High-level Declarative User Interfaces

[Poster Abstract]

Sofie Goderis
Vrije Universiteit Brussel
Programming Technology Lab
Pleinlaan 2, B-1050 Brussel, Belgium
sgoderis@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel
Programming Technology Lab
Pleinlaan 2, B-1050 Brussel, Belgium
tjdhondt@vub.ac.be

Categories and Subject Descriptors: D.2.2 Software Engineering: Coding Tools and Techniques

General Terms: Design.

Keywords: Separation of Concerns, Declarative Programming, User Interfaces.

1. PROBLEM DESCRIPTION

In order to survive in today's highly dynamic marketplace, companies must show a continuous and ever-increasing ability to adapt. This reflects on the adaptability requirements for the supporting software systems. Evolving a software system not only affects the source code responsible for the core application, but also the user interface. Our knowledge concerning software engineering tasks has grown considerably during the last 20 years and code entanglement has been tackled by several techniques such as aspect-oriented software engineering and component based software engineering. However few of these techniques have been applied onto user interfaces, especially for the concern of UI behaviour. Currently we still lack a clean way to separate and couple the user interface (UI) logic and the underlying application logic.

Adding application logic to a UI (e.g. the UI changes because of some business logic) results in mixing UI and application code. Adding UI logic (e.g. the UI changes because of some UI event), results in rather complex code to check the UIs state and undertake the appropriate actions. The code becomes even more complex when combining both application and UI logic. This kind of entanglement makes evolving and maintaining UIs hard.

2. SEPARATING USER INTERFACE AND APPLICATION LOGIC

In order to solve the problem of entanglement between the UI and the application, we apply the principle of separation of concerns to UIs. There are several parts to consider, namely the application logic, the UI logic consisting of visualisation and behaviour, and the UI-application behaviour. The application logic refers to everything that is not related to the UI, such as the application code itself, its interactions and the domain model. For instance in an e-business application, this is calculating the price to pay on checking out or determining if a discount is in place. The UI logic consists of visualisation and UI behaviour and interactions within this concern. Visualisation is what the UI looks like and what widgets are provided. Behaviour specifies how widgets relate to and influence each other. For example as long as the paying method has not been chosen, the proceed button is disabled. The UI and application concern interact with each other. This interaction describes how the UI concern hooks into the application. For instance creating the actual order and invoice because the user has clicked on the 'checking-out' button. This research focuses on the UI interactions and the interaction of the UI with the application.

3. DECLARATIVE USER INTERFACES

Current solutions to achieve a separation of concerns for UIs, only offer partial solution. The model-view-controller pattern [3] for instance only focusses on the interactions between the application and the UI, but neglects interactions within the UI. And user interface builders allow specifying a UIs visualisation and limited behaviour [4, 5, 1]. Especially when constructing more advanced and dynamic UIs, both approaches lack potential to avoid writing of complex code.

The goal we set in mind for separating UI logic from application logic, is to write down these concerns declaratively such that the UI is specified on a higher level. One needs to specify the UI logic and it has to be possible to generate certain parts of UI actions and interactions. We want the programmer to get rid of the burden of maintaining if-statements and call-back implementations (i.e. calling the application code from within the UI). Currently many of these issues are taken care of by ad hoc systems developed by the programmers on a need-by basis. We on the contrary want to provide a general solution that helps the programmer in creating UI logic. Therefore we use declarative programming as a means to express UI concerns and thereby offer a complete solution to tackle the UI - application entanglement.

Declarative programming describes and manipulates programs that deal with other programs in a declarative way. Logic facts are used to write down data or knowledge, while rules are used to reason about these facts and derive new facts. Declarative programming describes what the code does in contrast to how it is done.

3.1 UI visualisation

Low level UI visualisation describes for instance what a text field and a label look like while on a higher level one specifies what an input group for a user's name looks like. A more general visualisation is specifying that labels with certain properties should be positioned above an input field instead of next to it. The specification for the visualisation concern thus describes the visual components of a UI by means of facts and rules where different rule sets describe different levels of specifications, going from a low-level UI description to a higher-level description. By use of declarative rules, more general visualisations can be described.

Although the positioning of UI components is part of the UI visualisation, it can be influenced by the application. For instance when a user is younger than 18, the credit-card payment option is disabled and the UI components related to this payment option are removed from the UI. Therefore the UI visualisation for a youngster is different from the one for an adult. This means the visualisation has to change during its use and components have to be repositioned. Therefore an enhanced mechanism for automatically laying-out the components is required. A layout relation is for instance that a certain component has to positioned to the left of another component. Typically this can be transformed into a linear constraint equation by using the components' coordinates. A declarative reasoning mechanism is useful to perform this transformation. A linear equation constraint solver then resolves these layout relations and achieves automatic laying-out.

3.2 UI behaviour : Interactions at the UI level

A possible interaction at the UI level is disabling a 'checking-out' button as long as not all the required buying fields are filled out. Another example is specifying a default behaviour for a new-edit-save buttons group where the save and edit button are always disabled when a new button is enabled, such that this specification can be reused in different UIs. These kind of interactions between UI components are inevitable when creating UIs. A UI resides in a certain state, and certain actions or events will change that state. Often programmers express this behaviour by means of statecharts [2] which are then implemented manually into the application code, resulting once more in entangled code. Expressing statecharts is a declarative process that can be done by the use of facts and rules which in their turn can be translated into actual UI actions. Reusing a set of facts and rules allows for reusing certain states or default component behaviours.

If for a certain kind of user a business model requires extra data input, the UI behaviour changes because an extra UI form is added to the flow of the application. Also the visualisation is changed because some new buttons (e.g. previous and next) are added to existing parts of the UI. These runtime adaptations require dynamic interactions and the problem solving mechanism needs to react to these new facts only when a certain event happens. From a declarative point of view this means that, when a new fact is known (i.e. something happened), new conclusions and facts have to be inferred. Consequently using a forward chainer, and thus data-driven reasoning, is appropriate.

3.3 UI behaviour : Interactions between UI and application

Consider specifying that clicking the 'checking-out' button means an invoice is created and the delivery process is started. This UI event (i.e. clicking the button) triggers certain application events, thus both are linked together. Once more, how UI actions relate to application actions is described declaratively. Expressing that the UI has to be adapted because a regular client gets an extra reduction for Christmas, uses a combination of several business rules to influence the UI logic. The advantage of a declarative approach is that it allows to write such rules that reason about the underlying application model.

A UI is only useful if it is linked with an underlying application. This linking involves code generation (e.g. the if statement for the last example), but also code 'adaptation' (e.g. injecting this code) at the application level. We consider aspect oriented programming to tackle this problem.

3.4 Declarative UI Specification Framework

Combining the previous three levels of specifications, leads to a declarative user interface specification framework that will aid the programmer when developing the UI logic. Rules available in the framework can be reused, for instance the rules specifying how to create a Smalltalk label widget, or the rules to solve layout relations. This is especially interesting when reusing pre-defined layout strategies that put components in one column or multiple columns, or rows, labels on top of input fields, ... Unless the programmer wants to use a different specific specification (e.g. XML), a different layout strategy, a different reasoning mechanism, and so on, the programmer does not need to extend the framework with new rules or reasoning mechanisms.

We are implementing the declarative UI specification framework together with the necessary declarative reasoning mechanisms. As a starting point we use a declarative meta programming language (called SOUL [6]) which has been implemented on top of Smalltalk. It contains a primitive construct for evaluating Smalltalk expressions as part of logic rules. This allows SOUL clauses to reason about Smalltalk source code as well as executing a piece of Smalltalk code as part of a logic rule. We use SOUL to specify UIs by means of facts and rules, while the application logic is implemented in Smalltalk. SOUL's reasoning mechanism uses the UI facts and rules to generate an actual Smalltalk UI.

4. REFERENCES

[1] I. Chattopadhyaya. Gtk+ programming with glade. *Developer IQ magazine*, July 2002.

[2] I. Horrocks. *Constructing the User Interface with Statecharts*. Addison Wesley Professional, 1999.

[3] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, August/September 1988.

[4] B. A. Myers. Separating application code from toolkits : Eliminating the spaghetti of call-backs. In *UIST'91*, 1991.

[5] VisualWorks. Gui developer's guide. Technical report, Cincom Systems, Inc, 2002.

[6] R. Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, VUB, 2001.