Dr.Zygmunt Kyznar

An entity interface in the free structured design (description proposal)

In this paper an interfacing of entities is discussed in terms of the design methodology and a formalized specification. Statements and abstract types related to the interface are proposed. Some patterns of the usage are shown.

Introductory remarks

"An interface" is a very popular word used in the variety of meaning. In this paper we do not intend to discuss a hardware interface in terms of physical connections between devices. Moreover, we will not pay too much attention to a data linking which is an internal matter of the entity named data base. So, we interprete an interfacing as external connectors between entities or an external action to transform the body of a given entity according to the requirements of another entity.

We assume that the interface is necessary for any entity having a need to communicate with another one. The reason for interfacing is obvious in the case of an interaction between different types of entity, e.g.procedure and data base (when communication blocks or subschemas are used), but may be well-founded for objects of the same type, e.g.passing parameters between procedures.

To facilitate interface operations , as a rule common for many entities, the interface should be set up as a visible part of an invoked entity or established as a separate entity even.

To be used correctly by many users or to be a subject of computer aided design, an interface should be precisely described by means of the specification language. Because an interface is not simply collection of parameters and procedures, and is closer to a generator than a program, a description of it differs essentially

from the scope of DDL and programming languages. Also specification facilities of Ada language are fragmentary in terms of documentation and restricted to the programs and their environment.

This has been a reason for inventing the S&DL language proposal discussed here. That specification language is capable to express logical interfaces as well as physical ones. A logical interface is used at a conceptual schema level for communication between conceptual entities (like USER, DECISION, PROBLEM) and does not operate on physical data contained in data files. A physical interface acts at external level(and deals with a program source text) or at internal level when microprogramming methods are used.

Interfacing in the free-structured design

In a common sense, the structured design is a collection of methods, techniques and tools for designing wholes from components by means of relationships and interfaces. A 'free-structured' term has been introduced as opposite to the 'hierarchically structured' one and indicates a variable structure of systems not restricted to the 'well-structured' hierarchy.

Our critical view on hierarchical approach can be supported by following citation from Langefors B. [1]: "...commonly quoted rule (...) is the use of an hierarchical systems structure. This is often suggested on the ground that natural systems are often (or even always) built upon a hierarchical structure. Again this is not a very satisfactory state of affairs, by not providing an answer to the question why it does not give a sufficient insight. Also although a set of rules or principles is fairly generally accepted, not only does the literature lack good arguments for their support, but also an explicit listing of them is still missing and the rules are seldom operatively precise enough."

The reality shows a very high complexity and changeability of the business. Therefore,a step-by-step hierarchical decomposition is used so widely as a right way of structuring of the knowledge and is misused as an only way for structuring of information systems(how systems with static hierarchic structures can satisfy a changeable reality?).

The real purpose of the free-structured design method (FDM) is creating systems with variable structure and undefined borders using a library of standardized predefined blocks and specific tools named tuners and composers. Results of the FSD activity are application packages, which are only temporary (transient) reflection of the problem actual state. Thus the system is growing as a collection of many incarnations based on the same family of generalized procedures more orientated on computer technology than business areas.

An elementary building block should be constructed in such way, that it can be located in any place of the system and may cooperate with any other block. Therefore, an interfacing is so vital activity in the free-structured design and can be performed efficiently using computer-aided tools like specification language, program generators (tuners particularly) and composers (for assemblying of application packages).

Specification language proposal

A description written here refers to the third version of S&DL language. First version was presented in [2] and second in [3]. This paper contains fragmentary description focused on the interface in program environment and within program itself.

S&DL language consists of two subsets: SL specification language and &DL design language. Specification language is dedicated to describe every object involved in computerized information system. An object called 'entity' represent simple or complex unit which can be described and accessed as relatively independent whole.

An entity can be specified in three syntactic units: definition, body and interface. The first unit is obligatory and contains information describing the behaviour of entity and its attributes. A private part of definition may be written inside the body and then is accessible only for users having rights to use of body. An interface can be an entity itself and this is recommended when it is common for many entities.

A syntax of the SL specification language can be expressed as follows:

```
def LANGUAGE(SL)
   lexical_atoms ::= (<name> , <control_word> , <determiner> , <de-
                      -limiter> , <connective> , <attribute> ,
                      <statement> )
   <name> ::=  NAME  --entity type
              (NAME) --entity name=occurrence
               name  --nonentity object type
              (name) --nonentity object occurrence
                ()   --empty name=name created internally by system
                < >  --empty template
               <≠>   --template to be known at execution time
              <name> --template recognized by the name
               <•>   --template recognized by location=position
                       in list
              (name.name) --qualified name
               NAME(name) --entity occurrence to be not stored as
                            a separate position in the schema
               NAME(NAME) --entity occurrence to be stored
              (name=name) --names being synonims, when both names
                            remaines valid
              (name|name) --second name is replaced by the first one
                            only first name remains valid
              (name&name) --joined names
   <verb> ::= ( STATEMENT --specification statement ,
                command   --SOL language command )
   <control_word>::= word --word to be recognized by a syntax ana-
                          -lyzer
   <attribute>::= text
   <determiner>::=  ::=  --syntax definition
                    ::   --structural list
                    :=   --setting the value
                    =>   --control flow
                    ≡    --equivalence
                    --   --commentary
                    =    --contents
   <delimiter>::= <  >  --nonterminal text
                  {  }  --structural list
                  (  )  --list of names
                  [  ]  --optional usage
                    .   --end of statement if needed
```

⟨connective⟩ ::= / --"one-of" alternative
& --obligatory "and"
/& --cumulative usage,not obligatory occurrence.
¬ --negation
_ - --notational connectives inside the word
- --continuation mark between lines
line1-
-line2

⟨control_word⟩::= (label,body,flow,layout,begin,end,endef,loop, interface,time-out,procedure,subprogram,para-meter,input,output,operation,structural-descrip-tion, case, stack, data-buffer)

⟨statement⟩::= [IS]USED_IN ⟨names⟩,[IS]PART_OF ⟨name⟩,
USES ⟨names⟩ ,
[IS]INVOLVED-IN ⟨names⟩ ,
INVOLVES⟨ names ⟩,
[IS] CONTAINED-IN ⟨names⟩ ,
CONTAINS-OF ⟨names⟩ ,
[IS] SPECIFIED_IN ⟨name⟩ , ACTIVATED_BY ⟨name⟩ ,
[IS] ACTIVATED_UPON ⟨name/expression⟩ ,
ACTIVATES ⟨names⟩ ,
HAPPENS_WHEN ⟨expression⟩ ,
LINKED_TO ⟨names⟩ ,
DERIVED_FROM ⟨name⟩,
DIRECTED_TO⟨name⟩ ,
BELONGS_TO ⟨name⟩ ,
DEPENDS_ON ⟨expression⟩, INVOKED_BY ⟨name⟩,
INVOKED_FROM ⟨name⟩

A specification includes a user text also, e.g.description of requirements,restrictions, etc. Each syntactic unit can be equipped with a label which contains passwords, key description and some technical information necessary for file maintenance.

An entity definition consists of def control_word, entity type name, entity occurrence name, abstract type attribute, structure and flow description, user text, statements and endef control word.

There are distinguished following abstract types:

a) category type: process/object

b) function type: interface/driver/stub/main_program

c) structure type: whole/component/binder/cluster
--binder means collection of entities belonging to different types
--cluster is a collection of the same type entities

d) data type: real/integer/complex , array/aggregate/vector, segment/record , file , dbase , dset

e) status type: generic/ready/virtual/empty

Some of the listed above abstract types can be entities themselves and some entity types have the meaning of abstract types. For example, an interface abstract type can be defined as an INTERFACE entity type if it needs a separate definition. On the other-hand, a PACKAGE entity type represents a complex abstract type (family of generic programs). According to Deutch P. [4 p.49] we assume, that " a type is a precise characterization of structural or behavioral properties which is a collection of objects (actual or potential) all share. An instance of a type is an object which has the properties characteristic of the type." In the same time we agree with Thatcher J. that a data type is a family of sets together with a collection of operations or procedures on those sets [4 p.49] .

SL specification language covers all entities occurring in information system and its environment: problem, user, system, dbms, computer, programmer, analyst, decision, action, language, project, etc.
As it has been pointed, entities communicate and cooperate one with other by means of interfacing.
For example, interfaces are necessary to link following entities:

- languages,
- computers (in terms of internal code representation),
- data base management systems,
- dbms and user programs,
- main program and subprogram,
- user program and teleprocessing package,
- terminal and dbms,
- users,
- user and a query language .

Pattern of the interface specification

```
def INTERFACE ( ≠ & reent_proc_name)
   -- ≠ is a template for a program name which will be known at
     an execution time
   --reent_proc_name is a reenterable procedure name
   --reenterable procedure is that one which is only once brought in
     into a main storage and can satisfy requirements of any number
     of programs
   label
     password = xxx
     ...
   end label
   function_type : interface
   category_type : process
   status_type : generic/ready
   purpose
     interface retains the actual environment and the actual state of
     the reenterable procedure for each calling program( ≠ )
   end purpose
   language: PL&assembler
   oper_system : OS/VT
   call_rules
     indirect invokation from ≠
     password must be provided
   end call_rules
   restrictions
     calling program and reenterable procedure have to be written
     in the same programming language
   end restrictions
   requirements
     memory = 30 KB
     mass_storage = 30 MB disk
     OS/VT with LPA area
   end requirements
   LINKS PROCEDURE(name) TO PROGRAM( ≠ )
         TEMPORARY FOR CURRENT EXECUTION
         WHEN call(procedure_name) FROM ≠ .
   USES external resources: (buffer-handler, event-processor, stack-ha-
                             -ndler, dtuner ).
   IS DERIVED FROM PACKAGE(INTERFACE)
   IS ACTIVATED FROM event-buffer
   IS INVOLVED_FROM library(name)
   IPO --input processing output
       input: (reent_proc_name) CONTAINED_IN (sysl.linklib)
              parameters CONTAINED_IN stack( ≠ )
              data CONTAINED_IN buffer( ≠ )
       operation:--creation,loading and releasing areas neccessary for
                  each calling program,[tuning the body of reent-proc]
       output: DIRECTED_TO ( ≠ )
   end IPO
```

```
structural-description
  --this structural description is applicable to category_type:process
  --a process is recognized by a process code named pcode
  --an action is an optional part of the process
  --an event is recognized by event code named ecode
  PROCESS ::= {trigger,ACTION{ EVENT}, terminal-event}
    --this sentence is a predefinition of the process
  INTERFACE(<≠> & name )::
     {trigger(call) CONTAINED_IN event_buffer ,
      ACTION(initial) WHEN first(call)
        {EVENT(buffer-decl), EVENT(stack-decl),
          EVENT(verif-area-decl), EVENT(into-lpa-load)},
      ACTION(verification){EVENT(...), ... } ,
      ACTION(tuning) WHEN previous action not failed
        {EVENT(text-analyzer), EVENT(tune) WHEN body CONTAINS templates},
      ACTION(activation) WHEN (verification) not failed { ...} ,
      ACTION(last) WHEN stop-program(<≠>) {..},
      terminal-event(exit) } .

  pcode := pinterf
      number-of-actions = 5
      number-of-events  = 25
end structural-description

layout
  --storage structure of the entity
  procedure(main)
    number-of-lines = 52
    number-of-bytes =  5K
  procedure(initial)
    .....
  procedure(verification)
    .....
  procedure(tuning)
    (dtuner) INVOKED_FROM library(name)
      --(dtuner) is a dynamic tuner acting at the execution time
    .....
  procedure(activation)
    --it generates of call-statement of (reen-proc-name) based on activa-
      -tion record
  procedure(last)
    --it releases all resources involved in execution of (reent-proc-name)
  number-of-procedures = 6
  total-number-of-lines = 1200
  total-number-of-bytes = 15K
end layout
```

```
flow
  --(names)occurring in this section have been defined as actions
    or events in the structural description
  (initial) WHEN first(call) FROM <≠> DIRECTED_TO (reent-proc-name)
     declare:data-buffer,verification-area
     call stack-handler
     load (reent-proc-name) FROM (aysl.link.lib)
          WHEN NOT CONTAINED_IN  lpa
  (verification)
     check password CONTAINED_IN label
     verify call-statement against pattern(x)
     return-value := true/false
  (tuning)
     analyze def(reent-proc-name) whether do templates <≠> exist
        return-value := true/false
     call (dtuner) WHEN true  and execute (dtuner)
  (activation)
      set-up activation-record for <≠>
      call (reent-proc-name)
      store data in data-buffer(<≠>)WHEN return-value OF (reent-proc-na-
           -me) true
  (last)
     release resources {data-buffer(<≠>),activation-record(<≠>) ,
                        verif-area(<≠>), case( ≠ ) }
 end flow
end def

body
  --a body is an executive source program text
  --the body presented here is written in a simplified language(&DL) based
    partially on the language(Ada)
  begin
    'open' (interface-name)
      --this command invokes procedure(main)
    (main)
      --this procedure takes a new entry from event-buffer and controls
        the flow thru case(<≠>)
      'select' event FROM event-buffer
          WHEN ecode = call(reen-proc-name) and event-count > ∅
          'accept'
            event-count := event-count - 1
          end
          else 'terminate'
      end 'select'
```

```
        'execute' (initial)
            WHEN not the same (≠) or case((≠)) = 4
        end 'execute'
        case((≠)) := 2
        'execute' UPON case((≠))
            WHEN 2 ⇒ (verification)
                 3 ⇒ (activation)
                 4 ⇒ (last)
          'terminate' WHEN time-out
    end (main)

    (initial)
      --a link-pack-area = lpa is declared in a oper-system generation step
      --an event-buffer is declared by the event-processor
      'declare' data-buffer(≠)
        --a data-buffer is dedicated to store intermediate results when
          an interrupion has occurred
        ....
      end 'declare'
      'declare' case(≠)  data-type:integer 1
         case(≠) := 1
      end 'declare'
      'declare' stack(≠)
        --a stack(≠) is assigned to store an activation-record which
          provides a continuation from the point at which reenterable
          procedure was left off earlier
        (arg-name) < argument-value>  status-type = generic/ready
        ....
        (line-no)  <line-contents>  -for tuned lines of the body
       <number-of-arguments>
       <return-address>
       <saved-registers>
       <frame-pointer>
          --it points to the beginning of the current activation-record
       <argument-pointer>
          --it points to the block of storage which contains some additio-
            -nal parameters passed to the current state of subprogram
       <exception-handler-address>
        ....
      end 'declare'
    (verification)
      .....
      .....
      case(≠) := 3
    end (verification)
    (tuning)
      'select' argument FROM stack(≠)
        WHEN return-value of (verification) is true
        loop
          FOR position FROM 1 BY 1 UNTIL number-of-arguments
          WHILE status-type = generic
          'accept'
         'call' (dtuner)
        end loop
        IF none 'call' exception-handler with parameter = xxx
      end 'select'
    end (tuning)
    .....
    'close' (interface-name)
end body
```

Pattern of description of the reenterable procedure

```
def PROCEDURE(reent-proc-name)
  --this is a fragmentary definition focused on interaction with the
    interface
  --an example of the reenterable procedure is an interpretative report
    writer coded in assembler language
  category_type: process
  structure_type: component
  status_type: generic
  .....
  INVOKED_BY < ≠ > UPON call(reent-proc-name)
  ACTIVATED_BY INTERFACE(< ≠ > & reent-proc-name)
  SEQUENCED AFTER (verification)&(tuning) CONTAINED_IN INTERFACE()
  USES procedure(activation) CONTAINED_IN INTERFACE() .
  IS DERIVED_FROM (sysl.link.lib)
  IS CONTAINED_IN lpa.
  templates
    --templates description appear when status-type is generic
  <name>  oper-code:  1 --insert/
                      2 --delete/
                      3 --replace/
                      4 --copy with adress modification
          [line-number:<integer>]
            --a number of line which contains a template
  <name>  oper-code:  ....
    ....
  [precedence OF templates: (<name>, <name> , <name> ,  ....)]
     --this clause should be provided when line numbers are not spe-
       -cified
  end templates
  return-value DIRECTED_TO procedure(last)
  procedure(last) IS last PART_OF INTERFACE()
  structural-description
    CONTAINS_OF (page-composer, line-composer, summary-writer)
  < process structural description >
  end structural description
  flow
   <flow description >
  end flow
  layout
   <layout description >
  end layout
endef
```

Pattern of the specification of separate component of an interface

```
def TUNER(dtuner)
  --(dtuner) is an interpretative dynamic tuner which acts at execution
    time as an optional component of the interface between procedure
    and main program
  --(ctuner) is a conversational tuner which acts on a source text at
    precompilation time

  category-type: process
  status-type: ready
  function-type: interface

  IS INVOKED_BY (tuning)

                (tuning) IS PART_OF INTERFACE()

  IS ACTIVATED_BY (text-analyzer) iteratively WHEN templates

                  (text-analyzer) IS PART_OF (dtuner) .

  IS CONTAINED_IN library(name)

  structural-description

     (dtuner):: { trigger(call) FROM (tuning) WHEN status-type of proce-
                    -dure is generic

                  ACTION(text-analyzer){EVENT(line-selection), EVENT(sca-
                    -nner)} ,

                  ACTION(tune) { EVENT(template-selection), EVENT(transfo-
                    -rm)} ,

                  terminal-event }

  end structural-description

  IPO
    input: body OF (reent-proc-name) CONTAINED_IN lpa
           parameters CONTAINED_IN  stack(( ≠ ))

    operation: --tuning of body, storing modified lines in an activation-
                 record

    output: modified lines DIRECTED_TO (activation)
              (activation) IS PART_OF INTERFACE()

  end IPO

  IS SEQUENCED AFTER (tuning) BEFORE (activation)

  flow

    (text-analyzer)
       --iterative for each body line of the reenterable procedure

    (tune) WHEN templates
       --iterative for each accepted line
       --control flow upon case(dtuner) which depends on oper-code of
         templates

  end flow

  layout

    (text-analyzer)
    (tune)

  end layout

endef
```

```
body(dtuner)
  body.line-number := ∅
  loop
    FOR each line OF body(recnt-proc-name)
      'execute' (text-analyzer)
      loop
        FOR accepted line
        WHILE the same line-number or the same name
        'execute' (tune)
      end loop
    'exit' WHEN line = end body
    IF none accepted
      'call' exeption-handler with parameter = xxx
  end loop
  (text-analyzer)
    'select' line OF body(recnt-proc-name)
      body.line-number = body.line-number + 1
      'scan'characters OF line
        'accept' WHEN HAPPENS (≠)
      end
    end 'select'
  (tune)
    'select' templates FROM  def(recnt-proc-name)
      'accept'
        WHEN body.line-number = templates.line-number  or
             stack(≠). name = templates.name
      end
    end 'select'
    case(dtuner) := templates.oper-code
    'execute' UPON case(dtuner)
      WHEN 1 =  'insert' parameter FROM stack(≠)
           2 =  'delete' line
           3 =  'replace' line BY parameter
           4 =  'copy' line UPON address-modifier
    end 'execute'
    'store' IN activation-record(≠)
    end 'store'
    .....
end body
```

Concluding remarks

S&DL language discussed here can be considered as a ground for a manufacturing of a factory-tailored systems. SL specification sublanguage is a tool for description of objects involved in computerized information systems. &DL design sublanguage cooperates with SL and its purpose is creation of application software using tools like tuners and composers. The task of a tuner is converting skeleton texts of standard building blocks into texts that meet requirements of applications. Each building block has to fit into some standards which make possible flexible connections of it with other blocks thru interfaces. A composer deals with assembling elementary and main blocks into operable program, modifying data schema according to new data relationships, generating job control statements, etc. So, the final result of S&DL processor should be a family of programs , called "application package", based on the same components. Design (sub)language can be a selfcontained language, like Ada or proposed here &DL language, or may be an extension of host language like COBOL, PL/I or Pascal, and finally can be implemented as an intermediate language of macrolanguage type, main activity of which would be invoking ready-to-use procedures (e.g. text editors) embedded in operating system and in a macrolibrary.

References

[1] Langefors B. Theoretical analysis of information systems. Studentlitteratur Lund (cit.on page 48)

[2] Ryznar Z. S & DL Structured Design Language (proposal) Angewandte Informatik 12/81 526-533

[3] Ryznar Z. Free-structured Design of Computerized Information Systems (printiples and tools). Department of Mathematical Sciences University of Tampere Finland Report A62 october 1981

[4] Proceedings of workshop on data abstraction, databases and conceptual modelling. Pingree Park, Colorado June 23-26. 1980

[5] US DOD. Reference manual for the Ada programming language. Proposed standard document. July 1980