Journal of Web Engineering, Vol. 12, No.3&4 (2013) 317-359

ALEXANDRIA: A VISUAL TOOL FOR GENERATING MULTI-DEVICE RICH INTERNET APPLICATIONS

[1]LUIS OMAR COLOMBO-MENDOZA
[2]GINER ALOR-HERNÁNDEZ
Instituto Tecnológico de Orizaba, Orizaba, Mexico
[1]lcolombo@acm.org
[2]galor@itorizaba.edu.mx

ALEJANDRO RODRÍGUEZ-GONZÁLEZ
Centro de Biotecnología y Genómica de Plantas, Universidad Politécnica de Madrid, Spain
alejandro.rodriguezg@upm.es

RICARDO COLOMO-PALACIOS
Universidad Carlos III de Madrid, Madrid, Spain
ricardo.colomo@uc3m.es

Received August 3, 2012
Revised June 18, 2013

Rich Internet Applications (RIAs) Engineering is an emerging area of Software Engineering, which still lacks of adequate development approaches and tools for support compared to Web Engineering. Therefore, in most cases the development of RIAs is performed in an ad-hoc manner and it is just driven by a set of new frameworks, which are mainly classified into JavaScript-based and non-JavaScript-based frameworks. RIAs development involves design principles of Web and desktop applications because RIAs, which are a new generation of Internet applications, combine behaviours and features of these two kinds of applications. Furthermore, mobile devices such as smartphones and tablet computers are also being involved in RIAs development because of the growing demand for ubiquitous Web 2.0 applications; therefore, RIAs are known as multi-device RIAs. During the last few years different contributions have arisen with the aim of bridging the gap between the Web and the RIAs engineering support. These proposals which are either: 1) extensions of existing methodologies for Web and hypermedia applications development, or 2) Model-driven Development (MDD) methods for rich Graphic User Interfaces (GUIs) designing, which do not cover multi-device RIAs development. Furthermore, some proposals lack of support tools. Taking this into account, in this paper we propose a visual tool that implements a GUI pattern-based approach for code generation of multi-device RIAs. This visual tool called AlexandRIA is a source and native code generator for Rapid Applications Development (RAD), which allows automatically generating code based on a set of preferences selected throughout a wizard. In order to validate our proposal, two cloud services APIs-based multi-device RIAs are generated using AlexandRIA. Finally, a qualitative/quantitative evaluation was performed in order to accurate the legitimacy of our proposal against other similar academic and commercial proposals.

Key words: multi-device RIA, cloud services API, code generator, RAD tool
Communicated by: D. Schwabe & J. Vaderdonckt

1 Introduction

Rich Internet Applications (RIAs) are a new generation of Internet applications that combine behaviors and features of Web and desktop applications such as: 1) client-server architecture, 2) data-intensive handling and business logic execution both on the client-side and on the server-side, which results in advanced mechanisms of client-server communication, and 3) highly interactive multimedia content. Therefore, this kind of Internet applications allows users to do interactive data explorations through attractive visual interfaces increasing usability and performance requirements [1].

RIAs engineering is an emerging field of Software Engineering, which lacks of development approaches and tools for support compared with Web Engineering. For this reason, two main kinds of proposals have arisen in the literature: 1) extensions to existing methodologies for Web and hypermedia development, which are focused on data-intensive applications modeling and they ignore issues related to GUIs improvement and 2) Model-driven Development (MDD) methods for rich GUIs design, which are focused on legacy Web applications reengineering. However, these proposals are based on software modeling and although they finally allow generating source and executable code, at least in a semi-automatic way, they take too much development time and effort. Thus, the development of RIAs is performed in an ad-hoc manner and it is just driven by a set of well-known programming languages and frameworks such as Adobe® Flex, Microsoft® Silverlight™ and Google Web Toolkit™ [2].

Nowadays RIAs development demands design principles of Web and desktop applications, which are implemented by the so-called interaction design patterns. Furthermore, mobile devices such as smartphones and tablet computers have also been involved in RIAs development because of the ubiquitous requirements of Web 2.0 applications [3]. In this sense, RIAs are known as multi-device RIAs. This term covers RIAs that run as cross-browser Web applications, cross-platform desktop applications and applications for diverse mobile devices. A Web browser-based RIA is a Web 2.0 application that integrates desktop-like features and it can be based on HTML and JavaScript or on a RIA technology. A desktop RIA is a kind of RIA that is able to run off-Web browser. A mobile RIA is a native mobile application with an improved GUI. The architecture of these applications commonly has a services back-end which represents the business logic; in fact, the use of cloud services is increasingly common in the development of Web 2.0 applications.

Therefore, the adaptation of existing Web methodologies such as OOH [4], OOHDM [5], UWE [6] and WebML [7] in including RIA features is becoming more complicated. In this sense, new model-driven development (MDD) methods for designing rich GUIs such as RUX-Method [8], the work of Martinez-Ruiz, Arteaga, Vanderdonckt, Gonzalez-Calleros, & Mendoza [9] and the work of Valverde & Pastor [10] have considered a variety of devices and technologies for generating RIAs GUIs. However, these works do not cover multi-device RIAs development from the conception phase to the deployment phase. In contrast, the GUI pattern-based approach for code generation of multi-device RIAs we outline in this paper covers multi-device RIAs code generation, from the identification of the required type of application to the deployment of the generated application on target devices. This approach is mainly focused on GUIs aspects because it allows designing RIAs as compositions of interaction design patterns; nevertheless, it also considers other aspects of RIAs, namely the ability to distribute business logic operations between server and client and the use of sophisticated client-server communication mechanisms.

As a salient contribution, in this paper we propose a visual tool called AlexandRIA that automates an approach for code generation of multi-device RIAs. In fact, AlexandRIA is a code generator for Rapid Applications Development (RAD) that allows automatically generating source and native code based on a set of preferences selected throughout a wizard. AlexandRIA uses a set of reusable software components implementing interaction design patterns as well as diverse kinds of templates as the building blocks of the code generation engine. As a proof of concept, we have generated both the Web browser-based version and the native version of a mobile RIA based on popular cloud services APIs using AlexandRIA. Moreover, we have performed a two-part comparative evaluation in order to establish the appropriateness of AlexandRIA against other available RIAs development tools not only in terms of its features but also in terms of the features of the source and native code generated by using it.

This paper is structured as follows: section 2 presents a review of the state-of-the-art on RIAs engineering, section 3 describes the code generation algorithm behind AlexandRIA, section 4 describes the architecture and internal functionality of AlexandRIA, section 5 explains a case study for the generation of two cloud services APIs-based multi-device RIAs, section 6 presents the method aimed at assessing the proposed tool by comparing it against other available options, section 7 discuss the future directions. Finally the conclude remarks are made in section 8.

2 State of the art

In recent years, several works have been proposed with the aim of bridging the gap between the engineering support that traditional Web applications have and the lack of development methodologies, design methods, modeling languages and visual tools for support the development of Web 2.0 applications, specifically the design and development of RIAs. Linaje et al. [8] proposed a MDD method for designing RIA GUIs that allows engineering the adaptation of legacy model-based Web 1.0 applications to Web 2.0 GUIs. This method called RUX-Method broken down the GUI design in four levels: 1) concepts and tasks, 2) abstract interface, 3) concrete interface and 4) final interface. RUX-Method defines transformations between these design levels. Concepts and tasks are the data and business logic of the adapted Web application. GUI transformations are based on a RIA component library. Linaje, Preciado, Morales-Chaparro, Rodríguez-Echeverría, & Sánchez-Figueroa [12] proposed a MDD tool that implements the RUX-Method. This tool called RUX-Tool works together with the WebRatio® tool to obtain the content structure and the business logic of legacy WebML-designed applications and gives support to the automatic RIAs final interface code generation. The RUX-Tool is a RIA itself and its architecture has two sides: client and server side. The client side represents an Integrated Development Environment (IDE) whereas the server side represents an interface with the WebRatio® projects.

Bozzon et al. [7] proposed an extension of the WebML modeling language in order to capture the specificity of RIAs. This improvement applies to both data and hypertext design of conceptual modeling for Web applications. The extended models are able to capture the distinction between server and client-side entities and pages. Besides, in order to produce a running RIA, an extension of WebRatio® CASE tool architecture was provided by means of a client-side controller coded in LZX and the reuse of existing rules for server-side code generation. Valverde & Pastor [13] proposed a MDD approach to support the RIAs development. This approach introduces an interaction model that

defines the RIA GUI from the perspective of the user interaction and can be used inside a model-based Web methodology. The interaction model was composed of an abstract view, a concrete view and a set of model-to-code transformations. The abstract view was made up of abstract interaction design patterns whereas the concrete view is made up of RIA interaction design patterns which address technology-specific details. Urbieta et al. [5] proposed an extension of the Object-Oriented Hypermedia Design Method (OOHDM) as an approach for designing RIAs. This approach used the Abstract Data View (ADV) design model to allow specifying the structure and behaviors of rich GUIs; also it extends the ADV formalism with the aim of giving support to the composition of GUI objects that correspond to different concerns which crosscut each other. In particular, the behavioral crosscutting concerns are separately modeled but are finally integrated using weaving ADV-charts.

Rossi, Urbieta, Ginzburg, Distante, & Garrido [14] presented an approach for systematically transforming a legacy Web application into a RIA. In this approach, the concept of Web Model Refactoring (WMR) was applied to navigation or interface models with the aim of introducing rich interface functionality in a Web application. The improvements are called RIA refactorings and are described as compositions of RIA GUIs, which instead of implying changing the original ADV model are formally specified by composing new ADVs. Koch, Pigerl, Zhang, & Morozova [15] proposed a pattern approach for the MDD of RIAs. The authors propose applying RIA patterns at a modeling phase by using UML state machines which can be embedded into the models of all UML-conform Web methodologies. They also proposed an extension of the UWE presentation layer meta-model, which implies the definition of meta-attributes. Meta-attributes allow tagging existing UWE models. Wright [16] proposed a MDD-compliant and platform-independent modeling language for RIAs called Internet Application Modeling Language (IAML) and a CASE tool that supports it. IAML reuses some modeling aspects of UML. Besides, IAML was designed following a model inference approach in order to allow inferring missing knowledge of models. The proposed CASE tool is an Eclipse-based tool that allows generating deployable HTML/JavaScript Web applications from model instances in IAML using the OpenArchitectureWare framework. Gharavi, Mesbah, & Deursen [17] proposed an MDD approach for generating AJAX Web applications. This approach includes an UML-based meta-model for modeling AJAX GUIs and an AndroMDA AJAX cartridge for generating the corresponding final code which is based on the ICEFACES framework. The proposed AJAX meta-model was integrated in the Platform-Specific Model (PSM) layer of the AndroMDA tool whereas the AJAX cartridge manages the PSM transformations.

Sorokin, Montero, & Märtin [18] proposed an MDD framework for generating and evaluating Adobe® Flex applications. This framework defines four layers: task model, abstract interface model, concrete interface model and RIA final code. It also defines a set of transformations. The framework uses international standard task-based metrics in order to integrate usability evaluation facilities to the generated Adobe® Flex applications. The translation of the abstract interface to the MXML-based concrete interface is achieved using Extensible Stylesheet Language Transformations (XSLT) transformations. Paterno', Santoro, & Spano [19] proposed an XML-based User Interface Design Language (UIDL) called MARIA XML, which provides support for multi-device interactive applications based on Web services. Furthermore, the associated tool that supports such language was also proposed. MARIA XML allows designing user interfaces at both abstract and concrete levels; also it provides useful support for dynamic generation of user interfaces adapted to different devices. The associated tool called MARIA TOOL supports the dynamic generation through XSLT transformations.

Melia et al. [4] proposed a MDD process for Google Web Toolkit™ (GWT)-based RIAs that extends the OOH methodology. The proposed process called OOH4RIA uses the OOH domain and navigation models to define the server-side of a RIA and introduces presentation and orchestration models that represent the client-side of a RIA. The presentation model was formalized by a meta-model of RIA widgets. The orchestration model, which was based on the UML state machine diagram, captures interaction design patterns from widgets.

Machado et al. [6] proposed an extension to the UWE methodology for RIAs development. This extension called UWE-R covers navigation, presentation and processes aspects, which are defined using stereotypes. Navigation extensions allow modeling: 1) no-hypertext based nodes such as Flash and Java applets, and 2) asynchronously and visual continuity behaviors. Presentation extensions allow modeling widget containers common in RIA GUIs. Finally, processes extensions basically refer to the distinction between server-side and client-side processes. Martinez-Ruiz et al. [9] proposed a MDE method for designing RIA GUIs that is composed of four phases. In the first phase a task & domain model was defined. In the second phase a set of abstract user interfaces defined in UsiXML are generated using the IDEALXML tool. In the third phase the selected abstract user interface was transformed to a concrete user interface composed of native XAML, MXML or LZX components using XSLT transformations. In the fourth phase a final user interface code was generated using XSLT transformations. Dolog & Stage [20] proposed an UML-based design method for RIAs called ADRIA. This method employed task models for describing structural details of user interaction, also it proposes: 1) an interaction spaces model, which describes how particular tasks are supported by a user interface and 2) a guide model, which uses statechart diagrams for modeling behavioral details of user interaction. Stearn [21] described an application runtime engine for desktop and Web Mozilla-based RIAs. This framework called XULRunner was based on the Mozilla Firefox codebase. The XULRunner GUI layer mainly uses the XUL language for the definition of user interfaces but also supports other languages such as HTML. The XULRunner business logic layer relies on Cross Platform Component Object Model (XPCOM) which integrates JavaScript, C/C++, Java and Python programming languages. JavaScript is primarily used for controlling XULRunner-based applications.

Preciado, Linaje, Sanchez, & Comai [22] proposed an evaluation process based on main RIA features in order to obtain the suitability of an existing methodology for modeling RIAs. Web methodologies such as UWE and WebML, multimedia methodologies like OMMMA and hypermedia methodologies like HMT are compared in this work. The authors conclude that none of the evaluated methodologies allow properly modeling RIAs, although most of the modern Web and hypermedia methodologies are sufficiently flexible to be extended towards RIAs features. Valverde & Pastor [10] proposed a MDD GUI development process for Web 2.0 applications. This approach includes: 1) a generic and Web methodology-independent RIA meta-model, which combines both the static and dynamic views of RIA GUIs and it can be used to define technology-specific RIA meta-models and 2) a weaving meta-model that establishes the relationships between the RIA meta-model and the chosen Web methodology meta-model. This approach can be used inside any model-based Web methodology such as OWS. Rodríguez-Echeverría, Conejero, Linaje, Preciado, & Sánchez-Figueroa [23] proposed a MDD re-engineering process of WebML-designed applications into RIAs. In this work, the following meta-models are defined: 1) a CMM meta-model that is the WebML meta-model, 2) a RMM meta-model that is the CMM meta-model extended with RIA features and 3) a VMM meta-model that is the weaving meta-model for driving the model composition process; i.e., the generation of the final RIA

models which was performed by using generic ATL transformation rules. Martínez-Nieves et al. [1] proposed a model-based process for RIAs development that merges features from UWE methodology and ADV design model. This process called Phases Process for RIAs Development (PPRD) defines six phases: 1) requirements identification, 2) requirements modeling, 3) architecture definition, 4) conceptual modeling, 5) navigation modeling and 6) presentation modeling. Requirements, conceptual and navigation modeling phases employ UWE models whereas presentation modeling uses the ADV design model for designing the GUI appearance.

Finally, we have analyzed Adobe® AIR® Launchpad 3.0.1 which is a desktop tool developed by Adobe® that allows generating ready-to-compile source code of Adobe® AIR-based applications, i.e., desktop applications and applications for Android®, Apple® iOS and BlackBerry® Tablet OS platforms deployed on the Adobe® AIR® runtime. The generated source code is located in a folder structure which is ready-to-import in Adobe Flash® Builder® 4.5. Despite the similarities to our proposal, the major drawbacks of this tool are 1) it does not generate native code and 2) it does not consider the generation of Web applications.

In order to analyze more precisely the works described above and compare them with our proposal, we have defined a comparative table (see Table 1) which summarizes relevant issues of these researches and classifies them in: 1) extensions of Web and hypermedia methodologies, 2) purely RIAs development approaches and 3) RIAs GUI development approaches, which are used inside of existing Web methodologies. This classification is inspired by the classification proposed by Busch & Koch [2]. Based on results obtained from the comparative analysis summarized in table 1 we can conclude that current initiatives for RIAs engineering suffer from several drawbacks such as: a) they do not consider the multi-device nature of RIAs; i.e., they do not consider the convergence of Web applications, desktop applications and applications for mobile devices, b) Unlike our proposal, some are based on the MDD approach; therefore, they are focused on software modeling putting the automatic code generation in second place and c) many do not have tool support, which implies more development time and effort.

These deficiencies can be improved by: a) a development approach that integrates the multi-device feature of RIAs b) a visual tool that supports the development of multi-device RIAs enabling code generation in a fully-automatic way. This work tries solving the aforementioned deficiencies.

Proposal	Approach	Motivation	Underlying Web Methodology	Multi-device RIAs Support	Interaction Design Patterns Support	Tool Support
Extensions of Web and Hypermedia Methodologies with RIA features						
OOH4RIA [4]	Hypermedia methodology extension (meta-model)	GWT-based RIAs development	OOH	No	Yes (UML state machine diagrams)	No
[5]	Hypermedia methodology extension (ADV model)	RIAs development	OOHDM	No	Yes (ADV-charts)	No
UWE-R [6]	Web methodology extension (meta-model)	RIAs development	UWE	No	No	No
[7]	Web methodology	Data-intensive RIAs	WebML	No	No	WebRatio®

Proposal	Approach	Motivation	Underlying Web Methodology	Multi-device RIAs Support	Interaction Design Patterns Support	Tool Support
	extension (model elements)	development				CASE tool
RIAs Development Approaches						
PPRD [1]	UML model-based process for RIAs	RIAs analysis and designing	-	No	No	No
[9]	MDE GUI design method	RIAs development	-	No	No	No
PPMRD [11]	Code generation process for RIAs	RIAs development	-	Yes	Yes	No
IAML [16]	MDD-compliant modeling language for RIAs	RIAs development	-	No	No	EMF based-tool
[17]	MDE process for RIAs	AJAX–based RIAs development	-	No	No	AndroMDA
MARIA XML [19]	Modeling language for interactive applications	Web services-oriented interactive applications development	-	Partially (Multi-device GUI)	No	MARIA TOOL
ADRIA [20]	UML model-based analysis and design method	RIAs analysis and designing	-	No	No	Any CASE tool (UML)
[23]	MDE process for RIAs	Legacy WebML-designed apps. reengineering and RIAs development	-	No	Partially (related to synchronization)	No
RIA GUIs Design Approaches (used inside a Web methodology)						
RUX-Method [8]	MDE GUI design method	Legacy Web applications reengineering and RIAs development	Any model-based (concept. and navigation model)	Partially (Multi-device GUI)	No	[12]
[10]	GUI design method for MDE of RIAS (meta-model)	Legacy Web applications reengineering and RIAs development	Any model-based (navigation model)	No	Yes (Meta-models)	No
[14]	GUI reengineering method for MDE of RIAs	Legacy Web applications reengineering	Any model-based (Interface model based on ADVs)	No	Yes (ADV-charts)	No
[15]	Pattern-based GUI design method for MDE of RIAs	Legacy Web applications reengineering and RIAs development	Any UML model-based (interface model)	No	Yes (UML state machine diagrams)	MagicUWE

Table 1. Comparison of proposals for RIAs engineering

3 AlexandRIA's Code Generation Algorithm

Although there are several proposals related to RIAs development as is summarized in Table 1, there is no approach that entirely addresses multi-device RIAs. Furthermore, there is no proposal that covers the development of multi-device RIAs in an automatic or semi-automatic way. In order to achieve these drawbacks, we have proposed a code generation visual tool for multi-device RIAs development called AlexandRIA. The code generation approach implemented by AlexandRIA is a GUI pattern-

based approach like [15]; therefore, it is mainly focused on RIA's GUI details. However, it is inspired by some approaches such as [7], RUX-Method [8], UWE-R [6] and OOH4RIA [4] in order to incorporate high-level abstractions not only for designing rich GUIs but also for distributing business logic operations between client and server as well as for employing advanced client-server communication mechanisms. According to the classification of RIAs proposed by Preciado, Linaje, Sanchez, & Comai [24], not only standalone RIAs are covered by AlexandRIA but also distributed RIAs.

Considering that the demand for RIA technologies is driven by the increase in the development of cloud services APIs-based applications; our proposal is focused on generating cloud services APIs-based RIAs where the functionalities are implemented in terms of methods from cloud services APIs, i.e., in terms of server-side data and business logic operations. In this sense, business logic distribution is addressed at a domain-independent level where the operations at AlexandRIA-level, namely, data validation, data retrieving and data filtering can be distributed between client and server; it is important to notice that, mixed business logic operations are out of the scope of our proposal. On the other hand, data storage distribution is currently not addressed by AlexandRIA. As a proof concept, AlexandRIA uses cloud services APIs from popular Web 2.0 websites such as Twitter®, eBay® and Google Maps™, to mention but a few. In fact, as it is explained in section 4 of this work, the cloud services APIs underlying AlexandRIA are encapsulated by a set of predefined GUI/business logic components which are stored in a repository along diverse kinds of templates.

In detail, the code generation approach implemented by AlexandRIA comprises the following phases: 1) identifying the type of multi-device RIA to be generated as well as the target platform 2) defining the data and business logic operations by means of functionalities provided by cloud services APIs, 3) designing the GUI as an abstract composition of interaction design patterns, 4) linking GUI events to business logic operations at AlexandRIA-level in order to determine the distribution of these business logic operations between client and server, 4) refining the abstract GUI in order to obtain a concrete GUI 5) establishing the application configuration settings according to the type of multi-device RIA to be generated and the target platform 6) generating the source code starting from reusable GUI/business logic components encapsulating cloud services APIs and implementing interaction design patterns according to the concrete GUI, 7) compiling the source code and generating the executable code according to the target platform, if applicable. In detail, the GUI design method proposed as part of the code generation approach is based on the Abstract Data View (ADV) design model and it comprises two abstraction levels at two design levels: 1) abstract GUI at application's views level, where each view is modeled as a composed ADV made up of interaction design patterns modeled in turn as simple ADVs; here, the ADV design model is extended in order to allow developers to specify if the business logic operations at AlexandRIA-level, which are executed in response to certain GUI events must be executed either as client-side or server-side operations, 2) concrete GUI design at application's views level, which is achieved by simply adding look and feel details to each abstract GUI representing an application's view, 3) abstract GUI design at application level, where the application's GUI is modeled as a composed ADV made up of application views modeled in turn as composed ADVs and 4) concrete GUI design at application level, which is achieved by simply adding look and feel details to the application's concrete GUI.

Taking into account the functionality limitations, license issues and poor documentation of other available technologies for cross-platform RIAs development, we have selected Adobe® Flex 4.5 and PhoneGap® 1.0 for generating code of multi-device RIAs. Adobe® Flex is a free and open source framework for building Web, desktop and mobile RIAs with a common code base. It includes the MXML and ActionScript programming languages, which are compiled together into a single SWF file[a]. PhoneGap® is a free and open source framework that enables building native applications for mobile devices using JavaScript, HTML5 and CSS3 technologies[b]. Therefore, AlexandRIA is based on both ActionScript and JavaScript technologies. However, the code generation algorithm can be easily adapted and used with other related frameworks such as Microsoft® Silverlight™. The workflow of the AlexandRIA's code generation algorithm is explained below.

1. Input the required type of multi-device RIA, the target platforms, the set of cloud services APIs to be consumed (domain-specific business logic operations), the set of interaction design patterns to be implemented, the preferences related to distribution of the domain-independent business logic operations and the set of configuration settings. Here, only one of the following kinds of multi-device RIA can be generated: 1) Web browser-based RIA, 2) mobile Web browser-based RIA, i.e., a kind of Web browser-based RIA which is optimized to be displayed on mobile devices, 3) desktop RIA or 4) native (standalone) mobile RIA. For practical purposes, we will hereafter refer to multi-device RIA simply as RIA; similarly, we will hereafter to both Web browser-based RIA and mobile Web browser-based RIA as Web browser-based RIA. In addition, the following mobile platforms are supported by AlexandRIA: Android®, Apple® iOS, BlackBerry® Tablet OS and Windows® Phone. It is important notice that more than one mobile platform can be selected at a time. In this sense, if only mobile platforms supported by Adobe® AIR are selected, then only one set of source code files is generated, i.e., a set of multi-device source code files is generated. Finally, configuration settings include the following information: 1) properties independent of the type of multi-device RIA, 2) Web-specific properties 3) desktop-specific properties, 4) mobile-specific properties, 5) mobile platform-specific properties.
2. If a desktop RIA or a native mobile RIA is required then:
    a. Input the set of digital certificates needed to sign the installation files. To deploy both desktop RIAs and native mobile RIAs on target devices, installation files must be created. Installation files for mobile devices are commonly signed using digital certificates.
    b. Select the necessary configuration file templates according to both the required type of multi-device RIA and the target platforms.
3. Select the necessary application templates according to the required type of multi-device RIA and the target platforms.
4. Select the necessary GUI/business logic components according to the required cloud services APIs, the interaction design patterns and the target platforms.

[a] http://www.adobe.com/products/flex.html
[b] http://www.phonegap.com/

5. Create the appropriate application folder structure according to the required type of multi-device RIA.
6. Generate the source code files:
    a. Generate the application GUI/business logic files by instantiating the selected application templates and linking the selected GUI/business logic components.
    b. If a Web browser-based RIA is required then:
        i. Generate a server-side Web page that replicates client-side business logic operations according to the preferences related to distribution of business logic operations. As result, a PHP-based or JSP-based Web page is obtained.
    c. If a desktop RIA or a native mobile RIA is required then:
        i. Generate the application configuration files by instantiating the selected configuration file templates and linking the provided configuration settings.
7. Generate the native code files:
    a. If a Web browser-based RIA is required then:
        i. Generate a Web page wrapper (front-end) for the executable file. A simple HTML-based Web page embedding the SWF file is obtained.
    b. If a Web browser-based RIA, a desktop RIA or a native RIA for Android®, Apple® iOS and BlackBerry® Tablet OS-based devices is required then:
        i. Compile the source code files into an executable SWF file. SWF files run in a different way depending on the type of application.
        ii. Package and sign an installation file for each target platform by using the corresponding digital certificate. As result, air, apk, ipa and bar files can be obtained.
    c. If not:
        i. Package an installation file for Windows® Phone. As result, a xap file is obtained.
8. Output a ZIP file packaging the source code files within an application folder structure and the installation files, in the specific case of mobile RIAs.

4 AlexandRIA: Architecture and Functionality

Taking into account that the development of tools leveraging on high level primitives and code generation is a main research direction for RIAs engineering [24], we have developed a visual tool to automate the code generation of multi-device RIAs, which is a Web browser-based RIA itself. Accordingly to the Software Engineering Institute (SEI), software development organizations use tools to support many of the activities necessary to transform a set of customer needs into a useful product [25]. In this sense, the proposed tool called AlexandRIA is a tool for RAD that guides developers throughout a set of well-defined phases from a GUI pattern-based approach.

As a fundamental principle of RAD tools, code generators must generate source code from high-level constructs. AlexandRIA uses not only reusable GUI/business logic components as high-level constructs but also application and configuration file templates which can be reused according to the type of multi-device RIA to be generated and the target platform. In addition, RAD tools must generate executable code at least in a semi-automatic way [28]. AlexandRIA is in accordance with these principles; in fact, it automatically generates source and native code of multi-device RIAs, without requiring hand coding and much less requiring software modeling. According to the above, RAD tools allow saving development time, effort and budget enabling much higher productivity [29]. Furthermore, AlexandRIA automates a code generation approach which is inspired by some already existing RIAs development approaches in order to achieve the aforementioned goals; hence it may ensure less error-prone applications.

Furthermore, in RAD tools architecture, the core of the code generation automation is the repository. In this sense, AlexandRIA uses an Apache HTTP server-based repository which is mainly intended to store XML-based documents. Actually, the AlexandRIA's repository has a set of sub-repositories due to the diversity of XML-based documents to be stored.

AlexandRIA allows developers to generate both source and native code of: 1) Web browser-based RIAs 2) mobile Web browser-based RIAs 3) multiplatform desktop RIAs, and 4) native RIAs for mobile devices based on Android®, Apple® iOS, BlackBerry® Tablet OS and Windows® Phone platforms. Generated desktop RIAs are supported by Windows® and Mac OS® X operating systems.

4.1 Architecture Description

AlexandRIA has a layered design in order to organize its components. This layered design allows extensibility and easy maintenance because its tasks and responsibilities are distributed. In fact, maintenance is a major issue in the development of AlexandRIA; therefore, we propose a mechanism for easily adding and removing software components and templates from repositories.

The proposed architecture is shown in Figure 1. This architecture is composed of five layers; each layer has a specific function explained as follows:

- **Presentation:** This layer has a set of Graphic User Interfaces (GUIs) enabling the interaction between the users and the application by generating events on GUI controls. This set of GUIs is implemented in Adobe® Flex.
- **Integration:** This layer responds to user requests via GUI event handlers; i.e., it indicates to the services layer which services must be executed depending on requirements specified by users throughout the AlexandRIA's wizard phases. In this sense, user requirements are manipulated using XML-based documents. Furthermore, this layer is responsible for building the responses that are finally presented to users.
- **Services:** This layer implements the algorithm for generating source and native code of multi-device RIAs. It contains the code generation engine which includes class libraries for cross-platform RIAs frameworks such as Adobe® Flex 4.5 and PhoneGap® 1.0. Class libraries allow extending the code generation engine to other platforms. This layer also contains the maintenance engine which allows easily extending the functionalities provided by the applications to be generated. It is important to notice that, this layer is responsible for

retrieving the necessary XML-based documents, i.e., templates and software components, from repositories during code generation.

- **Data Access:** This layer represents the AlexandRIA's back-end. It has a set of PHP-based scripts acting as a template engine, i.e., it is responsible for instantiating templates during code generation. This layer is also responsible for managing non XML-based documents.
- **Data:** This layer has a set of GUI/business logic components providing functionalities to applications to be generated by means of cloud services APIs and a set of GUI/business logic components implementing interaction design patterns. These components are specialized according to the cross-platform RIA frameworks underlying to AlexandRIA. In this sense, it has both MXML/ActionScript-based components and HTML/JavaScript-based components. Cloud services APIs supported by AlexandRIA are: YouTube™ Data and YouTube™ Player, Google Custom Search™, Flickr®, Twitter® REST, eBay® Finding, Google Maps™, Last.fm®, Yahoo!® Weather and Delicious APIs. Similarly, AlexandRIA supports the following interaction design patterns proposed by Scott & Neil [26]: inline paging, auto complete, scrolled paging (carousel), virtual scrolling, refining search, virtual panning, drag and drop, list inlay, periodic refresh, brighten and dim (lightbox effect) and progress indication. This layer also has a set of application templates which are specialized in Adobe® Flex-based applications templates and PhoneGap®-based applications templates because of the cross-platform RIA frameworks underlying to AlexandRIA. AlexandRIA uses an application template for each type of multi-device RIA to be generated, i.e., for Web browser-based RIAs either desktop or mobile version, for desktop RIAs, and for native mobile RIAs. These application templates allow generating diverse kinds of multi-device RIAs covering a wide range of popular mobile operating systems. Besides the sub-repositories for XML-based documents, this layer has a sub-repository for storing digital certificate files. Furthermore, it also has a sub-repository for storing files generated by AlexandRIA as a result of source and native code generation processes.

4.2 Components Description

Each layer of the architecture contains a set of components whose functions are described below.

- **Wizard tool:** This component represents the source of the GUI events generated from the interaction between the final user and AlexandRIA. As Neil [27] suggests, rich design must be addressed from four design levels: 1) application structure, 2) screen layouts, 3) GUI controls and 4) user interactions. In order to guide the final user throughout a set of well-defined stages, this component was designed following the Wizard screen design pattern.
- **Maintenance GUI:** This component represents the source of the GUI events generated from the interaction between the administrator and AlexandRIA. In this sense, it allows administrator to provide and remove software components as well as application and configuration templates. It also allows removing resources already stored in the repositories.
- **Folder Structure Generator:** This component generates a folder structure based on the folder structure defined by the PPMRD process [11] for each application to be generated by AlexandRIA.

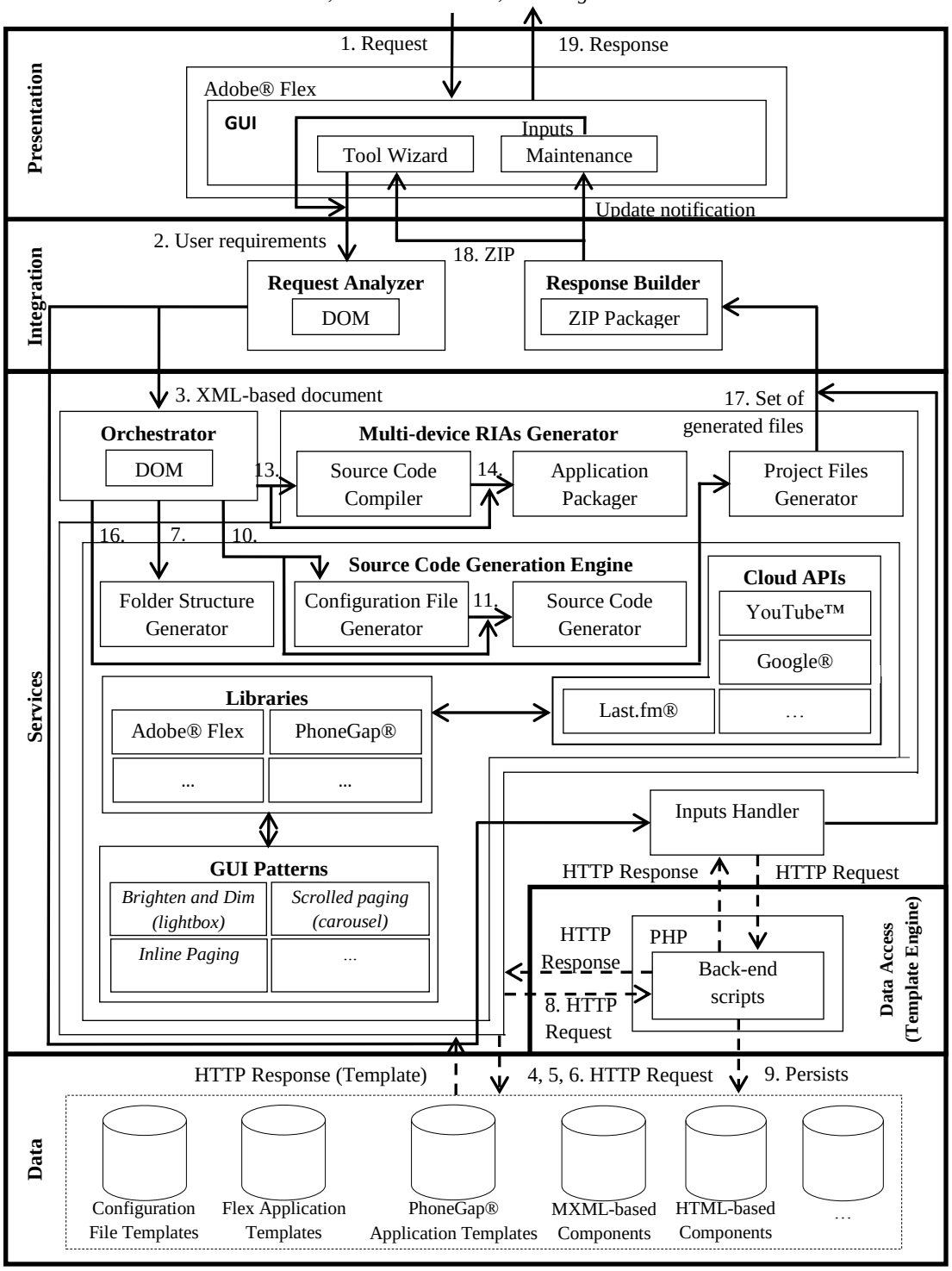

Figure 1. AlexandRIA Architecture

- **Request Analyzer:** This component listens to the user requests and determines the actions to be accordingly executed. In the particular case of code generation requests, this component generates XML-based documents for encapsulating the requirements specified by the users throughout the AlexandRIA's wizard phases, i.e., type of multi-device RIA, cloud services APIs, target platforms and configuration parameters.
- **Code Generation Orchestrator:** This component generates user preferences trees starting from XML-based documents encapsulating the requirements specified by users throughout the AlexandRIA's wizard phases; it also parses resulting XML trees and determines the services to be executed depending on the required type of multi-device RIA.
- **ZIP packager:** This component locates both the folder structure (source code files) and the installation files/wrapper (native code) into a single ZIP file which is presented to the user at the last phase of the AlexandRIA's wizard.
- **Configuration Files Generator:** In the case of Adobe® AIR-based applications, this component creates the corresponding AIR application descriptor file based on the parameters configured by the user. In the case of PhoneGap®-based applications, this component creates a configuration file which is an XML-based document used by the PhoneGap® Build cloud service for generating installation files.
- **Source Code Compiler:** This component invokes the Adobe® Flex application compiler to compile the generated MXML/ActionScript files (source code files) into SWF files.
- **Application Packager:** In the case of Adobe® AIR-based applications and depending on the target platforms indicated by the user, this component invokes either Adobe® AIR® SDK tools or Blackberry® SDK tools which allow packaging and digitally signing generated SWF files into installation files. In the particular case of Web browser-based applications, this component generates application wrappers using the Apache Ant tool. Besides, for packaging PhoneGap®-based applications this component uses the PhoneGap® Build cloud service.
- **Adobe® Flex Project Files Generator**: In the case of Adobe® Flex-based applications, this component generates the *.project*, *.actionScriptProperties* and *.flexProperties* files which are used by Adobe Flash® Builder® projects.
- **Source Code Generator:** This component primarily generates MXML/ActionScript or HTML/JavaScript source code files according to the required type of multi-device RIA, the target platforms, the interaction design patterns and the cloud services APIs selected by the user. This component is also responsible for generating PHP-based or JSP-based Web pages for server-side business logic operations.
- **Libraries:** This component has a set of class libraries for different cross-platform RIAs frameworks which allows generating source code for several target platforms.
- **Configuration File Templates Repository:** This repository has one configuration file template for each platform supported by AlexandRIA, i.e., there is one template for each platform supported by Adobe® AIR as well as one template for generating configuration files for Windows® Phone applications based on PhoneGap®. Configuration files for Adobe® AIR-based applications are known as AIR application descriptor files.

- **Back-end scripts:** This component has a set of scripts written in the PHP programming language which is responsible for performing low-level tasks related to the file system managing during code generation, e.g., files and folders writing, files and folders copying and execution of SDK command-line tools. In this sense, the *Back-end scripts* component is responsible for placing the generated source code files into the corresponding folder structures. In the case of Adobe® Flex-based applications, the folder structure must be composed of: 1) a folder called "src", which includes the application source code files such as MXML/ActionScript files, 2) a folder called "libs" that includes class libraries, i.e., SWC files and 3) a folder called "bin-debug", which includes files generated by the Adobe® Flex SDK compiler; it is worth mentioning that this folder structure is similar to the default folder structure used by Adobe® Flash® Builder® for creating projects. Furthermore, this component is responsible for managing digital certificates during native code generation.
- **Inputs handler:** This component is responsible for storing new resources into the corresponding repositories as well as removing already stored resources from repositories.
- **MXML/ActionScript-based Components Repository:** This repository enables persistence for the set of GUI/business logic components based on MXML and ActionScript.
- **HTML5/JavaScript-based Components Repository:** This repository enables persistence for the set of GUI/business logic components based on HTML5 and JavaScript technologies.
- **Flex Applications Templates Repository:** This repository enables persistence for Adobe® Flex-based applications templates, which are based on MXML and ActionScript technologies. There is one application template for each kind of Adobe® Flex-based application supported by AlexandRIA, i.e., for Web browser-based RIAs either desktop or mobile version, for desktop RIAs and for native mobile RIAs. Additionally, this repository enables persistence for Adobe® Flex project file templates.
- **PhoneGap Applications Templates Repository:** This repository enables persistence for PhoneGap®-based application templates, which are based on HTML5 and JavaScript technologies. There is one application template for each kind of PhoneGap®-based application supported by AlexandRIA, i.e., for Windows® Phone applications.
- **Certificate Files Repository:** This repository enables persistence for digital certificates files in PKCS12 format as well as for other resources provided by users. This repository also stores files generated by AlexandRIA during validation of certificate expiration dates, Certification Authorities (CAs) and passwords.
- **Output Repository:** This repository enables persistence for files generated by AlexandRIA as a result of source and native code generation processes. Basically, for each application required by the user, the folder structure proposed in the phase "creating the application folder structure" of PPMRD is generated and located in this repository.

4.3 Workflow Description

The relationships between the components of the AlexandRIA's architecture describe two main workflows in which the functions are executed in a sequentially way: 1) code generation and 2)

maintenance. Code generation workflow is composed of two main sub-workflows: 1) Web browser-based RIAs code generation and 2) code generation of both desktop RIAs and native mobile RIAs. These sub-workflows are triggered starting from phase 4 of the AlexandRIA's wizard.

The phases of both main sub-workflows to be processed during the execution of the tool are described below through a consecutive numbering; these sub-workflows describe the architecture's functionality which is depicted in Figure 2. For practical purposes, details about code generation for more than one mobile platform at a time are excluded.

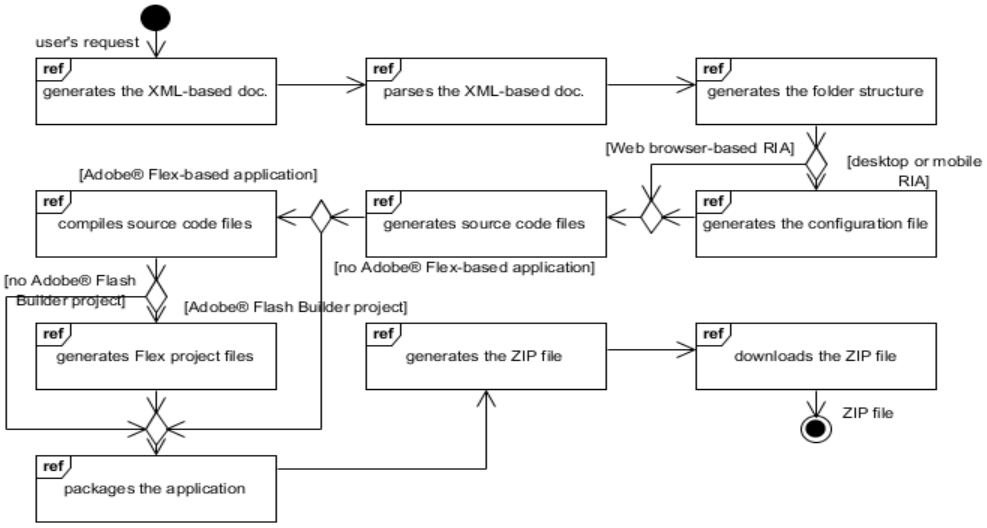

Figure 2. AlexandRIA's workflow

1. The user requests the generation of a multi-device RIA.
2. The *Tool wizard* component triggers a special GUI event when the user completes its phase 4.
3. Once the event that indicates the completion of the phase 4 of the AlexandRIA's wizard is identified by the *request analyzer* component, an XML-based document is generated for encapsulating the requirements specified by the user throughout the first three phases. Finally, this XML-based document is sent to the *code generation orchestrator* component.
4. The *code generation orchestrator* component creates the user preferences tree starting from the incoming XML-based document. This XML tree is parsed in order to identify the services that must be executed. Initially, if a desktop RIA or a native mobile RIA is required, then the *code generation orchestrator* component sends an HTTP request to the *configuration file templates repository* in order to retrieve the necessary *AIR application descriptor file template* or the *PhoneGap® configuration file template* according to both the required type of multi-device RIA and the target platform.
5. Once the appropriate configuration file template is retrieved, if a Web browser-based RIA, a desktop RIA or a native RIA for Android®, Apple® iOS or BlackBerry® Tablet OS-based mobile devices is required, then the *code generation orchestrator* component sends an HTTP request to the *Flex applications templates repository* in order to retrieve the appropriate

application template according to the required type of multi-device RIA. Besides, if a RIA for Windows® Phone-based devices is required, then the *code generation orchestrator* component sends an HTTP request to the *PhoneGap applications templates repository* for retrieving the corresponding *PhoneGap®-based application template*. This component analyzes the user preferences tree to identify the amount of cloud services APIs required by the user. AlexandRIA uses an application template for each type of multi-device RIA to be generated. Furthermore, it uses two special types of templates for Web browser-based and desktop RIAs that includes more than one cloud services API, these special kinds of template uses an Adobe® Flex *StackView* navigator component and an Adobe® Flex *TabNavigator* navigator component respectively. In this sense, in the case of Web browser-based and desktop RIAs code generation, if more than one cloud services API is required, then either a view-based application template or a tabbed application template is used, otherwise a single-view application template is used. Similarly, AlexandRIA uses these two kinds of template for generating mobile RIAs regardless of the amount of cloud services APIs required by the user. For instance, the view-based application template for Adobe® Flex-based mobile applications uses an Adobe® Flex *ViewNavigatorApplication* container component which implements a view-based navigation model. This approach allows designing mobile applications that are composed of a set of full-screen views, each of them is implemented by an Adobe® Flex *View* container component. Each application's view links an MXML/ActionScript component that implements just one cloud services API. Switching between views is controlled by an Adobe® Flex *ActionBar* control to which is added an Adobe® Flex *Button* control for each application view. Moreover, the view-based application template for PhoneGap®-based applications uses a div (HTML <*div*> tag) based layout for setting up an HTML external page for each application view. Each HTML external page links an HTML5/JavaScript-based GUI/business logic component implementing just one cloud services API by means of interaction design patterns, i.e., other HTML5/JavaScript–based GUI/business logic components. Switching between pages is allowed by using a tab-bar based on an HTML list (HTML <*ul*> tag).

6. Once the appropriate application template is retrieved, if a Web browser-based RIA, a desktop RIA or a native RIA for Android®, Apple® iOS or BlackBerry® Tablet OS-based mobile devices and the generation of an Adobe® Flash Builder 4.5 compatible project is required, then the *code generation orchestrator* component sends a series of HTTP requests to the *Flex applications templates repository* in order to retrieve the *.project*, *.actionScriptProperties* and *.flexProperties* file templates.

7. Once the Adobe® Flex project file templates are retrieved, the *code generation orchestrator* component calls the *folder structure generator* component.

8. The *folder structure generator* component sends an HTTP request to the *back-end scripts* component for generating the folder structure for the required multi-device RIA.

9. The *back-end scripts* component analyzes the incoming HTTP request in order to identify the requested low-level task. In the case of folder structure generation, this component creates a series of folders within the *output repository*. Finally, the *back-end scripts* component returns a value that indicates the successful or failed execution of the low-level operation.

10. If the value returned by the *back-end scripts* component indicates a successful operation and an Adobe® AIR-based application or a PhoneGap®-based application is required, then the *configuration file generator* component is called. Besides, if the value returned by the *back-end scripts* component indicates a successful operation and a Web browser-based application is required, then the *source code generator* component is directly called (phase 11).

11. The *configuration file generator* component extracts the configuration parameters contained in the user preferences tree and it uses them for creating a configuration file based on the previously retrieved configuration file template. The *configuration file generator* component sends an HTTP request to the *back-end scripts* component for writing the corresponding file. The *back-end scripts* component returns a value indicating a successful or failed operation. If the value returned by the *back-end scripts* component indicates a successful execution of the low-level operation, then the *source code generator* component is invoked.

12. Starting from the application template initially retrieved, the interaction design patterns to be implemented and the required cloud services APIs, the corresponding *MXML/ActionScript-based components* or *HTML/JavaScript-based components* are linked to the application templates. Finally, this component sends a series of HTTP requests to the *back-end scripts* component in order to: 1) write the corresponding client-side source code files using the applications templates, 2) copy the required *MXML/ActionScript-based components* or *HTML/JavaScript-based components* and 3) copy the libraries required for the linked components. In addition, the PHP-based or JSP-based Web page replicating client-side business logic operations according to the preferences related to distribution of business logic operations is generated, if necessary. All files generated and copied are located according to its kind within the appropriate folders of the folder structure previously created; e.g., the libraries are located into the *libs* folder whereas the server-side Web page is located within the root folder. If the operation performed by the *back-end scripts* component is successful, then the control is returned to the *code generation orchestrator* component.

13. Once the source code files are generated, if a Web browser-based RIA, a desktop RIA or a native RIA for Android®, Apple® iOS and BlackBerry® Tablet OS-based mobile devices is required, the *code generation orchestrator* component invokes the *source code compiler* component in order to validate the generated source code files. Besides, if a mobile RIA for Windows® Phone-based devices is required, then the *application packager* component is directly called (phase 15).

14. The *source code compiler* component sends an HTTP request to the *back-end scripts* component for requesting the execution of the appropriate SDK command-line tool that allows obtaining the executable code. In the case of Adobe® Flex-based applications, the *mxmlc* tool for compiling Web browser-based applications or the *amxmlc* tool for compiling Adobe® AIR-based applications are used. The *amxmlc* tool invokes the standard *mxmlc* compiler with an additional parameter. As outcome, the *back-end script* component generates the SWF file which is located within the *bin-debug* folder of the application folder structure. If the *back-end scripts* component indicates that the executable code was successfully generated, then the *source code compiler* component calls the *application packager* component.

15. The *application packager* component sends an HTTP request to the *back-end scripts* component for requesting the execution of the appropriate framework SDK command-line tool that allows generating the corresponding installation files. The necessary command-line tool is chosen according to both the required type of multi-device RIA and the target platform. If a desktop RIA or a native RIA for Android® or Apple® iOS-based mobile devices is required, then the Adobe® Flex ADT command-line tool is invoked. If a RIA for BlackBerry® Tablet OS-based mobile devices is required, then the BlackBerry® SDK *blackberry-air-packager* command-line tool is invoked. In the case of Web browser-based RIAs, this component generates an application wrapper for the previously generated SWF file. In the case of RIAs for Windows® Phone-based mobile devices, this component sends an HTTP request to the PhoneGap® Build cloud service. As outcome, an apk, ipa, bar, xap or html file is generated by the *back-end scripts* component and it is located within the root folder of the application folder structure. The *back-end scripts* component returns a value indicating a successful or failed operation. If the installation file or the application wrapper was successfully generated, then the control is returned to the *code generation orchestrator* component.
16. Once the installation file is generated, if the user required the generation of an Adobe® Flash Builder 4.5 project, the *code generation orchestrator* component invokes the *project files generator* component.
17. The *project files generator* component sends a series of HTTP requests to the *back-end scripts* component for requesting the generation of the *.project*, *.actionScriptProperties* and *.flexProperties* files. These files are generated in a sequential way and are located within the root folder of the application folder structure. The *back-end scripts* component returns a value indicating a successful or failed operation. If the project files were successfully generated, then the *response builder* component is invoked.
18. The *response builder* component sends an HTTP request to the *back-end scripts* component in order to package the source code folder structure and the installation file/wrapper into a single ZIP file which is downloaded from the Apache HTTP server at the last phase of the AlexandRIA's wizard. The *back-end scripts* component returns a value indicating a successful or failed operation. If the *back-end scripts* component indicates that the ZIP file was successfully generated, then the *tool wizard* component is invoked.
19. The *Tool wizard* component displays an option which allows developers to download the previously generated ZIP file once the code generation workflow is completed.
20. The user receives a reply to the request initially made.

In order to explain the AlexandRIA's functionality, a case study for generating code of two cloud services APIs-based multi-device RIAs is presented below.

5 Case Study: Generating cloud services APIs-based multi-device RIAs

AlexandRIA is a code generator that allows developers to generate both source and native code of cloud services APIs-based multi-device RIAs following a GUI pattern-based approach in a step-by-step way using a wizard. AlexandRIA is based on principles of RAD tools; in this sense, it uses a set of

reusable software components and templates [30] as high level constructs. These components implement functionalities provided by popular cloud services APIs such as Twitter® REST, Flickr® and Google Custom Search™ APIs, to mention but a few (see section 4 of this paper). For generating a multi-device RIA following the AlexandRIA's wizard is necessary that the user selects at least the type of multi-device RIA to be generated and the cloud services APIs to be consumed; finally, the application configuration must be specified. To explain the AlexandRIA's functionality, a case study for generating both a mobile Web browser-based RIA distributing business logic operations between client and server and a standalone RIA for multiple mobile devices using the Flickr®, Last.fm® as well as YouTube™ Data and YouTube™ Player cloud services APIs is presented below.

Let us suppose that a non-experienced developer needs to develop the same RIA for both mobile Web browsers (cross-browser) and different Android® and Apple® iOS-based devices (multi-device) using cloud services APIs provided by diverse Web 2.0 websites such as Flickr®, Last.fm® and YouTube™. In this context, there is a main constraint: the developer has no experience in the development of cloud services APIs-based applications; moreover, the developer does not have enough time for analyzing the required APIs and determining the methods that must be implemented. As a solution to this issue, AlexandRIA provides: 1) a set of reusable cloud services APIs-based GUI/ business logic components, 2) a set of Adobe® Flex-based applications templates and PhoneGap®-based applications templates and 3) a code generation engine which allows generating source and native code of cloud services APIs-based RIAs for different mobile Web browsers, platforms and devices following the steps described below:

1. The user accesses AlexandRIA via a Web browser by using a URL.
2. Once AlexandRIA is completely rendered, the user can select both the type of multi-device RIA to be generated and one or more target platforms. Here, only one of the following kinds of RIA can be selected: Web browser-based RIA, mobile Web browser-based RIA, desktop RIA or native (standalone) mobile RIA. In addition, possible target platforms for native mobile RIAs are Android®, Apple® iOS, BlackBerry® Tablet OS and Windows® Phone whereas desktop RIAs are multi-platform and Web browser-based RIAs are cross-browser, so that there are no options related to target platforms for these two kinds of multi-device RIA.
3. The next phase is the selection of both the cloud services APIs to be used and the interaction design patterns to be implemented by means of an abstract GUI design. As is explained in section 4 of this work, we have developed both a MXML/ActionScript-based GUI/business logic component and a HTML/JavaScript-based GUI/business logic component for each cloud services API and interaction design pattern initially considered by AlexandRIA. These components are reused irrespective of the types of multi-device RIA offered by AlexandRIA; i.e., they are multi-device themselves. Figure 3 depicts both the set of cloud services APIs supported by AlexandRIA and the cloud services APIs selected in this case study. Moreover, these components were designed using the ADV design model as is proposed in ([1];[5];[14]). In fact, when the user selects a cloud services API, a pre-built ADV model is displayed representing the application's view to be implemented. This ADV model is composed of the ADV models representing the interaction design patterns to be implemented by default as is depicted in Figure 4.

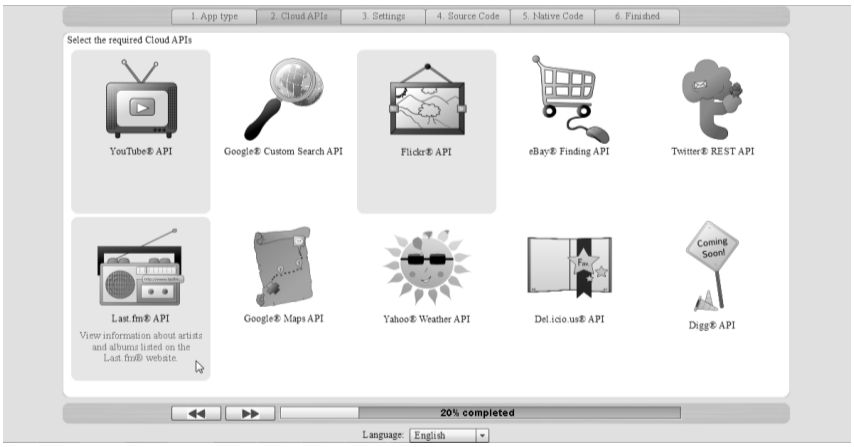

Figure 3. Selecting the required cloud services APIs

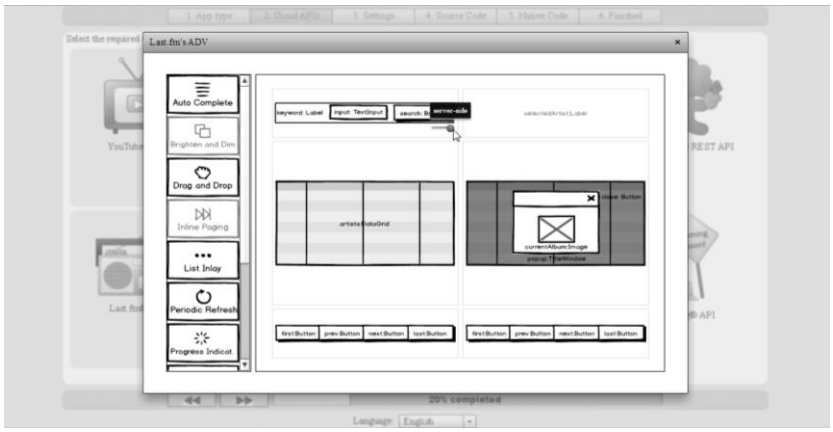

Figure 4. Designing the abstract GUI

Optionally, the user can customize the application's view by selecting other interaction design patterns from a palette including pre-built ADV models. In addition, the user can configure the distribution of the business logic operations behind the GUI components composing the interaction design pattern ADV models by labeling them either as "server-side" or "client-side" operations. It is important to notice that, only the operations at AlexandRIA-level, namely input data validation, cloud services data retrieving and cloud services data filtering, can be configured. Otherwise, these operations are implemented as client-side operations by default. As an example, the Last.fm® API selected in this case study provides two different methods that allow developers to search for an artist by name and get the top albums for an artist on Last.fm®, respectively. The pre-built AlexandRIA's "Lfm" component implements these functionalities by using the *inline paging* and *brighten and dim* interaction design patterns as is depicted in Figure 4. In this case, the notation used by the ADV is inspired by the names of the Adobe® Flex controls that are being represented by the ADV's elements. In detail, this component provides pagination of artist results; at the same time, it allows

navigating the top albums of the artist selected from the current page of artist results as well as displaying a modal pop-up dialog box including a full-size image of the selected thumbnail album. This functionality proves that applications generated by AlexandRIA conform not only to RIA's capabilities related to rich presentation but also to capabilities related to sophisticated client-server communication, especially the feature of retrieving data from different two or more simultaneous sources as is outlined in [22]. It is important to notice that, this feature is not set against the visual continuity of the GUI as is required by RIAs; in this case study, artist results and album results are refreshed independently of one other. Furthermore, this pre-built component can be customized by replacing the *inline paging* interaction design pattern by the *scrolled paging* pattern, for example, in order to navigate the top albums of the selected artist.

4. Once the required cloud services APIs and interaction design patterns are selected, the application configuration is settled according to the RIA type. In the case of code generation of native mobile RIAs, the parameters to be configured in this phase are: 1) parameters applicable to all offered mobile platforms, i.e., multi-platform parameters such as the "launching in a full-screen mode" and the "screen auto-orienting" features and 2) properties only applicable to the required target platforms such as the iPhone®/iPad® support in the case of applications for Apple® iOS-based devices, or the application install location in the case of applications for Android®-based devices. There are also parameters applicable to both kinds of standalone RIA, i.e., parameters applicable to native mobile RIAs and desktop RIAs, such as the application version number. Furthermore, there are parameters applicable to all RIA types, i.e., multi-device parameters such as 1) the application name, 2) the application title, 3) the DPI measure which allows automatically scaling the application for different screen densities, where density is the number of dots or pixels per square inch; this feature is a decisive factor in the development of density-independent applications for mobile devices, 4) the application template which can be a view-based template or a tabbed template; as can be inferred, the GUI design follows a bottom-up approach, 5) the application style which involves concrete details of the application's look and feel such as the background and fonts colors for the template. It is important to notice that, concrete GUI design at application's view level is not currently addressed by AlexandRIA. Finally, there is a parameter only applicable to Web browser-based RIAs either mobile or desktop versions, the programming language to be used for generating the server-side Web page comprising the server-side business logic; AlexandRIA currently supports PHP and JSP server-side technologies. In short, the user must indicate at least the application name and title; the other parameters are optional and can take default values. The settings defined in this case study for the multi-device parameters as well as for the parameters only applicable to each kind of RIA are depicted in Figure 5 and Figure 6, respectively.

   It is worth mentioning that, the look and feel of the Adobe® Flex-based applications generated by AlexandRIA is defined by means of Spark skins. A Spark skin is a MXML class controlling all visual elements of a component of the Adobe® Flex Spark architecture including layout.

5. Once the application configuration is finished, AlexandRIA displays a summary of the requirements specified by the user throughout the AlexandRIA's wizard phases, and it allows generating the application source code either as simple source code files or as an Adobe®

Flash Builder 4.5 project in the case of Adobe® Flex-based applications code generation. Here the application folder structure is generated and the source code generation process is immediately triggered.

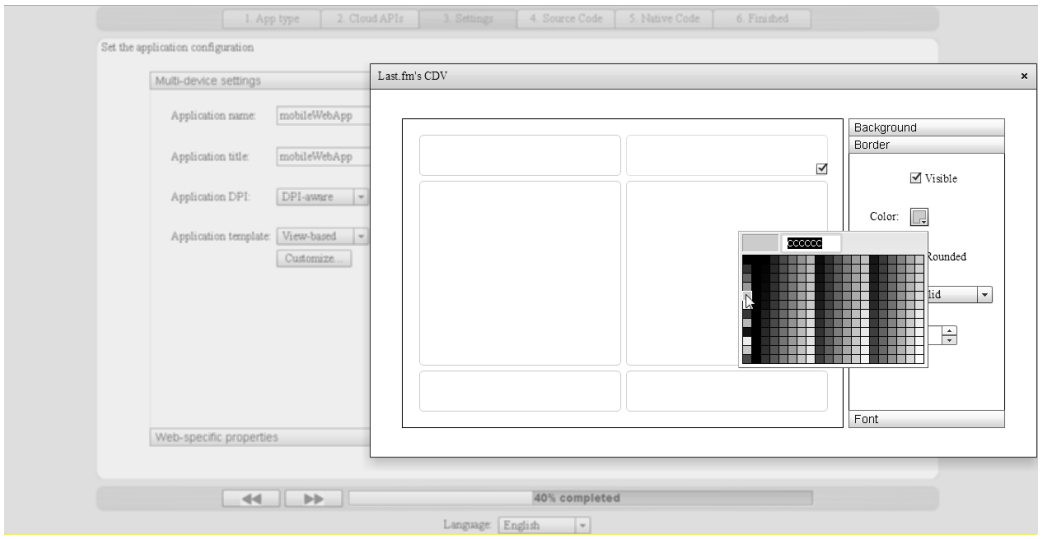

Figure 5. Designing the concrete GUI as part of the multi-device parameters setting

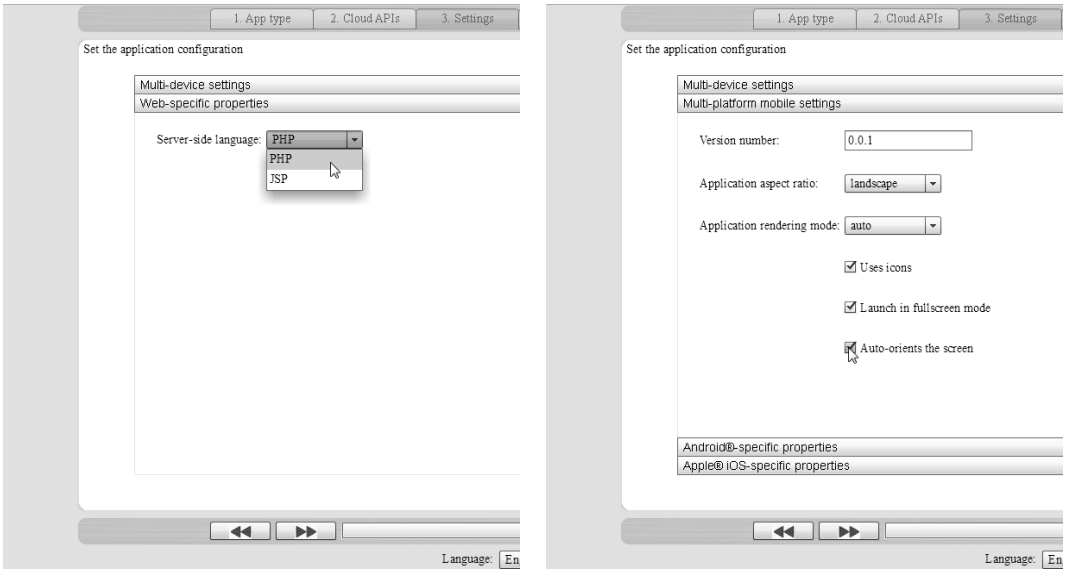

Figure 6. a) Setting the parameters specific to Web browser-based RIAs, b) Setting the parameters only applicable to mobile RIAs

6. The next phase is the generation of the corresponding native code. In the case of code generation of native mobile RIAs, AlexandRIA asks for the digital certificate files and

corresponding passwords needed to sign an installation file for each required target platform. In the specific case of code generation of applications for Apple® iOS-based devices also a provisioning profile must be provided. A provisioning profile is a file which associates an Apple® iOS-based device with a digital certificate and with a previously registered Apple® iOS application. This file can be created in the Apple® iOS provisioning portal. In order to ensure the success of the native code generation process, AlexandRIA performs real-time validations of digital certificates such as password validations and expiration date validations. Furthermore, AlexandRIA performs certification authority identity validations in the specific case of code generation of native mobile applications for Apple® iOS-based devices. In any case, once all required resources are provided and validated, AlexandRIA displays the "generate native code" option. Once the "generate" option is selected by the user, the native code generation process is triggered. A series of information messages is displayed while the native code is being generated. In the case of Web browser-based RIAs code generation, the application wrapper is generated in this wizard phase.

7. The last wizard phase is the automatic generation of the ZIP file that packages the previously generated source and native code. Once the ZIP file is generated, a "download ZIP file" option is displayed. The user downloads the ZIP file by clicking the "download ZIP file" option. Here, the AlexandRIA's wizard is finished. The user must unpack the ZIP file in order to obtain the application folder structure. The folder structure of native mobile RIAs contains multi-device source code files as well as the required installation file which are located in the root folder. In this case study, an ipa file and an apk file are provided because both Apple® iOS and Android® were selected as target platforms. On the other hand, besides source code files, the folder structure of mobile Web browser-based RIAs contains both an HTML-based Web page (wrapper) and a server-side Web page (back-end). In this case study, a PHP-based Web page is provided in order to implement server-side business logic. As can be inferred, Web-based RIAs generated by AlexandRIA must be executed by using an HTTP server like Apache HTTP Server or a servlet container like Apache Tomcat. Figure 7 depicts the native mobile RIA running on a Motorola XOOM™ tablet computer. Figure 8 depicts the execution of the previously generated mobile Web browser-based RIA using the Firefox Web browser on the same tablet computer.

The folder structures generated by AlexandRIA are compatible with the default folder structure of the Adobe Flash® Builder® 4.5 projects, they also contain the *.actionScriptProperties*, *.flexProperties* and *.project* files, which are used for these projects; therefore, they can be recognized as Adobe Flash® Builder® 4.5 projects. In this sense, to import a folder structure generated by AlexandRIA into Adobe Flash® Builder® 4.5, the "Import" option in the "File" menu must be used; as result, the folder structure is located in the projects explorer. Once the project has been imported, the source code can be edited and recompiled; the resultant application can be finally executed either on a device emulator, if applicable, or on the target device.

For example, if the developer needs to customize the look and feel of the native RIA generated in this case study, namely, the color of the selected row of the *DataGrid* controls, then the appropriate skin file must be edited. For this purpose, the developer must locate and open the "DataGridSkin.mxml" file, which is located within the "skins" folder of the project folder structure.

Once the application look and feel has been customized, the user can recompile the source code. The resultant application can be executed on a device emulator such as a Motorola XOOM™ emulator for immediately viewing the changes applied; additionally, installation files for other target platforms can be generated using the "Export release build" wizard of Adobe Flash® Builder® 4.5.

Figure 7. Screenshot of the native mobile RIA running on a Motorola XOOM™ tablet computer

Figure 8. Screenshot of the previously generated mobile Web browser-based RIA running on the Firefox Web browser on a Motorola XOOM® tablet computer

In order to validate the correct functioning of AlexandRIA, a set of complementary case studies for generating desktop RIAs with the same functionality of the RIAs generated can be easily implemented. Additionally, we have implemented multi-device RIAs based on cloud services APIs different to the cloud services APIs used in this case study, e.g., Yahoo!® Weather and Google Maps™ APIs. These multi-device RIAs are being tested also on a BlackBerry® PlayBook™ tablet computer.

6 Evaluation and Results

Software Engineering literature has proposed several works related to software development methods and tools. Although the absence of a standardized way to validate these proposals is still an issue, there are two main approaches: quantitative (objective) evaluations and qualitative (subjective) evaluations. One approach applicable to the assessment of both software engineering methods and tools is described in [31]. This work presented a method called DESMET which distinguishes between quantitative, qualitative and hybrid evaluations. It is aimed at performing comparative assessments of specific methods/tools in specific circumstances. According to DESMET, quantitative evaluations are based on identifying in measurable terms the effects of using a method/tool; in contrast, qualitative evaluations, also known as feature analysis, are aimed at establishing method/tool appropriateness in terms of the features provided by the method/tool. In the specific case of software development tools, the above mentioned proposal goes beyond the assessment of quality or usability. In fact, it is a framework for identifying the best way of performing an engineering task in specific circumstances among several alternatives. In this sense, we have proposed a new comparative evaluation method based on DESMET and focused not only on the assessment of AlexandRIA but also on the assessment of the code generated by using it.

In order to evaluate the suitability of AlexandRIA for generating source and native code of RIAs, it is necessary to consider the coverage of the features typically required by RIAs. In this sense, the suitability of some existing Web, Hypermedia and Multimedia methodologies to develop RIAs was surveyed by Preciado, Linaje, Sanchez, & Comai [22] nearly a decade ago. In this work, the suitability degree was established starting from the definition of the standard features of RIAs. In detail, the suitability of each methodology to support each RIA feature was determined in terms of a coverage degree, i.e., a qualitative score indicating how wide is the support provided. At the present time, the RIA features outlined in [22] are widely accepted. Furthermore, they are typically clustered into four categories related to the four main issues of RIAs development, namely, 1) rich GUI, 2) client-side and server-side data storage, 3) client-side and server-side business logic and 4) client-server communication.

The above analyzed evaluation method starts from a series of features typically required by RIAs and not from the features of the software methodologies evaluated. Thus, it can be also applied to the measurement of the suitability of software tools for RIAs development because they must support the implementation of the same features. Moreover, with the aim of evaluating the compliance with general principles of code generation, we have analyzed the code generation literature ([32];[33];[34]). Herrington [32] proposed a set of rules for designing, developing, deploying and maintaining code generators. This work is focused on the framework-based development approach, and it presents the common code generation solutions under this approach, namely, GUI generators, business logic

generators, database generators, Web services generators, documentation generators and unit tests generators. Furthermore, it outlines the workflows of different code generation techniques, covering their implementation in different programming languages. Vogel [33] presented a set of best practices in code generation. Starting from a generic code generation workflow, this work describes the variety of tools needed to implement code generation solutions in Microsoft® .NET by integrating with the Microsoft® Visual Studio IDE. In fact, this work is also focused on the framework-based development approach. Finally, Gunderloy [34] presented a set of essential skills for software developers working alone or in small projects. In this sense, this work treats source code generation as an essential skill for the framework-based development approach, and it outlines guidelines for deciding when and why to use commercial code generation tools for small software projects.

The above discussed works are related to the framework-based development approach and not to the systematic development approach. However, they outline general guidelines which are also applicable to the design, development and evaluation of code generation tools for the systematic development approach, especially for the systematic development of RIAs.

In order to evaluate the source and native code generated by AlexandRIA, we have analyzed some well-known models for software product quality such as McCall, ISO/IEC 9126 and Domey's quality models [35]. In general, these models identify a set of characteristics of software quality, and for each feature they define one or more software metrics. However, unlike the other proposals, the ISO/IEC 9126 is a widely recognized international standard providing a common language related to software product quality. In detail, the ISO/IEC 9126 is composed of four sections. The ISO/IEC 9126-1 section describes a two-part model for software product quality: 1) internal and external quality and 2) quality in use. This quality model is defined by means of general characteristics of software, which are further decomposed into general sub-characteristics, which in turn are derived into specific attributes. The internal quality is the totality of software attributes that can be measured during the development process whereas the external quality is the totality of software attributes that can be measured when the software is executed in a testing environment. The quality in use, on the other hand, reflects the user's view of the quality. The ISO/IEC 9126-2, ISO/IEC 9126-3 and ISO/IEC 9126-4 sections describes the software metrics for external, internal and quality in use attributes, respectively. Due to the generic nature of the ISO/IEC 9126 quality model, it needs to be refined according to the kind of software to be developed. In fact, because the attributes are the lower-level quality factors, they are not defined by the standard. These software quality models could be adapted and used to validate the applications generated by AlexandRIA, which could cause incorporating quantitative measures into the evaluation method because the software quality models are focused on this kind of measurement.

Finally, with the aim of evaluating the AlexandRIA's usability, we have analyzed several proposals aimed at assessing the usability of software products ([36];[37];[38];[39]). The ISO 9241-11section [36] of the ISO 9241 standard, for instance, defines usability as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. It also explains how to identify the information related to the context of use of a product which is necessary to take into account when specifying, designing or evaluating its usability in terms of measures of user performance (effectiveness and efficiency) and satisfaction. The guidance is given in the form of general principles and techniques, rather than in the form of specific methods. Under this approach, user performance is typically derived from objective measures, whereas

user satisfaction is derived from subjective measures. Because almost all the factors integrating the ISO's definition of usability, namely, effectiveness and satisfaction, corresponds to the software characteristics integrating the quality in use model defined by the ISO/IEC 9126-4 standard, the usability of a software product can also be measured as its quality in use. Aquino, Vanderdonckt, Condori-Fernández, Dieste, & Pastor [37] proposed a method for performing exploratory usability evaluations of multi-device and multi-platform GUIs generated using MDE approaches. This method is based on the ISO's definition of usability; and it is structured according to the template provided by the Goal Question Metric method. Lewis [38] proposed four psychometric characteristics-based questionnaires for measuring user satisfaction with system usability. Galitz [39] defined a set of general principles for designing and implementing GUIs, this compendium includes guidelines derived from various existing proposals including the W3C's design principles [40].

Although the first one proposal applies to the usability evaluation of products (hardware, software and services) related to office work with visual display terminals, it can also apply in other situations where a user is interacting with a product to achieve goals. In fact, the ISO 9241-11 definition of usability is generic; therefore, it can be adapted to the context of software development tools. As can be inferred from the above analyzed proposals, due to its hybrid nature, usability can be assessed using diverse techniques: 1) design principles and guidelines, 2) software metrics and 3) questionnaires capturing the user's perception for the usability aspects. Thus, both objective measures and subjective measures are used to this aim.

Because it is quite difficult to evaluate in a fully-quantitative way how legitimate is the solution provided respect to the aspects previously discussed, we have decided to use a two-part evaluation: 1) a qualitative method to measure diverse aspects of AlexandRIA, not only focusing on its usability but also on the support for the standard features of RIAs as well as on the compliance with the principles of code generation tools and 2) a quantitative method to measure the quality of the generated source and native code. Furthermore, in order to provide a comparative evaluation among a given set of tools for RIAs development, we have selected the RUX-Tool and AIR® Launchpad tools which were analyzed in section 2 of this paper. It is important to notice that, although Launchpad and AlexandRIA can be categorized as RAD code generators whereas RUX-Tool is an MDD tool, both are model-driven code generators in the sense that the code generation process starts from an abstract definition derived either from a software model or a user preferences tree.

For the qualitative evaluation method, on the one hand, a set of twelve desired tool features were identified. The presence of them in all the selected tools was assessed in order to determine the legitimacy of AlexandRIA and, at the same time, identify which of the alternatives is best in specific circumstances. Because this analysis requires subjective measures, a discussion is presented with the aim of justifying the results. It is important to notice that the tool features to be evaluated have been selected by this research group whereas the evaluation has been conducted by an external team composed of a software engineer, a software developer and a graphic designer. For the quantitative evaluation method, on the other hand, a set of six software metrics based on the internal metrics defined by the ISO/IEC 9126 standard were proposed. In addition, a basic code generation scenario was implemented using each of the selected code generation tools for RIAs development, so that the proposed metrics were employed under the same specific circumstances. In this case, the evaluation

was conducted by this research group because the code generation scenario does not contemplate the execution of the generated applications neither on a development environment nor a user environment.

6.1 Code Generation Tool Evaluation

6.1.1 Evaluation Design

This assessment method describes each of the three aspects that constitute the needs of a RIA developer; it also describes the features that a tool for RIAs development, in particular for RIAs code generation, must possess for satisfying each of the identified needs.

The scale used for the measurement of the identified aspects is a 3-point Likert scale [41] in which "1" represents the best score and "3" represents the worst score as follows:

- *3 points: strongly addressed (S.A.)*
- *2 points: partially addressed (P.A.)*
- *1 point: not addressed (N.A.)*

For each of the three aspects, the final score will be the highest score assigned by a member of the evaluation team. Finally, the overall evaluation for each software tool will be the sum of the final scores in the three aspects.

Need (N) 1: Support for the standard features of RIAs

The abstraction for the features distinguishing RIAs from traditional Web applications is the primary requirement of RIAs development tools. In this sense, we have considered the minimum necessary features proposed by Preciado, Linaje, Sanchez, & Comai [22] as follows: interaction design patterns (GUI transformations), visual continuity, temporal behavior, multimedia support, client-side and server-side business logic, client-side and server-side data storage, synchronous and asynchronous parallel requests to different sources and server-push. Moreover, the tool features ideally satisfying this aspect fits the "Language(s)/ Development environment" criteria of the evaluation proposed in [24], in the sense that they are abstract primitives supporting high-level specifications of RIAs.

Tool Feature (T.F.) 1. Support for rich GUIs: interaction design patterns, visual continuity, temporal behaviour and multimedia support

Tool Feature (T.F.) 2. Support for both client-side and server-side business logic

Tool Feature (T.F.) 3. Support for both client-side and server-side data storage

Tool Feature (T.F.) 4. Support for sophisticated client-server communication mechanisms: synchronous and asynchronous parallel requests to different sources, server-push

Need (N) 2: Compliance with the principles of code generation tools

The compliance with the principles of code generation tools, either RAD tools or MDD tools, is a major issue in this evaluation method. In this sense, we have considered the principles common among the code generation literature. Moreover, three of the five tool features ideally satisfying this aspect fit the other two criteria of the evaluation proposed by Toffetti et al. [24]. In detail, T.F. 1 and T.F. 2 fit

the "development process" criteria whereas T.F. 3 fits the "technology" criteria. In fact, T.F. 1 and T.F. 2 are indicators of how the resources involved in a RIA development project are covered, i.e., if the people responsible for carrying out the development activities are sufficiently engaged and whether other software tools are sufficiently involved.

Tool Feature (T.F.) 1. Integration within a RIAs development process

Tool Feature (T.F.) 2. Compatibility with external software tools such as IDEs and SDK tools

Tool Feature (T.F.) 3. Support for diverse frameworks and programming languages (Server-side and client-side)

Tool Feature (T.F.) 4. Version controlling

Tool Feature (T.F.) 5. Code generation engine extensibility

Need (N) 3: Usability

In order to evaluate the usability of AlexandRIA as is defined by the ISO 9241-11 standard [36], i.e. as the extent of effectiveness, efficiency and satisfaction with which a specified user can use it in an specified context to achieve specified goals, we have only considered the "satisfaction" factor because the evaluation of the user performance is not completely accurate for the purposes of this evaluation. In detail, the "satisfaction" factor has been assessed respect to the user's perception of usefulness as is proposed by Aquino et al. [37]. In particular, we have evaluated this aspect based on a set of tool facilities that can be used as indicators of the degree to which RIAs development projects can be improved in terms of time-saving and effort-saving which directly affects the developer productivity and indirectly affects the development team productivity [29].

Tool Feature (T.F.) 1. Generation of source code without depending on Web1.0 models from analysis and design activities

Tool Feature (T.F.) 2. Fully-automatic source code generation

Tool Feature (T.F.) 3. Fully-automatic executable code generation

6.1.2 Results

Table 2 and Figure 9 show to each aspect the final score obtained from the evaluation. Finally, the argumentation of the results for each aspect is discussed.

		AIR® Launchpad		RUX-Tool		AlexandRIA	
Aspect		**Score**	**Interpretation**	**Score**	**Interpretation**	**Score**	**Interpretation**
N.1	**T.F.1**	3	S.A.	3	S.A.	3	S.A.
	T.F.2	1	N.A.	2	P.A.	3	S.A.
	T.F.3	3	S.A.	1	N.A.	1	N.A.
	T.F.4	1	N.A.	2	N.A.	2	P.A.
N.2	**T.F.1**	1	N.A.	3	S.A.	1	N.A.
	T.F.2	3	S.A.	3	S.A.	3	S.A.
	T.F.3	2	P.A.	3	S.A.	3	S.A.
	T.F.4	1	N.A.	2	P.A.	1	N.A.
	T.F.5	1	N.A.	3	S.A.	3	S.A.
N.3	**T.F.1**	3	S.A.	1	N.A.	3	S.A.
	T.F.2	2	P.A.	3	S.A.	3	S.A.
	T.F.3	1	N.A.	1	N.A.	3	S.A.
Sum		**22**		**27**		**29**	

Table 2. Qualitative evaluation results

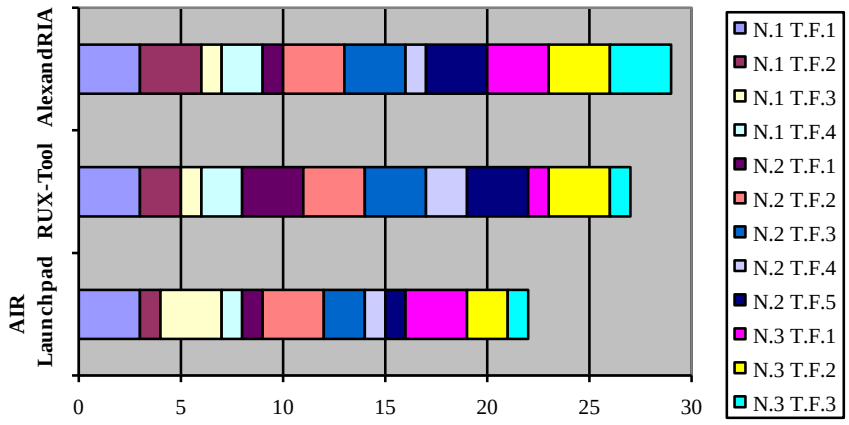

Figure 9. Qualitative evaluation results

Regarding to the compliance with code generation principles, the evaluation team has concluded that unlike AIR® Launchpad, RUX-Tool and AlexandRIA automate a RIAs systematic development approach; in detail, RUX-Tool implements a MDD design method defining three GUI specification levels: 1) abstract GUI design, 2) concrete GUI design and 3) final GUI design, where GUI is made up of GUI components at different levels of abstraction. In addition, it covers other aspects of RIAs development because it can be conceptually used with any Web methodology to obtain the data and business logic from the underlying Web 1.0 models. In this sense, each connection between RUX-Tool and a Web methodology requires the definition of a specific set of Extensible Stylesheet Language Transformations (XSLT) transformations. On the other hand, AlexandRIA implements a code generation approach for rapid development of RIAs, which has 7 stages covering from the

identification of the required type of multi-device RIA to the generation of the native code, if applicable. In both cases, only the design and implementation phases of RIAs development lifecycle are covered. Regarding to usefulness, the evaluation team has concluded that RUX-Tool and AlexandRIA are more useful than AIR® Launchpad because they conceptually cover multiple cross-platform RIAs frameworks. Currently, RUX-Tool and AlexandRIA have JavaScript (jQuery framework) as a common target RIA technology; furthermore, they have JSP as a common server-side technology. It is important to emphasize that RUX-Tool only covers Web browser-based RIAs code generation; in contrast, AlexandRIA automatically generates desktop RIAs and native mobile RIAs covering diverse desktop operating systems as well as a wide range of mobile operating systems and devices such as Windows® Phone-based and Apple® iOS-based mobile devices. Regarding to compatibility with external software tools, RUX-Tool ensures that the generated Web browser-based RIAs can be deployed on any application server compatible with the Java/JSP 2.0 standard. Moreover, both AIR® Launchpad and AlexandRIA ensure that the generated source code files can be imported into the Adobe® Flash Builder 4.5 IDE as projects. AlexandRIA also ensures that the generated server-side Web pages can be deployed either on any HTTP server interpreting PHP 5 scripts or on any application server compatible with the Java/JSP 2.0 standard, as appropriate; in addition, it ensures that generated installation files can be deployed on target operating systems using the appropriate runtime environment (Adobe® AIR). Regarding to the ability of the code generation engine to be extended, both RUX-Tool and AlexandRIA provide a means to extend the code generation engine; on the one hand, RUX-Tool defines two kinds of adaptation layers: 1) one called connection rules, which is aimed at catching the data and business logic operations from Web 1.0 models and adapting they to the RUX-Tool abstract GUI and 2) one called transformation rules, which is aimed at refining the abstract GUI to the other two abstraction levels. In this sense, the definition of a particular set of connection rules based on XSLT transformations is required in order to extend the code generation engine to a particular underlying Web methodology whereas the definition of a particular set of both abstract-to-concrete and concrete-to-final GUI transformation rules based on the eXtensible user Interface Components Language (XICL) is required in order to extend the code generation engine to a particular deployment platform, and consequently to a particular set of devices. On the other hand, AlexandRIA defines a mechanism to extend the code generation engine to other predefined sets of data and business logic operations by providing reusable GUI/business logic components encapsulating cloud services APIs. However, it also requires the definition of a particular kind of application templates and configuration file templates as well as the definition of the corresponding source code generation library in order to extend the code generation engine to other cross-platform RIA development frameworks, and consequently to other target devices. Finally, regarding to version control, the evaluation team has concluded that it may be useful for the purposes of maintaining multiple versions of a generated application; nevertheless, this principle is only addressed by RUX-Tool in the sense that the GUI code is generated within the folder structure of the corresponding backend (WebRatio) project, so that the RUX-Tool-generated code can be controlled by means of standard version control systems integrated into WebRatio as plugins.

Regarding to N. 3, the evaluation team has determined that only RUX-Tool requires analysis and design activities in order to obtain structural and hypertext WebML models by using WebRatio because these Web 1.0 models are a prerequisite for automatically obtaining the corresponding rich GUI model using RUX-Tool. Moreover, the tree evaluated tools allow obtaining source code without

requiring developers to write almost any code by hand. In fact, in the case of AIR® Launchpad, generated source code files are templates which can be used as a starting point for developing fully-functional applications; similarly, in the case of RUX-Tool, the modeling of specific or arbitrary business logic operations may require the definition of hand-written scripts. Finally, unlike AIR® Launchpad, both RUX-Tool and AlexandRIA allow automatically generating executable code without requiring developers to manually perform building, packaging or signing activities. The evaluation team has concluded that both AIR® Launchpad and AlexandRIA can save developer time and effort on developing RIAs, because these tools allow automatically generating RIAs source code based on a set of preferences selected through a wizard instead of from software models. Thus, only AlexandRIA allows developers generating source and native code of RIAs without requiring developers to carry out analysis, design, hand coding, building, packaging and signing activities. It is important to emphasize that, in the case of mobile RIAs code generation, most mobile operating systems requires that installation files be signed before deployment; in this sense, the executable code generated by AlexandRIA is signed during native code generation using digital certificates provided by developers. Additionally, AlexandRIA may save the time and effort required for analyzing cloud services APIs on developing cloud services APIs-based RIAs.

As can be inferred from the qualitative evaluation results, AlexandRIA completely addresses almost 70 percent of features that a tool for RIAs development must possess in order to satisfy the developer needs related to the support for the standard features of RIAs, the compliance with the principles of code generation tools and usability aspects. It rated 29 out of 36 in the qualitative part of the evaluation method we have proposed. According to the results, unlike RUX-Tool and AIR® Launchpad, AlexandRIA integrates high level primitives related to the definition of client-side business logic. In addition, AlexandRIA is the only RIAs code generation tool supporting native code generation of desktop and mobile RIAs (standalone) in a fully-automatic way, i.e. it covers digital signing activities.

6.2 Code Generated Evaluation

6.2.1 Evaluation Design

This assessment method is intended to measure the quality of the source and native code generated by the selected code generation tools, namely, RUX-Tool, AIR® Launchpad and AlexandRIA. For this purpose, we have selected the efficiency, maintainability and portability characteristics as well as the resource utilisation, changeability, analysability and adaptability sub-characteristics from the ISO/IEC 9126 standard. These sub-characteristics are related not only to software attributes relevant to multi-device RIAs but also to software attributes relevant to applications generated by using template-based code generation tools, which are ideally intended to be customized and extended. The quality factors to be assessed and the corresponding software metrics are described below.

Resource utilization: According to the quality model outlined by the ISO/IEC 9126 standard, the "resources utilisation" is a software quality factor. Formally, it is a component of the "efficiency" quality characteristic which can be interpreted as the ability of the software to execute the required tasks with the least amount of resources. In this sense, the quality of the data-intensive RIAs executing most of their business logic operations on the client-side can be adversely affected by the capacity of

the client-side resources, especially RAM. In addition, because mobile RIAs are aimed to be installed on mobile devices which typically have limited resources, the efficient usage of the available resources is a determining factor of their quality.

Changeability: The ISO/IEC 9126 standard defines the "changeability" as a factor affecting the software quality. In detail, it is a component of the "maintainability" quality characteristic which is defined as the ease with which a modification can be performed to the software. This is a determining factor on measuring the quality of the source code generated in a fully-automatic way, especially the source code generated by using template-based tools, which commonly do not generate fully-customized code but code based on predefined building blocks. As can be inferred, in this evaluation method the "changeability" is focused on the kind of maintenance related to both the implementation of new features and functionalities and the customization of the already implemented ones, i.e., the perfective maintenance. In this sense, the ease with which the GUI of an automatically generated multi-device RIA can be customized seems to be the most important scenario. It is important to notice that, according to [34], the possibility to extend the code generation engine (see N. 2 T.F. 5) rather than the generated code may be a deciding factor in the selection of a code generator.

Analysability: According to the quality model outlined by the ISO/IEC 9126 standard, the "analysability" is a factor affecting the software quality. In detail, as the "changeability", the "analysability" affects the "maintainability" quality characteristic. This factor reflects not only the maintainer´s spent effort on trying to diagnose for deficiencies or causes of failure but also the maintainer's spent effort for identification of parts to be modified in the software [34]. In the context of perfective maintenance stated in the above paragraph, the "analysability" can be considered a prerequisite of the "changeability" where the technical documentation plays a decisive role. In this sense, due to the nature of the software to be evaluated by means of this assessment method, only documentation related to the implementation phase of the software lifecycle is considered.

Adaptability: The ISO/IEC 9126 standard defines the "adaptability" as a factor affecting the software quality. In detail, it is a component of the "portability" quality characteristic. This factor reflects the impact the software may have on the effort of the user to adapt the software to different environments. In this sense, although portability must be an inherent ability of multi-device RIAs, especially desktop RIAs and native mobile RIAs, the access to platform-specific APIs is commonly necessary in order to bring native experiences to users. In fact, some cross-platform development frameworks provide mechanisms to access native features. Furthermore, mobile devices typically have different hardware capabilities among brands; even each brand commonly provides different hardware capabilities among models. Hence, the portability of mobile RIAs can be also adversely affected by the necessity of applications exploiting the hardware capabilities of mobile devices. Moreover, the portability of the source code generated is closely related to the ability of the code generator to support diverse frameworks and programming languages (see N. 2 T.F. 3).

Due to the quantitative nature of this evaluation method, a set of software metrics to measure the underlying software attributes need to be used. The metrics we have proposed to this aim are described in Table 3. They are focused on the measurement of source code attributes of multi-device RIAs, i.e. on the assessment of internal quality of multi-device RIAs; therefore, they are based on internal metrics defined in the ISO/IEC 9126 standard.

Quality Sub-characteristic	Source Code Attribute	Metric	Definition
Resources Utilization	Complexity	Memory usage	The estimated memory size that the product will occupy to complete a particular task.
Changeability	Cohesion, Coupling	Modification complexity	The estimated work time spent to change the software.
Analyzability	Volume	Inline documentation completeness	Ratio of the number of scripts, functions or variables having documentation to the number of implemented scripts, functions or variables.
	Volume	Compliance with coding standards	Ratio of the number of scripts, functions or variables that conform the suggested coding standards to the number of implemented scripts, functions or variables.
Adaptability	Abstraction	Software environmental adaptability	The number of implemented functions using platform-specific APIs.
	Abstraction	Hardware environmental adaptability	The number of implemented functions using device-specific APIs.

Table 3. Quality metrics to be considered on the quantitative evaluation

Due to the specific capabilities of AlexandRIA such as the capability to generate source code of RIAs based on more than one cloud services API at the same time and the capability to generate installation files of multi-device RIAs in a fully-automatic way, the case study presented in the previous section of this paper is not adequate to be used as a code generation scenario using RUX-Tool and AIR® Launchpad. Thus, in order to measure the quality of the code generated by each of the evaluated tools under the same circumstances, we have proposed a basic code generation scenario comprising a Web application that covers different standard features of RIAs, namely, the interaction design patterns support, the multimedia support and the support for retrieving data from one or more simultaneous sources, both in synchronous and asynchronous way (see N.1. T.F.1 and N.1 T.F.4). In this sense, we have outlined this code generation scenario as the "hello world" program of RIAs. It is also a means for justifying the results obtained from the qualitative analysis, at least the results corresponding to the aspects covered in this code generation scenario. It is worth mentioning that, this scenario does not involve the measurement of neither the software environmental adaptability nor the hardware environmental adaptability because these metrics are focused on evaluating standalone RIAs. Notwithstanding its simplicity, this code generation scenario still involves hand coding functionality; in fact, in the case of code generation using AIR® Launchpad, the scenario has to be almost entirely implemented by hand due to the absence of a sample application conforming to most of the requirements of the application to be generated. Going beyond this issue, AIR® Launchpad is able to generate desktop and native mobile applications only. Taking this into account, we have decided to exclude AIR® Launchpad from this evaluation with the aim of preserve the accurateness of the results.

In detail, the application to be generated must: 1) allow users to search for videos on YouTube™, 2) retrieve the corresponding video feeds from YouTube™ (YouTube™ Data API), 2) display the search results by view count using a list, 3) implement the drag and drop interaction design pattern to

allow users to drag videos from the search results list and drop they into a video playlist and 4) play the video playlist. In addition, the video results must be displayed by pages of ten results at a time (inline paging interaction design pattern).

6.2.2 Results

In the context of implementing the code generation scenario using RUX-Tool, the client for the YouTube™ Data API was implemented by using a WebRatio script unit, i.e., a piece of source code hand-written with the Groovy programming language, because of the specificity of this functionality; additionally, a WebRatio adapter unit was required in order to convert the XML-based document retrieved by the cloud services API in a new one that can be used to properly display the search results. Moreover, the drag and drop interaction design pattern was achieved by means of the *Vertical Draggable Box* and *Collapsible Droppable Tree* components, which together provide support for the drag and drop operation. These components are based on the jQuery UI *Draggable* and *Droppable* widgets, respectively, which require the explicit definition of JavaScript listeners for the events typically involved on a drag and drop operation. On the other hand, by using AlexandRIA, the required YouTube™ Data API client was achieved by simply selecting the "YouTube™ cloud services APIs" option as part of the preferences indicated throughout the wizard; similarly, in order to achieve the drag and drop interaction design pattern, the pre-built GUI/business logic component implementing the drag and drop interaction design pattern was selected as part of the preferences indicated throughout the wizard. This component is based on the Adobe® Flex Spark *DataGrid* component in the case of Web-based RIAs code generation, which includes built-in support for all the required events involved on a drag and drop operation by only setting the *dragEnabled* and *dropEnabled* properties to true.

In order to measure the memory usage of the application generated as a result of the code generation scenario described above, two particular tasks were identified. On the one hand, the "YouTube™ data retrieving" task comprises the monitoring of request sending, response handling as well as data parsing subtasks at client-side in the context of retrieving the first page of video feeds for a given query ("Shakira"). These subtasks are equivalent to the general-purpose business logic operations considered by AlexandRIA in business logic distribution. On the other hand, the "YouTube™ video playing" task comprises the monitoring of the request sending and response handling subtasks in the context of loading the SWF file of the first YouTube™ video in the page ("Waka Waka"). RUX-Tool generates AJAX-based GUIs accessing JSP-based back-ends generated by WebRatio; therefore, a servlet container is required to deploy a RUX-Tool-generated GUI. AlexandRIA generates either Adobe® Flex-based applications or PhoneGap®-based applications; therefore, AlexandRIA generated RIAs may require either the Adobe® Flash® Player plugin or the Adobe® AIR runtime, additionally, an HTTP server may be necessary if server-side business logic (PHP or JSP) is required by the user throughout the wizard. Because the RUX-Tool-generated applications can be profiled by employing any monitoring tool for JavaScript code, the built-in Google Chrome's developer tools were used to this aim. Due to the specific requirements of the AlexandRIA-generated RIAs, Adobe® Scout, a profiling tool for Adobe® Flash® Player and AIR, was used to profile the RAM memory usage of the application in performing the afore described tasks. The profiling was performed using the Google Chrome™ Web browser on the Windows 7 operating system, specifically on a 4 GB RAM, 2.8 GHz quad-core CPU laptop computer. Table 4 summarizes the profiling results.

Task / Code Generation Tool	RUX-Tool	AlexandRIA
YouTube™ data retrieving (client-side)	7 MB	43.920 MB
YouTube™ video playing	7.3 MB	58.744 MB

Table 4. Memory usage measurement results

As can be inferred from the above summarized results, in contrast to the native HTML Web browser support, the non-native support of Adobe® Flex-based GUIs (MXML) into Web browsers may have a significant cost in RAM memory consumption and other hardware resources because of the necessity of using the Adobe® Flash® Player plugin.

With the aim of measuring the modification complexity of the application generated as a result of the above outlined code generation scenario, only the look and feel modification complexity was considered in this evaluation. In particular, we have considered the customization scenario described in the case study. In this sense, custom event handlers for both the *mouse over* and *mouse out* events must be bounded to the items (defined as canvas by default) of both the *Vertical Draggable Box* and *Collapsible Droppable Tree* components in the context of customizing the default look and feel of the application generated by using RUX-Tool. RUX-Tool provides a graphical notation aimed at defining event handlers as sequences of three different kinds of actions, namely links, GUI transformations and methods, as well as conditions. In this sense, a GUI transformation action is required in order to change the background color of the canvas over which the mouse is hovered; it is also necessary to define an equivalent GUI transformation action aimed at resetting the background color when the mouse leaves a canvas. For each GUI transformation action, the target property as well as the value to be settled must be indicated. In the context of customizing the look and feel of the application generated by using AlexandRIA, the skin classes attached to the Adobe® Flex *DataGrid* components representing the search results list and the video playlist must be edited as is outlined in the case study. In detail, the "DataGridSkin.mxml" skin file generated by AlexandRIA includes a "hoverIndicator" part defining the appearance of the row of the *DataGrid* component over which the mouse is hovered; an Adobe® Flex *SolidColor* element called "hoverIndicatorFill" sets the background color. It is important to notice that Adobe® Flex automatically associates the skin parts to the involved mouse events so that it is not necessary to define event listeners and handlers. Table 5 summarizes the results obtained.

Code Generation Tool	Number of Stages	Time Spent
RUX-Tool	2 (event listeners definition, event handlers binding)	< 2 min.
AlexandRIA	2 (project importing, skin file edition)	< 2 min.

Table 5. Modification complexity assessment results

As a result of the code generation process using RUX-Tool, one INSTANCE and TEMPLATE file (resources) per each layout region defined as part of the GUI model were generated; a JSP file implementing a runtime resources resolution engine, three JavaScript files defining application's logic, three CSS files defining overall application's style, nine JavaScript files including jQuery UI library

files, jQuery plugins and one SWF Object library file, eighteen image files representing application's assets and one HTML file representing the application entry point (index) were generated. In addition, two CSS files representing a jQuery GUI theme and thirteen image files representing GUI assets were generated. As a result of the code generation process using AlexandRIA a total of four MXML files representing the Adobe® Flex application, the AlexandRIA's "Yt" component and the GUI/business logic components implementing the inline paging and drag and drop interaction design patterns, eight skin classes, fourteen icon files, three Adobe® Flex project files, one SWF file resulted from compilation and one HTML/JavaScript file representing the Web page wrapper were generated. Although the source code automatically generated by RUX-Tool from Web and GUI models is primarily edited through the models itself and not by hand, it is a best practice to generate so much developer-friendly source code as possible. Moreover, this is not the case of AlexandRIA because the source code must be able to be edited by the developer either by hand or by using Adobe® Flash Builder. According to this, the inline documentation completeness of both applications is assessed taking the business logic files as sample. In detail a total of two JavaScript files ("all.js", "ruxajax.js") containing eleven and one functions, respectively, was analyzed in the case of RUX-Tool. On the other hand, a total of two MXML files ("ytbComponent.mxml", "inlinePagingComponent.mxml") containing twelve and eight ActionScript functions, respectively, were analyzed in the case of AlexandRIA. The results obtained suggest that neither RUX-Tool nor AlexandRIA generates inline documented source code. It is important to notice that the inner JavaScript functions were not taken into account in the case of RUX-Tool. In both cases, the business logic files containing no functions were left out.

Finally, there are a lot of best practices and coding conventions for the programming languages (HTML, JavaScript, JSP, MXML, ActionScript, PHP) and frameworks (jQuery UI, Adobe® Flex, PhoneGap®) underlying to the code generators analyzed in this evaluation, which may be addressed by these tools as a primary step towards the optimization of the source code generated. However, only the code conventions for functions naming common to all the underlying programming languages were considered in this evaluation with the aim of measuring the compliance with coding standards of the applications generated as a result of the code generation scenario. It basically treats the evaluation of the compliance with the "lowerCamelCase" de facto standard of the functions defined in the business logic files also considered on evaluating the inline documentation completeness. Table 6 summarizes the results obtained.

Code Generation Tool	Sample Files	Partial Results	Final Results
RUX-Tool	"all.js" (11 functions)	0/11=0	0/12=0
	"rux-ajax.js" (1 function)	0/1=0	
AlexandRIA	"ytComponent.mxml" (12 functions)	12/12=1	20/20=1
	"inlinePagingComponent.mxml" (8 functions)	8/8=1	

Table 6. Results from compliance with coding standards assessment

It is important to notice that the handlers automatically generated by RUX-Tool starting from the links defined in the WebML hypertext model (WebRatio project) as well as the handlers directly defined in the RUX-Tool project are generated as JavaScript functions contained into the "all.js" file

so that it seems that the degree to which RUX-Tool-generated applications comply with the "lowerCamelCase" de facto standard depends to some extent on the names assigned by developers to the handlers. Nevertheless, this is not the truth because RUX-Tool automatically assigns an ID to each handler defined in the RUX-Tool project, which is used to generate the corresponding JavaScript function; similarly, the handlers generated starting from the underlying WebRatio project use IDs automatically assigned by WebRatio when the links were created. These IDs consist of sequences of words separated by underscores; therefore, the corresponding JavaScript functions do not conform to the "lowerCamelCase" coding convention. In the case of AlexandRIA, the source code generated mostly comes from a set of pre-built GUI/business logic components which have been developed taking into account not only de facto coding standards but also best practices concerning to setting up Adobe® Flex and PhoneGap® projects [42].

7 Future Directions

Conceptually the AlexandRIA's architecture has a set of libraries for different cross-platform RIAs frameworks. Therefore, we are currently working on optimizing the integration of the PhoneGap® 1.0 framework to allow generating native applications for the Windows® Phone mobile platform. In addition, we are analyzing the possibility to implement class libraries for other mobile platforms supported by PhoneGap® 1.0 such as Palm® webOS® with the aim of extending the range of mobile platforms supported by AlexandRIA. Furthermore, we are considering generating applications with functionality different from the functionality provided by the cloud services APIs; e.g., Web browser-based RIAs and desktop RIAs for managing databases and mobile RIAs that exploit hardware capabilities such as the camera and the GPS receiver of mobile devices.

Finally, we are working on the integration of RIA patterns from all rich design levels [27] to the generated applications i.e., screen layouts and application patterns in addition to the already supported interaction design patterns. The integration of these design principles and best practices means the delivering of better user experiences. At the same time, we are working to fully integrate the Abstract Data View (ADV) design model to the abstract GUI design using AlexandRIA, especially the use of ADV-charts for modeling not only structural aspects but also GUI transformations as a result of user interaction events and temporal behaviors.

Moreover, the implementation of a mechanism for automatically generating API wrappers (GUI/business logic components) as part of the code generation process represents another area of opportunity. This automatic mechanism for discovering and adding new Cloud services to AlexandRIA ecosystem could be based on a catalog of Cloud services APIs allowing not only reducing the amount of development time and effort required to provide a predefined set of API wrappers but also promoting a higher level of personalization in the code generation process. Going beyond, a process for multi-device RIAs design and development can be abstracted from the code generation algorithm implemented by AlexandRIA as a starting point on providing full support for engineering RIAs, i.e., a comprehensive multi-device RIAs development approach. In this sense, to cover the complete multi-device RIA lifecycle represents the primary step.

8 Conclusions

RIAs engineering is an emerging field of Software Engineering that everyday defines new boundaries. Nowadays, RIAs development involves the development of Web applications, desktop applications and applications for mobile devices. At this point, the GUI pattern-based approach implemented by AlexandRIA becomes relevant because it covers all phases of multi-device RIAs code generation, from the identification of the required type of application to the deployment of the generated application on target devices. AlexandRIA implements this code generation approach as a RAD approach using reusable software components and templates as high-level constructs with the aim of automatically generating both 1) source code ready-to-import into compatible Integrated Development Environments (IDEs) such as Adobe® Flash Builder 4.5 in the case of Adobe® Flex-based applications and 2) installation files ready-to-deploy on target devices, including digitally signed mobile applications. In this sense, AlexandRIA may allow RIAs developers to save development time and effort and, at the same time, it ensures less error-prone applications.

Comparing AlexandRIA against other RIAs development tools, the results show that AlexandRIA possess most of the features that a RIAs development tool must possess in order to satisfy most of the requirements of a RIAs developer. In this sense, it is important to notice that the features evaluated comes from RIAs engineering literature, code generation literature and software quality literature; therefore, they were previously validated. Furthermore, it can be inferred from the results that the quality of the applications generated by AlexandRIA respect to both the extent to which the source code can be customized and the extent to which the source code comply with coding standards is more acceptable than the quality of the applications generated by other RIAs development tools.

Finally, we want to emphasize that, unlike other proposals which are mainly based on MDE approaches and are focused on legacy Web applications GUIs improvement, AlexandRIA is focused on RIAs code generation putting the software modeling in second place. In detail, AlexandRIA automates a RIAs code generation approach centered on designing GUIs as compositions of interaction design patterns modeled by means of pre-built Abstract Data Views (ADVs). At the same time, it is inspired by diverse high level primitives already proposed in the RIAs engineering literature in order to cover most of the standard features of RIAs, namely rich GUIs, client-side business logic and sophisticated client-server communication mechanisms.

Acknowledgments

This work was supported by the General Council of Superior Technological Education of Mexico (DGEST). Additionally, this work was sponsored by the National Council of Science and Technology (CONACYT) and the Public Education Secretary (SEP) through PROMEP. This work was also supported by the Spanish Ministry of Industry, Tourism, and Commerce under the projects TRAZAMED (IPT-090000-2010-007), ENERFICIENCY (TSI-020400-2011-56), GECALLIA (TSI-020100-2011-244) and SEMOSA (TSI-020400-2011-51). Finally, this work was also supported by the Spanish Ministry of Science and Innovation under the project FLORA (TIN2011-27405).

References

1. L. A. Martínez-Nieves, V. M. Hernández-Carrillo, and G. Alor-Hernández, "An ADV-UWE Based Phases Process for Rich Internet Applications Development," in *Proc. 2010 IEEE Electronics, Robotics and Automotive Mechanics Conference*, 2010, pp. 45 –50.
2. M. Busch and N. Koch, "Rich Internet Applications. State-of-the-Art," Ludwig-Maximilians-Universität München, München, Germany, Rep. 0902, 2009.
3. A. C. W. Finkelstein, A. Savigni, E. Kimmerstorfer, and B. Pröll, "Ubiquitous Web Application Development - A Framework for Understanding," in *Proc. 6th Systemics, Cybernetics and Informatics*, 2002, pp. 431–438.
4. S. Melia, J. Gomez, S. Perez, and O. Diaz, "A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA," in *Proc. 8th Int. Conf. Web Engineering*, 2008, pp. 13 –23.
5. M. Urbieta, G. Rossi, J. Ginzburg, and D. Schwabe, "Designing the Interface of Rich Internet Applications," in *Proc.5th Latin American Web Congr.*, 2007, pp. 144 –153.
6. L. Machado, O. Filho, and J. Ribeiro, "UWE-R: An Extension to a Web Engineering Methodology for Rich Internet Applications," *WSEAS Trans. Inform. Sci. and Applications*, vol. 6, no. 4, pp. 601–610, Apr. 2009.
7. A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi, "Conceptual Modeling and Code Generation for Rich Internet Applications," in *Proc. 6th Int. Conf. Web Engineering*, 2006, pp. 353–360.
8. M. Linaje, J. C. Preciado, and F. Sánchez-Figueroa, "Engineering Rich Internet Application User Interfaces over Legacy Web Models," *IEEE Internet Computing*, vol. 11, no. 6, pp. 53–59, Nov. 2007.
9. F. J. Martinez-Ruiz, J. M. Arteaga, J. Vanderdonckt, J. M. Gonzalez-Calleros, and R. Mendoza, "A First Draft of a Model-driven Method for Designing Graphical User Interfaces of Rich Internet Applications," in *Proc. 4th Latin American Web Congr.*, 2006, pp. 32 –38.
10. F. Valverde and O. Pastor, "Facing the Technological Challenges of Web 2.0: A RIA Model-Driven Engineering Approach," in *Proc. 10th Int. Conf. Web Inform. Syst. Engineering*, 2009, pp. 131–144.
11. L. O. Colombo-Mendoza, G. Alor-Hernandez, and A. Rodriguez-Gonzalez, "A Novel Approach for Generating Multi-device Rich Internet Applications," in *Proc. 22nd Int. Conf. Electrical Communications and Computers*, 2012, pp. 361 –367.
12. M. Linaje, J. C. Preciado, R. Morales-Chaparro, R. Rodríguez-Echeverría, and F. Sánchez-Figueroa, "Automatic Generation of RIAs Using RUX-Tool and Webratio," in *Proc. 9th Int. Conf. Web Engineering*, 2009, pp. 501–504.
13. F. Valverde and O. Pastor, "Applying Interaction Patterns: Towards a Model-Driven Approach for Rich Internet Applications Development," in *Proc. 7th Int. Workshop Web-Oriented Software Technologies*, 2008, pp. 13–18.
14. G. Rossi, M. Urbieta, J. Ginzburg, D. Distante, and A. Garrido, "Refactoring to Rich Internet Applications. A Model-Driven Approach," in *Proc. 8th Int. Conf. on Web Engineering*, 2008, pp. 1–12.
15. N. Koch, M. Pigerl, G. Zhang, and T. Morozova, "Patterns for the Model-Based Development of RIAs," in *Proc.9th Int. Conf. Web Engineering*, 2009, pp. 283–291.
16. J. M. Wright, "A Modelling Language for Interactive Web Applications," in *Proc.2009 IEEE/ACM Int. Conf. Automated Software Engineering*, 2009, pp. 689–692.
17. V. Gharavi, A. Mesbah, and A. V. Deursen, "Modelling and Generating AJAX Applications: A Model-Driven Approach," Delft University of Technology, The Netherlands, Rep. TUD-SERG-2008-024, 2008.

18. L. Sorokin, F. Montero, and C. Märtin, “Flex RIA Development and Usability Evaluation,” in *Proc. 2007 Int. Conf. Web Inform. Syst. Engineering*, 2007, pp. 447–452.
19. F. Paterno’, C. Santoro, and L. D. Spano, “MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments,” *ACM Trans. Computer-Human Interaction*, vol. 16, no. 4, pp. 19:1–19:30, Nov. 2009.
20. P. Dolog and J. Stage, “Designing Interaction Spaces for Rich Internet Applications with UML,” in *Web Engineering*, LNCS 4607, Springer Berlin / Heidelberg, 2007, pp. 358–363.
21. B. Stearn, “XULRunner: A New Approach for Developing Rich Internet Applications 2007,” *IEEE Internet Computing*, vol. 11, no. 32, pp. 67 –73, Jun. 2007.
22. J. C. Preciado, M. Linaje, F. Sanchez, and S. Comai, “Necessity of Methodologies to Model Rich Internet Applications,” in *Proc. 7th IEEE Int. Symp. Web Site Evolution*, 2005, pp. 7–13.
23. R. Rodríguez-Echeverría, J. Conejero, M. Linaje, J. Preciado, and F. Sánchez-Figueroa, “Re-engineering Legacy Web Applications into Rich Internet Applications,” in *Web Engineering*, LNCS 6189, Springer Berlin / Heidelberg, 2010, pp. 189–203.
24. G. Toffetti, S. Comai, J. C. Preciado, and M. Linaje, “State-of-the Art and Trends in the Systematic Development of Rich Internet Applications,” *J. Web Engineering*, vol. 10, no. 1, pp. 70–86, Mar. 2011.
25. “A Framework for Software Product Line Practice, Version 5.0,” *Software Engineering Institute*. [Online]. Available: http://www.sei.cmu.edu/productlines/frame_report/tool_support.htm. [Accessed: 17-Jul-2012].
26. B. Scott and T. Neil, *Designing Web Interfaces: Principles and Patterns for Rich Interactions*, 1st ed. O’Reilly Media, 2009.
27. T. Neil, “Designing Rich Applications,” *Slideshare® website*, 2009. [Online]. Available: http://www.slideshare.net/theresaneil/designing-rich-applications. [Accessed: 28-May-2012].
28. CaSEMaker Inc. (n.d.). What Is Rapid Application Development? [Online]. Available: http://www.casemaker.com/download/products/totem/rad_wp.pdf.
29. G. Coleman and R. Verbruggen, “A Quality Software Process for Rapid Application Development,” *Software Quality Control*, vol. 7, no. 2, pp. 107–122, Jul. 1998.
30. P. Fraternali, “Tools and Approaches for Developing Data-intensive Web applications: A Survey,” *ACM Computer Survey*, vol. 31, no. 3, pp. 227–263, Sep. 1999.
31. B. Kitchenham, "DESMET: A Method for Evaluating Software Engineering Methods and Tools," Department of Computer Science, University of Keele, Staffordshire, U. K, Rep. TR96-09, 1996.
32. J. H. D, Code Generation in Action, Revised. Manning Publications, 2003.
33. P. Vogel, Practical Code Generation in .NET: Covering Visual Studio 2005, 2008, and 2010, 1st ed. Addison-Wesley Professional, 2010.
34. M. Gunderloy and Sybex, Coder to Developer: Tools and Strategies for Delivering Your Software, 1st ed. Sybex, 2004.
35. B. Kitchenham and S. L. Pfleeger, “Software Quality: The Elusive Target [Special Issues Section],” *IEEE Software*, vol. 13, no. 1, pp. 12 –21, Jan. 1996.
36. *Ergonomic requirements for office work with visual display terminals (VDTs) – Part 9: Guidance on usability*, ISO/IEC 9241-11:1998, 1998.
37. N. Aquino, J. Vanderdonckt, N. Condori-Fernández, Ó. Dieste, and Ó. Pastor, “Usability Evaluation of Multi-device/Platform User Interfaces Generated by Model-driven Engineering,” in *Proc. of 2010 ACM-IEEE Int. Symp. Empirical Software Engineering and Measurement*, 2010, pp. 30:1–30:10.
38. J. R. Lewis, “IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use,” *Int. J. Human-Computer Interaction*, vol. 7, no. 1, pp. 57–78, Jan. 1995.

39. W. O. Galitz, *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*, 3rd ed. Wiley, 2007.
40. B. Boss, "An essay on W3C's design principles," *W3C website*, 06-Mar-2003. [Online]. Available: http://www.w3.org/People/Bos/DesignGuide/designguide.html. [Accessed: 11-Apr-2012].
41. R. Likert, "A Technique for the Measurement of Attitudes," *Archives of Psychology*, vol. 22, no. 140, pp. 55, 1932.
42. S. Moore, "Flex best practices – Part 1: Setting up your Flex project," *Flex Developer Center*, 07-Jul-2008. [Online]. Available: http://www.adobe.com/devnet/flex/articles/best_practices_pt2.html. [Accessed: 11-Apr-2012].