Separation and composition of concerns

Mehmet Aksit
University of Twente
P.O. Box 217
7500 AE Enschede, The Netherlands
e-mail: aksit@cs.utwente.nl

Separation of concerns in the object-oriented model
In the (conventional) OO model, the “separation of concerns” principle is supported basically in three ways: (a) by defining objects as the models of the real-world concepts, which are “naturally” separated from each other; (b) by separating the concerns of providing an abstract object interface and the implementation of it; and © by grouping functions together around objects so that functions which are less related are structurally separated from each other.

Composition of concerns
To be able to construct complex software systems, the separated concerns must be put together using minimum effort. The OO model provide various ways in composing concerns together: (a) In the implementation part of an object, the structure and the behavior of the nested implementation objects can be composed under the definition of the encapsulating object; (b) Both inheritance and delegation mechanisms define composition of behavior. For example, in inheritance, the operations defined within a sub-class is composed with the operations of its super-class(es); © By defining a set-of protocols among cooperating objects, the software engineer may create more complex system structures provided that each component (or object) fulfil the protocol specification.

Application domain concerns
Application domains may define additional concerns. Consider for example, an electronic mail object. This object provides operations for defining the sender, receiver and the content of the mail. In addition, various operations are defined to approve and deliver the mail to its destination. An important requirement here is that only the “system objects” are allowed to invoke the approve and deliver operations. In addition to the previously mentioned concerns, a mail object has therefore an additional concern: “multiple-views”. Each mail object has a user-view and a system-view that restrict the operations of the mail object with respect to the clients of the mail object.

To implement the mail object, each concern must be mapped to an OO concept. For example, views can be implemented as operations. The operation approve, for instance, can be executed if the operation checkSystemView returns True. This implementation, however, blurs the separation of the multiple-views concern because the views are then realized within the operations of objects.

The problem appears if we want to extend/modify the mail object. For example, we may further partition the user view as sender and receiver views. We may extend the sender and receiver views to group-sender and group-receiver views. We may like to give a warning message if the same operation is invoked for the same mail object twice. In almost all these cases, the OO model cannot express these extensions without redefining the previously defined mail object. This is because the mail object cannot implement multiple-views on objects as a separate and composable concern.

Obviously, there can be many other concerns. For example, various synchronization constraints can be defined for the mail object. A request for send, for instance, can be delayed if the sender, receiver and the mail content have not been defined yet. Other possible concerns are for example, addressing the history information, evolution of behavior, coordinated behavior, security and reliability measures, real-time behavior, distribution aspects, etc. Since, the OO model may not separate these concerns adequately, the resulting programs are likely to be less adaptable and reusable (i).
The proposed design patterns in [11] cannot solve these problems adequately because the compositability features of the patterns are defined by the capabilities of the conventional OO model.

**Extensions to the object-model**

One may define a composable model for a particular concern by identifying the orthogonal components and the composition operators for that concern. For example, the multiple view problem can be modeled by representing views as explicit concepts and by defining accept functions between views and the operations of the mail object. These additional concepts and operators have to be integrated with the OO model. This can be realized in various ways:

1. Using special language constructs: Many research proposals introduce new language constructs and/or computation models to tackle a given problem. The introduced language constructs must be uniformly integrated with the compositability features of the underlying object model. Otherwise, the resulting programs cannot be fully composable. Most publications, expressiveness is the major concern and the proposed language constructs are hardly composable (ii).

2. Meta-level programming: Meta-level programming can be used to solve specific concerns at a meta-level. Reflection techniques can be used to keep different levels consistent with each other. For example, views and accept functions can be defined at a meta-level without modifying the basic structure of the mail object. The challenge here is to define reflection techniques which supports, in addition to the basic OO concerns, a large number of possible application defined concerns in a composable manner. Most research activities in this area do not address the compositability issues of the proposed meta-level concerns (iii).

**Composition of concerns**

Apart from our work, recently, a number of publications have appeared to address the compositability problems in object-oriented modeling [25] [24][10].

We have extended the conventional OO model with the concept of composition filters (CFs). Each message that arrives at an object (or sent from an object) is subject to evaluation and manipulation by the CFs of that object. CFs can be attached to objects expressed in different languages, such as C++ or Smalltalk [13] [9], and therefore allow extensions of programs written in different languages; the conventional OO model can be used to implement the basic concerns such as the mail object, and each additional concern can be expressed as a CF (iv).

Since different types of CFs show a similar structure, we have been investigating more primitive compositional structures than the ones provided by the CF model. We modeled the currently defined CFs using so-called message manipulators [27]. Message manipulators define logical operators for the received (and sent) messages to (or from) an object. These operators are for example, AND, Conditional-AND, OR, Conditional-OR, and Sequential manipulators. Each manipulator can be further decomposed by using sub-manipulators, etc. A “terminal manipulator” is composed of a constraint checker and accept and reject handlers. Constraint checkers are applied to messages. The accept and reject handlers are first class objects and represent the semantics of different concerns. Our conclusion here is that defining logical message composition operators with extensible semantics is a promising way for composing separated concerns together.

We believe that the concept of composition of different concerns must be also applied during the software development process (v). During software development, the so-called software artifacts are generated in various formats, from informal textual information to executable object-oriented programming concepts. Composability of design models require explicit representation of software artifacts in a composable way. In our recent work [7], we have applied fuzzy-logic based techniques to represent and compose various software artifacts. In contrast to deterministic object-level compositions, we found the fuzzy-logic based reasoning techniques more appropriate for representing design level concerns because fuzzy-logic can deal with design uncertainties.

The so-called architecture definition languages (ADLs)[29] are used to model and structure higher-level design concepts. Most architecture definition languages, however, do not adequately address the issue of evolution and composition of different architectural concepts [12]. In this direction, we are
currently defining an ADL based on the concept of composition of specializations of knowledge domains [6].

(i) Several publications identified the composability problems for certain concerns such as atomic transactions [14], synchronization [22] and real-time specifications [5]. [16] discusses the problem of separation of concerns in OO modeling.

(ii) For example, [18], [28], [15] and [19] introduce language constructs to model synchronization, real-time, coordinated behavior and multiple-views concerns, respectively.

(iii) A number of papers presented meta-level solutions for various problems, such as distributed architectures [23], atomic transactions [26], concurrent programming [17], operating system structuring [30] and compiler design [20].

(iv) Several different filter types have been defined in the past. For example, [1] illustrated how both inheritance and delegation can be simulated using filters. In [2] filters were introduced for defining reusable transactions. Language-database problems were addressed in [3]. In [4], filters were used to abstract coordinated behavior among objects. The application of composition-filters for real-time specifications was published in [5]. Composing synchronization and real-time specifications was published in [8].

(v) Propogation Patterns [21], for example, separate the concern of defining algorithms and class structures from each other.

References


