Separation of concerns and typing: a first stab

Hafedh Mili, Joumana Dargham, and Salah Bendelloul
Département d'Informatique
Université du Québec à Montréal
Case Postale 8888, Station Centre-Ville
Montréal, Québec H3C 3P8
Canada

Abstract

We use software of concerns in software engineering for two reasons: 1) to manage the complexity of the systems we try to build, and 2) to delay, as far as we can, the binding of the various pieces/concerns of the system to each other, so that they may be developed, tested, and maintained separately. We start by positioning various approaches along the binding time dimension, going from modeling to run-time binding. Next, we examine the possibility of performing run-time binding and its implications on type-safety.

1. Separation of concerns and binding time

Much progress in software packaging technologies has dealt with the separation of concerns and the delaying of the binding of the concerns to each other. The separation of concerns can be sustained as far as:

- *Requirements analysis time*: a number of methodologies recognize that the first step in eliciting the requirements of a system consists of indentifying the different usage scenarios/use cases (e.g. [Martin & Odell, 1992], [Jacobson et al., 1992]). This separation is useful for managing complexity, and to avoid including things that are not needed.
- *Modeling (analysis) time*: the OORAM methodoloy builds a model of an application in terms of a combination of *role models*, which are partial models describing specific collaborations between application entities [Reenskaugh,1995]. However, *role models* are combined before we move on to design and coding,
- *Design time*: typically, efforts that aim at keeping concerns separate through design, aim at separating design-level concerns from functional concerns. The idea here is to not design classes one by one, but to design class patterns/templates, and then map/instantiate those patterns/templates to analysis-level specifications of classes. This assumes that the analysis-level specifications are *complete* or *executable* in some language on some *virtual machine* and that design aims simply at satisfying non-functional requirements. Approaches based on this paradigm include [Mili & Li, 1993], the Shlaer& Mellor methodology and its accompagnying tool set [Mellor, 1992], and more generally, the combination of methodologies/tools that generate code from analysis-level specifications[1]. Approaches based on application-specific specification languages may be seen in this

1. The "design concern" may not be as easily evolvable, as it may be hardcoded in the CASE tool.

light, as well,

- *Coding time*: here we have to make a distinction between the code that implements the concerns and the code that uses them. A number of approaches have been proposed in the past decade that support the separate *coding of* concerns. These include subject-oriented programming (SOP) [Harrison/Ossher, 1993-96], aspect-oriented programming (AOP) [Kiczales et al., 1996], role components [VanHilst & Nokin, 1996], the GenVoca family of component generators [Batory et al., 1993-97], and view programming [Mili et al., 1999]. The integration of these concerns is evenly split between coding time and compile-time: in some ways, SOP and AOP integrate concerns automatically at compilation time (integration is a pre-process of compilation), but client code need not be aware of the fact that several concerns have been integrated. With GenVoca [Batory et al, 93-97] and role components [VanHilst & Notkin, 96], the integration is programmed in the user code as part of the *object declaration* process. With views, integration is part of the *object manipulation process*.
- *Run-time*: with regard to the concerns themselves, this means that the object code that corresponds to the various concerns is separate. It may mean in different objects, or in different memory spaces, altogether. This does not preclude communication from taking place. With regard to the code that uses or integrates the concerns, this means some measure of reflective facilities that enable users to activate different combinations of concerns during run-time.

With AOP, SOP, role components, and Genvoca approach, by the time we compile, we have vanilla flavour OO code. The Genvoca and role components approach [VanHilst & Notkin, 96] are somewhat based on the instantiation of aggregate templates. During run-time, the code that corresponds to the various concerns runs on the same object, for the case of SOP, AOP (except for the case of dynamic associations), and for role components. Genvoca, which may be seen as delegation/aggregation based, different objects execute different pieces. However, the various concerns are known at compile-time and will not evolve during the execution of the program. With view programming, the code that corresponds to the various concerns runs in different objects, and the set of concerns applicable to an object will be determined during time.

In the remainder of this paper, we look at typing issues from two perspectives: 1) from the perspective of code behaving in a coherent fashion with or without the addition of concerns, and 2) from the perspective of objects being able to handle the requests that are addressed to them.

2. Behavioural conformance

The very idea of separation of concerns is that the concerns being separated are not tightly related and the presence or absence of a certain concern has no or little effect on the other concerns. In other words, adding a concern should involve *conservative extension* of the same program, i.e. notwithstanding the new behaviour that was added, what used to work will continue to work in an identical or a qualitatively equivalent fashion. We divide this in two factors:

2.1 changing the interface

In what way does adding new concerns change the interface? the answer depends on the approach.

AOP and SOP: with both these approaches, the composition of subjects/weaving of aspects changes the interface (type) of classes. With SOP, the various subjects play a symmetrical role, and it is left to composition rules, in case of conflict (or coincidence) to prioritize subjects. With AOP, aspects are clearly add-ons, and local coincidence (e.g. introducing a method that exists already) is treated as a conflict (if not by the aspect weaver, at least by the compiler).

Role components adroitly generates versions of a class with additional responsibilities by simulating on-the-fly class creation through template instantiations. The new class, augmented with the new roles, will have a different name and will be a subclass of the original class. A similar approach is used in ***Genvoca*** where the final components are instantiations of predefined chains of template structures. With ***Genvoca***, the various roles are embedded in an aggregation hierarchy. With ***view programming***, the addition of concerns is aggregation-based, but doesn't change the (name of the) type of the receiver.

2.2 Behavioural composition

Changing the response of an object to an existing message isone of the basic premises/goals of ***SOP***: the object belonging to the composed subjects will respond according to the *combination* of the available subjects. Through the composition rules, a developer can use any combination of the available behaviours, but the important thing is that the developer does have those behaviour available. With ***AOP***, two mechanisms are used to change the response of objects to pre-defined messages: 1) advisories, and 2) dynamic associations (observers/constraints). However, it seems that the "introduce" operation can have undesirable behavioural side-effects:

- the introduction of a method that is defined in a superclass will override it even if it has different semantics (***doublecheck this****), and
- the introduction of new variables will override inherited ones[1].

Role components are "inheritance-aware", and aware of each other's existence, and hence, composition of behaviours is encoded in the roles themselves, through qualified references to inherited (other roles') behaviours[2]. The reliance on inheritance for composition forces a sequence of instantiation of the various role components [VanHilst & Notkin, 1996], possibly leading to splitting role components in case of cycles (***doublecheck***). With ***GenVoca***, the various components may be seen as layered services, and in some ways, generating a component will select a specific implementation for the same set of services. Thus, there is no real composition in the sense of having different components offer competing or complementary versions of the same behaviour. ***View programming***'s view on composition is similar to SOP and role components in the sense that the different subjects/roles/views provide their own versions of behaviours, and in the sense that the behaviour embodied in the various views may or may not rely on implementations available in other views/roles/subjects; unlike role components, delegation is not inheritance-based, and the views play a symmetrical role.

---

1. Java allows the redefinition of instance variables, which has the same semantics as method redefinition, except that the references to instance variables are statically bound.
2. A role component will look like
template<class WifeType,class SuperType> class *Husband*: public *Supertype* {....}
and a method within ***Husband*** might call upon the version of the same method inherited from the template parameter ***SuperType***.

3. Run-time composition of concerns and type safety

3.1 Issues

In the real world, objects change roles during their lifetime. From the time a person appears on the IRS records as a deductible expense, that person will keep changing roles until well beyond its death, regularly acquiring and relinquishing attributes and behaviour. Generally speaking, we need a mechanism for allowing objects to change behaviour during their lifetime, specifically when that change takes place within the *same* program run. Further, in the context of a distributed application, different sites/users may see different aspects of the same objects (different functionalities, different access rights/privileges, different quality of service parameters, etc.).

Allowing an object to change its behaviour in non-predictable ways during run-time is problematic. First, we have to make that behaviour somehow available on-demand. As we saw in section I, all of AOP, SOP, Genvoca and role components, integrate the various concerns/views during coding time by instantiating the appropriate classes, but the set of roles/subjects available to an object remain constant throughout the lifetime of the object. Additional constructs may be used to make those behaviours available on demand, as we show below.

Allowing objects to change their behaviour during run-time also *inevitably* compromises the type safety of programs in the sense of:

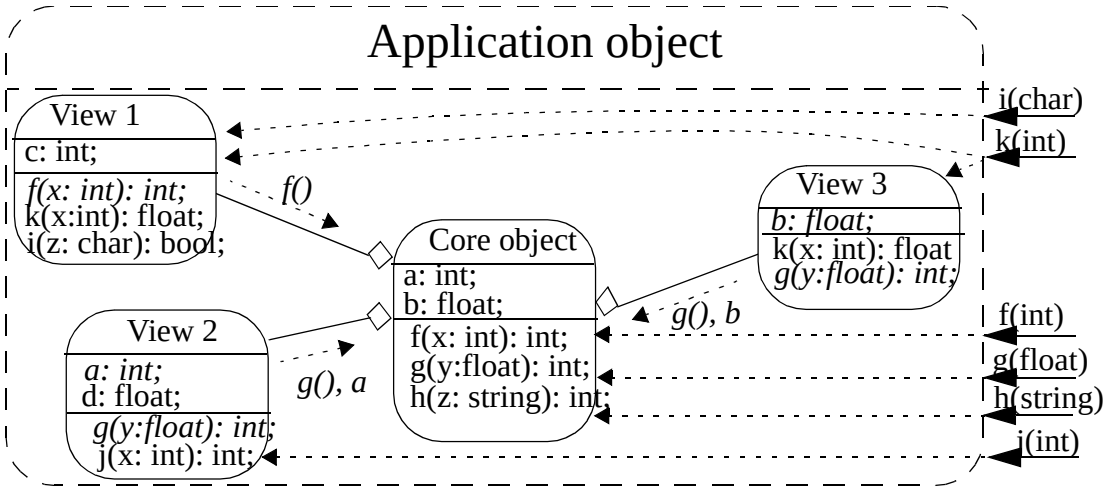
- precluding, or limiting the effect of static type checking, and
- opening the way for catastrophic behaviour in case the behaviour requested at any given point in time is not supported at that point

We call this *dynamic classification*. We should distinguish here between *dynamic typing*, which refers to the fact that a variable that is declared (compile-time) of a given type, will hold during *run-time*, an object of a different type, and *dynamic classification*, whereby the same object (same identity, same address) changes type/behaviour during run-time.

In terms of type checking, the best we can do, in statically typed languages, is to check that the behaviour requested of an object is *potentially* available to it—although we cannot ascertain that at that very moment in the computation, that behaviour is *actually* available; this is similar in principle to the "leap of faith" that C++ or Java programmers make when they downcast a variable declared of type *T* to some type *T'* which they believe is a type or a supertype of the object held by that variable at that point in the computation. With untyped/dynamically typed languages, there is nothing to do at compilation time.

In terms of invoking the proper combination of behaviours, depending on which roles/concerns are embodied in a given object, we have to implement some run-time dispatching method which will direct a message request to the method (or combination of methods) that is available to it at that time. This can be implemented in one of several ways, including:

- With typed languages, a combined interface that embodies all the potentially activated/attached concerns dispatches to specific private methods, depending on which concerns/roles are currently activated/available. This may be the approach used with C++-based or



Legend: - dotted arrows indicate delegation links.
- the resulting application object supports *all* the interfaces

Figure 1. A model of objects with views.

Java-based implementations,

- With dynamically typed, reflexive languages, we can modify the default method dispatching mechanism to direct a method call to the appropriate method combination. This method is appropriate for Smalltalk and CLOS (**I think**).

In the remainder of this section, we show how this is may be supported in view programming, SOP, and AOP.

3.2 Method dispatching with view programming

Figure 1 illustrates view programming's object model. An object is made up of a *core object* (core attributes and functionalities) and a time-varying set of *views* (roles, etc.). At any given point in time, the object supports the combined interface of all the views that are currently attached to it. Upon receiving a message, the object "somehow" figures out which methods are available to it at the time—based on the views currently attached— and then invokes the proper combination. In our C++ implementation, views are regular C++ classes which point to instances of the core class, and delegate to them some of their behaviour. Application objects are declared with the type of the core object, but different views may be attached and/or activated[1]through calls to an inherited method "attach(<view instance or view class name>)". In turn, developers are allowed to invoke the combined interface on the core object, even though the core class supports only a subset of the combined interface thanx to two mechanisms:

1. for each object, we create a special view that implements the combined interface—call it *composition view*—based on the set of views (view classes) available to the core class (list of #includes). The composition view implements the dispatching mechanism explained in section 3.1,
2. we built pre-processors that replace all behaviours invocations on the core object by delegated calls to the composition view[2].

Referring to Figure 1, if in the course of the program execution a call is made to method 'j(int)' at a point where view 2 is not attached, the implementation of 'j(int)' in the composition view will

1. an attached view that is not activated does not participate in behaviours, but preserves its state until it is activated again.

handle this any way we want, including simply by generating an exception saying in effect "behaviour not supported *at this time*".

We haven't studied the cases of Smalltalk and Java seriously, but one thing is certain: in Smalltalk, we would not need any code transformations, for two reasons: 1) the compiler does not check whether a method is supported by a given class, and 2) we can modify the language's basic method dispatching mechanism to implement the desired dispatching method.

3.3 Supporting run-time composition in SOP

At first hand, we can support run-time composition in SOP by doing the following: 1) supporting the activation and de-activation of subjects, and 2) generating different versions of the same methods corresponding to various combinations of active subjects, and 3) supporting the dispatching mechanism described earlier. With *n* subjects, the current configuration may be represented by a number between 1 and 2^n, and dispatch tables for each method could be 2^n long, with repeat entries. Notwithstanding the overhead in code size and dispatch time, we believe that keeping track of where things come from can be very tricky: it may not be possible or practical to deduce, from a single n-subject composition, what an m-subject composition might look like, for a subset of size m of those n subjects, because of all the degenerate (many to many) composition rules, for both classes, and for inheritance relationships. The problem here is that inheritance within subjects and composition across subjects interact. We don't have that problem with views which are generated "flat"[1]. If subject composition takes place on flattened subject hierarchies, doing this might be easier.

3.4 Supporting run-time aspect weaving in AOP

At first glance, dynamic associations may be triggered and inhibited at will, supporting some form of run-time weaving, provided that the objects involved in a dynamic associations need not store state specific to those associations. Those aspects that are woven into code using the "**introduce**" directive probably require similar mechanisms to run-time subject composition discussed above. Advisories appear to be fairly easy to compose in different combinations: the different advisories for the same method will keep their relative ordering, whichever subset is executed.

3.5 Supporting run-time role composition with role components

Role components rely *directly on inheritance* to compose roles, and dependencies between roles dictate the inheritance hierarchy of the various roles. At first glance, it does not appear to be an easy thing to do to get the behaviour that corresponds to a select subset of roles. Perhaps the only sub-compositions we can get are sub-chains of the role inheritance hierarchy that consist of prefixes of the entire chain (starting with the root class/role). We suspect that there might be more relaxed conditions (such as "definition completeness" of the subset of role components).

---

2. Because the code of view classes is generated by us (from templates parameterized by the core class we call *viewpoints*), we are able to handle the most common cases of the so-called "broken-delegation problem". In fact, broken delegation has become a security feature that we may turn on or off through a pre-processor switch [Mili & Dargham, 1998].

1. Viewpoints are organized in specialization/extension hierarchies, but the view classes generated from them make no reference to hierarchy.

3.6 Run-time composition of realms in Genvoca

This discussion does not appear to be relevant to Genvoca since the various realms/components correspond to layered services: for the components in any given realm, we need all the layers on top of which it was built. Perhaps the only thing we can do is to be able to peel off those layers in a dynamic fashion, by: 1) passing the objects around to functions that expect different types of parameters, and 2) using smart-pointer like constructors that will peel/wrap those objects while passing them around.

4. Summary

We use software of concerns to manage the complexity of the systems we try to build, and to delay, as far as we can, the binding of the various pieces/concerns of the system to each other, so that they may be developed, tested, and maintained separately. In this paper, we studied a number of approaches to separation of concerns from the perspective of binding time for the various concerns. Further, we identified an additional reason why separation of concerns is important: *dynamic classification* of objects. We discussed some of the issues raised by implementing dynamic classification, and explored to which it may be supported in the various approaches to separation of concerns.

References

- [Batory & Geraci, 1997] Don Batory and Bart Geraci, "Composition validation and subjectivity in GenVoca generators," *IEEE Transactions on Software Engineering*, vol. 23, no. 2, February 1997, pp. 67-82.
- [Harrison 93] William Harrison and Harold Ossher, ''Subject-oriented programming: a critique of pure objects,'' in *Proceedings of OOPSLA'93*, Washington D.C., Sept 26-Oct 1, 1993, pp. 411-428.
- [Kiczales & Lopez, 99] Gregory Kiczales and Cristina Lopez, ''Modularization revisited: aspects in the design and evolution of software systems,'' in *Tutorial notes*, TOOLS USA 99.
- [Martin & Odell, 1992] James Martin and James Odell, in *Object-Oriented Analysis and Design*, Prentice-Hall, 1992.
- [Mili & Li, 1993] Hafedh Mili and Haitao Li, ''Data abstraction in Softclass,'' in *Proceedings of TOOLS USA 93*, Santa Barbara, CA, Aug. 2-5th Prentice-Hall, pp. 133-149.
- [Mili et al., 1999] Hafedh Mili et al., ''View programming: towards a framework for the decentralized development and execution of OO programs,'' in *Proceedings of TOOLS USA'99*, Santa Barbara, CA, August 2-5, 1999, IEEE CS Press, pp. ??-??.
- [Ossher 96] Harold Ossher, Mathew Kaplan, William Harrison, Alex Katz, and Vincent Kruskal, ''Specifying subject-oriented composition,'' in *Theory and Practice of Object Systems* (*TAPOS*), 2(3), 1996. Special issue on "Subjectivity in Object-Oriented Systems."
- [Reenskaugh, 1995] Trygve Reenskaugh, in *Working with Objects*, Prentice-Hall, 1995
- [VanHilst & Notkin, 1996] M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs," in *OOPSLA'96*, San-Jose, CA, 6-10 Oct., 1996, pp. 359-369.