Abstract

Component composition has been over a decade a concept that supplements classical reference-based programming, but does not replace it to a larger extent. Though a new generation of component languages like ArchJava has pushed forwards composition from subcomponents, 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. To allow for a seamless composition process, we have developed the component language CompJava that introduces component fragments and plugs as means for composing a component both from subcomponents and structured units of code. In parallel, we have extended UML 2 component diagrams by the newly introduced concepts. This allows visualizing the compositional structure of components in order to better understand and communicate it. A graphical editor, called CompJava Designer, allows constructing relatively complex and distributed component systems by a seamless visual composition process.

1. INTRODUCTION

Component composition [19] 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 [4] [16], Enterprise JavaBeans [7] [10], Corba Component Model [5], and DCOM [19], define, with a few exceptions for special cases, only provided, but no required interfaces. Thus, composition remains a conceptual process that is realized by handling component references.

After the development of mathematically oriented composition-calculi, there have been efforts to make them available for a practical application [15]. A new generation of component languages based on that approach, like ArchJava [1] [2], and ACOEL [18], defines also required interfaces. A connectivity statement allows carrying out composition of subcomponents in an elegant way. But these languages fall back into class-based programming with reference handling when code is to be supplied e.g. as a filter among subcomponents. Another problem is that these languages do neither embody a distribution model nor services so that they are not apt to realize distributed systems. As a consequence, design on a conceptual level is done by composition, but its realization is done to quite a large extent by class-based programming, as program examples from ArchJava [1] [2] show.

To push forward composition was one of the objectives we had in developing the component language CompJava [17], which is 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.

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 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-plugin.

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 UML2 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 and service components is integrated in Eclipse and soon available on www.compjava.org.

CompJava allows composing components from subcomponents 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 is a graphical design tool available as Eclipse-plug-in. It allows composing components visually, 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 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 to build a bottom-level component from Java code or as a kind of filter for subcomponents. The design of a component fragment specifies the interface it implements; it specifies indirectly (via the wiring) the plugs or ports it may invoke methods from.

A component fragment is represented according to the selected implementation as an anonymous class, an inner class or as a method block (the latter depicted like an anonymous class without class head). A plug is used as a connection point for the “wiring”; it is depicted by a diamond.

We use instant messaging as a running example for a distributed application formed from services, distributed components and local components. The instant messaging system was developed by a student without prior knowledge of CompJava as a diploma thesis [9].

2.1 Example

The CompJava Designer (see “Fig. 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 CompJava diagram in “Fig. 1,” a is composition design diagram that shows the design of the composition of the MainWindow component of the chat client.

The MainWindow component presents, together with associated windows, all information except for the messages exchanged in chats. 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 (invisible at the right). It is composed from subcomponents with the types MainWindowGUIType, DiscussionWindowType, LoginWindowType, and ConfirmLoginWindowType, and from two component fragments implementing the interface MainWinEvent res. InnerEvent.

The inside of a parent component ports’ 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, LoginWindowType, and ConfirmLoginWindowType, or via the intermediary of plugs like pInnerEvent of type InnerEvent (see middle right) to component fragments.

2.2 CompJava Diagrams

The CompJava Designer allows to construct four different kinds of CompJava diagrams:

- A port interface diagram defines Java interfaces (in UML or text form) used as port interfaces.
- A component type diagram defines a component type. It specifies all port interfaces over which a component of that
type may collaborate with the outside. It specifies also the distribution-related property whether the ports are remotely invokeable (via RMI/Corba) or define services.

- A composition design diagrams designs the composition of a component, as shown in “Fig. 1.” It specifies: the component type (but not name) of subcomponents; component fragments; and the “wiring”. It specifies also whether the implementation of the component may be distributed, i.e. have remote subcomponents.

- A composition implementation diagram is created from a composition design diagram by selecting the subcomponents, which must have the component type; and by implementing the component fragments and possibly inner classes. The latter is done each in an automatically opened Eclipse Java editor window that provides the methods implementing the component fragment interface with empty implementations.

Once the composition implementation is terminated, one may start automatic code generation and compilation with the CompJava compiler. That means no programming of CompJava code and no separate textual programming of Java code is required in order to design and implement a component by composition (if a component does not contain additional classes). Code generation may be quite complete for static component architectures and less complete for dynamic architectures.

### 2.3 Composition Process

When component composition were 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 `MainWindowGUIType` and `DiscussionWindowType` 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 systematic introduction and definition of the local language constructs is given in [17].

#### 3.1 Component Type

Component types allow defining e.g. product line architectures or a component framework, and they allow separating the design of a component composition from its implementation. Further, 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. We use the declaration of the type `MainWindowType` of the `MainWindow` component as an example:

```java
interface MainWinEvent { ...};
interface MainWinInput { ...};

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

A component type specifies the ports, 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 an interface on which stronger restrictions are imposed than on a remotable interface. A service interface is constrained to expose 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 3) to illustrate some main constructs of CompJava. Example 1 presents schematically the code of the `MainWindow` component.

Example 1. Component `MainWindow` with subcomponents `MainWindowGUI`, `DiscussionWindow`, `LoginWindow` and `ConfirmLoginWindow`, component fragments, plugs and “wiring”:

```java
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<MainWindow> pMainWin; **
    plug<ConfirmLoginWin> pConfWin; **
    plug<InnerEvent> pInnerEvent; **
    attach This.eventIn to new MainWinEvent {
        ...
    }
}
```
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 “Fig. 2”) 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.

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 “Fig. 2” 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-removable of its ChatClient component and by the property non-removable of its subcomponent types. Both properties are the default in a CompJava diagram.
The ChatClient 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 ChatWindowControllerType, and from messaging-related subcomponents with types EventHandlerType, EventQueueType and PollingHandlerType (see “Fig. 3”).

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. 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.

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.

In the other direction, 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.

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 [17] 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 “Fig. 4” 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 “Fig. 4,” 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 [1] [2], ComponentJ [15] and ACOEL [18]. A version of ArchJava [3] 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.

Component Models. The new generation of component languages connects required and provided ports,
6. CONCLUSIONS

In order to push the composition of components a further step forward, the component language CompJava introduces structural units of component code that may participate via the intermediary plugs in the composition process.

An extension of UML 2 component diagrams by the new features has 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.

The graphical editor CompJava Designer allows a seamless visual design and implementation of the composition of relatively complex and distributed component systems. It generates automatically the CompJava and Java code.

CompJava component diagrams have proven their value and practical applicability in a relatively complex project [9], using a CompJava Designer prototype and partially simulating its work. The CompJava Designer has been made available recently; our first experiences are very encouraging.

7. ACKNOWLEDGEMENTS

Our thanks go to Michael Klenk for the design, implementation, and test of the chat application, in particular for Figures 3 and 4; and to the Ministerium für Wissenschaft und Forschung, Baden-Württemberg, for a partial project support.

8. REFERENCES

[5] CORBA Components; at cgi.omg.org