Report on OOPSLA 2001

James W. Cooper

OOPSLA is the Conference on Object-Oriented Programming, Systems, Languages, and Applications, sponsored by the SIGPLAN and SIGSOFT divisions of the ACM. This conference is one of the premiere OO programming conferences in the world and is one I highly recommend for any Java programmer. Unlike JavaOne, which deals primarily with products and low-level tutorials, the papers at OOPSLA have real meat, and provide a lot of valuable guidance. In addition, the opportunity to interact with prominent and skilled OO programmers from all over the world is invaluable.

This most recent OOPSLA was held in Tampa, FL in mid-October of 2001, and was attended by about 1350 programmers and technical people. The conference is quite selective, with only 27 full papers being accepted from over 140 submitted, but it also offers 70 different tutorials, 31 workshops, 20 demonstrations, 9 practitioner reports, 63 posters, and 4 special papers on Intriguing Technologies.

Needless to say, it is impossible to attend or absorb anywhere near everything at this conference, but the opportunity to interact with others in the field during breaks and social events is just as important as the formal program.

The conference began with a keynote lecture by Henry Petroski, a noted professor of Civil Engineering at Duke University, on Success and Failure in Design. The thrust of Petroski's lecture was that the design of large architectural elements such as bridges went through a number of evolutionary steps, starting with limited understanding of the physical forces and material strengths and weights and moving towards much more sophisticated design. The key to success in building these structures is a thorough understanding how failure can occur, and design that includes allowing for that failure. He noted that every time a major advance in construction techniques and bridge size took place, the new advance persisted for about 30 years, or about 1 architectural generation, before new failures began to be observed, usually because of lack of sufficient understanding of the various possible modes of failure that were already know. It is certainly Petroski's hypothesis that software architecture can and does benefit from thorough analysis of possible kinds of failure. He also feels that this is seldom actually done.

In another keynote address, noted consultant Tom DeMarco emphasized the need for *slack*, or room in people's schedules to innovate and change the way they do things. If they do not have this slack, their methods of carrying out their jobs can never inprove. This is the problem with workers being excessively busy and under the gun to produce constantly: they never have time to figure out how to do anything better.

DeMarco also reported on a study of "Death-March Projects," or projects that absolutely had to be done by a specified date, regardless of how much overtime the programmers had to put in. He noted that without exception, these projects were of stunning insignificance, not ones of major importance to the company. He explained that the real reason for Death-March projects was not their importance, but a desire to get then done

with minimum cost: keeping the number of days spent under control made these project less expensive. If the projects had really been that important, management would have been likely to spend more on them and let them go on longer.

In a session on Java performance issues, Bowen Alpern and his colleagues at IBM Research studied interfaces in Java. One of the reasons that Java is a single-inheritance language is to reduce complexity both for the programmer and the language implementer. You can approximate multiple inheritance through the use of interfaces, and in general, the compiled code for carrying out interfaces and inheritance at once can be quite complex and slow to execute. In fact, interfaces make some typical Java JITs up to 50 times slower, since the compiler must generate code to figure out whether an instance is of one subtype or another. However, the group reported that they had developed Interface Method Table (IMT) algorithms for handing this complex situation much more rapidly and the techniques they used in their research-grade Java compiler (Jalapeno) will presumably end up in product-level compilers in the near future. The message: don't shy away from using interfaces and inheritance in your programs. Any performance problems are at best temporary.

Don Box from DevelopMentor discussed the XML InfoSet as a high level representation of a data model in XML. He expressed the opinion that if the InfoSet representation had been completed sooner, then the development of SOAP would have been unnecessary, and that SOAP is now essentially obsolete. This view was not necessarily shared by panelists in a later panel on the future of distributed objects. Panelists from Microsoft, IBM, Oracle, Compaq and the Object Management Group all expressed support for SOAP in one degree or another.

Posters

I sometimes think that the posters at a conference can be the most interesting part. Posters usually represent work that is still in progress or just being finished. The authors are also frequently recent graduates with fresh ideas that are worth listening to. In a paper on "Why Java is not Suited for Object Oriented Frameworks," Dragos Manolescu and Adrian Kunzle rightly pointed out that the most common Java collections and interfaces return instance of a class of type Object and that the programmer must carefully cast these to the correct type to get the system to compile and execute properly. In fact, Java's static type checking provides a "false safety net," because of the explicit casts, which may not work at run-time. They did not, however propose any direct solutions to these problems, but wanted primarily to make the development community aware of them.

In another paper called "Must Java Development Be So Slow?" Albrecht Woss noted that continually starting up new instances of the JVM took a lot of unnecessary time. He is building a system in which there is one JVM running continuously and which will execute whatever Java is needed for compilation and execution. He plans to solve the problem of namespace collision between different versions of the same class executing at the same time, much as .NET does, by specifically making every process part of a separate package.

John and Elizabeth Towell of Carroll College reported on a MOO programming environment that gives the students the feeling of "being there," in which they can hold

objects and examine their methods as part of a simple adventure-like game. The point of the game is for students to gain a real internalization of what an object really is by feeling as if the objects were actual objects in the enviroment. The students were not computer scientists, but business school students, and most of them rated the project most highly as helping them internalize object concepts.

Much More on Extreme Programming

One of the buzzwords *du jour* this year was definitely Extreme Programming (XP). There were at least a dozen new books on this discipline and any number of talks, workshops and tutorials. As you will probably recall, the main tenets of XP are: Planning with customers, Functional testing and unit testing, Refactoring, Simple design, Simple code metaphors, Collective code ownership, Coding standards, Continuous integration of changes, Customer on site, Open workspace, the Forty Hour week and Pair programming. The two most stressed of these are the open workspace and pair programming.

Alistair Cockburn, who is a major XP evangelist, gave a talk analyzing office layouts and showing how private offices with inconvenient walks between them wastes so much time (say a minute per question) that each it can raise that cost of a nominal 6-month project by at least $50,000. The idea was that if you have to wait each time you need to ask a colleague a question because they aren't there or are busy or on the phone in their office, your productivity decreases significantly. He also talked most volubly about how information wafts through the air like perfume, and having programming projects carried out in an open close-packed work environment allows team members to overhear useful exchanges and participate more actively in development decisions. He also referred to an MIT Sloan School study by Thomas Allen that he said supported these arguments.

I remain of mixed minds about all this. There is no question that pair programming has been demonstrated to be valuable in team development projects, but some of rest of this sounds a like a bit too much soap for polishing the hog. People do not always work effectively in noisy environments, and not all programmers have identical or even pleasant work habits. The idea of doing all my work and development cheek by jowl with others seems to me to be very much of a distraction. Not all programming, even in a large team project is that interdependent. People do have their own specializations that they need to work out quietly before introducing them into the project.

But Are You Agile Enough?

The other new buzzword for the year was Agile Methodologies. The more I heard of this, the more I wondered if the Emperor really had many clothes on. The whole idea of the process is simply to be flexible enough to change anything you need to change in your development scheme at any time rather than getting locked in to the "way we've always done it." The books on agile development methodologies I thumbed through seemed to consist of a lot of fairly obvious ideas, stated rather repetitively. I don't dispute that the ideas should be followed, but the emphasis seemed a bit excessive. I'm sure, however, that someone will correct me on this if I am way off base.

Intriguing Technology Papers

Four papers were set aside as "Intriguing Technology," because although the works were not altogether original in the academic sense, they were so interesting and important that the organizers felt they should be heard and discussed.

One of the first of these I heard was a paper by Yoder, Balaguer and Johnson from the University of Illinois, on the "Archetecture and Design of Adaptive Object Models." Their central thesis was that changes in objects should take place in instances of the objects rather than in classes themselves. Thus, they developed a system where class behavior could be described in a database, and that description could be changed by the system's users rather than be programmers. This made it possible for the object model to adapt and produce new behavior without rewriting the code.

Finally, a very interesting paper by Richard Pawson, Simon Dobson and Robert Matthews described a radical approach to business systems design, in which the users operated directly on the objects. This is such an intriguing and powerful piece of work, that I'll discuss it in detail in a forthcoming column.

Summary

OOPSLA is a premier technical conference for OO programmers. I can't recommend it highly enough. One final note: a language called Ruby is now being promoted by those of the Smalltalk persuasion. There is, of course, a new book on Ruby just out. I was disappointed that it didn't have a Ruby-yacht on the cover.

References

1. Tom DeMarco. *Slack: Getting Past Burnout, Busywork and the Myth of Total Efficiency*, Broadway Bopoks, 2001.
2. Thomas J. Allen, *Managing the Flow of Technology*, MIT Press, 1984.
3. Alistair Cockburn, *Agile Software Development*, Addison-Wesley, 2001.
4. XML Infoset: www.w3.org/TR/xml-infoset/