From programming to software engineering:

Notes of an accidental teacher

Bertrand Meyer
ETH Zurich & Eiffel Software
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
- Recursive reasoning
- Structure
- Classification
- Complexity & impossibility
- Reuse
- Complexity & impossibility
- Function vs data
- Algorithmic reasoning
- Typing
- Scaling up
- Static vs dynamic
- Invariant
- Complexity
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”:

- **Description**: specify (requirements, systems, designs, implementations...) and document
- **Implementation**: build the products (includes both programming & design)
- **Assessment**: verify, validate, analyze, test, measure (both products and processes)
- **Management**: organize the work, communicate, collaborate
- **Operation**: deploy systems and oversee their proper functioning
- **Notation**: 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.

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

Computer experience

- ≥10 yrs: 54%
- 5-9 yrs: 42%
- 2-4 yrs: 4%

Programming experience

- Some O-O: 42%
- No O-O: 30%
- None: 18%
- ≥ 100 classes: 10%


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
- Recursive reasoning
- Syntax vs validity vs semantics
- Complexity & impossibility
- Algorithmic reasoning
- Typing
- Invariant
- Scaling up
- Static vs dynamic
- Complexity
- Structure
- Change
- Reuse
- Function vs data
- Classification
- Typing
- Static vs dynamic
- Complexity
- Structure
- Change
- Reuse
- Function vs data
- Classification
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

Result := max (Result, i.item)
end
Loop as approximation strategy

Result = \( a_1 \) = Max (\( S_1 \ldots S_1 \))

Result = Max (\( S_1 \ldots S_2 \))

Result = Max (\( S_1 \ldots S_i \))

Result = Max (\( S_1 \ldots S_n \))

Loop body:

\[ i := i + 1 \]

Result := max (Result, \( S_i \))

The loop invariant
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

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

\[\text{pivot} := \text{first\_element}\]
\[\text{first\_element} := \text{Void}\]
\[\text{until } \text{pivot} = \text{Void} \text{ loop}\]
\[i := \text{first\_element}\]
\[\text{first\_element} := \text{pivot}\]
\[\text{pivot} := \text{pivot\_right}\]
\[\text{first\_element\_put\_right}(i)\]
\[\text{end}\]
Reversing a list

$\text{pivot} := \text{first}_{-}\text{element}$

$\text{first}_{-}\text{element} := \text{Void}$

until $\text{pivot} = \text{Void}$ loop

$i := \text{first}_{-}\text{element}$

$\text{first}_{-}\text{element} := \text{pivot}$

$\text{pivot} := \text{pivot}.\text{right}$

$\text{first}_{-}\text{element}.\text{put}_{-}\text{right}(i)$

end
Reversing a list

\[
pivot := first_element \\
first_element := Void \\
until pivot = Void loop \\
i := first_element \\
first_element := pivot \\
pivot := pivot.right \\
first_element.put_right(i) \\
end
\]
Why does it work?

Invariant: from \texttt{first_element} following \texttt{right}, initial items in inverse order; from \texttt{pivot}, rest of items in original order

\texttt{pivot} := \texttt{first_element} \\
\texttt{first_element} := \texttt{Void} \\
\texttt{until pivot} = \texttt{Void} \texttt{loop} \\
\quad \texttt{i} := \texttt{first_element} \\
\quad \texttt{first_element} := \texttt{pivot} \\
\quad \texttt{pivot} := \texttt{pivot.right} \\
\texttt{end} \\
\texttt{first_element.put_right(i)}
Levenshtein distance

“Beethoven” to “Beatles”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>R</td>
<td>5</td>
</tr>
</tbody>
</table>

Operation:
- Insertion
- Deletion
- Substitution
Levenshtein algorithm

across \( r \): 1 |..| rows as \( i \) loop

across \( c \): 1 |..| columns as \( j \) invariant

\[
\begin{align*}
\text{if } & \text{source}[i] = \text{target}[j] \text{ then} \\
D[i, j] & := D[i-1, j-1] \\
\text{else} & \\
D[i, j] & := 1 + \\
& \min(D[i-1, j], D[i, j-1], D[i-1, j-1])
\end{align*}
\]

end

Result := \( D[\text{rows, columns}] \)
### Invariant: each $D[i, j]$ is distance from source $[1..i]$ to target $[1..j]$

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>T</th>
<th>L</th>
<th>E</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Actions:**
- **Insert (I):** Move up.
- **Delete (D):** Move left.
- **Replace (R):** Move diagonally up and left.

- **Example:**
  - From the green cell (4, 3) to the yellow cell (2, 2) involves an **Insert (I)** and a **Delete (D)**, which together form an **Replace (R)**.
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!

“Progressive opening of the black boxes”

TRAFFIC is graphical, multimedia and extendible

Michela Pedroni & numerous students
≈ 150,000 lines of Eiffel
(Approach 3: teaching a specific language)

First Java program:

class First {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
Our first “program”

```ruby
class PREVIEW inherit TOURISM
  feature explore

  -- Prepare & animate route
  do
    Paris.display
    Louvre.spotlight
    Metro.highlight
    Route1.animate
  end
end
```

Text to input
Supporting textbook

touch.ethz.ch

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

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)
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
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)