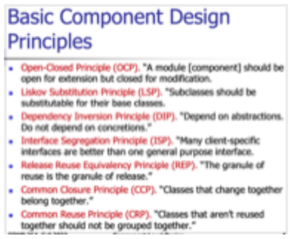
8: Component-Level Design

What is a component:
- A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.
- Object Oriented view: A component contains a set of collaborating classes.
- Traditional view: A component contains processing logic, internal data structures that are required to implement the processing logic.
- Process-related view: Building systems out of reusable software components.

Class-based Component-Level Design: classes with its attributes, methods diagram
Traditional Component-Level Design: Control component, problem domain component, infrastructure

Basic Component Design Principles:



Basic Component Design Principles

- Open-Closed Principle (OCP). "A module [component] 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 aren't reused together should not be grouped together."

Component-Level Design Guidelines:
- Components: Name for components that are specified as part of the architectural model
- Interfaces: Provide important information about communication and collaboration
- Dependencies and Inheritance: To model dependencies from left to right and inheritance from bottom to top.

Cohesion:
- Traditional view: single minded module
- Object Oriented view: Implies that a component encapsulates only attributes and operations that are related to one another and the component itself.
- Levels of cohesion:
  - Functional: Module performs one and only one computation
  - Layer: Occurs when a higher layer accesses the services of a lower layer, lower layers do not access higher layers

    - Communicational: All operations that access the same data are defined within one class.

Coupling:
- Traditional view: Component is connected to other components
- Object-Oriented view: Qualitative measure of the degree to which classes are connected to one another.
- Levels of coupling:
    - Content: Occurs when one component modifies data that is internal to another component.
    - Control: Occurs when control flags a passed to components to requests alternate behaviours when invoked.
    - External: Occurs when a component communicates or collaborates with infrastructure components.

Component Level Design:
1. Identify all classes related to problem domain
2. ‘’’ infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
4. Describe persistent data sources and identify the classes required to manage them.
5. Develop and elaborate behavioural representations for a class or component.
6. Elaborate deployment diagrams to provide implementation detail
7. Factor every component level design representation and always consider alternatives.

Component level design for WebApps:
WebApp component is well defined cohesive function, it incorporates elements of content design and functional design

WebApp focuses on content objects and the manner in which they may be packaged for presentation to a webApp end-user.

Construct WebApp functional components that are identical in form to software components for conventional software.
——————————————

Traditional Component-Level Design:
- Design of processing logic is governed by the basic principles of algorithm design and structured programming.
- Design of data structures is defined by the data model developed for the

system.
- Design of interfaces is governed by the collaborations that a component must effect.

Component-Based Software Engineering (CBSE):
- Commercial off the shelf (COTS) components available to implement the requirement?
- Internally developed reusable components available to implement the requirement?
- The interfaces for available components compatible within the architecture of the system to be built?

CBSE benefits:
- Reduced lead time: faster to build complete applications from existing components
- Greater return on investment (ROI): Save buying components rather redeveloping
- Leveraged costs of developing components: Reusing components in multiple app; save
- Enhanced quality: Components are reused and tested in many different app
- Maintenance of component based app: Can be easy to replace components with new components.

CBSE risk:
- Selection: Difficult to predict component behaviour
- Integration risks: Lack of standards between components
- Quality risks: Unknown design assumptions made for the components makes testing more difficult.
- Security risk: A system can be used unintended way.
- System evolution risks: Updated components may be incompatible with user requirements.

Component Refactoring:
- To improve quality is a good practice.
- Changing software and failing to document the changes can lead to increasing technical debt.
- Identify the most cost effective refactoring opportunities.