Software Realities

Editor: James Bach, 1198 South Fork Dr., Front Royal, VA 22630; voice(540)631-0600; fax (540) 631-9264; j.bach@computer.org

Improving The Development System Model

James Bullock, Rare Bird Enterprises

My impression is that successful projects belong to the same people, the kind of people who recognize problems and deploy countermeasures more effectively than others. In order to be successful at what they develop, these people apply far more than a single model not only to detect problems and identify countermeasures, but to plan. One of these models—a model that we all employ, consciously or not—I've called *the development system*.

Let me give you an example. I had the opportunity of working on the BSY-2 combat system for the SeaWolf submarine. This system consisted of millions of lines of code and thousands of interfaces, and it required that we provide and maintain rather large documents to describe it. I worked in the tools group that kept several of the design specifications in relational databases.

Code and design elements were re-used, resulting in a huge number of many-to-many relationships in the system's design specifications. A single change would impact many elements, from several sources. Manually grubbing though the interface specs (dozens of feet thick and housed by multiple contractors at multiple locations) would have been too slow and prone to error. Keeping the specs in relational databases allowed us to automate traceability and find change impacts with a query. It was the bill of materials system from hell.

When the new development workstation network went in - we were using a mainframe-class machine at the time - and all that publishing was supposed to happen on the workstations, I predicted it wouldn't work. The first run of the smallest of the hundreds of deliverable specs took over a week before it crashed. It never worked on that platform, although that platform, and those tools on that platform were successfully used for other development projects before and since.

In this case why was it that, as my old boss used to say, a process that works successfully for one hundred changes may collapse under ten thousand changes? When we ask questions about what went wrong—excluding questions about the particulars of the system we're building—we're asking questions about a model called the development system. We're asking questions and we're making observations about the personnel, tools and procedures used to develop a software system. The BSY-2 system presented many unique challenges to the development system used to build it.

The Development System

The development system is the collection of people, processes, and tools that implements the development sequence. If system development is a series of transformations from goals to requirements to design to code, the development system makes the transformations happen. The development system is perhaps best thought of as an information system that manipulates different descriptions of the system being built.

The Development system includes processes that directly implement the sequence from requirements to cod. It also

includes supporting processes like change management, issues resolution and even time reporting. In information systems terms, these processes correspond roughly to data and control flows, respectively.

Like other information systems, the development system has functions, but it also has attributes that describe how well it accomplishes those functions. (Tom Gilb introduces the idea of attributes in his *Principles of Software Engineering Management*.) Minimally, the important attributes of the development system are:

- *Volume*. It's harder to handle 10,000 things than 100. Will something that works for 100 requirements work for 10,000? The same question applies to defects, builds, target platforms, and test cases.
- *Process capacity*. Is it possible to process the required volume in the time available?
- *Concurrency*. How many people or events are in process at once?
- *Cycle time*. Will the individual processes execute quickly enough for the project to meet its schedule.
- *Operations.* These are attribute requirements—like availability and reliability—which can make transformations impossible. Can we run it enough to get the job done? (On BSY-2 the initial production timeline for some deliverable documents exceeded the mean time between failures for the host platform. Tuning and some redesign reduced this timeline by a factor of almost 1,000. Allowing greater concurrency, got us a total improvement on the order of 10,000. This scaling was never possible on the workstations, and the MTBF was worse.)
- *Deployment.* How broadly is it available? Can we get each function of the development system to everyone who needs to use it?
- *Process measures.* Any process in the development system has performance measures, like precision and repeatability. What are the process performance measures the development system has to meet? Overdesigning for a performance measure can be as bad as underdesigning for it.

For any particular project, the limiting process probably isn't what you think. Often, it's a control process. And for any particular process, the most important attribute probably isn't what you think either. Often, it's cycle time or concurrency, versus more obvious attributes.

By not considering the development system as a system, projects can either make the error of assuming that development system performance is a given or can fail to define the development system performance needed. Many projects make both errors simultaneously. When a project is in trouble, that's a good place to look. What is the limiting process? What are the important attributes of the processes in the development system? Is the development system being consciously created, or just accepted?

The Development System is a System

Some years before BSY-2, I built embedded controls for heat pumps. (The career transition makes sense; really it does.) We made a number of changes to the development system model in this period, just as relevant for this solo, and small-team development:

- We implemented reliable builds through scripting in the host's shell language. Build precision and repeatability went way up and we spent much less time searching for build bugs.
- Automated builds became nightly builds and eventually used a repeatable procedure for pushing the results out to the target system. We stopped forgetting what had changed. (An experienced developer who joined our small team once suggested that automated builds, and nightly builds were overkill. The whole team shouted the poor guy down, mainly because of how it simplified tracking changes.)
- We introduced design reviews prior to coding, and it greatly reduced design defects. However, we used only optional, informal reviews of bug fixes because errors and missed alternatives were rare enough without them.

This last change—design reviews—illustrates something that purists will probably object to.

For these systems (high-volume embedded controls) designed in this way (a very decoupled design) with this team of bright, responsible individuals, there was no benefit to *requiring* team design reviews of fixes. In part, this was because of the knowledge base built up by the earlier design reviews. Most defects were implementation errors. When there was a larger problem, or someone felt uncomfortable with a fix, we dealt with it in our regular meetings, or with informal reviews, both at the fixer's request.

These development system changes had an unexpected benefit: Rapid, reliable, repeatable changes to the control coefficients made tweaking the algorithms easier. After these development process changes were in place, I remember the engineer responsible for control algorithm design asking, "How long to get a chip with these algorithm changes and the rest of the software as is?" The look on his face was priceless when I said, "How's tomorrow?" He had been accustomed to delays of weeks, bad changes, and bad chips. Because of our development system changes, those heat pumps are more efficient than they would have otherwise been.

These examples illustrate that the development system is a system—everything is connected to everything else and sometimes in unexpected ways. Even the refinement of control algorithms, and eventually the efficiency of a product is influenced by build performance. Who would have thought it? It also illustrates that there isn't any one true development system model for every situation. We did just fine with only informal reviews of fixes.

Using the Model

My favorite piece by James Bach is "Process Evolution in a Mad World" (http://www.stlabs.com/testnet/docs/process.htm). The discussion his argument produced on newsgroups (and elsewhere) matters even more, I think. One of the observations from this discussion is that different problem domains require different balances between process control (which the advocates call "discipline" and "maturity") and open-ended problem-solving.

Large-scale tactical systems are embedded in and intertwined with hardware, procurement, facilities, training, and doctrine. Changes to software function have tremendous secondary costs with these system. These costs of change begin to accrue well before the early states of development. Embedded systems for commercial products have a similar level of external cost coupling, but the secondary costs are lower and start a bit later. Commercial heat pumps use existing production, distribution, and support to a greater degree than major new weapons systems. Put another way, for the control systems more of the cost of change was inside the box we shipped, so we could use a less extensive and less robust process to manage the secondary costs of change. We could also tolerate more change later in the game simply because change was cheaper.

On BSY-2, the support tools we built directly implemented parts of the development system. These tools had minor secondary costs of change compared to either the tactical system or embedded controls. Many of the procedures set up to manage the secondary costs of tactical system change were inappropriate for tool development. In addition, these processes assumed that analysis could predict problems—hardly true in terra incognita. When you're taking the tools somewhere they haven't been before, it's usually cheaper to try them and see what breaks.

As Bach suggests in his article, development should be exploratory and evolutionary. Our goal should be to design development to make opportunities for discovering value that the system might have, versus just increasing the performance in delivering the value the system is supposed to have. I'm sometimes a bit more of a formalist than Mr. Bach. I think development should be exploratory some of the time, not all of the time. Sometimes the costs of exploration are just too high. That said, I think as a profession we err toward too much definition, and miss opportunities to intentionally design down the costs of exploration. Often we design, or accept a development system that makes exploration and evolution prohibitively expensive, when there is no direct external reason to do so.

Since BSY-2, I've been exposed to enterprise-scale IT systems, which require a different relationship to the organization they support than products (like tactical systems, design specs, embedded controls, or shrink-wrap packages) have to the company that sells them. The differences in that relationship are most apparent in the requirements on the development system. For example, with very large IT systems, deployment is often the critical process (in the development system), and deployment robustness, timing, and coverage are often the critical attributes. Using a methodology that omits this process can lead to an incomplete development system, and project failure. We can build the functions just fine, but can't get them out to the people who must use them.

Different Situations / Different Development Systems

Different situations will require very different performance from the development system in terms of measurable attributes like volume, scale, and cycle time. Different situations will also require different emphasis on adaptability versus repeatability, control versus communication, and so on. To be successful, the development system must sometimes be profoundly different between projects. Certainly between types of systems: Building tactical systems is not like embedded controls, which are not like large IT systems, and so on.

Now that I've learned to recognize the development system model, I've noticed four more things:

- Any particular development system is good at some things and bad at others, which may or may not fit what a particular project needs.
- The most important development system performance measure is situational. It depends on what is being built.
- The mechanical requirements of the development system are pretty easily addressed. How many defects can we have and how quickly do we have to do builds? The policy and procedure issues are more subtle and prone to dogmatism.
- Maturity approaches trade some measures (throughput) for others (repeatability).

Successful project managers don't take the development system as a given, but rebuild it situationally to provide what their project needs. As a profession, we could all benefit by explicitly considering the development system we each have and how it applies to the project at hand. We could use tools and techniques from systems development itself to define and characterize the development system we need. Successful project managers do this kind of planning, in addition to the official PERT / Gantt project planning. It's part of why they succeed.

I think most software engineering techniques are really about changing the performance of the development system. But useful performance is situational. A truly useful change to the development system means it is useful in a particular case for a particular target system. In other words, we should measure usefulness by what it takes to solve the problem at hand, not by some general ideal development system, applicable to all efforts. There is no such animal. The tools group and the embedded control team I've described both abandoned known 'good' practices—and they were right to do so in those situations. They traded off one aspect of development system performance for a more important, meaning more relevant one. That was part of their success.

Improving the development system isn't as simple as improving development quality by reducing defects through reviews. Reducing defects may come at the price of fewer experiments or fewer new values from a system. What's it worth, and what does it cost? ❖

James Bullock *is Principal Consultant at Rare Bird Enterprises, where he develops methods and models for building successful systems. Contact him at jbullock@rare-bird-ent.com.*

The Article Development System

This is an edited version of the article published in the October 1999 issue of IEEE Computer, in James Bach's "Software Realities" column.

While developing this article, I spent 10 days dog-sitting in the boonies for some friends, hiding from work, and intending to use their e-mail account for the last couple of revisions. The day they left, their account was shut down - part of the password hijacking scam on AOL that has since made the news. This left some bad options: Track them down in England, take an excursion for a couple of days, sharing a car with an 85 pound Rotweiller-mix that thinks it's a puppy, or wait until they returned.

Obviously, the *Article Development System* changed, presenting a far different cycle-time for revisions than I, or the good people at IEEE Computer, had anticipated. The article got done and was, in the end, good enough. The experience provides an immediate, concrete example of the article's topic. And I learned another couple important development system attributes -- immunity to interruption, and recoverability (both how easy / hard to recover, and how long it takes).

While the preceding was revised from what appeared in IEEE Computer, it hasn't had the benefit of one more look from the outstanding editors there.