Deriving Default User Interfaces from Domain Contracts

Conference Paper · June 2007

CITATIONS
2

READS
941

1 author:

António Miguel Rosado da Cruz
Instituto Politécnico de Viana do Castelo
62 PUBLICATIONS **685** CITATIONS

SEE PROFILE

Deriving Default User Interfaces from Domain Contracts

António Miguel Rosado da Cruz [1]

antonio.cruz@acm.org

[1] E.S.T.G., I.P. Viana do Castelo, Av. do Atlântico , 4900-348 Viana do Castelo, Portugal

Abstract: This paper presents an approach for deriving default user interfaces from contracts of the domain model. The derived UI is intended to interact with the application's business logic (domain) model prototype, which is represented in the form of UML class diagrams complemented with contracts in VDM++.

Keywords: model driven development. User interface. Domain contract.

1. Introduction

Software development processes have been using, for decades, software models that help software engineers cope with complexity, when analyzing the problem domain or designing a software solution. Software models capture relevant parts of the problem and solution domains and are typically used as a means of communicating with the stakeholders (e.g.: clients, users, team members). Several software modeling methodologies have been developed. A review of some of the most popular methods and modeling techniques in software engineering, from the last decades, can be found in (Wieringa, 1998; Koch, 1999).

Most of the methodologies for modeling interactive applications use disparate views, or component models, to capture different aspects of the domain (task model, abstract and concrete presentation models or application model) (Pinheiro da Silva, 2002). It is, often, difficult to understand the whole because the views of the parts do not integrate well, i.e. it is typically hard to have an integrated view of every component models. This situation compromises the possibility of having code generated from the software models.

Model-driven development approaches, like OMG's Model Driven Architecture[1] (MDA), refocus the aim of modeling from human understandable to system

[1] "Model Driven Architecture" is a trademark of OMG.

recognizable (Kleppe et al., 2003). Model driven development processes, typically based on formal or semi-formal (with a core suitable to unambiguous interpretation) specification methods, have already achieved some capabilities for automatic generation of code for some of the code layers (e.g., data layer, business logic layer, user interface layer). Typically, the code for the data layer (e.g., database SQL script) can be fully generated from the model (Chaves, 2006), and all or part of the code for the business logic layer can also be generated for a given platform. But the user interface layer has been forgotten in the efforts to achieve automatic code generation from integrated software models. User interface model-based technologies often produce UI models that do not integrate well with the core of the application. The user interface layer has to cope with complexities both from the application core and the user.

The main contribution of this paper is to provide a model-driven approach to the generation of default user interfaces, starting from the system's business logic model represented as UML class diagram along with class contracts in VDM++. The class of system models considered is restricted to data oriented (business) applications, and the approach allows the generation of form based UIs.

The next section presents some concepts of modeling systems through contracts and model-driven development. Section 3 addresses the specification of contracts in VDM++ notation. Section 4 addresses the problem of deriving form-based user interfaces from domain components' contracts. Section 5 concludes the paper and sketches some directions for future work.

2. System modeling through contracts

Design by Contract[2] (DbC), aims at developing software that is trustable, in which people can rely on. In DbC, a contract is a means by which two elements agree in an explicit roster of mutual obligation and benefits (Schoeller et al., 2006; Meyer, 2006; Artima, 2004a, Artima, 2004b). In an object-oriented setting, components and classes act as providers and clients of services. Through a contract, two components or classes agree in a protocol for communicating, that is, which methods are exported (publicly visible) by the intervening classes and what type are its parameters and results.

A contract not only specifies the protocol for instances of classes or components' communication, but also specifies, as will be seen in section 3, what conditions have to be met in order for the communication to happen, that is, what conditions must the caller (client) assure in order to be allowable to call a given method and

---

2 "Design by Contract" is a trademark of Eiffel Software.

what conditions must the callee (provider) guarantee that hold after the method is returned.

A contract may also specify conditions that must always hold, as invariant conditions that are imposed to the state of a class' instances.

DbC can be viewed as a means to develop a system's design model and, as it is abstractly but formally defined, the room for ambiguity is strongly reduced, meaning that it may be machine understandable and that it may be used for model transformation activities. This way, DbC can be an approach to model driven development.

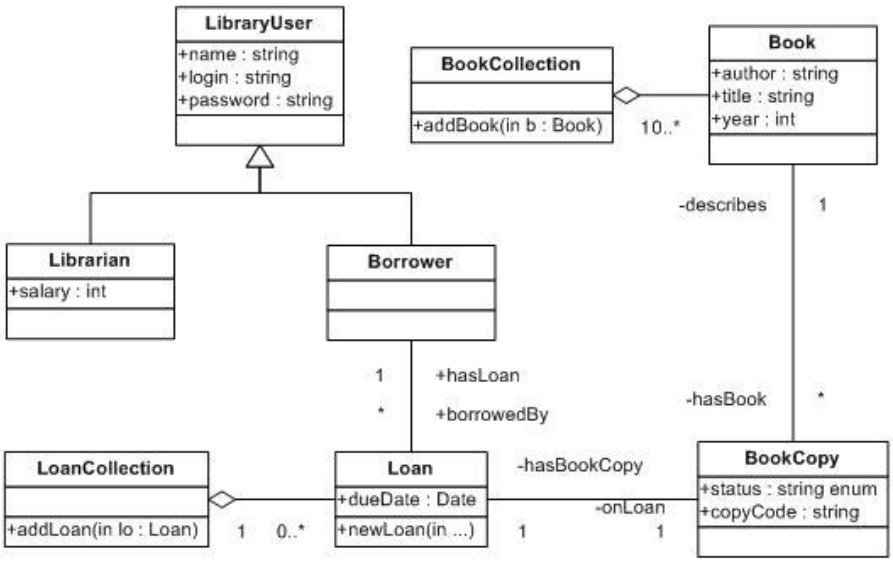


Figure 1 – Domain Model of the Library System [adapted from (Pinheiro da Silva, 2002)]

3. Contract Specification in VDM++

In order to present the VDM notion of contract, the UML class diagram of a library system case study is adapted from (Pinheiro da Silva, 2002). A UML domain model of the Library System is presented in figure 1. The domain classes' diagram is composed of classes that model objects identified in the domain analysis process.

LibraryUser, *Librarian*, *Borrower*, *Book*, *BookCollection*, *Loan*, *LoanCollection* and *BookCopy* are the «entity» classes of the Library System. The contract associated to each class is not clear in the UML class diagram. OMG provides a language for the specification of such contracts, though, namely Object Constraint Language (OCL). Nevertheless, in this paper will be used the VDM++ notation, because VDM allows, not only the specification of contracts (invariants, pre- and post-conditions), but also the full specification of each operation's functionality. This characteristic enables the use of the derived default UI to animate de model, that acts as a business logic prototype.

VDM++ is an object oriented extension to the Vienna Development Method (VDM) notation. VDM is a formal method's language for the specification of programs, developed in 1973 in the Vienna laboratory of IBM, which has been stated as an ISO standard (ISO/IEC 13817-1) in 1996 as VDM-SL (Fitzgerald & Larsen, 1998). VDM-SL enables the development of algebraic models, either purely functional or composed of modules with inner state. VDM++ adds the notion of class and all its associated concepts (inheritance, encapsulation, information hiding, etc.) to the abstraction facilities provided by VDM-SL.

By using the VdmTools© (IFAD, 1999; IFAD, 2000), it is possible to create a complete document that incorporates both textual non-formal english descriptions of the system and the VDM-SL or VDM++ model, through literate programming.

In VDM++, a class may have an inner state, purely functional methods, called functions, or non-purely functional methods, called operations. In purely functional methods, the return value depends only on the parameter values and has no side effects, while in non-purely functional methods, if they return a value, it may depend both from the parameters and the inner state of the class' instance and it may, as a side effect, update the inner state.

As referenced earlier, both in VDM-SL and in VDM++, specifications can be made at an abstraction level where the semantic of the operators is fully specified or, at a higher abstraction level where the semantic of the operators is partially specified through pre- and post-conditions. Invariants can be specified over abstract data types or over modules' inner variables or classes' instance variables, depending on whether VDM-SL or VDM++ is being used. In the second case, specifications can be called contracts.

Next is part of the VDM++ contract for class *Book*:

```
class Book

  instance variables
    private author : char*;
    public title : char*;
```

```
    public year : ℤ;
    public describes : BookCopy-set;
  inv (self.year > 1900) ∧ (self.year < 2050)
```

```
  operations

    public
    get-author () result : char*
      pre  true
      post result = self.author;

    ...
    public
    set-author (x : char*)
      pre true
      post self.author = x ;
```

```
end Book
```

The VDM++ contract for class *BookCollection*:

```
class BookCollection

  instance variables
    private bookCollection : Book-set;
  inv ∀b1,b2 ∈ self.bookCollection ·
           b1.get-author() ≠ b2.get-author() ∨
           b1.get-title() ≠ b2.get-title() ∨
           b1.get-year() ≠ b2.get-year()
```

```
  operations

    public
    addBook (b : Book)
      pre ¬∃x∈ self.bookCollection ·
            x.get-author() = b.get-author() ∧
            x.get-title() = b.get-title() ∧
            x.get-year() = b.get-year()
      post ∃x∈ self.bookCollection ·
            x.get-author() = b.get-author() ∧
            x.get-title() = b.get-title() ∧
            x.get-year() = b.get-year()
```

```
end BookCollection
```

And, the VDM++ contract for class *BookCopy*:

```
class BookCopy

  instance variables
    public status : char*;
    public copyCode : char*;
  inv  self.status ∈ {“new”, “good”, “bad”}

end BookCopy
```

The contracts specify what invariant conditions must always hold in each of the classes, what (pre-) conditions must hold for each one of the methods and what (post-) conditions must be verified after each method returns. For example, in class *Book* an invariant is defined that states that the value for *year* must be an integer greater than 1900 and less that 2050. Pre-condition of method *addBook* in class *BookCollection* states that the book being added to the book collection must not previously exist in the book collection.

4. UI derivation approach

For being able to systematize the generation of a user interface for a given system model, a set of mapping rules has been established for deriving a default user interface form from each of the classes, and a user interface navigation map from the associations between the classes. Table 1 shows the mapping rules for a simple class, like *BookCopy*, and for a class with an aggregated class, like *BookCollection*.

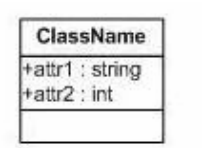
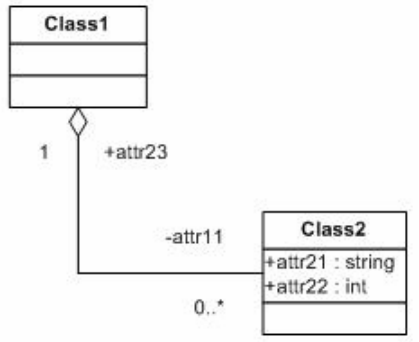
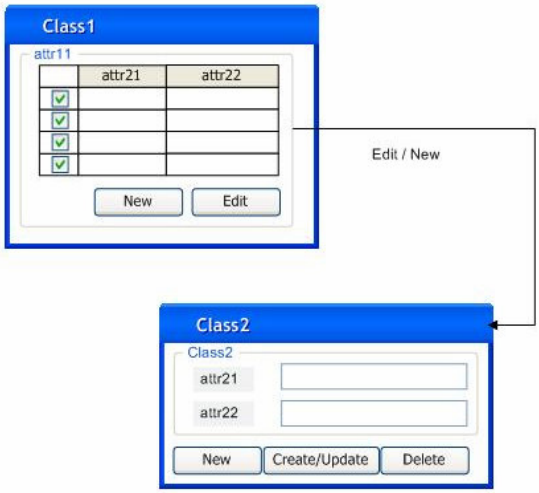
Figure 2 illustrates the rules for mapping the UML class model to forms-based UI, for the three classes specified earlier for the library system case study.

Each class gives origin to a form with a data entry field (e.g.: text box, or any other field for entering text), for each class’ public attribute and/or each get- /set- methods pair. A get- method with no corresponding set- method originates a read-only output field (e.g.: non-editable text box). Each class also originates a set of action fields (e.g.: buttons), namely New, Create/Update and Delete, except those classes that only have one field that aggregates instances from another class. An aggregation of classes like the one between classes *BookCollection* and *Book* (*BookCollection* has an attribute which type is *set of Book*) originates a collection (multiple items) output field (e.g.: data grid) with an Edit action field (e.g.: button) for each object in that list.

In order to better understand the UI generated by the mapping process, figure 2, shows a very concrete UI model. The widgets presented in this concretization of the UI are only an example. The process shall generate not a concrete UI but rather an

abstract one using, for example, a notation for abstract prototypes like the one presented in (Constantine et al., 2000).

Table 1 – Some of the Mapping Rules

UML Class diagram	Forms-based GUI
	
	

Each class invariant may also originate a validation rule that shall run after each operation has finished. From each method's pre-condition a validation rule may also be derived, that shall run before effectively running the operation's code. Indeed, each method's code shall only run if the validation rule succeeds, i.e. if the operation's pre-condition holds. Validation rules that only relate fields from the UI form, like the one derived from the Book's invariant, may run at the UI layer. Rules that relate fields from the form with data from other instances, of the same or any other class, shall be coded as services from the business logic layer, and invoked from the UI layer. Examples of this last scenery are the validation rules that are derived from the *BookCollection*'s invariant and from the *BookCollection*'s *addBook()* method.

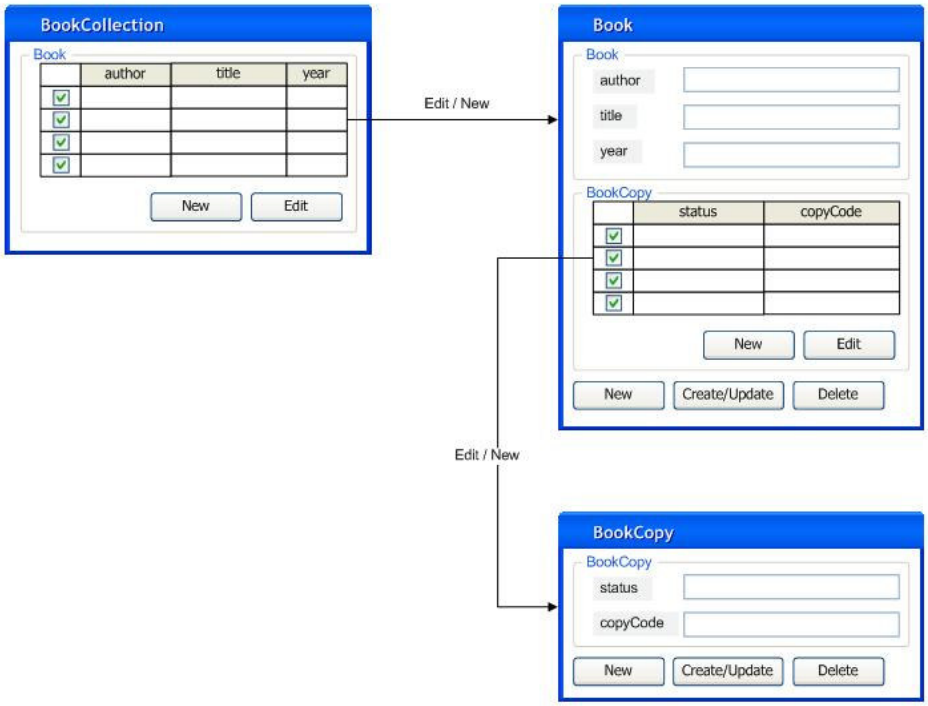


Figure 2 – User Interface abstract model for the classes *BookCollection*, *Book* and *BookCopy*.

5. Conclusions and future work

Developing software systems through the specification of contracts is a model-driven development technique, in which the system is modeled using object oriented methods complemented with a contract for each class. The contract involves the specification of logic conditions that must always hold during the lifetime of the class' instances and also of conditions that must be verified in order to be "legal" to invoke a given method and conditions that the method must guarantee that hold after returning.

VDM++ contracts could totally ignore the corresponding UML class diagram, since they can have more information. Nevertheless, UML class diagrams are easier to share with other stakeholders than VDM++ contracts. The VdmTools© allow the

round trip between Rational Rose© UML class diagrams and VDM++ specifications. Obviously, some information may be lost in the transformation from VDM++ specifications onto a UML class diagram.

Nonetheless, some of the invariants that guarantee the integrity of the system or component are partially implicit in the UML class diagram. For example, an instance of a *BookCopy* referenced by a *Loan*, through attribute *hasBookCopy* must be a member of set attribute *describes* of an instance of class *Book*. This is no more than a foreign key constraint, implicit in the class diagram. In VDM++ , if one wants to specify a clear and explicit contract that crosscuts several classes, one has to create a structure (class) that aggregates everything else, and define the crosscutting invariant in that top class. In (Balzer et al., 2005; Zhang et al., 2005) a study has been made of aspect contracts for contractualize crosscutting concerns, such as invariants that span several classes. From the classes' contracts, a set of mapping rules has been specified, in order to be able to systematize the process of generating form-based user interfaces.

From the contracts' invariants and pre-conditions a set of validation rules may be derived for increasing the usability of the generated user interface. Further study is needed to formally specify a mapping table from invariants and pre-conditions to validation rules at code level.

Also, the specification of the generated UI through Abstract Prototypes may not be sufficient for enabling an automation of the generation process. The abstract prototype is only another model in the model transformation chain created by the model-driven development approach. A default and customizable concrete UI should also be generated, either driven by a usability knowledge based repository, by a given user model or historic usage model or by any other means.

Several works (Kuo et al., 2005; Forbrig et al., 2004) present UI generation approaches for executable XML based user interfaces, but their start point is a declarative model of the user interface, not a model of the system's business logic. Declarative User Interface Models (UIMs) provide an abstract description of the UI that can be reused and can be refined to more concrete models. UIMs can be found at different levels of abstraction during the UI design process. UIM provide an infrastructure for allowing automated tasks in the UI design and implementation processes. In (Pinheiro da Silva, 2000) a comparative study between 14 model-based user interface development environments is presented.

As referenced earlier, the approach presented herein starts from the (business logic) application model and obtains a default user interface abstract presentation model. Also a default task model is implicit in the derived navigation map. The development of a tool that automates the presented UI derivation approach would help to test and refine the mapping rules between the application model (class contracts) and the UI model generated. Also, a default concrete presentation model

along with the data validation rules, generated from the classes' invariants and the methods' pre-conditions, is something that would be interesting to try to achieve.

The UI derivation approach makes little use of the specified methods. Is essentially uses the data structures (types, classes and its associations), with all action fields (e.g.: buttons), constructive operations. In fact, all UI operations serve as constructors of the data structures (e.g.: Create, Update, Delete). It would be interesting to have, in the future, action fields for business logic operations (e.g.: *calculateFineToPay(int dueDate)*, method to calculate de fine that has to be paid for delaying the devolution of borrowed books).

The derived UI allows the easy UI prototyping of the contracts that may be used to validate and refine the contracts themselves, by the client or end user, before advancing in the model transformation process towards a complete and executable application.

Being based in the same complete application model that is used for deriving the data layer and the business logic layer, the presented approach for deriving a user interface layer, promotes a complete and integrated model of the system being developed.

References

Artima (2004a). Contract-driven Development - A conversation with Bertrand Meyer, Part III - by Bill Venners. www.artima.com/intv/contestP.html. *Visited in 18th December 2006.*

Artima (2004b). Design by Contract - A conversation with Bertrand Meyer, Part II – by Bill Venners. www.artima.com/intv/contractsP.html. *Visited in 18th December 2006.*

Balzer, S., Eugster, P. & Meyer, B. (2005). Can aspects implement contracts? In: *Proceedings of RISE 2005 (Rapid Implementation of Software Engineering techniques)*, Heraklion, Greece (September 2005) to appear as Springer *Lecture Notes in Computer Science.*

Chaves, T.M.L. (2006). VooDooM: Support for understanding and re-engineering of VDMSL specifications. Master's thesis, University of Minho.

Constantine, L., Windl, H., Noble, J. and Lockwood, L. (2000). From abstraction to realization in user interface designs: Abstract prototypes based on canonical abstract components. Working Paper.

Fitzgerald, J. & Larsen, P. (1998). Modelling Systems. Practical Tools and Techniques in Software Development. Cambridge University Press.

Forbrig, P., Dittmar, A., Reichart, D. & Sinnig, D. (2004). From models to interactive systems tool support and XIML. *In: MBUI*.

IFAD (2000). The IFAD VDM-SL Language. Revised for V3.6.

IFAD (1999). The IFAD VDM++ Language. Revised for V6.3.0a.

Kleppe, A., Warmer, J. & Wim B. (2003). MDA Explained: The Model Driven Architecture – Practice and Promise. *Addison-Wesley Professional*.

Koch, N. (1999) *"A Comparative Study of Methods for Hypermedia Development"*. Technical Report 9905, Ludwig-Maximilians-Universität München, November 1999.

Kuo, Y., Shih, N., Tseng, L. & Hu, H.C. (2005). Forms-XML: generating form-based user interfaces for XML vocabularies. Advances in Web-Age Information Management. 6th International Conference, WAIM 2005. Proceedings (*Lecture Notes in Computer Science* Vol.3739) 925 –

Meyer, B. (2006). Dependable Software. *Lecture Notes in Computer Science*. To appear in "*Dependable Systems: Software, Computing"*, Networks. eds. Jürg Kohlas, Bertrand Meyer, Andr´e Schiper. Springer-Verlag.

Pinheiro da Silva, P. (2000). User interface declarative models and development environments: A survey. *Lecture Notes in Computer Science* 1946, 207–226

Pinheiro da Silva, P. (2002). Object Modelling of Interactive Systems: The UML_i Approach. PhD thesis, Faculty of Science and Engineering, University of Manchester.

Schoeller, B., Widmer, T. & Meyer, B. (2006). Making specifications complete through models. *Springer-Verlag Lecture Notes in Computer Science*. To appear in "*Architecting Systems with Trustworthy Components*".

Wieringa, R. (1998). A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv., 30*(4), 459–527.

Zhang, J., Gray, J. & Lin, Y. (2005). A Model-driven approach to enforce crosscuting assertion checking. *Workshop on Modeling and Analysis of Concerns in Software* (MACS 2005). Copyright 2005 ACM.