Lecture 11: Requirements Specification

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2002

Credit where Credit is Due

- Some material presented in this lecture is taken from section 4 of Maciaszek's "Requirements Analysis and System Design". © Addison Wesley, 2000

Goals for this Lecture

- Cover the material presented in Section 4 of the textbook
  - Introduce Requirements Specification
  - Provides more insight into OO Analysis
    - This chapter provides **many** examples

Requirements Specification

- Produces three types of models
  - State Models
    - Use Cases (some actors become classes)
    - Class Diagrams
  - Behavior Models
    - Activity Diagrams
    - Interaction Diagrams
  - State Change Models
    - State Chart Diagrams

Requirements Specification

- Models are developed iteratively
  - Taking into account use cases and constraints (developed during requirements elicitation)
- Each model, or diagram, represents a view into the system; the models, taken together, allow developers and customers to view the system from multiple perspectives
- We now examine each type of model in more detail

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

- State specifications provide a static view of the system
  - The attributes and associations of classes do not change dynamically
    - in typical OO languages, some OO languages, however do allow the operations and attributes of classes to vary dynamically at run-time
- 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 app. domain
    - aka business objects
- How to do this? The 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 (already covered)
  - 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

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

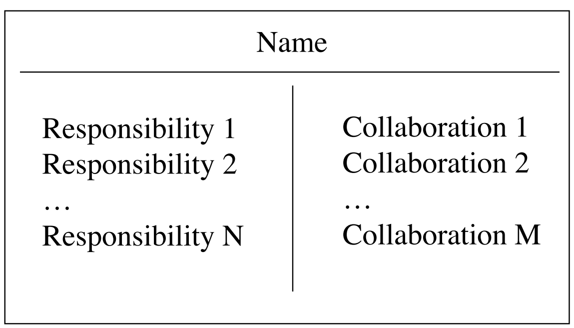
CRC Cards

- CRC Cards stands for
  – Class-Responsibility-Collaboration Cards
- Meant primarily as a brainstorming tool for analysis and design
  – Rather than use diagrams, use index cards
  – Rather than record attributes and methods, record responsibilities

Why index cards?

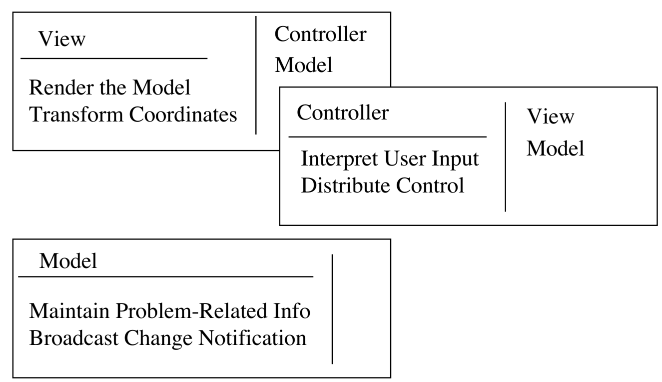
- Forces you to be concise and clear
  – and focus on major responsibilities
  – since you must fit everything onto one index card
- Inherent Advantages
  – cheap, portable, readily available, and familiar
- Affords Spatial Semantics…
  – Close collaborators can be overlapped
  – Vertical dimension can 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

Example CRC Card



Note: Collaborations are indicated by listing the names of other classes; Responsibilities are typically denoted as short English sentences

Example



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

Examples in Textbook

- Pages 112-133 work through four examples of class specification in detail
  - class discovery
  - then specifying
    - attributes
    - associations
    - aggregations/compositions
    - inheritance

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

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

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

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

Behavior Specifications

- 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
  - and can be modeled using activity diagrams;
- Activities are carried out by interacting objects;
  - interactions are modeled using sequence diagrams

More on Use Cases

- A use case represents
  - a complete piece of functionality
  - 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 reqs. doc
  - 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 examples on pages 137-140

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
- See example on page 142

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 an object state change
- Interactions can help determine operations; any message to an object in a interaction must be serviced by an operation

Discovering Message Sequences

- The sequence of messages in an interaction is determined by its associated activity
  - 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)
- See example on page 145

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

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
    - 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
  - See example on page 150