Dealing with Roles

Martin Fowler
fowler@acm.org

Anyone who runs a company deals with other companies. Such companies may act as suppliers of goods or services, they may be customers of your products, they may act as agents for selling goods to your customers, they may be regulatory agencies who you have to deal with to stay on the right side of the law.

People may do many things within a company. You have engineers, salesmen, directors, accountants, any of which may need different features in the computer systems within your organization.

Dealing with these situations is one of the most common situations in modeling. You have a group of objects which exhibit bunches of common behavior. They don't all have the same behavior, but they may have some common behavior. At first glance it sounds like a classic case for inheritance, but there are complications. An object might exhibit more than one bunch of behaviors, or may take on new bunches during its lifetime. You may have an agent who is also a customer, you have an accountant who becomes an engineer.

This paper is an analysis patterns paper, hence I'm looking at the alternatives from a conceptual point of view, rather than an implementation point of view. I'm asking how we represent our view of the world in our software, and I'm not looking at important implemenation issues such as performance, distribution, concurrency etc. I've provided some code samples, but they are there to illustrate the conceptual ideas rather than as statements of what the implementation should look like. The biggest consequence to this is that I'm concentrating much more on interface than on implementation. You will notice that in particular when you compare the use of *State Object* and *Role Object*. The implementation is essentially the same, the difference is all about interface.

I've divided up how to deal with roles into five broad categories. If most of the objects have the same behavior with few variations, then you can use just one type to handle all of them (*Single Role Type).*Conversely if they are very different and have no common behavior then just treat them all as seperate types (*Separate Role Type*). These are the simple cases, and often they are the right decision.

The complication comes when there is some similar and some different behavior. Often the most obvious way of doing this is to use subtyping to show the different behaviors (*Role Subtype*). This does not mean that you necessarily use subclassing and inheritance to implement it. These are analysis patterns and thus more concerned with concepts and interfaces than with implemenatations. The key behind this pattern is that clients think they are dealing with a single object that has multiple changable types. *Internal Flag*, *Hidden Delegate*, and *State Object* are three patterns that can help you maintain this illusion. Providing the illusion makes the client's life simpler, it can be well worth the effort.

The illusion is often not worth the effort. In this case it is worth having a seperate object for each of the roles, linked to a base object that ties things together (*Role Object*). In some cases it is better to think of the role as a relationship between two objects (*Role Relationship*).

In discussing these patterns I've also mentioned a couple of others in passing. With OO programs you try to avoid asking an object whether it is an instance of a type. But sometimes that is legitamate information for a client to use, perhaps for a GUI display. Remember that asking an object if it is an instance of a type is something different than asking it if it is an instance of a class, since the types (interfaces) and the classes (implementations) can be different. *Explicit Type Method* and *Parameterized Type Method* are two ways of getting this information.

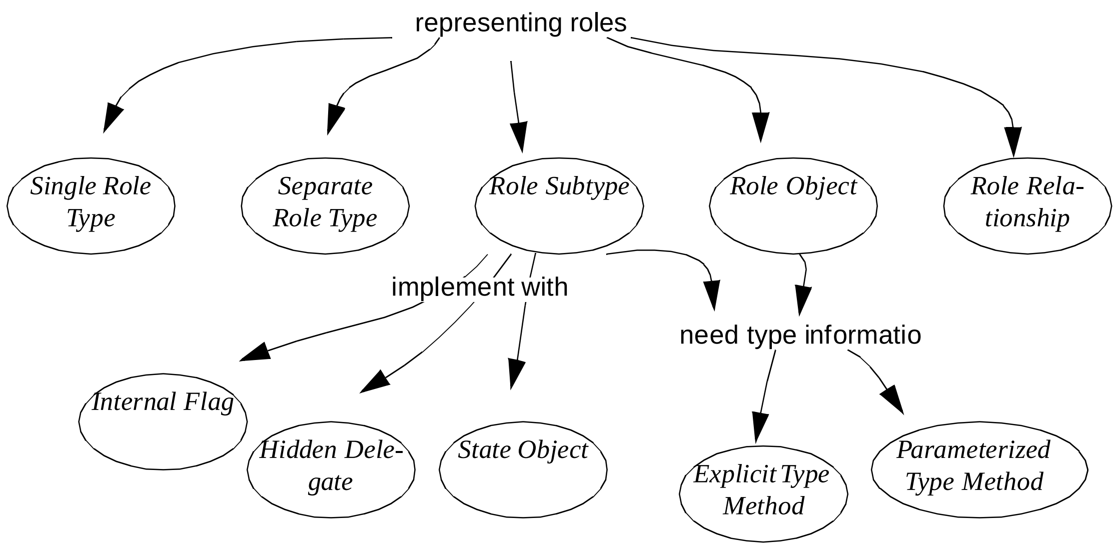
So this subject of roles brings with it several techniques. Like most issues in modeling there is no right way that works in all situations. This paper helps point out the options and the tradeoffs involved.

Problem	Solution	Pattern	p
How do you represent the many roles of an object?	Combine all the features of the roles into a single type	**Single Role Type**	4
	Treat each role as a separate type	**Separate Role Type**	4
	Make a subtype for each role. Put common behavior in the supertype	**Role Subtype**	5
	Put common features on a host object with a separate object for each role. Clients ask the host object for the appropriate role to use a role's features.	**Role Object**	14
	Make each role a relationship with an appropriate object	**Role Relationship**	17

Table 1: Table of Patterns

Problem	Solution	Pattern	p
How do you implement generalization?	Use an internal flag. Do method selection inside the class	**Internal Flag**	8
	Put the varying features inside a separate, private class. Delegate messages to this object when needed.	**Hidden Delegate**	10
	Create a hidden delegate for each subtype. Give them a common supertype with default behavior. The public class has a non-null link to the supertype. see [Gang of Four].	**State Object**	13
How do refer to the dynamic type of an object?	Use methods named isTypenam and beTypename	**Explicit Type Method**	8
	Use methods of the form hasType(typename) and beType(typename)	**Parameterized Type Method**	14

Table 1: Table of Patterns



The Simple Options

So you have engineers, salesmen, managers and accountants in your organization. Do you really need to differentiate between them?You need their job description, or some similar indicator. So, use a *Single Role Type* with an attribute either of string or of some simple job description type. If they don't have any significantly different features to them, then don't worry about trying to discriminate between them with any of the other patterns in this article.

Single Role Type

How do you represent the many roles of an object?

Combine all the features of the roles into a single type

✓ Simple

✗ Leads to a single complex type

This pattern is here because I've seen people do all sorts of things that are just not necessary. Its really a blank 'do nothing' case, but it deserves a mention because you should always ask 'is this sufficient?'. Perhaps for the moment you don't need any other pattern, but you are worried that version x of the system will need to do something at some point in the future. Well don't worry, leave it until then. The one really tangible benefit of object-oriented programming is that it allows you to change things easily, so do it then. Maybe you won't need to do it. Even if you do you may find the world looks rather different then, and the decisions you make now will not be valid. And this pattern is easy to migrate to the other patterns.

On the other hand consider a situation where you have engineers, salesmen, managers and accountants. They each have many different features. Since they are different you can make them different types: each role is *Separate Role Type*. This is what countless developers have done before you, and it carries the advantage that it separates these different types, removes any coupling between them and allows people to work on them without getting tangled up with worrying about the relationships between them.

But there are two big problems with *Separate Role Type*: duplicated features and loss of integrity. Engineers, salesmen, managers and accountants are all kinds of person, and as such they carry a lot of similar things around with them. Names, addresses, personnel information: anything that is general to people. All of these features will have to copied if you have *Separate Role Type*, so if there are any changes that affect these common features you have to track down all these copies and fix them all in the same way. This tedious task is what inheritance is supposed to fix.

The loss of integrity comes when our engineer John Smith adds some managerial responsibilities. If he is both an engineer and a manager we have to create separate objects for each role, and we cannot tell that they refer to the same person Such loss of information is important if there might be interactions between being an engineer and being a manager. This lack of integrity is a surprisingly common problem in business systems which often do this with customers and suppliers. Tying everything back together again can be quite complicated.

Separate Role Type

How do you represent the many roles of an object?

Treat each role as a separate type

✓ Simple

✗ Any shared behavior must be duplicated

✗ Difficult to deal with single object playing many roles

On the whole I don't like this pattern, because of these two problems. But you should look to see if the problems are really there. Maybe there is no common behavior, maybe you never have engineers that are managers (or it is so rare that you just don't care). If that is really the case then this pattern is fine. I would be surprised if this were the case, however, in any but a small system.

What if you are looking at a legacy system where they did this, and you need to change it to provide integrity? In these situations the patterns *Object Merge*[1] and *Object Equivalence*[2] from [Fowler] may help you out.

Using Subtyping

If your engineers, salesmen, managers and accountants have some similarities and some differences then an obvious route is that of subtyping as in Figure 1. Put the common features of each type into the person supertype, and put each additional feature of the subtypes into the appropriate *Role Subtype*. This pattern fits well our usual conception of subtyping. Engineers are special kinds of person, each instance of engineer is an instance of person, engineers inherit all the features of person but may add some features. An application can treat a collection of people at the supertype level if it doesn't care about the specialized features. If a person is both an engineer and a manager then it is of both the subtypes. If the party later becomes a retiree we add a retiree *Role Subtype* to the party.

Role Subtype

How do you represent the many roles of an object?

Make a subtype for each role. Put common behavior in the supertype

✓ Conceptually simple

✓ Interface is simple

✗ Cannot directly implement if there are multiple or changing roles

✗ Each new role causes the interface of the supertype to change

Oh dear, I hear the sad sounds of object-oriented programmers screaming. What have I done? Well its that last couple of sentences that's caused the problem. The major OO languages have single, static classification; while those offending sentences require multiple, dynamic classification. Single classification says that an object may be of only one type and inherit behavior from other types. Thus John Smith may be an engineer and thus inherit the features of person. To be a manager as well we need to create a combination engineer/manager type that multiply inherits from both engineer and manager. If we have many subtypes we will have combinatorial explosion of these combination subtypes — which is going to be a pain to manage. Multiple classification means that we can just give John Smith the engineer and manager types without having any relationship between the types. Static classification means that once an object is given a type it cannot change type, dynamic classification would allow an accountant to change to become a engineer.

So since OO programming languages have single, static classification most developers do not consider this pattern to be an option. In fact it is not as cut and dried as that. When doing analysis

1.Two objects are in fact the same therefore either: copy all the attributes of one over to the other and switch references, mark one as superseded, or link the two objects with an essence.

2.Some people think that two objects are the same therefore create an equivalence object between them.

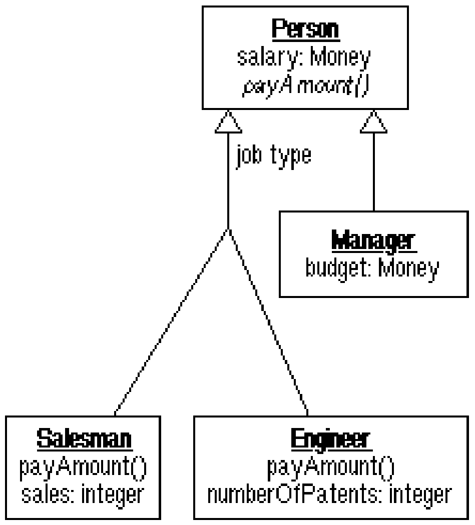


Figure 1. Using subtypes for the roles.(Notation is that of the UML [Fowler UML])

we are looking at capturing the domain experts view of the world. We then match this view of the world to the *interface* of our software components. Since we are looking at interface then this approach is possible.

Lets begin by looking at what our interfaces should be. Take the model of Figure 1. This figure implies the code in Listing 1. Everything defined in Listing 1 is an interface. This is what we need to manipulate the objects from the outside.

```
interface Person {
    public String name();
    public void name(String newName);
    public Money salary ();
    public void salary (Money newSalary);
    public Money payAmount ();
    public void makeManager ();
}
interface Engineer extends Person{
    public void numberOfPatents (int value);
    public int numberOfPatents ();
}
interface Salesman extends Person{
    public void numberOfSales (int numberOfSales);
    public int numberOfSales ();
}
interface Manager extends Person{
    public void budget (Money value);
    public Money budget ();
}
```

Listing 1. Java interfaces developed from Figure 1

So that's all very nice, but how do we implement it? There are several answers to how we implement subtyping (see [Fowler]) but here I will show three approaches: *Internal Flag*, *Hidden Delegate*, and *State Object*.

```
public class PersonImpFlag implements Person, Salesman, Engineer,
Manager{

// Implementing Salesman

    public static Salesman newSalesman (String name){
       PersonImpFlag result;
       result = new PersonImpFlag (name);
       result.makeSalesman();
       return result;
    };

    public void makeSalesman () {
       _jobTitle = 1;
    };

    public boolean isSalesman () {
       return _jobTitle == 1;
    };

    public void numberOfSales (int value){
       requireIsSalesman () ;
       _numberOfSales = value;
    };

    public int numberOfSales () {
       requireIsSalesman ();
       return _numberOfSales;
    };

    private void requireIsSalesman () {
       if (! isSalesman()) throw new PreconditionViolation ("Not
a Salesman") ;
    };


    private int _numberOfSales;
    private int _jobTitle;
}
```

Listing 2. Implementing the salesman type with flags

First I'll show how to do it with an *Internal Flag*. In this case we have a single person class that implements all four interfaces. For each partition of subtypes we need to indicate which subtype is appropriate in this case. The methods newSalesman, makeSalesman, and isSalesman (in Listing 2) show how we can do this. (For these examples I've named the type manipulation methods using *Explicit Type Method*). These methods can be added to the interface of person. When it comes to the operations defined on person we need to guard them to ensure they are only invoked on a salesman, Listing 2 does this with the private method requireIsSalesman. Any use of the salesman operations on a non-salesman results in a run-time error.

Explicit Type Method

How do refer to the dynamic type of an object?

Use methods named is*Typename* and be*Typename*

✓ Explicit interface

✗ If a new type is added the superclass's interface must change

The payAmount operation is a polymorphic operation, defined on person but implemented differently by the subtypes. We can implement this by defining a public operation that is essentially a case statement based on the type indicator. This kind of type based case statement is generally a bad idea in object-oriented software, but here it is fine because it is hidden within the person class. Since we cannot use polymorphic methods we use the internal case statement to imitate them

```
    public Money payAmount (){
       if (isSalesman()) return payAmountSalesman();
       if (isEngineer()) return payAmountEngineer();
       throw new PreconditionViolation ("Invalid Person");
    };

    private Money payAmountSalesman () {
       return _salary.add (Money.dollars(5).multiply
(_numberOfSales));
    };
    private Money payAmountEngineer () {
       return _salary.add (Money.dollars(2).multiply
(_numberOfPatents));
    };
```

Listing 3. The flags implementation of the polymorphic payAmount *method.*

The *Internal Flag* provides a reasonable implementation of this kind of more complex classification. It does, however, result in a complex person class which has to take on all the data and behavior of the subtypes, as well as provide the method selection capabilities. As these responsibilities grow they all get lumped into single class: which can become the kind of beast that will stalk your nightmares.

Internal Flag

How do you implement generalization?

Use an internal flag. Do method selection inside the class

✓ Supports multiple, dynamic classification

✗ The implementing class has the features of all the types it implements.

Another implementation is to use a *Hidden Delegate*. In this case the additional data and behavior required by the subtype is implemented in a separate class. This class is hidden because any client class does not see it. The client software still invokes methods on the person, the pers class then hands off the methods to the delegate when required. Listing 4 shows how this is done for the manager type. All the public methods declared in the manager interface have to be declared and implemented in person. The implementation in person checks that the receiving object is a manager, and if so delegates the request to the hidden manager object.

```
public class PersonImpHD implements Person, Salesman, Engineer,
Manager{
    // implement manager
    public void makeManager () {
       _manager = new ManagerImpHD();
    };

    public boolean isManager (){
       return (_manager != null);
    };

    private void requireIsManager () {
       if (! isManager()) throw new PreconditionViolation ("Not a
Manager") ;
    };

    public void budget (Money value) {
       requireIsManager();
       _manager.budget(value);
    };

    public Money budget () {
       requireIsManager ();
       return _manager.budget();
    };

    private ManagerImpHD _manager;
}
class ManagerImpHD {

    public ManagerImpHD () {
    };

    public void budget (Money value){
       _budget = value;
    };

    public Money budget (){
       return _budget;
    };

    private Money _budget;

}
```

Listing 4. Implementing manger using a hidden delegate

When using the hidden delegate in this way we move the manager specific behavior and data to the manager object. This reduces the complexity of the person class. It is a significant difference if the *Role Subtype* contains a lot of additional behavior and features. The bad news is that person class still has all the interface of the *Role Subtype* and also has to do the method selection: the decision of whether to pass the message to the *Hidden Delegate*.

Hidden Delegate

How do you implement generalization?

Put the varying features inside a separate, private class. Delegate messages to this object when needed.

- ✓ Supports multiple, dynamic classification
- ✓ Varying behavior is separated into the delegate
- ✗ Method selection is done by the public class.

We can deal with the method selection aspect of the problem by using the *State Object* of [Gang of Four]. This works very well when we have some mutually exclusive subtypes, such as salesman and engineer. The subtypes are each implemented with hidden objects. The hidden classes are given a superclass. The person class contains a reference to the superclass object. This hidden hierarchy now has the responsibility for determining method selection and type testing, and it can use inheritance and polymorphism to do this.

Listing 5 and Listing 6 show how we can use *State Object* for salesman. The data and behaviors are defined on the salesman hidden object. The superclass hidden object (JobHD) provides a default behavior, which is to indicate an error. The person provides the overall public interface and delegates the method in its entirety to the hidden delegate. See the contrast between this and the manager in Listing 4, in this case the person does nothing other than delegate.

The technique works particularly well for payAmount, as shown in Listing 7. In this case the state object requires some information from person. This is done by *Self Delegation* [Beck]: passing the person as an argument when the delegation is done.

```
public class PersonImpHD implements Person, Salesman, Engineer,
Manager{

    public static Salesman newSalesman (String name){
       PersonImpHD result;
       result = new PersonImpHD (name);
       result.makeSalesman();
       return result;
    };

    public void makeSalesman () {
       _job = new SalesmanJobHD();
    };

    public boolean isSalesman () {
       return _job.isSalesman();
    };

    public void numberOfSales (int value){
        _job.numberOfSales(value);
    };

    public int numberOfSales () {
       return _job.numberOfSales();
    };

    private JobHD _job;
```

Listing 5. The PersonImpHD class using a state object to implement the salesman interface.

```
abstract public class JobHD {

    private void incorrectTypeError() {
       throw new PreconditionViolation("Incorrect Job Type");
    };

    public boolean isSalesman () {
       return false;
    };

    public void numberOfSales (int value) {
       incorrectTypeError();
    };

    public int numberOfSales (){
       incorrectTypeError();
       return 0;//value not returned since exception is thrown,
compiler needs return
    };
}
public class SalesmanJobHD extends JobHD{

    public boolean isSalesman () {
       return true;
    };

    public void numberOfSales (int value){
       _numberOfSales = value;
    };

    public int numberOfSales () {
       return _numberOfSales;
    };

    private int _numberOfSales = 0;
}
```

Listing 6. JobHD and its subtype for salesman

```
public class PersonImpHD implements Person, Salesman, Engineer,
Manager{
    public Money payAmount (){
       return _job.payAmount(this).add(managerBonus());
    };
…
abstract public class JobHD {
    abstract public Money payAmount (Person thePerson);
…
public class SalesmanJobHD extends JobHD{

    public Money payAmount (Person thePerson) {
       return thePerson.salary().add (Money.dollars(5).multiply
(_numberOfSales));
    };
```

Listing 7. Implementing payAmount using a state object

If you want to use this with an incomplete partition, i.e. if a person might be neither a salesman or an engineer but a vanilla person, then you can either make JobHD a concrete class, or create a nullJobHd class for that case.

State Object

How do you implement generalization?

Create a hidden delegate for each subtype. Give them a common supertype with default behavior. The public class has a non-null link to the supertype. see [Gang of Four].

✓ Supports multiple, dynamic classification

✓ Varying behavior is separated into the delegate

✓ Method selection is done by inheritance and polymorphism.

✗ An effort to set up

I must stress the common theme behind all these alternatives: each implementation lies behind the same set of interfaces declared in Listing 1. We can switch any of these implementations without affecting any of the users of these interfaces.

I tend to use *Internal Flag* if there are not many features involved, and*Hidden Delegate* or *State Object* if things are more complex. Since they all the same interface you can safely work with the simplest approach to begin with and change it later when it begins to get awkward.

Turning the roles into separate objects

An alternative to *Role Subtype* is to use *Role Object*. In this case we consider engineer, salesman, accountant, manager, and retiree to all be roles that the employee mayplay. Person has an multi-valued association with person role where person role is the supertype of engineer, salesman, accountant, manager, and retiree. This, referred to by [Coad] as the *actor-participant* pattern, is a common technique for these circumstances.

The implementation looks very similar to using the *State Object* pattern, the difference lies in the interface. When using *Role Subtype* with *State Object*, the state objects are entirely hidden from the user of the class. To find a manager's budget a user asks the person object, which then makes the appropriate delegation. When using *Role Object*, however, the user asks the person object for its manager role, and then asks that role for the budget. In other words the roles are public knowledge.

Roles are most often used in the style of Figure 2, where each person has a set of role objects for its various roles. Listing 8 shows how person adds and accesses these roles. We begin with the person, as before, but changing the amount of sales for a salesman now requires two steps: first we find the salesman role for the person, then we change the value.

Role Object

How do you represent the many roles of an object?

Put common features on a host object with a separate object for each role. Clients ask the host object for the appropriate role to use a role's features.

✓ Direct implementation
✓ If you add a new role you don't have to change the host's interface.
✗ Awkward if roles have constraint
✗ Exposes role structure to clients of the host object

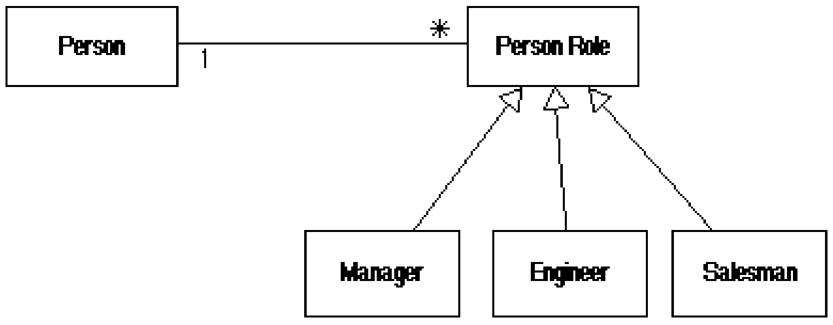


Figure 2. Using role objects for person.

In this case I have used a different approach for the type information. In Listing 6 I used a method isSalesman which defined in the superclass to be false and overridden in the subclass. I could use the same approach in this case. However, for variety's sake, I've shown a different approach which uses a parameterized hasType(String) method. The hasTyp approach has the advantage that we don't need to change the interface of person role or person when we add a new role. Its disadvantages are that it makes the interface less explicit and the compiler cannot check the correctness of the parameter. I usually prefer the explicit form, but find the parameterized form useful when interface changes due to new types become too much to be worth the compile time checking.

Parameterized Type Method

How do refer to the dynamic type of an object?

Use methods of the for hasType(*typename*) and beType(*typenam*)

✓ New types do not cause a change to the superclass
✗ Interface is less explicit

```
class Person {
    public void addRole(PersonRole value) {
       _roles.addElement(value);
    };

    public PersonRole roleOf(String roleName) {
       Enumeration e = _roles.elements();
       while (e.hasMoreElements()) {
          PersonRole each = (PersonRole) e.nextElement();
          if (each.hasType(roleName)) return each;
       };
       return null;
    };

    private Vector _roles = new Vector();
…
}
public class PersonRole{

    public boolean hasType (String value) {
       return false;
    };
    …
public class Salesman extends PersonRole{

    public boolean hasType (String value) {
       if (value.equalsIgnoreCase("Salesman")) return true;
       if (value.equalsIgnoreCase("JobRole")) return true;
       else return super.hasType(value);
    };

    public void numberOfSales (int value){
       _numberOfSales = value;
    };

    public int numberOfSales () {
       return _numberOfSales;
    };

    private int _numberOfSales = 0;
}
// To set a salesman's sales we do the following
       Person subject;
       Salesman subjectSalesman = (Salesman)
subject.roleOf("Salesman");
       subjectSalesman.numberOfSales(50);
```

Listing 8. Salesman as one of a set of roles on person.

This common use of roles does not really provide the same model as Figure 1. In Figure 1 there are definite rules as to what combinations of roles can be used. You must be one of either a salesman or engineer, and you may be a manager in addition. To preserve the rules we have two alternatives. One alternative is to change the addRole method to check the validity of a new role as in Listing 9. A better way, however, is to alter the interface of person to make the restriction more explicit as in Listing 10.

```
class Person {
    public void addRole(PersonRole value)throws CannotAddRole {
       if (! canAddRole(value)) throw new CannotAddRole();
       _roles.addElement(value);
    };

    private boolean canAddRole(PersonRole value){
       if (value.hasType("JobRole")){
          Enumeration e = _roles.elements();
          while (e.hasMoreElements()) {
             PersonRole each = (PersonRole) e.nextElement();
             if (each.hasType("JobRole")) return false;
          };
       };
       return true;
    };
…
```

Listing 9. Checking a added role is valid

```
class Person {

    public void jobRole(JobRole value){
       _jobRole = value;
    };

    public PersonRole jobRole() {
       return _jobRole;
    };
    private JobRole _jobRole;
}
```

Listing 10. Using an explicit interface to capture the constraints of job roles in person.

In the kind of situation here, where there are not too many roles and there are restrictions, I tend to prefer using *Role Subtype*. *Role Subtype* provides a simple single class interface. The disadvantage of *Role Subtype* is that the interface of person has to include all the methods defined on the roles, which is awkward if the role interface is large or volatile. If it is awkward to continually change the person's interface then that is the force that drives me towards *Role Object*. In most situations where that force is present the set of roles of Figure 2 makes the most sense. Any rules on combinations of roles must be captured: if they are simple I use the technique of Listing 9, if they are more complex then I will change the interface and use Listing 10 style. In practice I find that the situations that propel to a Listing 10 style interface are best solved using*Role Subtype* anyway. Remember that you can use one technique for some roles and a different one for other roles.

A useful variation on this pattern is to make the role objects decorators of the core object. This means that a client who uses only the features of employee person can deal with a single object and not need to know about the use of role objects. The cost is that when ever the interface of person changes, all the roles need to be updated. See [Bäumer et al] for more details on how to do this.

The final alternative in modeling roles that I'm discussing in this paper is *Role Relationship*. The need for *Role Relationship* comes up when you consider an organization with several different groups, where an employee might have roles in more than one group. A salesman may begin with one group, change to another, take retirement, but still do some work for another. In this case in each role we need to know not just the role, but also which group the employee keeps the role with. You can think of the role as an employment relationship between the employee and the group. As soon as you start thinking of a role as a relationship the *Role Relationship* pattern of Figure 3 starts to make sense.

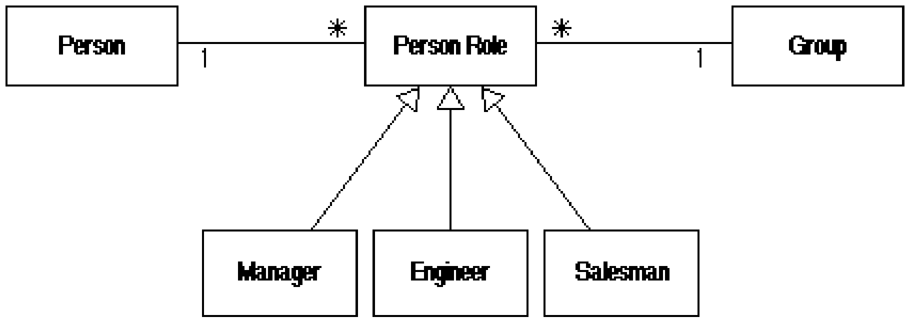


Figure 3. Treating the role as a relationship

Using and implementing *Role Relationship* is pretty similar to *Role Object* (see Listing 11), the main difference is in the fact that setting up and accessing the roles needs a group as a parameter. If there are constraints between the roles, then you can use the same techniques as with *Role Object*.

Role Relationship

How do you represent the many roles of an object?

Make each role a relationship with an appropriate object

✓ Natural if the host object may have more than one role in relationship to another object.

A common situation with this pattern is that the *Role Relationship* change over time, and a history of these changes needs to be kept. There might also be rules that indicate which kinds of people can have what kinds of *Role Relationship* with different kinds of groups. *Accountability* and related patterns in [Fowler] discuss these kinds of complications furthe .

Which Role to Choose

Whenever you have a situation that involves roles, never forget that you have several options to choose from in modeling it.

Are the role significantly different? If the differences are minor the I use *Single Role Type*. I don't worry about it not being flexible enough. If I need the flexibility later, it is easy to refactor it.

Are there any common behaviors between the roles? If not I might go for *Separate Role Type*. I'm always wary of the integrety issue though. This pattern can be more awkward to shift later on if this crops up.

Only if I know I have significant common and shared behavior, and I need to ensure integrety do I need to go to the heavyweight options. Here often the key decision is between *Role Subtype*

```
class Person {
    public void addRole(PersonRole value)throws CannotAddRole {
       _roles.addElement(value);
    };

    public PersonRole roleOf(String roleName, Group group) {
       Enumeration e = _roles.elements();
       while (e.hasMoreElements()) {
          PersonRole each = (PersonRole) e.nextElement();
          if ((each.hasType(roleName)) && (each.group() ==
group)) return each;
       };
       return null;
    };

    private Vector _roles = new Vector();
…
public class PersonRole{

    protected PersonRole (Group group){
       _group = group;
    };

    public Group group(){
       return _group;
    };

    protected Group _group;
…
public class Salesman extends PersonRole{

    public Salesman (Group group){
       super (group);
    };
…
```

Listing 11. Using a relationship for the role.

and *Role Object*. Many authors will tell you to always use *Role Object*, I don't agree. The key question I ask is what is the best interface for the users of my classes? Three indicators suggest *Role Subtype*: there aren't too many roles, new roles do not appear often, and there are significant rules about what combinations and migrations of roles that can occur. If those forces are present then I go with *Role Subtype*. That way I present a simpler interface. I make that decision, and then choose how to implement it. The implementation is a little harder than*Role Object*, but I'm saving my clients that same amount of work. Since there are many of them, that can be a considerable favor.

I would choose *Role Object* when either I have a lot of roles, or I often get new roles. These forces would lead to a large or volatile interface for a *Role Subtype*. By this point it would become just too much work to keep up the *Role Subtype*.

I consider *Role Relationship* when the roleness I'm interested in is all about a relationship between the core object and some other object. This is a particularly true if the core object can play the same role with many other objects, or if I want to track the changes to the roles that the core object plays over time.

A Role with Many Players

All of these patterns make a common assumption, that a role is something played by a single core object within some context. But this assumption is not true of all situations that suggest roles. When I ring up an airline to book a reservation, I am looking for someonw to play the reservation role for me, but I don't care which person plays it. In this situation the role is the key element but the person is irrelevent. I can ring up three times about the same reservation and I will usually have a different person each time. In this situation I don't usually know very much about the role.

A slightly different situation appears when you have consistent on-going dealings with a role where there is usually a single core object, but it changes over time. This might occur when you work with a regional salesman for your area. The actual person changes but the role stays the same. Again the role is more important to you than the person.

I'm not going to discuss these situations in this paper, at least not until a later version. I still need to think more about the patterns I have seen for this kind of problem. The latter case (a role with many actors over time) can be modeled using the *Post* pattern[Fowler].

Summing up

I wrote this article because I was frustrated. This frustration is good source for patterns. Often it's the frustration of seeing people do the same thing wrong time and time again. In this case the frustration stems from people who use one of these patterns all the time, and decry the others. I don't object to any of these patterns — what I object to is a practice of using one of them all the time.

Design is all about trade-offs, and every one of these patterns comes with its own set of trade offs. Any of the patterns in this paper can be the right pattern for your problem. Dogmatic adherence to one pattern is not the answer, you have to understand an evaluate the trade-offs. This truth is eternal in design.

Further Reading

Roles have long been a technique known to the object and data modeling communties. I considered writing about them in *Analysis Patterns* [Fowler], but considered the subject too trivial (rather like saying numbers can support arithmetic). I changed my mind because so often people will describe one of these patterns (particularly *Role Object*) as the only solution without talking about the full set of trade-offs. Hence there is no real treatment of the subject in *Analysis Patterns,* except for the discussion of Accountability (2.4) which is a use of *Role Relationship.*

[Bäumer et al] provides a in-depth discussion of implementing *Role Object*. Their approach makes the role object a decorator [Gang of Four] of the base object. This means that clients can deal with the role object without knowing that the base object is in fact a seperate object. If that helps simplify things for the client, it is a useful tactic.

[Gamma, Extension] discusses a pattern for extending the behavior objects using a similar approach to that of role objects. The solution is similar to that of *Role Object* but the intent is somewhat different. The Extension pattern is mainly about dealing with unanticipated changes to an objects services, while the *Role Object* pattern is more about dealing with an object playing many seperate roles.

[Schoenfeld] discusses a couple of uses of *Role Object* with people and documents.

[Hay] has many examples of role modeling problems dealt with in a relational flavor.

[Coad] describes the Actor-Partcipant pattern, which is essentially*Role Object*, and gives many examples of using it.

I should mention that a proper reference to everybody who has discussed using roles would consume more pages than this paper. Using roles in some way or another is to modeling what if-then-else statements are to programming. Thus there is a lot that has been written on the subject, usually in an off-had way that is characteristic when people write about very well-known constructs. My apologies to the many people I could have referenced here, but did not.

References

[Arnold/Gosling] Arnold, K. and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, MA, 1996.

[Beck] Beck, K. Smalltalk Best Practice Patterns, Prentice Hall, Englewood Cliffs, NJ, 1997

[Bäumer et al] Baumer D, Riehle D, Siberski W and Wulf M *The Role Object Pattern*, submitted for PLoP 97

[Coad] Coad, P., North, D. and Mayfield, M. *Object Models: strategies, patterns and applications*, Prentice Hall, Englewood Cliffs, 1995.

[Fowler] Fowler, M *AnalysisPatterns: reusable object models*, Addison-Wesley, Reading MA, in press.

[Fowler UML] Fowler M with Scott K, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1997.

[Gang of Four] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *DesignPatterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.

[Gamma, Extension] Gamma E, *The Extension Objects Pattern*, Submitted to PLoP 96

[Hay] Hay D, *Data Model Patterns: Conventions of Thought*, Dorset House 1996

[Meyer] Meyer, B. "*Applying "Design by Contract"*," IEEE Computer, 25,10, (1992), pp. 40–51.

[Odell] Martin J and Odell JJ *Object Methods: Pragmatic Considerations*, Prentice Hall 1996

[Schoenfeld] Schoenfeld A, *Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles*, submitted to PLoP 96