THE RUN-TIME CUSTOMIZATION OF JAVA RICH CLIENTS WITH THE COMA CLASS

R. Bacher, P.K. Bartkiewicz, P. Duval
DESY, Hamburg, Germany.

Abstract

Java rich client applications can now make use of a COMA (Container Object MAnager) class, increasing the flexibility of their Graphical User Interfaces and extending data presentation capabilities. COMA, when used in application, makes size and position of all GUI components configurable during application run-time. Furthermore, new graphical components such as ACOP [1] controls can be added by a drag-and-drop mechanism to the existing user interface, providing additional access to control system resources. Configuration changes to component properties, including those of any added components, can be saved to an XML file, which can be reloaded at the next application start. This allows the possibility of having user-defined extensions for existing, compiled, client applications. This paper describes the implementation of the COMA class and presents its benefits when used in connection with the ACOP beans.

INTRODUCTION: REACH VS. THIN CLIENT MODEL

In the client-server systems, client applications usually fall into one of two client programming paradigms, namely rich-clients or thin or simple-clients. The criterion used to categorize a client application as belonging to one of these paradigms is essentially the presence of the data or display processing logic in the client code.

We briefly describe here these models, weighing the advantages and disadvantages of both approaches.

Rich Clients

Rich client applications contain hard-coded logic, which processes data or display choices locally. In other words, data obtained from one or more servers is not simply attached to displayer widgets, but is filtered, tested, or compared against display criteria or user input prior to rendition. This 'business logic' generally provides a much greater flexibility and convenience in data manipulation, as it can be designed around specific application requirements. The functionality of the client application does not need to merely reflect a server's properties. Hence, a rich client creates a more sophisticated view of and offers a more sophisticated interaction with the data. Reaching the same level of sophistication by configuring simple displayer widgets is nearly impossible, as one is forgoing all of the advantages of a rich programming language such as java and an Integrated Developing Environment (IDE) such as Eclipse or NetBeans.

The significant disadvantage of the rich-client approach is that any modifications to an existing rich-client application are not easily configurable and consequently require the attention of an application developer familiar with the programming language and the application code. Simply adding or removing a component to or from the graphical user interface will likely violate the internal logic of the application and will likely require program recoding.

Thus writing rich client applications requires real programming skills and therefore ordinary end-users such as operators in a control room will need to contact professional programmers in order to customize client application to their needs.

Thin Clients

In contrast to the rich client, the thin or simple client does not contain any data processing or display logic. All incoming data is displayed using dedicated, data specific components (displayers). Likewise, user requests and commands are sent directly to the servers without any pre-processing or filtering. The graphical displayers used to either render incoming data or submit outgoing data and commands typically provide a data-transfer mechanism to access the control system endpoints. The thin client, then, reflects a set (or sub-set) of properties offered by the control system servers.

The absence of internal business logic means that all graphical displayers and components are uncoupled to one another. On the other hand, this feature enables the dynamic configuration of an entire application, for example by reading configuration file at an application startup. The configuration file will likely contain a list of attributes defining the data connections and data transfer parameters as well as graphical aspects of the components (such as geometry, location, color attributes etc.). In this scenario a thin client application might consist of two principal elements. The first is a generic container application, i.e. 'an empty window', which is capable of reading a configuration file and rendering its contents. That is, the generic container will know how to create and display the necessary graphical components and connect them to the appropriate data channels. The second element is the configuration file itself, which is specific to the application task at hand. This defines the application functionality and appearance. Creating a thin client application now means creating the configuration file. This task can in turn be accomplished by inexperienced application end-users (those not familiar with programming techniques). Typically, simple client development is supported by a specialized graphical editor, providing a palette of components and displayers, which can be dragged, dropped and resized onto a form representing the final application. Additional component parameters such as data sources, data transfer protocols or graphical aspects of the displayers etc. can be defined in an associated property window.

'HYBRID' CLIENT MODEL

Both of the client paradigms presented above have their advantages. We now present a third paradigm, which in effect constitutes a 'hybrid' model. Our hybrid model client application is then divided into two parts: one, corresponding to the 'rich-client part', contains data processing and display logic and the code which binds data to graphical components. This part is created by the software developer by using the Java programming language. The second is a 'thin-client part' created by the end user during the run-time of the application, by adding specialized graphical components to the program's user interface and/or performing simple component configuration, (without programming). Such a model preserves the flexibility in client data and display manipulation, using dedicated embedded logic, and offers at the same time the possibility of adding new features during the application run. This approach can be very useful for users operating on test stands, beam lines, and control rooms during machine studies, where the rapid extension of console program functionality is most desirable.

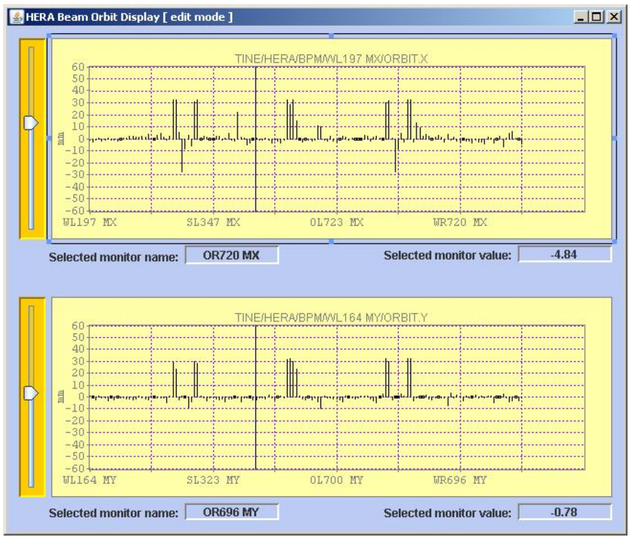
THE COMA CLASS: IMPLEMENTATION OF THE HYBRID CLIENT MODEL

The COMA (Container Object Manager) class is an implementation of the hybrid client model presented above. COMA, when instantiated by a Java rich client, turns one application window into thin client framework, preserving at the same time the entire application's business logic as well as all graphical elements belonging logically to the rich client application part.

The current version of COMA works with windows derived from the JFrame class. In order to make use of COMA one has to add a single line of code to his Java rich client application. This essentially creates a COMA object and passes a reference to the window frame:

```
new Coma( myJFrame );
```

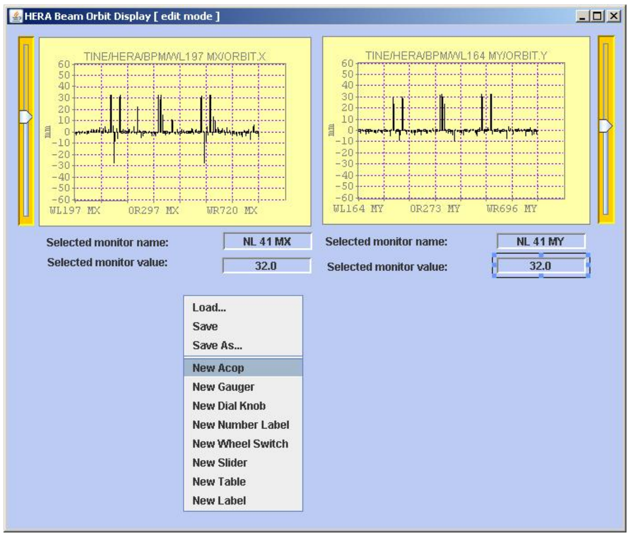
When the application starts COMA is 'inactive' and the behavior of the application remains unchanged. COMA becomes activated when the user presses a Ctrl key and simultaneously makes a right mouse button click on the application window. The active state of the COMA object is indicated by the window caption text ('[edit mode]') and a finger shaped mouse cursor. Now all graphical components contained by the window can be resized and moved to other positions. Resizing or setting a new position is very intuitive for even inexperienced users: a mouse click on a component evokes a resizing frame, as in most popular graphical editors. (fig 1).


HERA Beam Orbit Display [edit mode]
TINE/HERA/BPM/WL197 MX/ORBIT.X
WL197 MX
SL347 MX
OL723 MX
WR720 MX
TINE/HERA/BPM/WL164 MY/ORBIT.Y
WL164 MY
SL323 MY
OL700 MY
WR696 MY

Selected monitor name: OR720 MX Selected monitor value: -4.84

Selected monitor name: OR696 MY Selected monitor value: -0.78

Figure 1: Application window with Coma activated



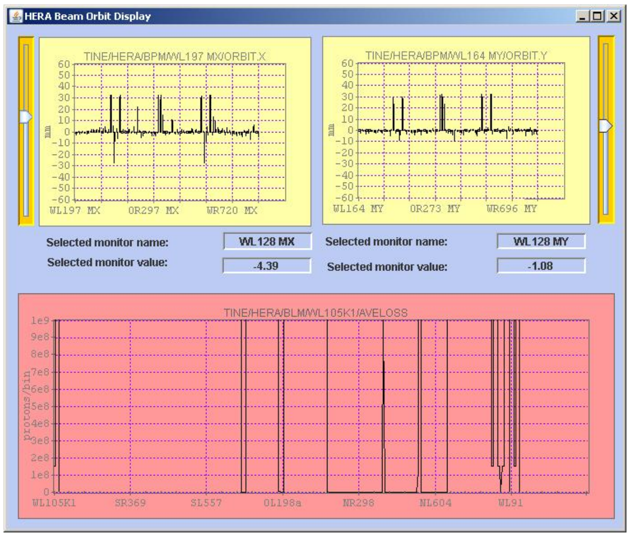
Selected monitor name: NL 41 MX Selected monitor value: 32.0

Selected monitor name: NL 41 MY Selected monitor value: 32.0

Figure 2: Adding an AcopBean component to re-arranged application window

The new graphical components can be also added to the window, by a simple drag-and-drop operation, or by invoking a pop-up menu. During a drag-and-drop action COMA accepts a string, representing a class name of the component to be instantiated. Having a full class name of a component COMA is able to create any component as long as the related jar-file is accessible and as long as the component follows the Java Beans specifications. Of course it makes sense to drop only those specialized components which offer the possibility of run-time customization of their parameters, such as control system connection parameters. The COMA edit-mode pop-up menu (fig. 2) currently offers only the AcopBean components, which provide run-time mechanisms for connecting to the TINE [2] control system. The geometry of newly added components can of course also be modified. Only those components added to the application during the run-time (i.e. not belonging to the 'rich part' of the application), can be removed (by selecting a component and pressing the Delete key).

The result of the run-time customization (fig.3) can be preserved for a later use. All changes made to the application using the COMA during the run-time can be saved to an XML configuration file. The configuration file contains information about any new graphical properties of the application window as well as all other properties, including the connections to control system parameters changed during the application run. During the ensuing application run a user can activate COMA and re-load one of the previously saved configurations. This feature of COMA makes it possible to have several extensions for a particular rich client application, reflecting specific usage contexts or users preferences.



Selected monitor name: WL128 MX
Selected monitor value: -4.39
Selected monitor name: WL128 MY
Selected monitor value: -1.08

Figure 3: Result of the run-time customization of the rich client application

COMA AS THE GENERIC THIN CLIENT APPLICATION

By adding the COMA class to an application which contains nothing but an empty JFrame, one can have a generic thin client application which can be used with various XML configuration files. In such cases the configuration file describes the entire client application.

In addition, this 'empty frame' application can also play the role of a simple graphical tool for generating those configuration files.

The COMA jar file itself contains an empty, generic thin client application. Since COMA offers a pop-up menu providing users with the choice of all available AcopBeans, building thin client applications for the TINE control system has become a very simple task. One begins by running the generic client frame, for example by executing in the command line window:

```
java -jar <path>coma.jar
```

The next step is to activate COMA, by pressing the Ctrl-key while clicking the right mouse button on the window surface. From the pop-up menu one can choose among a variety of AcopBeans, resize them, and move them to the desired locations. When all components are in position, one exits the COMA editing mode, again by pressing the Ctrl-key and issuing a right-mouse-click. Now using the property customizer window supplied by each member of AcopBeans family one can connect the controls to the TINE servers and also define some graphical aspects of controls, like colors, fonts, description texts etc. The thin client application is now ready without writing a single line of code or using any external editors or frameworks. If the application is to be reused, one should re-activate the COMA and choose from the pop-up menu the Save or Save-As option.

CONCLUSIONS AND CLOSING REMARKS

As presented in this paper, COMA provides a powerful yet simple and intuitive way for extending rich client applications to incorporate simple client techniques. The minimal effort of adding the COMA class to a project brings a lot of flexibility in configuring and extending the functionality of a rich-client application. At the same time, COMA allows raw simple client development at run-time. The generic, thin client application framework contained by the COMA jar file seems to be, in case of simple client applications, an alternative for complex thin client generating or configuring tools.

COMA plus AcopBeans provide a light-weight, framework-independent way of the rapid application development of both rich-clients and simple-clients.

Although COMA was designed to be used in the TINE control system environment, it can be easily adopted for other control systems.

REFERENCES

[1] "The ACOP family of beans : the framework independent approach" J.Bobnar at al, these proceedings
[2] http://tine.desy.de