Elements for the Definition of a Model of Software Engineering

Vincent Rosener and Denis Avrilionis
OneTree Technologies
Technoport – 66, rue de Luxembourg
L-4221 Esch/Alzette, Luxembourg
Tel: +352 54 55 80 820, Fax: +352 54 55 80 401
{vincent, denis}@onetreetech.com

ABSTRACT
For year now we have been working on the definition of something we could call a “model of software engineering”. Although this might sound very ambitious, this work is driven by concrete objectives and the results are applied in day to day work in the context of our consulting and software development projects. It is obviously impossible to present the full scope of our work in a few pages. We see GaMMa 2006 as an opportunity to expose our ideas and share our experience with people thinking and trying out things in the same area.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design - Methodologies
I.6.5 [Simulation and Modeling]: Model Development - Modeling methodologies


Keywords
Software construction, models, systemic theory, design science.

1. INTRODUCTION
We are software engineers. Engineering disciplines rely on models and theories. We are curious to understand what the model/theory behind software engineering is. Of course many other people have talked and worked on the subject. We do not pretend to solve the problem. Our metrics to verify our point of view is to apply it to the software we are building every day. And this seems to work nice so far.

To come to the point where we articulate our thinking as we do today, in the last ten years we: wrote hundreds of thousands of lines of code; worked in analyzing software counting millions of lines of code; designed and architected systems in the industry, finance and public administration; went through some basic readings in disciplines we feel are connected to software engineering.

We could have done more, but we have reached a point where we need to share our work with others in a constructive manner in order to move forward in our understanding of software engineering, thus this position paper to GaMMa 2006.

2. OBJECTIVES
Producing a software engineering model can be useful for many reasons, including elegance or intellectual value. Given our engineering background [1] we put the emphasis on how to use such model to write better software; ‘better’ in the sense of less subjective, repeatable, and conscious actions aiming at creating software.

Our objectives are:

- To apply effective Software Engineering, at a high quality level (e.g. in terms of correctness, reusability, flexibility) for building complex systems i.e. help software developers to create better software systems.

- To unify software engineering concepts (‘architecture’, ‘process’, ‘language’, or ‘method’) or techniques (‘OO’, ‘AOP’, ‘MDA’). Most of them have commonalities. The attempt of unification is not done for the sake of unification, but because artificial separations create ambiguity and make the construction of software more difficult.

- To benefit from work done in ‘Simulation’, ‘Systemic modeling’ (Von Bertallanffy [2], LeMoigne [3]), and the ‘Design Science’ (Simon [4]). Understand the commonalities between these fields and ‘Software Engineering’.

- To create a common ‘framework’ to handle the different activities of software construction (i.e. the so-called ‘analysis’, ‘design’, ‘implementation’, testing’), as well as other computer science fields (artificial intelligence, compilation theory, language theory) [5]. In fact, most of these activities share common concepts.

- To automate parts of this ‘model of software engineering’ whenever possible.

- To capitalize knowledge: common definition of the concepts (structural knowledge); methods to create the models, or to validate the models (behavioral knowledge).

- To define usable models of software, particularly models related to technical aspects like persistence, distribution, deployment. This enhances heavily MDA’s notion of platform.

- To capture the important knowledge of developers. Even in a heavily-tooled software environment, real and important
knowledge remains in the head of software developers. As such, some people will develop correct and reusable classes (e.g. in the context of an OO development [6]), and some will not. As pointed by Boehm [7], a factor 20 is possible between the best and the worst developer. What is the knowledge used by a ‘good’ developer? Is it possible to make such knowledge more explicit, or even capture it?

To reuse as much as possible existing elements or concepts, from various disciplines. Avoid new jargon or buzzwords.

3. MAIN CHARACTERISTICS
The characteristics of our approach are:

The central concept is the model.

The models are mostly represented in UML diagrams, but can also be textual documents, mathematical descriptions, or C++ code: we use whatever syntax is most appropriate to the aspect we want to reason about.

We handle various levels of abstraction (from philosophical elements to bits), because the intrinsic complexity of building software imposes to do so.

We realized prototypes for architectural decision-making, software testing and model transformation (at the implementation level).

We followed our model of software engineering to run real-life projects with customers in Finance, Telecom and Public Institutions.

In the following section we describe shortly the content of the model. An illustration of the model in a real-life example (one of our on-going projects) is given in section 5.

4. A BRIEF OVERVIEW
Software construction, like other engineering activities, is complex. It is therefore necessary to handle it at various level of abstraction. Figure 1 presents some core aspects of our work and the various levels of abstraction we are dealing with (due to space restrictions we had to make some choices on the aspects to show in this paper).

In our approach Software Engineering is a specialization of Engineering; Engineering in general has its foundations in science; and elements of description of Science have their foundations in Philosophy. For top-down consistence, it is necessary to understand these levels. Fundamentally, they are justifying the choices we make in software engineering. The relation between the models of different levels is the instantiation relation, except between ‘Engineering’ and ‘Software engineering’, where generalization is used (“Instantiation” and “Generalization” are used here in their OO meaning).

It is important to note that there is no definitive separation between the above levels. This is not (only) because of a lack of formalism, but also a consequence of the complexity we are facing. The operational part of our model (i.e. the part we are using in our day to day software construction) is the Software Engineering Part. All the above levels are there for the purpose of justification, validation and consistency.

4.1 Philosophical foundations
The objective of the “Philosophical foundations” level is to identify and describe the fundamental concepts we use in Software Engineering. For instance, it is now largely accepted in the Software Engineering Community that the notion of ‘Model’ is key for building software. Questions are still remaining: What is a model? What is the difference with other high-level concepts like “ontology”? What a model is composed of (what do we find in a model)? The philosophical foundations part attempts to give a first answer to these questions.

Figure 1. An overview of the model

To power of representation of this level is important, and its concepts are instantiated many times at lower levels of abstraction. We rely on “philosophical foundations” for the following points:

The separation of levels is described in the ‘Epistemology’ part, i.e. our global model is an instance of the ‘epistemology model’ (see fig. 1.)

All models of the software engineering are respecting the definition in ‘Metaphysics’ (see fig 1.)

Models are putting together concepts and relations. This aspect is defined in the theory of concepts (not shown in fig.1), and includes the notion of ‘extension’, intention’ [8]. The objective in introducing such aspects is to clarify as much as possible relationships between concepts (e.g. Generalization and Whole/Part).
4.2 Science

The “Science” level proposes models or theories for the description of scientific aspects of engineering (and thus software engineering). We propose a ‘Design Science’ Model, based on the work of Simon [4] and of the systemic theory [2]. Design science is about creating systems. In that process a clear view of the ‘system to build’ is necessary. To create a system, engineers have to make various decisions. We define a ‘decisional model’ (Fig. 2) that separates clearly the solution from the problem, and helps to find an appropriate set of solution to a set of objectives.

![Figure 2. An excerpt of the ‘decisional model’ (‘Science’ level)](image)

Another aspect of paramount importance when elaborating models is the separation between structure and behavior (see Fig. 3). Many scientific disciplines are applying such separation (physics, mechanics). Abstraction stands beyond modeling, and supposes to find what is common, and what is not. The structure is heavily independent from the time as opposed to behavior (note: ‘time’ is an aspect not shown in the paper but identified in the philosophical foundations level).

![Figure 3. Separating structure from behavior (‘Science’ level)](image)

Finally, additional aspects covered at the “Science” level but not presented in this paper is the identification of the relationships among concepts like ‘system’, ‘component’, ‘interface’, and ‘relation’.

4.3 Engineering

The “Engineering” level instantiates ‘design science’ models. It allows clarifying the purpose of various engineering models, in particular models aiming at describing an existing situation versus models clarifying the objectives for building a new system versus models aiming at proposing a solution to a given problem.

In addition, at the “Engineering” level we capture operational aspects of engineering activities. As such, we identified an ‘organization’ (e.g. a company) model and a ‘project model’ that deals with core management aspects and defined in ‘process model’ (see Fig. 4).

![Figure 4. An excerpt of the ‘process model’ (‘Engineering’ level)](image)

4.4 Software Engineering

The “Software Engineering” level is the primary focus of our work. We view software engineering as a specialization of Engineering. The models presented in the other levels are serving for justification and validation of the “Software Engineering” level. We specifically rely on this level to produce software in our projects. Figure 5 presents some essential models in “Software Engineering”:
Software acts in a given environment. That environment is most of the time heavily independent of computer science and directly related to the ‘domain’ or ‘business’ (or whatever other name is given to it). For a software developer, the first important thing is to have sufficient knowledge about this environment: it should be understood clearly, before creating software. In that context, software is a solution to a given ‘domain’ problem. Creating ‘domain’ models is therefore a very important aspect of the overall software construction process.

Software can be considered as a system in itself (at least as a subsystem). Depending on the type and the complexity of the software, several models could be necessary. The models to create a system are, beside of the previous ones, the ‘functional problem models’ and ‘technical problem models’ and their associated ‘functional solution models’ and ‘technical solution models’. Functional problems are strongly related to the domain.

In the “Software Engineering” level we also define rules to create, manage and validate the above models. We rely on the notion of ‘transformation’ (in the ‘MDA’ sense), but we also define the notion of ‘validation rule’ (e.g. a ‘testing rule’ – not shown in Figure 5 – is a validation rule). Transformations are used to produce the ‘functional’, ‘technical problem’, and ‘solution’ models. The set of transformations we defined so far are, for the moment, empirical.

5. An application example

This section gives some hints on how we use our model of software engineering in our day to day work.

We take the example of a European Institution project, which deals with supporting some business process between European member states and the European Commission related to subsidies distribution in the context of the Common Agricultural Policy. Figure 6 presents a part of the domain model of the project.

The Domain model shows that the system should apply rules defined in European regulations, which are fundamentally law articles. In order to process them automatically, we had to develop a hierarchy of these rules. These elements are part of the Functional Solution Model.

5.1 Functional solution model

A Functional solution model proposes a solution to an automation objective (present in the functional problem model not shown in here), independently of any technology. Figure 7 shows part of the hierarchy of rules.
Technical problem models deal with the core technical objectives the system should fulfill, for example, persistence, distribution, integrity, or error-handling. Figure 8 presents a part of the technical problem model we created in this project. The TPM was built by applying technical objectives to concepts of the domain model.

The TPM allows us to introduce technical aspects in the software process while maintaining a strong link with the knowledge of the domain captured in the domain model. For example, the TPM associates the problem of persistence to AgriculturalDeclaration. At this stage we apply iteration on the domain model to identify all dependencies of AgriculturalDeclaration to other concepts of the domain that are important in the context of persistence, i.e. aggregation and generalization. These dependencies are then added to the TPM. On the basis of the TPM and given some technical choices it is then possible to elaborate technical solution models.

5.3 Technical solution models

We illustrate technical solution models by pursuing the above example of persistence. For instance let’s suppose that the solution retained for persistence is a RDBMS, e.g. Oracle. We apply some transformations on the TPM to create one (or more) TSMs, for example to generate OO/relational mappings, or to create data schemas. Figure 10 shows a part of such transformation related to generating data schema’s DDL statements.

```vba
Sub CreateTable(classToTransform As Class)
    Print #1, "CREATE TABLE " + classToTransform.Name + "("
    ' create columns for variables
    For nI = 1 To classToTransform.Attributes.Count
        CreateAttributs(classToTransform.Attributes.GetAt(nI))
    Next nI
    ' create foreign keys for associations
    Dim clientRole As Role
    Dim supplierRole As Role
    Dim anAssoc as Association
    Dim ForeignTable As Class
    For nJ = 1 To classToTransform.GetAssociations.Count
        Set anAssoc = classToTransform.GetAssociations.GetAt(nJ)
        Set clientRole = anAssoc.GetCorrespondingRole(classToTransform)
        Set supplierRole = anAssoc.GetOtherRole(classToTransform)
        Set ForeignTable = supplierRole.Class
        If supplierRole.Navigable = True Then
            CreateForeignKey clientRole, anAssoc, ForeignTable
        End If
    Next nJ
    .......
End Sub
```

5.4 Methodology aspects

Our approach can be qualified as “agnostic” from the point of view of the process imposed by software methodologies. The models presented above cover most of the “requirements” of the methodologies we had to work with (RUP and XP for the moment). We usually apply transformations on our models (described in UML) to create the artifacts of the process imposed by the methodology of the project.

In the example presented here we have been imposed to work in RUP. Figure 9 shows a navigation path in the models created during the project, according to RUP’s phases, activities, artifacts and diagrams.
Requirements Artifacts are primarily the use case diagram generated from the domain model. Additional functional specifications are generated by the customer model (a model not shown in this paper and primarily capturing customers’ view of the system to build). Other artifacts are defined below.

Analysis Artifacts:

- The glossary of the domain. It proposes all important words of the domain, and thus the business assets and part of the business processes. If this document doesn’t exist, it will be generated from the classes of the domain model.
- The Activity diagram of the most important business process of the project.
- The State/Transition diagram of the Agricultural declaration, the core entity of the project.
- The class diagram of the domain.

Design Artifacts:

- The Functional Solution Model.
- The Technical Problem Model.
- The corresponding Technical Solution Models (TSM) for Persistence and User interface.

Implementation Artifacts:

- Java Code is generated from all TSMs, and completed manually for the dynamic part (methods and functions).
- The DDL (data definition language) code corresponding on the persistence TSM is generate automatically.
- The html pages corresponding to the User Interface TSM.
- The component diagram shows all piece of executable software develop so far for the project.

Testing Artifacts:

- Unit test suit generation created by application of model validation rules (see Fig. 5).

6. RELATED WORK

Our work is very much in line with the objectives of model-driven engineering, i.e. raising the importance of models in software engineering [9]. It is natural to work at a mega-modeling level in the context of technical solutions related to interoperability of modeling tools, and more generally, to model transformations like MDA. In our work we make no difference in naming between models, meta-models or mega-models. Every kind of model is there to serve a purpose and as such has a well defined position in the software construction process. Models might serve during analysis, during implementation or as translators among models. Our model of software engineering aims at capturing the rationale and some aspects of the elaboration process of software engineering models, thus, many of our models can be seen as meta-models.

Our work presents some similarities with some aspects of the specification of the Knowledge Discovery Meta-Model defined in the context of OMG’s Architecture-Driven Modernization task force [10]. Typically, in the specification of KDM there is clear separation between structure and behavior. Also, KDM’s separation of UI and data aspects can be seen as Technical Problem Models and/or Technical Solution Models. In our approach, the separation between structure and behavior appears already during analysis and captured primarily in the domain model. It could be interesting to investigate how domain models relate to the overall KDM approach and the work of the ADM task force in general.

7. CONCLUSIONS AND PERSPECTIVES

For years we have been gathering conceptual elements that guided us in facing the complexity of the activity of making software. We have distilled our knowledge and by trying to invent as less as possible we have stabilized relationships among some basic concepts of software engineering, but also of philosophy, design science, epistemology and the broader discipline of engineering. This paper presented the result of this work. The work is not complete; our metrics of completeness is the ability of our “model of software engineering” to cover the variety of model transformations in the context of our on going consulting and development projects. We decided to present our work at this stage to share our thoughts with people interested in modeling and model transformations of any kind and get feedback. We hope that some parts of our work might contribute in the understanding of (the potential Darwinian evolution of) software engineering models.

8. REFERENCES