

# Modular Concurrency

CUSEC Keynote

**Peter Grogono**

20 January 2006

## 1 Introduction

Good morning and welcome to CUSEC. This is the fifth CUSEC — I think — and it looks very exciting. It is the fourth CUSEC for which I've been asked to give a keynote and I am greatly honoured — not to mention somewhat surprised — to be asked again.

The organizers insist that Mid-January is the best time for CUSEC. So we've had CUSECs that were buried in snow, CUSECs that were bitterly cold — you guys from California and Florida may think this is cold, but you ain't seen nuttin' — and CUSECs where everything is covered in ice, like this one. The result, of course, is we get good attendance because no one dares to go outside.

Previously, I've given the opening keynote or the banquet speech. This year, I submitted my abstract as usual and found myself put back to the second day. I guess that's my punishment for proposing something serious. But that's OK, Chad Fowler got us off to a rousing start yesterday.

My theme today is my current work. To be more precise, my current collaboration. 1

The gentleman on the right is Brian Shearing. I am the one on the left. You can recognize computer people of my generation because they wear beards and have glasses with large rims. The men, that is. You can tell the academics from the industrial types because industrial types have hair.

It's important to mention Brian for several reasons. One is that he has all the ideas and does most of the work. More importantly, Brian has spent his working life developing software that people want, so you can't dismiss our project as an ivory-tower fantasy.

Brian and I go back a long way. Here we are, twenty years ago, discussing traps and pitfalls of object-oriented programming. We started talking about programming as students in 19... — a long time ago. And we still are. This morning, I will try to explain why the foundations are shaking, what needs to be done about it, are how we are doing it. 2

## 2 Background

I'll start with an anecdote from the early days of computing. It is taken<sup>1</sup> from the first book on computers that I read and recently re-read: *Faster Than Thought*, edited by Bertram Vivian Bowden, later Baron Bowden of Chesterfield. It was published in 1953, soon after

---

<sup>1</sup>page 332

large airlines started to have difficulty controlling their seat sales. They had to sell at least two-thirds of the seats in each flight to be profitable, and they had to avoid over-booking. *Plus ca change*. To solve this problem, they connected all booking offices by telephone to a central office in which a bunch of girls looked at flight information written on blackboards.

A digression: you can tell this is an old story because it refers to “girls”. When I was growing up, we were taught that there were a number of words — most of them had four letters — that you were not supposed to say to a girl. Nowadays, it’s OK to say those words but it’s not OK to say “girl”. Instead, we have to say something like Graduates with Intelligence, Resourcefulness, and Literacy Skills, or G-I-R-L-S, for short.

So these GIRLS used to sit and look at blackboards. The number of blackboards and girls got larger and the room expanded. This meant that the girls had to sit far away from the blackboards. So far away, in fact, that they had to use telescopes to read the blackboards. Eventually, the system was limited by the power of the telescopes. Asked how to manage continuing expansion, the chief designer said “get better telescopes”.

The point that I will try to get across today is that we are in a similar situation: our computing systems get ever larger and more complex, and we are looking for “better telescopes”. What we need is not better telescopes but a better system.

Airline booking systems didn’t use telescopes for very long: they figured that they could use computers. The SABRE reservation system, written by IBM for American Airlines, was introduced in 1962. Here are the girls — and even some boys — using it. 3

Of course, this is a familiar idea, and we are always being exhorted with “think out of the box” and similar slogans, but this is easier to say than to do. First of all, you have to recognize that there *is* a box.

Where, or what, is our box? Let’s start by looking at an apparent contradiction. On the one hand, there are experts telling us that the future lies in raising the level of abstraction: fifth-, sixth-, *N*th- level languages, model-driven design, UML 2, and so on. These experts also tell us that there’s not much to be done in programming language design because programming languages are not the problem.

But, if programming languages are not the problem, why do we keep getting new ones? C, C++, Eiffel, Java, PHP, Javascript, perl, Python, Ruby, AJAX, . . . there seems to be a new language every month.

As an aside, it seems to me that the more people like these languages, the more they resemble LISP.

Of course, these are not all programming *languages* in a strict sense, but they are used to build applications. Moreover, these languages are all either object oriented or their designers are busy apologizing for the fact that they are not and retrofitting them with objects. And some of these languages are developing creaky joints.

### 3 Failures of Object Oriented Programming Languages

Nowadays, most of us work with object oriented programming languages (OOPLs). That includes me. I am going to be critical of OOP so, before I get started, I would like to state up front I am actually rather fond of OOP. Most of my programming these days is done with OOPLs. I wrote my first OOP around 1978, in Simula, and was immediately hooked.

There are many things that are important in software engineering. I am sure you have learned about them in the courses you are taking. I believe that one important thing is that software engineers need *solutions that scale*. Programs are large and getting larger. Many industries work with million-line programs and civil aviation has passed the billion-line mark.

The things that I am worried about may not seem to matter much and it is perfectly true that they don't matter much in a thousand-line program. But they may matter a lot in a million line program.

**Encapsulation** One of the few things that all almost software engineers seem to agree about is that complex systems consist of components that should not be tightly coupled. They go around mumbling mantras about “high cohesion and low coupling” and trying to encapsulate everything in sight. This is such common knowledge that it has even reached the textbooks.

In programming languages, encapsulation means, roughly speaking, limiting access to data. So we find in Java, for example, ways of limiting access to data:

4

public scope	delegates	local inner classes
protected scope	classes	anonymous inner classes
private scope	interfaces	member inner classes
package scope	packages	static member inner classes
instance methods	static methods	

Fourteen different ways of controlling access! Moreover, at least two useful *classical* access control techniques — block structure (outside the method level) and nested methods — are *not* provided in Java. The decision to leave them out — assuming there was such a decision — probably takes us all the way back to Ken Ritchie's decision to ignore Algol's block structure when designing C for an 8K PDP/7.

I was going to contrast Java with C++ with respect to data access. I started to count the number of possibilities in C++, allowing for public, protected, and private for instance variables and inheritance, virtual and non-virtual inheritance, and I gave up counting — and that was before I had even considered friends.

Even worse, the fourteen ways that Java — or, in fact, any other OOPL — uses to control access to data fail to protect the data! Here's why.

**Interfaces and “calling out the back”** Another thing that all software engineers — and many other kinds of engineer — seem to agree about is the separation of interface and implementation. Of course, that’s closely related to encapsulation. I hope we all agree that the data members of an object should be private. So the *interface* of the object is defined by the methods it provides. Right? Well, up to a point.

Electrical engineers work with boxes. What comes out of the box is the interface. What’s inside the box is the implementation. Here’s a box. Is this its interface? 5

No, not quite. It has a back as well as a front, and here’s the back. The interface consists of both the *front*, with the user controls, and the *back*, with all the wires going to the rest of the world. 6

How does this relate to objects? The public methods of an object are just the front view: the user controls. The *interface* consists of the public methods provided to users *and* the methods of other objects that the object calls. But in design we tend to ignore the back and worry about the front only. UML encourages this, by the way.

There’s a serious consequence: objects have no control over their private data. To see this, I’ll need to get rather technical. (There will be a short quiz at the end of the talk.) Here’s a simple class, *Foo*. 7

```
class Foo
{
    private int x = 0
    private int y = 0
    invariant y = 2 × x
    public void f()
    {
        x += 1
        y += 2
    }
    public void g()
    {
        x *= 3
        y *= 3
    }
}
```

It has an invariant — you all know what an invariant is, of course — anyway, this invariant says that *y* is always twice *x*. The initial values,  $x = y = 0$ , respect the invariant. Both functions, *f* and *g*, maintain the invariant. Everything’s fine. 8

Now let’s give it a variable *b* and call *b*’s function *h*. Note that this does not change the (so-called) interface of *Foo*, because the new variable is private and the call is an implementation secret of *Foo.f*. 9

```
class Foo
{
    private int x = 0
```

```

private int y = 0
private Bar b
invariant y = 2 * x
public void f()
{
    x += 1
    b.h()
    y += 2
}
public void g()
{
    x *= 3
    y *= 3
}
}

```

Now look at class *Bar*. It's method *h* calls *Foo*'s function *g*.

10

```

class Bar
{
    private Foo f
    public void h()
    {
        f.g()
    }
}

```

This has unexpected consequences for *Foo*. Think of the interface as being the top of the box. The damage is done by the thread of control coming out of the **bottom** of the box — nothing to do with the “interface”.

11

Here is the sequence of assignments that take place and the resulting values:

12

$x = 0$	0
$y = 0$	0
$x += 1$	1
$x *= 3$	3
$y *= 3$	0
$y += 2$	2

The invariant now says  $2 = 2 \times 3$ . Oops! The call *b.h()* in class *Foo* is an important part of its behaviour — and therefore must be considered as part of its interface.

Note how *Foo* has lost control of its “private” internals. As a technical aside, note also that if *Foo* was a monitor allowing only one method to be executed at a time, this program would deadlock.

I've presented this as an example of the meaning of interface. But you should note that it is really quite a serious problem: objects have no control over their private data. This is a consequence of the fact that **objects can invoke functions in other objects**.

Apart from the “calling out the back” problem, this example shows that an important property of objects is the *allowable sequences of calls*, a property that is not handled by the specification techniques that I know of.

**Inheritance** Inheritance is supposed to be a key feature of the object model. But have you ever seen a precise and concise definition of inheritance? It’s a bit like the fable of the blind people and the elephant: a bunch of blind people feel up an elephant; each comes up with a different description of the elephant, depending on which part of it they touch, but none captures the “essence” of elephant. (This fable seems to have originated in India and has been claimed by Buddhists, Jains, and Sufis.) Actually, the analogy between inheritance and elephants is not that good: you would have to rerun the fable of the blind people and the elephant, but without the elephant.

**Behaviour and Aspect** We are told that the important thing about objects is their *behaviour*. But objects in the real world have many behaviours, not just one. Examples:

13

serves various <i>clients</i>	state <i>queries</i>
offers various <i>services</i>	metastate <i>queries</i> (reflection)
<i>logs</i> activities	usage <i>statistics</i>
<i>state control</i> : start/stop/suspend/resume/...	<i>testing</i>

OOPs do not support multiple roles well, or at all, although there have been some attempts to add “views” and other such devices.

You can call the roles “aspects” and then turn to *aspect oriented programming* is supposed to handle them. Does AOP succeed? Up to a point, but AOP is not modular — by definition, it’s “cross-cutting” — and makes restructuring difficult. AOP recognizes a significant problem, but just layers the solution over the problem. AOP will help for a while but, ultimately, it’s just a better telescope.

**Refactoring** For a long time, the big word was “reuse”. Now it’s “refactoring”.

Refactoring-in-the-small is easy: it’s mainly a matter of tidying up a program to make it more efficient, more maintainable, or more something else. Refactoring-in-the-large is more difficult. Typically, it means modifying a program to provide the same functionality on different hardware, different platforms, or with a radically different structure. Moving the smarts from server to client — or the other way round — it’s something that many companies would like to do, but find it’s infeasible. A complete rewrite would be cheaper.

Logically, much refactoring consists of moving the code around. It used to execute here, now it executes there. Physically, refactoring is much harder than it should be. This is because low-level implementation details are implicitly embedded in the code we write.

One of the goals of our work is to create *malleable software*: programs that can be rebuilt for different configurations without rewriting a line of code.

## 4 History

Sir Isaac Newton recognized that his accomplishments depended on previous work:

14

If I have seen farther than others, it is because I was standing on the shoulder of giants. — Isaac Newton

Richard Fateman has pointed out that computer scientists do not rely on previous work. In fact, they often seem to be completely unaware of it.

We should be standing on the shoulders, not the feet of those who worked on these problems. — Richard Fateman

Let's have another bash at Java.

When James Gosling (beard and glasses) introduced Java in 1994, he had a chance to stand on the shoulders of giants and create a really good programming language.

15

Who are “giants” Gosling might have stood upon? I would include:

- Edsger Wybe Dijkstra, beard and glasses, Dutch
- Charles Anthony Richard (Tony) Hoare, beard and glasses, English
- Per Brinch Hansen, glasses, no beard, Danish

16

17

18

To see why, we need a bit of history.

During the earliest days of computation, it was hard enough to get a program to run at all, and computers executed one program at a time — when they weren't breaking down. But it wasn't long before computers grew larger — physically if not mentally — and became more expensive. Keeping them waiting for the next program became economically undesirable.

Many people tried to design software so that more than one program could run at a time. Dijkstra, Brinch Hansen, and Hoare were the first to address the problem in a scientific way. Their work and its products — semaphores, monitors, communicating processes, and so on — made a considerable impact during the sixties and seventies.

19

We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. — Dijkstra

Another giant of that time was Michael Jackson — oops! wrong one, here is Michael Jackson the software engineer (no beard, no glasses). Jackson pointed out that *process decomposition* was often much more helpful and intuitive than *procedure oriented* design.

20

21

And now I would like to boast about the way in which my career fitted into this background. Fired up by Hoare's work, I did my masters thesis on implementing CSP. What was the level of interest in the industry? Nothing, nada, zilch. So I proceeded to do my doctoral thesis on the next academic fad — functional programming. What was the level of interest in the industry? Nothing, nada, zilch. Having blown it twice, I had no choice but an academic career.

What happened next?

First, all the good work disappeared into the guts of operating systems: THE, Multics, OS/360, George, Exec 8, Kronos, UNIX, VMS, and, eventually, Windows. Concurrency largely disappeared from mainstream programming. This is not to say that it doesn't exist: most serious applications today are *multi-threaded*, but they rely on seat-of-the-pants programming and not on the carefully developed theory of the sixties and seventies. And, of course, OS programmers still know how to do it.

Second, instead of concurrency, we got modules, abstract data types, and then object oriented programming. In Bertrand Meyer's words: 22

Born in the ice-blue waters of the festooned Norwegian coast; amplified (by an aberration of world currents, for which marine geographers have yet to find a suitable explanation) along the much grayer range of the Californian Pacific; viewed by some as a typhoon, by some a tsunami, and by some as a storm in a teacup — a tidal wave is hitting the shores of the computing world.

The origin of this unexpectedly purple prose was Simula, the language developed by Norwegians Kristen Nygaard and Ole-Johan Dahl in the mid-sixties. Simula was an object-oriented extension of a procedural language (Algol-60) and became the ancestor of *hybrid languages*, which include Object Pascal and Python but whose best-known exemplar is perhaps C++. Simula lives on in the form of Beta, developed in Denmark. 23

One of the many people attracted by Simula was Alan Kay. Kay believed that all programming languages aspire to LISP and his language, Smalltalk, was to objects what LISP is to functions. Adele Goldberg — no beard, no glasses — worked on the implementation of Smalltalk and helped it to grow from a Xerox research project into a sizable company. 24  
Goldberg was the manager of the lab at which the famous “incident” took place: the two Steves — Jobs and Wozniak — visited, saw the demos, did a bit of reverse engineering, and came out with the Apple Lisa (and later Macintosh). Smalltalk became the ancestor of *pure* OOPs, which include Eiffel, Ruby, and, of course, Java, designed by Gosling. 25

Here's an important contributor to OOP theory — Barbara Liskov, the first woman in the United States to receive a Ph.D. from a computer science department (Stanford 1965). 26  
Liskov defined the concept of *behavioural subtyping* which is now an important part of OO design methodology. She is currently working on a fault-tolerant, distributed, concurrent OOP called Argus. 27

The Best Part of Being an Engineer: I find a career in engineering to be very satisfying. I like making things work. I also like finding solutions to problems

that are both practical and elegant. And, I like working with a team of people; engineering involves lots of team work. I particularly like working with my students on our research projects. — Barbara Liskov

None of these OOPLS provide true concurrency (except Liskov's Argus). None of them makes use of the technology developed 30 to 40 years ago. At best, they provide thread libraries.

28

*It is astounding to me that Java's insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.*

Although the development of parallel languages began around 1972, it did not stop there. Today we have three major communication paradigms: monitors, remote procedures, and message passing. Any one of them would have been a vast improvement over Java's insecure variant of shared classes. As it is, Java ignores the last twenty-five years of research in parallel languages. — Per Brinch Hansen

Of course, Java has an important advantage over C++: Java is compiled to byte codes that run on the JVM, and the JVM can at least ensure the integrity of the instructions. C++ is normally compiled to machine code, and is at the mercy of the chip architect. C++ itself has no concurrency and relies on libraries such as Posix threads (p-threads). But it can be quite tricky to get pthreads to work properly on architectures with pipelines, speculative execution, multiple processors, and all the fancy features of modern CPUs. And that situation will only get worse as the hardware guys up the ante.

The mainstream response (i.e., UML-2) seems to be this: let us rise above the messy problems of programming languages; we will keep our hands clean in the pristine purity of abstraction.

But even the Rational guys admit there's a problem:

29

You can blame some of this model inaccuracy on the extremely detailed and sensitive nature of *current programming language technologies*. Minor lapses and barely detectable coding errors, such as misaligned pointers or uninitialized variables, can have enormous consequences. . . . If such seemingly minute detail can have such dire consequences, how can we trust models to be accurate, since models, by definition, are supposed to hide or remove detail? — Bran Selic (emphasis added)

The buzz nowadays is Model-Driven Design. I'm not opposed to that — I approve of attempts to raise the level of abstraction, although I find that many programmers do not agree. What I find interesting about MDD is that the models are based on data communication rather than the object model. But the models are built on top of implementation languages do not provide proper support for processes. That's a serious mismatch. A mismatch that we think can be fixed.

## 5 Our Work

Time now for a brief look at our work.

We — Brian and I — think the time has come to return to concurrency. We are not the only ones who think this. Recent work in concurrency includes:

30

- Joyce (Per Brinch Hansen)

Not terribly recent — 1989 — but some useful lessons for today. Joyce provides an important existence proof: it can be done.

- Hermes (IBM)

Hermes was an experimental systems PL developed by IBM based on capabilities. Communication between Java threads requires at least 10,000 instructions; Hermes requires 9.

Hermes was designed for OS programming. Most languages assume that the OS provides barriers that prevent your program interfering with other programs and *vice versa*. Hermes assumes that it's running on the metal and there are no such safeguards.

- Oz (Seif Haridi and Nils Franzén)

An amazing language/system that supports multiple paradigms — procedural, functional, logical, and concurrent — with elegance and efficiency. Millions of concurrent processes can run on an ordinary PC. <http://www.mozart-oz.org/> or Google on “Mozart Oz”.

- *occam- $\pi$*  (Fred Barnes *et al.*)

*occam- $\pi$*  is the practical realization of Hoare's formal model from 1978, Concurrent Sequential Processes. The minimum size of a Java thread is 32 kilobytes. The smallest *occam- $\pi$*  process uses 16 bytes.

- Separation Logic (John C. Reynolds *et al.*)

This seems to be the formalism that we need to reason about concurrent programming. I am trying to understand it.

Why do we emphasize concurrent processes?

- modern CPUs
- sequential coding is no longer good enough
- distributed applications
- processes are more natural than procedures

Our model is based on big ideas and small ideas ideas. Here is a short selection.

31

1. A component is a *cell* running a multithreaded process
2. Cells exchange data
3. A cell gets exactly the capabilities that it needs
4. Semantics is decoupled from deployment
5. Programs are scale-free
6. Tests are part of the code

**Cells** Here is a cell. The resemblance to a biological cell is deliberate. 32

The cell runs a process. The process may have several threads. Here we indicate the threads by circles; the straight lines indicate switching from one thread to another. 33

Here is another cell with its own process and threads. 34

The cells communicate (dashed line) using ports (small circles). Only data can be communicated: a cell cannot directly invoke a method in another cell. 35

**Capabilities** The second idea is to provide each cell with the things that it needs to do its job when it is initialized. The cell cannot ask for anything else.

Here's a simple example. If you write this in a Java program, it doesn't work. 36

```
public static void main(String[] args)
{
    ....
    Random generator = new Random();
    ....
}
```

To make it work, you have to add an `import` clause. 37

```
import java.util.Random;

public static void main(String[] args)
{
    ....
    Random generator = new Random();
    ....
}
```

In general, if you want to use a Java feature, you import the corresponding package, or part of a package, or whatever. Like Hermes, our system does not allow this: each cell is given what it needs, and cannot ask for more by `including`, `importing`, or any other trick.

**Decoupling semantics and deployment** The next idea is to decouple semantics and deployment. Here is the conventional model: we start with the syntax of a program, infer its semantics, and use the semantics to construct the implementation. 38

Here is the new model. Again, we start with syntax and infer semantics. Additionally, we choose an implementation — I prefer the term *deployment* — to obtain the program we need. 39

This example illustrates the general idea. We could do it in principle, although it is a bit far-fetched. 40

```
income := salary() - fed.tax() - prov.tax()
```

The *semantics* of this assignment statement are clear and conventional: the expression on the right has a value, and that value is assigned to the variable *income*. This meaning is quite independent of the *deployment* of the statement.

This statement might be a purely local computation on a single processor. At the other extreme, it might request *salary()* to be computed at the employer's site, *fed.tax()* to be computed in Ottawa, and *prov.tax()* to be computed in Quebec, with the result stored on the employee's computer.

(For non-Canadians, we have this weird tax system where we pay all kinds of different taxes to various levels of government. This would be more acceptable if the various levels of government ever did anything for us.)

**Scale-Free Software** Scale-free is a current buzzword, linked with fractals, power laws, chaos, and other exciting things. It means, roughly, “looks the same at all sizes”. Here's a photograph of sand. When I show this, I get comments such as “Where's the Lone Ranger?” 41 But in fact the area covered by that picture is about  $4 \times 3$  inches. You can't tell the scale because, to a first approximation at least, sand looks the same over a wide range of scales.

Most current languages have a particular syntax for each scale: variables, expressions, functions, classes, packages, . . . and there it stops. It is curious that one of the earliest languages, Algol60, was scale-free: nested Algol blocks can be built up to any level. Unfortunately, Algol's scale-free-ness was of the wrong kind: it produced great big monolithic programs that could not be split into modules, separately compiled, etc.

Designing a scale-free language seems hard. But I think it's important to give it a try, because the alternative is to live with a fixed hierarchy in which the largest unit is never quite big enough.

**Tests in the code** Testing is important. Several of the speakers yesterday said that in various ways and Cen Kamer gave a whole talk about it. Documentation is important, too, and some people say that the only documentation that ever gets read is the comments that are part of the code itself. We think that tests should be part of the code as well. Each function and each module comes with tests that can be run by selecting appropriate compiler options. Then there's at least a chance that programmers will test their code.

**Examples** What do our programs look like? Well, we haven't got very far yet. Our notation still looks a bit klunky, but most notations do in their early stages.

Here is a process that computes the sum of two numbers.

42

```

abstract type MessageProtocol = [];
type SumProtocol = MessageProtocol[p, q: int; result: *int];

type Sum = [
  main: process(init: *SumProtocol) =
    var params: SumProtocol;
    init ? params;
    params.result !move params.p + params.q
  end
]

```

To employ the service provided by Type Sum, we carry out the following steps.

43

```

var initProtocol: **SumProtocol;
new Sum({}) !alias initProtocol;
var sumFunction: *SumProtocol;
initProtocol ? sumFunction;

var return: *int;
sumFunction !copy (2, 3, alias return);
var x: int;
return ? x

```

This looks rather involved. We (that is, “the system”) have to create a cell, send it the capabilities that it's allowed to use, and then wait for it to send back its protocol (consisting of a port and allowable messages), here called `sumFunction`. Once all this is done, we can start to use the service that the cell provides.

Quite by chance, this program performs the exact task —  $2 + 3 = 5$  — that Cem Kaner discussed yesterday. Thank you, Cem.

Of course, no self-respecting programmer would accept all this verbiage. After introducing some abbreviations, we can write the sum process as

44

```

type Sum =
[
  main: process(p, q: int; result: *int) =
    result !move p + q
  end
]

```

and its invocation as

```

var return: *int;
Sum({}) !copy (2, 3, alias return);
var x: int;
return ? x

```

Give us another few months and we'll make it even shorter.

## 6 Conclusions

Why didn't these ideas work out the first time round — in the 1970s? The technology was not up to it. Computers were smaller, efficient implementations were unknown, expertise was limited. Now we are ready.

45

1. Object Oriented programming has had a good run, but even the best shows don't last forever
2. Concurrent programming, after lurking in the wings for decades, is moving to centre stage
3. One play, many shows  
Compare Java's "compile once, run anywhere".

Our project needs a name: suggestions are welcome!

What should you take away from this talk? I'm not suggesting that you get into wild research projects — at least until you have tenure. Here are some things that have helped me:

46

1. Learn the basic skills well  
Play saxophone like Chad Fowler, not Kenny Gee. When you have solid technique, you can begin to experiment. When you know the rules well, you can start to break them.
2. Question the foundations  
Take nothing for granted. Don't get into the mindset "computing = Java" (or whatever language). Choose the language that fits the task (if you're allowed to) and, if necessary, create a new language for the task.
3. Trust your intuition  
If you have a thorough knowledge of the tools of your trade, and something feels wrong, then it probably is wrong.

## List of Slides

1	Brian and Peter 2004 . . . . .	1
2	Brian and Peter 1984 . . . . .	1
3	Sabre users . . . . .	2
4	Java scope control . . . . .	3
5	Net-tuner — front view . . . . .	4
6	Net-tuner — rear view . . . . .	4
7	Class <i>Foo</i> . . . . .	4
8	Calling <i>Foo.f</i> . . . . .	4
9	<i>Foo.f</i> calls <i>Bar.g</i> . . . . .	4
10	Class <i>Bar</i> . . . . .	5
11	<i>Foo</i> is stabbed in the back . . . . .	5
12	Assignments break the invariant . . . . .	5
13	Behaviours of an object . . . . .	6
14	Newton and Fateman quotes . . . . .	7
15	James Gosling . . . . .	7
16	Edsger Dijkstra . . . . .	7
17	Tony Hoare . . . . .	7
18	Per Brinch Hansen . . . . .	7
19	Dijkstra quote . . . . .	7
20	Michael Jackson — the singer . . . . .	7
21	Michael Jackson — software engineer . . . . .	7
22	Meyer on OOP . . . . .	8
23	Kristen Nygaard and Ole-Johan Dahl . . . . .	8
24	Alan Kay . . . . .	8
25	Adele Goldberg . . . . .	8
26	Barbara Liskov . . . . .	8
27	Liskov quote . . . . .	8
28	Brinch Hansen quote . . . . .	9
29	Selic quote . . . . .	9
30	Recent work in concurrency . . . . .	10
31	Ideas . . . . .	10

32	A simple cell . . . . .	11
33	A cell with a multithreaded process . . . . .	11
34	Another cell with a multithreaded process . . . . .	11
35	Communicating cells . . . . .	11
36	Call to Random() in Java . . . . .	11
37	Call to Random() with import clause . . . . .	11
38	Syntax to semantics: the standard model . . . . .	12
39	Syntax to deployment . . . . .	12
40	Income is salary less tax . . . . .	12
41	Sand . . . . .	12
42	A process for adding . . . . .	13
43	Invoking the adder . . . . .	13
44	Acceptable abbreviations . . . . .	13
45	Conclusions — Our Project . . . . .	14
46	Conclusions — Lifestyle . . . . .	14