

Technical Report UNL-DI-6-2002

ComponentJ: The Reference Manual

João Costa Seco Luís Caires

Departamento de Informática
Universidade Nova de Lisboa
Joao.Seco@di.fct.unl.pt

November 2002

(last revision in March 2003)

Abstract: This document describes **ComponentJ**, a component-oriented language in the JVM that can create and manipulate components through expecific architectural constructions. The language ensures consistency of both static and dynamic compositions. We introduce the usage of **ComponentJ** by means of small examples. Next, we show the syntax of the language and detail some of the examples distributed with the compiler.

(this page is intentionally left blank)

Contents

1	Introduction	5
1.1	The ComponentJ programming language	6
1.2	The first component:SquareFloat	8
1.2.1	A Java client for SquareFloat	10
1.3	Programming with dependencies	12
1.4	Programming with composition	14
2	ComponentJ Language Reference	17
2.1	Syntax	17
2.2	Types	17
2.2.1	Primitives Types	18
2.2.2	Port Interfaces	18
2.2.3	Object Interfaces	18
2.2.4	Component Interfaces	19
2.3	Component Definition	19
2.3.1	Static Composition	19
2.3.2	Dynamic Compositions	19
2.3.3	Composition Operations	20
2.4	Component Instantiation	22
2.4.1	New expression	22
2.4.2	Factory method	22
2.5	Expressions	23
2.5.1	Method Call	23
2.5.2	Logic expressions	23
2.5.3	Comparison expressions	23
2.5.4	Arithmetic expressions	23
2.5.5	Composition, instantiation and port assigns	24
2.5.6	Dot expression	24
2.5.7	Literals	24
2.6	Statements	24

3	Programming Examples	25
3.1	Hello World	25
3.1.1	Dynamic Composition	26
3.2	Reusing the Observer Pattern	28
3.3	Dynamic Service Configuration	36
3.4	Native components	44
3.4.1	MySystemPrint component	45
3.4.2	MyVector generic component	47
4	ComponentJ Compiler	51
4.1	Setting up the environment	51
4.2	Compiler executable	51
4.3	ComponentJ framework	52
4.4	Packaging	52

Chapter 1

Introduction

`ComponentJ` is a programming language that raises composition from programming idiom to the level of programming abstractions. This kind of changes has occurred repeatedly in the history of programming languages. Mechanisms that reveal themselves to be simple, usable, and really fundamental are often transformed into programming language constructs where their primitive concepts are represented. Not only this allows for extra readability of the source code but also it allows the reasoning capabilities over the represented structures. `ComponentJ` is a component-oriented language whose implementation is based on an object-oriented model, it sees components as its primary construction mechanism and yet its computational entities are purely object-oriented. Components are instantiated into objects that provide services typed with interfaces in the sense of the common class-based languages. Also, `ComponentJ` objects have views separating the implementation of interfaces in ports with names which must be dereferenced explicitly.

Aggregates of instances based on object cross-referencing are a common programming pattern in object-oriented languages, and are used where inheritance is no longer a natural extension mechanism. This technique is prone to errors as it depends on the careful programmer to avoid them. Thus, verified component architectures instantiated into webs of objects are a valid alternative for the ad-hoc object aggregates.

The development of `ComponentJ` follows a theoretical component calculus that emphasizes on the manipulation of components as values resulting from run-time expressions, and where the correctness of the composition mechanisms is ensured. The calculus was first proposed in [3] and the associated model was motivated and illustrated in other papers and technical reports [4, 2, 5]. There are also some internal technical reports that illustrate the complete formal calculus with parametrically typed components, and the usage of `ComponentJ` in several applications [6][mailer example]

A prototype compiler that implements the language in the JVM (Java Virtual Machine) has been developed and can be downloaded from the URL <http://ctp.di.fct.unl.pt/~jcs/ComponentJ>. The initial implementation of the compiler translates `ComponentJ` code into Java code and then makes use of a standard Java compiler to produce ByteCode that can then be used in any JVM. The distribution package includes the compiler itself together with a small component framework and some examples developed both in `ComponentJ` and in Java. Furthermore, setup instructions are distributed with the compiler and are available in the given URL.

In this manual we first describe the main language features in terms of the component model and then illustrate the usage of the language by means of simple examples. Next, we show the formal syntax of the language constructions followed by a detailed description of some examples that are in the compiler distribution. Finally, we present the compiler options and its consequences.

1.1 The `ComponentJ` programming language

`ComponentJ` is a programming language that manipulates components and defines functionality in an integrated environment. It is based on a formal component model with the following major features:

Component orientation

Component reuse is based on a black-box notion of component, it is based on a public specification of its interface, a component type. The interface of a component is composed by two sets of ports typed with object oriented interfaces where the first set defines the services upon which the component depends and are needed to produce an instance, and the second set of typed ports are the services that the instances of the components yielding this component type will implement. Provided ports are named and represent the possible different views an instances will have. Components are created and manipulated by specific language constructs and can be manipulated as regular computational values. Besides the fact that components are used to produce objects that implement the declared interfaces, they can also be used in run-time compositions that result in new component values.

Component definition

Components can be created from scratch in `ComponentJ`, by composition and adaptation of existing components, or imported from the Java

platform with a minor adaptation effort. The definition of new functionality, either to adapt a component or to implement directly some interface, is done in specific compositional elements, method blocks. Inside these blocks, we can define state variables and methods that the instances of the component will treat in the usual object-oriented way. The code used to create methods is a subset of the Java programming language.

Component dependencies

Every external functionality needed to implement a component must be declared as a requirement of that component. This is achieved through required ports which must be fulfilled before instantiating a component, either by an explicit reference at instantiation time or connected to an available port in a composition. An alternative to declare dependencies is to aggregate the components that implement the desired functionality inside the same composition. This is highly restrictive because it defines *a priori* the implementation of certain services that may be implemented differently in other contexts. Not only components are more flexible as they are smaller.

Dynamic composition mechanisms

By considering components as language values resulting from language expressions `ComponentJ` has the possibility of deciding, at run-time, which elements are used in compositions. Composition expressions are verified at compile time based on the specification of the component values they use, thus making the dynamic composition a type safe operation. Substitutability of component values is checked based on a subtyping relation between component specifications.

Type safety

By assigning static type information to components, objects, and ports `ComponentJ` ensures that all compositions are consistent in their inner structure. Besides the access to the instances ports, the type system also ensures at compile time architectural properties of compositions solely based on the atomic operations of a composition. Furthermore, static type checking allows for the removal of all type information in the resulting code with a single exception, the dynamic loading of new components.

Parametric types

`ComponentJ` uses parametric polymorphism in components, objects and

interfaces. Components are values with open type parameters. However, objects and port interfaces are only used to type values when properly type instantiated.

Platform integration

The type system of `ComponentJ` is supported by the type structure of the JVM. Thus, the integration of `ComponentJ` compositions in the JVM is direct. Stub and skeleton classes are used to enforce the typing of components on Java programs and the usage of `ComponentJ` components without the need for dynamic type verifications.

Structural equivalence of component types

Although interface types are compared by name as in the JVM, component types are compared structurally based on the names and types of their ports. Based on this equivalence relation components from different developers, compliant with the same specification can be used in the same context.

Dynamic Loading

Dynamic loading of components establishes the border between the static typed language and the dynamically typed environment. Given a name the framework that supports `ComponentJ` should be able to check if exists a component with that name and see if the type it exhibits is compatible with the expected type in the `ComponentJ` program. By placing this dynamic checks in the loading process we can safely remove all other type information from the code.

1.2 The first component: `SquareFloat`

In order to illustrate the usage of the language we start by showing the definition of a simple component implementing an unary operation over `float` numbers, in this case it calculates the square of a number.

```
port interface IUnaryOperationFloat {
    float compute(float);
}

component SquareFloat {
    provides IUnaryOperationFloat op;
    declare m {
        float compute(float r) {return r*r;}
    }
}
```

```

    }
    plug m into op;
}

```

This `ComponentJ` code defines the component called `SquareFloat` whose instances calculate the square of a number by means of a method `compute`, that accepts a `float` and returns another, available in port `op` which is typed by the port interface `IUnaryOperationFloat`.

Observing the component definition of `SquareFloat` we see an operation `provides` that defines the availability of an implementation for `IUnaryOperationFloat` in a port called `op`. Furthermore, a method block is defined implementing the `compute` method that is made “visible” to the outside context by a connection to the port `op` performed by a `plug` operation. `SquareFloat` can be tested in a `ComponentJ` expression (in a method of some other component) like

```
(new SquareFloat).op.compute(2.0)
```

which instantiates the component, selects the port `op` of the newly create instance, and calls the method given as argument the real number 2.0.

In the present format, a `ComponentJ` source file is a sequence of interface and component declarations followed by a single `ComponentJ` expression. This expression at the end of a `ComponentJ` source file can be used to test the defined components, it works like the `main` method of a C program. In the future versions of the `ComponentJ` compiler it is foreseeable that this feature will cease to exist. Source files are structured like this for historical reasons related to component calculus. However, the proper usage of the components should be either in other `ComponentJ` components or in Java client applications.

As it was said before, components are typed values of the language and the type of component `SquareFloat` is defined in `ComponentJ` as,

```

component interface TUnaryOperationFloat {
    provides IUnaryOperationFloat op;
}

```

Taking this type as template, other components can be defined and used instead of `SquareFloat`. In particular, a component that computes the symmetric of a float number can defined like this:

```

component MinusFloat {
    provides IUnaryOperationFloat op;
}

```

```

    declare m {
        float compute(float r) {return -r;}
    }
    plug m into op;
}

```

They are both typed as `TUnaryOperationFloat` which means that they are interchangeable. As any regular type, `UnaryOperationFloat` can be used, for instance, to type in the formal parameters of a method like the one below.

```

...
double m(TUnaryOperationFloat c) {
    return (new c).op.compute(2.0);
}
...

```

Values of type `TUnaryOperationFloat` can then be used to call the method. In this case both components are used as arguments. The method `m` returns the result of the “operation” applied to the float value 2.0 depending on the component value used as argument.

```

...m(MinusFloat);
...m(SquareFloat);

```

This concludes this rather useless example whose purpose is solely to show the language capabilities.

1.2.1 A Java client for SquareFloat

A component written in `ComponentJ` can be used in a Java program in two different ways. One, is using the generic interfaces used to implement it in the JVM, which are defined in a small class framework. The other is using a special class, produced by the compiler from the component type called the *stubclass*. It provides a type safe factory method to instantiate the component and gives typed access to all the ports of the component instances.

To illustrate the difference between these two approaches we show the usage of a component by means of the generic framework:

```

class SquareClient {
    public void main(String args[]) {
        ComponentJ.IComponent c = SquareFloat.asValue();
    }
}

```

```

    Object o = c.createInstance(null, null);
    IUnaryOperationFloat op =
        ((_provides_op_IUnaryOperationFloat) o).get_op();
    System.print(op.compute(2));
}
}

```

It requires dynamic type verifications on every access to the component instance. It uses the interface `IComponent` that belongs to the `ComponentJ` framework and the interface `_provides_op_IUnaryOperationFloat` which is generated when a component that provides a port `op` typed with `IUnaryOperationFloat` is compiled.

On the other hand, the usage of a stub class abridges the access to the instances. It is a Java class obtained from a component type, with the same name as the type. So, based on the type `TUnaryOperationFloat`, the `ComponentJ` compiler produces a class that allows type safe access to the instance factory and to the ports of the instances it produces.

```

class SquareStubClient {
    public void main(String args[]) {
        TUnaryOperationFloat c =
            TUnaryOperationFloat.narrow(SquareFloat.asValue());
        TUnaryOperationFloat.InstanceType i = c.createInstance();
        System.out.println(i.get_op().compute(2));
    }
}
}

```

The stub class allows for a typed access to the instance ports and avoids the type cast of the previous example. The access to the component value is achieved by applying a narrow operation to the value returned by the singleton `SquareFloat.asValue()` which returns a reference of the generic type (`ComponentJ.IComponent`). This value is then transformed into an object typed with `TUnaryOperationFloat`.

This example is in the `SimpleComponents` directory of the compiler samples. It uses a Java class as the client of the components and also defines a new component where the components are used as values. The instructions to compile this example can be found in the `README` file in that same directory.

1.3 Programming with dependencies

Component instances are closed values, i.e. they don't depend on any value that may not exist in other contexts. On the other hand, components may depend on a set of services whose existence is required at instantiation time. As a counterpart, inside a component external functionality can only be used when declared in a required port or when it is implemented by one of its inner components.

The following example shows how to declare a required port and how to rely on functionality from the outside context. It describes a component that prints a string on a console. We type the console with the `IPrint` interface below and define our component based on that.

```
port interface IPrint {
    void print(string);
}

port interface IHello {
    void hello();
}

component HelloWorld {
    requires IPrint p;
    provides IHello h;
    declare m {
        void hello() {
            p.print("Hello, World!");
        }
    }
    plug m into h;
}
```

The Java class `HelloWorldClient`, shown below, uses the direct `ComponentJ` framework interface to instantiate the component `HelloWorld` and to access the ports of its instance. The initialization of the ports is done in a somewhat awkward way due to the generic nature of the framework interface. It is done dynamically by indicating the port names and corresponding objects that fulfill them. The types of the objects in the arrays are dynamically tested inside the factory method which ensures a good initialization of the instance. The access to the port is achieved by type casting the instance to the interface `_provides_h_IHello`, which has a getter method for the port.

```

class Console implements IPrint {
    public void print(String s) {
        System.out.println(s);
    }
}

class HelloWorldClient {
    public static void main(String args[]) {
        Object myObject =
            HelloWorld.createInstance(new String[]{"p"},
                                     new Object[]{new Console();});
        ((_provides_h_IHello)myObject).get_h().hello();
    }
}

```

On the other hand, we can again use a stub class to instantiate the component and access the instance in a type safe way. The component specification `THello` allows the production of a stub class with a constructor that has all the requirements as formal parameters. The class `HelloWorldStubClient` provides instances with references compatible with the port type without the need for a type cast. All the type verifications are performed dynamically in the narrow operation of the stub. A minor downside, not yet solved, is that the names of the ports are not verified in the stub, they are only referred on the constructor parameter names. However, the correspondence between parameters and ports of the components is performed correctly independently of its order.

```

component interface THello {
    requires IPrint p;
    provides IHello h;
}

class HelloWorldStubClient {
    public static void main(String args[]) {
        THello c =
            THello.narrow(HelloWorld.asValue());
        THello.InstanceType i = c.createInstance(new Console());
        i.get_h().hello();
    }
}

```

Not only the usage is simpler but it is also safer in the initialization of the ports.

1.4 Programming with composition

From the above examples we have seen that components can be built from scratch with or without the use of external functionality. Despite this possibility, the most common construction mechanism is the composition of existent components to obtain others. The basic composition mechanisms are the aggregation of components and the connection of their ports. Also there is the possibility of modifying their functionality. The following example illustrates a composition based on a native component written in Java, `MySystemPrint`, implementing the `IPrint` interface in a port named `p`. The component type that specifies this native component is:

```
component interface TPrint {
  provides IPrint p;
}
```

`MySystemPrint` and its type `TPrint` are used in the definition of `HelloUniverse` below where it is aggregated as an inner element and used by the method block to implement the functionality of the component. In this case, the functionality of the new component is simple, it consists only on printing a message in the console.

```
component HelloUniverse {
  provides IHello d;
  intro TPrint printer = MySystemPrint;
  declare m {
    printer.p.print("Hello, Universe!");
  }
  plug m into d;
}
```

This component mimics the functionality of the component `HelloWorld` and yet it does not depend on any external service. The `intro`¹ operation integrates `MySystemPrint` in the architecture of `HelloUniverse`, which results in the existence of an instance of it inside each instance of the outer component.

¹In the calculus the syntax of this operation is `uses`, and it is expected that `ComponentJ` changes to accommodate this modification.

This pattern can be considered in some cases an alternative to declaring a dependency and connecting the components on level above. However in this case the component is more specific and less flexible than `HelloWorld` whose port is left open to further configurations.

The sources of other examples can be found with the compiler and some of them are further described in chapter 3. In Particular `MySystemPrint` is described in section 3.4.1. The instructions for compiling the components and clients can be found also in each directory of the samples.

Chapter 2

ComponentJ Language Reference

In this chapter we provide an exhaustive reference of the language constructs. We start by the basic syntax of types and then move to composition operations and statements.

2.1 Syntax

We use a simple grammar to describe the syntax of the language expressions.

Symbol	Meaning
<code>port</code>	normal sans serif font means a token
<i>name</i>	italic sans serif font means non-terminal
<code>[...]+</code>	more than one occurrence (separated by commas or semicolon)
<code>[...]*</code>	zero or more occurrences (separated by commas or semicolon)
<code>[...]?</code>	optionally once
<code>[... ...]</code>	exclusive option
<code>[...]</code>	effectively, something between square brackets

2.2 Types

Types are the base for the static safety of **ComponentJ**. Components, objects and ports are typed with different kinds of interfaces which can have bounded type parameters. Bounded polymorphism allows for typed access to a common set of methods and yet have some flexibility and type safety.

A **ComponentJ** source file is made interface declarations and component static declarations. Finally, there may be a **ComponentJ** expression in the end of the file which is compiled into a Java class with a single **Main** method.

Types for ports, instances and components are defined under the keyword `interface`.

2.2.1 Primitives Types

ComponentJ uses some primitive types with direct correspondence to Java types. Besides `void`, `int`, `float`, and `boolean` we consider `string` as a primitive type instead of a class as in Java. We also want to extend the language to `double`, `long`, arrays and the remaining primitive types in JVM.

2.2.2 Port Interfaces

Port interfaces are compared to Java interfaces, are made of method declarations, and are used to type component ports.

syntax:

```
port interface name <[typevariable]*> [extends [name]+]?{
    [methodprototype ;]*
}
```

Methods are declared without any modifiers as all the methods in a port interface are accessible. The syntax of the method declaration is:

syntax:

```
methodprototype ::= type name([type]*)
```

There is a direct correspondence between port interfaces and Java interfaces. The exceptions are type variables in `ComponentJ` and the unused types and exceptions¹ in Java.

Subtyping is similar to the Java subtyping, explicitly by name. This rule includes type parameters. Type variables used in port interfaces can be instantiated with primitive types.

2.2.3 Object Interfaces

syntax:

¹Exceptions are not yet considered in present version of the compiler but are expected to become part of the language shortly.

```
object interface name ⟨[typevariable [<: type ]? ]*⟩{
    [provides type name;]*
}
```

Object interfaces type instances of components which implement the services declared in a component. Since instances are closed values, they do not have dependencies, its type is made only of provided port declarations. Object interfaces are also type parameterized.

2.2.4 Component Interfaces

syntax:

```
component interface name ⟨[typevariable [<: type ]? ]*⟩{
    [provides type name; | requires type name; ]*
}
```

Component types are defined based on provided and required ports. Type variables can be used in the types of the ports, either to parameterize them or directly in their typing.

2.3 Component Definition

Components are defined in two different scenarios. Either statically in a program toplevel, or as a run-time result of a composition expression.

2.3.1 Static Composition

Static compositions assign component values to names in the top-level of a ComponentJ program. These names can be used in other static component declarations as well as in any other expression in their scope.

syntax:

```
component name ⟨[typevariable [<: type ]? ]*⟩{
    [compositionoperation;]*
}
```

2.3.2 Dynamic Compositions

compose expressions define components that may depend on run-time values. This makes compositions dynamic as their inner values will only be known at run-time.

syntax:

```
compose ⟨[typevariable [<: type ]? ]*⟩{
    [compositionoperation]*
}
```

The values that can cross the expression boundary are the ones with component type. They are copied to the component at the moment of evaluation and used from then on to produce component instances.

```
...
UnaryOperationFloat c = compose {
    provides IUnaryOperationFloat op;
    methods m {
        float compute(float r) { return r/2.0;}
    }
    plug m into op
}
...
(new c).op.compute(2);
```

2.3.3 Composition Operations

The operations available in the scope of a composition range from port declaration to variable and method declarations. On one side, operations for declaring required ports, method blocks, and inner components introduce new local names inside a composition that become available to other operations. On the other side, operations introduce requirements inside a composition like the declaration of provided ports that must be satisfied and the aggregation of inner components with requirements. In order to make a composition consistent, requirements must be satisfied by available and compatible names. This is performed by a connection operation.

Syntax of provided port declarations:

```
provides type name;
```

The provided service is introduced in the component as a requirement that must be satisfied by a compatible source. This declaration reflects it self on the component type stating that the instances of this component will provide this functionality through the port declared in this operation.

Syntax of required port declarations:

requires type name;

The required ports are services that must be available in the instantiation of the component. Their existence is mandatory and therefore their presence inside the component can be trusted. Based on this premise the port name can be used to satisfy internal requirements like the provided ports or the requirements of the inner components.

Syntax of an inner component aggregation:

intro type name = expression;

The newly aggregated component adds its required ports to the inner requirements list of this component and its provided ports to the local name set of the composition. The provided ports can be used explicitly to connect to other ports or implicitly in the method blocks of the component.

Syntax of method block declaration:

```
declare name {
    [statevariabledeclaration | methoddeclaration ]*
}
```

In the original component calculus, state variables and methods are declared independently. However, in order to keep the familiarity with object oriented languages we keep the declaration of variables and methods together in one block. State variables and methods are declared without any modifier because state variables are always considered hidden outside the method block and methods are always visible inside the component and made visible outside exclusively through interface specifications.

The syntax for these declarations are:

```
statevariabledeclaration ::= type name;
methoddeclaration ::= type name([type name]*) {[statement]*}
```

Syntax of plug operation:

plug terminal into terminal;

A plug operation connects two plug terminals that may represent a method block or a port of the component, or a port of an inner component.

terminal ::= x | x.p

Compositions are considered to be consistent when all the inner requirements, provided ports and inner component's required ports, are satisfied through plug operations.

2.4 Component Instantiation

Components are instantiable both in Java and ComponentJ code. In ComponentJ we can use the `new` expression using it to fulfill the open dependencies of the component. On the other hand, by using the ComponentJ base framework one can use a factory method in the component values to produce instances relying on the arguments to fulfill the required ports.

2.4.1 New expression

Components are instantiated in a language expression, `new`. The expression allows for port references to be used to satisfy the requirements of the component. Thus, the expression has two parts, one that uses a component (the one to be instantiated) and another that is a sequence of plug assign expressions that correlate expressions to required ports.

syntax:

```
new expression<[type]*> [ [expression in name]* ]
```

2.4.2 Factory method

The factory method instantiates the component in Java through a dynamically typed and generic interface. It is called `createInstance` and receives two arguments, an array of strings representing the names of the ports and an array of objects that are to be connected to the ports. The order of the ports and expressions makes the correspondence between them. The alternative is to use stub classes that hide this dynamic correspondence. They implement a factory that has parameters typed as the required ports. The arguments are then placed internally in an array as first described and passed to the generic factory method. The advantage of this method comes from the fact that the stub class was generated automatically from the component type and all the correspondences are verified previously. On minor step back is that we must rely on the parameter identifiers to identify the ports. The example above of the definition `MySecondComponent` illustrates both situation of the usage of the factory method and of the stub class.

2.5 Expressions

2.5.1 Method Call

Method calls are always done on one reference. There is no implicit method call like `m()`, it must be preceded by an object (`o.m()`). This is part of the explicit character of the language and the non existence of the self reference and of dynamic binding of methods inside a component.

2.5.2 Logic expressions

Logic expressions follow the Java syntax and only covers the base operators.

Operator	Symbol
OR	<code> </code>
AND	<code>&&</code>
NOT	<code>!</code>

additionally there the literals `true` and `false` can be used as an expression.

2.5.3 Comparison expressions

`ComponentJ` also accepts comparison operators

Operator	Symbol
equality	<code>==</code>
inequality	<code>!=</code>
less than	<code><</code>
greater than	<code>></code>
less or equal	<code><=</code>
greater or equal	<code>>=</code>

They work like in the Java language comparing references when objects are involved.

2.5.4 Arithmetic expressions

Arithmetic operators can be combined using the regular precedences. All operators can be used with numbers (int and float), the `+` operator can also be used to combine strings.

Operator	Symbol
plus	+
minus	-
times	*
divide	/
unary minus	-
remainder	%

2.5.5 Composition, instantiation and port assigns

The composition and instantiation of components are also expressions but they were already described in sections 2.3 and 2.4.

2.5.6 Dot expression

Dot expressions are used to dereference ports of instances or methods of interfaces.

2.5.7 Literals

The literals can express values of several sorts, they are a subset of the JVM literals: integers, float, string, and boolean.

2.6 Statements

The statements in ComponentJ follow once again the Java syntax and are limited to following set: `if`, `while`, `return` and the block statement. Local variables can be declared inside a statement block using the conventional syntax.

localvariable ::= *type name* [= *expression*]?

Chapter 3

Programming Examples

In this chapter we show some programming examples written in `ComponentJ` that are downloaded together with the compiler. We start by a traditional example of printing a “Hello World” message in a console. We use it to demonstrate several features of the language like composing a native component with some extra functionality and dynamically composing the same ingredients based on run-time decisions. Due to the prototypical state of the compiler the compilation execution of the examples can be complicated. However, all samples contain a `README` file and a script to compile it (`compile.sh` or `compile.bat`).

3.1 Hello World

A simple example that takes an existing native component `MySystemPrint` which was built based on the skeleton class of type `TPrint`. The source file for this example is named `HelloComposition.cj`:

```
port interface IPrint {
    void print(string);
}

component interface TPrint {
    provides IPrint p;
}

port interface IDo {
    void doIt();
}
```

```

component HelloComponent {
  provides IDo d;
  intro TPrint p = MySystemPrint;
  declare m {
    void doIt() {
      p.p.print("Hello World!!");
    }
  }
  plug m into d;
}

```

This file defines a component, `HelloComponent` that aggregates the component `MySystemPrint` and makes use of it in method `doIt` to print the message

`Hello World!!`

in the console. The component `MySystemPrint` is described further on in section 3.4.

3.1.1 Dynamic Composition

We use the `Hello World` example to illustrate dynamic composition and extension of interfaces. In addition to the interfaces used in the previous example we need an interface that specifies a factory method. It receives a component value of type `TPrint` and returns another of type `TDo`.

```

component interface TDo {
  provides IDo d;
}

port interface IFactory {
  TDo make(TPrint);
}

```

We then define a component that implements this interface. The `compose` expression inside method `make` uses the parameter `printer` and aggregates its value in a newly created component.

```

component Factory {
  provides IFactory f;
  declare m {
    TDo make(TPrint printer) {

```

```

    return compose {
        provides IDo d;
        intro TPrint p = printer;
        declare mm {
            void doIt() {
                p.p.print("Hello World");
            }
        }
        plug mm into d;
    };
}
}
plug m into f;
}

```

Note that the composition is exactly the same as the one in `HelloComponent`.

The instances of this component can then be used with different components as arguments to call method `make`. We define a component with the type `TPrint` that extends the result of component `MySystemPrint` with a pair of brackets `< >`.

```

component AnotherSystemPrint {
    provides IPrint p;
    intro TPrint printer = MySystemPrint;
    declare m {
        void print(string s) {
            printer.p.print("<"+s+">");
        }
    }
    plug m into p;
}

```

Finally, we use it together with `MySystemPrint` to produce different components out of the same factory.

```

object interface OFactory {
    provides IDo d;
}

```

```

component FactoryTest {
    provides IDo d;
}

```

```

declare m {
  void doIt() {
    OFactory o = new Factory;
    (new o.f.make(MySystemPrint)).d.doIt();
    (new o.f.make(AnotherSystemPrint)).d.doIt();
  }
}
plug m into d;
}

```

This component can then be tested by the `ComponentJ` expression:

```
(new FactoryTest).d.doIt()
```

which produces the following result:

```

Hello World
<Hello World>

```

The compilation of the whole example `HelloWorld` is explained in the `README` file of the corresponding directory.

3.2 Reusing the Observer Pattern

To illustrate the use of generic components we follow the presentation made in [2] which implements the observer pattern as a set of reusable components [1]. This example is fully programmed in the set of samples distributed with the compiler. The implementation of the pattern using components enables the efficient reuse of the whole pattern wherever it is needed instead of following guidelines to program it from scratch. Any component can then be extended to comply with the observer pattern by composition. Not only it increases reuse as it also promotes modularity by keeping separate the list of observers and the core object.

To begin with, let us recall the structure of the pattern we have chosen to implement. There are two major roles in the observer pattern which are represented in two abstract classes (Figure 3.1): the *subject* of observation and the *observers*. These abstract classes are instantiated in concrete classes depending on the application. Then, each concrete observer object registers itself on the concrete subject object to manifest its interest in receiving a notification of a certain event (e.g. a state change). The notification is performed by calling some predefined method on the observer objects and

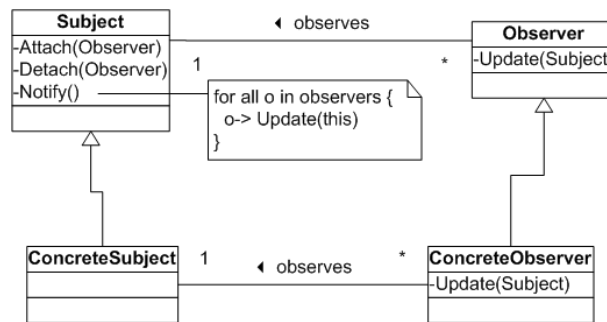


Figure 3.1: The observer pattern UML structure.

passing a reference to the subject as argument. Upon notification, observers can interact with the subject, by calling back some of its methods.

Given this quite generic description of the pattern, questions like the ones below are left open:

1. what interface type is really being observed?
2. what are the events that induce the notification of the observers?
3. what actions will the observers actually perform upon notification?

Our component model allows for an implementation that is flexible in answering the questions and yet respects the structure of the pattern. We will address these questions along with the example but first we start by specifying the basic roles of the pattern in the two port interfaces defined below.

```

port interface IObserver<X> {
    void update(X);
}

port interface ISubject<X> {
    void register(IObserver<X>);
    void unregister(IObserver<X>);
    void notifyObservers();
}
  
```

Note that the type parameter X of port interface `IObserver` is used to adapt this specification to the type of the object of observation which will be used as argument of its `update` method. Also, the parameter types of the methods `register` and `unregister` of the interface type `ISubject` are constrained to a type instantiation of `IObserver` using its own type parameter. This forces the compatibility between the registered observers and the subject of observation.

Given this specification, the observer pattern works as follows: An object typed with a particular instantiation of `IObserver` can be subscribed/unsubscribed to observe a subject typed with the corresponding instantiation of `ISubject`, by calling the methods `register/unregister`. We can then assume that its method `update` will be called whenever the method `notify` is called in the subject. Question (1) above is represented in our program by the type parameter `X` in the interface types, which leaves open (but clearly identified in the architecture) the actual interface type to be observed.

Defining a generic subject component Given the port interfaces defined above, we chose to implement the generic functionality of the pattern in a single component, we called it `CSubject` (Figure 3.2) and programmed it in `ComponentJ` as shown below.

```
component CSubject <Y> {
  requires Y object;
  provides ISubject<Y> subject;
  declare subjectMethods {
    IList<IObserver<Y>> observerList;
    IList<IObserver<Y>> empty;
    void register(IObserver<Y> o) {...}
    void unregister(IObserver<Y> o) {...}
    void notifyObservers() {...}
    void notifyIndex(int i) {...}
  }
  plug subjectMethods into subject;
}
```

where the method block `subjectMethods` is implemented as:

```
...
IList<IObserver<Y>> observerList;
IList<IObserver<Y>> empty;
void register(IObserver<Y> o) {
  if( observerList == empty )
    observerList = (new CList<IObserver<Y>>).list;
  observerList.add(o);
}
void unregister(IObserver<Y> o) {
  observerList.remove(o);
}
```

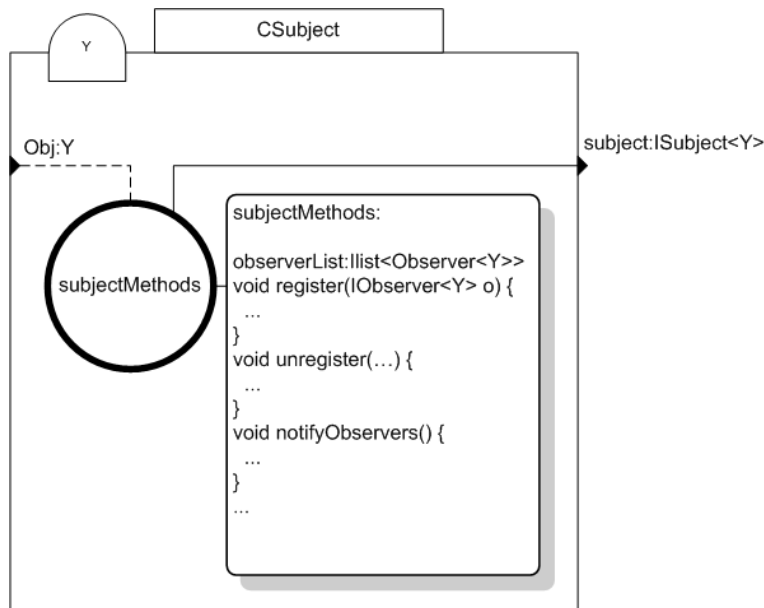


Figure 3.2: The CSubject component

```

void notifyObservers() {
    int k = observerList.getSize();
    int i = 0;
    while( i < k ) {
        subjectMethods.notifyIndex(i);
        i = i + 1;
    }
}
void notifyIndex(int i) {
    observerList.get(i).update(object);
}
...

```

This component provides an implementation for each of the methods of `ISubject`. The basic composition constructs of the calculus that can be seen in the definition are: the declaration of *required* and *provided* ports, used to communicate with the context outside the component (the **requires** and **provides** operations); declaration of *method blocks*, which implement functionality and hold state variables (the **declare** operation); and connections between these inner elements (the **plug** operation). Note that the lack of initializers and the `null` value forces the usage of a dummy variable, `empty`, to properly initialize the `observerList` variable. This is due to the prototypical

character of the language and its compiler. Note that the list of observers is implemented with an instance of the component `CList` which is also included in the sample distributed with the compiler.

`CSubject` is declared at the top level of the source file and its composition is associated to a static name (constant and known at compilation time). The `compose` expression establishes a local name space where all inner elements are visible and available, but references to external entities are inhibited. This means that a composition is completely opaque and all references to the “outside” must be explicitly made either through method parameters or through required ports. A *component value* will then be bound to the name `CSubject` and used in future compositions and expressions.

`CSubject` is specified to have a single required service typed with `X`, made explicit through the `requires` operation that assigns it to the port name `object`. Any composition using this component must satisfy this port with a compliant service by means of a `plug` operation. From another point of view, the required port `object` can be used inside the component as shown in the implementation of the method `notify`, where it is used as argument to call the method `update` of the members of the observers list. On the other hand, the service that the component provides is made available to the outside through port `subject`. Note that port names can be used in two different ways, either to dereference component instances to obtain references to their ports or to connect components inside compositions. Once dereferenced, ports of instances can be treated like object references in the usual object-oriented sense, they can be stored in variables, used as arguments, and their methods can be called.

The remaining elements of the `CSubject` composition are the method block that implements the component’s functionality, and its connection to the port `subject`. State variables are exclusively declared in method blocks and their scope is limited to the block.

In the present example the question concerning the events that trigger a notification, question (2), is highly dependent of the nature of the subject. Therefore, it is only answered when `CSubject` is composed with another arbitrary component to extend it with the subject role. The functionality of both components is scripted in order to intercept calls of the observed interface type and produce notification calls in the inner component playing the subject role, `subj`.

Adapting an object to the subject role Consider the existence of a component that implements a generic collection of objects, `CCollection`, according to the following specification:

```
port interface ICollection <X> {
    void add(X);
    void remove(X);
    X get(int);
}
```

```
component interface TCollection <X> {
    provides ICollection<X> collection;
}
```

The component `CObservableCollection` depicted in Figure 3.3 and defined below adapts `CCollection` to the subject role by intercepting the calls to the original component and triggering events on the subject part of the component. It is the composition of the components `CSubject` and `CCollection` and the scripting block that connects them in a way that causes the desired effect.

```
component CObservableCollection <Y> {
    provides ISubject<ICollection<Y>> subject;
    provides ICollection<Y> collection;

    intro TCollection<Y> col = CCollection<Y>;
    intro TSubject<ICollection<Y>> subj =
        CSubject<ICollection<Y>>;

    declare m {
        void add(Y element) {
            subj.subject.notifyObservers();
            col.collection.add(element);
        }
        void remove(Y element) {
            col.collection.remove(element);
        }
        Y get(int ix) {
            return col.collection.get(ix);
        }
    }
    plug m into collection;
    plug col.collection into subj.object;
    plug subj.subject into subject;
}
```

where the component type `TSubject` is given by the following declaration:

```
component interface TSubject <X> {
  requires X object;
  provides ISubject<X> subject;
}
```

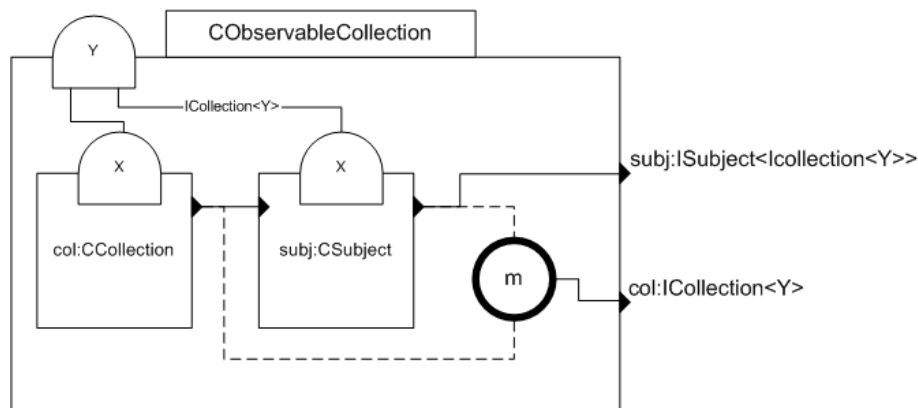


Figure 3.3: The `CObservableCollection` component

Although the interface type of the subject is defined (question 1 above) `CObservableCollection` is still a parametric component on the type of the collected items. It provides two services, one that represents the original functionality of `CCollection` (port `collection`), and another that provides the subject role (port `subject`). `CCollection` and `CSubject` are aggregated through `intro` operations that associate a local name to an internally created component instance. This means that instances of those components will be created inside of each instance of `CObservableCollection` and that the types of `CCollection` and `CSubject` are instantiated and unified in such a way that they become compatible.

In our component model, extension of functionality is achieved by connecting ports of inner components to the outside of the component or by programming methods based on the inner components as it is done in this example in the method block `m`.

Note that the type parameter of `CSubject` is instantiated with the interface type `ICollection<Y>`, where `Y` is a type variable introduced as a type parameter of the whole composition representing the type of the collected items. Therefore, the inner component `subj` requires a connection to a port of type `ICollection<Y>` and `col` provides a port, `collection`, with that the same type. Hence, the connection between the two can be correctly done.

The definition of this component answers the second question of the problem. By adapting an existing component in such a composition we are able to extend it with the subject role of the observer pattern. Finally, to complete the implementation of the pattern, we must address the third question. Note that the definition of the observers is only bounded by the specification of the `IObserver` interface, therefore any component providing a port typed with a compatible instantiation of `IObserver` is acceptable.

Implementing the observer role This adaptation needs a particular solution for each scenario, because it is tightly connected to the environment and cannot be factored out as part of the pattern. `CCollectionObserver` defined below implements the `update` method based on the native component `MySystemPrint` that makes use of a console to print messages. The implementation of `MySystemPrint` is described in section 3.4.1.

```
component CCollectionObserver <X> {
  provides IObserver<ICollection<X>> observer;
  intro TPrint printer = MySystemPrint;
  declare m {
    void update(ICollection<X> col) {
      printer.p.print("Updated the collection");
    }
  }
  plug m into observer;
}
```

In order to use a component specified as an observer together with another specified as a subject it is necessary to make an agreement between the type parameters of both. In this case, by instantiating both `CCollectionObserver` and `CObservableCollection` with the same type arguments.

Assembling the observer pattern Given instances of the components `CObservableCollection` and `CCollectionObserver`, the observer pattern can be played by registering the observers in the subject. The piece of code below is a sample scenario of that procedure.

```
...
CObservableCollection c = new CObservableCollection<string>;
CCollectionObserver o =
  new CCollectionObserver<string>;
c.subject.register(o.observer);
```

```
s.collection.add("Hello");
...
```

where `OObservableCollection` and `OCollectionObserver` are the component instances types below.

```
object interface OObservableCollection {
    provides ISubject<ICollection<string>> subject;
    provides ICollection<string> collection;
}

object interface OCollectionObserver {
    provides IObservable<string> observer;
}
```

In conclusion, `ComponentJ` can be used to define reusable programming patterns. The essential structure of the observer pattern is captured in the `CSubject` component and the `IObservable` and `ISubject` interfaces. The reuse is measured in two different axis. As the first aspect, the code that maintains the list of observers and performs the notification is reused. On the other hand, the parametric specifications of the components allow for reuse with different specifications.

In the next example, these components and types are reused to adapt other components to use the observer pattern in a more complex example.

3.3 Dynamic Service Configuration

In a recent paper [5] we used an example of a dynamically configurable service. We implement it here in `ComponentJ` to illustrate the features of the language.

The scenario of the example is a surveillance system that collects messages from a set of sensors and sends them to the various security guards on duty. In the security staff there are people with different ranks assigned to different missions at different locations using electronic devices to communicate with the system. Those devices range from desktop computers, PDAs in a wireless network, until cell phones reachable through SMSs. All these situations imply different treatment of the messages being delivered to the guards either in the content of the messages or in the delivery process. The system is composed by a message repository and a set of observer objects.

Each of these *user server objects* is dedicated to a guard that is logged in the system, it is responsible for retrieving the messages from the server,

filtering them according to the mission and profile of the guard, and send them in the proper format to the corresponding device.

For instance, a guard on patrol duty carrying a cell phone requires that the messages are transformed into text and that the events not concerning its route are ignored. On the other hand, a security guard in the office can monitor several video streams on a desktop computer and accepts a larger set of message types. Also, depending on the rank of the guard different sets of messages can also be selected. Furthermore, the system should allow extensions to new user profiles, devices or connections (e.g. mobile phones with MMSs can receive images or video clips.).

To cope with this multiplicity of scenarios we need a flexible and yet safe way of building the user server objects knowing that a single implementation covering all cases is neither safe or easily maintained. We use the capability of dynamically loading and creating components to build type safe components adequate for each situation.

The architecture of the whole system is depicted in Figure 3.4, it comprises components like an user manager that relates users to profiles, an element loader that loads components dynamically into the system in a type safe way, a session manager that holds the current state of the system in terms of users logged in, a configurator that produces components configured according to information about each session, and finally the message server that stores messages posted by the sensors and is observed by the user server objects.

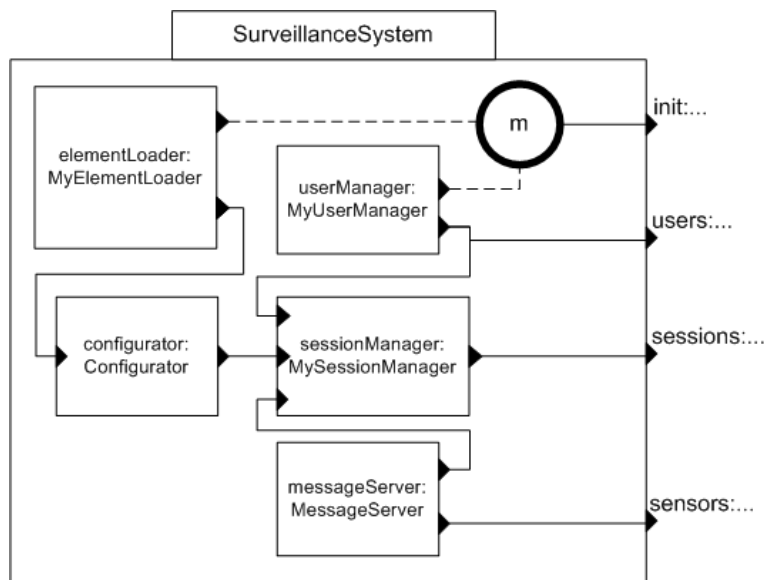


Figure 3.4: The surveillance system architecture.

This example is specified and implemented in the configurator example distributed with the compiler. For now we will focus on the reuse of the observer pattern in the message server and on the dynamic composition of user server components. Next, we will describe some other components and finally we will show the composition of the system as a whole.

Reusing the observer pattern To implement the `MessageServer` component, clearly an instance of the observer pattern, we will reuse the components designed in the example in section 3.2. The specification of the message server is given by the following interfaces:

```
port interface ISensorsManager {
    void addMessage(IMessage);
}

port interface IMessageServer extends ISensorsManager {
    int getLastId();
    IEnumeration<IMessage> getMessagesFrom(int);
}

component interface TMessageServer {
    provides ISubject<IMessageServer> subject;
    provides IMessageServer messages;
}
```

Messages are sequentially numbered and can be retrieved via a port typed with `IEnumeration<IMessage>`. The implementation of `MessageServer` uses a native component `MyVector` to implement the message repository and the component `CSubject` to implement the subject role. We use a method block to intercept the calls made to the port `messages` and notifies the observers whenever there is a new message.

```
component MessageServer {
    provides ISubject<IMessageServer> subject;
    provides IMessageServer messages;

    intro TSubject<IMessageServer> csubject =
        CSubject<IMessageServer>;
    intro TVector<IMessage> v = MyVector<IMessage>;

    declare m {
```

```

void addMessage(IMessage msg) {
    v.vector.add(msg);
    csubject.subject.notifyObservers();
}
int getLastId() {
    return v.vector.size();
}
IEnumeration<IMessage> getMessagesFrom(int f) {
    int i = f;
    IVector<IMessage> subset =
        (new MyVector<IMessage>).vector;
    while( i < v.vector.size() ) {
        subset.add(v.vector.get(i));
        i = i+1;
    }
    return subset.elements();
}
}
plug csubject.subject into subject;
plug m into csubject.obj;
plug m into messages;
}

```

Note that unlike dynamically typed languages like Java, the vector and enumeration objects are parametric and therefore provide type safe access to their elements.

In the composition of the `SurveillanceSystem` component (Figure 3.4), port `messages` is connected to a provided port with a smaller interface, `ISensorsManager`, which allows only the posting of new messages. The whole `IMessageServer` interface is only available inside the component and in this case it is only used by the observers that receive it through the `update` method. In the example, `MessageServer` can be found in the file `MessageServer.cj`.

Configurator Besides the message server, the system uses a component called `Configurator` that produces components capable of serving the connection between the server and a particular guard (filtering and formatting the messages according to the situation). The type of the returned components is `TUserServer` which is based on the interfaces `IUserServer` and a particular type instance of `IObserver`.

```
port interface IUserServer extends IObserver<IMessageServer> {
```

```

    void login(IConnection);
}

component interface TUserServer {
    provides IUserServer msgProvider;
}

```

The purpose of the `Configurator` component is to dynamically produce components whose behavior varies according to the input information. In particular a user server object as described above needs to use different elements in each situation: a filter and a message composer. The role of the filter is to select which messages are actually sent to its user, and the role of the composer is to format the messages in a way accepted by the user's device. Finally, we could represent different policies of message sending as caching, retrying, or agglomerating unimportant messages by using different base architectures. The component `SimpleHandler` below, specified by `TServerArchitecture`, provides a factory service that is basically a method that accepts two components and returns a new component that composes them in an predefined way (Figure 3.5).

```

port interface IServerArchitecture {
    TUserServer makeServer(TFilter<IMessage>, TComposer);
}

component interface TServerArchitecture {
    provides IServerArchitecture factory;
}

component SimpleHandler {
    provides IServerArchitecture factory;
    declare m {
        TUserServer makeServer(TFilter<IMessage> myFilter,
                               TComposer myComposer) {
            return compose {
                provides IUserServer msgProvider;

                intro TFilter<IMessage> msgFilter = myFilter;
                intro TComposer msgComposer = myComposer;

                declare serviceMethods {
                    IConnection con;

```

```

        int lastId;
        void update(IMessageServer server) {...}
        void login(IConnection myCon) {...}
    }

    plug serviceMethods into msgProvider;
};
}
}
plug m into factory;
}

```

where the methods in the method block `serviceMethods` are implemented in the following way:

```

...
void update(IMessageServer server) {
    IEnumeration<IMessage> enum = server.getMessagesFrom(lastId);
    IMessage msg;
    while( enum.hasMore() ) {
        msg = enum.next();
        if( msgFilter.filter.isGood(msg) ) {
            con.send(msgComposer.composer.createMessage(msg));
        }
    }
    lastId = server.getLastId();
}
void login(IConnection myCon) {
    con = myCon;
}
...

```

Note that, in the figure, the unknown components are shaded, their role in the architecture is already defined but its concrete implementation is not.

An instance of `SimpleHandler` can be called upon to create a new component based on two existent components.

```

component Configurator {
    provides IConfigurator config;
    requires IElementLoader loader;
}

```

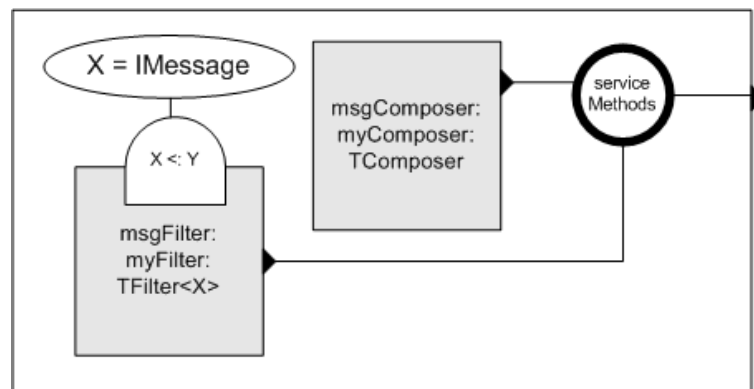


Figure 3.5: The user server architecture

```

declare m {
    TUserServer selectServer(string arch,
                             string comp,
                             string filt) {

        TServerArchitecture architecture =
            loader.getArchitecture(arch);
        TComposer composer = loader.getComposer(comp);
        TFilter<IMessage> filter = loader.getFilter(filt);

        return (new architecture).factory.makeServer(filter,
                                                    composer);
    }
}
plug m into config;
}

```

As a consequence of a login action the session manager obtains the profile of the user and calls the method `selectServer` from the `config` port of the configurator component giving it three names. The configurator makes use of the `loader` service and loads the component values corresponding to the architecture, the filter and the composer. The architecture, after being loaded is instantiated and its factory method called taking as arguments the other two component values.

The configured components here are then passed to the session manager who instantiates and initializes them to attend the requests of a particular user. The definition of the `Configurator` component can be found in the source file `SurveillanceSystem.cj` and the specification of the elements can be

found in the source file `ElementLoaderSpec.cj`. Sample architectures, filters and composers can be found in the source file `UserServerElements.cj`.

The session manager In the architecture of Figure 3.4 it is the inner component `sessionManager` (which is then implemented by the component `MySessionManager`) that receives the login requests, searches for an adequate profile, creates and instantiates the user server component and finally registers the resulting user server object as an observer to the messages in the `MessageServer` subcomponent. In spite of being possible its implementation in `ComponentJ`, it is implemented as a native component in Java. Its source can be found in the `MySessionManager.java` source file, and its specification (set of `ComponentJ` type declarations) can be found in the file `SessionManagerTypes.cj`.

Composing the elements together The other elements necessary to compose the whole system can be found in the example. The reason for not describing their construction here concerns only simplicity and brevity of presentation. The whole system is implemented in the component `SurveillanceSystem` below as depicted in Figure 3.4.

```
component SurveillanceSystem {
    provides SSIInit init;
    provides IUserManager users;
    provides ISessionManager sessions;
    provides ISensorsManager sensors;

    intro TElementLoader elementLoader = MyElementLoader;
    intro TUserManager userManager = MyUserManager;
    intro TConfigurator configurator = Configurator;
    intro TSessionManager sessionManager = MySessionManager;
    intro TMessageServer messageServer = MessageServer;

    plug elementLoader.loader into configurator.loader;
    plug configurator.config into sessionManager.config;
    plug userManager.users into users;
    plug userManager.users into sessionManager.users;
    plug messageServer.subject into sessionManager.subject;
    plug sessionManager.sessions into sessions;
    plug messageServer.messages into sensors;

    declare m {
```

```

void init(string usersURL, string elementsURL) {
    userManager.config.init(usersURL);
    elementLoader.config.init(elementsURL);
}
void save() {
    userManager.config.save();
}
}
plug m into init;
}

```

The resulting component is then used in the Java class `Main` that can be found in the source file `Main.java` where it is instantiated and used by some other classes that implement a simple user interface. A full description of the compilation procedure is included in the `README` file in the corresponding directory.

3.4 Native components

Although `ComponentJ` is a programming language with a high level of expressiveness, its real aim is to provide a glue language for components programmed in Java. So, it is of the most importance that users of the language learn the pattern that allows both the programming of native components and the adaptation of existing functionality.

In order to be integrated in a composition a native component must comply with a static specification. This is achieved by means of a *skeleton* class which is produced by the `ComponentJ` compiler from a component interface declaration. The compiler option `--skeletons` (or `-skels` for short) produces two classes for each component interface in the source file it is compiling. A base skeleton and a generated skeleton. The first class will be extended by the class implementing the native component, it contains all the information and functionality that the `ComponentJ` compiler needs to integrate it in a composition. The second one is a class emitted to be used as an example, programmers can edit the source file and modify it to program the native component.

We will now describe in more detail the implementation of two native components, `MySystemPrint` and `MyVector`.

3.4.1 MySystemPrint component

The component `MySystemPrint` is used often in the examples distributed with the compiler to print messages in the console. It is specified by the type declarations contained in the `ComponentJ` source file `TPrint.cj` and implemented in the Java source file `MySystemPrint.java`.

The specification of the component type `TPrint` below results in an abstract class called `TPrintBaseSkeleton` which is extended by the class `MySystemPrint`. The compiler command is `cjc --skeletons TPrint.cj`.

```
port interface IPrint {
    void print(string);
}

component interface TPrint {
    provides IPrint p;
}
```

Next, an inner class called `Instance` extends the an inner abstract class of the base skeleton, `InstanceSkeleton`. `Instance` must provide the functionality of the ports through assignments to the port providers by using the methods `set_X_provider` (where `X` is the port name) with new port provider objects as arguments. The method `createPartialInstance` must be implemented returning a component instance, and the method that represents the singleton, `asValue`, must also return a unique instance of the wrapping class.

```
import ComponentJ.*;

public class MySystemPrint extends TPrintBaseSkeleton {

    class Instance extends InstanceSkeleton implements IPrint {
        Instance() {
            set_p_provider(new PortProvider(this));
        }

        public void print(String s) {
            System.out.println(s);
        }
    }

    public IObject createPartialInstance() {
        return new Instance();
    }
}
```

```

    }

    static IComponent value = new MySystemPrint();
    public static IComponent asValue() {
        return value;
    }
}

```

Another way of implementing is to isolate the provided functionality in one separate class. This way is probably the easiest when adapting existing functionality.

```

import ComponentJ.*;

public class MySystemPrint extends TPrintBaseSkeleton {

    class Instance extends InstanceSkeleton {
        Instance() {
            set_p_provider(new PortProvider(new MyImplementation()));
        }
    }

    private class MyImplementation implements IPrint {
        public void print(String s) {
            System.out.println(s);
        }
    }

    public IObject createPartialInstance() {
        return new Instance();
    }

    static IComponent value = new MySystemPrint();
    public static IComponent asValue() {
        return value;
    }
}

```

The compilation of one of this classes against the interfaces and classes generated by the compiler completes the implementation of the native component `MySystemPrint`.

3.4.2 MyVector generic component

We also provide the implementation of a parametric vector which is based directly on the vector class from the Java framework. We start by defining some parametric port interfaces and a component interface.

```
port interface IEnumeration <X> {
    boolean hasMore();
    X next();
}
```

```
port interface IVector<X> {
    void add(X);
    void set(int,X);
    X get(int);
    IEnumeration<X> elements();
    int size();
}
```

```
component interface TVector<X> {
    provides IVector<X> vector;
}
```

The port interfaces allow for the definition of a parameterized vector that ensures the type of its elements within a `ComponentJ` program. However, in the native component implementation, the type variables are abstracted to `Object` and in the parameterized interfaces the type parameters are ignored. Besides following the already described programming pattern, the Java class `MyVector` also has an inner class to implement the `IEnumeration` interface.

```
import ComponentJ.*;

import java.util.*;

public class MyVector extends TVectorBaseSkeleton {
    class Instance extends InstanceSkeleton implements IVector {

        Vector v;

        Instance() {
            v = new Vector();
            set_vector_provider(new PortProvider(this));
        }
    }
}
```

```
    }

    public IEnumeration elements() {
        return new MyEnumeration(v.elements());
    }
    public Object get(int i) {
        return v.get(i);
    }
    public void add(Object o) {
        v.add(o);
    }
    public void set(int i, Object o) {
        v.setElementAt(o,i);
    }
    public int size() {
        return v.size();
    }
}

public class MyEnumeration implements IEnumeration {
    Enumeration base;
    MyEnumeration( Enumeration base ) {
        this.base = base;
    }

    public Object next() {
        return base.nextElement();
    }

    public boolean hasMore() {
        return base.hasMoreElements();
    }
}

public IObject createPartialInstance() {
    return new Instance();
}
static IComponent value = new MyVector();
static public IComponent asValue() {
    return value;
}
```

```
}

```

This class is generic and allows for a typed use of vectors in `ComponentJ` like in the component below:

```
component VectorTest {
  provides ITest test;
  intro TPrint printer = MySystemPrint;
  intro TVector<string> vs = MyVector<string>;
  intro TVector<int> vi = MyVector<int>;
  declare m {
    void doIt() {
      vs.vector.add("One");
      vs.vector.add("Two");
      vs.vector.add("Three");
      vi.vector.add(1);
      vi.vector.add(2);
      vi.vector.add(3);
      printer.p.print(vs.vector.get(2));
      if(vi.vector.get(2) == 3 )
        printer.p.print("Yes");
      string s = "";
      IEnumeration<string> e = vs.vector.elements();
      while(e.hasMore()) {
        s = s+e.next()+" ";
      }
      printer.p.print(s);
    }
  }
  plug m into test;
}
```

In the distribution of the compiler there is also a generic `Hashtable` that is implemented in the same way as this.

Chapter 4

ComponentJ Compiler

`cjc` is prototype of a command line compiler that works like a translator from ComponentJ to Java and relies on a standard Java compiler to perform the final step of producing ByteCode for the JVM.

4.1 Setting up the environment

In order to setup the environment variables for the compiler there are two scripts in the compiler directory (`env.sh` in Linux and `env.bat` in Windows). They basically set the environment variable `CJHOME` with the path of the compiler distribution. This allows the executable to find the jar files for the compiler and framework classes.

4.2 Compiler executable

The usage of the compiler is the following:

```
cjc [options] filename
```

And the options for the compiler executable are:

-skels or *--skeletons*

This options makes the compiler generate a skeleton class for each component type it encounters.

-stubs

With this option all compiled component types produce client stub classes which ensure the structural equivalence of component types in the Java world. This option is on by default because the stub class also identifies the component type in a separate compilation scenario.

-v or *--verbose*

Displays a lot of messages to the standard output. Otherwise these messages are stored in a `log.out` file.

-ast

Prints the abstract syntax tree after reading the source file.

-gst

Prints the generated syntax tree of the Java translation of the `ComponentJ` source file.

--stacktrace

ThisX options makes the compiler to produce a stack trace if it halts because of an unexpected problem.

4.3 ComponentJ framework

The compiler makes use of a small set of classes and interfaces located in a Java package called `ComponentJ`. Every time a `cjc` source file is compiled the framework is used to support the implementation of `ComponentJ` abstractions (Components, Instances, Ports).

4.4 Packaging

In the present version of the compiler all the Java code produced by the compiler and resulting `.class` files are placed in a `tmp` directory below the current directory.

Ideally, a component should be packaged into a jar file with all the classes and interfaces. When in use a component should be registered in some repository (a registry) that makes it available in the system. This can be done either by unpacking all the jar files into a directory present in the classpath or by placing all the jar files in the classpath.

Bibliography

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
- [2] João Costa Seco. Adding type safety to component-oriented programming. In *First FMOODS PhD Student Workshop*, 2002.
- [3] João Costa Seco and Luís Caires. A basic model of typed components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2000.
- [4] João Costa Seco and Luís Caires. Parametrically typed components. In *WCOP'2000, Fifth Workshop on Component Oriented Programming*, 2000.
- [5] João Costa Seco. Type safe composition in .NET. In *First Microsoft Research Summer Workshop*, Cambridge, 2002.
- [6] João Costa Seco and Luís Caires. The parametric component calculus. Technical Report UNL-DI-7-2002, New University of Lisbon, 2002.