

Eiffel : clairs objets du désir

Click [here](#) for the slides (12 MB)

Bertrand Meyer, Inaugural Lecture
ETH Zürich, 18 November 2002



Ältere Wissenschaften und Ingenieurs-disziplinen haben, Herr Rektor, Mitglieder der Schul-leitung und Departements-leitung, Meine Damen und Herren, reichlich Zeit gehabt, ihre Wahrheiten zu sammeln. Der Major-General in den *Pirates of Penzance* ist stolz darauf,

Older scientific and engineering disciplines have had ample time to accumulate their collections of truths.

The Major-General in the Pirates of Penzance is proud to know a few from mathematics; he is:

einige aus der Mathematik zu kennen; er ist

*... teeming with a lot o' news —
With many cheerful facts about the square of the hypotenuse.*

(Musical extract)

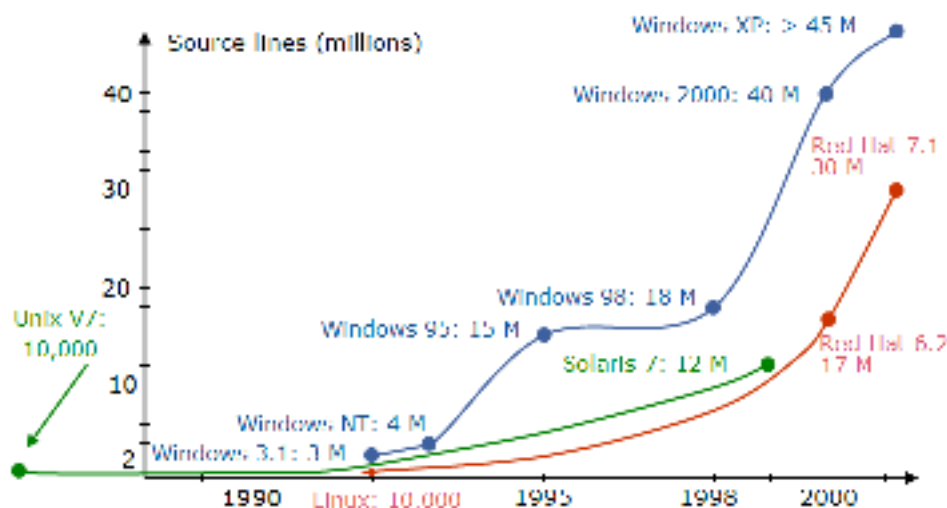
Computer science is younger, but already we are

*teeming with a bit o' Knuth
With many cheerful stacks, linked lists and double-ended queues*

We want to teem with more, in particular software *components* embodying knowledge about little reusable software machines. Before I come to this let me discuss some of what makes software unique.

1 Where software is going

ETH achieved world fame in software through insistence on elegantly engineered programs, which, in particular, remain small. My illustrious predecessor Niklaus Wirth and his colleagues have had on the industry an effect as profound as it is diverse, but *that* part of the message has not been heeded. Consider the evolution of Windows.



These are numbers of source lines of code for the program, mostly C and C++. In a decade, they went from three million to over forty-five.

This is not just Microsoft; Sun's Solaris system, a descendant of the

original Unix, and the open-source Linux, both started at about ten thousand lines but have also followed a steep path. One can see a similar progression with space, telephone, defense systems. Some banking systems are venturing into that territory.

It is hard to picture forty-five million lines of code. It's also hard to think of non-software products of comparable complexity, even sophisticated artefacts such as a Boeing 747 or a TGV.

The most relevant comparisons may be with human social and political systems, such as cities. But a city is not an engineering product; it can still function when lots of things go wrong. A cable can break on the Limmatquai and block trams 4 and 15, but Zürich won't come to a halt. The country's flagship airline may go down, and the country survives. Software is not at all like that. The smallest detail, say inverting two source lines, can cause havoc. Resetting one bit in a two hundred megabyte executable can stop it from working. This is like changing *one* letter in *one* book of *one* person in this room, if you each have a hundred books. On the other hand, we all suspect that someone could switch a few millions of bits in the binary of Microsoft Word and it would be a while before anyone noticed. That's one of the maddening things about software: it doesn't exhibit the continuity properties to which we are used in the physical world.

So our task is not just to handle complex problems but to handle them right. Indeed this is the definition of our field:

Software engineering

The task of building potentially large and complex systems so that they are correct, and can change.

Just two of the requirements, correct and complex, or correct and extendible, would be hard enough, but we must deal with all three.

We cannot just dismiss the second one by rejecting complexity. No doubt, as Wirth has so often argued, some of the complexity in big systems is self-inflicted, recalling the coat of Joseph K.'s investigator, which, Kafka writes,

infolgedessen, ohne daß man sich darüber klar wurde, wozu es dienen sollte, besonders praktisch erschien.

And as a consequence, although it wasn't clear what use they might have, they appeared especially practical.

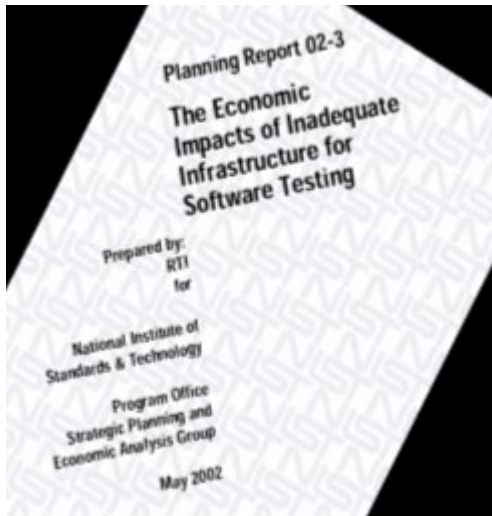
But some of the complexity is inherent. Windows or Linux must support compatibility with older versions, countless third-party devices, user interfaces in Swahili and Schwizertuutsch, daylight saving time rules for every country. That complexity will not go away; it is the duty of software engineering to provide ways to tame it.

At this exercise we are doing better than before, but still not well enough. Consider the so-called CHAOS reports by the Standish Group, starting in the early nineties and regularly updated, covering US software projects, for a total of a quarter trillion dollars. Just over one project in four is a success, almost twice as many as in ninety-four but still not much; about the same number fail *completely*, abandoned without producing a system. The rest are what the report calls "challenged" projects: they yield a result, but not what the users wanted, and often at a much higher cost than expected in both time and money.

We may draw some consolation from the improvements the report found from 1994 to 1998, especially — another argument for those who favor simplicity — for small projects; but even so the overall result still appear dismal.

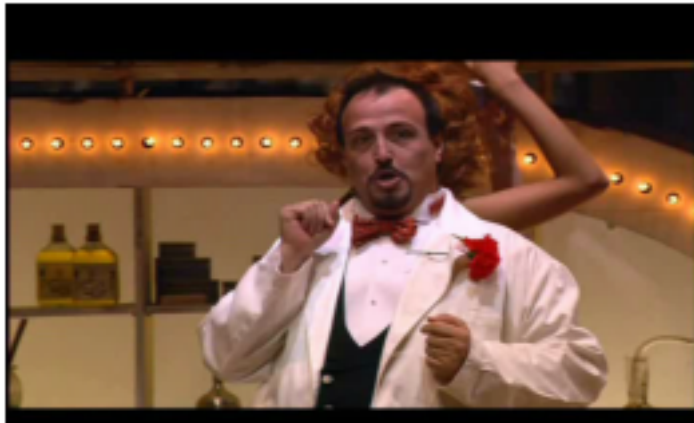
The report's authors, by the way, are good at problem analysis but seem rather clueless with respect to the solutions; the rules they suggest for success are all management-related, having to do with better communication with the users and management — good advice to be

sure, but charmingly ignoring that this is a technology field and that nothing will improve much without some technological remedies. It's as if in 1960, faced with a potential explosion of transatlantic traffic, the advice had been to manage ocean liners with better cooperation from travel agents. What we need in software is not better organized ships but jet planes.



An even more recent report from the National Institute of Standards in the US tells us more about quality. It claims to be about testing, but really describes a more general problem: the unreliability of software systems. We're all used to programs that don't work, but here is a quantified estimate: "insufficient testing" as the report calls it, really meaning bugs, costs 60 billion dollars annually. For reference, this is almost twice the budget of the Swiss state, before ETH budget cuts. Even with some skepticism as to the accuracy of such sweeping statistics, that's a lot of money, and of bugs.

Now it is not my intention to pontificate about how bad software is and how stupid programmers are. First, things have improved at least a little, as even the Chaos reports show. But also, like anyone who doesn't just write about software but actually writes software, I know how hard it is. I sure can boast about the price reporting system, written in Eiffel, that runs at the Chicago Board of Trade. But after all, it's a quarter to eleven in Chicago now, trading is at its peak, and I have no way to promise you, or myself, that the Eiffel garbage collector won't fail in the next minute, forcing the exchange to close. That kind of Angst is enough to keep one modest.



Not everyone is modest, and our field has no lack of supposedly perfect cures, recalling Doctor Dulcamara in the *Elixir of Love*.

(Musical extract)

2 The power of object modeling

I have my own Elixir. It's called Eiffel, an object-oriented method for developing quality software. Let me try to give you a taste of it.

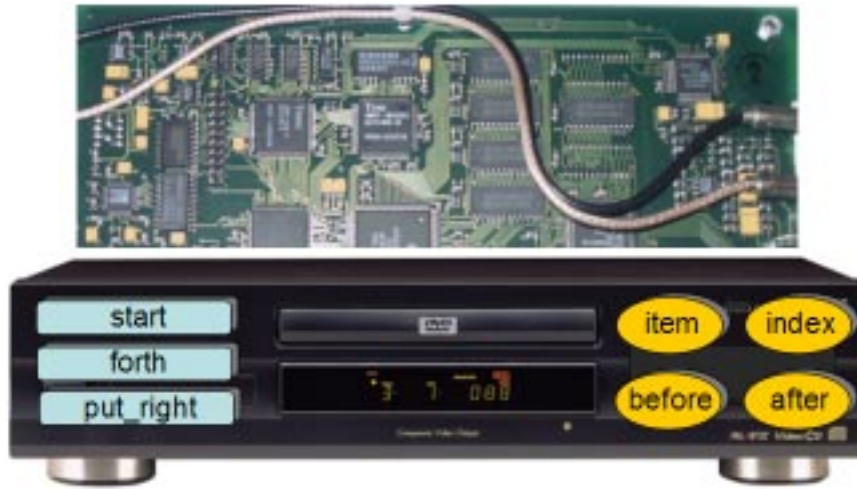
In object-oriented development a running program is made of software machines called **objects**. This is a pretty good name but also dangerous as it suggests that every software object represents a physical object from the world out there; only some of them do. You must understand "object" to mean just what I said, a software machine.



Like any machine, an object offers functions; picture it with some buttons that you can press for these functions.

Any machine has an *interface* and an *implementation*:

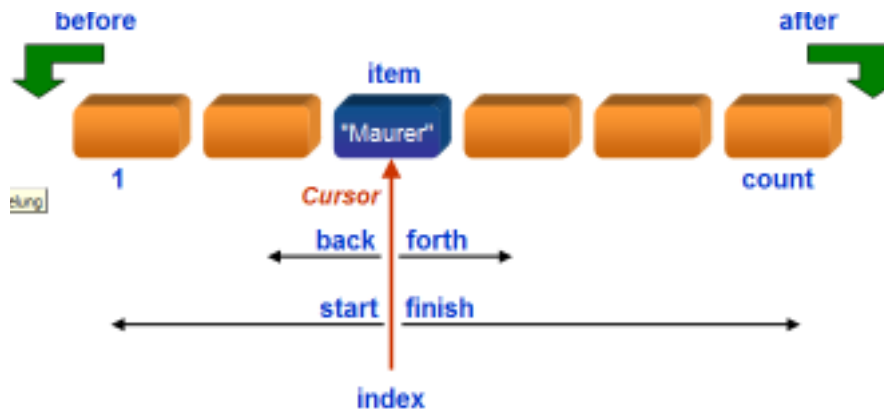
- To its users, which we will call its **clients**, the machine presents an interface, the set of available operations with official properties which we'll call **contracts**.
- To carry out the operations the machine has an implementation, which may be simple or complex but is none of the client's business; in fact if you try to open it you'll void the warranty.



We call this **information hiding**. The reason it's so important in software is our obsession with change. The clients of our objects are not people, but other objects. If we let them rely on internal properties of their suppliers, we preclude smooth evolution: a change somewhere in the system can cause an effect anywhere else, so that

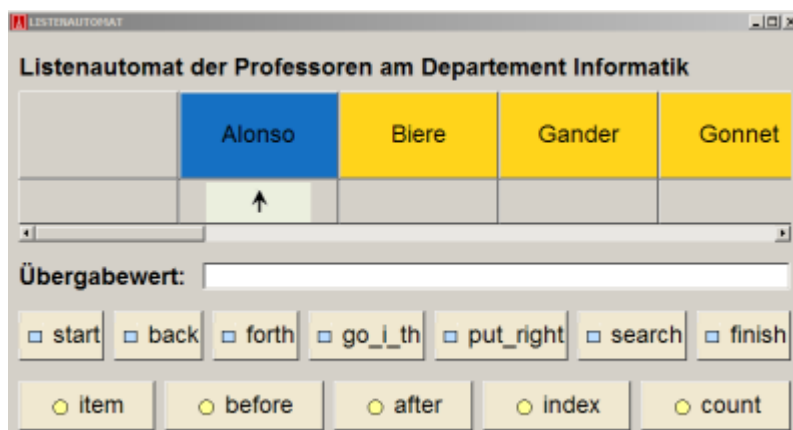
redoing anything often means redoing everything. That's why Eiffel is draconian about information hiding. Some other approaches are more tolerant, but it's not a way to be nice to developers and their customers. A little discipline makes all the difference; three years later when someone is desperately trying to make a change, it can be the difference between a few hours' work and a whole new expensive project. It pays to be strict.

Let me show you an object.



It's a *list* object, the kind that turns on any true software engineer. It contains other objects, representing professors. The operations refer to a current position, or **cursor**. Some move the cursor: *forth* advances it by one position, *start* takes it to the first position.

We can play with the list machine to insert a new item. Let's see this live.



*Live demo.
Only a few
screenshots
are shown.*

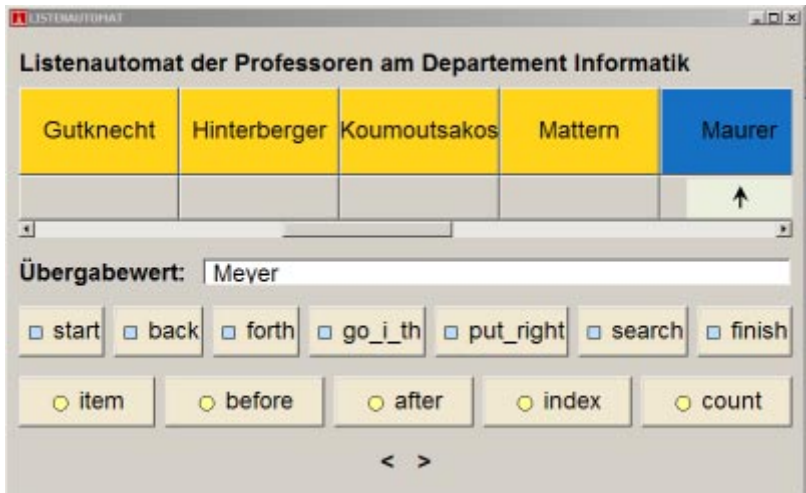
The cursor is on the first element. We push *forth* a few times; at any step pushing *item* gives us the item at cursor position.

The square buttons, like *start* and *forth*, are command buttons; a command changes the object. The round buttons, like *item*, represent queries; a query returns a result, here the value of the item, but doesn't change anything.

This is one of the principles of the Eiffel method:

Command-query separation principle
Asking a question shouldn't change the answer.

It's one of those design rules that can help preserve the sanity of complex systems, not to mention the sanity of people maintaining them.

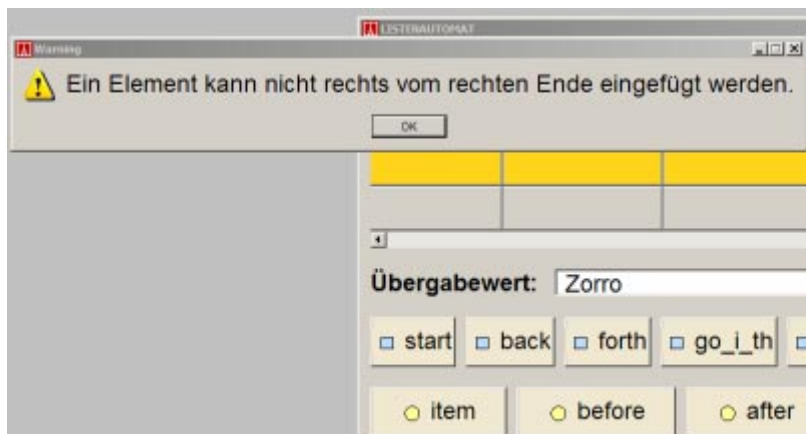


I use *forth* a few times to advance to the entry labeled *Maurer*. I'll use the command *put_right* to insert a new item to the right of the cursor. Type it in.



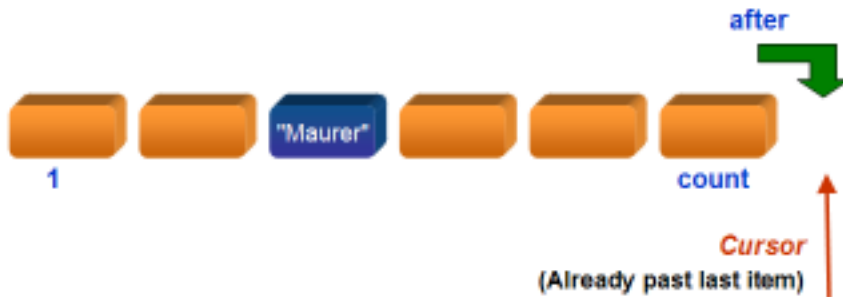
That's it.

Now let's try to do this after the end: *finish* to go to the end, then *forth*, then *put_right*.

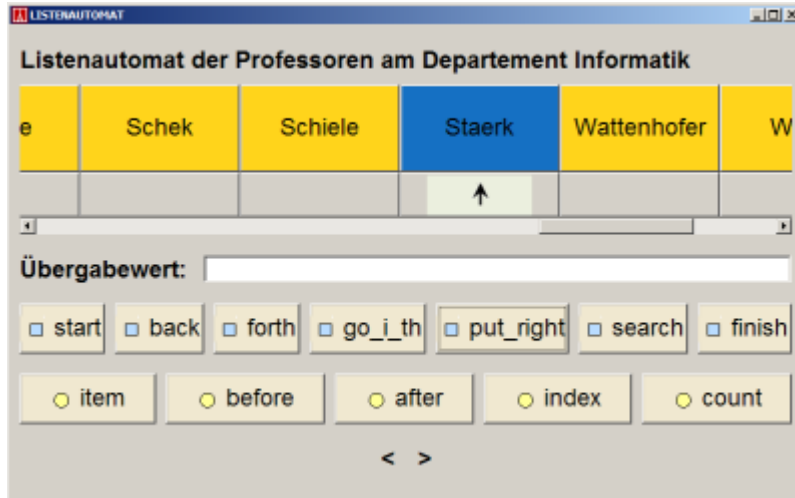


(Voices: "Zürück!", from the Magic Flute)

Oops! I have violated a contract.



The mistake is clear: we may not insert an element to the right of nothing!



Just back up three positions, using *back* to where we can insert my colleague Roger Wattenhofer in advance of his own inaugural lecture in January.

End of first live demo

What I have shown you so far is not Eiffel software, no software at all, it's only software execution. Let's take a peek at the underlying software. We'll examine some of the *classes* that make up the example system that I have been running in this example.

```

Editor
deferred class LIST [G] inherit

    CHAIN [G]
        export
            {ANY} remove
        redefine
            forth, is_equal
        end

    feature -- Comparison

        is_equal (other: like Current): BOOLEAN is
            -- Does 'other' contain the same elements?

```

A class describes a type of object. Here is the class *LIST* from the EiffelBase library, shown in a screenshot from EiffelStudio, the development environment.

It's declared as *LIST [G]*, list of *G*, where *G* denotes the type of objects in the list — professors in our last execution.

The class doesn't give us an object but a *kind* of object, a type; if the object is a machine, the class is a machine tool. We software engineers have an advantage over our colleagues who deal with more material devices: once we have figured out the machine tool, we can use it at run time to get as many machines as we like, just for the price of asking.

```

before: BOOLEAN is
  -- Is there no valid cursor position to the left
do
  Result := (index = 0)
end

feature -- Cursor movement

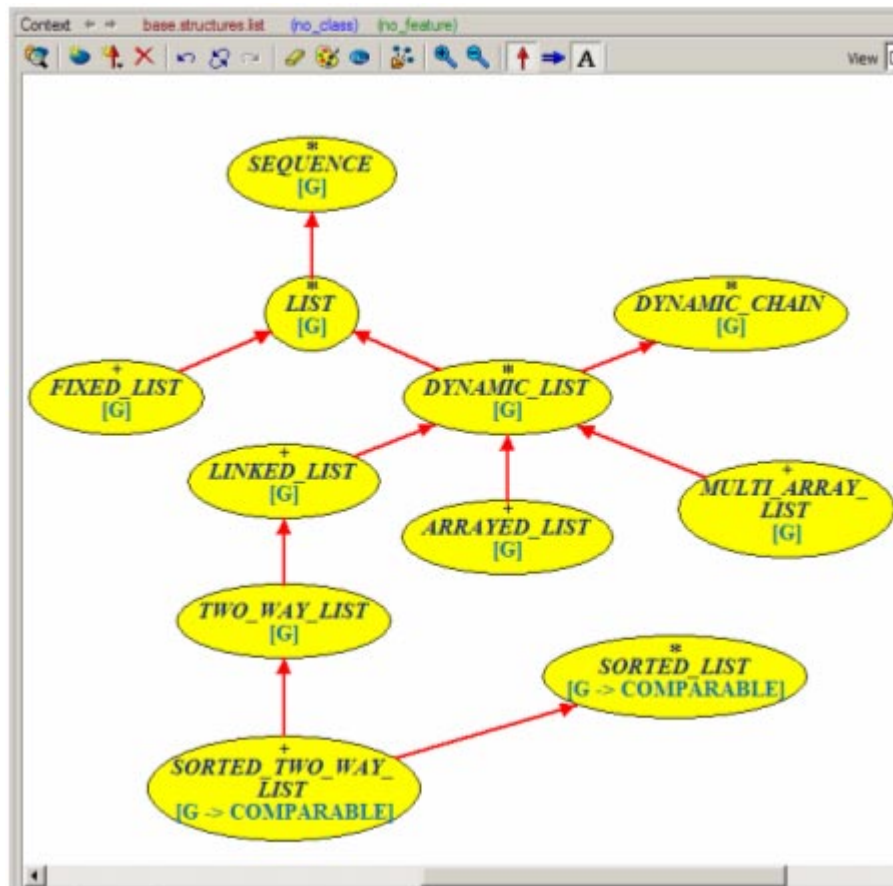
  forth is
    -- Move to next position; if no next position,
    -- ensure that 'exhausted' will be true.
  deferred
  ensure then
    moved_forth: index = old index + 1
  end

```

The class text expressed in Eiffel describes the operations applicable to any of the corresponding objects. Here for example are the query *before* and the command *forth*.

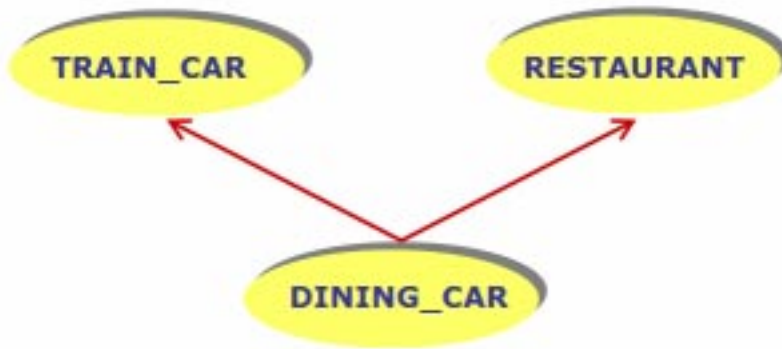
There's no implementation of *forth*; it says **deferred**. That's because the class doesn't describe a particular kind of list, only the abstract notion.

To see its place in the overall order of things, we look up the inheritance structure in the EiffelStudio Diagram Tool.



Red arrows say “inherits”, meaning specializes. *LINKED_LIST* and *ARRAYED_LIST* specialize the notion of *LIST*, providing their own variants of operations like *forth*; EiffelStudio lets you track these variants easily.

Inheritance is critical to organize classes into taxonomies. Because of the incredible power that software gives us to build any machine we can dream up, just for the cost of describing it, we could quickly engulf ourselves in an ocean of classes. Inheritance gives us classification, without which there's no scientific discourse.



The complexity of the phenomena we have to describe can't accommodate a tree structure, so we need **multiple** inheritance, the ability for a class to have more than one parent — to specialize more than one abstraction. A dynamic list is a list and a dynamic chain, just as a dining car is a train car and a restaurant, or a

topological vector space is a vector space and a topological space.

Single or multiple, inheritance can be misused like anything else, but it's one of our primary intellectual tools for coping with complexity.

Next, contracts.



Remember how we couldn't insert at a meaningless place in the list.

Such constraints should be specified. To ensure the correctness of our classes and operations we must carefully state their abstract properties, or contracts.

```

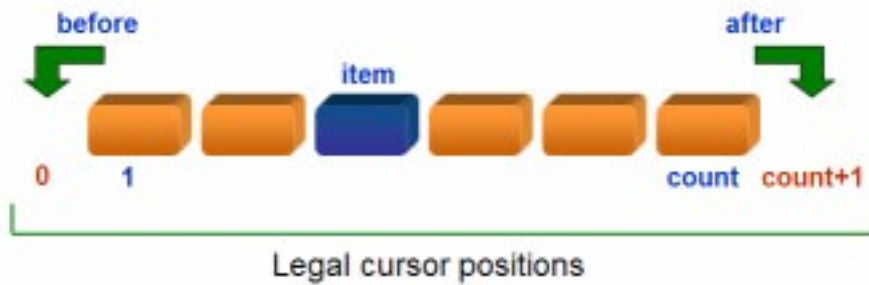
put_right (v: like item) is
  -- Add 'v' to the right of cursor position.
  -- Do not move cursor.
  require
    extendible: extendible
    not_after: not after
  do
    p := new_cell (v)
    count := count + 1
  ensure
    new_count: count = old count + 1
    same_index: index = old index
  end

```

The contract of a command such as *put_right* consists of a precondition, marked **require**, and a postcondition, **ensure**.

The precondition says we must not be *after* the last element of the list. The postcondition says that there is one more element, and the cursor hasn't moved.

There's also, for the class as a whole, a **class invariant** to describe global properties of the corresponding objects.



For example, in our picture of a list object, we wanted to let the cursor go one position to the left of the first element, one position to the right of the last element, but no further.

```

invariant

  prunable: prunable
  before_definition: before = (index = 0)
  after_definition: after = (index = count + 1)
  -- from CHAIN
  non_negative_index: index >= 0
  index_small_enough: index <= count + 1

```

Legal cursor positions

This is stated explicitly by a clause in the class invariant: the index of the cursor, *index*, must remain between zero and *count* — the number of elements — plus one.

Like business contracts between companies, software contracts define mutual obligations and benefits between the various elements of a system, clients and suppliers. I cannot emphasize enough how important this notion is to the quality of the software:

- Contracts help us get the software right by worrying about correctness initially, not as an afterthought.
- They profoundly affect *testing* by defining clearly what to test for; in my view no serious testing, of the kind likely to make a dent into the 60 billion dollars of the NIST report, can ignore these techniques. Our group is working on test case generation through contracts.
- Contracts are a *management* tool, enabling managers to track and control the development from a high-level view.
- They're also an excellent technique for *documenting* software, especially reusable software like the list classes we saw, meant for use by many programs. If you write a program using *LINKED_LIST*s, the documentation you'll see for the class is its contract form, retaining the preconditions, postconditions and invariants of the class, a form of the documentation that is both high-level and precise, and is extracted automatically from the text by EiffelStudio, showing only interface information.

Here we are seeing it in HTML in a browser:

indexing

```

description: "Sequential, one-way linked lists"
status: "See notice at end of class"
names: linked_list, sequence
representation: linked
access: index, cursor, membership
contents: generic
date: "$Date: 2002/01/06 04:03:48 $"
revision: "$Revision: 1.23 $"

```

class interface

```
LINKED_LIST [G]
```

create

```

make
    -- Create an empty list.
ensure
    is_before: before

```

feature -- Initialization

```

make
    -- Create an empty list.
ensure
    is_before: before

```

feature -- Access

```

cursor: CURSOR
    -- Current cursor position

first: like item
    -- Item at first position
require -- from CHAIN
    not_empty: not is_empty

```



If you remember our earlier picture of the machine with its interface and its implementation, this means that you get the interface automatically.

It's one of many *views* you can get, in line with an important principle of the Eiffel method, the Single Model principle:

Single Model principle

All the information about a system should be found in the system's text.

This rejects the more common approach of using multiple models, such as an overall picture in a graphical notation like UML, then a program text, then documentation. Here we elevate program text to a much higher status; we don't hide the program, we don't call it "*code*", we try to make it *clear* enough, *simple* enough, *general* enough, *beautiful* enough to be considered our most cherished product.

The key word is **model**. A great insight of Kristen Nygaard, co-inventor of object-oriented programming, was that programming isn't just for talking to computers, it's for talking to other people about systems and their models.

"To Program Is To Understand"

Kristen Nygaard

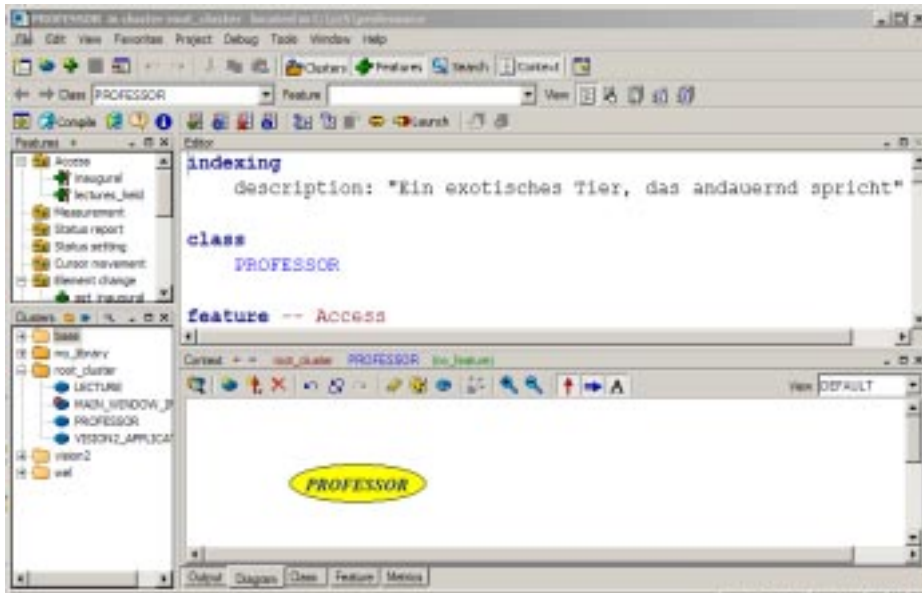
A program is an **operational model** of a physical or virtual system. Object technology provides powerful modeling techniques; we've already seen a few:

- The *object*, a machine equipped with operations (commands and queries) and contracts.
- The *class*, describing a set of objects with the same operations and contracts; from a class the execution can create as many objects as needed.
- *Inheritance*, letting us organize classes into cogent taxonomies.
- We've seen how a class can be a *client* of another, using its operations through an official interface and information hiding.
- We've seen how to use *contracts* to specify the precise semantics of classes and their operations.
- We even saw *generic* classes, like *LIST [G]* where *G* represents any possible type, so that we can use the same class for lists of integers and lists of professors.

These object-oriented concepts, as Nygaard had foreseen, are really about modeling systems of any kind. Many are original; they will increasingly be recognized as a contribution of software engineering, far beyond computers and programming, to ontology, the art of describing things, and to the scientific method in general. It's not just that art imitates life, but that the art of describing art turns into an art of modeling life.

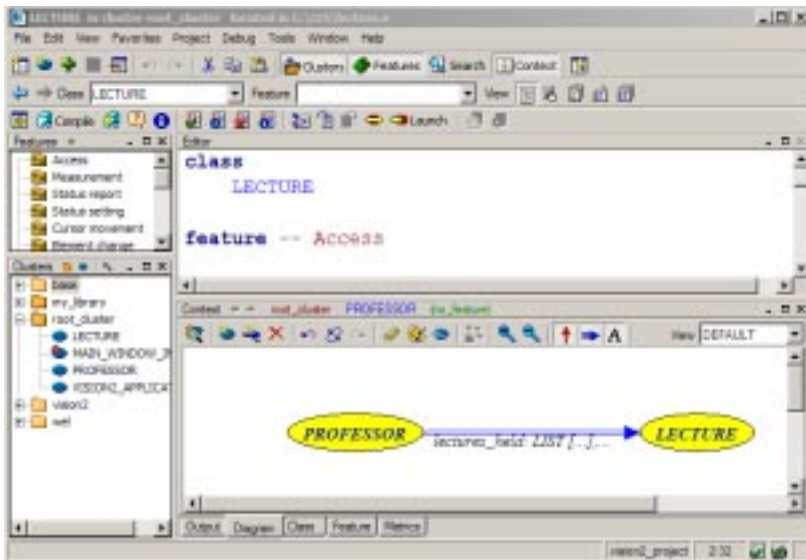
Let's indeed model some aspects of *academic* life, using Eiffel.

Start of second live demo. Only some screenshots are shown.

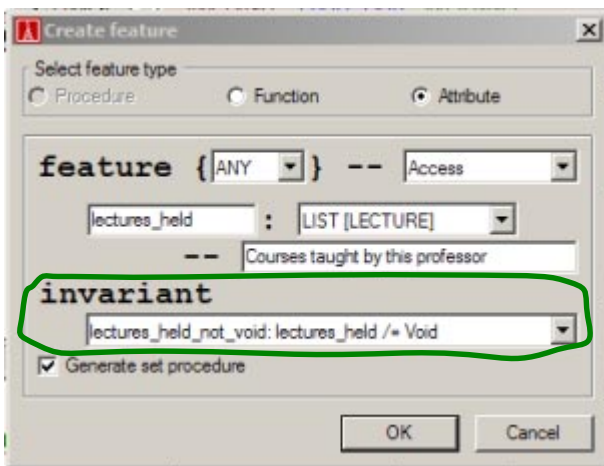


With remarkable lack of imagination I use the notion of *PROFESSOR*, modeling it with a class. We use EiffelStudio's graphical tools; ellipses are classes.

There's a notion of lecture, another class, another ellipse.



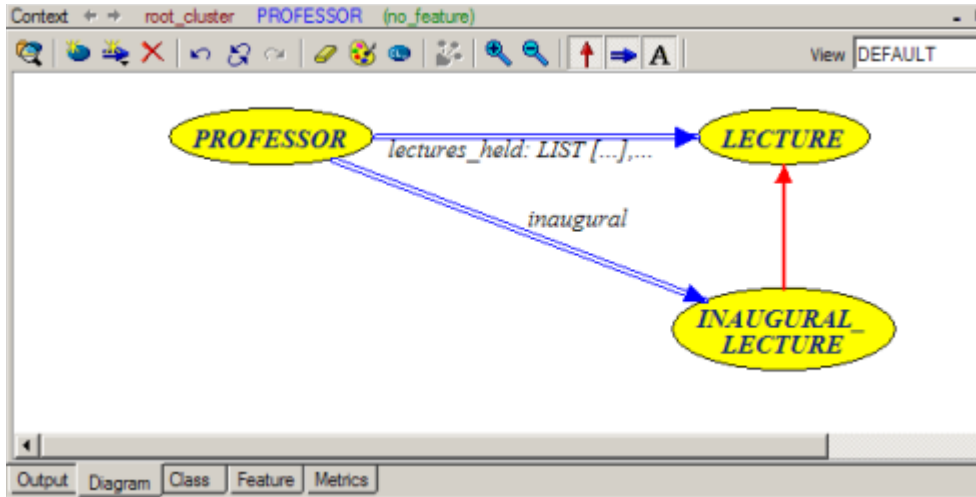
A professor holds lectures; we say that class *PROFESSOR* is a client of *LECTURE*, drawing a client link between the two.



This asks us to enter a query name, *lectures_held*, to denote a *LIST* of lectures.

We specify that the list must exist, even if empty, with an invariant requiring *lectures_held* to be non-void.

A useful principle is "Compile early and often"; I compile now. The compiler is our friend, here to check that we don't embarrass ourselves by stating two incompatible properties. Our system compiles fine.



Some lectures are inaugural; I add a class *INAUGURAL_LECTURE*, and make it inherit from *LECTURE*, using graphical input as before except that now we get a single red line,

representing inheritance, rather than the double blue line representing client.

These two relations, *client* and *inheritance*, are the only possible ones between classes; simplicity of the modeling concepts is essential if they are to help us reason about systems, not become objects of study on their own. We go back to class *PROFESSOR* and make it a client of *INAUGURAL_LECTURE* through a feature we call *inaugural*, and include the invariant stating that there must be an inaugural lecture.

But here we stop. Consider *INAUGURAL_LECTURE*; is this a separate abstraction, with its own operations and contracts? No. I succumbed to *taxomania*:

Taxomania: the inheritance disease

- **Symptom:** Patients see inheritance links everywhere, whether or not they reflect proper abstractions
- **Cure:** Study software engineering at ETH

Inheritance is a beautiful mechanism but not always applicable. Fortunately, as you can see, medicine now has a cure. I remove class *INAUGURAL_LECTURE*, and recompile.

```

class
  PROFESSOR

feature -- Access
  inaugural: INAUGURAL_LECTURE
  lectures_held: LIST [LECTURE]

Error: type is based on unknown class.
What to do: use an identifier that is the name of a class in the universe.
Class: PROFESSOR
Unknown class name: INAUGURAL_LECTURE
  
```

Oops! The compiler spots that *PROFESSOR* cites a class that doesn't exist, *INAUGURAL_LECTURE*. *inaugural* should just denote a *LECTURE*.

We fix this, recompile and get the compiler's imprimatur.

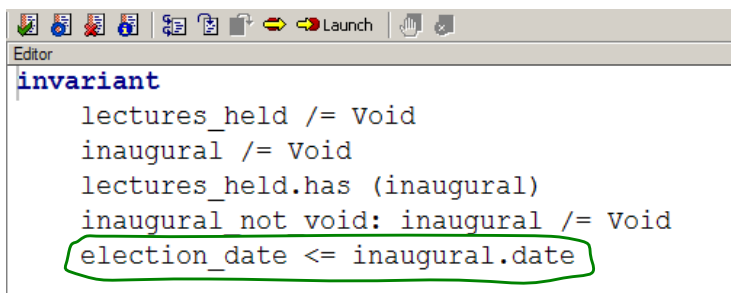
The compiler is our guardian angel, always available to check the consistency of what we write. Of course compilers should also produce machine code, but so far we have no code, and use the compiler as an invaluable tool for modeling our system.

Let's do just a bit more modeling.

```
invariant
  lectures_held /= Void
  inaugural /= Void
  lectures_held.has (inaugural)
  inaugural_not_void: inaugural /= Void
```

An inaugural lecture is one of the professor's lectures. We use an invariant clause in class *PROFESSOR* to state that *lectures_held*, the list of lectures, contains, or *has*, the inaugural lecture.

Next, there is a time aspect to this. We'll use class *DATE*, a reusable class from the EiffelTime class library, to express that a professor has an *election_date* in ETH terminology. A lecture also has a date, so we add to class *LECTURE* a query of type *DATE*, calling it just *date*.



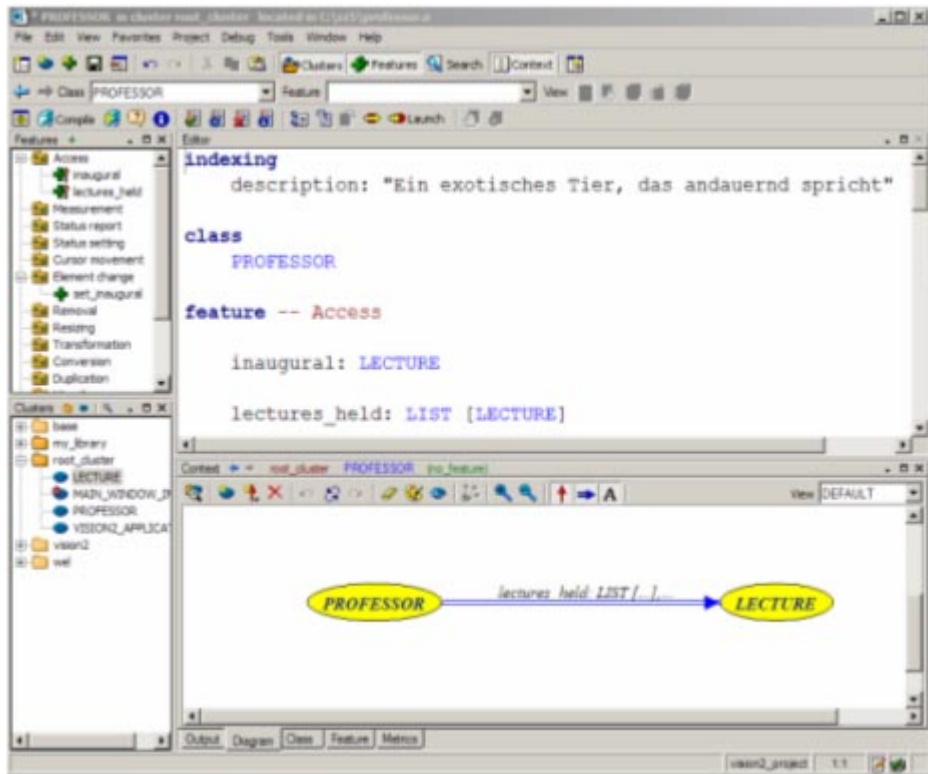
```
invariant
  lectures_held /= Void
  inaugural /= Void
  lectures_held.has (inaugural)
  inaugural not void: inaugural /= Void
  election_date <= inaugural.date
```

I can hear you crying for an invariant: no one may hold an inaugural lecture before being elected; we add to class *PROFESSOR* an invariant clause *election_date* less than or equal to *inaugural.date*.

End of second live demo

We could continue adding details; indeed this *is* the process of building software in Eiffel: identify the right abstractions and specify them precisely. The process applies a set of principles that constitute the Eiffel method:

- First, focus on **data abstractions** — types of objects characterized by the applicable operations and contracts.
- Next, **information hiding**: restrict the clients of a class to these operations, forbidding any bypass, however trivial, because we know this poisons the architecture and closes off long-term evolution.
- Use the same techniques throughout the process, for analysis, architecture, design, implementation, maintenance, debugging. It's called **seamless development**, removing artificial gaps between project steps. The Eiffel notation supports this throughout, as a tool not just for coding but for thinking. Nygaard again: *To program is to understand*.
- This goes with the **Single Model** principle: concentrate all information in the software text, and rely on tools to extract useful views, some abstract, some concrete. We saw some of these tools, to produce documentation, diagrams of inheritance and client relations, variants of an operation such as *forth*, and HTML for any of this.
- Another consequence of seamless development is the ability to change the system late in the process even if the change affects functionality. This is called **reversibility** and addresses the constant demands for change that characterize most software projects.
- You've seen other consequences in the **environment**, which lets you work either graphically, as we did before, or directly on the text. The more commonly accepted technique is to do an overall design with a graphical tool, then move to programming tools — compilers, debuggers — for low-level coding. Here there's no low or high level, just a system which we build using a consistent set of tools.



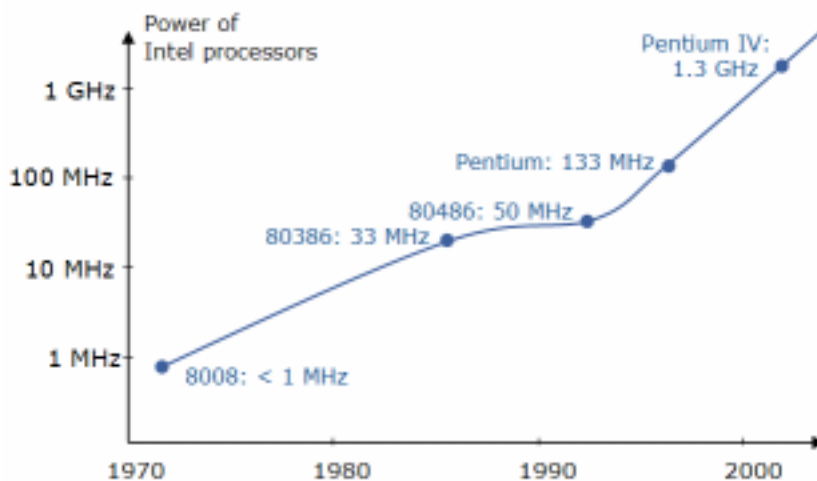
Sometimes it's more convenient to use graphics, sometimes you just type text; the tools ensure that the graphical and textual views remain consistent.

This **text-graphics equivalence** helps software evolution; if the model and its implementation were treated as separate documents, you couldn't hope that implementation changes will be reflected in the analysis. Here both are just facets of the same product.

- Next, we use contracts to characterize classes and operations not only structurally but semantically; this notion of **Design by Contract** pervades the method and the entire development process, not just the technical part but project management too.
- **Inheritance** allows us to keep on top of the complexity arising from the many variants of some notions.
- And a constant obsession, **reusing** the best software elements.

3 Towards trusted components

This concept of **reuse** is essential to the advancement of software engineering.



The reasoning is simple. Like the operating systems I cited at the start, our applications are ever more ambitious; so are our customers' expectations, justified by the continuing extraordinary improvement in hardware — Moore's law — captured in this comment that it took Intel twenty-six years to get to one Gigahertz, then eight months to two Gigahertz.

This makes ever more burning the fundamental software engineering questions: correctness and change. The paradox is that it's possible in principle to get much better software by applying the best techniques of software engineering. But that may cost more and take longer than most projects can accept in our very competitive industry. The slogan is “**good enough software**”. In the short-term it may work, but global effects are disastrous.

The only way I see out of this dilemma is to capitalize on the commonality of the ground elements of many programs; similar patterns occur again and again. This is one of the attractions of object technology, with its ability to capture useful abstractions in libraries of reusable classes. We've already used such libraries: EiffelBase gave us *LIST*, EiffelTime gave *DATE*, EiffelVision gave the graphics. There have also been binary component models such as Borland's Delphi, Microsoft's COM, Sun's Java Beans and most recently Microsoft .NET, yielding components at a higher level of granularity, ready to be deployed into applications.

How can reusable components shape software's future? They *legitimate perfectionism*. Managers often dismiss programmers' perfectionism as harming the timeliness of releases, and push instead for “good-enough software”. Components are different: “good-enough” is no longer good enough as the scale of reuse magnifies the likelihood that any small deficiency will harm some program; and perfectionism now makes economic sense, as the scaling effect justifies the extra effort. That's why building reusable components is the most exciting task in software today. Having spent constant efforts to developing quality components for many years, I can testify to the intellectual challenge and pleasure; it's the opportunity to do things right.

It's also, of course, a great opportunity to do things wrong. How do you know that your application, built with components from many sources, isn't going to fail out of a defect in one of them? We can't ignore the comparison with electronics, which also relies on components, but with a draconian approach to quality. Anywhere you go in an electronics factory or lab, you see people checking the quality of everything that comes in and goes out. No electronics company would last long without a rigorous qualification process for the components it receives and produces. In software, for all the noise that we hear about components and reuse, we are very far from having adopted such an attitude.

Hence the idea of trusted component, which I put at the center of any effort to improve the state of software today.

Trusted Component: definition

A trusted component is a reusable software component with known and guaranteed quality properties.

If we manage to make this notion a reality, we may see a different industry in the future based on quality components, not just believed to be of high quality but built with that obsession in mind and certified by proper agencies, such as I hope will exist at ETH.

This argument for trusted components is **political** and **economic**. Stable human systems are the ones that succeed in matching cumulative optimization of individual self-interest with global optimization of the common good. That's the theory behind capitalism. Without components, the self-interest of a software manager does *not* favor excellence, since the most likely consequence of perfectionism is to delay the project and losing the market; it favors *good enough* software. Worse, to improve my software, *I* am the one who must work harder.

With components, it's the interest of both my component supplier and me that components be good. And if I find one not good enough, I have every reason to demand better quality, since someone else will do the job. Optimization of the common good matches combined optimization of individual benefits.

There's also a **sociological** argument. Much software comes from non-professionals. Six million people program in Visual Basic, of whom few studied computer science, even fewer at ETH. Although we must teach our best methods to non-computer-scientists, the effect we can

have on such a large population is limited. Our strongest lever is to give them impeccable components of ever broader scope. They can still mess up when combining and extending them, but the more and better we give them the less this will happen.

We must meet a number of challenges before turning this vision of trusted components into reality. They are also research goals, in three categories.

- First, building components; we can't just preach, we must show the way. This is the opportunity for development of extensive component libraries, in line with the great ETH Computer Science tradition of building things.
- The second goal is to develop a **Component Quality Model** to assess today's and tomorrow's components, and build a **Component Certification Center** to apply the model to real components, commercial or open-source, with obvious interest to the industry, including, one hopes, local industry.
- The last point is about **proving** properties of components, starting with classes, which already present challenges if we factor in such aspects as pointers, concurrency, threading, persistence.

This is the mathematical notion of proof. You may be surprised to hear of proofs in connection with programming, usually considered a more informal activity. In fact, there is an entire body of work devoted to this. It's called *formal methods*.

In the end, a program is a mathematical object; assuming the hardware satisfies precisely defined functionality, the program's properties are entirely defined by its text.

So it should be possible to prove them, with some interesting consequences. *Testing*, for example, is in principle no longer necessary: when you've proved a theorem in mathematics, you don't pay testers to search for counter-examples. In practice it's not as simple, but formal methods technology has progressed enough to be applied to sizable mission-critical systems in defense, trains, aerospace, where imperfect software can cause loss of life.

Formal methods are applicable more broadly, but they still scare away project managers in areas where bugs don't kill people quite immediately.

Once again components redefine the discussion; the prospect of widely available components with mathematically proved properties is too exciting to pass. The challenges are numerous, but it's one of the most exciting goals in software technology today.

4 Teaching

I have described a number of key techniques, tools, and research directions. The final question is how they relate to the curriculum.

In my mind these ideas are just ripe for teaching. We don't need to make our students go through the successive steps that we in our time had to climb, to make ontogeny repeat phylogeny. We should present them right from the start with the best of modern software technology. I won't go into the details of what I've called, borrowing from electrical engineering, the "**inverted curriculum**". In software this involves giving students, from the outset, considerable amounts of existing, high-quality software to discover and use. One should be a consumer before becoming a producer. The best way to learn is by osmosis, by imitation.

The software should also be exciting enough to 19-year-olds; it should involve graphics, multimedia. To be scientifically rigorous, a teaching program doesn't have to be boring.



They don't know it yet, but that is the treatment that awaits, next year, poor, innocent, unsuspecting future Informatik students. Die einen werden aus Schlieren, Schwyz oder Schaffhausen kommen; alcuni verranno da Lugano, da Locarno sia da Vogorno; certains accourront peut-être de Vallorbe, de Verbiers ou

bien de Vevey; может-быть даже и некоторые из Омска или Томска ; right now they live happily in their hometowns, with no idea of the tortures to come. Of course you can still, Herr Rektor, Herr Departementvorsteher, stop this before it's too late.



Sihl/kon Valley

When teaching software we must develop high-level system skills without neglecting the low-level hands-on programming part, sometimes called **hacking**, a term I want to use here without any pejorative connotation.

We cannot teach *only* hacking, if only because any company that just needs an implementation job will find it increasingly attractive to outsource it to India or Egypt rather than ask a more expensive local developer. We need to teach **abstraction** mechanisms, all these modeling techniques and principles I cited earlier, because a successful software engineer must be an architect who sees the big picture. But we need to teach *also* hacking, because the global view is not enough, and to succeed in software you must have written programs yourself at all levels of abstraction. Besides, someone needs to check that code outsourced to Bangalore.

The seamless development approach of the Eiffel method, which I have briefly tried to illustrate, spans the full spectrum, from the most abstract modeling and architectural tasks to the most concrete implementation steps.

A famous citation tells us that philosophy always comes late, after history has done its damage, unaffected by the Owl of Minerva — the philosophers. Teaching software faces the same contradiction; if you teach principles early — letting the owl fly at dawn — students have little idea what you are talking about, and might miss the lessons of abstraction, reuse, correctness, formal methods. But if they go to industry and start programming without having learned this discipline, the owl of Minerva may fly too late.

I don't quite know the answer to this dilemma. But I do know that the ideas we must teach form the basis of a solid emerging science. Not everyone agrees; in some universities, computer science is still a poor relative of electrical engineering or mathematics. We have strong ties to both. Without electrical engineering there are no computers; computer science is to electrical engineering as the art of making love is to the art of making beds.

The relation to mathematics is just as close; I mentioned earlier that a program is for all practical purposes a mathematical object. The main difference is that where mathematics deals with difficult problems, we deal with problems that are often difficult but even more often complex. I mentioned transportation systems developed and proved formally; the proof of a famous one takes thirty thousand elementary steps — thirty thousand lemmas. Many, although not all, are trivial to a human mathematician; but no human will tackle a proof of thirty thousand steps, or, if he did, convince anyone that it's right.

So the challenges we face are different. In tackling them, we have pushed some ideas further than other disciplines:

Software concepts	
Structure	Information hiding
Reuse	Taxonomy
Coping with change	Language
Abstraction	Distinguishing between the static and the dynamic
Complexity	Recursive reasoning
Scaling up	Invariants
Distinguishing between specification and implementation	

Many were of course familiar before; to take the last two, recursion comes from the mathematical notion of induction, and invariants arise throughout physics. But computing science give these concepts a full new extent.

Also unique to our field is its peculiar mix of abstract and concrete, mathematics and engineering, ideas and products; angel and beast. By calling it engineering we acknowledge the applied nature of our work, comparing ourselves to people who build bridges and cars and circuits. But our products have a different relation to the physical world. They may influence it, like the products of engineering, but are themselves immaterial, like those of mathematics, or for that matter philosophy.



This paradox affects everything; for example why it's so critical to learn to distinguish specification from implementation. In other fields no one would confuse a thing with the plan of the thing; you can fall off the bridge, but you will not fall off the plan of the bridge; the approaching car may hurt you but you don't fear the drawing of the car. In software the distinction is far more fragile; anyone who has worked on specification knows how good specifications sneakily start resembling good programs. We're not always sure whether we are looking at a pipe or the picture of a pipe.

This is why we must be careful when teaching students to specify, formally or not.

Or take the recurring debate on software copyrights and patents. Elsewhere the two are exclusive: what's copyrightable is the expression of ideas, like a novel, a movie script; what's patentable is an invention, like a mechanical device. You cannot patent a novel, or copyright a device. Yet people apply both to software.

This half-angel half-beast nature underlies many of the challenges of our field. Where else could we in the same breath describe something as an industry and as a set of ideas? Mathematics is pure ideas, but it's not industry; no shop buys and sells theorems. Electrical engineering defines an industry, and its products are more than ideas; if I put my fingers into this outlet, it's not a pure idea that will burn me.

Software is a multi-hundred billion dollar industry, with products bought and sold all the time, fortunes being built, fortunes — as you may have noticed recently — being lost; and yet that industry rests on products that have no more reality than a mathematician's proof or a poet's dream.

Diese Industrie ist, Herr Rektor, Herr Departementsvorsteher, was wir verstehen, entwickeln und unterrichten müssen: l'industrie des idées pures.

The industry of pure ideas

Sources

Major-General's aria from Gilbert and Sullivan's *The Pirates of Penzance*, Stratford Festival (Canada), Acorn Media.

Kafka, *Der Prozess* (The Trial), chapter 1

Dulcamara's first act aria by Simone Alaimo from Donizetti's *L'Elisir d'Amore*, Brian Large, Roberto Alagna, Angela Gheorghiu, dir. Evelino Pido Decca

"Zurück!" from *Die Zauberflöte*, Klemperer recording, Gedda/Janowitz/Berry/Unger/Popp/Frick/Schwartzkopf, EMI Classics.

Catalog aria by José van Dam in from Losey's *Don Giovanni*, dir. Lorin Maazel, with Kiri Te Kanawa.

René Magritte: *Ceci n'est pas une pipe*.

Hegel, *Foundations of the Philosophy of Law*, preface.

Acknowledgments: Slides, Ruth Bürkli; software examples, Karine Arnout; technical assistance (video, software), Bernd Schoeller; support during the presentation, Karine Arnout and Volkan Arslan; additional support, Volkan Arslan and Susanne Cech.