Visually Modelling Data Intensive Web Applications to Assist End-User Development

Vincenzo Deufemia[1], Chris D'Souza[2,3], Athula Ginige[2]

[1] University of Salerno
Via Giovanni Paolo II, 132
84084 Fisciano(SA), Italy
+39 089 963346
deufemia@unisa.it

[2] University of Western Sydney
Locked Bag 1797
Penrith, NSW 2751, Australia
+61 2 9685 9097
a.ginige@uws.edu.au

[3] Australian Catholic University
40 Edward Street
North Sydney, NSW 2060, Australia
+61 2 9739 2553
christopher.d'souza@acu.edu.au

ABSTRACT

Due to problems in correctly understanding user requirements the Information System (IS) development community have recognised the need to involve end-users in the development and maintenance of web applications. End-users perceive web applications through user interfaces (UIs) and commonly use sketches of UIs to express their requirements. Thus, it would be desirable to provide an end-user development methodology centred on UI modelling techniques. In this paper a visual modelling approach is presented to empower end-users in developing data intensive web applications starting from user interface descriptions. The modelling language follows a holistic approach by representing both static and behavioural information of a web application in one visual model. The visual model allows the specification of the look-and-feel of the application through mock-ups, and the user interactions through links, annotations, and widget references. End-users are guided during the modelling process by providing a summary view to manage the design of complex applications and a data model view to improve the quality of the generated applications.

Categories and Subject Descriptors

D.1.7 [**Software**]: Programming Techniques – *Visual Programming*. H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical user interfaces*, *Prototyping*.

General Terms

Algorithms, Design, Human Factors, Languages.

Keywords

End-User Development (EUD), Human Computer Interaction, Visual Languages, Web Application Modelling.

VINCI '13, August 17-18, 2013, Tianjin, China.

http://dx.doi.org/10.1145/2493102.2493105

1. INTRODUCTION

The growing demand for developing web applications quickly, coupled with problems in correctly capturing user requirements have led to the need for involving end-users in the software development and maintenance process [1]. In recent years end-users have demonstrated a great interest in developing the web, especially with social-networking sites, but most of the web pages they build simply present information. On the other hand, the development of data-intensive web applications [2], where the primary goal is to make dynamic data accessible to a variety of users, requires considerable skill in programming and web technologies. This is because of the steep learning curve required to master the tools and methodologies for the design and development of such systems. Some attempts have been made to reduce the burden of the developmental process by providing visual modelling tools. For example WebRatio is a tool for modelling web applications using a visual modelling language called WebML [3]. However WebML developers need to have a thorough knowledge of low-level development details, which is not intuitive to most end-user developers [4].

End-users perceive web applications through user interfaces (UIs) and commonly use sketches of UIs to express their requirements [5]. Thus, they may be empowered by letting them specify the structure and the behaviour of the application through a development methodology centred on the modelling of UIs. Such a methodology should take into account the iterative nature of the web development process, where the design moves from high-level descriptions to increasingly specific details [6], and also the need to manage the frequent changes in the business requirements. Hence the methodology must support the constant refinement of the design ideas resulting in the evolution of the application.

Our research is aimed at addressing such problems by providing end-users with a visual modelling methodology supported by an effective tool to rapidly generate web applications. The modelling method should not only be able to unambiguously capture the complete requirements of users with low technical skills, but must also support the adaptability and reusability concerns of the users.

This paper discusses a visual modelling approach for empowering end-users to develop data intensive web applications starting from user interface descriptions. In particular, the visual model allows the end-user to define accurate representations of web applications through the specification of mock-ups by incorporating graphics and information content, as well as dynamic behaviour. The dynamic behaviour specification is captured through links and annotations on the web pages. Moreover, data dependency notations similar to that used in spreadsheets is exploited for modelling the relationships between data in user interfaces.

In the proposed modelling process, end-user web developers start to explore the design ideas by expressing visual models close to the way they perceive the web application. During the modelling process the end-users are supported by highlighting the information structures of the web pages and the navigational relationships between them thereby providing them with a summary view of the application status. Moreover, they can fix any inconsistencies in the visual model by validating it against an automatically generated ER diagram. A textual specification equivalent of the visual model is used to derive the ER diagram. The textual language targets designers who are not likely to know programming and yet need to refine and improve the specification for customization purpose.

This paper is organized as follows. Section 2 briefly reviews some existing work on web application modelling. Section 3 describes the approach for data intensive web applications modelling together with a motivating scenario. Section 4 discusses the visual modelling language and the support for the design process. Section 5 presents the textual language equivalent of the visual modelling language, and Section 6 highlights the process for automatically deriving the ER diagrams. Finally, summary and concluding remarks are included.

2. RELATED WORK

In [7] Valverde and Pastor provide a specification of web application UI meta-model as a combination of static views and dynamic views. The static view identifies the fundamental UI element types in the web application while the dynamic view identifies the fundamental behavioural changes to the UI due to user interaction. They use the UI meta-model with the OOWS modelling method to engineer the web application [8]. Sections 2.1 provide an overview of Valverde and Pastor's RIA UI meta-model. Section 2.2 discusses current UI modelling languages, while Section 2.3 discusses end-user empowerment through end-user friendly modelling languages.

2.1 UI Meta-Model

In [7] the UI is defined as a composition of widgets. A widget is a visual component of the UI whose main responsibility is to handle data and user interactions. A widget is abstracted as an entity with a set of properties. Five types of widgets are identified based on their interactive functionality:

Data View Widget: A widget to display data.

Input Widget: A widget to input data.

Navigation Widget: A widget to capture the target from which the UI is perceived.

Service Widget: This widget initiates the execution of a service from business logic.

Layout Widget: This widget contains other widgets.

When a user interacts with widgets, events are triggered which cause reactions on either the same widget or on other widgets [7]. These reactions are in the form of:

Property Change: This reaction results in a change of the UI properties of any target widget.

Data Request on Demand: This reaction results in a request for information from the server to a data view widget, if the information is not already available on the client side.

Functional Invocation: This results when a service widget triggers an event resulting in a requests-response communication with the business logic.

Input Validation: This reaction results in a validation of input data and a message if there is a problem with the input data.

Navigation: The navigation reaction results in changing the point from which the application's UI is perceived by the user due to an event triggered from a navigation widget.

In addition, the dynamic view uses event rules to define reactions on target widgets for each event from a source widget.

2.2 UI Modelling Languages

UI modelling languages are generally employed to enable designers to generate UIs from various models such as domain, presentation and task. The generated UIs can then be customized by the designer to expedite the UI development.

For example, Teallach is a User Interface Description Language (UIDL) that enables designers to build a UI from task, domain, and presentation models at logical and physical levels, and it also maps the concepts from one model to another [9]. USIXML is another UIDL that expresses and manipulates UIs at different levels of abstractions [10]. These levels include Task & Concept (T&C), abstract UI (AUI), concrete UI (CUI) and Final UI (FUI) level. The T&C level describes common end-user interaction task objects in a given domain. The AUI level defines interaction space objects by grouping task objects according to requirements but without considering the specificities of concrete layout and navigational elements. The CUI level defines objects from the AUI level with layout and navigation specifications but without considering the platform in which the rendering occurs. The FUI level defines the CUI objects with respect to a specific computing platform.

All UIDLs aim to provide designers a mechanism to bridge the time gap that exists during the user-interface engineering tasks of design, development, and evaluation. To reduce the time gap, UIDLs derive the UIs from other models such as the domain model and the task model. However designing using these models require a good understanding of the concepts of tasks and or domain specifications using the intricacies of the modelling language. Hence they are not intuitive to end-user developers. End-user developers on the other hand may be empowered by exploiting their requirements' expertise through a visual approach reflecting the way they perceive web applications. That is why XML or HTML based visual tools such as Balsamiq [11], Azure or Dreamweaver are popularly used to capture end-user requirements. However, most of these tools do not generate the complete application from the UI specifications. That is, though they are end-user friendly they are not end-user developer friendly.

A more recent proposal for collecting requirements of Web applications and managing their evolution is discussed in [12]. It uses WebSpec, a domain-specific language to capture navigation, interaction, and UI features in Web applications. WebSpec uses UI mock-ups to improve the understanding between different stakeholders and also to reduce the development time. The language is designed to be intuitive and supporting scalability but it uses programming language like syntax while visually expressing the navigation behaviour. Though this is acceptable for IS professionals it fails to exploit the requirement expertise of end-user developers.

2.3 Empowering End-User Developers through User Friendly Modelling Language

Ko *et al.* [1] have observed that there is very little research done on end-user modelling of requirements and specification for interactive and web-based applications. Providing natural language like descriptions of requirements is one approach to empower end-user developers, where in domain level keywords are mixed with end-user defined terms in the language [13][14]. In [15], Liang and Ginige define a Smart Business Object Modelling Language (SBOML) that uses succinct, pseudo-English sentences to model relations among business objects. For example, the SBOML statement "*in organization, employee has first name, last name and might have many office (has room number, building id)*" is user-friendly, as it is easily understood by end-users. SBOML develops a platform specific model of a web application from the specification and also supports the rendering of the UI based on default mappings between data elements and UI elements. Though SBOML is not a end-user UI modelling language, it demonstrates that web applications can be built by empowering end-users to exploit their requirements' expertise. This indicates that an end-user friendly modelling language must be intuitive and hide detailed level specifications.

3. A VISUAL MODELLING PROCESS FOR END-USER DEVELOPMENT OF WEB APPLICATIONS

End-user developers should be supported with appropriate tools to capture and model web application requirements efficiently and effectively. Visual language representations are popularly used among the End-User Development (EUD) community to lower end-user barriers in the specification of their needs and knowledge [16][17]. Lowering the barriers mean users with low IS skills or experience should be able to develop applications with minimum time and effort. Further as end-users gain expertise in the particular domain, a faster way to express the user interface may be helpful. To this end, an alternative text based modelling language equivalent to the visual language represents a valid choice for more expert end-users.

Figure 1 illustrates the proposed web application modelling process. The main idea is to exploit the mock-up of the user interface by enriching it with user interaction information to automatically derive: the data model, the view, and the control logic of the web application. During developmental process the visual tool provides a summary graph to highlight the core informational and navigational structures in the application. The visual tool also auto generates an ER diagram for validating the visual model. The ER diagram is obtained from an auto generated textual UIDL equivalent of the visual model. An end-user acquires a working prototype by simply sketching the mock-up of the user interface, and may use the textual model to define behaviours requiring complex computations. The modelling process can be separated into three phases: the visual/textual specification of the web application, the derivation of the domain knowledge, and the generation of the business logic. In this paper we focus our attention on the first two phases.

The specification process in Figure 1 starts with the end-user developer creating the mock-up of the user interface using a GUI toolkit. The specification includes the graphical layout of the web pages along with their interaction behaviour (see Section 4.1), which is then used to derive other models. The output of the visual specification is an XML document which is used to auto generate the summary view, the textual UIDL code, and a view template. The data model (ER diagram) is derived from the UIDL code. The view template defines concrete details such as the actual position of widgets in a page along with their styling details but without the platform specific details. Hence the view template is the CUI equivalent of the USIXML. On the other hand the UIDL code is a high-level textual specification encoding the AUI equivalent information of the visual specification, which can be refined by expert end-users. Further details of the summary view are provided in Section 4.2, while Section 5 discusses the UIDL constructs and Section 6 explains the derivation of the ER diagram through inference rules. The view templates and the data model can be used to derive the controller (business) logic of the application though its discussion is beyond the scope of this paper. In the following we present a running example that allows us to better explain the proposed modelling process.

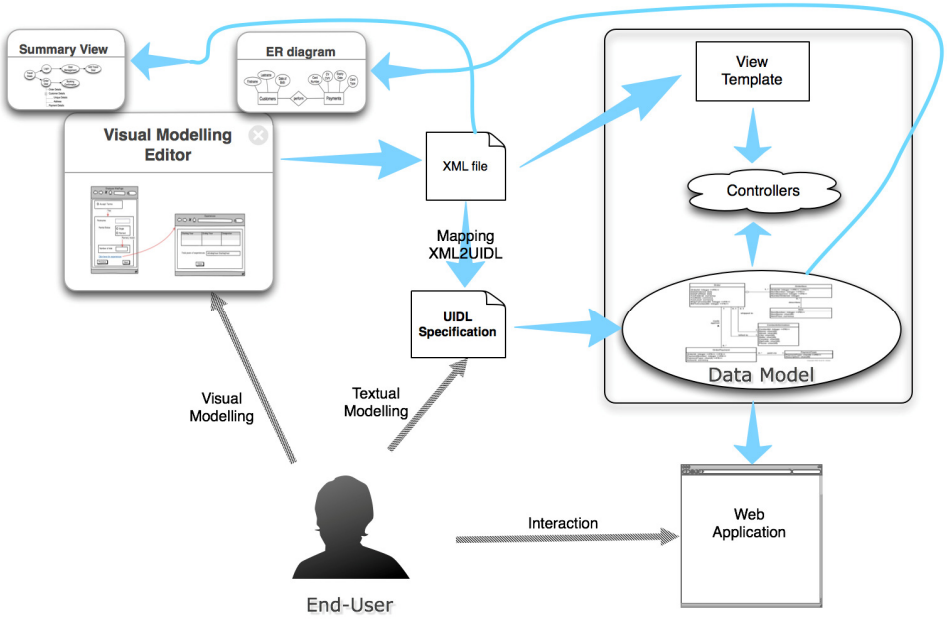

Figure 1. The proposed end-user modelling process.

3.1 Running Example

Figure 2 provides an overview of a web application for managing travel deals. The *Travel Deals* page contains a *Search Deals* container and an *Available Deals* container. Containers are similar to layout widgets identified in Valverde and Pastor's meta-model of the UI [7]. Further details about the need for containers and the various types of containers are available in Section 5. Users may use the search facility on the *Travel Deals* page to scout for specific deals. On clicking a *Booking* button corresponding to a deal, they are led to the *Order Deal* page where they can enter their personal and payment details to make an order. A successful completion of an order will result in the display of the *Booking Confirmation* page. The *Login* link on the Travel Deals page is meant for employees of the enterprise for adding various types of deals by navigating through a *Deal Management* page. On clicking a button in the *Deal Management* page, the *Add Travel Deal* page is reached, from where a new travel deal can be added.

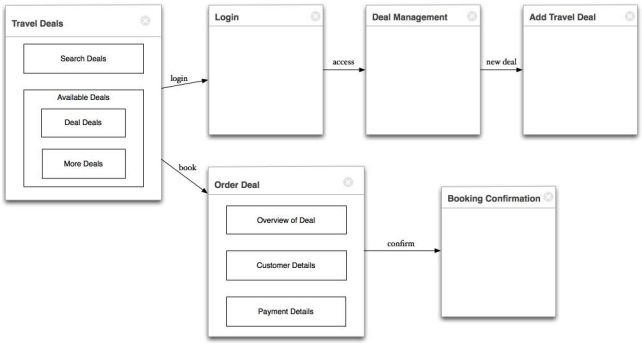

Figure 2. An overview of a web application for managing travel deals.

4. VISUALLY MODELLING WEB APPLICATIONS

Screen mock-ups are commonly used as prototypes for the user interface screens of systems to enable end-users to review and give feedback on the functional requirements of the developing application. A number of tools are available to support the creation of interactive mock-ups of web applications, including Balsamiq [11] and iPlotz. These tools allow end-users to visually define the layout of the web pages and also to capture some interaction requirements. However, it is difficult to capture behavioural information using mock-ups alone, so they are generally used in conjunction with other models such as user-stories [18], use-cases [19], or informal annotations [20].

In the context of end-user development, user-stories, use-cases, and informal annotations are not appropriate to unambiguously capture the requirements. This is because their informal nature inhibits the automatic derivation of other models during the development of the application [21]. On the other hand the visual model can be enhanced by integrating explicit behavioural specification within the model itself. This way the visual model encompasses a holistic approach rather than the amalgamation of two or more models. For example, if a widget requires a validation function, the validation specifications are written using a callout linked to the widget, rather than as a separate model. This results in a visually intuitive model, which reduces the cognitive load on end-user developers. Moreover, the information integrated into the visual model makes it easier to unambiguously extract the domain knowledge and the related business logic.

Figure 3 depicts the visual model of the web application for managing travel deals. The model has been created using Balsamiq, which is a visual tool for the specification of mock-ups and is available as a web application and also as a desktop application. It has been selected because it has a rich set of visual symbols that can be easily extended. Furthermore, it generates XML-based models of the mock-ups that are platform-independent making it a suitable visual UIDL. Figure 3 is discussed in more detail in the next section on the modelling of the behavioural information using screen mock-ups.

Additionally, end-users need to be guided during the modelling process. This can be done by providing updated summary graphs to manage the design of complex applications and also by providing a data model to validate the visual model. This is discussed in Sections 4.2 and 4.3, respectively.

Note that we do not discuss the visual modelling aspects of the structural information because the mock-ups are implicitly structural in nature.

4.1 MODELLING BEHAVIOURAL INFORMATION

As discussed in Section 2, Valverde and Pastor's dynamic model of the user interface captures five essential aspects of behavioural information [7]. The following subsections describe how the behavioural information is incorporated into the visual model.

4.1.1 Navigational Information

The navigation reaction results in changing the point from which the application's UI is perceived by the user due to an event triggered from a navigation widget. This represents a transition between a navigational widget and a page or a container. For example, in Figure 3, clicking on *login* widget on the *Travel Deals* page causes the user to navigate to the *Login* page. This navigational information is visually modelled using a red arrow (displayed as dark grey on black and white prints).

4.1.2 Data Request on Demand

The data request on demand reaction is modelled using an assignment statement '=' in the widget that displays the requested data. For example, the expression '=*Deal Details*' on the *Travel Deals* page causes a data request from the database storing the travel deals. In the visual model this is represented by referencing the data source widget in the expression. In our case the original source of data is the *Deal Details* widget in the *Add Travel Deal* page. These dependencies are captured using an approach similar to cell referencing in spreadsheets. Additionally the = expression notation also supports the evaluation of complex expressions. For example, the *Expiry Time* widget on the *Travel Deals* page is dependent on the expression involving the *Expiry Date* that is set in the *Add Travel Deal* page and the current time.

4.1.3 Functional Invocations

Functional invocations are triggered by service widgets on the occurrence of an event. This will invoke the specified functional action. This is visually captured by a red arrow along with an optional event and an action. The target of the arrow can either be the source widget itself or a different widget. If the arrow connects two different widgets then it also signifies the incorporation of navigational information. On the other hand, a self-pointing arrow results in a functional invocation that populates the widget or changes its properties. For example the self-pointing arrow on the *Country* widget in the *Order Deal* page causes the same widget to be populated with data. On the other

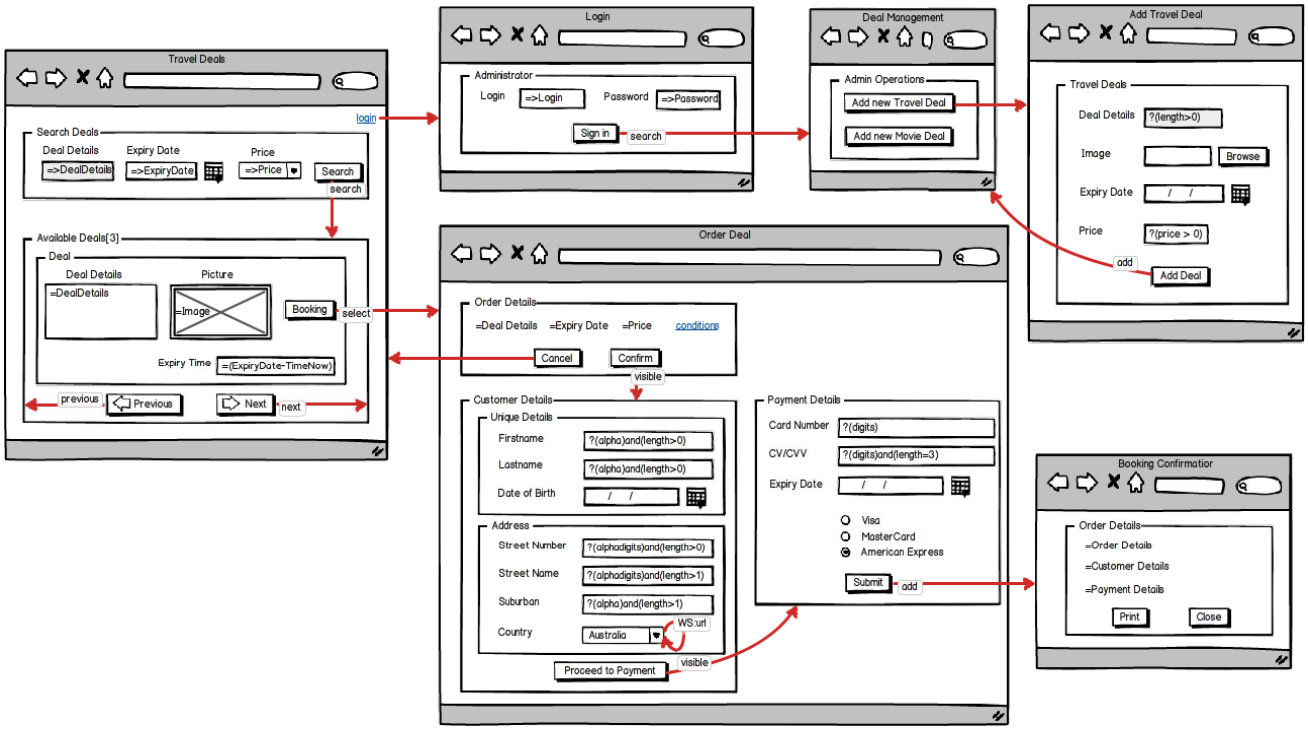

Figure 3. Overview of UI mock-up of a web application for managing travel deals.

hand, the arrow between the *Submit* button in the *Order Deal* page and the *Booking Confirmation* page models functional cum navigational information.

An arrow can have two kinds of actions associated with it: those that refer to local methods and those referring to web service invocations. The latter are identified by a URL specification beginning with the keyword *WS*. For example, the label *url* in the self-pointing arrow on the *Country* widget signifies the invocation of a web service that retrieves the set of countries to populate the widget.

On the other hand, clicking the *Submit* button triggers the invocation of the local *add* method. In general the local method may require one or more arguments denoting the input data and a target data source against which the input data needs to be checked. The '=>identifier' notation is used to indicate the target data source. For example, the search action associated with the *Search* button in the *Travel Deals* page receives three arguments corresponding to the information in the widgets: *Deal Details*, *Expiry Date*, and *Price*. These arguments will match against the target data source indicated by the => notation in the respective widgets.

4.1.4 Input Validation

The input validation is modelled using the symbolic character '?' followed by a validation rule in the appropriate widget. A set of default validation rules can be applied for various data types. For example, the *Card Number* widget has associated the validation rule *?*(*digits*) to constrain users to input only digits. The validation rules can also include simple conditional operators, for example the expression *?*(*length>0*) associated with the *Deal Details* widget in the *Travel Deal* container indicates that the text should have at least one character. In addition, validation rules can be combined with logical operators, for example, *?*((*digits*) *and length=3*)) indicates a rule for a text of three digits. Similarly more complex validation rules can be expressed as pattern expressions.

4.1.5 Property Changes

Property changes occur as a result of a user generated event or an external event, such as the automatic page reload. In the first case, the property changes are modelled as a red arrow between the source widget triggering the changes, and the target undergoing the change. The arrow is also annotated with two attributes. The first, optional, attribute refers to the *event* causing the property change, while the latter defines the *action* to be performed. For example, the annotated arrow from the *Confirm* button on the *Order Deal* page indicates the action of setting the visible attribute of the *Customer Details* container, when the button triggers an event. In the case of external events, the only difference is the absence of source widgets. Note that, as for functional invocations, the arrows used to model property changes can also embed navigational information.

4.2 Supporting the Modelling Process with Web Application Summary Graphs

In general a web application consists of many pages with a high degree of interlinks. Hence for an end-user the design of such an application can be quite complex. Thus, when a page is being designed, the end-user developer must be made aware of the overall context in which it is being designed in relation to the already designed pages. This can be accomplished by providing a summary of the current status of the modelled pages in the web application. The summary should, not only reveal the links between the modelled pages, but also draw attention to the high-level informational structures within each page.

A Web Application Summary (WAS) graph has been proposed to give an overview of the application to end-users. Such a graph is automatically generated from the visual model and is composed of

nodes representing the web pages and arcs representing navigational links. Moreover a collapsible tree structure is used to visually represent the high-level information structures in a node. Such a structure can also highlight potential hierarchies of nested container widgets within a web page.

Figure 4 depicts the automatically generated WAS graph for the visual model shown in Figure 3, in which the collapsible tree for the *Order Deal* node is illustrated.

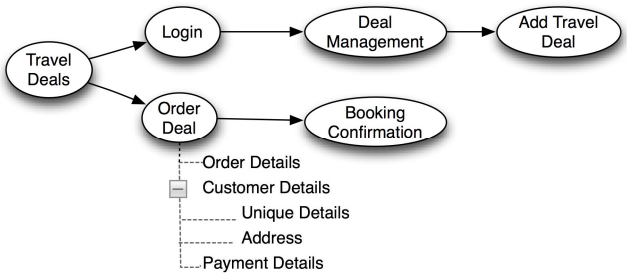

Figure 4. The WAS graph highlighting the navigational and information of the web application modelled in Figure 3.

4.3 Validating the Quality of the Designed Model with ER Diagrams

End-users may be interested in validating the data model generated from the visual model of the UI to identify omissions and/or mistakes. Advanced end-user developers in particular may use the data model to improve the quality of the visual models, where a data model represents the domain information of the web application. Even though end-users do not have the expertise to design a new data model from scratch, they do have the business knowledge to verify the presence of inconsistencies in a given data model. For example, if a data table contains a mix of attributes related to a *Travel Deal* entity as well as a *Customer* entity then the end-user can rectify such inconsistency by fixing the visual model. Obviously, we do not expect end-users to fix all the inconsistencies but the more conspicuous ones can be managed before the generation of the whole application. Generally such inconsistencies in the ER diagrams occur as a result of poor design of the mock-ups.

The data model, which is automatically constructed from the visual model, is expressed as a conventional Entity-Relationship (ER) diagram (Figure 9 depicts the diagram generated from the model in Figure 3). An ER diagram is easy to understand, and is a commonly adopted technique to communicate conceptual models between domain experts and IT experts [22].

The process of mapping the visual model into the ER diagram is driven by rules governing the design of pages as a hierarchical group of conceptually related widgets. More details of these rules are given in the following section, while in Section 6 we present the process for automatically obtaining the ER diagram.

5. TEXTUAL USER INTERFACE DESCRIPTION LANGUAGE (UIDL) CONSTRUCTS

The existing UIDLs are mostly XML based languages which are not end-user developer friendly. A textual UIDL needs to have simple constructs with no html tags yet it must have the ability to define basic UI elements using natural language like syntax. Further the language must have flexibility for extension. Also it should be easy to map the UIDL code to other models such as the physical UI model and the domain model.

The textual UIDL has constructs to describe the UI of an application as a set of pages where a page is made of zero or more containers. A page itself acts like a default container and is delimited by [ ] brackets. A container represents a logical grouping of widgets and is delimited by { } brackets. A widget is either a navigation widget or a data widget. Following Valverde and Pastor's meta-model of the UI [7], the language uses a navigation widget to change the target from which the UI is to be perceived and/or to initiate the execution of a service from the business logic. Similarly it also uses data widget to represent either a data view widget or a data input widget. Each widget may have zero or more properties, which are separated from each other by commas and delimited by ( ) brackets. A property can either be keywords or user defined identifiers or constants. Widgets in a container are separated by commas and containers in a page are also separated by commas. In this paper keywords of the language will be denoted by underlined text (on coloured prints they are also displayed in blue coloured underlined text). Further for economy of space when some code is intentionally omitted, it will be denoted by three dots (. . .).

In addition, a data widget property of referencing data in another widget is captured as a hierarchical path specification in the form: "widget *w* in a container *c* in a page *p*". The usage of containers eases the specification of the organization the UI. Thus each web page is a container which can encapsulate other containers.

The layout widget identified in Valverde and Pastor's meta model is extended to define additional layout widgets in the form of container constructs of the language:

Unique data widget container. This is a container of one or more data widgets whose data is treated as unique. Such a container may be required to uniquely identify a set of similar business entities. A unique data widget container results in compound primary key in a database table. For example while ordering a deal (see Figure 3) a requirement can be to uniquely identify a customer using the first name, last name and the date of birth. In the following segment of the UIDL code, *firstname*, *lastname*, and *date_of_birth* are end-user specified identifiers for widgets that respectively capture the first name, last name and the date of birth of a customer.

```
unique data (
 firstname (string) , lastname (string) ,
 date_of_birth (string ... )
)
```

Dynamic widget container. This container facilitates the organisation of a set of dynamically created widgets for displaying existing data. A widget in a dynamic container can be a data view widget or a navigation widget. A dynamic widget container is commonly used to present an entity's information using various internet media types. The media type can be a combination of rich text, image and audio-video information. For example the *Order Details* container in the *Order Deal* page is a dynamic widget container for presenting order details in textual form. The corresponding UIDL code can be represented as:

```
dynamic order_details_container{
  deal_details (string ...),
  expiry_date(string ...),
  price(string...),
  conditions(link ...)
}
```

The source of data for data widgets in a dynamic container can be a database table or a method call or from a web service. If the data source is from a database table in the same application it can be specified through UI widgets. For example additional information of the *deal_details* widget in the above code may be specified as follows:

```
Deal_details(...default value
  Computationally dependent on
  (page add_travel_deal,
   container travel_deal,
   widget deal_details
  )
)
```

The code segment indicates that the default source in the *deal_details* widget is the *deal_details* widget located in the *add_travel_deal* page (see Figure 3). The keywords "computationally dependent on" indicate that the *deal_details* widget will be affected if the *detail_details* widget in the *add_travel_deal* page is changed. For more details on the various types of dependencies refer to [23]. Alternatively a method call or web service call may be specified as well. This makes the language flexible to manage changes in the application requirements.

For-each widget container. A for-each widget container is a dynamic widget container with additional information for displaying repeated sets of dynamic widgets.

For example in Figure 3, the presence of the digit 3 as an argument of the *Available Deals* container specifies three sets of *Available Deal* containers to be displayed at a time.

```
for-each 3_available_deals_container{
  dynamic deal_container{
   booking (button ...),
   description(string ...),
   picture(image...),
   expiry_time(string ...)
  }
}
```

Notice that the *previous* and the *next* button in the *available_deals* container are not represented in the code though it is shown in the visual model in Figure 3 because they are implicit to the *for-each* container.

Multi-data widget container. This is yet another form of a dynamic widget container for presenting repetitive information generally as rows in a HTML table. Each row of the table represents a repetition of a set of widgets. For example if the list of available deals in Figure 3 is to be presented as a row of deals, then the code would be:

```
multi-data deal_container{
 booking (link ...),
 description(string ...),
 photo(image...),
 expiry_time_countdown_timer(string ...)
}
```

Grouped-widget container. This is a container for a logical grouping of widgets. For example the EUD_UIDL for the address of a customer with respect to Figure 3 can be coded as follows:

```
address{
  street_number (string),
  street_name (string),
  suburb (string),
  country (string)
}
```

As indicated earlier the properties of a widget are represented within ( ) brackets. Some of the other properties are: data type of a widget, whether it is read only or read write, default value if any, whether it is to be hidden or not or whether it has computational dependency on other widgets.

6. DERIVING DATA MODELS FROM UIDL

The constructs of the UIDL for data widgets can be used to derive the data model from the end-user UI description code. This requires identifying whether the data widgets contain data that needs to be persisted or not. A widget yielding data that needs to be persisted is hereafter called *Database Field Yielding Widget* (DFYW). Derivation of data models from the UIDL involves overcoming several challenges:

1) identifying the DFYWs,
2) identifying groupings of such data to associate them with a database table, and
3) finding relationships among groupings to represent the corresponding database table relationships.

Note that navigation widgets are not DFYWs but they contribute towards identification of relationships among database tables. These three points will be discussed in the following sections.

6.1 Identifying a DFYW

A DFYW is a data input widget whose data is required to be persisted in a database. Such widgets may or may not have default data values. Normally input widgets are used to capture data that is required to be stored, although there are few exceptions to this rule.

With respect to the example illustrated in Figure 3, the following cases of data input widgets are identified as potential DFYWs:

1) the *Search Deals* container in the *Travel Deals* page,
2) the *Administrator* container in the *Login* page,
3) the *Customer Details* and *Payment Details* containers in the *Order Deal* page, and
4) the *Travel Deal* container in the *Add Travel Deal* page.

However the data input widgets for the login and the search are meant to capture temporary data with an intention to cross check with already existing data in the database. Hence they are not DFYW candidates. In addition, a data input widget is sometimes used to receive confirmation from the user before allowing interaction with the rest of the UI. These confirmation type data input widgets are also not DFYWs. Similarly data input widgets for update too are not DFYWs.

The above discussion indicates that Valverde and Pastor's model for input widget types need to be extended by four sub-types:

1) cross-checking input widget
2) confirmation input widget
3) updating input widget
4) persistent data input widget

Of the above four input widget types the first three have dependencies with other widgets while the persistent data input widgets will have no dependency. The cross-checking or confirmation or updating type of input widgets has a computational dependency with other widget(s). That is if the other widget is either deleted or changed it will affect these widgets because the data in the other widget(s) is computationally linked to the data in these widgets. Hence the UIDL code for such input widget types will have additional properties to indicate computational dependencies with other widgets. For example the

EUD_UIDL code for the *Deal Details* data input widgets in the *Search Deals* container is as follows:

```
search_deals_container{
 deal_details (..., editable,
    computationally dependent on
    (page add_travel_deal,
     container travel_deal,
     widget deal_details
    ) ...
}
```

In this example the editable and computational dependency property together indicates that it is a cross-checking data input widget type.

Furthermore, as indicated earlier a DFYW can have default values, which may be assigned automatically by the system. For example the *Expiry Date* widget in the *Add Travel Deal* page in Figure 3 may be automatically initialised depending on the date. The default values of such widgets may also be initialised from web services or other method calls or other widgets.

In summary, a DFYW is defined as an input widget that is used to gather persistent data from the user and not as an input widget to gather data for cross-checks or for confirmations or for updates.

Apart from the above cases that identify whether an input widget should be considered as a DFYW, additional rules can be suggested to identify when a data widget should not be considered DFYW. These include:

1) Data view widgets: A data view widget has a non-editable property along with references to their data source. A HTML table is also normally used as data view widget to populate information from pre-existing sources.
2) Data-Nav widgets: Some widgets behave as data view widgets as well as navigational widgets. Hereafter these will be termed data-nav widgets. A data-nav widget is a navigation widget in which the navigation link label is dynamically created from pre-existing data source(s). An example of a data-nav widget list is set of customer links where each link's label is the customer's full name, an already identified data source. Data-nav widgets behave as data-view cum navigation widgets and are ignored because they do not denote new sources of persistent data.

6.2 Identifying groupings of data to be associated with a database table

The second challenge of identifying the grouping of data to be associated with a database table can be solved by specifying related DFYWs in containers. The motivation for using containers is that they can potentially be associated with a corresponding database table. For example the grouped widget container discussed in Section 5, groups all address related widgets in an address container which can be easily associated with an Addresses table. This is intuitive to most end-users and this policy can be further re-iterated through adequate documentation and training. A container with at least one DFYW is hereafter called database field yielding container.

Another advantage of using containers to group widgets is that it allows the names of the widgets to be unique only within a container. This makes it easy for the end-user developer to specify the UI without needing to worry about repetitive names of widgets in the various pages of the UI.

As discussed in Section 5, the notion of containers also enables the end-user developer to define DFYWs whose values are required to be unique together but not individually, as in the case of composite primary keys. Such widgets can be associated with the "unique" container.

It is possible that a container can contain other containers in a hierarchy of groupings. For example a web page may have a container for user registration, which in turn may have a container for user address. Such hierarchies can help in identification of table relationships too. However this will be discussed in the next section.

The notion of containers also makes it easy to define data-nav widgets within containers since such widgets normally appear as list of widgets of the same type. For example a dynamic navigation list of all customers is a group housed in a container. From the discussion in Section 6.1 it follows that such a group is to be ignored while identifying database tables from the UI specification.

The usage of containers makes it easy to formulate rules for the identification of groups of widgets that need to be ignored during the derivation of the data model. Applying the concepts discussed in this section and in Section 6.1, the following observations can be made while identifying database tables from the mock-up in Figure 3:

1) The *Search Deals* container is ignored because all its input widgets are of cross-checking data input widget types.
2) The *Available Deals* container is ignored because it is a dynamic widget container.
3) The *Administrator* container is ignored because all its input widgets are of cross-checking input widget types.
4) The *Order Details* container in the *Order Deal* page is ignored because it is a dynamic container.
5) The *Unique Details* container is considered a composite primary key of a *Customer Details* table.
6) The *Address* container yields an *Addresses* table.
7) The *Payment* container yields a *Payments* table.
8) The *Travel Deal* container yields a *Travel Deals* database table.

6.3 Finding relationships among groups to represent the corresponding database table relationships

Database table relationships can be identified either from transition links (navigation links) between database field yielding containers or implicitly from nested database field yielding containers or from the source of the data view widgets in dynamic containers or from the source of cross-checking data input widgets. However it is possible that a container may not carry any data widget. Hence such a container cannot be treated as a source of a database table. That is, not all containers are sources of database table relationships. Scenarios a) to g) below describe some inference rules to identify these relationships.

By default all relationships will be treated as many-to-many which may be altered by using keywords one-to-many or many-to-one or one-to-one in the inner container.

a) *A dynamic container (or a container with cross-checking data input widgets) sources data from two or more containers*

   If the container sources data from two or more containers, it implicitly indicates a relationship among the source data containers. For example, the *Order Deals* container in the *Booking Confirmation* page sources the *Deal Details* from the *Travel Deals* container in the *Add Travel Deal* page and also

sources the *Payment Details* from the *Payment Details* container in the *Order Deal* page. This indicates that there is a relationship between *Payment Details* and *Travel Details* (see Figure 5).

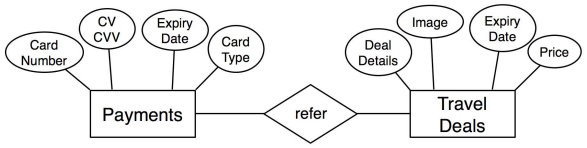

Figure 5. Relationship between Payments and Travel Deals.

b) *A dynamic container (or a container with cross-checking data input widgets) targets a database field yielding container (or vice versa)*

This case indicates a relationship among the data sources of the container widgets and the database field yielding container. In Figure 3, the *Order Details* container targets the *Customer Details* container, which indirectly indicates a relationship between the *Travel Deal* and the *Customer Deal* containers because the source of the widgets in the *Order Detail* container is the *Travel Deal* container (see Figure 6).

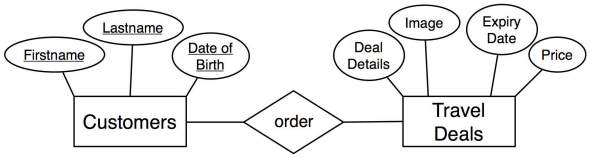

Figure 6. Relationship between Customers and Travel Deals.

c) *A database field yielding container includes one or more database field yielding container*

This case indicates that a relationship exists between the two containers. In Figure 3, the *Customer Deal* container includes an *Address* container and a *Unique Details* container. Hence a relationship can be established between a Customer table and the Addresses table (see Figure 7). The *Unique Details* container identifies the composite primary key of the *Customer* table.

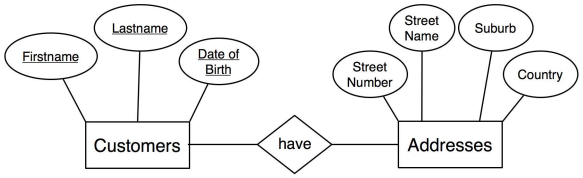

Figure 7. Relationship between Customers and Addresses.

d) A database field yielding container includes one or more non database field yielding container (or vice versa)

In such a case if the inner container(s) has no data view widget or cross-checking data input widget then no relationship is established, otherwise case b) inference rules will apply. Similar rules can also be applied for the inverse case.

e) A non database field yielding container includes one or more non database field yielding container

The two containers are simply ignored if no data view widget or cross-checking data input widget exists in them because they are not database field yielding containers. Otherwise case b) inference rules will apply.

f) A database field yielding container targets another database field yielding container

If the target of a navigation widget in a database field yielding container is itself a database field yielding container then a relationship can be established from the container encapsulating the navigation widget. Hence, a relationship is found between the *Customer Deals* table and *Payments* table (see Figure 8).

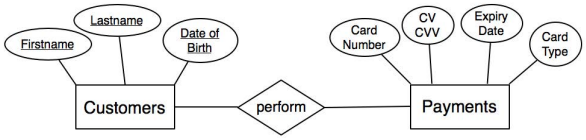

Figure 8. Relationship between Customers and Payments.

g) A database field yielding container targets a non database field yielding container (or vice versa)

If the non database field yielding container has data view widgets or cross-checking data input widgets then a relationship is established among the database field yielding table and the corresponding source tables of the data view widgets or cross-checking data input widgets. Otherwise the search is recursively carried in the nested containers (if any) until a relationship can be established. Similar rules will also apply for the inverse case.

In summary, the various relationships identified using the above mentioned inference rules yield the data model in Figure 9.

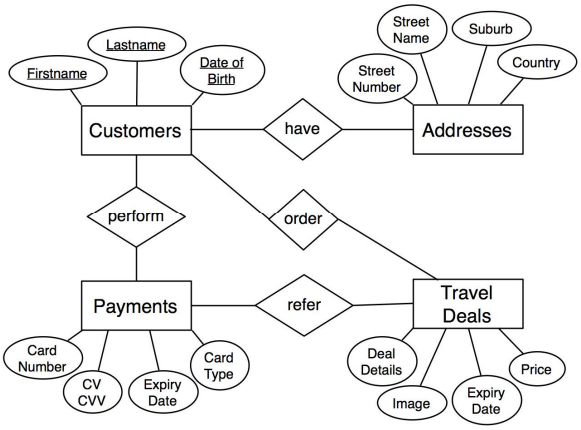

Figure 9. The final data model.

7. CONCLUSIONS AND FUTURE WORK

This paper introduces a visual modelling approach for empowering end-users to develop data intensive web applications starting from user interface descriptions. The modelling language follows a holistic approach by representing both static and behavioural information of a web application in one visual model. End-users are guided during the modelling process by providing a summary view to manage the design of complex applications and a data model view to improve the quality of the generated applications. The proposed approach also supports advanced end-user developers through an automatically created textual model equivalent of the visual model without using HTML or technological terms. A running example illustrates how a data model can also be derived from the textual model. The modelling approach is grounded in existing UI models. The modelling notations and the inference rules specified in this paper have been

implemented and the working prototype successfully produces SBOML statements representing the data models using the visual model as an input.

In the future we plan to improve the visual model with error feedback, user suggestion, and auto-completion mechanisms to guide the end-user during the development process by using visual language compilers [24]. We also intend to improve the layout of the generated pages by inferring the relationships between widgets through the analysis of user interactions [25].

8. REFERENCES

[1] 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., and Wiedenbeck, S. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages.

[2] Ceri, S., Fraternali, P., Paraboschi, S. 1999. Design principles for data-intensive Web sites. *SIGMOD Rec.* 28, 1, 84-89.

[3] Ceri, S., Daniel, F., Matera, M., and Facca, F.M. 2007. Model-driven development of context-aware web applications. *ACM Trans. Internet Technol.* 7, 1, Article 2.

[4] Fogli, D. and Parasiliti Provenza, L. 2011. End-user development of e-government services through meta-modeling. In *Proc. of 3rd International Symposium End-User Development*. IS-EUD 2011. LNCS, 6654, Springer-Verlag, Berlin, Heidelberg, 107-122.

[5] Escalona, M. J. and Koch, N. 2004. Requirements engineering for web applications – A comparative study, *Journal of Web Engineering* 2, 3, 193-212.

[6] Newman, M.W., and Landay, J. A. 2000. Sitemaps, storyboards, and specifications: a sketch of Web site design practice. In *Proc. of the 3rd Conference on Designing Interactive Systems*. ACM, New York, NY, USA, 263-274.

[7] Valverde, F. and Pastor, O. 2009. Facing the technological challenges of web 2.0: A RIA model-driven engineering approach. In *Proc. of the 10th International Conference on Web Information Systems Engineering*. WISE '09. Springer-Verlag, Berlin, Heidelberg, 131-144.

[8] Valderas P., Pelechano V., and Pastor, O. 2007. A transformational approach to produce Web applications prototypes from a Web requirements model. *International Journal of Web Engineering and Technology* 3, 1, 4–42.

[9] Griffiths, T., Barclay, P.J., Paton, N.W., Mc Kirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C.A. and da Silva, P.P. 2001. Teallach: a model-based user interface development environment for object databases. *Interacting with Computers*, 14, 31-68.

[10] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. and López-Jaquero, V. 2005. USIXML: A language supporting multi-path development of user interfaces engineering human computer interaction and interactive systems. In *Engineering Human Computer Interaction and Interactive Systems*. Springer-Verlag, Berlin, 200-220.

[11] Balsamiq, http://www.balsamiq.com/, last accessed June 15th, 2013.

[12] Luna, E. R., Rossi, G., and Garrigós, I. 2011. WebSpec: a visual language for specifying interaction and navigation requirements in web applications. *Requirements Engineering* 16, 4, 297-321.

[13] Little, G. and Miller, R.C. 2006. Translating keyword commands into executable code. In *Proceedings of ACM Symposium on User Interface Software and Technology*. UIST'06. ACM, New York, NY, USA, 135-144.

[14] Liu, H. and Lieberman, H. 2005. Programmatic semantics for natural language interfaces. In *Proceedings of ACM Conference on Human Factors in Computing*. CHI'05. ACM, New York, NY, USA, 1597-1600.

[15] Liang, X.D. and Ginige, A. 2007. Enabling an end-user driven approach for managing evolving user interfaces in business web applications: a web application architecture using smart business object. In *Proc. of International Conference on Software and Data Tech*. SciTePress, 70-78.

[16] Neumann, C., Metoyer, R. A., Burnett, M. 2009. End-user strategy programming. *J. Vis. Lang. Comput.* 20, 1, 16-29.

[17] Pérez, F., Valderas, P., and Fons, J. 2011. Towards the involvement of end-users within model-driven development. In *Proc. of the Third international conference on End-user development*. IS-EUD'11. LNCS, 6654, Springer-Verlag, Berlin, Heidelberg, 258-263.

[18] Cohn, M. 2004. *User Stories Applied: for Agile Software Development*. AddisonWesley.

[19] Homrighausen, A., Six, H., and Winter, M. 2002. Round-trip prototyping based on integrated functional and user interface requirements specifications. *Requir. Engineering* 7, 1, 34-45.

[20] Moore, J. M. 2003. Communicating requirements using end-user GUI constructions with argumentation. In *Proc. of International Conference on Automated Software Engineering*. ASE'03. IEEE CS Press, 360-363.

[21] Giese, M. and Heldal, R. 2004. From informal to formal specifications in UML. In *Proc. of the 7th International Conference Unified Modelling Language*. UML'04. LNCS, 3273, Springer-Verlag, Berlin, Heidelberg, 197-211.

[22] Casati, F., Daniel, F., De Angeli, A., Imran, M. Soi, S., Wilkinson, C. R., Marchese, M. 2012. Developing mashup tools for end-users: on the importance of the application domain. *International Journal of Next-Generation Computing* 3, 2.

[23] D'Souza, C., Ginige, A., Liang, X. 2012. End-user friendly UI modelling language for creation and supporting evolution of RIA. In *Proc. of the 7th International Conference on Software Paradigm Trends*. SciTePress, 190-198.

[24] Costagliola, G., Deufemia, V., Risi, M. 2005. Sketch grammars: A formalism for describing and recognizing diagrammatic sketch languages. In *Proc. of 8th International Conference on Document Analysis and Recognition*. ICDAR'05. IEEE CS Press, 1226-1230.

[25] Deufemia, V., Giordano, M., Polese, G., Simonetti, L. M. 2013. Exploiting interaction features in user intent understanding. In *Proc. of the 15th International Asia-Pacific Web Conference*. APWeb'13. LNCS, 7808, Springer-Verlag, Berlin, Heidelberg, 506-517.