XOOTIC magazine

June 2001-Volume 9-Number 1

POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

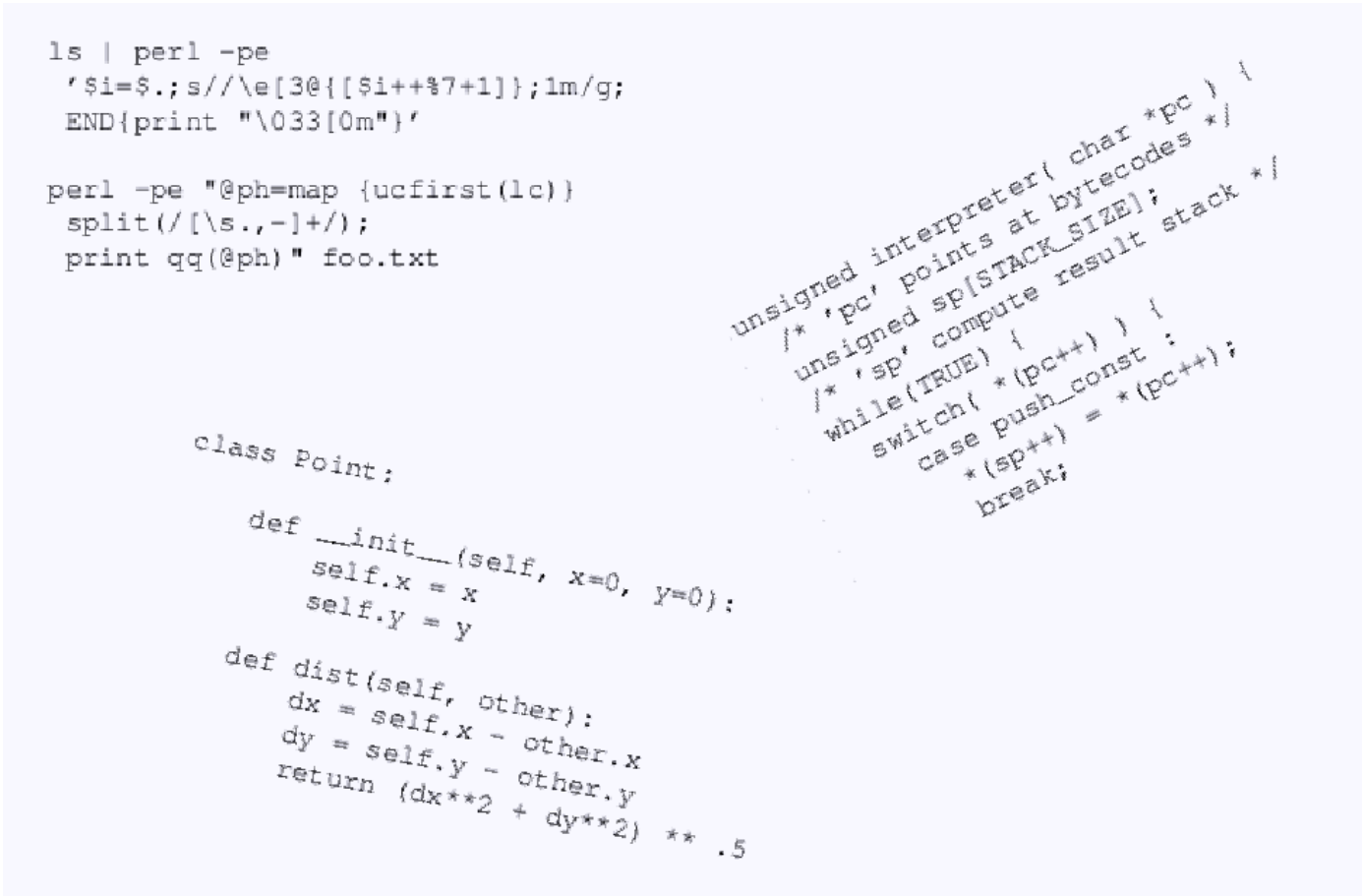
Programming Languages

Perl

Python

Embedded Java

Xootic Survey 2000



Contents

Programming Languages
Editorial Preface 3

XOOTIC Survey 2000
Gertjan Schouten (on behalf of the survey committee) 5

Perl
Ed Knapen 11

Java in Embedded Systems
Menno Lindwer 15

Python
Victor Bos 27

Overview Latest OOTI Reports
. 34

Advertorials

AAS 4
Philips Nederland 10

Colofon

XOOTIC MAGAZINE
Volume 9, Number 1
June 2001

Editors
V. Bos
N.H.L. Kuijpers
Y. Mazuryk

Address
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122
5600 MB Eindhoven
The Netherlands
xootic@win.tue.nl
http://www.win.tue.nl/xootic/

Secretariat OOTI
Mrs. C.I.T. Kolk-Koenraat
Post-masters Programme Software Technology
Eindhoven University of Technology, HG 6.57
P.O. Box 513
5600 MB Eindhoven
The Netherlands
tel. +31 40 2474334
fax. +31 40 2475895
ooti@win.tue.nl
http://wwwooti.win.tue.nl/

Printer
Offsetdrukkerij De Witte, Veldhoven

Programming Languages

Editorial Preface

An important choice to be made for every software project is the choice for a suitable programming language. This choice is often made implicitly as a project is usually the follow-up of an earlier one, or is part of a larger development programme. If a project team has a choice at all, this choice is often determined by experience within the team, processing power and memory consumption, or available libraries. Sometimes a project team explicitly decides to learn a new language and even migrates existing software to that new language. In this way object-orientation is introduced in many organisations to obtain more flexible, reusable and extendable code. Such a migration path is a large investment which often only pays off in future projects.

In this issue of the Xootic Magazine three programming languages are subject of discussion. You will learn why a scripting language like Perl should not only be used by system administrators, how Java can be applied to program embedded systems, and how Python is not only used for prototyping but can be used to build the final product as well. Besides all ins and outs of Perl, Embedded Java and Python, you will also learn about the results of the Xootic Survey that was filled in by almost half of the Xootic members last Fall.

Before diving into the contents of this magazine, we would like to take the opportunity to introduce a new editor and say good-bye to another. For several years, Eibert Engelsman has been one of the editors of this magazine. We would like to thank him for his inspiring ideas, his contributions to the editorials and cover arts, and of course for the hard labour of reviewing and editing a large number of articles. We are also happy to introduce to you our new editor Yarema Mazuryk. He has just recently started the OOTI programme and is already willing to be active for Xootic as an editor of the Xootic Magazine.

The magazine blasts off with the results of the Xootic Survey 2000 drawn up by Gertjan Schouten, on behalf of the Xootic Survey Committee. Next, Ed Knapen describes the post modern philosophy behind Perl. He not only briefs us on the language itself, but also explains the link between Perl, Apostle Paul and perlmonks.com. Menno Lindwer advocates that although Java is certainly not the most obvious language for embedded systems, recent developments in both hardware and software make that Java will become a more logical choice for future, perhaps interconnected, embedded systems. Finally, Victor Bos leads us through something completely different: the Python scripting language, or, rather, programming environment.

Enjoy!

Nico Kuijpers
Editor

Advertorial: AAS

Page 4 (should be even)

XOOTIC Survey 2000

Gertjan Schouten (on behalf of the survey committee)

In September 2000, the bi-annual Xootic questionnaire was sent out again to all Xootic members to ask them about their current and future work, and about their opinion of OOTI and Xootic. In the past months, the returned questionnaires were analysed and the results were presented to the Xootic members in March 2001. This article presents the survey results.

Introduction

The Xootic survey has become a two-yearly tradition. It provides valuable feedback to both the OOTI and the Xootic board on their program and their activities. Previous surveys were held in 1993, 1994, 1996 and 1998 (see Xootic magazine September 1993, September 1994, April 1996 and October 1999, respectively). In the beginning of 2000, Rian Wouters, Dietwig Lowet, Harold Weffers, Bernard Venemans and myself set out to organise the survey for 2000. The first thing we did, was to take the previous questionnaire and modify it according to suggestions for improvement that resulted from the previous survey.

The major changes as compared to the previous questionnaire are:

- The language: this was the first questionnaire in English.
- The OOTI questions: these were formulated by OOTI to ask for specific information needed to improve the OOTI program. In this survey, only the younger generations were asked to answer these questions.
- More predefined options were added: some extra answers, provided to us in the 'other' option-field of the questions from the previous survey, were added in the answer-lists.

Furthermore, the questions about current and future function/working environment have been combined, and the split in generations has been extended in the sense that the older generations did not have to fill in the OOTI questions: the OOTI program has changed a lot since those generations followed the program, a.o. as a result of their answers to previous surveys!

You have to be careful not to change too much in a questionnaire, otherwise the results are difficult/impossible to compare with previous surveys. That is why the remainder of the questions were left (more or less) unchanged. The questionnaire, together with a memo from OOTI about the new Software Technology Training Program, was sent to every Xootic member early September 2000. Table 1 shows the number of surveys that were sent out and the number of surveys that were returned this year as well as previous years.

Survey	Nr sent	Nr received	Percentage
1993	22	17	77%
1994	41	24	59%
1996	88	43	49%
1988	155	69	45%
2000	189	88	47%

Table 1: History of returned questionnaires

Table 2 shows the returned questionnaires per generation. We see a large increase of returned forms for the generation "September 1996 - April 1998". The probable explanation for this is that this generation has a lot of non-Dutch members, who had difficulties filling in the Dutch questionnaire of 1998.

Generation	1998	2000
1988 - Dec 1991	16	15
Sep 1992 - Jan 1994	18	19
Sep 1994 - Mar 1996	26	25
Sep 1996 - Apr 1998	9	15
Aug 1998 - Aug 2000		12
Unknown		2

Table 2: Number of returned questionnaires per generation

The questionnaire was returned this year by 76 ex-OOTIs and 12 OOTIs. In the results concerning professional career the answers from the 12 OOTIs have been excluded. Hence in these results, about every 1.3% is one person.

Employer

The questions about the current employer are intended to get an impression of the employers where Xootic members are working. Figure 1 shows the major branches where ex-OOTIs are currently working. Compared to the result of the previous survey, there are few changes. The most striking changes are: Automation consultancy has increased from 2% to 11% and Electrical industry has dropped from 15% to 7%.

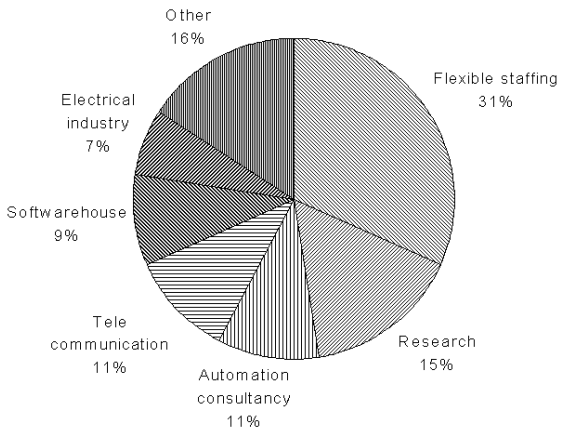


Figure 1: Branch distribution

It seems that switching jobs after 4 to 6 years is becoming a trend! If you compare the numbers between 1998 and 2000 (Table 3), you see that after 4 to 6 years, the majority of ex-OOTIs has left their first employer. Note that the percentage of persons at their first employer in the generation ”1988 - December 1991” has increased from 25% to 27%, however both percentages represent four persons.

The main reason why the current employer was chosen is *nature of the work* (rated 7.9 on a scale from 0 to 10) followed by *career perspective* (6.2), *company culture* (6.1) and *salary* (5.8).

Figure 2 shows that the final project of OOTI and (nowadays) a direct approach by the company or a person working for the company are the most successful strategies for recruiting ex-OOTIs. The ”open application” dropped from 47% in 1996, via 25% in 1998 to 14% in 2000!

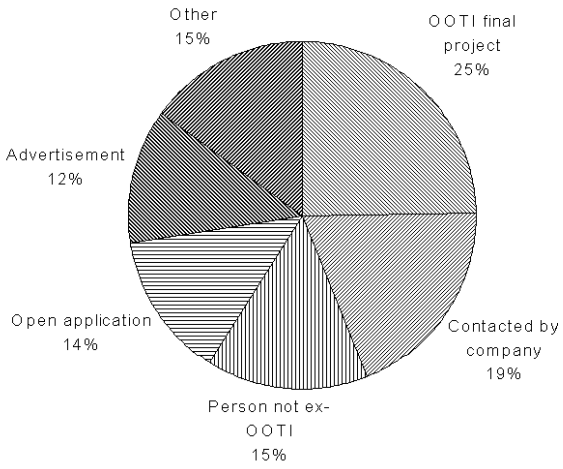


Figure 2: How did we get our current job?

Function

The results of the questions concerning current & future function and working environment tell us something about our daily work and our expectations. If you look at Figure 3, you will notice that the Xootic members currently still have very technical jobs: 73% are software/system engineer/architect or researcher, compared to 64% two years ago.

Generation	1st employer 1998	1st employer 2000	2nd employer 1998	2nd employer 2000	3rd employer 1998	3rd employer 2000	4th employer 1998	4th employer 2000
1988 - Dec 1991	25%	27%	69%	40%	6%	13%		20%
Sep 1992 - Jan 1994	61%	32%	28%	47%	11%	16%		5%
Sep 1994 - Mar 1996	96%	76%	4%	20%		4%		
Sep 1996 - Apr 1998		87%		13%				

Table 3: Number of employers

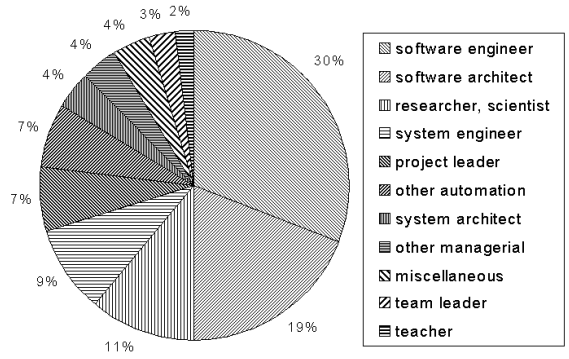


Figure 3: Current functions[1]

The 'future' situation (i.e. the desired function within 5 to 10 years) is shown in Figure 4. Just as two years ago, nobody plans to be (or become) a *software engineer* 5 to 10 years from now. Approximately 44% wants to advance in technical functions. The managerial functions (team/project leader, other managerial and miscellaneous) have grown from 18% now to 42% within 5 to 10 years.

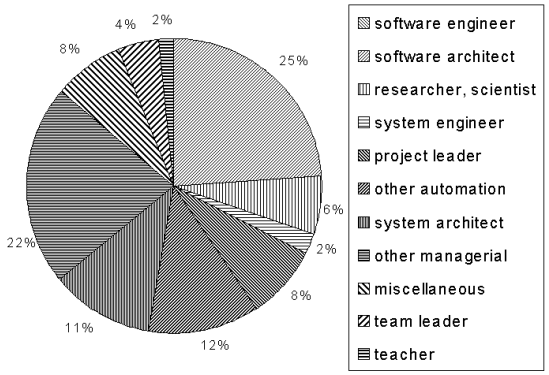


Figure 4: Future functions

Ex-OOTIs are working in a less multi-disciplinary environment! All areas of expertise showed a drop in percentages (*Logistics* dropped most from 18% to 10%). The top-three of disciplines that ex-OOTIs come into contact with in their daily work (not including *Computing science*) is as follows: *Electrical engineering* (43%), *Telecommunication* (31%) and *Information technology* (28%).

Skills

There were several questions concerning the tools and methods that are used in the workplace of ex-OOTIs. One positive side to OOTI: formal methods are now being used! 16% of the returned questionnaires indicated that Chi, (Process) Algebra, Spin/Promela, State machines, START, Petrinets in Expect/Cosa, MSCs, IDL, OCL or process Networks were used in the direct working environment.

Object-oriented techniques are used more often: UML is used by 64% (was: 32%) and OMT is used by 33% (was: 37%). Design Patterns are used by more than 50% of the Xootic population. C++, C and Java are still the main implementation languages being used (63%, 57% and 50% respectively, versus 65%, 58% and 43%, two years ago). One third indicated to work with scripting languages such as VB Script, Perl, Python or Tcl/Tk.

Windows NT is used most frequently as a host platform. The dominant target platforms are: Windows NT (56%), Unix (43%), Java platform (34%) and pSOS (27%). Linux is being used as a target platform by 22%.

Distributed - and Component technologies are very popular amongst ex-OOTIs: around 40% uses these technologies and the same percentage of Xootic members are interested, and around 10% has taken courses in these technologies. 74% uses HTML, 35% is interested in XML.

The waterfall model is still the most widely used process model (34%), followed by RUP (30%).

Xootics are most interested in Extreme Programming (32%) (hype?) and RUP (30%). Xootic members want to know more about the following skills: Coaching, Creativity (both 23%), PSP/TSP (19%) and Project Management (18%).

Working conditions

This section gives us an indication of the conditions of employment. Table 4 shows the current salaries of the 72 ex-OOTIs that filled in the question.

In this survey, we added questions about part-time work and RSI. Currently, 6 ex-OOTIs (equals approximately to 8%) are working part-time (32, 36 and 38 hours a week answered by 3, 2 and 1 persons, respectively) and 22 ex-OOTIs (29%) would like to work part-time (24, 32, 34 and 36 hours a week reported by 1, 15, 1 and 5 persons, respectively). 47% of the ex-OOTIs reported to have no signs of RSI, 42% responded ”sometimes”, 3% ”quite often” and 8% ”very often”.

OOTI training program

The questions about the current Software Technology program had a completely new form. The current courses were listed and the trainees of OOTI that started their program after August 1994 were asked to indicate the value/usefulness of the individual courses and whether they have applied the knowledge gained from the courses in their work. Finally, one was asked to indicate the amount of time OOTI should allocate to each course.

The top 5 of *most useful* courses:

1. Industrial Design and Development Project
2. Workshop Software Engineering
3. Object-Oriented Analysis and Modeling
4. Technical Writing and Editing
5. System and Software Architecture

The top 5 of *least useful* courses:

1. Workshop on Declarative Method (PVS)
2. Workshop on Constructive Method (SPIN)
3. Formal Methods in the Software Life Cycle
4. Seminars with Industry (FM)
5. Control and System Theory

According to the ex-OOTIs, the program should allocate *more time* to:

1. Design Basics (+202%)
2. Development Environments (+56%)
3. Requirements Engineering (+41%)
4. Personal Software Process (PSP) Basics (+41%)
5. Software Process Improvement (SPI) Basics (+37%)

The results of the survey indicate that the OOTI program should allocate *less time* to:

1. Workshop on Constructive Method (SPIN) (-46%)
2. Workshop on Declarative Method (PVS) (-46%)
3. Formal Methods in the Software Life Cycle (-41%)
4. Seminars with Industry (FM) (-27%)
5. Modeling Performance of Computer Systems (-26%)

Xootic

Just like last time, the main reason to be a member of Xootic is to stay in touch with other Xootic members. To stay informed about the TU/e and/or OOTI is the second reason. Lectures are the most appreciated Xootic activity. Suggestions for possible topics are about emerging technologies, such as .Net, C#, XML and Embedded Linux. Xootic should organise lectures more often, or organise ”X4X” lectures (Xootic-4-Xootic, like CMG’s ”Pro-4-Pro”).

Suggestions for activities are:

- Company visits/excursions (e.g.: nuclear plant, logistics centre, mobile phone centre)
- Trips (one weekend/one week to Italy, Spain, . . .)
- Short courses
- Golf clinic
- ’Wadlopen’
- Visit museums/theatre
- Regular dinner/drinks to keep in touch
- A (yearly) meeting for the ”older” generations
- Panel discussions/discussion groups

Other suggestions are:

- Easy access to OOTI/Xootic publications

Generation	≤ 60	≤ 70	≤ 80	≤ 90	≤ 100	≤ 110	≤ 120	≤ 150	> 150
1988 - Dec 1991					5	3	1	2	4
Sep 1992 - Jan 1994				3	5	8			3
Sep 1994 - Mar 1996	1		4	12	6				
Sep 1996 - Apr 1998		4	6	3	1	1			

Table 4: Salary distribution in HFL 1000 (absolute numbers of ex-OOTIs)

- Organise activities on a more central location in the Netherlands

Conclusion

The results of this survey are very valuable for OOTI and Xootic. It allows them to measure the quality of the program, steer the program and verify whether changes to the curriculum have the desired effect. The results can also be used to identify trends and interests of Xootic members and to take advantage of this information. This report only gives a summary of the survey results. More detailed information has been given to the OOTI and Xootic boards. The survey committee received some questions to correlate e.g. function with salary and signs of RSI and function. However, due to the small number of Xootics, these correlations do not present any reliable or significant information. For instance, one person with a managerial function, indicating to have signs of RSI very often, could lead to the 'conclusion' that 10% of people with managerial functions frequently have signs of RSI.

The survey committee also received some recommendations:

- The questions about the OOTI program were difficult to answer by ex-OOTIs who did not follow the courses, or could not remember the content of a course by name only (the course ID did not help much!)
- Use the web to do the survey

We would like to pass on these recommendations to the Xootic Survey 2002 Committee.

We would like to thank all Xootic members who returned the questionnaire for their co-operation. Without their effort, we could not have presented these results! Also, we would like to thank the Xootic Survey 1998 Committee for their support and useful input. One word of special thanks goes to Lettie Werkman, who helped us to improve the quality of the English language of the survey.

The Xootic Survey 2000 Committee: Rian Wouters, Dietwig Lowet and Gertjan Schouten.

Advertorial: Philips Nederland

Page 10 (should be even)

Perl

Ed Knapen

"Practical Extraction and Report Language or Pathologically Eclectic Rubbish Lister? Perl has devoted fans and fierce enemies. This paper describes Perl's post modern philosophy, shows some of its exotic operators and explains the link between Perl, Apostle Paul and perlmonks.com."

What is Perl?

Perl is a high-level scripting and programming language originally created by Larry Wall. It derives from the C programming language and to a lesser extent from sed, awk, Unix shells, and at least a dozen other tools and languages. Perl provides few but very powerful sets of data types (numbers, strings, and references) and structures (hashes and lists). Hashes (associative arrays) use strings as indices. Lists are numerically indexable and can also act as stacks, queues, or even double-ended queues. Perl emphasizes support for common application-oriented tasks: important features include built-in regular expressions, "text munging", file I/O, and report generation.

Here are two command line Perl scripts that should give you an impression:

```
ls | perl -pe
 '$i=$.;s//\e[3@{[$i++%7+1]};1m/g;
 END{print "\033[0m"}'

perl -pe "@ph=map {ucfirst(lc)}
 split(/[\s.,-]+/);
 print qq(@ph)" foo.txt
```

Officially, Perl is an acronym for *Practical Extraction and Report Language*, but the alternative *Pathologically Eclectic Rubbish Lister* is often used as well. Its roots are in UNIX but today you will find Perl on a wide range of computing platforms, including Mac, Windows and EPOC.

Not surprising given its origins, Perl is almost the perfect tool for system administrators: it allows the easy manipulation of files and process information, and easy automation of all kinds of tasks.

But Perl's process, file, and text manipulation facilities make it also particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, networking, and world wide web programming. Besides system administrators, these strengths make it especially popular with CGI script authors (most CGI programs are written in Perl), but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should use it as well.

Who is Larry Wall?

What Linus is to Linux, Larry is to Perl. Son of a pastor in Los Angeles, Larry Wall started off as a programmer and system administrator with a rich heritage of ideas and skills. Among these was the notion that everybody can change the world. He majored in Natural and Artificial Languages and attended grad school in linguistics. After the advent of Perl and the Perl book, which became a best seller, came royalties and a position with the publisher, O'Reilly & Associates. Today, O'Reilly pays Larry to do whatever he likes, as long as it helps Perl. And he generally eats breakfast at lunchtime.

In the beginning

On October 18, 1987, Perl 1.0 was posted to the Usenet group comp.sources. In Larry's own words:

"The beginnings of Perl were directly inspired by running into a problem I couldn't solve with the tools I had. Or rather, that I couldn't easily solve. As the Apostle Paul so succinctly put it, "All things are possible, but not all things are expedient." I could have solved my problem with awk and shell eventually, but I possess a fortuitous surplus of the three chief virtues of a programmer: Laziness, Impatience and Hubris. I was too lazy to do it in awk because it would have been hard to get awk to jump through the hoops I was wanting it to jump through. I was too impatient to wait for awk to finish because it was so slow. And finally, I had the hubris to think I could do better."

Perl was created and has been evolving by combining all cool features from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, Python, etc. Or, turning it around, by leaving out all the unwanted features of all these languages.

As for the name, Larry wanted a short name with positive connotations, looked at every three and four-letter word in the dictionary and rejected them all. Eventually he came up with the name "pearl", with the gloss *Practical Extraction and Report Language*. The "a" was dropped because of the existence of some obscure graphics language with the same name.

Post modern

Perl is unique in its aim to be post modern, as opposed to being based on modernism. Post modern is, according to Larry, what the American culture has become, not just in music and literature, but also in fashion, architecture and in overall multi cultural awareness.

To Larry, modernism was based on a kind of arrogance that elevated originality above all else, and led designers to believe that if they thought of something cool, it must be considered universally cool. That is, if something is worth doing, it is worth driving into the ground to the exclusion of all other approaches. Look at the use of parentheses in Lisp or the use of white space as syntax in Python. Or at the mandatory use of objects in many languages, including Java.

In contrast, post modernism allows for cultural and personal context in the interpretation of any work of art. It's the origin of the Perl slogan: "There's More Than One Way To Do It!" The reason Perl gives you more than one way to do anything is a belief that computer programmers want to be creative, and they may have many different reasons for wanting to write code a particular way. What you choose to optimize for is your concern, not Perl's. Perl supplies the paint (be it strings, associative arrays or objects), but the programmer paints the picture.

A second Perl philosophy is its aim for "No Limits". Maximum string or array lengths or similar boundaries are not or hard to find. Usually the only limit is the amount of free memory in your computer: the whole Linux kernel can be read into one (binary) string, patched and written back again using Perl.

Exotica

Perl contains over 50 special variables, most of them a combination of the $-sign (indicating a scalar variable) and a single character, for often used information. So is $_ the default input and pattern-searching space, while $. is the current input line number, $> the effective uid of the process, $+ the last bracket matched by the last search pattern, etc.

In addition, there are over 50 operators, like the usual +, ++, +=, etc., but also more unusual ones, like <> to read from a filehandle, <=> for numerical comparison (returning -1, 0, or 1) and =~ for search, substitute or translate.

Example:

```
open (SRC, $_[0]) ||
```

```
   die "Can't find source file";
while (<SRC>) {
   # match 'type' keyword
   if ( /type\s*=\s*"(\w*)"/ )
      { print $1."\n"; }
}
close SRC;
```

Many syntactic elements can be omitted, like parentheses around function arguments. The following two statements are equally valid:

```
print "Hello world";
print("Hello world");
```

Even semantic elements can sometimes be omitted. To read input from a file,

```
while ($_ = <STDIN>) { print $_; }
```

can be abbreviated to

```
while (<>) { print }
```

Perl allows execution of statements to depend on modifiers (if, unless, while, until). The following statements are all equivalent:

```
if ($energy < 0) { $nLives--; }
$nLives-- if ($energy < 0);
$nLives-- unless ($energy >= 0);
unless ($energy >= 0) { $nLives--; }
($energy < 0) && $nLives--;
($energy >= 0) || $nLives--;
```

In Perl, you can use the form that fits best with your ideas about what highlights the most important part of the statement.

As many other string based scripting languages, Perl interprets variables either as numbers or as strings depending on the context. A similar context dependency holds for scalars and arrays. If, for example, an array is assigned to a scalar variable, the length of the array will be assigned.

Passing arguments to a subroutine can only be done by resorting to list-context functions to retrieve the values, like:

```
do processFile($fileName);

sub processFile {
   print "Reading " . $_[0] ." file\n";
   open (SRC, $_[0]) ||
      die "Can't find source file";
   ...
}
```

Applications

Another Perl anecdote from Larry:

"A couple of years ago, I ran into someone at a trade show who was representing the National Security Agency. He mentioned to someone else in passing that he'd written a filter program in Perl, so without telling him who I was, I asked him if I could tell people that the NSA uses Perl. His response was, "Doesn't everyone?" So now I don't tell people the NSA uses Perl. I merely tell people the NSA thinks everyone uses Perl. They should know, after all."

Perl is used on Wall Street, in CGI scripts, in the robots and spiders that navigate the Web and build much of the various on-line databases. If you've ever been spammed, your e-mail address was almost certainly gleaned from the Net using a Perl script. The spam itself was likely sent via a Perl script.

Personally, my first Perl script was a 15-liner called "bgr", which changed the background picture on my OOTI machine every five minutes or so.

There are 800 or so reusable extension modules in the Comprehensive Perl Archive Network (CPAN). Glancing through those modules will give the impression that Perl has interfaces to almost everything in the world.

Comparison

Perl can be compared with other scripting languages like Tcl, Javascript and Python. Of these three, Tcl is the closest relation. Compared to Perl, Tcl's syntax is clean and simple, consisting of only a few building blocks. This makes it easier to teach non-

programmers Tcl. On the other hand, Perl gives you more power of expression. While you can certainly write awful and unreadable Perl programs, Perl's syntax and vocabulary also allow programmers to express exactly what they think, without having to resort to unnecessary constructions.

One advantage of Tcl over Perl is the availability of the graphical toolkit (Tk). This extension of Tcl is so tightly integrated that Tcl is normally referred to as Tcl/Tk. With Tk, three or four lines of code is all it takes to create a window with a clickable button or an editable input field. Since Tcl/Tk is also an interpreted language, you can play around with fonts, colours and dimensions until your interface is just right, without the need for recompilation. It is possible to use graphical extensions in combination with Perl (even the combination Perl/Tk is possible), but these are more awkward to use than the integrated Tcl/Tk pair.

Both Perl and Tcl are implemented in C and can be embedded into your own application code, to extend it with the power to interpret scripts.

Objects were only introduced in version 5 of Perl, which made a seamless integration impossible. If you are an object wizard who wants to write system administration scripts using objects, then maybe a language like Python is a better option for you.

Perl has the largest user base of all scripting languages. For whatever you need, chances are there is a package at CPAN available that does it. This is especially true in the field of CGI scripts. So if you need a quick start for a script that requires database connectivity or XML parsing, then Perl is a good choice.

Future

Last summer, Larry Wall announced the start of the development of Perl 6. In contrast to the first five versions, which followed an evolutionary development, a more organised approached with community input has been set up. If you have a desire to help in the crusade to make Perl a better place then peruse the Perl 6 developers page and get involved. The first alpha is expected by Summer 2001.

Links

For more information on Perl, try the following links:

www.perl.com
www.cpan.org
www.perldoc.com
www.perlmonks.com
www.perl.org/perl6

Biography Born in Elsloo (Limburg), The Netherlands, in 1970, Ed Knapen graduated in computing science from the Eindhoven University of Technology, The Netherlands. In 1995, he graduated in the postgraduate programme on software technology at the Stan Ackermans Institute in Eindhoven. This programme was concluded with a project carried out at the National Aerospace Laboratory (NLR) in Amsterdam, The Netherlands. Since then he has been employed by NLR to work on research and development in the field of application of information and communication technology in airport operations, air transport and air traffic control. His Perl programs are in use at NLR, in European aerospace companies and institutes and by an international chess organisation.

Java in Embedded Systems

Menno Lindwer

For several reasons, Java is not the most obvious language for embedded systems. It requires much more memory than most other languages and even with JIT compilers, it runs a lot slower. Therefore, systems running Java applications are more expensive than systems with the same functionality running natively. In many embedded systems industries (consumer electronics, networking, etc.), each additional cost is a market barrier. Besides that, Java is an inherently non-real-time language. However, some recent developments have turned Java into an obvious choice for those software tasks that do not require hard real-time operation, such as user interfaces. From the onset, Java was intended to reduce software development cost, software distribution cost, and lead time, which, in embedded systems, are rapidly becoming the dominant factors. Besides that, digital devices are increasingly required to interoperate with other devices and network servers. These new features require platform independent software, meaning that networked devices do not need to be aware of the internal architectures of their peers. Java comes with an extensive networking library and is compiled into a standardised platform independent distribution format, making ideal for such products. This leaves the reduction of Java execution cost in embedded systems as an interesting field for Research and and a challenge for Development...

Introduction

Java [1] is not the most obvious language for embedded systems. The reasons are manyfold. However, some recent developments have turned the tables.

Compared to conventional programming languages, such as C or C++, Java execution is expensive in terms of memory use, processor cycles, and power consumption. Until recently, the increased cost has proven to be a market barrier in embedded systems industries, such as consumer electronics and networking. To the users of many embedded systems, the increased functionality does not justify the increased cost. Besides that, the Java language is inherently 'real-time-unfriendly' [15]. It does not offer adequate constructs for specifying timing behaviour. The garbage collection and dynamic loading/linking features inhibit deterministic behaviour [15]. The programmer is shielded off from the underlying machine, giving him/her no handles to circumvent the problems.

However, recently system requirements (e.g. dynamic upgrade, networking, interoperability [6]) and business requirements (e.g. short time-to-market) have emerged that match quite perfectly with the mix of features offered by Java. This renewed interest has breathed new life into a number of optimisation efforts. It should be pointed out that none of the research topics are really specific to Java or invented specially for Java. But many have gained interest, funding, and momentum because of the possible application in Java. Some of those efforts, such as research into garbage collection algorithms goes back a long time [14]. Other developments, such as Just-In-Time (JIT) compilation [8], are quite recent. These optimisation ef-

forts in turn have brought the application of Java in cost-constrained embedded systems very near to commercial viability. In fact, at this moment several companies are introducing Java-enabled devices in one of the most cost-constrained industries: smartcards.

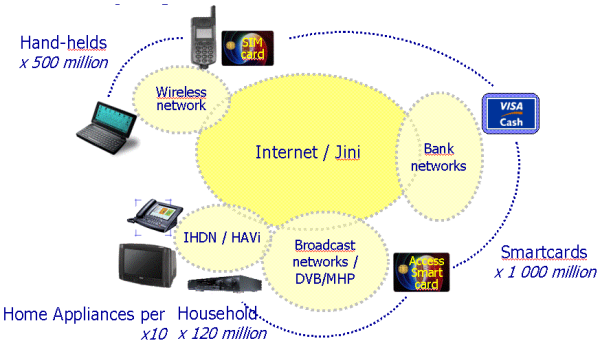


Figure 1: Markets for networked Java-enabled devices

Besides re-iterating the much-publicised software engingeering advantages of Java in an embedded context, this article aims to convey a deeper understanding of the performance and cost issues at play. I hope to show that Java's high execution and memory cost are not caused by singular features or failures in the Java system, but rather can be attributed to a multitude of deliberate considerations. This means that the law of 'retained misery' (Wet van behoud van ellende) almost always applies for attempts at optimisations. Therefore, good understanding of the issues is necessary, in order to prevent system designers from choosing solutions that exactly do not quite solve the problem or that are far more expensive than necessary.

This article is organised as follows: The next section discusses the business reasons, as related to the software development productivity gap, for applying Java. The second section investigates the impact of Java's feature set on execution cost. The third section takes us to the beef of the matter, namely the technical solutions for decreasing Java's execution cost. As dessert, the fourth section describes some hardware approaches for accelerating Java execution. The fifth section is the proverbial CFA (Conclusions, Future work, and Acknowledgements).

How is Java supposed to help increase software development productivity?

Java is an object oriented language [1]. Typical constructs from object oriented languages, such as inheritance, are usually considered beneficial for software development productivity. However, this section is concerned with the features that are more specific to Java. Most of what is discussed below is not really new. But Java is the first widely used language that wraps them all in a nice, well-marketed package.

The following aspects of Java were included to improve software development productivity:

- **Language simplicity:** The language is easy to learn, i.e. the syntax is kept very close to that of C (C++). However, many of the constructs for which C++ is regarded as complex, have been left out. Apparantly, this has not made the language less useful. A good example is the use of interfaces, instead of multiple inheritance (although some people question the use of either). Giving classes multiple interfaces is almost as powerful as multiple inheritance. However, it avoids problems such as classes inheriting the same class more than once.

  Because Java is easy to learn, it quickly gained a large developer base. This means that it will be more easy to find qualified software development personnel. Besides that, Java's simplicity allows other professionals than software developers to understand Java code. Therefore, other stakeholders can more easily participate in quality assurance of software projects.
- **Strong typing:** The Java language is strongly typed. This means that the compiler can statically check many commonly made mistakes, such as passing wrongly typed arguments, inadvertently losing or adding sign extensions at assingments, pointer arithmetic problems, etc.

  In fact, some researchers claim that, because of Java's strong typing, compilers have more knowledge about the way the code will execute and can therefore apply more aggressive optimisations. This means that, in theory, Java code could run faster than C code...
- **Exception handling:** The software designer

can define the application level at which exceptions should be caught. The language offers constructs such that, at the levels below that one, developers can treat them transparantly (i.e. just pass them on). Of course the language also offers constructs for easy handling of exceptions at the level defined for that purpose. As an extra (and obvious) safety precaution, exceptions that do slip through that level will eventually be caught by the run-time environment.

- **Array boundary checking:** The code is guaranteed not to violate array boundaries. Software developers should still check for array boundaries. But if this fails, at least the state of the system does not get corrupted. The exception mechanism offers a standardised construction for handling those situations. Also, array boundaries are part of the language and can therefore be taken into account when writing loops. In fact, the exception catching mechanism can be legally used to end array handling loops (which is not to say that this is good programming practise :-)!
- **Automatic memory management:** Java does not have explicit memory allocation, nor can the programmer explicitly return memory. Memory is implicitly allocated during creation of objects. The language assumes that the operating environment contains a garbage collector that should appropriately reclaim memory. Many languages (including C) actually do not have constructs for memory allocation and de-allocation; they are part of the library structure. In Java and C++, implicit memory allocation is part of the language. In Java, garbage collection is part of the language, in the sense that an explicit construct has been (purposely) omitted.
- **Platform independence:** This is achieved by compiling Java to an intermediary language (Java virtual machine language or Java bytecode, JBC). It is not a language feature. In fact, Java can very well be compiled directly into native code of any processor [5]. In principle, it is not possible to compile languages such as C and C++ to JBC, a.o. because Java does not require JBC to offer direct memory manipulation. Java Virtual Machine language interpreters (JVMs) are available for most (embedded) platforms. Combined with the next item, platform independence can result in enormous savings of development cost, because one does not need to maintain different software versions for different platforms.
- **Rich standardised set of APIs:** This set, including implementations (mostly in Java) was released together with the language. Software developers can safely assume implementations of these libraries to be available on any platform that runs the targeted flavour of Java. This means that developers can concentrate on solving the real problems. They do not have to spend effort studying APIs for many different platforms or implementing code for such basic operations, as set handling, sorting, hash tables, graphics primitives, etc.

The combination of these features is rumoured to result in a productivity increase per developer of a factor of 2.

Impact of Java's feature set on system cost

As far as the production cost of embedded devices is concerned, features such as language simplicity, strong typing, and exception handling come more or less for free. The other features come at a high cost.

Cost of performance penalty

The language specifies that every array access has to be checked against array boundaries. During an experiment on a real-life software system (a 15 KLoC simulation module, written in C), array bounds checking code could be switched off, increasing performance by 10%. Take into account that this module was only part of the complete simulator, which also contained a simulation kernel and several other simulation modules, together emulating a silicon system. Software, instrumented with Purify (a memory consistency checking tool), runs factors slower than production code. Several projects [20] report overhead, caused by garbage collection, to be in the range of 5% to 35%. Interpretation overhead (when using a regular JBC interpreter, not a JIT compiler) is usually reported to account for about a factor 5 to 10. Together, these features result in a slow-down of a factor 20 to 40

over the same functionality, implemented in C. This performance penalty translates into higher system cost. The processing elements inside an embedded system are usually dimensioned very carefully to exactly match the requirements of the software. Every unnecessary resource causes the eventual product to be more expensive and thereby lose market share. An exact factor for the increased system cost is difficult to give. It is usually not necessary to actually dimension the system 40 times larger than otherwise would be required. On the other hand, just scaling up the clock speed of the system is not enough. In order for a processor to actually benefit from higher clock speed, it should also have bigger caches, wider memory lanes, faster on-board buses, more complex board designs, etc. A common way of increasing the Java performance is the application of a Just In Time (JIT) compiler, which reduces interpretation overhead (associated with executing JBC, Java's intermediary virtual machine language). However, even if a JIT compiler were to remove all interpretation overhead, Java is still about a factor 4 slower than native code (because of the other performance costs, such as array bounds checking and garbage collection).

Cost of increased memory requirements

The increased memory requirements are due to four factors:

- The JVM is a relatively large piece of software. The smallest full implementations have footprints of about 100KB. Because of optimisations, fancier threading mechanisms, and fancier user interface layers, this can increase to about 500KB. Since, in many cases, the JVM will be part of the firmware of a system, it will reside in ROM. ROM is much cheaper than RAM. Therefore, one would be tempted to discard this cost. However, RAM is faster than ROM, so that many embedded systems copy firmware to RAM, upon startup...
- Next to the JVM, a full Java system requires about 9MB of Java run-time libraries. For several reasons, it makes sense to place this code in rewritable memory. In a networked environment, this code definitely is eligible for updates. Besides that, for performance reasons, most JVMs modify the instructions as they execute them (turning dynamically linked code into semi-statically linked code). This requires the libraries to be placed in RAM.
- Java memory management is relatively expensive (in terms of memory utilisation). This is partly due to programming practises, partly it is inherent to the use of a garbage collector. Current programming practise results in the constant generation of many short-lived objects. For example, function results, as used in expressions often are objects, even though they could just as well be scalar types (integers, booleans). Returning an object causes that object to be created on the heap. However, immediately after evaluation of the surrounding expression, the returned function results become redundant.
- The garbage collector requires that the system contains more heap memory than strictly required by the application. Otherwise, the garbage collector would have to be activated whenever an object becomes redundant. Together, memory allocation and de-allocation require about 2MB of RAM, in order to run meaningful user-interface oriented applications.

  Specially mobile applications, provided by NTT DoCoMo's iMode (a Japanese mobile phone operator), show that careful design of Java software can result in useful applications that require only a few 10s of KB for dynamic memory allocations.

All-in-all, the minimum requirement for a full Java system is about 10MB of ROM and 2 MB of RAM. This comes on top of storage for the actual Java application code (which is assumed to be about the same as for the same application in native code[1]) and the requirements of the underlying operating system (which is still required when running Java).

It should be noted that most embedded systems will not contain the full set of Java libraries. Part of the confusion around Java technology stems from the plethora of application domain specific subsets and extensions to the full Java API set. Experiments have shown that the full set can be brought back to about 500KB for mobile applications, by removing user interface and character conversion rou-

[1] On the one hand, JBC is about a factor 2 more compact than RISC code. On the other hand, JBC is packaged in Java class files, which contain a lot more data than just the JBC. Only some of that data gets discarded during loading.

tines. When disregarding the performance penalty of ROM, and when using specially designed applications, the minimal footprint for a Java system ends up at about 1MB ROM and 100 KB RAM. Again, these costs come on top of the requirements for the actual (Java) applications and OS.

It is up to the system designer to choose the appropriate API set and live with the consequence of not being able to support all Java applications.

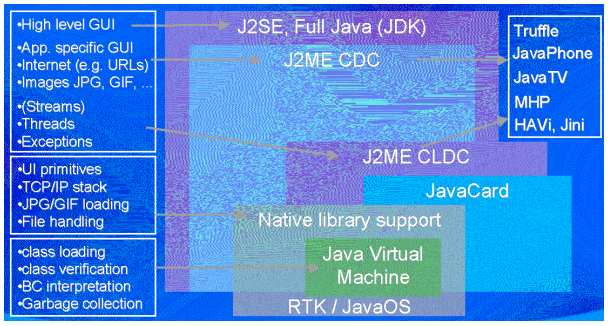


Figure 2: The complexity of the Java technology chart stems from the fact that it consists of many API sets, most of which are not precisely subsets of each other. The left column lists the functionality of the APIs. The rightmost box gives a number of product dependent API extensions

Cost of increased system complexity

These costs are difficult to quantify in a generic sense. But we can give an indication of the issues at play. What is meant here are the costs associated with having to design and maintain software and hardware components that are more complex than would be strictly required for native operation.

The simplest scenario is where efficient execution (i.e. interpreter performance) and graphics (user interface) are not required. There are few examples of such systems, because devices without user interfaces usually constitute high-volume, low cost markets. Anyway, in that scenario, the only engineering cost is associated with porting a bare-bones Java interpreter to the target system. An experienced software engineer spends about half a man-year on porting, testing, and verifying a software stack like Sun's KVM. Given frequent updates, both in Java interpreter software technology and hardware platforms, the same cost will probably recur for maintenance on a yearly basis.

A more complex and realistic scenario would be the higher-end hand-held and mobile devices, in wich Java execution is added for user interface purposes and simple applications. Because of the kinds of applications, it is not required to have high-performance execution. And because of the market positioning, it is feasible to incorporate extra processing power. In this scenario, assuming that the platform already provides some degree of graphics support, the software development cost is increased by another 2 man-years for porting and verifying the native parts of the Java user interface toolkit (e.g. Sun's Abstract Windowing Toolkit, AWT). The maintenance cost will remain at about half a man-year, annually.

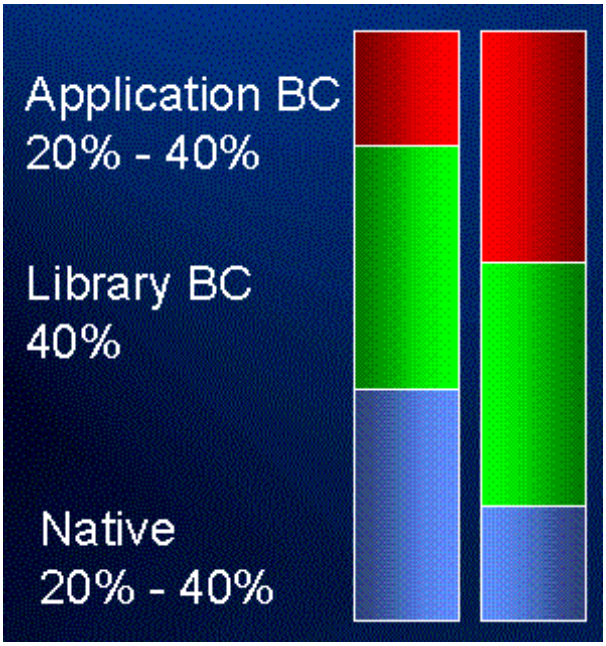


Figure 3: Applications spend 20% (large apps, right bar) to 40% (small apps, left bar) of execution time on native code. Consequently, 60% to 80% of time is spent on bytecode interpreting.

The next scenario are the medium to high-end consumer devices, such as set-top boxes. In the near future, they will adhere to standards such as Multimedia Home Platform (MHP [3]), Home Audio Video Interoperability (HAVi [6]), and Jini, which apply Java for complex tasks, combining system control and advanced user interface technology. Despite the market positioning (medium to high-end), the consumer price for such systems does not allow for the inclusion of PC-class hardware. In the first versions of these devices, the processor speed is limited to about 300MHz. Internal memory is in the range of 16MB to 32MB. Harddisks are not

yet part of the package. The heavy use of Java in advanced user interfaces requires an optimised Java interpreter, sophisticated graphics stack, and native multithreading support. Several companies deliver speed-optimised interpreters, often in combination with JIT compilers. Because of their complexity, these systems require significant up-front and running licensing fees. Therefore, a choice for any package requires an extensive selection process. Usually, this selection process involves experimental ports of several rival software stacks onto simulators of the projected hardware system (during those preliminary experiments, the actual hardware is often still in the design phase). This selection phase may already involve several man-years work...

This paragraph dealt with some of the issues, surrounding the complexity of adding Java suppport to several types of embedded systems. Even though the list of issues per scenario and the set of scenarios are not complete, I hope this paragraph gives an idea of what to expect.

What Technologies are used to decrease Java's execution cost?

Obviously, the choice of technologies depends on the actual costs of the bottlenecks, as discussed in previous sections. For example, it makes no sense to optimise thread synchronisation for small embedded devices that are not expected to perform much multi-threading. However, in most cases, it does make sense to write the main interpreter loop in assembly, since this is where most JVMs spend about 80% of their time [20].

When analyzing technologies, we can make several more or less orthogonal categories: hardware versus software, memory versus speed, and domain specific versus generic. Conveniently, this set of categories can be represented as a cube with more or less orthogonal sides, see Figure 4. For example, JIT compilers are generic software enhancements, which impact the speed of the interpreter, at the cost of increased memory utilisation.

In the following sections, we will categorize and discuss a number of common optimisations to Java execution mechanisms. What we will see is that, as is usually the case, most optimisations involve trade-offs, where an improvement on one axis of the cube means a degradation on another axis.

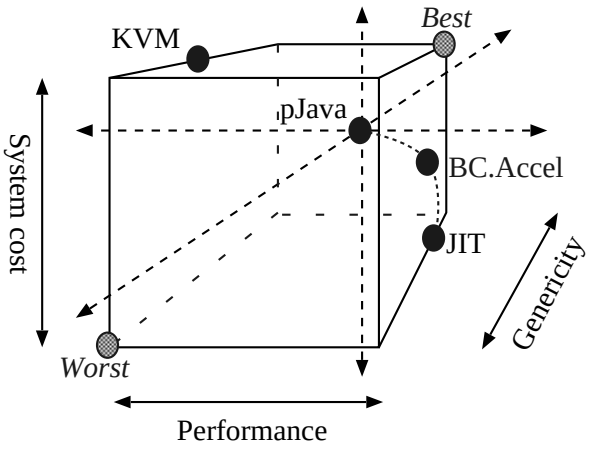


Figure 4: Cube of execution enhancing technologies, indicating some postitions, relative to Sun's standard interpreter (pJava).

JIT Compilers

JIT compilers [8] were already categorized as generic software solutions for increasing Java execution speed, at the cost of increased memory utilisation. It is therefore questionable whether they actuallly decrease execution cost. If memory is more expensive than processor silicon, this may not be the case.

Paradoxically, the pure JIT (Just In Time) compiler systems can also be called "Just Too Late", because they start compiling a (Java byte) code sequence at the exact moment the user/system (first) needs that particular function. Especially on embedded systems with relatively light processors, this initial call may take a long time. Also, this behaviour is particularly disruptive to real-time operation.

In order to prevent the Just Too Late behaviour and decrease memory cost of pure JIT compilers, profiling JIT compilers were introduced [HotSpot, 21]. Besides a compiler, such a system also contains a conventional interpreter-based execution mechanism. Initially, all code is executed by the interpreter. For every distinct code block (usually method), the frequency of its invocations is measured. When this frequency exceeds a certain threshold, the code block gets JIT-compiled. This approach generally decreases memory requirements, because no memory is waisted on the translation of blocks that are executed infrequently.

However, we do have to take into account that the JVM has grown larger, because of the extra interpreter and profiling software. It also remains to be seen whether this approach performs as well as a JIT-only solution, since initial interpretation runs and profiling efforts may decrease overall performance.

Subset interpreters

These are domain specific software optimisations for reducing memory utilisation, usually at the expense of performance.

JavaSoft's KVM [21] and JavaCard [21] are examples of interpreters that do not support the full set of Java bytecodes.

The same goes for JVMs and library implementations that support only a subset of the standard Java APIs. Usually, those subsets are restricted in terms of user interface capabilities. For example, the Truffle [21] user interface library can only handle one application window at any time. It is implemented almost fully in Java, thereby reducing the required native functionality to a minimum (basically just pixel drawing). However, because almost all functionality is implemented in Java and supplied as Java bytecodes, Truffle is also relatively slow.

Specialised processors

These are generic hardware solutions for accelerating bytecode execution. Depending on (non-Java) legacy code requirements, the inclusion of a general purpose processor might still be necessary. In that case, the solution will come at the cost of increased silicon area and increased system complexity, both in terms of hardware and software system design.

Examples of specialised processors are PicoJava [16], Moon (Vulcan ASIC), and Shboom (Patriot Sciences). These are all processors that run the complete Java bytecode set natively. Keep in mind that the Java Virtual Machine language represents a Complex Instruction Set Computer (CISC). In fact, some Java bytecodes are extremely complex, involving memory allocation, initialisation, string table searches, and/or bytecode loading. On a regular software interpreter, this requires thousands of cycles. Normally, CISCs contain microcode, splitting complex operations in sequences of more basic operations. Microcode can be seen as a kind of processor-internal 'software'. Specialised Java processors can implement most bytecodes using microcode. However, the really complex bytecodes can not be implemented using such an extremely low-level language. Therefore, in spite of the promise of generic Java programmability, heavy investments in software development environments for those processors do have to be made.

And even if C/C++ software development environments are available for those specialised Java processors, they usually still do not run all the required legacy software. For example, because the legacy software was programmed in assembly or requires the support of an operating system that is not available for the Java processor. This would mean that a general purpose CPU needs to be added to the hardware system. If the project can afford to develop its own ICs, the additional direct cost is limited to a few euros worth of silicon per product. However, if the project has to rely on off-the-shelf hardware, extra ICs and increased circuit board size have to be added to the bill of material and product form factor. In terms of system design, going from a single-CPU to a multiprocessor solution adds a whole new set of problems, such as communication protocols, cache coherency protocols, and resource access arbitration. This gets aggravated in the case of heterogeneous multiprocessor designs, consisting of different types of processors.

Of course, a specialised Java processor (even a heterogeneous multiprocessor, incorporating a Java processor) probably contains less silicon than a single general purpose processor, offering the same Java performance. However, the question is, can't we find a more optimal approach, especially regarding the system design issues?

Bytecode accelerator hardware

Like specialised processors, these are hardware solutions for accelerating bytecode execution [11, 13]. However, they assist a general purpose processor in executing Java. Therefore, the complete solution always consists of a processor and an accelerator. Since this processor is relieved of many of the Java execution tasks, it can be relatively small. Besides that, the accelerator module itself should be signifi-

cantly smaller than the Java processors in the aforementioned heterogeneous designs.

In its simplest form [2], the accelerator is actually a translator from Java bytecodes to CPU native instructions. It can be seen as an instruction-level JIT compiler, implemented in hardware. Because it is implemented in hardware, it can perform its tasks in parallel to the processor doing the execution of the generated code. Because the translation takes place at instruction level, the system requires very little storage for intermediate results (a matter of several bytes, rather than several megabytes for a software JIT compiler).

One instance of such an accelerator will be discussed in more detail in the next section.

Graphics accelerators

Measurements have shown that, for meaningful Java applications, 2D graphics processing takes 10% to 20% of all processing time [20]. The reason is that most Java applications are user-interface intensive. After optimising Java bytecode processing, the relative impact of this factor will increase to 20% to 50% of all processing time. This means that graphics acceleration only becomes an issue after bytecode acceleration.

Graphics acceleration is a domain specific optimisation. It only has use in environments that require media processing or have graphical user interfaces and large screens with some degree of color depth.

Obviously, adding a graphics accelerator means higher hardware costs.

Multi-level and hardware garbage collectors

As was mentioned before, garbage collection also accounts for a significant amount of performance loss. As with graphics, this is very application dependent. Garbage collection seems to be a good candidate for acceleration through hardware. Some attempts have been made, including in the author's own projects [12]. In fact, it is not very difficult to implement certain garbage collection algorithms in hardware [14].

However, garbage collection algorithms themselves require substantial and variable amounts of memory. This can only be efficiently achieved by integrating the garbage collection logic with the memory devices. But the memory device business in very a cost-sensitive commodity market. Specially designed garbage collected memory chips can not be produced in sufficient numbers to make them commercially viable.

Another approach to at least alleviate the garbage collection bottleneck is to implement several types of software algorithms. Some algorithms are particularly good at quickly finding a large number of short-lived objects. Other algorithms are more thorough, but also more time consuming. Therefore, the heap is divided in a space for short-lived objects and a space for older objects. The former ones are scanned quickly. Objects that have survived a number of those scans are moved to the latter space, which is scanned with the thorough procedure. The performance benefit results from the expensive procedure having to scan only part of the heap.

Optimised thread synchronisation

Java is a multithreaded language, heavily oriented towards re-use. This means that designers of Java classes alsways have to take into account that multiple threads may wish to concurrently access the internal data structures of those classes. Every object that may be accessed concurrently has to be protected against multiple threads interfering with each other's changes. Therefore, Java objects are synchronised very conservatively. The synchronisation operations involve threads performing operating system calls for claiming exclusive access, getting blocked as long as the claim can not be rewarded, and relinquising the claims when the operations have finished. These operating system calls are very expensive. A lot of time can be saved if one can utilise the fact that actual interference is very rare.

A Hardware approach to accelerating Java execution

At Philips Research, we've been working since the end of 1996 on hardware for Java acceleration in embedded systems. The work started from the following constraints:

- chip area increase should be minimal (e.g. much less than size of low-end 32-bit RISC CPUs),

- memory utilisation should not increase, compared to software interpreter,
- solution should be compatible with modern RISC CPUs (since general purpose CPUs remain necessary),
- solution should be modular (i.e. have minimal impact on other components in an embedded system), in order to facilitate re-use,
- performance increase should be at least a factor 5 over a regular software interpreter.

We found a solution in the form of a translator module, which assists general purpose CPUs in executing Java bytecodes. We called the module Virtual Machine Translator (VMI). Later, we found [2], which gives a good description of many of the concepts. VMI is very small. Essentially, it consists of tables that direct the translation. These tables can be implemented in a very compact way. VMI needs very little computational logic, since most computations take place on the general purpose CPU.

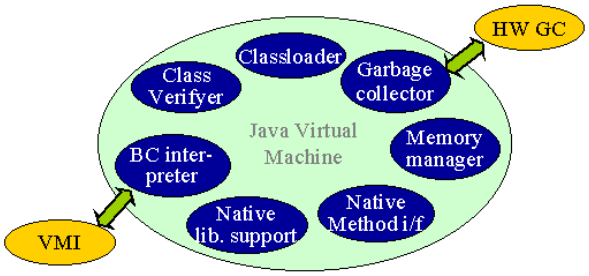


Figure 5: From a software point of view, the bytecode interpreter module is simply replaced by hardware (as the garbage collector module might be)

Since part of the Java interpretation task is now implemented in hardware, the memory utilisation actually decreases slightly (we need less code to implement the Java interpretation software). Since the actual operations take place on the general purpose CPU (remember that VMI is only a translator), there are no problems with data coherency between the two processing elements. Contrary to most other accelerators, VMI has been developed completely separately from the CPU. CPU and VMI only communicate through the on-chip system bus. Currently, most integrated microcontroller devices contain standardised on-chip buses. Therefore, building VMI for a specific on-chip system bus, means it is compatible with all CPUs that can be attached to that bus. The fact that VMI communicates only through a standardised bus also means no other parts of the hardware system need to be modified.

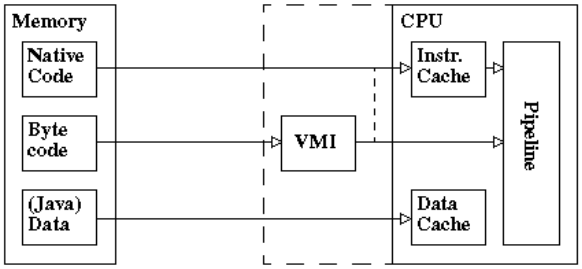


Figure 6: From an abstract hardware point of view, VMi is placed between the memory and the CPU pipeline, feeding the pipeline with translated bytecodes

After having indicated how the solution is intended to solve the problem, while keeping within the constraints, it is now time for some more technical detail.

Most computer systems contain at least a CPU (Central Processing Unit) and a memory. The CPU can be seen as a robot, which is able to execute sequences of instructions. For example, a car construction robot repeatedly executes instructions that tell it to move, pick up components, attach components, measure parts of the construction, etc. In order to assemble a complete car, such a robot executes thousands of those instructions. In the same way, CPUs execute billions of instructions for a simple task, such as drawing an image on a screen or printing a document. The CPU reads those instructions from the aforementioned memory. Thus we find the instructions for the Java applications in the memory and require the CPU to fetch and subsequently execute them. However, general purpose CPUs do not understand the Java instructions (also called 'bytecodes'). This is where the Java Virtual Machine software comes in. It translates the bytecodes into instructions that the CPU does understand. This means that next to the functionality of the bytecodes themselves, the CPU needs to spend time on the interpretation task. A very simple interpreter for some of the bytecodes could be programmed as follows:

```
1.unsigned interpreter( char *pc ) {
2.  /* 'pc' points at bytecodes */
3.  unsigned sp[STACK_SIZE];
4.  /* 'sp' compute result stack */
5.  while(TRUE) {
6.    switch( *(pc++) ) {
```

```
7.      case push_const :
8.        *(sp++) = *(pc++);
9.        break;
10.      case pop :
11.        sp--;
12.        break;
13.      case add :
14.        *(sp-2)=*(sp-2)+*(sp-1);
15.        sp--;
16.        break;
17.     case ret :
18.        return *(sp-1);
19.        break;
20.    }
21.  }
22.}
```

The above code does not need to check stack under/overflow or code overrun conditions, because in Java this is done statically.

Notice that the above instructions (push_const, pop, add, ret) are about as powerful as regular CPU instructions. However, the while-switch-case-break construction (lines 5, 6, 7, 9, etc., in the code above) usually requires between 10 and 40 CPU instructions per iteration. The actual functionality of the bytecodes (lines 8, 11, 14, 15, and 18 in the code above) requires between 5 and 10 CPU instructions. The reason is that the stack pointer-relative addressing introduces an extra indirection and because the stack pointer itself needs to be updated. This means that a CPU needs to execute 15 to 50 instructions for operations for which it would normally require 1 or 2 instructions. This means a 7x to 50x interpretation and execution overhead per bytecode.

Going back to the accelerator concepts:

In order to reduce the interpretation overhead, the program counter is moved from the CPU into the accelerator. The accelerator now reads the bytecodes from the memory and determines the location in its translation tables of the corresponding sequence of CPU instructions. It performs this task within the time the CPU needs to execute the previous translation. Thereby, the while-switch-case-break bottleneck is completely removed.

In order to reduce the time needed for the actual functionality (remember that push, pop, add, and ret require 5 to 10 CPU instructions), the stack pointer is also moved from the CPU into the translator. Now, instead of just providing the corresponding sequence of translated instructions, including stack pointer indirections, the translator simplifies the translation by substituting the stack values in the instruction sequences (inspired by [4]) and doing the stack pointer updates internally. The resulting translation sequences have an average length of about 2 CPU instructions. All-in-all, the translator provides a speed-up on the above bytecodes of at least a factor 15.

Conclusions, Future Work, and Acknowledgements

Java is becoming an important language for embedded systems programming. However, before Java-based products can become a success, the cost of the Java execution mechansism has to be reduced.

Most companies providing Java execution mechanisms advertise their solutions citing a single benchmark (e.g. [17]). In this article, I hope to have made it clear that performance is not the only factor at stake and that JVMs are such complex systems that a single-point measurement of performance can not give an accurate indication of relative qualities.

The interest in incorporating Java in embedded systems is still increasing. Despite Moore's law (prescribing that compute power will steadily increase), there is a continuous need to taylor Java implementations to the strict requirements of embedded systems. Java acceleration technologies seem to offer interesting advantages, but their commercial viability still needs to be proven. On the short term (during 2001), JIT compilers will find their way into systems with little real-time and memory restrictions. On the somewhat longer term (before 2003), we will see bytecode accelerators opening up extremely constrained devices to the Java language. 2D graphics accelerators are already used in embedded systems with heavy user interfaces. The sophistication of garbage collection systems is constantly increasing, but much work remains to be done here. It is questionable whether garbage collection hardware will ever become viable.

I would like to thank the members of the Java Hardware Accelerator project at Philips Research for their enthousiasm, in particular Otto Steinbusch (currently at Philips Semiconductors), Narcisse Duarte (currently at Canal+), and Selim Ben-Yedder. I've also had many valuable discussions

with Pieter Kunst, Nick Thorne, Harald van Woerkom, and Paul Stravers.

References

[1] K. Arnold, J. Gosling, D. Holmes, *The Java Language Specification*, Addison-Wesley 2000, ISBN 0-201-70433-1

[2] E.H. Debaere, J.M. van Campenhout, *Interpretation and Instruction Path Coprocessing*, The MIT Press, 1990, Cambridge MA, USA

[3] Digital Video Broadcast Multimedia Home Platform, http://www.mhp.org/html_index.html

[4] M.A. Ertl, *Implementation of Stack-Based Languages on Register Machines*, PhD thesis Technische Universitaet Wien, Vienna 1996

[5] The Free Software Foundation, *The GNU Compiler for the Java Programming Language*, http://www.gnu.org/software/gcc/java

[6] HAVi, http://www.havi.org

[7] J. Hoogerbrugge, L. Augusteijn, *Pipelined Java Virtual Machine Interpreters*, 9th International Conference on Compiler Construction, April 2000, Berlin, Germany

[8] A. Krall, R. Grafl, *CACAO - A 64 bit JavaVM Just-in-Time Compiler*, Institut fuer Computersprachen, Technische Universitaet Wien, Vienna, 1998

[9] M. Levy, *Java to Go: Part 1; Accelerators Process Byte Codes for Portable and Embedded Applications*, Cahners Microprocessor Report, February 2001

[10] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1996-09

[11] M. Lindwer, *Versatile Java Acceleration Hardware*, 2001, to appear...

[12] X. Miet, *Hardware for (Java) garbage collection*, ENST, Paris, France, October 2000

[13] Nazomi, *Nazomi Communications; High Performance Java Technology for Mobile Wireless and Internet Appliances*, http://www.nazomi.com

[14] K. Nilsen, *Progress in Hardware-Assisted Real-Time Garbage Collection*, Iowa State University, 1995, http://www.newmonics.com/dat/iwmm_95.pdf

[15] K. Nilsen, *Issues in the Design and Implementation of Real-Time Java*, NewMonics, Inc., April 1996, http://www.newmonics.com/dat/rtji.pdf

[16] J.M. O'Connor, M. Tremblay, *PicoJava-I: The Java Virtual Machine in Hardware*, pages 45-57, IEEE Micro, 1997-03/04

[17] Pendragon Software, *CaffeineMark 3*, http://www.pendragon-software.com/pendragon/cm3/info.html

[18] Philips Research, *Mobile phones, set-top boxes, ten times faster with new Philips accelerator for Java*, January 2001, http://www.research.philips.com/press-media/010101.html

[19] Philips Semiconductors, *Java hardware accelerator for embedded platforms, Philips Semiconductors World News*, November 2000, http://www.semiconductors.philips.com/publications/content/file_680.html

[20] O.L. Steinbusch, *Designing Hardware to Interpret Virtual Machine Instructions; Concept and partial implementation for Java Byte Code*, Master's thesis, Eindhoven University of Technology, February 1998, TUE-ID363006

[21] Sun Microelectronics, *JavaSoft; The Source for Java Technology*, http://www.javasoft.com

Biography. Menno Lindwer is a Senior Scientist at Philips Research in Eindhoven (The Netherlands). He has been involved in hardware design (methodology) since 1991, graphics acceleration since 1995, and Java acceleration since 1996. Menno holds a Master's Degree in computing science from Twente University of Technology (1991) and a post master's degree in software technology from Eindhoven University of Technology (1993).

Other interests include object oriented design, simulator technology, and system-on-silicon architecture. Menno joined Philips Research in 1995. Currently, he is in charge of the Platform Independent Processing and Java Hardware Acceleration projects at Philips Research in Limeil-Brevannes (France) and Eindhoven (The Netherlands). Previous work experience includes a.o. artificial intelligence systems, research in delay insensitive asynchronous circuits, and performance analysis of 3D graphics accelerators.

Python

Victor Bos

"And now for something completely different..." Python is a scripting language with clear syntax and semantics, support for object orientation, and an extensive standard library. In contrast with many other scripting languages Python code is readable and, therefore, reusable. This makes Pyhton a useful tool for software development, since it can be used to implement prototypes as well as production versions of applications.

Introduction

Python is a *scripting* or *extension* language similar to Perl [12], Tcl/Tk [8]. In his foreword to *Programming Python* [6] Python's creator Guido van Rossum wrote (See [11]): *"I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers."* The ABC language, [5], has never become popular, which is partly caused by its peculiar syntax, but it was well designed. In addition to ABC, Python was influenced by Modula-3, an object oriented descendant of Pascal meant for system programming, see[7]. This resulted in a scripting language with clear syntax (which is not common for scripting languages) and powerful language constructs.

Furthermore, Python comes with an extensive standard library that provides the programmer access to a huge set of routines. Therefore, a Python programmer usually does not have to spend much time to implementation details of standard routines like matching a regular expression on a string, or accessing operating system functionality to create processes, pipes, etc. Instead, a Python programmer can just look up the relevant Python *modules* in the standard library and use them to solve her/his problem.

The language

The syntax of Python is quite standard, as will be shown in examples throughout this article. However, there are some controversial aspects. *Indentation* of groups of statements is one of them. If a group starts on a new line, all its statements should be indented by the same number of columns. For example, a while loop is written as:

```
while i<n and f(i)<f(n):
    a[i] = f(i)
    i = i + 1
```

The statements `a[i] = f(i)` and `i = i + 1` form a group. Since indentation is used to indicate groups, no group delimiters like `{` and `}` or `begin`–`end` are needed. Programmers unfamiliar with Python might find this irritating, however, it is not a drawback. Experienced programmers (in no matter what language) have usually adopted their own style of indentation for groups of statements. Since Python does not prescribe the number of columns of indentation, these people can keep on using their own style in Python. Furthermore, the code does not get messed up with group delimiters.

Build-in data structures Python has the following data structures build-in: integers, floats, strings, tuples, lists, dictionaries, and functions. There are no booleans, which is a shortcoming not only of Python but of most scripting languages. Integers, floats, and strings are standard data structures which we will not discuss here. A tuple is an *immutable*

sequence of elements, that is, it is a sequence of which the elements cannot change once the tuple is created. Lists are mutable sequences of elements; elements can be added to and removed from lists. A very powerful build-in data structure is the *dictionary*. A dictionary is a look-up table or associative array containing key-value pairs. Hashing is used to look up a key in a dictionary which means dictionaries have fast access times. Finally, functions are first-class objects in Python. Therefore, Python programs can be a mix of functional and imperative programs. A *lambda-syntax*, known from many functional programming languages, is used to denote anonymous function. For example, the function that adds two elements could be written in Python as `lambda x, y: x + y`. Since this is a normal object, it can be assigned to variables, as will be shown later.

Universal object model Python has a *universal object model*, which means that every piece of data in a Python program is an object. As usual in object oriented programming languages, an object has attributes that define the state of the object and methods to allow other objects to perform operations on the object. For example, the following Python code defines a class `Point` of objects with an x and a y coordinate and a method `dist` to compute the distance between two objects.

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2) ** .5
```

In the definition of `dist`, the two argument `**` operator is used; `x ** y` raises `x` to the power `y`.

The example shows at least two syntactic peculiarities. First of all, Python has special syntax for special methods like the `__init__` method. The special syntax, which in my opinion is quite ugly, is an identifier that starts and ends with two underscores. The `__init__` method is special, since it is a constructor of the `Point` class and will be called whenever a point is created, for example, the point $(1, 2)$ is created by calling `Point(1,2)`. Note that the coordinate arguments of `__init__` have default values, `x=0` and `y=0`, so the point $(0, 0)$ could be created by calling `Point()`. Other special methods are used to overload operators and build-in functions. For example, the `__add__` method can be used to overload the `+` operator. By extending the `Point` class with the following definition of `__add__`, we can write `p1 + p2` in order to add the points `p1` and `p2`.

```
def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)
```

The other strange part of the examples above is the `self`-parameter of `__init__`, `dist`, and `__add__`. This parameter is a self-reference to the object on which the method is invoked. Whereas in most object-oriented languages there is usually no need to make the reference to an object itself explicit, in Python it is. Furthermore, the self-reference is always the first parameter of the method. By convention it is called `self`, but the programmer is free to choose another identifier.

So, in a constructor (`__init__`) `self` refers to the object that is created and in a normal method (`dist` or `__add__`) `self` refers to the object on which the method is invoked. In some programming languages, `this` is used instead of `self`, e.g., C++ [9] and Java [1].

Unlike many object-oriented programming languages, the set of attributes and the set of methods of an object are not constant during its lifetime. For example, the following code creates a `Point` object, changes its x-coordinate, and adds a color attribute.

```
p = Point()
p.x = p.x + 4
p.color = "yellow"
```

Programming styles Python supports three programming styles: procedural, object-oriented, and functional programming. Furthermore, these styles can be mixed arbitrarily. Of course, an unrestricted mix of these three styles will not improve readability and maintainability of the program and it is therefore wise to stick to one style as much as possible. However, programming styles are meant to ease programming and not to restrict the freedom of the programmer. Therefore, if in a given situation one particular style is not adequate, it should be possible to switch to another style. Python supports programming using multiple styles, whereas a

pure functional language or a pure object oriented language does not.

The following Python listing is an example showing the three programming styles. First we take the `Point` class again and extend it with the special method `__str__`. This method will be called if a `Point` object should be represented by a string, e.g., in order to print it. After the class definition, two functions are defined: `closerToOrig` and `findMax`. The functions are not part of the `Point` class, because their indentation is not the same as the indentation of the class body. The function `closerToOrig` takes two points and determines if the first is closer to the origin, i.e., `Point(0,0)`, than the second. The `findMax` function is a generic function that takes a non-empty list of elements and a compare function `lessthan`. The compare function determines if its first argument is less than its second argument. Note that `closerToOrig` is such a compare function.

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2) ** .5

    def __str__(self):
        return ("(" + str(self.x) +
                ", " + str(self.y) +
                ")")

def closerToOrig(p0,p1):
    return (p0.dist(Point(0,0)) <
            p1.dist(Point(0,0)))

def findMax(list, lessthan):
    if len(list)>0:
        m = list[0]
        for i in list[1:]:
            if lessthan(m, i):
                m = i
        return m
    else:
        print "No max in empty list"
```

Given a list of elements and a suitable compare function on the elements, `findMax` finds a maximal element in the list with respect to the compare function. For instance, given a list of points, `findMax` can be used to determine a point that is at least as far from the origin as all other points. For example, consider the following Python code. On the first line, a list `l` of three points is created. On the second line, this list is printed. The `map` function takes a function and a list and applies the function on each element in the list. The function `str` returns a string representation of its argument. If applied to a `Point`, it calls the special method `__str__` defined above. The third line creates a list of numbers representing the distance of the points in list `l` to the origin. Finally, the fifth line calls the `findMax` function with arguments `l` and `closerToOrig` in order to find a point in `l` that is at least as for from the origin as all other points in `l`.

```
l = [Point(3,4), Point(), Point(2,1)]
print map(str, l)
d = map(lambda x: x.dist(Point()), l)
print map(str, d)
m = findMax(l, closerToOrig)
print m
```

The output of this Python code is:

```
['(3, 4)', '(0, 0)', '(2, 1)']
['5.0', '0.0', '2.2360679775']
(3, 4)
```

Standard library

Python comes with an extensive standard library organized in *modules* and *packages*. Furthermore, the standard library is mostly platform independent. People familiar with Java will recognize much of the functionality, like network programming, threads, and a standard windowing toolkit. In addition, it includes modules that define Perl-like regular expressions and powerful string operations. In this section, I will discuss some functionality of Python's standard library. For more detailed information, see [6, 2].

Internet Internet programming is one of the most important application domains of Python. One of the reasons for Python's popularity is that the standard library provides functionality by which both server and client side Internet applications can be written. For example, the modules `urlparse` and `mimetools` provide functionality to manipulate url strings and mime encoded messages, respectively. In addition to these modules, there are modules to process HTML, XML, and SGML documents,

modules that provide HTTP servers, and modules to write CGI scripts. The fact that many CGI scripts are written in Python and that there exist full size web-applications, like Zope (http://www.zope.org/), shows that Python is popular among internet application programmers.

Operating system services Python has build in functionality to read and write files. In addition, the standard library offers functionality to handle files and directories, sub-processes, streams, and pipes. The sub-processes need not be Python programs, but can be any program that runs on your system. In this way, Python can be used to control different applications or as a communication means between different applications.

Profiling Python comes with a *deterministic profiler*. The online Python reference describes deterministic profiling as follows:

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

A profiler is an important tool for an extensible scripting language, since it enables software developers to analyze an application thoroughly and make the right decisions about which routines are time critical and should be implemented in a system programming language, and which routines are less time critical and can therefore be written in the scripting language. Below, I will explain the possible role of the Python profiler in a software development process.

Serialization Serialization is the transformation of a (run-time) data structure into a sequence of bytes such that it is possible to recover the original data structure from the sequence of bytes. In Python's standard library, several modules exist to serialize arbitrary objects. Furthermore, serialization is platform independent. Therefore, it is quite easy to store the current state of an application as a sequence of bytes in a file, transfer it to another computer (which also runs Python), and to continue with the application in the same state on that computer. Usually, serialization is applied not to complete applications, but to some crucial data structures of the applications that should be available the next time the application is executed.

Threads Python supports multi-threaded applications. The threading modules resemble to some extent the threading mechanism of Java. Multi threading is very useful for writing server applications. For example, an HTTP server is usually written using multiple threads. In its main loop it waits for a client to make a connection. As soon as a client makes a request, the server creates a new thread that processes the request. During the processing of the new thread, the main loop is ready to accept a new request.

Windowing toolkit A common application domain of Python is graphical user interfaces. Since the standard library has a windowing toolkit, named by `tkinter` and derived from Tcl/Tk's UI widgets, writing a user interface in Python has the advantage of being platform independent.

Python glue

One of the goals of Python is to act as a glue language that connects different applications and libraries. To be more precise, Python was developed to be used in an open environment in which Python programs could be integrated with non-Python programs. Therefore, Python was developed to be embed-able as well as extensible and interfaces of how to embed and extend Python are well documented, see http://www.python.org/doc/current/ext/ext.html, Chapters 14 and 15 of [6], or Appendix B of [2]. As a glue language, Python greatly facilitates reuse of existing code, for example, see [3].

Embedding Python means integrating the Python interpreter in another application such that Python programs can be run from within the other application. This effective adds all of Python's scripting power to the hosting application. Extending

Python means integrating applications or libraries in the Python interpreter such that its is available from within Python programs. It is possible to embed and extend Python at the same time. As usual, such a union based on equality can be very fruitful.

If Python is used to glue applications and libraries together, care should be taken that it does not replace techniques especially designed to act as an interface between software components. In fact, using Python as a glue language and using a standardized interface technique should be orthogonal design decisions. For example, if the application is supposed to be available at some *object market*, see [10], its interface should be defined using a standardized interface technique, e.g., CORBA or XML, instead of Python.

So, if there are good arguments to use CORBA in a situation where Python is not used for integration, then it should still be used if Python is used for integration. This claim can be turned around as well: if Python can be used for integration, then using a standardized interface technique is probably too much overhead. As is explained below, integrating existing code with Python requires the interface (C/C++ header files) of the code to be available which can be problematic in a commercial environment. However, in that case, integration without Python is at least as big a problem.

A prerequisite of extending Python with a given library is that the interface of the library is defined in C-header files or that the source code is available in C or C++. This is a limitation, since there are useful libraries out there for which no C-header files exists. However, for almost any subject there exist C and C++ libraries as well or if the source is available in, say, Fortran, then writing a C-header file for it is not too difficult. Furthermore, if Python should be integrated with Java applications, one should consider using *Jython*: a Python implementation written in Java, see http://www.jython.org. It is said that Jython-Java integration is better than the conventional Python-C/C++ integration, since no recompilation of Java code is needed due to Java's reflection API. However, since I have no experience with Jython, I will only focus on the Python-C/C++ combination.

Extending Python with an existing library effectively means that a wrapper for the library has to be created and together with the wrapper, the library has to be turned in an object file that can be loaded dynamically e.g., shared libraries or DLLs, or that is linked statically with the Python interpreter. The wrapper should take care of the translation between data structures of Python and the data structures of the library. The conversion between C/C++ and Python data structures is documented extensively and, therefore, after some reading, not difficult.

Tools have been developed that create wrappers automatically. SWIG is one of such tools and stands for *Simplified Wrapper and Interface Generator*, see http://www.swig.org/. SWIG is not just a tool to create wrappers and interfaces for Python, it can also generate interfaces for other languages, e.g., Perl and Tcl/Tk. SWIG comes with extensive documentation and the SWIG user guide (available on http://www.swig.org/doc.html) has devoted one chapter to the combination of SWIG and Python.

Example of a Python Extension Since extensibility of Python is one of its most powerful features, I spend the remainder of this section to describe my experience with extending Python with an 'off-the-shelve' BDD library. A BDD (*binary decision diagram*) is a data structure to store boolean functions [4] space efficiently. For this article, it is not necessary to explain BDDs, but it suffices to give some examples of what can be done with BDDs. First of all, BDDs manipulate boolean function symbolically. For example, given a BDD for two boolean functions f_0 and f_1, there are BDD operations to compute a BDD for the function $\mathit{and}(f_0, f_1)$ defined by

$$\mathit{and}(f_0, f_1)(b) = f_0(b) \wedge f_1(b).$$

There are also operations to compute other common boolean operations, like $\vee$, $\rightarrow$, etc. In addition to these symbolic operations on boolean functions, a BDD library provides routines to determine if a boolean function (represented by a BDD) can return *true* for some concrete values of its arguments. That is, there are routines that determine if a boolean formula can be satisfied. Given that almost any problem defined formally can be translated into a problem defined in boolean formulas, BDD libraries can be used an many areas. Historically, BDDs have been applied mostly to tasks in digital system design, verification, and testing.

The BDD library I chose is called *BuDDy* and its

source code is freely available. It can be downloaded from http://www.itu.dk/research/buddy/. There is no good reason why I chose this BDD package; it just happened to be the first package I found that was freely available and installed without problems on my machine. BuDDy is written in C and has some additional definitions to use it in C++.

Extending Python with BuDDy was not a complicated task, thanks to SWIG. The main difficulties were in dealing with pointer arguments and function pointers, since SWIG does not process them automatically. So, in these cases I had to write some extra code in a so-called SWIG interface file. After that, SWIG generates the wrappers which could be compiled and linked with the original BuDDy code into a Python module. Note that the BuDDy code is left unchanged

So, the functionality of BuDDy is now available to Python programs. However, it is at a somewhat low level; python programs directly call C functions to generate and manipulate BDDs. Furthermore, since garbage collection of objects created by BuDDy is left to the programmer, the Python code quickly becomes a unreadable mess of function calls and temporary variables. Note that this is more a problem of BuDDy than of Python; the C-examples that come with BuDDy exhibit the same mess of function calls and temporary variables. To make it better accessible, BuDDy has a C++ class that takes care of automatic garbage collection and overloads some operators such that function calls can be written as operator applications. I did the same in Python and wrote a class that defines BDDs as normal Python objects. Also, I overloaded some Python operators in the same way the C++ class did. As a result, the Python code is at least as readable as the C++ code. For example, the following listing shows some lines of C++ code of an implementation of the N-queens problem in C++ using BuDDy (here, `X` is a two dimensional array of bdds and `a`, `b`, `c`, and `d` are bdd variables):

```
bdd a=bddtrue,
    b=bddtrue,
    c=bddtrue,
    d=bddtrue;
int k,l;

/* No one in the same column */
for (l=0 ; l<N ; l++)
   if (l != j)
      a = a & (X[i][j]
               >> !X[i][l]);
```

The corresponding lines of Python code for the N-queens problem reads:

```
a = bddtrue
b = bddtrue
c = bddtrue
d = bddtrue

# No one in the same column
for l in range(0,N):
    if (l != j):
        a = a & (X[i][j]
                 >>  -X[i][l])
```

Software development with Python

Sometimes, scripting languages are said to be good for prototyping, but not for real application development. A prototype bears the associations of 'quick and dirty' and 'to be thrown away.' However, Python is more than just a prototype language. Due to its clear syntax and its universal object model, reuse of Python programs is a very attractive option. Therefore, a substantial part of Python code of a prototype of an application could very well end up in the code of the final application.

So, what is Python's role in a software development process? First of all, it can be used for prototyping; like any scripting language, it enables programmers to write quickly a mock up of an application in order to analyze the feasibility of the project.

A simplified and Python centered, view on software development could be described as follows. Firstly, determine user requirements of the application and build a prototype in Python. Next, a development cycle is started that consist of assessment of the prototype, estimation of costs of improving the prototype, and finally a decision whether to improve the prototype or to abort the cycle and declare the current prototype the final application.

During each cycle, assessment of the prototype can lead to new or more precise requirements. Analysis of the prototype shows, among others, computation intensive code, which could be implemented in a system programming language. The Python profiler is very useful to detect computation intensive code. If, after some runs of the development cycle, all computation intensive code is implemented in a system programming language, there is probably not much more speed to gain. At that time, it

is a waste of time to translate the remaining Python code into a system programming language.

Conclusions

In this article I have discussed the Python language. Python is a scripting language with clear syntax and an extensive standard library. It supports, but does not enforce, procedural, functional, and object oriented programming styles. Unlike many other scripting languages, Pyhton code is readable and, therefore, reusable. Reusability is even more supported by Python's platform independence. Python can play an important role in software development, since it is a powerful tool for prototyping as well as for implementing the final application. If the application contains computational intensive code, which will be too slow if programmed in a scripting language like Python, the extension interface of Python makes it very easy to implement this code in a system programming language like C/C++. Furthermore, together with its embedding interface, the extension interface of Python enables efficient integration with existing applications and libraries.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1997.

[2] David M. Beazley. *Python Essential Reference*. New Riders, 2000.

[3] David M. Beazley and Peter S. Lomdahl. Feeding a large-scale physics application to python. In *Proceedings of the 6th International Python Conference*, San Jose, California, October 1997.

[4] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] Leo Geurts, Lambert Meertens, and Steven Pemberton. *The ABC Programmer's Handbook*. Prentice-Hall, 1990. To be republished by the CWI. See also http://www.cwi.nl/ steven/abc/.

[6] Mark Lutz. *Programming Python*. O'Reilly & Associates, first edition, October 1996.

[7] Greg Nelson, editor. *System Programming with Modula-3*. Series in Innovative Technology. Prentice Hall, 1991.

[8] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000.

[10] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[11] Guido van Rossum. Foreword for Programming Python, May 1996. See [6]. Also available on http://www.python.org/doc/essays/foreword.html.

[12] Larry Wall, Tom Christiansen, and Randal L. Schwarz. *Programming Perl*. O'Reilly & Associates, 2nd edition, 1996.

Biography Since January 1998, Victor Bos is a PhD student at the Eindhoven University of Technology. He is involved in formal methods research at the computer science department. His current interest lies in applying formal method techniques to industrial engineering and therefore he works closely together with the Systems Engineering Group at the department of Mechanical Engineering. He was an OOTI from 1996–1998. In December 1995, he received his masters degree in computer science at the University of Groningen.

Overview Latest OOTI Reports

The post-masters programme OOTI is concluded with a design project. The final reports of these projects are in general publicly available, unless stated otherwise. The following reports have been published lately.

- Venemans, B.M.
  Redesign of a flexible cross-platform communication utility,
  Keywords: Embedded software / Host-target communication /
  ISBN 90-444-0092-4, 34 p., March 2001
- Garcia, P. and B. Xu
  Introduction of Presentation State into EasyVision,
  Keywords: Presentation State / EasyVision / OOTI
  ISBN 90-444-0-0093-2, 45p., March 2001
- Manolache, C.D. and M.F. Zelina
  In-Home Network Simulation Framework II,
  Keywords: JINI / In-home Network
  ISBN 90-444-0088-6, 78p., March 2001

Copies of these reports are available through the secretariat of the post-masters programme Software Technology (OOTI), tel +31 40 247 4334.