Separation and Composition of Overlapping and Interacting Concerns

António Rito Silva
INESC/IST Technical University of Lisbon
Rua Alves Redol n°9, 1000 Lisboa, PORTUGAL
Rito.Silva@{ACM.org | INESC.pt}

Abstract

This position paper presents some of the problems we had and the results we achieved in the last 4 years of work when defining and developing DASCo [24]. DASCo (Development of Distributed Applications with Separation of Concerns) is an approach for developing object-oriented concurrent and distributed programs using a separation of concerns strategy. In this position paper we emphasize separation and composition of non-orthogonal concerns. The interested reader can obtain more information from the DASCo web page at `http://www.esw.inesc.pt/~ars/dasco`.

1. Introduction

Separation of concerns approaches have to deal with two different issues: abstractions and integration mechanisms. The former describes solutions for domain-specific aspects of the software, e.g. object synchronization. The latter are responsible for integrating abstractions among themselves and with the functionality object, e.g. the integration of an object synchronization abstraction with the functionality object.

Non-separation of concerns approaches, called unification approaches, do not separate the abstractions from the object model. For instance, in the POOL-T [2] object model, all objects possess an internal activity, they are called sequential objects. That way, integration mechanisms are not necessary, because whenever an object is defined it is immediately a sequential object, it is from the very begin integrated with an abstraction for object concurrency. The advantage of unification approaches is in the simplicity they present to the final programmer. For instance, object concurrency abstractions are more transparent. However, unification approaches, to be flexible, have to include in the object model composition and extensibility primitives [6] associated with abstractions, which, sometimes, have unexpected repercussions on the semantics of the object model. The inheritance anomaly problem [15] is a well-known situation, extensively discussed in the literature, that results of unification approaches [17].

To avoid the flexibility problems of unification approaches, separation of concerns approaches, e.g. [3], integrate abstractions only in the final program. The gain is in flexibility, for abstractions are separated from functionality, thus facilitating their customization and reuse.

The first requirement for separation of concerns approaches is to provide flexible abstractions for software concerns.

The separation of concerns forces their later composition. Two perspectives exist regarding concern composition: orthogonal and non-orthogonal. The orthogonal perspective considers abstractions to be completely orthogonal being enough to join them [1, 5, 16], e.g. join them sequentially, while the non-orthogonal perspective considers that concern composition must consistently integrate the abstractions parts exhibiting semantic overlapping [18, 12]. Note that the orthogonal composition perspective is domain-independent, since there is no semantic overlapping. In that case the composition can be mostly supported by the integration mechanism.

The second requirement for separation of concerns approaches is to support abstraction composition.

In terms of integration mechanisms, separation of concerns approaches may be based on design solutions [7], on compilers [3], or on reflection [14].

The *Proxy* pattern [7] is an example of an integration mechanism based on design solutions. For instance, when applying the *Proxy* pattern to the functional object, the interface is separated from the implementation. The integration of abstractions is done through a new implementation of the functional object's interface which uses the previous functionality implementation. Thus, the integration does not change the functional object's implementation, facilitating incremental development. Moreover, clients need not be changed for the interface they use remains the same. The disadvantage of this solution is that all functional classes must have an abstract interface and objects creating instances of

these classes must use an *Abstract Factory* [7] so that they may ignore the actual class implementation they are using.

When using a compiler as an integration mechanism [3], code generation is automatic and avoids the need for functional classes to have abstract interfaces. However, this integration mechanism does not allow run-time substitution of concurrency support constructions.

Reflection [14] is an integration mechanism[1] which allows control and customization of a computational system's behavior through changes to a meta-system built of meta-objects [10]. Meta-objects should support the abstractions. It is the responsibility of the reflection's object model to integrate the abstractions. In addition, reflection allows run-time meta-object replacement.

The third requirement for separation of concerns approaches is to describe abstractions and abstraction composition independently of integration mechanisms.

Obviously, the flexibility provided by separation of concern approaches pay a price on simplicity. It would be important to have a good balance between flexibility and simplicity.

The fourth requirement for separation of concerns approaches is to balance abstraction flexibility and use simplicity.

2. DASCo Approach

DASCo approach is based on design patterns, composition patterns, and object-oriented frameworks to, respectively, describe abstractions, describe abstractions composition, and implement and integrate abstractions and their compositions. This approach fulfills the four requirements mentioned in the previous section.

When introducing DASCo approach we will use examples from concurrent programming concerns, as object synchronization and object concurrency. A complete description of these concerns can be found in [22].

2.1. Design Patterns

A design pattern [7] is defined for each abstraction. Design patterns must have the following properties in order to describe flexible abstractions:

- *Expressiveness*. Several abstraction policies are supported by specializing the design pattern associated with the abstraction. For instance: an object concurrency pattern has, among others, specializations for sequential and concurrent objects.

[1] Briot *et al.*[4] considers reflection as classifying a set of approaches. This classification is centered on the properties of integration mechanisms.

- *Modularity*. Each concern's design pattern must separate the abstraction from the object's functionality. Modularity facilitates incremental development, for it is possible to develop and replace the implementation of the abstraction separately from the object's functionality.

- *Reusability*. Each abstraction and functional objects must be separately reusable and extendible. For instance, synchronization and functionality must not be reused together using, for instance, inheritance. That way, synchronization can be integrated, *a posteriori*, with each concrete class to be synchronized.

2.2. Composition Patterns

Abstraction composition is obtained from the composition of each concern's design patterns. For instance, to obtain an abstraction for a synchronized and concurrent object it is necessary to compose design patterns corresponding to the object synchronization and object concurrency abstractions.

The composition of design patterns also constitute an abstraction described by a design pattern which participants are built by composing each involved pattern's participants. In addition, the composition pattern's collaborations are built from the collaborations of each pattern participating in the composition.

From our experimentation, we concluded that orthogonal composition is rather restrictive. In many situations there is semantic overlapping between the different abstractions. Moreover, we verified that some composition abstractions have their own policies that are not trivially inferred from each of the composed abstractions: *"The whole is more than the sum of its parts"* [22].

So, in order to support abstraction composition, composition patterns must possess the following properties:

- *Conservation*. The properties of each of the participating design pattern's properties must be preserved. For instance, the expressiveness of each design pattern must be maintained isolated in the composition pattern.

- *Consistency*. In spite of conservation, the semantic overlapping between abstractions should be handled. To deal with the synergy generated by the abstraction composition, the composition pattern should describe the new policies, that result from the composition. It should also describe what are the consistent combinations of overlapping parts and what are the restriction to policy composition. A matrix can be used to identify new policies and the associated participants patterns policies combination restrictions.

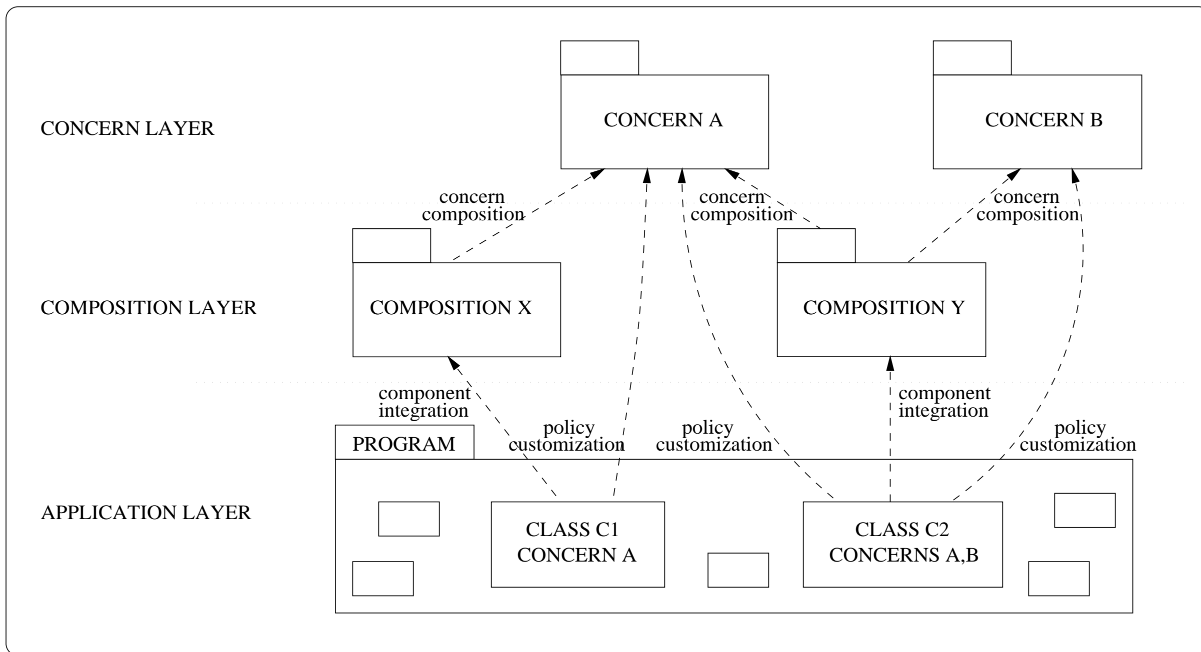

Figure 1. Architecture of the Three-layered Object-oriented Framework with Separation of Concerns.

We have found that not all the policies composition restrictions can be easily described with generality. When the policy is not object-independent there is no generic rule to describe those restrictions. This happens because the abstraction depends on the object implementation. For instance, consider a readers/writers policy on an object that uses a update-in-place recovery policy [23], the object is immediately updated by invocations. In this situation two write operations on the same attribute can not execute concurrently. However, if a deferred-update policy [23] is used, the invocations execute on different copy objects, two write operations can execute concurrently without generating a crash.

2.3. Three-layered Object-oriented Framework

Since the design patterns and composition patterns describe abstractions without considering a specific integration mechanism, DASCo approach satisfies the third requirement. To fulfill the fourth requirement it is necessary to consider abstractions implementation because simplicity will be provided by interfaces and encapsulation.

Design and composition patterns are implemented in a three-layered object-oriented framework with separation of concerns. In this framework, classes implementing design and composition patterns are grouped within components[2]. In the object-oriented framework's concrete case, components consist of classes implementing design and composition patterns, and are instantiated either through instantiation of the explicit interface's parameters or through specialization of some of their classes.

Figure 1 represents a component diagram. It illustrates the architecture of the three-layered object-oriented framework with separation of concerns.

The object-oriented framework's three layers are:

- **Concern.** The concern layer contains classes implementing each of the design patterns. These classes are grouped within components called concern components. In addition to ensuring the design patterns' properties, concern components must provide a set of classes to support composition of concern components and customization of concern policies to be done, respectively, in the composition and application layers.

- **Composition**. The composition layer contains classes implementing composition patterns. These classes are grouped within components called composition components. In addition to ensuring the composition patterns' properties, composition components must define a *minimal interface*. A composition component's minimal interface is instantiated to integrate the component in the application layer. Using the minimal interface,

[2] A component is defined by Nierstrasz & Dami[19] as being an abstraction of a software module which encapsulates its implementation's details and has an explicit interface through which systems can be built by instantiating its parameters and connecting to other components.

the final programmer does not need to know the details of the patterns' composition, which are encapsulated by the component, thus simplifying the framework's use. The singular composition of concern components, e.g. `Composition X` in figure 1, is needed to give the corresponding concern component a minimal interface.

- **Application**. In the application layer composition components are integrated in the final program and concern components are customized so that they provide the policies required by the program's functional objects. For instance, to obtain a concurrent and synchronized object it is necessary to integrate the functional object and the concurrent synchronized object composition component using its minimal interface. In addition, it is necessary to customize the corresponding concern components by specializing some of their classes. It should be noted that the composition component must be independent from customizations of concern components. Integration and customization in the application layer must respect the following properties:

  - *Transparent Integration*. The integration of composition components with functional objects must be as transparent as possible, so that functional objects and their clients need not be changed. This property depends on the integration mechanism to be used.
  - *Incremental Customization*. The customization of concern components must be independent from the customizations of the remaining concern components. For instance, incrementally changing a synchronized object into a concurrent and synchronized object should require only, from the customization point of view, the concurrency component's customization. The previous customization of the synchronization component is reused. This property establishes a compromise between conservation and consistency properties of composition patterns, the complexities of composition are hidden by the composition component but concern components customization still orthogonal.

The object-oriented framework implements the property of pattern reusability either as white-box reuse or as black-box reuse. Customization of concern component classes is white-box reuse, which has great flexibility but requires the programmer to know about the design pattern's internal structure. However, if the programmer uses predefined policy implementations then black-box reuse is being done for the programmer has only to create instances of previously customized classes and associate them with the corresponding invocations. In this situation, arrow `policy customization` in figure 1 indicates, in fact, that a previously customized policy is being used.

3. Related Work

DASCo follows a multi-dimensional separation of concern [25] where abstractions are hyperslices. In this section we discuss other separation of concerns approaches.

Subject-oriented programming [9, 20] is a separation of concerns approach which defines classes by composing them from subjects. Each subject provides a particular point of view of a class. A subject can be related with a DASCo design pattern. Subject composition rules do not distinguish composition from customization.

Kicsales *et al.*[11] defines a set of rules for defining modules using reflection, called open implementation [13]. These modules seek to establish a trade-off between black-box reuse and white-box reuse. Thus, each module offers a meta-interface through which the programmer is able to choose the module's implementation details, e.g. the implementation type of a list module. As conclusion of his experiments with open implementation, Haines[8] indicates, as this approach's main problem, the difficulties in composing meta-interfaces which are not completely orthogonal. In our opinion, this problem arises from not defining two composition levels, for composition and customization are carried out using the meta-interface.

The decomposition in components of finer granularity, proposed in the CodA reflective architecture [16] seeks to avoid the problems of abstraction composition. CodA defines seven meta-objects: `send`, `accept`, `queue`, `receive`, `protocol`, `execution`, and `state`. The seven meta-objects which form the architecture must be customized and composed among themselves in order to support the required concerns. Due to the fine granularity of each meta-object, these have a small number of responsibilities and their composition is orthogonal, for it is sufficient to specialize each of them separately. The disadvantage of CodA is in the complexity associated with meta-object composition, for they do not have a correspondence with concerns and thus it is necessary to understand all roles a meta-object has in different concerns. Again, this problem arises from not having two composition levels as proposed in the three-layered framework. In contrast, in the DASCo approach it is easier to define each concern's customization, for it is separated from composition, and the composition component encapsulates the complexity associated with the composition of concern components.

Composition filters [1] extend the object model with various non-functional components - composition filters - which

intercept object messages, both incoming and outgoing, expressing restrictions such as, for instance, on synchronization. Composition filters are sequentially composed and the composition semantics is given by the logical operators: `AND`, if a message has to pass through both filters; and `OR`, if a message needs only to pass through one of them. So, composition is orthogonal. It has been seen that consistent compositions are not necessarily sequential or nested compositions. This leads to the representation of each concern by several filters, to allow non-sequential and non-nested compositions. In this way, the goal of having a filter for each concern is not accomplished. The solution becomes closer to the fragmentation present in the CodA approach [16], with the complexity problems already mentioned above.

Aspect-oriented programming [12] consists of a component language to program functionality, one or more aspect languages to program concerns, and an aspect weaver which combines languages. Programs are separately built using the different languages. Aspect-oriented programming allows non-orthogonal composition and delegates on the aspect weaver the resolution of complexity associated with concern combination. The weaver is the integration mechanism. We have seen that several composition policies may exist, being the programmer's responsible for choosing one of them. Using aspect-oriented programming, it will be necessary, in this context, to define a language capable of expressing the several composition policies: a language capable of composing aspect languages. This language is not considered, to the extent of our knowledge, by aspect-oriented programming approaches. In addition, the relationship between these composition languages and the aspect weaver is not clear.

The only approach, we are aware of, that distinguishes an application layer from a composition layer is described by Mulet *et al.*[18] in the context of meta-object composition. In this approach, a composition architecture is defined which hides from the final programmers the details of meta-object implementation and of their composition. In addition, the approach identifies three properties meta-objects must possess in order to be composed: encapsulation, independence (modularity), and exclusiveness. However, the approach does not consider the possibility of concern customization by the user, for their customization is done jointly with their composition.

Role model design [21], where (i) design patterns may be described using role models; (ii) composition patterns may be described as compositions of the role models of the participating design patterns; and (iii) object-oriented frameworks may be described using class models including role models of both design patterns and composition patterns, seems to be a suitable formalism to describe DASCo design patterns, pattern composition and object-oriented frameworks.

4. Conclusion

The approach proposed in this paper is based on separation of concerns and uses design patterns to separately describe abstractions. Implementations of design patterns follow a three-layered architecture with separation of concerns. The minimal interface property of the architecture's composition components simplifies their use by the programmer of functional classes, for it limits the set of concepts the programmer must know and handle. The properties defined for the abstractions and the three-layered architecture do not impose the use of a specific integration mechanism.

In order to control the complexity associated with the development of programs different abstraction levels have been defined. Each corresponds to a different kind of user/programmer:

- The final programmer may use the object-oriented framework knowing only the policy, for instance, readers/writers, needed for the functional object. In this case, the programmer uses a predefined policy of the object synchronization pattern. If the programmer wants to customize the policy according to specific needs, then a specific synchronization policy has to be implemented. For this, the programmer must possess the knowledge described by the abstraction and know how it is implemented in the corresponding concern component. In this situation, the programmer is not required to know about the particularities of composition patterns nor about their implementation.
- Framework programmers need to have more knowledge, both of specific domain abstractions: object synchronization, object concurrency, and object recovery; and of the patterns' implementation and their composition.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1994.

[2] P. America. Pool-t: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[3] C. Atkinson, S. Goldsack, A. di Maio, and R. Bayan. Object-oriented concurrency and distribution in dragoon. *Journal of Object-Oriented Programming*, 4(1):11–18, March/April 1991.

[4] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[5] I. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclasses in som. In *OOPSLA'94*, pages 427–439, Portland, Oregon, October 1994.

[6] S. Frolund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In *ECOOP'92*, pages 185–196, Utrecht, The Netherlands, June/July 1992.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[8] M. Haines. An Open Implementation Analysis and Design for Lightweight Threads. In *Conference on Object-Oriented Programming Systems, Languages and Applications, Proceedings*, pages 229–242, Atlanta, Georgia, USA, October 1997.

[9] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA '93 Proceedings*, pages 411–428, Washington, DC, September 1993.

[10] G. Kicsales, J. des Rivieres, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[11] G. Kicsales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *19th International Conference on Software Engineering*, Boston, MA, USA, May 1996.

[12] G. Kicsales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, February 1997.

[13] C. Maeda, A. Lee, G. Murphy, and G. Kicsales. Open implementation analysis and design. In *ACM Symposium on Software Reusability (SSR)*, 1997.

[14] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87 Proceedings*, pages 147–155, Orlando, Florida, October 1987.

[15] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[16] J. McAffer. Meta-level Programming with CodA. In *ECOOP'95*, pages 190–214, Aarhus, Denmark, August 1995.

[17] C. McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[18] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA'95*, pages 316–330, Austin, USA, October 1995.

[19] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Components*, pages 3–28. Prentice-Hall, 1995.

[20] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *OOPSLA'95*, pages 235–250, Austin, USA, October 1995.

[21] D. Riehle and T. Gross. Role model based framework design and integration. In *Conference on Object-Oriented Programming Systems, Languages and Applications, Proceedings*, pages 117–133, Vancouver, Canada, October 1998.

[22] A. R. Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks*. PhD thesis, Instituto Superior Técnico - Technical University of Lisbon, March 1999.

[23] A. R. Silva, J. Pereira, and J. A. Marques. Object Recovery. In R. Martin, D. Riehle, and F. Buschman, editors, *Pattern Languages of Program Design 3*, chapter 15, pages 261–276. Addison-Wesley, 1997.

[24] A. R. Silva, P. Sousa, and J. A. Marques. Development of Distributed Applications with Separation of Concerns. In *IEEE Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995.

[25] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, May 1999.