1 Component:

Szyperski's definition of a component: **"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."**

"To enable composition, a software component adheres to a particular component model and targets a particular component platform." They should be "…in an executable form that remains composable."

Solution to the software crisis was envisioned as "software ICs" as far back as 1968 (Doug McIlroy).

Why components? Largely because every mature engineering discipline has used them, and continue to use them.

There is not a general consensus as to the distinction between components and objects. It may be they are merely "a new way of presenting objects to the market" [Allen/Frost, 1998].

Interestingly, in comparison to hardware engineering which has continually revolutionised its product according to Moore's law, there has arguably been no fundamental improvement in software engineering since 1956 (the invention of high level languages).

2 Similarities and differences between Components and Objects

Are there any extra overheads when using one in preference to the other?
Both separate interface from implementation.
Component systems are extended by composition rather than inheritance.
The focus of component based development is on deployment.

As a rule of thumb, the larger the component, the less likely it will be an ideal match to a requirement.

3 Terms

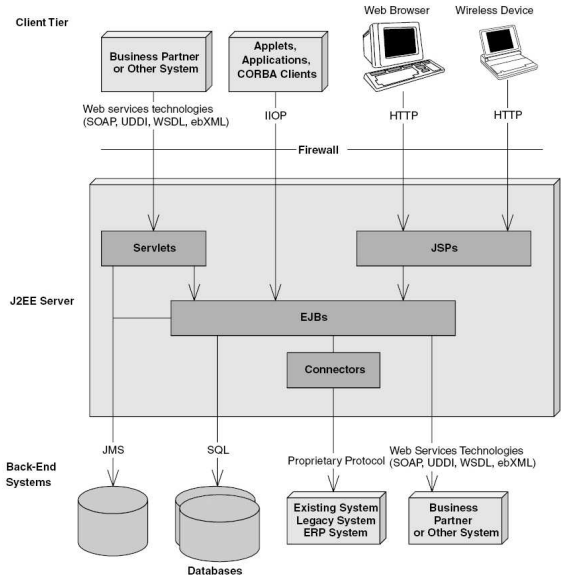
Encapsulation is the gathering together of data and processes to provide or perform a coherent task [me, 2006]
Encapsulation does not in itself have to *hide* what it is doing, but it does need to incorporate what it needs to do whatever it is doing.
'black boxing' is a form of encapsulation, albeit one which also hides the way in which it does what it is doing. *Information hiding* (a term coined by D.L.Parnas in 1971) does encapsulate information, although these terms are not synonymous. What Parnas suggested was that instead of encapsulating the procedural steps, the design decisions be encapsulation instead.
Inheritance is a means of acquiring all the functionality of an object, and extending the functionality of that object. It is hierarchical. If the functionality of the ancestor objects changes, so too does that of its descendents. Multiple inheritance allows functionality from more than one object to be inherited (this is falling out of favour, in place of *Interfaces* which are *implemented* by objects).

Polymorphism is the ability to refer to an object using an ancestor and yet be able to call an appropriate descendent method, by virtue of the ancestor specifying that functionality must be provided, or over-ridden by the descendent. It allows something (e.g. a method) to appear in multiple forms, or for different things to appear the same (single method call result in different code execution).
Messaging, refers to the asynchronous exchange of data between computers, providing service requests and event notification, and allowing for load balancing and distribution of functionality across clusters of computers. Realised through the use of message-driven beans, this allows parts of an enterprise application to be uncoupled.
Abstraction is both a *process* and an *entity*, since it is possible produce an abstraction (the entity) by abstracting (the process). Rather as with flying a rocket away from the earth, we see fewer details and more of the bigger picture – so it is with abstraction.
Substitutability is the principle of substituting a descendent object for a parent object without altering the 'correctness' of an application.
Plain Old Java Objects (POJOs) are classes that don't implement infrastructure framework-specific interfaces. Used with non-invasive frameworks such as Spring, Hibernate, JDO, and EJB 3, which provide services for POJOs, they future proof application business logic by **decoupling** it from volatile, constantly evolving infrastructure frameworks. POJOs are simpler and faster to test and develop, and are more portable, not being tied to any particular framework. POJOs + e.g. Spring provide what EJB2 provides.
Inversion of Control (IoC) relates to the situation where the framework is in control, and calls event handlers which the developer supplies, such as Windows applications. Security is enhanced since it is the framework which is in control.
Dependency injection is similar, and refers to the passing of dependencies via *constructor* arguments, via the *interface* or via *setters.*
Spring is a lightweight container, supporting IoC, annotations, uses XML, integrates with Hibernate. Acting as an *application assembler*, it instantiates, configures and wires up components, handing (i.e. **injecting**) references to other objects. Spring.NET works with .NET. Spring and EJB3 can **complement** each other. **PicoContainer** is another dependency injection framework. Spring sits above the App Server, EJB3 is part of it.
MVC (Model-View-Controller) is an architecture (or *pattern*) that simply separates data model, UI and control logic.
RMI *vs.* RPC a *remote procedural call* is an invocation from a process on one machine to a process on another machine. They provide a simple way to perform cross-process or cross-machine networking. *Remote method invocation* takes the concept one stage further and allows for *object* communications, i.e. it allows object **methods** to be invoked. RMI-IIOP is a CORBA-compliant version of RMI.

Whitebox: user can see the implementation (Glassbox) and manipulate it (Whitebox).
Blackbox: user can only uses interfaces, and cannot see the implementation.
Greybox: part of the implementation is revealed to the user (e.g. can see private members, but not access them).
Personally, I don't like the thought of *greybox* and prefer to stick to Black & White!

Static type checking is performed at compile time, and validates correct use of types.
Late binding is the linking of references to objects at runtime. Also called *dynamic binding* (think DLLs).
Exception handling is the ability to deal with errors (exceptions) at runtime by way of interrupt/callback mechanism.

4 EJBs and the J2EE framework



Benefits of Enterprise beans in the J2EE framework:

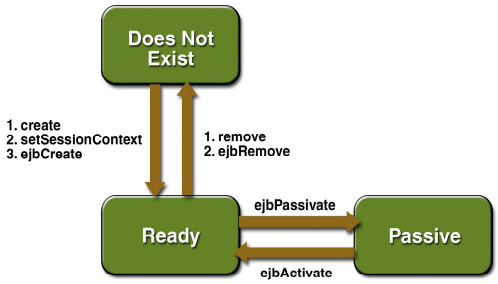
1. **Simplify** development by ceding responsibility to the **EJB container** for system-level services (e.g. transaction management and security authorization). Beans are **managed components**,
2. Clients are **thinner**, and client effort is focused on presentation,
3. Enterprise beans are **portable** to any J2EE server,
4. **Scalability**! Bean location is transparent to the client,
5. **Flexibility**! Multiple types of client can access EJBs.

Session and Entity beans

A session bean encapsulates *processes* or *tasks* or *transactions*, holds application logic and can be thought of as a *personal servant* of a client program; an entity bean encapsultates business concepts such as *product*, or *customer*, holds domain logic (rules and constraints) and models persistent business objects. For example a `CreditCardBean` would be an entity bean, but `CreditCardVerifierBean` would be a session bean.

A client does not directly instantiate beans – this can only be done by the EJB container. The client can request a bean be created or removed; the actual bean methods are called by the container.

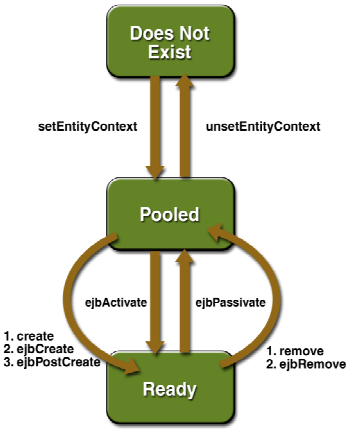
Lifecycle of a **session bean**: can be created and removed by the client. The EJB container can also set a **stateful** bean into a **passive** state,



but a **stateless** bean *cannot* be passivated (consequently, they may offer better performance)

Both the client and the EJB container invoke methods on the bean, but only the container can passivated a stateful session bean.

Lifecycle of an **entity bean**:



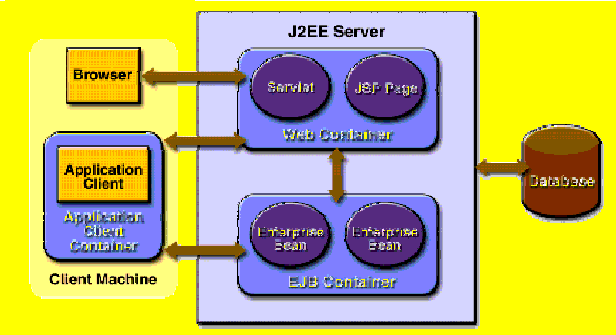
The EJB container creates the instance of an entity bean. After instantiation the bean is in the pooled state and is not associated with any object identity.

The client can **create** and **remove** an entity bean – this just moves it from the ready state to the pooled state.

Also, the container can passivate/activate a bean as required, which also moves it between the ready and pooled states.

Entity beans can be **shared**.

The EJB container is used to deal with system-level responsibilities, replication, load-balancing – general scalability issues. It separates presentation from business logic.



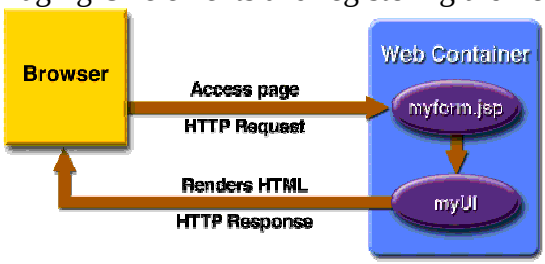
The EJB container creates the instance of an entity bean. After instantiation the bean is in the pooled state and is not associated with any object identity.

The client can **create** and **remove** an entity bean – this just moves it from the ready state to the pooled state.

EJB2s have two interface, home and remote. The **home** interface defines the **lifecycle methods** (e.g. `create` and `remove`), the **remote** defines the **business methods**. EJB3 does not have this distinction.

The **Web container** is the host environment for Java **Servlets** and **JSPs**. **Servlets** service and respond to HTTP requests. Exceptions can result in a 'page not found'-type response. Life cycle: Web container loads the servlet class (if not exists), creates an instance and calls the `init` method, all transparent to the client, and then invokes the `service` method with the user request. Servlets are similar to CGI but are faster, being in Java byte code rather than needing interpreting (e.g. *Perl*) and platform independent. **JSPs** are text documents which contain both HTML/XML/etc (static) and JSP elements, which create dynamic content. JSP elements can be expressed in XML, so the whole JSP document can be an XML file. **JSPs service requests as a servlet**, ∴ lifecycle is the same as a servlet. When a request is mapped to a JSP page, the web container first checks whether the JSP page's servlet is older than the JSP page. If the servlet is older, the web container translates the JSP page into a servlet class and compiles the class. During development, one of the **advantages** of **JSP pages** over **servlets** is that the build process is performed automatically.

JavaServer Faces is a server-side user interface API and 2 custom tag libraries, for managing UI elements and registering their events and validators on the server.



The JSP file includes JavaServer Faces tags. It offers a clean separation between behaviour and presentation. JSP on its own does part of this but cannot manage UI elements as stateful objects on the server, as JavaServer Faces does.

Persistence – by providing transaction functionality (e.g. to a database) the EJB and Web containers allows EJBs etc to exhibit persistence properties without containing SQL. **Bean-managed** persistence requires the bean to handle all the persistence code, but offers more flexibility. **EJB-managed** persistence allows portability between EJB containers/J2EE servers. **EJBQL** provides independence from proprietary SQL.

Naming
Nameservers **resolve** and **bind**. Resolving provides a reference to a component, given a name (think DNS & domain names *vs.* IP addresses). Binding creates a link between a name and a distributed system component. Directory services include X.500, LDAP, DNS, Microsoft Active Directory, and **JNDI** (Java Naming & Directory Interfaces).

The EJB model makes use of RMI & JNDI to allow a client to interact with EJBs.
Java Remote Method Invocation (**RMI**) is a distributed leasing mechanism used to handle the lifetime of an object. An EJB offers its functionality to clients as an **RMI remote interface**. When deployed, its location is registered in the naming service. A client can then use JNDI to look up the location of an EJB, interact with the EJBs home to obtain an instance, and then use the business functionality offered by the EJB. Note that JNDI is an API and not a service in itself. JNDI is a service-independent interface to many directory services such as those listed above.
Note that RMI is **tightly-coupled**, whereas messaging is **loosely-coupled**. The **Java Message Service** (JMS) is both *asynchronous* and *reliable*.

Advantages/disadvantages of **client-server** *vs.* ***n*-tier** architecture:
Client-Server is just a 2-tier system. Simpler to implement than *n*-tier, but not as scalable, flexible or resilient. Easier to distribute (fewer moving parts). However, **distributed systems** have several **advantages** over single-server ones, namely scalability, fault-tolerance, load-balancing, heterogeneity, resource sharing. Not all systems are distributed by nature – some are restricted by h/w or physical location.

Various **layers** of an ***n*-tier** architecture include: presentation (client-side), business application (session beans), domain logic (entity beans, e.g. banking,), persistence (data access).

5 EJB3 *vs.* EJB2.x

EJB 3.0 is **backward compatible** to EJB 2.x clients.
XML **deployment descriptors** can be replaced with **annotations** (e.g. `@Id`, `@TransactionAttribute` and `@SecurityDomain`). (e.g.).
Stateless beans are tagged with `@Stateless`; stateful beans are tagged with `@Stateful`.
Entity beans are POJOs.
The CMP (Container Managed Persistence) has been simplified in EJB3 and is now more like Hibernate or JDO.
In EJB3 methods like ejbPassivate, ejbActivate, ejbLoad, ejbStore, etc. are no longer required
Interceptors are new in EJB3 and allow developers to intercept lifecycle and business methods.
Dependency injection is new in EJB3.
Inheritance is now allowed in EJB3, so beans can extend base code.
EJB3 includes **Object Relational Mapping** (**ORM**).
EJB3 no longer needs home and remote interfaces
The native SQL queries are supported as an EJB-QL (Query Language) enhancement.

★ With EJB3, to enable **clustering** for a bean, use the `@Clustered` annotation on the bean class. A clustered bean has load-balancing and failover of a request. Stateful beans also need to have their state replicated across the cluster.

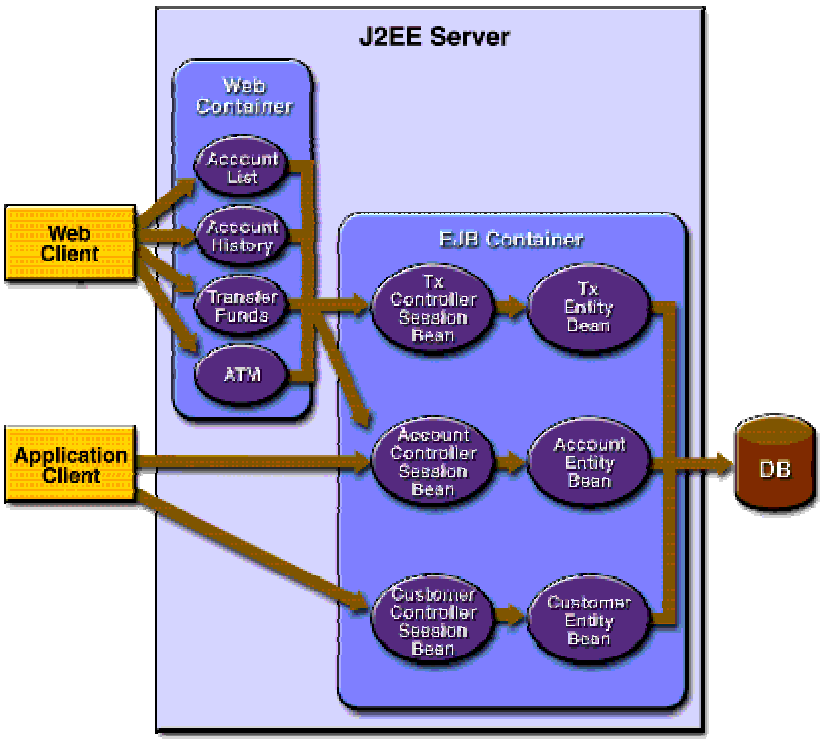
A J2EE application is typically deployed as an *EAR (Enterprise Application Archive)* file, containing all the necessary *WARs (Web Application Archive)*, *EJB-JARs (EJB Jars)* and *client JARs*. Each component has a *deployment descriptor* (an XML file) which defines its configuration and requirements.

6 CORBA & .NET

Common Object Request Broker Architecture (**CORBA**) and Microsoft's **.NET** are two more distributed architectures. CORBA is run by the Object Management Group (OMG), J2EE is linked to Sun but runs on many platforms and .NET is tied to Microsoft (.NET framework).

The middleware is simply an object bus – orchestrating calls to different services. Object Transaction Monitor (**OTM**).

7 Example application:



This diagram illustrates a 4/5-tiered architecture: client / web-server / application logic / business logic / persistence. The J2EE server includes both a web container and an EJB container. Client applications are both standalone and browser-based. Transactions are controlled by session beans; interactions with core business data is via entity beans.