Integrating a Tool into Multiple Different IDEs

Lutz Prechelt and Matthias Peter
abaXX Technology AG, Stuttgart
lutz.prechelt|matthias.peter@abaxx.de

Abstract

abaXX Techology produces component-based platform products that help in the construction of web-based systems, in particular process portals, using Java2 Enterprise Edition (J2EE) technology. Most parts of these products are API-based and hence require support by appropriate construction tools. Much support is available in leading J2EE IDEs, but some specialized tools have to be provided in addition. Since the products are platform-independent, the tools should ideally work in many different IDEs, too.

This position paper shortly describes the issues encountered in designing one of these tools in such a way that it is portable to both Eclipse 2.0 and IntelliJ IDEA 3.0 (and possibly others as well).

1. The starting point: Web UI Framework, Vanilla Portal, PortalBuilder

The Web UI Framework is one of abaXX' J2EE component products. It consists of the base framework (similar to Jakarta Struts [2]) for representing a Model-View-Controller design style for web-based user interfaces, a powerful tag library, a Parts framework for hierarchically modular UI construction and configuration, and the corresponding runtime system.

The Parts framework defines the notion of Part, a fragment of a UI dialog page having its own view, controller, and model. Parts appear to be somewhat similar to Portlets, but in fact they are much more lightweight, can be arbitrarily nested by using containers (CompositePart) with dynamically controlled layout, and can have their look-and-feel be centrally modified by Decorators.

The structure of a portal is defined in a file called parts.xml; see an excerpt in Figure 6.

Along with the Web UI Framework we ship the Vanilla Portal, an almost-empty portal frame containing a few generic reusable Parts and predefining the directory structure, naming conventions etc., thus making setting up a new portal development project quick and easy.

On top of the Vanilla Portal comes the third major element, the PortalBuilder: an application for interactively modifying a live portal on the Parts level. The whole portal (see Figure 1) is switched into 'edit mode' (see Figure 2) and then Parts can be introduced, removed, moved, and (re)configured. One can even introduce new (not yet implemented) Parts, will immediately get to see a dummy representation, and can then add actual views and controllers incrementally. During all of these activities, the full functionality of the portal proper is always visible and available for use.

2. The tool and the integration goals

During the implementation phase of a Part (writing the view JSP, controller class, and model bean), one would not normally want to work with the PortalBuilder, but rather with an IDE. Nevertheless, some of the functionality of the PortalBuilder is relevant then, too – namely, entering, reviewing, and modifying parts.xml parameters for the given Part, its parent Part (container), and children Parts, if any.

For simplifying this task, we offer a specialized tool, the *parts.xml editor* (see Figure 4) that allows for generating and editing these entries and that ensures their syntactic and semantic integrity. For maximum benefit, the parts.xml editor needs to be integrated into the IDE.

The following integration between parts.xml editor and IDE would be nice:

(1) User starts the editor from within the IDE.
(2) Editor recognizes which parts.xml is relevant and where to find it; editor loads and saves it.
(3) Editor understands where to find any resource mentioned in the parts.xml (JSP, controller class, model class, decorator, layout, etc.); can make the IDE load and show/edit any of these.
(4) Editor can create lists of candidate resources in any of the various categories from (3) and offer them in selection lists.
(5) Editor makes semantic checks of internal consistency of Parts descriptions. (Note that this does not really require integration.)

(6) Editor makes semantic checks of consistency between Part description and the resources mentioned therein, such as (in increasing order of complexity): JSP exists, controller class exists, controller class is indeed a controller class, all events declared in Part description are fired somewhere, etc.

So far, we have implemented (1), (2), (3), (5), and parts of (4), but only basic parts of (6).

3. Integration issues

We have currently implemented the parts.xml editor for three different contexts:

- The IntelliJ IDEA 3.0 IDE ([4], see Figure 3)
- The Eclipse 2.0 IDE ([3], see Figure 4)
- The abaXX Workflow Modeler 3.2 tool (see Figure 5)

The Workflow Modeler is an editor that manipulates process descriptions for the abaXX Workflow Engine, a process execution component that integrates process control logic, calls to business logic, and GUI page flow.

3.1. GUI issues

The GUIs of different IDEs are neither technically nor conceptually identical (or sufficiently similar).

For example, the parts.xml editor tool is programmed in Java Swing (Java Foundation Classes), which is fine for IntelliJ IDEA, because Swing is both its native technical GUI platform as well as its standard look and feel. The same is true for the Workflow Modeler.

For Eclipse, however, Swing is foreign: Eclipse, although also based on Java, is built using a special, native GUI library. While the look-and-feel issues arising out of this can be overcome, at least for a tool as simple as the parts.xml editor, the technical integration becomes a problem: The parts.xml editor cannot easily be shown as a fully integrated subwindow of an Eclipse session, but appears as a separate window on top.

For more advanced tools, these problems will become worse.

3.2. Semantic integration issues

The repository structure and services of different IDEs are very different.

With respect to the integration wish list from Section 2, this means that all functions that require advanced access to the IDEs fact base are difficult to design in a portable fashion. They essentially have to be re-done for each new IDE.

For example, while the file-based integration functions such as most of (3) and simplified versions of (4) can be ported reasonably well, the advanced parts of (6) dig so deep into the IDEs internal model of Java programs that their design will invariably be very different for different IDEs. In some cases (IDEA may be one of them) it may even be impossible to provide this integration, because too little of the respective functionality of the IDE is documented and exposed to the tool developer.

3.3. Conceptual issues

Underlying all of the above technicalities is a much more fundamental problem. Different IDEs approach their problem in conceptually different ways.

For instance, while IntelliJ IDEA follows mostly a rather pragmatic approach, working in a file-level, text-based manner wherever possible, other tools that are more inclined towards a Modeling/CASE Tool kind of approach will not just have different technical mechanisms inside, but will require an add-on tool to have a totally different form and approach in order to get a good conceptual fit. In the case of the parts.xml editor this could mean for instance to have visual representation (and direct manipulation) of the inheritance relationships and event relationships between parts, rather than just a parts tree and attribute table.

4. Conclusion

Integrating a development tool tightly and adequately into more than one kind of IDE is a difficult task. The standardization that would be required for making this task easier is not currently in place, neither on a technical level (APIs, GUI look, basic GUI feel), nor, much more importantly, on a conceptual level (repository structure, service architecture, overall presentation and operation styles).

5. References

[1] abaXX Technology AG, “abaXX.components”, *http://www.abaxx.com*.

[2] Apache Software Foundation, “Jakarta Struts”, *http://jakarta.apache.org/struts/index.html*.

[3] eclipse.org, “Eclipse”, *http://www.eclipse.org*.

[4] JetBrains Inc., “IntelliJ IDEA”, *http://www.intellij.com/idea/*.

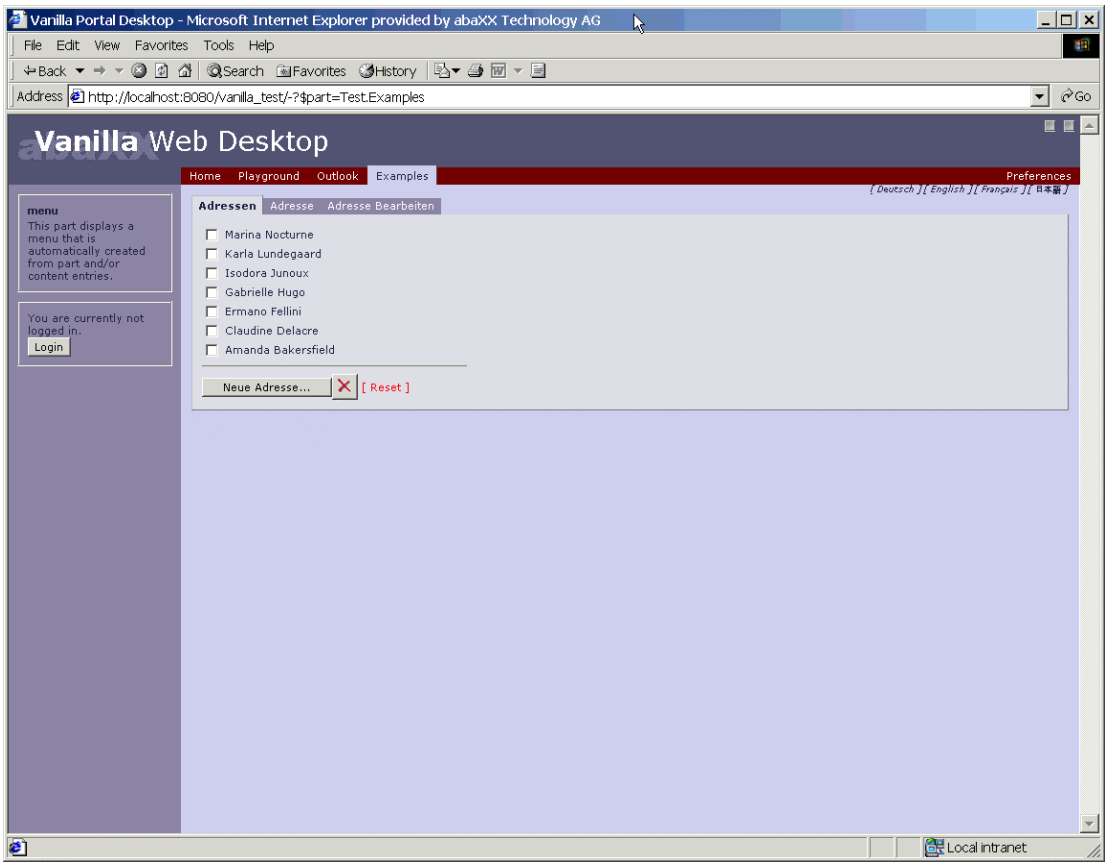

Figure 1: A small portal with 5 Parts: banner, menubar, empty menu, login, and a mini-application

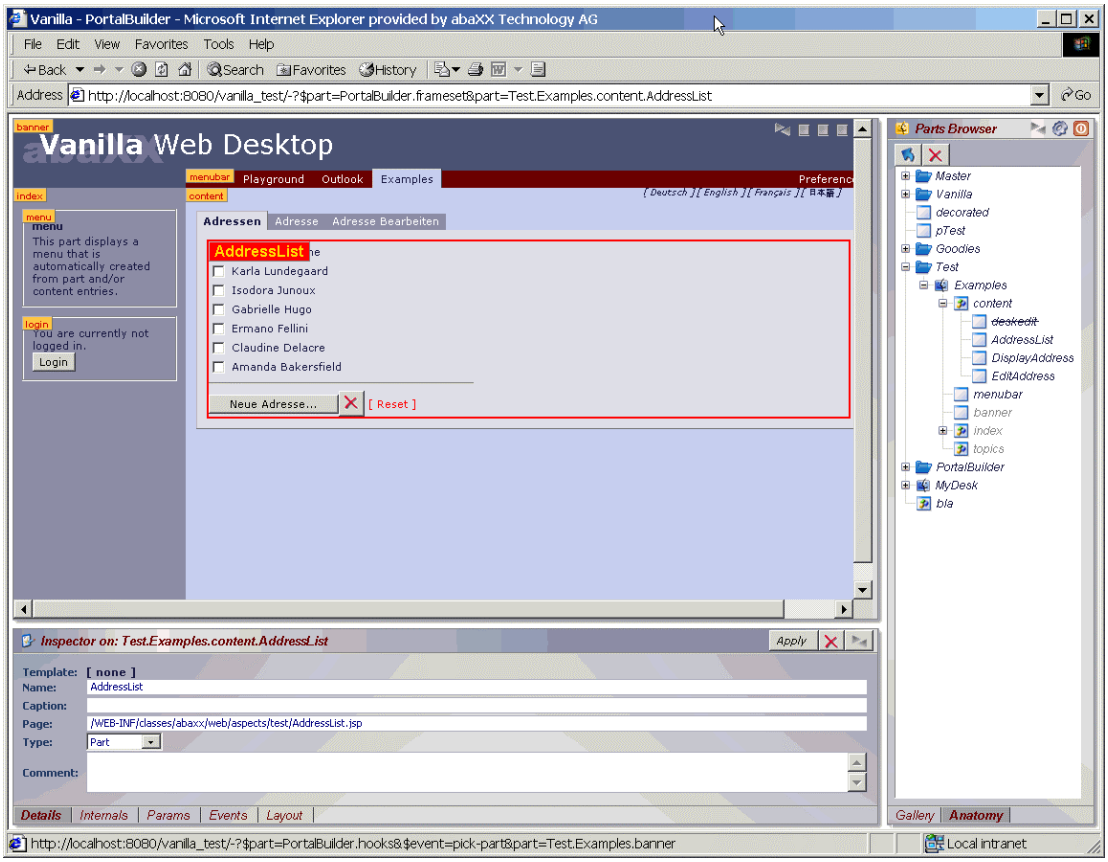

Figure 2: The same portal in PortalBuilder mode. The PartsBrowser shows some of the portal's Parts hierarchy.

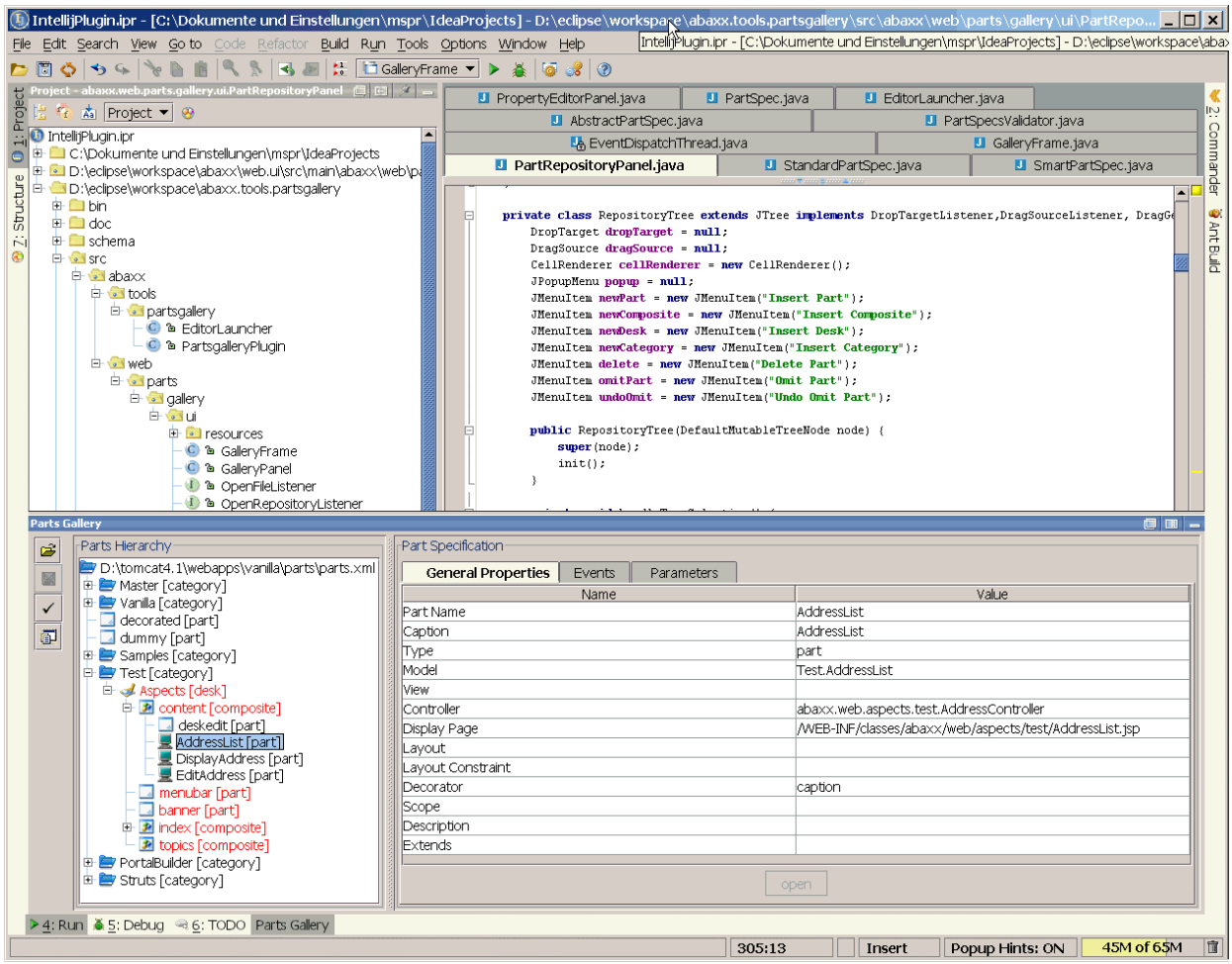

Figure 3: The parts.xml editor tool window within the IntelliJ IDEA IDE.

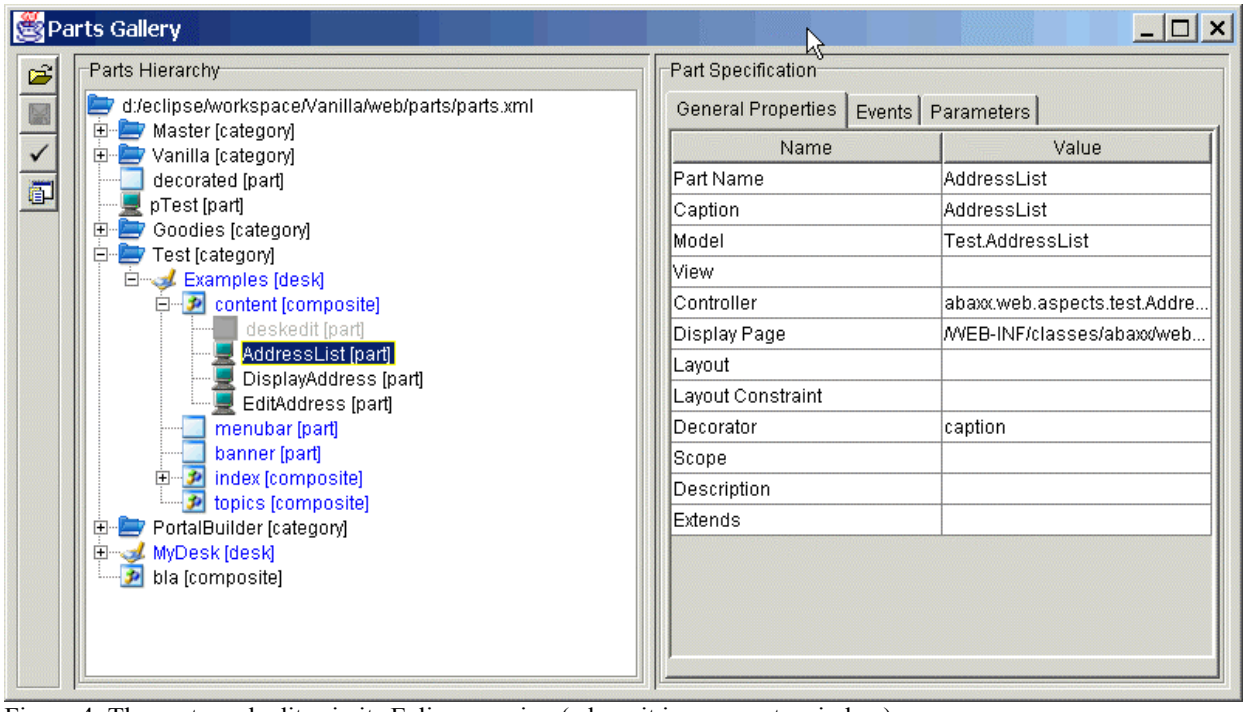

Figure 4: The parts.xml editor in its Eclipse version (where it is a separate window)

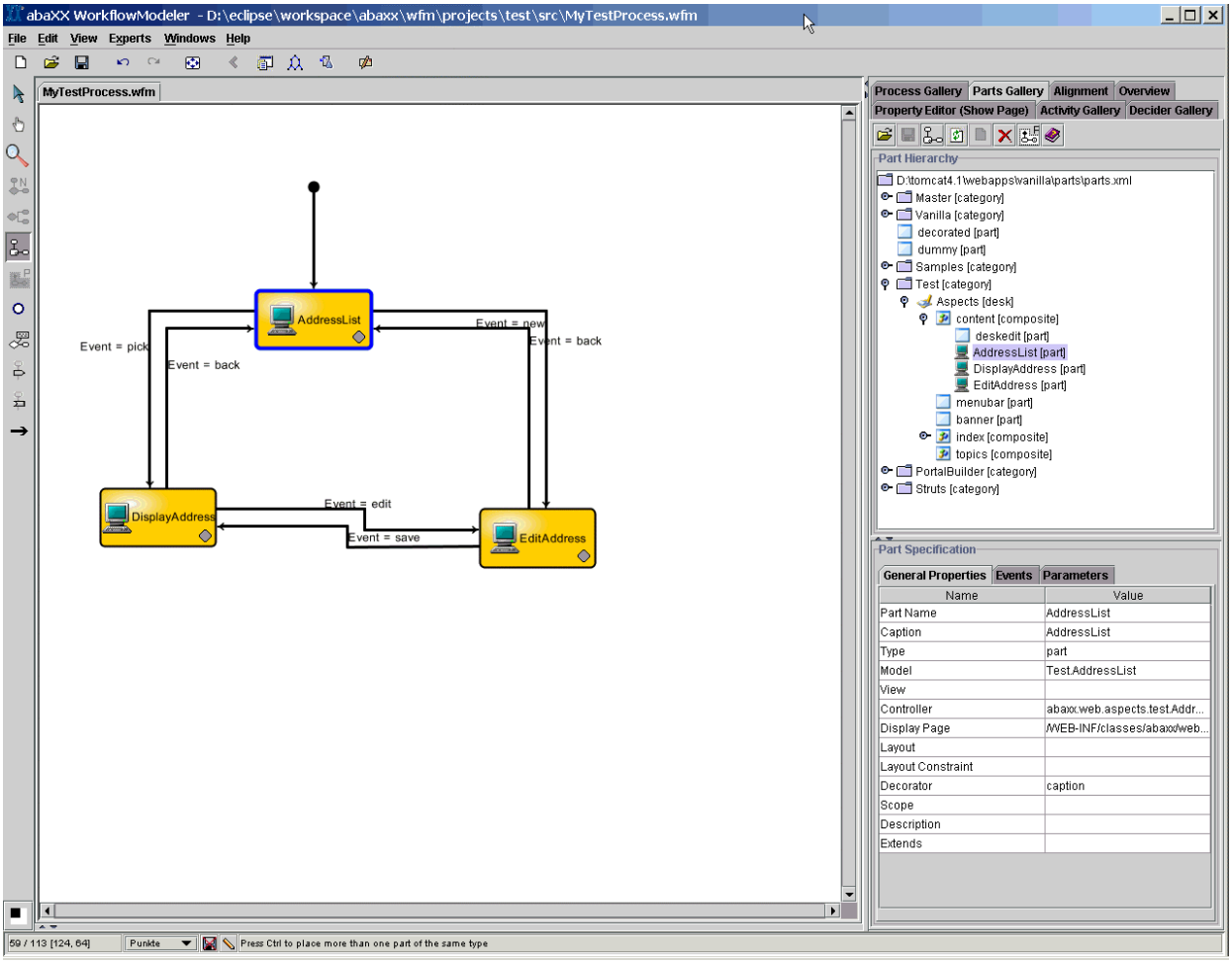

Figure 5: The parts.xml editor as a plugin to the abaXX Worflow Modeler.

```
<desk name="Examples">
        <theme>
                <styles>
                        .portal-content, .menubar a.selected { background-color: #ccccee; }
                        .portal-banner { color: #aaaaff; }
                </styles>
        </theme>
        <content layout="tabbed-switch" visual="abaxx.web.parts.CompositePart">
                <part name="deskedit" omit="true" />
                <part name="AddressList" controller="abaxx.web.aspects.test.AddressController"
                  model="Test.AddressList" url="/WEB-INF/classes/abaxx/web/aspects/test/AddressList.jsp"
                  decorator="caption">
                        <event name="pick" target="DisplayAddress" />
                        <event name="new" target="EditAddress&gt;create" type="redirect" />
                        <event name="delete" />
                        <event name="reset" />
                </part>
                <part name="DisplayAddress" controller="abaxx.web.aspects.test.AddressController"
                  model="Test.Address" url="/WEB-INF/classes/abaxx/web/aspects/test/DisplayAddress.jsp"
                  decorator="caption">
                        <event name="edit" target="EditAddress" />
                        <event name="back" target="AddressList" flags="populate,validate" />
                </part>
        [...]
```

Figure 6: Excerpt from the parts.xml file