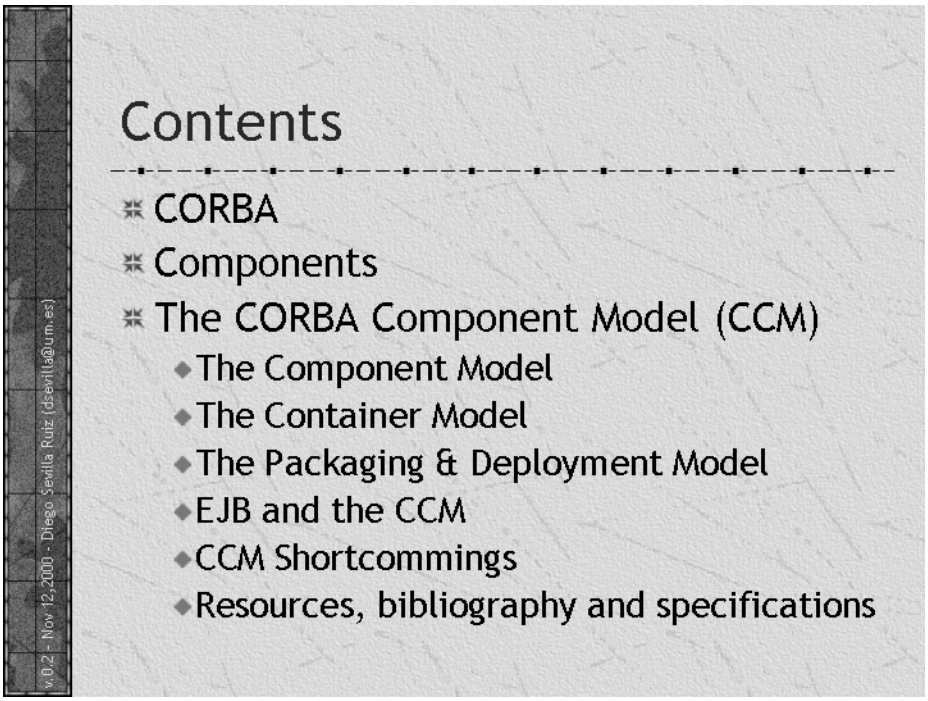
This document is an introduction to the new Component Model of CORBA 3. The document corresponds closely to the transcription of the talk on the CCM (the *CORBA Component Model*) I gave at the "Primer Taller de Ingeniería del Software basada en Componentes Distribuidos" (IScDIS'2000) (http://webepcc.unex.es/~juan/iscdis00) held within the "V Jornadas de Ingeniería del Software y Bases de datos" (http://jisbd.infor.uva.es) at Valladolid, Spain on November 9, 2000.

This document will be always available at http://www.ditec.um.es/~dsevilla/ccm/, "*The CCM Page*".

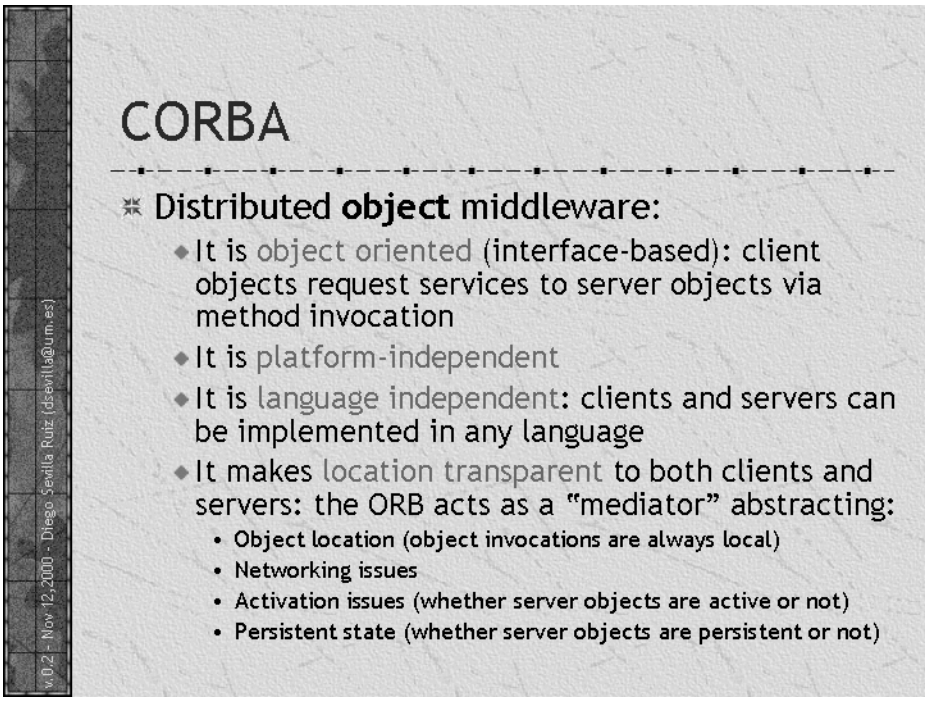
Slide version: 0.2 (November 12, 2000)

This document version:
`$Revision: 1.3 $` (`$Date: 2000/12/27 10:25:27 $`).



These are the contents:

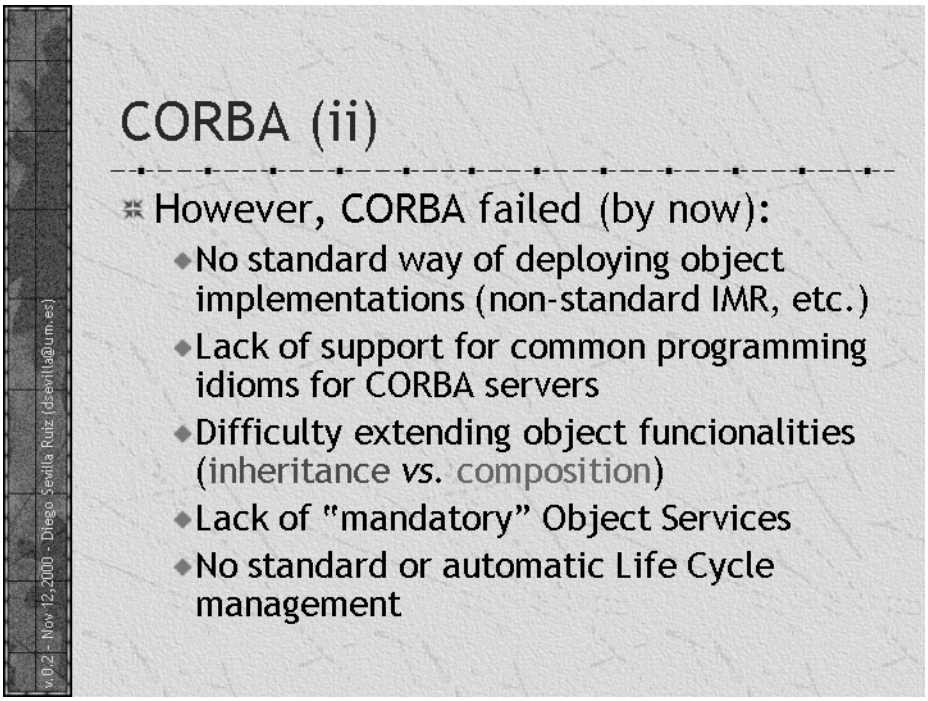
- First, I outline the basics of CORBA, showing its strong and weak points, and what has to be improved.
- Next, I describe what I understand as a component. This first definition of a component will be guided by a set of requirements we impose to software development in general, requirements that lead to the concept of "component".
- Finally, the real body of this document is the introduction to the new *CORBA Component Model* (CCM) of CORBA 3, which is supposed to mitigate all the actual CORBA drawbacks.



CORBA stands for *Common Object Request Broker Architecture*. CORBA's principal strength is that it is an *object oriented middleware*, that is, there is a clear distinction between the interface and the implementation. Moreover, once the interface is specified using the *Interface Definition Language* (IDL), the implementation can be done using any programming language for which a mapping exists (mappings exist for nearly all languages). This is also true for clients of these services, that can also be written in any programming language.

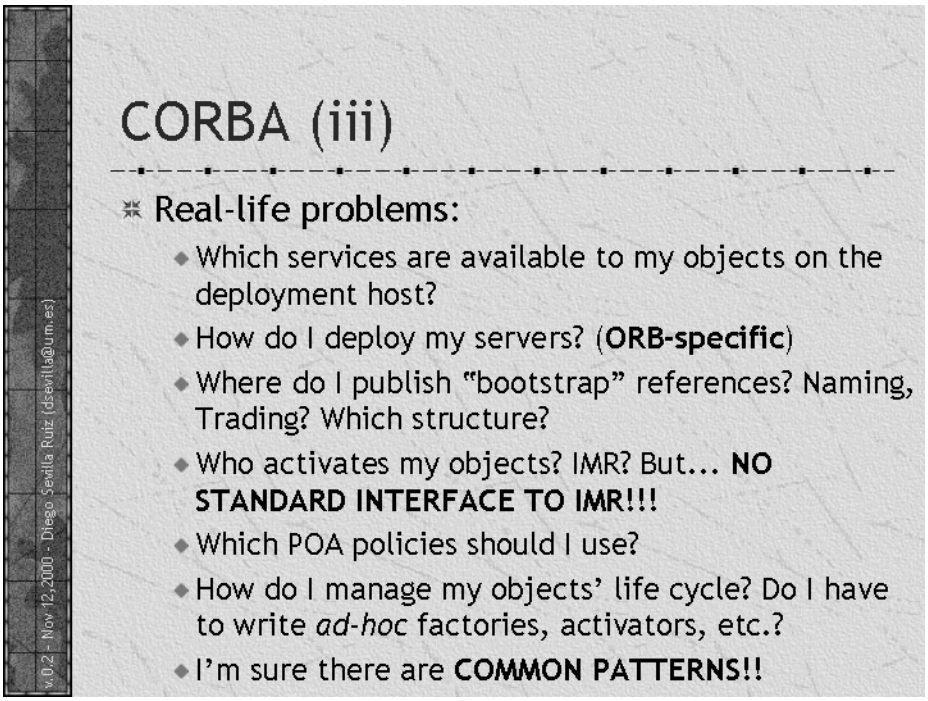
Through the use of IDL, CORBA allows writing platform and language-independent code. As a middleware, CORBA acts as a mediator between clients and servers. Its core, the *Object Request Broker* (ORB), abstracts:

- Object location. Method invocations to server objects are always local. The ORB is in charge of redirecting them to the right destination.
- Networking issues. The programmer must not care of networking issues at all. The ORB will use the network (if necessary) to reach the server object.
- Activation issues. Server objects are automatically activated.
- Persistent state. The client does not care if server objects are persistent or not.
- etcetera.



However, CORBA still has some problems that haven't been addressed successfully. For example:

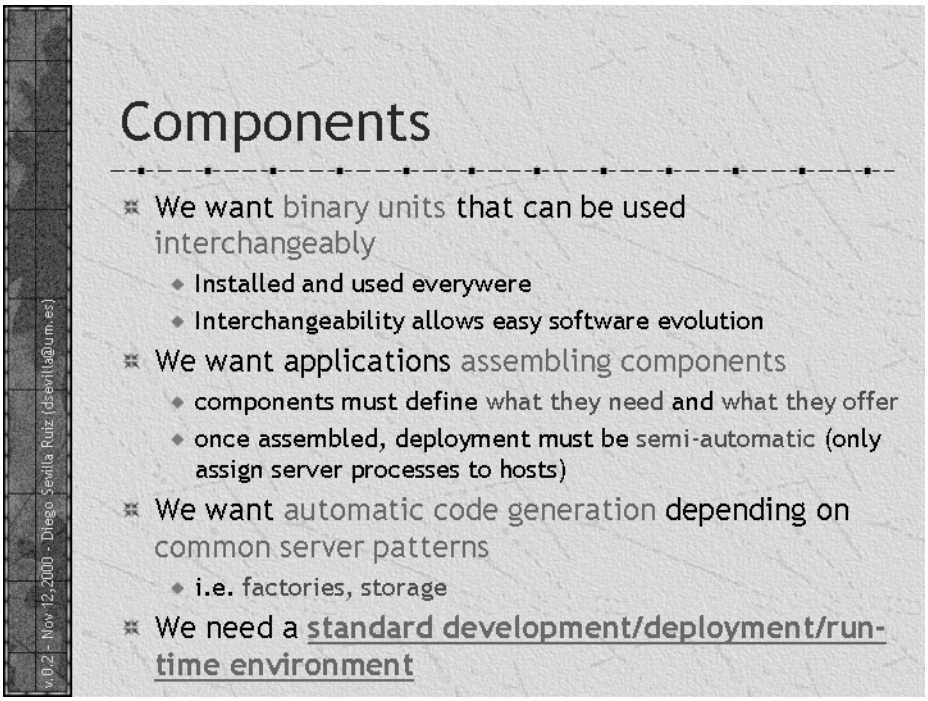
- There is not a standard way of adding new servers to a CORBA system: executable file? Dynamic libraries? How are them initiated? Each ORB has a different support of the IMR (*Implementation Repository*), etcetera.
- Although almost every CORBA server implements a fixed pattern: the factory one, programmers need to write their factory code again for each new type of object. All factories have similar requisites: persistent or transient references, objects identified or not by a primary key, objects with persistent state or not...
- There is no automatic life cycle management.
- It doesn't exist a common set of object services that ALL ORBs implement by default.



In real life, a programmer faces a series of problems with CORBA:

- What services will be available to my objects at run-time? Does it depend on the ORB being used?
- How do I decide and specify where each CORBA server goes? And what if the topology changes? Must I then modify and even recompile my application?
- How do I publish initial references to my servers? Should I use the naming service or the trading service? What structure? Does it depends on each program? Do I have to write it for each new application?
- Who activates my server programs and when are they activated?
- What POA policies should I use? Do I have to repeat the POA-creating and object management for "similar" POAs?
- Do I have to re-write factories although they have similar behavior?

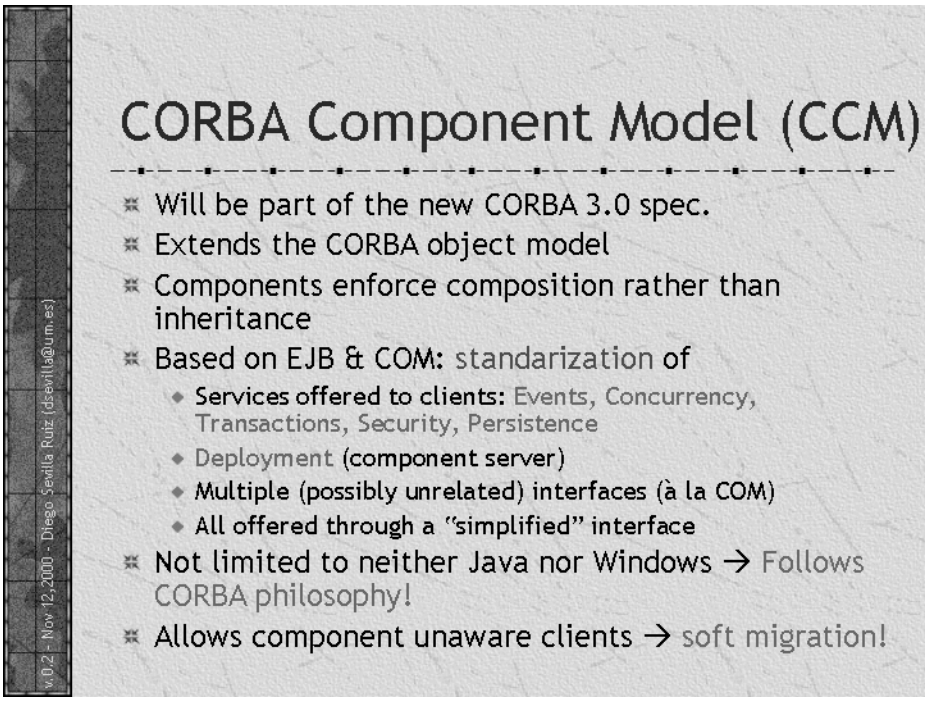
It seems obvious that there are certain common patterns that could be used...



The solution comes by using component technology. Within the software community, it seems that it's not clear what a component is. In other words, the word "component" has been used to name different things, as it is a very generic word.

In this document we are going to define components attending to the requirements they must address. Thus, we want:

- binary *interchangeable* units that could be installed everywhere. Interchangeability enables graceful software evolution, as you can substitute one component by another more efficient (or more complete).
- building applications assembling components. Components must state what do they need and what do they offer. Moreover, the deployment of the applications must be flexible enough to allow both initially distributing components among the available machines and subsequent adaptation to changes in the topology through the application's lifetime.
- automatic code generation based on common design and implementation patterns: code for factories, assembly code, code for storing and retrieving persistent objects' state, etc.
- summing up, **a standard development, implementation and deployment environment covering the whole application's deployment cycle**.



The CORBA Component Model (CCM) is a proposal to mitigate all this problems. This model is part of the not-yet-released CORBA 3.0. Components extend the CORBA object model, meaning that components are a new meta-type in CORBA extending the `interface` meta-type, inheriting all its features and introducing others. These new features will be studied in this document.

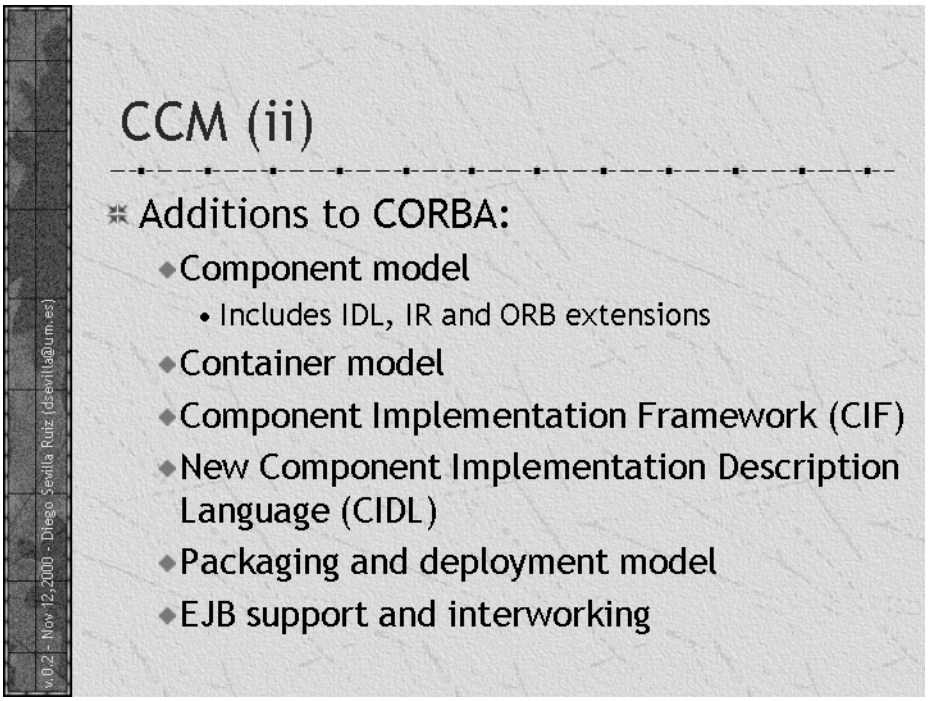
Components enforce composition rather than inheritance. Inheritance ties a class to its ancestors and clients without explicitly exposing its dependencies.

CCM is based on EJB (*Enterprise Java Beans*) and COM (*Component Object Model*), that is, in the standarization of:

- Services offered to component implementations
- Deployment (through a Component Server)
- Multiple (possibly unrelated) interface support (as in COM)
- Everything offered through a *simplified* interface

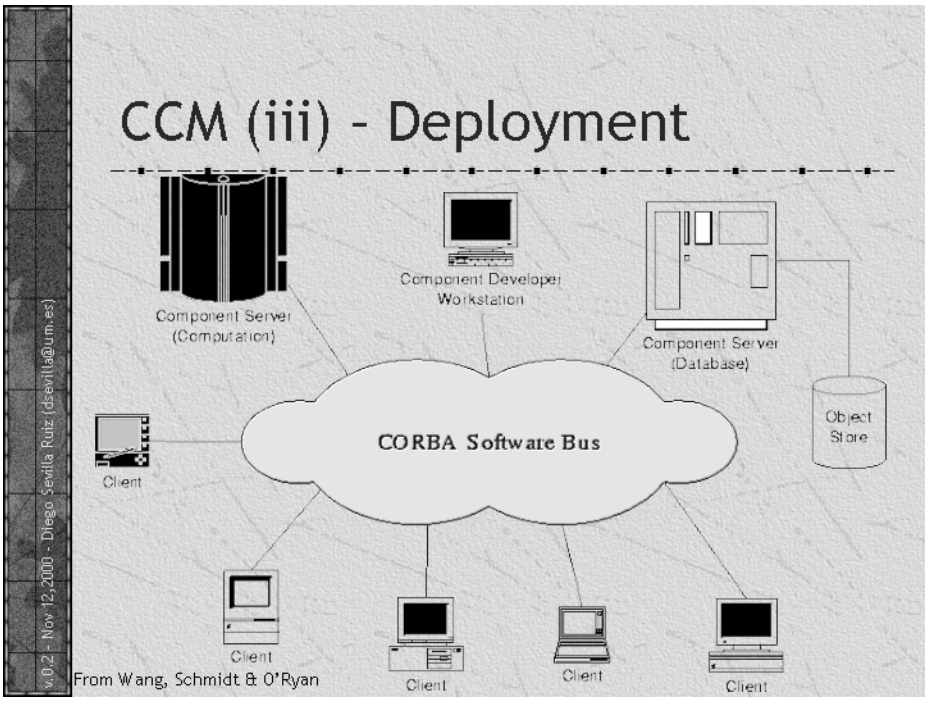
Moreover, it is not limited to neither Java (as EJB) nor Windows (as COM): it follows CORBA philosophy.

It also supports clients that are not component-aware, allowing a soft migration of previous CORBA systems to the new component technology.

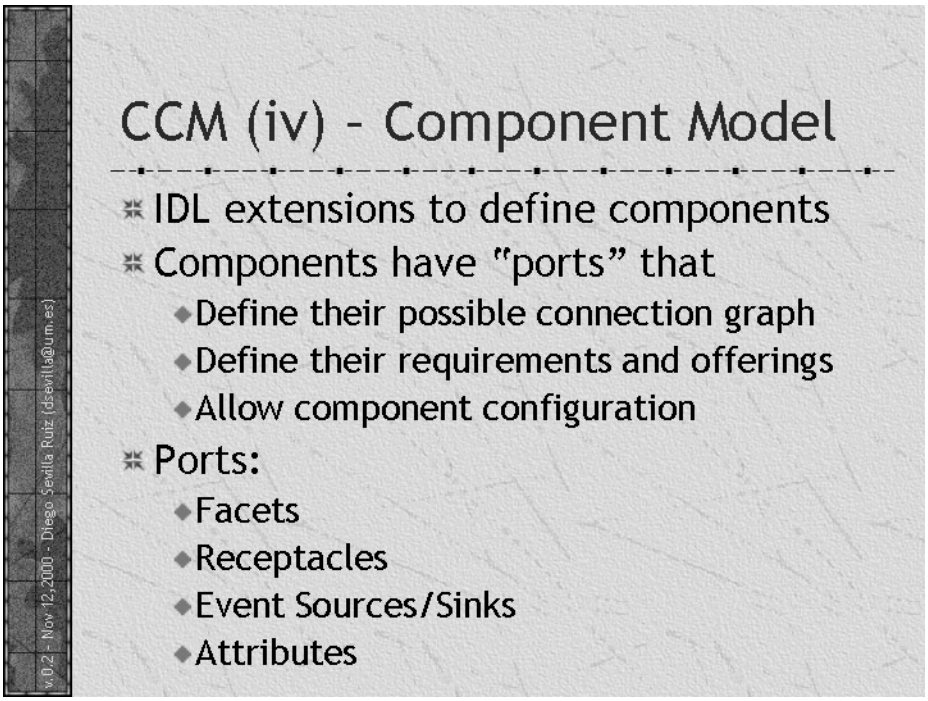


CCM and CORBA 3 add some features to CORBA we'll see just next:

- The Component Model, including IDL, IR (*Interface Repository*) and ORB extensions
- The Container Model
- The Component Implementation Framework (CIF)
- The new Component Implementation Description Language (CIDL)
- The Packaging and Deployment model
- EJB support and interworking

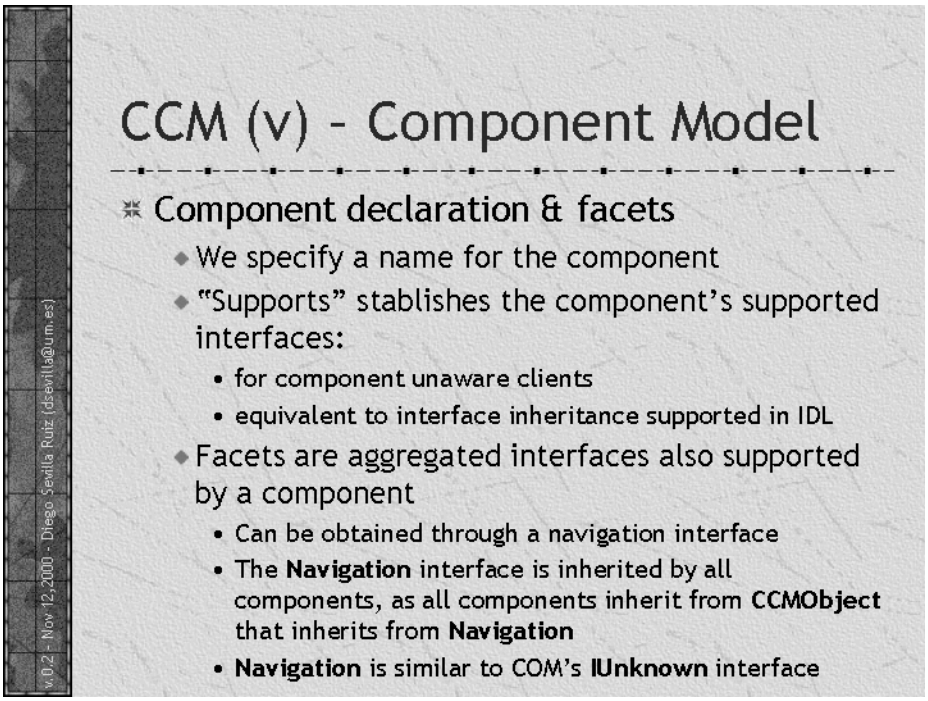


This is the architecture of a CCM application: there are one or more component servers and some client workstations. Component servers store components and perform the work required by clients. As you can see, there is a clear distinction between clients and servers, as in EJB.



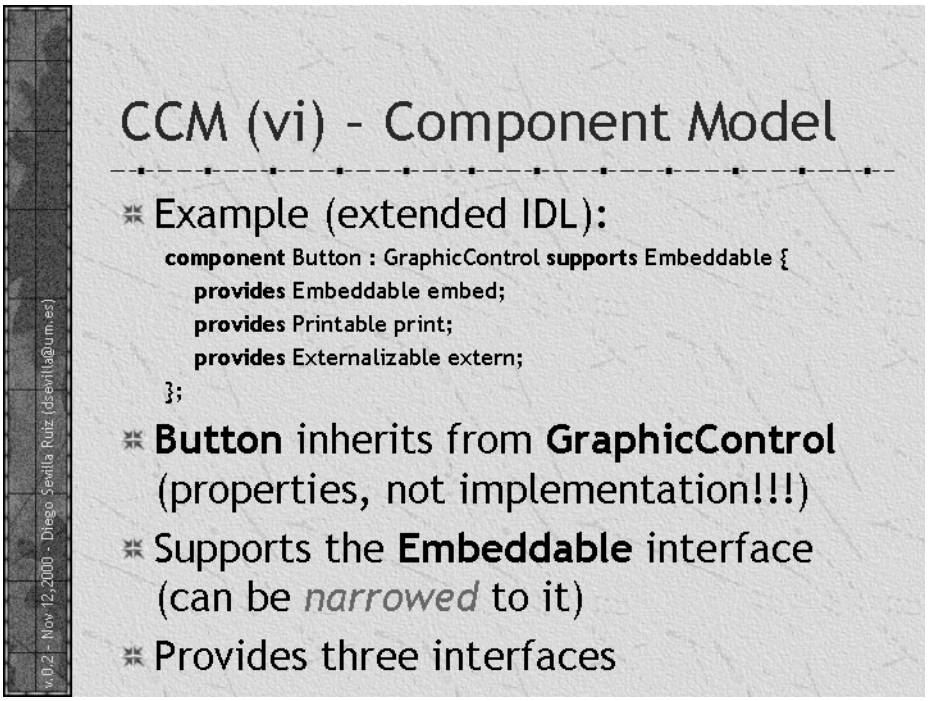
We start looking at the Component Model. It specifies observable features of components. CORBA 3 defines some IDL extensions that allow the definition of components. Component have "ports" that allow them to communicate with the outside world, to communicate with other components, and allow them to be configured.

There are several kind of ports: *Facets*, *Receptacles*, *Event sources/sinks* and *Attributes*.

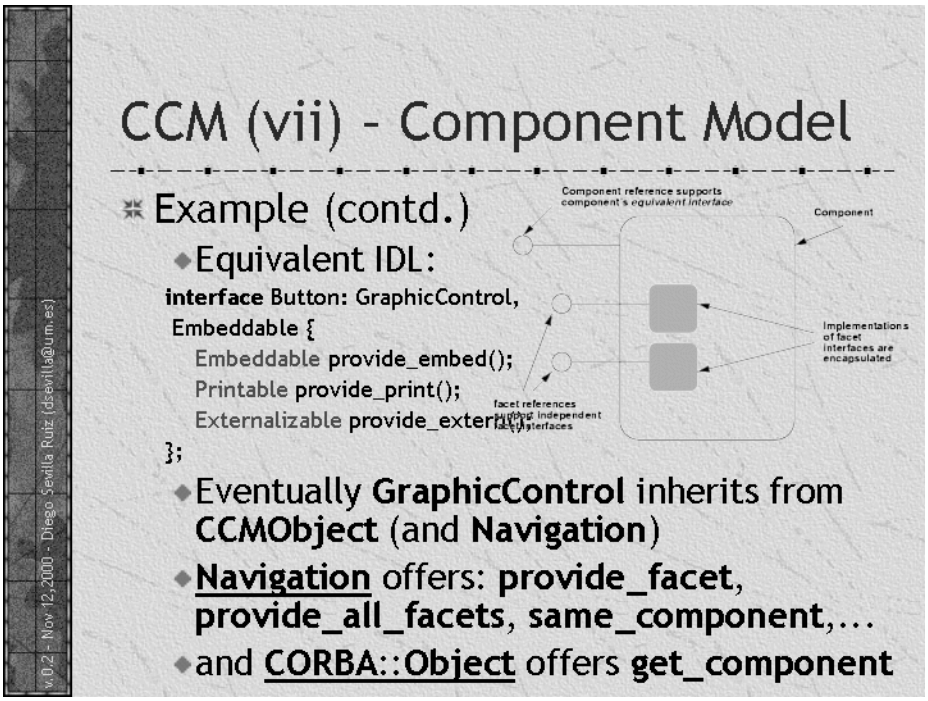


Each component has a name (the same as interfaces, remember that component's meta-type inherits from interface's), and a set of **support**'ed interfaces. Again, equivalent to the interfaces an interface inherits from. The set of supported interfaces allows component-unaware client to use this component (and its set of supported interfaces).

For component-aware clients, components define a set of interfaces they implement, called *facets*. You can ask for facets, obtain all facets a component implements, ... These functions are provided by the `Navigation` interface, from which all components inherit (they all inherit from the interface `CCMObject`, and this one inherits from the `Navigation` interface). `Navigation` is similar to the `QueryInterface` method of `IUnknown` for COM components.

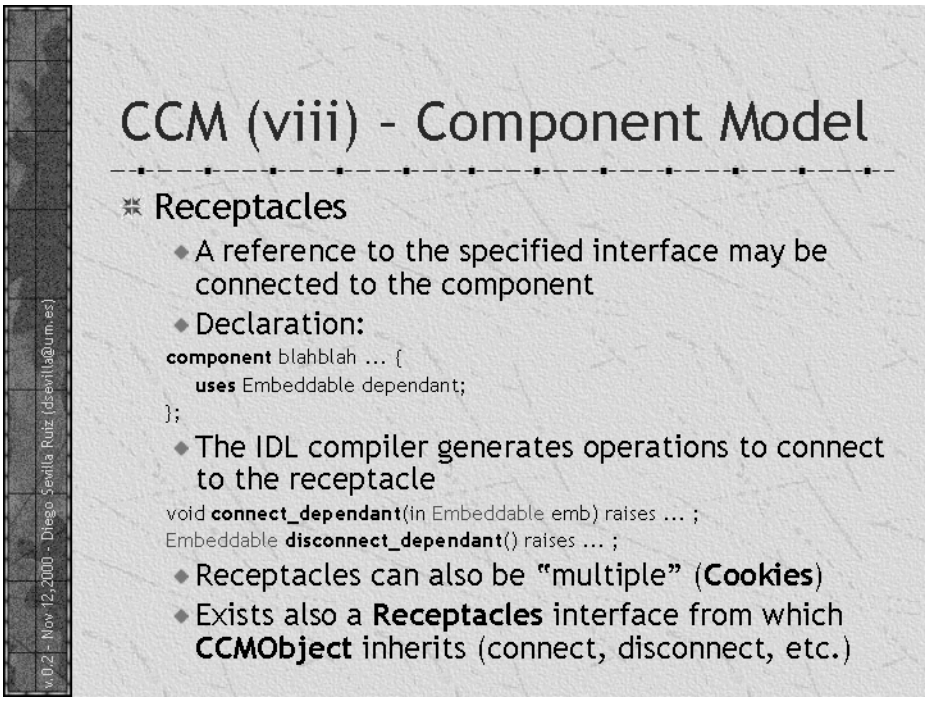


An example: The `Button` component inherits from `GraphicControl` (that is, it has the same properties), and also supports the `Embeddable` interface. Clients that are not aware of components, can still `narrow` this component instance to the `Embeddable` type. For component-aware clients, it offers a different set of interfaces: `Embeddable`, `Printable` and `Externalizable`, perhaps to be included in some predefined component framework.



The equivalent IDL is shown in the slide. You can also see a figure representing a CORBA component, in which different facets can be implemented using CORBA objects inside the component. Note how the operations `provide_<interface>` have been generated to return the reference to each facet type. The `CORBA::Object` interface, from which all CORBA objects inherit, offers a new operation, `get_component`, returning the component a facet belongs to.

The `Navigation` interface offers operations to ask for component's facets in a generic way, with operations such as `provide_facet`, `provide_all_facets`, etc.



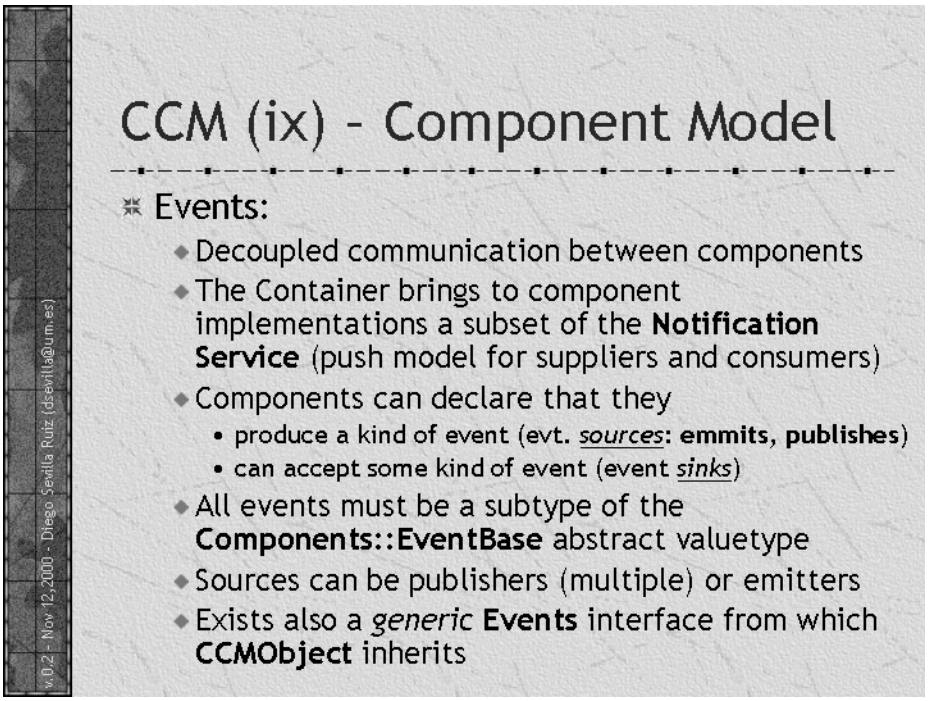
Receptacles represent connection points between components. An instance of the required component type can be "plugged" into the receptacle a component has provided for it. Thus, receptacles represent an explicit client relationship between components.

The component `blahblah` shown in the slide needs an instance of `Embeddable` for its internal function (that is, an instance of this component may use the provided `Embeddable` instance within its lifetime).

As you can see in the slide, the IDL compiler generates two operations for each "`uses`": `connect` and `disconnect`. This connection will be performed at assembly time.

Receptacles can also be multiple, representing a 1:N relationship in which a component uses a series of other component instances. In this case, a `Cookie`-based identification mechanism is used to address the problem of object equivalence in CORBA.

Same as `Navigation` for facets, a `Receptacles` interface (from which `CCMObject` inherits) is provided for dealing with component's receptacles in a generic way.



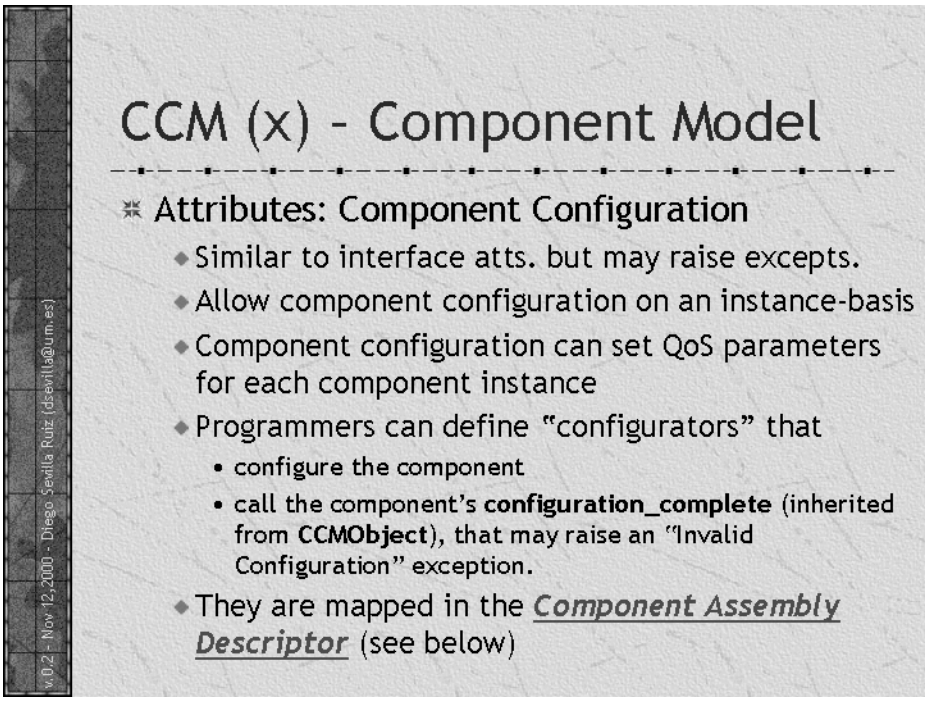
Components can also produce or consume a certain kind of event. Events represent another way of connecting components in which the producer is de-coupled from the consumer. A third entity (the event channel) is in charge of contacting the set of consumers of each event. Receptacle communication was direct, in contrast. As known, an event channel is adequate for at least two reasons:

1. It leverages the producer of managing the set of consumers demanding its events (the producer would have to implement the `connect` and `disconnect` operations, and to enter a loop to send the events to the consumers).
2. In distributed environments, contacting each consumer can be prohibitive in time, managing unilateral consumer disconnection, etc. The component implementation producing the events delegates on the event channel the communication with its consumers. The channel can implement optimized group communication (for example, using multicast protocols, etc.)

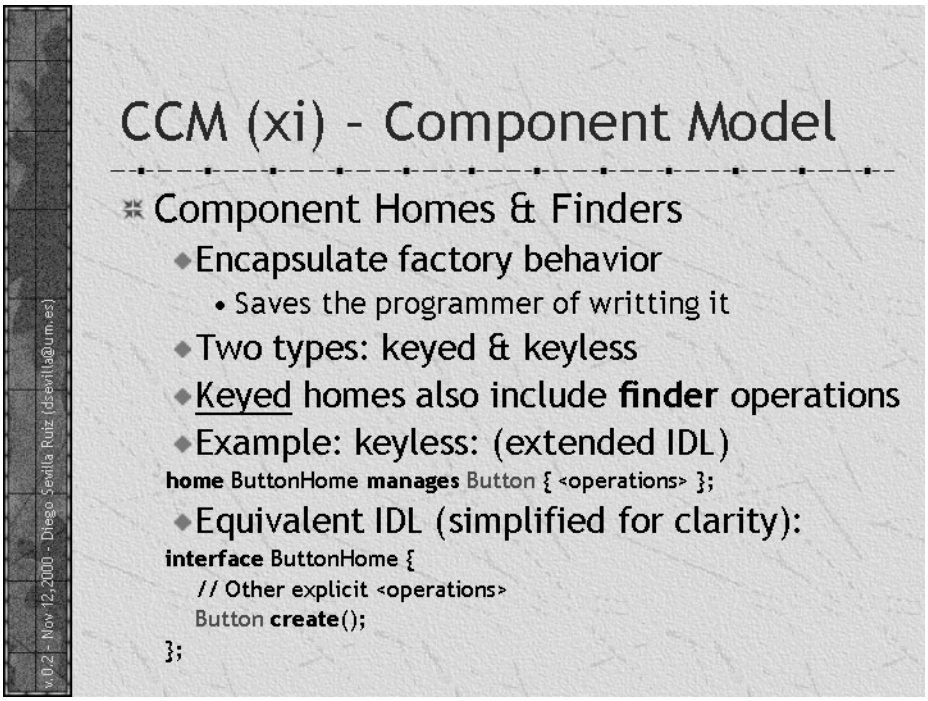
The container offers a notification service to component implementations, in which both producers and consumers are in *push* (active) model.

In their IDL, components can declare that they produce some kind of event (*event sources*, that can be *emmitters* if a 1:1 communication is allowed, or *publishers* if the channel allows 1:N communication) or that they consume certain kind of event (*event sinks*).

All event types must be a sub-type of the abstract valuetype `Components::EventBase`. As before, `CCMObject` also offers a `Events` interface for generic event connection management.

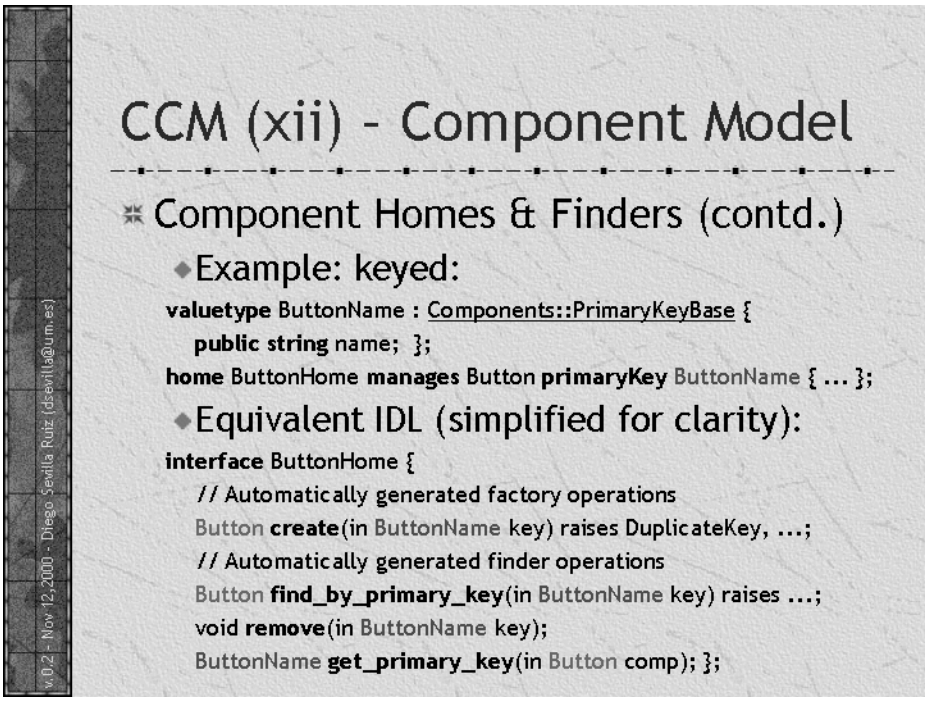


Attributes allow component configuration at assembly time (just before running). They are similar to interface attributes, but can raise exceptions. Programmers may create "configurators", objects that configure a component: they establish attribute values and call the `configuration_complete` method (inherited from `CCMObject`) to signal when the configuration has been completed. The component may raise the "Invalid Configuration" exception if the attribute values are not valid or achievable. The attributes, jointly with their possible values and types, are specified in the *Component Assembly Descriptor*, that we'll see later.

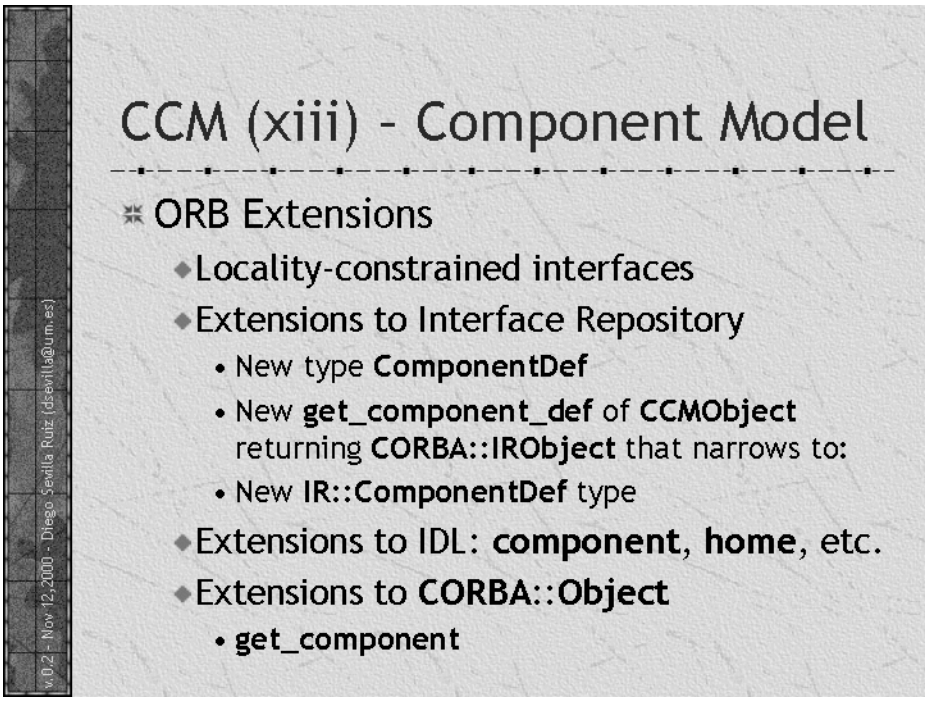


As we stated in the introduction, the factory pattern is very common in CORBA. "`home`" declaration describes a factory (whose code is generated automatically by the framework (CIF) depending on the component category) that can create objects identified by a primary key (*keyed*) or not (*keyless*).

The slide shows an example factory for keyless objects.

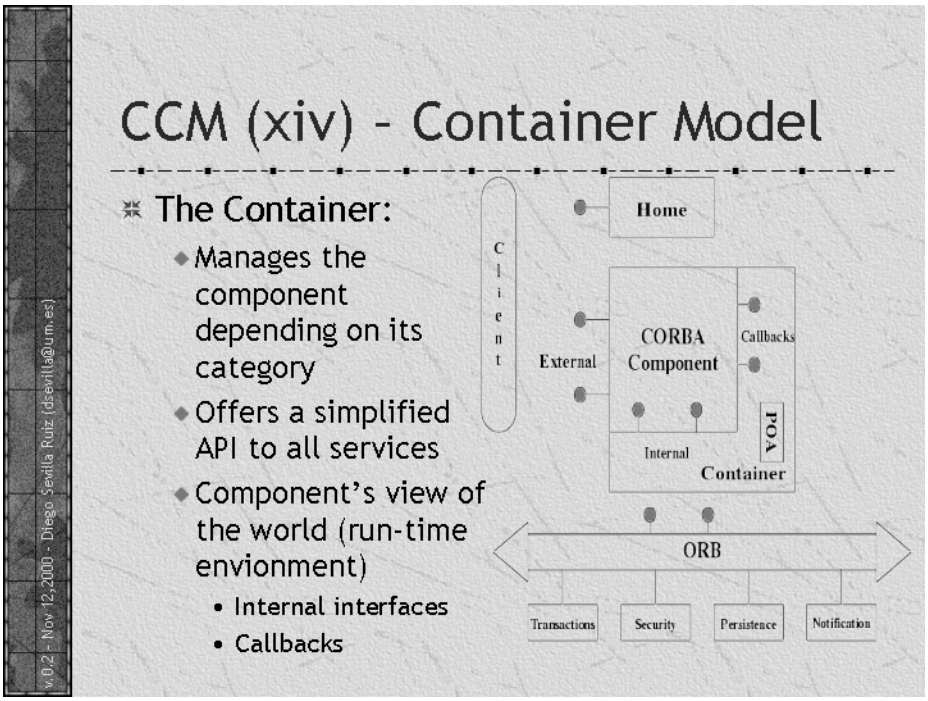


The slide shows an example of factory which manages a set of objects identified by primary key. The primary key must be a *valuetype* inheriting from the abstract valuetype `Components::PrimaryKeyBase`. Keyed factories offer *finder* operations for finding an object given its primary key (for example, `find_by_primary_key` in the slide.)



The component model introduces some extensions to CORBA:

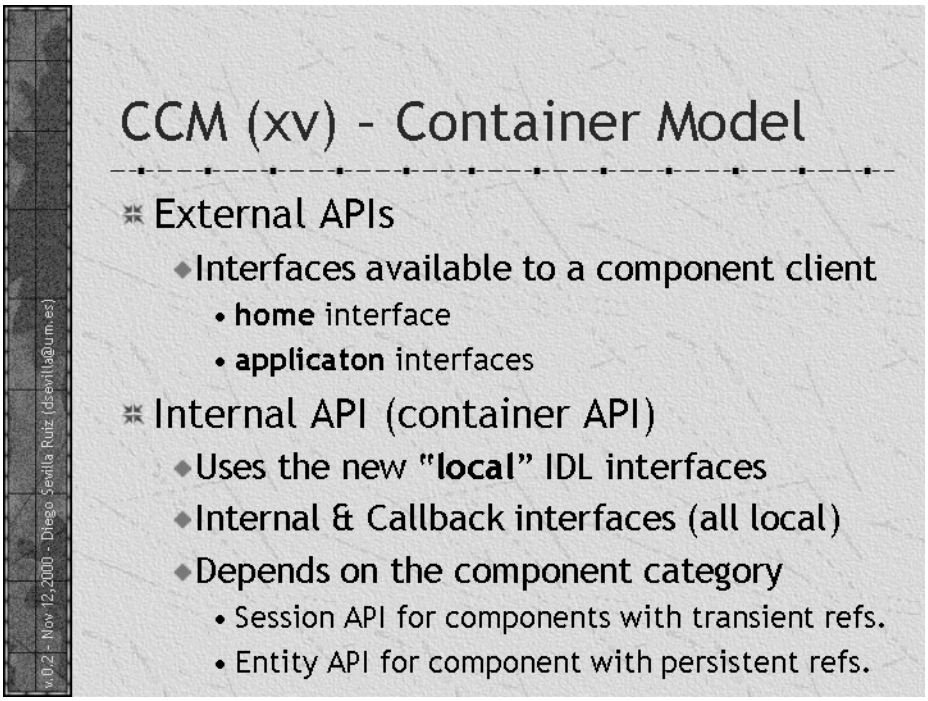
- locality-constrained interfaces: cannot be send through the network (cannot be marshaled). Are used for co-located communication (for example, between the component instance and the container).
- As a new meta-type has been added, the Interface Repository (IR) must be augmented with a new type: `IR::ComponentDef`. `CCMObject` must also support the `get_component_def` operation.
- The IDL has been augmented with new keywords: `component`, `home`, etc.
- Finally, the `get_component` operation has been added to `CORBA::Object`.



The *Container* manages component instances depending on its *category*. It offers all its services through a simplified API: it becomes the component's view of the world. The container offers a series of local interfaces (internal interfaces) for establishing component's context and allows component instances to implement a series of *Callback interfaces* (also local).

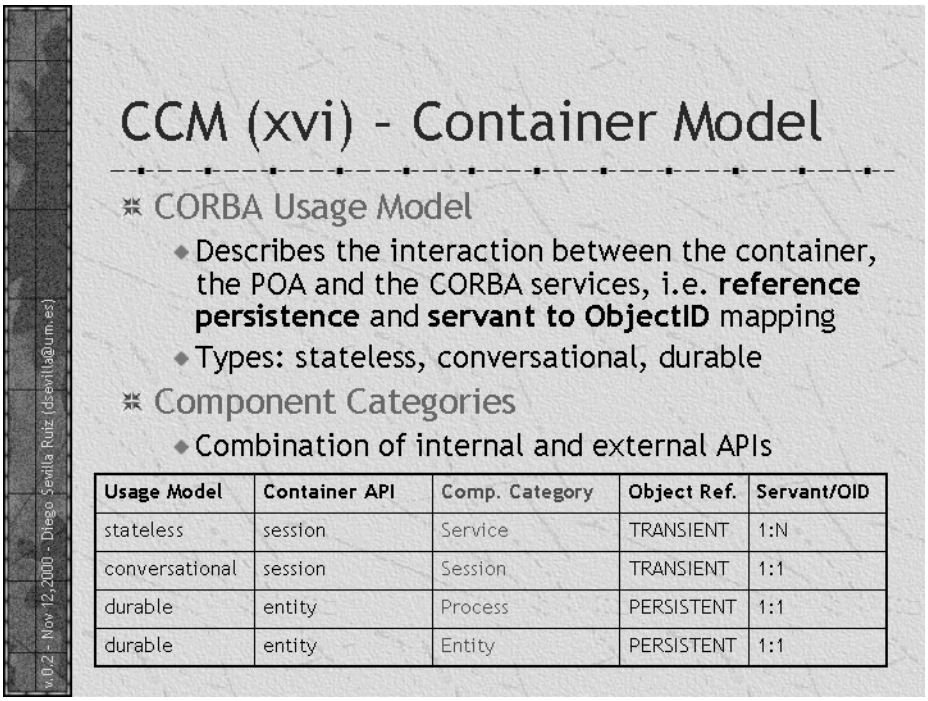
As shown in the slide, each container manages a component. It implements a POA with the component's requested features (more on this later).

Clients can use the interfaces offered by the component instance (*External Interfaces*) and the *Home* interface.



The type of internal API offered to components (*Container API*) depends on **component category** (more on this later):

- *Session API* for components with transient references
- *Entity API* for components with persistent references (components which have persistent identity)



Usage Model	Container API	Comp. Category	Object Ref.	Servant/OID
stateless	session	Service	TRANSIENT	1:N
conversational	session	Session	TRANSIENT	1:1
durable	entity	Process	PERSISTENT	1:1
durable	entity	Entity	PERSISTENT	1:1

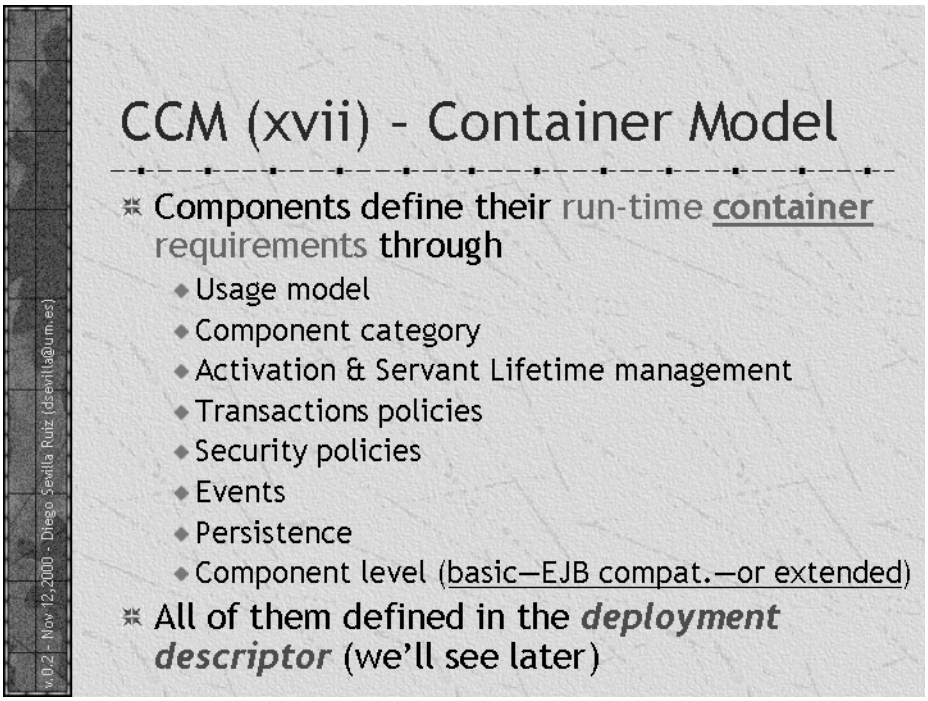
The *CORBA Usage Model* is an abstract description of the relation between the container, the POA and the different CORBA services. For example, it defines whether the component is going to use persistent references or not, or what's going to be the mapping between *servant*s and *ObjectId*s. CORBA Usage Models:

- **stateless**
- **conversational**–state tied to the conversation with a given client.
- **durable**–it outlives any given client.

The combination of internal and external APIs jointly with the CORBA Usage Model define four **component categories**. Components must specify which category they adhere to:

- **Service**–Models a service without state, similar to a Web server.
- **Session**–Models a conversation with a client. Component's life is tied to the session established with the client. A shopping cart could be an example of Session component category.
- **Process**–Models business processes which have their own lifetime within the lifetime of the application. For example, each process of a Workflow application could be a component of this category. Each process must be taken in account by several entities till its end.
- **Entity**–Models persistent entities relevant to the application domain being modeled. They outlive the application's lifetime by definition, and are created and destroyed explicitly.

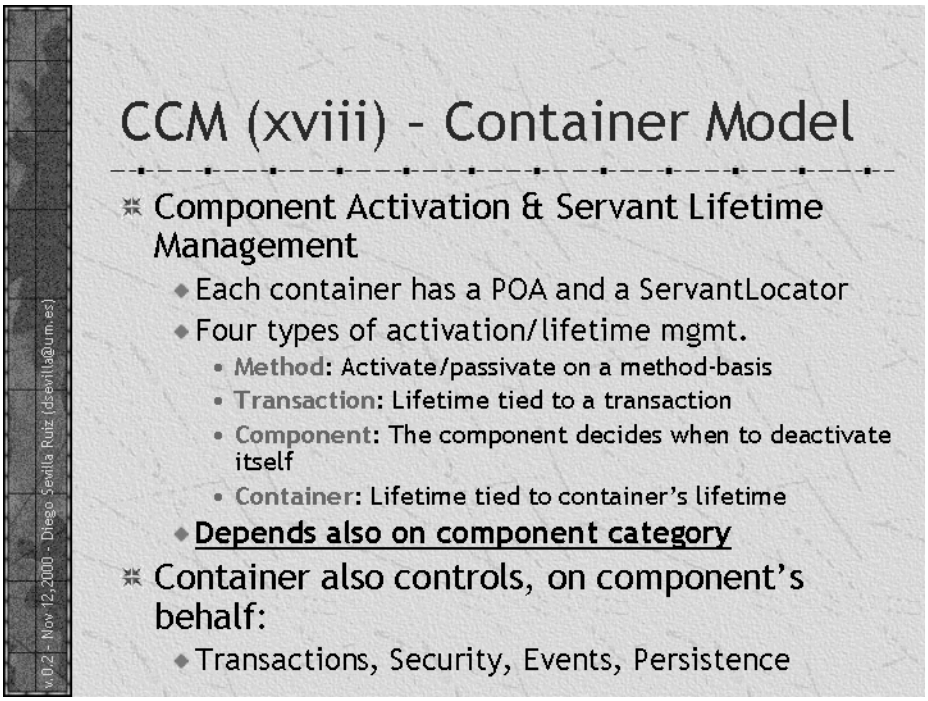
The *Session* and *Entity* categories have been designed to be conceptually equivalent with EJB 1.1 component types.



Components state their run-time container requirements through a set of indicators. We've seen some of them, and we'll see others next:

- CORBA Usage Model
- Component Category
- Activation & Servant lifetime management
- Transaction policies
- Events
- Persistence
- Component level: basic (EJB 1.1 compatible) or extended (supports the whole component model)

All of them defined in the *Deployment Descriptor*, that we'll see later.

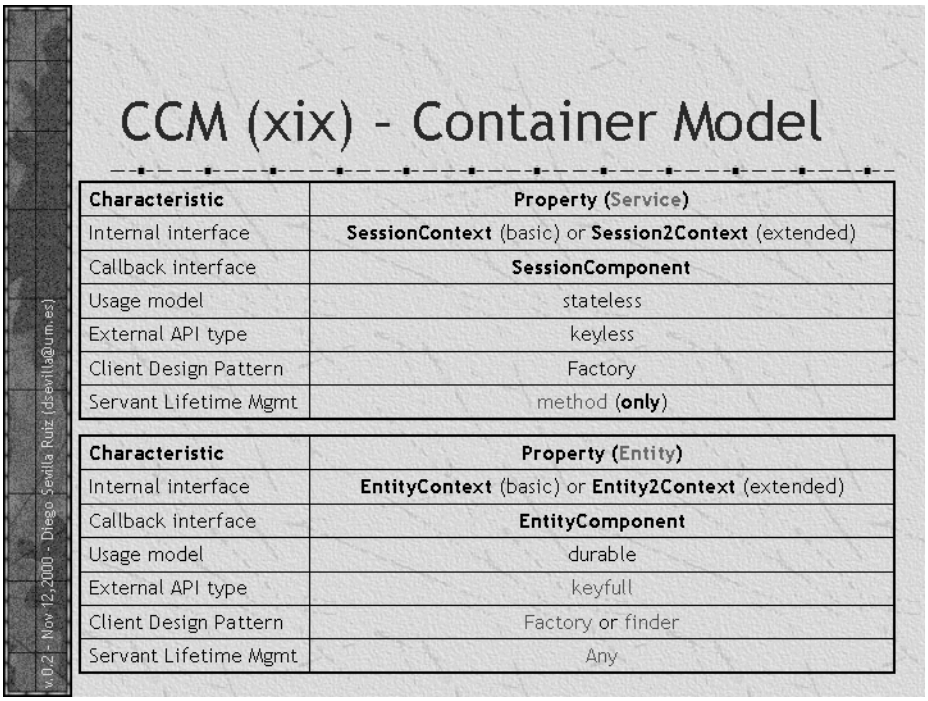


As shown before, each container maintains its own POA (*Portable Object Adaptor*) and implements a *Servant Locator*. Thus, the container must activate and deactivate component instances when needed according to component requirements. The component can specify what activation and servant lifetime management policy requires for its instances:

- **Method**–The component instance is activated before and deactivated after each method invocation.
- **Transaction**–The lifetime of the instances is tied to the transaction they belong to.
- **Component**–The component instance decides when to deactivate itself (obviously, it must tell the container to do so in its behalf).
- **Container**–The instance's lifetime is the same than the container's.

Note that component category also limits the values this policy can hold. For example, *Service* components can only choose a *Method* policy value, as expected.

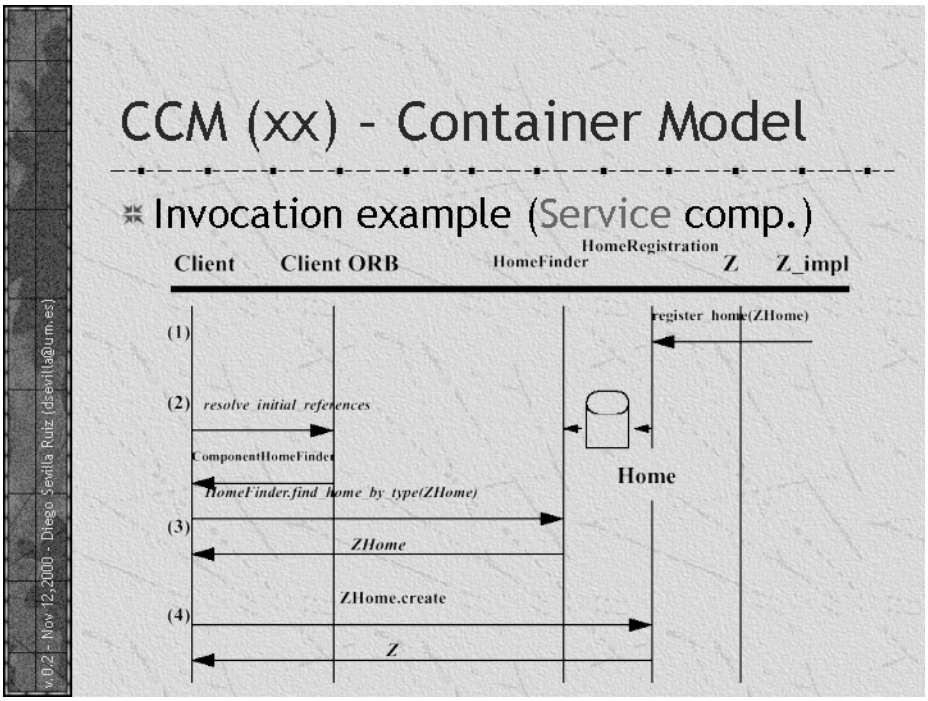
The container also controls, on component's behalf: transactions, events, persistence, security, etc.



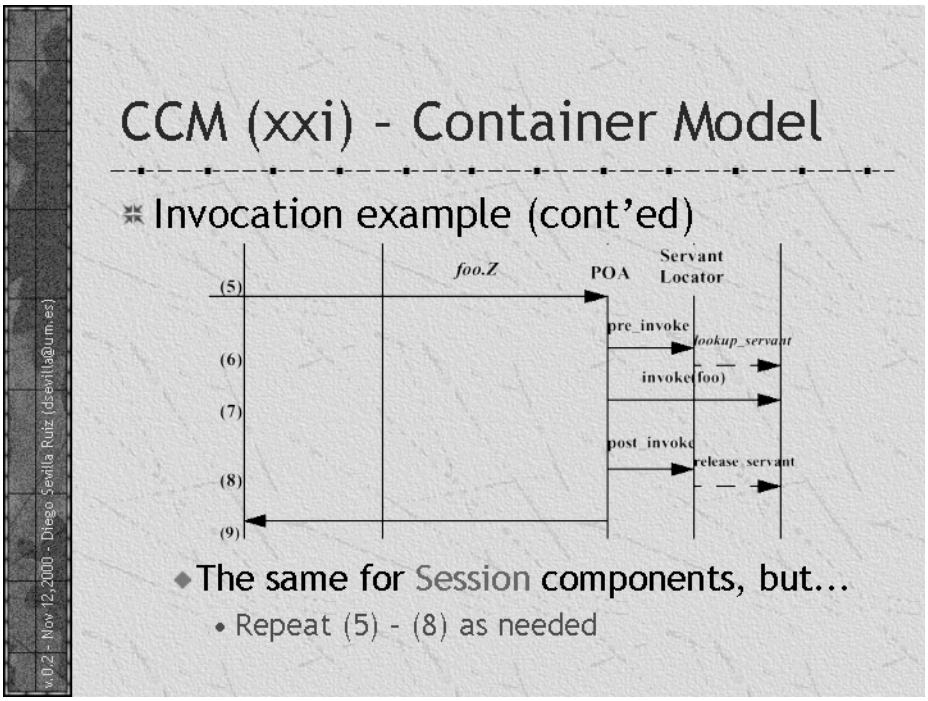
The tables above show the most important features of two of the four component categories. Table values show the differences between both categories: *Service* and *Entity*. They show what interfaces the container offers: `SessionContext` or `Session2Context` (respectively for basic and extended component levels) for *Service* or *Session* components and `EntityContext` (resp. `Entity2Context` for extended level) for *Entity* or *Process* component kinds (categories). These interfaces are used by the container to set an "execution context" to component instances (transactional context, security role, etc.) The component can ask the container for its context at any time.

Callback interfaces must be implemented by component instances and also depends on component category.

Note also that *Service* components have no primary key, so they offer a factory interface (and not a finder one) and that they only are allowed to use a "*method*" lifetime policy. Compare the values with the ones for *Entity*.

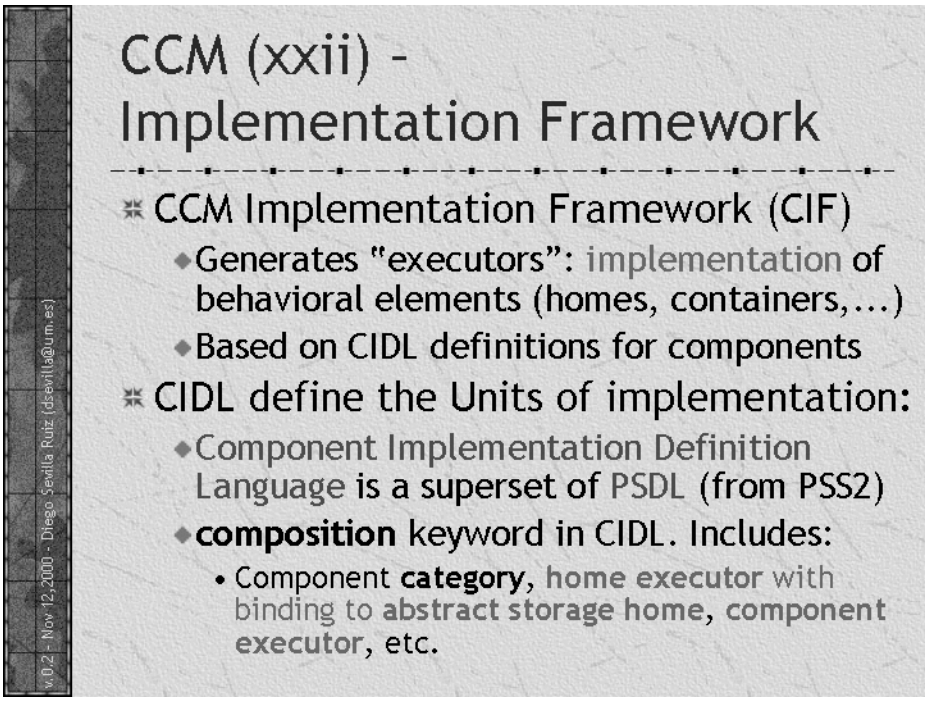


The slide shows a client performing an invocation on a *Service* component instance. After resolving the "`ComponentHomeFinder`" reference (new in CORBA 3), the client can search for the factory for this component (`ZHome`). Then, it is ready for creating a new instance of the component `Z` through the standard `create()` method of `ZHome`. (continues)



Once the client holds a reference to the component Z's instance, it can start invoking methods on it. Note how the invocation gets through the POA of the Z's container. The POA uses the *Servant Locator* to search for the corresponding *servant*, activate it, and then redirecting the invocation to the new activated servant. Once the invocation ends, the POA can deallocate this servant or maintaining it in a servant pool.

Session components can choose not to be deactivated on each method call. Instead, they can be active during the whole conversation with the client.

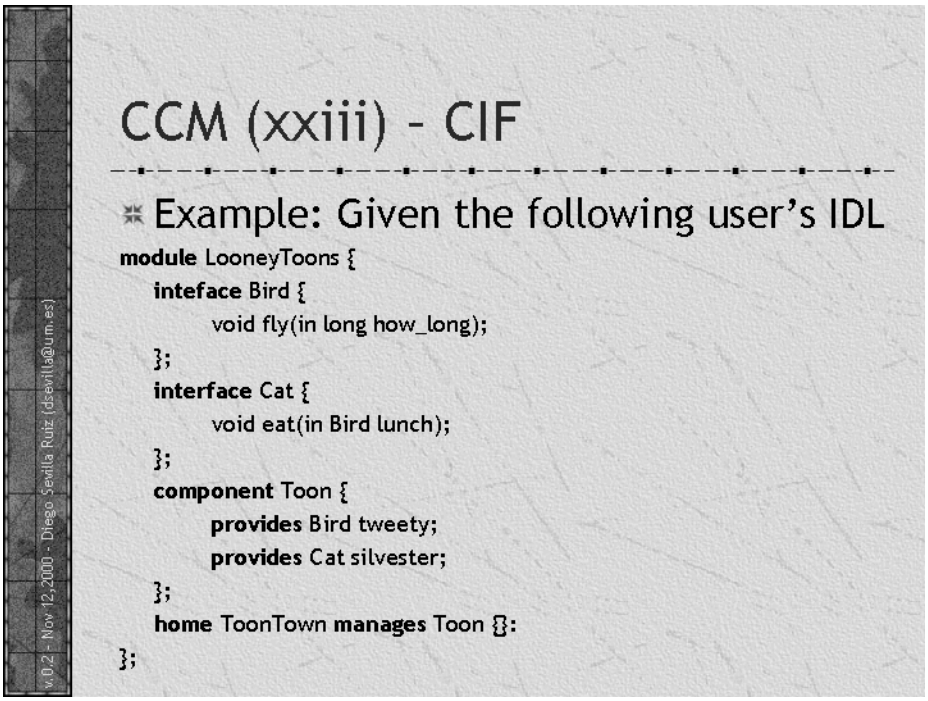


The CCM Implementation Framework (CIF) is a set of classes and tools that help in the implementation of components. The idea of a component being a binary unit is clear in the CCM, so the CIF defines the concept of "*executor*" to name implementation artifacts that show some kind of behavior (for example, home implementation, container, the component implementation itself, etc.)

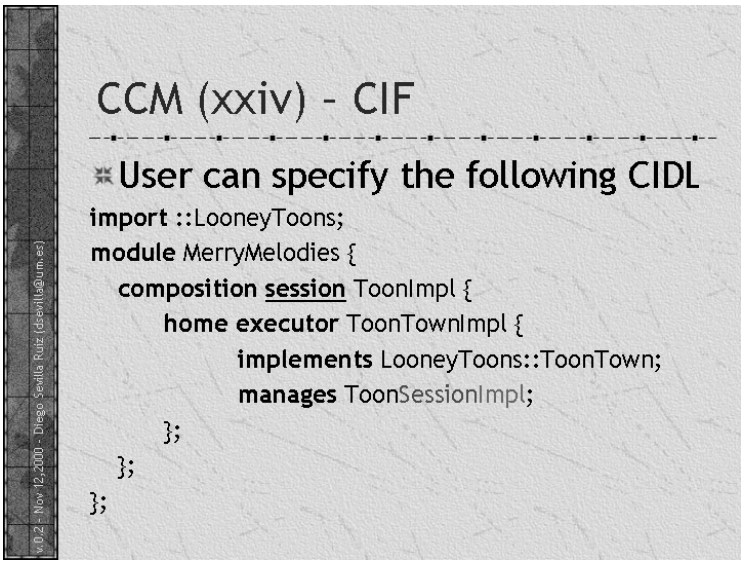
The CIF takes a component's CIDL definition written by the programmer. CIDL stands for *Component Implementation Definition Language* and is a superset of the Persistent State Service's PSDL (*Persistent State Definition Language*). It specifies how the implementation of a component is going to be split in different classes (these executors will be mapped to object files such as DLLs or `.class` files in the packaging descriptor) and how the state elements are associated to each one.

The most important keyword in CIDL is "`composition`", which specifies a component from the implementation point of view, showing its category, the names of the classes implementing the home and the component itself and the mapping between the implementation and the state elements (remember that CIDL is a superset of PSDL).

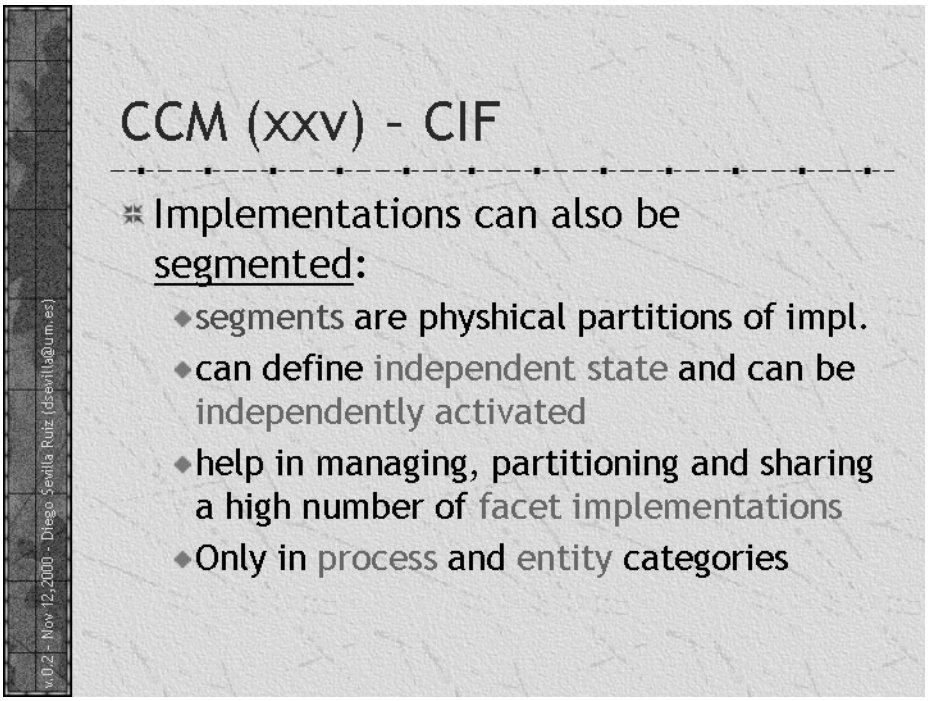
Following slides offer a simple example of CIDL. You can look at the references at the end for more examples. Note, however, that as there is no implementation of the CCM, the examples are somewhat incomplete.



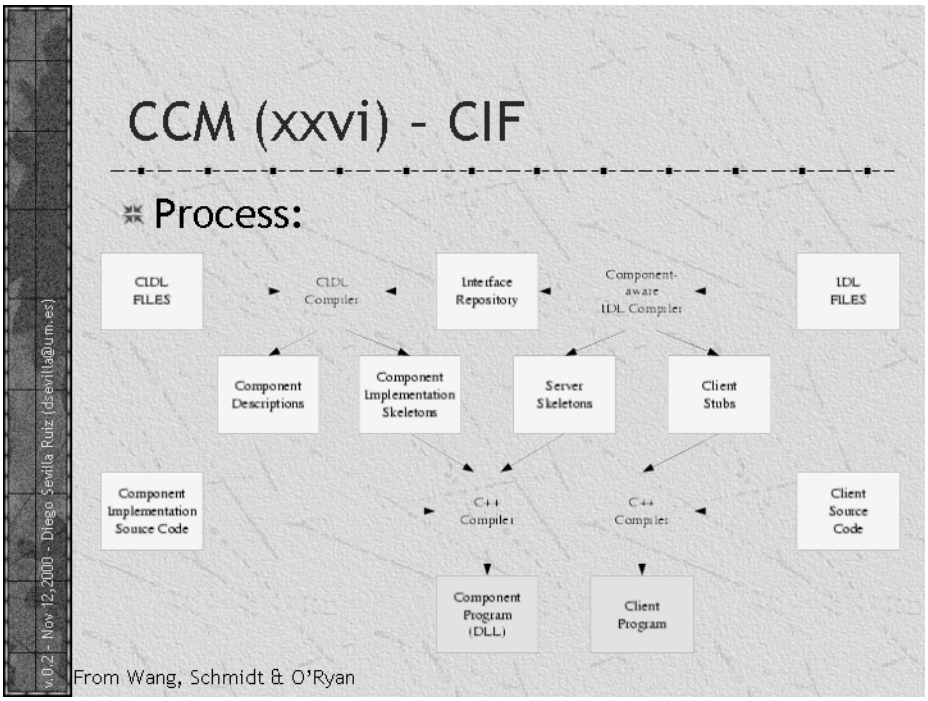
Example: given the previous extended IDL, the programmer could specify...



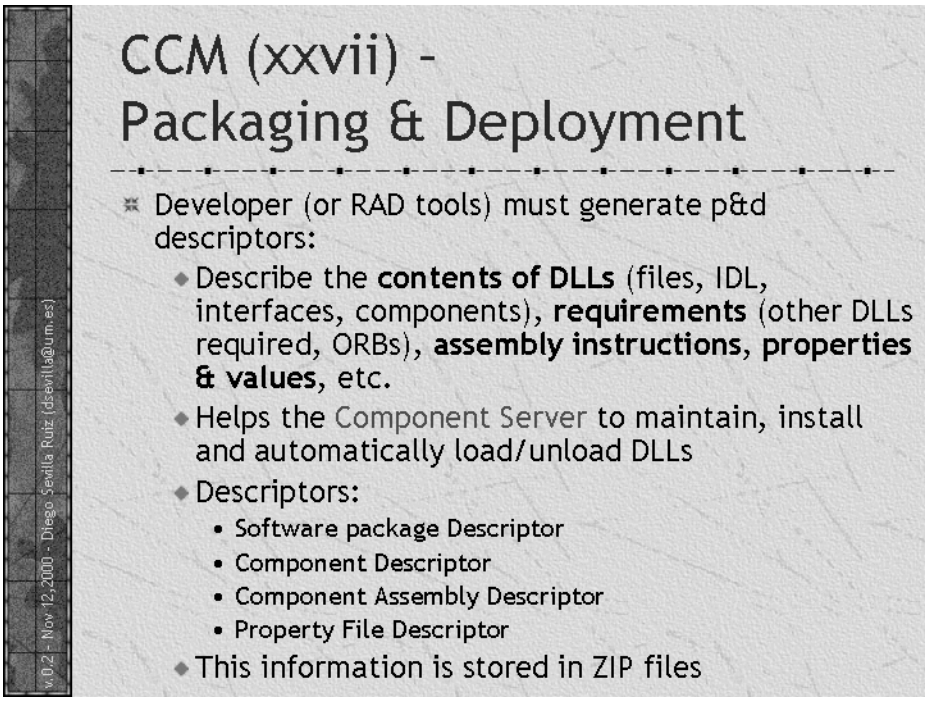
this CIDL, which includes the definition of the component category, the name of the *executor* that implements the *home* and, finally, the *executor* implementing the component itself: `ToonSessionImpl`.



Implementations can also be segmented and divided in independently loadable units. Thus, for example, different components can share facet implementations, or you can load the needed segments of a component instead of loading it completely. Each segment can state its persistence requirements (mappings to the defined state elements). Segments allow the splitting a component in code chunks that can be mapped to different DLLs (Dynamic Loading Libraries).



The slide shows the process imposed by the CIF. From the CIDL specification, a CIDL compiler generates the needed component descriptions and the *skeletons* for its implementation. Using the component code written by the programmer, the C++ compiler can generate the DLL corresponding to the binary representation of the component (you can also generate a `.zip` file containing the DLL an component descriptors using other tools provided by the framework). The process is almost identical for any other programming language.



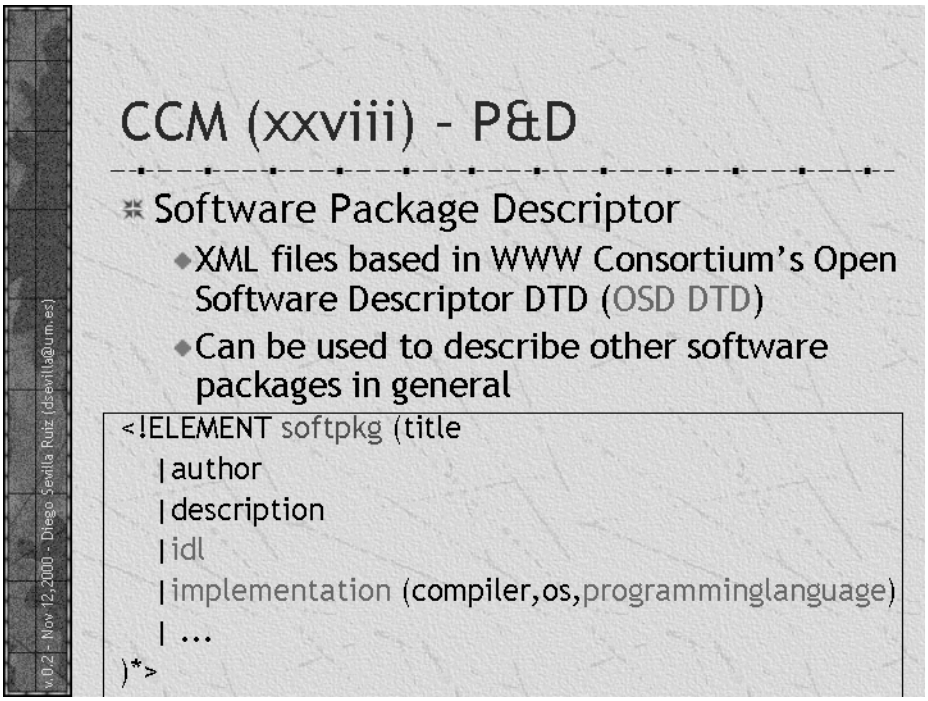
CCM also specifies a packaging and deployment model. The developer (or the development tool) must generate descriptors which specify component's features: DLL contents, IDL files, interfaces, other DLLs or components required, assembly instructions, properties and values, etc.

All this information helps the component server to manage, install, and dynamically load components as needed.

Following slides will show the different descriptors defined by the CCM:

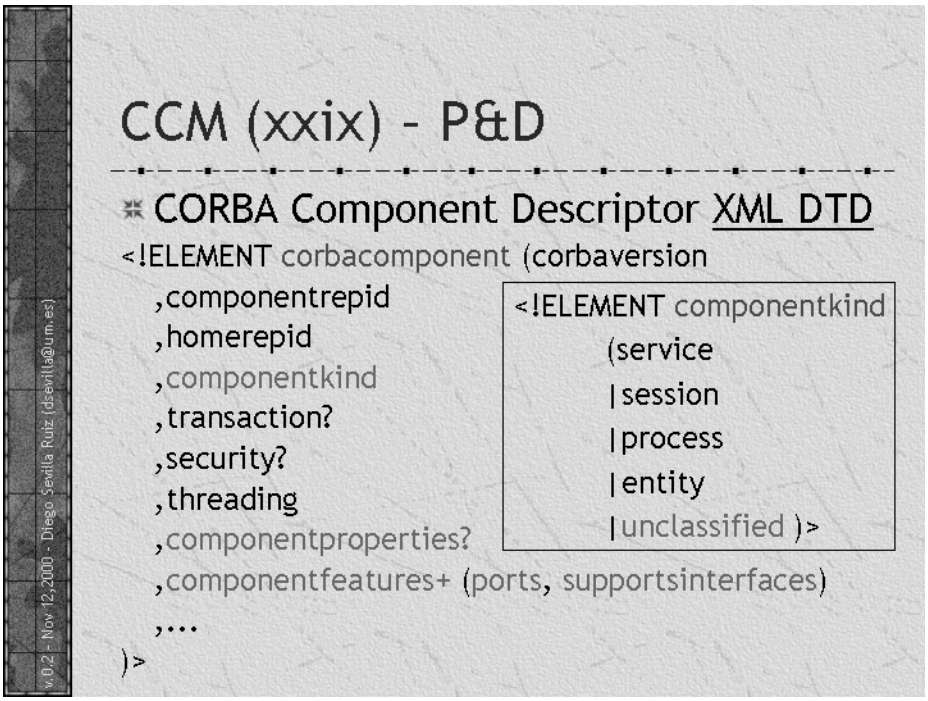
- *Software Package Descriptor*
- *Component Descriptor*
- *Component Assembly Descriptor*
- *Property File Descriptor*

Each descriptor is described using a XML file that can be included within the component's `.zip` file.

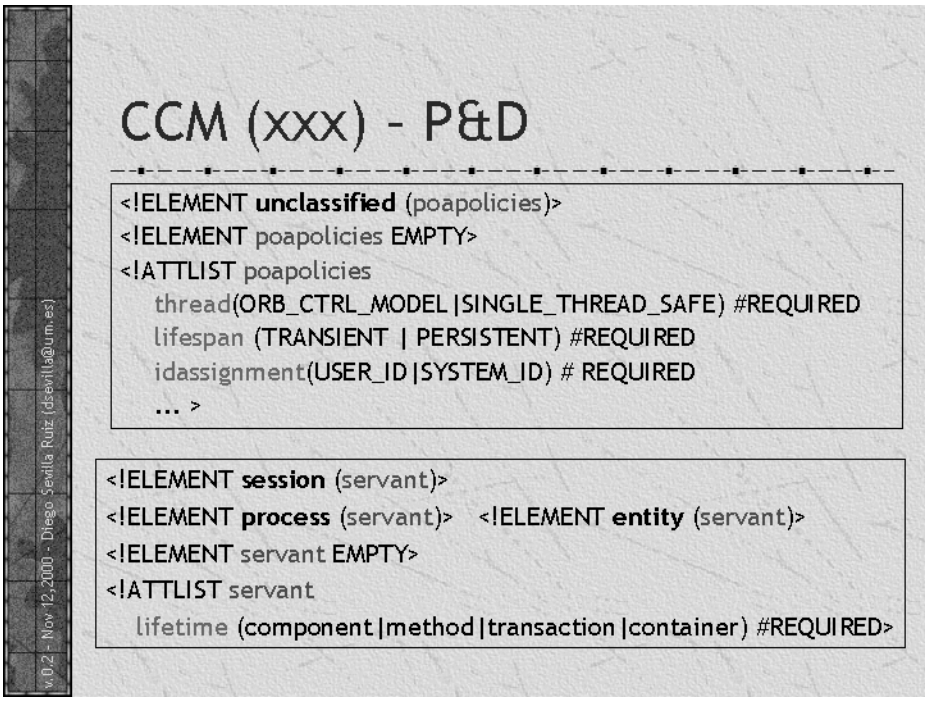


The Software Package Descriptor is a XML file describing the package in which a component is included. It is based on the WWW Consortium's *Open Software Descriptor Data Type Definition* (OSD DTD), http://www.w3.org/TR/NOTE-OSD.html.

As can be seen in the slide, it can be used to describe software packages in general.

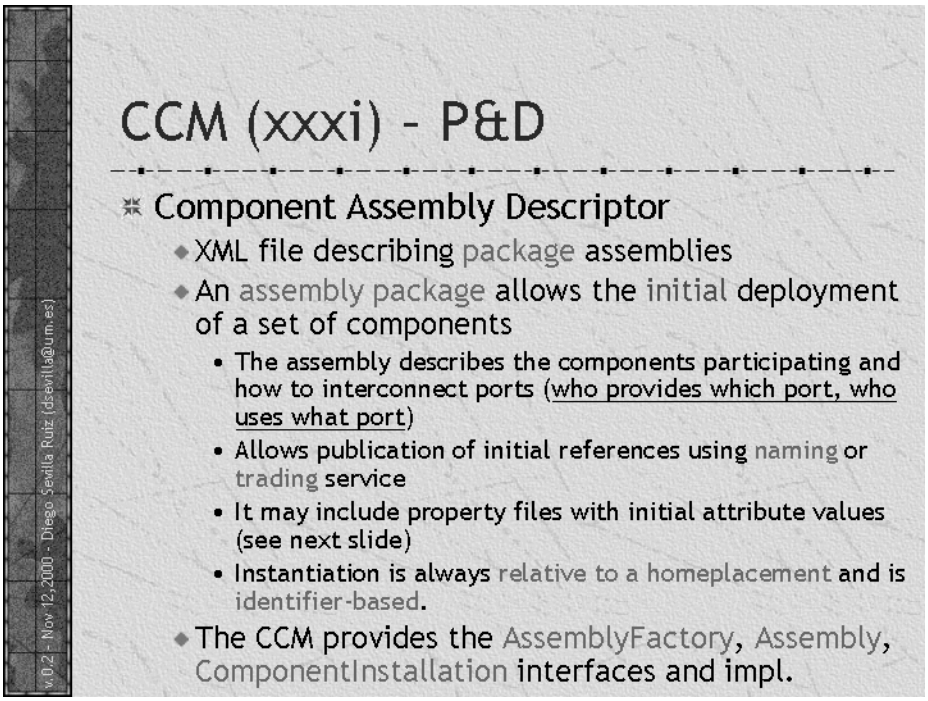


As you can see, it specifies all the features a component offers. In the slide I've highlighted `componentfeatures` and `componentkind`, that, as you can see in the detail, allows to specify the component category, or even specifying none at all, as you can see in the next slide.



The "`unclassified`" element has "`poapolicies`" as a child node, that is, it allows specifying POA policies that will be applied to the component container. This is a fine grained description of component's requirements if the component cannot be fit in any of the four component categories. It is useful in understanding what a "component category" means.

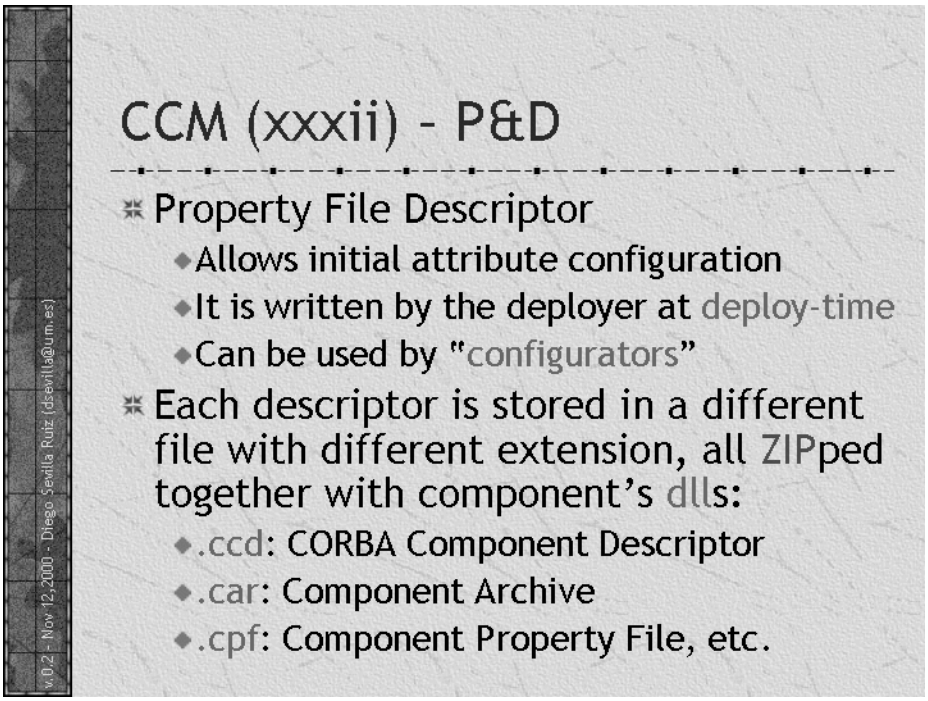
To specify servant lifetime, it's worth looking at the DTD: Session, Process and Entity components can specify the servant lifetime policy using the `lifetime` element.



The *Component Assembly Descriptor* specifies what components take part on an assembly. An assembly can be a complete application or a service requiring some components. It also specifies what ports of each component connect to other component's ports (each port is given a name which is relative to this assembly).

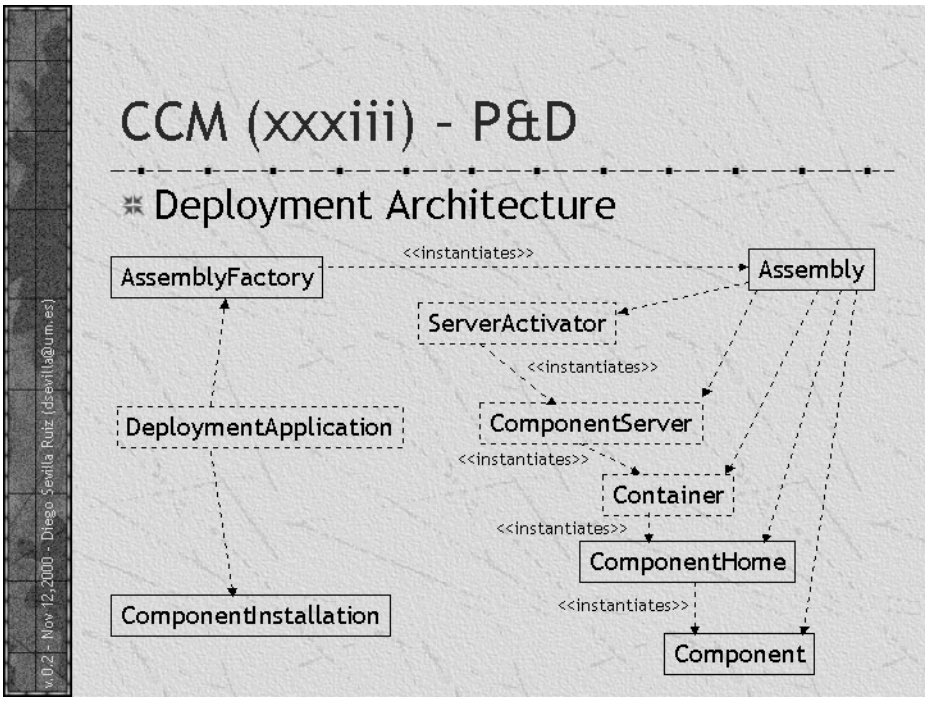
Moreover, it allows specifying how the initial references will be offered to clients: using the naming service or the trading service, how the references will be structured, etc.

The assembly can reference files with properties and values providing initial component configuration suitable for this assembly.



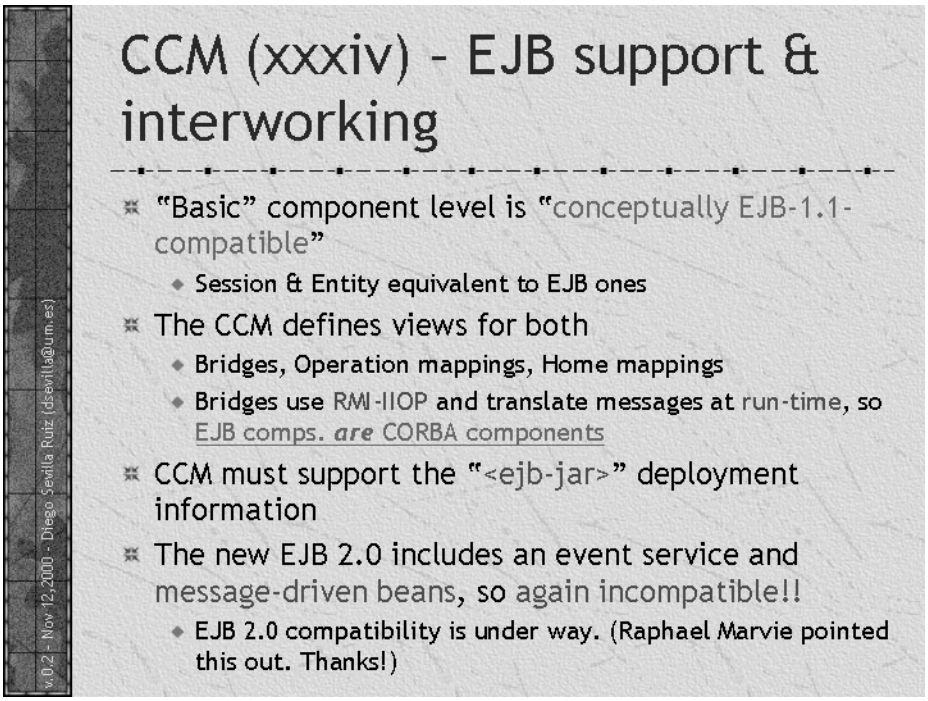
The *Property File Descriptor* specifies the set of component attributes and their initial values in the corresponding assembly. It can be generated by the application deployer at deployment time.

Each descriptor we've seen can be written in a different file that is included within the `.zip` file for that component.



Once an assembly has been constructed, the assembly process must be carried on. Thus, the components must be installed in each machine in which they're going to be run. In the case a component is not installed in a required machine, the CCM offers a series of interfaces (`ComponentInstallation`) for performing this installation: the component voyages to the machine and is installed there.

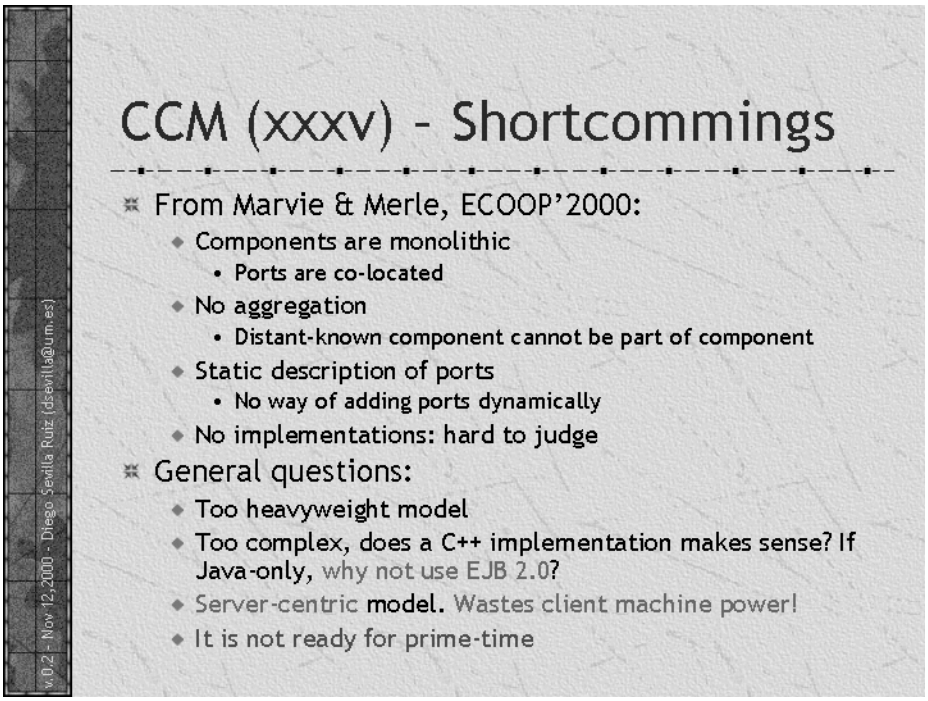
Each machine involved in an assembly (for instance, the set of a company's servers) must be running a daemon offering the `ServerActivator` interface. This interface is in charge of on-demand activation of the `ComponentServer`, which will create the containers hosting the component instances assigned to this machine. The `Assembly` establishes the initial values using the property files for each component taking part on the assembly, and it eventually calls the `configuration_complete` method of each component, causing the whole application to start running.



The CCM specification has been designed to be a superset of the EJB 1.1 spec. Thus, it contains all the concepts of the latter and can be said "conceptually equivalent". For instance, Session and Entity component categories are logically equivalent to the ones defined in EJB. Moreover, to allow a soft migration, the CCM defines two component levels: basic (compatible with EJB) and extended (with all the features studied).

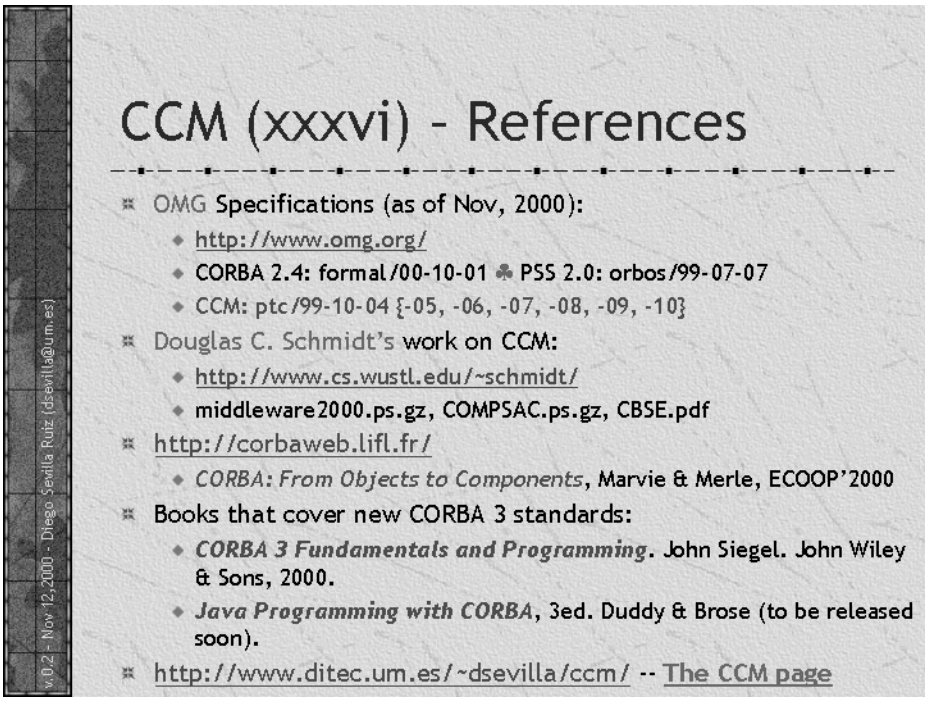
CCM offers "views" which allow EJB to use a CCM component and vice versa. As both APIs are different, the CCM defines an on-line translation of method calls. Thus, an EJB component **is** a CCM (basic) component and vice versa. CCM must also accept EJB's deployment descriptors "`<ejb-jar>`".

The new EJB 2.0 introduces a new, event-driven, bean type, making again both specifications incompatible. However, as Raphaël Marvie pointed out, the OMG is studying the compatibility with EJB 2.0.



Although the CORBA Components specification has not been implemented and is too early to judge (as Marvie & Merle point out), the specification has some shortcomings that could be addressed:

- Ports are co-located (port implementation is physically coupled with the component)
- Remote known components cannot be aggregated. Component facet implementations must be local.
- Port description is static and cannot be changed at run time (add ports, dynamic component construction, etc.)
- The model is a complex one, thus limiting the number (and perhaps the quality) of the implementations
- Does a C++ implementation make sense? Is it even possible? If not, why not using EJB 2.0 for Java instead?
- It is a server-centric model. It does not take in account client machine power and does not address user interface issues (another component model for that?).
- **IT IS NOT READY YET!!**



In this slide you can find some references related to CCM. My intention is to maintain "*The CCM Page*" highly updated. You can reach it at

http://www.ditec.um.es/~dsevilla/ccm/

In this page you can find the latest references I find about the CORBA Component Model.

I would like to thank Eduardo Martinez Graciá for revising the Spanish version of this slides, and Marcos Menárguez Tortosa for introducing me in EJB and CCM.

Diego Sevilla Ruiz
December, 2000