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.

The huge amount of many-to-many relationships in the system meant that a single change would impact a large number of elements and many deliverable documents. Manually grubbing through 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.

If system development is a series of transformations from requirements to code, the development system makes the transformations happen.

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.

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 tools, personnel, and procedures used to develop a software system.

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

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 an information system that manipulates different descriptions of the system being built. It includes processes that directly implement the sequence from requirements to code. It also includes supporting processes like change management, issue resolution, and even time reporting. In information systems terms, these processes correspond roughly to data and control flows.

Like other systems, the development system describes its own functions, but it also has attributes that describe how well it accomplishes those functions. Minimally, the important attributes—outlined by Tom Gilb in his *Principles of Software Engineering Management* (Addison-Wesley, 1988)—of the development system model 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 attributes*. These are attribute requirements—like availability and reliability—which can kill a model's usefulness. Can we

run it enough to get the job done?
- *Deployment*. How broadly is it available? Can we get it to everyone who needs to use it?
- *Process measures*. Any process in the development system has performance measures, like precision and repeatability, but overdesigning for performance (at the price of, say, cycle time) 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 the 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.

THE SYSTEM IS A SYSTEM

Some years before BSY-2, I built embedded controls for heat pumps. (The career transition makes sense; really, it does.) In the course of that work,

- 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. I stopped forgetting what I had changed. (An experienced developer who joined our small team later suggested that automated nightly builds were overkill. The whole team shouted the poor guy down, mainly because of how nightly builds 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.

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

For these systems (high-volume embedded controls) designed in this way (a very decoupled design) with this team 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. In part, this was due to the team; they were exceptionally responsible, asking for help when they needed it. It was also due to the delivery protocols for these systems. Shipping required a 100 percent retest with 100 percent success.

> For any particular project, the limiting process probably isn't what you think.

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 the control algorithm design asking, "How long to get a chip with these coefficient changes and the rest of the current software as is?" The look on his face was priceless when I asked him, "How's tomorrow?" He had been accustomed to bad changes, bad chips, and weeks of delays. 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. It also illustrates that there isn't any one true development system model for every situation.

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. These costs can be accrued even in the early states of development.

Embedded systems for commercial products have a similar level of integration with other elements, 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 the tools group built had minor secondary costs for change compared to either the tactical system or to embedded controls. Many of the procedures set up to manage the huge secondary costs of changes to the tactical system were inappropriate for tool development in which the costs were lower. In addition, these processes assumed that analysis could predict problems—hardly true in terra incognita. The tools group existed because the available tools and techniques wouldn't handle this tactical system. When you're taking the tools somewhere they haven't been before, it's usually cheaper to try them and see what breaks.

Bach suggests in his article that 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 or delivering the value the system is supposed to have. I'm sometimes a bit

Continued on page 124

Continued from page 120

more of a formalist than 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. But I think as a profession we err toward too much definition. As a result, we tend to miss design opportunities to reduce the costs of exploration.

> Successful project managers don't take the development system as a given, but rebuild it to provide what their project needs.

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.

DIFFERENT SITUATIONS

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 has to be profoundly different for each one.

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 for a particular target system is situational.
- 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 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. A truly useful change to the development system means it is useful in a particular case on a particular target system. In other words, we should measure usefulness by what it takes to solve the problem at hand, not by some arbitrary, general universal problem. 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.

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.