Automatic verification and validation wizard in web-centred end-user software engineering

David Lizcano [a, *], Javier Soriano [b], Genoveva López [b], Javier J. Gutiérrez [c]

[a] Universidad a Distancia de Madrid, UDIMA, Madrid, Spain
[b] Universidad Politécnica De Madrid, Madrid, Spain
[c] Universidad de Sevilla, Spain

Keywords:

End-user software engineering

Web engineering

Reliability

End-user programming

visual programming

human-computer interaction

A B S T R A C T

This paper addresses one of the major web end-user software engineering (WEUSE) challenges, namely, how to verify and validate software products built using a life cycle enacted by end-user programmers. Few end-user development support tools implement an engineering life cycle adapted to the needs of end users. End users do not have the programming knowledge, training or experience to perform devel- opment tasks requiring creativity. Elsewhere we published a life cycle adapted to this challenge. With the support of a wizard, end-user programmers follow this life cycle and develop rich internet applica- tions (RIA) to meet specific end-user requirements. However, end-user programmers regard verification and validation activities as being secondary or unnecessary for opportunistic programming tasks. Hence, although the solutions that they develop may satisfy specific requirements, it is impossible to guarantee the quality or the reusability of this software either for this user or for other developments by future end-user programmers. The challenge, then, is to find means of adopting a verification and validation workflow and adding verification and validation activities to the existing WEUSE life cycle. This should not involve users having to make substantial changes to the type of work that they do or to their priori- ties. In this paper, we set out a verification and validation life cycle supported by a wizard that walks the user through test case-based component, integration and acceptance testing. This wizard is well-aligned with WEUSE's characteristic informality, ambiguity and opportunisticity. Users applying this verification and validation process manage to find bugs and errors that they would otherwise be unable to identify. They also receive instructions for error correction. This assures that their composite applications are of better quality and can be reliably reused. We also report a user study in which users develop web soft- ware with and without a wizard to drive verification and validation. The aim of this user study is to confirm the applicability and effectiveness of our wizard in the verification and validation of a RIA.

1. Introduction

The number of end-user programmers (people who program to achieve the result of a program primarily for personal rather than public use) (Ko et al., 2011) grows year by year and is much greater than the number of professional developers (Scaffidi et al., 2005). According to the US Bureau of Labor and Statistics, compared with three million professional programmers in the United States, there were more than 60 million end-user programmers (EUPs) using spreadsheets and databases at work in early 2014, many writing formulas and dashboards to support their job (Cao et al., 2014). This has spawned a lot of interest in research into all aspects of software development by end-user programmers and user-centred software engineering.

At the start of our research, we defined a model designed to enable EUPs to handle components tailored to their experience and problem domain knowledge (Lizcano et al., 2011a). The defined model is based on components of different levels of abstraction. Components and connectors, that is, elements that can be used to set up a data flow among components of the same level, become less detailed as we move up the hierarchy. We analysed the features required by a visual EUP-centred development environment empowering EUPs to effectively handle the defined components and connectors in (Lizcano et al., 2011b).

The next step was to define web end-user software engineering (WEUSE) and to specify the analysis, design and implementation stages of a RIA life cycle as enacted by an EUP (Lizcano et al., 2013). We proposed suitable mechanisms for supporting the activ-

* Corresponding author.
E-mail addresses: david.lizcano@udima.es (D. Lizcano), jsoriano@fi.upm.es (J. Soriano), glopez@fi.upm.es (G. López), javierj@us.es (J.J. Gutiérrez).

ities of the RIA life-cycle phases. The aim was to empower EUPs to successfully compose the component graph and help them to rapidly build a RIA adapted to a specific problem that they need to solve. The reported WEUSE achieves this goal using a development wizard (DW) composed of well-defined and structured tasks. This wizard provides users with guidance to develop RIAs, which is outside the realms of their knowledge. The DW, implemented within an end user-centred visual development environment, called FAST, walks EUPs through these life-cycle stages. FAST is an EUP-centred integrated development environment developed as part of a 7th European framework programme project.

A RIA built by an EUP is an application composed of a set of components with inputs and outputs. The bottom-level components are able to invoke services and access web resources. They are designed and then published in repositories shared by the proprietors of the services/resources that are to be made visible. In order to promote the end-user development (EUD) philosophy, they are made accessible to EUPs or third parties. Users use EUD tools (like Yahoo! Pipes and Dapper, Kapow, JackBe, FAST, and so on) to access the above repositories and create personal compositions by connecting up some of these bottom-level components with others and/or building composite applications.

Even though EUPs run some tests to check the goodness of the RIA that they have developed, they do not really perform a systematic verification and validation process in order to reliably check that the built RIA is error free.

In this paper, we describe a WEUSE verification and validation stage as a combination of test case generation and test case execution for testing a RIA at component, integration and acceptance level. This verification and validation process assures that the developed RIA conforms to the end-user specification and meets the needs for which it was built. During this stage, the users are assisted by a *What You See Is What You Test* testing wizard (TW) that walks untrained users through the implementation of a user-centred component, integration and acceptance testing plan.

Note that the aim of the testing process is to find and locate as many defects as possible, whereas the aim of the debugging process is to fix and remove the detected defects. In traditional software engineering, the testing team is responsible for testing, whereas the development team carries out debugging. In the case of WEUSE, however, the EUP performs both tasks. To do this, the EUP first executes the TW and then fixes the bugs that the wizard has detected and not automatically corrected. In the last analysis, the aim of the WEUSE verification and validation process described in this paper is to check that the software system meets the specifications and serves its intended purpose by applying the testing and debugging processes.

Effective verification and validation should prevent errors being compounded by the reuse of buggy software. Therefore, EUPs need to have access to an automatic system to validate their developments. Verification and validation should not, however, take too much time or effort because EUPs view it as being an unnecessary and unimportant process.

The WEUSE verification and validation stage includes three levels of testing: component, integration and acceptance testing. The unit components published in the catalogues have been built by specialised software providers and should, in principle, work correctly. However, EUPs may publish parameterisations and compositions based on these unit components, and these new ad-hoc components should be tested. Component and integration tests are run by the TW automatically to check that end-user software is error free. Inputs are the endpoints of the range of each of the expected data types. The TW runs acceptance tests based on data requested from users. It uses black-box testing to check how the data flow through the RIA. The TW displays the intermediate and final data in tabular format, indicating whether or not the processed internal and external data reach their destination. End users, who are problem domain experts, should then have no trouble with data checks. Based on the execution of the test cases that it generates, the TW will analyse components that generate any erroneous data items identified by users and ask the DW to suggest an alternative design (based on data flows among components or other equivalent components published in the catalogue). The RIA component, integration and acceptance tests are further divided into two stages: wizard-driven test case generation and test case execution, as described in Section 3.

The remainder of the article is divided as follows. Section 2 introduces research on the provision of support for the verification and validation of EUDs. There is no support for such tasks in the RIA field, but we have used earlier proposals in other EUD fields, such as spreadsheets, in our research. Section 3 reports the proposed verification and validation process for WEUSE, defining the types of test cases to be executed at each level and implementation details of how the TW supports EUPs. Section 4 documents our working hypotheses and the user study conducted to examine the evaluation criteria. This section includes detailed information on the user study to assure that it can be replicated in other fields. It also includes a statistical study of the results. Section 5 discusses threats to validity. Finally, Section 6 reports our conclusions, setting out the major contributions of our research to the state of the art and future lines of research that we intend to undertake.

2. Related work: end-user PROGRAMMER VERIFICATION and validation

Spreadsheets were the first real examples of developments by EUPs in the field of EUD (Rothermel et al., 2001; Chambers and Erwig, 2009). Over recent years errors made by EUPs have led to huge financial losses and defective quality at small and large companies alike (losses that were documented up to 20 years ago (Panko, 1995) and are still occurring). In response, research focused on *What You See Is What You Test* has been conducted in order to give users access to systematic testing procedures to validate their spreadsheets (Burnett et al., 2002).

Surprise-Explain-Reward is one What You See Is What You Test strategy (Wilson et al., 2003). This strategy employs surprise to draw the user's attention to software engineering tasks. Users are then encouraged, through explanations and rewards, to follow through with appropriate actions. This strategy has its roots in three areas of research—research about curiosity (psychology), Blackwell's model of attention investment (psychology/HCI), and minimalist learning (educational theory, HCI)—and has been used in work on model-driven methods (Burnett, 2009). We have used this strategy to provide users with visual guidance on the testing procedure in order to check that the right RIA has been built right. The procedure is explained in Section 3.

As mentioned in the introduction, we regard verification as a process of checking that the RIA conforms to the specifications given by the analyst and designer, roles played in this case by the end user. This process checks that the program implements all the sentences specified by the user to define the RIA (each sentence is a use case). On the other hand, validation, also performed by the user, checks that the result of executing the RIA satisfies user needs.

In the following, we summarise the end-user software engineering verification and validation activities identified in the state of the art, the problems that EUPs face within each activity and what solutions are now available for these problems.

- Generation of test cases for RIA use case verification and validation:

End-user developers do not usually specify or describe requirements. They regard this as unnecessary effort because they are going to do all the developing themselves. This causes problems throughout the remainder of the development process (Costabile et al., 2006; Fischer and Giaccardi, 2006; Mørch and Mehandjiev, 2000; Segal, 2007). RIAs should be verified and validated by generating and executing test cases to check that the RIA conforms to the requirements specified by the user. EUPs require guidance for specifying such requirements.

It is common practice in software engineering to define the functionality of an application by means of use cases. For this purpose, some end-user software engineering experts use special-purpose questions or a similar mechanism to elicit use cases (Ko and Myers, 2004; Obrenovic and Gasevic, 2009; Burnett et al., 2002). RIA use cases are defined in the requirements analysis phase (in our approach by the DW), but at present there are no approaches for creating test cases from end-user use cases.

- Provision of traceability for components, connections and compositions for each use case:
  Users should have a record of the elements that they decide to use in each use case, how they are connected and how they are composed during the analysis, design and implementation phases. However, EUPs do not have a record of how these elements trace back to the use cases or potential errors.
  Some authors propose the use of coloured Petri nets to plot the graph elements used in each use case and illustrate how they are connected and composed (Cai et al., 2011). Graph navigability assures that requirements are traceable for design, implementation and testing. Therefore, this paper uses coloured Petri nets to provide support for traceability. These coloured Petri nets are used not for interaction with the user but internally by a wizard (TW). This wizard provides step-by-step guidance on testing, storing the manner in which the data flows are produced and giving users visual instructions that track the input data through their conversion into application output data.
- Component testing:
  Web components not tested by providers may later cause the RIA to fail. On the other hand, access to the source code of the components is not open, and black-box testing is the only option. Users have no knowledge of programming, flow-control structures or data types. Neither do they know how to run black-box tests on components with whose internal workings they are unfamiliar.
  There are alternatives for component testing based on the use of systematic testing tools in the EUD spreadsheets field (Davis, 1996; Chambers and Erwig, 2009). They propose black-box tests for spreadsheets. This idea can be applied in the WEUSE field, as suggested in this paper.
- Integration testing:
  For these tests, EUPs would need to be acquainted with the technical specifications of the components, data types and syntax used to define the component inputs and outputs for connection with other components. End-user software engineering gives EUPs recommendations on which components to connect to others and how, but this basic knowledge is not enough to analyse whether the source of the error is the component, the connection or the data processing.
  Existing approaches propose isolating white-box components (which they regard as mere functions) and analysing the data flow between components and the target flow of data types (Yoon and Garcia, 1998). This idea has been used to support TW integration testing.
- Acceptance testing:
  EUPs do not know how to properly validate an RIA built for a specified purpose (Mackay, 1990). As soon as they find that their application works properly in any particular case, EUPs get their program up and running without bothering to run any further tests. One of the major causes of poor quality end-user developments is end-user overconfidence in the solution, which ends up generating other errors (Chambers and Erwig, 2009).
  Rothermel et al. (2001), Burnett et al. (2002), Igarashi et al. (1998), Clermont (2003), Fischer and Giaccardi (2006) propose What You See Is What You Test black-box testing or partial prototype validations using boundary value analysis. We use Surprise-Explain-Reward. This strategy employs colours to illustrate the final and intermediate data, thereby offering guidance on how to correct any errors. It also provides internal component traceability, which is helpful for identifying the source of the error.
  Chambers and Erwig (2009) suggest automatically comparing the solution specification against the result generated by the complete end-user composition using unit inference and checking systems.
- Error correction:
  Users without programming skills cannot perform this activity properly (Sutcliffe and Mehandjiev, 2004). EUPs are not specifically trained to perform tests and detect errors, and they are even less qualified to identify the possible causes of and correct a detected error.
  Several researchers propose using an iterative hypothesis/confirmation/refutation process to correct errors in each component or element used (Keijiro et al., 1991). This is the approach used by our prototype. This approach generates a hypothesis on which component or connector might be causing each detected error. A wizard then puts forward an alternative using a different yet functionally equivalent element. The user then has to accept or reject the respective change. Other authors suggest using a wizard to manage control and data dependencies (Janner et al., 2009). Another proposed option is for a community of EUPs sharing debugging problems, solutions or parts of solutions to afford social and collaborative support (Fischer et al., 2006).

If the user wants to add the TW-tested RIA to the catalogue, the RIA must pass crowdsourced testing by different testers (a technique that has proven its potential as reported by Brambilla et al., 2012). To do this, it is provisionally added to the catalogue for testing and is made permanent when it passes the crowdsourced testing.

There is no specific proposal of a tool or process to verify, validate and debug end-user software engineering web compositions, containing a formalised schedule of activities, tasks, artefacts, etc. In this paper, we propose a specific solution for this problem. To solve the problems that EUPs will face at each testing level, we use a wizard providing guidance for EUPs. This wizard exploits most of the ideas specified above.

3. User-centred verification and validation in WeUSE

In this section we present our approach to the verification and validation of composite web applications built by EUPs. The proposed WEUSE published elsewhere (Lizcano et al., 2013) sets out an iterative and incremental life cycle. User-performed tasks categorised by disciplines (requirements, analysis, design and composition) are executed at each iteration, driven by the DW. The DW provides guidance on how to perform these tasks and decides when to start a new iteration that increases the set of requirements to be addressed.

During the *requirements discipline*, the DW prompts users to draft a natural language description of the problem that they want to solve. To do this, they use short sentences expressing simple

system functionalities (e.g., display tourist information about a particular destination). The DW parses this natural language to identify keywords. If accepted by users, these keywords are used as input for searching for catalogued components that the EUP community has tagged in the same way. Component catalogues are repositories with associated tag sets. These repositories make up a dynamic folksonomy collaboratively tagged by providers and users. They can be searched using the retrieved keywords in order to find the required components.

In the *analysis discipline*, users are given a list of possible components for use based on the requirements and DW-driven search. At this point, the DW offers information on what each component does, and what its inputs and outputs are.

In the *design and composition discipline*, the DW provides visual aids to help users link the selected components using connectors that set up valid data flows. End users have problems with this (Kuttal et al., 2013). The DW highlights inputs and outputs that are compatible depending on the respective data type. Also, the DW stores successful connections used in satisfactory designs by other users. These are used to recommend component interconnections. The visual aids rely on data type checking, valid component uses recorded in previous EUP designs, etc. If users so wish, they can add new requirements to increase web application functionality by reiterating the life cycle. At the end of the process, EUPs can manually test their RIA using application data to check its goodness. Users tend not to spend very much time on this manual testing activity.

EUPs have two testing options after DW execution. They can test the developed RIA manually or they can use the TW. The TW is an optional, semiautomatic feature for verifying and validating that the RIA operates correctly.

Like the DW, the TW is based on an interactive and incremental approach. It is governed by the activity diagram illustrated in Fig. 1 and explained step by step throughout this paper. First, the TW prompts users to briefly state their problem. The TW applies natural language processing to the above problem statement in order to retrieve sentences that it converts into test cases. The sources of information used by the TW are: a domain ontology, the database generated by the DW and the component folksonomy associated with the problem domain (this process is detailed in (Lizcano et al., 2013). After test case generation, the TW drives component, integration and validation tests, prompting user interaction using a visual interface implementing Surprise-Explain-Reward techniques. At the end of the testing phase, users are taken through the application publication process.

The TW switches between two activities: test case generation and test execution based on the generated test cases. Test cases are generated in the background by the TW without direct user intervention. In this activity, the TW receives information from the DW (use cases, keywords, components, component aggregations and connections). During test execution, the TW automatically executes the component and integration test cases generated in the respective iteration. It then interacts with the user to run acceptance testing, the only point on which the TW is not transparent. When the DW starts another iteration in order to add functionality to the RIA, another TW iteration reruns the (automatic) component, (automatic) integration and (visual user) acceptance tests.

Fig. 2 shows an overview of the iteration of the WEUSE life cycle. In this process, the TW is fed uses cases, keywords, lists of used components, component aggregations and component connections in order to prepare the test cases and run component, integration and acceptance tests for the respective iteration as follows.

The detected errors are recorded in a log file. This log can be queried later by conversion to either a plain text file or tabular list in csv format or by viewing a sequence of screens illustrating each detected error and its status.

3.1. Test case generation

EUPs specify natural language requirements as sentences that define specific concepts for development. Each sentence is a system functionality and matches a use case. In order to design and develop the target RIA, the DW parses these requirements specified in a natural language to define the use cases (each sentence ending in a full stop will represent a use case) and keywords. Parsing relies on problem domain ontologies used to locate the keywords, plus keyword synonym dictionaries and social tags used to annotate the catalogued components and compare the target keywords with tags. These ontologies must be developed by problem domain experts and will be dependent on the domain of the target RIA. Keywords are retrieved by an engine built as part of the FAST project and documented in (Lizcano et al., 2013).

The DW generates one RIA use case for each sentence of the textual description and stores the component identifiers of each of the identified use cases stated by the user. Table 1 describes how this activity works.

Suppose that a user wants to build a RIA to search for flight and accommodation options for an event scheduled in his personal organiser. The DW will prompt the user to give a natural language description of requirements, describing each requirement in a separate sentence. An example of part of the natural language description of the target RIA might be:

> […]
> *P0. Mashup as personal organiser for events*
> *S1. Search a flight and hotel for an event entered in the organiser.*
> *S2. Display tourist information on the target destination.*
> *[…]*

Running through the sentences of the requirements description, the parser will come across sentence S1: "*Search a flight and hotel … in the organiser*". The DW will generate a use case for this requirement. Analysing the keywords, the algorithm implemented by the DW will recognise "search", "flight", "hotel, "event" and "organiser". The system will compare these keywords with the syntactic analysis ontology (capable of analysing direct objects for entered verbs and identifying these verbs in order to search the catalogued components for target functionalities), domain folksonomies and the DW database. The current prototype has folksonomies for very specific domains (travel/leisure services, banking, office automation, etc.), especially designed by expert domain ontologists with our guidance. Nevertheless, other more general and more varied ontologies and folksonomies are to be adopted in the future. Additionally, the DW stores data on each design, implementation and use of the EUP, data on the components that have been used more often (and in combination with which other components) and the use of each component depending on the keyword searches. At the end of this process, it will search the catalogue for components related to "event", "organiser", "flight search", "hotel search" to be built into the RIA. The DW will use these inputs to locate several components in the catalogue, including personal organisers, flight search engines, hotel search engines, etc. The user will analyse these components and select (based on their functional definitions, inputs and outputs) which to use. For example, he may choose the Rumbo flight and the Expedia hotel search engines to compose a data flow. This entire process is documented in (Lizcano et al., 2013).

Supposing that the organiser has been used in previous development iterations and is therefore already part of the partial RIA developed by the user before analysing sentence S1, the user will

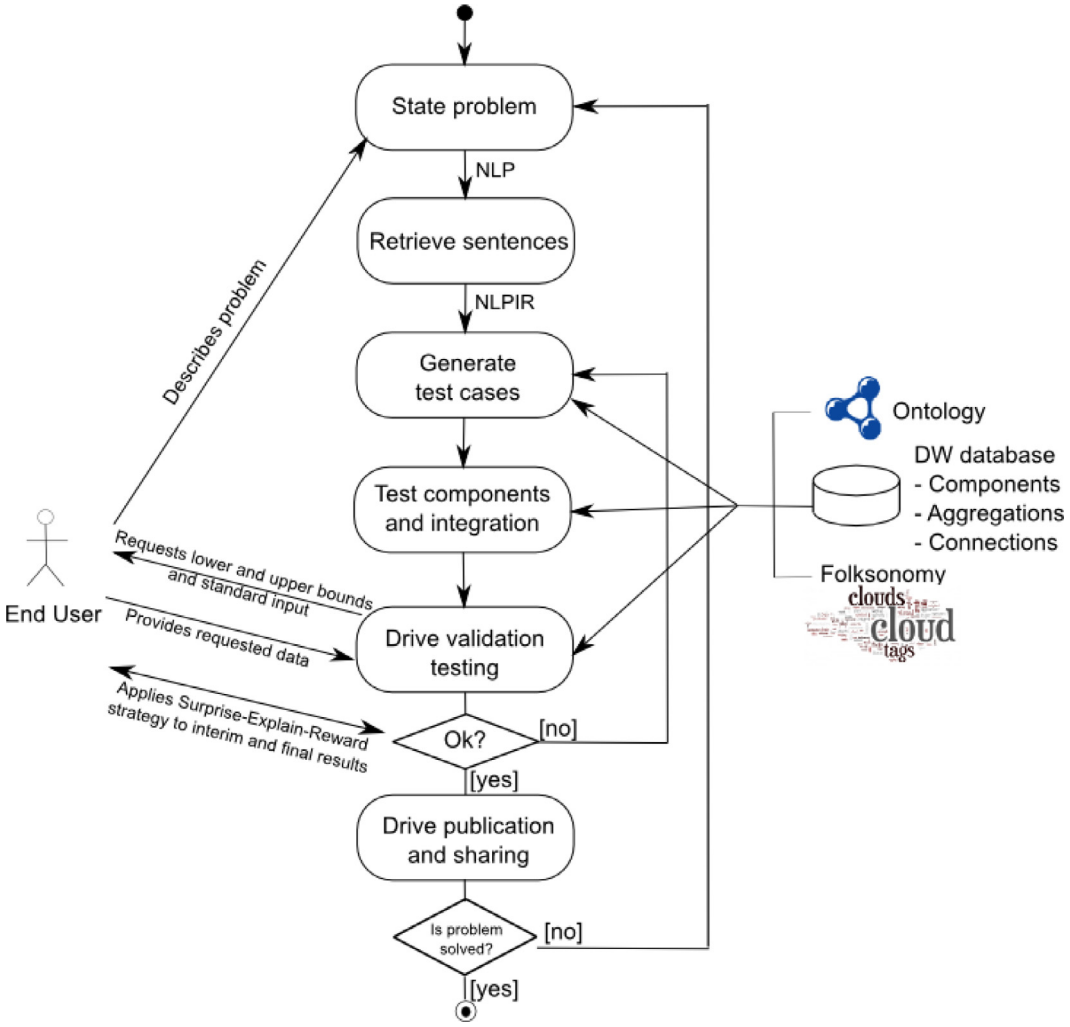

Fig. 1. Activity diagram of the developed TW.

Table 1
Operation of test case generation in TW.

User inputs	TW inputs from DW	TW processing	Outputs
Natural language problem description	Sentences analysed by the DW as use cases Keywords retrieved from the description entered and validated by users Lists of used components List of alternative components Links between components Aggregations of components	TW analyses the components, links and aggregations built by the user using the DW.3 TW generates several test cases for each use case.	Three outputs for each use case: • List of components (catalogued component id and URL) • Graph of component aggregations • Coloured Petri net with data flows among components These elements are used to produce the test cases.

link the above organiser to the new (Rumbo and Expedia) components following DW recommendations. The user will make these connections by simply clicking on the visual representations of the components in the EUD tool IDE. These representations include checkboxes for component inputs and outputs. The visual links between these checkboxes will be generated when specified by the user (Lizcano et al., 2014).

The partial prototype of already linked components is attached to sentence S1 of the original natural language description, keywords and other alternative components offered to the user (and will be used as an alternative if the tests reveal a component to be unsuitable). The TW will use the design existing in the partial prototype to build a test case covering the requirements that appear in this sentence.

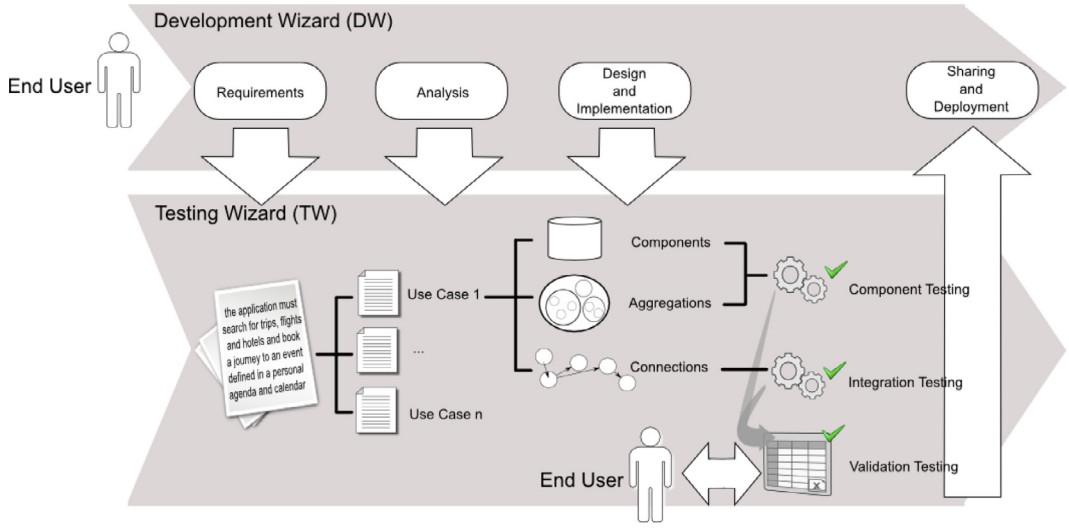

Fig. 2. Overview of an iteration of the WEUSE life cycle with testing.

For each use case, the TW receives all the above information: the sentence that was the source of the use case, the processed keywords, a list of components used, aggregations and connections illustrating how components are linked and related, and a list of alternative components (with the same types of inputs, outputs and functionality) as the components selected by the user during design, and moves on to the second activity.

The TW will run automatic component and integration tests and user-driven acceptance tests on these components. For acceptance testing, it will request from the user three values for each input of each component that is part of the architecture satisfying the use case: the minimum and maximum data items, and a standard or customary data item in the respective problem domain of each input. The maximum and minimum values that the programming language is capable of representing for the respective data types will be displayed for the user, and the user can either accept these values or specify different maximum and minimum values that make more sense in the problem domain (for example, a minimum value for flight date might be today's date instead of the minimum date that can be represented in the programming language date and time format). Rather than default values, these are the maximum and minimum values dictated by the data type used to represent the variable. The user may accept these values or assign others. They are the endpoint values for the data type in question.

An example will clarify this mechanism for selecting the three specified values: the TW will request the three data for "outbound flight date", "return flight date", "flying from", "flying to" and "number of passengers" for the Rumbo flight search engine, plus three test data for "check-in date" and "check-out date", as well as "number of rooms" and "room type" for the Expedia hotel search engine. For dates, the TW uses the first and last date that can be represented (in standard date and time format), but the user can modify these default values. For destinations, the end-user components have checklists including all the possible airports to which airlines operate with alphabetically sorted values, although geographical coordinates can also be used. In this case, the maximum and minimum coordinates that can be represented in the cardinal system are used. Finally, positive integers are used for the number of rooms: the system would recommend 0 as the minimum and 2^n-1 as the maximum, where n is the number of bits used. However, users might consider 10 to be a better maximum value for the maximum number of rooms for a single hotel booking. Another noteworthy point is that the TW does not analyse the data semantics. For example, it will never be able to guess that the return flight date must be later than the outbound flight date. The component published by the service provider (the travel agency) is responsible for the semantics. Therefore, if the constraint is not met, the component should output an on-screen error message or disable the respective data item at run time. The TW will use these three data to generate five exhaustive test cases: one using all minimum data, one using data just above the minimum (calculable by the TW), one with all the standard data, one with data just under the maximums and, finally, one with all maximum data. This process is explained in detail throughout the article.

The boundary value-based testing technique that we propose is a well-established technique. As the combinatorial characteristics of the input parameters of each component participating in the user-built RIA are not known, we ruled out a combinatorial testing techniques (t-way, pairwise testing) as a possible option for the proposed TW.

3.2. Test case execution

The TW then runs three types of test. Two of these tests are automatic and a third requires end-user intervention. These are component, integration and acceptance tests, respectively.

3.2.1. Component tests

Component tests are run fully automatically without EUPs having to intervene. They have two goals: a) check that the components used in the RIA do not produce any run-time exception or error, and b) check that the components built by aggregating other less abstract components conform to the constraints (preconditions and postconditions) of the lower-level components. Table 2 explains how component testing works.

The TW traverses the graph of use case components running automatic tests using unit testing with parameter value coverage. To do this, the component input used by parameter value coverage covers all the possible standard values for a data parameter within the range of the target input type. Parameter value coverage then checks that the resulting output is within the range of the target output data. For example, if the component input is an integer for which an integer-type output is generated, parameter value coverage tests are run automatically for the range of integers: a positive integer, a negative integer, zero, int.maxvalue and

Table 2
Operation of component testing.

User inputs	Inputs from earlier TW stages	Support elements	TW processing	Outputs
None	List of test cases	Ontology of used components describing inputs and outputs	TW generates test data banks for each basic and aggregated component, generating data across the entire input range and checking that the outputs are within the target output range	List of components that generate unexpected data
	List of components (catalogued component id and URL)	Definition of EUD tool data type ranges		List of keywords associated with the component that failed in order to search the repository for equivalent components, i.e. components with the same associated tags
	Graph of component aggregations			

Table 3
Operation of integration testing.

User inputs	Inputs from earlier TW stages	Support elements	TW processing	Outputs
None	List of test cases	Ontology of used components describing inputs and outputs	TW traverses the coloured Petri net. It generates test data banks for each component starting a data flow across the input range and checks that the generated output is within the target output range.	List of components that generate unexpected data
	List of components (catalogued component id and URL) Coloured Petri net with data flows among components	Definition of EUD tool data type ranges		List of data flows that generate unexpected data List of keywords associated with the component that failed in order to search the repository for equivalent components

int.minvalue, checking that a value in the integer range is generated in each case. If the EUD tool outputs an exception or error, the TW captures the resulting exception and transfers control back to the EUD tool.

Parameter value coverage lists were created for each data type processed by the prototype. For example, the list to be tested for a string is: a null string, an empty string or "", spaces, tabs, a new line, a valid string, an invalid string and unicode characters.

Components may execute external invocations that can have stateful effects on other systems. It is not always acceptable to automate such operations during component testing. The platform running the TW (for example, FAST or the Yahoo Pipes dashboard) knows which external accesses to services can have stateful effects on other systems during component execution, as the component uses the platform as a proxy for these requests. These invocations, which typically use HTTP verbs on REST service URIs, SOAP invocations or similar, are not externally redirected; they are tested locally by analysing the structure of the invocation for correctness based on regular expressions and ontologies used to check the invocation for validity. External service testing is beyond the scope of this automatic process. The current prototype is capable of analysing whether a text string is a valid email address and whether a HTTP request is valid.

Any user intending to run this type of test without the TW would have to instantiate the component in the sandbox of the IDE provided by the mashup platform and manually test each possible input data value one by one.

3.2.2. Integration tests

Integration tests are also run automatically without end-user involvement. Their aim is to check that components of the same level of abstraction that have been linked to each other using data flows interact correctly. Integration tests check the data flow between components for the use cases defined in the requirements rather than their aggregations. Table 3 details how integration testing works.

The TW traverses the graph of connections between the use case components that are being tested and generates the data flows for each test case. In this process, it simulates input values (using the parameter value coverage approach) to check that the outputs conform to target data types for each connection across the range of target data types. Again these automatic tests use the parameter value coverage lists that have been created for each type processed by the prototype.

Any user intending to run this type of test without the TW would have to instantiate each component pair in the sandbox of the IDE provided by the mashup platform and test (for data type consistency) each possible valid data flow one by one in order to check whether the elements have been correctly integrated.

3.2.3. Acceptance testing

The automatic component and integration tests check that data flows between components and internal data processing do not raise exceptions. Once these tests are complete, the entire RIA needs to be tested to check that the intermediate and final output data conform to the semantics expected by the user for particular input data. To do this semantic checking, the user has to participate in the tests that are driven by the TW. Table 4 details how acceptance testing works.

The TW analyses the RIA inputs and prompts the EUPs to enter a typical data set for each input. This prompt consists of a form displayed by the TW, shown in Fig. 3.

For each case, the TW asks users to enter the lowest feasible or acceptable value, the highest feasible value and a frequent or common value for the above data item depending on the purpose to which the RIA is to be put. These data will be used to run the above five test cases. These TW settings can be modified to request more data if the tests suggest a posteriori that further

Table 4
Operation of acceptance testing.

User inputs	Inputs from earlier TW stages	Support elements	TW processing	Outputs
List of test data (minimum, standard and maximum)	List of test cases	Ontology of used components describing inputs and outputs	The TW uses test cases to run five acceptance tests for each input data item (minimum, standard and maximum value of the data item in the use case). The data type test ranges are specified by the ontologies associated with the respective data type. The TW builds tables containing outputs and intermediate data for each data flow. Data items or data flows containing detected errors will be coloured red. Void data items will be coloured amber. The TW queries users and launches the analysis phase of the DW using a list of alternative components if they detect errors.	Partial or total RIA validation or reiteration of the DW analysis phase
Identification (on_click event) of erroneous final or intermediate outputs	List of components (catalogued component id and URL) Coloured Petri net with data flows among components List of components that generate unexpected data List of data flows that generate unexpected data	Definition of the range of each data type used (parameter value coverage lists created for each data type)		List of keywords associated with the failed component in order to search the repository for equivalent components

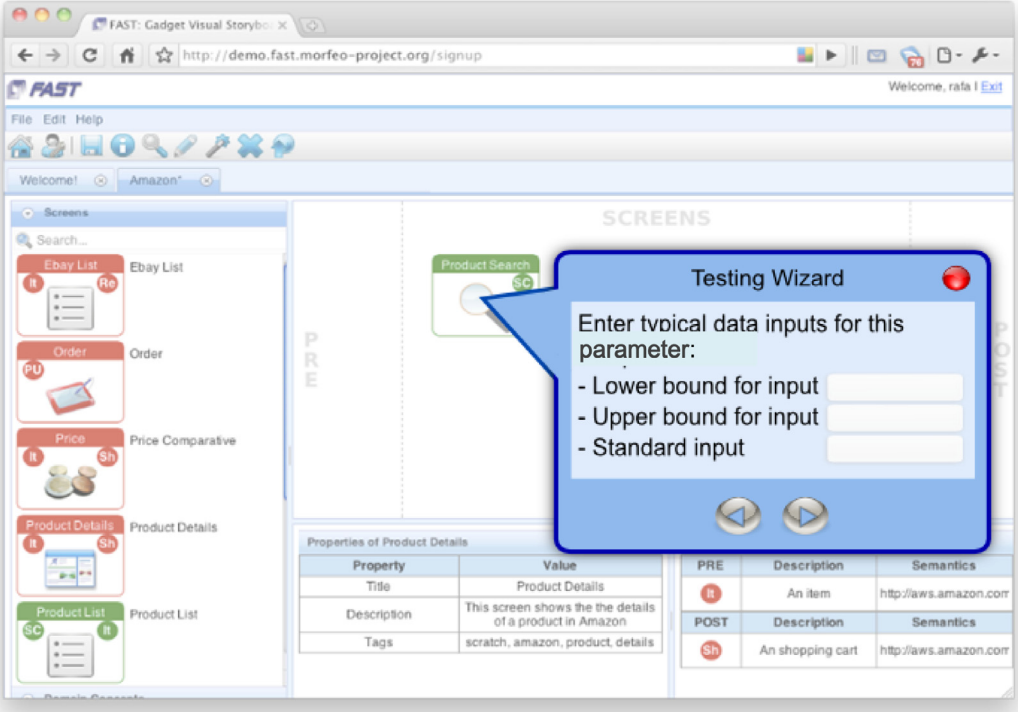

Fig. 3. TW test data input prompt for each use case to be tested.

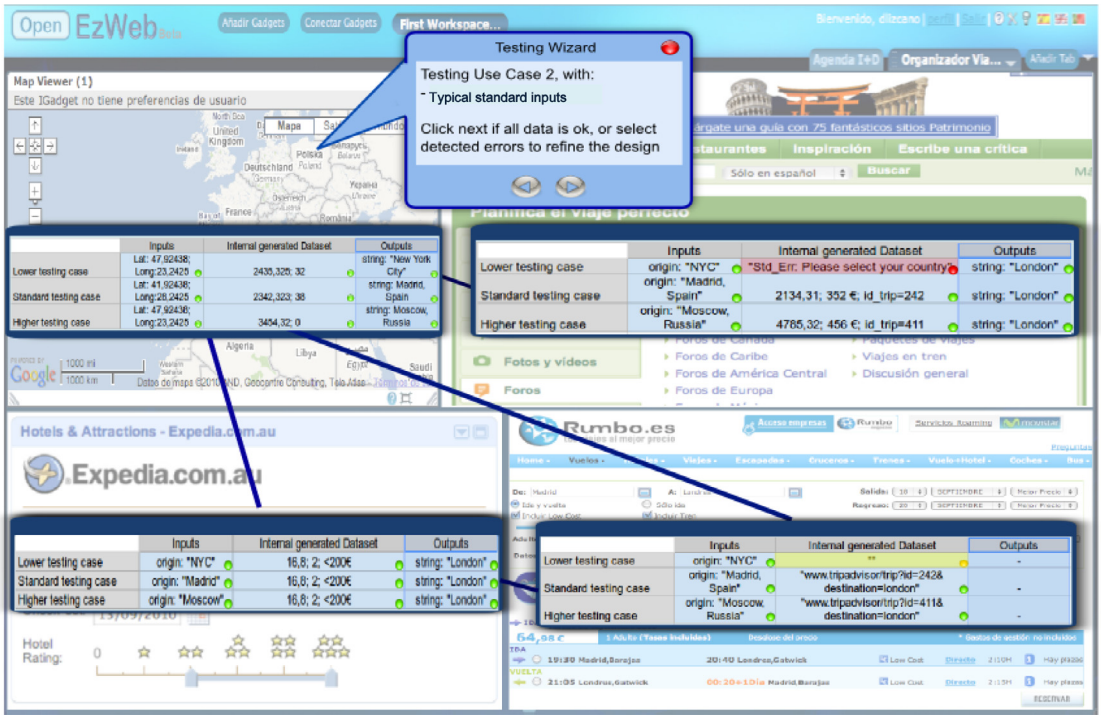

Fig. 4. WYSIWYT tables displayed to users.

acceptance tests are required. Besides, many semantic constraints, such as what type of numbers or strings are allowed, are specified by the domain ontology underlying the TW, as well as by the internal component code provided by the software or service provider, such as, for example, the return flight date must be later than the outbound flight date, a standard check performed by the component without user invention. The TW is currently a prototype, applicable to components whose inputs are numerical (integers or decimals), dates, text strings and geolocations. Boundary value analysis is applicable to all these data types. Boundary value analysis, described in (Pressman, 2010), proved to be useful in the tests conducted using the prototype. In order to test a range of values, boundary value analysis checks how the program behaves at the edges of the valid partitions containing the selected data set. Boundary value analysis is mostly used to check ranges of numbers and dates. Therefore, we used boundary value analysis to validate each test case also using standard data input (representatives of routine RIA use for the EUP). There are several approaches for running boundary value analysis tests on text strings. Some authors advocate converting the alphanumerical parameters to their UTF-8 or ASCII alphanumerical codes and running the boundary value analysis tests as on any numerical parameter (Jain et al., 2010). We take a similar approach, albeit based on the features that are usually troublesome in web component executions: we test the null string, a minimum length string, a maximum length string, a string containing one character less than a maximum string, and a standard string input by the user. And for two-value geolocations (latitude and longitude), we use the North Pole as the baseline, and test the minimum latitude-longitude values (0,0), the maximum globally meaningful values, a location close to the North Pole located in the Northern Hemisphere and a location close to the opposite pole in the Southern Hemisphere, a user-specified location, plus the current location of the TW if the browser has access to this contextual information.

Therefore, the TW uses five RIA input values as application inputs and runs the application. It generates a visual table showing, for each input combination, the final visual result of the execution, the outputs and the intermediate values that they generate. For example, Fig. 4 shows an acceptance test that uses standard values for each component. The user visualises tables with the final and intermediate values in order to validate the test case. If the data item does not match the target data type, the TW colours the respective item red. Correct data types with a "suspect" value (like, for example, a null data item) are coloured amber. These colours are what constitute the surprise within the deployed Surprise-Explain-Reward system. Users, acting as RIA developers and end users, should either accept validated test cases or check the box at the side of any error that they detected.

The TW only generates the five acceptance test cases related to the above five options; however, it can be configured to run more acceptance tests. These are the minimum five combinations run as part of a standard boundary value analysis test for black-box testing programs with numerical inputs in software engineering (Pressman, 2010). Although this is only a subset of the universe of possible combinations, the tests were representative enough to find most of the bugs during prototype testing (as described in Section 4). The same philosophy applies to boundary value analysis techniques designed for both numerical inputs and non-numerical inputs. The procedure for our string attributes is: a) if they are character strings subject to a finite set of possibilities (for example, a list of months of the year or a list of destination airports located in Europe), the applicable endpoint values will be the first and last item of the ordered list (January and December in the first case where there is an established order and Amsterdam and Zurich in the second where the order is alphabetical); b) if they are arbitrary strings, the user will have to accept the data flows generated with the endpoint values used in the component tests, that is, a null string, a minimum non-null string (a character) and a maximum string. In any case, the TW could be configured to run many more than just these five combinations. The problem in this case is that the Surprise-Explain-Reward mechanism may not be rewarding enough for the user to actively engage in visually analysing so many data combinations, considering that EUPs are generally casual programmers.

Table 5
Comparative analysis of RIA verification and validation by EUPs with and without the TW.

Stages		Without TW Problems	With TW Benefits	Drawbacks
Test case generation		Users are not usually experienced enough to generate test cases. Users tend to run unsystematic acceptance tests of one or two complex, common or prototypic cases. This may be insufficient.	The TW systematically generates test cases from the problem natural language description. The test cases account for the entire range of RIA functionalities.	The user is obliged to spend time and effort on use cases that may be uncustomary in the RIA domain.
Test case execution	Component tests	Users are generally unfamiliar with component data and maximum and minimum data. Neither do they know how to use scripting to systematise or automate tests. They would have to manually test data using each component at run time and check the results against the benchmark. It is very time-consuming for EUPs to have to perform this repetitive task manually, and they end up losing interest and do not run the tests.	TW runs these tests automatically and is capable of identifying errors in software components which are regarded as black boxes.	Some boundary data errors may lead to a component not being used even though the user had no intention of using the respective data in the application domain. It is only valid for standard data types in the current prototype.
	Integration tests	Users are unfamiliar with syntactic and semantic data flows between web components. Therefore, all they can do is unsystematically perform imprecise acceptance tests for small groups (or prototypes) of interconnected components. Generally, EUPs find integration testing for complex RIAs (more than five components) hard to do, because it involves building and testing partial prototypes.	The TW automatically tests each pair of components belonging to a partial or total RIA prototype, discovering integration errors that could otherwise remain hidden.	It involves pre-processing components in order to tag their inputs and outputs and thus identify possible data flows. It is only valid for standard data types in the current prototype.
	Acceptance testing	Users are able to perform full acceptance tests that may indicate that there is an error. However, they will not be able to identify the source of the error. Users do not usually have enough information to trace the source and cause of and correct the error (by altering a data connection or replacing a component), as they do not have the intermediate execution data.	The TW includes a Surprise-Explain-Reward mechanism that walks the user through structured data input for each test case and displays visual information on the intermediate and final data, as well as recommendations or tips for solving the identified problem.	The TW uses visual aids which may be insufficient for RIAs containing a lot of components. The acceptance testing mechanism is time consuming, and may be viewed by users as a waste of time. The help may be unsuitable if the component generates an erroneous intermediate data item for which there is no alternative in the catalogue.

3.3. Comparative analysis of RIA verification and validation by EUPs with and without the TW

As shown by the user study conducted according to the evaluation criteria reported in Section 4, EUPs are unable to develop reliable RIAs. This is a significant barrier to efficient job performance and their participation in Web 2.0. The proposed system aims to overcome this obstacle.

Table 5 describes the problems of verifying and validating a RIA without the TW. It also summarises the benefits and drawbacks of using the TW. As illustrated in Table 5, both test case generation and execution is troublesome for EUPs because they are unfamiliar with testing methodology. The big advantage of using the TW, whose positive results are reported in Section 4, is that it automates component and integration testing and uses the Surprise-Explain-Reward methodology for acceptance testing. EUPs are asked to visually inspect the displayed input, output and intermediate data generated by each component and check that they are consistent, that is, Surprise is used to get the user to identify and correct errors. The TW marks any detected inconsistency among target data types by means of a red visual warning sign. It shades any out-of-range or null values amber. If users detect an error, they are asked to click on the first intermediate value that they suspect to be erroneous or wrong. The TW uses the graph of components and internal coloured Petri net built by the TW to infer which component or components are related to the data

item. This is the Explain part of the system. It helps users to trace the surprise data item by inferring the components that generated this item. The TW notifies the DW, the component is reanalysed, and users are asked to check its parameters and components. If the user is unable to detect or explain a component error, the DW uses the catalogue to suggest other elements that perform the same function (meet the requirement inferred from the user's natural language description). If a component fails, its associated tags are used to run another search of the folksonomy, and users are given a new list of components retrieved by the search to examine. As each component includes its description, tags, inputs and outputs, users can choose the one that best meets their needs. Remember that there are countless end-user components available in repositories like Programmable Web or Github and many alternatives for all possible components. Additionally, the component inputs and outputs and behaviour are again visually highlighted. This helps users to understand which intermediate data they should get based on the final data destination. Then the TW is activated in order to run the component, integration and acceptance tests again. If the error is not corrected (there is no other similar component or the problem is not detected), the iteration ends. The user is asked to reiterate the WEUSE life cycle in order to refine the requirements description and component analysis conducted in the previous iteration. If the error is corrected, users then have the Reward of having corrected the error, clearly indicated by the green-coloured final and intermediate data.

4. User study: analysis and discussion

Elsewhere (Lizcano et al., 2013, 2012 and 2011a, 2011b) we presented a framework (FAST) supporting a WEUSE life cycle. This framework has a built-in wizard (DW) that walks EUPs through the activities and phases of this software life cycle and empowers EUPs to develop a solution that meets their needs. In this paper, we present the WEUSE verification and validation stage using a wizard (TW) that drives this stage. This TW helps to debug solutions, which is especially important for users intending to publish a good quality solution in a catalogue for future use by other users. This user study aims to examine the following evaluation criteria (EC):

- EC1: Are EUPs that develop RIAs without the verification and validation stage led by the TW able to produce reliable RIAs? What errors do they make? Do these errors compromise the solution to their particular problem or reuse by future users?
- EC2: Are EUPs that undertake the verification and validation stage led by the TW as outlined in this paper able to produce reliable RIAs? Are they able to detect and correct errors, bugs and design and implementation complications for a particular problem?

In order to check the above criteria, we stated a relatively complex typical problem (about 22–24 components) and asked two equivalent samples of EUPs to solve the problem. One sample used only the DW, which is part of the FAST framework that drives the WEUSE reported in (Lizcano, 2012; Lizcano et al., 2013), and the other used the combination of DW and TW in the FAST framework as described in this paper. Both samples received comparable training on the tool that they were to use, as detailed later.

We aim to list, characterise and observe the severity of the errors and failures found in the reported results for both samples in order to check the two evaluation criteria. This way, we can check, in response to EC1, how many errors the sample not using the TW made and whether the errors compromised component execution and reuse. Similarly, we check, in response to EC2, whether the other sample managed to elude such errors thanks to the verification and validation system driven by the TW and whether testing and debugging took a lot longer.

The size of the selected sample is 120 EUPs, and it is divided randomly into two homogeneous groups that are not biased by gender, age, educational attainment, previous employment or experience, as shown in Table 6. As shown in Table 6, none of the users have programming skills, except for four that are knowledgeable about mashup tools (in this case iGoogle) and two with high-level Enterprise Service Bus (ESB) knowledge. Table 5 also shows how the sample was divided into the two sets of users under study (60 that did not use the TW and 60 that used the TW). The sample was recruited by the FP7 Fast project consortium and the NESSI (Networked European Software and Services Initiative) platform through an informative web portal and invitations sent out to work and research centres, magazines, etc. A very large sample was gathered, from which 120 end users were selected. The selected end users resembled typical web-active EUPs. These users were invited to two events sponsored by the NESSI platform and by the Service-Front End Alliance led by the FAST project (a coalition of over 10 FP7 projects). They were offered free registration to attend the events and an entitlement to later beta use licences of the presented software products. These licences are attractive for small and medium-sized enterprises and end users related to the ICT world. A total of 64 people attended the Madrid meeting in 2012 and another 56 attended the meeting held in Brussels in 2013. These 120 separate users thus had the opportunity to participate in the hackathon targeting end users rather than programmers. The same training, instrumental and procedural steps were enacted at both English-language meetings.

An important step in this study was to validate the sample, statistically testing that the user characterisation of neither the entire sample nor the two groups was skewed. An ANCOVA study (Appendix B, Table B.1) confirms that the groups are not at all skewed with respect to the descriptive variables of the sample composed of the two groups. Additionally, a second ANCOVA (Appendix B, Table B.2) was conducted to analyse whether the errors made by each user were similar or depended on any of the qualitative or quantitative user characteristics. If they did, users within the sample of 120 individuals that had the respective profile could cause a bias. Again we did not detect any bias at all in the selected sample.

All EUPs were asked to build a web application to search and book transport and hotels, and gather tourist information for destinations entered in a personal organiser. Appendix A reports the proposed statement.

Fig. 5 shows an example of one of the successful solutions developed by users during the user study. Fig. 5 (top) is a RIA with two separate tabs. The left-hand tab (primary tab) shows a calendar workspace, with a personal calendar manager, a business calendar manager and information on each event. The right-hand tab (secondary tab) displays information for managing a trip to any event in the primary tab. This information is shown in Fig. 5 (bottom), which includes a map, a flight search engine, a hotel search engine, and a tourist information component.

As stated above, the requested application requires the use of from 22 to 24 components. Components include two mashups implemented as two different tabs for managing dates and times (calendar workspace) and managing the actual trip (organiser workspace), widgets, visual components, operators, backend services, etc. Component assembly requires the creation of 20 data flows among components. Users had access to component catalogues containing 650 components of different levels of abstraction. Therefore, they found it hard to locate the right components, integrate these components properly (there are over 200 possible syntactically correct data flows for the 22 to 24 components) and put together a valid solution. For a full and detailed description of

Table 6
Sample characterisation.

Characterisation	EUPs (120)	Without TW Group	With TW Group
Gender			
Male	62	31	31
Female	58	29	29
Age			
<20 years	22	12	10
20–34 years	28	15	13
35–49 years	26	12	14
50–64 years	24	11	13
>65 years	20	10	10
Educational attainment			
Secondary school	29	14	15
Vocational training	31	16	15
Bachelor's degree	29	15	14
Master's degree	31	15	16
Employment			
Student	33	16	17
Researcher	35	18	17
Employee	52	26	26
Experience and previous knowledge			
Mashup platforms	4	2	2
Web services (SOAP, ESB, BPEL, etc.)	2	1	1
HTML, CSS	0	0	0
Java, J2EE	0	0	0
JavaScript, AJAX	0	0	0
Php, ASP	0	0	0
OO programming	0	0	0
C, C++, C#	0	0	0
Scripting, Perl	0	0	0
Haskell, Prolog	0	0	0

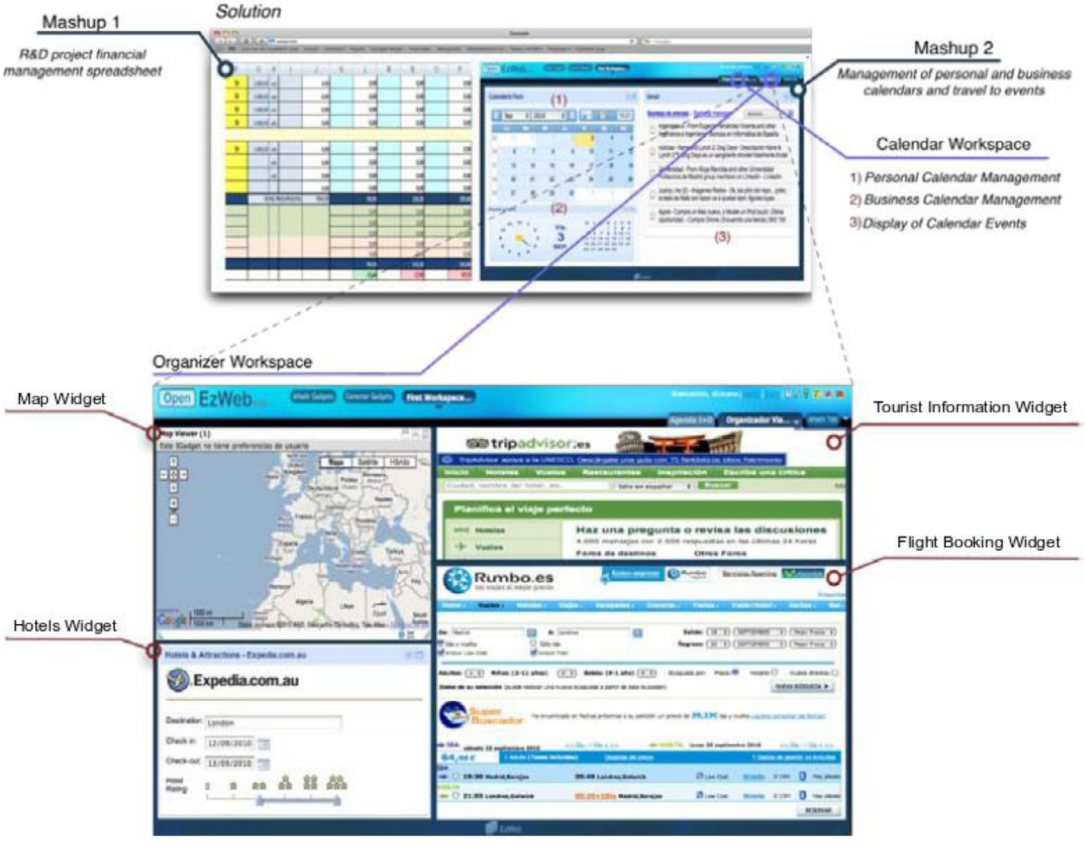

Fig. 5. Example of a final solution of the problem.

Table 7
Statistical data about development time taken with and without TW.

	Development time (minutes)							
Tool used	N	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	σ
Without TW	53	82.00	118.97	128.72	136.06	140.65	297.00	8.86
With TW	54	85.00	122.50	140.00	154.40	202.50	310.00	9.17

Table 8
Errors detected by experts in RIAs developed by users that did not use TW.

		Test level at which errors were identified by experts			
Error type	N	Component testing	Integration testing	Acceptance testing	User-detected errors
Auxiliary output errors	12			12	0
Components requiring parameterisation	3			3	0
Output constraint	8		8		0
Malfunction for negative latitudes	24	24			0
Total	47	24	8	15	0

these components and data flows and the development processes enacted by the sample of users using or not using TW guidance, see (Lizcano, 2016).

4.1. EC1: are EUPs that develop RIAs without the verification and validation stage led by the TW able to produce reliable RIAs? What errors do they make? Do these errors compromise the solution to their particular problem or reuse by future users?

The 60 non-TW users received basic training in end-user software engineering and specifically WEUSE. Of these 60 users, 32 were recruited in Madrid and 28 in Brussels, and the materials and teaching staff were the same in both sessions. The training schedule was:

- Theory session (4 h): introduction to and familiarisation with widget and mashup development, intercommunication technology, component validation using boundary value analysis-based black-box testing and application validation using standard input data.
- Practical session (4 h): solution development using the DW (2 h and 30 min) and manual testing (90 min)
- Case study (variable): hands-on laboratory experience of developing the proposed composite application.

After a development time of on average 130 min, 53 of the 60 EUPs managed to output a composite application that met the set requirements.

Table 7 shows the times taken by the normally distributed sample to develop this RIA. The times taken by the group that used the TW, whose work plan, training and activity is explained later, are added for interpretability.

A group of three experts, who were not involved in the study design and planning, ran structural and functional analyses of the 53 composite applications built by the EUPs. In order to reduce the workload and assure a detailed and conscientious analysis, the experts, computer scientists with extensive knowledge of RIA development, shared out the 53 applications, and each inspected about 18 applications. They conducted component, integration and acceptance testing (Lizcano et al., 2013). They also used white-box tests to inspect the source code and off-the-shelf components (taken from a component catalogue supplied by external software providers) used in each application. Although applications worked properly for the set requirements, several different types of errors were detected. Table 8 shows the statistical data on how many errors were identified in the RIAs built by this group of 53 EUPs.

As shown in Table 8, experts detected up to 47 hidden errors in the 53 RIAs that appeared to work properly. The list below characterises the 47 detected errors:

(a) Errors in auxiliary outputs that are not detected because they are not the main target output: 12 errors. They were detected in acceptance testing. They are outputs that are generated incidentally and not directly used by users. These outputs may be necessary if the component is reused in the future, and they could generate an unexpected error.
(b) Components that require parameterisation and are incorrectly configured by users: three errors were detected. They were also detected in acceptance testing. One of these errors occurred when a filter operator was applied. This operator processes a set of RSS inputs and either produces or deletes outputs containing items related to a keyword. One user set up the component to do the opposite to what he really wanted it to do. Another two errors were due to the component configuration. The component was set up to use a different date input than other sample users used. The component expected a YY-MM-DD format, whereas users used a DD-MM-YY value. The solution is either to set up the component to change the format or use an intermediate operator to perform this operation. This is not however something that users can do unaided.
(c) Component refinement to conform to an output constraint which had to be an integer and output a floating-point number. Eight errors were detected during integration testing by the group of experts. The result is correct, but does not conform to the element constraint type. This may lead to errors in future developments.
(d) Google maps-based component found to malfunction at negative latitudes. Twenty-four errors were detected in component tests conducted by the group of experts. It is an element that visualises a particular region of a map. Intensive tests detected that the map displays a mirror region of the Northern Hemisphere in the Southern Hemisphere because it processes the signs incorrectly. This incident had not been identified because users were located in Europe (and the provider in the USA).

Table 9
Statistical study examining EC1.

N	Detected errors				
	a) Wrong aux. out	b) Configuration error	c) Data type error	d) Data semantics error	Were there any undetected errors?
26	-	-	-	-	No errors were detected in 26 applications
8	-	-	-	Yes	
8	Yes	-	-	Yes	
6	-	-	Yes	Yes	27 applications had at least one detected error
2	Yes	Yes	-	-	
2	Yes	-	Yes	Yes	
1	-	Yes	-	-	
Total	12	3	8	24	

Therefore, the 47 detected errors can be divided into one of the above four error types: a) wrong auxiliary output, b) configuration error, c) output data type error or d) semantic errors in special data types. Table 9 shows the distribution of these errors across the 53 applications.

Of the 53 applications, no errors were detected for 26 applications, and 27 applications had at least one of the above errors. Looking at the failures, we wondered:

- How many applications could produce semantic or functionality errors in future uses? Applications containing error a), totalling 12 RIAs.
- How many of the 27 applications with detected errors will malfunction for their end user? Applications containing fault b), totalling three RIAs.
- How many of the 27 applications with detected errors could produce potential integration errors in future uses after publication by the user? Applications containing fault c), totalling eight RIAs.
- How many of the 27 applications with detected errors could produce more errors and bugs in future uses after publication by the user? Applications containing fault d), totalling 24 RIAs.

In conclusion, 24 out of the 53 applications appeared to operate properly for the purpose that they had been built by users (as they did not contain faults that would prevent the application from running). Nevertheless, they did contain errors. After a thorough study, therefore, only 26 were apparently error-free applications that could be safely published for future use (49% of the 53 applications that were examined), whereas a non-negligible 45% of applications pose problems for future use by other users. This suggests that the WEUSE development process requires a wizard to help EUPs remove errors. This is the aim of the TW-led verification and validation stage in WEUSE.

We found that a fair number of errors appear to be related to the poor quality of the components used in the composition, which end-user programmers can do nothing about. Also quite a few errors are related to a single component: Google Maps. This raises the question of whether the catalogue might have had an impact on the results of the study. As we will see in Section 5, the catalogue is, according to the Programmable Web visits and use ranking, fed by the most commonly used components in the field of EUP and is the most popular catalogue resource in the domain of EUP tools.

4.2. EC2: are EUPs that undertake the verification and validation stage led by the TW as outlined in this paper able to produce reliable RIAs? Are they able to detect and correct errors, bugs and design and implementation complications for a particular problem?

The next step in the statistical user study is to check the results of the reported TW. In order to assure the validity of the later results, we first have to check that the group using the TW is equivalent (not statistically skewed) to the group that did not use the TW. To do this, we conducted an ANCOVA of the user group dependent variable against the descriptive variables for each user. Appendix B reports the results of this ANCOVA, which shows the validity of the division into two groups. Note that the data output by the ANCOVA has no need of further validation, for example using post-hoc Tukey HSD tests.

The 60 users of the TW group received the same training in EUD and specifically web EUD. The training schedule was:

- Theory session (4 h): introduction and familiarisation with widget and mashup development, intercommunication technology and component validation using boundary value analysis-based black-box testing and application validation using standard input data values.
- Practical session (4 h): solution development on the DW (two and a half hours), and TW tutorial explaining how to interact with the TW, illustrating its interface, how and where to enter the requested data and how to interpret the colours used to indicate surprise in our What You See Is What You Test mechanism (90 min).
- Case study (variable): hands-on laboratory experience of developing the proposed composite application.

This group of 60 users were set the task of solving the same problem using the TW. The task was completed by exactly the same percentage of users as in the experiment without TW. However, it took these 54 users substantially longer to do the job, as shown in Table 7. This increase in development time is caused, as statistically proven later, partly by the interaction of the user with the TW.

Again the same group of three experts thoroughly analysed the resulting products in search of errors. This does not pose a threat to the study, as the fact that they were already acquainted with both the problem and the application type was immaterial. In actual fact, it would be less time-consuming for them to detect errors, which is their only goal.

Table 10 shows the errors detected by the TW in each phase, which are exactly the same as the errors that it managed to correct.

On this occasion the experts found no more than two errors in the RIAs built by the users. The TW had detected almost all the errors made by users, and the application was distributed and

Table 10
Statistical data about TW performance during the user study.

	Errors in the development				
	Testing level				
Support tool	N	Component	Integration	Acceptance	Detected	Corrected
With TW	50	26	7	17	48	48

Table 11
One-way ANOVA test for development time (in minutes).

One-way ANOVA Summary statistics	$\alpha = 0.05$					
Groups	N	Sum	Mean	Variance		
Without TW	53	416	66.06	10.14040816		
With TW	54	423	84.40	11.8044898		
Source of variations	Sum of Squares (SS)	df	Mean Square	F	P-value	F critical
Between group	0.49	1	0.49	8.3307426	0.00446573	3.938110878
Within group	1075.3	93	10.97244898			
Total	1075.79	94				

published. The two undetected errors were both related to date-based data types, where the user linked outputs using YY-MM-DD with DD-MM-YY data types, causing a similar number of errors as in the experiment without the TW. The TW displayed the intermediate data generated in the boundary value analysis tests, but the users did not notice the problem. The number (48) and type of errors detected by the TW was actually very similar to the bugs identified according to EC1 (where EUPs made 47 errors). For a description of this study, see (Lizcano, 2012). This time, however, the TW corrected bugs automatically during component and integration testing (detecting internal component errors and suggesting alternative components that did pass the tests) and helped the user to detect and correct errors during acceptance testing. This high TW success rate has been confirmed in real EUP development portals. Preliminary data from the field applications to which we have had access suggest that around 90% of the errors appearing in other similarly complex applications, components, data flows and problems tend not to appear if the TW is used (Hoyer et al., 2013). Additionally, the two undetected errors could have been put right if the end users had looked at the erroneous data flow caused by the mistaken date format. Subsequent versions of the TW should remedy this weakness.

To conclude this part of the study, we need to check the development times with a *t*-test or ANOVA. We conducted an ANOVA (Table 11) in order to compare the development time taken to solve the proposed problem with and without the TW and see the impact of TW use on development. The first step was to conduct a linear correlation study of the quantitative variable measuring the development time with and without the TW, and this study returns a value $cor = 0.95254468$ (close to 1). The correlations, which are close to 1, suggest that development time is statistically dependent on the tool used.

Looking at Table 11, we find that the F value 8.33 is much greater than its critical value of 3.9381. This means that, at a significance level of $\alpha = 0.05$, the variation of the mean development times can be said to be due to the specific tool used (with or without TW), and this variation is statistically significant. The mean development time is almost 20 min longer, which is a near 28% increase with respect to the original RIA built without the TW. To check whether this increase is statistically correct and common to all developments using the TW, we would have to tackle a wide range of different problems. This is beyond the scope of this paper. Suffice it to say that there is a significant increase in the time taken by EUPs. However, the high error rate suggests that users will find it worth their while to assume the additional workload of using the TW in order to build quality products.

Users sometimes need to get the job done as quickly as possible, even if they make mistakes along the way. For example, even though it might prevent reuse in the future, a bug in an application may be tolerable as long as it has no impact under the present circumstances. If so, users would put off finding and fixing the bug until later. Therefore, the FAST tool offers users the option of using or not using the TW: the user can apply just the DW in situations where speed is paramount or use the TW as well in circumstances where errors would not be tolerable.

Note that users who did not have access to the TW did not execute a big enough set of tests and test cases to detect the errors, as most (over 90% of the sample) only used a single validation test based on a standard value entered as input for each generated RIA input port. This is the standard test enabled by countless EUP tools (Yahoo! Pipes and Dapper, Kapow Katalyst software, and so on), which this study proves to be insufficient.

Finally, note that the proposed TW is a prototype that is undergoing improvement for inclusion in the FP7 WireCloud as part of a cluster of FI-WARE projects (FI-WARE Project, 2012). This will provide for further validation. Note that the two unbiased samples of 60 users used in this study are significant in both cases. It is noteworthy that the users who did not use the TW failed to detect 47 errors, whereas the users who used the TW detected and corrected 48 errors.

4.3. Opinions of users and experts about the proposed verification and validation process

At the end of the evaluation, we conducted a qualitative survey of each of the two samples of 60 users that participated in the evaluation [Lizcano 16]. This survey included five items to be rated on a five-point Likert scale (0—totally disagree, 1—disagree, 2—neither agree nor disagree, 3—agree, 4—totally agree), plus a couple of open questions about their general impressions of the verification and validation process. The items were:

1. I found it easy to detect errors using component tests.
2. I found it easy to detect errors using integration tests.
3. I found it easy to detect errors using acceptance tests.
4. I was qualified enough to enact the RIA verification and validation process with all the training I received before the experiment (theory session—4 h), including (for TW users) the instructions given by the TW during the user study.
5. End-user software requires verification and validation.

Table 12
Subjective survey completed by both samples.

#item	Users without TW (60)			Users with TW (60)		
	Mean	Std. Dev	Std. Error	Mean	Std. Dev	Std. Error
1	1.00	0.222	0.010	3.75	0.631	0.095
2	0.85	0.470	0.038	3.85	0.779	0.050
3	0.05	0.345	0.015	3.05	0.114	0.419
4	1.85	0.644	0.774	3.35	0.054	0.748
5	2.90	0.015	0.080	3.90	0.453	0.734

Table 12 shows the results for each item.

As Table 12 shows, the EUPs who did not use the TW found it almost impossible to perform component, integration and acceptance tests. Neither did they consider themselves to be qualified enough to enact the verification and validation process (as shown by the results for item 4). On the other hand, TW users found it much easier to perform all the tests (component and integration tests were, of course, simpler to execute than acceptance tests, whose difficulty level was rated as acceptable, because component and integration tests require less user intervention). Finally, the experience appears as a whole to have had a positive impact on the sample using the TW, as they consider it to be important to perform verification and validation. A possible explanation for this is that they realised just how many errors they managed to detect and correct thanks to the TW. On the other hand, the sample that did not use the TW does not consider testing to be as important, perhaps because they were unaware during testing of how many uncorrected errors there were in the RIA (they did not learn about these errors until the experts analysed the RIAs after they had taken the survey).

Regarding open questions, each sample was asked about the strengths and weaknesses of the proposed RIA development process that they were asked to apply during the experiment:

- The majority of the sample that did not use the TW stated that the listed tests were too hard to perform unaided. With respect to the strengths, they pointed out that the DW had been very useful for building the RIA, and component-based software appeared to be easier to test than monolithic software.
- The majority of the sample that used the TW stated that the visual system prompting test data and colouring the final and intermediate data output in response to the data inputs is very useful for testing purposes. They also mentioned that the DW was helpful for developing the RIA. With respect to the biggest weakness, the majority stated that it took too long to perform acceptance tests and suggested that they should be automated. (A priori this does not appear to be feasible as it is hard to predict the operation of a RIA composed of unspecified parts selected from a catalogue containing hundreds of available components.)

These results are not statistically significant enough to conduct ANOVAs to check whether there are differences of opinion with regard to each item. They do, however, illustrate the strengths and acceptance of the proposed TW.

Additionally, we consulted a panel of five WEUSE experts (from among the partners of the FP7 projects in which the proposed TW is to be adopted) to gather their opinion on the proposed TW system. Beforehand, we detailed how the system was structured and worked and presented the results of the experiment. The experts agreed that the Surprise-Explain-Reward-based system with the built-in TW is well aligned with web end-user development needs. One expert stated that EUPs should be given even more visual and textual help for RIA acceptance testing after component integration. The experts also positively rated the implementation of the TW for each testing level, but suggested that data type testing should be more aligned with the problem domain. These issues have been taken up as future research lines.

Additionally, the results of the user survey were, they thought, understandable. With respect to the results for item 5, they pointed out that the fact that users who did not use the TW were unaware of the need for proper program verification and validation and were keen to have instant use of a program to solve their problem, led them to consider this activity as unimportant.

5. Discussion on threats to validity

This discussion of threats to automatic verification and validation in WEUSE refers to five aspects of validity, which can be summarised as follows:

- Construct validity: This aspect of validity reflects the extent to which the operational measures that are studied really represent what the researcher has in mind and really meet the evaluation criteria. The threat would be that the research failed to demonstrate that the use of the TW to automatically verify and validate a RIA produced a highly reliable product. This threat is mitigated by the type of statistical studies conducted and the fact that the collected data have normal distributions. This guarantees the applicability of these studies.
- Internal validity: This aspect of validity is of concern when examining causal relations. When the researcher is investigating whether a factor affects an investigated factor, there is a risk that the investigated factor is also affected by a third factor. There is a threat to internal validity when the researcher is not aware of the third factor and/or does not know to what extent it affects the investigated factor. User recruitment is the source of this type of threat: in this case, the participants are end users recruited via the web portals of the partners of the projects and platforms set to use the TW in the near future. As it was not possible to bring the 120 users face to face at a single event, there is a threat that the group of 64 users in Madrid received different training or experienced different conditions than the 56 users in Brussels. We tackled this threat by having the same people conduct exactly the same experiment at both venues, providing the same training and using the same material, same organisation and same tools. There is also the possibility of the 64 users from one venue having interacted with the 56 at the other, as the experiments were not simultaneous. According to the results of the ANCOVA study (Table 1) in Appendix B, characterised with respect to the detected errors, the sample was found not to be biased, which it would have been if there had been such an interaction. Additionally, we believe that the reward offered to users for participating in the study (free user accounts for the beta version of FAST and beta licences for all the software presented at the congresses, as well as free registration for the event) was proportionate. This counteracts the threat of compensation causing selection bias and potentially invalidating the study. Another possible threat in this respect is of the training received by study participants closely resembling the tasks that they have to do. In order to rule out this threat, the participants were given general-purpose training on the tool and were not asked to develop solutions similar to the problems that were used in the study.
- External validity: This aspect of validity is concerned with the extent to which the findings are generalisable and how interesting the findings are to other people outside the investigated case. This is a possible threat because the user study is based on a single web application and, considering the success of the TW, the stated problem type, which is typical of web applications, is well adapted to the wizard philosophy. As explained

under future research, findings will be more generalisable in the future, as the research project using the TW will provide more quality and error-free information gathered from real-world developments using the presented prototype in the coming years. Another possible external validity threat is that future TW users will not receive the training that the users participating in the experiment were given. To prevent this validity threat, we published a free 30-min tutorial on TW (which was used in our study) on the YouTube channel of the FP7 projects related to this paper. We also posted 8 h' worth of seminars by means of recorded hangout sessions to provide potential TW end users with all the information that they require.

- Reliability: This aspect is concerned with the extent to which the data and the analysis are dependent on specific researchers. There is the threat of fewer errors being detected than if more than one expert analysed the RIAs (each expert analysed 20 RIAs) using more demanding coverage types than parameter variable coverage. In this respect, note that we selected accredited experts with lengthy experience in the sector that were not directly involved in this study or concerned with its working hypotheses in order not to condition the data and data analyses. Additionally, the catalogue and components were not developed by the work group that conducted this research. Therefore, the testing environment is representative of what an EUP using such a development tool will come across. These catalogues are maintained by third parties (like Programmable Web), fed by service and resource providers and ranked by popularity and use. Thus, the research group has had no bearing on this possible reliability threat.
- Potential use: there is a trade-off between the time taken using or not using the TW and the resulting quality. We decided to make TW use optional, but this poses the following question: Will users continue to use a technique that is more time consuming? It is unlikely to be used unless users are aware of the satisfaction and benefit to be gained from using the TW to correct existing or potential errors. We will discuss this issue in the conclusions.

6. Conclusions, limitations and future lines

In previous articles concerning WEUSE we have shown that EUPs have need of a development wizard to take them through the analysis, design and implementation stages to build their RIAs (Lizcano et al., 2012, 2013). This paper presents two main contributions.

First, it notes that one of the key weaknesses of RIAs built by EUPs is that they are not thoroughly verified and validated by means of prescribed testing activities. EUPs do not understand the verification and validation process and have need of support to tell them how to go about testing. A testing wizard is unquestionably a useful tool for users without programming skills. This paper provides evidence that applications built by EUPs are prone to errors. These errors affect the RIA that they have built and, if they are published in a catalogue, compromise future reuses of the solution by other users. As component and application reuse and social sharing is the cornerstone of the network effect empowering all EUD approaches, it is vital to assure that the catalogue components are error free.

Second, the paper puts forward an automatic framework to support verification and validation that has been demonstrated to be valid. The reported user study shows that the better results with respect to EC2 can be attributed to the adoption of this TW tailored for EUPs. It is the TW that makes the difference between the results for EC1 and EC2.

The results in the field of WEUSE are promising. Proof of this is that the DW used in combination with the TW has been built into the FAST tool and is being used by Spanish public administrations to promote digital spaces of interaction with citizens using EUD technologies. Saragossa citizens and visitors have access to a portal that they can use to build their own composite web application and organise their leisure activities in the city, book hotel rooms, museum tickets, put together their personal tourist guide, book entertainment, etc. (Tejo-Alonso et al., 2015). There is a catalogue of components for locating addresses on the city map, and accessing tourist, historical, hotel and restaurant information. This portal has about 500 components, over half a million total visits and composite web applications and receives about 100 visits per day. There are no quantitative statistical studies, but qualitative surveys suggest that acceptance by citizens is very satisfactory (Lizcano, 2016). Thanks to the reported TW, the error rate has dropped sharply, whereas the quality of the developed elements has increased significantly. This improves the prospects for future reusability.

On the other hand, Andalusia's Regional Government (FI-WARE PPP, 2012) has set up a portal enabling public administration users to use FAST to build their own web applications. It has about 600 components, and citizens can build a web application in order to complete public administration formalities, such as tax payments, fine and traffic management, and official document, census, employment record management, city or intercity road traffic information, etc. With about 50 visits per day, it has been well received by citizens (Lizcano, 2016). They have used the TW presented in this article to build good quality web applications, as reported by external experts in (Lizcano, 2016). It is true that most of the applications built are very simple, as these users are not well acquainted with portal use. Most of the applications (76%) are based on at least five interrelated components. But 18% have as many as 10 components, and 6% include from 15 to 20 components, integrated by a host of error-prone data flows. This portal includes an optional 4-h massive open online course (MOOC) to acquaint users with the portal and with end-user programming.

Additionally, we are using and testing the proposed TW within two on-going FP7 research projects: 4CaaSt (4CaaSt Project, 2012) (Building the Platform-as-a-Service of the Future) as part of its Mashup-as-a-Service solution, and FI-WARE (FI-WARE Project, 2012) (Future Internet Core Platform) as part of its applications and services ecosystem and generic delivery framework enablers for EUPs to build application mashups. The project users are experts in wide-ranging domains and are proficient users of the Internet and office automation software. However, they have very little technical or programming knowledge. So we expect to study users as part of these projects in order to update and refine our approach. The use of FAST as part of the technology foundation (FI-WARE) behind the European public-private partnership on the Internet of the Future (FI-PPP, 2012) extends the life cycle to R&D projects covering areas as far apart as the transportation of goods and people, energy efficiency or bank management in the framework of smart cities.

The presented verification and validation mechanism has several weaknesses. First, it is a prototype that handles components whose possible inputs are confined to numerical data, locations, text strings and dates. Second, automatic component and integration testing coverage is based on the parameter value coverage method, and acceptance testing coverage relies on boundary value analysis, where users are asked to input standard data for the problem to be solved. The proposed RIA acceptance testing mechanism uses five test cases—all minimum values (feasible minimum for this problem established by the user), all standard values, all maximum values (feasible maximum for this problem established by the user), and values just above and just below the minimum and maximum—, but this testing coverage is unquestionably small compared with all the possible combinations of a high number n of

components (5^n). Finally, automatic testing implies an added workload for EUPs not used to spending time on RIA validation. The decision to use the TW is a trade-off between EUP development time and product quality. All these weaknesses, plus the fact that data type testing should, as far as possible, be aligned with the problem domain, require further research. We are now taking steps to try to reduce their impact on the automatic EUSE verification and validation process.

But the main line of future research is to study the feasibility of building a framework to support an integral life cycle (including both development and testing tasks) for other end-user approaches (apart from RIAs and web-based composite applications) and applying the life cycle to other fields such as spreadsheets, mail filtering, What You See Is What You Test web content development applications, etc., to enable users to build non-trivial and quality solutions.

The proposed TW is a prototype that, we believe, works well. However, further statistical studies with larger samples are required to better identify the strengths and weaknesses of this wizard.

Acknowledgements

The authors would like to thank the 120 EUPs of the user study for their interest, participation and time. This work has been partially supported by the EU co-funded IST projects FAST: Fast and Advanced Storyboard Tools (GA FP7-216048), FI-WARE: Future Internet Core Platform (GA FP7-285248), and 4CaaSt: Building the Platform as a Service of the Future (GA FP7-258862); and by the Pololas project (TIN2016-76956-C3-2-R) and by the SoftPLM Network (TIN2015-71938-REDT) of the Spanish the Ministry of Economy and Competitiveness. We would like to thank Fernando Alonso Amo for his support for and collaboration in the reported research, which has been very helpful for drafting this paper.

Appendix A. Statement of problem 1 for evaluation according to EC1 and EC2

As part of a R&D project in which he is participating, a higher education worker has to make numerous national and international trips. The project has several partners of different types and origins.

The R&D project has a web-based general agenda shared by all the project partners. All face-to-face meetings are posted in this agenda, specifying the meeting date and time, venue and agenda. The higher education institution employing the user actively cooperates with two travel agencies, one specialised in high-speed trains and the other in long-distance flights, and both manage all the travel and accommodation options at the full range of hotels.

1. The user consults the shared R&D project agenda every day to check whether there is a new meeting that he should attend.
2. If there is to be a meeting, he checks his personal organiser to find out whether he can attend the meeting and fills in the details of the new meeting, the meeting agenda, etc.
3. The user looks up the meeting venue, and searches for it on a map. Then, he accesses the travel agency services and checks what travel options they offer, as well as price. Normally he compares two options and chooses one agency or the other depending on the travel options, length of stay and price.
4. If the trip is to last longer than a day, the user searches hotels near to the meeting venue and checks the prices per room and night offered by the travel agencies.
5. The department employing the user has a spreadsheet-based software program that manages the department-run R&D project budget. It contains spreadsheets that can be used to check the travel budget currently available for each project and manage new expenses. It is the user's job to calculate how much the travel and chosen accommodation will cost, add this up, check that there is enough money available for the trip and deduct it from the project budget.
6. Then the user makes the bookings one by one.
7. Finally, the user checks the Internet for information about his destination, demographic characteristics, weather forecast, etc.

Appendix B. ANCOVA statistical study in order to validate the sample and its division into two equivalent unskewed groups and to analyse the impact of EUP characteristics on errors

Analysing Table B.1, we find that the coefficient of determination R^2 is very low (0.103). This suggests that there is a high percentage of variability in the modelled mean variable (user group) so that gender, age, educational attainment, employment and previous experience (the quantitative and qualitative variables for each individual) are equally random in the two groups. This value of R^2 and adjusted R^2 suggests that the division into groups is highly independent of the user characteristics (98.6%). The model error values, MSE (mean squared error) and MAPE (mean absolute percentage error), are very high (well above the ideal value 0), again suggesting that the model does not precisely explain the division into groups. Additionally, DW (Durbin-Watson statistic) values are not close to 0. This implies that there is no autocorrelation among the qualitative variables, without which the study would not be valid. Finally, Cp (Mallows's Cp statistic) suggests that the division should not be biased at all in 118 out of 120 observations. The model results for all the variables taken together validate the sample, indicating that there is no bias related to the qualitative and quantitative variables characterising the users and the group division for the study.

In the following we discuss the statistical study conducted to investigate whether or not there is a correlation between the descriptive characteristics of the EUPs and the errors during the study.

Analysing Table B.2, we find that the coefficient of determination R^2 is very low (0.098). Therefore, there is a high percentage of variability in the modelled mean variable so that gender, age, educational attainment, employment and previous experience (the quantitative and qualitative variables for each individual) appear to explain only 9.8% of detected errors. The other values are due to other unknown variables. This value of R^2 and adjusted R^2 suggests that the errors made by EUPs are largely (96.4%) independent of the user characteristics. The model error values MSE and MAPE are very high, again suggesting that the model does not precisely explain the behaviour of the errors variable in the sample. DW values are not close to 0 so there is no autocorrelation among the qualitative variables. Finally, Cp suggests that the model is able to exactly explain the results of only one of the 120 individuals. We have conducted a Type I and Type III sum of squares analysis. Taken together, the model results validate the whole sample, indicating that there is no bias related to the qualitative and quantitative variables characterising all 120 users and their recruitment for the study.

Looking at the $Pr > F$ values of the Type I sum of squares of the ANCOVA model, we find that the characteristic that is most related to the user errors is age (the greatest $Pr > F$ in the study, equal to 0.372). We examined user age and found no statistical evidence of a direct correlation between age and the errors found according to EC1 and EC2, as $Pr > F$ is still much lower than 1. In any case, as the sample and subgroups are balanced with respect to age, there are no problems of bias if the percentage of individuals in each age group is equivalent.

Table B.1
ANCOVA to validate the 60–60 division of the sample.

Goodness of fit statistics								
Observations	Sum of weights	df	R^2	Adjusted R^2	MSE	MAPE	DW	Cp
120	120	64	0.103	0.014	5.421	4.751	1.235	2

Analysis of variance					
Source	df	Sum of squares	Mean squares	F	Pr > F
Model	34	5.932	0.169	1.184	0.251
Error	65	9.072	0.142		
Corrected Total	99	15.004			

Table B.2
ANCOVA of errors against sample characterisation.

Goodness of fit statistics								
Observations	Sum of weights	df	R^2	Adjusted R^2	MSE	MAPE	DW	Cp
120	120	69	0.098	0.036	6.241	3.998	1.247	1

Analysis of variance					
Source	df	Sum of squares	Mean squares	F	Pr > F
Model	39	6.421	0.175	1.289	0.251
Error	65	10.112	0.139		
Corrected Total	119	14.001			

Computed against model = Mean(Y)
Type I sum of squares analysis:

Source	df	Sum of squares	Mean squares	F	Pr > F
2.- Gender	1	0.027	0.013	0.284	0.178
3.- Age	1	0.214	0.125	0.935	0.372
4.1- Education	3	0.841	0.352	0.984	0.287
4.2-Employment	2	0.211	0.100	0.589	0.269
5.- Experience and previous knowledge	28	4.274	0.205	1.417	0.169

Type III sum of squares analysis:

Source	df	Sum of squares	Mean squares	F	Pr > F
2.- Gender	1	0.027	0.013	0.174	0.154
3.- Age	1	0.214	0.125	0.857	0.241
4.1- Education	3	0.841	0.352	0.872	0.112
4.2-Employment	2	0.211	0.100	0.687	0.125
5.- Experience and previous knowledge	28	4.274	0.205	1.588	0.158

References

4CAAST PROJECT. Official web site.

Brambilla, M., Tokuda, T., Tolksdorf, R., 2012. Crowdsourced web site evaluation with crowdstudy. In: ICWE 2012, LNCS, vol. 7387. Springer-Verlag, Berlin, Heidelberg, pp. 494–497.

Burnett, M., 2009. What is end-user software engineering and why does it Matter? In: Proceeding IS EUD '09 Proceedings of the 2nd International Symposium on End-User Development. Springer-Verlag, Berlin, Heidelberg, pp. 15–28. ©2009 ISBN: 978-3-642-00425-4, doi:10.1007/978-3-642-00427-8_2.

Burnett, M., Sheretov, A., Ren, B., Rothermel, G., 2002. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. IEEE Trans. Software Eng. 28 (6), 576–594.

Yoon, B., Garcia, O., 1998. Cognitive activities and support in debugging. In: Fourth Symposium on Human Interaction with Complex Systems, HICS, p. 160.

Cai, L., Zhang, J., Liu, Z., 2011. A CPN-based software testing approach. J. Software, North Am. 6, 468–474.

Cao, J., Fleming, S., Burnett, M., Scaffidi, C., 2014. Idea garden: situated support for problem solving by end-user programmers. Interact Comput. 29 May 2014, doi: 10.1093/iwc/iwu022.

Chambers, C., Erwig, M., 2009. Automatic detection of dimension errors in spreadsheets. J. Vis. Lang. Comput. 20 (4), 2009.

Clermont, M., 2003. Analyzing large spreadsheet programs. In: Proceedings of the 10th Working Conference on Reverse Engineering,. Victoria, B.C., Canada, pp. 306–315. November 13-16.

Costabile, M.F., Fogli, D., Mussio, P., Piccinno, A., 2006. End-user development: the software shaping workshop approach. In: Lieberman, H., Paternò, F., Wulf, V. (Eds.), End User Development. Springer, Dordrecht, The Netherlands, pp. 183–205.

Davis, J.S., 1996. Tools for spreadsheet auditing. Int. J. Hum. Comput. Stud. 45, 429–442.

Fischer, G., GiaccardI, E., 2006. Meta-design: a framework for the future of end user development. In: Lieberman, H., Paternò, F., wulf, V. (Eds.), End User Development Empowering People to Flexibly Employ Advanced Information and Communication Technology. Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 427–457.

Fi-Ware Project, 2012, Official web site. http://www.fi-ware.eu.

FI-WARE PPP Office, 2012, Official web site. http://www.future-internet.eu/home/future-internet-ppp.html.

Hoyer, V., Fuchsloch, A., Kramer, S., Moller, K. and López, J. 2013. Evaluation of the FAST implementation. Technical Report D6.4.1, FAST Consortium February 2013. https://files.morfeo-project.org/fast/public/M24/D6.4.1_ScenarioEvaluation_M24_Final.pdf.

Igarashi, T., Mackinlay, J.D., Chang, B.-W., Zellweger, P.T., 1998. Fluid visualization of spreadsheet structures. In: Proceedings of the 1998 IEEE Symposium on Visual Languages. Halifax, NS, Canada, pp. 118–125. September 1-4.

Jain, A., Sharma, S., Seema, S., Juneja, D., 2010. Boundary value analysis for non-numerical variables: strings. Orient. J. Comput. Sci. Technol. 3 (2), 323–330.

Janner, T., Siebeck, R., Schroth, C, Hoyer, V., 2009. Patterns for enterprise mashups in B2B collaborations to foster lightweight composition and end user development. In: Proceedings of the IEEE 7th International Conference on Web Services. Los Angeles, CA, pp. 976–983. July 6-10.

Araki, K., Furukawa, Z., Cheng, J., 1991. A general framework for debugging. IEEE Software 8 (May(3)), 14–20.

Ko, A.J., Myers, B.A., 2004. Designing the Whyline: a debugging interface for asking questions about program failures. In: Proceedings of the ACM Conference on Human Factors in Computing Systems. Vienna, Austria, pp. 151–158. April 24-29.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., 2011. The state of the art in end-user software engineering. J. ACM Comput. Surv. 43 (April(3)) Article 21.

Kuttal, S.K., Sarma, A., Rothermel, G., 2013. Debugging support for end user mashup programming. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13). ACM, New York, NY, USA, pp. 1609–1618. doi: 10.1145/2470654.2466213 http://doi.acm.org/10.1145/2470654.2466213.

Lizcano, D., Alonso, F, Soriano, J., Lopez, G., 2014. A component- and connector-based approach for end-user composite web applications development. J. Syst. Software 94 (1), 108–128.

Lizcano, D, Alonso, F, Soriano, J, Lopez, G., 2013. A Web-centred approach to end-user software engineering. ACM Trans. Software Eng. Methodol 22 (October(4)). Article 36, 29 pages. doi: 10.1145/2522920.2522929 http://doi.acm.org/ .

Lizcano, D. 2012, Description of the development process enacted by surveyed users. Technical Report. http://apolo.ls.fi.upm.es/eud/solution_development_method.pdf.

Lizcano, D. 2016, Technical reports about EUSE and EUD. Available at http://apolo.ls.fi.upm.es/eud.

Lizcano, D., Alonso, F., Soriano, J., Lopez, G., 2011a. A new end-user composition model to empower knowledge workers to develop rich internet applications. J. Web Eng. 3 (10), 197–233.

Lizcano, D., Alonso, F., Soriano, J., Lopez, G., 2011b. End-user development success factors and their application to composite web development environments. In: Proceedings of the Sixth International Conference on Systems, ICONS 11. St. Maarten, The Netherlands Antilles, pp. 99–108. January 23–28.

Mackay, W.E., 1990. Patterns of sharing customizable software. In: Proceedings of the ACM Conference on Computer-Supported Cooperative Work. Los Angeles, USA, pp. 209–221. October 7–10.

Mørch, A.,, Mehandjiev, N.D., 2000. Tailoring as collaboration: the mediating role of multiple representations and application units. J. Comput. Support. Cooperative Work 9 (1), 75–100.

Obrenovic, Z., Gasevic, D., 2009. Mashing up oil and water: combining heterogeneous services for diverse users. IEEE Internet Comput. 13 (6), 56–64.

Panko, R., 1995. Finding spreadsheet errors: most spreadsheet models have design flaws that may lead to long-term miscalculation. Information Week (May) 100.

Pressman, R.S., 2010. Software Engineering: A Practitioner's Approach, fourth ed. McGraw-Hill Higher Education. R. S. Pressman & Associates, Inc, Columbus, USA ISBN: 0073375977.

Rothermel, G, M., Burnett, LI, L, C., Dupuis, Sheretov, A., 2001. A methodology for testing spreadsheets. J. ACM Trans. Software Eng. Methodol. 10 (1), 110–147.

Scaffidi, C., Shaw, M., and Myers, B. 2005. The "55M end user Programmers" estimate revisited. Technical Report CMU-ISRI-05-100, Carnegie Mellon University.

Segal, J., 2007. Some problems of professional end user developers. In: Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing. Coeur d'Alene, Idaho, USA, pp. 111–118. September 23–27.

Sutcliffe, A., Mehandjiev, N., 2004. End-user development: tools that empower users to create their own software solutions. Commun. ACM 47 (9), 31–32.

Tejo-alonso, C., Fernandez, S., Berrueta, D., Polo, L., Fernandez, M.J., and Morlan, V. 2015. EZaragoza, a tourist promotional mashup. Technical Report. http://idi.fundacionctic.org/eZaragoza/ezaragoza.pdf.

Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., Rothermel, G., 2003. Harnessing curiosity to increase correctness in end-user programming. ACM Conference on Human Factors in Computing Systems. ACM, New York.