Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Software-Komponentenmodelle
–
Seminar im Wintersemester 2006/2007
an der Universität Karlsruhe (TH),
Institut für Programmstrukturen und Datenorganisation,
Lehrstuhl für Software-Entwurf und -Qualität

Herausgeber:
Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, Ralf Reussner

Interner Bericht 2007-10

Universität Karlsruhe

Fakultät für Informatik

ISSN 1432 - 7864

http://sdq.ipd.uni-karlsruhe.de

Inhaltsverzeichnis

Open-Source Component Models: Bonobo and KParts

Sebastian Reichelt
Supervisor: Michael Kuperberg

Karlsruhe University, Karlsruhe, Germany

Abstract. Most modern desktop computers are controlled via graphical user interfaces. For different tasks, the user usually has to start separate programs with visually separated user interfaces. Complex tasks require either large monolithic programs or the possibility to reuse part of the functionality provided by one program in another. Such reuse can be achieved using component frameworks, which may be generic or specialized for graphical applications.
In Microsoft Windows, OLE is a widely known specialized component framework built on top of the generic COM framework. The open-source desktop systems GNOME and KDE support graphical components using the Bonobo and KParts subsystems, respectively. This paper is an analysis of these two open-source component frameworks.
Despite similar goals, Bonobo and KParts are designed and implemented very differently. In this paper, they are compared from a user's, programmer's, and system architect's point of view. The analysis includes the relation of Bonobo and KParts to COM and other frameworks, and to the general concept of component-based development. Finally the two frameworks are evaluated based on their goals, strong and weak points are identified, and the chances of unifying them are investigated.

1 Introduction

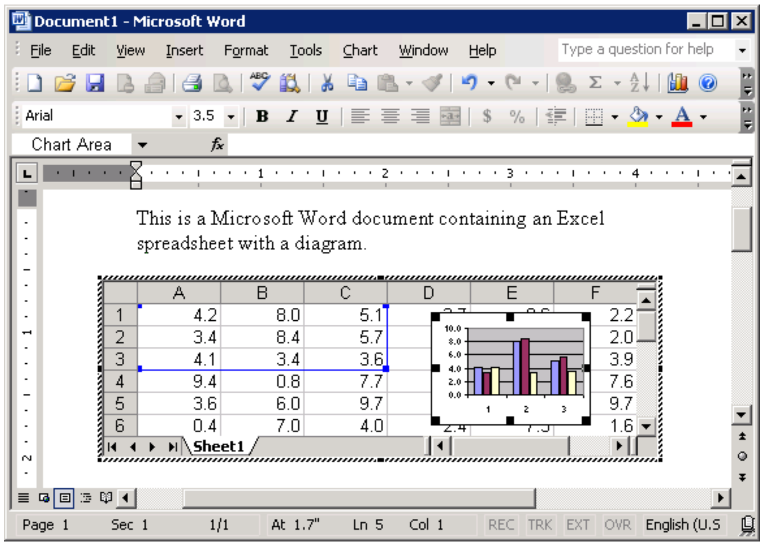
Construction of software from components has become a popular paradigm in software engineering. The idea behind software components is that large software products should be composed of smaller components that can be reused and combined in different ways [1]. A component may be an object or a collection of objects; the goal of component-based development is to achieve reusability for larger pieces of software than classes and objects.

GNOME and KDE are desktop environments for Unix-based systems such as Linux. A desktop environment can be described as software providing a graphical interface for users to launch programs and manage documents. Conceptually, programs are executed within the desktop environment, using its services for interaction with the user and with other programs.

In many operating systems, such as Microsoft Windows and Apple Mac OS, the functionality of a desktop environment is considered an integral part of the system; therefore the term "desktop environment" is not commonly used. These systems also provide other services for programs, including facilities for using parts of one program in another:

In Windows, this is achieved using a component framework called OLE (Object Linking and Embedding), based on COM (Component Object Model) [2]. COM is a specification of how interfaces can be defined, exported, and used; without any limitation on the purpose of those interfaces. A common use case is that interfaces provide access to objects which do not need to be part of the same program. OLE defines interfaces that enable programs to build an internal representation of compound documents, i.e. documents which contain other documents possibly maintained by different programs. Since these documents must be saved to and loaded from persistent storage including the contained documents, OLE also includes a certain form of persistence support. When a user edits a compound document, the user interfaces of the programs involved must be integrated; this is achieved through the use of OLE as well (see Fig. 1).

Fig. 1. Screenshot of a Microsoft Excel document embedded into Microsoft Word via OLE


Document1 - Microsoft Word
File Edit View Insert Format Tools Chart Window Help
Type a question for help
Arial
3.5
Chart Area
Sheet1
Page 1 Sec 1 1/1 At 1.7" Ln 5 Col 1 REC TRK EXT OVR English (U.S

This is a Microsoft Word document containing an Excel spreadsheet with a diagram.

	A	B	C	D	E	F
1	4.2	8.0	5.1	[illegible]	[illegible]	2.2
2	3.4	8.4	5.7			2.0
3	4.1	3.4	3.6			3.9
4	9.4	0.8	7.7			7.6
5	3.6	6.0	9.7			9.7
6	0.4	7.0	4.0	2.4	[illegible]	1.6

OpenDoc provided similar functionality in Mac OS, but this project has been abandoned [3]. However, interaction between full applications is very common in Mac OS, for example via scripting or drag&drop.

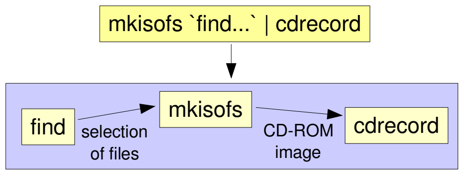
The Bonobo component framework is an attempt to implement features known from OLE in the GNOME desktop environment [4]. It is based on CORBA instead of COM for inter-process communication, using a GNOME-specific implementation. Bonobo mainly consists of a set of CORBA interfaces for compound documents, persistence, and integration of user interfaces. It also manages the association of MIME types with editors.

KParts is a KDE component framework designed to provide a subset of these features [5]. Components are defined as C++ classes, and always include a user interface implemented as a Qt widget with additional menu and toolbar information. Extensions supporting compound documents exist as part of KOffice. Association of MIME types with components is also supported, to permit loading components automatically as file viewers.

In Unix-based systems, desktop environments are optional; many programs written for one desktop environment can be executed in another, or without any desktop environment. The desktop environment services are often implemented using libraries, which can be loaded independently of the environment currently running. While applications designed for different environments can coexist, usually they cannot communicate with each other, since communication services are specific to the desktop environment.

Traditional text-based Unix programs are often combinable in a way that resembles the idea of software components. Many of them read data from the standard input stream, process it, and emit the result to standard output. The user can connect the output of one program to the input of another using pipes (see Fig. 2). If interfaces are incompatible, string manipulation programs are often used in between. Programs conforming to this scheme can be reused in different contexts, and combined to fulfill more complex tasks. However, compound documents are not covered by the pipe concept. Moreover, the concept does not apply to interactive programs, such as graphical applications.

Fig. 2. Usage of pipes in Unix



2 Comparison of Bonobo and KParts

Bonobo and KParts cover similar areas of software development. In this section, we will compare the ideas behind Bonobo and KParts, as well as the resulting architecture design.

2.1 Goals

In the official description of Bonobo [6], the GNOME developers declare the increasing complexity of programs as the main problem Bonobo is intended to solve. The GNOME project aims at keeping code simple. New functionality and features are seen as the reason for increasing complexity. A component architecture limits complexity by “reducing the amount of information a programmer needs to know about the system.” An implication is that new features do not have any impact on complexity if software is written as components.

As a secondary goal, components are expected to be reusable. The idea is that since components are defined by the interfaces they export and use, they can be combined to build larger applications.

In the KParts documentation [5], reusability is specified as the main goal. The issue of code complexity is not addressed. KDE developers focus on reusing functionality of one application within another, which is fundamentally different from the intentions of GNOME developers: writing simple reusable components to build large programs. Programmers following the KDE concept implement only those parts as components which are explicitly intended to be reused. In fact, KParts is designed specifically to ease extraction of components from existing component-less applications.

More detailed descriptions of the Bonobo and KParts design reveal that a “component” often refers to a graphical object. In Bonobo, one use case of components is the creation of compound documents, which usually means that an object which is editable on screen and printable is embedded into another. In the KParts documentation, a component is explicitly defined as “a widget, the functionality that comes with it, and the user interface for this functionality”. Thus the intended use of KParts is restricted to the embedding of user interfaces.

There are, however, several scenarios where the user interface of an applications is embedded into the interface of a different application. Both descriptions mention editors for compound documents (for example in office applications). In Bonobo, complex widgets (such as a chart plotter) are recommended to be implemented as components too, to achieve loose coupling between the application and the widget. In KParts, a specific scenario is a generic program (such as a file manager) that loads viewer components on demand to display files of different types.

A majority of GNOME and KDE features are clones of features of other desktop environments, including Microsoft Windows [7]. Traditionally, authors of open-source desktops have always included most of the features known to users. This leads to the hypothesis that the most compelling reason for the implementation of Bonobo and KParts was the feature of compound documents in Microsoft Windows, supported by the OLE technology which is based on COM. COM supports a much wider range of components than embeddable documents; it is an interface standard comparable to CORBA. Apparently Bonobo is intended to be a full counterpart to OLE, as the use of CORBA indicates. KParts, on the other hand, seems to be designed to support only the features of OLE that users get in touch with.

2.2 Relation to Other Concepts

Bonobo The official Bonobo website primarily relates Bonobo to the concept of pipes in Unix [6]. Especially for administrative tasks, complex applications are often built from simple programs connected via pipes. Bonobo applications can be connected using CORBA interfaces. Using the same CORBA interface, however, implies that applications are specifically designed to cooperate.

Furthermore, Bonobo is designed to provide features similar to Java Beans: Users can customize a component by selecting other components to provide the services it requires. Especially, Bonobo encourages that the components are explicitly designed for customization.

Finally, Bonobo is comparable to, and was inspired by, Microsoft OLE [4]. OLE was designed to provide features defined in CORBA for desktop systems, as well as additional capabilities not directly supported by CORBA, such as persistence. Bonobo uses CORBA and therefore inherits the CORBA features. The additional capabilities are defined by special CORBA interfaces. Like OLE, Bonobo also provides default implementations for these features.

KParts The KParts documentation mentions IBM/Apple OpenDoc and Microsoft OLE as frameworks for the creation of graphical components [5]. OpenDoc is aimed mainly at compound documents, which KParts provides basic support for. OLE contains a superset of the KParts features.

Bonobo is also mentioned explicitly. This indicates that KParts was developed as a response to Bonobo.

KParts is the successor of OpenParts, a similar component framework used in KDE 1. OpenParts used CORBA; it was replaced because of the high overhead on both speed and code complexity [8].

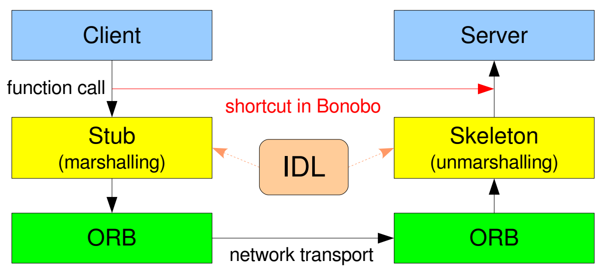
2.3 Design

Bonobo As mentioned before, Bonobo is based on CORBA [6], which is an industry standard for interoperation of software components [9]. In other contexts, CORBA is typically used to access remote objects over a network; Bonobo only deals with objects on the same computer, possibly implemented by different processes.

In CORBA, components are specified by interfaces using a specific IDL (Interface Definition Language). For the implementation of an interface, an IDL compiler generates "skeletons" from the IDL definition, which transform network requests from clients into function calls specific to the programming language used. Thus the implementation code is independent from the network transport, but must be written in the format expected by the skeleton code, as defined by the CORBA standard. Similarly, clients call interface methods by executing a language-specific CORBA-generated "stub", which performs all tasks necessary to access the possibly remote object (see Fig. 3).

The C and C++ language bindings of stub and skeleton code are the aspect of CORBA that influences Bonobo the most. Other parts of CORBA found

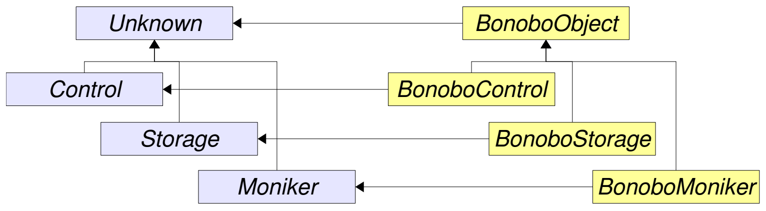
Fig. 3. CORBA architecture



in Bonobo are memory management and exception handling calls. Extended CORBA features, such as object-by-value and transactions, are not used.

The core of Bonobo is a set of CORBA interfaces to define components in general, which are not restricted to visual controls [10]. On top of these basic interfaces, Bonobo defines interfaces and implementations for advanced features such as component lifetime management, compound documents and persistence, and also for controls implemented as GTK+ widgets (see Fig. 4). A Bonobo component can extend these interfaces to define its own actions, in addition to regular GTK+ signals and so-called "property bags". As a help for programmers, Bonobo also contains code to hide the CORBA infrastructure behind the GTK+ toolkit, so clients can use components like normal widgets.

Fig. 4. Bonobo IDL interfaces and GTK+-based implementations



The GNOME project promotes the use of CORBA throughout all parts of the desktop [11,12]. Non-graphical components are essentially defined by regular CORBA interfaces; Bonobo adds only very basic features such as reference counting and interface querying. Therefore, in this paper, we will discuss only visual controls.

Since existing free implementations of CORBA such as MICO were too inefficient in accessing local components, the GNOME developers created a new implementation, called ORBit. According to the official website, it is CORBA 2.4-compliant, and supports IIOP, the standard protocol to access CORBA objects over TCP/IP networks [13]. Visual Bonobo controls, however, are not designed to be usable across networks.

To locate and possibly start the server implementing a particular component, clients usually employ OAF, the Bonobo Object Activation Framework [12]. Fed with an XML file containing the location of the server file and information on supported CORBA interfaces and MIME types, the OAF automatically selects the appropriate server. It transparently handles both shared libraries and executable programs implementing a CORBA server loop. Conceptually, each program implements only one single object, but this object can act as a factory creating other objects on demand, by implementing a factory interface.

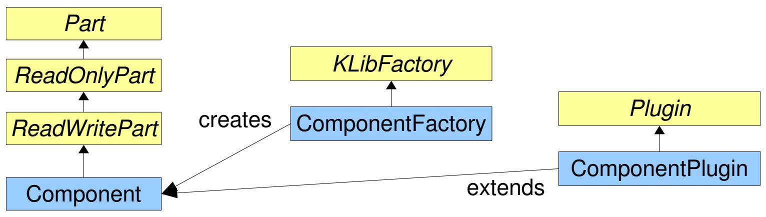
If a client wishes to access an object that already exists in a serialized form, such as a local file or an Internet resource, it does not request a new object of an appropriate type from the OAF, but utilizes a so-called moniker [10]. It passes the name of the object as well as the name of the requested interface to the moniker, which resolves the name, loads the object, and returns the interface. The moniker hides the specifics of loading the object, for example file and HTTP monikers exist. It interprets the name according to the type of moniker, e.g. as a file name or URL. The idea seems to be that the object already exists, and the component is only providing access to it in the form of an interface.

KParts The KParts design is fundamentally different from the Bonobo design, as components are defined as C++ classes instead of CORBA interfaces [5]. Since C++ classes can be used only within the same process, KParts are always implemented as shared libraries.

Shared libraries are the common Unix solution to code reuse. Programs can link to shared libraries either at build time (which implies that the file names of the libraries are fixed) or at execution time. Shared libraries export interfaces in the form of function prototypes (usable and implementable in C or any compatible language) or abstract classes (usable and implementable only in C++). Definition and use of interfaces would already qualify shared libraries as components, if there was a standard procedure for gluing together components with compatible interfaces. Instead, most libraries explicitly define the other libraries they depend on, and programs link to libraries at build time.

The aspect that differentiates a KPart from a regular library is that it exports classes which inherit from a specific abstract 'Part' class. The Part class defines what a KPart is: An object containing a Qt widget. Qt is the widget toolkit used by KDE, like GTK+ in GNOME. By convention, the widget constituting a part is set within the constructor of the class, by calling a method implemented by the abstract Part class. Another method is used to define the actions implemented by a component, and to embed appropriate menu and toolbar items in the surrounding window. They are defined in an XML file; the method connects them to so-called "slot" methods of the part. The GUI of any application using

Fig. 5. KParts architecture



the part must be defined as an XML file as well, which is optional otherwise. Merging of menus and toolbars is handled by merging the XML files.

KParts also defines two abstract subclasses of the 'Part' class, which the component may inherit from instead: 'ReadOnlyPart' or 'ReadWritePart'. They contain additional abstract methods to load the contents of the widget from a file, and to save them, respectively; the component must provide a custom implementation for these methods.

By exporting classes derived from one of these three abstract classes, a library becomes a component because applications can use it without explicitly depending on it. The code using the class refers only to one of the abstract classes, and not to the specific widgets or actions used to implement the component. Therefore, however, the domain of components is restricted to Qt widgets with additional actions.

Instead of linking directly to a library, an application may support all available components by asking KDE to create a KPart instance implementing a specific MIME type [7]. For this to work, the library must be registered with KDE by providing a "desktop file" defining the capabilities of both applications and KPart libraries. This feature is similar to the OAF in GNOME, but it is restricted to the ReadOnlyPart class, whereas OAF can handle all Bonobo interfaces. The user may set the preferred applications and parts to use when viewing or editing files of a specific MIME type, choosing from a list of available applications. Therefore components may be exchanged and glued together even by users.

Since clients of dynamically loaded shared libraries can access only functions with a known signature, but cannot determine exported classes or even create objects of classes unknown at compile time, KPart libraries intended to be loaded dynamically must implement one single function returning a factory object. The factory object inherits from an abstract factory class defined by KParts, and implements a method to create a KPart object. The parent widget to embed the component into is passed to the method as a parameter.

KParts can be extended using so-called "plugins", which define additional actions operating on the component [14]. In a way, plugins take on the role of variation points in the context of component-based development. However, they are developed specifically for one single part; the interface between a part and the plugin is the C++ class defining the part. Parts do not need to define specific interfaces that plugins connect to. Therefore, even though plugins can be used to construct software from separate pieces, plugins are not components.

2.4 Example

Web Audio Store The common “Web Audio Store” example of this seminar is not applicable to these two component frameworks, as the audio store is a web-based application whereas GNOME and KDE are desktop systems.

Specifically, KParts is restricted to graphical components containing Qt widgets, as defined by the ‘Part’ class as the root of all components. None of the components constituting the web server are graphical, and the interfaces do not deal with graphics in any way. The web browser is usually a graphical program, but its internals are not relevant in the example. One aspect that may be mapped to KParts is the encoding adapter filling a variation point. If the entire application was graphical, such a feature could be implemented as a plugin (as described above): If the audio store KPart contained a method to set the database adapter, a plugin could define an action calling this method with the encoding adapter.

Since Bonobo is a more general framework, many parts of the example could be implemented in Bonobo. All interfaces except for the web user interface would be defined in CORBA, as extensions of the basic Bonobo ‘Unknown’ interface. Network communication is supported, but for that purpose the client would need to be a CORBA application, instead of a web browser using HTTP. Components would be libraries or individual programs, each implementing one or more CORBA interfaces. Construction of the entire application from individual components could be performed manually by loading libraries or starting programs, obtaining references to the components, and passing them to other components as parameters in the interface methods. Another option would be to register all components with OAF, and executing OAF queries for the respective services when initializing the application. This would keep variation points configurable through OAF.

However, such an implementation of the audio store would actually be based mainly on CORBA alone; the contribution of Bonobo would be minimal. The example does not require any extended features of Bonobo such as persistence (since that is handled by the SQL server). The component framework defined by CORBA is the subject of another paper in this seminar. Moreover, the implementation could not be used to compare Bonobo to KParts. Therefore we chose to use a different example instead.

Minimal Example To compare how Bonobo and KParts are used from a programmer’s point of view, we have written a small example component in both frameworks. We have chosen to reduce the example to the minimal code necessary to specify a component, to analyze only the complexity introduced by the component frameworks. Therefore the component exposes just a simple label widget.

The Bonobo version was derived from the example included in the Bonobo source code. It is implemented as a C program; therefore it needs to contain a main function. Bonobo defines a C macro to make this easier:

```
BONOBO_ACTIVATION_FACTORY
    ("OAFIID:Bonobo_Sample_Factory",
     "Bonobo Sample Factory",
     VERSION,
     control_factory,
     NULL)
```

The resulting main function turns the program into a factory for a Bonobo component, using the factory function 'control_factory', implemented as follows:

```
BonoboObject *control_factory (BonoboGenericFactory *this,
                               const char *component_id,
                               void *data)
{
    g_return_val_if_fail (component_id != NULL, NULL);

    if (!strcmp (component_id, "OAFIID:Bonobo_Sample"))
        return bonobo_sample_control_new ();

    return NULL;
}
```

The factory function receives an id string defining the component to be created. It must compare this string against supported component names, and then return a pointer to a 'BonoboObject', which is a class implementing the basic features of the 'Bonobo::Unknown' interface, namely reference counting and interface querying.

Here, the actual object creation is extracted into a function:

```
BonoboObject *bonobo_sample_control_new (void)
{
    GtkWidget *widget;

    widget = gtk_button_new_with_label ("Sample Text");
    gtk_widget_show (widget);

    return BONOBO_OBJECT (bonobo_control_new (widget));
}
```

At first, we create a GTK+ widget, which is a simple label in our case. Then we invoke a function provided by Bonobo to create an object implementing the 'Bonobo::Control' interface, which takes a widget as an argument.

There is no need to define a custom interface for the control; the resulting program contains all of the functionality necessary to embed the widget into another GTK+ application. New interfaces can be created as CORBA IDL files and processed by the 'orbit-idl' program to create stub and skeleton code.

To register the component with OAF, an XML file must be placed in the OAF directory:

```
<oaf_info>

<oaf_server iid="OAFIID:Bonobo_Sample_Factory" type="exe"
            location="/path/to/program">
    <oaf_attribute name="repo_ids" type="stringv">
        <item value="IDL:Bonobo/GenericFactory:1.0"/>
    </oaf_attribute>
    <oaf_attribute name="name" type="string"
                   value="Bonobo Sample Factory"/>
    <oaf_attribute name="description" type="string"
                   value="Factory for the Bonobo sample control"/>
</oaf_server>

<oaf_server iid="OAFIID:Bonobo_Sample" type="factory"
            location="OAFIID:Bonobo_Sample_Factory">
    <oaf_attribute name="repo_ids" type="stringv">
        <item value="IDL:Bonobo/Unknown:1.0"/>
        <item value="IDL:Bonobo/Control:1.0"/>
    </oaf_attribute>
    <oaf_attribute name="name" type="string"
                   value="Bonobo Sample Control"/>
    <oaf_attribute name="description" type="string"
                   value="Bonobo control wrapping a label"/>
</oaf_server>

</oaf_info>
```

The component could be embedded dynamically into file managers and browsers if it implemented a Bonobo interface to load data from a file. In that case the OAF file would also contain the MIME types the component supports, so the client can use an appropriate OAF query to locate the component, or transparently load the component using monikers.

The KParts example was derived from the KParts tutorial [5]. In KParts, the core of the component is the class definition:

```
class SamplePart: public KParts::Part
{
    Q_OBJECT
public:
    SamplePart (QWidget *parent, const char *name = 0L);
    virtual ~SamplePart () {}
};
```

The class can also contain additional methods and fields. Typically the widget will be included as a private field, since the methods need to access it.

The constructor implements the creation of the widget as a child of the widget passed as an argument:

```
SamplePart::SamplePart(QWidget *parent, const char *name)
    : KParts::Part(parent, name)
{
    setInstance (new KInstance("samplepart"));

    setWidget (new QLabel ("Sample Text", parent));

    setXMLFile ("sample_part.rc");
}
```

The XML file 'sample_part.rc' usually defines the actions to be merged into the menus and toolbars of the main window. In this case, there are none:

```
<!DOCTYPE kpartgui>
<kpartgui name="SamplePart" version="1">
    <MenuBar/>
    <StatusBar/>
</kpartgui>
```

If the component is to be loaded component dynamically, it must be implemented as a shared library exporting a factory object:

```
extern "C"
{
    void *init_libsample ()
    {
        return new SampleFactory;
    }
}
```

The 'SampleFactory' class is an implementation of the abstract 'KLibFactory' class:

```
class SampleFactory: public KLibFactory
{
    Q_OBJECT
public:
    SampleFactory (QObject *parent = 0, const char *name = 0);
    ~SampleFactory ();

    virtual QObject *createObject (QObject *parent,
                                   const char *name,
                                   const char *classname,
                                        const QStringList &args);

    static KInstance *instance();

private:
    static KInstance *s_instance;
};
```

The ‘createObject’ creates an instance of the component:

```
QObject *SampleFactory::createObject (QObject *parent,
                                      const char *name,
                                      const char *,
                                      const QStringList &)
{
    if (parent && !parent->inherits ("QWidget"))
    {
        kdError()
            << "SampleFactory: parent does not inherit QWidget"
            << endl;
        return NULL;
    }

    SamplePart *part = new SamplePart ((QWidget *) parent, name);

    emit objectCreated (part);
    return part;
}
```

To be embeddable in a file manager or browser, the component would need to inherit from ReadOnlyPart instead of Part, implement the abstract ‘openFile’ function, and provide a desktop file describing supported MIME types.

The comparison shows that the amount of additional code needed in Bonobo and KParts is approximately equal. In KParts, custom class definitions are required, whereas in Bonobo, existing classes can be used – unless custom interfaces must be implemented. On the other hand, the C++ code used in KParts is a little more readable, as the C code used in Bonobo contains macro calls for simple operations such as downcasting object pointers.

3 Evaluation

In this section, we will determine whether the Bonobo and KParts frameworks meet their goals, how successful the projects are in general, what domains they are applicable to, and what their future may look like.

3.1 Achievement of Goals

Bonobo The primary goal of Bonobo is managing the increasing complexity of applications. Specifically, if additional features are added to a program, the source code should not become more complex. A programmer who is unfamiliar with the code should be able to extend the application acquiring as little knowledge as possible.

In this sense, Bonobo does not add a lot of functionality to CORBA. Managing complexity is already a goal of the CORBA framework. The usual circumstances of using CORBA, however, are that applications are already naturally separated into components by running on different computers. CORBA was designed mainly as a standard for communication between such components [9]. For this purpose, it is widely used in industry scenarios and has proven to be well-suited. The step from defining interfaces for existing components to separating an application into components by defining interfaces is not very far. Therefore, in theory, the use of CORBA alone could be seen as a good guarantee to manage complexity well.

The benefit gained from separating applications into components could be spoiled by adding complexity to the internals of all code. CORBA, while leading to less complex code than explicit network communication, is known for complicated constructs related to exception handling, string values, and for the problem of being tied to CORBA data types [9]. Bonobo is able to hide these constructs behind GTK+ wrapper classes for controls, but when introducing custom interfaces, programmers have to deal with the complexity. Moreover, the multiple layers of wrappers have separate reference counting mechanisms which must be kept synchronous [15].

The goal of reusability is achieved very well by OAF, as components can be selected based on the CORBA interfaces they provide. Compound documents provide reusability for users; in Bonobo, they can include graphical as well as non-graphical components, so users can freely choose between alternatives at all levels. User configuration of preferences for the selection of components does not seem to be supported yet, but supposedly such a feature could be added easily.

As a conclusion, Bonobo achieves its goals almost as well as OLE does. More work seems to be necessary in the field of persistence, as the required interfaces are apparently not stable yet. Transaction processing is planned, which is important to be compliant with the CORBA standard [16].

Bonobo components, however, are not as flexible as Unix command-line programs connected via pipes. Using pipes, a user can connect any two programs that read from the standard input stream and/or write to the standard output stream. If they use incompatible data formats, string manipulation is often sufficient to convert the data. In Bonobo, programs can only be connected if they use the same CORBA interfaces. Special programs are necessary to convert data between incompatible interfaces.

Moreover, using CORBA for the entire feature set of GNOME is an ongoing mission. It can be successful only if the interfaces for all components are standardized and included in the Bonobo specification, so all applications can rely on the ability to reuse components based on these interfaces [16].

KParts The goal of embedding user interfaces of customized viewers into generic programs is achieved completely, with support for user preferences and rather easy extraction of user interfaces from applications. There is a dispute about whether such user interfaces should include editors or just viewers [7]. Currently only viewers are supported for safety reasons, but the concept does not exclude editors.

Less support exists for compound documents. KOffice documents can be embedded into each other with the help of KParts, but this is a KOffice-specific extension [5]. KParts does not deal with issues such as persistence that arise from the support of compound documents. However, it is likely that the KOffice embedding framework can be used outside of KOffice, too.

Ultimately, reusability was the main goal of KParts. This goal is not really met: Reuse can happen only if the code to be reused exports all of its functionality through an interface. In KParts, such an interface is defined as a C++ class. The framework that creates components dynamically, however, cannot differentiate between components with different functionality; it only operates on the level of embedding widgets and menus. An application cannot, for example, request a function plotter exporting an interface specifically for defining function data. It can only link to the component at build time, but in that case there is no additional benefit over a regular shared library. For proper reuse of components, a framework would need to support connecting components based on their interfaces.

3.2 Acceptance

Neither Bonobo nor KParts have any competition in their environments, which can be regarded as an indication for wide acceptance by the open-source community. KParts was the result of a long discussion about possible alternatives, and was adopted very quickly, which shows that it is well-accepted [7].

The need to divide software into components, however, does not seem to be equally important to all developers. Large monolithic programs are still the common case in both KDE and GNOME. Monolithic code is often easier to write because different parts of a program can be referenced directly, but maintainance is usually more difficult.

3.3 Applications

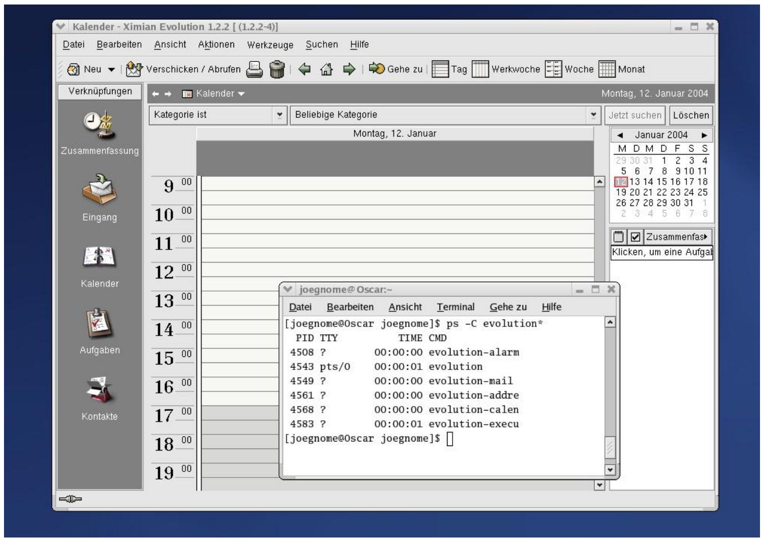
These are examples of applications using Bonobo or KParts.

Bonobo

– The Nautilus file manager uses Bonobo internally to separate its views [17].

- Nautilus can be extended with viewers for file types.
- Gnumeric embeds charts and other objects in spreadsheets via Bonobo [6].
- Evolution is a program comprising separate Bonobo components for mail, address book, calendar, etc. [18] (see Fig. 6).
- The Mozilla web browser can embed Bonobo controls in web pages, and is embeddable itself, on systems where Bonobo is supported [19].
- GNOME-DB is a database framework based on Bonobo [20]
- OpenOffice developers are working on Bonobo support [21].

Fig. 6. Screenshot of Bonobo use in Evolution [22]



```
[joegnome@Oscar joegnome]$ ps -C evolution*
  PID TTY          TIME CMD
 4508 ?        00:00:00 evolution-alarm
 4543 pts/0    00:00:01 evolution
 4549 ?        00:00:00 evolution-mail
 4561 ?        00:00:00 evolution-addre
 4568 ?        00:00:00 evolution-calen
 4583 ?        00:00:01 evolution-execu
[joegnome@Oscar joegnome]$
```

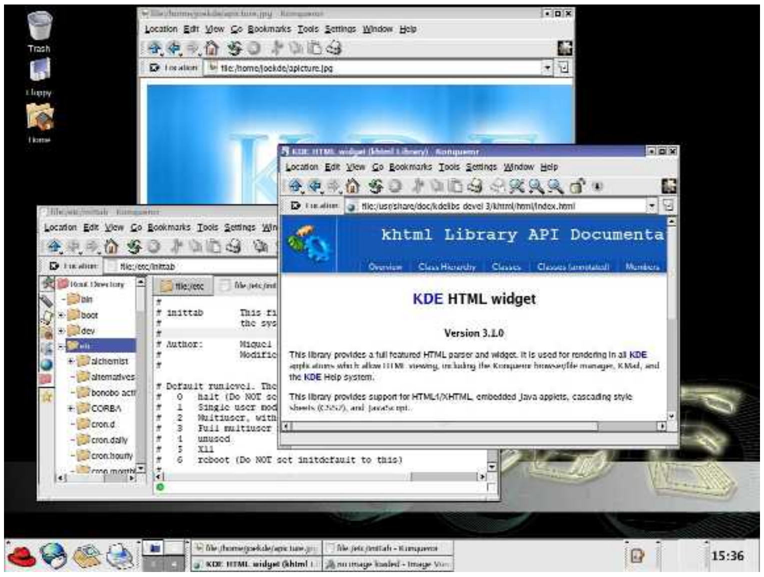
KParts

- The Konqueror file and web browser can view the contents of files using viewer components [7] (see Fig. 7).
- KOffice contains a generic shell application, and supports compound documents.
- KDevelop embeds a selectable GUI editor component [23].

3.4 Documentation

Both Bonobo and KParts are well documented. A lot of tutorials with examples exist. The reference documentation sets contain overviews of the class hierarchies with extensive descriptions.

Fig. 7. Screenshot of KParts use in Konqueror [22]


Trash
Floppy
Home
Location Edit View Go Bookmarks Tools Settings Window Help
file:/home/jockde/apicture.jpg
KDE HTML widget (khtml Library) - Konqueror
file:/usr/share/doc/kdelibs devel 3/khtml/html/index.html
khtml Library API Documenta
Overview
Class Hierarchy
Classes
Classes (annotated)
Members
file:/etc/inittab
Root Directory
bin
boot
dev
etc
alchemist
alternatives
bonobo acti
CORBA
cron.d
cron.daily
cron.hourly
15:36

KDE HTML widget

Version 3.1.0

This library provides a full featured HTML parser and widget. It is used for rendering in all KDE applications which allow HTML viewing, including the Konqueror browser/file manager, KMail, and the KDE Help system.

This library provides support for HTML4/XHTML, embedded Java applets, cascading style sheets (CSS2), and JavaScript.

```
#
# inittab       This fi
#               the sys
#
# Author:       Miquel
#               Modific

# Default runlevel. The
#   0 - halt (Do NOT se
#   1 - Single user mod
#   2 - Multiuser, with
#   3 - Full multiuser
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
```

3.5 Tool Support

For the CORBA layer of Bonobo, the IDL compiler generates stub and skeleton code, but GTK+ wrapper classes need to be written by hand. For graphical controls, custom IDL interfaces are usually not necessary, and the task of wrapping a widget into a Bonobo control is simple enough to do without a tool. Embedding such controls in client applications should be possible using the GUI editor Glade; however, its Bonobo support was never finished [24].

For KParts, there exists a tool to create a template for a component along with a "shell" application that embeds this component [25]. Qt Designer, the most frequently used GUI editor for KDE applications, does not natively support KParts, as it is related only to the Qt widget set, but not to KDE. KFormDesigner, a KDE-specific editor under development, can be assumed to support KParts, since it is implemented as a KPart itself [23].

3.6 Problems

In CORBA, performance of method calls is a severe bottleneck. Marshalling and unmarshalling method parameters and transferring them over a network takes a non-trivial amount of time. OpenParts, the predecessor of KParts, which relied on CORBA for communication, was abandoned because of its poor performance. However, the GNOME developers claim that the performance overhead of ORBit is almost as low as the overhead of virtual functions in C++ if both components are running in the same process [13]. This is achieved by bypassing the marshalling and unmarshalling, and calling the implementing function directly [26]. Even in this case, there can be an implicit overhead because data structures cannot be accessed directly.

Bonobo attempts to hide CORBA interfaces behind special data structures. These data structures are different on the client and server. In other words, the code on the client side and on the server side are connected only through CORBA and Bonobo, and very independent in other aspects. Instead, they are both tied to different parts of the Bonobo framework. Such Bonobo-specific artefacts reduce code readability and introduce a permanent dependency on Bonobo.

At first sight, the most significant limitation of KParts seems to be its confinement to the C++ language. A closer look reveals that the entire KDE desktop carries this restriction, since it is imposed by the Qt widget set. C++ has a very rich feature set and a complicated binary interface, so code written in other languages cannot easily be combined with C++ code. Even different compilers and different versions of the same compiler can have incompatible C++ binary interfaces [25]. On the contrary, the standard C binary interface is commonly supported by other languages, so other language bindings based on the C implementation can be implemented for Bonobo [27].

3.7 Generality

While the KParts framework is heavily restricted to the Qt toolkit, parts of Bonobo could be applicable to a wider range of software. The restriction to a toolkit also implies that KParts is not portable beyond KDE. Bonobo is explicitly designed to be independent of the desktop environment and the X Window System [20].

If it is at all possible to unify Bonobo and KParts, the result would have to be derived from Bonobo. All graphical aspects of Bonobo are related to the GTK+ toolkit; different interfaces would be required to be able to use Qt instead. Furthermore, C++ support of Bonobo would need to be improved, to provide C++ wrapper classes for KDE programmers. The effort may pay off, perhaps non-graphical components such as database engines could be used from both GNOME and KDE in the future.

4 Conclusion

Both Bonobo and KParts are successful solutions to the requirements of document embedding and modern file or web browsing. To some extent, they support code reuse and construction of complex applications from simpler parts. Bonobo meets this goal almost as well as OLE does, but adds the complexity of the CORBA layer. KParts provides little more functionality than user interface embedding.

For the time being, users of open-source desktop environments still have to cope with large monolithic programs where all features are built in, instead of extensible software built from small components. The benefits of the latter approach are obvious and widely agreed upon in the field of software development: Ultimately, it leads to faster innovation, as it takes a lot less effort to build a component than to integrate a new feature into one or more existing programs. Bringing innovation back to Unix is a major motivation for the GNOME project [16]; Bonobo is definitely a step in the right direction. However, free software developers usually choose the most convenient way of implementing their projects. In most cases, this means traditional component-less development.

How can GNOME or KDE designers solve this issue? In our opinion, first of all writing and using components must become as simple as possible. For GNOME, a new programming language with integrated support for Bonobo is an option, but would require a lot of work. Full integration of Bonobo with C++ is a more realistic alternative, and it would even be usable in KDE as well.

However, even the simplest component-based development methods cannot be any simpler than component-less programming, at least as long as there are no reusable components to build software with. The only way to overcome this barrier might be to abandon the concept of programs and build the entire desktop environments from components. If the developers manage to take this step, open-source desktop environments could be the first systems that are entirely component-based. That would be an important advantage over all other systems currently available.

References

1. Reussner, R.H.: Component based software engineering - lecture 1 (2006)
2. Wikipedia: Object Linking and Embedding `http://en.wikipedia.org/wiki/Object_Linking_and_Embedding`.
3. Wikipedia: OpenDoc `http://en.wikipedia.org/wiki/OpenDoc`.
4. Binnema, D.J.C.: The art of writing a Bonobo control (2001) `http://www.djcbsoftware.nl/projecten/bonobo-controls/bonobo-controls.html`.
5. Faure, D.: KDE 2.0 development: Creating and using components (KParts) `http://developer.kde.org/documentation/tutorials/kparts/`.
6. GNOME Developer's Site: Bonobo `http://developer.gnome.org/arch/component/bonobo.html`.
7. Bölöni, L. In: Programming KDE 2.0. C M P Books (2001) 201–222
8. Fremy, P.: KDE technology: KPart components `http://phil.freehackers.org/kde/kpart-techno/kpart-techno.html`.
9. Wikipedia: Common Object Request Broker Architecture `http://en.wikipedia.org/wiki/CORBA`.
10. GNOME: Bonobo API Reference Manual. `http://developer.gnome.org/doc/API/bonobo/book1.html`.
11. Leskien, H.: Das GNU-Desktop-System GNOME (1999) `http://www.fh-wedel.de/~si/seminare/ws99/Ausarbeitung/gnome/gnome0.htm`.

12. Meeks, M.: Introduction to Bonobo (2001)
`http://www.ibm.com/developerworks/webservices/library/co-bnbo1.html`
`http://www.ibm.com/developerworks/webservices/library/co-bnbo2.html`
`http://www.ibm.com/developerworks/webservices/library/co-bnbo3.html`.
13. GNOME: ORBit2
`http://www.gnome.org/projects/ORBit2/`.
14. KDE: kparts: Framework for KDE graphical components.
`http://api.kde.org/cvs-api/kdelibs-apidocs/kparts/html/`.
15. Pryor, J.: Bonobo component architecture (2002)
`http://courses.cs.vt.edu/~cs5204/fall02/ProjectArchive/PryorReport.pdf`.
16. Meeks, M.: Bonobo and free software GNOME components. In: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001) 607–619
17. Warkus, M.: Neues in der Verwaltung. Linux Magazin (11) (2000)
`http://www.linux-magazin.de/Artikel/ausgabe/2000/11/Nautilus/nautilus.html`.
18. Ganslandt, B.: Gnomogram. Linux User (1) (2002)
`http://www.linux-user.de/ausgabe/2002/01/018-gnomogram/gnomogram17.html`.
19. Savannah: mozilla-bonobo browser plugin
`http://www.nongnu.org/moz-bonobo/`.
20. Binnema, D.J.C.: Bonobo components: Architecture and application (2001)
`http://www.djcbsoftware.nl/projecten/nluug-bonobo/`.
21. OpenOffice.org: Projekte
`http://de.openoffice.org/about-ooo/about-projects.html`.
22. Bornhold, J., Hoyer, M.: Open Source Komponentenmodelle - seminar slides
23. KDE-Apps.org: KFormDesigner
`http://kde-apps.org/content/show.php?content=14796`.
24. GNOME: Glade
`http://glade.gnome.org/`.
25. Brucherseifer, E., Welwarsky, M.: Wiederverwendung in KDE. iX (2) (2002) 113+
26. Lebl, G.: GNOMEnclature: Intro to Bonobo (2000)
`http://www-128.ibm.com/developerworks/library/l-gn-cor/index.html`.
27. Ruiz, D.S., Lacage, M., Binnema, D.J.C.: GNOME & CORBA. (2000)
`http://developer.gnome.org/doc/guides/corba/html/book1.html`.

Seminar: The Palladio Component Model

Erik Burger[1]

Chair for Software Design and Quality (SDQ), Universität Karlsruhe(TH)
Erik.Burger@ira.uka.de

Abstract. Performance predictions and statements about the Quality of Service at an early stage of a component-based modeling process require the support of QoS attributes by the underlying component model. Based on the component model in UML 2.0, the Palladio Component Model defines a software development process that includes user roles, development workflows and the component model itself, enriched by principles for predicting QoS. Service Effect Specifications and stochastical means are used to make predictions that adapt to the various contexts in which a system is used after deployment.
In this paper, the principles of the PCM will be presented, and a short evaluation of its current state of development will be given.

1 Introduction

The *Palladio Component Model (PCM)* is a project that is currently being developed at the Chair for Software Design and Quality at the University of Karlsruhe, Germany. It focuses on the integration of *Quality of Service (QoS)* concept into *component-based software engineering (CBSE)*, enabling performance predictions and guarantees based on various techniques.

The specification of the Palladio Component Model contains not only technical concepts, but also relies on a process model for software development that integrates QoS considerations into the workflow of software projects, introducing individual user roles for the development process.

In the course of this seminar paper, the background of component-based software engineering will be shortly introduced, then, the ideas of QoS will be described. After an introduction into the concepts that the PCM is based on, its structure will be presented. In an attempt to evaluate and classify the PCM, its user-friendliness and tool support will be discussed. Finally, the common modelling example of the WebAudioStore will be applied to the PCM.

2 Background

2.1 The History of Component Based Software Modeling

The idea of components in software development is even older than the idea of object-oriented programming. It was first adressed during a NATO conference in 1968 by Douglas McIlroy [1]. There, the componentization of software is introduced as a means of managing the growing complexity of software projects. The main aspect was the demand that prefabricated components be used to build software, so that the software industry would profit from a new market of software components.

Since then, many concepts of using pre-existing software have been developed, not least because object-orientation has become omnipresent in software engineering.

2.2 Components in UML 2.0

Since the Palladio Component Model has a notation that is close to the notation of components in UML 2.0, it is useful to take a look at the specification of component diagrams in the latter standard.

In UML at version 2.0, *components* are introduced as a language unit and thus as an integral part of the standard. They are put into the context of the MOF metamodel, describing them as units that encapsulate the state and behavior of a number of classifiers [2, sect. 8.3.]. This concept of encapsulation is also used to hide the inner details of a component and is called *black boxing*. The metaclass definition of a component can be seen in Fig. 1.

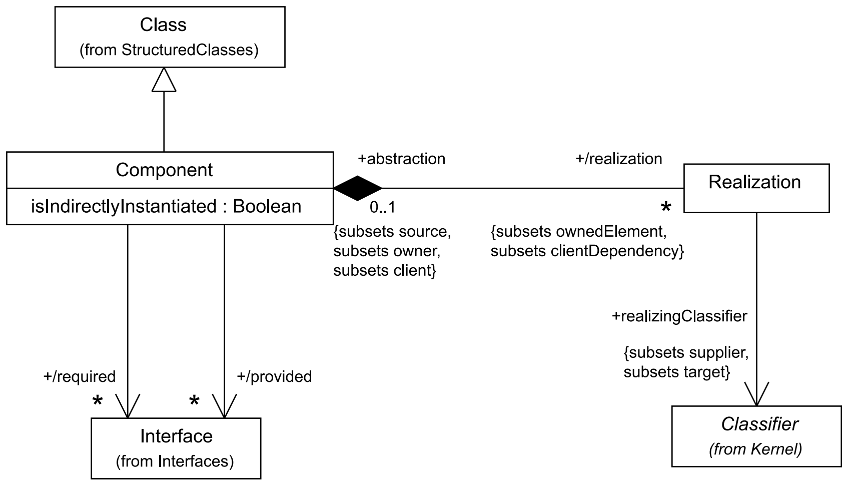


Fig. 1. Metaclass definition of components in UML 2.0 (from [2])

UML defines the semantics of components and the structure of component-based modeling as a complete concept. Since we are interested in the Palladio Component Model, we will only focus on the notational definitions, since UML is a widespread modeling language, and its component model is the subject of another paper of this seminar.

Interfaces

Fig. 2 shows the layout of a component that provides and requires interfaces. The component itself is a classifier box with an optional component symbol in the upper right corner. Provided interfaces are drawn as closed circles, required interfaces are open semi-circles. If a component uses another component, it connects to it by an assembly connector that joins these two symbols.

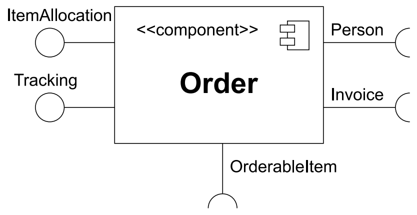


Fig. 2. Interfaces in UML (from [2])

Composition

Components can consist of other components. In contrast to the "black-box view", where the internals of a composite component are hidden, UML also defines an internal ("white-box") view of a component. In Fig. 3, an example of a component that contains other components can be seen.

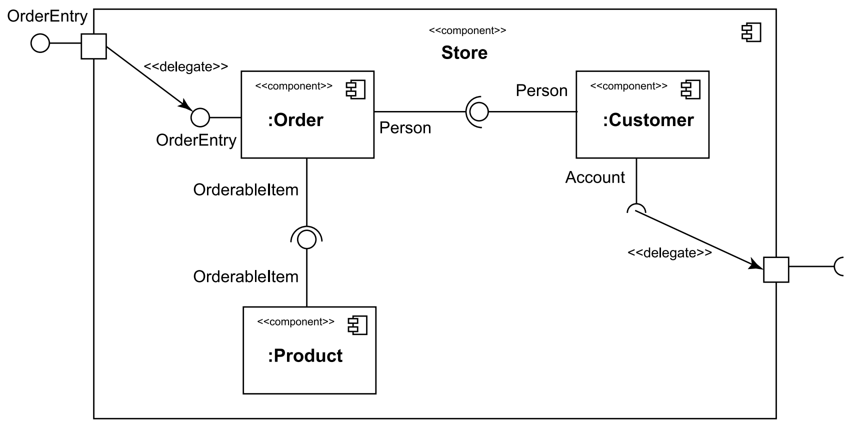


Fig. 3. Internal structure of a component (from [2])

3 Motivation

3.1 Quality of Service

The term "Quality of Service" is well-known from the world of computer networks and the internet as a means of prioritization in packet-oriented communication. Similarly, in the context of software engineering, QoS stands for the measuring of non-functional properties of software, like reliability, usage of resources and performance. These properties can be measured for systems that are already running. For software developers, however, it is desirable to be able to judge the behaviour of systems at early stages of development, so that design decisions can be based on these jugdements.

Component-based software design facilitates the judging of QoS properties of a complex system, since it can be broken down into the behaviour of the single components of which the system consists. Based on QoS assertions of the single components, the behaviour of a system can be calculated in advance if the components' behaviour is known and well-defined.

In order to consider QoS attributes at an early stage in component-based development, the component model must offer an infrastructure that supports evaluation on quality of service based on abstract resource models. The Palladio Component Model contains several concepts that allow statements on QoS. These concepts will be presented in the following chapters.

3.2 Performance Prediction

The performance of a system can be measured in terms of response time, throughput and resource utilisation, for example. The methods for predicting these values at an early stage of development are not numerous at the current state of software engineering technology, and thus they are not commonly used. Apart from the benefits mentioned in the above section which CBSE has for predicting and judging performance, the split of user roles however complicates the process compared to object-oriented programming, where development and implementation are performed by the same user. Furthermore, the black boxing mentioned in sect. 2.2 makes predictions depending on the control flow inside the component impossible.

Further aspects that are important for performance prediction are the resource environments of a system and input parameters. The Palladio Component Model provides concepts that take these aspects into account; they are described in the following chapters.

4 Concepts used in the Palladio Component Model

4.1 User Roles

In component-based software engineering (CBSE), the process of developing is divided into several roles. Usually, one distinguishes between the role of the *com-*

ponent developer, who is responsible for the development of the single components, and the *software architect*, who builds applications from these components.

In order to make statements about the Quality of Service of a component-based architecture at an early stage, further roles have to be introduced. The Palladio Component Model defines three additional roles: the *deployer*, the *QoS analyst* and the *domain expert*. These roles belong to the process model introduced by Koziolek and Happe in [3]. The definition of these roles is presented in the following sections.

For each role name, there is a number of synonymous names that can be used instead of the ones chosen for the PCM. A listing of synonyms can be found in [4, sect. 2.1.6.]

Component Developer

The *component developer* is one of the two classical roles in component- based software development. Component developers specify and implement components so that they can be deployed by independent third-parties. The complete specification of components includes an abstract and parametric description of the component and its behaviour, regarding functional as well as non-functional properties. The first ones define how the components' provided services call the required services, while the latter include QoS statements and performance attributes based on resource demands.

The specifications are usually produced by the component developers themselves.

Software Architect

The *software architect* (frequently called *system architect*) is the other role that is also present in other common CBSE models. Software architects are the driving force in a CBSE process. They design the architecture of a software project and are responsible for the delegation of work to the other user roles. The information provided by the other roles are integrated by them, and they also have to estimate whether there are missing values.

Software architects decompose applications into components, which can then be selected from an existing repository. If a proper component cannot be found, an abstract specification of the desired component is provided by the software architect. For the selected components, the software architect specifies the component connections to build the systems assembly. If a QoS analyst is consulted, non-functional properties of the system also have to be described.

A system with a definition of its boundaries can be compared to a composite component (see sect. 2.2). However, the purpose of composite components is to be used inside other composite components, whereas systems only interact with other systems. The structure of a system can be seen in Fig. 4

The software architect is also responsible of putting the components into a system assembly context. Contexts are mandatory for components, see sect. 5.3.

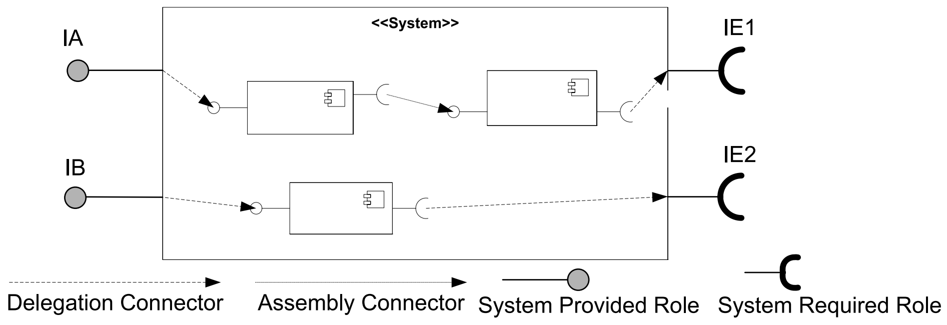


Fig. 4. A system and its assembly

After the systems assembly has been completed, it is handed over to the system deployer.

Deployer

The main task of the *deployer* role is to specify a resource environment in which components can be deployed. The specification of the environment ranges from information about the hard- and software that is used to the allocation of components to the available resources. Finally, the concrete setup of servers, clients, middleware and the network technology used is also defined.

The deployer adapts the platform-independet resource demand specifications of the component developer to the actual system configurations. Since the predictability of QoS properties also depends from deployment options, the parameters of deployment are specified in advance. The specification of an execution environment together with the allocation of components allows predictions on the QoS of the final system. The deployer can try different deployment scenarios to find out the setting that suites the non-functional requirements of the system.

Domain Expert

The role of a *domain expert* is common in the development of larger software systems. As business domain experts, they have special knowledge of the business domain in which the software is used. Their main field of activity is the requirements analysis. They also contribute to feasibility studies and the specification of functionality and business logic of the system. Domain experts can evaluate the users' work habits and know user behaviour. Thus, they can make estimations on user arrival rates, populations and think times. Their estimations are founded on experience with similar systems and market analysis.

Domain experts assist software architects in early stages of QoS analysis by specifing user behaviour, so that the software architect can use these data as a base for parameter inputs, which effect the QoS of the system.

QoS Analyst

The *QoS analyst* role also has an assisting character. QoS analysts help software architects interpret the results of the QoS analysis. In doing so, they retrieve QoS information from the requirements and, where missing, supplements data, e.g. resource demands. Information from other roles is integrated by the QoS analysts, and missing values are estimated. Furthermore, they use QoS analysis tools and pre-interpret the results of these tools.

The necessity of having this role is under discussion, since on the one hand, many of its tasks can be carried out by the software architect. On the other hand, supplementing incomplete data requires special skills that a software architect might not have. It can be argued that the software architect should not be bothered with the details of the QoS domain and remain in a superior position.

Since QoS analysis is often done by specialists nowadays, and the tools are not advanced enough to have automatic analysis, the role of a QoS analyst is present in the Palladio Component Model, although the goal of QoS predictions is the abolishment of this role. In the future, QoS analysis and predictions should be an integral part of software design and thus be carried out by the software architect.

4.2 Service Effect Specification

The concept of the *Service Effect Specification (SEFF)* has been introduced by Reussner in [5] as a description of how provided services of a component call the required services. The usage of SEFFs preserves the principle of black-boxing, since the inner structure and algorithms of the components are not visible, but gives the component developer the possibility to specify QoS attributes for components depending on context information. These context dependencies can include input parameters, resource usage and the usage of required services.

The description of SEFFs can have several forms. In simple cases, it can be just a list of required services which are called by the provided service. For more sophisticated analysis, a finite state machine (FSM) or a Petri net can be used to model the SEFF.

For the Palladio Component Model, the resource demanding Service Effect Specification (RDSEFF) is used to describe the behaviour of components. Every basic component in the PCM can contain an arbitrary number of SEFFs for every provided service (see Fig. 5), but only a maximum of one SEFF per type (e.g. FSM, Petri nets). RDSEFFs will be described in the following paragraphs.

Currently, SEFFs have to be specified by *component developers* manually, but with future tool support, semi-automatic generation is aimed at.

Behaviour

In Fig. 5, a `ResourceDemandingSEFF` is displayed inheriting from `ServiceEffectSpecification` and `ResourceDemandingBehaviour`. The latter is used to

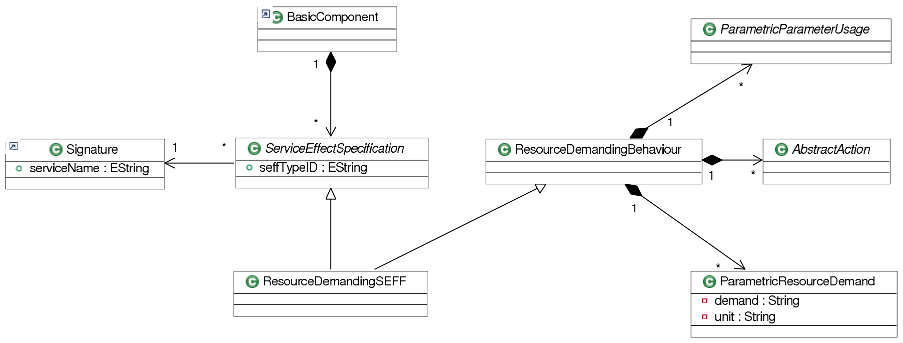


Fig. 5. RDSEFF behaviour (from [6])

model performance and reliability properties of a component service. It references to `AbstractActions` (see next paragraph) which model calls to required services, resource usage and control flow.

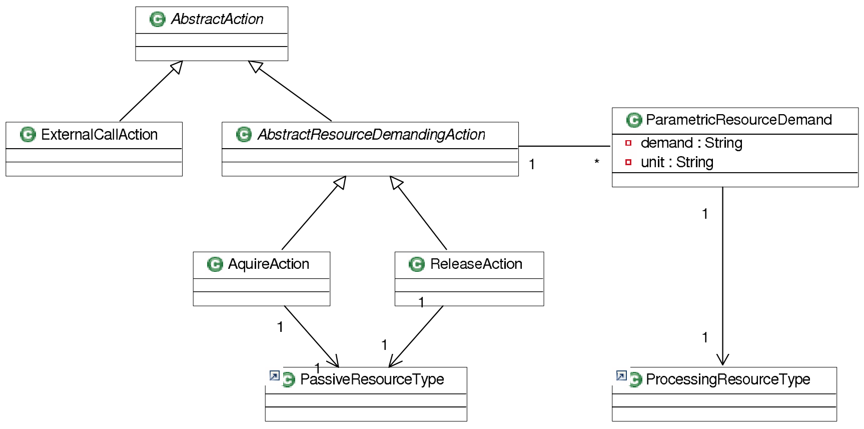


Fig. 6. Abstract actions (from [6])

Abstract Actions

As Fig. 6 shows, an `AbstractAction` has the subtypes `AbstractResourceDemandingAction` and `ExternalCallAction`. The first can place loads on the resources used by the component. On the level of abstraction where component-based modeling takes place, the concrete resource instances are not known and thus, abstract resource types are used. Upon deployment, they have to be instantiated with the actual resources of the system.

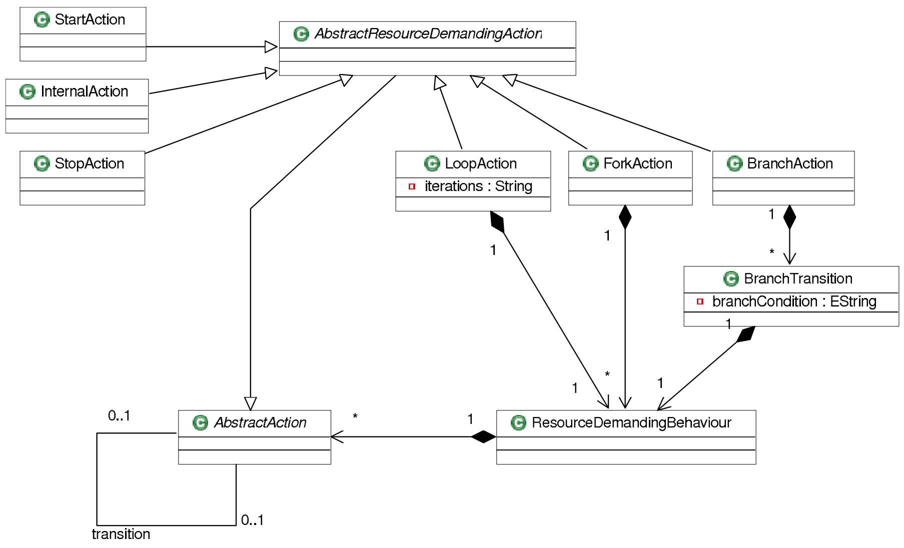


Fig. 7. Actions (from [6])

Actions

A `ResourceDemandingBehaviour` references several specialisations of `AbstractResourceDemandingAction` and can be seen in Fig. 7. An `InternalAction` is used to encapsulate internal operations of a single model element, assuring that the black-box principle mentioned above is kept. Further actions include `StartAction`s and `StopAction`s, which together with `LoopAction`s and `ForkAction`s model the control flow of actions.

4.3 Parametric Dependencies

The behaviour of a software component is heavily influenced by the nature of its input parameters. They can have effects on the performance and resource demands.

In the Palladio Component Model, several principles are included that reflect the dependency of input parameters and component behaviour. SEFFs are linked to usage profiles in an abstract way, using random variables to model the distribution of input variables. This is done by the component developer; later on, domain experts specify usage models that reflect realistic scenarios. Thus, QoS predictions can be made based on the the dependencies between input paramters and resource demands.

The decisions based on the nature of parameters include the selection of proper required services, the parameters passed to these services and resource demands. With the selection of services, branch conditions inside the SEFFs model a control flow that calls the proper services. Also, the number of loop iterations can depend on an input parameter, resulting in the number of calls to an external service or the parameters passed to them. Finally, resource demands are calculated in a parameter-dependant way, and thus, execution times can be determined and shortened.

Resource demands are expressed in the PCM using random variables. For random variables, probability distributions are defined, which in the case of the PCM are propablity mass functions (PMF), since they use discrete variables.

A detailed description of parametric dependencies can be found in [6, chapter 4], and in [3].

5 Structure of the Palladio Component Model

5.1 Components

Components are the core elements of a component model. A component combines functional and non-functional aspects. The definition of provided and required roles corresponds to that of provided and required interfaces in the UML standard (see Sect. 2.2).

In the Palladio Component Model, components are organised in a three-level hierarchy that can bee seen in Fig. 8. The difference between the metamodel side and the model side is essential. While in the vertical direction, the associations describe inheritance on the meta-model side, they stand for a conformity relation on the model side. The corresponding types on the same horizontal level indicate that the right hand side model types are instances of the meta-model types.

Inheritance is not allowed on the model side. The PCM does not support inheritance for components; instead, the *conforms* relation indicates sub- and supertype relations on the model side. On the metamodel side however, the *conforms* relation is only used as a helper construct, since component types do not reference upper layers. Conformity is expressed by OCL constraints instead.

The *Provided Component Type* is situated on the top layer of the metamodel as well as the model. For a provided type, only the definition of its provided roles is mandatory. The required roles may be defined, but can also be omitted.

For the *Complete Component Type*, the definition of both provided and required roles is mandatory.

The *Implementation Component Type* can exist in two forms: the *basic component*, where no internal connection structures can be specified, and the *composite component*, for which all internal components have to be defined.

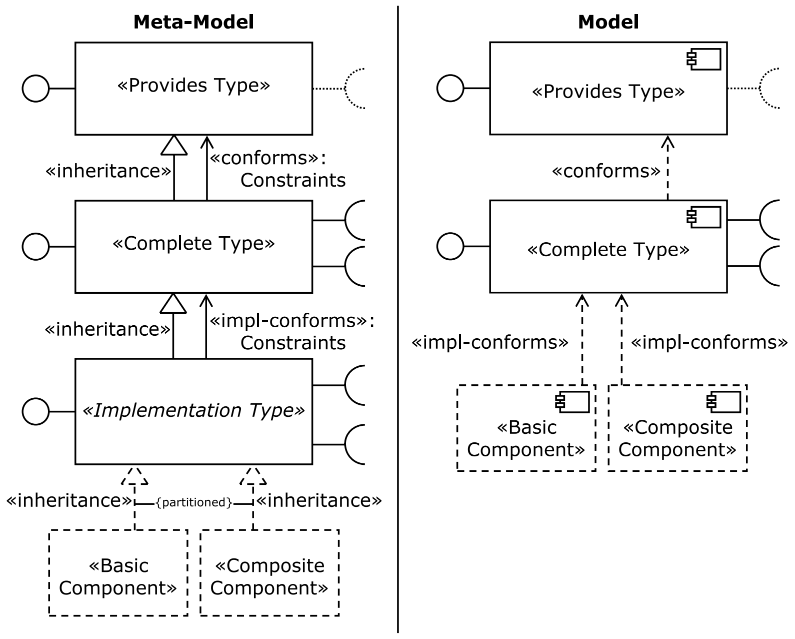


Fig. 8. Component type layers (from [4])

5.2 Interfaces

Interfaces serve multiple purposes in the PCM. They can be used to build a type system using inheritance, which, however, must be acyclic. Multiple inheritance is allowed. Another purpose of interfaces is the specification of allowed communication between components. Therefore, they serve as contracts, defining provided and required *roles* of a component. (Not to be confused with user roles).

The external view of components uses interfaces as the only means of reference to them; components can reference basic types and interfaces, but not other components themselves.

Finally, part of the inner state of a component is described by its interfaces, although interfaces themselves are stateless. The complex state of components is described in the following paragraph about protocols.

Signatures

Signatures define the services that an interface provides. For each service, an interface must define a unique signature which consists of an identifier, the ordered set of parameters, a return type and a collection of exceptions that may be thrown. In these terms, interface signatures are comparable to method signatures in object-oriented programming languages.

An interface can have multiple signatures (at a minimum of one signature) that need not necessarily reference different services. In contrast to this, different interfaces can offer services with identical signatures, but different semantics.

Protocols

A protocol is a set of calling sequences that defines a correct order of those calls. Protocols are not mandatory for interfaces. If a protocol is not specified for an interface, the trivial protocol seen in Fig. 9 is assumed. Protocols can be modeled using several concepts, which are not limited by the PCM specification. The common modeling techniques include simple lists, finite state machines and Petri nets.

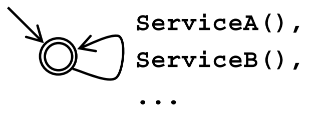


Fig. 9. The trivial protocol for interfaces

The PCM currently does not support protocols spanning multiple interfaces. The set of individual protocol states for each interface results into the *complex states* of the component.

A defined behaviour of components can only be expected if the interface protocols are obeyed [7, sect. 2.8.2].

5.3 Context

Since component developers cannot make assumptions on the environment in which their components are going to be used, it is necessary to define parametric context dependencies in order to be able to make QoS predictions and assertions. The PCM stresses three types of context factors, which will be described in the following.

System Assembly Context

The System Assembly Context defines the "horizontal" composition of a system, meaning the ways components connect to other components via their provided and required interfaces. The view on the single components is with respect to the black-box principle, so that only external properties of the single components are considered.

Hierarchy Context

In contrast to the System Assembly Context, the Hierarchy Context defines the "vertical" composition of a component-based system, regarding the inner structure of composite components. The knowledge of hierarchy context does not violate the principles of encapsulation, as the inner structure of composite components is still not visible to the (external) user. However, if a component is used inside another composite component, the information on this compositional aspect belongs to the hierarchy context.

Allocation Context

The Allocation Context deals with the allocation of components to hardware and software resources. The principle of resources is described in detail in Sect. 5.5. A resource environment to which components are allocated consists of resource containers holding processing resources and passive resources. These containers are connected via linking resources. A component in an assembly context can have multiple allocation contexts that place the component on different resource containers [4, sect. 3.3.5].

Usage Context

As part of the process model introduced in [8], a usage model is defined for the PCM, which models user behaviour, and thus, typical input data. These circumstances define the Usage Context. Within the Usage Contexts, parametric dependencies (see sect. 4.3) can be used to model the properties of typical input data.

5.4 Connectors

Connectors define the binding between two roles. They also indicate the direction of the control flow, which is delegated from the required roles to the provided roles at the other end of the connector.

There are two types of connectors: *delegation connectors* and *assembly connectors*. Both can be seen in Fig. 10, which will serve as an example for the explanation of both connector types. Connectors are visualized as arrows with dashed lines.

Delegation Connectors

Delegation connectors connect components at different hierarchy levels. In our example (Fig. 10), the external roles of the composite component A are connected to the corresponding roles of inner components. Roles of inner components can only be connected with outer roles if they are defined for the containing component. Delegation connectors are used for the vertical composition in the hierarchy context (see Sect. 5.3).

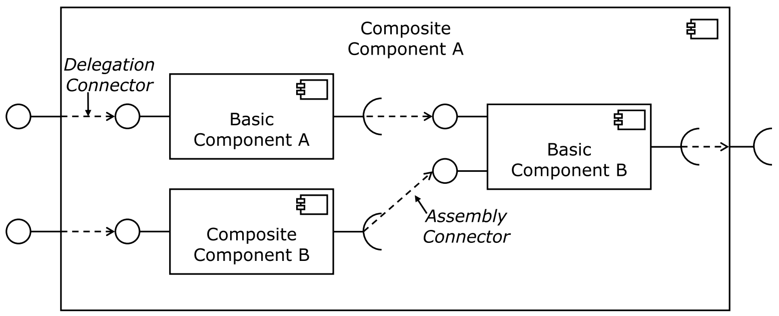


Fig. 10. A composite component showing different connector types (from [4])

Assembly Connectors

Assembly Connectors connect required roles in an assembly context with provided roles in another assembly context. Service calls of the component at the side of the connection where the required role is attached are delegated to the component at the other end of the connection, where the provided role is attached. Assembly connectors are used in the System Assembly Context (see Sect. 5.3) for horizontal compositions.

5.5 Resources

Resources belong to the field of deployment, and thus they are within the responsibility of the deployer role. A *resource environment* in the PCM is defined as a set of *resource containers* and the connections between. A resource container represents a physical entity, meaning a hardware system. The resources inside the container are categorised by a *resource type* system, which also corresponds to the physical occurences of hardware. System deployers and component developers develop a set of common resource types which is then stored in a *resource repository*.

The resource types are divided into two types: *active* or *processing resources* and *passive resources*. For connections between resource containers, there is a special type called *communication links*.

Passive Resources

Passive resources are owned by another resource, usually a process or a thread. Before they can be accessed, passive resources have to be acquired, and since resources can be limited, this means that the demanding process or thread may have to wait until he can take posession of the resource.

The process of acquiring and releasing passive resources has to be described in a SEFF of the resource itself.

Active/Processing Resources

Active resources can exist for themselves. They can actively execute a task, so they are also called processing resources. Their performance can be measured with certain metrics, depending on the type of the resource

A classical example for active resources are processors and disks.

6 Evaluation/Classification

In this chapter, the Palladio Component Model is evaluated in terms of usability, software support, and general aptitude for component-based software engineering. Based on the current status of development, a classification of PCM is attempted with respect to other existing component-based modeling techniques.

6.1 Developer Tool Support

For the Eclipse IDE[1], a set of plugins for the PCM exists. They are still under development. Based on the Eclipse Modeling Framework (EMF), a graphical editor can be used to create component repositories and manipulate models that instantiate the PCM metamodel. Functionality is limited at the current status of development, for example, not all diagram types are available.

6.2 Documentation

The documentation of the Palladio Component Model is also still in the process of being completed. The description of core concepts is almost complete, but technical references are missing. The official documentation's structure is very much oriented on the user roles and arranges even longer passages, such as the description of SEFFs, into subchapters of the user role descriptions.

For the metamodel, a generated UML documentation is also available.

6.3 Popularity

Since it is still in the stage of development, the PCM is only used in the scientific environment, especially at the institutions where it has been developed. It is used for research; actual utilisation for a software project in and industrial context is not known to the author.

6.4 Domain Specificness

The PCM is characterized by its focus on QoS principles, which makes it exceptional from other component models. The integration of QoS and performance predictions, however, does not narrow the field in which it can be used. It is a generic CBSE model that can be used to model any kind of system. It is not limited to object-oriented software development.

[1] http://www.eclipse.org

6.5 Usability

Since the notation is kept close to the UML 2.0 standard, software developers familiar with component-based software engineering can acquire the concepts used in the PCM quite easily. However, there are a lot of individual techniques that have been combined to form the PCM, so the universality of it requires studying not only technical concepts, but also organizational parts such as the workflow model or the user roles.

6.6 Abstraction Level

As a result of its domain-independent design, the level of abstraction at which the PCM operates is high. However, the development process specifies the concretisation of an abstract model in the phase of deployment, where parametric values are instantiated depending on the environment that the system is in. In this way, the PCM describes a process for different levels of abstraction, while the emphasis is on the early, more abstract stages of development.

7 Common Example: WebAudioStore

The example WebAudioStore has been used in [8] to describe the performance of a system in dependency of input parameters. Fig. 11 shows the simplified architecture for the example.

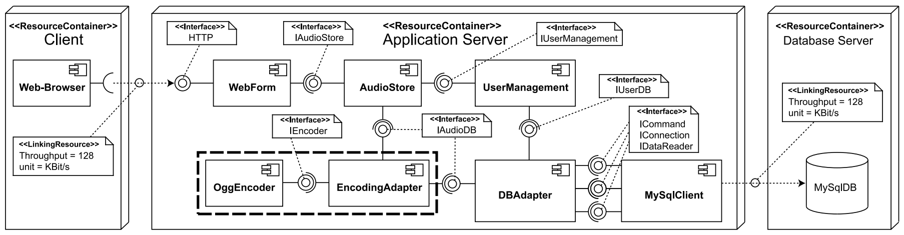


Fig. 11. The architecture of the WebAudioStore (from [8])

7.1 Structure

The WebAudioStore is an internet based service that allows users to upload and download files via a web interface. As it can be seen in Fig. 11, the store consists of three resource environments: client, application server and database server. The application server is accessed via its provided HTTP interface, and accesses a database server via SQL. The components `EncodingAdapter` and `OggEncoder` are displayed inside a dashed box because a design alternative is presented for them in [8, Sect. 5.2.].

7.2 Use Case

For the different use cases that occur in the WebAudioStore scenario, Service Effect Specifications are used to model the behaviour of components. As an example, we will describe the use case scenario “upload files”. Fig. 12 shows a FSM used as a SEFF for the procedure of uploading files: Users access a `WebForm` component to upload the files, implicitly passing parameters such as file number and size. This is reflected in the annotations for the `LoopAction`, where a collection of files is passed as an input parameter, determining the number of iterations for the loop. Inside the loop, another external action is called, and again, the annotations show the parameters that are passed during that call, including the file sizes.

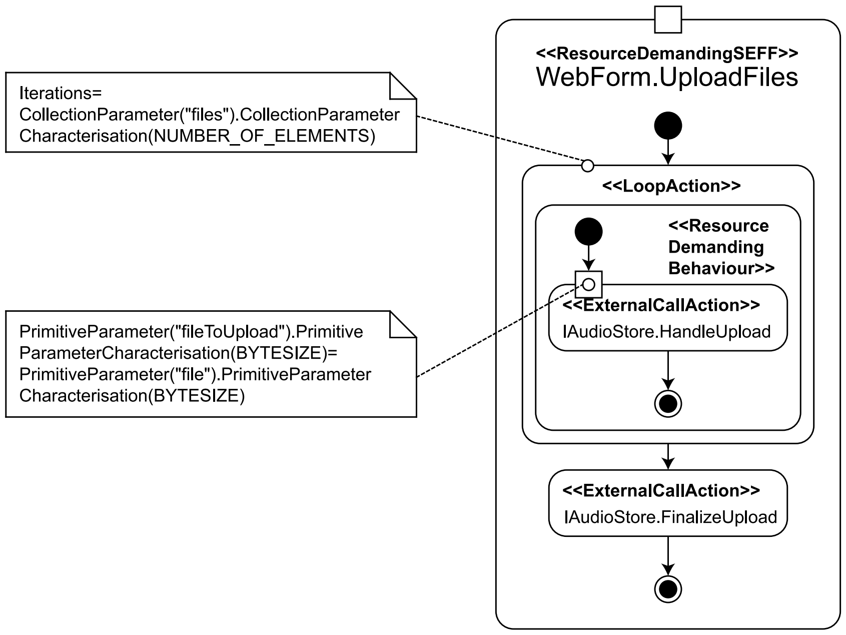

Iterations=
CollectionParameter("files").CollectionParameter
Characterisation(NUMBER_OF_ELEMENTS)
PrimitiveParameter("fileToUpload").Primitive
ParameterCharacterisation(BYTESIZE)=
PrimitiveParameter("file").PrimitiveParameter
Characterisation(BYTESIZE)
<<ResourceDemandingSEFF>>
WebForm.UploadFiles
<<LoopAction>>
<<Resource
Demanding
Behaviour>>
<<ExternalCallAction>>
IAudioStore.HandleUpload
<<ExternalCallAction>>
IAudioStore.FinalizeUpload

Fig. 12. The WebAudioStore use case “upload files” (from [8])

7.3 Performance Improvement

The design alternative seen in the dashed box of Fig. 11is introduced to improve the data transfer times by using a codec with higher efficiency. The components `AudioStore` and `DBAdapter` could also connect directly via the `IAudioDB` interface. Instead, the `EncodingAdapter` component is inserted, which allows smaller file sizes, but also needs more computational time. Performance prediction is used to estimate a speedup.

In [6], the design alternative is modeled using parametric dependencies. A usage profile for the use case “upload files” is created, specifiyng a probabilistic mass function (PMF) for the parameters “file size” and “number of files”. These data are collected by the Domain Expert, who has knowledge of typical user behaviour. Based on these parameters, a propability distribution is computed for the number of CPU operations. In combination with the actual hardware configuration from the allocation context, execution times are gained. These times are used to calculate response times in the scenarios with and without the additional components. The SEFFs of the two additional components are used to gain information on the internal control flow.

7.4 Development Process

In a distributed development process, the single components together with their SEFFs are developed by the component developer. The software architect then composes the whole system and, aided by a domain expert, instantiates the variables from a usage scenario, in this case, average file sizes and numbers. A QoS analyst may assist the software architect in interpreting the results of an analysis with the instantiated variables. The deployer then decides on the systems used for deployment, based on the QoS analysis.

8 Conclusion

Regarding its stage of development, the Palladio Component Model offers a large variety of concepts for different fields of component-based software engineering. Although some areas are not yet fully covered by the existing specification, the principles of introducing Quality of Service into component-based software engineering are fundamentally described. The Palladio Component Model combines several principles like Service Effect Specification and parametric dependencies on the technical side, while also delivering process models for a QoS based approach to workflow management. The extension of the two standard user roles to the new concept of five user roles eases modularization of a component based development process and also covers processes outside the core software development.

With the integration into the Eclipse IDE using EMF, and the orientation on the UML standard, the Palladio Component Model complies to widely-used industry standards and is thus not restricted to the research context from which it emerged.

Glossary

component-based software engineering (CBSE) Software engineering based on the composition of systems from components.

finite state machine (FSM) A finite collection of states, transitions and actions which describe a structured behaviour.

Meta Object Facility (MOF) Formal language and framework for specifying metamodels. OMG standard. It is used to define how meta data is organized.

Palladio Component Model (PCM) A component model developped at the Chair for Software Design and Quality at the University of Karlsruhe, Germany.

Quality of Service (QoS) Requirements on the quality of the behaviour of objects, e.g. in a software system.

resource demanding Service Effect Specification (RDSEFF) A special type of Service Effect Specification designed for performance and reliability predictions.

Service Effect Specification (SEFF) A description of how a provided service of a component calls its required services. An abstraction of the control flow through the component.

Unified Modeling Language (UML) A widely used graphical language for object modeling and specification. OMG standard.

Bibliography

1. McIlroy, D.: Mass produced software components. In Naur, P., Randell, B., eds.: Software Engineering, Report on a conference sponsored by the NATO science committee in Garmisch, Germany, Scientific Affairs Division, NATO (October 1968) 138–155
2. OMG: Unified Modeling Language: Superstructure (August 2005)
3. Koziolek, H., Happe, J.: A quality of service driven development process model for component-based software systems. In: Lecture Notes in Computer Science. Volume 4063., Springer-Verlag (2006) 335–343
4. Reussner, R., Becker, S., Happe, J., Koziolek, H., Krogmann, K., Kuperberg, M.: The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe, Germany (November 2006)
5. Reussner, R.H.: Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. PhD thesis, Universität Karlsruhe (TH) (2001)
6. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP '07: Proceedings of the 6th international workshop on Software and performance, New York, NY, USA, ACM Press (2007) 54–65
7. Krogmann, K.: Entwicklung und Transformation eines EMF-Modells des Palladio Komponenten-Meta-Modells. Master's thesis, Carl-von-Ossietzky-Universität Oldenburg (2006)

8. Koziolek, H., Happe, J., Becker, S.: Parameter Dependent Performance Specifications of Software Components. In: Proceedings of the Second International Conference on Quality of Software Architectures (QoSA2006). (2006)

The SOFA Component Model

Igor Goussev

Faculty of Informatics, University of Karlsruhe,
Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract. Building software applications from components has become a powerful and widespread tendency in the modern software engineering. Nowadays, many software applications are composed from reusable components. This paper introduces SOFA - an innovative platform for building large software systems. On the base of the SOFA component model, some features of component-based programming are presented, such as hierarchical top-down design, dynamic reconfiguration, versioning, etc. It is also demonstrated how the concepts of SOFA can be used for modelling arbitrary systems.

1 Introduction

For a successful development of component-based applications, it is necessary to have the possibility to describe the functionality of components and thus to distinguish among different components that provide the same set of interfaces and finally to choose the most suitable implementation. Such a description should satisfy some requirements. First of all, it should be quite abstract in order to provide freedom for further implementation. On the other hand, it should be detailed enough to allow application designers to get essential information about the usage of particular components. And finally, in order to allow the automatic checking of correctness of component use and binding, such a description should be very precise [9].

1.1 Component-based Development

Why should component based development be used for building large software systems? In order to answer this question, it is sufficient to give the following two arguments [2]:

1. There exist a lot of software modules which provide services, therefore reusing them is desirable in order to facilitate the software development process;
2. Component-based technology is more effective because it eliminates debugging of the reused parts.

There are many different views on what a component is and what its features are. But as a common conclusion of different approaches, a component can be defined as a black-box entity with well defined interfaces and behavior. One of the key features of a component is its reusability in different environments without knowing or modifying components internals [6]. From a design view, a component can be viewed as gray-box entity with the internal structure visible as a set of interconnecting subcomponents.

A component model is the set of rules that define creation, composition, life-cycle, and other features of a component. A particular implementation of a component model is called component system/platform [7].

The concepts of component-based development has been already realized in many systems - both industrial and academic. The main purpose of industrial systems is to provide a stable runtime. In many cases, it is done by neglecting such features, such as hierarchical architecture, multiple communication styles, behavior description, etc. The advantage of the academic component models is their rich set of features, especially allowing for powerful design with the help of hierarchical architectures, behavior specification, different communication styles, etc. The weak point of the academic component models lies in the insufficient runtime support. Many of the academic systems are only design-oriented, they have no runtime environment. A big gap between the industrial models and academic component models is a consequence of the fact, that it is very difficult to properly balance the semantics of advanced features, so that they can be well realized at design time, and correctly employed at runtime [6].

1.2 Project SOFA

The *SOFA (SOFtware Appliances)* is an academic project managed by the Distributed Systems Research Group at the Charles University in Prague [7]. The goal of this project is to develop a component system with a number of sophisticated features, such as the hierarchical composition, component trading and licensing, on demand distribution across the global network, run-time updating, and both design time and run-time protocol checking.

In SOFA, an application is composed of a set of dynamically downloadable and updatable components. Key issues addressed by SOFA include:

- dynamic component downloading,
- dynamic component updating (DCUP),
- hierarchical top-down design supported by reasoning on behavior compliance,
- distributed deployment,
- versioning.

The goal of this paper is to introduce the key features of the SOFA component model and show how the main concepts of SOFA could be used for modelling an arbitrary system.

The paper is structured as follows. The basic concepts of the SOFA architecture are described in Section 2. In Section 3, the key aspects as of behavior protocols are introduced. In Section 4, an overview of the DCUP architecture is provided. Section 5 presents component change and versioning, which is one of the key features of SOFA. An advanced component system SOFA 2.0 is introduced in Section 6. The evaluation of SOFA features is given in Section 7. Section 8 presents the on-line store for music files *WebAudioStore* which is a common expample for modelling. This section shows the use of SOFA for modelling an arbitrary system. Finally, the main issues are summarized in the concluding Section 9.

2 SOFA Architecture Overview

The concept of a *component* is one of central concepts in the component-based programming. Therefore, this section starts with the definition of a component in the SOFA architecture. Then, it will be shown how components are described by the *component definition language (CDL)*. Further, the concept of a *connector*, which is an architectural element for supporting interactions among components in a system, will be presented. At the end of this section, the SOFAnode and the SOFAnet are briefly introduced.

2.1 Components

In the SOFA component model, an application is viewed as a hierarchy of nested software components [2, 3]. In analogy with the classical concept of an object as an instance of a class, a software component is an instance of a component template.

In general, a *template* T is a pair $<F,A>$, where F is a *template frame* and A is a *template archtecture*[2]. The frame F defines the set of interfaces that any component which is an instance of T will possess. The frame F of a template T reflects a black-box view of the component. Interfaces are defined in the SOFA component definition language (CDL). An interface type can be declared as a *provided* interface or a *required* interface. A provided interface of a component specifies a service any component will offer to its clients. A required interface specifies a service any component will need from an external component.

Template architecture describes the structure of a concrete version of the corresponding template frame implementation by instantiating direct subcomponents and by specifying the necessary component interconnections via interface ties.

The SOFA architecture provides four types of interface ties [3]:

1. *binding* of a requires-interface to a provides-interface between two subcomponents,
2. *delegating* from a provides-interface of component to a subcomponent's provides-interface,
3. *subsuming* from a subcomponent's requires-interface to a requires-interface of component,
4. *exempting* an interface of a subcomponent from any ties.

The architecture of T reflects a grey-box view on the component. The architecture can be specified as primitive, which means that there are no subcomponents and the template frame implementation will be given in an underlying implementation language out of the scope of the architecture specification. When an architecture is not primitive, it is necessary to recursively specify ties of the nested components. The nested components are again instances of templates, each of them containing its architecture. Consequently the SOFA component is recursively defined up to the finest level of granularity.

2.2 CDL Specification Language

The specification of SOFA components is done by the component definition language (CDL). The complete syntax of CDL is provided in [7]. In this paper, the use of CDL is demonstrated on the simple example of a data base server, designed as a DB component, which is an instance of Database template (Figure 1) [3, 9].

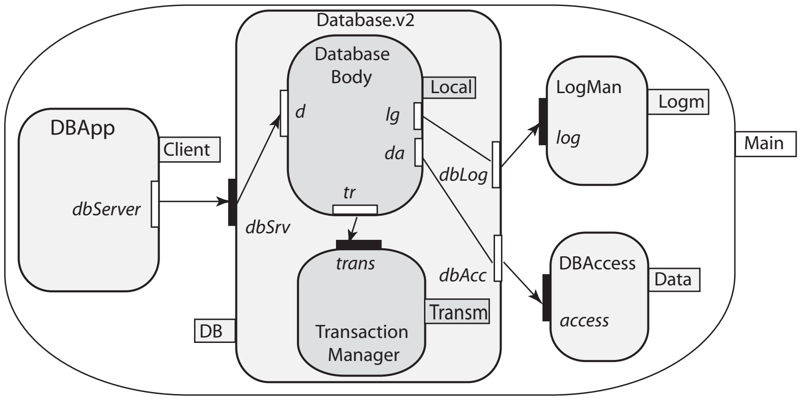


Fig. 1. Database architecture [9]

The component *DB* provides the *Insert*, *Delete*, and *Query* operations for inserting, removing and querying records of the database. To describe all these operations, CDL has the interface construct, which specifies the interface type as a set of method names. For this example, the interface can be defined as follows [9]:

```
interface IDBServer {
  void Insert(in string key, in string data);
  void Delete(in string key);
  void Query(in string query, out string data);
};
```

The database server is connected to the underlying database via the *IDatabaseAccess* interface type and to the logging component via the *ILogging* interface type [9]:

```
  interface ILogging {
    void LogEvent(in string event);
    void ClearLog();
  };

  interface IDatabaseAccess {
    void Open();
    void Insert(in string key, in string data);
    void Delete(in string key);
    void Query(in string query, out string data);
    void Close();
  };
```

The frame specification includes declarations of provides- and requires-interfaces. The frame construct for the database server can be specified as follows [9]:

```
 frame Database {
   provides:
     IDBServer dbSrv;
   requires:
     IDatabaseAccess dbAcc;
     ILogging dbLog;
 };
```

There are many possibilities to specify the template architecture of the *Database*. The first one is to declare it as primitive (it means, *DB* has no subcomponents) [9]:

```
  architecture Database version v1 {
    primitive;
  };
```

The other possibility is to to declare the *Database* as a structured component. For the example in Figure 1, the declaration could be done as follows [9]:

```
 architecture Database version v2 {
   inst TransactionManager Transm;
   inst DatabaseBody Local;

   bind Local:tr to Transm:trans;
   subsume Local:lg to dbLog;
   subsume Local:da to dbAcc;
   delegate dbSrv to Local:d;
};
```

This architecture specification shows the way how the subcomponents are instantiated and how their ties are specified (distinguishing bind, subsume and delegate cases).

2.3 Connectors

Connectors are used for the component interconnection [1]. The basic tasks of a connector are the following ones [10]:

- control and data transfer,
- interface adaptation and data conversion,
- access coordination and synchronization,
- communication intercepting,
- dynamic component linking.

Each of these tasks covers a different aspect of the component interactions.

In SOFA, a connector is specified in a similar way as a component - as a pair <*connector frame, connector architecture*>. The connector frame is specified by a set of role instances. A role is a generic interface of the connector. Its goal is to be tied to a component interface. The connector architecture describes the inner structure of a connector. It can be simple or compound. A simple connector contains only primitive elements, directly implemented by the underlying environment). A compound connector includes only instances of other connectors or components [1].

SOFA provides three types of predefined connectors *(CSProcCall, EventDelivery, DataStream)* and user-defined connectors.

2.4 SOFAnet and SOFAnode

SOFAnode is single environment for developing, distributing and running SOFA applications [1]. SOFAnode consists of five logical parts. The central part of a SOFAnode is a *Template Repository (TR).* TR includes an implementation of components and their CDL description. The other parts a SOFAnode provide the following services:

- the environment for developing of components (the MADE part),
- automatic distribution of components among SOFAnodes (the IN and OUT parts),
- the environment for launching and running of component applications (the RUN part).

Not all SOFAnode's have to include all these parts.

A homogeneous network of SOFAnodes is called SOFAnet [2] and could be viewed as a directed graph of SOFAnodes. Figure 2 presents a sample scheme of several SOFAnodes connected to form a SOFAnet.

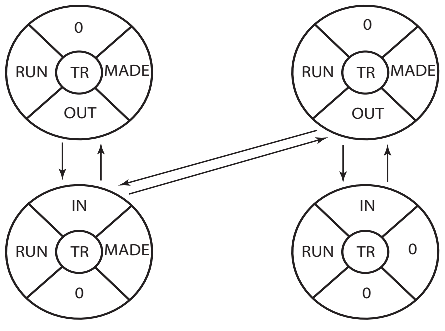


Fig. 2. SOFAnet example [1]

3 Behavior protocols

3.1 Model of communication

A communication among software components could be viewed as sequences of communication events [3, 9]. An event can be a *request* or a *response*. In order to model the computational activity of a component, a concept of an *agent* is introduced. As it was defined in [3, 9]:

"*An agent is a computational entity handling sequences of events*".

An agent can emit and absorb an event, or it can perform an internal event. At a time, not more than one event can be handled by an agent. Communication between agents is realised via peer-to-peer external connections. Let A and B be agents. Then an external connection C between A and B can be defined as the pair of one-way channels denoted $C_{A->B}$resp. $C_{B->A}$.

The events emitted by A to B are sequentially transmitted by the channel $C_{A->B}$. In B, these events are absorbed in the order they were delivered. In the similar way, via C the events emitted by B are transmitted from B to A via $C_{B->A}$. An agent can communicate with a finite number of agents and two agents can be connected by a finite number of connections.

There are *primitive* or *composed* agents. A composed agent consists of two or more smaller agents. Let P be a composed agent and consist of two agents A and B. The connections of P are the union of the connections of A and B. In this case, internal connections of P are the external connections of A and B via which A and B communicate between each other.

A hierarchy of agents is called a *system* [9]. Every agent is a part of a system. An agent with no external connections is called the *root* of a system.

It is assumed in the communication model, that events are atomic. It means they cannot be devided into sub-events. Events can be also denoted by event tokents. For example, for a method name a, event tokens *!a↑, ?a↑, !a↓* and *?a↓* denote emitting a method call, accepting a method call, emitting a return and accepting a return, respectively.

A sequence of event tokens denoting events occuring on a agent is a *trace*. For example, the trace $<!a\uparrow;\ ?a\downarrow>$ describes activity of a caller (emitting a method call followed by accepting the return), while the trace $<?a\uparrow;\ !a\downarrow>$ shows what the caller does (accepting a call and emitting the return).

3.2 Syntax and semantics of behavior protocols

The set of all traces, which can be produced by an agent, is denoted by the *behavior* of an agent. Behavior described as a set of traces in not very human-readable. Moreover, the set of all traces, which can be produced by an agent, can be infinite. Therefore, the notation of behavior protocols is used [1, 9].

A *behavior protocol* is a regular-like expression on the set of all event tokens, which goal is to generate a set of traces [1, 9].

Some of operators used in behavior protocols are as follows [9]:

$a*$	repetition; it is equivalent to a; a ; ... ; a.
$a\|b$	and-parallel; the traces of a and b are generated in parallel and the resulting trace is an arbitrary interleaving of them.
$a\|\|b$	or-parallel; it stands for $a + b + (a\|b)$
a ; b	sequencing; a trace generated by a is followed by a trace generated by b.
$a + b$	alternative; the resulting trace is generated alternatively by a or by b.

For example, the behavior protocol *a; (b + (b; b))* means that the event a occurs at first, and then the event b occurs once or twice.

3.3 Behaviour protocols in SOFA CDL

Components can be associated with agents. Thus, the concepts of behavior protocols can be applied also for components.

SOFA contains three types of protocols: interface, frame and architecture protocols[1, 3, 9].

An *interface protocol* is a part of an interface type definition and located after the methods definitions, such as in [9]:

```
interface ITransaction {
  void Begin();
  void Commit();
  void Abort();
 protocol:
  ( Begin ; ( Commit + Abort ) )*
 };
```

This protocol shows the use of an interface for simple transactions. The first method to be called is *Begin*. It starts a transaction process. As the next step, either *Commit* or *Abort* are performed to finish the transaction. The sign *** shows that this sequence can be repeated any number of times.

A protocol generating behavior of a frame is called *frame protocol*. The following example demonstrates the use of a frame protocol on the *Database* frame (see Figure 1) [9]:

```
frame Database {
  provides:
    IDBServer dbSrv;
  requires:
    IDatabaseAccess dbAcc;
    ILogging dbLog;
  protocol:
    !dbAcc.Open ;
    ( ?dbSrv.Insert { ( !dbAcc.Insert ; !dbLog.LogEvent )* } +
    ?dbSrvDelete { ( !dbAcc.Delete ; !dbLog.LogEvent )* } +
    ?dbSrvQuery { !dbAcc.Query* }
     )* ;
    !dbAcc.Close
 };
```

In this example, the nested calls reflect the need for logging every modification of the database. For example, when the *Insert*-operation is performed, any number of *dbAcc.Insert* calls can be executed. After each of these calls is finished, the modification is logged by performing *dbLog.LogEvent*. In the similar manner, the *Delete*-operation is performed. For *dbAcc.Query*, it is simply forwarded to the underlying database without any logging.

There exists third type of protocols: *architecture protocol*. Architecture protocols are not written by a programmer, they are automatically generated by CDL compiler from frame protocols of an architecture's subcomponents [1].

4 Dynamic Component Update (DCUP)

The DCUP architecture is a specific architecture of SOFA components which enables their safe updating at runtime [2]. One of the key features of DCUP is doing updates with no human intervention at end-user side. The DCUP extends the SOFA component model with additional implementation objects and by a special technique for updating a component at runtime.

DCUP consists of two parts - *permanent* and *replaceable*. On the other hand, the component is divided into a *functional* part and a *control* part (see Figure 3). The corresponding interfaces are called *control interfaces* and *functional interfaces*. The control interface is used only for managing tasks. The functional interface corresponds to the component interface described by the SOFA component model, presented in Section 2.2.

The DCUP architecture several contains specific control objects: *Component Manager (CM), Component Builder (CB), Updater, ClassLoader*, and *Wrapper*. Every DCUP component contains one instance of CM and one instance of CB. The CM can be interpreted like the heart of the permanent part, because it is responsible for controlling the runtime lifecycle of the component [1, 2]. The main task of the CM is to coordinate updates. The CB is a key object of the replaceable part of a DCUP component. It is associated with a particular component version. Therefore, it is replaced together with component versions each time when a new updating occurs. The main task of a CB is to build component internals (in the case of composed component - subcomponents, in the case of primitive component - implementation objects).

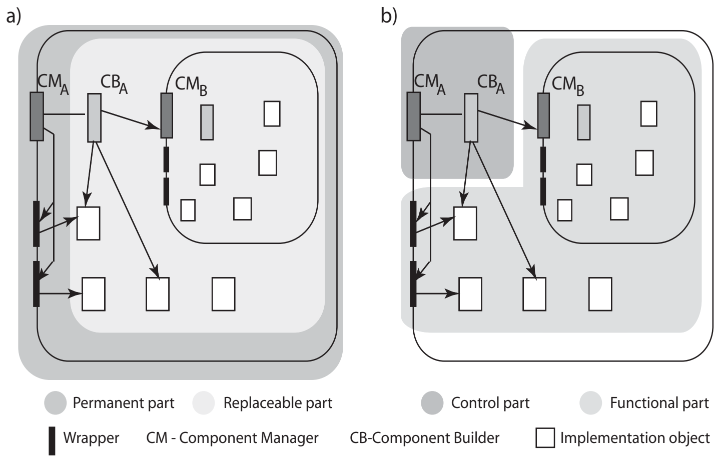


Fig. 3. Structure of a DCUP component [2]

5 Component Change and Versioning in SOFA

5.1 Determining Component Compatibility

The component compatibility can be defined as follows [5]. Let *V1* and *V2* be two revisions of one component and *P1* and *P2* denote the set of provided interfaces of the revisions *V1* and *V2*. In the similar manner, *R1* and *R2* denote the set of required interfaces and *B1* and *B2* the behaviors of two revisions.

To compare *P2* with *P1*, *R2* with *R1* and *B2* with *B1*, the general set operators like $\subseteq$, $\supseteq$ are used. The set operator Δ denotes mutation or incompatibility (for example, $P2 \Delta P1$ is equal to $P2 \not\supset P1$ AND $P2 \not\subseteq P1$). The usage of set operators for comparing the interfaces of two revisions is straightforward. To compare the behaviors of two revisions, their behavior protocols have to be considered. As mentioned in Section 3, a behavior protocol describes a set of traces, which can be produced by an component. Let *pr1* denote the protocol of revision *V1*, *pr2* denote the protocol of revision *V2*. Then, the set of all traces generated by *pr1* can be compared with the set of all traces generated by *pr2*. In other words, the behaviors of two revisions of a component can be compared. For example, B2⊆B1 means, that the behavior of the revision *V2* is smaller or equal than the behavior of the revision *V1*, or in other words, the set of all traces generated by the protocol of the revision *V2* is a subset of all traces generated by the protocol of the revision *V1*.

Then compatibility of *V2* is with respect to *V1* could be defined as follows [5]:

- *equal*, if P2 = P1 AND R2 = R1 AND B2 = B1 ;
- *strongly compatible*, if P2⊇P1 AND B2⊆B1 AND R2⊆R1 ;
- *conditionally compatible*, if
  P2⊇P1 AND ((R2⊇R1 OR R2*Δ*R1) OR (B2⊇B1 OR B2*Δ*B1)) ;
- *incompatible*, if P2⊂P1 OR P2*Δ*P1.

This definition shows that the version *V2* strongly compatible with respect to *V1*, if it provides more services, has less behavior, and requires less from the other components. This case is also denoted a *'contravariant'* change. The version *V2* is incompatible if the provided interface was reduced or mutated. The conditionally compatible version has specialised or mutated required interfaces and/or behaviors. It is also denoted a *'contravariant'* change [5].

5.2 Revision Numbers and Change Indications

The interface and frame versioning in SOFA uses the analysis of changes done in a new component version. Its results are placed into two data structures: the *hierarchical revision number* and an *indication of the types of changes* [5]. The hierarchical revision number is a part of the CDL. Thus, both the developers and the tools can see what changes have been done. The indication of the types of changes at each level is a more specific information for the tools.

SOFA revision number is specifed for interfaces as an ordered pair *(P,B)*, for component frames as an ordered triple *(P,B,R)* of natural numbers, where *P,R* and *B* denote the the revision numbers of provided interfaces, required interfaces and behavior protocols, respectively. Any change from the previous revision in the provided interfaces, required interfaces or behavior protocols is mapped into an increase of the corresponding part of the revision number. Thus, the revision number could be directly derived from the types of last changes. This contrasts with the usual practice of arbitrarily choosing a *Major.minor* scheme [5].

The following example shows this principle. The both parts *P* and *B* of revision number for the `IDBServer` interface were incremented after a new method had been added and some changes in behavior protocol have been made [5]:

```
interface IDBServer rev 1.1 {
  void insert(in int key, in tdata data);
  void delete(in int key);
  void query(in tdata mask, out tdata data);
  protocol: ( insert + delete + query )*
};

interface IDBServer rev 2.2 {
  void set(in dbProperties prop);
  void insert(in int key, in tdata data);
  void delete(in int key);
  void query(in tdata mask, out tdata data);
  protocol: set ; ( insert + delete + query + set )*
};
```

The change type indication is defined for interfaces as an ordered pair *(p,b)*, for component frames as an ordered triple *(p,b,r)*, where the value of each item *p,b*, or *r* is equal to *math, gen, spec* or *mutation.* In the example, presented above, the interface change type indication for revision 2.2 will be *(spec, mutation)* because the set of interface methods is extended and the new protocol is incomparable to the previous one.

Revision and change identification are involved in several operations in the SOFA framework. The exact meaning of revision numbers and change types allows to determine, before the new version replaces the current one, if the new configuration can be consistent. For example, an application contains revision 2.0.0 of the *Database* component. A new revision 3.1.0 is available as an update, and is the corresponding change type indication is *(spec, spec, match).* The difference between the revision numbers shows that the provides interfaces and behavior protocols have been changed. The value of the change type indication indicates that it was a 'covariant' change.

6 SOFA 2.0 - an advanced component system

6.1 Limitations and obstacles of SOFA

SOFA has become a very innovative platform for building large software systems. On the other hand, its usage revealed several limitations and obstacles. This is not only specific for SOFA, many other component systems suffer of similar problems, namely [6]:

– limited support for dynamic reconfigurations,
– no structure of the control part of a component,
– unbalanced support for multiple communication styles.

Dynamic reconfiguration. This is a modification of the application architecture during the runtime [6]. It means adding and removing components at runtime, passing references to components, etc. Dynamic reconfiguration is a very complex feature of component based development. Its main problem lies in the fact that it is very difficult to specify the dynamic behavior of of an application at design time. There are at least two naive solutions of this problem. The first one is to forbid dynamic reconfiguration. But this solution is not appropriate, because dynamic changes of an architecture are inherent part of many applications. The second naive solution is to neglect dynamic reconfigurations at design time and allow for arbitrary ones at runtime. This is not acceptable either, because it leads to an uncontrolled modification of the architecture. Therefore, dynanic reconfigurations should be taken into consideration at design time.

Control part of components. In addition to provided and required interfaces, there is also a third type of interfaces - control interfaces or controllers [6]. Controllers provide access to non-functional aspects of components, such as life-cycle management, reconfiguration, etc. An explicit modeling of the control part of components can improve the functionality of the runtime environment.

Multiple communication styles. There are four types of communication styles in SOFA - method invocation, message passing, streaming, and distributed shared memory [8]. Communication style influences the way components can be bound together. For example, it is even possible to connect a required interface to another required interface [6]. Therefore, it is very important to provide an explicit support for communication styles. Without such a support, it is very difficult to realize different architectural styles. Although SOFA supports multiple communication styles, this support is unbalanced in SOFA and not well integrated with other SOFA concepts.

6.2 SOFA 2.0

The component system SOFA 2.0 is based on the experience with SOFA. In SOFA 2.0, the main concepts and general design of the SOFA system remained the same. Components and connectors are first-class entities. Hierarchical components are also employed in SOFA 2.0. On the other hand, SOFA 2.0 introduces new concepts which have not existed in SOFA (e.g., the dynamic reconfiguration, explicit controller) and also improves several concepts already existing in the original SOFA (e.g., multiple communication styles, deployment) [6].

7 Evaluation

The evaluation of the some component systems including the original version of SOFA was done in [12]. SOFA has been classified as one of the most advanced component systems(together with Koala and KobrA). Among the abvantages of SOFA, was mentioned its ability to: compose components at design time, store composed components in a repository, reuse the stored components (including composite ones) in further compositions [6, 12].

The original version of SOFA includes limitations, which were improved in SOFA 2.0.

7.1 Tool Support

There are two implementations of SOFA. They were developed by the Distributed Systems Research Group at the Charles University in Prague.

The first one is the SOFA implementation in Java. It provides an environment for developing and running SOFA/DCUP components [1]. It is distributed under the GNU Lesser General Public License [14]. The second implementation is the experimental implementation in C++, developed by Tomas Kalibera and described in his master thesis [13].

7.2 Documentation

The SOFA implementation in Java is well documented. There are a lot of manuals describing various aspects of SOFA, such as user's manuals, reference manuals, demos, etc. All documentations of SOFA can be found and downloaded from the home page of SOFA project [7].

7.3 Distribution

SOFA is a typical academic component-based system. Howerer, it is used in some industrial and business projects (e.g., France Telecom, INRI). The project SOFA is a member of ObjectWeb open source consortium.

7.4 Domains specialization

SOFA is used for building component-based applications for embedded systems and for business software (e.g., France Telecom.

7.5 Applicability

The development of SOFA component can be devided into several steps. First, the interfaces, component frame and architecture are described. This description is compiled by CDL compiler and stored in the *Type Information Repository (TIR)*. Then, the code generator generates Java types, *Component Builder (CB)*, and *Component Deployment Descriptor (CDD)*. When the Java types are generated, the implementation of the component have to be provided. Then, the complete component is put to *Template Repository (TR)*. Finally, the *Application Deployment Descriptor (ADD)* is made from CDDs of all componets used in the application, and the component is ready for instantiating [1].

The binary distribution of the SOFA implementation in Java includes all essential tools for developing a SOFA applications and can be downloaded from the home page of SOFA project [7].

7.6 Abstraction layer

In analogy with the classical concept of an object as an instance of a class, a SOFA component is an instance of a component template, which is a pair <*Frame,Architecture*>. Thus, components in SOFA are defined on the class abstraction layer.

7.7 Type level

SOFA specifies the architectural infrastucture of a system and describes components on the type level. The implementation of components have to be made in separate Java classes.

8 Common example for modelling

The goal of this section is to demonstrate how SOFA CDL could be used for modelling an arbitrary system. As a common example for the modeling, it was suggested to take a component-based on-line store for music files *WebAudioStore*, described in [11].

This on-line store works in the following way. The store provides two functions: buying and selling music files. To sell music files, users of *WebAudioStore* select one or several files from their hard drives and click the upload button, which initiates the upload process. Similarly, for buying music files, the files to be downloaded from *WebAudioStore* have to be selected at first. Then, a click on the download button starts the download process.

The modelling of the *WebAudioStore* with the SOFA describes the stucture of components in terms CDL (interface, frame and architecture) and their interactions as behavior protocols. The example in this section shows the case for uploading multiple files.

The first layer of the *WebAudioStore* architecture consists of three containers: *Client*, *Application Server*, and *Database Server*. In terms of SOFA, a container can be represented as a component. Thus, in the case of the *WebAudioStore*, the above-mentioned containers can be denoted by the components `Client`, `ApplicationServer`, and `DatabaseServer`.

As shown in [11], the `ApplicationServer` is the only composed component. It consists of seven smaller components interconnected to each other. The other two components, `Client` and `DatabaseServer`, are represented as primitive components, whose frame implementations must be given in an underlying implementation language out of the scope of the architecture specification.

The architecture of the *WebAudioStore* has an design alternative, when the uploaded audio files have to be compressed before they are transmitted to the data base server. This is achieved by putting two additional components (`EncodingAdapter` and `OggEncoder`) between the `DBAdapter` and the `AudioStore` in the basic architecture of the *WebAudioStore*.

In this section, the file compression performed by the `EncodingAdapter` and the `OggEncoder` is considered in detail. The interconnection between the

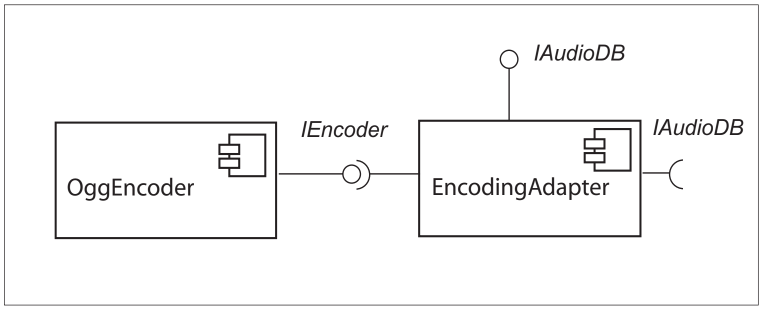


Fig. 4. Components of *WebAudioStore* components for file compression [11]

`EncodingAdapter` and the `OggEncoder` can be found in Figure 4. The both components are interconnected to each other via the interface `IEncoder`.

Figure 5 shows the scenario for file compression performed by these components. The `EncodingAdapter` calls the method *EncodeFile* of the component `OggEncoder`, which initiates a compession process in the `OggEncoder`. The compession process includes three steps, which are performed by the methods *WriteFileToDisk, ExecuteEncoder, ReadEncodedFileFromDisk*.

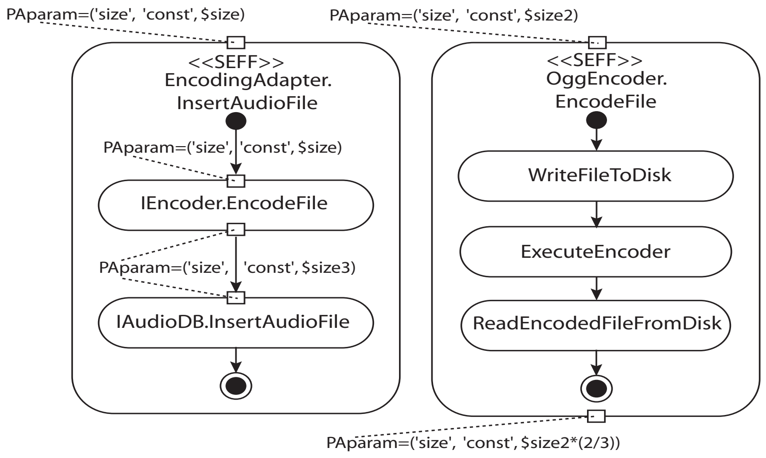

PAparam=('size', 'const', $size)
PAparam=('size', 'const', $size2)
<<SEFF>>
EncodingAdapter.
InsertAudioFile
<<SEFF>>
OggEncoder.
EncodeFile
PAparam=('size', 'const', $size)
IEncoder.EncodeFile
PAparam=('size', 'const', $size3)
IAudioDB.InsertAudioFile
WriteFileToDisk
ExecuteEncoder
ReadEncodedFileFromDisk
PAparam=('size', 'const', $size2*(2/3))

Fig. 5. SEFFs of the EncodingAdapter and OggEncoder [11]

There are at least two possibilities for modelling this situation. The first way is to represent the methods *WriteFileToDisk, ExecuteEncoder*, and *ReadEncodedFileFromDisk* as internal methods of the component `OggEncoder`. In this case, these methods do not appear in any interface declaration, and they are defined in an underlying implementation language.

The second possibility is to build a new component, which provides interface for performing the methods *WriteFileToDisk, ExecuteEncoder*, and *ReadEncodedFileFromDisk*. Let this new component be denoted by `EncoderInternal` and its provided interface by *IEncoderInternal*. The modified interconnection between the compression components can be found in Figure 6.

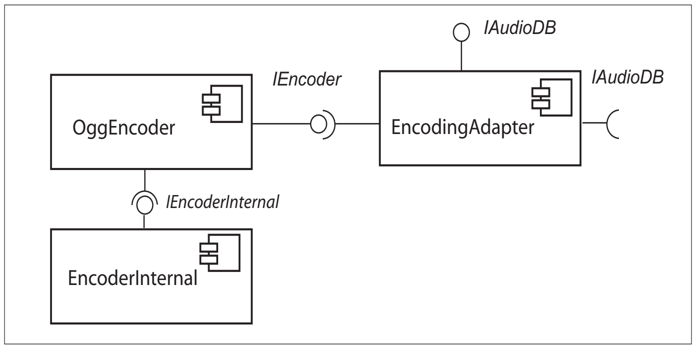


Fig. 6. Components of *WebAudioStore* components for file compression (modified)

To describe the file compression in terms of SOFA CDL, it is necessary give the declaration of interface types at first. An interface type includes a list of public methods and possily an interface protocol. The corresponding interface type for the interface *IEncoder* and *IEncoderInternal* could be defined as follows:

```
interface IEncoder_Type {
    void EncodeFile();
   protocol:
     EncodeFile*
 };

interface IEncoderInternal_Type {
    void WriteFileToDisk();
    void ExecuteEncoder();
    void ReadEncodedFileFromDisk();
  protocol:
    WriteFileToDisk; ExecuteEncoder; ReadEncodedFileFromDisk
 };
```

After the interface declarations have been done, the desciption of the component frames can be done. A component frame incudes lists of provided and required interfaces and a frame protocol. The corresponding frames for the `EncodingAdapter`, `EncoderInternal` and the `OggEncoder` could be descibed in the following way:

```
frame EncodingAdapter {
   provides:
     IAudioDB_Type IAudioDB_1;
   requires:
     IAudioDB_Type IAudioDB_2;
     IEncoder_Type IEncoder;
   protocol:
     ?IAudioDB_1.InsertAudioFile {!IEncoder.EncodeFile;
     !IAudioDB_2.InsertAudioFile}
};

frame EncoderInternal {
  provides:
    IEncoderInternal_Type IEncoderInternal;
  protocol:
    ?IEncoderInternal.WriteFileToDisk;
    ?IEncoderInternal.ExecuteEncoder;
    ?IEncoderInternal.ReadEncodedFileFromDisk
 };

frame OggEncoder {
   provides:
     IEncoder_Type IEncoder;
   requires:
     IEncoderInternal_Type IEncoderInternal;
   protocol:
     (?IEncoder.EncodeFile {!IEncoderInternal.WriteFileToDisk;
     !IEncoderInternal.ExecuteEncoder;
     !IEncoderInternal.ReadEncodedFileFromDisk})*
};
```

The frame description of the `EncodingAdapter` includes also the interfaces *IAudioDB1* and *IAudioDB2*, which enables the interconnection to the other two components: the `AudioStore` and the `DBAdapter`. The code (interface types and frames) for all components of of the *WebAudioStore* architecture is presented in the appendix. The descriptions of the interfaces and frames include also behavior protocols. In most cases, the protocol is a simple sequence of methods to be called. In some frames, the regular expression for protocol includes also an operator * to show a possible repetition of a method or a group of methods.

A version of the architecture of the composed component ApplicationServer could look like as follows:

```
architecture ApplicationServer version v1 {
   inst WebForm cWebForm;
   inst AudioStore cAudioStore;
```

```
    inst UserManagement cUserManagement;
    inst DBAdapter cDBAdapter;
    inst MySqlClient cMySqlClient;
    inst OggEncoder cOggEncoder;
    inst EncodingAdapter cEncodingAdapter;
    inst EncoderInternal cEncoderInternal;

    bind cWebForm:IAudioStore to cAudioStore:IAudioStore;
    bind cAudioStore:IUserManagement to
         cUserManagement:IUserManagement;
    bind cUserManagement:IUserDB to cDBAdapter:IUserDB;
    bind cAudioStore:IAudioDB to cEncodingAdapter:IAudioDB_1;
    bind cEncodingAdapter:IAudioDB_2 to cDBAdapter:IAudioDB;
    bind cEncodingAdapter:IEncoder to cOggEncoder:IEncoder;
    bind cDBAdapter:ICommand to cMySqlClient:ICommand;
    bind cDBAdapter:IConnection to cMySqlClient:IConnection;
    bind cDBAdapter:IDataReader to cMySqlClient:IDataReader;
    bind cOggEncoder:IEncoderInternal to cEncoderInternal:EncoderInternal;

    delegate input to cWebForm:input;
    subsume cMySqlClient:output to output;
}
```

This architecture version illustrates the way how the subcomponents are instantiated and how their ties are specified (distinguishing bind, subsume and delegate cases).

9 Conclusion

In this paper, the key ideas of the SOFA component model were presented. It was also shown how the main conceps of SOFA could be used for modeling arbitrary systems. The key features of the SOFA component model can be summarized as follows [2]:

- SOFA component model separates definitions of component interface from the description of component architectures, which allows multiple versions of architecture per an interface;
- It provides an update operation with clearly defined relationship to versioning;
- The mechanism of behavior protocols defininės the interaction between components in clear and precise way.

With respect to updaiting, DCUP components support safe component state transition during dynamic updates by executing transitions at well defined moments. There is also no need for any human intervention at the end-user side during dynamic updates.

All these features represent SOFA as an innovative component-based system. On the other hand, SOFA is not widespread compared to the other component-based systems (CORBA, EJB, COM). There are no books, which describe SOFA. The only sources of information about SOFA are the documentations from the home page of SOFA project and the conference papers, mostly written by the authors of SOFA. Moreover, there is a few information about concrete systems based on SOFA. But all these lacks are not drawbacks of SOFA as a component-based system. They only show how difficult to bring a new product to the software market nowadays.

References

1. SOFA Users manual, http://nenya.ms.mff.cuni.cz
2. Plasil,F., Balek,D., Janecek,R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating. Proceedings of ICCDS'98, Annapolis, IEEE CS, 1998
3. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
4. Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Proceedings of CBSE 2006, Vasteras near Stockholm, Sweden, Jun2006
5. Brada, P.: Component Change and Version Identification in SOFA, Proceedings of SOFSEM'99, Czech Republic, 1999
6. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, Aug2006
7. SOFA project, http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html, 2000.
8. Bure, T., Plil, F.: Communication Style Driven Connector Configurations, In Software Engineering Research and Applications, LNCS3026, 2004
9. Plasil, F., Visnovsky, S., Besta, M.: Bounding Component Behavior via Protocols, Proceedings of TOOLS USA '99, CS IEEE, TOOLS 30, pp. 387-398, Aug 1999
10. Bale, D., Plasil, F.: Software Connectors: A Hierarchical Model, Tech. Report No. 2000/2, Dep. of Sowfware Engineering, Charles University, Prague, Revised 11/25/2000, Nov 2000
11. Koziolek H., Happe J., Becker S.: Parameter Dependent Performance Specifications of Software Components, Proceedings of CBSE 2006, Västerås, Sweden, 2006
12. Lau, K.-K., Wang, Z.: A Taxonomy of Software Component Models, Proc. of EUROMICRO-SEAA05, Porto, Portugal, Sep 2005
13. Kalibera, T.: SOFA Support in C++ Environments, master thesis.
14. GNU Lesser General Public License, http://www.gnu.org/copyleft/lesser.html

Appendix: The SOFA CDL declarations for the WebAudioStore components

The interface declarations

```
interface IWebForm_Type {
  void UploadFiles();
};

interface IAudioStore_Type {
   void HandleUpload();
   void FinalizeUpload();
 };

interface IAudioDB_Type {
   void InsertAudioFile();
   void InsertAudioFileInfo();
   void FinalizeUpload();
 };

interface IEncoder_Type {
   void EncodeFile();
  protocol:
   EncodeFile*
};

interface IEncoderInternal_Type {
     void WriteFileToDisk();
     void ExecuteEncoder();
     void ReadEncodedFileFromDisk();
   protocol:
     WriteFileToDisk; ExecuteEncoder; ReadEncodedFileFromDisk
};
```

The declarations of the interface types

```
frame WebForm {
 provides:
   IWebForm_Type input;
 requires:
   IAudioStore_Type IAudioStore;
 protocol:
   ?input.UploadFiles {!IAudioStore.HandleUpload*;!IAudioStore.FinaliseUpload}
 };
```

```
frame AudioStore {
 provides:
   IAudioStore_Type IAudioStore;
 requires:
   IUsermanagement_Type IUsermanagement;
   IAudioDB_Type IAudioDB;
 protocol:
   ?IAudioStore.HandleUpload {!IAudioDB.InserAudioFile}
 };

frame DBAdapter {
 provides:
   IUserDB_Type IUserDB;
   IAudioDB_Type IAudioDB;
 requires:
   ICommand_Type ICommand;
   IConnection_Type IConnection;
   IDataReader_Type  IDataReader;
 protocol:
   ?IAudioDB.InserAudioFile;?IAudioDB.InserAudioFileInfo;?IAudioDB.FinalizeUpload
};

frame EncodingAdapter {
 provides:
   IAudioDB_Type IAudioDB_1;
 requires:
   IAudioDB_Type IAudioDB_2;
   IEncoder_Type IEncoder;
 protocol:
   ?IAudioDB_1.InsertAudioFile {!IEncoder.EncodeFile; !IAudioDB_2.InsertAudioFile}
};

 frame EncoderInternal {
   provides:
     IEncoderInternal_Type IEncoderInternal;
   protocol:
     ?IEncoderInternal.WriteFileToDisk;
     ?IEncoderInternal.ExecuteEncoder;
     ?IEncoderInternal.ReadEncodedFileFromDisk
  };

 frame OggEncoder {
    provides:
      IEncoder_Type IEncoder;
    requires:
      IEncoderInternal_Type IEncoderInternal;
    protocol:
      (?IEncoder.EncodeFile {!IEncoderInternal.WriteFileToDisk;
      !IEncoderInternal.ExecuteEncoder;
      !IEncoderInternal.ReadEncodedFileFromDisk})*
 };

frame UserManagement  {
 provides:
   IUsermanagement_Type IUsermanagement;
 requires:
   IUserDB_Type IUserDB;
};

frame MySqlClient {
 provides:
   ICommand_Type ICommand;
   IConnection_Type IConnection;
   IDataReader_Type  IDataReader;
 requires:
   IMySqlDB_Type output;
};

frame ApplicationServer {
 provides:
   IWebForm_Type input;
 requires:
   IMySqlDB_Type output;
};
```

Das Koala Komponentenmodell

Dimitar Hodzhev

Zusammenfassung In dem Bereich von Customer Electronics(CE)[1] muss immer kompliziertere und umfangreichere Software für kürzerere Zeit geschrieben werden. Die fördert die Benutzung von Software-Komponentenmodelle.
Das Thema dieses Artikels ist das Koala Komponentenmodell für Philips CE-Geräte. Das Koala Komponentenmodell wird betrachtet und eingeschätzt. Außerdem kriegt der Leser einen Überblick über den CE-Bereich und dessen Anforderungen. Es wird die Frage untersucht, warum man da ein Komponentenmodell benutzen will und was CBSE ist. Dieser Artikel antwortet auch die Frage,warum Koala entwickelt wurde.

Key words: Component-Based Software Engineering(CBSE)[2], Software-Komponentenmodell, Koala, CE, Philips

1 Einführung

1.1 Einführung in Customer Electronics (CE)

Customer Electronics (CE) ist ein Bereich in der Elektronik. Er beinhaltet Geräte wie Fernsehgeräte, Videorecorder, Radios, Walkie-Talkies, Hi-Fi Geräte, Haushaltunterhaltungsgeräte, PDAs, Spielkonsolen sowie Internet-Haushaltgeräte. Diese Geräte sind für Gelegenheitsbenutzung oder Unterhaltung und nicht für professionelle Zwecke gedacht. Trotzdem werden die Qualitäts- und Funktionalanforderungen immer anspruchsvoller.
Fast alle heutige CE Geräte bestehen aus Hardware und eingebetteter Software. Am Anfang war das Schreiben von CE-Software nicht so schwer, aber in den letzten ein Paar Jahre ist man auf die folgenden Grundprobleme gestoßen[10]:

– Die Größe und die Komplexität der CE-Software in den einzelnen Produkten steigt rasch. Dies kann grob durch das Moor'sche Gesetz beschrieben werden - Verdoppelung der Code-Größe alle zwei Jahre.

– Die Vielfalt der Produkten und deren Software wächst sehr rasch.

– Die Entwicklungszeit soll dagegen sehr verkürzt werden.

Philips hat schon seit 1978 in seinen Fernsehgeräte Software eingebettet[11]. Früher hat man bei den Fernsehgeräten und Videorecorder durch die Software

[1] deutsch: 'Kundenelektronik'
[2] deutsch: 'Komponentenbasierte Softwaretechnik'

nur grundlegende Funktionen der Hardware programmiert. Mit der Zeit wurden andere Aufgaben wie *Signalverarbeitung* oder *Datenverarbeitung* der Software übergeben. Die CE-Software hat sogar neue Funktionen wie etwa schöne graphische Bedieneroberflächen oder elektronische Benutzerhandbücher ermöglicht. Heutzutage werden die Computer- und CE-Bereiche zusammengelegt und so entstehen neue Services - z.B. WebTV usw.
Die CE Geräte sind nicht mehr einzelne Modelle, sondern werden in s.g. Produktenfamilien zusammengefasst. Die Geräte innerhalb einer Produktfamilie haben die gleiche bzw. sehr ähnliche Hardware, unterscheiden sich aber z.B. in dem Preis, den Funktionen, der Ausstattung, den uterstützten Standarten und anderen Kriterien. Das heißt die Software einzelnen Geräte ist sehr unterschiedlich. Dadurch wächst die Softwaregröße immer mehr. Abbildung 1 stellt dar, wie sich die Größe der in Hochleistungsfernsehgeräte eingebetteten Software im Zeitraum von 1978 bis 2004 verändert hat[11].

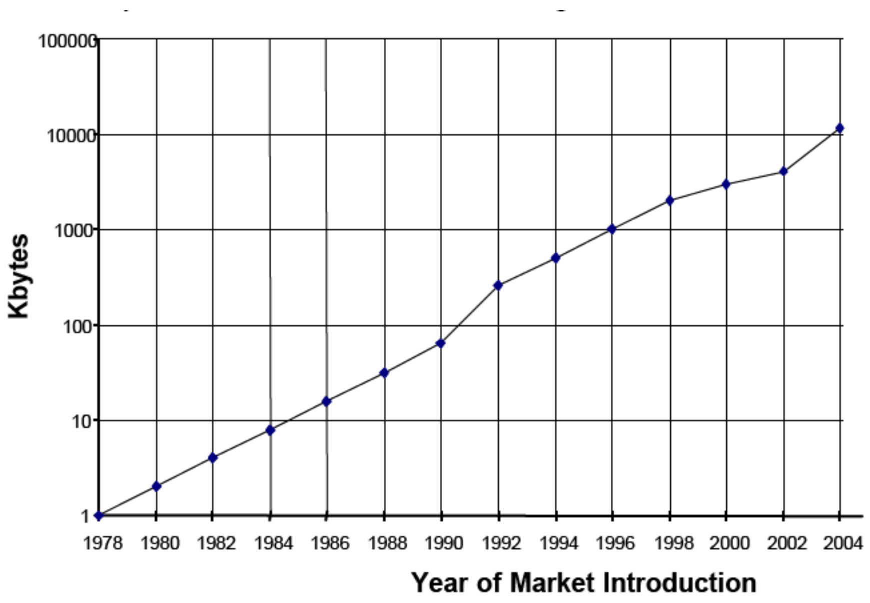


Abbildung 1. Größe der in Hochleistungsfernsehgeräten eingebetteten Software[11]

Auf der anderen Seite ist der Markt sehr dynamisch und stellt immer größere Anforderungen. Die Firmen sollen in der Lage sein neue Modelle eines Gerätes bzw. neue Produkte möglichst schnell zu produzieren. Typischerweise werden jeder 6 Monate neue Produkte auf den Markt gebracht[11].
Weitere Herausforderungen sind die Kostenminimierung des Herstellen und der

Entwurf von s.g. Combo-Geräte (z.B. Drucker-Scanner-Kopier oder TV-DVD).

1.2 Motivation

Wir haben schon gesehen, dass die Hersteller von CE-Geräte die Herausforderung haben, immer komplizierte und umfangreiche Software für kürzerer Zeit zu schreiben. Mit dem konvetionellen Programmieren ist diese Aufgabe nicht zu schaffen, sogar wenn mehrere Leute angestellt werden.
Die Wiederverwendung von Software-Komponenten ermöglicht das Benutzen von gleicher Software in verschiedenen Produkten. Das erspart viel Entwicklungszeit und Mühe. Der klassische Ansatz - definieren von Bibliotheken - kann nur in bestimmten Domäne wie wissenschaftliche und graphische Bibliotheken erfolgreich sein. Die Bibliotheken sind zwar wiederverwendbar, aber können die Ähnlichkeiten bzw. die Unterschiede im Anwendungsstruktur nicht behandeln.
Die Entwickler haben Objekt-orientierte Frameworks erfunden,um Anwendungen zu schreiben, die gemeinsame Daten und Strukturen teilen. Das Framework liefert ein Skelett, das der Programmi in verschiedene Art und Weise benutzen kann. Dieser Ansatz macht die Entwicklung von Software schneller, solange die Anwendungen ähnliche Struktur haben. Leider ist der Strukturwechsel schwer, weil die Struktur im Framework eingebettet ist. Also hat man Zusammenhang zwischen Framework und Komponenten[10, 3].

Die Komponentenbasierte Software-Entwicklung (engl. Component Based Software Engineering)(CBSE) bietet eine Löesung unserer Probleme. Wichtige Vorteile sind die folgende:

- CBSE stellt wiederverwendbare Komponente zur Verfügung.
- Komposition von verschiedener alten Komponenten, um eine neue Anwendung zu schreiben.
- bessere Zuverlässigkeit, da die einzelne Komponenten allein sorgfällig späzifiziert, entworfen und getestet sind.

Die Idee für Softwarekomponenten kommt aus der Hardware. Ein Stück Software (Softwarekomponente) ist z.B. einem integrierten Schaltkreis (IC) ähnlich - realisiert bestimmte Aufgabe(n), ist unabhängig und kann autonom produziert und verkauft werden[2].

Der komponentenbasierte Ansatz ermöglicht den Software-Entwickler mehrfache Konfigurationen durch Strukturänderungen und Inhaltsänderungen. Die Softwarekomponente ist gekapseltes Stück Software mit einer Schnittstelle zur äußeren Welt, die man in vieler verschiedenen Konfigurationen benutzen kann. Klassische Beispiele in Desktop-Anwendungssoftware sind der *button*, *tree view* usw. Bekannte Software-Komponentenmodelle sind COM/ActiveX, JavaBeans

und CORBA[10].
Das Ziel der Komponentenbasierten Softwaretechnik (CBSE) ist das Bauen von Software mit Hilfe von vordefinierten Software-Komponenten.
Eine Software-Komponente kann ein binärer Codeblock sein. Diese Komponente kann man direkt anwenden, aber das Innere der Komponente bleibt verborgen - s.g. black box.
Eine Software-Komponente kann auch ein Stück Quellcode sein. Diese Art von Komponenten muss man zuerst kompilieren und erst dann benutzen. "Das Innere“der Komponente kann geprüft und sogar geändert werden - s.g. *white box*, obwohl dies meisten nicht gestattet ist - Änderungen sollen nur von den Komponenten-Entwickler gemacht werden.
Eine Software-Komponente kann auch eine Sammlung von Modellen, die verschiedene Aspekte der Komponenten darstellen, sein. Z.B. das Dokumentationsmodell, Ausführungsmodell (ausführbare Form der Komponente) und Perfomanzmodell einer Komponente.
Um ein Software-System zu bauen muss man die einzelne Komponente zusammenstellen (engl. compose). Da es viele verschiedene Komponenten-Typen gibt, gibt es auch viele verschiedene Kombinationen für Zusammenstellen. Die Definition der Komponenten und die Regeln zum Zusammenstellen bestimmen das Komponentenmodell. Wenn man Software mit Hilfe einem bestimmten Komponentenmodell erzeugt, dann soll man die Regeln und die Methoden bzw. Verfahren (Techniken) des gewählten Komponentenmodell folgen. Z.B. in COM soll jede Komponente einen globalen eindeutigen Bezeichner haben; Koala benutzt das C Mechanismus um den Funktionsaufruf mit dem entsprechenden Fuktionsrumpf zu binden usw.
Also eine Software-Komponententechnik ist eine Sammlung von Techniken, die die Komposition von Komponenten in einem bestimmten Komponentenmodell ermöglicht. Es ist möglich, dass ein Komponentenmodell verschiedene Software-Komponententechniken haben kann. Alle Werkzeuge, Bibliotheken usw., die die Software-Komponententechnik eines Komponentenmodells implementieren, bilden die Komponentenarchitektur. Z.B. die Koala Komponentenarchitektur enthaltet den Koala Compiler, das Werkzeug zum Binden und das Visualisierungsprogramm.
Im Leben eines Software-Systems gibt es mehrere Phasen (Zeitpunkte), wo man Komponenten zusammenstellen soll: Entwicklungsphase, Einsatzphase usw. Für jede Phase kann ein anderes Komponentenmodell eingesetzt werden müssen. Z.B. bei der Entwicklungsphase - Komposition von Quelldateien (hier spielen die Quelldateien die Rolle der Komponenten)/Z.B. Koala Komponenten-Modell oder die Quellbaum (engl. *source-tree* Komponenten-Modelle)/, bei der Einsatzphase (*deployment-time*) - Komposition von den verschiedenen Teile einer Anwendung und dann Installieren auf die gezielte Maschine/Da ist RPM ein typisches Komponenten-Modell/, Zusammenstellen von Komponenten (binäre Form- z.B. '.dll'-s) beim laufenden System - da ist COM ein typisches Komponenten-Modell usw.
Deswegen bezieht sich CBSE auf verschiedene Komponentenmodelle. Eine große

Herausforderung von CBSE ist das Verbinden und das Nutzen aller im Leben eines Software-Systems gebrauchten Komponentenmodelle in einer einheitlichen Weise [4].

2 Das Koala Komponentenmodell

Viele Komponentenmodelle werden mit einer oder mehreren Programmiersprachen (z.B. Visual Basic, Java) benutzt um Konfigurationen von Komponenten zu konstruieren. Dieser Ansatz hat aber einen großen Nachteil: es ist schwer den Vorgang der Konfiguration zu visualisieren und deswegen ist auch schwer die Struktur der Software zu modellieren.
Um die Modellierung der Struktur zu erleichtern und dabei von den Konzepte der CBSE zu profitieren, kombiniert man ein Softwarekomponentenmodell und eine Architekturbeschreibungssprache (engl.*Architectural Description Language*)(ADL).
Mit Hilfe von ADL kann man die Struktur im Sinne von beinhaltenden Komponenten explizit beschreiben. Durch diese Beschreibung wird die Vielfalt und die Komplexität einzelnen Produkte sichtbar. Deswegen ist die ADL ein wertvolles Werkzeug für die Software-Architekten.
Kombination aus ADL und einem Komponentenmodell gibt es schon (z.B. COM). Im Gegensatz zu Desktop-Softwareanwendungen muss man aber bei CE-Software (und vor allem im Bereich der Entwicklung von Videorekorder und Fernsehgeräte) die folgende spezifische Anforderungen berücksichtigen:

– Viele Verbindungen zwischen den Komponenten sind konstant und sogar zur Konfigurationszeit bekannt. Um Laufzeit-Overhead zu vermeiden soll statisches Binden, wo möglich, benutzt werden.

– Hochleistungsgeräte sollen später durch dynamisches Binden von aktualisierten Komponenten (Software-Updates einer Komponente) verbessert werden.

– Man braucht explizite Notation für die Require-Schnittstellen.
  Abbildung 2 zeigt den Bedarf von Require-Schnittstellen.
  Sei A eine Software-Komponente und seien B1 und B2 zwei Versionen einer anderen Komponente. Wenn Komponente A Zugriff auf B1 braucht, dann standardweise A importiert B (s. Abb1/1a). Dies führt aber dazu, dass es spezifisches B1-'Wissen'(Information) in A gibt und das verhindert das weitere Kombinieren von A mit B2(s. Abb1/1b). Eine Lösung wäre, dass man eine abstrakte Komponente definiert, die A importiert und lässt das System sich für B1 oder B2 entscheiden. Mögliche Kombinationen sind A mit B1 oder A mit B2. Dann wäre aber nicht möglich Produkt 3 (s. Abb1/1c) zu konstruieren. In diesem Fall das Binden an die richtigen Komponente (B1 oder B2) wird erst beim Laufzeit festgestellt (gemäß bestimmter Bedingung, die man beim Laufzeit weisst).
  Die bessere Entscheidung ist diese Bindungsinformation von den wirklichen Komponenten auszulagern. Dann sagt man, dass Komponente A braucht

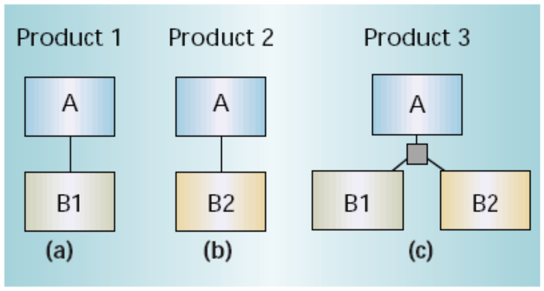


Abbildung 2. Der Bedarf von Require-Schnittstellen[10]

(engl. *require*) eine Schnittstelle von bestimmtem Typ und B1 und B2 bieten (engl. *provide*) eine solche Schnittstelle (allerdings unterscheidet sich die Implementation dieser Schnittstelle).

Darwin ist eine vor Koala existierende ADL, die als Basis für Koala ADL benutzt wurde.Obwohl, dass Darwin für Verteilte Systeme entworfen wurde, hatte er schon viele Eigenschaften, die man sich beim Entwurf von Koala sich wünschte:

– Explizite hierarchische Struktur.

– Komponente mit Require- und Provide-Schnittstellen.

Allerdings sollte man noch die folgende Aspekte beim Entwurf von Koala ändern bzw. berücksichtigen:

– Die leichte Ergänzung von Glue-Code(*Glue-Code: Zusätzlicher Code, der zwei Komponenten anpasst und die Verbindung zwischen denen ermöglicht. Eine Komponente hat die Require-Schnittstelle, die andere die Provide-Schnittstelle, aber die Schnittstellen uterscheiden sich leicht(z.B. die Provide-Schnittstelle hat eine Funktion mehr oder eine Funktion hat mehr Aufrufsparameter usw). Deswegen braucht man den Glue-Code*) zwischen den Komponenten (ohne weitere Hilfskomponenten zu erzeugen)

– Vielfältiges Parametermechanismus, das das Definieren von verschiedenen Parameter erlaubt und Code-Optimierung abhängig von der Parametereinstellungen gestattet[10].

2.1 Historischer Überblick

Die Koala ADL und das Koala Komponenten-Modell wurden von Philips Labs enwickelt. Ziel war die CBSE Uterschtützung beim Entwurf von CE-Geräte (vor allem Videorekorder, DVD-Player, Fernsehgeräte). Die Entwicklung von Koala hat Mitte 1996 begonnen.

Rob van Ommering, Frank van der Linden (Philips Research Laboratories), Jeff Kramer, Jef Maggee(Imperial Collage, London) sind die bedeutendsten Personen bei der Entwicklung von Koala.
Koala Komponentenmodell wurde in dem Zeitraum von September 1996 bis März 1997 definiert. Der erste Koala Compiler wurde von January 1997 bis Juni 1997 implementiert. Die Idee dabei war eine Koala unterstützte TV-Software-Entwicklung ab Sommer 1997. Dies hat man leider abgebrochen[3] und man sollte eine neue Software-Architektur zur TV-Software-Entwicklung bis Sommer 1998 erfinden. Diese Architektur (als MG-R[4] bekannt) war 2000 bei der Entwicklung von einem Fernsehmodell erfolgreich eingesetzt. Bis 2002 waren alle Fernsehmodelle durch MG-R entwickelt.
Das Komponentenmodell COM und die Darwin ADL spielen eine grosse Rolle bei der Koala-Entstehung.
Abbildung 3 zeigt die Namenherkunft von Koala.

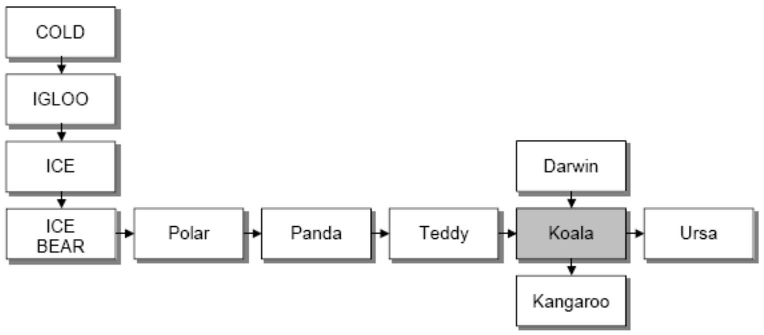


Abbildung 3. Ableitung und Herkunft von den Koala Namen[11]

COLD, IGLOO, ICE und ICE waren entsprechend eine Sprache, eine Bibliothek, eine Entwicklungsumgebung und GUI zum Schreiben von formalen Spezifikationen. Polar und Panda waren eine graphische Sprache und ein graphischer Editor für COLD. Teddy und Ursa sind Werkzeugen für Architekturvisualisierung und Architekturverifikation. Darwin ist eine ADL für Verteilte Systeme. Kangaroo ist eine ADL zur Beschreibung von benutzerdefinierte Schnittstellen.
Heute ist Koala das Standardwerkzeug in Philips beim Schreiben von CE-Software[11]. Die aktuelle Version von Koala ist v2 (August 2003), die v1 erweitert und ist mit ihr weitgehend kompatibel[1].

[3] „*...Due to other circumstances, this was cancelled, and research received the assignment of designing a new component-based software architecture for televisions in 1998...*“[11]

[4] Das Komponentenmodell bzw. das Projekt heißt doch Koala

2.2 Das Konzept von Koala

Wie schon erwähnt, die Struktur von Koala besteht aus einer ADL und einem Komponentenmodell. Koala beinhaltet die folgende drei Sprachen:

– Komponentendefinitionssprache (engl. *Component definition language*)(CDL) - dient zur Komponentenbeschreibung bezüglich der Komponentendefinition und der Schnittstellendefinition.
– Schnittstellendefinitionssprache (engl. *Interface definition language*) (IDL) - beschreibt die Fuktionsprototype und Konstanten bezüglich der Datentypendefinition.
– Datentypendefinitionssprache (engl. *Data type definition language*) (DDL) - beschreibt neue Datentypen bezüglich vordefinierter Typen.

In Koala gibt es die folgende grundlegende Konzepte:

– *Komponente*- eine Art Verkapselungseinheit.
– *Schnittstelle* - eine kleine zusammenhängige Sammlung von Funktionen.
– *Modul* - eine Code-Einheit.
– *Verbindung*(engl. *binding*) - eine Verbindung zwischen Schnittstellen.
– *Repository* - ist der Ort, wo alle Komponentendefinitionen, Schnittstellendefinitionen und Datentypdefinitionen aufbewahrt sind.

Die Koala Terminologie unterscheidet sich ein bisschen von dieser in CBSE. In Koala sind die Komponentendefinitionen (besser gesagt-die Komponententypdefinitionen), was eigentlich die Klassen in den Objekt-orientierten Sprachen (OOL) sind. Eine Komponentenintanziierung in Koala ist wie die Erzeugung eines Objektes in OOL. Die Rolle der Komponente (im Sinne von der CBSE-Konzept) spielt die s.g. Package. Ein Package ist Sammlung von Klassen(Komponenten) und Schnittstellen. Die Benutzer von Packages dürfen den Code innerhalb des Package nicht ändern, also nur kompilieren und ihn benutzen.
Die Koala Komponenten können nur durch vordefinierte Schnittstellen mit der Umgebung oder mit anderen Komponenten kommunizieren. Der Quellcode ist den anderen Entwicklern/Software-Architekten komplett sichtbar(es gibt keine binary- oder andere Zwischenformate, man kauft die Komponente als Quellcode).
Die Komponenten werden in Repositories mit der dazugehörigen Information für Versionsnummer und Kompabilitätsinformation gelagert und gespeichert. Außerdem können die Komponenten parametrisiert werden, um in verschiede Umgebungen passen zu können[7].
Die wichtigste Merkmale von Koala sind:

– Von Darwin abgeleitete ADL.
– Die Idee für Schnittstellen wurde von Microsoft COM inspiriert.
– Explizite *Require*- und *Provide*-Schnittstellen.
– *Module* - zum Einfügen von zusammensetzendem Glue-Code und das gas Ganze als Modul zu kapseln.

- *Diversity-Schnittstellen* und *Switches* - zur Behandlung der Produktvariation.
- Verschiedene *Reflexionsmechanismen* für selbst anpassende Komponenten.
- Partielle Einschätzung - zur Behandlung der Ressource-Einschränkungen
- Koala implementiert alles in C.
- Zur einfachen Visualisierung und leichter Benutzung von nicht Informatikern (v.a. Elektrotechnikern) werden die Koala Komponenten und die Verbindungen als Schaltplan (ICs, Drähte ...) dargestellt.

2.3 Grundlegende Elementen (Bausteine)

Komponenten Eine Komponente ist ein gekapseltes Software-Stück. Es ist Entwicklungs- und Architektureinzelteil. Eine Komponente muss konfigurationsunabhängig sein, damit man sie später wieder benutzen kann.
Eine Komponente kommuniziert durch Schnittstellen mit ihrer Umgebung. Die bietet Funktionalität durch Definieren von Provide-Schnittstellen und benutzt selbst andere Funktionalität durch Definieren von Require-Schnittstellen. Alle äußerliche Funktionalität ist durch Require-Schnittstellen realisiert (im Gegensatz zu COM, wo viele notwendige Funktionalität schon im Code der Komponente explizit geschrieben ist), sogar den Zugriff auf das Betriebssystem oder andere globale allgemeine Services. In Koala wird eine Komponente durch ein Rechteck und eine Schnittstellen durch klein Quadrat mit einem Dreieck drin dargestellt(s. Abbildung 4).

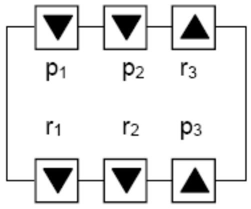
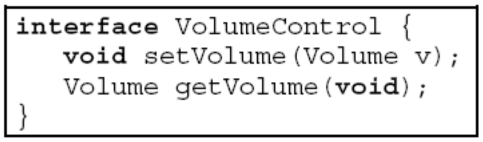


Abbildung 4. Beispiel für eine Komponente[12]; p1, p2, p3 sind Provide-Schnittstellen, r1, r2, r3 sind Require-Schnittstellen.

Die Dreiecke bestimmen die Richtung eines Funktionsaufrufs. Diese Notation ist absichtlich so gewählt, damit es leichter und verständlicher für die nicht Informatiker im TV-Bereich wird (ähnelt Schaltplan).
Ein Komponententyp ist eine an sich isolierte wiederverwendbare Komponente. Eine Komponenteninstanz ist die Erscheinung dieser Komponente in bestimmter Konfiguration.
Normalerweise sind die Komponenten nur ein mal instanziierbar (d.h. die kommen nur in einer bestimmten Konfiguration vor), aber man kann sie auch mehrfachinstanziierbar machen (explizit als solche deklarieren).

Jede Komponente (Komponententyp) hat zwei Namen, die global eindeutig sein müssen: ein vollständiger Vollname und ein kürzerer Name (Alias)[12].

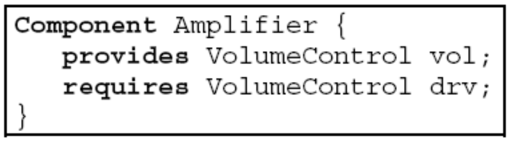
Schnittstellen Eine Schnittstelle ist kleine Sammlung von semantisch zusammengehörten Funktionen.
Ein Schnittstellentyp ist eine syntaktische und semantische Beschreibung von einer Schnittstelle, wobei eine Schnittstelleninstanz ist das Vorkommen der Schnittstelle innerhalb einer Komponente.
Ein Schnittstellentyp wird in Koala IDL beschrieben. Das ist eigentlich standarte IDL Notation, die ähnlich wie COM und Java Schnittstellendefinitionen ist. Man zählt einfach alle Funktionprototypen innerhalb der Schnittstelle in C Syntax auf(s. Abbildung 5).



```
interface VolumeControl {
   void setVolume(Volume v);
   Volume getVolume(void);
}
```

Abbildung 5. Beispiel für Schnittstellendefinition[12]

Die Schnittstellen beinhalten nur Funktionen. Typen innerhalb einer Schnittstelle werden automatisch freigegeben, d.h. wenn man eine bestimmte Schnittstelle benutzt/bietet, darf man auch die Typen benutzen, die drin benutzt worden sind. Die Konstanten werden wie Funktionen behandelt. Es gibt zwei Typen von Schnittstellen: Require-Schnittstelle und Provide-Schnittstelle. Die haben dieselbe Bedeutung wie diejenige in UML 2.0. Die Schnittstellen werden statisch bei der Entwurfsphase gebunden.
Ein Komponententyp wird in der Koala Komponentendefinitionssprache (CDL) beschrieben. Man soll jeweils die Namen von Schnittstellentyp und Schnittstelleninstanz aufzählen (s. Abbildung 6).



```
Component Amplifier {
   provides VolumeControl vol;
   requires VolumeControl drv;
}
```

Abbildung 6. Beispiel für Komponentendefinition[12]

Die Schlüsselwörter *provides* und *requires* bestimmen die Rolle der Schnittstelle innerhalb der Komponente. Die Komponenteninstanznamen sollen eindeutig innerhalb einer Komponente sein. Der Name einer Komponente oder der Name einer Schnittstellen müssen global eindeutig sein[12].

Konfiguration Man errichtet eine Konfiguration durch Instanziierung von Komponenten und Binden ihrer Schnittstellen.
Die Require-Schnittstelle muss immer zur genau einer Provide-Schnittstelle gebunden sein. Die Provide-Schnittstelle kann aber zur beliebig vielen (0..*) Require-Schnittstellen gebunden sein(s. Abbildung 7).

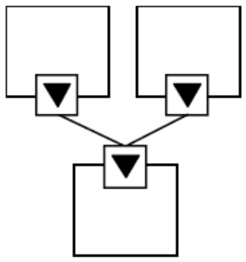
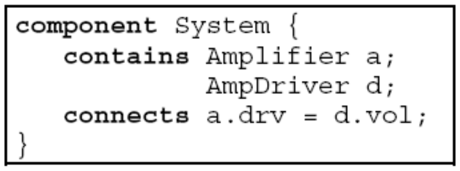


Abbildung 7. Beispiel für Konfiguration[12]

Eine Konfiguration wird auch in CDL geschrieben. Die Konfiguration besteht aus zwei Blöcke: Deklaration von der Komponenteninstanzen (mit Hilfe von dem Schlüsselwort *contents*) und Beschreibung des Schnittstellenverbindung (mit Hilfe von dem Schlüsselwort *connects*). Die Beschreibung sieht wie Hardwarekomponenten und die entsprechende netlists“[5] aus.
Die Komponenteninstanzen sollen eindeutige Bezeichner innerhalb einer Konfiguration haben[12].
Abbildung 8 stellt eine beispielsweise Konfiguration dar.



```
component System {
   contains Amplifier a;
            AmpDriver d;
   connects a.drv = d.vol;
}
```

Abbildung 8. Beispiel für Konfigurationsdefinition[12]

[5] Eine Datei, welche die Verbindungsinformation einer elektronischen Schaltung beschreibt. Ein netlist beschreibt die Konnektivitat zwischen Bestandteilen in einem gegebenen elektronischen System.

Implementation Die Komponenten werden in C implementiert. Das ist wegen der CE-Randbedingungen (Ressource-Beschränkung).
Also eine Komponente ist im Endeffekt eine Sammlung von C und header-Dateien in einem Verzeichnis. Andere Deteien und Zugriff auf den sind innerhalb des Verzeichnis erlaubt, aber das Referenzieren von anderen Dateien außerhalb des Verzeichnis ist verboten.
Sei f eine Funktion. f ist in der Provide-Schnittstelle p der Komponente C (der kurze Name von C sei c) als c_p_f implementiert. Die Funktion f wird aber in der Require-Schnittstelle r als r_f aufgerufen. Hier besteht die Frage, wie wird der Aufruf von r_f mit c_p_f gebunden ? Das wird eifach durch Hinzufügen von dem Macro `#define r_f(...) c_p_f(...)` gelöst. Diese Macros werden automatisch vom Koala Compiler (z.B. Koala 3.5.0.0) automatisch erzeugt. Er erzeugt von den CDL und IDL Dateien die nötige C und header Dateien.
Diese Technik ist sehr sinnvoll bei Anwendungen bis etwa 106 Zeilen von Code[12].

2.4 Speziale Elementen

Die Bausteine, die schon beschrieben wurden, reichen schon, um ein komplettes System zu bauen, aber man braucht weitere Elementen um die Vielfalt von Fähigkeiten modellieren zu können.

Compound-Komponenten Wenn man grosse Systeme (mit 100 oder mehr Komponenten) baut ist es schwer alle innerhalb einer Beschreibung zu binden. Die Beschreibung wird gross und unübersichtlich. Deswegen ist das Model von der Komponenten rekursiv, d.h. jede beliebige Konfiguration von Komponenten kann auch als (neue) Komponente mit Provide- und Require-Schnittstellen weiter benutzt werden.
Die Verbindungsregel lautet jetzt: Die Dreieckspitze einer Schnittstelle soll zu genau einer Dreieckbasis gebunden werden. Eine Dreieckbasis dagegen kann mit beliebig vielen Dreieckspitzen (0..*) gebunden werden. Diese Regel erlaubt sogar “Kurzschlüsseßwischen Schnittstellen, die am Rand einer Komponente liegen. Das letzt erlaubt eine Technik für reine Schichtenmodellierung ohne zusätzliches Implementierungs-overhead.
Abbildung 9 zeigt ein Beispiel für eine Compound-Komponente[12]. Da ist eine Konfiguration von zwei Komponenten wieder als eine neue Komponente (Compound-Komponente) zu betrachten. Die Kompound-Komponente wird durch ihre Schnittstellen benuzt und ihre innere Struktur ist nicht mehr von Bedeutung.

Glue-Schnittstellen Nicht immer ist eine direkte Verbindung zwischen Schnittstellen möglich. Solche direkte Verbindung braucht komplette Anpassung der

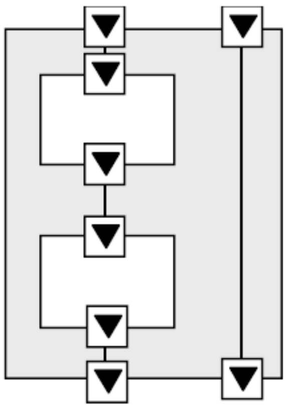


Abbildung 9. Beispiel für Compound-Komponente[12]

beiden Schnittstellen und das ist sehr selten der Fall.
Dafür werden in Koala Module benutzt. Ein Modul hat eindeutigen Namen innerhalb des Komponententyps, in dem er gebraucht wird.
Wenn die Dreieckspitze einer Schnittstelle zu einem Modul gebunden ist, dann sind die Funktionen von der Schnittstelle in diesem Modul implementiert. Umgekehrt, wenn die Dreieckbasis einer Schnittstelle ist zu einem Modul gebunden, dann werden die Funktionen der Schnittstelle möglicherweise von dem Modul benutzt(s. Abb.10: die Provide-Schnittstelle(oben) bietet z.B. mehrere Funktionen, als die Require-Schnittstelle(unten) benutzt).
Koala generiert für jeden Modul eine header-Datei. Der Entwickler der Komponente fügt eine oder mehrere C Dateien mit der Implementation dieser Funktionen zu[12]. Abbildung 10 stellt ein Beispiel für Anpassung von Schnittstellen durch Module dar. Da sind zwei Komponenten mit Hilfe von einem Glue-Modul verbunden. Die Schnittstellen (Require und Provide) unterscheiden sich (z.B. es wird eine Funktion mehr angeboten) und der Glue-Modul ermöglicht die Verbindung zwischen denen.

Mehrfache Instanziierung Die meisten Komponenten kommen nur einmal (wirden für einen bestimmten Zweck gebraucht) innerhalb eines System-Konfiguration vor und deswegen sind auch nur einmal instanziierbar. Es kann aber auch sein, dass eine Komponente irgendwelche Services implementiert und daher in verschiedene Stellen im System mehrfach gebraucht werden soll. In diesem Fall muss sie auch mehrfach instanziiert werden. Prinzipiell ist das in Koala möglich, aber die Komponenten sollen explizit als mehrfach instanziierbar (*multiple instantiable (MI)*) definiert.
Koala implementiert dies folgendes. Jede Provide-Funktion in einer MI-Komponente hat extra noch einen Parameter, der ein Zeiger auf Instanzdatenstruktur ist. Diese Datenstruktur wird von Koala automatisch generiert. Die ganze Verbindung

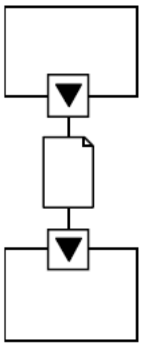


Abbildung 10. Beispiel für Anpassung zweier Schnittstellen mit Hilfe von Glue-Modul[12]

wird bei Instanziierung gemacht und bleibt für die Benutzung der Funktionen in der Require-Schnittstellen verborgen.
Dies wurde ähnlich wie die Implementationstechnik für Klassen und Objekten in C++ gemacht.

Package Packages (wie in Java) ermöglichen gleiche globale Namen für verschiedene Komponententypen bzw. Schnittstellentypen. Die Namen müssen eindeutig innerhalb eines Package bleiben.

2.5 Mannigfaltigkeit und Binden von Komponenten

Alle bisher betrachtete Konstrukte und Techniken bringen die Komplexität zustande. Zur Handhabung der Mannigfaltigkeit von CE-Geräte (viele kleine Unterschiede/Abweichungen in Hardware oder in der Benutzeroberfläche) braucht man die folgende weitere Konstruktionen. Man unterscheidet zwischen interne Mannigfaltigkeit (innerhalb einer Komponente - z.B. Komponenten, die sich leicht in der Provide-Schnittstellen unterscheiden) und strukturelle Mannigfaltigkeit (zwischen den Komponenten - Ermöglichen von verschiedenen sich leicht unterscheidenden Konfigurationen auf der Basis von dieselben Komponenten).

Kompatibilität von Schnittstellen Die Entwickler von Koala folgen das COM Konzept für Schnittstellen: Wenn eine Schnittstelle schon definiert ist, darf sie nicht mehr geändert werden. Doch die neuen Generationen von Komponenten bräuchten die Definition von neuen Versionen (Arten) einer schon existierenden Schnittstelle (die sich nur leicht von dem usprünglichen Schnittstellen-Typ unterscheidet - z.B. neue Funktion drin). Dann haben wir beim Konfigurieren das Problem mit der Kompabilität zwischen den verschiedenen Variationen einer Schnittstelle. Seien I1 und I2 zwei sich leicht unterscheidenden Arten von der Schnittstelle I. Sei I2 eine Obermenge von der Funktionen in I1. Das Problem jetzt ist, wie kann man die Require-I1 zur Provide-I2 binden. Man kann das Problem durch Glue-Code lösen, aber das führt zu Syntax- und Code-overhead. Die elegante Koala's Lösung ist einfach die Verbindung zwischen Dreieckspitze der Require-I1 und Dreieckbasis der Provide-I2 zu erlauben, wenn die erste Schnittstelle (I1) Untertyp der zweiten (I2) ist. D.h., dass die Funktionen in I1 eine Teilmenge von diesen in I2 sind (genau unserer Fall)[12].

Diversity-Schnittstellen Damit die Komponenten mehrfach verwendbar sein können, müssen sie auch möglichst viele Parameter beinhalten. Das ist wegen Ressource-Beschränkung(v.a. Speicherplatz - RAM[6] und ROM[7]) nicht direkt möglich. In Koala werden diese nötige Parameter durch s.g. Diversity-Schnittstellen realisiert. Eine Diversity-Schnittstelle ist eine Require-Schnittstelle, die Parameter (s.g. Diversity-Parameter) beinhaltet. Anhand dieser Parameter kann die Schnittstelle verändert werden. Da aber eine Schnittstelle keine Parameter haben darf, sind die als Funktionen in der Require-Schnittstelle deklariert. Die Implementierung dieser Funktionen bleibt ausserhalb der Komponente (im Gegensatz zu Visual Basic, wo die *Properties* /eine Art Paramter/ innerhalb der Komponente implementiert sind)[12].
Die Diversity-Schnittstellen ermöglichen die interne Mannigfaltigkeit. Abbildung 11 zeigt die graphische Notation für Diversity-Schnittstellen. Die Diversity-Schnittstelle hat dasselbes Symbol wie eine normalle Require-Schnittstelle, nur das Kästchen, wo das Dreieck sich befindet, ist rot gefüllt(auf Abb.11 ist es leider weiss).

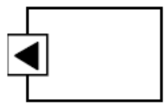


Abbildung 11. Diversity-Schnittstelle[12]

Switches Switches behandeln die strukturelle Mannigfaltigkeit. Angenommen, dass die Komponente A die Schnittstelle B1 in einem und die Schnittstelle B2 in einem anderem Produkt braucht(z.B. universale Software für zwei ähnliche Fernsehermodelle). Man kann schon zwei Konfigurationen machen (eine mit B1 und eine mit B2) und dann die als Compound-Komponenten weiter nutzen, aber wenn A gross ist, dann wäre das zu viel Code-Duplikation. Koala bietet die Switches an dieser Stelle. Das ist eine Art Modul, das zwischen der Require-Schnittstelle von A und den Provide-Schnittstellen von B1 und B2 steht(s. Abbildung 12).

[6] Random Access Memory
[7] Read-only Memory

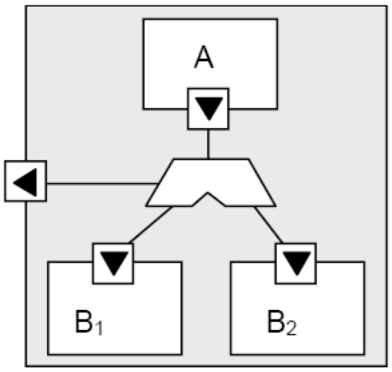


Abbildung 12. Beispiel für Switch-Einsatz[12]

Der Switch kann Verbindungen zwischen seiner Switch-Spitze und einer Komponente, die am Switch-Boden gebunden ist, routen. Die Switchspitze soll mit genau einer Dreieckspitze einer Schnittstelle gebunden sein. Jeder von seinen Switchboden kann mit unterschiedlicher Dreieckbasis verbunden werden.
Der Switcheinstellungen werden durch eine Diversity-Schnittstelle gesteuert und so wird die gewünschte Kombination gewählt[12].

Optionale Schnittstellen Sei das folgende Szenario: Es gibt neue Version für eine Komponente, die eine neue Provide-Schnittstelle bietet und eine neue Require-Schnittstelle erfordert. Man kann den Komponententyp anders eindeutig benennen, aber die Komponente wird nicht automatisch in die alten Konfigurationen eingesetzt.
Deswegen ist in Koala die Hizufügung von Schnittstellen in schon existierende Komponenten erlaubt, wenn die Schnittstellen als optional deklariert worden sind(s. Abbildung 13).

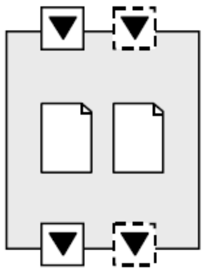


Abbildung 13. Komponente mit optionalen Schnittstellen. Die optionale Schnittstelle wird mit gestrichelter Box dargestellt[12]

Diese Technik ist ähnlich wie das COM Query Interface Mechanismus. Jede optionale Schnittstelle hat extra eine boolische Funktion namens *iPresent*. Die ist *true*, wenn die Dreieckspitze der optionalen Schnittstelle mit der Dreieckbasis einer nicht optionalen (standard) Schnittstelle verbunden wird. Die Funktion ist *falsch*, falls die Dreieckspitze überhaupt nicht gebunden ist. Wenn die Dreieckspitze mit einem Modul verbunden ist, dann ist die Funktion in dem Modul definiert. Wenn die Dreieckspitze der optionalen Schnittstelle mit einer weiteren optionalen Schnittstelle verbunden ist, dann wird der Funktionswert von der anderen optionalen Schnittstelle geerbt. In der Require-Schnittstelle kann man *iPresent* aufrufen, um festzustellen, ob die Funktionen zum Laufzeit vorhanden sind und ggf. die benutzen[12].

3 Das Gemeinsame WebAudioStore-Beispiel

Das Koala Komponenten Modell wurde für Software für Fernsehgeräte entwickelt und deswegen ist für das Modellieren von Anwendungen wie das WebAudioStore-Beispiel nicht geeignet.
Abbildung 14 zeigt als Beispiel ein Teil einer möglichen Fernsehsoftwareplattform.
Diese Konfiguration hat vier Komponenten: FrontEnd, TunerDriver, HipDriver, HopDriver und BackEnd. Das Ganze ist als eine Compound-Elemente gebaut. Es gibt zwei Provide-Schnittstellen ('pprg'- Programm, 'ppic'-Bild und 'pini'- Initialisation) und zwei Require-Schnittstellen ('fast' und 'slow'). Der TunerDriver und der HipDriver sind zum schnellen I2C[8]-Service verbunden (s. die Require-Schnittstelle 'fast'). Der HopDriver ist zum langsamen I2C-Service verbunden(s. die Provide-Schnittstelle 'slow'). Der Glue-Modul 'm' ermöglicht die Verbindung der verschiedenen komponenten-spezifischen Initialisation-Schnittstellen.

4 Bewertung und Zusammenfassung

Ich habe sehr wenig Werkzeuge für Koala gefunden. Auf der offizielle Koala-Seite[1] sind die folgende Koala Werkzeuge zu finden:

– *Koala 3.5.0.0* - Koala Compiler (befehlorientiert) für Windows. Generiert C und header-Dateien von den Koala Beschreibungen und Definitionen.
– *Koala Maker 2.4.0.0* - liest die C und header-Dateien und generiert *make file*.
– *Koala Modell* - Parser von Koala und Objektmodell als VB-kompatible COM Komponente.
– *Koala Viewer* - Programm zur Visualisierung von Koala Komponentendiagrammen.

[8] I2C (fur Inter-Integrated Circuit, gesprochen I-Quadrat-C bzw. I-squared-C) ist ein von Philips Semiconductors entwickelter serieller Bus fur Computersysteme. Er wird benutzt, um Gerate mit geringer U"bertragungsgeschwindigkeit an ein Embedded System oder eine Hauptplatine anzuschlie?en.

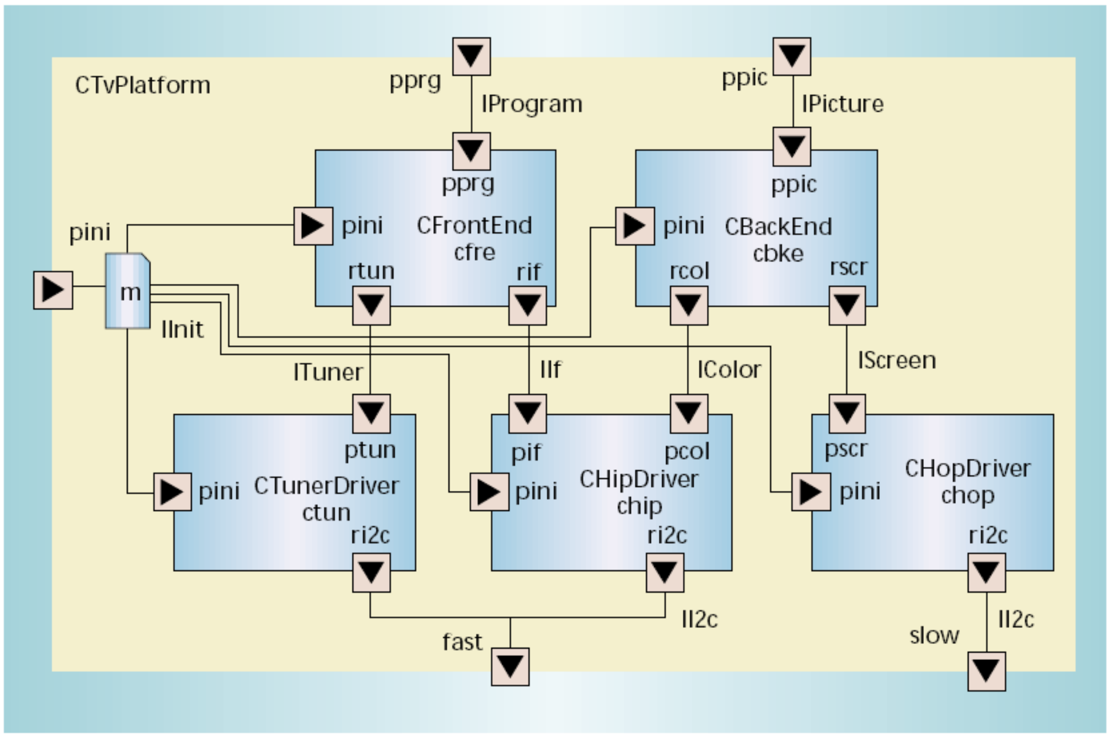


Abbildung 14. Fernsehsoftwareplattform[10]

Die Vorteile von Koala sind:

– Ideal für Ressource-begrenzte-Systeme (CE benutzen meistens billige Hardwarekomponenten, damit die Kosten gering bleiben). Das war eigentlich eine der Hauptaufgaben bei der Entwicklung von Koala.
– Das Modell bietet eine grösse Wiederverwendbarkeit der Komponenten.
– Die Wartung der Komponenten ist einfach.
– Das Modell ist sehr begreiflich, sogar für Leute, die sich nur mit Hardware auskennen.
– Eine Komponente(im Sinne von CBSE) ist quellcode orientiert, d.h. keine binäre Komponenteneinheiten. Dies erleichtert das Testen (white box vs. black box), das Finden von Fehlern und sowie die Erweiterungen existierender Komponenten.
– Koala hat statische Konfiguration - das ist sehr wichtig für Bereiche wie CE (abgesehen vom Office/Internet Bereich). So werden die Analyse, das Testen und die Ressourcenverbrauchschätzung besser uterstützt.

Koala hat aber auch und einige Nachteile:

– Koala unterstützt die Analyse von Laufzeiteigenschaften des System nicht.
– Koala hat keine explizite Uterstützung für Testen und Debugging. Diese Aufgabe wird dadurch erleichtert, dass die Komponenten quellcode basiert sind und das Interaktionsmodell zwischen den sehr einfach ist.

Das Koala Komponenten Modell ist sehr stark in der Entwicklung von CE-Familien-Geräte(Fernseher, DVD-Player, Videorekorder) bei Philips eingeprägt. Aus der Sicht der industriellen Anforderungen für Eingebet Systeme ist Koala Modell mit PECOS Komponententechnologie auf der ersten Stelle für Komponentenmodell und bieten sehr guten Support. Natührlich gibt es kein SSilver bulletünd kein Modell kann alle Anforderungen ableisten.
Koala ist für gezieltes Betriebssystem implementiert. Spezifischer Compiler routet alle Komponenten untereinander und zu dem Betriebssystem (durch die Koala Anschlüsse).
Obwohl das nächste Ziel der Entwickler das Unterstützung von UML 2.0 Komponentendiagrammen in Koala ist, bleibt das verbreiten von Koala außerhalb Philips noch unklar[7].

Literatur

1. Die offizielle Koala Seite im Internet:
`http://www.extra.research.philips.com/SAE/koala/`, Zugriff am 24.02.2007
2. Muskens,J., Chaudron,M.R.V and Lukkien,J.J.: *A Component Framework for Consumer Electronics Middleware*, Department of Mathematics and Computer Science, Technische Universität Eindhoven
3. *Component-based software engineering for consumer electronics*:
`http://www.nexwave-solutions.com/frontend/asps/technology_whitepaper.asp`, Zugriff am 24.02.2007

4. de Jonge, M.: *Introduction from the session chair Component Models and Technologies*, Proceedings of the 29th EUROMICRO Conference New Waves in System Architecture"(EUROMICRO'03), IEEE 2003
`http://ieeexplore.ieee.org/iel5/8716/27589/01231565.pdf`, Zugriff am 24.02.2007
5. Lau, Kung-Kiu and Whan, Zheng: *A taxonomy of software component models*
6. Lee, E., Neuendorffer, St.: *Concurrent models of computation for embedded software*, Technical Memorandum UCB/ERL M04/26, University of California, Berkeley, July 22, 2004
7. Müller, A., Akerholm, M. et al.: *An industrial evaluation of component technologies for embedded systems*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE
8. *The Koala Language 1.0* (in PDF format):
`http://www.extra.research.philips.com/SAE/koala/docs/KoalaLanguage.pdf`, Zugriff am 24.02.2007
9. *The Koala Yellow Pages v1.3a* (in Windows HTML Help format):
`http://www.extra.research.philips.com/SAE/koala/docs/KoalaYellowPages.chm`, Zugriff am 24.02.2007
10. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, March 2000, p78-85
`http://www.liacs.nl/~marcello/CBSE/koala.pdf`, Zugriff am 24.02.2007
11. van Ommering, R.: *Building product populations with software components*, ISBN 90-74445-64-0, December 2004
`http://ieeexplore.ieee.org/iel5/7889/21739/01007973.pdf`, Zugriff am 24.02.2007
12. van Ommering, R.: *Koala, a component model for consumer electronics product software*, In: Frank v. d. Linden (Ed.): ARES '98, LNCS 1429, pp. 76-86, 1998, Springer-Verlag Berlin 1998
`http://www.extra.research.philips.com/SAE/koala/pub/ares2_rvo.pdf`, Zugriff am 24.02.2007
13. van Ommering, R.: *The Koala Language v2.0* (in PDF format), August 2003,
`http://dissertations.ub.rug.nl/FILES/faculties/science/2004/r.c.van.ommering/appendix.pdf`, Zugriff am 24.02.2007
14. Koziolek, H., Happe, J., Becker, S.: Parameter Dependent Performance Specifications of Software Components. Technical report, Graduate School Trustsoft, University of Oldenburg, Germany and Chair for Software Design and Quality, University of Karlsruhe, Germany (2006)

XPCOM Cross Platform Component Object Model

Fouad ben Nasr Omri

Institute for Program Structures and Data Organization (IPD)
Chair for Software Design and Quality (SDQ)

Abstract. Component-based software engineering aims to design and to construct software systems using reusable software components. This software technology has attracted attention beacause it reduce considerably developmental costs. Components are indeed considered as higher level of abstraction than objects. They are defined as a unit of composition that can be independently exchanged in the form of object code. XPCOM (Cross Platform Component Object Model) is a framework for writing cross platform modular software. In this seminar, we present XPCOM starting by introducing XPCOM component basics then dicussing these components from the perspective of the software developer. We present the architecture of XPCOM and the utilities it brings for component based software developers.

1 Introduction

The Cross Platform Component Object Module (XPCOM) is a framework which aims at the development of component based softwares. XPCOM provides several tools, libraries and services allowing the developer to write modular cross-platform code and to replace or upgrade a component without breaking or having to recreate the application.

The most famous use of XPCOM is in Netscape's browser package - Communicator. When Netscape decided to redo their browser in 1998, its goal was to write an open-source suite of modular components that comprised a complete Internet platform [1].

This seminar is organised as follows: Section 2 present the differences and the anology between XPCOM and Microsoft COM. Section 3 shows the necessary basics of XPCOM and section 4 gives Mozilla as an example of a product made of XPCOM. The internal architecture of XPCOM from the perspective of the software developer is provided in section 5, while the modelling example is listed in section 6.

2 Motivation: Microsoft COM and Mozilla XPCOM

It may wonder the user of XPCOM or Microsoft COM whether XPCOM is like Microsoft COM [1]. The Component proxy support is one area of distinction. A component proxy is a fake component, which impersonates another component that cannot be accessed directly by the application using it [1]. This would be the case for example when the component exists in another process or on another machine.

Microsoft COM supports an elaborate proxy mechanism, coordinating how applications access components either they run as separate programs or even on different machines. Microsoft COM allows creating components with different threading models or with a specific threading model. Components can be created to run inside the application or as a seperate application. "This built-in proxy mechanism in Microsoft COM provides the appropriate amount of thread safety for components that have their own threading constraints" [1].

XPCOM is tailored toward providing component support at the application level. Accessing a component remotely is done by writing a proxy to marshal data back and forth to the remote object [2].

Microsft COM and XPCOM components are not compatible or interchangeable. A wrapper or a glue code is required for the two to operate together. A good example is the embedding wrapper for Mozilla that allows the browser engine to appear as an Microsoft COM ActiveX control while the browser engine internally operates on XPCOM components [1]. The biggest contrast between XPCOM and Microsoft COM is that XPCOM technology is open source, the source code for the libraries that make up the XPCOM architecture are fully available to inspect, trace, and debug.

3 Overview of XPCOM

3.1 Interfaces

All XPCOM interfaces are defined with the Interface Definition Language (IDL). IDL provides a language-neutral way to describe the public methods and properties of a component. Interfaces allow developers to encapsulate the implementation and inner workings of their software. They represent the endpoints of the communication channels between different components. An interface defines the expected behaviour and expected responsibilities so that other software and components knows how to use the component. Applications use the interface to interact with each other and the system.

For software maintainability and reliability the abstraction between component boundaries is crucial. In XPCOM, developers are shielded from the inner workings of components and rely on the interface to provide access to the needed functionality [2]. The base interface in XPCOM is the *nsISupports* interface, mother of all interfaces in XPCOM. Figure 1 shows the *nsISupports* interface.

Two fundamental issues in component and interface-based programming are *component lifetime* called also *object ownership*, and *interface querying.* The *nsISupports* provides solutions to both of these issues for XPCOM developers.

Object Ownership Components in XPCOM may implement any number of different interfaces. "Components must keep track of how many references to them clients are maintaining and delete themselves when that number reaches zero" [2]. An integer inside the component indicates the reference count which is automatically incremented when the client instantiates the component. The component deletes itself if its reference count hits zero. The decrementation of the reference count is the responsibility of the client. In this context the system of reference counting can cause some real problems when, for example, a client uses an interface and forgets to decrement the reference count. This may lead to a memory leak. "The system of reference counting is, like many things in XPCOM, a contract between clients and implementations. It works when people agree to it, but when they don't, things can go wrong" [2].

The *nsISupports* interface supplies the basic functionality for dealing with interface discovering and reference counting. The following members of the interface `QueryInterface`, `AddRef` and `Release` provide respectively the functionality for getting the right interface from an object, incrementing the reference count and releasing objects one thy are no more used.

```
class Sample: public nsISupports {
      private: nsrefcnt mRefCnt;
      public: Sample();
      virtual ~Sample();
      NS_IMETHOD QueryInterface(const nsIID &aIID,
      void **aResult);
      NS_IMETHOD_(nsrefcnt) AddRef(void);
      NS_IMETHOD_(nsrefcnt) Release(void);
};
```

Fig. 1. The nsISupports Interface [2]

Object Interface Discovery Inheritance is a very important topic in object oriented programming. In XPCOM, all classes implement the *nsISupports* interface. The *QueryInterface* feature of the *nsISupports* interface allows clients to find and access different interfaces based on their needs. XPCOM uses the special *QueryInterface* method that casts the object to the right interface if that interface is supported.

A tool named *"uuidgen"* generates for every interface a 128 bit *universally unique identifier (UIID).* In the context of an interface, this number is called

```
Sample::Sample()
{
  // initialize the reference count to 0
  mRefCnt = 0;
}
Sample::~Sample()
{
}

// typical, generic implementation of QI
NS_IMETHODIMP Sample::QueryInterface(const nsIID &aIID,void **aResult)
{
  if (aResult == NULL) {
    return NS_ERROR_NULL_POINTER;
  }
  *aResult = NULL;
  if (aIID.Equals(kISupportsIID)) {
    *aResult = (void *) this;
  }
  if (*aResult != NULL) {
    return NS_ERROR_NO_INTERFACE;
  }
  // add a reference
  AddRef();
  return NS_OK;
}

NS_IMETHODIMP_(nsrefcnt) Sample::AddRef()
{
  return ++mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) Sample::Release()
{
  if (--mRefCnt == 0) {
    delete this;
    return 0;
  }
  // optional: return the reference count
  return mRefCnt;
}
```

Fig. 2. Implementation of nsISupports Interface [2]

IID. The client passes the IID of an interface into the *QueryInterface* method of an object, if the client wants to find out if the object supports the given inteface. In the Case that the object supports the requested interface, the object adds a reference to itself and passes back a pointer to that interface, otherwise an error is returned [2].

The basic encapsulation of the IID is a class named *nsIID*. The first parameter of the *QueryInterface* is a reference to the nsIID class. The IID argument is then checked against the nsIID class. If there is a match, the object's *this* pointer is cast to *void*, the reference count is incremented, and the interface returned to the caller. If there is no match, the class returns an error and sets the out value to *null* [2].

3.2 XPCOM Identifiers

In order to simplify the process of dynamically finding, loading, and binding interfaces, all classes and interfaces are assigned IDs. In addition to the IID interface identifier, XPCOM uses other types of identifiers naimly the contract ID and the class ID.

The Class Identifier A class ID or CID identifies a class or component in the same way that an IID uniquely identifies an interface. For example the CID for *nsISupports* is :`00000000-0000-0000-c000-000000000046` "If the class to which a CID refers implements more than one interface, that CID guarantees that the class implements that whole set of interfaces when it's published or frozen" [2].

The Contract ID A contract ID is a human readable string. It is used to access a component. The format of the contract ID is the *domain* of the component, the *module*, the *component name*, and the *version number*, separated by slashes. A contract ID refers to an implementation but is not bound to any specific implementation as the CID. It only specifies a set of interfaces that it wants implemented by any number of different CIDs.

3.3 Factories

In order to instantiate objects is typically the *n*ew constructor used:
```
 SomeClass* component = new SomeClass();
```
The *factory design pattern* can be used to encapsulate object construction. The factory determines the actual concrete type of the object to be created. It creates objects without exposing clients to the implementations and initializations of those objects. In XPCOM, factories are implementations of the *nsIFactory* interface, and they use a factory design pattern to abstract and encapsulate object construction and initialization.

In general, factories needs to store state, at minimum they need to store information about which objects they have created, or whether an object is a *singleton*. If a factory creates a singleton object, subsequent calls to the factory must return the same object. A class that assures a maximum of one object of its type at a given time and provides a global access point to this object is a Singleton pattern.

A class that implements the functionality of a factory can derive from the *nsISupports* interface, which allows managing the lifetime of the factory objects. Deriving from the *n*sISupports has other benefits as for example supporting other interfaces including the *nsIClassInfo* which provides information about underlying implementation such as the language of implementation (C++, JavaScript, etc.), or a list of the interfaces implemented by a class.

3.4 XPCOM Services

There is a kind of object known as a *s*ervice, of which there is there is always only one copy. If a client wants to access the functionality provided by a service, they talk to the same instance of that service. The client does not create instances of them because only one should exist. Services provide general functions which either get or set global data or perform operations on other objects. Instead of calling `createInstance()`, the client call `getService()` to get a reference to the service component. Other than that, services are not very different from other components. "Providing this single point of access to functionality is what the singleton design pattern is for, and what services do in an application (and in a development environment like XPCOM)" [2].

In XPCOM, in addition to the component support and management, there are many other services helping writing cross platform components. Among these services we can cite the cross platform file abstraction providing uniform and powerful access to files, directory services giving the developer the ability to maintain the location of application and system-specific locations, an event notification system that allows passing of simple messages, and memory management in order to ensure that every client is using the same memory allocator [2].

3.5 XPCOM Types [2]

Method Types

- `NS_IMETHOD`: Method declaration return type. XPCOM method declarations should use this as their return type
- `NS_IMETHODIMP`: Method Implementation return type. XPCOM method implementations should use this as their return time.
- `NS_IMETHODIMP_(type)`: Special case implementation return type. Some methods such as AddRef and Release do not return the default return type. This exception is regrettable, but required for COM compliance.
- `NS_IMPORT`: Forced the method to be resolved internally by the shared library.
- `NS_EXPORT`: Forces the method to be exported by the shared library.

Reference Counting

– `NS_ADDREF`: Calls AddRef on an nsISupports object.
– `NS_IF_ADDREF`: Same as above but checks for null before calling AddRef.
– `NS_RELEASE`: Calls Release on an nsISupports object.
– `NS_IF_RELEASE`: Same as above but check for null before calling Release.

Status Codes

– `NS_FAILED`: Return true if the passed status code was a failure.
– `NS_SUCCEEDED`: Returns true is the passed status code was a success.

Variable Mappings

– `nsrefcnt`: Default reference count type. Maps to an 32 bit integer.
– `nsresult`: Default error type. Maps to a 32 bit integer.
– `nsnull`: Default null value.

Common XPCOM Error Codes

– `NS_ERROR_NOT_INITIALIZED`: Returned when an instance is not initialized.
– `NS_ERROR_ALREADY_INITIALIZED`: Returned when an instance is already initialized.
– `NS_ERROR_NOT_IMPLEMENTED`: Returned by an unimplemented method.
– `NS_ERROR_NO_INTERFACE`: Returned when a given interface is not supported.
– `NS_ERROR_NULL_POINTER`: Returned when a valid pointer is found to be nsnull.
– `NS_ERROR_FAILURE`: Returned when a method fails. Generic error case.
– `NS_ERROR_UNEXPECTED`: Returned when an unexpected error occurs.
– `NS_ERROR_OUT_OF_MEMORY` : Returned when a memory allocation fails.
– `NS_ERROR_FACTORY_NOT_REGISTERED`: Returned when a requested class is not registered.

3.6 Writing Components in Different Languages

For languages other than C++ it is crucial to have a programmatic control and access to the interfaces in order to call their methods. Interfaces are defined in a cross-platform language neutral development environment with the *interface definition language* (IDL). XPCOM uses its own variant of the IDL, which is called XPIDL. XPIDL has some limitations, it doesn't support the multiple inheritence of interfaces and it necessitates that method names are unique. However, XPIDL is offering some important functionalities. XPIDL allow the user to generate a binary representation of interfaces. This representaion is called *type library* and is a file with the extension *.xpt*. This is what the *XPConnect* responsible for. "XPConnect is the layer of XPCOM that provides access to XPCOM components from languages such as JavaScript" [2]. An interface is called *r*eflected when its implementing component is accessed from a language other than C++.

4 XPCOM as Foundation for Mozilla

All the functionality of the Mozilla browser such as searching, managing cookies, bookmarks and others are defined in XPCOM components. In the following section we want to present some frequently used components of XPCOM.

4.1 Cookie Manager

If the user accesses the Cookie Manager dialog to organize, view or delete cookies, the user uses the *CookieManager* component. The functinality of the *CookieManager* component is available through the interface `nsICookieManager` interface which presents the following public methods:

- `removeAll`: removes all cookies from the cookie list.
- `enumerator`: enumerates through the cookie list.
- `remove`: removes a cookie from the cookie list.

Figure 4.1 [2] shows how could the *CookieManager* component be called through JavaScript. "The contractual arrangements that XPCOM enforces open up the way to *binary interoperability*- to being able to access, use, and reuse XPCOM components at run-time." [2].

4.2 The WebBrowserFind Component

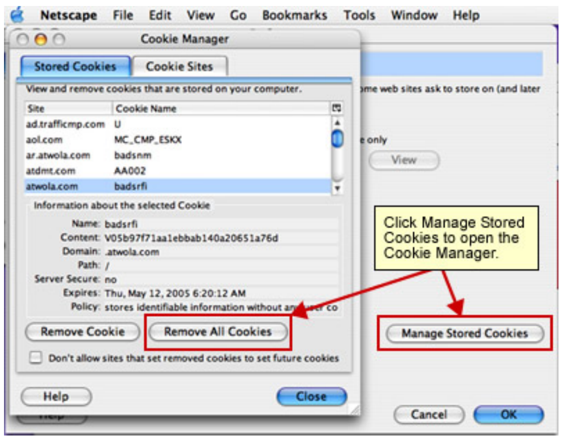
The *W*ebBrowserFind component is used in high-level browser, providing the following fiunctionalities through the `nsIWebBrowserFind` interface [2]:

- `findNext`: finds the next occurence of the search string.
- `findBackWards`: a boolean attribute adjusting `findNext()` to search backwards up the document.
- `searchFrames`: a boolean attribute indicating whether to search subframes of current document.
- `matchCase`: a boolean attribute indicating whether to match case in the search.
- `entireWord`: a boolean attribute specifying whether the entire word should be matched or not.

5 XPCOM Internals

5.1 Creating Components

The most common type of XPCOM components is the ones written in C++ and compiled into a shared library, a DLL on Windows systems and DSO on Unix [2]. It is important for the developer to understand the relationship between the shared library containing the component he wrote and the XPCOM framework. This relationship is illustrated in figure 6.



View and remove cookies that are stored on your computer.

Site	Cookie Name
ad.trafficmp.com	U
aol.com	MC_CMP_ESKX
ar.atwola.com	badsnm
atdmt.com	AA002
atwola.com	badsrfi

Information about the selected Cookie

Name:	badsrfi
Content:	V05b97f71aa1ebbab140a20651a76d
Domain:	.atwola.com
Path:	/
Server Secure:	no
Expires:	Thu, May 12, 2005 6:20:12 AM
Policy:	stores identifiable information without an[illegible] co

☐ Don't allow sites that set removed cookies to set future cookies

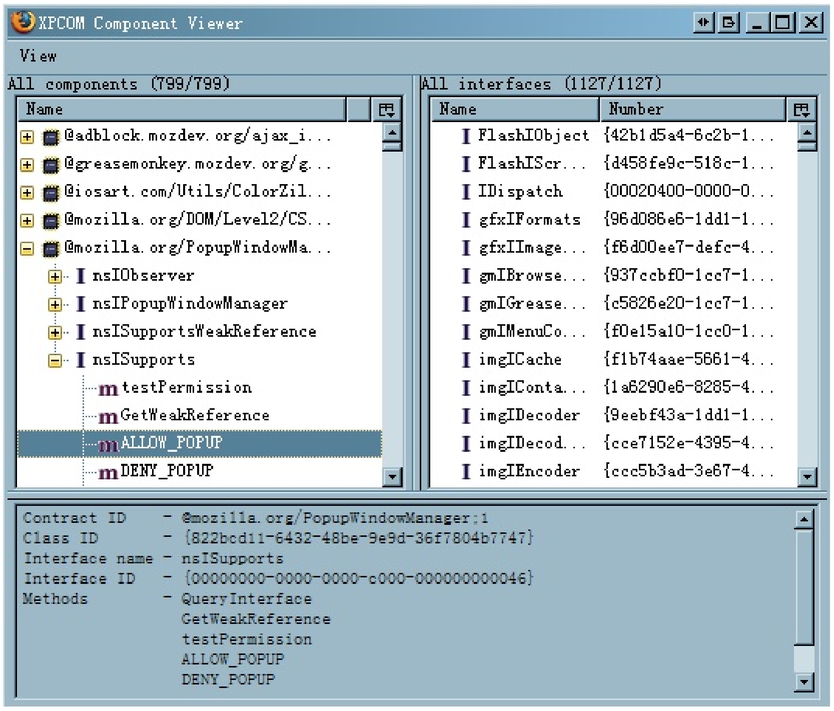
…me web sites ask to store on (and later

…e only

Fig. 3. Cookie Manager Dialog

```
var cookiemanager = Components.classes["@mozilla.org/
             cookiemanager;1"].getService();
    cookiemanager = cookiemanager.QueryInterface
        (Components.interfaces.nsICookieManager);
// called as part of a largerDeleteAllCookies() function
        function FinalizeCookieDeletions() {
    for (var c=0; c<deletedCookies.length; c++) {
      cookiemanager.remove(deletedCookies[c].host,
                           deletedCookies[c].name,
                          deletedCookies[c].path);
                          }
              deletedCookies.length = 0;
                        }
```

Fig. 4. Getting the CookieManager Component in JavaScript [2]



XPCOM Component Viewer

View

All components (799/799)

```
Name
+ @adblock.mozdev.org/ajax_i...
+ @greasemonkey.mozdev.org/g...
+ @iosart.com/Utils/ColorZil...
+ @mozilla.org/DOM/Level2/CS...
- @mozilla.org/PopupWindowMa...
    + I nsIObserver
    + I nsIPopupWindowManager
    + I nsISupportsWeakReference
    - I nsISupports
        m testPermission
        m GetWeakReference
        m ALLOW_POPUP
        m DENY_POPUP
```

All interfaces (1127/1127)

Name	Number
FlashIObject	{42b1d5a4-6c2b-1...
FlashIScr...	{d458fe9c-518c-1...
IDispatch	{00020400-0000-0...
gfxIFormats	{96d086e6-1dd1-1...
gfxIImage...	{f6d00ee7-defc-4...
gmIBrowse...	{937ccbf0-1cc7-1...
gmIGrease...	{c5826e20-1cc7-1...
gmIMenuCo...	{f0e15a10-1cc0-1...
imgICache	{f1b74aae-5661-4...
imgIConta...	{1a6290e6-8285-4...
imgIDecoder	{9eebf43a-1dd1-1...
imgIDecod...	{cce7152e-4395-4...
imgIEncoder	{ccc5b3ad-3e67-4...

```
Contract ID    - @mozilla.org/PopupWindowManager;1
Class ID       - {822bcd11-6432-48be-9e9d-36f7804b7747}
Interface name - nsISupports
Interface ID   - {00000000-0000-0000-c000-000000000046}
Methods        - QueryInterface
                 GetWeakReference
                 testPermission
                 ALLOW_POPUP
                 DENY_POPUP
```

Fig. 5. XPCOM Component Viewer

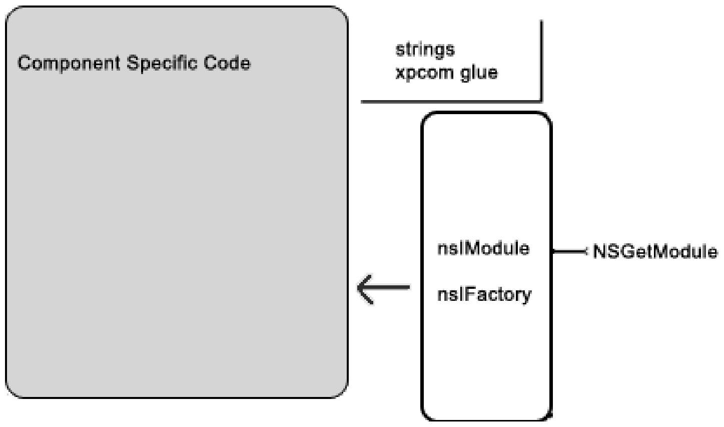


Fig. 6. A Component in the XPCOM Framework [2]

A component or a module which is compiled into a library must export a single method named `NSGetModule`. "This method gets called during the registration and unregistration of the component, and when XPCOM wants to discover what interfaces or classes the module/library implements" [2].

The *component directory* of an XPCOM application contains modules where the components reside. Those modules are defined in shared library files. A set of default libraries stored in this components directory makes up a typical Gecko installation, providing functionality that consists of networking, layout, composition, a cross-platform user interface and others [2].

A more basic view of the relationship betwwen the components and the files and interfaces that define them is illustrated in figure 7

5.2 Accessing Components

It is important to understand the XPCOM intialization process in order to understand why and when a new created component library gets called. XPCOM starts when an application makes a call to initialize it. For example a web browser that embeds Gecko initializes XPCOM at startup through the embedding APIs. The main purpose of the API at the startup level is to change which *component directory* XPCOM searches when it looks for XPCOM components [2].

In order to track information about the components to a local system, XPCOM is using special files named *manifests*. The are two types of manifests used by XPCOM:

1. Component Manifests: is a file listing all registered components and storing details on what each component can do. Directly when XPCOM starts up, it looks up for the component manifest and reads it into an in-memory database: "The component manifest is a mapping of files to components and components to classes" [2]. It contains the following information:
   - Location on disk of registered components with file size.
   - Class ID to Location Mapping.
   - Contract ID to Class ID Mapping.
2. Type Library Manifests: is a file located in the *component directory* and is called *xpti.dat*. This file contains the following information:
   - Location and search paths of all type library files.
   - Mapping of all known interfaces to type libraries where these structures are defined

   Using these two manifests, XPCOM determines the installed component libraries and the implementations to each interface.

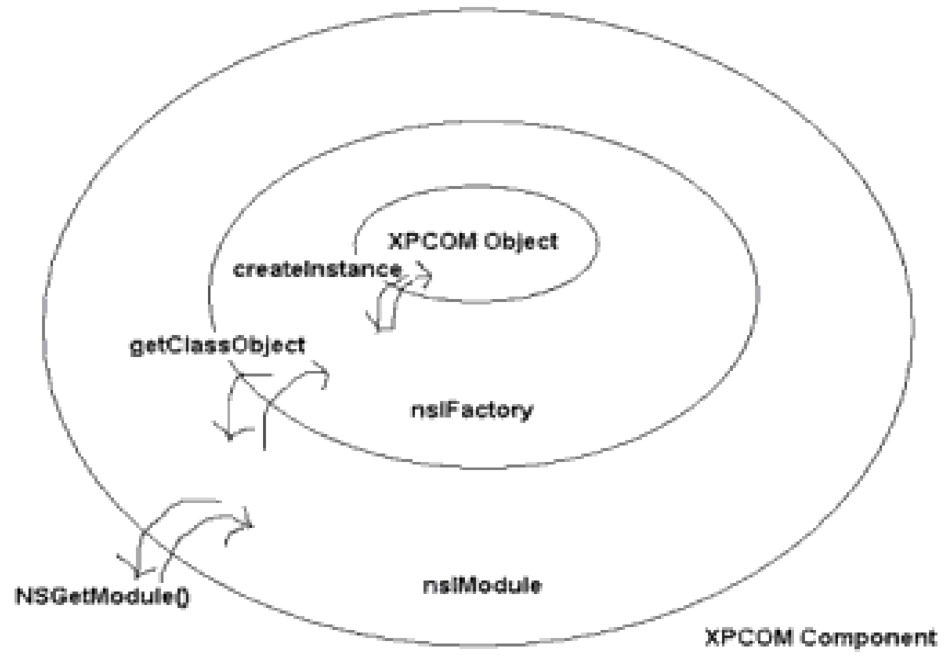


Fig. 7. View of XPCOM Component Creation [2]

6 Applicability of XPCOM on our modelling example

As it is illustrated in figure 7, every XPCOM component has three main parts. From the innermost and moving outward, the first part is the XPCOM object containing the business logic. The next part is the `nsIFactory` object providing a basic abstraction of the XPCOM object enabaling the construction of new instances of an object which matches a given CID and IID. The third part is the `nsIModule`, an abstraction of the `nsIFactory` object. Through the `nsIModule` we can ask for implementations details about the XPCOM object, like for example whether it is singleton or not, or the implementation language.

We will try to explain the creation of an XPCOM component giving explicitely the needed steps to make a component working properly.

All XPCOM components need to be registered before they can be used. A component can be registerd by runnig the application *regxpcom*. Each component library has to inclose implementaions of the interfaces `nsIModule` and `nsIFactory`. The *factory* and *module* could be considered as layers or glue to plug the XPCOM object into the XPCOM system. Before explaining how the coding for the registartion process is done, let's take a look first of all to the required includes and constants which we will need and discuss later. `nsIModule.h` and `nsIFactory.h` are included beacuse they define the module and factory interfaces as well as a couple of important macros. We must also include `nsIComponentManager.h` and `nsIComponentRegistrar.h` providing functions such as `RegistarFactoryLocation` that are required to implement the module and factory classes in our code. Then a series of `nsIID` variables has to be initialized. These variables are destinated for the Module, Factory, Supprts and ComponentRegistrar interfaces. The variables are initialized as classes to hadnle the 128 bit identifiers used by XPCOM to support the contractual relationships between the client and component interfaces. Defined as classes these identifier variavles provide us with methods like `Equals()` that can be used to facilate comparisions in the code.

Now let's try to understand the registration process. By registering a component for the first time (via *XPInstall* or *regxpcom*), XPCOM tries to load the library of the component and find the symbol `NSGetModule` [2]. The XPCOM's *Component Manager* and the location of the shared library where the component lives are then passed to `NSGetModule`. "The *Component Manager* is an interface implemented by XPCOM that encapsulates the creation of objets and provides summary information about all registered components, the location on disk is passed via another interface named `nsIFile`. An `nsIFile` object is usually a file directory on a local volume, but it may represent something on a network volume as well" [2].

By writing a XPCOM component, is the `nsIModule` implementad to do all registration, unregistration, and object creation. `nsIModule` has four methods that must be implemented:

– **Registration Methods:**
  The `RegisterSelf` call allows XPCOM to know exactly what a component

supports. Analog must also the method `UnregisterSelf` be implemented. First of all, the `NSGetModule` entry point is called in the component library. This returns an interface pointer to a `nsIModule` implementation. Then XPCOM calls `RegiterSelf` with the following parameters:

  - `nsIComponentManager`
  - The location of the component being registered as a `nsIFile` object
  - The `aLoadStr` parameter that distinguishes components that are loaded from the same location specified by the `nsIFile` parameter.
  - The `aType` parameter specifies what kind of loader to use on the component

  The two last parameters are usually passes into the `nsIComponentRegistrar` methods, which are then used to tell XPCOM what implementation is in the component library [2]. The method `RegisterFactoryLocation` of `nsIComponentRegistrar` is resposible for that mechanism and is the third method that must be implemented by `nsIModule`.

- **Creating an Instance of the component:**
  After the registration of the component, anyone that uses XPCOM cann access the new classes if they know either the contrat ID or CID [2]. An instance of a component is created by calling the method `CreateInstance` which do need the following parameters:
  - A CID specifying the component the client code is looking for.
  - A parameter for aggregation
  - the interface used to talf to the component
  - The out variable which will contain a valid object

By calling `CreateInstance`, XPCOM looks through all registered components to find a match for the given CID. XPCOM the loads the component library associated with the CID if it is not loaded already. XPCOM then class the function `NSGetModule` on the library. Finally, it calls the method `GetClassObject` on the module. This method is expected to return an `nsIFactory` object for a given CID/IID pair [2]. This method must be implemented also in the component code. Actually we need to create a factory object for each object that we have registered with XPCOM.

The main function that must be implemented in the `nsIFactory` interface is `CreateInstance` [2].

At this point we have created a module that implements most of the generic component functionalities. But when an application starts up, registered components are created and notified via the general purpose observer interface `nsIObserver` [2]. "Observer are objects that are notified when various events occur. Using them is a goog way for objects to pass messages to each other without the objects having explizit knowledge of one another" [2].

If we want that our object registers for an event, it first must implement the `nsIObserver` interface. The observer service implementing `nsIObserverService` can notify our object of registerd events as follows:

When the notification is made the `nsIObserverService` broadcasts the notification from the caller of the `NotifyObserver()` to the `nsIObserver` object's `Observe()` method.

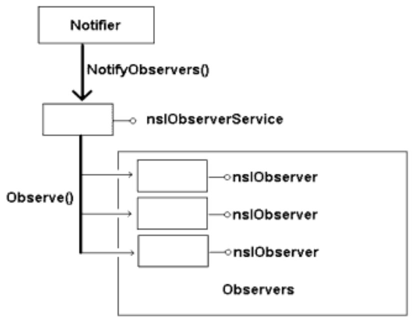


Fig. 8. Observer interfaces [2]

"The `nsIObserver` is a generic interface for passing messages between two or more objects without defining a specific frozen interface, and it is one of the ways in which extensibility is built into XPCOM" [2].

We are then able to model and implement the different components of the WebAudioStore. Each XPCOM component we implement must then provide the `NSGetModule` function, which then represent the communication point of the component with the XPCOM framework. Provided functionalities are then implemented in the XPCOM object that contains the business logic of the component. Required services are then called and gathered by calling `CreateInstance`, which gives access to the functionalities of a component by passing its CID and the interface used to talk to it as input parameters.

7 Utilities of XPCOM

XPCOM reduce the complexity of the component to implement by providing a big set of services. The component administration is provided by the *Component Manager* that encapsulates the creation of objects and provides summary information about all registered components. Moreover files and directories are in XPCOM abstracted and encapsulated by interfaces. The file interfaces are useful to find and manipulate files that are relative to the application. "The *Directory Service* provides directory and file locations in a cross platform uniform way to make this easier" [2]. This service stores the location of various common system locations, such as the directory containing the running process, the user's `HOME` directory, and others [2]. This service can be expanded so that components or applications their own special locations define and store as for example an application plugin directory.

The notification mechanism is an another service provided by XPCOM, allowing different XPCOM components to register for different events or notifications. The notifications are oraganized in categories and managed by the *Category Manager*. A component registers for a notification via the observer service of `nsIObserver`.

XPCOM provide also a functionality to manage and administrate the used memory and avoid memory leaks or crash. This is provided by the interface `nsIMemory`. this interface is used to allocate and deallocate memory and provides for notifications in cases of low-memory situations. The functionalities provided by this interface are defined through its methods: `Alloc, Realloc, Free, HeapMinimize` and `IsLowMemory` [2].

References

1. Parrish, R.: An introduction to XPCOM. http://www−128.ibm.com/developerworks/webservices/library/co-xpcom.html (2001)
2. Turner, D., Oeschger, I.: Creating XPCOM Components. http://www.mozilla.org/projects/xpcom/book/cxc/pdf/cxc.pdf (2003)