Identity Crisis

by Vladimir Gitlevich
vlad@domaindrivendesign.org

March 14, 2004, conference room.

Steven is at the white board with a marker, a stack of paper in his hand, drawing sequence diagrams, explaining account lookups. There are ten other developers in the room, some nodding, some scrawling doodles, bored.

"...a customer gives the banker his account number. The banker enters the number and clicks OK button. This invokes `LookupAction`. `LookupAction` creates a `LookupBean`, sets its lookup type to account lookup, sets the account number, sets the bean on the `CallSession` and then calls `LookupService`'s `lookup` method. The `lookup` method gets the `LookupBean` from the session. It then checks what lookup type it is, and in this case it is account lookup. So it takes the account number from `LookupBean` and calls `lookupAccount` method, and passes it the account number from the bean. Then we create an XML message request, then Axis calls the message, and it returns a response. The response can contain either a single account with all of the details, or it may contain a list of duplicate accounts. If we get duplicate accounts, we show duplicate accounts dialog where account numbers are shown with owner's names. Banker chooses the right account and we look it up again..."

I interrupt: "What are the duplicate accounts? I mean, why are there duplicates?"

"Because the message can return them." replies Steven, patiently.

"Uhm... Yes, I understand that. But what does it *mean* to have these 'duplicate accounts'?"

"It means the schema that defines the message response specifies that either one account, or many can be returned. If it returns many, we call that 'duplicate'".

Steven is frustrated. I am both frustrated and puzzled: why on earth would there be duplicate accounts in a bank?

March 15, 2004, afternoon, my desk

I have been looking at the lookup code. It is buggy and troubled. Everything is either a bean or some sort of utility object or a Struts action. I can't even begin to understand all this machinery without understanding, say, `Account`. The trouble is that `Account` is implemented as a huge bean with nearly a hundred attributes. So far, I've made sense of about 10. And the only constructor it has is a default one, which tells me nothing.

```
public class Account {

    private double _balance;
    ...
    private String _companyName;
    private String _divisionId;
    private String _accountNumber;
    private String _accountType;

    public Account() {
    }
}
```

Clearly, `Account` is an entity. As an entity, it's gotta have an identity. Is it identified just by an account number? But then, there are the duplicates... I start looking into the messaging code that translates the web service response to this account bean. I notice that whenever an account is mentioned in the message code, in addition to number there are always two other, seemingly random, attributes: type and division id. What are these? Are they a part of the identity? Why is it not just a number? Is it all that difficult to generate a number unique within this bank? Too many questions...

An unpleasant hollow feeling is building up in my gut: I suddenly realize I can't rely on any part of the code. I don't know how it's supposed to work: should I get one account now? Many? Why? It is as if I just discovered that the floor I've been walking on is all eaten by termites and any step could be my last. Yikes!

March 16, 2004, Sergio's office

"Sergio, I am trying to understand the 'duplicate account' thingie. Steven keeps talking about it, I keep seeing it popping up in the code, there is even a method `processDuplicateAccounts` somewhere, yet nobody seems to be able to explain to me what that means. Can you?"

"Well... First of all, let's get our terminology straight. There is no such term as 'duplicate account' in the bank. There are, however, accounts that have the same number but belong to different customers."

"How does that happen? And how do we then differentiate them from one another?"

"First, how that happens. There are two reasons."

Sergio gets up from his chair and seats himself on the edge of his desk. "The first reason is that accounts have types, like checking, savings, money market, etc. We keep adding new types all the time. For us, they are very different products. Each type is numbered separately. So it is a valid business case to have a different type of account with the same number. This is why number alone is not sufficient: you have to specify the type.

The second reason is in how banks grow. Banks grow by acquisition: they buy one another. Now, let's say you have a checking account 123456 in Ittybank, and I have a checking account 123456 in Megabank. Then Megabank buys Ittybank, integrating its customers and their accounts, and here is your scenario again: you and I both wind up having a checking account number 123456 in Megabank. This is why, in addition to the number and type, we also need to track which bank the account originally came from. We call it 'division id'.

So, to answer your question: to uniquely identify an account, you need to specify its number, type and division id. If you just provide a number, you might get more than one account. Which is what they call 'duplicates' - they really aren't. "

"Huh, so this means we've been doing a search by a partial key.... Interesting..." I thank him and go back to my drab cubicle.

March 16, 2004, my desk

Now I know exactly what account identity is. But how do I highlight it among the hundred of other attributes to communicate it to others?

One way is to document it: put the three ingredients of the identity together on top of the `Account` class and write a comment. Another - to create a constructor that takes these three parameters. Which involves hunting through the entire source for the default constructor and replacing it with the new one, ensuring each time that `Account` is instantiated with a meaningful identity. By

hand. This isn't refactoring, this is shoveling. So I'll find all references, go through each one...

Deep breath. I think I am getting stuck in the technicalities at this point. I know that my mind needs focus, a nucleus of thought crystallization, and that when it can't find one, it latches on to whatever it can, typically to the readily available mental scaffolding of *the how*. But *what* am I actually trying to do? The impulse is to quit thinking and start coding. Instead, I step back and play with the language.

I keep hearing myself say “account identity”. Seems like it's a concept we've been missing in our model: “account with this identity” certainly sounds better than “account with this number, type and division id.” Making this concept explicit also gives me the language to explain the “duplicate accounts” in a way meaningful in the domain: “this account lookup is done by partial identity, which is why multiple accounts may come back.” Very nice! Here comes a welcome addition to our ubiquitous language.

An explicit identity also defines `Account`'s invariants much more expressively: you can't have an account with no identity, as it just doesn't make sense. And the identity of such an account is defined as its number, type and the id of the division in which it was opened. Sounds very clear to me.

I create `AccountIdentity` class. This is what I sketch out:

```
public class AccountIdentity {
      private String _number;
      private String _type;
      private String _divisionId;

      public AccountIdentity(String number, String type, String divisionId) {
      ...
}

public class Account {
      private AccountIdentity _identity;

      public Account(AccountIdentity identity) {
            _identity = identity;
      }
}
```

It turns out that `AccountIdentity` is a good place to validate things like account number, as there are some business rules determining what it should look like.

The next step is to override `equals` (and `hashCode`) on `AccountIdentity`, which is what I do. Although it is very straightforward (if the attributes of identities are the same, these identities are equal and identify the same account), there are lots of rules defining exactly how to do that: account numbers need to be normalized, types mapped before comparison, etc. So it turns out rather complex, and once again I am glad to encapsulate all this complexity here.

On to `Account`: why not just implement `equals` on `Account` by delegating it to `equals` of `AccountIdentity`? Right?

Uhm. Not that fast. Account is an entity. And thus the question I need an `Account` object to

answer is “are you referring to the same account as that other `Account` object?” That is, `equals` doesn't make sense on entities: accounts don't equal each other, but two `Account` objects may either refer to the same account, or to two different ones. What does make sense on entities is `sameAs`.

So I should do the right thing and implement `sameAs`. And what do I do with `equals`? Leave the default implementation? Override it and have it always return *false* to express the idea that no two accounts are equal? Have it throw an exception?

On the other hand, I'll want Java's collections framework and a lot of other machinery that make assumptions about `equals` to behave. This requires a contract between `equals` and `sameAs`: if one `Account` instance refers to the same account as the other, they are equal. And to express this contract I could implement `sameAs` and then delegate `equals` to it...

Either way, for any of this to make a difference I'll have to bring up the subject of sameness of entities in a meeting, probably to everyone's annoyance: my feedback has been that developers are busy developing, and spending time on “that Vlad's refactoring stuff” is a luxury they can't afford. Oh well: after all, `Account` has barely graduated from the “bean” category...

In the end I just override `equals` and put identity comparison there: for now. And instead of talking in the meeting, I decide to use a subtler tactics: from now on, whenever I talk about account identity, I will use the language of sameness instead of that of equality. I hope this will make it stick...

March 23, 2004, my desk

It took me more than a week to implement what I wanted (there were 347 references of `Account`'s default constructor in the code). Not only does it now look tighter and more concise, but I found some problems in code while implementing it. This is the original code:

```
Account account = new Account();
account.setNumber(number);
account.setType(type);
account.setDivisionId(divisionId);
```

No invariants are enforced.

This is what I refactored it to:

```
Account account = new Account(
      new AccountIdentity(number, type, divisionId));
```

It turns out that some of these problems were caused by somebody having forgotten to set one of the identity parts on the account, and then something else would throw an exception because a necessary part of identity was missing. With the account identity communicated in the new way, such omission would have not been possible.

My gut begins to relax.

Now I am thinking of making lookups more explicit by introducing `AccountRepository`. It

will make the lookup code even more expressive:

```
List accountIdentities = repository.lookupAcccountsByNumber(number);
// banker picks the first one from the list
AccountIdentity identity = accountIdentities.get(0);
Account account = repository.lookupAccountBy(identity);
```

January 2005, meeting room

Sergio and I are discussing a functional specification he wrote for a project that will allow customers to transfer funds between an account in our bank and an account in another bank.

"...in order to perform this transfer," says Sergio "we need to identify both the source and the destination accounts. We know how to identify an internal account. But to identity an account in another bank, we'll have to support another form of account identity. It is kind of the industry standard, actually. It is defined by account number and the bank's routing number (RTN). Do you... do you think it will be difficult to do?"

"No, it won't: we'll just permit the identification of an account in this new way. Remember how you helped me figure out what account identity was? Since we kept talking about it, I decided to capture this concept in a class of its own. So we're just going to add some new smarts to this class to support RTN. The rest of the application should be unconcerned: account identity will still mean, look and act the same. By the way, is there a mapping between our bank's division id and RTN?"

"Yes, there is... Why is this important?"

"Because this mapping will let us compare accounts defined either way..."

June 2005, somewhere between cubicles

I am walking past Steven's cubicle. With office privacy being what it is, I can hear him instructing Ann, a freshly hired developer. Steven is saying something about account identity. Overcome with curiosity, I stop, lean over the cubicle wall and listen in. We nod hello to each other and they continue.

"Let's say a caller gives the banker an account number" says Steven. "The banker enters this number and winds up with one of the following scenarios: the application finds one account, or it finds more than one account. The reason for the second scenario is that we need more than just a number to unambiguously identify an account: we need its complete identity. Do you know what constitutes account identity?"

"No. What?"

"It is the following three things: account number, account type, like checking or savings, because a checking account can have the same number as a savings account, and its division id. The division id is the id of the bank's division where the account was originally opened..."

Now we're talking! With just this one concept made explicit, the focus has shifted from *the how* of the infrastructure to *the what and why* of the domain.

Hearing the new ubiquitous language echoing back, propagating on its own is eerie and exciting. I don't want to disturb this. I quietly walk on down the hall with a happy smile on my face. The identity crisis is over.