Deeper

- What is the difference between requirements and specifications?
  - *Some requirements may also appear in the specifications, but requirements are more likely to be expressed in terms of user experience, while specifications are expressed in terms of component functionality.*
  - *There may be many specifications that result from the architecture and design rather than directly from the requirements.*

Deeper

- Give examples of s/w system components

Compile time	*programs, files, classes*
Link time	*object modules, libraries*
Install time	*load modules, programs plug-ins*
	Dynamically Loadable Libraries
	configuration and data files
Run time	*processes, daemons, servers*

Deeper

- Try to find the line between specification and design?
  - *Functional specifications are the contract between a component and the rest of the system. They define correct component behavior, but do not specify how the component should be implemented.*
  - *Technical specifications often capture key design decisions (to provide guidance to the implementors), and so become design documents.*

Deeper

- How can learning design patterns improve our system and component designs?
  - *Knowing good solutions to common problems will enable us to easily solve those problems when we encounter them. This will improve both our speed and the quality of our work.*

Deeper

- How can studying design patterns improve our ability to design systems and components (even where studied patterns do not apply)?
  - *Studying architectures and reflecting on how they work contributes to our ability to recognize and analyze problems, and to device solutions that better address the full range of constraints.*

Deeper

- Why can't design patterns solve most of our architectural problems for us?
  - *Every new architectural problem has a unique context and set of constraints. Design patterns can contribute elements to the solution, but it still remains to propose a complete set of elements, that operate as an integrated whole, and satisfy all of the system's requirements.*

Deeper

- How do the patterns we read about compare with these criteria:
  - *adapter* — *better behaved*
  - *façade* — *hide complexity*
  - *proxy* — *hide complexity*
  - *strategy* — *hide complexity*
  - *abstract factory* — *hide complexity*
  - *singleton* — *hide complexity*
  - *object pool* — *hide complexity*

Deeper

What do we mean by a class should be well abstracted?

Start out with a clear notion of the ADT

- *a single abstract object type*
- *a convenient and powerful building block*
- *with a reasonably intuitive meaning*
  - *(not necessarily an immediately obvious one)*

Take a general view of the underlying ADT

- *consider a wide range of implementations*
  - *try not to presume or preclude any of them*
- *consider a wide range of applications*
  - *try not to presume or preclude any of the*

Deeper

- What do we mean by a class should be cohesive?

start out with a clear notion of the ADT

a single abstract object type
with a reasonably intuitive meaning
(not necessarily an immediately obvious one)

methods/properties should be complete

provide everything needed to use the ADT
enable full exploitation of its capabilities

methods/properties should be consistent

don't incorporate other ADT's into the class
don't add application-specific options

Deeper

- What do we mean by a class should employ good information hiding?

all public methods and properties

should follow naturally from the abstraction

public methods must be sufficient to ...

fully exploit capabilities of the ADT
enable application-specific sub-classes

public method interfaces should not ...

subject clients to implementation complexity
over-constrain possible implementations

everything else should be private

Deeper

- Why is the open/closed principle a good test of abstraction?

Specific applications can extend the class

adding application specific attributes
adding application specific methods
creating an application-tailored sub-class

They should not require changes to base class

If it does, the base class wasn't well abstracted. It is not a suitable foundation for my application.
Don't override the base-class methods.
Go back to the base-class and fix it.

Deeper

- Why is the Liskov substitution principle a good test of abstraction?

A subclass will substitute for its base class if it does not incompatibly override any base class methods.

You should have no need to override any operations in the base class. If you do have to override a method in the base class, that means the base class was not a good foundation upon which to build.

It's was not well abstracted for your purposes.

Deeper

- Why is the dependency inversion principle a good test of abstraction?
  - *If you have to look inside of the base class to use it for your application, it was not well enough abstracted.*

Deeper

- What is the difference between programming <u>in</u> a language and programming <u>into</u> a language?
  - *If you start out with a good design, and translate it into some language, you are programming <u>into</u> that language.*
  - *If you limit your design concepts to notions that are explicitly supported by a particular language, you are programming <u>in</u> that language.*

Deeper

- Why does the common closure principle make sense?
  - *If class B depends on the implementation of class A, and we deliver them in separate packages, we make it possible for someone to get an updated class A without getting the correspondingly updated class B.*
  - *This is an accident waiting to happen.*

Deeper

- Give a few examples of a unidirectional reference?
  - *If an appointment object contains, within it, a date object.*

  - *If a course object includes a pointer to the object for the associated instructor.*

  - *A car object is (in some way) associated with the object for the current driver.*

Deeper

- What is the difference between aggregation and composition?
  - *Composition is aggregation with strong ownership.*
  - *In simple aggregation, the whole includes a reference to the part, but the part may outlive the whole and be referenced by other objects.*
  - *In a composition relationship, the part is owned by and ends with the whole.*

Deeper

- Give a few examples of bidirectional associations?
  - *One course, one roster, and one grade book might all be mutually associated with one-another.*
  - *The association could be implemented with pointers, a shared index, or a higher level object that contained or referenced the others. In this last case, the higher level object might (if it has properties and/or methods) be an instance of an association class.*

Deeper

- What makes UML class diagrams good for sketching out a design?
  - *The language is fairly complete, but you are free to leave out as much as you want.*
  - *You can list only the methods and properties that are relevant to the current picture.*
  - *You can show only the relationships that are relevant to the current picture.*

Deeper

- How do dependencies differ from associations?
  - *Associations are structural relationships between objects (e.g. one contains or points to the other).*
  - *Dependencies are relationships between classes, where one class uses the information or services of another (e.g. being derived from or making calls to).*

Deeper

- Why is the same notation used for constraints and comments?
  - *Both provide supplementary information about the object, attribute or method.*
- How might a constraint differ from a comment?
  - *A constraint might be part of a declaration (e.g* **readonly***) or an assertion that could turn into code.*

Deeper

- What appears in class diagrams, but wouldn't appear in an object diagram?
  - *Class inheritance/derivation information has no place in an object diagram.*
  - *Type declarations might be present (as reminders) on attributes, but method declarations make very little sense.*

Deeper

- Would it make sense to show interface providers on an object diagram?
  - *Definitely. If the interface provider was bound to an object at run time, it could be very important to note which implementation was being used.*