

Separation of Concerns

Erik Ernst

Dept. of Computer Science, University of Aarhus, Denmark
ernst@daimi.au.dk

Abstract. Separation of concerns is a crucial concept in discussions about software engineering consequences of programming language design decisions, especially in AOSD. This paper proposes a way to formalize this concept, and argues that the given formalization is useful even if it is used primarily in an informal way.

1 Introduction

The connection between software engineering and language design is very complex, but the degree to which a given language design supports separation of concerns is a crucial issue, especially in AOSD where it is a core effort to support separate expression of concerns which would otherwise be tangled into each other and scattered across systems. Separation of concerns is crucial for numerous software engineering properties.

Firstly, one of the best complexity management devices is the support for composing a system from components—in a broad sense, where a component could be a C++ [9] class, a package in the Java [3] language, an Standard ML [7] functor, etc. Correctness is supported by specifications—e.g., a simple class interface, a formal specification, a natural language document, etc. Componentization greatly reduces large-scale complexity, because the internals of each component may be hidden, and each component is simpler to implement because it just needs to satisfy the specification—concrete usage of the component may be ignored. In summary, separation of concerns into components may enhance comprehensibility, because of simplicity, ensured by locality.

Secondly, when a complex system is componentized, system development is made more flexible because components can be modified rather independently (except when changes at the architectural level are required). Moreover, the software is well suited for the construction of many different systems by using components in many different configurations. This enables, e.g., software product lines. In summary, separation of concerns into components may enhance speed and flexibility (in both design and implementation), reusability, and configurability.

It is well-known that the list of *-ilities* could be made much longer, but we consider these issues especially important.

In short, we really want separation of concerns. Sad only, that we don't know what it means!

This is of course a harsh statement, but consider just how hard it is to come up with a precise definition of ‘concern’, or what exactly it means for a concern to be ‘separated’ from others. The contribution of this paper is to propose a definition of the notion of a concern at the technical level, supplemented with a conceptual characterization, and to propose a way to make it more precise when/if a given concern is separated. The approach has a formal core, and it can be made entirely formal; but we suspect that this will be highly unwieldy. It is more likely that it

can be useful as a mental framework in everyday analysis of concerns and their separation.

The rest of the paper is organized as follows. Section 2 presents a couple of mathematical concepts which are needed in the formalization of concerns and separateness, which is then presented in Sect. 3. Finally, Sect. 4 concludes.

2 Preliminaries

In order to present the formalization of separated concerns, we need to mention a couple of mathematical definitions. First consider the well-known notion of a homomorphism [1]:

Definition 1 (Homomorphism). *Given two sets S and T , two operations $\oplus : S \times S \rightarrow S$ and $\odot : T \times T \rightarrow T$, and a function $\varphi : S \rightarrow T$; φ is then a binary homomorphism from (S, \oplus) to (T, \odot) iff*

$$\forall x, y \in S. \quad \varphi(x \oplus y) = \varphi(x) \odot \varphi(y). \quad (1)$$

Given S, T , and φ as above, and functions $f : S \rightarrow S$ and $g : T \rightarrow T$; φ is then a unary homomorphism from (S, f) to (T, g) iff

$$\forall x \in S. \quad \varphi(f(x)) = g(\varphi(x)). \quad (2)$$

It is common to define only binary homomorphisms, but it easily generalizes to n -ary operations, and we need the unary variant. Intuitively, a binary homomorphism can be described as a function which is able to *translate* the elements of a set along with an operation (S, \oplus) to another “universe” (T, \odot) , because \oplus plays the same role in S as \odot does in T . Similarly, a unary homomorphism translates a pair (S, f) into another pair (T, g) .

The unary homomorphism property (2) can also be written as $\varphi \circ f = g \circ \varphi$. Using a notation from category theory [8], we can express this by saying that the following diagram commutes:

$$\begin{array}{ccc} S & \xrightarrow{f} & S \\ \varphi \downarrow & & \downarrow \varphi \\ T & \xrightarrow{g} & T \end{array} \quad (3)$$

Diagram (3) means exactly the same thing as equation (2). However, such a diagram is very useful when describing our formalization of a separated concern, as we shall see in the next section.

3 Formalization of Separated Concerns

First, note that we consider the notion of a *concern* to originate at the conceptual level, as a natural language concept that is essentially the human-thought counterpart of a component at the technical level (in the broad sense used in Sect. 1). In other words, when we discuss ‘concerns’ we are discussing something which is essentially non-technical and non-formalizable. For this reason, we will propose a definition of a concern at the technical level which is so flexible that basically *anything* could be a concern. The idea is that each concern should be rooted at the conceptual level, and the technical notion of a concern should just follow the conceptual concern like a shadow. This means that the technical notion of a concern is rather useless technically, and it only becomes interesting when we ask how ‘separate’ a given concern is, and this is what we will present a formalization of in the following.

3.1 Setting the Stage

Assume that we have a programming language \mathcal{L} . Let \mathcal{P} be the set of all programs written in this language. You may think of \mathcal{P} as the set of all correct abstract syntax trees in \mathcal{L} . Moreover, let \mathcal{S} be the set of all possible semantic values for \mathcal{L} , such that every program $p \in \mathcal{P}$ has a semantics $s \in \mathcal{S}$. Finally, let $\llbracket \cdot \rrbracket : \mathcal{P} \rightarrow \mathcal{S}$ be the semantic function, such that $\llbracket p \rrbracket$ is the semantics of the program p . We use the symbol \perp to denote the bottom element of \mathcal{S} , i.e., the semantics of a program that stops in an error state, including syntactically wrong programs—which may be considered to stop in an error state just before the first instruction. In this setting we will define separation of concerns.

3.2 The Concern Diagram

We claim that the following diagram, the *concern diagram*, is a useful starting point for the definition of separated concerns:

$$\begin{array}{ccc}
 \mathcal{P} & \xrightarrow{\pi_{\text{syn}}} & \mathcal{P} \\
 \llbracket \cdot \rrbracket \downarrow & & \downarrow \llbracket \cdot \rrbracket \\
 \mathcal{S} & \xrightarrow{\pi_{\text{sem}}} & \mathcal{S}
 \end{array} \tag{4}$$

In this diagram, $\pi_{\text{syn}} : \mathcal{P} \rightarrow \mathcal{P}$ is a *syntactic reduction function*. It maps programs to programs, and the intention is that it is given as argument a program that includes a specific concern, and it then returns the same program except that it does not include the concern. Programs not including the concern should be returned unchanged. In other words, the syntactic reduction function removes the concern from the program.

Similarly, $\pi_{\text{sem}} : \mathcal{S} \rightarrow \mathcal{S}$ is a *semantic reduction function*. It maps semantic values to semantic values, i.e., it transforms the meaning of a program to the meaning of another program. The intention is that it transforms the meaning of a program containing a specific concern into the meaning of the same program where that concern has been removed.

As all diagrams, the concern diagram (4) should commute, i.e., we seek reduction functions such that for every program p , $\llbracket \pi_{\text{syn}}(p) \rrbracket = \pi_{\text{sem}}(\llbracket p \rrbracket)$. In other words, we seek such reduction functions that it makes no difference whether we remove the concern and then execute the program, or we execute the program and then reduce the semantics to remove the effect of the concern (typically, semantic reduction means skipping some actions).

The reason why it is meaningful to use a concern diagram, i.e., to require that $\llbracket \pi_{\text{syn}}(p) \rrbracket = \pi_{\text{sem}}(\llbracket p \rrbracket)$, is that this ensures consistency between the syntactic and the semantic level. In other words, we use the syntactic reduction function as a tool to specify exactly what the concern looks like, and we use the semantic reduction function to specify exactly how the concern works. Since we do this relative to the system where the concern is expressed, we are forced into describing the degree of separation: If π_{syn} is complex then the concern is not well separated at the syntactic level, and if π_{sem} is complex then the concern is not well separated at the semantic level. In the former case we would say that the concern is scattered or cross-cutting (as in [5]), and in the latter case we would say that the concern and the rest of the system exhibit complex semantic interdependencies.

It is our opinion that the most valuable and manageable kind of separation includes both levels.

3.3 Semantic Values

Not all choices for the structure of \mathcal{S} are equally useful. We should use a representation of the semantics of programs which is close to the actual execution semantics, and hence not so close to the program syntax. Consider the opposite extreme—we could use the program itself as a representation of its semantics, and the semantic reduction function could then trivially be the same function as the syntactic reduction function. However, this would seriously reduce the value of this analysis framework. We believe that it is crucial to be able to distinguish between (1) a concern which is well separated in both the syntactic and the semantic dimension, and (2) a concern which is well separated syntactically, but semantically tangled. In fact, we don't even consider this controversial, so the interesting contribution here would be to improve the precision in the characterization of whether, how, and to which degree separation is present.

A useful simplification is to consider each program along with its input, i.e., to choose one particular set of input data and stick with that. This corresponds to considering just one execution of the program. Later, we can generalize and argue that a given conclusion holds for all possible sets of input data, i.e., for the complete semantics of the program. Beyond that, we may also be able to generalize to multiple programs, or even all programs containing a given concern.

With this approach we can represent the semantics of a program as a deterministic sequence of simple operations, similar to a sequence of Java byte code instructions, but without jumps of any kind. Think of this as a log file created while singlestepping the program, only logging instructions which modify the store. The semantic reduction should now describe how to transform such a sequence, typically by specifying which instructions to skip. To be able to do this, it is often necessary to annotate each instruction in the sequence with, e.g., the name/signature of the currently executing method and the class of the enclosing object.

3.4 Instantiating the Concern Diagram

We may approach the goal of satisfying $\llbracket \pi_{\text{syn}}(p) \rrbracket = \pi_{\text{sem}}(\llbracket p \rrbracket)$ at many different levels of generality.

Consider the case where a program p_0 contains a specific concern, and p_1 is the same program except that the concern has been removed. In other words, assume that $\pi_{\text{syn}}(p_0) = p_1$. We must ensure that $\llbracket p_1 \rrbracket = \pi_{\text{sem}}(\llbracket p_0 \rrbracket)$, and since π_{sem} is the only unknown in this equation, we seek a semantic reduction function such that the equation holds.

If we consider the diagram not only for one argument value but, e.g., for all programs containing the given concern, we must find a semantic reduction function π_{sem} which will remove the meaning of the concern from the meaning of any program containing this concern. This may be hard, but it would be a very precise characterization of the effect of the concern.

Consider the well-known example of using an AspectJ [4, 2] aspect `Logging` (similar to `aspect ShowAccesses` in [6]) to log executions of some methods. In this case, the syntactic reduction would simply delete the aspect `Logging` from the program, and the semantic reduction would skip all operations on the log file performed in code of this aspect. Skipping operations may leave behind some actions with no lasting effect—e.g., assignment to a local variable which is never used—and the semantic reduction should remove such actions in order to arrive at exactly the same semantics as the semantics of the program without the `Logging` aspect.

However, if an aspect has base code visible side-effects—e.g., by assigning to a boolean instance variable whose value is later used to determine which branch to take somewhere in base code—then it is *not* sufficient to skip the assignment

instruction. This is because the semantics (especially if it is similar to an instruction trace) will still work as if the assignment had taken place. This reveals that side-effects escaping to base code may easily cause deep semantic entanglement.

It may seem that this makes our whole approach useless, because practical problems will always be much too complex to handle. However, we believe that it will be quite useful, because thinking in this manner puts a strong spotlight on the difference between real semantic separation, and deep, but subtle semantic tangling.

3.5 Defining Concerns and Separation

We have chosen to describe a concern, at the technical level, as anything which may be removed by applying a function, π_{syn} . Of course, this is an extremely broad definition. However, we believe that this is indeed a reasonable choice, because the notion of a concern is rooted in the real world. The interesting concept is that of a *separated concern*, and it is actually possible to define this concept now.

We define separation as a property of a reduction function, i.e., a concern may be either *syntactically* or *semantically* separated from the rest of the program. We have shifted the problem now, because we make both the syntactic and the semantic reduction explicit, and because the notion of removing a concern entirely is easier to make precise than the notion of separation.

It is ultimately up to the user of this analysis framework to decide on how much separation is implied by a given reduction function, but we can give some typical examples:

- A syntactic reduction function exhibits a high degree of separation if it has no effect except that it deletes a single subtree from the program (considered as an abstract syntax tree); a reduction function that removes an AspectJ aspect would be an example of this.
- A syntactic reduction function exhibits a low degree of separation if it has no effect except that it deletes a large number of scattered subtrees of the abstract syntax tree; a reduction function that removes a lot of scattered `println(..)` statements which happen to perform logging would be an example of this.
- A syntactic reduction function exhibits a very low degree of separation if it introduces other changes than deleting program elements; an example of this could be a reduction function which changes the number of arguments of a given method and overriding versions of it, as well as all invocations of them.
- A semantic reduction function exhibits a high degree of separation if it has no other effect than skipping the initialization of one or more variables (global/static variables, or instance variables) as well as all operations on them; the `Logging` example has this structure (using a global/static variable), just like other AspectJ advice operating only on state in the aspect or introduced by the aspect—which must then not be used by other parts of the program, either.
- A semantic reduction function exhibits a low degree of separation if it is complex.

4 Conclusion

We have presented an abstract semantic characterization of separation of concerns. It is based on the notion of reduction functions. The idea is that we use a syntactic reduction function whose effect is to remove the concern syntactically, and we use a semantic reduction function whose effect is to remove the meaning of the concern from the meaning of the program. These two functions must be chosen such that it makes no difference whether we remove the concern syntactically first and then investigate the meaning of the resulting program, or we take the meaning of the program and then remove the meaning of the concern (this is the same as saying

that the concern diagram (4) must commute). We believe that this approach to separation of concerns provides new insight because it emphasizes the difference between syntactic and semantic separation, and it offers a concrete way to describe this separation in a manner which may be fully formalized, and which will guide the line of thinking even if it isn't.

References

1. R. B. J. T. Allenby. *Rings, Fields, and Groups – an Introduction to Abstract Algebra*. Hodder and Stoughton, Ltd., London, England, 1983.
2. Gregor Kiczales et al. Aspectj home page. <http://aspectj.org/>.
3. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Publishing Company, 2000.
4. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 327–353, Heidelberg, Germany, 2001. Springer-Verlag.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.
6. Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJTM. In *Object-Oriented Technology – ECOOP'98 Workshop Reader*, LNCS 1543, pages 398–401, Heidelberg, Germany, 1998. Springer-Verlag.
7. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.
8. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Massachusetts, 1993.
9. Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, Reading, MA, USA, 1991.