Active Components

Semester Thesis

Samuel Willimann
<wsamuel@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Dr. Beat Signer

Global Information Systems Group
Institute for Information Systems
Department of Computer Science

17th July 2006

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

The *Integration Server* (iServer) architecture is an extensible cross-media link server platform enabling links between different types of media [2]. It has been applied within the *PaperWorks* and *Paper++* projects to define links between paper and digital content and vice versa. The philosophy of the iServer architecture is to provide basic link functionality (including user management etc.) which then can be extended to support different kinds of new physical or digital content.

More recently, the concept of *Active Components* has been introduced to support not only links to different media types (e.g. paper, movies, etc.), but also to integrate small pieces of program logic. However, it has not yet been possible to define the structure of these Active Components (such as properties, methods, etc.) in an abstract fashion, and the authoring of Active Components has been limited to raw XML input only.

As part of this semester project, a visual authoring tool has been created, which is supposed to simplify the design and development of Active Components by providing a simple user interface for schema specification and class binding definitions.

Another effort went into enhancing the existing framework [1] to allow Active Components to communicate with each other, essentially through a very simple form of remote method invocation, and to investigate and eliminate other limitations of the framework.

Contents

1 Introduction

1.1 iPaper and iServer

As part of the sixth European framework program, *PaperWorks*[1] is a collaboration of different European partners designated to develop and assess innovative concepts and systems to enrich the use of paper in everyday settings. Being one of these partners, the *Global Information Systems group* (GlobIS)[2] at ETH Zurich has been developing the server technologies responsible for managing the digital media and the links between paper and digital resources. The framework is based on a client-server architecture. A special pattern is printed on sheets of paper, encoding physical coordinates and a unique document number. This information is read by a digital pen, which sends the data to a transformer component on the client device. In most cases an HTTP request is then generated and transmitted to the link management server (iServer) that processes the request and returns the information linked to the specified document area. The *Extensible Information Management Architecture* (XIMA) passes the response back to the client device in the appropriate format.

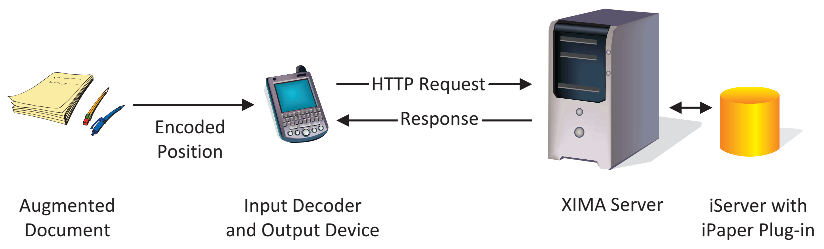


Figure 1.1: Functional overview of interactive paper

[1]www.paper-works.org
[2]www.globis.ethz.ch

1.2 Active Components

In order to understand the motivation of this work, it is necessary to know the basic concept of *Active Components*. Document areas as described above have usually been linked to simple media contents (resources) such as web pages or movie clips. The use of resources was rather limited, because upon every location event from the pen, only one resource could be triggered. The introduction of a new resource type, the *Active Component*, then allowed adding program logic and performing small tasks, such as capturing notes, using text-to-speech functionality, or sending OLE commands to external applications.

1.3 Motivation and Goal

Despite this great advantage, Active Components have been subjected to some restrictions however. The goal of this thesis is to investigate and eliminate some of these restrictions and to support the user in designing custom Active Components more easily. Therefore, a simple authoring tool is introduced in Chapter 4.

1.3.1 Active Component Intercommunication

Sometimes it would be an advantage if Active Components could exchange messages or if one component could use another component's functionality. The idea of Active Component intercommunication is described in Section 2.3.1. In order to demonstrate the benefit of Active Component intercommunication, a small example application is presented in Section 3.4.

1.3.2 Active Component Properties

In the existing framework, properties of Active Components are represented internally as pairs of strings $p = (id, v)$, where id denotes the property's identifier and v its value. In order to simplify the use of other property types, i.e. integers, booleans or even objects, the mechanism of storing and retrieving data from or to the database has been adapted as described in Section 3.3.1.

1.3.3 Authoring Support

The Active Component framework has become quite a complex system. Managing all the different files and settings turns out to be cumbersome at times. As a first step to reduce this complexity, our authoring tool supports the user in several ways:

- Definition of Active Components (schemas), i.e. their public properties and methods
- Definition of Active Component bindings, i.e. provide a Java representation (stub and logic) for a given identifier

2 Design

To discuss the decisions made in the design of the new Active Component framework, we first look at some existing infrastructures coping with distributed systems. We are then going to investigate which concepts are really used in the Active Component framework and where its design diverts from existing solutions.
In the following three sections, a quick description of some of the most popular solutions of distributed computing frameworks, Java RMI, CORBA, and Microsoft DCOM, is given.

2.1 Existing Concepts of Distributed Computing

2.1.1 Java RMI

Remote Method Invocation (RMI) [7, 8] is a Java application programming interface for performing *remote procedure calls* (RPC). It is not a new concept though, because even C programmers have been using RPC semantics to execute a function on a remote host. What makes RMI different is that in Java it is necessary to package both data and methods and ship both across the network (RPC works on data structures primarily), and the recipient must also be able to interpret the object after receiving it.
In order to accomplish this, client and server must both know the exact interface of the remote object, which, of course, has to be the same on both sides. Hence, a client can only call a method of a remote object (*adaptor*) which implements the same interface as its corresponding stub (*proxy*) on the client side.
The correct methods are looked up and bound by the RMI Registry. A sketch of the RMI mechanism is depicted in Figure 2.1. RMI allows entire objects to be passed and returned as parameters, unlike many RPC-based mechanisms which require parameters to be either primitive data types, or structures composed of primitive data types. That means that any Java object can be passed as a parameter—even new objects whose class has never been encountered before by the remote virtual machine.

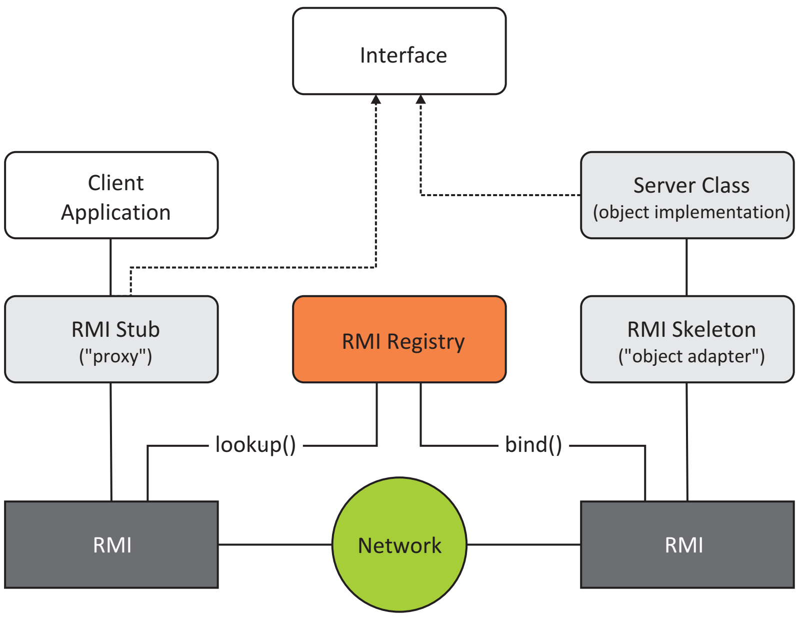


Figure 2.1: Java RMI

However, RMI is strongly tied to the Java language, and interoperability between Java programs and other legacy systems often involves developing an appropriate interface.

2.1.2 CORBA

The *Common Object Request Broker Architecture* (CORBA) [6, 7, 8] defines APIs, communication protocols, and object/service information models to enable heterogeneous applications written in various languages running on various platforms to interoperate. CORBA therefore provides platform and location transparency for sharing well-defined objects across a distributed computing platform.

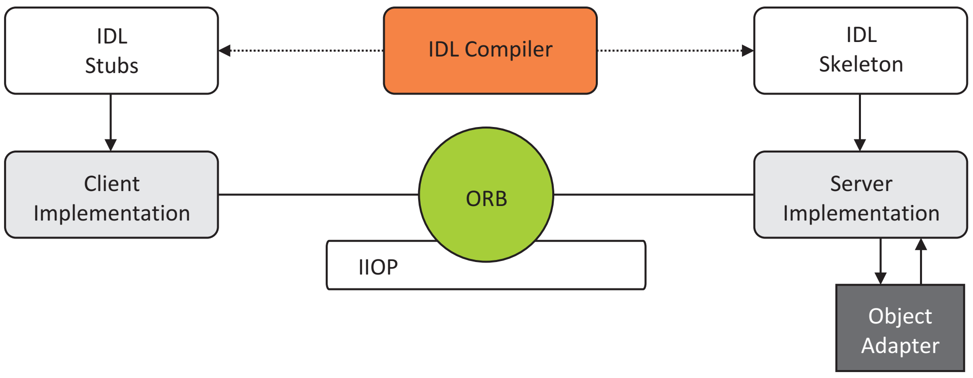


Figure 2.2: The CORBA model

In a general sense CORBA "wraps" code written in some language into a bundle containing additional information on the capabilities of the code inside, and how to call it. The resulting wrapped objects can then be called from other programs over the network.

CORBA uses an interface definition language (IDL) to specify the interfaces that objects will present to the world. CORBA then specifies a “mapping” from IDL to a specific implementation language like C++ or Java. This mapping precisely describes how the CORBA data types are to be used in both client and server implementations.

2.1.3 Microsoft DCOM

The *Distributed Component Object Model* (DCOM, [4, 5, 6, 8]) is Microsoft’s solution for supporting distributed computing with objects and is an extension of the *Component Object Model* (COM). DCOM was a major competitor to CORBA, but has been deprecated in favor of Microsoft .NET, which will not be discussed further here.

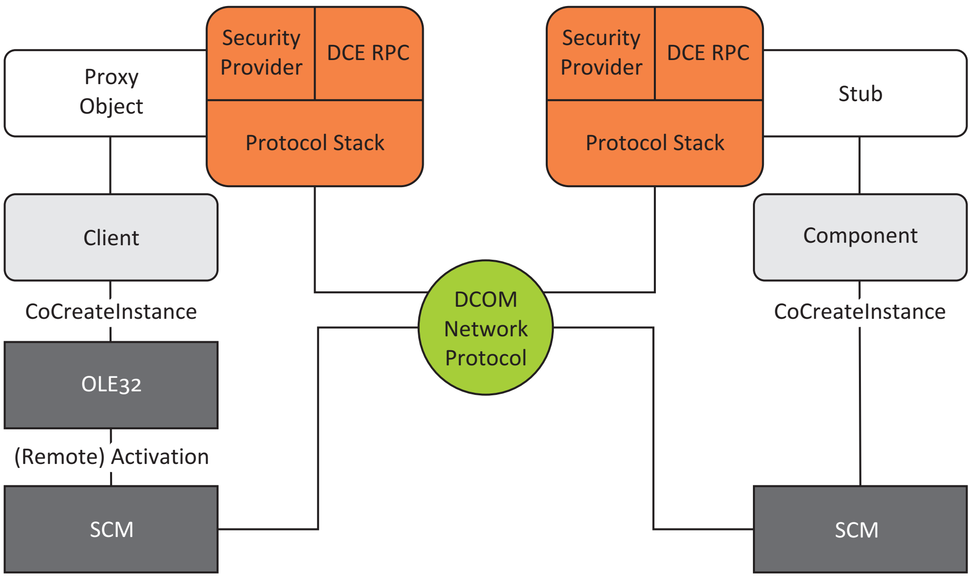


Figure 2.3: Microsoft DCOM

The COM libraries look up the appropriate binary (dynamic-link library or executable) in the system registry, create the object, and return an interface pointer to the caller.
For DCOM, the object creation mechanism in the COM libraries is enhanced to allow object creation on other machines. In order to be able to create a remote object, the COM libraries need to know the network name of the server. Once the server name and the Class Identifier (CLSID) are known, a portion of the COM libraries called the service control manager (SCM) on the client machine connects to the SCM on the server machine and requests creation of this object. Like CORBA, DCOM is language independent.

2.2 The Existing Active Component Framework

Before we get into describing the design of the new Active Component framework and its improvements, we give a quick overview of the existing architecture. The following diagram depicts the current Active Component framework.

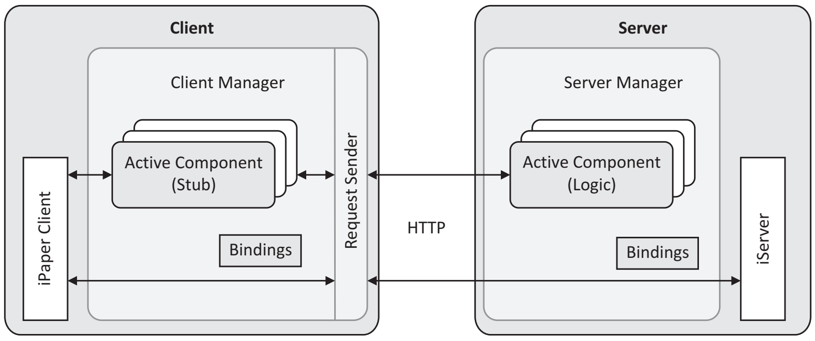


Figure 2.4: The previous Active Component framework

Upon every event from the pen, an HTTP message containing the corresponding location coordinates is sent to iServer, which then searches for the resource linked to this location. If the resource is an Active Component, its logic is initiated on the server side and its properties are loaded from the database. The new logic subsequently sends an XML representation of itself (i.e. its properties) back to the client which instantiates the matching component stub with these properties.
The same unique identifier is used on both sides to identify an Active Component. The corresponding class names are resolved by the *Active Component Resolver* which reads the bindings from an XML file. Bindings are tuples $b = (id, logic, stub)$, where

id	unique identifier of the Active Component
$logic$	complete package and class name of the component's logic
$stub$	complete package and class name of the component's stub

Both, stub and logic, will only stay "alive" as long as it takes to handle the request. In other words, those objects are created literally with every click. Both classes implement a specific method which is executed as soon as they have been initialized. In most cases, the objects are instantly terminated after the execution of this method.

2.2.1 Limitations

The following list contains some of the limitations of the existing framework that we tried to solve in this project. The subsequent section describes in detail how the various problems were approached and how the new enhanced system works.

- Active Components are isolated. They cannot communicate with each other.
- Properties of Active Components are only provided as pairs of strings and have to be loaded and parsed manually by the developer.

- Properties (and Active Components themselves) cannot be made persistent; thus Active Components are stateless.
- Although iPaper is an important part of the framework, the Active Components are too tightly connected with the iPaper framework. In this respect, the design is not generic enough.
- Creating a new Active Component involves small modifications in several files distributed all over the system. There is no tool that takes care of these tedious but necessary modifications automatically.

2.3 The Extended Active Component Framework

Essentially, the overall design of the framework has not changed much. We tried to keep as much functionality from the original framework as possible. Although the modifications are subtle, they significantly reduce the amount of necessary code in both new and existing Active Components.
In order to implement those modifications, the existing framework has been extended, the class hierarchy restructured, and new types introduced. The packages and classes of the new framework are described in full detail in Chapter 3.

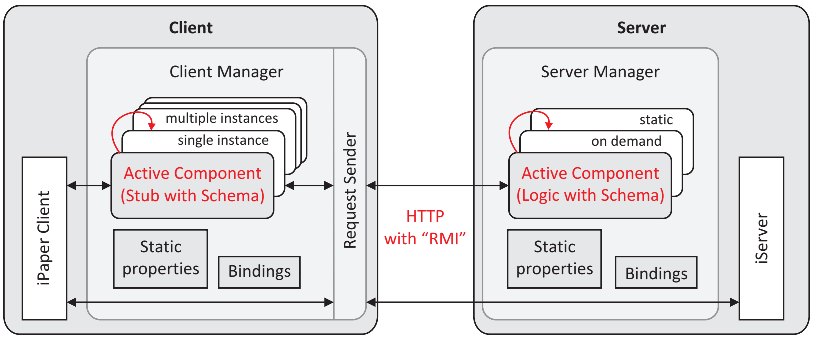


Figure 2.5: The extended Active Component framework

We are now going to take a closer look at the new features, what benefits they can offer and how they exactly work.

2.3.1 Remote Method Invocation

Similarly to Java's RMI, we wanted to allow Active Components to interact with other components through a simple mechanism of remote method invocation. As explained earlier, we were looking for a much simpler approach than the one provided by RMI though. Basically, we only wanted to allow Active Components to exchange simple data such as integers or strings, or to trigger (idempotent) methods. The terminology of *remote method invocation* may be misleading, because it has got nothing to do with Java RMI. But the fact remains that it actually *is* a remote method invocation mechanism, initiated by Active Components, and delegated by the Active Component managers.

Any Active Component can call a method of any other component, if it knows the name of the method as well as the number and types of the expected arguments. Due to the distributed architecture of the framework, Active Component stubs cannot directly access methods of Active Component logics (or even the objects themselves), and vice versa. The delegation of method invocations is therefore handled by the Active Component manager objects.
Usually, Active Components have a very short lifecycle. In order to be able to invoke remote methods, it must be possible to load Active Components dynamically (in case they have not been instantiated yet), and to prevent them from being destroyed immediately after the action event. While the Active Component manager takes care of the dynamic component loading, the user can specify whether the component should be destroyed directly in the class implementation. This approach allows a component to dynamically determine whether it needs to remain active, or whether it can be disposed of.

2.3.2 Persistent logics with a static state

The introduction of persistent logics is a direct consequence of the remote method invocation mechanism described above. Methods can only be invoked on Active Components which are currently loaded, which makes it necessary to load them dynamically if they are not already running.
Furthermore, persistent components can also be used as "aggregators" which collect data from other components. Imagine an interactive reservation form where the user can choose between different options. Only if the user has specified all required data and selects the OK button the request will be processed.

2.3.3 Global properties

If the developer does not intend to make an Active Component logic persistent, but still wants to allow different components to share common values, global properties can be used instead. Global properties can be set and read by any Active Component. They will be available as long as the system is running, unless they are explicitely removed by an Active Component.
Global properties are a very quick and simple way to share data among Active Components, but there are also some limitations. For instance, Active Component stubs can only access properties written by another stub, and Active Component logics can only access properties written by other logics. The properties are not shared across the network.
Furthermore, properties are not protected by access rights, but can be read by any other component, and they remain in memory as long as they are not explicitely removed. But these restrictions could be eliminated in the future by introducing the notion of a *lease* (where properties are deleted automatically if they are not used for a certain amount of time), as well as assigning owners to the properties.

2.3.4 Generic Action Events instead of Pen Events

The previous version of the Active Component framework was very much focussed on iPaper-related components. iPaper is unquestionably the most important part of the framework (iPaper contains three times as many classes as iServer itself), but the existing implementation was not generic enough in terms of new application areas for Active Components. Most components explicitely asked for a `TimestampedLocation` (a specific type of pen events) and a `BufferedInputReader` (for input devices such as a digital pen or a mouse) to perform their tasks. We therefore introduced a generic class `ActionEvent`, from which future events should inherit. The generic action event allows the developer to design components that are not directly related with iPaper but still want to make use of action events.

2.3.5 Multiple stubs running at the same time

Sometimes, Active Components need more time to perform their tasks than the user is willing to wait. Instead of one single stub handling all action events, an idle stub from a stub pool could be selected. The stub manager keeps a list of all Active Component stubs of the same type and takes care of the stub selection.

2.3.6 Simplified design process with authoring tools

In order to put a new Active Component into operation, it is not sufficient to just write a new class. Instead, several files have to be coordinated properly. Above all, the Active Component schema has to be defined and the bindings have to be entered into the global binding table. It gets even worse if components are renamed or deleted, which can lead to unused files and obsolete links.
Since the authoring tool is an essential part of this project, a comprehensive documentation can be found in Chapter 4.

3 Implementation

In Chapter 2 we have seen the new concepts of the extended Active Component framework. In this chapter we are going to take a closer look at how they have actually been implemented. A selection of books and websites was used as primary Java programming references [9, 10, 11].

3.1 Package Structure

The following packages have been either added or extensively modified during the development process:

1. `org.ximtec.iserver.activecomponent`
2. `org.ximtec.iserver.activecomponent.schema`
3. `org.ximtec.iserver.activecomponent.event`
4. `org.ximtec.iserver.authoring.*`

(1) contains abstract classes for Active Components and Active Component managers, as well as concrete base classes for Active Component stubs/logics and their corresponding managers. It also contains the identifier resolver and the binding manager. (2) contains the schema for Active Components as well as three classes used in schemas: a method representation, a property representation, and a type representation (for types like integer, string, etc.), which are all described in the next section. (3) contains a generic base type for action events, which replaces the old iPaper-specific *PenEvent* and provides a more general approach for event handling. Finally, (4) and all its subpackages contain classes of the authoring tool described in Chapter 4, including all the GUI classes.

3.2 The Classes

Although the class hierarchy (Figure 3.1) did not change notably, the implementation of some classes has been completely altered and will be explained in detail in the following sections.

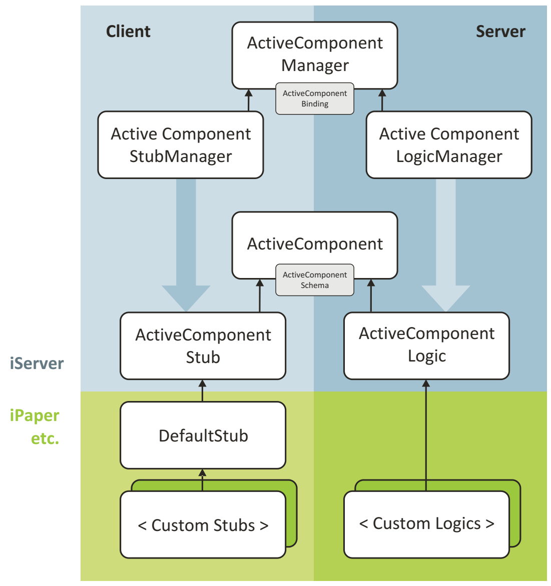


Figure 3.1: Class hierarchy

3.2.1 ActiveComponent

`ActiveComponent` is an abstract class providing basic functionality for Active Components, such as accessing its own name, identifier, or parameters. The two private methods `setProperty` and `getProperty` are responsible for getting and setting the component's properties dynamically. They are internally used to load and store the properties and do not have to be called explicitely by the developer. The class also provides a simple timeout mechanism which is used by the managers (see below) to unload a component if it takes longer than expected to perform its actual task.
We implementd all important Active Component behavior directly in this abstract class. This has the advantage that developers of new Active Components only need to inherit from this base class in order to create a fully functionable Active Component, and almost no additional code is required. There are only two `ActiveComponent` subclasses, namely `ActiveComponentLogic` and `ActiveComponentStub`:

3.2.2 ActiveComponentLogic

`ActiveComponentLogic` represents an Active Component on the server side. Only `ActiveComponentLogic` objects have access to the iServer database (and hence can read or write properties). The core method of an `ActiveComponentLogic` is called `handleActionRequest` and takes an `HttpServletRequest` as an argument. This method can be overridden by subclasses in order to implement the server-side functionality of an Active Component.

The logic also has two other functions, `init` and `finish`, which are invoked right after a new Active Component is initialized or before it is terminated, respectively.

3.2.3 ActiveComponentStub

`ActiveComponentStub` represents an Active Component on the client side. It can be used if the client device needs to display information to the user, e.g. rendering a webpage or providing audio feedback etc., or if other local computations have to be performed. Active Component stubs are only created upon a request by the server. Therefore, the Active Component logic has to send all its shared properties to the client. For this purpose, the properties are transformed into an XML representation using JDOM [11]. The stub reads the values from this message and uses the method `setProperty` to initialize its fields accordingly.
The stub's event handler, `handleEvent`, is more generic than the logic's event handler. In general, it is called exactly once (usually by the client application). It takes an `ActionEvent` argument, which can be extended to handle almost every event type imaginable. In the specific case of iPaper applications, the specialized subclass `TimestampedLocationEvent` contains iPaper-specific event arguments such as a `TimestampedLocation` (essentially a page ID and the coordinates selected by the user) and an `BufferedInputDevice` (e.g. the digital pen, the mouse etc.).

3.2.4 ActiveComponentManager

In order to manage Active Components at runtime an `ActiveComponentManager` is used on the client as well as on the server side. These managers are responsible for creating and maintaining Active Components and for managing their lifecycle. The abstract class `ActiveComponentManager` again has only two subclasses, namely `ActiveComponentLogicManager`, which handles logic objects on the server, and `ActiveComponentStubManager`, which handles stubs on the client. The former needs an *OM Instance* to create a logic object, the latter only the corresponding key-value-tuples of the logic's public properties.
The managers are also part of the remote method invocation mechanism. As described earlier, they delegate method invocations from Active Components to other Active Components, or to their counterparts on the client or the server side respectively. Therefore, managers provide two different versions of the method invocation routine:

3.2.5 ActiveComponentBinding

The framework does not know which logic actually belongs to which stub. In fact, any logic could be used with different stubs and vice versa. Thus, each stub has to be explicitely bound to a specific logic by the developer and every binding is tagged with a unique identifier (which will be referred to as the *Active Component identifier*). Using a globally available list that contains all the bindings for a specific application, the `ActiveComponentResolver` can then map any given identifier to the corresponding logic or stub class name. Bindings are stored in a simple XML file format. In order to edit the list of bindings efficiently, the *Active Component Binding Editor* (see Section 4.3.1) should be used.

`invokeStubMethod(...)`	Invokes a method on an Active Component stub, i.e. on the client side. If the call is made by another Active Component stub, it will directly be redirected to the corresponding object. If the call is made by an Active Component logic however, the manager will send the request to its counterpart located on the server, which will then take care of the method call. This remote request (as well as the method's return value) will be transmitted in a simple XML representation (see Appendix B.3) via HTTP.
`invokeLogicMethod(...)`	Works exactly like `invokeStubMethod(...)`, but stub/logic and client/server are interchanged.

3.2.6 ActiveComponentSchema

`ActiveComponentSchema` is a new class of the Active Component framework. Schemas describe the interface of Active Components, i.e. their public properties, methods and the respective types of those entities. When logics send their XML representation to the client, only the properties defined in the schema are transmitted, independently of their actual modifier. This enables the developer to precisely define which properties are to be shared between stub und logic, and which ones are only used locally.
The different entities (properties and methods) are stored in a straightforward way: Both properties and methods are stored in two separate collections:
Each property is represented as a tuple $f = (id, t, v, r, w)$, where

id	field identifier (`java.lang.String`)
t	type (`org.ximtec.iserver.activecomponent.schema.Type`)
v	standard value, if no value is specified in the document definition file
r	*readable*: property can be read by a public getter method
w	*modifiable*: property can be set by a public setter method

Each method is represented as a tuple $m = (id, t, a)$, where

id	method identifier (`java.lang.String`)
t	return type (`org.ximtec.iserver.activecomponent.schema.Type`)
a	method arguments

Finally, arguments are represented as a list of tuples $a = (id, t)$, where again, id is the identifier and t the type of the argument. Strictly speaking, id is not required to perform a remote method invocation, but the method identifier is used by the Binding Editor. The detailed XML Schema of the schema file format is given in Listing B.2.

3.2.7 Schema Types

Together with schemas, special data types have been introduced as well. The goal was to strictly separate types used in schemas from native Java types in order to allow interoperability with other programming languages. Table 3.1 shows all types currently defined in the `Type` class.

Schema Type	Java Type	Contents
ac:string	`java.lang.String`	string value
ac:integer	`int`	integer number
ac:double	`double`	floating point number
ac:boolean	`boolean`	boolean value
ac:date	`java.util.Date`	date and time
ac:object	`java.lang.Object`	any object or binary data

Table 3.1: Types defined in `org.ximtec.iserver.activecomponent.schema.Type`

3.3 Improvements and Extensions

3.3.1 Internal Property Handling

An Active Component's properties are stored on the server as pairs of strings. In the existing framework, it was necessary to parse them manually. This was cumbersome and error prone. The new framework handles the parsing of properties automatically. Developers do not have to care about type conversion anymore—instead, the properties can directly be accessed like normal fields.

Property parsing is done during the instantiation of an Active Component. The properties are available as soon as the Active Component is loaded by one of the Active Component managers. Depending on whether an Active Component logic or a stub is initiated, the properties are read from an OMS resource on the server or parsed from an XML representation of name-value-pairs sent by the server respectively. In both cases, the Active Component itself will be looking for an appropriate class field definition using Java reflection. If a field is indeed declared, its appropriate value will be parsed. In order to determine the corresponding type of a property, the Active Component schema (which has already been loaded at this point) is consulted.

3.3.2 Static Logics

As explained in Section 2.3.2, static logics can be very handy. To make a logic static, we just need to prevent a component from being destroyed after the action event.

The old framework provides a method `setDone` (a member of the *ActiveComponent* class) to mark an Active Component as finished and to clean up internal data structures. We overloaded this method in order to provide more flexibility.

```java
public void isDone(boolean keepInMemory) { ... }
```

If this method is invoked with the argument `true`, the component will not be destroyed, but kept in memory for later use.

3.3.3 Multiple Stubs

The stub manager can distinguish between stubs with different types by comparing their identifiers, but in order to make it capable of distinguishing between different stubs of the same type, we need to uniquely identify every single stub. In our implementation, this is done with a simple integer value, which is incremented with every new stub. However, this is not the best solution, mainly because stubs can be destroyed at any time, and the index may not be valid any more. A better solution would involve a globally unique identifier (GUID) associated with every active stub. This identifier is created when a stub is instantiated, and the user can query the manager for a specific stub by passing its unique identifier.

3.3.4 Generic Action Events

As stated in Section 2.2.1, Active Components have been connected too tightly with the iPaper framework. As it is shown in Listing 3.1, the method name suggests a direct connection to pen events (which, of course, had actually been the case), and the method arguments were iPaper-specific. It simply was not possible to call the event handler from a different device other than a digital pen.

Listing 3.1: Existing signature of the pen event handler

```
public void handlePenEvent(TimestampedLocation location,
                           BufferedInputDevice reader);
```

The new method signature is significantly different, but it is still working in exactly the same way. The method name was changed, and the arguments were moved to a new class `TimestampedActionEvent` located in `org.ximtec.iserver.activecomponent.event`. It is a subclass of the more general type `ActionEvent` located in `org.ximtec.iserver.activecomponent.event`, which now serves as a base class for all future action events which require more specific parameters.

Listing 3.2: New signature of the general event handler

```
public void handleEvent(ActionEvent e);
```

3.4 The Demo Application

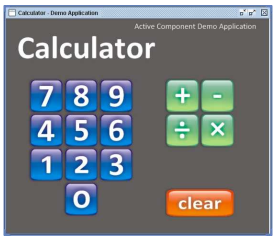

Calculator - Demo Application
Active Component Demo Application
Calculator
7
8
9
4
5
6
1
2
3
0
+
-
÷
×
clear

Figure 3.2: The demo application

To demonstrate the new capabilities of the Active Component framework, we consider a simple *Calculator* application shown in Figure 3.2. It consists of only four different Active Components which are described in detail in Table 3.2.

Active Component	Description	Attributes
CALC_NUMBERS	Two-dimensional slider control representing the digits 0 to 9 (two cells are not used).	default
CALC_OPERATIONS	Two-dimensional slider control representing four basic arithmetic operations.	static
CALC_CLEAR	Default button which will reset the calculator control.	default
LOGGER	Logs all calculations. If the user clicks on the button "Log" (not present in the screenshot), the log will be read out.	static, hidden

Table 3.2: The demo application components

Clicking on a number will generate a remote method invocation, and the selected number will be sent to the logic of the CALC_OPERATIONS control. The CALC_OPERATIONS control waits until an operation is complete (i.e. the user has selected a number fikkiwed by an operation and terminated by a second number). The CALC_OPERATIONS logic invokes a remote method by sending the result to the TTS engine on the client side. At the same time, the operation will be appended to the server-side log.
This application demonstrates the Remote Method Invocation in both directions, as well as static logics and static properties.

3.5 Conclusion

It is virtually impossible to describe all modifications in detail, because often, only a few lines of code were modified. As stated at the beginning, we wanted to preserve the existing functionality, and add some new features. Nevertheless, even well-established base classes such as the `ActiveComponent` itself underwent some changes, and although this did not affect the classes' general behavior, it certainly facilitated their use. Let us look at the source code of a simple Active Component to see the differences between the existing and the new framework more clearly.
We do not comment this example (except for comments in the source code), because the differences between the two versions are obvious, and a detailed description has already been given in this chapter. Nevertheless, it sums up most of our modifications and shows how these features are actually used in a real Active Component.

3.5.1 Source Code Comparison

Listing 3.3: An Active Component stub written in the existing framework

```
package org.ximtec.ipaper.newframework.stub;

import /* ... */
```

```
public class SampleStub extends DefaultStub {

 private static final String ADVICE = "Custom␣User␣Advice.";

 public String getUserAdvice() {
  // Return user advice
  return ADVICE;
 }

 public void init(String servletPath, TimestampedLocation location,
                  BufferedReader reader) {
  // Initialize properties...
 }

 public void handlePenEvent(TimestampedLocation loc,
                            BufferedReader reader) {

  // Note the inflexible, iPaper-specifc method signature

  // Properties are stored in a hash map (as pairs of strings)
  Rectangle rect = new Rectangle(
     (double)getParameter("x"),
     (double)getParameter("y"),
     (double)getParameter("width"),
     (double)getParameter("height"));

  doSomething(rect); // ...

  // Terminate this stub
  setDone();
 }

 public void finish() {
  // Clean up and perform last-minute actions
  // (restricted to the server side)
 }
```

Listing 3.4: An Active Component stub written in the new framework

```
package org.ximtec.ipaper.newframework.stub;

import /* ... */

public class SampleStub extends DefaultStub {

 private static final String ADVICE = "Custom␣User␣Advice.";

 // The component's properties
 protected double x, y, width, height;

 public String getUserAdvice() {
  // Return user advice
  return ADVICE;
 }

 public void handleEvent(ActionEvent e) {
  super.handleEvent(e);
```

```
    // Handle the ActionEvent
    Rectangle rect = new Rectangle(x, y, width, height);
    doSomething(rect); // ...

    // Terminate this stub
    setDone();
  }

  public void finish() {
    // Clean up and perform last-minute actions,
    // e.g. send some text to the TTS engine (on the client)

    // Prepare the method call argument
    Property[] args = new Property[] {
      new Property("textToSpeech", Type.STRING, ADVICE)
    };

    // Invoke the remote method using this argument
    ActiveComponentStubManager.getInstance().
      invokeStubMethod("TTSEngineStub", "speak", args);
  }
}
```

4 Authoring

In the early stage of this project, the authoring application consisted of several independent tools. Very soon it became clear that it was actually more sensible to integrate the tools into a single editor—not only to speed up the development process, but also in anticipation of integrating this application into the existing authoring tool for the Active Component framework. The authoring tool developed during this semester project is merely the beginning of a much more comprehensive authoring system. We will discuss this in detail in Chapter 5.1.

4.1 Documentation

In this section, we are going to explain some basic programming concepts used in the Active Component framework. This should help future developers to become acquainted with the programming practices used in the framework. We will also explain how the authoring tool can be integrated into other applications.

4.1.1 A framework of singletons

The Active Component framework is designed in a way that it is sensible to only have exactly one instance of the `ActiveComponentStubManager` on the client side and, correspondingly, exactly one instance of the `ActivComponentLogicManager` on the server side. Therefore, both classes are provided as singleton objects, and have to be accessed as follows:

```
// Import the ActiveComponent classes
import org.ximtec.iserver.activecomponent.ActiveComponentStubManager;

// getInstance() will return a manager singleton
ActiveComponentStubManager.getInstance(). ...
```

4.1.2 Active Components are managed

Active Components are managed by their corresponding manager. This means that Active Components should be neither accessed directly, nor instantiated manually. Instead, the mechanisms described in Section 2.3 should be used (i.e. remote method invocation or static properties).
If a reference to an Active Component object is really required, `getActiveComponent(...)` can be called, which will return exactly this—assuming a component with the given identifier is currently running. Otherwise, it will return `null`.

4.1.3 Location of Schemas and Bindings

Schema file names are composed of the Active Component's full identifier, followed by the file extension ".acschema", and the files are usually stored in a folder named *schemas*, located in the *resources* directory of the client application, or *Tomcat* respectively. Bindings are always stored in a file named "ActiveComponents.xml" in the aforementioned *resources* directory.
However, the current locations might not be appropriate if the framework is executed on systems with different configurations, and although this is not the case at the moment, the user should at least be allowed to adapt these directories in future versions of the framework. Actually, this applies to all configuration and definition files. A better file management could be introduced together with the suggestions mentioned in Section 5.1.5.

4.1.4 Running the Authoring Tool

Binding Editor

The binding editor can be started either by executing `org.ximtec.iserver.authoring.Authoring` with the command line argument `binding_edit`, or by direct instantiation as shown below:

```
import org.ximtec.iserver.authoring.gui.BindingListEditor;

(new BindingListEditor()).setVisible(true);
```

Schema Editor

The schema editor can be started similarly to the binding editor, but three arguments are required to instantiate the object successfully:

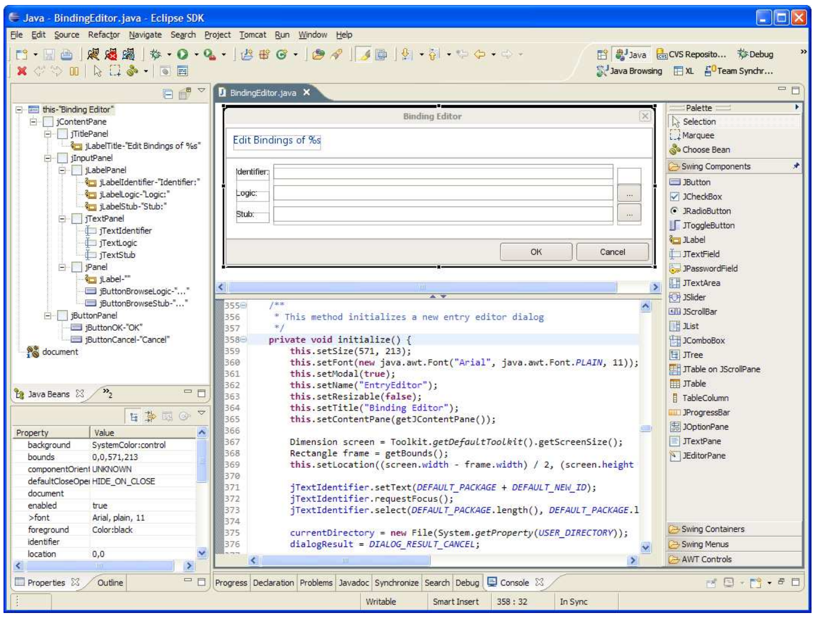
```
import org.ximtec.iserver.authoring.gui.SchemaEditor;

(new SchemaEditor(arg0, arg1, arg2)).setVisible(true);
```

Argument `arg0` contains the schema file name (with relative or absolute path) where the Active Component schema is stored, `arg1` contains the corresponding binding data (which is not directly available in the schema file), and `arg2` contains the list of all Active Component bindings (as it is used for instance in the binding editor). The third argument could actually be omitted, but at the moment, it is needed by the binding editor as a "callback object" in order to update its internal list entries. This could certainly be simplified in the future.

4.1.5 Visual Editor

To facilitate the development of a graphical user interface in Eclipse, the *Eclipse Visual Editor* was used to implement windows and dialog boxes based on the *Swing* library. Although the Visual Editor tends to create a lot of code even for small windows, it has proven to be a valuable tool for creating the user interface of the authoring tool. However, the source code is very likely to be refactored during the integration of the GUI into the new *iServer* system to make it more concise and independent of any third-party visual editor.



Property	Value
background	SystemColor:control
bounds	0,0,571,213
componentOrient	UNKNOWN
defaultCloseOpe	HIDE_ON_CLOSE
document	
enabled	true
>font	Arial, plain, 11
foreground	Color:black
identifier	
location	0,0

```
    /**
     * This method initializes a new entry editor dialog
     */
    private void initialize() {
        this.setSize(571, 213);
        this.setFont(new java.awt.Font("Arial", java.awt.Font.PLAIN, 11));
        this.setModal(true);
        this.setName("EntryEditor");
        this.setResizable(false);
        this.setTitle("Binding Editor");
        this.setContentPane(getJContentPane());

        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        Rectangle frame = getBounds();
        this.setLocation((screen.width - frame.width) / 2, (screen.height

        jTextIdentifier.setText(DEFAULT_PACKAGE + DEFAULT_NEW_ID);
        jTextIdentifier.requestFocus();
        jTextIdentifier.select(DEFAULT_PACKAGE.length(), DEFAULT_PACKAGE.l

        currentDirectory = new File(System.getProperty(USER_DIRECTORY));
        dialogResult = DIALOG_RESULT_CANCEL;
```

Figure 4.1: The Eclipse Visual Editor

4.2 Design of the Authoring Tool

The authoring tool developed during this semester project covers two aspects of the design of new Active Components. On the one hand, Active Component bindings can be managed and on the other hand Active Component schemas can be defined without writing a single line of code.

4.2.1 The Binding Editor

The main window class of the binding editor is called *BindingListEditor* and it is located in the following package:

```
org.ximtec.iserver.authoring.gui
```

The binding editor does not provide any command line arguments. All the settings are stored in the local file `authoring.properties`. Currently, only three values are used by the binding editor:

Property	**Description**
localDirectory	local base directory of the client application
remoteDirectory	remote base directory of the server application
bindingDirectory	base directory where the list of bindings is located

The former two directories are used to store the local and the remote copy of the schema file respectively. The binding directory defines the location where the binding list is placed. The default values will work fine if both client and server run on the same machine, but they need to be adjusted if the program is executed in a "real" distributed environment.
If the property file does not exist (e.g. if the editor is executed for the first time), a dialog will be displayed and the user is asked to confirm the default settings. This dialog is also accessible from the binding editor's menu for later changes.
The binding editor loads the list of Active Component bindings from an XML file called `ActiveComponents.xml` (for the XML schema see Listing B.1). The parsed entries are stored in *ActiveComponentsDocument*, which extends the *AbstractTableModel* class. This makes it very easy to add, remove and edit entries, to sort them and to display the data in a standard *JTable* component.
The binding information for every Active Component can be entered using the *BindingEditor* class, which provides three simple text fields to specify the binding attributes (identifier, stub, logic). All text fields are checked against some regular expressions to prevent the user from entering invalid data (e.g. all values should start with a letter and must not contain spaces). A simplified auto-completion looks for similar package names as the user is typing and automatically inserts the appropriate text. It is also possible to directly select an existing class file, from which the correct package and class name is extracted by reflection. At the moment, this feature is rarely used, because the class files are usually generated *after* the binding entry. However, this could change in the future (see Section 5.1.5).

4.2.2 The Schema Editor

The schema editor could actually be executed as a stand-alone application, but because the schemas are dependent on the binding information and vice versa, is has to be instantiated with the following three parameters:

Parameter	**Description**
String	complete path and file name of the schema file
ActiveComponentBinding	the corresponding binding $b = (id, stub, logic)$
ActiveComponentsDocument	the complete binding list

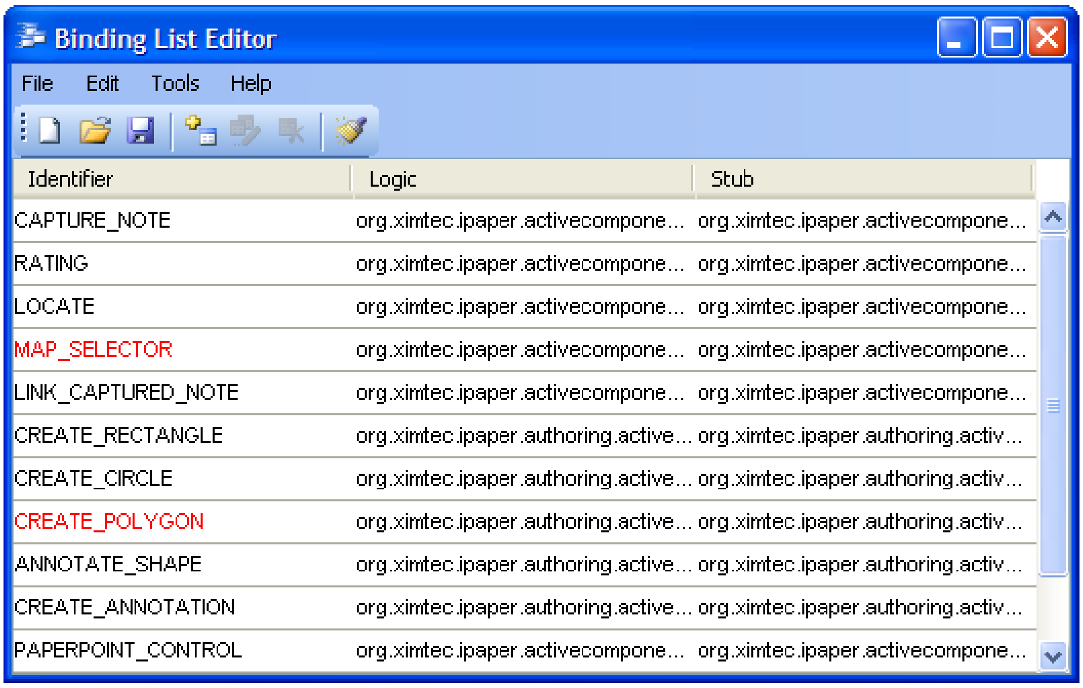
The schema editor displays the interface of an Active Component, consisting of public properties and methods. Two additional dialogs are used to define them. Like the binding information, an Active Component schema is also stored in an XML file (one per Active Component, for the XML schema see Listing B.2). However, during runtime, the schema is represented by the class *ActiveComponentSchema*. This class is also responsible for generating Java source code (`getStubJavaCode()` and `getLogicJavaCode()`).

4.3 Authoring Tool Manual

This section explains how the authoring tool is used. Although the interface is straightforward, some of the input fields need some further explanation.

4.3.1 Maintaining Bindings

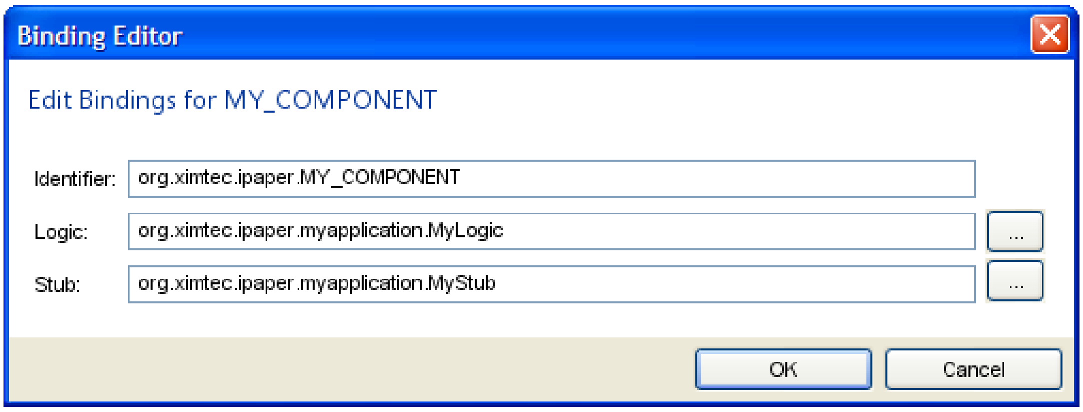
In order to start the *Active Component Binding Editor*, the class `Authoring` from the package `org.ximtec.iserver.authoring` has to be executed with the argument `binding_edit`. The Binding Editor is loaded and a list of all available bindings is displayed.



Identifier	Logic	Stub
CAPTURE_NOTE	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...
RATING	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...
LOCATE	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...
MAP_SELECTOR	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...
LINK_CAPTURED_NOTE	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...
CREATE_RECTANGLE	org.ximtec.ipaper.authoring.active...	org.ximtec.ipaper.authoring.activ...
CREATE_CIRCLE	org.ximtec.ipaper.authoring.active...	org.ximtec.ipaper.authoring.activ...
CREATE_POLYGON	org.ximtec.ipaper.authoring.active...	org.ximtec.ipaper.authoring.activ...
ANNOTATE_SHAPE	org.ximtec.ipaper.authoring.active...	org.ximtec.ipaper.authoring.activ...
CREATE_ANNOTATION	org.ximtec.ipaper.authoring.active...	org.ximtec.ipaper.authoring.activ...
PAPERPOINT_CONTROL	org.ximtec.ipaper.activecompone...	org.ximtec.ipaper.activecompone...

Figure 4.2: The Active Component Binding Editor

In case no components have been defined yet, the list will be empty. New entries can be created at any time by simply clicking the ADD ENTRY button in the toolbar or selecting ADD ENTRY from the EDIT menu.
In the following window, three different properties can be defined. Make sure you enter framework-compliant values (for details see Table 4.1).
Note that only alphanumeric characters are allowed and that all values must start with an alphabetic character. Package paths can contain dots, and the identifier may contain one or more underscores. The input is automatically checked against these rules.
The Binding Editor helps in defining new bindings even quicker. As soon as you start typing, the editor looks for similar entries in the table and suggests a matching value using autocompletion. If you do not know the exact package path or name of the new component, you can browse for the corresponding class file—the rest is done by the editor. This feature assumes an existing (and compiled) component class though, which presumably will not be at hand,



Edit Bindings for MY_COMPONENT

Identifier: org.ximtec.ipaper.MY_COMPONENT

Logic: org.ximtec.ipaper.myapplication.MyLogic

Stub: org.ximtec.ipaper.myapplication.MyStub

Figure 4.3: A binding is edited

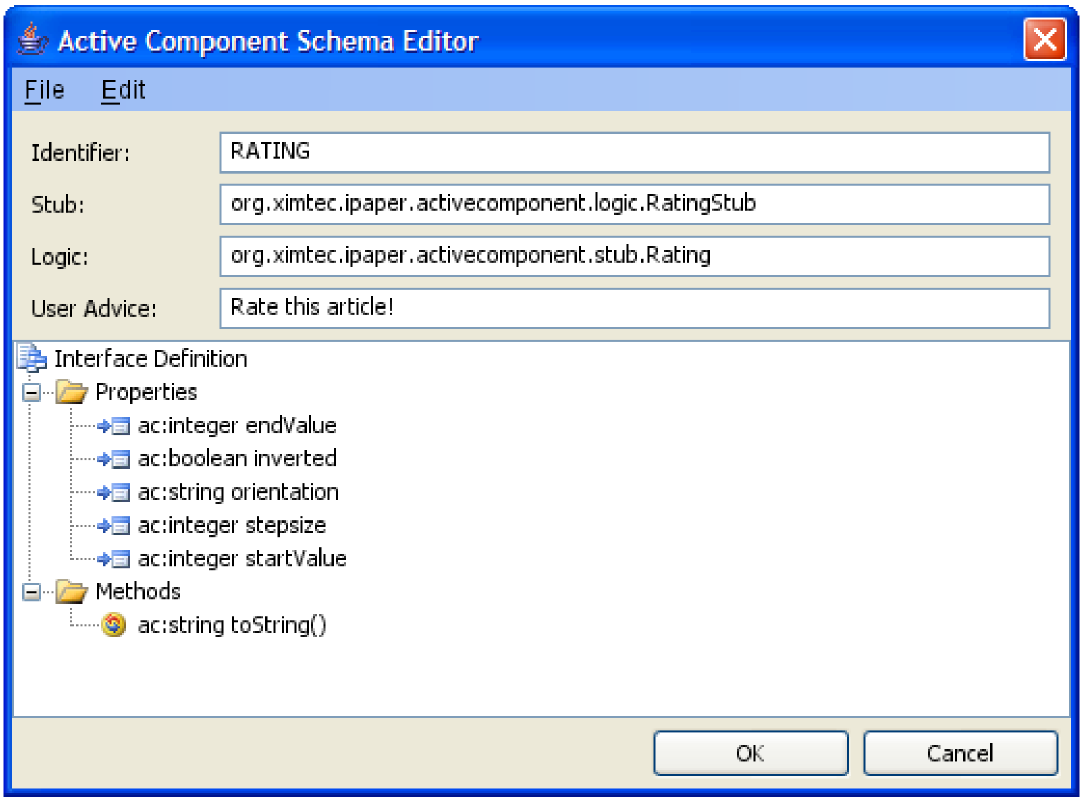
value	**description**
identifier	a unique identifier for the component, usually composed of the package path of your application's components and a proper name (preferably in uppercase), e.g. `org.ximtec.ipaper.myapplication.MY_COMPONENT`
stub	the full name of the stub class name, including its absolute package path, e.g. `org.ximtec.ipaper.myapplication.stub.MyStub`
logic	the full name of the logic class name, including its absolute package path, e.g. `org.ximtec.ipaper.myapplication.logic.MyLogic`

Table 4.1: Naming conventions for Active Component bindings

since it is the purpose of the Schema Editor to define components *after* their binding entry has been entered.
Note that some of the list entries may be displayed in red. This means that for this particular binding, no schema has been specified yet.

4.3.2 Editing Schemas

If you want to define or modify the schema of an Active Component, double-click on the corresponding binding entry, or select EDIT SCHEMA from the EDIT menu. The Schema Editor will be opened.

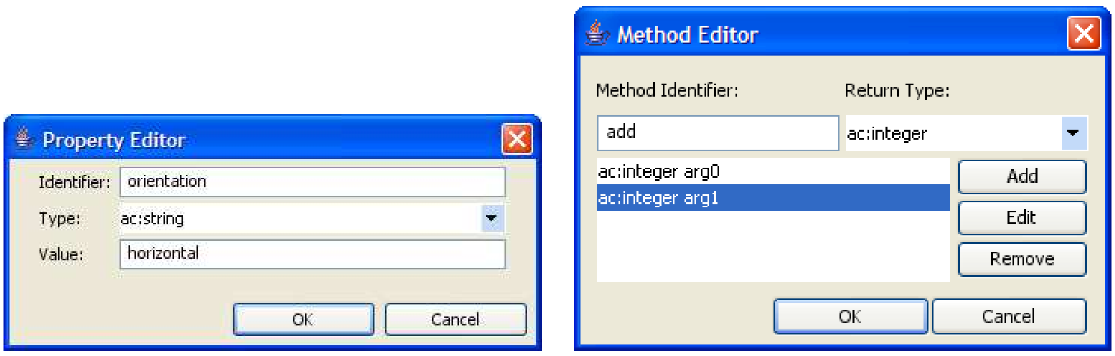


Identifier: RATING
Stub: org.ximtec.ipaper.activecomponent.logic.RatingStub
Logic: org.ximtec.ipaper.activecomponent.stub.Rating
User Advice: Rate this article!

ac:integer endValue
ac:boolean inverted
ac:string orientation
ac:integer stepsize
ac:integer startValue
ac:string toString()

Figure 4.4: The Active Component Schema Editor

Beside attributes like identifier, stub and logic, the Schema Editor allows you to define properties and methods of an Active Component. The interface is straightforward, and you should be able to define your first Active Component schema with ease.
Remember that you only need to define properties which have to be available on both sides (logic and stub). Private properties that are only used locally can be declared directly in the generated class files.
If the schema definition is complete, you can generate Java class stubs for both the Active Component logic and the Active Component stub. Select CREATE JAVA CLASS STUBS from the FILE menu and select a folder where you want the class files to be stored. The following example shows two very simple class files generated by the schema editor.



Identifier: orientation
Type: ac:string
Value: horizontal

Method Identifier: add
Return Type: ac:integer
ac:integer arg0
ac:integer arg1

Figure 4.5: Define properties and methods

```
package org.ximtec.ipaper.activecomponent.stub;
import /* ... */

public class SampleStub extends DefaultStub {

  private static final String ADVICE = "This␣is␣a␣sample␣stub!";

  protected int counter;
  protected boolean paused;

  public SampleStub(ActiveComponent resource) {
    super(resource);
  }

  public String getUserAdvice() {
    return ADVICE;
  }
}
```

```
package org.ximtec.ipaper.activecomponent.logic;
import /* ... */

public class SampleLogic extends ActiveComponentLogic {

  private static final String ADVICE = "This␣is␣a␣sample␣logic!";

  protected int counter;
  protected boolean paused;

  public int getCounter() {
    return counter;
  }

  public SampleStub(ActiveComponent resource) {
    super(resource);
  }

  public String getUserAdvice() {
    return ADVICE;
  }
}
```

5 Conclusion and Future Work

In this chapter, we will discuss some aspects of the Active Component framework which could be improved in other semester projects, i.e. we will show where the framework could be extended, or where it should be redesigned.

5.1 Future Work

5.1.1 Eliminating Binding and Schema Duplication

As shown in Figure 2.5, the binding table which is needed to bind stubs and logics to Active Components as well as all the different schema definition files have to be deployed at both the client and the server side. Synchronization is crutial, because the absence of currect and up-to-date binding information or schema definition will render an Active Component inoperative.
At the moment, this concern may not appear to be justified, because Active Components are usually designed once and remain unchanged thereafter. But the system could be extended to allow Active Components to be loaded or bindings to be changed dynamically.

5.1.2 Replacing Schema Types

Active Component schemas make use of a special `Type` class (see Section 3.2.6), which are used to simplify the conversion from schema types to the XML format and vice versa. They are also used in the remote method invocation mechanism for the same reason, but although the transformation from and to XML is straightforward, the remote method call itself has become unnecessarily complicated (method arguments have to be converted to Active Component schema types). This issue should be tackled as soon as possible in order to facilitate the RMI mechanism, probably using serialization.

5.1.3 Documentation and Developer Manual

Although the source code of the Active Component framework is well-commented, there is no reference material whatsoever describing how the system has to be set up or how Active Components are actually created. As the development on the authoring tool progresses, a documentation ought to be compiled, serving as a “manual” for the Active Component framework.

5.1.4 Bits and Pieces

In order to set up an Active Component, several things need to be done:

- ► Define the Active Component’s interface and create class stubs (using our authoring tool)
- ► Add a new entry to the binding list on the local and the remote host (using our authoring tool as well)
- ► Set up an interactive iServer document (e.g. an iPaper sheet) and create a link to the new Active Component
- ► Compile and re-run the system

Some of these steps are already taken care of by our authoring tool, others still need to be done manually. Furthermore, a mechanism similar to Java’s dynamic class loading could be introduced to the Active Components framework as well, in order to dynamically integrate new Active Components at runtime.

5.1.5 Integrated Authoring Tool and Eclipse Integration

The process of creating Active Components involves a lot of code-writing. For small components, constant switching between the development environment and the authoring tool for Active Components may be tolerable to a certain extent. But especially when it comes to reverse engineering (i.e. if the users changes the Active Component’s interface by directly modifying the source code), the external authoring tool is not appropriate and slows down the development process drastically. Actually, reverse engineering has not yet been taken into account at all.
We propose that the authoring tool should be directly integrated into the existing Eclipse IDE. The aforementioned conversion between Active Component schemas and the corresponding Java source code (and vice versa) could then be done automatically by the IDE, and we could take advantage of its refactoring mechanisms.

A

Appendix A

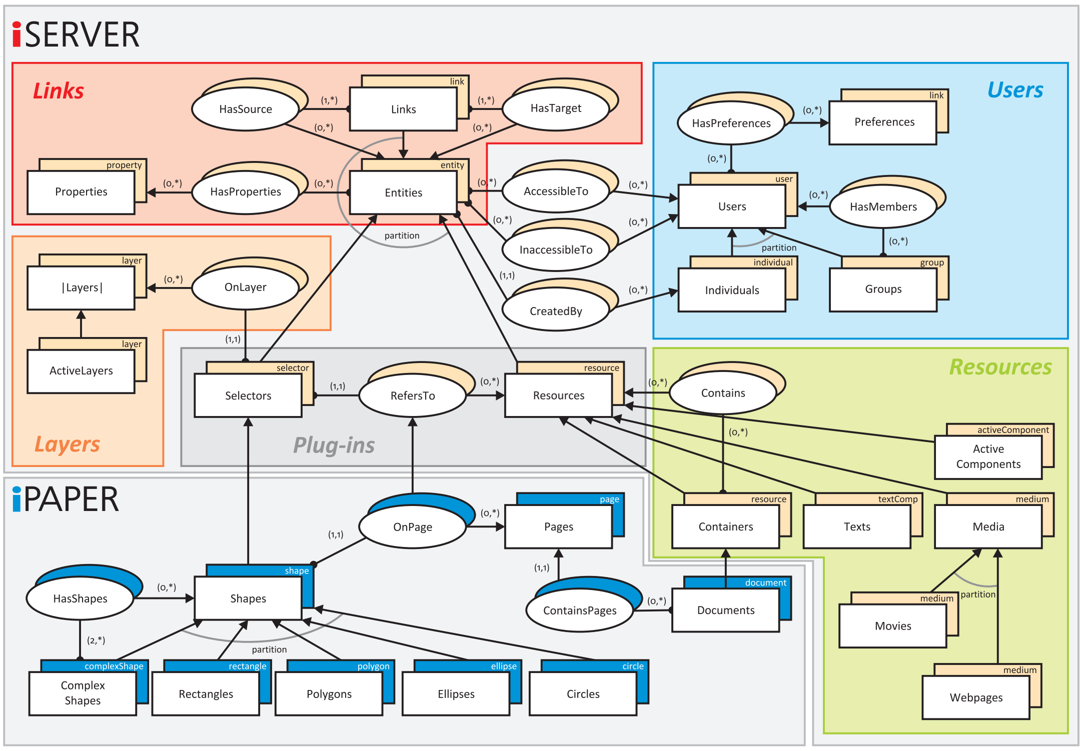


Figure A.1: iServer and iPaper Object Model

B
Appendix B

Listing B.1: XML Schema of the Binding File Format

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.globis.ethz.ch/iserver">
 <xsd:element name="activeComponents">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="activeComponent" type="bindingEntryType"
                    minOccurs="0" maxOccurs="unbounded"/>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="bindingEntryType">
   <xsd:sequence>
     <xsd:element name="identifier" type="xsd:string"/>
     <xsd:element name="stub" type="xsd:string"/>
     <xsd:element name="logic" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>

</xsd:schema>
```

Listing B.2: XML Schema of the Schema File Format

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.globis.ethz.ch/iserver">
 <xsd:element name="activeComponentSchema">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="identifier" type="xsd:string"/>
       <xsd:element name="stubPackage" type="xsd:string"/>
       <xsd:element name="logicPackage" type="xsd:string"/>
       <xsd:element name="superStub" type="xsd:string"/>
```

```
      <xsd:element name="superLogic" type="xsd:string"/>
      <xsd:element name="properties">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="property" type="propertyType"
                     minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="methods">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="method" type="methodType"
                     minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="propertyType">
  <xsd:sequence>
    <xsd:element name="identifier" type="xsd:string"/>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="value" type="xsd:string"/>
    <xsd:element name="readable" type="xsd:boolean"/>
    <xsd:element name="modifiable" type="xsd:boolean"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="methodType">
  <xsd:sequence>
    <xsd:element name="identifier" type="xsd:string"/>
    <xsd:element name="returnType" type="xsd:string"/>
    <xsd:element name="arguments">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="argument" type="propertyType"
                   minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

</xsd:schema>
```

Listing B.3: XML Schema of a Remote Method Invocation

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.globis.ethz.ch/iserver">
 <xsd:element name="activeComponentSchema">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="className" type="xsd:string"/>
      <xsd:element name="methodName" type="xsd:string"/>
      <xsd:element name="arguments">
```

```
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="argument" type="argumentType"
                           minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="argumentType">
    <xsd:sequence>
      <xsd:element name="identifier" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Acknowledgements

I would like to express my sincere thanks to my supervisor, Beat Signer, who not only let me take part in the PaperWorks project, but also supported me patiently during this thesis and provided a lot of valuable information. Since this work is based on the Active Component framework designed and developed by Philipp L. Bolliger, I would like to thank him for the awsome work he has done during his thesis. I am also grateful for the great feedback that I was given by some members of the GlobIS group after my project presentation, which gave me an idea about where the project could be heading in the future. Finally, I would like to thank Prof. Moira C. Norrie for letting me participate in one of her group's projects.

Bibliography

[1] **Active Components for iServer:** In Consideration of Paper-Based Authoring
Philipp L. Bolliger (April 2005)

[2] **Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces**
Beat Signer (2005)
Dissertation ETH No. 16218, Zurich, Switzerland

[3] **Paper Augmented Digital Documents**
François Guimbretière (July 2003)
Human-Computer Interaction Lab, University of Maryland

[4] **DCOM Architecture**
Markus Horstmann, Mary Kirtland (July 1997)

[5] **The Distributed Component Object Model**
The Dalmatian Group, Inc.
`www.dalmatian.com/com_dcom.htm`

[6] **Wikipedia**
`http://en.wikipedia.org/wiki/CORBA`
`http://en.wikipedia.org/wiki/DCOM`

[7] **A comparison of two competing technologies**
David Reilly (2000)
`www.javacoffeebreak.com/articles/rmi_corba`

[8] **Java RMI, CORBA or COM?**
Prithvi Rao (November 1998)
`www.usenix.org/publications/java/usingjava13.html`

[9] **The Java Cookbook**
Ian Darwin (June 2001, O'Reilly Press)

[10] **The Java Developers Almanac 1.4**
Patrick Chan (March 2002)

[11] **Easy Java/XML integration with JDOM**
Jason Hunter, Brett McLaughlin (May 2000)
`http://www.javaworld.com/javaworld/jw-05-2000/jw-0518-jdom.html`