A Framework For Workflow-oriented Scripting in Java Applications

Mathias W. Richter[1]

Cap Gemini (Schweiz)
CH-8048 Zürich, Switzerland
mathias@acm.org

Abstract. This paper describes FlowServer, a Java-based framework which allows the vertical integration of workflow into any Java application. FlowServer provides a scripting language which combines an interpreter for a subset of the Java programming language with process abstractions as found in workflow definition languages. This approach integrates access to compiled software components on the Java platform and other platforms (through any Java-based middleware). We first describe different FlowServer-based application architectures. Then we give an overview of the workflow definition language provided by the framework. Finally, we briefly discuss a few design and implementation characteristics.

1 Introduction and Related Work

Business applications (implemented in Java or not) typically belong to the category of *forms-based applications*. Forms-based applications present the user with forms or tables of fields for viewing, entering and modifying business data. Forms-based applications usually implement more or less modal interfaces to *processes* or *workflows* which a user passes through to perform a certain task.

A variety of workflow management systems (WFMS, e.g. [8, 9, 16]) for implementing forms-based applications realizing business-processes already exist (e.g. [2, 7, 14]). Some of these systems have also started to integrate Java as implementation platform (e.g. [3]). However, such WFMS cannot be embedded into Java-based applications, because they do not offer the necessary interfaces and because applications often do not need the features of the rigid workflow architectures offered by these systems. A seamless and flexible vertical integration of workflow-oriented components into Java applications is useful for two main reasons:

[1] This work was performed while the author was employed at iFace AG, Aarauerstr. 55, CH-4600 Olten

- Forms-based applications implementing business processes are required to integrate a variety of software components on the basis of varying middleware (e.g. EJB, RMI, Corba, JDBC [13, 15]).
- User and integration requirements are too specific to be implemented using a general-purpose workflow management system.

To fulfil these requirements, an embeddable process- or workflow-oriented component should possess the following characteristics:

- *lightweight*: The component should not force the embedding application to incorporate any element of a full-featured workflow management architecture (e.g. a database for persistent workflow state) beyond the notion of processes and activities.
- *scripted*: The component should provide a scripting-based workflow definition language (WDL) which allows the flexible adaptation of the processes/workflows using classes of the embedding application and the Java platform.

The FlowServer framework was designed to possess these characteristics and provides a set of classes for specifying, interpreting and executing processes consisting of activities.
Motivation for the approach described here was an existing software product based on a distributed software architecture. Whenever the product was introduced in a new company, the implemented business processes had to be adapted. This was difficult, because the business processes provided by the product where implemented using an object-oriented metaphor. Thus, we needed an embeddable, approach to scripting-based implementation of business processes which could be adapted without changing any application classes.
The FlowServer framework provides an interpreted workflow definition language (WDL). This WDL provides workflow-specific abstractions (process/activity definitions, process/activity state), where activity definitions consist of blocks of Java source code. These Java code blocks are executed depending on the state of the according activity instances. At execution time, this code is interpreted using a Java-based interpreter [10, 11] for a subset of the Java programming language.

2 FlowServer Application Architectures

Usually, forms-based (Java) applications follow a simple architectural pattern where the user interacts with a user interface created by the application. The application itself is coarsely separated into classes implementing application logic and classes implementing the user interface.[2] The flow of the business processes supported by the application is implemented by the application classes. This approach has several disadvantages: The implementation of the business processes is distributed over a

[2] For now, we neglect aspects of distributed computing in enterprise environments.

number of classes. Also, it is difficult to change the business processes since their implementation is distributed over the application classes. Finally, the flow of the actual business processes is hard to understand, because the business processes are implemented using an object-oriented paradigm instead of using a process-oriented paradigm.

The FlowServer framework allows applications to „factorize“ the business process implementation into process definitions (see Figure 1). These are executed by a flow engine as part of the framework. The advantages are obvious: Business processes are implemented in an independent subsystem of the architecture using a process-oriented metaphor. Because the business processes are implemented using a scripting approach, business processes can be changed independently of the development cycle of the embedding application.

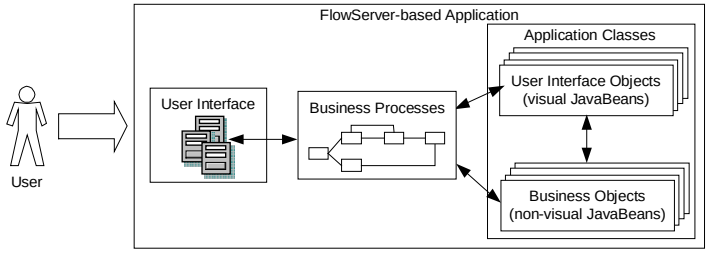

Fig. 1. A generalized picture of a FlowServer-based application architecture.

The basic operation of the FlowServer framework is simple: A flow engine executes processes (see Figure 5 for an example).[3] A process consists of activities and further processes (i.e. subprocesses). An activity is the atomic unit of execution.[4] When the user makes a transition from one activity to the next, this next activity „is executed“ and the user can interact with the result of that execution. The nature of that result depends entirely on the activity definition: Usually, it is a window displaying data or some entry form and some change in the process or activity state.

The user can then view and interact with the data displayed in the user interface. Whenever the user feels ready to move on, she chooses the next activity from a list of activities according to the activity map (i.e. the process structure) which is displayed in a separate user interface element.

An application embedding the FlowServer framework must provide a user interface for choosing a process, and for navigating within the specified process structure. The application must also redirect the navigation requests of the user to the flow engine which in response executes the chosen target activity (or subprocess).

This is how the engine „executes“ an activity: An activity as part of a process is defined by means of different *scripts* and activity variables which keep the state of an activity instance during process execution. A script consists of Java statements

[3] To be more precise, the engine creates a process instance from a process definition and executes this instance.

[4] Again, we actually mean activity instances created from activity definitions.

and expressions. Which of the defined scripts of an activity is executed depends on the state of the corresponding activity instance. There are scripts for startup, first-time execution, repeated traversal and teardown of an activity instance. Scripts typically access „business objects“ implementing the actual application logic (e.g. non-visual JavaBeans), gather data from objects implementing application logic, alter process/activity state, and create a user interface for viewing and modifying the gathered data (e.g. using visual JavaBeans). Figure 2 summarizes the resulting standalone application architecture.

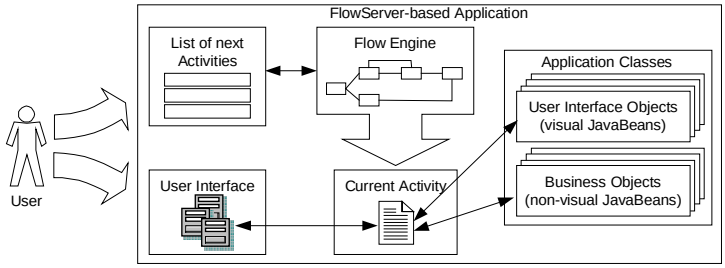

Fig. 2. The architecture of a FlowServer-based application.

We assumed that standalone applications embedding FlowServer components require no distributed services. For applications in an enterprise environment this is hardly ever the case. Rather, business processes implemented by an application use and integrate services of local and remote business objects or database servers.

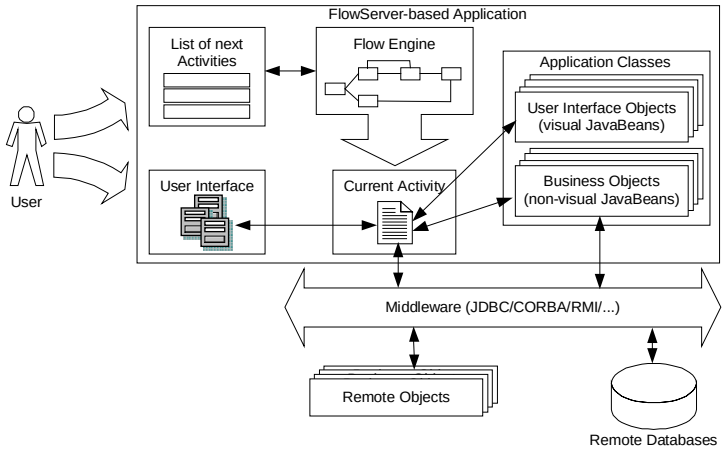

Fig. 3. A FlowServer-based application architecture in a distributed environment.

Access to remote objects and services through any Java middleware products is supported by the FlowServer framework, because the activity scripts provide full access to the Java platform and all available classes (they *are* Java code, as a matter

of fact). Thus, it is possible to use arbitrary middleware services from within processes and activities (see Figure 3), as long as the according middleware implementations (i.e. Java classes) are accessible for the application (i.e. either through the classpath environment or through some network class loader).
A higher degree of distribution can be achieved, if process execution and user interaction of the FlowServer-based application are distributed (see Figure 4): The flow engine is part of a server application which manages client connections, request handling, etc., and executes server-side processes on behalf of clients. As a result, multiple clients can connect to the server. A client software is responsible for displaying the user interface generated for the current activity, for displaying the list of next activities and for handling the according user inputs.
There are several choices of protocols for the exchange of user interface descriptions and data between the client software and the application server. The chosen protocol also determines form and content of the user interface description residing on the server-side:

1. The client can be a HTML-browser. User interface descriptions are HTML-forms, the protocol used between the client and the server is HTTP [6]. The application server offers Servlets to deliver the according HTML-pages which are stored with or rendered dynamically by the application classes.
2. The client can be an applet in an HTML-Browser or a standalone Java application. The used protocol can be a proprietary protocol for user interface description and data exchange (e.g. using XML [1], or ULC [5]) based on Corba, RMI or any other middleware for the Java platform.
3. The client can be an arbitrary standalone application implemented in an arbitrary programming language. Then the protocol used can be a proprietary protocol for user interface description and data exchange based on Corba as middleware.

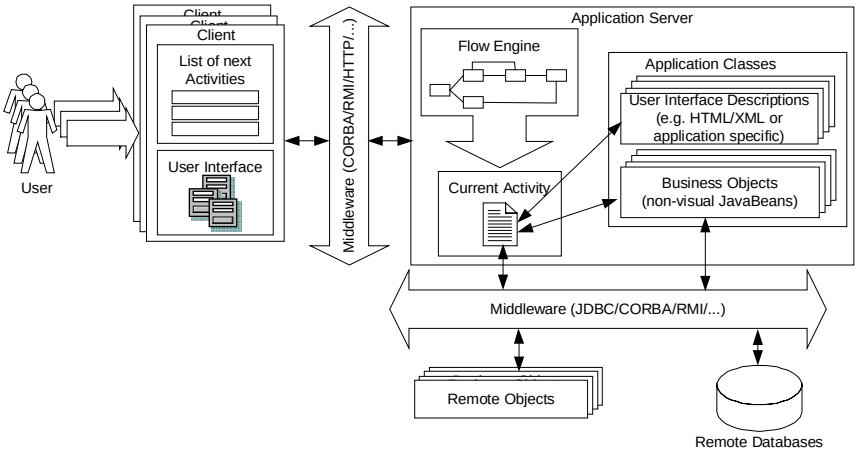

Fig. 4. Server-side process execution and client-side user interaction.

The FlowServer framework currently provides support for creating multithreaded servers which manage client connections based on Corba. These classes handle client representations on the server, multiple processes for each client, the association of clients with processes and of processes with engines executing them.
The implementation of the client presenting the user interface and its interaction with the server managing the engines for the executed processes is left open. This interaction includes requesting the server to start a process or advance within a process, providing data that the user has entered or modified to the server, and processing the user interface descriptions delivered by the server.

3 The FlowServer Workflow Definition Language

The FlowServer WDL allows to define processes which are executed by a flow engine (see Figure 5). A process definition consists of two parts: The *activity definition part* defines the different activities of which a process consists. The *structure definition part* defines different sequences of the defined activities which the user can pass through.
A process can declare local data to keep a process state which is accessible to all activities of the process. A process structure always consists of exactly one start activity and exactly one end activity.
An activity is defined by means of different *scripts*, where each script is executed according to the state of an activity instance which is created from the definition. There is a startup script (executed at instance creation time), an execute script (executed exactly once when an instance is passed through for the first time), a traverse script (executed if an instance is passed through repeatedly) and a teardown script (executed when the instance is destroyed). Furthermore, there are precondition and postcondition scripts which determine whether an instance can be executed and whether the execution was successful. The activities defined as part of the process in Figure 5 use postconditions to call validation routines of the used software components.
The scripts consist of Java source code, which is interpreted when the flow engine executes an activity. For interpreting this Java source code, the FlowServer framework uses the Iava interpreter [10, 11]. The Iava interpreter implements a strict subset of the Java programming language. This subset is best descibed as follows: Any code in a Iava script can be used as method body in a Java class declaration without change. Any function defined in a Iava script can be used as method declaration in the body of a Java class declaration.
Iava offers mechanisms for tight integration with the embedding class. In the case of the FlowServer framework, this is the flow engine. Local process data keeping process state as well as local activity data keeping activity state is declared using Java data types which can be used directly within scripts.

The example in Figure 5 defines a process for booking a flight. The process consists of activities for entering the passenger data, choosing the flight details. After choosing the flight, it has to be paid by credit card. Users can cancel the process at any time by proceeding to the last activity of the process.

```
process BookFlight {
     local data { // process state
        Person passenger; ReservationSystem host;
        FlightChooser flight; CreditCardTransaction credit;
     }
     activities { // definition of activities
        activity EnterPassenger display name "Enter Passenger Data" {
           execute {
              passenger=new Passenger();
              passenger.openEntryWindow();
           }
           postcondition {
              boolean valid=passenger.validateInput();
              if (valid) passenger.closeEntryWindow();
              return valid;
           }
        }
        activity ChooseFlight display name "Choose a Flight" {
           execute {
              host=new ReservationSystem();
              host.connectTo("swissair");
              flight=host.getFlightChooser();
              flight.openEntryWindow();
           }
           postcondition {
              boolean valid=flight.validateInput();
              if (valid) flight.closeEntryWindow();
              return valid;
           }
        }
        activity Pay {
           execute {
              credit=new CreditCardTransaction(passenger.getName());
              credit.openEntryWindow();
           }
           postcondition {
              boolean valid=credit.validateInput();
              if (valid) credit.closeEntryWindow();
              return valid;
           }
        }
        activity Finish {
           input data {
              boolean commit;
           }
           execute {
              if (commit) {
                  credit.commitPayment();
                  if (credit.isPaid()) host.bookFlight(passenger,flight,credit);
              }
           }
        }
     }
     structure { // definition of activity map
        first EnterPassengerData;
        after EnterPassengerData: (ChooseFlight or Finish mapping(false to commit));
        after ChooseFlight: (Pay or Finish mapping(false to commit));
        after Pay: Finish mapping(true to commit);
        last: Finish;
     }
}
```

Fig. 5. A complete process definition in FlowServer WDL.

The last activity “Finish” is parameterized with a boolean flag “commit” which is passed according to the transition used to reach the last acitivity. Only if a flight has been chosen and paid for, the finish activity is invoked with a “true” value for com-

mitting the booking. If the user choses to proceed to the finish from some earlier position in the process, “false” is passed meaning that no booking can be committed. In the example, we assume existence of different “business objects” which are realised as JavaBeans. For reasons of simplicity, we assume that these JavaBeans also implement the user interface for entering and viewing data they represent. The “Person” bean represents data for a passenger. The “ReservationSystem” bean represents the host reservation system (resp. a connection to it) and delivers services such as a list of flights to a certain destination, a schedule of flights for a certain date, etc. The “FlightChooser” bean provides a graphical user interface for displaying and modifying the data delivered by the “ReservationSystem” bean. Finally, the “CreditCardTransaction” bean provides user interface and logic for processing and performing online credit card transactions.
The fact each process must begin with a specified activity (“first”) and end with another specified activity (“last”) allows developers to formulate complete processes with transactional behaviour (as in Figure 5 where the booking is committed upon successful completion of the process).
The example shows a few further features of the FlowServer WDL which we will not discuss here for reasons of brevity.

4 Design and Implementation of the FlowServer Framework

The FlowServer framework implementation follows a three-layered approach which is reflected in the package hierarchy of the framework: The *design layer* (package `flowserver.design`, 10 interfaces) consists of interface definitions, where each interface represents a core abstraction of the framework (e.g. definition, instance, activity definition, activity instance, process definition, process instance, flow engine, etc.).

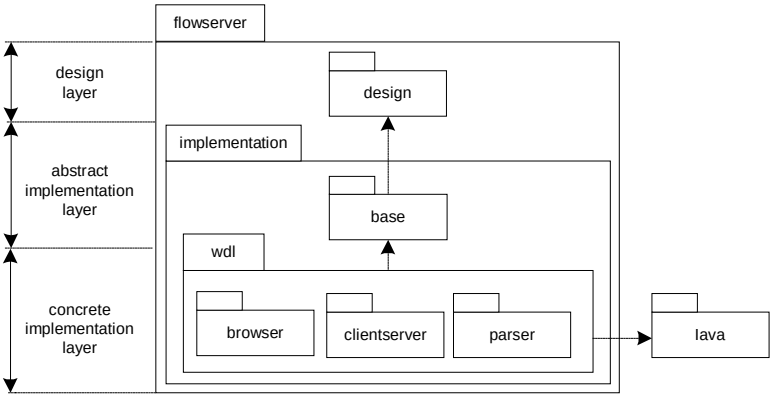

Fig. 6. The three-layered architecture of the FlowServer framework and the according package hierarchy.

The *abstract implementation layer* (package `flowserver.implementation.base`, 6 classes) provides mostly abstract classes which serve as implementation base for one or more concrete implementations. The concrete implementation employing the scripting approach based on the workflow definition language and the Iava interpreter as described in the previous section is located in the concrete implementation layer (package `flowserver.implementation.wdl`, 10 classes). This concrete implementation groups some more functionality in subpackages (`flowserver.implementation.wdl.browser`, 6 classes, `flowserver.implementation.wdl.clientserver`, 6 classes, 1 interface, `flowserver.implementation.wdl.parser`, 35 mostly parser-related classes). The WDL parser was implemented using JavaCC [12].

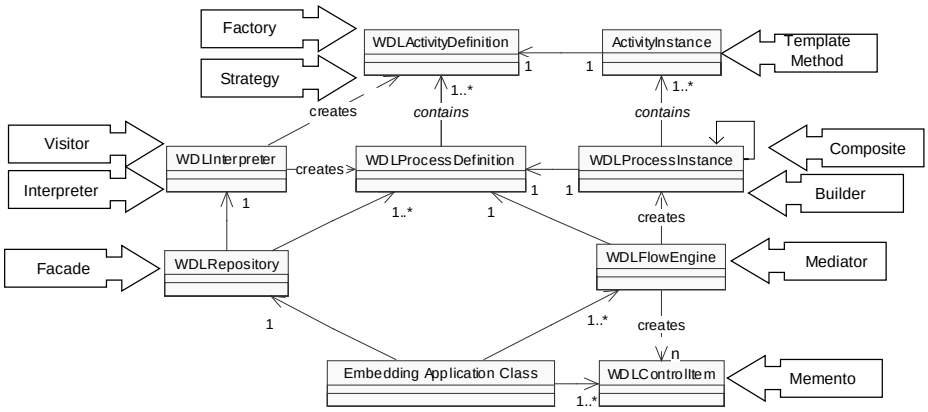

Fig. 7. The relationship between the core FlowServer classes (in the concrete implementation layer) and an embedding application class[5].

The abstract implementation layer and the concrete implementation are characterized by a high degree of *design pattern density* [4]: Used patterns include Strategy/Template Method (for executing the different scripts of an activity), Interpreter (for the WDL-interpreter), Visitor (for the WDL parse tree), Composite (for processes/subprocesses), Facade (for hiding details of interpreting the WDL), Mediator (for the flow engine executing activities and processes), Factory/Builder (for creating process/activity instances from definitions).
An embedding application class only has to interface with three classes of the concrete implementation layer (see Figure 7): `The WDLRepository` acts as a facade for having the `WDLInterpreter` create and for managing `WDLProcessDefinition` objects. The `WDLFlowEngine` is the mediator which manages the execution of processes and their activities. The flow engine creates a memento of the current state of process execution by means of `WDLControlItem` objects which are used to display the process navigation to the user.

5 Conclusions

In this paper we have described the FlowServer framework which allows the vertical integration of workflow-oriented processing based on a scripting approach into any Java application. The framework provides a workflow definition language which integrates with an interpreter for a subset of the Java programming language [10, 11]. This 1approach combines the advantages of workflow-based modeling of business processes with the advantages of a scripting approach for the Java platform. Scripts “glue” together existing software components and do this using Java syntax and semantics with access to the complete Java platform. We have chosen to inte-

[5] Some multiplicities have been left out for reasons of insignificance to the overall architecture.

grate the scripting-based implementation of functionality with the structural definition of processes in the WDL. Alternatively, the WDL could have been designed to only define the structure of processes based on compiled software components (i.e. JavaBeans) implementing the functionality.
It was a design goal to provide a "lightweight" notion of workflow which can be integrated into applications already implementing object persistence, mapping of processes to organizations, and other characteristics of workflow systems [8]. FlowServer is successfully used as basis for a process-based application server using business objects and user interface descriptions provided by a Smalltalk-/Corba-based server in the financial domain.
Future work mainly aims at improving the expressiveness and the ease-of-use of the FlowServer WDL. This includes, for example, a graphical workflow editor which provides a more intuitive interface for creating process definitions than the purely textual WDL. Further research is directed towards integrating the FlowServer framework with distribution-transparent user interface frameworks (e.g. XML-based technologies [1]).

References

1. Bray, T., et. al.: Extensible Markup Language. REC-xml-19980210, W3-Consortium (1998)
2. Eder, J., et. al.: The Workflow Management System Panta Rhei. In Dogac, A., et. al. (eds.): Advances in Workflow Management Systems & Interoperability, Springer (1997)
3. Fujitsu Software Corporation: i-Flow. http://www.i-flow.com
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
5. Gamma, E., et. al.: Eine realistische Applikationsarchitektur für Multi-Tier Java-basierte Clients in der Praxis. In Maffeis, S., Tonniessen, F., Zeidler Ch. (eds.): Erfahrungen mit Java, dPunkt Verlag, 1999 (in german)
6. Fielding, R., et. al.: Hypertext Transfer Protocol 1.1. RFC 2068. W3-Consortium (1997)
7. IBM Corporation: FlowMark. http://www.ibm.com
8. Jablonski, S., Bussler, C.: Workflow Management: Modeling, Concepts, Architecture & Implementation. Thompson International Computer Press (1996)
9. Miller, J., et.al.: CORBA-based Runtime Architectures for WFMS. Journal of Database Management. Special Issues on Multidatabases. Vol. 7. No. 1. (1996)
10. Richter, M. W.: Iava – Yet Another Interpreter for the Java Platform. Submitted for publication in A. Wellings (ed.): Software – Practice & Experience, Wiley
11. Richter, M. W.: The Iava Homepage. http://members.tripod.com/mathias
12. Sankar, S.: The JavaCC Compiler-compiler. http://www.sun.com/suntest/products/JavaCC, http://www.metamata.com/javacc
13. Siegel, J.: Corba Fundamentals and Programming. Wiley (1996)
14. Staffware Inc.: http://www.staffware.com
15. Vogel, A., Rangarao, M.: Programming with EJB, JTS and OTS. Wiley (1999)