

Understanding separation of concerns

Hafedh Mili , Amel Elkharraz & Hamid Mcheick

Laboratoire de recherche sur les TEchnologies du Commerce Électronique (LATECE)

UQAM, PO Box 8888, downtown station, Montréal (Qc) H3C 3P8, Canada

hafedh.mili@uqam.ca

Abstract

The separation of concerns, as a conceptual tool, enables us to manage the complexity of the software systems that we develop. There have been a number of approaches aimed at modularizing software around the natural boundaries of the various concerns, including subject-oriented programming [Harrison & Ossher, 93], composition filters [Aksit & Bergmans, 1992], aspect-oriented programming [Kiczales et al., 97], our own view-oriented programming [Mili et al., 99-02], and many others. The growing body of experiences in using these approaches have identified a number of fundamental issues such as what is a concern, what is an aspect, which concerns are inherently separable, and which aspects are composable. To address these issues, we need to focus on the semantics of separation of concerns as opposed to the mechanics—and semantics—of aspect-oriented software development methods. We propose a conceptual framework based on a transformational view of software development. In particular, we distinguish between essential separability and inseparability, which characterize requirements, from accidental separability and inseparability, which characterize the realizations of those requirements.

1 Introduction

“Separation of concerns” is a general problem-solving idiom that enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. Underlying this idiom is the hope that the solutions to these subproblems can be composed relatively easily to yield a solution to the original problem. The history of programming languages may be seen as a perennial quest for modularisation boundaries that best map (back) to “natural modularisation boundaries” of requirements. Aspect-oriented software development methods are no different, and most of the research on AOSD has focused on the semantics of aspects and aspect composition, i.e. the solution domain, as opposed to the semantics of concerns and concern separation and composition, i.e. the problem domain. Yet, the early case studies have shown that

these conceptually elegant techniques weren’t intuitive to use (see [Kersten & Murphy, 99], [Kendall, 99], [Herrmann & Mezini, 00]). Further, a great number of users of these techniques were caught up in the “how-to” of language constructs, with no regard for the conceptual appropriateness of the AOSD technique for the problem at hand. Further, the various techniques seem to offer orthogonal, but equally useful constructs, with no clear guidelines as to which method is appropriate for which problem.

We believe that better understanding of the AOSD techniques will result from a characterization of, 1) the *input* of software development, and 2) the *process* of software development, to help characterize, if not identify, which concerns are separable, and which development steps are most likely to affect the separation (or separability) of the resulting artifacts. We propose a conceptual framework based on a transformational view of software development. In this context, all the requirements on a software product, be they functional (related to input/output relations) or otherwise (related to *how* the output is produced), are inputs in these transformations. These requirements fit into general areas, or *concerns*, which may end up embodied in separate or same artifacts. We distinguish *essential separability* and *inseparability*, which characterize *requirements*, from *accidental separability* and *inseparability*, which characterize the *realizations* of those requirements in development artifacts. Accidental inseparability can be remedied by better language design and user education. Accidental separability should even be discouraged as the conceptual complexity is often increased and not reduced, and maintenance of the resulting program is often made harder.

2 Understanding the separation of concerns problem

Design, in and of itself, is a very complicated cognitive task bringing to bear a host of knowledge types and sources and a myriad of problem solving skills [Dasgupta, 1991]. When the artifacts, themselves, are

complex, a number of the conceptual and methodological tools fall apart because of scalability problems. A number of researchers have shown that complexity is an *essential* property of design activities *in general*, due in part to the *inevitably incomplete formulation* of the problem, and in part to our inability to cope *simultaneously* with all of the constraints of a given problem (our *bounded rationality* [Simon, 1982]).

The *separation of concerns* technique is a general problem solving heuristic that consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation that, 1) they be composable, and 2) the resulting solution is nearly optimal. For this heuristic to yield satisfactory results, the concerns that we are trying to treat separately must be fairly independent, to start with, so that they don't interfere with each other. Further, the problem solving activity itself needs to yield solutions that are composable.

In this section, we try to define the separation of concerns problem for the case of software. In this case, the "problem" is a set of requirements, and the "problem solving" process is the software development process. We first start by characterizing the software development process. In section 2.2, we try frame the separation of concerns problem.

2.1 A transformational view of software development

Simply put, software development may be seen as the process of going from precise specifications of what is to be done (requirements), to precise specifications of how it is to be done. Dasgupta identified two kinds of requirements in any design problem, *empirical requirements*, which specify externally observable or empirically determinable qualities that are desired of the artifacts, and *conceptual requirements*, which specify adherence to a particular style [Dasgupta, 1991]. For the case of software, there are two kinds of externally observable qualities, *functionality*—the *what*—on one hand, and run-time behavior—the *how*, including performance, and the like. Accordingly, we see three major categories of requirements for software development:

- 1) *Requirements of functionality*. These requirements specify an input/output relationship. To satisfy these requirements, we need a function that takes an input/output relationship and returns a function that returns the output for a given input
- 2) *Run-time requirements*. These are requirements on run-time behavior such as

performance, distribution, the underlying machine (virtual or otherwise), etc.

- 3) *Requirements on the software artifacts*. These are requirements dealing with things such as modularity, reusability, choice of programming language, adherence to specific programming style, etc.

These correspond closely to the categories of architectural qualities identified by [Bass et al., 1998]. Describing a program using an executable specification languages may be seen as performing a first step of the design process, i.e. ensuring functionality. Later steps can worry about run-time behavior and artifact quality.

In practice, these three sets of requirements are addressed simultaneously. Further, except in new projects where a complete system is built from the ground up, new functionality often has to integrate into an existing architecture, which embodies a specific point in the design space that addresses a set of run-time and artifact requirements. However, for the purposes of our presentation, we will assume that the three major design dimensions are commutative; two design transformations T_1 and T_2 are said to be commutative if given D_i , the description of the software at step i , we have $T_2 \circ T_1 (D_i) = T_1 \circ T_2 (D_i)$ (see e.g. [Baxter, 1992]). With this mind, let us propose a first-cut description of software development.

Handling functional requirements: operationalizing requirements:

This first transformation handles functional requirements. Given a relation $R: A \times B$, we need to obtain a function $f: A \rightarrow B$, such that for all $a \in A$, $f(a) \in \text{Image}_R(a)$. We say that $f(\cdot)$ is an implementation of R . R describes the relationship that must exist between the input and the output; $f(\cdot)$ provides an effective procedure for computing the output, given the input. If $R(\cdot, \cdot)$ is not a *function* (i.e. some elements of A have more than one image), then $f(\cdot)$ picks one element. Automatic programming consists, to a great extent, of "automating the operationalization of requirements". This transformation may be described by a relation $OR: \{R\} \times \{f(\cdot)\}$ from the set of relations to the set of functions. Let \underline{R} be the set of relations and \underline{F} the set of functions. OR is thus a subset of $\underline{R} \times \underline{F}$. This relation may be known intensionally (as we just described), or extensionally (through exemplar pairs). Automating this step consists of finding a function $g: \underline{R} \rightarrow \underline{F}$ such that given a relation $R \in \underline{R}$, $g(R(\cdot, \cdot)) = f(\cdot)$ where $(R, f(\cdot)) \in OR$. We say that g is an implementation of OR .

Handling run-time requirements. These include *performance requirements* and *execution model*. These

requirements are handled differently from functional ones. Whereas the operationalization of requirements associates a requirement with *any* function that implements the requirement, here we are picky about the properties of such functions. For example, such functions have to be efficiently computed. Instead of the relation *OR* shown below, we now have a subrelation *EOR* (Efficient Operationalization of Requirements) such that $EOR \subseteq OR$, where $\text{Domain}(EOR) = \text{Domain}(OR)$, but $\text{Image}_{EOR}(R) \subseteq \text{Image}_{OR}(R)$. In other words, out of all the functions that implement *R*, we pick the ones that are efficient.

Issues related to the execution model include things such as distribution, synchronization, and security. This does not change the function that is computed but changes things about where the different pieces are executed and how¹. We can represent the execution of function *f*(.) as follows: $EX: \underline{F} \times I \times M \rightarrow O \times M$. *EX* takes as argument the function to be computed and its input, and produces its outputs, while modifying the state of the machine that executed it (*M*). Those are the side effects on the machine, and may involve things such as establishing or terminating connections, modifying the state of data on permanent storage, logging, collecting statistics, etc. If $f(i) = o$, then $EX(f(\cdot), i, s) = \langle o, s' \rangle$ where s' is the state of the machine after it has finished execution of function *f*(.). If *f*(.) is written in machine language, then *EX*(.) is the function implemented by the CPU (the hardware's fetch, load, execute cycle).

Generally speaking, *EX* is a composition of several functions. For example, with a virtual machine architecture, $EX(f(\cdot), i) = H(VM(f(\cdot), i, s)) = \langle o, s' \rangle$, where *H* is the hardware execution, and *VM* is the virtual machine. The virtual machine itself could consist of a set of layered (composed) services or parallel services. An example of layered services is $VM(f(\cdot), i) = VM_1 \circ VM_2(f(\cdot), i, s)$. An example of parallel services is represented as $\langle VM_1; VM_2 \rangle(f(\cdot), i, s)$ where *VM*₁ and *VM*₂ are two services that are performed in parallel but such that the end result is the pair $\langle o, s' \rangle$. It may be that $VM_1(f(\cdot), i, s) = o$, while $VM_2(f(\cdot), i, s) = s'$, or, provided that $s = \langle s_1, s_2 \rangle$, $VM_1(f(\cdot), i, \langle s_1, s_2 \rangle) = \langle o, \langle s'_1, s_2 \rangle \rangle$ and $VM_2(f(\cdot), i, \langle s_1, s_2 \rangle) = \langle s_1, s'_2 \rangle$. In other words, *VM*₁ and *VM*₂ modify different parts of the state of the executing machine. The output itself may be computed by one or two of the virtual machines.

¹ Some aspects of security may be represented as functional requirements: adding the requester (another program or a user-id) as an input parameter..

Handling requirements on the artifacts. This involves taking into account the packaging of the function *f*(.) based on a number of criteria, including a reasonable division of labor, reusability, cohesion and coupling of the resulting modules, etc. It also includes things such as the choice of a programming language, programming style, etc. With object orientation, we end up implementing more than we need to for any particular use: classes are supposed to accommodate the needs of an application domain, but any application may use only a subset of that. Reuse (code sharing) happens by breaking down functions in such a way that the same sub-functions appear in two different functions (or systems). Let us take a problem *R*(.), and its realization, some function *f*(.). Idem for a problem *R'*(.) with realization *f'*(.). If we can write $f = f_{post} \circ g \circ f_{pre}$ and $f' = f'_{post} \circ g \circ f'_{pre}$, then we reduce the amount of new code to be developed².

2.2 Framing the separation of concerns problem

For the purposes of our discussion, we define a *concern* as a set of related requirements. *The basic premise of separation of concerns approaches is that requirements have nice properties, and to the extent that we can associate artifacts with concerns, we would like the artifacts to have similar properties!* Precisely, the “separation of concerns” methods rely on the existence of a development homomorphism such as the one illustrated in Figure 1. Assume that requirements are represented by predicates, and let $A_p = OR(P(\cdot))$ be the artifact that corresponds to predicate *P*(.). Development (represented by the thick arrow) is a homomorphism if there exists an operator \oplus defined on artifacts such that $OR(P(\cdot) \wedge Q(\cdot)) \equiv OR(P(\cdot)) \oplus OR(Q(\cdot))$.

We have some intuitions about cases where this homomorphism between requirements and artifacts holds. For example, given two requirements defined by relations $R_1: A \rightarrow B$, and $R_2: B \rightarrow C$, we know of several operators \oplus such that $OR(R_2 \circ R_1) \equiv OR(R_1) \oplus OR(R_2)$. For example, if the implementation adopts the call-and-return style, the operator \oplus consists of the call relationship between procedures. If the publish-and-subscribe style is used, the operator \oplus consists of registering $OR(R_1)$ as a publisher of some message, and $OR(R_2)$ as a subscriber to that message.

² Developing with reuse may be framed as follows: given a desired function *f*(.), and an available library function *g*(.), find two functions *h*(.) and *i*(.) such that $f(\cdot) = h(\cdot) \circ g(\cdot) \circ i(\cdot)$, and such that $\text{dev_cost}(h(\cdot)) + \text{dev_cost}(i(\cdot)) \leq \text{dev_cost}(f(\cdot))$.

If we view requirements as predicates on the solution,

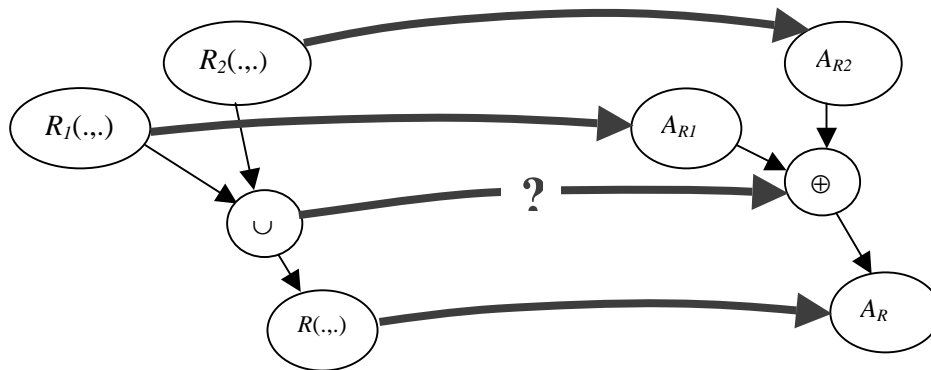


Figure 1. Development is a homomorphism from requirements to artifacts.

The advantages of this homomorphism include reusability, configurability, and separate maintenance. A number of object-oriented programming constructs and design idioms may be seen in this light. The new generation of separation of concerns techniques may be seen as defining new modularization boundaries for requirements, that are different from the ones afforded by regular object-oriented programming, and that are *realizable* in artifacts that are *composable* according to some composition operator. For example, OORAM uses *role models* [Reenskaugh, 1995] as new behavioral modules, and *role synthesis* to compose role models. Subject-oriented programming defined *subjects* [Harrison & Ossher, 93] as new modular structures, and *subject composition*, as a composition mechanism [Ossher et al., 96]. Aspect oriented programming defines *aspects* as new module boundaries, and *aspect-weaving* as a way of composing aspects with regular classes [Kiczales et al., 97]. Our own view-oriented programming uses *viewpoints* as a way of representing domain-independent business processes, and *view instantiation* and *attachment* as a way of adding that behavior to objects [Mili et al., 99],[Mili et al., 02].

Notwithstanding the case of OORAM, where the emphasis is on requirements level separation (*role models*) and composition, the other approaches have focused on the mechanics of artifact composition, sometimes losing sight of, 1) the requirements that these artifacts are supposed to embody, and 2) whether that composition (or separation) makes sense, from a requirements point of view. Further, even in those cases where AO techniques seemed appropriate, there were sometimes better non-aspect oriented solutions (see e.g. [Robillard & Murphy,2001]).

then requirements are clearly composable using logical composition (\wedge)—whether the resulting conjunction has solutions or not. However, for the homomorphism of Figure 1 to hold:

- 1) the composed requirements have to be independent, and
- 2) the development transformations have to preserve such independence so that the resulting artefacts may be combined.

In the next section, we try to characterize both conditions.

3 Characterizing the separability of requirements

In this section, we attempt the overly ambitious goal of answering two dual questions:

- 1) Given two requirements, under what conditions can they be “developed” separately, and can their realizations (aspects) be composed at will. The answer to this question will help determine the *domain* or *operating range* of the development homomorphism we illustrated in Figure 1. We refer to this problem as the *composability of requirement realizations*.
- 2) Given a realization that addresses several concerns, under what conditions can that realization be untangled into separable aspects, each of which addressing a subset of concerns. The answer to this question may help us assess which systems may be re-engineered in such a way that different concerns are addressed in separate—and readily reusable—aspects. We refer to this problem as the *separability of requirement realizations*.

In addition to its practical importance, an answer to the second question will also help us understand why case studies have not been as convincing as the textbook

cases that the original method authors have presented in support of their techniques:

We identified in section 2.1 three distinct kinds of requirements, requirements of functionality, run-time requirements, and requirements on the software artifacts themselves. Does it even make sense to try to address the composability of requirement realizations when talking about one functional requirement, and one requirement on the software artefacts. We should address the issue of whether run-time requirements even have realizations, before we concern ourselves with composing them. Accordingly, we start our discussion by first characterizing the ways in which requirements in each category are handled (individually). We will argue that run-time requirements can be represented as functional requirements on the virtual machine; requirements on artefacts are more difficult to formalize.

In section 3.2 looks at the *composability of requirement realizations* problem for the case of functional requirements. We examine the problem from a purely mathematical point of view, reducing the separability of two requirements, seen as (input,output) relations, to conditions on their domains and ranges. This will enable us to address composability issues between runtime requirements or between functional requirements, but not between a functional requirement and a run-time requirement, say³; this kind of (cross-type) composition will be discussed in section 4.

Section 3.3 tries to answer the *separability of requirement realizations* for functional requirements by looking at the problem of decomposing a function into separate sub-functions. We look at a range of decomposition/recomposition operators with different semantics preserving properties.

3.1 Handling the different types of requirements

3.1.1 Handling run-time requirements

We consider run-time requirements to be functional requirements on an imaginary virtual machine that will execute the program in the context of the real machine. The virtual machine will add a number of services including distribution, persistence, security, and others.

Persistence services may be seen as providing the program with an execution environment (a virtual

³ This is something that AspectJ is presumably well-suited for, but that also explains some of its weaknesses.

machine) that persists automatically the objects that the program manipulates. Most object-oriented databases operate this way (Versant, ObjectStore): developers write programs that manipulate persistent objects in a seamless fashion. It is as if databases come with their own run-time object model, built on top of the host language object model. We later see how this is actually implemented—interestingly, a limited form of aspect-oriented programming.

Distribution is similar to persistence in principle. Lest we oversimplify, distribution may be seen as providing a virtual machine whose run-time representation of objects accommodates remote objects, with what that implies in terms of referencing and in terms of method invocation. Consider the following CORBA or RMI-like code sequence:

```
Bank bank =
naming.bind( "//www.mycompagny.com/mybusinessdomain/bank23" );
Client cl =
bank.getCustomer( "JohnDoe234" );
String address =
cl.getAddress();
```

Notwithstanding the first line, which suggests the use of a naming service, the subsequent lines are indifferent from the location of the objects. We could imagine the same program being run in local mode, where the default Java virtual machine run-time representation of objects is used, and “a distributed Java virtual machine” that uses a level of indirection for run-time object representation to access remote objects, and that invokes an ORB to execute methods. Existing implementations of distribution use a slightly different implementation but the idea is the same.

The way distribution and persistence have been commonly implemented present some commonality. Transparency to the developer dictates a virtual machine metaphor. However, both techniques instrument user code with service-specific code that invokes those services (persistence or remote access). With Java-style persistence (e.g. ObjectStore), the code that is injected is added directly to the compiled Java bytecodes. With distribution, the IDL compiler injects, along with user code, code that is meant to be executed by the distribution virtual machine.

The same can be said about some aspects of security. Both authentication and encryption can be easily (and naturally) implemented at the virtual machine level: one involves encrypting exchanged data (through method calls), and the other authenticates the caller. In fact, Java’s own security model is supported and enforced by

the virtual machine. J2EE's security model is enforced by the containers—a higher level yet virtual machine.

One reason why virtual machine-like implementations of these services are not common—with the exception of security, for which we want no loopholes—is performance. The other is selectivity: because these services involve an overhead, if we embed it in the virtual machine, then all objects will use it, whether they need it or not. With this code injection mechanism, the code will only be injected in those objects/classes that need it.

As mentioned above, common implementations of persistence use a variant of aspect oriented programming: persistence code is added into designated class files (typically specified in configuration files) so that object creation, accessing, and modification accesses the database client. The same is true for distribution, where client-side stubs (proxies) go through the ORB to get the data they need.

Viewing run-time requirements as functional requirements on the virtual machine helps us understand which services are separable and/or composable, *in principle*, and also helps us understand which solutions are feasible under which situations, and understand some of the anomalies that arise from composing virtual machine-level services.

3.1.2 Handling requirements on the artifacts

Requirements on the artifacts deal with development-time “abilities”, with no regard for functionality or performance. Such requirements include understandability, reusability, maintainability, etc. Let $R(\cdot)$ be a functional requirement, and $f(\cdot)$ be an operationalization of $R(\cdot)$, i.e. $f(\cdot) \in OR(R(\cdot))$. The various “abilities” on the artifacts can typically be written as constraints on various metrics on the artifacts, such as:

- $M_i(f(\cdot)) = \text{MIN}_{g \in OR(R(\cdot))} (M_i(g(\cdot)))$ (relative constraint) or
- $M_i(f(\cdot)) \leq \alpha$, for some constant α (absolute constraint)

These *meta-level* constraints determine the packaging of the functionality⁴.

⁴ Normally, we would distinguish between algorithms/procedures and packaging. Algorithms determine how outputs are produced from inputs, and determine performance. Packaging puts boundaries around parts of the algorithm to promote various qualities.

Separation of concerns is a requirement on software artifacts that is being addressed with AOSD techniques. Thus, our discussion of how development affects separation of concerns will be limited to the development activities related to accommodating functional requirements and those related to handling run-time requirements.

3.2 Composable requirements

Given a development transformation T , we consider two requirements R_1 and R_2 to be T -composable if:

- 1) we can associate separate realizations to them ($T(R_1)$ and $T(R_2)$), and
- 2) there exists a composition operator \otimes on their realizations that satisfies them both, i.e. $T(R_1 \wedge R_2) = T(R_1) \otimes T(R_2)$

We showed in section 2.1 that functional requirements are transformed using an *operationalization operator*— OR , turning an input-output relation into a *function* that produces the output given the input. Having argued in section 3.1.1 that run-time requirements are nothing but functional requirements on the virtual machine, we look at the problem of composing two functional requirements through the operationalization operator.

We would like the operationalization of functional requirements to be additive at least in those cases where the two requirements have disjoint domains. Consider two relations R and R' such that $\text{Domain}(R) \cap \text{Domain}(R') = \Phi$. The simplest way of implementing $R \cup R'$ is by taking $f(\cdot) \oplus f'(\cdot)$, where $f(\cdot) \oplus f'(\cdot) = g(x)$ such that:

$$g(x) = f(x), \text{ if } x \in \text{Domain}(R) \\ = f'(x), \text{ if } x \in \text{Domain}(R')$$

In other words, the simplest $OR(\cdot)$ would behave as follows:

$$OR(R \cup R') = f(\cdot) \oplus f'(\cdot)$$

Note that if we take into account reuse, then we may be able to write $f = f_{post} \circ g \circ f_{pre}$ and $f' = f'_{post} \circ g \circ f'_{pre}$. We do have $\text{Domain}(f'_{pre}) = \text{Domain}(f')$ and $\text{Domain}(f_{pre}) = \text{Domain}(f)$, and thus $\text{Domain}(f_{pre}) \cap \text{Domain}(f'_{pre}) = \Phi$, but we don't know whether $\text{Domain}(f_{post})$ and $\text{Domain}(f'_{post})$ are disjoint, and we can't write $OR(R \cup R')$ (or $f(\cdot) \oplus f'(\cdot)$) as $[f_{post}(\cdot) \oplus f'_{post}(\cdot)] \circ g \circ [f_{pre}(\cdot) \oplus f'_{pre}(\cdot)]$.

If the relations have intersecting domains, we can define them as follows: $R = R_1 \cup R_2$ and $R' = R'_1 \cup R'_2$ such that: $\text{Domain}(R_1) = \text{Domain}(R) - \text{Domain}(R')$, $\text{Domain}(R'_1) = \text{Domain}(R') - \text{Domain}(R)$, and $\text{Domain}(R_2) = \text{Domain}(R'_2) = \text{Domain}(R) \cap$

Domain(R'). In this case, the relation to implement is $R_1 \cup R'_1 \cup (R_2 \cup R'_2)$, where R_1 , R'_1 , and $R_2 \cup R'_2$ have mutually disjoint domains. Thus, we have $OR(R_1 \cup R'_1 \cup (R_2 \cup R'_2)) = OR(R_1) \oplus OR(R_2) \oplus OR(R_2 \cup R'_2)$.

This relationship is trivially satisfied in case $R_2 = R'_2$. This is the ideal case in the sense that both requirements agree on what the output should be for the same inputs. In that case, the two requirements (R_1 and R_2) may be seen as two restrictions of the same relationship defined on the domain $\text{Domain}(R_1) \cup \text{Domain}(R'_1)$. If the two relationships disagree on the output, then we have a problem. We see two levels of disagreement. The first level of disagreement is illustrated in the following example. Consider the two relations $R_1 = \{ (x,y) \mid 0 < x < 100, \text{ and } x^2 = y \}$ and $R_2 = \{ (x,y) \mid 50 < x < 150, \text{ and } x^2 = y \}$. The intersection of the two domains consists of the interval $[50..100]$. If both the realizations of R_1 and R_2 use the positive square root of x —or both use the negative square root—then we are fine. If they use different square roots, then we have a problem. This incompatibility is due to an inconsistent choice of

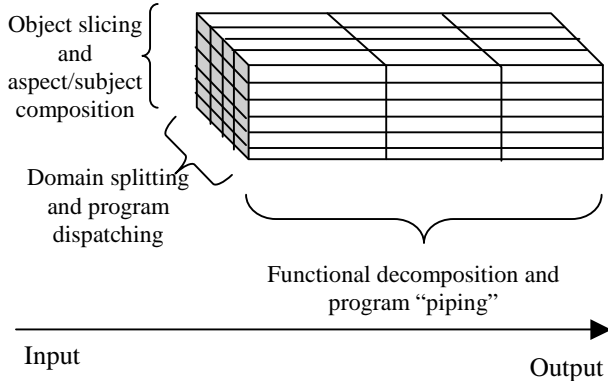


Figure 2. Comparing three decomposition paradigms

realizations, and is a common and acceptable course of action. Intuitively, what we need in this case to make sure that we use consistent realizations. This is not unlike the problem of choosing consistent specializations when we instantiate a framework, i.e. the kind of situations for which things such as the *factory pattern* is applicable.

The second level of disagreement is the case where the requirements themselves disagree, i.e.:

$$\exists x \in \text{Domain}(R_1) \cap \text{Domain}(R_2) \text{ s.t. } R_1(x) \neq R_2(x)$$

In our view, this is not a case for separation of concerns methods to handle: the requirements disagree, so there is no point in trying to compose the artifacts.

3.3 Separable requirements

Given a development transformation T , we consider a requirement R (an element of the domain of T) to be *T-separable* if there exist, 1) two requirements R_1 and R_2 , 2) a composition operator \bullet defined on the domain of T —the requirements—and, 3) a composition operator \otimes on the image of T —the *artifacts*—such that:

- 1) $R = R_1 \bullet R_2$
- 2) $T(R) = T(R_1) \otimes T(R_2)$

This is nothing but the good old divide-and-conquer analytical development paradigm. With structured analysis and design (and programming), the operator is functional composition, in the mathematical sense, and \otimes is “piping”, in the programming sense (the output of a program or procedure is used as an input to the other). Functional decomposition is not only useful for reducing complexity, it is also useful for reuse.

Another valuable pair of operators corresponds to the combination of domain splitting and dispatching. Consider the requirement R where $\text{domain}(R) = D = D_1 \oplus D_2$ —the symbol \oplus referring to disjoint union (partition). Let T be the operationalization of requirements ($OR(\cdot)$), and $R_1 = R|D_1$, and $R_2 = R|D_2$. Then:

$$OR(R(\cdot)) = \begin{cases} \text{if } x \in D_1 \text{ call } OR(R_1) \\ \text{if } x \in D_2 \text{ call } OR(R_2) \end{cases}$$

We are all familiar with these two techniques, and have used them—and should continue to do so—to good measure. Aspect-oriented development techniques advocate *other* pairs of decompose/recompose or split/join operators which are specific to the object-oriented context. These new pairs of operators operate simultaneously on functions and data, along the lines of object or class hierarchy slicing (see e.g. [Tip et al.,1996]). In this case, instead of considering the input domain (D) as consisting of simple value, we consider it as a tuple (of state variables), and functions (object methods) may operate on various “sub-tuples”.

Figure 2 illustrates the three decomposition paradigms. For each paradigm, we mention the decomposition technique used on requirements, and the corresponding composition technique used on the corresponding artifacts. In the next section, we look more closely at the problem of sliceability of requirements. We start with a strict definition of sliceability which supports unrestricted (commutative) recomposition of the artifacts. We then propose a weaker form of sliceability which requires an ordered (non-commutative) recomposition.

3.3.1 Sliceability

Let $R \subseteq A \times B$, let $f(\cdot) = OR(R)$, and assume that $A = S_1 \times S_2 \times \dots \times S_i \times S_{i+1} \times \dots \times S_n \times I$ and $B = S_1 \times S_2 \times \dots \times S_i \times S_{i+1} \times \dots \times S_n \times O$. We say that R (or f) is *sliceable* if there exist two functions $f_1(x_1, \dots, x_i, i)$ et $f_2(x_{i+1}, \dots, x_n, i)$ such that $f(x_1, \dots, x_i, x_{i+1}, \dots, x_n, i) = f_1(x_1, \dots, x_i, i) \bullet f_2(x_{i+1}, \dots, x_n, i)$. In other words, the function f can be computed as the concatenation of two functions.

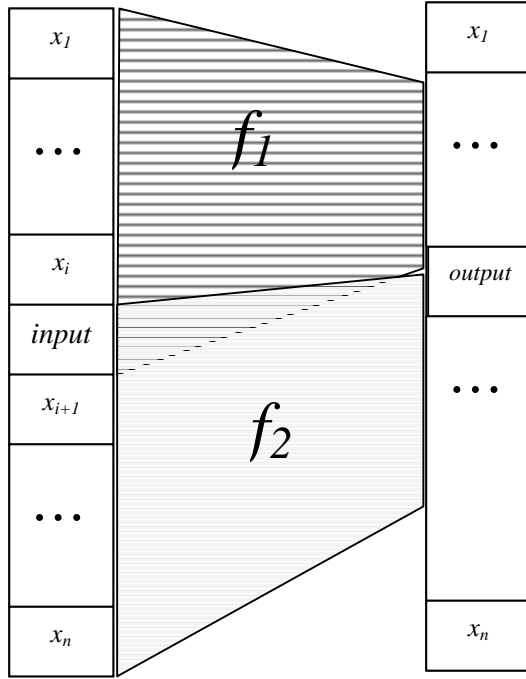


Figure 3. A function is *sliceable* if it can be written as the concatenation of two functions that access and modify disjoint parts of an object

The idea of sliceability is related to the idea that a relation may be written as a subset of the product of two relations. For example, let R_1 and R_2 be two binary relations. We can define the relation $R_1 \times R_2$ as follows: $\langle x_1, x_2, y_1, y_2 \rangle \in R_1 \times R_2$ if and only if $\langle x_1, y_1 \rangle \in R_1$ and $\langle x_2, y_2 \rangle \in R_2$.

Intuitively, the sliceability of a corresponds to the case where we have two functions that take the same input and that use and modify different parts of an object, i.e. they correspond to two disjoint slices of the same data (or object). Sliceable functions can be put together, with no problem. Notice that we require that both functions take the input (which may be either a real input or a method selector), and that the output is produced between them. In the context of an object-oriented program, if we have a method that returns void but

modifies the state of the object, then each subfunction will have modified its slice. If the function returns a value, then we might be able to find a subset of state variables based on which the output is computed, and the slice may be made along that. Note, however, that not all relations/functions are sliceable. A function that averages the state variables will not be sliceable⁵.

Subject-oriented programming (and hyperspaces) works best with this ideal case in mind. Problematic cases occur when the sliceability hypothesis fails. Interestingly, the broken delegation problem can be understood in terms of sliceability of functions. Broken delegation happens when a function that occurs on one side (i.e. in a single object fragment) calls a separable function that occurs on several object fragments (see e.g. [Bardou & Dony, 1996]): the result is no longer separable.

3.3.2 Effective sliceability

Let $R \subseteq A \times B$, let $f(\cdot) = OR(R)$, and assume that $A = S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times I$ and $B = S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times O$. Let $f(\dots)$ be a function that implements R . Let $f_1(\dots)$ and $f_2(\dots)$ be two functions with domains $S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times I$. If $f(x_1, \dots, x_i, x_{i+1}, \dots, x_n, i) = \langle x'_1, \dots, x'_i, x'_{i+1}, \dots, x'_n, o \rangle$, we use the notation $f|_i$ to refer to the projection of f over the set S_i , i.e., $f|_i(x_1, \dots, x_i, x_{i+1}, \dots, x_n, i) = x'_i$. Similarly, we define $f|_j$ as the projection of f over the set $S = S_i \times \dots \times S_j$ for some i and j . Let $Ref(f)$ be the set of variables used in the computation of $f(\dots)$ and $Mod(f)$ be the set of variables modified by f , i.e. the set of variables $\{x_i\}_i$ such that $f|_i(x_1, \dots, x_i, x_{i+1}, \dots, x_n, i) = x'_i \neq x_i$. A function $f(\dots)$ is said to be *effectively sliceable* if and only if there exist two functions $f_1(x_1, \dots, x_n, i)$ and $f_2(x_1, \dots, x_n, i)$ such that:

$$Mod(f_1) \cap Ref(f_2) = \Phi$$

$$Mod(f_2) \cap Ref(f_1) = \Phi$$

$$Mod(f_1) \cap Mod(f_2) = \Phi$$

$$Mod(f_1) \cup Mod(f_2) = Mod(f)$$

$$f|_{Mod(f)}(x_1, \dots, x_n, i) = f_1|_{Mod(f_1)}(x_1, \dots, x_n, i) \bullet f_2|_{Mod(f_2)}(x_1, \dots, x_n, i), \text{ and}$$

$$f|_o(x_1, \dots, x_n, i) = f_1|_o(x_1, \dots, x_n, i) \bullet f_2|_o(x_1, \dots, x_n, i)$$

for some ordering of the state variables x_1, \dots, x_n . Figure 4 illustrates the first three equalities in a Venn Diagram.

Note that a *sliceable* function is also *effectively sliceable*. An interesting property of *effectively sliceable* functions is that the component functions may be executed in any sequence.

⁵ Intuitively, intensive functions (of the state variables) are not separable, whereas extensive functions are.

There are other cases of sliceability, but in this case, the subfunctions have to be executed in a particular order. We call this *temporal sliceability*. Temporal sliceability is a weaker condition than effective sliceability, and is described as follows. Let $R \subseteq A \times B$, let $f(\cdot) = OR(R)$, and assume that $A = S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times I$ and $B = S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times O$. Let $f(\dots)$ be a function that implements R . Let $f_1(\dots)$ and $f_2(\dots)$ be two functions with domains $S_1 \times \dots \times S_i \times \dots \times S_j \times \dots \times S_n \times I$. Using the same notation as above, we say that function $f(\dots)$ is said to be *temporally sliceable* if and only if there exist two functions $f_1(x_1, \dots, x_n, i)$ and $f_2(x_1, \dots, x_n, i)$ such that:

$$f|_{Mod(f)}(x_1, \dots, x_n, i) = f_1|_{Mod(f_1)-Mod(f_2)}(x_1, \dots, x_n, i) \bullet f_2|_{A-(Mod(f_1)-Mod(f_2))}(x_1, \dots, x_n, i).$$

$Mod(f_1) - Mod(f_2)$ represents the set of variables that are modified by f_1 but not by f_2 . Some of these variables may, however, be referenced by f_2 and we don't care about that. Obviously, the relationship between f_1 and f_2 is not a symmetrical one, and the functions have to be executed in a particular order.

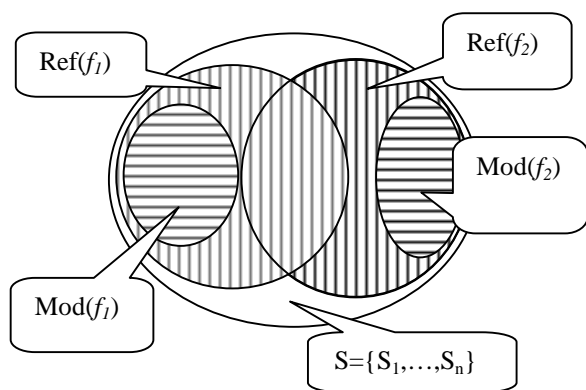


Figure 4. A function is *effectively sliceable* if it can be written as the concatenation of two functions that, 1) modify disjoint parts of an object, and 2) don't refer to the parts that the others modify

In [Mili, 1996], we showed that provided that methods of objects do not modify objects other than the executing objects or their components, any method that computes a function and modifies the receiver object can be decomposed into a sequence of pure functional and purely side-effectual functions. To compose two hybrid functions, we can decompose them along the purely functional versus purely side-effectual dimensions, find the smallest granularity decomposition between the two, and then compose them slice-by-slice.

The major problem, of course, is our tendency to code “service-oriented functions”, i.e. functions that are application level but that are coded at the domain class

level. These functions are not composable because they address an application specific need, each. You would want to compose them because they embody a general behavior that is not encapsulated elsewhere. Obviously, not choosing the right granularity is a problem, and leads to methods that are not composable.

4 Discussion

This is a very preliminary investigation into the principles of separation of concerns and the foundations of the techniques that promote separation of concerns. The yardstick by which innovations in software engineering are to be assessed has always been—and rightly so—to determine the problem that a given method, technique, or tool, solves. Separation of concerns is only useful to the extent that once the concerns have been addressed separately, we are able to re-combine the individual and partial solutions into one that addresses all of them.

Some of the case studies that are available in the literature show cases where concern separation is difficult in practice [Kersten & Murphy, 99], [Kendall, 99], [Herrmann & Mezini, 00]. Others showed that *aspect/subject* composition is difficult, even in cases where the aspects or subjects embody distinctly different concerns [Mili et al., 96], [Kersten & Murphy, 99], [Murphy et al., 01]. Others yet have reminded us of simple solutions to separation of concern problems (see e.g. [Robillard et al., '99]).

We attempted to frame the separation of concerns in software development in terms of homomorphisms of development transformations, and then we tried to determine the “operating range” of these homomorphisms. This preliminary work raised more questions than it answered, and some of the answers are reassuringly common-sensical, but are worth stating:

- Not all requirements (concerns) are composable in the sense that they lead to composable artifacts. Viewing requirements as input-output relations, we identified simple conditions on the domains and images of these requirements, which essentially say that the requirements should not be conflicting. In particular, method cancellation through subject composition or aspect weaving is no less dangerous than with inheritance: they are both a sign of either a violation of intent, or of sloppy realization (implementation).
- We should treat aspects that embody run-time requirements differently—and separately—from aspects that embody functional (domain) requirements. We framed run-time

requirements (persistence, fault-tolerance, etc.) in terms of *functional requirements of the virtual machine*. In an ideal world, such concerns should also be handled by virtual machine—or more generally, meta-level— aspects. However, performance considerations may suggest otherwise at the risk of inducing composability problems⁶.

- Not all programs that implement several concerns can or should be re-engineered into separate aspects. The underlying concerns/requirements may not be separable (essential inseparability), or the current implementation may not lend itself to such a separation (accidental inseparability). Object slicing can help with accidental inseparability.

This work is in its infancy. We have started to take a closer look at the existing AOSD methods and the case studies to judge the usefulness of the above framework. We were able to explain known difficulties with subject-oriented composition (see e.g. [Ossher et al.,95-97], [Mili et al.,96]) and view attachment [Mili et al., 99-02] in terms of violations of some of the principles outlined above. More work is needed with the other methods and the case studies to ascertain the usefulness of our framework.

5 Bibliography

[Aksit et al., 1992] M. Aksit, L. Bergmans, and L. Vural, “An object-oriented language-database integration model: the composition filters approach”, in *Proc. of ECOOP 92*, Springer Verlag 1992.
[Bardou & Dony, 1996] D. Bardou & C. Dony, “Split objects: a disciplined use of delegation within objects,” in *proc. OOPSLA’96*, pp. 122-137, 1996.
[Bass et al.,1998] L. Bass, P. Clements & R. Kazman, *Software Architecture in Practice*, 1998.
[Baxter, 1992] I. Baxter, “Design Maintenance Systems,” *CACM*, vol. 35, no. 4, April 1992, pp. 73-89
[Dasgupta, 1991] S. Dasgupta, “The Nature of Design Problems,” in *Design Theory and Computer Science*, Cambridge University Press, 1991, pp. 13-35.
[Harrison & Ossher, 93] W. Harrison & H.Ossher, “Subject-oriented programming: a critique of pure objects,” in *Proc. OOPSLA’93*, pp. 411-428.

[Herrman & Mezini, 2000] S. Herrmann & M. Mezini, “On the Need for a Unified MDSOC Model: Experiences from Constructing a Modular Software Engineering Environment”, MSDOC workshop, OOPSLA’2000
[Kendall, 1999] E. A. Kendall. *Role Model Designs and Implementations with Aspect Oriented Programming*. In *Proc. OOPSLA’99*, pages 353-369, Denver, Co., 1999
[Kersten & Murphy, 1999] M. Kersten & G. Murphy, “Atlas: a case study in building a Web-based learning environment using aspect-oriented programming”, in *proc. OOPSLA’99*, pp. 340-352, 1999.
[Kiczales et al., 1997] G. Kiczales, J. Lamping, C. Lopez, “Aspect-Oriented Programming,” in *Proc. ECOOP’97*.
[Mili, 1996] H. Mili, “On behavioral descriptions in object-oriented modeling”, *Journal of Systems and Software*, summer 1996.
[Mili et al.,99] H. Mili, A. Mili, J. Dargham, O. Cherkaoui & R. Godin, "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," *Proceedings of TOOLS USA '99*, Aug. 1-5, 1999, Prentice-Hall, pp. 211-221
[Mili et al., 2002] H. Mili, H. Mcheick & J. Dargham, “CorbaViews: Distributing objects with several functional aspects,” *Journal of Object Technology*, August 2002,.
[Ossher 96] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, “Specifying subject-oriented composition,” in *TAPOS*, 2(3), 1996.
[Robillard & Murphy, 2001] M. Robillard & G. Murphy, “Analyzing Concerns Using Class Member Dependencies,” Position paper for the ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering, 2001.
[Simon, 1982] H. A. Simon, *Models of bounded rationality*, vol. 2, Cambridge, MA (MIT Press, 1982).
[Reenskaugh, 1995] Trygve Reenskaugh, in *Working with Objects*, Prentice-Hall, 1995
[Sullivan, 2001] G. T. Sullivan, “Aspect-Oriented Programming Using Reflection and Metaobject Protocols,” *CACM*, Oct. 2001, 44 (10), pp. 95-97
[Tip et al., 1996] F. Tip, J-D Choi, J. Field, and G. Ramalingam, “Slicing class hierarchies in C++”, In *Proc. OOPSLA’96*, San Jose, CA, pp. 179-197.

⁶ With aspectJ, if we have two aspects, one that adds a function, and one that traces all functions, the order of aspect weaving is important...the thing is, the trace aspect is meta-level: it is a functional aspect of the virtual machine (to execute methods and produce a log) defined *intensionally* for all methods. But by making it a user-level aspect, we define it *extensionally*, i.e. on all the *currently defined/available* user functions, and hence, the order dependency!