CAS CS 411
Lec 11

Migration Strategies

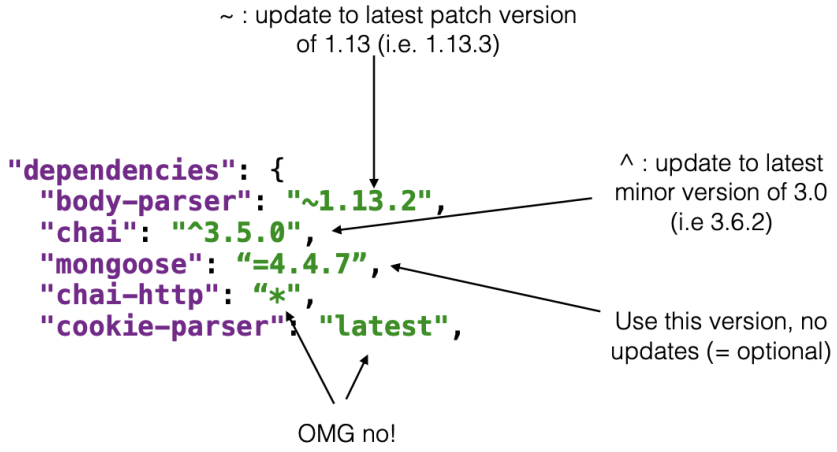
1. Motivation for Migration
    a. Migration involves moving from something that we have to something that we want or need
    b. Sometimes it's due to market pressure; we must remain competitive
    c. Other times it's that our system is old and slow or not flexible enough and is at end-of-life
    d. Maybe someone offered us a great deal on a new system
    e. Migrations can make up a significant portion of the work in many organizations
2. Gap Analysis
    a. In each case we must first establish a baseline - where we are right now
    b. It isn't unusual for a company to not have a good handle on this
        i. Systems grow organically over a period of time
        ii. Staff comes and goes
        iii. The company might have lots of stovepiped apps
    c. The first step, then, is to do a thorough inventory of all of the physical components
    d. Once that's in place we must do the same for the business processes and flows
    e. The goal is to be able to document fairly precisely the current situation
    f. Automation is your friend here…use tools to dump ERDs of databases, create maps of LAN topologies, and so on
3. Future State
    a. The next step is to describe as closely as possible what the future state should be
    b. For a web app, it might be an increased number of users or additional services
    c. Maybe you want to move from a centralized db to a distributed topology
    d. It's important to paint a clear picture of the desired end state
4. The Gap
    a. Now you have two points in time…present and future
    b. The real work is to answer the question, "how do we get from here to there?"
    c. That's the gap analysis
    d. What you want to produce is a strategy, a series of steps, that will move you toward the desired end state in a timely manner
    e. The gap analysis becomes a project plan for the migration
5. Approaches
    a. Generally speaking there are three major approaches to migration
        i. Modify the existing system

    ii. Scrap the existing system and start over (green field)
    iii. Don't touch the existing system, but add an adapter layer to provide new functionality
  b. Which approach to take depends on many factors!
6. Questions to ask
  a. Is the existing system capable of handling the end state in terms of performance?
  b. Will the existing system and its infrastructure still be supported in the future?
  c. Is there a substantial sunk cost in the existing system?
  d. Is the end state really the end or is additional future work planned?
  e. Is there an opportunity to leverage this work to add functionality to other systems?
7. 1. Modify
  a. If the gap is small, and the infrastructure can support the end state, it might make sense to use this approach
  b. Incremental upgrades carry relatively low risk, and it's easy to back out if something goes wrong
  c. A downside - you are usually stuck with whatever the existing technologies are
  d. There might be opportunities to improve existing functionality as part of the upgrade
  e. These kinds of projects make up the bulk of work at most companies
  f. Usually there comes a point when it just isn't economically feasible to continue modifying the existing platform, and so we move to…
8. 2. Scrap
  a. You might have no choice…sunsetting hardware, end-of-life software, or the gap is so large that it makes sense to start over
  b. This approach usually has the most constraints…you typically need to preserve the functionality of the old system while you are at the same time dropping it off a cliff
  c. Even though constrained, there's significant opportunity to future-proof the system by using newer methods and technologies
  d. For green field projects there's a lot of up-front analysis to do
  e. When modifying, you usually can get away with slacking off on the as-built analysis of existing functionality, since it isn't going away
  f. For green field designs each piece of functionality must be recreated exactly and so must be exhaustively documented
9. Testing Green Field
  a. Typically the old system is kept running and work is routed to both the old and new systems

    b. For example, a trading application would feed trading floor data to both systems, operating in parallel, and at the end of the day they should produce identical results
    c. Another approach is to generate data on the old system and feed it to the new system
10. Two Words About Testing
    a. Upstream and downstream
    b. Systems usually don't exist as stand-alones, they must interact with other systems
    c. There will be processes upstream from the new system providing inputs, and processes downstream taking its outputs
    d. Testing the new system requires testing both upstream and downstream processes in addition to the new system
11. 3. Wrap
    a. The third approach is to leave the existing system in place and running, and add a layer that provides additional capabilities
    b. This is becoming more popular, since the sunk cost is preserved and the additional functionality can be provided in just about any technology
    c. We use the facade or adapter patterns to accomplish this
12. A Wrap Example
    a. Say you have a large mainframe that handles back-end account processing for banking customers; it's been there for a decade and works really well
    b. Marketing insists that we'll see a 25% uptick in share if we could only provide customers with a web page to access their accounts
    c. IT says no way, no how, is there any possibility that the mainframe can spit out web pages
    d. The solution?
    e. Wrap it up; add an adapter layer that speaks HTML and Javascript on one side and mainframe no the other
    f. Even better, add two layers…what's the second layer?
    g. …an abstraction layer that use a common language such as XML to frame requests and responses
    h. Why? Now instead of just getting a web layer, for the same price you have a multifunctional adaptation layer that future-proofs the application
    i. Need SOAP messages for a web service instead of HTML? No problem, just add a new config to the abstraction layer
13. Disadvantages of Wrapping
    a. It's not all puppies and roses…for one thing, you've just dug a hole and planted your legacy application firmly in it by building things on top of it
    b. Basically you are gaining in the short term by deploying a simpler solution, but in the long run it might cost significantly more (maintenance, replacement cost, etc)

    c. Performance might suffer either here or downstream…you are asking a system to do more work than perhaps it has been designed for
    d. Sometimes, though, this is your only choice
    e. There might be political reasons to keep the old system up
    f. Maybe you no longer have access to the existing system's code…just the interfaces
    g. Another disadvantage…you are constrained by what the legacy system is capable of doing
14. Data
    a. This will usually be your biggest headache…migrating data
    b. If you are lucky, the old system and the new share a common schema and use the same database vendor…any new functionality is built with new data structures
    c. Normally there's a significant effort involved in moving data from one system to another
    d. Plus, it isn't just the data, it's all the relationships, views, stored procedures, and other internals
15. A Few Approaches
    a. They are basically the same as before…scrap, wrap, or modify…with a few additional quirks
    b. Scrapping almost never is the answer since most of the business' assets are in its data
    c. Wrapping can work in a pinch but really should be used only as a temporary measure, unless the legacy data can be set to read-only
16. ETL
    a. Extract, Transform, Load
    b. ETL tools sit between the legacy and the new database
    c. They let us write rules for transforming one to the other, including relational information
    d. Typically, stored procedures and triggers aren't candidates for ETL since they tend to be vendor-specific
    e. Tools range from free, open-source solutions to commercial products like IBM DataStage ($150,000 per server)
    f. ETL tools allow you to adjust schemas, for example a charstr(50) might become a int(10) in the new schema by just lopping off the last 40 chars
    g. For small projects it might be faster to write your own ETL tool in Python, PERL, etc
    h. Whichever approach is used, extensive testing is necessary to confirm that nothing has been lost in translation

17. Dependencies
    a. Any changes in platform or application might introduce a nightmarish tangle of dependencies
    b. For example, moving from Drupal 5.4 to Drupal 8.1 requires also upgrading from PHP4 to PHP5
    c. PHP5 doesn't support MySQL4.2, so you need to upgrade to MySQL5.7
    d. MySQL5.7 doesn't support the v4.2 style triggers, but 5.5 does, so that's what you go with
    e. MySQL5.5 requires OpenSSL10.2 for secure db connections
    f. OpenSSL10.2 requires libc v9.7a
    g. PHP4 crashes with libc v9.7a
    h. These dependencies exist upstream and downstream as well…we want to avoid having multiple versions of platform software running at the same time
    i. Dependency issues often crop up during integration test, since devs tend to reflexively install whatever it takes to ge the code running and forget to document the changes
18. Here's the Problem…



    a.
    b. At the very least, freeze the packages and then
        i. DO NOT UPDATE them without a valid reason
        ii. If updating, understand exactly what the update does
        iii. AS WELL AS any dependencies the new update requires
        iv. Then re-freeze
19. Backwards Compatibility
    a. Both upstream and downstream processes might have dependencies on the very system that you are changing
    b. If this is the case, backwards compatibility must be planned for to support what are about to become legacy clients
    c. A very simple way to handle this is with an adapter pattern that provides legacy interfaces on one side and translates the calls to the new platform