Lecture 17: Maciaszek's Take on Design

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2003

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

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
  - "boundary" classes and controller classes
- Analysis models thus become design models; we also create new models specific to the design phase

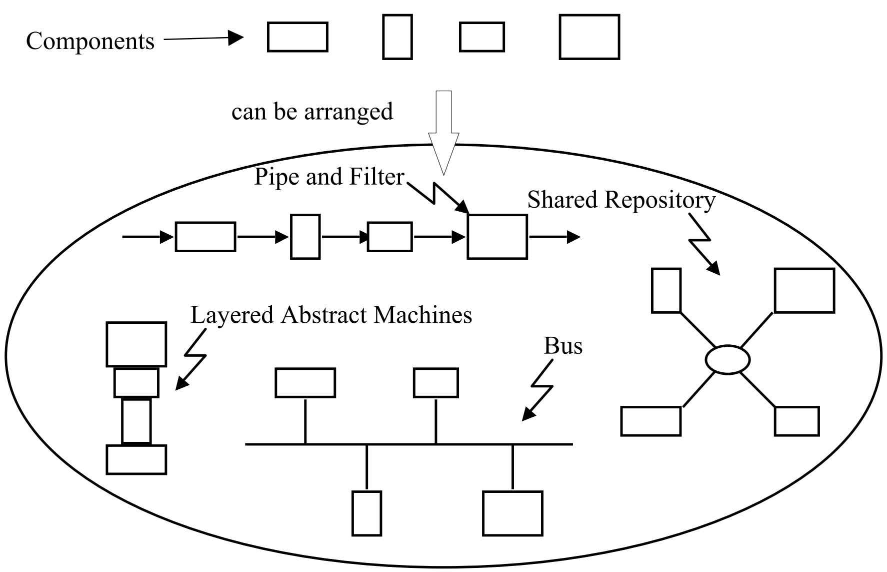
Intro. to Design, continued

- 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 is concerned with the selection of (what Maciaszek calls) a *solution strategy* and with a system's *modularization*
  - The solution strategy determines how a system's modules (or subsystems) are arranged
- 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



Reuse Strategies

- After selecting an architectural style, 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 (stored within itself)
  - is replaceable (by other components)
  - fulfills a clear function
  - may be nested within other components

Component Diagrams

- A component diagram shows components and their relationships
  - A *dependency relationship* indicates that one component requires the services of another component
    - Notated with a dotted line that points to the required component; Figure 6.6 on page 205
  - A *composition relationship* indicates that one component contains another component
    - Notated with the standard UML composition notation (black diamond)
  - A component diagram can use the "lollipop" notation to indicate the interfaces supported by a component
    - See figure 6.8 on page 207

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
  - Think: "A set of classes is compiled into a component"
  - A component can implement only one class, although typically this is not the case
  - Abstract/Entity 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
  - This figure means that each class in the Timetable package has been "covered" by at least one of the three components

Components vs. Class/Interface

- Components are thus collections of classes; Each component may implement one or more interfaces;
  - These interfaces may not have a one-to-one correspondence with the methods of the component's implemented classes
  - But, the classes are included in the component to help carry out the activities supported by the component's interface(s)

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"
    - (see figures 6.9 and 6.10 on pages 207-208)
    - 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)

Detailed Design

- In OO A&D, detailed design is a direct continuation from analysis
  - Our objective is to transform analysis models into design models that can be implemented by developers
- Architectural design impacts detailed design by selecting a target hardware/software platform (or platforms) and by selecting the components that will deploy our implemented design into the "real world"

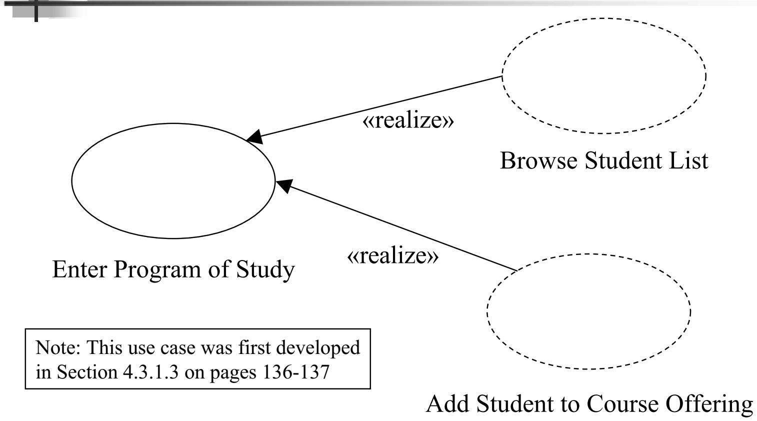
Detailed Design, continued

- In analysis, we simplify models by abstracting away (or deferring) technical details that would either
  - get in the way of understanding our application domain, or…
  - lead us down an implementation path too early and hence constrain our choices later in development
- In detailed design, we do the opposite
  - we start with analysis models and add technical details, or…
  - “drill down” a layer of abstraction on a particular analysis model and start creating a “design time” model from scratch

Collaboration

- The UML uses the term “collaboration” to refer to sets of objects collaborating to perform a task
  - In particular, collaborations are used to specify the *realization* of use cases and operations
- Collaborations are notated as ellipses with dashed borders (see next slide)

Collaboration Example



Comments on Example

- Each collaboration needs an associated model that displays the details of the collaboration
  - Think “Interaction Diagram”
  - In particular, a collaboration diagram
    - Similar to a sequence diagram (indeed one can be converted into the other) but emphasizing different aspects of the collaboration
      - sequence diagrams emphasize the order of messages between objects
      - collaboration diagrams emphasize the associations between objects and the messages that flow over these associations

Collaboration Diagrams

- Collaboration diagrams consist of objects
  - object names are interpreted as "role : class"
  - collections are shown as "stacks of objects"
- Associations between objects are shown; messages travel across the association in the direction indicated
  - Messages can be numbered to show the exact order in which messages are generated
  - As with other UML diagrams, messages sent to collections can be prefixed with an asterick ("*") to indicate that the same message is sent to each member of the collection
- The collaboration diagram in Fig. 6.14 (pg. 210) corresponds to the sequence diagram in Fig. 2.33 (pg. 66)

Realization of Use Cases

- Collaborations are to design, what use cases are to analysis
  - They help "drive" their respective stages
  - However, due to differences in abstraction, typically multiple collaborations are needed to realize a single use case
- Collaborations consist of a structural part and a behavioral part
  - The structural part is the subset of the class diagram that covers each of the objects participating in the collaboration
    - developing a collaboration during design will lead to the original class diagram being updated with new operations along with operation signatures
  - The behavioral part is an interaction that defines the specific interaction of the collaboration's objects

Example

- Maciaszek provides an example of realizing a use case with a collaboration on pages 217-221
  - Makes use of the "Enter Program of Study" use case for the University Enrollment example
- First some backtracking
  - First, look at the class diagram on page 129
  - Second, look at the sequence diagram on page 130
- Now, the collaboration
  - First, the structural part on page 218
    - boundary and controller classes have been added
  - Second, the behavioral part on page 219
    - This is not quite equivalent to the sequence diagram because it includes the boundary/controller classes

Summary

- Maciaszek's Take on Design
  - High Level Design
    - Architectural Style, Components, Deployment
  - Low Level Design
    - Collaborations
      - Consist of "elaborated" class diagrams and interaction diagrams that build on existing analysis models but finally take into consideration the "machine concepts": e.g. boundary and controller classes, specific toolkits, computing platforms, etc.