On Abstraction and Informatics

Presented at *ISSEP 2011*
26 October 2011, Bratislava, Slovakia

Tom Verhoeff
Eindhoven University of Technology
Dept. of Mathematics & Computer Science
Software Engineering & Technology

www.win.tue.nl/~wstomv

T.Verhoeff@tue.nl

My Goal

Get

more, better, and *explicit* attention for

abstraction *itself*,

in the informatics curriculum.

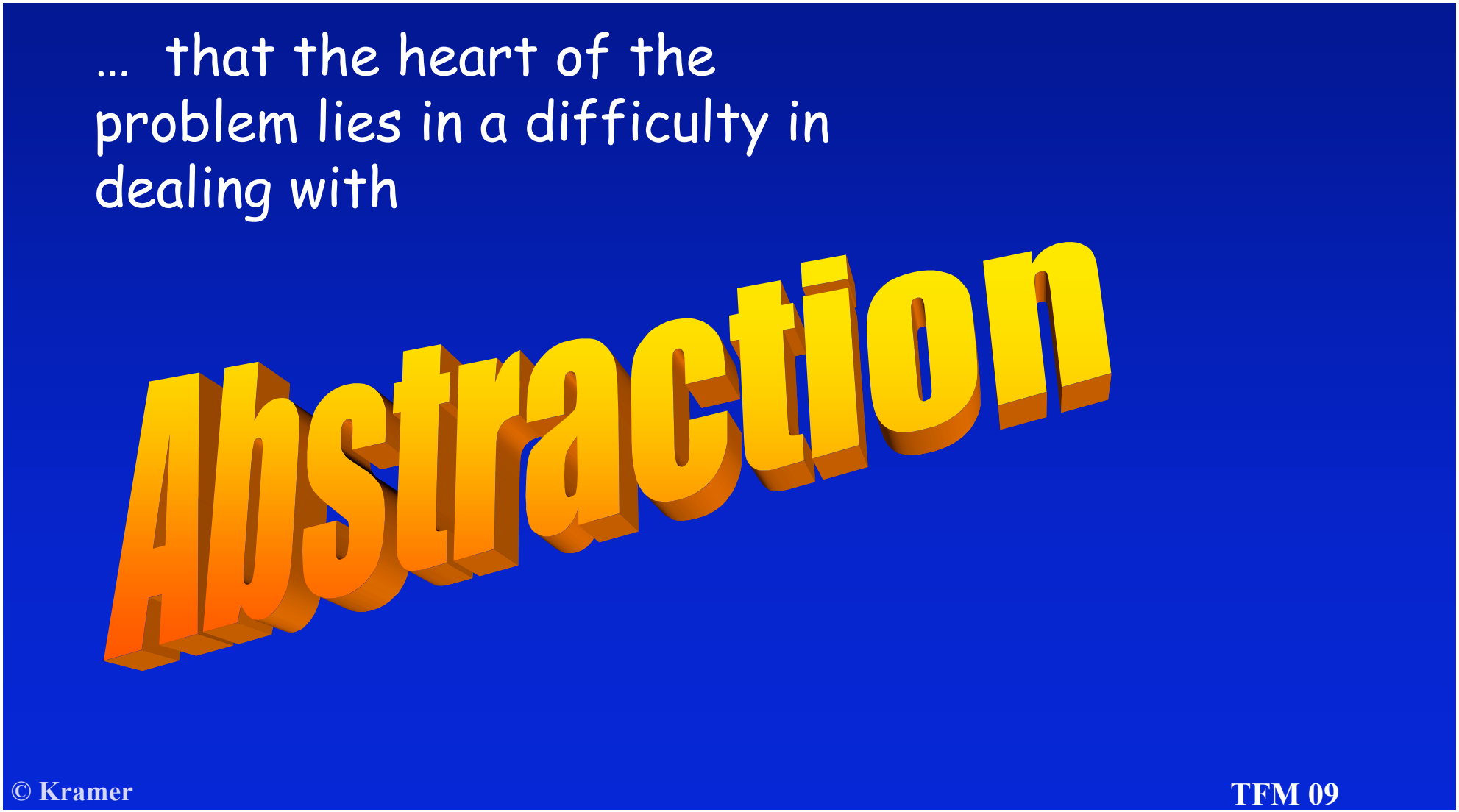
Abstraction and Modelling
a complementary partnership

Jeff Kramer

Imperial College
London

"Is Abstraction the key to Computing?" CACM April 2007

Jeff Kramer believes . . .



Why is Abstraction Important in Informatics?

Software is abstract!

"Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner."

Keith Devlin CACM Sept.2003

© Kramer

TFM 09

Jeannette Wing on Computational Thinking (2008)

"The essence of computational thinking is *abstraction*.

In computing, we abstract notions beyond the physical dimensions of time and space.

Our abstractions are extremely general because they are symbolic, where numeric abstractions are just a special case.

. . . [O]ur abstractions tend to be richer and more complex than those in the mathematical and physical sciences."

Abstraction in Informatics Curriculum

- Underexposed
- Not treated well enough
- Implicit

Students sometimes (often?) get the (mistaken) impression that

abstraction = vagueness and imprecision

Informatics teachers often do not know either . . .

Abstraction in Mathematics

- How to explain what a fraction is to someone who does not know?
- Start overspecific: a pair of integers (a, b) with $b > 0$
- Equivalence relation abstracts from unintended distinctions:

$$(a, b) \sim (c, d) \iff ad = bc$$

- Define fractions by *dividing out* the equivalence:

$$\mathbb{Q} \;=\; \left(\mathbb{Z} \times \mathbb{Z}^{+}\right) / \sim$$

- A fraction, in the abstract, is an equivalence class in $\left(\mathbb{Z} \times \mathbb{Z}^{+}\right)$ under the equivalence relation $\sim$

Abstractions in Programming: Client-Server Contracts

Procedural abstraction applied to compute 100^{th} Fibonacci number:

```
{ Contract:  precondition 0 <= n;  return n-th Fibonacci number }
function fib ( n : integer ) : integer;
    begin ... { implementation omitted } ... end;

begin
    writeln ( fib ( 100 ) )
end.
```

- **Client** calls (invokes) the function
- **Server** implements the function
- **Contract** is the *abstraction*: binds/(de)couples client and server

Procedure Abstraction involves multiple abstractions

- *Implementation details* hidden from client

  To use the abstraction (at design time), the client designer does not need to know about the implementation.

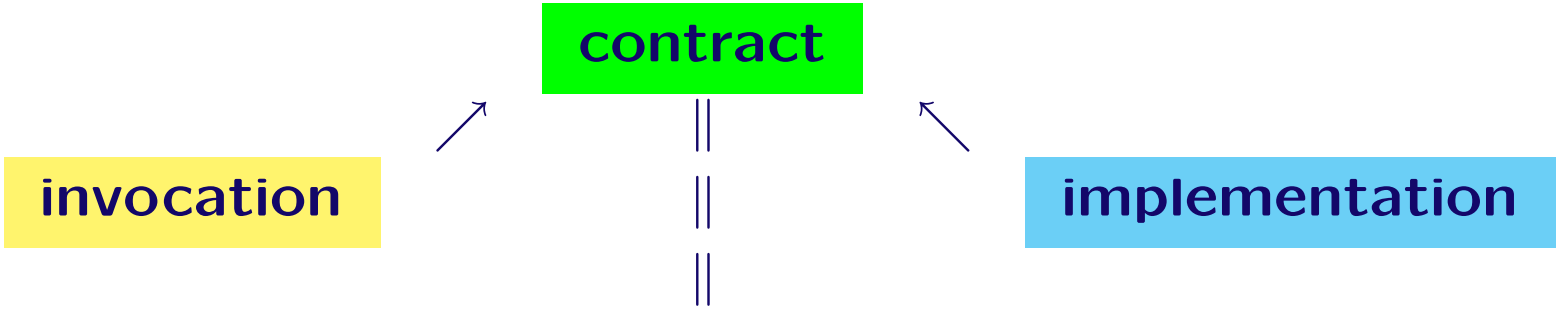
- *Identity/value of things operated on* hidden from server

  To implement the abstraction (at design time), the server implementer does not need to know where the data is and what its value is; the parameters abstract from that.

- *Context of usage* hidden from server

  Neither does the server implementer need to know in what context the facility is used.

Reasoning through contracts is central to Divide and Conquer



- Reason about invocation (call) in terms of contract, and reason about implementation in terms of (same) contract.
- *Never* reason about invocation and implementation together. Thus, 'divide' fails, leading to complexity, and not to 'conquer'.

Also applies to Data Abstraction, Iteration Abstraction, . . .

Recursion is hard to master without contractual reasoning.

Musical Intermezzo

Jacob van Eyck (approx. 1590 – 1657)

Dutch carilloneur (church bell player) and recorder player/composer

Prelude for recorder

Teaching Abstraction: Forward Chaining

In the development process and in forward-chaining, you (learn to)

1. Draw up a contract for an abstraction
2. Validate the contract
3. Design/implement it (can be split)
4. Review the design/implementation
5. Test the implementation
6. Use the abstraction

Teaching Abstraction: Backward Chaining

In backward-chaining, you learn to

1. Use an abstraction, given its contract
2. Test an abstraction, given its contract
3. Review a given design/implementation
4. Design/implement a given contract
5. Validate contracts
6. Draw up a contract for an abstraction

Conclusion

- Abstraction needs *explicit* attention in informatics curriculum
- Abstraction has many facets, requiring appropriate terminology and teaching methods
- Backward chaining can be used for teaching abstraction
- Needs follow-up research, e.g. definition of TRUCs for abstraction

  TRUC = Testable, Reusable Unit of Cognition (Meyer, 2006)

For details, see my article in the proceedings (on CD-ROM), or at

`www.win.tue.nl/~wstomv/publications/issep-2011-on-abstraction.pdf`