Обширный подход к преподаванию программной инженерии

Bertrand Meyer
ETH Zurich, ITMO & Eiffel Software

X Всероссийская конференция
"Преподавание информационных технологий в Российской Федерации"
Москва, 17-ого Мая 2012

Content

1. Challenges of teaching programming & software engineering
2. Our method at ETH: Outside-in, *Touch of Class* textbook
3. Why use Eiffel?
4. Some other courses
5. Software Engineering Laboratory at ITMO

LASER summer school, 2012

Ivar Jacobson: UML

Martin Odersky: Scala)

Andrei Alexandrescu: (C++ & D)

Eric Meijer: C# & Linq

Simon Peyton-Jones: Haskell

Guido van Rossum: Python

Bertrand Meyer: Eiffel

Elba Island, Italy, 2-8 septembre 2012

http://laser.inf.ethz.ch

- 1 -

Challenges of teaching programming & software engineering

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
- Classification
- Complexity & impossibility
- Reuse
- Algorithmic reasoning
- Function vs data
- Typing
- Scaling up
- Complexity
- Static vs dynamic
- Invariant

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.

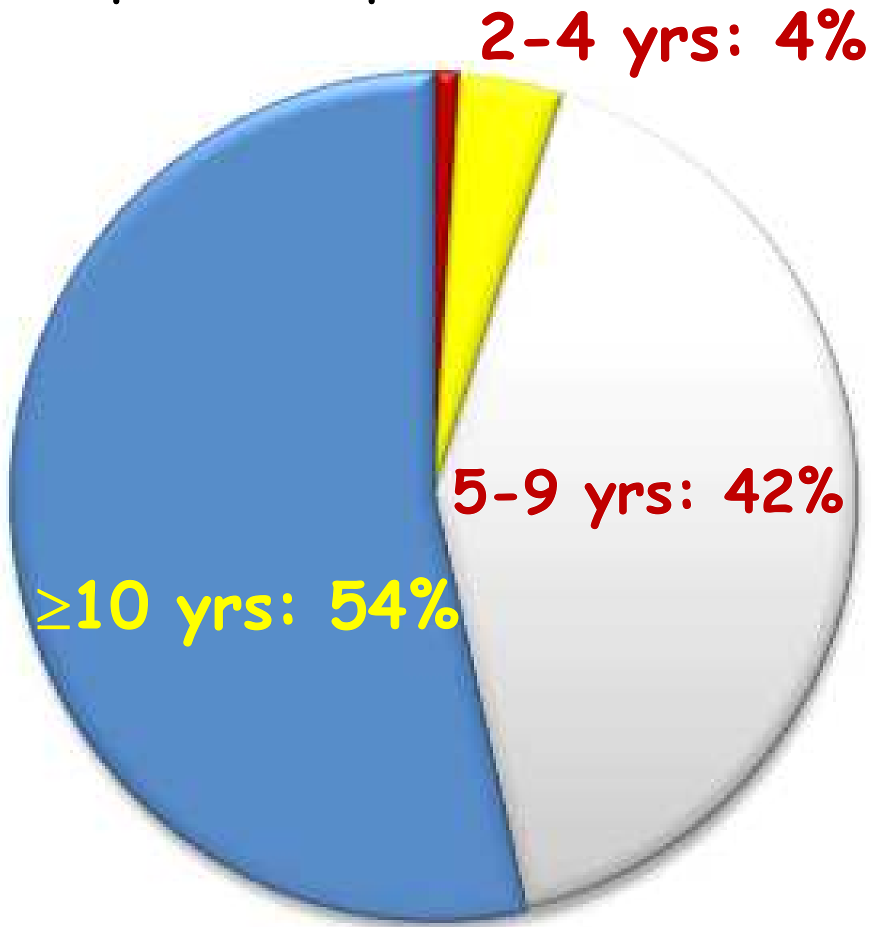
As surely as farmers complain about the weather, computing academics will complain about students' programming abilities.

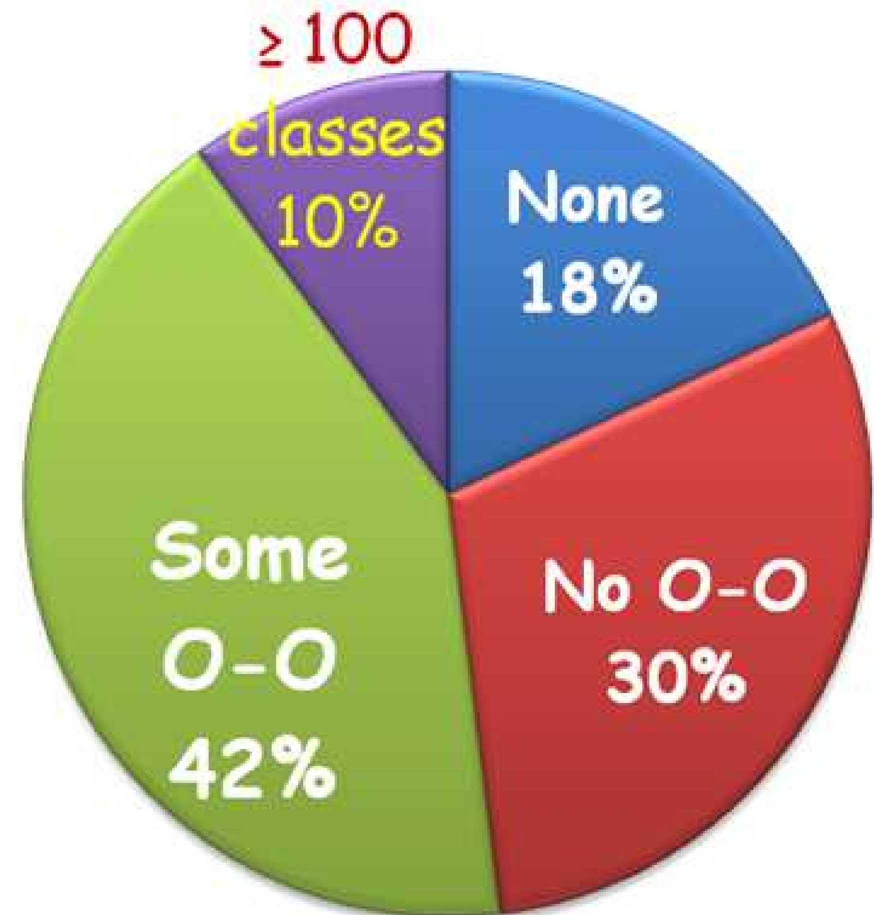
Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*, 10th Conf. on Australasian computing education, 2008

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





Averages over 6 years, 2003-2008 (yearly variations small)

Full year-by-year figures: Pedroni, Meyer, Oriol, *They know more than we think!*, CACM Blog, 2012

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

- 2 -

Our method at ETH: Outside-in, *Touch of Class* textbook

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

Scaling up

Typing

Complexity

Static vs dynamic

Invariant

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**!

The idea of a loop

Possible solutions

Exit condition

INVARIANT

Body

Previous state of computation

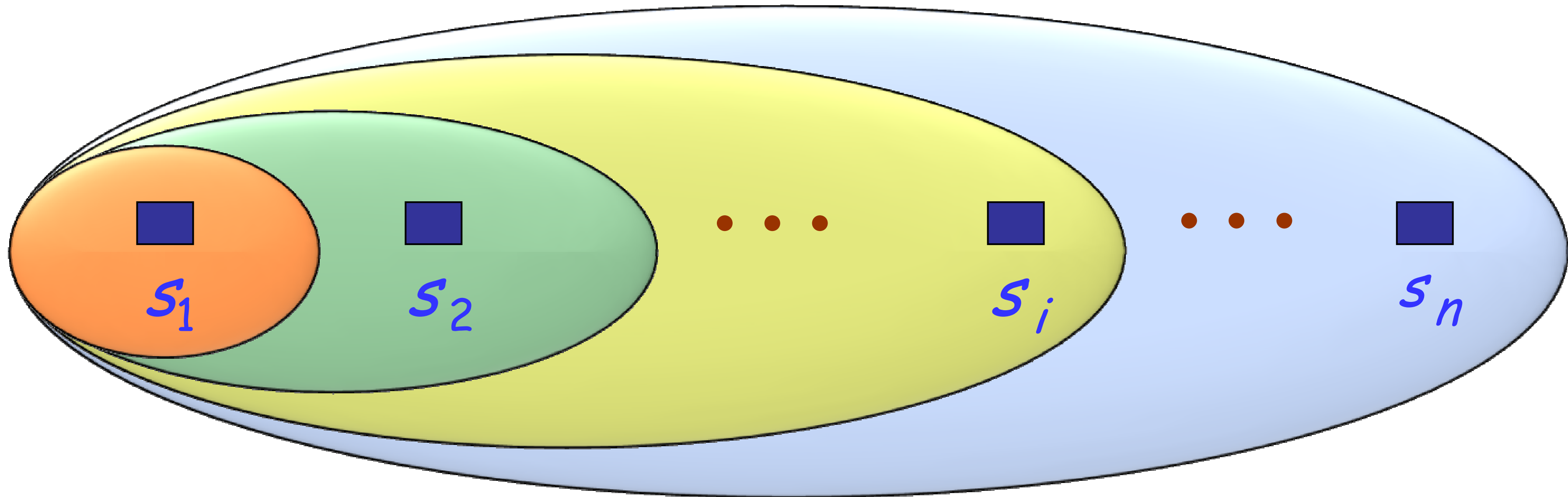
Body

Initialization

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

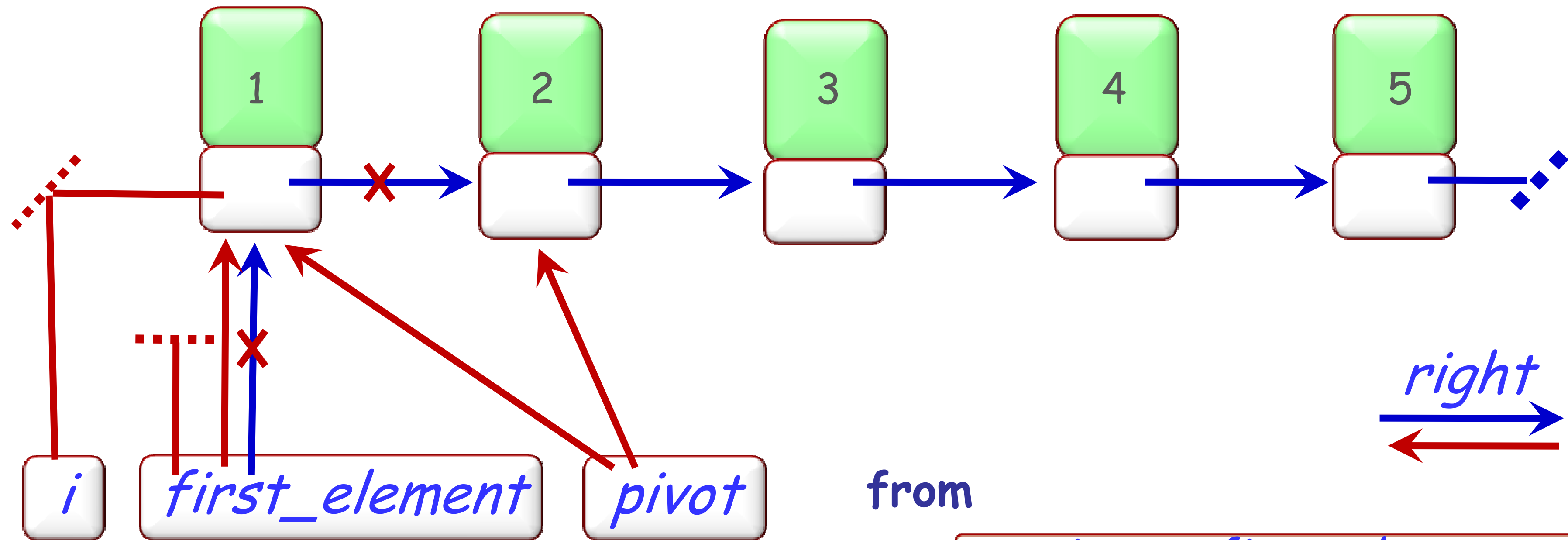
Result := *max* (**Result** , s_i)

Result = Max (s_1 .. s_i)

The loop invariant

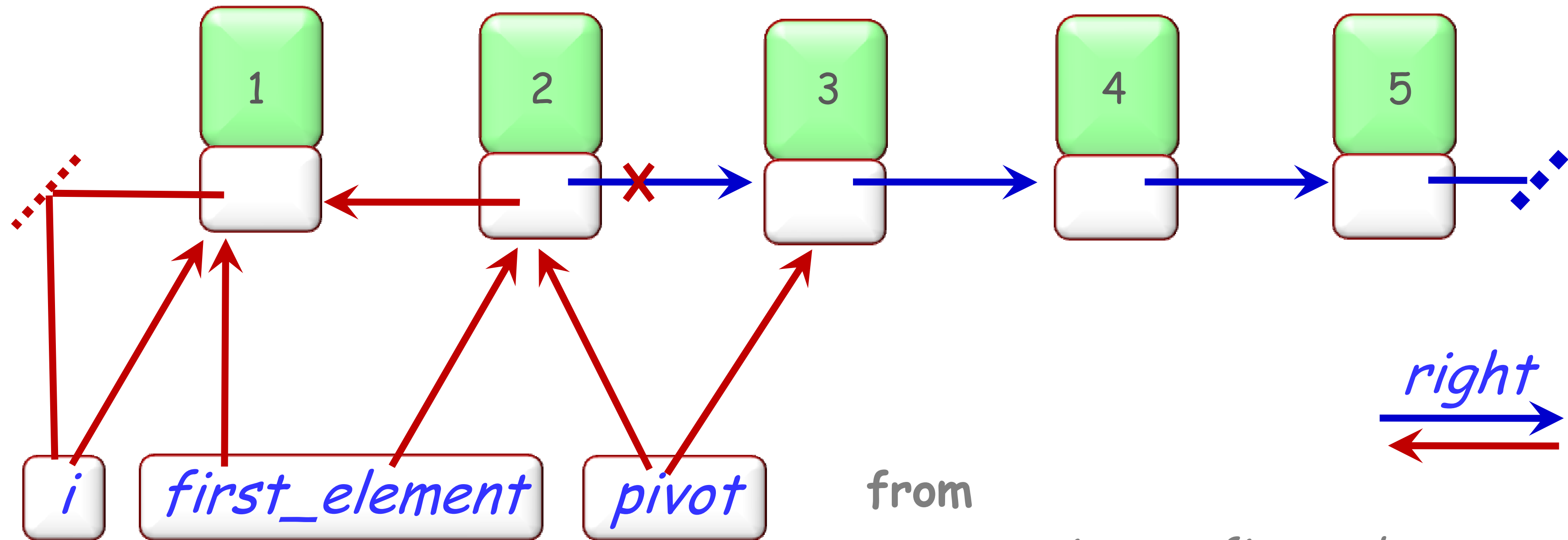
Result = Max (s_1 .. s_n)

Reversing a list



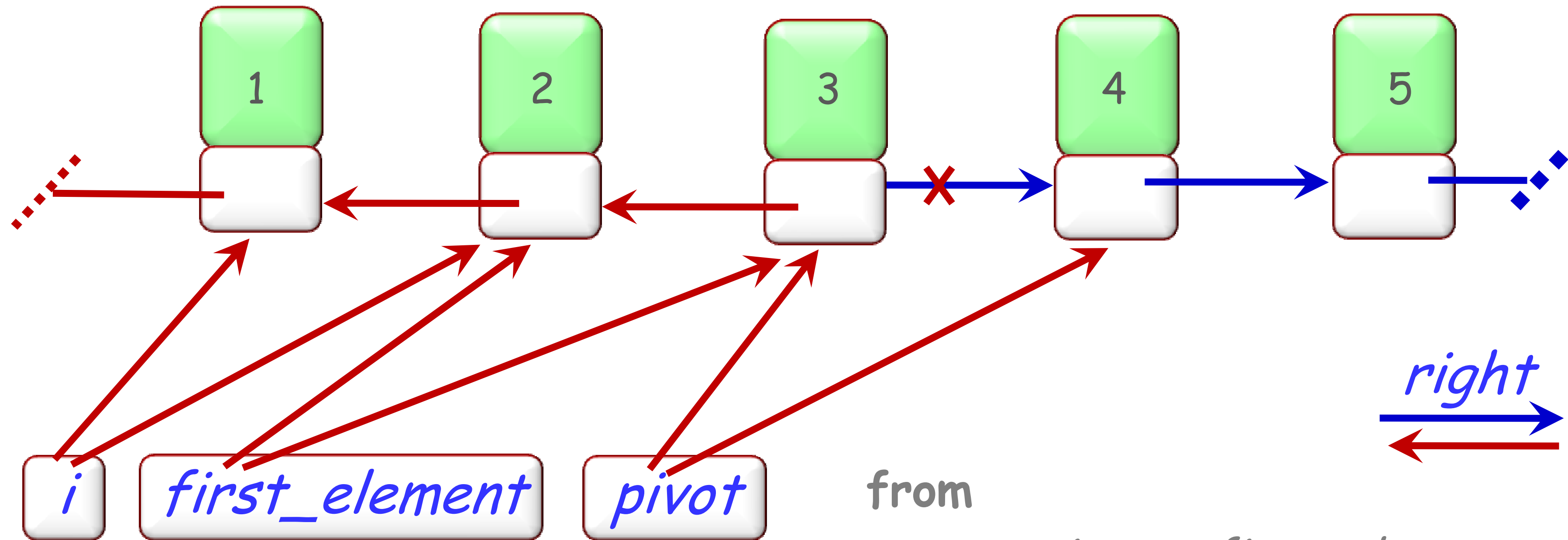

```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list



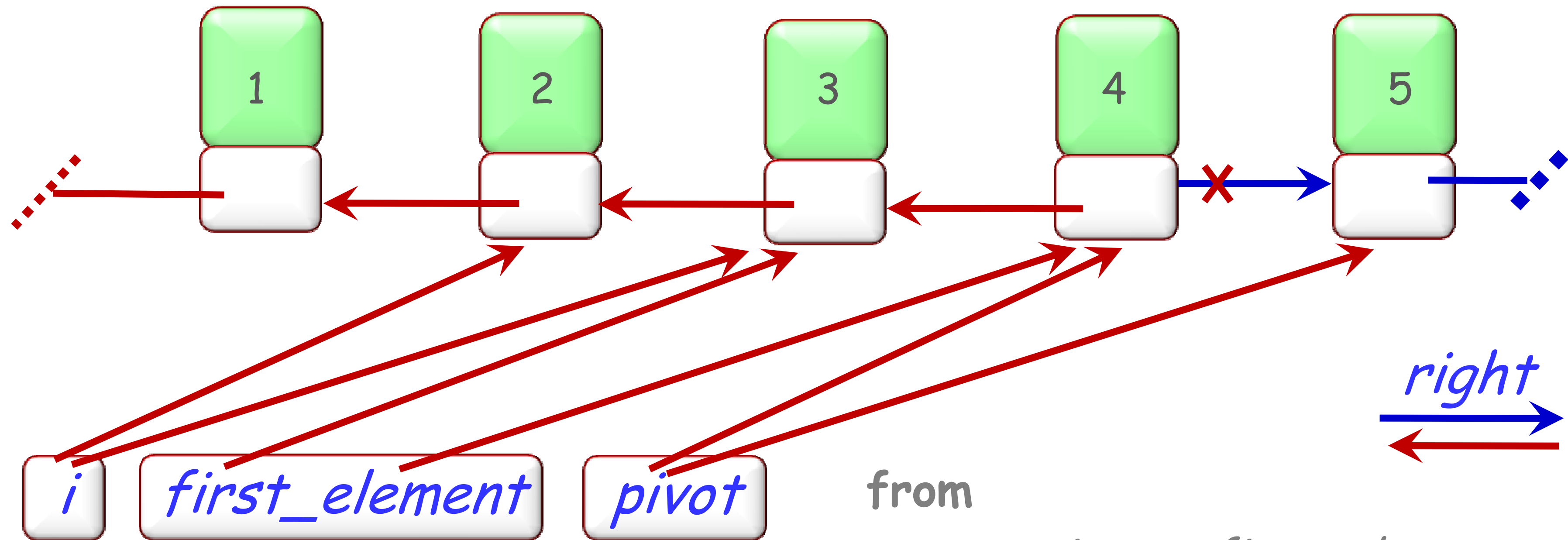

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



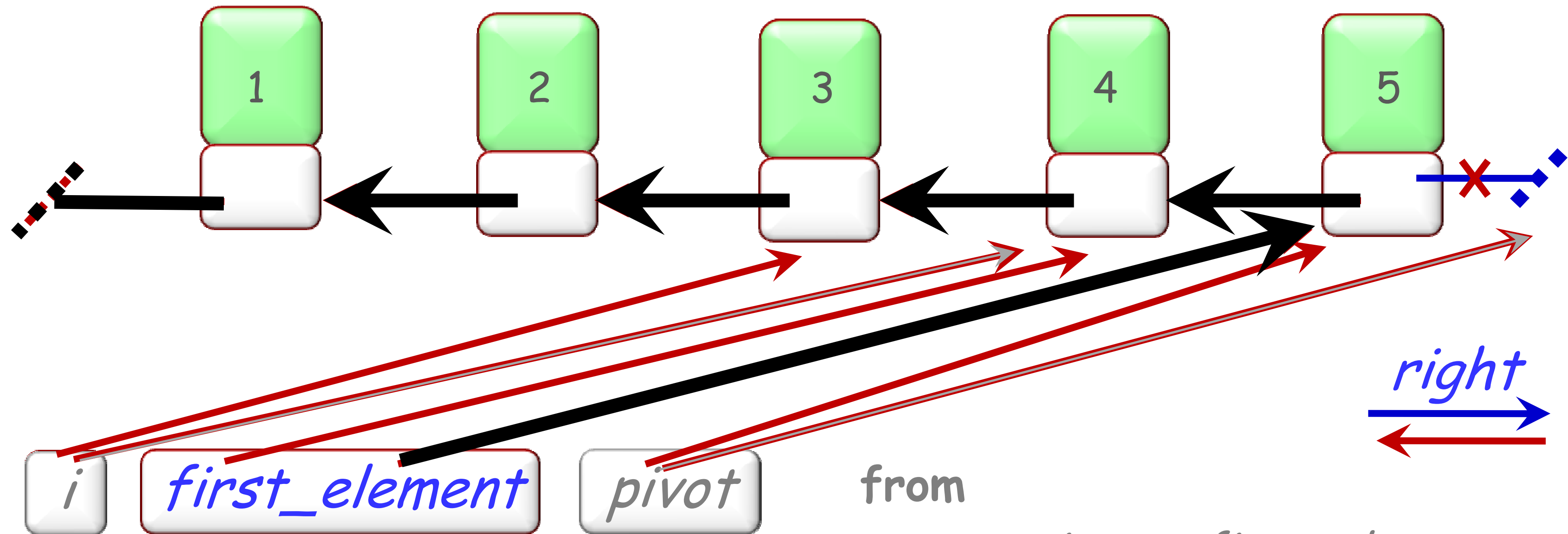

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

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?

1 2 3 4 5

i *first_element* *pivot*

right

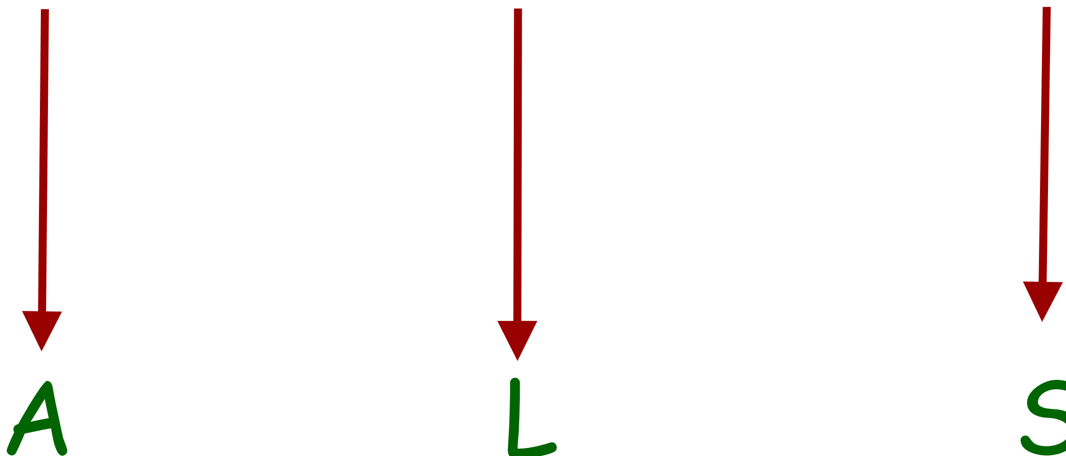
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

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

Levenshtein distance

"Beethoven" to "Beatles"

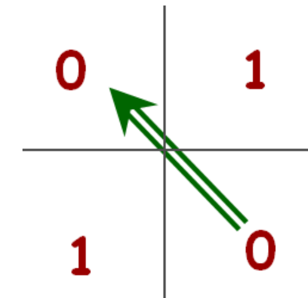
B E E T H O V E N

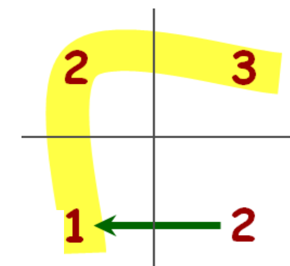


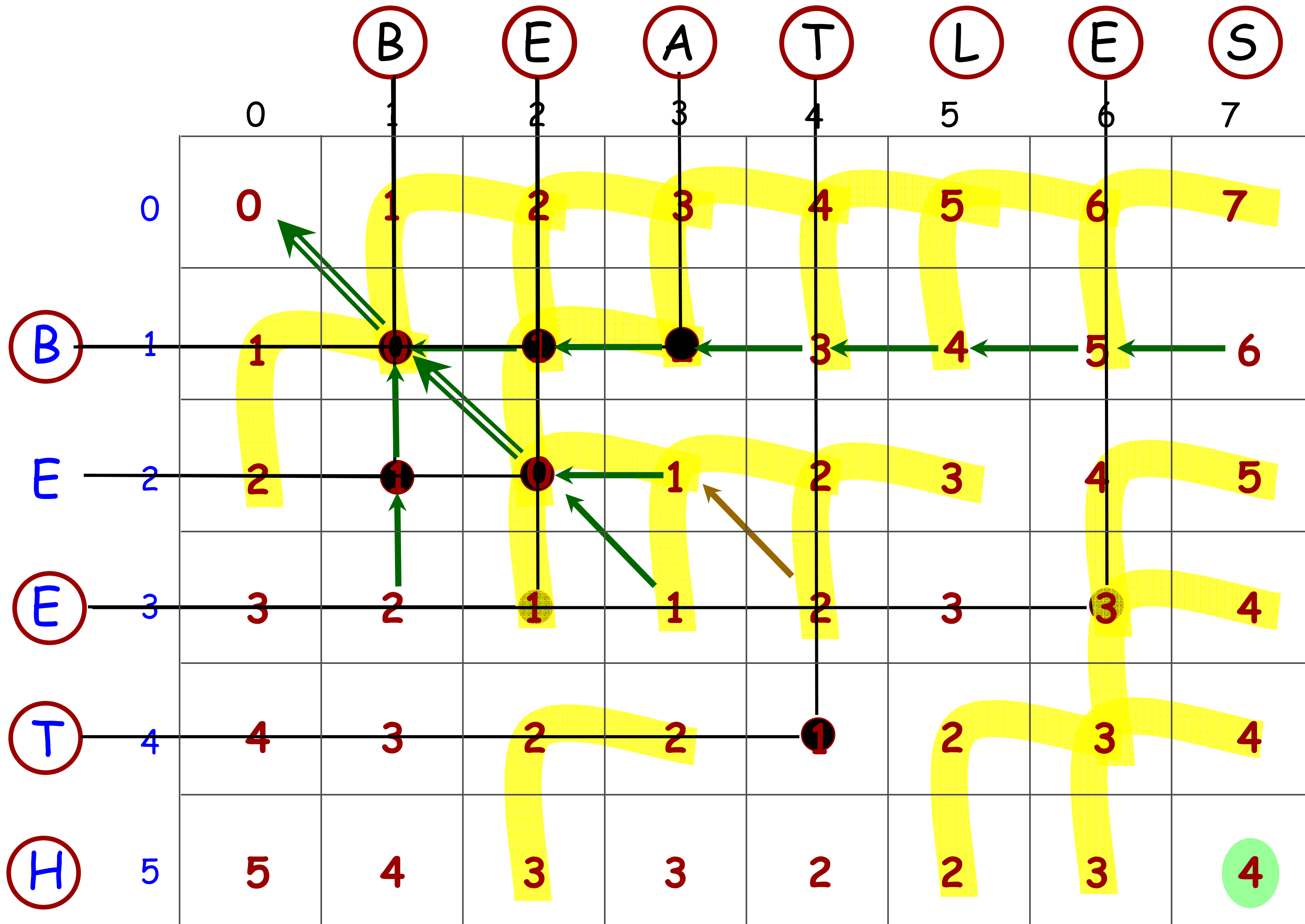
Operation	–	–	R	–	D	R	D	–	R
Distance	0	0	1	1	2	3	4	4	5

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]
```



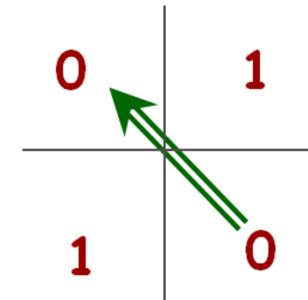


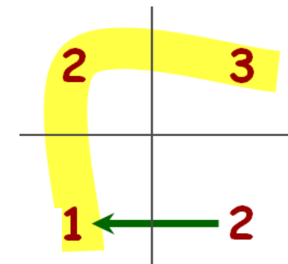


D

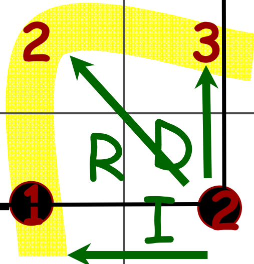
Levenshtein algorithm

```
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* $[1..j]$

I ← Insert

D ↑ Delete

R ↖ Replace

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, Nadia Polikarpova & students

≈ 150,000 lines of Eiffel

"Progressive opening of the black boxes"

TRAFFIC is graphical, multimedia and extendible

(Approach 3: teaching a specific language)

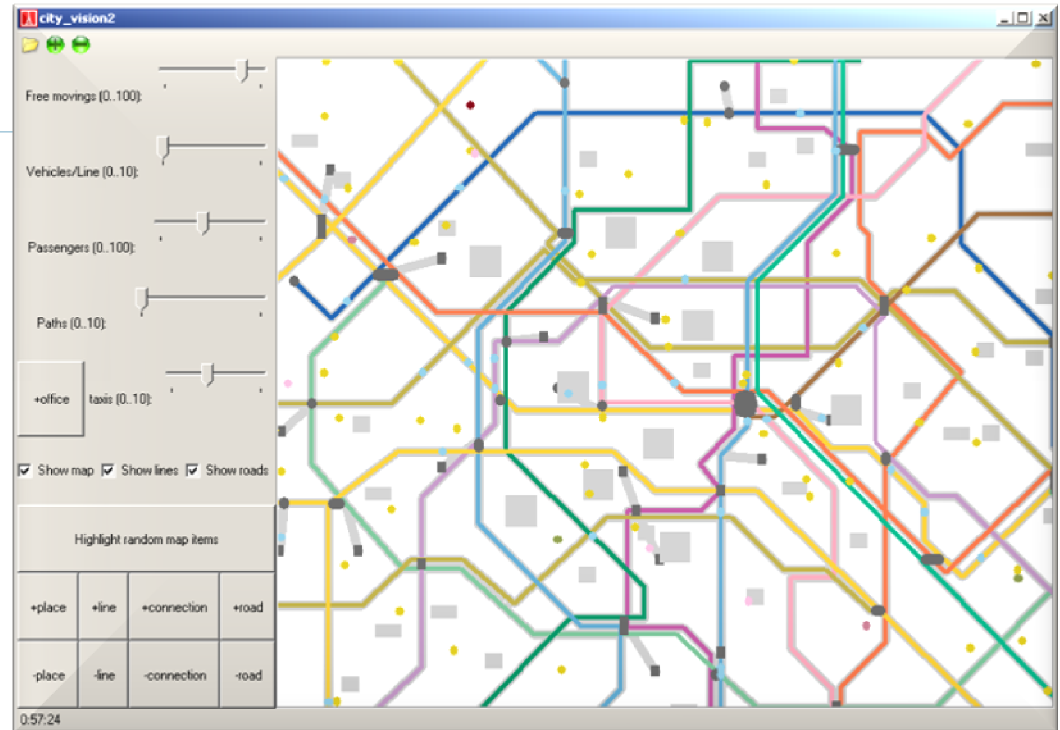
First Java program:

```
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

City TimeTable

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

Load buildings

Delete buildings

Zoom in

Zoom out

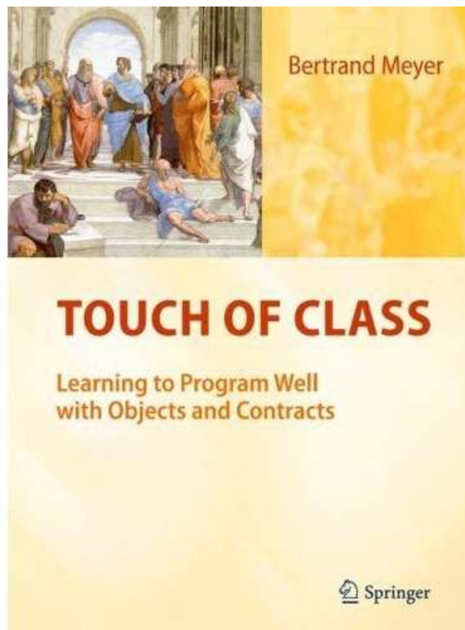
Show sun

Show buildings

Simulate time

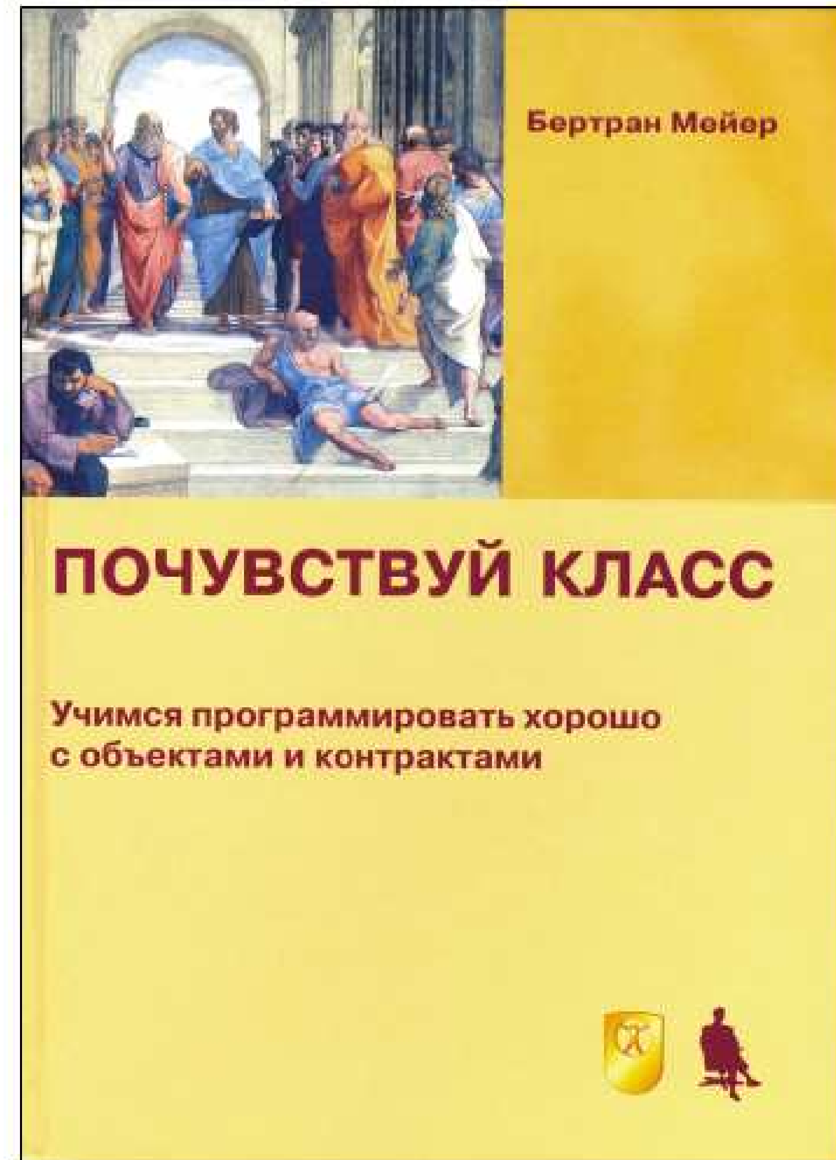
Supporting textbook

touch.ethz.ch



Springer, 2009

«ИНТУИТ»: БИНОМ. Лаборатория знаний, 2011



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…

- 3 -

Why use Eiffel?

Eiffel: the negatives

No major industry power behind Eiffel

- *But: has made its mark anyway*

Smaller community

- *But: active, competent and enthusiastic*

Not hyped, old technology that did not make it

- *But: has proved to be more than a fad*

Not enough reusable components

- *But: easy to interface with e.g. C, C++*

Negatives that are not

"*All the best ideas will appear in my favorite language anyway*"

- Answer: have you tried to use Code Contracts?

"*There are no Eiffel programmers to be found*"

- Answer: Eiffel as a language has no tricks or mysteries. Training someone in Eiffel means teaching them sound O-O software engineering
- Lots of graduates with Eiffel experience (e.g. ETH)

"*What about this covariance thing?*"

- Answer: the problem has been solved

"*Not good for developing Web applications!*"

- Partly true until recently, but no longer with EWF

Eiffel: the reality

Method, language, environment

- **Language**: ISO standard since 2006
- **Method**: comprehensive set of software engineering principles
  - Extends across the entire lifecycle
  - Directly supported by the language
- **Environment**: comprehensive set of tools, available across numerous platforms

What is object-oriented programming?

Applying the concept of abstract data type (ADT)

Objects, not operations, are the structuring criterion

Organize programs as combinations of object types

Every object type characterized by:

- Applicable operations (commands & queries)
- Properties of these operations

Organize these types into hierarchies (inheritance)

"Ask not what the system does, ask what it does it to"

Eiffel: why?

Extendibility

Reliability (Design by Contract™, strong typing, covariance, exception handling…)

Scalability and maintainability

Integrated Method + Language + Environment

Portability + Performance

Eiffel: the reality

Method, language, environment

- **Language**: ISO standard since 2006
- **Method**: comprehensive set of software engineering principles
  - Extends across the entire lifecycle
  - Directly supported by the language
- **Environment**: comprehensive set of tools, available across numerous platforms

Eiffel for teaching

Similar & different!

Simple, not pitfalls, easy syntax (e.g. semicolon is optional)

"One good way to do anything"

Very powerful constructs (agents, loops...)

Emphasis on interfaces and information hiding

e.g. no `x.a := v`

Students like it!

Some principles of the Eiffel method

- Abstraction (based on abstract data types)
- Information hiding
- Seamlessness
- Reversibility
- Design by Contract
- Open-Closed principle
- Single choice principle
- Single model principle
- Uniform access principle
- Command-query separation principle
- Option-operand separation principle
- Style matters

Design by Contract: applications

- Getting the software right
- Analysis
- Design
- Implementation
- Debugging
- Testing
- Exception handling
- Using inheritance properly
- Management
- Maintenance
- Documentation

Design by Contract: the basic idea

Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users)

This goal is the element's contract

The contract of any software element should be

- Explicit
- Part of the software element itself

Contracts are expressed through

- Routine preconditions
- Routine postconditions
- Class invariants
- Loop invariants etc.

Eiffel: the language

- Classes
- Uniform type system, covering basic types
- Strongly typed, **void-safe**
- Genericity
- Agents: event-driven, functional etc. programming
- Inheritance, single and multiple
- Conversion
- Covariance
- Built-in **Design by Contract** mechanisms
- "Once" mechanisms, replacing statics and globals

Void safety

Null pointer dereferencing

x.f (...) with x void (null)

is **not** possible in Eiffel

Basics of the approach:

- Types "attached" by default
- If type detachable, it must be certain that any call x.f (...) will always be applied to non-void x

- 4 -

Some other courses

Teaching software engineering

Basic courses:

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

Advanced courses:

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

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, Rio Cuarto (Argentina)

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):

http://se.ethz.ch/dose

- 5 -

The Software Engineering Laboratory at ITMO

SEL at ITMO

Software Engineering Laboratory |

Лаборатория Программной Инженерии

Создана в июне 2011

"Мегагрант" с финансовой поддержкой компании **mail.ru**

Остаются открытые позиции!

- Аспиранты и Кандидаты (на полной ставке)
- Временные гранты ("sabbaticals") для исследователей, 2 до 6 месяцев

ITMO-SEL: mode of working

Reach for the highest international standards

Publish only in the best international venues

Collaborate closely with ETH team

Be open to the external world

- Microsoft Summer School on concurrency (Aug. 2012)
- European Software Engineering Conference (ESEC, Aug. 2013)
- Weekly seminar http://sel.ifmo.ru/seminar

Try for the best!

The scientific goal

Produce an environment where verifying software is part of the everyday, normal experience

"VAMOC": Verification As A Matter of Course

EVE: Verification As a Matter Of Course

AutoProof

Inter. prover

Alias analysis

Sep. logic prover

Invariant inference

Suggestions

Suggestions

EVE

Arbiter

AutoTest

Test case generation

Test execution

Test results

AutoFix

Fix generation

Topics and projects

Static verification:

- Proofs: Boogie etc.
- Alias calculus
- Calculus of Object Programs

Dynamic verification:

- Fully automatic testing (AutoTest)
- Fix suggestions (AutoFix)

Practical specification

- Full contracts (MML)

Concurrency

- SCOOP

General lessons learned

1. Reach for the highest intellectual goals
2. Tools, technology and especially languages matter
3. Teach skills supporting concepts
4. The goal of software engineering is quality

sel.ifmo.ru (ITMO lab)

se.ethz.ch (ETH chair)

touch.ethz.ch (intro textbook)

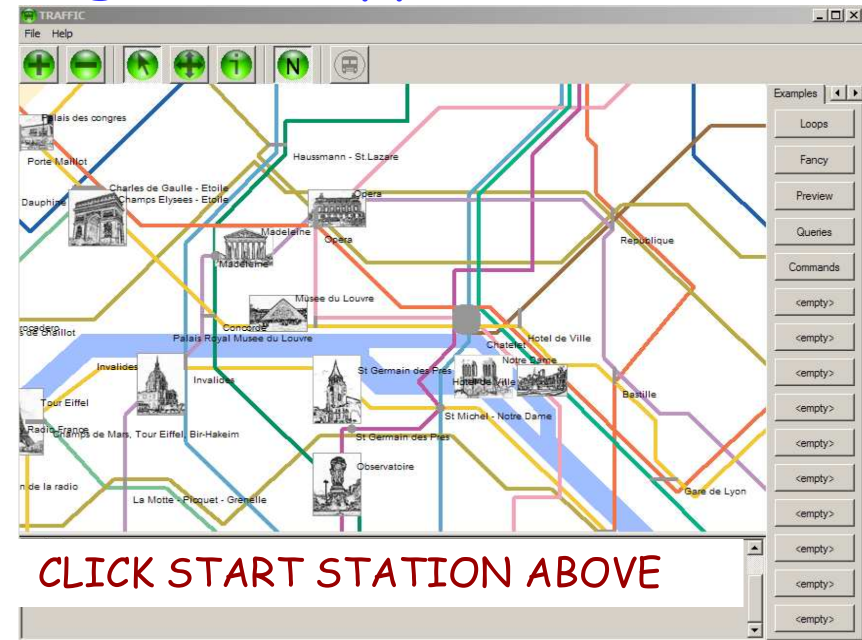
se.ethz.ch/dose (distributed course)

www.bertrandmeyer.com (blog)

eiffel.com (languages & tools)

Agents

How to program an event-driven (e.g. GUI) application in Eiffel:



1. Define event type, e.g. left_click

2. Subscriber:

```
map.left_click.subscribe (agent find_city)
```

3. Publisher:

```
left_click.publish ([x, y])
```

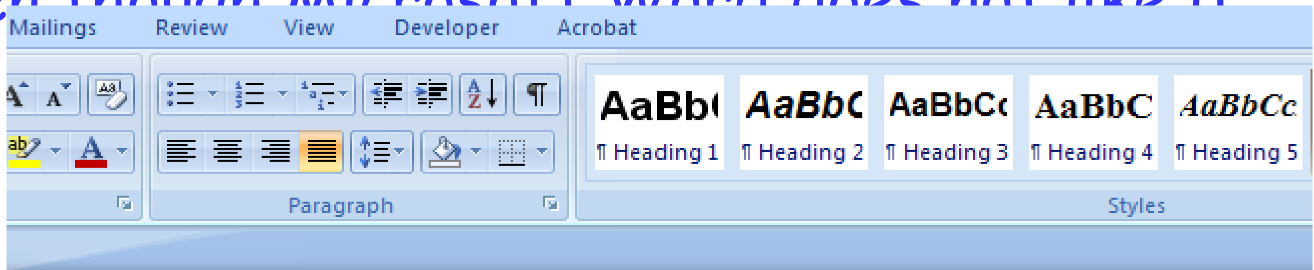
Multiple inheritance

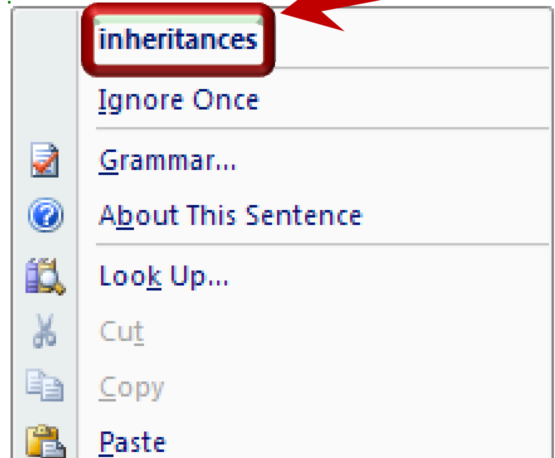
Forget all you have heard!

Multiple inheritance is **not** the works of the devil

Multiple inheritance is **not** bad for your teeth

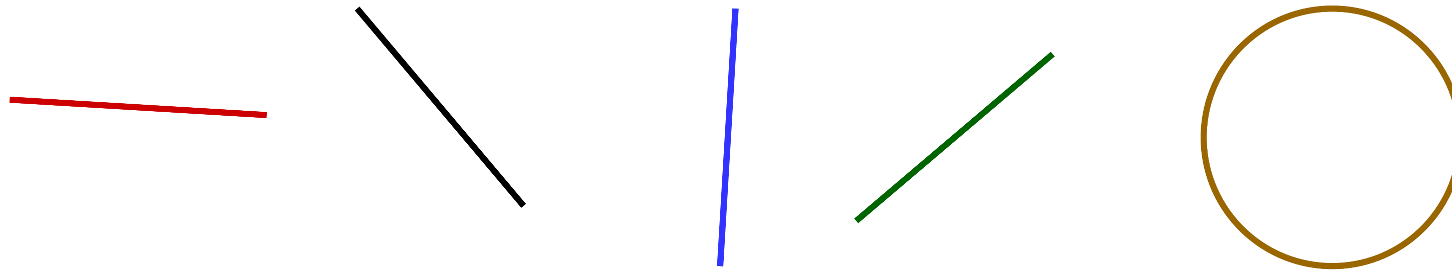
(Even though Microsoft Word does not like it:



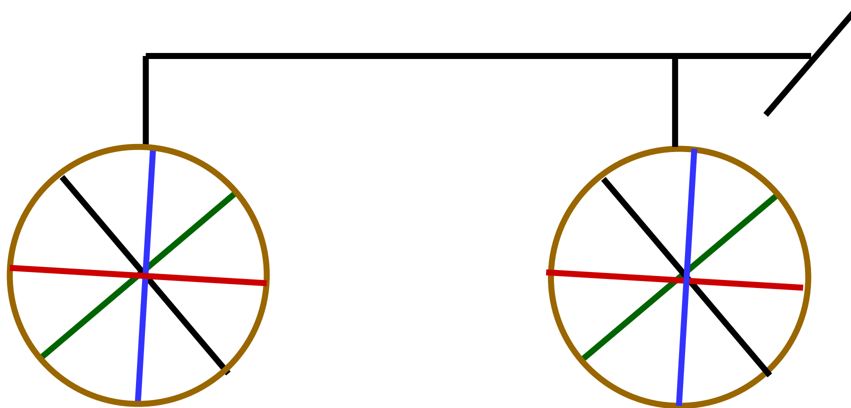


)

Multiple inheritance: Composite figures

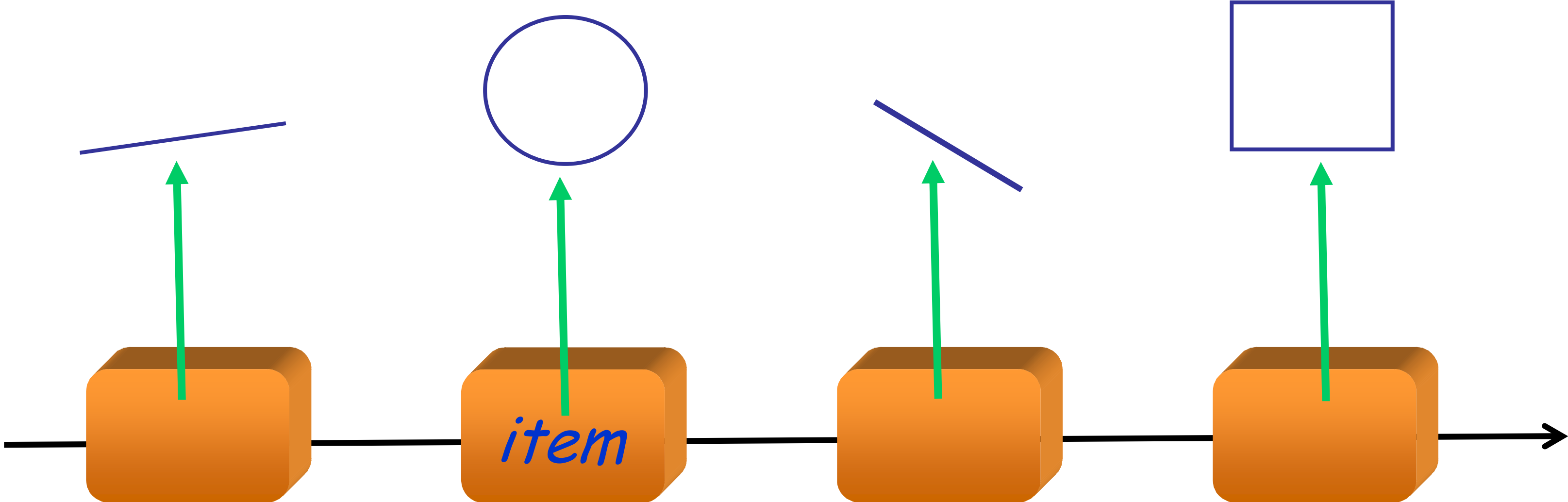


Simple figures



A composite figure

A composite figure as a list



Defining the notion of composite figure

center
display
hide
rotate
move
…

FIGURE

LIST [FIGURE]

count
put
remove
…

COMPOSITE_FIGURE

Composite figures

```
class COMPOSITE_FIGURE inherit
        FIGURE
        LIST [FIGURE]
feature
        display
                -- Display each constituent figure in turn.
        do
                from start until after loop
                        item.display
                        forth
                end
        end
        ... Similarly for move, rotate etc. ...
end
```

Requires dynamic binding