Position paper: the use of retro-construction tools

Vincent Ribaud, Philippe Saliou

Position paper: the use of retro-construction tools

Vincent Ribaud
Université de Brest,
Informatics,
20 avenue Le Gorgeu, 29200 Brest
Email: ribaud@univ-brest.fr

Philippe Saliou
Université de Brest,
Informatics,
20 avenue Le Gorgeu, 29200 Brest
Email: psaliou@univ-brest.fr

***Abstract*—** To the question "If the construction phase can be completely automated, what do we need to teach students of software engineering about it?" we answer "teach them the tools that are able to automate the construction phase"

I. Introduction

'THE question we set for Software Engineering Workshop is "If the construction phase can be completely automated, what do we need to teach students of software engineering about it?" In our opinion, the shortest answer is "teach them the tools that are able to automate the construction phase". Hence, we should teach them how to provide inputs to such tools, and how the tools transform these inputs in software. From the point of view of constructing software, such inputs are design models (in the broad sense of model).

A. Topics

If we consider the SWEBOK topics addressed with these issues (http://www.computer.org/portal/web/swebok), they belong to two Knowledge Areas (KA) and are listed here: KA Software Design: 1. Software design fundamentals, 3. Software structure and architecture, 5. Software design notations, 6. Software design strategies and methods; KA Software Construction: 3. Practical considerations.

Annex D of SWEBOK presents a classification of KA topics according to Bloom's taxonomy: Knowledge (K), Comprehension (C), Application (AP), Analysis (AN), Synthesis (S), Evaluation (E). A closer look on the subtopics of these selected topics let us discard the subtopics classified K or C, because such knowledge will be taught somewhere in a software engineering degree, at least on a broad level. Then, we can consider the topics remaining together with the associated Bloom level in brackets.

For the KA Software Design: 1. Enabling techniques (AN); 3. Architectural structure and viewpoints (AP), Architectural styles (AN), Design patterns (AN);

5.Structural descriptions (AP), Behavioural description (AP); 6. General strategies (AN), Structured design (AP), Object-oriented design (AN).

For the KA Software Construction: 3. Construction design (AN), Construction language (AP), Coding (AN), Construction testing (AP), Construction quality (AN), Integration (AP).

B. Topics

Model-based engineering provides a systematic approach for producing software systems [1] based on the transformation of elements of a given abstraction level towards constructs of the level immediately inferior. We have to distinguish those models suitable for data from those suitable for processing, in particular related to time: data models are structural and static, whereas processing models are event-based and dynamic.

Automated tools for the construction phase are based on code generation. This is a typical top-down approach that might be hard for students. The proposal is to use an inductive method and the retro-engineering feature of such tools to let them learn the transition from design to code. Michalski defines inductive learning as "*a process of acquiring knowledge by drawing inductive inferences from teacher- or environment-provided facts. Such a process involves operations of generalizing, transforming, correcting and refining knowledge representations.* [2]"

II. Data Models

A. Physical Data Models

In an Information System or in any software using a database, a physical data model comprises a set of programming constructs in the SQL language. Learning the construction of a physical data model can benefit from the assistance of retro-engineering tools. Any modelling tool yields an automatic transformation from a logical model (using the relational model) to a physical model (using SQL language). Since the transformation process is completely controlled, many tools provide the user with an inverse transformation (called retro-engineering or retro-design depending on the tool) from the physical to the logical level.

The retro-feature ability yields new learning activities: from the retro-engineered logical (relational) model produced by the tool, students can reproduce the physical (SQL) model with the same or another tool, compare and analyse differences, infer transformation rules used by the tool, criticise tools' generation choices, look for better way of

using the tool. All these activities rely on verbs associated with the Analysis level (or higher) of Bloom's taxonomy.

B. Logical Data Models

A conceptual data model describes the organization and structure of a domain data with the help of entities, relationships, and properties, and this model is generally known as an entity-relationship (E-R) model. Transforming a conceptual (E-R) model in a logical (relational) model relies on a set of rules, normally mastered by any graduates in computing. What students are lacking is a confrontation with complexity, heterogeneity and legacy. A learning experience on a school case with few entities and relationships, easily transformed in few tables does not provide students with an understanding of the issues of a large Information System such as a banking IS or a transport reservation system.

There is a need for having at our disposal a large system composed of heterogeneous sub-systems, developed over periods of several years with different methods and different technologies by successive teams; and to have at our disposal the corresponding physical and logical models for each sub-system, and in some cases to have a maintained synchronization between conceptual and logical levels. We believe that major software vendors such as Oracle or Microsoft should be able to provide academy with such setting.

Most CASE tools master the transformation from the conceptual (E-R) level to the logical (relational) level and some CASE tools are mature enough to offer inverse transformation - an operation called retrofit. Hence, there are a lot of learning activities that can use the inductive, followed by deductive, scheme. Students will work on components for which they master the logical model, thanks to previous activities. They perform the retrofit of relational (logical) models and obtain an E-R (conceptual) model. Because the retrofit is incomplete and imperfect, students have to correct, enhance and complete it. Different options for retrofitting models can be experienced and observation of the results illustrates the transformation rules used by the CASE tool one way or another. Then students can transform the new E-R model into a relational model using the various options offered by the CASE tool. They will have to analyze, compare, explain, criticize, evaluate (Analysis, Synthesis, Evaluation levels).

III. Processing Models

On the processing modelling side, we do not have robust models but we do have a profusion of open-source code and documentation.

A. Physical Processing Models

If we consider a program (in a programming language) to be a physical processing model, then an algorithm is a logical processing model. Why should we start from algorithms? The presentation of complex algorithms to students is a tricky task, whereas they are able to handle parts of complex programs. Learning sessions may familiarize students with real-scale programs: they will assemble, slightly modify, rewrite, etc. software components and will work on samples of growing size. When an understanding of a part of the system (a sub-system) grows, we may ask students to describe the code organization in packages and to create a summary of functions' packages. They can also reorganize the code according to a set of naming and coding rules. Any system observation seen as a black box – description of results, identification of rules, list of services, recognition of patterns, etc. – can also be used to produce a more abstract model and it uses explanation, summary, or generalization, all activities arising out of an inductive approach.

B. Logical Processing Models

There is unfortunately no agreement on the semantics and conditions of use of different processing abstract models to be found in most modern methods. However, Krutchen proposed in [3] five views that have profoundly inspired UML genesis. The use case view is singular because it guides and explains the other views. The logical view is the object model of the design (where an object-oriented design method is used). The development view describes the static organization of the software in its development.

An example of inductive activity is the grouping of different software units into packages. Students can group units that have similar data usages in the same package. The grouping can be operated on code organization, analysis of components hierarchy, identification of components dependencies, i.e. which components are used by other components (which are indeed dependent). Once a broad understanding of the system or sub-system is achieved, students may perform a reorganization of the system components, called a refactoring. Refactoring seems to operate at logical and physical levels only yet it requires analysis and synthesis activities which reveal an underlying conceptual model that will be modified and transformed again into a logical model.

IV. Conclusion

What is remarkable in software engineering is the simultaneous existence of ever-more abstract representations, and the fact that this existence allows the engineer to think equally, and even simultaneously, within several levels. An experienced engineer studying a logical model 'sees' the various physical models that are implied, and conversely, using a physical model, 'sees' the underlying data model, and is therefore able to think about and act at both levels at once. Although this also happens in processing models to a certain extent, it remains a special feature of data models. However some CASE tools offers very powerful processing abstraction based on the Model-View-Controller paradigm and other design patterns.

REFERENCES

[1] S.A. Bohner, and S. Mohan, "Model-Based Engineering of Software: Three Productivity Perspectives", in *33rd Annual IEEE Software Engineering Workshop*, 2009, pp. 35-44.

[2] R.S. Michalski, "A theory and methodology of inductive learning", in *Artificial Intelligence*, vol. 20 (2), February 1983, pp. 111-161.

[3] P. Krutchen, "The 4 + 1 View Model of Architecture", in *IEEE Software*, vol. 12 (6), 1995, pp. 42-50.