Active Components as a Method for Coupling Data and Services – A Database-Driven Application Development Process

Beat Signer[1] and Moira C. Norrie[2]

[1] Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
bsigner@vub.ac.be
[2] Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
norrie@inf.ethz.ch

Abstract. In the area of highly interactive systems, the use of object databases has significantly grown in the past few years due to the fact that one can, not only persistently store data in the form of objects, but also provide additional functionality in terms of methods defined on these objects. However, a limitation of such a tight coupling of objects and their methods is that parts of the application logic cannot be reused without also having instances of these objects in the new application database. Based on our experience of designing multiple interactive cross-media applications, we propose an approach where we distinguish between regular database objects containing the data and so-called *active components* storing metadata about specific services. Active components are first class objects which, at activation time, can perform some operations on the server as well as on the client side. Since active components are standalone lightweight components, they can be dynamically bound to single objects or semantically grouped sets of objects and be automatically invoked by different forms of database interactions. The database-driven development of arbitrary client and server-side application functionality not only simplifies the design of highly interactive systems, but also improves the reuse of existing components across different systems.

1 Introduction

In recent years, there has been a rapid growth in the number of applications where the same data can be accessed by different input modalities as well as from a variety of input devices. These types of highly interactive applications generally require an adaptation of the content as well as the form of interaction in order to become accessible from different client devices. Nevertheless, the database is often seen purely as a storage container for data, with any complex interaction handled and implemented in an application-specific manner.

If, however, a database can not only store data but also some general application logic, this functionality can be reused in the development process of a specific application, thereby simplifying the design of new applications in terms of time and cost. Furthermore, by reusing existing application logic, the corresponding functionality gets refined and optimised over time leading to more stable and less error-prone applications.

Of course, the idea of modular software development and the reuse of components is not a new one and there exists a variety of different solutions for component-based software development such as the OSGi Service Platform[3]. Also Web Services and service-oriented architectures (SOAs) offer a method for loosely coupling different distributed components and composing complex applications out of simple building blocks and services.

However, for many existing solutions, the configuration and use of the services still requires substantial programming skills and often the solutions are too heavyweight for applications that should run on devices with limited resources. They are based on a simple remote method invocation (RMI) mechanism where a client-side proxy component offers the functionality of a remote service. While this simplifies the development of applications with a distributed application functionality, it does not really provide a method for designing reusable components for more complex client-server interaction. Based on our experience in developing multiple interactive cross-media applications and working together with, not only programmers, but also designers and artists, we identified a need for a less programming intensive solution for reusing functionality in the development of these kinds of applications. We present a solution where application functionality can not only be executed remotely on the server but also run on the client side and enable complex interaction between client and server-side application functionality.

In this paper, we show how database-driven application development can be simplified through the concept of active components. Some motivational examples for database-driven client and server-side functionality are provided in Sect. 2. Related work in terms of solutions for reusing component-based application functionality in software engineering and also active databases is discussed in Sect. 3. In Sect. 4, we introduce the concept of active components and outline the basic idea of storing data as well as services in an object database. We then present our architecture for executing active components in Sect. 5 and provide some details about the prototype implementation of our new active component-based approach in Sect. 6. Different active component use cases are presented in Sect. 7 before providing concluding remarks in Sect. 8.

2 Motivation

To motivate our approach, we will first provide some application scenarios where data managed by a database system is accessed through a combination of database-driven client- and server-side services. We will later show how this kind

[3] http://www.osgi.org

of database-driven client-server interaction can be realised based on our active component solution for coupling data and services.

If we think about the evolution of the Web and how rich Internet applications (RIAs) are nowadays used to mimic the behaviour of desktop applications within a browser, we can see that a similar behaviour can be achieved by using some form of active database content that is deployed to and executed on the client side. RIAs normally need a browser plug-in or a virtual machine to run the client-side components, which is the equivalent of a runtime environment for active components deployed to the client side as introduced later in this paper. If we implement an active component runtime environment as a browser plug-in, we can execute the client-side active component directly within a web browser for the access to and manipulation of any remote data stored in an object database.

While in most service-oriented architectures there is an explicit remote execution of services offered by a server, we would like to introduce a scenario where services are executed implicitly by accessing database objects. A service could for example be associated with specific object instances, object types or semantic collections of database objects. If such a database object with an associated active component is accessed, the linked active component is automatically loaded by an active component runtime environment. An example where active components are associated with specific media on the type level is shown in Fig. 1.

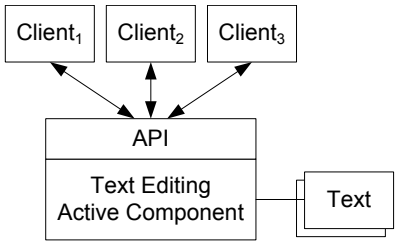

Fig. 1. Media service component

The idea is that specific applications (clients) no longer directly deal with raw media types but rather use services offered by active components coupled to these media types. In the example shown in Fig. 1, all objects of type `text` have been associated with a special text editing active component. This text editing component provides some basic text editing functionality such as the insertion or deletion of a set of characters within a text which can be reused and shared by different client applications. Of course we would easily have the possibility to run different text editing active component implementations at remote sites while still having control over who is currently accessing a specific text resource based on a given server-side text editing component. The text editing component could be used by standard word processors (e.g. Microsoft Word) as well as browser-based text editors or a basic editing active component executed within the active component runtime environment of small portable devices.

Since the text editing functionality is not implemented as methods on the text data objects themselves, we could still exchange the services provided for specific classes or instances of media types by simply dynamically reconfiguring the associated active components. Another advantage is that we could provide editing functionality for rich media types out of the database and the number of supported formats could be extended in a flexible way. The active component runtime environment would be in charge of providing a layer for the basic functionality by running the media service active components whereas specific input and output devices might provide different user interfaces on top of these media service components. Since we can not only define active component services on a type level but also use role-driven service invocation based on object classification, our text objects in Fig. 1 could be dynamically reconfigured over time and bound to other services (e.g. a logger service) based on their classification. Note that type-based as well as role-driven service invocation is a powerful mechanism for automatically triggering any object-related functionality which otherwise would have to be explicitly accessed by static library method calls.

Another use case we would like to address is the processing of data generated by physical objects such as sensors. In today's "Internet of Things" where more and more physical objects become integrated with the digital world by sensing for example some environmental parameters, it becomes important that input from these objects can be easily integrated with digital information spaces as represented by a database. If we can guarantee that some kind of active component runtime environment is available for the augmented physical objects, then the client-server active component communication provides a lightweight solution for updating sensor data in a database. At initialisation time, a physical object could query for a specific active component which would then define the application logic on the client and server side and provide functionality for database updates.

In the next section, we present related work in terms of solutions for active databases as well as component-based and service-oriented architectures. We then introduce our database-driven and active component-based development process for applications such as the ones presented in this section.

3 Related Work

In component-based software engineering (CBSE) [1], the emphasis is on building modular components with well-defined interfaces which can then be composed to develop more complex systems. This allows components to be aggregated in a distributed manner as well as within a single local system. More recent technologies for software components in distributed computing are Web Services [2] and service-oriented architectures (SOAs) [3]. While these technologies provide a solution for language independent reuse of business logic as well as data exchange, there is still quite some effort required for a developer to register and make use of existing Web Services. Also, data exchange over the transport layer is well suited to standard business applications, but does not suit the processing

of real-time and streaming data as produced, for example, by different types of physical sensors. More recently there have also been some efforts to apply SOA principles to DBMSs to provide loosely-coupled database services [4], but in this case the principles are applied to the DBMS rather than to the applications.

While Web Services mainly focus on digital services, there exist other approaches trying to integrate physical devices as elements of a component-based architecture. For example, as part of a research project, the OSGi model has been generalised to support the "Internet of Things" by turning physical devices and objects into loosely coupled software modules that interact with each other through service interfaces [5]. Since OSGi can use direct method invocations without requiring a transport layer, it is much faster than a Web Service approach. In contrast to Web Services, OSGi components can also directly react to the appearance and disappearance of new services. However, the extended OSGi model deals with integration on a rather low level in terms of different transport protocols such as Bluetooth and is less concerned with higher level concepts directly supporting the application developer.

The connection and integration of devices with a service oriented architecture is the idea of the Service-Oriented Device Architecture (SODA) [6] and some of its implementations such as DBNet [7]. The SODA solution is effective for connecting powerful client devices but less suited to realising lightweight services as required by devices with limited computing and communication resources.

Database systems were traditionally designed to store application data which was then accessed and manipulated by one or more application programs. While the data was managed by the database system, the application logic normally formed part of a specific application accessing the database. The handling of any application logic outside of the database potentially results in a replication of functionality if multiple applications are going to use the same data and implement the same or similar functionality. Of course, this often led to maintenance problems if parts of the application logic had to be updated at a later stage since changes to different implementations were necessary.

The idea to move functionality from the application layer into the database was originally introduced to perform integrity checks within the database. Nowadays almost all commercial relational database systems support triggers as a form of automatically executing some functionality, for example implemented as stored procedures [8], within the database. Triggers are usually represented as event-condition-action (ECA) rules which were adopted as the main means of representing business logic in the paradigm of active database systems [9]. The event describes the happening, inside or outside of the database, to which a rule might respond. Upon a specific event, the condition part of an ECA rule is checked and, if necessary, the corresponding action is executed.

There were also proposals to extend object databases with ECA rules to support the active database paradigm. For example, the TriGS system [10] was an active object database based on GemStone [11]. In these systems, the application-specific interaction with data is often defined by methods on the data objects and the ECA rules are mapped to invocations of these methods.

In contrast to active object databases, we propose a clear distinction between regular data objects and active components providing services based on these data objects. This is mainly due to the fact that functionality provided by the active components is sometimes closely related to the interaction with different input and output modalities. Therefore, this functionality should not be implemented as methods on the class level since this would restrict its reuse across different classes of database objects without (mis)using inheritance. However, this additional extrinsic object behaviour should not be implemented via various static library calls but be designed as active components that are bound to data objects and can access and update any information managed by these data objects and also create new data objects.

Furthermore, an active component should not only be able to be triggered by a single event but also process successive events if required. These long-lived interactions—note the similarity to long-lived transactions—are very helpful in the design of more complex types of interactions with data where the single triggering of a method is not sufficient since additional input data (e.g. streaming data) should be processed by an active component. While active object databases support the execution of methods on database objects, our active component-based solution also provides a mechanism to deploy and run parts of the application functionality on the client side, thereby enabling rich types of long-lived interactions between client- and server-side active components.

4 Approach

The active component concept was originally developed as part of a general link server for cross-media information management [12]. While the server initially supported the cross-linking of arbitrary types of digital and physical resources, we were looking for a way to integrate, not only data and information in terms of different resources such as web pages, movie clips and sounds, but also services as represented by small software components that would be executed when a link is activated.

The concept of active components has been generalised and can now be instantiated as an *active component runtime environment* on top of an existing object database system. The activation of active component services is managed by the active component runtime environment. In addition to the set of regular database objects $\mathscr{O} = \{O_1, O_2, \ldots, O_m\}$, the database now also has to persistently handle a set of active components $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$.

The active component runtime layer enables the definition of associations or links l_i between arbitrary database objects O_j and active components A_k. The active component runtime environment shown in Fig. 2 checks for any associated active components after each query processed by the database system. If a returned object O_i has an associated active component A_j, the active component service is started and gets a handle to the database object O_i. Note that, in the current version, only queries resulting in single database objects may also trigger an associated active component service. Result sets are currently treated

without the additional active component features but we plan to investigate how this could best be handled in future research. The coupling of database objects with active component services can be achieved on the *instance level* as well as on the *schema class* level. Let us assume that our object database contains the classes or types $\mathscr{T} = \{T_1, T_2, \ldots, T_n\}$. A link l_i can then be defined between a type T_j and an active component A_k which implies that the active component service is also bound to all the subtypes of T_j. For example, an association l_i can be defined between a type `contact` and a service A_k that operates on objects of a contacts database. Each time a contact object is accessed, the active component runtime layer ensures that the associated service is started and gets access to the contact object.

Our object-model further distinguishes between the typing of objects for representing behavioural properties and the semantic classification of objects by grouping them into collections [13]. Therefore, a third possibility for the service binding is to define an association l_i between an active component A_j and one or multiple of these semantic collections $\mathscr{C} = \{C_1, C_2, \ldots, C_n\}$ resulting in *role-driven service invocation*. Since these role-driven services are no longer bound to the object type, objects can easily gain and lose service functionality over time by simple reclassification (service evolution). Note that the kind of implicit service invocation presented in this section is not available in most service-oriented architectures where services have to be invoked explicitly.

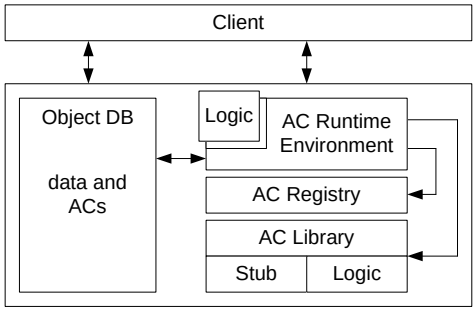

Fig. 2. Active component runtime environment

A second way of accessing a service offered by an active component is to query the active component directly by its name (shown in Fig. 3). After a query is sent to the database, the active component runtime environment checks whether the returned object is an active component A_i. In the case that the database returns an active component, the object is not forwarded to the requester but instead the corresponding service is invoked. Note that in this direct form of service invocation, the active component will not get a handle to any linked data object O_j.

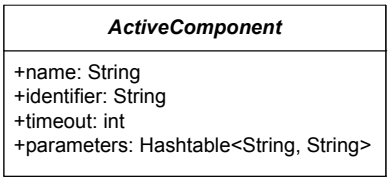

Fig. 3. Active component database object

Since an active component only contains some data about the service to be invoked, we need a way of getting access to the actual program code to be executed. The `identifier` field provides a unique identifier which is used to lookup the corresponding stub and logic classnames in an *active component registry* and fetch the classes from the *active component library*. An example of an entry in the active component registry is shown in Fig. 4. After a classname has been retrieved from the active component library, the Java reflection mechanism is used to dynamically load the corresponding Java class and initialise it with any data provided by the active component database object.

identifier	org.ximtec.iserver.activecomponent.BROWSER
stub	org.ximtec.iserver.activecomponent.stub.BrowserStub
logic	org.ximtec.iserver.activecomponent.logic.BrowserLogic

Fig. 4. Active component registry entry

Due to the fact that some active components will deal with multiple input events and we can never be sure whether further data has to be processed, an active component may have an optional `timeout` field that defines after how many milliseconds without a new input event an active component should be terminated. Last but not least, each active component can contain an arbitrary number of properties in terms of key/value string pairs defining different parameters of the service to be invoked.

By decoupling the functionality offered by an active component service and the data object stored in the database, one gains flexibility in reusing the corresponding functionality since it is no longer implemented as a class method and therefore no longer tightly coupled to a specific class of objects. Furthermore, through the use of the active component registry and library, the implementation of a specific service offered by an active component can be easily updated or replaced at any time. The introduction of active components as first-class objects eliminates the need to introduce artificial class hierarchies just for the sake of reusing some application functionality and leads to a cleaner and more flexible integration of data and the corresponding services.

5 Architecture

As stated earlier, our active component-based solution enables not only the remote invocation of server-side services but also more complex and richer types of interaction with any database content. Often it is not sufficient for an active component to react to and process a single event, but instead it needs to establish some long-term interaction with a client application or device. In addition to the active components managed by the active component runtime layer in combination with the DBMS (active component logic), we therefore also support the concept of active components that are deployed to the client side by the database (active component stub) as highlighted in Fig. 5.

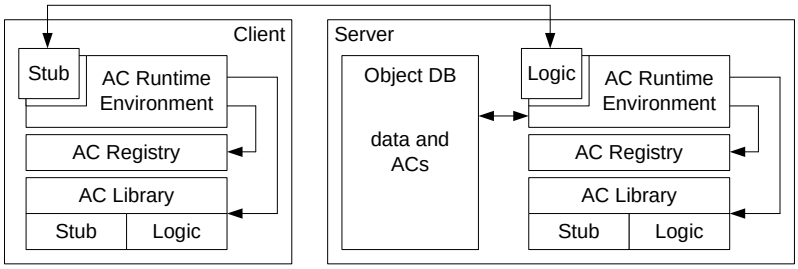

Fig. 5. Client-server active component runtime environment

In this new client-server active component scenario, the first phase is still the same. After the active component runtime environment has detected that an active component has been returned as a result of a query, the corresponding active component logic is instantiated and initialised on the server side. In the next step, a representation of the active component including its unique identifier and all the other fields is sent to the client side. The active component response is detected by the client-side active component runtime environment and, after a lookup in the active component registry, the corresponding active component stub class is fetched from the active component library and executed on the client side. In Fig. 5, the active component registry as well as the active component library are part of the client environment. However, these two lookup services could also be accessed remotely.

An interesting aspect of the client-server active component approach is that the active component stub can take over the control over any input from the client and communicate directly with the server-side active component logic instance. It is up to the implementation of a specific active component stub to define any criteria for the termination of an active component service by calling a special `setDone()` method. Note that the terms client and server are to be interpreted on a conceptual level and do not necessarily imply that the active component stub and logic instances have to run on remote sites. It is even

possible that an active component stub interacts with an active component logic within the same virtual machine.

We can basically distinguish three different types of active component services that can be driven by the database. If only data has to be created, updated or deleted on the database side, an active component logic instance can be used to implement this kind of functionality as shown in Fig. 6. After a database request has been processed, the corresponding active component logic is loaded and a confirmation message (OK) is sent to the client. Note that the confirmation for the client can either be sent after the active component stub has been successfully initialised or after the execution of the active component logic has finished. This solution is closely related to the AOODB approach with the difference that interactions may also be driven explicitly from outside rather than being based only on the internal database state. An example of such a server-side active component service could be an active component that provides some business logic and updates multiple database objects when invoked.

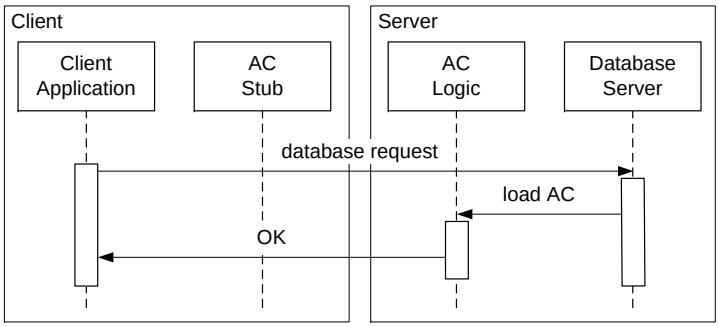

Fig. 6. Server-side application functionality

The second type of service involves the client component only. In this case, illustrated in Fig. 7, the server-side active component logic just sends an active component specification to the client side without executing any application functionality. The client-side active component runtime environment loads the corresponding active component stub instance and executes its functionality. An example of a client-side active component could be a `Movie` active component that is deployed to the client together with the URI of a movie clip as a parameter and opens the movie in the client's default movie player.

The advantage of this database-driven execution of client-side functionality is that it becomes easier to deploy specific functionality to different client devices. As long as a client device provides an active component runtime environment, an active component stub can be executed on different devices. Another advantage of the database-driven deployment of client-side services is that we can avoid any redundant installation and update of services on different client devices since the functionality is deployed to these devices on demand.

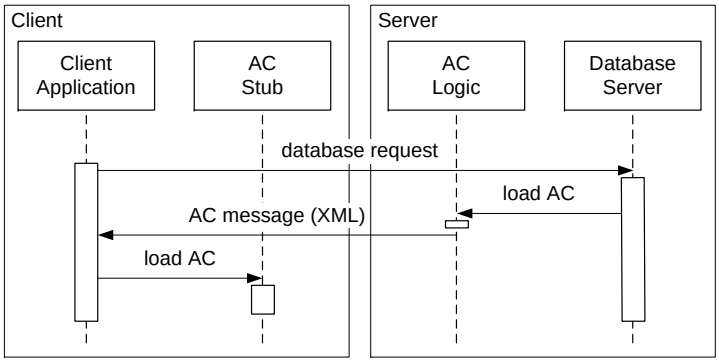

Fig. 7. Client-side application functionality

The most powerful active component service solution involves the combined execution of application functionality on the server as well as on the client side and potential communication between the client- and server-side active components as highlighted in Fig. 8. This flexible approach supports a variety of use cases ranging from consistency checks on the client side before sending update queries to the database to the filtering and streaming of real-time data. In Sect. 7, we will provide some examples of how this client-server active component approach has been used for implementing highly interactive user interfaces to databases.

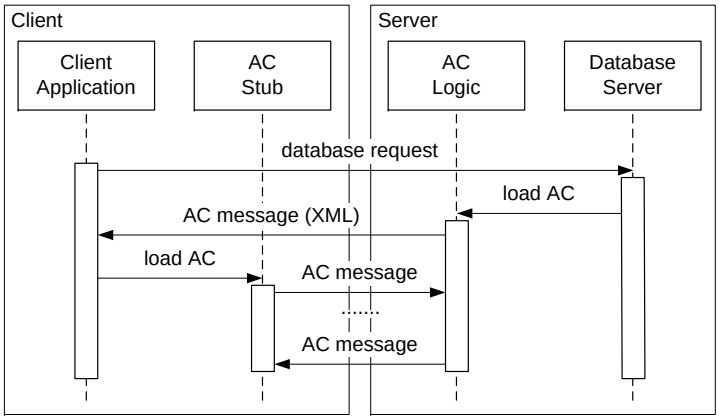

Fig. 8. Client- and server-side application functionality

Note that the second as well third type of active component services, where part of the active component functionality is executed on the client side, are not supported by AOODB solutions or Web Service approaches.

6 Implementation

A first prototype of the active component framework for database-driven services has been realised in Java on top of our own object-oriented data management framework [14]. However, the presented concepts are general enough to be implemented in other programming languages and environments. Since our application scenario was mainly dealing with various client devices accessing and interacting with information stored within our object-oriented database system, we have chosen a classical client-server architecture where the client communicates with the server over the HTTP protocol. The active component communication does not have to be limited to a single protocol and different configurations are possible.

While our OODBMS stores active components in terms of database objects containing the relevant information to initialise the services at request time, it is up to the client- and server-side active component runtime environments to start the corresponding services. For each uniquely identifiable service, the corresponding stub and logic Java classes have to be registered in the active component registry. The client- and server-side active component runtimes use this information provided by the active component registry to dynamically load the classes.

The active component logic and stub classes share some common features such as all the metadata provided by the active component database object as well as an initialisation method as shown in Fig. 9. After an active component stub has been loaded, its `init()` method is invoked. The active component metadata is then serialised in XML and sent to the client-side active component runtime environment. After deserialising the XML message, an active component stub is instantiated on the client side. The active component stub provides an enhanced initialisation method where the component gets not only access to its configuration data (`ACConfiguration`) but also a handle to the client (device) initiating the interaction. Subsequent events are processed by the `processEvent()` method and there might be some potential communication with the server-side active component logic. Any request from an active component stub to its corresponding active component logic is sent in XML format and processed by the active component logic's `handleActionRequest()` method which generates an appropriate response.

It is up to the client-side active component to decide when its work has to be finished and the component has to be unloaded. As soon as the active component stub's `setDone()` method gets invoked as part of the component's program logic, the active component stub is unloaded by the client-side active component runtime environment. Note that before its removal from the system, there is an upcall to the active component's `finish()` method. This enables the active component developer to perform any necessary cleanup and release of acquired resources such as database or network connections. In addition, a client- or server-side active component has an optional timeout parameter and is terminated automatically if it has been idle for longer than a given amount of time.

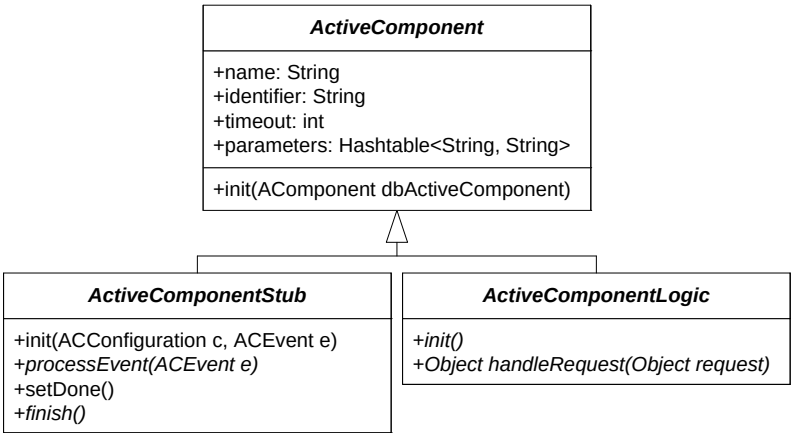

Fig. 9. Active component stub and logic

There are some *resource-specific active components* which require additional information from the client triggering the active component. These components can be reused in different applications but they always have to be used in combination with a client providing the appropriate input data. On the other hand, *generic active components* do not depend on any additional information from the client. An example of a generic active component is a `Browser` active component opening the system's default web browser with a given URI parameter.

Note that while the current implementation is based on Java, it is also possible to support active components implemented in other programming languages. It would even be possible to have stub and logic components that are implemented in different languages given that the communication could, for example, be over XML. It is up to the active component registry and active component library to provide access to an active component in the required programming language based on the active component identifier. Of course in order to support active components implemented in other programming languages, we would also have to implement additional active component runtime environments.

By providing the corresponding active component runtime environment on top of our data management framework, we also plan to implement some of the examples introduced earlier in Sect. 2.

7 Use Case

A major advantage of having the active components as first class objects within the system rather than implementing the corresponding functionality within a method that is tightly bound to a specific database object is that it becomes easier to reuse functionality of existing active components by inheritance. To illustrate this, we provide an example of several active components that build on top of each other and have been implemented as part of our interactive paper platform (iPaper) [15, 16] for processing digital pen input. Digital pen and

paper technology[4] enables the continuous capturing of a pen's position on ordinary paper augmented with a position encoding pattern. The captured digital information can, for example, be processed by an active component and used to trigger digital actions and services. Note that we only show the details for the stub components since, for this specific task, most of the interaction takes place on the client side.

As part of a specific iPaper application, we wanted to capture pen stroke information from a digital pen, perform intelligent character recognition (ICR) on the captured handwriting and output the recognised text using a text-to-speech (TTS) engine. Instead of implementing this functionality as a monolithic piece of program, the active component-based approach enabled us to separate the functionality into several reusable active components highlighted in Fig. 10.

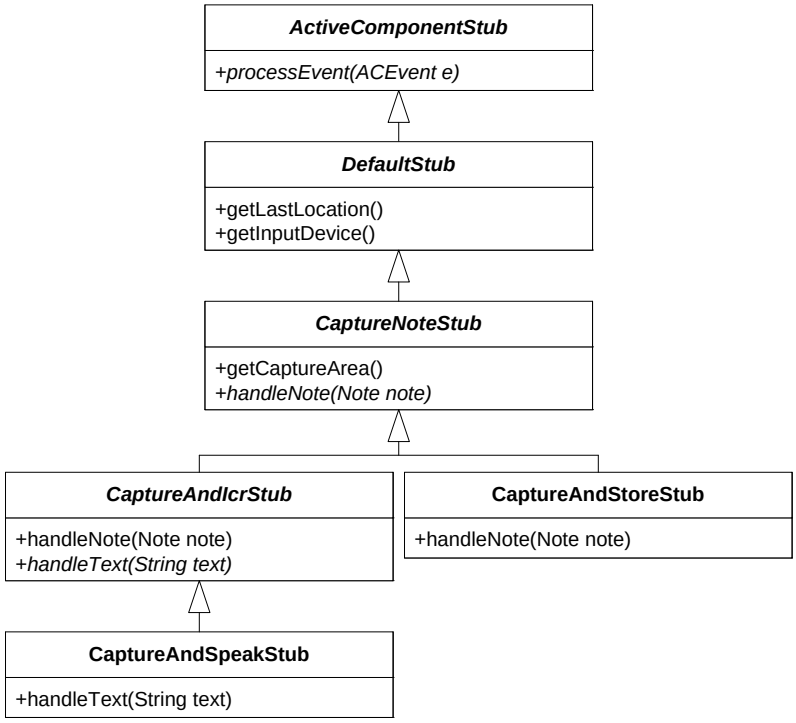

Fig. 10. Active component examples

A first client-side active component is the `DefaultStub`, an extension of the `ActiveComponentStub` providing some general functionality required by many interactive paper active components. For example, the `DefaultStub` provides access to the last pen position (`getLastLocation()`) that was processed by the

[4] http://www.anoto.com

active component or offers a handle to the buffered input device. Note that the `DefaultStub` is an abstract class and therefore it is not possible to directly instantiate any `DefaultStub` active component.

The `CaptureNoteStub` active component extends the `DefaultStub` class and offers the functionality to capture handwritten notes. For example, the `CaptureNoteStub` communicates with its server-side logic component to get information about the capture area from the interactive paper database. This information is accessible via the `getCaptureArea()` method and is used as a criteria to finish the capture process and terminate the active component as soon as the pen leaves the predefined capture area. The `CaptureNoteStub` is still an abstract class which can be accessed by other services that would like to build on top of a capture service. As a result of the capture process, there is an upcall to the abstract `handleNote()` method with the captured note as an argument, as soon as the capture process finishes. Note that the configuration of a `CaptureNoteStub` active component contains a variety of other key/value properties to, for example, define whether a captured note should be cropped.

A simple active component that makes use of the `CaptureNoteStub` service is defined in the `CaptureAndStoreStub` class. By overriding the `handleNote()` method, the `CaptureAndStoreStub` active component gets access to the captured note and stores it in the local file system. The configuration parameters of the `CaptureAndStoreStub` component include information about the format of the document to be stored (e.g. jpeg or gif) as well as the path and filename.

This example of an active component storing the captured note in the file system was only introduced to show that an active component's functionality, in this case the one of the `CaptureNoteStub`, can be reused by many different active components. Our goal is to further process the capture information and therefore we implement a `CaptureAndIcrStub` that takes the output of the `CaptureNoteStub` component, performs some intelligent character recognition on the stroke data and returns a text in string form. The `CaptureAndIcrStub` is again an abstract class and the `handleText()` method has to be overridden by any concrete subclass.

Last but not least, the `CaptureAndSpeakStub` class is an extension of the `CaptureAndIcrStub` component implementing the `handleText()` method and feeding the text to a text-to-speech engine. A summary of the method upcalls within the inheritance hierarchy of the `CaptureAndSpeakStub` class is shown in Fig. 11.

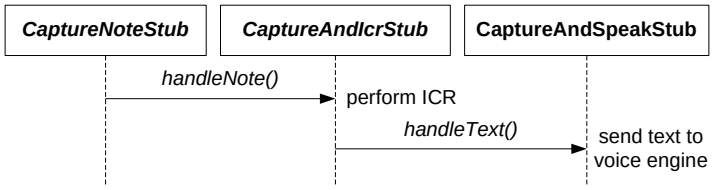

Fig. 11. Active component method calls

Figure 12 outlines the interaction between the `CaptureAndSpeakLogic` and `CaptureAndSpeakStub` components as part of a capture process. After the `CaptureAndSpeakLogic` and `CaptureAndSpeakStub` components have been initialised, the `CaptureAndSpeakStub` sends a request to the server-side active component to get information about the capture area. The `CaptureAndSpeakStub` then autonomously processes any positional input from the digital pen until the pen leaves the predefined capture area. It finally applies an intelligent character recognition (ICR) algorithm to the captured data and sends the resulting string to a text-to-speech (TTS) engine. While it was not the goal of this active component to store the captured information, this functionality can be easily realised by sending a message with the captured data to the server-side active component. This server-side storage of captured information based on active components was used in the EdFest interactive paper-based festival guide [17] for sharing comments. In the EdFest application, active components were not only used to persistently store captured information but also to send requests to external databases.

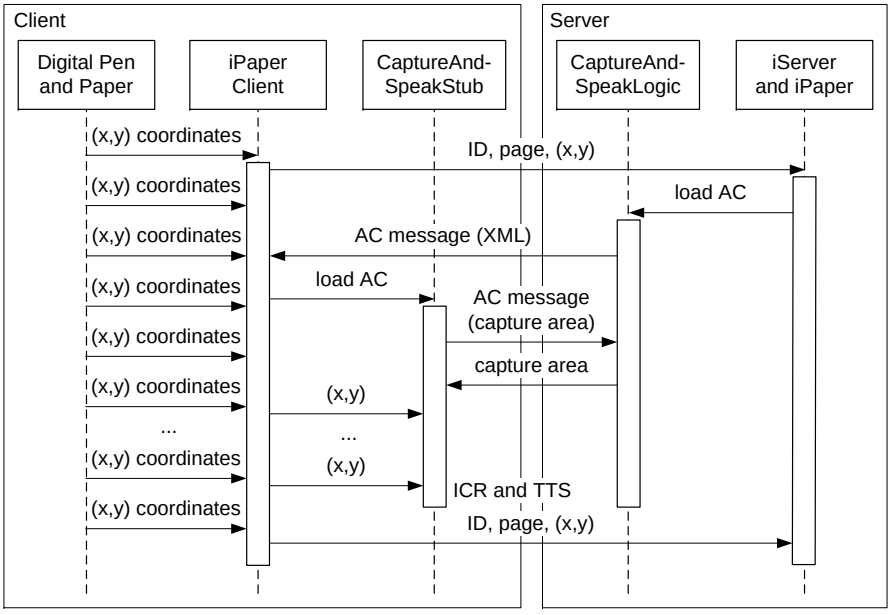

Fig. 12. Client and server active component interaction

To achieve the task of capturing some pen-based input, recognising the handwritten information and producing the corresponding voice output, we have defined four reusable components. There are several advantages of this fine granularity of functionality offered by the different active components. First of all, the frequent reuse of the components should enhance the overall quality of the

components over time. Since there is a growing set of active components, it also becomes easier for the developer to design and implement extensions of existing active components. Another advantage is that the code size of a single active component is relatively small and therefore it is easy to understand the functionality that it offers. Furthermore, we gain flexibility by storing the active component configurations in a database since all of its parameters can easily be adapted at runtime without a recompilation of the source code. However, the reusability of active component functionality within and across applications is only one aspect. As we have shown, another important benefit is the definition of modular and component-based client-server interaction which goes far beyond the "simple" remote service invocation offered by other solutions.

Our database-driven application development process based on active components has been successfully used to realise multiple highly interactive cross-media applications. These applications included artistic installations [18] as well as a variety of interactive paper applications [16, 19].

8 Conclusions

We have presented a database-driven approach for developing highly interactive applications based on active components. While many systems focus on the integration of services on the protocol level, our approach provides a high-level lightweight solution for an application developer to implement and reuse modular services. The clear separation of data objects and services provided by active components further simplifies the reuse of services with various types of data. Since active components are first-class database objects, their associated services can easily be configured within the database. A service provided by an active component can be flexibly associated with data objects on the instance, type or classification level. Role-driven service invocation provides a flexible mechanism for runtime object evolution in terms of services that are bound to a specific object. Further, the client-server active component runtime environment provides a powerful solution for executing parts of the application on the client side and for establishing a client-server service communication. While service-oriented architectures allow for executing remote services, our active component-based approach enables the definition and execution of autonomous and encapsulated client-server services. The automatic deployment of services to the client side not only simplifies the installation and maintenance of new functionality but also enables a richer form of interaction between client-side services and application data that is stored on the server side.

Acknowledgements

We would like to thank Samuel Willimann and Philipp Bolliger for their work on the prototype implementation of the active component framework. We would further like to thank Stefania Leone and Alexandre de Spindler for their valuable comments on the paper.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Professional (November 2002)
2. Papazoglou, M.: Web Services: Principles and Technology. Prentice Hall (September 2007)
3. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall (November 2004)
4. Subasu, I.E., Ziegler, P., Dittrich, K.R., Gall, H.: Architectural Concerns for Flexible Data Management. In: Proc. of SETMDM 2008, EDBT Workshop on Software Engineering for Tailor-made Data Management, Nantes, France (March 2008)
5. Rellermeyer, J.S., Duller, M., Gilmer, K., Maragkos, D., Papageorgiou, D., Alonso, G.: The Software Fabric for the Internet of Things. In: Proc. of IOT 2008, First Intl. Conference on the Internet of Things, Zurich, Switzerland (March 2008) 87–104
6. de Deugd, S., Carroll, R., Kelly, K., Millett, B., Ricker, J.: SODA: Service Oriented Device Architecture. IEEE Pervasive Computing **5**(3) (2006) 94–96
7. Tok, W.H., Bressan, S.: DBNet: A Service-Oriented Database Architecture. In: Proc. of DEXA 2006, 17th Intl. Conference on Database and Expert Systems Applications, Krakow, Poland (September 2006) 727–731
8. Eisenberg, A.: New Standard for Stored Procedures in SQL. ACM SIGMOD Record **25**(4) (December 1996) 81–88
9. Paton, N.W., Daz, O.: Active Database Systems. ACM Computing Surveys (CSUR) **31**(1) (March 1999) 63–103
10. Kappel, G., Retschitzegger, W.: The TriGS Active Object-Oriented Database System – An Overview. ACM SIGMOD Record **27**(3) (September 1998) 36–41
11. Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., Williams, M.: The Gem–Stone Data Management System. In: Object Oriented Concepts, Databases and Applications. ACM Press (1989)
12. Signer, B., Norrie, M.C.: As We May Link: A General Metamodel for Hypermedia Systems. In: Proc. of ER 2007, 26th Intl. Conference on Conceptual Modeling, Auckland, New Zealand (November 2007) 359–374
13. Norrie, M.C.: Distinguishing Typing and Classification in Object Data Models. Information Modelling and Knowledge Bases VI **26** (1995) 399–412
14. Kobler, A., Norrie, M.C.: OMS Java: A Persistent Object Management Framework. In: Java and Databases. Hermes Penton Science (May 2002) 46–62
15. Norrie, M.C., Signer, B., Weibel, N.: General Framework for the Rapid Development of Interactive Paper Applications. In: Proc. of CoPADD 2006, 1st Intl. Workshop on Collaborating over Paper and Digital Documents, Banff, Canada (November 2006) 9–12
16. Signer, B.: Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces. PhD thesis, ETH Zurich (May 2006) Dissertation ETH No. 16218.
17. Signer, B., Grossniklaus, M., Norrie, M.C.: Interactive Paper as a Mobile Client for a Multi-Channel Web Information System. World Wide Web Journal **10**(4) (December 2007) 529–556
18. Vogelsang, A., Signer, B.: The Lost Cosmonaut: An Interactive Narrative Environment on Basis of Digitally Enhanced Paper. In: Proc. of VS 2005, 3rd Intl. Conference on Virtual Storytelling, Strasbourg, France (December 2005) 270–279
19. Signer, B., Norrie, M.C.: PaperPoint: A Paper-Based Presentation and Interactive Paper Prototyping Tool. In: Proc. of TEI 2007, First Intl. Conference on Tangible and Embedded Interaction, Baton Rouge, USA (February 2007) 57–64