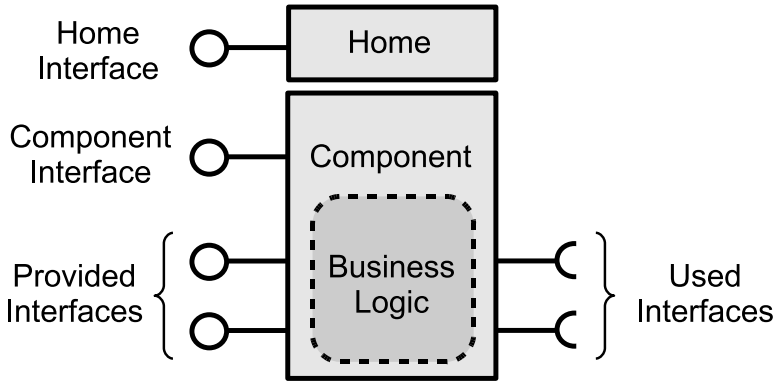
http://ccmtools.sourceforge.net

CCM Tools

User's Manual



Egon Teiniker

January, 2007

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.

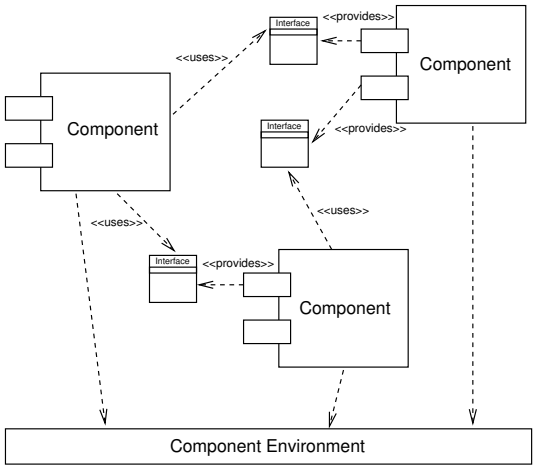


Figure 1.1: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

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. **??** should help to structure the diffent concepts in context of software components.

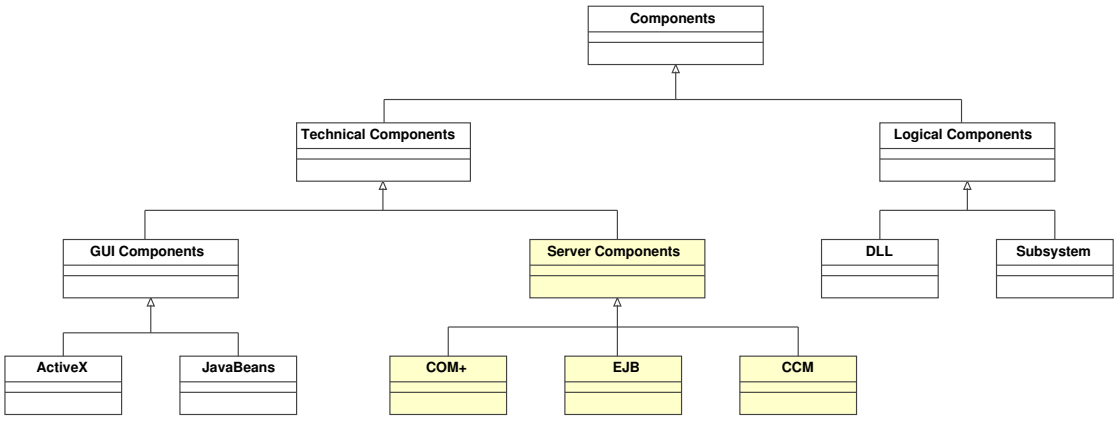


Figure 1.2: A taxonomy of components.

- **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

Hello World Example

As a quick tour through CCM Tools, we implement a simple Hello World component example. Each development step and developer role will be described in more detail in one of the next sections, here we give a general overview.

Step 1: We define a component using the *Interface Definition Language* (IDL). This simple component only provides a single interface containing a single method. Don't forget to define a home for this component type. The following IDL definitions are stored in a file called `Hello.idl`:

```
module world
{
    interface Hello
    {
        string sayHello();
    };

    component Server
    {
        provides Hello hello;
    };

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

Step 2: Generate a uniform IDL3 structure from the single `Hello.idl` file:

```
> ccmidl -idl3 -o server/idl idl3/Hello.idl
```

This uniform IDL3 structure separates between interfaces (including parameter type and exception definitions) and components (including their homes). Such a separation makes sense because an interface can be used by many component definitions.

```
server/
 |-- idl
 |   |-- component
 |   |   `-- world
 |   |       |-- Server.idl
 |   |       `-- ServerHome.idl
 |   `-- interface
 |       `-- world
 |           `-- Hello.idl
```

Step 3: Generate an empty component skeleton from the uniform IDL3 structure:

```
> ccmtools c++local -o server/interface \
                    -Iserver/idl/interface \
                    -Iserver/idl/component \
                    server/idl/interface/world/*.idl

> ccmtools c++local -a -o server/component/Server \
                    -Iserver/idl/interface \
                    -Iserver/idl/component \
                    server/idl/component/world/Server*.idl
```

CCM Tools generate the following file structure which represents a local component's implementation. Code contained in the `GEN_*` directories establishes the component's structure (= *component logic*), while code stored in the `Server` directory represents the functional part of a component (= *business logic*).

```
server
 |-- idl
 |-- component
 |   `-- Server
 |       |-- GEN_ccmtools_local_world
 |       |-- GEN_ccmtools_local_world_share
 |       |-- ServerHome_impl.cc
 |       |-- ServerHome_impl.h
 |       |-- Server_hello_impl.cc
 |       |-- Server_hello_impl.h
 |       |-- Server_impl.cc
 |       |-- Server_impl.h
 |       `-- world_ServerHome_entry.h
 `-- interface
     |-- GEN_ccmtools_local_world
     `-- GEN_world
```

Step 4: Implement the component's business logic. The component's business logic must be embedded in the generated component logic. To implement the `sayHello()` method of the `Hello` interface, we extend the generated `Server_hello_impl.cc` file:

```
std::string
Server_hello_impl::sayHello()
    throw(Components::CCMException)
{
    // TODO : IMPLEMENT ME HERE !
    return "Hello from Server component!";
}
```

Step 5: Now we can implement a client that uses the Hello World component. For this simple case, we implement the client as a `_check*` file that will be automatically executed from a `make check` command.

```
   server/component/server
    |-- test
    |   `-- _check_world_Server.cc
```

The following client code snippets are stored in the `_check_world_Server.cc` file:

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

#include <Components/ccmtools.h>
#include <world/ServerHome_gen.h>

using namespace std;
using namespace world;

int main(int argc, char *argv[])
{
    int error = deploy_world_ServerHome("ServerHome");
    if(error)
    {
        cerr << "BOOTSTRAP ERROR: Can't deploy component homes!" << endl;
        return(error);
    }

    try
    {
        Components::HomeFinder* homeFinder =
            Components::HomeFinder::Instance();

        ServerHome::SmartPtr home(dynamic_cast<ServerHome*>(
            homeFinder->find_home_by_name("ServerHome").ptr()));

        Server::SmartPtr component;
        Hello::SmartPtr hello;
```

```
        component = home->create();
        hello = component->provide_hello();
        component->configuration_complete();

        string s = hello->sayHello();
        cout << "sayHello(): " << s << endl;

        assert(s == "Hello from Server component!");

        component->remove();
    }
    catch(Components::Exception& e)
    {
        cerr << "CCMTOOLS ERROR: " << e.what() << endl;
        return -1;
    }
    catch(...)
    {
        cerr << "UNKNOWN ERROR!" << endl;
        return -1;
    }

    error = undeploy_world_ServerHome("ServerHome");
    if(error)
    {
        cerr << "TEARDOWN ERROR: Can't undeploy component homes!" << endl;
        return error;
    }

    Components::HomeFinder::destroy();
}
```

Additionally, we create some marker files which tell `confix` which package name, version and subdirectories should be used.

```
> ccmconfix -confix2 -o server -pname "hello_world" -pversion "1.0.0"
```

To compile the component and run the unit test, simply type:

```
> confix2.py --packageroot=‘pwd‘/server --bootstrap --configure \
            --make --targets=check
```

After all, we are happy to see the following output at the end of the client's build process:

```
sayHello(): Hello from Server component!
PASS: hello_world_component_Server_test__check_world_Server
```

```
==================
All 1 tests passed
==================
```

Of course, to implement a component for a simple 'Hello from Server component!' message is somewhat academical, but this example shows how simple a component development cycle can be. The intent of this section was to define the main activities in component development, which are:

- Define a component's structure using IDL.
- Generate an empty component skeleton (called component logic).
- Implement a component's business logic.
- Implement a component's (test) client.

In the following sections, we will explore each of these steps in more detail. However, keep this big picture in mind.

Chapter 3

Interface Definition Language

3.1 Introduction

In the CCM Tools framework, a subset of OMG's Interface Definition Language (IDL3) is used to define components, interfaces and parameters, as shown in Fig. **??**.

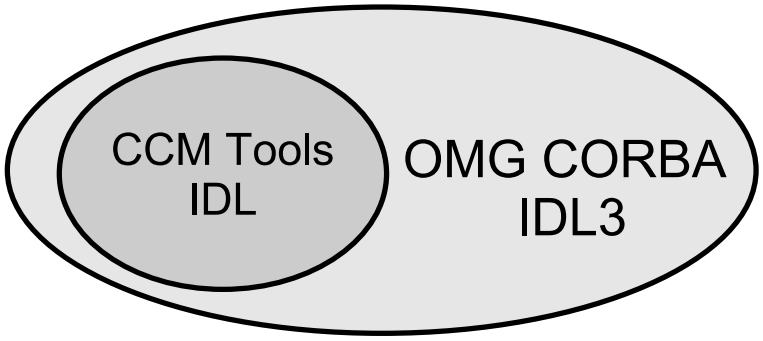


Figure 3.1: CCM Tools support a subset of OMG's Interface Definition Language.

Using an explicit IDL, we can define the structure of component–based software systems completely independent of any particular programming language (e.g. C++ or Java). Also, a clear separation between system design and implementation is guaranteed.

3.2 Source Files

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

3.2.1 File Naming

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

3.2.2 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.

3.2.3 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_ */
```

3.2.4 Definition Order

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

3.2.5 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.
```

3.2.6 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.

3.2.7 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.

3.3 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. The IDL parser searches for the definition of an identifier from the innermost scope outward toward the outermost scope.

Example:

```
module world
{
    /** Some IDL definitions */
};
```

In addition, modules can contain other modules, so you can create nested hierarchies.

Example:

```
module world
{
    /** Some IDL definitions */

     module europe
     {
         /** Other IDL definitions */
     };
};
```

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).

Example:

```
module world
{
    /** Some IDL definitions */
};

// ...

module world
{
    /** Other IDL definitions */
};
```

The CCM Tools don't support global scope IDL definitions, thus, every IDL artefact must be placed within at least one module.

3.4 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.

3.4.1 Integer Types

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

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

3.4.2 Floating–Point Types

- `float` (IEEE single–precision, size $\geq$ 32 bits)
- `double` (IEEE double–precision, size $\geq$ 64 bits)

3.4.3 Characters

- `char` (ISO Latin–1, $\geq$ 8 bits)
- `wchar` ($\geq$ 16 bits)

3.4.4 Strings

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

3.4.5 Booleans

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

3.4.6 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.

3.4.7 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`. It is recommended to use a `typedef` construct to introduce any types in your interface definition files.

Example:

```
typedef any GenericType;
```

3.5 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.

3.5.1 Named Types

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

Example:

```
module world
{
    typedef long TimeStamp;
}; // end of module world
```

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.

3.5.2 Enumerations

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

Example:

```
module world
{
    enum Color
    {
        red,
        green,
        blue
    };
}; // end of module world
```

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.

3.5.3 Structures

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

Example:

```
module world
{
    struct TimeOfDay
    {
        short hh;
        short mm;
        short ss;
    };
}; // end of module world
```

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.

3.5.4 Sequences

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

Example:

```
module world
{
    typedef sequence<Color> Colors;
}; // end of module world
```

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

3.6 Interfaces

The focus of IDL is on interfaces and operations. IDL interfaces define only the interface to an object and say nothing about the object's implementation. This has the following consequences:

- By definition, everything in an interface is public. Things are made private by simply not saying anything about them.
- IDL interfaces don't have member variables. Member variables store state, and the state of an object is an implementation concern.

IDL interfaces form a namespace. You can nest the following constructs inside an interface: constant definitions, attribute definitions, and operation definitions.

Example:

```
module world
{
    interface IFace
    {
        /** Constant definitions  */

        /** Attibute definitions  */

        /** Operation definitions */
    };
}; // end of module world
```

It is important to note that IDL operations and attributes define the only communication path between objects. The kinds of information traveling along the communication path are the parameters, return value, and exceptions of an operation.

3.6.1 Constant Definitions

IDL permits the definition of constants, thus, you can define floating–point, integer, character, string, boolean, and octet constants. IDL does not allow you to define a constant of type `any` nor a user–defined complex type.

Example:

```
module europe
{
    interface ConstantsTest
    {
        const boolean BOOLEAN_CONST = TRUE;
        const octet OCTET_CONST = 255;
        const short SHORT_CONST = -10;
        const unsigned short USHORT_CONST = 7;
        const long LONG_CONST = -7777;
        const unsigned long ULONG_CONST = 7777;
        const char CHAR_CONST = 'c';
        const string STRING_CONST = "1234567890";
        const float FLOAT_CONST = 3.14;
        const double DOUBLE_CONST = 3.1415926;
    };
}; // end of module europe
```

3.6.2 Attributes

An attribute can be used to create something like a public member variable. In fact, an attribute defines a pair of operations the client can call to sent and receive a value. Note that IDL attributes don't define storage or state.

Example:

```
module america
{
    struct Person
    {
        long id;
        string name;
    };

    interface AttributeInterface
    {
        attribute long longAttr;
        attribute double doubleAttr;
        attribute string stringAttr;
        attribute Person personAttr;
    };
}; // end of module america
```

Attributes can be of any type, including user–defined complex types.

3.6.3 Operations

An operation definition can occur only as part of an interface definition, and must contain:

- A return result type
- An operation name
- Zero or more parameter declarations

Example:

```
module austria
{
    interface SimpleInterface
    {
        /**
         * This is the simplest possible operation, because
```

```
             * op requires no parameters and does not return a value.
             */
            void op();
        };
    }; // end of module austria
```

Notice that a parameter must be qualified with one of three **directional attributes**:

- **in**
  The `in` attribute indicates that the parameter is sent from the client to the server.
- **out**
  The `out` attribute indicates that the parameter is sent from the server to the client.
- **inout**
  The `inout` attribute indicates a parameter that is initialized by the client and sent to the server. The server can modify the parameter value, so, after the operation completes, the client–supplied parameter value may have been changed by the server.

Example:

```
module styria
{
    interface AnotherInterface
    {
        long op(in long p1, inout string p2, out double p3);
    };
}; // end of module styria
```

Operation names are scoped by their enclosing interface and must be unique within that interface, so **overloading of operations is not possible in IDL**.

3.6.4 Exceptions

IDL uses exceptions as a standard way to indicate error conditions. Basically, an exception is defined much like an IDL structure, and can contain an arbitrary amount of error information of arbitrary type.
Operations may raise more than one type of exception, and must indicate all the exceptions they may possible raise. It is illegal for an operation to throw an exception that is not listed in the `raises` expression.

Example:

```
module world
{
    exception SuperError
    {
    };

    exception FatalError
    {
        string message;
    };

    module europe
    {
        interface IFace
        {
            long op(in string name) raises (SuperError, FatalError);
        };
    }; // end of module europe
};
```

IDL does not support exception inheritance. That means that you cannot arrange error conditions into logical hierarchies and catch all exceptions in a subtree by catching a base exception.

3.6.5 Inheritance

IDL interfaces can inherit from each other. A derived interface can be treated as if it were a base interface, so in all contexts in which a base interface is expected, a derived interface can actually be passed at runtime (some call it polymorphism).

Example:

```
module america
{
    interface SuperType1
    {
        attribute long attr1;
        long op1(in string str);
    };
}; // end of module america

module europe
{
```

```
        interface SuperType2
        {
            attribute long attr2;
            long op2(in string str);
        };

        interface SubType : america::SuperType1, SuperType2
        {
            attribute long attr3;
            long op3(in string str);
        };
    }; // end of module europe
```

As shown in the example, IDL supports multiple inheritance too.
Note that any form of **operation or attribute overloading is illegal** in IDL.

Chapter 4

Component Model

4.1 Introduction

...

4.2 CCM Tools Component Model

A component type is a specific, named collection of features that can be described by an **Interface Definition Language** (IDL) and encapsulates its internal representation and implementation.

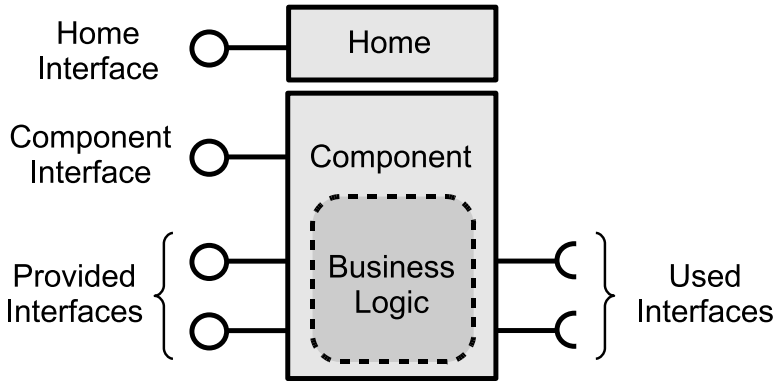


Figure 4.1: CCM Tools component model.

To describe software components, some additional keywords have been introduced to the Interface Definition Language (IDL). A component definition may contain the following surface features:

- Attributes
- Supported interfaces
- Provides interfaces
- Used interfaces

Additionally, a home definition must be declared for every component type. These component homes act as a factory for component instances.

Example:

```
#include <world/CommonInterface.idl>
#include <world/FirstInterface.idl>
#include <world/SecondInterface.idl>

module world
{
    /*
     * A component description collects zero or more surface
     * features to a new component type.
     */
```

```
        component SimpleComponent supports CommonInterface
        {
            /*
             * Supported interfaces can be used to add attributes and
             * operations to the equivalent component interface (it's
             * a kind of interface inheritance).
             */

             /*
              * Component attributes can be used to configure a
              * particular component instance, and are added to the
              * equivalent component interface.
              */
            attribute string version;

            /*
             * Provided interfaces must be implemented by the component's
             * business logic, and can be used by clients or other
             * components.
             */
            provides FirstInterface first;

            /*
             * Used interfaces are implemented by another component.
             * A component's business logic call operations on used
             * interfaces to communicate with other component instances.
             */
            uses SecondInterface second;
        };

        /*
         * A home definition must be declared for every component type.
         */
        home SimpleComponentHome manages SimpleComponent
        {
            /*
             * A home definition may contain zero or more factory
             * definitions. Each factory method can be used to
             * create an instance of the given componet type.
             */
            factory createWithVersion(in string version);
        };
    }; // end of module world
```

A component description will be transformed into a set of equivalent interfaces which define the component's API for clients and other component instances. In the following sections these equivalent interfaces will be described in detail.

4.2.1 Home Interface

4.2.2 Component Interface

The component home interface implicitly provides a `create()` operation to create instances of the managed component type.

4.2.3 Provided Interfaces

4.2.4 Used Interfaces

Chapter 5

Login Example

5.1 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.

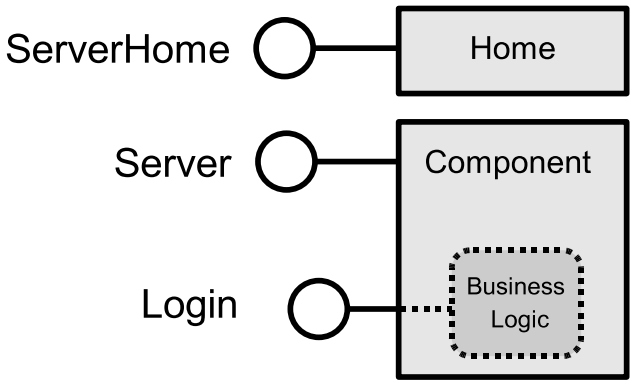


Figure 5.1: A simple component example.

You will see how to define a simple component (Fig. **??**) in the CCM Tool's Interface Definition Language (IDL) and how to use CCM Tools to generate code in different programming languages.

The following sections are structured in form of CCM Tools use cases. Each use case describes a usage scenario for a particular set of CCM Tools features. At the end of a use case, you will have a running component example.

This chapter contains a lot of CCM Tools stuff which will be described very brief, never mind, you can find more exhaustive explanations in the related manual chapters.

5.2 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 may 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 { };
};
```

We store these IDL definitions in a file called `Login.idl`:

```
Login
`-- Login.idl
```

Here we can give only a short description of these IDL artifacts, you can find more information in chapter **??** 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, operations can throw one or more exceptions. Before 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 also 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 in the same 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 the features of the CCM Tools component model.

5.3 IDL Repository Directory

From the `Login.idl` file, the CCM Tools generate an **IDL Repository Directory**. This `idl3repo` directory contains all defined IDL artifacts in separated files and in a uniform structure.

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

After this step, you should see the following directory structure:

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

In this IDL 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.

5.4 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 modular C++ applications.

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

- Model the component's structure in IDL (see section **??**).
- Generate the local component logic.
- Implement the component's business logic.
- Implement a local component client.

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

5.4.1 Generate the local component logic

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, you can see the following directory structure:

```
Login/c++/server
  `-- src
    |-- component
    |   `-- Server
    |       |-- GEN_ccmtools_local_application
    |       |-- GEN_ccmtools_local_application_share
    |       `-- application_ServerHome_entry.h
    `-- interface
        |-- GEN_application
        `-- GEN_ccmtools_local_application
```

Basically, all directories starting with 'GEN_' 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 component non functional behavior. Such changes do neither affect component interfaces nor your business logic implementation which realizes the functional behavior of components.

5.4.2 Implement the component's business logic

Component business logic will 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 `*_impl.* files` 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_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 empty business logic operation skeletons.

It is a good idea to generate these application 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 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
Server_login_impl::isValidUser(const application::PersonData& person)
    throw(Components::CCMException, application::InvalidPersonData )
{
    if(person.name.length() == 0)
        throw application::InvalidPersonData();

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

Now, we can use Confix to build this component example. To tell Confix which directory should be built, `Confix2.*` files must be created in each source code directory. Of course, you can delegate this work to the CCM Tools:

```
> ccmconfix -confix2 -o ./src -pname "login" -pversion "1.0.0"
```

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

```
> confix2.py --packageroot=‘pwd‘/src --bootstrap --configure --make
```

Now we are ready to test this local C++ component implementation.

5.4.3 Implement a 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_application_Server.cc`:

```
#include <Components/ccmtools.h>
#include <application/ServerHome_gen.h>

using namespace std;
using namespace application;

int main(int argc, char *argv[])
{
  if(deploy_application_ServerHome("ServerHome"))
  {
      cerr << "ERROR: Can't deploy component homes!" << endl;
      return -1;
  }

  try
  {
      Components::HomeFinder* homeFinder = Components::HomeFinder::Instance();

      ServerHome::SmartPtr serverHome(dynamic_cast<ServerHome*>(
          homeFinder->find_home_by_name("ServerHome").ptr()));

      Server::SmartPtr server;
      Login::SmatrPtr login;

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

      // Implement your test cases here !!!

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

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

Each functional test case can be inserted into this unit test template shown above. This code snipped is very similar for all simple component unit tests (see section **??** for a more sophisticated test setting). It deploys the component home object, creates a component instance, uses the component's equivalent interface to get a facet, and completes the configuration phase.

After this setup process, we can execute our component test cases (we will discuss the implementation of these test cases later).

Finally, we remove the component instance and undeploy the component home object.

Our first test case 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(277, "eteinik", "eteinik", 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 case shows the component's behavior for an invalid `PersonData` structure. This test expects an `InvalidPersonData` exception to succeed.

```
try
{
    PersonData person(0, "", "", 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 large applications you will need a lot of disk space.

To run these unit tests, we use Confix again:

```
> touch src/component/Server/test/Confix2.dir
```

```
> confix2.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_component_Server_test__check_application_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.

5.5 Use Case 2: Remote C++ Components

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

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

- Model the component's structure in IDL (see section **??**).
- Implement the local component (see section **??**).
- Generate the remote component logic.
- Implement a minimal CORBA server.
- Implement a remote component client.

For a given local C++ component implementation, a set of CORBA adapters and converters can be generated which establish a remote component logic. This step from local to remote C++ components is completely automated by the CCM Tools.

5.5.1 Generate the remote component logic

Currently, CORBA middleware is used for inter–process communication, thus, local C++ interfaces must be adapted to CORBA objects and vice versa.

To realize CORBA interactions, we need CORBA stub and skeleton classes generated by a particular IDL compiler. While components are modeled in IDL3, usual IDL compilers assume IDL2 (without keywords like `component` or `home`). For that reason, a CCM Tools generator transforms IDL3 to IDL2, as defined in the CCM specification:

```
> ccmidl -idl2 -I../../idl3repo/interface -I../../idl3repo/component \
         -o src/component/Server/GEN_ccmtools_corba_stubs           \
         ../../idl3repo/interface/application/*.idl

> ccmidl -idl2 -I../../idl3repo/interface -I../../idl3repo/component \
         -o src/component/Server/GEN_ccmtools_corba_stubs           \
         ../../idl3repo/component/application/Server*.idl
```

In this example, all generated IDL2 files are stored in a directory called `GEN_ccmtools_corba_stubs`:

```
Login/c++/server
|-- component
|   `-- Server
|       |-- GEN_ccmtools_corba_stubs
```

In this `GEN_ccmtools_corba_stubs` directory we call the IDL compiler for every single file (the CCM Tools provide a script that can do this in a single call):

```
> cd src/component/Server/GEN_ccmtools_corba_stubs
> ccmtools-idl -mico -I${CCMTOOLS_HOME}/idl *.idl
> cd ../../../../
```

The transformed IDL2 files include interfaces which are part of the installed CCM Tools, therefore, the IDL compiler needs the include path set to `CCMTOOLS_HOME/idl`.

As a glue between the local C++ interfaces and the CORBA stubs and skeletons, we generate CORBA adapters and converters:

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

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

All generated source files are stored in the `GEN_ccmtools_remote_*` directory:

```
Login/c++/server
‘-- src
    |-- component
    |   ‘-- Server
    |       |-- GEN_ccmtools_remote_application
    |       ‘-- GEN_ccmtools_corba_stubs
```

Remember, there is no reason to check–in generated component logic files into a CVS like system because this code can be generated from the IDL repository at every time.

That's it, we have extended the local C++ component from section **??** to a remote component that can be accessed via CORBA [1] middleware.

5.5.2 Implement a minimal CORBA server

Before implementing a remote client, the remote component must be started as a stand-alone CORBA server. To keep things simple, we implement this CORBA server in a single `_check_ccmtools_remote_application_Server.cc` file:

```
#include <cstdlib>
#include <iostream>
#include <string>

#include <ccmtools/remote/CCMContainer.h>

#include <CORBA.h>
#include <coss/CosNaming.h>
```

[1] There is no technical reason for using CORBA as remote communication mechanism. Future versions of CCM Tools could provide other middleware (e.g. SOAP) adapters too.

```
#include <ccmtools/remote/application/ServerHome_remote.h>
#include <ccmtools_corba_application_Server.h>

using namespace std;
int main (int argc, char *argv[])
{
  int argc_ = 3;
  char* argv_[] =
  {
    "",
    "-ORBInitRef",
    "NameService=corbaloc:iiop:1.2@localhost:5050/NameService"
  };
  CORBA::ORB_var orb = CORBA::ORB_init(argc_, argv_);

  // Register all value type factories with the ORB
  ::ccmtools::remote::register_all_factories(orb);

  // Deploy local and remote component homes
  int error = 0;
  error += deploy_application_ServerHome("ServerHome");
  error += deploy_ccmtools_remote_application_ServerHome(orb, "ServerHome");
  if(!error)
  {
      cout << "ServerHome server is running..." << endl;
  }
  else
  {
      cerr << "ERROR: Can't deploy components!" << endl;
      return -1;
  }
  orb->run();
}
```

We save this `_check_application_ccm_remote_Server.cc` file into the component's `test` directory:

```
Login/c++/server
`-- src
    |-- component
    |   `-- Server
    |       `-- test
    |           `-- _check_ccmtools_remote_application_Server.cc
```

This minimal CORBA server code initializes the ORB and deploys the local and remote component. Remote component deployment also implies the registration of the component home object at the CORBA name service.

Make sure that a CORBA name service is running on your box (e.g. `orbd` which ist a Java tool):

```
> orbd -ORBInitialPort 5050
```

Finally, we run Confix to build all source files and to start the unit test:

```
> ccmconfix -confix2 -o src -pname "login" -pversion "1.0.0"

> confix2.py --packageroot=‘pwd‘/src --bootstrap --configure \
            --make --targets=check
```

At the end of this build process, you should see the following output:

```
ServerHome server is running...
```

5.5.3 Implement a remote component client

We implement a remote component client as a simple unit test. Because this client runs in a separate process (probably on a different machine), we have to generate and build the CORBA stub and skeleton classes for the client–side again:

```
> ccmidl -idl2 -I../../idl3repo/interface -I../../idl3repo/component \
         -o src/component/Server/GEN_ccmtools_corba_stubs            \
         ../../idl3repo/interface/application/*.idl

> ccmidl -idl2 -I../../idl3repo/interface -I../../idl3repo/component \
         -o src/component/Server/GEN_ccmtools_corba_stubs            \
         ../../idl3repo/component/application/Server*.idl
```

In addition to the generated stubs and skeletons, we store the remote test client in a file named `_check_ccmtools_remote_client.cc` in the following directory structure:

```
Login/c++/client
‘-- src
    |-- component
    |   ‘-- Server
    |       ‘-- GEN_ccmtools_corba_stubs
    ‘-- test
        ‘-- _check_ccmtools_remote_client.cc
```

The remote client's implementation follows the structure of a local client but uses the IDL to C++ mapping specified by the OMG.

```
#include <cstdlib>
#include <iostream>
#include <string>

#include <CORBA.h>
#include <coss/CosNaming.h>

#include <ccmtools/remote/CCMContainer.h>
#include <ccmtools_corba_application_ServerHome.h>

using namespace std;
using namespace ccmtools::corba::application;

int main (int argc, char *argv[])
{
    int    argc_    = 3;
```

```
char* argv_[] =
{
    "",
    "-ORBInitRef",
    "NameService=corbaloc:iiop:1.2@localhost:5050/NameService"
};
CORBA::ORB_var orb = CORBA::ORB_init(argc_, argv_);

CORBA::Object_var obj =
    orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var nc =
    CosNaming::NamingContextExt::_narrow(obj);

// Find ComponentHomes in the Naming-Service
obj = nc->resolve_str("ServerHome");
ServerHome_var home = ServerHome::_narrow(obj);

// Create component instances
Server_var server = home->create();
Login_var login = server->provide_login();
server->configuration_complete();

    // Run test cases
try
{
  PersonData person;
  person.id = 277;
  person.name = CORBA::string_dup("eteinik");
  person.password = CORBA::string_dup("eteinik");
  person.group = USER;

  CORBA::Boolean result = login->isValidUser(person);

  if(result)
  {
      cout << "Welcome " << person.name << endl;
  }
  else
  {
      cout << "We don't know you !!!" << endl;
  }
}
catch(InvalidPersonData& e)
{
  cout << "Error: InvalidPersonData" << endl;
}

try
{
  PersonData person;
  person.id = 0;
  person.name = CORBA::string_dup(""); // Here we create an error!!!
```

```
        person.password = CORBA::string_dup("");
        person.group = USER;

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

      // Destroy component instances
      server->remove();
}
```

Again, we use Confix to build and run the test client:

```
> ccmconfix -confix2 -o src -pname "login-remote-client" -pversion "1.0.0"

> confix2.py --packageroot=`pwd`/src --bootstrap --configure \
             --make --targets=check
```

When you see the following output on the console, you have successfully implemented the first distributed component application:

```
Welcome eteinik
OK, caught InvalidPersonData exception!
PASS: login-remote-client_test__check_ccmtools_remote_client
==================
All 1 tests passed
==================
```

The important point is that we did not change the business logic implementation. Instead, we reused the local component and generated a remote layer using the CCM Tools.

5.6 Use Case 3: Local Java Components

In this section, we implement a local Java component and a collocated test client. This CCM Tools use case is intended for developers who implement large and modular Java applications.

The implementation of local Java components requires the following activities:

- Model the component's structure in IDL (see section **??**).
- Generate the local component logic.
- Implement the component business logic.
- Implement a collocated component client.

In the following sections, we use the IDL definitions stored in the IDL repository directory as starting point for all Java code generations.

5.6.1 Generate the local component logic

A local Java component logic includes a set of Java interfaces (`-iface` option) and a pure Java implementation (`-local`) which delegates client calls to the hosted business logic. Local Java component logic can be generated from the IDL repository using the CCM Tools:

```
> mkdir java
> mkdir java/server
> cd java/server

> ccmjava -iface -local                                          \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                           \
          ../../idl3repo/interface/application/*.idl

> ccmjava -iface -local                                          \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                           \
          ../../idl3repo/component/application/Server*.idl
```

These code generation steps result in the following file structure:

```
Login/java/server
‘-- src-gen
    ‘-- application
```

In contrast to C++ component logic, in Java we store all generated files in a temporary `src-gen` directory.

5.6.2 Implement the component business logic

A set of generated interfaces act as the borderline between component- and business logic. A business logic developer is free to realize the implementation classes as long as the right interfaces will be implemented.

As a CCM Tools feature, these implementation classes can be generated with default implementations (mostly empty method skeletons). These application class skeletons are generated in a separate directory called `src`:

```
> ccmjava -app                                                    \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src                                                \
          ../../idl3repo/component/application/Server*.idl
```

Note the strict separation between generated component logic (`src-gen`) and business logic (`src`). As a developer, you are responsible for the `src` directory tree - you should use a CVS like code versioning system for this directory.

```
Login/java/server
|-- src
|   `-- application
|         |-- ServerHomeFactory.java
|         |-- ServerHomeImpl.java
|         |-- ServerImpl.java
|         `-- ServerloginImpl.java
`-- src-gen
```

There is a direct relationship between IDL definitions and generated business logic skeleton classes:

- `ServerHomeFactory.java`
  For each component home, a factory class is generated which implements a `create()` method that acts as entry point for business logic instantiation.
- `ServerHomeImpl.java`
  For each component home, an implementation class is generated which provides an implementation of the default `create()` method.
- `ServerImpl.java`
  For each component, an implementation class is generated which provides default implementations of the component's callback operations.
- `ServerloginImpl.*`
  For each facet, an implementation class is generated which provides skeletons for business logic implementations.

Note that these implementation files can not be overwritten by the CCM Tools. Generators replaces only untouched implementation files, otherwise the re–generated files are stored with a '`.new`' suffix.

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

```
public boolean isValidUser(PersonData person)
    throws CCMException, InvalidPersonData
{
    if(person.getName().length() == 0)
       throw new InvalidPersonData();

    if(person.getId() == 277
       && person.getName().equals("eteinik")
       && person.getGroup() == Group.USER)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Ant is used to build the Java component example. Here is an adequate `build.xml` file:

```
<project name="LoginServer" default="compile">

  <property name="build" location="build" />
  <property name="src" location="src" />
  <property name="src-gen" location="src-gen" />

  <path id="compile.classpath">
      <pathelement path="${java.class.path}" />
  </path>

  <target name="init" description="" >
      <mkdir dir="${build}" />
  </target>

  <target name="compile" depends="init" description="" >
      <javac srcdir="${src-gen}:${src}" destdir="${build}"
             debug="on" source="1.5" target="1.5">
          <classpath refid="compile.classpath" />
      </javac>
  </target>

  <target name="clean" description="" >
      <delete dir="${build}" />
  </target>
</project>
```

You can start the Ant build process with:

```
> ant
```

OK, you have implemented the firts Java component.

5.6.3 Implement a collocated component client

To test the local component, a simple `ClientLocal` class is implemented in the component's `src` directory:

```
Login/java/server
|-- src
|   |-- ClientLocal.java
|   `-- application
`-- src-gen
```

The following listing shows a client's setup and tear down code. Before a component type can be used, we call the component's `deploy()` method which instantiates the component home object and register it to the local `HomeFinder` singleton. On the other hand, before we terminate an application we call a component's `undeploy()` method to free the registered component home object.

```
import application.*;
import Components.HomeFinder;
import ccmtools.local.ServiceLocator;

public class ClientLocal
{
    public static void main(String[] args)
    {
        try
        {
            ServerHomeDeployment.deploy("ServerHome");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // TODO: client's business logic implementation

        try
        {
            ServerHomeDeployment.undeploy("ServerHome");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

The next listing shows a client's business logic implementation. Note that all interactions between client and component are based on generated interfaces.

```
try
{
    HomeFinder homeFinder = HomeFinder.instance();
```

```
    ServerHome home = (ServerHome) homeFinder.find_home_by_name("ServerHome");
    Server server = home.create();
    server.configuration_complete();
    Login login = server.provide_login();

    try
    {
        PersonData person = new PersonData(277, "eteinik", "eteinik", Group.USER);
        boolean result = login.isValidUser(person);

       if (result)
       {
           System.out.println("Welcome " + person.getName());
       }
       else
       {
           System.out.println("We don't know you...");
       }
    }
    catch (InvalidPersonData e)
    {
        System.err.println("Error: InvalidPersonData!");
    }

    try
    {
        PersonData person = new PersonData(0, "", "", Group.USER);
        login.isValidUser(person);
        assert(false);
    }
    catch (InvalidPersonData e)
    {
        System.err.println("OK, caught InvalidPersonData exception!");
    }

    server.remove();
}
catch (Exception e)
{
    e.printStackTrace();
}
```

To run this test, start the Ant build process and execute the local client from the command line:

```
> ant

> java -enableassertions                                \
       -cp $CCMTOOLS_HOME/lib/ccm-runtime.jar:./build \
       ClientLocal
```

Now, you should see the following console output:

```
Welcome eteinik
OK, caught InvalidPersonData exception!
```

Well done!

5.7 Use Case 4: Remote Java Components

In this CCM Tools use case, we implement a remote Java component and a local test client which uses a Java client library component. This use case is adequate for developers who implement large and distributed Java applications.

The implementation of remote Java components requires the following activities:

- Model the component's structure in IDL (see section **??**).
- Implement the local component (see section **??**).
- Generate the remote component logic.
- Implement a minimal CORBA server.
- Generate the client library component.
- Implement a local component client.

You will see in the next sections that most of the development steps needed to extend local components to remote Java components can be completely automated by the CCM Tools.

This section assumes that there is already a local Java component implementation which can be transform into a remote one.

5.7.1 Generate remote component logic in Java

We use CORBA middleware to overcome process boundaries. For noncritical applications you can use the Java build–in CORBA ORB. To turn a local Java component into a remote component, we have to generate CORBA stubs and skeletons as well as a bunch of adapter and converter classes.

Generation of CORBA stubs and skeletons includes an IDL3 to IDL2 transformation and multiple calls to an external IDL compiler. We can use the CCM Tools script `ccmtools-idl` to call Java's build–in IDL compiler:

```
> ccmidl -idl2                                                  \
         -I../../idl3repo/interface -I../../idl3repo/component \
         -o ./src-gen/idl2                                     \
         ../../idl3repo/interface/application/*.idl

> ccmidl -idl2                                                  \
         -I../../idl3repo/interface -I../../idl3repo/component \
         -o ./src-gen/idl2                                     \
         ../../idl3repo/component/application/Server*.idl

> ccmtools-idl -java                                    \
               -I${CCMTOOLS_HOME}/idl -I./src-gen/idl2  \
```

```
              -o ./src-gen                                \
              ./src-gen/idl2/*.idl
```

Another pair of CCM Tools calls create CORBA adapter and converter classes:

```
> ccmjava -remote                                                \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                           \
          ../../idl3repo/interface/application/*.idl

> ccmjava -remote                                                \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                           \
          ../../idl3repo/component/application/Server*.idl
```

After all, you can see the following directory structure:

```
Login/java/server
‘-- src-gen
    |-- application
    |   ‘-- ccm
    |       |-- local
    |       ‘-- remote
    ‘-- idl2
```

Each single file in this `src-gen` directory has been generated, so, there is no need for a CVS like check–in.

5.7.2 Implement minimal CORBA server in Java

For starting up a remote component, a minimal CORBA server can be implemented which initialize the ORB (by passing command line parameters), deploys the remote component home and runs the ORB.

```
import org.omg.CORBA.ORB;
import ccmtools.local.ServiceLocator;

public class Server
{
    public static void main(String[] args)
    {
        try
        {
            // Set up the ServiceLocator singleton
            ORB orb = ORB.init(args, null);
            ServiceLocator.instance().setCorbaOrb(orb);

            ccmtools.remote.application.ServerHomeDeployment.deploy("ServerHome");
            System.out.println("ServerHome server is running...");
            orb.run();
        }
        catch (Exception e)
```

```
        {
            e.printStackTrace();
        }
    }
```

This server class called `Server` is stored in the `src` directory:

```
Login/java/server
|-- src
|   |-- Server.java
|   `-- application
```

Don't forget to start a CORBA name service, because a component deployment implies the registration of the component home object.

```
> orbd -ORBInitialPort 5050
```

We can run the same Ant build script as we have used for building the local Java component.

```
> ant
```

Finally, we can start the minimal CORBA server to activate our remote component:

```
> java -enableassertions                                  \
       -cp $CCMTOOLS_HOME/lib/ccm-runtime.jar:./build  \
       Server                                             \
       -ORBInitRef NameService=corbaloc:iiop:1.2@localhost:5050/NameService
```

Your console output should look like:

```
ServerHome server is running...
```

5.7.3 Generate a client library component in Java

We have seen that a local component can be extended to a remote component without any business logic changes. To bring the same advantage to the client side, CCM Tools support so called **Client Library Components** which are a local proxies for remote components. Client library components implement the same interfaces as local components and delegate each local call to the corresponding remote components.

Remote clients need CORBA stubs and skeletons of the used IDL interfaces. As a matter of course, this redundant step can be skipped if you develop both remote component and client on the same box.

```
> ccmidl -idl2                                                \
         -I../../idl3repo/interface -I../../idl3repo/component \
         -o ./src-gen/idl2                                     \
         ../../idl3repo/interface/application/*.idl

> ccmidl -idl2                                                \
         -I../../idl3repo/interface -I../../idl3repo/component \
```

```
        -o ./src-gen/idl2                                    \
        ../../idl3repo/component/application/Server*.idl

> ccmtools-idl -java                                  \
               -I${CCMTOOLS_HOME}/idl -I./src-gen/idl2 \
               -o ./src-gen                           \
               ./src-gen/idl2/*.idl
```

On top of CORBA stubs and skeletons we generate the client library component:

```
> ccmjava -iface -clientlib -remote                          \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                       \
          ../../idl3repo/interface/application/*.idl

> ccmjava -iface -clientlib                                  \
          -I../../idl3repo/interface -I../../idl3repo/component \
          -o ./src-gen                                       \
          ../../idl3repo/component/application/Server*.idl
```

All generated files are collected in the temporary `src-gen` directory.

5.7.4 Implement local component client in Java

Based on the generated client library component, we can implement a component client much like a simple local Java component client.

As shown in the following listing, only the client's setup and tear down sections are different to a local component client implementation:

```
import org.omg.CORBA.ORB;
import application.*;
import Components.HomeFinder;
import ccmtools.local.ServiceLocator;

public class Client
{
    public static void main(String[] args)
    {
        try
        {
            ORB orb = ORB.init(args, null);
            ServiceLocator.instance().setCorbaOrb(orb);
            ServerHomeClientLibDeployment.deploy("ServerHome");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // TODO: client's business logic implementation
        //       (see collocated client implementation)
```

```
        try
        {
            ServerHomeClientLibDeployment.undeploy("ServerHome");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

We store the client class in a `src` directory:

```
Login/java/client
|-- src
|   `-- Client.java
`-- src-gen
```

As Ant build script we can reuse the script from the server–side.

```
> ant
```

Make sure that the remote component is running before you start the client with:

```
> java -enableassertions                                  \
       -cp $CCMTOOLS_HOME/lib/ccm-runtime.jar:./build     \
       Client                                             \
       -ORBInitRef NameService=corbaloc:iiop:1.2@localhost:5050/NameService

Welcome eteinik
OK, caught InvalidPersonData exception!
```

Congratulations, the remote Java component is working now!

5.8 Use Case 5: Mirror Component Concept

A serious problem in component testing is to satisfy all component ports without having other components to connect to. In a software system, a component always is connected to other components via facets and receptacles. To develop and test a single component, the CCM Tools support the generation of IDL mirror component definitions.

A **Mirror Component** represents the complement of a given component. For every facet in the original component there is a receptacle in the mirror component and vice versa. Within the mirror component, a developer can implement test cases which describes the desired behavior of a component.

Implementation of local mirror component tests require the following activities:

- Model the component's structure in IDL (see section **??**).
- Implement the local component (see section **??**)
- Generate the mirror component definition.
- Generate the local mirror component logic.
- Implement mirror component test cases.

In the following sections, we will implement mirror component test cases for a local C++ component implementation.

5.8.1 Generate the mirror component definition

To create an IDL mirror component definition from the `Server` component we use the following CCM Tools call:

```
> cd Login

> ccmidl -idl3mirror                                  \
         -Iidl3repo/interface -Iidl3repo/component  \
         -o idl3repo                                  \
         idl3repo/component/application/Server*.idl
```

Now, the IDL repository directory contains the new mirror component definition:

```
Login
|-- idl3repo
|   |-- component
|   |   `-- application
|   |       |-- Server.idl
|   |       |-- ServerHome.idl
|   |       |-- ServerHomeMirror.idl
|   |       `-- ServerMirror.idl
```

These new files are:

- `ServerMirror.idl`
  This file contains the mirror component's IDL definitions:

```
#include <application/Login.idl>

module application {
    component ServerMirror
    {
        uses ::application::Login login;
    };
}; // /module application
```

- `ServerHomeMirror.idl`
  This file contains the IDL definition of the mirror component's home:

```
#include <application/ServerMirror.idl>

module application {
    home ServerHomeMirror
        manages ::application::ServerMirror
    {
    };
}; // /module application
```

Based on these new IDL definitions, we can use existing CCM Tools generators to realize this mirror component.

5.8.2 Generate the local mirror component logic

A mirror component can be generated in the same way as a component under test. In addition to the mirror component logic, the `c++local-test` generator creates a `_check_*` file which handles the test setup where both components will be instantiated and connected by their facets and receptacles.

```
> cd c++/server

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

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

These generated mirror component files are stored parallel to the other component directory:

```
Login/c++/server
‘-- src
    |-- component
    |   |-- Server
    |   ‘-- ServerMirror
    |       |-- GEN_ccmtools_local_application
    |       |-- GEN_ccmtools_local_application_share
    |       |-- ServerHomeMirror_impl.cc
    |       |-- ServerHomeMirror_impl.h
    |       |-- ServerMirror_impl.cc
    |       |-- ServerMirror_impl.h
    |       |-- application_ServerHomeMirror_entry.h
    |       ‘-- test
    |           ‘-- _check_application_Server.cc
```

5.8.3 Implement mirror component test cases

As the mirror component's business logic, we implement two test cases. We used the `-a` option to force the CCM Tools to generate application class skeletons. In the `ServerMirror_impl.cc` class we implement the following code snippet:

```
void
ServerMirror_impl::ccm_activate()
    throw(Components::CCMException)
{
    try
    {
        SmartPtr<Login> login =  ctx->get_connection_login();
        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 << "We don't know you !!!" << endl;
            }
        }
        catch(InvalidPersonData& e)
        {
            cout << "Error: InvalidPersonData!!" << endl;
        }

        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;
        }
    }
    catch(Components::Exception& e)
    {
        cerr << "ERROR: " << e.what() << endl;
    }
}
```

`ccm_activate()` is a callback method which will be called from the component logic during the client's `configuration_complete()` call. Within this callback method, we implement these test cases. To call operations on the original component's facet, we use `get_connection_login()` in the mirror component test case to get the connected receptacle reference.

Finally, we use Confix to build both components and to run the unit test:

```
> ccmconfix -confix2 -o src -pname "login" -pversion "1.0.0"

> confix2.py --packageroot=`pwd`/src --bootstrap --configure --make --targets=check
```

Not surprisingly, you should see the following output on your command line:

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

The mirror component test concept handles the complete test setup for you, thus, you can focus on your test cases only.

5.9 Summary

In this chapter, a lot of different CCM Tools use cases have been shown. Each use case satisfies a particular architectural requirement, but the main advantage of CCM Tools comes from a combination of these use cases.

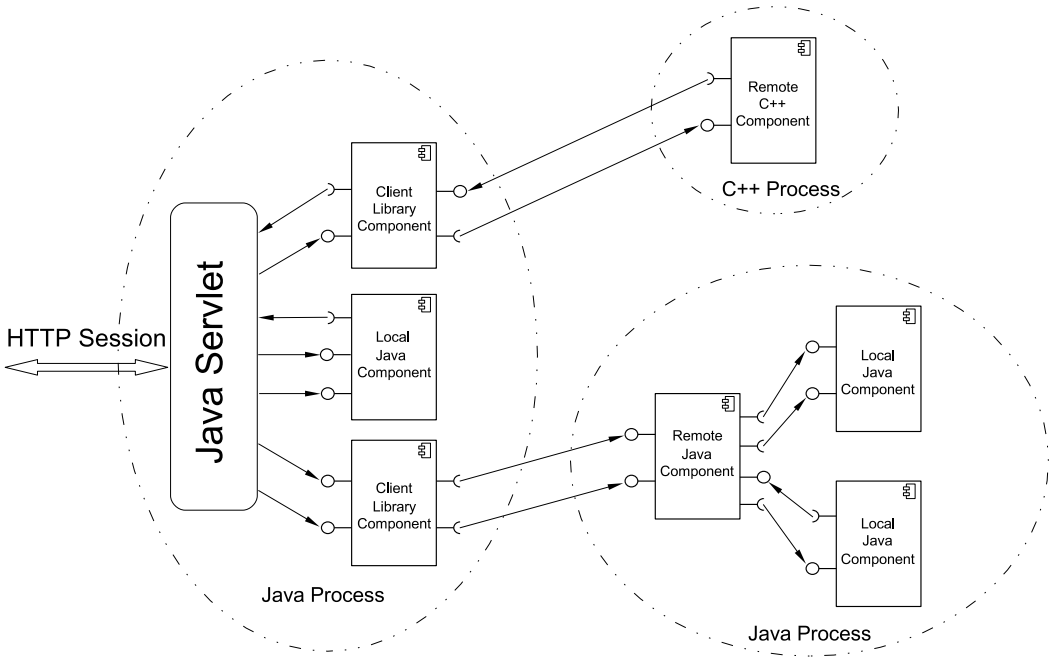


Figure 5.2: Combining different CCM Tools use cases.

Fig. **??** shows a possible scenario where different use cases are combined to realize a heterogeneous and distributed software system. Based on CORBA middleware, we can interact between C++ and Java remote components as well as between remote components and Java client library components.

As component developers, we defined components in IDL and implement business logic in generated implementation skeleton classes. Structural code needed to establish components which can be instantiated and connected via facts and receptacles, either local or remote, is completely generated by the CCM Tools.

Appendix A

CCM Tools Commands

A.1 ccmconfix

NAME: `ccmconfix` - Confix input files generator.

SYNOPSIS: `ccmconfix OPTIONS`

DESCRIPTION: The `ccmconfix` generator can be used to generate Confix input files. Confix needs a `Makefile.py` file in every source code directory (note that Confix2 assumes `Confix2.dir` and `Confix.pkg` files instead of `Makefile.py`).

OPTIONS: The `ccmtconfix` generator handles the following options:

- `-h,--help`
  Prints out a short description of the available command line parameters.
- `-V, --version`
  Prints out the current version of installed CCM Tools.
- `-o, --output <path>`
  Specifies the directory where the generation of Confix input files starts. The generator writes these files in the specified output directory as well as in every subdirectory (use the `etc/ccmtools.properties` file to define a list of directories which will be ignored by the generator).
- `-makefiles`
  Forces the generator to write `Makefile.py` files as used by Confix 1.x.
- `-confix2`
  Forces the generator to write `Confix2.dir` and `Confix.pkg` files as used by Confix 2.x.
- `-pname, --packagename <name>`
  Forces the generator to write `PACKAGE_NAME()` and `PACKAGE_VERSION()` definitions in the top–level `Makefile.py` (or the `Confix2.pkg`) file.

- `-pversion, --packageversion <version>`
  Specifies the used `PACKAGE_VERSION()` number. By default, a "0.0.0" string is used.

SEE ALSO: `Confix Manual`

A.2 ccmidl

NAME: `ccmconfix` - Start script for IDL generators.

SYNOPSIS: `ccmidl OPTIONS FILES`

DESCRIPTION: The `ccmidl` script is used to run all kinds of IDL generator backends based on a given set of IDL input files.

OPTIONS: The `ccmtconfix` generator handles the following options:

- `-h, --help`
  Prints out a short description of the available command line parameters.
- `-V, --version`
  Prints out the current version of installed CCM Tools.
- `-I <path>`
  Specifies a path that will be used from a preprocessor to find included IDL files.
- `-o, --output <path>`
  Specifies the directory where the generated IDL files are stored to.
- `-idl3`
  Generates IDL3 source files for the IDL3 repository directory (for each IDL3 artefact a separate source file will be generated).
- `-idl3mirror`
  Generates IDL3 source files for a mirror component.
- `-idl2`
  Generates equivalent IDL2 source files which can be used as input for a regular IDL compiler.

FILES: CCM Tools start scripts can handle single IDL files or a list of IDL files. The following examples show the usage of IDL files:

```
ccmidl -idl3 -o idl3Repo *.idl
ccmidl -idl3mirror -o idl3Repo Test.idl
ccmidl -idl2 -o corba_stubs Test.idl Helper.idl
```

SEE ALSO:

A.3 ccmjava

NAME: `ccmjava` - Start script for Java generators.

SYNOPSIS: `ccmidl OPTIONS FILES`

DESCRIPTION: The `ccmidl` script is used to run all kinds of Java generator backends based on a given set of IDL input files.

OPTIONS: The `ccmtjava` generator handles the following options:

- `-h, --help`
  Print out a short description of the available command line parameters.
- `-V, --version`
  Print out the current version of installed CCM Tools.
- `-I <path>`
  Specify a path that will be used from a preprocessor to find included IDL files.
- `-o, --output <path>`
  Specify the directory where the generated IDL files are stored to.
- `-iface`
  Generate local Java interface definitions from the given IDL files.
- `-local`
  Generate local Java component logic implementations for the given IDL files.
- `-app`
  Generate buisness logic skeletons for the given IDL files.
- `-remote`
  Generate remote Java component logic implementations for the given IDL files.
- `-clientlib`
  Generate local Java component proxies which are used to access remote components via local Java interfaces.

FILES: CCM Tools start scripts can handle single IDL files or a list of IDL files. The following examples show the usage of IDL files:

```
ccmjava -iface -o  *.idl
ccmjava -iface -local -app -o src-gen Test.idl TestHome.idl
```

SEE ALSO:

A.4 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.
- `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.
- `idl3`
  (!!! deprecated !!!)
  Generates IDL3 source files.
- `idl3mirror`
  (!!! deprecated !!!)
  Generates IDL3 source files for a mirror component.
- `idl2`
  (!!! deprecated !!!)
  Generates equivalent IDL2 source files.

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

SEE ALSO: `ccmidl`

A.5 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.6 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.

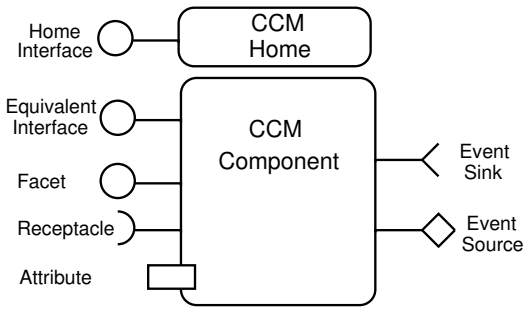


Figure C.1: Pictorial representation of a CCM component that supports a `Home` and an `Equivalent` interface as well as synchronous (facet, receptacle) and asynchronous ports (event source, event sink).

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. **??**) 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. **??**) is very similar to EJB. Components run in a **CCM Container** that provides the runtime environment for CORBA components.

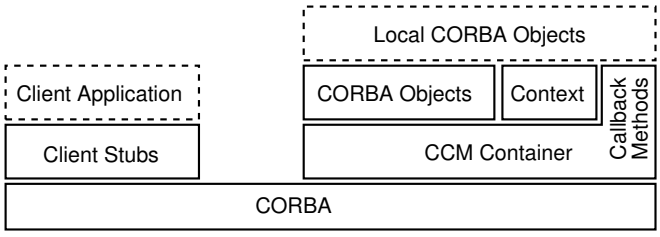


Figure C.2: The CORBA Component Model (CCM) architecture.

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.

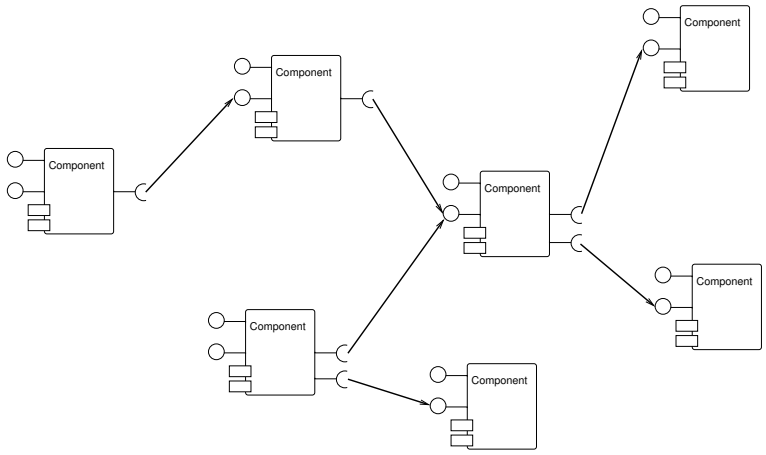


Figure C.3: Component assembly

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 [**?**, **?**].