VISUAL COMPOSITION OF COMPONENT SYSTEMS

Hans Albrecht Schmid
University of Applied Sciences Konstanz, Brauneggerstr. 55, Konstanz, Germany

Christian Martin Baranowski
SEITENBAU GmbH, Robert-Gerwig-Str. 10-12, Konstanz, Germany

Keywords: Visual component composition, components, component language, component fragment, distributed component systems.

Abstract: Component composition has been remaining over a decade a (design) concept, but not found its way into practical programming which is usually still done in the classical reference-based way. A new generation of component languages like ArchJava has pushed forwards composition of subcomponents. But these languages fall back into class-based programming of methods when Java program code is to be written e.g. as a filter among subcomponents. In contrast, the CompJava Designer, a graphical editor, allows constructing relatively complex and distributed component systems for practical applications by a seamless visual composition process. It uses extended UML 2 component diagrams that allow visualizing the compositional structure of components in order to better understand and communicate it. The designer is based on the component language CompJava that has introduced component fragments and plugs as means for composing a component both from subcomponents and structured units of code.

1 INTRODUCTION

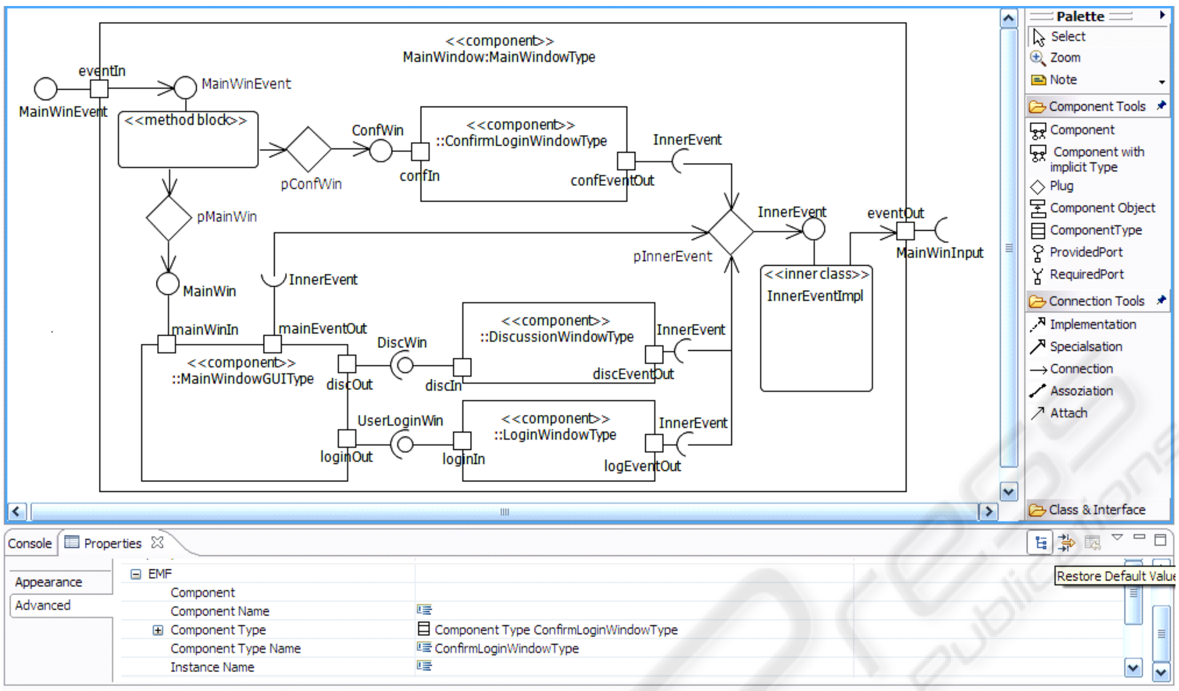
Component composition (C. Szyperski, 1997) is less error-prone than class-based programming with reference handling and provides for a much clearer and cleaner architecture. However, it has been for a decade a concept that supplements classical reference-based programming, but does not replace it to a larger extent.

Classical component models, like CORBA (K.Seetharaman, 1998), Enterprise JavaBeans (Sun Microsystems, 2001), Corba Component Model, and DCOM (C. Szyperski, 1997), define, with a few exceptions for special cases, only provided, but no required interfaces. Thus, the composition of components is just a conceptual process that must be realized by handling component resp. class references.

After the development of mathematically oriented composition-calculi, there have been efforts to make them available for a practical application (J.C.Seco and L.Caires, 2000). A new generation of component languages based on that approach, like ArchJava (J. Aldrich et al, 2002), and ACOEL (V.C.Sreedhar, 2002), defines also required interfaces. A connect-statement allows carrying out composition of subcomponents in an elegant way. But these languages fall back into class-based programming with reference handling when component code is to be supplied e.g. as a filter among subcomponents. As a consequence, design on a conceptual level is done by composition; but its realization is done to quite a large extent in the same way as class-based programming, as program examples from ArchJava (J. Aldrich et al, 2002) show. Another weakness is that these languages do neither provide for a distribution model nor services so that they are not apt to realize distributed systems.

To push forward composition was one of the objectives we had in developing the component language CompJava (H.A.Schmid, 2007) based on concepts from the new component language generation. It allows for composing a component both from subcomponents and structured units of code, introducing for that purpose component fragments and plugs.

Additionally, it embodies also a distribution model for the seamless composition of a system from local components, distributed components and services.



EMF
Component
Component Name
Component Type: Component Type ConfirmLoginWindowType
Component Type Name: ConfirmLoginWindowType
Instance Name

Figure 1: CompJava Designer showing of the composition of the MainWindow component from subcomponents and component fragments.

During the development and work with CompJava, we noticed that we usually visualized the compositional structure of non-trivial components in order to to better understand and communicate it. We have put the visualization on a sound base by defining CompJava diagrams, which extend UML 2 component diagrams (OMG, 2007) with component fragments, plugs and the associated "wiring". Their transformation into the CompJava language is defined precisely.

A graphical editor, called CompJava Designer, was the last step towards a visual composition of components. We show in this paper that relatively complex distributed component systems may be constructed seamlessly by visual composition, without referring to the textual form of the component language. The CompJava Designer was constructed as an Eclipse plug-in from the model-driven framework GMF, which is also an Eclipse-plug-in. Let us mention shortly without going into detail that the CompJava Designer is generated from the same meta-model that is used by the CompJava compiler in order to represent the result of syntactical analysis for code generation.

This paper presupposes some basic familiarity with components and UML 2 component diagrams. Its organization is the following. Section 2 presents the CompJava Designer with its different kinds of diagrams and the composition process. Section 3 gives a rough overview about the (textual) component language as a background for the visual design. Section 4 presents a non-trivial example for the composition of a distributed chat application system. Section 5 discusses related work.

2 COMPJAVA DESIGNER

CompJava is a distributed Java-based component language. The first non-distributed version has been available since winter 2003/2004, three more compiler and language versions followed. The current version with distributed components is integrated in Eclipse and available on www.compjava.org.

CompJava allows composing a component from subcomponents, which are instances of other components, and from the newly introduced component fragments. It defines component types, components and component instances, similarly as Java defines interfaces, classes and class instances. Each component has a component type. One may create any number of component instances from a component.

The CompJava Designer (used to create all presented diagrams except for Figure 2) is a graphical design tool available as Eclipse-plug-in. It allows the visual composition of components by constructing CompJava diagrams and generating code from them. For a graphical representation of component composition, we use CompJava component diagrams which are UML 2 component diagrams (OMG, 2007) enriched with component fragments and plugs.

Component fragments structure the component code so that a component has (practically) no methods. A component fragment may be used as a kind of filter for subcomponents or for building a component directly from Java code. A plug is used as a connection point for the “wiring“.

We use the composition of a chat client of an instant messaging system as a running example. Instant messaging is a distributed application formed from services, distributed components and local components. Here we present mainly local aspects. The instant messaging system was developed by a student without prior knowledge of CompJava as a diploma thesis (M. Klenk, 2002).

2.1 Example

The CompJava Designer (see Figure 1) displays: a CompJava diagram in its main window; a palette with tools for constructing CompJava diagrams; and (bottom) a textual editor for properties of CompJava diagram objects. The diagram is a CompJava composition design diagram that shows the design of the composition of the *MainWindow* component of the chat client.

MainWindow is a component of component type *MainWindowType* (analogous to class and interface). It presents in encapsulated windows all information about chats and users except for the exchanged messages. *MainWindow* receives new information from the server about events via the provided *MainWinEvent* port, and it sends off information about events from its user via the required *MainWinInput* port. It is composed from subcomponents with the types *MainWindowGUIType*, *DiscussionWindow-Type*, *LoginWindowType*, and *ConfirmLoginWindowType*, and from two component fragments implementing the interface *MainWinEvent* res. *InnerEvent*.

The inside of the ports of a parent component like *MainWindow* may be “wired“ to ports of subcomponents or to component fragments like that implementing *MainWinEvent*. Ports of subcomponents may be “wired“ to ports of other subcomponents, like the required ports of *MainWindowGUIType* to provided ports of *DiscussionWindowType* and *LoginWindowType*, or via the intermediary of a plug (depicted by a diamond) like *pInnerEvent* of type *InnerEvent* (see middle right) to a component fragment like the one implementing the interface *InnerEvent*.

The “wiring“, depicted by arrows, which represents the connect- and attach-statements of the CompJava language (compare 3.2), is subject to consistency constraints. The conditions to be fulfilled for composing components are: the port-matching constraint is that a provided (port) interface extends (incl. equals) a required (port) interface; and the n:1 multiplicity constraint is that a required port is connected or attached to only one provided port res. plug, and a plug is attached to only one component fragment or provided port. These constraints are checked in real-time by the editor.

2.2 CompJava Diagrams

The CompJava Designer allows constructing four different kinds of CompJava diagrams:

- A **Port Interface Diagram** defines port interfaces in the form of Java interfaces (in UML or text form).
- A **Component Type Diagram** defines a component type specifying all port interfaces by which a component of that type may collaborate with the outside. It specifies also the distribution-related property whether the ports may be invoked remotely (via RMI/Corba) or may define services.
- A **Composition Design Diagram** shows the design of the composition of a component (see Figure 1). It specifies: the component type (but not name) of subcomponents; component fragments; the wiring; and also whether the implementation of the component may be distributed, i.e. it may have remote subcomponents. The design of a component fragment specifies the interface it implements; it specifies indirectly (via the wiring) the plugs or ports from which it may invoke methods
- A **Composition Implementation Diagram** shows the implementation of the composition of a component. It is constructed from a composition design diagram by selecting the subcomponents, which each must have the specified component type; and by implementing the specified component fragments, and possibly inner classes. A component fragment is implemented either by an anonymous class, an inner class or as a method block (the latter is depicted like an anonymous class without class

head) (for a more details see (H.A.Schmid, 2007)). One implements a component fragment in an automatically opened Java editor window, which provides the methods implementing the component fragment interface, with empty implementations to be filled out.

In contrast to UML2 component diagrams, we distinguish between composition design and implementation diagrams since

1. in general, one should distinguish design and implementation
2. when designing the composition of a component, only the component types of the subcomponents, but not their implementation should have to be known.

Once the implementation of a composition is finished, one may start automatic code generation and compilation using under the cover the CompJava compiler. For designing and implementing the composition of static component architectures, there is usually no programming of CompJava code and no separate programming of Java code required.

2.3 Composition Process

Component composition is a fully hierarchic process so that components may be nested to an arbitrary level. The intuitive diagram of Figure 2 kind of collapses the formal diagrams (which contain only one nesting level) of Figures 3, 4 and 5.

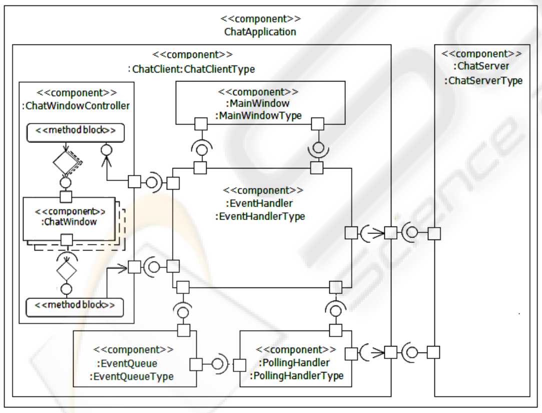


Figure 2: The figure shows component composition for only one nesting level.

When composition was a strict top-down process, one would perform the following steps one after another:

1. specifying the types of the highest-level component like *MainWindowType,* and of its subcomponents like *MainWindowGUIType* and *DiscussionWindowType*;
2. specifying port interfaces like *MainWinEvent* and *MainWinInput*;
3. designing the composition of the parent component like *MainWindow* from subcomponents of a given type like like *MainWindowGUIType* and *DiscussionWindow-Type* and from component fragment specifications, and design the "wiring";
4. implementing the composition.

However, component composition is in reality not a strict top-down process, but re-iterates the different process steps. It may be necessary to specify or modify port interfaces and component types when designing the composition or even implementing the composition of a component. Therefore, the CompJava Designer allows specifying or modifying port interface diagrams or component type diagrams together with composition design diagrams and composition implementation diagrams.

Section 4 describes the composition design of the client component of the chat application, which forms a relatively complex real-world example.

3 COMPJAVA

This section gives a rough overview about the (textual) component language as a background for the visual design. A systematical introduction and definition of the local language constructs is given in (H.A.Schmid, 2007).

3.1 Component Type

Component types allow defining e.g. product line architectures or component frameworks, and they allow separating the design of component composition from its implementation. That means the composition of a component from subcomponents may be designed with the subcomponents types, prior to the design of the subcomponents, as section 2 has shown.

For example, consider the type *MainWindowType* (of a *MainWindow* component):

```
interface MainWinEvent { ...};
interface MainWinInput { ...};
   component type MainWindowType {
     port eventIn provides MainWinEvent;
     port eventOut requires MainWinInput;
}
```

A component type specifies the ports via which a component may collaborate with other components, using Java interfaces as port interfaces. E.g. the MainWindowType defines the port eventIn with the provided interface MainWinEvent and the port eventOut with the required interface MainWinInput. CompJava allows declaring also event ports and port arrays.

A remotable component type, which is denoted by the modifier "remotable", imposes the remotability restriction on the (local) port interfaces: the provided and required interfaces must expose only types with a copy-semantics, or references to distributed components. A service component type imposes the restriction that all port interfaces must be service interfaces. A service interface is constrained to exposing only Java primitive types or serializable types that are formed essentially by data.

3.2 Components

A component has the component type indicated by the ofType-clause; it implements the provided interfaces, possibly using operations of required interfaces. A distributed component, like *ChatServer*, is composed from subcomponents, which may be allocated remotely from the component. This implies that the subcomponents have remotable types (or service types).

We use the MainWindow component (see visual design in section 2) to illustrate some main constructs of CompJava. Example I presents schematically the code of the MainWindow component.

Example I. Component *MainWindow* with subcomponents *MainWindowGUI*, *Discussion-Window*, *LoginWindow* and *ConfirmLoginWindow*, component fragments, plugs and "wiring"

```
component MainWindow ofType MainWindowType {
    //port eventIn provides MainWinEvent;
    // port eventOut requires MainWinInput;
    MainWindowGUIType m =
                new MainWindowGUI();            *
    DiscussionWindowType d =
                new DiscussionWindow();         *
    LoginWindowType l =
                new LoginWindow();              *
    ConfirmLoginWindowType cl =
                new ConfirmLoginWindow();            *
    plug<MainWin> pMainWin;                     **
    plug<ConfirmLoginWin> pConfWin;             **
    plug<InnerEvent> pInnerEvent;               **
    attach This.eventIn to new MainWinEvent { ***
        ...
    }
    attach This.pInnerEvent to new InnerEvent {       ***
        ...
    }
    connect This.pMainWin to m.mainWinIn;
      ****
    connect m.loginOut to l.loginIn;                *****
    connect m.discOut to d.discIn;                  *****
    connect m.mainEventOut to This.pInnerEvent;
      ****
    connect This.pConfWin to cl.confIn;             ****
    connect cl.confLogEventOut to This.pInnerEvent;
      ****
    connect d.discEventOut to This.pInnerEvent;
      ****
    connect l.logEventOut to This.pInnerEvent;  ****
}
```

At component initialization time, the *MainWindow* creates the subcomponents *MainWindowGUI*, *DiscussionWindow*, *LoginWindow* and *ConfirmLogin-Window* (as indicated by a *).

It declares the plugs *pMainWin*, *pConfWin*, and *pInnerEvent* (as indicated by **). A plug is required as intermediary to connect a port of a subcomponent to a component fragment that implements the plugs interface. A plug has an (interface) type; it is a kind of a light-weight port for use within a component, passing invocations from a subcomponent to a component fragment or vice versa.

Further, *MainWindow* creates the component fragment implementing the *MainWinEvent* interface in the form of an anonymous class, attaching it to the inside ot the *eventIn* port with the type *MainWinEvent* of the component instance (denoted by "This"), and the one implementing the *InnerEvent* interface attaching it to the plug *pInnerEvent* of type *InnerEvent* (as indicated by ***).

A component fragment implements an interface and is either an anonymous class, an inner class (both as defined by Java) or a method block (defined by CompJava as a block containing only methods). An attach-statement attaches the inside of a provided port or a plug to a newly created component fragment.

Further, *MainWindow* connects the ports of subcomponents to plugs (as indicated by ****) or among themselves (as indicated by *****) or with the inside of a parent component port (not shown).

A connect-statement may connect a required port of a subcomponent (instance), like *loginOu* of *MainWindowGUI m*, to a provided port of a subcomponent (instance), like *loginIn* of *LoginWindow l*. The compiler checks all consistency constraints. A connect-statement may also connect a port of a subcomponent with the inside of a matching port of the (parent) component or a matching plug.

4 COMPOSING A CHAT APPLICATION

This section describes the design of the chat client of the running example. Apart from some coding prototypes, the design was done as described in section 2.3 by composing components from subcomponents and component fragments, specifying the component types with the port types, the main responsibilities and the required wiring. The CompJava Designer was still in a prototype stage; so we had to simulate partially its work constructing CompJava diagrams with other tools and transforming them manually into code. A current larger project is using the CompJava Designer; first experiences are good.

4.1 Visual Design of Chat Application

The outmost component of the chat application has the *ChatApplType*. We specify with the CompJava Designer in a component type diagram that *ChatApplType* defines no ports. In the sequel, we describe the design process without referring to the use of CompJava Designer.

We design the *ChatAppl* component (see Figure 3) to be composed from the service subcomponents (more precisely: instances of them) with the type *ChatClientType* and *ChatServerType*. When we specify these two types, we design the basic working mode of the system. We decide that the chat client has only required ports, and correspondingly the chat server only provided ports. Consequently, the client polls the server for new messages and other information from other clients.

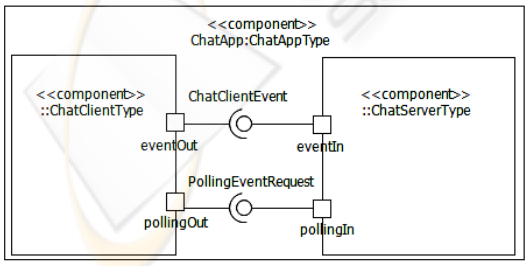


Figure 3: Design of the *ChatAppl* component composed from service components with *ChatClientType* and *ChatServerType*.

The next decision is whether client and server have each one port or two ports. It goes together with the specification of the port interfaces. It may be required to look deeper into the design of the client and server component in order to make a sound decision.
We have defined two ports: a port with the interface *ChatClientEvent* that is used (seen from the client side) to send off messages or requests entered by the client user, and another port with the interface *PollingEventRequest* which is used to poll for new messages for the chats a user participates in, and other information.

After specification of the types *ChatClientType* and *ChatServerType*, we design the *ChatAppl* component as Figure 3 shows, connecting the matching ports of the *ChatClientType* and *ChatServerType*.

4.2 Visual Design of Chat Client

The *ChatClient* collaborates remotely with the *ChatServer* over a Web service. It invokes the *ChatClientEvent* service for sending messages or requests entered by the user, and the *PollingEventRequest* service for polling for events of other users and new messages of the chats a user participates in.

The *ChatClient* component has the *ChatClientType* which defines its ports. It has a non-distributed implementation, i.e. it is composed from subcomponents allocated on the same network node. Each of its subcomponents is designed only for local collaboration, which allows using the more efficient reference semantics. This is expressed visually by the property non-distributed of the *ChatClient* component and by the property non-remotable of its subcomponent types. Both properties are the default in a CompJava diagram.

The *ChatClient* (see Figure 4) displays a main window with sub-windows and a conference window for each chat or conference, and it organizes sending and receiving messages and events to and from the server. It is composed from window-related subcomponents with types *MainWindowType* and *ChatWindowController-Type*, and from messaging-related subcomponents with types *EventHandler Type, EventQueueType* and *PollingHandlerType.*

A component of *MainWindowType* displays the main window. As described, it receives new information about events from other clients via the provided *MainWinEvent* port, and it sends off information about events from its user generated in own threads via the required *MainWinInput* port. A component of *ChatWindowControllerType* displays the chat windows and may create and delete them, receiving and sending off new chat messages and user events via the *ChatWinEvent* res. *ChatWinInput* port.

A component of *EventHandlerType* receives (from the windows) chat messages via its provided *ChatWinInput* port and user events via the same and the *MainWinInput* port, both in the form of operation invocations originating from different window threads. After adding mainly some administrative information, *EventHandlerType* sends user messages and events to the chat server via the required *ChatClientEvent* port.

A component of *PollingHandlerType* has an own thread; it polls for messages and events from the chat server via the required *PollingEventRequest* port and stores them in the *EventQueue*. *EventHandlerType* fetches the incoming messages and events via the *eventIn* port from the *EventQueueType* in an own thread and passes them after removing some administration information to the respective window. The thread is in a wait-state when the *EventQueue* is empty. *EventQueueType* provides each a port for storing and fetching messages and events.

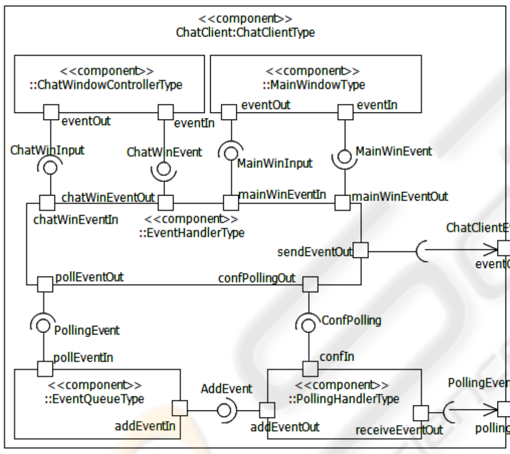


Figure 4: *ChatClient* composed from MainWindow, EventHandler, ChatWindowController, Event- Queue, and PollingHandler.

The *PollingHandler*, *EventQueue*, and *EventHandler* components are composed from Java code in the form of component fragments, whereas *MainWindow* and *ChatWindowController* are composed from subcomponents and component fragments.

4.3 Visual Design of Dynamic Component Architectures

The *ChatWindowController* is a medium to low level component with a dynamic component architecture. It is composed from two component fragments and a variable number of subcomponents. It displays a chat window for each chat, creating res. deleting chat windows dynamically upon a corresponding user interaction or upon receiving a corresponding user event. It is composed from a variable number of subcomponents of *ChatWindowType*, realized by a *ChatWindowType* array, and from two component fragments that serve as a kind of filter between the ports of *ChatWindowController* and those of *ChatWindowType*.

CompJava allows for dynamic architectures which require a creation and connection of components not only at component initialization time, but also dynamically during the execution of methods (see (H.A.Schmid, 2007) for details). It allows declaring an array of component type, like that of *ChatWindowType*, and creating or deleting subcomponent instances dynamically. It allows declaring port arrays (not shown in the example) and plug arrays. Each plug of the plug array *pWinControl* of type *WinControl* is connected with the *winEventIn* port of the corresponding *ChatWindowType* instance, and each *winEventOut* port is connected to the plug *pWinEvent* of type *WinEvent*.

The component fragment on top of Figure 5 implements the operations defined by the *ChatWinEvent* interface, which are invoked via the *eventIn* port. It passes via the *pWinControl* plug new messages to the subcomponents of *ChatWindowType*, and opens and closes an instance of them when a user requires in the *MainWindow*. It implements a simple administration of *ChatWindowType*'d component instances in order to reuse a closed instance when a new one is to be opened.

The subcomponent of *ChatWindowType* contains a chat window built with the Swing library that displays the messages, and allows to type in new messages and to close a chat by the owner.

The component fragment on bottom of Figure 5, implements the *WinEvent* operations invoked from the *winEventOut* port of the *ChatWindowType* subcomponents via the *pWinEvent* plug. The interface *WinEvent* defines two operations, one passing a new message entered in a chat window of *ChatWindowType*, and the other one removing a chat window when a chat is closed by the owner. Note that the CompJava Designer can generate for the dynamic creation of subcomponents and the connection of their ports only code samples but not the code. The reason is that the creation may be done at run-time e.g. in the methods of component fragments.

5 RELATED WORK

Component Languages. CompJava is based on and improves on local component language concepts from ArchJava (J. Aldrich et al, 2002), ComponentJ (J.C.Seco and L.Caires, 2000) and ACOEL (V.C.Sreedhar, 2002). A version of ArchJava (J. Aldrich et al, 2003) extends the syntax of connect patterns and expressions, so that a user may realize remote collaborations among components with user-defined connector types. But this is quite complex and may have the consequence that either structural distribution problems are detected only at run-time, or that a component may type-check correctly with one kind of connector but not with another one.

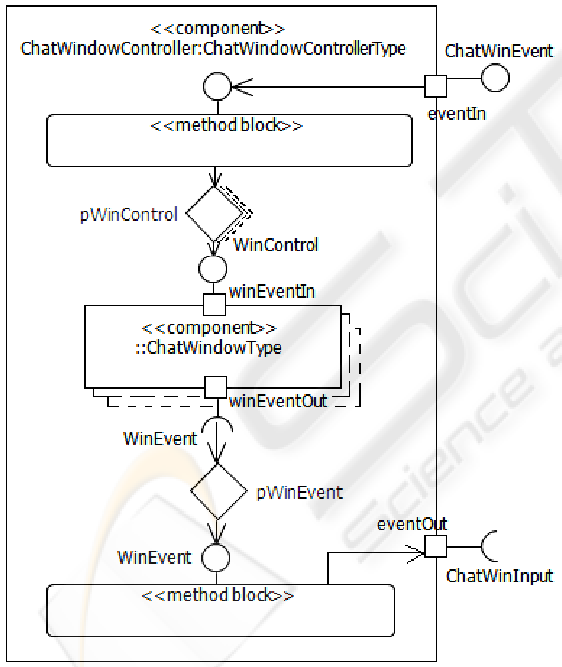


Figure 5: CompJava component diagram for *ChatWindowController* composed from an array of *ChatWindow* subcomponents, component fragments and plugs.

Component Models. The new generation of component languages connects required and provided ports; it does not require or allow the handling of component references like classical component models: CORBA (K.Seetharaman, 1998), Enterprise JavaBeans (Sun Microsystems, 2001), Corba Component Model, DCOM (C. Szyperski, 1997) and Dotnet. (W. Emmerich, 2002) gives an overview on distributed component technologies and their software engineering implications. KOALA, a component technology for resource-constrained environments like TVs (R.van Ommering et al, 2000) is a local technology though distribution via middleware may be embedded in it.

Connectors and Architecture Description Languages (ADL). ADLs describe primarily local systems as the ADL classification framework shows; it does not have any distribution-related item as an architecture modeling feature (N.Medvidovic and R.P.Taylor). But e.g. (N. .Medvidovic et al, 1999) uses connectors to encapsulate middleware and provide remote access among components (E.Dashofy et al, 1999).

The ADL classification framework distinguishes connectors that are modeled explicitly as first class entities, and implicitly modeled ones. CompJava has the latter ones, parameterized by the interface type.

6 CONCLUSIONS

The graphical editor CompJava Designer allows a seamless visual design and implementation of the composition of relatively complex component systems. It uses CompJava diagrams, an extension of UML 2 component diagrams, which have proven very valuable in order to visualize the compositional structure of components for a better understanding and communication. The transformation of a CompJava component diagram into the CompJava language is straightforward and precisely defined for static architectures.

The CompJava Designer is based on the component language CompJava, which pushes composition of components a further step by introducing component fragments that may participate via plugs in the composition process. It generates automatically the CompJava and Java code.

The CompJava Designer and CompJava component diagrams have proven their value and practical applicability in relatively complex projects, one being a chat application (M. Klenk, 2002), another one an Internet component framework for card games with dynamically attachable new games. The CompJava Designer has been made available recently; our first experiences are very encouraging.

ACKNOWLEDGEMENTS

Our thanks go to Michael Klenk for the design, implementation, and test of the chat application; to Tobias Seckinger and Reiner Weinhold for cooperating in the development of the CompJava Designer; and to the Ministerium für Wissenschaft und Forschung, Baden-Württemberg, for a partial project support.

REFERENCES

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

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

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

Special section on CORBA, 1998. *Communications of the ACM*. Vol.41, No10.

E.Dashofy, N.Medvidovic, R.P.Taylor, 1999. *Using Off-The-Shelf Middlewareto Implement Connectors in Distributed Software Architectures*. ICSE'99.

Sun Microsystems, 2001. *Enterprise JavaBeans Specification Version 2.0.* www.java.sun.com.

W. Emmerich, 2002. *Distributed Component Technologies and their Software Engineering Implications.* Procs. ICSE.

M. Klenk, 2006. *Entwurf einer Chatapplikation mit der Komponentensprache CompJava.* Diploma Thesis, Faculty for Informatics, University of Applied Sciences Konstanz.

R. Monson-Haefel, 2001. *Enterprise JavaBeans, O'Reilly, Sebastopol.*

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

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

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

R.van Ommering, 2002. *Building Product Populations with Software Component.*

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

K.Seetharaman, 1998. *The CORBA Connection.*

H.A.Schmid, M.Pfeifer, 2007: *Engineering a Component Language: CompJava.* Selected Papers from ICSOFT 2006 Springer Lecture Notes.

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

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

OMG, 2007. Unified Modeling Language Specifications 2.1.1 UML component diagrams. www.uml.org