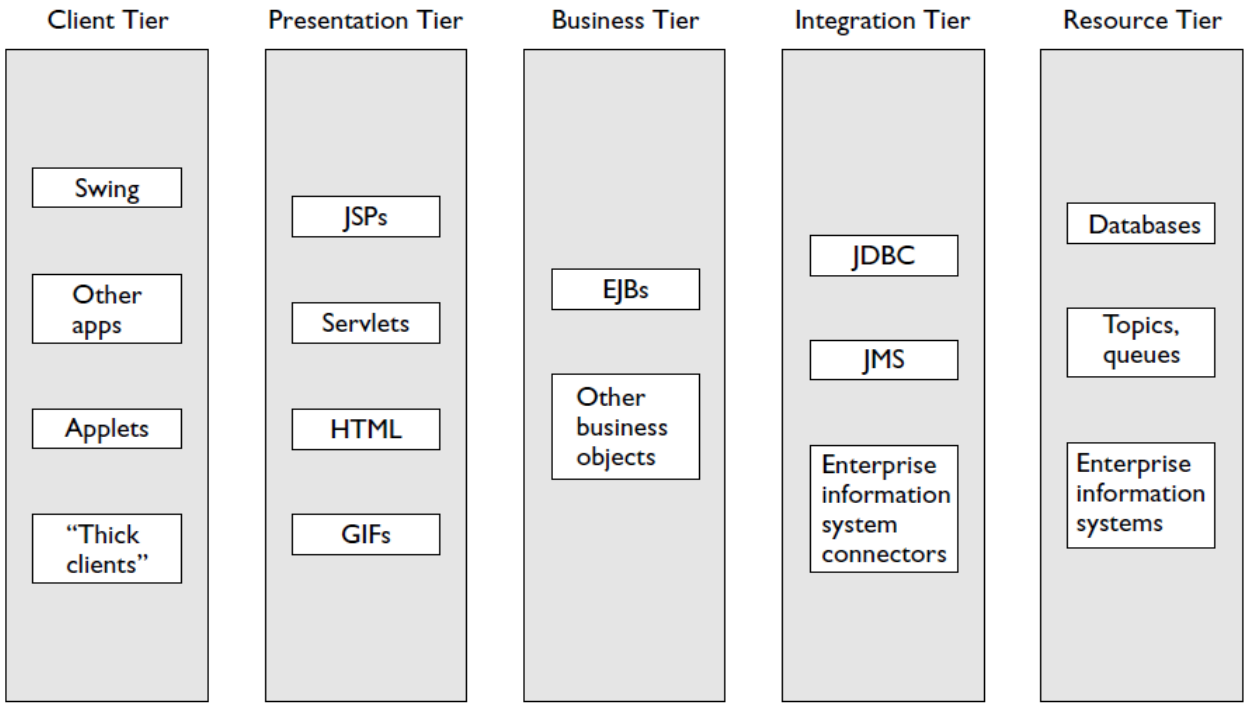
Design Patterns for SCWCD

1. What is a design pattern? When will you need it?
    a. A design pattern is like an architectural blueprint for solving some common problem, and J2EE applications exhibit a whole range of common problems that have been solved hundreds of times before. So when you encounter your own J2EE design issues, you may not have to think for yourself, but instead adopt an off the-shelf design pattern.
2. The features of design patterns are described according to a set of headings that act as a template for each pattern. Explain (in terms of the Core J2EE Patterns book).
    a. The Core J2EE Patterns book talks in terms of following headings:
        i. **Context**—where in the J2EE environment you are likely to encounter a need for the pattern.
        ii. **Problem**—an account of the design problem that a developer needs to solve.
        iii. **Forces**—motivations and justifi cations for adopting the pattern.
        iv. **Solution**—how the design works, both in structural terms (depicted with UML diagrams) and implementation terms (often backed up with example code).
        v. **Consequences**—what are the pros and cons when the pattern is applied.

    This set of headings is not standard between champions of design patterns.

3. Sun's J2EE patterns are based on a tiered approach to application development. What are those tiers?
    a. Tiers are:
        i. Client Tier
        ii. Presentation Tier
        iii. Business Tier
        iv. Integration tier
        v. Resource Tier
    b. Tiers are shown in below diagram:



4. What do you know about EJBs?
    a. Enterprise JavaBeans are designed to be self-contained business components. An Enterprise JavaBean is composed of a number of classes, some written by developers, some generated. However, a particular Enterprise JavaBean—maybe representing some business entity such as "Customer"—can be considered as a unit in its own right.
    b. EJBs run in an Enterprise JavaBean container.
    c. You can't make an EJB with the new keyword. The container supplies the life cycle as in the case of servlets. Whereas you fire an HTTP request to make use of a servlet in a servlet container, you execute JNDI (Java Naming and Directory Interface) code to get hold of the services an EJB can offer within its EJB container.
5. List down any six design patterns?
    a. **Intercepting Filter:** used to check or transform a request or a response.
    b. **Front Controller:** a "gateway" for all requests—can be used to check requests and control navigation centrally.

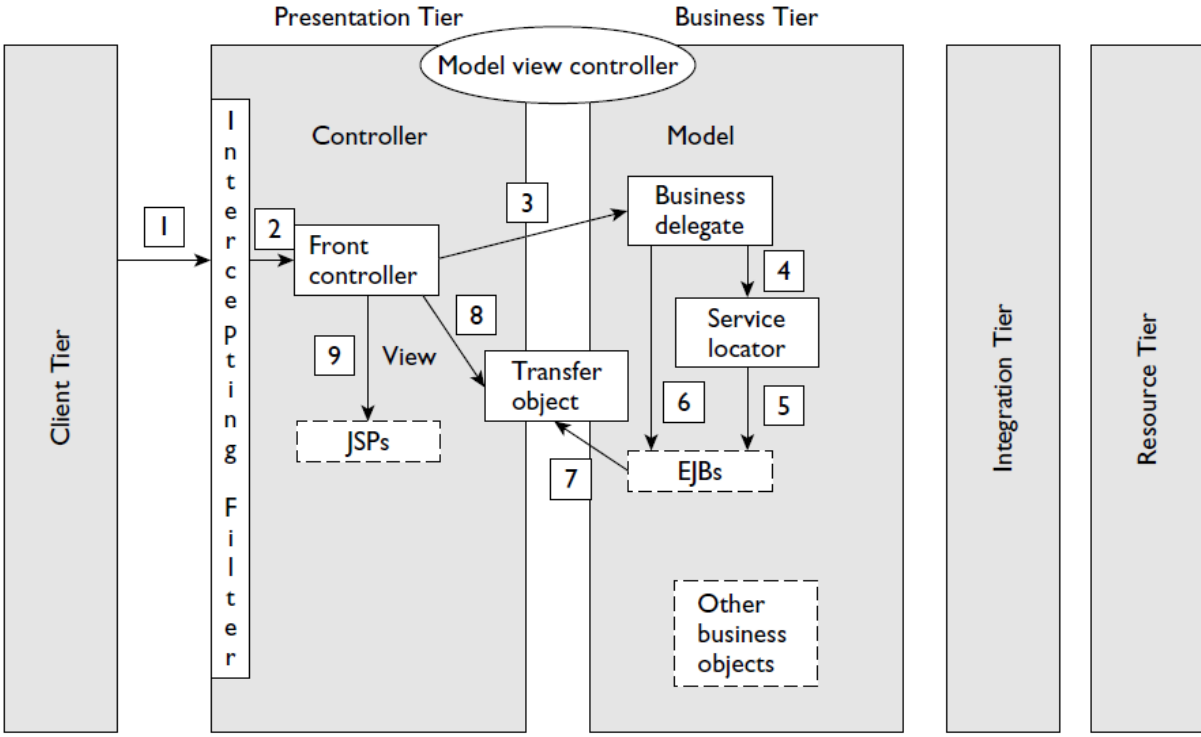
c. **Model View Controller:** used to keep presentation and business concerns independent. The model holds the business data, the view presents the data, and the controller mediates between the two.

d. **Business Delegate:** used to provide a friendly front end to business-level APIs (especially when those APIs involve Enterprise JavaBeans).

e. **Service Locator:** used to locate services (!), which usually means executing highly technical code to get back references to resources such as Enterprise JavaBeans and JMS queues.

f. **Transfer Object:** encapsulates the data returned from complex business objects (usually EJBs again).

6. Show how the above six design patterns fit into J2ee tier architecture?



a. A request is trapped by an Intercepting Filter (1)—if the filter allows it, the request passes on to a Front Controller (2). The Front Controller makes use of a Business Delegate to obtain business data (3). The Business Delegate uses a Service Locator (4) to find the business component it needs (5)— often an EJB. The Business Delegate requests data (6), which it gets back in the form of a Transfer Object (7), and returns this to the Front Controller. The Front Controller

takes what it needs from the Transfer Object (8), then forwards to an appropriate JSP (9) (which—in all likelihood—uses data originating from the Transfer Object). Model View Controller is the collection of components represented by the JSP (for the view), the Front Controller (evidently the controller!), and the Business Delegate (fronting the business model).

Intercepting Filter Pattern

7. How does an Intercepting Filter Pattern work?
    a. A filter manager intercepts a request on its way to a target resource. From the content of the request (usually from its URL pattern), the filter manager calls on the help of a filter chain object, which creates the filter objects needful to the request (if they haven't been created already). The filter chain also dictates the correct sequence of filters to apply. The filter chain object infers the appropriate sequence from some information in the request: As with the J2EE filter mechanism, the basis for this is the request's URL pattern.
8. Can we write a customized Intercepting Filter and use some other criteria altogether for the selection of filters in the chain than URL pattern?
    a. If you write code for a customized Intercepting Filter, you can use some other criteria altogether for the selection of filters in the chain. It is just that in this case, the J2EE pattern has to be implemented in your application server because it's a mandatory part of the servlet specification—and the servlet specification goes with URL matching (and servlet naming) as the basis for filter selection.
9. How will you decide whether some or other functional requirement should be placed in a filter?
    a. Two easy tests are:
        i. Is the requirement discrete? Whether or not the filter can perform its work independently of other considerations. If other filters are placed before or after, will the current filter remain unaffected? If yes, we can use filters.
        ii. Is it common to more than one type of request (and/or response) from the web application? If it's executing logic that is specific to one particular

resource, then perhaps the code belongs with that particular resource, not in a filter. Filters are meant to be more general purpose.

10. List down few scenarios where you will use a filter?
    a. You might want to do some processing on the request before reaching the targeted resource:
        i. You want to check something about the origin of the request.
        ii. You might want to detect how the request is encoded, and maybe re-encode it before it reaches the core of your application.
    b. You might want to do some processing on the completed response before returning it to the requester:
        i. Translate the response in some way—to make it more intelligible to another computer system or to a human.
        ii. Tag your response in some general way—perhaps adding a copyright notice.
11. What are the drawbacks in dedicating a servlet as a "gateway" to your application instead of using a filter?
    a. You have to design the approach yourself for using a servlet as a "gateway", but might be the same thing you can do with filters with much ease.
    b. It's easy to overburden your "gateway" servlet with too many responsibilities.
    c. Depending on your design, you'll need to change servlet code to add new filter logic or alter the sequence of the filter chain.
12. List down the benefits of using the Intercepting Filter pattern?
    a. Control is centralized for chosen activities (like encryption) but in a loosely coupled way (the encryption happens independently of anything else going on around it).
    b. Filters promote the reuse of code. It's easy to plug a filter in where you want it.
    c. Including a filter doesn't involve recompilation: Everything is controlled from the deployment descriptor, declaratively. You can juggle your filter order without recourse to javac (though dependent on your application server, you might have to remake and redeploy your WAR file).

13. Give any disadvantage of using filters?
    a. Information sharing between filters is likely to be inefficient. You may well need to reprocess an entire request or entire response all over again in each filter in the chain.

Front Controller Pattern

14. What is Front Controller pattern?
    a. The Front Controller acts as a centralized point of access for requests. Front Controller is a presentation tier pattern. The pattern works by having an object (the Controller object) as the initial point of contact for requests to a web application. The controller may validate the request according to some criteria. It may also extract information from the request that determines which page to navigate to next. As a gateway, it might control access through authentication and authorization rules. It may delegate to business processes. Almost invariably, it plays a crucial role in controlling (screen) flow through an application.
15. Does the framework Struts uses Front Controller pattern? Explain.
    a. Yes. Struts is an example of a Front Controller. The URL mapping *.do maps to a master servlet (acting as a Front Controller). Dependent on what is before the .do, the master servlet delegates to a class of type Action.
16. List down few benefits of using Front Controller pattern?
    a. Central control of navigation.
        i. Instead of allowing one JSP to directly link to another, you plant logical links in your JSPs instead—sending you via the Front Controller, which makes the ultimate decision about forwarding to the next JSP.
    b. Central control of requests.
        i. You might want this control so that you can easily track or log requests.
    c. Better management of security.
        i. With centralized access, you can cut off illegal requests in one place. Authentication and authorization can also be centralized
    d. To avoid embedding control code within lots of separate resources.
        i. This is an approach that easily leads to a copy-and-paste mentality.

17. Give any two drawback for the Front Controller pattern?
    a. The Front Controller pattern can lead to a single point of failure. However you can always mitigate this by distributing your application, as long as your application server supports this.
    b. It is very easy to overload a servlet acting as a Front Controller with too many responsibilities and too much code. However we can ensure that Front Controller, wherever necessary, delegates to appropriate helper classes.
18. Which design pattern will be the best for the below scenarios (Front Controller or Intercepting Filter pattern)?
    a. Controlling the flow of navigation from one page to another
        i. Front Controller
    b. Converting responses from one form of XML to another
        i. Intercepting Filter (as long as there is a general way of translating the XML for many different responses)
    c. Zipping up the response
        i. Intercepting Filter
    d. Executing different blocks of business logic dependent on a parameter in the request
        i. Front Controller (dispatching to other classes that contain the business logic)
    e. Stopping a request dead in its tracks if your application dislikes its encoding.
        i. Intercepting Filter
    f. Enforcing J2EE authentication and authorization.
        i. Neither Intercepting Filter nor Front Controller. You want to protect individual components declaratively in the deployment descriptor as far as possible.
    g. Enforcing custom authentication or authorization rules.
        i. Depends: could be Intercepting Filter or Front Controller. Use Intercepting Filter if these are “blanket” rules, applying to all (or most) resources. If authentication and authorization needs to be closer to business logic, use Front Controller.

Model View Controller

19. What is the basic idea of Model View Controller pattern?
    a. The main idea of MVC is to separate your presentation logic (**V**) from your business logic (**M**). The idea is to have a "controller" (**C**) component that mediates between the two.
20. What are the major responsibilities of View component in MVC?
    a. To detect each user action and transmit it to the controller component.
    b. To receive information back from the controller and organize this appropriately for the recipient.
21. What are the responsibilities of the controller in MVC?
    a. To supply the view with the information it needs—usually by obtaining information from the underlying model.
    b. To interpret user actions and take appropriate actions—usually updating the model and switching view (or both).
    c. To receive changes from the model that come from other sources, and update views as appropriate.
22. What are the model's responsibilities in MVC?
    a. To look after the underlying data. It must respond to requests for information and update from the controller. It must inform the controller of external changes to the data that didn't arrive through the controller.
23. List down few reasons for using the MVC pattern?
    a. You want to reduce dependencies between the view code and the business model code. That way,
        i. You can graft on additional views much more easily (so you might have a web client and a Swing client showing alternative views of the same model data).
        ii. You can change the view without affecting the business model, and vice versa.
        iii. The controller is the only volatile component that might be affected by changes at either end.

b. You are using a framework that has MVC built into it. Struts is a very popular framework. If you are in a development team that uses it, you are forced into a set of programming standards that abide by MVC rules.
c. You want the maintenance benefit that comes from separating the layers. Expert Java programmers can concentrate on the controller and model ends of the application. Web designers can concentrate on the JSP view side of things.

24. Many J2EE architects would avoid using the JSTL SQL components altogether. Why?
    a. Many J2EE architects would avoid using the JSTL SQL components altogether, just because they fundamentally violate MVC rules. Because the JavaServer Page is normally the view, and the database is ultimately a model, we are not having a controller in between when we use JSTL SQL components from an SQL page.
25. Give any disadvantages of using MVC?
    a. MVC increases the number of moving parts in your application. There is more overall complexity.

Business Delegate Pattern

26. Give the basic idea of Business Delegate Pattern?
    a. The Business Delegate pattern works by placing a layer in between a client (possibly a Front Controller object) and a business object. That way, even when business object interfaces change, it may well be possible to absorb the change within the business delegate object—without changing the interface that this presents to its clients. The other service a Business Delegate typically provides is encapsulation of all the ugly details that might be involved in getting hold of the business object in the first place, and coping with any errors thrown up.
27. List down the major uses of Business Delegate Pattern in J2EE?
    a. It is used to:
        i. Encapsulate EJB (and other complex business object access) code.
        ii. Encapsulate JNDI code (but often defers to the Service Locator pattern for that).
        iii. Present simple business interfaces to presentation-tier clients.
        iv. Cache (sometimes) the results of expensive calls on business objects.

28. Which layer does the Business Delegate belong to?
    a. In physical terms, the classes involved are likely to remain close to your presentation code (the EJB server itself is potentially remote). Perhaps for this reason, it's described as a client-side business abstraction and often regarded as belonging to the (physical) presentation tier.
29. List down for reasons for using the Business Delegate pattern?
    a. To reduce coupling between clients in the presentation tier and actual business services, by hiding the way the underlying business service is implemented.
    b. To cache the results from business services.
    c. To reduce network traffic between a client and a remote business service.
    d. To minimize error handling code (particularly network error handling code) in the presentation tier.
    e. Substituting application-level (user-friendly) errors for highly technical ones.
    f. If at first the business service doesn't succeed, the Business Delegate class might choose to retry or to implement some alternative API call to recover a situation. A business service failure doesn't immediately have to be passed on to a client.
    g. Make naming and lookup activities happen within the Business Delegate—or code that the Business Delegate uses (see Service Locator).
    h. To act as an adapter between two systems (B2B-type communication—where a visible GUI isn't involved). The delegate might interpret incoming XML as a business API call.

Service Locator Pattern

30. What is the basic idea of Service Locator Pattern?
    a. Service Locator allow a Business Delegate class to further offload the aspects of Business Service access code, which normally involves looking up the business service in the first place and getting some kind of reference to it. Thereafter, the Business Delegate can happily call APIs on the business object. So Service Locator encapsulates the mysteries of finding business objects (services).
31. Is it valid to include "service locator" code in methods within your "business delegate" class?

a. Service Locator is like a subset of Business Delegate—you might even include "service locator" code in methods within your "business delegate" class. It's a valid implementation of the pattern. However, the separation of different classes is the preferred way.

32. Give the reasons for using the Service Locator pattern?
    a. To hide JNDI (or similar) code that gets hold of a reference to a service. Even though JNDI code is pure, portable Java, the parameters fed into JNDI code can be vendor-specific, so locating all these details in one place (rather than leaking them all over your business objects) is desirable if you want to port your application later with minimum hassle.
    b. By locating JNDI (or similar) code in one place—or a few places—you avoid duplication of code that gets references to business resources (services) you need to use across many business objects.
    c. To minimize network calls—the Service Locator can decide which JNDI calls need to be made and when it has appropriate references to remote objects already. This can, of course, improve performance. The operation of looking up some resource in a JNDI-compliant naming and directory service is expensive.

Transfer Object Pattern

33. Give the basic idea of Transfer Object Pattern?
    a. The idea of the Transfer Object pattern is to encapsulate and hold data—usually the data that would be available inside a more complex (and possibly remote) object, such as an Enterprise JavaBean (EJB). Structurally, the object representing a Transfer Object is very simple. It's a Java Bean—not an Enterprise JavaBean—just a bean. Deal with the EJB business object directly can be expensive in networking and processing terms. A Transfer Object can be used to get all required data together and updating the data back in a similar way. Classes for Transfer Objects are always implement Serializable so that they can be returned from remote method calls.
34. Transfer Object design pattern was formerly known as …….
    a. Value Object

35. Give some reasons for using Transfer Object Pattern?
    a. Transfer Objects simplify your life on the client side. You avoid direct dependence on potentially complex business objects and deal instead with relatively simple bean-like objects.
    b. You avoid the expense of repeated interaction with remote business objects. This occurs when you accumulate dribs and drabs of data from those business objects by a series of expensive remote calls. Better to pass all the data you are likely to need in one remote "hit." This may seem profligate if you never use some of these attributes, but this is likely to be a small overhead compared even to one unnecessary network call.
36. Give some drawbacks to using the Transfer Object pattern?
    a. A Transfer Object is likely to duplicate code—notably in the attributes and getter and setter methods that shadow those of its associated Enterprise JavaBean (or other business component). You could consider coupling the Transfer Object and business component by having the business component extend the Transfer Object and so inheriting its attributes and getter and setter methods. Alternatively, you might have a tool available to generate Transfer Object code directly from your business component.
    b. Transfer Objects can also go stale. Vendors of EJB containers go to great lengths to ensure that EJBs keep up to date with changes from multiple users of the EJB. You may happily update values in the Transfer Object you are using. Yet in the meantime, someone may have sneaked up and updated the underlying business component. To protect against this, you may introduce complexity into your code—by adding some form of version control or locking to your system. When you deal directly with EJBs, you could end up attempting to solve problems that EJB container designers have already solved.