AOP and Reflection for Dynamic Hyperslices

R. Chitchyan, I. Sommerville

Computing Department, Lancaster University, Lancaster, LA1 4YR
{rouza | is}@comp.lancs.ac.uk

Abstract

In this paper we present a Model for Dynamic Hyperslices which uses a particular Aspect-Oriented (AO) approach – Hyperspaces – for decomposition and reflection as a means for composition of software modules. This model allows for structured, dynamic, incremental change introduction and rollback, thus, supporting run-time evolution yet preserving component modularity. The applicability of the model is illustrated through a schema adaptation scenario.

1. Introduction

Aspect-Oriented software development (AOSD) is a methodology for software development with an emphasis on the separation of concerns principle. AOSD takes the next step, after OO, in developing well modularised software, by separating the crosscutting concerns. A significant part of work in AO community focuses on software evolution support [1, 2] as well as dynamic change due to run-time weaving of aspects [3-5]. AOSD itself provides new mechanisms, such as joinpoints, pointcuts and introductions, that can be used to facilitate dynamic change, but it does not explicitly provide any structured methodology to manage and support dynamic change in software.

Such a structure, however, is provided by reflective approaches to software development [6, 7]. In reflection the main emphasis is on transparent manipulation of the base level via adaptation of the meta level. Meta level is a handle for controlling the base. This is particularly useful for "non-invasive" run-time adaptation of the base code and dynamic re-configuration using meta-object protocols.

Thus, we suggest that the meta-object protocols of reflection provide control and manipulation mechanism that, in combination with the modularisation and change introduction capabilities of AO, could lead to well-modularised, dynamically evolvable software systems.

This approach has been adopted in development of the Dynamic Hyperslices Model briefly outlined and illustrated through an example in section 2 of this paper. Some implementation-related issues for the model are examined in section 3 and the discussion is summarised in section 4.

2. The model for Dynamic Hyperslices

2.1 Outline of the model

The Dynamic Hyperslices model [8, 9] is intended to support the dynamic evolution of non-stop systems, i.e. systems that cannot be easily taken offline due to high costs of their downtime (e.g. telephone and banking), environmental safety (e.g. nuclear plants), loss of human life (e.g. life support systems) and such like. The model uses the Hyperspaces approach [10-12] to decompose the software system into "single-minded" modules (e.g. a module for `Health` feature of the `Person` object) and the power of reflection [6, 7] along with filters (as discussed in the Composition Filters approach [13]) and architectural connectors [14, 15] for unit composition and run-time manipulation.

The Dynamic Hyperslices aims to provide a composition mechanism that allows all the primary concerns, decomposed in accordance with the Hyperspaces approach, to endure in the composite concerns after composition.

In the Hyperspaces approach [10-12] the software is modelled as a set of modules (called hyperslices) each of which represents only one single concern. These hyperslices are then composed using matching units (e.g. method names) in different hyperslices as join-points. Composition-related concerns are not treated as first class entities, but are transitory units which integrate with primary hyperslices into a composed unit. Composition is a compile-time process and the final composed module has no recollection of its composite parts.

We maintain the decomposition principles of Hyperspaces, but differ in our composition approach. We use connectors for composition. Filters form part of our *composition connectors* where connectors connect hyperslices and not (necessarily) complete object classes or (OO) components. Our connectors don't simply match provided/required services, or specify roles for connected components, but rely on

a dynamically updateable composition strategy to build up functionality of coarser-grain components (e.g. object classes) from primary hyperslices[1], as well as carry out the communication between the member hyperslices at run time.

In short, the model:

- Uses the Hyperspaces decomposition approach in separating concerns into single-minded hyperslices (or primary concerns).

- Requires that an additional dimension for Composition concerns is specified in each Hyperspace-type decomposition. This additional dimension contains connector-concerns. At the composition stage the connector concerns are used to compose other concerns.

- Utilises a composition connector to integrate any primary/composite concerns. Consequently, any interaction between other concerns is channelled through a set of connectors.

- Provides connectors with capability to reflect upon their immediately connected concerns, while still keeping these internals hidden from all other connectors and hyperslices.

The Dynamic Hyperslices approach is illustrated using an example of dynamic schema adaptation[2] in the following sub-section.

2.2 Illustrative Example

Dynamic database schema adaptation is desirable since, in database-centric environments (for instance in banking sector), downtime of the central database system is very costly. Consequently, the ability to seamlessly incorporate change into a database schema at run-time promises significant financial gains.

Figure 1(a) below depicts a certain (oversimplified) Object Database schema. The organisation that owns this database has kept records of its clients, the (financial) services that it provides and the registrations that its clients have undertaken for the provided services (e.g. Instant Savings Accounts). Assume the organisation wants to improve its service provision and so intends to encourage the clients to fill in newly introduced questionnaires about the services they use. To motivate the clients, for each filled in questionnaire the clients registration record will be credited with a free quantity of service (e.g. extra 0.1% of interest gained).
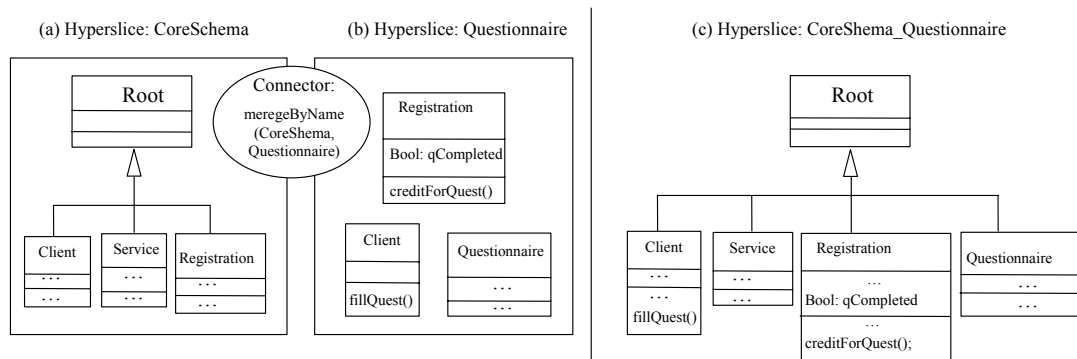

Figure 1: Illustration of the model for Dynamic Hyperslices.

Using the Hyperspaces decomposition approach, we adopt the existing schema as a composite hyperslice CoreSchema (presented in Figure 1.a). Then, we design and develop the newly required set of functionality as a separate hyperslice Questionnaire (presented on Figure 1.b), that has only those concerns that deal with the issues related to the questionnaire. The Questionnaire hypeslice consists of newly introduced Questionnaire class, Client class which has only one method (fillQuest) to allow clients to fill in the questionnaires, and the Registration class which has a Boolean variable qCompleted to indicate wheather the questionnaire for the given registration has been completed, and a method for crediting the registration with additional free quantity of service for each completed questionnaire.

---

[1] Thus, the composition strategy in the connectors can be perceived as a kind of "merger algorithm" for producing higher order artifacts. Here the "merger" is performed through run-time message manipulation within connectors, without physically merging the hyperslices.

[2] More about this subject can be found in [16].

The Connector, depicted as an oval, linking `CoreSchema` and `Questionnaire` hyperslices is the run-time composition mechanism that combines the separate hyperslices into a composite `CoreSchema_Questionnaire` hyperslice view (presented in Figure 1.c). However, while the general view of the updated schema will be that presented on the right of Figure 1 (i.e. part *c*), the initially independent hyperslices for core schema (1.a) and the questionnaire (1.b) will be retained intact as presented on the left side of Figure 1 (i.e. parts *a*, *b*, and the connector). The connector will retain the composition information which leads to the change of the schema, allowing for rollback to the previous version, if required. The connector is also the communication mechanism between the composed hyperslices as well as their clients.

Thus, in this section we have briefly outlined the possible applicability of the Dynamic Hyperslices approach to a database schema evolution problem. We have discussed how with the Dynamic Hyperslices approach a coherent view of the evolving database schema can be retained, along with the details of the historical change (contained in connectors), allowing for rollback, if required. Yet, our approach avoids the space usage overheads and coarseness of retained change history, as is the case with the traditional schema versioning approaches (when several versions of the same schema are kept) [17, 18]. We also avoid the pitfalls of class versioning approaches [19, 20] – when a copy of each new version of each class is retained – which result in overcomplicated schema and loss of a single coherent view on it (due to many versions of the same class).

3. Discussion on Implementation Issues

The Dynamic Hyperslices model is currently under development. In this section, we talk about some initial ideas for its implementation.

As discussed earlier, the composition in our model is carried out through reflective adaptation. The partial structure of the meta-object level of the model is presented below:
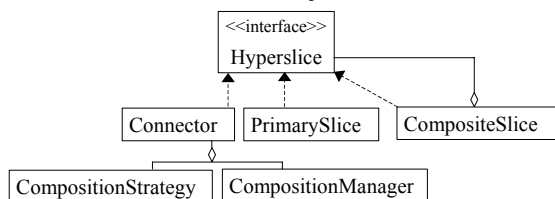

Figure 2: Partial structure of the meta-model for Dynamic Hyperslices

Figure 2 states that all primary and composite slices and the connectors are hyperslices. The connectors, primary, and composite slices can be composed into new composite slices. The composition details are provided through the composition strategy which is a part of the composition connectors. The composition process is monitored and validated by the `CompositionManager` element of the connectors.

The base level (i.e. application) programmer using the Dynamic Hyperslices model does not need to be aware of the above meta-level. The link between the base and the meta levels is established at load time via an AspectJ aspect which introduces and initialises the corresponding reference variables in the base and meta levels.
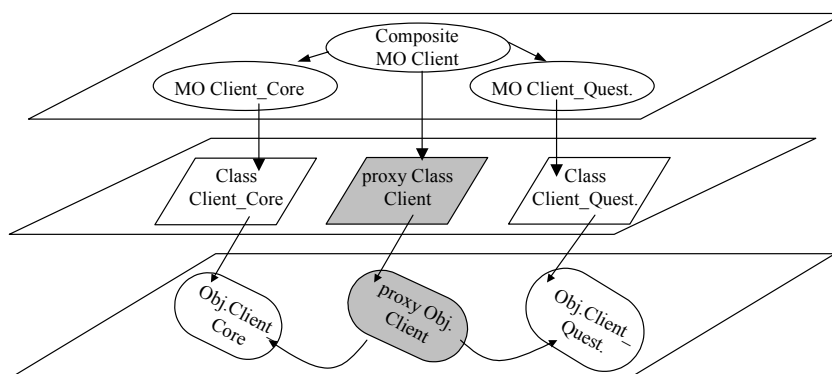

Figure 3: High-level workings of the model for Dynamic Hyperslices.

The high-level working of the system, also illustrated in Figure 3 above, is as follows:

- The Base and Meta level link is established at load time, with a meta-object created per each loaded class;
- Composed slices are represented by a proxy class at the base level and a composite meta-object at the meta level. Instantiation of a composed class results in instantiations of its components;
- All calls to the base level objects are passed to their meta-objects. The meta-objects resolve each calls in accordance with the composition strategy used and filter it down the composition chain to the resolved primary slice which executes the call;
- The topmost composite meta-object refers to the "combined" interface of all composition participants. This combined interface is displayed to all clients of the composite slice

The implementation is being undertaken mainly with Java and AspectJ. It is likely that byte-code manipulations tools (such as BCEL or Javassist) will also be used. While our preferred option is to maintain module integrity all through its life cycle, including the run-time, we are aware that in medium to long term this approach will have noticeable performance overheads. Consequently we plan to consider various optimisation strategies, e.g. guarded integration of "stable" compositions into coarse-grained slices, with only guard checked for changes, rather then the whole composition chain; or permanent integration of certain changes into module structures (at the system maintainer's discretion) to improve performance in critical places.

Another challenging issue is that of instance adaptation, i.e. how to make objects consistent with the evolved classes. For example (going back to our example in section 2.2) how will the instances of Client class, created before composition of Questionnaire slice, handle requests to fill in questionnaire? Our present intent is to use conversion of the objects to the new definitions of their classes with a hyperslice for instance conversion handling. Thus, the instance adaptation strategy itself will be evolvable, in correspondence with the evolving schema.

4. Summary and Future Work

In the present paper we have suggested that reflection and AO can be used as complementary technologies, with reflection particularly well suited for dynamic reconfiguration and adaptation and AO as a modularisation mechanism.

We have employed the above principle in the development of the Dynamic Hyperslices model, where we use a particular AO decomposition mechanism (i.e. that suggested by the Hyperspaces approach) in combination with a reflection-based composition (via our composition connectors). The applicability of this model has been illustrated though a schema evolution scenario.

While the Dynamic Hyperslices model simplifies the change introduction and module (i.e. hyperslice) development process, it requires some consideration for the complexity of slice composition. However, the proposed model also provides for treating the composition concerns themselves as 1st class entities, similar to any other slices. Implementation and refinement of the composition mechanism is one of the prime tasks to us at the present time. Some other implementation related issues, besides those already discussed in section 3, are the development of checks for correctness of composition, consideration of ways of incorporating domain-specific knowledge into composition process.

References

[1] A. Rashid and P. Sawyer, "Object Database Evolution using Separation of Concerns," *ACM SIGMOD Record*, vol. 29, pp. 26-33, 2000.

[2] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Support for Evolution from the Design Stage," in *Workshop on Software and Organisation Co-Evolution*, 1999.

[3] A. Popovici, G. Alonso, and T. Gross, "Just In Time Aspects: Efficient Dynamic Weaving for Java ." presented at 2nd International Conference on Aspect- Oriented Software Development, Boston, USA, 2003.

[4] E. Truyen, W. Joosen, and P. Verbaeten, "Run-time Support for Aspects in Distributed System Infrastructure," in *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, 2002.

[5] J. Boner and A. Vasseur, "AspectWerkz Web Site, http://aspectwerkz.codehaus.org," 2004.

[6]   P. Maes, "Concepts and Experiments in Computational Reflection," presented at OOPSLA, 1987.

[7]   G. T. Sullivan, "Aspect-Oriented Programming Using Reflection and Meta-Object Protocols," *Communications of ACM*, vol. 44, pp. 95-97, 2001.

[8]   R. Chitchyan, I. Sommerville, and A. Rashid, "A Model for Dynamic Hyperspaces," presented at Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD 2003), 2003.

[9]   R. Chitchyan and I. Sommerville, "Composing Dynamic Hyperslices," presented at Workshop on Correctness of Model-based Software Composition   (ECOOP 2003), Darmstadt, Germany, 2003.

[10]   H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns using Hyperspaces," IBM Research Report 1999.

[11]   P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," presented at Proc. 21st International Conference on Software Engineering (ICSE 1999), 1999.

[12]   P. L. Tarr and H. Ossher, *Hyper/J user and Installation Manual*: IBM Research, 2000.

[13]   L. Bergmans and M. Aksit, "Composing Crosscutting Concerns using Composition Filters," *Communications of the ACM*, vol. 44, 2001.

[14]   M. Shaw, "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status," presented at Studies of Software Design, Proceedings of a 1993 Workshop, 1996.

[15]   D. Balek, "Connectors in Software Architectures (PhD Thesis)," in *Faculty of Mathematics and Physics*. Prague: Charles University, 2002.

[16]   A. Rashid, "A Database Evolution Approach for Object-Oriented Databases," in *Computing Department*: Lancaster University, UK, 2000.

[17]   W. Kim and H. T. Chou, "Versions of Schema for Object-Oriented Databases," presented at 14th International Conference on Very Large Databases, 1988.

[18]   B. S. Lerner and A. N. Habermann, "Beyond Schema Evolution to Database Reorganisation," presented at Proceedings of ECOOP/OOPSLA, 1990.

[19]   S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning," *SIGMOD Record*, vol. 22, pp. 16-22, 1993.

[20]   A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in Object-Oriented Databases," presented at OOPSLA, 1986.