Recommendations for MDA CASE Tools for Efforts Reducing in Software Modeling

José Eduardo Belix
Sérgio Martins Fernandes
Selma Shin Shimuzu Melnikoff
Edison Spina

The Department of Computing and Digital Systems Engineering (PCS)
University of São Paulo - Polytechnic School
Av Professor Luciano Gualberto, travessa 3, nº158 - Sala C2-42.
Cidade Universitária - São Paulo - SP
CEP: 05508-900
Brazil

Abstract: - The OMG's MDA (Model Driven Architecture) defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform [1]. The MDA proposes automatic code generation from UML models. The MDA process is divided in 3 models: the PIM (Platform Independent Model), a system's model that has no platform considerations. The PSM (Plataform Specific Model) is generated from PIM and considers the platform's technological issues, and the source code, generated from PSM.
For the model-driven software development vision to become reality, CASE Tools must support this automation [2].
This paper focuses on defining recommendations for these MDA CASE Tools. The objective is to reduce the necessary efforts to represent the PIM model, by the adoption of predefined solutions for the software to be generated.

Key-Words: - MDA, MDD, CASE TOOLS, Generative Programming, Software Architecture.

1 Introduction

The MDA states that code must be generated from UML Model, in a process that begins with the PIM. This initial model is refined and transformed into the PSM, which is transformed into source code.
This paper proposes that the MDA CASE Tools must be elaborated considering the commonalities between the different products [3], which raises two points:

- Its is advantageous to the developer that the MDA Case tools provide solutions (specifications or implemented functionalities) that could aid to define/implement the commonalities of these applications, reducing the efforts in modeling software;
- Considering that these provided solutions establish some parts of the software design, the developer must direct his efforts to what really needs design definitions.

1.1 Taking Decisions

Conscientiously or not, a set of decisions is always taken in a software development effort. These decisions can deal with high-level software issues, like the choice of an architecture pattern; or can deal with more specific issues, like data persistence.
These decisions represent different solution characteristics. An architecture pattern, for example, "describes a particular recurring design problem that arises in specific contexts, and presents a well-proven generic scheme for its solution" [4]. The pattern is a solution, but provides no implementation.
On the other hand, decisions for more specific issues can result in solutions with the corresponding complete implementation. The use of ADO (ActiveX Data Objects) to access persistent data in Visual Basic projects is an example.
[5] presents a taxonomy to classify these issues. The taxonomy is about abstractions, which are defined as "composed new solutions from existing ones to solve problems expect to be encountered during system refinement". The taxonomy is:

- White-box abstraction
  Provides a description of the solution, but not an implementation. It can be applied in many contexts. Patterns are examples of white-box abstractions.
- Black-box abstraction
  Provides a fully implemented solution that is completely opaque to the user. To provide a completely implemented solution, such abstraction can only be applied to problems that require that exactly solution. Features supplied by libraries or compiled languages are generally black-box abstractions.
- Gray-box abstraction
  Represents the intermediate level: provides source code for a partial or complete implementation of the solution, which can be completed or modified when needed.

2 A Practical Example

A MDA CASE Tool can be configured to generate business applications (problem domain) from a set of design decisions (solution domain). As an example of solutions that do not provide implementation (white-box abstractions), the tool must generate software based on the Layers Pattern [4] in three-tier configuration, using DAO Pattern [6] for data persistence. DAO classes implement a CRUD-based interface [7], and the Use-Case Controller pattern [8], [9] must be used to map the requirements specifications to the implementation.

2.1 Applying the Decisions

This design example will be used for the development of online retail sales software. In the Use Case "Customer Maintenance", there is a customer information consult, which selects customer for ID and validates the dealer's password (two business objects interacting to allow this consult). The representation of this behavior is:

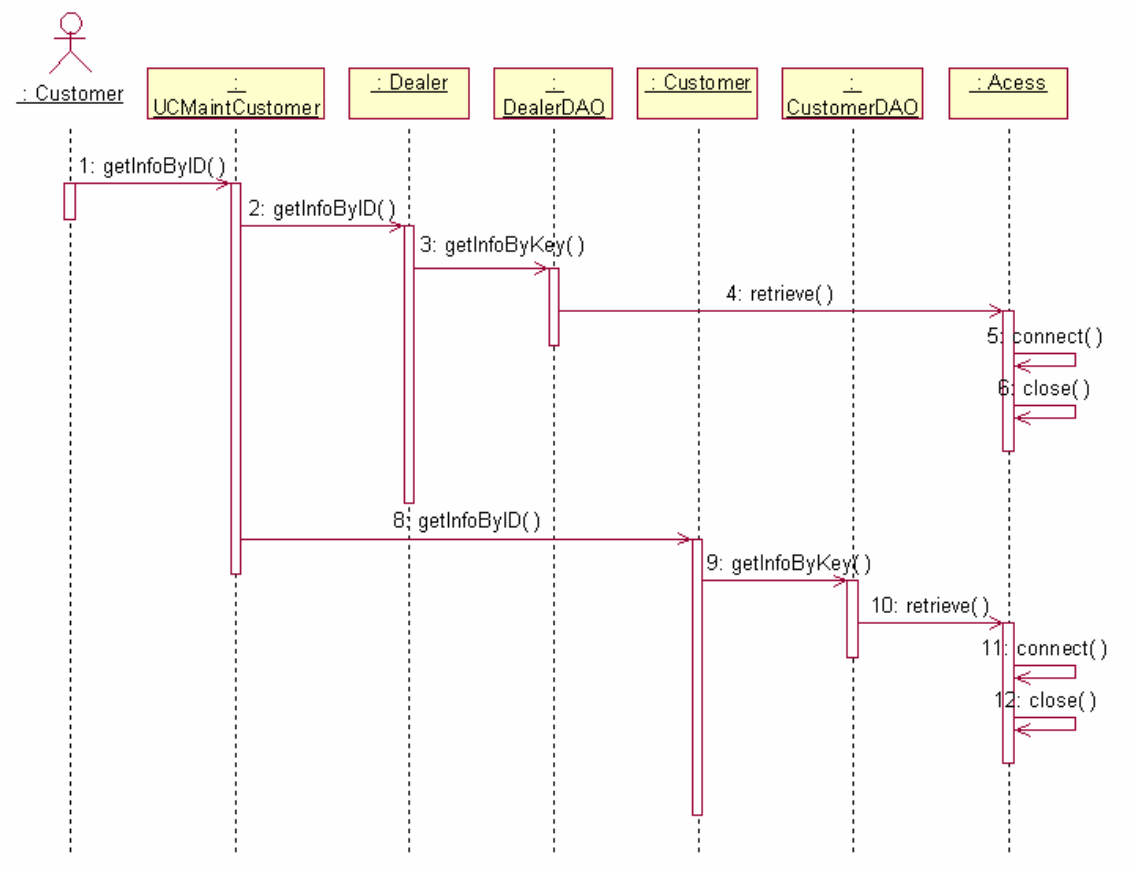

Figure 1 - A Use-Case path

This sequence diagram represents the necessary messages for this consult within the proposed design context. The DAO and ACCESS classes, for example, are consequence of the adopted decisions.

This diagram does not contain all the information for a CASE Tool to generate the necessary code. Despite being representative to the humans' cognition, a method called "getInfoByID" represents nothing to a computer. It is necessary to represent the detailed behavior of the method in UML.

Another possible way to solve this problem is applying solutions that really implement source code. Once the CRUD principles were applied, the source code for persistence method implementation could be partially written and be automatically configured for the CASE Tool to any specific class (the implementations are all very similar - this is a gray-box abstraction case). The ACCESS class (responsible for DB connection and SQL statements execution) would not demand further parameterization and could be automatically implemented. With these latter abstractions, the CASE Tool would be able to completely implement all the persistence methods. In this way, despite the fact that the software architecture remains the same, the CASE Tool user would only need to specify the following behavior:

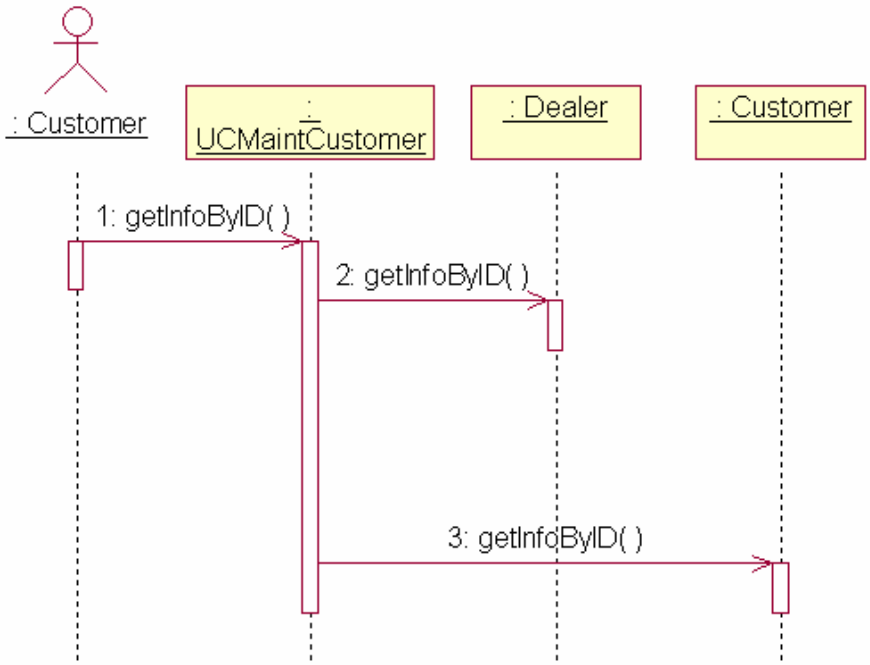

Figure 2 - The Needed Representation

In this case, the developer would only describe the Use Case to the point where he/she could really decide about the application design. There is no reason to model what is already implemented and no further consideration is required. "When deciding how to model something, determining the correct level of abstraction and detail is critical to providing something that will be of benefit to the users of the model. Generally speaking, it is better to model the artifacts of the system - those 'real life' entities that will be constructed and manipulated to produce the final product. Modeling the internals of the web server, or the details of the web browser is not going to help the designer and architects of a web application" [10].

2.2 Constraints are Useful

A MDA CASE Tool customized to work within the design decisions presented in this paper would not work when modeling a digital signal processing system, which uses the Pipes & Filters architecture pattern [11]. On the other hand, the adopted solutions greatly facilitate the business application model task, as suggested in this paper.

A possible way is that tools be elaborated with different templates, each one customized for different software development conditions. Even in business applications, it would be useful to have templates for .Net with DAO strategy, or J2EE applications with container managed persistence, etc…

3 Conclusions

This paper supports the need of MDA CASE Tools to work within the adoption of predefined solutions, which can be classified in a spectrum varying from just specification to fully implemented code. This constraints the scope of possible domain problems where tools can work, but is able to specialize the tools in a way to reduce the amount of necessary efforts to model software.

References:

[1] OMG. “Model Driven Architecture (MDA). Document Number ormsc/2001-07-01” Object Management Group, july 2001.

[2] Sendall, Shane; Kozaczynski, Wojtek. “Model Transformation: The Heart and Soul of Model-Driven Software Development” IEEE SOFTWARE, Volume20, Issue5, pg 42-45, Set/Oct 2003.

[3] Bosch, Jan. “Product-line architectures in industry: a case study”. Proceedings of the 21st international conference on Software engineering. May, 1999.

[4] Buschmann Frank; Meunier Regine; Rohnert Hans; Sommerlad Peter; Stal Michael. “Pattern-Oriented Software Architecture. A System of Patterns”. John Wiley & Sons Ltd., Chichester, UK, 1996.

[5] Greenfield Jack; Short, Keith. “Software Factories: Assembling Applications with Models, Frameworks, and Tools”. Wiley Publishing, 2004.

[6] Alur, Deepak; Crupi, John; Malks, Dan. “Core J2EE Patterns”, Sun Microsystems Press. 2001.

[7] Brandon, Daniel. “CRUD Matrices for Detailed Object Oriented Design”. Journal of Computing Sciences in Colleges. December 2002.

[8] Aguiar, Ademar; Souza, Alexandre, Pinto, Alexandre. “Use-Case Controller”. EuroPLoP 2001.Sixth European Conference on Pattern Languages of Programs. 2001.

[9] Evans, Gary. “Getting From Use Cases to Code Part 1: Use-Case Analysis”. IBM – The Rational Edge. July 2004. Available on <http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jul04/5383.pdf>.

[10] Conallen J. “Modeling Web Application Architectures with UML”. Communications of ACM, October 1999, Vol 42, nº 10, pgs 63-70.

[11] François, Alexandre R.J. “Software Architecture for Computer Vision: Beyond Pipes and Filters”. Available on <http://iris.usc.edu/~afrancoi/pdf/sacv-tr.pdf>.