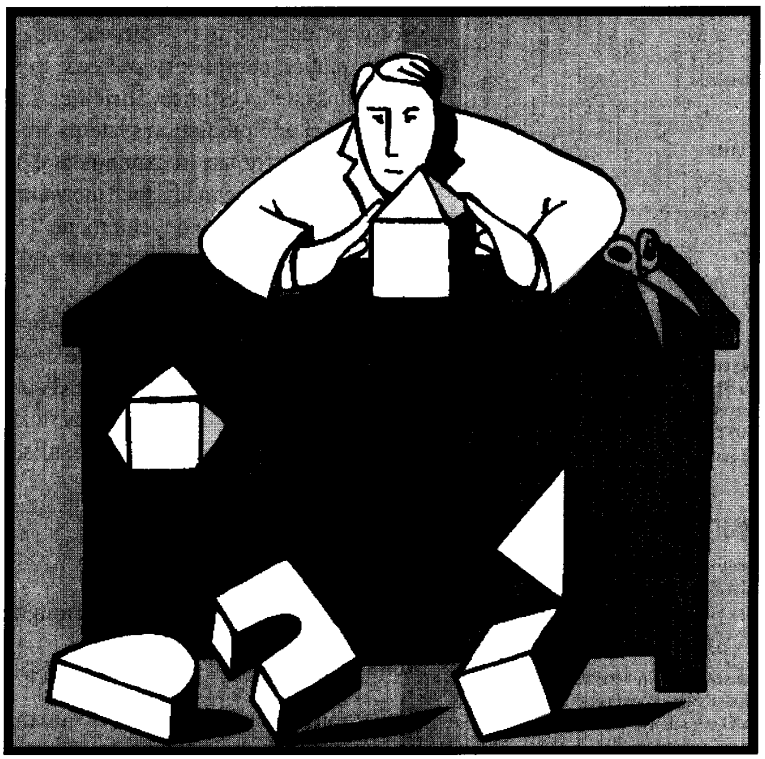
FEATURE

OPERATIONAL PROTOTYPING: A NEW DEVELOPMENT APPROACH

By selectively building throwaway prototypes on an evolutionary base, developers can avoid the dangers of retrofitting quality into a quick-and-dirty prototype.

ALAN M. DAVIS, University of Colorado at Colorado Springs



With the escalating costs of software development and increasing numbers of systems being built that fail to satisfy customer needs, more and more organizations are turning to prototyping. The results have been mixed. Some organizations build quick-and-dirty software — throwaway or rapid prototypes — in an attempt to discover the customer's true needs, throw the prototype away, and then build a production-quality system based on the now-understood requirements. Others build quality systems from the start — evolutionary prototypes — which they then evolve over time.

In a futile attempt to get the best of both worlds, many developers try to retrofit quality onto a rapid prototype and deliver it as a production-quality system — usually with the disaster of an unstable system and horrendous maintenance costs.

Some systems lend themselves well to the quick-and-dirty approach. Others are better off with an evolutionary prototype. Still other systems are good candidates for a combination of approaches. Unfortunately, this combination has usually meant trying to extensively evolve a rapid prototype. The approach I describe here — operational prototyping — avoids this danger by layering a rapid prototype atop a solid evolutionary base. It has been applied successfully in large-scale development, as the box on pp. 76-77 describes.

0740-7459/92/0900/0070/$03.00 © IEEE

TRADITIONAL PROTOTYPING

A prototype is the partial implementation of a system built expressly to learn more about a problem or a solution to a problem. Aircraft designers might build a prototype to tell them more about flight characteristics, for example; automobile designers might build one to show them ergonomics, drive-train behavior, or suspension characteristics.

The creation of a prototype has been a standard practice in many engineering and manufacturing industries for decades. The benefit of building a prototype instead of the actual object is that there is no manufacturing risk. If a company builds a real aircraft or automobile, there is the risk of tooling up a factory only to discover that the product doesn't work acceptably.

There is no manufacturing risk in software, but there is considerable development risk.[1] A software prototype implements part of the presumed software requirements to learn more about actual requirements or about alternative designs that could satisfy the requirements. Without prototyping, developers could easily create software that satisfies an incorrect set of requirements.

There are two distinct types of software prototypes described in the literature: throwaway and evolutionary.[2-4]

Throwaway. A throwaway prototype

- is built as quickly as possible,
- implements only requirements that are poorly understood — those whose need is not definite or whose desirable external behavior is unclear (after all, why waste money experimenting with understood requirements, learn nothing, and then discard the prototype?),
- is used experimentally,
- is used to learn which of the alleged requirements are real and which are not, and
- is discarded after the desired information is learned.

After the prototype is complete, the developer writes the software requirements specification,[5,6] incorporating what was learned, and constructs a full-scale system based on that specification.

Throwaway prototypes work very well in isolation to verify relatively small parts of complex problems.

Evolutionary. In contrast to a throwaway prototype, an evolutionary prototype

- is built in a quality manner (including a software-requirements specification, design documentation, and thorough test),
- implements only confirmed requirements (after all, why implement poorly understood requirements when you know you'll probably understand them much better after you build and implement the first prototype?),
- is used experimentally,
- is used to determine what requirements exist that haven't been thought of.

When the prototype is complete, the developer modifies the software-requirements specification to incorporate what was learned. The system is redesigned, recoded, and retested. This process is repeated indefinitely.

Evolutionary prototypes work well when most of the critical functions are well understood.

> Throwaway and evolutionary prototyping have almost nothing in common except the word "prototyping."

Comparison. Throwaway and evolutionary prototyping have almost nothing in common except the word "prototyping." As the first columns 2 and 3 of Table 1 show, they are built differently, implement different functions, serve different purposes, and have different outcomes.[6] Many factors influence a decision to select one approach or the other, including the amount of resources, whether or not users must understand a functioning system early on, and how much overlap there is between critical and well-understood functions.[2]

What approach is used also depends on the requirements, the range of which is shown in Figure 1. The vertical labels in the figure concern how well we understand requirements:

- Well understood means we know exactly what the users need.
- Poorly understood means we know the users need something but we're not sure how a system would behave that satisfies that need.
- Unknown means we are blind to their existence.

The horizontal labels in the figure con-

TABLE 1
COMPARISON OF PROTOTYPING AND CONVENTIONAL DEVELOPMENT

Characteristics	Throwaway prototyping	Evolutionary prototyping	Conventional development
Development approach	Quick and dirty; sloppy	Rigorous; not sloppy	Rigorous; not sloppy
What is built	Poorly understood parts	Well-understood parts first	Entire system
Design drivers	Development time	Ability to modify easily	Depends on project
Goal	Verify poorly understood requirements and then throw away	Uncover unknown requirements and then evolve	Satisfy all requirements

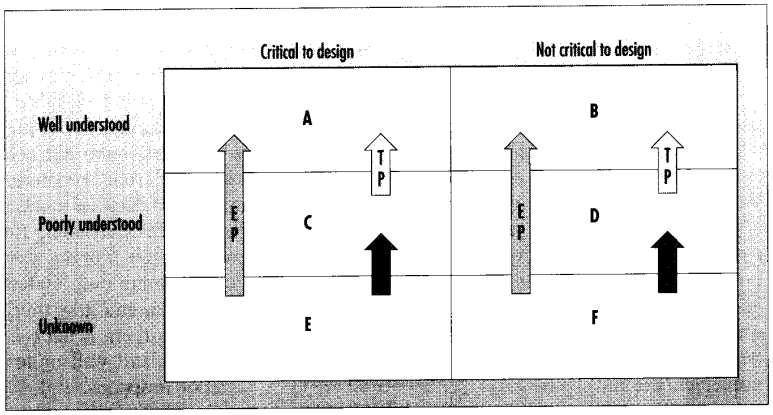


Figure 1. Range of problem requirements and the role of throwaway (TP) and evolutionary (EP) prototyping.

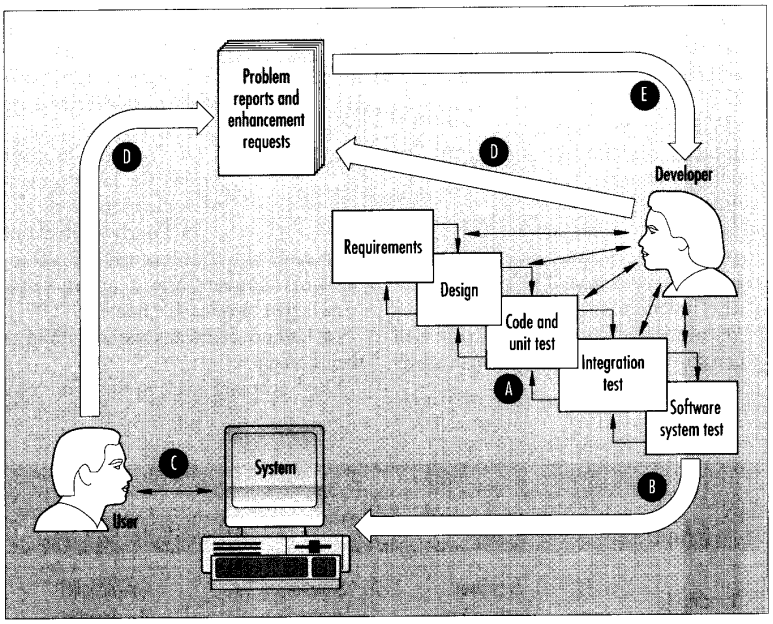


Figure 2. Conventional development: (A) Develop system, (B) deliver system, (C) use system, (D) report problems, (E) receive, prioritize, and schedule changes.

cern how critical requirements are to the design.

♦ Critical to the design means that these requirements are driving the eventual architecture. An example is requiring the system to update the tracks of 500 ships simultaneously every second.

♦ Not critical to the design means that these requirements are not driving the design. An example is requiring the system to flash the track of any ship whose position has been updated.

As the figure shows, a throwaway prototype implements a poorly understood requirement and migrates it to the well-understood class. An evolutionary prototype implements a well-understood requirement, expecting users and developers to uncover previously unknown requirements. Clearly, we do not want to lock in on a specific architecture until after requirements critical to the selection of that architecture are well understood. Thus, an evolutionary prototype should not be attempted unless region A in the figure is much bigger than region C (and region E, although there is no way to know the size of region E or F except by conjecture).

Throwaway prototypes are an effective way to move requirements from region C (and D) to region A (and B).

PROTOTYPING AND CONVENTIONAL DEVELOPMENT

Conventional software development is usually performed in a series of iterative phases in a laboratory environment, as shown in Figure 2. Systems are installed at customer sites. Users report problems and enhancement requests to developers. Developers meanwhile are also detecting problems and opportunities for enhancement. All these requested changes are funneled into software-configuration management.[8] During this process, the requests are approved or rejected, and the resulting changes are coordinated, prioritized, and scheduled.

As Figure 3 shows, throwaway prototyping follows little or no formal method. Customers and potential users experiment with the prototype in the laboratory. Once developers learn the necessary information, they update the software-requirements specification and begin conventional development.

As Figure 4 shows, evolutionary prototyping follows a conventional development approach to achieve adequate levels of built-in quality. To minimize risk, the developer does not implement poorly understood features. The partial system is sent to customer sites. As users work with the system, they detect opportunities for new features and give requests for these features to developers. Developers then take these enhancement requests along with their own and use sound configuration-management practices to change the software-re-

quirements specification, update the design, recode, and retest.[7]

Although Figures 2 and 4 appear to be similar, there are some differences, as the last two columns in Table 1 show. The primary difference is that conventional development attempts to build the entire set of requirements; whereas, evolutionary prototyping acknowledges that we do not understand all the requirements and builds only those that are well understood.

Conventional development and prototyping may be combined — but with caution. For example, in a complex system, developers may use throwaway prototyping for some parts and evolutionary prototyping for others. If such an approach is taken, however, the developer must avoid trying to convert the code of a throwaway prototype into a real product.

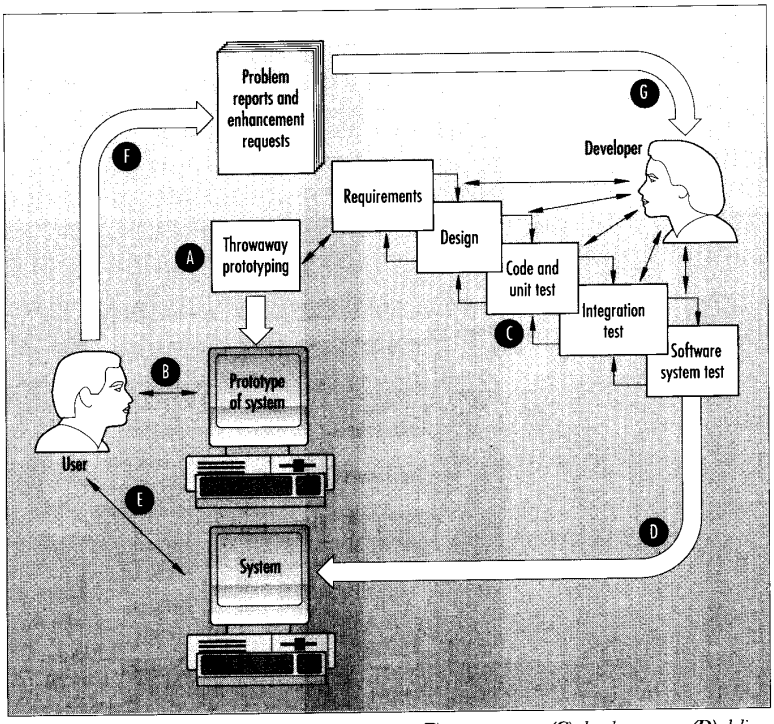


Figure 3. Throwaway prototyping: (A) Build prototype, (B) use prototype, (C) develop system, (D) deliver system, (E) use system, (F) report problems, (G) receive, prioritize, and schedule changes.

OPERATIONAL PROTOTYPING

For some systems, neither throwaway nor evolutionary prototyping alone is acceptable. Most of the requirements for these systems are either critical to the design and well understood, not critical to the design and poorly understood, or unknown. In addition, these poorly understood requirements make sense only in the context of a working system. Throwaway prototyping alone is ineffective because the poorly understood requirements are not critical and the user cannot judge their adequacy when they are implemented without looking at the way the rest of the system behaves.

Evolutionary prototyping alone is ineffective because it doesn't help clarify poorly understood requirements and it tends to surface unknown requirements. These unknown requirements never become well understood because the emerging requirements are too complex, and evolutionary prototyping is not good for implementing experimental features rapidly. For these systems the ideal solution is one that combines rapid results with stability.

Operational prototyping offers this balance because throwaway prototypes are selectively built atop evolutionary proto-

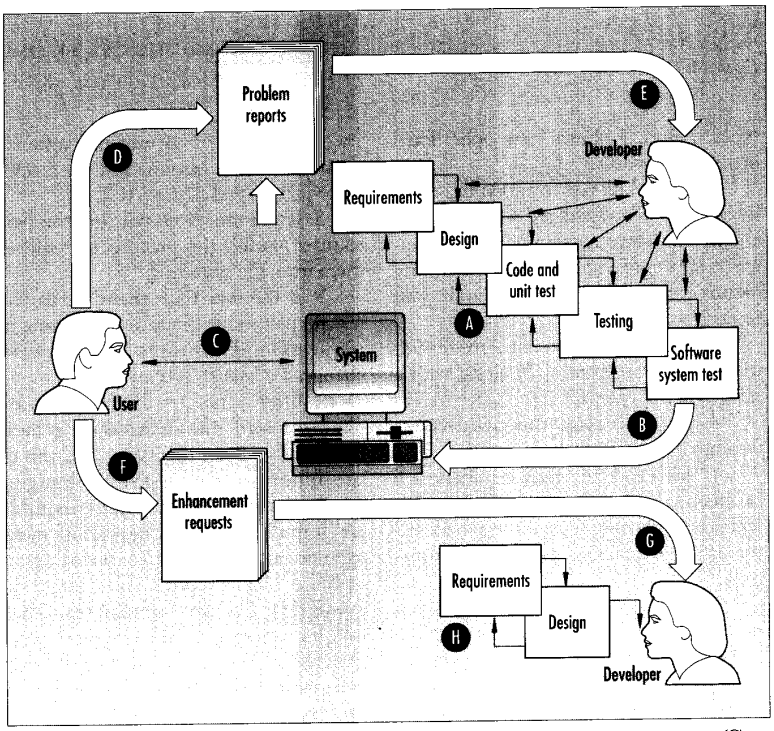


Figure 4. Evolutionary prototyping: (A) Develop understood parts of the system; (B) deliver system; (C) use system; (D) report problems; (E) receive, prioritize, and schedule changes; (F) report enhancements; (G) receive, prioritize, and schedule enhancements; and (H) evolve system.

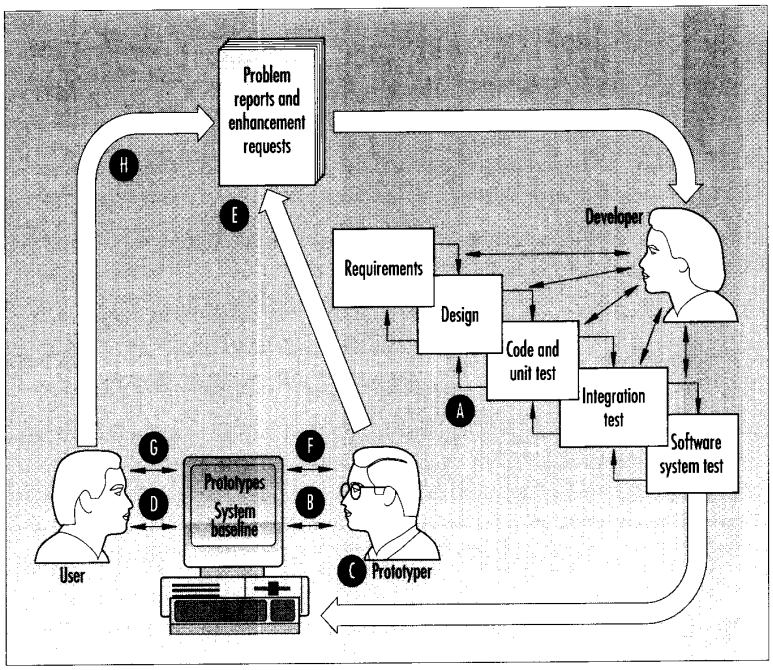


Figure 5. Operational prototyping.

types. Figure 5 shows how operational prototyping works:[8]

1. An evolutionary prototype (a quality product) is constructed and made into a baseline using conventional development strategies, specifying and implementing only requirements that are well understood (A).

2. Copies of this baseline are sent to multiple customer sites (B) along with a trained prototyper (C).

3. At each site, the prototyper watches the user at the system (D).

4. Whenever the user experiences a problem, the prototyper records it (E), which frees the user from having to record the problem either on paper or electronically.

5. Whenever the user recognizes a new need (dialogue like "I sure would like to be able to do *x*."), the prototyper records the request (E).

6. As soon as the user leaves, the prototyper constructs a quick-and-dirty throwaway prototype (F) on top of the quality baseline software.

7. The user then uses the software composed of the baseline and the throwaway-prototyped changes (G).

8. If the user does not find the new features useful, the prototyper removes them from the system.

9. If the user finds them useful, the prototyper writes feature-enhancement requests and captures the software changes for later reference (E).

10. Over time, a collection of such changes and enhancements arises. Eventually, the prototyper returns to the laboratory with these changes. These feature-enhancement requests are funneled into configuration management along with conventionally generated problem reports and change requests (H). The development team then follows conventional development practices (A) to incorporate the new features in a quality fashion into the evolutionary prototype, creating a new baseline (B). The code from the site-generated throwaway prototypes is never incorporated in the baseline.

11. Steps 2 through 10 are repeated indefinitely.

Operational prototyping has several implications for configuration management, quality assurance, and general project management.

Configuration management. Configuration management is the process of managing and controlling changes to the software.[7] In this context, software includes requirements specifications, design documents, code, test plans, test results, user documents, and any other development products. Configuration management typically consists of

- establishing a standard naming convention for all parts of the software;
- regularly defining baselines;
- defining and enforcing a set of change procedures that ensures all parties understand how to submit a request for change, all people with a stake in a decision are informed of requested changes, opinions of all stakeholders are considered in the decision of whether or not to incorporate a change, all affected parties are informed of decisions, and changes are incorporated as planned; and
- carefully maintaining records to make it easy to trace all baseline changes to the reason for change.

Because operational prototyping uses evolutionary prototyping as a base, configuration management is pretty much the same for the two approaches. However, in operational prototyping, designers must take care to use the right configuration-management approach for the different development parts.[7]

In configuration management for operational prototyping, shown in Figure 6, major software components are identified and controlled independently. As development progresses, baselines are created, and the product is fielded. As problems are reported, they are handled in the usual manner — change-request forms are filed (usually electronically), changes are reviewed by a configuration-control board, prioritized, approved, and scheduled for future releases. The change-control board, which represents both developers and users, makes decisions about the rela-

tive priorities of changes and resolves any conflicts between these changes. These maintenance-type changes are made to the product, grouped together into releases, made into new baselines, and delivered to field sites.

Meanwhile, prototypers are making changes to software versions in the field. These changes must be made quickly and cannot afford the overhead of configuration management. They need not be controlled to any great extent: only one person is involved, quality is not an overriding concern, and the changes are only temporary. The only configuration management necessary is the creation of a record of who made what change to the code when. This type of code control is provided automatically by such tools as the Source Code Control System of AT&T's Unix, Digital Equipment's Code Management System, or SofTool's Change and Configuration Control System. Once control is abandoned on a version of software, sound configuration management cannot be practiced on it. For this reason, prototypers must remove patches from fielded versions before returning the prototype to the laboratory.

Each change made by the prototyper in the field represents the fulfillment of a new user requirement. The changes themselves are not worth saving, but the requirements that they represent are. The prototyper records these requirements by creating change-request forms for each one. It is helpful to add a new field to the standard form that indicates that this change request represents a change already prototyped in the field and approved by site users. These change requests are then funneled into the normal configuration-management process, as shown in Figure 6. Configuration-management personnel must take care to resolve any conflicts between independently prototyped changes.

Quality assurance. Quality assurance is the set of practices and procedures necessary to ensure that the software delivered to customers satisfies their needs and is stable and reliable. Some practices include quality reviews and inspections,[9] and the creation and enforcement of standards.[5] Like configuration management, quality assurance differs for throwaway and evolutionary prototype aspects of operational prototyping.

Throwaway prototype segments are designed merely to provide the user with quick functionality and then are thrown away. Thus, for this part of operational prototyping, reviews, inspections, development standards, and other quality-assurance measures are pointless.

For the evolutionary base, however, you should use the same quality-assurance techniques appropriate for full-scale development. There's no shortcut to quality. You can slightly reduce or at least postpone some standards, but you won't gain much in decreased development costs.

Management challenges. In addition to configuration management and quality assurance, new project-management problems and challenges arise with the use of operational prototyping.

Retaining quality prototypers. Successful prototypers possess a set of unique skills:

- They are willing and able to work for long periods. The prototyper must work with the user while the user is experimenting with the system. The prototyper must also develop new prototype software while the user is not experimenting with the system — for example, while the user sleeps.
- They have excellent user rapport. The prototyper must work hand in hand with the user and must thus be flexible, understanding, respectful, patient, and so on.
- They have detailed knowledge of the application domain. Naturally, the user doesn't want to train the prototyper, so the prototyper must already know the application and its associated jargon.
- They have excellent quick-and-dirty development skills. The prototyper has to generate working code fast. Of course, it helps if the prototyper worked on the original system, and if the system is written in a language that makes it easy to create software rapidly.

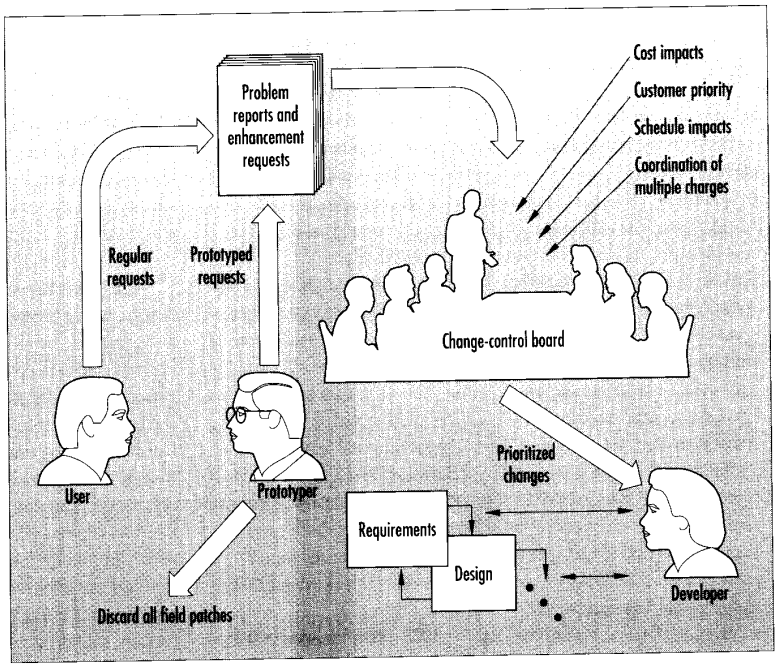


Figure 6. Configuration management for operational prototyping.

Fighting the urge to keep throwaways in the field. You *must* insist that prototypers remove prototype software from the field before they leave because by its very nature, this software is unstable and unpredictable. It isn't always easy to enforce this rule. The user may have grown accustomed to a new feature, and will be greatly displeased to lose a capability. Also, it is hard to respond to a user who says, "I'm paying you to be here, and I want you to leave me the new capabilities when you go."

Fighting the urge to use patches in the next version. You must also suppress the urge to incorpo-

PROTOTYPE OCEAN SURVEILLANCE TERMINAL: CASE STUDY OF OPERATIONAL PROTOTYPING

Prototype Ocean Surveillance Terminal — a desktop ocean-surveillance system developed by BTG, Inc., for the US Navy — and its derivatives have been deployed at more than 200 sites and played a major role in the 1990 Middle East War. Although POST evolved slowly and painfully over several decades, it is now a highly successful, real-time, mission-critical project that has benefited a great deal from the use of operational prototyping.

Application. To fulfill its mission, a Navy combat vessel needs to know the location and identification of other ships that might pose a threat. Such ships may contain sensors that can detect the vessel's location or weapons that can destroy the vessel.

Until the early 1980s, gathering the enormous quantity of raw identification and location data, digesting it, and providing it to Navy vessels was the responsibility of a half dozen number-crunching, land-based sites — called Ocean Surveillance Information Systems — located around the world.

In the early 1980s, a variety of reasonably priced high-performance desktop workstations appeared on the market. A group of individuals at BTG, Inc., envisioned packaging some of the OSIS capability into a workstation and deploying it aboard Navy vessels. This would enable ships not only to function more independently, but also to generate specific information more rapidly. Once workstations were installed, both BTG and the US Navy believed that a host of other functions would become evident. But neither could define those functions ahead of time.

Consequently, BTG agreed to build a prototype and experiment with the concept. Preliminary investigations of prototyping technology revealed some useful concepts but none met all the Navy's needs:

- They had to deploy a working system quickly.
- They could articulate only a miniscule subset of overall system functions.
- They wanted to play with a working system, provide feedback, and quickly see a new desired capability.
- Product accuracy, stability, and robustness were critical.

BTG developers knew they needed a new prototyping method — particularly one that gave them rapid development of experimental features while providing the highest standards of quality control, configuration management, stability, and robustness — but they didn't know exactly how to get there.

1983-1984: Throwaway Prototyping Era. In December 1983, BTG was awarded a six-month, $30,000 contract to experiment with electronic intelligence correlation on a Hewlett-Packard 9845. This simple tracker system, called Belt, was built in a quick-and-dirty fashion using Rocky Mountain Basic, but it worked. Basic was chosen because it was readily available in 1983 on the host machine, and the belief was that you could use it to build systems rapidly although not necessarily in a way that made them easy to maintain. BTG used throwaway prototyping to ensure that they had enough well-understood requirements to create a baseline.

Belt was installed for user experimentation aboard the *USS Carl Vinson*, an aircraft carrier, and at a Washington, DC, shore-based site. There was no quality evolutionary baseline in the laboratory. Patches made in the field were left there and the two sites evolved independently. Bruce Gregor, Chip Bumgardner, and Kris Heim served as on-site prototypers, making regular trips between the two sites to implement new features. During one memorable trip, Gregor had completed his prototyping activities aboard the *Carl Vinson* and was ready to return home. Unfortunately, the ship was about to make a long journey in the opposite direction, so it dropped him off at Diego Garcia, a remote island in the Indian Ocean. He received permission to board the next ship out, hoping to get one step closer to home. The ship he boarded took him to Tokyo, where he was promptly taken into custody by Japanese customs officials because he had no visa!

In summer 1984, the Navy and BTG decided to port the software to the HP9020, rename it the Prototype Ocean Surveillance Terminal, and add new functions, but not to significantly rewrite it. The three original prototypers were joined by Joe Smith and Drew Cohen. In the next six months, this team of five struggled to maintain the product's stability while servicing the half dozen customer sites. The product was understandably unstable.

1984-1988: Operational Prototyping Era. In late 1984, BTG decided to keep rotating prototypers at the customer sites, but also to rewrite the product in a quality fashion on the HP9020 to reduce product instability. Although no one could identify it as such, this marked the beginning of operational prototyping. After POST was redesigned, it had all the features the team had gathered from all the sites and was fully documented.

By late spring 1985, the new

rate the field-generated patches into the next version of the system for the same reasons you leave those patches at the customer site. This, too, is hard. Users will say, "Look, I've already seen that capability on my system. What do you mean you aren't done designing and coding it?"

Selecting a language. In general, languages that make rapid development easier result in applications that are relatively difficult to maintain and vice versa. Easily maintained languages make rapid development harder. Because operational prototyping has both needs, you must make a choice:

- When most requirements are poorly understood, consider throwaway prototyping. POST was a throwaway prototype for more than a year before the team understood enough requirements to begin developing the baseline.
- When some requirements are well understood and stable but many more are suspected, consider operational prototyping.
- Every two to three years, consider rewriting the baseline from scratch. Hundreds of new features were added to POST annually in response to feedback gathered at user sites. Under this constant flux, the system required a major redesign every two to three years — which is similar to what Frederick Brooks recommended in *Mythical Man Month* (Addison-Wesley, 1975): Plan to discard at least one baseline; you will anyway.
- Hire flexible top performers to serve as prototypers. Some hired for POST already knew the application domain. Those who didn't spent at least a year with the system in the laboratory to learn the application before they were considered qualified to be on-site prototypers.

product was at dozens of sites worldwide. Cohen rode three to four different ships as a prototyper in the Pacific, each trip lasting one to two months. Smith and Heim circulated as prototypers among sites in Scotland, Virginia, Washington, DC, and seven to eight ships in the Atlantic fleet. The team and even other members of BTG, faced other dangerous situations. During the Libyan incursion in 1986, for example, the POST aboard the *USS America* in the Persian Gulf developed a faulty board. All the prototypers were busy, so BTG's president flew to Italy, was airlifted onto the *America*, replaced the board, and was airlifted back — all during battle.

In another visit, Heim was lowered by helicopter onto the deck of a destroyer and departed the same way — something he hadn't learned as a computer science major!

Gregor, Bumgardner, and the others maintained and upgraded the laboratory baseline in addition to their other prototyping tasks. As each prototyper returned to a customer site, he delivered the newest release (with new features incorporated into the baseline in a quality manner) to the customer and immediately started building throwaway prototypes atop the baseline.

By spring 1985, there were far more sites than prototypers, and BTG could not assign a prototyper to every system. In May 1985, Ken Sepeda joined the POST team and started moving from ship to ship as a full-time trainer and collector of user feedback. At this point, there were four or five regular prototypers and two or three laboratory-based developers and maintainers.

In mid-1986, one subset of customers wanted to continue operational prototyping using the stable baseline system and on-site prototypers. Another subset wanted a more traditional, rigorous, laboratory-based full-scale development. The former system was renamed Paws; the latter retained the name POST. The combined POST and Paws projects by this time were keeping about a dozen people busy. Throughout 1987 and 1988, Paws development continued with prototypers going aboard ships and developers working in the laboratory. POST continued to become more stable and controlled, and developers were using traditional configuration-management and quality-assurance practices. User requests for changes were funneled directly from users to the change-control board, not through prototypers. Occasional ship-based prototyping occurred, but it was more the exception than the rule.

By 1988, Paws had been continually and extensively modified for four years and could no longer serve as a stable, reliable base for all the change requests from customer sites. Thus, BTG completely rewrote it to form a new evolutionary baseline, which they renamed the Advanced Tracking Prototype. ATP contained approximately 120,000 lines of Rocky Mountain Basic, and went to 40 sites worldwide. In winter 1988, it was stabilized and given more stringent configuration-management controls. One or two prototypers traveled regularly for two to five weeks at a time to customer sites. At the same time, BTG spun off a US Air Force version, called Constant Source, which evolved independently.

1988-Present: Lessons Era. Since 1988, BTG and other companies have spun off numerous baselines, which have been renamed for particular customers. BTG also developed an entirely new version from scratch on Unix using C at the request of several customers.[1] There are now more than 70 ATP systems, between 70 and 100 POST systems, approximately 20 Constant Source, and four to five Paws systems deployed worldwide.

The POST experience was invaluable as a breaking ground for operational prototyping. Many lessons were learned:

REFERENCES

1. A. Incorvaia, A. Davis, and R. Fairley, "Case Studies on Reuse," *Proc. Compsac*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 301-306.

♦ Build throwaway prototypes using a different language than the baseline product. The drawback is working in an environment with multiple languages, especially when building throwaway prototypes that alter the existing code.

♦ Select a language that is at neither extreme. A middle-of-the-road language would be moderately maintainable and moderately time efficient. The drawback is that neither goal is optimized.

Of course, selecting a language may be no more or less difficult than it is for other process models, since language selection is often driven by personal preference rather than language properties.

Selecting an architecture. The architecture you select must be resilient to extensive change. The techniques recommended for many years for building evolvable software apply here as well: information hiding, low coupling, high cohesion, table-driven software, software architectures that mimic the architecture of the real world, and so on. Under the constant onslaught of change, however, you can expect to create completely new baselines regularly.

Working with widely dispersed personnel. Managing a large software project when all developers are collocated is difficult. Distributing a dozen or so developers to customer sites to do prototyping may seem to compound the difficulties, but prototypers can be successful only if they are free to satisfy the customers' needs on site experimentally, without management intervention.

Maintaining customer satisfaction. Short-term customer satisfaction using operational prototyping is relatively easy to maintain by providing highly skilled and responsive prototypers to the sites. Long-term customer satisfaction is more difficult. You must instill trust in the customer that a removed prototyped function will reappear in a quality manner in an upcoming release.

Operational prototyping offers the best of both the quick-and-dirty and conventional-development worlds in a sensible manner. Designers develop only well-understood features in building the evolutionary baseline, while using throwaway prototyping to experiment with the poorly understood features. Throwaway prototypes never remain in the evolving product.

I see operational prototyping as merely the next approach in the continuing search for more effective methods, however. The maturation of engineering principles appears to follow a series of plateaus. In the early days of bridge-building, many bridges collapsed because of faulty engineering practices. These failures drove us toward finding improved techniques. A better technique was found, and it worked for a while. Finally, we attempted to apply it to a new type of problem, it failed, and the cycle repeated.

This maturation process applies to software engineering as well. Throwaway prototypes work very well in isolation for small, relatively static problems. Attempts to apply them in more dynamic situations have failed. Operational prototyping is the next plateau. It worked for one system. Hopefully, others will apply it to their problems, and it will work for them as well. Eventually, it will be applied to some problem with unforeseen attributes, it will fail, and the search for a better prototyping technique will continue. ♦

ACKNOWLEDGMENTS

Kris Heim provided invaluable assistance by discussing the history of POST. The sidebar in this article would have been impossible without him. Ed Bersoff deserves credit for recognizing the importance of, and successfully selling customers on, the concept of a desktop ocean-surveillance system. Bruce Gregor, Chip Bumgardner, Kris Heim, Drew Cohen, Joe Smith, Monte Rodgers, and Ken Sepeda were the folks whose long hours of work and great dedication made POST a success.

REFERENCES

1. R. Charette, *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, 1989.
2. L. Alexander and A. Davis, "Criteria for the Selection of a Software Process Model," *Proc. Compsac*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 521-528.
3. A. Davis et al., "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Trans. Software Eng.*, Oct. 1988, pp. 1453-1461.
4. H. Gomaa, "Software Prototypes: Keep Them or Throw Them Away?" in *InfoTech State of the Art Report on Prototyping*, M. Lipp, ed., Pergamon Press, Oxford, 1986.
5. "Defense System Software Development, DoD-Std-2167A," US Dept. of Defense, Washington, DC, 1988.
6. A. Davis, *Software Requirements: Analysis and Specification*, Prentice Hall, Englewood Cliffs, N.J., 1990.
7. E. Bersoff and A. Davis, "Impacts of Life Cycle Models on Software Configuration Management," *Comm. ACM*, Aug. 1991, pp. 104-118.
8. E. Bersoff et al., *Software Configuration Management*, Prentice Hall, Englewood Cliffs, N.J., 1980.
9. M. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.*, July 1986, pp. 744-751.

Alan M. Davis is a professor of computer science and El Pomar chair of software engineering at the University of Colorado at Colorado Springs and at the Colorado Institute for Technology Transfer and Implementation. He is the author of *Software Requirements: Analysis and Specification* (Prentice Hall, 1990) and the author or coauthor of more than 40 papers on software and requirements engineering. He is also the coeditor of *IEEE Software*'s Manager column with Winston Royce and associate editor of *Journal of Systems and Software*.

Davis holds a BS in mathematics from the State University of New York at Albany and an MS and a PhD in computer science from the University of Illinois. He is a senior member of the IEEE. His address is University of Colorado, CS Dept., 1867 Austin Bluffs Pkwy., Suite 200, PO Box 7150, Colorado Springs, CO 80933-7150; Internet adavis@zeppo.uccs.edu.