Eating an Elephant, One Bite at a Time

Christian Buckley

January 13, 2004

---

Return to nature with this very earthy look at managing change. Like an elephant, development projects can be large, intimidating, and hard to handle. But with the right tools (such as a good configuration management tool, or a shovel), you can tame the beast.

The African Elephant is the largest land animal. Males will grow to 10-12 feet tall at the shoulder, and weigh between 10,000 and 12,000 pounds when full grown. The skin of an elephant can be from 0.25 to 1.5 inches thick to withstand blistering sun and torrential rains. Its diet consists purely of vegetation in the form of grasses, tree limbs, tubers, fruits, vines and shrubs. They will spend up to 16 hours each day foraging for the 300+ pounds of vegetation they must consume to meet their nutritional needs. Their digestive system is geared towards processing massive quantities of bulk, of which only 45 percent is actually digested and used. The partially digested feces are an ecologically important method of seed dispersal, and one species of plant actually must be passed through an elephant's gut in order to germinate and grow!

Sizing Your Elephant

You are the Build and Release Manager of a large product that is developed at six locations across the globe (very similar in sheer size to our African Elephant friend above). The current build and release cycle takes two days to integrate, with an additional 24 hours to construct the integrated code. Testing takes another 48 hours, which needs to occur before the Product Validation team checks the soundness of the installation and pushes the final product out to the customer. In a best-case scenario, the complete cycle takes more than five days. The count begins at the time an engineer has checked in code to fix a bug, and completes when the defect has been verified. This effort can be massive and cumbersome. No wonder product schedules slip. No wonder the overhead costs for distributed computing are not realized until the product is ready to ship out the door. Distributed computing has benefits, but if not managed and controlled, the integration and distribution of code and executables can be overwhelming.

Care and Feeding of your Elephant

The skin of an elephant can be from 0.25 to 1.5 inches thick to withstand blistering sun and torrential rains -- and like the elephant; your product can also wear a "thick skin." So how can you decrease product development cycle times while maintaining product quality?

Well, you can decrease the integration costs of distributed teams -- or even non-distributed teams. How? You need to integrate early and often. But with build cycles over five days long, how do you
integrate often? Five days come and go quickly, and it means the Build and Release teams will be spending all of their time building, testing, and releasing the product, with no time to look at process improvement and optimization. It is a cyclical -- and deadly -- problem. It's the Goose and the Golden Egg problem -- you need golden eggs right now, and so you kill the goose -- only to destroy your one and only source of golden eggs. The same is true with Build and Release teams. If they spend all of their time pushing buttons, they won’t find the improvements that you need to decrease build cycle times, and to increase productivity.

Elephants will spend up to 16 hours each day foraging for the 300+ pounds of vegetation they must consume to meet their nutritional needs. That's a lot of vegetation. Do you have that kind of time for your product?

The reward for process improvement is increased quality and decreased development cycle time. But how do you get there? Well, we can think of three methods to manage the build and release of large multi-site development efforts:

• Decrease build cycle times, in order to get results back to the software developers. (Decrease development cycle)
• Decrease test cycle times while maintaining code coverage and use case coverage. (Increase product quality)
• Accurately report results of build and test cycles. (Increase schedule estimation accuracy)

**Build Cycle Reduction Solutions**

In this article, we're going to focus on build cycle reductions. Why only build cycle reductions? Well, like the elephant, whose digestive system is geared towards processing massive quantities of bulk -- of which only 45 percent is actually digested and used, we thought it would be helpful to attack these concepts one article at a time. In our next two articles, we will begin a discussion on test cycle reduction solutions and automated build and test systems. We plan to walk through the typical build and test cycles, and, if that huge government grant comes through, we may even venture into the product build and release cycle.

The first place that most CM managers look for efficiencies is always in build cycle reduction. "How can I get the code to compile and link faster?" Well, there are several different approaches to the problem, such as revamping the make system, buying more hardware to make it go faster, and automating as many manual steps as possible.

**Make system reconstruction**

When looking at revamping a make system, you need to be involved with the software engineering architect. Many times the software engineering architect was the one who originally designed the product, and will have a good idea of the dependencies and methods used to build the product. If the product has some history, you will quickly find out that the code is just short of a sentient life form -- doing whatever it likes, despite the architects' original designs. Okay, enough about religion. The code seems to evolve over time and the dependencies and compilation rules can easily become none consistent and hard to manage. Here are some ideas on how to approach a problem such as this: look for multiple compilations of the same code, component-ize your code, remove circular dependencies, and then decrease dependencies between components.
• **Multiple paths to compile the same code.**
Although everyone knows better, this problem typically pops up when an engineer needs a library in another directory and cannot wait for the next clean build to get the library to test his stuff. Makefiles are changed and inadvertently checked into ClearCase, which then makes it into the build. If you are lucky, the build will break and the problem will be noticed and fixed. Worse -- and most often the mistake -- it is not caught and your build becomes takes that much longer because of the additional time to build that library. You may think that this is fine, since 'clearmake' and 'make' are smart enough to avoid building the target again. Don't be tempted to think that all is well in your make system because of this process. Different environments, targets, etc. can allow the library to be built more than once. And when looking at parallel-izing the code, this problem can literally paralyze your build. How do you find the problem? This is a hard problem to find, but clearmake can help, looking at configuration records is a good place to find how a library or binary has been built. The other approach would be to start looking at makefile and making sure that the system only builds files in their home directory. Most code has some good notion of source-code directory hierarchies.

• **Component-ize your product**
While you are going through all of your make files and directories, you should start to look at the componentization of your code. A component will typically match the directory hierarchy in some fashion. If you have determined that a component spans two or more directories look at putting them together under another directory with the name of the component. It is also okay to have a hierarchy of components and subcomponents. This will make it much easier when you start parallelizing the build.

• **Remove circular dependencies**
If the product you are building has been around for some time, you will see circular dependencies between the components that you have just defined. They typically present themselves as rebuilding the same library twice with different dependencies. Circular dependencies cause your work to act like a car driving 25 miles per hour on a freeway with traffic going 70 mph. It is a stupid thing to do and it takes forever for you to get to your destination -- not to mention every other person driving on the freeway. The easiest method to removing the circular dependency is to create another component upon which the original two components are dependent. All of the shared code should be made available in the new component. Another approach is to place all of the common code in one of the components. This last approach can be problematic for future growth of the product, so be careful.

• **Decrease dependencies**
Using a tool like Rational Rose or Sniff, you can see the dependencies between your components. If your dependency graph looks like a map of San Jose or Los Angeles, you should probably consider simplifying it. This can be done by shifting code from component to component, creating new common components with global dependencies, and creating a hierarchy of components and sub-components. The fewer dependencies you have between components the more parallelism you will obtain and the faster your build will be.

A recent project went through the exercise of reworking a make system that had de-evolved into a mess of circular dependencies. Over a span of two weeks we took a build that averaged 24 hours to less than 8 hours -- just by cleaning up the component structure of the product. The product was
still being built sequentially, and we got a trifold improvement in speed. When we parallel-ized the build, we achieved another trifold improvement.

**Throw hardware at it**

In the last three years, the big UNIX machine shops have started pushing their "Server Farm" solutions. Every engineer and CM'ers dream has come true -- 48 CPUs, 96 Gigabytes of RAM, and 2.8 Terabytes, all in a single rack. Even with more hardware, you can continue having speed problems if you don't address the build system directly. You also need to plan out your hardware configuration -- if not, you will have all that horsepower, but you will still be driving in the slow lane. There are several things to consider for speed improvements: VOB server layout, Compute machines, centralized tools, and network configuration.

- **VOB servers**
  
  With some changes to Clear Case in version 4.x, the lock manager no longer limits VOB servers. The old configurations of more (smaller) VOB servers have given way to smaller numbers of large machines. We have used machines such as the Sun E420R, 4 CPU, 8 Gigabytes of RAM as a VOB server, serving 60 VOBs and over 100 users. The E420 is connected to a fiber channel RAID array of 375 Gigabytes mirrored. With the number of VOBs and users increased, you must look at the network connections to the VOB server, as well. Most of these big servers can handle multiple network cards, or the larger network cards like a gigabit card. Remember, several different machines in your server farm will hammer your VOB server.

- **View Server**
  
  In most cases, we have not been thrilled with view servers. Compilation on the machine that a view resides is generally faster than accessing the view over the net. But when you have a parallel build and a farm of fast machines, you need to have a centralized view server. The network connection to this machine should be fast -- or there should be multiple network cards. The faster card is a better solution, so we recommend a gigabit network card in the view server, as well.

- **Compute Machines**
  
  The purpose of the compute machine is to compile or test your product. You are hunting for pure speed and memory. These machines are typically multiple processor machines with as much memory as you can put into them. The network connection on these machines can be a 100 BaseT network card, which is standard for most servers. Putting faster cards in the compute machines is not a good idea, however, as the VOB server and view servers will most likely become the slowest part of your network connection.

- **Network Configuration**
  
  Get a switch and make it big. Server farms that perform builds are typically very network-intensive. A switch with a big back plane will help the information flow uninhibited. All of the machines should be connected directly to the switch, including the VOB, View, and Compute machines.

- **File Servers**
  
  If you have a server farm that has several users and products you may want to consider one of the large file servers. Most of them come with multiple large gigabit network cards, and fast disks. This can increase your performance and decrease your maintenance.
In Conclusion?

The partially digested feces are an ecologically important method of seed dispersal. Interestingly, there's even one species of plant that actually must be passed through an elephant's gut in order to germinate and grow?

OK, maybe this is not the best way to close out an article, but our point is simple -- large, multi-site software projects are much like the elephant analogy we used throughout this article: they're big, bulky, and natively inefficient. But there are ways that you can make them more efficient -- train them, if you will. And amid all those systems and all of that bulk, there is that seed that germinates and grows. That is your project -- your software.

But you should probably wash your hands after handling.