Chapter 10 – Component-Level Design

Overview
The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design. Component-level design occurs after the data and architectural designs are established. The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation. Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps.

Component Definitions
- Component is a modular, deployable, replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- Object-oriented view is that component contains a set of collaborating classes
  - Each elaborated class includes all attributes and operations relevant to its implementation
  - All interfaces communication and collaboration with other design classes are also defined
  - Analysis classes and infrastructure classes serve as the basis for object-oriented elaboration
- Traditional view is that a component (or module) reside in the software and serves one of three roles
  - Control components coordinate invocation of all other problem domain components
  - Problem domain components implement a function required by the customer
  - Infrastructure components are responsible for functions needed to support the processing required in a domain application
  - The analysis model data flow diagram is mapped into a module hierarchy as the starting point for the component derivation
- Process-Related view emphasizes building systems out of existing components chosen from a catalog of reusable components as a means of populating the architecture

Class-based Component Design
- Focuses on the elaboration of domain specific analysis classes and the definition of infrastructure classes
- Detailed description of class attributes, operations, and interfaces is required prior to beginning construction activities

Class-based Component Design Principles
- Open-Closed Principle (OCP) – class should be open for extension but closed for modification
- Liskov Substitution Principle (LSP) – subclasses should be substitutable for their base classes
- Dependency Inversion Principle (DIP) – depend on abstractions, do not depend on concretions
- Interface Segregation Principle (ISP) – many client specific interfaces are better than one general purpose interface
- Release Reuse Equivalency Principle (REP) – the granule of reuse is the granule of release
- Common Closure Principle (CCP) – classes that change together belong together
- Common Reuse Principle (CRP) – Classes that can’t be used together should not be grouped together

Component-Level Design Guidelines
- Components
  - Establish naming conventions in during architectural modeling
  - Architectural component names should have meaning to stakeholders
  - Infrastructure component names should reflect implementation specific meanings
  - Use of stereotypes may help identify the nature of components
- Interfaces
  - Use lollipop representation rather than formal UML box and arrow notation
  - For consistency interfaces should flow from the left-hand side of the component box
  - Show only the interfaces relevant to the component under construction
- Dependencies and Inheritance
  - For improved readability model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)
  - Component interdependencies should be represented by interfaces rather that component to component dependencies
Cohesion (lowest to highest)

- Utility cohesion – components grouped within the same category but are otherwise unrelated
- Temporal cohesion – operations are performed to reflect a specific behavior or state
- Procedural cohesion – components grouped to allow one be invoked immediately after the preceding one was invoked with or without passing data
- Communicational cohesion – operations require same data are grouped in same class
- Sequential cohesion – components grouped to allow input to be passed from first to second and so on
- Layer cohesion – exhibited by package components when a higher level layer accesses the services of a lower layer, but lower level layers do not access higher level layer services
- Functional cohesion – module performs one and only one function

Coupling

- Content coupling – occurs when one component surreptitiously modifies internal data in another component
- Common coupling – occurs when several components make use of a global variable
- Control coupling – occurs when one component passes control flags as arguments to another
- Stamp coupling – occurs when parts of larger data structures are passed between components
- Data coupling – occurs when long strings of arguments are passed between components
- Routine call coupling – occurs when one operator invokes another
- Type use coupling – occurs when one component uses a data type defined in another
- Inclusion or import coupling – occurs when one component imports a package or uses the content of another
- External coupling – occurs when a component communicates or collaborates with infrastructure components (e.g. database)

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
   a. Specify message details when classes or components collaborate.
   b. Identify appropriate interfaces for each component.
   c. Elaborate attributes and define data types and data structures required to implement them.
   d. Describe processing flow within each operation in detail.
4. Identify persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioral representations for each class or component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level diagram representation and consider alternatives.

WebApp Component-Level Design

- Boundary between content and function often blurred
- WebApp component is defined is either a:
  o well-defined cohesive function manipulates content or provides computational or data processing for an end-user or
  o cohesive package of content and functionality that provides the end-user with some required capability

WebApp Component-Level Content Design

- Focuses on content objects and the manner in which they may be packaged for presentation to the end-user
- As the WebApp size increases so does the need for formal representations and easy content reference and manipulation
- For highly dynamic content a clear structural model incorporating content components should be established

WebApp Component-Level Functional Design

- WebApps provide sophisticated processing functions
  o perform dynamic processing to create content and navigational capability
  o provide business domain appropriate computation or data processing
  o provide database query and access
  o establish interfaces with external corporate systems
- WebApp functionality is delivered as a series of components developed in parallel
- During architectural design WebApp content and functionality are combined to create a functional architecture
The functional architecture is a representation of the functional domain of the WebApp and describes how the components interact with each other.

**Traditional Component-Level Design**
- Each block of code has a single entry at the top
- Each block of code has a single exit at the bottom
- Only three control structures are required: sequence, condition (if-then-else), and repetition (looping)
- Reduces program complexity by enhancing readability, testability, and maintainability

**Design Notation**
- Graphical
  - UML activity diagrams
  - Flowcharts – arrows for flow of control, diamonds for decisions, rectangles for processes
- Tabular
  - Decision table – subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events
- Program Design Language (PDL)
  - Structured English or pseudocode used to describe processing details
  - Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
  - Free syntax of natural language for describing processing features
  - Data declaration facilities for simple and complex data structures
  - Subprogram definition and invocation facilities

**Component-Based Development**
- CBSE is a process that emphasizes the design and construction of computer-based systems from a catalog of reusable software components
- CBSE is a time and cost effective
- Requires software engineers to reuse rather than reinvent
- Management can be convinced to incur the additional expense required to create reusable components by amortizing the cost over multiple projects
- Libraries can be created to make reusable components easy to locate and easy to incorporate them in new systems

**Domain Engineering**
- Intent is to identify, construct, catalog, and disseminate a set of software components applicable to existing and future products in a particular application domain
- Overall goal is to establish mechanisms that allow for component sharing and reuse during work on both new and existing systems
- Includes three major activities: analysis, construction, dissemination

**Domain analysis steps**
1. Define application domain to be investigated
2. Categorize items extracted from domain
3. Collect representative applications from the domain
4. Analyze each application from sample and define analysis classes
5. Develop an analysis model for classes

**Component Qualification**
- Ensures that a candidate component will: perform function required, fit into the architectural style specified for system, and exhibit required quality characteristics
- Factors to consider
  - Application programming interface (API)
  - Development and integration tools required by the component
  - Run-time requirements (resource usage or performance)
  - Service requirements (component dependencies)
  - Security features
Component Adaptation
- Seeks to avoid conflicts between the qualified component and the application architectures to facilitate easy integration
- Common adaptation techniques
  - White-box wrapping – integration conflicts removed by making code-level modifications to the code
  - Grey-box wrapping – used when component library provides a component extension language or API that allows conflicts to be removed or masked
  - Black-box wrapping – requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts

Component Composition Architectural Ingredients
- Assembles qualified, adapted, and engineered components to populate established application architecture
- Ingredients for creating an effective infrastructure to enable components to coordinate with one another to perform common tasks
  - Data exchange model – similar to drag and drop type mechanisms should be defined for all reusable components, allow human-to-software and component-to-component transfer
  - Automation – tools, macros, scripts should be implemented to facilitate interaction between reusable components
  - Structured storage – heterogeneous data should be organized and contained in a single data structure rather several separate files
  - Underlying object model – ensures that components developed in different languages are interoperable across computing platforms

Representative Component Standards
- Object Management Group (OMG)/CORBA (common object request broker architecture)
- Microsoft COM (component object model)
- Sun JavaBeans Components (Bean Development Kit)

Analysis and Design for Reuse
- The requirements model needs to be analyzed to determine the elements that point to existing reusable components
- Specification matching can help find components to extract from the reuse library
- If no reusable components can be found, developers should design a new component keeping reuse in mind
- When designing for reuse developers should consider several key issues
  - Standard data – standard global data structures identified in application domain and components developed to use these data structures
  - Standard interface protocols – three interfaces levels should be established (intramodular, external technical, and human/machine)
  - Program templates – used in architectural design of new programs

Classifying and Retrieving Components
- Describing reusable components
  - concept - what the component does
  - content - how the concept is realized
  - context - specifies conceptual, operational, and implementation features of the software component within its domain of application
- Reuse environment elements
  - component database capable of storing software components and classification information to allow their retrieval
  - library management system to allow access to database
  - software component retrieval system that enables client software to retrieve components and services from library server
  - CBSE tools that support integration of reused components into a new design or implementation