

# Software Development via Algebraic Specification

Karel Richta

[richta@fel.cvut.cz](mailto:richta@fel.cvut.cz)

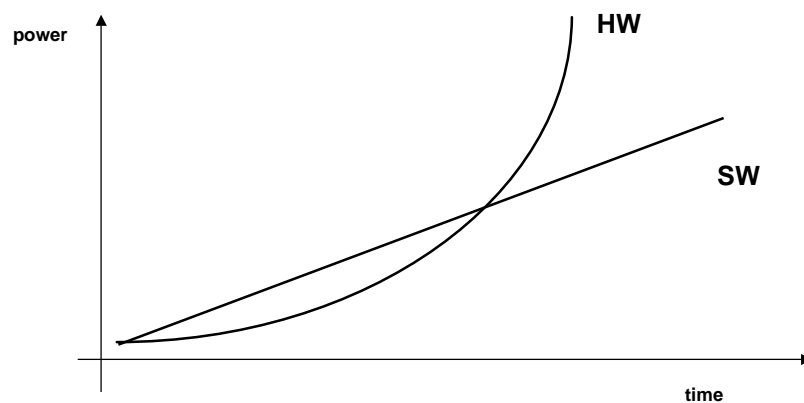
## Outline

- From the code-driven style of work to the specification driven one
- Software technology looks for a methodology, how to develop a software system from the description
- With given presumed properties, in the given time and budget, and with required quality assertions

## The present state of art in software technology

- We never have enough software
- Software rarely comes to market in time
- But the software crisis exists

## Software Crisis



## The present state of art in software technology

- We newer have enough software
- Software rarely comes to market in time
- But the software crisis exists
- We need new powerful software technologies

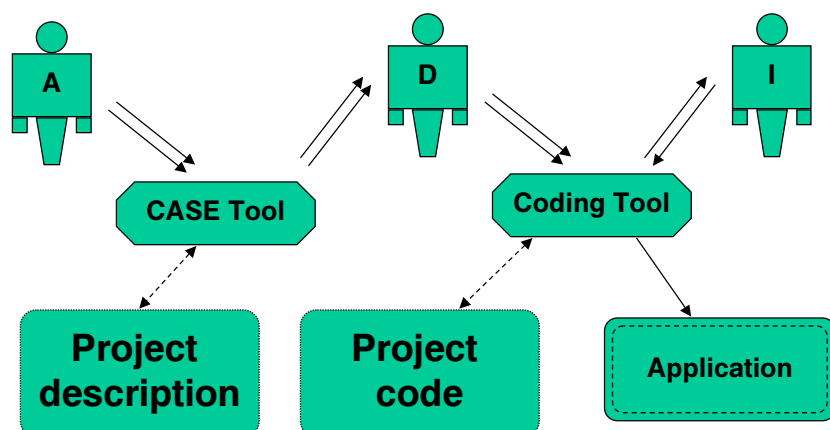
## Splendor and Misery of CASE

- What present CASE knows how to do it:
  - Check license agreement
  - Graphical editors with balancing – documentation
  - Basic data model development from the description
- What present CASE does not know how to do it:
  - Integrity constraints
  - Functional description
  - Dynamic description

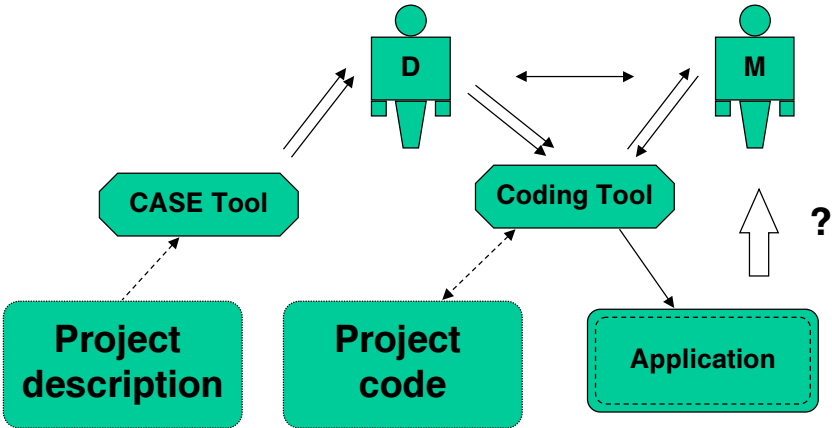
## Formal Specifications in Software Technology

- The idea that software technology requires new techniques and tools that enable specification-driven style of software development instead of present code-driven style. Such tools have to be based on formal specification, but any automated or semiautomated processing of software specification requires a formal background

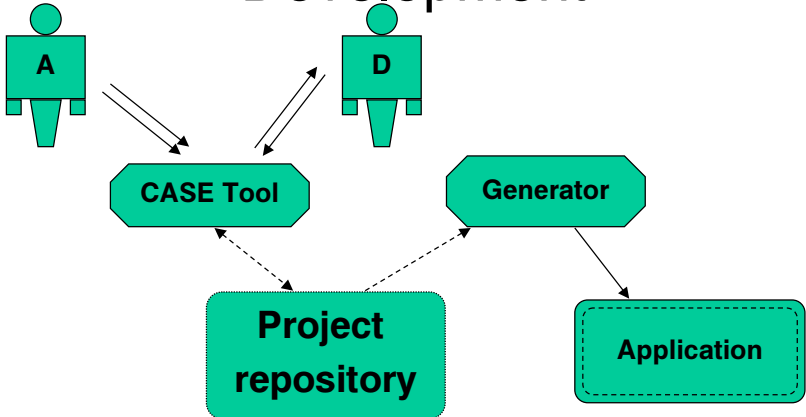
### Code-Driven Development



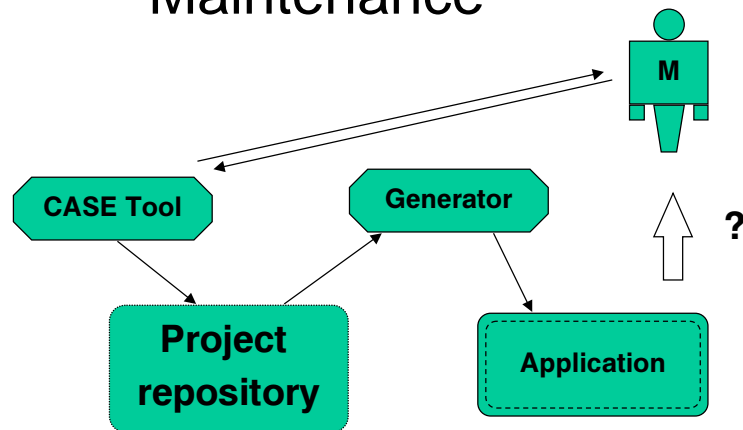
# Code-Driven Maintenance



# Specification-Driven Development



## Specification-Driven Maintenance



## Example: Data Modeling

- Conceptual Data Model
- Add design decisions
- Logical data model
- Add implementation decisions
- Physical data model
- Integrity constraints?

## What is needed ?

- Similar tools for:
  - Integrity constraints
  - Functional model
  - Dynamic model
- Formal specifications can help

## Why formal specifications ?

- The definition in the natural language is inherently ambiguous

# Specification in the Natural Language

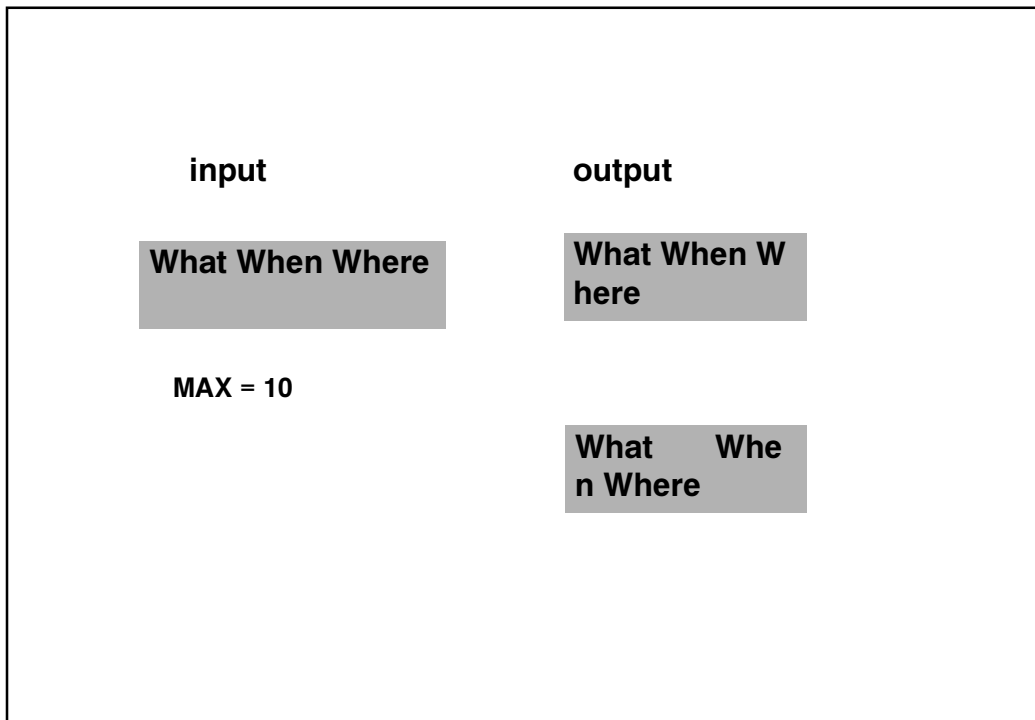
## Peter Naur: The Line Formatting Problem

The input of a program is a text file containing SPs and NLs. The output have to be a text file satisfying following rules:

- The output line can begin only where SP or NL is in input
- Each output line is maximally filled
- No output line contains more than MAX characters

## Question: Is it a correct specification?

- Output and input texts may be different
- What if a long word occurs in input ?
- What to do with separators (copy them or replace them by the only one)
- The output is ambiguous
- And so on, ...
- A: No, it isn't!
- Does exit better specification?



## Another specification in the natural language

### Goodenough, Gerhart: The Line Formatting Problem

The input of a program is a text containing words and gaps. The output have to be a text containing same words as the input in the same order and satisfying following rules:

- The output line can begin only where there is a gap in the input
- Each gap in input is replaced by SP or NL in output
- Each output line is maximally filled, and
- No output line contains more than MAX characters
- ... (the whole definition consists of 25 lines)

## Q: Is this a correct specification ?

- Do we have to separate words in the output ?
- ... 5 line (Naur)  $\Rightarrow$  5 errors
- ... 25 line (Goodenough, Gerhart)  $\Rightarrow$  25 errors
- A: No, it isn't!
- The problem lies in the natural language

## Mathematical Language?

- BreakChar = { SP, NL }
- ValidChar = { a, b, ... }
- Char = ValidChar  $\cup$  BreakChar
- Seq = Char\*
- Text = Seq
- Solution: *form*: Text  $\rightarrow$  Text
- Precondition: input text t has to be correct
- Postcondition: *form*(t) is correct and formatted

## Problem specification

PRE = { t: Text. t "does not contain word longer than MAX" }

POST =

- *form*(t) "contains separators instead of gaps" and
- *form*(t) "does not contain line longer than MAX" and
- *form*(t) "is maximally filled"

## Problem Specification – 2

### Abbreviations:

- *Is-Correct*(t) – „t does not contain too long word“
- *Is-Compressed*(t,u) – „u is a compressed version of t – the only difference is that u contains separators where t contains gaps“
- *Is-Trimmed*(u,n) – „u does not contain line longer than n characters“
- *Is-Filled*(u,n) – „u is maximally filled in lines of length n“
- *Is-Formatted*(t,u) – „u is a formatted version of t“

## Problem Specification – 3

- PRE = *is-Correct*(t)
- POST = *is-Compressed*(*form*(t),t) and *is-Trimmed*(*form*(t),MAX) and *is-Filled*(*form*(t),MAX)
- $is-Formatted(t, form(t)) \Leftrightarrow is-Correct(t) \Rightarrow$   
 $(is-Compressed(form(t),t) \wedge is-Trimmed(form(t),MAX) \wedge is-Filled(form(t),MAX))$

## Problem Specification – 4

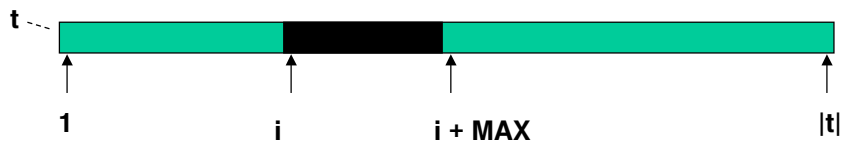
- The implementation of the line formatting problem is any function  $form: Text \rightarrow Text$  such that:

$\forall t \in Text . is-Formatted(t, form(t))$



## How to describe e.g. is- Correct ?

- *Is-Correct* ... means t does not contain a word longer than MAX



$\forall i \in \{1, \dots, |t| - \text{MAX}\} .$

$t[i] \in \text{ValidChar} \Rightarrow \exists j \in \{i, \dots, i + \text{MAX}\} . t[j] \in \text{BreakChar}$

## Splendor and Misery of Formal Specification

- Splendor: it has to be precise
- Misery: it has to be precise

## Q: Is the specification correct?

Specification of the sorting problem:

- $\text{Seq} = \text{Int}^*$
- Solution:  $\text{sort}: \text{Seq} \rightarrow \text{Seq}$
- Precondition: none (any  $t$  of the type  $\text{Seq}$  can be sorted)
- Postcondition:  $\text{sort}(t)$  is sorted



## What does it mean sorted ?

Specification of the property sorted:

- $\text{is-Sorted}: \text{Seq} \rightarrow \text{Bool}$
- Precondition: none (any  $t$  of the type  $\text{Seq}$  can have the property  $\text{is-Sorted}$ )
- Postcondition:  $\text{is-Sorted}(t) = \text{true}$  if  $t$  is sorted, otherwise  $\text{is-Sorted}(t) = \text{false}$
- $\forall t \in \text{Seq} . \text{is-Sorted}(t) \Leftrightarrow (|t| \leq 1) \vee (t = xyu \wedge x \leq y \wedge \text{is-Sorted}(u))$

## Sorting Problem Specification

- The implementation of the sorting problem is any function  $sort: Seq \rightarrow Seq$  such that:
- $\forall t \in Seq . is\text{-Sorted}(sort(t)) = true$



Q: Is this a correct specification ?

Let us try to define the function foo:

- $foo: Seq \rightarrow Seq$
- $\forall t \in Seq . foo(t) = \langle 1,2,3 \rangle$
- We can simply prove that:
- $\forall t \in Seq . is\text{-Sorted}(foo(t)) = true$

but foo is not what we expected?

**A: No, it is not a correct specification of the sorting problem!**

## A specification is always correct !

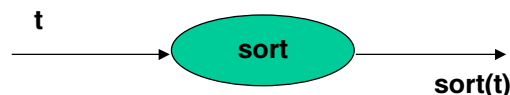
- Any syntactically correct specification is a correct specification of something
- The specification by the predicate *is-sorted* specifies only subset of sorted sequences
- The specification of the sorting problem requires more sophisticated definition
- This problem cannot be solved
- Only prototyping can help

## Sorting Problem Specification - 2

- The implementation of the sorting problem is any function

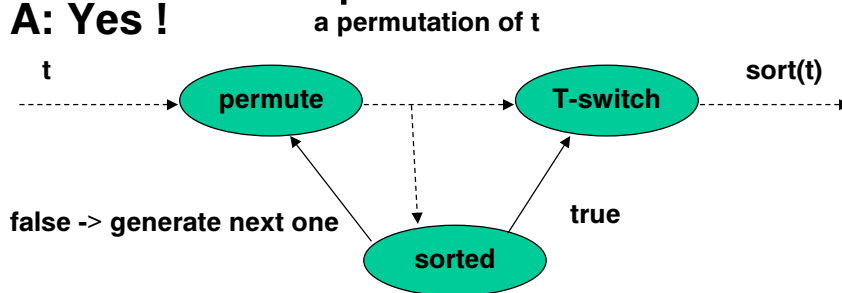
*sort*: Seq  $\rightarrow$  Seq such that:

$\forall t \in \text{Seq} . \text{is-Sorted}(\text{sort}(t)) = \text{true} \wedge$   
 $\text{is-Permutation}(t, \text{sort}(t))$



Q: Can an implementation be derived from the specification ?

A: Yes !



not very efficient - complexity  $\Theta(|t|!)$

## Testing, validation and verificatio

n

- **testing**

$t \in \text{Seq} . \text{sorted}(\text{sort}(t)) = \text{true} \wedge$   
 $\text{is-Permutation}(t, \text{sorted}(t))$

- **validation (also prototyping)**

$\forall t \in X \subset \text{Seq} . \text{sorted}(\text{sort}(t)) = \text{true} \wedge$   
 $\text{is-Permutation}(t, \text{sorted}(t))$

- **verification**

$\forall t \in \text{Seq} . \text{sorted}(\text{sort}(t)) = \text{true} \wedge$   
 $\text{is-Permutation}(t, \text{sorted}(t))$

## So what ?

We need a different form of a specification - executable specification that:

- the specification can be prototyped
- the specification can be converted into an implementation
- the implementation can be verified according to the specification

## Executable specification - OBJ3

- Stanford
- algebraic specification
- implemented as a rewriting system
- symbolic evaluation - prototyping
- modular structure
- generic modules

## Sorted predicate in OBJ3

```
obj SORTED [ X :: POSET ] is
  protecting LIST[X] .
  op sorted : List -> Bool .
  vars L : List .
  vars E E' : Elt .
  cq sorted(L) = true if (|L| <= 1) .

  cq sorted(E' L) = true if E <= E' and sorted(E' L) .

endo
```

## Specification of sorting in OBJ3

```
obj SORTING [ X :: POSET ] is
  protecting LIST [X] .
  op sorting_ : List -> List .
  op unsorted_ : List -> Bool .
  vars L L' L'' : List .
  vars E E' : Elt .
  cq sorting L = L if unsorted L /= true .
  cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
  cq unsorted L = false if (|L| <= 1) .
  cq unsorted L E L' E' L'' = true if E' < E .

endo
```

## Quick-sort in OBJ3

```
obj QUICKSORT [ X :: POSET ] is
  protecting LIST [X] .
  op piv-le : Elt List -> List .      *** members less or equal
  op piv-gt : Elt List -> List .      *** members greater than
  op quicksort_ : List -> List .
  vars E E' : Elt .
  var L : List .
  eq quicksort nil = nil .

  eq quicksort E L = (quicksort piv-le(E, L)) E (quicksort piv-gt(E, L)) .

endo
```

## Q: Is quicksort correct implementation of sorting ?

- We have to prove:

$$\forall l \in \text{List} . \text{sorting}(l) = \text{quicksort}(l)$$

- OBJ3 is not intended for proofs but prototyping
- We have to use some proof assistant (e.g. LEGO)

# Why OBJ3 and LEGO?

