A2 Component Framework

Version 1.01a
September 30, 2009

S. Stauber, ETH Zürich

Abstract

To simplify and speed-up the process of designing graphical user interfaces, the A2 component framework has been enhanced and a component editor tool has been implemented. This document outlines the enhanced A2 component framework and provides an overview of the new component editor tool.

Table Of Contents

1 Component Framework

This section provides a brief introduction to the A2 component framework. Subsection 1 explicates the component hierarchy and component serialization. Subsection 2 introduces Repositories which are containers that hold externalized components.

1.1 Components

1.1.1 Hierarchy

The classes involved in the component type hierarchy can be partitioned into three main groups:

- *XML* is responsible for component externalization and hierarchical composition
- *Repositories* establishes a link to Repositories (see section 1.2 Repositories)
- *WMComponents* provides enhanced thread-safety and reflection as well as the basic functionality of visual components

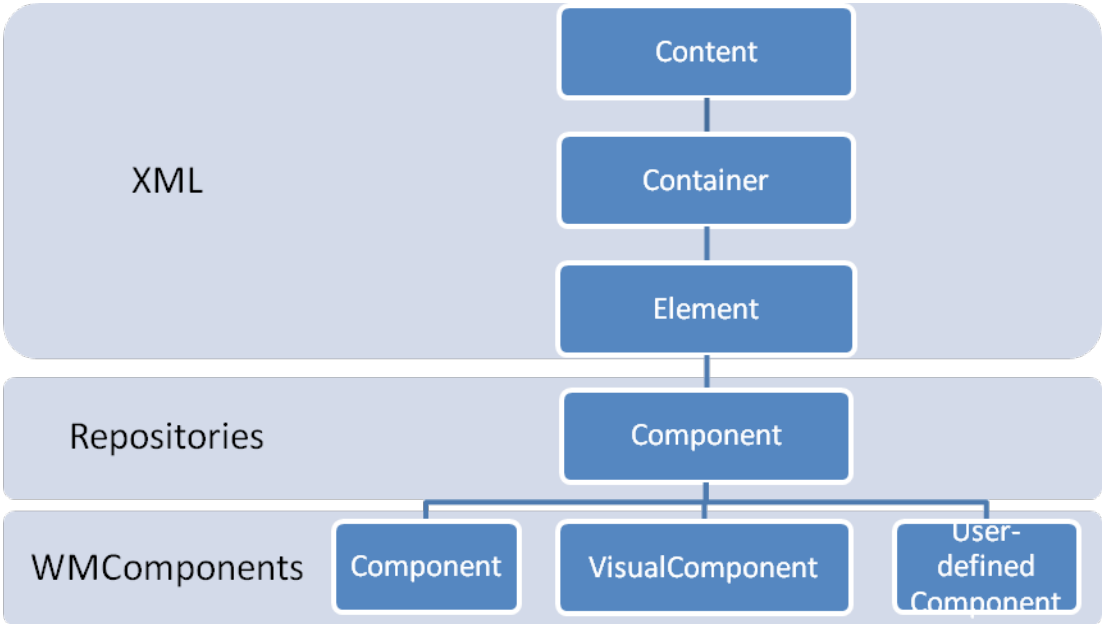

Figure 1: Component Hierarchy

The base class of all components is the class `Content` which essentially specifies the interface for component externalization.

```
1  Content* = OBJECT
2     PROCEDURE Write*(writer : Streams.Writer; indent : LONGINT);
3  END Element;
```

Figure 2: Content Class Interface

The class `Container` is responsible for handling the containment relation. Content instances can be added, removed and enumerated.

```
1  Container* = OBJECT(Content)
2     PROCEDURE AddContent*(content : Content);
```

```
3     PROCEDURE RemoveContent*(content : Content);
4     PROCEDURE GetFirst*() : Content;
5     PROCEDURE GetNext*() : Content;
6  END Element;
```

Figure 3: Container Class Interface

The Element class introduces names, generic attributes and establishes the hierarchical structure of the component composition by maintaining the parent relation.

```
1  Element* = OBJECT(Container)
2     PROCEDURE SetName*(CONST name : ARRAY OFH CHAR);
3     PROCEDURE GetName*() : Strings.String;
4     PROCEDURE SetAttributeValue*(CONST name, v : ARRAY OF CHAR);
5     PROCEDURE GetAttributeValue*(CONST n : ARRAY OF CHAR) : S;
6     PROCEDURE GetParent*() : Element;
7  END Element;
```

Figure 4: Element Class Interface

So far, we can express hierarchical structures composed of named elements that can optionally have generic attributes. These structures can be externalized to XML by calling the Write procedure of the root element.

```
1  Component* = OBJECT(XML.Element)
2     PROCEDURE SetRepository*(
         r : Repository; CONST name : Name; id : LONGINT);
3     PROCEDURE GetRepository*(
         VAR r : Repository; VAR name : Name; VAR id : LONGINT);
4  END Component;
```

Figure 5: Component Class Interface

The Component class establishes the link from components to repositories (see section 1.2 Repositories). Subclasses of this class can be stored and retrieved to/from repositories. Components within repositories are identified by the triple (*Repository*, *component name*, *component ID*). The component ID makes it possible to refer to specific component instances.

```
1  WMComponents.Component* = OBJECT(Repositories.Component)
2  VAR
3     properties- : WMProperties.PropertyList;
4     events- : WMEvents.EventSourceList;
5     eventListeners- : WMEvents.EventListenerList;
6     id-, uid- : WMProperties.StringProperty;
7     sequencer- : WMMessages.Sequencer;
8
9     PROCEDURE Acquire*;
10    PROCEDURE Release*;
12
13    PROCEDURE FindByUid*(CONST uid : ARRAY OF CHAR) : Component;
14 END Component;
```

Figure 6: WMComponents.Component Class Interface

The class WMComponents.Component adds several mechanisms. A locking scheme using a combination of sequencers and hierarchy locks can be used to implement thread-safe components. The sequencer is also useful to decouple the caller from the callee. Properties describe properties of components that are automatically serialized and can be queried by component inspection tools. Please note that the Properties are more powerful than the generic attributes in means of type-safety and observer pattern.

Each component provides self-reflection information (Figure 1, lines 3-5). These fields are used by component inspection tools to access properties, event sources and event listeners. Also, the fields are used for runtime component wiring. For locking, a recursive hierarchy locks is used, i.e. lines 9-10 lock the whole component composite – not just the component. Lines 11-13: Modification of the component composite and search for components within the composite.

```
1  VisualComponent* = OBJECT(WMComponents.Component)
2  VAR
3     bounds-, bearing- : WMProperties.RectangleProperty;
4     alignment- : WMProperties.Int32Property;
5     fillColor- : WMProperties.ColorProperty;
6     visible-, takesFocus-, needsTab- : WMProperties.BooleanProperty;
7
8     PROCEDURE KeyEvent*(ucs : LONGINT; flags : SET; VAR ks : LONGINT);
9     PROCEDURE PointerDown*(x, y : LONGINT; keys : SET);
10    PROCEDURE PointerUp*(x, y : LONGINT; keys : SET);
11    PROCEDURE PointerMove*(x, y : LONGINT; keys : SET);
12    PROCEDURE PointerLeave*(x, y : LONGINT; keys : SET);
13    PROCEDURE WheelMove*(dz : LONGINT);
14
15    PROCEDURE DrawBackground*(canvas : WMGraphics.Canvas);
16    PROCEDURE DrawSubComponents*(canvas : WMGraphics.Canvas);
17    PROCEDURE DrawForeground*(canvas : WMGraphics.Canvas);
20 END VisualComponent;
```

Figure 7: VisualComponent Class Interface

WMComponents.VisualComponent is the base class of all components that can draw themselves to a canvas, i.e. can visualize themselves. All visual components have some properties describing their representation in common. Lines 8-13 is the component's interface to user input. Lines 15-16 makes a component draw itself into the specific canvas.

1.1.2 Externalization and Internalization

All components can be externalized to XML. The benefit of using XML as external representation over a proprietary format is that standard tools can be used to view, debug and edit externalized components. The A2 component editor tool, for example, can directly operate on externalized components that are stored within repositories (see section 2 The Component Editor).

To make it possible to externalize hard-coded component composites, the Component.internal flag has been introduced. This flag must be set for all components that are instantiated by code of their super-components. These components are simply not externalized (but their children are). An implication is that these components cannot be edited by the component editor tool - they are under complete control of their super-components.

Since all components are subclasses of the `XML.Element` class, components can be directly created by the XML parser. A component composite in its externalized form is a XML document. The element names correspond to particular components.

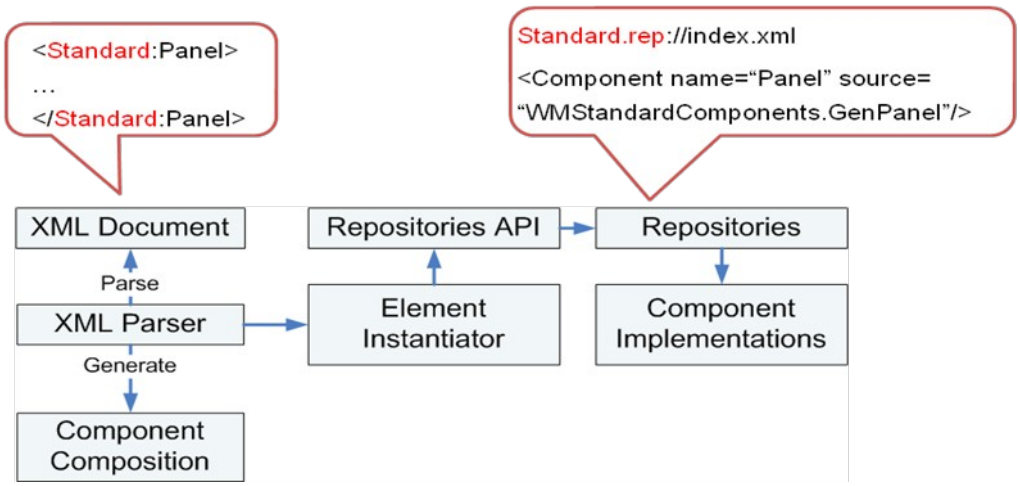

Figure 8: Component Generation

Figure 8 depicts how this works: When the XML parser encounters an XML element, it queries the element instantiator to create an instance for that element based upon the element name. The element instantiator tries to load the corresponding repository and to locate the component within that repository. If such a component exists, the Repositories API is used to generate an instance of the component. Remember here that the Repositories API also supports singleton components. In this case, it would just return a reference to the singleton instance. If the element instantiator is not able to create an instance of the component, the XML parser creates an instance of the XML.Element class and continues.

1.2 Repositories

Repositories are containers that group components, dictionaries and other data. Behind the scenes, repositories are implemented as archive files that contain a special index file describing the components and dictionaries within the repositories. Content can be partitioned into indexed and non-indexed content: Indexed content is described in the repository index and accessed using the Repository API. Non-indexed content are just files that are contained within the archive. It is accessed using Archives API.

1.2.1 Index

Each repository contains the special file `index.xml` that describes the indexed content in the repository. This index can be seen as an archive type independent extension of the archive index[1]. The index is used to establish an indirection between repository content names and archive filenames and to store additional attributes for repository contents.

```
1  <Repository>
2     <Header>
3        <Version>1.0</Version>
4     </Header>
5     <Components/>
6     <Dictionaries/>
7  <Repository>
```

[1] The archive index establishes a link between a path and filename and the actual file stored within the archive.

Figure 9: Repository Index

The index file is divided into three sections: The `Header` section contains meta-data about the repository, the `Components` section describes the components contained in the repository and the `Dictionaries` section where all dictionaries contained in the repositories are listed.

1.2.2 Components

For each component in the repository, there is an index entry that describes how the component can be instantiated.

Figure 10: Component Index Entries

```
1  <Repository>
2     <Components>
3        <Component name=”Editor” source=”WMEditors.GenEditor”/>
4        <Component name=”Nr” id=”1” source=”Models.GenInteger”/>
5        <Component name=”Mixer” source=”mixel.xml”/>
6     </Components>
7  <Repository>
```

Line 3 shows an example of a component that is instantiated using a generator procedure. Line 4 describes a component also instantiated by a generator procedure but bound to the repository. Bound means that the repository will only create a single instance of the component and always returns this instance when the corresponding component is retrieved from the repository. Finally, line 5 is an example of an externalized component stored in the repository.

1.2.3 Dictionaries

Each repository may contain an arbitrary number of so-called dictionaries. A dictionary is a named entity that essentially maps pairs of the form (language, string) to strings. For each dictionary and each language the dictionary supports, there are entries in the index file of the repository.

```
1  <Repository>
2     <Dictionaries>
3        <Dictionary name=”Demo”>
4           <Language name=”en” source=”demo.en.txt”/>
5           <Language name=”de” source=”demo.de.txt”/>
6        </Dictionary>
8     </Dictionaries>
7  <Repository>
```

Figure 11: Directory Index Entry

The Dictionary index entries establish mappings between the pairs (Dictionary, Language) and the respective files that contain the string-to-string mappings. These files are encoded in UTF8 and consist of a sequence of (word, translation) pairs.

```
1  “tDemo” = “Demo”
2  “tSentence” = “tSentence is translated into this sentence”
```

Figure 12: Language Translation File

The reasoning behind supporting multiple dictionaries within a single repository is that repositories can contain an arbitrary number of possibly independent components that support localization. Using separate dictionaries for each component does not create an artificial dependency on other components so that copying or moving components into different repositories remains simple. Additionally, dictionaries of components that are not used must not be loaded into system memory. It is also important for third-party component implementers to be able to implement standalone components that are not dependent on the presence and the content of a system-wide dictionary.

To translate a given word using a specific dictionary, the following syntax is used:

Simple Translation: “::” + *RepositoryName* + “:” + *DictionaryName* + “:” + *Word*
Complex Translation: “::” + *RepositoryName* + “:” + *DictionaryName* + “:” + *String*
whereas each occurrence of “:” *x* “:” in *String* will be replaced by the translation of *x*.

Simple translations are expected to be the common case and are considerably more efficient than complex translations since there is no need to dynamically allocate memory. Essentially, simple translations transform a string into another string. This mechanism is, however, not generic enough to support special cases as for example the caption of a button that looks like “Undo (1)” whereas the number in the parentheses is dynamically computed. This caption can be translated using the complex translation “::RepositoryName:DictionaryName::tUndo: (1)”, i.e. complex translations make it possible to translate only parts of a string leaving other parts unchanged.

The API for performing a language translation is just one single procedure:

```
Repositories.Translate(CONST string : ARRAY OF CHAR, languages :
Localization.Languages) : Strings.String;
```

Note that the string which is returned is encoded in UTF8 and, more importantly, is considered to be immutable. The latter makes it possible to use such strings by reference which means that no additional dynamic memory has to be allocated in case of simple translations.

The component framework provides support for changing the language at runtime. For `StringProperty` instances, the translation is performed automatically by the framework, i.e. when the language is changed, each `StringProperty` is asked to perform a language translation if necessary. For other strings, the programmer is responsible to carry out the translation upon request. To make this possible, each `Window` and `Component` instance have a pre-defined procedure

`LanguageChanged(languages : Localization.Language)` that is called by the system whenever the user sets a different language.

1.2.4 Non-indexed Content

The fact that repositories come with a dual interface (Repositories API, Archives API) enables the user to store arbitrary data into repositories. This can be convenient for data as images, audio or text files where the creation and maintenance of repositories index entries would just be a burden for the programmer. Also, non-indexed data can be easily added, removed or modified by any operating systems and/or applications that support the archive file type in use.

1.2.5 API

Repositories can be created using the CreateRepository procedure. Creation means that an archive file and the index file within that archive file are created. To load a Repository, the procedure ThisRepository is used. Note here that Repositories are cached, i.e. ThisRepository only loads a repository if it is not already loaded. Therefore, a repository has to be explicitly unloaded when changes to the repository file shall become visible similar to Oberon modules.

```
PROCEDURE CreateRepository*(CONST filename : ARRAY OF CHAR; VAR res :
LONGINT);
PROCEDURE ThisRepository*(CONST name : ARRAY OF CHAR) : Repository;
PROCEDURE UnloadRepository*(CONST name : ARRAY OF CHAR; VAR res : LONGINT);

PROCEDURE PutComponent*(
   component : Component;
   CONST RepositoryName, componentName : ARRAY OF CHAR;
   VAR res : LONGINT);
PROCEDURE GetComponent*(
   CONST RepositoryName, componentName : ARRAY OF CHAR;
   refNum : LONGINT;
   VAR component : Component;
   VAR res : LONGINT);
PROCEDURE GetComponentByString(
   CONST string : ARRAY OF CHAR;
   VAR component : Component;
   VAR res : LONGINT);
PROCEDURE RemoveComponent*(
   CONST RepositoryName, componentName : ARRAY OF CHAR;
   refNum : LONGINT;
   VAR res : LONGINT);
PROCEDURE UnbindComponent*(
   CONST RepositoryName, componentName : ARRAY OF CHAR;
   refNum : LONGINT;
   VAR res : LONGINT);
```

To insert a component into a repository, the procedure `PutComponent` is used. This will create an entry in the index file as well as serialize the component into the repository. The component can later be retrieved using `GetComponent` or `GetComponentByString` or removed by using `RemoveComponent`. Unbinding a component is only relevant for single-instance components.

1.3 Models

Models encapsulate data and provide access methods to the data. This enables component implementations that operate on data and are not dependent on its representation.

The basic functionality of all models include

- Generic access methods supporting automatic format conversion
- Serialization
- Notification of observers

```
1  Wrapper* = RECORD END;
2
3  Model* = OBJECT(Repositories.Component)
4  VAR
5     onChanged- : WMEvents.EventSource;
6
7     PROCEDURE SetGeneric*(CONST w : Wrapper; VAR res : LONGINT);
8     PROCEDURE GetGeneric*(VAR w : Wrapper; VAR res : LONGINT);
9  END Model;
```

Figure 13: Model Interface

1.3.1 Example

To implement a model for a particular data type, the programmer needs to create a subclass of Models.Model that holds the data as field(s) and at least implements the methods SetGeneric and GetGeneric. Typically, an implementation will also provide some class-specific access methods that directly operate on the data type of the model.

```
1  Integer* = RECORD(Wrapper) value* : LONGINT; END;
2
3  IntegerModel* = OBJECT(Model)
4  VAR
5     PROCEDURE Set*(value : LONGINT);
6     PROCEDURE Get*() : LONGINT;
7     PROCEDURE Add*(value : LONGINT);
8  END Model;
```

Figure 14: Integer Model Interface

```
1  PROCEDURE Update(model : Models.Model);
2  VAR string : Models.String; res : LONGINT;
3  BEGIN
4     ASSERT(model # NIL);
5     model.GetGeneric(string, res);
6     IF (res = Models.Ok) THEN
7        KernelLog.String(string.value); KernelLog.Ln;
8     END;
9  END Update;
```

Figure 15: Model Usage Example

1.4 Wiring

Components can be customized by setting their properties and then be made persistent. But there must also be a way of expressing inter-component relationships and interactions. Currently, the framework offers several ways of doing so.

1.4.1 Hard-coded Wiring

The most tedious but also most flexible way to wire components is to hard-code the wiring. Since manual instantiation of graphical components is very cumbersome and takes up a lot of lines of code, the frameworks supports loading component compositions from Repositories and then wire them.

```
1  VAR
2     vc : WMComponents.VisualComponent;
3     button : WMStandardComponents.Button;
4  BEGIN
5     vc := WMComponents.GetVisualComponent("Repository:Component");
6     IF (vc = NIL) THEN
7        button := WMStandardComponents.FindButton("bold", vc);
8     END;
9  END Update;
```

Figure 16: Loading and Wiring Components

In this example, the component composite *Component* is loaded from the Repository *Repository. On* success, a button with uid *bold* is searched in the component composite *vc*.

This approach allows a partial separation of the graphical representation from the actual business logic. The representation can be created using the Component Editor Tool and stored into a Repository. Slight changes in the representation do not affect the business logic and does not require the corresponding modules to be recompiled. Nevertheless, the business logic does depend on some parts of the representation since it expects the presence of widgets with particular types and UIDs (in this example, a button widget with the UID *bold)*.

1.4.2 Reference Properties

A new property type `ReferenceProperty` has been introduced that encapsulates references to other components. The references are represented as string which makes it easy to serialize them.

```
ReferenceProperty* = OBJECT(Property)
```

```
    PROCEDURE Set*(component : Repositories.Component);
    PROCEDURE Get*() : Repositories.Component;
    PROCEDURE SetAsString(CONST string : ARRAY OF CHAR);
    PROCEDURE GetAsString(VAR string : ARRAY OF CHAR);
END ReferenceProperty;
```

Figure 17: Excerpt of ReferenceProperty Interface

References can be set as strings of the form *RepositoryName ":" ComponentName ":" ComponentID* or alternatively *"cmd:"* ModuleName "." CommandName, whereas "ModuleName.CommandName" is a command procedure that is supposed to generate a component.

Using reference properties, inter-component relationships can be made persistent. Typical examples of usage are widgets that may references to models.

1.4.3 Models

The ability to specify the data models to be used by a string allows models to be seen as part of the component interface.

1.4.4 Events

As shown in section TODO, subclasses of WMComponents.Component indicate event sources and listeners.

1.4.5 Automatic Wiring

For some components, it is possible to automatically create the wiring at instantiation time. An example of such a component is the tab component.

```
<Standard:TabPanel>
    <Standard:TabControl/>
    <Standard:Tab>
        … content of tab …
    </Standard:Tab>
    <Standard:Tab>
        … content of tab …
    <Standard:Tab>
    </Standard:Tab>
</Standard:TabPanel>
```

The user creates a tab panel. As next step, a TabControl has to be added to the composite. Adding actual tabs will cause the tab panel to wire them automatically so that the user can press on tab buttons to change the tab. As long as the business logic does not care about the selection of tabs, there is no need to additionally wire the tabs.

2 The Component Editor

The section provides a brief overview of the Component Editor Tool and its usage. The Component Editor consists of at least four windows which are explained in detail later:

- The GUI Editor window controls all other windows
- The Repositories Window displays currently loaded repositories and its components
- The Structure Window shows the hierarchical structure of the component composition in the currently active Edit Window
- The Properties Window can be used to edit the properties of the currently selected component

Additionally, there is one Edit Window per currently opened document where component editing actually takes place.

2.1 Editing

The editor has two modes: In edit mode, all components don't react to user input and can only be edited. In Use-Mode, the components cannot be edited but directly used and tested.

Select components
Components in the edit window can be selected directly in the edit window by left-clicking them or spawning a selection frame by holding the left mouse button down and moving the mouse. A right click selects the point of insertion. Components can also be selected in the Structure Window. A left click selects the component, a right click selects the point of insertion.

Add components
Exactly one component is selected as point of insertion all the time. When a component is added to the composition, it will be added as child of this component. To added a component to the point of insertion, the user can

- Select the component to be added in the Repository Window and then…
    - press the **Add** button
    - drag the component to the edit panel
    - enable the paint mode and spawn the component in the edit panel
- Press CTRL-V to insert the component of the clipboard (in present)

Remove components
To remove all currently selected components, press the Delete button or the "Delete" key.

Edit components
The selection frame can be resized and moved using the left mouse button. Per default, the frame snaps to the grid. To move or resized the selection frame pixel-wise, hold down the ALT key while moving/resizing. In the case that a single component is selected, its properties can be edited in the Properties Window.
To move the selected component to the front, press the ToFront button.

2.2 Main Window

The main window has control over all other windows. Closing it closes the Component Editor Tool.

Figure 18: Component Editor Main Window

The ***Open*** and ***Save as*** buttons are used to open and store documents. Pressing one of those buttons will display a dialog window asking for the filename of the document to be opened respectively stored.

The button ***Add*** adds the component currently selected in the Repositories Window to the current component composite at the insertion point.

Delete removes the currently selected component, whereas ***ToFront*** brings it to front**.**

GetXML shows the XML description of the component composite currently selected as insertion point.

Load loads the component currently selected in the Repositories Window and ***Store*** stores the component composite currently selected as insertion point into a repository.

Paint toggles between paint-mode and select-mode. In paint mode, the component currently selected in the Repositories Window can be drawn into the edit window.

Edit Mode toggles between edit-mode and use-mode.

The next two rows of buttons can be used to toggle the visibilities of some windows and elements:


Toggle visibility of the edit window


Toggle visibility of the repositories window


Toggle visibility of the structure window


Toggle visibility of the properties window


Toggle visibility of the snap grid. Note that snapping remains active independently of the visibility of the snap grid


Toggle visibility of the helper lines. Helper lines aid the designer to align components respective each other.


Toggle visibility of the component frames. If enabled, show the rectangular frame of all visual components. This is useful for arbitrary shaped components.

The next three lines display the current mouse cursor position (X, Y) in the edit window, the bounds of the current selection frame (left, top) to (right, bottom) as well as its width (w) and height (h).

2.3 Editing Window(s)

The Edit Panel windows show the visual representation of component composites. In edit mode, components can be selected and edited. In use mode, the component composite behaves exactly the same way it would in a real system.

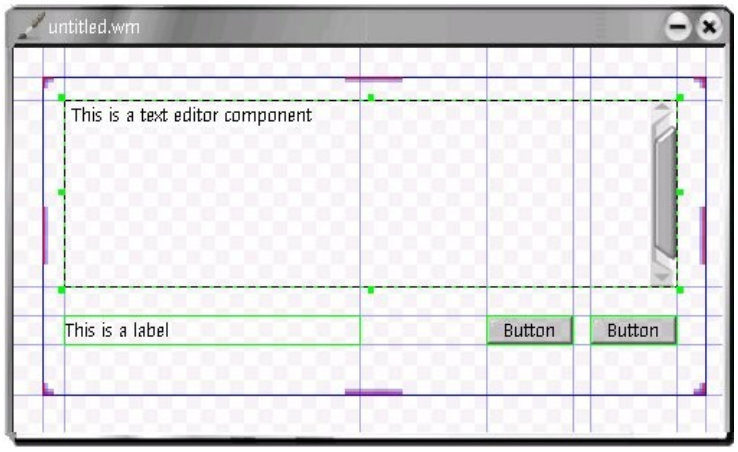

Figure 19: Component Editor Edit Window

The current insertion point is highlighted using a dark blue border (see Figure 19). The direct parent of the currently selected components (if any) is indicated using a red border.

2.4 Repositories Window

The Repositories window offers the same functionality as the standalone Repository Inspector tool (section 3.2 Repository Inspector). Additionally, it is used to select particular components that will then be added or loaded when pressing the Add respectively Load button in the main window. Components can also be dragged to the Edit Window.

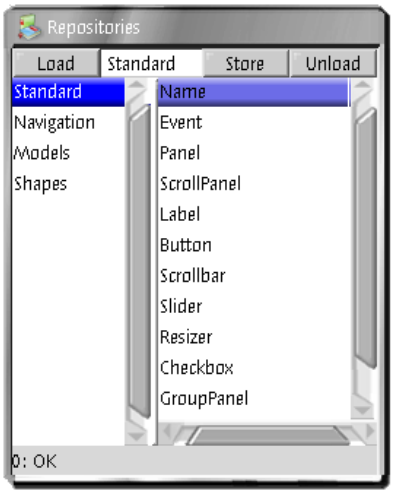

Figure 20: Component Editor Repositories Window

2.5 Structure Window

This window displays the structure of the component composite. A left-click selects the component. A right-click selects the component as place where new components are added to the composite.

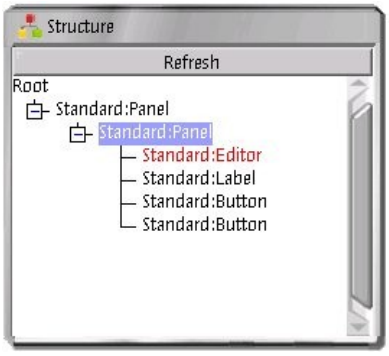

Figure 21: Component Editor Structure Window

The current insertion point is highlight by a blue background color. All selected components are represented using red text color.

2.6 Component Properties Window

The component properties window has four tabs. Standard Properties and Extended Properties display the properties of the currently selected component in an editable fashion. Event sources and event listeners currently only list the respective things.

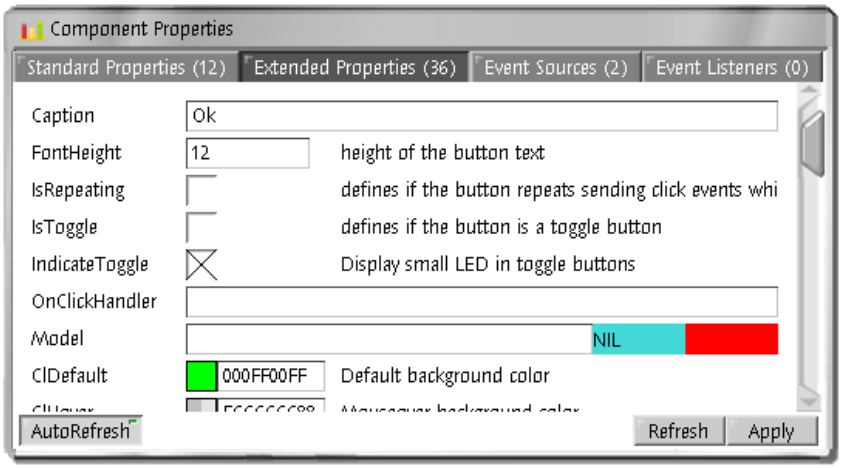

Figure 22: Component Editor Component Properties Window

2.7 Keyboard Shortcuts

CTRL-SPACE	Toggle paint mode
CTRL-C	Copy current selection to clipboard
CTRL-X	Cut current selection to clipboard
CTRL-V	Paste clipboard content
CTRL-A	Select all components
Cursor Keys	Move current selection frame one unit or one pixel (ALT held down)
Delete	Delete current selection

3 Other Tools

3.1 The Component Inspector

The inspector is mainly used as component testing and debugging tool. It displays a tree with all currently opened user windows and their component structure. For each component, it can show its properties and events. The properties can be edited and the current state of a component can be stored to a Repository.

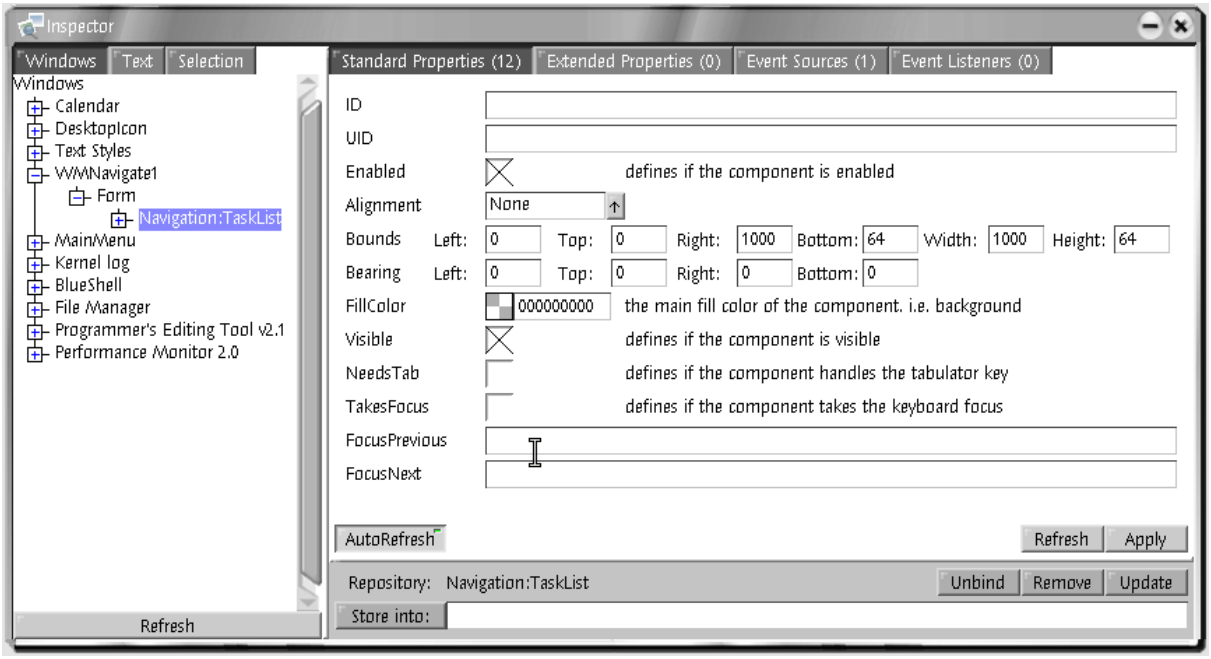

Figure 23: Component Inspector

3.2 Repository Inspector

The Repository Inspector essentially displays all currently loaded Repositories and their indexed content. It can be used to load Repositories and store the current state of Repositories. It is a testing and debugging tool. It is a testing and debugging tool.

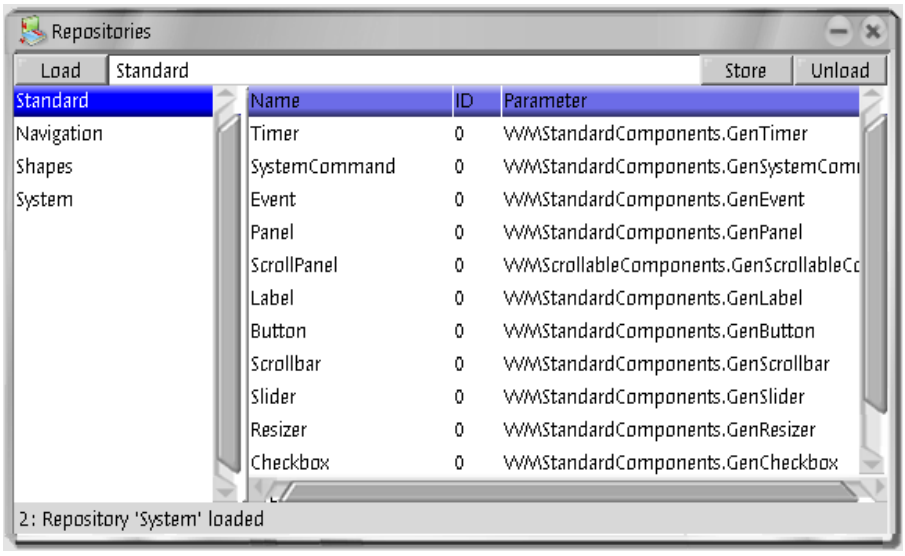

Figure 24: Repository Inspector

3.3 Code Generator

The command WMBuilderTransformer.Transform can be used to transform a component composition stored in a repository into a piece of Active Oberon code.

4 Examples

This section provides some examples.

4.1 Localization

This example demonstrates how an application with localization support can be implemented. The demo application is a window that contains a single button. The presence of a repository with the name Demo is assumed.

```
1  MODULE Demo;
2  IMPORT Repositories, WMWindowManager, WMComponents, WMStandardComponents;
3  TYPE
4    Window = OBJECT(WMComponents.FormWindow)
5
6      PROCEDURE &Init(width, height : LONGINT; alpha : BOOLEAN);
7      VAR b : WMStandardComponents.Button;
8      BEGIN
9        Init^(width, height, alpha);
10       NEW(b); b.alignment.Set(WMComponents.AlignClient);
11       b.caption.SetAOC("::Demo:Dic:Caption");
12       SetContent(b);
13     END Init;
14
15     PROCEDURE LanguageChanged(l : WMComponents.Languages);
16     BEGIN
17       LanguagesChanged^(l);
18       SetTitle(Repositories.Translate("::Demo:Dic:WindowTitle", l);
19     END LanguageChanged;
20
21   END Window;
22
23 PROCEDURE Open*;
24 VAR w : Window;
25 BEGIN
26   NEW(w, 320, 240, FALSE); WMWindowManager.DefaultAddWindow(w);
27 END Open;
28
29 END Demo.
```

In line 11, the caption property of the button is set to "::Demo:Dic:Caption". Remember that this is the syntax of a simple translation (see TODO). This string will be translated at initialization time and whenever the system language is changed without any further programming efforts. Line 18 is an example of how to explicitly handle language changes. The inherited procedure LanguageChanged will be called whenever the system language changes.

4.2 Wiring

This example shows how a externalized component composition can be wired to code.

```
1  MODULE Demo;
2  IMPORT Repositories, WMWindowManager, WMComponents, WMStandardComponents;
3  TYPE
4    Window = OBJECT(WMComponents.FormWindow)
5
6      PROCEDURE &Init(width, height : LONGINT; alpha : BOOLEAN);
7      VAR composition : WMComponents.Component;
8      BEGIN
9        Init^(width, height, alpha);
10       composition := WMComponents.GetComponent("Demo:Wiring");
11       IF composition # NIL THEN
12         SetContent(composition);
           b := WMStandardComponents.FindButton("ThisButton", composition);
           b.onClick.Add(HandleButton);
         END;
13     END Init;
14
15     PROCEDURE HandleButton(sender, data : ANY);
16     BEGIN
17       (* This procedure is called when the button is pressed *)
19     END HandleButton;
20
21   END Window;
22
23 PROCEDURE Open*;
24 VAR w : Window;
25 BEGIN
26   NEW(w, 320, 240, FALSE); WMWindowManager.DefaultAddWindow(w);
27 END Open;
28
29 END Demo.
```

5 References

Frey, T. (2005). *Bluebottle: A Thread-safe Multimedia and GUI Framework for Active Oberon* (1. ed., Vol. 6). Konstanz: Hartung-Gorre Verlag GmbH.

Muller, P. J. (2002). *The Active Object System: Design and Multiprocessor Implementation.* Zurich: Diss. ETH No. 14755.