From: www.cio.com

How to Avoid Cloud Code Breakage, Part 2

– David Taber, CIO

October 19, 2011

Last week, we covered how to evaluate code that is developed to extend cloud applications. Now we're going to take a look at coding and system modification strategies that can make the system more fragile over time. Since CRM systems have requirements that seem to evolve endlessly, durability of your code is a key success factor to long-run success of these systems.

But before I start, a disclaimer: the examples and terms I use as examples apply to the Salesforce.com environment. Other application environments and platforms use different terms (and even different abstractions), but I simply don't know them as well -- so please don't interpret this as shilling for Salesforce.com.

The Tension Between Declarative Development and Programming

Cloud-based applications have a bias towards what the vendors call "declarative development," because it's unambiguous, easier to learn, and easier to control in a SaaS environment. So most cloud applications will make heavy use of validation rules on fields, constraints on fields and tables, and workflows on objects. This works fine in the early days, and provides a surprising amount of power and functionality.

The trouble comes when you add code, particularly anything that creates new records. When developing that new trigger, class, or integration "listener" service, the coder may be working in a development or sandbox environment that isn't configured identically to the production system. So when pushing the code to production, all kinds of error conditions are thrown -- and they may not be reproducible in the dev environment. Unfortunately, the error messages can be very ugly for users -- and they may not even provide many clues for the troubleshooter.

So, the first set of tips:

1. Make sure that developers work in a recently-refreshed "sandbox," so they aren't surprised by the configuration of the production environment.

2. To the degree possible, enable integration adaptors and other plug-ins in the sandbox, so that developers can see the ramifications of state changes (particularly the "reflection" of error conditions back from outside sources).

3. Once you start developing extensions and functionality around an object, remove all validation rules and re-implement them inside your low-level code so that you can predictably trap and control error conditions.

4. In a similar vein, replace any workflows that do field updates with low-level code implementations.

5. Create an administrative policy that makes it very difficult to create new validation rules or workflow-driven field updates.

6. Make sure you put code in to provide guard-rails in front of field or table constraints, essentially pre-checking the values to ensure they don't hit any walls.

7. Check every field for NULL, and check every set, list, or map for EMPTY before you try to use it in any logic (yes, even any error- checking logic).

8. As mentioned in Error Handling in the Clouds, write classes that handle all application errors in real time send them as messages to centralized error-logging services elsewhere in the cloud.

Table-Driven, Up to a Point

Everyone knows that hard-coding values in the middle of a class or trigger is a bad idea, and everyone should be putting such parameters at least in static-final variables in the declarations section of each module. Even better, move those variables into a lookup table or resource file that is loaded on every run of the code.

Even though databases like to be normalized and nearly anything can be made into a lookup, it's quite easy to be overly abstract and generic. Excessive pointer-chasing leads to lowered understandability by anyone who wasn't the original developer, and can slow the app down (or even push against a cloud environment's governor limits). So, the next few tips:

9. Do put configuration parameters (such as pick-list values, allowable states, or configuration options) into lookup tables. Do include a comments column in each of these

tables, with remarks that a human can read to understand the semantics, behavior, and update history of the table and values. If your cloud system supports it, keep this table in memory ("custom settings") vs on-disk for lower latency.

10. Do put these look-up tables under configuration control. At the very least, lock down the access and make sure that these tables are backed up regularly.

11. Don't be lazy with the table and field naming -- this is where sloppiness upfront costs you at troubleshooting time. (One example, of a table named "folly," comes to mind.)

The Cloud Demands Agile, XP, or TDD Coding Styles

I don't know of a cloud environment that actually precludes large modules, waterfall development, or excessive nesting/branching. Which means, you'll find examples of those every once in a while. But once and for all, we need to jettison these styles in order to foster solid, lasting code.

12. Objects are not just for UI. They're there to support understandability, re-use, and refactoring. But don't go nuts: make sure your objects really do support understandability -- or none of the other benefits will occur.

13. Keep modules small, simple, and separable. Check out the KISSS Principle, which also leads to easier test and debug.

Don't Bump Up Against Platform Limits

Cloud platforms may impose governor limits for certain kinds of operations, such as database queries or in memory table-creates. When you first develop a bit of functionality, make sure that you aren't burning more than 50% of those maxima in your initial release. It won't be long before you have new requirements and fire-drills that will mean you'll need even more of those resources.

14. Use in-memory caching of data ("bulkification" and "dynamic SQL") rather than hitting the database every time. Use future and batch classes to handle bulk workloads and data sets.

15. Make sure your test code gets to 100% code coverage unless there's a solid engineering reason why it can't. Perform real tests of logical outcomes (using asserts on positive and negative test cases), not just idle code exercises. And don't pad code with no-op statements to artificially drive up coverage statistics.

David Taber is the author of the new Prentice Hall book, "Salesforce.com Secrets of Success" and is the CEO of SalesLogistix, a certified Salesforce.com consultancy focused on business process improvement through use of CRM systems. SalesLogistix clients are in North America, Europe, Israel, and India, and David has over 25 years experience in high tech, including 10 years at the VP level or above.

Follow everything from CIO.com on Twitter @CIOonline.