The Lazy Programmer

Judith Bishop

Computer Science Department
University of Pretoria
Pretoria 0002, South Africa

jbishop@cs.up.ac.za

ABSTRACT

The concept of laziness is defined according to the practice, advocated by Dijkstra, of writing a program correctly from the beginning, so as to rule out costly testing and re-writing. I trace the meaning of laziness in the life of my friend Derrick Kourie, and contrast it with some of my pet ideals, such as languages, tools and design patterns. The limits of being a Lazy Programmer in today's object-oriented and concurrent world are explored. How one adapts the principles of laziness in teaching is revealed from recent work of Derrick's. History plays a part in all we do, and references that reflect our path as computer science academics in South Africa are included.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – Classes and objects, Inheritance, Patterns

General Terms

Algorithms, Design, Reliability, Human Factors, Languages, Concurrency

1. INTRODUCTION

1.1 The anecdote

I am writing this paper for my dear friend and colleague, Derrick Kourie. I have known Derrick since 1970, when we both attended NCFS conferences together, planning protests against forced removals, the loss of academic freedom, the banning and imprisonment of priests and student leaders, and the attacks on schools and missions. NCFS stands for National Catholic Federation of Students and had representatives from all the universities. I was from Rhodes, Derrick was from UP, and we met on common ground at Marianhill in Natal when Steve Biko tried to force a walk out of black students from multi-racial organizations into SASO [13]. In 1970, he did not succeed, thanks to the power of prayer and diplomacy of our mighty leader and friend, Jan d'Oliveira. Or maybe our members just loved my bagpipes. Derrick often says he remembers me marching around playing away "while Rome burned" as it were.

1.2 The man

Derrick is the perfect good friend. He is articulate, smart, warm of heart and always ready for a chat or a skinner. It is seldom that there is not someone in his office seeking advice or sharing a joke. And his jokes are heard in high places, with the Vice Chancellor on his email list!

Derrick loves good food and casts scorn on people who skimp on the good things of life. These include very simple pleasures, like a once a day cigarette from the Campus Kiosk – an excuse to walk across our lovely gardens and get away from the colleagues beating a trail to his door maybe?

Figure 1 Derrick, Kourie at Moyo, January 2006

I perceive Derrick as deeply religious in the very best sense. His faith in God and the future is unshakable, his love for people shines through everything, and he never loses hope in hopeless cases. Many a student has been steered to a pass for an MSc though his meticulous editing, and many a paper has squeaked in because of his fine wordsmithing. Derrick's family mean everything to him, and he is a proud and devoted parent. I remember when I had to return from the UK lock, stock and barrel in 1991, under stressful circumstances which placed severe strains on my faith, he told me: "Don't worry – God has a place for everyone in heaven – with parents in the front row."

1.3 The background

In 1972, the programming scene was dominated by languages such as Fortran, Algol, Cobol and PL/I. IBM (Big Blue), with its strangle-hold on software bundled with hardware, was the enemy of academics – a mantel that it has since been relieved to pass on to Microsoft. The Garmisch Software Engineering Conference had taken place four years ago, and people were well aware that there was a software crisis looming.

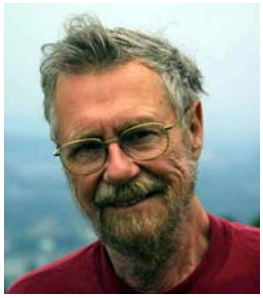

Figure 2 Edsger Dijkstra

Enter one of the greatest computer scientists who ever lived, Edsger W Dijkstra (1930-2002). Dijkstra was in his prime, working, at Burroughs Corporation in the US, when he received the ACM Turing Award 1972. His address given in response to

the award was entitled "The Humble Programmer" and it became a classic for computer scientists, gaining more than 2 000 citations to date [7]. Dijkstra's simple but revolutionary idea was that the key to producing reliable software is to avoid introducing bugs, rather than eliminate them later. In concluding his speech, he made the now famous quote:

> *"We shall do a much better programming job provided that we approach the task with a full appreciation of its tremendous difficulty; provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers"[7].*

Dijkstra did not only pontificate – though he was very good at that. He also produced the earliest multi-programming operating system (the THE) [10], and was well known for his classic papers on cooperating sequential processes [8], and his memorable indictment of the go-to statement back in 1968 [9]. It is interesting to note that Derrick's research group Espresso has close ties with Dijkstra's old group at TU Eindhoven through Bruce Watson.

Dijkstra also had a keen interest in teaching and in 1989 published an article sarcastically called "On the Cruelty of Really Teaching Computer Science" [6] in which he challenged teachers to follow a formal mathematical approach to programming, ending once again with a rallying cry:

> *"Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding. Within a few months, they find their way in a new world with a justified degree of confidence that is radically novel for them; within a few months, their concept of intellectual culture has acquired a radically novel dimension."[6]*

Figure 3 Cover of the SAJS Issue, January 1991

In that same year, 1989, I gave my inaugural lecture as a professor at the University of the Witwatersrand, coincidentally addressing a similar topic. My talk was ambiguously entitled: "Computer Programming: Is It Computer Science?" My conclusion was that there is a

> *"vast contribution that computer scientists have made, and are still making, to the goal of making programs more readable, writable and reliable. There is a crying need for these advances to be more widely known and accepted, and for computer scientists to take their rightful places as the experts in the programming arena, to be called in, as Wirth says, when the going gets tough.".[17]*

And what was Derrick's response to all this? It was twofold. He started teaching the Dijkstrasian approach to program construction to Honours students in 2003, which he has successfully presented ever since. It is now called FAC751 (Formal Aspects of Computing) and attracts 10-15 students a year. But taking a broader view of teaching, he presented a keynote address at SACLA, entitled, not entirely tongue in cheek, "On the Benefits of Bad Teaching" [19].

In this address, he itemized the characteristics of good teaching: the selection of appropriate course material, good organization, good delivery, good reinforcement procedures and good assessment procedures, but then went on to argue that if one is lazy and does not reach perfection, or even high standards, in these areas, students will not be irreparably harmed.

Good teaching is inherently time-cost inefficient, and by underplaying, neglecting or ignoring it we might actually be advancing our students' academic maturity! This observation is so much like that of Dijkstra's that I shall add one more quotation from the master:

> *"I was recently exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen it was such an obvious case of curriculum infantilization that its author should be cited for contempt of the student body". [6]*

1.4 The topic

Derrick claims he is lazy and that he likes the good life too much to be diligent. His impressive list of publications, books and successful postgraduates belies this notion. But he has espoused laziness in the Dijkstrasian sense: if a program can be written correctly the first time, surely we can save a lot of trouble in debugging, and all go home early? His work on program construction, and the notes for FAC751 are testimony to this ideal.

Predictably, Derrick and I don't always see eye to eye on programming (especially when it comes to programming standards!) and so it is in recognition of my deep respect and affection for a great man who is dear to all of us, that I willingly take up my pen to write an article that sees the world through his eyes – or rather, through bifocals. That is, I intend weaving Derrick's ideas with my own, exploring the ideal of laziness and seeing how indeed we can put it to work for our advantage. In so doing, I shall attempt to summarize some of Derrick's ideas on Software by Construction, and intersperse them with my own views from the world of programming languages and distributed systems. Then I shall take a few steps back and examine the limits of laziness: how lazy can we be, before it gets out of control? And that's enough for a Lazy paper!

2. APPROACHES TO LAZINESS

2.1 Software by construction

The basis of lazy programming is to confine oneself to intellectually manageable programs [7]. This is not as severe a restriction as one might imagine, provided we apply our

intellects in the first instance, rather than head directly for the keyboard. Moreover, every large program consists of many small ones (as component developers will assure us). Then, we proceed to construct the program according to mathematical rules, so that at the end, we *know* it is correct and no bugs are possible. Going home time!

In [18], Kourie summarizes the process as follows.

- Characterize what we expect a piece of code to achieve (the code's postcondition).
- Characterize the starting off scenario (i.e. the precondition) that should be in place in order for a particular piece of code to attain the required postcondition.
- Specify pre- and postconditions in an appropriate notation such as Dijkstra's GCL.
- Evolve, in a series of refinement steps, code that solves the problem— i.e. code whose execution is guaranteed to end up in a state consistent with the postcondition, provided that it is starts off in a state that adheres to the pre-condition.
- Refine incrementally the specifications for various problems according to established refinement laws such as Hoare triples or Morgan's laws.

This process differs significantly from ordinary programming in that the notation is firmly mathematical, and that the derivation of the code proceeds according to the application of laws, rather than by the programmer's intuition and inventiveness. Nevertheless ingenuity and manipulative prowess are still required in large measure. It is just that the anticipated payoffs are much higher using this method, i.e. a bug free result. Figure 4 shows an extract from Kourie's notes, where he goes through the logical steps required to set up the postcondition for a loop.

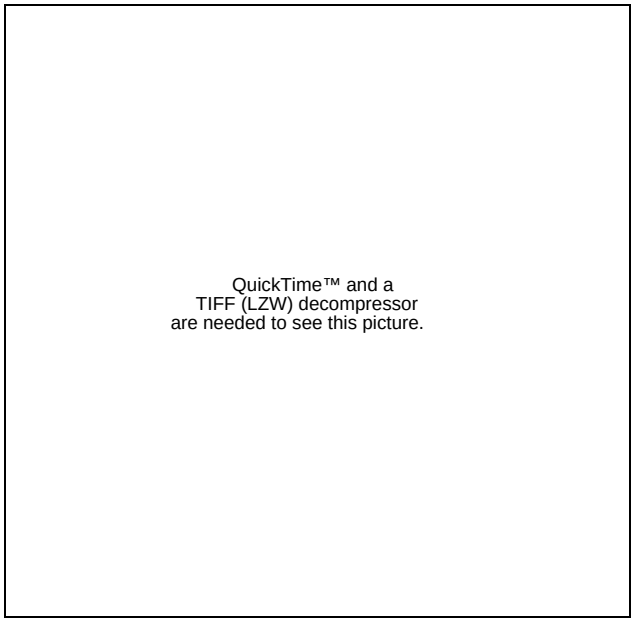

Figure 4 Example of software by construction

2.2 The place of design patterns

In a mathematical system based on laws, one builds up an arsenal of established algorithms that can be slotted in later. In my parallel world of object-oriented programming, these would be equivalent to the methods, classes or APIs that are typically built in the course of large system construction. The hope is that through testing small parts and coming to some assurance that they work, their deployment in larger units will be successful.

A different aspect of modern programming is design patterns, which encapsulate common circumstances and show how they should be developed. For example, an observer pattern identifies subject and observer roles, and defines the interaction between them ([15] and Figure 5).

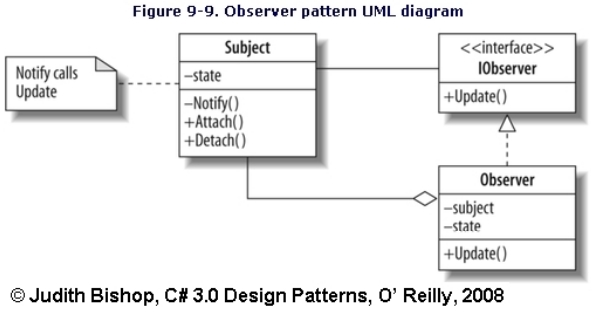

Figure 5 The Observer Design Pattern

I find myself thinking about how design patterns fit in with software by construction, and the closest seems to be the refinement rules. Design patterns have names, which establish their place in discourse, and they are always represented in the same way. That is the essence of a pattern. They do not have to be implemented in the same way, though, as different languages will always provide different levels of abstraction to work from [16].

2.3 Advances in languages

In comparing my situation to that of Derrick's, I find that the major difference between us concerns types. Software by construction concentrates on the algorithmic niceties and complexities of looping around arrays. My programming hurdle is always to set up the classes, collections and permissions so that an accurate view of a complex world can be depicted in a program, with maximum abstraction and security.

In my world, laziness with respect to languages can be espoused in two ways. As a language enthusiast, I tend to embrace new features and quickly find how they can be turned to my advantage, so I can be more lazy and let the compiler find silly bugs. For example, automatic properties in C# 3.0 relieve the programmer of much tedious and error-prone get-and-set coding, as in

```
string Name {get;}
```

which sets up read-only access to a private local string field. I have found that for very complicated type related programs, where inheritance, generics, polymorphism and overloading are all intertwined, it is essential to have the compiler there, checking the rules. Time and time again, once the compiler lets the program through, I have had the heady experience of the program running clear first time. This is surely one of the goals of Software by Construction, only I am being ultra-lazy by getting a compiler to help me.

Another opportunity for laziness is to follow the herd. I have noticed that over the 50 years of language development, defaults

for features have become more and more sensible. Values are now initialized by the system to zero or null; switch statements take you where you would like to go after a branch is executed; operator precedence has been sorted out so that the number of parentheses has halved. What amazes me is that there are programmers and teachers who still use the old ways, e.g.

```
int i=0; // initialization redundant
if ((i<n) && (a[i]!=x))
        // inner parentheses unnecessary
```

The first example is the result of JVM technology, the second in advances in compiler construction. This lazy programmer says thanks to both communities for their efforts, and she would certainly write:

```
int i;
if ((i<n && a[i]!=x)
```

2.4 Verification and tool support

Over the years I have experienced a confusion that has probably been shared by many other programmers who do not come into day-to-day contact with formal methods. I heard about program verification, and assumed that this would be a brilliant idea. Write a program, submit it to a verifier, and get a yay or naycomes out the back. Naturally that was not the idea at all. Verifiers do exist, but in order for them to do their work, they need a specification, and the specification is exactly the maths that software by construction builds up. Dijkstra makes the point that one should not first make the program and then prove its correctness [7], since that would only increase the poor programmer's burden. However, there is nothing wrong with having a helping hand with the proof that goes hand in hand with the construction.

In 2005, I was involved by association in one of the most famous verifiers, ACL2, when I was chair of the ACM Software System Award Panel that recognized the work of Boyer, Moore and Kaufmann.

> *"ACL2 is a very large, multipurpose system. You can use it as a programming language, a specification language, a modeling language, a formal mathematical logic, or a semi-automatic theorem prover, just to name its most common uses."[12]*

I have not delved into ACL2 but it would seem to me that lazy programmers would regard it as a boon. There is a library of proofs, and in the modern community style, one can go in and add more. I do, however, have some experience with Spec#, Microsoft's contribution towards a more cost effective way to develop and maintain high-quality software, and have been teaching it in COS333 Programming Languages for five years now.

The Spec# system consists of an **extended language** with non-null types, checked exceptions, method contracts in the form of pre- and postconditions as well as object invariants; a **compiler** that statically enforces all this and records contracts as metadata for consumption by downstream tools; and the Spec# **static program verifier**. This component (codenamed Boogie) generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

What makes Spec# exciting for me is that it lives in the real world: it even guarantees maintaining invariants in object-oriented programs in the presence of callbacks, threads, and inter-object relationships. Some of the best minds in the discipline were called in to develop the science behind this (see for example [20]). An example of a Spec# program is given in Figure 6.

```
public void sortArray(int[]! a)
  modifies a[*];
  ensures forall{int j in (0: a.Length), int i
in (0: j); a[i] <= a[j]};
{
 for (int k = 0; k < a.Length; k++)
  invariant 0 <= k && k <= a.Length;
  invariant forall{int j in (0: k), int i in (0:
j); a[i] <= a[j]};
 {

  // Inner loop – see next slide
 }
}
```

Figure 6 Program specification in Spec#

Although the notation is not as concise or mathematically elegant as GLC, it has a familiar feel for some programmers. Since Spec# has all the power of GLC and more, as a Lazy Programmer, I would not go into the world of specification without a tool like this.

2.5 Lazy evaluation

Last on my list is lazy evaluation, which used to be something that functional programmers whispered about in the past, but which is now coming into the mainstream as databases are being connected up over the internet, and programs can interrogate unknown sizes of data. I now use it regularly in small and large programs through the new yield-based iterator in C# 3.0 which is connected to the `select` syntax. `select` is a statement in C# 3.0 which mirrors the SQL version and can connect to data sources in memory or on databases or over the internet. The program stays the same. For example, the following request:

```
var selection = from p in family
•      where p.Birth > 1980
•       orderby p.Name
•       select p;
}
```

links up with a user-defined yield-based iterator for a tree as in Figure 7.

```
//C# 3.0
public IEnumerable <T> Preorder {
• get {return ScanPreorder (root);}
}
••
// Enumerator with T as Person
•  private IEnumerable <T>
     ScanPreorder (Node <T> root) {
•    yield return root.Data;
•    if (root.Left !=null)
•      foreach (T p in ScanPreorder (root.Left))
•          yield return p;
•    if (root.Right !=null)
•      foreach (T p in ScanPreorder(root.Right))
•          yield return p;
•    }
•  }
```

Figure 7 Yield-based iterator for a tree

The remarkable bit is that all the above code does absolutely nothing! It is really lazy, until along comes the loop statement:

```
foreach (Person p in selection)
•      Console.Write(p+"   ");
```

Then the values for selection, as defined by the select statement with its filters are generated one by one and returned via the yield statement.

It is interesting to note that Java is not as lazy as C# 3.0 since it does not have a yield statement. It does have a foreach, but two foreaches cannot interact in the way above (like coroutines). therefore Java is restricted to working with simple linear or predefined collection types.

3. THE LIMITS OF LAZINESS

3.1 Not all problems are small

It might be naïve to claim that programming in this way is ***the*** silver bullet that will solve the software crisis, but it is certainly one way. Others have proposed and supported the methodology, notably Gries, in his seminal book "The Science of Programming" [4] and Hoare [2], who referred to it as the axiomatic method. However, by the 1980s, interest had picked up in data abstraction, and was moving away from "straight line programs" as they were termed by Liskov and Guttag [1]. It is therefore not so much a question of size, but of the nature of the program. These days too much of what we do is oriented away from number crunching, bin packing and sorting, and towards data access and manipulation, with semantics, networking, user interfaces and security being highly important aspects of the whole system.

No matter what the form, most people acknowledge that more formalism is necessary in critical situations. Niklaus Wirth also makes the point that our software is just too big, and that we could have leaner programs in the first place [21]. He would certainly get a badge for Laziness in this respect, although he still must be the one person who has invented more mainstream languages, as well as computers, ever. Question: guess how many. (Answers in the talk.)

3.2 It's an age thing

The acceptance of a lazy approach to programming requires either that you are caught very young, or that you have a certain maturity of outlook. As Derrick will know, the Jesuits had the view that they had to grab the hearts and minds of children by the time they were 7! Rogers and Hammerstein had a similar notion in South Pacific:

> *You've got to be taught before it's too late,*
> *before you are six or seven or eight,*
> *to hate all the people your relatives hate –*
> *you've got be carefully taught!"*

On the other hand, age can also make us more set in our ways, less open to trying new things. Our laziness becomes a trap, and might even prevent us from adopting new ways which could enable us to be more productive with less effort – i.e. ever more lazy.

3.3 The advance of the multi-cores

Sometimes technical advances push us out of a nice lazy path. The current advance of the multi-core processors is one such. How are we to program them, and how will our current methods adapt in the face of multiple processors? We have been through this trauma before in the 1980s notably with transputers. Both Derrick and I spoke at the first conference on Parallel Processing in South Africa in 1989 [22]. It would be interesting to track the development of Derrick's think away from object orientation towards algorithms over the past 15 years – catch him at tea.

It was Hoare who was the most famous for CSP, the basis of much of the concurrent programming we do today [3]. Derrick teaches an updated version of CSP (using a tool!) to second years in the COS226 Concurrency course which has been running for more than 10 years now, every since Jeff Kramer visited me in 1996. The notation in the Jeff and Jeff book [14] also now has assertions, as shown in Figure 8.

```
/** Concurrency: State Models and Java Programs
 *              Jeff Magee and Jeff Kramer
 *
 */

const N = 2
range Int = 0..N

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
                 ).

LOOP = (mutex.down->enter->exit->mutex.up->LOOP).

||SEMADEMO = (p[1..N]:LOOP
              || {p[1..N]}::mutex:SEMAPHORE(2)).

fluent CRITICAL[i:1..N] = <p[i].enter, p[i].exit>

assert MUTEX = []!(CRITICAL[1] && CRITICAL[2])

assert MUTEX_N = []!(exists [i:1..N-1] (CRITICAL[i] && CRITICAL[i+1..N] ))
```

Figure 8 FSP Specification with assertions

The tools that accompany this notation do enable visualization (sorry, Dijkstra) and also allow for testing for properties such as liveness and progress, which are essential in the concurrent world.

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Figure 9 **LTSA** Animation

Derrick is also a fan of David Harel, whose work on Statecharts earned him the 2007 ACM Software System Award [5].

Where we are going now, it is hard to say. New languages and methodologies are popping up, such as Erlang and Skala, leaving us not much time to laze about. Certainly, the objective seems to be to keep all the computers as busy as can be. If that means we can write a program once and have it run on lots of workers, we will have achieved a lot.

4. CONCLUSION

One of the joys of being an academic is having academic freedom, loosely defined as the right to think and write according to one's own interests and convictions. Despite the pressures of historical precedent or current fashion, an academic can work on a single problem for decades, or define new directions every week. Derrick is your true academic. He lets his mind wander, he keeps up with what is new, but he also makes the old his own. He has published papers in spectacularly prestigious journals, and his research group is the envy of us all. His work on and promotion of Software by Construction goes hand in hand with all the other research and teaching he does. That would not have been possible if he had

not indeed, from a very early age, decided to espouse the ideals discussed above and become a Very Lazy Programmer.

REFERENCES

[1] Barabar Liskov and John Guttag, Abstraction and specification in program development, MIT Press, 1986.

[2] C. A. R. Hoare: An Axiomatic Basis for Computer Programming. Commun. ACM 12(10): 576-580 (1969)

[3] C. A. R. Hoare: Communicating Sequential Processes. Commun. ACM 21(8): 666-677 (1978)

[4] David Gries, the Science of Programming, Springer-Verlag 1981

[5] David Harel: Statecharts in the making: a personal account. HOPL 2007: 1-43

[6] Edsger W. Dijkstra, E W, On the Cruelty of Really Teaching Computing Science, Comm. ACM 32(12):1398:1410, 1989

[7] Edsger W. Dijkstra, E W, The Humble Programmer, (1972 ACM Turing Award Lecture). Comm. ACM 15(10): 859-866,1972

[8] Edsger W. Dijkstra: Guarded commands, non-determinancy and a calculus for the derivation of programs. Proc. Language Hierarchies and Interfaces , 111-124, 1975

[9] Edsger W. Dijkstra: Letters to the editor: go to statement considered harmful. Commun. ACM 11(3): 147-148, 1968

[10] Edsger W. Dijkstra: The Structure of "THE"-Multiprogramming System. Commun. ACM 11(5): 341-346, 1968

[11] http://research.microsoft.com/SpecSharp/

[12] http://www.cs.utexas.edu/~moore/acl2/acl2-doc.html

[13] http://www.sahistory.org.za/pages/governence-projects/organisations/SASO/saso-history.htm

[14] Jeff Magee and Jeff Kramer, Concurrency: State Models & Java Programs, Wiley, 2nd ed, 2006

[15] Judith Bishop, C# 3.0 Design Patterns, O'Reilly, 2008

[16] Judith Bishop, Language features meet design patterns: raising the abstraction bar , Workshop on the Role of Abstraction in Software Engineering, at ICSE 2008, Leipzig, May, 2008

[17] Judy M Bishop, Computer programming: is it Computer Science?, SA Journal of Science, 87(Jan-Feb): 22-33, 1991

[18] Kourie D G, FAC751: Formal Aspects of Computing: Software-by-Construction, Notes, Department of Computer Science, University of Pretoria, February 2008

[19] Kourie D G, On the Benefits of Bad Teaching, SACLA 2001 Procedings, ppiv-viii, (Keynote address) June 2001.

[20] Mike Barnett, Rob DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. JOT 3(6), 2004.

[21] Niklaus Wirth: A Plea for Lean Software. IEEE Computer 28(2): 64-68 (1995)

[22] RJ van der Heever and DG Kourie, Design of distributed systems: object-oriented event-driven approach, Parallel processing: technology and applications, IOS Amsterdam, 113-123, 1989