COMPUTER SCIENCE

13 - Requirements Specifications

Rick Adrion

Software Requirements Specification

- How do we communicate the Requirements to others?
  - It is common practice to capture them in an SRS
  - But an SRS doesn't need to be a single paper document
- Purpose
  - Communicates an understanding of the requirements
  - explains both the application domain and the system to be developed
- Contractual
  - May be legally binding!
  - Expresses an agreement and a commitment
- Baseline for evaluating subsequent products
  - supports system testing, verification and validation activities
  - should contain enough information to verify whether the delivered system meets requirements
- Baseline for change control
  - requirements change, software evolves

Audience

- Users, Purchasers
  - Most interested in system requirements
  - Not generally interested in detailed software requirements
- Systems Analysts, Requirements Analysts
  - Write various specifications that interrelate
- Developers, Programmers
  - Have to implement the requirements
- Testers
  - Determine that the requirements have been met
- Project Managers
  - Measure and control the analysis and development processes

Appropriate Specification

- Consider two different projects:
  - Tiny project, 1 programmer, 2 months work
    - Programmer talks to customer, then writes up a 5-page memo
  - Large project, 50 programmers, 2 years work
    - Team of analysts model the requirements, then document them in a 500-page SRS

	Project A	Project B
Purpose of spec?	Crystalizes programmer's understanding; feedback to customer	Build-to document; must contain enough detail for all the programmers
Management view?	Spec is irrelevant; have already allocated resources	Will use the spec to estimate resource needs and plan the development
Readers?	Primary: Spec author; Secondary: Customer	Primary: programmers, testers, managers; Secondary: customers

What about the SIS RFP/RFB?

- RFP = 'SRS' written by the procurer (or by an independent RE contractor)
  - general enough to yield a good selection of bids
  - specific enough to exclude unreasonable bids
- Bidders response to the RFP
  - specific enough to demonstrate feasibility and technical competence
  - general enough to avoid over-commitment
- Selected developer – more detailed 'SRS'
  - developer's understanding of the customers needs
  - basis for evaluation of contractual performance
  - IEEE Standard recommends SRS jointly developed by procurer and developer

When to issue RFP/RFB?

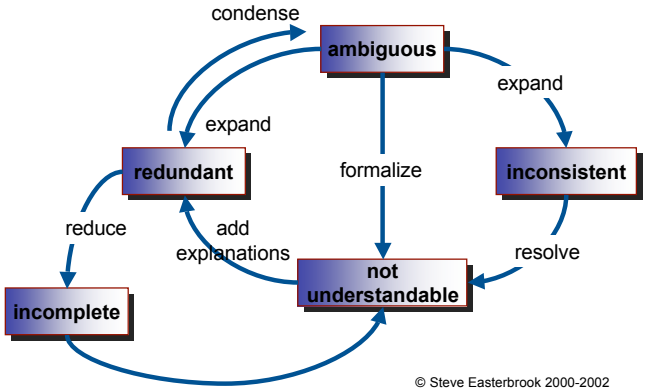
- Early (conceptual stage)
  - Can only evaluate bids on apparent competence & ability
- Late (detailed specification stage)
  - Lots of work for the procurer
  - the appropriate RE expertise may not be available in-house!

Hints & Guidelines

- Validity (or "correctness")
  - expresses only the real needs of the stakeholders (customers, users,…)
- Completeness
  - specifies all the things the system must do and all the things it must not do!
  - Conceptual Completeness
    - e.g. responses to all classes of input
  - Structural Completeness
    - e.g. no "to be determined" functions
- Consistency
  - not self-contradictory
  - satisfiable
  - all terms used consistently
  - inconsistency can be hard to detect especially in timing aspects and business logic
- Necessary
  - doesn't contain anything that isn't "required"
- Ambiquity
  - every statement can be read in exactly one way
  - defines confusing terms
    - e.g. in a glossary
- Verifiablility
  - includes a process exists to test satisfaction of each requirement
  - every requirement is specified behaviorally
- Understandablility (Clarity)
  - e.g. by non-computer specialists
  - technical notations should only be used as backup (e.g. in an appendix)
- Modifiability
  - easy to change and modify
  - good structure and cross-referencing
  - must be kept up to date!

see *IEEE-STD-830-1993*

There is no Perfect SRS



Typical mistakes

- Noise
  - text that carries no relevant information to any feature of the problem.
- Silence
  - a feature that is not covered by any text.
- Over-specification
  - text that describes a feature of the solution, rather than the problem.
- Contradiction
  - text that defines a single feature in a number of incompatible ways.
- Ambiguity
  - text that can be interpreted in at least two different ways.
- Forward reference
  - text that refers to a terms or features yet to be defined.
- Wishful thinking
  - text that defines a feature that cannot possibly be validated.
- Jigsaw puzzles
  - distributing key information across a document and then cross-referencing
- Duckspeak requirements
  - requirements that are only there to conform to standards
- Unnecessary invention of terminology
  - e.g. 'user input presentation function'
  - e.g. 'airplane reservation data validation function'
- Inconsistent terminology
  - inventing and then changing terminology
- Putting the onus on the development staff
  - i.e. making the reader work hard to decipher the intent
- Writing for the hostile reader
  - fewer of these than friendly readers

from Steve Easterbrook © 2000-2002; he *Adapted from Kovitz, 1999*

Use Appropriate Notations

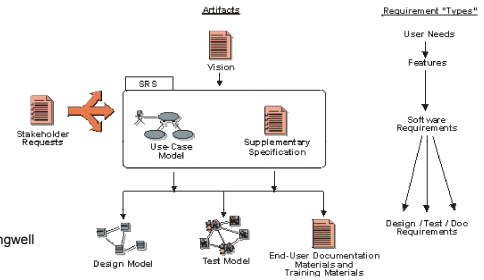
- Natural Language?
  - "The system shall report to the operator all faults that originate in critical functions or that occur during execution of a critical sequence and for which there is no fault recovery response."
    - (adapted from a real NASA spec for the international space station)
- A decision table?

Originate in critical functions	F	T	F	T	F	T	F	T
Occur during critical seqeunce	F	F	T	T	F	F	T	T
No fault recovery response	F	F	F	F	T	T	T	T
Report to operator?								

- Use cases?

SRS using a UML "package" construct

- may include a single document, multiple documents, use case specifications and even the graphical use case model which describes relationships amongst the use cases.
- controls the evolution of the system throughout the development and release phases of the project



Probasco and Leffingwell
Rational Software

SRS using a UML "package" construct

- the following use the SRS Package:
  - The system analyst creates and maintains the Vision, the use-case model overview and supplementary specifications, which serve as input to the SRS and are the communication medium between the system analyst, the customer, and other developers.
  - The use-case specifier creates and maintains the individual use cases of the use-case model and other components of the SRS package,
  - Designers use the SRS Package as a reference when defining responsibilities, operations, and attributes on classes, and when adjusting classes to the implementation environment.
  - Implementers refer to the SRS Package for input when implementing classes.
  - The project manager refers to the SRS Package for input when planning iterations.
  - The testers use the SRS Package to verify system

SRS format and style

- Modifiability
  - well-structured, indexed, cross-referenced, etc.
  - redundancy should be avoided or must be clearly marked as such
  - An SRS is not modifiable if it is not traceable...
- Traceability
  - Need a way of referring to each requirement
  - Backwards - the specification must be "traced"
    - each requirement traces back to a source or authority
    - e.g. a requirement in the system spec; a stakeholder; etc
  - Forward - the specification must be "traceable"
    - each requirement will eventually trace forwards to parts of the design that satisfy it

SRS format and style

- Traceability links are two-way
  - other documents will be traced into the SRS
  - every requirement must have a unique label.
  - a given term, acronym, or abbreviation means the same thing in all documents
  - a given item or concept is referred to by the same name or description in the documents
- In short:
  - demonstration of completeness, necessity and consistency
  - clear allocation/flowdown path (down through the document hierarchy)
  - a clear derivation path (up through the document hierarchy)

IEEE Standard for SRS

1. Introduction
   - Purpose
   - Scope
   - Definitions, acronyms, abbreviations
   - Reference documents
   - Overview
2. Overrall Description
   - Product perspective
   - Product functions
   - User characteristics
   - Constraints
   - Assumptions and Dependencies
3. Specific Requirements

Appendices

Index

Identifies the product & application domain

Describes contents and structure of the remainder of the SRS

Describes all external interfaces: system, user, hardware, software ;also operations and site adaptation, and hardware constraints

Summary of major functions, e.g. use cases

Anything that will limit the developer's options (e.g. regs, reliability, criticality, hardware limitations, parallelism, etc)

All the requirements go in here (i.e.this is the body of the document) IEEE STD provides 8 different templates for this section

IEEE std offers different templates

- External stimulus or external situation
  - e.g., for an aircraft landing system, each different type of landing situation: wind gusts, no fuel, short runway, etc
- System feature
  - e.g., for a telephone system: call forwarding, call blocking, conference call, etc
- System response
  - e.g., for a payroll system: generate pay-cheques, report costs, print tax info;
- External object
  - e.g. for a library information system, organize by book type
- User type
  - e.g. for a project support system: manager, technical staff, administrator, etc.
- Mode
  - e.g. for word processor: page layout mode, outline mode, text editing mode, etc
- Object
  - e.g., in a patient monitorng system, objects include patients, sensors, nurses, rooms, physicians, medicines, etc.

Template of SRS 3 organized by object

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Classes/Objects
3.2.1 Class/Object 1
3.2.1.1 Attributes (direct or inherited)
3.2.1.1.1 Attribute 1
…
3.2.1.1.n Attribute n
3.2.1.2 Functions (services, methods, direct or inherited)
3.2.1.2.1 Functional requirement 1.1
…
3.2.1.2.m Functional requirement 1.m
3.2.1.3 Messages (communications received or sent)
3.2.2 Class/Object 2
…
3.2.p Class/Object p
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Maciaszek vs. IEEE

Requirements Document
Table of Contents

1. **Project Preliminaries**
   1.1 Purpose and Scope of the Product
   1.2 Business Context
   1.3 Stakeholders
   1.4 Ideas for Solutions
   1.5 Document Overview
2. **System Services**
   2.1 The Scope of the System
   2.2 Function Requirements
   2.3 Data Requirements
3. **System Constraints**
   3.1 Interface Requirements
   3.2 Performance Requirements
   3.3 Security Requirements
   3.4 Operational Requirements
   3.5 Political and Legal Requirements
   3.6 Other Constraints
4. **Project Matters**
   4.1 Open Issues
   4.2 Preliminary Schedule
   4.3 Preliminary Budget

Appendices
Glossary
Business Documents and Forms
References

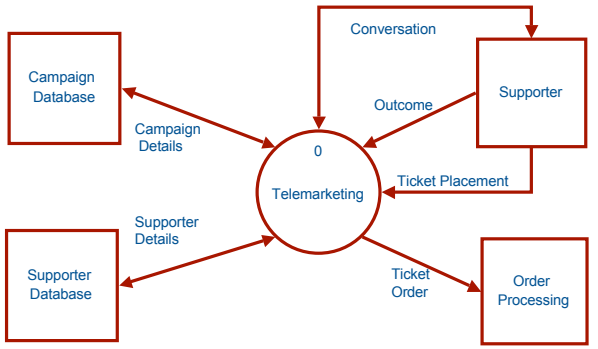
1. Introduction
   - Purpose
   - Scope
   - Definitions, acronyms, abbreviations
   - Reference documents
   - Overview
2. Overrall Description
   - Product perspective
   - Product functions
   - User characteristics
   - Constraints
   - Assumptions and Dependencies
3. Specific Requirements

Appendices
Index

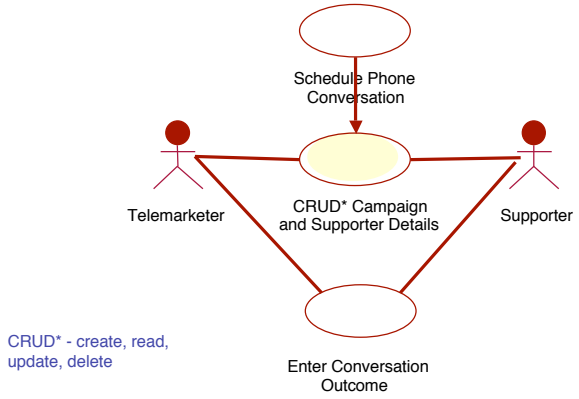
Requirements Negotiation & Validation

- Negotiation
  - Based on draft of document
- Validation
  - Based on (almost) complete document
- Issues
  - Scope
  - Dependencies
  - Risks
  - Priorities

System scope model



Business use case model



Requirements dependency matrix

Requirement	R1	R2	R3	R4
R1	X	X	X	X
R2	Conflict	X	X	X
R3			X	X
R4		Overlap	Overlap	X

Requirements risks

- Technical
- Performance
- Database integrity
- Development process
- Political
- Legal
- Volatility

Credits

- Many of the following are derived from:
  - Kenneth M. Anderson © University of Colorado, 2003
  - Maciaszek **Requirements Analysis and System Design** © Addison Wesley, 2000

Requirements Specification

- Three types of models
  - State Models
    - Use Cases (some actors become classes)
    - Class Diagrams
  - Behavior Models
    - Activity Diagram
    - Interaction Diagrams
  - State Change Models
    - State Chart Diagrams
- Models are developed iteratively
  - accounting for use cases and constraints developed during requirements elicitation
  - each representing a view into the system
  - taken together, the models allow developers and customers to view the system from multiple perspectives

State specifications

- The state of an object is determined by the values of its attributes and associations
  - A BankAccount may be "overdrawn" when its balance is negative
- Since object states are determined from data structures, the models of the data structures (e.g. classes) are called state specifications
- State specifications provide a static view of the system
  - The attributes and associations of classes do not change dynamically
- The main task is to specify the classes of an application domain
  - only attributes and associations; operations are derived from the behavior specification

State Specification

- Define entity classes
  - Persistent classes in the application domain
- Process is highly dependent on the analyst's
  - knowledge of class modeling
  - understanding of the application domain
  - experience with similar and successful designs
  - ability to think forward and predict consequences willingness to revise the model iteratively

Discovering Classes

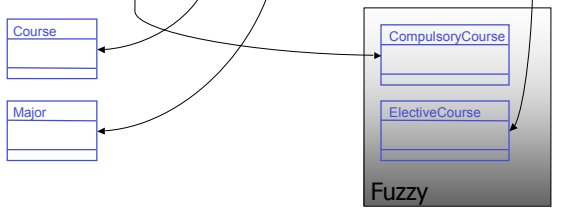
- Four Approaches
  - Noun Phrase Approach
  - Common Class Patterns
  - Use Case Driven
  - CRC (Class-Responsibility-Collaboration)

Noun Phrase Approach

- Examine the requirements and underline each noun
  - Each noun is a candidate class
- Divide list of candidate classes into
  - Relevant Classes
    - Part of the application domain; occur frequently in reqs
  - Irrelevant Classes
    - Outside of application domain
  - Fuzzy Classes
    - Unable to be declared relevant with confidence; require additional analysis
  - Experience will eventually enable designers to avoid generating irrelevant classes

Find Classes from requirements

- Consider Maciaszek's University Enrollment system:
  - each university major has a number of compulsory courses and a number of elective courses.



University Enrolment - Maciaszek

- More requirements:
  - A **course** can be part of any number of **majors**
  - Each **major** specifies minimum total credits required
  - Students may combine **course offerings** into **programs of study** suited to their individual needs and leading to the degree/major in which enrolled

Relevant classes	Fuzzy classes
Course	CompulsoryCourse
Major	ElectiveCourse
Student	Sudyprogram
CourseOffering	

Noun Phrase Approach

- may help in identifying domain objects
- not good at identifying objects that live in the application domain
  - Thus, it can help at the beginning of analysis, but you will not return to it as you move into design
  - Finding good objects during design means identifying abstractions that are part of your application domain and its execution machinery
  - Objects that are part of your application domain will have a tenuous connection, at best, to real-world things
    - e.g. what's the correspondence of a scrollbar to the real world

Common Class Patterns

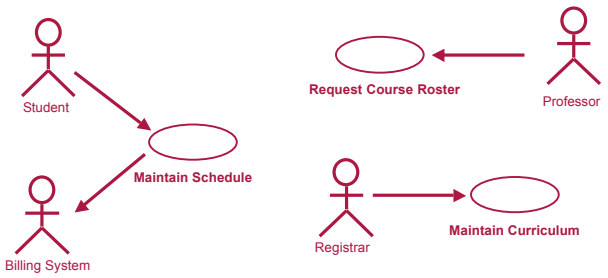
- Derive classes from the generic classification theory of objects
  - Concept class
    - a notion shared by a large community
  - Events class
    - captures an event that demarks intervals within a system
  - Organization class
    - a collection or group within the domain
  - People class
    - roles people can play
  - Places class
    - a physical location relevant to the system

Common Class Patterns

- Rumbaugh proposes a different scheme
  - Physical Class (Airplane)
  - Business Class (Reservation)
  - Logical Class (FlightTimeTable)
  - Application Class (ReservationTransaction)
  - Computer Class (Index)
  - Behavioral Class (ReservationCancellation)
- These taxonomies are meant to help a designer think of classes, however it is difficult to be systematic
- Probably only useful during early analysis

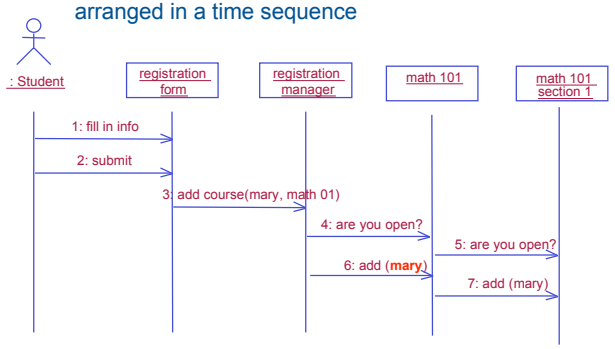
Use Case Diagram

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

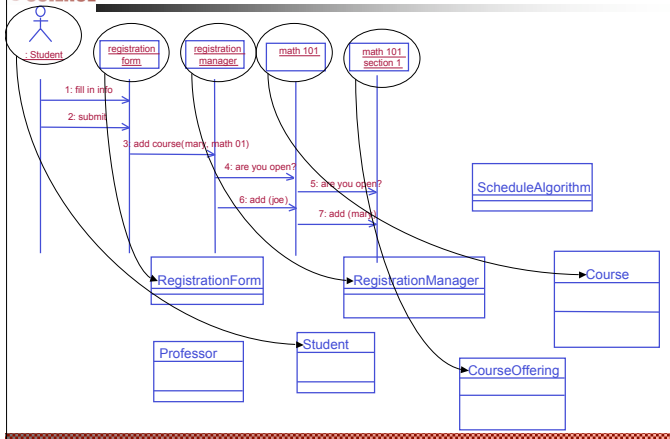


Sequence Diagram

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



Classes



Maciaszek's Guidelines

- Each class must have a statement of purpose in the system
- Each class is a template for a set of objects
  - avoid singleton classes
- Each class must house a set of attributes
- Each class should be distinguished from an attribute
  - e.g. Color may be an attribute of a Car class, but may be needed as a full class in a paint program
- Each class houses a set of operations that represents the interface of the class
  - operations can be derived from the statement of purpose

CRC Cards

- CRC = **Candidates, Responsibilities, Collaborators**
- Meant primarily as a brainstorming tool for analysis and design
- In place of use diagrams ⇒ use index cards
- In place of attributes and methods ⇒ record responsibilities
  - See **Object Design** by Wirfs-Brock and McKean, © 2003

Index Cards

- On the unlined side of the index card
  - write an informal description of each candidate class' purpose and role
- On the lined side of the index card
  - identify responsibilities and collaborators

Document
Purpose: A Document acts as a container for graphics and text
Role: Container
Pattern: Composite

Document	←candidate
Knows contents	TextFlow
Knows storage location	
Inserts and removes	
text,graphics, other elements	

responsibilities *collaborators*

Not Just Index Cards

- Post-It Notes can be used for even less "structure";
  - might be easier when brainstorming

Document
Purpose: A document Represents a container that holds text and/or graphics that the user can enter and visually arrange on pages

Why index cards?

- Forces you to be concise and clear and focus on major responsibilities since you must fit everything onto one index card
- Inherent Advantages
  - cheap, portable, readily available, and familiar
  - Affords Spatial Semantics…
  - Close collaborators can be overlapped
  - Vertical dimension can be assigned meanings
  - Abstract classes and specializations can form piles …which provides benefits
  - Beck and Cunningham report that they have seen designers talk about a new card by pointing at where it will be placed

University Enrolment - Maciaszek

- A student's choice of courses may be restricted by timetable clashes and by limitations on the number of students who can be enrolled in the current course offering.
- A student's proposed program of study is entered on on-line enrolment system SPIRE
- The system checks the program's consistency and reports any problems
- The problems need to be resolved with the help of an academic adviser
- The final program of study is subject to academic approval by the Department head (or delegate) and it is then forwarded to the Registrar
- The student's academic record to be available on demand
- The record to include information about the student's grades in each course that the student enrolled in (and has not withdrawn without penalty)
- Each course has one professor in charge of a course, but additional professors (inc instructors) may also teach in it
- There may be a different professor in charge of a course each semester
- There may be professor for each course each semester

University Enrolment - Maciaszek

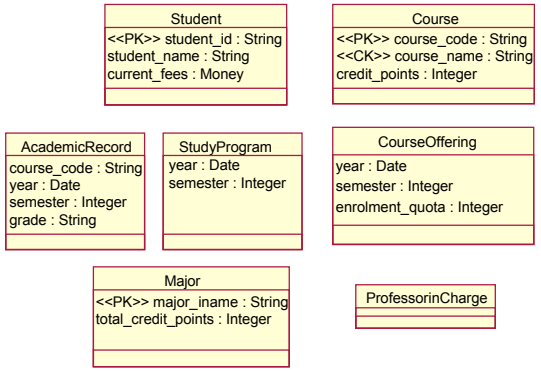
- More requirements:

Relevant classes	Fuzzy classes
Course	CompulsoryCourse
Major	ElectiveCourse
Student	Studyprogram
CourseOffering	

Specifying Attributes

- Attributes are specified in parallel with classes
  - initial set of attributes will be "obvious"
  - important to initially select attributes that help to determine the states of the class
  - additional attributes can be added in subsequent iterations

Maciaszek Solution



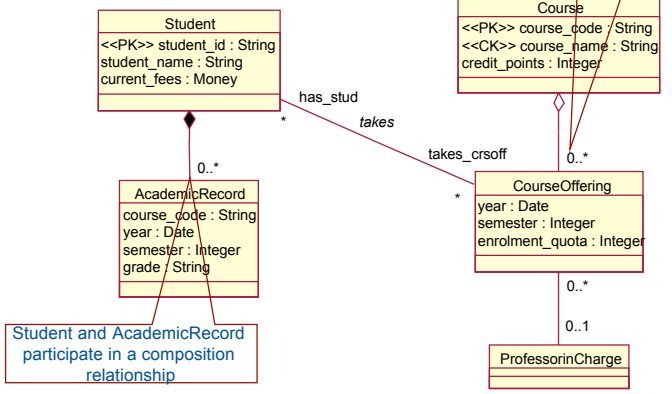
Specifying Associations

- Associations connect objects in the system
  - they facilitate collaboration between objects
- Specifying associations involves
  - naming them
  - naming the roles
    - especially useful in self associations
    - note, a role name becomes an attribute in the class on the opposite end of the association
  - determining multiplicity
- What associations might we put into the University example?

Specifying Aggregation/Composition

- "Whole-part" relationships between composite and component classes
  - UML models aggregation as a constrained form of association
- Maciaszek suggests additional power
  - ExclusiveOwns and Owns
  - Has and Member
- Litmus test: "has" or "is-part-of" is needed to explain relationship

Maciaszek Solution



Specifying Generalizations

- Looking for common features among classes
  - Move common features up a class hierarchy and specialized features down
- Apart from inheritance, generalization has two objectives
  - substitutability and polymorphism
- Litmus test: "can be" and "is-a-kind-of" required to explain relationship
- Are there any generalizations that we can make in the University example?

Behavior Specifications (I)

- Behavior of a system, as it appears to an outside user, is specified in use cases
  - During analysis, use cases specify "what" a system needs to do (not "how")
- Use cases require computations to be performed
- Computations are divided into activities
  - can be modeled using activity diagrams
- Activities are carried out by interacting objects
  - interactions are modeled using sequence diagrams

Behavior Specifications (II)

- Provide an operational view of the system
- Main Tasks
  - Define use cases and determine which classes are used to execute a use case
  - Identify operations on classes
- State specifications in analysis typically reveal entity classes
- Behavior specifications will often reveal controller classes and boundary classes (user interface classes)

Maciaszek's Take: Use Cases

- A use case represents
  - a complete piece of functionality
    - including main and alternate flows of logic
  - a piece of externally visible functionality
  - an orthogonal piece of functionality
    - use cases can share objects but execute independently from each other
  - a piece of functionality initiated by an actor
  - a piece of functionality that delivers value to an actor

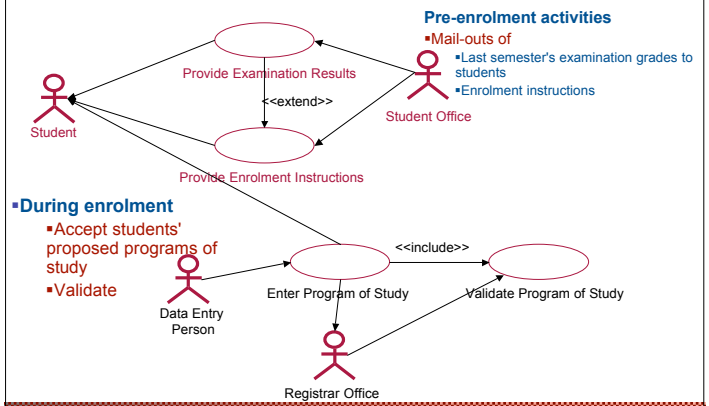
Finding Use Cases

- Use cases are discovered via analysis of
  - requirements in the requirements documents
  - actors and their purpose
- Jacobson suggests asking the following questions concerning actors to help identify use cases
  - What are the main tasks performed by the actor
  - Will an actor access or modify information in the system
  - Will an actor inform the system about changes in other systems?
  - Should an actor be informed about unexpected changes in the system?

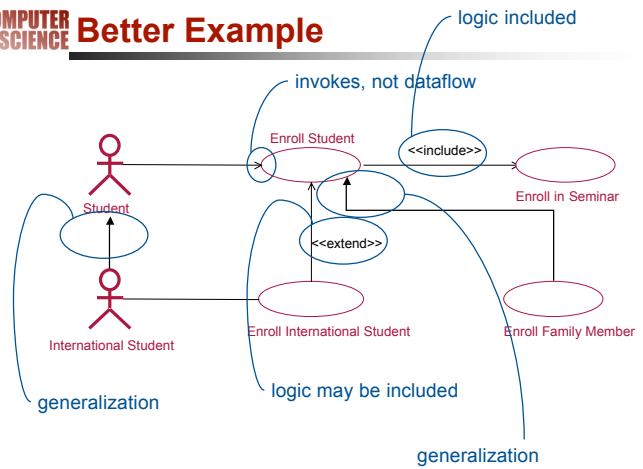
Use Case Relationships

- Association
  - a communication path
- Generalization
  - a specialized use case can change any aspect of the base use case
- Include
  - directly includes steps of another use case
- Extend
  - customize an extension point
- See (poor) example on page 137 for the University Enrollment case study
  in general, the material from Lecture 9 and 10 supercedes any use case material provided by Maciaszek
  February 20, 2003 © University of Colorado, 2003 9

Example 4.12



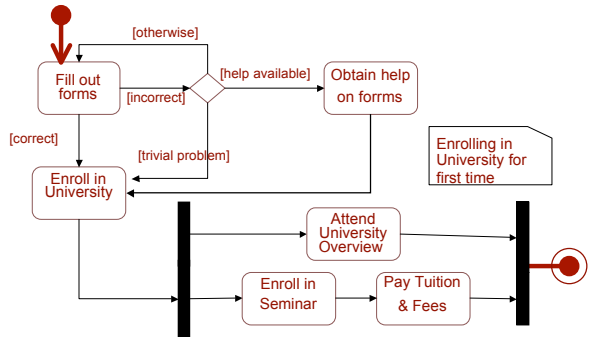
Better Example



Modeling Activities

- Activities capture the flow of logic within a system
  - both sequential and parallel control can be modeled
- Since activities do not reference classes, they can be created without the need for a class diagram
- Most often used to graphically represent the steps of a use case
  - can show main flow and extensions at once
- Activities are best discovered by analyzing the action steps of use cases, with verbs indicating candidate activities

University enrollment example



Modeling Interactions

- One level of abstraction below activities
- Interaction models require at least one iteration of state specification to be performed
  - Since we need to have classes to which each object belongs
- Interaction diagrams do not model object state changes; however they may show the actions that lead to a state change
- Interactions can help determine operations; any message to an object in a interaction must be handled by an operation (actually a method)
  - Recall that a method implements an operation; indeed there may be many methods available for a single operation

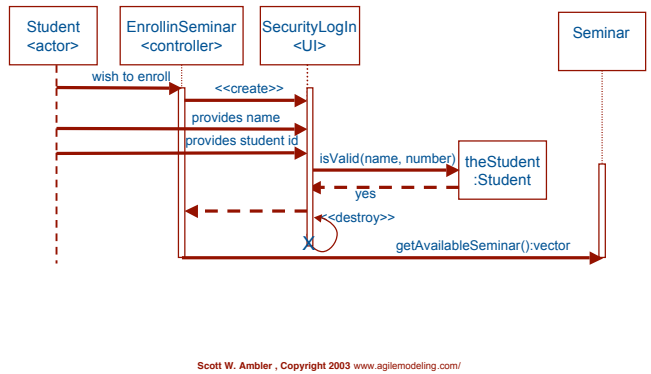
Discovering Message Sequences

- The sequence of messages in an interaction is determined by its associated activity (from the activity diagram)
  - The event that starts the activity is the first message in the interaction
  - The event that ends the activity is the last message in the interaction
  - We need to figure out what occurs in between; typically straightforward

Specifying message sequences

- Useful to distinguish between
  - signals
    - asynchronous inter-object communication
    - often shown with "half-arrow notation"
  - Calls
    - synchronous inter-object communication control returns to caller (usually)

Sequence Diagram



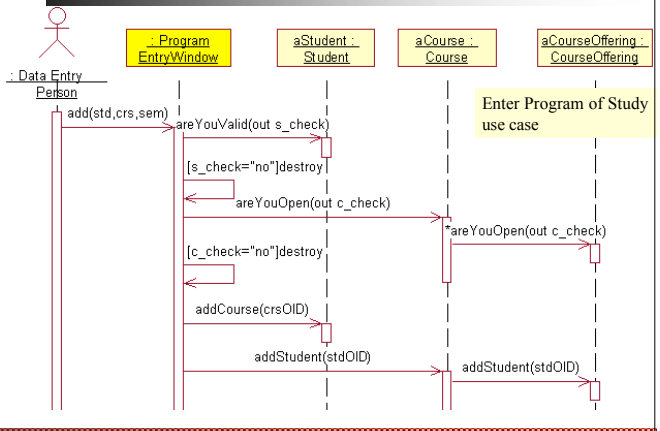
Defining Operations

- A public interface of a class consists of operations that offer services to entities external to the class
  - operations are best discovered from sequence diagrams,
    since every message must be serviced by an operation
- Other operations can be found using the CRUD (create, read, update, delete) paradigm; classes need to provide these services regardless of their domain specific functionality

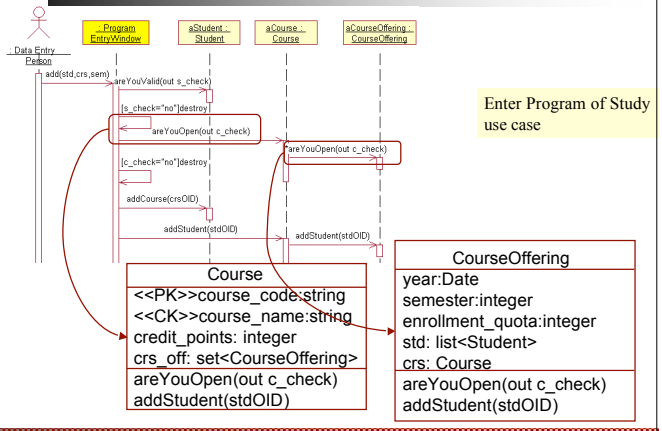
Course
<<PK>>course_code:string <<CK>>course_name:string credit_points: integer crs_off: set<CourseOffering>
areYouOpen(out c_check) addStudent(stdOID)

CourseOffering
year:Date semester:integer enrollment_quota:integer std: list<Student> crs: Course
areYouOpen(out c_check) addStudent(stdOID)

Example 4.17 – Maciaszek



Example 4.17 – Maciaszek



State Change Specifications

- Defines how an object changes state over time in response to particular events
  - States are discovered by analyzing the values of attributes and determining which have special interest to use cases
  - e.g., having or not having a phone number is a state for a customer; the specific value of the phone number is irrelevant to the state