The Six Concerns for Separation of Concerns

Mehmet Akşit
Bedir Tekinerdoğan
Lodewijk Bergmans

Software Engineering, Dept. of Computer Science, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{aksit|bedir|bergmans}@cs.utwente.nl

Abstract

Despite a common agreement on the necessity of the application of the separation of concerns (SOC) principle, there is not yet a consensus for its key issues. The separation of the concerns is usually based on the adopted programming paradigm, the applied method or even the programming language. This paper presents the so-called six 'C' properties that can be applied as a guideline for defining and evaluating the approaches that adopt the SOC principle.

1. Introduction

One of the most important principles in software engineering is the separation of concerns (SOC) principle. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability and reusability.

Despite a common agreement on the necessity of the application of the SOC principle, there is not a well-established understanding of the notion of concern. The separation of the relevant concerns is usually based on the adopted programming paradigm, the applied method or even the programming language. In object-oriented methods, for example, the separated concerns are modeled as objects and classes, which are generally derived from the entities in the requirement specification and use cases. In structural methods, concerns are represented as procedures. In aspect-oriented programming, the term concern is extended with the so-called *crosscutting* properties such as synchronization, memory management and monitoring.

We think that there is a need of renewed understanding of the requirements for the SOC principle. For this purpose, this paper presents the so-called six 'C' properties: *Concern-oriented, Canonicality, Composability, Computability, Closure property* and *Certifiability*. In the following sub-sections, by using an example we will explain these in more detail.

2. The Six C Properties

2.1 Concern-Oriented

In natural language the word concern usually means 'the matter for consideration', or 'marked interest or regard' [7]. The concern-oriented property ensures that the software system is relevant and as such is valid for the context as defined by the problem. However, identifying the relevant concerns is not trivial. To support this, we distinguish basically between the following two concern types:

- *Problem domain concerns,* represent the concerns as it is defined from the client perspective. It basically focuses on the functionality of the system as the client expects it.
- *Solution domain concerns,* represent the concerns as defined by the solution techniques.

Assume for example that a software system has to be designed and implemented for a group of car dealers connected together through a network. This system must provide services such as repair and maintenance, registration, tax management and insurance. Further, the system must minimize the material and labors costs, and optimize the speed performance. To reduce the number of spare parts, the dealers share some of the components stored in their stocks. For example, expensive car models may only be kept by a selected number of dealers. Further, the components that are not frequently required may be shared among the dealers that are not far from each other. To minimize the costs, the software system

can keep track of these shared car models and spare components.

The above requirements include the problem domain concerns, which can also be represented in more detail by use-cases and scenarios.

The problem domain concerns, however, may not include the necessary concerns for implementing the software system. This is because, many important concerns may be transparent to the user.

We claim that the relevant concerns must be derived from the solution domain [1]. For example, sharing components across a network requires mechanisms for managing consistency. The theory of atomic transactions, for example, can be adopted to solve this technical problem. Further, mathematical techniques may be used to optimally distribute the components across the stocks. This requires knowledge on optimization techniques. In addition, to tune the system performance, the transaction system may be monitored and controlled. This requires knowledge on control systems. All these solution domains are necessary for implementing a robust and fast distributed stock management although they are not explicitly included in the requirement specification.

The utilization of the solution domain concerns does not exclude the problem domain concerns, however. The problem domain concerns are particularly useful in identifying the solution domain concerns and for designing user-interfaces.

2.2 Canonical

The canonical form property requires that the concerns are general and represented as succinct as possible. The generality property refers to the common and stable parts of the software system. The succinct property avoids redundancy and as such unnecessary complexity. These properties are important in designing robust, adaptable and reusable software systems.

Canonical models can be identified by comparing the multiple relevant solutions for the same problem. For example, after comparing a large number of transaction techniques, we have identified the following set of common concerns: Transaction, Transaction Manager, Policy Manager, Data Manager, Scheduler and Recovery Manager [6]. Similarly, the scheduler concern could be decomposed further into the following common sub-concerns: Synchronization Scheme, Synchronization Strategy and Performance Detector.

Numerous synchronization schemes exist of which each performs differently depending on the execution context. To tune the performance of the transaction system with respect to the changing context, it is necessary to specify the stable and the variable parts so that the synchronization scheme can be adapted. This inherently requires the canonicality property.

2.3 Composable

To define higher-level concerns requires composition operators for manipulating, combining and extending the concerns.

Assume that the concern C_3 is composed from C_1 and C_2 using the composition operator $\oplus$:

$$C_3 = C_1 \oplus C_2.$$

The composition operator may be applied at different binding times such as compile-time or run-time. In object-oriented languages, for example, inheritance and aggregation are provided as compile-time and run-time mechanisms, respectively. The operator $\oplus$ may also represent a more complex composition mechanism such as middleware.

Sometimes, the composition cannot be realized if the adopted language (1) cannot represent the necessary concerns separately and/or (2) cannot provide the appropriate composition operators.

Some concerns may be naturally non-separable, which are termed as *crosscutting concerns*. Hereby crosscutting refers to inevitable scattering of concerns to multiple abstractions.

For example, in the car dealer management system, to tune the performance of the transaction system, various software modules need to be monitored. This requires repetitive implementation of the monitoring code in the corresponding operation. The monitoring code is said to crosscut the transaction system because it is scattered across many operations in different modules. Because of the tangled code, crosscutting hinders the adaptation and extension of the monitoring concern.

Aspect-oriented programming languages [4][2] aim at providing implementation mechanisms for composing the crosscutting concerns.

2.4 Computable

The fourth issue is to express the concerns in a common computable platform. The *computability* property is necessary for creating executable software systems. For this the concerns must be expressed as first class abstractions in the implementation language so that they can be manipulated individually. This means that given the set of solution abstractions as defined by SA = (sa_1, sa_2, sa_3, .. ,sa_n), we can define an implementation model IA in a language such that:

$$IA = (ia_1, ia_2, ia_3, .. ,ia_n)$$

where there is a bijective mapping from SA to IA. This is to say that every abstraction in SA, can be separately represented by an abstraction in IA. Implementation abstractions may correspond to the abstractions provided by the implementation environment such as Java language constructs, CORBA IDL, operating systems API's, etc.

In theory, for every individual technical problem, an ideal implementation language can be defined. In practice, however, there is a large set of technical problems that require different implementation languages. Given the restricted resources, however, this is not a viable option and usually a general-purpose language is used instead. This may cause certain concerns crosscut in the implementation. For example, in current object-oriented languages, synchronization and real-time constraint specifications cannot be represented and composed as separate concerns [5]. This may hinder the adaptability and reusability of these concerns in the implementation.

2.5 Closure

The fifth issue is to preserve the canonical structure and the composition operations of the design in the implementation platform. This is called the closure property, and is necessary for maintaining the quality factors of the design at the implementation level. The closure property can be expressed using the following function:

$\oplus : C \times C \rightarrow C.$

Here, the composition operator $\oplus$ is a function that maps the product of two concerns into a new concern.

In the transaction system design example, the *Scheduler* and *RecoveryManager* concerns are expressed separately. The *DataManager* concern composes these concerns together. Because of the closure property, scheduling and recovery mechanisms can be replaced freely at run-time.

2.6 Certifiability

The sixth issue is to certify the implementation of the concerns with respect to functional and/or quality requirements. The *certifiability* property is necessary for evaluating and controlling the quality of design and implementation models.

In the transaction design example, certifiability can be provided by using the semantics of solution domain abstractions. For example, the correct adaptation of the concern *Scheduler* has been specified in the solution domain. This information can be utilized in verifying the correctness of the adaptation of *Scheduler* [6]. Furthermore, simulation techniques can be used to justify the tuning of the performance.

3. Conclusions

In this paper, we have identified six fundamental key issues of the separation of concerns principle, that we termed as the six 'C'-properties: Concern-Oriented. Canonical, Composable, Computable, Closure, Certifiable.

References

[1] G. Arrango. *Domain Analysis Methods*. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[2] [Bergmans & Akşit 96] L. Bergmans and M. Aksit, *Composing Synchronisation and Real-Time Constraints*, Journal of Parallel and Distributed Computing 36, pp. 32-52, 1996.

[3] [Bernstein et al. 87] P.A. Bernstein, V. Hadzilacos & N. Goodman, N. *Concurrency Control & Recovery in Database Systems*, Addison Wesley, 1987.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997.

[5] S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, pp. 107-150, 1993.

[6] B.Tekinerdoğan. *Synthesis-Based Software Architecture Design*, PhD Thesis, Dept. of Computer Science, University of Twente, The Netherlands, 2000.

[7] On Line Webster's Dictionary, 2000