Component Level Design – Q&A

1. How does the object-oriented view of component-level design differ from the traditional view?
   The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domains. Classes are elaborated by specifying messaging details, identifying interfaces, defining attribute data structures, and describing process flow for operations. In the traditional view, three of components are refined: control modules, domain modules, and infrastructure modules. This requires representations to be created for data structures, interfaces, and algorithms for each program module in enough detail to generate programming language source code.

2. Describe the differences between the software engineering terms coupling and cohesion?
   Cohesion implies that a component or class encapsulates only the attributes and operations closely related to one another and to the class itself. Coupling is a qualitative measure of the degree to which components are connected to one another.
3. What are the steps used to complete the component-level design for a software development project?
   - Identify all design classes that correspond to the problem domain.
   - Identify all design classes that correspond to the infrastructure domain.
   - Elaborate all design classes that are not acquired as reusable components.
   - Identify persistent data sources (databases and files) and identify the classes required to manage them.
   - Develop and elaborate behavioral representations for each class or component.
   - Elaborate deployment diagrams to provide additional implementation detail.
   - Factor every component-level diagram representation and consider alternatives.

4. The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for object-oriented and traditional software.Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

   Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component. In object-oriented languages (e.g. Java or Smalltalk) components are classes or objects. In traditional languages (e.g. C or Fortran) components are functions or procedures. In mixed languages like C++ components might be either functions or classes.

5. Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?
   The Open-Closed Principle (OCP) states module [component] should be open for extension but closed for modification. The designer creates the designer should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the

need to make internal (code abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

6. Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?

   Dependency Inversion Principle (DIP) states, depend on abstractions. Do not depend on concretions. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

7. Select a small portion of an existing program (approximately 50 to 75 source lines). Isolate the structured programming constructs by drawing boxes around them in the source code. Does the program excerpt have constructs that violate the structured programming philosophy? If so, redesign the code to make it conform to structured programming constructs. If not, what do you notice about the boxes that you've drawn?

8. All modern programming languages implement the structured programming constructs. Provide examples from three programming languages.

9. Why is "chunking" important during the component-level design review process?

   People can on keep track of a small number (5 to 9) of things at a time in short term memory. Chunking allows reviewers to combine related concepts into larger pieces or bigger chunks. The components can serve as chunks (if the components are highly cohesive and loosely coupled) making it easier for reviewers to keep track of the interactions of several components during a design review rather than a large number of individual classes and their methods.