Lecture 17: Intro. to Design (Part 1)

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 6 of Maciaszek's "Requirements Analysis and System Design". © Addison Wesley, 2000

Goals for this Lecture

- Cover the material presented in Section 6.1 of the textbook
  - Introduce
    - Software Design
      - Architecture and Components

Introduction to Design

- Design consists of two major activities
  - Architectural Design
    - aka "High-Level Design"
    - layering of classes and packages
  - Detailed Design
    - aka "Low-Level Design"
    - develop collaboration models that realize the functionality of a system's use cases

Intro. to Design, continued

- Design is a low-level model of a system's architecture and its internal workings
- In design, models created during analysis (state, behavior, state change) are elaborated with technical details
  - such as the target software/hardware platform
- Analysis models thus become design models; we also create new models specific to the design phase

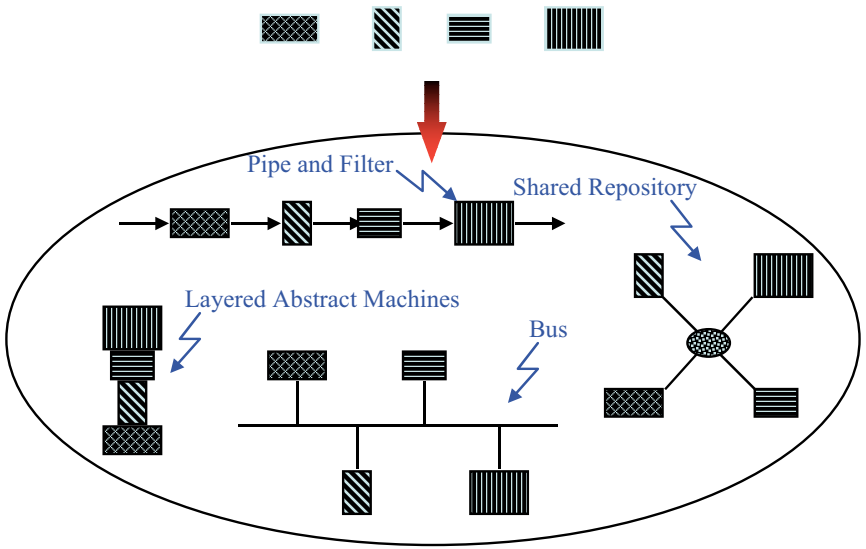
Intro. to Design, continued

- The description of a system in terms of its subsystems and modules is called its *architectural design*
- The description of the internal structure of each module is called a *detailed design*
- Detailed design develops interfaces for each module along with recommendations for data structures and algorithms that can help meet a system's non-functional requirements

Software Architecture

- The architectural design of a system involves a selection of (what Maciaszek calls) a *solution strategy* and with its *modularization*
- A solution strategy involves picking an *architectural style* that helps address the environmental constraints of a software system
  - For instance, does the system require remote access to a database (n-tier architecture)? Do multiple users have to share data they store locally (peer-to-peer architecture)

Software Architectural Styles



Client-Server Architecture

- Maciaszek spends some time talking about the standard client-server architecture
  - a client is a logical computing process that makes requests on a server process
    - clients typically interact directly with a user
  - a server is a logical computing process that services client requests
    - servers typically manage access to a data store
  - We say "logical" because a computing process may be a "client" in one request but a server in some other request (such as the middleware in a n-tier architecture)

N-tier Architecture

- See figure 6.2 on page 198
  - client-server architectures are sometimes generalized to contain additional layers where "application logic" is placed
    - this is also similar to Maciaszek's Boundary-Control-Entity design pattern
  - without a separate application layer, the application logic either goes in the client (thick client) or the server (thin client)

Database Terminology

- In architectures that contain a database component (typically located on a server)
  - an *active* database is one which can be programmed beyond standard data management
  - a database program is called a *stored procedure*; it is stored directly in its database
    - it can be invoked via a procedure/function call from a client or another stored procedure
  - a trigger is a special stored procedure that cannot be explicitly called; instead it automatically fires when a particular change is made to its database
    - triggers enforce integrity and consistency of a database

Application/Database Interaction

- When a database is present, a decision must be made as to what components implement which type of functionality
  - user interface
    - toolkit that "displays" information to user
  - presentation logic
    - logic that controls ui (the control part of the BCE)
  - application functionality
    - functional characteristics of the system
  - integrity logic
    - enterprise-wide business logic (application independent)
  - data access
    - logic that manages data persistence
- See Figure 6.3 on page 200

BCED Approach

- Maciaszek extends his Boundary-Control-Entity approach with a fourth layer: Database
  - See Figure 6.4 on page 200
- Database package separates classes that access a particular database from the core application domain objects that are stored in the Entity package
  - helps to make system database independent
    - well, sort of, writing database-independent SQL is an art

Types of Databases

- There are three main types of databases
  - relational databases
  - object-relational databases
  - object databases
- A fourth type is now emerging
  - XML database
- Typically, these technologies do not compete with one another
  - they excel at different issues and address the needs of different application domains; as a result the decision on a database technology must be based on current and expected application needs (more on this in chapter 8)

Reuse Strategies

- After selecting an architecture, it is good to spend time evaluating opportunities for software reuse
  - reusing software can save time and money
    - assuming the reused software has been deployed in some other project providing opportunities for finding and eliminating bugs
  - reuse involves selecting a granularity
    - are you going to reuse a class, a package, a system?

Reuse Strategies, continued

- Maciaszek considers three levels of granularity
  - a class
  - a component
  - a solution (pattern)
- Associated with these three granularities are
  - Toolkits (class libraries)
  - Frameworks
  - Analysis and Design Patterns

Toolkit Reuse

- A toolkit emphasizes code reuse at a class level
- Two types of toolkits
  - Foundation toolkits
    - primitive and structured data types and collections (e.g. String, Date, List, …)
  - Architecture toolkits
    - a toolkit that implements a particular architecture, such as a database or a GUI

Framework Reuse

- A framework emphasizes design reuse at a component level
- A framework typically implements an application architecture
  - A developer can produce a new application by subclassing or reusing framework classes and writing application-specific code
    - MacOS X provides Cocoa and Carbon frameworks for this purpose

Pattern Reuse

- A pattern is a documented solution that has been shown to work well in a number of situations
  - We shall discuss design patterns in more detail later this semester (and you will have a chance to implement a few as well!)

Components

- Architectures are made up of components connected together in a particular fashion (e.g. pipe-and-filter)
- A component is a physical part of a system; here physical refers to be stored on disk as well as being executable
- UML defines five standard component stereotypes
  - Executable (directly executable module)
  - Library (a static or dynamic object library)
  - Table (database table)
  - File (stored on disk)
  - Document (human-readable document)
- UML notation for components is shown on page 204

More on Components

- A component
  - is a unit of independent deployment
  - is a unit of third-party composition
    - meaning, it can be "plugged into" other components
  - has no persistent state
  - is replaceable
  - fulfills a clear function
  - may be nested within other components

Component Diagrams

- Figure 6.6 on page 205
  - A component diagram shows components and their relationships
    - A dependency relationship indicates that one component requires the services of another component
    - A composition relationship indicates that one component contains another component
    - A component diagram can use the "lollipop" notation to indicate a particular interface supported by a component

Components vs. Packages

- A package is a logical part of a system
  - logically, every class of a system belongs to a particular package
- Physically, every class is implemented by at least one component
  - A component can implement only one class, although typically this is not the case
  - Abstract classes are frequently implemented by more than one component

Components vs. Packages, cont.

- Packages tend to group classes horizontally by static proximity within an application domain
  - such as placing all control classes into a control package
- Components tend to group classes vertically based on behavioral proximity
  - such as instantiating a boundary class, its control class, and relevant entity classes within a component to support a particular use case
- Packages are often "implemented" via several components; see figure 6.7 on 206

Components vs. class and interface

- Like classes, components implement interfaces (see figure 6.8 on 207)
  - however, components are physical entities deployed on computers; classes are logical entities that require components to implement them
  - furthermore, components may not reveal all of the interfaces of its classes

Deployment

- UML provides a deployment diagram for documenting system architectures
  - Deployment diagrams consist of nodes (notated as cubes, or with special icons) that are connected via "connection relationships"
    - connection relationships can be labeled with a network protocol that indicates how the nodes communicate or with a phrase that characterizes the connection in some way (such as "nightly download")
  - Components can be placed within nodes (nodes execute components) to indicate how a system is to be physically implemented
    - (see figure 6.11 and 6.12 on pages 208-209)