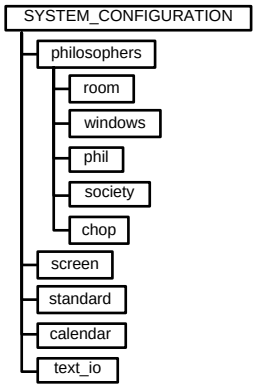
SYSTEM_CONFIGURATION IS

ROOT_OBJECTS
--|\\Groslulu\home4\stood\stood4.3\libs\calendar|--,
--|\\Groslulu\home4\stood\stood4.3\examples\nt_console|--,
--|\\Groslulu\home4\stood\stood4.3\examples\philosophers|--,
--|\\Groslulu\home4\stood\stood4.3\examples\screen|--,
--|\\Groslulu\home4\stood\stood4.3\libs\standard|--,
--|\\Groslulu\home4\stood\stood4.3\libs\text_io|--

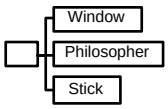
GENERIC
--|\\Groslulu\home4\stood\stood4.3\libs\discrete_random|--,
--|\\Groslulu\home4\stood\stood4.3\examples\random_generic|--

END

Design Tree

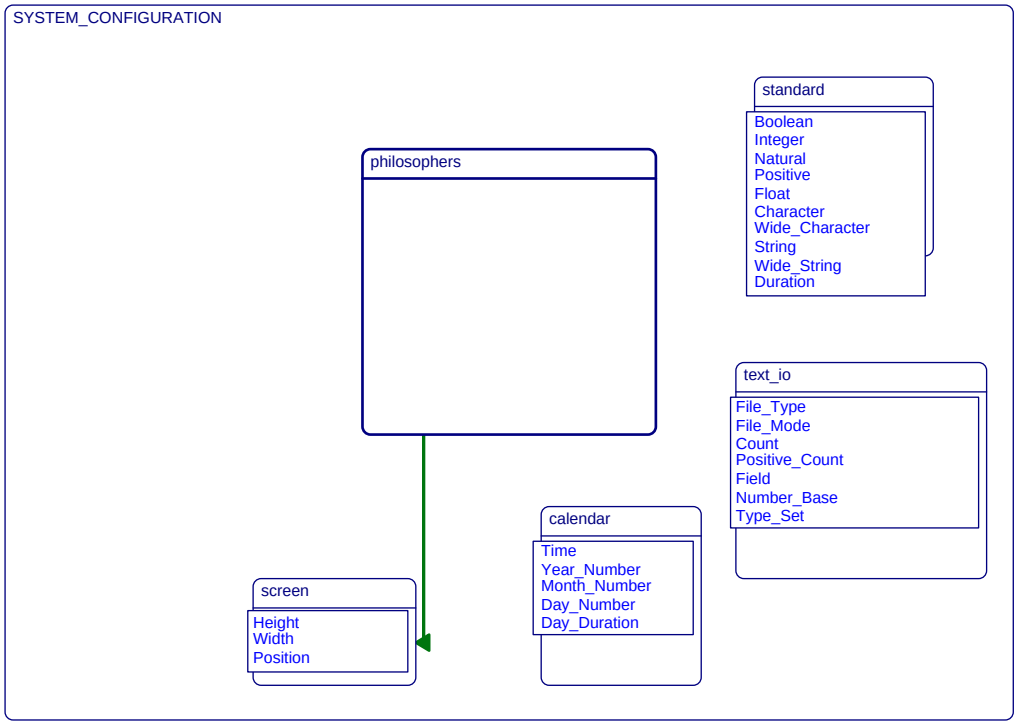


Inheritance Tree



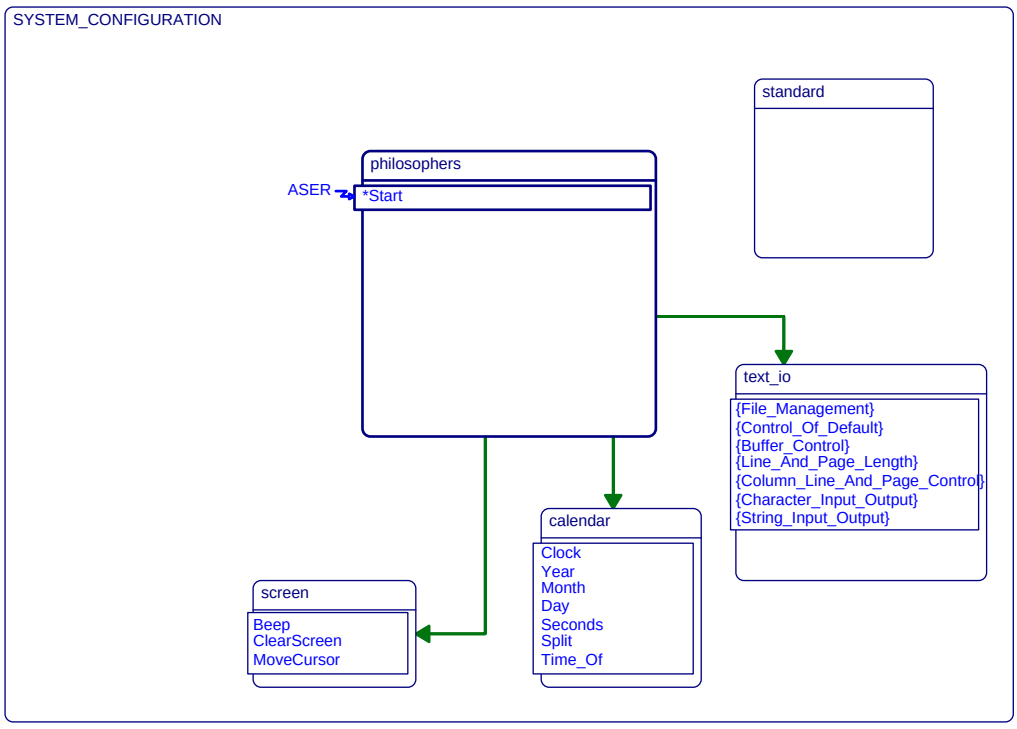
Structural (types) Diagram

type



Functional (oper.) Diagram

operation



List of Requirements

BR1/Concurrent_philosophers:
BR2/Shared_chopsticks:
BR3/Dynamic_events_report:
BR4/Synchronous_events_report:
BR5/Dining_room_states:
BR6/Philosopher_states:
FR1/Prepare&begin_diner:
FR2/Provide_chopsticks:
FR3/Report_events:
FR4/Open&initialize_window:
FR5/Write_messages:
FR6/Start_eating:
SR1/Dining_room_seats:
SR2/Philosophers:
SR3/Chopsticks:
SR4/Display_windows:
SR5/Simulation_timing:

OBJECT philosophers IS

PASSIVE

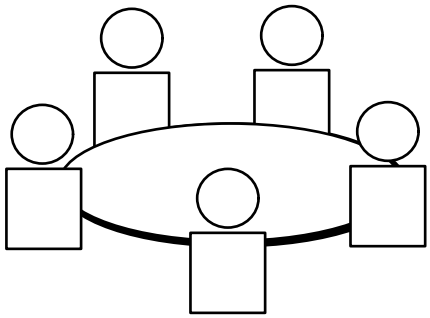
pragmas

```
PRAGMA line_feed
  (option => 1)
PRAGMA line_feed
  (option => 2)
PRAGMA main
  (operation_name => start,
   unit_name => run)
PRAGMA no_subunits
PRAGMA comment
PRAGMA compiler
  (name => gnat,
   options => --| |--)
```

DESCRIPTION

PROBLEM

Sketch of the Problem



Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Analysis of Requirements

Structural Requirements (text)

This application should manage:
SR1: the dining room (cf.SR1/Dining_room_seats:)
SR2: five instances of Philosopher (cf.SR2/Philosophers:)
SR3: five instances of shared chopstick (cf.SR3/Chopsticks:)
SR4: and display graphically simulation events on screen windows (cf.SR4/Display_windows:)

Philosophers and chopsticks implementation is shared into two distinct parts:

- abstract description within a relevant HOOD4 class
- concrete instantiation as data inside dining room

Functionnal Requirements (text)

The unique functional requirement at this higher level module is to launch the application. Start procedure is used as main subprogram of the application.
FR1: start the diner (cf.FR1/Prepare&begin_diner:)

Behavioural Requirements (text)

BR1: Each Philosopher should behave concurrently. (cf.BR1/Concurrent_philosophers:)
BR2: Each chopsticks must be shared between two Philosophers. (cf.BR2/Shared_chopsticks:)
BR3: Dining room must initiate the simulation and report events dynamically. (cf.BR3/Dynamic_events_report:)

Local Environment

Parent General Description (text)

In addition to usual Ada libraries (STANDARD, TEXT_IO and CALANDAR), a dedicated environment module is used to display information to the screen.
This "screen" module acts as a display device interface for our application. Two implementations may be used:
- an ANSI terminal emulator for UNIX workstations.
- a console interface for Windows 95 or NT PCs.

A generic module "random_generic" provides an interface to Ada.Numerics.Discrete_Random to implement an integer pseudo-random number generator.

SOLUTION

General Strategy (text)

This application was manually reverse engineered from Michael B. Feldman's Ada sources.
Each package pair is represented by a hood module, but of various kinds. Chosen strategy was to be able to re-generate code as close as possible from original one.
Other design choices could of course lead to other solutions.

Identification of Child Modules (text)

Philosophers application may be broken down into five modules:
- "society" simply provides a list of philosopher's name and ID. It is designed as a simple passive HOOD4 object.
- "room" describes the simulator logics, and instanciates statically main control task, and dynamically each philosopher. It is designed as an active HOOD4 object.
- "phil" is an abstract description of a dining philosopher. It is designed as an active HOOD4 class.
- "chop" is an abstract description of a shared chopstick. It is designed as a passive HOOD4 class with concurrency constrained operations.
- "window" is an abstract simple window manager. Is is designed as a passive HOOD4 class.

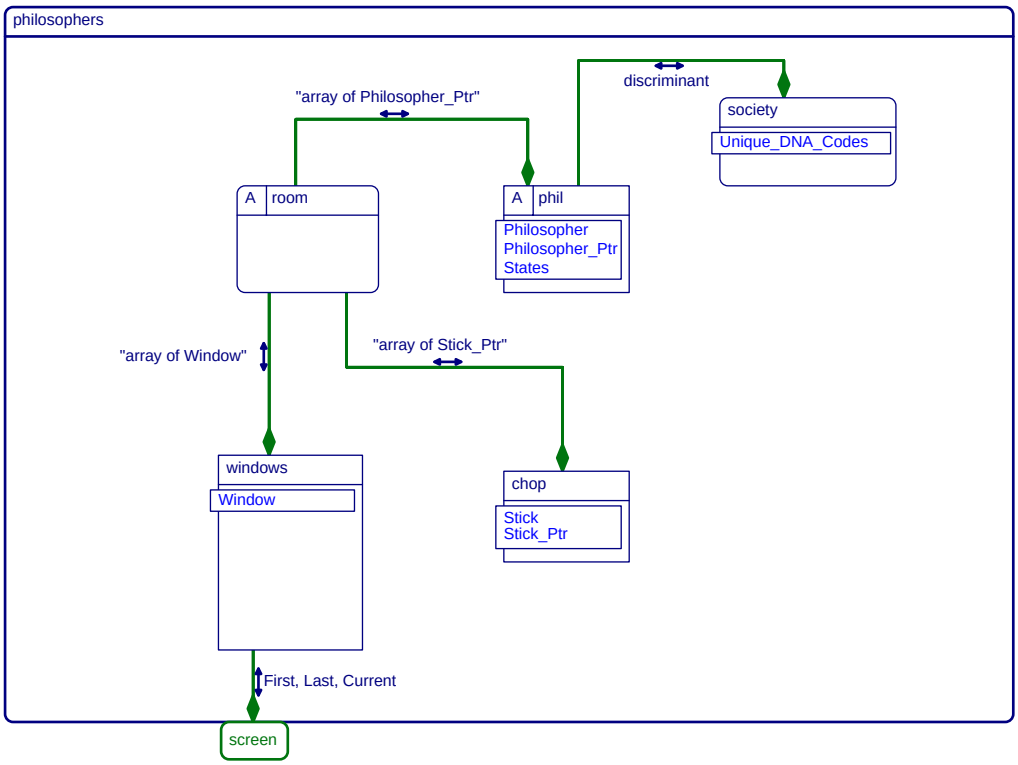
Structural Description

Identification of Data Structures (text)

None.

Structural (types) Diagram

type



Functional Description

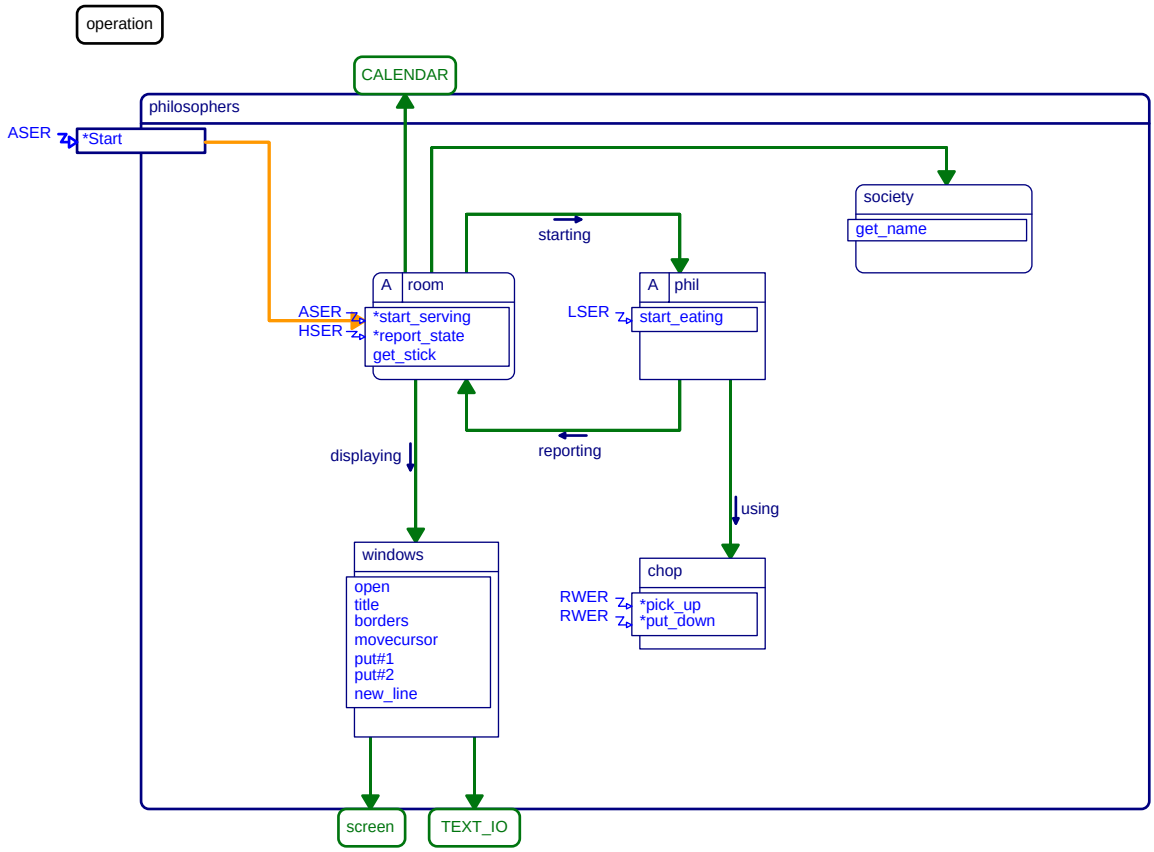
Identification of Operations (text)

Start is the main subprogram which calls room.start_serving.

Grouping Operations (text)

None.

Functional (oper.) Diagram



Behavioural Description

Identification of Local Behaviour (text)

Start is asynchronous.

Justification of Design Decisions (text)

Design choices comply with original Ada source code.
Another solution would have been to instantiate statically each Philosopher, and perhaps each chopstick. In this case th
they would have been designed respectively as instances of active generic modules and instances of passive generic mo
modules.

PROVIDED_INTERFACE

OPERATIONS

Start

operation spec. description (text)
Main procedure.

operation declaration (hood)
```
Start;
```

real time attributes (hood)
```
WCET
```

OBJECT_CONTROL_STRUCTURE

obcs spec. description (text)
The application is launched asynchronously.

constrained operations
```
Start CONSTRAINED_BY ASER STATE;
```

REQUIRED_INTERFACE
```
OBJECT calendar;
  TYPES
    Time;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    Clock;
  EXCEPTIONS
    NONE
OBJECT screen;
  TYPES
    Height; Position; Width;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    MoveCursor; ClearScreen;
  EXCEPTIONS
    NONE
OBJECT standard;
  TYPES
    Natural; Integer; Positive; Boolean; Character; String; Duration;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
OBJECT text_io;
  TYPES
    NONE
  CONSTANTS
    NONE
  OPERATION_SETS
```

```
  NONE
OPERATIONS
  Put; New_Line;
EXCEPTIONS
  NONE
```

INTERNALS

OBJECTS

```
room;
windows;
phil;
society;
chop;
```

OPERATIONS

Start

implemented_by

```
room.start_serving
```

OBJECT_CONTROL_STRUCTURE

implemented_by

```
room;
```

END philosophers

OBJECT room IS

ACTIVE

DESCRIPTION

PROBLEM

Statement of the Problem (text)

Room manages the simulation:
- intanciates Philosophers and Sticks.
- assigns each Philosopher a seat and his chopsticks.
- creates windows on the screen.
- displays information dynamically inside each window.(cf.FR3/Report_events:)

Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Analysis of Requirements

Structural Requirements (text)

Room must manage:
SR1: dining room seats (cf.SR1/Dining_room_seats:)
SR2: Philosophers (cf.SR2/Philosophers:)
SR3: chopsticks (cf.SR3/Chopsticks:)
SR4: display windows (cf.SR4/Display_windows:)
SR5: simulation timing (cf.SR5/Simulation_timing:)

Functionnal Requirements (text)

Room provides thre functional services:
FR1: prepare and begin the diner. (cf.FR1/Prepare&begin_diner:)
FR2: provide chopsticks to Philosophers. (cf.FR2/Provide_chopsticks:)
FR3: report events to outside world. (cf.FR3/Report_events:)

Behavioural Requirements (text)

Dining room behaviour should be as follow:
BR3: start diner asynchronously (cf.BR3/Dynamic_events_report:)
BR4: report event synchronously (cf.BR4/Synchronous_events_report:)
BR5: Dining room has two possible states (Waiting or Dining) (cf.BR5/Dining_room_states:)

Local Environment

Parent General Description (text)

Please refer to parent module description.

SOLUTION

General Strategy (text)

Dining room is designed as an active HOOD4 object, as it must have its own control flow.
Structural element (types, constants and data) are all hidden inside internal part.
Behaviour is described by a HOOD4 STD (State Transtion Diagram) and constraints on provided operations, and enca
encapsulated inside a HOOD OBCS (Object Control Structure).
Implementation of functional services are descibed directly inside HOOD OPCSs (Operation Control Structures).

Code generator will produce a package containing a task called OBCS.
(cf.BR5/Dining_room_states:)

Structural Description

Identification of Data Structures (text)

Structural elements are all hidden inside internals of this module, as none of them is required from outside.
They are listed below:

SR1:
- type Table_Type describes the table.
- constant Table_Size specifies the size of the table (5 seats).
- data Phil_Seats is used to allocate a seat to each Philosopher.
SR2:
- data P1 to P5 are five instances of Phil.Philosopher.
- data Phils is an array of pointers on these Philosophers.
SR3:
- data S1 to S5 are five instances of Chop.Stick.
- data Sticks is an array of pointers on these chopsticks.
SR4:
- data Phil_Windows is an array of five Windows.window.
SR5:
- data Start_Time is initialized when simulation starts.
- data T provides current simulation time.

Functional Description

Identification of Operations (text)

Operations provided by dining room are:

FR1:
Start_Serving: sets up the table and start the diner.
It is called by main procedure of the program.
FR2:
Get_Stick: implements the request from a Philosopher to pick up a chopstick.
This procedure didn't exist inside original Ada code where chopsticks were declared as public data, and were thus d directly visible from Phil module. Provided data being forbidden when designing with HOOD, chopstick instances w were declared within the internals, and Get_Stick access function was added to implement remote requests.
FR3:
Report_State: is used by Philosophers to indicate in which internal state they are.

Grouping Operations (text)

None.

Behavioural Description

Identification of Local Behaviour (text)

Dining room behaviour is represented by a State Transition model and constraints on operation execution requests:

BR5: Dining room has two distinct states:
- waiting state, identified by internal state variable "started" set to FALSE, where Room may only receive "Start_Se "Start_Serving" execution requests.
- dining state, identified by internal state variable "started" set to TRUE, where Room may only receive "Report_Sta "Report_State" execution requests.

BR3: Start_Serving has an asynchronous execution request (ASER).

BR4: Report_State has a highly synchronous execution request (HSER), so that Philosopher 's internal state doesn't change while current state is displayed on relevant window.

Justification of Design Decisions (text)

A few changes in initial source code were required to fit HOOD4 design rules:
- Sticks variable was initially declared within package spec, which is forbidden with HOOD, so it was moved into pack package body, and an additional access function (Get_Stick) may be used to pick_up on one of the five chopsticks.
- As chopsticks are instances of a protected type, which is thus a limited type, they cannot be returned directly by an a access function. Sticks thus became an array of pointers on chopsticks.
- To comply with standard HOOD code generation rules, task entries are not directly called from outside. Remote clien clients should call Room.Start_Serving and Room.Report_State which remame task entries of the same name. For the s

same reason, relevant bodies are implemented into additional internal OPCS_Start_serving and OPCS_Report_State pr procedures.
- Within original code, main control task was called "Maitre_D". Its name become "OBCS" when generated from a HO HOOD design. Task body is also automatically generated from STD and operation constraints: this implies changes to code structure.

PROVIDED_INTERFACE

OPERATIONS

start_serving

operation spec. description (text)
Room.Start_Serving is called by main procedure and renames OBCS.Start_Serving task entry.
This procedure has no parameter.
(cf.FR1/Prepare&begin_diner:)

operation declaration (hood)
```
start_serving;
```

real time attributes (hood)
```
WCET
```

report_state

operation spec. description (text)
Room.Report_State is called by Phil.Start_Eating and renames OBCS.Report_State task entry.
This procedure has four parameters:
- Which_Phil: identifies actual Philosopher sending the message.
- State: current state of sender.
- How_Long: length of current state (or identifier of used chopstick).
- Which_Meal: current meal.
(cf.FR3/Report_events:)

operation declaration (hood)
```
report_state(
  Which_Phil : in Society.Unique_DNA_Codes;
  Which_State : in Phil.States;
  How_Long : in Natural := --|0|--;
  Which_Meal : in Natural := --|0|--
);
```

real time attributes (hood)
```
WCET
```

get_stick

operation spec. description (text)
Room.Get_Stick is an access function to internal Sticks variable.
It requires a chopstick ID (Which_Stick) to return a pointer to relevant protected object.
(cf.FR2/Provide_chopsticks:)

operation declaration (hood)
```
get_stick(which_Stick : in Positive) return Chop.Stick_Ptr;
```

real time attributes (hood)
```
WCET
```

OBJECT_CONTROL_STRUCTURE

obcs spec. description (text)

A dedicated state variable manages current state of dining room.
This variable "Started" has a default value of "FALSE" and become "TRUE" after Start_Serving has been executed.
Start_Serving and Report_State have both STATE and protocole constraints.

constrained operations

```
start_serving CONSTRAINED_BY ASER STATE;
report_state CONSTRAINED_BY HSER STATE;
```

REQUIRED_INTERFACE

```
OBJECT calendar;
  TYPES
    Time;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    Clock;
  EXCEPTIONS
    NONE
OBJECT chop;
  TYPES
    Stick_Ptr; Stick;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
OBJECT phil;
  TYPES
    States; Philosopher; Philosopher_Ptr;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    start_eating;
  EXCEPTIONS
    NONE
OBJECT society;
  TYPES
    Unique_DNA_Codes;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    get_name;
  EXCEPTIONS
    NONE
OBJECT standard;
  TYPES
    Natural; Integer; Positive; Boolean;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
```

```
    NONE
OBJECT windows;
  TYPES
    Window;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    open; borders; title; put#1; new_line;
  EXCEPTIONS
    NONE
```

DATAFLOWS

```
starting => phil;
displaying => windows;
```

INTERNALS

TYPES

Table_Type

type description (text)

Identifies the possible locations around the table.

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
subtype Table_Type is Positive range 1..Table_Size;
```

CONSTANTS

Table_Size

constant description (text)

Specifies the total number of seats around the table. It is limited to five in this example.

constant definition (ada)

```
Table_Size : constant := 5;
```

DATA

S1

data description (text)

First chopstick shared between seats 5 and 1.

data declaration (ada)

```
S1 : aliased Chop.Stick;
```

data access from pseudo_code
```
(da) room.S1 IS USED BY NONE
```

data access from Ada code
```
(da) room.S1 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.S1 IS USED BY NONE
```

data access from C++ code
```
(da) room.S1 IS USED BY NONE
```

S2

data description (text)
second chopstick shared between seats 1 and 2.

data declaration (ada)
```
S2 : aliased Chop.Stick;
```

data access from pseudo_code
```
(da) room.S2 IS USED BY NONE
```

data access from Ada code
```
(da) room.S2 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.S2 IS USED BY NONE
```

data access from C++ code
```
(da) room.S2 IS USED BY NONE
```

S3

data description (text)
Third chopstick shared between seats 2 and 3.

data declaration (ada)
```
S3 : aliased Chop.Stick;
```

data access from pseudo_code
```
(da) room.S3 IS USED BY NONE
```

data access from Ada code
```
(da) room.S3 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.S3 IS USED BY NONE
```

data access from C++ code
```
(da) room.S3 IS USED BY NONE
```

S4

data description (text)
Fourth chopstick shared between seats 3 and 4.

data declaration (ada)
```
S4 : aliased Chop.Stick;
```

data access from pseudo_code
```
(da) room.S4 IS USED BY NONE
```

data access from Ada code
```
(da) room.S4 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.S4 IS USED BY NONE
```

data access from C++ code
```
(da) room.S4 IS USED BY NONE
```

S5

data description (text)
Fifth chopstick shared between seats 4 and 5.

data declaration (ada)
```
S5 : aliased Chop.Stick;
```

data access from pseudo_code
```
(da) room.S5 IS USED BY NONE
```

data access from Ada code
```
(da) room.S5 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.S5 IS USED BY NONE
```

data access from C++ code
```
(da) room.S5 IS USED BY NONE
```

Sticks

data description (text)
An array of pointers to the chopsticks.

data declaration (ada)
```
Sticks : array (Table_Type) of Chop.Stick_Ptr;
```

data access from pseudo_code
```
(da) room.Sticks IS USED BY NONE
```

data access from Ada code
```
(da) room.Sticks IS USED BY
  (op) room.get_stick [R]
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.Sticks IS USED BY NONE
```

data access from C++ code
```
(da) room.Sticks IS USED BY NONE
```

P1

data description (text)
First Philosopher.

data declaration (ada)
```
P1 : aliased Phil.Philosopher(My_ID => 1);
```

data access from pseudo_code
```
(da) room.P1 IS USED BY NONE
```

data access from Ada code
```
(da) room.P1 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.P1 IS USED BY NONE
```

data access from C++ code
```
(da) room.P1 IS USED BY NONE
```

P2

data description (text)
Second Philosopher.

data declaration (ada)
```
P2 : aliased Phil.Philosopher(My_ID => 2);
```

data access from pseudo_code
```
(da) room.P2 IS USED BY NONE
```

data access from Ada code
```
(da) room.P2 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.P2 IS USED BY NONE
```

data access from C++ code
```
(da) room.P2 IS USED BY NONE
```

P3

data description (text)
Third Philosopher.

data declaration (ada)
```
P3 : aliased Phil.Philosopher(My_ID => 3);
```

data access from pseudo_code
```
(da) room.P3 IS USED BY NONE
```

data access from Ada code
```
(da) room.P3 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.P3 IS USED BY NONE
```

data access from C++ code
```
(da) room.P3 IS USED BY NONE
```

P4

data description (text)
Fourth Philosopher.

data declaration (ada)
```
P4 : aliased Phil.Philosopher(My_ID => 4);
```

data access from pseudo_code
```
(da) room.P4 IS USED BY NONE
```

data access from Ada code
```
(da) room.P4 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.P4 IS USED BY NONE
```

data access from C++ code
```
(da) room.P4 IS USED BY NONE
```

P5

data description (text)
Fifth Philosopher.

data declaration (ada)
```
P5 : aliased Phil.Philosopher(My_ID => 5);
```

data access from pseudo_code
```
(da) room.P5 IS USED BY NONE
```

data access from Ada code
```
(da) room.P5 IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.P5 IS USED BY NONE
```

data access from C++ code
```
(da) room.P5 IS USED BY NONE
```

Phils

data description (text)
An array of pointers to the Philosophers.

data declaration (ada)
```
Phils : array (Table_Type) of Phil.Philosopher_Ptr;
```

data access from pseudo_code
```
(da) room.Phils IS USED BY NONE
```

data access from Ada code
```
(da) room.Phils IS USED BY
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.Phils IS USED BY NONE
```

data access from C++ code
```
(da) room.Phils IS USED BY NONE
```

Phil_Windows

data description (text)
An array of windows. One window for each seat.

data declaration (ada)
```
Phil_Windows : array (Table_Type) of Windows.Window;
```

data access from pseudo_code
```
(da) room.Phil_Windows IS USED BY NONE
```

data access from Ada code
```
(da) room.Phil_Windows IS USED BY
  (op) room.report_state [R]
  (op) room.start_serving [R]
```

data access from C code
```
(da) room.Phil_Windows IS USED BY NONE
```

data access from C++ code
```
(da) room.Phil_Windows IS USED BY NONE
```

Phil_Seats

data description (text)
An array to indicate which seat each Philosopher occupies:
Philosopher 1 occupies seat 1;
Philosopher 2 occupies seat 3;
Philosopher 3 occupies seat 5;
Philosopher 4 occupies seat 4;
Philosopher 5 occupies seat 2;

data declaration (ada)
```
Phil_Seats : array (Society.Unique_DNA_Codes) of Table_Type;
```

data access from pseudo_code

```
(da) room.Phil_Seats IS USED BY NONE
```

data access from Ada code

```
(da) room.Phil_Seats IS USED BY
  (op) room.report_state [R]
  (op) room.start_serving [R]
```

data access from C code

```
(da) room.Phil_Seats IS USED BY NONE
```

data access from C++ code

```
(da) room.Phil_Seats IS USED BY NONE
```

T

data description (text)

Current time obtained by Calendar.Clock.

data declaration (ada)

```
T : Natural;
```

data access from pseudo_code

```
(da) room.T IS USED BY NONE
```

data access from Ada code

```
(da) room.T IS USED BY
  (op) room.report_state [R]
```

data access from C code

```
(da) room.T IS USED BY NONE
```

data access from C++ code

```
(da) room.T IS USED BY NONE
```

Start_Time

data description (text)

Time when application is launched.

data declaration (ada)

```
Start_Time : Calendar.Time;
```

data access from pseudo_code

```
(da) room.Start_Time IS USED BY NONE
```

data access from Ada code

```
(da) room.Start_Time IS USED BY
  (op) room.report_state [R]
  (op) room.start_serving [R]
```

data access from C code

```
(da) room.Start_Time IS USED BY NONE
```

data access from C++ code
```
(da) room.Start_Time IS USED BY NONE
```

Started

data description (text)
State variable to switch between "waiting" and "dining" states.
Initial state is "Waiting".

data declaration (ada)
```
Started : boolean := false;
```

data access from pseudo_code
```
(da) room.Started IS USED BY NONE
```

data access from Ada code
```
(da) room.Started IS USED BY
  (op) room.start_serving [R]
```

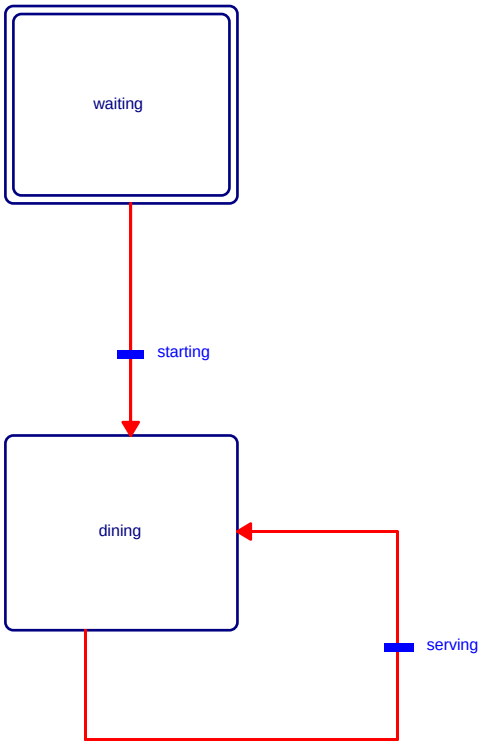
data access from C code
```
(da) room.Started IS USED BY NONE
```

data access from C++ code
```
(da) room.Started IS USED BY NONE
```

OBJECT_CONTROL_STRUCTURE

obcs body description (text)
OBCS is automatically generated from STD and operation constraints.

state transition diagram



waiting

exiting transitions
starting

state description (text)
Initial state. Started is set to FALSE.

state assignment (ada)
```
Started := false;
```

state test (ada)
```
not Started
```

dining

entering transitions
starting, serving

exiting transitions
serving

state description (text)
Running state. Started is set to TRUE.

state assignment (ada)
```
Started := true;
```

state test (ada)
```
Started
```

starting

transition event
start_serving

transition from
waiting

transition to
dining

trans description (text)
This transition is triggered by Start_Serving execution request.
No additional condition, neither exception code is required.

serving

transition event
report_state

transition from
dining

transition to
dining

trans description (text)

This transition is triggered by Report_Sta te execution request.
No additional condition, neither exception code is required.
Current state is not changed.

OPERATION_CONTROL_STRUCTURES

OPERATION start_serving IS

operation body description (text)

Performs following actions:
- Calculates Start_Time;
- Puts chopsticks on the table;
- Assigns Philosophers to seats at the table;
- Opens and draw a window to observe each seat;
- Assigns right and left chopsticks to each Philosopher;

used operations

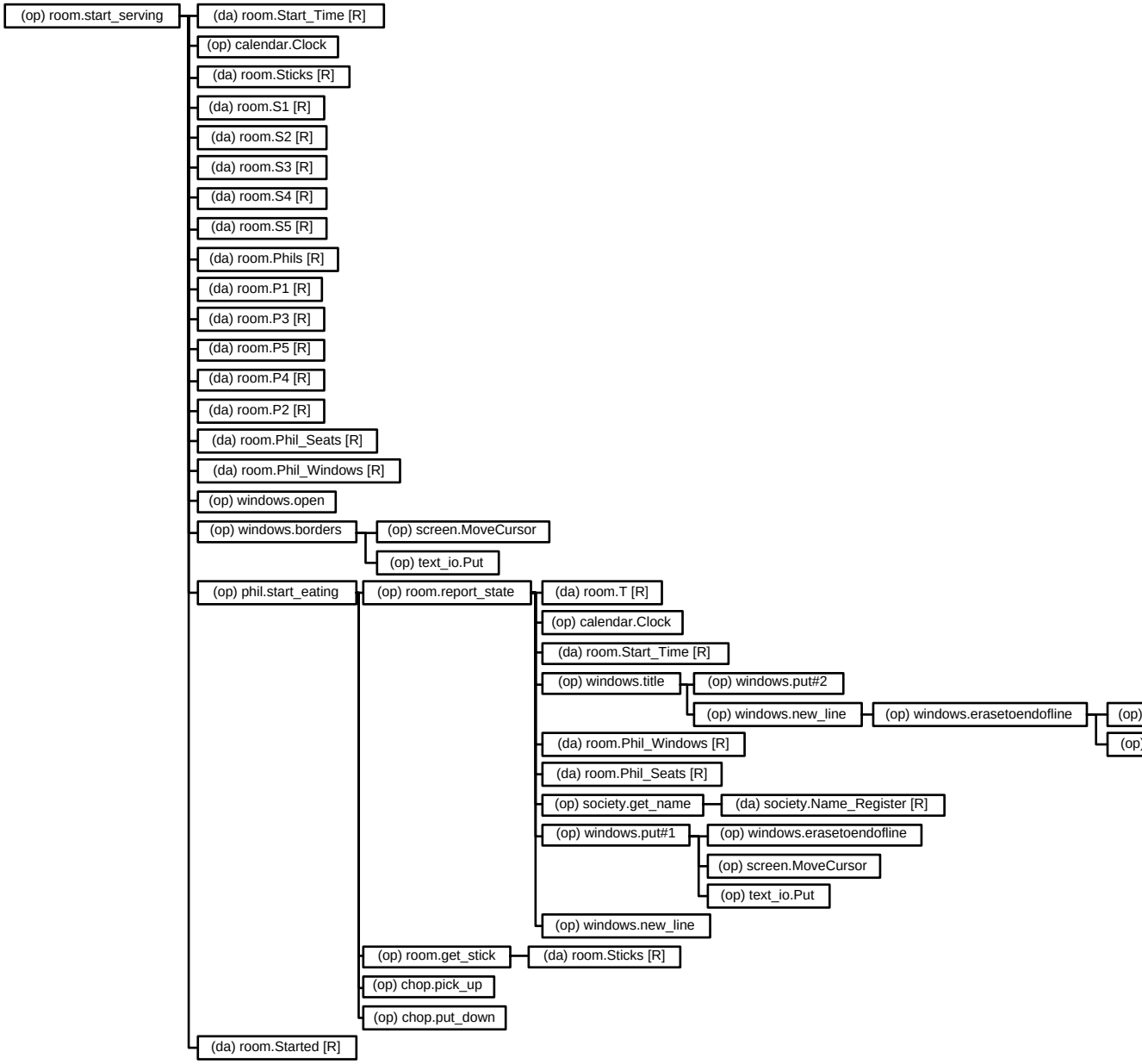
```
calendar.Clock
windows.open
windows.borders
phil.start_eating
```

operation code (ada)

```
begin

  -- starting date is stored:
  Start_Time := Calendar.Clock;

  -- chopsticks are put on the table:
  Sticks :=
    (S1'Access,
     S2'Access,
     S3'Access,
     S4'Access,
     S5'Access);

  -- philosophers are assigned to seats at the table
  Phils :=
    (P1'Access,
     P3'Access,
     P5'Access,
     P4'Access,
     P2'Access);

  -- which seat each phil occupies:
  Phil_Seats := (1, 3, 5, 4, 2);

  -- a window is open for each seat:
  Phil_Windows :=
    (Windows.Open(( 1, 24), 7, 30),
     Windows.Open(( 9,  2), 7, 30),
     Windows.Open(( 9, 46), 7, 30),
     Windows.Open((17,  7), 7, 30),
     Windows.Open((17, 41), 7, 30));

  -- windows borders are drawn:
  for Which_Win in Phil_Windows'range loop
    Windows.Borders(Phil_Windows(Which_Win),'+','|','-');
  end loop;

  -- philosophers are assigned their chopsticks:
  Phils (1).Start_Eating(1, 1, 2);
  Phils (3).Start_Eating(3, 3, 4);
  Phils (2).Start_Eating(2, 2, 3);
```

```
Phils (5).Start_Eating(5, 1, 5);
Phils (4).Start_Eating(4, 4, 5);

-- dining room state changes:
Started := true;
```

call tree from Ada code



END start_serving

OPERATION report_state IS

operation body description (text)

Performs following actions:
- Calculates current time;
- Displays a message on relevant window.

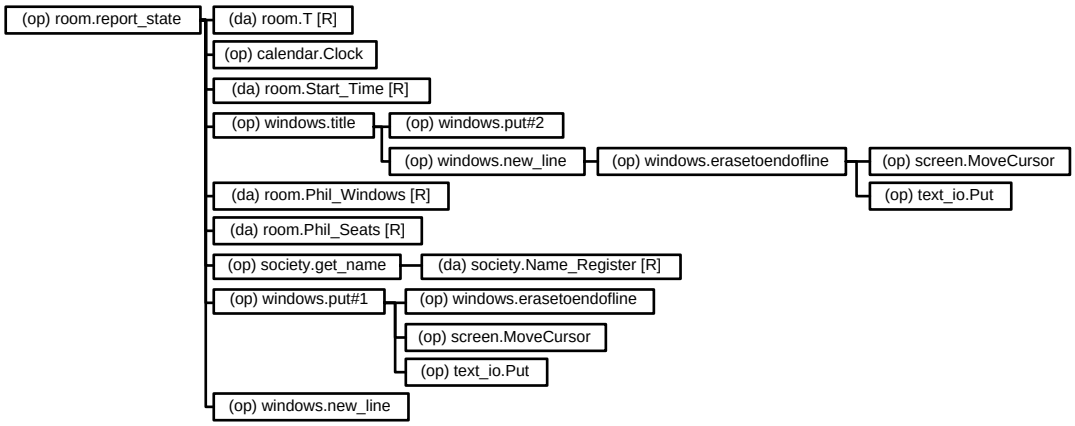
used operations

```
calendar.Clock
windows.title
society.get_name
windows.put
windows.new_line
```

operation code (ada)

```
begin

  T := Natural (Calendar.Clock - Start_Time);

  case Which_State is

    when Phil.Breathing =>
      Windows.Title(
          Phil_Windows(Phil_Seats(Which_Phil)),
          Society.Get_Name(Which_Phil), '-');
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Breathing...");
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Thinking =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Thinking"
          & Integer'Image (How_Long) & " seconds.");
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Eating =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Meal"
          & Integer'Image (Which_Meal) & ","
          & Integer'Image (How_Long) & " seconds.");
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Done_Eating =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Yum-yum (burp)");
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Got_One_Stick =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "First chopstick"
          & Integer'Image (How_Long));
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Got_Other_Stick =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Second chopstick"
          & Integer'Image (How_Long));
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

    when Phil.Dying =>
      Windows.Put(
          Phil_Windows(Phil_Seats(Which_Phil)),
          "T =" & Integer'Image (T) & " " & "Croak");
      Windows.New_Line(Phil_Windows(Phil_Seats(Which_Phil)));

  end case; -- Which_State
```

call tree from Ada code



END report_state

OPERATION get_stick IS

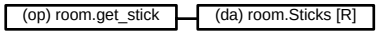
operation body description (text)

Just returns a pointer to specified chopstick.

operation code (ada)

```
begin
  return Sticks(Which_Stick);
```

call tree from Ada code



END get_stick

END room

CLASS windows IS

PASSIVE

pragmas

```
PRAGMA init_bloc
  (init_op => initialize)
```

DESCRIPTION

PROBLEM

Statement of the Problem (text)

Manager for simple, nonoverlapping windows for alpha-numeric console.

Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Analysis of Requirements

Structural Requirements (text)

SR4: This modules implements an abstract simple window manager. (cf.SR4/Display_windows:)

Functionnal Requirements (text)

FR4: Open and initialize a window. (cf.FR4/Open&initialize_window:)
FR5: Write messages at a specified location on the window. (cf.FR5/Write_messages:)

Behavioural Requirements (text)

Windows are passive unshared objects. There is no particular behaviour requirements.

Local Environment

Parent General Description (text)

Please refer to parent module description.

SOLUTION

General Strategy (text)

Window module is designed as a passive HOOD4 class.
It describes a window data structure and all relevant functional services.

Code generator will produce a package containing a tagged type.

Structural Description

Identification of Data Structures (text)

SR4: type Window is a class which attributes describe first, last and current positions of the cursor on the screen.

Functional Description

Identification of Operations (text)

Following operations are primitive operations of type Window:
FR4:
- Initialize: package init block is used to clear the screen.
- Open: instanciates a new window and initialize its attributes with passed values.
- Title: writes a title and optionnaly a separation line.
- Borders: draws top, right, left and bottom lines.

FR5:

- MoveCursor: sets current cursor position.
- Put#1: writes a character at current cursor position.
- Put#2: writes a string at current cursor location.
- New_Line: puts current cursor position at the beginning of next line.
- EraseToEndOfLine: erases from current location to the end of line.

Grouping Operations (text)

None

Behavioural Description

Identification of Local Behaviour (text)

None

Justification of Design Decisions (text)

A few changes in initial source code were done to best fit HOOD4 design rules:
- Window was declared as a HOOD4 class, so code generator produces a tagged type by default.
- Name of the parameter of type Window in all primitive operation declarations was set to "me".

PROVIDED_INTERFACE

TYPES

Window

type description (text)

First : coordinates of upper left corner;
Last : coordinates of lower right corner;
Current : current cursor position.

class inheritance (hood)

```
INHERITANCE NONE
```

type attributes (hood)

```
ATTRIBUTES First : screen.Position, Last : screen.Position, Curre
Current : screen.Position
```

type enumeration (hood)

```
ENUMERATION NONE
```

type pre-declaration (ada)

```
type Window is private;
```

OPERATIONS

open

operation spec. description (text)

Pre: UpperLeft, Weight, and Width are defined
Post: returns a Window with the given upper-left corner, height, and width

operation declaration (hood)

```
open(
  UpperLeft : in Screen.Position;
  Height : in Screen.Height;
  Width : in Screen.Width
)   return Window;
```

real time attributes (hood)
WCET

title

operation spec. description (text)
Pre: me, Name, and Under are defined
Post: Name is displayed at the top of the window me, underlined with the character Under

operation declaration (hood)
```
title(
  me : in out Window;
  Name : in String;
  Under : in Character
);
```

real time attributes (hood)
WCET

borders

operation spec. description (text)
Pre: All parameters are defined
Post: Draw border around current writable area in window with characters specified.
Call this BEFORE Title.

operation declaration (hood)
```
borders(
  me : in out Window;
  Corner : in Character;
  Down : in Character;
  Across : in Character
);
```

real time attributes (hood)
WCET

movecursor

operation spec. description (text)
Pre: me, and P are defined, and P lies within the area of me
Post: Cursor is moved to the specified position.
Coordinates are relative to the upper left corner of me, which is (1,1)

operation declaration (hood)
```
movecursor(me : in out Window; P : in Screen.Position);
```

real time attributes (hood)
WCET

put#1

operation spec. description (text)
Pre: me, and Ch are defined.
Post: Ch is displayed in the window at the next available position.
If end of column, go to the next row.
If end of window, go to the top of the window.

operation declaration (hood)
```
put#1(me : in out Window; Ch : in Character);
```

real time attributes (hood)
```
WCET
```

put#2

operation spec. description (text)
Pre: me, and S are defined.
Post: Ch is displayed in the window, "line-wrapped" if necessary

operation declaration (hood)
```
put#2(me : in out Window; S : in String);
```

real time attributes (hood)
```
WCET
```

new_line

operation spec. description (text)
Pre: me is defined.
Post: Cursor moves to beginning of next line of me;
line is not blanked until next character is written

operation declaration (hood)
```
new_line(me : in out Window);
```

real time attributes (hood)
```
WCET
```

REQUIRED_INTERFACE

```
OBJECT screen;
  TYPES
    Height; Position; Width;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    MoveCursor; ClearScreen;
  EXCEPTIONS
    NONE
OBJECT standard;
  TYPES
    Character; String;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
OBJECT text_io;
  TYPES
    NONE
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    Put; New_Line;
  EXCEPTIONS
    NONE
```

INTERNALS

OPERATIONS

erasetoendofline

operation spec. description (text)
Used to erase partially the screen.

operation declaration (hood)
```
erasetoendofline(me : in out Window);
```

real time attributes (hood)
```
WCET
```

initialize

operation spec. description (text)
This HOOD operation will not be generated as an Ada procedure, but as package initialization block.
This result is obtained by setting pragma "init_bloc(initialize)" when generating Ada code.

operation declaration (hood)
```
initialize;
```

real time attributes (hood)
```
WCET
```

OPERATION_CONTROL_STRUCTURES

OPERATION open IS

operation body description (text)
Instanciates a new Window named "Result"
Sets Result attributes (Current, First and Last)
Returns Result.

operation code (ada)
```
  Result : Window;
begin
  Result.Current := UpperLeft;
  Result.First := UpperLeft;
  Result.Last := (Row => UpperLeft.Row + Height - 1,
      Column => UpperLeft.Column + Width - 1);
  return Result;
```

END open

OPERATION title IS

operation body description (text)
Sets cursor at the beginning of first line.
Writes title string
If "Under" is blank then continue
else draw a separation line
Reduces writable area as required.

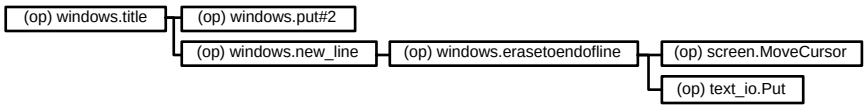
used operations

```
windows.put
windows.new_line
```

operation code (ada)

```
begin

  -- Put name on top line
  me.Current := me.First;
  Put(me, Name);
  New_Line(me);

  -- Underline name if desired, and reduce the writable area
  -- of the window by one line
  if Under = ' ' then
    -- no underlining
    me.First.Row := me.First.Row + 1;
  else
    -- go across the row, underlining
    for Count in me.First.Column..me.Last.Column loop
      Put(me, Under);
    end loop;
    New_Line(me);
    -- reduce writable area
    me.First.Row := me.First.Row + 2;
  end if;
```

call tree from Ada code



END title

OPERATION borders IS

operation body description (text)

Draws top line border.
Draws the two side lines.
Draws the bottom line of the border.
Make the Window smaller by one character on each side.

used operations

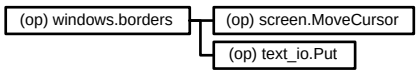
```
screen.MoveCursor
text_io.Put
```

operation code (ada)

```
begin

  -- Put top line of border
  Screen.MoveCursor(me.First);
  Text_IO.Put(Corner);
  for Count in me.First.Column+1 .. me.Last.Column-1 loop
    Text_IO.Put(Across);
  end loop;
  Text_IO.Put(Corner);

  -- Put the two side lines
  for Count in me.First.Row+1 .. me.Last.Row-1 loop
    Screen.MoveCursor((Row => Count,Column => me.First.Column));
    Text_IO.Put(Down);
    Screen.MoveCursor((Row => Count,Column => me.Last.Column));
```

```
    Text_IO.Put(Down);
  end loop;

  -- Put the bottom line of the border
  Screen.MoveCursor((Row => me.Last.Row,Column => me.First.Column
me.First.Column));
  Text_IO.Put(corner);
  for Count in me.First.Column+1 .. me.Last.Column-1 loop
    Text_IO.Put (Across);
  end loop;
  Text_IO.Put(Corner);

  -- Make the Window smaller by one character on each side
  me.First := (Row => me.First.Row+1,Column => me.First.Column+1)
me.First.Column+1);
  me.Last := (Row => me.Last.Row-1,Column => me.Last.Column-1);
  me.Current := me.First;
```

call tree from Ada code



END borders

OPERATION movecursor IS

operation body description (text)

Cursor position passed as parameter is relative to window boundaries.

operation code (ada)

```
-- Relative to writable Window boundaries, of course
begin
  me.Current.Row := me.First.Row + P.Row;
  me.Current.Column := me.First.Column + P.Column;
```

END movecursor

OPERATION put#1 IS

operation body description (text)

If at end of current line then move to next line.
If at beginning of current line then erase the entire line.
Writes given character.

used operations

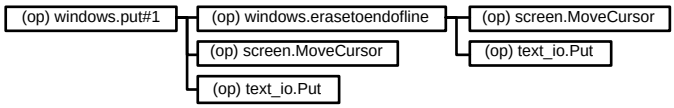
```
windows.erasetoendofline
screen.MoveCursor
text_io.Put
```

operation code (ada)

```
begin

  -- If at end of current line, move to next line
  if me.Current.Column > me.Last.Column then
    if me.Current.Row = me.Last.Row then
      me.Current.Row := me.First.Row;
    else
      me.Current.Row := me.Current.Row + 1;
    end if;
    me.Current.Column := me.First.Column;
  end if;

  -- If at First char, erase line
```

```
  if me.Current.Column = me.First.Column then
    EraseToEndOfLine(me);
  end if;

  Screen.MoveCursor(To => me.Current);

  -- here is where we actually write the character!
  Text_IO.Put(Ch);
  me.Current.Column := me.Current.Column + 1;
```

call tree from Ada code



END put#1

OPERATION put#2 IS

operation body description (text)

Uses put#1 to write each character of the string.

operation code (ada)

```
begin
  for Count in S'Range loop
    Put(me, S (Count));
  end loop;
```

END put#2

OPERATION new_line IS

operation body description (text)

If cursor is at beginning of a line then first erase this line.
If cursor is on last line then put it on first line.
Else put it on next line.

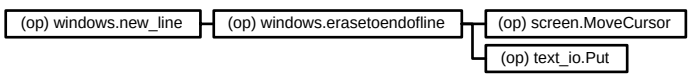
used operations

```
windows.erasetoendofline
```

operation code (ada)

```
begin
  if me.Current.Column = 1 then
    EraseToEndOfLine(me);
  end if;
  if me.Current.Row = me.Last.Row then
    me.Current.Row := me.First.Row;
  else
    me.Current.Row := me.Current.Row + 1;
  end if;
  me.Current.Column := me.First.Column;
```

call tree from Ada code



END new_line

OPERATION erasetoendofline IS

operation body description (text)

Puts blank characters from current cursor position to the end of current line.
Current cursor position remains unchanged.

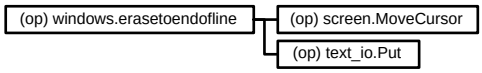
used operations

```
screen.MoveCursor
text_io.Put
```

operation code (ada)

```
begin
  Screen.MoveCursor (me.Current);
    for Count in me.Current.Column .. me.Last.Column loop
      Text_IO.Put (' ');
    end loop;
  Screen.MoveCursor (me.Current);
```

call tree from Ada code



END erasetoendofline

OPERATION initialize IS

operation body description (text)

Clears the screen.

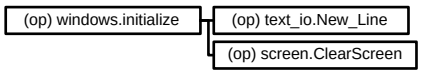
used operations

```
text_io.New_Line
screen.ClearScreen
```

operation code (ada)

```
begin
  Text_IO.New_Line;
  Screen.ClearScreen;
  Text_IO.New_Line;
```

call tree from Ada code



END initialize

END windows

CLASS phil IS

ACTIVE

pragmas

```
PRAGMA discriminant
  (type_name => Philosopher,
   attribute_name => --|My_ID|--)
```

DESCRIPTION

PROBLEM

Statement of the Problem (text)

Phil is a module describing an abstract Philosopher.
Philosophers behave autonomously as soon as they are allowed to start eating.
To eat, they need to grab two chopsticks which are shared with their two neighbours.

Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Analysis of Requirements

Structural Requirements (text)

SR2: type Philosopher provides an abstract description of their individual behaviour. (cf.SR2/Philosophers:)

Functionnal Requirements (text)

FR6: Let Philosophers start eating. (cf.FR6/Start_eating:)

Behavioural Requirements (text)

BR6: While eating, each Philosopher changes sequentially its internal state in following order: Breathing, Got_One_
Got_One_Stick, Got_Other_Stick, Eating, Done_Eating, Thinking and Dying. Change of state is not triggered by ex
external requests, but by release of shared chopsticks and internal waitng delays. (cf.BR6/Philosopher_states:)

Local Environment

Parent General Description (text)

Please refer to parent module description.

SOLUTION

General Strategy (text)

Phil is designed as an active HOOD4 class with a single constrained operation.

Code generator will produce a package containing a task type.

Structural Description

Identification of Data Structures (text)

SR2:
- type Philosopher with a unique attribute (My_ID) which is implemented as a discriminant.
- type Philosopher_Ptr is a pointer to a Philosopher.

Functional Description

Identification of Operations (text)
FR6: entry Start_Eating

PROVIDED_INTERFACE

TYPES

Philosopher

class inheritance (hood)
```
INHERITANCE NONE
```

type attributes (hood)
```
ATTRIBUTES My_ID : society.Unique_DNA_Codes
```

type enumeration (hood)
```
ENUMERATION NONE
```

Philosopher_Ptr

type attributes (hood)
```
ATTRIBUTES NONE
```

type enumeration (hood)
```
ENUMERATION NONE
```

type definition (ada)
```
type Philosopher_Ptr is access all Philosopher;
```

States

type attributes (hood)
```
ATTRIBUTES NONE
```

type enumeration (hood)
```
ENUMERATION NONE
```

type definition (ada)
```
type States is (
  Breathing, Thinking, Eating, Done_Eating,
  Got_One_Stick, Got_Other_Stick, Dying);
```

OPERATIONS

start_eating

operation declaration (hood)
```
start_eating(
  me : in out Philosopher;
  Who_Am_I : in Society.Unique_DNA_Codes;
  Chopstick1 : in Positive;
  Chopstick2 : in Positive
);
```

real time attributes (hood)
WCET

OBJECT_CONTROL_STRUCTURE

constrained operations
start_eating CONSTRAINED_BY LSER;

REQUIRED_INTERFACE

```
OBJECT chop;
  TYPES
    NONE
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    pick_up; put_down;
  EXCEPTIONS
    NONE
OBJECT room;
  TYPES
    NONE
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    report_state; get_stick;
  EXCEPTIONS
    NONE
OBJECT society;
  TYPES
    Unique_DNA_Codes;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
OBJECT standard;
  TYPES
    Positive; Duration;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
```

DATAFLOWS

```
reporting => room;
using => chop;
```

INTERNALS

TYPES

Think_Times

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
subtype Think_Times is Positive range 1..8;
```

Meal_Times

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
subtype Meal_Times is Positive range 1..10;
```

Life_Time

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
subtype Life_Time is Positive range 1 .. 5;
```

DATA

Think_Length

data declaration (ada)

```
package Think_Length is new Random_Generic(
  Result_Subtype => Think_Times);
```

data access from pseudo_code
```
(da) phil.Think_Length IS USED BY NONE
```

data access from Ada code
```
(da) phil.Think_Length IS USED BY NONE
```

data access from C code
```
(da) phil.Think_Length IS USED BY NONE
```

data access from C++ code
```
(da) phil.Think_Length IS USED BY NONE
```

Meal_Length

data declaration (ada)
```
package Meal_Length is new Random_Generic(
  Result_Subtype => Meal_Times);
```

data access from pseudo_code
```
(da) phil.Meal_Length IS USED BY NONE
```

data access from Ada code
```
(da) phil.Meal_Length IS USED BY NONE
```

data access from C code
```
(da) phil.Meal_Length IS USED BY NONE
```

data access from C++ code
```
(da) phil.Meal_Length IS USED BY NONE
```

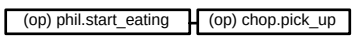
OPERATION_CONTROL_STRUCTURES

OPERATION start_eating IS

used operations
```
room.report_state
room.get_stick
chop.pick_up
chop.put_down
```

operation code (pseudo)
```
chop.pick_up
```

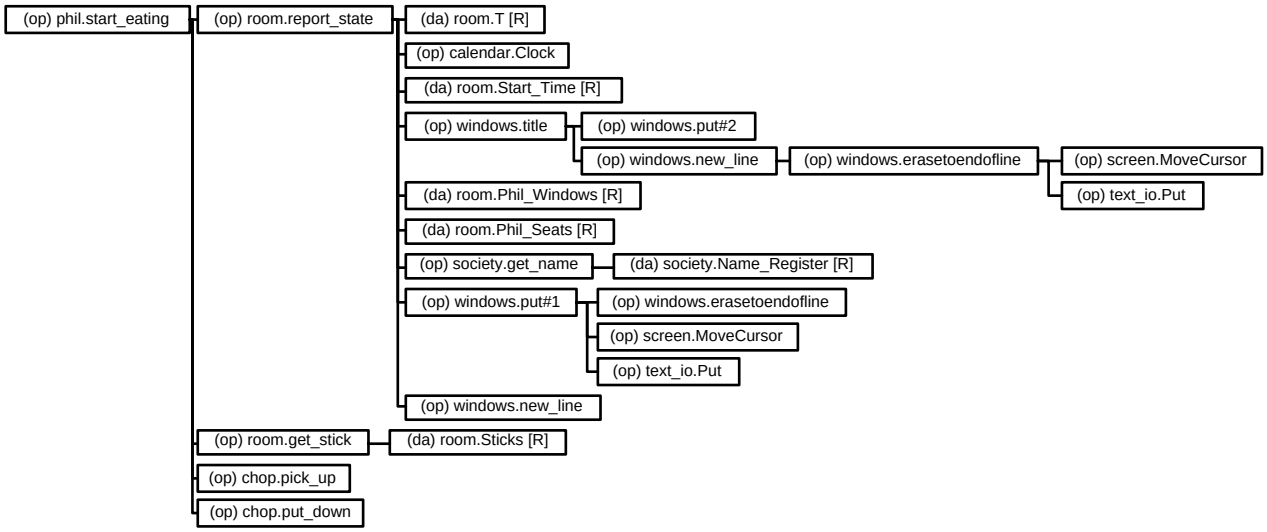
call tree from pseudo_code



operation code (ada)
```
  Meal_Time : Meal_Times;
  Think_Time : Think_Times;

begin

  Room.Report_State(Who_Am_I, Breathing);

  for Meal in Life_Time loop

    Room.Get_Stick(Chopstick1).all.Pick_Up;
    Room.Report_State(Who_Am_I,Got_One_Stick,Chopstick1);

    Room.Get_Stick(Chopstick2).all.Pick_Up;
```

```
      Room.Report_State(Who_Am_I,Got_Other_Stick,Chopstick2);

      Meal_Time := Meal_Length.Random_Value;
      Room.Report_State(Who_Am_I,Eating,Meal_Time,Meal);

      delay Duration(Meal_Time);

      Room.Report_State(Who_Am_I,Done_Eating);

      Room.Get_Stick(Chopstick1).all.Put_Down;
      Room.Get_Stick(Chopstick2).all.Put_Down;

      Think_Time := Think_Length.Random_Value;
      Room.Report_State(Who_Am_I,Thinking,Think_Time);

      delay Duration(Think_Time);

    end loop;

    Room.Report_State(Who_Am_I,Dying);
```

call tree from Ada code



END start_eating

END phil

OBJECT society IS

PASSIVE

DESCRIPTION

PROBLEM

Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Local Environment

Parent General Description (text)

Please refer to parent module description.

PROVIDED_INTERFACE

TYPES

Unique_DNA_Codes

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
subtype Unique_DNA_Codes is Positive range 1..5;
```

OPERATIONS

get_name

operation declaration (hood)

```
get_name(Code : in Unique_DNA_Codes) return String;
```

real time attributes (hood)

```
WCET
```

REQUIRED_INTERFACE

```
OBJECT standard;
  TYPES
    String; Positive;
  CONSTANTS
    NONE
  OPERATION_SETS
    NONE
  OPERATIONS
    NONE
  EXCEPTIONS
    NONE
```

INTERNALS

DATA

Name_Register

data declaration (ada)

```
Name_Register : array(Unique_DNA_Codes) of String(1..18) :=
  ("Philosopher #1    ",
   "Philosopher #2    ",
   "Philosopher #3    ",
   "Philosopher #4    ",
   "Philosopher #5    ");
```

data access from pseudo_code

```
(da) society.Name_Register IS USED BY NONE
```

data access from Ada code

```
(da) society.Name_Register IS USED BY
  (op) society.get_name [R]
```

data access from C code

```
(da) society.Name_Register IS USED BY NONE
```

data access from C++ code

```
(da) society.Name_Register IS USED BY NONE
```

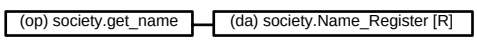
OPERATION_CONTROL_STRUCTURES

OPERATION get_name IS

operation code (ada)

```
begin
  return Name_Register(Code);
```

call tree from Ada code



END get_name

END society

CLASS chop IS

PASSIVE

DESCRIPTION

PROBLEM

Referenced Documents (text)

This application is the HOOD version of "Dining Philosophers - Ada95 edition" from Michael B. Feldman, The Georg
George Washington University, July 1995.
HOOD adaptation was performed by Pierre Dissaux, TNI, June 1998, with STOOD toolset.

Local Environment

Parent General Description (text)

Please refer to parent module description.

PROVIDED_INTERFACE

TYPES

Stick

class inheritance (hood)

```
INHERITANCE NONE
```

type attributes (hood)

```
ATTRIBUTES In_Use : Boolean := --|false|--
```

type enumeration (hood)

```
ENUMERATION NONE
```

Stick_Ptr

type attributes (hood)

```
ATTRIBUTES NONE
```

type enumeration (hood)

```
ENUMERATION NONE
```

type definition (ada)

```
type Stick_Ptr is access all Stick;
```

OPERATIONS

pick_up

operation declaration (hood)

```
pick_up(me : in out stick);
```

real time attributes (hood)
WCET

put_down

operation declaration (hood)
```
put_down(me : in out stick);
```

real time attributes (hood)
WCET

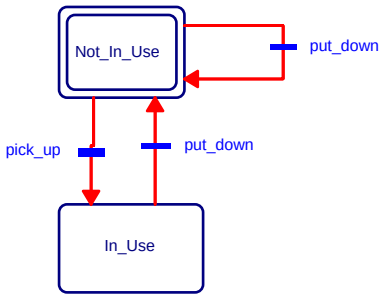
OBJECT_CONTROL_STRUCTURE

constrained operations
```
pick_up CONSTRAINED_BY RWER STATE;
put_down CONSTRAINED_BY RWER STATE;
```

INTERNALS

OBJECT_CONTROL_STRUCTURE

state transition diagram



Not_In_Use

entering transitions
put_down, put_down

exiting transitions
pick_up, put_down

In_Use

entering transitions
pick_up

exiting transitions
put_down

pick_up

transition event
`pick_up`

transition from
`Not_In_Use`

transition to
`In_Use`

trans condition (ada)
```
not In_Use
```

put_down

transition event
`put_down`

transition from
`In_Use`

transition to
`Not_In_Use`

put_down

transition event
`put_down`

transition from
`In_Use`

transition to
`Not_In_Use`

OPERATION_CONTROL_STRUCTURES

OPERATION pick_up IS

operation code (ada)
```
begin
  In_Use := True;
```

END pick_up

OPERATION put_down IS

operation code (ada)
```
begin
  In_Use := False;
```

END put_down

END chop