Chapter 17 Notes

Component-Based SWE = an approach to SW systems development based on reusing SW components.

CBSE = the process of defining, implementing, and integrating or composing loosely coupled, independent components into systems.

Why CBSE?

* SW system are becoming larger and more complex
* Customers are demanding more dependable SW that is delivered and deployed more quickly
* CBSE lowers complexity and deployment time

Essentials of CBSE

* Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation.
* Component standards that facilitate the integration of components. These standards are embodied in a component model. They define, at the minimum, how component interfaces should be specified and how components communicate. Some models go much further and define interfaces that should be implemented by all conformant components.
* Middleware provides SW support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. Middleware may also provide support for resource allocation, transaction management, security, and concurrency.
* A development process that is geared to CBSE. You need a development process that allows requirements to evolve, depending on the functionality of available components.

Design principles of CBSE

* Components are independent so they do not interfere with each other’s operation. Implementation details are hidden. The component’s implementation can be change without affecting the rest of the system
* Components communication through well-defined interfaces. If these interfaces are maintained, one component can be replaced by another, which provides additional or enhanced functionality.
* Components infrastructures offer a range of standard services that can be used in application systems. This reduces the amount of new code that has to be developed.

**Components and Component Models (Section 17.1)**

* Component = An independent SW unit that can be composed with other components to create a SW system. It can also be viewed as a service provider.
* Components are independent, and they are the fundamental unit of composition in a system.
* Component Characteristics
  + Standardized
    - Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment
  + Independent
    - A component should be independent – it should be possible to compose and deploy it without having to use other specific components.
  + Composable
    - For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
  + Deployable
    - To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed.
  + Documented
    - Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
* Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:
  + The component is an independent executable entity that is defined by its interfaces.
  + The services offered by a component are made available through an interface and all interactions are through that interface.
* Components have 2 related interfaces. These interfaces reflect services that the component provides and the services that the component requires to operate correctly:
  + The “provides” interface defines the services provided by the component. This interface is the component API. It defines the methods that can be called by a user of the component. In a UML component diagram, the “provides” interface for a component us indicated by a circle at the end of a line from the component icon
  + The “requires” interface specifies what services must be provided by other components in the system if a component is to operate correctly. If these are not available, then the component will not work. In UML component diagrams, the “requires” interface is a semicircle at the end of a line from the component icon.
* Critical Difference between component as an external service and as a program element is:
  + Services are completely independent entities and don’t have a “requires” interface
* **Component Models (Section 17.1.1)**
  + A component model is a definition of standards for component implementation, documentation, and deployment. These standards are for component developers to ensure that component can interoperate.
  + Basic Elements of an Ideal Component Model
    - Interfaces
      * Components are define by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition. The model should also specify the language used to define the component interfaces. Some component models must require specific interfaces that must be defined by a component.
      * Includes Interface Definition, Composition, and Specific Interfaces
    - Usage
      * In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. Component meta-data is data about the component itself, such as information about its interfaces and attributes. This is used by users to find out what services are provide and required. Components are generic entities and, when deployed, they have to be configured to fit into an application system.
      * Includes Naming Convention, Customization, Meta-Data Access
    - Deployment
      * The component model includes a specification of how components should be packaged for deployment as independent, executable entities. Because components are independent entities, they have to be packages with all supporting SW that is not provided by the component infrastructure, or is not define in a “requires” interface. Deployment information about the contents of a package and its binary organization. Components will have to change in response to changing requirements. The component model may therefore include rules governing when and how component replacement is allowed. And finally, the component model may define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.
      * Includes Packaging, Documentation, and Evolution Support
  + For components that are implemented as program units, the component sets out the services to be provided by the middleware that supports the executing components. The services provided by a component model implementation fall into 2 categories:
    - Platform Services
      * Enable components to communicate and interoperate in a distribute environment. These are the fundamental services that must be available in all component-based systems
    - Support Services
      * Common services that are likely to be required by many different components.
  + The middleware implements the component services and provides interfaces to these services.
* CBSE Processes (Section 17.2)
  + Are processes that support CBSE.
  + 2 types of CBSE Processes
    - Development for Reuse
      * This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components
      * Objective is to create one or more reusable components
    - Development with Reuse
      * This is the process of developing new applications using existing component and services
      * Objective is to discover reusable components and design your system to make the most effective use of them
  + Processes
    - Component Acquisition
      * Process of acquiring components for reuse or development into a reusable component.
    - Component Management
      * Concerned with managing a company’s reusable components ensuring that they are properly cataloged, stored, and made available for reuse.
    - Component Certification
      * Process of checking a component and certifying that it meets its specification
  + CBSE for Reuse (Section 17.2.1)
    - The process of developing reusable component and making them available for reuse through a component management system.
    - CBSE for reuse is most likely to take place within an organization that has made a commitment to reuse-driven SWE.
    - To make components reusable, you have to adapt and extend the application-specific components to create more generic and therefore more reusable versions.
      * This adaption has a cost -> you must decide:
        + Whether a component is likely to be reused

If the system includes stable domain abstractions (aka business objects that don’t change much like the abstraction of a Bank Account), then it can probably be reused

* + - * + Whether the cost savings from future reuse justify the costs of making the component reusable

You must consider the costs of changes that are required to make the component reusable (costs of documentation, validation, etc.)

* + - Changes you may make to make a component more reusable
      * Removing application-specific methods
      * Changing names to make them more general
      * Adding methods to provide more complete functional coverage
      * Making exception handling consistent for all methods
      * Adding a configuration interface to allow the component to be adapted to different situations of use
      * Integrating required components to increase independence
    - The problem of exception handling is
      * Component should not handle exceptions themselves (because each application will want to handle them differently)
      * But, not handling exceptions will:
        + Lead to bloated interacts that are harder to understand
        + The operation of the component may depend on local exception handling and changing this may hurt the component’s operation
    - Reusable components are usually more dependable
    - As you add generality to a component, you increase its reusability but this means the component has more operations and is more complex which makes it harder to understand.
    - There is a trade-off between reusability and usability of a component.
    - Legacy System can be reusable
      * To make these reusable, you must make a wrapper that defines the component interfaces. Although the wrapper is a fairly complex piece of SW, the cost of wrapper development is often much less than the cost of re-implementing the legacy system.
    - Once you have developed and tested a reusable component or service, this then has to be managed for future reuse. Management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions.
    - A company with a reuse program may carry out some form of Component Certification
      * Certification = someone apart for the developer checks the quality of the component and ensures that it has reached a quality standard, before it is made available for reuse.
  + **CBSE with Reuse (Section 17.2.2)**
    - Process
      * Outline System Requirements
      * Identify Candidate Components
        + Component Search
        + Component Selection
        + Component Validation
      * Modify Requirements according to Discovered Components
      * Architectural Design
      * Identify Candidate Components
      * Compose Components to Create System
    - Differences between CBSE with Reuse and Software processes for original software development are:
      * The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements.
      * Requirements are defined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported. User may be willing to change their minds if this means cheaper or quicker delivery
      * There is a further component search and design refinement activity after the system architecture has been designed. Some apparently usable components may turn out to be unsuitable or do not work properly with other chosen components.
      * Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing adaptors that reconcile the interfaces of incompatible components.
    - During the architectural design stage, you may choose a component model and implementation platform.
    - In the Identify Candidate Components, you need to find a component that meet your needs and do some (but not detailed) checking the component is suitable. In the later stage, after the system architecture has been designed, you should spend more time on component validation.
      * Component Validation involves developing test cases for a component and developing a test harness to run component tests.
        + Cons

The component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. Components are usually specified informally therefore, you don’t have enough information to develop a complete set of tests

* + - * + You may also have to check whether or not the component has any malicious code or unneeded functionality.

Problem is that the component may activate unneeded functionality by itself and thus, waste computing power

* **Component Composition (Section 17.3)**
  + Is the process of integrating components with each other, and with specially written “glue code” to create a system or another component.
  + 3 Types of Composition
    - Sequential Composition
      * Create a new component from 2 existing component by calling the existing components in sequence.
      * Services offered by A are called and the results from A are used in the call to services offered by B.
    - Hierarchical Composition
      * Component A directly calls on component B for services.
      * This means the “provides” interface of B must match the “requires” component of A
    - Additive Composition
      * Occurs when 2 or more components are put together (Added) to create a new component, which combines their functionality. The “provides” and “requires” interfaces of the new component is a combination of the corresponding interfaces in A and B.
    - \*\*In all of these cases, you may write glue code that links components.
  + 3 Types of Interface Incompatibility
    - Parameter Incompatibility
      * The operations on each side of the interface have the same name but their parameter types or the number of parameters differ
    - Operation Incompatibility
      * Names of the operations in the “provides” and “requires” interfaces are different
    - Operation Incompleteness
      * “Provides” interface of a component is a subset of the “Requires” interface of another component or vice versa.
    - \*\*\*In all cases, you solve the problem by writing an adaptor that reconciles the interfaces of the 2 components being reused.
  + Object Constraint Language (OCL)
    - Part of UML designed to describe constraints in UML object models. It allows you to express predicates that must always be true, that must be true before a method has executed, and that must be true after a method has executed.
  + When you create a system by composing components, you may find that there are potential conflicts between functional and non-functional requirements. The decisions where you may have to take trade-offs into account are:
    - What composition of components is most effective for delivering the functional requirements for the system?
    - What composition of the components will make it easier to adapt the composite component when its requirements change?
    - What will be the emergent properties of the composed system? These are properties such as performance and dependability. You can only assess these once the complete system is implemented.