

Wrapping Legacy Software for Reuse in a SOA

Harry M. Sneed
AneCon GmbH, Wien
E-mail: Harry.Sneed@anecon.com

Abstract: Legacy programs, i. e. programs which have been developed with an outdated technology make-up for the vast majority of programs in many user application environments. It is these programs which actually run the information systems of the business world. Moving to a new technology such as service oriented architecture is impossible without taking these programs along. This contribution presents a tool supported method for achieving that goal. Legacy code is wrapped behind an XML shell which allows individual functions within the programs, to be offered as web services to any external user. By means of this wrapping technology, a significant part of the company software assets can be preserved within the framework of a service oriented architecture.

Keywords: Service Oriented Architecture, legacy software, system integration, wrapping, web services, XML, WSDL

1 Legacy Software

Legacy programs can be divided into three basic categories in regard to the degree of dependence on their environment.

- programs which are not dependent on their environment,
- programs which are partially dependent on their environment,
- programs which are totally dependent on their environment [1]

The first category includes programs written in the conventional languages Fortran, COBOL, and C/C++. These programs can be readily reused in any environment which has a compiler to compile them. The second category encompasses programs written in a language which uses run time or link time functions. To this category belong PL/I, Smalltalk, and Forté. These programs can be reused in another environment, but only if their runtime routines are substituted by compilable modules written in the host language itself.

The third category consists of all of the 4th generation language programs, such as ADS-Online, Natural, CSP and Oracle Frames, requiring a specific environment to run in. Such software can not be reused in another environment. It is environment dependent. Therefore, the only way to reuse these programs is to keep them in their native environment and to build runtime links to that environment.

One can summarize from this observation, that the more elementary a programming language is, i.e. the less bells and whistles it has, the easier it is to reuse. This is something that managers should consider when choosing a development technology. They must choose between short range productivity and long range reusability and portability. [2]

2 Service oriented architecture

The main goal of a service oriented architecture is to make the software functionality available to all who need it and who are authorized to use it. Not only that, they should also be able to combine the functionality in any way they deem appropriate, i.e. to embed it as steps in their business processes. By invoking the methods offered by the service architecture they can fulfill the functions referred to within their business process language – BPEL – procedures without having to code and test them themselves. The price for that is modeling the evolution of legacy systems to the WSDL interface, setting the parameters according to the interface specification. [3]

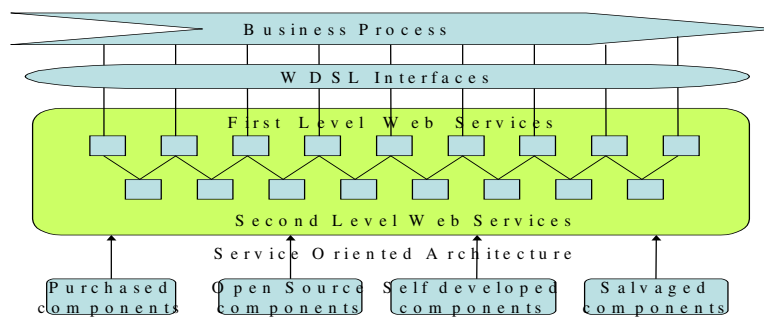


Figure 1: Sources of Web Services

Seen from this perspective, the service oriented architecture is similar to a giant subroutine library, with the difference that the user must not copy it onto his computer, compile it and link it with his own programs. He can access it at execution time via the internet. This way he is assured of always using the latest versions and does not have to worry about continual updates. The price he has to pay is twofold.

First, there is a performance price. Sending subroutine calls with long parameter lists across the networks requires time. The more services that are invoked and the more parameters passed, the longer will be the transmission times. An XSLT-based transformation as a single WSDL interface can take up to 500 milliseconds. Therefore, it is advisable to minimize both the number of calls and the number of parameters. [4]

The second price is that of complexity, resulting in more effort to use the service. The greater the functionality of the individual web service, the more complex is the interface to it. Not only will the service method require more input parameters, it will also produce more results, all of which have to be specified in the interface. The web service

call becomes increasingly complicated, with layers of nested data and parameter lists until it becomes more and more like a program itself. At some point, one must ask if it were not simpler to code the service oneself rather than spending so much time and effort to build up a WSDL interface. On top of that, complex interfaces require many test cases to validate and are error prone. It could well be, that it requires more effort to test the web service interface than it does to program the service oneself. [5]

The conclusion here is that web services must remain simple in order to be readily usable and to be built into the user's business processes with a minimum of effort and a maximum of performance. In this respect service oriented architectures are similar to other technological solutions of the past. They can reduce the development and maintenance effort of the user by offering ready made software functionality, but they extract a price in comprehending and testing as well as in performance. The goal of the SOA designer should be to keep these costs as low as possible by offering a large number of simple elementary services with interfaces that are easy to serve and to test. The designer should strive to minimize the number of input parameters and to return no more results than necessary for any one invocation. In other words, the web services should be limited and their interfaces as narrow as possible. [6]

Having set these design goals the question then comes up as to where the web services come from, i.e. how are they supplied. As with all standard components there are four basic sources:

- they can be bought from a web service vendor,
- they can be borrowed from the open source community,
- they can be developed individually or
- they can be taken from existing applications.

There are several vendors which now offer off the shelf web services including the major software producers Microsoft, IBM, SUN and SAP. [7] For a user company bent on building up a service oriented architecture it is advisable to consult the catalogues of these vendors and to purchase those services which fit their requirements. Of course, the user then becomes dependent on the vendor to maintain the services purchased, but this has always been the price of standard solutions.

The same applies to the open source community. Here too scores of individual software services are available and the number is constantly increasing. The draw back here is that the user must maintain the services himself, i.e., he is dependent on the ability of his own programming staff to comprehend the foreign code and to adjust it to his local needs. That requires knowledge, time and tools. The comprehension problem with web services is no less than with any other foreign software components. [8]

Developing the web services oneself is always an alternative. Large user organizations can set up a special development group to produce such common services, just as was the case with the common subroutine libraries and the common class libraries. There is no real difference here, only the interface languages changes. Instead of processing parameter lists or linkage sections, the developers now have to deal with WSDL schemas. Besides developing the services, the user also has to test them. This could be

an obstacle to many users who are not versed in testing technology. Testing a WSDL interface is more demanding than testing a GUI. The GUI can be created and validated visually at test time. The WSDL is basically invisible. The interface has to be generated by a program and sent to the target service via a middleware product. The results have to be received by a program, recorded and validated against the expected results. All of this requires tools and experienced testers, something most user organizations do not have. [9]

The fourth and final source of web services is the existing software. Every user organization which has been using information technology for any length of time will have accumulated a significant amount of legacy software. Some of this software will be tightly coupled to the environment for which it was developed, in particular the presentation software which is presenting maps or GUIs. Other parts of the software will be tightly coupled with a particular database system, namely the data access software. In so far as the same database is used for web applications, this software can be reused. A third and significant part of the application software will be devoted to processing the business logic, which have to be mined out of the existing code. [10]

Locating and salvaging such business-oriented software is referred to as code mining or software recycling. It is similar to salvaging valuable building blocks from the ruins of an old building in order to reuse them in a new edifice. The technology for doing this has been available since the mid 1990's and has been well covered in the reengineering literature. [11] What is new here, is the attempt to reuse these old code blocks as web services in a service oriented architecture. The technology for doing that is the subject of this paper.

The advantage of reusing one's own code as opposed to the other sources of web services is obvious. Using off the shelf web services is inexpensive, but such services will seldom fulfill the exact requirements of any particular user organization. At best they can be used to supplement the user's own unique services. Besides, since they do not belong to the user, the user is dependent upon the supplier to maintain and evolve them. Developing new web services from scratch is an enticing alternative, especially for developers eager to experiment with the new technology, but one always underestimates the effort required to test new services and to bring them up to a quality standard where they can be relied upon. [12]

The costs of developing high quality web services are for many users simply too high. Even large organizations cannot or will not afford it. So, developing one's own web services is a long range goal which can be achieved in the course of many years, but it is not something that can be achieved within a short run. That leaves the user with a choice of either retrofitting his business processes to accommodate the standard web services available or reusing his existing software which was built from the beginning to fit his particular business processes.

3 Candidates for web services

According to the marketing director of the Software A.G., the core functionality of most public administration offices is buried deep in their existing application software. [13] It is futile to attempt to reproduce it in another form. The only practical solution is to wrap it and make it available as a universal public service. What is not mentioned here is that this functionality must first be salvaged and brought up from the depth in order to reuse it. E-government has become a prime candidate for the techniques of software recycling.

The same applies to the functionality in business administration. There are scores of individual company specific tasks unique to every enterprise. Typical examples are the modes of payment, the granting of credit, the computation of interest rates and the handling of privileged customers. Conventional business processes are full of such user specific solutions which have been evolved and tuned over many years. They are an essential part of the company operation.

The problem is that this customized business logic is not readily accessible. It must be identified through various mining techniques as pointed out by Aversano and Tortorella in their work on salvaging public administration systems for eGovernment applications. [14] One is fortunate to even find a particular business function in a single module. In a bank application reengineered by the author the opening of an account was scattered across five different components, thus violating the principle of locality of reference. None the less, the functionality was present and distinguishable from the other functions around it, even though they shared some common code.

In reusing existing code, the first task is to identify the candidates for a web service. User organizations wanting to move to a service oriented architecture must make a portfolio analysis of their existing applications and to list out the essential application operations. In doing so, it will be necessary to break the complex operations down into elementary operations which are self contained logical units. In an order entry application the basic operations might be

- confirming the credibility of the customer,
- reducing the stock,
- billing the customer and
- handling back orders.

The elementary business operations such as reducing the stock can be reused directly in another context. The billing of the customer is, however, a too complex operation, which has to be further broken down into

- aggregating the billing items
- computing the sales tax
- obtaining the customer address data
- producing the bill
- dispatching the bill.

These elementary operations are candidates for web services. They have a limited number of arguments, i.e. input variables and a single compound result. As such they can be fitted conveniently into any business process.

The second step is to assess the business value of these reuse candidates. Ben-Menachem suggests a classification scheme, which involves categorizing the items, calculating each item's value and assigning a value coefficient. Existing software components can be categorized by language, purpose, type and criticality. Calculating an item's value is based on cost analysis of the development costs, the maintenance costs, the estimated replacement costs and the annual business value contributed by that item. In assigning a value coefficient, the business value over a three year period minus the maintenance costs is divided by the costs of replacement, i.e. redevelopment of that item.

Business_Value – Maint_Costs

Cost of Replacement

The reusable code items are then ranked based on their value coefficient. The ranking shows which business operations have the highest potential as web services. [15]

The key to defining suitable web services is the granularity of the services. They must be broken down to a level of granularity where each service performs a single well defined transformation or computation upon a limited set of parameters to provide a singular result. Furthermore, they should be stateless. A web service should not be required to maintain its own state. If a web service is invoked a second time, the user cannot expect for it to remember what the result of the last invocation was. The preservation of persistent objects such as the article data in the order entry example is the responsibility of the overlying business process. Prior to invoking the web service "Stock – reduction", the article data would have to be retrieved from the article data base and afterwards restored in the altered state.

It is true that the business processes will become over burdened with the many web service invocations, but this way it will not be necessary to constantly change the web services. One has to choose here between two evils. Either the control logic is included in the web services or it is contained within the business process. The web services should remain as constant as possible. All changes should be made at the business process level, by changing the order of web service invocation, by altering the parameters or by invoking additional services.

The essence of a successful service oriented architecture is according to Prof. Scheer, the father of the ARIS business process modelling system, flexibility. [16] The architecture must be adaptable to changes in the business environment with a minimum of effort and time. This goal can only be achieved if the underlying services are kept at a low level of complexity. The complexity should be built into the business processes where it can be more readily managed.

4 Creating web services from legacy code

There are three basic steps required to create web services from legacy code.

- salvaging the legacy code
- wrapping the salvaged code and
- making the code available as a web service.

These three steps will be described in the following sections.

4.1 Salvaging the legacy code

To be able to salvage code from an existing legacy code base it is first necessary to locate that code and to determine if it is worth reusing. It is not a problem to analyze and evaluate the code of a few small programs. That can be done by any programmer familiar with the code using a comfortable text editor. It is quite different to analyze several hundred large programs in search of a few reusable blocks of code. Here too a domain expert is required, but he must be supported by automated reverse engineering tools.

The key to discovering the business operations are the results which they produce. By identifying the variables which are returned by the functions processing the business operations one can also identify the functions. If the programs were structured in such a way that the business functions were assigned to one code block such as a function in C, an internal procedure in PL/I, a subroutine in Natural, or a paragraph in COBOL, then this task would be simple, but they seldom are. More likely a business function is scattered throughout several blocks of code in several modules. On the other hand, one block of code may be processing several business functions. So there is a n:m relationship between code blocks and business operations.

By making a data flow analysis based on the final results, it is possible to trace the result back through all of the statements which contributed toward producing it. Once the statements are identified, then it is possible to locate in what code units, i.e. procedures, paragraphs, subroutines, etc., they are in. Only those units are then copied from the original source together with the variables they refer to. This technique is known as "Code stripping". It was originally used in testing to verify the path leading to a given output. However, it applies equally well to the task of extracting elementary business operations. [17]

The essential point here is that a business operation is defined as an algorithm for computing a given result. This result may be a yes or no answer for instance to determine whether a customer is a VOP customer or not. There may also be several results produced in different places, for instance an order entry process which not only confirms the fulfillment of that order, but also updates the amount of the item ordered and generates a billing position and a dispatch order. To extract the code for processing an order, it would be necessary to identify all of the data objects affected as a result of that processing.

The next step after identifying the code of a business operation, is to extract that code and to reassemble it as a separate module with its own interface. This is done by copying the impacted code units into a common framework and by placing all of the data objects they refer to into a common data interface. In C the interfaces are parameters of the type structure, in Cobol the objects are level 1 items in the linkage section, in PL/I the objects are based data structures with the pointers to them as parameters to the main procedure. The end result will be, in all cases, a subroutine with a call interface. The original input arguments will be input parameters and the original output arguments output parameters. In this respect the business logic code will have been disconnected from the original user interface and made into a self contained subprogram. This is a prerequisite to wrapping it. [18]

A useful byproduct of this code reengineering process is a documentation of the existing business operations. For each data result of a particular use case, the conditions, assignments, computations and IO operations required to produce that outcome are presented in the form of a data flow tree. The final result state builds the root node of the tree. The other nodes are the arguments and intermediate variables which flow into that final result. The branches of the tree represent the state transitions, which are triggered by conditional statements such as if, case- and loop statements. Since a business operation is an intersection of control and data flow, it is necessary to depict both perspectives.

With the aide of these diagrams, it becomes possible for the user to decide whether an existing operation, implemented within a legacy system is worth reusing as a public function in a service oriented architecture. The decision requires a full comprehension of the current operation as well as a notion of its economic value. If an operation has a high economic value and a low level of implementation, it may be better to rewrite it again as a separate entity. Operations with an acceptable implementation and a medium to high economic value are the prime candidates for reuse.

4.2. Wrapping the legacy code

Once a business operation has been located, documented and found worthy of reuse, the next step is to wrap it. The goal of the wrapping process is to provide the component extracted from the legacy code with a WSDL interface. The technique used is to transform each entry into a method and to transform each parameter into an XML data element. The data structures will become complex elements with one or more sub-elements. The methods will have their arguments and results as references to the data element descriptions. Both the methods and the parameters will be built into an XML schema.

(see Sample 1: Input Interface Schema)

The tool SoftWrap has been developed to automate this transformation for the languages PL/I, COBOL, and C/C++. Besides, creating the WSDL interface description, it also enhances the wrapped component with two additional modules. One module is for parsing the incoming message and extracting the data from it. The extracted values are

then assigned to the corresponding arguments in the wrapped component. The other module is for creating the return message from the results produced by the wrapped component. In this way an elementary operation can be reused as a web service without having to change the code. The two generated subroutines act as a bridge between the WSDL interface and the call interface of the original code.

(see Sample 2: Output Interface Schema)

In PL/I these two subroutines are implemented as external procedures, in COBOL as subprograms and in CPP as separate classes. The purpose is to avoid manual manipulation of the legacy code, since manual intervention is not only costly, but also error prone. To be effective wrapping must be automated. This simple fact has been acknowledged by the major EAI vendors, who offer wrapping solutions for entire programs and databases. [19] The approach proposed here differs from these other commercial solutions, in that it deals not with the original online transactions, screens and programs, but with artificially constructed segments of code extracted from the original programs for the sole purpose of being used as web services.

The motivation of the EAI vendors is to enable the user to link together diverse applications via a hub software. The goal of a service oriented architecture is to replace the existing applications altogether by a series of fine grained components which the user can assemble into dynamic applications on demand. This difference in strategy makes it impossible to reuse the old programs as they are. Nevertheless, the logic they contain can be reused, but only if it is extracted from the original context and transformed into another web compatible one.

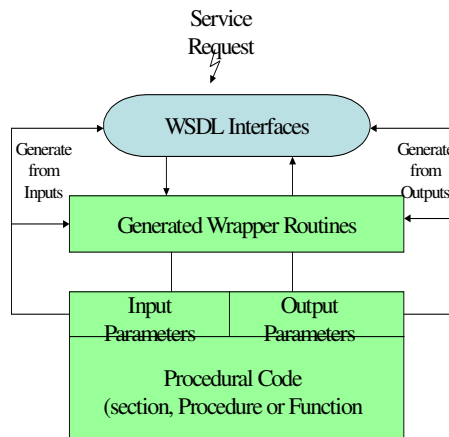


Figure 3: Wrapping Salvaged Components

4.3 Linking the web services to the business processes

The third and final step in creating web services from legacy code is to link the web services to the overlying business processes. This is made by means of a proxy component. The business process actually invokes the proxy which is available in the same address space as the process definition. Like in CORBA the proxy checks the parameters and generates the WSDL interface which is then dispatched by some message service such as MQ-Series to the application server. This proxy technique has been referred to by Aversano and Canfora in a previous paper discussing the wrapping of legacy application for web access. [20]

On the application server there is a scheduler, which receives the incoming message, determines which web-service is to be performed and forwards the WSDL contents to that particular service, in our case the wrapped legacy code. The wrapper of the code parses the XML input data and moves the values to the appropriate addresses in the wrapped component.

Once the wrapped component has been executed, its result is transformed by the wrapper into an XML output data structure, which goes back to the scheduler to be transmitted back to the web client. In this way the business process can be executed on any client anywhere and still is able to access the legacy functions on the original application server. [21]

5 Case Study of a legacy web service

The example selected to demonstrate the integration of legacy components into a service oriented architecture is that of a calendar function extracted from the legacy software of a Swiss bank. The function accepts a date, a language code and an adjustment parameter. It returns the day of the week in the language indicated by the language code, adjusted either to the left or the right of the text line depending on the text adjustment parameter. This function was originally implemented in Assembler and was later converted to COBOL within the scope of a major migration project. Later it was decided to reuse it as a service in a web architecture.

5.1 Extracting Business Operations

Within the COBOL code, the function was a separate section, so the first step in wrapping it was to extract it from the source text of the program and to place it in a separately compilable module with its own data division and linkage section. The result of the business operation for computing the name of the week-day-name, the variable referred to as DAY-NAME, was used to locate the code that computed it. The first reference to DAY-NAME was found in an initialization paragraph where it was set to spaces, the second reference in the leap year processing paragraph where it was set to a default value and the third reference in a paragraph which prepared the output.

A trace of the input arguments showed that they were only referenced in these three sections of code. Therefore, these three paragraphs were cut out of the original procedure

division and placed in a new procedure. The variables they used – the arguments, the result, the intermediate variables and the constants – for instance the table of weekdays in the three Swiss languages were scattered throughout the original Data Division. They were collected together to form a new Working-Storage section. The three input parameters and the one output parameter were placed in the Linkage-Section. The result of this reengineering process was a new COBOL module with its own Data-Division, working-storage, linkage-section and procedural code.

The procedural code consisted of the three paragraphs extracted from the original program – initialization, leap year handling and output setting. All of these reengineering steps are performed automatically by the SoftWrap tool. The user need only identify the input and output variables of the operations he wants to extract. The tool SoftWrap has been described in previous reports. [22]

5.2 Wrapping Business Operations

The second step, also performed by SoftWrap, is to wrap the new module extracted from the old code. This entailed generating a driver module which reads the input parameters – date, language and adjustment – from a WSDL input file and writes the result – the day of the week or error message – into a WSDL output file. In addition, SoftWrap also produces an XML schema both for the input and the output file.

The schema describes the structure and the attributes of the parameters. Besides the usual XML attributes such as name, type and occurrence, each data element has some additional attributes necessary to convert the XML data types into COBOL data types and to set them into or to receive them from the corresponding COBOL address. For example, it is necessary to know that the DAY-NAME is at the 10th position in the parameter string, that it is 10 bytes long and that it is a character field.

The generated driver module parses the schema in order to interpret the incoming WSDL message with the date, the language code and the adjustment code, and to create an outgoing WSDL message with the day of the week or an error message.

In this way, the calendar function has been wrapped. It can be invoked via a WSDL interface with the three input parameters and the name of the method to be executed. When the calendar method has been executed, it will then return the result to the business process requesting it.

5.3 Integrating Business Operations

It remains now to implement the business operation by invoking it from a business process. The language for implementing business processes is BPEL4WS. BPEL4WS establishes links to partners, defines the link types, declares the parameters to be sent and the results to be received, and invokes the web services. [23] The sample process script sets the parameters for the date, language and adjustment and then identifies the service by name as depicted in Sample 3. (see Sample 3: User Business Process)

A WSDL interface is generated by the BPEL interpreter with the input and output parameters, the function name, the messages, the port type with its input and output messages and finally, the SOAP binding description. This is all created from a standard template so that the author of the business process has nothing to do with it. He only sees the result which is returned, namely the day of the week. It is important that all of these technical details be hidden from the designer of the business processes to a great an extent as possible. (see Samples 3 & 4.: WSDL Messages)

6 Conclusion

Web Services offered within the framework of a Service Oriented Architecture promise to make applications more flexible, easier to compose and cheaper to develop. [24] In this paper it has been demonstrated how legacy code can be reused to help construct such web services. It would be unwise to ignore the vast amount of proven legacy software available within corporations and public administrations, when migrating to a service oriented architecture. Before developing or purchasing new service components, one should try to reuse the old ones. The technology for doing so is available. The approach presented in this paper is only one of several similar ones developed at the RCOST research institute in Benevento, at the University of Bari and at the IBM Research Institute in Toronto. [25] It has been proven there and elsewhere that specific business functions can be extracted from existing programs, wrapped and integrated into an eBusiness application framework. Doing so avoids the cost and risks of having to develop them from scratch. The savings is the difference between the cost of salvaging and wrapping the legacy functions as opposed to the cost of designing, coding and testing. It promises to be significant.

The approach described in this paper has been applied successfully for the integration of both COBOL and C++ programs.[26] There exists a PI/I version, but it has yet to be proven in practice. The main problem has turned out to be reentrancy. The state of the data contained within a wrapped web service is that of the last caller. Thus, if different processes are using the same service, their data will be mixed. One solution is to store the internal data state in a temporary database under the id of that user. The other solution is to have a scheduler. Both solutions have advantages and disadvantages. However, this is not a problem specific to wrapped legacy code, but to all web services. It has to be solved in order for this technology to be accepted.

References

- [1] Miller, H.: Reengineering Legacy Software Systems, Digital Press, Boston, 1998, p. 13
- [2] Warren, Ian: The Renaissance of Legacy Systems, Springer Pub., London, 1999, p. 2
- [3] Lavery,J./Boldyreff,B./Ling,B./Allison,C.: "Modelling the evolution of legacy systems to Web-based systems", Journal of Software Maintenance and Evolution,Vol.16, Nr. 1,2004, p.5
- [4] Litoiu, M.: "Migrating to Web Services – a performance engineering approach" Journal of Software Maintenance and Evolution, Vol. 16, Nr. 1, 2004, p. 51

- [5] Tonella, P./ Ricca, F.: Statistical Testing of Web Applications, *Journal of Software Maintenance and Evolution*, Vol. 16, Nr. 1, 2004, p. 103
- [6] Krafzig,D./Banke,K./Slama,D: Enterprise SOA, The Coad Series, Prentice-Hall Pub., Upper Saddle River, N.J., 2004, p. 6
- [7] Egyed, A./Müller, H./Perry, D.: “Integrating COTS into the Development Process”, *IEEE Software*, July, 2005, p. 16
- [8] Gold, N./Bennett, K.: “Program Comprehension for Web Services”, *Proc. of 12th IWPC*, IEEE Computer Society, Bari, June, 2004, p. 151
- [9] Sneed, H.: “Testing a Web Application”, *Proc. of 6th Web Site Evolution*, IEEE Computer Society, Chicago, 2004, p. 3
- [10] Bovenzi, D./ Canfora, G./ Fasolina, A.: “Enabling Legacy System Accessibility by Web Heterogeneous Clients”, *Proc. of 7th CSMR-2003*, IEEE Computer Society Press, Benevento, March, 2003, p. 73
- [11] Bodhuin, T./Guardabascio, E./ Totorella, M.: “Migrating COBOL Systems to the WEB”, *Proc. of 9th WCRE-2002*, IEEE Computer Society, Richmond Va., Nov. 2002, p. 329
- [12] Tilley, S./ Gerdes, J./ Hamilton, T./ Huang, S./ Müller, H./Smith, D./Wong, K.: “ On the business value and technical challenges of adapting Web services”, *Journal of Software Maintenance and Evolution*, Vol. 16, Nr. 1, 2004, p. 31
- [13] Vorsamer, A.: “Java Tools help with Host-Integration”, in *Computer Zeitung*, Nr. 32, August, 2005, s. 19
- [14] Aversano, L./ Tortorella, M.: “An Assessment Strategy for identifying legacy system evolution requirements in eBusiness Context”, *Journal of Software Maintenance and Evolution*, Vol. 16, Nr. 4, 2004, p. 255
- [15] Ben-Menachem, M.: “Web Metadata Standards – Observations and Prescriptions”, *IEEE Software*, February, 2005, p. 78
- [16] Scheer, A.W.: “Where will the Program Code remain”, in *Computerwoche*, Nr. 15, April, 2005, p. 22
- [17] Sneed, H./ Erdoes, K.: “Extracting Business Rules from Source Code”, *Proc. of 4th IWPC-1996*, IEEE Computer Society, Berlin, March 1996, p. 240
- [18] Sneed, H.: Extracting Business Logic from existing COBOL Programs as a Basis for Reuse”, *Proc. of 9th IWPC-2001*, IEEE Computer Society, Toronto, May, 2001, p. 167
- [19] Hasselbrink, W.: “Information System Integration” *Comm. Of ACM*, Vol. 43, No. 6, June 2000, p. 33
- [20] Aversano, L./Canfora, G./Cimitile, A./ DeLucia, A.: “Migrating Legacy Systems to the Web” *Proc of 5th CSMR*, IEEE Computer Society Press, Lisbon, March 2001, p. 148
- [21] Sneed, H.: “Wrapping Legacy COBOL Programs behind an XML Interface” *Proc. of 8th WCRE-2001*, IEEE Computer Society Press, Stuttgart, Oct. 2001, p. 189
- [22] Sneed, H.: “Program Interface Reengineering for Wrapping”, *Proc. of 4th WCRE*, IEEE Computer Society Press, Amsterdam, Oct. 1997, p. 206
- [23] Juric, M./Mathew, B./ Poornachandra, S.: *Business Process Execution Language for Web Services*, Packt Pub., Birmingham, U.K., 2004, p. 17
- [24] Jones, S.: “Towards an acceptable Definition of Services”, *IEEE Software*, May 2005, p. 87
- [25] Zou, Y./ Lau, T./ Kontogiannis, K.: “Model Driven Business Process Recovery”, *Proc. of 11th WCRE-2004*, IEEE Computer Society Press, Delft, N.L. Nov. 2004, p. 224
- [26] Sneed, H./ Sneed, S.: *Web-based System Integration*, Vieweg Verlag, Wiesbaden, 2004, p. 257

Sample 1 : Input Schema generated from COBOL Module

```
<schema name = "xm059i"
  xmlns= "XSDCOB">
  <XSDCOB:complexType type = "#file" name = "xm059i"
    content = "eltOnly" model = "closed">
    <XSDCOB:complexType type = "#params" name = "DayofWeekRequest"
      content = "eltOnly" model = "closed" level = "02"
      occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "unbounded">
      <XSDCOB:element type = "#dec" name = "DAY"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0001" lng = "0002"
        pic = "99" usage = "DISPLAY"/>
      <XSDCOB:element type = "#dec" name = "MONTH"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0003" lng = "0002"
        pic = "99" usage = "DISPLAY"/>
      <XSDCOB:element type = "#dec" name = "YEAR"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0005" lng = "0004"
        pic = "9999" usage = "DISPLAY"/>
      <XSDCOB:element type = "#dec" name = "LANGUAGE"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0009" lng = "0001"
        pic = "9" usage = "DISPLAY"/>
      <XSDCOB:element type = "#char" name = "ALIGNMENT"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0010" lng = "0001"
        pic = "X" usage = "DISPLAY"/>
    </XSDCOB:complexType>
  </XSDCOB:complexType>
```

Sample 2 : Output Schema generated from COBOL Module

```
<schema name = "xm059o"
  xmlns= "XSDCOB">
  <XSDCOB:complexType type = "#file" name = "xm059o"
    content = "eltOnly" model = "closed">
    <XSDCOB:complexType type = "#params" name = "DayofWeekResponse"
      content = "eltOnly" model = "closed" level = "02"
      occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "unbounded">
      .....<XSDCOB:element type = "#char" name = "RETURN-CODE"
        content = "TextOnly" model = "closed" level = "02"
        occurs = "1" minOccurs = "0001" maxOccurs = "0001"
        pos = "0000" lng = "0002"
        pic = "XX" usage = "DISPLAY"/>
      <XSDCOB:element type = "#char" name = "DAYOFWEEK"
        content = "TextOnly" model = "closed" level = "03"
        occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001"
        pos = "0011" lng = "0010"
        pic = "X(10)" usage = "DISPLAY"/>
    </XSDCOB:complexType>
  </XSDCOB:complexType>
</schema>
```

Sample 3 : User Business Process in BPEL4WS (Fragment of Code)

```
<process name = "Calender"
  xmlns:calender = "http://anecon.com/sneed/sample/" >
  <partnerLinks>
  <PartnerLink name = "CalenderUser"
```

```

        partnerLinkType = "calender:User"
        myRole = "Provider"
        partnerRole = "User" />
</partnerLinks>
<variables>
  <!-- inputs for Calender Functions -->
  <variable name = "Day" messageType = "calender:DayofWeekRequest"/>
  <variable name = "Month" messageType = "calender:DayofWeekRequest"/>
  <variable name = "Year" messageType = "calender:DayofWeekRequest"/>
  <variable name = "Language" messageType = "calender:DayofWeekRequest"/>
  <variable name = "Alignment" messageType = "calender:DayofWeekRequest"/>
  <!-- outputs for Calender Functions -->
  <variable name = "ResponseCode" messageType = "calender:DayofWeekResponse"/>
  <variable name = "DayofWeek" messageType = "calender:DayofWeekResponse"/>
</variables>
<assign>
  <copy>
    <from variable = "Current_Day" part = "Date" />
    <to variable = "Day" part = "DayofWeekRequest" />
  </copy>
  <copy>
    <from variable = "Current_Month" part = "Date" />
    <to variable = "Month" part = "DayofWeekRequest" />
  </copy>
  <copy>
    <from variable = "Current_Year" part = "Date" />
    <to variable = "Year" part = "DayofWeekRequest" />
  </copy>
</assign>
<!-- call Calender Service to provide Day -->
<invoke partnerLink = "CalenderUser"
  portType = "CalenderStatusPT"
  operation = "GetDayofWeek"
  inputVariable = "DayofWeekRequest"
  output Variable = "DayofWeekResponse" />
<assign>
  <copy>
    <from variable = "DayofWeek" part = "DayofWeekResponse" />
    <to variable = "WeekDay" part = "LetterHeader" />
  </copy>
</assign>
</process>

```

Sample 4 : Input Message from User Business Process

```

<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE "xm059i" SYSTEM "xm059i.xsd">
<xm059i>
  <DayofWeekRequest>
    <DAY>12</DAY>
    <MONTH>10</MONTH>
    <YEAR>1977</YEAR>
    <LANGUAGE>3</LANGUAGE>
    <ALIGNMENT>1</ALIGNMENT>
  </DayofWeekRequest>
</xm059i>

```

Sample 5 : Output Message to User Business Process

```

<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!--DOCTYPE XM0590 SYSTEM "XM0590.xsd"-->
<XM0590>
  <DayofWeekResponse>
    <RETURN-CODE>00</RETURN-CODE>
    <DAYOFWEEK>MERCOLEDI</DAYOFWEEK>
  </DayofWeekResponse>
</XM0590>

```