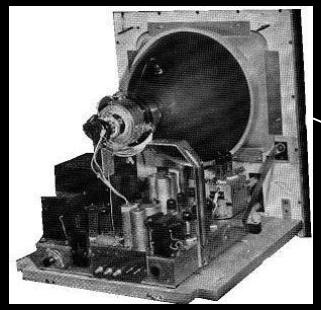
ID Masters Module

Modeling and Specification in Action

Introduction to Software Engineering

prof .loe feijs

How computer programs are made



1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Machine language

```
$TEST:    PUSHF
          SETST     4
          SETST     3
          SKBIT     15
          CLRST     4
          PUSH      0
          LD        0,MANT
          SKBIT     15
          CLRST     3
          LI        0,0
          SKSTF     4
          SETBIT    0
          SKSTF     3
          SETBIT    1
          ST        0,$STAT
          PULL      0
          PULLF
          RTS
NORM:     SKNE      0,$NUL
          JMP       $H
          JMP       $I
$H:       SKNE      1,$NUL
          JMP       $J
          JMP       $I
$J:       LI        2,0
          RTS
$I:       PUSHF     ETC
```

Concepts:

- memory
- arithmetic
- logic
- stack
- jumps

average productivity:2.5 lines per hour

-- Anonymous.

Like the old joke, "He has experience, he wrote over 350 kloc personally... Then he discovered loops."

High-level language

```
      SUBROUTINE TOASC (N,M,NADE)
C ***************************
C TOASC CONVERTS INTEGER N  *
C OF MAX M DIGITS TO ASCII  *
C RESULT IN ARRAY NADE      *
C***************************
      DIMENSION NADE(M)
      L=N
      I=0
      DO 10 J=1,M
      K=M-J
      I=N/(!)**K)
      NADE(J)=I+48
      N=N-(I*(10**K))
10    CONTINUE
      N=L
      RETURN
      END
```

Concepts:

- types
- variables
- If-then-else
- for, while, repeat, etc.
- procedures, parameters

average productivity: 2.5 lines per hour

Structured programs

```
procedure straightselection;
      var i,j,k: index;
          x : item;
begin for i := 1 to n-1 do
          begin k := i;
                x := a[i];
                for j := i+1 to n do
                    if a[j].key < x.key then
                       begin  k := j;
                              x := a[j]
                       end;
                a[k] := a[i];
                a[i] := x;
     end
end
```

Prof. Edsger Dijkstra,
1930-2002

http://www.acm.org/classics/oct95/

Concepts:

- indentation
- data records
- nested scopes
- elimation of goto
- recursive procedures
- axiomatic theory

average productivity: 2.5 lines per hour

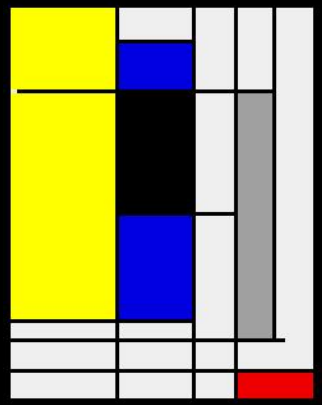
Object-orientation

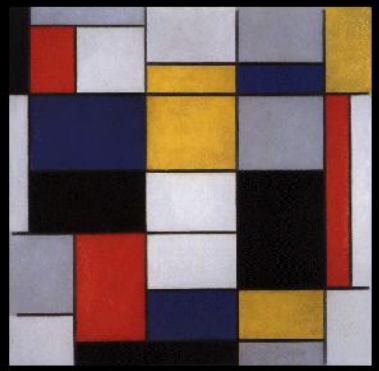
Concepts:

- encapsulation
- inheritance
- polymorphism

average productivity: 2.5 lines per hour

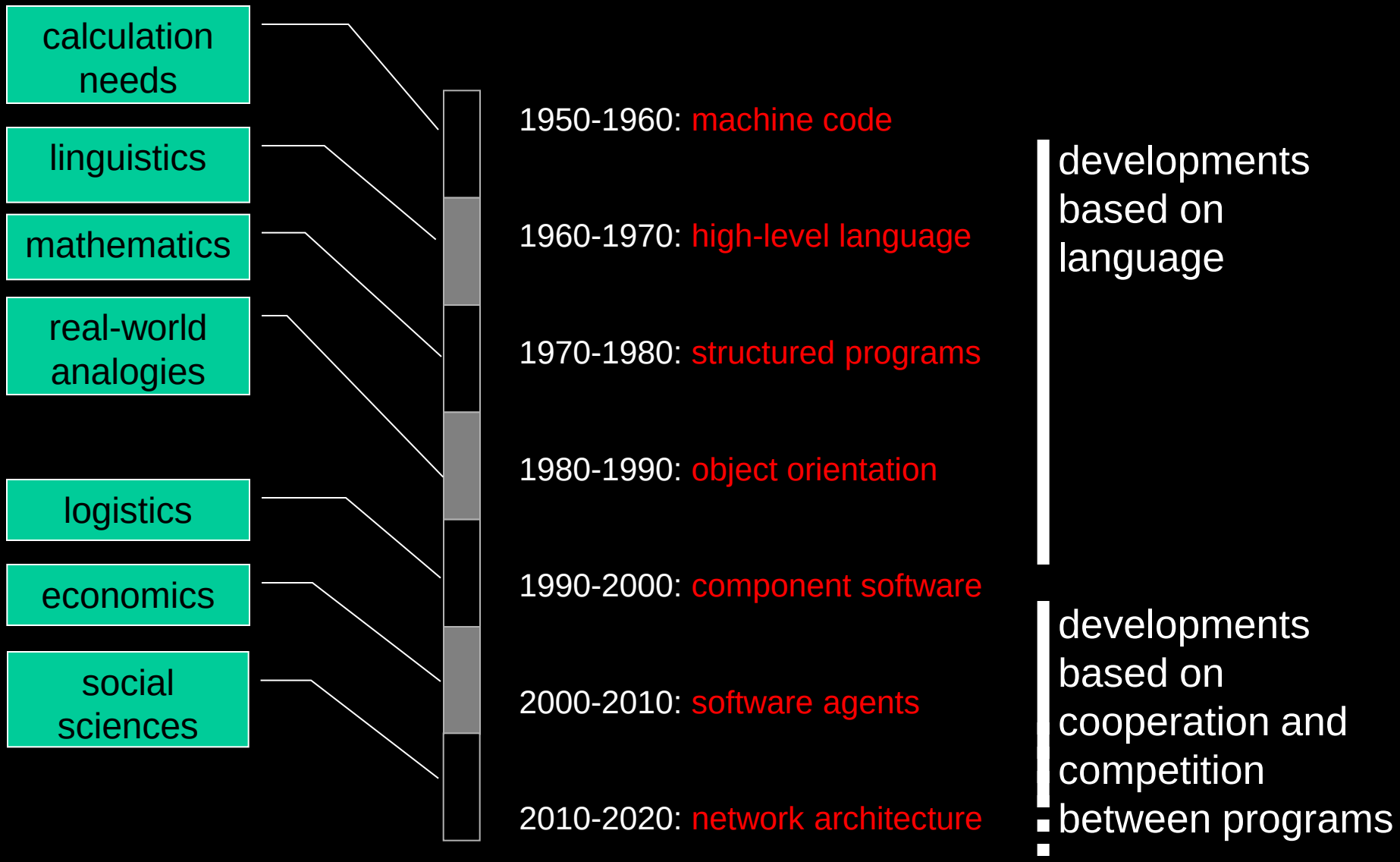
```
procedure Painting.pntA;
var
  i : integer;
begin
  self.color := LightGrey;
  self.MaxCell := 1 + random(25);
  for i := 0 to self.MaxCell do begin
      self.Cells[i] := mkCell(self.mkKernelA([]));
  end; {for}
  Delay(100);
end;

procedure Painting.pntB;
var
  i : integer;
begin
  self.color := LightGrey;
  self.MaxCell := 1 + random(50);
  for i := 0 to self.MaxCell do begin
      self.Cells[i] := mkCell(self.mkKernelB([]));
  end; {for}
  Delay(100);
end;
```





Feijs, Matematica e cultura, Venezia, 2006

Sources and drivers of the innovations



Metaphor: a program is like a …
machine set-up

1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

1980-1990: object orientation

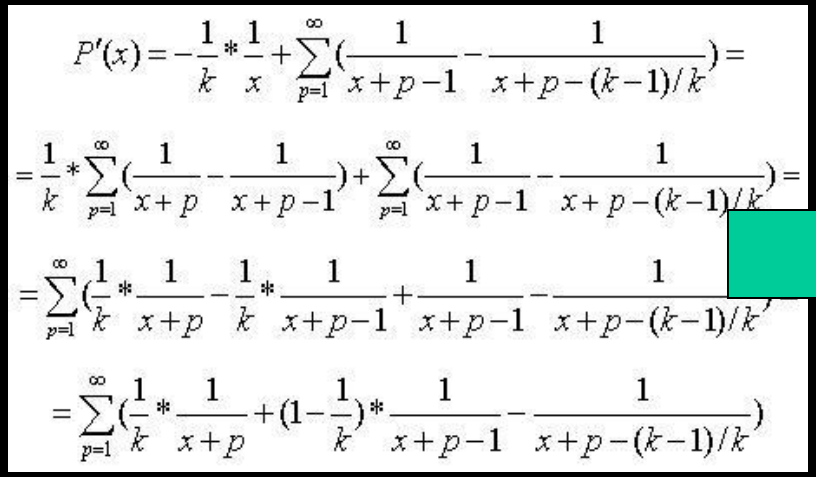
1990-2000: component software

2000-2010: software agent

2010-2020: network architecture

Metaphor: a program is like a …

formula



1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

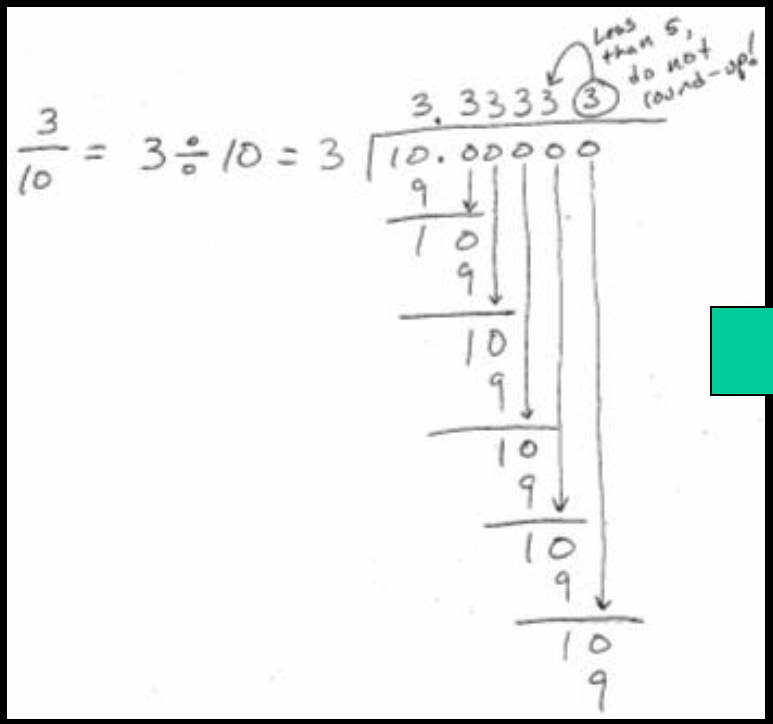
1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Metaphor: a program is like a …
step-by-step procedure (algorithm)



1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

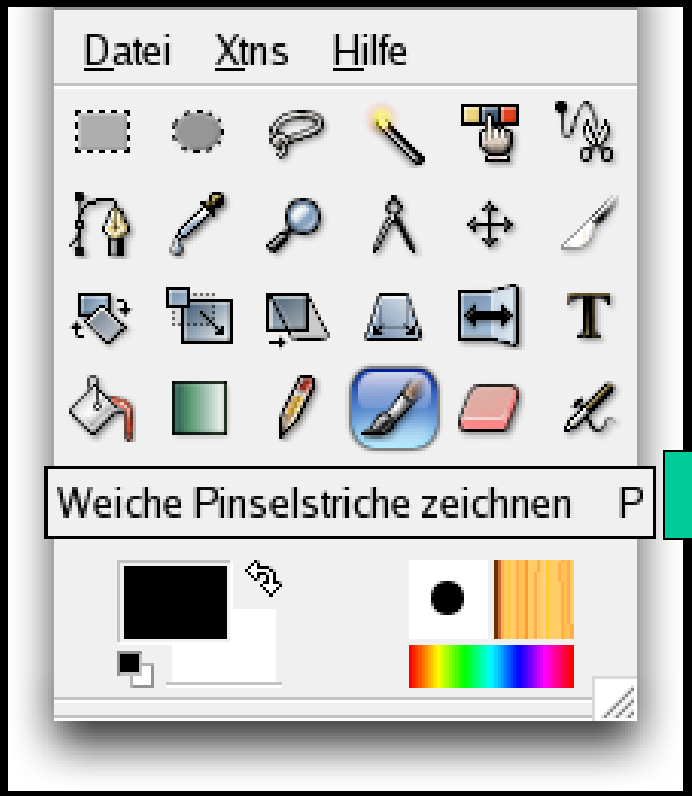
1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Metaphor: a program is like a …
set of real world objects



1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Metaphor: a program is like a …
set of hardware components

1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Metaphor: a program is like an … agent

1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

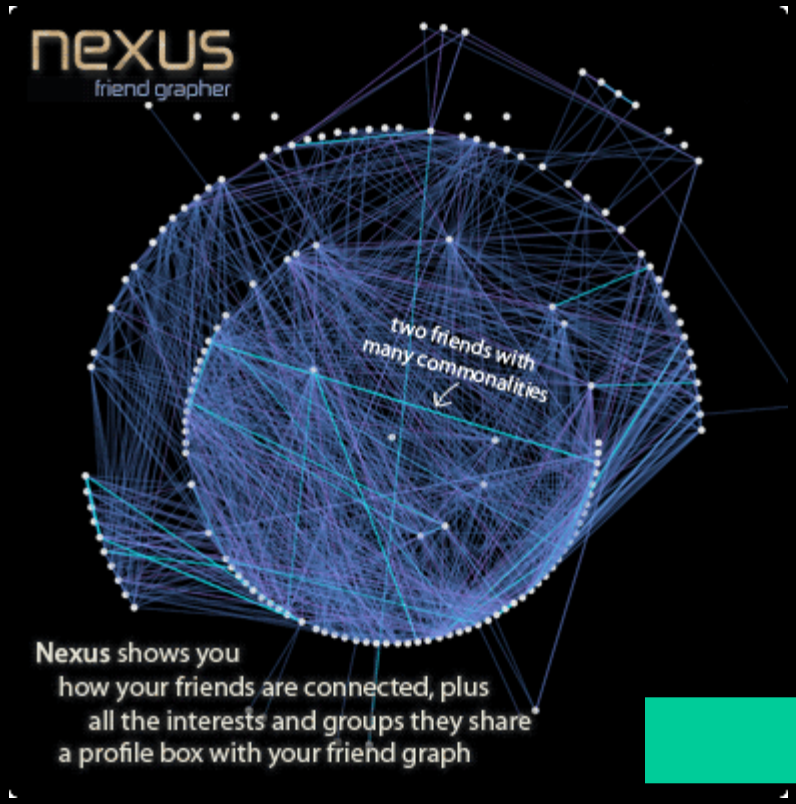
1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Metaphor: a program is like a …
member of a community



1950-1960: machine code

1960-1970: high-level language

1970-1980: structured programs

1980-1990: object orientation

1990-2000: component software

2000-2010: software agents

2010-2020: network architecture

Component software

Concepts:

- registration
- interface specification
- downward compatibility
- language independence

Software agents

Concepts:

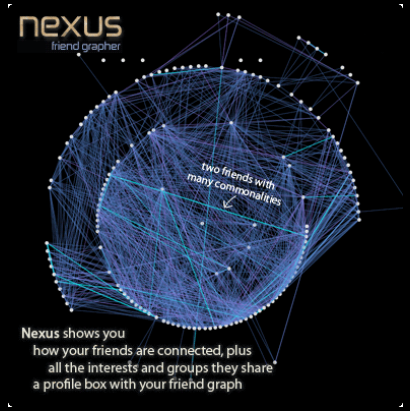
- security
- authentication
- emergent behaviour
- economic and game theory

Multimedia purchasing apparatus Espacenet
http://worldwide.espacenet.com/publicationDetails/biblio?DB=EPODOC&II=9&ND=3&adjacent=true&locale=en_EP&FT=D&date=20060310&CC=KR&NR=20060022673A&KC=A

Network architecture

Concepts:

- protocols
- uses relations
- part-of relations
- large scale on-line communities



Software complexity

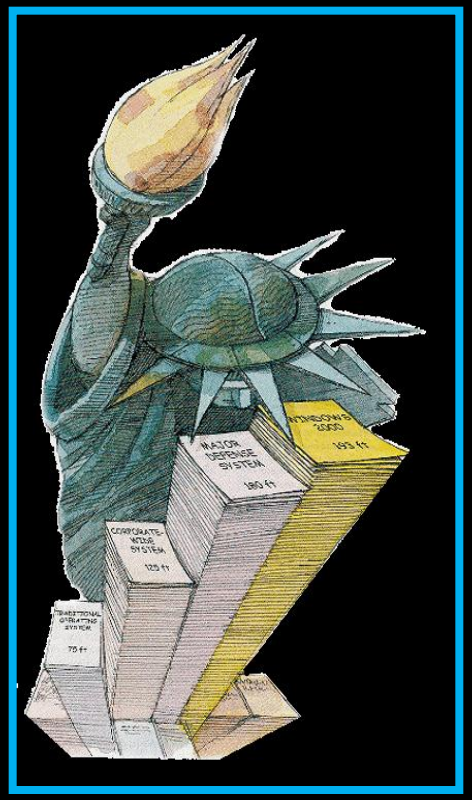
Drawing application : ± 4K lines

High-end television : ± l.5 M lines

Telephone exchange: ± 6M lines

PC operating System: ± 30M lines

Methods and tools needed



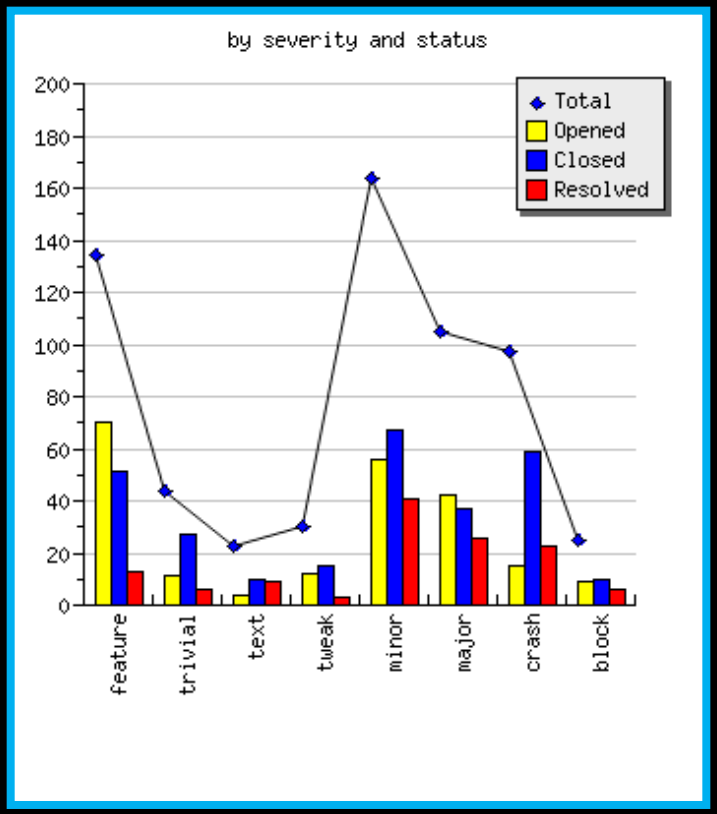
Software complexity symptom fighting approaches

bug management tools

expensive gurus

planning tools

outsourcing



Model based approaches

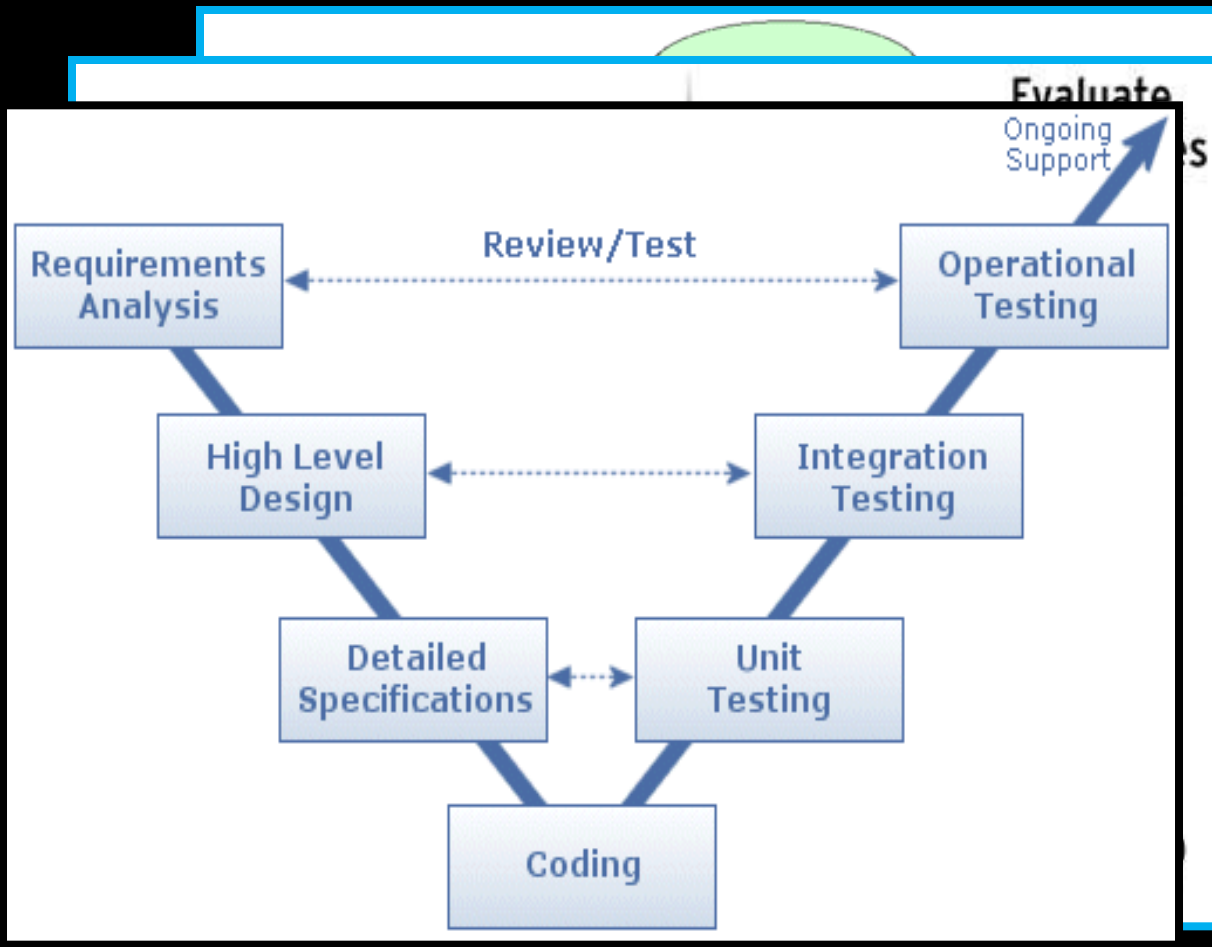
- testing
- life-cycle models
- software specification
- software architecture verification

Testing

- Automatic test generation testing generation feijs Google Scholar
- Coverage analysis
- Regression tests

Life cycle models

- waterfall mode
- spiral model
- V model



Software specification:

Flow charts, Nassi-Sneidermann diagrams, SDL, Yourdon diagrams, Message sequence charts, Entity relationship diagrams, Class diagrams, ITU, OSI ...

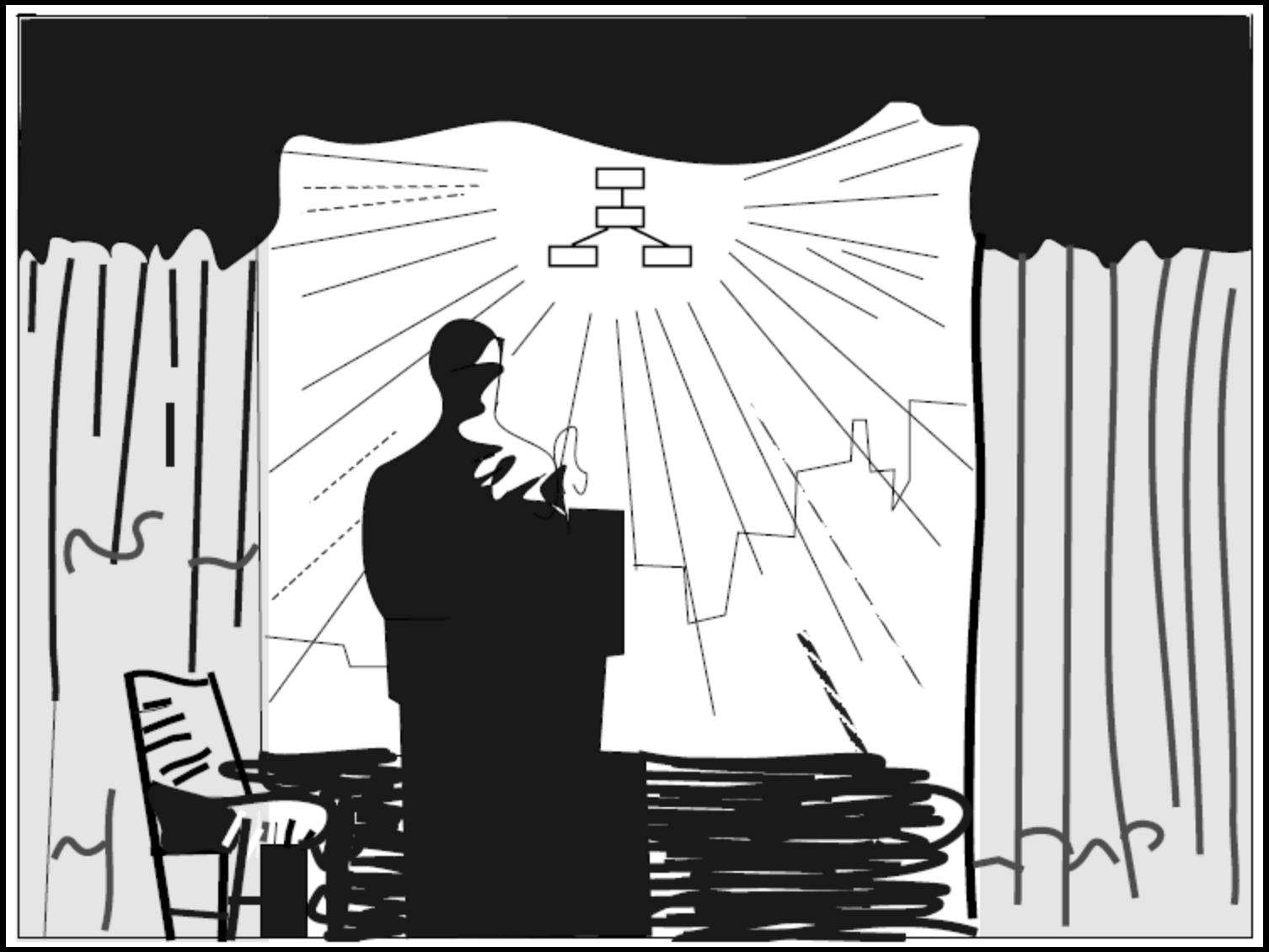
UML: the Unified Modeling Language

- describing user behaviour
- describing software behaviour

Software architecture verification:

- Relation partition algebra
- Initial Example
- Applications

The software architecture is presented.



Source: “Architecture Visualisation and Analysis, Motivation and Example” by L. Feijs and R. Van Ommering, in slightly modified form presented at the ARES International Workshop on Development and Evolution of Software Architectures for Product Families 1996

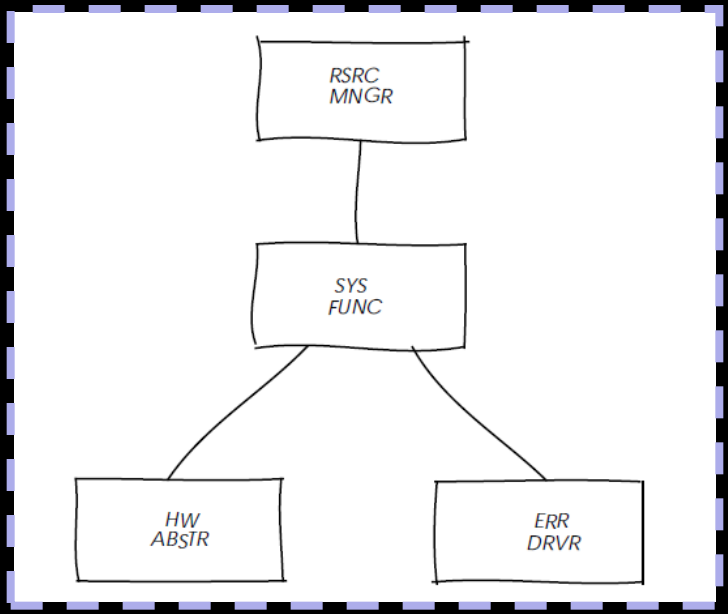
"Dear friends, Figure 2 is our software architecture: there are four software components, which I will explain now.

•RSRC MNGR is the Resource Manager, which will contain the main procedures of all our processes and these will be scheduled by the HW and SW of our platform.

•SYS FUNC contains the System Functions, and this is the heart of our system. This will provide the data transformations our customers are waiting for.

•HW ABSTR is the minimal Abstraction of the special Hardware of our platform.

•ERR DRVR is the Error Driver which provides for error printing and contains a driver for the special error LED. "



```
/* Component: ERR_DRVR */
err_pr() { led_33(); }
led_33() { err_pr(); }
```

```
/* Component: HW_ABSTR */
#include "ERR_DRVR.h"
power() { err_pr(); i2c(); }
i2c() { }
```

```
/* Component: SYS_FUNC */
#include "HW_ABSTR.h"
#include "ERR_DRVR.h"
a() { b(); c(); }
b() { power(); }
c() { d(); g(); }
d() { i2c(); }
e() { f(); }
f() { g(); err_pr(); led_33(); }
g() { h(); }
h() { err_pr(); }
```

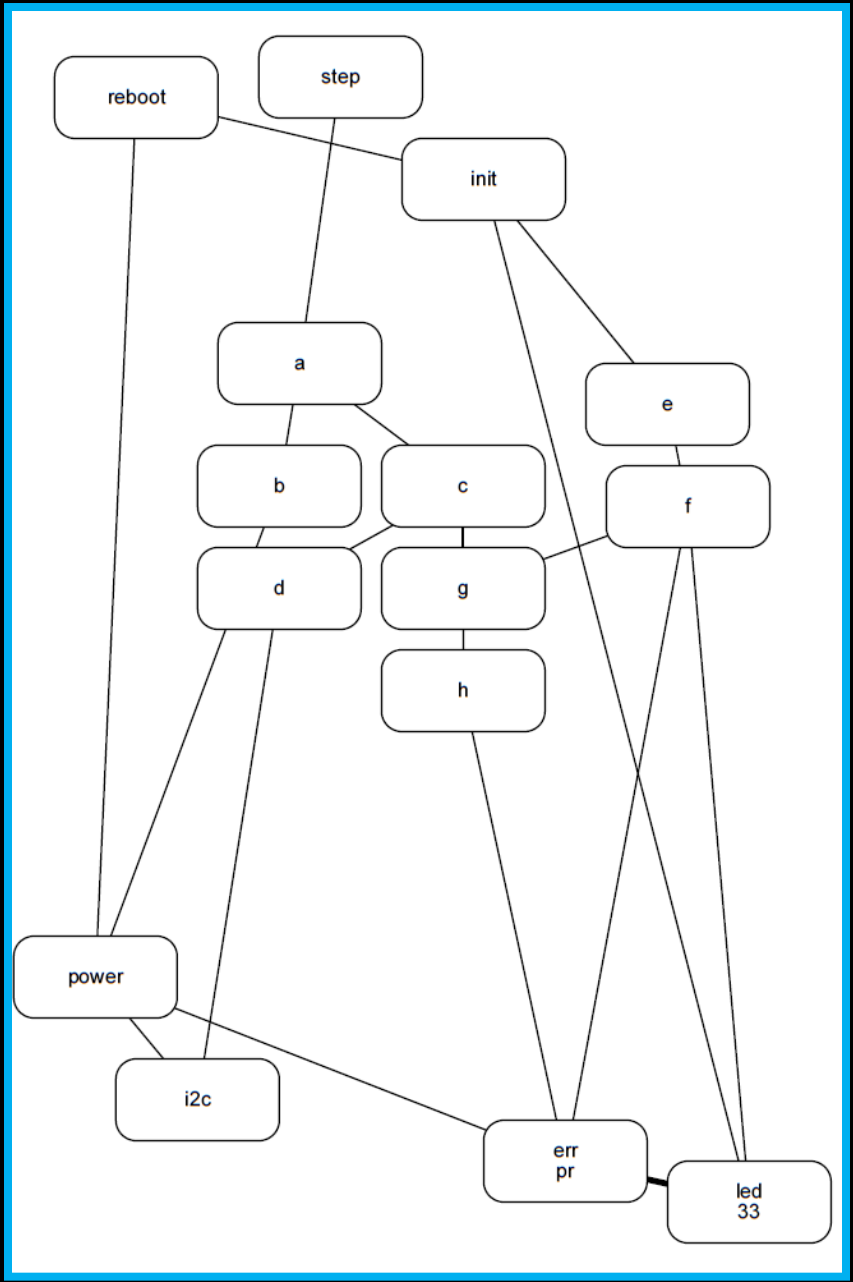
```
/* Component: RSRC_MNGR */
#include "SYS_FUNC.h"
#include "HW_ABSTR.h"
#include "ERR_DRVR.h"
init() { e(); led_33(); }
reboot() { power(); init(); power(); }
step() { while (1+1==2) a(); }
```

```
err_pr  led_33
led_33  err_pr
power   err_pr
power   i2c
init    e
init    led_33
reboot  power
reboot  init
reboot  power
step    a
a   b
a   c
b   power
c   d
c   g
d   i2c
e   f
f   g
f   err_pr
f   led_33
g   h
h   err_pr
```

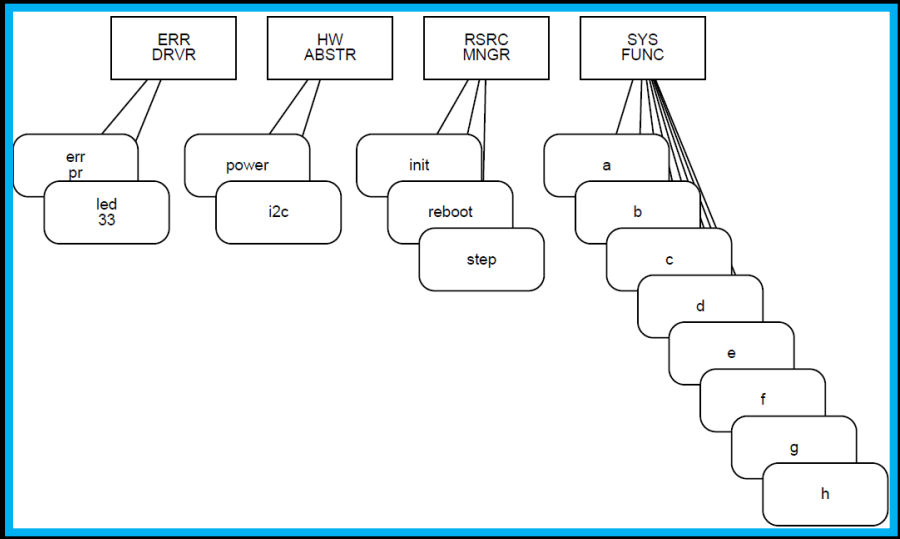
the essential ‘use’ information is easily extracted and stored in a file called uses.

The essential information of Figure 2 is an intended ‘use’ relations on components, which is as follows:

```
RSRC_MNGR  SYS_FUNC
SYS_FUNC   HW_ABSTR
SYS_FUNC   ERR_DRVR
```



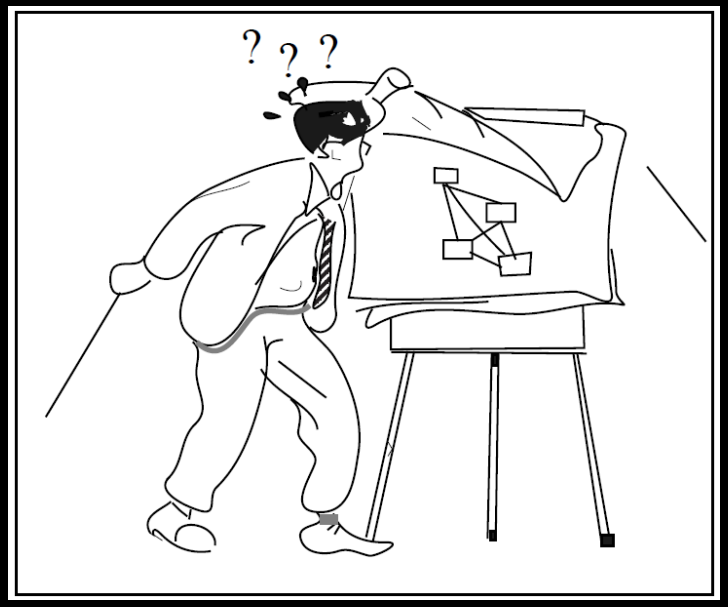
Uses relation at the C function level

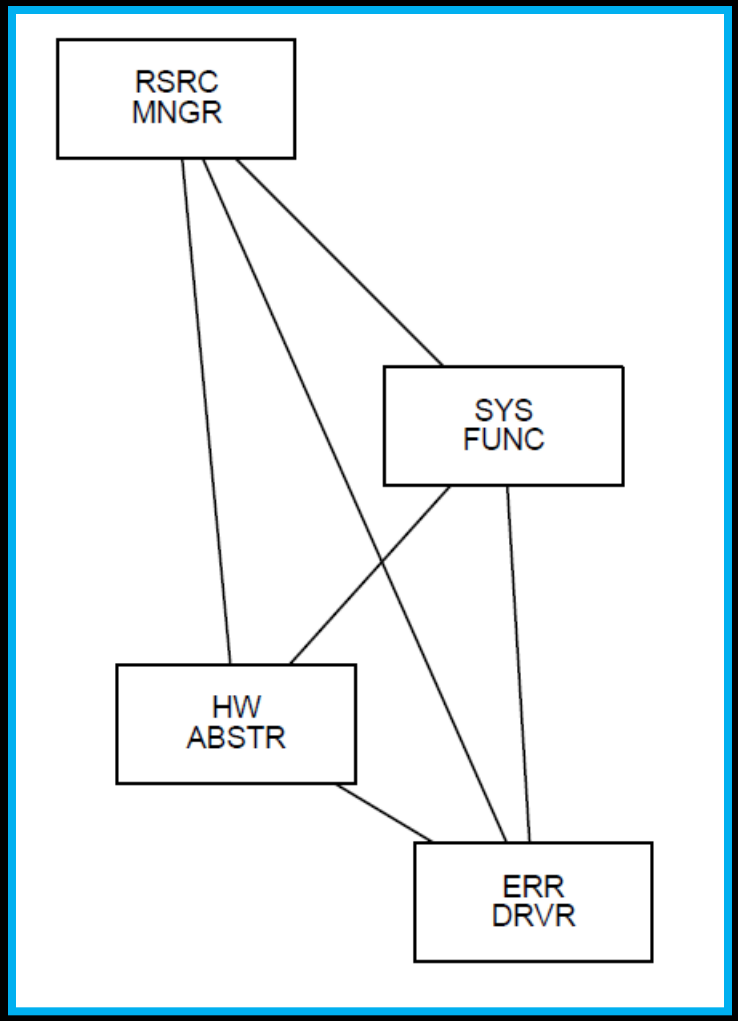


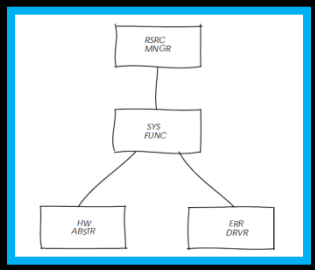
Part-of relation between functions and components

Calculate **use** lifted by **part-of**

```
HW_ABSTR   ERR_DRVR
RSRC_MNGR  SYS_FUNC
RSRC_MNGR  HW_ABSTR
RSRC_MNGR  ERR_DRVR
SYS_FUNC   HW_ABSTR
SYS_FUNC   ERR_DRVR
```

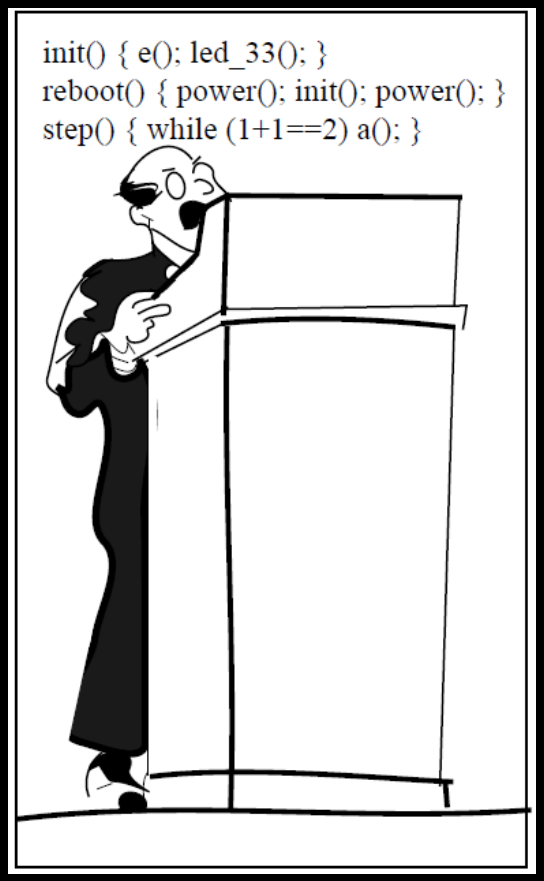




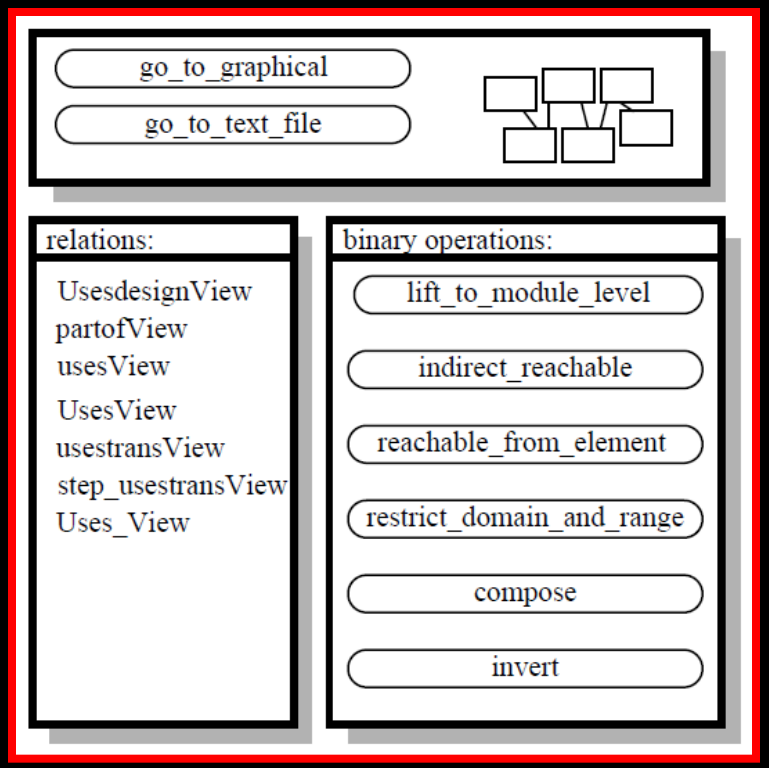


The architect discovers the real system

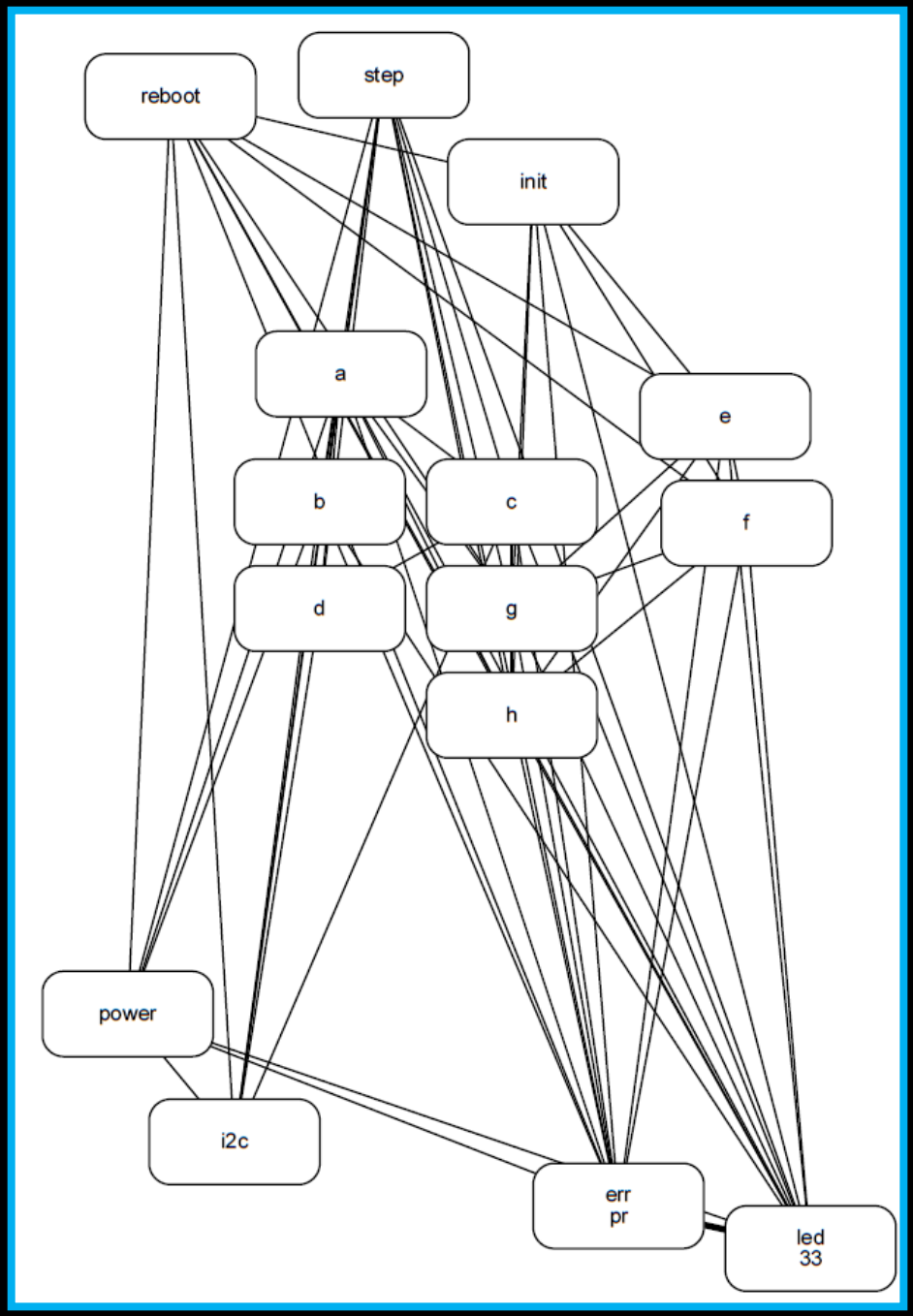
The resource manager component RSRC MNGR has three C functions, init, reboot, step, each of which can be viewed as an independent main program. Of these, init and reboot are tied to the hardware reset interrupt and the software interrupt (trap), whereas step is supposed to be called in an eternal loop. The architected component-level 'use' relation of Figure 2 has been made with the step function in mind. But everybody knows that for initialization and rebooting one has to do some low level tricks every now and then. For example reboot has to call power and indeed, this causes a direct 'uses' line from RSRC MNGR to HW ABSTR. This explains why Figure 5 has more lines than Figure 2. And if you look at it this way, we have in fact respected the original architecture.



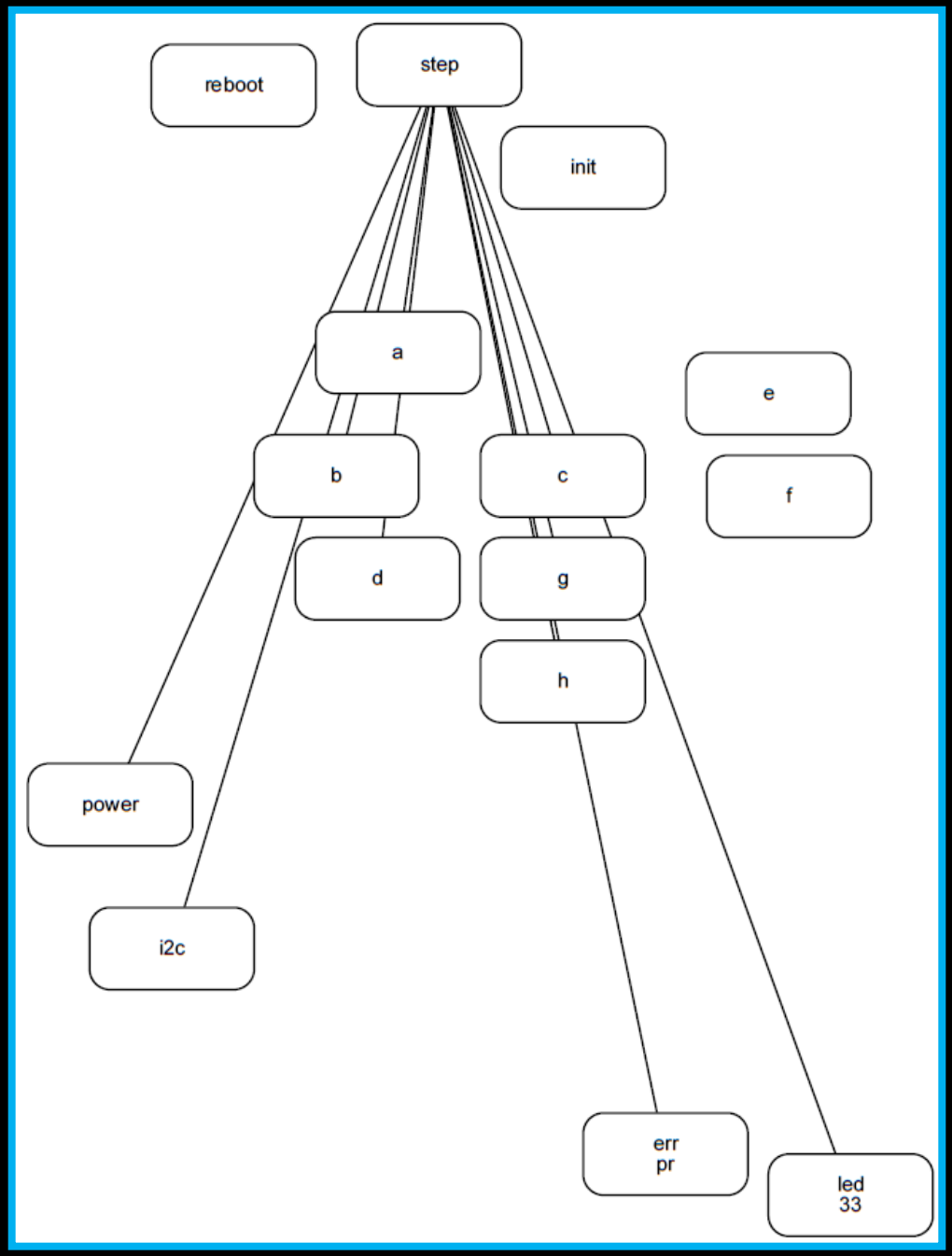
The programmer explains why RSRC MNGR must use ERR DRVR.



The relational calculator



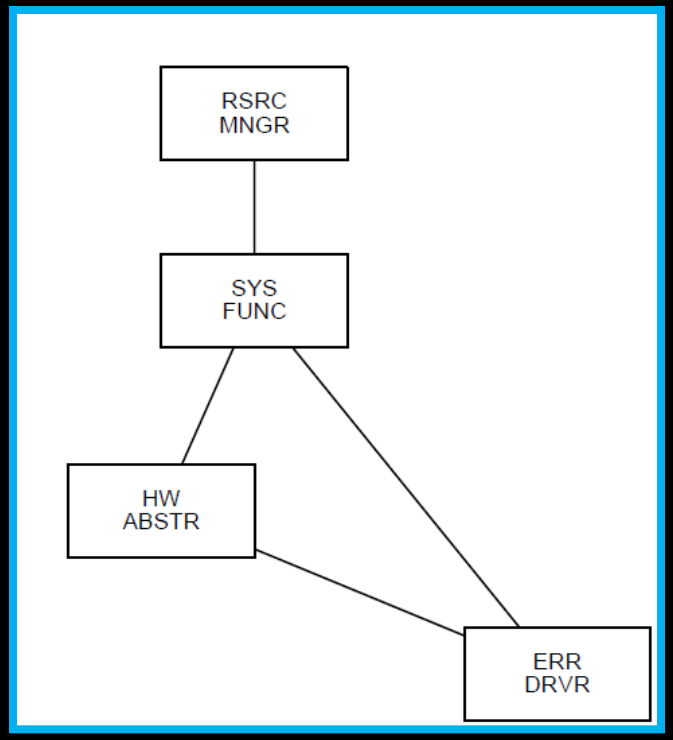
Transitive closure of **uses**



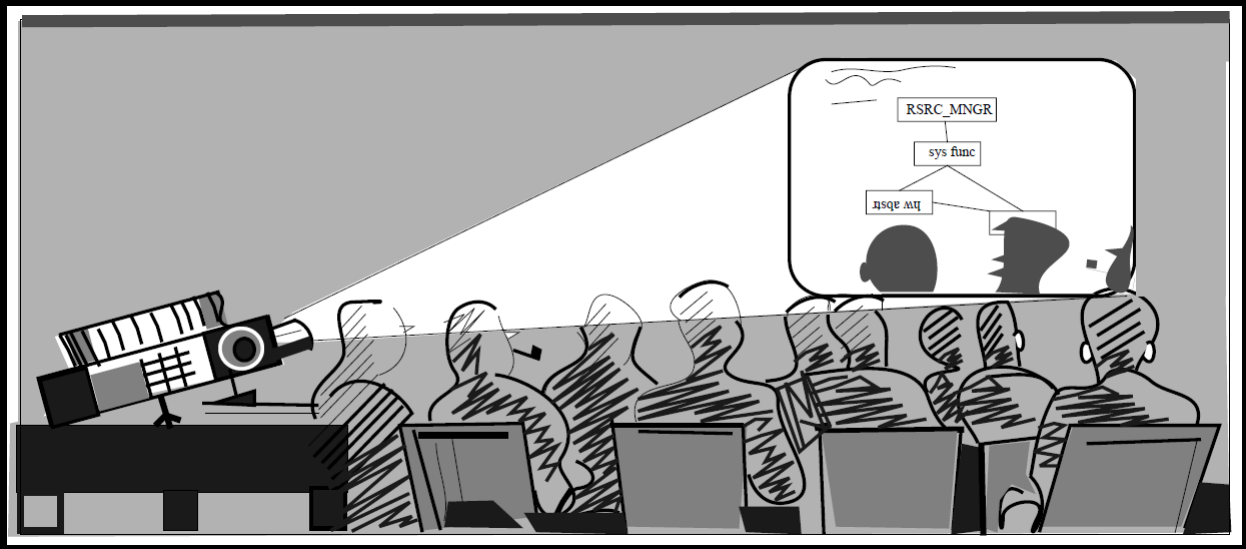
a
b
c
d
err_pr
g
h
i2c
led_33
power
step

Set of elements reachable from **`step`**

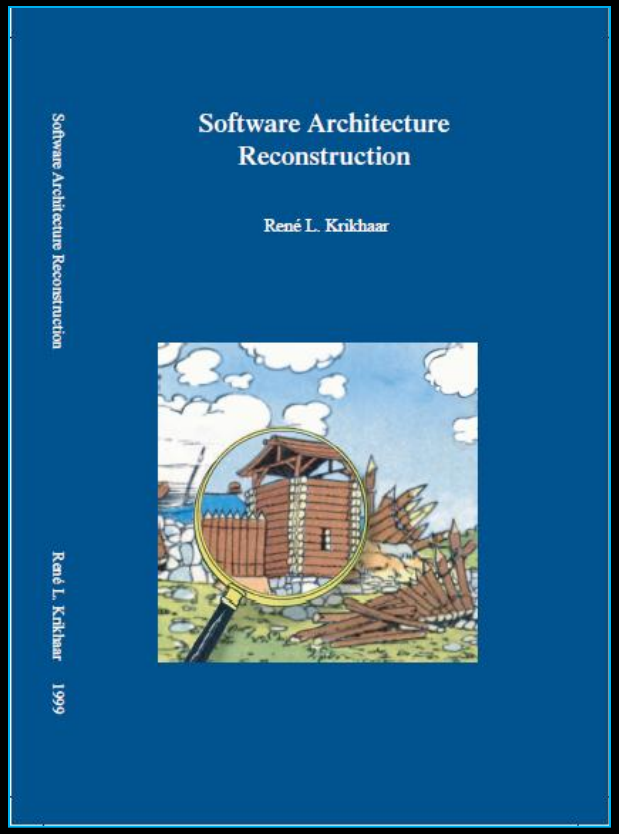
Functions transitively **uses** connected to step function.



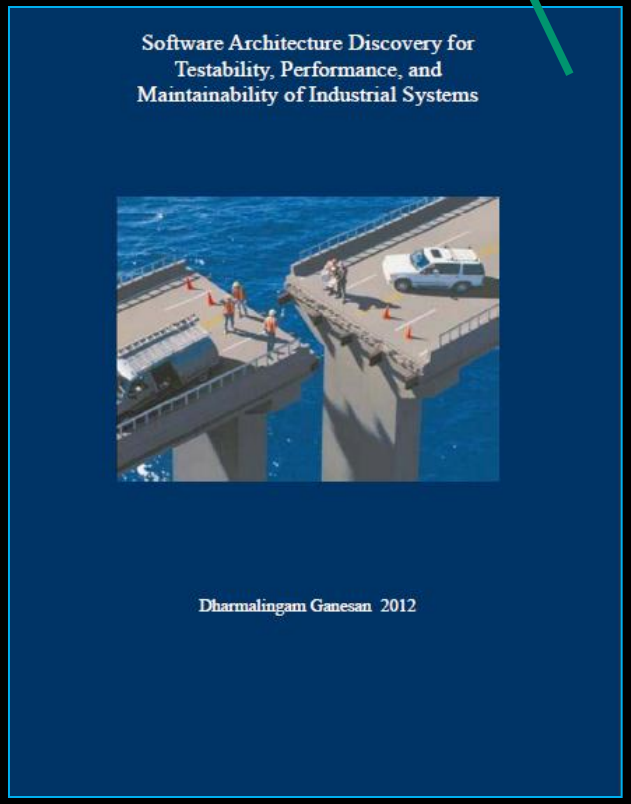
recalculated lifted **use** relation after restricting use to set reachable from `step`



The team arrives at a common understanding of the software architecture



Chapter 2: *Verifying Architectural Design Rules of a SPL. In this chapter, we cover the* research questions RQ2.1 and RQ4.1. The high-level research questions are a) how can we analyze whether or not the specified product line architectural rules are followed in the implementation? and b) how are the implemented decisions related to business goals? These research questions were investigated using the NASA's core flight software product line (CFS) as the case study. This chapter was published at the international conference on software product line (SPLC), in 2009 [97].



Thank you for your attention