Chapter 4
Requirements Specification

© Pearson Education Limited 2005

Topics

- Architectural prerogatives
  - BCED in RASD 1ed → PCMEF in RASD 2ed
- State specifications
- Behavior specifications
- State change specifications

Architectural design

- Design
  - detailed
  - architectural
- Object dependencies → complexity and supportability
- Architectural model
  - hierarchical layers
  - restrictions on object inter-communications
  - RASD 1/e → BCED (boundary, control, entity, database)
  - RASD 2/e → PCMEF (presentation, control, mediator, entity, foundation)

PCMEF framework

- The presentation subsystem
  - classes that handle the graphical user interface (GUI) and assist in human-computer interactions.
- The control subsystem
  - classes capable to understand what program logic is looking for in entity objects
  - asking the mediator layer to bring entity objects in memory from the database.
- The entity subsystem
  - manages business objects currently in memory
  - container classes
  - containers are linked
- The mediator subsystem
  - mediates between entity and foundation subsystems to ensure that control gets access to business objects
  - manages the memory cache and synchronizes the states of business objects between memory and the database
- The foundation subsystem
  - classes that know how to talk to the database
  - produces SQL to read and modify the database

Architectural principles

- DDP – downward dependency principle
- UNP – upward notification principle
- NCP – neighbor communication principle
- APP – acquaintance package principle
- EAP – explicit association principle
- CEP – cycle elimination principle
- CNP – class naming principle
DDP, UNP, NCP

- **DDP**
  - higher PCMEF layers depend on lower layers
  - lower layers should be designed to be more stable

- **UNP**
  - upward communication that minimizes object dependencies
  - lower layers rely on interfaces and event processing (publisher/subscriber protocols) to communicate with objects in higher layers

- **NCP**
  - objects can communicate across layers only by using direct neighbors
  - chains of message passing

APP, EAP, CEP, CNP

- **APP**
  - separate layer of interfaces to support more complex object communication under strict supportability guidelines
  - subsystem of interfaces only
    - other objects in the system can use these interfaces, and may then in turn delegate access to interfaces with their own clients in other subsystems
    - downward communication, but only above the APP-layer
    - APP is decoupled from the CMEF layers (and, therefore, without creating dependencies on CMEF layers)

- **EAP**
  - legitimizes run-time object communication in compile-time data structures

- **CEP**
  - cyclic dependencies, between classes in the same subsystem, or between classes in different subsystems
  - avoidable, but can be neutralized
    - extra classes to reduce a network of calls to a hierarchy
    - purposeful use of interfaces

- **CNP**
  - name of each class and each interface in the system should identify the subsystem/package layer to which it belongs
  - ensuring that each class begins with a single letter identifying the PCMEF layer (i.e. P, C, etc.)
    - EVideo means that the class is in the entity subsystem
    - IMVideo means that the interface is in the mediator subsystem

State specifications

- **Object state** is determined by the values of its attributes and associations

- **State specification**:
  - Model of data structures
  - Static view on the system
  - Class operations left out in initial specs
  - Emphasis on entity classes ("business objects")

Modeling classes

- **Cornerstone of OO development** – a system is a set of collaborating (and classified) objects

- **Iterative and incremental process**

- **CASE tool**
  - For collaborative development
  - For personal productivity otherwise

Discovering classes

- **No two analysts will come up with the identical class models for the same application domain**

- **Discovering classes** (details in the textbook)
  - Noun phrase
  - Common class patterns
  - Use case driven
  - CRC
  - Mixed

Guidelines for class discovery

- **Statement of purpose**

- **Description for a set of objects**
  - Singleton classes
  - Houses a set of attributes
    - Identifying attributes - keys
    - OID
  - Class or attribute?
  - Houses a set of operations (what does the class do?)
Example 4.1 – University Enrolment

Consider the following requirements for the University Enrolment system and identify the candidate classes:

- Each university degree has a number of compulsory courses and a number of elective courses.

<table>
<thead>
<tr>
<th>Degree</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant</td>
<td>Fuzzy</td>
</tr>
<tr>
<td>CompulsoryCourse</td>
<td>ElectiveCourse</td>
</tr>
</tbody>
</table>

More requirements:

- Each course is at a given level and has a credit-point value
- A course can be part of any number of degrees
- Each degree specifies minimum total credit points value required for degree completion
- Students may combine course offerings into programs of study suited to their individual needs and leading to the degree in which they enrolled

Example 4.1 – University Enrolment (solution)

<table>
<thead>
<tr>
<th>Relevant classes</th>
<th>Fuzzy classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>CompulsoryCourse</td>
</tr>
<tr>
<td>Degree</td>
<td>ElectiveCourse</td>
</tr>
<tr>
<td>Student</td>
<td>Studyprogram</td>
</tr>
<tr>
<td>CourseOffering</td>
<td></td>
</tr>
</tbody>
</table>

Specifying classes

- In Class Diagram
  - Each class given a name (and possibly a code)
  - Singular noun
    - Recommendation: multiple words joined; each word starting with a capital letter (e.g. PostalAddress)
  - Meaningful
  - Short (less than 30 characters)
- Class properties to be defined
  - Attributes (initially those that capture interesting object states)
    - Recommendations:
      - small letters; underscore to separate words (e.g. street_name)
      - start with a small letter; capitalize successive words (streetName)
  - Operations (can be delayed till later analysis stages or even till design)

Example 4.4 – University Enrolment

Refer to Example 4.1

Consider the following additional requirements from the Requirements Document:

- 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.
Example 4.4 – University Enrolment (solution)

- EDegree
  - <PK> degreeName : String
  - totalCreditPoints : int

- ECourse
  - <PK> courseName : String
  - creditPoints : short

- ECourseOffering
  - year : short
  - semester : short
  - enrolmentQuota : int

- EStudyProgram
  - year : short
  - semester : short

- EStudent
  - <PK> studentId : String
  - studentName : String

Discovering associations

- Side effect of discovering classes
- Some attributes are associations
- “Dry-run” of use cases to discover more associations
- Avoid ternary associations
- Cycles of associations that do not commute

Specifying associations

- Naming associations
  - Recommendation – small letters; capitalizing the first letters of successive words (e.g. empTask)
- Naming association roles
- Determining multiplicity
  - Lower and/or upper multiplicity bounds can be omitted initially
- Rolenames for recursive associations

Modeling aggregation and composition

- Four semantics for aggregation possible
  - ExclusiveOwns (e.g. Book has Chapter)
    - Existence-dependency
    - Transitivity
    - Asymmetricity
    - Fixed property
  - Owns (e.g. Car has Tire)
    - No fixed property
  - Has (e.g. Division has Department)
    - No existence dependency
    - No fixed property
  - Member (e.g. Meeting has Chairperson)
    - No special properties except membership

Discovering aggregation

- Discovered in parallel with discovery of associations
- The litmus test phrases
  - “has”
  - “is-part-of”
- Can relate more than two classes
Specifying aggregation

- **UML supports**
  - Aggregation
    - By-reference semantics
    - Hollow diamond
    - Corresponds to Has and Member aggregations
  - Composition
    - By-value semantics
    - Solid diamond
    - Corresponds to ExclusiveOwns and Owns aggregations

Example 4.9 – University Enrolment (solution)

Modeling generalization

- **Common features abstracted into a more generic class**
- **Subclasses inherit (reuse) superclass features**
- **Substitutability** – subclass object is a legal value for a superclass variable (e.g., a variable holding Fruit objects can have an Apple object as its value)
- **Polymorphism** – the same operation can have different implementations in different classes
- **Abstract operation** – implementation provided in subclasses
- **Abstract class** – class with no direct instance objects
  - A class with an abstract operation is abstract

Discovering and specifying generalization

- **Some discovered in parallel with discovery of associations**
- **The litmus test phrases**
  - "can-be"
  - "is-a-kind-of"
- **Multiple inheritance possible**
- **Solid line with an arrowhead pointing to the superclass**

Example 4.10 – Video Store

Modeling interfaces

- **Interfaces**
  - do not have attributes (except constants), associations or states
  - they only have operations, but all operations are implicitly public and abstract
  - parameters are declared (i.e., turned into implemented methods) in classes which implement these interfaces.
- **Interfaces do not have associations to classes but they may be targets of one-way associations from classes**
  - this happens when an attribute that implements an association is typed with an interface, rather than with a class
  - the value of any such attribute will be a reference to some class that implements the interface
- **An interface may have a generalization relationship to another interface**
  - this means that an interface can extend another interface by inheriting its operations
Discovering and specifying interfaces

- Interfaces are not discovered from the analysis of the application domain.
- They are discovered based on design considerations:
  - Fundamental for enforcing architectural frameworks, such as the PCMEF framework.
  - Interface reveals only a limited portion of the behavior of an actual class.
- Class that uses (requires) the interface can be indicated by a dashed arrow pointing to the interface:
  - The arrow can be stereotyped with the keyword «use».
- Class that implements (realizes) the interface is indicated by a dashed lined with a triangular end:
  - The line can be stereotyped with the keyword «implement».

Example 4-11 – Contact Management

- Consider classes EContact and EEmployee:
  - Some attributes in common (firstName, familyName).
  - Operations that provide access to these attributes can be extracted into a single interface.
- There is a need to display information about overdue events to the screen:
  - Presentation-layer class has the responsibility to display a list of overdue events together with names of contacts and employees, as well as with the additional contact details (phone and email) to contacts.
- Propose a model such that the presentation class uses one or more interfaces implemented by EEmployee and EContact to support part of the “display overdue events” functionality.

Example 4.12 – University Enrolment

- Only to exemplify:
  - To illustrate complex relationships between objects.
  - To demonstrate changes to objects over time.
  - To illustrate object collaboration.

Behavior specification

- Depicted in use cases.
- Determines which classes are involved in execution of use cases:
  - Main class operations identified.
  - Message passing between objects captured.
  - Control classes and boundary classes considered.
- Computations modeled in Activity Diagrams.
- Interactions modeled in Sequence Diagrams or Collaboration Diagrams.
### Modeling use cases

- Complete piece of functionality
  - Main flow
  - Subflows
  - Alternate flows
- Piece of externally visible functionality
- Orthogonal piece of functionality
- Piece of functionality initiated by an actor
- Piece of functionality that delivers an identifiable value to an actor

### Discovering use cases

- Discovered from
  - Requirements identified in the Requirements Document
  - Actors and their purpose in the system
- Questions to ask
  - What are the main tasks performed by each actor?
  - Will an actor access or modify information in the system?
  - Will an actor inform the system about any changes in other systems?
  - Should an actor be informed about unexpected changes in the system?

### Specifying use cases

- Actors
- Use cases
- Four kinds of relationships
  - Association (between actor and use case)
  - Include (stereotyped with the word: «include»)
    - Included use case is always necessary for the completion of the activating use case
  - Extend (stereotyped with the word: «extend»)
    - Another use is activated occasionally at specific extension point
  - Generalization
- Relationships to be used with restrain

### Example 4.13 – University Enrolment

- Activity Diagrams
  - Flow of logic
    - Sequential control
    - Concurrent control
  - Can be used at different levels of abstraction
    - To define execution of a use case
    - To define execution of an operation

### Discovering and specifying actions

- The execution proceeds from one action state to the next
- An action state completes when its computation is completed
- Actions can be discovered from the narrative specifications of use cases
- Actions are connected by transition lines
- Synchronization bars (fork and re-join)
- Branch diamonds (branch and merge)
- External events not normally modeled on activity graphs
Example 4.17 – Video Store

Modeling interactions

- **Sequence Diagrams**
  - Show an exchange of messages between objects arranged in a time sequence
  - More useful in analysis

- **Collaboration Diagrams**
  - Emphasize the relationships between objects along which the messages are exchanged
  - More useful in design

- Can be used to determine operations in classes

Message sequences

- Actions in Activity Diagrams are mapped to messages to Sequence Diagrams
- Message can be a:
  - **Signal**
    - Denotes asynchronous inter-object communication
    - The sender continues executing after sending the signal message
  - **Call**
    - Denotes synchronous invocation of an operation
    - The return message can return some values to the caller or it can just acknowledge that the operation completed

Example 4.18 – UE (centralized interaction)

Example 4.18 – UE (distributed interaction)

Modeling public interfaces

- Determined by the set of operations that the class offers as its service
- In analysis
  - Signature of each operation is defined
    - Operation name
    - List of formal arguments
    - Return type
- In design
  - Algorithm of a method that implements the operation is defined
- Operation can have
  - Instance scope
  - Class (static) scope ($^2$ in front of operation name)
Discovering class operations

- From Sequence Diagrams
  - Message to an object must be serviced by an operation in that object
- From expected object responsibilities, including the CRUD operations
  - Create – a new object instance
  - Read – the state of an object
  - Update – the state of an object
  - Delete – i.e. destroy itself

Example 4.19 – UE (centralized interaction)

Example 4.19 – UE (distributed interaction)

State change specifications

- Statechart Diagrams
- For each class that exhibits an interesting dynamic behavior
- Changes to some attributes signify state changes

Specifying object states

- State transition fires when a certain event occurs or a certain condition is satisfied
  - transition line does not have to be labeled with an event name
  - condition itself (written in square brackets) can fire the transition
- Transition can be triggered by
  - Signal event
  - Call event
  - Change event
  - Time event

Example 4.19 – Video Store
Summary

- The critical importance of **architecture** in system development (BCED → PCMEF)
- **State specifications** describe the IS world from the static perspective of classes, their attribute content and their relationships
  - There are many methods of class discovery
  - Class diagrams visualize classes and relationships: associations, aggregations and generalizations
- **Behavioral specifications** describe the IS world from the operational (functional) perspective
  - Use case diagrams provide simple visualization – each use case is given narrative specification
  - Other behavioral diagrams include activity diagrams, interactions diagrams, and addition of operations to classes.
- **State change specifications** describe the IS world from the dynamic perspective
  - Statechart diagrams allow modeling of state changes