Object Oriented Design

Chapter 06

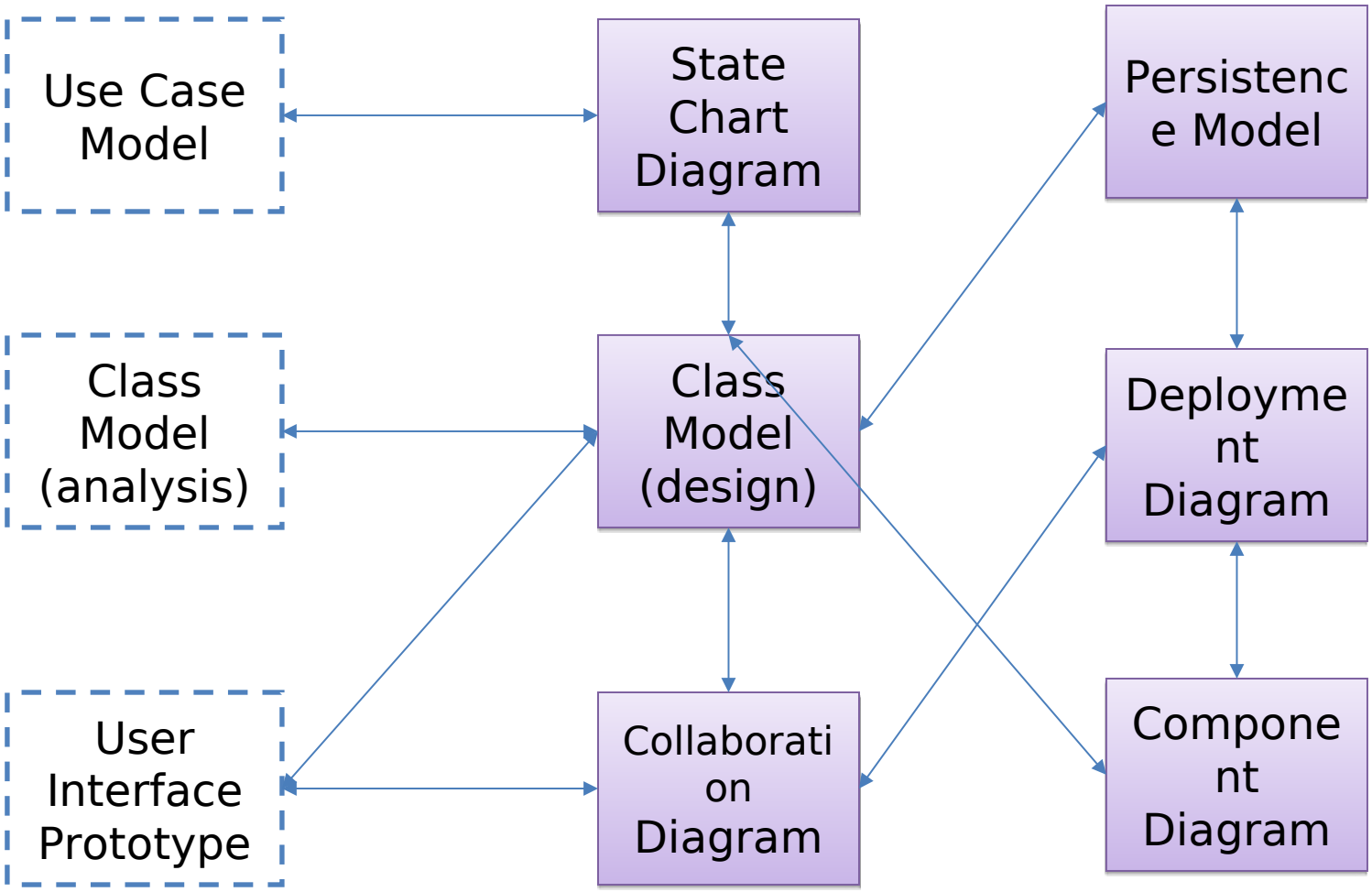
Outline

- evolve your analysis model into a design model
- layer the architecture of your system
- develop and design class model
- apply design patterns effectively
- develop state chart diagrams
- develop collaboration diagrams
- develop a component based design
- develop a deployment model
- evolve your UI prototype
- take advantage of common object design tips

Evolve into design

- Purpose of design is to determine how you are going to build your system
- Analysis and design are highly interrelated and iterative
- Your analysis class model evolves into your design class model
- Decide on high level issues
  - Do you intent to take a pure object oriented approach Vs Component based approach
    - Pure OO – s/w is built from collection of classes
    - Component based – from components (discussed later)
  - Will you follow a common business architecture
  - Will you follow a common technical infrastructure

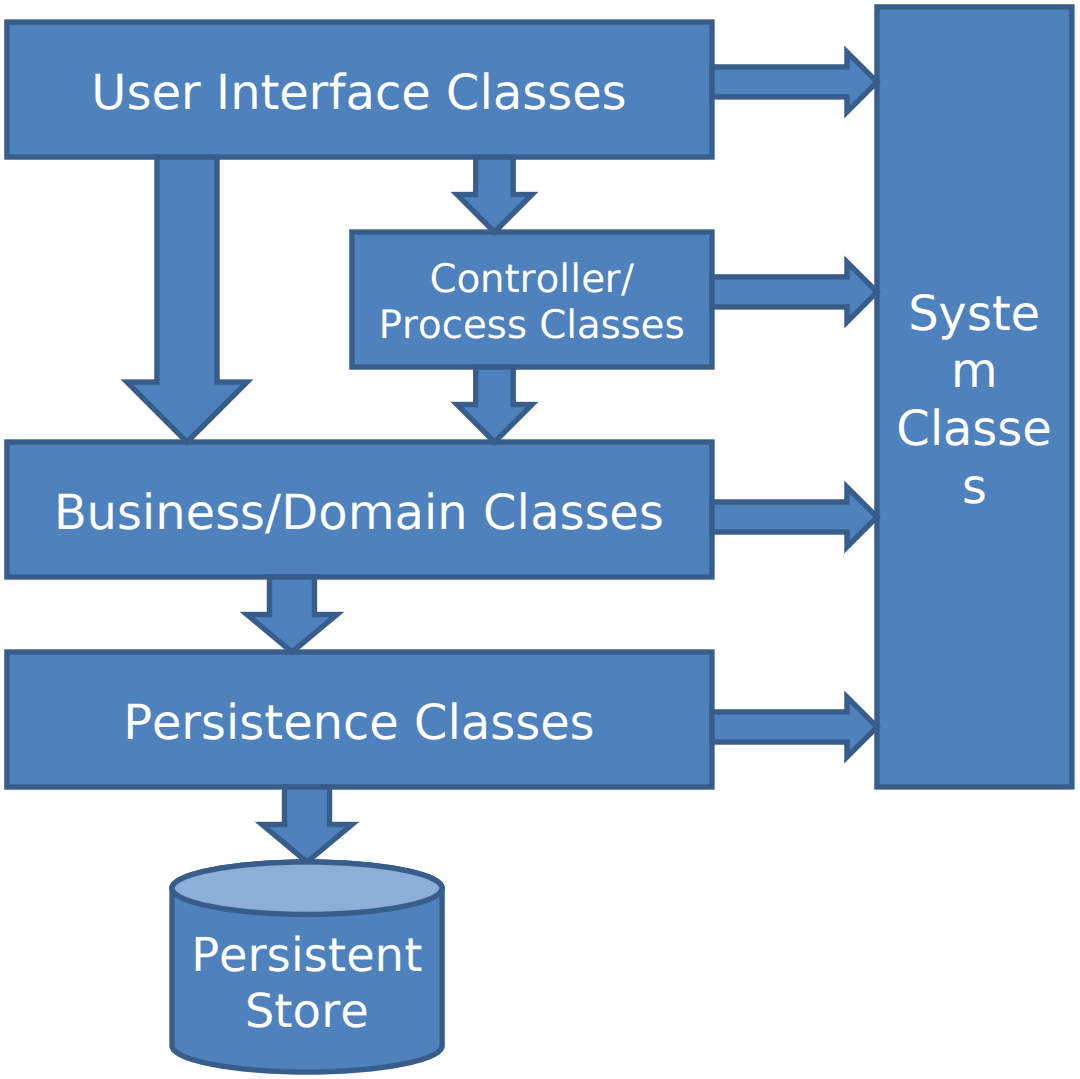
Design Artifacts and their relationships



Layering your models

- Organizing your software design into collections of classes or components that fulfill a common purpose
- Qualities of a good layer
  - You should be able to make modifications to any given layer without affecting any other layers
  - Layers should be modularized
    - Rewriting layer or replacing
- Collaboration between classes is allowed within a layer
- By restricting the flow of messages to only one direction, you dramatically increase the portability of your system by reducing the coupling between class
- All types of classes may interact with system classes
  - Fundamental software features such as inter-process communication

Layering your Models



Layers

- User Interface classes – implements the major UI element of your system
- Business behaviour classes
  - Business/domain classes
    - Implements the concepts pertinent to your business domain such as “student” or “seminar”
  - Controller/ process classes
    - Implements business logic that involves collaborating with several business domain classes
- Persistence Classes – encapsulate the capability to store, retrieve and delete object permanently without revealing details of the underlying storage technology
- System Classes – provide operating system specific functionality for your applications, isolating your software from the operating system

User Interface Layer

- Contains the code for the user interface part of an application
  - Choice of GUI, menu, editing screen, and report classes
- University System
- UI for any given system can take on many possible forms even though the underlying business is still the same .
- UI classes are often identified as part of your UI prototyping efforts, as well as sequence modeling
- Often referred to as interface classes or boundary classes

The Controller Process Layer

- Purpose is to implement business logic that pertains to several objects, particularly objects that are instances of different classes
- Stereotype <<controller class>>
- Rework of the sequence diagram
  - Refactoring of the controller class
    - to interact only with business classes
    - Introduction of a new user interface class representing the main menu
    - Controller class “EnrollinSeminar”, now only manages interactions between business classes

Business Domain Layer

- Also called an analysis or entity class
- Subject Matter Experts (SMEs) are often the people who identify these classes
- The business layer enables you to encapsulate the basic business functionality without having to concern yourself with user interface, data management, or system management issues

Persistence Layer

- Provides the infrastructure for the storage and retrieval of objects
  - Helps to isolate your application from changes to your permanent storage approach
- The persistence layer, by encapsulating data management functionality, increases the maintainability, extensibility and portability of your application
- Message from business class layers to the persistence class layer
  - “create a new object”, “retrieve this object from the database”, “update this object” or “delete this object”
- Persistent layer provides access to permanent storage
- Goal – reduce the maintenance effort that is required whenever changes are made to your database
- The persistence layer isolates you from the impact of changes to your storage strategy

System Layer

- The system layer provides access to the operating system and non OO resource
- Differences between operating systems can make it tough if you are writing an application that needs to work on many different platforms
- Wrap features of an operating system- Port an application - You need to create classes that wrap specific features of the operating system
- System classes encapsulate non OO functionality by wrapping it with OO code

Class Modeling

- The design class model will reflect the wide variety of technology decisions you make
- Focus is on the solution domain not on the problem domain
- Introduce change to your class model based on implementation technologies
- Implement business rules using business rules engine – your business class will invoke the rule, instead of directly implementing them in methods

Inheritance Techniques

- The sentence rule works 99.9 percent of the time
- Beware of implementation inheritance
- Any kind of class can inherit from any other kind
- You should be able to substitute an instance of a subclass or an instance of a super class
- Beware of multiple inheritance
- Beware of inheritance based on common data attributes
- Super classes should know nothing of their subclasses
- Factor commonality s high as possible in your class hierarchy
- A subclass should inherit everything

Association and dependency

- Model the scaffolding for your associations
- Multiplicity must be shown
- Question multiplicities involving minimums and maximums
- Associations and dependencies are inherited
- Collaboration goes hand in hand with relationships
- Model a unidirectional association when collaboration
- Model a dependency when one of the instances is transient
- Model a dependecy when an object interacts only with a class, but not the instance
- Do not model implied association
- The Liskov substitution principle applies to mirror hierarchies

Aggregation and Composition

- The advice for association applies to aggregation and composition
- The sentence rule should make sense for aggregation and composition
- You should be interested in both the whole and the part
- You need to understand how the whole and the parts collaborate with each other
- The majority of the interaction is from the whole to the part
- Don’t confuse inheritance with aggregation

Modeling Methods During Design

- On design class diagrams, indicate the visibility, name, parameters, return value, and stereotype of methods
- Names of parameters, as well as their types and default values should be indicated for each methods
- Scope of a method, weather it is a static method that works on the class or an instance method, that works on instances of the class
  - Static Methods are underlined, instance methods are not

The Student Class with its methods fully modeled

Student
name phoneNumber emailAddress studentNumber averageMark
+isEligible(name:string, studentNumber:StudentNumber):boolean +Student(studentNumber: StudentNumber): Student <<constructor>> + getCoursesTaken (): Vector + purchaseParkingPass() +getAverageMark(): long - setAverageMark (newAverageMark:long)

The UML format for an operation signature

```
Visibility      name(param1:type1 = default1, ...): return Type
<<stereotype>>
```

Naming Method

- Use a full description, using mixed case with the first letter of any non initial word capitalized
- “methodName()”
- First word – strong action verb

Example Names for member function		
“Bad” Name	“Good “ Name	Issue
openAcc()	openAccount()	An abbreviation was replaced with the full word to make the meaning clear
mailingLabelPrint()	printMailingLabel()	The verb was moved to the beginning of the name to make it active
Purchaseparkingpass()	purchaseParkingPass()	Mixed case was applied to increase the readability of the name
saveTheObject()	Save()	The name was shortened because the term “the Object” did not add any value

Method Visibility

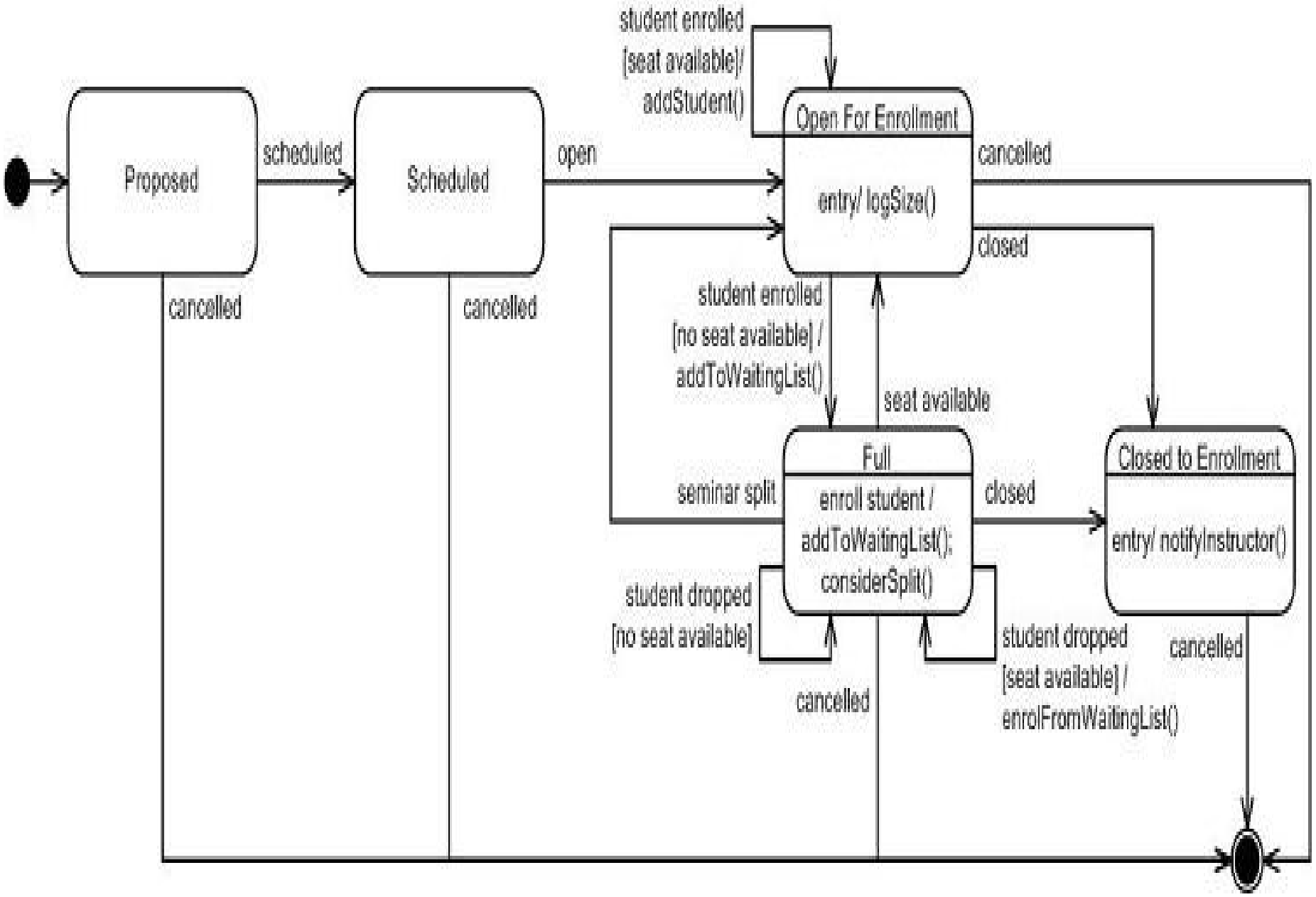
- How a method is accessed by objects
- Three levels of visibility
  - Public, Protected and Private
- To reduce coupling within your system, the general rule of thumb is to be as restrictive as possible when setting the visibility of a method
  - A method doesn’t have to be public, then make it protected, if it doesn’t have to be protected make it private

UML method visibilities

Visibility	Symbol	Description	Project Usage
Public	+	A public method can be invoked by any other method in any other object or class	When the method must be accessible by objects and classes outside of the class hierarchy in which the method is defined
Protected	#	A protected method can be invoked by any method in the class in which it is defined or any subclasses of that class	When the method provides behaviour that is needed internally within the class hierarchy, but not externally
Private	-	A private method can only be invoked by other methods in the class in which it is defined, but not in the subclass	When the method provides behaviour that is specific to the class. Private methods are often the result of refactoring

State Chart Modeling

- Objects know things and do things
- Some objects are incredibly more complicated
- The understand complex classes better, particularly those that act in different manners depending on their state, you should develop one or more UML state chart diagrams
- Depict the various states that an object may be in and the transitions between those states
- State – stage in the behaviour pattern of an object
  - Initial and final state
    - Initial – creation state is the one that an object is in when it is first created
    - Transition – progression from one state to another and will be triggered by an even that is either internal or external to the object


Proposed
scheduled
Scheduled
open
Open For Enrollment
entry/ logSize()
student enrolled
[seat available]/
addStudent()
cancelled
closed
cancelled
cancelled
student enrolled
[no seat available] /
addToWaitingList()
seat available
Full
enroll student /
addToWaitingList();
considerSplit()
seminar split
closed
Closed to Enrollment
entry/ notifyInstructor()
student dropped
[no seat available]
cancelled
student dropped
[seat available] /
enrolFromWaitingList()
cancelled

How to draw state diagrams

- Identity the initial/creation state
- Identify the final states
- Identify as many other application, “Real world” states as possible
- Identify potential sub states
- Identify the transition leaving a state
- Identify the target state to which a transition leads

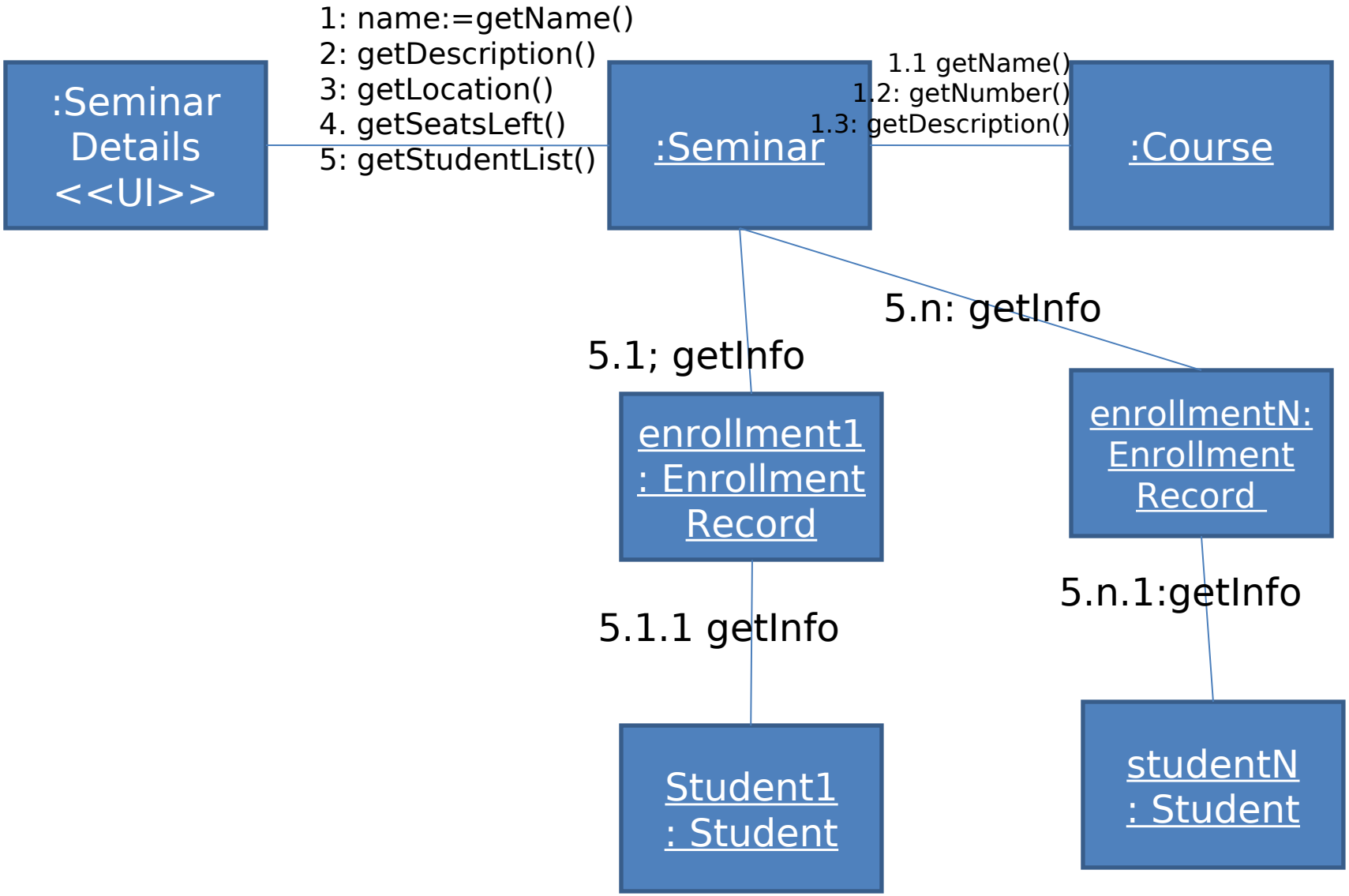
When and how state diagrams be used

- It is a dynamic modeling technique one that focuses on identifying the behaviour within you system

Collaboration Diagrams

- Depict a birds eye view of the interactions between objects
- Shows the message flow between objects in an OO application and also imply the basic associations between classes
- Rectangle – various objects involved that make up the application
- Lines – represent the relationships (associations, aggregation, composition, dependencies)
- Optionally you may indicate the sequence number in which the message

A collaboration diagram for a simple university

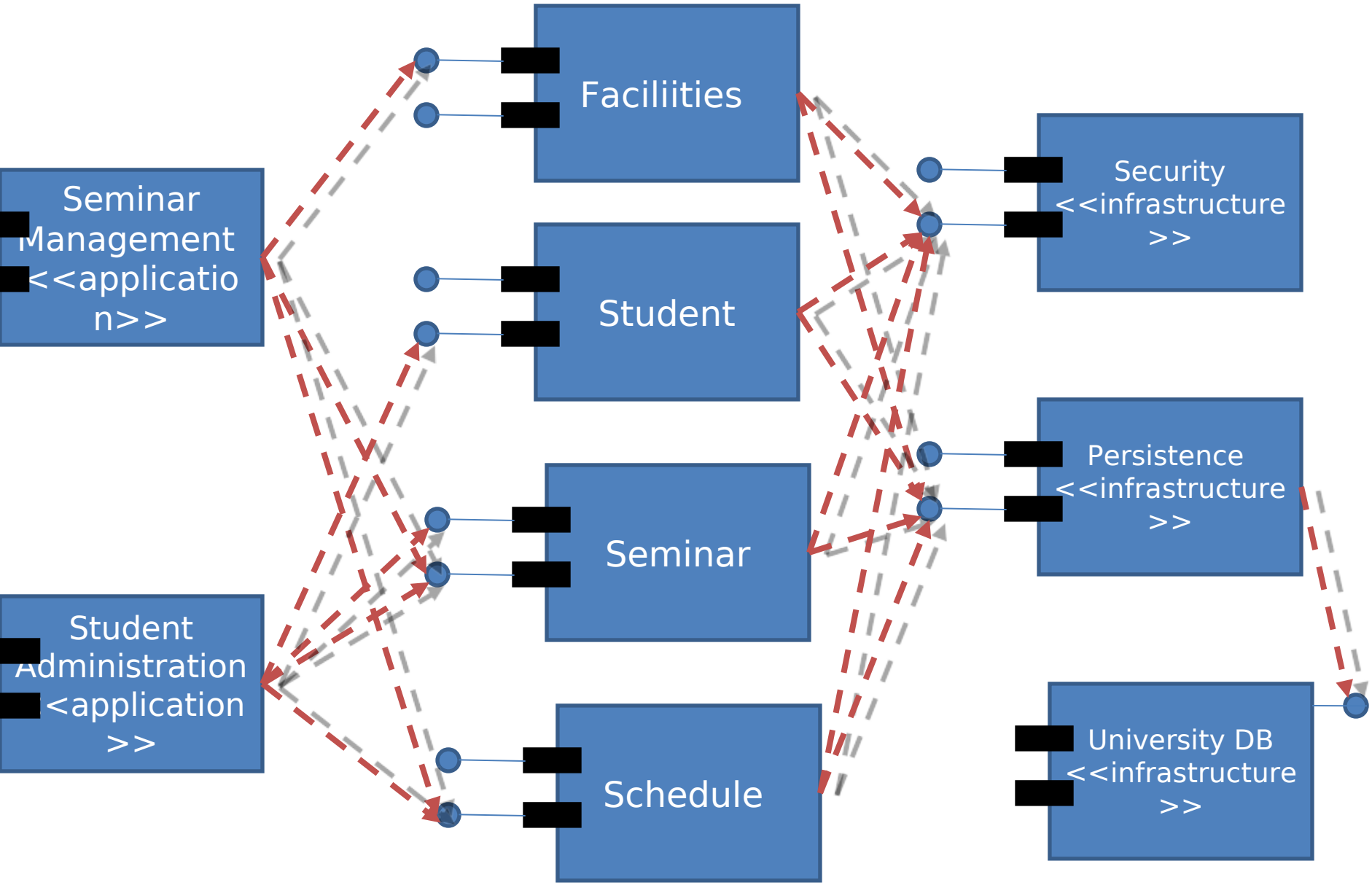


Drawing Collaboration Diagram

- With collaboration diagramming the basic idea is that you identify
  - The scope of the diagram
  - The objects
  - The relationships between the objects
  - The messages passed between the objects
- Draw whenever you want to fully understand the behavior of an OO application and to show the objects that makeup the application and the message flow between them

Component Modeling

- OO is a preferred foundation from which to build components
- UML includes a component diagrams that can be used to analyze and design your component based software
- Components are modeled as rectangles with two smaller rectangles jutting out from the left hand side
- Components have dependencies on the interfaces of other components, modeled using the standard UML dependency notation


Seminar Management <<application>>
Student Administration <<application>>
Faciliities
Student
Seminar
Schedule
Security <<infrastructure>>
Persistence <<infrastructure>>
University DB <<infrastructure>>

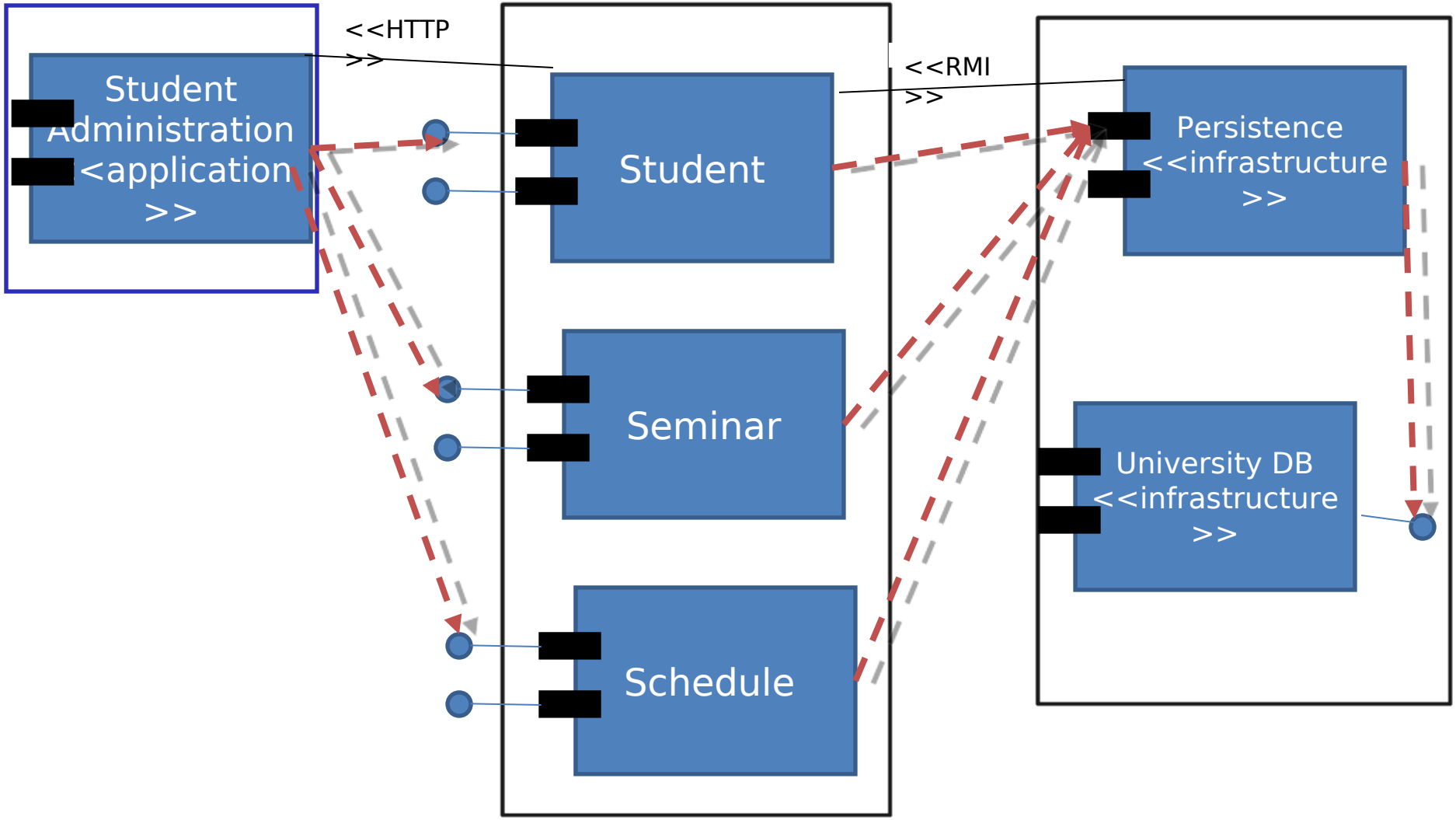
How to develop a component model

- Five steps exist
  1. Handle non-business/domain classes
  2. Define class contracts
  3. Simplify inheritance and aggregation hierarchies
  4. Identify domain components
  5. Define domain component contracts

How to develop Deployment model

1. Identify the scope of the model :typically a deployment configuration for a single application
2. Identify the distribution architecture :will your application will have two tiers,three btiers or more
3. Identify the node and their connection:
4. Distribute components to nodes

N.B: the following takes three tiers client/server approach : telling you that you have UI tier, an application server tier and database server tier


Student Administration <application >>
<<HTTP >>
Student
Seminar
Schedule
<<RMI >>
Persistence <<infrastructure >>
University DB <<infrastructure >>

Relational Persistence Modeling

- Useful if you are using relational database
- Used as a mechanism to make your objects persistent
- Different from the design of a class diagram
- Persistence models are often called data models or entity relationship (ER) models – are used to communicate the design of a database

Keys and object identifiers

- A key uniquely identifies a row in a table
- Primary key – preferred key for a data entity
- Secondary key – alternate way to access rows within a table
- Foreign key – used to maintain relationships between rows
- Use stereotypes - <<primary key>>, <<foreign key>>
- OIDs enable you to simplify your key strategy within a RDB
- Enables you to automate the maintenance of relationships between objects.
- OID should be unique within a class hierarchy

Mapping Objects to RDBs

- Mapping Attributes to Columns
  - Attributes map to zero or more columns; zero or more attributes map to a single column
- Mapping Classes to tables
  - Classes map to one or more tables; one or more classes can map to a single table
- Implementing Inheritance in a Relational Database
  - 3 basic choices when mapping inheritance
    - Use one data entity for an entire class hierarchy
    - Use one data entity per concrete class
    - Use one data entity per class

Mapping Association, Aggregation and Composition

- Difference between association and aggregation/composition
  - The tightness the objects are bound to each other
  - Aggregation/Composition - Anything you do to the whole in the database, you almost always need to do to the parts, not the case with association
- Implementing associations in RDB
  - Maintained through the use of foreign keys
- Implementing Many to Many Associations
  - Use associative table
-

Mapping Association, Aggregation and Composition

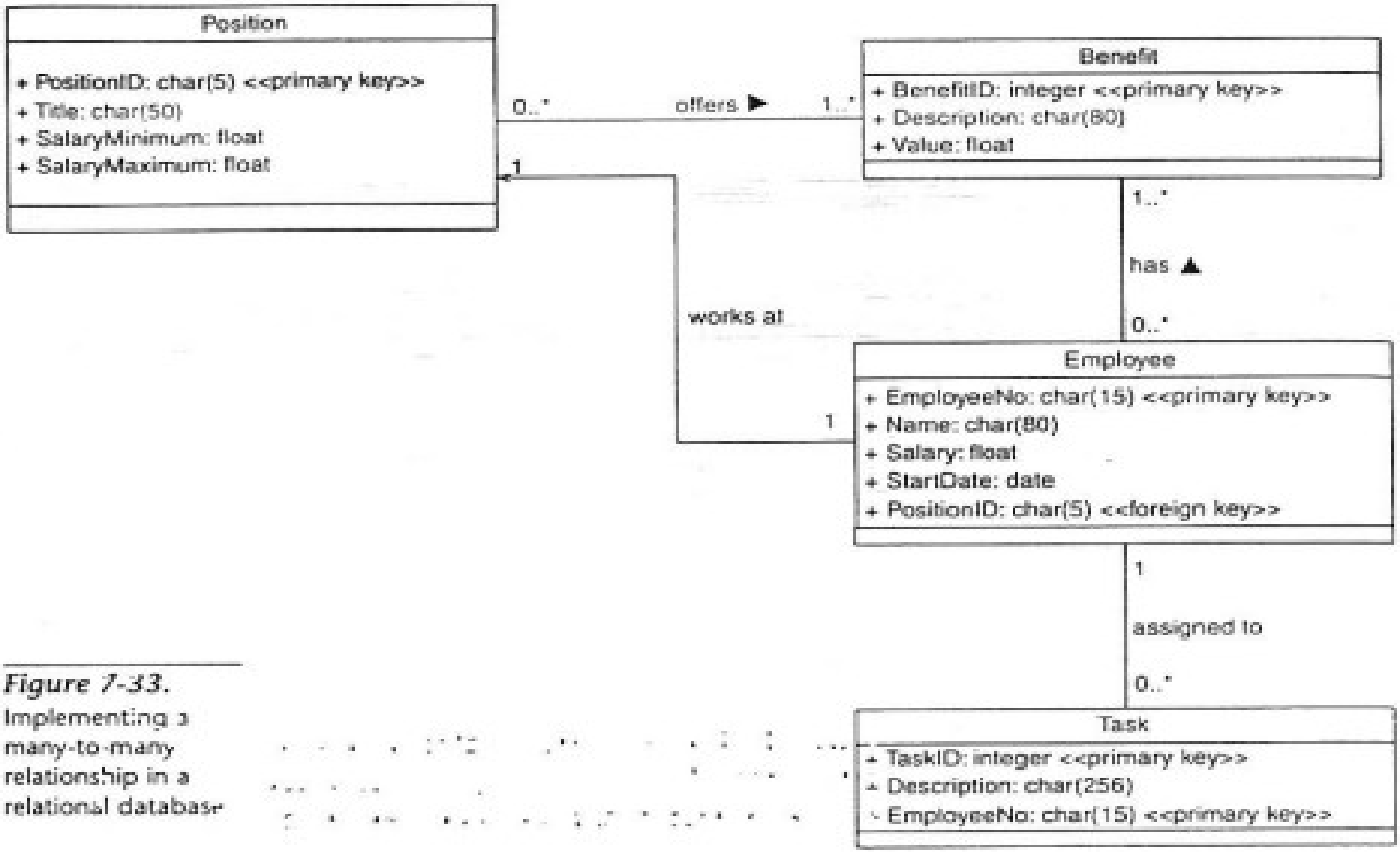


Figure 7-33.
Implementing a many-to-many relationship in a relational database

User Interface Design

- Clean up effort - Focuses on applying common user interface design principles and techniques
- Complex task, on that requires a wide range of skills to be successful
- Developers need to have an understanding of the basics of user interface design

UID principles

- The structure principle
- The simplicity principle
- The visibility principle
- The feedback principle
- The tolerance principle
- The reuse principle

Techniques for improving your UID

- Consistency, Consistency, Consistency
- Set standards and stick to them
- Explain the rules
- Support both novices and experts
- Navigation between major user interface items is important
- Navigation within a screen is important
- Word your messages and labels appropriately
- Understand your widgets
- Look at other application with a grain of salt

Contd...

- Use color appropriately
- Follow the contrast rule
- Align field effectively
- Expect your users to make mistakes
- Justify data appropriately
- Your design should be intuitable
- Don't create busy user interfaces
- Group things effectively

User interface flow diagramming

- Evolve your user interface flow diagram to reflect the design information being captured in your user interface prototype and your sequence diagrams
- User interface implementation choices are now reflected in the UI flow diagram.
  - GUI - <<window>> and <<dialogue box>>
  - Web based - <<HTML page>>
- The types of transitions between the various UI elements are also shown

User Interface Design Standards and Guideline

- Adopt and industry standard and modify it as needed
- Follow and enforce the guidelines
- Obtain training in user interface design and in the guidelines

Design Tips

- Focus on the problem not the technique
- Don't forget your technical infrastructure
- Document your style guideline
- Develop a technical prototype
- Requirements then analysis then design and then code
- Use Computer Aided System Engineering (CASE) tools effectively
- Document complicated things
- Do not over document
- Design for change
- Design for your implementation environment judiciously
- Expect and act on feedback
- Indicate application of patterns