Towards the Usage of Dynamic Object Aggregations as a Foundation for Composition*

Gustaf Neumann
Department of Information Systems
Vienna University of Economics and BA, Austria
gustaf.neumann@wu-wien.ac.at

Uwe Zdun
Specification of Software Systems
University of Essen, Germany
uwe.zdun@uni-essen.de

ABSTRACT

Many current programming languages offer insufficient support for the aggregation relationship. They do not offer a language support for composite structures (object hierarchies, whole/part hierarchies) on the object-level. Instead they rely on techniques, like embedding or referencing through pointers, which do not fully incorporate the semantics of aggregation. As a superior technique we present the language construct dynamic object aggregations.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Object aggregation, object composition

1. INTRODUCTION AND RELATED WORK

In almost every software system objects that are composed out of other objects exist. In this paper we will focus on the enhancement of the aggregation relationship, commonly found in object-oriented design concepts, but only weakly supported by current programming languages. Our view of the aggregation relationship is influenced by investigations on the level of modules, like [7], in database systems, like [13], and in object-oriented concepts/languages, like in C++, Java, [3], or [8]. In general we can distinguish between aggregation of descriptive structures (aggregation of classes) and aggregation of instances (aggregation of objects). Several languages offer aggregation of descriptive structures through nested classes (e.g. Java, Beta, C++), which can be used as a form of static composition, as shown in [1].

We see aggregation as a composition technique for design and implementation of software systems, e.g. applicable for descriptive structures (like classes), software components, and objects. *Composition* means any assembly of several parts to a whole. Composition is more general than aggregation, e.g. composition of an object with a role or portion-object composition [12] can be modeled/implemented through other techniques (especially if parts share data with the whole/other parts). Note, that the definition of composition in UML is rather comparable to our definition of aggregation, while aggregation in UML is comparable to our view association.

In order to implement aggregation its semantics must be strict and automatically ensured. The language support for aggregation that we present in this paper is implemented in the object-oriented scripting language XOTcl [11]. XOTcl is a language offering a dynamic object and class system, read/write introspection, extensibility through components, and several high-level language constructs. These include the interception techniques per-object mixins [9] and filters [10]. Furthermore, it provides assertions, meta-data, nested classes, and dynamic object aggregations.

2. DYNAMIC OBJECT AGGREGATIONS

Definition 1 (Object aggregation). *An object system supports aggregation iff every object is allowed to aggregate other objects. The aggregated (inner) objects are part of the aggregating (outer) objects.*

The process of inserting one object into another object is called aggregation, the inverse step is called disaggregation.

Definition 2 (Object hierarchy). *Through aggregation objects form object trees (object hierarchies) with global objects as roots. Each object is member of exactly one object tree.*

As a consequence of this definition, every aggregated object is part of exactly one other object (which might be a global object). Another consequence of the definition is that an object can not contain itself (directly or transitively).

*Accepted for publication at Symposium of Applied Computing (SAC'00), Como, Italy, March, 2000.

Definition 3 (Dynamic object aggregation). *An object system supports dynamic aggregation iff arbitrary objects may be aggregated or disaggregated at arbitrary times during execution.*

The opposite of dynamic aggregation is static aggregation, which permits only the creation and deletion of aggregates, but not the dynamic change of the aggregation in an already created structure. The following operations are used to construct/modify an object hierarchy:

- *Object creation*: Every object is created with an identifier that is unique in the scope where it was created.
- *Object hierarchy restructuring*: A copy/move/delete operation works on the subtree of the object hierarchy starting with the named object.

The restructuring operations affect an object and all its contained objects. These operations may not violate the tree property of the object hierarchy.

As an example we model video films. Default part of each film is an intro. Therefore the property object is aggregated in the constructor `init`:

```
Class Film
Class Intro
Film instproc init args { Intro [self]::intro }
```

Using the `Class` command the two new classes are created. Afterwards the constructor of the class `Film` aggregates a new object `intro` for each new film. All objects are accessible through a fully qualified name containing "::" as separators. But in the common cases the explicit full qualification is not necessary, because in XOTcl methods, the current object can be accessed via the `self`-command, which is the reference to the current object. E.g. a certain film `starWars` gets an object `starWars::intro` automatically:

```
Film starWars
```

The implied constraints of the dynamic aggregation relationship are preserved automatically. A part of a video film we record from TV may be commercials, which are not default ingredients of films. Therefore, we aggregate them dynamically.

```
Class Commercials
Commercials starWars::commercial1
```

All relationships are dynamically changeable. At arbitrary times during run-time a new object may be aggregated or destroyed. E.g. if we want to cut the commercials from our star wars film copy, we model the situation by:

```
starWars::commercial1 destroy
```

Often an aggregated object is not destroyed but moved into another aggregating object. The XOTcl method `move` provides this functionality. Another common behavior is implemented by the `copy` method which clones the actual object to a destination object. The two methods have the syntax:

objName `move` *destination*
objName `copy` *destination*

E.g. if we want to reuse an imperial march object of star wars for star wars 2, we can just copy the object:

```
starWars::imperialMarch copy starWars2::imperialMarch
```

Information about the current aggregation relationship of objects can be obtained through introspection using the `info` method with the following syntax:

objName `info children`
objName `info parent`

E.g. a song player can ask the object whether `imperialMarch` is part of the film object or not, before it tries to play the song.

3. AGGREGATION FOR COMPOSITION

In the previous sections we the presented language construct with its implied semantics. Now we will present some of our current research topics, where we use the language construct as a composition technique.

3.1 Sharing Aggregated Objects

An intrinsic property of the aggregation according to the definition in Section 2 is that an object may only be part of one aggregation. This notion of the aggregation can be observed in reality very often, e.g. a room may be a part of only one building. Nevertheless, many modeling languages permit overlapping aggregations, e.g. the description of the whole-part pattern in [4] names the ability to share parts as an advantage of the pattern.

Despite the violation of the semantics of aggregation-/part of relationship, as it is used in the every day life (and defined in Section 2), there are several reasons for sharing aggregated objects. Often one and the same object is part of several orthogonal hierarchies. E.g. songs that are part of several films may be also part of a song collection. Often sharing is necessary for other reasons, like saving of storage, as in the flyweight pattern [6].

A mechanism enabling sharing, but differing from normal aggregations, would resolve this contradiction. Here, a link object adapts to the "real" aggregated object (as in the adapter pattern [6]). But a conventional implementation that explicitly forwards all calls suffers from the necessity to change all link objects if the real object changes. This is elaborate and error prone. A better solution is to use an interception technique, like filters [11] that automatically adapts all requests to the real object.

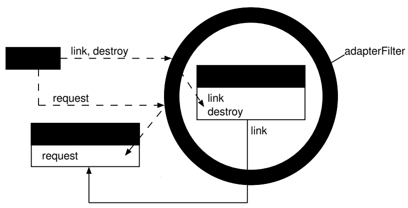

Figure 1: Link Realized through an Adapter Filter

Figure 1 shows the behavior of a link in general. The link contains a reference to an adaptee, named `link`. A special getter/setter instance method, also named `link`, is capable of changing this reference. Beneath this method only the destructor `destroy` reaches the link object. All other requests are adapted to the link's adaptee (the "real" object) automatically and transparently. As an example we can reuse a song from a film in a song collection:

```
Link songCollection::march1 -link starWars::imperialMarch
```

3.2 Aggregation Patterns

Dynamic object aggregations can be used as a foundation for proper implementation of design patterns that rely directly on aggregation, like the composite pattern [6], the cascade pattern [5], and the whole-part pattern [4]. In order to language support the patterns their further semantics have to be ensured additionally.

The composite pattern implements an aggregation hierarchy of objects of the same base type and assumes that all calls to parent objects are forwarded to all their children recursively. In [10] we show how to combine dynamic object aggregation with filters and meta-classes in order to perform these semantics automatically.

The cascade pattern describes a composite hierarchy, where a complex whole is additionally structured through layers, which are themselves composites. To implement cascade properly the implementation of composite has to be enhanced with the constraint that cascaded composites may only aggregate components of their component type. This constraint may be implemented through assertions [11].

The whole-part pattern represents a form of aggregation where a whole aggregates parts of arbitrary types that form a semantic unit. The whole aggregates its constituent components (the parts), arranges collaboration between these, and provides a common interface, that ensures that parts are only accessed through the whole. The pattern may also be implemented using dynamic object aggregations. Filters, per-object mixins, or assertions are capable of hindering clients to access parts directly.

3.3 Miscellaneous Applications

Other applications for the dynamic object aggregation language construct, which we are currently investigating, are:

- *Role implementation*: Roles may be implemented through per-object mixins if they should operate on a shared data space, as shown in [9]. If each role needs its own data space, an implementation through aggregation can be used, as in the role-object pattern [2]. We are currently building a framework, which lets aggregated roles and the whole appear as an opaque object to clients.
- *Meta-level composition*: In meta-object protocols a base-level is controlled by a meta-level. Objects on the meta-level may be singular objects. Often a structuring into a chain can be found. Further structuring into a meta-level composer controlling its constituent parts can be achieved through aggregation. The meta-level composer can implicitly handle tasks, like concurrency or adaptations of meta-object operations.
- *Aggregating relationships*: In [3] the current view on aggregation is enhanced with the notion of relations that are parts of an aggregate and relations that are themselves aggregates, but no implementation concept for the constructs is given. It would be interesting to investigate in how far such a concept may be combined with a language support for aggregation.

4. CONCLUSION

We have presented a new language construct called dynamic object aggregations which language supports the widely used aggregation relationship. This relationship is mainly used to compose objects out of other objects. This form of composition can be found in almost every object-oriented software system. The language construct preserves the semantics and asserts the implied constraints of the aggregation relationship automatically. Therefore it eases the use of the relationship and prevents errors. Finally, we have presented an approach for sharing aggregates, described how to implement certain design patterns that rely on aggregation, and named open compositional problems, which may be solved through dynamic object aggregations. XOTcl is available from `http://nestroy.wi-inf.uni-essen.de/xotcl/`.

5. REFERENCES

[1] G. Banavar. Nesting as a form of composition. In *Proc. of CIOO Workshop at ECOOP*, July 1996.

[2] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. In *Proc. of 4th Conference on Pattern Languages of Programms (US)*, 1997.

[3] C. Bock and J. Odell. A more complete model of relations and their implications: Aggregation. *Journal of Object-Oriented Programming*, 11(5), September 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[5] T. Foster and L. Zhao. Cascade. *Journal of Object-Oriented Programming*, 11(9), Feb. 1999.

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

[7] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[8] T. Hartmann, R. Junghans, and G. Saake. Aggregation in a behavior oriented object model. In O. Madsen, editor, *Object-Based Distributed Processing*, pages 57–77. LCNS 615, Springer-Verlag, 1992.

[9] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.

[10] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, May 1999.

[11] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: 7th USENIX Tcl/Tk Conference*, Austin, Texas, February 2000.

[12] J. Odell. Six different kinds of composition. *Journal of Object-Oriented Programming*, 5(8), January 1994.

[13] J. Smith and D. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), 1977.