Objecten, Rational Software, Telelogic, Unisys
- MACIASZEK, L.A. (2001): Requirements Analysis and System Design. Developing Information Systems with UML, Addison Wesley Copyright © 2000 by Addison Wesley
- "Analysis and Design with UML," Rational Copyright © 1997 by Rational Software Corporation
- "Practical UML: A hands-on introduction for developers," Copyright © 2002 TogetherSoft, Inc.




## UML Overview

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting
- the artifacts of software systems

University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

## UML Goals

- Define an easy-to-learn but semantically rich visual modeling language
- Unify the Booch, OMT, and Objectory modeling languages
- Include ideas from other modeling languages
- Incorporate industry best practices
- Address contemporary software development issues
  - scale, distribution, concurrency, executability, etc.
- Provide flexibility for applying different processes
- Enable model interchange and define repository interfaces

University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

## Why is UML important?

- Analogy
  - Architects design buildings
  - Builders use the designs to create buildings
  - Blueprints are the standard graphical language that both architects and builders must learn as part of their trade
- UML has emerged as the software blueprint language for analysts, designers, and programmers alike
  - provides a common vocabulary to talk about object-oriented software design.



University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2003

## O-O problem solving

- underlying tenet begins with the construction of a model
  - a **model** is an abstraction of the underlying problem
  - the **domain** is the actual world from which the problem comes
- Models consist of **objects** that interact by sending each other **messages**
  - have things they know (**attributes**) and things they can do (**behaviors** or **operations**)
  - values of an object's attributes determine its **state**
- **Classes** are the "blueprints" for objects
  - a class wraps attributes (data) and behaviors (methods or functions) into a single distinct entity
  - objects are **instances** of classes.





## Unifying Concepts in UML

- classifier-instance dichotomy
  - e.g., an object is an instance of a class OR a class is the classifier of an object
- specification-realization dichotomy
  - e.g., an interface is a specification of a class OR a class is a realization of an interface



## Foundation Concepts

- Building blocks - the basic building blocks of UML are:
  - model elements (classes, interfaces, components, use cases, etc.)
  - relationships (associations, generalization, dependencies, etc.)
  - diagrams (class diagrams, use case diagrams, interaction diagrams, etc.)
- Well-formedness rules
  - Well-formed: indicates that a model or model fragment adheres to all semantic and syntactic rules that apply to it.
  - UML specifies rules for:
    - naming
    - scoping
    - visibility
    - integrity
    - execution (limited)
  - However, during iterative, incremental development it is expected that models will be incomplete and inconsistent.



## What is use case modeling?

- use case model
  - a view of a system that emphasizes the behavior as it appears to outside users. A use case model partitions system functionality into transactions (‘use cases’) that are meaningful to users (‘actors’).



## Use-Case diagrams

- emphasis is on *what* a system does rather than *how*
- Use case diagrams are closely connected to scenarios
  - a **scenario** is an example of what happens when someone interacts with the system, e.g.,
    "A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot."
  - a **use case** is a summary of scenarios for a single task or goal
  - an **actor** is who or what initiates the events involved in that task



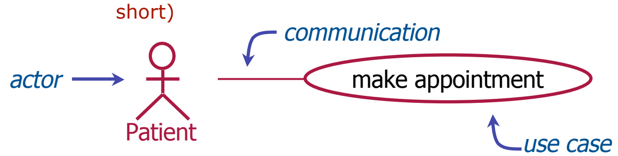
## *Use Case Modeling:* Core Elements

| Construct | Description | Syntax |
|---|---|---|
| use case | A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. | UseCaseName |
| actor | A coherent set of roles that users of use cases play when interacting with these use cases. | ActorName |
| system boundary | Represents the boundary between the physical system and the actors who interact with the physical system. | |



## Example

- **Make Appointment**
  - use case for the medical clinic
    - actor is a **Patient**
    - connection between actor and use case is a **communication association** (or **communication** for short)






## *Use Case Modeling:* Core Relationships

| Construct | Description | Syntax |
|---|---|---|
| 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 | 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. | <<extend>> |

## An example

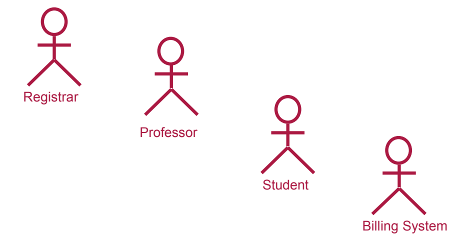
- The ESU University wants to computerize their registration system
  - The Registrar sets up the curriculum for a semester
    - One course may have multiple course offerings
  - Students select 4 primary courses and 2 alternate courses
  - Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
  - Students may use the system to add/drop courses for a period of time after registration
  - Professors use the system to receive their course offering rosters
  - Users of the registration system are assigned passwords which are used at logon validation





## Actors

- An actor is someone or some thing that must interact with the system under development








## Use Cases

- A use case is a pattern of behavior the system exhibits
  - Each use case is a sequence of related transactions performed by an actor and the system in a dialogue
- Actors are examined to determine their needs
  - Registrar -- maintain the curriculum
  - Professor -- request roster
  - Student -- maintain schedule
  - Billing System -- receive billing information from registration





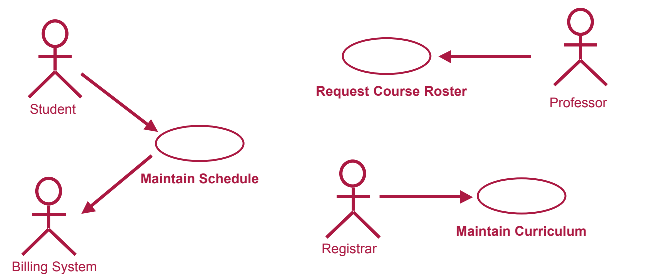

## Documenting Use Cases

- A flow of events document is created for each use cases
  - Written from an actor point of view
- Details what the system must provide to the actor when the use cases is executed
- Typical contents
  - How the use case starts and ends
  - Normal flow of events
  - Alternate flow of events
  - Exceptional flow of events

COMPUTER SCIENCE

## Use Case Diagram

- Use case diagrams are created to visualize the relationships between actors and use cases






COMPUTER SCIENCE

## Maintain Curriculum Flow of Events

- This use case begins when the Registrar logs onto the Registration System and enters his/her password. The system verifies that the password is valid (E-1) and prompts the Registrar to select the current semester or a future semester (E-2). The Registrar enters the desired semester. The system prompts the Registrar to select the desired activity: ADD, DELETE, REVIEW, or QUIT.
- If the activity selected is ADD, the S-1: Add a Course subflow is performed.
- If the activity selected is DELETE, the S-2: Delete a Course subflow is performed.
- If the activity selected is REVIEW, the S-3: Review Curriculum subflow is performed.
- If the activity selected is QUIT, the use case ends.
- ...





COMPUTER SCIENCE

## Documenting use cases

- **Brief Description**
- **Actors** involved
- **Preconditions** necessary for the use case to start
- **Detailed Description** of flow of events that includes:
  - **Main Flow** of events, that can be broken down to show:
    - **Subflows** of events (subflows can be further divided into smaller subflows to improve document readability)
  - **Alternative Flows** to define exceptional situations
- **Postconditions** that define the state of the system after the use case ends



COMPUTER SCIENCE

## Narrative use case specification

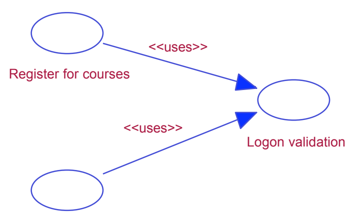
| Use Case | Add a course to the curriculum |
|---|---|
| Brief Description | This use case allows a Registrar to enter a new course. ... |
| Actors | `Registrar` |
| Preconditions | Registrar has a valid password (E-1), has selected a semester default or E-2), and has selected the `Add` (S-1) function at the system prompt |
| Main Flow | The system enters the `Add a Course` subflow |
| Alternative Flows | The `Registrar` activates the `Delete, Review,` or `Quit` functions |
| Postconditions | If the use case was successful, the Registrar has accessed the `Add a Course` function |



## Uses and Extends Relationships

- As the use cases are documented, other use case relationships may be discovered
  - A **uses** relationship shows behavior that is common to one or more use cases
  - An **extends** relationship shows optional behavior








## University Enrolment - Maciaszek

- The **university** offers
  - Undergraduate and postgraduate degrees
  - To full-time and part-time students
- The **university structure**
  - Divisions containing departments
  - Single division administers each degree
  - Degree may include courses from other divisions
- **University enrolment** system
  - Individually tailored programs of study
  - Prerequisite courses
  - Compulsory courses
  - Restrictions
    - Timetable clashes
    - Maximum class sizes, etc.

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

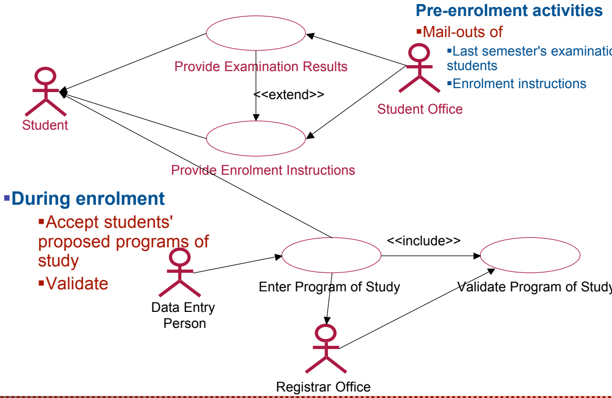
## University Enrolment (cont)

- The system is required to
  - Assist in pre-enrolment activities
  - Handle the enrolment procedures
- **Pre-enrolment activities**
  - Mail-outs of
    - Last semester's examination grades to students
    - Enrolment instructions
- **During enrolment**
  - Accept students' proposed programs of study
  - Validate for prerequisites, timetable clashes, class sizes, special approvals, etc.
- Resolutions to some of the problems may require consultation with academic advisers or academics in charge of course offerings

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

## Example 4.12

**Pre-enrolment activities**
- Mail-outs of
  - Last semester's examination grades to students
  - Enrolment instructions




- **During enrolment**
  - Accept students' proposed programs of study
  - Validate

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

## 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
- When defining use cases in text, use nouns and verbs accurately and consistently to help derive objects and messages for interaction diagrams (see Lecture 2)
- 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
- Large numbers of use cases should be organized into packages



## Use Case Realizations

- The use case diagram presents an outside view of the system
- Interaction diagrams describe how use cases are realized as interactions among societies of objects
- Two types of interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams





## sequence diagram

- an interaction diagram that details how operations are carried out
  - what messages are sent and when
  - are organized according to time
  - time progresses as you go down the page
  - objects involved in the operation are listed from left to right according to when they take part in the message sequence.

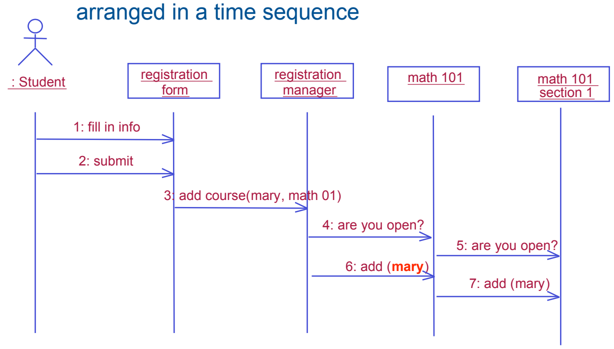
| Symbol | Meaning |
|---|---|
| ⟶ | simple message which may be synchronous or asynchronous |
| - - → | simple message return (optional) |
| ⟶ | a synchronous message |
| ⇀ | an asynchronous message |



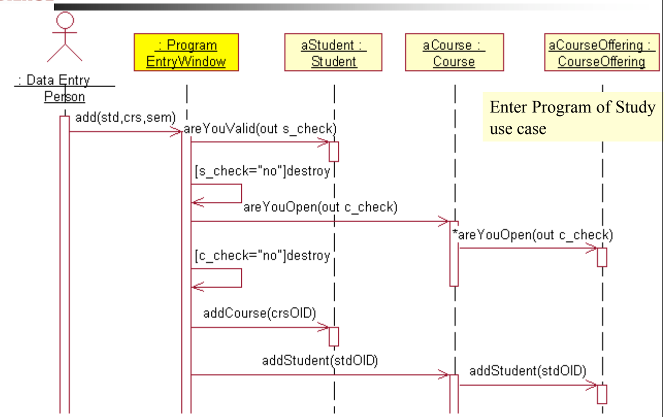
## Sequence Diagram

- A sequence diagram displays object interactions arranged in a time sequence






## Example 4.17 – Maciaszek






## Collaboration Diagrams

- also interaction diagrams
- focus on object roles instead of the times that messages are sent