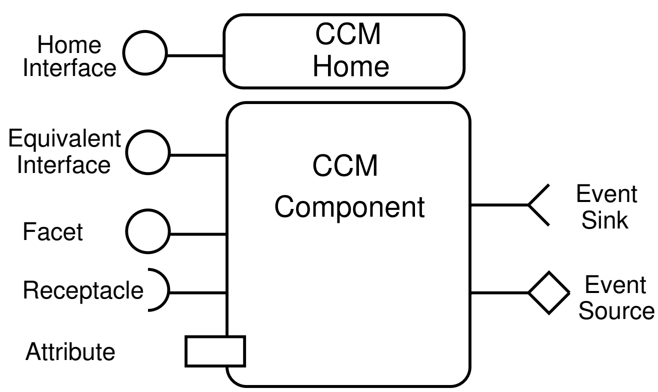
http://ccmtools.sourceforge.net

CCM Tools User's Manual



Egon Teiniker

August, 2006

Contents

Chapter 1

Introduction

1.1 Component–Based Software Engineering

Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. Components are for composition. Composition enables prefabricated components to be reused by rearranging them in new composites. Traditional software development can broadly be devided into two approaches:

- At one extreme, a project is developed entirely from scratch, with the help of only programming tools and libraries.

- At the other extreme, everything is build of standard software which is bought and parametrized to provide a solution that is close enough to what is needed.

The concept of component software represents a middle path that could solve this problem. Although each bought component is a standardized product, the process of component assembly allows the opportunity for significant customization. *Component–Based Software Engineering* (CBSE) [**?**, **?**, **?**] has become recognized as a new subdiscipline of software engineering. The major goals of CBSE are:

- To provide support for development of software systems as assemblies of components.

- To support development of software components as reusable entities.

- To facilitate the maintenance and upgrade of systems by customizing and replacing their components.

Software components were initially considered to be analogous to hardware components in general and to integrated circuits (IC) in particular. But software technology is an engineering discipline in its own right, with its own principles and laws. Therefore, such analogies break down quickly when going into technical details.

1.2 Software component definition

Software components are the main part in CBSE. Therefore, we need a precise definition of this term. Unfortunately, there are several different component definitions in literature. A major problem is the multiple overloading of the term *Component* in the software world.

1.2.1 Syntactic specification of software components

Clemens Szyperski [?] defines a component by enumerating the characteristic properties of a software component:

Definition (Szyperski) 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 party.

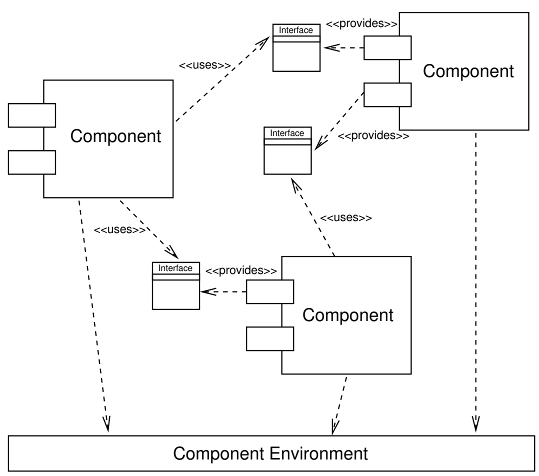


Figure 1.1: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

This component definition contains important terms which must be discussed in detail [?]:

- **A unit of composition**: The purpose of components is to be composed with other components. A component–based application is thus assembled from a set of collaborating components.

- **Contractually specified interfaces:** To be able to compose components into applications, each component must provide one or more interfaces. These interfaces form a contract between the component and its environment. The interface clearly defines which services the component provides - it defines its responsibilities.

- **Explicite context dependencies only:** Software usually depends on a specific context, such as the availability of database connections or other system resources. One particularly interesting context is the set of other components that must be available for a specific component to collaborate with. To support the composability of components, such dependencies must be explicitly specified.

- **Can be deployed independently:** A component is self–contained. Changes to the implementation of a component do not require changes to other components, as long as the interface remains compatible.

- **Third parties:** The engineers who assemble applications from components are not necessarily the same as those who created the components. Components are intended to be reused - the goal is a kind of component marketplace in which people buy components and use them to compose their own applications.

Szyperski's definition does not have satisfactory support for specification of nonfunctional properties. The following definition, introduced by Ivica Crnkovic [**?**], summarize the common aspects of component definitions, including nonfunctional features, found in literature:

Definition (Crnkovic) : To be able to describe a component completely and to ensure its correct integration, maintenance and updating, the component should consist of the following elements:

- A set of interfaces provided to, or required from, the environment. These interfaces are particularly for interaction with other components, rather than with a component infrastructure or traditional software entities.
- An executable code, which can be coupled to the code of other components via interfaces.

To improve the component quality, the following elements can be included in the specification of a component:

- The specification of nonfunctional characteristics, which are provided and required.
- The validation code, which confirms a proposed connection to another component.
- Additional information, which includes documents related to the fulfilling of specification requirements, design information, and use cases.

A difficulty in CBSE is deciding how to deal with nonfunctional aspects of communication, cooperation, and coordination included in a component architecture. These nonfunctional properties should be possible to compose and easy to control. A clear separation of nonfunctional requirements gives a component more context independence.

1.2.2 Semantic specification of software components

Most techniques for describing interfaces are only concerned with the signature part, in which the operations provided by a component's interface are described, and thus fail to address the overall behavior of the component. Ivica Crnkovic describes five levels of formalism for such semantic specification:

- **No semantics**: The focus is exclusively on the syntactic parts of the interfaces, represented by interface description or programming languages.

- **Intuitive semantics**: Here we use plain text, unstructured descriptions and comments about a component and its parts.

- **Structured semantics**: The semantics are presented in a structured way but need not be in accordance with any particular syntax or formalism.

- **Executable semantics**: The semantic aspects are expressed in a way that can be executed and verified by the system during run time (assertions can be used to express preconditions and postconditions and to test them during run time). Note that client code may also take advantage of executable assertions by checking the pre- and postconditions of an operation call.

- **Formal semantics**: Programs can be proved to have consistent and sound semantics. Formal specification languages such as VDM and Z are examples of approaches on this level [**?**].

Specifications that include syntactic and semantic information are often called **Contracts**. As mentioned by Meyer [**?**], a contract lists the global constraints that the component will maintain (the invariant). For each operation within the component, a contract also lists the constraints that need to be met by the client (the precondition) and those the component promises to establish in return (the postcondition).

1.2.3 Objects versus components

The term *Object* and *Component* are often thought to be very similar, but there are significant differences:

- **Granularity.** In contrast to a programming language object, a component has a much larger granularity and therefore usually more responsibilities. Components were introduced to group objects to larger entities to reduce the overall complexity of a software system.

- **Multiple interfaces per component.** An object typically implements a single class interface, which may be related to other classes by inheritance. In contrast, a component can implement many interfaces, which need not be related by inheritance. Components can provide navigation operations to move between different component interfaces. Navigation in objects is limited to moving up or down an inheritance tree via cast or narrow operations.

- **Extensibility.** While objects are implemented in a particular programming language, components are not restricted in that way. Components can be viewed as providers of functionality that can be replaced with equivalent components written in any programming language. This extensibility is facilitated via the **Extension Interface** design pattern [**?**], which defines a standard protocol for creating, composing, and evolving groups of interacting components.

- **Improved communication.** Components have a more extensive set of intercommunication mechanisms (synchronous/asynchronous, local/remote, messages/methods) than objects.

- **Higher–level execution environment.** Component models define a runtime execution environment, called component container, that operates at a higher level of abstraction than access via ordinary objects. The container provides additional levels of control for defining and enforcing policies on components at runtime.

1.2.4 A taxonomy of components

Because of its generic definition, the term component is used to describe rather different software concepts. The component taxonomy shown in Fig. 1.2 should help to structure the diffent concepts in context of software components.

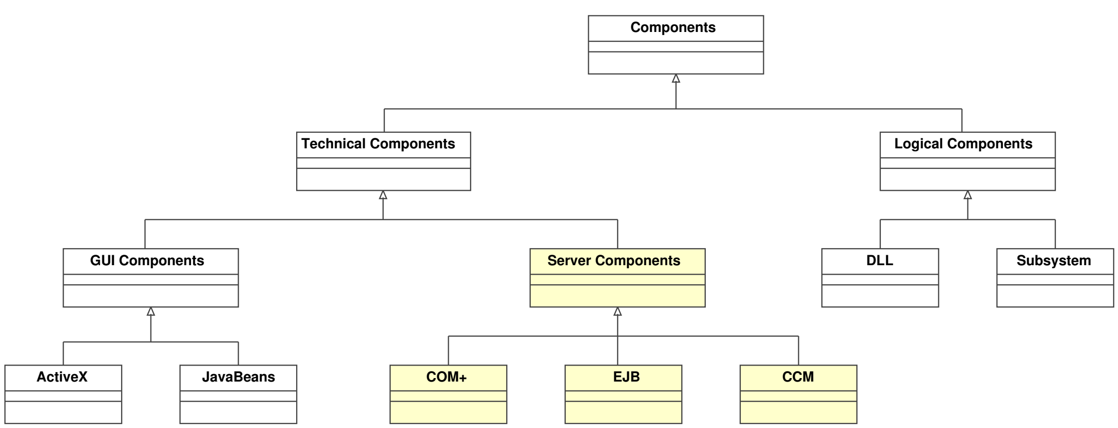


Figure 1.2: A taxonomy of components.

- **Logical Components:** A logical component is simply a package of related functionality. In can be some kind of **Subsystem**, a **DLL** or a complete, standalone application that runs as part of a larger system. Logical components are mainly a way to keep the complexity of a system under control, and to organize version control or project management issues. There is also the notation of a **Business Component** [**?**] - an aggregation of data, domain and user components that embody a complete subsystem.

- **Technical Components:** These are technical building blocks to asembly applications. A technical component can not run without a runtime environment called container. A container handles the technical concerns like transactions, security, failover or load–balancing for the components. Technical components are either used in client applications or on the server.

  – **Client components:** the container for client components is typically an IDE where the components are configured at development time. The most popular examples are **ActiveX Controls** and **JavaBeans**

  – **Server components:** usually encapsulate business logic in multi–tier systems and the container is typically a part of an application server. There are three mainstream component technologies: **COM+**, **Enterprise JavaBeans** (EJB) and **CORBA Component Model** (CCM). These server components are never used as client components, because the containers are rather complex and not available at the client side.

Chapter 2

Login Example

2.1 Introduction

As a quick tour through component–based software development using CCM Tools, we implement a simple component that provides single interface to its clients.

2.2 Component Definition

We define components using the CCM Tools *Interface Definition Language* (IDL), which is actually a subset of the CORBA IDL3 specification. As shown in the following listing, a component definition can imply the definition of a component home, one or more interfaces, operation parameters and exceptions.

```
module application
{
    enum Group { GUEST, USER, ADMIN };

    struct PersonData
    {
        long id;
        string name;
        string password;
        Group group;
    };

    exception InvalidPersonData
    {
        string message;
    };

    interface Login
    {
        boolean isValidUser(in PersonData person)
          raises(InvalidPersonData);
    };

    component Server
    {
        provides Login login;
    };

    home ServerHome manages Server { };
};
```

Here we can give only a short description of these IDL artefacts, you can find more information in chapter 4 “Interface Definition Language“ of this manual.

- **Modules** (e.g. `application`).
  Modules combine related IDL definitions into a logical group and prevent pollution of the global namespace.

- **User Defined Types** (e.g. `Group`, `PersonData`).
  In addition to build–in types like `long`, `boolean`, `string`, etc. a component designer can define its own types using, for example, `enum` and `struct` declarations. Such user defined types can act as operation parameters as well as attribute types.

- **Exceptions** (e.g. `InvalidPersonData`).
  To report an error condition, an operation can throw one or more exceptions. Befor we can declare an exception as part of an operation's raises section, we have to define the exception which is pretty similar to defining a structure type.

- **Interfaces** (e.g. `Login`).
  An interface defines a named set of operations and attributes. Each operation definition contains a result type, operation name, parameter list (which can be empty) and an optional exception list. In IDL, each operation parameter includes a passing direction:

  - `in`: the parameter is passed from the caller to the callee.
  - `out`: the parameter is passed from the callee back to the caller.
  - `inout`: the parameter is passed from the caller to the callee, modified and sent back to the caller.

- **Components** (e.g. `Server`):
  A component uses interfaces to define input and output ports called facets and receptacles. While a facet's interface is implemented within its own component, a receptacle's interface uses implementations of connected facets of other components.

- **Component Homes** (e.g. `ServerHome`):
  To have an entry point for component instantiation, we define a component home. In the case of an empty home definition, a standard `create()` operation will be generated from the CCM Tools.

From these few lines of IDL, we can generate a lot of structural code which implements all features of the CCM Tools component model, thus, you can focus on your business logic.

2.3 IDL Repository Directory

First of all, we store the IDL source code in a file called `Login.idl` and tell the CCM Tools to generate an **IDL Repository Directory**. This `idl3repo` directory contains all defined IDL artefacts in separated files and in a uniform structure.

```
> ccmidl -idl3 -o ./idl3repo Login.idl
```

After this step, we should have the following directory structure:

```
|-- Login.idl
|-- idl3repo
|   |-- component
|   |   `-- application
|   |       |-- Server.idl
|   |       `-- ServerHome.idl
|   `-- interface
|       `-- application
|           |-- Group.idl
|           |-- InvalidPersonData.idl
|           |-- Login.idl
|           `-- PersonData.idl
```

In the repository, which is the starting point for all other CCM Tools activities, there are two subdirectories:

- The `interface` directory contains all IDL interface, parameter, exception, etc. definitions.
- The `component` directory contains all component and home definitions.

Each IDL artefact is stored in its own file within a directory that conforms to the defined IDL module hierarchy. For example, the interface `Login` has been defined in the module `application`, thus, this interface is stored in the directory `interface/application` in a file named `Login.idl` within the IDL repository.

From the developers point of view, it does not matter if the component definitions are stored in a single or in multiple source files, the generated `idl3repo` directory tree is the same in both cases.

2.4 Use Case 1: Local C++ Components

To introduce the first CCM Tools use case, we implement a local C++ component and a collocated unit test. This use case is adequate for a developers who implements large but modulare C++ applications.

The implementation of local C++ components requires the following activities:

- Model a component's structure in IDL (see section 2.2).
- Generate component logic (= component skeletons).
- Implement a component's business logic.
- Implement a component's client.

It is an inportant point that the modeling of IDL interfaces and components is completely independent of component implementations. As you will see, we use the IDL artefacts stored in the IDL repository directory to generate both C++ and Java implementations.

2.4.1 Generate component logic

Let's start development. From the IDL repository directory the CCM Tools generate a component skeleton which establishes the component's structure, provides C++ interfaces to clients or other components and uses the C++ runtime environment.

```
> mkdir c++
> mkdir c++/server
> cd c++/server

> ccmtools c++local -I../../idl3repo/interface -I../../idl3repo/component \
                    -o ./src/interface                                     \
                    ../../idl3repo/interface/application/*.idl

> ccmtools c++local -I../../idl3repo/interface -I../../idl3repo/component \
                    -a -o ./src/component/Server                           \
                    ../../idl3repo/component/application/Server*.idl
```

After this code generation step, the following directory structure has been created:

```
Login/c++/server
  `-- src
    |-- component
    |   `-- Server
    |       |-- CCM_application_ccm_local_component_Server
```

```
|       |-- CCM_application_ccm_local_component_Server_share
|       `-- application_ccm_local_component_Server_ServerHome_entry.h
`-- interface
    |-- CCM_application_ccm_local
    `-- CCM_application_ccm_local_adapter
```

Basically, all directories starting with 'CCM_' contain component logic which is completely generated (so there is no need to check–in these directories into a CVS like system).

The component logic fills the gap between a component's interfaces and its business logic implementation.

Note that generated component logic can change between different CCM Tools versions to improve a component's non functional behavior. Such changes do neither affect component interfaces nor your business logic implementation which realizes the functional behavior of components.

2.4.2 Implement business logic

Component business logic must be embedded in the generated component logic. To make life easier, we used the `-a` option during code generation. This flag forces the code generator to generate application skeletons.

You can find these application skeletons in the `src/component/Server` subdirectory:

```
Login/c++/server
  `-- src
    |-- component
    |   `-- Server
    |       |-- ServerHome_impl.cc
    |       |-- ServerHome_impl.h
    |       |-- Server_impl.cc
    |       |-- Server_impl.h
    |       |-- Server_login_impl.cc
    |       |-- Server_login_impl.h
```

As a developer, you are responsible for these files because they represent the component's business logic (you should check–in these files into a CVS like system).

There is a direct relationship between IDL and these business logic files:

- `ServerHome_impl.*`
  For each component home, an implementation class is generated which provides an implementation of the default `create()` operation. Additionally, the `ServerHome_impl.cc` file contains the implementation of the global:

```
create_application_ccm_local_component_Server_ServerHome()
```

function which represents the business logic entry point used by the generated component logic.

- `Server_impl.*`
  For each component, an implementation class is generated which provides default implementations of the component's callback operations.

- `Server_login_impl.*`
  For each facet, an implementation class is generated which provides operation skeletons for hosting business logic implementation.

It is a good idea to generate these applikation skeletons only once - when starting component implementation. Small changes in IDL definitions can be appended pretty easy to these implementation classes manually.

Note that these implementation files are not overwritten by the CCM Tools. The generator replaces only untouched source files, otherwise the new generated files are stored with a `.new` suffix.

To implement the Login example's business logic, you open the `Server_login_impl.cc` file and implement the following code snippet:

```
bool
login_impl::isValidUser(
    const application::ccm::local::PersonData& person)
    throw(::ccm::local::Components::CCMException,
          application::ccm::local::InvalidPersonData )
{
    if(person.name.length() == 0)
        throw application::ccm::local::InvalidPersonData();

    if(person.id == 277
       && person.name == "eteinik"
       && person.group == USER)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Now, we can use Confix to build our component example. To tell Confix which directory should be built, a `Makefile.py` file must be created in each source code directory. You can delegate this work to CCM Tools:

```
> ccmconfix -makefiles -o ./src -pname "login" -pversion "1.0.0"
```

Finally, you start Confix to build all generated and manually implemented source files:

```
> confix.py --packageroot=`pwd`/src --bootstrap --configure --make
```

Hey, we a now ready to test this local C++ component implementation.

2.4.3 Implement local component client

Instead of a real client with a complex GUI, we simply implement a unit test for the component we have built in the last section.

We create a `src/component/Server/test` directory and store the following code in a file called `_check_local_component_Server.cc`:

```
#include <cassert>
#include <iostream>

#include <WX/Utils/debug.h>
#include <WX/Utils/smartptr.h>

#include <ccm/local/Components/CCM.h>
#include <ccm/local/HomeFinder.h>

#include <application/ccm/local/component/Server/Server_gen.h>
#include <application/ccm/local/component/Server/ServerHome_gen.h>

using namespace std;
using namespace WX::Utils;
using namespace ccm::local;
using namespace application::ccm::local;

int main(int argc, char *argv[])
{
  SmartPtr<component::Server::Server> server;
  SmartPtr<Login> login;
  Components::HomeFinder* homeFinder = HomeFinder::Instance();
  if(deploy_application_ccm_local_component_Server_ServerHome("ServerHome"))
  {
      cerr << "ERROR: Can't deploy component homes!" << endl;
      return -1;
  }

  try
  {
      SmartPtr<component::Server::ServerHome> serverHome(
        dynamic_cast<component::Server::ServerHome*>
          (homeFinder->find_home_by_name("ServerHome").ptr()));
```

```
        server = serverHome->create();
        login = server->provide_login();
        server->configuration_complete();

        // Implement your test cases here !!!

        server->remove();
    }
    catch(Components::Exception& e )
    {
        cout << "ERROR: " << e.what() << endl;
        return -2;
    }

    if(undeploy_application_ccm_local_component_Server_ServerHome("ServerHome"))
    {
      cerr << "ERROR: Can't undeploy component home!" << endl;
      return -3;
    }
}
```

This code snipped is verry similar in all unit tests of such simple components. It deploys the component home object, creates a component instance, uses the component's euqivalent interface to get a facet, and completes the configuration phase. Following the setup process, we can execute our component test cases (we will discuss the implementation of these test case later). Finally, we remove the component instance and undeploy the component home object. Each functional test case can be inserted into this unit test template shown above.

Our first test shows the usage of the `Server` component and its `login` facet. We fill the `PersonData` structure with valid data and call the `isValidUser()` operation. Depending on the component's result we print out a message to the console.

```
try
{
    PersonData person;
    person.id = 277;
    person.name = "eteinik";
    person.password = "eteinik";
    person.group = USER;

    bool result = login->isValidUser(person);
    if(result)
    {
        cout << "Welcome " << person.name << endl;
    }
    else
    {
        cout << "Sorry, we don't know you !!!" << endl;
    }
}
```

```
catch(InvalidPersonData& e)
{
    cout << "Error: InvalidPersonData!!" << endl;
}
```

The second test shows the component's behavior for an invalid `PersonData` structure. The test expects an `InvalidPersonData` exception to succeed.

```
try
{
    PersonData person;
    person.id = 0;
    person.name = "";
    person.password = "";
    person.group = USER;

    login->isValidUser(person);
    assert(false);
}
catch(InvalidPersonData& e)
{
    cout << "OK, caught InvalidPersonData exception!" << endl;
}
```

It is up to you to decide if you put both test cases into the same `_check_*` file or to implement each test case in its own file. Note that each `_check_*` file will end in a separate executable, thus, for huge applications you will need a lot of disk space.

To run these unit tests, we use Confix again:

```
> touch src/component/Server/test/Makefile.py
> confix.py --packageroot=`pwd`/src --bootstrap --configure \
            --make --targets=check
```

At the end of this build process, you hopefully see an output like:

```
Welcome eteinik
OK, caught InvalidPersonData exception!
PASS: login__check_local_component_Server
==================
All 1 tests passed
==================
```

Of course, to implement a component for such a simple functionality is somewhat academical, but this example shows how simple a component development cycle can be by using CCM Tools.

2.5 Use Case 2: Remote C++ Components

In the second CCM Tools use case, we implement a remote C++ component and a remote unit test client based on CORBA middleware. This use case is adequate for a developers who implements large and distributed C++ applications.

2.5.1 Generate remote component logic

2.5.2 Implement remote component client

2.6 Use Case 3: Local Java Components

2.7 Use Case 4: Remote Java Components

2.8 Summary

Chapter 3

Component Model

3.1 Introduction

...

3.2 CCM Tools Component Model

. . .

3.3 Local Component Structure

The implementation of server–side components is made up of different parts, as shown in Fig. 3.1, that are either manually written, generated by tools or existing libraries.

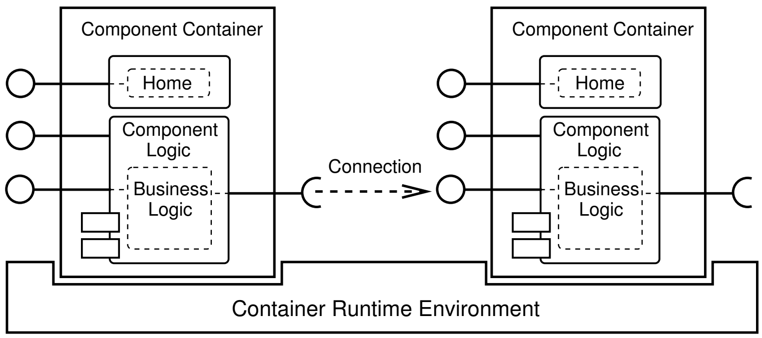


Figure 3.1: A component implementation covers business logic, component logic, component container and container runtime environment.

Business logic: a component's business logic is written by the component developer to implement a domain specific functionality. Basically, business logic should not deal with technical aspects like contract verification or middleware API.

Component logic: business logic is embedded in a layer of generated code called component logic. The interaction between business logic and component logic is well defined in terms of interfaces (context interface, callback interface). Component logic handles technical aspects as well as a component's life-cycle. Additionally, component logic is glue code that fits a component into a generic component container.

Component container: a component type is hosted by a component container that manages instances of that component type. While component logic is generated for each particular component type, the component container is a generic part of the component platform.

Container runtime environment: a component platform also supports a set of libraries and services that can be used by component containers. Any middleware used here is also part of this runtime environment.

From this implementation schema, two different component views can be deduced:

External view: a component provides its ports defined by interfaces. All client calls to these ports are routed through the component container and the generated component logic before business logic functionality is executed.

Internal view: a component's business logic can call methods on a context object, which is part of generated component logic, and has to implement a callback interface that allows the component container to handle the component's life–cycle.

Conforming with the concept of component model and middleware separation, we have implemented a local version of LwCCM components. These local components host business logic and are completely independent from a particular middleware technology.
Implementations without middleware are much leaner and allow finer grained components which are simpler to reuse. Especially in languages that do not provide a native component model (like C++), a local version of LwCCM can be useful.

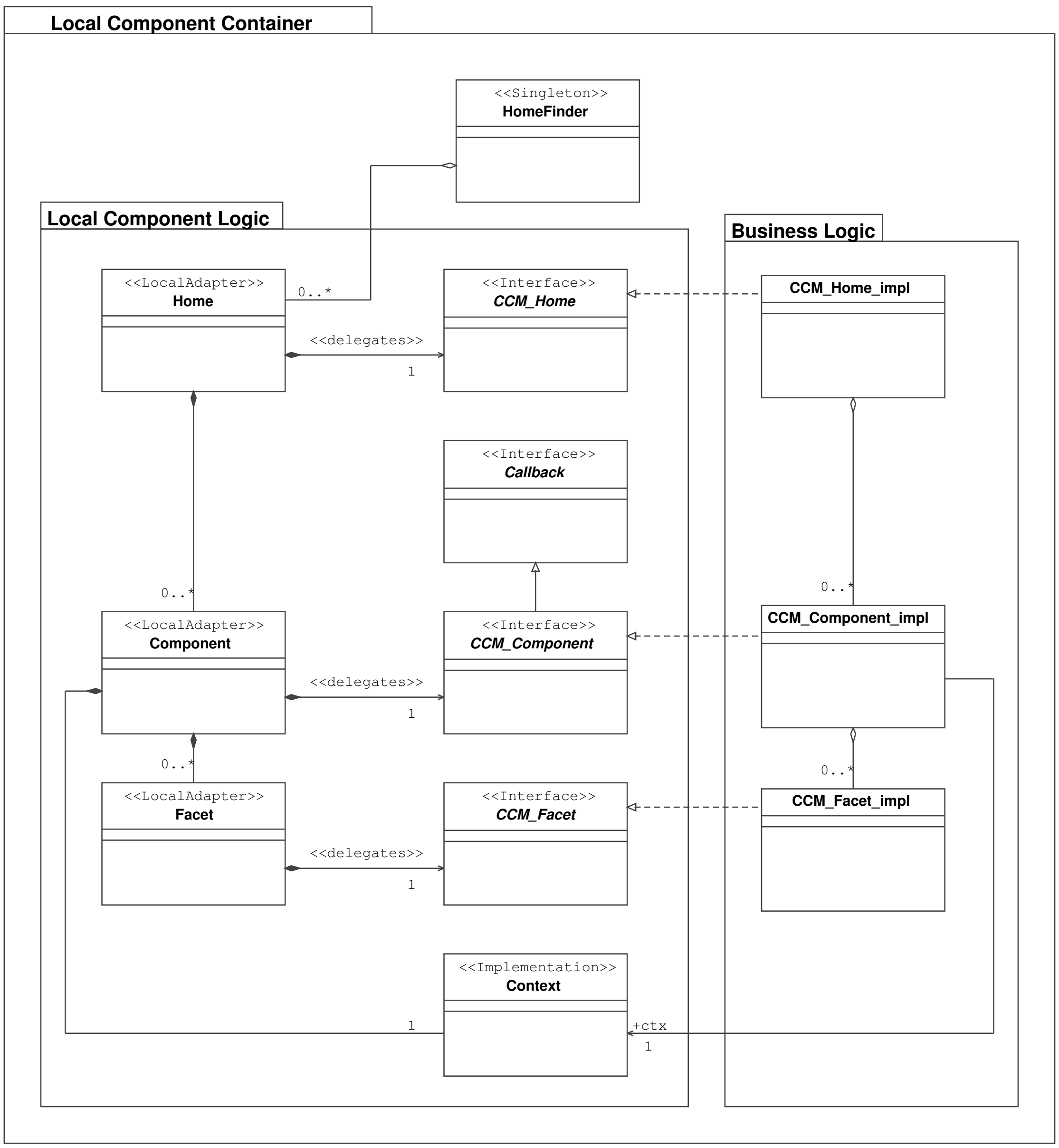


Figure 3.2: This simplified structure of a local component implementation shows the relationship between business logic, local component logic and a local component container.

Based on this simplified class diagram, we can analyze the following interactions between business logic and component logic:

Calling component methods: usually, a component client calls methods on a component's interface. These interfaces can be either a component home, a component equivalent interface or a facet. Invocations on all these interfaces follow the same structural pattern, as shown in Fig. 3.3. A component's client calls methods on generated adapter classes that implements a component's interfaces. These adapters delegate calls to generated interfaces which are implemented by business logic classes.

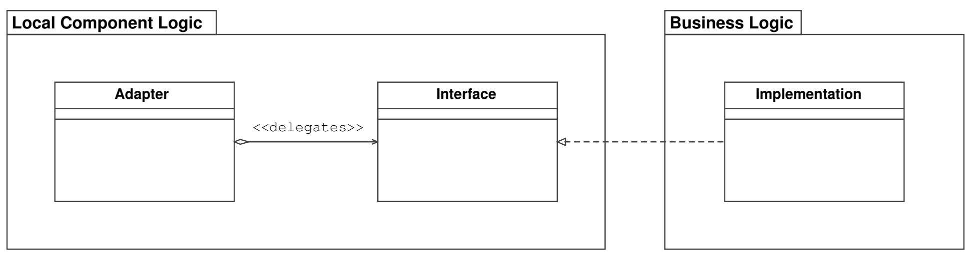


Figure 3.3: Structure of a local component.

This pattern implies two benefits:

- The adapter layer brings in an indirection that can be used to execute some pre- and post-invocation functionality.
- Business logic can implement a well defined interface which is generated from the component model.

Invoking callback methods: as shown in Fig. 3.2, a generated component interface inherits the `Callback` interface that defines methods which will be used by the component logic to control the business logic life-cycle.

Using context methods: while calls from component clients as well as calls from component logic to the callback interface are directed from outside to the business logic, there is also a need for interaction between business logic and generated glue code.

In the other direction, from business logic to component logic, a `Context` object is used to provide access to container functionality. Additionally, business logic gets receptacle references from this context object, that are used to call methods defined by these receptacle interfaces. Remember, receptacles are connected to facets which implement these common interfaces.

An important part of a local component container is the `HomeFinder` class. This class is implemented as a singleton [?] and manages component home instances. When a local LwCCM component is deployed, an instance of the component's home is created and registered by the `HomeFinder` using a unique name. After that, the `HomeFinder` can be used to retrieve home references by their name.

3.4 Nested Component Composition

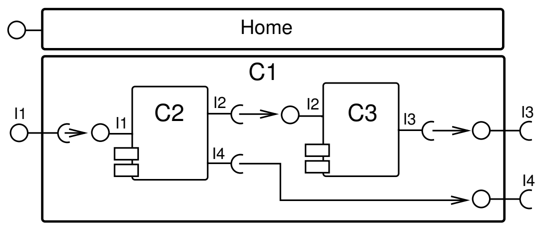


Figure 3.4: C_{Ref} super component example.

Using the session facade pattern [?], we are able reduce a nested component composition into a flat component assembly that can be described by a simple CAD file. While the inner components $C2$, $C3$ keep unchanged, the outer component $C1$ must be transformed into a special LwCCM component - the facade component, as shown in Fig. 3.5.

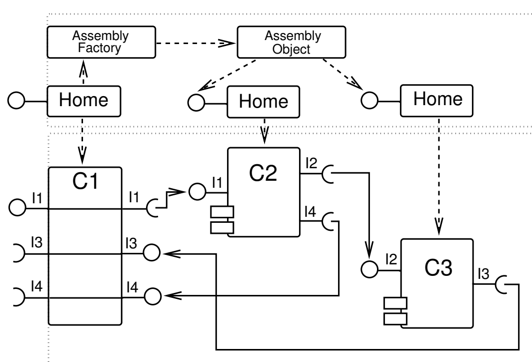


Figure 3.5: Using the session facade pattern, a nested component composition can be reduced to a flat assembly as defined in LwCCM.

The LwCCM facade component provides two complementary kinds of ports for each port defined in $C1$:

- **Public ports.** A public port is visible to component clients and can be accessed as a regular LwCCM component port.

- **Private ports.** A private port is a LwCCM port that is not visible to component clients. Private ports are used to connect the facade component with their inner component instances. Technically, private ports are implemented like public ports, but after the configuration phase, private ports can not be accessed by component clients.

All information about components and their connections within a super component are represented by an *Assembly Object*. Such an assembly object is assigned to a facade component instance, and can be seen as a part of the facade component itself. For each facade component instance, an instance of the corresponding assembly object must be created. To give a facade component's home the ability to create assembly object instances, an *Assembly Object Factory* must be assigned to a facade component home during component deployment. With this approach, we can use regular LwCCM components and assemblies to realize the nested component concept.
The implementation of a facade component's business logic is straightforward, each call to a facet method delegates to the corresponding receptacle and vice versa.

To give a client the illusion of a single component, a facade component has to handle some tasks behind the scenes. These are defined by the following sequence diagrams.

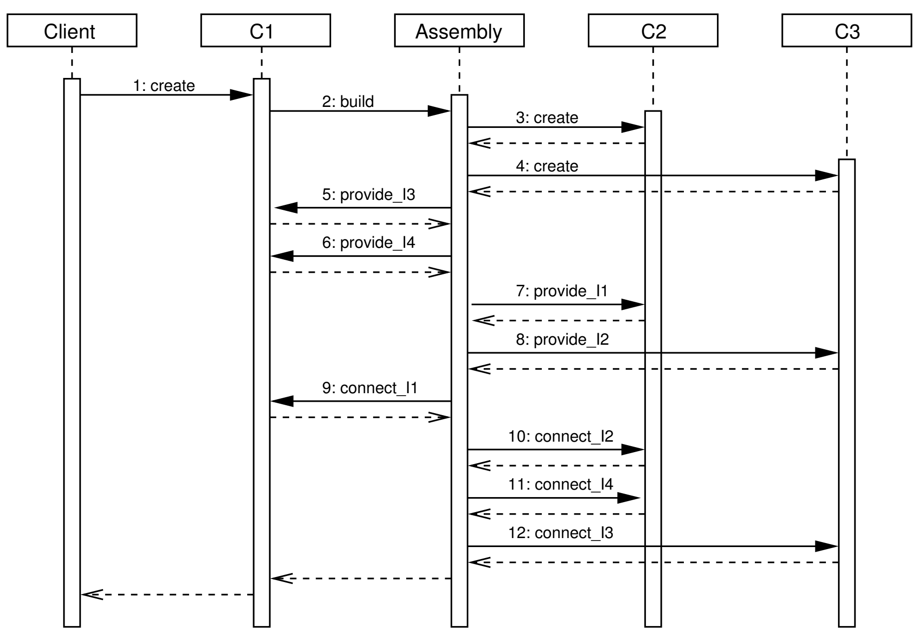


Figure 3.6: Create a super component of Fig 3.5.

From a client's point of view, there is only one component C_1 that can be instantiated by C_1's home (Fig. 3.6). In fact, C_1 calls the `build` method of the associated assembly object. This assembly object in turn instantiates C_2 and C_3 and establishes

all defined connections between these component instances. After these activities, the assembly object returns to C_1 that finishes its create method.
Of course, C_1 itself can be part of another super component or connected to other components as well. LwCCM defines the end of a component's configuration phase by calling the `configuration_complete` method on each component instance. In the case of a super component, this call must be delegated to all subcomponent instances (Fig. 3.7). Before C_1 can return from `configuration_complete`, it has to lock all private ports to prevent clients from directly accessing contained component instances.

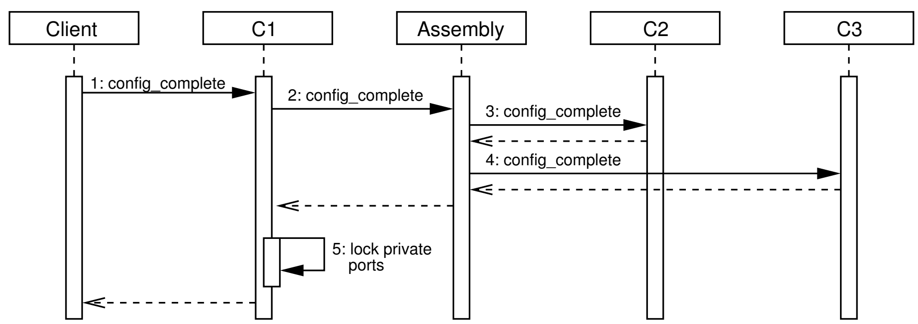


Figure 3.7: Completion of super component configuration phase.

Fig. 3.8 shows how a super component instance is removed: C_1 triggers the assembly object to disconnect and destroy all contained component instances.

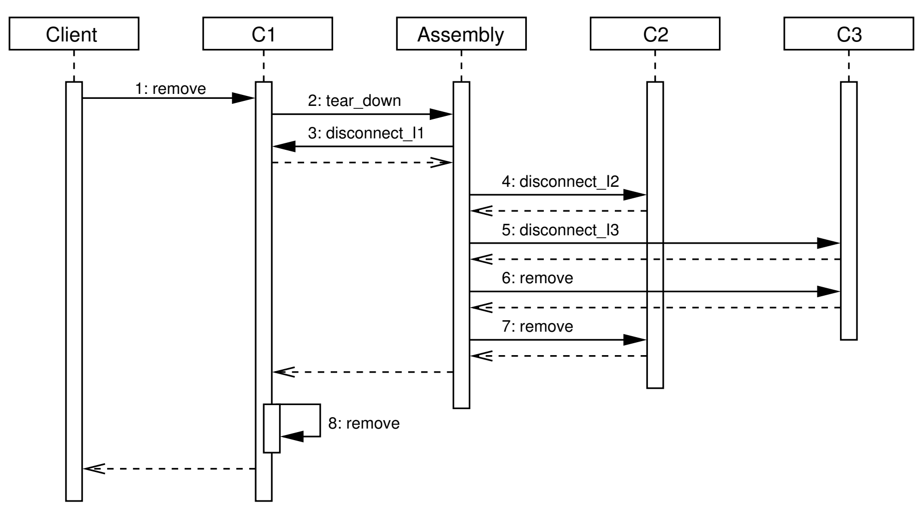


Figure 3.8: Removal of a super component.

A component client can handle the super component in the same way as a regular LwCCM component. Also, a simple LwCCM component can be seen as a special case of a super component with an empty assembly. Thus, both components and assemblies has been reduced to a single concept.

3.5 Remote Component Structure

While we have implemented a local version of CORBA Components, the LwCCM specification defines remote components only. Remote components are built up from CORBA objects that implement defined IDL interfaces. Because of the specified mapping from IDL3 to IDL2, the generated IDL2 files can be processed by every existing IDL compiler. In addition to CORBA stubs and skeletons, remote component logic as well as CORBA component containers must be implemented too. To be compliant to the LwCCM specification, we have developed a way to adapt local components into remote LwCCM components - the **Local Component Adapter Concept** (LCAC).
LCAC allows to add remote communication for each port transparently for business logic. Fig. 3.9 shows how a given local component implementation can be extended to a remote LwCCM component.

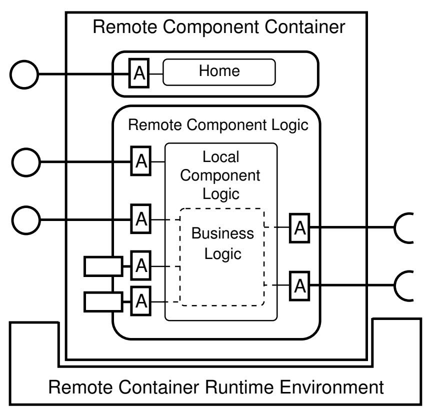


Figure 3.9: A local component can be embedded in a remote component logic that will be managed by a remote component container.

The point is that we can use local components without changing them. Thus, for a remote accessible component that provides at least one remote port, some additional code will be involved.

Remote component logic. A glue code layer is responsible for embedding a local component into a remote CORBA component. This remote component logic hosts a local component. That means, its local component logic and business logic. Such a structure ensures that local ports can be used side by side to remote ports.

Adapter set. For a given IDL interface that defines a component port's syntax, a local and a remote implementation is generated. Using a set of adapter classes, these two worlds can fit together transparently. In addition to component ports, adapters must be provided for component homes as well as the component's equivalent interface.

Remote component container. For each remote component type, a generic component container is used to manage CORBA component instances. In contrast to a local component container that can have a simple structure, a remote container is also responsible for sophisticated *Quality of Service* (QoS) tasks.

Remote container runtime environment. With increasing QoS functionality, the requirements to a remote container runtime environment are growing too. Besides an *Object Request Broker* (ORB), that handles CORBA requests, libraries for multi–threading and process management implementation must be available.

This adapter concept is a powerful tool especially in heterogeneous environments. Besides the choice between local and remote connections, a deployment process can also decide to use different middleware technologies.
The fact that a local component is wrapped by a remote component becomes obvious from Fig. 3.10. All classes of a local component remain unchanged, while some new "remote" classes have been added.
The remote structure is very similar to a local component's structure (Fig. 3.2), thus, we can compare interactions between a local and a remote component with interactions between business logic and local component logic:

Calling component methods. A remote client calls methods on a remote adapter that delegates this calls to a local component which uses a local adapter to delegate these calls to business logic. In each adapter, pre- and a post-invoke processing can take place (Fig. 3.11).

These two indirection layers allow non–functional extensions to components without changes in business logic. This separation of concerns is a cornerstone in component based development.

Invoking callback methods. Callback methods implemented by business logic can be triggered either from local or remote component logic and their corresponding container implementations to control a component's life cycle.

Using context methods. Component business logic uses the `Context` object to access container functionality as well as component receptacles. In the case of remote components, receptacles can be either local or remote ports. Both kinds of receptacles can be accessed via local context object methods. While local receptacles are connected directly to local facets, remote receptacles are intercepted by a receptacle adapter.

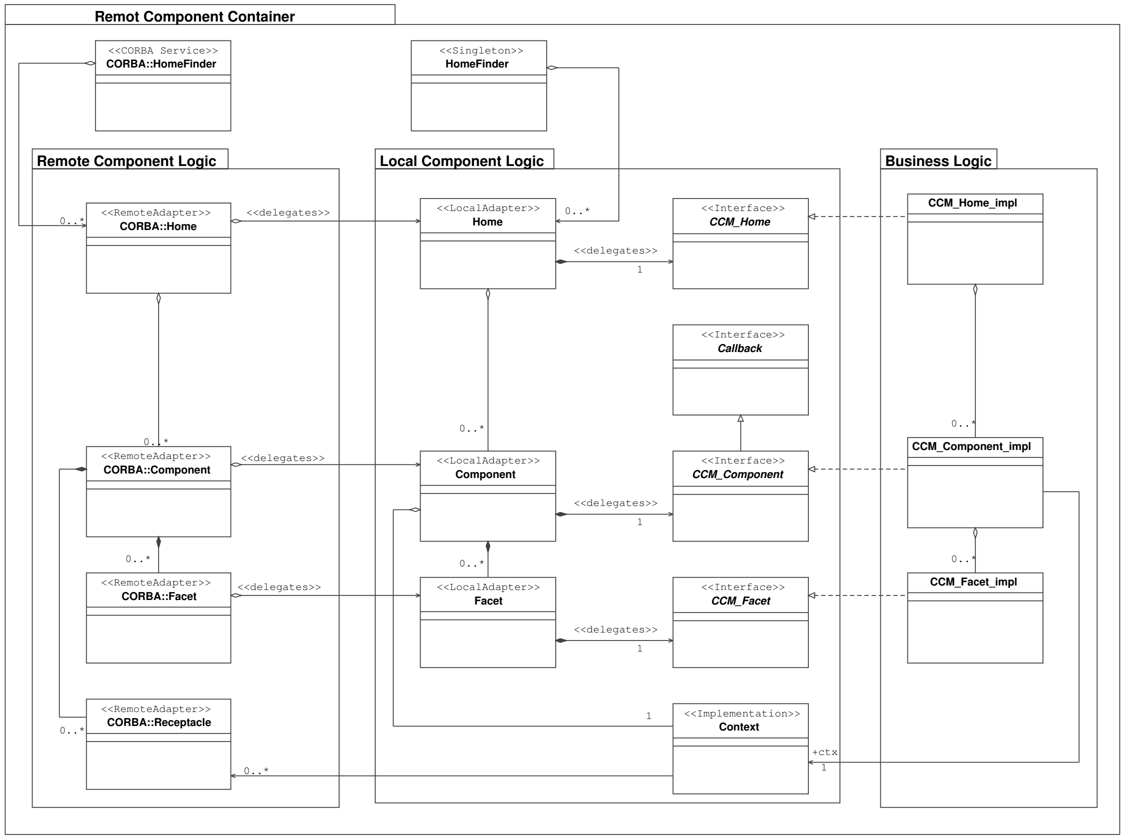


Figure 3.10: Simplified structure of a remote component implementation, showing the relationship between corresponding local and remote components.

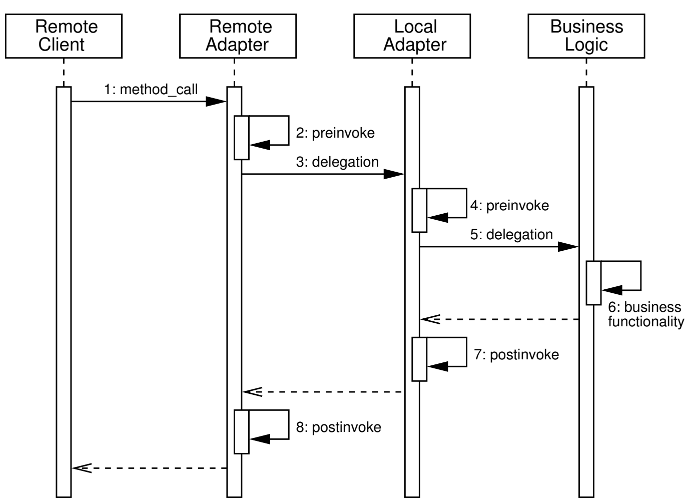


Figure 3.11: A remote call of a component's method is delegated twice allowing pre- and post-invoke processing.

Based on LCAC, an existing LwCCM container implementation could be used to host local components, thus, we could combine an existing CORBA application server with the presented extensions in the context of local components.

Chapter 4

Interface Definition Language

4.1 Introduction

In the CCM Tools framework, a subset of CORBA's Interface Definition Language (IDL) is used to define components, interfaces and parameters.
...
Independent of the used implementation language (e.g. C++, Java, etc.)
...

4.2 Source Files

The IDL specification defines a number of rules for the naming and contents of IDL source files.

4.2.1 File Naming

The names of source files containing IDL definitions must end in `.idl` (for example, we can define a file named `Components.idl`).

4.2.2 File Format

IDL is a free–form language. This means that IDL allows free use of spaces and newline characters. Layout and indentation do not carry semantics, so you can choose any textual style you prefer, but keep in mind that IDL is programming language independent so don't use language specific prefixes or names.

4.2.3 Preprocessing

IDL source files are preprocessed. The preprocessor's behavior is identical to the C++ preprocessor (actually, the CCM Tools use the GNU C preprocessor `cpp`).
The most common use of the preprocessor is for `#include` directives. This permits an IDL definition to use types defined in a different source file. You may also want to use the preprocessor to guard against double inclusion of a file:

```
#ifndef _MYFILENAME_IDL_
#define _MYFILENAME_IDL_

// some IDL definitions

#endif /* _MYFILENAME_IDL_ */
```

4.2.4 Definition Order

IDL constructs (modules, interfaces, type definitions) can appear in any order you prefer. However, identifiers must be declared befor they can be use.

4.2.5 Comments

IDL definitions permit both the C and the C++ style of writing comments:

```
/**
 * This is a legal IDL comment.
 * Note that you can use tools like doxygen to extract
```

```
 * comments from IDL files.
 */

// This comment extends to the end of this line.
```

4.2.6 Keywords

IDL uses a number of keywords, which must be spelled in lowercase (e.g. `interface`, `struct`, etc.). There are three exceptions to this lowercase rule: `Object`, `TRUE` and `FALSE` are all keywords and must be capitalized.

4.2.7 Identifiers

Identifiers begin with an alphabetic character followed by any number of alphabetics, digits, or underscores. Unlike C++ identifiers, IDL identifiers can't have a leading underscore.
Identifiers are case–insensitive but must be capitalized consistently. This rule exists to permit mappings of IDL to languages that ignore case in identifiers (e.g. Pascal) as well as to languages that treat differently capitalized identifiers as distinct (e.g. C++, Java).
IDL permits you to create identifiers that happen to be keywords in one or more implementation languages, but to make life easier, you should try to avoid IDL identifiers that are likely to be implementation language keywords.

4.3 Basic IDL Types

IDL provides a number of build–in basic types. The CORBA specification requires that language mappings preserve the *size* of basic IDL types. To avoid restricting the possible target environments and languages, the specification leaves the size and range requirements for IDL basic types loose.

4.3.1 Integer Types

- `short` (range from -2^{15} to $2^{15}-1$, size ≥ 16 bits)
- `long` (range from -2^{31} to $2^{31}-1$, size ≥ 32 bits)
- `unsigned short` (range from 0 to $2^{16}-1$, size ≥ 16 bits)
- `unsigned long` (range from 0 to $2^{32}-1$, size ≥ 32 bits)

4.3.2 Floating–Point Types

- `float` (IEEE single–precision, size ≥ 32 bits)
- `double` (IEEE double–precision, size ≥ 64 bits)

4.3.3 Characters

- `char` (ISO Latin–1, ≥ 8 bits)
- `wchar` (≥ 16 bits)

4.3.4 Strings

- `string` (ISO Latin–1, variable–length)
- `wstring` (variable–length)

4.3.5 Booleans

Boolean values can have only the values `TRUE` and `FALSE`.

4.3.6 Octets

The IDL type `octet` is an 8–bit type that is guaranteed not to undergo any changes in representation as it is transmitted between processes.

4.3.7 Type any

Type `any` is a universal container type. A value of type `any` can hold a value of any other IDL type (e.g. `long`, `string`, or even another value of type `any`). Type `any` is useful when you don't know at compile time what IDL types you will eventually need to transmit between client and server, you can find out at runtime what type of value is contained in the `any`.

4.4 User–Defined IDL Types

In addition to providing the build–in basic types, IDL permits you to define complex types: enumerations, structures and sequences. You can also use `typedef` to explicitly name a type.

4.4.1 Named Types

You can use `typedef` to create a new name for a type or to rename an existing type. Example:

```
typedef long TimeStamp;
```

Be careful about the semantics of IDL `typedef`. It depends on the language mapping whether an IDL `typedef` results in a new, separate type or only an alias. To avoid potential problems, you should define each logical type exactly once and then use that definition consistently throughout your specification.

4.4.2 Enumerations

An IDL enumerated type definition looks much like the C++ version.
Example:

```
enum Color
{
    red,
    green,
    blue
};
```

This example introduces a type named `Color` that becomes a new type in its own right - there is no need to use a `typedef` to name the type.

4.4.3 Structures

IDL supports structures containing one or more named members of arbitrary type, includeing user–defined complex types.
Example:

```
struct TimeOfDay
{
    short hh;
    short mm;
    short ss;
};
```

This definition introduces a new type called `TimeOfDay`. Structure definition form a namespace, so the names of the structure members need to be unique only within their enclosing structure.

4.4.4 Sequences

Sequences are variable–length vectors that can contain any element type.
Example:

```
typedef sequence<Color> Colors;
```

A sequence can hold any number of elements up to the memory limits of your platform.

4.5 Modules

IDL uses the `module` construct to create namespaces. Modules combine related definitions into a logical group and prevent pollution of the global namespace.
Identifiers in a module need be unique only within that module. Modules do not hide their contents, so you can use a type defined in one module inside another module.
Modules can contain any definition that can appear at global scope. In addition, modules can contain other modules, so you can create nested hierarchies.
Modules can be reopened. Incremental definition of modules is useful if specifications are written by a number of developers (instead of creating a giant definition inside a single module, you can break the module into a number of separate source files).

4.6 Interfaces

4.6.1 Constant Definitions and Literals

4.6.2 Operations

4.6.3 Attributes

4.6.4 User Exceptions

4.6.5 Inheritance

4.7 Components

4.7.1

Appendix A

CCM Tools Commands

A.1 ccmtools

NAME: `ccmtools` - Frontend to start available CCM Tools generators.

SYNOPSIS: `ccmtools TYPE [OPTIONS] FILES`

DESCRIPTION: The `ccmtools` script is used to run a particular component generator backend based on a set of IDL files. Depending on `TYPE` and `OPTIONS` a particular code generator is selected to create the desired output.

TYPE: Currently, the following generator types are supported:

- `c++local`
  Generates local C++ component logic.
- `c++local-test`
  Generates a test client for a pair of local C++ component and mirror component.
- `c++dbc`
  Generates a set of Design by Contract adapters for a local C++ component.
- `idl3`
  Generates IDL3 source files.
- `idl3mirror`
  Generates IDL3 source files for a mirror component.
- `idl2`
  Generates equivalent IDL2 source files.
- `c++remote`
  Generates a set of remote C++ adapters that establish a standard compliant CORBA component where a local C++ component can be embedded.

- `c++remote-test`
  Generates a test client for a pair of remote component and mirror component.

OPTIONS: In addition to the generator types, the `ccmtools` script handles the following options:

- `-a, --application`
  Forces the local C++ generator to create business logic implementation skeletons (`*_impl.*` files).
- `-h, --help`
  Prints out a short description of the available command line parameters.
- `-Ipath`
  Specifies a path that will be handled from a preprocessor to find included IDL files.
- `-o DIR, --output=DIR`
  Specifies the directory where the generated code will be written.
- `-V, --version`
  Prints out the current version of installed CCM Tools.

FILES: This `ccmtools` script can handle single IDL files or a list of IDL files. The following examples show the usage of IDL files:

```
ccmtools idl3mirror -o test/idl3mirror Test.idl
ccmtools c++local -a -o test Test.idl Helper.idl
ccmtools c++local-test -o test *.idl
```

SEE ALSO:

A.2 ccmtools-idl

NAME: `ccmtools-idl` - Run an IDL compiler to generate CORBA stub and skeletons.

SYNOPSIS: `ccmtools-idl OPTION FILES`

DESCRIPTION: The `ccmtools-idl` script is a IDL compiler wrapper for Mico ORB and Java ORB, and hides the different call notations. This script also allows to process more than one IDL file at the same time. Note that this script assumes that both IDL compilers are installed correctly.

OPTION: The `ccmtools-idl` script supports of the following options:

- `-h, --help`
  Prints out a short description of the available command line parameters.
- `-Ipath`
  Specifies a path that will be handled from a preprocessor to find included IDL files.
- `--mico`
  Forces the use of Mico's IDL compiler. Thus, the generated stub and skeletons are implemented in C++.
- `--java`
  Forces the use of Java's build in IDL compiler. Thus, the generated stub and skeletons are implemented in Java. Note that Java's IDL compiler only supports CORBA 2.x but no CORBA 3.0 extensions like `component`, `home`, etc.
- `-V, --version`
  Prints out the current version of installed CCM Tools.

FILES: This `ccmtools-idl` script can handle single IDL files or a list of IDL files. The following examples show the usage of IDL files:

```
ccmtools-idl --mico CarRental.idl
ccmtools-idl --java CarRental.idl Customer.idl
ccmtools-idl --mico *.idl
```

SEE ALSO: `Mico manual, Java IDL documentation`

A.3 uml2idl

NAME: `uml2idl` - Convert an UML XMI file into an IDL and an OCL file.

SYNOPSIS: `uml2idl XMI-FILE PREFIX`

DESCRIPTION: The `uml2idl` script runs a Java program that converts a UML diagram stored in an XMI 1.1 file into corresponding IDL and OCL files. The IDL file is created in respect to the *UML Profile for CCM*, while the OCL file collects all OCL expressions defined in the UML diagram.

XMI-FILE: That's the name of the input XMI 1.1 file which holds the UML class diagram (e.g. when using MagicDraw 9.0, the file name looks like `Name.xml.zip`).

PREFIX: The generated IDL and OCL files are named `PREFIX.idl` and `PREFIX.ocl`.

SEE ALSO: `UML Profile for CORBA, UML Profile for CCM`

Appendix B

CCM Tools Installation

B.1 Prerequisites

To install the CCM Tools, the following programs must be available:

Java SDK $\geq$ **1.5.x** (`http://java.sun.com/j2se`)

Apache Ant $\geq$ **1.6.x** (`http://ant.apache.org`)

Python $\geq$ **2.4.x** (`http://python.org`)

cpp $\geq$ **3.3.x** (`http://www.gnu.org`)

To build the generated C++ components, we also need:

Confix $\geq$ **1.5.x** (`http://confix.sourceforge.net`)

gcc $\geq$ **3.3.x** (`http://www.gnu.org`)

mico $\geq$ **2.3.11** (`http://www.mico.org/`)

B.2 How to get it

The project is hosted at Sourceforge (`http://ccmtools.sf.net`). See the web site for releases and announcements.
You can also subscribe to the `ccmtools-announce` mailing list for CCM Tools release announcements. The `ccmtools-users` mailing list provides a forum for discussion about using the CCM Tools.

B.3 Binary distribution

Installing the CCM Tools from a binary package is quite simple:

```
$ tar xvzf ccmtools-x.y.z-bin.tar.gz
```

This package comes with the following structure:

```
ccmtools-x.y.z
|-- bin
|-- lib
`-- templates
    |-- CppLocalTemplates
    |-- CppLocalTestTemplates
    |-- CppRemoteTemplates
    |-- CppRemoteTestTemplates
    |-- IDL2Templates
    |-- IDL3MirrorTemplates
    `-- IDL3Templates
```

Finally, you can set your environment variables:

```
 $ export CCMTOOLS_HOME=<CCM_INSTALL_PATH>
 $ export PATH=$CCMTOOLS_HOME/bin:$PATH

 # Additionally, the following settings are needed for using remote
 # components based on the Mico ORB
 $ export CCM_NAME_SERVICE=corbaloc:iiop:1.2@localhost:5050/NameService
 $ export CCM_COMPONENT_REPOSITORY=${CCMTOOLS_HOME}
 $ export CCM_INSTALL=<MY_INSTALL_PATH>
```

Note that you also need a C++ runtime environment to compile and run the generated components. These C++ runtime packages must be installed from source.

B.4 Source distribution

B.4.1 CCM Tools package:

Installing the CCM Tools from source requires the following steps:

```
$ tar xvzf ccmtools-x.y.z.tar.gz
```

Alternatively, you can check out an up-to-date version from CVS:

```
$ cvs -d :pserver:anonymous@ccmtools.cvs.sf.net:/cvsroot/ccmtools login
Password: <press enter>
$ cvs -d :pserver:anonymous@ccmtools.cvs.sf.net:/cvsroot/ccmtools co ccmtools
```

To build the CCM Tools we use Ant:

```
$ cd ccmtools
$ ant install -Dprefix=<CCM_INSTALL_PATH>
```

Don't forget to set your environment variables properly (as described in the 'Binary distribution' section).

B.4.2 Java runtime package:

To access remote CCM components from Java clients, we have to install a Java client's runtime environment called `java-environment`:

```
$ tar xvzf java-environment-x.y.z.tar.gz
```

Alternatively, you can check out an up-to-date version from CVS:

```
$ cvs -d :pserver:anonymous@ccmtools.cvs.sf.net:/cvsroot/ccmtools \
      co java-environment
```

To build and install the java-environment we use Ant:

```
$ cd java-environment
$ ant install -Dprefix=<CCM_INSTALL_PATH>
```

To used this runtime library from a Java client, don't forget to set the `CLASSPATH` variable:

```
$ export CLASSPATH=<CCM_INSTALL_PATH>/lib/Components.java:$CLASSPATH
```

B.4.3 C++ runtime packages:

As shown in Fig. ??, to compile and run generated CCM components, we need a C++ runtime environment.
To build and install C++ environment packages as well as generated C++ components, we use `Confix`. `Confix` is a build tool that is based on `automake` and `autoconf` - visit the `confix.sf.net` page to read the exhaustive manual.
It's a good idea to create a CCM Tools profile in Confix' configuration file (`.confix`), as described in the Confix manual.

```
ccm_tools_profile = {
    'PREFIX': '<MY_INSTALL_PATH>',        # use your own path!
    'BUILDROOT': '<MY_BUILD_PATH>',       # use your own path!
    'ADVANCED': 'true',
    'USE_LIBTOOL': 'true',
    'CONFIX': {
    },
    'CONFIGURE': {
       'ENV': {
          'CC': 'gcc',                     # use your own path!
          'CXX': 'g++',                    # use your own path!
          'CFLAGS': "-g -O0 -Wall",
          'CXXFLAGS': "-g -O0 -Wall",
          },
       'ARGS': [
        '--with-mico=<MICO_INSTALL_PATH>/lib/mico-setup.sh'
# use your own mico install path!
       ]
    },
}

PROFILES = {
    'ccmtools': ccm_tools_profile,
    'default' : ccm_tools_profile
}
```

It's important that you substitute your own paths in the `.confix` file.
We can configure the `ccm_tools_profile` as default profile, thus we don't need to use the `--profile=ccmtools` confix option. Additionally, we advise to set the `ADVANCED` flag to `true` instead of using the `--advanced` command–line option.
To install the CCM Tools runtime packages, the following steps are needed:

```
 $ tar xvjf wx-toolsbox-x.y.z.tar.bz2
 $ cd wx-toolsbox-x.y.z
 $ confix.py --bootstrap --configure --make --targets="install"
```

```
$ tar xvjf wx-utils-x.y.z.tar.bz2
$ cd wx-utils-x.y.z
$ confix.py --bootstrap --configure --make --targets="install"

$ tar xvzf cpp-environment-A.B.X.tar.gz
$ cd cpp-environment
$ confix.py --packageroot=`pwd`/ccm --bootstrap --configure \
            --make --targets="install"
```

Note that you can alternatively check out an up-to-date version of the `cpp-environment` package from CVS:

```
$ cvs -d :pserver:anonymous@ccmtools.cvs.sf.net:/cvsroot/ccmtools \
      co cpp-environment
```

Perfect, all tools and libraries have been installed and are ready to work!

Appendix C

CORBA Component Model

Developing CORBA applications that make use of advanced features of the ORB and rely on services such as security, notification, persistent state and transactions requires a substantial development effort. The OMG addresses these problems by introducing the concept of CORBA components.

C.1 CORBA Component definition

CCM Component Definition: A component is a basic meta–type in CORBA 3.0 and is denoted by a component reference. A component type is a specific, named collection of features that can be described by an IDL component definition. A component type encapsulates its internal representation and implementation.

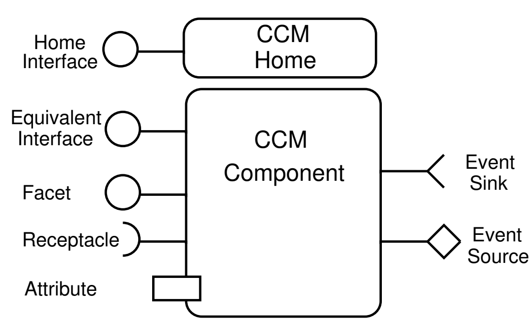


Figure C.1: Pictorial representation of a CCM component that supports a `Home` and an `Equivalent` interface as well as synchronous (facet, receptacle) and asynchronous ports (event source, event sink).

CCM defines a component architecture and a container framework in which the component life cycle takes place. In the CCM specification [?], the following component types are defined:

Service Components do not have any state. The lifetime of service components is restricted to the lifetime of a single method call. A service component is equivalent to a stateless EJB session bean.

Session Components have transient state. Typically, a session component will have the lifetime of a client interaction. Session components are equivalent to stateful EJB session beans.

Process Components have persistent state but no primary key. They are used to model business processes, usually tasks with a well–defined lifetime.

Entity Components have persistent state and a primary key. They are used to model persistent entities in a database that may have transactional behavior. CCM defines two forms of persistence support:

- **Container–managed Persistence** (CMP): The component developer simply defines the state that is to be made persistent and the container automatically saves and restores state as required.
- **Self–managed Persistence** (SMP): The component developer assumes the responsibility for saving and restoring state when requested to do so by the container.

The external view of a CCM component (Fig. C.1) is defined by the following interfaces:

- **Component Home Interface:** describes an interface for managing instances of a specific component type. The home interface may define *Factory Methods* and *Finder Methods* to create and retrieve component instances. A home definition can optionally have *Supported Interfaces* that means that the home interface inherits from these interfaces.

- **Component Equivalent Interface:** is the component's main interface. *Attributes* and *Supported Interfaces* are included in the equivalent interface as well as navigation methods to access the component's ports.

- **Provided Interfaces:** a component type may provide several implemented interfaces to its clients in the form of *Facets*. Facets are intended to be the primary vehicle through which a component exposes its functional application behavior to clients during normal execution. Provided interfaces follow the concept introduced by the *Extension Interface* pattern [**?**].

- **Used Interfaces:** a component definition can describe the ability to use object references upon which the component may invoke operations. When a component accepts an object reference in this manner, the relationship between the component and the referent object is called a connection. The conceptional point of connection is called a *Receptacle*.

In addition to the presented interfaces, CCM supports a publish/subscribe event model. **Event Sources** hold references to consumer interfaces and invoke various forms of push operations to send events. Component **Event Sinks** provide consumer references, into which other entities push events. An *Emitter* can be connected to at most one provider, while a *Publisher* can be connected to an arbitrary number of consumers. The possible dependencies between these interfaces are defined in the CCM **Interface Repository Metamodel**.

All interfaces of a CORBA component are described in the OMG **Interface Definition Language** (IDL) which is part of the CORBA 3.0 specification. The use of IDL makes the component definition independent of programming languages. The OMG has defined language mappings that describe the realization of IDL constructs in a particular programming language.

To describe the structure and state of component implementations, the OMG defined the **Component Implementation Definition Language** (CIDL) as a superset of the *Persistent State Definition Language*. The **Component Implementation Framework** (CIF) defines the programming model for constructing component implementations. The CIF uses CIDL descriptions to generate programming skeletons that automate many of the basic behaviors of components.

C.2 CORBA Component Container

The CCM architecture (Fig. C.2) is very similar to EJB. Components run in a **CCM Container** that provides the runtime environment for CORBA components.

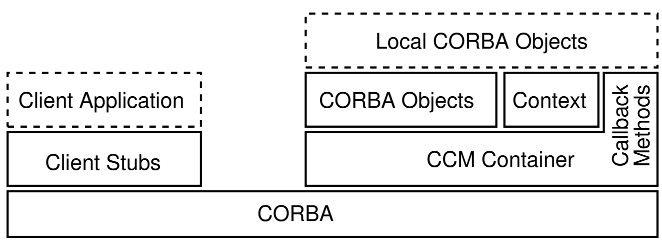


Figure C.2: The CORBA Component Model (CCM) architecture.

Containers are built on top of the *Object Request Broker* (ORB), the *Portable Object Adapter* (POA) and CORBA services and define three forms of interfaces:

- **Internal Interfaces** are local CORBA interfaces that provide container functions to the CORBA component. Internal interfaces are used by the component developer and provided by the container.

- **Callback Interfaces** are local CORBA interfaces that are invoked by the container and implemented by a CORBA component.

- **External Interfaces** are remote CORBA interfaces that describes the contract between the component developer and the component client. External interfaces are used by the client and implemented by the component developer. All remote calls are made on the container's implementation of external interfaces and delegated to local CORBA objects that implement the component's functionality (*Interceptor* pattern).

When a component instance is instantiated in a container, it is passed a reference to its context, a local CORBA interface used to invoke services. This **CCMContext** serves as a bootstrap and provides accessors to the other internal interfaces including access to the runtime services implemented by the container.

The CORBA component model defines container mechanisms and services that manages components at runtime:

- **Instance Pooling.** The life cycle of service components, the component is activated on every operation request, forces the concept of instance pooling to reduce the costs of instance creating and destroying.

- **Life Cycle Management.** To manage the component's lifecyle a container invokes callback methods depending on the container type. To handle all component types, CCM supports two kinds of container APIs, the session container API and the entity container API.

- **Concurrency.** CORBA components support two threading models, *serialize* and *multithread*. A threading policy of serialize means that the component implementation is not thread safe and the container will prevent multiple threads from entering the component simultaneously. A threading policy of multithread means that the component is capable of mediating access to its state without container assistance and multiple threads will be allowed to enter the component simultaneously. Threading policy is specified in CIDL.

- **Transactions.** CORBA components may support either *self–managed transactions* (SMT) or *container–managed transactions* (CMT). A component using SMT is responsible for transaction demarcation via *CORBA Transaction Service* or the container's `UserTransaction` interface. A CMT component defines transaction policies in the associated component descriptor.

- **Security.** The container relies on CORBA security to consume the security policy declarations from the deployment descriptor and to check the active credentials for invoking operations. Access permissions are defined by the deployment descriptor associated with the component.

C.3 Component packaging and deployment

After implementation, a **Packaging** and **Deployment** process must be defined. A package, in general, consists of one or more XML descriptors and a set of files. The descriptors describe the characteristics of the package and point to its various files:

- **Software Package Descriptor.** This descriptor consists of general information about the software followed by one or more sections describing implementations of that software. The descriptor file has a .csd (*CORBA Software Descriptor*) extension.

- **Component Descriptor.** The CORBA Component descriptor specifies component characteristics, used at design and deployment time. A component descriptor file has a recommended .ccd (*CORBA Component Descriptor*) extension.

- **Property File Descriptor.** The property file is used at deployment time to configure a home or component instance. A configurator uses the property file to determine how to set component and component home property attributes. The property file descriptors have a .cpf (*Component Property File*) extension.

C.4 Component assembly

The CCM deployment architecture, defines **Assemblies** build up of existing CCM components. A component assembly archive file contains a set of component archive files and a component assembly descriptor:

- **Component Assembly Descriptor.** A component assembly descriptor consists of elements describing the components used in the assembly, connection information, and partitioning information. It is a template for instantiating a set of components and introducing them to each other. Component descriptors have a .cad (*Component Assembly Descriptor*) extension.

The CCM assembly concept allows the creation of assemblies only at deployment time. At runtime, a single component can not connect itself to another component.

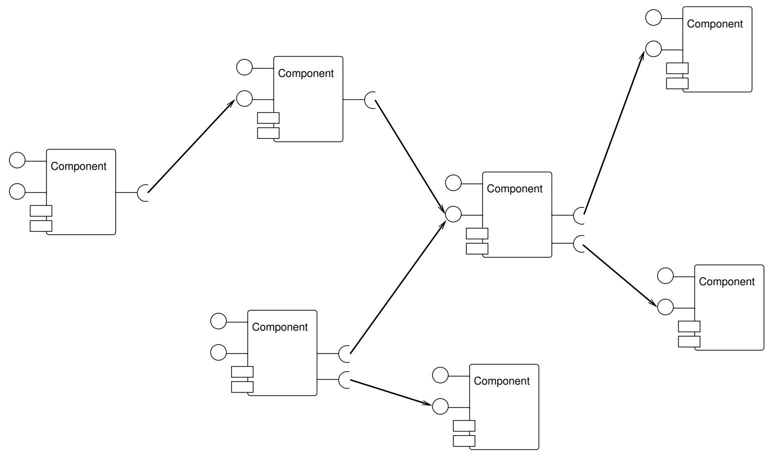


Figure C.3: Component assembly

C.5 Light Weight CORBA Component Model

Many of today's embedded CORBA applications are unable to use the available enterprise CCM due to design constraints. These constraints include small code size in embedded environments and limited processing overhead for performance conservative applications.

To overcome this problem, LwCCM was submitted to the OMG [?]. The purpose of this profile is to specify a lightweight version of the CCM. The principal aim of LwCCM is to have a component model sufficient to compose applications with CORBA components without all optional features that are not part of the "core" capabilities of CCM. The choices made in the profile follow rules established to suit embedded environments:

- **Redundancy.** If several ways of requesting a service exist, only one is retained.
- **Interoperability and Compatibility with full CCM.** During deployment, a leightweight component should be deployable by a full CCM deployment application. Connections between a leightweight component and a full CCM component must be possible. Implementations of leightweight components should be source compatible with the full CCM.
- **Persistence.** The LwCCM does not need to manage any kind of persistence as described in the CCM specification.
- **Transactions.** Transactions are not a feature commonly used in embedded systems thus they are not included in the LwCCM profile.
- **Security.** Security will not be treated in the LwCCM profile.

- **Introspection.** Not all introspection operations are retained in this profile because they are not essential to perform the deployment of components.

- **EJB Integration.** There is no integration of *Enterprise JavaBeans* defined in LwCCM because EJB are not required for embedded targeted environments.

- **Deployment and Configuration.** Instead of the *Packaging and Deployment* chapter of CCM, LwCCM is based on the OMG *Deployment and Configuration* specification [**?**]. This includes also the definitions of component and assembly descriptor files and their XML DTDs.

- **CCM Implementation Framework.** The whole *Component Implementation Definition Language* (CIDL) chapter as well as the *CCM Implementation Framework* (CIF) chapter are excluded from the LwCCM profile.

  The CIDL is redundent with IDL definitions because all functional descriptions of the component (facets, reseptacles, events and attributes) is done with the IDL files. The way to assign a component category (service or session) to a component can be done via an XML description file that will be used with the IDL files to generate container code and skeletons.

This profile tries to be as compliant as possible to the OMG **Minimum CORBA** and **Lightweight Services** specifications [**?**, **?**].