# ETL Doctor

# SAP BusinessObjects Data Services

# Team Development and "Continuous Integration"

*Jeff Prenevost*
*2011-11-08*

## Introduction

In software development projects with multiple developers, things can go astray with disconcerting ease. This is as true for team development in SAP BusinessObjects Data Services (BODS) as for Java or C++. If Tom makes two week's worth of changes to his code units – which all work fine for him – and Kathy does likewise, when it comes time to integrate their pieces together, they will likely begin their descent into "integration hell." To avoid hell requires religious adherence to a methodology that guards against the divergence of parallel streams of development. Our recommendation is for BODS development teams to follow the practice of **Continuous Integration**.

In a nutshell, Continuous Integration prevents divergence by requiring full daily builds of everybody's committed code – and of required daily code commits. We didn't create Continuous Integration -- it's an accepted methodology with an established literature – but have attempted to add value by translating it here into BODS-specific terms. Continuous Integration informs much of the Automated Continuous Integration Testing (ACIT) facility in the practice of Agile Data Warehousing (ADW) as described by Ralph Hughes in his book of the same name.

Your understanding will benefit by consulting the following background material:
http://en.wikipedia.org/wiki/Continuous_integration
https://jazz.net/library/article/474/
http://www.martinfowler.com/articles/continuousIntegration.html
http://c2.com/xp/IntegrationHell.html (this one is pretty funny)
http://www.martinfowler.com/articles/originalContinuousIntegration.html -- a foundational article

## Continuous Integration - General Principles

Continuous Integration enjoins the development team to adhere to the following principles:

1. Maintain a Single Source Repository.
2. Automate the Build.
3. Make Your Build Self-Testing.
4. Everyone Commits To the Mainline Every Day.
5. Every Commit Should Build the Mainline on an Integration Machine.
6. Keep the Build Fast.
7. Test in a Clone of the Production Environment.

8. Make it Easy for Anyone to Get the Latest Executable.
9. Everyone can see what's happening.
10. Automate Deployment.

These principles work, but they need to be translated into terms and technologies specific to BODS. In the following, we're ignoring testing, both in general and as handled in an ACIT facility, as well as how to automate certain operations; both are simply too large to be addressed here, where the focus is more on team dynamics and do's/don'ts.

In what follows, we'll use "commit" to mean "check-in", and the terms "mainline" and "code base" will be synonymous with the latest version of the job or jobs in the BODS central repository.

## Continuous Integration in BODS

### 1. Use a Single Central Repository and Strive to Keep Everything There

In a team working on a single project, we have to have a single DS central repository for code which contains everything necessary for a "build." We recommend that teams create a central repository "BODS_CENTRAL" for the purpose.

There's a lot of talk about "builds" in the literature. In DS terms, what runs is the job, so, for us, "build" = job. Some people say it should be a project, and that would be OK – the "build," then, would be the set of jobs. This doesn't affect the general discussion.

We can have multiple jobs in a single DS central repo, and those jobs can use shared components (like custom functions and tables), and that's all fine as long as we have a single central repo.

We do not recommend creating multiple central repositories for different code life cycle phases (typically, dev, QA, and prod). Central repositories intrinsically maintain versions, and we can use the code labeling feature to label our code with version numbers if desired.

Martin Fowler emphasizes that the repository must contain everything, and it should be possible for a developer to start w/ a virgin local repository, get the latest job, and run it successfully, with no external dependencies. A Data Services central repository is not a full-featured, Subversion-style repository, and we can't easily add Word documents, DDL scripts, etc. But by properly documenting the code that *can* get added, we can, at least, refer to such dependencies, and make our jobs self-documenting and self-contained to the maximum possible extent. The general rule should be: by performing a 'Get latest version' of a job, *everything* required to run the job should be either directly present or referred-to within the job.

We encourage the practice of using BODS to create tables, vs. doing that with a modeling tool and DDL outside of BODS. Special "setup environment" or "create tables" jobs can be created, using template tables as the target, in lieu of external DDL scripts, and this helps keep everything self-contained in the central repository. BODS jobs can also be written to be self-checking and self-initializing, running scripts that check for the existence of objects (typically DBMS tables) and conditionally taking action to initialize those tables. Where you have need for advanced logical or physical data modeling, this won't work, or will only work partially down – it would be a stretch to write lots of advanced DDL script and execute it from BODS (although, yes, you could) – but for many purposes, regarding tables, all you need is the table and a primary key, which a BODS template table will handle.

## 2. Centralize and Standardize System Configurations

System configurations are not, unfortunately, objects we can put in a central repository, but they need to be treated like "code." Designate a person to be the "keeper" or manager of system configurations, data store objects, data store configurations, and substitution parameters, which all work in concert. Post the ATL files for system configurations and substitution parameters in a central network share. Each developer on the team should, every day, do an import of the latest official system configuration and substitution parameter ATL files from this share.

## 3. Everybody Starts Fresh on Everything Every Morning

Each developer must start the day, each day, every day, by performing a 'Get Latest Version' of the job (or jobs) in question and all dependents, for any job or jobs the developer intends to work on that day. Each developer should also import the latest ATL files for the system configurations and substitution parameters.

The point of 'continuous integration' is to continuously (at least daily) integrate-and-test to avoid *serious* divergence and speed overall efficiency and code quality. Will developers experience unpleasant surprises after doing complete "Get Latest Version" operations? Of course. But the surprises should always be from *recent* changes, and relatively easy to find and resolve. It is always easier to code in isolation *in the short term,* but the short-term productivity gains of ignoring coordination are paid for in spades later. Thus: developers are not allowed to pick-up where they left off on modifications to units which they've had continuously checked-out and uncommitted for days. At the beginning of each work day, each developer must re-align to the "mainline" or "code base," and start from there – from an up-to-date base.

## 4. Developers Always and Only Check-Out Units They Intend to Modify Soon

Once a developer has a fresh code base, they check-out the specific unit or units they intend to modify *soon* – within a few hours. They do *not* preemptively check-out large branches of code, containing a number of units far in excess of what they could reasonably modify within "soon."

If they want to make large "structural" changes to flow units such as workflows and conditionals, and want to perform check-out-with-dependents of the root workflow of a large branch to get everything at once for some major reorganization effort, then (in this typical example) they should immediately undo the checkout for all the dataflows within and any small workflows known to be irrelevant to this high-level restructuring.

Developers should think of checking-out an object as an act of *communicating* to their team members: "Hey, everybody – I'm actively working on this, right now." If you check something out, but aren't working on it, you're misleading and confusing your team members.

## 5. Avoid "Checkout Without Replacement"

The "checkout without replacement" operation causes confusion, because it's almost always used to get changes to a unit uploaded to the central repository in the absence of having properly checked-out the object beforehand.

Let's say that on Tuesday, unbeknownst to each other, Tom and Kathy both decide to work on dataflow DF_ABC, but only Tom remembers to check it out. If Kathy had remembered, she would find that Tom already had DF_ABC out, and would be warned that whatever she intended to do in parallel would need to be manually merged with Tom's changes – remember, a primary benefit of checking-out objects is to communicate. Indeed, Kathy should find something else to do – it makes little sense for her to work on DF_ABC if Tom's got it checked-out and, presumably, is making changes she can't see yet. But Kathy forgets checking-out, and forgets to check to see if anybody else has DF_ABC checked-out, and starts

making complicated adjustments to the dataflow. At around 3pm that day, Tom finishes with his changes and checks-in the code. At 6pm, Kathy is finished for the day, and, satisfied with her changes, needs to get her code committed to the central repository. Only then does she remember that she did the day's work on an object she hadn't checked-out.

What should she do? She can certainly perform a "checkout without replacement" and upload her new version of DF_ABC. But she worked off a "dirty" code base – her coding didn't reflect Tom's changes, which paralleled hers and were committed in advance. Tomorrow morning, when Tom performs a 'Get Latest Version' on the job, his changes to DF_ABC will all have disappeared in favor of Kathy's, and after a round of recriminations and hurt feelings, they'll need to manually piece through their parallel efforts, that is, will need to spend some time in integration hell.

Avoid "checkout without replacement."

If a developer *does* forget to checkout, however, all is not lost. The developer has two options:

1. If the object in question has not been versioned by anybody else since the beginning of the day, then the developer can safely perform a 'checkout without replacement' and check the changes back in. No harm done.
2. If the object *has* been versioned, the offending developer should go ahead and perform a 'checkout without replacement' and check-in, creating a new version, but then immediately use the comparison features in Data Services to see how the two versions differ and work with the other developer to integrate as necessary… expressing apologies.

## 6. Always Test Against the Very Latest Code Before Checking-In

Another way of saying this is "Never knowingly break the current job" or "Never commit from dirty code."

Before checking-in changed units, developers are responsible to make sure it passes both their own unit testing (of course) and "mini-integration testing," i.e., running the relevant jobs successfully, against *up-to-the-minute latest* code. If you work on a given unit till, say, 3pm, you can and should assume that other developers will have committed updates to other units earlier that day. They haven't been made in relation to *your* changes – you haven't committed yet, so they've been working off the version as of that morning – but they've done their part to not break the job(s). Before you check-in your unit(s), you must perform a 'Get Latest Version' of the entire job again, as you did that morning, and make sure your units still work with the changes that have been committed so far from everybody else. (Your code units are in a checked-out state and will not be overwritten by a "Get Latest Version" operation.)

## 7. Everyone Commits Everything Daily

Developers should check-in their changed code at *least* once per day, and preferably more often. Code should not be left in a checked-out state overnight. The divergence that leads to integration hell grows the wider the longer code is modified and not returned to the mainline, and under Continuous Integration, a day's worth of changes is the limit of tolerance for this divergence.

What if a developer checks-out a unit in the morning, works all day to make changes, and still doesn't have it working by the end of the day? Then – in general – that developer is biting off more than they can chew, and needs to decompose the work into smaller pieces.

A developer may not check-in broken code to the mainline – period. If he finds himself at the end of the day with a broken, in-process unit, and there are no smaller pieces of it which can be committed, then he will simply need to make a duplicate of the code and undo the morning's checkout operation on the still-broken unit. He is not allowed to retain it overnight in a checked-out state. In the morning, he'll get a

fresh copy of the unit in question (with no guarantee that someone else may not have modified it during the night), and will need to start afresh on the changes.

## Comments on Testing

Full standards and methodologies around testing are out-of-scope for this document, and building an ACIT facility is, in particular, a very large topic and good-sized project unto itself. Testing relates to Continuous Integration general principles 3 and 7, is intrinsic to success, and deserves full explication and development elsewhere. But a few comments are in order here.

### Unit Tests

Developers need to make sure their unit of code works before committing. What does that mean? What it *should* mean is that a constellation of automated tests surrounding the unit all fail to produce results – that all the tests are empty. This penumbra of tests should represent everything that a pair of developers can think of as relevant to the unit and what it's supposed to do. In ETL development, for testing business rules in particular, this can run to many tests covering all sorts of odd data-driven scenarios. The standard ADW development methodology specifies that these testing programs be conceived of and written before the code itself, as in Extreme Programming, but whether or not that practice is followed, a unit needs to be tested – in some "reasonable fashion" -- and those tests should be automated.

### Integration Tests and ACIT

In Automated Continuous Integration Testing, the latest build is automatically tested through a large, pre-defined set of scenarios, with both a "Data Scenario Simulator" and "Time Series Simulator" engines, all in conjunction with a Test Data Repository (TDR) and a team dedicated to making the whole thing run. "Continuous" can mean anywhere from once per day to every time a commit is made to the mainline, if the tests run fast enough and the commits are infrequent enough. The tests are authoritative, and once the ACIT infrastructure is in-place, integration testing devolves to merely inspecting the results of the latest automated test run results for the presence of any "positives" from the tests. ACIT saves time and, ultimately, money, and dramatically increases quality, but comes at a substantial up-front cost.

In the absence of an ACIT facility, developers or testers must perform integration testing (i.e., "run the jobs") more-or-less manually against more-or-less manually managed sets of test data. In the worst case, results (typically, spreadsheets of data) are given to hapless inspectors who are supposed to scan them and report back on issues.

Whatever the capabilities, in a standard repository landscape, we dedicate the local repository "BODS_QA" to integration testing. This repository is the one in the middle in the typical dev > test > prod code migration path. The latest job or jobs are copied into this repository from which "real" integration tests are then run, bigger in scope than what a developer runs prior to commit, which may be simply a "smell test" run of the job or jobs with a truncated sample data set.