Impact of User Interface Generation on Maintenance

Tomas Cerny
dept. Computer Science and Engineering
Czech Technical University,
Charles Square 13, Prague, Czech Republic
Email: tomas.cerny@fel.cvut.cz

Vaclav Chalupa
dept. Software Engineering
Czech Technical University,
Charles Square 13, Prague, Czech Republic
Email: chaluva2@fit.cvut.cz

Michael J. Donahoo
dept. Computer Science,
Baylor University,
Waco, TX, USA
Email: jeff_donahoo@baylor.edu

***Abstract*—User interface development and maintenance is one of the most time consuming parts of software application development. User interfaces provide numerous graphical visualizations of user data, these are often influenced by multiple factors such as available interface elements, data constraints, layouts, user operating devices or user rights. The complexity of the development and maintenance raises from duplicated and restated information. Duplication occurs multiple times in the application, for example data model already contains data constraints and these are restated in the interface. Not only information but also decisions are duplicated when an interface element is selected and bound to a particular data field. Machine driven code-inspection and its transformation to user interface brings the way to address complex efforts related to user interface. In this paper we present code-inspection approach to automate user interface development and maintenance, we also provide a case study that compares the approach with manual development.**

***Keywords*—Maintenance, Code-inspection, UI generation**

I. Introduction

According to Kennard [1] around 48% of total application code and 50% of application time is devoted to implementing user interfaces (UIs). This becomes even higher when we pay efforts to provide highly usable UI of the application. At the present time, it is common that applications provide rich UIs, this means that the interface is more interactive. For example in case of rich UI forms they integrate validation or contextual help to assist user with interaction. The situation is even more complicated with web applications where the UI part is implemented in different (often not type-safe) language from the rest of the application. The common issue shared among different projects is duplicated and restated information or decision in multiple parts of the application. This can be seen for example in rich forms [2] that serve as the communication interface between user and system. Rich form consists of elements that are bound to data entity fields. Each form field is of a given type such as text, selection box, number, date and so on, it is selected based on the field properties. As next, each form field contains constraints such as maximum length, allowed format, minimum value etc, to support direct validation. Second of all, multiple variations of rich forms may exist for the same data entity, this might be related to different use cases, wizards or user rights. Third of all, the user interface may differ for different users, based on their cultural expectation, device capabilities or settings. Example life-cycle for a rich form design can be seen in Fig. 1. Manual development of user interface must capture multiple concerns at the same time, this results in complex source code that is hard to read and thus hard to maintain. It becomes even worse when multiple variations of a UI component are required for the same data. Developer must either include conditional blocks in the component or copy the component and modify it. Having to develop complex UI components is one task, but after that we must consider its maintenance. For example, data field constraint changes, new data field is added, new classes and associations exist, changes are necessary to all date UI elements in the application or changes of user rights are needed. Maintenance tasks of the described UI are error-prone and complex because the information it contains is duplicated or it spreads over multiple places in the application.

In this paper we suggest that UI components that introduce restated information or duplicated decision should not be developed manually, but rather should be generated. The already captured information are machine-inspected and transformed into UI. The transformation uses user-defined templates, which are used to decorate and extend the inspected information. This allows the developer to produce any kind of output and at the same time to maintain various concerns separately. It is also possible to generate multiple versions of UI for the same information to serve different scenarios. All possible inconsistencies among subsystems that may occur in manual development are avoided.

This paper is organized as follows in Section II we describe the approach in more details. A case study showing benefits of the approach is provided in Section III. In Section IV we discuss related work. The Section V concludes.

II. Code-inspection Approach

The aim of our approach is not to generate the entire UI part, but rather components that are often complex and introduce restated information. This means that for example page definitions, navigation among pages or composition of various UI blocks such as menus, buttons or control actions are left on the developer. The rational behind this is that these UI parts are not replicas but novel information. Generated UI components are accessible to the developer as any other component. They have settings to customize its use for given conditions. Example of the generated parts are collections or detailed reports, editable and searchable forms or any kind of data views. In the simplest case a data entity defined by a data

978-1-4673-0089-6/12/$26.00 ©2012 IEEE

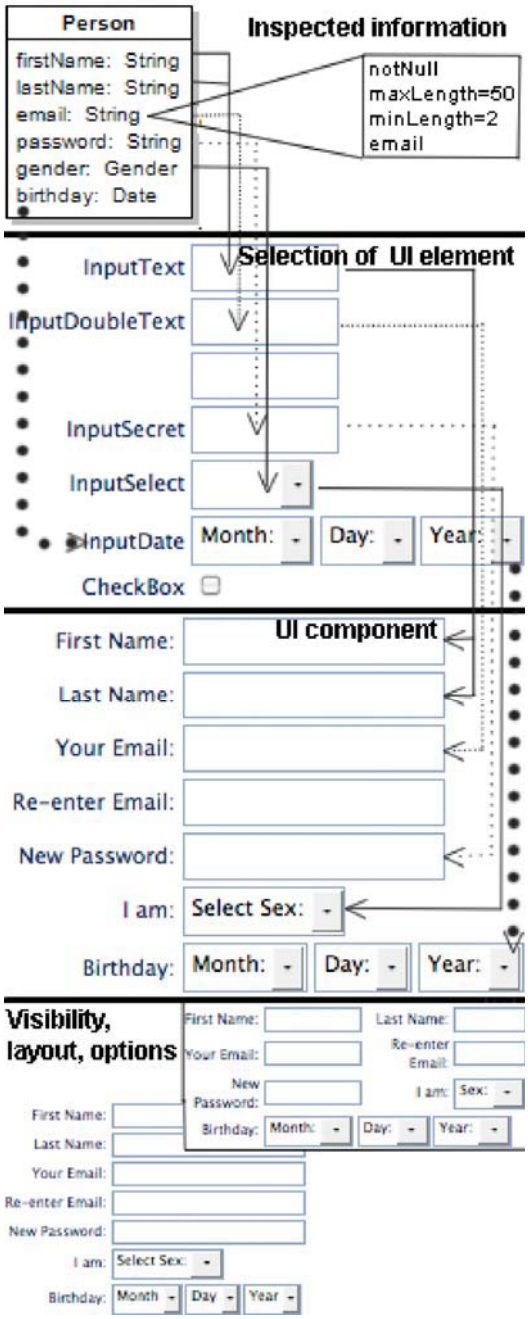

Fig. 1. Example rich form development : composition of UI elements binding to specific data class

model is transformed to a plain form. In general, this entity can be used in UI in multiple mutations (as shown in Fig. 1). It can be shown with additional entities or as a field subset in wizards. All variations can be composed from generated fragments.

The life-cycle of the approach is shown in Fig. 2, it starts with source code inspection, which builds information meta-model. Meta-model is then used by the transformation. Each meta-model entry is used to find an appropriate UI mapping.

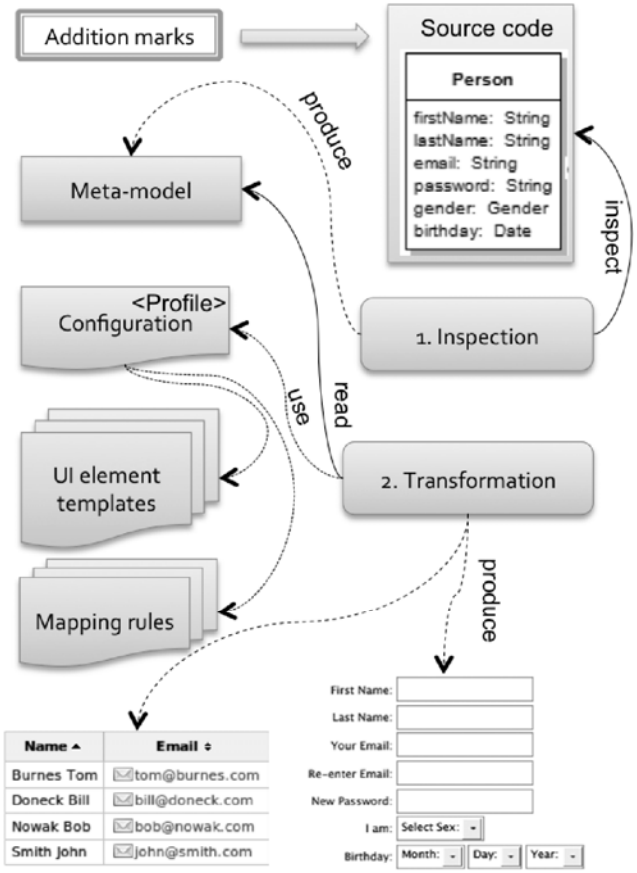

Fig. 2. Inspection and transformation schema

Once the proper template is determined, the meta-model entry properties are applied in it. Generated UI component is produced as a composite of selected and interpreted entries.

The main source of information to inspect is application data model. Data models of nowadays applications contain object-relational mapping marks [3] [4] and validation constraints [5]. These should be inspected as well, because as shown in [2] they apply in rich forms and also determine selection of appropriate UI elements. Similar to object-relational mapping and validation, addition marks can be defined [6] to determine the purpose of the field. The meta-model is built for each data entity, its fields and its constraint. Furthermore, the meta-model can be dynamically extended by the designer. The transformation part uses the meta-model to find an appropriate transformation template to use. A configurable mapping of meta-data to templates exists, this uses field meta-information to define its rules. For example when a field has length exceeding 200 characters and its data type is textual then template “A” is used, or when a textual field is annotated with email then a template “B” is used, etc. Template “B” contains UI information that are expected to bind to an entity field, which captures email. The template can use any meta-information that can occur in the meta-model. Once the template is selected it is interpreted and all field meta-information are embedded

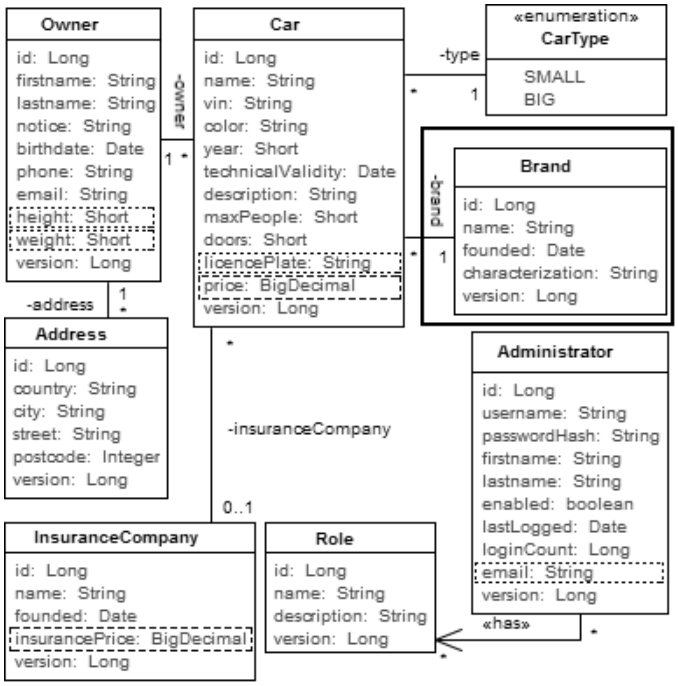

Fig. 3. Case study data model

into the template. Once the whole entity is transformed an new component is built.

The above process considers different configuration profiles with different mappings, templates, addition data model marks, user rules, or different output layouts. With the above properties it is possible to generate any kind of UI fragment for any purpose. In order to support wizards all generated components provide component settings to disable/enable fields, which allows conditional selection and other operations. Furthermore, this approach is capable to generate any type of output not only UI. Inspected information can be transformed to XSL, XML, HTML, SQL, etc.

III. Case Study

In this chapter we compare the approach with manual development and consider its impact on maintenance. We built a tool that implements our approach as described in Section II and use it for the evaluation [7]. For the study we use J2EE application implemented in framework Seam. The application data model is shown in Fig. 3. It distinguishes 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 which UI components are used for which entity, it also shows functionality provided to a given user role.

The aim of our evaluation is to show that the approach is suitable for applications that are normally developed manually. The expected benefits are reduced coupling [8], duplication and maintenance. To recognize coupling we look at how many files of the application must be modified (MoF) when the data model changes. From the maintenance perspective we look at

TABLE I
UI components generated from the data model

Class / UI fragment	Search	Edit	Detail	Passwd. change	List	Assoc. list	Inverse Assoc.
Administrator	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 fragment	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

how many logical lines of code (LOC) must be modified with the data model change. As next we consider duplication of security conditions (SeC) in UI components related to changes in data model. To evaluate the significance of our approach we apply the above metrics on comparison of manual development and our approach.

The UI part of the manually developed application consists of **1052 LOC** and **29 SeC**. The impact of a change to the data model on UI components can be seen from Table I, it may invoke up to 6 changes in binding UI components. Once we apply the proposed approach, the dependency reduces to the mapping rules and transformation templates. With the approach the UI part of the application consists of **709 LOC**. These lines are for mapping rules, templates and data model marks, and **10 SeC** in the view. We should also consider that mapping rules are independent of software application and thus can be reused among projects or loaded from an external library. If we do not consider these rules as part of the application then the total **LOC** reduces to **225**.

As next, we consider common maintenance scenarios. We consider addition of new fields (a,b), new class (c), changes in field order (d) and a new aspect that should be reflected in all fields (e). Each scenario is applied to our case study from Fig. 3, all changes are shown in dashed, dotted and solid boxes. Considered scenarios are evaluated by our metrics for manual development and automated approach in Table II. Clear benefits of the approach are significant reduction of LOC, reduction of files that must be modified (MoF) and view SeC. Our approach centralizes UI element definitions, this makes it easy to apply global changes (internationalization, help, etc.) to existing UI components. For such case only one change is needed in centralized location (template), in manual development such change would require to search through the whole application for given fragments and apply the change per each its use.

TABLE II
EVALUATION OF THE MAINTENANCE AND CREATION PERSPECTIVE

a. New class field with existing UI widget

In this scenario new fields are added to classes. Existing UI widgets are reused. We add email to the *Administrator*, height and weight to the *Owner* and licensePlate to the *Car* (see dotted box in Fig. 3).

manual aprch.	12 MoF, 126 LOC, 2 new SeC
automated aprch.	3 MoF, 6 LOC, 0 SeC (SeC - mapping rules)

b. New class field with new UI widget

New fields are added to class. New UI widgets are developed. We add price to the *Car*, and licensePrice to the *InsuranceCompany* (see dashed box in Fig. 3).

manual aprch.	8 MoF, 51 LOC, 4 SeC
automated aprch.	8 MoF, 48 LOC, 0 SeC (mapping rules/templates)

c. New class in the data model

New class *Brand* associated to *Car*, 5 new fields in the class (see solid box in Fig. 3). New search, edit and detail forms, list and association list.

manual aprch.	7 MoF, 121 LOC, 4 SeC
automated aprch.	9 MoF, 41 LOC, 1 SeC (mapping rules - class)

d. Change of field order in the UI fragment

Different order of fields for *Car* form.

manual aprch.	2 MoF, 54 LOC
automated aprch.	1 MoF, 2 LOC (modified join points)

e. Internationalization support

Application support for two languages via text resources

manual aprch.	29 MoF, 213 LOC
automated aprch.	25 MoF, 32 LOC (template modification)

IV. RELATED WORK

Alternative approaches to simplify user interface maintenance can be found. We could use aspect-oriented design [9], model-driven development [2] [6] [10] [11] [12] [13] or user interface generators [1] [14] [15]. A successful approach should be easy to adopt by industry or by legacy projects [16].

Aspect-oriented design [9] addresses code duplication that cannot be addressed neither by objects nor by design patterns. This approach suggests that an aspect captured by the application is managed separately from other aspects. Aspects are then weaved together through aspect weaver. Each aspect contains join points, which are the connection points to other aspects. This approach is similar to our approach, the difference is the code-inspection that is not part of aspect-approach. Our approach is easy to adopt by legacies because existing code can be applied.

Model-driven development (MDD) and model-driven architecture [10] uses models such as UML models to capture all information. The application source code is then transformed from models. This brings platform independence on one side, but on the other side all information must be captured by model. Extensions to UML class diagram to capture UI specific information is provided by [2]. Other approaches [11] [12] [13] look at human-computer interaction (HCI) and model-driven support. In [13] authors argue that there exists a large gap between HCI and MDD, they also discuss whether capturing of addition information in design models should be considered as layer violation, in conclusion they state that HCI concerns cannot be described independently from other concerns. Similar rational is brought by Torres et al. [17], who extends UML models with profile for object-relational mapping. Addition of behavioral models is investigated by [11], behavioral models can determine page flow, and use case navigation. Our approach does not inspect behavioral aspects, because these are not replicated in the application. Model-driven approach requires designer to re-implement legacy application into models. This could be automated by our approach where an existing application is inspected and gathered information are transformed into XML model that can be interpreted by a CASE tool.

UI generators can be classified [18] in three categories into interactive graphical specification tools, model-based generation tools and language-based tools. The first two categories, in which can be placed graphical drag and drop tools or XML model tools, in fact need to restate information which does not reduce maintenance efforts. Language-based tool can derive information already captured in source code. There exist few language-based frameworks. The first one is Naked objects [14], it was developed a an doctoral thesis and applied for information systems of Irish government. In Naked objects, addition marks are added to the source code in the lowest layers. These marks are considered for component selection in the generation process. FormBuilder [15] inspects existing applications and generates UI components. It allows to reuse existing information captured by application, it can capture addition information in XML or by addition marks in source code. MetaWidget [1] similar to FormBuilder generates UI components, although it aims to generate the entire UI part with control. The main drawback is that MetaWidget provides limited amount of UI components to use, which degrades expressiveness and corporate culture.

V. CONCLUSION

In this paper we suggest that machine driven code-inspection and transformation can simplify user interface development and reduce maintenance efforts. Rather than to repeat the same decisions on UI element selection, mapping rules are defined and reused over the project. This allows to centralize UI element definitions to one location, any future change to the element does not need to be replicated in the application. This approach allows to separate concerns and manage them independent of each other. In a study we provide is shown how the approach reduces lines of source code of UI part of studied application. Reduction of maintenance efforts will grow with the size of project, the amount of UI elements used in an application does not grow with its size, but the UI elements are normally replicated and this provides potential for reduction.

In future work we want to apply this approach on reverse engineering of existing projects and their transformation to UML or other platforms. As next, we plan to apply our approach to a large legacy application and measure the impact on maintenance reduction.

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] T. Cerny and E. Song, "A profile approach to using uml models for rich form generation," in *Information Science and Applications (ICISA), 2010 International Conference on*, 2010, pp. 1 –8.

[3] 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

[4] "Jboss hibernate: Java persistence framework," 2010. [Online]. Available: http://www.hibernate.org

[5] "Hibernate validator, open-source validation library for hibernate framework," 2010. [Online]. Available: http://www.hibernate.org/subprojects/validator.html

[6] 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.

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

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

[9] 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.

[10] 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.

[11] J.-H. Wu, S.-S. Shin, J.-L. Chien, and W. Chao, "An extended mda method for user interface modeling and transformation," in *Fifteenth European Conference on Information Systems, IHsieh M-C (2007)*, Osterle H, Schelp J, Winter R eds., 2007, pp. 1632–1642.

[12] J.-l. Perez-medina, S. Dupuy-chessa, and A. Front, "A survey of model driven engineering tools for user interface design," in *In Proc. of 6th Int. workshop on Task Models and Diagrams (TAMODIA'2007)*. Berlin: Springer, 7-9 Nov. 2007, pp. 84–97.

[13] 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*. Press, 2003.

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

[15] 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.

[16] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000, pp. 35–46.

[17] A. Torres, R. Galante, and M. S. Pimenta, "Towards a uml profile for model-driven object-relational mapping," in *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, ser. SBES '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1109/SBES.2009.22

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