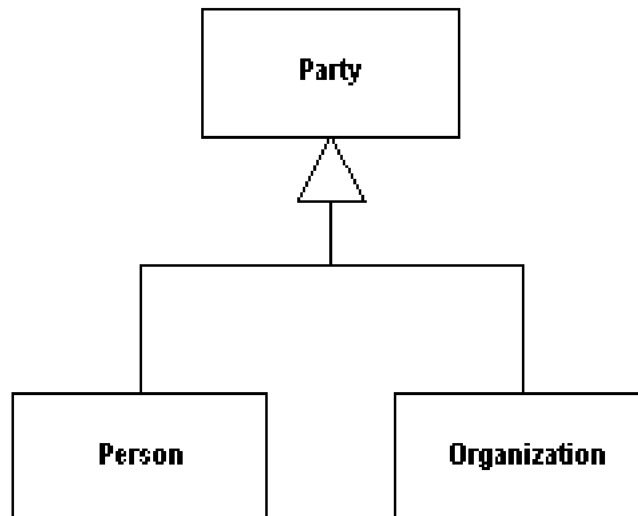
Party

<< 人と組織単位の抽象化の図>>



例　：　通信事業者の顧客は、個人または、企業である。
それらは、パーティであり、時にはサブタイプとして扱われる。

皆さんのアドレス帳をご覧いただきたい。何が見えるだろうか？　私のものと同じようなら、住所や電話番号、それに奇妙なメールアドレスがたくさんある。それらは、何かに結びつけられているだろう。これらの目的は人だが、よく見ると会社も見つかる。筆者はよく「オークグローブタクシー会社」に電話するが、その会社で特に私が話しをしたいと思う相手はいない。ただ、タクシーをよこしてほしいだけなのである。

私が、アドレス帳をモデル化しようとするならば、郵便番号と電話番号を持っている人や、会社を選ぶだろう。しかしその結果生じる重複は、筆者の目にはつらいものである。従って、筆者は、人と会社の汎化をしようと考える。こうしたことは、誰もが知っていて使ってはいるが、誰も名

づけていない概念－名前のない概念の古典的なケースである。私は、人／組織、プレーヤ、法人といったさまざまな名前の数え切れないほどのデータモデルでこうしたことをみてきた。

私は、「パーティ」という言葉が気に入っている。ここ最近の数年間の間に「パーティ」がかなり一般的に使われるようになってきたことを嬉しく思う。

Making it work

私は通常、パーティを人や組織のスーパータイプとして定義する。そうすると、会社の部署や非c なチームの住所や電話番号も含められるようになる。

パーティ上に人や組織単位に共通であるどんな振る舞いであっても配置する。そして、サブタイプ上にその振る舞いのひとつもしくは、その他の振る舞いを配置することにしてみる。

サブタイプ上に振る舞いが、スーパータイプ上でも意味を成すかどうかについて考えてみると、ほとんどの場合、スーパータイプ上でうまくマッチしていることに驚くだろう。

When to use it

明らかに Party を使用する場合は、モデルの中に人と組織が存在し、その中で共通の振る舞いを見つけられる時である。しかし人と組織の区別を必要としない場合、さらにこのケースも考慮すべきである。このような場合では、常にパーティクラスを定義するだけにし、サブタイプは与えない方がよい。

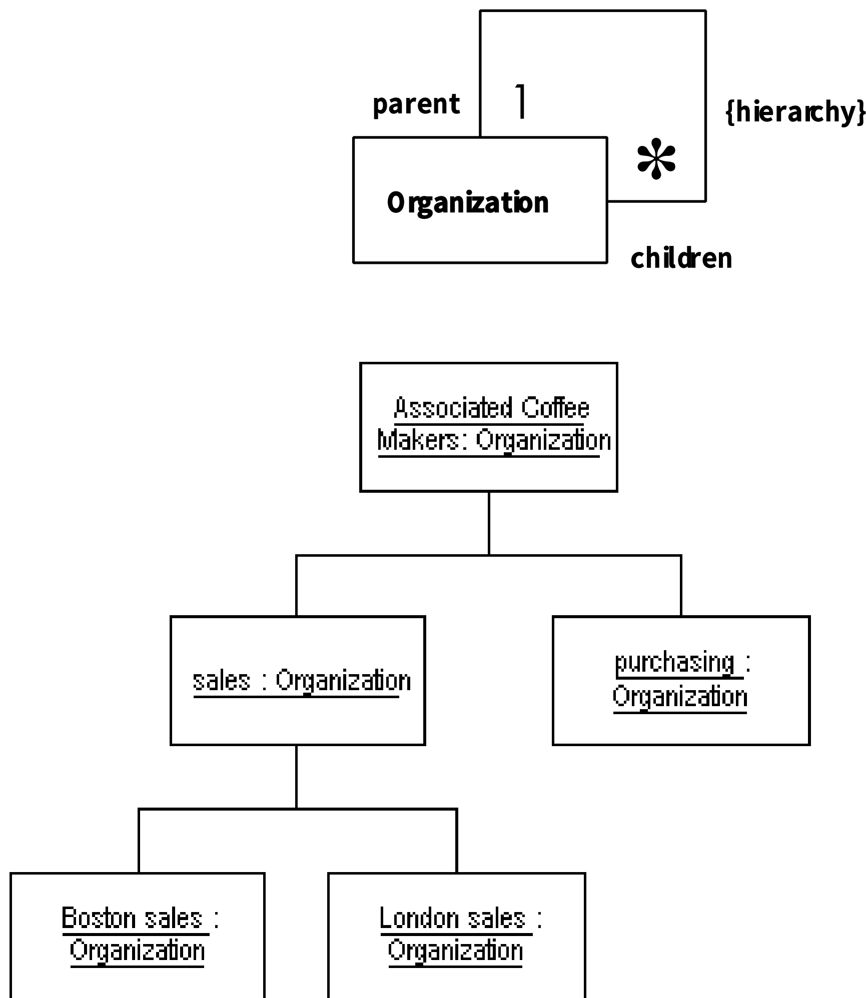
パーティが企業内において異なるロールを果たしている時は、その行為の中にロールパターンが含まれている。特に Role Object（★）を使うことは助けになるだろう。それは、パーティとの共通パターンであり、しかも盲信するものではないが、この問題に対するより多くの議論のため

に Role Object（★）を見ていただきたい。

パターンとは、共通の振る舞いがあるかどうかを見つけ、またそのスーパータイプとしてパーティを使っているのかどうかを見つけ出すことである。最近よくパーティが使われるようになったので、パーティを使うことでコミュニケーションがうまくとれるようになってきたように感じる。

Organization Hierarchy

<<企業内の組織単位の階層を表現した図>>



Making it work

階層は、組織の中の共通構造である。これは、階層が、人間が複雑さに対処するために自然で共通のテクニックを表現しているからである。階層をモデル化することは、物事を共通にすることである。それは容易でもあり複雑でもある。

階層のモデル化が容易なのは、それに再帰的関連を見出せるからである。以前、私がデータモデ

ルを作成した際にその時指導して下さった先生は、「豚の耳」と呼んだことさえある。しかし、再帰的関連は、全体像を伝えられない。多くの場合、ひとつの親がいるが、そのひとつが階層のトップでありえるだろうか？　階層には、さらに循環（あなたの親がその孫でありえない）を持てないというルールがある。ＵＭＬでは、これに対するための規定がないので、私は何が起こっている（進行している）のかを示すために、関連付けとして強制的に{hierarchy}を用いる。

しかしながら、強制的に用いる{hierarchy}でさえ、厳密には曖昧である。だから、{hierarchy}は、木(１つのトップ)と森(複数のトップ)の差を伝えられないのである。モデルにおいて木と森の差が何なのかを述べることが重要な場合があったとしても、ほとんどの場合それはあまり重要視されていないのである。

ダイアグラムでは、親及び子という用語を使用する。これら、関連ロールの名前は、扱いにくい。親は、大抵の会社でよく機能するが、子は親ほどではない。けれども、子は最良の名前であると考えている。階層や他の種類のダイレクトグラフ構造について話す場合は、親と子供の使用は非常に有用なメタファである。「ロンドンでの売上とジャワでの仕入れは、いとこのようなものだ。」という言い回しは、風変わりに聞こえるが、私が何を言いたいかを容易に判断できる。メタファは、大抵、少し奇妙に聞こえるのだが、私達にとっては有用な言葉である。

When to use it

ソフトウェアに影響するような階層的な会社構造の場合、このパターンを使用するべきである。

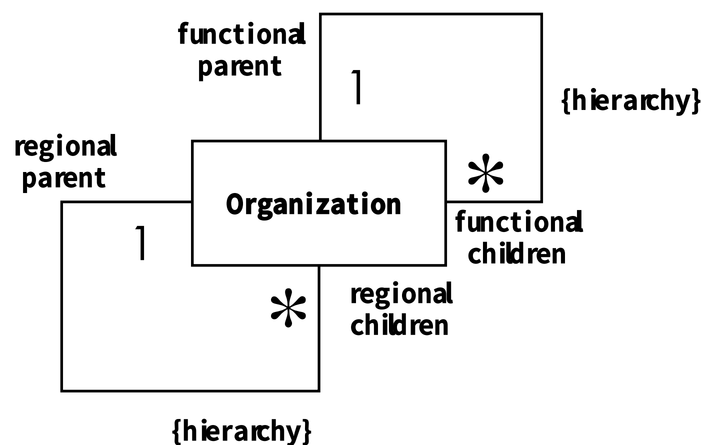
階層的　：　階層を含んだパターンならば、さほど複雑にはならないだろう。要求がより複雑な場合、パターン(いくつかの提案が下にある)をひねるか、あるいは Accounttability（★）を代用

できる。

ソフトウェアに影響する ： あなたのしていることが影響する場合、単純に会社組織を認識する必要があるだろう。階層を認識しないのならば何が起こるかを自問するべきだ。結果が思わしくない場合、それは、結合の手間(とりわけ結合を維持する手間)をかけるべきではない。パーティや組織のリストに、このパターンを後から加えることは簡単である。

GOF ファンは、このパターンが GOF の Composite パターンによるアプリケーションであることに注目するだろう。しかし、Composite クラスおよび Leaf クラスが離れないように、多少変えている。確かに、それを実装する際に Composite パターンの使用を考慮すべきであり、その際に Composite クラスおよび Leaf クラス間の振る舞いの区別をしないことが極めて一般的である。

単一の階層関連に制限する必要はない。組織が地域と機能的な管理について異なる構造の場合、図 0.13 のような対になる階層を使う。これは、その機能的な親としてロンドンオフィスの販売部門とその地域的な親としてのロンドンオフィスを意味する。もちろん多くの階層になるとともに、混乱するだろう。ここでは Accountability（★）を使用すべきである。



≪図 0.13 対になる階層を使用している≫

類似したロジックは、より一般的なグラフ構造によって複数の親を持つために、階層を必要としない。ここではより慎重になり、違う種類の親が実際にいるかどうかを自問するべきだ。多くの場合、複数の階層あるいはAccountability（★）がより役立つことだろう。とりわけAggregating Attribute（★）が使えないときには、一度ひとつの親をなくすと、階層は簡単に集約できなくなる。

Sample Implementation

このパターンの実装で大切なのはクラス上の正しい種類のインタフェースを得ることだ。それは、必要な構造へと導くための正しい操作を作成することを表す。このサンプルコードでは、組織のためにRegistry（★）として静的変数を使用している。

<ソースコード>

```
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);
    }
```

肝腎なのは、親および子を保つために操作が必要になってくることだ。この場合、親を格納するフィールドを用いたり、登録から一目で子を決定したりする。

<ソースコード>

```
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 = (Oraganization) it.next();
                if (org.getParent() != null)
                   if (org.getPrent().equals(this)) result.add(org);
         }
         return result;
    }
```

その後、これらのメソッドで、さらに構造を操作する他のメソッドを加えることができる。いくつかの例をあげる。

<ソースコード>

```
public Set getAncestors() {
       Set result = new HashSet();
       if (parent != null) {
                   result.add(parent);
                   result.add(parent.getAncestors99;
       }
       return result;
}

public Set getDecendents() {
       Set result =new HashSet();
       result.addAll(getChildren());
       Iterator it = getChildren().iterator();
       While (it.hasNext()) {
                    Organization each = (Organization) it.next();
                    result.addAll(each.getDecendents());
       }
```

```
        return result;
}

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

周知のメタファをどのように使うかということは、メソッド名がこれらのクエリによって返されるものと明白に関連することに注目すべきだ。ただこれらのクエリは操作に直接、役立つものではない。このことは階層に循環を含まないことを保証する。循環は以下のコードをチェックすることを妨げる。

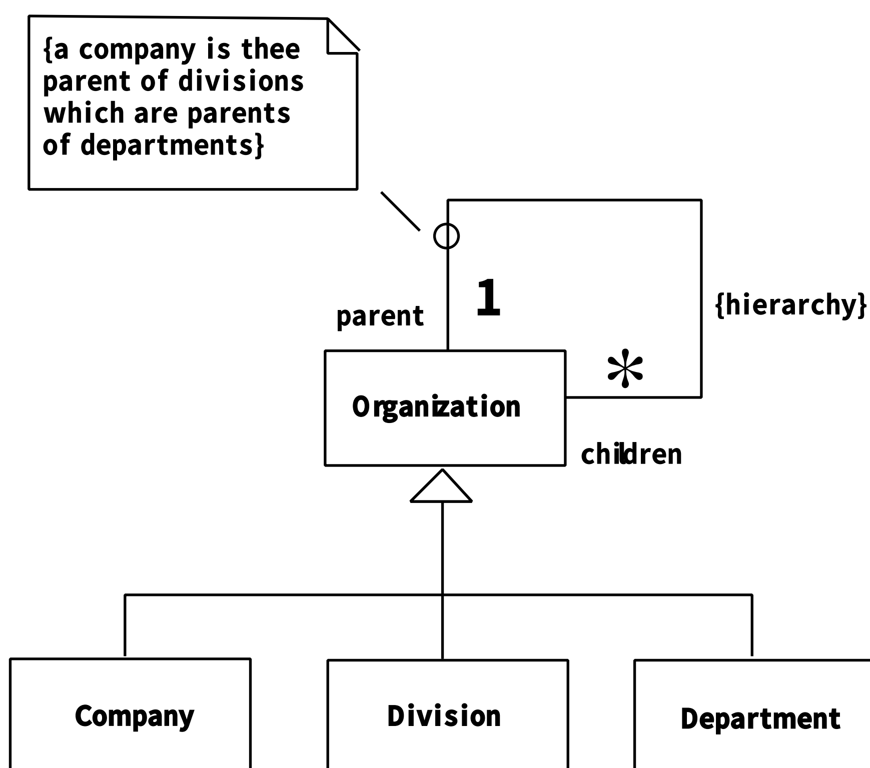
<ソースコード>

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

生成時よりも修正時の検査だけを必要とすることに気づいてほしい。さらに親をセットし、確実だと思っていても、それをリリースする前にテストしてもらいたい。

Variations:Subtype for Levels

ときには、階層に様々なレベル間の変化を見つけるかもしれない。したがって、企業は、部門ごとに分割されるかもしれない。３つのレベルの組織に特定の振る舞いがない場合、単一の組織クラスで目的に適うことだろう。しかしながら、変化がある場合、図 0.14 ではサブタイプを形成することが有益かもしれない。



<<図 0.14 様々なレベルのサブタイプを使用している>>

この場合、スーパータイプ上で関連を置くか、あるいはサブタイプ上にいくつかの関連を置くか、選択することができる。その決定はどのような操作をするかによるだろう。組織上でのよく理解された操作はよりよい結果につながるだろう。本当に重要ならサブタイプ上に関連を置いてほしい。もし両方で必要ならば、どちらにもインタフェースを提供することができる。そうすることで、多くの場合、重なった振る舞いを削除できるとともに、より洗練された組織上で、単一の関連を見つけられる。親の割り当てでは、制限を強化できる。もし、組織のグループ間に共通の振る舞いがない場合、組織クラスは完全に捨てられる。けれども、私はこれを決して行わない。通常、いくつかの重なった振る舞いが親のどこかにあるものだ。

Sample Implementation

親のチェックを行う唯一の方法は、実行時におけるチェックコードを使うことで親をオーバーライドすることだ。

<ソースコード>

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

もちろん、これは実行時におけるチェックだけであるので、コンパイル時のチェックの方がよいかもしれない。コンストラクタのオーバーロードで部分的にはそれが可能である。

<ソースコード>

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

ここでの修正をチェックすることはより困難だろう。それを直すためには、組織の修正を削除し、サブタイプ上でだけ適切な修正を行わなければならない。

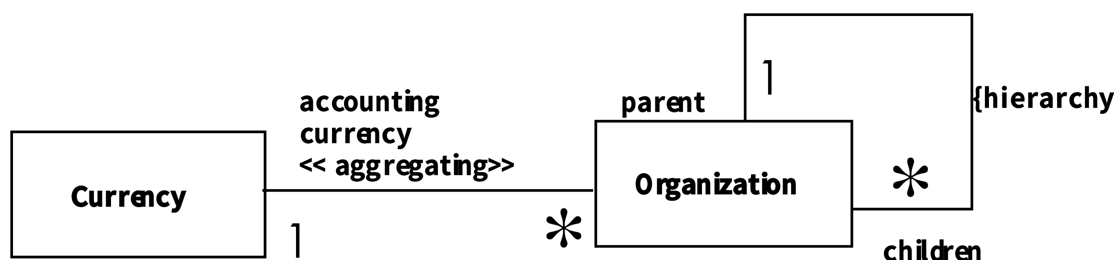
Aggregating Attribute

親から子にその属性値を使わせる

多くの場合、組織の属性は親から子へ継承していることに気づくだろう。実際、上部の会社、営業部門、ボストン営業支部と、すべてで US ドルが会計上の通貨として利用されている。決してこれは偶然ではなく、事実、子は特に指定しなくても、親で使われている会計上の通貨を利用できる。

Making it work

これをモデル化するのはとても難しい。細かく見ると、すべての組織は通貨を持つ。すなわち多重度は１対 1 である。しかし実装上、通貨は任意に指定されている。必要なフィールドが null である時は、その親を見るべきと示している。私は図 0.15 で、関連の集約にステレオタイプを使ってモデル化した。また、これは標準的な UML では対応できないので、少々の修正は必要であろう。この関連の集約または属性の考えは、コンポジットを使う場合はいつでも採用できるため、手軽な拡張といえる。



<<図 0.15 Aggregation association for currency >>

When to use it

私はこのパターンが、階層構造のどんな形態でも自然に表現できることに気がついた。疑問を解く鍵は、親の値の変化が、すべての子に影響を与えることである。もしすべての子の間で親と属性が違わないなら、それはまさに Aggregating Attribute（★）である。(親がデフォルトを提供し、そのデフォルトは子に継承され、そして親の値で子を変化させない場合は、Aggregating Attribute （★）とはいえない。)

もちろんこれはひとつの親である場合にだけしかなしえない。もし複数階層だった場合もしくは Accountability（★）を使っている場合は、より複雑になる。これらの状況では、振る舞いの集約のためにどちらの親が基本として使われているか、はっきりと指し示す必要がある。そして扱っているものが一般的なグラフ構造ではなく、階層構造であることを確認してほしい。もし、より一般的なグラフ構造が本当に複数の親を必要とするならば、Aggregating Attribute（★）を使うことができない。なぜなら複数の親が存在するからである。

Sample Implementation

集約された属性は、適切なフィールドが null かどうか問い合わせるためのクエリメソッドを必要とする。

<ソースコード>

```
class organization
 Currency getAccountingCurrencey().{
      If (accountingCurrency != null)
```

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

通貨を指定するのは必須となっているが、取得しようとしたときに null が返ってきてしまった場合はエラーになる。それを防ぐために例外を起こす。それは例外を見つけることで、入り込んでしまったバグを炙り出すようなものである。もちろんバグを減らすことができ、正しい値をチェックしセットする事もできるであろう。

<ソースコード>

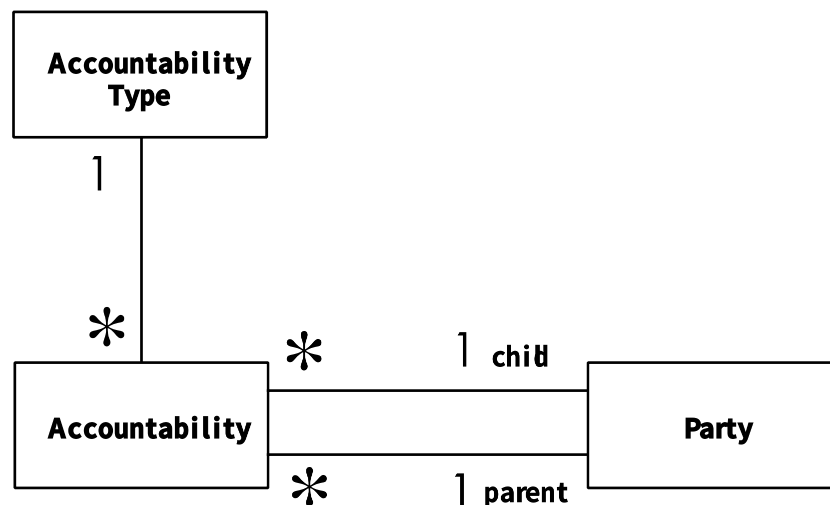
```
void setAccountingCurrency(Currency arg) {
    asssertValodAccountingCurrency(arg);
    accountingCurrency = arg;
}


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

もし参照が深い階層ならば、オブジェクトをたどっていくのに遅くなってしまうことは明確である。そんな場合は、親の値を子にキャッシュさせることができる。この方法は、内部的な振る舞いは変わってしまうが外部的なものには影響がない。よって、パターンは残り、重い実装のために書かれたテストも理想的な形として動作するだろう。

Accountability

パーティ間の関係の複雑なグラフを表現する。

<<Accountability の図>>

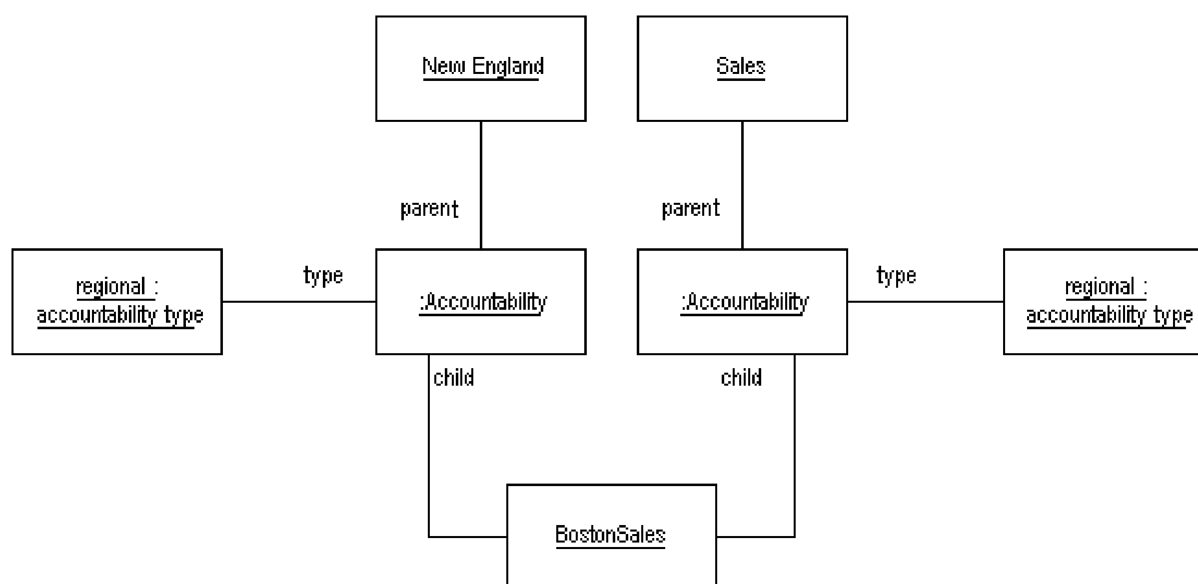


１つか２つの階層からなる組織を取り扱うなら、Organization Hierarchy（★）が最も単純である。しかし、より大きな組織ではこれでは手にあまる。パーティ間の異なった、それぞれ固有の意味を持つ関連のグループがしばしば見られる。もしあなたの階層がバイアグラを注入されたウサギのように繁殖を始めたなら、Accountability（★）に注視する時だ。

Making it work

Accountability（★）は必要な柔軟性を与えるため Typed Relationship（★）を使用する。責任関係の各インスタンスは 2 つのパーティ間のリンクを表し、責任関係型は、そのリンクの特質を示す。これにより任意の数の組織上の関連を扱うことができる。必要とする関連の各種類に対

して責任関係型のインスタンスを作成し、パーティ同士をその型の責任関係とリンクすればよい。



★【訳者注：図右の regional は functional の間違い】

図 0.16 ボストン営業所は地域構造上ニューイングランド支店の下にあり、役割構造上営業部の下にあることをこのインスタンス図は示す。

責任関係は Organization Hierarchy（★）と類似したナビゲーションを表現するのに使用できる。主な違いは、この振る舞いの多くは責任関係型により限定されることである。「ボストン営業所の親は誰か」という代わりに「ボストン営業所の地域の親は誰か」という。

Accountability（★）はもっと複雑な変化にも向いているようで、最も特異なものは、知識レベルを用いてどのような種類のパーティ同志がリンクできるかのルールを捉えることである。(P.★参照)

When to use it

Accountability（★）は Organization Hierarchy（★）では十分でない組織構造を表現すると

き使用する。Accountability（★）は Organization Hierarchy（★）より使用法がかなり複雑であるため、必要になるまで使用しないこと。主に、数個の異なるライン組織を記録しなければならないとき用いる。もし 2,3 しか必要ないなら Organization Hierarchy（★）、ラインの数が増加するなら Accountability（★）が単純である。

Organization Hierarchy（★）を Accountability（★）にリファクタリングするのは難しくない。始めはどちらもパーティ上で使用できる。第１ステップは、クライアントが元の階層のリンク操作に Accountability（★）インタフェースを使用できるようインタフェースを合理化する。これが終わると、旧インタフェースを落とすか、便利なら保持することができる。不要になるまで古い実装を新しいものと並べておくことができる。

Sample Code

責任関係の基本的実装は３つのクラス party，accountability および accountability type を持つ。accountability type の単純な形式は、興味ある振る舞いは持たない。

```
class AccountabilityType extends mfNamedObject {
   public AccountabilityType(String name) {
         super(name);
   }
}
```

このサンプルでは、accountability と party 間で双方向リンクを使用している。

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

   Acountability (Party parent, 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 mfNamedObject{
    private Set parentAccountabilities =new HashSet();
    private Set childAccountabilities =new HashSet();
    public Party(String name){
            super(name);
    }
    void friendAddChildAccuntability(Accuntability arg){
            childAccountabilities.add(arg);
    }
    void friendAddParentAccountability(Accountability arg){
            parentAccountabilities.add(arg);
    }
}
```

構造を作成するために必要なコードを示す。構造のほとんどのナビゲーションは、Organization Hierarchy（★）のサンプルコードと同様のコードを含む。しかしながら、親と子を見つけるためのコードは若干複雑になる。

```
class Party...
    Set parents(){
            Set result = new HashSet();
            Iteratr 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;
    }
```

これらのオブジェクトをこのようなコーディングで使用できる。

```
  Class Tester
        AccountabilityType supervision = new AccountabilityType("Supervises");
        Party mark = new Party("mark")
        Party tom = new Party("tom")
        Party stMarys–new party ("StMarys");
    Public void setUp() {
        New acountability (stMarys,mark,appointment);
        New Accountability (stMarys,tom,appointment);
    }
    public void testSimple(){
        assert(stMarys.children().contains(mark));
        assert(mark.parents().contains(stMarys));
    }
```

しばしばひとつの責任関係型に従ってナビゲーションする必要がある。

```
  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(tm ,mark,supervisin);
        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

Organization Hierarchy（★）では、責任関係構造のなかでループを生じない保証を得るため、

パーティ間をどのようにつなぐかについてあるルールに従わなければならない。

ここに不変のオブジェクトとして責任関係を取り扱う Organization Hierarchy（★）との違い

がある。責任関係を作成するときチェックを必要とする。責任関係を作成できるかどうかチェックし、作成するというアプローチをとる。これはコンストラクタではできないため、ファクトリメソッドが必要である。

```
class Accountability...
   private Accountability (Party parent,Party child,Accuntability Type 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,Accuntability Type 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;
   }
```

世界最高速のアルゴリズムではないが、キャッシュでも解決しない。

これでループをチェックできる。

```
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().cntains(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

本文責任関係の基本型はきわめて自由で、どのような組み合わせのパーティをどのような責任型

と組み合わせることも許される。あるアプリケーションではそれでよいが、しばしば責任関係を組み合わせられる制限を設けたくなる。Knowledge Level（★）を使うことによりこれができる。責任関係の知識レベルはパーティ型の組を必要とし、パーティ型が互いにどのように繋がるかを説明する責任関係型を設定する。

ConectionRules

Connection Rules(図 0.17)は単純であるが知識レベルの融通性を与える。各責任関係型はパーティの親子のペアを定義するコネクションルールの組を持つ。図 0.18 は知識レベルをどのように構成するかのひとつの例を示す。

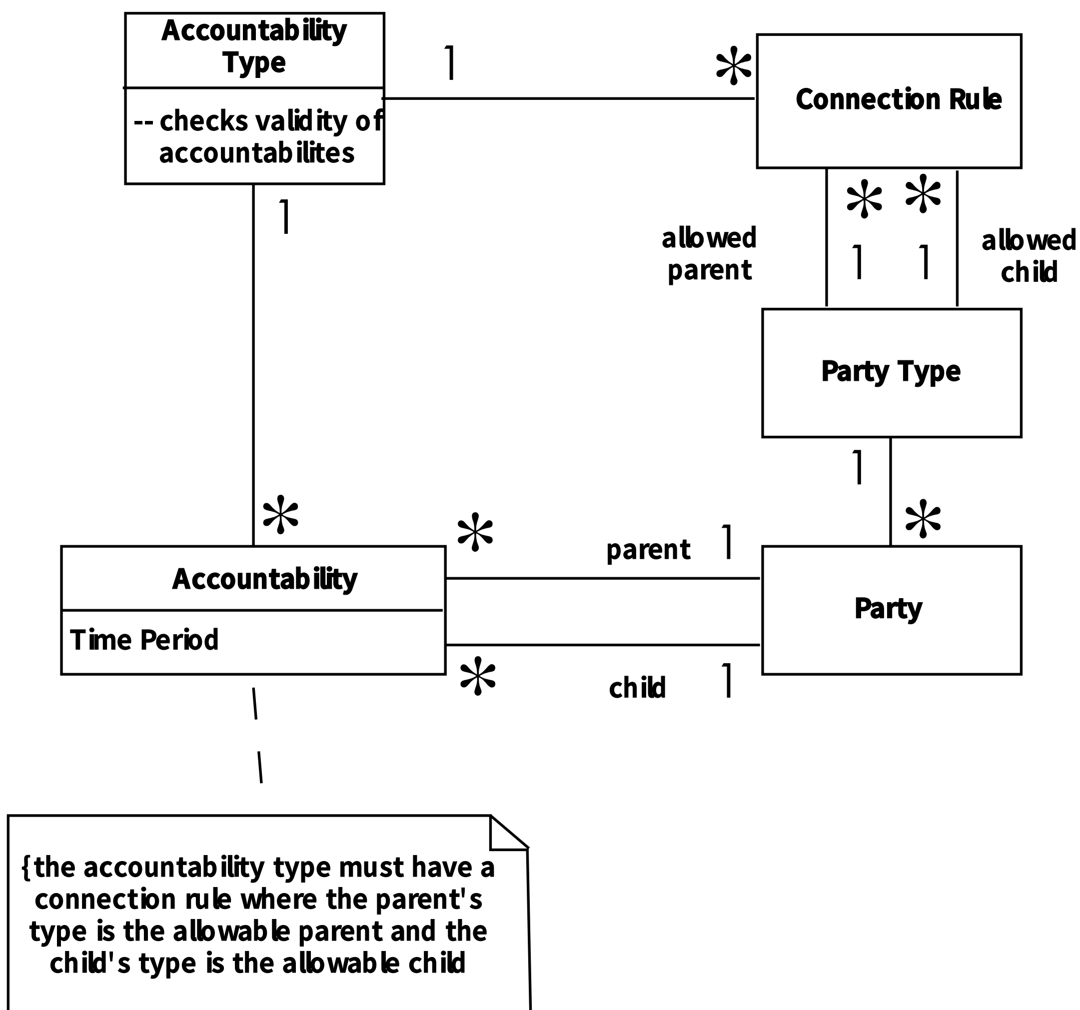


図 0.17 方向付きグラフ(directed graphs)の知識レベル

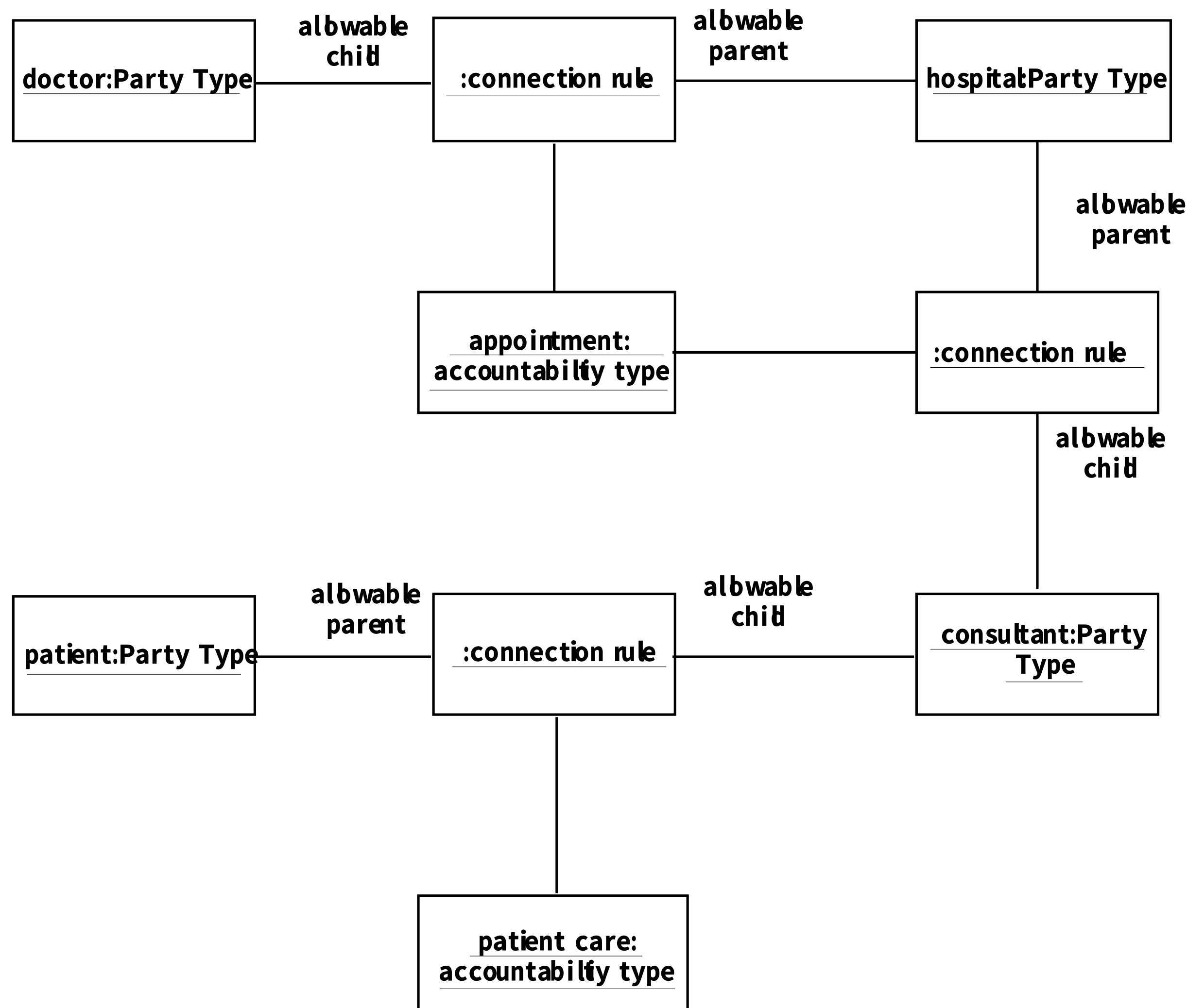


図 0.18 この例はふたつの責任関係型を示す。appointment 責任関係型は doctor および consultant を hospital との appoint を許す。patient care 責任関係型は、consultant の patient に対する責任を記録する。

責任関係をチェックするのに知識レベルを使用する。責任関係を作成するとき、その型が正当性をチェックする。異なる責任関係型は異なる関係ルールを定義するため、チェックは責任関係型に置くのが最善である。従って、ひとつのオブジェクトを置き換えることによりルールを変更したいなら、責任関係型は置き換えるポイントになる。

Sample Code

作成済のサンプルコードに、知識レベルのコネクションルールを追加する例を示す。

```
class PartyType extends mfNamedObject {
   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;
   }
}
```

ルールは accountability type に追加される。このようにルールを追加する方法は複数存在するので、コネクションルールを保持するのにサブクラスを作成する。

```
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));
   }
```

構造的に、コネクションルールは許される親と子の型を保持する。

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

これでコネクションルールを持った責任関係型がセットアップできる。

```
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.addC nnectinRule(h spital,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 ("StMary's",hospital);
```

珍しい構造のもとでの確認の振る舞いを見よう。責任関係はすでに基本ルールとしてチェックの振る舞いを持つ。適当な知識レベルによりこの振る舞いに、責任関係型に正しいパーティ型をチェックするリクエストを追加することができる。(図.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.canCreateAccontability(parent,child);
  }
```

これで責任関係型はコネクションルールを用いて親と子のチェックができる。

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

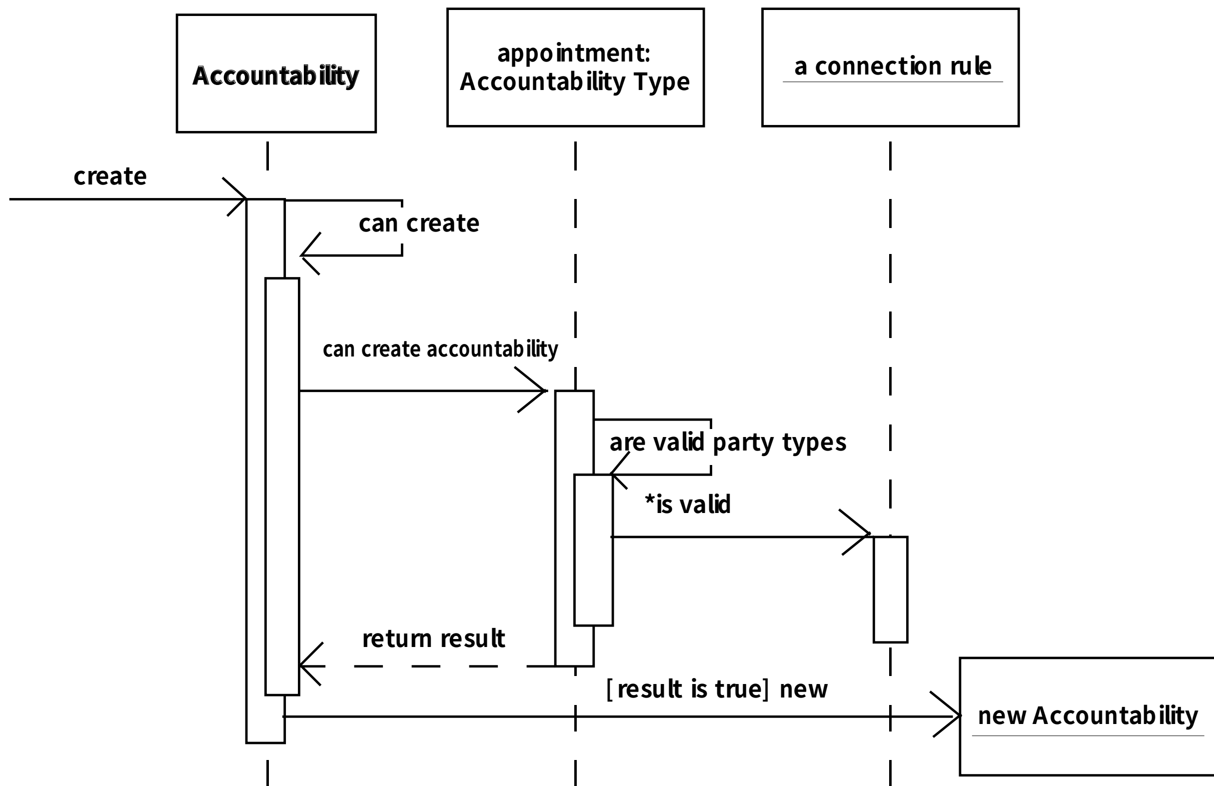


図 0.19　知識レベルによる責任関係チェックのための対話

これでコネクションルールのエラーなく責任関係を作成できる。

```
class Tester...
  public void testN C nnectinRule(){
        try {
              A ccountability.create(m ark,stM arys,appointm ent);
              fail("created acc untability w ithout connection rule");
        }catch (Exception ign re){}
        assert(!stM arys.parents().c ntains(m ark));//am  I paranoid?
  }
```

Hierarchic Accountability Type

責任関係を使用するとき、いくつかの責任関係型は階層が必要であるが、他はその必要がないことがある。責任関係型の確認を行うメソッドに選択できる階層ルールを組み込むことができる。

Sample Code

サンプルでは責任関係型のフラグを用いた。

```
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 は責任関係型のルール表現する最も一般的方法である。しかしながら時と共にパーティ型のレベルの厳密な集合が存在するという状況に遭遇する。その例は、国のパーティは州を子に持ち、それらは郡を子に持ち、それらは市を子に持つという地域の細分化である。これは Connection rules で表現できるが、責任関係型のレベルをリストする方が簡単である。

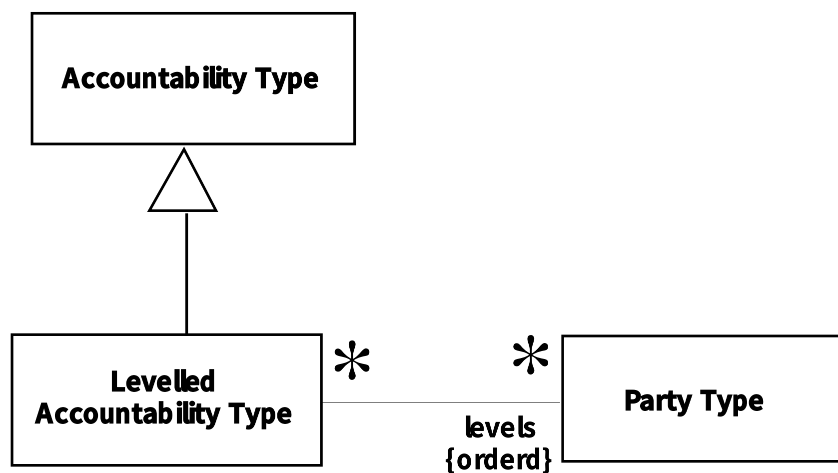


図 0.20　レベル付き責任関係型

Sample Code

レベルのセットアップには異なるテストが必要となる。

```
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);
        melrse = 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 ignre){}
  }
  public void testSameLevels(){
        try {
              Accountability.create(ma,nh,region);
              fail();
        }catch (Exception ignre){}
  }
  public void testSkipLevels(){
        try {
              Accountability.create(ma,melrose,region);
              fail();
        }catch (Exception ignre){}
  }
}
```

しかしながらレベル付き責任関係はきわめて単純である。レベルのリストをセットできることと、

責任関係型に存在するメソッドからパーティ型の正当性をチェックできることが必要である。

```
class LevelledAccountabilityType extends AccuntabilityType {
  private PartyType [ ] levels;
  public LevelledAccountabilityType(String name){
         super(name);
```

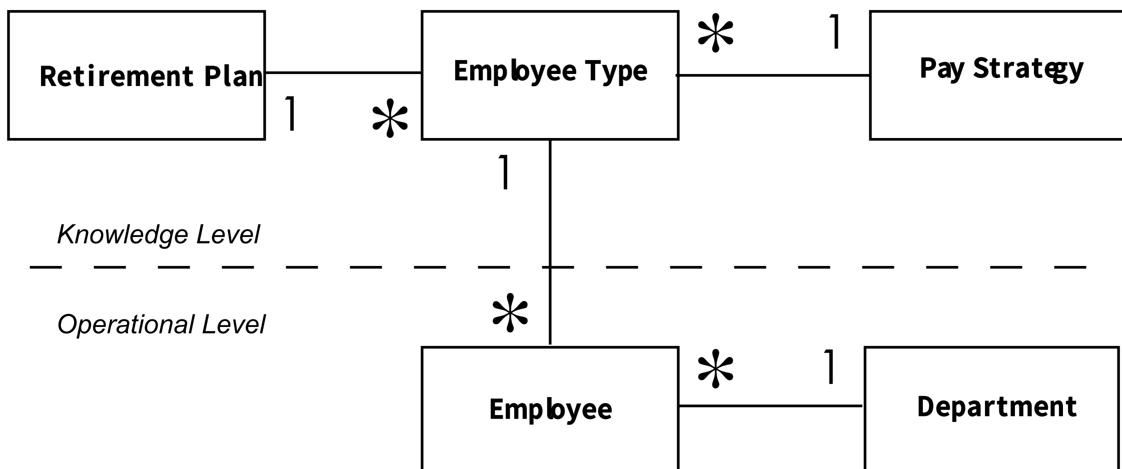
```
    }
    void setLevels(PartyType [] arg){
        levels = arg;
    }
    protected boolean areValidPartyTypes(Party parent,Party child){
        for (int i=0;i<levels.length;i++){
            if (parent.type().equals(levels [i]))
                return (child.type().equals(levels [i+1 ]));
        }
        return false;
    }
}
```

このサンプルはレベルをスキップできない。つまり city を state の子にできない。レベルスキップが必要なら areValidPartyTypes を追加するのが簡単である。（読者の演習問題とする）しかしながらあるレベルはすきっぷし、他はそうしないなら、コネクションルールを用いた方がよい。

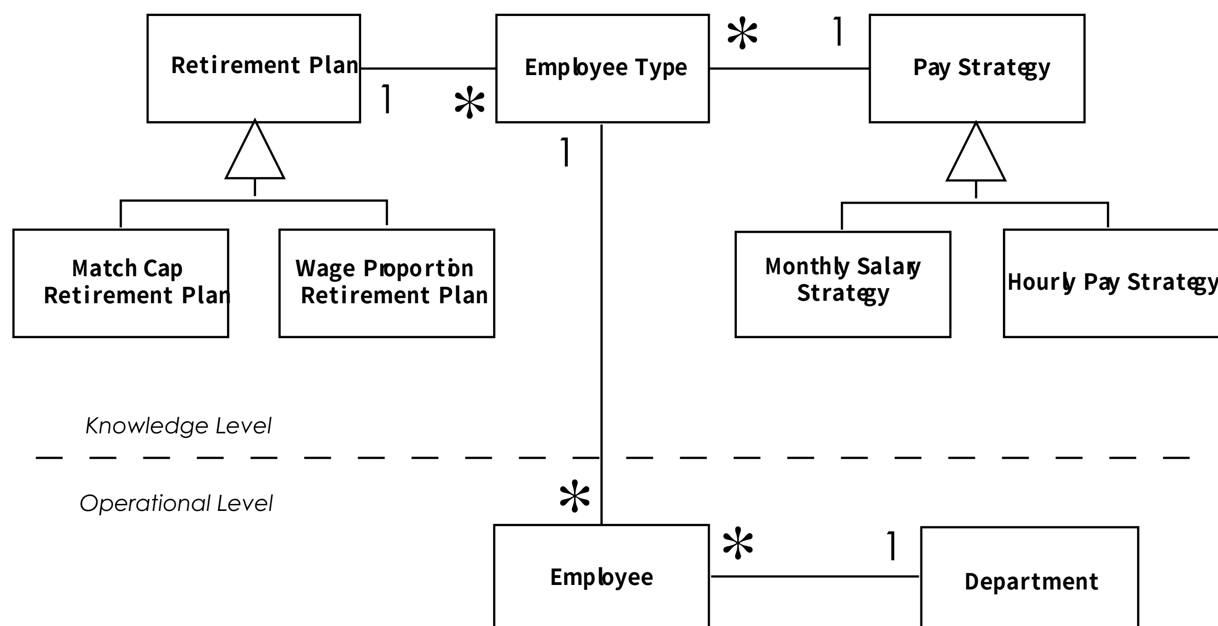
Knowledge Level

あるオブジェクトの集まりがどのように振る舞えばよいかを

記述した、別のオブジェクトの集まり(メタ・レベル＜meta level＞)とも言う)



Knowledge Level（★）は、抽象的なパターンのひとつで、それがどんなものであるかを記述するのは大変難しい。とは言っても、少しでも複雑なオブジェクトの系ではしばしば見い出されるものである。Knowledge Level（★）が現れるのは、あるオブジェクトの集まりのインスタンスが、操作オブジェクトの集まりの振る舞いに影響を与える場合である。こういった知識オブジェクトを生成し、つなぎ合わせることによって、システムを「プログラミング無しで」変更できるようにするのが典型的な例である。



<<図 0.21 従業員(employee)を用いた知識レベルの例>>

図 0.21 は従業員(employee)に関する知識レベルを用いた例である。ここで意図しているのは、報酬と退職のポリシーをひとつに組み合わせることによって、従業員の型というものを作り出すことである。こうすると、各従業員はどのポリシーを採用するかを示す従業員型で印付けられることになる。従業員の就業形態をカスタマイズする必要が出てきたら、報酬体系と退職計画の別のインスタンスをもつ、新しい従業員型(Employee Type)のインスタンスを生成すればよい。これらのオブジェクトの選択と結合が、システムの構成を決めていることになる。

Terminology

この本では、前の版と同様、Knowledge Level（★）という用語を使う。とはいうものの、設計者の仲間うちでこれがまったく標準的な用語として使われていると主張するつもりはない。まだ今のところ標準的な用語といえるようなものは存在していないので、私は取りあえず知識レベル(knowledge level)という用語にこだわろうと思う。

私は英国国営保健サービスのCosmosプロジェクトで働いているときに、操作レベル(operational level)、知識レベル(knowledge level)という用語を使い始めた。このプロジェクトで、私は医師と看護婦の合同チームについて仕事をしていたのである。我々がその用語をひねり出したのは、そのときのモデルの見え方にまさにぴったりしていたからなのだ。操作レベルでのオブジェクトは医療者の日々の操作的な振る舞いを表していたし、知識レベルのオブジェクトは彼らの医療上の知識を表していた。

一般的によく使われる別の用語としては、「メタ・データ(meta-data)」のメタ(meta)のような言い方がある。メタとは「〜について」という意味のギリシャ語なので、メタ・データとは「データについてのデータ」ということになる。「メタ・オブジェクト」や「メタ・レベル」について喋っている連中をよく見かけるだろう。もしこの概念を表す用語の投票をすることになったら、「メタ何とか」は多分リストのトップになるに違いない。しかしそれでも私としては「知識レベル」を推したい。ソフトウェア関係以外の人にも意味が通じると思うからだ。

別の言い方としては「アクティブ・オブジェクト・モデル(Active Object Model)」がある。この言い方の根拠は、知識オブジェクトが操作オブジェクトの振る舞いを決定するアクティブな部分を司っていることからだ。しかしこの用語は広く使われているわけではないし、操作レベルが何か受動的なもののように取られてしまうのであまり好みではない。

私の個人的な習慣としては、図の上側に知識レベルを、下側に操作レベルを書き、その間を点線で区切る(そうしない場合もある)。これもまた、私がこれが役立つものだと分かったときに関わっていたCosmosプロジェクトでの習慣である。しかしこれはどんな標準にも含まれていないし、私の世界への影響力など、私が期待するほどのものでないのは確かだ!

知識レベルと操作レベルの間を「何とか」と「何とか型」という関連で結合するのは、一般的な慣習である。これは非常によく使われるようになってきたので、私も他の名前の付け方を提唱す

るのをためらってしまう。型オブジェクト(type　object)パターン

に従って、「何とか型」を「何とか」の型オブジェクト(type　object)と言うこともある。

Making it work

知識レベルは全体的に考えるには大変込み入ったものであり、普通の人はこれを先進的なオブジェクト指向テクニックだと見なすだろう。難しい点は、ソースコードをほとんど(あるいはまったく)変えずに変更を扱えるようにするためのオブジェクトの集合を選び出すところにある。どの組み合わせならばうまく動作するかをあらかじめ見極めるのはたいがい難しい。そして、もっとも複雑なフレームワークと同様、設計は進化し続けなければならない。

これをどうやってやるのかを抽象的に述べることはできない。その代わりに、知識レベルを含むような個々のパターンをよく見て、そこから何か使えそうな事柄を学び取ろう。そうすれば、あなたも特定のパターンを使うことができるようになるはずだ。また、知識レベルをよく見ることで、あなた自身の環境に対する知識レベルがあなたを助けてくれることもあるだろう。

When to use it

　人が知識レベルを使いはじめる、あるいは使おうとし始める。そのときよくあるのは、これでプログラマでない人が「プログラミング無しで」システムの挙動を変更できるようになるだろうと考える場合だ。こういうのはしょっちゅうあることなのだが、私の経験ではこれは「幻の黄金郷」に過ぎない。知識がどのように働くか、知識をどのように構成するかを理解することはまだまだ非常に込み入った課題であって、普通のプログラマの手には余るようなことなのである。実

際、経験豊富なプログラマでさえ、知識レベルを扱うのが非常に難しいと実感することも多い。

私の考えでは、知識レベルの利点は設計の経験を積んだ人が、他の方法でやるよりはすばやく変更をできるようになることである。

これらの変更の多くを実行時に行うことができるという事実も、また魅力であろう。しかしこれには警告を伴う。つまりシステムを「プログラミング無しで」変更できるからと言って、システムをテスト無しで変更していいと言うことにはならないからだ。実際、知識レベルに対するテストはより重要になり、しかもたいていはテストをするのが難しくなる。知識レベルはそもそも大変動的なものを捕らえるためのものであり、だからこそテストも大変難しい課題になるのである。

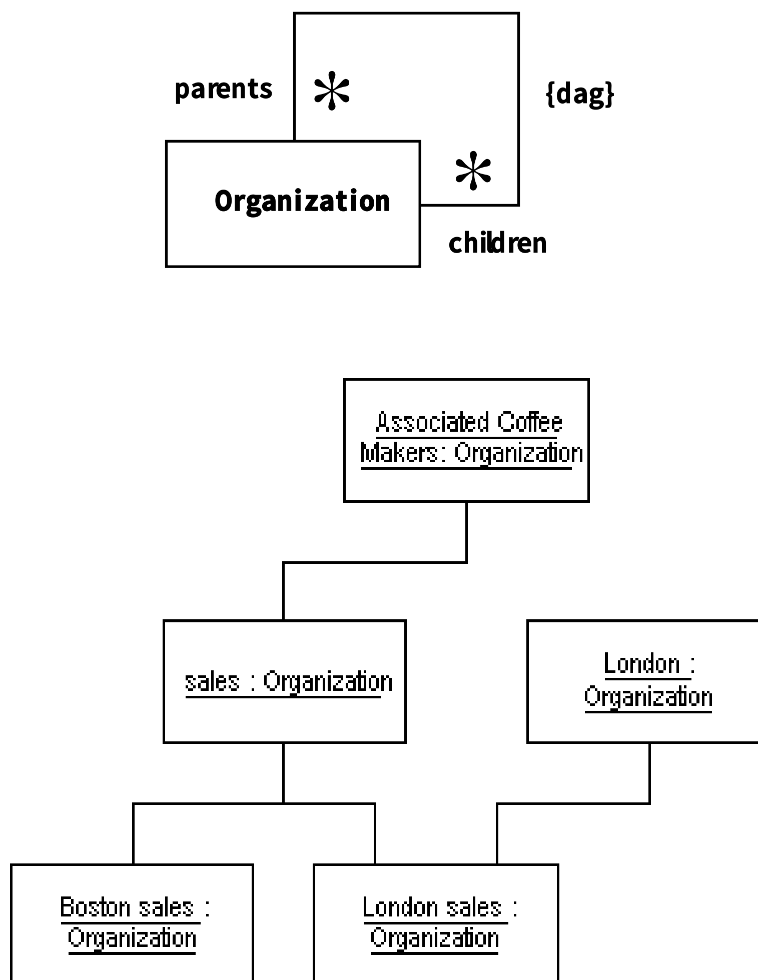
「プログラミング無しで」変更できると言うことはまた、変更をツールを使わずに行わなければならないということでもある。知識レベルだからといってデバッグが不要になるわけでも、構成管理が不要になるわけでもない。あなたの「プログラミング不要」環境がデバッグ、構成管理、テストのためのツールを用意していない(し、これからも用意するつもりがない)ならば、あなたはそれらの道具無しでやるか、自分で作るしかないのである。知識レベルのためのグラフィカル・エディタを作る人々がよくいるが、実際には「プログラミング不要」のいちばん難しい部分は編集以外の部分であるのが普通だ。

これらのことはすべて、知識レベルを利用する上での重要な警告になるが、もしこれらのことであなたがおびえてしまうようならば、ここは考えどころである。これは軽く使うようなパターンではない。しかし、複雑なビジネス・システムでは知識レベルがそのコストに見合うようなポイントもいくつかはあるものである。例えば一定しないルールが大量にあるような場合には、知識レベルは大いに意味があるだろう。我々は実際のところ、これらを使うにあたってのガイドラインがどんなものであるかということについて、まだあまり言うべきことを持ち合わせていない。しかし、これらはめったに使われることはないが、本当に必要な場合にはそれが本質的な役割を

果たすということは分かっている。ちょうどパラシュートみたいにね。

Object Graph

グラフ構造で結合されたオブジェクトの集まり



オブジェクト・グラフ(Object Graph)は、同じ先祖の型を持つ多数のオブジェクトが再帰的な構造で結合されているような状況を表すのに一般的に用いられる方法である。これらは以下のようなものに見い出される、組織構造、作業細分化構造(WBS)、成果物構造、(どこまでも続く)...

オブジェクト・グラフについて抽象的に語るとすると、通常はノードとリンクについて語ること

になる。上図において、組織がノードであり、親子の関連のインスタンスがリンクである。

しかし、抽象的なオブジェクト・グラフについてうまく説明するのは難しいことに気付いた。それでこのパターンを組織構造(organizational structures)の章に含めることにした。というのも組織構造は Object Graph（★）が実際に使われている、よい例題を提供してくれるからである。このパターンについては、私は実際にはいくつかの用語をひねり出すだけである。このパターンを使い始めれば、これらの用語を使うとオブジェクト・グラフについて話すのに便利だということが分かると思う。また、この考え方の抽象的な性質についても述べるつもりである。

今まであなたが実際に使ってきたほとんどのグラフ構造は循環を含まないものであったと思う。これはつまり、あなたの祖先を辿っていっても自分に会うことはないということだ。

Variations

オブジェクト・グラフの主題には、よく使われる変奏がたくさんある。

オブジェクト階層(Object Hierarchies)は各ノードがひとつだけの親ノードを持つ。階層には制約が多いが、ただひとつの親を持つという事実から、いくつかの興味深い振る舞いが許される。例えば Aggregating Attribute（★）。Organization Hierarchy（★）は、オブジェクト階層の一例である。複数の親を持つ構造は、しばしば束、ネットワーク、あるいは威嚇的に(数学的には正しいのだが)閉路を含まない有向グラフ(directed acyclic graph)(縮めて DAG)などと呼ばれる。

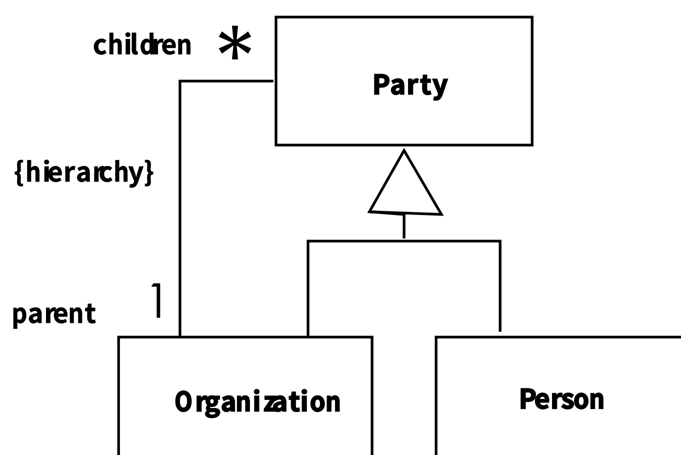
構造中のリンクは、関連あるいは別個の関連オブジェクトによって表される場合がる。Organization Hierarchy（★）では単純に関連を使っているが、Accountability（★）では関連オブジェクトを使っている。Accountability)（★）では、accountability クラスが関連オブジ

ェクトになっている。関連オブジェクトを使っている場合には、関連が型付けされていることが多いのに気付くだろう。こうすると、同一のノード集合の上に複数のグラフを定義することができるようになるのである。この点に関しては、Accountability（★）で複数の組織構造について議論した。

図 0.22 に見られるように、リーフ・ノードとリーフでないノードを別々のサブタイプに分ける場合もある。リーフである場合とそうでない場合とでノードの振る舞いに重要な違いがある場合にはこうすべきである。しかし、この場合でもインタフェースのできるだけ多くを親の方に入れるのがいいやり方だと思う(たとえ、それがあまり意味が無さそうに思えたとしても)。

ひとつの例として numberOfChildren メソッドを考えてみよう。一見したところ、子供を持つのは Organization だけなのだからこのようなメソッドは Organization に入れるべきであると思うだろう。しかし、このインタフェースを Party に定義しておき、Person では 0 を返すようにそのメソッドを実装しておくのが便利なのである。そうしておけば、例えば numberOfDescendents のようなメソッドを大変簡単に書けるのが分かるだろう。

パターンの達人ならば、図 0.22 を見て、composite パターンの古典的な形であると見抜くだろう。実際 composite が階層オブジェクトに適用されるのが普通である。



<<図 0.22 リーフ・ノードとリーフでないノードをオブジェクト・グラフの中で分ける>>

Making it work

Object Graph（★）を組織的なパターン(特に Organization Hierarchy（★）、Accountability（★）で利用するという点について議論してきた。Object Graph（★）を利用するにあたって必要なことの多くは、この中で見つかると思う。

ここで再度繰り返しておく価値があると思うのは、リンクの名前の付け方についてである。最近では私はいつでも parent/child という名前を使おうとしている。たとえそれがドメインによく合うものではないとしても。親しみのある関係というものは、この種の関係のメタファとして強力なものである。例えば私が「マーケティング部門は私の大叔父にあたる」と言ったとしたらとても変に聞こえるだろうが､それがどんな関係かについては明確に理解してもらうことができる。

コンピュータ・プログラムで、グラフ構造をどう表すかについては、膨大なアカデミックの成果がある。このほとんどはあなたが知っておかなければならない事柄から程遠いとは言え、私はいつもどれほど多くの人がこれらの成果について何も知らないかということに驚かされるものだ。もしグラフ構造を扱っていて何かしら困難な点に出くわしたら、グラフ理論、アルゴリズム、データ構造などの玄妙な成果に少し鼻を突っ込んでみるとよい。多分あなたの抱えている問題に対する答えを見つけることができると思う。

When to use it

この構造をいつ使うべきかという選択は、特定のドメインの、特定のパターンの適用ごとに考えるのがよい。そういうわけで「いつ、どんな種類の Object Graph（★）を使ったらよいか」という問い方より、「いつ Organization Hierarchy（★）あるいは Accountability（★）を使ったらよいか」という問い方のほうが役に立つと思う。

しかし Organization Hierarchy（★）や Accountability（★）についての説明と議論を読めば、

何がしかの有用なガイドラインは与えられると思う。特に関連オブジェクトは単なる関連にくらべていくぶん込み入ったものであるので、必要なときに限ってそれを使うということを心に留めておいてほしい。

翻訳：河合 昭男
山田 正樹
長瀬 嘉秀
矢野 大介
窪田 寛之
久保 雅恵

原文名：Analysis Pattern2　by Martin Fowler