

Chapter TBD

Dynamic Object Model

Dirk Riehle

Bayave Software GmbH
dirk@riehle.org

Michel Tilman

Real Software
mtilman@acm.org

Ralph Johnson
Computer Science Department
University of Illinois
johnson@cs.uiuc.edu

Intent

Allow a system to have new and changing object types without having to reprogram the system. By representing the object types as objects, they can be changed at configuration time or at runtime, making it easy to change and adapt the system to new requirements.

Also Known As

Object System, Runtime Domain Model, Active Object Model, Adaptive Object Model

Motivation

Consider a banking system for handling customer accounts like checking or savings accounts. One option is to design a hierarchy of account classes, starting with a root class Account. However, banks provide many different types of accounts. It is not uncommon for a large bank to have more than 500 types of accounts. Many of these account types vary only by a few attributes, but these differences are important to the bank and need to be represented.

Rather than implementing 500 account classes, you decide to use the Type Object¹ pattern [Johnson+1998] to represent each class as an object. You introduce a class AccountType whose instances represent a specific type of account, and a class Account whose instances represent a specific account of a customer, as shown in Figure 1. Instances of AccountType serve as type objects for instances of Account. All properties that are the same for a specific type of account go into the AccountType class (name of this type of account, interest rate for this type of account,

¹ The Type Object pattern centralizes common information about a set of instance objects in a Type Object that is shared by all instances. Type Objects provide application domain specific information, such as behavior shared by their instances or a description of the list of properties of their instances, rather than implementation information.

etc.). All properties that may vary within instances of the same AccountType go into the Account class (account number, current balance, etc.).

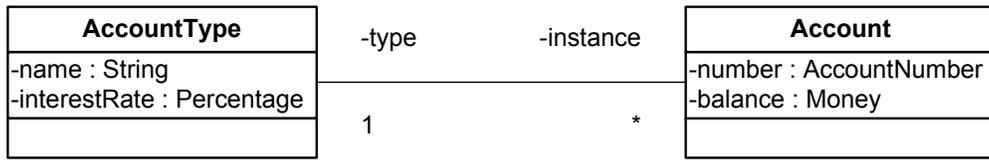


Figure 1: Account and AccountType class.

After a while you recognize that the Account and AccountType classes get bloated with a large number of fields and methods, most of which remain unused most of the time. After all, the two classes represent the union of some 500 account types! To slim down the Account class, you decide to use the Property List² and Value Holder³ patterns [Riehle1997a, Foote+1998].

The Account class now holds a collection of instances of the new Property class. An instance of the Property class represents the value of an attribute that was formerly an attribute of the Account class, see Figure 2. The Property instances stores these values as generic Object instances, and it is up to a client of an account to properly interpret this value.

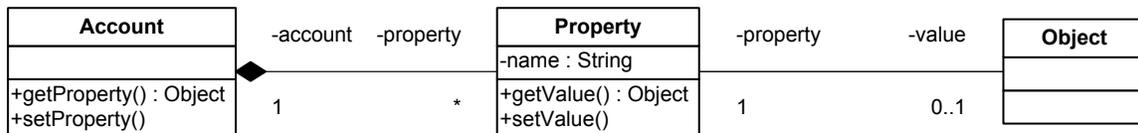


Figure 2: Account with Property and Value objects.

Properties of Account like ‘name of owner’ or ‘balance’ now become instances of a generic Property class. Their name is stored in the name attribute of the Property class, and the Account class uses it to select a Property instance. A Property instance can hold any object that represents the value of the property.

However, something still worries you: The properties are not being type-checked. Therefore, a client might mistakenly set a string as the value of the balance of an account. So you need to check whether a value provided for a property is of the correct type, is in a proper range of values, etc. Also, you need to define whether a certain type of Property is acceptable for an Account in the first place.

For example, a Swiss number account may not have an owner name. It does not have a property “owner name” and must not be given one. Thus, you use Property List again and define a collection of PropertyType objects for AccountType so that an Account instance can check with its AccountType type object whether a specific Property is acceptable or not.

² Objects implementing the Property List pattern contain a generic and extensible set of properties. Clients may access the list of properties, and use a naming scheme (e.g. Strings) to set and get properties. Properties may even be added or removed at runtime. An example implementation is the Java HashMap.

³ The Value Holder pattern abstracts the concept of a variable in programming languages. It is often used in combination with the Observer pattern. This allows interested parties to be notified when the value of the ‘variable’ changes.

For this, we use the Type Object pattern again, and use the type objects to check access to a property. Any given AccountType can now have a collection of PropertyType objects, which represent the types and their allowed values for a given account type. Any property, dangling of an account object, can be checked for validity through the account's type object, which provides the property type objects.

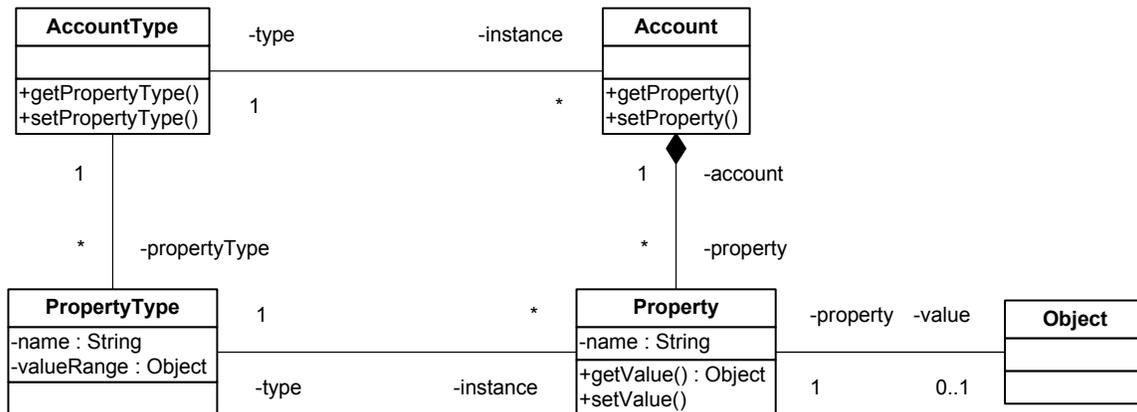


Figure 3: The Account and AccountType classes with their type definitions and instance properties.

What can be said about the design, as shown in Figure 3? First, we distinguish a type level from an instance level. On the left of each figure, we see the type objects, and on the right, we see the instance objects. Fowler also calls the type level the “knowledge level” and the instance level the “operational level” [Fowler1996]. Effectively, the type level is a model (on the left of the diagrams) of what makes up valid instances of the model (on the right of the diagrams).

This design lets you introduce new types of accounts at runtime without programming new classes. You simply create a new instance of AccountType and configure it with its PropertyType objects. You could now sit together with some bankers and explore new ideas for accounts and their behavior in real-time (probably in a computational sandbox, though).

The AccountType class in Figure 3 no longer mentions the ‘name’ and ‘interest rate’ attributes, so where did they go? We want to identify each type of account, so we retain the name attribute for all account types. Not all type of accounts have interest rates though, but savings accounts do. Hence we introduce the class ‘SavingsAccountType’ that subclasses AccountType with an extra ‘interest rate’ attribute. Now each SavingsAccountType has an interest rate attribute representing a value shared by all instances of the SavingsAccountType. At the instance level we perform the same analysis. Some attributes (not their values), like balance, are shared by all instances, other attributes may be shared only by specific instances of Account. In both cases we have to decide whether to represent these attributes as dynamic property types or as regular member fields of Account or of a subclass thereof (see the Sample Code section for a more detailed description).

Problem

The Dynamic Object Model pattern solves several different problems. Some systems only have one of these problems, others have several:

- A system is difficult to understand, change, and evolve, because it is complex. The system seems complex because there are so many types of objects, but they differ only in a few fields.
- A system requires frequent changes and rapid evolution. New types of objects must be created at runtime. For example, end-users may need to specify these new types of objects and they need to interact immediately with the objects without having to rebuild the system.
- A system needs a domain-specific modeling language, perhaps because it should be used by end-users, perhaps because it needs custom type validation, or perhaps because it needs to generate complex behaviors from that model.

Usually a Dynamic Object Model starts out as a way of making a system simpler and easier to change. Later it becomes apparent that it is possible for users to specify the changes without involving programmers, and then it becomes apparent that the system now has a domain-specific modeling language. But some Dynamic Object Models do not allow end-users to define new types, and sometimes a Dynamic Object Model starts with the need for a domain-specific modeling language.

Languages like Smalltalk support class modification at runtime, even when classes have instances. They also allow developers to adapt-within limits-the meta-model describing how classes look like and how they behave. Java-like languages on the other hand are much more limited in this regards, even disregarding class incompatibility problems resulting from serialization in a multi-version environment. So why not use one of these more dynamic programming languages? For a start, many customers have already standardized on one of the more widely used, but often more static, languages, and introducing yet another language is usually not an option. But the problem goes deeper than this, as it involves more than the mere technical issue of choosing the right implementation language. Usually one of the main underlying motivations of Dynamic Object Models is to bring ‘configuration’ of the solution closer to the end-user. In several of the Known Uses examples, for instance, domain experts participate heavily in the process of developing end-user applications. Imposing general purpose programming languages and traditional IDE’s on these users then misses the stated goals. As a result, systems driven by Dynamic Object Models are often accompanied with a set of domain-specific high-level development tools.

Solution Structure

The core of the Dynamic Object Model pattern consists of the classes Component and ComponentType, Property and PropertyType, and their respective clients. It is a composite (compound) pattern consisting of several other patterns. Figure 4 shows that it is composed of the Type Object, Property List, and Value Holder patterns. Figure 4 uses UML collaboration specification diagrams to show the participating patterns. Each collaboration specification is displayed as a dashed ellipsis.

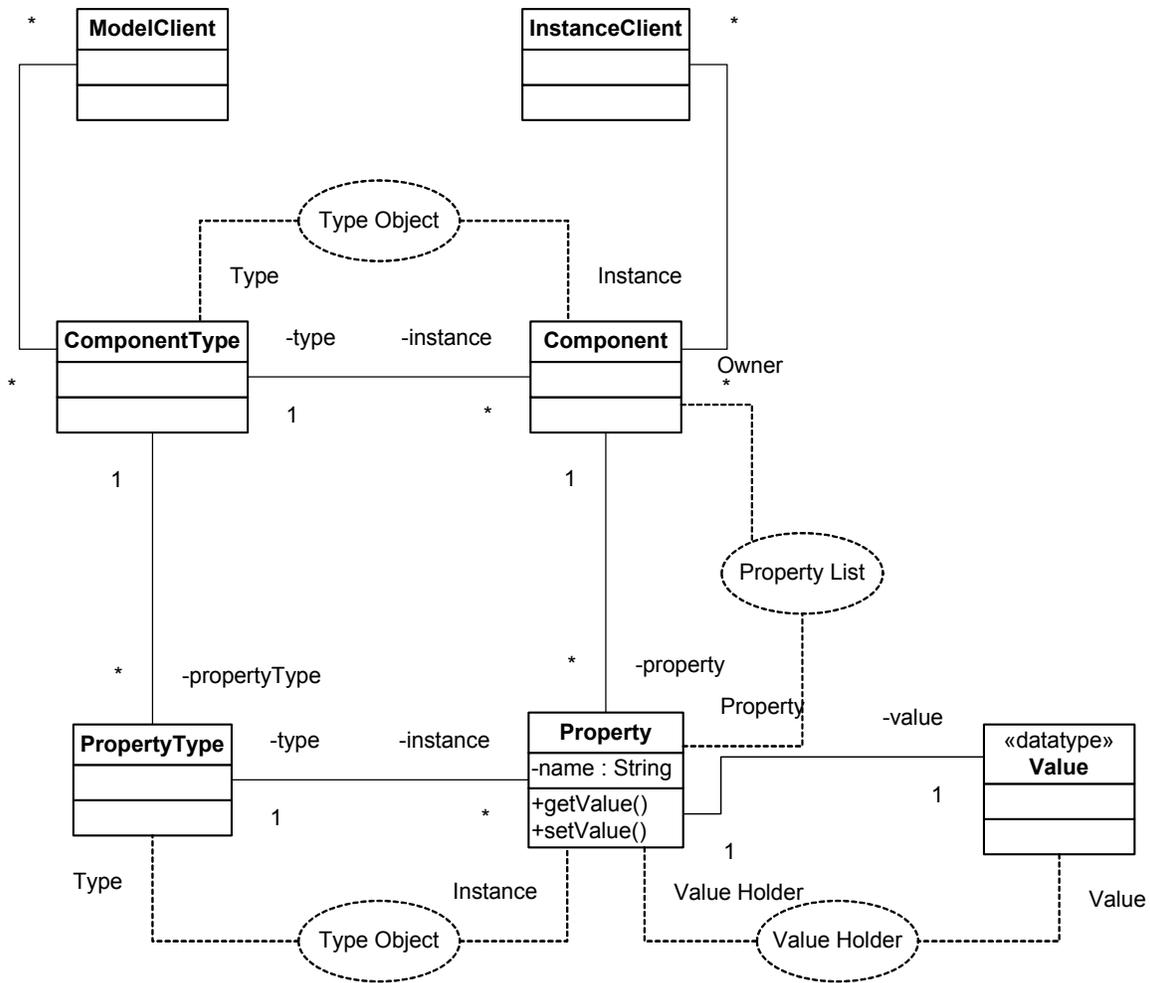


Figure 4: Structure diagram of the Dynamic Object Model pattern.

Figure 4 shows how the Type Object pattern separates the type level from the instance level. In the Type Object pattern, an Instance object delegates type-specific behavior to its Type object [Johnson+1998]. Client objects may work both with the Instance and the Type object. The Type object serves as a specification of what is acceptable to its Instance objects. New instances of the Type class and hence new types of objects can be introduced at runtime.

In the Property List pattern, an Owner object maintains a set of Property objects that Client objects may request to learn more about the Owner object [Riehle1997a]. Property objects are got and set dynamically, typically using strings as property names. The Property List pattern allows for runtime extension of the Owner's properties.

In the Value Holder pattern, a Client object retrieves a Value object from a Value Holder object [Foote+1998]. The Value Holder serves as an adapter for the Value that it makes available to the Client object using a homogeneous interface. Value objects can be any kind of value, primitive or non-primitive [Cunningham1995, Bäumer+1998].

Thus, a Component class (Account in the motivation section) is the *Owner* of a set of property objects, the *Client* of a property object acting as a value holder, and the *Instance* of the ComponentType object.

The Property class (Attribute in the motivation section) is the *Property* of an account object, the *Instance* of a PropertyType object, and the *Value Holder* of value objects.

The Value class is the *Value* of a Property.

The ComponentType class (AccountType in the motivation section) is the *Type* of a Component instance and the *Client* of its PropertyType objects.

The PropertyType class (AttributeType in the motivation section) is the *Type* of its Property instances.

Tradeoffs

The Dynamic Object Model has both advantages and disadvantages.

Simplicity

The Dynamic Object Model pattern reduces the number of ‘real’ classes. This is a design issue, much in the same way a developer might factor out common orthogonal variations on a theme by means of plug-in components. As such this approach is essentially independent of the underlying programming language. This reduction is sometimes several orders of magnitude. However, a system based on the Dynamic Object Model has its own kind of complexity. It is a composite pattern, which means that it uses several patterns that are tightly woven together. Many programmers do not know these patterns and find the system hard to understand. The Type Object pattern is particularly hard for programmers of traditional object-oriented languages because it represents as instances what are normally represented as classes. A Dynamic Object Model represents one logical class as a set of instances; one instance of ComponentType and several instances of PropertyType. For a programmer, this design is more complex and harder to understand than a traditional class inheritance hierarchy. When programmers understand the design, it seems simple and easy to change. However, it can take a long time for a programmer to understand it. This is even true for, say, Smalltalk programmers, who are fairly accustomed to classes being first-class citizens.

The Dynamic Object Model pattern provides a complete, explicit model that can be consulted at runtime to automate many functions. When developing a generic store-retrieve relational database application, for instance, you can generate forms, query screens and SQL statements on the fly. Adding a new feature often means extending the Dynamic Object Model in a single place rather than changing every type of domain object. For this to work, your model must be able to describe things like properties, relationships and totality constraints, and you must be able to describe how object structures and data types are mapped onto the database. Up to a certain level we can also do this in statically typed languages like Java. But, as the Dynamic Object Model pattern leaves all this up to the system designer, these model elements can all be domain-specific, which makes them both easier to implement (because they don’t have to be fully general) and easier to understand. Nevertheless, as the model grows more detailed, it becomes harder to understand.

Flexibility

Type definitions can be created at runtime and can be changed at runtime. This makes applications easy to extend and easy to change. Part of the type definition can be constraints between types, and these can be changed at runtime as well. Of course, the more that can be

changed at runtime, the harder it is to implement the system. For example, perhaps existing objects must be updated. Can objects change their type? All things are possible, but not all things are expedient. As a system gets more flexible, it gets slower. A system based on the Dynamic Object Model has the potential to be very slow. Most systems are limited by something other than the Dynamic Object Model, such as a database, a network, or the user, but there is always a performance penalty in both time and space. The designer has to decide how flexible to make the system. Most systems that use a Dynamic Object Model find that lack of understandability is a bigger problem than lower performance, but this might be because designers are more afraid of performance problems than of understandability problems.

Languages like Smalltalk offer many of these benefits while still offering sufficient performance for most types of business applications. But without discipline this flexibility may lead to chaos. Systems using Dynamic Object Models may encapsulate these features in components performing, for instance, some sort of consistency checks. Most of the systems described in the Known Uses section use dynamic facilities of the implementation language, but they typically use them sparingly and in a tightly controlled way.

End-user configuration

The Dynamic Object Model pattern lets end-users define key concepts from their application domain at runtime, without lengthy development cycles in between. The Dynamic Object Model pattern acts like a (possibly domain-specific) language for users to describe their domain problems. End-users need their own specialized development environment, because classic development environments are designed for software developers. Although end-user configuration means that programmers have less work to do, when a new feature cannot be added by the user then it is harder for the programmer to know how to add it. Should the Dynamic Object Model be extended? Should the entire feature be added to the Dynamic Object Model, or only part? Perhaps the end-users could have implemented the feature themselves, but just didn't know how?

End-users usually don't understand the importance of testing, documentation, and configuration management. If they understand the importance, they don't know how to do it. The end-user environment must help them with these activities. End-users will make mistakes, so sometimes they will report a bug when the mistake is really their own. End-user configuration is more likely to be successful when there is close communication between the end-users and the programmers developing the core of the system.

Programming Environment

Programmers can no longer rely on their familiar development tools, such as browsers to edit, view and version Type Objects. Other traditional tools break down because they are not effective anymore. Debuggers and inspectors, for instance, still work, but they are harder to use: type objects appear as any other field in an inspector, and to retrieve the value of a property, you must navigate through the implementation structures. You need to provide new tools that replace or enhance the existing tools. This need has been captured by many, for example as the Visual Builder pattern of Roberts and Johnson [Roberts+1998]. An important concern is where to store the models and how to version and maintain the models in a multi-user environment. You can store for instance models in a database or as XML files in a source control system.

The inverse problem exists if we want to directly use a general purpose programming language and make it available to domain-experts, as, in general, we can not afford to present end-users with the same tools as regular developers.

Dynamic behavior

The core of the Dynamic Object Model pattern provides only a structure to which dynamic behavior needs to be hooked up to. However, there is no standardized way to do so (like a programming language for a traditional system). Hence you have to add a whole bunch of further patterns (like Strategy, Chain of Responsibility, Interpreter, or Observer) to do so [Johnson+1998]. The Dynamic Object Model is easiest to use when there is little type-specific behavior.

Runtime typing

The Dynamic Object Model pattern introduces full-fledged typing information at runtime, including property types. This may be helpful in dynamically typed languages like Smalltalk, for instance if we want to map objects onto a relational database.

Portability

The ‘language’ described by the Dynamic Object Model pattern is essentially independent of the implementation language. Putting more information in configuration files or repositories and less in classes in the implementation language means that the system can be ported by rewriting the implementation and reusing the configuration. In a sense this is no different from write-once-run-anywhere programs written in Java or VisualWorks Smalltalk running on top of a virtual object machine (except that these virtual machines are readily available for the major platforms).

Extensions

You can extend the Dynamic Object Model pattern by adding structural features and by adding behavioral features. On the structural side, you enhance the core pattern by adding relationships like inheritance, aggregation, associations, and role-playing.

The two most common structural relationships are inheritance and aggregation:

- *Inheritance.* Give every Type Object a link to its supertype, thereby building up a single inheritance tree. This gives you the power of type reuse. With it come the problems of preserving inheritance semantics. Can inherited properties be overwritten? Is an instance of a derived type just one instance or several? We can learn from traditional programming languages, but they also provide a variety of answers to these problems, and no general solution.
- *Aggregation.* Give every Type Object a list of aggregate member types for its instances. An instance object must conform to this structure set up on the type level. A Dynamic Object Model diagram enhanced with the aggregation relationship is not truly commutative: on the instance level, an object owns its aggregated subobjects, while on the type level, a Type Object references, but does not own, the aggregated Type Objects.
- *Constraints.* Some domain models have a lot of constraints between their various Instance objects. These constraints make it difficult to use the Dynamic Object Model pattern, if they need to be addressed. To overcome this problem, you can introduce Constraint objects that describe how attribute values of Instance objects

and links between Instance objects can and cannot relate. Effectively, you are adding a constraint solving system to the domain model.

If you want to provide more than one relationship type between Type Objects, consider introducing explicit *relationship type objects* that describe how a type relates to another type. Such a relationship type object can provide adornments like names, multiplicity, and visibility.

You can also apply the Dynamic Object Model pattern recursively:

- *Recursive application.* Make ComponentType a subclass of Component to reuse its features. This lets you introduce further type levels. In Figure 3, there is only one instance and one type level. By making the Component/ComponentType relationship recursive, you can introduce new types of Type Objects, which lets you handle not just one domain-specific language, but several.

To deal with behavioral aspects, you can include several patterns, including:

- *Strategy*, to hook up individual aspects of behavior to instance objects,
- *Chain of Responsibility*, to connect objects with each other to delegate functionality,
- *Interpreter*, to make a whole instance object hierarchy compute some algorithm, and
- *Observer*, to implement ‘rules’ that check or implement consistency or that automate computations whenever particular events (such as state changes) occur at the instance *or* at the type level.

Johnson and Oakes discuss this topic in more detail [Johnson+1998].

Implementation

A simple implementation of the Dynamic Object Model pattern is straightforward, as is shown in the Sample Code section. However, such an implementation may be too inefficient. Let us therefore examine some possible performance bottlenecks.

The core Dynamic Object Model pattern provides property access. Anything else is left to the designer, such as how to invoke behavior or how to map the classes of the Dynamic Object Model pattern to a (typically) relational database. However, even these problems can be handled by observing a key guideline: *use the type information of the system wherever possible to make ever stronger assumptions about runtime execution.*

Let us examine how we can use type information to speed up property access. There are two crucial issues that are poorly handled by a naive implementation, but that can be handled nicely by using type information.

- *Type-checking property access.* In a straightforward implementation, properties are accessed using strings as their name. To check the name for validity, the string is used in at least one dynamic hashtable lookup, which typically leads to a PropertyType object. Because it occurs with every property access, this lookup is costly.

We can overcome this problem by requiring clients to use unambiguous keys to identify a property. Such a key should be immutable (a value object) and handed out by the component itself. This way, the component can make sure that the key

will always be a valid key, so that type checking property access can be omitted.

One option for such a key is the `PropertyType` object of a property itself. Client code that is written in terms of `PropertyType` objects received from a component (rather than strings) is guaranteed to ask the Component only type-safe questions. In general, however, it is better to introduce dedicated key objects.

- *Accessing the property.* Given a valid name or key for a property, a straightforward implementation requires another dynamic lookup: from the key to the actual property, typically stored in a hashtable. Here, the costly part is the calculation of the hash code and the lookup in the table.

However, because component types don't change frequently, we can separate two different phases in the lifetime of a component type and its instances. Most of the time, the type definition is stable, and nothing changes. This regular operating phase is occasionally interrupted by short phases, in which we change the type.

Most property accesses take place during regular operation with a stable `ComponentType` definition. During this time, we assign each `PropertyType` (and hence each key) a unique index into an array. We then replace the properties hashtable with an array and store component properties in that array exactly at those indices provided by their keys. This reduces the dynamic hashtable lookup to a simple indexed array lookup.

The second performance improvement highlights one drawback of our ever-smarter implementations: the need to perform more bookkeeping. It is interesting (and not surprising) to note that the techniques described here are similar to what happens in a virtual machine that allows for runtime modification of classes with existing instances, see [Riehle+2001]. In fact, we can borrow several ideas from interpreter and VM implementations, as well as from other domains. Which techniques work best for you ultimately depends on your application requirements.

Sample Code

We describe how we use the pattern to model accounts as illustrated in the Motivation section by means of the following Java code snippets. We do not present the 'final' solution at once. Instead, we retrace some of the steps in the Motivation section. We do not show all member fields, methods or classes.

Let us begin with a class `SavingsAccount` that captures what makes up a savings account:

```
public class SavingsAccount {
    protected Money balance;
    protected Percentage interestRate;

    public Money getBalance() {
        return balance;
    }

    public Percentage getInterestRate() {
        return interestRate;
    }
}
```

```

public synchronized void deposit(Money deposit) {
    balance = balance.add(deposit);
}

public synchronized void withdraw(Money amount) {
    Money newBalance = balance.subtract(amount);
    if (newBalance.getAmount() >= 0) {
        balance = newBalance;
    }
}

public void accrueDailyInterest() {
    deposit(InterestCalculator.calcDailyInterest(getBalance (),
getInterestRate()));
}
}

```

Each instance of the SavingsAccount class contains a field called ‘interestRate’ that stores the interest rate for the account. The SavingsAccount class provides a number of fields that are value objects, like Money and Percentage⁴. For simplicity’s sake, let’s assume that each day the method ‘accrueDailyInterest’ is called and the interest is added to the account’s balance.

Obviously, it is not very efficient to make every Account object store the interest rate. We could make it a static field of the SavingsAccount class, but this makes changing the interest rate rather difficult, in particular if it involves database persistence. It is better to provide a SavingsAccount type object class that provides the field ‘interestRate’ for all the different variations of SavingsAccount that our anonymous bank provides to its customers:

```

public class SavingsAccountType {
    protected Percentage interestRate;

    public Percentage getInterestRate() {
        return interestRate;
    }

    public synchronized void setInterestRate(Percentage ir) {
        interestRate = ir;
    }
}

```

The SavingsAccount class can now retrieve the interest rate from its type object and use it to calculate the daily interest payments.

```

public class SavingsAccount {
    protected Money balance;
    protected SavingsAccountType type;

    public Money getBalance() {
        return balance;
    }
}

```

⁴ For an efficient implementation of such value objects, please see www.jvalue.org.

```

    }

    public SavingsAccountType getType() {
        return type;
    }

    public void accrueDailyInterest() {
        Percentage interestRate= getType().getInterestRate();
        deposit(InterestCalculator.calcDailyInterest(getBalance (),
getInterestRate()));
    }
}

```

We can now change the interest rate for all accounts of a specific SavingsAccount type by changing the field in the SavingsAccountType object.

Next to our SavingsAccount class, we are also designing and implementing a CheckingAccount class. Because SavingsAccount and CheckingAccount have so many fields in common, we introduce a superclass Account that captures fields like owner id, balance, and most importantly, the reference to the type object. For the type object, we introduce a class AccountType, which provides fields shared by all types of accounts:

```

public class Account {
    protected PartyId ownerId;
    protected Money balance;
    protected AccountType type;

    public AccountType getType () {
        return type;
    }
}

public class AccountType {
    protected String name;

    public String getName() {
        return name;
    }
}

```

However, not all fields are common to all classes. For example, a savings account typically has no overdraft limit, but a checking account has. We could now use inheritance to add the different fields for different subclasses of Account and AccountType. However, as initially noted, the class hierarchy can quickly become so deep that handling and changing it becomes unwieldy. Our bank, successful in its business, not only has individual retail customers, but also wealthy individual (private banking) customers, corporate clients, pension funds, and others, all of which come with special requirements that need to be catered for.

Quickly losing the oversight of the resulting class hierarchy, we decide to use a property list to hold the fields of an account. We drop the SavingsAccount class and extend the generic Account class with a property list (actually a hashtable of generic property objects). The property list maintains all fields for savings accounts or for checking accounts, and others that are not stored in a dedicated member field.

In the following code we provide a uniform way to access properties, regardless of their representation as regular fields or as entries in the property list.

```
class Account {
    protected PartyId ownerId;
    protected Money balance;
    protected Map properties;
    protected AccountType type;

    public AccountType getType () {
        return type;
    }

    public Money getBalance() {
        return balance;
    }

    public Object getProperty(String name) {
        if (isFieldPropertyName(name)) {
            return getFieldProperty(name);
        }
        else {
            return getDynamicProperty(name);
        }
    }

    protected Object getFieldProperty(String name) {
        if (name.equals("balance")) {
            return getBalance();
        }
        else {
            return null;
        }
    }

    protected Object getDynamicProperty (String name) {
        return properties.get(name);
    }

    public synchronized void setProperty(String name, Object value) {
        if (isFieldPropertyName(name)) {
            setFieldProperty (name, value);
        }
        else {
            setDynamicProperty(name, value);
        }
    }

    protected void setFieldProperty(String name, Object value) {
        // no code for balance, but maybe for additional fields.
    }

    protected void setDynamicProperty(String name, Object value) {
        properties.put(name, value);
    }
}
```

There may be valid reasons to retain some fields considered common to all types of accounts as individual fields rather than dynamic property types. This usually allows, for example, for more efficient database querying (assuming that the property list will be stored as a non-queryable BLOB). Also, for some types of fields, it may not be acceptable to directly set them, so that access needs to be controlled. (For example, the balance is either added to or subtracted from, but it is never directly set a value.)

As we can see from the implementation of Account, it is possible to set arbitrary properties to an instance of it. This is certainly not desirable, because some accounts may not even know properties that are falsely set to them! Hence, we extend our AccountType implementation and provide means to describe what properties are valid for a specific type of account.

First of all, we must capture the types of properties that an Account instance may receive. Hence, we conceive a PropertyType class that describes one particular type of property available for instances of a given type of account:

```
class PropertyType {
    protected String name;
    protected Class type;
    protected boolean isMandatory;

    public String getName() {
        return name;
    }

    public boolean isSupertypeOf(Class type) {
        return type.isAssignableFrom(this.type);
    }

    public boolean isValidValue(Object value) {
        // add correct test, possibly delegate to a strategy
        return true;
    }
}
```

Now we make the AccountType class provide a set of PropertyType objects, each of which represents a property its instances may or must have. Again, using a hashtable to store the property type objects is a reasonable choice.

```
class AccountType {
    protected String name;
    protected Map propertyTypes;

    public boolean hasPropertyType(String name) {
        return propertyTypes.containsKey(name);
    }

    public PropertyType getPropertyType(String name) {
        return (PropertyType) propertyTypes.get(name);
    }

    public boolean isValidProperty(String name, Class type, Object
value) {
        PropertyType propertyType = getPropertyType(name);
```

```
        if (propertyType == null) {
            return false;
        }
        return propertyType.isSupertypeOf(type) &&
propertyType.isValidValue(value);
    }
}
```

We now have shown how to model and implement the motivating example using the Dynamic Object Model pattern.

Known Uses

Most object-oriented programming languages work according to this model. However, the relationships we describe are subdued to efficient implementations and hence are non-obvious. An important question is whether the programming language supports modification at runtime. Another key feature is the ability to adapt the meta-model towards a specific business domain. Clearly, languages like Java with their limited introspective reflective facilities currently fall short, which means we must turn towards languages like Smalltalk or CLOS. Even the current aspect-oriented extensions of Java lack various benefits of the Dynamic Object Model pattern. Another useful perspective is to extend our comparison to prototype-based languages, as the Dynamic Object Model pattern contains traces of both class-based and prototype-based concepts.

Fortunately, there are also many systems that make the model explicit. And as the Sample Code section demonstrates, we can fairly easily implement such systems in most modern languages.

At Argo we developed a framework to support Argo's administration when the organization itself was in a great state of flux. The framework provides generic components and tools (such as query screens, overview lists and authorization rulebase) driven at runtime by the business model. We use the Dynamic Object Model pattern with several extensions (see Extensions section) to implement the business model (this includes organizational model, data, documents, relationships and business rules) [Tilman+1999]. A lot of effort went into making sure we get good performance, while retaining the flexibility of Dynamic Object Models [Tilman1999].

A second application of Dynamic Object Models consists in a business rules engine to allow the Belgian's Social Security to adapt the business rules to legislative changes [Tilman2005].

At UBS, we developed Dynamo 1 and Dynamo 2. Dynamo 1 was a research prototype used to sell the idea to bankers [Riehle+1998]. It featured a full-fledged type level with all relevant relationship types. After successfully getting contracts, we implemented Dynamo 2 in the corporate client business [Wegener1999]. Dynamo 2 uses a slimmed down version of the Dynamo 1 model that is similar to the core of the pattern as described here. It was used to capture the plethora of types of loans available to corporate clients of UBS.

At SKYVA, we developed a UML virtual machine that loads UML models and runs them [Riehle+2001]. The virtual machine was part of a larger development effort that comprised an integrated development environment (IDE) for UML, several domain-specific language extensions of UML, as well as runtime support for dynamic object models on both the type and instance level in the form of libraries. UML and its extensions are cast as dynamic object models, and the virtual machine is their implementation. The implementation made use of many well-known interpreter and virtual machine techniques like those discussed above, for example using

keys rather than strings, perfect hashtables, etc. Of particular interest is the focus on domain-specific languages and tooling to support working in these domains.

The Hartford has a framework for representing insurance policies that is a Dynamic Object Model [Johnson+1998].

The thesis by Witthawaskul describes a simple Dynamic Object Model [Witthawaskul2001] for editing databases.

Related Patterns

As a composite pattern, the Dynamic Object Model pattern imposes constraints on its composed patterns:

- compared to the Type Object pattern, types now explicitly describe the structure of their instances;
- compared to the Property List pattern, properties are now constrained by their types;
- when extending the pattern with Composite, the components of an aggregate are constrained by their type.

The Object System pattern [Noble2000] provides components with properties, but does not introduce a type level distinct from the instance level. Effectively, it is a type-less generic object model.

Martin Fowler's book Analysis Patterns introduces the concepts of knowledge level and operational level (for accountability) [Fowler1997]. Essentially, knowledge level means the same as type level, and operational level means the same as instance level. Fowler describes various aspects of the knowledge level for accountability, which can be viewed as a domain-specific use of the Dynamic Object Model pattern.

Robert Haugen's Dependent Demand pattern refers to Fowler's knowledge level [Haugen1997]. The knowledge level underlies all operations of the demand networks (order networks) that he describes for supply chain management. Through this indirection, Dependent Demand relies on the Dynamic Object Model pattern, and most instances of the Dependent Demand pattern are likely to use the Dynamic Object Model pattern.

Acknowledgements

This chapter was initially shepherded for and workshopped at PLoP 2000. We would like to thank Joshua Kerievsky, our shepherd, and Robert Haugen for helping us improve the description of the Dynamic Object Model pattern. We would further like to thank the workshop participants and the anonymous reviewers of the PLoPD 5 submission for helping us improve the pattern description as well.