DYNAMIC OBJECT MODEL

R E Johnson

Rapporteurs: Dr Robert Stroud and Dr Rogério de Lemos

DYNAMIC OBJECT MODEL

Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
johnson@cs.uiuc.edu

Introduction

Recently I have seen many examples of a type of architecture that was new to me. Half of the demonstrations at OOPSLA'97 were examples of this architecture. I have not found any descriptions of this architecture, yet the number of systems that I have seen indicates that it is widely used. This architecture leads to extremely extensible systems, often ones that can be extended by non-programmers. Like any architectural style, it has costs. It is not efficient of CPU time, but is usually used where efficiency isn't important. A bigger problem is that the architecture can be hard for new developers to understand. I hope this paper will help eliminate this problem.

The architecture has many names, sometimes called just a "reflective architecture" or a "meta-architecture". However, it is more specific than just a reflective architecture. It was called the "Type Instance pattern" in a tutorial at OOPSLA'95[GHV95]. This paper calls it the "Dynamic Object Model architecture". Most of the systems I have seen with a Dynamic Object Model are business systems that manage products of some sort and are extended to add new products, so I have called it the "User Defined Product architecture" in the past[JO98].

Most object-oriented systems have a static object model. In other words, the object model does not change at run-time, but is fixed when the program is designed. A system based on a Dynamic Object Model stores an object model in a database and interprets it. Changing the object model will immediately result in a changed behavior. The object model is usually easy to change because there are usually special purpose user interfaces for changing it.

The Dynamic Object Model has been used to represent insurance policies[JO98], to bill for telephone calls, and to check whether an equipment configuration is likely to work. It has been used to model workflow[DT98], to model documents, and to model databases.

The Structure of the Dynamic Object Model

The Dynamic Object Model architecture is made up of several smaller patterns. The most important is Type Object, which separates an Entity from an EntityType. Entities have Attributes, which are implemented with the Property pattern, and the Type Object pattern is used a second time to separate Attributes from AttributeTypes. The Strategy pattern is often used to define the behavior of an Entity Type. As is common in Entity-Relationship modeling, a Dynamic Object Model usually separates attributes from relationships. Finally, there is usually an interface for non-programmers to define new EntityTypes.

Type Object

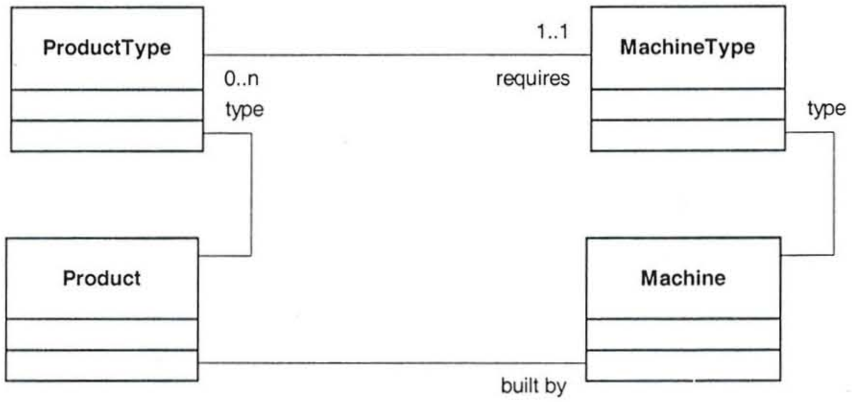
Most object-oriented languages structure a program as a set of classes. A class defines the structure and behavior of objects. Most object-oriented systems use a separate class for each kind of object, so introducing a new kind of object requires making a new class, which requires programming.

However, there is often little difference between new kinds of objects. If the difference is small enough, the objects can be generalized and the difference between them described by parameters.

For example, consider a factory scheduling system for a factory that makes many kinds of products. Each product has a different set of raw materials and requires a different set of machine tools. The factory has many kinds of machines, and has varying numbers of each. Each type of product would have a plan that indicates how to build it. The plan indicates the types of machines that are needed, but not the particular ones that are to be used. The factory scheduling system takes a set of orders and produces a schedule that ensures those orders are built on time. It assigns each order to a particular set of machines, checking that there are enough machines of a particular type to do all the work needed in a day. When the factory builds a product, it might record its BuildHistory so that quality control inspectors will know the exact machines that were used to build it.

One way to associate plans with products is to introduce a subclass of Product for each type of product, and to define an operation in each subclass to return the plan. In the same way, there would be a subclass of Machine for each type of machine. However, the only difference between MachineTypes is the number of instances and their name. Further, a plan needs to refer to machine types, and some languages (like C++) make it hard to have an object point to a class or to create an object from a class with a particular name. There should be a MachineType object that knows all the machines in the factory of a particular type. A Plan will refer to a MachineType either by name or by direct reference. A system for designing Plans might require more information about a MachineType, but a system for scheduling will not. If MachineType is a separate class then Machines are general enough that there is no reason to subclass them. In the same way, the only difference between types of products is probably the plans used to make them. It is not necessary to make a subclass of Product for each type of product; make a class ProductType and create instances of ProductType instead of subclasses of Product.

Manufacturing model



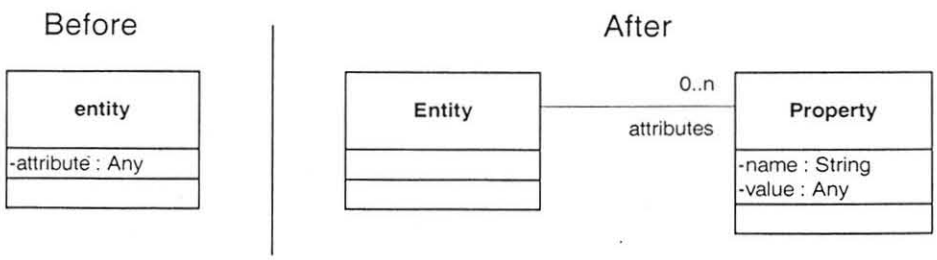
The Type Object pattern split a class into two classes, one the type of the first, and then to replace subclasses of the original with instances of the type class. It can be used in the factory scheduling system to replace subclasses of Product and Machine with instances of ProductType and MachineType. It can be used in an airline scheduling system to replace subclasses of Airplane with instances of AirplaneType (Coad 1992). It can be used in a telecommunications billing system to replace subclasses of NetworkEvent with instances of NetworkEventType. In all these cases, the difference between one type of object and another is primarily their data values, not their behavior, so the Type Object pattern works well.

Property

The attributes of an object are usually implemented by its instance variables. A class defines the instance variables of its instances. If objects of different types are all the same class, how can their attributes vary?

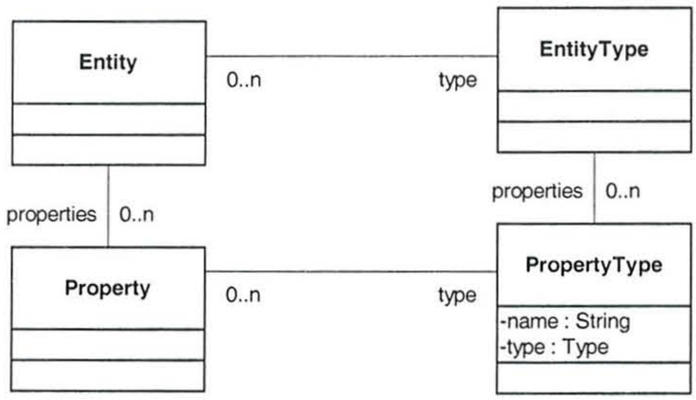
The solution is to implement attributes differently. Instead of each attribute being a different instance variable, make an instance variable that holds a collection of attributes.

Property Pattern



The core of a Dynamic Object Model is a combination of Type Object and Property. The Type Object pattern divides the system into Entities and EntityTypes. Entities have properties. But usually each property has a type, too, and each EntityType then specifies the types of the properties of its entities. A PropertyType is usually more like a variable declaration than like an abstract data type. It often keeps track of the name of the property, and also whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following:

Dynamic Object Model



Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of Properties to represent a single record, and can use RecordType and PropertyType to represent a table.

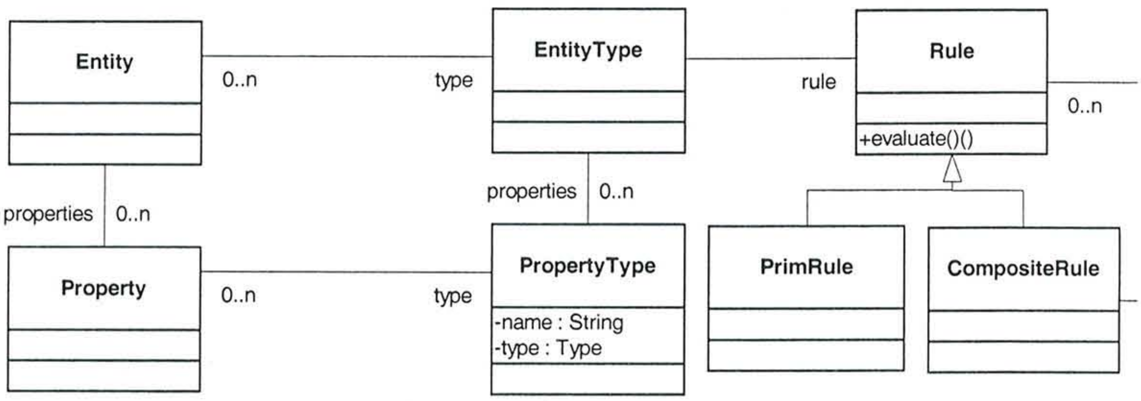
But usually different kinds of objects have different kinds of behaviors. For example, maybe records need to be checked for consistency before being written to a database. Although many tables will have a simple consistency check, such as ensuring that numbers are within a certain range, a few will have a complex consistency checking algorithm. Thus, Property isn't enough to eliminate the need for subclasses. A Dynamic Object Model needs a way to change the behavior of objects.

Strategy

A strategy is an object that represents an algorithm. The strategy pattern defines a standard interface for a family of algorithms so that clients can work with any of them. If an object's behavior is defined by one or more strategies then that behavior is easy to change.

Each application of the strategy pattern leads to a different interface, and thus to a different class hierarchy of strategies. In a database system, strategies might be associated with each property and used to validate them. The strategies would then have one public operation, validate(). But strategies are more often associated with the fundamental entities being modeled, where they implement the operations on the methods.

Dynamic Object Model



Entity-Relationship

Attributes are properties that refer to immutable values like numbers, strings, or colors. Relationships are properties that refer to other entities. Relationships are usually two-way; if Gene is the father of Carol then Carol is the daughter of Gene. This distinction, which has long been a part of classic entity-relationship modeling and which has been carried over into modern object-oriented modeling notations, is usually a part of a dynamic object-model architecture. The distinction often leads to two subclasses of properties, one for attributes and one for relationships.

One way to separate attributes from associations is to use the Property pattern twice, once for attributes and once for associations. Another way is to make two subclasses of Property, Attribute and Association. An Association would know its cardinality. A third way to separate attributes from associations is by the value of the property. Suppose there is a class Value whose subclasses are all immutable. Typical values would be numbers, strings, quantities (numbers with units), and colors. Properties whose value is an Entity are associations, while properties whose value is a Value are attributes.

Although this is a common pattern, I am not sure why it is used. Perhaps it is just a more accurate model. Or perhaps it is used by habit because designers have been trained in Entity-Relationship modeling. It is interesting that few language designers seem to feel the need to represent these relationships, but most designers of systems with dynamic object models do.

User Interface for Defining Types

One of the main reasons to design a Dynamic Object Model is to extend the system by defining new types without programming. Sometimes the goal is to enable users to extend the system without programmers. But even when only the developers will define new types, it is common to build a specialized user interface for defining types. For example, the insurance framework at the Hartford has a user interface for defining new kinds of insurance, including the rules for calculating their price. Innoverse, a telephone billing system, has a user interface for defining geographical regions, monetary units, and billing rules for different geographical regions expressed in various monetary units. The Argos school administration system lets has a user interface for defining new document types and workflows.

Types are often stored in a centralized database. This means that when someone defines new types, applications can use them without having to be recompiled. Often applications are able to use the new types immediately, while other times they cache type information and must refresh their caches before they will be able to use the new types.

The alternative to having a user interface for creating and editing type information is write programs to do it. In fact, if programmers are the only ones creating type information then it is often easier to let them do it by writing programs, since they can use their usual programming environment for this purpose. But the only way to get non-programmers to maintain the type information is give it a user interface.

Advantages of Dynamic Object Models

If a system is continually changing, or if you want users to be able to extend it, then the Dynamic Object Model architecture is often useful. The alternative is to pick a simple programming language that is flexible and easy to learn. In fact, a Dynamic Object Model is a kind of programming language. Visual Basic could be thought of as based on a Dynamic Object Model, though its internals are hidden and it is hard to be sure.

Systems based on Dynamic Object Models can be much smaller than alternatives. One architect told me that his 50,000 line system had more features than systems written without a dynamic object model that took over 3 million lines of code. I am working on replacing a system with several millions lines of code with a system based on a dynamic object model that I predict will require about 20,000 lines of code. This makes these systems easier to change by experts, and (in theory) should make them easier to understand and maintain.

Disadvantages of Dynamic Object Models

A Dynamic Object Model is hard to build. The systems that I've seen use it have all been designed by experienced architects. What happens when the system is maintained by less experienced programmers? These systems are often hard for less experienced developers to understand. This is by far the biggest disadvantage of this architecture, and architects should choose it cautiously and plan to spend more than usual on documentation and training.

A system based on a Dynamic Object Model is an interpreter, and can be slow. Most of the systems I've seen have been fast enough with only a little optimization. However, I've also seen a few in which some of the features were too slow.

A system based on a Dynamic Object Model is defining a new language. It is a domain-specific language that is often easier for users to understand than a general-purpose language, but it is still a language. When you define a new language, you have to define support tools like a debugger, version control, and documentation tools. This is extra work. If you let users define their own types, you have to teach them good software engineering practices like testing, configuration control, and documentation. Is it worth the effort? Some designers do not worry about this and their projects usually come to a bad end. Others avoid these problems by only allowing developers to define new types. Others train their users. There are many ways around this problem, but it is a problem that should be faced and not ignored.

Summary

A Dynamic Object Model provides an interesting alternative to traditional object-oriented design. Like any architecture, it has both advantages and disadvantages. The more examples we study, the better we will understand its strengths and weaknesses. Please contact me if you have used this architecture in the past and can provide more examples or if you know of any papers that describe this architecture or aspects of it.

Bibliography

[Coad 92] Peter Coad, "Object-Oriented Patterns". *Communications of the ACM.* 35(9):152-159, September 1992.

[DT98] Martine Devos and Michel Tilman, *Repository-based Framework for Evolutionary Development*, 1998. http://www.argo.be/OoFrame/

[FY98] Brian Foote and Joseph Yoder, *Metadata and Active Object-Models,* At PloP'98, Allerton Park, August 1998. Also http://www-cat.ncsa.uiuc.edu/~yoder/papers/patterns/Metadata/metadata.pdf

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GHV95] Erich Gamma, Richard Helm, and John Vlissides, *Design Patterns Applied*, tutorial notes from OOPSLA'95.

[JO98] Ralph E. Johnson and Jeff Oakes, The User-Defined Product Framework, 1998. http://st.cs.uiuc.edu/pub/papers/frameworks/udp

[JW97] Ralph Johnson and Bobbie Woolf, Type Object, In *Patttern Languages of Program Design* 3, Robert Martin, Dirk Riehle, and Frank Buschmann ed., Addison-Wesley, 1997, pp. 47-66.

DISCUSSION

Rapporteur: Dr Robert Stroud

Lecture One

During his presentation, Professor Johnson remarked that it was well known that multiple inheritance was a bad solution to a particular problem. Given that this was community knowledge, Professor Balzer wanted to know how a pattern would point people away from using multiple inheritance. Professor Johnson replied that the circumstances for using multiple inheritance were domain specific - it was known that multiple inheritance could get you into trouble but it could also work nicely in some circumstances and perhaps rules could be written to explain this.

Professor Balzer wasn't satisfied with this answer and wanted to know how this knowledge was documented. Mr Jackson rephrased the question in the following way - each pattern describes the circumstances in which it is applicable but what about the circumstances when it wasn't applicable and would be a bad design choice? Professor Johnson replied that everything was bad in some circumstances and you could usually figure out what the drawbacks were for a given pattern by thinking about it. However, he agreed that it was important to describe the drawbacks as well as the advantages of each pattern.

Commenting further on the issue of multiple inheritance, Professor Johnson said that plenty of people had written about multiple inheritance and when it was useful. Java put sensible restrictions in the language to constrain the use of multiple inheritance - in particular, you could use multiple inheritance for different interfaces but not for merging implementations where you needed more control. In contrast, the Lisp community liked automatic merging of implementations but whilst this might be OK for an MIT PhD student, it wasn't something for the average C++ programmer which was why the C++ gurus say "don't do this". Java went further and put the rules into the language. MIT PhD students might find this limiting but it was probably a good design decision for Java.

Returning to the original question, Professor Johnson said that there was knowledge and it could probably be generalised but tended to be domain specific. A programmer in the insurance industry needed to know what was useful and worked well in their particular domain.

Professor Shaw had a question about the exposition of patterns. She agreed that there was a place for describing the circumstances in which a pattern was applicable but it was also necessary to talk about the domain. How should this information be organised - by pattern or by domain? Professor Johnson replied that the material should be organised in whatever way communicated it best to the reader. However, you shouldn't try and put too much into a pattern - the Design Patterns book probably represented an upper limit of what people could handle. But you needed to figure out how to organise the descriptions so that people could easily access the useful bits.

Mr Jackson said that this wasn't just a matter for the writer - it was important that the exposition should match the reader's expectations. Professor Johnson felt that the problem was how to find useful patterns, some rough, some polished. If you wanted to get some information about what was on at the movies, you tended to ask people who liked the same kind of thing that you did. So having this kind of guide was useful. Professor Randell remarked that the ACM collections of subroutine libraries represented nearly 50 years experience of this kind of thing in Computing.

Later in the talk, Professor Johnson described the Strategy pattern and Professor Balzer asked if the change of strategy was just a switch and didn't involve changing details. Professor Johnson agreed that the Strategy pattern did not help you to choose the algorithm

- you could parameterise strategy objects but this would be a different use of the pattern. Professor Randell remarked that these all sounded like binding issues and Professor Johnson agreed that binding was a very important part of many patterns.

Dr Kay remarked that the Active Object-Model architecture that Professor Johnson had described was identical to the schema for ICL's data dictionary which had been shipped 20 years ago. Professor Johnson agreed that the architecture could be used to solve that kind of problem but said that it had the drawback of taking more bits, being less efficient, and being harder to understand because it was more abstract. Dr Kay said that the biggest disadvantage was that query optimisers didn't understand it. Professor Johnson agreed but said that this hadn't been a problem for the particular instance of the pattern he had been describing because the programmers had been using an OODBMS and writing queries in a high level language.

Professor Shaw asked for clarification about the relationship between components and their children. She was concerned that the diagram was structural but had a behavioural interpretation.

A member of the audience commented that the diagram was useful for run-time type checking and that this was appropriate if you were happy to do your own type checking. Professor Randell drew an analogy with his earlier remarks about binding. Professor Johnson agreed, saying that another way of thinking about this was that you were effectively inventing a new language. However, people were reluctant to think of it in these terms, especially in traditional DP shops.

Professor Hall remarked that back in the old days of traditional databases, this might have been called a meta schema. There was a need to talk about the processes underlying an architecture, levels of architectural description and choices about binding decisions. The issue was whether to do this automatically or manually by encoding the rules in the schema. Professor Johnson agreed that this was an important issue and something that he saw a lot. However, it wasn't clear to him how this could be raised to a principle and taught.

Professor Dobson observed that in many cases you might find that the rules encoded in the model were owned by some separate regulatory authority rather than being properties of the types themselves - you might want to make this clear in the model. Professor Johnson felt that the rules belonged to variables and should be kept together with them. Rules needed to be expressed as constraints on things and it might be necessary to change variable names to keep things consistent. However, he accepted that the different points of view were an important issue and remarked that a related Telephone Company example was more complicated because it involved different levels and issues of who owned the meta information.

Dr Kay remarked that one of the reasons that people built such models was because their programming languages didn't have reflection. Professor Johnson agreed but observed that ironically this work had been done in Smalltalk which was probably the most reflective programming language used in business! Programmers were reluctant to accept that what they were doing was inventing another OOPL. Although they did use some of the reflective facilities in Smalltalk, they tended to apologise for doing so - in any case, these patterns didn't depend on the use of reflection. He thought this was a cultural thing - C++ programmers wouldn't do this.

Dr Kay replied that some people tried to do computation in SQL. Professor Randell observed that people used whatever tool they were used to. He recalled the joke about an engineer being someone with hammer looking for a nail and said that we shouldn't be ashamed about this but should be prepared to learn the limitations of our tools. For example, it would not be appropriate to install windows using a hammer. To general laughter, Dr Perry joked that on the contrary, installing (Microsoft) Windows with a hammer was probably about the best thing to do with it!

DISCUSSION

Rapporteur: Dr Rogério de Lemos

Lecture Two

During the talk Professor Randell mentioned that twenty or thirty years ago in order to solve a problem, it was fashionable to invent a new language or to incorporate into an existing language new mechanisms that would encapsulate features that before were programmed by hand, and it was noticed at the time that in the design of fashionable languages very few people could do it very well while the rest did some dreadful things. He continued by querying whether there was a degree of similarity with the process of identifying and publishing new patterns. Professor Johnson agreed with the statement, and added that once a language has been around for a long time it is difficult to make people use new features that were created in a research environment.

After Professor Johnson presented the way in which Christopher Alexander's architectural patterns had influenced the ideas around design patterns, Professor Brooks emphasised that patterns really worked. Professor Randell made the comment that Christopher Alexander, in one of his book "Notes in the Synthesis of Form", had made an attempt to produce a logical framework to solve problems that would only work if the solution to the problem was known beforehand. He also said that years after of the book was published, Alexander himself admitted that some of ideas of the book were impractical. Professor Brooks added that the power in pattern languages is the knowledge associated with patterns.

Mr Jackson asked the speaker to clarify the following conflicting view: on the one hand, there was the impression that the twenty three patterns described originally in the book "Patterns in Concurrent Programming" plus the two or three patterns presented on the previous talk by the speaker appeared to be sufficient, while on the other hand, there was this ambition to identify and publish even more patterns. Professor Johnson answered that the aim was to write patterns that can be used by everyone, perhaps a thousand patterns, but he did not know the right number, he also mentioned that the patterns described in the book were specifically for object-oriented programming.

After Professor Johnson presented the benefits of using design patterns, Dr de Lemos asked whether the speaker could also enumerate some negative aspects. Professor Johnson answered that there were a lot of people using patterns without understanding them, and that patterns have been defined without a coherent form of organisation.