Self-configuring user interface components

Pietu Pohjalainen
Department of Computer Science
University of Helsinki
pietu.pohjalainen@cs.helsinki.fi

ABSTRACT
In development phases of a software, its user interface is crucial to acceptance. In early phases, rapid prototyping helps in gaining sponsors for the development project. During development, the user interface is updated to meet changing requirements and, finally, maintenance-related tasks consume a major portion of effort. Some of this exertion is inherent and unavoidable, but very often it is just unnecessary overhead which is hindered by tedious internal dependencies being out of synchrony. In this paper, we show how a self-configuration via software introspection combined with semantic mapping of backend methods can be used to maintain quality of a user-interface even under pressure of changing requirements.

Author Keywords
self-configurating components, user interface, software engineering

ACM Classification Keywords
D.1.2 Automatic Programming: Self-configuring components; H.5.2 User Interfaces: Prototyping

INTRODUCTION

In software engineering, development of user interfaces is an important, yet often a time-consuming task. This originates partly from the inherent properties of software development: changes in operating environment or operational requirements propagate to the user interface as well. However, part of the spent effort is due to shortcomings in the internal development environment, defects in the architecture of the software, or limitations in the programming language.

In this paper, we present an introspection-based technique to reduce the required user interface maintenance effort in context of user interface components. The idea is that when composing the user interface from ready-made components, the binding expressions contain a lot of redundancy. By changing the binding code to be generated via introspective routines, we can get rid of copy-pasted code fragments, thus lessening required work for doing intended modifications. The method reduces user interface developers' need to concentrate their attention to small level details and allows them to rapidly generate consistent user interfaces.

The rest of the paper is structured as follows. The next section gives a brief overview to the current state of component-based user-interface construction methods with Java Server Faces (JSF). After that follows the case study with a running example, followed by a short evaluation of the case study. The next section reviews related work which after follows the conclusion.

COMPONENT BINDING IN THE USER INTERFACE

Let us consider the button as a common user interface component. The component has a binding place for a callback for determining whether the button should be active or disabled. There is also another binding place which specifies the callback for generating a tooltip when the user is hovering over the button component with the mouse. When the button happens to be in a disabled state, the user might be wondering the reason. For good usability, the reason could be told to the user via the tooltip. For an illustration, see Figure 1 where a disabled button has an attached tooltip explaining why the user can not press the button.

Figure 1. A tooltip explanains the UI behaviour

In modern component-based user-interface libraries, the components in the user interface, such as a textfield, or a button, are parametrized to conform to their current context. A binding language is used for this parametrization. For example, in JSF [2] the glue-code between the standard behavior defined by the component and the parametrized behavior could be as shown in Figure 2.

In the component binding sections, the attributes of disabled,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEMAIS 2010, February 7th, 2010, Hong Kong, China.

style and title define whether the component is active or disabled, what style-sheet should be used to render the component and what text is shown as the tooltip of this component. The binding is done with a special expression language, as defined in the JavaServer Pages specification [1]. Expressions to be executed during rendering of the component are marked by placing the expression inside the metacharacters of #{ and }. External configuration is used to bind names in certain namespace to classes of the host language environment.

```
<h:commandButton
 ...
 disabled="#{not
   (loginManager.loggedIn and
    loginManager.feedbackLevel >= 5)}"
 style="#{loginManager.loggedIn or
    loginManager.feedbackLevel >= 5)
    ? 'button'
    : 'button-disabled'}"
 title="#{not (loginManager.loggedIn
    and loginManager.feedbackLevel >= 5)
    ? 'You need to be logged in and
    have feedback level of 5 before
    posting with signature'
    : 'Post a comment'}">
</h:commandButton>
```

Figure 2. JSF component code for the Post-button

In this example, there is a correlation between three attributes of the component. Each of the attributes is defined by means of login status and the feedback level of the user: only users who are logged in and have feedback level greater than five can post comments, others need to post anonymously. Clearly, the implementing code for these callbacks go hand in hand. Let us say that the required feedback level for posting comments non-anonymously is changed to three. Now the changed rules for disabling the button need to be mirrored also in the corresponding binding expressions.

In this text we show how this kind of code copying can be attacked. This case study demonstrates how the concept of self-configuration can be applied in software engineering via usage of introspection and reflection mechanisms that are available in current programming environments. We target to JSF, as it is a widely used standard in web development and its component binding language is easily analyzable. This enables a reflective analysis of the component bindings, and makes the analysis more practical than self-configuration of components built in full-fledged general purpose programming language.

SELF-CONFIGURATION OF UI COMPONENTS

Our solution to reducing this duplication is built on two components: a generic evaluator of binding expressions and a domain-specific semantic model of expression elements. The target is to enable faster prototyping of user interfaces without sacrificing code quality.

The generic evaluator decomposes an arbitrary binding expression to its atomic elements. In Figure 2 the *disabled*-status would be a natural choice of being analyzed into form of $\neg(loggedIn \wedge feedbackLevel \geq 5)$ and exports the status to be used in other binding expressions. This evaluator comes as a ready-made component to the platform and is re-usable in different projects.

A semantic model of the application domain is used to produce domain-specific parts in the binding expressions. For example, the two domain-specific methods in Figure 2 are *LoginManager.loggedIn* which tells the login status of the user and *LoginManager.feedbackLevel* which returns the feedback level of the user. These two conditions are combined not only in the *disabled*-status evaluation, but also when producing the tooltip for the button. The net result is that the user does not know without external reference if he is not allowed to push the button because he has not yet logged in, or due to insufficient feedback level.

An obvious solution to the problem is to write specific branches for both of the cases by using the tertiary if-operation. However, enumerating all the possibilities in such a way gets rather complicated, as the number of different possible combinations grows exponentially when the number of atomic parts grows in the expression. Because the expressions are also reused in many contexts, it would be only a matter of time before the specifications are no longer equivalent.

Instead, our target is to remove the duplication between the user interface's *disabled* status and its corresponding tooltip. Figure 3 shows the same functionality, implemented as a self-configuring user-interface component. There are two differences: first, the used stylesheet is chosen by referring to another expression in the component's bindings via the *disabled*-status instead of copying the original expression. The another difference is that the tooltip for the component is generated by a special function *getTooltip*.

```
<h:commandButton
  ...
  disabled="#{not
      (loginManager.loggedIn and
       loginManager.feedbackLevel >= 5)}"
   style="#{!MyBean.disabled ? 'button' :
           'button-disabled'}"
   title="#{MyBean.tooltip}">
</h:commandButton>
```

Figure 3. Self-configuring component

By introducing domain-specific semantic mappings between elementary expressions and their meaning in the user interface, the generic evaluator can produce more detailed feedback in this situation. As the generic evaluator has decomposed the expression into its elementary parts, it can determine the exact reason why the button in question is disabled. This information can then be used to automatically generate related binding expressions, thus enabling better quality and faster development.

The semantic mapping describes the bridge between the com-

ponent binding expressions and the user's vocabulary: the method call of $LoginManager.loggedIn$ is mapped to a natural language description of "you are {not} logged in". The other method call, $LoginManager.feedbackLevel$ is mapped to "feedback level is lower than {%1}". The generic evaluator uses these strings to automatically generate the correct tooltips without introducing additional maintenance overhead.

Technical details

The *getDisabled* method in MyBean re-evaluates the expression in the *disabled*-attribute of the component and based on that result, the component chooses the correct stylesheet to use. It is important to notice that the original expression is now referred to in the evaluation, thus the semantic link between the used style sheet and the original expression is remaining. This is already an improvement over the the previous example. The *getTooltip* method in MyBean is a more complex case. In addition to the information of which expression to use as the original specification, it contains the domain-specific semantic mapping for generating the tooltip.

There are two different re-evaluation strategies used in this example. The first one is a simple reusing of the original expression. In this case, the only benefits are cleaner syntax in the component specification and the re-established link between original behavior-specifying expression. The second strategy is to use a generic evaluator with semantic mappings to generate a more meaningful error message for the user.

Generic evaluator

The generic evaluator performs its job in four phases. First it decomposes the given expression into atomic propositional formulae, then builds a truth table for the expression. Third step is to select appropriate semantic descriptions to be returned and the final step is to build up the response.

Decomposing specification expressions

The expression language used with JSF is rather simple to analyze. The language contains only expressions, with literals, connectives and operators. Literals can be boolean, floating point, integers, strings or nulls. The two available connectives are and ($\wedge$) and or ($\vee$). A set of standard operator consists of relational operators for equality and comparison operators of equals, not equals, greater than, less than ($=$, $\neq$, $>$ and $<$), arithmetic operators of plus, minus, multiplication, division and modulo ($+$, $-$, $*$, $\div$ and $\mod$) and unary operators of negation and minus ($\neg$ and $-$). Finally, bridging to methods in the host language, which is usually Java, is provided via value bindings and method bindings.

A value binding expression is a unifying shortcut for gaining access to accessor methods (setters and getters) of objects in the host language. For example, in Figure 3 the expression #{MyBean.tooltip} is a value binding which tells to use methods named $getTooltip$ and $setTooltip$ to in the object named $MyBean$ in the host environment when the user interface component needs to receive or modify the value of its title. A method binding is a way to express the method name in object in the host language's environment, which should be called in certain situation.

Although the syntax contains a few elements that are designed to give friendly programming experience for a programmer who does not remember the exact syntactical and semantical constructs, the language is easy parse into object representation, which can be used for further analysis.

Building a truth table

Execution of logical expressions in many programming languages is short-circuited. This causes the execution to be stopped as soon as the final result is known. For example, when the evaluation of A in expression $A \vee B$ returns true, the expression B is not executed at all, because the expression will be true regardless of its return value.

Implementing short-circuited semantics is one possible solution to selecting the semantic description that should be shown to the user. Unfortunately this tactic has the shortcoming of then producing only the most immediate reason for the component not to be active. Consider again the expression $A \vee B$. The component under processing is not active if either of the reasons, A or B holds. With short-circuited evaluation when both of the conditions hold, the user is only informed about the condition A. Frustration occurs when the user changes the condition A only to find out that the condition B, which he was not aware at all, is still blocking his access.

For this reason, we build a truth table for the expression. A truth table is a enumeration of all possible combinations of atomic conditions in the expression. For example, the truth table for expression $\neg(loggedIn \wedge feedbackLevel \geq 5)$ is as shown in Figure 4.

$loggedIn$	T	T	F	F
$feedbackLevel \geq 5$	T	F	T	F
Result	F	T	T	T

Figure 4. Truth table for the expression

From the truth table, we can see that with this particular expression, if either of the atomic propositional formulae returns false, then the whole expression returns false.

A truth table clarifies the functions of atomic components of an expression. However, for an expression with n atomic components, there are 2^n columns in the table. If the number of atomic expressions is a concern, we can use a combined truth table, which prunes redundant columns, replacing the truth value in a column with symbol T/F, representing both true and false. However, in practice the number of atomic expressions is fairly low, so the potential exponential size of the truth table is seldom a practical problem.

Selecting semantic descriptions

From the truth table, we can see what conditions should be changed to activate the component. In this phase, we also

consider our current conditions and use these with the semantic map for domain-specific descriptions.

We start by evaluating the whole expression in a non-short-circuited manner and memorize the result of each of the atomic propositions. The full set of atomic results is used to find the corresponding column in the truth table. For example if the user is logged in, but has only feedback level of 3 instead of expected 5, the second column is chosen in the Figure 4. From this column we select each description for atomic conditions that are different than what is expected to get the final result to change its truth value.

There are three concerns in this phase. The first concern is the change in the expected execution semantics. The original expression might contain side-effects that rely on short-circuited semantics. In this case, fully evaluating the expression can perform erroneously, although the original expression always works correctly. For this reason, the evaluator can be instructed to use non-short-circuited semantics in its evaluation, as a fallback for this case. A drawback with this option is that only the most immediate error message can be shown to the user.

The second concern is about how to know which column in the truth table is the desired one. In Figure 4 it is not a problem, as there is only one column which unambiguously activates the component in question. But in general there is no guarantee that there would be exactly one set of atomic conditions which yields activation for the component. For this reason, the evaluator can be given a target value for each of the atomic propositions, which should be preferred in case of ambiguity.

The third problem is that some of the atomic propositions are not meant to be shown to the user. This may be because the propositions might be regarded as just part of technical implementation, or maybe the information should be shown only conditionally. Whatever the reason for not showing the explanation for a certain atomic proposition, the generic evaluator can be instructed to leave some propositions out by not giving a corresponding semantic description.

Building up the response

The final phase for the generic evaluator is to yield a comprehensible message to be displayed for the user. This is done by selecting each atomic proposal whose value was found out to differ from the value expected by the desired column in the truth table. Each of the associated messages are processed through a simple parametrization filter, which allows the messages to contain situation-specific parametrization.

For example, in the semantic message *You are {not} logged in* the middle not is used to parametrize the outgoing message. If the associated atomic proposition returned true, then that part is not included in the message. This way, the same message can be used in many contexts. Other parametrization possibilities include using {%1} for referring to the other member of a relational operator. For example, in the semantic message of *You have feedback level lower than {%1}* the last part is replaced by the value received from the actual expression.

Finally, each single message is concatenated to produce the final message which is shown to the user.

EVALUATION

The generic evaluator presented in the previous section is a way to move the burden of updating dependencies within a software from the programmer to a runtime introspective component. This generates a minor overhead for the execution environment. It can be noted that the whole concept relies on backend methods returning consistent values over subsequent invocations. However, this tends not a problem, as the overall philosophy with JSF accessors is that they are not destructive for the accessed data and should have no visible side-effects.

The current implementation is library based, meaning that no changes to the base frameworks were made. This makes it easy to reuse the solution in other projects using JSF. Having made no changes to the base framework also caused few implications, which make it clumsier than necessary for the programmer. Also some optimization opportunities have been missed.

Our first complaint with this solution is the binding between user interface components in the presentation layer and backing software. Currently, there is no concept of *this component* in the backing bean. This means that every component that is using the generic evaluator for generating tooltips needs a binding methods of their own; the first task of the bound method is to find the corresponding user interface component from its environment.

The second shortcoming is associated with the missed optimization opportunities. The generic evaluator re-evaluates the bound methods quite a many times, as the the framework provides no opportunity to memorize the previously evaluated results. By changing the framework to provide such opportunities, or maybe using a memoizing aspect component, the architecture could be changed to evaluate each bound method only once, thus saving in rendering time.

RELATED WORK

The scope of this paper is to present a way for reducing effects of unnecessary copying of program logic by characterizing an introspective framework for expressing relationships that are identified at programming time. The motivations is that seldom there is a way for annotating these relationships in the code; self-analyzing code is too often considered to be too complex to write and maintain. There is a certain degree of irony, as instead an anti-pattern of copy-and-paste [5] is often used.

Detection of software clones is an established activity in the area of reverse engineering, and a number of papers have been written on the topic [3, 4]. In this field, automated tools are applied to detect code duplicates for a human engineer to refactor.

In many situations, however, the code cloning happens due to limitations in the programming environment – the problem is not about finding or understanding that code has been duplicated, but rather that there just is no better way than copy and paste to express the relationship between two code elements. This has proven to be laborous when analysing a full-fledged, general purpose programming language. For example, previously we applied similar approach in analysing database queries generated by an object-oriented program [10]. The provided level of abstraction turned out to be rather low-level, which made the analysis much harder than analysis of JSF's expression language. For these situations, Hammouda et al. have introduced the concept of *maintenance patterns* [8] for expressing advice for future maintainers of a certain piece of software.

Maintenance patterns are documentation for a maintenance programmer, for performing anticipated maintenance tasks. Often these tasks require changes to more than one code location; the documentation in the maintenance pattern carries this knowledge from the original designer to the maintenance programmer (who might be the same person, who has already forgotten the required actions).

Automatic refactorings, such as hierarchy restructuring [9] are nowadays a standard in modern development environments. For example, the Eclipse IDE has gained much popularity, partly due to its large offering of automated refactorings for Java code [7]. However, automated refactorings tend to be applicable only to structural dependency handling, while our self-configuring components are targeted to annotate and handle conceptual dependencies.

Many web programming frameworks, such as Ruby on Rails [11] or Lift [6] have been organized around the concept of interweaving scripting instructions for generating web page output with templating engines. Many frameworks also offer sophisticated means for automatically generating the final output.

Although the techniques shown here are totally applicable in these environments, the expressive power of a general purpose language makes it much harder to self-analyze the meaning of a part of a program. Generative approaches can be seen as complementary techniques to attacking a similar problem. However, with generative techniques, there often is the question of how to keep the model and generated parts synchronized, especially if it is possible to make changes to the generated parts of the program.

CONCLUSION

In this paper's context, we have employed the Java Server Faces framework and its expression language to demonstrate the need for self-configuring components. The intention is to reduce maintenance work by reducing the number of internal dependencies in the software. This is supposed to prevent the deterioration of software usability as the internal structure of the software is automatically maintained.

A combination of introspecting software components with semantic augmentation is not widely employed in current technologies. However, reducing the number of internal dependencies within user-interface code is crucial when attempting to reduce the required maintenance effort in software projects. This is important, as there is a link between usability and the required maintenance effort. If the software is hard to maintain, its usability will deteriorate over time. On the other hand, if maintenance tasks are easy for the programmer, the usability stays in the originally intended level.

REFERENCES

1. JSR 152: JavaServer Pages specification, version 2.0. Technical report, Sun Microsystems, 2003.
2. JSR 252: JavaServer Faces 1.2. Technical report, Sun Microsystems, 2006.
3. B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Washington, DC, USA, 1995. IEEE Computer Society.
4. M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE '00: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107, Washington, DC, USA, 2000. IEEE Computer Society.
5. W. J. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.
6. D. Chen-Becker, T. Weir, and M. Danciu. *The Definitive Guide to Lift: A Scala-based Web Framework*. Apress, Berkely, CA, USA, 2009.
7. D. Geer. Eclipse becomes the dominant Java IDE. *Computer*, 38(7):16–18, 2005.
8. I. Hammouda and M. Harsu. Documenting maintenance tasks using maintenance patterns. In *CSMR '04: Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 37–47, Washington, DC, USA, 2004. IEEE Computer Society.
9. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *In Proceedings of the 11th annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250. ACM Press, 1996.
10. P. Pohjalainen and J. Taina. Self-configuring object-to-relational mapping queries. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 53–59, Modena, Italy, 2008. ACM.
11. S. Ruby, D. Thomas, and D. Hansson. *Agile Web Development with Rails*. Pragmatic Bookshelf, third edition, 2009.