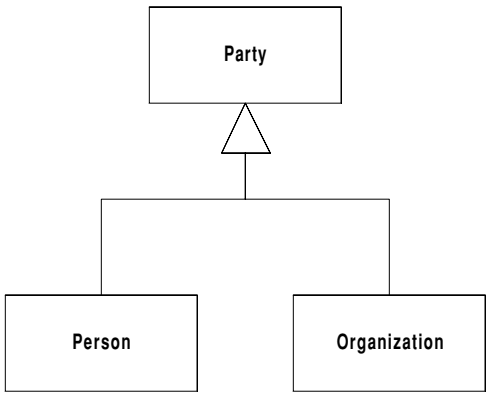
Party

An abstraction of people and organizational units



Example: *A telephone utility's customers may be individuals or businesses. Many aspects of dealing with customers are the same, in which case they are treated as parties. Where they differ they are treated through their subtype.*

Take a look through your address book, what do you see? If its anything like mine you will see a lot of addresses, telephone numbers, the odd email address… all linked to something. Often that something is a person, however the odd company shows up. I call 'Oak Grove Taxi' frequently, but there's no person I want to speak to — I just want to call a cab.

If I were to model an address book, I might choose to have people and companies, each of which have postal addresses and telephone numbers, but the resulting duplication is painful to the eye. So I need a supertype of person and company. This class is a classic case of an unnamed concept — one that everybody knows and uses but nobody has a name for. I have seen it on countless data models on various names: person/org, player, legal entity….

The term I prefer is party, and I've been glad to see that over the last few years it's turned into a fairly standard name.

Making it work

I usually define party as the supertype of person and organization. This allows me to have addresses and phone numbers for departments within companies, or even informal teams.

Put any behaviror that is common to people and organizational units on Party, only put things particular to one or the other on the subtype. When you put behavior on the subtype, think about whether it makes sense on the supertype, often you may be surprised how well things fit on the supertype

When to use it

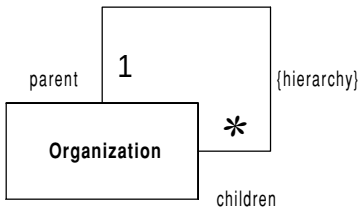
The obvious case to use *Party (15)* is when you have people and organizations in your model and you see common behavior. However you should also consider this pattern when you don't need to distinguish between people and organizations. In this case it's useful just to define a party class and not to provide the subtypes.

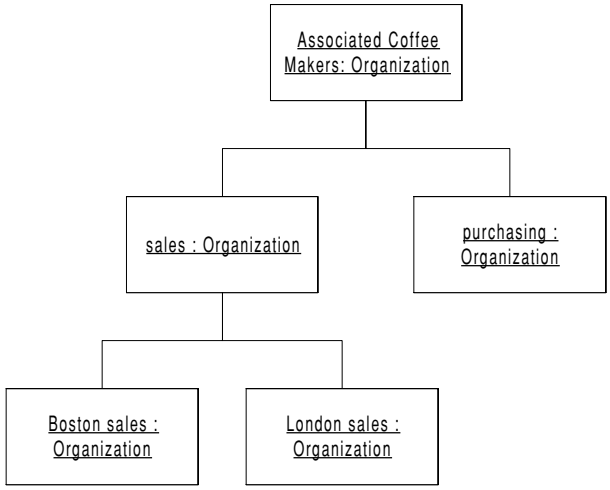
If you see parties playing different roles in a business, you'll often see one of the role patterns in play. Coad in particular is a big fan of *Role Object (84)*. It's a common pattern with party, but isn't one to turn to blindly, but take a look at *Role Object (84)* for more discussion of that issue.

The main point of this pattern is to look for it to see if you have common behavior, and if so to use the name Party for that supertype. The name has become quite widely used these days, so choosing that name helps in communication.

Organization Hierarchy

Represents the hierarchy of organizational units within a business





Making it work

Hierarchies are a common structure in organizations. This is because they reflect a natural and common technique for human beings to deal with complexity. Modeling a hierarchy is thus a common thing to do, yet it is both easy and complicated.

It's easy because you can show it with a recursive association, my old data model teachers called it a "pigs ear". However the recursive association does not tell the whole story. In most cases there is one parent, but how about for the one at the top? Hierarchies also have rules, such as the fact that you can't have cycles (your grandparent cannot be your grandchild). There is no standard notation for dealing with this in the UML. I use a constraint {hierarchy} on the association to suggest what is going on.

Even the {heirarchy} constraint, however, is strictly imprecise. It doesn't tell you the difference between a tree (one top) and a forest (multiple tops). If this is important to you model you should say what it is, most of the time I find it isn't that vital.

In the sketch I use the terms parent and children. Naming these association roles can be tricky. Parent works well with most companies, but child isn't as good as subsidiaries. However I've come to the view that children is the best name. The use of parent and child is a very useful metaphor when discussing hierarchies, or indeed any other kind of directed graph structure. I can use such phrases as "sales in London is a cousin of purchasing in Java" and, although it sounds wacky, you can easily figure out what I mean. The metaphor gives us a useful vocabulary which is worth the fact that it often sounds a little odd.

When to use it

You need to use this pattern when you have a hierarchic corporate structure that affects the software you are using.

- hierarchic: because the pattern handles a hierarchy not anything more complex. If your needs are more complicated you can tweak the pattern (there are some suggestions below) or use *Accountability (27)* instead.
- affects the software: because you only need to capture the corporate organization if it really affects what you are doing. It's important to ask yourself what would happen if you didn't capture the hierarchy. If the consequences are not painful then it's not worth carrying the cost of the links (and above all the costs of maintaining the links). Remember that it's easy to add this pattern in later to an exisiting list of organizations or parties.

GOF fans will note that this pattern is an application of the GOF composite pattern, although a somewhat degenerate one as the composite and leaf classes aren't pulled out. Certainly you should consider using composite when you implement it, and bear in mind that it's quite common not to have distinguishing behavior between composites and leaves in this use.

You don't need to limit yourself to a single hierarchic association. If an organization has different structures for regional vs functional management, you can use a pair

of hierarchies as in Figure 0.13 This allows sales in the London office to have sales as its functional parent and the London office as its regional parent. Of course as you get more hierarchies this will get messy, and this is the point where you'll want to use *Accountability (27)*.

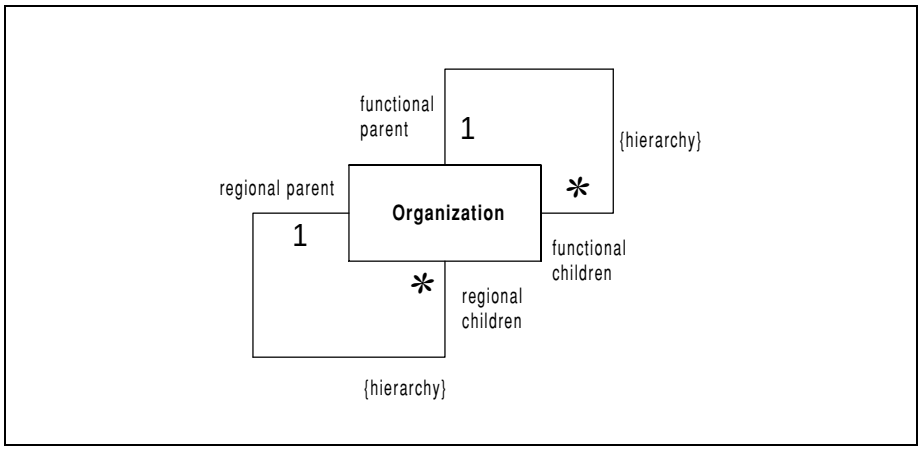


Figure 0.13 *Using a pair of hierarchies*

Similar logic can lead to abandoning the heirarchy in favor of a more general graph structure to allow you to have multiple parents. Here I would be more cautious. Ask yourself whether you really have different kinds of parents that should be distinguished. In many cases you'll find that multiple hierarchies or *Accountability (27)* will serve you better. Once you lose the single parent, you lose the ability to easily aggregate up the hierarchy, in particualar you won't be able to use *Aggregating Attribute (24)*.

Sample Implementation

The key thing about implementing this pattern is to get the right kind of interface on the classes. This means providing the right mix of operations to navigate the structure how you need it to be navigated.

For this sample implementation I'm using the static variable as a *Registry (2)* for organizations.

```
class Organization ...
  private static Map instances = new HashMap();
  private String name;
  void register () {
      instances.put(name, this);
  }
  static void clearRegistry() {
      instances = new HashMap();
  }
  static Organization get(String name) {
      return (Organization) instances.get(name);
  }
```

At the core, we need operations to get hold of the parent and the children. In this case I use a field to store the parent, and determine the children by a lookup from the registry.

```
class Organization ...
  private Organization parent;
  Organization (String name, Organization parent) {
      this.name = name;
      this.parent = parent;
  }
  Organization getParent() {
      return parent;
  }
  Set getChildren() {
      Set result = new HashSet();
      Iterator it = instances.values().iterator();
      while (it.hasNext()) {
          Organization org = (Organization) it.next();
          if (org.getParent() != null)
             if (org.getParent().equals(this)) result.add(org);
      }
      return result;
  }
```

With these methods in place, we can then add other methods to navigate the structure further. Here are some examples.

```
public Set getAncestors() {
    Set result = new HashSet();
    if (parent != null) {
        result.add(parent);
        result.addAll(parent.getAncestors());
    }
    return result;
}

public Set getDescendents() {
    Set result = new HashSet();
    result.addAll(getChildren());
    Iterator it = getChildren().iterator();
    while (it.hasNext()) {
        Organization each = (Organization) it.next();
        result.addAll(each.getDescendents());
    }
    return result;
}

public Set getSiblings() {
    Set result = new HashSet();
    result = getParent().getChildren();
    result.remove(this);
    return result;
}
```

Notice how using the familial metaphor means that the method names clearly communicate what's being returned by these queries.

Not just are these queries useful for navigation, they are also essential to ensure that you don't include a cycle in the hierarchy. The cycle can be prevented with check along the following lines.

```
void setParent(Organization arg) {
    assertValidParent(arg);
    parent = arg;
}
void assertValidParent (Organization parent) {
    if (parent != null)
        Assert.isFalse(parent.getAncestors().contains(this));
}
```

Notice that you only need the check on the modifier, you don't need it on creation. Also if you're thinking of setting the parent and then trying the assert, run a test before you release it.

Variation: Subtypes for Levels

In some cases you may find variation between the various levels in the heirarchy. So an enterprise may be divided into divisions that are divided into departments. If there

is no specific behavior to the organizations at these three levels, then a single organization class will do the trick for all of them. However if there is variation, then it can be worthwhile to form subtypes for them as in Figure 0.14

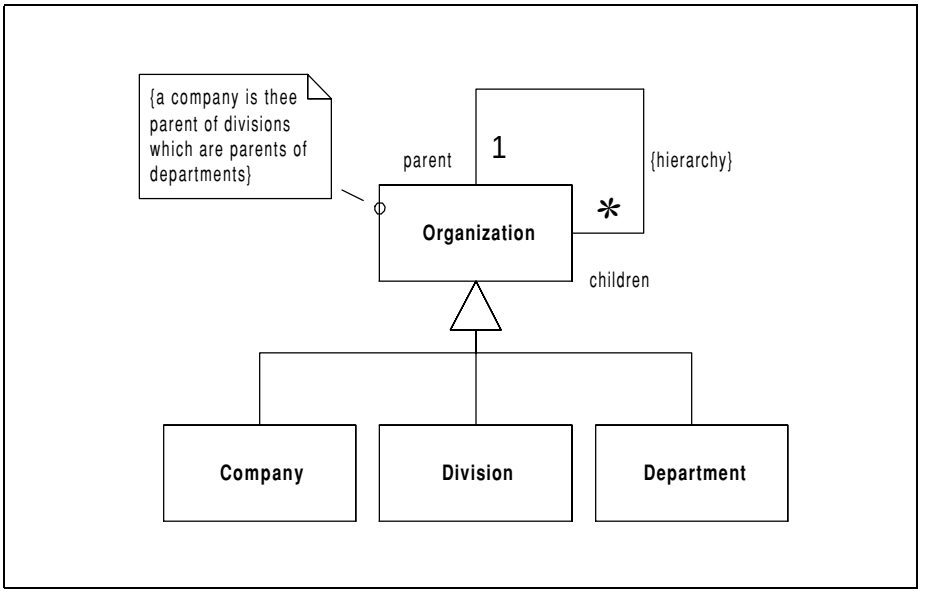


Figure 0.14 *Using subtypes for the various levels*

In this case you have a choice, is it better to put the association on the supertype, or have several associations on the subtypes. The decision rests on how you work the navigation. If you more often find you navigate around organizations, then it's better on the organization. If the explicit levels tend to matter more, then put the associations on the subtype. If both occur then you can provide both interfaces. I usually find the single association on organization easier to work with as it usually removes duplicate behavior. You can enforce the constraint when you assign the parent.

If there is no common behavior between the organizational groups, then you can ditch the organization class completely, although I confess I've never done this. Usually I find there's some duplicate behavior lying around somewhere that I'd rather have on the parent.

Sample Implementation

One way of doing the parent checking is to override the parent checking code to put a run time type check into place.

```
class Division extends Organization ...
  void assertValidParent (Organization parent) {
      Assert.isTrue(parent instanceof Company);
      super.assertValidParent(parent);
```

Of course this is only a run time check, and you may prefer a compile time check. You can partly do this by restricting the type on the constructor.

```
class Division...
  Division (String name, Company parent) {
      super(name, parent);
  }
```

Putting this checking into the modifier is more awkward. To get it right you need to remove the Organization's modifier and put appropriate modifiers only on the subtypes.

Aggregating Attribute

Allow a child to use an attribute value from a parent

Often you may find that an organization shares attributes from its parents to its children. Thus the head company, the sales division, and the Boston sales group all use US dollars as their accounting currency. This is not a coincidence, in fact the intent is that a child uses the accounting currency of the parent unless there is a specific override.

Making it work

Showing this on a model can be awkward. From the specification perspective we would say that every organization has an accounting currency, ie the multiplicity is 1. But the implementation picture says the currency is optional. You need a field, but a null would indicate you should look at the parent.

I model this with a stereotype to show an aggregating association as in Figure 0.15 .Again there's no standard UML way to show this, so a little light extending makes sense. This notion of an aggregating association or attribute can crop up whenever you have some form of composite, so it's a handy extension.

tbd ref to SanFrancisco

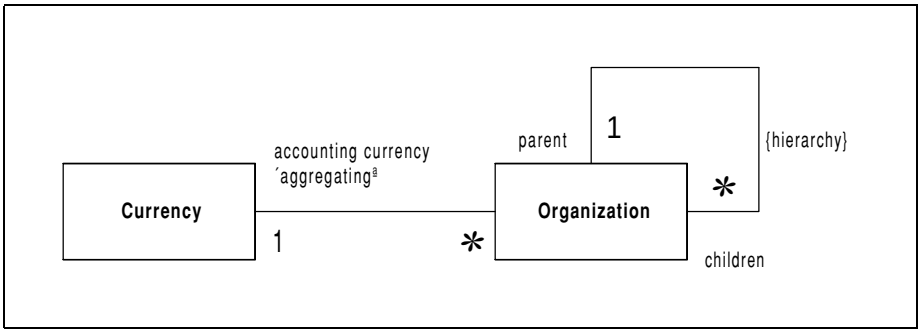


Figure 0.15 *Aggregating association for currency*

When to use it

I find this pattern comes naturally with any form of hierarchic structure. The key question to ask is whether a change in a value for a parent should affect all the children. If so, then is sounds like *Aggregating Attribute (24)*. (If the parent provides a default which is adopted by a child but subsequent changes to the parent do not change children with the same value, then that's not *Aggregating Attribute (24)*.)

Of course this will only work well when there is a clear, single parent. If you have multiple hierarchies, or you're using *Accountability (27)* this becomes more complicated. In these situations you need to indicate clearly which parent you're using as the basis for the aggregating behavior, and be sure that you are dealing with a hierarchy rather than a more general graph structure. If you have a more general graph structure which really does need multiple parents, then you're hosed and can't use *Aggregating Attribute (24)*, which is a good reason to try to avoid multiple parents.

Sample Implementation

An aggregating attribute needs a query method that checks the parent if the appropriate field is null.

```
class Organization...
  Currency getAccountingCurrency() {
      if (accountingCurrency != null)
          return accountingCurrency;
      else {
          if (parent != null)
             return parent.getAccountingCurrency();
          else {
             Assert.unreachable();
             return null;
          }
      }
  }
```

Since the association is mandatory, it is an error for the accounting currency to return a null, so instead it raises an exception. That way should a bug sneak in, you'll get an exception closer to the pesky insect. Of course a chance of a virtual cockroach can be reduced, if not eliminated, by ensuring the setter checks for the correct values.

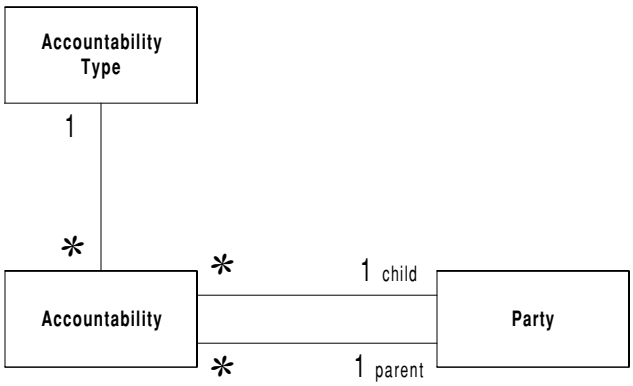
```
void setAccountingCurrency(Currency arg) {
    assertValidAccountingCurrency(arg);
    accountingCurrency = arg;
}

void assertValidAccountingCurrency(Currency arg) {
    Assert.isFalse (arg == null && getParent() == null);
}
```

Obviously, if you have a deep hierarchy with frequent access than looking up the hierarchy each time will be rather slow. So in those cases you can cache the parent's value in the children. While this alters the internal behavior it doesn't alter the external behavior, so the pattern is still present and any tests written for the slow implementation should still work for the optimized version.

Accountability

Represents a complex graph of relationships between parties



If you are dealing with an organization with a single hierarchy, or even a couple, then *Organization Hierarchy (17)* is the simplest way to deal with things. However larger organizations grow beyond this. You often find a host of different relationships between parties, all of which carry their own meaning. If your hierarchies start breeding like viagra infused rabbits, it's time to look to *Accountability (27)*.

Making it work

Accountability (27) uses a *Typed Relationship (85)* to provide the necessary flexibility. Each instance of accounability represents a link between two parties, the accountability type indicates the nature of the link. This allows you to handle any number of organizaational relationships. Create an instance of accountability type for each kind of relationship you need, and connect together the patries with accountabilities of those types (see Figure 0. 16)

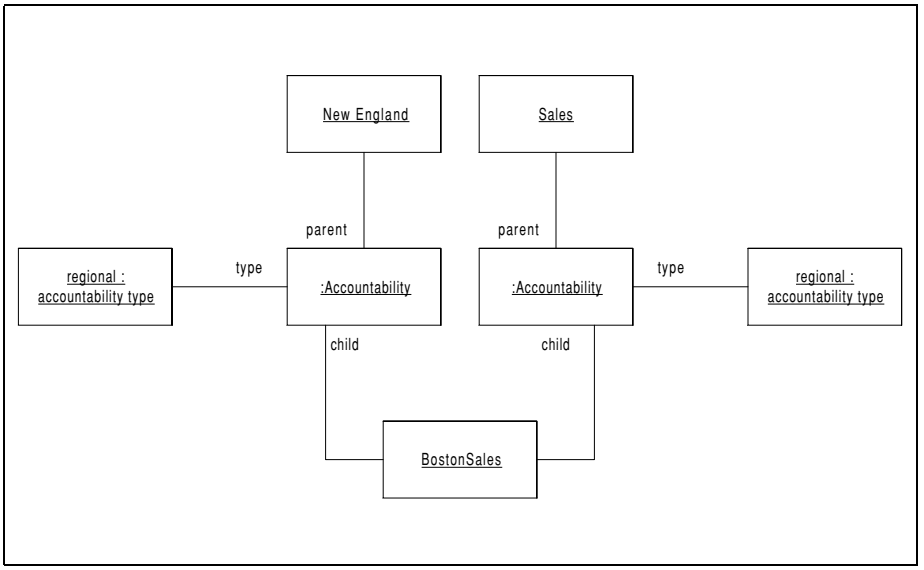


Figure 0.16 *An instance diagram showing BostonSales as a child of NewEngland under the regional sturcture and a child of Sales under the functional structure.*

You can then use accountabilities to provide similar navigational behavior to what you have with *Organization Hierarchy (17)*. The main difference is that much of this behavior is now qualified by the accountability type. So instead of saying "who are the parents of Boston Sales" you say "who are the regional parents of Boston Sales".

Accountability (27) tends to encourage more complicated variations, the most significant one is to use a knowledge level to capture rules about what kinds of parties can be connected together (See page 32).

When to use it

Use *Accountability (27)* when you need to show organizational structures but *Organization Hierarchy (17)* won't suffice. Remember that *Accountability (27)* is a good bit more complicated to use than *Organization Hierarchy (17)*, so don't use it until you need it. The main driver is that you have to record several different lines of authority. If you only need a couple, then using *Organization Hierarchy (17)* will do the trick, but if the lines increase in number, then *Accountability (27)* ends up being simpler.

It's not difficult to refactor an *Organization Hierarchy (17)* into using *Accountability (27)*. Initially you can use both patterns together on the party. The first step is to rationalize the interface so that clients can use the *Accountability (27)* interface to manipulate links held on the original hierarchies. Once you've done that you can

choose either to dropthe old interface, or to retain it for convenience. You can keep the old implementation side by side with the new for as long as it's not getting in the way. The interface is the thing to sort out first.

Sample Code

The basic implementation for accountability involves three classes: party, accountability, and accountability type. In the simple form accountability type has no interesting behavior.

```
class AccountabilityType extends mf.NamedObject {
  public AccountabilityType(String name) {
      super(name);
  }
}
```

In this sample I use bidirectional relationships to link accountability and party.

```
class Accountability {
  private Party parent;
  private Party child;
  private AccountabilityType type;

  Accountability (Party parent, Party child, AccountabilityType type) {
      this.parent = parent;
      parent.friendAddChildAccountability(this);
      this.child = child;
      child.friendAddParentAccountability(this);
      this.type = type;
  }
  Party child() {
      return child;
  }
  Party parent() {
      return parent;
  }
  AccountabilityType type() {
      return type;
  }
}
```

```
class Party extends mf.NamedObject {
  private Set parentAccountabilities = new HashSet();
  private Set childAccountabilities = new HashSet();
  public Party(String name) {
      super(name);
  }
  void friendAddChildAccountability(Accountability arg) {
      childAccountabilities.add(arg);
  }
  void friendAddParentAccountability(Accountability arg) {
      parentAccountabilities.add(arg);
  }
}
```

This is the code needed to create structures. Much of the navigation of the structure then involves code that is along the similar lines to the sample code for *Organization Hierarchy (17)*. However the code for finding parents and children is a little more complicated.

```
class Party...
  Set parents() {
      Set result = new HashSet();
      Iterator it = parentAccountabilities.iterator();
      while (it.hasNext()) {
          Accountability each = (Accountability) it.next();
          result.add(each.parent());
      }
      return result;
  }
  Set children() {
      Set result = new HashSet();
      Iterator it = childAccountabilities.iterator();
      while (it.hasNext()) {
          Accountability each = (Accountability) it.next();
          result.add(each.child());
      }
      return result;
  }
```

You then can set up and use these objects with code like this.

```
class Tester...
      AccountabilityType supervision = new AccountabilityType("Supervises");
      Party mark = new Party("mark");
      Party tom = new Party("tom");
      Party stMarys = new Party ("St Mary's");
  public void setUp() {
      new Accountability (stMarys, mark, appointment);
      new Accountability (stMarys, tom, appointment);
  }
  public void testSimple() {
      assert(stMarys.children().contains(mark));
      assert(mark.parents().contains(stMarys));
  }
```

Also you often need to carry out navigation along a single accountability type.

```
class Party...
  Set parents(AccountabilityType arg) {
      Set result = new HashSet();
      Iterator it = parentAccountabilities.iterator();
      while (it.hasNext()) {
          Accountability each = (Accountability) it.next();
          if (each.type().equals(arg)) result.add(each.parent());
      }
      return result;
  }
class Tester...
      AccountabilityType appointment = new AccountabilityType("Appointment");
  public void testParents() {
      Accountability.create(tom, mark, supervision);
      assert(mark.parents().contains(stMarys));
      assert(mark.parents(appointment).contains(stMarys));
      assertEquals(2, mark.parents().size());
      assertEquals(1, mark.parents(appointment).size());
      assertEquals(1, mark.parents(supervision).size());
      assert(mark.parents(supervision).contains(tom));
  }
```

Cycle Checking

Just as with *Organization Hierarchy (17)* we have to follow certain rules about how we connect together our parties, in particular we have to ensure that we don't create cycles in our accountability structure.

Here there is a difference to *Organization Hierarchy (17)* in that we treat accountabilities as immutable objects. As such we need to do the checking when we create the accountability. We'll follow the approach of first checking to see if we can create an accountability, and only then do we create it. We can't do this in a constructor, so we need a factory method.

```
class Accountability...
  private Accountability (Party parent, Party child, AccountabilityType type) {
      ...
  }
  static Accountability create (Party parent, Party child, AccountabilityType type) {
      if (!canCreate(parent, child, type))
          throw new IllegalArgumentException ("Invalid Accountability");
      else return new Accountability (parent, child, type);
  }
  static boolean canCreate (Party parent, Party child, AccountabilityType type) {
      if (parent.equals(child)) return false;
      if (parent.ancestorsInclude(child, type)) return false;
      return true;
  }

class Party...
  boolean ancestorsInclude(Party sample, AccountabilityType type) {
      Iterator it = parents(type).iterator();
      while (it.hasNext()) {
          Party eachParent = (Party) it.next();
          if (eachParent.equals(sample)) return true;
          if (eachParent.ancestorsInclude(sample, type)) return true;
      }
      return false;
  }
```

Not the fastest algorithm in the world, but nothing that a well placed cache couldn't solve.

This checking now allows us to prevent cycles.

```
class Tester...
  public void testCycle() {
      Accountability.create(mark, tom, supervision);
      try {
          Accountability.create(tom, mark, supervision);
          fail("created accountability with cycle");
      } catch (Exception ignore) {}
      assert(!mark.parents().contains(tom)); //just be sure!
      AccountabilityType modelMentor = new AccountabilityType();
      Accountability.create(tom, mark, modelMentor); // okay to create with different type
      assert(!mark.parents().contains(tom)); //now okay
  }
```

Using a Knowledge Level

The basic form of accountability is quite loose, allowing any combination of parties to be organized in any fashion with any accountability types. While this works well for some applications, often you want to put some constraints into place to limit the way

accountabilties can be organized. You can do this by using a *Knowledge Level (42)*. The knowledge level for accountability requires a set of party types, and sets up the accountability types to describe how the party types should be connected together.

Connection Rules

Connection rules (Figure 0.17) provide a simple yet very flexible form of knowledge level. In this form each accountability type contains a group of connection rules each of which defines one legal pairing of parent and child party types. Figure 0.18 shows an example of how you might configure one of these knowledge levels.

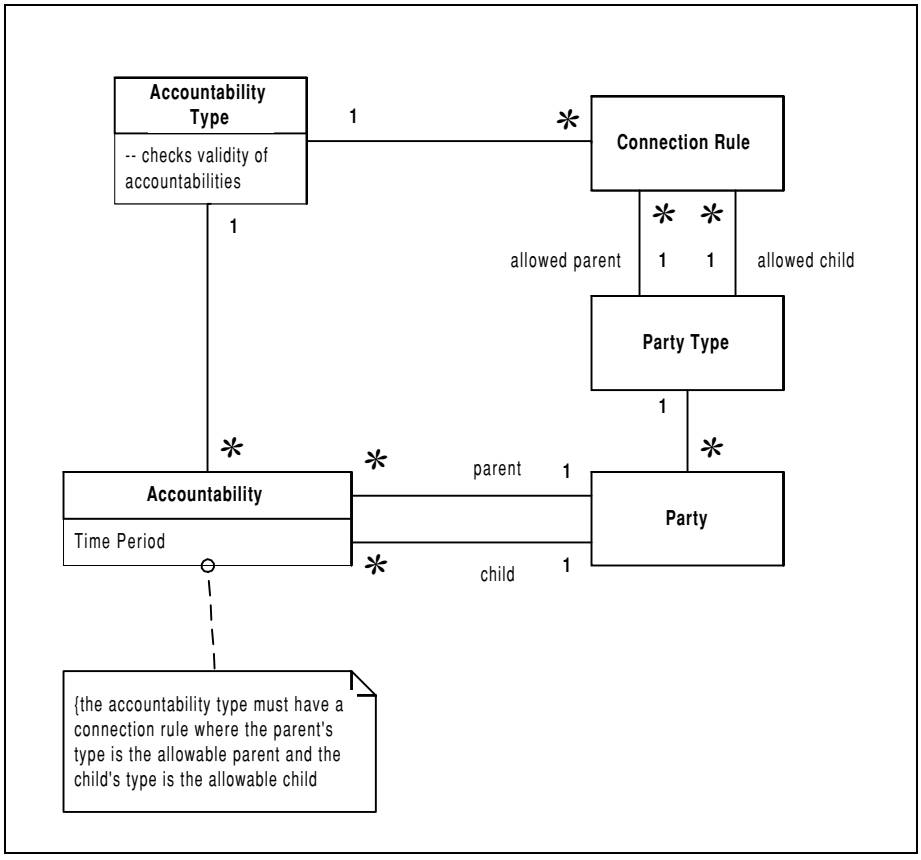


Figure 0.17 *Knowledge level for directed graphs.*

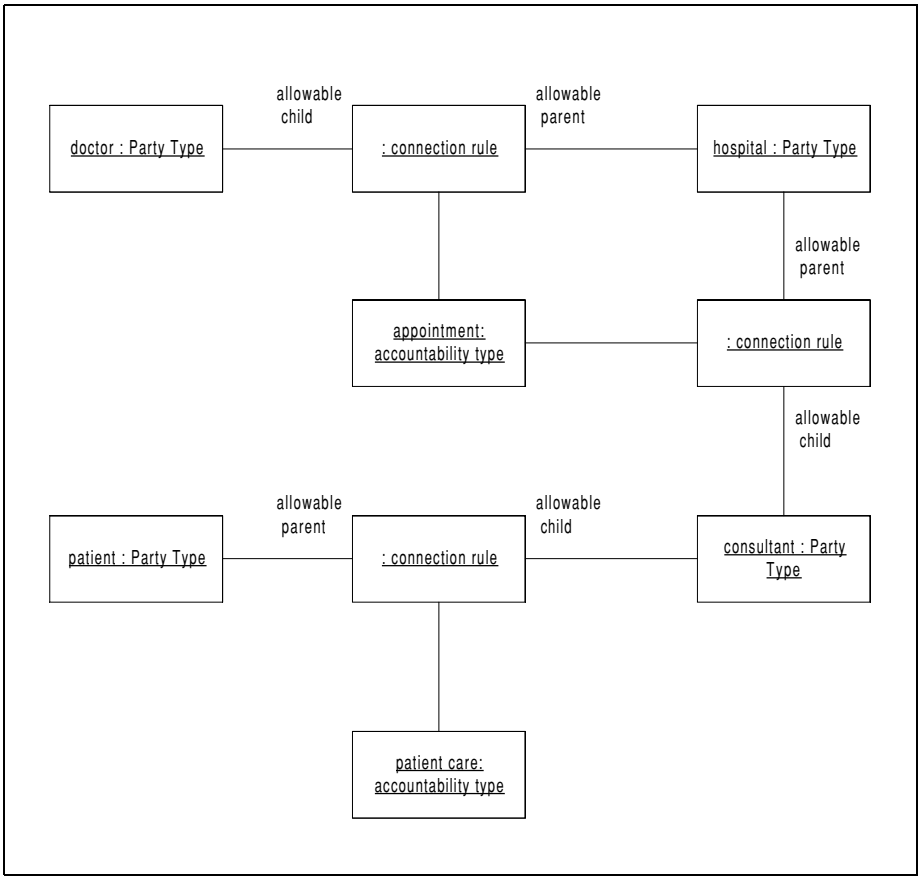


Figure 0.18 *This example shows two accountability types. The appointment accountability type allows consultants and doctors to be appointed to hospitals. The patient care accountability type records consultants' responsibility towards patients.*

You then use the knowledge level to check accountabilities. Whenever you create an accountability, the accountability uses uses its type to check to see if the accountability is valid. It is best to place the checking behavior on the accountability type because different accountability types will define different rules about connectivity. So if you want to change the rules by just replacing a single object, the accountability type is the point of replacement.

Sample Code

Here is an example that adds knowledge level connection rules to our exisiting sample code.

We need a simple structural change to party to include the party type.

```
class PartyType extends mf.NamedObject {
  public PartyType(String name) {
      super(name);
  }
}
class Party ...
  private PartyType type;
  public Party(String name, PartyType type) {
      super(name);
      this.type = type;
  }
  PartyType type() {
      return type;
  }
}
```

The rules are added to the accountability type. As we shall see there are more than one way you can set up rules like this, so I'll make a subclass to hold the connection rules.

```
class ConnectionAccountabilityType extends AccountabilityType ...
  Set connectionRules = new HashSet();
  public ConnectionAccountabilityType(String name) {
      super(name);
  }
  void addConnectionRule (PartyType parent, PartyType child) {
      connectionRules.add(new ConnectionRule(parent, child));
  }
```

Structurally, the connection rule just holds its allowable parent and child types.

```
class ConnectionRule {
  PartyType allowedParent;
  PartyType allowedChild;
  public ConnectionRule(PartyType parent, PartyType child) {
      this.allowedChild = child;
      this.allowedParent = parent;
  }
}
```

We can then set up an accountability type with connection rules.

```
class Tester...
  private PartyType hospital = new PartyType("Hospital");
  private PartyType doctor = new PartyType("Doctor");
  private PartyType patient = new PartyType("Patient");
  private PartyType consultant = new PartyType("Consultant");
  private ConnectionAccountabilityType appointment
      = new ConnectionAccountabilityType("Appointment");
  private ConnectionAccountabilityType supervision
      = new ConnectionAccountabilityType("Supervises");

  public void setUp()...
      appointment.addConnectionRule(hospital, doctor);
      appointment.addConnectionRule(hospital, consultant);
      supervision.addConnectionRule(doctor, doctor);
      supervision.addConnectionRule(consultant, doctor);
      supervision.addConnectionRule(consultant, consultant);

      mark = new Party("mark", consultant);
      tom = new Party("tom", consultant);
      stMarys = new Party ("St Mary's", hospital);
```

With the structure out of the way, now let's look at the validation behavior. The accountability already has behavior for checking basic cycle rules. With the knowledge level in place we can add to this behavior to request the accountability type to check for the correct party types (Figure 0.19).

```
class Accountability...
  static boolean canCreate (Party parent, Party child, AccountabilityType type) {
      if (parent.equals(child)) return false;
      if (parent.ancestorsInclude(child, type)) return false;
      return type.canCreateAccountability(parent, child);
  }
```

The accountability type can now check the parent and the child through the connection rules.

```
class AccountabilityType...
  boolean canCreateAccountability(Party parent, Party child) {
      return areValidPartyTypes(parent, child);
  }

class ConnectionRuleAccountabilityType...
  protected boolean areValidPartyTypes(Party parent, Party child) {
      Iterator it = connectionRules.iterator();
      while (it.hasNext()) {
          ConnectionRule rule = (ConnectionRule) it.next();
          if (rule.isValid(parent, child)) return true;
      }
      return false;
  }

class ConnectionRule...
  boolean isValid (Party parent, Party child) {
      return (parent.type().equals(allowedParent) &&
          child.type().equals(allowedChild));
  }
```

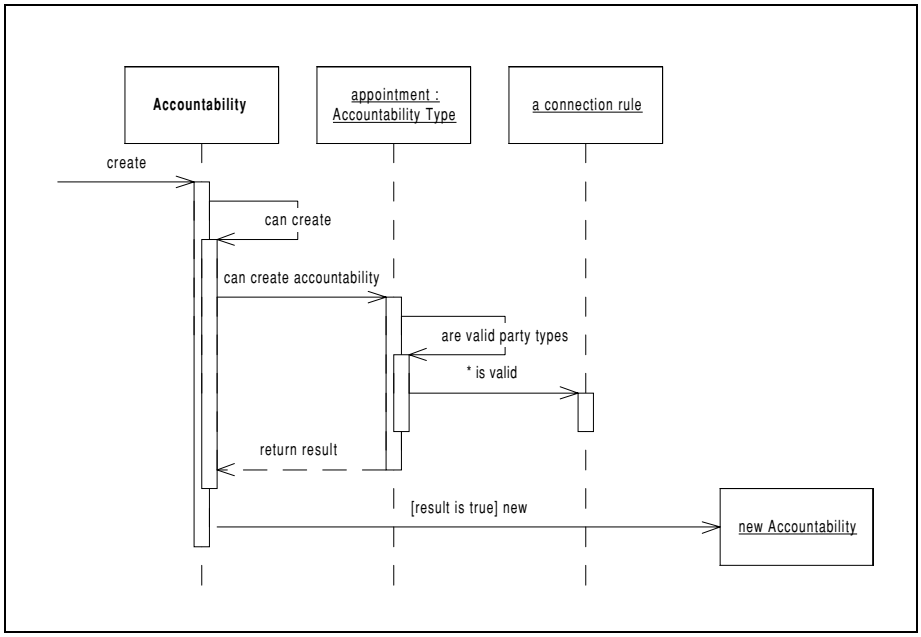


Figure 0.19 *Interactions for checking an accountability with the knowledge level*

With this in place any attempt to create an accountability without a connection rule should fail.

```
class Tester...
  public void testNoConnectionRule() {
      try {
          Accountability.create(mark, stMarys, appointment);
          fail("created accountability without connection rule");
      } catch (Exception ignore) {}
      assert(!stMarys.parents().contains(mark)); // am I paranoid?
  }
```

Hierarchic Accountability Type

When you're using accountabilties, you may well find that some of your accountability types need to maintain a hierarchy, while others don't need that. You can enforce hierarchic rules selectively within the validation method of the accountability type.

Sample Code

For the code sample I do this with a flag on the accountability type.

```
class AccountabilityType ...
  private boolean isHierarchic = false;
  void beHierarchic() {
      isHierarchic = true;
  }
  boolean canCreateAccountability(Party parent, Party child) {
      if (isHierarchic && child.parents(this).size() != 0) return false;
      return areValidPartyTypes(parent, child);
  }
}
```

Levelled Accountability Type

Connection rules are the most common way of representing the rules for an accountability type. However from time to time you run into situations where there is a strict set of of levels of party types. An example of this is a regional breakdown where national parties have children that are states, which have children that are counties, that have children that have cities.

You can represent this with connection rules, but a simpler alternative is to list the levels on the accountability type.

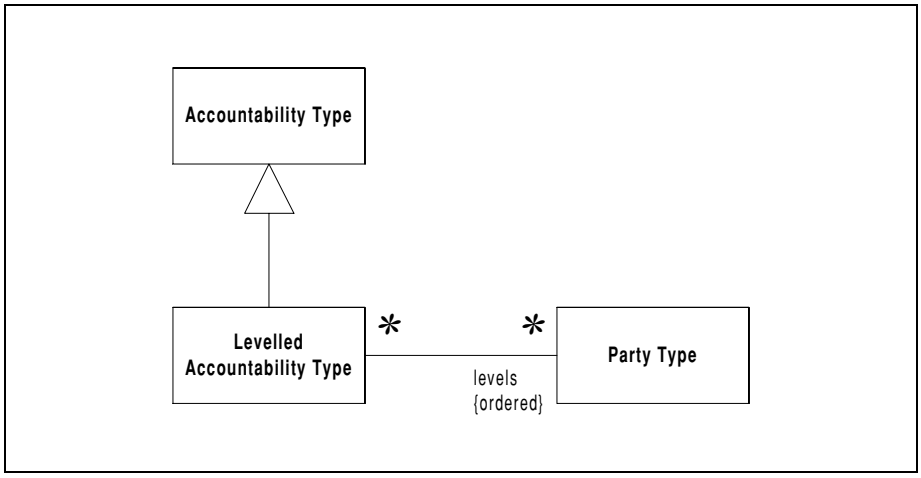


Figure 0.20 *Levelled Accountability types*

Sample Code

Setting up levels needs a different test fixture.

```
public class LevelledTester extends junit.framework.TestCase {
  private PartyType nation = new PartyType("nation");
  private PartyType state = new PartyType("state");
  private PartyType county = new PartyType("county");
  private PartyType city = new PartyType("city");
  private Party usa, ma, nh, middlesex, melrose;
  private LevelledAccountabilityType region = new LevelledAccountabilityType();

  public LevelledTester(String name) {
      super(name);
  }
  public void setUp() {
      PartyType[] levels = {nation, state, county, city};
      usa = new Party("usa", nation);
      ma = new Party("ma", state);
      nh = new Party("nh", state);
      middlesex = new Party("usa", county);
      melrose = new Party("usa", city);
      region.setLevels(levels);
      Accountability.create(usa, ma, region);
      Accountability.create(usa, nh, region);
      Accountability.create(ma, middlesex, region);
      Accountability.create(middlesex, melrose, region);
  }
  public void testLevels() {
      assert(melrose.ancestorsInclude(ma, region));
  }
  public void testReversedLevels() {
      try {
          Accountability.create(ma, usa, region);
          fail();
      } catch (Exception ignore) {}
  }
  public void testSameLevels() {
      try {
          Accountability.create(ma, nh, region);
          fail();
      } catch (Exception ignore) {}
  }
  public void testSkipLevels() {
      try {
          Accountability.create(ma, melrose, region);
          fail();
      } catch (Exception ignore) {}
  }
}
```

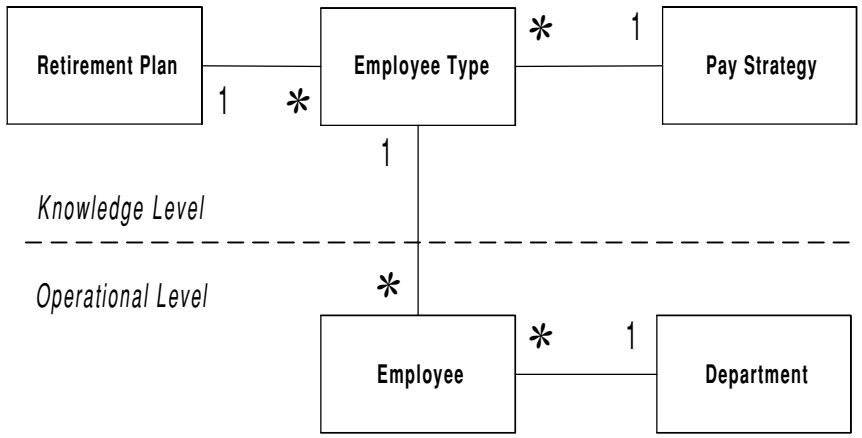
However the behavior of the levelled accountability type is quite simple. We need to be able to set the list of levels and to check for valid party types from the exisiting method in accountability type.

```
class LevelledAccountabilityType extends AccountabilityType {
  private PartyType[] levels;
  public LevelledAccountabilityType(String name) {
      super(name);
  }
  void setLevels(PartyType[] arg) {
      levels = arg;
  }
  protected boolean areValidPartyTypes(Party parent, Party child) {
      for (int i=0; i<levels.length; i++) {
          if (parent.type().equals(levels[i]))
             return (child.type().equals(levels[i+1]));
      }
      return false;
  }
}
```

This sample does not allow you to skip levels, i.e. you can't have a city as a child of state. If you want to allow level skipping it's easy to amend `areValidPartyTypes` to support this (an exercise for the reader). However if you want to skip some levels but not others then you're better off with connection rules.

Knowledge Level

A group of objects that describe how another group of objects should behave (also known as a meta level)



Knowledge Level (42) is one of those abstract patterns that is difficult to describe on it's own, although one that you soon recognize in more complex object systems. You have a *Knowledge Level (42)* when you have a group of objects whose instances affect the behavior of a group of operational objects, typically allowing you to alter a system "without programming" but instead by creating and wiring together some of these knowledge objects.

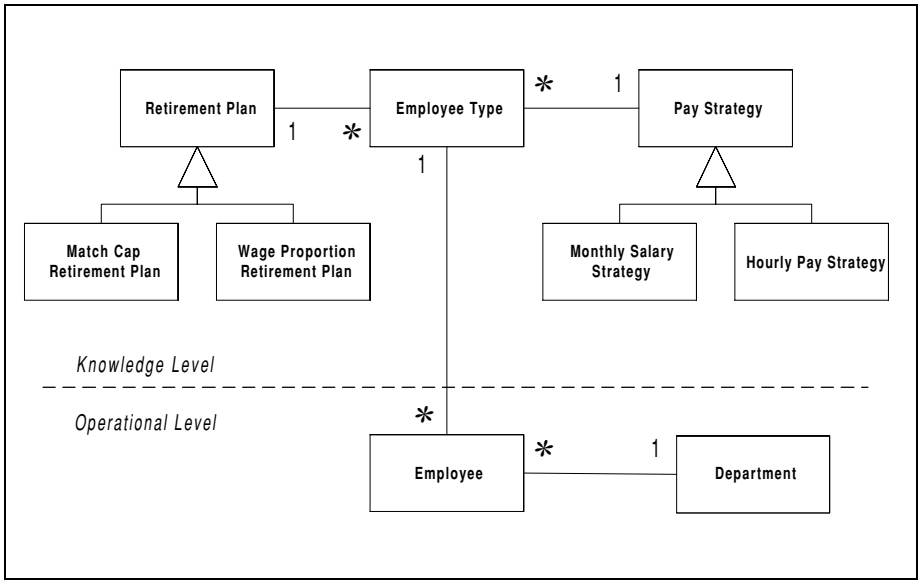


Figure 0.21 *An example knowledge level using employees*

Figure 0.21 shows an example of using a knowledge level for employees. The intention here is that employee types are created with common combinations of payment and retirement policies. Employee's are then marked with an employee type to indicate which policies they use. If you need to customize how employees work you would create a new instance of Employee Type connected to new instances of payment strategy and retirement plan. This choosing and wiring together of objects is how the system is configured.

Terminology

In this book, as with the previous edition, I use the term *Knowledge Level (42)*. However I can't claim that this is a completly standard term in design circles. There isn't such a standard term as yet, which is why I stick with knowledge level.

I started using the terms operational and knowledge levels while working on the Cosmos project for the UK National Health Service. In this project I worked on a joint team with doctors and nurses. We coined the terms together because they seemed to fit the way we looked at the model. The objects in the operational level represented the day to day operational behavior of the clinicians, while the knowledge level represented their clinical knowledge.

Another term you commonly come across is *meta* as in meta-data. Meta is greek for 'about' so meta-data means 'data about data'. You often hear people talk about meta-

objects, or meta-levels. If I had to do a poll on terms for this concept using meta something would probably top the list. However I still prefer knowledge level as I think it means more to non-software people.

Another term is 'Active Object Model'. The rationale behind this is that the knowledge objects take an active part in determining the behavior for the operational objects. However the term isn't that widely used, and I don't like the connotation that the operational level is somehow passive.

Another personal convention is my habit of drawing the knowledge level at the top of the diagram and the operational level at the bottom often, but not always, separated by a dashed line. Again this is a Cosmos habit which I've stuck with as I find it useful. However it's not part of any standard, I guess my influence in the world is more limited than I might like!

It's a common convention to make the connection between the knowledge and oeprational level through associations between a 'thing' and a 'thing type'. This has become so prevalent that I would now hesitate before using any other naming scheme. The thing type is often referred to as the 'type object' of the thing, following the type object pattern.

Making it work

Knowledge levels are pretty complicated things to put together, and as such people usually look upon them as an advanced OO technique. The difficulty lies in choosing a set of objects that really does allow you to handle most changes without touching much, or any of the source code. It's often hard to see what combinations will work in advance, so like with most complicated frameworks, the design needs to evolve.

It's not possible to say how to do this in the abstract. Instead you need to look at specific patterns that involve knowledge levels and see what you can learn from them. It may be that you can use that specific pattern. Or it may be that looking at knowledge levels will help you with a knowledge level for your circumstances.

When to use it

When people start using, or considering to use, a knowledge level; the common hope is that this will allow non-programmers to change the system behavior 'without programming'. While this may occur from time to time, in my experience it is an eldorado. Understanding how the knowledge works and how to configure it is still a pretty complicated exercise, not one suited for casual programmers. Indeed often experienced programmers find knowledge levels hard to work with.

In my view the benefit of a knowledge level is that it allows people who are experienced with the design to make changes much more quickly than they would otherwise be able to do.

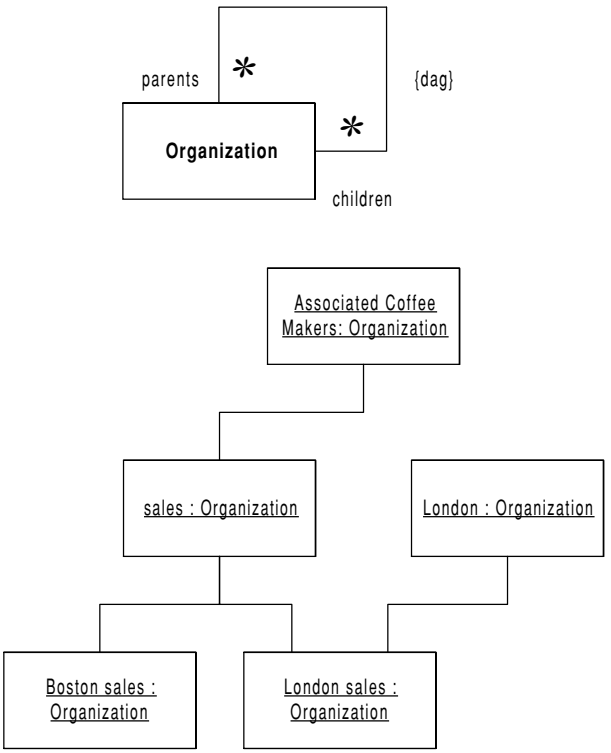
The fact that many of these changes can be made at run time is also appealing. However this also comes with warnings. Just because you can change the system 'without programming' doesn't mean you can change it without testing. Indeed testing can be more critical for knowledge levels and often harder to do. Knowledge levels are there because what they capture is very dynamic, so testing becomes quite an exercise.

Also the fact that changes are done 'without programming' means that you usually have to do your changes without tools. Debugging does not go away with knowledge levels, nor does configuration control. If your 'non-programming' environment doesn't provide debugging, configuration, and testing tools (as it won't) then you'll either have to do without or roll your own. I often see people building graphical editors for knowledge levels, but actually editing is rarely the hardest part of 'non-programming'.

All of these are serious caveats for using a knowledge level, if they scare you a bit, then that's the point. This is not a pattern to be used lightly. However in a complex business system there always seem to be a few spots where knowledge levels are worth their cost. When you have a large amount of volatile rules, then a knowledge level can make a lot of sense. We don't really have enough of a handle yet on what the guidelines are for using them, but we do know know that they should be used rarely, but when you need them they are essential. Rather like parachutes.

Object Graph

A group of objects connected together in a graph structure.



Object graphs are a common way of representing situations where many objects of the same fundamental type can be connected in a recursive structure. You see them in organizational structures, work breakdown structures, product structures... the list goes on and on.

When talking about object graphs in the abstract, you usually talk about nodes and links. So in the sketch organizations are the nodes, and the instances of the parents/children association are the links.

However I find it hard to talk sensibly about object graphs in the abstract. So I've included this pattern in the organizational structures chapter because organizational structures provide a good example of how to use *Object Graph (46)* in practice. For this pattern I'm really just going to coin some terms that you may find useful in talking about object graphs when you run into them, and to point out the abstract nature of the ideas.

Most graph structures you come across in practice are acyclic. That means that if you look at your ancestors you won't find yourself.

Variations

There are a number of common variations on the object graph theme

Object Hierarchies have only one parent to each node. Although hierarchies are more restricitive they do permit some useful behaviors based onthe fact that there is only one parent, such as *Aggregating Attribute (24)*. *Organization Hierarchy (17)* is an example of an object hierarchy. Structures that permit multiple parents are often referred to lattices, networks, or the intimidating (but mathematically correct) directed acyclic graph (or dag for short).

The links in a structure may be represented through an association or through a separate association object. *Organization Hierarchy (17)* just uses the association while *Accountability (27)* uses an association object. In *Accountability (27)* the accountability class is the association object. Often when you find association objects you'll find the association is typed, allowing you to define multiple graphs over the same set of nodes, as in the multiple organizing structures discussed in *Accountability (27)*.

You may find the leaf and non-leaf nodes broken out into separate subtypes as in Figure 0.22. You should do this if there is significant differences in the behavior for the leaf and non-leaf cases. However it is a good idea to try to put as much of interface up on the parent as possible, even when it doesn't look like it makes sense.

An example of this would be a method `numberOfChildren`. On first sight it seem that such a method should be on organization as only organizations have children. However it is useful to define the interface on party and implement it in person to return 0. If you do that you'll find that a method such as `numberOfDescendents` is much easier to write.

Patterns afficionados will recognise Figure 0.22 as the classic shape of the composite pattern. Indeed composite usually applies to hierarchic object graphs.

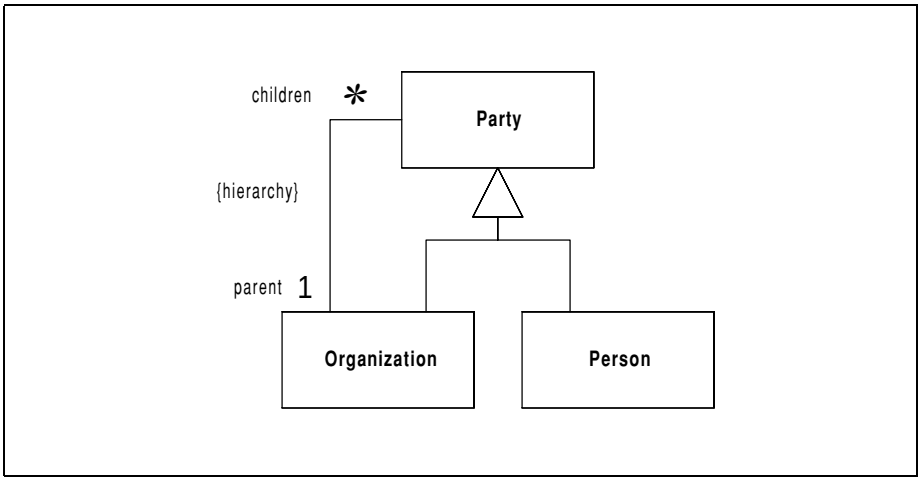


Figure 0.22 *Breaking out the leaf and non leaf nodes in an object graph.*

Making it work

I've discussed the issues in making *Object Graph (46)* work in the organizational patterns: in particular *Organization Hierarchy (17)* and *Accountability (27)*. Much of what you need for any use of *Object Graph (46)* you can find in there.

One point worth reiterating is the way to name the links. These days I always try to use parent/child, even if that naming does not fit the domain very well. Familial relationships are a powerful metaphor for these kinds of relationships: if I say the marketing department is my grand-uncle it sounds silly, but you know exactly what the relationship is.

There is a huge body of academic work on representing graph structures in computer programs. Most of this is far more than you would ever need to know, but it repeatidly suprises me how many people don't know about this work. If you're having difficultiy doing something with a graph structure, dig out some abtruse work on graph theory, algorithms, or data structures — you may well find an answer to your problem.

When to use it

The choice of when to use the structure is usually better discussed with respect to the particular applications of the pattern in particular domains. So it's more useful to say "when should I use *Organization Hierarchy (17)* or *Accountability (27)*" than to ask "when should I use *Object Graph (46)* and if so what kind?"

However reading the narrative and discussions around *Organization Hierarchy (17)* and *Accountability (27)* will give you some useful guidelines. In particular bear in mind

that using association objects is a good bit more complicated than a simple association, so only go that far if you need to.