Managers Don't Code: Making Web Services Middleware Applicable For End-Users

Alexander Hilliger von Thile, Ingo Melzer, Hans-Peter Steiert

DaimlerChrysler AG – Research and Technology,
P.O. Box 2360, 89013 Ulm, Germany
{alexander.hilliger_von_thile, ingo.melzer, hans-peter.steiert}
@daimlerchrysler.com

Abstract. Today's web-pages are primarily designed for occasional usage. Professional users therefore use special applications that use Web Services increasingly. As the number of internet-users grows we argue that there is a disregarded growing gap between these professional- and occasional-users we refer to as experienced users. For this group of users with little or no programming-skills web-pages are inefficient but professional applications would be inexpedient. In this paper we describe how to make Web Services applicable for experienced web users. To support single Web Service calls efficiently we present a keyboard-controlled browser-embedded console with command auto-completion that wraps Web Services. To support multiple calls and automation we present a web-based IDE that allows visual composition of Web Service calls and simple control-structures that can be used on demand without installation and programming-skills.

1 Introduction

Since its birth, the internet has grown into a tremendous popularity. Today the number of users still grows and what's even more interesting: these users use the internet more frequently. As the internet was primarily used by programmers and scientist in its early years, it became a mainstream medium in the meantime. But most popular web-sites are primarily designed for occasional usage only. That's why professional users need to use special client-applications. The interface between these services and the application was based on HTTP-site requests and -responses in many cases where an official API was not available. The client-applications were sending HTTP-GET requests by automatically building URLs or HTTP-POST request by simulating a HTML-form transmission to the server. The response from the server – usually a HTML-page – was processed by special parsers. Several publications can be found about this way of web-based query and automation ([1], [2] and [3]). However, using web-pages as an interface has many problems mainly due to page-layout-dependencies [3]. Meanwhile many popular web-sites like Amazon.com and eBay provide official APIs using Web Services to ease program development for their services. But Web Services have programmers as their target group and are therefore uninteresting for the vast majority of internet-users. Today this group of users has

only two choices: using the web-pages, which is inefficient for frequent, experienced users or buying a special application for every service, assuming that their usage-scenario is covered by an application at all. In this paper we describe how to fill this gap-in-between by providing mechanisms to make Web Services available to non-programmers.

We therefore focus on the group of experienced web-users that can be characterized as follows:

- they are non-programmers (little or even no programming-skills)
- use services on the web frequently
- web-site navigation is inefficient for their usage scenario (compare prices of multiple auctions, monitor weather-data, traffic-conditions or validate addresses)

If we take eBay as an example and assume that only 10% of eBay's customers fall into this category of users we would address the need of 9.49 million people [4].

The experienced web-users usage-scenarios can be divided into two categories:

- **query:** (nested) Web Service calls are used to query specific data (i.e. prices)
- **automation:** conditional expressions can be used to execute actions, such as a set of queried values can be compared and a specified action (nested Web Service calls) could be triggered (get cheapest price of all previously selected auctions $a_1..a_n$ and place bid if price is below my limit L)

In chapter 2 we describe mechanisms required to make Web Services accessible for experienced web-users. In chapter 3 we show how to support efficient queries and in chapter 4 how automation of queries can be realized. Section 5 describes related work. We conclude in chapter 6.

2 Considerations

Using Web Services directly without a toolkit by i.e. typing a SOAP-message is of course inefficient. In this section we want to analyze which requirements an appropriate toolkit for Web Service usage has to meet.

We identified the following requirements:

1. Abstraction from Web Service complexity (SOAP, HTTP-connection, bindings, endpoints)
2. Abstraction from programming-complexity (classes, functions, control-structures)
3. No local installations necessary (runtime environments, IDEs)

The first level of abstraction is realized by using an existing framework to wrap Web Services complexity. We used Apache AXIS [5] to generate stubs as Java-classes. We use Java because it is widespread, supports late binding that allows us to dynamically add classes during runtime and it has reflection-mechanisms allowing compiled classes to be analyzed for method- and field declarations. Java might be simple

compared to other programming-languages but is still too complex for our target group. We therefore add another layer of abstraction by using JavaScript to use the generated Java-classes.

```
simple_statement ::= [objectName =]
                     (object_instantiation | object_method_call);

object_instantiation ::= new [package.]classname([parameterlist])

object_method_call ::= [objectName.]method_name([parameterlist])
                       {.method_name([parameterlist])}

parameterlist ::= (objectName | object_instantiation |
                  object_method_call)[, parameterlist]
```

Fig. 1. EBNF-notation of a simple JavaScript statement

JavaScript (later standardized as ECMAScript) [6] is an object-oriented programming language with an easy to learn syntax. For now let us focus on simple statements as defined in fig. 1 using EBNF notation [7]. In the Java-philosophy everything is an object. A program consisting of simple statements creates objects explicitly using the object_instantiation-statement (see fig. 1) or implicitly by using a number or defining a String using quotation marks. A name can be assigned to these objects optionally. This name is used to reference the object and call its member-functions (object_method_call) with parameters (parameterlist) which are also object-references (existing or newly created) or results of nested method calls. The return value – if present – can be assigned a name to as well.

Programs as described above are basically simple from a developer's perspective; however the JavaScript-syntax with its nested curly braces, brackets and parenthesis is still not suitable for non-programmers. In the next chapter we will therefore determine how to map above simple statement to a web-based query console meeting our abstraction requirements. To allow automation using control-structures (*if*, *for*, etc.), define event-listeners and allow scheduling of function-calls easily without the need to write code we will use a graphical model presented in section 4.

Requiring a user to read a manual to understand what methods a Web Service offers and how they are used is unacceptable for our target group. But modern IDEs features like code-completion (context-based list of variables and methods), method and parameter descriptions (such as JavaDoc of current method) require appropriate meta-data. The easiest way to get this meta-data is using a reflection mechanism that allows the extraction of methods with their parameter types from compiled classes. But neither the names of the parameters nor the usage-description can be reflected. However the WSDL-description contains the names of the parameters, since they are easy to parse the names can be extracted from them. Usage-description is not contained within the WSDL-file and even if a UDDI entry exists for the Web Service the usage-description is not machine-readable.

A lot of modern tools and programming languages support in place documentation. Two examples are JavaDoc and POD (plain old documentation, used by Perl). The idea is to maintain technical representation and the verbose documentation in one document. Many modeling tools also allow adding comment to diagrams and WSDL

has the “documentation” tag which can be embedded at various places inside of a WSDL-document. These pieces of information which are not very useful for computers reading the documents should be kept when generating code. For example, during the generation of Java stubs out of a WSDL-document, the verbal content of the source file should be transferred into JavaDoc.

At the moment, a useful execution of this step is not possible, because there are no specifications which clearly state how this additional documentation should be structured and which kind of information should be placed at which level. Therefore, if a generated document is used in a chain to generate a third document, an initially helpful verbal documentation will be misplaced and mostly useless.

Keeping documentation in separate files and using an external system for documentation might be a good additive, but is not enough to be used as the only documentation source. The risk of an old and outdated documentation is just too high. Developers will modify some files such as the code or the model and will “forget” to update the external documentation. Also, having the needed documentation right at your fingertips eases life a lot and reduces the risk of misinterpretations.

To meet the last requirement (avoid local installations), both solution strategies are web-based and controlled using a browser. Apart from the benefit that end-users do not have to cope with installation and administration procedures this solution enables platform-independent on-demand usage as well as user’s scheduled program-execution even if their local workstation is offline.

3 Query-Console

Many people associate writing queries with writing SQL statements used to query database management systems. But most web-sites do not allow a direct access to their internal databases; their content is queried using predefined HTML-forms which limit the type of queries to a predefined subset defined by the site-owner [2]. Query-results are wrapped into HTML-pages making it difficult to extract them (see section 1).

Today more and more web-sites offer Web Services for automation and query. This reduces the problems of parsing semi-structured web-sites to get their relevant content but it does not solve the problem of how to query the content. The revealed APIs differ from service to service which means that using a new service to execute a query requires understanding the web-site’s API first. Compared to the database-world this would mean that you not only have to understand the schema of a database but also have to understand the operations defined to access the data itself.

Web Service based queries are therefore characterized as programs with special hierarchical access operations defined by the service. Because they are programs they are in contrast to SQL non-descriptive and data-access is hierarchically navigation based like in hierarchical databases instead of (object-) relational because a predefined path (axis) exists to the data.

In this section we want to analyze how to enable non-programmers to efficiently execute queries based on Web Service-calls. To avoid the full complexity of

programming we will confine on the functionality described in section 2 that can be realized as follows.

F1: **Prepare a Web Service for first time usage**: this requires specification of the wsdl-files's URL and an user friendly name to later reference the generated Web Service's stub. The wsdl-file has to be processed by a stub-generator such as wsdl2java from the Apache Axis project. The programming-language used for the implementation has to support late binding to allow dynamical adding of new stubs (classes or libraries) during runtime.

F2: **Create a new object** (instance of a class), such as a Web Service stub
- F2.1: Specify a class an instance should be created from (usually pick one from a list)
- F2.2: List all constructors of this class and their required parameters to invoke a constructor. This requires a mechanism to enable the extraction of interface-definitions. Java-reflection for example allows easy extraction (such as method names and parameter types) from compiled classes. However, names of the parameters and further meta-information (usage description) have to be supplied separately. Unfortunately this meta-data cannot be extracted from UDDI because its entries are not fully machine-readable
- F2.3: Select a constructor to invoke
  - F2.3.1: Supply parameters to invoke constructor – parameters are:
    - F2.3.1.1: references to existing objects (i.e. referenced by their name)
    - F2.3.1.2: newly created objects (see F2)
    - F2.3.1.3: native objects (created implicitly, i.e. a number or string-literal)
- F2.4: Invoke the constructor
- F2.5: If the constructor's invocation succeeds a new object has been created. This can either be used directly as described in F3.2 or a name can be assigned to the object to allow a later usage as a reference (see F3.1)

F3: **Use an object created with the mechanism described in** F2
- F3.1: To use an object it's assigned name (see F1, F2.5) has to be specified (usually by picking one out of a list)
- F3.2: Using meta-data (i.e. generated using reflection-mechanisms, see F2.2) a list is created containing all methods with their required parameters
- F3.3: Select a method to invoke
  - F3.3.1: Supply parameters: these can be specified as described in F2.3.1
- F3.4: Invoke the method
- F3.5: If the call succeeds and the method has a return value it can be used directly as described in F3.2 or a name can be assigned to the object to allow a later usage as a reference (see F3.1)

The functionality described above is nothing new and can be found in many existing implementations that provide online Web Service tutorials like XMethods.com [8]. However these solutions are primarily for educational or explorative Web Service usage and are not intended for frequent usage. Many implementations are designed beginner-friendly which results in long wizard-driven mouse-click-streams. Saving a

query or its results is usually not possible and many implementations support usage of a single Web Service only, meaning that join operations or sub-queries using multiple Web Services are not possible, i.e. if one Web Service returns an address and another Web Service uses the address's zip code to lookup specific data for that region.

We want to introduce a keyboard-controlled approach using a browser-embedded query-console. Keyboard-controlled usage has advantages for frequent users. There is no mouse-keyboard switching between data-entry (usually with a keyboard) and option selection (usually with a mouse), navigation is menu-driven via cursor-keys or shortcuts. This speeds up usage. The downsides are all disadvantages from shells (DOS-prompt) and early terminal applications: they are not intuitive and skills are required. Non-experts have to read manuals perpetually because commands and their parameters are easily forgettable.

The '*what can I type here'* problem in keyboard-controlled environments has already been solved for environments with a defined syntax. Modern IDEs feature source-code-completion by analyzing the current source-code, matching it with syntactical-rules, looking up meta-data for the current context and finally presenting a list of valid method-calls, object-references and other meta-data such as help-texts, etc.

In our scenario the current source-code corresponds to the query, the syntax-rules are well known (see fig. 1 for a simplified version) and relevant context information can be looked up easily using Java-reflection mechanisms. This allows auto-completion for Web Service based queries making skills optional and preventing shell-typical disadvantages as described above.

We realized a prototype with the functionality as described above. Our solution is a keyboard-controlled and browser embedded console that features auto-completion for class- and object-selection (due to too many alternatives for a select-box) and a menu-driven selection mechanism for method- and constructor selection. The prototype has been realized using Java with Apache AXIS. XML-documents are generated from meta-data (reflected class interface information) for the current context. These documents are transformed to HTML using XSLT-transformations. The HTML-output is being processed by a Java-servlet.

4 Automation

The query-mechanism described above already allows faster usage of services with greater flexibility than common HTML-forms. However, several scenarios like automatic execution of such queries or comparing their results is not covered. In this chapter these automation-scenarios are to be addressed. They are:

- **sequences of operations:** execution of multiple queries
  i.e. get price of item I, get price of item P
- **conditional expressions:** if-then-else statements
  i.e. if price of I < price of P then buy I
- **scheduled operations:** automatic execution at predefined times
  i.e. at 12:00 daily: check stocks
- **event-triggered operations:** automatic execution at predefined events
  i.e. on parcel arrival send mail
- **set-based operations:** execution of for/for-all statements
  i.e. for all stocks in watchlist compare prices

Today, these scenarios are out of scope for non-programmers and they are too inconvenient to specify for many programmers, due to having to write and maintain a special program for them.

A solution for programmers is fairly simple: our approach is to bring the IDE-concept to the web by embedding a JavaScript editor with code-completion into a web-browser. Helper-methods ease stub-generation for Web Service usage, scheduling of method-invocation and event handling. Hereby simple automation-scenarios (short programs) can be specified by programmers without the need of prior software-installation/configuration assuming this IDE is offered by a service provider.

For non-programmers this IDE is still unusable due to the need of having to learn a programming-language. This problem is well-known and as old as programming itself, therefore a variety of concepts exist to abstract from this complexity, such as so called *easy-to-learn* functional languages like BASIC, descriptive languages like SQL (originally designed for managers), and visual (do not mix up with symbolic) languages [9]. Mapping a programming-language to graphical symbols (and reverse) is trivial – just visualize the parser's result (usually a tree). The challenge is finding a representation that is useful, such as one that can be understood easily.

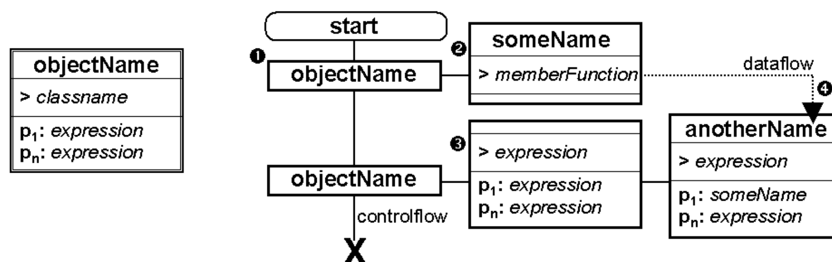

Fig. 2. example of how to map the simple statements from fig. 1 to graphical symbols

To enable simple, spontaneous usage of Web Services we introduce the representation as depicted in fig. 2 that is tailored to our needs as described in section 2. This example visualizes the following sequence of simple JavaScript statements.

```
objectName  = new classname(p1, p2, p3);
someName    = objectName.memberFunction();
anotherName = objectName.someFunction(p1, p2, p3).
              anotherFunction(someName, p2);
```

Instantiation of classes and Web Service stubs is represented using a double-bordered box: *objectName* is the name of the assigned variable, *classname* specifies the name of the class to be instantiated and $p_1,...,p_n$ is the *parameterlist* (see fig. 1) required to call the constructor of the class. Objects are referenced by their name, see ❶ and can be used to call member-functions, see ❷: *memberFunction* defines the name of the method to call and *someName* is the optional name of the variable its return-type is assigned to. If the member-function or alternatively any expression requires parameters they are specified using a list, see ❸. If a parameter is a reference to an object, dataflow is indicated by a dotted line, see ❹. A solid vertical line indicates the control-flow. Graphical symbols for execution scheduling, event-subscription, set-operations and conditional expressions are described in appendix A.

Symbols like these are easier to learn than a programming-language; however the user still does not know how to combine them. Instead of a drag&drop-GUI we recommend the usage of context-sensitive menus known from code-completion. Clicking somewhere opens a menu with *what can I do here* alternatives, such as change a parameter or call a method. This ensures that modifications to the visual-model comply with a syntactically and semantically correct JavaScript program at all times.

Experiments with sample Web Services revealed that even simple services require multiple instantiations and method-calls even for simple use-cases. The context-sensitive menus as described above do not explain the service and API-documentation – if available – is targeted for programmers only. Therefore, we use an automatically generated documentation using graphical symbols that are a simplification of an UML class-diagram to visualize coherence between classes and their methods.

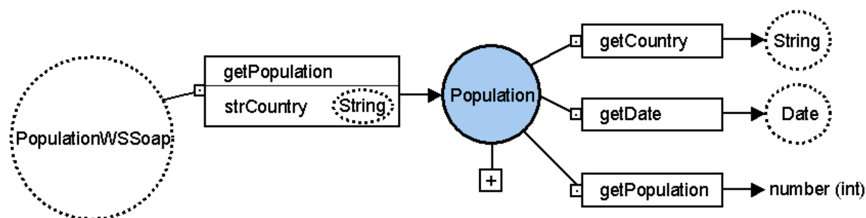

Fig. 3. example of how to document a Web Service using graphical symbols

Figure 3 depicts how an existing Web Service for querying the population of a country is documented using our graphical symbols. Classes (*Population*) are represented using a circle, its methods by boxes (such as *getPopulation*). Required parameters are displayed as a list of parameter-name and type. The return-value of a method is depicted by an arrow. To avoid complex class-diagrams, some methods are

omitted. These can be included by clicking the plus-icon. Another way to keep the diagram small is using references. These are depicted by dotted-circles, such as *Date*. Clicking on *Date* shows the documentation of the date-class.

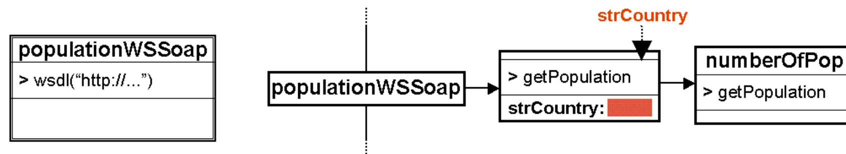

Fig. 4. generated method-call for getPopulation()

This documentation allows the user to interactively explore a Web Service. It is also used to automatically generate method-calls for the user. Clicking on the getPopulation method (the one without parameters) generates a method call as depicted in figure 4. Required parameters (the name of the country) are marked red and can be specified by the user either by entering a constant or the name of an object to reference it. The documentation mechanism described above can also be used for reverse-search if you need to know where a String to specify the name of the country comes from all methods returning this type can be listed. To keep this list short only methods of the current package are shown.

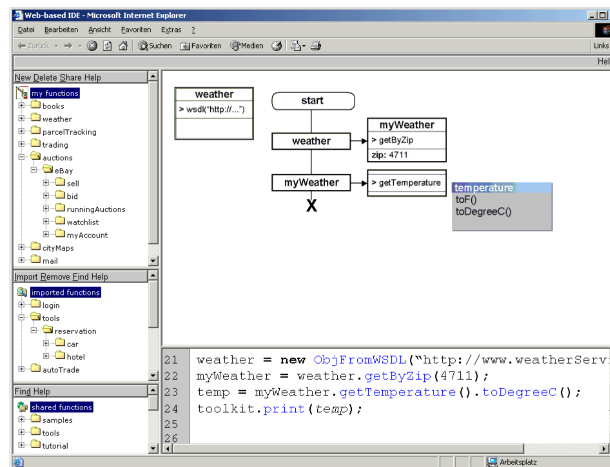

Fig. 5. Screenshot of the web-based IDE (integrated development environment) prototype

We currently realize a prototype (see fig. 5) with the following functionality. On the right side both, the graphical model and the JavaScript code can be edited, changes are synchronized between them automatically. The JavaScript editor uses DHTML, the graphical editor either SVG or generated images. On the left side folder-trees are

displayed for three categories. The first (upper-left) shows the methods created by the user, the last shows shared methods from other users and the middle shows imported methods picked from the shared-list. Shared-methods enable reuse and can be used as a tutorial/reference by beginners.

Security issue

Warning: If you are interested in providing a service as shown above, take precautions that users can only use a safe subset of classes (such as not *java.io.File*), that programs always terminate (limit iterations of for-loops) and that programs can not be used as spam-generators (limit amount of mails sent).

5 Related work

Bringing an IDE to the web does not include a feature which is big, important, or completely new. However, it is an interesting new combination of a selection of known techniques customized for a new target group of experienced web users. In this section we will describe relevant concepts used by existing applications.

HTML-Form based Interfaces: There are a number of generic clients available on the Internet which allow interacting with existing Web Services. However, those interfaces are mainly for programmers that only explore a service. Form based interfaces are not built for a frequent usage and do therefore not offer automation or even saving functionalities.

Dynamic and adaptive composition of services: Some systems support a graphical composition of Web Services. Most often, those systems have their origin in the processes world and are mainly made for modeling processes. Code fragments can be generated out of the process-models. One example is eFlow [10] which supports the dynamic generation of Web Services sequences. Graphical front-ends for BPEL4WS can also support this job.

Graphical Entering of Workflow: There is a variety of tools which support graphical modeling of process representations. These can be activity diagrams as used by the Unified Modeling Language, UML, or Petri Nets which have been used for many years now. Such a graphical support helps especially novice and non experts to do their work and enter the processes in a syntactically correct way.

IDE – Code Completion: Almost all modern integrated development environments such as eclipse offer a feature commonly known as code completion. On entering the name of an object to call one of its methods, a list box containing all methods which match the so far entered name is generated and displayed.

Code Sharing: The generated code and history of executed steps during the use of our concept can often speed up future uses. This benefit can be made available to other developers, too. They can use the generated information and functions like libraries of programming languages which are also often shared by programmers. The same idea has successfully been applied to many open source projects and huge parts of the libraries of many programming languages have been developed this way.

MDA: An important concept of model driven architectures, MDA, is the use of a model as central element. The required code is generated from the information of the graphical model and modified to execute additional jobs. If the architecture of the application has to be changes, only the model is modified and the code is regenerated. Therefore, the documentation is always up to date, because the model illustrates the architecture and is source for the generation of the code at the same time.

6 Conclusions

In this paper we argued that the trend of many web sites to open their API by providing Web Service interfaces could be used not only to satisfy the need of application programmers but also enable end-users that use web sites prevalently to better support their usage scenarios by enabling efficient query and automation mechanisms.

This group of ‘experienced web users’, as we call them, wants to perform actions on web sites which are not well supported by the site’s graphical front ends. Unfortunately, the Web Service interfaces provided by many web sites today do also not meet the requirements of this group of users. Although those interfaces provide the needed functionality for extended usage, two restrictions impede their application: First, even experienced web users do not want to use a fully fledged programming environment. Since it has been designed for general programming tasks it is much too complex for the simple query and automation tasks this kind of users want to perform. Something more simple is needed that enables a steep learning curve with short time to productive usage. Second, a runtime environment is required at the users workstation. Most of the experienced web users do neither have the skills nor the resources nor do they want to spend the efforts of installing and operating such an environment. For this reasons we have proposed to provide such an environment as a web-based service and to reduce the functionality of this service to the core programming elements.

In section 2 we have described the requirements such a service has to fulfill: It has to abstract from Web Service complexity (SOAP, HTTP-connection, bindings, endpoints) and programming-complexity (classes, functions, control-structures). By this we ensure that it is easy to use. Further we introduced a reduced subset of programming elements which we think is sufficient for supporting the tasks of a experienced web user. Hence, the user does not need to cope with the complexity of programming with Web Services in a conventional IDE.

In the following sections 3 and 4 we presented two typical usage scenarios and how they are supported by the web-based environment we propose. To meet the last

requirement introduced in section 2 (avoid local installations) our solution for both scenarios is web-based and uses a browser as a front end.
The first scenario is to perform queries which may span several Web Services. Well-known query languages, e.g. SQL, are neither able to cope with the service-oriented interfaces nor with heterogeneous data sources nor with semi-structured data. Nevertheless, in our opinion such queries can be expressed in a function-oriented programming style with nested service calls and this way of expressing queries can easily be mapped to a web-based query console. Since this is only a subset of the second scenario we described the web-based programming environment in section 4.

The second scenario extends the query console by additional programming elements, e.g. conditional execution. In section 4 we have shown how a steep learning curve is made possible through a mixture of graphical and textual programming. In addition, the meta data available for Web Services enables us to support the user by context-sensitive suggestions about what to do next. In an example we demonstrated that even non-programmers can use this interface and become productive fast.

Altogether, the main messages of our paper are:

- There exists a large group of users which is not supported well so far, i.e. there is a reason to provide such a service.
- It is possible to provide a service which does meet the requirements and restrictions of this group of users, i.e. there is no reason not to implement such a service.
- The Web Service technology available today is sufficient to implement this kind of service, i.e. there is no reason not to start now.

Nevertheless, one question remains open: Why is no such service available?

References

1. Konopnicki, D., Shmueli, O.: W3QS: A Query System for the World Wide Web. 21st Conference on Very Large Databases, Zurich, Switzerland, 54—65, 1995
2. Levy, A. Y., Rajaraman, A., Ordille, J. J.: Querying Heterogeneous Information Sources Using Source Descriptions. In proceedings of the 22nd VLDB Conference, Bombay, India, 1996
3. Ashish, N., Knoblock, C. A.: Semi-Automatic Wrapper Generation for Internet Information Sources. Conference on Cooperative Information Systems (1997) 160-169
4. eBay news
   http://presse.ebay.de
5. Axis: Apache Web Services Project
   http://ws.apache.org/axis/
6. JavaScript ECMA Standard: ECMA-262
   http://www.ecma-international.org
7. Extended Backus-Naur Form (EBNF), ISO/IEC 14977 (1996)
8. XMethods Web Services collection
   http://xmethods.com
9. Burnett, M. M.: Visual Language Research Bibliography
   http://web.engr.oregonstate.edu/~burnett/vpl.html
10. Casati, F. , Shan, M.-C: Dynamic and Adaptive composition of E-services. Information Systems (2001) 26:143–163

Appendix A Graphical symbols

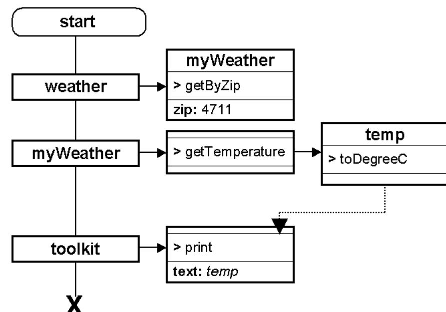

Fig. A.1 sequence

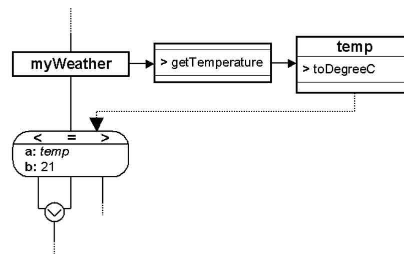

Fig. A.2 simple conditional expression

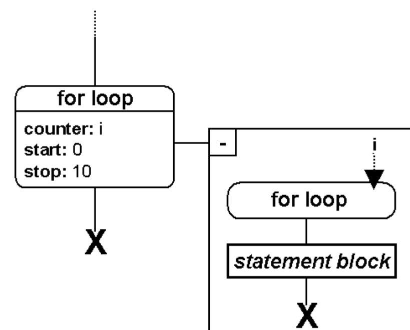

Fig. A.3 for-loop

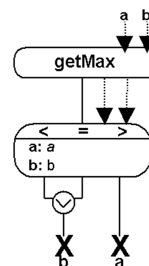

Fig. A.4 method declaration with parameters and return-type

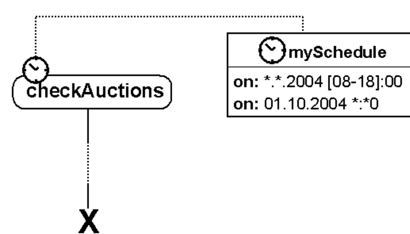

Fig. A.5 execution scheduling

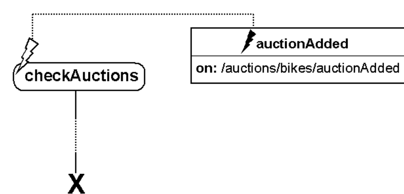

Fig. A.6 event subscription using a topic-path expression from web-services notification