Chapter 6

Underpinnings of System Design

Topics

- Software Architecture – Architectural Design
  - Distributed architecture
  - Three-tier architecture
  - Programming databases
  - Reuse strategy
  - Component
  - Deployment
- Collaboration – Detailed Design
  - Collaboration diagram
  - Realization of use case
  - Realization of operation

Distributed architecture

- Distributed processing versus distributed database

Communication network

C/S

C/S
Three-tier architecture

- Thick versus thin client architecture

Novell NetWare
- SPX/IPX
- X terminal

* Unix
- Database

Windows NT
- Business Rules
- Server

Corporate Database
- Data Warehouse

User Layer

Active Database Logic

Programming databases

- Active database
  - Stored procedure
  - Trigger

- Application-database interaction
  - User interface
  - Presentation logic
  - Application function
  - Integrity logic
  - Data access

Application-database interaction

User Interface
- Presentation Logic
- Application Function (Development)

Middleware - exchange protocol (eg. native interface or ODBC)

Application Function
- Integrity Logic
- Data Access

Database
BCED approach

a single letter prefix (B, C, E or D).

DatabasePackage - loadMe(anObject) with the database data and
saveMe(anObject) to the database.

System software

Client
- Native DB interface
- ODBC/JDBC

Server
- Relational DB
- Object-relational DB
- Object DB

Reuse strategy

Granularity of reuse
- Class
- Component
- Solution Idea

Strategies for reuse
- Toolkits (class libraries)
  - Foundation
  - Architecture
- Frameworks
- Analysis and design patterns
Component

- A physical part of the system, a piece of implementation, a software program
- UML - five standard stereotypes for components
  - Executable (i.e., a directly executable module)
  - Library (i.e., a static or dynamic object library module)
  - Table (i.e., a database table)
  - File (i.e., a source code or data document)
  - Document (i.e., a human-readable document)

Component characteristics

- A unit of independent deployment (never deployed partially)
- A unit of third-party composition (i.e., sufficiently documented and self-contained to be "plugged into" other components by a third-party)
- Has no persistent state (i.e., cannot be distinguished from copies of its own; in any given application, there will be at most one copy of a particular component)
- Replaceable part of a system – can be replaced by another component that conforms to the same interface
- Fulfills a clear function and is logically and physically cohesive
- May be nested in other components

Component diagram

```
<<executable>>
InvoiceDLL

<<stored procedure>>
MaintainInvoiceUSP

<<executable>>
InvoiceEXE

<<executable>>
InvoiceDLL
```
Component vs package

- The **package** is a logical part of the system.
- At the logical level, every class belongs to a single **package**.
- At the physical level, every class is implemented by at least one **component** and it is possible that a component implements only one class.
- Abstract classes defining interfaces are frequently implemented by more than one **component**.
- Packages are typically larger architectural units than components. They tend to group classes in a horizontal way – by static proximity of classes in the application domain.
- Components are vertical groups of classes with behavioral proximity – they may come from different domains but they contribute to a single piece of business activity, perhaps a use case.

**Timetable**

![Diagram of Room Allocation](image)

Component vs class & interface

- Like classes, the components realize interfaces:
  - A component is a physical abstraction deployed on some computer node.
  - A class represents a logical thing that has to be implemented by a component to act as a physical abstraction.
- A component reveals only some interfaces of the classes that it contains.
- The interface that a component realizes may be implemented in a separate class. Such a class is called a **dominant class**.
  - Since the dominant class represents the interface of the component, any object inside the component is reachable from the dominant class via composition links.
  - "A dominant class subsumes the interface of the component".
Collaboration notation

<<realize>>
Browse Student List
<<realize>>
Enter Program of Study
Add Student to Course Offering

Collaboration diagram

gCust 
Customer

genNew

gConfWin 
ConfigurationWindow

getConf

*getConf (out item_rec)

gConfItem 
ConfigurationItem

displayComputer (item_recset)

gComp 
Computer

Message notations

loanPlease (in amount_req, out amount_granted)

amount_granted := loanPlease (amount_req)
Types of messages

- **Read messages** (interrogative, present-oriented messages)
  
  ```java
  aBank1.openingHours(in weekday, out hours)
  ```

- **Update messages** (informative, past-oriented messages)
  
  ```java
  aCustomer1.newCreditRating
  (in credit_rating, effective_date)
  ```

- **Collaborative messages** (imperative, future-oriented messages)
  
  ```java
  aBank1.loanPlease(in amount_req, out amount_granted)
  ```

Overriding vs overloading

- **The overriding** constitutes the basis for polymorphism. It means that there exist several methods with the same name in different classes.

- **The overloading** also means that there exist several methods with the same name but in the same class.
  
  - For example, apart from the previously defined method `loanPlease`, we may have another `loanPlease` method in the class `Bank`.
  - This second `loanPlease` method would include an additional argument specifying the minimum loan amount that a customer is prepared to take, as shown below:
    
    ```java
    aBank1.loanPlease(in amount_req, minimum_amount, out amount_granted)
    ```

Self messages

```java
longServiceLeave()
currentLeave()
leaveEntitlement()
anEmployee : Employee
```
Asynchronous messages

Callbacks

Sequence vs collaboration diagram

A Sequence Diagram puts the emphasis on the time sequence of messages between objects
- awkward and imprecise in representing alternative message paths – something that the Activity Diagrams excel in
- cumbersome in representing larger collaborations with many objects (although a careful arrangement of object lifelines can frequently improve the readability by a whole factor).

A Collaboration Diagram can explicitly show static relationships between objects along which the messages can flow
- provide for a better precision when visualizing such things as a polymorphic message
- permits showing more objects on the same graphical area
- the messages can be fully specified and annotated
Summary

- Typical IS applications are based on the **Client/Server** architectural principle
- **Three-tier systems** extend the basic C/S architecture
- BCE hierarchy of packages extended with a database interface package to create **BCED hierarchy**
- **The reuse choices** are between a toolkit reuse, framework reuse, and pattern reuse
- The reuse from external sources has to be aligned with the internal design of packages, components, classes, interfaces, computational nodes
- **The detailed design concentrates on collaborations**
- **Structural aspects of collaboration** are modeled in **Class Diagrams**; **behavioral aspects** – in **Collaboration Diagrams**
- **The OnLine Shopping** guided tutorial (separate Lecture Notes)