Towards Smart User Interface Design

Tomas Cerny
Dept. of Computer Science and Engineering
Czech Technical University
Prague, Czech Republic
tomas.cerny@fel.cvut.cz

Vaclav Chalupa
Dept. of Software Engineering
Czech Technical University
Prague, Czech Republic
chaluva2@fit.cvut.cz

Michael J. Donahoo
Dept. of Computer Science
Baylor University
Waco, TX, USA
jeff_donahoo@baylor.edu

***Abstract*—User interface development and maintenance presents a burden for many developers. UI development approaches often restate information already captured in the application model such as entity attributes, validation, security, etc. Changes in application model often require many subsequent changes to the UI. Such duplication creates additional maintenance requirements for synchronization (at a minimum) and often is a source for errors (i.e., when model and UI disagree). Adding to the difficulties of creation and maintenance, typical UI implementations often tangle multiple concerns together such as presentation, validation, layout, security, etc. In this paper, we provide an approach that reduces information duplication and untangles mixed concerns. The capability of runtime UI generation can render user-specific UI, reduce conditional evaluation, and integrate third party security frameworks. To evaluate our approach, we provide a case study that demonstrates reduction of maintenance efforts, separation of concerns and performance of runtime UI generation.**

***Keywords*—User interface development, model-driven development, automated data visualization, code inspection**

I. Introduction

To be successful, large enterprise applications must provide users with well-designed, friendly interfaces that assist them with interaction. For this reason developers invest a lot of efforts in the UI design. This brings on the other side a lot of development and maintenance efforts. Research of [1] has shown that around 48% of total application code and 50% of application time is devoted to implementing UIs. Application costs seem strongly influenced by the UI development. In fact, the situation is even more complicated when we talk about maintenance. Applications introduce dependencies and coupling between lower layers and UI components. For example, once a data model is modified, the change must be reflected in UI as well. Otherwise, an inconsistency between layers will exist. Applications using languages without type safety for the UI design may discover inconsistencies very late, as no validation mechanisms exist.

Manual application development cannot easily address coupling between data model and UI visualization. One solution allows the the data model to directly produce its components such as forms and tables. Such expectation from the data model is complicated since data presentation may be visualized in multiple ways. In addition, there exist factors such as user rights, application context, and input validation that influence the visualization. We propose a mechanism that produces UI components based on information captured in data model, greatly reducing the costs related to UI development and maintenance.

In this paper, we discuss existing approaches on development and maintenance of UI part of enterprise applications. We suggest automation through source code inspection and information transformation. To achieve this we define key characteristics for the approach to provide flexibility to produce any kind of UI component that could be developed manually. Such automation avoids inconsistency errors often made manually, it allows us to simplify UI internationalization, tooltips, client-side validation or various forms of UI for different users. Furthermore, it is possible to generate UI in runtime, which can influence the result based on user rights and context, or support integration with third-party frameworks. In some cases, runtime UI generation reduces conditional evaluation in UI and improve performance. In order to explore benefits and disadvantages of the automation a case study is provided. In the study we define an evaluation metrics to compare manual and automated approach applied to a web application development. This study demonstrates that automation provides considerable reduction of coupling between UI and the data model as well as reduction of source code and maintenance efforts.

This paper is organized as follows. In Section II we classify existing approaches on UI development. We discuss related work in Section III. Our approach to automation is described in Section IV. In the next chapters we provide a case study and approach evaluation. The work is concluded in Section VII.

II. Background

Existing approaches to UI development and maintenance can be divided into two categories: *restate-to-extend* and *inspection-based* [2]. The *restate-to-extend* category requires software developers to restate information already captured elsewhere in the application in order to extend it for the presentation. Such duplication can be a source for errors and additional maintenance efforts due to duplicate synchronization. Examples of *restate-to-extend* include manual development, interactive graphical and model-based generation tools.

Inspection-based category uses information already captured by the application accessible by code inspection (often in a data model). The effort is placed on the information origin that must capture sufficient information to derive a specific aspect (in this case presentation). This can be seen as a

978-1-4673-1401-5/12/$31.00 ©2012 IEEE

disadvantage since not necessarily all information are captured at the same location. Examples of *inspection-based* approach are language-based tools [3] [2].

There exists a two approaches to capture additional information in modern frameworks [4] [5]. On one side extensions are captured separately in XML that restate existing information in the source code, on the other side similar extension is achieved by annotations that bind to the annotated element. Similar approaches are seen in UML language. We can use two diagrams to capture an extension information, but in some cases an UML stereotype can be used [6].

Inspection-based approach to build UI has the advantage of no need to restate an existing information, while a specific aspect must be derivable from the origin. This could mean that the origin of the information may need to capture a large amount of information, which can get tangled, decrease cohesion and increase coupling [7]. What might be experienced is something called cross-cutting concerns. One possible solution to this is aspect-oriented programming (AOP) [8]. The aim of AOP is to deal with cross-cutting concerns in an application. AOP defines a *join point* to be a place where different concerns cross in a given module. Multiple of those *join points* are then described by a *pointcut*, which also specifies separately defined code, an *advice*. Different aspects can be defined and added to the origin with clear separation of concerns. An aspect is then weaved through an aspect weaver into the origin to produce the aimed result (in our case UI). This greatly supports maintainability and separation of concerns.

III. Related work

Significant amount of work has been done in the *restate-to-extend* category of the UI visualization of data models [1] [9] [10] [11] [18]. Multiple tools can be found online allowing drag and drop UI widget placement and binding with the underlying data model. Multiple advanced Integrated Development Environments (IDE) provide such functionality as well. The main drawback of the approach is information duplication, complex maintenance and a tendency to introduce an error in the maintenance. Similar drawbacks are introduced by tools using models. Those models get stale once the origin information is modified. One promising approach is model-driven development (MDD) [12] [13], where UML model is the base artifact where the modification is made. The model is then transformed to source code producing consistent application, thus consistent UI [14] [15]. An approach extending UML data model to allow UI component generation [16] [17] describes meta-information necessary to capture by data model in order to fully generate desired UI components with user input validation.

Inspection-based UI visualization of data models has three main contributors. Naked objects [3] introduces the idea of rich data objects that provide enough information for deriving the UI. The framework of naked objects is successfully used for the Irish government to build information system for the department of social protection.

Tools MetaWidget [1] [2] [19] and FormBuilder [20] apply source mining and a module inspection to Java-based applications. The main differences between the tools is that MetaWidget contains fixed collection of UI widgets which developer cannot modify. FormBuilder provides access to the UI widget collection, which makes it possible to adapt to any framework, mobile applications, domain specific languages or web services. This also allows developer to use a security framework, view template framework or form field access control.

An inspection-based approach can also be applied to cultural interfaces in which interface differ for users with different heritage. [21] stresses the importance that culture plays in human-computer interaction, although, no implementation or study is provided. Our approach and tool we provide has the capability to be applied for the purposes of cultural interfaces.

IV. Approach to the UI design

Our approach to design UI is *inspection-based*. It can be seen simple to generate basic views allowing CRUD operations, presenting plain forms and tables. Most likely a production level application does not only contain plain forms, but diverse layouts, different sets of fields for various users, it applies user input validation, or offers large palette of fancy UI widgets. In addition, some of UI components for given data are custom. Furthermore, developer may want to apply the approach to an existing legacy application as an improvement to the maintenance. Thus it must be capable of adoption to multiple programming practices and styles. Our approach should be capable to adapt to such requirements.

Kennard and Leaney [22] aim to describe key characteristics necessary for UI generation. Although, they identify a need for inspection, different transformations, UI widget libraries, widget adornments and layouts, their characteristics are not complete. For example large enterprises and legacies often embed an external security framework to apply authorization to users. It is common that security restricts users to see a subset of existing data in the UI. Kennard's characteristics could apply the security through different transformations or widget adornments, but such approach would result with tangled code and low performance of the generated component. In addition, it would mean that the security rules are restated in the configuration for the approach and that is what we want to avoid. Security framework cross-cuts the system, as well as the UI, thus it should considered as a key element for the approach. As next, multiple legacy systems are integrated through a portal technology. Such systems may need to use different UI widget libraries or sometimes even mixes of those. We must expect that there exists a significant diversity in the UI components and the approach should allow complete customization to the transformation elements. This requirement can be achieved through configuration be exception.

In summary the approach assumes an existence of the following inputs:

- Existing system to inspect and to build meta-model
- Existing security framework in the system
- Transformation profile
  - Mapping rules
  - UI widget templates
  - Layout templates

The life-cycle starts with system code inspection that populates a meta-model and injects security rules to it. Gathered meta-information are applied in transformation based on selected transformation profile. This profile provides rules how to map given meta-information to UI widget templates in order generate UI component. UI widget templates are code templates that use special markup that is interpreted by the transformation in a way that the meta-model decorates its information to the markup. This way an UI component contains the same information as captured in the inspected code (data model). The configuration of exception means that a general configuration exists in the profile for the standard transformation. If it is necessary a new mapping rule can be added or a nee UI widget template can be developed. This provides the designer with full control over the generated output.

So far we discussed automated approach, but we did not mention when should the UI generation take place. In fact two options are possible. Either UI components are generated statically before application deployment or the UI generation might be delayed till runtime, in such a case an UI component is generated on demand when it is necessary. The runtime option fits better to existing security frameworks, they often interpret application context in runtime to resolve user rights. It may result in multiple versions of the same UI component for various user roles. In contrast, for manual development it is common to create one complex UI component with conditionals. Such component is better from the maintenance perspective than multiple replicas, but this decreases cohesion and readability. With automated approach we do not need worry about maintenance of the UI component as it is generated.

Based on our research and analysis we provide key characteristics necessary for automated UI generation adaptable by legacy systems.

1) Inspection of existing heterogeneous systems
2) Security framework awareness
3) Adjustable transformations of inspected results
4) Customizable UI widget templates for a selected platform
5) Applying multiple layouts

Based on description of our approach we implement a tool capable of code-inspection and UI generation both statically and in run-time for Java applications. To evaluate the approach with comparison to a manual development we provide a case study in Section V.

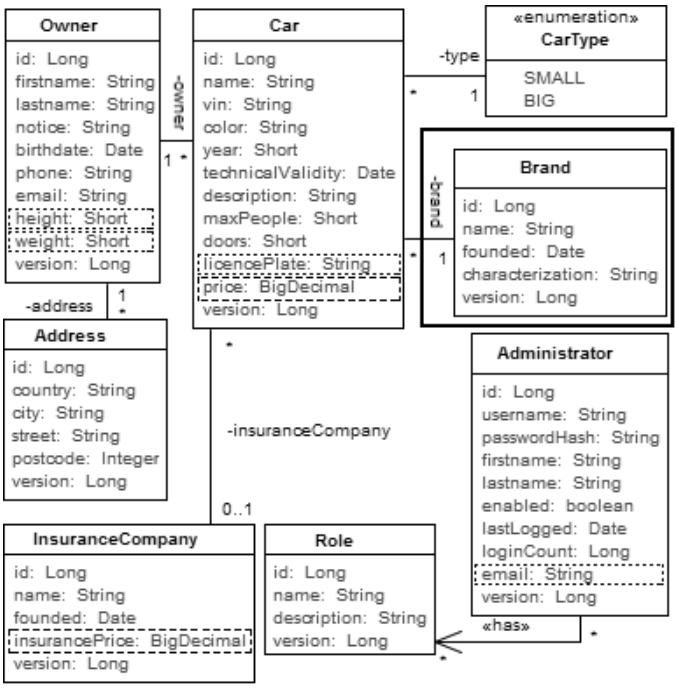

Fig. 1. Case study data model (maintenance changes in boxes)

TABLE I
UI COMPONENTS GENERATED FROM THE DATA MODEL

Class / UI component	Search	Edit	Detail	Passwd. change	List	Assoc. list	Inverse Assoc.
Administr.	X	X	X	X	X	X	
Address	X	X	X		X	X	
Car	X	X	X		X	X	
InsuranceCom.	X	X	X		X		X
Owner	X	X	X		X		X
Role	X	X	X		X	X	
Security restrictions for the case study							
User role / UI component	Search	Edit	Detail	Passwd. change	List	Assoc. list	Inverse Assoc.
Guest	X				X		
Administrator	X	X	X	X	X	X	X
Editor	X	X	X		X	X	X
Reader	X		X		X	X	X

V. CASE STUDY

In this chapter we provide a study comparing manual development with our approach. As next, we evaluate static and on-demand versions of the approach, regards performance. For the study we use J2EE application implemented in framework Seam. The data model of the application used for our evaluation is available at Fig. 1. The application differentiates four user roles (Guest, Admin, Editor, Reader). The presentation layer uses UI components for searches, modification with validation, detail page, lists of elements and association lists. Table I shows available features and components used for given entity, bellow it also shows access rights for considered user roles. Our tool is used for our approach in this study.

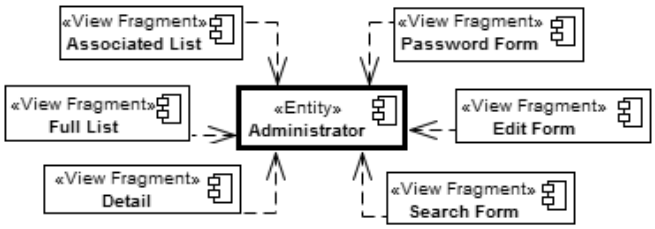

Fig. 2. UI component/data model dependencies with manual approach

A. Evaluation of creation and maintenance impacts

To see the impact of our approach on development and maintenance we compare manually developed application with one applying our approach [23]. The manually developed application is built with specific UI design and consistent conventions. This application is evaluated regards its size of UI part, coupling in UI and required changes for selected maintenance scenarios. The same application is refactored to apply our approach on UI part. The resulting UI of the modified application looks alike the original application. The modified application is evaluated regards the same criteria as the original one.

Regards maintenance and development we consider the amount of files in the UI part of the application that must be modified (MoF) when the data model changes. We also look at how many source lines of code (SLOC) must be modified to reflect the change. As next, we consider the amount of security conditions (SeC) in the UI. Each considered maintenance scenario is evaluated by these factors MoF/SLOC/SeC (MSS). For the measurement we use a tool Unified Code Count [24], for the Java code we consider logical lines of code, but for XML and XHTML physical lines are considered (without comments and blank spaces).

Manually developed application consists of 4290 SLOC[1], the UI part embeds 29 SeC. Once a data model class changes it may invoke up to 6 immediate changes in coupled UI components as drawn in the dependency diagram in Fig. 2. These dependencies reflect maintenance efforts. As follows, we consider selected maintenance scenarios (a-e). When we add new class fields (a) *email, height, weight* and *license plate* to the data model, as shown in dotted boxes in Fig. 1, then all coupled UI components must update. We can use components for texts and numbers, which needs us to modify 12 files (MoF), add 126 SLOC with 2 new SeC. This scenario makes $12/126/2$ MSS changes. Slightly different scenario is to add fields with new data type (b) to the data model (*price* and *licensePrice*). New components for prices are used. For this case we need to make $8/51/4$ MSS modifications. To add a new class (c) *Brand* with 5 field and association to *Car*, new UI components must be made for searches, modification, details and also for associations. This task requires $7/121/4$ changes of MSS. As next, we consider a scenario when we aim to change field order (d) for the class *Car* in the UI forms.

[1]Values in SLOC: 924 Java, 3175 XHTML and 191 XML

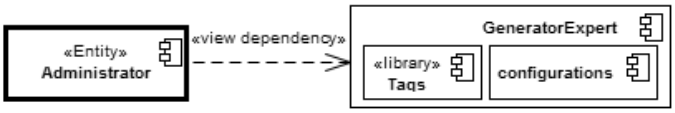

Fig. 3. UI component/data model dependencies with automated approach

We need to access both forms and modify $2/54/0$ MSS. As last we apply text resources (e) for all UI labels and texts to support internationalization. All UI components are accessed and updated with $29/213/0$ MSS.

When applying our approach all UI components with re-stated information are removed. Instead we develop generalized UI widget templates for texts, numbers, etc. in editable and readable form. These templates are decorated with the tool-inspected information in the generation process. There are 27 templates with total of 184 SLOC. For the inspection of existing source-code information to the meta-model and consequent transformation to the UI components we need mapping rules. These rules map the meta-information to the UI widget templates to assemble a UI component. These mapping rules could be shared among projects, in our case we have 3 profiles with total of 162 SLOC. The dependency of the data changes reduce accordingly to the transformation rules and templates as shown in Fig. 3. The modified application consists of 3758 SLOC[2], total amount of conditionals in UI reduces to 10 SeC. What also change are the efforts for maintenance scenarios. The addition of new fields with existing data type (a) reduces to $3/6/0$ MSS. Although, similar reductions is not seen for not existing template type, this is because a new template must be developed. To add fields with new data type (b) we need $8/48/0$ MSS. It is important to emphasize that for large projects this scenario is rare to occur. When we add a new class (c) efforts reduces to $9/41/1$ MSS. Field order (d) can change simply by data model marks setting, which results in $1/2/0$ MSS changes. Internationalization addition is reduces as well, since no replicas exist the task requires $27/32/0$ MSS.

Both evaluations by our metrics are summarized in Table II. It can be seen from the study that benefits of the approach are significant reduction of SLOC in maintenance and project evolution, reduction of files that must be modified (MoF) and view SeC. The approach makes it easy to extend existing components with new aspect and propagate the change to all components (internationalization, help, etc.). This study does not cover all possible UI operations, but it demonstrates common cases used in existing systems.

B. Performance evaluation

In the previous section we show reduction of maintenance efforts and dependencies, but you do not buy it. Lets consider the following. An UI component should render differently to a specific user. For example UI component for *Car* shows

[2]SLOC: 1036 for Java (including data model marks 81 SLOC and direction to the inspection 31 SLOC), 2268 for XHTML (including templates) and 454 for XML (mapping rules and configuration)

TABLE II
EVALUATION FROM THE MAINTENANCE AND CREATION PERSPECTIVE

Scenario	MoF	SLOC	SeC
Manually developed $4290 = 924 + 3175 + 191$ SLOC (Java/XHTML/XML)			
a. New class field with existing UI widget	12	126	2
b. New class field with new UI widget	8	51	4
c. New class in the data mode	7	121	4
d. Change of field order in the UI component	2	54	
e. Internationalization	29	213	
With our approach $3758 = 1036 + 2268 + 454$ SLOC (Java/XHTML/XML)			
a. New class field with existing UI widget	3	6	0
b. New class field with new UI widget	8	48	0
c. New class in the data mode	9	41	1
d. Change of field order in the UI component	1	2	
e. Internationalization	27	32	

TABLE III
APPLICATION LOAD TIMES

1. Plain UI component

Administrator			
Type	Manual	On-demand	On-demand (cache)
Time [ms]	150.35	156.85	145.75
Relative [-]	1	1.043	0.969

Guest			
Type	Manual	On-demand	On-demand (cache)
Time [ms]	92.85	98.75	87.80
Relative [-]	1	1.063	0.945

2. Rich UI component

Administrator			
Type	Manual	On-demand	On-demand (cache)
Time [ms]	257.25	263.8	256.75
Relative [-]	1	1.025	0.998

Guest			
Type	Manual	On-demand	On-demand (cache)
Time [ms]	203.8	217.05	188.25
Relative [-]	1	1.065	.924

to *Administrator* 11 fields, *Writer* 4 fields, but to *Reader* only 2 fields and to *Guest* just one. This means that we either need to have 4 different versions of UI component for *Car* with the same purpose to serve different user roles or we need to develop one complex component with multiple conditionals. If we consider that we have for *Car* a component for search, modification, detail, etc. we may end up with many files to maintain, often we rather develop a complex file with conditionals, such file tangles multiple concerns together, has low cohesion and is hard to maintain.

With our approach it is possible to automate user role specific component generation with no impact on maintenance. Furthermore, there is possibility to generate user specific UI components at application deploy time, or on-demand in runtime. The on-demand version can be farther influenced with an external security framework that evaluates actual user context.

We compare performance of manually developed components (multiple versions) with on-demand versions of our approach. Our evaluation considers application response times for both types of UI components. The on-demand generation can be further extended with caching of previously requested components. In the evaluation we consider plain UI components as well as UI components with addition components to see the impact in real environment. Regards the security we consider two users a *Guest* and *Administrator*. *Administrator* should see 11 fields and the *Guest* only 1 field. Results are provided in Table III. In both plain and rich UI components we see that on-demand generation is slightly slower (4-6% in response time), but with caching the reponse times improve. The benefits of on-demand generation turn out well mostly for components that contain many conditionals.

VI. EVALUATION OF THE APPROACH

In previous section we provide an evaluation of the approach in comparison with manual development. The maintenance and creation evaluation shows that our approach positively affects both of the parameters. The weakest part can be seen as an addition of a field that requires a new UI widget. Fortunately this does not seem to be a frequent scenario, as a common library exists and can be shared among projects. The time spent with the component development will amortize over time with it's reuse. For example, if we would have a similar project with 10 data classes, then XML part would not change, XHTML would experience approximately 28% code reduction, Java part would consider addition marks (9%), although the exact value would be influenced by many other parameters (security, associations, expected UI features). The main benefit that can be seen from the approach is a centralization of a decision. This means that once we need a change to be applied in almost every UI component (form, table..), we only need to make minimum amount of changes and do not need to replicate the modification in multiple resources. Our approach provides benefits similar to aspect-oriented programming, that allows us to separate different concerns. These concerns often tangle together in the source code in manual development, which complicates its readability and cohesion.

The performance evaluation shows that the approach can be applied in both manual and on-demand (runtime) versions. On-demand version provide comparable performance in terms of application load times. This on-demand UI generation provides benefits of integration of additional aspects such as user preferences, security context, client browsing capabilities etc. It is also important to note that with our approach we can generate multiple versions of UI components for different users. This may result in better performance than manual ap-

proach since one complex manually developed UI component with conditionals must be evaluated at runtime.

Our approach is applied in a large enterprise application used in production. It is used to generate UI components to simplify development. Our practical experience is positive as this approach saves a lot of time that would be spend with propagation of new or changed information to the UI. We find this approach error-free since not man factor is involved for tedious development.

VII. Conclusion

In this paper we discussed existing approaches to UI design. Out of currently known approaches only one does not need to duplicate information already available in the application. Often such approach does not provide UI expected by various developers and may not fit legacy systems. Based on our research and related work we identify key characteristics for the approach to provide full customization for any kind of application.

Our approach is evaluated in a case study that compares it with manual development. Results of the study show reduction in maintenance, elimination of duplicates and reduced coupling. It also shows that manual development often combines multiple concerns together that makes the maintenance more complex. Our approach can separate concerns such as security, user preferences, etc. Furthermore, our approach allows to generate UI on-demand. Our evaluation shows that performance of on-demand version is similar as static one at deployment. On-demand UI generation brings new perspectives for developers, it is possible to integrate it with actual runtime context, third party security frameworks, user specific decisions or browser capabilities.

Our future work will focus on a platform independent visualization capable of transformations of inspected metadata, or provided inputs (such as XML. XLS) into various domains, such as UI, web services, testing data, platform specific data models, etc. In addition we will consider model-driven development that provides abstraction to the approach. Such platform independent visualization could be used by any CASE tool to transform its models into desired output.

References

[1] R. Kennard and J. Leaney, "Towards a general purpose architecture for ui generation," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1896 – 1906, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210001597

[2] R. Kennard and S. Robert, "Application of software mining to automatic user interface generation," in *SoMeT'08*, 2008, pp. 244–254. [Online]. Available: ttp://dx.doi.org/10.3233/978-1-58603-916-5-244

[3] R. Pawson and R. Matthews, "Naked objects: a technique for designing more expressive systems," *SIGPLAN Not.*, vol. 36, no. 12, pp. 61–67, 2001.

[4] L. DeMichiel and M. Keith, "Jsr 220: Enterprise javabeans version 3.0. java persistence api," 2006. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[5] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase, "The java ee 5 tutorial for sun java system application server 9.1," 2010. [Online]. Available: docs.oracle.com/javaee/5/tutorial/doc/javaeetutorial5.pdf

[6] A. Torres, R. Galante, and M. S. Pimenta, "Towards a uml profile for model-driven object-relational mapping," in *SBES '09: Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 94–103.

[7] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[8] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," in *In ECOOP'97-Object-Oriented Programming, 11th European Conference*, vol. 1241. Springer, June 1997, pp. 220–242.

[9] K. E. Kelly, J. J. Kratky, S. Speicher, K. Wells, and G. Chia, "Ibm xml forms generator," 2006. [Online]. Available: http://www.alphaworks.ibm.com/tech/xfg

[10] H. Ceylan, E. Karaca, A. Zerr, and A. Moshchenikov, "Xml window toolkit," 2011. [Online]. Available: http://wiki.eclipse.org/E4/XWT

[11] M. Birbeck, "Xforms implementations, w3c xforms group wiki," 2007. [Online]. Available: http://www.w3.org/MarkUp/Forms/wiki/XForms%5FImplementations

[12] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using uml 2.0: Promises and pitfalls," *Computer*, vol. 39, no. 2, p. 59, 2006.

[13] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[14] J.-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front, "A survey of model driven engineering tools for user interface design," in *Proceedings of the 6th international conference on Task models and diagrams for user interface design*, ser. TAMODIA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 84–97. [Online]. Available: http://portal.acm.org/citation.cfm?id=1782434.1782446

[15] J. W. Jespersen and J. Linvald, "Investigating user interface engineering in the model driven architecture," in *In Proceedings of the Interact 2003 Workshop on Software Engineering and HCI. IFIP*. IFIP Press, September 2003.

[16] T. Cerny and E. Song, "A profile approach to using uml models for rich form generation," *Information Science and Applications (ICISA), 2010 International Conference on*, pp. 1–8, apr. 2010. [Online]. Available: http://dx.doi.org/10.1109/ICISA.2010.5480315

[17] T. Cerny and E. Song, "Uml-based enhanced rich form generation," in *Proceedings of the 2011 Research in Applied Computation Symposium (RACS 2011)*, November 2011, pp. 192–199.

[18] T. Cerny, V. Chalupa, L. Rychtecky, and T. Linhart, "Machine-driven code inspection to reduce restated information," in *Lecture Notes in Information Technology*, 2012, accepted for publication.

[19] R. Kennard, E. Edmonds, and J. Leaney, "Separation anxiety: stresses of developing a modern day separable user interface," in *Proceedings of the 2nd conference on Human System Interactions*, ser. HSI'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 225–232. [Online]. Available: http://portal.acm.org/citation.cfm?id=1689359.1689399

[20] T. Cerny and M. J. Donahoo, "Formbuilder: A novel approach to deal with view development and maintenance," in *In SofSem 2011 Proceedings of Student Research Forum*. OKAT, January 2011, pp. 16–34.

[21] J.-M. Oh, Y. S. Lee, and N. Moon, "Towards cultural user interface generator principles," in *Proceedings of the 2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*, ser. MUE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 143–148. [Online]. Available: http://dx.doi.org/10.1109/MUE.2011.37

[22] R. Kennard and J. Leaney, "Is there convergence in the field of ui generation?" *J. Syst. Softw.*, vol. 84, pp. 2079–2087, December 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.034

[23] T. Cerny, V. Chalupa, and J. Donahoo, "Impact of user interface generation on maintenance," in *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE 2012)*, 2012, accepted for publication.

[24] "Unified code count," 2012. [Online]. Available: http://sunset.usc.edu/research/CODECOUNT/