Dealing with Properties

Martin Fowler
fowler@acm.org

Almost every object you create needs properties: some statement about the object. A person may have a height, a company may have a CEO, a flight may have a flight number. There are a number of ways to model properties. In this article I will explore some of these ways and when you may want to use them. I've often seen patterns touch on this subject, but they usually only cover part of the picture. Here I want to cover the issue more broadly to give a better discussion of the options.

The most common, and the simplest case is to use a *Fixed Property*, that is just declare the attribute on the class. In the vast majority of cases that is all you need to do. Fixed properties begin to fail when you have a large amount of them, or you need to change them frequently, possibly at run time. These forces lead you to the varieties of *Dynamic Property*. All dynamic properties have the quality of a parameterized attribute where to query a property you need to use a query method with a parameter. The simplest of these is the *Flexible Dynamic Property* where the parameter is just a string. This makes it easy to define and use properties, but is difficult to control. If you need that control you can use a *Defined Dynamic Property* where your parameters must be instances of some class. A further step of control allows you to strongly type your dynamic property with a *Typed Dynamic Property*.

As your property structures get more involved, you should consider making them *Separate Properties*, which makes the property an object in its own right. If you need multi-valued properties you can consider a *Typed Relationship*. The most complex example in this theme is where you want rules to control what kinds of objects have what kinds of properties – this needs a *Dynamic Property Knowledge Level.*

Another variety of property entirely is the *Extrinsic Property*, a pattern you use if you want to give an object a property, but not alter its interface to support it.

Problem	Solution	Name	page
	Give it a specific attribute for that fact. This will translate to a query method and probably an update method in a programming language.	**Fixed Property**	4

	Provide a parameterizable attribute which can represent different properties depending on the parameter	**Dynamic Property**	4
	Provide an attribute parameterized with a string. To declare a property just use the string.	**Flexible Dynamic Property**	5
	Provide an attribute parameterized with an instance of some type. To declare a property create a new instance of that type	**Defined Dynamic Property**	7
	Provide an attribute parameterized with a instance of some type. To declare a property create a new instance of the type and specify the value type of the property.	**Typed Dynamic Property**	9
How do you represent a fact about an object, and allow facts to be recorded about that fact	Create a separate object for each property. Facts about that property can then be made properties of that object.	**Separate Properties**	10
How do you represent a relationship between two objects? (How do you represent multi-valued dynamic properties?)	Create a relationship object for each link between the two objects. Give the relationship object a type object to indicate the meaning of the relationship. (The type object is the name of the multi-valued property.)	**Typed Relationship**	14
How do you enforce that certain kinds of objects have certain properties when you use dynamic properties?	Create a knowledge level to contain the rules of what types of objects use which types of properties	**Dynamic Property Knowledge Level**	16
How do you give an object a property without changing its interface?	Make another object responsible for knowing about the property.	**Extrinsic Property**	18

What is a property?

This is not a silly question. When people bandy around the term 'property' they can mean many different things. To some a property is an instance variable or data member of a class. To others they are kind of thing that goes in a box in a UML diagram. So before I start this article I have to define my usage of the word.

For my purposes a property is some information about an object that you can obtain by a query method. It may be a value type (such as int in Java) or an instance of a class. You may be able to update the property, but not necessarily. You may set the property when you create the object, but again not necessarily. The property may be stored as an instance variable or data member, but it does not have to be. The class may get the value from another class, or go through some further calculation. I am, therefore, taking an interface view of properties rather than an implementation view. This is a regular habit of mine in design: for me the essence of object-orientation is that you separate interface from implementation, and make interface more important.

Fixed Properties

Fixed properties are by far the most common kind of properties that we use. A *Fixed Property* is declared in the interface of the type. It gives the name and the return type for the property. Figure 1 shows properties modeled in UML, Listing 1 shows how the query methods for these properties would show up in Java. I've picked the example to show that this discussion applies equally well to UML attributes as well as UML associations. It also applies to calculated values (age) as well as those that are reasonably stored data (date of birth).

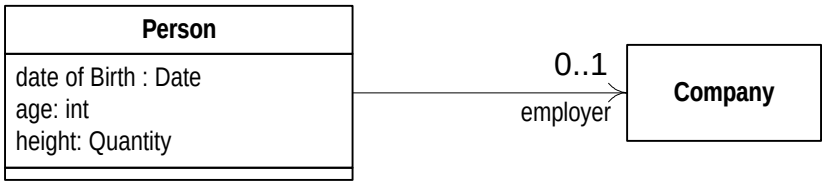

Figure 1. A Person modeled with fixed properties

```
class Person {
   public Date getDateOfBirth();
   public int getAge();
   public Quantity getHeight();
   public Company getEmployer();
   public void setDateOfBirth (Date newDateOfBirth);
   public void setEmployer (Company newEmployer);
```

Listing 1. Java operations for Figure 1

Query operations usually follow some naming convention. In smalltalk you always name the query after the name of the property (dateOfBirth). C++ never had a fixed convention, some people just named it after the property, others used the 'get' convention (getDateOfBirth). Java started with no particular convention but now most people adopt the get convention. Personally I find the 'get' irritating when you read code so I would prefer to leave it out, but the Java style is to use the get, so I use it now. You should ensure that you follow the same convention whether you are using stored or derived values. Clients of the person class should not know or care whether age is stored or derived.

The presence of modifier operations depends on whether you want the values to be directly modified. If you do then you will provide a modifier operation according to some naming scheme, eg setDateOfBirth(Date). Different conventions exist as to the return value. You can return the new value of the property (Date), the object being modified (Person), or nothing (void). I prefer to return void on modifiers, to help make clear the difference between a modifier and a query.

Fixed Property

How do you represent a fact about an object?

Give it a specific attribute for that fact. This will translate to a query method and probably an update method in a programming language.

✓ Clear and explicit interface

✗ Can only add properties at design time

You may wish to provide arguments to the constructor for properties. Typically you want to set enough properties in the constructor so that you construct a well-formed class.

Properties that you don't want to be modified directly should not have a modifier operation. This might be the case for the age property, if you only want that determined by calculation from the date of birth. It also would be the case for an immutable property: one that does not change for the lifetime of the class. When you think of making a property immutable, remember to take human error into account. While the date of birth is an immutable property for humans in the real world, you might make a mistake typing into a computer system, and thus make it mutable. Software often models what we know of the world, rather than the world itself.

Fixed properties are by far the most common form of properties that you will come across. They are that way for good reasons: they are simple and convenient to use. You should use fixed properties as your first and most common choice for representing properties. In this paper I am going to give a lot of alternatives to fixed properties. For certain situations these alternatives are better, but most of the time they are not. Remember that as we go through the alternatives. I use fixed properties 99% of the time. Other varieties are more complex, which is why I'm spending most of this paper on them — and also why I prefer not to use them!

Dynamic Properties

The key thing about fixed properties is that you fix them at design time, and all instances at run time must follow that decision. For some problems this is an awkward restriction. Imagine we are building a sophisticated contact system. There are some things that are fixed: home address, home and work phone, email. But they're all sorts of little variations. For someone you need to record their parent's address, another has a day work and evening work numbers. It's hard to predict all these things in advance, and each time you change the system you have to go through compiling, testing, and distribution. To deal with this you need to use dynamic properties.

Dynamic Property

How do you represent a fact about an object?

Provide a parameterizable attribute which can represent different properties depending on the parameter

✓ Can add properties at run time

✗ Unclear interface

There are several variations on dynamic properties, each of which make different trade-offs between flexibility and safety. The simplest approach is *Flexible Dynamic Properties*. The

essence of this pattern is that you add a qualified association to the person whose key is a simple value, usually a string (see Figure 2 and Listing 2). If you want to add a vacation address to a person Kent, you just use the code in Listing 3. You don't need to recompile the person class. You could even build a GUI or a file reader that would add properties without recompiling the client either.

Figure 2. UML model for Flexible Dynamic Properties

```
class Person {
   public Object getValueOf(String key);
   public void setValueOf(String key, Object value);
```

Listing 2. Java methods for Figure 2

```
kent.setValueOf("VacationAddress", anAddress);
Address kentVactation = (Address) kent.getValueOf("VacationAddress")
```

Listing 3. Using dynamic properties

Said like that, you might wonder why anyone would ever use fixed properties, since dynamic properties such as this give you so much more flexibility. Of course there is a cost, and it lies in the reduction of the clarity of the dependencies between parts of the software. It is all very well adding a vacation address property to a person, but how do you know to get it back again? With fixed properties you can just look at the interface of person and see the properties. The compiler can check that don't ask an object to do something it doesn't understand. With a dynamic property you cannot do any design-time checking. Furthermore the interface of person is harder to see. Not just do you look at Person's declared interface – you also have to find the dynamic properties, which will not be present in the class interface. You have to find that part of the code that sets the property (which usually will not be in the Person class at all) and dig it out.

Flexible Dynamic Property

How do you represent a fact about an object?

Provide an attribute parameterized with a string. To declare a property just use the string.

Not just is the property hard to find, it also creates a nightmare dependency. With fixed properties the client code has a dependency to the person class – a dependency that is easy to keep track of. If you change the name of the property the compiler will let you know, and tell you what code you need to change to fix things. But the *Flexible Dynamic Property* creates a dependency into some arbitrary piece of code. It could be code that belongs to a class that is not even visible to the client. What happens if someone changes the key string? What happens if someone changes the type of object they put into the key string? Not just can the compiler do nothing to help you, you don't even know where to start looking for potential changes.

Flexible Dynamic Properties show this problem at it's most extreme. The property could be created at design time by any client of Person. If another client of person uses that same

property you have a dependency between two classes that is very hard to find. Furthermore properties can be added at run time by reading a file or by a GUI. It is impossible to find out, even at run time, what are the legal dynamic properties for a person. True you can ask a person if it has a property for vacation address – but if there isn't one does that mean that that person does not have a vacation address, or does it mean that there is no such property as vacation address? And if it has no such property now, that doesn't mean it won't have one a few seconds later.

Another key disadvantage of flexible dynamic properties is that it is difficult to substitute them for operations. One of the key advantages of encapsulation is that a client that uses a property cannot tell whether it is stored as part of an object's data, or computed by a method. This is a very important part of the object approach. It allows you not just a regular interface for both purposes, but also to change your mind without the client knowing. In the presence of subtyping you can even have a supertype store the property and subclasses compute or vice versa. However if you want to use dynamic properties the only way you can change stored data to a calculation is to put a specific trap in the general accessor for the dynamic property along the lines of Listing 4. This code is likely to be brittle and awkward to maintain.

```
class Person {
   public Object getValueOf (String key) {
      if (key = "vacationAddress") return calculatedVacationAddress();
      if (key = "vacationPhone") return getVacationPhone();
      // else return stored value…
```

Listing 4. Replacing a dynamic property with an operation

Other forms of dynamic property help you solve some of these problems but not all. The essential disadvantage of a dynamic property is that you lose the clear interface and all design-time checking. Different approaches to dynamic properties give you different abilities to do run-time checking. If you need dynamic properties, and there are certainly situations when you do, then you just have to give up design time checking and a explicit design time interface. The only question is how explicit an interface and how much checking you can do at run time. With Flexible Dynamic Properties you don't get any of either.

You find dynamic properties often in databases because it is often a pain to change the database schema, especially if there would be a lot of data to migrate. Interfaces of distributed components, such as in CORBA, also often use dynamic properties for a similar reason. There are a lot of distant clients using the interface, so you are reluctant to change it. In both of these cases it is not so much a distinction between compile-time and run-time, as much as a distinction between design-time and production.

If all you are doing is displaying and updating information via a GUI, and the code never makes a fixed reference to the keys (i.e. you never see code like Listing 3), then you are pretty safe with a *Flexible Dynamic Property*. This is because you have not set up a nasty dependency to some arbitrary string as a key. Otherwise you should consider one of the other approaches to dynamic properties.

The first step towards more run-time checking is the *Defined Dynamic Property*. The key difference between a defined as opposed to a flexible dynamic property is that the key used by the dynamic property is no longer some arbitrary string, but is now an instance of some class (Figure 3).

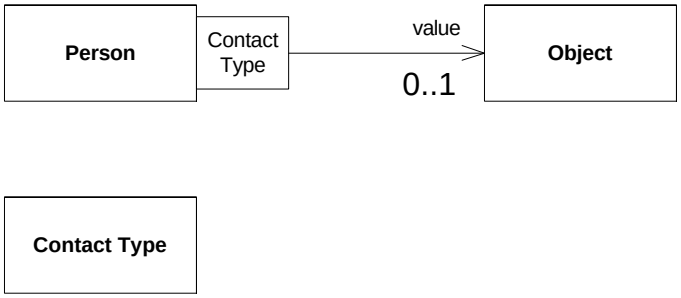

Figure 3. A Defined Dynamic Property

On the face of it using defined dynamic property does not change much. Indeed the code interface is almost identical (Listing 5 and Listing 6). But now the choice of keys is no longer entirely arbitrary, it is limited by the instances of contact type. Of course this still allows you to add properties at run time – you just create a new contact type. However now there is at least somewhere someone can look to get a list of potential keys without having to trawl through the program text. Any keys added at design time can be collected in a loader routine in the contact type class. You can easily provide services to find out the legal keys at run time.

```
class Person {
   public Object getValueOf(ContactType key);
   public void setValueOf(ContactType key, Object value);
```

Listing 5. Java interface for Figure 3

```
class ContactType {
   public static Enumeration instances();
   public static boolean hasInstanceNamed(String name);
   public static ContactType get(String name);
```

Listing 6. Services for a defined property type

In particular you can now set up some checking to prevent errors due to someone asking for a dynamic property that does not exist as in Listing 7. I'm throwing an unchecked exception here because I consider that the precondition for get() is that the client provides a legal name for a contact type. The client can always fulfill this responsibility by using hasInstanceNamed(), but most of the time client software will hang on to contact type objects, not strings.

Defined Dynamic Property

How do you represent a fact about an object?

Provide an attribute parameterized with an instance of some type. To declare a property create a new instance of that type

Usually the contact types will be held in a dictionary, usually indexed by a string. This dictionary could be a static field of contact type, but I prefer to make it a field on a *Registrar*.

Removing a contact type still has awkward consequences. In java the dynamic properties would still be present on those objects that had them, unless you write some complicated clean up code. I usually make it a rule to never delete the keys for a Defined Dynamic Property. If you want to use a new name you can easily alias the contact type by giving it one name but placing it in the defining dictionary several times.

```
class ContactType {
   public static ContactType get(String name) {
      if (! hasInstanceNamed (name)) throw new IllegalArgumentException("No
such contact type);
      // return the contact type
   }

// use with

Address kentVactation =
               (Address) kent.getValueOf(ContactType.get("VacationAddress"));
```

Listing 7. Checking use of legal contract types

At this point you might be wondering about what this has to do with conceptual modeling, after all I am writing a lot of code and discussing design trade-offs. This is an important conceptual issue because the conceptual choice you make affects the implementation options you have. If you choose to use flexible dynamic attributes in your conceptual model, you are making it pretty hard to use either defined dynamic attributes or fixed attributes in your implementation. One of the reasons you do conceptual models is to explore what is fixed and what is changeable in your users' concepts. If total flexibility was my only goal then I would always use Figure 4. With this I can model any situation in the world. But this model is not very useful. Its uselessness comes from the fact that it does not indicate what is fixed. When you are doing a conceptual model you need to be aware of how your choices affect things in implementation – otherwise you are abdicating your responsibility as a modeler.

I often find that people discover dynamic properties, and then want to use them everywhere. The flexibility is so great, they get all the extendibility they ever want. Yes sometimes you need dynamic properties. but never forget there is a price. Only use them when you really need them. After all it is easy to add them later if you have to.

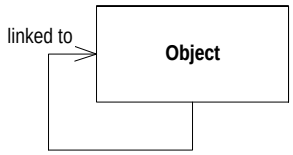

Figure 4. A model that can model any domain uselessly.

The *Defined Dynamic Property* allows you to indicate rather more about what properties you have. These properties are still untyped. You cannot enforce that the value of the vacation address in Figure 3 is an address. You can do something about that by using a Typed Dynamic Property.

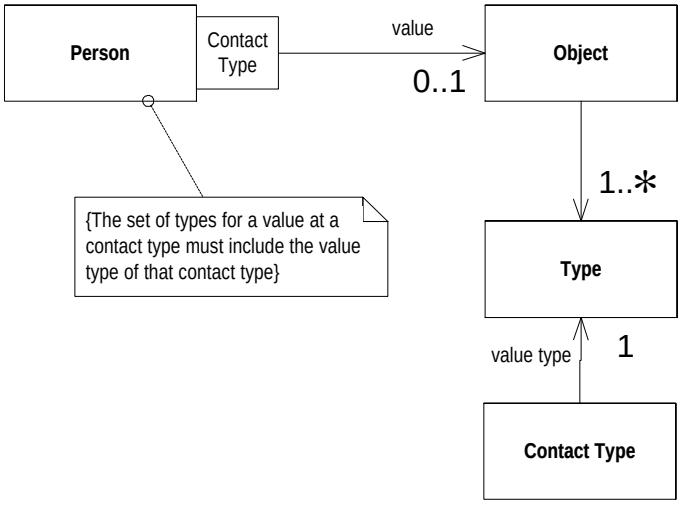

Figure 5. Model for typed dynamic property using qualified associations

A *typed dynamic property* adds type information to the defined dynamic property (Figure 6 and Figure 5). Here the instances of contact type do not just indicate what properites the person has, they also indicate the type of each property. The type constrains the value, along the lines of Listing 9.

```
class Person {
   public Object getValueOf(ContactType key);
   public void setValueOf(ContactType key, Object value);
class ContactType {
   public Class getValueType();
   public ContactType (String name, Class valueType);
```

Listing 8. Operations for typed dynamic properties

```
class Person {
   public void setValueOf(ContactType key, Object value) {
      if (! key.getValueType().isInstance(value))
         throw IllegalArgumentException ("Incorrect type for property");
      // set the value
```

Listing 9. Doing the type checking

Doing type checking like this can help to avoid errors, but it still is not as clear as fixed properties. The checking is at run time, not at design time, and is therefore not as effective. Still it is better than no checking at all, particularly if you are used to a strongly typed environment.

Typed Dynamic Property

How do you represent a fact about an object?

Provide an attribute parameterized with a instance of some type. To declare a property create a new instance of the type and specify the value type of the property.

As we delve deeper into dynamic properties, we find richer examples of *reflection*, an architectural pattern which appears when we get runtime objects that are able to describe themselves. [POSA] discusses *reflection* in much more detail than I intend to go into here.

Dynamic properties provide a reflective capability, even in those languages that don't support reflection themselves. Even with a language that does provide some reflection, the reflection of dynamic properties is more focused – so you can provide an easier to use interface.

Using all of those qualified associations can get rather hard to follow. Another way of providing *typed dynamic properties* is to use the *separate property* pattern. The essence of the separate property pattern is that it makes the property an object in its own right (Figure 6 and Listing 10). You can get hold of the properties of a person, and then with each property get value and type information.

Separate Properties

How do you represent a fact about an object, and allow facts to be recorded about that fact

Create a separate object for each property. Facts about that property can then be made properties of that object.

Separate properties and *qualified associations* are two alternatives that are always available for dynamic properties. So far I've described flexible and defined dynamic properties with qualified associations because the qualified association presents a much easier interface to use. You can use flexible and defined dynamic properties with separate properties if you wish, although I'm not going to go into that here. As we get to the complexity of typed dynamic properties and onwards the separate property style becomes more of an advantage.

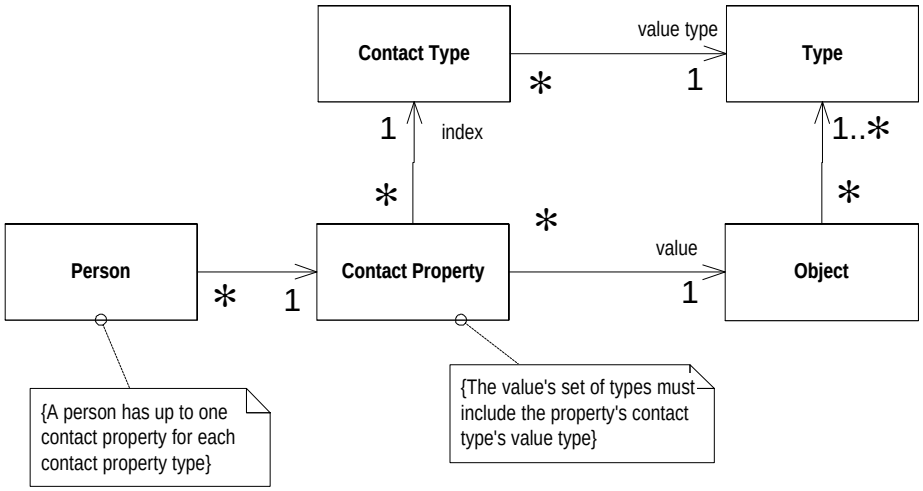

Figure 6. Model for typed dynamic property, using a separate property

```
class Person {
   public Enumeration getProperties();

class ContactProperty {
   public Object getValue();
   public Class getType();
   public ContactType getIndex();
```

Listing 10. Operations for typed dynamic properties using separate properties

Separate properties and qualified associations are not mutually exclusive. You can easily provide both interfaces at the same time. This way you get the advantages of both. Of course that makes the interface more complex, so think first about what the clients of person need. Give them the interface they need, don't make it over complex. But if they need both

qualified associations and separate properties, then that is a reasonable option. I would normally always use qualified associations for the value, but it may not make as much sense for the type.

You can also consider the interface/implementation differences here. In this paper I want to concentrate on the conceptual – which maps to the interface of software rather than its implementation. But it is worth mentioning that you can provide a qualified association interface while using separate object in the implementation. You are only using separate properties in the interface if a client of person can obtain a contact property object. Often it is useful to hide the separate property to simplify the interface for the client.

One of the big advantages of the Separate Properties is that it allows you put information about the property in the property. Such information might include who determined the property, when it was determined, and the like. The Observation pattern in [Fowler AP §3.5] build upon this theme in some depth. Much of what I've described there in terms of observations is worth considering if you need separate properties. (I see the observation pattern as a use of separate properties.)

You might be wondering at the contrast between separate properties (a pattern) and qualified associations (a UML modeling construct). You can also think of qualified associations as a pattern, a pattern of association. Indeed I did do this in [Fowler AP §15.2]. I have found it helpful to think of modeling constructs as patterns, for it helps me think about the trade-offs in using them. This is particularly useful when you are comparing them to something that is more clearly a pattern, such as separate property. Of course things that begin as patterns can be turned into modeling constructs, particularly if you use UML stereotypes. The historic mapping pattern [Fowler AP §15.3] is a good example of that. I represent that using the «history» stereotype. Is it a pattern or a modeling construction? Maybe it's a floor wax and a desert topping too.

Dynamic Properties with Multi-valued associations

My examples above have concentrated on cases where there is a single value for each key in the dynamic properties. But you can also have cases where there are multiple items in the dynamic properties. With Person you might consider a friends property which is multi-valued. There are two ways to deal with this, one that is simple and unsatisfying, another that is satisfying but (too) complex.

The simple approach is just to say that the value of a dynamic property can be a collection. We can then manipulate it like any other object with the same interface (Listing 11). This is nice and simple, since we don't have to do anything to the basic dynamic property pattern. (The example here is with a typed dynamic property, but it will work with all of them.) It's unsatisfying however because it is not really the way we would like to deal with multi-valued properties. If friends were a fixed property, we would want an interface along the lines of Listing 12. I don't like exposing the vector in these cases. By doing so the person class loses the ability to react when we add or remove elements. It also removes our ability to change the type of collection we are using.

```
Person aPerson = new Person();
ContactType friends = new ContactType("Friends", Class.forName("Vector"));
Person martin = new Person("Martin");
martin.setValueOf("Friends", new Vector());
Person kent = new Person("Kent");
martin.getValueOf("Friends").addElement("Kent");
Enumeration friends = martin.getValueOf("Friends").elements();
```

Listing 11. Using a collection value in a typed dynamic property

```
class Person {
   public Enumeration getFriends();
   public void addFriend(Person arg);
   public void removeFriend(Person arg);
```

Listing 12. Operations for a fixed multi-valued property

So can we get an interface along the lines of Listing 12, when we have dynamic properties? Well if we try hard enough we can, as you can see in Figure 7. But it is a complex model. With some clever coding we can hide much of that complexity behind the interface (Listing 13 and Listing 14) and make it reasonably convenient to use (Listing 15). But the client still needs to know which properties are single valued and which are multi-valued and all the checking for the right usage can only occur at run time. All this complexity is painful – a lot more painful than using fixed properties. I would be very reluctant to go this far.

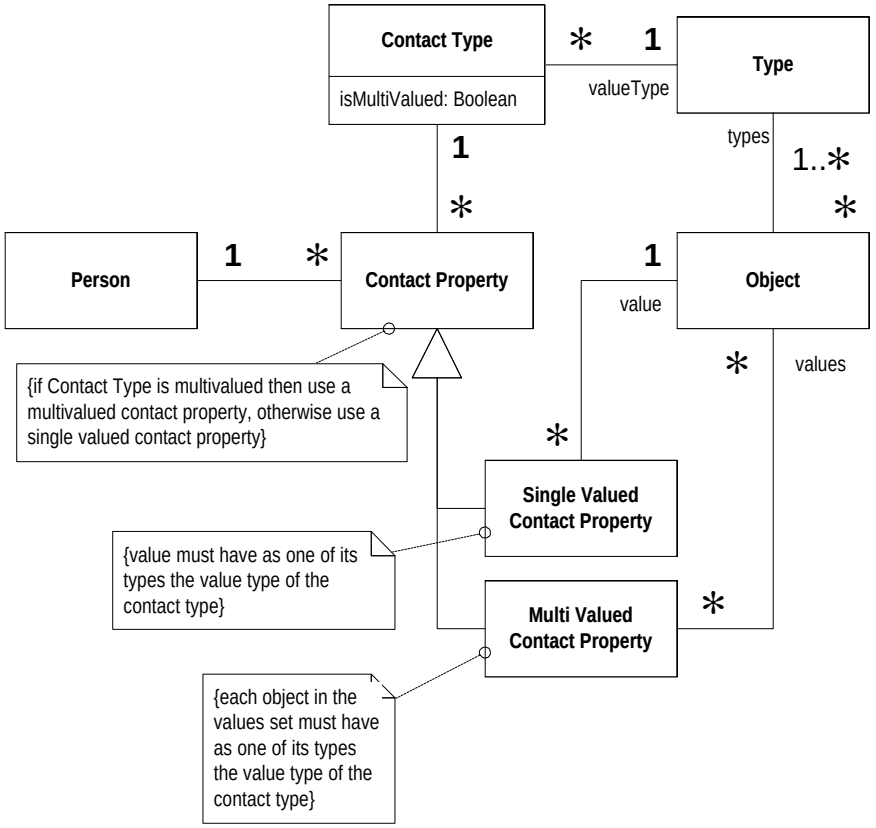

Figure 7. Satisfying but overly complex support of multi-valued dynamic properties

```
class Person
   public Object getValueOf(ContactType key);
   public Enumeration getValuesOf(ContactType key);
   public void setValueOf(ContactType key, Object newValue);
   public void addValueTo(ContactType key, Object newValue);
   public void removeValueFrom(ContactType key, Object newValue);
class ContactType
   public Class getValueType();
   public boolean isMultiValued();
   public boolean isSingleValued();
   public ContactType(String name, Class valueType, boolean isMultiValued);
```

Listing 13. Operations for Figure 7

```
class Person
   public Object getValueOf(ContactType key) {
      if (key.isMultiValued())
         throw IllegalArgumentException("should use getValuesOf()")
      //return the value
   }
   public void addValueTo(ContactType key, Object newValue) {
      if (key.isSingleValued())
         throw IllegalArgumentException("should use setValueOf");
      if (! key.getValueType().isInstance(newValue))
         throw IllegalArgumentException ("Incorrect type for property");
      //add the value to the collection
```

Listing 14. Checking for usage of operations of Listing 13

```
fax = new ContactType("fax", Class.forName("PhoneNumber"), false);
Person martin = new Person("martin");
martin.setValueOf("fax", new PhoneNumber("123 1234");
martinFax = martin.getValueOf("fax");

friends = new ContactType ("friends", Class.forName("Person"), true);
martin.addValueTo("friends", new Person("Kent"));
```

Listing 15. Using the operations of Listing 13

This complexity appears because we have both multi valued and single-valued properties. There is an inherently different interface to dealing with these, hence the complexity. Of course we can get situations with only multi-valued properties. A common pattern for this is the *Typed Relationship* pattern (Figure 8). Here a person may have number of different employment relationships with a number of different companies (or even several with the same company).

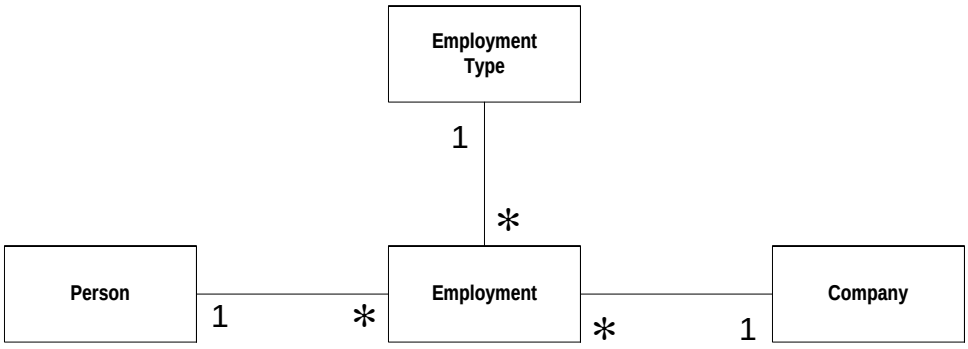

Figure 8. An example of a typed relationship

```
class Employment {
   public Employment (Person person, Company company, Employment Type type);
   public void terminate()
…}
class Person {
   public Enumeration  getEmployments();
   public void addEmployment (Company company, EmploymentType type);
```

Listing 16. Java interface for Figure 8

As we think of this, it should soon occur to you that this is really very similar as using a *Defined Dynamic Property* that is multi-valued, but expressed using a *Separate Property* rather than a *Qualified Association*. (Or is that sentence just too much of a mouthful?) Indeed Figure 9 shows how you can use a Defined Dynamic Property interface in this situation. This view of things is true, and so a typed relationship doesn't add anything that new to this pattern

language. But typed relationship is a very common pattern in modeling circles, and many may not realize its connection with the dynamic properties patterns.

Typed Relationship

How do you represent a relationship between two objects? (How do you represent multi-valued dynamic properties?)

Create a relationship object for each link between the two objects. Give the relationship object a type object to indicate the meaning of the relationship. (The type object is the name of the multi-valued property.)

The strengths of typed relationship are that it works well with bi-directional relationships, and that it provides a simple point to add properties to the relationship. (The latter, of course, is a feature of *separate properties*.) You can add a sophisticated knowledge level to this pattern, along much the same lines as *typed dynamic properties*. However you should consider the interface implications. Typed relationships force users to be aware of the employment object, as indeed does any use of separate properties. Indeed people tend to see the property object as a full fledged object in its own right, rather as some property of the person (or company). But the qualified association can often provide a simpler interface for many purposes. So whenever you see, or you are considering using, a typed relationship; you should also consider the qualified association form. You can use it in either or both directions, and use it either in addition to the typed relationship, or in addition to it.

The two models are not quite identical however. If you use Figure 9 you are indicating that an employer can only be an employer once for a particular employment type. Figure 8 has no such constraint in the diagram as it stands, although most modelers would imply such a constraint, unless the employment had additional attributes. Often, of course, the employment does have the additional attributes. A common example is a date range (as in *accountability* [Fowler, AP §2.4].

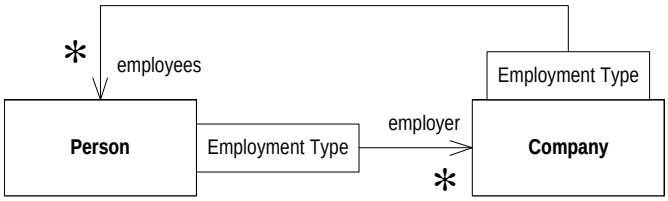

Figure 9. Using qualified associations for the same situation as Figure 8

```
public Person
   public void addEmployer (Company company, EmploymentType type);
   public Enumeration getEmployersOfType (EmploymentType type);
```

Listing 17. Interface for Figure 9

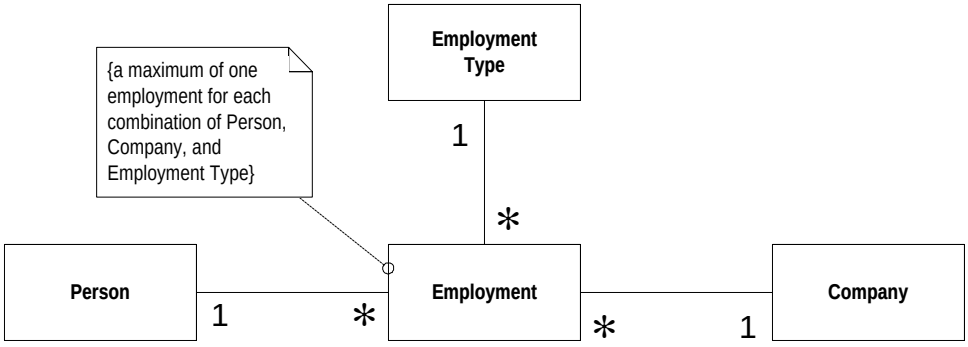

Figure 10. Typed relationship, showing the constraint that is often assumed.

Different Kinds of People

So far we are assuming we only have one kind of person, and any properties we define for a person are valid properties for all people. However you do get situations where you have different types, and different properties for different types. A manager may need a property for department managed, an executive may need a property for executive washroom key number (in a not very 90's company).

Don't worry, I hear the cries of "use inheritance stupid". Indeed this is one of the situations that is often used for subtyping. Actually it gets rather more involved than that, particularly when you start thinking of the various roles a person may play. I've written a whole paper on the subject of modeling roles [Fowler roles]. The roles patterns consider cases where we are interested in variations in operations and in variations of fixed properties. But in this case I want to explore the overlap of different kinds of object with the notion of dynamic properties. This overlap begets the *dynamic properties knowledge level* pattern (a name that is getting a little too large for my taste).

To use the pattern we give the Person a *type object* of Person Type. We can then say that the person type's association to contact type indicates which properties are available for people who have that person type. If we try to use, or ask for, a property on a person the person type can be used to check that the usage is correct.

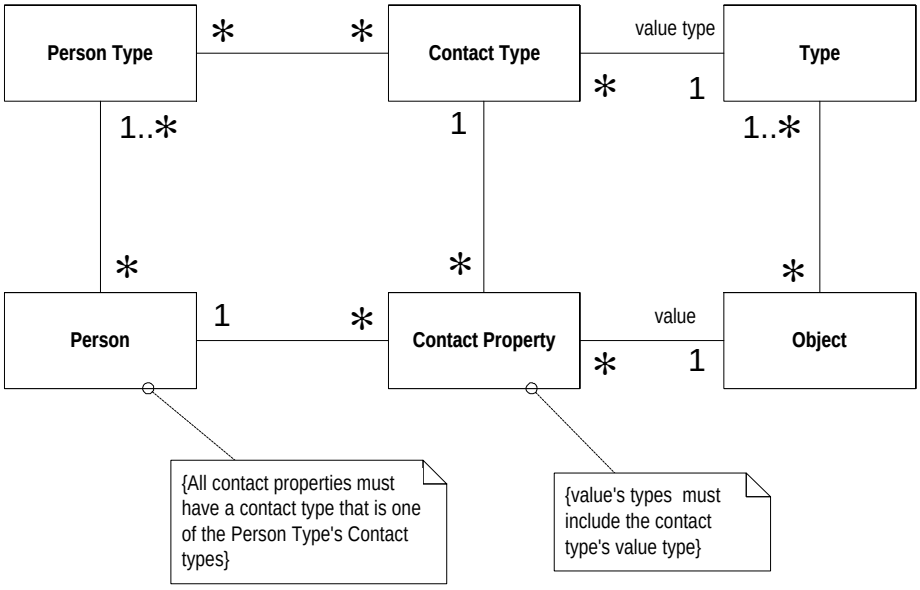

Figure 11. Dynamic Property Knowledge Level

```
class Person {
   public Object getValueOf(ContactProperty key);
   public boolean hasProperty(ContactProperty key);
   public void setValueOf(ContactProperty key, Object newValue);

class PersonType {
   public boolean hasProperty(ContactProperty key);
   public Enumeration getProperties();
```

Listing 18. Operations for Figure 11

```
class Person {
   public Object getValueOf (ContactProperty key) {
      if (!hasProperty(key))
         throw IllegalArgumentException("Innapropriate key");
      //return the value
```

Listing 19. Checking for an appropriate key with a dynamic property knowledge level

As we start to use a knowledge level like this, the separate property becomes more and more important. In this case we soon start to stop thinking of it as a property, rather as some object in its own right. The dividing line between what is a property and what isn't is very blurred, and it really depends upon your view of things.

Dynamic Property Knowledge Level

How do you enforce that certain kinds of objects have certain properties when you use dynamic properties?

Create a knowledge level to contain the rules of what types of objects use which types of properties

Some Summary Points on Dynamic Properties

Various kinds of dynamic properties make up the bulk of this paper. But I have to reiterate that dynamic properties are something I like to avoid if at all possible. Dynamic properties

carry a heavy burden of disadvantage: the lack of clarity of the interface, the difficulty in using operations instead of stored data. It is just that sometimes you have little choice but to use them, and it is on these occasions that this paper should come in useful to give you some alternatives and the trade-offs between then.

Dynamic properties appear most where there are difficulties in changing the interface. People who work with distributed object systems like them, at least in principle, because it allows them to alter an interface without compromising clients – and in a distributed system it may be very difficult to find who your clients are. But you should still be wary of doing this. Any time you add a new key to your keys for the dynamic properties you are effectively changing the interface. All the dynamic properties are doing is replacing a compile time check for a run-time check. You still have the same issues of keeping your clients up to date.

Another common use of dynamic properties is in databases. Here it is not just due to the problem of interface, but also (if not primarily) due to problems of data migration. Changing a database schema does not only cause a potential change to programs using the schema, it also may force you to do a complicated data conversion exercise. Again dynamic properties allow you to change things without changing the database schema, and thus not needing to do any data conversion. In large databases this can be a compelling advantage.

A property you don't know about

There is a last but important kind of property that I want to add to this paper. This is the case of an object having a property without realizing it. This occurs when the property is implied by another object and the way it relates to you.

Consider an object that manages database connections. It creates a bunch of database connections and hands them out to other objects when requested. When a client is done with the connection it can return it to the manager to be available for someone else. You could do this by adding an isBusy property to the connection. Figure 12 shows an alternative using an *extrinsic property*. Whether the connection is free or busy is determined by which collection in the connection manager it lies in. If you had a connection, you could not ask it whether it is free or busy, instead you would have to ask the connection manager whether a particular connection is free or busy. You can tell this because the composition associations are one way. The connection has no knowledge of the connection manager. In a sense the free/busy status isn't a property of connection at all. Yet at least in some sense it is, which is why I mention it.

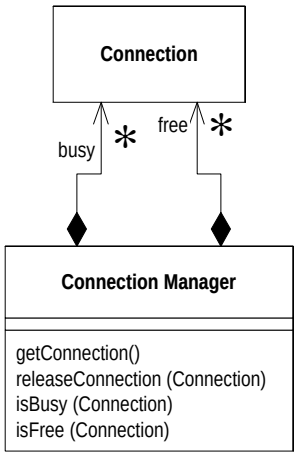

Figure 12. Using an external collection property

In a pure conceptual modeling sense this pattern does not make much sense. But there are practical implementation reasons why you may want to use it. If you want all changes to this property to go through a connection manager, then this approach makes this clear. In particular this is a natural style when you want the connection manager to give you a free connection and you don't care which one.

Another reason to use an *extrinsic property* is if the connection class is provided by someone else and you can't change its interface. You can add the new property without changing the connection class at all.

Extrinsic Property

How do you give an object a property without changing its interface?

Make another object responsible for knowing about the property.

The big problem with the extrinsic property is that it leads to an awkward and unnatural interface. Usually if you want to know something, you just find the appropriate object and ask it. Here you need to find the object that holds the external collection, and ask it about the appropriate object. In some circumstances, such as this one, it seems reasonable. But most of the time I would prefer to let objects know about their own properties (in which case I call them *intrinsic* properties).

Final Thoughts

As I finish this paper, I feel the need again to urge you not to use what I've been writing about here unless you really do need it. The advantages of fixed properties are great. If you need something else, then I hope this paper gives you some ideas and some guidance. But fixed properties are always your first choice.

References

[Fowler, AP] Fowler, Martin. *Analysis Patterns: Reusable Object Models*, Addison-Wesley 1997

[Fowler, roles] Fowler, Martin. *Dealing with Roles*, http://www.awl.com/cseng/titles/0-201-89542-0/awweb.htm

[POSA] Buschman *et al*, *Pattern Oriented Software Architecture*, Wiley 1997