Improving User Involvement through a Model-Driven Requirements Approach

José Matías Rivero, Esteban Robles Luna, Julián Grigera, Gustavo Rossi
LIFIA
Facultad de Informática – Universidad Nacional de La Plata
La Plata, Buenos Aires, Argentina
{mrivero, erobles, jgrigera, gustavo}@lifia.info.unlp.edu.ar

***Abstract*—Model-Driven Web Engineering (MDWE) methodologies have proven to be a mature way of developing web applications, dramatically increasing productivity during development. However, after more than a decade of evolution, the artifacts and processes used to gather requirements have not changed substantially. At the same time, the capacity of quickly adapting to emergent domain-specific requirements (a feature that became popular with the massive adoption of agile approaches) has become hard to achieve in these methodologies. In this context, in order to implement this kind of refined requirements as fast as possible, changes are usually applied directly to the generated application, losing the abstraction and its inherent productivity provided by the Model-Driven process. Another way of implementing this kind of changes is by extending the modeling language, but this implies a high effort and, again, a consequent productivity loss. In this paper we propose a model-driven development approach called *MockRE* that captures requirements using User Interface prototypes (mockups) that end-users can understand completely. The process and tooling presented here allows end-users to express requirements annotating the mockups with textual descriptions, and also generating a running application in the same way that MDWE environments do. Developers may later use these initial specifications placed by end-users as valuable model concepts that can be refined through direct coding in a non-intrusive way. Through this strategy, MockRE intends to make a more extensive reuse of end-users specifications throughout the whole developing process.**

***Index Terms*—model-based requirements engineering; model-driven development, mockups, agile**

I. Introduction

Model-Driven Web Engineering methodologies like WebML [1], UWE [2], OO-H [3] or OOHDM [4] have proposed an alternative, more productive way of building Web Applications, using models to describe their features and obtaining a final application through automatic code generation. These methodologies have effectively defined a less error prone development process while, at the same time, they have adapted their languages or *metamodels* to the current trends in the Web field (e.g. RIA features, business models, interaction patterns, etc.). However, they have not shown drastic improvements in the requirements gathering stage, neglecting new industry standard techniques like paper or digital mockups. In this context, for instance, WebML, UWE and OOHDME propose using Use Cases [2], [4], and OOHDME in particular added navigational specifications by using User-Interaction Diagrams or Scenarios [5].

Though these artifacts may be enough to capture the basic requirements of a Web Application, they have two disadvantages: (1) they use technical jargon (e.g. pre-conditions, post-conditions, flows, states, etc.), which is not easily understood by end-users and (2) the translation of these artifacts to models requires manual intervention of developers in a semi-automatic (when posible) derivation process. While the former can lead to misguided requirements that end-users are unable to correct, the latter can lead to well-known human errors in the translation process. As a side effect of this last problem, not having a clear and direct transformation between the requirements artifacts and the modeling concepts potentially threatens requirements traceability.

On the other hand, agile methodologies propose to reduce the requirements gathering stage by using short requirements specifications (e.g. User Stories [6]). In addition, combining User Stories together with user interface mockups has become a common trend in the industry [7] to gather presentation and interaction requirements that cannot be captured with standalone Stories. Since mockups represent an intermediate language between end-users and developers [8], they work both as way of describing concrete requirements for end-users and as a technical UI descriptions for developers. In [7] we presented a technique to annotate User-Interface models, helping the binding between requirements descriptions and concrete implementations (UI widgets), thus enabling requirements traceability during development.

To summarize, MDWE methods have not changed the artifacts used to capture requirements, leading to misunderstandings in the modeling stage that can harm the potential productivity increment that models use can provide.

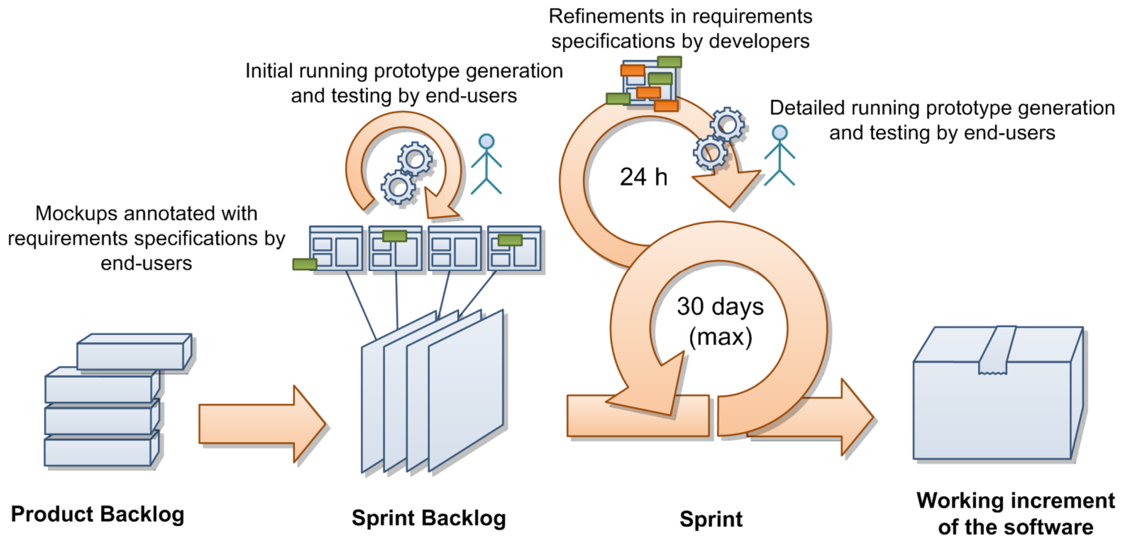

Refinements in requirements specifications by developers
Initial running prototype generation and testing by end-users
Detailed running prototype generation and testing by end-users
24 h
Mockups annotated with requirements specifications by end-users
30 days (max)
Product Backlog
Sprint Backlog
Sprint
Working increment of the software

Fig. 1. Overview of the Scrum-adapted MockRE process.

Agile approaches, on the other hand, make more emphasis in ensuring that requirements are correctly captured by using mockups and showing quick prototypes to end-users as fast as possible; as a result, they can adapt more easily to changes and detailed requirements through direct coding. However, the direct coding practice also leads to repetitive tasks (like coding CRUD operations [9]) and human errors.

As a result, neither MDWE nor agile approaches can, at the same time, use requirements artifacts that: (1) are easy to understand by end-users, (2) are capable of being fully traced in the final application, (3) allow to generate automatic runnable prototypes from its definition and (4) enables the incorporation of manual code in addition to the behavior generated from models through code-generation (for detail requirement specification) without breaking the models and threatening the traceability between the artifacts and the implementation.

In this paper we propose a development approach that supports these features in order to apply the agile principle of "*Working software is the primary measure of progress*". Since this approach relies heavily on mockups to gather requirements and specify features, we will call it *Mockup-Based Requirements Engineering* or *MockRE* for short. The rest of the paper is structured as following: in Section II we discuss some background to the topics discussed in this work; in Section III we introduce the MockRE approach both procedurally and technically, and in Section IV we summarize the content of the paper and also include some further work that we are pursuing.

II. BACKGROUND

Model-Driven Web Engineering (MDWE) methods have evolved greatly during the last 2 decades, and they have shown how the development of Web Applications can be improved using code generation and high-level models instead of direct coding. Some of the more remarkable methodologies are WebML [1], UWE [2], OO-H [3] an OOHDM [4]. In a similar way, model-based methodologies oriented to define interaction requirements intend to define and express how end-users interact with the application. One of these methodologies is WebSpec [10], which expresses user interaction using states (called interactions), transitions, pre-conditions and post-conditions. MoLIC [11], another interaction modeling technique, proposes to model interaction representing a turn-taking between user and designer, forming conversation threads with structures of topics and subtopics. In [12], stereotyped UWE activity diagrams are used to capture and model Computer-Human Interaction and non-interactive operations in order to generate content, navigation, presentation and process models. Without this support, this kind of MDWE models must be built linearly, leaving presentation and detailed interaction details to the end of the development, with the consequent risks.

User Interface mockups have become an interesting and useful tool in requirements elicitation, since they act as an intermediate language between end-users and developers [8]. Also, their adoption in development processes (especially in the requirements gathering phase) are evident through the plethora of tools that appeared during the last years like Balsamiq[1], Pencil[2], Mockingird[3] and MockFlow[4], among many others. Mockups, used together with User Stories, have proven to be useful in agile approaches to early assess usability issues, concretize requirements and also make cost estimations more precise [13], [14]. Apart from User Stories,

[1] Balsamiq Mockups - http://www.balsamiq.com/products/mockups - Accessed: 27-Apr-2013.

[2] Pencil Project - http://pencil.evolus.vn/ - Accessed: 27-Apr-2013

[3] Mockingbird - https://gomockingbird.com/ - Accessed: 27-Apr-2013

[4] MockFlow - http://www.mockflow.com/ - Accessed: 27-Apr-2013

mockups have been also used with Use Cases [15]. In addition, statistical studies have been conducted showing that mockups provide general improvements in the development process without imposing high costs to it [16].

In the Model-Driven context, User Interface sketches were used as a basis or to provide improvements in the modeling process. In [17], task models are created from annotated sketches. Also, in [8] mockups are used as a foundation to specify interaction requirements in a storyboard-like manner. The WebSpec approach [10] relies on mockups to specify interaction requirements. In addition, UI sketches assembled in a form-like fashion have been used to interact with end-users and generate content models [18].

In our previous work [7], [19] we propose to annotate mockups built with popular tools to quickly obtain content, navigation and presentation models, trying to break the classic linear modeling structure in MDWE approaches and, at the same time, pursuing a better end-user integration in the process using mockups as common language. However, in these papers, mockup annotations and tooling strategies were too technical for end-users to do the modeling or any part of it – except for mockup building. Also, since we relied in existing MDWE approaches like UWE and WebML, the approach was limited to the development infrastructure of those methodologies, which focus in specifying common patterns in Web Applications to improve the productivity during their development, but not tackling detailed requirements related to domain-specific businesslogic as in most applications. In this work we propose to tackle both aspects: on the one hand, we use a textual annotation tool that allows end-users to annotate HTML mockups interactively through wizards when necessary, without requiring any technological knowledge. On the other hand, we propose a code-based and non-intrusive way of describing and implementing detailed interaction, without requiring changes in any language or infrastructure. The main motivation of this approach is to reuse requirements initially expressed by end-users throughout the whole developent process, thus facilitating requirements traceability and also taking advantage of the productivity of code generation from specifications that Model-Driven methodologies provide.

III. THE MOCKRE APPROACH

In this section we describe the MockRE approach in detail. In the first sub-section we provide general details of the MockRE development process. In the following sub-section, we describe how the approach intends to improve requirements gathering and feature specification by integrating the end-user more intensively in such tasks and using mockups as an understandable language. Finally, in the last sub-section we describe how the generated specifications from end-users can be refined to generate functional working prototypes while developers can easily and quickly extend the generated prototype in a non-intrusive and reusable way either tuning model concepts properties or using direct coding through an external API.

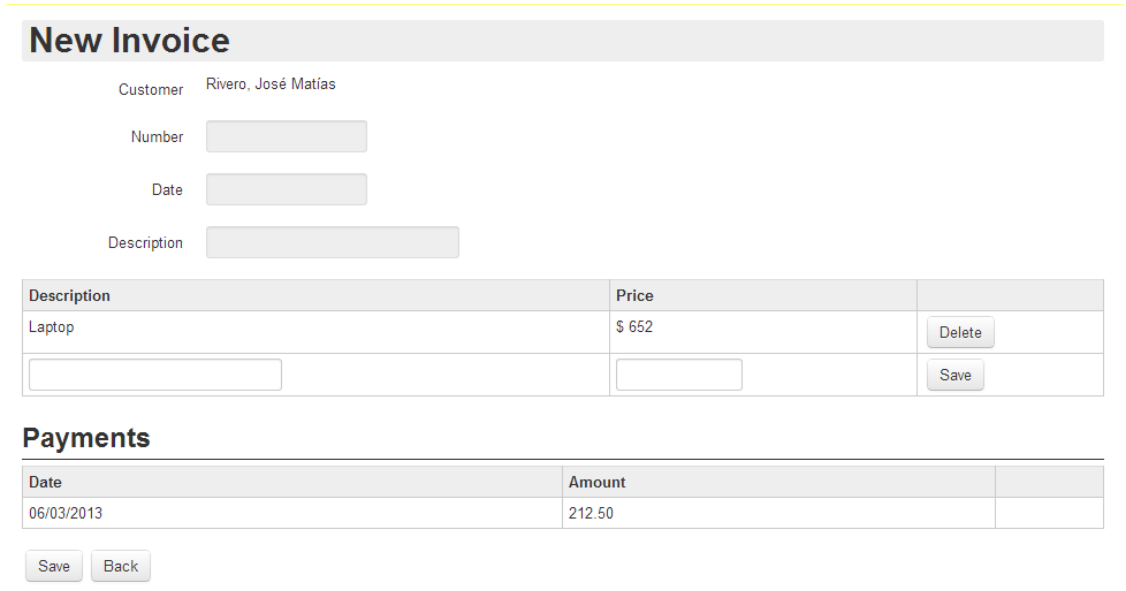
A. Overview of the Process

Since we want to apply agile practices that are meant to quickly generate runnable applications and thus reduce the risks during the development, we have chosen the Scrum [20] agile process as a template since it is one of the most used in industry[5]. The Scrum process starts with the construction of a Product Backlog, which is a list of all the features that the software product must have, prioritized by value delivered to the customer. Then, the product is built iteratively in *Sprints*. Every Sprint starts with a planning meeting to develop a detailed plan for the iteration in which the most important features remaining in the Product Backlog (a subset of the backlog) are broken down into tasks, forming the Sprint Backlog. Once this backlog is carefully defined to take no more than one month of work long, the development team starts to solve all the issues detailed in it. A short Daily Scrum Meeting is done every workday to share the Sprint work progress and new problems found during it. Finally, at the end of a Sprint, a potentially shippable application is demonstrated to the Product Owner and a Product Backlog reprioritization is done if needed, while the goal for the next Sprint is defined.

In our modified MockRE Scrum process (see Fig. 1), every item in the Sprint Backlog has to be related to a set of one or more mockups (built by developers with end-user presence or by end-users themselves) that defines it visually with some level of detail. Once mockups have been built for all the Backlog items, end-users annotate the mockups (assisted by developers if necessary) using an interactive tool that helps them to give semantics to visual elements. Since visual elements present in mockups are perfectly understood by end-users (that are accustomed to interact with Web User Interfaces) they can use the requirements specification tool by themselves. Though not evident for end-users, in this so-called *end-user mode,* the tool assists to specify data, navigation, and business logic requirements (among others) through a click-through interface, interactive menus and very simple wizards to fill extra data when necessary. Finally, end-users can switch the tool to *demo runtime mode*, which allows them to test an interactive version of the mockups (a runnable application) as the tool executes the annotated requirements from the *tagged* mockups.

Developers can refine the requirements specifications generated by end-users in order to better match their desires, using the tool's *developer mode*. Since some features are technically complex for end-users (e.g., RIA behavior, data operations like associations, etc.), they are omitted in the previous step and can be only refined by developers. The *developer mode* provided by the tool allows the incorporation of manual fine-tuning of model properties in order to implement the refined requirements if possible. Then, the application can be run again through the *demo runtime* to show how it should behave in direct presence of end-users.

[5] 7th Annual State of Agile Development Survey, VersionOne – http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf - Accessed: 27-Apr-2013


New Invoice

Customer	Rivero, José Matías
Number	
Date	
Description	

Description	Price	
Laptop	$ 652	Delete
		Save

Payments

Date	Amount	
06/03/2013	212.50	

Fig. 2. An example HTML mockup of an invoicing web application.

TABLE I. SUBSET OF THE TEXTUAL ANNOTATIONS PROVIDED BY THE LANGUAGE.

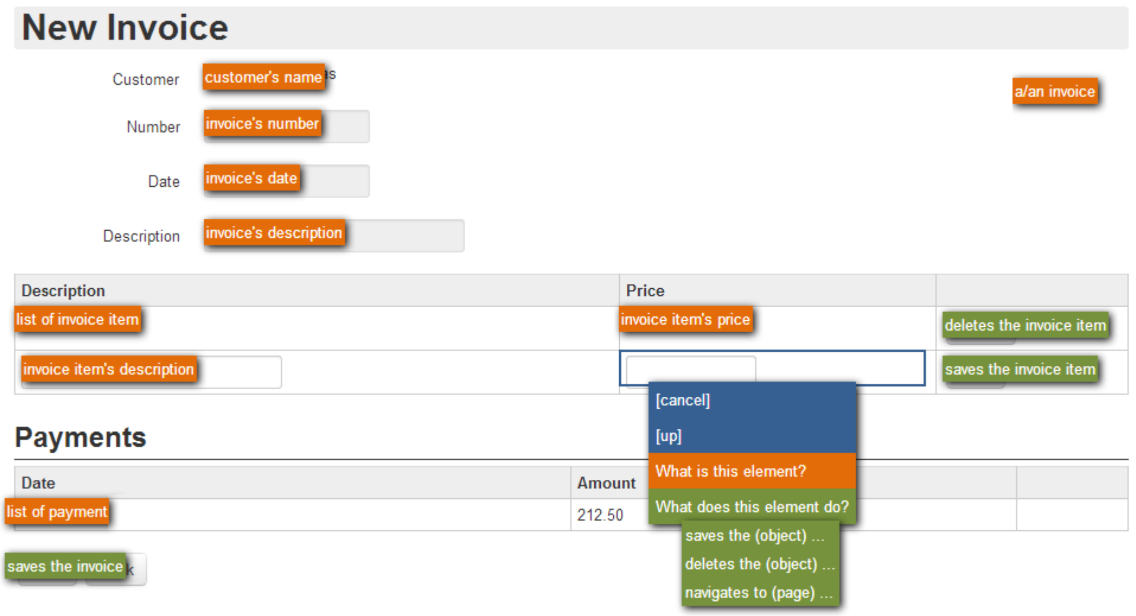
Annotation syntax	**Semantics**	**Can be placed over**
`a/an <item-type>`	The underlying element shows or allows to edit an object of type `<item-type>`.	Composite elements (e.g, non-empty `<div>`s or `<form>`s
`a list of <item-type>`	The underlying element shows or allows to edit a list of objects of type `<item-type>`.	Composite elements (e.g, non-empty `<div>`s or `<form>`s
`<item-type>'s <property>`	The underlying elements shows or allows to edit the property `<property>` of type `<item-type>`.	Simple elements (e.g, `<input>` or `<span>` with text content).
`saves the <item-type>`	The already defined `<item-type>` will be saved when clicking on this element	Action triggering element (e.g. `<button>`, `<a>`, etc.)
`deletes the <item-type>`	The already defined `<item-type>` will be deleted when clicking on this element	Action triggering element (e.g. `<button>`, `<a>`, etc.)
`navigates to <mockup-name>`	A navigation to mockup named `<mockup-name>` will be triggered when clicking on this element	Action triggering element (e.g. `<button>`, `<a>`, etc.)

Finally, since the requirements specifications language is formed by a limited set of constructs and concepts (as any metamodel in the context of a Model-Driven approach), their semantics have limits in some specific cases (e.g., defining custom business logic or special interaction patterns). Then, following the agile code-based spirit to generate working software as fast as possible, developers can refine the specifications with custom code. However, changing the generated code of the generated demo application through the aforementioned tool can lead to inconsistencies between specifications and the code that implements them. In this context, a *code regeneration* can simply erase the improvements that developers wrote by hand, implying big losses in productivity. Then, instead of changing the generated code, developers can just *inject* the desired behavior by calling a specific API on every requirements specification initially done by end-users, following the Strategy Design Pattern [21]. Details of how this is accomplished will be mentioned later.

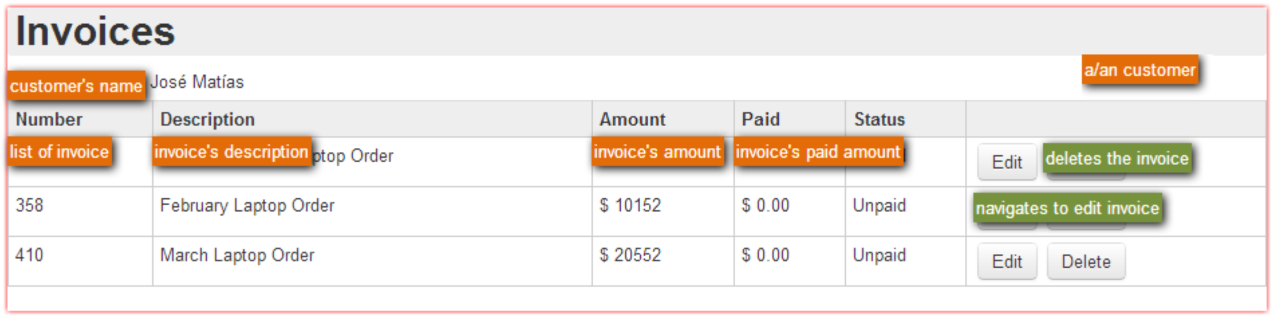
To exemplify the approach, throughout the paper we will use an invoicing Web Application example, since it is a very common and well understood domain and, at the same time, usually requires custom business logic, which is not always easy to model in *pure* MDWE environments. While the Product Backlog of such an application can be very extensive, one potential Sprint Backlog of the development process can be conformed by two basic User Stories:

US1. *As a User, I want to create a new invoice.*

US2. *As a User, I want to list and edit my invoices*.



(a)



(b)

Fig. 3. Screenshots of the tool during an annotation session (*end-user mode*).

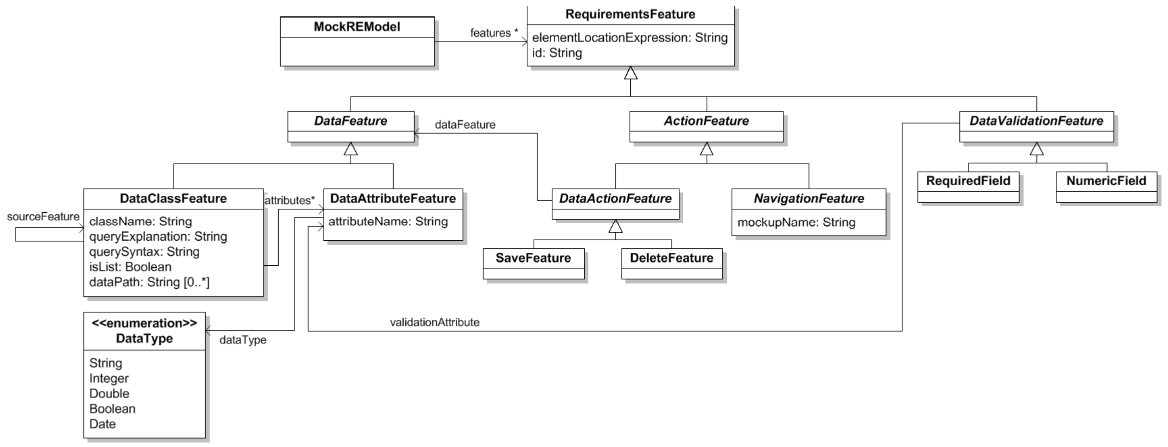


Fig. 4. Excerpt of the MockRE metamodel, showing its data, action and validation features.

Following with the MockRE Scrum process, mockups for both Stories have to be defined. For space reasons, we only show mockups for US1 (see Fig. 2), in which prototypes of the user interface that allows creating a new invoice is depicted. An additional mockup can be defined to list the invoices and, using the one already defined in Fig. 2 to edit existing invoices, mockups for US2 will be enough to implemented the functionality of the Story.

It is important to note that mockups can be translated to HTML format (if necessary) using an already developed *Mockup-to-HTML* tool [7] or by hand. Since a styled HTML presentation of the Web Application has to be constructed soon or later within the web development process, this does not impose an additional cost to the development. On the contrary, if tool-dependent mockups are used (like those that can be built using Pencil or Balsamiq), a first version of the HTML structure can be generated using the tool described in [7]. Once all the HTML mockups have been built and associated to the corresponding Stories, the requirements specification stage can begin.

B. *End-user Requirements Specifications using Mockups*

Since mockups have been built with presence and acceptance of end-users, they can be used by them to specify initial requirements directly - or with developers' assistance if necessary. In any case, an important feature that the approach proposes is to use a textual description language that can be easily understood by end-users and, at the same time, easy to parse.

The language we proposed enriches mockup components (at this stage, HTML elements) with textual annotations, specifying interaction and data requirements. Instead of using a storyboard-based strategy (describing how the end-user interacts with the application), MockRE uses a widget-centric approach describing the role and features of every individual widget. However, an interaction storyboard can be easily inferred by observing the annotations.

The placement of a new annotation using the tool is done through the following steps:

1. One mockup of the Sprint Backlog is shown to the end-user, allowing him or her to highlight the important mockup components - from the interaction or behavioral point of view, i.e., the things that are involved in user interaction and/or can change dynamically depending on the user behavior or underlying data.
2. The end-user clicks on the component identified.
3. A menu is shown, detailing all the potential annotations that can be placed over the mockup component, depending on its type and the annotations already introduced.
4. Depending of the annotation type, one or more extra parameters are required through a special form or wizard presented to the end-user.

Some of the sentences of the MockRE textual language are listed in Table I. It is important to note that a demo of the application using the *demo runtime* feature of the tool can be executed by the end-user at any moment during the requirement specification process. A screenshot of the tool in the middle of an annotation process can be observed in Fig. 3. In that figure, the status of the original HTML mockup of Fig. 2 is shown during an end-user annotation session (Fig. 3.a). Also, an annotated HTML mockup showing a list of invoices (Fig. 3.b) that can potentially fulfill the remaining requirements expressed in US2 is depicted.

In order to avoid technical jargon as much as possible, when then end-user clicks on a relevant component identified (step 2), a menu is presented in which the different annotations are grouped in questions like *What is this element?* or *What does this element do?* (see Fig. 3.a). Also, since end-user may not be familiar with the containment structures present in the HTML mockups, the tool uses heuristics to infer the correct component when the end-user choses an annotation that is not applicable over the selected element. For instance, if the user choses to apply the `a/an <item-type>` annotation over a textbox (an `<input>` HTML tag), then the highlighted element is transferred to the surrounding composite tag (e.g, `<div>` or `<span>`)

C. Introducing Refinements Through Modeling

While annotations seem to be only textual descriptions created with tool assistance by end-users, they are in fact graphical projections of an underlying formal requirements specification model created in the back. The requirement metamodel that we defined for MockRE is depicted in Fig. 4. When working in *end-user mode*, creating a new annotation with the tool implies the creation of their projected model elements with some of their attributes completed by end-users through a wizard. Also, technical values regarding to model elements that are not easily understood by end-user are set to a default or calculated value using heuristics. Some of these values also are converted between its original and *end-user friendly* representation - for instance, converting camel case to spaces and vice versa. After end-user completes the first requirements specification step (creating a first version of the requirements model), developers (using the tool in *developer mode*) can do a *model fine tuning* to match the original requirements in detail - see the screenshot depicted in Fig. 5).

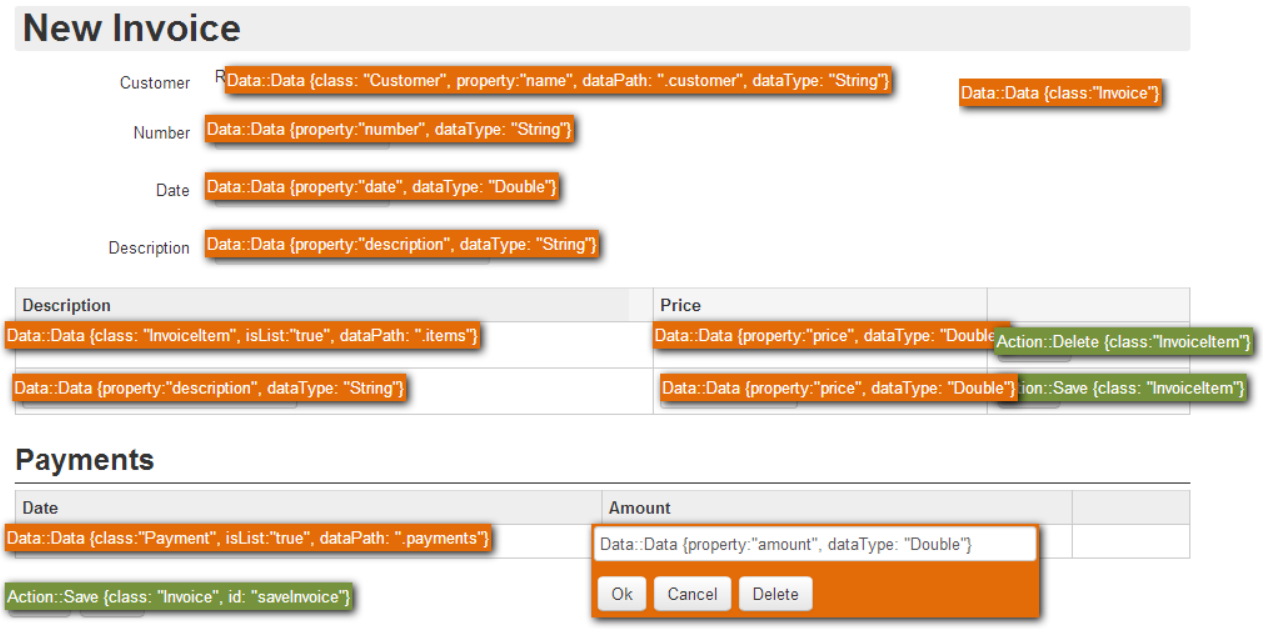
The MockRE metamodel, partially described in Fig. 4, has a MockRE model (`MockREModel`) as the main concept, which contains a set of `RequirementFeature`s. Every `RequirementsFeature` is a requirements specification according to a particular concern or aspect, and has to be mapped to a UI element through a location expression – `elementLocationExpression` attribute. For instance, when modeling Web Applications, XPath[6] expressions can be used for such purpose, while when developing desktop applications a widget id can be used instead. From a different perspective, every `RequirementsFeature` is a developer-related concept that can be projected through a textual annotation, friendly for end-users. For instance, when a user places the `saves the invoice item` (see Fig. 3.a) annotation, in fact is creating a `SaveFeature` instance in the underlying MockRE model and associating it to a `DataClassFeature` with `class = InvoiceItem`. How some of the MockRE model elements are generated from the structured annotations is shown in Table II.

The most important features are `DataFeature`s (`DataClassFeature`s and `DataAttributeFeature`s) that describe data structures and relationships, and `ActionFeature`s that specify actions that have to be taken when interacting with the UI. Among `DataFeature`s, `DataClassFeature`s denote that the associated UI elements over which they are applied are related to a specific type of domain object – parameterized by the `className` property. Thus, the associated element shows or allows editing one or more attributes of this kind of object. From the end-user point of view, this feature corresponds to a recognizable business object manipulated daily and can be easily specified with `a/an <item-type>` and `a list of <item-type>` annotations. On the other hand, `DataAttributeFeature` binds a concrete attribute of the object already mapped. Regarding `ActionFeature`s, they are divided among those that are related to data operations like CRUD (`DataActionFeature`s) and those that are associated to other interaction behaviors like navigation (`NavigationFeature`s). `DataActionFeature`s are related to `DataFeature`s in order to fulfill the required data operations. While more types and subtypes of `RequirementsFeature`s currently exist in the metamodel, we only focused in some of them for space reasons in order to give a big picture of the approach.

As an example, the mockup shown in Fig. 3.a depicts the invoicing application mockup of Fig. 2 in which the `invoice` and `invoice item` objects have been identified and associated by an annotation placed directly by an end-user. After the annotation step, the end-user can run the application by itself and see how it works, which implies that every annotation will add some explicit behavioral semantics to the former static HTML mockup. For instance, the `a/an invoice` annotation and the `invoice's <property>` annotations allows mapping all the data related to a new `Invoice` to the UI components that shows and allows to edit them . The `a list of invoice item` populates the list of `InvoiceItem`s (note the automatic spaces to camel case conversion) to be related in the `Invoice` to be created. A new `InvoiceItem` will be created when clicking in the button annotated with `saves the invoice item`, gathering the required data using the annotated field `invoice item's price`. Also, after clicking in that button, the invoice item list will be updated automatically. Finally, the `a/an invoice` and `saves the invoice` annotations specify that, when clicking on the `Save` button, a new `Invoice` with all the collected data from the UI, including the `InvoiceLine`s and the individual `Invoice` property values like the `invoice's price` will be created. It is important to note that, since the annotations establish a declarative binding between UI elements and the business objects, the same interface can be used both to create a new invoice and to edit an existing one – requiring only an object transfer within a navigation, which is a concept out of the scope of this paper for space reasons.

In terms of the MockRE model underlying the annotated mockup in Fig. 3.a, the `a/an invoice` and `a list of invoice item` annotations are formally represented by two `DataClassFeature` elements, being both related by the `sourceFeature` association - see Fig. 4). However, end-users are not obliged to provide a name to that association, so a generic one is used (see the `a list of` association in Table I). While providing this name may be irrelevant for end-users, for developers this association name is very important from the data modeling point of view, since more than a relationship between the same types of elements can occur.

[6] XML Path Language (XPath) Version 1.0 - http://www.w3.org/TR/xpath/ - Accessed: 27-Apr-2013



New Invoice

Customer Data::Data {class: "Customer", property:"name", dataPath: ".customer", dataType: "String"}

Data::Data {class:"Invoice"}

Number Data::Data {property:"number", dataType: "String"}

Date Data::Data {property:"date", dataType: "Double"}

Description Data::Data {property:"description", dataType: "String"}

Description Price

Data::Data {class: "InvoiceItem", isList:"true", dataPath: ".items"} Data::Data {property:"price", dataType: "Double" Action::Delete {class:"InvoiceItem"}

Data::Data {property:"description", dataType: "String"} Data::Data {property:"price", dataType: "Double"} ion::Save {class: "InvoiceItem"}

Payments

Date Amount

Data::Data {class:"Payment", isList:"true", dataPath: ".payments"} Data::Data {property:"amount", dataType: "Double"}

Action::Save {class: "Invoice", id: "saveInvoice"}

Fig. 5. Screenshot of the tool during an annotation refinement session (*developer mode)*.

```
mockreEngine.getFeature('saveInvoice ').on('before ', function(invoice) {
        if (invoice.lines.length == 0) then {
                alert('The invoice must have at least one line ');
                return false;
        }
        return true;
});
```

Listing 1. Code stub showing how a *feature object* behavior can be enriched through manual coding using its API.

Then, in the second modeling stage, developers can complete the relationship name (adding an association on the `dataPath` property in the `DataClassFeature` with `class = InvoiceItem` using the tool in *developer mode* (see Fig. 5). The same applies for all the data type introduced through the `dataType` attribute in every property annotation, which default to `String` when annotated in *end-user mode*. In this stage, developers can also specify `ids` for the MockRE model elements to access and refine their implementation, as is explained in the following sub-section – see the `id` attribute in `RequirementsFeature` class depicted in Fig. 4.

From the implementation point of view, every annotation type (and metamodel concept) has a corresponding so-called *feature class* in the implementation platform. For every annotation, at demo runtime the parameterized *feature class* is instantiated into a *feature object* and ran. Every *feature object* enriches the UI in order to fulfill their semantics from the implementation point of view (i.e., changing the DOM, attaching events, etc.). Thus, the code generation required to run the application in demo mode consist only in instantiating and running the corresponding *feature objects*.

D. *Refinements Introduction using Code*

After testing the application through the demo runtime provided by the tool, the end-user can discover extra requirements that are not actually modeled or implemented. Some of these requirements are considered by MockRE and thus can be added through new annotations or direct additions to the underlying MockRE model. However, in real applications with minimal business rules and a relatively complex domain model, it is not realistic to assume that all the requirements can be tackled using MockRE concepts. Following with the previous example, the end-user may simply ask (through a new User Story) for checking that the `Invoice` must have at least one line before being effectively created. Since a lot of this kind of business-related validations can be done depending of their specific domain, it is impossible to define a language to cope with all of them.

One of the main reasons to encapsulate the semantics in feature objects (as aforementioned in the previous sub-section) is to preserve the model concepts abstraction in the implementation, so as to avoid generating scattered code to implement annotations behavior. Such way of code generation structure imposes several constraints and limitations when has to be refined: (1) since behavior code is scattered and mixed around the application, it is harder to be interpreted by developers, (2) after having made the refinements, a code re-generation can simply erase them and (3) it can introduce the *Shotgun Surgery* [22] bad smell in the final implementation, since repetitive and scattered code may potentially be generated. To quickly add additional refinements to the generated implementation avoiding these issues, every feature object encapsulates the behavior that it adds to the static mockup and provides a custom API that allow redefining part of it, following an approach similar to the Strategy Design Pattern [21]. For instance, a `SaveFeature` object provides `before` and `after` hooks, which allow executing operations before it tries to persist an object and after it has been persisted. With the proposed approach, in a few lines of code and using JavaScript as a destination platform, a function can be hooked to the `before` event, doing the aforementioned extra validation and canceling the `Invoice` persistence if necessary – see Listing 1.

From the implementation point of view, a version of the framework has to be implemented for every well-known web technology – currently, we have implemented a functional proof of concept version in JavaScript. This framework, in the context of the Model-Driven terminology is part of a *Domain Framework* [23]. Developers can reference the feature objects through the provided `id` (using tool in *developer mode*) to further get them in the code and add the desired hooks – in Listing 1, this is accomplished within the `mockreEngine.getFeature('saveInvoice')` line. If in this or the next iteration the end-user decides to regenerate the demo application to see it running after doing one or more changes in the annotations, since the generated code only consist in instantiations of the corresponding feature objects, the rest of the added behavior is preserved automatically. The code in Listing 1 shows how the *non-empty invoice validation* aforementioned is implemented.

It is important to note that, since we exemplified the approach using the JavaScript language executed in the context of a web browser, all the detailed behavior that can be added is limited to the client-side of the Web Application. This imposes limitations to the kind of operations that can be done when adding behavior using manual coding, forcing in the worst case to create special API methods in the server-side that can be invoked from the client-side to accomplish actions involving server behavior – like, for instance, integrating with other services or APIs. However, when using the MockRE API to refine requirements in the context of a server-side environment (for instance, under JEE), the developer team is free to make use of all the computational power of the Web Application backend to enrich the original requirements with advanced operations like executing distributed transactions, integrating with other APIs as aforementiond, etc.

TABLE II. MOCKRE END-USER ANNOTATIONS AND THEIR METAMODEL REPRESENTATION

End-user annotation	MockRE metamodel representation
`a/an <item-type>`	`DataClassFeature`, with `className = <item-type>` and `isList = false`
`a list of <item-type>`	`DataClassFeature`, with `className = <item-type>` and `isList = true`
`<item-type>'s <property>`	`DataAttributeFeature`, with attributeName = `<property>`, related to a `DataClassFeature` with `className = <item-type>` through the `attributes` association
`saves the <item-type>`	`SaveFeature` associated to the corresponding `DataClassFeature` that specifies the object to be saved
`deletes the <item-type>`	`DeleteFeature` associated to the corresponding `DataClassFeature` that specifies the object to be deleted
`navigates to <mockup-name>`	`NavigationFeature` with `mockupName = <mockup-name>`

IV. Conclusions and Further Work

In this paper we introduced the MockRE process, a requirements gathering and Model-Driven methodology centered on end-users. MockRE uses user interface mockups both to allow end-users to start the requirements and modeling specification from mockups and also to facilitate their interaction with the development team. We exemplified the approach with the development of a core part of a common invoicing application. The mockups used in the example can be easily built by end-users using common tools and can be translated by hand or semi automatically to HTML by developers. We show how, instead of starting fulfilling the concrete requirements expressed in potential User Stories associated to mockups for a given iteration through direct coding, MockRE proposes that end-users annotate the static mockup using a wizard-like tool in which they can express the role of every visible object in the UI. Since end-users are familiar to the visual metaphors present in the interface, they can do great part of the work by themselves and test the modeled application at any point of the specification task, through the *demo runtime* feature provided by the tooling. If the application does not work as expected, developers can refine the annotations, or introduce detailed code behavior without breaking the model abstraction and preserving the inherent productivity of the Model-Driven process and also maintaining the original requirements specifications placed by end-users.

Regarding the future work, we are working on extending the semantics of the annotations and their underlying metamodel representations. As can be observed in the partial metamodel description included in this paper, every different concern included in it (data specifications, data manipulation actions, data validation, etc.) is defined as a class hierarchy in which subclasses may be related to other concepts. Thus, new classes, their end-user friendly representation in the tool, and finally their added semantics (when running a demo) are being implemented. Among the most important, we are considering user interface manipulation, RIA features, and visual API integrations (like social and maps widgets).

Porting the execution framework introduced in this work to different modern Web technologies like JEE and ASP.NET represents another interesting path of work. At the same time, since the use of client-side behavior through JavaScript and the new HTML5 standard is increasing notably, we are making emphasis on polishing the current application running framework to ease support for the development under these technologies. In this context, we are working on providing data connectors to switch from demo-purpose data storage to production ones, improving the performance of the demo runtime (for instance, using HTML templating instead of DOM manipulation) and adding support for most well-known JavaScript libraries and frameworks.

Finally, we are evaluating the approach with several real-world applications in order to assess its feasibility and check their direct advantages in comparison to traditional code-based agile methodologies using mockups.

V. References

[1] S. Ceri, P. Fraternali, and A. Bongio, "Web Modeling Language (WebML): a modeling language for designing Web sites," *Computer Networks*, vol. 33, no. 1–6, pp. 137–157, Jun. 2000.

[2] N. Koch, A. Knapp, G. Zhang, and H. Baumeister, *Uml-Based Web Engineering*. London: Springer, 2008, pp. 157–191.

[3] J. Gómez and C. Cachero, "OO-H Method: extending UML to model web interfaces," in *Information modeling for Internet applications*, P. van Bommel, Ed. Idea Group Inc (IGI), 2003, pp. 144–173.

[4] G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, "Modeling and Implementing Web Applications using OOHDM," in *Web Engineering: Modelling and Implementing Web Applications*, G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Eds. London: Springer London, 2008, pp. 109–155.

[5] P. Vilain, D. Schwabe, and C. S. de Souza, "A diagrammatic tool for representing user interaction in UML," pp. 133–147, Oct. 2000.

[6] M. Cohn, *User stories applied: for agile software development*. Addison-Wesley, 2004, p. 268.

[7] J. M. Rivero, G. Rossi, J. Grigera, E. R. Luna, and A. Navarro, "From interface mockups to web application models," in *12th International Conference on Web Information System Engineering*, 2011, pp. 257–264.

[8] K. S. Mukasa and H. Kaindl, "An Integration of Requirements and User Interface Specifications," in *6th IEEE International Requirements Engineering Conference*, 2008, pp. 327–328.

[9] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999, p. 431.

[10] E. Robles Luna, G. Rossi, and I. Garrigós, "WebSpec: a visual language for specifying interaction and navigation requirements in web applications," *Requirements Engineering*, vol. 16, no. 4, pp. 297–321, Jun. 2011.

[11] U. B. Sangiorgi and S. D. J. Barbosa, "MoLIC Designer: Towards Computational Support to HCI Design with MoLIC," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems - EICS '09*, 2009, p. 303.

[12] N. Koch and S. Kozuruba, "Requirements Models as First Class Entities in Model-Driven Web Engineering," *Current Trends in Web Engineering*, vol. 7703, pp. 158–169, 2012.

[13] J. Ferreira, J. Noble, and R. Biddle, "Agile Development Iterations and UI Design.," in *AGILE 2007 Conference*, 2007, pp. 50–58.

[14] A. Martin, R. Biddle, and J. Noble, "The XP Customer Role in Practice: Three Studies.," in *Agile Development Conference*, 2004, pp. 42–54.

[15] A. Homrighausen, H.-W. Six, and M. Winter, "Round-Trip Prototyping Based on Integrated Functional and User Interface Requirements Specifications," *Requirements Engineering*, vol. 7, no. 1, pp. 34–45, 2002.

[16] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano, "On the effectiveness of screen mockups in requirements engineering," in *2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010.

[17] J. I. Panach, S. España, I. Pederiva, and O. Pastor, "Capturing Interaction Requirements in a Model Transformation Technology Based on MDA.," *J. UCS*, vol. 14, no. 9, pp. 1480–1495, 2008.

[18] R. Ramdoyal, A. Cleve, and J.-L. Hainaut, "Reverse Engineering User Interfaces for Interactive Database Conceptual Analysis," in *22th International Conference in Advanced Information Systems Engineering*, 2010, pp. 332–347.

[19] J. M. Rivero, J. Grigera, G. Rossi, E. R. Luna, and N. Koch, "Towards Agile Model-Driven Web Engineering," *Lecture Notes in Business Information Processing*, vol. 107, pp. 142–155, 2012.

[20] J. Sutherland and K. Schwaber, "The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process." [Online]. Available: http://assets.scrumfoundation.com/downloads/2/scrumpapers.pdf?1285932052. [Accessed: 09-Dec-2012].

[21] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, p. 395.

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999, p. 464.

[23] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society, 2008.