Mellor’s Problem:

A case study comparing Booch and Shlaer-Mellor.

Introduction

There has been a very interesting thread on comp.object lately. Steve Mellor (co-author of the Shlaer-Mellor OO method) participated in a design shoot-out with several other people on the net. A simple problem statement was posted. The participants designed solutions and then posted them to the net. Then they discussed the relative merits of their solutions.

There is a lot to learn from following discussions like this. Therefore, I have gathered what I believe to be the best of the articles from this newsgroup thread, and organized them into this issues Engineering Notebook column.

> *Warning: The following contains “net-language” that some sensitive readers ought to find offensive.* ;)

A little history

Since the late 80’s there has been a menagerie of competing OO analysis and design methods and notations. The list is surprisingly long. However, there are only a few that could be said to be real “contenders” in the methodology market. Booch, with his Booch-94 method, James Rumbaugh with his OMT method, Ivar Jacobson with his Objectory method, and Steve Mellor and Sally Shlaer with the Shlaer-Mellor method.

I remember going to OOPSLA in 1993 and seeing Booch, Jacobson, Mellor, Rumbaugh and Rebecca Wirfs-Brock (another methodologist) debate their methods in a panel discussion named: “Shootout at the OO corral”. It was actually quite clever since each panelist was tasked with the job of describing some other panelist’s method in a positive light. At the end of the panelists’ presentations, the floor was opened to questions. So I found a microphone and asked the group whether or not they thought it was time to try to unify their various methods.

The response was mixed. Booch was in favor of unification; Rumbaugh was undecided; and Mellor against.

A year and a half later, Rational announced that James Rumbaugh had been hired to work with Booch on a method which unified OMT and Booch-94. This unification is well under way. Booch and Rumbaugh released their 0.8 document describing the unified method and notation at OOPSLA 95; this past fall.

At roughly the same time, Rational made another announcement. Ivar Jacobson had been hired to work with Booch and Rumbaugh on the unified method. And so, by the closed of 1995, three of the four “contenders” had joined forces to unify their methods and notations into a single object oriented analysis and design method for the software industry.

Shortly after Rational made this announcement, Project Technologies Inc. (Steve Mellor’s and Sally Shlaer’s company) made a public announcement of their own (which they posted on comp.object). The Sidebar contains the announcement, however the gist of it is that PTI applauded the collaboration of Booch, Rumbaugh and Jacobson because it represented a consolidation of the “elaborationists” and that the industry could now choose between either “elaboration” or “translation” when choosing OO methodologies. The announcement went on to say that the Shlaer-Mellor method was an important “Translational” method.

Martin’s Complaint and Challenge

In response to a completely different article, but making reference to PTI’s announcement, I posted the following:

```
newsgroups: comp.object

>Larry_Kovnat@wb.xerox.com (Larry Kovnat) wrote:

>>I recently attended a course in the Shlaer-Mellor method.  While
>>monitoring this group, I have not noticed any articles on this method.
>>I would appreciate posts or email comparing this technique to others.
>>Any aspects would be of interest at this point: simulation, code-
>>generation, relationship to SEI SW maturity model, etc.

"Timothy L. Crouch" <tcrouch@berkshire.net> writes:

>The major difference between Shlaer-Mellor and methods such
>as Booch and OMT is that Shlaer-Mellor is based on translation
>while the other methods are based on elaboration.

>In Shlaer-Mellor, a model is created which is translated into
>code using a translation model.  The translation model specifies
>how the elements of the solution model are to be translated into
>code.

From: rmartin@rcmcon.com (Robert Martin)

This has always been somewhat puzzling to me, PTI's (Project
Technology Inc.  Shlaer's and Mellor's company) long
advertizements on this group notwithstanding (See the advertizement
disguised as a press release regarding PTI's support for Rational
acquiring Objectory.)

As I understand it, analysts create a model of the application which
captures every bit of the needed behavior, but contains none of the
```

PROJECT TECHNOLOGY’S SHLAER AND MELLOR VOICE SUPPORT OF RATIONAL’S ACQUISITION OF OBJECTORY.

MOVE SEEN AS FINAL STEP IN CONSOLIDATION OF OBJECT DEVELOPMENT MARKET-- EXPECTED TO REDUCE MARKET CONFUSION AND ACCELERATE GROWTH.

October 19, 1995 -- Berkeley, CA -- Sally Shlaer and Stephen J. Mellor, of Project Technology, Inc. today voiced their support of Rational Software Corporation,s (Santa Clara, CA) acquisition of Objectory AB (Kista, Sweden). As part of that agreement, Ivar Jacobson, Objectory,s founder, will work with Rational,s Jim Rumbaugh and Grady Booch to define a single notation for elaboration-based software development.

Shlaer and Mellor believe the acquisition marks the final step in a trend toward consolidation of the industry into two camps: elaboration and translation. The trend began last year with the alliance between Rational Software Corporation and the Martin Marietta Information Group’s Advanced Concepts Center (ACC). According to Ms. Shlaer and Mr. Mellor, this latest move will further reduce confusion between the two approaches and focus the industry on the differences between the translation and elaboration software development paradigms. Sally Shlaer and Stephen J. Mellor are the creators of the leading Shlaer- Mellor Method for systems development, and pioneers in the development of translation of analysis models directly into code.

“This announcement marks the culmination of the convergence of the elaborative methods,” said Stephen J. Mellor, senior vice president of Project Technology. “With the elaboration approach, developers iterate through the stages of analysis, design and code for each subsystem of the overall design. Conversely, the Shlaer-Mellor Method is based upon the translation paradigm, where analysis models are transformed into application code. A characteristic of the translation approach is the complete separation of the application from implementation details, which enables developers to create and maintain complex software systems at the graphical model level.”

The differences between the elaboration and translation approaches were described in a recent article in Computer Design magazine. About the elaboration paradigm, Tom Williams wrote, “The work of three methodologists [Jacobson, Rumbaugh and Booch] is gradually merging.... The common thread, which is more important than notational niceties, is an application developed by successively elaborating and refining the analysis model and implementation.” In the same article, Mr. Williams went on to explain, “Shlaer-Mellor is called translative because you create separate models for the application as well as the software architecture, then generate code from the application model and fit it to the architecture via a set of translation rules. Among other things, application and architecture are divided and can be assigned to separate teams which can work in parallel.”

“While it is flattering that the other methodologists are beginning to leverage some of the Shlaer-Mellor concepts, such as domains, it is important to point out that the two approaches are fundamentally different,” continued Sally Shlaer, director of research of Project Technology. “The translation approach enables developers to work at a higher level of abstraction during the entire life-cycle of a project. Developers can work in parallel on different domains of the system, at a graphical, model-based level. Because the application models and architecture have been verified as correct, testing, integration and time-to-market are shortened. Perhaps most importantly, users are also able to maintain the system at the model level, greatly simplifying the time and cost associated with the overall lifecycle of a system.”

implementation details. (e.g. whether containers are built from hash tables or linked lists). The analyst then “colors” parts of his

```
analysis model to direct the translation process to product the proper
kind of code.  (those containers that should be hash tables get one
color, those that should be linked lists get another).  Then "poof"
you run some magic program (that some other team is supposed to write,
or that perhaps you can buy from PTI, I am not sure) and code is
generated.

If I appear skeptical, I am.  I would like to see Schlaer and Mellor
describe this process in more detail.  They do not lack for words on
the net, but the words that appear never go further than what I have
described.  I would like to see a really solid example.
```

Mellor's Counter Challenge

Now, Mellor had been participating in a different thread in which he had been refuting the need for "manager" objects. However, having seen my challenge, and much to his credit, Mellor picked up the gauntlet and posted the following:

```
From: steve@projtech.com
Subject: Re: Managers and Discount Policy

One [suggestion] was echoed by Mr Robert Martin.  Namely
that we should take, or build, an example and do as I proposed in my
initial post--Build a number of solutions and then evaluate them
for 'goodness,' and *then and only then* decide if the solution is 'OO'.

With that suggestion then, I offer an example [...]. I think
it would be instructive to see several proposed solutions.
I have one in mind that--I believe--is very robust and is unlikely
to change in the face of changing requirements.  I shall hold
that in reserve for a while. [...]

Here's the problem.

Background

In several states in the US, and in many European countries, utilities
such as electricity, are regulated by some commission that sets rates.
In California, there is the California Public Utilities Commission (PUC)
that set rates for each electric company in the state.  The PUC manages
a "rate book" that says who is charged for what.  It's about an inch
thick, and each page (or three or five pages) constitutes a 'Rate
Schedule'.
For each rate, the qualifications for the rate are described, and the
precise dollar amount for each kilowatt hour (kwh) is prescribed.
(I have no idea what a kwh costs, so I'll make up numbers.)

Each month, the PUC holds a several-day meeting to discuss, among
other things, changes in the Rate book.  Sometimes, it's just a
matter of changing from 5c to 5.5c per kwh, but more often there's
a new schedule defined for certain types of customers.  (Here I'm
deleting all sorts of ranting of the libertarian, free market sort.)
Subtext:  The schedules therefore  change all the time, for
```

```
purely political....[ranting deleted]

Rate Schedule Examples

There are several kinds of customers defined, at present: Residential,
Business and Industrial.  There are several Residential type customers.
One of these is the "Lifeline" residential customer, designed to
provide cheap electricity to people that can't afford the 'normal' rate.
For these customers, we charge a flat rate of 3c per kwh, all year round,
for the first 100 kwh.  After that, we charge 5c/kwh to 200kwh.
If they're using more than 200 kwh, they don't qualify for Lifeline
service.

Normal residential customers are charged depending on where they live.
The state is divided into three territories, though there's a fourth one
in the works, according to climate.  The 'climate' idea  represents
expected minimum electricity usage for the area _at a certain time_.
We charge more in the summer than the winter (because of air-
conditioning).
We charge according to the following rules:

* In Terr 1, 6c in Summer; 7c in Winterr

* In Terr 2, we charge 7c in winter and 6c in summer

* In terr 3, we charge 6.5c flat rate

Business customers are charged on a sliding scale: 9c for the first kwh,
down to 5c at 1000kwh, 5c thereafter.  This is computed _per site_, so
a business with several offices cannot lump the consumption together.

Industrial customers are charged according to their willingness to
accept power interruption.  If they can accept any interruption,
for any length, with no warning, we discount the business rates by 20%.
If they accept an interruption with one hour's notice, they get a 10%
discount.  If they want more than 1hr's notice OR they insist on us
restoring power within 2 hours of the initial interruption, they get
a 5% discount, all on the business rates.
```

Martin's Solution

Now this intrigued me. This was the first time, in my experience, that Steve Mellor had offered to participate in a design comparison. I was a little disappointed that he was holding his solution "in reserve", but being always anxious to learn something new, took up Mellor's counter challenge and posted a solution based upon Booch's and Rumbuagh's new unified method.

```
rmartin@oma.com posted to comp.object:

OK.  How do we begin.  First of all, why should this problem be solved
with OO?  After all, a procedural solution is pretty simple:

double CalculateRate(Customer& c)
{
  double rate = 0;
```

```
switch (customer.type)
{
  case consumer:
    switch(customer.rate)
    {
      case lifeline:
      if (customer.kwh <= 100)
      {
         rate = customer.kwh * .03;
            break;
      }
      else if (customer.kwh <= 200)
      {
          rate = 3 + (customer.kwh - 100) * .05;
          break;
      }
      // Customer does not qualify for lifline,
      // fall through to territorial case.

      case territorial:
        switch (CalculateTerritory(customer.address))
          {
          case 1:  // spec says territory 1 and 2 use same rate.
          case 2:
            rate = customer.kwh * (IsWinter() ? .07 : .06);
            break;
          case 3:
            rate = customer.kwh * .065;
            break;
          }
        break;
      default:
        assert(0); // something awful happened.
      }
    break; // consumer

  case business:
  case industrial:
    for (Iterator<Site*> i(customer.sites); i; i++)
      {
        Site* s = *i;
        rate += CalculateSlidingScale(s->kwh);
      }// for each site

    if (customer.type == industrial)
      switch(customer.industrialRate)
      {
        case interruptable:
          rate *= .8;
          break;
        case oneHourNotice:
          rate *= .9;
          break;
        default:
          rate *= .95;
          break;
      }// industrial rate
      break; // business + industrial
```

```
    } // customer.type

  return rate;
}
```

Ok, now this is a pretty simple little function. I know that it uses some pretty questionable features (i.e. falling through case statements, etc) but it is so simple that it hardly matters. ;)

So why would I be interested in using OO for this problem?

The reason, in this case, is primarily political. That little stretch of code up there is bait for class action suits. And you had better believe that utility companies are sensitive as hell about that. If you go fiddling around in that simple little algorithm up there, and you happen to screw up the industrial customers while changing the policy of the business customers, then you are going to create immense hassles and costs for the company.

By putting the entire rate policy in the same module, as part of the same algorithm, we risk interdepencies between them. A single change to one part of the policy has the potential to affect them all. Certainly every rate policy should be thoroughly tested whenever *any* change is made to any part of the module.

It is in the companies best interest to separate the policies so that each is in its own module, and so that the interdependencies between them are eliminated. Once done, a change to one part of the rate policy can affect only those customers governed by that policy. There is no risk that any other policy could be affected.

So, how do we begin?

First, a key (See Figure 1). I am using Booch's and Rumbaugh's Unified Notation version 0.8. This looks a lot like OMT, so you OMT-heads will probably recognize it rightaway.

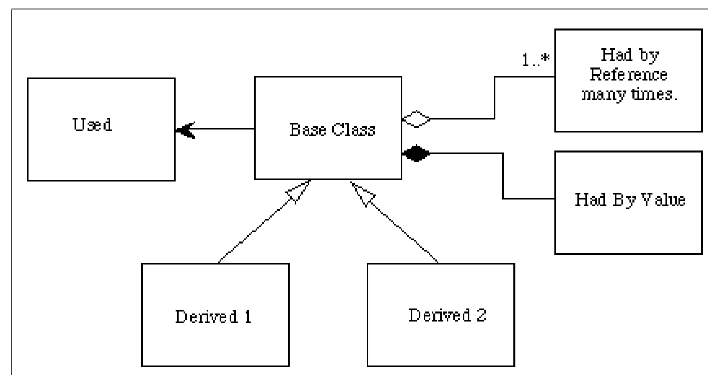

Figure 1
Unified Notation 0.8 Lexicon

One might think that there is a natural decomposition as in Figure 2.

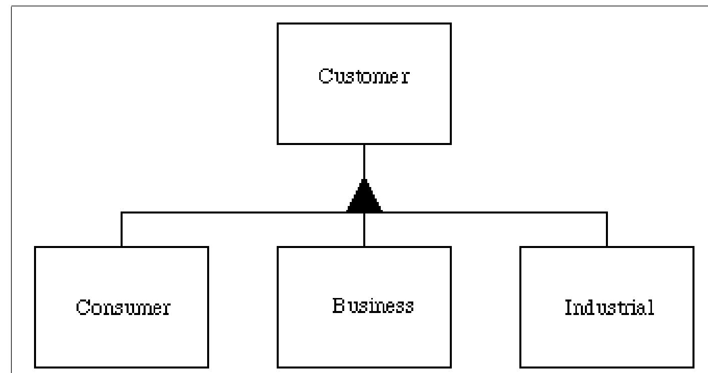

Figure 2
Possible Customer Hierarchy

However, this probably won't work in practice because it seems likely that buisness customers will need to change into Industrial customers; and it is even possible that Consumer's will change into Businesses from time to time. Thus, the 'type' of the customers needs to be an attribute that can change. (See Figure 3.)

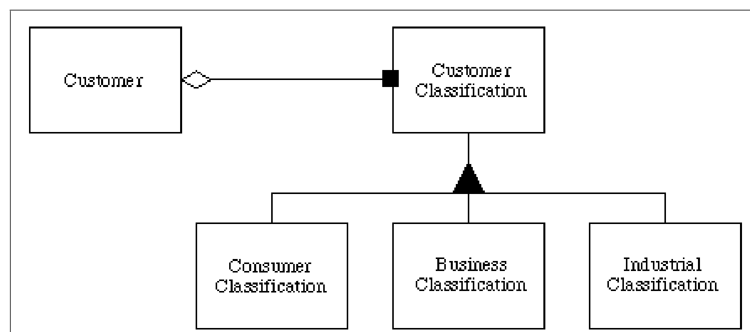

Figure 3
Customer Classification Hierarchy

Business and industrial customers can have more than one site. Why can't consumers? Probably they can, and the spec just doesn't know it yet. So we can add a list of Sites to the Customer class. However, we would like to separate the interface of the Customer class from its implementation. Thus we will create an abstract Customer class. (See Figure 4.)

Why have we done this? Consider the following algorith:

```
for (Iterator<Customer*> i(theCustomers); i; i++)
{
```

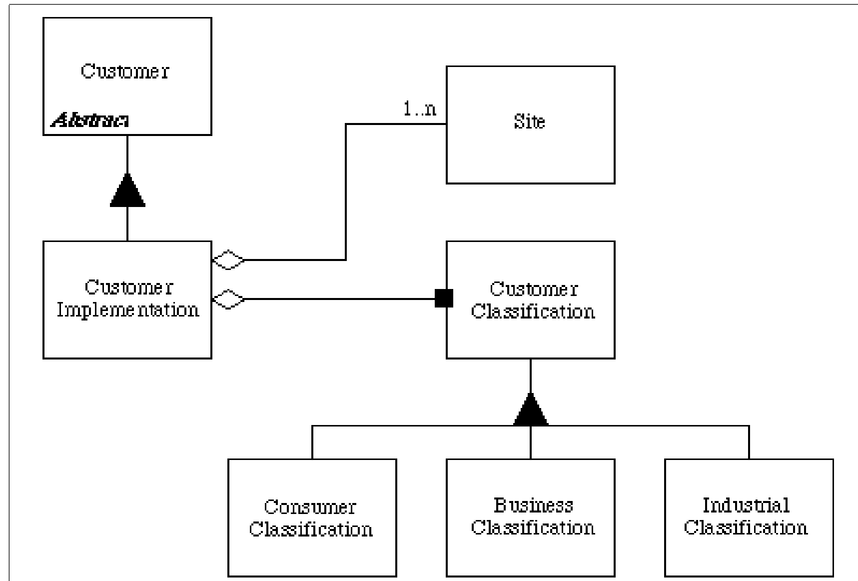

Figure 4
Abstract Customer, Sites and Classifications

```
      Customer* c = *i;
      Invoice inv(c->GetBillingAddress, c->CalcCurrentRate());
      inv.Mail();
    }
```

Although this is extremely simplistic, it shows what we are after. We should be able to write functions that manipulate customers, that have no dependence at all on the type of customer, or the manner in which the rates are calculated. Consider that the above algorithm would not be affected by any conceivable change to the class structure shown above.

The customer class simply delegates the CalcCurrentRate message to the contained ConsumerClassification object for each of the contained Site objects as follows:

```
double CustomerImplementation::CalcCurrentRate()
{
  double rate = 0;
  for (Iterator<Site*>i(itsSites); i; i++)
    rate += itsCustomerClassification->CalcCurrentRate(*i);
  return rate;
}
```

Now, lets look at the structures beneath the various classifications. One thing that the spec makes frighteningly obvious is that all these rate policies are going to change, and change often. Thus, we don't want the class structure above the rate policies to depend upon the policies themselves. So we will apply the principles of OOD to manage

those dependencies. (See Figure 5.)

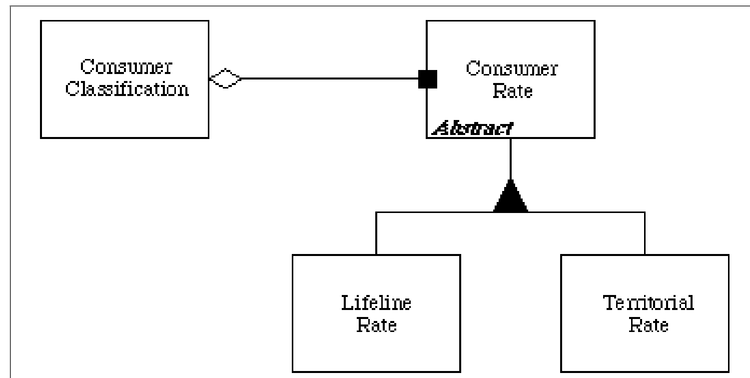

Figure 5
Consumer Rates

Here we see that each ConsumerClassification object will contain an instance of some derivative of the abstract ConsumerRate class. Thus, A consumer can start out using the LifelineRate, and then later change to use the TerritorialRate.

LifelineRate is a simple class. It can be implemented as follows:

```
class LifelineRate : public ConsumerRate
{
public:
  virtual double CalcCurrentRate(Site*) const;
private:
  static TerritorialRate disqualifiedRate;
};

double LifelineRate::CalcCurrentRate(Site* s) const
{
  double rate=0;
  if (s->GetKWH() <= 100)
    rate = s->GetKWH() * .03;
  else if (s->GetKWH() <= 200)
    rate = 3 + (s->GetKWH() - 100) * .05;
  else
    rate = disqualifiedRate.CalcCurrentRate(s);
  return rate;
}
```

TerritorialRate is a bit more complex. The problem here is the determination of territory. It seems likely that these territories will shift around alot. We probably don't want to record a territory into a customer. If we did, then when territories change the whole database would have to be updated. Rather, we probably want to calculate the territory on the fly. The word "probably" indicates that we aren't sure. So we should decouple this decision from TerritorialRate by using another abstract class. (See Figure 6.)

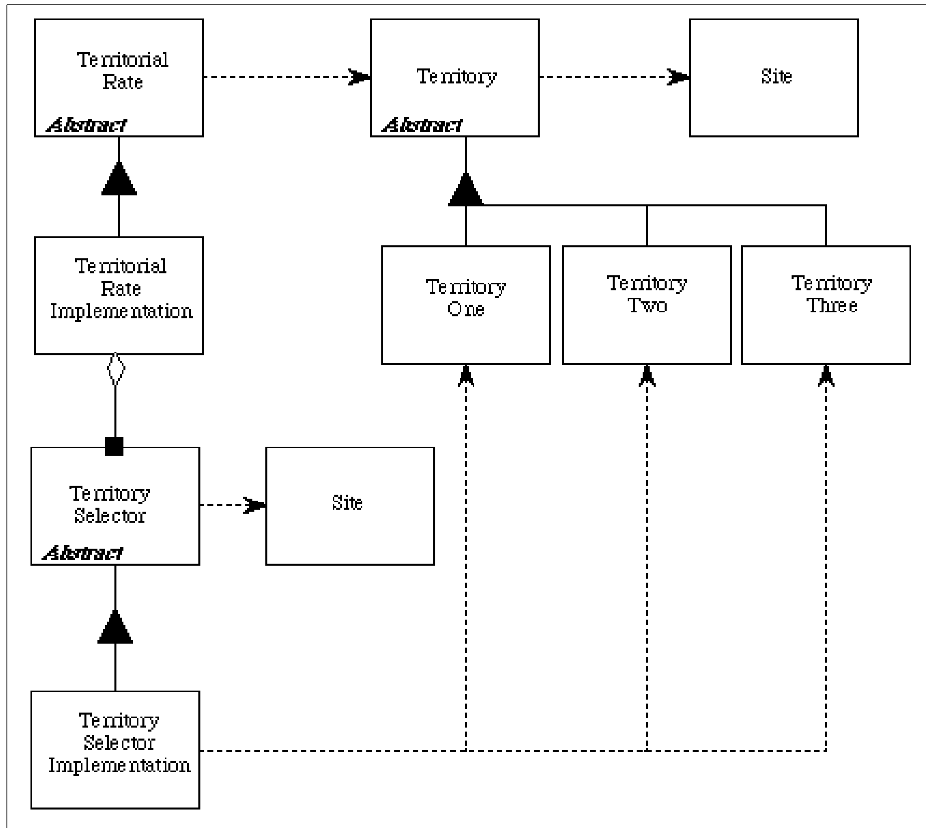

Figure 6
Territorial Rates

The idea here is that TerritorialRateImplemtation makes use of an abstract class called TerritorySelector to choose the Territory that the Site is in. (See the interaction diagram in Figure 7.)

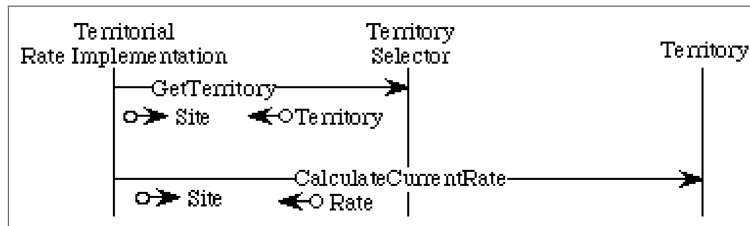

Figure 7
Territory Selection Interaction Diagram

The mechanism by which TerritorySelector determines the Proper Territory is implemented in TerritorySelectorImplementation. This abstraction has been built so that none of the upper levels depend upon that particular implementation. Territories know how to calculate their own rates.

Business classification can be built using a similar pattern. (See Figure 8.)

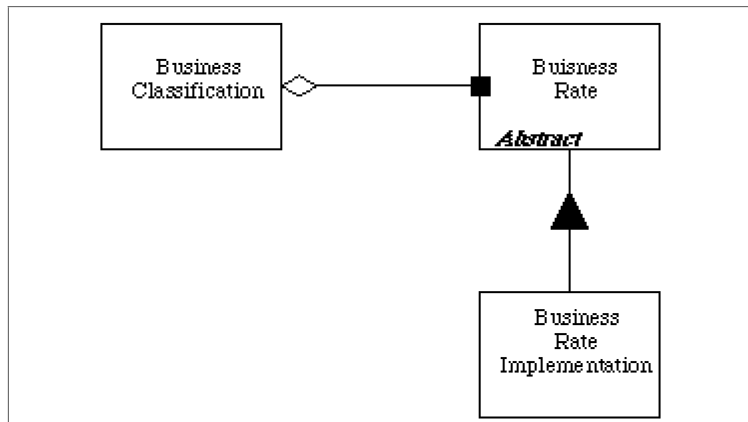

Figure 8
Business Rate

Here the implementation of the sliding scale for businesses is placed in the CalcCurrentRate(Site*) method of BusinessRateImplementation.

Finally, the IndustrialClassification uses a similar structure. (See Figure 9.)

In this case, the CalcCurrentRate(Site) message gets delegated down to the InterruptionPolicy class. The actual calculation of rate is done in the InterruptionPolicy derivatives.

=============

One could complain that there is a lot of delegation in this model. That there are abstract classes that seem gratuitous since their derivatives do nothing more than delegate to still other abstract classes. Consider the path of the CalcCurrentRate method.... It is sent to the Customer abstract class, and is caught by CustomerImplementation which delegates it to CustomerClassification for each of the contained Site objects. In the case of an industrial user, IndustrialClassification catches the message and delegates it to IndustrialRate. IndustrialRateImp catches is and again delegates it to InterruptionPolicy. Finally, one of the InterruptionPolicy derivatives catches it and calculates the rate.

Indeed, this seems excessive. Why not simply have IndustrialClassification contain a pointer to the InterruptionPolicy. Why the extra level of indirection?

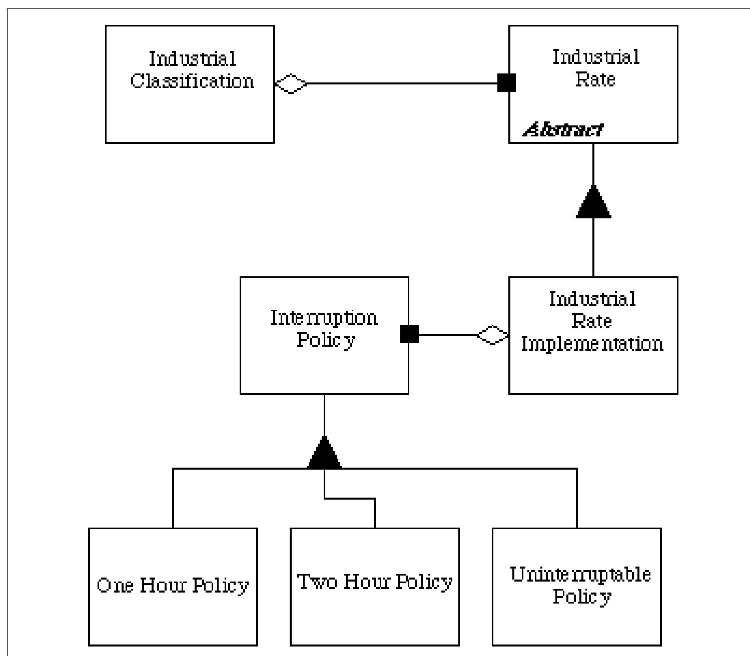

Figure 9
Industrial Rates

```
My jusitification is simply that industrial policy will change.  It
may soon be based upon something very different from an Interruption
policy.  Or it might be based upon an interruption policy AND a
territory.  I have a "feeling" that the "Classification" derivatives
are rather instable.  That there will be lots of "stuff" put in them.
i.e. all the "Industrial" specific data ought to be placed in
IndustrialClassification.  All the "consumer" specific stuff ought to
be placed in "ConsumerClassification".  Thus, I want this "stuff"
protected from the likely changes to rate policy.
```

There were several other people who posted various solutions. There was one particularly interesting solution using Prolog, and another using Forth. Space does not permit me to describe these solutions here; but for those who are interested, they are available through the Deja News service on the world wide web: `http://dejanews.com/`

Steve Mellor's Solution

Steve posted a very large solution to this problem. Much larger than mine. With lots of diagrams and documentation. In fact, it is so large, that I cannot include it all in this article.

However, as voluminous as his solution was, it is not difficult to summarize. In fact, Steve did an excellent job of summarizing it in the last few paragraphs of his posting:

```
The most obvious observation is that there is no inheritance.
This is because I selected the problem so that there was none.
Certainly, my first reaction when faced with Residential,
Business and Industrial customers is to believe that I can
factor the Customer into three mutually exclusive subtypes.
But then I apply some of the rules of S-M OOA.
When I examine the 'types' of customers, all I can see is
names of the different kinds.  The behavior for a presumed
business customer is the same as that for an industrial one.
So they must be the same.

But what about the discount?  Doesn't that apply only to
Industrial customers?  Yes.  But there is no reason why a
Discount cannot apply to any customers.  In any case, the
Industrial Customer does not have any different behavior
caused by the discounting.  All that is more properly
associated with computation of the bill, not the customer.

Several people have remarked (since Robert's solution was
proposed) that the rates change so you should not hard-code
these into the classes.  I agree. Jan Pukite suggested that
these values _could_ go into a data base.  I agree.

The abstraction as in following. [Read each sentence _slowly_
and turn it over in your mind.]  "Hmm.  What is invariant
in this problem?  Well, you get to pay for electricity...
But everyone pays different rates!  True, but everyone pays
a given amount for some # of kwh, you know 9c for 0-200
kwh in Business 42b, and then 7c for the next 150.  OK,
so we could make a rate schedule object...  But the object
has different numbers of attributes according to how many
levels there are.  Augh!  But I could abstract each level
out as a single object with a rate, start kwh, end kwh...
Sounds great.  Let me verify that by reading the rate book.
Looks good!  ARRGH!  There's one that slopes!  AARGH!
Wait a sec, if I abstract the levels, let's call them tiers,
as (x1, y1) (x2, y2) with x as the rate and y as the kwh,
I can just compute this out in Cartesian space.  Alright!

Now, can I think of anything else on which the pricing
varies?  Well, there's the discount.....  Hmm, but that
applies to any account; actually, no, it depends on the
_type_ of account.  If it's 0, that's OK.  That way I can
roll with the punches when a discount is applied to other
kinds of account... OK.  Now let's see if this view holds
up with the rate book expert..."

Note how the rules of S-M OOA cause you to partition the
problem into what Randy Picolet says Grady Booch calls
"canonical objects."  We _start_ from there and aggregate,
only if required, later.

Next we observe that the rate schedules and tiers are passive;
they have no interesting dynamics.  But each time I apply
the schedule or the tier, I get a new charge.  So I can
```

```
have many "computations" according to the schedule, eh?

I have called these objects "Computation" because that is
why they are here.  I could have chosen less attention-getting
names ("Charge under Schedule" and "Charge for Tier", say),
but I want to be clear that this is why they're here:
to compute the charge for the rate and the tier.  If we didn't
need to do that, we wouldn't have them.
```

So, what Steve has done is to generalize the algorithm for computing rates, so that the exact same algorithm can be used for Residential, Business and Industrial customers. This was very clever. However, I had this to say about it.

```
> *IF* the [generalized algorithm] is a concept that the PUC will
> *never* violate, then your abstraction is quite nice.  However, if
> there is a chance that the PUC will come up with a rate that violates
> that concept (i.e. charging based upon the previous 12 months usage,
> etc) then your application breaks in a general way.
```

In other words, I was complaining that Steve's solution depended strongly on the assumption that his generalized rate algorithm was valid; that the PUC would never introduce a new kind of rate that did not fit his generalized algorithm. I continue:

```
> On the other hand, with my solution, any individual change to any of
> the rate policies will cause a change to one small part of the
> application.  The rest remains unchanged and unaffected.  i.e. the
> rest can be linked in from object code, no recompilation is necessary.
```

To this argument Steve eventually replies:

```
I agree with what you say.  If you believe that each Account has its
bills computed according to random criteria, then I too would write
a structure that allowed each type to do its own thing (with the
'help' of inheritance to share apparently common logic at random).
```

To which I respond.

```
Thank you.  Now all we are discussing is the stability of the spec.  I
assumed that it was not stable, and that over time the rates would
indeed be randomly distributed.  (I think this is wise considering the
stress that the spec put on the fact that the PUC would make lots of
changes to the rate book).  You assumed that there was something
intrinsic about the rates that would never change.  And that accounts
for the differences in our two respective models.  Had you made the
assumption I made, your model would have looked rather like mine.
```

Conclusion

I have omitted much of the discussion. There was a fair amount of grunting and name calling and all the other interesting things that transpire on the net. But in the end, it appears that we came to terms. Having begun in very different positions, we managed to finally

agree that, had we made the same assumptions about the stability of the spec, we would have done things in a more similar way.

There is much to be learned from comparative exercises like this. For one thing, I think it is very interesting that the issue of "translation" vs. "elaboration" was barely broached in the discussion regarding this design. (That issue dominates some other threads.) The "translational" aspect of the Shlaer-Mellor method did not seem to play a very important role in determining the structure of the system.