I came across the Sept./Oct. issue on the Web. As most of my own work uses C, the theme of dynamically typed languages caught my interest. I was somewhat disappointed, however, that the guest editors’ introduction seemed to miss the mark with respect to the differences between dynamic and static typing.

In “Dynamically Typed Languages,” the guest editors espouse the benefits of dynamically typed languages over their statically typed counterparts, suggesting, for example, that “even very simple approaches to testing capture virtually all, if not all, the errors that a static type system would capture,” or that “[c]reating dynamic Web pages in languages such as C was painful, and the dynamically typed language Perl quickly became synonymous with Web systems.” While I certainly won’t argue that dynamically typed languages have some advantages—my own experience developing a Web-based system in C has been less than optimal—it seems to me that many of the assertions in the article assign too much credit or blame to the concept of data types, ignoring other facets of the languages in question.

For example, the claim that static typing offers virtually no benefit to software quality given a “very simple approach to testing” rings hollow to me. I can’t deny a certain bias coming from my own experience, but I would argue that all other things being equal, static typing provides a significant advantage to quality because it guarantees—by definition—that type-related errors can’t occur in a valid program.

As just one example, consider the case of a function that takes an integer parameter. As development proceeds or requirements change, it might become necessary to change the parameter to a structured type. In a statically typed language such as C, the program won’t even compile unless every function call is updated to pass the new type. In a dynamically typed language, on the other hand, the program will run fine unless and until it encounters an old-format call at runtime. You might expect “very simple” tests to miss some code paths, letting bugs slip by unnoticed until a particular problematic path happens to be triggered. If the language performs implicit type conversion, such as Perl with scalars and lists, the function might seem to run successfully nonetheless, causing even more rigorous tests to overlook the problem.

Now, when you consider the actual languages in common use, the extra work required mitigates this advantage to quality; the endless lines of preparation required to perform even simple tasks in C provide that much more nesting space for bugs. However, I would suggest that this difference is simply a result of the fact that languages such as Perl or Lua are higher-level languages, performing more “under the hood” than C or C++. Comparing the Perl program

```perl
my $string = "";
while (defined($string = <STDIN>)) {
    $string =~ s/first_(\[a-z\]+)/second_$1/g;
}
```

We welcome your letters. Send them to software@computer.org. Include your full name, title, affiliation, and email address. Letters are edited for clarity and space.
As software complexity increases and available development time decreases, traditional languages are simply being stretched beyond their limits.
rors to be statically detected. However, there are fundamental theoretical and practical limitations that mean that this can never happen. So, statically typed languages are implicitly able to find only a subset of errors at compile time. That means that even with statically typed languages, extensive testing is required to narrow down runtime errors. In our experience, the extra error checking designed to root out dynamic type errors is small.

Refactoring is an interesting issue. Statically typed languages can help make sure that certain types of refactorings are sound before letting a program run. However, we feel that users of statically typed languages are often less keen on experimenting with deeper refactorings; every small change requires the compiler to revalidate the entire system, when sometimes the user just wants to see what effect a refactoring has on local code. In our mind, this probably leads to a score draw.

On a final note, people who write bug-free code (with or without a compiler’s help) are few and far between. We expect that someone with such obviously high skills as yourself would probably be writing high quality code in assembler. Tools help developers, but ultimately, developers are the key.

**Simple code is beautiful code**

Rebecca J. Wirfs-Brock asked a good question in her article, “Does Beautiful Code Imply Beautiful Design?” (November/December 2007). Undoubtedly, coding is fun to most developers. However, what is beauty? Is beauty a guiding principle of programming or just an idealistic fantasy? Beauty in code is abstract and could be subjective.

I believe that simple code runs faster. Other people can easily understand, debug, and maintain it. However, could we write beautiful code to implement a poor design? Programming code is inevitably related to its design, which can affect the whole program’s outcome. If a design has a faulty structure, it’s far from concise and will certainly lead to many bugs and throw others for a loop. Good design involves a great deal of thought, creativity, and effort. In other words, design can mean everything, or it can mean absolutely nothing for coding. How can we have beautiful code without beautiful design?

“Can complex designs that are implemented by complex code ever be considered beautiful?” The purpose of design is to create an object that provides a specific set of functionalities with good structure as a result of allowing easy modification and extension. In this sense, beautiful code should be modular code that can clearly express the developer’s plan for handling all possible cases; that is, it should handle different requests in different circumstances without abnormal termination. Therefore, neither good design nor good code should be complex.

Test-driven coding—that is, preparing working test cases that programmers can test against as they code—is surely helpful for the design process. Compilation free of syntax and parse errors is baseline competence. It may evolve programmers’ thinking and encourage them to write nice code and polish it in the first place to ensure against failure.

There’s a relation between mathematics...
and programming. A strong mathematical background contributes not only to logical design but also to good programming ability. Most programmers are taught to write code but seldom to read it; nor do most of them learn logic theory. After all, beauty is only skin deep, but character is to the bone. Beauty is simplicity. Apparent correct operation isn’t sufficient, and moving on to coding before creating a proper design will not result in beautiful code.

As Buckminster Fuller put it, “When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.”

Hong-Lok Li
Information technology manager,
University of British Columbia
lib@ams.ubc.ca

Agile versus the rest

I enjoyed Hakan Erdogmus’s column “Agile’s Coming of Age … or Not” (November/December 2007). I guess I should admit to providing training in software development life-cycles, of which agile is just one model. Without getting overreligious about it, agile is just an outbreak of applying common sense to the project’s development style. The zealot approach to selling an SDLC (any one of them) is often necessary to break the chains of conventional thinking. However, as we and our processes mature, we can take a more considered view of successes, failures, and the reasons for them. For many companies, the attraction of a process is in the use of the singular … there’s just one of them. A mature approach to software life-cycle selection would look to apply process components according to the project’s needs. That can lead to cherry-picking, which can lead to team involvement, risk analysis, and other healthy pastimes. Provided that the essential delivery criteria are clear, cherry-picking is a good approach. Of course, it must be within a standardized framework.

Here in the UK, such a project management framework exists—Prince2. If you look at the various SDLCs, you can abstract from them a set of common process components. This is essentially what Prince2 is: an abstraction and a framework into which you insert whatever SDLC you want. With Prince2, water-fall works out of the box, and agile works similarly (see the recent DSDM [Dynamic Systems Development Method] Atern, www.dsdm.org/atern, for a discussion on this). You can regard Prince2 as an abstract base class from which the specific life cycles inherit.

One interesting characteristic of IT projects is relative staff productivity. In my time in Germany, where measurement was a common activity, we could measure competent programmer-productivity ratios of between 7 and 10. This variation is, in my experience, not matched in most other industries. Building and managing teams from such individuals (perfectly possible) can lead to outrageously successful projects. Such teams would succeed regardless of the SDLC, and this is why I find the emphasis on SDLCs and the vacuum around individual performance discussions quite interesting. Reports on a project’s success, regardless of the methodology, are pretty pointless without some information as to the skill set and experience of those involved. We all know that the most successful project estimates are obtained when we have prior experience with the type of project. This is also true for the execution of those projects.

I think the most useful aspect of the agile-vs.-the-rest debate has been the questioning of some assumptions about projects, project control, and methodologies, and an awareness that we can think about new ways of doing things. Add to these factors the introduction of object-oriented languages, test-driven development, and powerful development environments, and it would be remarkable if our SDLCs weren’t in need of modification. So, the discussions are good, and the pendulum will swing from side to side until we finally understand what we need from an SDLC. Then we can abstract it and cherry-pick. I think this process is under way right now.

George Brooke
Oak Lodge Consulting
george@oaklodgeconsulting.co.uk

When intuition isn’t useful

I have some thoughts on Robert Glass’s article “Intuition’s Role in Decision Making” from the January/February issue. I’ve always believed that a person must be “qualified” to have useful intuition. Only when a person has sufficient training and experience in a given field does that person’s intuition in that field become truly useful. A highly skilled software engineer’s intuition can be a valuable tool when designing a complex software system. On the other hand, that person’s intuition might not be so helpful when trying to decide what color to paint the bathroom.

James E. Hewson
Software engineer, Bellatrix Systems
jhewson@bellatrix.com

The importance of debugging

I’m a graduate student at the University of North Carolina in Greensboro in the Master of Science in Information Technology and Management program. Recently, in my systems analysis course, we read Hakan Erdogmus’s article “What’s Good Software, Anyway?” from the March/April 2007 issue. As someone whose time is partially spent in development, I very much enjoyed the article and found Hakan’s opinions both insightful and engaging. But as someone who also spends time in service and support, I was somewhat surprised to see that he had little to say about the importance of integrating serviceability and debugging into an application. Much like using coding standards and documentation, good debugging not only enables a new owner of old code to gain a better understanding of the application flows, but it also greatly facilitates the debugging and unit test of code a new owner might be adding to the application.

David T. Britt
Software engineer, IBM
dbritt@us.ibm.com

Only when a person has sufficient training and experience in a given field does that person’s intuition in that field become truly useful.