$Id: sg_rewards.html,v 1.3 2003/04/09 22:26:32 jhannah Exp $

Core, Select Guest Rewards, and You

This is a case study on how the transformation of a section of our Select Guest rewards software from procedural into object oriented programming has vastly improved code modularity, readability, consistency, scalability, and documentation (without even writing any).

We'll explore this case study in a series of programmer meetings. In those meetings I hope to give everyone a concrete example of how Core works and the inherent benefits that are gained by using it. Feel free to express all your questions, doubts, concerns, and confusions. If this is your first concrete exposure to OO development all of those reactions are perfectly normal and expected. OO is not an easy thing to get used to, but once it's in your toolbox you'll be glad to have it at your disposal.

(Of course, any suggestions for further improvement are especially welcome!)

We're taking our first steps from the "way we've always done things" to the "way we're doing things from now on." As these explorations, debates, and dialogues unfold, I believe you'll be convinced and excited about the future of software development at OmniRez. I know I am. *-grin-*

Jay Hannah, 4/09/2003

Table of Contents

The Task at Hand

Select guest rewards? What's that?

Well, there's a bunch of documentation: http://omniweb-omaha/resmis/docs/rewards/

To summarize, the point of our select guest rewards delivery software is to send mileage allocation reports (files) to every airline we are affiliated with, each in their own formats, delivery mechanisms, timelines, and miscellaneous administrivia.

As with any software development, we must always think in terms of code modularity. What are the factors involved in this software?

1. Factors common to **all airlines**
    1. CRS data structure
        1. Reading
        2. Updating
        3. Error handling
    2. Internal notification
    3. Scheduling, maintenance windows
    4. Logging / error handling
    5. Omni security precautions
        1. Internal audit notifications
        2. NEVER send more than X miles!
2. Factors specific to **each individual airline**
    1. Output formats
    2. Delivery scheduling
    3. Airline notification (Email? FTP trigger files?)
    4. Delivery mechanism (FTP? Email?)
    5. Security (FTP IPs and logins? PGP encryption?)

The Source Code

Procedural

- http://cvs/source/specific/sguest/perl/rewards/deliver/rewards-deliver-aa.pl v1.22
- http://cvs/source/specific/sguest/perl/rewards/deliver/rewards-deliver-dl.pl v1.13
- http://cvs/source/common/perl/SG_rewards.pm v1.7

Design strategy:

1. All AA specific code goes in rewards-deliver-aa.pl
2. All DL specific code goes in rewards-deliver-dl.pl
3. All XX specific code will go in rewards-deliver-xx.pl
4. All common code goes in SG_rewards.pm

Object Oriented / Core

- http://cvs/source/MVC_dev/Model/omares/crs/Airline_Rewards.pm v1.6
  - http://cvs/source/MVC_dev/Model/omares/Prod/fg_rewards.pm v1.15
  - http://cvs/source/MVC_dev/Model/omares/Prod/fg_stay_summry.pm v1.15
  - http://cvs/source/MVC_dev/Model/omares/Prod/fg_master.pm v1.12
- http://cvs/source/specific/sguest/perl/rewards/deliver/rewards-deliver-aa.pl v1.24
- http://cvs/source/specific/sguest/perl/rewards/deliver/rewards-deliver-dl.pl v1.15
- http://cvs/source/common/perl/SG_rewards.pm v?????? ???????

Design strategy:

1. Airlines_Rewards handles all CRS data operations (examinations, business rules, modifications, output)
   1. fg_rewards handles table I/O
   2. fg_stay_summry handles table I/O
   3. fg_master handles table I/O
2. All AA specific code goes in rewards-deliver-aa.pl
3. All DL specific code goes in rewards-deliver-dl.pl
4. All XX specific code will go in rewards-deliver-xx.pl
5. All common procedural code goes in SG_rewards.pm

Cracks in the Code

Borrowed from Tanja de Bie [1]:

> *It seems most Americans know the story of the Dutch boy who saved the Netherlands from a flood by putting his finger in the dike till help arrived.*
>
> *Not surprisingly the story is little know in the Netherlands. Any Dutch child could tell you putting your finger in a dike on the verge of breaking is not a smart move. The dike would still break and you would drown. Try sandbags, lots of them. Still the story is an echo of the historic importance of dikes in the Netherlands.*

Discovering a Problem

If we accidentally sent a million miles to an airline that would be bad. Really bad. Buying airlines miles costs Omni money. Real, cash money. A million mile mistake would cost Omni a lot of real, cash money. That would be bad. Really bad.

So how do we stop this mistake from happening? Easy, right? We just ensure that our programs which send airline miles can never send too many miles. All we need is a simple test like

```
if ($miles > 500) { die "AAaaa!!! Mileage exceeded!!"; }
```

and we're all set. Bang, done. No sweat. Right?

Oops. There's a problem. The files that we send the airlines are these huge ugly fixed width things. Our output is controlled by complex sprintf statements like this:

```
my $record = sprintf('%1s%3s%5s%7s%-20s%-1s%04d%02d%02d%04d%02d%02d'.
   '%03d%07d%05d  %-4s %-10s%-10s%-15s' ,
   $posting_transaction_type,    # %1s
   $partner_id,                  # %3s
   $service_term_code,           # %5s
   $airline_ff_id,               # %7s
   uc($airline_last_name),       # %-20s
   uc($first_initial),           # %-1s
   $arryear, $arrmonth, $arrday, # %04d%02d%02d
   $depyear, $depmonth, $depday, # %04d%02d%02d
   $length_of_stay,              # %03d
   $miles,                       # %07d
   $prop,                        # %05d
   $cro_resno,                   # %-10s
   ''                            # 15 space filler, %-15s.
  );
```

No biggie. But what if `$airline_last_name` is longer than 20 characters? What does `sprintf("%-20s", "J" x 30)` return? The answer is that it returns a string of 30 Js, not 20. Uh oh. If fields before `$miles` are longer than their sprintf format we're in trouble -- our entire output line could get shifted over and we could effectively send mileage to the airlines which is not the value of `$miles`, circumventing our 1-line safety net above. Doh.

Solution? After creating `$return` above, we need to re-parse it for the `$miles` value and run our safety net tests on **that** value. Phew, I feel better. We know what to do.

Implementing our Fix Procedurally

Alright, we've got a problem, we've got a solution -- let's implement it!

How would we implement this is in our procedural source code? For my demonstrations, I'll walk you through all this. For the purposes of this HTML document, I'll just outline the discovery process here. Don't worry if you can't follow along with this list, just read it to get the general idea.

1. Ok, looks like our sprintf's are in
    1. `rewards-deliver-aa.pl::get_new_aa_posting_record` and
    2. `rewards-deliver-dl.pl::get_new_dl_posting_record`

    We'll add checks there.
2. But wait, these checks should be the same for AA and DL and XX, and the 15 other airlines we want to add over the next 5 years. I should have a central routine.
3. No sweat, I'll just write a central routine in `SG_rewards.pm`. It'll be smart enough to know what airline I'm talking about and check appropriately.
4. Cool, all done. But now my program aborts if any miles allocation is out of whack. It would be a lot better if I could error handle individual allocations and still deliver all the other allocations. It seems I want to add custom error handling for `$miles` too. No sweat.
5. Where is all my error handling done? Hmm... looks like both the AA and DL programs (and XX and the 15 other programs we'll be writing) all have subroutines
    1. `generate_delivery_and_update_fg_rewards()` and
    2. `check_and_append_reward()`

    which contain all the checks and error handling. Cool, let's dive in.
6. ... ??? ...
7. What? I don't understand this code at all! I mean, look at it! At a glance, I have no idea what this does. After studying it for an hour and reading all the subroutines, I understand what it does now, but I've got a raging headache and I haven't even begun to add my new tests. Once I add my tests, it'll take me hours to make sure my flow control is right...
    - ... and what happens when I have to add another test? I need **more** code? There's already 172 lines here, I'll need more?!
    - ... and this is only one airline! I need to do all this again for DL, then again for XX, then again for each of the 15 other airlines we'll be adding? That's thousands of lines of code just for flow control!
8. ...
9. Ok, stop. Deep breath. Walk around a little. This isn't working. There has got to be a better way.
10. Why is the code out of control? What went wrong? Let's fix the problem.
11. Here are some observations:
    1. `check_and_append_reward()`
       I have to pass 6 arguments to this each time? A database handle, a file handle, a scalar, a reference to a hash of arrays, a scalar, and a hash reference? Ouch! I can't remember all that! I'm going to be copying and pasting a lot of code.
    2. `determine_omni_status_for_this_delivery_to_airline()`
       Why does it take 6 lines of code **just to CALL** the subroutine that determines reward status? What a pain to have to keep track of and pass 6 arguments over and over again!
    3. ```
       my $sth = prepare_updatestr($dbh, '020');
       ...
       if(!defined $sth->execute('200', $sequence_number, $fg_id,
           $fg_reward_hashref->{'cro_resno'})) {
       ```
       Wow. That's some crazy black magic! On the one hand, I have to hand it to the author for creativity. On the other hand **I need to get this change out**, and it took me 20 minutes just to understand what's going on! I can't copy and paste this methodology dozens of times into multiple programs!
12. It seems my subroutine calls are all strangling themselves in massive, custom argument lists (of nested complex data structures), each different from the last. How do I centralize all these pieces of information so I don't have to pass everything around all the time? I don't want to make a bunch of globals. Even knowing what to try to put in SG_rewards is getting really hard... uhhh...
13. ...
       ```

Implementing our Fix w/ Core

What if

```
Omni::SG_rewards::determine_omni_status_for_this_delivery_to_airline(
   $dbh, $fg_id, 'AA',
   $fg_reward_hashref->{'airline_ff_id'},
   $fg_reward_hashref->{'airline_first_name'},
   $fg_reward_hashref->{'airline_last_name'});
```

could be written as

```
$reward->omni_status;
```

and

```
process_valid_los(
   $fg_reward_hashref->{'arrival_date'},
   $fg_reward_hashref->{'depart_date'},
   $fg_reward_hashref->{'cro_resno'})
)
```

could be written as

```
$reward->los;
```

and

```
get_new_aa_posting_record(
   'OM001', $fg_reward_hashref->{'airline_ff_id'},
   $fg_reward_hashref->{'airline_last_name'},
   substr($fg_reward_hashref->{'airline_first_name'}, 0, 1),
   $fg_reward_hashref->{'arrival_date'},
   $fg_reward_hashref->{'depart_date'},
   $fg_reward_hashref->{'miles'},
   $hid_hashref->{$fg_reward_hashref->{'prop'}},
   $fg_reward_hashref->{'cro_resno'},
   $partner_id
);
```

could be written as

```
$reward->aa_posting_record('OM001', $maximum_miles);
```

and would automatically re-parse the output string for illegal mileage for you?

How much easier would 15 rewards-delivery-XX.pl programs be to maintain?

What if you didn't have to worry about

- manual table I/O
- `*_ar` tables
- `sequence_number` fields
- `who_stamp` fields
- `when_stamp` fields

- impacting business rules you're not even aware of?

What if dozens of powerful features, developed by other programmers, were immediately available to you to use or ignore at your discretion?

Welcome to the promise of Core.

Let's study our new (OO/Core) source code for a moment.

Table I/O is gone from our delivery programs. Now we're using the `omares/Prod/fg_*` classes to manipulate these tables. Here's the kicker: **I didn't have to write those classes.** In Core, you don't ever have to write table classes -- Sean wrote a program that generates CRS table classes automagically. In Core, all table I/O is handled automagically for you. Everyone and everything does table I/O the same way. When features are added, all systems can enjoy the benefits.

Airline_Rewards is a "container" class which holds my table classes for me. I just call my constructor:

```
my $reward = new Model::omares::crs::Airline_Rewards;
```

And now my magical `$reward` object has all relevant reward data in it. I don't have to pass `$reward` a complex list of arguments, it has embedded within it all the data that I need -- conveniently stored in CRS table format. (Note: OK, first I have to `$reward->load()` my class, but I'll save those details for the presentations.)

In fact, my magical `$reward` object can now do all sorts of crazy things:

```
$reward->aa_posting_record;           # Record in exact American Airlines format.*
$reward->dl_posting_record;           # Record in exact Delta Airlines format.*
$reward->los;                         # Length of stay.
$reward->omni_status;                 # What should the new status be?**
$reward->change_omni_status(200);     # Change Omni status to 200.***
$reward->xml;                         # Record in XML format!****

*    Runs string validation tests (read_string_for_miles_ok()) for me!
**   Runs all validation tests and jumps through all business logic hoops for me!
***  Updates fg_rewards and inserts into fg_rewards_ar for me!
**** Just for fun: By using Core, we've magically inherited Sean's XML
     output routines (written for Passkey and Mercury)! Why would we want
     that? We wouldn't. But it's there just in case we would want to some
     day. Are you excited about Core yet? -grin-
```

What we've done is abstracted all table I/O and related business logic into a 100% re-usable class. What's left in the delivery programs is logic specific to delivery for that specific airline (like what file header AA wants on their file). What's left in `SG_rewards.pm` is the procedural code that doesn't use CRS data and is used by multiple programs (like how to FTP a file to AA).

The code works. It'll be orders of magnitude easier to add new tests and airlines to now. As Core evolves, the toolset at it's disposal will continue to grow, ready to be used if rewards delivery ever chooses to.

This document has just scratched the surface -- we'll dive through all the gory details in our meetings.

Lines of Code

Is our new solution more source code or less? One factor in answering this question is that a lot of the new source code is template code -- code that I didn't have to write to use.

	old	new	template
rewards-deliver-aa.pl	838	284	-
rewards-deliver-dl.pl	408	158	-
SG_rewards.pm	391	370	-
Airline_Rewards.pm	-	343	40
fg_rewards.pm	-	0	200
fg_stay_summry.pm	-	0	210
fg_master.pm	-	0	250
	1637	1155	700

If you don't count the template source, the new solution not only has all the previous benefits, but is also a 30% code reduction. If you do count the template source, the code base grew by 13%.

Conclusion

Perhaps this document helps you start to see the benefits of Core. Perhaps it has raised more questions than it has answered. Either way, I look forward to exploring this use case in great detail with you in our meetings. Hopefully this has given you a head start on some of the high level information you'll need as we continue to pursue Core more aggressively.

Enjoy the voyage!

Jay Hannah, 4/09/2003

Footnotes

[1] http://ac.roleplayingguild.com/november1998/ac111998301.cfm

---

-eot-