

# *Separation of Concerns*

## *in*

# *Software Engineering Education*

**Naji Habra**  
Institut d'Informatique  
University of Namur  
Rue Grandgagnage, 21  
B-5000 Namur  
+32 81 72 4995  
nha@info.fundp.ac.be

## **ABSTRACT**

Separation of concerns is the main principle of Software Engineering. It represents a key element in the teaching process of any Software Engineering methodology. The paper relates the experience of the University of Namur in introducing the separation of concerns principle in its educational scheme through an extended student project.

## **1. INTRODUCTION : SEPARATION OF CONCERNS & SOFTWARE ENGINEERING EDUCATION**

The major aim of Software Engineering Education is to teach Software Engineering principles such as separation of concerns, rigor, modularity, abstraction, anticipation of change, etc [3]. The challenge is that students should not only understand those principles as theoretical concepts but they should also be able to use them efficiently in many practical and very different situations. The risk lies in the fact that a principle learned in one or two particular contexts remains misunderstood because students associate it inadequately with some peculiarities of the studied case(s). This problem is more sharply set for the separation of concerns principle.

The practical facet of this theoretical principle is mainly the methodological skill allowing Software Engineering practitioners to make adequate decomposition and composition. However, in general, decomposition and composition are made across the different phases of the life cycle, on basis of different dimensions and using different formalisms. We believe that acquiring such methodological skill is a key aspect of students' training in Software Engineering. A good methodologist have a hindsight on the different possible dimensions of the software being developed (data view, functional, used view, distribution view, ...). He is able to manage different dimensions of separation of concerns and shift smoothly from one dimension to another across the different phases and/or the different projects. Of course, this *portrait* does not represent the typical methodologist; the caricature of Martin Fowler<sup>1</sup> has some authentic basis. Still, we believe that our portrait is not completely imaginary and, as a teacher, we hope it is not completely unreachable.

---

<sup>1</sup> “-What is the difference between a methodologist and a terrorist ? – You can negotiate with a terrorist.”

Obviously, methodological skill is mainly acquired by experience. Nevertheless, university training which fixes the basis and prepares to such experience should stress on the flexibility of any methodological approach and on the variety of the possible types of separation of concerns.

However, training usually materializes by very few software realizations in which students make use of only one formalism. Students' natural trend is thus to develop a rigid (quasi-algorithmic) methodology in which decomposition steps are made according to one separation of concerns dimension which become the dominant dimension. A major problem in Software Engineering education is to avoid transmitting a rigid methodology involving one dominant separation of concerns.

The Software Engineering educational scheme we present hereafter aims at training students to acquire a flexible methodological skill allowing them to have in mind several dimensions of separation of concerns along the different stages of software development and to adapt them adequately to the particular case they deal with.

## 2. CONTEXT

Our curriculum is spread over five years and leads to a Master degree. The central part of the Software Engineering teaching is given during the fourth year which includes two complementary SE modules: one 30-hour module for theoretical lectures and one 120-hour module for the SE project. Projects are achieved by groups of five to six students each and are supervised by a team of experienced staff members.

The two SE modules constitute the core of a large corpus extending over the last three years of the curriculum. On the one hand, several modules of the third year concern techniques like OO programming, distributed programming, databases design and human-machine interface design. These courses involve exercises on specific small or medium size problems. They also require exercising on larger projects and the SE project of the fourth year appears to be the ideal place for that. On the other hand, modules of the fifth year related to management aspects make use of the SE project as a first hands-on experience with such topics.

## 3. UNDERLYING PRINCIPLES

The aim of the software process to be followed by students to achieve their Software Engineering project is twofold. On the one hand, the Software Engineering project is the opportunity to carry out the development of a sizeable software all *along its life cycle* and to get a practical experience, being as close as possible to a realistic production environment. On the other hand, the projects aims at bringing them to understand the different dimensions of a software in general and to acquire a flexible methodological skill.

The methodological notions underlying the students' Software Engineering project are made explicit to the students before they start. These notions can be summarized as follows:

- A software has different dimensions according to which the separation of concerns can be made.
- Each of these dimensions leads to a particular view which can be helpful in understanding the system being developed or decomposing it or both.
- In the early stages of the software life cycle (e.g. requirements analysis), the main goal is to *understand* the system to be developed while in latter stages (e.g. architectural design and coding), the main goal is to produce a manageable system with properties like *reusability and maintainability*. Therefore, the separation of concerns needed should not necessarily follow the same dimension; a type of separation of concerns that is useful for understandability is not necessarily adequate for reusability nor vice versa.

- We identify, at least for the students' project, the following dimensions :
  - a *data* dimension which is interested in the “entities” or the “pieces of information” the system deals with;
  - a *functional* dimension which is concerned by the “functions” or the “services” the system should offer;
  - a *user view* dimension which is involved in the different “scenarios” or the different “functional units” considered under the users' viewpoint;
  - a *reusing* dimension which is concerned by the “components” of the system as they are implemented in the chosen language;
  - a *distribution* dimension which is interested by the “components” of the system as they are implemented on different platforms.
- These dimensions are not independent nor orthogonal, but they are not necessarily incompatible with each other. For example, the users' view may correspond to a functional view in which each “scenario” corresponds to one “big” functional service. A reuse-based decomposition may also correspond to a data decomposition. However, such correspondences do not hold in the general case and we can not establish a general methodology on such assumptions.
- Viewing a dimension as the dominant one (i.e. *representing* the system and *decomposing* it mainly according to that dimension) may offer an adequate methodological guideline for a particular software or even for a category of similar ones.
- In fact, one of the big problems of Software Engineering education is related to this possibility. Students appreciate having a well-guided strict methodology to be followed as it is. So, using one dimension, (e.g. the “data” dimension or the “user scenarios” dimension), as the basis of the whole software life cycle is a very attractive idea. Moreover, this idea corresponds to methods proposed in former lectures (e.g. databases and use-interface lectures) and it works for the particular class of the problems concerned. One of the main aims of the SE project is that the students should definitely avoid to think that such mechanical methods are universal.
- Unfortunately, a part of the object-orientation literature reinforces this risk by imposing the “class” as a universal encapsulation paradigm when decomposing into “classes” becomes the universal separation of concerns along the whole life cycle. Of course, the different dimensions are hidden behind the ambiguity of the used vocabulary. At the analysis stage, a “class” could correspond to the encapsulation of pure data (with the influence of the entity-relationship approaches [1]), to the encapsulation of data together with the functions related to it (with the influence of abstract data type approaches [4]) or even to a pure functional encapsulation (with the influence of bad OO programming where any group of functions could be gathered in a “class”). At development stages, the same word “class” becomes an encapsulation of “reusable units” and the type of separation of concerns is not the same. The problem of the “tyranny” of the dominant dimension in the separation of concerns is hidden by a vocabulary misunderstanding.
- The process proposed to the students for their Software Engineering project aims notably at remedying to the problem of the dominant dimension; the idea is to lead them to use different dimensions at the different stages, to maintain the traceability between those dimensions and to justify their choices at each stage.

#### 4. PROPOSED PROCESS

- Students carry out the software project across all its life cycle.
- *For the requirements analysis stage*, they work with the use cases approach to identify the big scenarios of the future system; those scenarios are described in a structured natural language. Identified non-trivial operations are defined by a couple of precondition/postcondition. At this point, the chosen dimension is the users' view on the system. Of course some big "services" and big "data chunks" begin to appear at this stage.
- *The analysis stage* involves several modeling achieved in parallel; each modeling is made according to one dimension. Thus, students produce different schemas.
  - On basis of the use cases scenarios, they identify the big "functions" of the system; then they decompose these functions in a classical top down way.
  - They identify the different classes of data involved in the system and produce a classical information system modeling. They use classical criteria learned in the database engineering lectures and develop a model showing classes and relationships.
  - The user-interfaces are issued directly from the use cases; each interaction scenario with an external actor gives rise to the description of a dialogue that corresponds to the user-view on the scenario. For this model, they use the criteria and the formalism learned in the user-interface lecture.
  - On basis of the non-functional requirement (about security constraints, configurations, etc) they produce what we call a distribution model. This model describes which part of the system will run on which server and/or which client. For this modeling, they use the classical literature about the so-called client/server architecture (two-tiers, three-tiers, etc).

For those four models which represent four dimensions of the software, they should highlight the association between their components. For example, a user interface involves naturally operations that appear, at some level of abstraction, in the model of the functions decomposition. The different parts (i.e. the tiers) in the distribution architecture corresponds to a collection of functions, data classes, interfaces or a mix of them. Associations are made explicit by the of naming convention chosen and/or other devices in the flavor of those presented in [6].

*A logical architectural design* stage consists of a creative process of consolidation and conciliation of the four dimensions above mentioned. The aim is to get general, reusable and maintainable components, in the very classical sense of Parnas [5]. For this stage, we emphasize the fact that separation of concerns should be made according to a new dimension. Of course, this dimension could give rise to a decomposition which is close to the data dimension or to the functional one, etc. However, it is important to keep in mind the underlying objective of the architectural design, notably, reusability and maintainability.

- *A physical architectural design* consists of the translation of the above architecture in terms of physical units available in the used language(s): Java classes or interfaces, packages, RMI calls, etc.
- *The coding and testing* phases involve classical programming and test activities.

## 5. FEEDBACK & EVALUATION

The problem chosen is a software for a fictitious full-automated stock-market place which involves services such as handling traders' accounts, managing offers of trades, fixing opening and closing prices, furnishing statistics, etc. The idea is to choose a problem for which finding an adequate architectural design is not a trivial task.

We receive from the students a wide variety of solutions that can not be developed here because of the lack of space. The different models required for the analysis phase present certain similarities. Most groups identify data classes like "traders", "companies", "trade offers", etc. They also identify user interfaces for "trading interface", "consulting interface", "account managing interface", etc. Functional decompositions are rather alike and the distribution models proposed are conformed to multi-tiers known patterns [2].

Besides, the patterns of the architectural design proposals are very disparate. Some architectural design proposals are based on classical two-tiers or three-tiers schemas where each tier is decomposed more or less according to the corresponding analysis model. But a number of architecture proposals show original encapsulation units. Such new components seem to emerge from the idea that the dimension concerned at this stage is related to an architectural criteria (reusability, etc) and not to a comprehensibility criteria. In all cases, the traceability with the analysis models is maintained. At the end of the project, the different architectures are presented and evaluated comparatively.

The experience shows clearly that the logical architectural design remains the most creative stage of the life cycle and presenting it as such to the students appears to be rewarding. In fact, the process proposed helps them to understand the different dimensions of the system under construction and, in particular, their interconnection.

## 6. REFERENCES

1. Chen, "The Entity-Relationship model : towards a Unified View of Data", *ACM transactions in Database Systems*, Vol.1, N°1, 1976.
2. M. Fowler, *Analysis Patterns : Reusable Object Models*, Addison-Wesley Series in Object-Oriented Software Engineering, Addison-Wesley, 1997
3. C. Ghezzi, M. Jazayeri & D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall Inc., 1991.
4. J. Guttag, "Abstract Data Types and the Development of Data Structures", *Communication of the ACM*, Vol.20, N°6, 1977.
5. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communication of the ACM*, Vol.15, N°12, 1972.
6. P. Tarr, H. Ossher, W. Harrison & S.M. Sutton Jr., "N-degrees of Separation: Multi-Dimensional Separation of Concern", *Proceedings of the International Conference on Software Engineering (ICSE 21)*, May 1999.