The Forgotten Phase

Development models often consider testing an afterthought, but there *are* models that focus on testing. This month, we'll examine the V Model—but is it flawed, too? Part 1 of 4. BY ROBIN F. GOLDSMITH AND DOROTHY GRAHAM

THE SOFTWARE WORLD HAS LONG FOUND IT helpful to define the phases of the development lifecycle. In addition to the classic waterfall approach, there are spiral or iterative processes, rapid application development (RAD) and—more recently—the Unified Process (known as RUP because of its origins at Rational Software). But testing often gets short shrift.

Just as development has its models, so, too, does testing. However, these processes tend to be less well known, partly because many testers have gained most of their expertise on the job. After all, even though testing constitutes about half of development time, most formal academic programs that purport to prepare students for software careers don't even offer courses on testing.

Experts keep proposing new models because developers find value in them, but also feel limited by existing models. For example, no approach has been presented more fully than the RUP, yet even it has significant gaps—so many, in fact, that *Software Development* contributing editors Scott Ambler and Larry Constantine filled four volumes with the collected articles from the magazine that describe best practices and gap-filling in the RUP. Incidentally, one of those gaps is the RUP's failure to address test planning (see "Plan Your Testing," by Robin Goldsmith, in *The Unified Process Inception Phase* [CMP Books, 2000]).

The V Model

The V Model, while admittedly obscure, gives equal weight to testing rather than treating it as an afterthought.

Initially defined by the late Paul Rook in the late 1980s, the V was included in the U.K.'s National Computing Centre publications in the 1990s with the aim of improving the efficiency and effectiveness of software development. It's accepted in Europe and the U.K. as a superior alternative to the waterfall model; yet in the U.S., the V Model is often mistaken for the waterfall.

In fact, the V Model emerged in reaction to some waterfall models that showed testing as a single phase following the traditional development phases of requirements analysis, high-level design, detailed design and coding. The waterfall model did considerable damage by supporting the common impression that testing is merely a brief detour after most of the mileage has been gained by mainline development activities. Many managers still believe this, even though testing usually takes up half of the project time.

Testing Is a Significant Activity

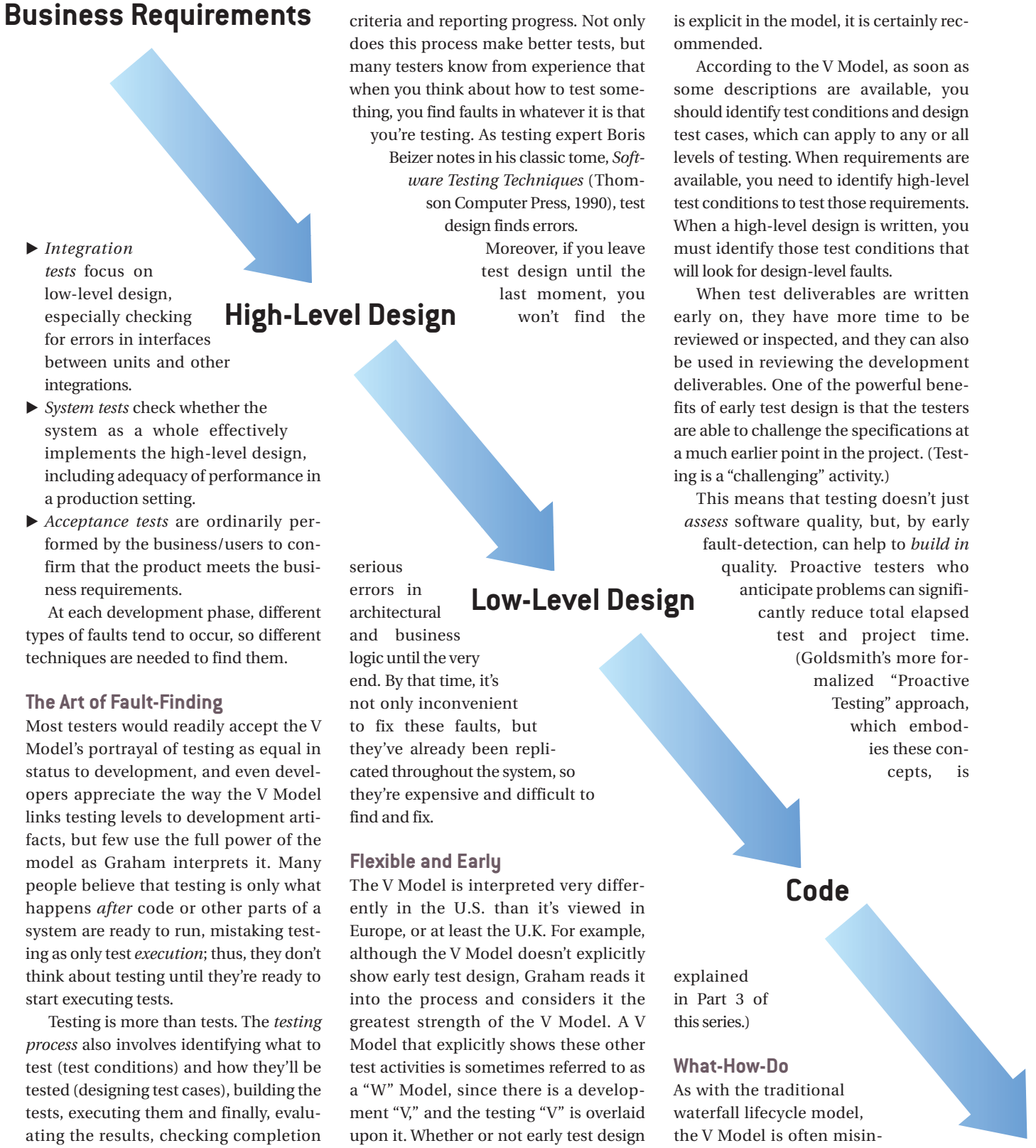
The V Model portrays several distinct testing levels and illustrates how each level addresses a different stage of the lifecycle. The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side (see next page). (Note that some organizations may have different names for the various development and test activities.)

On the development side, we start by defining business requirements, then successively translate them into high- and low-level designs, and finally implement them in program code. On the test execution side, we start by executing unit tests, followed by integration, system and acceptance tests.

The V Model is valuable because it highlights the existence of several *levels of testing* and delineates how each relates to a different development phase:

- *Unit tests* focus on the types of faults that occur when writing code, such as boundary value errors in validating user input.

Robin F. Goldsmith *is the president of Go Pro Management Inc. consultancy in Needham, Mass., which he cofounded in 1982. A frequent conference speaker, he trains business and systems professionals in testing, requirements definition, software acquisition and project and process management. Reach him via www.gopromanagement.com.* ***Dorothy Graham*** *is the founder of Grove Consultants (www.grove.co.uk) in the U.K., which provides consultancy, training and inspiration in software testing, test automation and inspection. With Tom Gilb, she is coauthor of* Software Inspection *(Addison-Wesley, 1993) and with Mark Fewster, coauthor of* Software Test Automation *(Addison-Wesley, 1999). In 1999, she was awarded the IBM European Excellence Award in Software Testing.*



- *Integration tests* focus on low-level design, especially checking for errors in interfaces between units and other integrations.
- *System tests* check whether the system as a whole effectively implements the high-level design, including adequacy of performance in a production setting.
- *Acceptance tests* are ordinarily performed by the business/users to confirm that the product meets the business requirements.

At each development phase, different types of faults tend to occur, so different techniques are needed to find them.

The Art of Fault-Finding

Most testers would readily accept the V Model's portrayal of testing as equal in status to development, and even developers appreciate the way the V Model links testing levels to development artifacts, but few use the full power of the model as Graham interprets it. Many people believe that testing is only what happens *after* code or other parts of a system are ready to run, mistaking testing as only test *execution*; thus, they don't think about testing until they're ready to start executing tests.

Testing is more than tests. The *testing process* also involves identifying what to test (test conditions) and how they'll be tested (designing test cases), building the tests, executing them and finally, evaluating the results, checking completion criteria and reporting progress. Not only does this process make better tests, but many testers know from experience that when you think about how to test something, you find faults in whatever it is that you're testing. As testing expert Boris Beizer notes in his classic tome, *Software Testing Techniques* (Thomson Computer Press, 1990), test design finds errors.

Moreover, if you leave test design until the last moment, you won't find the serious errors in architectural and business logic until the very end. By that time, it's not only inconvenient to fix these faults, but they've already been replicated throughout the system, so they're expensive and difficult to find and fix.

Flexible and Early

The V Model is interpreted very differently in the U.S. than it's viewed in Europe, or at least the U.K. For example, although the V Model doesn't explicitly show early test design, Graham reads it into the process and considers it the greatest strength of the V Model. A V Model that explicitly shows these other test activities is sometimes referred to as a "W" Model, since there is a development "V," and the testing "V" is overlaid upon it. Whether or not early test design is explicit in the model, it is certainly recommended.

According to the V Model, as soon as some descriptions are available, you should identify test conditions and design test cases, which can apply to any or all levels of testing. When requirements are available, you need to identify high-level test conditions to test those requirements. When a high-level design is written, you must identify those test conditions that will look for design-level faults.

When test deliverables are written early on, they have more time to be reviewed or inspected, and they can also be used in reviewing the development deliverables. One of the powerful benefits of early test design is that the testers are able to challenge the specifications at a much earlier point in the project. (Testing is a "challenging" activity.)

This means that testing doesn't just *assess* software quality, but, by early fault-detection, can help to *build in* quality. Proactive testers who anticipate problems can significantly reduce total elapsed test and project time. (Goldsmith's more formalized "Proactive Testing" approach, which embodies these concepts, is explained in Part 3 of this series.)

What-How-Do

As with the traditional waterfall lifecycle model, the V Model is often misinterpreted as dictating that development and testing must proceed slavishly in linear fashion, performing all of one step before going on to perform all of the next step, without

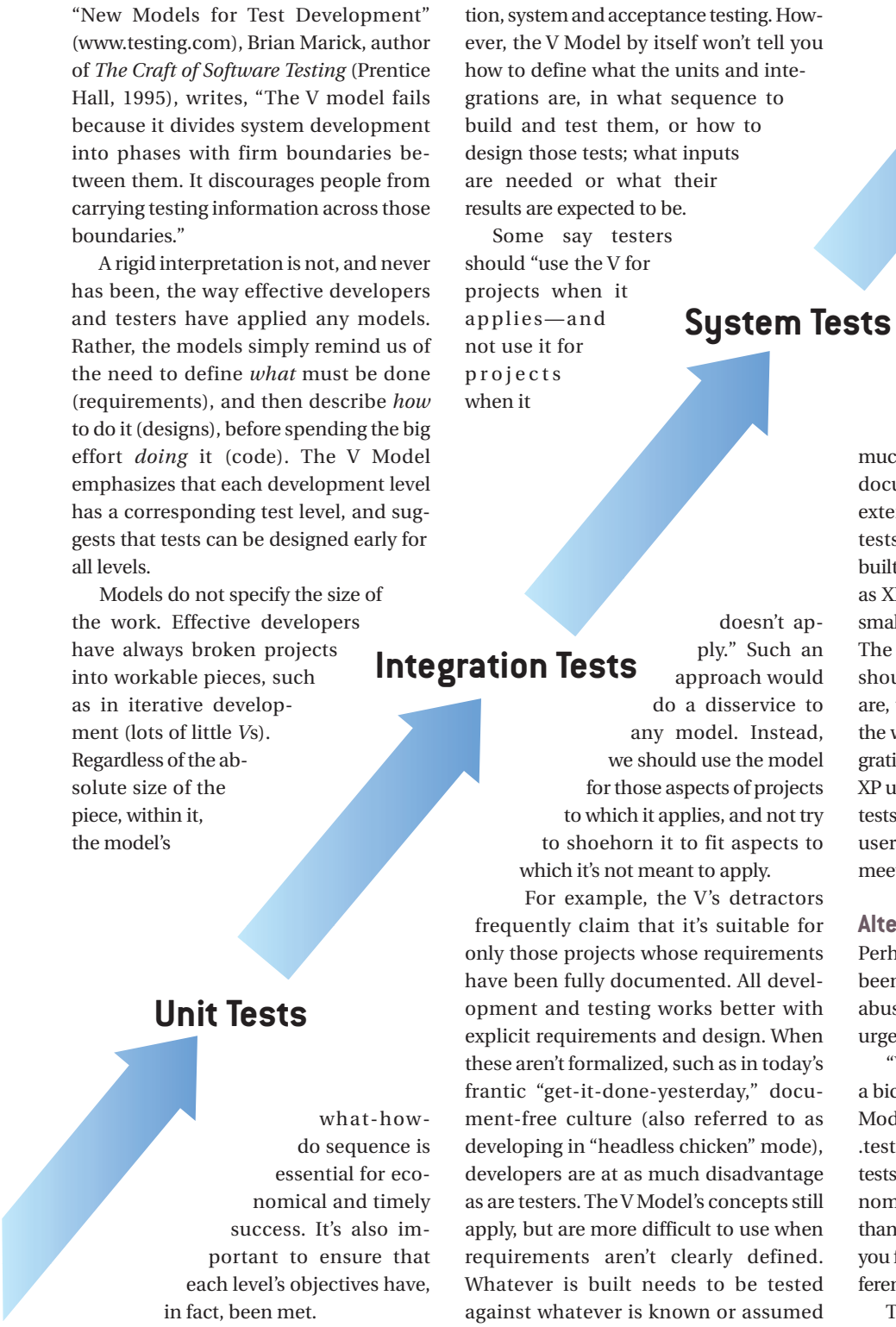
The V Model

The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side.

allowing for iteration, spontaneity or adjustment. For example, in his paper, "New Models for Test Development" (www.testing.com), Brian Marick, author of *The Craft of Software Testing* (Prentice Hall, 1995), writes, "The V model fails because it divides system development into phases with firm boundaries between them. It discourages people from carrying testing information across those boundaries."

A rigid interpretation is not, and never has been, the way effective developers and testers have applied any models. Rather, the models simply remind us of the need to define *what* must be done (requirements), and then describe *how* to do it (designs), before spending the big effort *doing* it (code). The V Model emphasizes that each development level has a corresponding test level, and suggests that tests can be designed early for all levels.

Models do not specify the size of the work. Effective developers have always broken projects into workable pieces, such as in iterative development (lots of little *V*s). Regardless of the absolute size of the piece, within it, the model's what-how-do sequence is essential for economical and timely success. It's also important to ensure that each level's objectives have, in fact, been met.



What V Won't Do

The V Model *is* applicable to all types of development, but it is not applicable to all *aspects* of the development and testing processes. GUI or batch, mainframe or Web, Java or Cobol; all need unit, integration, system and acceptance testing. However, the V Model by itself won't tell you how to define what the units and integrations are, in what sequence to build and test them, or how to design those tests; what inputs are needed or what their results are expected to be.

Some say testers should "use the V for projects when it applies—and not use it for projects when it doesn't apply." Such an approach would do a disservice to any model. Instead, we should use the model for those aspects of projects to which it applies, and not try to shoehorn it to fit aspects to which it's not meant to apply.

For example, the V's detractors frequently claim that it's suitable for only those projects whose requirements have been fully documented. All development and testing works better with explicit requirements and design. When these aren't formalized, such as in today's frantic "get-it-done-yesterday," document-free culture (also referred to as developing in "headless chicken" mode), developers are at as much disadvantage as are testers. The V Model's concepts still apply, but are more difficult to use when requirements aren't clearly defined. Whatever is built needs to be tested against whatever is known or assumed about what the code is supposed to do and how it's supposed to do it.

Detractors also might say the V is unsuited for techniques like Extreme Programming (XP), in which programming is done quickly in pairs and tests may be written before the code. First, let us state emphatically that we encourage both of these practices; and let us point out that XP also emphasizes finding out the requirements before coding to implement them.

The V says nothing about how or how much requirements and designs should be documented. Nor does the V, except in extensions like Graham's, define when tests of any level should be designed or built. The V says only that unit tests, such as XP stresses, should be run against the smallest pieces of code as they are written. The V doesn't define how those units should fit together; only that when they are, the integrations should be tested, all the way up to the largest end-to-end integration: the system test. Even though some XP users refer to these tests as "acceptance tests," they in no way diminish the need for users to ensure that the system in fact meets the business requirements.

Alternative X?

Perhaps partly because the V model has been around for a while, it's taken a lot of abuse, especially in these days of Internet urgency.

"We need the V Model like a fish needs a bicycle," says Marick. In his paper, "New Models for Test Development," at www.testing.com, Marick elaborates: "Some tests are executed earlier than makes economic sense. Others are executed later than makes sense. Moreover, it discourages you from combining information from different levels of system description."

The V Model's supporters and critics do agree on one thing: Although no model is a perfect representation of reality, in practice, some models are useful. Next month, we'll examine the X Model, which we've based on concepts Marick espouses.