Considering User Interfaces to REA Systems

Jacob Winther Jespersen
IT University of Copenhagen
Glentevej 67, DK-2400 NV, Denmark
jwj@itu.dk

ABSTRACT
Every system development team forms ties to their system's prospective users by way of the terminology used and concepts represented in the system's user interface. To this end, studies of the system application domain naturally inform the development process and with that the user interface design. However, when the study process purposefully abstracts beyond the original domain, its relevance as background to the design of user interfaces becomes unclear.

This paper argues that concrete information systems derived from *meta-level* domain considerations – such as REA [1] – pose a challenge with regards to development of good user interfaces that is not taken up by existing techniques. We provide an analysis of REA as subject for user interface design, and we outline a work-in-progress approach to user interface development in a REA context that addresses some of the challenges.

INTRODUCTION
REA [1] looks at business activity as a continuum of *economic exchanges*. Its primary objective is to allow application of computation to a composition of economic exchanges, e.g. as found in corporate enterprises and business supply chains. To the extent that it is possible to model business activity as consisting of Resources, Events and Agents, one can use the REA axioms to compute the soundness of a given composition of exchanges. Furthermore, a working REA system provides both the key accounting figures known from traditional double-entry bookkeeping as well as dynamic overviews of the state of business beyond the bottom-line, and a foundation for validating business activity as it unfolds.

Assuming that REA-based systems are able to keep the promise of the theory behind them, the question becomes how to facilitate adaptation of the generic REA model to individual businesses. This paper will start by outlining a likely system development scenario for a REA-based system, and relate it to a relevant and currently successful business model. We then focus on the task of building a user interface to such a system in the scenario described. This starts by discussing two distinct approaches, and ends with an outline of our current line of research that draws from both approaches.

ASPECTS OF REA
Clearly, REA is formulated in response to the needs of people in charge of organizing and running businesses even if the rationale behind REA is not necessarily accessible to them. For this reason, a REA-based system will have to look as a 'normal' accounting system in the sense that it must appear to handle accounts, invoices, payments, ledgers, etc. and only internally represent these elements by appropriate REA constructs. With the same argument, one would expect a REA-based system to provide the pledged business overviews aligned with usual terminology or well-known abstractions, and not in REA terms.

Since successfully modeling a given business in terms of Resources, Events and Agents requires fluency with REA principles, it seems unlikely that businesses initially will do that on their own. Instead, REA may do well in the *Vendor-Partner-Customer* business model that is currently successful in commercial information systems development. In this value chain, a Vendor relies on Partners to adapt his domain specific software to fit Customers; Partners are often third parties but occasionally part of the Vendor or Customer organization.

REA Technology
The key to appreciate the Vendor-Partner-Customer setup is the specificity of the domains that the systems address. Usually a Vendor targets a single domain, e.g. accounting or financing, thus he produces domain-specific tools and components. For example, if a Vendor seeks to facilitate his Partners to build individual accounting systems, he concentrates on providing a solid and competitive accounting core functionality, i.e. the functionality needed in every accounting system. He need not consider exactly what differentiates a car-dealer from other kinds of dealerships in the accounting sense. Knowing about such differences is left to Partners who are supposed to have specialized knowledge of the business domains they work in.

Since REA by nature formulates principles for a 'core accounting functionality,' a Vendor in this domain could choose to base his technology on REA, and rely on his Partners – whom he could teach to 'think' in terms of REA – to work with individual customers to model their businesses in a way that makes REA technology applicable.

USER INTERFACE DEVELOPMENT

We will now turn to consider the conditions for development of user interfaces to an imaginary REA-based system. We place the discussion in the context of the Vendor-Partner-Customer business model. Even if this value chain is not required to bring about REA-based systems, we believe the 3 parties represent relevant players in the realization.

In the following discussion, we will assume that a Vendor is in fact able to realize a core technology based on REA, and that Partners have received the training required to effectively model real businesses in terms of Resources, Events and Agents. We also constrain the discussion to user interface issues even if many requirements – e.g. flexibility with regards to business changes – naturally apply to other aspects of the system.

This section presents a possible architecture for development of user interfaces to REA-based systems, and introduces two relevant perspectives on user interface design in this context.

Success criteria

We believe there are 2 critical success factors to user interface development in the given situation:

1. The ease with which the Partner is able to utilize the generic REA constructs to accommodate individual Customers' businesses, and the flexibility of such utilization in dealing with changes.

2. The degree to which REA-based systems honor traditional requirements to usability, e.g. measured in terms of ease of learning, efficiency, understandability, or other factors.

Based on this assumption, we have looked at ways to successfully develop a REA-based system's graphical user interface (GUI). For the current line of research, we have been interested in equipping the Partner with the means to specify and maintain descriptions of user interfaces that are *complete* (in the sense that they contain enough information to be executable), *essential* (in the sense that unnecessary details – e.g. technical ones – are abstracted away) and *expressive* (in the sense that the necessary presentation and interaction styles are possible). Furthermore, specifications should be easy to overview, and simplicity in its concepts be given high priority.

Relative to the success factors mentioned, we find the requirement to cope with changes in the way business is carried out to be the most challenging, thus this is a central theme in the research. And, in line with the argumentation earlier, we expect the Partner to assume the responsibility for managing changing requirements at his Customers as long as these requirements fall within the scope of available REA technology.

Specifying user interfaces

Intuitively, a user interface specification bridges the gap between the system's data abstraction – maybe objects in an object oriented implementation – and the user interface abstraction, e.g. *widgets* such a windows, buttons, menus, etc. in a GUI.

In relation to the Vendor-Partner-Customer scenario, a possible depiction of the use of such specification – that may be understood as a user interface (UI) *model* – is shown in Figure 1.
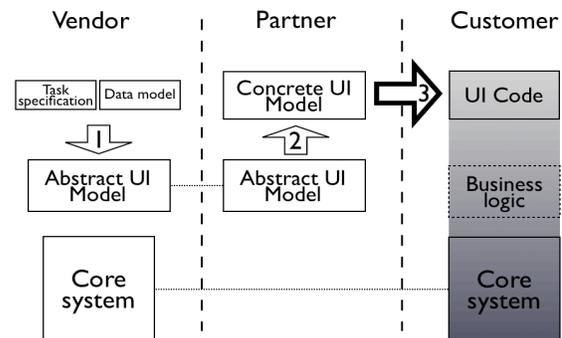

Figure 1. Depiction of the Vendor-Partner-Customer relationship with emphasis on the user interface realization. The gray area marks the final system.

Ideally, the Vendor (with the core system in place) is able to analyze the domain he is working in – possibly expressed in a task specification and a data model, or in alternative ways – and then (1) design an appropriate *abstract user interface model*. This model explicitly represents the points of variability available to Partners, and exists in conjunction with (3) some means of executing the *concrete user interface model* that a Partner produces (2) to Customers on an individual basis.

This basic architecture and the notion of abstract versus concrete user interface models is known from research in *model-based* user interface development [2, 7], and we adopt it for our purposes as it is able to support the Partner role as we see it.

Two perspectives on user interface design

With this outline of the architecture in place, we then consider the relationship between the model and what is being modeled. There are at least two different views on what is being represented in a concrete user interface.

Clearly, at system runtime, system objects appear as – and inside – the user interface widgets in order to allow user interaction. While this could lead to the belief that building a good user interface is a matter of visualizing system objects one by one, e.g. as it is done in [3], we found that to be inadequate in this case.

In preparation for the arguments behind this observation, and the presentation of our approach, we will present two distinct and useful perspectives on what designing a user

interface is about. The central question that divides them is the understanding of whether the designer designs an interface to a system, or augments a domain with computation?

As a trivial framing of the question, consider the data a given system stores about a Customer. That data represents a carefully selected set of information (name, address, etc.) about a real customer, which is judged to be sufficient for the purpose of that system. Now, within the system the customer's data takes on a 'life of its own' in the form of a *data object*. As an object, it is moved, duplicated, referenced and maybe eventually deleted; symbolically, it remains a Customer.

System as subject

One may consider the task of user interface design as being about making proper visualizations of the *objects in a system* and giving the user access to the functions (methods, if you will) that exist in a given context of objects. In this view, which we will refer to as the *system-as-subject* perspective, the relation between an object and the 'real thing' it represents (e.g. a Customer) may almost vanish in light of the object as phenomenon. By itself, a system object can be a complex matter; it may have fields, methods, relations, etc. and it is hardly what one expects to encounter at face value in a user interface. We may of course choose to let objects contain a proper representation of their internals (pretty printing). Either way, the system objects are *what are really there* (i.e. directly available for computation) even if the user interface design may try to mimic the application domain and present system objects in a way that suggests a connection to real phenomena.

Domain as subject

In a different perspective, user interface design is about making proper representations of the *application domain* and the activities that take place there, thus we will call it the *domain-as-subject* perspective. Here, the user interface frames users' actions in the domain and augments the domain with relevant computation. A notion of system objects as phenomena is entirely absent; the designer is primarily concerned with representing phenomena in the application domain in a way that supports users' need to overview and manipulate the domain. The representation need not be *true* in any sense as long as it enables users to act in a sensible way, also when something unexpected happens. In this perspective, system objects exist merely as coincidental data containers of which some amount is required to realize a window on the screen.

For a number of reasons, neither of the perspectives alone sustains development of a user interface. That the two perspectives can and do necessarily co-exist is exemplified in the way many existing systems require users to sometimes point-and-click to manipulate system objects as phenomena (e.g. click Save to synchronize with persistent storage) and sometimes point-and-click to signify a change

in the phenomena that the object represents (e.g. editing the address field when a Customer relocates). Thus, the user interface designer usually has to maintain a dual view of the system objects, and offer the user means of manipulation in both views.

OUR APPROACH

For the development of user interfaces to a REA-based system, we may try to orient ourselves relative to the two general perspectives laid out. This section ends with an outline of the concrete user interface specification scheme we are currently investigating.

For pragmatic reasons, we recognize that user interface design cannot drive development of a REA technology; Partners must generally *make do* with a Vendor's chosen balance between requirements to performance, scalability, security, etc. Similarly, user interface development in this case must align with the system composition and not vice versa.

This line of thought quickly ends up being an argument for the system-as-subject perspective; in light of a wide range of de facto system characteristics it may not be feasible to let the domain-as-subject perspective dominate. Conversely, there are several straightforward technical options for realizing user interfaces in the system-as-subject perspective, e.g. [4]. In the most radical of these approaches, every system object is entirely self-contained with regards to means of visualization and interaction with users [3].

Still, the validity and value of a REA-based system is obviously very dependent on the domain. Such a system supports organizing and running businesses, and it operates directly with concepts in that domain, so surely we must assume the domain-as-subject perspective when designing its user interface. We also note that most (if not all) user interface design methodologies, e.g. the *virtual windows* method [5], are firmly grounded in this perspective.

However, REA is not bound to the domain by its central concept directly; it is bound only indirectly through a specialized interpretation of the domain. In itself, the REA abstraction is meaningless to the end-users of a REA-based system; at its best, the end-users will just note an extraordinary analytic power and possibly a small sticker that says 'REA inside.' So, even if REA as concept permeates REA technology, it has no direct role to play in the user interface design.

A combined view

As usual, one needs to strike a balance between the two different perspectives when developing user interfaces. For a REA-based system, it is necessary to align the user interface development with a preconceived REA technology, and at the same time insist that real domain considerations – *without* the REA abstraction – drive the design.

The system-as-subject perspective makes it relatively easy to accommodate changes in business. New or revised system objects can initially be treated generically and gradually acquire presentation and/or interaction characteristics by specification. In the REA context, the generic handling of an unrecognized data type may vary dependent on its meta-type: Resource, Event, or Agent.

However, radical approaches in this perspective will generally fail to honor basic usability requirements. System objects cannot easily on their own account for a multiple, diversified, and context dependent behavior. One may encode such behavior on a per object basis and achieve a properly coordinated interplay of objects in the user interface, but it will result in a specification that is difficult to overview and maintain.

Dynamic inheritance

Based on the results from our research so far, we are pursuing a style of declarative specification that builds on the concept of *dynamic inheritance* [6]. This subsection will outline the rationale, and provide a very simple example.

The intuition behind using dynamic inheritance in this situation lies in a basic phenomenological observation of user interfaces to information systems. For brief moments in time, system objects actually *turns into* widgets and behaves accordingly. For example, when allowing a user to edit a Customer's address information, logically one may facilitate this by making the 'address' data object *dynamically inherit* from a *Window* class. As a Window, the data object will be visible on the screen, but not readily editable. To this end, we must also make the fields enclosed in 'address' dynamically inherit a widget that allows the user to interact with a text, e.g. *TextEditField*.

A simple example

Figure 2 shows a very simple example of a specification of how the Customer data entity dynamically inherits widgets that makes it editable for the user. Lines 1-10 defines two data types (Customer and Address) in a usual object oriented way. You may note that Customer inherits from an imaginary REA construct, but that is not a central point in the example.

Lines 11-35 hold the dynamic inheritance declarations. It starts with two different declarations of CustomerDetail. The first (line 11) is unconditioned and 'transform' the Customer type to a Window widget, and the declaration body (lines 12-17) transforms the Customer fields to TextEditField widgets. The second CustomerDetail declaration (line 19) has a predicate (Country = "US") discriminating the instances of CustomerDetail that inherit an extension to the CustomerDetail body. The extension body (lines 21-26) overrides the treatment of the 'address' part according to US customary, i.e. it handles the 'state' field, but leaves 'name' and 'credit' untouched.

```
1:    class Customer extends REA.Agent {
2:        String name;
3:        Address address;
4:        Number credit;
5:    }
6:    class Address {
7:        String streetName;
8:        String state;
9:        String zipCode;
10:   }
11:   do CustomerDetail as Window {
12:       do name as TextEditField
13:       do address as {
14:           do streetName as TextEditField
15:           do zipCode as TextEditField
16:       }
17:       do credit as NumberEditField
18:   }
19:   do CustomerDetail where (Country = "US")
20:   as CustomerDetail {
21:       do address as {
22:           do streetName as TextEditField
23:           do state as TextEditField
24:           do zipCode as NumberEditField
25:               with (5 as MaxFieldLen)
26:       }
27:   }
28:   do REA.Agent as Window {
29:       do Object as GenericEditField
30:   }
31:   do Customer* as List {
32:       do Customer as List.Clickable
33:       do List.Clickable where (clicked = true)
34:           as CustomerDetail
35:   }
```

Figure 2. Example of user interface specification based on the notion of dynamic inheritance. Keywords in italic.

The declaration in lines 28-30 states that instances of REA.Agent inherits from Window, and its body makes every field inherit a GenericEditField widget. Declarations concerning meta-types (REA.Agent in this case) are by nature fallback strategies and only effectuated if not a more specialized declaration governing a particular REA.Agent exists. This is an illustration of how a Partner in a sensible way can handle the existence of unanticipated system objects based on knowledge of the type system.

In the example, there is additionally a declaration covering the Customer type directly (lines 31-35). This declaration shows inheritance at an aggregation level, which is signified by the asterisk following the type name. This facility allows the user interface designer control of the appearance of collections of instances, e.g. in order to form lists, matrices, etc. Line 32 makes Customer inherit List.Clickable which in turn, i.e. when the user clicks at one of the Customer instances in List, fulfills the predicate (clicked = true) of the declaration in line 33.

There are numerous details concerning the semantics in this style of specification that are left out in this short presentation, partly due to concerns for space, partly as it is work-in-progress.

CONCLUDING REMARKS

We have designed and implemented a prototype of a runtime system that executes user interface models based on the style of specification presented in this paper.

Our preliminary results indicate that this kind of user interface specification allows the user interface designer to control presentation and interaction characteristics of system objects dependent on context, which is vital to even begin a design that is expected to support users carrying out real work, and fulfill basic usability requirements.

In the portrayed Vendor-Partner-Customer value chain, we imagine the Partner plays the role of user interface designer as part of his task to accommodate a Vendor's core REA technology to fit a Customer. We expect Partners to gradually standardize on designs that fit the business market they address, and buy or make user interface widgets specialized for a domain's established style of visual appearance and task support.

The tests carried out with the prototype so far demonstrate that the notion of dynamic inheritance is suitable to encapsulate a basic variability with regards to composition of widgets, e.g. as when clicking a list shows details for the selected item. We have not yet adequately modeled complex widget compositions.

In terms of the desired characteristics of the specification – completeness, essentiality, and expressiveness – it is too early to judge. However, we note that a recurring and often troublesome theme in user interface development, event handling, is not burdening the specification so far, which is a plus for its essentiality, and simplicity in concepts.

Future work

There are several aspects of the user interface modeling presented here that we need to develop further to judge if it is a useful way to build user interfaces to REA-based systems. Specifically, we have not accounted for controls of the visual appearance, nor for the absence of *forms*, which is a dominant concept in current information systems development.

Furthermore, it is not yet well understood if or how workflows, i.e. predefined sequences of user tasks, should be manifest in the user interface model.

Lastly, it is currently not known if the Partner is able to adopt this style of specification, and if there are technical concerns that prevent it.

REFERENCES

[1] McCarthy, W.E. *The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment*, The Accounting Review (July 1982) pp. 554-78.

[2] Szekely, P. *Retrospective and Challenges for Model-Based Interface Development*. In Proc. of WS on *Computer-Aided Design of User Interfaces*, Namur University Press, 1996.

[3] Pawson, R., Matthews, R. *Naked Objects*. John Wiley and Sons Ltd., 2002.

[4] Engelson et al. *Automatic Generation of User Interfaces from Data Structure Specification and Object-Oriented Application Models*. ECOOP 1996.

[5] Lauesen, S. & Harning, M. B. *Virtual Windows: Linking User Tasks, Datamodels, and Interface Design*. IEEE Software 2001. 7/8. 67-75.

[6] Chambers, C. *Predicate Classes*. ECOOP 1993.

[7] Jespersen, J.W. *Investigating User Interface Engineering in the Model-driven Architecture*. Proc. of WS on Bridging the Gap: SE and HCI at Interact '03.