Automating the Development of Device-Aware Web Services: A Model-Driven Approach

Achilleas Achilleos, Nearchos Paspallis and George A. Papadopoulos
Department of Computer Science, University of Cyprus
Email: [achilleas, nearchos, george]@cs.ucy.ac.cy

***Abstract*—The huge growth of the mobile devices market and the fast-changing requirements of mobile users, increase the need to develop native Web Service clients that can be deployed on both mobile and desktop devices. Existing work attempts to address heterogeneity, in order to simplify the development of device-aware Web Services. This paper defines the Presentation Modelling Language, which allows defining clients as graphical user interface models that are then transformed to platform-specific code. Also, the Web Services Description Language is used to define and generate the proxy classes that enable service communication from the generated clients. A functional device-aware Web Service is developed, showcasing the high-degree of automation achieved and the multi-platform nature of the approach.**

***Keywords*-model-driven; Web Services; cross platform development; code generation; device-aware; mobile services;**

I. Introduction

Mobile devices are hawking the marketplace over their stationary counterparts. Thus, mobile users requirements are rapidly increasing in terms of running more complex services on these mobile devices [1]. Also, the evolution of technologies (e.g. J2ME, C#) and the introduction of new ones (e.g. Android), creates additional requirements and restrictions when developing service-clients [2]. These arise from the graphical user interface (GUI) and resource limitations (e.g. screen, battery) imposed, when developing service clients that utilise Web Services (WS) from different devices.

To overcome these issues, research work focuses on developing clients, which can consume WS directly via the Web platform; e.g. using HTML forms. This confines the development process in using pure HTML and a plain web browser, so as to build and utilise Web Services across multiple devices and platforms. Hence, the following inherent constraints arise [3]: (i) difficulty in implementing sophisticated, interactive and refined functionality, and (ii) security implications presented when the secure HTTPS (HyperText Transfer Protocol Secure) protocol is used, since many mobile devices do not support it. In contrast, this work proposes a model-driven approach that allows building native web-based applications (rather than HTML-based) deployed across devices and platforms.

The Model Driven Architecture (MDA) paradigm [4] is combined with the Web Services technology, to uncouple the development of platform-specific clients from the implementation of the Web Service functionality; producing a fully functional device-aware Web Service. In particular, we support development for the following categories of devices: (i) Resource-rich devices; e.g. desktops, laptops, (ii) Resource-competent devices; e.g. Netbooks, iPad and (iii) Resource-constrained devices; e.g. mobile smartphones such as Google Nexus One, iPhone, Nokia N8. Our category set is defined based on an initial categorisation of devices performed by Ortiz et al. [5] and extended by adding and addressing the second category.

Foremost, the Presentation Modelling Language (PML) is defined that enables the design and automates the implementation of service clients. Also, the Web Services Description Language (WSDL) is exploited to design and generate device-specific proxy classes, which facilitate WS communication. Thus, designers define GUIs and collections of service communication endpoints as graphical models, which are transformed to different implementations and deployed on mobile devices to enable Web Service access. Hence, the WS functionality is coded manually in a single technology; e.g. J2ME, C#.

The paper is structured as follows: Section 2 presents the motivation of this work and Section 3 introduces the architecture of the proposed model-driven, Web Service oriented approach. The following section presents the PML, defined as an Eclipse Modelling Framework (EMF) metamodel. Section 5 maps PML concepts to the Android technology (i.e. transformation rules) and presents the WSDL code generation capabilities. Section 6 presents the case study and performs an evaluation using the Lines of Code (LoC) software metric. Concluding, Section 7 presents findings and future plans of this work.

II. Related Work

The development of GUIs is a taunting but essential task when building software applications. In particular, the development effort is largely increased when the same software service is developed for different platforms with various restrictions and requirements [2]. Initial work on GUI modelling, focuses on the definition of the GUI structure as presentation diagrams and its behaviour as hierarchical statechart diagrams [6]. The models are then transformed to Java-based GUI code, which can be extended for implementing functional multimedia desktop applications. Link et al. [7] propose a tool-based approach for the model-driven development (MDD) of GUIs for various target platforms. The key objective is to model the GUI properties of a software application and transform them into code. This automates the GUI implementation and reduces the effort to develop software applications.

A more recent approach [8] takes research work a step further, by addressing the development of fully functional software applications for mobile platforms. This approach allows non-expert users to easily design specialised mobile applications. The authors state the following in their work: "it still takes a large amount of skill and familiarity with different APIs to create a simple mobile application". To tackle development complexity, the Mobile Applications (MobiA) modelling tool is developed that allows designing mobile applications, which could then be transformed to platform-specific code. This work does not develop though any transformation tools. Moreover, the main functionality of the application is implemented and deployed on the actual device. This imposes a burden on resource-constrained mobile devices and does not favour interoperability.

Dunkel and Bruns [9] propose a simple and flexible approach for the development of mobile applications. A model-driven approach is presented that allows defining the client's GUIs and the service workflow using graphical models. These models are then transformed into XML-based descriptions (i.e. XForms code). The XForms W3C standard is selected because of its close correlation with the Mobile Information Device Profile (MIDP) of J2ME, which allows mapping easily XForm elements to MIDP elements; i.e. client code. The approach could address additional mobile implementations, by extending the code generators. The authors acknowledge a deficit of their approach, which arises from the necessity to integrate and use various Unified Modelling Language (UML) tools. These tools do not fully support metamodelling and provide proprietary and not yet stable code generation.

Ortiz et al. [5] state that mobile devices heterogeneity and their non-stop use in everyday life activities reveals the necessity to access WS from these mobile devices. The main objective of this work is to adapt the result of the WS invocation, based on the client's device type. Hence, the authors propose a service-side, aspect-oriented approach that allows developers to extend the implemented WS. This enables the adaptation of the WS invocation result in accordance to the client's device. The WS code is not directly affected since additional aspect code is implemented, which intercepts the invocation of the service operation and adapts it according to the device type detected. This approach suffers from three issues (as noted by the authors[10]): (i) client code is implemented that declares the device invoking the WS, (ii) response time is slightly increased since the service-side aspect code requires to process and adapt the response depending on the device type, and (iii) the complex task of implementing different service-clients (e.g. clients GUIs) is not considered.

Finally, Kapitsaki et al. [1] present an approach that automates the development of composite context-aware Web applications. The approach proposes complete separation of the web application functionality from the context adaptation. In particular, a methodology is adopted that utilises the UML for the design and automatic generation of a functional context-aware web application. However, the approach automates the development of context-aware Web applications (intended mainly for mobile users), which are formulated by existing third-party web services. This approach raises some security implications, since in many cases it is not possible to develop clients that interact with these services; i.e. permissions required. Also, although UML profiling is supported by different UML tools, it does not provide a standard way to access model stereotypes, so as to impose model constraints and map models to the context-aware implementation.

III. The Proposed Architecture

This work bridges MDA with WS, to exploit their potentials and overcome their limitations. Fig. 1 presents the architecture proposed in this work, which separates each developed device-aware Web Service into two constituent parts. The client-side comprises the GUIs and the necessary proxy classes, which facilitate respectively the interaction of the user with the device and the communication with the service by exchanging Simple Object Access Protocol (SOAP) messages. Both the GUI and proxy classes are generated from the abstract models (i.e. PML, WSDL), as shown in Fig. 1.

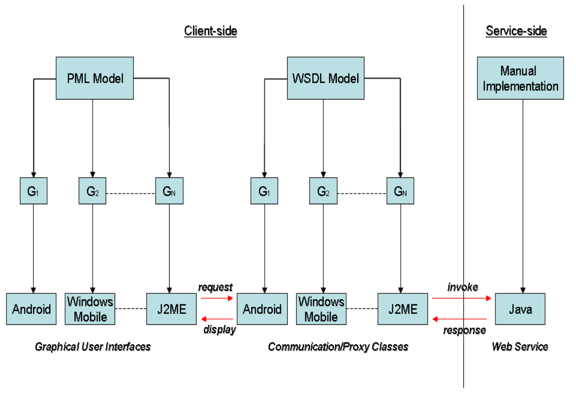


Figure 1. Model-Driven, Web Service-oriented Architecture.

The principal target is to define abstract PML models that include the necessary information for generating the GUI implementation of service clients. Thus, each PML model includes information that describe GUI elements (e.g. label), properties (e.g. label's text) and relationships (e.g. panel contains button). This information is described in the PML metamodel (section IV). On the basis of the metamodel definition, the code generators (section V) have been developed $(G_1, G_2, ..., G_N)$, which are tailored towards different platforms, so as to enable the transformation of PML models to platform-specific code.

Furthermore, existing WSDL code generation tools are used that enable the transformation of WSDL models into platform-specific proxy classes. Hence, if we assume that Android is the target platform, the generated Android GUI classes use the Android proxy classes to consume the WS, receive the response message and display the information on the Android mobile device. The same reasoning applies for all platforms, since the model-driven approach enables the generation of different implementations. This allows

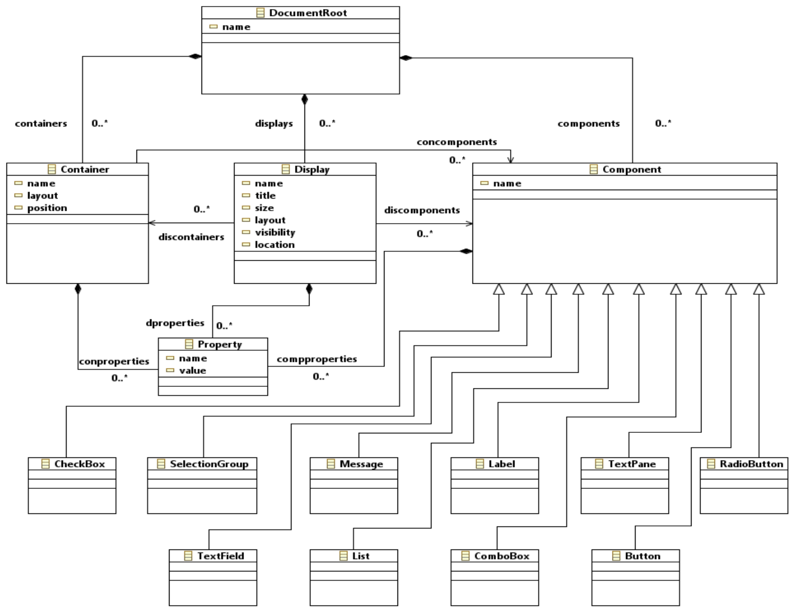


Figure 2. Presentation Modelling Language metamodel.

building device-aware Web Services and deploying them across different platforms; see Fig. 1.

The WS functionality, is the only part that must be manually implemented by developers. However, it is coded in one implementation (in this work it is Java-based), since the WS platform enables clients implemented in various technologies (e.g. J2ME, Windows) to consume the same WS. This is possible since the communication is performed using SOAP, which is a simple protocol for exchanging XML-based structured information in computer networks. It also relies on Remote Procedure Call (RPC) and HyperText Transfer Protocol (HTTP) for connection negotiation and message transmission. This enables interoperability (proven characteristic of WS [1]) between various platforms, and it simplifies and speeds up the development of fully-functional, device-aware Web Services.

IV. Presentation Modelling Language

The PML is defined as an EMF metamodel, presented in Fig. 2, which describes the modelling elements, their associations and properties. This information defined in the metamodel enable the design of GUIs in the form of visual abstract PML models. The metamodel was defined by examining and evaluating various GUI elements, associations and properties that share similarities across platforms. Example cross-platform elements are containers, labels, text boxes, etc. Moreover, typical associations between objects exist across platforms, such as container (e.g. panel) *includes* component (e.g. label).

As illustrated in Fig. 2 the top element defines the whole PML model, which includes the rest of the graphical modelling elements. It is specified using the *DocumentRoot* metaclass that includes the *name* property and aggregations to the rest of the model elements. First, the *displays* aggregation designates that each model contains zero to many displays (i.e. device screen). To reduce complexity we impose an OCL rule that allows to define merely one display element per model, so as to improve the model's comprehension and reduce its complexity.

The display model element is specified as an instance of the *Display* metaclass, which includes graphical properties that are common across platforms. First, the *name* property defines the actual name of the display element, while the second property defines the title shown onto the main display of the device. The size and layout properties define the actual size of the display and the layout of container elements onto the main display. Concluding, the visibility property specifies if the display is visible or not, while the location property defines its position onto the screen.

Following, the *containers* aggregation defines the containment relationship between the display and its container elements. These components are described as the containers of secondary components, such as labels, buttons, etc. The PML defines also these secondary components, which are common and widely-used in all implementation platforms. In particular, the *Component* metaclass defines the parent of the following child metaclasses: *Message, Label, Button, TextPane, RadioButton, ComboBox, CheckBox, TextField, List and SelectionGroup* modelling elements. Hence, the designer is able to instantiate these child metaclasses and define different secondary graphical components using an abstract representation.

The *discontainers, discomponents and concomponents* associations define the relationships between the above modelling elements. Foremost, the *discontainers* association describes that the main display element may include different container elements. Moreover, the *discomponents* association defines the containment relationship of the display element directly with various secondary components; e.g. labels, buttons. Also, the *concomponents* association describes that container elements may as well include different secondary components.

Finally, the *Property* metaclass is a key element, since it allows describing miscellaneous graphical properties for the modelling elements. The aggregation associations *dproperties, conproperties and compproperties* define clearly that each element may contain various properties, which are defined as instances of the metaclass. Properties are defined as keywords, which are recognised and transformed via the transformation rules defined within code generators. In addition, the following OCL constraint ensures that the designer can define only keywords (i.e. properties) supported by the PML. Thus, the flexibility is provided to extend the PML, by adding keywords to the OCL rule and the generators. The specification of OCL rules completes the PML definition and allows generating a supporting Presentation Modelling Framework (PMF), which includes an editor with drag-and-drop capabilities for designing and validating PML models.

context Property

inv: Property.allInstances()→forAll(p: Property | p.name = 'text' **or** p.name = 'title' **or** p.name = 'message' **or** p.name = 'rows' **or** p.name = 'columns' **or** p.name = 'lineWrap' **or** p.name = 'stringArray' **or** p.name = 'command')

V. Building the Code Generators

To support the transformation of PML models to the necessary target implementations, it is imperative to define coherent transformation rules that compose the code generators. This ensures the correctness of the generated code operational semantics. The development of the generators is performed using our previously proposed MDD environment [11], enabling as a result the transformation of PML models to the target GUI implementations. Also, this approach exploits existing code generators to transform WSDL models to the required source code (i.e. proxy classes) that enables the communication with the WS.

A. Presentation Modelling Language Code Generation

The MDD environment [11] includes the openArchitectureWare (oAW) software tool that enables the development of code generators by defining model-to-text transformation rules. The tool's most important components are the *Xpand* template language and the workflow execution engine. Foremost, the *Xpand* language supports the definition of advanced code generators as templates, which capture the transformation rules and control the output document generation; e.g. J2ME, C#, HTML. Finally, the workflow execution engine drives the code generation on the basis of the defined templates and the input model.

Fig. 3 presents the code generation process, which is driven by the workflow execution engine. The most critical parts of the process are the PML model and the templates. Also, a workflow script is defined, which is executed via the workflow engine, calling the necessary Java classes

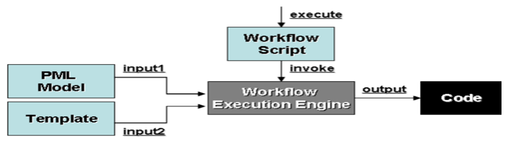


Figure 3. The PML code generation process.

of the oAW tool that drive generation. Listing 1 presents part of the Android-specific template definition that allows demonstrating how code generation is achieved. Note that the required scripts and code generators are defined for all platforms but Android is presented here as an example. Lines 1-4 define the path to extension functions, define the *DocumentRoot* metaclass as the root element and access the *displays* collection. Next, the *Display* metaclass is defined as the primary element (i.e. line 6), which enables access to its associated containers, the secondary components and their properties to facilitate code generation.

Lines 25-46 define a key template part, which allows iterating through the collection of display containers. This enables access to containers' properties and the secondary components associated to these containers. Lines 29-45 define a second loop that iterates through the collection of secondary components, which are associated with the current container in the first iteration. Thus, the properties and associations of all secondary components can be accessed through the corresponding variable reference and the required conditional statements can be defined that drive generation according to the component type.

Listing 1. Part of the Android-specific template definition

```
1.  <<EXTENSION extensions::AndroidExtensions>>
2.  <<DEFINE Root FOR pres::DocumentRoot>>
3.  <<EXPAND Display FOREACH displays>>
4.  <<ENDDEFINE>>
5.  <<REM>>Define Display - Primary Element.<<ENDREM>>
6.  <<DEFINE Display FOR pres::Display>>
    ...
20. <<REM>>Create method for activity.<<ENDREM>>
21. /** Called when the activity is first created. */
22. public void onCreate(Bundle savedInstanceState) {
23. super.onCreate(savedInstanceState);
24. <<REM>>Set title and layout of Activity.<<ENDREM>>
25. <<FOREACH this.discontainers AS discon->>
26. this.setTitle(<<this.title>>);
27. TableLayout <<discon.name>> = new TableLayout(this);
28. <<REM>>Create associated components.<<ENDREM>>
29. <<FOREACH discon.concomponents AS concomp->>
30. <<IF concomp.metaType.name.matches("pres::Label")->>
31. <<concomp.name>> = new TextView(this);
32. <<concomp.name>>.setText(<<concomp.compproperties.
    select(e|e.name.contains("text")).value.first()>>);
33. <<ELSEIF concomp. ...... .matches("pres::Button")->>
34. <<concomp.name>> = new EditText(this);
    ...
44. <<ENDIF>>
45. <<ENDFOREACH>>
46. <<ENDFOREACH>>
    ...
58. <<ENDDEFINE>>
```

Each conditional statement allows checking the type (i.e. metaType) of the current secondary component and generating the necessary source code. For instance, the logical statement defined at line 30 ensures that the current component is an instance of the *Label* metaclass, so as to generate the code that implements a *TextView* component. Using the same reasoning the whole template is defined, which transforms the PML model to Android code. Additional templates were defined, which drive PML model transformation to various GUI implementations.

B. Web Service Description Language Code Generation

This subsection presents briefly the transformation of WSDL models to the corresponding proxy classes that support the communication with the WS. The WSDL serves as a platform-independent specification language that allows describing the functionality of WS using abstract models. Thus, different technologies have developed their individual code generation tools, which enable the transformation of WSDL models to implementation classes that permit to invoke and retrieve responses from WS. In this work we exploit existing WSDL tools, such as the Axis2 *wsdl2java* tool or the .NET *wsdl* tool, so as to transform an input WSDL model to fully annotated platform-specific code. Further details on the implementation of these tools is out of the scope of this work.

VI. Case Study: The *BookStore* Prototype

A. Prototype Overview

The prototype is a *BookStore* device-aware Web Service that allows searching for books stored in a repository using their title and returning to the users the necessary details of the book. Primarily, the service-side main functionality of the prototype is manually implemented by the developer in Java. The implementation utilises the Java Open Database Connectivity (ODBC) standard, which enables access to database management systems (DBMS) for querying and retrieving data. The second step involves the definition and validation of the PML and WSDL models and their transformation to different platform-specific service-clients.

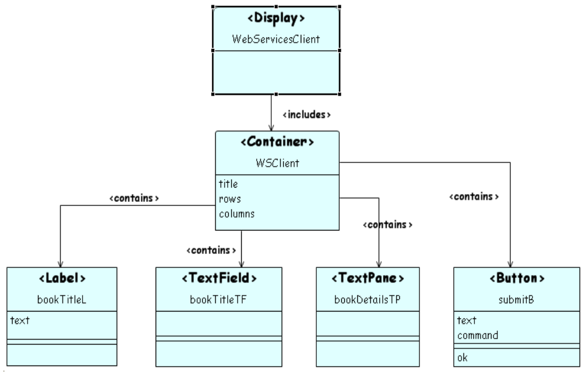


Figure 4. The *BookStore* Web Service Presentation Model.

Fig. 4 presents the PML model that defines the GUIs of the service-clients. At the top we have an instance of the *Display* metaclass, which represents the main display of the GUI. It also includes additional properties (not shown in the figure) and associations. For instance, the display is associated with a single container component (i.e. *Container* instance), which includes its own properties and contains secondary components; e.g. label. These secondary components are defined as instances of the respective metaclasses and include their own properties. Finally, the model is validated, prior to code generation, to ensure that no errors exist in the PML model.

Listing 2. The GUI code generated for the Android target platform.

```
1.  package client.gui;
    ...
11. public class BookStoreWSAndroidClient
12.      extends Activity implements OnClickListener {
    ...
19. public void onCreate(Bundle savedInstanceState) {
20.  super.onCreate(savedInstanceState);
21.  this.setTitle("BookStoreWSClient");
22.  TableLayout WSClient = new TableLayout(this);
23.  bookTitleL = new TextView(this);
24.  bookTitleL.setText("Enter Book Title:");
25.  bookTitleTF = new EditText(this);
     ...
31.  bookDetailsTP = new TextView(this);
32.  bookDetailsTP.setText("");
33.  /** TODO starts */
     ...
45.  /** TODO ends */
46. }
47. public void onClick(View event) {
48.   if (event.equals(submitB)) {
49. /** TODO starts */
50. AndroidProxy proxy_stub = new AndroidProxy();
51.   try { String result = proxy_stub.
52.  getBookDetails(bookTitleTF.getText().toString());
53.   bookDetailsTP.setText(result);
54.   } catch (Exception e) {
55.    e.printStackTrace();
56. }
57.   /** TODO ends */
58. }
59. }
```

Listing 2 shows the Android-specific code generated from the PML model. Lines 20-24 of the code, are generated via the rules defined in lines 23-32 of Listing 1. These rules transform the *Display*, *Container* and *Label* elements and their properties shown in Fig. 4. The developer implements manually a few lines of code (*TODO* branches), which have to do with the tasks of placing components to the container and invoking the WS (i.e. lines 49-57) using the proxy classes generated from the WSDL model; Fig. 5. Thus, the implementation effort is reduced, since a large percentage of the code is generated for different implementations. This reduces the coding effort, addresses device heterogeneity and portability across platforms.

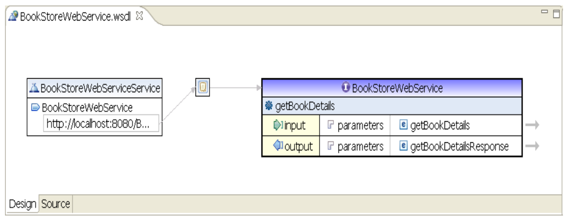


Figure 5. The *BookStore* Web Service Description Language Model.

B. Evaluation using the Lines Of Code Software Metric

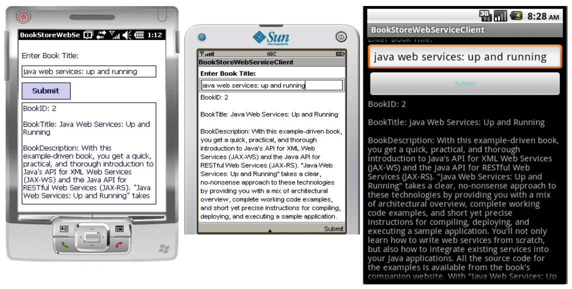
This subsection evaluates our approach using the LoC metric, which allows demonstrating the high-degree of automation achieved and the decrease of the implementation effort. Table I presents the results obtained by comparing the generated code with the fully-functional code of each service-client. The WS main functionality is implemented only once using Java and is consumed by different clients. Hence, the WS code (i.e. 55 LoC) is considered only when deriving the percentage for all target implementations.

Table I reveals that a significant part of the service-clients code has been automatically generated from the models; i.e. percentages are above 75%. Furthermore, our experience in defining code generators [11] suggests that transformation rules can be optimised, so as to achieve higher-degree of automation. Also, the diversity of technologies reveals the merits of the transformation, i.e. rapidly adaptable to additional platforms; e.g. Apple iOS.

Table I
EVALUATION RESULTS USING LOC SOFTWARE METRIC.

Metric (LoC)	**Generated Code**	**Overall Code**	**Generated/ Overall (%)**
Java	295	316	93.35%
J2ME	206	225	91.56%
Android	74	93	79.57%
Windows Mobile	80	94	85.11%
Windows Desktop	116	123	94.31%
All Platforms (Average)	771	851	90.60%

These results, although confined to the case study, allow extracting the necessary conclusions as to the development efficiency of the proposed approach. We need to acknowledge though, that implementing less lines of code does not necessarily indicate a decrease in the development effort. This is because developers must familiarise themselves with the code generators, so as to be able to extend easily the generated implementation. We argue that the approach can be more beneficial in the long term, as developers learn the specifics of the generated implementation.



Enter Book Title:

java web services: up and running

BookID: 2

BookTitle: Java Web Services: Up and Running

BookDescription: With this example-driven book, you get a quick, practical, and thorough introduction to Java's API for XML Web Services (JAX-WS) and the Java API for RESTful Web Services (JAX-RS). "Java Web Services: Up and Running" takes

BookStoreWebServiceClient

Enter Book Title:

java web services: up and running

BookID: 2

BookTitle: Java Web Services: Up and Running

BookDescription: With this example-driven book, you get a quick, practical, and thorough introduction to Java's API for XML Web Services (JAX-WS) and the Java API for RESTful Web Services (JAX-RS). "Java Web Services: Up and Running" takes a clear, no-nonsense approach to these technologies by providing you with a mix of architectural overview, complete working code examples, and short yet precise instructions for compiling, deploying, and executing a sample application.

BookStoreWebServiceClient

java web services: up and running

BookID: 2

BookTitle: Java Web Services: Up and Running

BookDescription: With this example-driven book, you get a quick, practical, and thorough introduction to Java's API for XML Web Services (JAX-WS) and the Java API for RESTful Web Services (JAX-RS). "Java Web Services: Up and Running" takes a clear, no-nonsense approach to these technologies by providing you with a mix of architectural overview, complete working code examples, and short yet precise instructions for compiling, deploying, and executing a sample application. You'll not only learn how to write web services from scratch, but also how to integrate existing services into your Java applications. All the source code for the examples is available from the book's companion website. With "Java Web Services: Up

Figure 6. The *BookStore* Web Service Deployed on Different Devices.

Finally, the key contribution lies in the ability to address device heterogeneity and portability across platforms, while at the same time alleviating the computation burden from the clients (especially for mobile devices) via the Web platform. In addition, the need to overcome the inherent limitations of HTML (see section I) when developing client applications directed us towards a model-driven, Web Service oriented approach. Fig. 6 demonstrates the cross-platform nature of the proposed approach through the deployment of the prototype on different platforms.

VII. CONCLUSION

This research work showcases that the model-driven development of device-aware Web Services is both feasible and applicable. In particular, the proposed approach reveals that by combining the MDA paradigm with the Web Services technology, we can exploit their potential and overcome their limitations. First, the MDA paradigm provides the desired platform-independence, while the Web Services technology ensures interoperability amongst platforms and lessens the clients computational load. The initial prototype demonstrated the cross-platform applicability and the development efficiency obtained. An inherent limitation of the approach, is the necessity to learn how to use the languages and realise the code generated from the models; so as to extend it. Our immediate objective is to extend this work by considering user preferences and performing also a performance analysis evaluation.

REFERENCES

[1] G. M. Kapitsaki, D. A. Kateros, G. N. Prezerakos, and I. S. Venieris, "Model-driven development of composite context-aware web applications," *Information and Software Technology*, vol. 51, no. 8, pp. 1244–1260, 2009.

[2] D. Dern, "Cross-platform smartphone apps still difficult," *IEEE Spectrum*, 2010.

[3] M. Bloice, F. Wotawa, and A. Holzinger, "Java's alternatives and the limitations of java when writing cross-platform applications for mobile devices in the medical domain," in *Proceedings of 31st ITI Conference*, 2009.

[4] Y. Singh and M. Sood, "Model driven architecture: A perspective," in *in Proceedings of IEEE ACC Conference*, 6-7 2009, pp. 1644–1652.

[5] G. Ortiz and A. G. de Prado, "Adapting web services for multiple devices: A model-driven, aspect-oriented approach," *IEEE Congress on Services*, pp. 754–761, 2009.

[6] S. Sauer, M. Drksen, A. Gebel, and D. Hannwacker, "Guibuilder: A tool for model-driven development of multimedia user interfaces," 2006.

[7] S. Link, T. Schuster, P. Hoyer, and S. Abeck, "Focusing graphical user interfaces in model-driven software development," in *Proceedings of the ACHI Conference*. IEEE Computer Society, 2008, pp. 3–8.

[8] F. T. Balagtas-Fernandez and H. Hussmann, "Model-driven development of mobile applications," in *Proceedings of the ASE Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 509–512.

[9] J. Dunkel and R. Bruns, "Model-driven architecture for mobile applications," in *Proceedings of the 10th BIS Conference*. Springer-Verlag, 2007, pp. 464–477.

[10] G. Ortiz and A. G. de Prado, "Mobile-aware web services," *in Proceedings of the IEEE UBICOMM Conference*, vol. 0, pp. 65–70, 2009.

[11] A. Achilleos, K. Yang, and N. Georgalas, "A model driven approach to generate service creation environments," in *Proceedings of IEEE GLOBECOM*, nov. 2008, pp. 1 –6.