Dynamic Object Model

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de

dirk@riehle.org, www.riehle.org

Michel Tilman

UNISYS Belgium
www.unisys.com

mtilman@acm.org, http://users.pandora.be/michel.tilman

Ralph Johnson

Computer Science Department
University of Illinois at Urbana-Champaign

johnson@cs.uiuc.edu, http://st-www.cs.uiuc.edu/users/johnson

1 Intent and Summary

A system with a Dynamic Object Model allows the types of objects to change at runtime. This includes adding new types, changing existing ones, and changing the relationships between types. Taken together, all types and their relationships form a domain-specific model. Underlying such a dynamic object model is a framework that acts much like a domain-specific modeling language.

Dynamic Object Model is a compound pattern[1] that at its core composes the Type Object, Property List, and Value Holder patterns.

2 Also Known As

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

3 Motivation

Imagine you are developing a banking system for handling customer accounts like checking or savings accounts. You first think about a nice class hierarchy of account classes, starting with a root class Account. However, your domain experience tells you that banks provide many different types of accounts. It is not uncommon for a large

[1] A compound pattern is a pattern that is best described as a recurring composition of other patterns [Riehle+1997a, Vlissides+1998].

bank to provide more than 500 types of accounts as products to their clients. Many of these accounts vary only by a few parameters, but they are still distinct enough to require modeling these distinctions.

You quickly give up modeling 500 Account classes. Remembering the Type Object pattern [Johnson+1998] you decide to 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. 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 class AccountType (name of this type of account, interest rate for this type of account, etc.). All properties that may vary within instances of the same AccountType go into the Account class (account number, current balance, etc.).

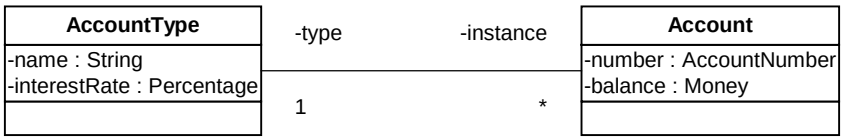

Figure 1: Account and AccountType.

After a short time, however, your Account and AccountType classes get bloated with fields and methods that represent and access the different properties all types of Accounts may have. After all, the class interfaces represent the union of some 500 account types! Remembering the Property List and Value Holder patterns [Riehle1997a, Foote+1998], you decide to model the properties that an Account instance may have using a list of property objects. Properties of Account like name of owner or balance now become instances of a generic Property class. The instances of the Property class become value holders for any type of object.

However, the problem is not going away. You still have to check every access to an Account property for validity. After all, you do not want that a programming mistake let's someone set the "balance" property a value like "John Doe". Effectively, you need to check access to a property. This brings us back to the Type Object pattern. You decide to introduce a PropertyType class that can check whether a given value for a Property is valid. Therefore, you link every Property to a PropertyType object that carries out these checks.

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 set an owner name. Hence it does not know a property "owner name" and must not be set 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.

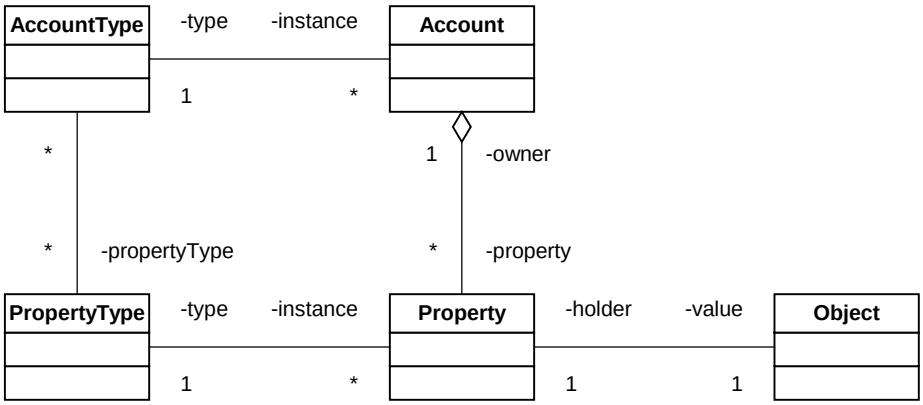

Figure 2: AccountType/PropertyType and Account/Property.

What can be said about the design? 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 properties. 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). There is more to say, which we will do in the rest of the description of this pattern for runtime metamodels.

4 Structure

The core of the Dynamic Object Model pattern consists of the classes Component/ComponentType, Property/PropertyType and their respective clients. The pattern is illustrated in Figure 3. It composes the Type Object, Property List, and Value Holder patterns. Figure 3 uses UML collaboration specification diagrams to show the participating patterns. Each collaboration specification is displayed as a dashed ellipsis.

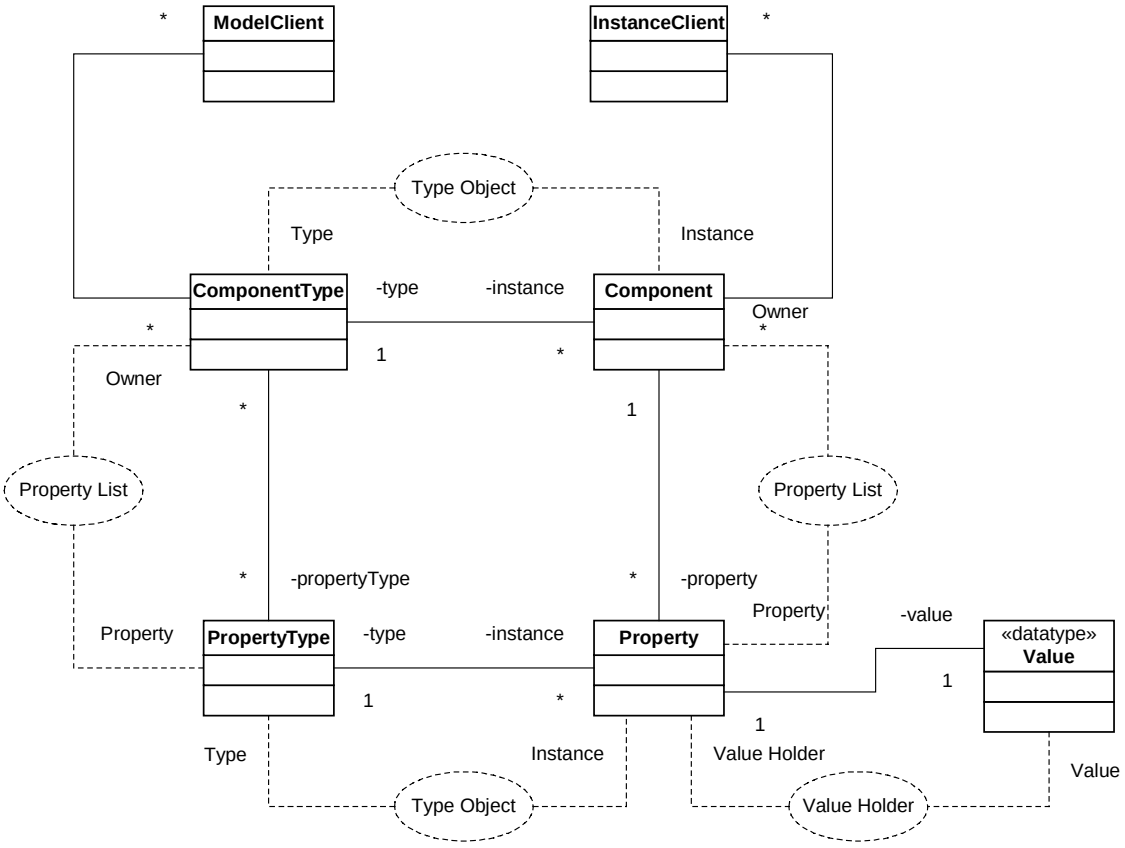

Figure 3: Class diagram of the Dynamic Object Model pattern.

In Figure 3, the Type Object classes form the type level, and the Instance classes form the instance level. Figure 3 uses a shorthand form for displaying UML collaboration specification diagrams. This shorthand consists of the ellipsis naming the pattern and links to classes that take on ClassifierRoles as named on the link from the ellipsis to the class figure.

We now focus on each of the pattern descriptions, described using a full-blown UML collaboration specification diagram. Such a collaboration specification consists of ClassifierRoles and their associations. A ClassifierRole

describes a role an instance of a class may play at runtime. Each class (or UML Classifier) may provide several ClassifierRoles according to the number and type of roles its instances may play.

A ClassifierRole in a collaboration specification below is displayed as a box, much like a regular class. However, the label of a ClassifierRole box in a UML collaboration specification diagram starts with a "/", followed by the name of a ClassifierRole, followed by ":", followed by the class that provides this ClassifierRole. A ClassifierRole describes a role that instances of a class may play as part of an object collaboration. If the class name after the ":" is omitted, the ClassifierRole is not tied to a specific class.

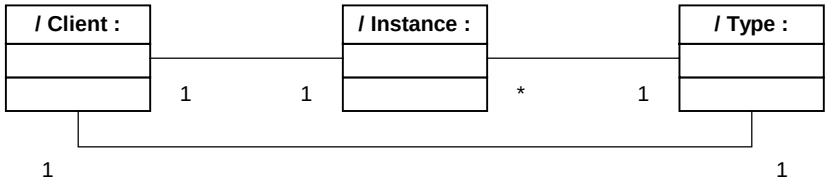

Figure 4: The Type Object pattern illustrated using a collaboration specification diagram.

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.

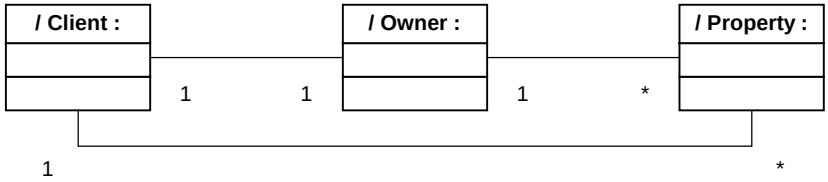

Figure 5: The Property List pattern illustrated using a collaboration specification diagram.

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 get and set dynamically, typically using strings as property names. The Property List pattern allows for runtime extension of the Owner's properties.

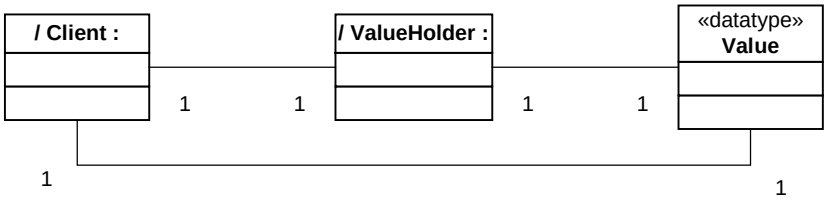

Figure 6: The Value Holder pattern illustrated using a collaboration specification diagram.

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].

5 Classes and Roles

The Component class (Account in the motivation section) provides the following ClassifierRoles:

- *Owner* (of /Property:Property instances)
- *Client* (of /ValueHolder:Property instances)
- *Instance* (of a /Type:ComponentType instance)
- *Client* (of a /Type:ComponentType instance)

The Property class (Attribute in the motivation section) provides the following ClassifierRoles:

- *Property* (of /Owner:Component instances)
- *Instance* (of a /Type:PropertyType instance)
- *Value Holder* (of /Value:Value instances)

The Value class (Value in the motivation section) provides the following ClassifierRoles:

- *Value* (of a /ValueHolder:Property instance)

The ComponentType class (AccountType in the motivation section) provides the following ClassifierRoles:

- *Client* (of /Owner:Component instances)
- *Type* (of /Instance:Component instances)
- *Owner* (of /Property:PropertyType instances)
- *Client* (of /Property:PropertyType instances)

The PropertyType class (AttributeType in the motivation section) provides the following ClassifierRoles:

- *Property* (of /Owner:ComponentType instances)
- *Type* (of /Instance:PropertyType instances)

6 Applicability

Use the Dynamic Object Model pattern if

- you have many types of objects that differ only in a few fields and you want to reduce the number of classes;
- you want to create new types of objects with different properties at runtime;
- you want to build applications that let end-users configure many types of objects;
- your applications require frequent changes and fast evolution;
- you need an explicit model of the structure of your objects and you want to access this model at runtime;
- you want to provide automatic type validation (in dynamically typed languages);
- you want a domain-specific modeling language.

6.1 Advantages

- *Explicit model*. The Dynamic Object Model pattern provides you with a (typically high-level) model that can be consulted at runtime. For instance, when developing a generic store-retrieve relational database application you can generate forms, query screens and SQL statements on the fly using the available meta-information. 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.

- *End-user configuration.* The Dynamic Object Model pattern lets end-users define the key concepts from their application domain at runtime, without lengthy development cycles in between. Effectively, the Dynamic Object Model pattern provides a (possibly domain-specific) language that users use to describe their domain.
- *Introduces runtime typing.* The Dynamic Object Model pattern introduces full-fledged typing information at runtime. This is particularly helpful in dynamically typed languages like Smalltalk.
- *Runtime object type creation.* Using the Dynamic Object Model pattern, new and complex types of objects can be created at runtime. They describe the structure of their instances: their properties and the constraints on the values their properties can assume.
- *Domain-specific typing.* The Dynamic Object Model pattern introduces typed properties. This feature has a lot of useful applications, for instance when generating forms for data entry and when validating input. The behavior and semantics of these functions are typically domain-specific.
- *Controlled dynamic type change.* The Type Object pattern allows instance objects to dynamically change their type. The Dynamic Object Model pattern extends this behavior by controlling in a generic way the type changing process and when the changes may take place.
- *Runtime component type modification.* The Dynamic Object Model pattern allows for runtime modification of types and hence supports flexibility and fast evolution of applications. However, dynamic type modification requires us to resolve several questions. For instance do we allow for live update of existing objects?
- *Language independent.* The 'language' described by the Dynamic Object Model pattern is essentially independent from the implementation language. Thus porting of applications described in terms of this domain-specific language usually involves less rewriting when porting to another environment.
- *Fewer classes*. The Dynamic Object Model pattern reduces the number of 'real' classes. This reduction easily amounts to several orders of magnitude.

6.2 Disadvantages

- *Increased design complexity*. One logical class is now represented as a set of instances from a larger set of classes. A class is represented by 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.
- *Increased runtime complexity*. One logical object now consists of several objects, together with their relationships. An instance of a domain class is now represented by instances of ComponentType, PropertyType, Component, Property and values/value objects. The runtime state of a system becomes harder to understand than the state of a traditional system.
- *New development tools.* Programmers can no longer rely on their familiar development tools, such as browsers to edit and view types of components. Other traditional tools break down because they are not effective anymore. Debuggers and inspectors, for instance, still work, but they are much harder to use: type objects appear as any other field in an inspector, whereas we should be able to view components as instances of their component type. 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].
- *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. See [Johnson+1998] for further discussion.
- *Performance and space usage penalties.* A straightforward implementation of the Dynamic Object Model pattern typically leads to performance penalties and increased memory usage. How much is acceptable depends on the application domain. However, in contrast to general-purpose object-oriented languages, you can more readily customize the implementation towards a particular usage.

7 Extensions

The Dynamic Object Model pattern can be extended in two different ways: by adding structural features and by adding behavioral features. On the structural side, the basic metamodel can be extended by adding new relationship types like inheritance, aggregation, associations, role-playing, and others.

The two most common relationships are inheritance and aggregation:

- *Inheritance.* Every type object gets 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 different answers to these problems, and no general solution.
- *Aggregation.* Every type object holds a list of aggregate member types for its instances. An instance object must conform to this structure as set up on the type level. Here, type level and instance level are not commutative. On the instance level, an object owns its aggregated subobjects, while on the type level, a type object only references the types of aggregated objects.

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

Another extension is to apply the Dynamic Object Model pattern recursively:

- *Recursive application.* You can 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.

On the dynamic side of things, you can use

- Strategy, to hookup 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.

Please see Ralph Johnson’s and Jeff Oakes’s article on this topic [Johnson+1998].

8 Implementation

A simple implementation of the Dynamic Object Model pattern is straightforward, as is shown in the Sample Code section. However, such an implementation is typically inefficient, possibly rendering the system useless due to unacceptable performance. Let’s therefore examine the possible performance bottlenecks.

In the context of this pattern, it all boils down to property access. Beyond this pattern, there are further issues, for example 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: *to 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 hashCode and the lookup in the table.

  However, because component types don't change every other second, 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 takes place during regular operation with a stable ComponentType definition. During this time, we assign each PropertyType and hence any 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 suprising) to note that the technique described here is similar to what happens in a virtual machine that allows for runtime modificaton of classes with existing instances. In fact, we can borrow several ideas from interpreter and VM implementations, as well as other domains. Which techniques work best for you ultimately depends on your application requirements.

9 Sample Code

We describe how we use the pattern to model accounts as illustrated in the Motivation section. The following code examples are provided in Java.

Let's 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 synchronized Money deposit(Money deposit) {
        balance+= deposit;
    }

    public synchronized Money withdraw(Money amount) {
        Money newBalance = balance - amount;
        if (newBalance >= 0) {
            balance = newBalance;
        }
    }

    public void accrueDailyInterest() {
        Money interest = InterestCalculator.calcDailyInterest((), getInterestRate());
        deposit(interest);
    }

    ...
}
```

Each instance of the SavingsAccount class provides a field called "interestRate" that provides the interest rate for the account. The SavingsAccount class provides a number of fields that are value objects, like "Money" and

"Percentage"[2]. 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 SavingsAccountType type;
    ...

    public SavingsAccountType getType() {
        return type;
    }

    public void accrueDailyInterest() {
        Percentage interestRate= getType().getInterestRate();
        Money interest = InterestRateCalculator.calcDailyInterest(
            getBalance(), interestRate
        );
        deposit(interest);
    }
    ...
}
```

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 String getTypeName() {
        return type.getName();
    }

    ...
}

public class AccountType {
    protected String name;
    ...
```

[2] For an efficient implementation of such value objects, please see www.jvalue.org.

```
    public String getName() {
        return name;
    }

    ...
}
```

However, not all fields are common to all classes. For example, a savings account typically has no overdraw 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, checking accounts, and others that are not stored in a dedicated 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 Hashtable properties;
    ...

    public Money getBalance() {
        return balance;
    }

    public Object getProperty(String name) {
        Object result= getFieldProperty(name);
        if (result == null) {
            result= getListProperty(name);
        }
        return result;
    }

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

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

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

    protected void setFieldProperty(String name, Object value) {
        // no code for balance, but for other properties.
        ...
    }

    protected void setListProperty(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. 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 need to capture the types of properties that an Account instance may receive. Hence, we conceive a PropertyType class that describes one particular type of property as 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(type);
    }

    public boolean isValidValue(Object value) {
        // check value, possibly delegate to a strategy
    }

    ...
}
```

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 Hashtable propertyTypes;
    ...

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

    public boolean isValidProperty(String name, Class type, Object value) {
        if (!hasPropertyType(name)) {
            return false;
        }
        PropertyType pt = getPropertyType(name);
        return pt.isSupertypeOf(type) && pt.isValidValue(value);
    }

    ...
}
```

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

10 Known Uses

Most object-oriented programming languages work according to this model. However, the relationships we describe are subdued to efficient implementations and hence non-obvious. Fortunately, there are many systems that make this model explicit.

At Argo we developed a framework to support its 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].

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.

EbXML, an emerging standard for e-business model and model instance exchange is an object-oriented design that features the Dynamic Object Model pattern. The design is based in part on an object-oriented adaptation of REA, a business modeling ontology originally from accounting, to e-business and data exchange [Haugen+2000, EbXML2000].

11 Related Patterns

By combining several patterns, the Dynamic Object Model pattern imposes more constraints on its "primitive" 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

We would like to thank Joshua Kerievsky, our shepherd, and Robert Haugen for helping us improve the description of the Dynamic Object Model pattern.

References

Bäumer+1998 Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, and Heinz Züllighoven. *Values in Object Systems.* Ubilab Technical Report 98.10.1. Zurich, Switzerland: UBS AG, 1998. An open-source implementation is available from www.jvalue.org.

EbXML2000 OASIS. See www.ebxml.org. OASIS, 2000.

Cunningham1995 Ward Cunningham. “The Checks Pattern Language of Information Integrity.” In *Pattern Languages of Program Design.* Addison-Wesley, 1995. Page 147-162.

Foote+1998 Brian Foote and Joe Yoder. “Metadata.” In *Technical Report #WUCS-98-25* (PLoP ’98). Dept. of Computer Science, Washington University: 1998.

Fowler1997 Martin Fowler. *Analysis Patterns.* Addison-Wesley, 1997.

Haugen1997 Robert Haugen. “Dependent Demand.” In *Technical Report #WUCS-97-34* (PLoP ’97). Dept. of Computer Science, Washington University. 1997.

Haugen+2000 Robert Haugen and William E. McCarthy. “REA, a Semantic Model for Internet Supply Chain Collaboration.” Available from http://www.supplychainlinks.com/Rea4scm.htm.

Johnson+1998 Ralph Johnson and Bobby Woolf. “Type Object.” In *Pattern Languages of Program Design 3.* Addison-Wesley, 1998. Page 47-66.

Johnson+1998 Ralph Johnson and Jeff Oakes. “The User-Defined Product Framework.” Unpublished manuscript, available from http://st-www.cs.uiuc.edu/users/johnson/papers/udp.

Noble2000 James Noble. “Prototype-Based Object System.” In *Pattern Languages of Program Design 4.* Addison-Wesley, 2000.

Riehle1997a Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97-1-1. Zürich, Switzerland: Union Bank of Switzerland, 1997. Available from www.riehle.org/papers.

Riehle1997b Dirk Riehle. “Composite Design Patterns.” In Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97). ACM Press, 1997. Page 218-228.

Riehle+1998 Dirk Riehle and Erica Dubach. “Why a Bank Needs Dynamic Object Models.” Position Paper for *OOPSLA ’98 Workshop 15 on Metadata and Active Object Models.* UBS AG, 1998. Available from www.riehle.org/papers.

Roberts+1998 Don Roberts and Ralph Johnson. “Patterns for Evolving Frameworks.” In *Pattern Languages of Program Design 3.* Addison-Wesley, 1998. Page 471-486.

Tilman+1999 Michel Tilman and Martine Devos, A Reflective and Repository-Based Framework, p.29-64, Implementing Application Frameworks (M.E. Fayad, D. C. Schmidt, R. E; Johnson ed.), Wiley Computer Publishing

Tilman1999 Michel Tilman, “Active Object-Models and Object Representations”, Position Paper for the OOPSLA ’99 *MetaData and Active Object-Model Pattern Mining Workshop*. Available from http://users.pandora.be/michel.tilman/Publications.

Vlissides1998 John Vlissides. “Composite Design Patterns--They Aren’t What You Think.” *C++ Report*, June 1998.

Wegener1999 Hans Wegener. “Dynamo 2 System Documentation.” UBS AG, 1999.