From programming to software engineering:

Notes of an accidental teacher

Bertrand Meyer

ETH Zurich & Eiffel Software

ETH
Chair of
Software Engineering

About these slides

This is the slide set for my Education Keynote at ICSE (International Conference on Software Engineering), Cape Town, South Africa, 5 May 2010.

Usual caveats apply: this is only supporting material, not all of it understandable independently of the talk. Many of the original slides (in particular the programming-related examples) include animation, not visible in this version.

URLs are clickable and have associated screen tips.

“Accidental”*

*Post-talk note: slide removed

Thanks to…

Michela Pedroni, Manuel Oriol, Martin Nordio, Peter Kolb, Till Bay, Roman Mitin, Karine Arnout and many others

Content

1. Definitions: programming and software engineering
2. Lessons from experience: teaching programming
3. Lessons from experience: teaching software engineering
4. General lessons

Teaching programming: concepts or skills?

Quiz

Your boss gives you the source code of a C compiler and asks you to adapt it so that it will also find out if the program being compiled will not run forever (i.e. it will terminate its execution).

1. ❐ Yes, I can, it's straightforward
2. ❐ It's hard, but doable
3. ❐ It is not feasible for C, but is feasible for Java
4. ❐ It cannot be done for any realistic programming language

Teaching programming: concepts or skills?

Skills supporting concepts

Teaching programming: some critical concepts

Specification vs implementation, information hiding, abstraction

Notation

Change

Syntax vs validity vs semantics

Structure

Recursive reasoning

Reuse

Classification

Complexity & impossibility

Function vs data

Algorithmic reasoning

Typing

Scaling up

Complexity

Static vs dynamic

Invariant

Software engineering definitions

SWEBOK, Wikipedia:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

The application of engineering to software.

Parnas (cited in Ghezzi, Jazayeri, Mandrioli): "**The multi-person construction of multiversion software**"

*Post-talk note: the discussion explained why this definition is unsatisfactory.

Teaching software engineering

"DIAMON":

- **D**escription: specify (requirements, systems, designs,implementations...) and document
- **I**mplementation: build the products (includes both programming & design)
- **A**ssessment: verify, validate, analyze, test, measure (both products and processes)
- **M**anagement: organize the work, communicate, collaborate
- **O**peration: deploy systems and oversee their proper functioning
- **N**otation: devise and apply appropriate formalisms

The natural path

Notation

Implementation

Description

Assessment

Management

Operation

1. Definitions: programming and software engineering

2. Lessons from experience: teaching programming

3. Lessons from experience: teaching software engineering

4. General lessons

Introductory programming teaching

Teaching first-year programming is a politically sensitive area, as you must contend not only with your students but also with an intimidating second audience — colleagues who teach in subsequent semesters….

Academics who teach introductory programming are placed under enormous pressure by colleagues.

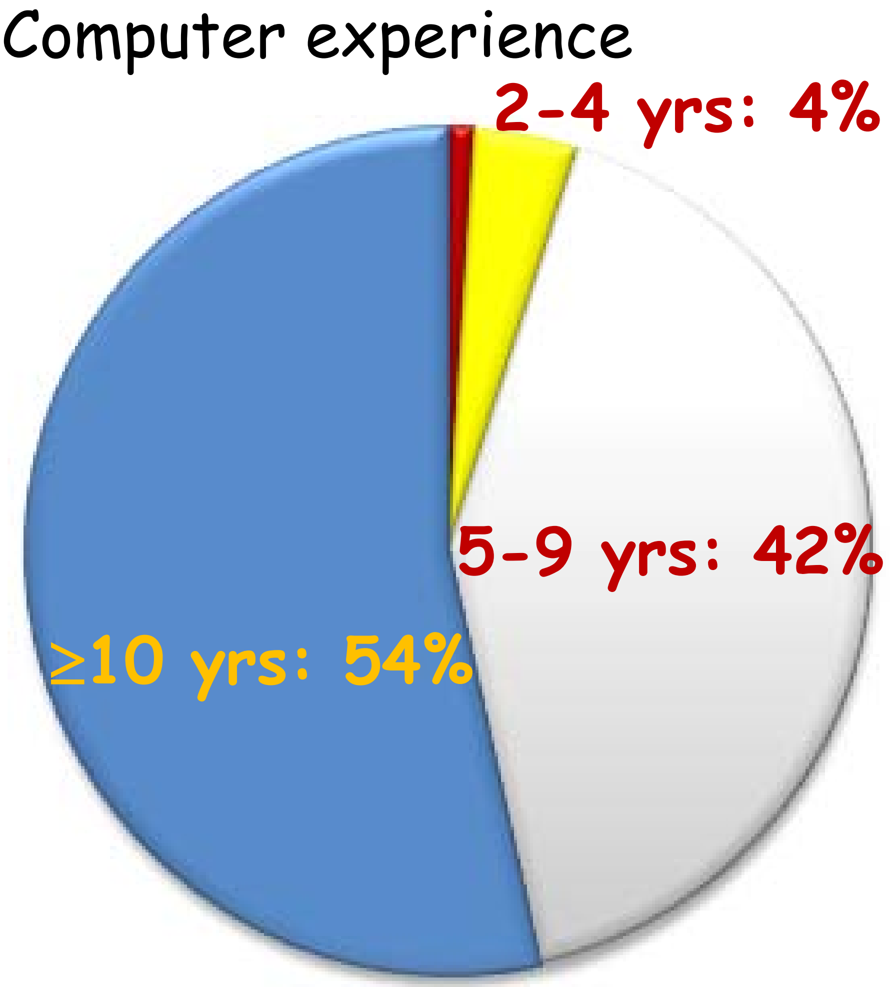
As surely as farmers complain about the weather, computing academics will complain about students' programming abilities.

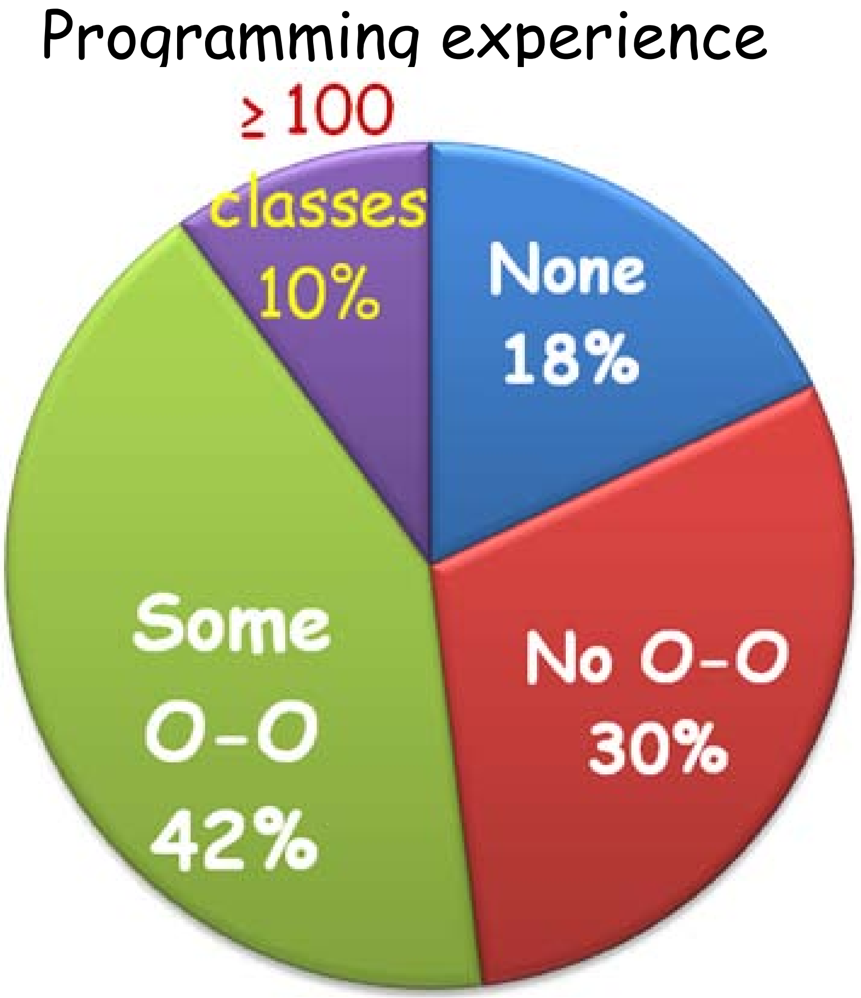
Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*,
10th Conf. on Australalasian computing education, 2008

Some challenges in teaching programming

- Ups and downs of high-tech economy, image of CS
- Offshoring and globalization raise the stakes
- Short-term pressures (e.g. families), IT industry fads
- Widely diverse student motivations, skills, experience

The Facebook generation: 1st-year CS students





For year-by-year figures & analysis: see Pedroni, Meyer, Oriol, *What do beginning CS majors know?*, 2009, se.ethz.ch/~meyer/publications/teaching/background.pdf

Averages, 2003-2008 (yearly variations small)

Ways to teach introductory programming

- 1. “Programming in the small“
- 2. Learn APIs
- 3. Teach a programming language: Java, C++, C#
- 4. Functional programming
- 5. Completely formal, don’t touch a computer

Our approach: Outside-In (inverted curriculum)

Concepts or skills?

Skills supporting concepts

Teaching programming: some critical concepts

Specification vs implementation, information hiding, abstraction

Notation

Change

Syntax vs validity vs semantics

Structure

Recursive reasoning

Reuse

Classification

Complexity & impossibility

Function vs data

Algorithmic reasoning

Typing

Scaling up

Invariant

Complexity

Static vs dynamic

Invariants: loops as problem-solving strategy

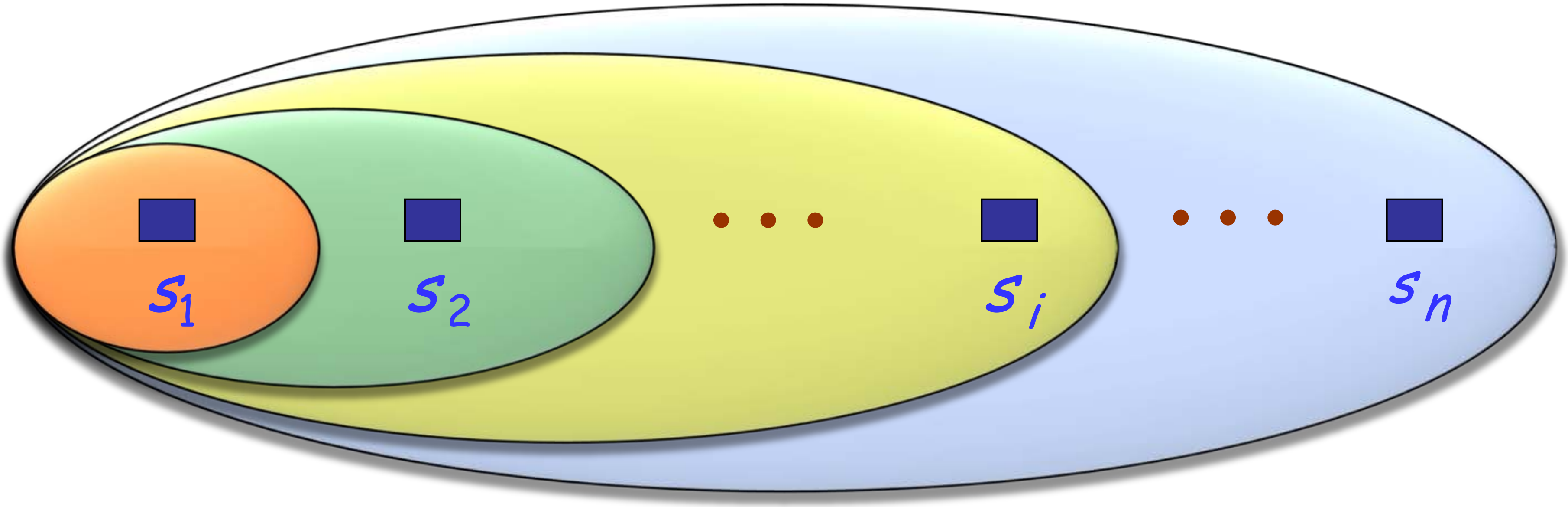
A loop invariant is a property that:

- Is easy to **establish initially** (even to cover a trivial part of the data)
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**!

Computing the maximum of a list

```
from
        ???
invariant
        ???
across structure as i loop
        Result := max (Result, i.item)
end
```

Loop as approximation strategy



Result = a_1 = Max (s_1 .. s_1)

Result = Max (s_1 .. s_2)

Loop body:

i := i + 1

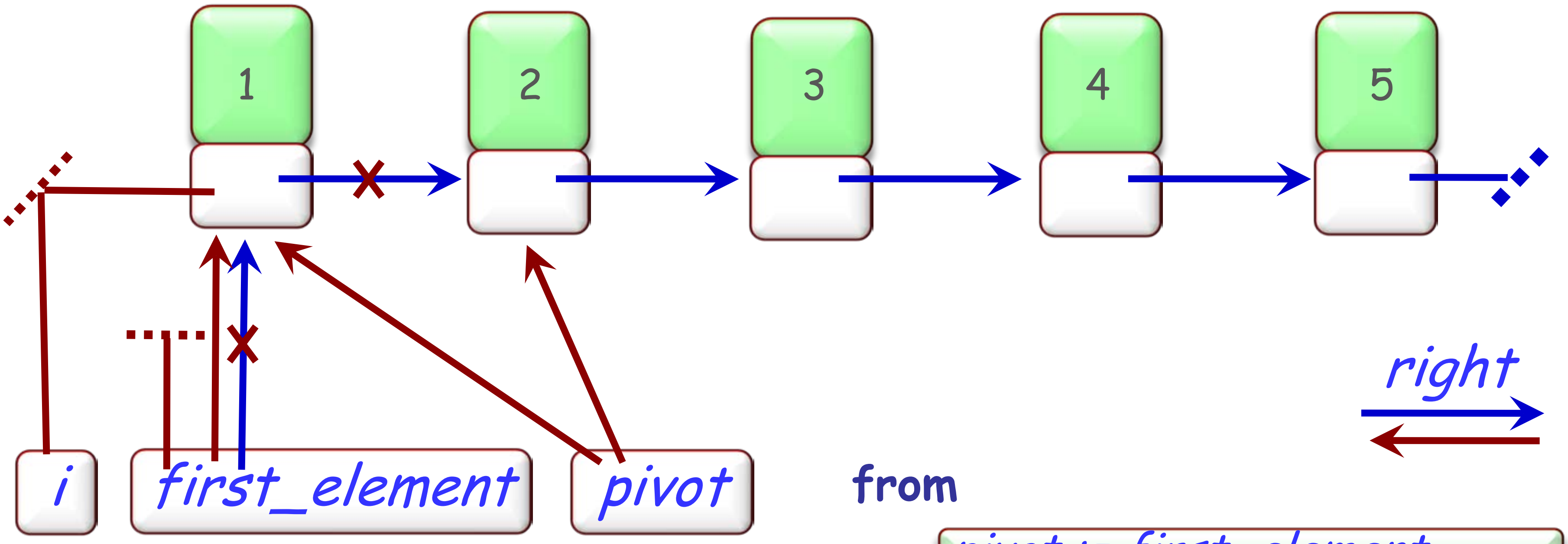
Result := *max* (**Result** , s_i)

Result = Max (s_1 .. s_i)

The loop invariant

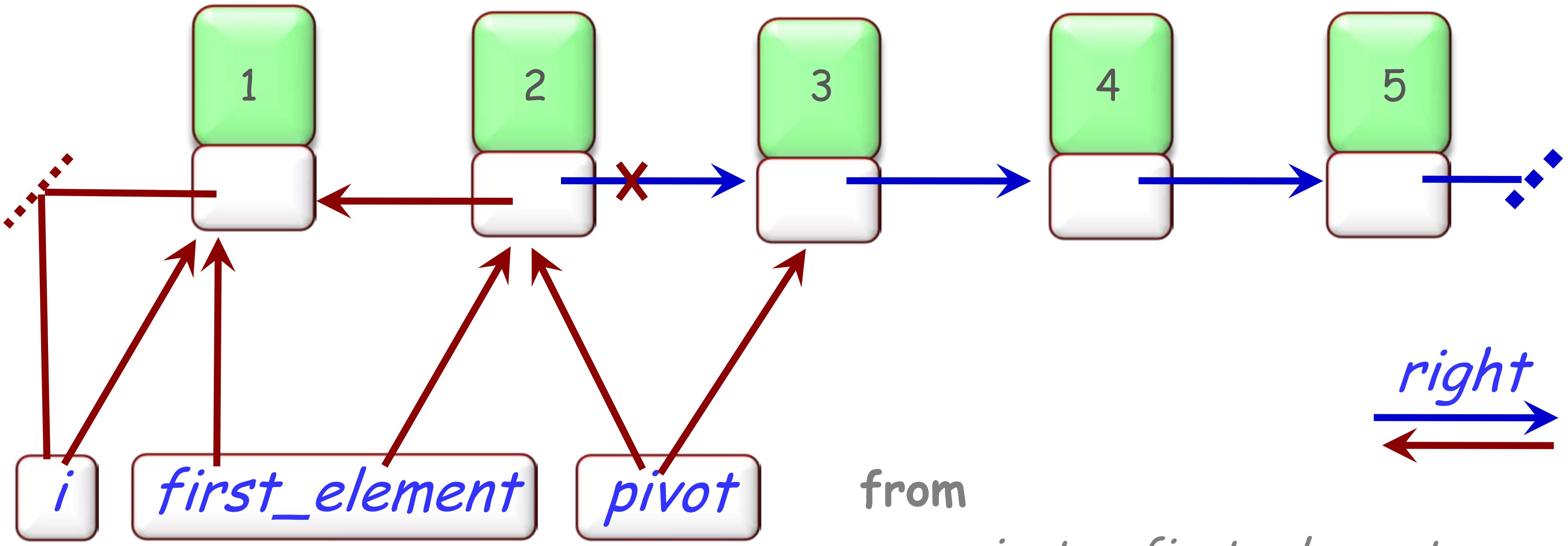
Result = Max (s_1 .. s_n)

Reversing a list



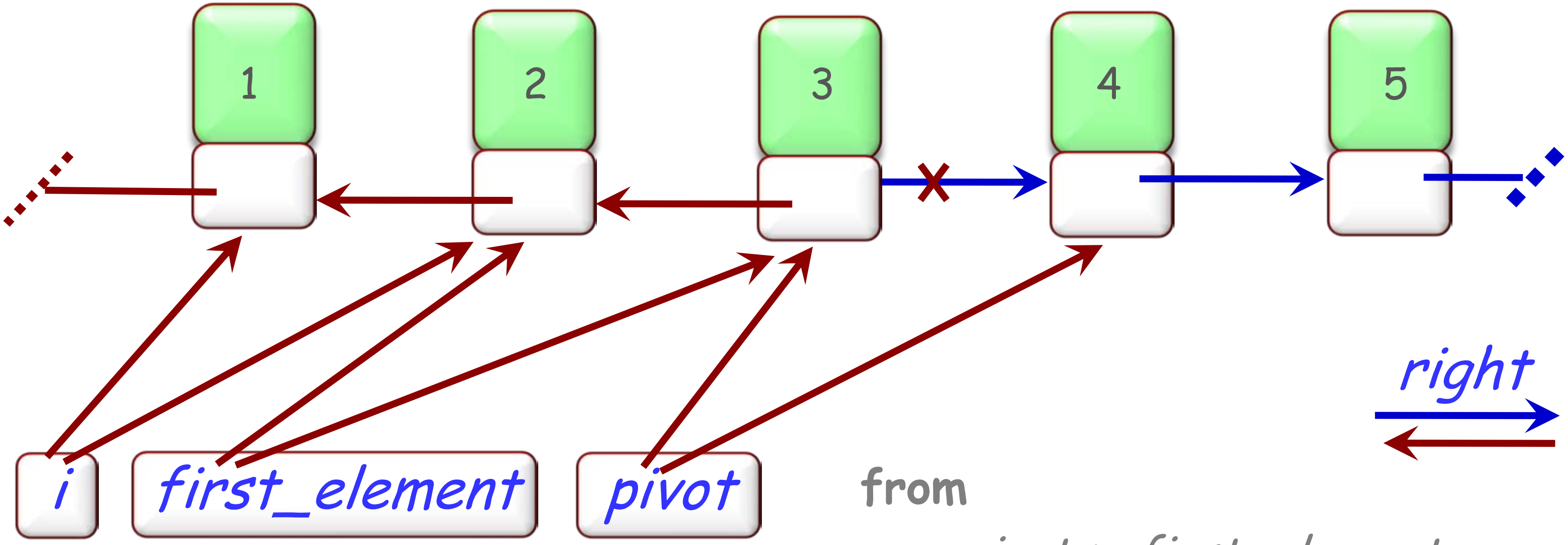
```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list



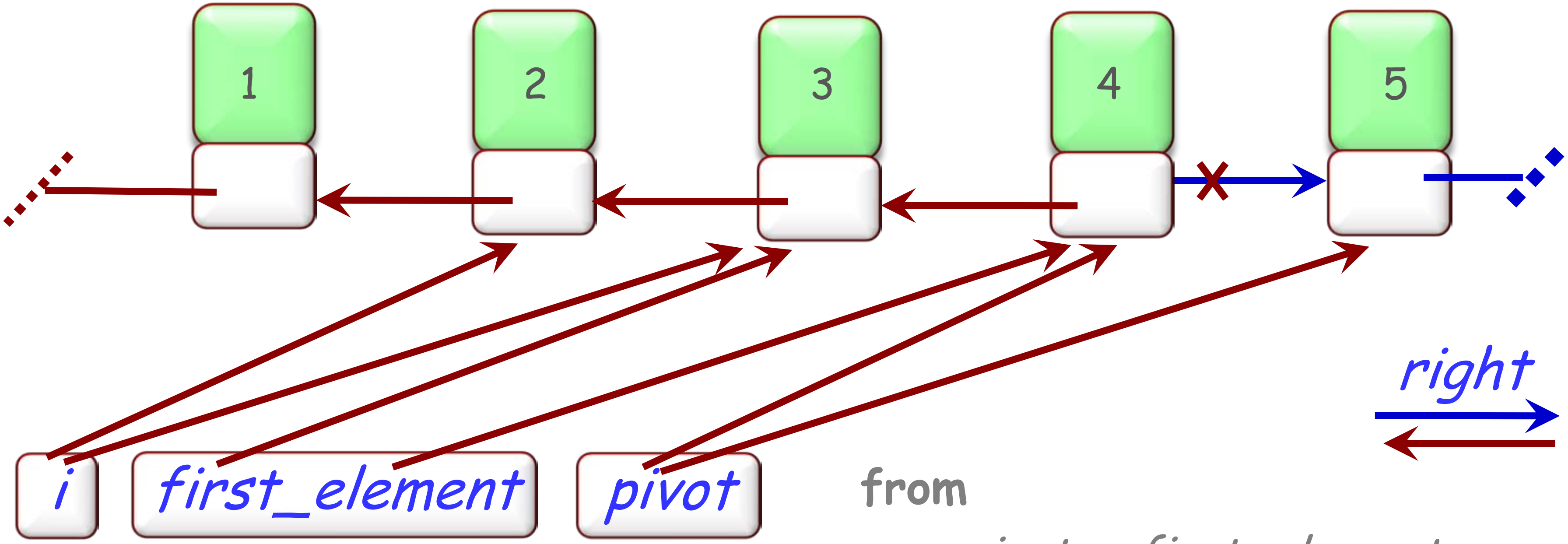
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



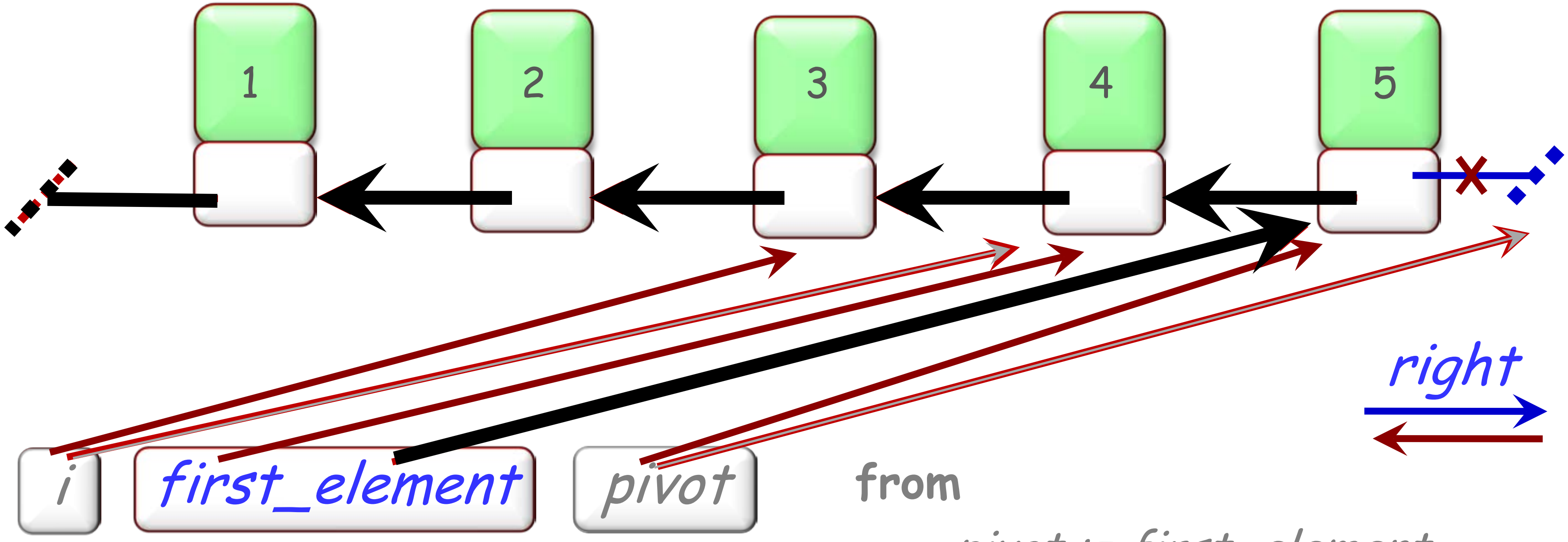
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



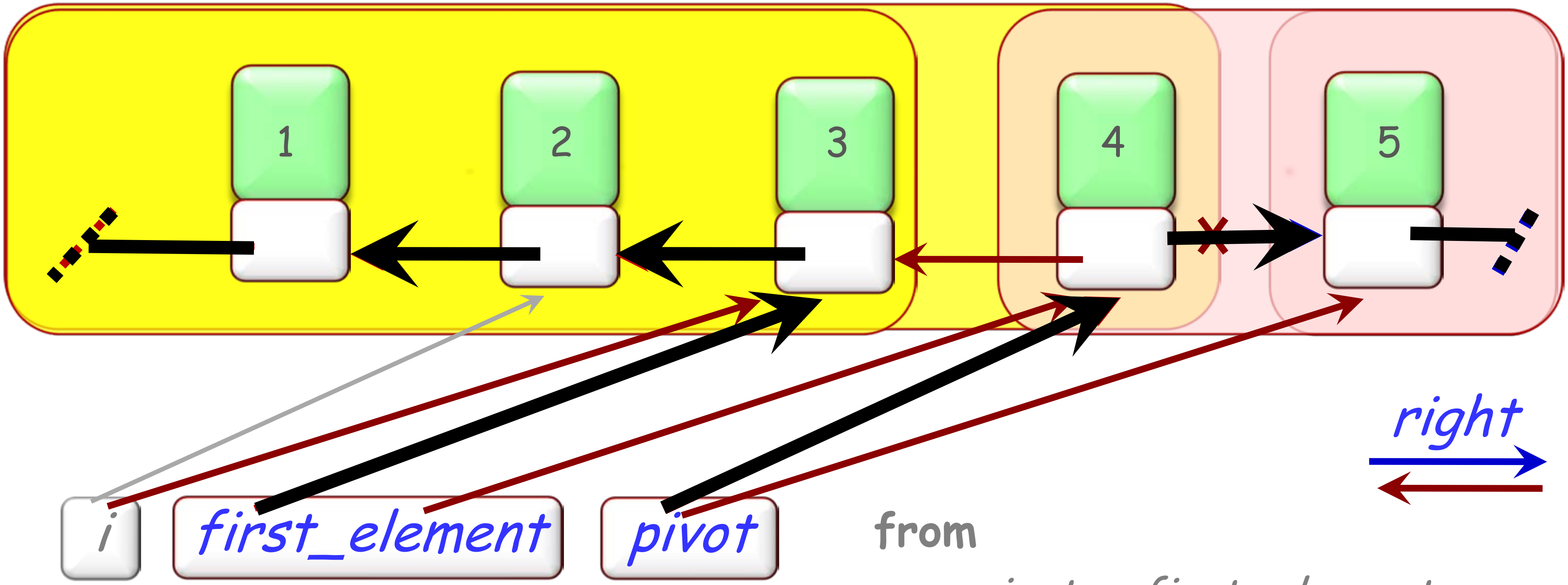
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Why does it work?



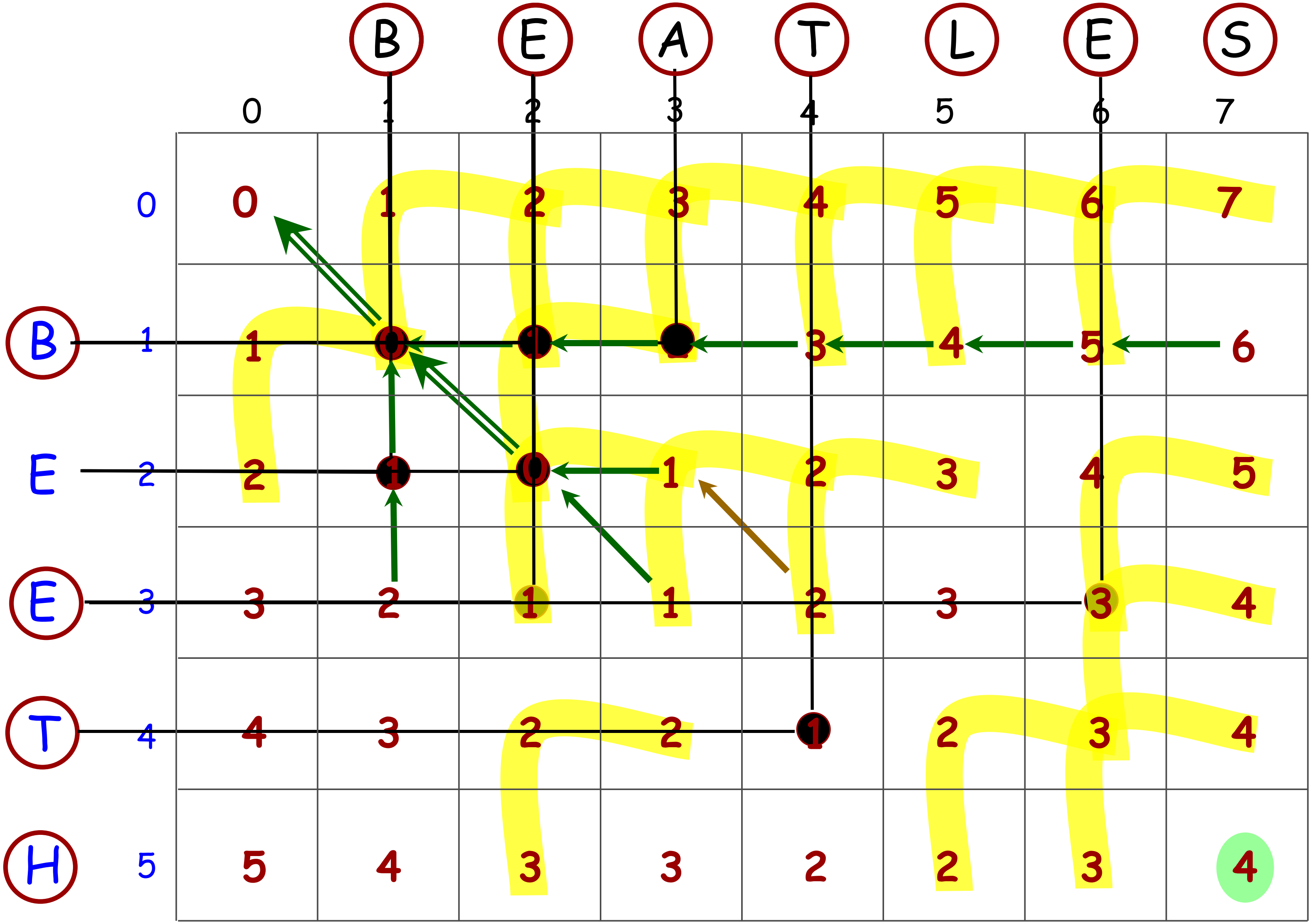
Invariant: from *first_element* following *right*, initial items in inverse order; from *pivot*, rest of items in original order

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Levenshtein distance

"Beethoven" to "Beatles"

	B	E	E	T	H	O	V	E	N
			A			L			S
Operation	–	–	R	–	D	R	D	–	R
Distance	0	0	1	1	2	3	4	4	5

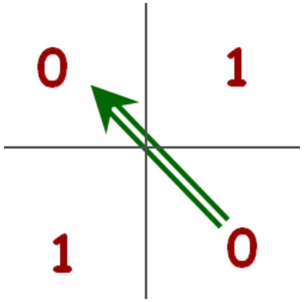


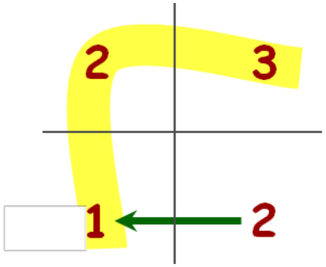
			B	E	A	T	L	E	S
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
B	1	1	0	1	2	3	4	5	6
E	2	2	1	0	1	2	3	4	5
E	3	3	2	1	1	2	3	3	4
T	4	4	3	2	2	1	2	3	4
H	5	5	4	3	3	2	2	3	4

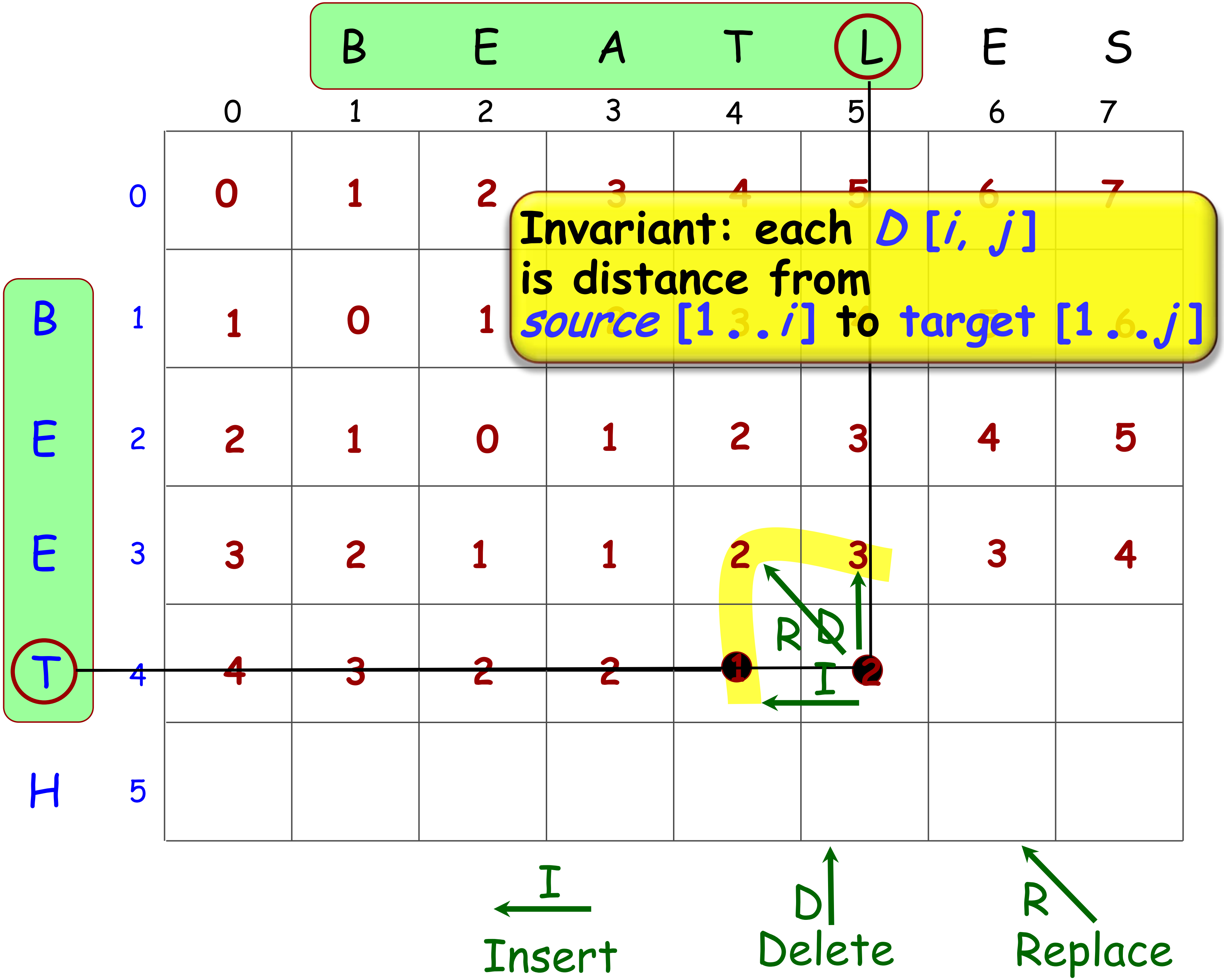
D

Levenshtein algorithm

```
across r : 1 |..| rows as i loop
    across c : 1 |..| columns as j invariant
        -- For all p : 1 .. i, q : 1 .. j–1, we can turn source [1 .. p ]
        -- into target [1 .. q ] in D [p, q ] operations
    loop
        if source [i ] = target [ j ] then
            D [i, j ] := D [ i -1, j -1]
        else
            D [i, j ] := 1 +
                    min (D [i -1, j ], D [i , j - 1], D [i - 1, j - 1])
        end
    end
end
Result := D [rows, columns]
```







			B	E	A	T	L	E	S
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
B	1	1	0	1					
E	2	2	1	0	1	2	3	4	5
E	3	3	2	1	1	2	3	3	4
T	4	4	3	2	2	1	2		
H	5								

Invariant: each *D* [*i*, *j*] is distance from *source* [1 . . *i*] to *target* [1 . . *j*]

Concepts or skills?

Skills supporting concepts

Outside-in (Inverted Curriculum): intro course

Fully object-oriented from the start, using Eiffel

Design by Contract principles from the start

Component based: students use existing software (TRAFFIC library):

- They start out as consumers
- They end up as producers!

Michela Pedroni & numerous students

≈ 150,000 lines of Eiffel

"Progressive opening of the black boxes"

TRAFFIC is graphical, multimedia and extendible

(Approach 3: teaching a specific language)

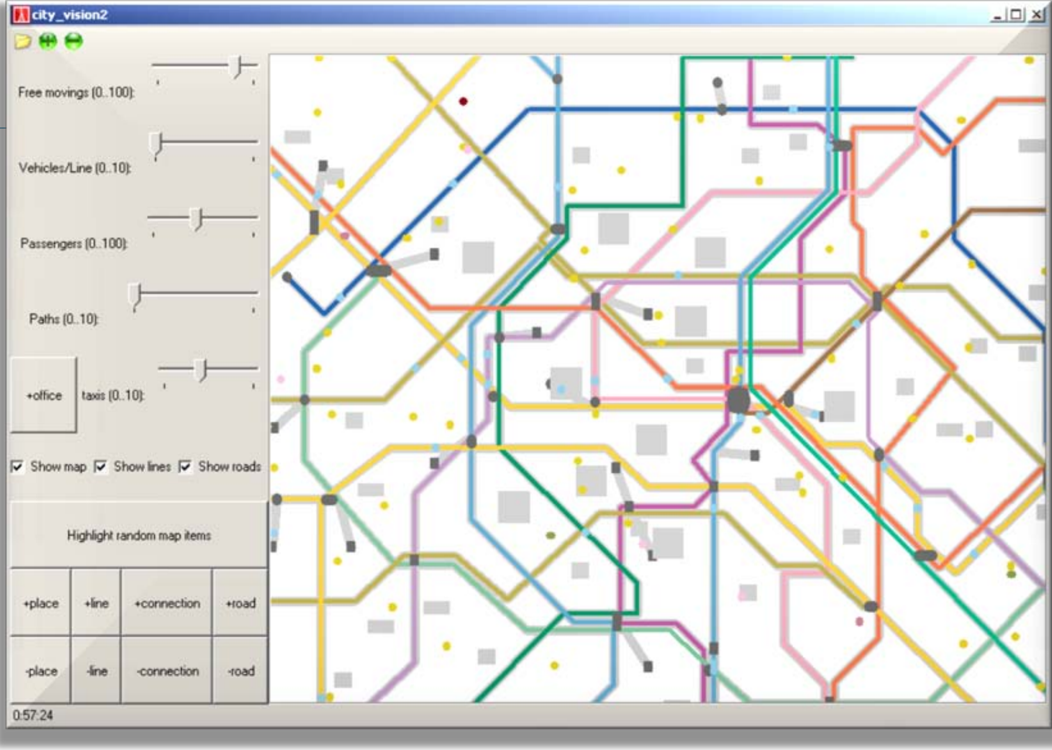
First Java program:

```
class First {
        public static void main(String args[])
        { System.out.println("Hello World!"); } }
```

You'll understand when you grow up!

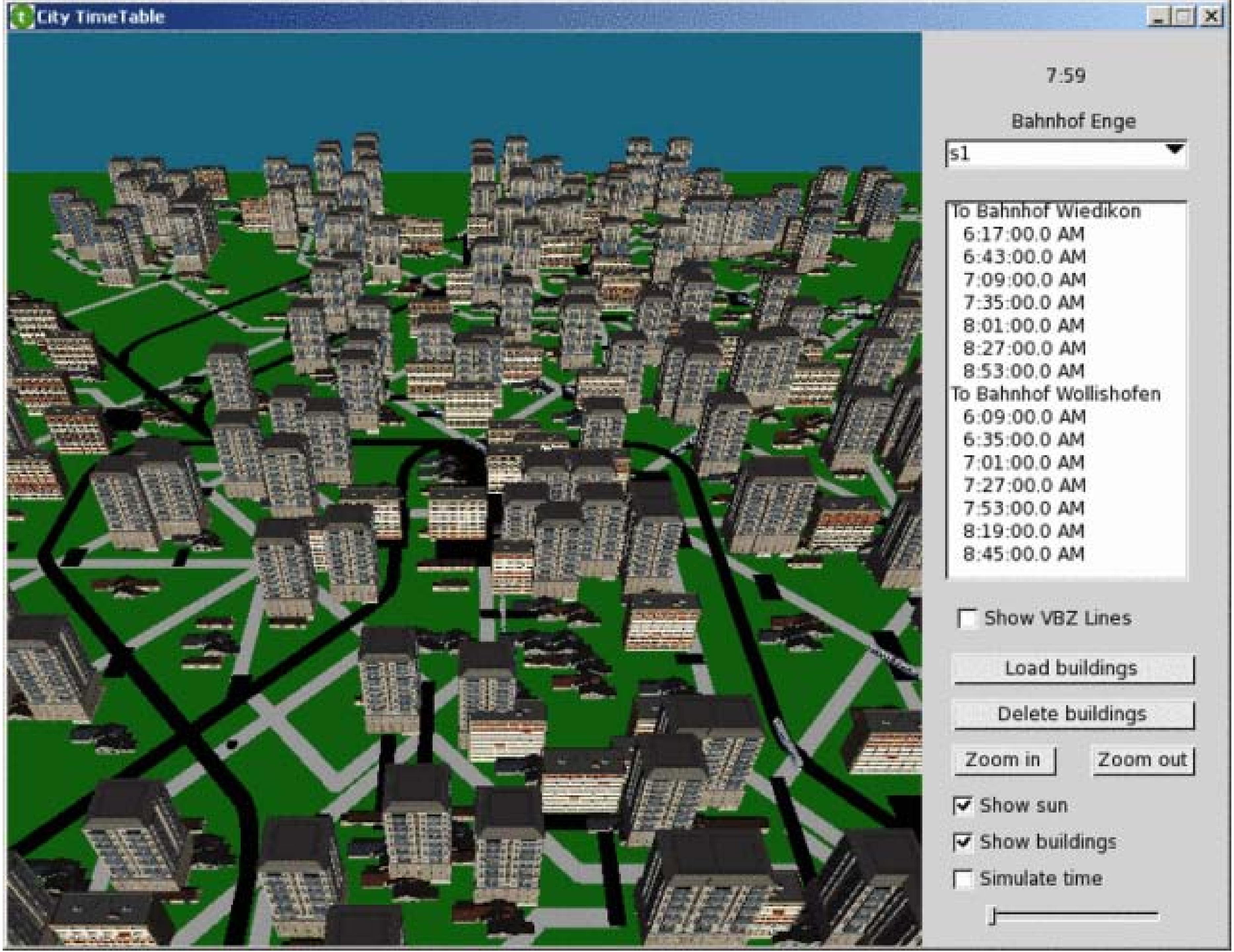
Do as I say, not as I do

Our first "program"



```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Prepare & animate route
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

Text to input



7:59

Bahnhof Enge

s1

```
To Bahnhof Wiedikon
  6:17:00.0 AM
  6:43:00.0 AM
  7:09:00.0 AM
  7:35:00.0 AM
  8:01:00.0 AM
  8:27:00.0 AM
  8:53:00.0 AM
To Bahnhof Wollishofen
  6:09:00.0 AM
  6:35:00.0 AM
  7:01:00.0 AM
  7:27:00.0 AM
  7:53:00.0 AM
  8:19:00.0 AM
  8:45:00.0 AM
```

☐ Show VBZ Lines

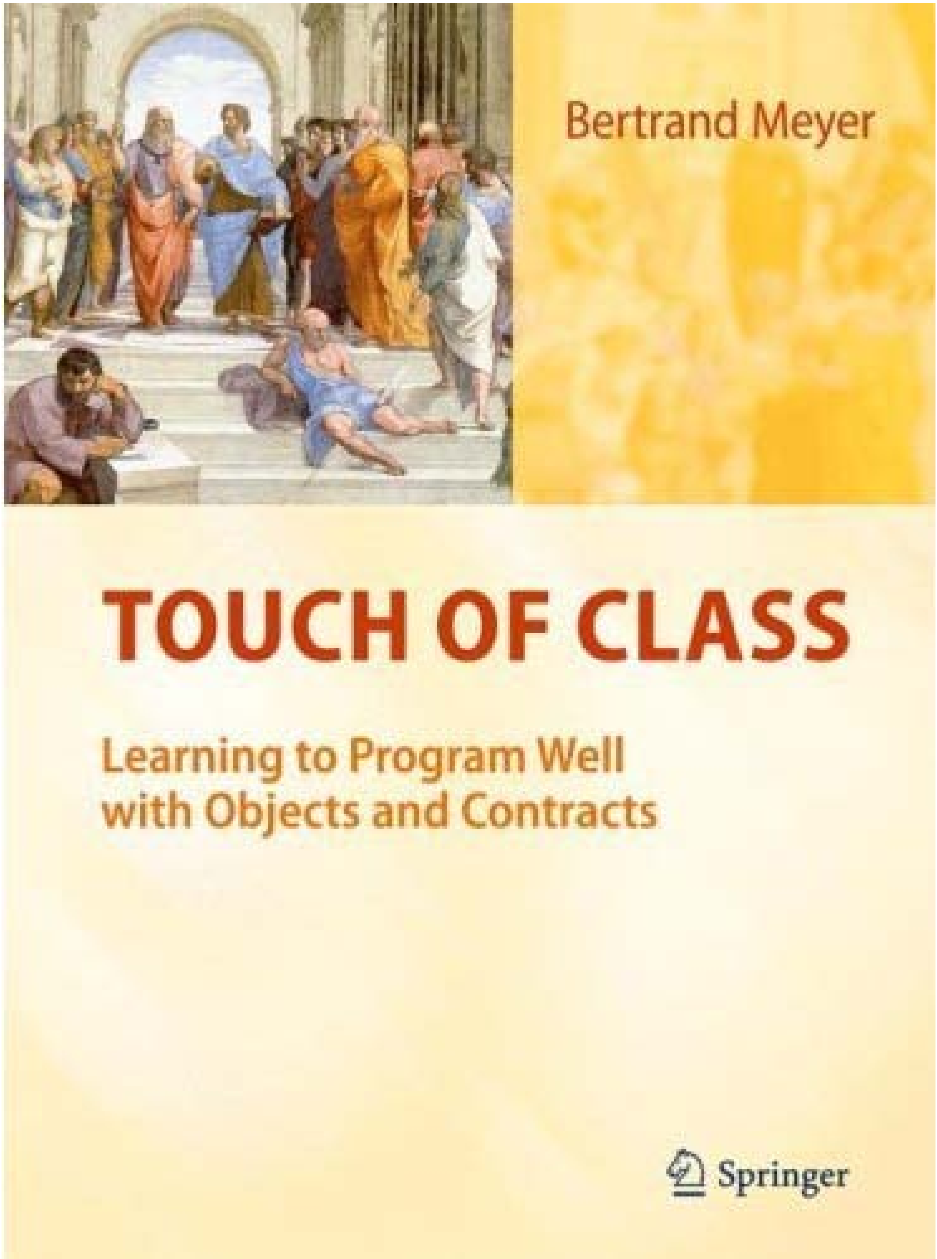
☑ Show sun

☑ Show buildings

☐ Simulate time

Supporting textbook

touch.ethz.ch



Bertrand Meyer

TOUCH OF CLASS

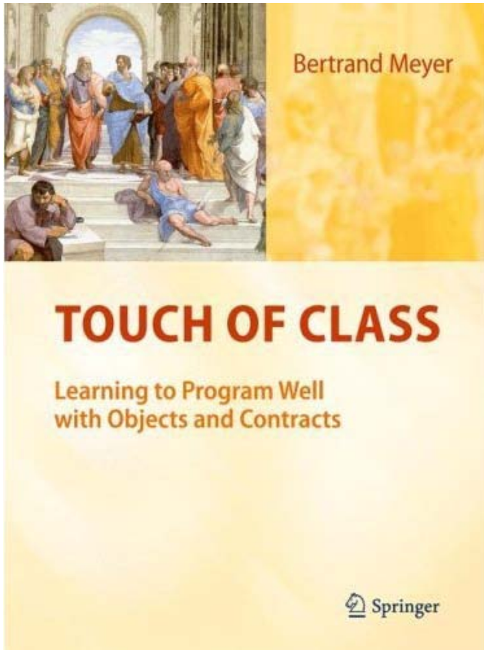
Learning to Program Well
with Objects and Contracts

Springer

Springer, 2009

Principles of the ETH course

- Reuse software : inspiration, imitation, abstraction
- See lots of software
- Learn to reuse through interfaces and contracts
- Interesting examples from day one
- Combination of principles and practices



touch.ethz.ch

Traditional topics too: algorithms, control structures, basic data structures, recursion, syntax & BNF, …

Advanced topics: closures & lambda-calculus, some design patterns, intro to software engineering…

1. Definitions: programming and software engineering
2. Lessons from experience: teaching programming
3. Lessons from experience: teaching software engineering
4. General lessons

Teaching software engineering

Basic courses:

- Software engineering (3rd year)
- Software architecture (2nd year)

Advanced courses:

- Distributed & outsourced software engineering (DOSE)
- Software verification
- (etc.)

Some principles for SE/SA courses

Basic goal: cover what a good programming student does not know about SE

- Do not attempt a catalog
- Do teach key industry practices (e.g. UML)
- Emphasize non-"**I**" & non-"**N**"parts
- SE is not SA
- *A university is not a company*
- Emphasize falsifiable knowledge
- Include a project (see next)

- **D**escription
- **I**mplementation
- **A**ssessment
- **M**anagement
- **O**peration
- **N**otation

Principles for SE course projects

- Include implementation
- Students implement what they specify
- Swap development & testing
- Manage collaboration
- Spell out project's pedagogical goals
- Choose which industry characteristics to include & not

The object-oriental bazaar

One thing I would like to know…

Designing good non-multiple-choice exam questions for SE
Example from a medical textbook*:

Case History B

A woman (24 years of age; height: 1.70 m; weight: 60 kg) is in hospital due to a tremendous thirst, and she drinks large amounts of water. Since she is producing 10 or more litres of urine each day, the doctors suspect the diagnosis to be diabetes insipidus. The vasopressin concentration in plasma (measured by a RIA method) is 10 fmol per l. […] The extracellular volume (ECV) is 20% of her body weight. […]

1. Calculate the secretion of vasopressin (in mg/hour) from the neurohypophysis of a normal 60-kg person and of this patient […]

4. Estimate the relation between this concentration and that of a healthy individual.

5. Does this ratio have implications for the interpretation of her special type of diabetes insipidus?

6. Is it dangerous to lose 10 litres of urine per day?

*www.mfi.ku.dk/ppaulev/chapter26/Chapter%2026.htm

Distributed software engineering

Today's software development is multipolar

University seldom teach this part!

"*Software Engineering for Outsourced and Offshore Development*" since 2003, with Peter Kolb

Since 2007: **Distributed & Outsourced Software Engineering (DOSE)**

The project too is distributed. Currently: ETH, Politecnico di Milano, U. of Nijny Novgorod, Odessa Polytechnic, U. Debrecen, Hanoi University of Technology

The DOSE project

Setup: each group is a collection of teams from different university; usually 2 teams, sometimes 3

Division by functionality, not lifecycle

Results:

- Hard for students
- Initial reactions often negative
- In the end it works out
- The main lesson: **interfaces & abstraction**

Open to more institutions (mid-Sept to mid-Dec 2010):

http://se.ethz.ch/dose

1. Definitions: programming and software engineering
2. Lessons from experience: teaching programming
3. Lessons from experience: teaching software engineering
4. Lessons: general

Hindering SE teaching & research

(More on these issues on my blog, bertrandmeyer.com)

1. **No systematic postmortem on software disasters**

   such as:

   Ariane 5

   (Lions/Kahn, see Jézéquel & Meyer)

   Tokyo Stock Exchange

   (Tetsuo Tamai, *Social Impact of Information System Failures*, IEEE *Computer*, June 09)

2. **Difficulty of funding programmer positions**

3. **Need better empirical software engineering**

General lessons learned

1. Whatever we teach should be falsifiable
2. Let us not lower our intellectual guards
3. Tools and languages matter
4. Teach skills supporting concepts
5. Technology is key
6. Programming is at the center of software engineering
7. We are still at the beginning, but should be proud

se.ethz.ch (chair)

touch.ethz.ch (intro textbook)

se.ethz.ch/dose (distributed course)

bertrandmeyer.com (blog)

eiffel.com (languages & tools)

Eiffel Software

ETH Chair of Software Engineering