Prof. Dr. Tom Gross
http://www.cmlab.de
Weimar, 21/7/2005

Collaboration Bus:
Developer Documentation

Nicolai Marquardt
10599
<firstname>.<lastname>(at)medien.uni-weimar.de

Abstract. Sensor-based infrastructures are often developed from a technical perspective with a strong focus on base technology for sensing information, for processing the captured data, and for adapting the behaviour. An important aspect of these infrastructures is to control the sensor-actuator-connections and to let the users specify their own preferences. In this project we have developed a graphical editor that provides adequate abstractions from base technology, and allows end-users to specify system behaviour by specifying information flows from selected sensors, and sensor data to the envisioned system reactions. The users can furthermore share their own, personal pipeline composition amongst their colleagues or friends, and can also use their shared compositions to create new configurations. In the following developer documentation, we will explain the architecture of the framework and the implementation details. This includes information about the XML data exchange format, the class communication and the structure of the software components.

Table of Contents

1 Introduction

In this first chapter we will give an overview of the *Collaboration Bus* concept, explain some application scenarios and give an introduction for developers (with a short summary of the used libraries in the Java classes).

1.1 Overview of Collaboration Bus

Sensor-based infrastructures are often developed from a technical perspective with a strong focus on base technology for sensing information, for processing the captured data, and for adapting the behaviour. An important aspect of these infrastructures is to control the sensor-actuator-connections and to let the users specify their own preferences. In this project we have developed a graphical editor that provides adequate abstractions from base technology, and allows end-users to specify system behaviour by specifying information flows from selected sensors, and sensor data to the envisioned system reactions.

1.1.1 Pipelines between Sensors, Operators and Actuators

Each connection between sensors, operators and actuators is implemented with pipeline compositions (SensWidget). The user can easily add new of these pipeline compositions using the integrated editor: at first, he can discover the available sensor sources (e.g., movement sensor, temperature, telephone sensor, instant messenger status) of the infrastructure and add them to the pipelines. Then he can specify rules and conditions by adding pipeline components of a set of filters and operators. For each of these processing components, the condition parameters can be selected (e.g., the event value threshold, occurred events counter, period of time, search strings). Finally, the actuators can be specified, to execute reactions at the users computer system or the real environment. Here, the editor provides the option to specify the mapping between the pipeline output and the actuator commands (e.g., display message, activate light source, send email, mute the sound volume).

1.1.2 Personal Repository and Sharing

All the pipeline compositions of the users will be stored in their own personal repository, that can either be local ore remote located on the server. We have

implemented a central interface, to let the users control each pipeline composition, especially to activate, deactivate the composite and start the editor. Furthermore the users can use an integrated sharing mechanism, to provide their own pipeline compositions to other users. Thereby they can decide to provide them the complete pipeline composition, the template of the composition or only the final processing value. In an analogous manner they can add the composition to their own template repository, to build new compositions based on this template. Using this functionality, the users can easily share their pipeline trajectories amongst each other, and can also benefit from the template mechanism.

The *Collaboration Bus* application uses XML serialization to create descriptions of the pipeline composition as well as the complete personal repository. This serialization method is a central part of the flexible instantiation and sharing mechanism, and will be described in detail later in this documentation.

1.1.3 Visualization

Due to the fact that it can be difficult for the user to overlook the composition of a set of sensors, filters and actuators, the user can activate graph visualizations while editing the pipelines. They display relations between incoming and outgoing events of the pipeline in real-time, and let the user easily adjust the pipeline settings while seeing the consequences of his changes at the same time. Furthermore the user can choose between various graph visualizations, to obtain a deeper insight into the pipeline configuration.

1.1.4 Key Features of Collaboration Bus

To summarize the features of the application in just four points, the following aspects describe the main parts of the software tool:

- **Personalized:**
  Instead of using only a few predefined and generic operational networks we use highly personalized sensor-actor-relations in pipeline compositions
- **Overview and control:**
  Let the users control and edit their own created relations
- **Easy-to-use and fast:**
  These sensor-actor-relations have to be defined and executed in just a few minutes, and the software should provide methods for reusing and copying existing compositions
- **Sharing knowledge:**
  For cooperative work, users can share their sensor-actor-relations
  (and the compositions provide an adaptive behaviour)

1.2 Application Scenario

This control and editor software can be used for applications at the private home of the users as well as business areas. The software can easily connect sensors and actuators from remote locations and build a new envisioned application of the user in a few seconds (see Figure 1).

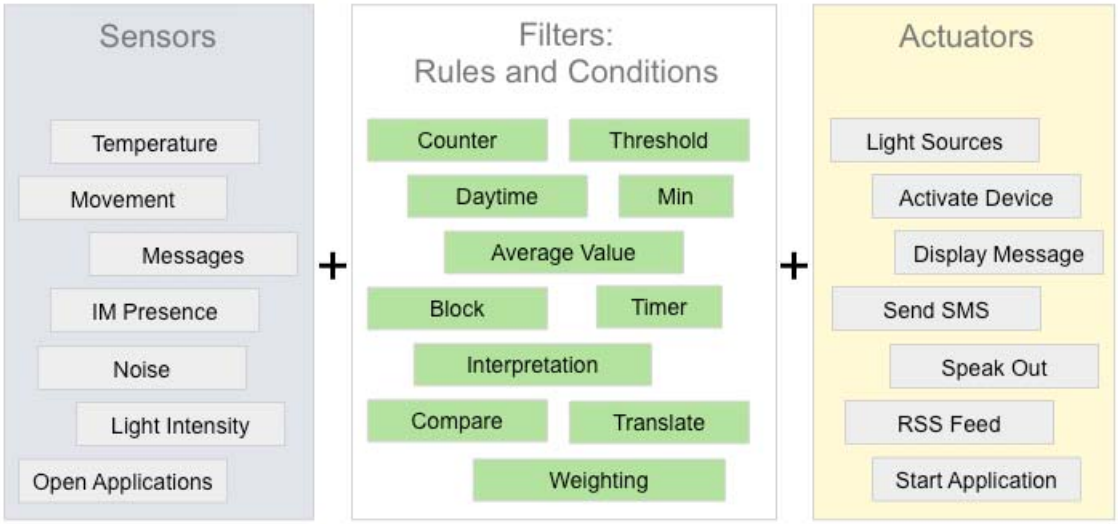

Figure 1. The component categories for the *Collaboration Bus* pipelines: sensors, filters and actuators.

1.2.1 Intelligent Telephone

The user wishes to control the sound volume of his music players and starts his calendar software in dependence of using the office telephone. A simple binary detection sensor of the telephone will be used as the first input source of our pipeline. The second input source checks whether the user is currently logged in at his/her office. In the second step, the condition modules check the telephone sensor state as well as the login information. Finally the user adds the desired actuators: if the pipeline detects that the phone is used, an AppleScript will be activated to mute the volume of the Mac, the ESB infrared control is used to mute the Sound-System, and another AppleScript will finally start the iCal application, so the user can input new appointments during the phone call. When the phone call ends, the application will fade in the music after a few seconds.

1.2.2 Informal Awareness

In this second scenario, the user would like to get informal information about the current activity of his project colleagues and friends. The user can add the PRIMI instant messenger state sensor as the first source of the pipeline, and some further sensors of the Embedded Sensor Boards as additional sensor value sources. Then the user can add the keyword filter to check the PRIMI sensor to match the names of the project colleagues. When the user has finished appending the other filters for the ESB sensors (movement, noise), he/she finally adds the actuators: all events will be collected and displayed as an RSS feed in the personal screensaver. If the message occurrence reaches another threshold, the *Collaboration Bus* system will additionally send the user a SMS via the SMS-Gateway.

1.2.3 CML Lab Info Channel

The users in the two remote located labs of the CML wish to obtain more information about the activities of the members in the other lab. They have the idea of a universal info channel of the lab activities as a RSS feed, that can be integrated in ticker tape displays or as an screensaver visualization (e.g., with Mac OS X). The create a new

pipeline composition, and add the following information sources: the current members logged in at the server, the people in the CML PRIMI chat, the current CVS submits of the programming groups, the current temperature of the two labs, and the average values of the movement and noise sensors in the two labs. As the actuator component they add the RSS feed generator, and publish the RSS file to an internal server location. Now every member of the lab can address this RSS file and add it to his favorite display notification (e.g., the screensaver).

1.2.4 Further Scenarios and Use Cases

The software design of *Collaboration Bus* provides easy-to-use tools to create especially sensor-actuator processing pipelines with 5 up to 15 components. Here is a short list of scenarios, where these pipeline compositions can be used to control the system behaviour:

- "Let me control some devices at home from my mobile phone; or activate them in dependence of the movement activity or temperature"
- "Notify me if at least five of my friends are online and set their state to available in the instant messenger and activate the messenger software"
- "If an email of the CML members arrive, please read out the message headline with the Mac OS X speech synthesizer."
- "Turn off the light sources at home if there was no movement for a longer period of time"
- "Create an ambient display at my desktop for the activity at home"
- "Each time I activate the television dim the light sources around"
- "If three important email messages arrive, please send me a SMS notification"
- "If the temperature at home is below 15 degrees, and it is after 6 p.m. please activate the heating and send me notification of the temperature after 15 minutes"
- "Please notify me when Tareg and Christoph are both available for a chat in the instant messenger or if they are in their office and not at the telephone or in a meeting"
- "Mute the audio volume if someone opens the office door"
- "If the temperature is above 29 degrees Celsius, then activate the fan"
- "If there are more than 20 CVS submits of the CBUS project, and it is after 8 p.m. then please send me a short summary as email message"

These are just a few examples to illustrate the variations of sensor-actuator relations that can be defined to create a sensor-based system that acts to the needs of the users. All these examples can be composed with the *Collaboration Bus* editor with no efforts.

1.3 Development Prerequisites

The *Collaboration Bus* Framework has been developed for Java (Version 1.4.2) under the Mac OS X 10.4 environment. Some parts of the software can only executed on Mac OS X, these are especially the OS notifications (with Growl), the AppleScript executions, and the Quaqua user interface toolkit [Randelshofer 2005a].

For the development in Java we used the Eclipse IDE, Version 3.1 M6. All source files, libraries and the image, XML and property files are located in the CVS remote

directory “CBUS” of the Cooperative Media Lab CVS server and in the `Collaboration_Bus_Sources.zip` file in the BSCW. To compile and distribute the project you can use the integrated build process of eclipse as well as the `build.xml` file in the project root directory.

1.4 Libraries

In the Java classes of the project we used libraries for handling XML data files, XML-RPC connections, RSS parsing and writing, XML serialization, graph libraries and some Mac OS X related functionality. All libraries are included in the `/lib/` directory of the project root, and have to be added to the Java classpath.

In the following list we will give a short overview of all included libraries, and further details of the application of these libraries will follow in the implementation chapters:

- **JFreeChart 1.0.0 rc1**
  With this library you can create various chart diagrams, with static as well as dynamically added data items. We used this library for the time plot charts of the sensor values, the occurrence of events and for the local and global event counter. [Jfree 2005b]
- **Jaxen 1.1 Beta**
  XML parser library, needed with the XML-API to create XPath expressions to address specified nodes in the XML file. [Codehaus 2005a]
- **JCommon 1.0.0 rc1**
  Class library needed for `JFreeChart` and provides methods for the GUI layout, XML parsing, text utilities, etc. [Jfree 2005a]
- **JDOM 1.0**
  This library enables the XML parsing based on the Document Object Model and is required for the several XML parser API. [JDOM 2005]
- **Quaqua 3.1.1**
  Library that provides special functionality for Mac OS X. This includes layout managers, striped tables and lists, browser panels, special design for buttons and other Swing components. [Randelshofer 2005a], Documentation: [Randelshofer 2005b]
- **OpenScripting**
  Library to send AppleScript commands to the operating system from within the Java software. Also needed for the integrated growl notification methods. The growl notification system for Mac OS X includes a class file with Java binding [Forsythe et al. 2005]
- **RSS4J 0.91**
  To read and write the XML files for RSS feeds, this library includes some methods to provide an easy access. The RSS feeds will be used in the RSS actuator. [ChurchillObjects 2005]

- **TableLayout**
  Provides similar layout like the gridbag layout, but is still more comfortable to instantiate. Just pass the desired dimensions of the layout table to the constructor, and address the row and column in the add method for components. [Barbalace 2005]
- **XStream 1.1.2**
  For the serialization of various objects of the framework we used the XStream XML serialization methods. Any members of the classes will be stored in the XML file (this includes primitive data types, vectors, hashtables and complex objects) and the library can instantiate an object of the original class again. No getters and setters for the members of the class are required. [Codehaus 2005b]
- **Xerces and XML-API 2.7.0**
  XML parser of the Apache XML project, used with the JDOM library [Apache 2005a]
- **XML-RPC 1.1**
  The connection to the Sens-ation server instance is established via the XML-RPC gateway. The classes of the library can instantiate the XML-RPC client as well as a new XML-RPC web server. [Apache 2005b]
- **XOM 1.1b1**
  Tree-based API for processing XML documents in Java [Harold 2005]

2 Collaboration Bus Framework

In the following section we explain the architecture of the *Collaboration Bus* software and we will give a general overview of the structure. Furthermore we explain the implementation details of the main processing components.

2.1 Overview of the Software Architecture

The *Collaboration Bus* software includes several components and user interfaces. The main software components are the classes for the user repository and pipeline compositions. In the instances of these classes the application can build the representations of all the pipeline compositions of the user. These two classes build the model of the MVC structure for the personal user settings.

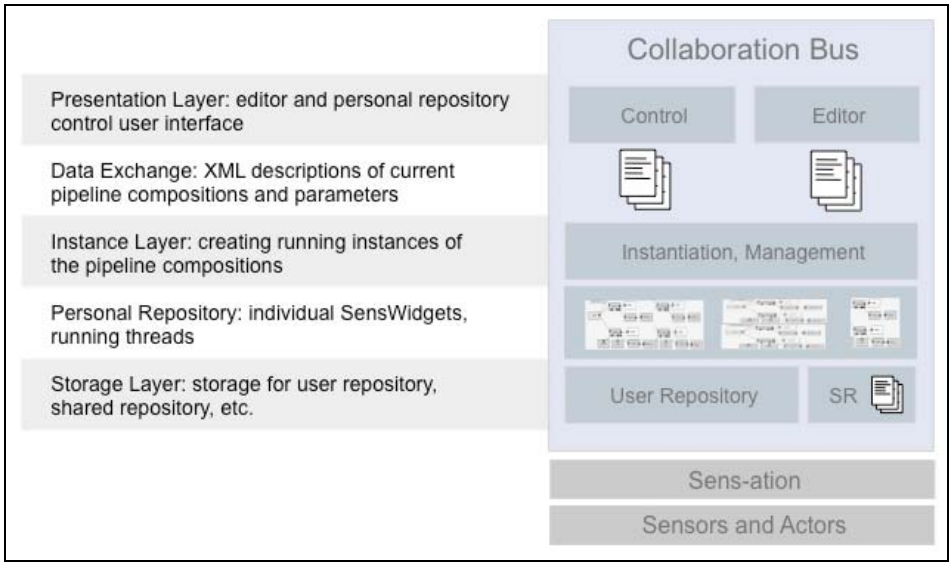

Figure 2. The application layers of *Collaboration Bus*.

The classes of the Control GUI are responsible for displaying the personal repository of the user and to provide access to the application tools to change the composition settings. The Editor GUI instances are started from within the Control class, and they represent the view/controller for the component model (`PersonalRepository` and `ProcessingContainer`).

All components from within a single pipeline composition are derived from the `AbstractComponent` class, and the concrete classes are the sources, filter and actuators. The can be connected via flexible pipelines to transfer the events from one component to its subsequent component.

The visualization classes are instantiated from the editor component to enable the deeper insight for the user into the pipeline event transmission. The editor starts the desired visualization class, and registers this class as the global observer for all pipeline events at the `ProcessingContainer`.

The remote repository server classes are the server side part of the control/editor user interfaces. The client application can connect to the remote repository server to authenticate a user, load and save his personal repository and the shared repository as well. The shared repository contains all the pipeline compositions shared by the users of the system, and is also serialized to XML data.

Furthermore there exist some utility classes with capsulated helper methods, especially a factory class with static methods for the creation of GUI elements, or a class for string separation and search methods.

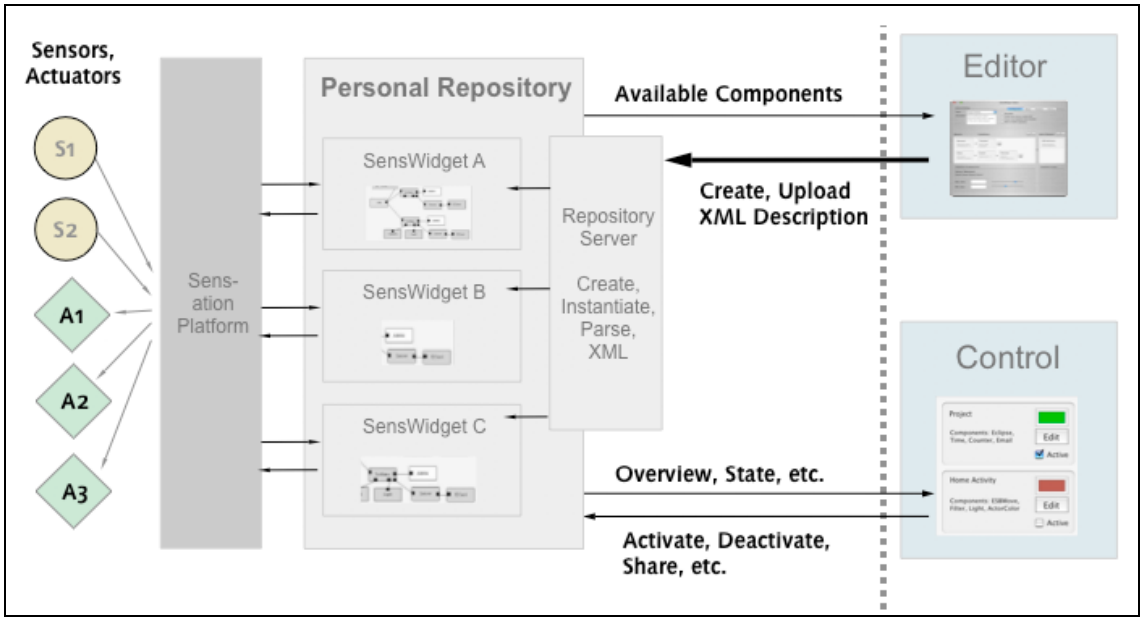

Figure 3. Overview of the *Collaboration Bus* application, with the personal repository as the central storage of all pipeline compositions.

2.2 Container Components

The container components can handle the pipeline compositions (`ProcessingContainer`) as well as the personal repository of a user (`PersonalRepository`). These components contain the single components of a collection, and provide various methods to add and remove components, and to copy and share them as well.

2.2.1 Processing Container

The `ProcessingContainer` object contains all processing components and the connected pipelines of a pipeline composition. The various components (sources, filters and sinks/actuators) will be stored in the hashtable components, and with their unique `componentID` as the key. The method `generateID()` can create these unique IDs for the components. The class provides methods to add components (and to connect these components with the needed pipelines) and to remove components, where the remaining components will be restructured and the pipeline network will be rebuilt to close the processing path (`removeComponent(ID)`). The `initPipeConnections()` initializes the specified pipe connections between components, and calls the register method at the source component for the given sink component of the pipeline.

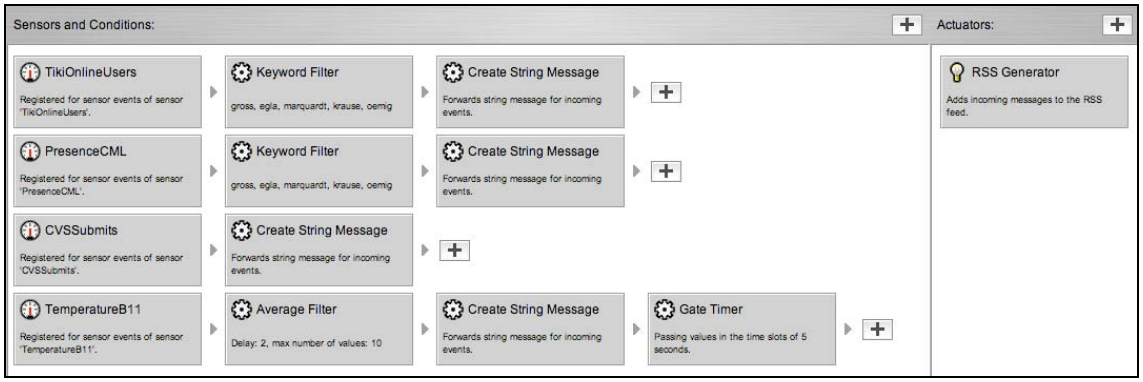

Figure 4. Pipeline composition of a processing container (RSS info channel).

To start and stop the pipeline composition inside the `ProcessingContainer`, the `start()` method iterates through all registered components to activate the components threads. If interface components would like to receive all the events of the active pipeline, they can register at the `ProcessingContainer` as a global event observer, and will receive all the occurred events in the processing pipeline. The components of the `ProcessingContainer` calls the `notifyForward(String componentID, PipeValue pipeValue)` method, and the method iterates through all registered observer to send them the occurred event (with `componentID` as identification, and the forwarded `pipeValue` as the event value).

To support the sharing mechanism with abstract templates, the class provides the `reset()` method, to call the explicit components `reset()` method of all sensor sources, and to remove the actuator elements and their pipeline connections. These template containers can be shared without the source sensor and actuator information.

2.2.2 Personal Repository

The `PersonalRepository` object contains all the processing containers (with the single pipeline compositions) of a user. It stores all the containers in a vector as instances of `ProcessingContainer`. It includes methods to create new containers in the repository (`addContainer(ProcessingContainer)`) with an integrated mechanism to solve naming conflicts of the components, and provides also methods to remove containers. With the method `stopAll()` the repository can stop the active processing pipelines of all stored `ProcessingContainers`; this is a method to provide the thread-safe and secure interruption of all pipelines with a command from the `Control` user interface class.

Figure 5 illustrates the nested structure of the container components: The main container is the `PersonalRepository`, and it includes the various `ProcessingContainers`. Inside of the `ProcessingContainers` are the pipeline compositions with the sensor source, filter and actuator components.

The `PersonalRepository` will be serialized to XML data with the `XMLUtility` method toXML(ProcessingContainer), and is the large-scale storage of all the user related configurations. In the serialized XML string form, it can be send to the repository server instance, and is then part of the user entry at the server.

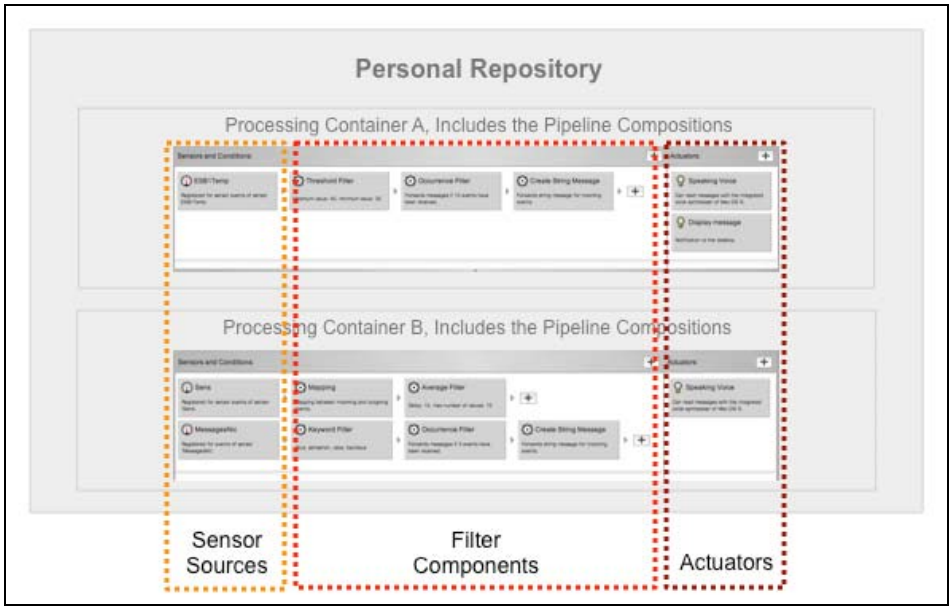

Figure 5. The structure of the personal repository and the processing container.

3 Pipeline Components

The `AbstractComponent` is the base class for all three types of components in a pipeline composition. All sources, filters and actuator components are derived from this base class. The class implements the basic functionality, which all components need so that they can be edited and controlled with the `Control` and Editor `GUI` classes. The derived subclasses will be stored in `ProcessingContainer` objects, and all components obtain their unique `componentID`. The type of a component can either be `AbstractComponent.TYPE_SOURCE`, `AbstractComponent.TYPE_SINK` or `AbstractComponent.TYPE_FILTER`. This type of the component is applied during the constructor call of the base class.

Each component has its own queue with the incoming values of the predecessor component (only the sources do not need this event queue). With the sinks vector, the component can manage its own vector with references to the successor components. Each time, when the component has to pass an event to the successor, it can call the `forward(PipeValue)` method of the `AbstractComponent` class, and the method will iterate through the sinks vector and calls the `push(PipeValue)` method of each of the subsequent components. The successor components will add the incoming `PipeValue` to the event queue, and the `run()` method of the thread can dequeue the value when the component has finished the processing of the former `PipeValues` in the event queue. The `run()`, `start()` and `stop()` methods are implemented in the base class, and only for the `run()` method it is necessary to overwrite the method in the derived classes. All components will be started with the `start()` method from the processing container, and the processing container interrupts the components as well.

3.1 Sensor Sources

The source components are responsible for the forwarding of incoming sensor events of the registered `sensorID`. The class has the reference to the singleton instance of the `SensorRegistry` as private member, and if the `start()` method is called, the module forwards the registration to the `SensorRegistry`, that is connected to the XML-RPC server. The `SensorRegistry` is then again registered at the *Sens-ation* server to get the event notification of the sensors, of which the components of the *Collaboration Bus* software are interested in. If a new event of the sensor occurred, the *Sens-ation* server sends the event to the `SensorRegistry`, this object forwards this event to all registered observers, and finally the observer sensor source object pushes this value into the pipeline to the next subsequent component.

The registration at the *Sens-ation* server instance is implemented with calling the callback register method of the XML-RPC gateway. With calling this method, the `SensorRegistry` module transmits the current IP and port number for the call back method. This callback notification method is only available at the Sens-ation server, if the client uses a static IP address and is not behind a firewall technology or registered in an internal subnet.

3.2 Filters and Processing

The filter components of the processing pipelines can be used to specify the personal preferences of the interpretation, manipulation or filtering of the incoming source data events. Each of these filter components has explicit operator functions, e.g. to calculate the average value, to search for keywords or to generate string messages from incoming numeric values. These components can be assembled in any combination, because of the generic exchange format between pipeline elements with the `PipeValue` object.

All filter components are derived from the `AbstractComponent` class, which provides the basic functionality of the pipeline components. Each filter has to implement the abstract `run()` method, that will be executed when the `ProcessingContainer` starts all the components in a pipeline. The `run()` method contains the processing loop (while condition until interrupted is set to true) that will check if the `EventQueue` has new elements, and if so, it dequeues the `PipeValue` object and extracts the current event. This is the core of the processing step, because now the object can process this event, to finally send the processed event to the subsequent component in the pipeline with the `forward()` method. Each filter component also has the own `start()` and `stop()` method, and this can be used for the initialization of objects before the thread will execute the `run()` method (or at the end when the thread is interrupted).

Since each of the processing filters has own private members, the filter has to provide getter and setter methods for the access of the interface wrapper in the `DrawComponent` method of the editor. In general, each filter component can receive any input data format; nevertheless the component has to decide, if it can process the incoming data. The adaptive behaviour is realized with the forwarding of string events, that will be parsed each time a filter need the data in the integer or double format. With this method, you can couple an average filter component with a sensor that produces text messages, because if the source sends strings with numeric values, the pipeline can

still interpret these values. But as we can easily see: if the source sensor sends text strings with characters, an successor `AverageFilter` component can not handle these events. In this case the filter component still ignores the incoming values.

3.2.1 Average Filter

The `AverageFilter` can process any numeric data input events (integer, double), and can be used to interpolate the incoming event stream. The filter stores a vector of past incoming values, and can calculate the average of these numeric values to forward it to the next component in the pipeline. The parameters to define in this filter is at first, how many values the filter has to use for that calculation, and as a second parameter, if the filter has to wait for a delay time, until he forwards the calculated values (this means, that the filter waits, until he has received the specified number of values until he forwards the average).

The calculation of this filter component can for example interpolate the oscillating values of a movement or noise sensor source, and eliminates all outlier values. The result is a straightened event value series, with a tolerable deviation form the original values.

3.2.2 Gate Timer Filter

If the user does not want to forward any incoming event of the pipeline, he can decide to use the `GateTimerFilter`. With this filter component the user specifies the interval time that the component blocks incoming events after it has passed one event. For example, if the `timeInSeconds` parameter is set to 5, then the filter component waits five seconds until it received the first incoming event. The first event will be forwarded to the successor, and the filter blocks all further incoming events until the time has passed. The component starts a reminder thread that will deactivate the blocking tag of the filter after the specified time (here: 5 seconds). Then the filter will wait for new incoming events again.

3.2.3 Keyword Filter

If the user wants to search in the incoming events of a pipeline for strings, the `KeywordFilter` provides this functionality. The user specifies the keywords to search for, they will be stored in the private `keyWords` vector, and if new incoming events occurred, the filter searches in the event string for the specified keywords.

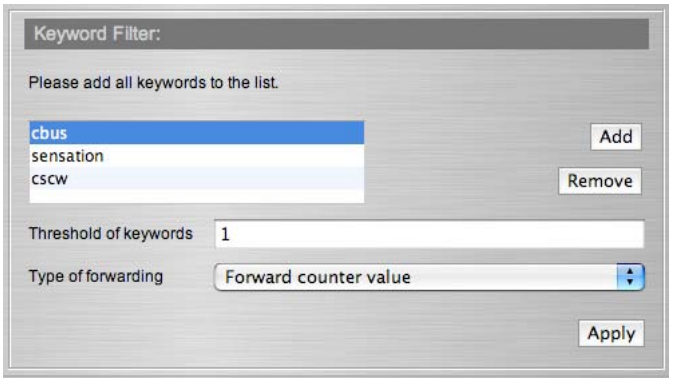

Figure 6. The keyword filter control panel.

To specify what to do if the filter finds keywords, the `forwardType` member can be set to one of the following constants: `KeywordFilter.FORWARDTYPE_COUNTER` to forward the number of keywords found, `KeywordFilter.FORWARDTYPE_LASTEVENT` to forward the original incoming event, and `KeywordFilter.FORWARDTYPE_FOUNDEVENTS` to forward the list of all found keywords (as a comma separated list). Furthermore the user can set the `occurrence` member to the value of keywords the filter has to find to forward the message (e.g., if the occurrence member is set to 3, the filter will forward the message if it finds at least three keywords in the incoming message).

In the most cases, this filter component is used to find keywords in incoming string messages of a email sensor or instant messaging sensor, e.g. to find the messages related to a special project or sent by a specified person.

3.2.4 Occurrence Filter

The `OccurrenceFilter` is a simple filter that counts the incoming events and only forwards the event if the number of incoming events has reached the value of the `counterMax` member. Then the filter forwards the last of the incoming events, with the saved values that arrived in the meantime as the optional history values of the `PipeValue`. These optional history values are sometimes used at the later successor components to find out the calculation steps of the predecessor filter components.

3.2.5 String Generator

This filter is helpful to create new string messages to forward in the pipeline, and to embed the incoming values of the `PipeEvent` in this string. To create a new message string, the user can type in the text of the message, and the insert the `$$` placeholder for the current event value, and the `$sensor$` placeholder for the initial `sensorID` of the source sensor. If new events are forwarded to the filter, it replaces the placeholder and forwards the event to the successor.

3.2.6 Threshold Filter

This filter can decide if the incoming numeric events are in between the specified limits of lower and upper threshold. If the event value is between the maximum and the minimum limit, then the filter forwards the original event without modifications. As all the filter components that handles numeric values, this filter tries to parse the string values to either integer or double values; all event values that can not parsed to one of these numeric data types are ignored.

3.2.7 Translation Mapping

This component can be used for the mapping of incoming event values to any outgoing event value. The component contains a hash table for all the input-output pairs that are available for the translation. If incoming keywords are found in the hash table, then the filter will forward the related "output" value into the pipeline.

These pairs of values represent the simple rule of *IF...THEN...* conditions (e.g., if the input of the binary telephone sensor is "true" then forward the value "telephone is used", or another example: if the input of the movement sensor is "10" then forward the

command for the relay board "on 3" to activate the third relay port). If there has to be forwarded an event for incoming values that are not in the list, then just specify an entry with the keyword `$other$` as incoming value and set the related outgoing event.

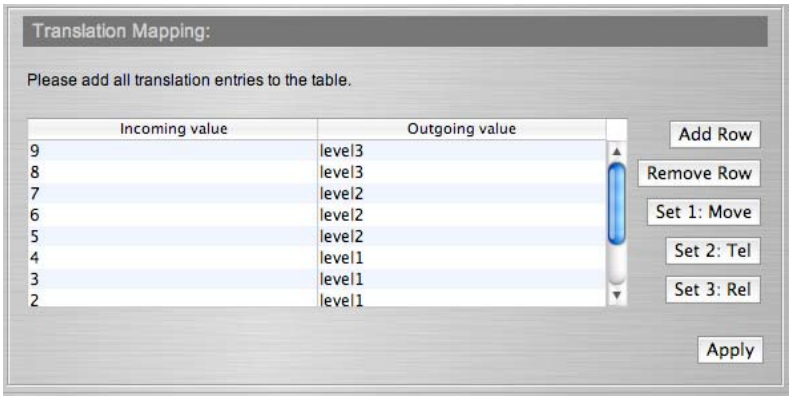

Figure 7. View and control interface of the translation mapping component.

For the user interface visualization and input of the mapping table, the `DrawFilter` class can instantiate an object of the class `DrawTableList`. This object encapsulates the mapping functionality of the table and allows adding and removing pair entries. There are also three mapping presets (e.g., for the relay board activation and deactivation commands) that the user can insert and modify the execution commands.

3.3 Sinks and Actuators

The actuators are the sink components of each pipeline composition. They can execute software, send messages or control hardware, and therefore act as the notification or action interface to the user.

3.3.1 Colour Panel

This actuator displays a colour panel in a new window that can fade between three levels that will be displayed as different colours of the panel. The commands for the actuator are `"level1"`, `"level2"` and `"level3"` to set the current colour level. If the actuator receives new level information, the panel starts the cross fade method to change to the new colour.

3.3.2 Growl Notifications

Growl is a global notification system for Mac OS X (http://www.growl.info), and this actuator uses the Java binding of this notification system to display messages to the screen of the user. The actuator accepts all the incoming messages as string value and sends these string values to the `ActuatorGrowl` class that encapsulates the communication with the growl Java binding class. This class `com.growl.Growl` finally transfers these messages to the Growl notification service that displays the message to the screen. These notification of the global system growl software stay only a few seconds in one of the corners of the screen and fade out automatically.

3.3.3 Relay Board

The remote connected relay interface board (8 binary channels) can be controlled with the `SinkRelayInterface` class. It is possible to activate and deactivate each of the eight ports of the relay board. The actuator receives commands with the following pattern: `<on|off> <port-number-1> ... <port-number-n>` (where the port numbers identify the ports to activate or deactivate). The `run()` method of the actuator parses the incoming command and calls the `command(String command, String ports)` method to establish a new connection to the remote located XML-RPC server for the relay board control. If the Boolean fallback tag of the actuator is set to true, then the actuator will send the command via `the command(String command, String ports, int time)` method, with the value of `fallbackTime` member as time parameter. After this period of time (in seconds) the relay board will deactivate the port.

3.3.4 RSS Feed

RSS feeds can be read with many web browsers, news feed readers and displayed as screensaver (as seen in Mac OS X), so that this actuator element can create info data that can be displayed with all of these example software programs. This class opens the RSS feed (specified in the `RSSFile` member) and adds or changes the news feed entries in this file. The actuator parses the incoming messages, and interprets the first part of the message as the headline, and the second part of the message as body. The dividing symbol between headline and body text is the semicolon. If the actuator finds existing messages with the same headline text, then the existing body of the entry will be changes (instead of adding a new message).

3.3.5 Sound Control

This actuator component uses the AppleScript scripting language of Mac OS X to control the sound volume of the operating system. The component receives the messages "mute on" and "mute off" to disable the sound speakers and to enable the speakers again.

3.3.6 SMS Notifications

The intention of this gateway is, to let the *Collaboration Bus* system send the user information messages to their mobile phone. This component displays the incoming messages to the screen, and the destination number can be set with the `number` member of the object. The message could be send to the remote SMS gateway via XML-RPC, but this feature is still under construction, because of difficult implementation of the SMS gateway at the Mac OS X system.

3.3.7 Synthetic Speech

This actuator can use the integrated voice synthesizer of Mac OS X to read out the incoming messages. One of the existing voices provided of Mac OS X speech synthesizer can be selected (and a dialog of the available voices will be displayed with the GUI wrapper of this component). Any incoming string message of the predecessor component will be send to the voice synthesizer via an AppleScript command.

3.3.8 Start Mac OS X Application

This is again an actuator that uses AppleScript, this time to start an application and display the main window. There are two possible ways of choosing the application to start: one method is to specify the application in the member variable `application`; the other method is to activate the Boolean flag `useSpecifiedAppInValue`, and to transmit the string of the application as the incoming event value. With these two methods the component can either react to only valid incoming application identifiers, or simply react to any incoming event that passed the pipeline composition.

3.4 Pipe Values

Instances of the `PipeValue` class represent the values for transmissions between pipeline components. The `PipeValue` can be initialized with a `SensorValue` (these are the *Sens-ation* objects for transmission of sensor events) object or with a specified `sensorID` and the event as a string value. For each `PipeValue` the `sourceComponent` has to be specified, this is for the later components in the pipeline to have the possibility to address the source of an event in the pipeline.

Each `PipeValue` object can save a list of history events in the values vector, so that not only the current value will be passed to the next processing component, but also the history of processed values. The successor components can decide if they need these additional values, but most of the implemented filter components only need the current event value for their processing.

3.5 Dynamic Instantiation

The `ComponentFactory` provides a static method to instantiate filter and actuator objects that are inherited from the `AbstractComponent` class. The method `createComponent()` gets the class name as string parameter and instantiates a new object of the specified class. This new class object will then reassigned to the base class object `AbstractComponent`. This object will be returned from the method, and can then be added to the `ProcessingContainer` instance.

The `ComponentDescriptions` are used to register filter and actuator components to the `ActuatorHandler` and `ComponentHandler`. Each of the available components of the platform has to be registered with a complete instance of this `ComponentDescriptions`. The member values describes the component:

- `name`: the name of the actuator or filter
- `description`: to explain the functionality of the component; will be displayed at the component select panels of the GUI
- `className`: the class name of the component as string; this name is used for the dynamic object instantiation of the `ComponentFactory` instance
- `dataIn`: specify the input of the component (e.g., numeric, string)
- `dataOut`: specify the output of the component
- `parameterEntryList`: a vector with all registered parameters of the component; these are instances of the `ParameterEntry` object

3.6 Remote Connections

To establish the connection between the remote repository server and the client GUI, the *Collaboration Bus* `Control` window creates the singleton instance of the `RemoteConnection` object. The `connect(server, port)` method gets the server and port specified in the connection tab of the Control window. It opens a new socket connection, and provides the access methods to this open connection to all classes that obtain a reference to the singleton instance. These access methods include:

- `login(name, password)`: sends the username and the password to the server and tries to login
- `disconnect()`: ends the current user connection and closes the socket
- `getRepository()`: get the current repository of the user as serialized XML. This string data is returned, and has to be parsed to the `PersonalRepository` object with the `XMLUtility` method `fromXML()`
- `setRepository(data)`: upload of the current user repository; serialized to XML. This personal description is saved in the server "as it is" and is whether parsed or interpreted. This allows very fast response times of the server
- `getSharedRepository()`: receives the current entries in the shared repository of the remote server. This string also has to be parsed via the `XMLUtility` class to the `SharedRepository` object
- `addToSharedRepository(data)`: adds the processing container to the shared repository entries of the remote server

4 GUI

The *Collaboration Bus* framework provides three main interface views: the control interface for the overview of the current personal repository of the user, the editor interface to change the configuration of the pipeline composition, and the graph visualizations to display the data flow in the pipelines in real-time.

4.1 Control User Interface

The `Control` java class instantiates the main control interface for *the Collaboration Bus* framework. With the control interface the user can create new pipeline compositions, start and stop existing ones as well as open the editor window for an existing pipeline to change the preferences and component arrangement.

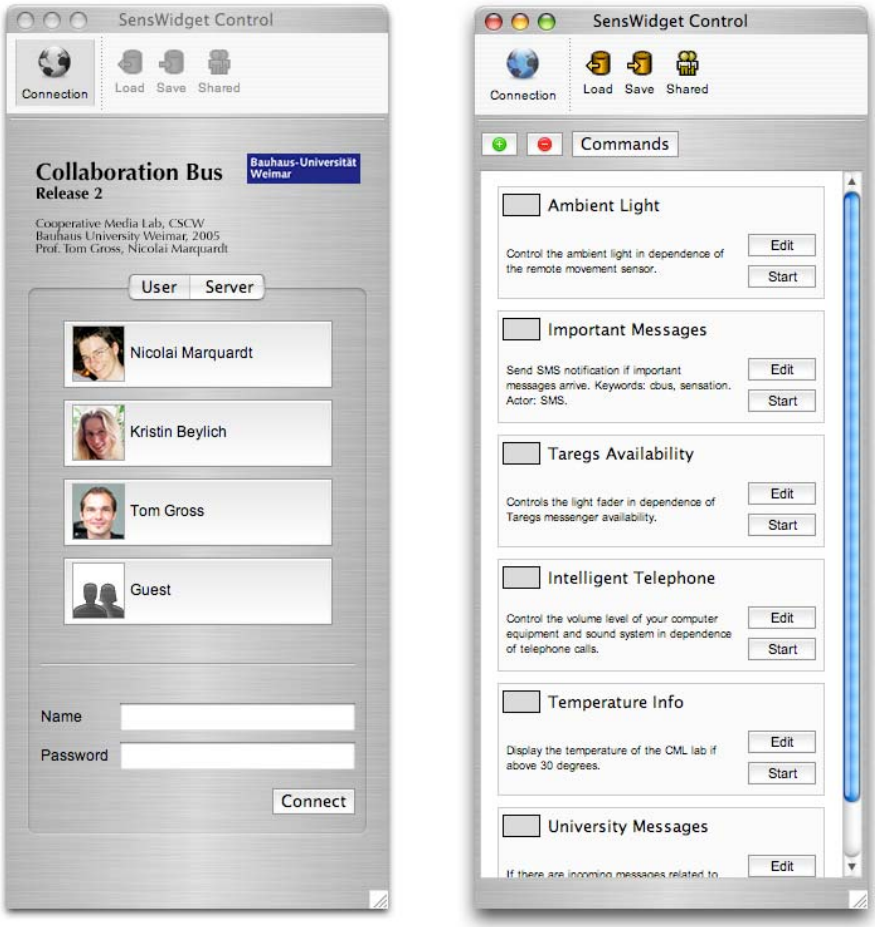

Figure 8. Login screen of the control GUI and the personal repository view.

The control interface starts with the login screen for the user account at the repository server, and the user can select his user name, the password and specify the remote connection settings. By default, the control interface will connect to the local `RepositoryServer` instance. The second connection is to the *Sens-ation* server instance, where the application can retrieve the current registered sensors, the values and the location information.

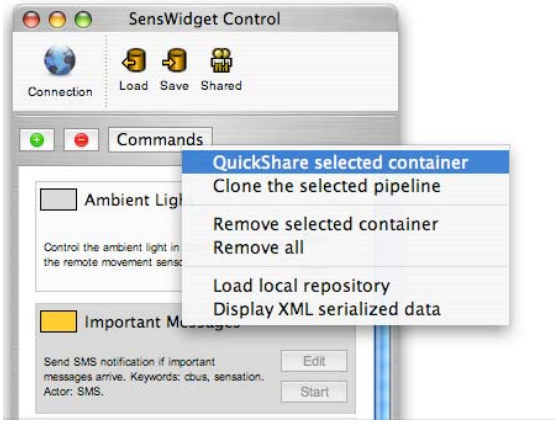

Figure 9. Context menu of the processing container view: share, clone and remove containers.

If the users connects to the server instances, the `Control` class tries to login with the given username and password, and if the connection was successful, the repository server sends the personal repository of the user as serialized XML string. The Control class creates the `PersonalRepository` class with the `XMLUtility` helper method (using the XStream library [Codehaus 2005b]) and changes the interface appearance to the repository view of the user. Each of the processing containers in the personal repository of the user will be added to the flow panel, and the `Control` class also adds the three command buttons to the top of the panel: the add and remove button as well as the command button, with menu entries for quick sharing, to clear the complete personal repository and the XML source view window (see Figure 9). For each of the processing containers in the personal repository, the `Control` class instantiates a `DrawProcessingContainer` object, to display the container information and provide the two `JButtons` to start the editor window (and set the container state to "edited") and to start or stop the container pipelines. If the processing container is active, the colour panel of the `DrawProcessingContainer` will turn to green, when edited it will turn to orange.

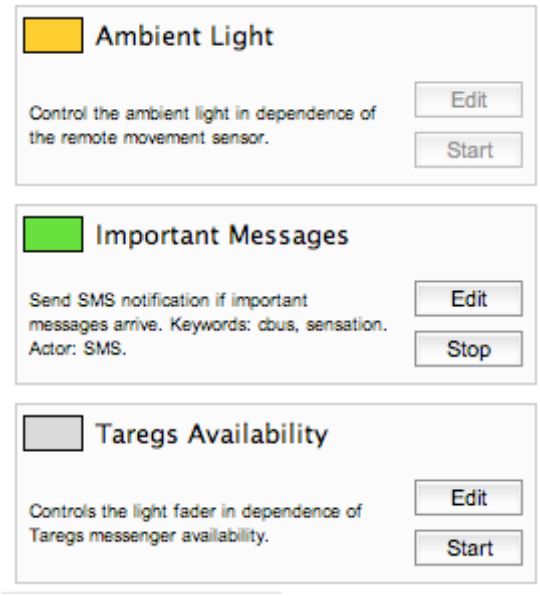

Figure 10. `ProcessingContainers`: in *edit* mode (orange), active (green) and inactive (grey).

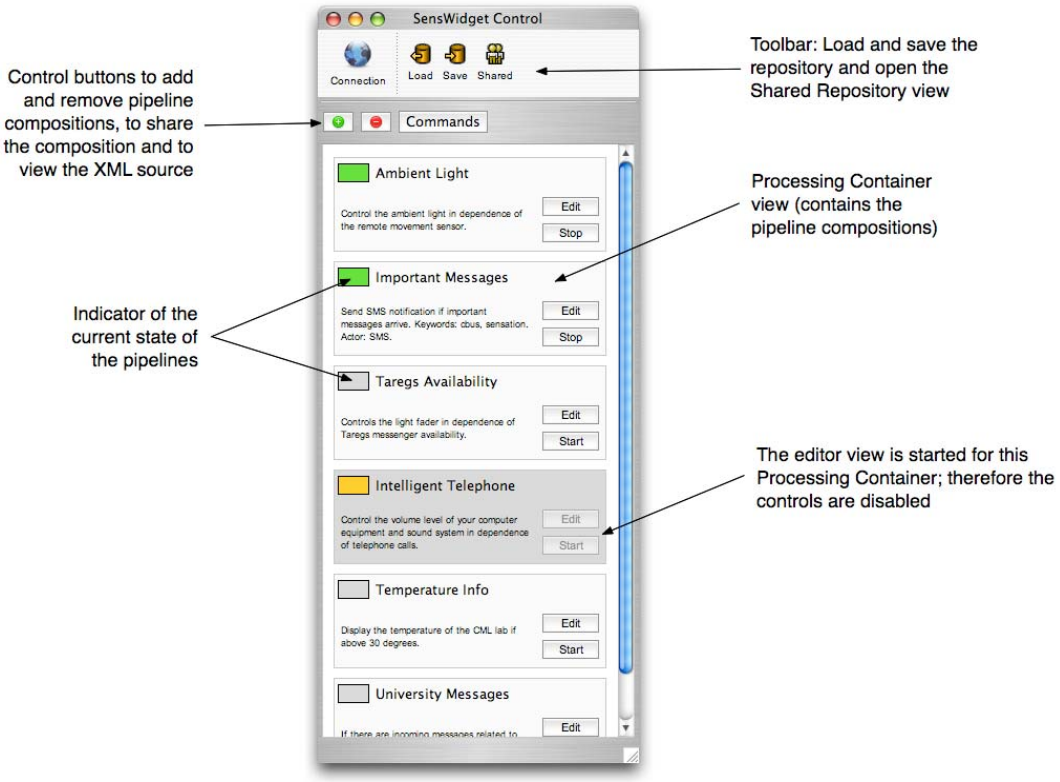

Figure 11. The elements of the personal repository view.

The `Control` class includes various methods to initialize and change the GUI components and view. The `init()` method initializes the main window, with all `JPanels` for the user login, the specification of the repository server and *the Sensation* server and the view of the processing containers. The main control buttons are arranged in a `JToolBar` on the top of the `JFrame`, and it includes the toggle button for switching between the connection view and the personal repository view. The next two buttons can be used to load (`readPersonalRepositoryFromServer()`) the XML data of the personal repository from the server, and to write the data to the server (`writePersonalRepositoryToServer()`). The last button opens the window with the current entries of the shared repository. This window will be explained later in this chapter.

The class provides several methods for the handling of the complete repository as well as the single processing components. New processing containers will be added with the methods `addProcessingContainer(ProcessingContainer pc)` and `addEmptyProcessingContainer()`. The first method is e.g. called from the `Assistant` GUI dialog, to create a new pipeline with the selected components in the dialog window. The second method creates a new empty container in the personal repository, containing no sensor sources or actuators. To clone one of the existing pipeline compositions, the method `clonePipeline()` can create a new processing component that contains the same components (sensors, filters, actuators) as the current selected composition.

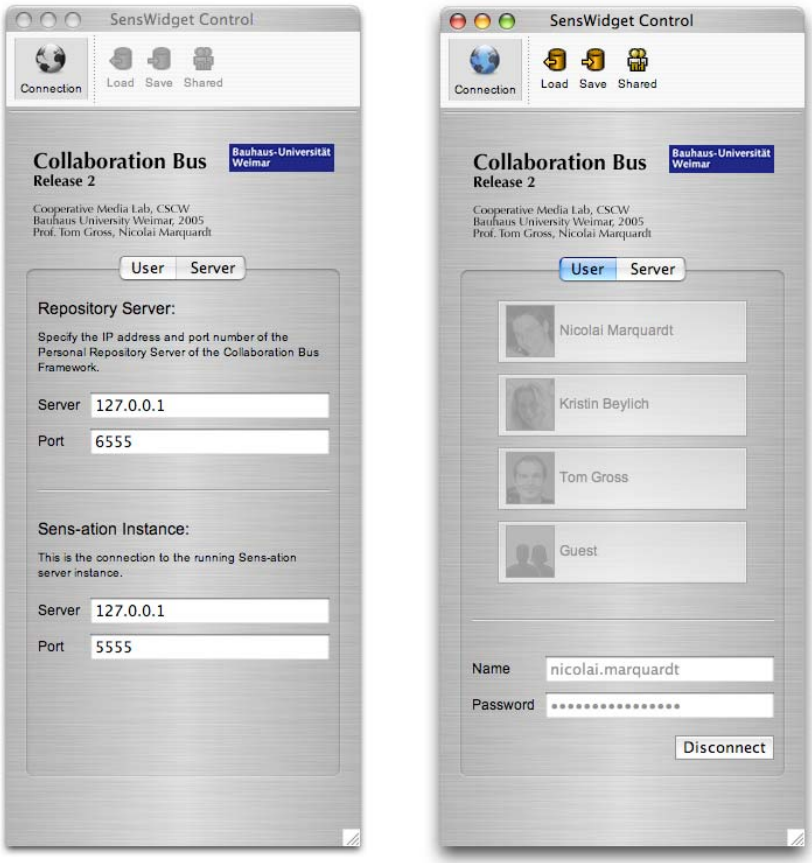

Figure 12. Preference panel for the server connection and the disabled login window.

The Control interface also includes the methods for accessing the current shared repository of the server. Using the `getSharedRepository()` method, the `SharedRepositoryTable` class will connect to the repository server and receives the shared repository entries (this will be explained later in this document). Using the `addToSharedRepository()` method, the current selected processing container can directly added to the shared repository (using the default settings for sharing) and will be registered at the repository server. This method is implemented as a "Quick Shared" method, because no optional settings are specified, as they are in the Editor window in the Sharing tab. This special view for the sharing mechanism is explained in the next section about the Editor window.

4.2 Editor User Interface

The Editor is responsible for the user interface to change the pipeline composition, to change the preferences and to publish compositions to the shared repository. The instance of the editor is created from within the Control class (from the nested `DrawProcessingContainer`) and is responsible for one processing container (the `DrawProcessingContainer` object is the reference parameter of the constructor and will be accessed to retrieve the reference to the `ProcessingContainer`).

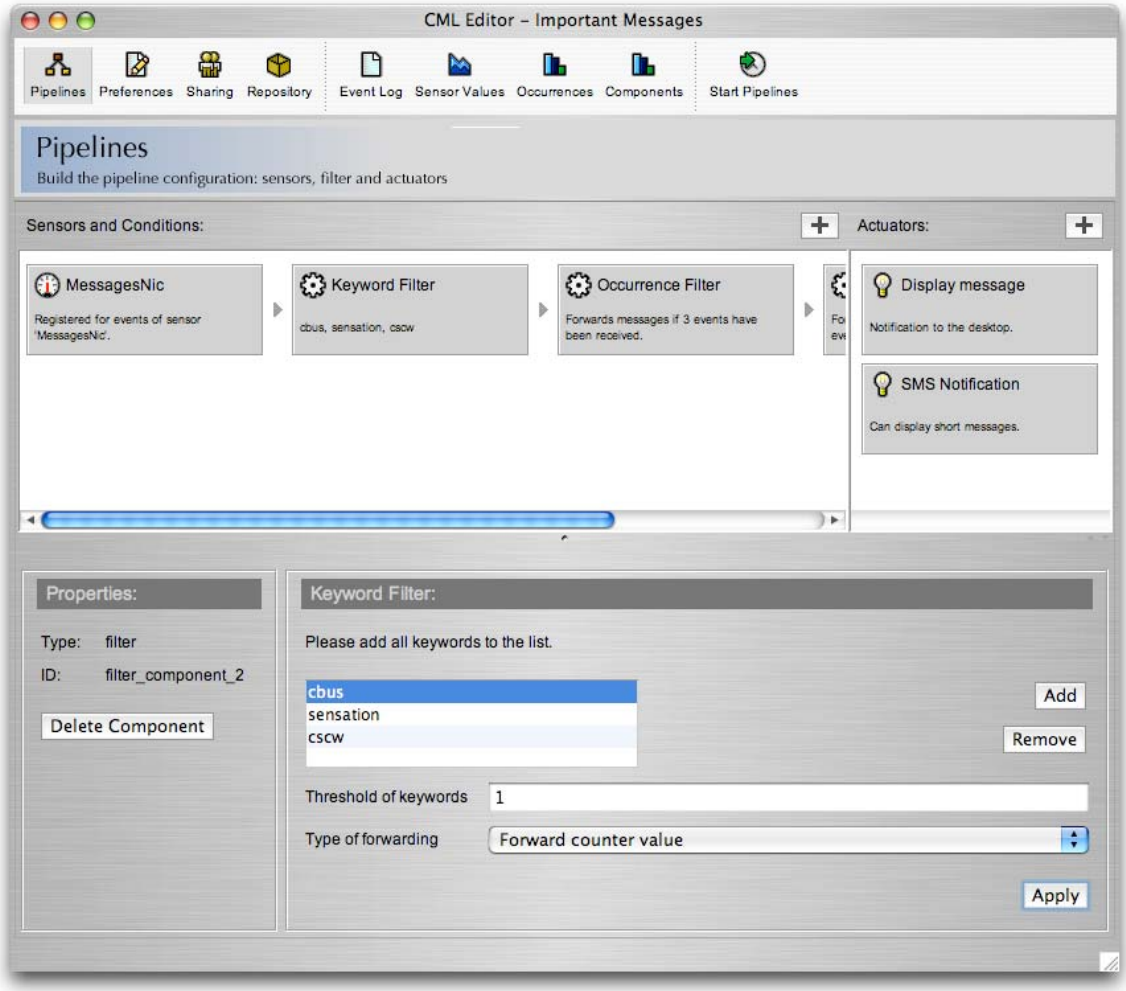

Figure 13. The editor main window: create new pipeline compositions and set the preferences of components.

In the initialization phase of the constructor, the editor creates the various GUI elements for the user interaction. On the top of the window, a toolbar with buttons for the main methods of the editor is inserted with the `buildToolbar()` method. The first four buttons can select the current view of the editor: Pipelines, Configuration, Sharing and Repository.

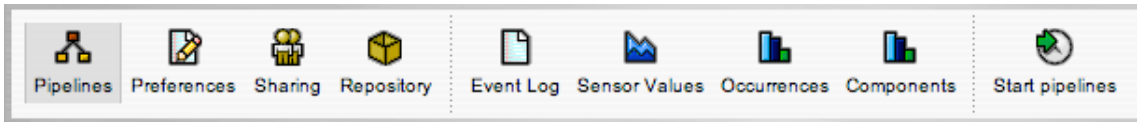

Figure 14. Editor toolbar: the four view selectors on the left side and the buttons for the visualization on the right side.

The pipeline view is initialized with the `buildPanelPipelines()` method, and contains three main parts: in the upper left part is the configuration panel of the source sensors and the filters and operator components, in the upper right part is the panel for the actuator elements. Below these two areas is the configuration view of the current selected component: the displayed components will be changed in dependence of the selected component. Responsible for the view are the `DrawComponent`,

`DrawSensor` and `DrawActuator` classes that wrap the interface descriptions for each of the registered components.

The configuration panel is initialized with the `buildPanelConfiguration()` and provides interface elements for the configuration of properties of the processing container (e.g., name, description and timing behaviour). At the right side of the panel, the interface displays the bar chart visualization of the global event counter of the sources and the actuators. This visualization shows the relation between incoming and outgoing events and can be extended with the additional chart visualizations of the editor.

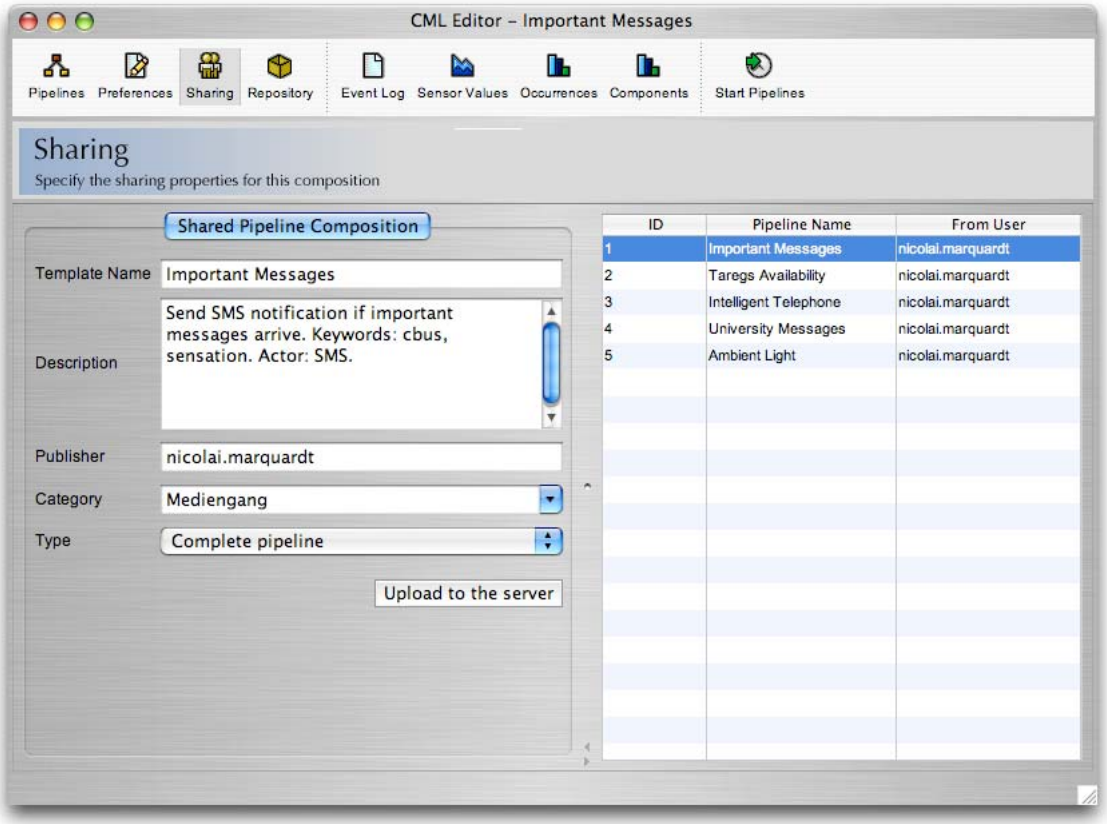

Figure 15. The shared repository view of the editor.

The panel for the shared repository upload (initialized with `buildPanelShared()`) contains two areas, divided with the `SplitPane` container: the left side displays input boxes for some parameters of the `SharedRepositoryEntry`, and the right side contains the view table of the remote located shared repository of the server. In this table you see a list of all available shared pipeline compositions. If the user activates the upload mechanism of the repository, the method `uploadToSharedRepository()` creates a clone object of the current container, adds this container to a new `SharedRepositoryEntry`, sets the additional parameter (specified in the user interface) and serializes the final object to XML. The method publishes this string to the remote repository server with the `RemoteConnection` instance.

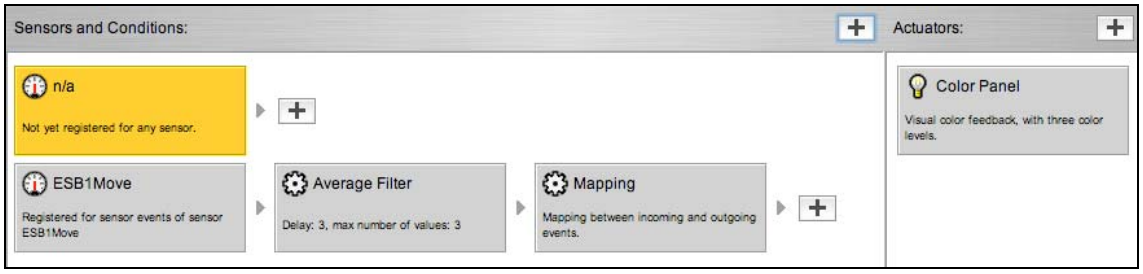

Figure 16. Create a new pipeline in the editor view.

For the dynamic creation of all components in the pipeline, some methods provide the functionality of adding and removing components, as well as some methods are responsible for the validation of all pipelines in the personal repository. The `addComponent()` method opens a new `ComponentSelectPanel` instance, so the user can decide which component filter he would like to add to the pipeline. The `ComponentSelectPanel` then calls the `addComponentToPipeline()` method of the editor to add the chosen component. This method not only instantiates and adds the component to the processing container, it also updates all the pipeline connections, so that the pipelines are correctly connected.

To add a new pipeline stream, the `addPipe()` method will be called. It creates a new sensor source component (will be displayed orange, because it is not initialized) and adds this component to the repository. The `addActuator()` method creates a new actuator object (here the system also displays the option dialog: the `ActuatorSelectPanel()`). Everytime a new actuator is added to the pipeline composition, the `connectActuatorPipes()` method guarantees the proper connection of all components (at the end of the pipeline stream) to the actuators.

4.3 GUI Dialogs

4.3.1 Discovery Panel for Actuators

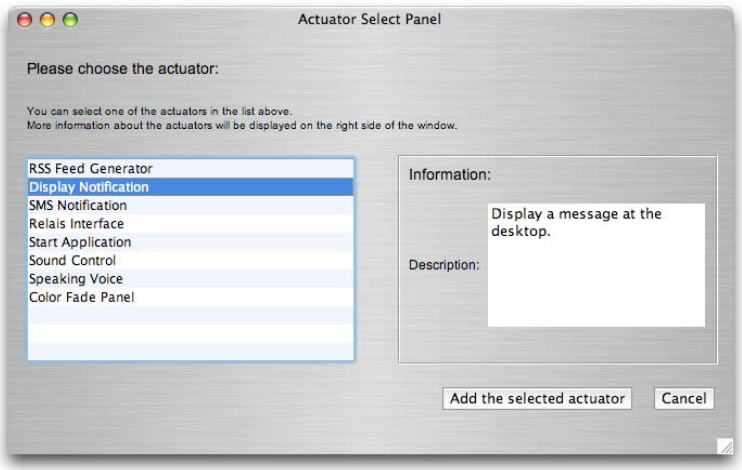

Figure 17. Select panel for the actuator components.

This class creates a `JFrame` window to display all registered actuators of the framework. This is a split view window: on the left side of the window is the `JList` component with the list of all actuator components (registered at the `ActuatorHandler` instance), and on the right side there is an information panel with a description of the selected actuator in the `JList`. There is a value change listener registered to the `JList` to receive a notification each time the selected value changes. If this listening event occurs, the class can update the information panel.

If the user has selected the desired actuator component and presses the "Add the selected actuator" button, the action handler calls the `addActuator` method of the parent editor. This reference of the editor was specified with the constructor call of the `ActuatorSelectPanel`. The `addActuator()` method of the editor gets the class name of the selected actuator in the `JList`, and therefore can dynamically instantiate the needed actuator class and add it to the processing container.

4.3.2 Select Filter Components

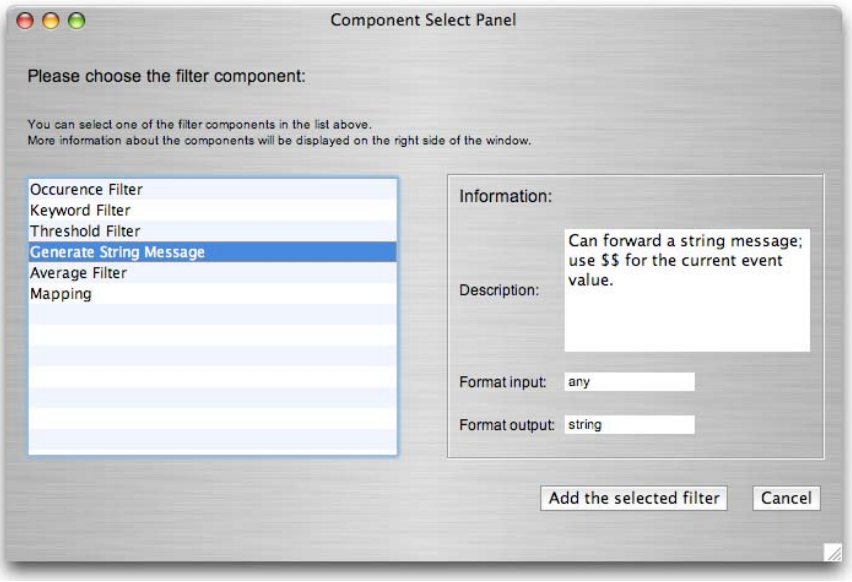

Figure 18. The filter component select panel.

This class creates a `JFrame` with the interface to select a processing component for the pipeline. The available components are added to a `JList`, and if the selected value of the list changes, the information panel on the right side of the window shows the updated component information. This is a similar mechanism to the one used within the `ActuatorSelectPanel`. If the user selects the `JButton` "use", the registered action handler will obtain the component description of the selected filter component with

```
ComponentDescription cd =
            componentHandler.getComponentDescription
            ((String)list.getSelectedValue());
```

and then call the `addComponentToPipeline()` method of the editor reference

```
editor.addComponentToPipeline(componentID, cd.getClassName());
```

The reference to the editor and the ID of the subsequent component are the obligatory parameters of the constructor. The new selected filter component will be inserted between the subsequent filter and the actuator component. The editor method `addComponentToPipeline()` will execute all necessary changes in the pipeline list to connect all components properly.

4.3.3 Assistant

The `Assistant` class is the dialog that is shown by the control interface if the user would like to add a new pipeline composition to his personal repository. The class shows a complete `JFrame` with areas to specify the composition name, select the sensor source and select the desired actuator as well. The sensor selection is implemented with the `JBrowser` component of the Quaqua library [Randelshofer 2005a] and initializes the `DefaultMutableTreeNode` object with the complete structure of available sensors of the connected *Sens-ation* server. The static singleton object `SensorRegistry` is used for the connection to the *Sens-ation* server to obtain the current list of all sensors. Then the `buildModel` method instantiates a DOM object of the received sensors XML file and uses XPath expressions to address each sensor description in the XML file. This is for example the XPath expression to select all sensor nodes:

```
XPath.selectNodes(jdomDocument, "/Sensors//Sensor");
```

or the selection of all sensors with a specified location ID:

```
XPath.selectNodes("/Sensors//Sensor
         [@LocationID='" + location + "']");
```

The XPath object extracts the desired nodes of the XML document, and we can easily iterate through these nodes to obtain additional information of each sensor. We classify the `sensorIDs` in four categories, to provide the user an easy selection of the desired `sensorID`. These four categories are:

1) `SensorID`: simply display all sensor IDs registered at the Sens-ation server
2) `Location`: display the sensor IDs as subentries of the available locations
3) `Users`: Group all sensors together with the same owner of the sensor
4) `SensorType`: here we use the sensor type parameter of each sensor entry to group all sensors with the same type together (e.g., all temperature or all movement sensors)

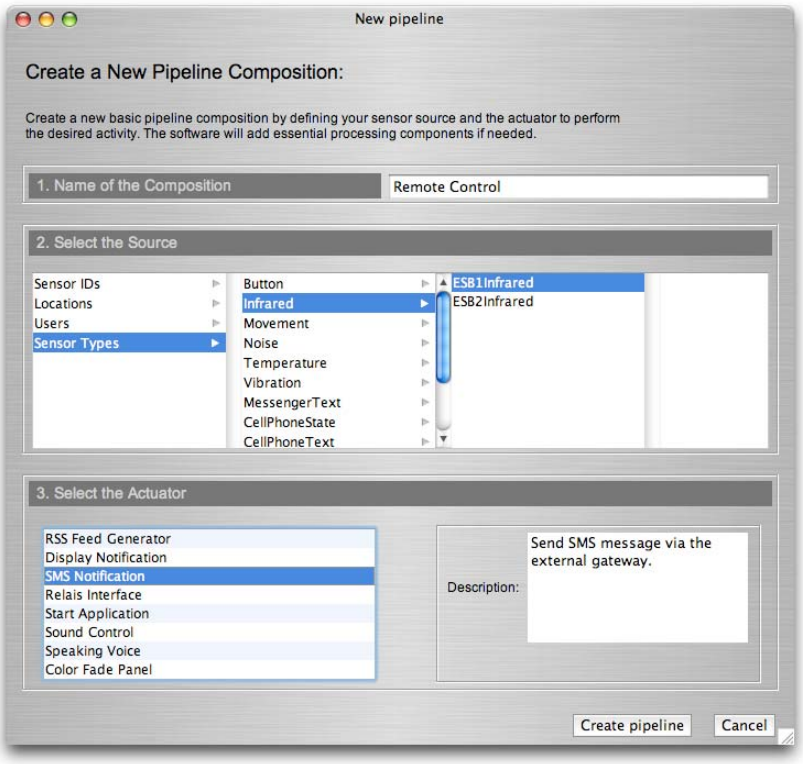

Figure 1. Assistant window, to guide the user while creating new pipeline compositions.

The actuator selection is implemented with the `AssistantActuatorSelect` class. This class file creates a `JPanel` with the current registered actuators in a `JList` component on the left side and the information for the selected actuator in a `JPanel` on the right side. The update mechanism works similar to the method explained for the `ActuatorSelectPanel`. If the user has inserted all needed information (write name, select source sensor and select actuator) and presses the "Create pipeline" button, the class will call the `createPipeline()` method. This method creates a new processing container as well as the objects for the source sensor and the actuator component. These two objects will be inserted to the processing container, and then the method creates the needed pipeline connections between the two components. Finally, the created processing container object will be added to the personal repository of the user. This completes the creation of the first, simple pipeline; the user can immediately start this pipeline composition and the selected actuator will react in dependence of the source sensor. But even though this is a complete pipeline composition, there are no processing components (e.g., filters) added to the pipeline. Thus the user can start the pipeline editor to specify his personal preferences and for example reduce the incoming sensor value events with using the filter processing components.

4.3.4 Draw the Processing Container

This is an interface wrapper class to display the processing container in the `Control` GUI. The class encapsulates the handling of the source processing container, the editor and also controls the user interface elements for the single container: name label, description label, current state colour label, edit button and the start/stop toggle button. If the editor is started, the class sets the edit flag of the processing container to true, and

changes the GUI controls. It will be notified, if the editor has been closed and saved all preferences to the processing container, so it can change the view back.

4.4 Graph Visualizations

The graph visualizations are added to the Editor window to give the user an overview of the inter-pipeline communication. Therefore we add the two main visualizations: the bar chart display, and the time plot of the sensor values.

4.4.1 Time Bar Chart

This chart visualization can display the occurred values of the components in a processing pipeline. It is divided into six bar chart areas; from 0 seconds up to 60 seconds (in the past). Every ten seconds, the `run()` method of the thread calls the `getEventCounter()` of each of the registered components of the processing container, and shifts all values to the next "ten seconds" time slot. Therefore we obtain an overview of the events in the pipeline of the last 60 seconds, and can compare all the occurred events in each of the pipelines single components.

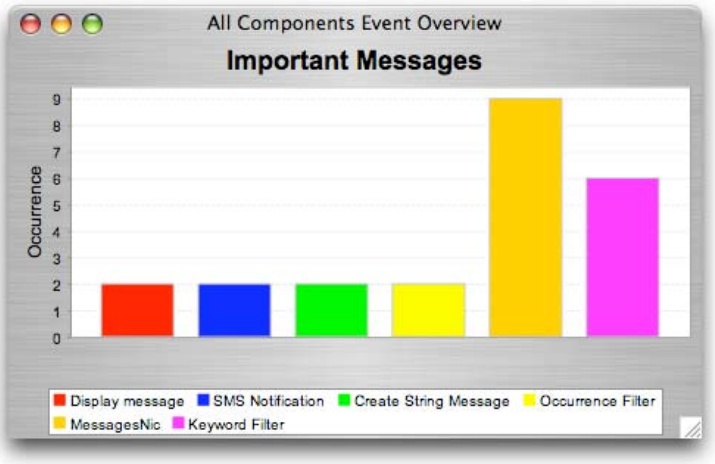

Figure 19. The visualization of the forwarded events of each component.

4.4.2 Time Plot Chart

This chart window enables the visualization of all occurred events in a processing pipeline, as well as their numeric values (if the event values can be parsed to a numeric value; integer or double). The `TimePlotWindow` class creates a new `JFreeChart` [Jfree 2005b] and a control slider for the displayed time interval. With this slider, the period of time that is displayed with the chart visualization can be increased or decreased. For each of the components in the processing container, the `TimePlotWindow` class creates a new instance of the `TimePlotChart` class with the `newPlot()` method. These classes are responsible for the handling of the components values and the graph is registered at the processing container as a global event observer, and therefore receives all events that occur inside the pipelines. The processing container will call the `notifyContainerEvent(String componentID, PipeValue pv)` method of the `TimePlotWindow` class, and

the parameters are the `componentID` of the component that fired the event, and as second parameter the `PipeValue`. Each of the registered `TimePlotChart` windows can be removed with the `removePlot(ID)` method.

Figure 20. The time plot visualization and overview of the pipeline events.

5 Repository and Collaborative Sharing

All the pipeline compositions will be stored in the personal repository of the users, as well as the shared compositions are stored in the shared repository. This chapter gives an overview of the repository structure, the XML description and serialization, and the shared repository.

5.1 Technology

To store the personal repository of the registered users and the shared repository as well, the `RepositoryServer` provides an access interface via socket connections. The client software application (this is especially the Control class) connects to the server socket (at the default port 6555) and the server creates a new socket-handling module of the class `RepositoryServerThread`. This object is responsible for parsing the incoming client requests.

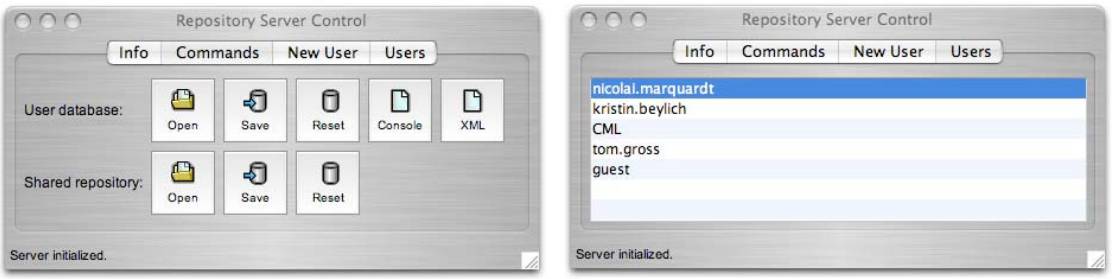

Figure 21. The control panels of the repository server

At first, the user has to login with a valid username and password using the `[LOGIN]` command of the `RepositoryServerThread`. If the login was successful, the client can receive the current personal repository of the user with the command `[GET_REPOSITORY]` and write the personal repository back to the server with `[SET_REPOSITORY]` (and the serialized XML data as parameter). To receive the current shared repository, the client can use the command `[GET_SHARED_REPOSITORY]` and to add a processing container to the shared repository `[ADD_TO_SHARED_REPOSITORY]`.

For the user interface for the repository server, the `RepositoryServer` can create a new instance of the `RepositoryServerControl` class. This class has a simple user interface for the administration of the remote repository server. The `commands` tab includes buttons to load and save the various repositories, and also to reset the repositories to the state of the backup file. The other two buttons can open a new system console for routing the `System.out` to the text console, and can display a new text window with the current user entries in the XML file.

5.2 Shared Repository

The `SharedRepository` object contains a vector with the collaborative shared entries of pipeline compositions; structured in the `SharedRepositoryEntry` object. Beyond the name and the description of the processing container, the entry contains the name of the publisher, a category entry (as a string, to group the shared entries) and a type that specifies, if the shared processing container is a complete pipeline composition (`SharedRepositoryEntry.TYPE_CONTAINER`) or an abstract template without the sensor and actuator information (`SharedRepositoryEntry.TYPE_TEMPLATE`). If the type is set to the `TYPE_CONTAINER` flag, then the processing container will be stored with the complete description of all sensor sources as well as the actuator components. If the type is set to `TYPE_TEMPLATE`, then the source components are set to their initial state, and the actuator component are removed from the pipeline composition, as well as the `reset()` method of the processing components is called to initialize these components.

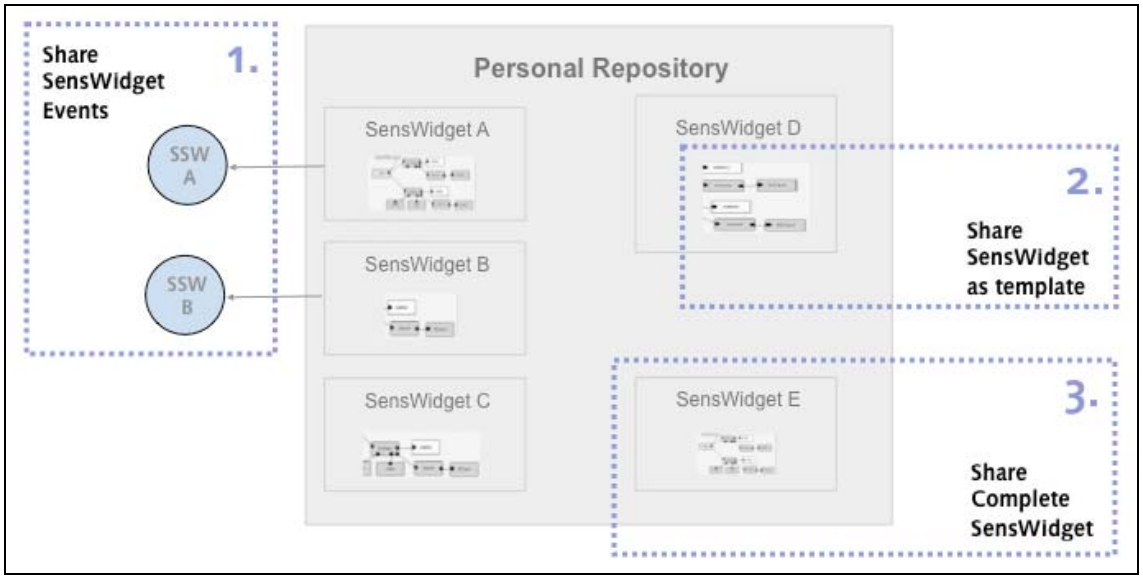

Figure 22. The sharing methods: sharing events, templates or complete compositions.

5.3 XML Serialization

The repository server stores the structure of the user table in the XML file specified in the `collaborationbus.properties` file. Via the XML serialization, all user information can be received by the remote repository server (and by default saved into the `SharedRepository.xml`, or the `ServerRepository.xml` for the user library).

The shared repository file of the server contains the `sharedRepositoryEntries` with the information of the shared `ProcessingContainer`. Each entry includes the publisher, the `type` (complete or template), the category and a description. The last entry is the serialized `ProcessingContainer` (see Figure 23).

```
<sharedRepository>
  <sharedContainerVector>
    <sharedRepositoryEntry>
      <publisher>Nicolai Marquardt</publisher>
      <type>Complete pipeline</type>
      <category>Cooperative Medie Lab</category>
      <description>Information channel of the CML lab.</description>
     <processingContainer>
     [...]
     </processingContainer>
   </sharedRepositoryEntry>
   [...]
</sharedRepository>
```

Figure 23. The XML structure of the shared repository file

The serialized `ProcessingContainer` includes the information of the pipeline composition (all the components: sensor sources, filters and actuators), the name and description, the Boolean tags for the editor and the active pipeline as well as the tags for

the sharing mechanism. The `pipes` entry contains all pipeline connections between the single components and define the `pipeValue` data flow.

```
<processingContainer>
  <name>CML RSS Info</name>
  <description>Information channel of the CML lab.</description>
  <active>false</active>
  <edited>false</edited>
  <components>
    [...]
  </components>
  <pipes>
    <pipeEntry>
      <sourceID>source_component_1</sourceID>
      <sinkID>filter_component_2</sinkID>
    </pipeEntry>
    <pipeEntry>
      <sourceID>filter_component_2</sourceID>
      <sinkID>filter_component_3</sinkID>
    </pipeEntry>
    <pipeEntry>
      <sourceID>filter_component_3</sourceID>
      <sinkID>filter_component_4</sinkID>
    </pipeEntry>
  </pipes>
  <sharePrivileges/>
  <tagTemplate>false</tagTemplate>
  <tagReuse>false</tagReuse>
  <tagShareable>false</tagShareable>
  <idCounter>6</idCounter>
</processingContainer>
```

Figure 24. Serialized XML processing container entry

The pipeline components are stored in the components hash table of the `ProcessingContainer`. They are serialized as entries with all not transient members, and therefore full configured components of the pipeline (as seen in Figure 25 with the keyword filter component, the entries in the keywords vector and the other member values: the occurrence of the minimum key words, the forwarding type, the component ID, etc.).

```
<entry>
    <string>filter_component_8</string>
    <filterKeyword>
        <keyWords>
          <string>gross</string>
          <string>egla</string>
          <string>marquardt</string>
          <string>krause</string>
          <string>oemig</string>
        </keyWords>
        <occurrence>1</occurrence>
        <forwardType>Forward the found keywords</forwardType>
        <id>filter_component_8</id>
        <componentType>filter</componentType>
        <eventCounter>1</eventCounter>
        <globalEventCounter>1</globalEventCounter>
        <name>Keyword Filter</name>
        <description>gross, egla, marquardt, krause, oemig</description>
        <isActive>false</isActive>
        <interrupted>true</interrupted>
        <isInitialized>false</isInitialized>
        <sinks>
          <filterStringGenerator
```

```
            reference="../../../../entry/filterStringGenerator"/>
        </sinks>
        <pc reference="../../../.."/>
    </filterKeyword>
</entry>
```

Figure 25. Entry of the components hashtable in the ProcessingContainer

5.4 Shared Repository View

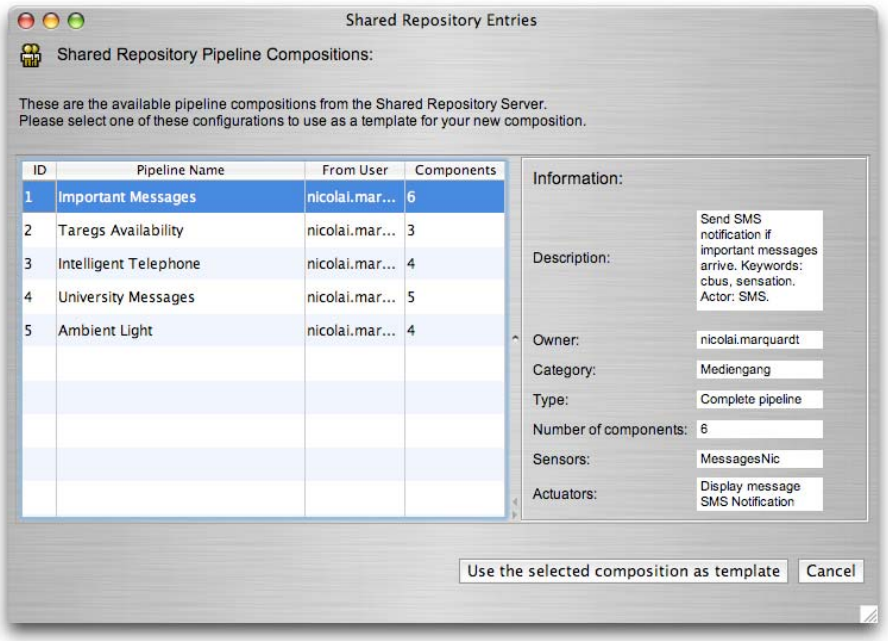

Figure 26. The shared repository view

All the current registered shared repository entries can be displayed with the `SharedRepositoryTable`. It initializes the `DefaultTableModel` of the `JTable` with the entries obtained from the remote repository server instance. It iterates through the hashtable of `SharedRepositoryEntries`, and arranges the information in the `JTable` GUI component and the `JPanel` information part. This panel displays all further information of the selected shared pipeline composition in the `JTable`:

- Description: additional information of the publisher of the pipeline composition
- Publisher: has shared his own personal pipeline composition
- Category: can be one of the suggested entries or a new added category. This is helpful for classification of the shared repository entries and can later be used for search algorithms.
- Type: this can be `TEMPLATE` (without the sensor information and actuators) or `COMPOSITION` (complete processing container)
- Number of Components: the number of used sensors, filters and actuators
- Sensors: the sensors added to the pipeline composition
- Actuators: all actuators of the composition

6 Conclusion

6.1 Summary

In this research project we have developed an application to enable the end users the creation of new sensor-actuator relations, with their own personal preferences and without the barrier of complex configuration settings or programming details. The software tries to hide as much as possible all the details of the underlying base technology, as for example the sensor infrastructure, the sensor and actuator registration and the registration for sensor events.

Furthermore the users cannot only handle all the pipeline compositions for themselves, but also collaborative share the pipelines with their colleagues and friends via the Shared Repository of the server. With a minimum effort, each user can profit from this central repository of the shared pipeline compositions, and of course the users can adapt the used shared repository template to fit to their needs (by specifying their own personal properties of the pipeline network).

6.2 Future Work

Although we have implemented the main components of the functionality for the repository and pipeline compositions, there are still some aspects of the system that can be improved or implemented in future releases.

6.2.1 Evaluation of shared pipeline compositions

It could be interesting to evaluate the common pipeline compositions of the users (especially these in the shared repository). Perhaps it is possible to find common patterns in these compositions and then develop algorithms to provide suggestions for “reasonable” compositions to the user.

6.2.2 Graphical mapping user interface

The mapping interface at the filter component is still far away from an intuitive usage of the user. A better user interface design could be a graphical mapping interface, where the users can drag and drop the desired input and output commands and can draw connections between these commands simply by selecting the commands. Furthermore there can be tools to apply ranges of values, to facilitate the configuration of numerical mapping.

6.2.3 Coupling with the user authentication algorithm of the *Sens-ation* platform

An important aspect is also the introduction of a system-wide authorization and authentication system, to secure the access to the sensor values and pipeline compositions. Therefore the *Collaboration Bus* repository storage and the sensor value access should be integrated in the security system of the *Sens-ation* platform.

References

Barbalace, D. TableLayout, Layout Manager for Java, http://www.clearthought.info/software/TableLayout/, Version: Release May 2005 (Accessed 10/07/05).The Apache Software Foundation: Xerces, XML Parsing Library for Java, http://xml.apache.org/xerces2-j/, Version 2.7.0 (Accessed 10/07/05).

ChurchillObjects.com: RSS4J, Java Classes to Create and Parse RSS XML Files, http://www.churchillobjects.com/c/13005.html, Version 0.91 (Accessed 10/07/05).

Cooperative Media Lab Sens-ation, Sensor-based Infrastructure, http://cml.medien.uni-weimar.de/tiki-index.php?page=Sens-ation (Accessed 10/07/05).

Eliott Rusty Harold XOM, Tree-based API for Processing XML Files in Java, http://www.cafeconleche.org/XOM/, Version 1.0 (Accessed 10/07/05).

Forsythe, C. et al.: Growl – Global Notification System for Mac OS X, http://growl.info/documentation/developer/java/ (JavaDoc for the Java binding class) (Accessed 10/07/05).

JDOM – Java XML API, http://www.jdom.org/, Version 1.0 (Accessed 10/07/05).

JFree.org Project: JCommon, Class Library for Java, http://www.jfree.org/jcommon/ (Download), http://www.jfree.org/jcommon/jcommon-0.8.9.pdf (Documentation), Version 1.0.0 rc1 (Accessed 10/07/05).

JFree.org Project: JFreeChart, Free Java Class Library for Generating Charts, http://www.jfree.org/jfreechart/index.php (Download), http://www.jfree.org/jfreechart/javadoc/, (JavaDoc), Version 1.0.0 rc1 (Accessed 10/07/05).

Randelshofer, W. Quaqua Look and Feel, Mac OS X Enhancement Library, http://www.randelshofer.ch/quaqua/download.html, Version 3.1.1 (Accessed 10/07/05).

Randelshofer, W. Quaqua Look and Feel – Documentation, http://www.randelshofer.ch/quaqua/guide/index.html (Manual) and http://www.randelshofer.ch/quaqua/javadoc/index.html (JavaDoc) (Accessed 10/07/05).

The Apache Software Foundation: XML-RPC API for Java, http://ws.apache.org/xmlrpc/, Version 1.1 (Accessed 10/07/05).

The Codehaus Open-Source Project Repository: Jaxen – Xpath Engine for Java, http://www.jaxen.org/, Version 1.1 Beta (Accessed 10/07/05).

The Codehaus Open-Source Project Repository: XStream, Java Library to Serialze Objects to XML and Back Again, http://xstream.codehaus.org/, Version 1.1.2 (Accessed 10/07/05).