On-the-fly contextual adaptation with the RoleAdapter Pattern

Marc-Antoine Parent
Centre de Recherche Informatique de Montréal (CRIM)
550 Sherbrooke West, suite 100
Montréal (Québec) Canada H3A 2N4
+1 514 840 1234
maparent@crim.ca

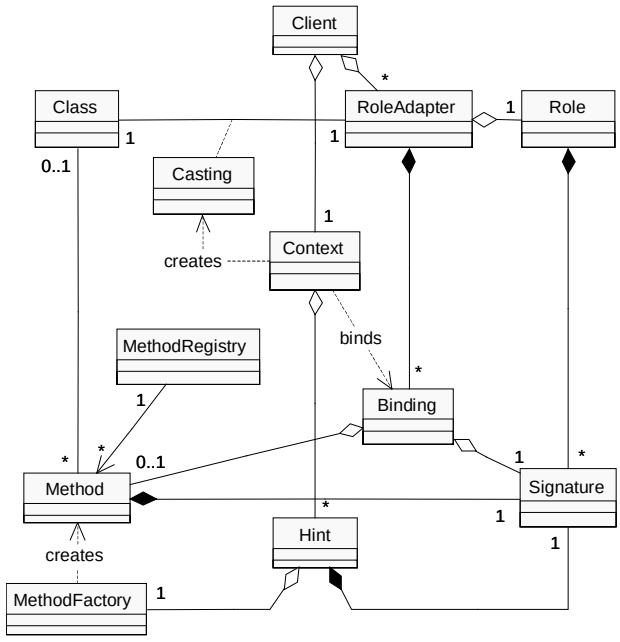
ABSTRACT
The RoleAdapter Design Pattern allows using objects of any model as if they implemented any given programmatic interface, with contextual behavior. To achieve this, it makes objects from many basic building blocks of OOP, like methods, method signatures, interfaces, etc. This allows clients of a data model to define, at run-time, an interface for any data model they have to use. Objects encapsulating methods, defined independently for the model, are chosen and bound to the signatures included in the interface according to external configuration hints. Since the adaptation is done local to a context, different view instances can show different aspects of a complex model. The resulting composite definition of interface is similar in intent to that of subject-oriented programming, but achieved wholly within a traditional OOPL like Java.

Keywords
Design Patterns, Roles, Subject-Oriented Programming

INTRODUCTION
The RoleAdapter pattern allows us to use objects of any model as if they implemented any given programmatic interface, with contextual behavior. Traditional OO design emphasizes that the data model should be generic enough to be used with many views. The views, on the other hand, must reflect the model's structure in their display, and hence are more closely tied to one model's structure. We cannot normally reuse view components with different models, unless they share an interface. In CASE tools, each node in a class diagram is involved in both an inheritance and an aggregation hierarchy. We would want to display the hierarchy obtained from these distinct interfaces in the same reusable display tool.

Ideally, each view we use should contribute to the model's interface at run-time, as in Subject-Oriented Programming [1]. Within conventional OO languages, many Design Patterns are aimed at altering a frozen object's interface. The original Adapter [2] does this, but only at compile-time, and it does not scale well; in a complex system of objects, one adapter is built for each object of the original system, relationships between adapted objects have to be translated into relationships between adapters, etc. Other relevant Patterns are Erich Gamma's Extension [3] pattern, Baümer and Riehle's Role objects [4], etc. Unfortunately, those patterns require the object to follow some extension protocol, which we wanted to avoid.

DESIGN
RoleAdapters are defined on arbitrary classes (not instances) at run-time. To that end, many basic constructs of OOP were made into first-class objects: the method implementation; its signature (name, return and parameter types); the role (or interface: the set of signatures that a class must implement when used in a certain role by a client), etc. (cf. diagram.)



The client expects to use objects according to a set of Signatures defined in a Role. Those Signatures will correspond to Methods. Binding objects, grouped in RoleAdapters, will contain the corresponding pairs. Initially, Bindings only contain Signatures; the unbound RoleAdapter is then equivalent to the Role. As a client attempts to make an instance of a certain class play a certain Role, it asks the Context to create a valid Class-RoleAdapter relationship (Casting) by fulfilling each of the RoleAdapter's Bindings with a Method that satisfies the Bindings's Signature.

In order to find appropriate Methods, each Context owns a distinctive set of Hints, containing factories that will attempt to build the Method from its Signature (if the required Signature fits that of the Hint.) Configuring different views' Contexts with different Hints allows for fine-grained simultaneous access to various aspects of a model at run-time. The MethodFactory's job may be simple, like creating a Method of a given class, or retrieving an existing Method with an alternate Signature from the Context, but more complex MethodFactories allow us to specify arbitrarily complex derived Method objects.

If no Hint specifies how to build our Method, the Context queries the context-independent MethodRegistry for a model-specific Method that adequately fulfills the Binding. The set of those model-specific Methods amounts to a dispersed adapter for the model. The candidate Casting, and any Casting registered previously, establish equivalence classes between the Signatures of those Methods defined strictly in terms of classes, and more abstract Signatures owned by, and defined in terms of Roles. In case of failure, the Context asks a broader Context that may have further Hints (using Chain of Responsibility.)

After the binding phase, the client can access the Method through the RoleAdapter as follows:

```
roleAdapterXXX.bindingYYY.theMethod.
              execute(objectZZZ, params...);
```

instead of, traditionally:

```
objectZZZ.methodYYY(params...);
```

The latter code is also what will often be found in the Method object's `execute` method. But the flexibility we gain offsets the cost of doubling all virtual calls.

Defining Methods as first-class objects allows us to build and define complex structures from them. For example, we can compose links between objects through Object Composition on accessor-like Methods. Similarly, we have found it useful to define some setter-like Methods to be observable, through wrapping them in an observable Decorator. We also use the Observer pattern to create dynamic Methods, from a Binding that observes the Context for changes in the Hint definitions. In all cases, these complex Methods are used interchangeably with the atomic ones. We believe that these objects are an interesting Design Pattern in their own right, inspired by Strategies and reminiscent of ValueModels [5].

CONCLUSION

Using this pattern, we believe that we have achieved a sizeable fraction of the advantages of subject-oriented programming within the confines of a traditional object-oriented language. Readers interested in the uses of the RoleAdapter pattern should refer to papers on the Giza framework [6].

ACKNOWLEDGMENTS

The author wishes to thank CRIM for its support of our team's research in information visualization. Many thanks go to Luc Beaudoin, who made Giza necessary by inventing innovative visualization techniques. The project to apply them to diverse data models received support from Frances de Verteuil, director of the HCI group at CRIM. Many people at CRIM contributed to the Giza architecture in various ways, most notably Louis Vroomen who made it yield visible results long before it should have. Ruedi Keller, of Université de Montréal, also offered useful input.

REFERENCES

1. Harrison, W. and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects), in *Proceedings of OOPSLA'93* (Washington DC, 1993), ACM Press, 411-428
2. Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading MA, 1995.
3. Gamma, E. Extension Objects, in R. C. Martin, D. Riehle, and F. Buschmann (eds.). *Pattern Languages of Program Design 3*, (Reading, MA, 1998), Addison-Wesley. See also Johnson, R. E. Facet. Available at http://st-www.cs.uiuc.edu/users/johnson/facet/facet.html.
4. Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. Role Object, in *Proceedings of PLoP '97*. Tech Report WUCS-97-34. Washington University, Dept. of Computer Science, 1997. Paper 2.1, 10 p.
5. The discrete charm of ValueModels, in *ParcNotices 4*, 2, (Sunnyvale, CA, Summer 1993), ParcPlace Systems, 1, 8-9.
6. Papers describing the Giza framework are available at http://www.crim.ca/~vroomen/mainPages/visual/giza.html.