I read Warren Harrison’s “The Dangers of End-User Programming” (From the Editor, July/August 2004) with interest, since end-user computing has been a major issue at several points in my software development career. I spent 15 years in administrative computing at a major research university, and too frequently, students would be hired (with no supervision) to write department-level programs that handled every imaginable form of information—for research projects, departmental billing, financial management, student activities, and so forth. When these “programmers” graduated, there was no one to maintain the code, no one who knew exactly what the code was supposed to do, and (surprise) no documentation.

There’s another level of “programmer” that’s more immediately critical to security but often equally clueless in the discipline of software development: security and system administrators. Firewalls, routers, switches, distributed administrative tools, database management software, and so forth all involve more and more programming-like structures that require the same level of control of developed software. But the infrastructure of a software development life cycle (with requirements, design, version control, and testing) isn’t established, and many of the developers, especially in medium and small organizations, are self-taught. Mistakes by individuals at this level can have immediate and devastating consequences.

At the moment, there’s an extremely target-rich environment for security issues.

Carol Woody
Senior technical staff
Software Engineering Institute

Your column on end-user programming was really an essay about my office.

The software product we produce was created initially by the company’s founders (electrical radio frequency design engineers), who had no software engineering training save software development enabled by a language called LabView. This language is ideal for developing instrumentation-related software due to the availability of code libraries that run instruments, graph results, and so forth.

After four years of development, they reached a wall of complexity they couldn’t climb using the techniques they knew. I was hired to get them over the wall and “improve” the quality of the software.

Our first object-oriented redesign meetings were met with rejection. My coworkers were accustomed to working in isolation and were unwilling to change their habits to work as a larger team. I’d say, “We need to define the object’s interface.” They’d respond, “I won’t know what my interface will be until I’m done.” They worked from more of a tinkering approach: “get a little bit working, get another little bit working, get those little bits working together, … finished.” They couldn’t

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.
work “backwards” from the interface to the implementation. It took several months to get them to agree to define (design) their objects prior to starting work on the software. In fact, I forbade them to use the development tool until a preliminary design specification was developed.

In one and a half years, we took the software from a collection of stand-alone tools to an integrated product.

Jim West
Senior software engineer
Summitek Instruments
j.west@ieee.org

Your observations hit true to home based on my experiences in the software industry. The company I work for, Analytical Graphics, provides analysis and visualization tools for the aerospace industry. Our products’ results are often used to make mission-critical design or operational decisions. I constantly find myself on the front line working with customers to explain why there are “differences” between our applications’ answers and the ones they obtain from their home-grown products (often written by end users). I’m careful to call them “differences” and not use words like “right” or “wrong” because in the aerospace industry, many of the differences come down to understanding the modeling choices and decisions that each application encompasses. Some models are more accurate or valid than others for a given circumstance, but this is often a religious issue, not a technical one. And some customers’ home-grown tools have many years of experience and knowledge wrapped up in them and, from time to time, have exposed bugs in our tools. Following the “people in glass houses” rule, I’ve found it’s important to understand the issues that are raised before saying a particular tool or answer is right or wrong.

Given that background, though, I’d like to raise an issue that I don’t think you addressed in your article; namely, what’s driving end users to write their own software instead of relying on professionals? For some, it’s in their nature to want to learn new things. But I suspect that for most, it’s simply the desire to accomplish a given task more efficiently. I’ve known many an engineer who would cheerfully learn Matlab, Perl, or Excel from scratch when faced with performing the same repetitive analysis over and over with different inputs. They are wagering that the time spent on the front end going up the learning curve will be well spent given the efficiencies they’ll obtain over the long term (which could only be a few weeks). In a sense, they are driven by the perception that their time is better spent not doing the “boring” part—that’s what computers are for.

The quality of the resulting software products that they produce runs the gamut from “I can’t believe this actually works” to “Microsoft just made me an offer and I’m going to retire.” (The latter phrase really isn’t a quality assessment as much as it is a marketing and positioning statement, but you get my meaning.) But these tools have one
very endearing aspect—they actually do the job they are supposed to do. Not most of the job, not 99 percent of the job, but all of it. And, should there be a problem, they think they can fix it (whether they actually can find and fix it is often overestimated). But they believe they can, so they write their stuff because they have control.

So why don’t they rely on professionals? I think they would cheerfully hand it over to a professional, but they can’t find one and there isn’t any money in the budget to hire one. And the few professionals that they might have aren’t available to respond in the time frame they need because they’re booked on another project. So, Joe End User picks up his Perl book and has at it, or tries to find a COTS product that will do most of it. The question with the COTS product is, how much of the problem does it address, and is it flexible enough for the end user to solve the rest of the problem while working within the constraints imposed by the COTS tool?

Somewhere in the middle of all this, the application or tool becomes bigger than a breadbox. Now, the single end user can no longer maintain it, fix it, add new capabilities, and so on because he doesn’t have the time or knowledge to do so. Woe to us if this tool goes from a desktop hack job to get something done to a mission-critical app. Rather than lament when this occurs, we should cheer because this particular tool has survived the evolutionary process. Most end-user tools die a quick death because they are no longer needed or failed to address the problem. So die they should. But if we see one that has survived, we owe it to ourselves to recognize it, pat it on the back, and figure out what it’s going to take to either bring it up to par or replace it with something built by professionals. Management may complain that there isn’t enough money or time, and will have to make all sorts of cost-benefit decisions. But at the end of the day, it’s often the professionally built tool that wins because it offers the security and quality that the end-user version did not.

When faced with the prospect of deploying a mission-critical application, the choice between a professionally built tool and a home-grown one is pretty clear. The challenge lies in recognizing which tools are becoming mission critical in the first place. Many an Excel spreadsheet and macro has become mission critical over time because it did something so useful as to give the end user or business a competitive edge. But they didn’t know that would be the case when they first started out; it was simply something useful. I suspect this was the case in the example of the Florida contractor you mention in your article.

I normally dislike the analogy that some draw between constructing a building and constructing software. It falls apart on many levels, but in this case I think the shoe fits. The average person, when faced with the prospect of crossing a creek, will initially just wade across it. But if he’s doing it over and over, he’ll throw together a few boards and build a crude bridge. When this structure can’t handle the load, he’ll
reinforce it (or build a new one) to keep things going. At some point though, the unprofessionally built bridge falls apart (hopefully no one is hurt). Thus, the Professional Engineer is created, brought in to design the bridge correctly from the beginning, to identify when and under what circumstances it will fail, and to take personal responsibility if it doesn’t work as planned. He’s backed by a professional crew of construction workers with years of experience working these projects. The quick and dirty bridge across the creek has become mission critical to the people using it, so they’ve brought in the big guns to do it right. But if it never becomes mission critical, they don’t mind the occasional dunking in the creek when something goes wrong. To summarize, we should focus on a few key things:

■ Encourage end users to build tools that help them and make them successful.
■ Train software professionals to recognize when a tool is becoming mission critical so that they can take the necessary steps to repair or replace it.
■ Train management to recognize when they need to bring in the professionals so their “bridge” doesn’t collapse. And yes, it’s going to cost.

Tom Johnson
Vice president of engineering
Analytical Graphics
tj@agi.com

Process improvement isn’t addressing the matter of developers’ skills. Unfortunately, many full-time IT developers fit your description of dilettante programmers, notwithstanding the fact that many have had an education that ostensibly prepared them to be professionals.

This isn’t just a matter of learning the proper processes. Being able to quote chapter and verse of the CMM(I) and SWEBOK is no guarantee of being able to deliver high-quality software. This is because developing software is not like 19th-century manufacturing. At heart, it’s an intellectual activity, and it’s futile to expect that it can be reduced to some sort of paint-by-numbers exercise in the way of Winslow Taylor’s Principles of Scientific Management.

If you can’t reason effectively about abstract models of systems and the purposes they are to serve, then regardless of what process you are following, you will not detect errors or omissions in requirements, not devise effective representations of them, and not produce designs that satisfy them. The project will devolve into chaotic trial-and-error.

Agile methods offer no alternative. They emphasize a form of prototyping over analysis, but a prototype is like a scientific experiment, and no one, I hope, would suggest that science is merely a matter of following procedures.

What can be done about this situation? First, we must recognize that it exists, and IT must stop regarding developers as interchangeable parts of a machine. Second, the education of developers must put more emphasis on the application of abstract reasoning and effect the transition from working by trial-and-error to working by design. Teaching developers how to prove the correctness of software would be a good start.

The education industry has been busy satisfying the demand to appear qualified in development, sometimes by avoiding anything too demanding. This is particularly true in corporate training, where, in a reversal of educational orthodoxy, the students grade the instructors. To temper this eagerness to please, and also to balance the current preoccupation with assessing organizations’ competence, we need some independent evaluation of developers’ skills. This suggests something akin to the licensing of engineers, with the caveat that any scheme that merely sanctifies the status quo is worse than useless. The same goes for anything that is biased towards rote learning, as is the case for too many of the industry’s certification programs.

Developing high-quality software is difficult, and usually harder than we think. While a process is necessary to maximize the effectiveness of capable developers, it’s not a substitute for them, and instead of seeking a silver bullet in process reform, we should pay more attention to human intellectual skills.

Andrew John Raybould
Programmer
andrew-raybould@ieee.org

Thanks for your thought-provoking article. Two things are worth mentioning.

First, all technology suffers from the risk of defective products developed by...
unskilled practitioners; defective cars, buildings, and so forth make the headlines every day. However, the situation with (at least some) software is perhaps different: With most other products, the substantial manufacturing and distribution cost dwarfs the actual cost of development, making it more important to hire (and pay for) developers who know what they are doing. Since the manufacturing and distribution costs of, say, a Web applet are essentially zero, there is no real investment to protect by paying for skill.

Second, the idea that there are two distinct classes of software developers (“pros” and “dilettantes”), one of which uniformly turns out great software and the other which turns out junk, is neither accurate nor helpful. I’ve worked at many companies where the “pros” were turning out dangerous, poorly-designed, untested, and bug-ridden software; I’ve also worked with former dentists, cardiac surgeons, and others who have written carefully designed, well thought-out, and tested code. (There’s also no reason to assume that some languages—for example, Perl, Python, and so forth—are the province of hackers while other languages indicate a professional at work; in my experience, the choice of language on a project is almost always dictated by staff training, legacy, compatibility, tool, and other pragmatic issues, and does not correlate much with the developers’ skills.)

Much more helpful than pitting “pros” against “dilettantes,” language A users against language B users, and so forth—are the province of hackers while other languages indicate a professional at work; in my experience, the choice of language on a project is almost always dictated by staff training, legacy, compatibility, tool, and other pragmatic issues, and does not correlate much with the developers’ skills.)

I can’t dispute what you say in your editorial; the question is what to do about it.

What we can’t do about it is prevent end users from programming. Indeed, the trend seems to be toward ever more end users programming ever more software in ever more varied, and critical, contexts. Nothing we can do as software engineers, and nothing the Homeland Security people can do, has any hope of putting that particular genie back in the bottle. Whatever its limits in terms of correctness and security, end-user programming produces far too much vital software to give up. Your editorial documents this. It seems probable that end users, not professionals, have written the large majority of corporate spreadsheets, the bread-and-butter database customizations in small to medium sized companies, Web sites, and scientific software. There’s no way we can replace that volume of code, or keep up with the volume of new code needed, even if the people writing and using this software were willing to pay us to do it.

I can see only three choices for making end-user code less buggy and more secure. In the very long run, we could train coming generations of end users to write better code. This would take something like a course in software development required for high school graduation nationwide. The course would have to teach some end-user language but should spend as little time on that as possible. The meat of it would be teaching debugging, testing, and security practices. I’m not optimistic about seeing such a course requirement, and if we do see it, I’m not optimistic about seeing it done properly. Even in the best case, this approach won’t yield any significant benefits for years (but once the benefits arrived, they would be permanent and widespread).

In the shorter run, we can try to educate existing end-user programmers. This is what the EUSES (End Users Shaping Effective Software) project is attempting. I wish them luck—until I read your article, I’d never heard of them (or anyone else doing the same job). If I’ve never heard of them, how many end users have? The job of reaching fifty or sixty million people who don’t think they need our help isn’t going to be easy.

That leaves one final approach: tools engineering. End users use software tools—spreadsheets and databases, scripting languages and HTML editors, and so forth—to program. These tools can’t be designed to completely prevent mistakes, but they can be designed to make them more or less likely. A badly designed tool can make certain classes of programming mistakes almost inevitable. On the other side of the coin, a well-designed tool can eliminate some classes of problem altogether, or greatly limit them. (That was the whole point of the Java sandbox.) Since professional software engineers generally write the software tools end users use, this is where we have the most leverage to influence the correctness and security of a huge amount of very important code. This might require some changes in our design practices, though. We’ll need to think about “usability” in terms of the kinds of software our software makes it easy or hard for our users to develop.

We’ll need to think about the security not just of our code but of the code developed with our code.

If we try to limit problems by limiting our users, they’ll just use somebody else’s product. We need to actively encourage safe practices, not try to fence the users in.

Randall Smith
Staff software engineer
 QUALCOMM
rsmith@qualcomm.com

In the very long run, we could train coming generations of end users to write better code.

George A. Rappolt
Principal software engineer
 Hologic
grappolt@hologic.com
As software is encompassing every aspect of society, it’s only natural that individuals from every aspect of society should express themselves through software.

Is this a threat to the jobs of some professional developers? Yes, surely. As the art of programming is no longer viewed as arcane, a lot of skilled professionals will be able to create solutions that work for them in their domain. This will lessen the demand for some software engineering work.

Does this imply that the software constructed by end users is inherently less reliable? Definitely not. A majority of professional software developers overestimate their understanding of the domain and underestimate the effort required to produce quality software.

Unskilled end users often have a more down-to-earth approach to software development. They know their shortcomings and seek advice from software professionals when needed.

Most colleges, universities, and technical schools don’t teach their students how to engineer software. They teach some basic programming, algorithms, operating system theory, modeling, analysis, requirements gathering, and so forth. The art of constructing quality software isn’t what you’re taught at school. You have to pick that up outside of the academic world.

Sure, I have seen end users write terrible software, but I’ve seen so-called professionals do even worse. The gain of having end users with software development knowledge is immense. It significantly reduces communication barriers between software developers and end users, resulting in more correct software being built to meet real demands of the business.

Today, one of my colleagues, an end user from business school, completed a scripted regression test suite for the system we are maintaining. The suite will complement our other test suites perfectly, but it outshines them with respect to the way it does functional testing of our value chain. Only the end user was capable of doing that, with an end user’s inherent knowledge of the value chain.

Gunnar Ronning
Software engineer
gunnar.ronning@telenor.com

Warren Harrison responds:
I was extremely gratified to receive the many emails and letters commenting on my column in the July/August 2004 issue on the dangers of end-user programming. Many of these letters shared three major observations.

1. End-user programming is here to stay.
   As soon as we started distributing personal computers with BASIC embedded in ROM in the late 1970s, the province of programming was no longer limited to professional developers with formal training in software development techniques. The introduction of spreadsheet macros and query-by-example interfaces to databases were just more nails in this particular coffin. Trying to dissuade end users from programming would be as effective as teaching abstinence in high school sex education classes. The genie is out of the bottle, and we’re not going to get the cork back in.

2. We can make the situation better by targeting end users with tools specif-
ically designed to leverage their domain knowledge while minimizing the propagation of unexpected behavior. Is there any reason Hal in accounting really needs to be able to establish a socket connection with a remote machine? Should we really expect Joe in personnel to develop the search algorithms for the corporate directory lookup system?

Tools to both provide constrained development frameworks as well as libraries of components that end users can safely plug together will contribute greatly to addressing the concerns many of us have regarding end-user programming.

3. Programming isn’t easy or error free, even for “professional developers.” Why pick on end users?

As many of you pointed out, often the domain knowledge that the end user brings to bear on a problem enables them to develop software that may rival the professionally developed version. However, that personalized domain knowledge also creates the problem. In today’s hyper-interconnected environment, a singular focus on the problem domain without a corresponding focus on the environment in which the application must operate puts us all at risk. So this application opens some ports? It uses a user-entered variable as an email address without checking for meta characters? Can professional developers write such software? Of course they can, but we have a right to expect better from professionals. Can end users be taught to consider these issues? Of course, but once they’ve taken that step—from figuring out how to copy instructions from the book to the editor to solve their individual problem, to the bigger picture of how their artifacts relate to the rest of the community—they’re really not the kind of end-user programmers I originally wrote about, and I welcome them to the community of software developers.

I invite interested readers to visit the European Spreadsheet Risks Interest Group at www.eusprig.org for more information about the risks of end-user programming.

**Letters**

**ROI Article: Clarifications and Corrections**

I very much appreciated finally seeing some efforts to quantify ROI for software projects or process changes (“Calculating ROI for Software Product Lines,” Günter Böckle et al., May/June 2004). I specifically enjoyed the scenario about software product lines, which clearly illustrated an otherwise hard-to-approach topic. However, I found some issues that should be clarified, as the topic is increasingly relevant and certainly needs a good foundation—as the authors attempt.

The authors introduce one of the most famous discussions around reuse, namely the degree of commonality. This concept is key, as it drives all further decisions. If commonality is below a certain margin, depending on multiplicity of releases, the value proposition will change dramatically—and can even turn to negative numbers. Unfortunately, this discussion was extremely short and not really based on any tangible experience. Fair enough that they suggest a dedicated exercise. But where does the number 70 percent come from? Why such a high number? Why is it “probably closer to 70 percent”? The entire article is laid out as a hypothetical discussion around a single example, but this topic is too sensitive to remain just a number. And the entire ROI reasoning is based on this number. So, although the proposed approach is basically helpful, it lacks guidance for application. It’s valid for a situation involving high commonality and many variants of the asset base over a short period of time. It would have been helpful to show the major impact factors’ dependencies, namely degree of commonality and number of variants. Often commonality is overestimated. With fewer variants and less commonality, the ROI is negative.

Page 28 explains that $C_{cab} = 150\% \times 70\% \times C_{prod} + C_{scoping}$. Later it sets $C_{unique} = 30\% \times 20\% \times C_{prod}$ per product. However, it remains unclear how this will sum up for the first concrete product built from the asset base. Actually, the marginal cost of the first real product is at least $C_{unique} = 30\% \times C_{prod}$. It can’t be reduced to 20% because there’s nothing to build in parallel. In my experience, it typically costs much more because of integration, stabilization, and “gluing” cost (that is, making the first concrete product out of an initial asset). Dramatically underestimating costs makes consulting easy but will backfire in real-world situations.

Page 28 also suggests that sometimes it would be better to redo the entire code asset to build the asset base rather than extracting it from real code. That might be correct if maintainability is terrible, but again, I disagree about its cost. Extracting stable code is normally less expensive because it’s already stabilized from field usage. This decision depends on stability, maintainability, commonality, and multiplicity of variants later on.

Finally, Figure 1 has reversed the colors indicating cost evolution for product line development versus the traditional approach.

Christof Ebert
Director of software coordination and process improvement
Alcatel
christof.ebert@alcatel.com

Günter Böckle, Paul Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid respond:

Thank you for taking time to comment on our work. Our purpose is to develop a structured model that not
only supports high-level, quick computations to guide managerial decisions but also, with appropriate implementations, supports lower-level, more detailed computations that guide project planning. We only presented our high-level model in this article. We’d like to address each of your points.

Our article was an effort to explore a situation in a somewhat unique manner. This didn’t leave room for an indepth analysis of commonality’s impact on ROI. The 50 percent and 70 percent figures are purely assumptions of the scenario. However, the point of those assumptions was that in our experience (in contrast to yours), the amount of commonality is often underestimated. The scoping practice of a software product line is intended to discover much more commonality than is casually apparent. In fact, we’ve worked on many product lines where the level of commonality reached 90 percent. Obviously, the amount of commonality among a set of products depends totally on the set of products belonging to the product line. As you point out, the ROI for a proposed product line might be negative if the amount of commonality is too small. The point of developing ROI analysis techniques is that the product line approach isn’t always advantageous, and this computation is needed to determine when it is and is not appropriate.

We certainly agree with you that the first concrete product will identify areas in the core asset base that need to be modified. This becomes a matter of bookkeeping. We would add the cost of that modification to the cost of the core asset base rather than to the product. The first team is pioneering for that follow, and their cost basis shouldn’t be penalized. This can be accomplished in various ways. Our approach leaves the method of implementing these cost functions open for each product line organization.

On page 28, we say “Assume that it’s easier to build the core asset base from scratch.” Again, this is purely an assumption of the scenario. In practice, we’ve seen both extremes and situations where some of the core assets come from existing code and some from scratch. We agree that a number of factors influence which is easier.

Regrettably, there were two errors in Figure 1. A corrected version is included here. The original legend in Figure 1 reversed the graphs’ colors. Also, the original graph used a value for $C_{\text{reuse}}$ that assumed the maintenance mode, which was the main point of the scenario $30\% \times 20\% \times C_{\text{prod}}$, rather than the “from scratch” cost of $30\% \times C_{\text{prod}}$, which was the point of the “reality check.” This error only applied to the “reality check.” While the values changed, the break-even point didn’t change, and the error didn’t apply to any of the computations used for ROI or to the stated conclusions.

![Figure 1. Part of the spreadsheet Sebastian created for Kurt (corrected version).](image)

**Friendly MDA Amendments**

Thank you for publishing Dave Thomas’s excellent article “MDA: Revenge of the Modelers or UML Utopia?” (Design, May/June 2004). While I agree with Thomas’s observations about UML, especially about the bad and the ugly, and about the need for domain modeling and domain-oriented programming, I have some friendly amendments that may provide a (somewhat) more optimistic approach to MDA.

An essential part of MDA that tool vendors, and even many users of MDA, do not mention, is the computation-independent model—“sometimes called a domain model.” As the MDA Guide points out (www.omg.org/cgi-bin/doc?omg/03-06-01.pdf), the CIM bridges the gap between domain experts and design and construction experts. Perhaps CIM isn’t so popular because code cannot, and should not, be automatically generated from it. At the same time, the idea of a domain model that bridges the business-IT gap by using concepts and constructs understandable to both business and IT experts is very helpful.

What are these concepts? Where do we get them? We certainly don’t want to invent them for every project or stage of a project. Fortunately, in 1995, the ISO published a standard defining a precise, well-structured system of common concepts: the Reference Model of Open Distributed Processing.

RM-ODP specifies semantics in a syntax-, methodology-, and tool-neutral manner. It provides precision without programming. The Foundations of RM-ODP are very short—only 18 pages. The reader doesn’t have to figure out what a particular term means, or guess about concepts left undefined because “everyone knows what this means.”
How often have we overheard or participated in violent discussions about the “real” meaning of a class, a type, a component, a composition, an aggregation, an activity, and so on? The relationships among these concepts are also precisely defined in RM-ODP. Nothing is radically new here.

Many of these concepts have been formulated and discussed for years in IT, mathematics, philosophy, systems analysis, and programming. Some have been used in engineering, business, law, and other fields for centuries. And they’ve been used successfully in various modeling approaches within the framework of the three-schema database architecture. The (extensible) system of these concepts defined in RM-ODP can be (and has been) used to specify not only business domains, but also the semantics of IT systems and software components like those mentioned by Dave Thomas. Just as American businesses rely on the standard Uniform Commercial Code, system specifiers ought to rely on the standard RM-ODP.

Haim Kilov
Independent consultant
Affiliate professor, Stevens Institute of Technology
haimk@acm.org

Dave Thomas responds:
Thanks for your letter. My comments focused on the current MDA/UML hype to provide a caution when one sees that programs can be automatically or “automagically” generated from models.
The RM-ODP is indeed an elegant metaframework. However, as with most metamodels, the correct mapping of concepts into the metamodel where it is formal or structured English still remains a challenge. I agree that CIM models are very useful, but it’s not the focus of current models to code transformation approaches of MDA proponents. Clearly, we want to extract the best ideas from RM-ODP, CIM, and so on, but this requires accepting that there’s an inherent translation from domain space to code space that machines might not be able to easily automate.
The alternative is to avoid Esperanto and complex translators (transformations) and instead focus on letting users express their computations directly in their own language—that is, to raise the level of the computational substrate on which they work.

Tata Research Development and Design Centre
Pune, India

Tata Research Development and Design Centre (TRDDC) is a leading applied research center for software engineering and process engineering in India. Since its inception in 1981, scientists at TRDDC have developed advanced solutions for a range of highly complex industrial requirements, ranging from operating systems and compilers in the 1980’s to decision support tools and model-driven architectures. TRDDC works in collaboration with industry groups as well as funded teams from leading academic institutes. Software tools developed at TRDDC are used for industry-leading applications and have resulted in significant improvements in productivity and quality. More details about TRDDC can be found at www.tcs-trddc.com.

Applications are invited for research positions at TRDDC to augment existing groups and grow new groups. Areas of interest include decision support systems, natural language processing, requirements analysis, program analysis, software architectures, formal methods, software testing and automated software development especially for embedded systems and distributed systems, materials science and process engineering. Successful candidates may be placed at Pune, Hyderabad, Bangalore or Delhi in India depending on the area of work. TRDDC offers a compensation package on par with the Indian software industry.

Applicants should have excellent academic qualifications and research orientation with a minimum of 2-3 years of R&D experience in leading academic or industrial centers. A PhD in a relevant Computer Science, Process Engineering or Material Science discipline would be an asset and fresh doctoral graduates are particularly invited to apply. Also welcome are applications from more senior R&D staff with work of international standing.

Applications should include a full CV with details of academic qualifications, research experience, publications and patents and employment history. Names of three independent references who can be contacted for confidential reports should be provided.

Please send applications by October 31, 2004 to careers.hrd@tcs.com with the subject: “TRDDC Recruitment” or send to: TRDDC Recruitment, 54B, Hadapsar Industrial Estate, Hadapsar, Pune – 411 013, Maharashtra, India.

Tata Research Development and Design Centre is the R&D center for Tata Consultancy Services, India’s first and largest software consultancy organization (www.tcs.com).

TCS is an equal opportunity employer and committed to recruiting/hiring based solely on merit.