ENGINEERING A COMPONENT LANGUAGE: COMPJAVA

Hans Albrecht Schmid, Marco Pfeifer
University of Applied Sciences Konstanz, Brauneggerstr. 55, D - 78462 Konstanz, Germany

Keywords: Components, Component language, Component composition, Component fragment, Connections.

Abstract: After first great enthusiasm about the new generation of component languages like ArchJava, ComponentJ and ACOEL, a closer inspection and use of these languages identified together with their strong points some smaller, but disturbing drawbacks. These might impede a wider acceptance of component languages, which would be harmful since the integration of architecture description with a programming language increases the emphasis on, and consequently the quality of application architecture. Therefore, we took an engineering approach to the construction of a new Java-based component language without these drawbacks. That means, we derived general component language requirements; designed a first language version meeting the requirements and developed a compiler; used it in several projects; and re-iterated three times through the same cycle with improved language versions. The result, called CompJava, which should be fairly stable by now, is presented in the paper.

1 INTRODUCTION

The new generation of component languages, like ArchJava (Aldrich, May 2002) (Aldrich, 2002), ComponentJ (Seco, 2000), ACOEL (Sreedhar, 2002), and to a smaller degree, KOALA (van Ommering, 2000) (van Ommering, 2002) made enthusiastic about the new way of program construction without reference handling. These languages integrate architecture description with a programming language. Thus, they push the more abstract architecture-description-language (ADL) based approach (see ADL classification framework (Medvidovic, 2000), (Medvidovic, 1999)) forward towards a direct use. Our experience confirms that this increases the emphasis on, and consequently the quality of application architecture.

However, a closer inspection and use of component languages identified together with their strong points some small, but disturbing drawbacks.

For example, ArchJava components behave like classes with regard to some aspects. A component class generates implicitly its type, and inheritance is defined primarily as implementation inheritance among components. Further, though a component is not a class, it may inherit implementation from a class. ArchJava re-defines constructs for concepts, like interfaces, which it shares with Java. ACOEL shows no symmetry with regard to the attachment of code to provided and required interfaces. More drawbacks and details are given in section 2.

It seems that these drawbacks might impede a larger acceptance and broader use of component languages. Therefore, we designed a new component language that does not have these drawbacks, following a sound engineering approach. We derived a list of component language requirements from the identified drawbacks. We constructed a component language that covers the requirements (the first version being available fall 2003). Then, we used the language in projects, and had three iterations with improved language definitions. Now, the language will be quite stable.

Section 3 gives an overview about distinguishing structuring principles of CompJava, and section 4 introduces its type concept. Section 5 shows how components are composed in a structured way from component fragments, and section 6 shows how they are composed from subcomponents. Section 7 presents dynamic architectures using a Web server example.

2 LANGUAGE REQUIREMENTS

This section describes drawbacks identified in component languages and derives specific requirements from them. These component language requirements complement general, but unlisted requirements, defined by a kind of intersection of the features of existing languages.

Albrecht Schmid H. and Pfeifer M. (2006).
ENGINEERING A COMPONENT LANGUAGE: COMPJAVA.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 98-106
DOI: 10.5220/0001320100980106

Embedded OO-Programming Language
A component language embeds a programming language and uses its constructs to implement components. ArchJava which embeds Java has ports with both provided and required interfaces. It defines the interfaces of a port either by listing, after the keyword provides or requires, operation specifications, or by listing method implementations. But you cannot define the interfaces of a port using Java interfaces. Thus, the identical concept "interface" is described by different constructs in the component language and the OO-language, which is certainly a drawback.

On the other hand, ArchJava allows to derive components from classes, like the worker component from the class Thread (Aldrich, May 2002). But how can a component, which is not a class, but a first-class citizen of its own, inherit implementation from a class?

Therefore, **requirement 1** is: a component language should not reinvent constructs for concepts it shares with its programming language. On the other hand, it should not intermingle differing concepts in the component language and programming language.

Component Inheritance
ArchJava transfers the type concept of class-based languages directly to components. It defines a component type implicitly as the type that is generated by a component class, and it defines inheritance in such a way that a derived component inherits from a base component both the component type and its implementation.

This has two drawbacks. A definition of a component type that is independent from the implementation is required to define e.g. a product line architecture or a component framework. A product line architecture defines product component types which are implemented by different product components. Similarly, a component framework defines a set of collaborating component types which are implemented by different components. Second, a component should not inherit the implementation from another component, but should be composed with the other component in order to reuse its functionality. Therefore, **requirement 2** is that the definition of component types and inheritance among them should be provided, but implementation inheritance among components should be disallowed.

Component Encapsulation
ArchJava allows that a parent component invokes directly internal methods of a subcomponent which are not defined by a provided port. This breaks the encapsulation of the subcomponent. Further, a graceful evolution is inhibited since it is not possible that a sibling subcomponent invokes these methods instead of the parent component at a later point of the evolution. On the other hand, ACOEL allows that a parent component exposes a reference to a subcomponent in a port. When it passes that component reference to a sibling component, ports of the sibling component may be connected to ports of the subcomponent. That means a component may be at the same time a subcomponent of two different components. This breaks a sound architectural structure.

Requirement 3 is that a component should be completely encapsulated, i.e. it should collaborate only via its ports with external code. As a consequence, a subcomponent of a component must not collaborate with other components outside of its parent component. Therefore, the passing of component or port references should be restricted or prohibited.

Interface Symmetry
ArchJava has a complete symmetry among provided and required interfaces with regard to their definition and their use, since a port may comprise both of them. ACOEL (Sreedhar, 2002) has a symmetry with regard to their definition, but not with regard to their use. A mix-in allows to put a filter between a provided port and the implementing class. But it does not allow to put a filter between the implementing class and a required port.

Requirement 4 is that the definition and the handling of provided and required ports should be symmetrical.

Ports and Connectors
An ArchJava port may combine a provided and a required interface, like:

port *port1* **provides** *m1, m2* **requires** *m3, m4*;

As usual, a port with a required interface I_1 may be connected to a port with a provided interface I_2 when I_2 is a subtype of I_1. But an ArchJava connector may fork the calls from a required interface I_1 to several provided interfaces like I_2 and I_3 if each is a supertype of I_1, and their union is a subtype of I_1, and their intersection with regard to I_1 is empty. For example, with *port2* and *port3*:

port *port2* **provides** *m3, m6* **requires** *m1, m5*;
port *port3* **provides** *m4, m5, m6* **requires** *m2, m3*;

ArchJava allows to connect *port1*, *port2*, and *port3* by a connect statement. If *port1* would require additionally *m6* the connection would not be correct and rejected. This is not easy to check and understand for a programmer; it might be considered as a new kind of spaghetti problem (without dining philosophers). Though it is easy for a compiler to check what happens, we should disallow it.

Requirement 5 is that the definition of ports and connectors should be made in a way that is easily understandable to a programmer.

Collaboration of Subcomponent Ports with Code
ArchJava defines private ports in order to connect component code with a port of a subcomponent. However, a private port is a contradiction in itself since the ports of a component define its interfaces to the outside, i.e. the points of collaboration with external code: So what is the semantics of a private port? It is even more confusing that ArchJava allows to connect two private ports; what does that mean? Our conclusion is that the concept of private ports is questionable. **Requirement 6** is that an adequate construct should connect component code with a port of a subcomponent.

Implementation Isomorphy with OO-Based Approach
ArchJava generates one component class which lists the provided methods of all public and private ports of the component. The generated code does not group together the methods which implement the operations of the same port. Similarly, the required operation of all ports are always invoked from that list of methods. There is no way to group the methods that invoke the operations of the same port. This is in contrast to the usual OO-based implementation of a component where the provided methods of each port are implemented by a different class, and the required methods of each port are usually invoked by methods from different classes.

Therefore, **requirement 7** is that the code generated from a component should have at least some isomorpy with corresponding code written in class-based OO-languages.

Implementation Efficiency
The efficiency of the code generated from a component language may not be a primary concern when large architectural components with powerful operations are realized. But in many cases, the efficiency of a frequently performed operation invocation matters. Consider e.g. a scanner, used e.g. as a subcomponent of a compiler, which is certainly not a lightweight component. If it fetches the next character from a source file over a required interface with a getCharacter-operation (compare section 6), the efficiency of that frequently performed operation invocation has a strong influence on the scanner overall performance.

Requirement 8 is that the code generated from a component language should have about the same efficiency for basic constructs, like e.g. operation invocation over connected ports, as an equivalent (but not tricky) class-based implementation. We state that requirement due to its importance for a wide acceptance of component languages, though we cannot cover it in this paper for space reasons.

3 COMPJAVA OVERVIEW

Distinguishing features of CompJava are, besides the definition of component types and component type inheritance, its structuring facilities for component construction. CompJava allows not only, like the new generation of component languages, to compose components from subcomponents in a structured way. It allows to compose them also in the same way from code building blocks, or from a combination of subcomponents and filters formed by code building blocks.

CompJava has code building blocks called component fragments. A component fragment might be considered as a simply structured light-weight component without ports: it provides exactly one interface, and it requires usually one interface. The provided interface of a component fragment is explicitly indicated in the form of a Java interface; the required interfaces of a component fragment are implicitly given by the visible ports and plugs of the enclosing component. There are three implementation variants of a component fragment: anonymous class, inner class and method block; from which a user may select the suitable one.

CompJava introduces plugs which are used mainly for connecting component fragments with subcomponent ports.

Ports of subcomponents are connected with the connect-statement to other ports or plugs. Component fragments are attached to the inside of the component ports or to plugs with an attach-statement. Thus, CompJava allows to compose

1. a low-level component from component fragments, as illustrated by Figure 1 a)
2. a high-level component from subcomponents, as illustrated by Figure 1 b)
3. a medium/high-level component from a combination of subcomponents and component fragments that are used as filters, as illustrated by Figure 1 c)

in a clear, clean and structured way.

For a graphical depiction of the composition of a component, we have enriched UML 2 component diagrams with component fragments and plugs (depicted by a diamond). A component fragment is represented according to the selected implementation as an anonymous class, an inner class or as a method block (depicted like an anonymous class without class head).

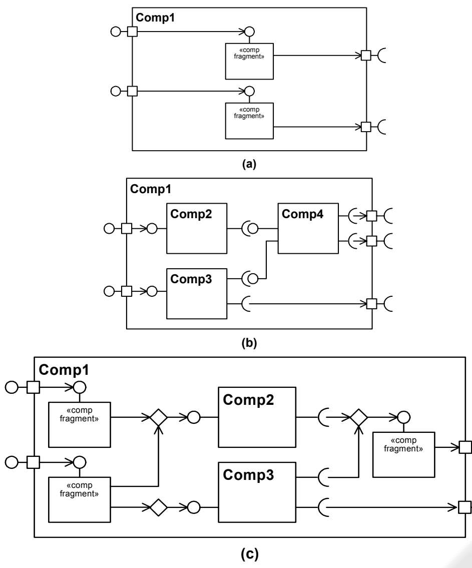


Figure 1: Composition of a component from component fragments (a), from subcomponents (b), and from a combination of them (c).

The first version of the CompJava compiler has been available since winter 2003/2004, three more versions followed. The new version to be available in fall 2006 will be integrated in Eclipse. The CompJava Designer is a graphical design tool that allows to draw enriched CompJava component diagrams and to generate component code skeletons. It is an Eclipse-plugin and in prototype stage, to be available spring 2007.

The following sections introduce the CompJava language and shows that their constructs satisfy the requirements. We use a compiler as a running example. The compiler component is composed from a scanner, parser and other subcomponents.

4 COMPONENT TYPES

Let us consider first the scanner component. We define the provided interface of the scanner as a Java interface. It includes all scanner-related responsibilities, like setting the file name of the source file to be processed, and fetching the next token from it.

```
interface ScannerIF {
  Token getNext();
  void setSource( String sourceName);
}
```

Since the *ScannerIF* interface includes all source file processing related responsibilities, the component type *ScannerType* is defined with a single provided port.

```
component type Scanner1Type {
    port in provides ScannerIF;
}
```

A component type defines all interfaces of a component. That means components are completely encapsulated: all methods in a component, except for the main method, can be invoked from outside only via provided ports, and all methods can invoke an outside method only via required ports.

A port has either a provided, a required or an event interface. A port declaration gives the port name and after the corresponding keyword the associated interface. An event port is similar to a required port, but its operations must not have results, and several provided ports of event listeners may be connected to it. As we show in section 7, a component type may also define port arrays or port vectors.

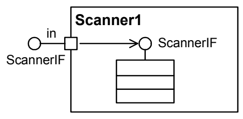
A component type may extend another component type, like an interface may extend another one. It inherits all ports, and it may extend the interface of inherited provided ports or may add provided ports.

5 LOW-LEVEL COMPONENTS

This section shows how low-level components are composed from component fragments.

Implementing Provided Ports

A component has a component type (indicated by the ofType-clause). It implements all the provided ports, and may invoke operations from the required ports specified by its component type. In the Scanner1 component (see Figure 2), an attach-statement attaches the inside of the provided port *in* to a component fragment, an anonymous class implementing the *ScannerIF* interface.



```
component Scanner1 ofType Scanner1Type{
  //port in provides ScannerIF;
  attach This.in to new ScannerIF {
    private File sourceFile;
    void setSource(String name){//open sourceFile}
    char getChar(){//next char from sourceFile}
    Token getNext(){
      Token current = new Token();
      char c = getChar();
      while ( c != separator ){
        current.append( c );
        c = getChar(); }
      return current; }
  };
}
```

Figure 2: The *Scanner1* component with port *in* providing the *ScannerIF* implemented by a anonymous class.

An attach-statement may be used to attach the inside of a provided port to a component fragment that implements an interface I. The condition is that I extends (including equals) the port interface; it is checked at compile time. A component fragment may be a Java construct: an instance of an anonymous class, as shown, or an instance of an inner class. The inside of a port is indicated by the keyword *This*, which stands for the component instance, followed by the port name. The declaration of inner and anonymous classes follows the Java standard; the only difference is that they are used inside of a component instead of a class.

When a component, like *Scanner1*, is quite small and not composed from other components, it might be a disadvantage that its implementation generates two object instances: one of the application-specific component fragment and another one of the component class. Therefore, CompJava allows also that a component fragment is formed by a method block. A method block is a sequence of methods that implement a given interface (see Figure 3). A method block is not a Java construct, but an analogon to a Java block, which is a sequence of statements. When different provided ports are each attached to a method block, there is the restriction that their interfaces must have an empty intersection.

Consequently, CompJava provides component fragments which include method blocks, inner classes or anonymous classes, in order to structure the implementation of a component.

Accessing Required Ports

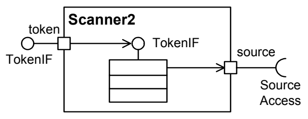
The Scanner mixes up two different concerns, scanning the program character stream, and handling of the source file to be parsed. Similarly, the *ScannerIF* interface mixes up two different concerns, accessing the tokens which the scanner creates, and determining the source file to be parsed. We should separate the different concerns, scanning and source file handling. To this purpose, we define two interfaces, *TokenIF* and *SourceAccess*:

```
interface TokenIF {
  Token getNext();
}
interface SourceAccess {
  char getChar();
}
```

The new scanner component does not include the source file handling but fetches the source file characters via a required interface. We define the component type *Scanner2Type* with a provided interface *TokenIF* and a required interface *SourceAccess*:

```
component type Scanner2Type {
  port token provides TokenIF;
  port source requires SourceAccess;
}
```

The *Scanner2* component attaches the *token* port to a component fragment, a method block. It implements the *TokenIF* and scans the source file in order to determine the next token. When it needs the next character from the source file, it simply invokes the *getChar*-operation defined in the *SourceAccess* interface via the inside of the required port *source*.



```
component Scanner2 ofType Scanner2Type  {
  //port token provides TokenIF;
  //port source requires SourceAccess;
  attach This.token to TokenIF {
    Token getNext(){
      Token current = new Token();
      char c = This.source.getChar();
      while ( c != separator ){
        current.append( c );
        c = This.source.getChar(); }
      return current; }
  };
}
```

Figure 3: The *Scanner2* component with port *token* providing the *TokenIF* implemented by a method block, and port *source* requiring the *SourceAccess* interface.

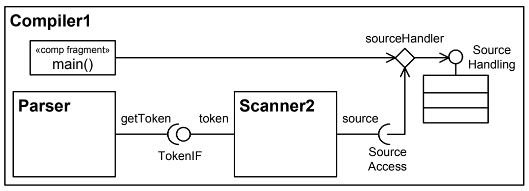
6 COMPONENT COMPOSITION

A compiler is a top-level component that is composed from a scanner, a parser etc. For that reason, we declare its type without any ports. The type of the parser defines a required interface *TokenIF*, and other ones which we do not consider.

```
component type CompilerType {}
component type ParserType {
     port ...;
     port getToken requires TokenIF;
}
```

Subcomponents

A component may be composed from subcomponents. E.g. the *Compiler1* component (see Figure 4) is composed from a scanner, a parser, and other subcomponents like a code-generator which we disregard.



```
interface SourceFile {
  void setSource( String sourceName);
}
interface SourceHandling extends
                   SourceFile, SourceAccess { }
component Compiler1 ofType CompilerType {
  ParserType myParser = new Parser();
  Scanner2Type myScanner = new Scanner2();
  connect myParser.getToken to myScanner.token;
  plug<SourceHandling> sourceHandler;
  connect myScanner.source to sourceHandler;
  attach sourceHandler to new SourceHandling{
    private File sourceFile;
    void setSource(String name){//open sourceFile}
    char getChar(){
      //read next char from sourceFile
    }
  };
  public void main( String[] args )
  { String sourceName = args[1];
    new Compiler1();
    This.sourceHandler.setSource(sourceName);
    //start parser via a plug and port not shown
  }
}
```

Figure 4: Component *Compiler1* composed from subcomponents *Parser* and *Scanner2* and a component fragment implementing the interface *SourceHandling.*

A component may contain subcomponent declarations and connect-statements that are processed with the initialization of the component.

A subcomponent declaration declares a subcomponent variable, like *myParser* and *myScanner*, of a component type; it may assign to it an instance of a matching component created with the *new* operator and the component constructor, like a *Parser* resp. a *Scanner2* instance.

A connect-statement connects a required port of a subcomponent (instance), like *getToken* of *Parser*, to a provided port of a subcomponent (instance), like *token* of *Scanner2*, as Figure 4 shows. A constraint checked by the compiler is that a required port can be connected to only one provided port; but many required ports may be connected to the same provided port. An event port may be connected to many provided ports. The compilation of a connect-statement includes port-matching, i.e. checking if the provided port interface extends (incl. equals) the required port interface. We may use a connect-statement also to connect a port of a subcomponent directly with the inside of a matching port of the (parent) component.

Connecting Subcomponent Ports with Plugs

The *Compiler1* component contains a component fragment, an anonymous class implementing the interface *SourceHandling*, which the source port of the *Scanner2* should invoke. However, a connect-statement does not allow to connect a subcomponent port with a component fragment. Therefore, we introduce plugs which replace private ports of ArchJava.

A plug is a generic construct that exceeds the generic possibilities provided by parametric interfaces or classes. The generic expression “plug<interface>” generates a plug of the interface type. It might be considered as a variable on which only a very limited set of operations may be executed: it may be used in connect- and attach-statements, or it may be used in a component fragment to invoke an operation defined in the plug interface.

The Compiler component (see Figure 4) declares a plug of the interface type *SourceHandling* named *sourceHandler*. The plug is used to pass operation invocations from the required port of the scanner subcomponent to a component fragment of the compiler component, which does all handling of the source file.

A connect-statement connects the required port *source* of the scanner with that plug, matching at compile time whether the plug interface extends the required port interface. The main method, which gets the filename of the source file passed as a parameter, invokes the *setSource*-operation via the same plug.

An attach-statement may attach a plug to a component fragment, as shown in Figure 4. It checks at compile time whether the interface of the component fragment extends the plug interface. The constraint is that the same plug may appear only once on the left-hand side of an attach- or connect-statement, but several times on their right-hand side and/or be used for operation invocations.

Factoring Out SourceHandling

Suppose that we want to reuse the anonymous source handling class with the interface *SourceHandling* shown in Figure 4. Then we should factor it out and transform it into a separate source file processing component with the component type *SourceType*.

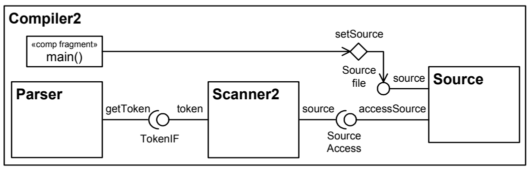
```
component type SourceType {
  port source provides Sourcefile;
  port accessSource provides
                          SourceAccess;
}
```

The component Source contains a *SourceHandling* component fragment that is identical to the component fragment used by the *Compiler1* component (see Figure 4). Since we want to attach both provided ports to the same component fragment, we declare the plug *sourceHandler* of type *SourceHandling*. It is attached to the component fragment with an attach-statement. The inside of each provided port is attached to the plug with each an attach-statement.

```
component Source ofType SourceType {
  plug<SourceHandling> sourceHandler;
  attach This.source to This.sourceHandler;
  attach This.accessSource to This.sourceHandler;
  attach This.sourceHandler to new SourceHandling{
    private File sourceFile;
    void setSource(String name){//open sourceFile}
    char getChar(){
      //read next char from sourceFile
    }
  };
}
```

Figure 5: Component *Source* with the provided ports *source* and *accessSource* attached to plug *sourceHandler* attached to an anonymous class as component fragment.

The component *Compiler2* (see Figure 6) is identical to *Compiler1*, except for replacing the *SourceHanding* component fragment by the *Source* component. It connects the port *source* of *Scanner2* with a connect-statement to the *accessSource* port of *Source*. The plug *setSource* is declared and connected to the *source* port of the *Source* component with the objective that the *main* method may invoke via that plug the *setSource*-operation of the *source* port.



```
component Compiler2 ofType CompilerType {
  ParserType myParser = new Parser();
  Scanner2Type myScanner = new Scanner2();
  SourceType mySource = new Source();
  connect myParser.getToken to myScanner.token;
  connect myScanner.source to
                          mySource.accessSource;
  plug<Sourcefile> setSource;
  connect This.setSource to mySource.source;

  public void main( String[] args )
  { String sourceName = args[1];
    new Compiler2();
    This.setSource.setSource( sourceName);
    //start parser via a plug and port not shown
  }
}
```

Figure 6: Component *Compiler2* composed from subcomponents *Parser*, *Scanner2* and *Source.*

7 DYNAMIC ARCHITECTURES

The language constructs described so far allow to construct component systems with a static architecture, i.e. a static hierarchy of collaborating component instances. Though that is sufficient for a large class of systems, there are other ones that require a dynamic creation and connection of components.

A component instance may be created dynamically in a method of a component fragment with a new-operator and component constructor in the same way as shown e.g. in Figure 4. Dynamically created components are connected at run-time with a reconnect-statement which is similar to a connect statement. A component should document explicitly all kinds of architectural interactions that are permitted between its subcomponents. To this purpose, a component uses connection patterns (as introduced by ArchJava (Aldrich, May 2002) (Aldrich, 2002)) to describe the set of connections that can be made at run-time using *reconnect*-statements.

Since in a dynamic architecture, a component may have a variable number of subcomponents of the same type, we introduce component arrays and vectors (as a parametric Vector parameterized with a component type). Since it may also be required that a connection is made from the port of a component to a variable number of sibling components, we introduce port arrays or port vectors as arrays or parameterized vectors of an interface type.

Though the primary emphasis of component and port arrays resp. vectors is on dynamic architectures, they may be of use also for static architectures with repetitive elements.

For example, consider a *WebServer* component. It has one *Router* and many *Worker* subcomponents. The *Router* receives incoming HTTP-requests and passes them through a required port of the port array *workers* to the connected *Worker* subcomponent that serves the request. The *WebServer* starts the *Router* via its provided port *start* and the plug *start*.

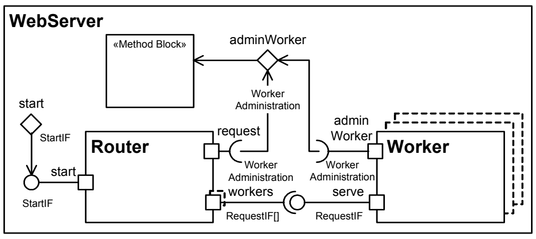
Figure 7 shows a shortened version of the *WebServer*. The running version with about three times the length of the presented version may be obtained from the authors. We present, in contrast to (Aldrich, May 2002), an optimized solution that reuses idle *Worker* instances and their connections. A *Worker* contains a *WorkerThread* class. When an *httpRequest* is invoked via the *serve* port of a *Worker*, the *WorkerThread* is (re-) started by a notify-statement and takes up work with a call of its method *handleRequest*. When it has finished the processing of an HTTP-request, it goes into a wait state.

The *WebServer* has declared an array of *Worker* components. It connects the provided *serve* port of each *Worker* instance after its creation dynamically to the matching port of the required port array *workers* of the *Router* component.

The *WebServer* performs the administration of the *Worker* instances in the method block implementing the *WorkerAdministration* interface, which is attached to the *adminWorker* plug. It has a *setIdle*-operation which is invoked by a *Worker* after having finished the processing of an HTTP-request, and similar operations. The *requestWorker*-operation checks if an idle *Worker* is available, and returns its index. Otherwise, it creates a new *Worker* instance if the maximum worker number is not yet reached. It connects dynamically a *Worker*'s *serve* port to the matching port of the *workers* port array of the *Router*, and its required *adminWorker* port to the *adminWorker* plug.

The *WebServer* has connected the required *request* port of the *Router* to the *adminWorker* plug. In that way, both the *Router* and all *Worker*'s can invoke operations of the worker administration, like *setIdle* or *requestWorker* when required.

The code of the *WebServer* component is easy to understand, in contrast to the code shown in (Aldrich, May 2002).



```
interface StartIF {
  void listen();
}
interface WorkerAdministration {
  void requestWorker();
  void setIdle( int workerId);
}
interface RequestIF {
  void httpRequest(InputStream in,
                           OutputStream out);
}

component type WebServerType { }
component type RouterType {
  port start provides StartIF;
  port request requires WorkerAdministration;
  port workers requires RequestIF[];
}
component type WorkerType {
  port serve provides RequestIF;
  port adminWorker requires WorkerAdministration;
}

component WebServer ofType WebServerType {
 final RouterType theRouter = new Router();
 WorkerType[] workers = new WorkerType[10];

 plug<StartIF> start;
 plug<WorkerAdministration> adminWorker;
 connect theRouter.request to This.adminWorker;
 connect This.start to theRouter.start;
 connect pattern RouterType.workers to
                              WorkerType.serve;
 connect pattern WorkerType.adminWorker to
                     plug<WorkerAdministration>;

 public static void main(String[] args) {
  new WebServer( ...).run();
 }
 void run() {
   This.start.listen();
 }
 attach This.adminWorker to WorkerAdministration {
  void setIdle( ...) { ...}
  int requestWorker(){
   if( no worker idle & workerID < maxWorkerID){
    workers[workerID] = new Worker(dir, workerID);
    reconnect workers[workerID].adminWorker to
                          This.adminWorker;
    reconnect theRouter.workers[workerID] to
                      workers[workerID].serve;
    return workerID; }
   //other methods...
  } };
}
component Router ofType RouterType {
 //port start provides StartIF;
 //port request requires WorkerAdministration;
 port workers = new RequestIF[10];
 attach This.start to StartIF {
  void listen() {
   ServerSocket server = new
            ServerSocket( This.request.getPort());
   while (true) {
    workerID = This.request.requestWorker();
    Socket sock = server.accept();
    This.workers[workerID].httpRequest(
    sock.getInputStream(),sock.getOutputStream());
   } }};
}
component Worker ofType WorkerType {
 //port serve provides RequestIF;
 //port adminWorker requires WorkerAdministration;
 WorkerThread myThread; //started by constructor
 BufferedReader in; // HTTP-request
 PrintWriter out; //   HTTP-response
 attach This.serve to RequestIF{
  synchronized void httpRequest(
           InputStream in, OutputStream out){
   this.in = new BufferedReader(new
                    InputStreamReader(in) );
   this.out = new PrintWriter(new BufferedWriter(
                new OutputStreamWriter(out)));
   myThread.notify();
  }
 };
 class WorkerThread extends Thread {
  //several data attributes and methods
  protected void handleRequest() {
    // open requested file and send answer ...
    out.println("HTTP/1.0 200 OK");
    // ... and file contents to Browser
  }
  public synchronized void run() {
   while (true) {
    this.wait();
    handleRequest();
    This.adminWorker.setIdle(this.workerNo);
   }  }
 } //end WorkerThread
}
```

Figure 7: Component *WebServer* composed from a worker administration component fragment together with one *Router* and a variable number of *Worker* subcomponents

8 CONCLUSIONS

CompJava, to be available for a wider use in fall 2006 via http://www-home.fh-konstanz.de/~schmidha/, composes components in a clear and simple way from two kinds of building blocks: component fragments and subcomponents. We have introduced component fragments that may be considered as very simply structured lightweight components without ports. There are three implementation variants covering different performance and reusability requirements. Component fragments allow to structure low-level components in an adequate way, and they serve as filters for medium to high level components.

These building blocks with well-defined and clear interfaces are attached/connected either directly or via plugs to themselves or to ports of the parent component.

Clean and efficient dynamic architectures are composed from dynamically instantiated and connected subcomponent instances together with component arrays and port arrays resp. vectors.

CompJava has been extended for use as a distributed component language as described in (Schmid, 2005).

REFERENCES

Aldrich, J., Chambers, C., Notkin, D., 2002, May. ArchJava: Connecting Software Architecture to Implementation. In *Procs ICSE 2002*, May 2002.

Aldrich, J., Chambers, C., Notkin, D., 2002. Architectural Reasoning in ArchJava. In *Procs ECCOP 2002*, Springer LNCS, Berlin.

Aldrich, J., Sazawal, V., Chambers, C., Notkin, D., 2003. Language Support for Connector Abstractions. In *Procs ECCOP 2003*, Springer LNCS, Berlin.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995 Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*, 1995.

Medvidovic, N., Rosenblum, D.S., Taylor, R.P., 1999, May. A Language and Environment for Architecture-Based Software Development and Evolution. In *Procs ICSE 1999*.

Medvidovic, N., Taylor, R. P., 2000. A Classification and Comparison Framework for Software Architecture Description Languages.

van Ommering, R., van der Linden, F., Kramer, J., Magee, J., 2000, March. The KOALA Component Model for Consumer Electronics Software. In *IEEE Computer*.

van Ommering, R., 2002. Building Product Populations with Software Components. In *Proc. ICSE 2002*.

Seco, J. C., Caires, L., 2000. A Basic Model of Typed Components. In *Procs. ECOOP 2000*, Springer LNCS, Springer, Berlin.

Schmid, H. A., Pfeifer, M., Schneider, T., 2005. A Middleware-Independent Model and Language for Component Distribution. In *Proc. SEM 2005*, ACM Press, New York.

Sreedhar, V. C., 2002, May. Mixin' Up Components. In *Procs ICSE 2002*.

Szyperski, C., 1997. Component Software, Beyond Object-Oriented Programming. *Addison-Wesley*