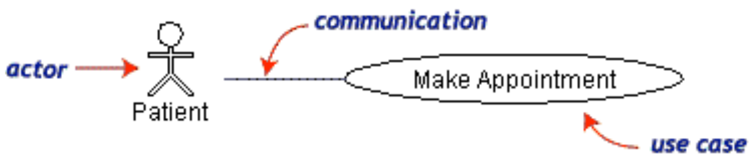
Pete Morgan's Sun Certified Enterprise Architect Part 1 Notes

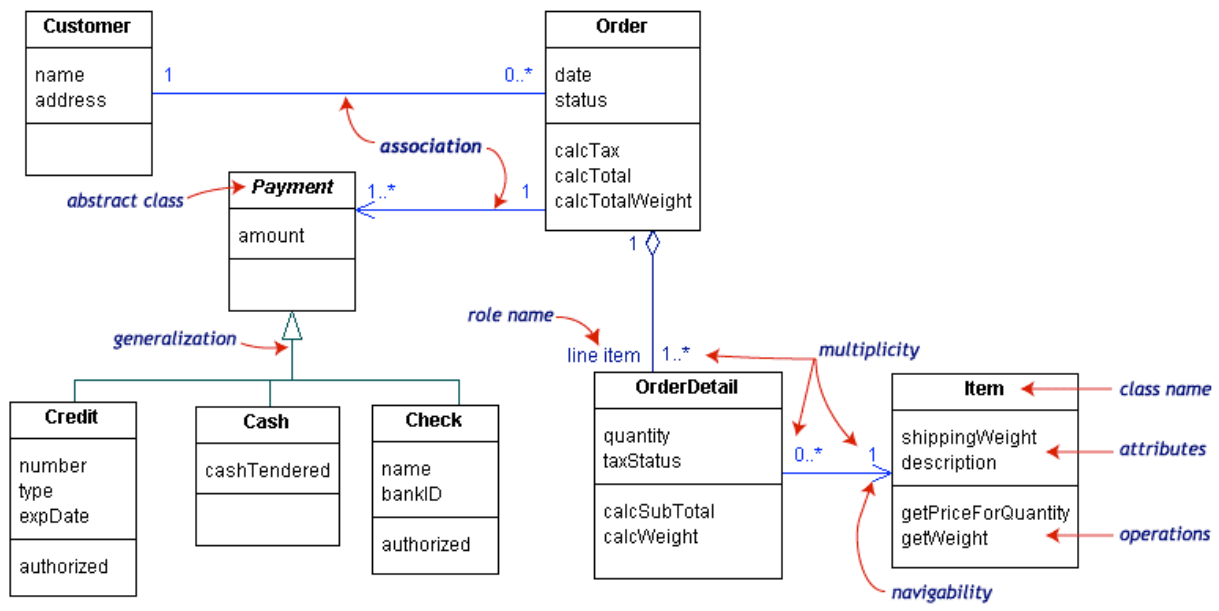
Concepts

Draw UML Diagrams
Interpret UML diagrams.
State the effect of encapsulation, inheritance, and use of interfaces on architectural characteristics.

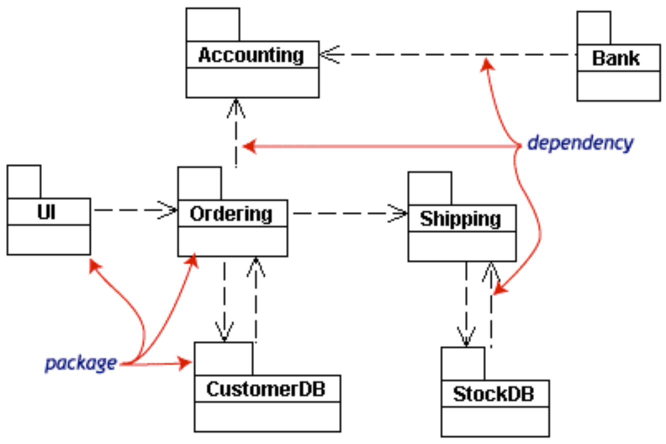
Use Case Diagrams

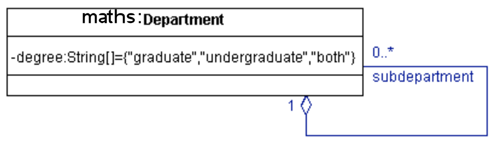


Class Diagrams



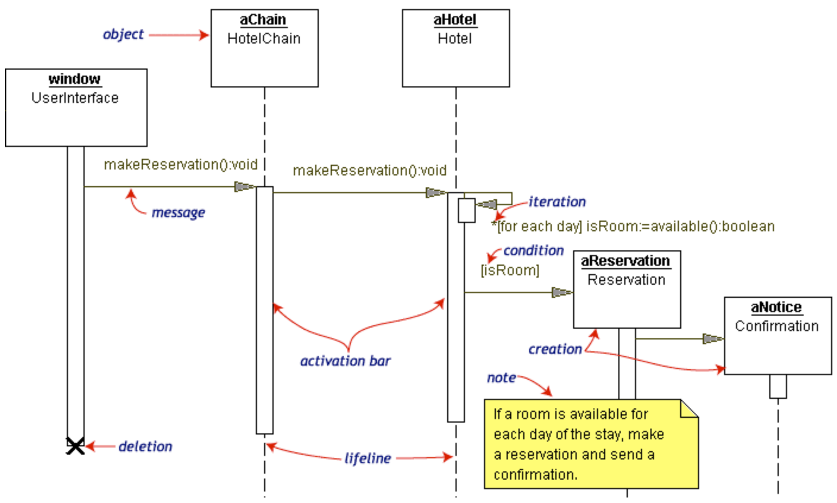
Package and Object Diagrams



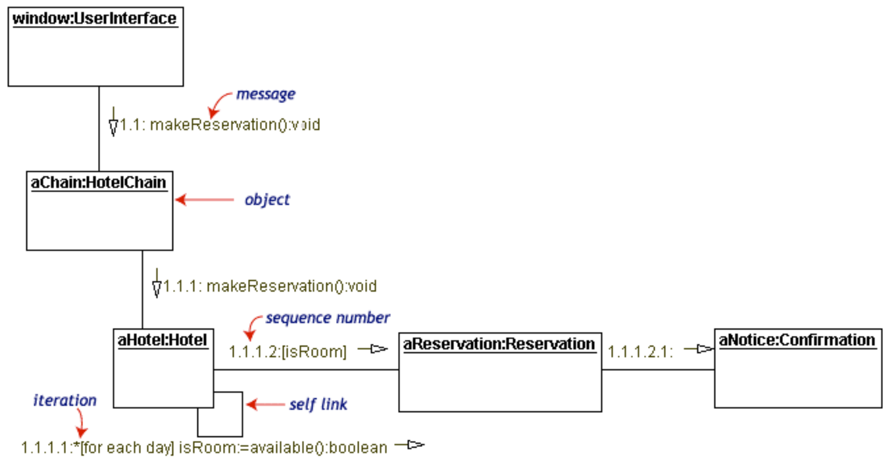


+ = public, # = protected, - = private

Sequence Diagram

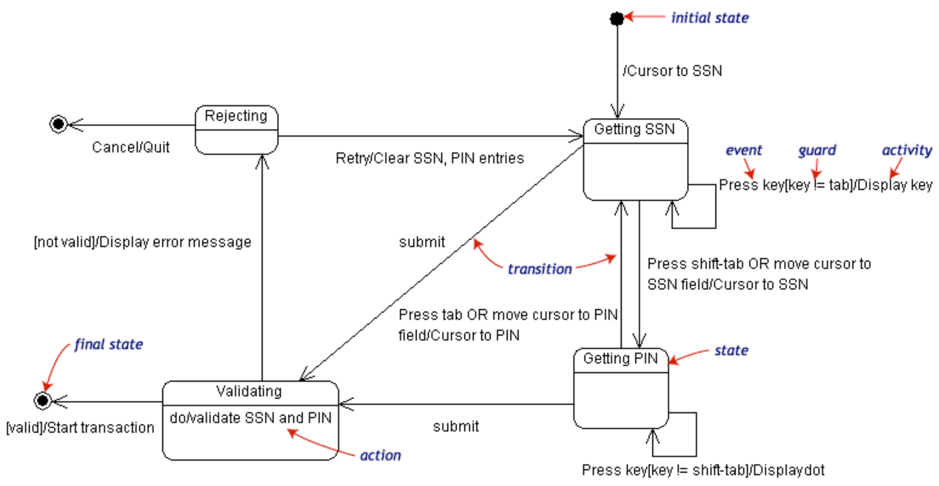


Colaboration Diagram



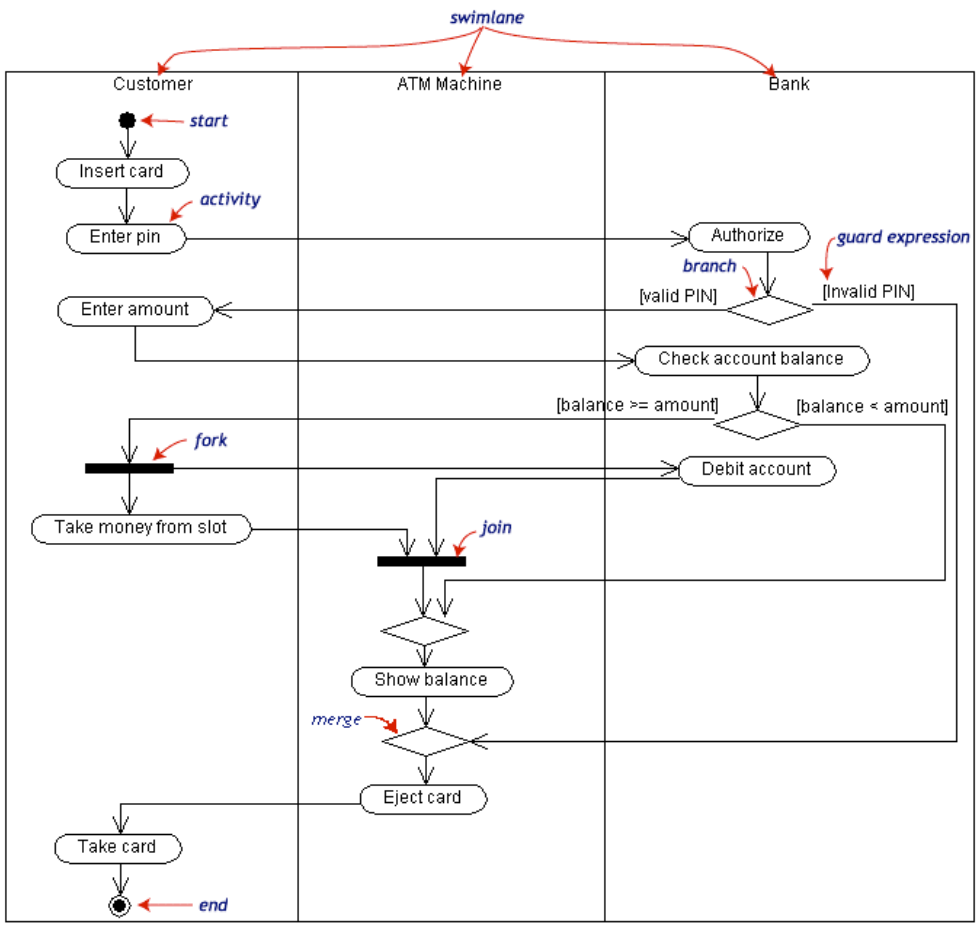
Note: if class name is ':ClassA' then it is an anonymous instance. If class is 'x:ClassA' this is instance x of ClassA.

Statechart (State Transition) Diagrams

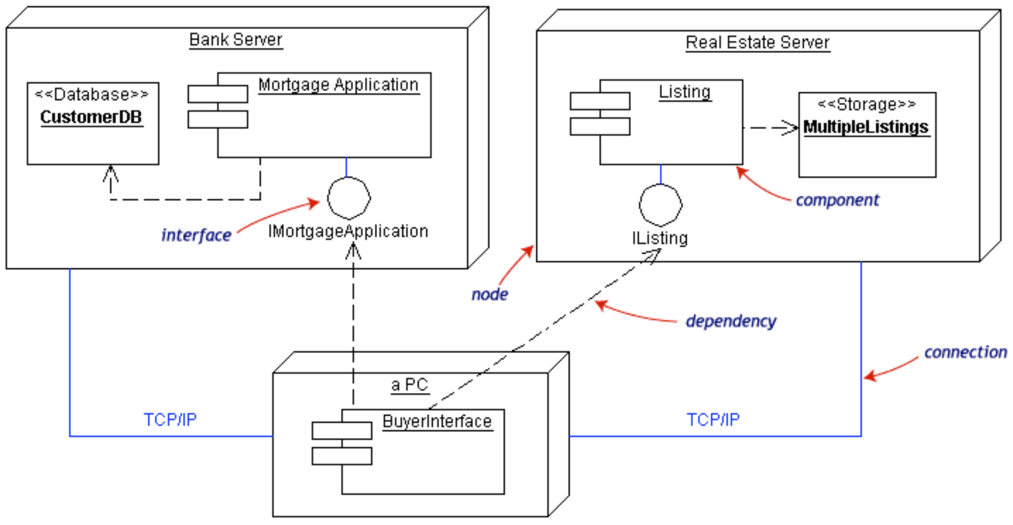


initial state
/Cursor to SSN
Rejecting
Cancel/Quit
Retry/Clear SSN, PIN entries
Getting SSN
event
guard
activity
Press key[key != tab]/Display key
[not valid]/Display error message
submit
transition
Press shift-tab OR move cursor to SSN field/Cursor to SSN
Press tab OR move cursor to PIN field/Cursor to PIN
final state
Getting PIN
state
Validating
do/validate SSN and PIN
[valid]/Start transaction
submit
action
Press key[key != shift-tab]/Display dot

Activity Diagrams



Component and Deployment Diagrams



Note: The components diagram is shown inside the physical hardware which is made up of nodes. The overall diagram in the example is a deployment diagram. So it is two for the price of one.

What isn't Generally Realised

- Abstract Class is Generalised
- Interface is Realised.
- So that's, A->G and I->R, it's alphabetical!

UML extension

- Stereotypes - << new thing >>, eg: <<Servlet>> on a class.
- Tagged values – new element specific info?
- Constraints – {constraint name}, eg: {subject} on an association

OO Concepts

- Encapsulation - hide how things are done behind an interface
- Inheritance - 'is a' relationship
- Use of Interfaces - contract between two classes, no direct concrete to concrete class relationship and therefore more flexible, extendable and less coupling, Liskov Substitution Principal.

Recognise the effect on each of the following characteristics of two tier, three tier and multi-tier architectures: scalability maintainability, reliability, availability, extensibility, performance, manageability, and security.
Recognise the effect of each of the following characteristics on J2EE technology: scalability maintainability, reliability, availability, extensibility, performance, manageability, and security.
Given an architecture described in terms of network layout, list benefits and potential weaknesses associated with it.

Scalability

- As load increases, increased support is required to keep QoS (Quality of Service).
- The more scalable the system, the greater capacity supported.
- Vertical Scaling: add more memory, disk space, CPU, etc.
    - J2EE – you can manage more EJBs, JSPs, etc. with more memory.
    - Pros: cheap and easy to manage, few/no changes to source code.
    - Cons: Single point of failure therefore less reliable and available.
- Horizontal Scaling: add servers, cluster.
    - J2EE – support for distribution, clustering and load balancing.
    - Pros: reliable, available, capacity and performance.
    - Cons: pricey, harder to manage than Vertical, more complex.

Availability

- Service/resource is always accessible.
- Obtained through horizontal and vertical scaling
- Planning for failure by building in redundancy.

Reliability

- Integrity and consistency of application and transactions.
- As load increases the system should still accurately process requests.
- Poor reliability means poor scalability.

Maintainability

- To correct flaws in the existing services without impacting other components of the system.
- To enhance the maintainability of a system: low coupling, modularity, and documentation.
- Modularity achieved with MVC and other patterns.

Flexibility

- Ability to change to meet new requirements in a cost-effective way.
- Improves availability, reliability and scaling.
- Reduces performance and manageability.

Extensibility

- Adding/modifying functionality without effecting the existing services.
- Easier when system has low coupling, interfaces and encapsulation.
- J2EE supports componentisation, MVC, etc.

Performance

- Speedy, lack of bottlenecks.

- J2EE pools resources, connections, etc.
- Usually measured in response time per user or transaction throughput.

Manageability

- Manage and ensure health of system.
- Monitor QOS and dynamically change things to improve it.
- Reduced by horizontal scaling and increased security.

Security

- Privacy – info not disclosed or modified.
- Higher security means higher cost, less performance and more complex.
- Principals are:
  - Identify – authentication
  - Authority – authorization
  - Integrity – modify data only in allowed ways
  - Privacy – disclose in authorized ways to authorized people/systems.
  - Audit-ability – log actions.
- Use VPNs, Firewalls, SSL, signed applets.
- Denial of Service attacks – attacks that effect availability.

Distinguish appropriate from inappropriate techniques for providing access to a legacy system from Java code given an outline description of that legacy system

Tier Comparison

What	***1-Tier*** *One big application with presentation, business logic and data access all in it.*	***2-Tier*** *"Fat Client". Tier 1:Presentation/Business and Data Model. Tier2: Database and may also contain business logic if use stored procedures. Separate presentation from data access rules so less coupling that 1-Tier.*	***N-Tier*** *"Thin Client". Client Tier, render for presentation (eg; Browser, Applet, Java Swing/AWT). Presentation Tier, generate user interface (eg: JSPs, Servlets, Java Beans), Business Tier, business and data objects (eg: Session and Entity Beans), Data Tier, Data source (eg: RDBMS).*
Scalable	Not really	Hard to scale (remote application deployment).	Horizontal and vertical both possible, resource pooling, etc
Maintainable	Hard to maintain (control evolution of product), could use Web Start.	Okay to develop but business and presentation mixed, but hard to deploy (could use Web Start) and remotely administration/ monitor/ trouble shoot.	Harder to maintain applications as more complex, but easier to maintain QoS, and maintaining layered code is easier.
Reliable	Single point of failure but data consistency easy	Database is single point of failure.	Can have redundancy, fault tolerance, fail-over, etc
Available	A single point of failure, but a failure will only effect one client.	Database is single point of failure that would effect multiple clients.	24x7 with fault tolerance
Extensible	Hard to manage QOS as load increases.	Hard to re-deploy because remote – fundamentally hard and expensive to deploy and manage distributed software. Also, presentation and business logic mixed makes it harder.	Loose coupling, components allow for modification and extension without effecting existing code due to the layering
Performance	Okay but hard to manage QOS as load increases	Bad, connection to database a bottleneck. No pooling across all clients so each need at least one connection, therefore high network bandwidth	Pool resources, load-balanced servers (scalable horizontally), but more network traffic.
Manageable	Easy to manage app. but hard to manage QOS of app. when load increases.	Single point of failure and hard to re-deploy because remote	Need to monitor the web container (servlet, HTTPSession, etc.) and EJB container (pools, threads, transactions, queues, etc.) and physical connections between the tiers. However, easier to manage QOS
Security	Okay as all in one place	Better than 1 Tier, could have some authentication and audit in database tier. However, client has the control so problematic.	Can be applied at each tier but complex to implement. Can use Firewalls at each layer

Legacy Connectivity - Java Techniques

- **JMS -** Asynchronous, message-based communication with legacy systems and services. Loose coupling to other system; sender and receiver's only contract is the format of the message.

- **JDBC** – access a legacy system's data source directly. If no driver could use the JDBC-ODBC bridge. Advantage: easy and quick to do. Disadvantage: increased data coupling between apps, no access to business rules, data might not be in schema ideal for new system. The schema becomes the contract between the new and old system, it is really hard to extend the functionality if one needs to alter the schema.

- **JNI** – Wrap C-APIs to access legacy systems and services. Should be only as a stop gap.

- **Java IDL** – Support for calling CORBA from Java (could be from Servlets, EJBs or Java classes). Synchronous, peer-to-peer access of legacy systems and services.

- **Middle-ware** with Java on one side and Legacy API on the other.

- **Applet or application** that communicates with legacy system using sockets (may be signed and trusted if need to access protected resources) or to COM and CORBA using Microsoft bridge or Java IDL. Better for manageability to use an Applet because downloaded but slower and cumbersome (no integration with web site's look & feel (CSS) and slow byte code verification).

- **JCA** – an API that is focused on connection management, transaction management, and security to legacy apps. Tight coupling to legacy API but through interfaces.

Legacy Connectivity - Non-Java Techniques

- **Screen scrapers** - A Screen Scraper is an application that translates an existing client interface into a set of objects (offer an Object API to the legacy system). They are particularly useful when the client interface is tightly coupled to the other tiers of the system and also used if the legacy system does not have a published interface or the documentation has been lost. Screen scrapers usually function as a terminal emulator on one end and an object interface on the other. Advantages; low-level object-based interface to the legacy app; allows you to build a new GUI over the existing client interface; can be implemented without any disruption to existing legacy applications, and also the training requirements for the new system should be minimal (eh?) Disadvantages; any changes to the legacy interface can break the new GUI – very tightly coupled; prone to causing errors in the new GUI because of unexpected outputs from the legacy interface; prone to causing the new GUI to "freeze" when the legacy interface is expecting input that the screen scraper in unaware of; slow, and the application cannot be modified to allow additional functionality.

- **Object Mapper Tool (Object Wrappers)** – Object mapping tools can be used if you choose to ignore the existing legacy interface and access the underlying tiers directly. These tools are used to create proxy objects that access legacy system functions and make them available in an object-oriented form. You could wrap up an existing non object-oriented system and then use CORBA to communicate with it, or alternatively to use any published interface to the legacy system (for example, the legacy system might support TCP sockets). Advantages; more effective than screen scrappers because they are not dependent on the format generated by the existing legacy interface; object based access to the Legacy System Disadvantages; you have to write the wrapper code your self.

- **SOAP** - Could wrap and expose a SOAP API for connectivity to Java and .NET. However, way easier if the service methods are stateless and do not need to collaborate in a transaction (transactions over SOAP are possible but there aren't many implementations). There's a number of implementations that allow web services to be created from Java classes (that could be wrapping a legacy system). Some are quite low level and require a high degree of programming (eg: Apache AXIS), other's are higher and allow faster development but with a courser level of control (eg: The Mind Electric's GLUE).

- **Off Board Server** - A server that executes as a proxy for a legacy system. It communicates with the legacy system using the custom protocols supported by the legacy system. It communicates with external applications using industry-standard protocols. Kind of an Adapter pattern really.

List the required classes/interfaces that must be provided for an EJB.
Distinguish stateful and stateless Session beans.
Distinguish Session and Entity beans.
Recognise appropriate uses for Entity, Stateful Session, and Stateless Session beans.
State benefits and costs of Container Managed Persistence.
State the transactional behaviour in a given scenario for an enterprise bean method with a specified transactional deployment descriptor.
Given a requirement specification detailing security and flexibility needs, identify architectures that would fulfil those requirements.
Identify costs and benefits of using an intermediate data-access object between an entity bean and the data resource.

What is an EJB

A Server side component that encapsulates business logic. There are three types, Session, Entity and Message-Driven. The benefits are:

- A container that provide system level services so the developer can focus on the business problem. Eg: Distributed Transactional support, security, resource pooling, persistence, etc.
- Enforces a good separation of presentation, business logic and data storage.
- Portable to other J2EE vendors so no vendor lock-in.
- Scaleable – vertically and horizontally.

Session Bean

Represents the client inside the J2EE server, performs work for the client, shielding the client from the complexity of the business tasks. There are two types; stateful – where the client interacts with the been and the bean maintains the 'conversation' from method call to method call; and stateless – where the client interacts with the bean but it does not maintain a conversation. Stateless session beans support multiple clients and offer better scaleability. Also, under load stateful beans are swapped out to secondary storage, this obviously has a performance cost. Session Beans are suitable when:

a. The state of the bean is not persistent, existing only for a short time.
b. You wish to mediate between the client and other components, or manage the workflow of other components on behalf of the client - a façade.
c. Stateful – the bean state represents the interaction between the client and server, the bean needs to hold information about the client across method invocations. Eg: shopping cart. Note: they can't be re-entrant – will throw exception.
d. Stateless - there is no client to server state, the methods are generic for all clients – the method is not tied to things that have occurred in the other method conversations. Often used for fetching large amounts of read-only data rather than creating lots of Entity EJBs, and acting as a façade to other beans.

Entity Bean

An entity bean represents a business object in a persistent storage medium. They are persistent, allows <u>shared access</u> and have primary keys (that implement java.io.serializable, have public constructor, have public attributes, and redefine equals and hashCode). They can collaborate in distributed transactions. There are two types of Entity Bean; Bean-Managed and Container-Managed. Container-Managed : there is no SQL in the code, the container performs the database access, they are therefore more flexible for porting to other database vendors, the mapping of attributes to the database schema is done in the deployment descriptors using EJB QL. Bean-Managed: the developer codes the object to database mapping. Both are suitable when:

- The bean represents a business entity not a procedure.
- The bean's state must be persistent beyond that of the app server.

Message-Driven Bean

An enterprise bean that allows the processing of asynchronous messages. It acts as a JMS message listener. Messages come from any JMS producer/sender. Currently only consume JMS messages but other kinds in 2.1 spec. They resemble stateless session beans in that they are stateless and they may be pooled. A single message driven bean can process messages from multiple clients. There is an 'onMessage' method that is called when a message arrives. Also, the method is called within a distributed transaction. The are suitable when:

- You need to consume asynchronous messages from within an EJB container.

EJB Interfaces and Classes

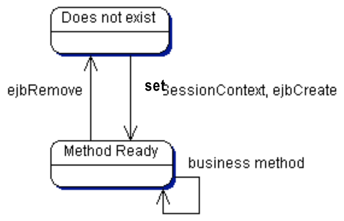
Remote Access – may run EJB in different machine or different JVM to client, however to the client the remoteness is transparent (proxy). Needs a remote interface, home interface and bean implementation of interfaces. Remote interface is for the business methods on the bean, the home interface is a factory for creating EJBs and for the life cycle, and in the case of Entity beans the finder methods. Also, business methods on Home are for methods that are involved on all instances of an Entity Bean class – like a static (but not static). Parameters and returned values are 'pass by value' – copies.

Local Access – runs in the same JVM as the client, to the client the location is not transparent, it is often an entity bean. There needs to be a local interface, a local home interface and a bean implementation of the interfaces. If you define a container managed entity bean with a relationship to another entity bean the relationship, it must be local. The participating local entity beans must be in the same EJB Jar file. The benefit is increased performance. Tightly coupled EJBs should have a local relationship however if you need to be able to scale and distribute requests across the cluster the calls should be remote. Parameters and returned values are 'pass by reference' – like normal Java calls.

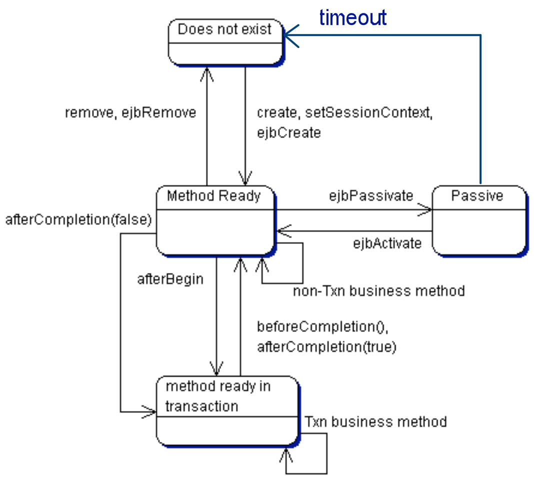
Note: Remote and Local interfaces are not used by Message-Driven beans

Life Cycles

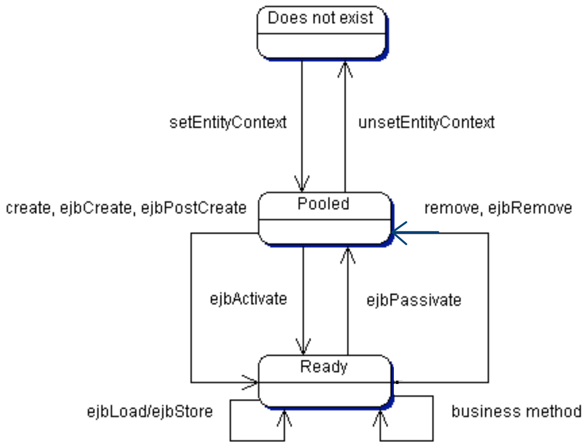
Stateless Session Bean



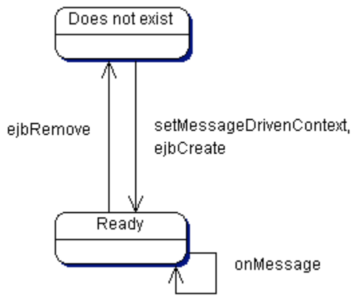
Stateful Session Bean



Entity Bean



Message-Driven Bean



Coding Rules

Session Bean – Stateless

File	*Rules*
Bean Class	• Implement the javax.ejb.SessionBean interface. • Class must be defined to be public, not abstract or final. • Have one public, non-static/final, no arg ejbCreate that returns void. • Have empty implementations of ejbActivate and ejbPassivate – they are there just to make it use the same I/F as stateful session beans. • The container calls ejbCreate and ejbRemove when it wants to (not in response to a create or remove on the Remote I/F), often it will pool the requests and simply re-use an already existing bean for another client. • Have a public default constructor. • Not use the finalize method. • It can get 'this' pointer as an EJBObject via context.getEJBObject(). • No finder methods.
Home Interface	• The Home Interface must extend javax.ejb.EJBHome. • Have a similar 'create' method to the 'ejbCreate' method mentioned above, except it returns the remote interface, has the same parameters and throws java.rmi.RemoteException and javax.ejb.CreateException. • The Home acts like a factory creating instances of the EJBs. • Stateless session beans do not have find methods on the home because stateless beans are all equivalent and are not persistent.
Remote Interface	• The Remote Interface must extend javax.ejb.EJBObject. • Have the business methods with the same signature as in the EJB class. • Any exceptions must be declared as well as java.rmi.RemoteException for each method. • The implementation of the Remote I/F acts like an proxy to EJB object. • Calling create and remove does not call ejbCreate and ejbRemove on the bean, instead it simply gets a reference to an EJBObject from the pool and invalidates the reference (put back in pool).

Session Bean – Stateful (as Stateless except..)

File	*Rules*
Bean	• May have multiple create methods but still returning void. • Can implement that SessionSynchronization interface and provide the session life cycle methods (afterBegin(),beforeCompetion() and afterCompletion(boolean)). • Have proper implementations of ejbActivate and ejbPassivate to perform work before and after writing and reading from secondary storage. It swaps out all non-transient attributes of the class. • ejbRemove is invoked when the client calls remove on the remote I/F • In ejbRemove do the same kind of clean up as in ejbPassivate.
Home	• Have a similar 'create' methods to the 'ejbCreate' methods mentioned above, except they return the remote interface, have the same parameters and throws java.rmi.RemoteException and javax.ejb.CreateException.
Remote	*{Same as Stateful really}*

Entity Bean – Bean Managed Persistence

File	*Rules*
Bean	• Must implement javax.ejb.EntityBean. • It must have at least one public ejbCreate (that returns a primary key object) and ejbPostCreate (that return void) methods • The other life-cycle methods • No finalize • A non-abstract and non-final default constructor. • Must have an ejbRemove method that normally deletes an entry from the database. Able to throw a javax.ejb.RemoveExpcetion as well the runtime exception javax.ejb.EJBException for system errors. • Must have an ejbLoad and ejbStore methods. These are used to synchronise the entity with the database, loading and storing respectively. • Must have setEntityContext() and unsetEntityContext() that are called when the object is newly created and about to be destroyed to/from the pool. • At least one finder method to locate entities, the ejbFindByPrimaryKey(…) throws FinderException. It can have other 'finder' methods (starting with prefix 'ejbFind'). They return a primary key object or collection of primary key objects. If none found it must return an empty collection, not null. If finding an single instance and it cannot be found throw an ObjectNotFoundException. • The concrete bean must can have 'home' business methods that start with the naming convention 'ejbHome'. They are on all instances of the class but aren't in fact static (performed on the bean in the pooled state!). They must not access the object's attributes. They must **not** throw RemoteException? • For entity beans the ejbPassivate method notifies the entity bean that it is being disassociated with a particular entity prior to reuse or for dereferencing and possible garbage collection. Unlike stateful session beans the state does not need to be stored in secondary storage as by definition it already exists in permanent storage.
Home	• The home interface must extend EJBHome as in a session bean. • They have at least one create method (following the same rules as session beans – naming convention). • Business methods (less the ejbHome prefix). • Finder methods (less the ejb prefix). • Every method must throw RemoteException. • Has a remove method on home that takes a PK (Session Beans don't), so able to remove without instantiating a bean.
Remote	• The remote interface once again extends EJBObject • Contains the business methods with the same signature as in the concrete class plus throwing RemoteException.

Entity Bean - Container Managed Persistence (as BMP except..)

Difference	*CMP*	*BMP*
Class definition	EJB 2.0 Abstract EJB 1.1 Not Abstract	Not Abstract
Database access calls	Generated	Coded
Persistent State	Abstract get/set methods and no attributes, these are specified in the deployment descriptor	Attributes
Access method for persistent and relationship fields	Required	None
FindByPrimaryKey method	Generated	Coded
Finder methods	Generated & EJB QL.	Coded
Select methods	Generated, not callable by client, only by business methods in the bean. They are like finder methods except they can return things other than local/remote interfaces, abstract and public – template method pattern. They can be called on the bean in the pooled or ready states.	None
Return from ejbCreate	Null	Primary key
Number of ejbCreate methods required	0	1

Message-Driven Bean

Bean	• Implement the javax.jms.MessageListener and javax.ejb.MessageDrivenBean interfaces. • The class must be public and not abstract or final. • Implement the 'onMessage' method • Implement one 'ejbCreate' and one 'ejbRemove' methods. • Have a public default constructor. • There's no remote or local interfaces. • You specify the queue or topic to use in the deployment descriptor via JNDI names.
Home	• None
Remote	• None

Types of Exception

System Exception

If you get a system exception throw the run-time exception, javax.ejb.EJBException. Eg: SQL insert fails because database is full. If thrown the container might destroy the bean instance. **Causes a transaction rollback**.

Application Exception

When there is a business logic problem. There are two types, customised and predefined. A customised exception is one you've coded yourself, for example NoMoneyLeftException. Predefined exceptions are found in the java.rmi and javax.ejb packages, for example ejbCreate() should throw a CreateException to indicate an invalid input parameter. If an object has timed out then a java.rmi.NoSuchObject(Local)Exception is thrown. **Does not cause the rollback of the transaction**! So must do a context.setRollbackOnly() if want to rollback.

CMP v BMP

- CMP: Requires less code to be written and maintained
- CMP: Focus on business logic therefor faster development and quicker time to market.
- CMP: Beans are portable across database vendor or schema changes.
- CMP: The container may offer caching services so perhaps more efficient code, especially finder methods.
- BMP: It is not possible to capture every relationship in EJB QL, some greater persistence control with BMP.
- BMP: You need a EJB 2.0 vendor, EJB 1.1 CMP was embarrassingly crap.
- BMP: More efficiently use database.
- CMP: BMP recommends an extra DAO layer.
- BMP: Best with complex data objects/relationships that need to be fast.

Transactions

ACID – Atomic, Consistent, Isolated and Durable. Use the `SessionSynchronization` interface if you want to be informed when the transactions are being committed/rolledback (only on Stateful Session Beans). It adds additional lifecycle methods. Transactions can be managed by the bean (BMT), that's Session and Message Driven Beans only, or the container. Container manager transaction (CMT) can be influenced by context.setRollbackOnly(). Container managed transactions are preferable but Bean managed gives you finer transactional control. The container managed transaction scopes:

- Required – create a new transaction if there isn't one already, otherwise join it.
- RequiresNew – if an existing transaction, pause it, the start a new transaction, do the work and commit/rollback, then resume the existing transaction. If no existing transaction just start and stop one for this method. Note: This is not a nested transaction as the inner one has no effect on the outer.
- Mandatory – use existing transaction, throws a TransactionRequiredException if no transaction already.
- NotSupported – stop the current transaction, make the call and carry on – may improve performance.
- Supports – uses a transaction if there is one, otherwise doesn't bother.
- Never – throws a RemoteException if a transaction exists, otherwise it carries on fine and doesn't start a transaction.

Isolation Levels of java.sql.Connection – Dirty Read = read a changed value in a concurrent transaction before that transaction has decided to commit or rollback; Non-Repeatable Read = A transaction re-reads data it has previously read and finds that the data has been modified by another transaction (that committed since the initial read). The data changes within a transaction from one read to the next; Phantom Read = A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction. That is, new rows have been added and others deleted since starting the transaction, read again and the it's all changed. Serializable level makes the transaction appear to take a snap shot of the data when you start it; nothing changes and the world is fixed until you commit or rollback.

Level	Dirty Reads	Non-repeatable Reads	Phantom Reads
TRANSACTION_NONE	N/A	N/A	N/A
TRANSACTION_READ_UNCOMMITTED	Yes	Yes	Yes
TRANSACTION_READ_COMMITTED	No	Yes	Yes
TRANSACTION_REPEATBLE_READ	No	No	Yes
TRANSACTION_SERIALIZABLE	No	No	No

Deployment Descriptors

EJBs deployed to JAR files.
Deployment descriptor is called 'META-INF/ejb-jar.xml' (XML format since EJB 1.1)

EJB container

State the benefits of bean pooling in an EJB container.

State the benefits of Passivation in an EJB container.

State the benefit of monitoring of resources in an EJB container.

Explain how the EJB container does lifecycle management and has the capability to increase scalability.

Bean Pooling

- For performance - re-cycle objects and make more efficient use of resources.
- EJBs are heavy objects so need to avoid create and deletes where possible.

Passivation

- To free up resources, swap out unused objects and activate if required.
- Only available to stateful session and entity beans because these are the two types with state. No point for stateless.
- Do not store EntityConext/SessionContext, UserTransactions, JNDI contexts as transient as they are passivated by the container (see below).
- Responsibilities in ejbPassivate() method, leave the instance's non-transient attributes as:
    1. A serialized object
    2. A null
    3. Remote or Local I/F
    4. Remote or Local Home I/F
    5. SessionContext or EntityContext
    6. Reference to a environment naming context (java:comp/env JNDI context).
    7. UserTransaction
    8. Reference to a Resource Manager or Connection Factory
    9. Timer

  For example: close all JDBC connections and assign attribute to null.

- Then the container looks after saving and restoring the attributes following Serializable semantics plus those other types of attributes as mentioned above.
- If the container cannot passivate an object it may destroy the instance.
- The container follows Serializable semantics except it does not reset 'transient' attributes. These are therefore discouraged.

Life-cycle Management

- Instance pooling
- passivation/activation.

Protocols

Given a scenario description, distinguish appropriate from inappropriate protocols to implement that scenario.
Identify a protocol, given a list of some of its features, where the protocol is one of the following: HTTP, HTTPS, IIOP, JRMP.
Select from a list, common firewall features that might interfere with the normal operation of a given protocol.

TCP/IP (Transport Control Protocol over Internet Protocol)

IP, the basic protocol of the internet, enables the unreliable delivery of individual packets from one host to another. IP makes no guarantees about whether or not the packet will be delivered. TCP adds the notion of connection and reliability.

HTTP 1.0 (Hyper Text Transfer Protocol)

Port 80. Built on top of TCP/IP. Stateless, connection less and directional. Client opens a connection (using a URL – the direction) and sends request to the server, the server responds using the same connection, then the connection is closed. It is up to the client to optionally maintain the session (through cookies or URL re-writing).

HTTP 1.1 (Hypertext Transfer Protocol)

Port 80. Same as HTTP 1.0 except there is an option to keep the connection alive from request to request. This would be known as connection oriented if it weren't for the fact that either side can potentially drop the connection at any time (Apache does after 15 seconds), as such it is still 'connection-less'. However, some people think it is connect oriented because you need a connection to be open to do a request at all. I disagree, but I get the impression this is what Sun thinks, I suppose that is what counts for the exam at least, it is very confusing.

HTTPS (HTTP with SSL – Secure Sockets Layer)

Port 443. Adds security to HTTP. Supports the uses of X.509 digital certificates to authenticate the sender. SSL uses 40 or 128 bit key encryption. Encrypts both requests and responses. Also provides a maintenance of state from one request to the other.
Note: Not to be confused with S-HTTP, a security enhanced version of HTTP.

SOAP (Simple Object Access Protocol)

Light weight protocol for exchange of information in a decentralised, distributed environment. It is an XML based protocol. It can be used with other protocols for transport but typically this is HTTP(s).

RMI/JRMP (Remote Method Invocation with Java Remote Method Protocol)

Port 1099
Connection based
Supports pass by value (if Serializable or primitive) and pass by reference.
Java to Java only (Java 1.1) but using JNI can talk to/from other languages.
Need to run/expose a `RMIRegistry`
Compile Interface with `rmic` tool to get stubs and skeletons.
Easy to develop/use, portable, Java security and garbage collection.
Client needs to open a socket to the server.
Will tunnel over HTTP if cannot get through firewall.
Faster than RMI/IIOP because less generic.

RMI/IIOP (Remote Method Invocation with Internet Inter ORB protocol)

Supports pass by value (in CORBA 2.3) else only data values (arguments and return types)
To allow <u>CORBA services to use your services</u> it is recommended you use RMI/IIOP, for example to access an EJB from a CORBA service.
Interoperate with other IIOP services - but only if remote I/Fs are defined as Java RMI I/Fs.
Legacy connectivity to CORBA (not always) plus also able to connect to RMI/JRMP, both.
Java-to-IDL mapping required (in CORBA 2.3).
Objects passed over not garbage collected – must manually ‘unexportObject’
Easier to program than Java IDL because uses RMI model.

Java IDL

An alternative to RMI/IIOP, same as normal CORBA programming – write the IDL, implement it, etc. business as usual.
Available in Java 1.2
To <u>use an existing CORBA service</u> it is recommended you use Java IDL.
Interoperate with non-Java services via IIOP.
Uses `idlj` (formally `idltojava`) compiler

IIOP (Internet Inter ORB Protocol)

Port 535
Interoperate with CORBA 2.0
Open standard are defined by OMG to avoid vendor lock-in.
IIOP includes embedded port and address information so hard to let through firewalls that are set up for ports and addresses in TCP/IP protocol.
Allows call backs, the client opens a connection to the server, and the server opens a connection back to the client for the response.
Pre-CORBA 2.3 can only pass data values and return types not actual objects.

DCOM (Microsoft’s Distributed Component Object Model)

Heavily used on Windows.
There are CORBA to DCOM bridges.
Very similar to CORBA.
Uses DCOM IDL for interface definition.
Supports multiple interface inheritance
GUID in interface to uniquely identify it.
Uses Windows registry as naming service.

FTP (File Transfer Protocol)

Port 21. Used to copy files between computers-usually a client and an archive site. It's old-fashioned, it's a bit on the slow side, it doesn't support compression, and it uses cryptic Unix command parameters. But the good news is that you can download free apps that shield you from the complexities of Unix, and you can connect to FTP sites using a Web browser.

POP3 (Post Office Protocol)

Port 110. The current champ in Internet e-mail mailbox access standards, but its limitations - basically, you connect to a server and download *all* your messages, which are then deleted from the server-discourage flexibility. Of course, some clients let you leave all messages on the server, and/or refuse to download messages above a certain size. Still, as messages become longer-with multimedia (such as sound or video) objects and the likes-we'll want some flexibility in what we retrieve and when we retrieve it.

IMAP4 (Internet Message Access Protocol)

Port 143. IMAP provides a means of managing e-mail messages on a remote server, similar to the POP protocol. But IMAP offers more options than POP, including the ability to download only message headers, create multiuser mailboxes, and build server-based storage folders.

SMTP (Simple Mail Transport Protocol)

Port 25. Used to send e-mail messages.

Firewalls and Protocols

A firewall is a system to prevent unauthorised access to/from a private network. Typically close ports for unauthorised access. However port 80 for HTTP will be open so can use tunnelling (SOAP and RMI can avoid firewall configuration issues by piggy backing (tunnelling) on HTTP). However, is firewall avoidance a good thing if effectively you are allowing requests through to your systems? Other protocols required ports to be opened explicitly.

Application of J2EE technology

Select from a list those application aspects that are suited to implementation using J2EE.

Select from a list those application aspects that are suited to implementation using EJB.

Identify suitable J2EE technologies for the implementation of specified application aspects.

J2EE

- Distributed, requires transaction, security, resource management.
- JSP/Servlet – suited to user aspects.
- EJBs – suited to business logic, workflow management, persistence, transactions, security, conversation state.

Technologies

- JNDI – Java Naming and Directory service Interface API
- JMS – Messaging MOM, API.
- RMI/JRMP – Distributed comms and RPC in java.
- RMI/IIOP – Distributed comms and RPC, interpretability with OMG ORBs.
- JDBC – Java Database Connectivity API.
- EJB – Distributed components for business logic
- Servlets – HTTP Controller.
- JSP – HTTP, presentation centric
- JTA – Java Transaction API.
- JavaMail – Java Mail APIs.
- JAF – Java Activation Framework, reflection, etc.

From a list, select the most appropriate design pattern for a given scenario. Patterns will be limited to those documented in Gamma et al. and named using the names given in that book.

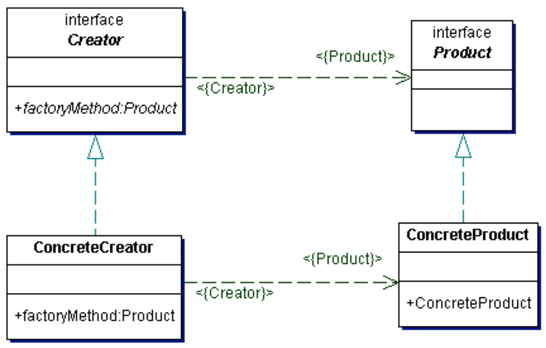
State the benefits of using design patterns.

State the name of a Gamma et al. design pattern given the UML diagram and/or a brief description of the pattern's functionality.

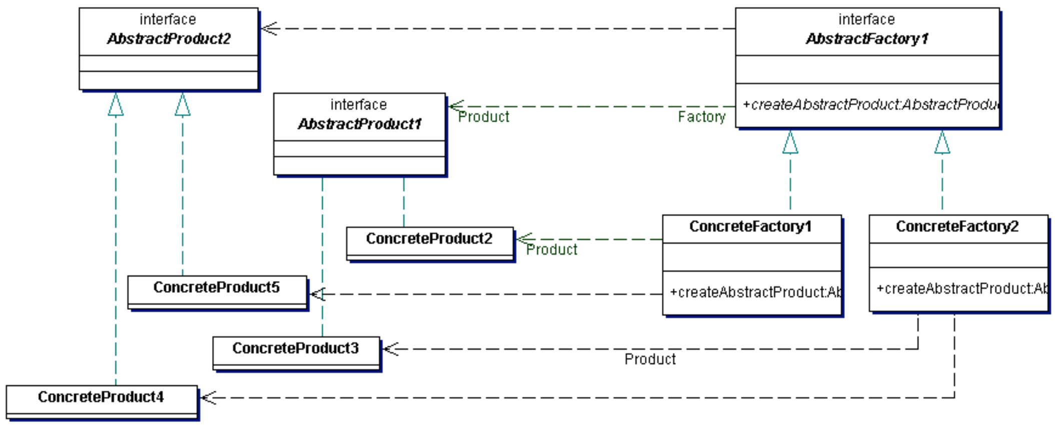
Select from a list benefits of a specified Gamma et al. design pattern. Identify the Gamma et al. design pattern associated with a specified J2EE feature .

GOF Creational Patterns

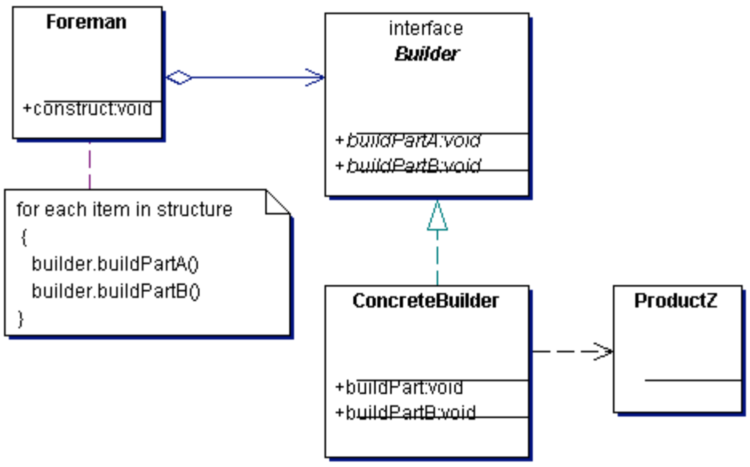
- **Factory Method** - decision making which returns several possible subclasses based on a hint. Sub-classes decide which class to create.



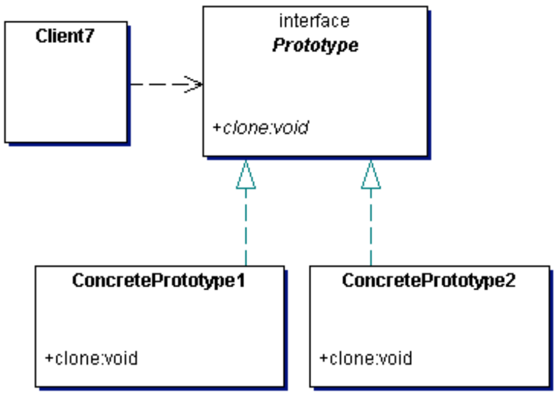
- **Abstract Factory** - an interface to a Factory for several families of related objects. Use when want to create a bunch of classes together, if just want one object then use Factory Method, eg: EJBHome.



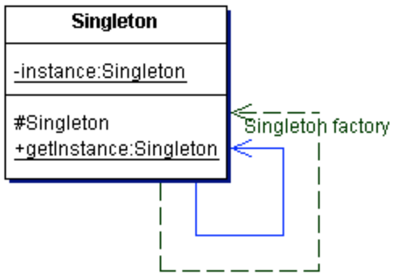
- **Builder** - separate construction of complex objects from representation, so several representations can be created.



- **Prototype** - copy or clone an object to recycle it. From the client's point of view it doesn't know how and by whom the objects are being created. No need for factories to create objects.

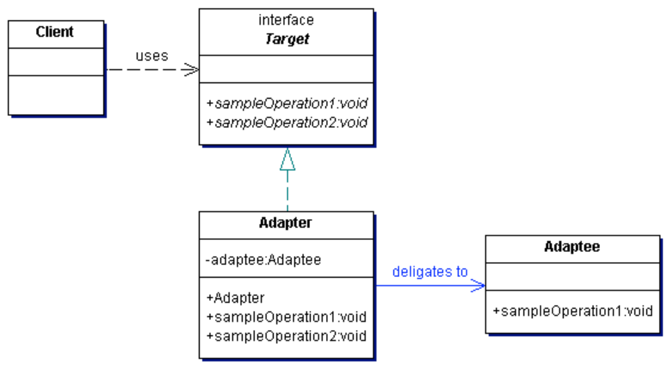


- **Singleton** - single instance, global point of access, could have controlled number of instances.

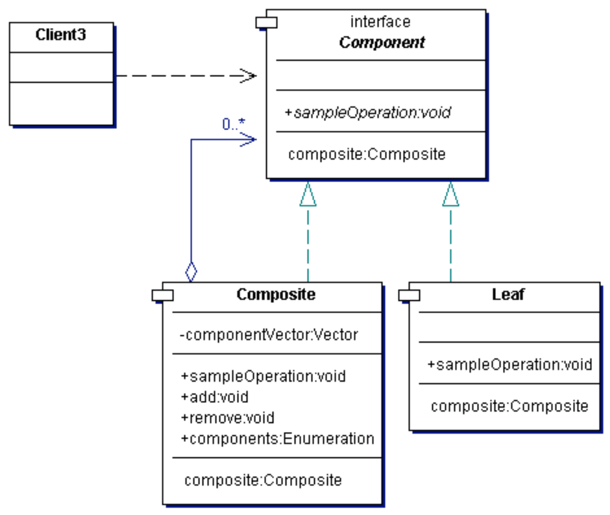


GOF Structural Patterns

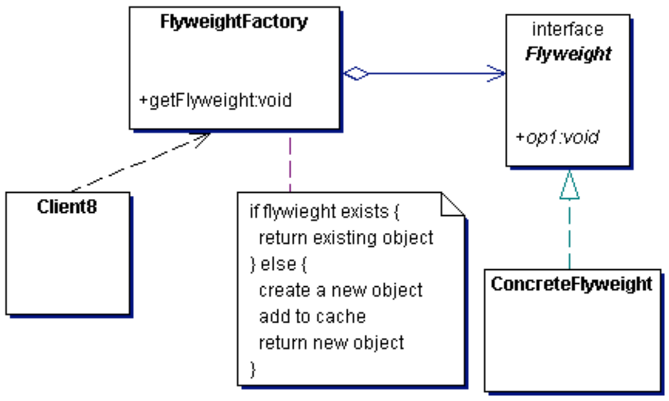
- **Adapter** - convert the programming interface of one class to another. Allows unrelated classes to work together easily. Class that exposes an nice interface and wraps and calls another class. The intent is to make classes look the same as a particular class. Allows classes to work together that wouldn't be able to because of incompatible interfaces.



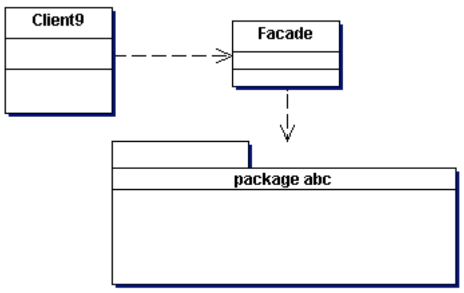
- **Composite** - to represent a part or collection of parts.



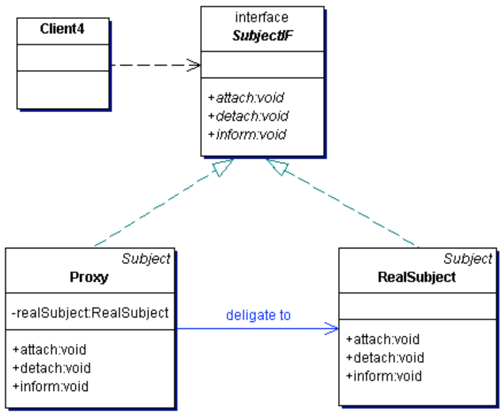
- **Flyweight** - do not store all the details of an instance as attributes. Instead have just the intrinsic stuff as attributes and pass the other 'extrinsic' details as part of a method call. The intrinsic stuff is in the class and no need to create a complex inheritance hierarchy just re-use the same class because it doesn't have the unused attributes. Each object does not hold isn't own state, instead held externally.



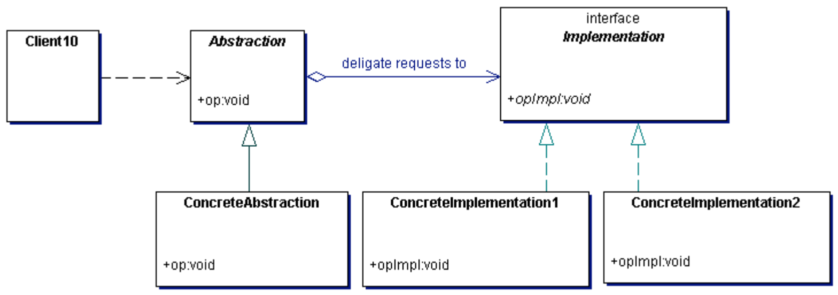
- **Façade** - Provide a simple interface to a complex framework of classes, eg: JDBC interface. Eg: surrogate or placeholder for remote class.



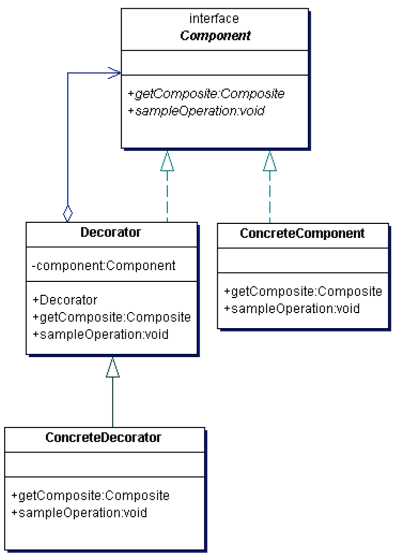
- **Proxy** - Wrap another object, pass calls on, but can postpone creation of the wrapped object, cache, etc. Eg: EJB, RMI stubs and EJBObject.



- **Bridge** - Like an adapter but designed to separate the interface from the implementation without changing the client code. Allows interface and abstraction to vary. Eg: JNDI and Business Deligate.

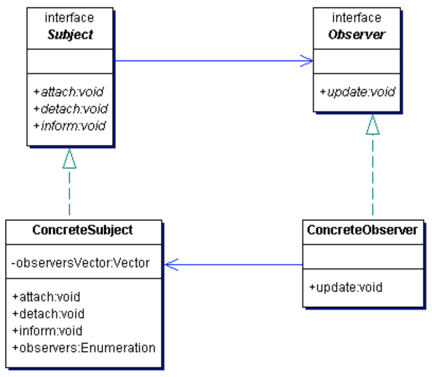


- **Decorator** - modify the behaviour without having to create a new derived class. The decorator calls a specific decorator instances to add their decorations as required. Kind of delegate to another class some aspects of the object that may be changeable. *Note: the composite pattern in it.*

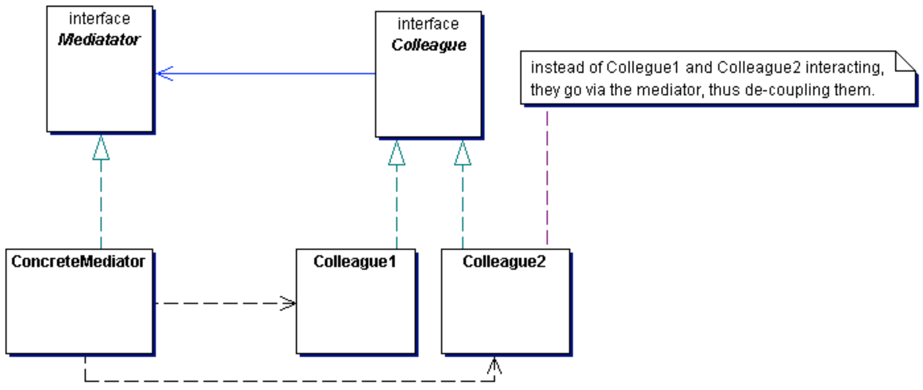


GOF Behavioural Patterns

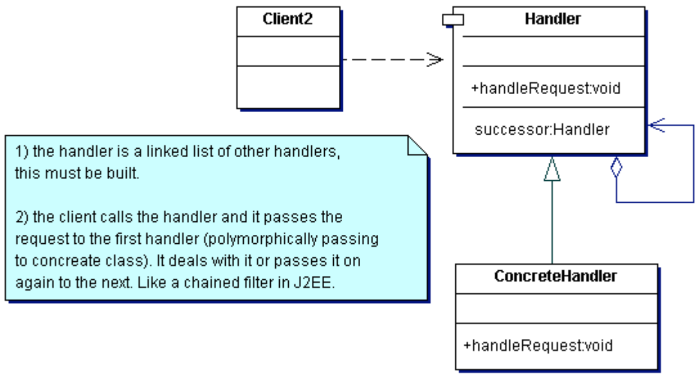
- **Observer** - objects that contain the data are separate from the objects that display the data. The observers observe changes in the data (subject). The observers register an interest in the subject and it calls the observers back when something changes. Publish/Subscribe pattern.



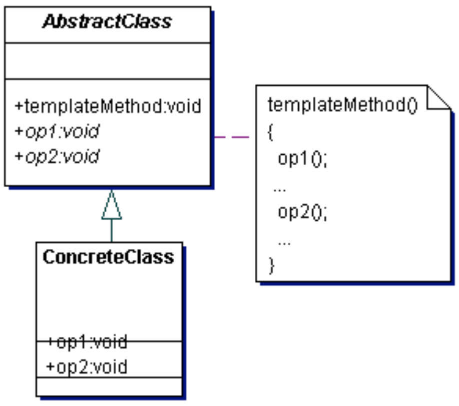
- **Mediator** - to avoid too much inter-class dependency use a mediator intermediary class. This class is informed of changes and it in turn informs those classes with an interest. Prevents every class needing to know about all the other classes.


interface
Mediatator
interface
Colleague
instead of Collegue1 and Colleague2 interacting, they go via the mediator, thus de-coupling them.
ConcreteMediator
Colleague1
Colleague2

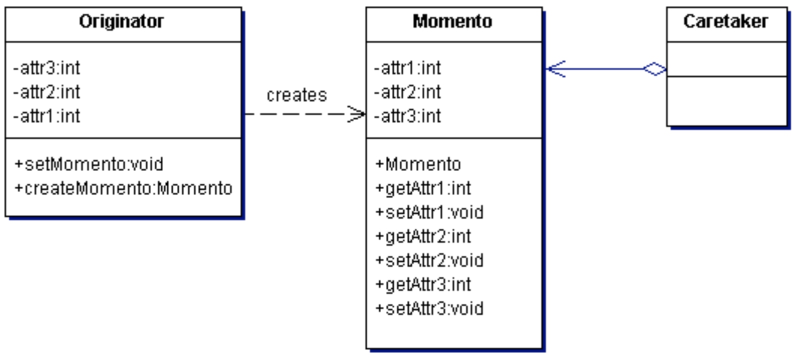
- **Chain of Responsibility** - a chain of classes where a request is passed to them all. The class with an interest in a particular request then deals with it. The chained classes do not need to know about each other. The sender is de-coupled from the receiver.


Client2
Handler
+handleRequest:void
successor:Handler
1) the handler is a linked list of other handlers, this must be built.
2) the client calls the handler and it passes the request to the first handler (polymorphically passing to concreate class). It deals with it or passes it on again to the next. Like a chained filter in J2EE.
ConcreteHandler
+handleRequest:void

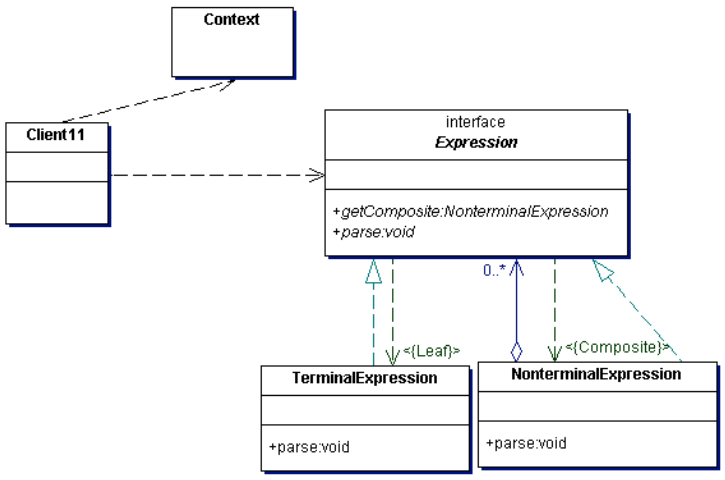
- **Template Method**- define an algorithm in a class but leave some details to be implemented by a sub-class. Some parts of the algorithm in the base others in the derived classes.



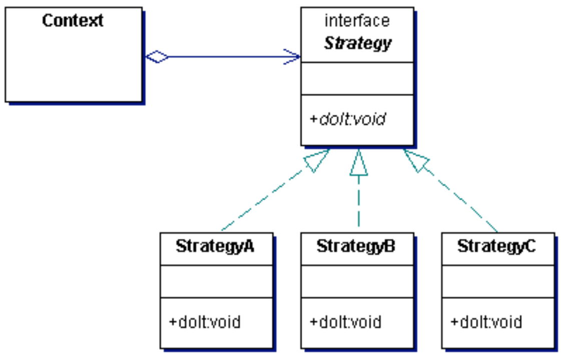
- **Momento** – capture and store an object's internal state. The client uses the origination class but wants keep it's state without breaking encapsulation. So, it asks it to create a small bean momento object. Then the client gives this to another class to take care of it. Later, if the client wants to restore the origination to it's old state it passes the old momento to the originator and it copies the state back into itself.



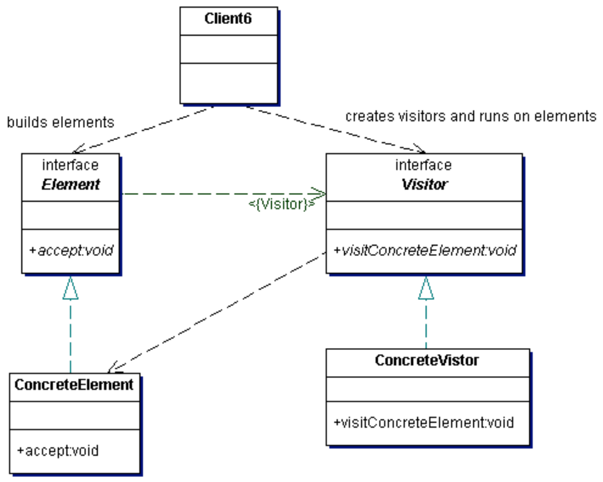
- **Interpreter** - when you pass a query string to a class, it interprets the string and performs the operation. The interpreter handles parsing, etc and produces varying kinds of output. *Note: the composite pattern in it.*



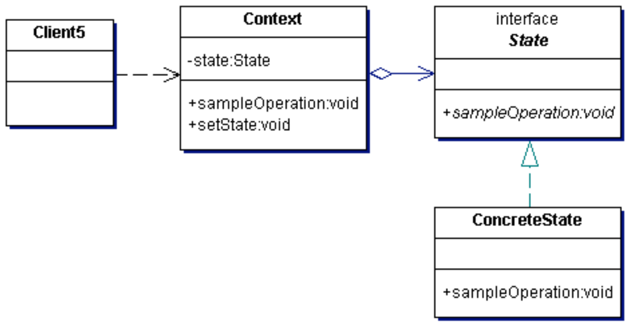
- **Strategy** - encapsulate algorithms in a class, the class chooses the correct algorithm for the particular context. Similar to state, state is a strategy with only state algorithms.



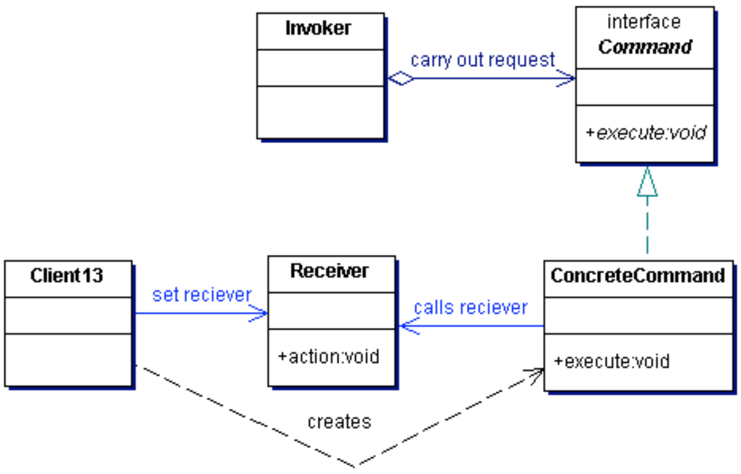
- **Visitor** - adds a function to a class. Instead of putting the code in multiple classes, such as a draw method on Square and Triangle, simply write another class with it in and get the Square and Triangle to call the public draw method of the visitor. Adds functionality without changing the class. Allows you to centralise things on place not multiple classes with similar algorithms, etc.



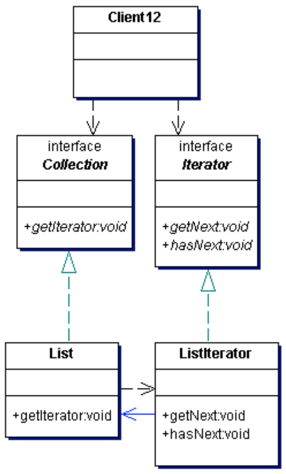
- **State** - similar to strategy except to do with switching between states. The class encapsulates the state transitions.



- **Command** - separate execution of a command from the interface. Separate controller logic and allows plug-able new commands and undo. Encapsulate a request in an object.



- **Iterator** - move through a list using standard interface without worrying about internal representations of the data. Eg: Resultset.



GOF summary

Pattern Name	*Example*	*Type*
Abstract Factory	EJBHome	Creational
Builder		Creational
Factory Method		Creational
Prototype		Creational
Singleton		Creational
Adapter		Structural
Bridge		Structural
Composite	AWT/Swing containers	Structural
Decorator	Chained java.io stream classes	Structural
Façade	Some Session EJBs	Structural
Flyweight		Structural
Proxy	Impls of EJBObject and EJBHome	Structural
Chain of Responsibility		Behavioural
Command		Behavioural
Interpreter		Behavioural
Iterator	ResultSet	Behavioural
Mediator		Behavioural
Momento		Behavioural
Observer	AWT/Swing event model	Behavioural
State		Behavioural
Strategy		Behavioural
Template Method		Behavioural
Visitor		Behavioural

***Note**: although a design pattern may produce higher quality solutions, encourages re-use and produces recognisable solutions, they also may be slower and more complex.*

J2EE Presentation Tier Patterns

- **Intercepting Filter** – eg: servlet filters, plug-able, chained, like decorator, reusable, flexible.
- **Front Controller** - central point of control, typically a Servlet, delegate to command objects, reusable, C in MVC.
- **View Helper** - Bean or Tag in JSP, encapsulate business logic away from view, M in MVC, reusable, make use of Value Objects and Business Delegates
- **Composite View** - include static/dynamic content in view, can use tags or XSL to add extras, reuse, flexible, manageable, slow performance.
- **Service To Worker** - when delegate takes a request works with business to determine next view, then delegate to it, sophisticated in choice of next view, uses front controller, business delegate, helpers, etc, get data from dbase early and potentially use in view choice.
- **Dispatcher View** - identical to Service To Worker except the dispatcher does less and the access to the dbase is done later. No access to dbase/EJB tier to determine the view, instead all data required is in the request, later in view get data from dbase/EJB.

J2EE Business Tier Patterns

- **Business Delegate** - proxy/façade to business tier, hide impl details, hide caching, service naming lookup, wrap and shield client from exceptions from business tier, eg: RemoteException, JMS Exception, transparent retry/recovery, cache results.
- **Value Object** - reduce network traffic, copy values, entity/session bean or DAO may create item on server, client may create and pass over to server. May introduce stale data problems.
- **Session Façade** - to avoid the client having to manage remote relationships, encapsulate on server, better performance, simpler to develop to, typically a session bean to manage entity beans, acts as controller on server side, less remote calls, central transaction control, central security control.
- **Composite Entity** - don't have fine grained entity beans, instead course grained and dependant objects = composite entity, less entity beans so less network traffic, greater performance, could lazy load dependants and even have dirty flag to not over store data.
- **Value Object Assembler** - build Value Objects, interact with business objects, DAOs, EJBs, etc to create a complex VO, the client just uses the VOAssembler and is shielded from it all, VOAssembler could be a session bean, separate object creation out, Vos improve performance, less chattiness.
- **Value List Handler** - iterator finder method on server, get 'n' VOs starting at a position, access to DAO directly not use finder methods on Entity EJBs, potentially stateful session bean, better when return lots of VOs, potentially cache results on server, only requested data transferred there more efficient.
- **Service Locator** - wrap JNDI, abstract, reduce code complexity, single point of control, could cache, JNDI or JMS service location.

J2EE Data Tier Patterns

- **DAO** - Data Access Object, encapsulates and abstracts data access and vendor specific'ness, manage connections, etc, create VOs, easier migration to CMP, reduce complexity, extra layer.
- **Service Activator** - replace by message driven beans, stand-alone service, call EJBs when required asynchronously.

Messaging

Identify scenarios that are appropriate to implementation using messaging, EJB, or both.

List benefits of synchronous and asynchronous messaging.

Select scenarios from a list that are appropriate to implementation using synchronous and asynchronous messaging.

A Definition

Messaging is a way for software components to communicate. It is loosely coupled in that the sender does not need to know about the receiver and vice-versa. The only contract is the format of the message. In fact the sender and receiver do not even have to be available at the same time.

JMS API (Java Messaging Service)

Generic API on top of a MOM (Message Oriented Middle-ware, eg: MQSeries) layer in the same way as JDBC is on top of SQL for RDBMS. Also, it supports asynchronous messaging and the aim to delivery once and only once (reliability). Found in package javax.jms.*

When and when not to use

When to:
- Use for performance; stick on queue and reply, no blocking for lengthy processing.
- Smooth load balancing; the least busy machine will 'pull' the message from the queue.
- Integrate with legacy system that uses a MOM.
- Parallel processing by launching a number of messages and continue.
- Need reliable asynchronous communication.

When no to:
- When your not sure the operation will succeed, can't throw exceptions.
- When you need to return a value.
- When you want a 'simple' system; a MOM is less manageable.

J2EE 1.3

- EJB and Web Container objects can send and receive synchronous JMS messages.
- EJB and Web Container objects can create asynchronous JMS messages.
- EJB Container supports Message Driven EJBs for receiving asynchronous messages.
- The sending and receiving can participate in distributed transactions and concurrent consumption of messages. That is, when you commit a message is guaranteed to be delivered, if the business logic fails on the consumer side is another matter.

Asynchronous v Synchronous

- **Synchronously**. A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously**. A client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage method, which acts on the contents of the message.

When you think about it the messaging is in fact two separate synchronous process; one to send a message from a publisher/sender to a queue/topic; and one to receive from a queue/topic. They can be combined to be both synchronous and asynchronous.

Architecture

- JMS Provider – system that implements the interfaces, the MOM.
- JMS Clients – Java producers and consumers of messages.
- Native Clients – non-Java producers and consumers of messages.
- Messages – objects passed between clients, there are various types, TextMessage, BytesMessage, MapMessage, etc.
- Point-to-point messaging – one consumer of a message, no timing dependancies between the client and server, consumer acknowledges the message has been consumed.
- Publish-Subscribe – producer publishes (sends) messages to a 'topic'. Consumers subscribe to the topic to receive the messages. There can be multiple publishers and subscribers. One message can be sent to many consumers. The producer and consumer have a timing dependancy – the producer must remain alive until the consumers have consumed.
- Publish-Subscribe with durable subscriptions – as above but allows the producer to not be alive whilst they are being consumed, thus more flexible.
- Destination Factory – administered by J2EE and found via JNDI, accessed by generic vendor neutral interfaces, used to specify a target for messages (queue or topic name).
- Connection Factory - administered by J2EE and found via JNDI, accessed by generic vendor neutral interfaces, used to create a connection to a JMS provider (the MOM and type of, eg: queue or topic).

State three aspects of any application that might need to be varied or customized in different deployment locales.

Match the following features of the Java 2 platform with descriptions of their functionality, purpose or typical uses: Properties, Locale, ResourceBundle, Unicode, java.text package, InputStreamReader and OutputStreamWriter.

Aims

- Allow a program to run worldwide by not hard coding aspects that need to change from locale to locale. These aspects being things such as:
  - Messages
  - Help
  - Sounds
  - Colours
  - Graphics
  - Dates/Times
  - Currencies
  - Numeric formats
  - Measurements
  - Phone numbers
  - Addresses

The classes

- java.util.Properties – configuration data, can be loaded/saved to/from a stream, can be locale specific.
- java.util.Locale – a geographic/political/cultural region, simply represents it as an identifier.
- java.util.ResourceBundle – all the locale specific details bundled together per locale, you can write a class that inherits from ResourceBundle or create file with .properties extension, uses a <name>_lang_COUNTRY_VARIANT, eg MyResourses_en_ENG_UNIX or MyResources_ge_CH, ListResourceBundle – backed with a class to deal with resources as a list, and PropertyResourceBundle – deal with resources in properties files.
- Formatting – classes that inherit from java.text.Format to parse/format, eg: DateFormat, MessageFormat and NumberFormat. Eg:

```
NumberFormat                        currencyFormatter                        =
NumberFormat.getCurrencyInstance(currentLocale);
String currencyOut = currencyFormatter.format(new Double("123423423.31");
// "$123,423,423.31" in en_US
// "123 423 423,31F" in fr_FR
// "123.423.423,31" in de_DE
```

- Locale-Specific String Classes – Collator = string comparison, CollationElementIterator = iterate through international strings, CollationKey = Collator key, BreakIterator = find line breaks, sentences, etc, CharacterIterator = bi-direction Iterator through Strings.
- UNICODE 2.0 (2.1) – two byte chars, 16 bits.
- java.io.InputStreamReader – bridge to convert a stream of non-Unicode bytes to Unicode chars. Eg:
  ```
  InputStreamReader r = new InputStreamReader( new FileInputStream("test.txt"),"UTF8");
  ```
- java.io.OutputStreamWriter – bridge to convert a string of Unicode chars to non-Unicode bytes. Eg:
  ```
  OutputStreamWriter osw  = new OutputStreamWriter(new FileOutputStream("test.txt"), "UTF8"));
  osw.write(str);
  ```

Select from a list security restrictions that Java 2 environments normally impose on applets running in a browser.

Given an architectural system specification, identify appropriate locations for implementation of specified security features, and select suitable technologies for implementation of those features.

Applets

Applets require the installation of their own security manager, by the browser vendor. In general, since you do not know who you are downloading an Applet from, they are considered 100% untrustworthy and run in a 'sandbox'. The sandbox allows the executable code to run on the client's machine but with restrictions on what it can do. The sandbox prevents applets from:

- accessing the file system
- opening sockets to destinations other than the IP address from which it originated.
- acting as a socket server
- accessing some system properties
- starting other programs on the client
- making native calls
- creating new 'trusted' windows
- installing their own class loader or security manager.
- no restriction on memory or CPU usage though, can create lots of threads.

Signed Applet (Java 1.1) – authentication of the signed classes (only one needs to be to be considered a 'signed applet') downloaded. The authentication is to check they have not been replaced during transmission by a malicious attacker. There needs to be a certificate to sign the classes with. The 'trusted' classes is then allowed (without restriction in Java 1.1 and via the policy file in Java 1.2) to perform operations outside of the sandbox. The process to developing and shipping a trusted class is:

- Interacting with the security model, that is; calling specific operations to enable or disable privileges.
- Packaging the trusted class in a JAR (Netscape) or CAB (IE) file. JAR files are created using the 'jar' program and CAB files using the 'cabarc' or 'dubuild' programs.
- Get a digital certificate from a CA or a self-issued certificate for testing purposes.
- Sign the trusted class or its container with the certificate. The signature assures the receiver that someone is accountable for it's actions. Message digests created by the signing tool verifies that the class or container is unchanged since it was signed. To sign use the 'signtool' (Netscape), 'signcode' (IE) or 'jarsigner' (JVM).
- Configure the environment (Browser) to grant the 'trusted' class permission. Either through the browser config or java.policy file.

Policytool (Java 1.2) – creates a '.java.policy' file that grants permissions to Applet, thus allowing Applets to do more, held on client's normally PC in user's home directory. The file is collection of policy entries, entries list permissions for a code source (URL), many specify a location of a key-store of keys to check signed classes against. Implements the 'principal of least privilege' – give application only the privileges that is needs to carry out its functions and no more.

Security Features/Terms

- ***Cryptography***
  - java.security and javax.crypto packages – contains standard algorithms but no encryption due to US export laws.
  - New in Java 1.1
  - Symmetric encryption – faster than asymmetric, single key to en/decrpt, however need to pass the same key to both parties, example: DES.
  - Asymmetric encryption – public/private keys, no need to pass the same keys so safer than Symmetric but slower, encrypt with the receiver's public key, example: RSA.
  - Session Encryption – PGP, a hybrid of asymmetric and symmetric. It has the connivence of asymmetric and the speed of symmetric. PGP creates a session key, which is a one-time-only secret key. This session key works with a very secure, fast conventional encryption algorithm to encrypt the plaintext; the result is ciphertext. Once the data is encrypted, the session key is then encrypted to the recipient's public key. This public key-encrypted session key is transmitted along with the ciphertext to the recipient. The recipient's copy of PGP uses his or her private key to recover the temporary session key, which PGP then uses to decrypt the conventionally-encrypted ciphertext.
  - Message Digest – one-way function to get a hash-value of a msg, used to create a 'finger print' to check data integrity, Message Digest Functions - eg: MD4/5.
  - Digital Signature – encrypted message digest using private key, sent with msg to ensure the message hasn't been changed. Also known as a MAC – Message Authentication Code.
  - Digital Certificates – A digital certificate is a message that is signed by a certification authority that certifies the value of a person or organisation's public key. Contains names of entity for whom certificate was issued- the 'subject', public key of subject and digital signature of CA to verify the certificate.
  - Encryption algorithms are based on mathematically difficult problems - for example, prime number factorisation, discrete logarithms. Elliptic curves can provide versions of public-key methods that, in some cases, are faster and use smaller keys, while providing an equivalent level of security. Their advantage comes from using a different kind of mathematical *group* for public-key arithmetic

Asymmetric Steps

i. Sender calculates the message digest using a message digest algorithm.
ii. Sender encrypts the message using the public key of the reciever.
iii. Sender encrypts the digest using the sender private key.
iv. Sender sends the message and digest to the receiver.
v. Receiver decrypts the message using the sender's private key.
vi. Receiver calculates the message digest using the same message digest algorithm as before (i).
vii. Receiver decrypts signature to get message digest using sender's public key.
viii. Receiver checks the digests are equal
ix. If digests equals then use the message..

Benefits

i. Unforgettability – only signer has private key.
ii. Verifiability – public key available so anyone can verify message.
iii. Single-use – signature unique to each message.
iv. Non-repudiation – only the signer has his private key so only she could have signed it thus proving it was from her.
v. Sealing – digitally sealed, message cannot be altered without invalidation.

- ***SSL*** – Port 443, secure sockets layer, sits on top of TCP/IP layer and below application layer (eg: HTTP, LDAP, IMAP), can use a variety of encryption algorithms, detect modified or inserted data in messages. **SSL Server Authentication** – confirm the identity of the server using standard public-key cryptography to check the server's certificate and public id are valid and have been issued by a CA that the client trusts. **SSL Client Authentication** – the same as server but to check the identity of the client (optional). **Encrypted SSL Connection** – all information between parties is encrypted to provide confidentiality and temper detection.

- ***Sandbox*** – Byte code verifier to check language safety constraints; access controller to check methods on stack against permissions (has a 'doPrivileged' method to allow code to do things outside the sandbox if they are allowed via primissions); Security Manager; Class Loader and java.security package.

- ***Cypher* –** cryptographic algorithm. The SSL protocol supports the use of a variety of different cryptographic algorithms, or ciphers, for use in operations such as authenticating the server and client to each other, transmitting certificates, and establishing session keys. Clients and servers may support different cipher suites, or sets of ciphers, depending on factors such as the version of SSL they support, company policies regarding acceptable encryption strength, and government restrictions on export of SSL-enabled software. Among its other functions, the SSL handshake protocol determines how the server and client negotiate which cipher suites they will use to authenticate each other, to transmit certificates, and to establish session keys.

- ***Diffie-Hellman*** - The Diffie-Hellman key agreement protocol (also called exponential key agreement) was developed by Diffie and Hellman [DH76] in 1976 and published in the ground-breaking paper "New Directions in Cryptography." The protocol allows two users to exchange a secret key over an insecure medium without any prior secrets. The Diffie-Hellman key exchange is vulnerable to a *middleperson attack*. This vulnerability is due to the fact that Diffie-Hellman key exchange does not authenticate the participants. Possible solutions include the use of digital signatures and other protocol variants.

- ***Java Security Classes***
  a. java.security.PriviledgedAction – Interface for an action that needs security.
  b. java.security.PriviledgedExceptionAction – Exception from priviledged action.
  c. java.security.GuardedObject – used to wrap an object you intend to guard.
  d. java.security.CodeSource – location (URL) and certificate(s) used to verify signed code from the location.
  e. java.security.KeyPairGenerator – Factory for creating private and public keys.

- ***Keystore* –** database of 'keytool', stores private and public keys, each entry has: alias, 1 or more certificates, optional private key protected by a password.

- ***JAAS* (Java Authentication and Authorisation Service)** – Authentication - based on Pluggable Authentication Modules (PAMs) with a framework for both client and server, uses a LoginContext class that has life-cycle methods called as part of authentication (two phase commit and chain of responsibility pattern). Authorization – based on the notion of classes being members of protected domains, theads of execution each having an Access Control Context object and an authorisation policy that is enforced by an AccessController.

- ***JCE* (Java Cryptography Extension)** – new for Java 1.2, in Java 1.0 all remote code was un-trusted and ran in a sandbox with no way to get out, in Java 1.1 signed remote code was trusted and unsigned was un-trusted, the signed code could do anything, the unsigned code was in the sandbox again, in Java 1.2 (with JCE) the introduction of Policies allowed code to be restricted to a finer degree, it could widen the sandbox. JCE also introduces interfaces for certificates, in particular the X.509 v3 certificate implementation. Sun provides some encryption algorithms but you can get others from other providers.

- ***JSSE* (Java Secure Sockets Extension)** – defines a set of APIs for using SSL and other technologies like SSL. There's a reference implementation in the packages.

- ***DMZ*** – Demilitarised Zone, zone of computers between private network and outside public network, prevents direct access to private network, typically has the companies web pages.

- ***Firewall*** – The basic security functions of any firewall are to examine data packets sent through the firewall, and to accept, reject or modify the packets according to the security policy requirements. Also they hide the network topology. There are different types:
  - **Packet Filters** – all traffic from un-trusted to trusted areas are passed through the filter. It inspects the packets and rejects or allows them based on rules. The rules typically deal with TCP/IP – TCP contains headers with source and destination ports, and IP contains headers with source and destination addresses. So they can port and address filter.
  - **Stateful Packet Filters (SPFs)** – similar to the above but takes into account some details of the TCP/IP rules. Attackers cannot send packets that fraudulently appear to part of an existing connection. Bigger problem with UDP.
  - **Proxies** – break up connection from client to server. To server they deal directly with client and vice versa. However, a proxy can check legality of data, eg: GET command before passing it on. Also it can increase bandwidth and response times by caching data.

So filters tend to look at protocol level information where as proxies can look deeper at the contents of the data flow. Filters are less memory and CPU intensive and less complex (easier to manage, configure, get audit info from). Could use all three firewall strategies – known as 'defence in depth'.

- ***VPN (Virtual Private Network)*** – constructed on top of a public network but use encryption and other security mechanisms to ensure authorisation and data integrity, handy and cheap to use internet or other existing public networks, tend to be proprietary software/hardware to achieve but new spec IP Security (IPSec) is part of new IP 6.0. Typically there are encryption when data is sent to the public network and decryption when it is removed (software that is part of a company's firewall). The applications that use the VPN as unaware of the encryption going on.

- ***Victim Hosts*** - victim hosts are hosts which face security threat from outsider since their IP addresses are exposed. To prevent this, you need to install firewall or packet filtering router to restrict access from outside (eg: close telnet port, disable broadcast forwarding).

- ***DOS Attack (Denial of Service)*** - Smuf-ing – send PING to Internet broadcast address and route the replies to a victim thus flooding them with useless traffic.

- ***Auditing*** – to know who has been allowed into your system, vital to keep these logs secure.

- ***Load Balancing/Sharing*** – round robin DNS: URL -> list of IP addresses, choose each in turn but need to set life of caches (Time To Line – TTL) low enough to get good balancing but not too low for too much DNS access and make it slow. However, doesn't cope well if one machine dies, it will still route some of the traffic to the dead machine, also no support for server affinity. Because DNS RR doesn't take a metric such as CPU usage into account it is considered to be a load sharing solution. Alternatively, reverse proxy that the DNS points at. It is a single point so easy to configure, easy to get logs, complete control therefore server affinity, copes with crashes but more complex and single point of failure.

  Server Affinity or Sticky Load Balancing – route requests from the same client to the same back-end machine. This is necessary if you use Stateful Session or Entity EJBs. There can be only one valid server to pass subsequent requests to. Load distribution with Server Affinity recognises that multiple server are acceptable targets for requests but it also recognises that some request are best directed to a particular server. Weak affinity – attempts to enforce the affinity, but not always guaranteed. Strong affinity – guaranteed that affinity is respected.

Duties of an Architect
Phases of Development

An Architect

Creates an architecture to meet the service level requirements (the 'ilities') and attempts to make the designers as productive as possible.

Phases of Development and Architects role

1. Requirements Analysis – help define customer needs, build proof of concepts, help with the development of a project plan. Produce a System Requirements Document – use-cases, object diagram, sequence or collaboration, and activity diagrams.

2. Design – the duties of an architect are:

    - Reconcile technical issues against business requirements, therefore need to be business and technology aware.
    - Consider 'ilities' (reliability, scalability, etc.).
    - Consider integration with other systems.
    - Consider security issues.
    - Document architecture. System Architecture Document – class, package, component, deployment and state chart diagrams.
    - Plan how to bring a system into production.
    - Build Prototypes to check meeting user requirements; speed, etc.
    - Plan a migration strategy.
    - Plan training

3. Implementation – manage feedback and update design.
4. Validation – analysis and testing to verify requirements.
5. Deployment
6. Operations and maintenance

1. This is my last woe,
   B2B and B2C – is this something I need to know?
   I'm guessing no,
   Well then it's time to go…