
Stood 4.1

User's Manual

part II: Architectural Design

1. Modules and Components.....p. 3
2. HOOD Diagrams..... p. 7
3. Graphical edition.....p. 37
4. State-Transition Diagrams..... p.135
5. Inheritance Tree..... p.147



Pierre Dissaux



1. Modules and Components

A HOOD **Application** is described as set of cooperating sub-systems. From the most general point of view, these sub-systems will be called **Modules**.

These **Modules** build a tree structure describing various levels of abstraction for the same **Application**. At the highest level, the overall **Application** may be handled by a unique **Module**, called the **Root Module**. At the lowest level, leaves of the tree are called **Terminal Modules** and will contain all the information regarding a very limited part of the **Application**. At intermediate levels, branches of the tree, called **Non Terminal Modules**, help in organizing and partitioning the **Application**.

Each **Module**, whatever its position in the hierarchy is, may encompass software **Components** of the **Application**. These **Components** may be of following kinds:

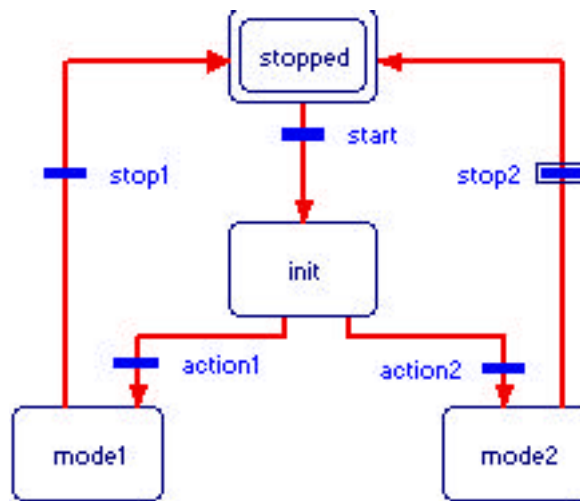
- Functional elements: **Operations, Exceptions**
- Data description elements: **Types, Constants, Data**
- Behavioural elements: **States, Transitions**

Architectural design with **STOOD** aims to build a complete **HOOD** hierarchy of **Modules**, complying with methodological recommendations, and to identify all necessary **Components**. This strong software structure will then be ready to be filled up during detailed design phase.

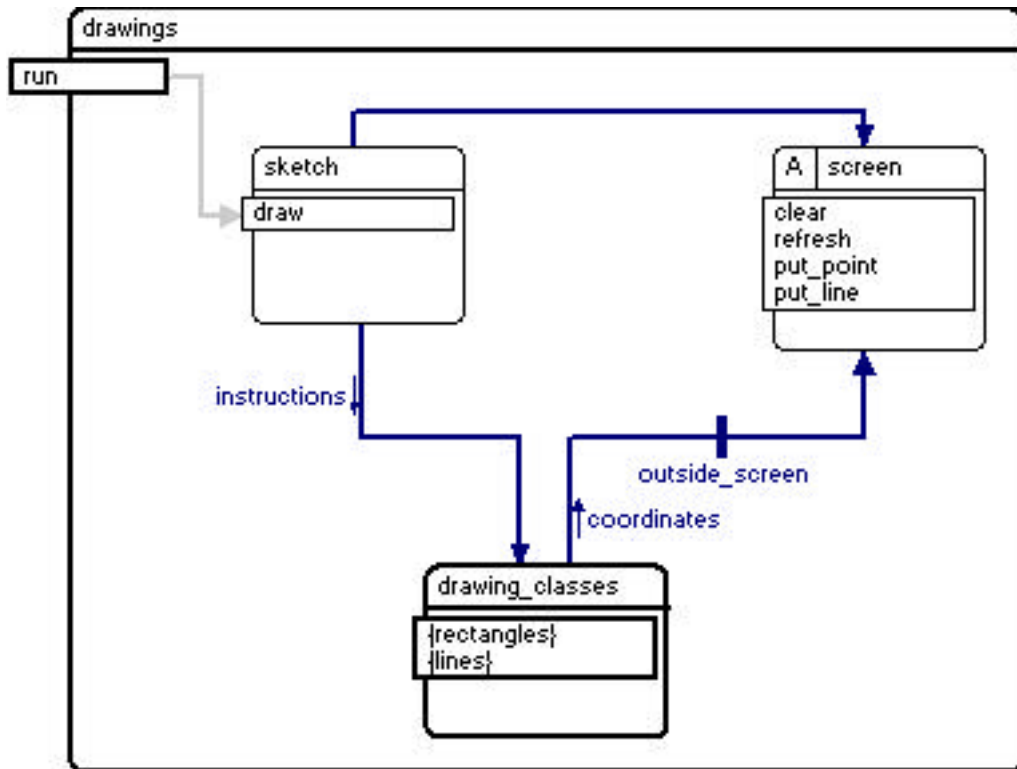
This process may of course follow an iterative way, and **STOOD** will automatically keep internal model consistency at any time.

Practically, results of architectural design phase with **STOOD** is stored in one unique file called `Stood.dg`, and located in a directory named as the **Application**. Take care to save your work from time to time, in order to keep this file as up to date as possible. If backup process on your system does not allow to save the overall **Application** directory, the one file to save to be able to recover an architectural design is `Stood.dg`.

To perform architectural design tasks with **STOOD**, graphical edition is required, and both multi-views **HOOD** diagrams and **State-Transition Diagrams** need to be drawn:



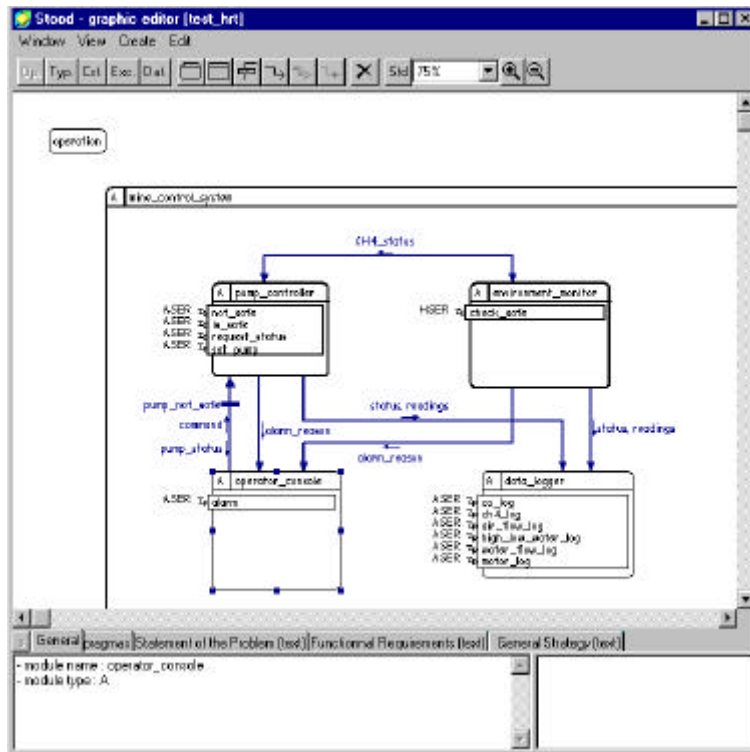
a State-Transition Diagram



a HOOD diagram



2. HOOD Diagrams editor



2.1 Drawing area.....	p. 9
2.2 Text input area	p.14
2.3 Window menu.....	p.18
2.4 View menu.....	p.20
2.5 Create menu.....	p.21
2.6 Edit menu.....	p.30

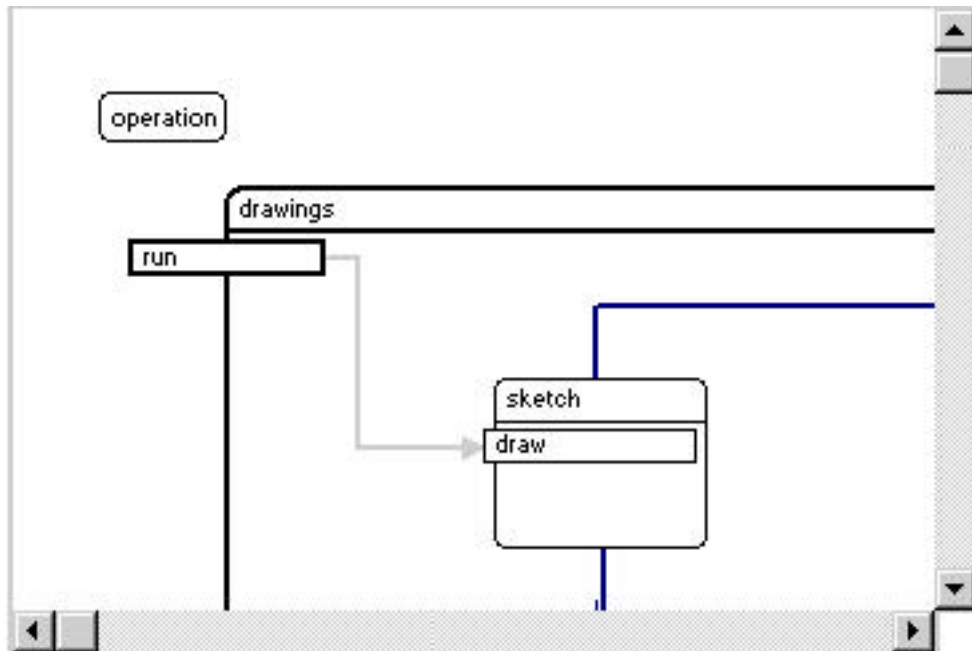
A **HOOD** diagram editor may be opened by one of the following actions:

- Select *graphic editor* item of *editors* menu in *main editor*
- Press *gra* button on button bar of *main editor*
- Select *duplicate* item of *window* menu in another *graphic editor*

Before opening a **HOOD** diagram editor, a **Project** and an **Application** would better have been selected. In this case, displayed diagram will show current **Root Module**. If no **Application** was selected before opening graphical editor, or if current **Application** is deselected, no diagram will be displayed and no graphical edition will be possible until another **Application** is selected in *main editor*.

A *graphic editor* is composed of a large drawing area where graphical edition may be performed, a small text input area where additional information about graphical entities may be entered, a menu bar and a button bar.

2.1. Drawing area



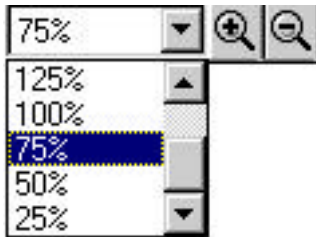
Drawing area shows only one **HOOD** diagram at a time, describing a particular level of abstraction in **HOOD** hierarchy. Inside drawing area, it is possible to select, create, move, resize and delete graphical elements. General principles of graphical edition are explained below, but more detailed information regarding practical edition of each **HOOD** entity is provided in § 3.

Drawing area may be switched on one of the five provided views to show **Operations**, **Types**, **Constants**, **Exceptions** or **Data**. Current view is mentioned in a label at top left side of this window area.

2.1.1. Diagram scale

It is generally not possible to display a full **HOOD** diagram without making graphical entities too small to be readable. Scale of drawing area may be controlled by dedicated buttons and list at the right hand of graphic editor button bar.

Proposed scaling rates are from 25% up to 400%. They may be obtained by clicking on zoom-in (+) or zoom-out (-) buttons, or directly selecting chosen rate on the scrolling list.



2.1.2. Select, move and resize graphical entities

Within drawing area, graphical entities may be selected by a simple click on left mouse button. Dragging handles are displayed on current selected component. There is no possible multiple selection for graphical entities.

While selected, some entities may be moved or resized. To move an entity, select it and drag it without releasing left mouse button. To resize it, drag one of the dragging handles. Some entities have fixed location or size. There is no "undo" function for move and resize operations. Take care to save your diagrams from time to time, and before major changes.






If new location of a move is not valid, a dialog box will be displayed:

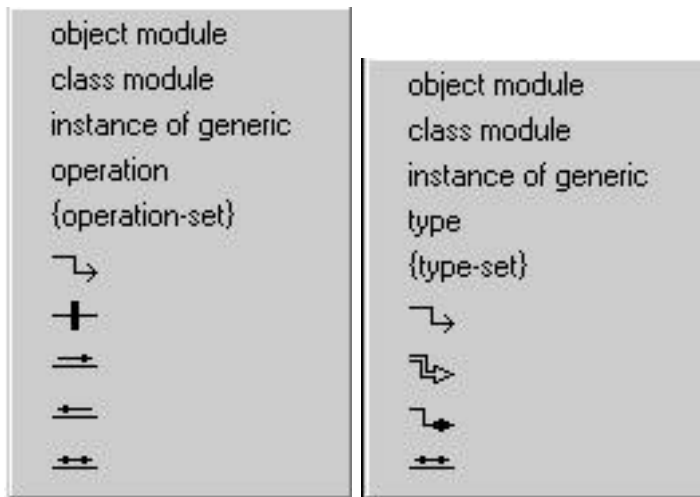






Some move may have other effects than only affecting the display. For instance, moving a **Module** box inside another one will change **Design Tree** structure. Please refer to detailed description for each kind of entity.

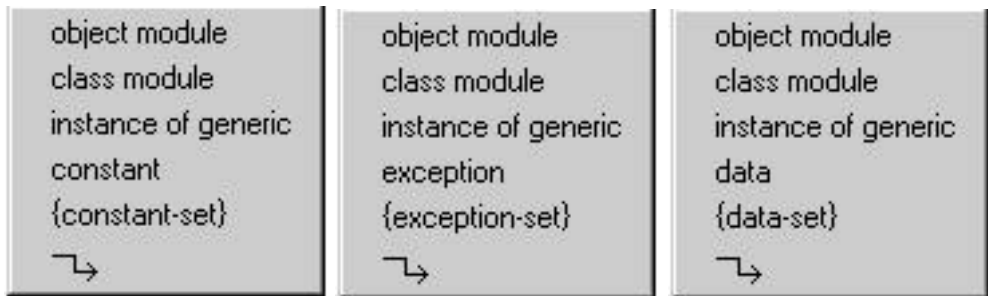
2.1.3. Create new graphical entities


An easy way to create new graphical entities, is to use drawing area pop-up menu. This menu appears when clicking on center mouse button on **UNIX** platforms, or right mouse button on **Windows** platforms, and when mouse pointer is located inside drawing area.

When current view is *operation*, drawing area pop-up menu contains items to create **Object**, **Class** or **Instance_Of Modules**, **Operations** or **Operation_Sets**, **Op_Use** and **Implemented_By** relationships: , **Exception Flows**: , **direct**: , **reverse**:  or **bidirectional**: 
DataFlows. Please refer to *create* menu description for further details (§ 2.5).

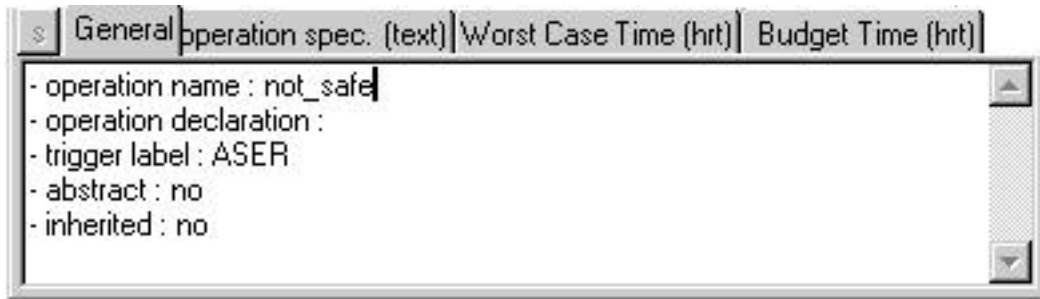


When current view is *type*, drawing area pop-up menu contains items to create **Object**, **Class** or **Instance_Of Modules**, **Types** or **Type_Sets**, **Type_Use** and **Implemented_By** relationships: , **Inheritance** relationships: , **Attributes** relationships:  or labels: . Please refer to *create* menu description for further details (§ 2.5).



When current view is respectively *constant*, *exception* or *data*, drawing area pop-up menu contains items to create **Object**, **Class** or **Instance_Of Modules**, **Constants** (resp. **Exception** or **Data**) or **Constant_Sets** (resp. **Exception_Sets** or **Data_Sets**), or **Implemented_By** relationships: . Please refer to *create* menu description for further details (§ 2.5).

2.2. Text input area



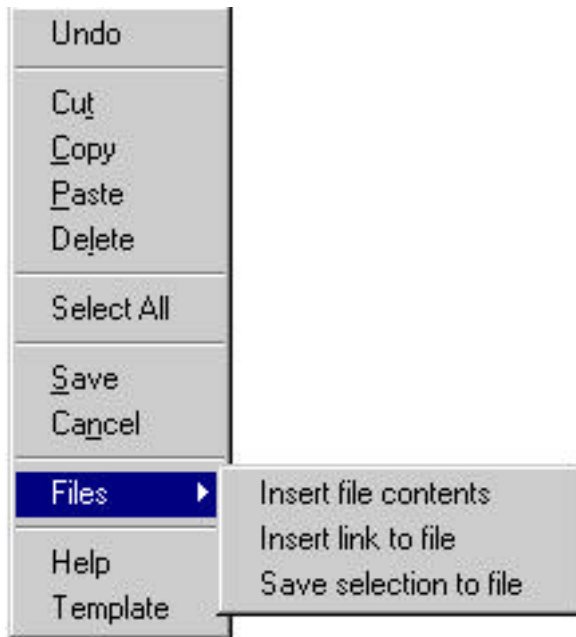
As soon as a graphical entity has been selected inside drawing area, additional information must generally be inserted within text input area. This information may be simply a name, or a set of properties, regarding the kind of selected entity.

This text input area may contain one or several tabs. *General* tab is always present, and other tabs may be added or removed with a tool configuration action (refer to part I of this user's manual). *General* tab contains zero, one or several fields composed of a fixed generic field name, and relevant variable value for current selected entity. Take care not to alter field names while typing field values, else a syntactic error will occur. Would this happen, please cancel changes before going on (select *cancel* item in text input area pop-up menu).

Please refer to § 3 to get a description of the fields to be entered in *General* tab, for each kind of graphical entity :

- For a **Module**: refer to 3.1.3
- For an **Operation**: refer to 3.3.3
- For an **Operation_Set**: refer to 3.4.3
- For an **Exception**: refer to 3.5.3
- For a **DataFlow**: refer to 3.6.6.3
- For an **Exception_Flow**: refer to 3.6.7.3
- For a **Type**: refer to 3.7.3
- For a **Type_Use** label: refer to 3.8.7.3
- For a **Constant** or **Data**: refer to 3.9.3

While pressing center or right mouse button, while mouse pointer is located inside text input area, a pop-up menu shows general purpose text editing functions:



Undo: Cancel previous text change or *cut*, *paste* or *delete* command.

Cut: Copy to a text buffer and erase currently selected text.


Copy: Copy currently selected text to a text buffer.

Paste: Paste text buffer contents at insertion point.

Delete: Erase currently selected text.

Select All: Select all contents of text input area.

Save: Save text changes.

Note that, in text input area, pushing  button acts as *save* command.

Cancel : Restore previously saved version of the text.

Insert file contents : Open a standard dialog box to select a file which contents will be pasted at current insertion point.

Insert link to file : Open a standard dialog box to select a file which location will be used as a include link for some documentation tools.

Save selection to file : Open a standard dialog box to select a file in which current text will be copied.

Help : Provide contextual help regarding current textual edition. Please refer to detailed chapters related to each kind of graphical entity. These help files may be customized by editing the files contained in `config/ods_help` configuration directory (refer to part I of this User's Manual).

Template : Provide a template current textual edition. Please refer to detailed chapters related to each kind of graphical entity. These template files may be customized by editing the files contained in `config/ods_template` configuration directory (refer to part I of this User's Manual).

2.3. Window menu of graphic editor

Window menu from main menu bar provides general functions to control *graphic editor*.



Help: provide general information about *graphic editor*. This information is displayed in a dialog box.



This help information may be customized by changing contents of `gra` and `gra.more` files in `config/help` configuration directory.

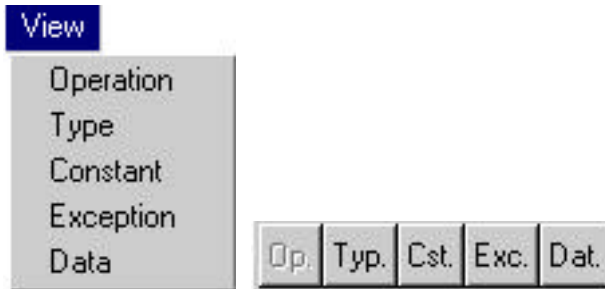
Duplicate : open another *graphic editor*. Practically, it is simply a shortcut to *graphic editor* item of *editors* menu of *main editor*.

Print : direct print of current **HOOD** diagram on standard printer. For **Windows**, used printer is the default one. For **UNIX**, a **PostScript** file is created in working directory, and direct printing is performed if a printer queue has first been defined in `fastprint.sh` file of `internalTools` configuration directory.

Quit : close current graphic editor, but has no effect on recent graphical editing actions that remain valid.

2.4. View menu of graphic editor

HOOD4 diagrams should be described with two separated but consistent views: a structural view that shows data structures (**Types**) and their dependencies, and a functional view that shows services (**Operations**) and their relationships. In order to ease creation and handling of other kinds of graphical components, **STOOD** provides three additional views to manage **Constants**, **Exception** and **Data**. Note that all these views refer to the same set of **Modules**.



Only one view of one **HOOD** diagram may be shown at a time on a *graphic editor*. Current displayed view may easily be identified by a dedicated label at top left side of drawing area. *view* menu from *graphic editor* main menu bar, or predefined switches from button bar should be used to change current view.


menu item	Operation	Type	Constant	Exception	Data
button	Op.	Typ.	Cst.	Exc.	Dat.
label	operation	type	constant	exception	data


2.5. Create menu of graphic editor

create menu of main menu bar of *graphic editor* should be used to create graphical entities. Items list of this menu varies as regards current view (refer to § 2.4 to change view if required). Some menu items are the same for each view, whereas other items are specific to current view.

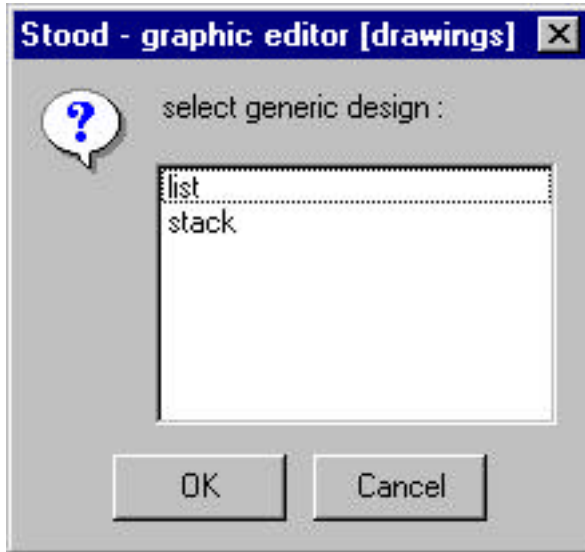
2.5.1. create menu items that are common to all views

Creation and setting of **HOOD Modules** does not depend on current view. That is why relevant menu items are always proposed by *create* menu. Shortcuts may also be available with buttons.

object module or  : Create a new **HOOD Object** (refer to § 3.2.1). A grey rectangle is shown at mouse pointer location, and may be dragged to chosen place. A mouse button click will then actually create a new **Object**. It will be given a default name. Refer to § 3.1.3 to learn how to change properties of a **HOOD Module**.

class module or  : Create a new **HOOD Class** (refer to § 3.2.2). It behaves exactly the same as for **HOOD Objects**.

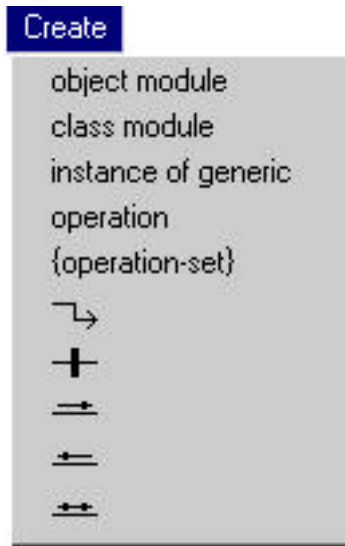
instance of generic : Create a new **HOOD Instance_Of** (refer to § 3.2.4). When this menu item is selected, a dialog box provides the list of visible **Generic Applications** which could be taken as a template for new **Module**.




After having chosen one of proposed **Generics**, create process becomes similar to creating regular **Objects** or **Classes**.



For a given **Project**, the list of proposed **Generic Applications** may be controlled at first level with *system editor*, and more generally by setting a new path (SavePath property in *stood.ini* or *.stoodrc* initialization file) to search **STOOD Applications** on your system or network. Please refer to part I of this User's Manual for more detailed information.


2.5.2. create menu items for Operation view





operation or  : When choosing this menu item or pressing appropriate shortcut button, a new **Operation** is created. Please refer to § 3.3.2 for further details.


{operation-set} : When choosing this menu item, a new empty **Operation_Set** is created. Please refer to § 3.4.2 for further details. As an extension to **HOOD4**, **Internal Operation_Sets** may also be created with **STOOD**.

 or  : When choosing this menu item or pressing appropriate shortcut button, a new **Op_Use** or **Implemented_By** relationship is created. Please refer respectively to § 3.6.2 or 3.10.2 for further details about creating **Op_Use** or **Implemented_By** relationships for **Operations** and **Operation_Sets**.

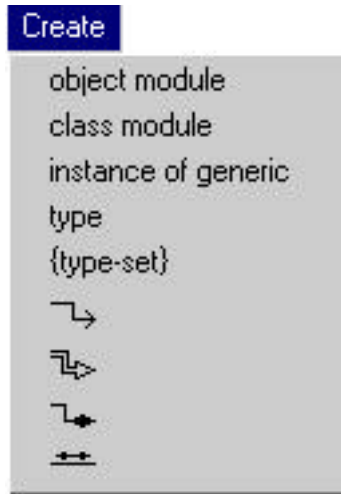
 : When choosing this menu item, a new **Exception_Flow** is created on an existing **Op_Use** relationship. Please refer to § 3.6.6.2 for further details.


 : When choosing this menu item, a new direct **DataFlow** is created on an existing **Op_Use** relationship. Please refer to § 3.6.5.2 for further details.

 : When choosing this menu item, a new reverse **DataFlow** is created on an existing **Op_Use** relationship. Please refer to § 3.6.5.2 for further details.



 : When choosing this menu item, a new bidirectional **DataFlow** is created on an existing **Op_Use** relationship. Please refer to § 3.6.5.2 for further details.



2.5.3. create menu items for Type view






`type` or  : When choosing this menu item or pressing appropriate shortcut button, a new **Type** is created. Please refer to § 3.7.2 for further details.

`{type-set}` : When choosing this menu item, a new empty **Type_Set** is created. Note that **Type_Sets** are **STOOD** extensions to **HOOD4**. Please refer to § 3.4.2 for further details.

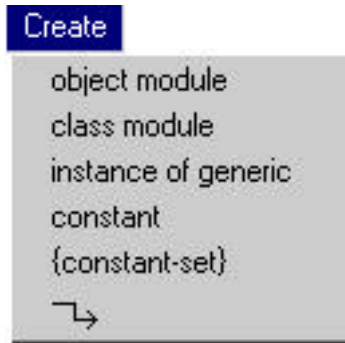
 or  : When choosing this menu item or pressing appropriate shortcut button, a new **Type_Use** or **Implemented_By** relationship is created. Please refer respectively to § 3.8.2 or 3.10.2 for further details about creating **Type_Use** or **Implemented_By** for **Types**.


 or  : When choosing this menu item or pressing appropriate shortcut button, a new **Inheritance** relationship is created. Please refer to § 3.8.4 for further details.

 or  : When choosing this menu item or pressing appropriate shortcut button, a new **Attributes** relationship is created. Please refer to § 3.8.3 for further details.



 : When choosing this menu item, a new label is created on an existing **Type_Use**, **Inheritance** or **Attributes** relationship. Please refer to § 3.8.7 for further details.

2.5.4. create menu items for Constant view

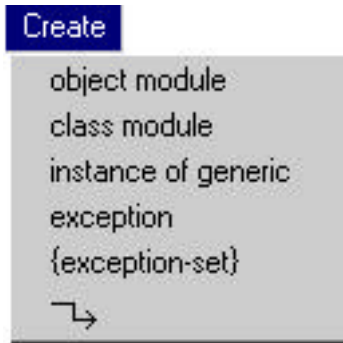



`constant` or  : When choosing this menu item or pressing appropriate shortcut button, a new **Constant** is created. Please refer to § 3.9.2 for further details.

`{constant-set}` : When choosing this menu, a new empty **Constant_Set** is created. Note that **Constant_Sets** are **STOOD** extensions to **HOOD4**. Please refer to § 3.4.2 for further details.



 or  : When choosing this menu item or pressing appropriate shortcut button, a new **Implemented_By** relationship is created. Please refer to § 3.10.2 for further details about creating **Implemented_By** relationships for **Constants**. Note that "use" relationship for **Constants** have no meaning for **HOOD** (even if **STOOD** enables their creation).

2.5.5. create menu items for Exception view

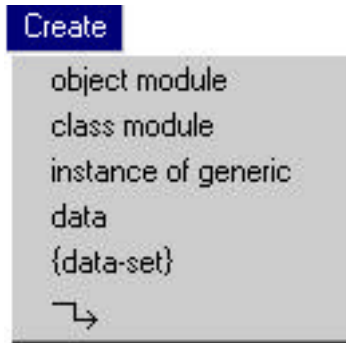



exception or  : When choosing this menu item or pressing appropriate shortcut button, a new **Exception** is created. As an extension to **HOOD4**, **Internal Exceptions** may also be created with **STOOD**. Please refer to § 3.5.2 for further details.

{exception-set} : When choosing this menu, a new empty **Exception_Set** is created. Note that **Exception_Sets** are **STOOD** extensions to **HOOD4**. Please refer to § 3.4.2 for further details.

 or  : when choosing this menu item or pressing appropriate shortcut button, a new **Implemented_By** relationship is created. Please refer to § 3.10.2 for further details about creating **Implemented_By** relationships for **Exceptions**. Note that "use" relationship for **Exceptions** have no meaning for **HOOD** (even if **STOOD** enables their creation).



2.5.6. create menu items for Data view



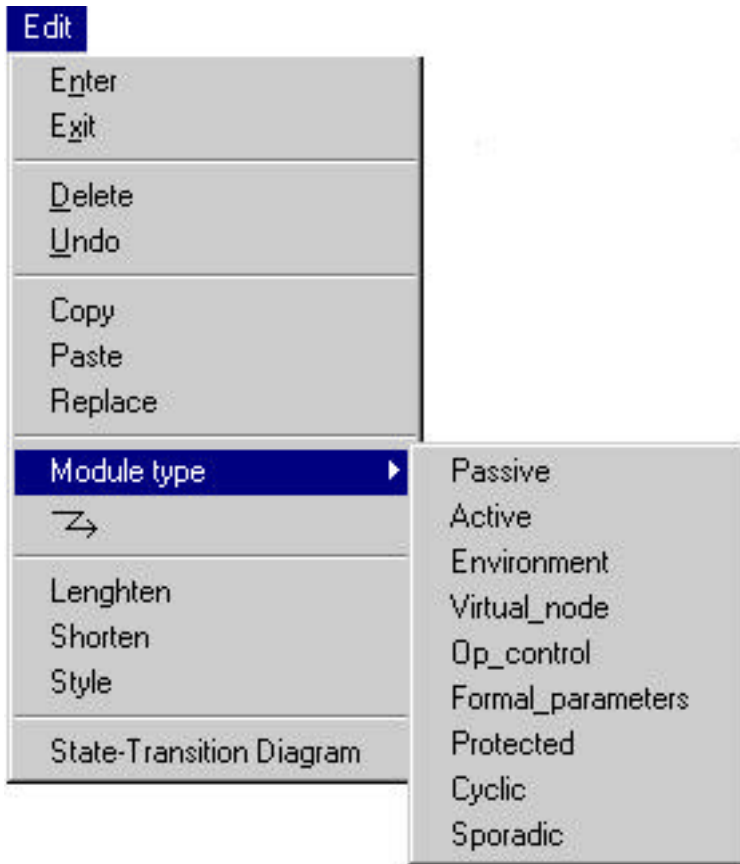
data or  : When choosing this menu item or pressing appropriate shortcut button, a new **Data** element is created. Please refer to § 3.9.2 for further details. With standard **HOOD** configuration, **Provided Data** are prohibited. Attempt to create **Provided Data** will raise a warning message:



{data-set} : When choosing this menu, a new empty **Data_Set** is created. Note that **Data_Sets** are **STOOD** extensions to **HOOD4**. Please refer to § 3.4.2 for further details.


 or  : neither **Implemented_By** nor "use" relationships should never be used on *data* view with standard **HOOD** configuration.

2.6. Edit menu of graphic editor



When graphical entities has been created within drawing area, *edit* menu provides a set of additional editing functions related to these **Modules**, **Components**, or relationships. If chosen menu item is not appropriate regarding currently selected graphical entity, a warning message is displayed in a dialog box.

2.6.1. edit menu items related to all entities

Delete or  : remove selected entity from the diagram, but also deletes all relevant information from **STOOD** internal storage. Last deleted entity may be recovered with *undo* menu command.

Undo : recover last deleted entity. This command does not undo the other editing actions.

2.6.2. edit menu items related to Modules


Enter : go down one step in **HOOD** hierarchy. Selected child **Module** of current diagram becomes parent **Module** of a new lower level diagram. Same effect may be obtained in double-clicking inside the child **Module** box. As an extension to **HOOD4**, **STOOD** also displays diagrams for **Terminal Modules**.


Exit : go up one step in **HOOD** hierarchy. Parent **Module** of current diagram becomes a child **Module** of a new higher level diagram. Same effect may be obtained in double-clicking outside the parent **Module** box. As an extension to **HOOD4**, **STOOD** also displays a pseudo **HOOD** diagram for the **System Configuration**. So, if current parent **Module** is the **Root Module**, then it will become a pseudo child of a pseudo parent representing current **System Configuration**.

Copy : open a dialog box to copy one of the **Modules** of current **Application**. A **Module** (and its sub-hierarchy) may be copied into a local clipboard, or to build a new **Root Module**. Please refer to part I for further details.

Paste : create a new **Module** (and its sub-hierarchy) inside current **Application** from local clipboard contents.

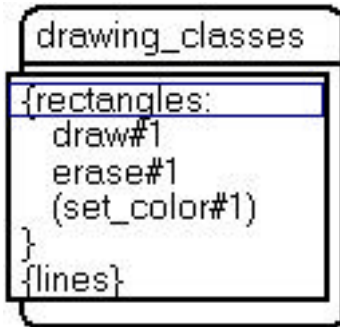
Replace : open a dialog box to replace one of the existing **Modules** of current **Application**. A **Module** (and its sub-hierarchy) may be replaced by local clipboard contents, or by another **Root Module**. Please refer to part I for further details.

Module type or  : this menu item provides a direct way to change the kind of currently selected **Module**. Please refer to § 3.2 for further details about allowed kinds of **Modules**. The kind of a **Module** may also be controlled from text input area (refer to § 3.1.3).

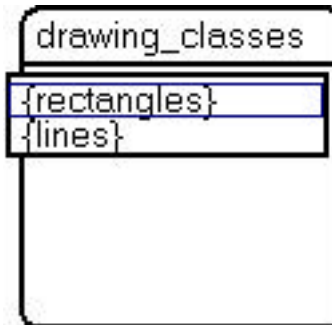
State-Transition Diagram or  : open a **State Transition Diagram** for currently selected **Module**. Please refer to § 4 for further details.

2.6.3. edit menu items related to Components

Enter : when a closed **Set** is selected, this menu command opens it. Same effect may be obtained by double-clicking on closed **Set** name.




an open Set



a closed Set

Exit : when an open **Set** is selected, this menu command closes it. Same effect may be obtained by double-clicking on open **Set** name.

 : when a **Constrained Operation** is selected, this menu command clears relevant trigger labels. When a **Not Constrained Operation** is selected, it only adds a trigger arrow, but actual trigger labels need to be entered within text input area (refer to § 3.3.3.3).

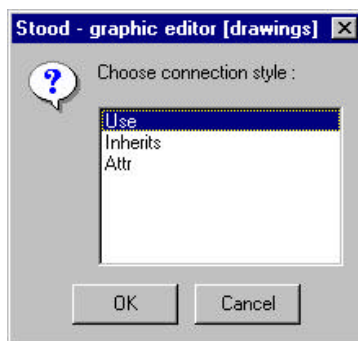
2.6.4. edit menu items related to relationships

Lenghten : this menu command adds a loop to increase the number of handles of selected relationship. One of the existing handles should be selected first to define the location of the loop to be added.



Shorten : this menu command has the opposite effect to previous one. If possible, all useless loops will be flattened. This action may have be performed several times to remove all existing loops.

Style : when current view is *type*, this menu command opens a dialog box that may be used to change selected relationship into **Type_Use**, **Inheritance** or **Attributes**.



Same effect may be obtained by double clicking on a relationship.



3. Graphical edition of HOOD entities

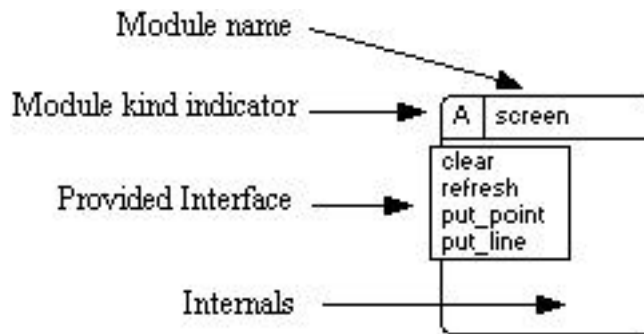
This chapter provides detailed information about editing actions for each kind of graphical entity:

3.1 HOOD Modules.....	p 39
3.2 Kinds of HOOD Modules.....	p 49
3.2.1 HOOD Object	
3.2.2 HOOD Class	
3.2.3 Generic Module and Formal Parameters	
3.2.4 Instance_Of Generic Module	
3.2.5 Op_Control Module	
3.2.6 Environment Module	
3.2.7 Summary of Module kinds	
3.3 Operations.....	p 71
3.4 Operation_Sets, and other Sets.....	p 83
3.5 Exceptions.....	p 89
3.6 Use, DataFlows, Exception_Flows.	p 95
3.7 Types.....	p 105
3.8 Type_Use, Attributes, Inheritance..	p 113
3.9 Constant and Data.....	p 123
3.10 Include and Implemented_By.....	p 129



3.1. HOOD Modules

As explained in § 1, **HOOD Modules** are basic structuring entities of **HOOD Applications**. Creating, deleting or moving **Modules** will change the architectural properties of the **Application**. **Modules** are represented by boxes in **HOOD** diagrams.

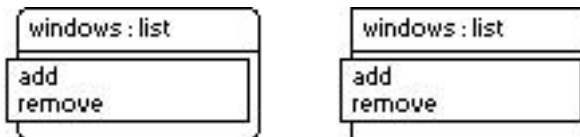


Graphical formalism may differ as regards the various kinds of **Modules**, as described in § 3.2:

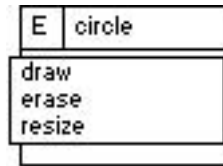
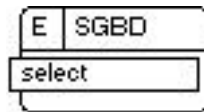
A plain **Object** or a **Class**:



An **Instance_Of** generic **Object** or generic **Class**:



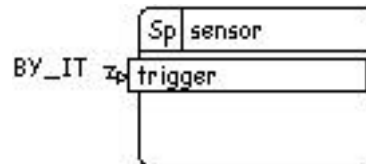
An **Environment Object** or **Class**:



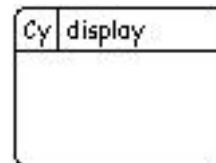
An **Op_Control Object**:



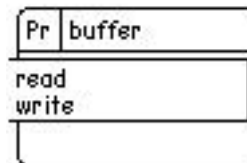
A **Sporadic Object (HRT-HOOD)**:



A **Cyclic Object (HRT-HOOD)**:

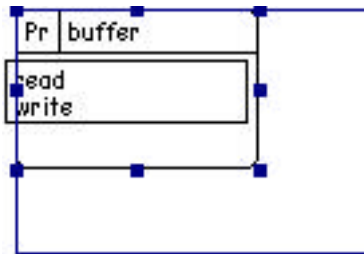


A **Protected Object (HRT-HOOD)**:



3.1.1. Select and resize an existing Module



A single mouse click within the box of a **Module** will select it. When a **Module** is selected, dragging handles are shown on the borders of its box. These handles may be used to resize the box:



Note that **Provided Interface** box cannot be resized.

3.1.2. Create a new Module

As described in § 2, following commands may be used to create a new **Module**:

- **object module** item of *create* menu or drawing area pop-up menu.
- **class module** item of *create* menu or drawing area pop-up menu.
- **instance of generic** item of *create* menu or drawing area pop-up menu.
-  or  buttons.
- **Paste** item of *edit* menu (to create a local copy of clipboard contents).

After having performed one of these actions, a grey box shape appears on display area. It should be located at wished place with mouse pointer and fixed with a single mouse click.

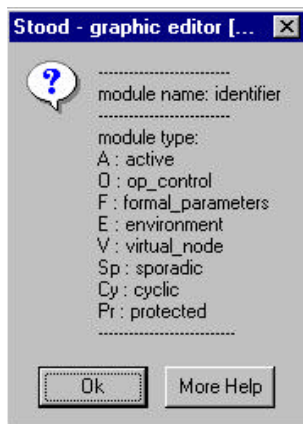
Newly created **Module** is given a default name based on parent **Module** name, with a variable suffix to comply with name unicity rule. If **Paste** command was used, default name is `clipboard`.

3.1.3. Edit Module name and kind

When a **Module** is selected, its properties can be edited within General tab of text input area:



Help item of text input area pop-up menu provides on-line information about these two fields:



Note that contents of these help box may be customized, by editing `gra_txt_obj` and `gra_txt_obj.more` files in `config/help` configuration directory (refer to part I).

3.1.3.1. *module name*

While naming a **Module** in *module name* field, please take care not to use forbidden characters nor a too long string. Restrictions may come from:

- Compliancy with **HOOD** identifiers naming rules. These rules are checked when accepting text input, and a dialog box will be displayed in case of lexical or syntactic error.
- Target language (**Ada**, **C**, **C++**) naming rules. Except abnormal use of reserved words, uncompliancies to these rules will not be checked by **STOOD**, and will lead to compilation errors.
- Current file system naming rules. **Module** names are used to create files in **Application** storage area. Uncompliancies to these rules could lead to information loss at detailed design phase. In case of **Applications** shared between **UNIX** and **Windows** platforms, **Operation** names should fit both constraints.

3.1.3.2. *module kind*

module type field may be used to change current kind of selected **Module**. Allowed changes depend on current context. Please refer to § 3.2 for further details about possible kinds of **Modules**.

At this level, a single letter should be typed to set **Module** kind. Another way to perform the same action is to use **Module type** item of *edit* menu. In some cases, this letter appear in the top left corner of **Module** box:

-
- essential ones:

A for: **Active**
E for: **Environment**
O for: **Op_Control**

- to be used in very particular cases:

F for: **Formal_Parameters**
V for: **Virtual_Node**

- **HRT-HOOD Objects** types:

Sp for: **Sporadic**
Cy for: **Cyclic**
Pr for: **Protected**

Note that new **Modules** types may be added by tool configuration. Refer to part I of this User's Manual for further details.

3.1.4. Move a Module

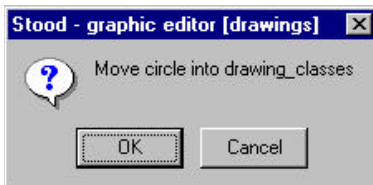
When a **Module** is selected, it can be moved by dragging mouse pointer while keeping left mouse button pressed. Special events may happen when mouse button is released while mouse pointer reaches some particular areas (drag and drop):

- if destination is not a valid location for a **Module**, following dialog box will be displayed:




- if destination is a sibling **Module**, this means that current **Module** will fall down one step in **HOOD** hierarchy.
- if destination is outside parent **Module** box, this means that current **Module** will climb up one step in **HOOD** hierarchy.

These two last actions should be confirmed in a dialog box:



3.1.5. Delete a Module

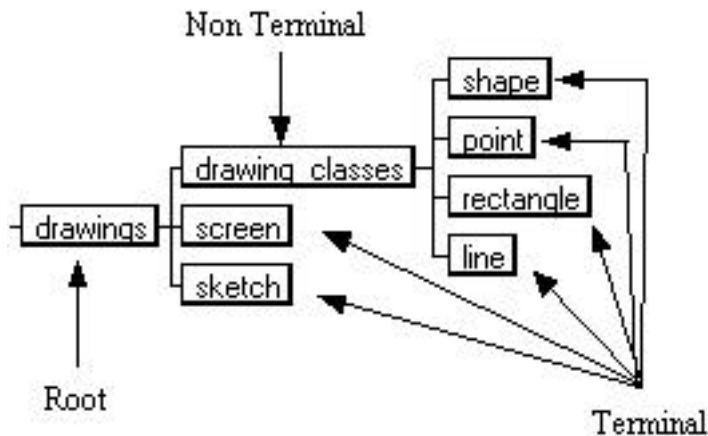
To delete a **Module**, it should be selected first, and **Delete** item of *edit* menu or  button should be used.

There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item. This undelete command can be used only for last delete command.



3.2. Kinds of HOOD Modules

General speaking, a **HOOD Module** is a set of software components describing part of or a whole **Application**. We already saw that a **Module** could be **Root**, **Terminal** or **Non Terminal** in a **Design Tree** (refer to part I).



Modules can also be classified by other properties, and especially depending on the kind of **Components** they actually contain. All combinations between these two classifications are not valid, and **HOOD** rules specify which kind of **Modules** may be **Root**, **Terminal** or **Non Terminal** within an **Application** hierarchy. A summary of valid combinations is provided in § 3.2.7.

Within **STOOD** *graphic editor* drawing area, **Terminal** and **Non Terminal Modules** may be distinguished by the thickness of their border line:

Non Terminal Module:



Terminal Module:

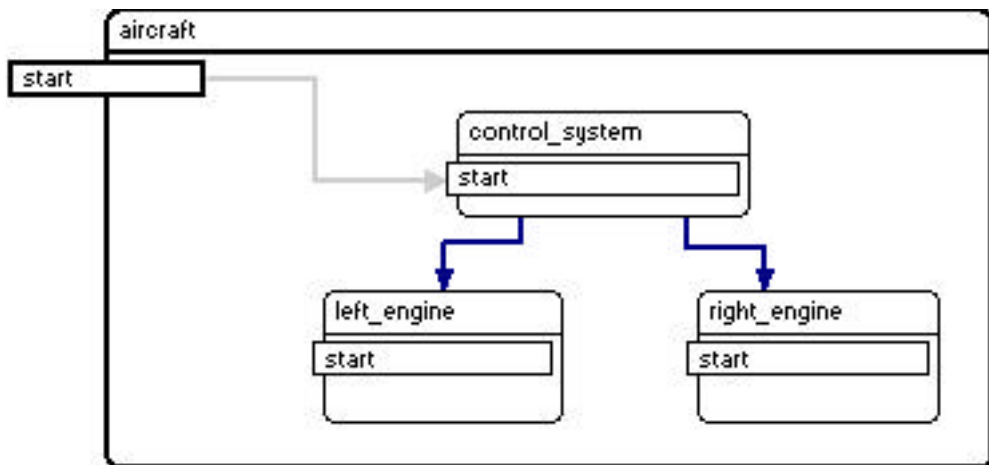


3.2.1. HOOD Object

3.2.1.1. What is a HOOD Object ?

Historically, **HOOD Modules** used to be called “objects”. This ambiguous terminology should be avoided. Anyway, **HOOD Objects** still exist in **HOOD4**, but they are now opposed to **HOOD Classes**. Both **Objects** and **Classes** being **Modules**. Compared to other Object Oriented methods, **HOOD** brings the particularity to mix **Classes** and **Objects** in the same models.

At terminal level, **HOOD Objects** are **Modules** that contain **Data** and **Operations** altering these **Data**. In the context of Object Oriented Modelling, they should be considered as “singletons”, that is the unique instances of non-formalized **Classes**. They are also the right design option to describe static instances of substantial part of the **Application**. For example, in a **HOOD** model of an aircraft, two **HOOD Objects** could be used to describe right and left engines, and highlight their interactions with control system.



3.2.1.2. *Calling Operations of a HOOD Object*

When calling an **Operation** belonging to a **HOOD Object**, identity of the receiver should be specified by using a dotted notation (in **Ada**, or similar name-space specification for other languages). In previous example, control system could start the two engines as follow:

```
left_engine.start;  
right_engine.start;
```

3.2.1.3. *Creating HOOD Objects*

HOOD Objects may only be created in a *graphic editor* (except **Root Objects** that are created from *main editor* (refer to part I). To create a new **Object**, open a **HOOD** diagram at specified level in the hierarchy, and then use *object module* item of *create* menu or drawing area pop-up menu.

3.2.2. HOOD Class

Explicit description of Object Oriented classes is a new feature of **HOOD4**. Even if nothing forbade their identification as Abstract Data Types in **HOOD 3.1**, this was often an argument to deny Object Orientation of **HOOD**. This point was highlighted by an old terminology mistake: classes in **HOOD 3.1** were dealing with **Generic Modules**.

HOOD4 now fully support **OO** classes, inheritance, polymorphism, and mapping to **Ada95** and **C++** or **Java** is quite direct. Anyway, **HOOD4** remains highly compatible with **HOOD 3.1**, and it is possible to build a full **HOOD4** design without any class.

3.2.2.1. HOOD Class Module and Type

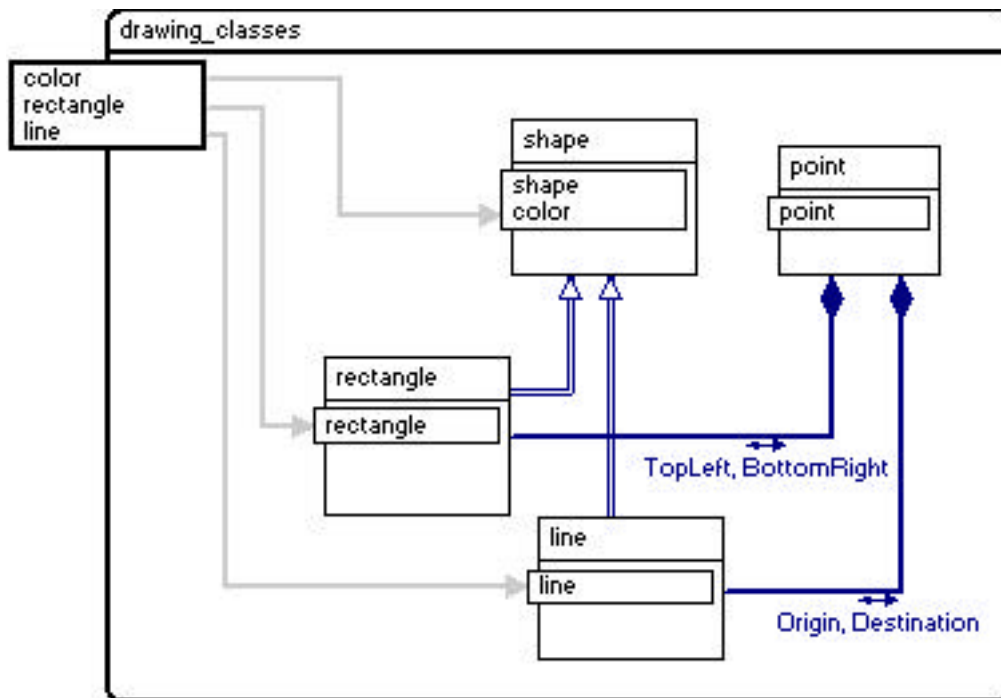
As opposed to **HOOD Objects**, **HOOD Classes** do not contain applicative **Data** (except when modelling “class variables” or “static data members”). They describe a data structure (**Type**) and all **Operations** dealing with this data structure. In **HOOD**, this data structure is called main **Type** of the **Class**. The **Class** may encompass secondary **Types** used locally or directly linked to main one (pointer on main **Type**, ...).

It is interesting to note that, like in **Ada95**, the two main properties of **OO** classes are split in **HOOD**: Encapsulating features are managed by a **Class Module** (containing a consistent set of a **Class Type** and its **Operations**), and instantiating features are supported by the **Class Type** itself. Of course, when producing **C++** code, both merge into a unique class.

A given **Class Module** should not define several **Class Types**. In the case where such a situation appears, **STOOD** will map the **Class Module** to first **Class Type** found (that is the first element of **Provided Types** list).

3.2.2.2. HOOD Class and Class library

HOOD Classes are thus minimal sets of highly consistent software components. This is why they appear as being an optimal choice for **Terminal Modules** of **HOOD** hierarchy. A strict compliance to **HOOD** rules for **Application** breakdown (identifying sub-**Modules** to get the highest consistency and lowest coupling), should lead to define **HOOD Classes** ! It is mandatory for a **HOOD Class** to be a **Terminal Module**.



If while designing, it seems necessary to create a **HOOD Module** providing several **Class Types**, this **Module** should be **Non Terminal** and will require to be broken down into elementary **Class Modules** providing each of them a unique **Class Type**.

Such a design structure is a **Class** library and may spread in depth over several levels of **HOOD** hierarchy. This solution provides a way to offer different formalized interfaces to a set of **Classes**, where implementation entities may easily be hidden to users. For instance, **Abstract Classes**, that cannot be instantiated, may be hidden at higher level of **Class** library if subclassing needs to be forbidden.

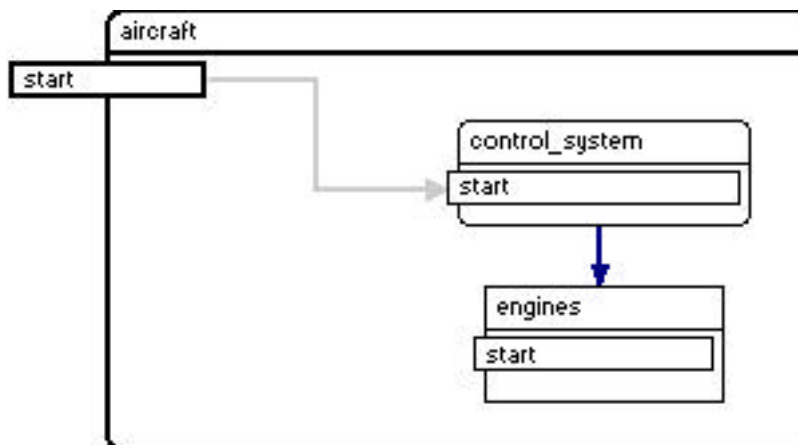
3.2.2.3. *Operation receivers*

One of the main differences between a **HOOD Object** and a **HOOD Class** is that the latter need first to be instantiated to be “used” (that is for its **Operations** to be called). When calling an **Operation** belonging to a **HOOD Class**, identity of the actual receiver (one of the current instances of the **Class**), should be specified among **Parameter** list. This particular **Parameter** (**Operation** receiver), is called **me** in **HOOD4**.

When a **HOOD Module** is a **Class**, **me Parameter** is automatically inserted when a new **Operation** of this **Class** is created. This **Parameter** has for **Type** main **Type** of the **Class**, and an **in out** mode (receiver may be modified by **Operation** execution).

3.2.2.4. Calling Operations of a HOOD Class

If we design previous example with **HOOD Classes**, we would define a **Class** engines providing (at least) a **Type** engine (whatever its name is) and an **Operation** start.



Control system could thus declare internal variables being two instances of this **Class** and call their respective **Operation**:

```
-- in Ada
-- with engines; use engines;
  l,r : engine;
  start(l);
  start(r);
```

```
// in C++
// #include engines.h
  engine l,r;
  l.start();
  r.start();
```

3.2.2.5. *Designing with HOOD Classes*

It is often uneasy to make the right design choice between an **Object** and a **Class** for a given **Application** entity. Software analysis phase outputs could be a guide for that, but they are generally highly dependent on the actual analysis process: a conventional Functional Analysis will often lead to **Objects**, whereas Object Oriented Analysis approaches will identify **Classes** for everything.

In general case, there is no direct mapping, and it is a full design choice to represent a given **Application** entity as an **Object** or a **Class**. Here are a few criteria for choosing a **Class**:

- Similar entities appear frequently, and there is no need to manage them individually at architectural design phase.
- Entities are dynamically instantiated at run-time (there is no way to define a precise number of instances at design phase).
- Static interactions (is kind of, is part of) between a set of **Modules** seems more important to be described at architectural design phase than their dynamic interactions (uses operations of).
- A **Module** is clearly a basic unit for general purpose reuse.

3.2.2.6. Creating HOOD Classes

Classes may only be created in *graphic editor* (except **Root Classes** that are created from *main editor*: (refer to part I). To create a new **Class**, open a **HOOD** diagram at specified level in the hierarchy, and then use *class module* item of *create* menu or drawing area pop-up menu.

In some cases, you may also make an **Object** become a **Class**. If the **Object** contains a main **Type** and **Operations** having at least a **Parameter** of this **Type**, it may be changed into a **Class** by setting main **Type** property *class* to *yes*.

To perform this change, select relevant **Object**, change display view to *types*, select main **Type**, and modify *class* property inside *General* tab of text input area. This change may be reversed.



3.2.3. Generic Module

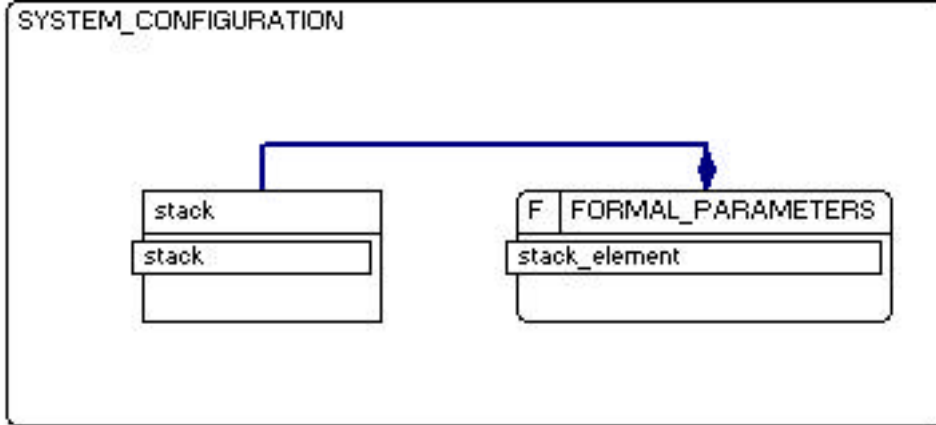
In addition to **OO** classes derivation and instantiation, genericity provides another powerful way to manage components reuse. **Generics** (previously badly named “classes”), were early introduced among **HOOD** concepts to describe parametric **Modules**.

Generic Modules are particular kind of **Modules** for which some internally used **Components** are temporarily left undefined. Typical implementations of **HOOD Generic Modules** are **Ada** generic units and **C++** templates.

3.2.3.1. *Formal_Parameters*

These undefined **Components** are called “formal parameters” of **Generic**. When describing contents of a **Generic Module**, formal parameters may be used anywhere, but are not actually part of it. To define formal parameters in a correct way regarding **HOOD** visibility rules, they need to be located outside the **Generic Module**.

Practically, when defining a **Generic Module**, another very particular **Module**, called **Formal_Parameters** need be created to provide parametric **Components**. These **Components** do not need to be fully defined, but only declared, and may be **Operations**, **Types**, **Constants** and **Exceptions**.



3.2.3.2. Designing with Generic Modules

Generic Modules are an efficient way to provide reusable entities. A classical example is the definition of a stack of elements. Whatever this stack is designed as a **HOOD Object** (that is it actually contains stack elements), or a **HOOD Class** (it only defines stack structure and operations), it will be a lot more reusable if we can describe it with the **Type** of stack element as a formal parameter. **Type** of stack elements will thus be only declared, and referred everywhere it is needed in stack implementation.

In order to comply with **HOOD** visibility rules, and to emphasize the fact that a **Generic Module** is a reusable entity and should thus be accessed from everywhere in an **Application**, it should be a **Root Module**. That is why, when creating a new **Root Module** in *main editor* (refer to part I of this User's Manual), you may choose *generic*. There is no other way to create a **Generic Module**.

3.2.4. Instance_Of a Generic Module

A **Generic Module** is never fully designed, due to the incomplete definition of its formal parameters. To practically use a **Generic Module**, it is required to fully define these parameters. A same **Generic Module** may of course be used many times in a given **Application**, by setting different actual values to formal parameters.

3.2.4.1. Instantiating Generic Modules

The fact of setting values to formal parameters is called instantiation of relevant **Generic Module**, and the result of this affectation is a new kind of **HOOD Module** called **Instance_Of**.

When an **Instance_Of** is created, it must refer to a precise **Generic Module**, and fully define all formal parameters. Definition of formal parameters (which then become “actual parameters”) may be allocating a value or making a reference to an existing and visible **Component**. There is no other needed work to complete an **Instance_Of** definition.

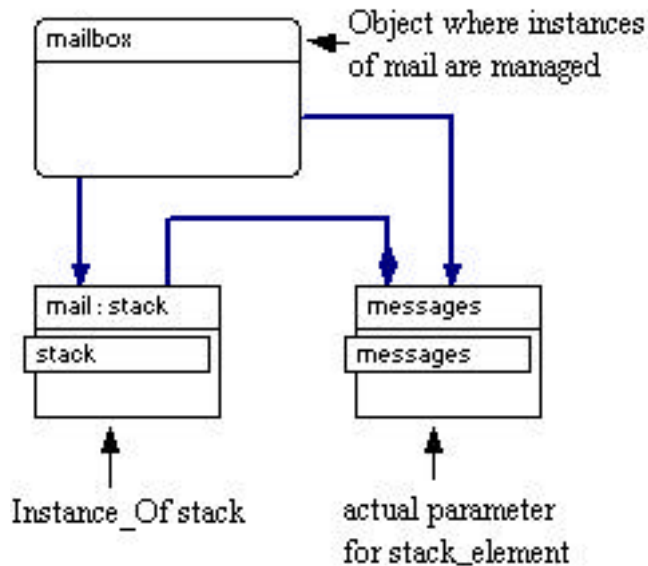
3.2.4.2. Creating an Instance_Of

Creation of **Instance_Of Modules** can be performed in *graphic editor*, with *instance of generic* item of *create* menu or drawing area pop-up menu. A list of currently visible **Generic Modules** will then be displayed, and one of them should be selected.

To make a **Generic Module** visible from a given **Application**, it should be declared in *root modules* list of current **Project**. If it is not, use *system editor* to add it to the list, and if it doesn't even appear in *system editor*, change `SavePath` initialization property. Please refer to part I for further details.

Provided Interface of an **Instance_Of** is automatically copied from relevant **Generic**. This interface will also be automatically updated each time the **Application** is loaded, in order to follow **Generic** changes. Definition of actual parameters must be performed within a textual editor (refer to part III).

An **Instance_Of** may also be a **Class**, if it was created from a **Generic Class**. Note that, in this case, two sequential instantiations are required to use this piece of code somewhere. First instantiation transforms **Generic Class** into **Instance_Of Class**, and second instantiation builds variables which **Type** is main **Type** of this **Class**.



3.2.4.3. *Libraries of Generic Classes*

Due to its tidy visibility rules, **HOOD4** provides a very efficient way to organize **Classes** libraries, and especially libraries of **Generic Classes** (for instance libraries of templates in **C++**). **Generic Classes** may be grouped into a common **Generic Module** from which they will share the unique formal parameters list.

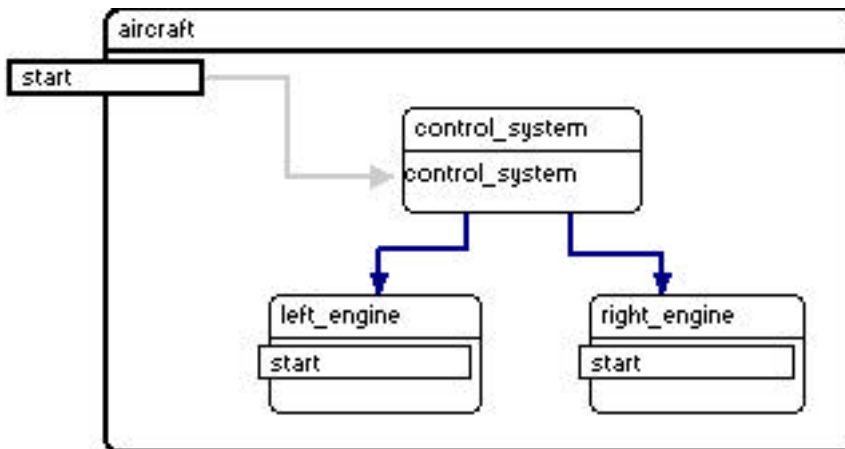
A **Generic Module** could for example be designed to define various kinds of collections (stacks, lists, ...) of elements which **Type** could be a common formal parameter. This constrained design structure provides an actual warranty for an efficient use and maintenance of the library.

3.2.5. Op_Control Module

HOOD Classes are the Object Oriented way to use **HOOD Modules**. At the opposite, there is a “pure” Functional way to define **HOOD Modules**. These kinds of **Modules** are called **Op_Controls**. They have the strong particularity to contain only one **Operation**, and nothing else.

To create an **Op_Control**, it is first needed to create a plain **Object**, then to change its *module type* property, in text input area, by setting it to O. **Op_Controls** cannot be **Root Modules** and cannot be broken down.

Existence of **Op_Controls** is often justified by the need to create “dispatching” entities. Nevertheless, this task may easily be performed by a plain **HOOD Object**. In our previous "aircraft" example, if we limit the functions of the "control_system" to the action of starting the two engines, we could use an **Op_Control** for that. Practically, it is likely that control_system would have to perform other tasks. This situation often occurs, so we strongly recommend to avoid using **Op_Controls**.



3.2.6. Environment Module

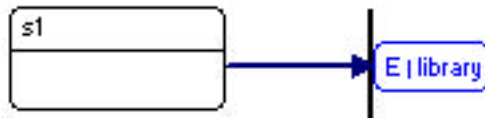
HOOD Projects generally encompass several **Applications** or libraries. We already described in part I how such a **Project** could be defined thanks to *main editor* and *system editor*. Anyway, when designing a given **Application**, it is likely that explicit references to the other members of the **Project** will be required.

In accordance to **HOOD** rules, only **Provided Interface** of external **Root Modules** should be seen from current **Application**. In other words, other **Application** of the same **Project** or libraries can be represented locally by a dedicated **Terminal Module** containing only a **Provided Interface** and no **Internals**. These **Modules** are called **Environment**, and due to **HOOD** visibility rules, they must be located at the same level as current **Root Module**.

Environments may be **Objects** or **Classes**, **Active** or **Passive**. **Generics** are in effect kinds of **Environments**, but we saw previously that, due to the need of instantiation, they are managed in a different way (refer to § 3.2.3).

In **STOOD**, **Environments** may be represented graphically like other **Modules**. In this case, they appear in a pseudo **HOOD** diagram of current **System Configuration**. To get access to this diagram, go up once from **Root** level. It is not mandatory to represent each member of current **Project** by an **Environment**, but this may be helpful to control finely dependencies and code generation. On the contrary, some **Environments** may be omitted graphically to reduce complexity of very large **Projects**, of when there is a too large number of libraries.

To create an **Environment**, first select **System Configuration** diagram in *graphic editor*, and create a new **Object** or **Class** at this level. **Environments** may also be created when drawing **Use** relationships which refer to external **Modules**. In this last case, an **Environment** box is created outside parent **Module** borders:



When giving a name to an **Environment**, in text input area, two events may occur:

- Given name is the same as one of the elements of *root modules* list in *main editor* (that is one of the **Applications** or libraries of current **Project**). In this case, **Environment** is automatically updated from relevant **Root Module** interface. This update will occur each time current **Application** is loaded, so **Environment** interface will follow relevant external **Application** changes.
- Given name does not match any name of known **Applications** in the **Project**. In this case, **Provided Interface** of the **Environment** will have to be created manually, and a warning message will appear when checking **HOOD** rules. To highlight these unbounded **Environments**, they are tagged by a letter **E**.

In all cases in **STOOD**, an **Environment** remains only a local copy of a true or virtual other **Application** or library of current **Project**. They should never be broken down with **Include** relationship. If their local interface is changed and is not consistent with remote reference interface, these modifications won't be saved, and will be lost at next loading.of the **Application**.

3.2.7. Summary of HOOD Modules kinds

3.2.7.1. Main Module Kinds towards HOOD hierarchy

Following table summarize the allowed combinations of **HOOD Module** kinds towards their location in **Design Tree** hierarchy.

	Root	Non Terminal	Terminal
HOOD Object	—	—	—
HOOD Class	—	✘	○
HOOD Generic	○	—	—
Instance_Of	—	✘	○
Op_Control (O)	✘	✘	○
Environment (E)	○	✘	○

Forbidden: ✘ Allowed: — Mandatory: ○

3.2.7.2. Passive and Active Modules

HOOD Modules may also be classified as regards their dynamic behaviour inside the **Application**:

- **Passive** (P: default value)
- **Active** (A)

An **Active Module** is supposed to run one or several independent execution threads (tasks, processes, ...), whereas a **Passive Module** executes its **Operations** within calling thread. An **Application** without any **Active Module**, will have only one execution thread (the one triggered when launching main procedure). To enable **Active Modules** communications, **Operations Constrained by Protocol** should be used. These **Constraints** define various kinds of synchronization protocols between separate threads. Please refer to § 3.3.3.3 for further details.

Except **Op_Controls**. and **Formal_Parameters**, all kinds of **Modules** may be **Active** or **Passive**. An **Active Module** is identified graphically by a **A** letter in top left corner of its box. This indicator sometimes conflicts with other information, typically the **E** letter for **Environment**. As an **Environment** may easily be identified by its location within **Design Tree**, **STOOD** will display **A** rather than **E** each time it is possible.

3.2.7.3. Other kinds of Modules

We didn't describe above in details other kinds of **HOOD Modules** that may be mentioned in other parts of this manual or in other papers about **HOOD**. Here are briefly some of them:

- **Virtual Node** (**v**) for distributed **Applications** (refer to part I).
- **Multiple Instance_Of** (not supported).
- **Cyclic** (**C_y**), **Sporadic** (**S_p**) and **Protected** (**P_r**) **Objects** (**HRT_HOOD**).

3.2.7.4. Changing Modules kinds

When creating a new **HOOD Module** inside **STOOD graphic editor**, its default kind will be:

- **Passive Environment**, if current diagram is the one of the **System Configuration**.
- **Terminal Passive**, else.

Terminal Passive may be changed into other kinds of **Modules** as follow:

- into a **Terminal Active**: change *module type* property to **A**.
- into an **Op_Control**: change *module type* property to **O**.
- into a **Non Terminal**: create child **Modules** inside its diagram.
- into a **Design**: copy *as root module* then delete it.
- into a **Generic**: copy *as generic* then delete.
- an **Object** into a **Class**: set *class* property of its main **Type** to **yes**.
- a **Class** into an **Object**: set *class* property of its main **Type** to **no**.

Module kind may be changed with *module type* field of text input area when relevant **Module** box is selected within drawing area. **Module type** item of *edit* menu may also be used for that purpose.

3.3. Operations

HOOD Operations describe functional services of a **Module**. During code generation, they are directly mapped to subprograms (**Ada**) or functions (**C** and **C++**).

To be used by a remote client **Module**, an **Operation** must be declared in **Provided Interface** of server **Module**. To avoid an **Operation** from being used from outside, it should be located inside the **Internals** of the **Module**.

Concept of “protected” **Operation** does not exist in **HOOD**. Such **Operations** are supposed to be visible only from sub-**Classes**. To create a **C++** protected **Operation**, it should be located in **Provided Interface** of its **Class Module**, and an additional flag will have to be set before code generation (**Pragma protected**, refer to part IV).


Operations are defined by a declarative part, located in the **Provided Interface** or the **Internals**, and a body (**Operation Control Structure: OPCS**) always located in the **Internals** of **Terminal Modules**. Only declarative part may be managed during architectural design phase. **Operation** bodies should be defined during detailed design phase (refer to part III).

In **STOOD**, *graphic editor* is the best place to create, move, delete and edit declarative part of an **Operation**. Display area should be first switched to *operation* view. Some of these actions may also be performed in text editors.

3.3.1. Select an existing Operation

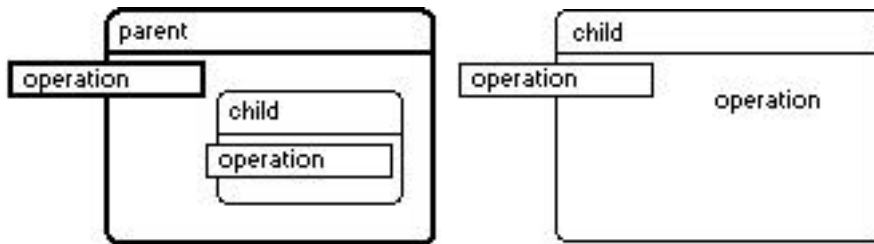
Within drawing area, an **Operation** may be selected by a single click on its name. The name of currently selected **Operation** is surrounded by a thin rectangle, so it may easily be identified. Another single click will deselect it.

3.3.2. Create a new Operation

As described in § 2, when display area shows *operation* view, **operation** item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Operation**. After having performed one of these actions, a string "operation" appears and may be moved with mouse pointer. It can be dragged to one of the following locations, as regards required scope:

- The **Provided Interface** of parent **Module**.
- The **Provided Interface** of a child **Module** if any.
- The **Internals** of parent **Module** if there is no child.

After having chosen the right location, a single click will fix the **Operation**. It may be moved later if required (refer to § 3.3.4).



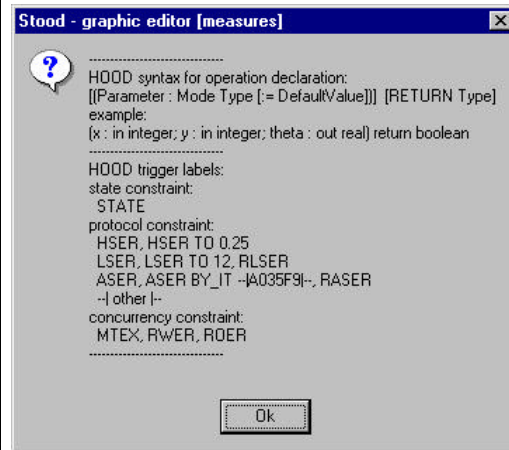
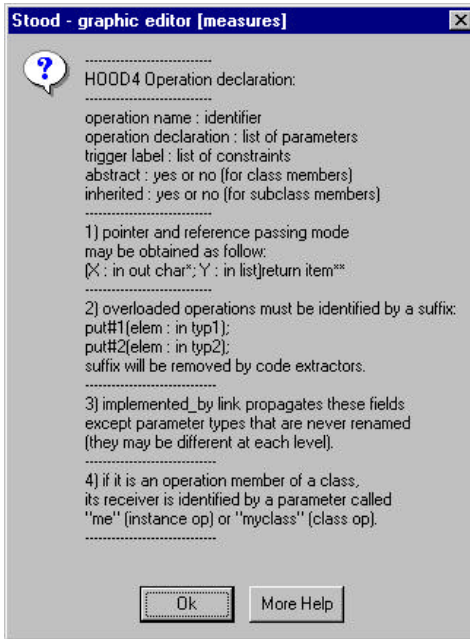
Another indirect way to create **Operations** is by drawing **Implemented_By** links for **Operation_Sets**. This is described in § 3.10.

3.3.3. Edit Operation declarative part

When an **Operation** is selected in drawing area, its declarative part may be edited inside *General* tab of text input area. Declarative part of a **HOOD Operation** contains five fields that are described below.

```
operation name : set_color#1
operation declaration : (me : in out rectangle; paint : in color)
trigger label :
abstract : no
inherited : yes
```

Help item of text input area pop-up menu provides on-line information about these five fields:



Note that contents of these help box may be customized, by editing `gra_txt_ope` and `gra_txt_ope.more` files in `config/help` configuration directory (refer to part I).

3.3.3.1. *operation name and overloading*

operation name field must be used to insert actual name of selected **Operation**. Default name is `operation` and should be changed into a more explicit name. If it remains unchanged, a warning message will appear while checking **HOOD** rules, as **Operation** is a reserved word and should not be used for identifiers. When giving a name to an **Operation**, please take care not to use forbidden characters nor a too long string. Restrictions may come from:

- Compliancy with **HOOD** identifiers naming rules. These rules are checked when accepting text input, and a dialog box will be displayed in case of lexical or syntactic error.
- Target language (**Ada**, **C**, **C++**) naming rules. Except abnormal use of reserved words, uncompliancies to these rules will not be checked by **STOOD**, and will lead to compilation errors.
- Current file system naming rules. Operation names are used to create files in **Application** storage area. Uncompliancies to these rules could lead to information loss at detailed design phase.

Overloading of **Operations** is allowed by **HOOD**. This means that several **Operations** may have the same name inside a same **Module**. **Operation** declaration field should be different for each of them, however. In order to help handling overloaded **Operations**, **STOOD** provides an extended syntax for **Operation** names. This syntax consists in a suffix identifying each overloading:

```
draw#1  for:    draw(me : in out line)
draw#2  for:    draw(me : in out rectangle)
```

This suffix (built with a “#” followed by one digit) represents full **Operation** declaration. This suffix is automatically removed during code generation. Note that while designing for C++, suffix “#0” represents the destructor of a **Class**, and will be changed into relevant valid syntax at code generation.

the_class#0 *means:* ~the_class

3.3.3.2. operation parameters list

operation declaration field must be used to declare **Operation Parameters** list. This **Parameters** list needs to be written in compliance with **HOOD** syntax (and not target language syntax), which is derived from **Ada** one. **STOOD** specific extensions help in using some C/C++ syntactic features.

HOOD Parameters are typed variables or constants, with a “passing mode” additional property. Passing modes deal with both **DataFlow** direction (**in**, **out** or **in out**, like in **Ada** syntax) and **Parameter Type** specifiers (nothing, * or &, like in C/C++ syntax). Default values may also be specified optionally.

General syntax for accepting **Operation Parameters** list in **STOOD** is as follow: ([...] means 0 or 1; {...} means 0 or more)

```
(1) Par_list ::= [(Param2{; Param2})]
                [return [Spe]Return_type]
(2) Param ::= Par_name : Dir [Spe]Par_type
                [:= Par_value]
```

Par_name means **Parameter** name and is an identifier. A **Parameter** name may be used only once within a given **Parameters** list. When an **Operation** belongs to a **Class** (C++ member functions or **Ada95** primitive operations), a dedicated **Parameter** name **me** should be used to identify **Operation** receiver. In the particular case of static member functions, dedicated **Parameter** name **myclass** should be used.

Par_type means **Parameter** type and is a reference to an existing local or remote **Type**. If it is remote, dotted notation should be used.

Return_type means return type and is a reference to an existing local or remote **Type**. If it is remote, dotted notation should be used. Note that there is no direct way to specify here a `const` return **Type**. This feature will be obtained by setting an additional flag before code generation (**Pragma** `return_const`: refer to part IV).

Par_value means **Parameter** default value and may be a reference to an existing local **Constant**, **Data** or **Operation** with a return value, a reference to a remote **Constant** or remote **Operation** with a return value (with dotted notation), or a string delimited by double quote characters. If default value itself contains double quote characters, it should be surrounded by **HOOD** text field separators: `--" a string"--`.

Dir means **DataFlow** direction and specifies whether a **Parameter** is produced by client (`in`), by server (`out`) or produced by client and modified by server (`in out`). Note that this information may also appear on **HOOD** diagrams when drawing **Use** relationships (refer to § 3.6.5). These passing modes come directly from **Ada** language definition, but may also be favourable when using **C** or **C++**. `in` flow direction will generally be used to control `const` keyword insertion during **C** and **C++** code generation.

Spe means **Parameter** type specifier and indicates whether a **Parameter** is passed *by_value*, *by_pointer* or *by_reference*. Default passing mode is *by_value*, and to ease **Parameters** list input in **STOOD**, *by_pointer* should be written *** and *by_reference* should be written *&*. These tokens come obviously from **C/C++** syntax, but a *** passing mode in **Ada95** will be translated into *access*. More complex passing modes may also be specified directly inside **Parameters** list, like **** or **&*, but array arguments cannot. To specify an array argument, a new **Type** should be first declared, and may then be referred inside a **Parameters** list.

Operations Parameters may also appear as **DataFlows** carried by **Use** relationships on **HOOD** diagrams (refer to § 3.6.5).

Please note that this syntax differs lightly from the one of **HOOD** Standard Interchange Format (**SIF**). **STOOD** *sif extractor* performs automatically all required syntax transformations to fit **HOOD4** requirements.

3.3.3.3. *real-time properties*

trigger label field should be used to specify behavioural constraints of the **Operation**. A **HOOD Module** external behaviour is driven by the way its **Provided Operations** react. **HOOD** specifies a set of predefined **Operation Constraints** to manage **Real-Time** features of an **Application** at architectural design phase.

Operation Constraints may be classified as follow:

- **State Constraints** indicate to a client that called **Operation** receptiveness may depend on server internal **State**. If a non buffered printer is busy, another request will be refused: “print” **Operation** of “printer” **Module** is constrained by **STATE**. Internal **States** and **Transitions** of a **Module** should be described with a **State Transition Diagram (STD)**: refer to § 4).
- **Protocol Constraints** refer to multi-tasking **Applications** synchronization **Operation** calls. A “wait-reply” protocol will be simply identified by a **HSER (Highly Synchronous Execution Request) Constraint**. An “acknowledge” protocol will be identified by a **LSER (Loosely Synchronous Execution Request) Constraint**. Other **HOOD4** predefined protocol **Constraints** are: **ASER (ASynchronous Execution Request)**, **ASER BY_IT** (hardware interrupt), **RLSER (Reporting LSER)**, and **RASER (Reporting ASER)**. A “time-out” may also be specified with **TO** constraint.

-
- **Concurrency Constraints** may be used to provide coordinated access to shared **Data** within a multi thread **Application** (mutual exclusion). **HOOD4** predefined concurrency constraints are: **MTEX** (**MuTual EXclusion**), **ROER** (**Read Only Execution Request**) and **RWER** (**Read Write Execution Request**).

Some **Constraints** require an additional parameter. It identifies interrupt source by a string for **ASER BY_IT**, and a numeric delay value for **TO**.

```
ASER BY_IT "timer 1ms"  
TO 1.5
```

A same **Operation** may carry several **Constraints**, and *trigger label* description field should be filled in with a list of elementary **Constraints** separated by a space character. Note that only first listed **Constraint** will be displayed on relevant **HOOD** diagram, not to overload too much graphical description. For a good understanding of the diagram, please take care to order listed **Constraints** and put most important one at the beginning. **Constraints** order have no other effect.

Practical use of **Operation Constraints** is often limited by actual available implementation of regarding concepts into target context (language and executive environment). For **Ada Applications**, and especially with **Ada95**, implementation of most of them is quite straightforward, and protocol **Constraints** will lead to task entries and concurrency **Constraints** to protected entries or procedures. For other languages, or for dedicated **Ada** run-time environments, a specific mapping should be defined first. Note that **STOOD** offers powerful customization capabilities in order to implement such specific mapping into code generators.

3.3.3.4. *object-oriented properties*

abstract and *inherited* fields should be used to provide additional information regarding polymorphism of **Operations** belonging to a **Class** inheritance hierarchy. These two fields are flags that could be set to `yes` or `no` (`no` is default value).

An **Abstract Operation** should belong to a **Class** that has sub-**Classes**. This flag indicates that this **Operation** is not implemented locally and must be redefined inside at least one of the sub-**Classes**. With **STOOD**, to highlight **Abstract Operations** in **HOOD** diagrams, their name are surrounded by square brackets. At code generation, no body will be produced for **Abstract Operations**.

An **Inherited Operation** should belong to a sub-**Class**, and is simply a local reference to relevant super-**Class Operation**. An **Inherited Operation** should have the same name and **Parameters** list as inheriting one, except the **Type** of its receiver (`me`). With **STOOD**, to highlight **Inherited Operations** in **HOOD** diagrams, their name are surrounded by rounded brackets. They help controlling **HOOD** visibility rules, and are not translated at code generation. Use of **Inherited Operations** is optional.

Note that a sub-**Class Operation** may also overload a super-**Class Operation** of the same name. In this case, *abstract* and *inherited* flags should be kept to `no`.

3.3.4. Move an Operation

An **Operation** may be moved inside a same diagram, under certain conditions. In all cases, moving an **Operation** is a simple "drag and drop" action, performed by following sequence:


- select wished **Operation** without releasing left mouse button.
- drag it while keeping left mouse button pressed.
- release left mouse button at destination location.

Allowed move actions are:

- changing the rank of selected **Operation** within the list.
- moving an **Operation** from or to an open **Operation_Set**
- moving from **Internals** to **Provided Interface** (or reverse)
- moving to **Provided Interface** of sibling **Modules**.

Note that moving from parent to child **Provided Interfaces** (or reverse) are forbidden.

3.3.5. Delete an Operation

To delete an **Operation**, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

3.4. Operation_Sets

It is likely that for actual **Applications**, the number of **Provided Operations** of some **Modules** become too large to be displayed on a diagram without making it unreadable. This may even appear in a simple **Terminal Class**. Hopefully, **HOOD** provides a dedicated view for **Operations**, but this is not always sufficient.

Another operational need is to define functional entities at a higher level of abstraction than an elementary **Operation**. This often appear when describing high level **Modules** interfaces, where identifying “I/O operations” could be more convenient than describing in detail all those **Operations**. Please note that all **Sets** should be fully filled in at the end of design process. **STOOD** provides solutions to automatically update **Set** contents, while drawing **Implemented_By** relationships (refer to § 3.10).

To fit these two requirements, **HOOD** provides the concept of **Operation_Set**, which represents a group of elementary **Operations**, or of other lower level **Operation_Sets**. To ease understanding, all **Operations** belonging to a given **Operation_Set** should be logically linked. This logical link should follow current design style: it may be functional (put, get, read, write, ... are all I/O **Operations**), or more **Object-Oriented** (open, close, print, ... may work on a same file).

Operation_Sets are identified by their name, which appear on **HOOD** diagrams surrounded by braces. An **Operation_Set** may be open or closed, to show or hide its contents. To open a closed **Set** or close an open **Set**, just double-click on its name in drawing area. You may also select relevant **Set** on the diagram, and use *enter* (to open) or *exit* (to close) items of *edit* menu.

Theoretically, **Operation_Sets** should be located only within **Provided Interface** of a **Module**. As an extension, **Internal Operation_Sets** can easily be introduced into **STOOD** for **Terminal Modules** if required (please contact technical support). Note that **STOOD** also extends **HOOD Operation_Set** concept to other components. It is thus possible to define **Type_Sets**, **Constant_Sets**, **Data_Sets** and **Exception_Sets**. Following chapters refer to all kinds of **Sets**.

3.4.1. Select, open and close an existing Set

Within drawing area, a **Set** may be selected by a single click on its name. The name of currently selected **Set** is surrounded by a thin rectangle, so it may easily be identified. Another single click will deselect it.

A double-click on a closed **Set** will open it, whereas a double-click on an open **Set** will close it. *enter* and *exit* items of *edit* menu may also be used to open and close **Sets**.

3.4.2. Create a new Set

As described in § 2, when display area shows *operation* (respectively *type*, *constant*, *exception* or *data*) view, `{operation-set}` (resp. `{type-set}`, `{constant-set}`, `{exception-set}`, or `{data-set}`) item of drawing area pop-up menu, or of *create* menu may be used to create a new **Operation_Set** (resp. **Type_Set**, **Constant_Set**, **Exception_Set**, or **Data_Set**). After having performed one of these actions, a string called `operation_set` (resp. `type_set`, `Constant_Set`, `exception_set`, `data_set`) may be moved with mouse pointer. It can be dragged to one of the following locations:

- The **Provided Interface** of parent **Module** (except for **Data_Sets**).
- The **Provided Interface** of any child **Module** (except for **Data_Sets**).
- The **Internals** of parent **Module** if there is no child.

After having chosen the right location, a single click will fix the **Set**. It may be moved later if required (refer to § 3.4.4).

Another indirect way to create **Sets** is by drawing **Implemented_By** links for higher level **Sets**, as described in § 3.10.

3.4.3. Edit Set properties and contents

When a **Set** is selected in drawing area, its **HOOD** definition appears inside *General* tab of text input area:



```
set: { set_name: set_contents }
```

set_name represents the name of the **Set**. When creating a new **Set**, its default name will be *operation_set* (or *type_set*, *constant_set*, *exception_set*, *data_set*, regarding current view). Note that **Operation_Set** is a **HOOD** reserved word and should be changed into a more explicit applicative name, else a warning message could appear while checking **HOOD** rules.

set_contents is the list of contained **Components** (**Operations**, **Types**, **Constants**, **Exceptions** or **Data**) or other **Sets** of the same kind. This field cannot be modified textually: it is automatically deduced from graphical description. Default contents is empty, and to add an element to a **Set**, first open it, and then move the element (an elementary **Component** or another **Set**) between the two braces. **Set** definition inside text input area will be automatically updated.

3.4.4. Move a Set

A **Set** may be moved inside a same diagram, under certain conditions. In all cases, moving a **Set** is a simple "drag and drop" action performed with following sequence:


- select wished **Set** without releasing left mouse button.
- drag it while keeping left mouse button pressed.
- release left mouse button at destination location.

Allowed move actions are:

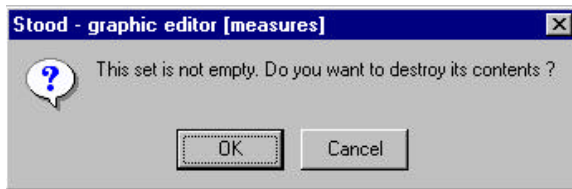
- changing the rank of selected **Set** within the list.
- moving an **Set** from or to another open **Set**
- moving from **Internals** to **Provided Interface** (or reverse)
- moving to **Provided Interface** of sibling **Modules**.

Note that moving from parent to child **Provided Interfaces** (or reverse) are forbidden.

3.4.5. Delete a Set

To delete a **Set**, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

Note that a **Set** must be empty to be deleted. Else, following warning message will be displayed:



3.5. Exceptions

A **HOOD Exception** may be defined to deal with errors or other exceptional situations that arise during **Application** execution. An **Exception** comes in response to a previous **Operation** call. Called **Operation** may “raise” an **Exception** and calling **Operation** may “handle” it or “propagate” it to its own caller. **Exceptions** that are not handled inside applicative code, will be processed by run-time executive or produce an error.


A **HOOD Exception** should be declared only inside **Provided Interface** of a **Module**. As an extension, **Internal Exceptions** can easily be introduced into **STOOD** for **Terminal Modules** if required (please contact technical support). **Exceptions** are fully defined by their declarative part. **Exceptions** “raising” and “handling” code are fully defined inside relevant **Operation** bodies.

To specify along which **Use** relationship a raised Exception is propagated, an **Exception Flow** may also be inserted in the diagram (refer to § 3.6.6).

3.5.1. Select an existing Exception

Within drawing area, an **Exception** may be selected by a single click on its name. The name of currently selected **Exception** is surrounded by a thin rectangle, so it may easily be identified. Another single click will deselect it.

3.5.2. Create a new Exception

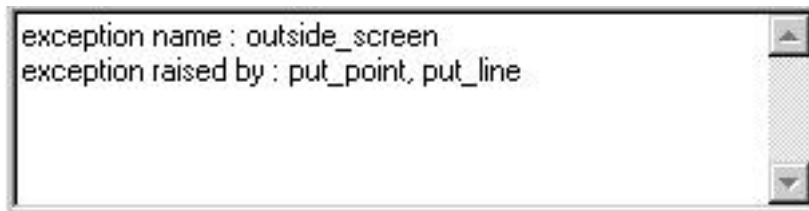
As described in § 2, when display area shows *exception* view, `exception` item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Exception**. After having performed one of these actions, a string called `exception` appears and may be moved with mouse pointer. It can be dragged to one of the following locations:

- The **Provided Interface** of parent **Module**.
- The **Provided Interface** of any child **Module**.
- The **Internals** of parent **Module** if there is no child.

After having chosen the right location, a single click will fix the **Exception**. It may be moved later if required (refer to § 3.5.4)

3.5.3. Edit Exception properties

When an **Exception** is selected in drawing area, its declarative part may be edited inside *General* tab of text input area. Declarative part of a **HOOD Exception** contains only two fields that are described below.



3.5.3.1. exception name

exception name field must be used to insert actual name of selected **Exception**. Default name is `exception` and should be changed into a more explicit name. If it remains unchanged, a warning message will appear while checking **HOOD** rules, as **Exception** is a reserved word and should not be used for identifiers.

When giving a name to an **Exception**, please take care not to use forbidden characters nor a too long string. Restrictions may come from:

- Compliancy with **HOOD** identifiers naming rules. These rules are checked when accepting text input, and a dialog box will be displayed in case of lexical or syntactic error.
- Target language (**Ada**, **C++**) naming rules. Except abnormal use of reserved words, uncompliances to these rules will not be checked by **STOOD**, and will lead to compilation errors.

-
- Current file system naming rules. **Exception** names are used to create files in **Application** storage area. Uncompliances to these rules could lead to information loss at detailed design phase.

3.5.3.2. list of raising operations

exception raised by field must be used to specify the list of local **Provided Operations** that could actually raise selected **Exception**. List separator is a comma.

3.5.4. Move an Exception

An **Exception**

all cases, moving an **Exception** is a simple "drag and drop" action, that may be performed by following sequence:


- select wished **Exception** without releasing left mouse button.
- drag it while keeping left mouse button pressed.
- release left mouse button at destination location.

Allowed move actions are:

- changing the rank of selected **Exception** within the list.
- moving an **Exception** from or to an open **Exception_Set**
- moving from **Internals** to **Provided Interface** (or reverse)
- moving to **Provided Interface** of sibling **Modules**.

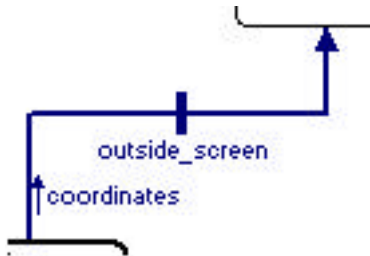
Note that moves from parent to child **Provided Interfaces** (or reverse) are forbidden.

3.5.5. Delete an Exception

To delete an **Exception**, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.



3.6. Op_Use relationships



When *operations* view of **HOOD** diagrams is displayed, only two kinds of relationships are shown. **Implemented_By** relationships specifies existing links between parent **Operations** and local child **Operations** (refer to § 3.10). All other dependencies between **Modules** that are shows on *operation* view are called **Use** relationships (they are also sometimes called **Op_Use**).

Use relationships show functional links between two distinct **Modules**, that's why they are displayed only on *operation* view of **HOOD** diagrams. A **Use** relationship between **Module** “client” and **Module** “server” means that “at least one **Operation** of client calls at least one **Provided Operation** of server”.

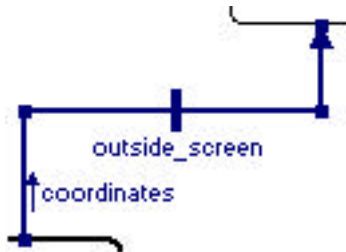
Use relationships do not describe all elementary functional calls between each **Operation**, that could quickly become unreadable. On the contrary, they provide a high level description of functional dependencies between **Modules**. They act as the same level as a *with* clause in **Ada** or a *#include* in **C/C++**. More detailed dependencies can be analysed with **HOOD Required Interface** descriptions and **STOOD** cross references tables (refer to part III).

Use relationships provide the unique way to exchange variable **Data** between **HOOD Modules**. Practically, these **Data** are carried by actual values of called **Operations Parameters**. An use relationships is thus a medium for **DataFlows** between **Modules**. Note that graphical description of **DataFlows** remains optional on **HOOD** diagram. Their formal and exhaustive definition is fully provided by **Operations Parameters** list (refer to § 3.3.3.2).

In accordance to **HOOD** visibility rules, a **Use** relationship may only be drawn from a local **Module** to a “sibling” **Module**, an “uncle” **Module** or an **Environment Module**.



If “server” **Module** provides **Exceptions**, the only way to catch such an **Exception** inside “client” **Module**, is a response to a previous server **Provided Operation** call. An **Use** relationships is thus also a medium for **Exception_Flows** between **HOOD Modules**. Like **DataFlows**, graphical descriptions of **Exception_Flows** are optional on **HOOD** diagrams.

3.6.1. Select an existing Use relationship

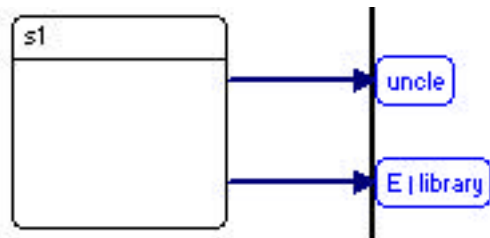


Within drawing area, a **Use** relationship may be selected by a single click on one of
of
it may easily be identified.

3.6.2. Create a new Use relationship

As described in § 2, when display area shows *operation* view,  item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Use** relationship. After having performed one of these actions, source **Module** should be selected first by a single click, then destination **Module** should be selected by another single click.


Source must be one of the child **Modules** of current diagram, and destination may be one of its siblings or existing uncles, or an **Environment Module**. When the area outside parent borders is used as destination, and is not an existing uncle box, a new **Environment** box is created.



3.6.3. Move an Use relationship

An **Use** relationship may be moved by dragging one of its handles with the mouse. If additional handles are required, select one of existing handles, and use **Lenghten** item of *edit* menu to create new segments. If useless segments need to be removed, use **Shorten** item of *edit* menu (refer to § 2.6.4).

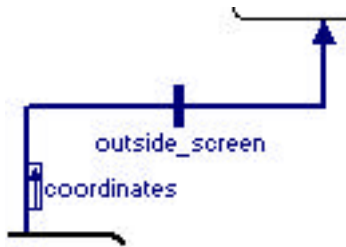
3.6.4. Delete an Use relationship

To delete an **Use** relationship, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

Deletion of a **Use** relationship also deletes related **DataFlows** and **Exception_Flows**.




3.6.5. DataFlows




3.6.5.1. Select an existing DataFlow



Within drawing area, a **DataFlow** may be selected by a single click on its arrow. The arrow of currently selected **DataFlow** is surrounded by a thin rectangle, so it may easily be identified.

3.6.5.2. Create a new DataFlow

As described in § 2, when display area shows *operation* view, , , or  items of drawing area pop-up menu or *create* menu may be used to create a new empty **DataFlow** list. After having performed one of these actions, a small arrow appears and may be moved with mouse pointer. It can be dragged only along one of the existing **Use** relationships.

Direct **DataFlows** () represent **in Parameters**, reverse **DataFlows** () represent **out Parameters**, and bidirectional **DataFlows** () represent **in out Parameters**.

After having chosen the right location, a single click will fix the **DataFlow**. It may be moved later if required.

3.6.5.3. *Edit DataFlow list*

When a **DataFlow** is selected in drawing area, *data flow list* may be edited within text input area.




This list should be related to the **Parameters** that are actually passed by the **Operation** calls associated to current **Use** relationship. Nevertheless, there is no need to copy in *data flow list* all those **Parameters**, but simply to highlight most important ones by writing an informal label. No consistency check between *data flow list* and actual **Parameters** will be performed. The goal of *data flow list* is to help understanding **HOOD** diagrams.

data flow list may contain any string enclosed by double quote characters, or a list of valid identifiers delimited by commas. This string is used as a label for relevant **DataFlow** on the diagram. When the string needs to contain double quote characters, **HOOD** text delimiters --| |-- should be used to enclose the overall string.

3.6.5.4. *Move a DataFlow*

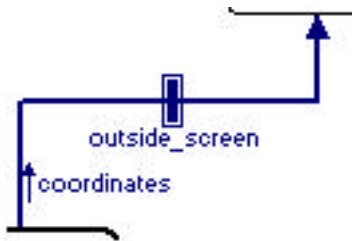
A **DataFlow** arrow may be moved with the mouse, along carrying **Use** relationship. It cannot be moved to another **Use** relationship.

3.6.5.5. *Delete a DataFlow*

To delete a **DataFlow**, it should be selected first, and **D**elete item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **U**ndo menu item.


3.6.6. Exception_Flows

3.6.6.1. Select an existing Exception_Flow



Within drawing area, an **Exception_Flow** may be selected by a single click on its bar. The bar of currently selected **Exception_Flow** is surrounded by a thin rectangle, so it may easily be identified.

3.6.6.2. Create a new Exception_Flow

As described in § 2, when display area shows *operation* view,  item of display area pop-up menu, or *create* menu may be used to create a new empty **Exception_Flow** list. After having performed one of these actions, a small bar appears and may be moved with mouse pointer. It can be dragged only along one of the existing **Use** relationships.

After having chosen the right location, a single click will fix the **Exception_Flow**. It may be moved later if required.

3.6.6.3. Edit Exception list

When an **Exception_Flow** is selected in drawing area, *exception list* may be edited within text input area.



This list should be related to the **Exceptions** that are actually raised by the **Operation** calls associated to current **Use** relationship. Nevertheless, there is no need to copy in *exception list* all those **Exceptions**, but simply to highlight most important ones by writing an informal label. No consistency check between *exception list* and actually raised **Exceptions** will be performed. The goal of *exception list* is to help understanding **HOOD** diagrams.


Listed **Exceptions** are also supposed to be **Provided** by destination **Module** of the **Use** relationship.

exception list may contain any string enclosed by double quote characters, or a list of valid identifiers delimited by commas. This string is used as a label for relevant **Exception_Flow** on the diagram.

3.6.6.4. *Move an Exception_Flow*

An **Exception_Flow** bar may be moved with the mouse, along carrying **Use** relationship. It cannot be moved to another **Use** relationship.

3.6.6.5. *Delete an Exception_Flow*

To delete an **Exception_Flow**, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

3.7. Types

HOOD **Types** describe **Data** structures of a **Module**. During code generation, they are directly mapped to types or subtypes (**Ada**) or typedef or class (**C** and **C++**).

To be used by a remote client **Module**, a **Type** must be declared in **Provided Interface** of server **Module**. To avoid a **Type** from being used from outside, it should be located inside **Internals** of the **Module**. Concept of “private” **Type** does not exist in **HOOD**. To create an **Ada** private **Type**, it should be located in **Provided Interface** of its **Module**, and a specific field will be filled in during detailed design phase (refer to part III).


Types are defined by a **HOOD** declarative part, located in the **Provided Interface** or the **Internals**, and a target language description (**Type** body) also located in the **Provided Interface** or the **Internals**, or both (for **Ada** private **Types**). Only declarative part may be managed during architectural design phase. **Type** bodies may be automatically produced at code generation but may also be explicitly defined during detailed design phase (refer to part III).

In **STOOD**, *graphic editor* is the best place to create, move, delete and edit declarative part of a **Type**. Some of these actions may also be performed in text editors.

3.7.1. Select an existing Type

Within drawing area, a **Type** may be selected by a single click on its name. The name of currently selected **Type** is surrounded by a thin rectangle, so it may easily be identified. Another single click will deselect it.

3.7.2. Create a new Type

As described in § 2, when display area shows *type*, `type` item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Type**. After having performed one of these actions, a string called `ttype` appears and may be moved with mouse pointer. It can be dragged to one of the following locations:

- The **Provided Interface** of parent **Module**.
- The **Provided Interface** of any child **Module**.
- The **Internals** of parent **Module** if there is no child.

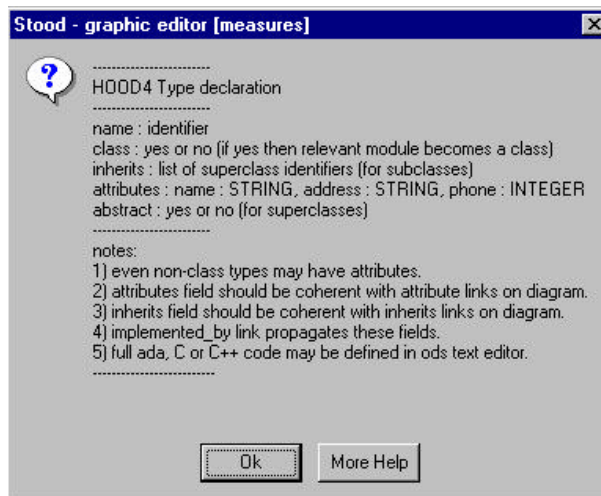
After having chosen the right location, a single click will fix the **Type**. It may be moved later if required (refer to § 3.7.4).

3.7.3. Edit Type declarative part

When a **Type** is selected in drawing area, its declarative part may be edited inside *General* tab of text input area. Declarative part of a plain **Type** contains three fields, whereas it will contain two additional fields for a **Class Type**. These five fields are described below.

```
name : rectangle
class : yes
inherits : shape.shape
attributes : TopLeft : point, BottomRight : point
abstract : no
```

Help item of *General* tab of text input area pop-up menu provides on-line information about these five fields:



3.7.3.1. *type name*

name field must be used to insert actual name of selected **Type**. Default name is `type` and should be changed into a more explicit name. If it remains unchanged, a warning message will appear while checking **HOOD** rules, as **Type** is a reserved word and should not be used for identifiers.

When giving a name to a **Type**, please take care not to use forbidden characters nor a too long string. Restrictions may come from:

- Compliancy with **HOOD** identifiers naming rules. These rules are checked when accepting text input, and a dialog box will be displayed in case of lexical or syntactic error.
- Target language (**Ada**, **C**, **C++**) naming rules. Except abnormal use of reserved words, uncompliancies to these rules will not be checked by **STOOD**, and will lead to compilation errors.
- Current file system naming rules. **Type** names are used to create files in **Application** storage area. Uncompliancies to these rules could lead to information loss at detailed design phase.

3.7.3.2. *type attributes*

attributes field may be used to list **Type** structure elements if any. **HOOD Attributes** may be mapped to **Ada** record elements, **C** struct elements or **C++** data members. This field will be left empty for **Types** without structure elements. General syntax for accepting **Type Attributes** list in **STOOD** is as follow: ([...] means 0 or 1; {...} means 0 or more).

(1) *Att_list* ::= [(Att²{, Att²)}]

(2) *Att* ::= *Att_name Att_type* [:= *Att_value*]

Att_name means **Attribute** name and is an identifier. An **Attribute** name may be used only once within a given **Attributes** list.

Att_type means **Attribute Type** and is a reference to an existing local or remote **Type**. If it is remote, dotted notation should be used. Note that, unlike **Operation Parameter Types**, no specifier may be attached to **Attribute Types**. To define pointer, reference or array **Attributes**, a new **Type** should be declared first, else full declaration will have to be deferred until coding.

Att_value means **Attribute** default value and may be a reference to an existing local **Constant**, **Data** or **Operation** with a return value, a reference to a remote **Constant** or remote **Operation** with a return value (with dotted notation), or a string delimited by double quote characters. If default value also contains double quote characters, it should be surrounded by **HOOD** text field separators: --|" a string" |--.

Types Attributes may also appear as labels carried by **Attributes** relationships on **HOOD** diagrams (refer to § 3.8).

3.7.3.3. *to be or not to be a class*

We saw above that a **HOOD Module** is a **Class Module** if its main **Type** is a **Class Type** (refer to § 3.2.2). In fact, as regards its **HOOD** declarative part, a **HOOD Type** may be of following kinds:

- an elementary or array **Type** (numeric, enumeration, string, ...), for which only *name* field is required at this level.
- a structured **Type** (record, struct), for which *name* and *attributes* fields should be filled in.
- a true Object Oriented **Class** (tagged record, class), for which *name*, *attributes*, *inherits* and *abstract* fields should contain appropriate values.

Additional *class* field should be used to change a structured **Type** (*class* : no) into a **Class** (*class* : yes) and reverse.

inherits field should be used to specify super-**Classes** if any. Multiple inheritance is allowed by **HOOD** and **C++**, but not supported by **Ada95**. Default contents of *inherits* field is empty except if **Inheritance** links were drawn on the diagram (refer to § 3.8).

abstract field should simply set to *yes* if selected **Class** is an abstract one (in **C++**, a **Class** having pure virtual member functions should be abstract). Default value of this field is *no*.

3.7.4. Move a Type

A **Type** may be moved inside a same diagram, under certain conditions. In all cases, move of a **Type** is a simple "drag and drop" action, that is performed by following sequence:


- select wished **Type** without releasing left mouse button.
- drag it while keeping left mouse button pressed.
- release left mouse button at destination location.

Allowed move actions are:

- changing the rank of selected **Type** within the list.
- moving a **Type** from or to an open **Type_Set**
- moving from **Internals** to **Provided Interface** (or reverse)
- moving to **Provided Interface** of sibling **Modules**.

Note that moves from parent to child **Provided Interfaces** (or reverse) are forbidden.

3.7.5. Delete a Type

To delete a **Type**, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

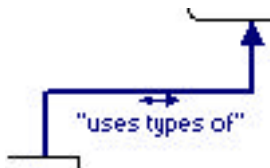


3.8. Type_Use, Attributes and Inherits relationships

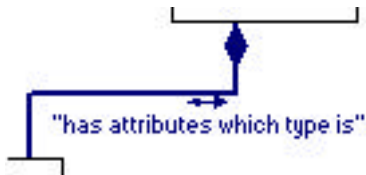
When *type* view of **HOOD** diagrams is displayed, four kinds of relationships may be shown. **Implemented_By** relationships specifies existing links between parent **Types** and local child **Types** (refer to § 3.10). All other dependencies between **Modules** that are shows on *type* view are **Type_Use**, **Attributes** or **Inheritance** relationships.

These three last relationships show structural links between two distinct **Modules**, that's why they are displayed only on *type* view of **HOOD** diagrams.

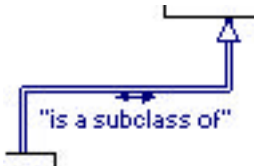
A **Type_Use** relationship between a **Module** “client” and a **Module** “server” means that “at least one **Type** of client refers at least one **Provided Type** of server”.



An **Attributes** relationship means that “at least one structured **Type** of client has at least one **Attribute** of a **Type Provided** by server”. In this case, “client” may of course be a **Class**.



An **Inheritance** relationship means that “client **Class** is a sub-**Class** of at least one **Class Type Provided** by server”. In this case “client” must be a **Class**, and “server” should be a **Class** or a **Class** library.



Like **Op_Use** relationships, **Type_Use**, **Attributes** and **Inheritance** relationships do not describe all elementary references between each **Type**, that could quickly become unreadable. On the contrary, they provide a high level description of structural dependencies between **Modules**. They act as the same level as a `with` clause in **Ada** or a `#include` in **C/C++**. More detailed dependencies can be analysed with **HOOD Required Interface** descriptions, and **STOOD** *cross references tables* (refer to part III: detailed design).



In accordance with **HOOD** visibility rules, a **Type** relationship may only be drawn from a local **Module** to a “sibling” **Module**, an “uncle” **Module** or an **Environment Module**.

Type_Use, **Attributes** and **Inheritance** relationships may optionally carry a label on **HOOD** diagram, which should be consistent with *attributes* and *inherits* fields of “client” **Type** (refer to § 3.7.3). In addition, *inheritance tree* provide a synthetic view of all **Inheritance** relationships that have been defined in the **Application** (refer to § 5).

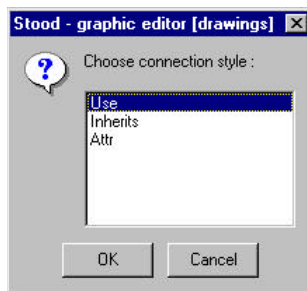
3.8.1. Select an existing Type relationship

Within drawing area, a **Type** relationship may be selected by a single click on one of its segments. Dragging handles are then shown at the ends of each segment, so it may easily be identified.



3.8.2. Create a new Type_Use relationship


As described in § 2, when display area shows *type* view,  item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Type_Use** relationship. After having performed one of these actions, source **Module** should be selected first by a single click, then destination **Module** should be selected by another single click. Like **Op_Use** relationships, destination of a **Type_Use** may be a **uncle** or **Environment** box outside parent borders.

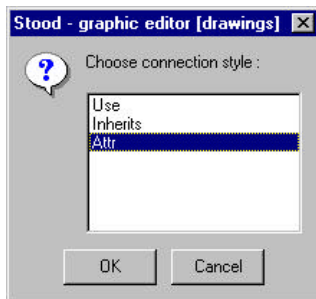
An alternate way to create a **Type_Use** relationship is to select an existing **Attributes** or **Inheritance** relationship and double click on it or activate **Style** item of *edit* menu, and use displayed dialog box to set connection style to: Use.



3.8.3. Create a new Attributes relationship



As described in § 2, when display area shows *type* view,  item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Attributes** relationship. After having performed one of these actions, source **Module** should be selected first by a single click, then destination **Module** should be selected by another single click. Like **Op_Use** relationships, destination **Module** of **Attributes** relationships may be an uncle or **Environment** box outside parent borders.

An alternate way to create a Attributes relationship is to select an existing **Type_Use** or **Inheritance** relationship and double click on it or activate  item of *edit* menu, and use displayed dialog box to set connection style to: `Attr`.

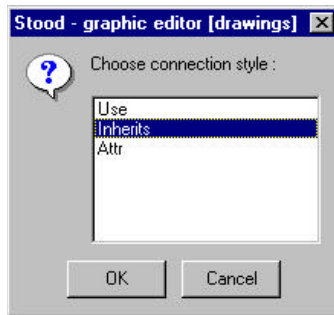


Note that in some cases, **HOOD Attributes** relationship may be a design implementation of **Class** aggregation or composition that was identified during **OO** Analysis phase. If used **OO** Analysis formalism was **OMT** or **UML**, this relationship was also represented by a line terminated by a diamond, but at the opposite side. This could be a source of misunderstanding.

3.8.4. Create a new Inheritance relationship

As described in § 2, when display area shows *type* view,  item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Inheritance** relationship. After having performed one of these actions, source **Class** should be selected first by a single click, then destination **Class** or **Class** library should be selected by another single click. Like **Op_Use** relationships, destination **Module** of **Inheritance** relationships may be a uncle or **Environment** box outside parent borders.

An alternate way to create an **Inheritance** relationship is to select an existing **Attributes** or **Type_Use** relationship and double click on it or activate **Style** item of *edit* menu, and use displayed dialog box to set connection style to: *Inherits*.




inherits field of declarative part of source **Class** is generally automatically updated when **Inheritance** relationship is created. In case of **Class** libraries, first listed **Class** will be used by default, and actual super-**Class** may have to be specified manually.

3.8.5. Move a Type relationship

Any **Type** relationship may be moved by dragging one of its handles with the mouse. If additional handles are required, select one of existing handles, and use **Lenghten** item of *edit* menu to create new segments. If useless segments need to be removed, use **Shorten** item of *edit* menu (refer to § 2.6.4).

3.8.6. Delete a Type relationship

To delete any **Type** relationship, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.

Deletion of any **Type** relationship also deletes related labels.


3.8.7. Type relationships labels

3.8.7.1. Select an existing label



Within drawing area, a **Type** relationship label may be selected by a single click on its arrow. The arrow of currently selected label is surrounded by a thin rectangle, so it may easily be identified.

3.8.7.2. Create a new label

As described in § 2, when display area shows *type* view,  item of drawing area pop-up menu or *create* menu may be used to create a new empty label on any **Type** relationship. After having performed one of these actions, a small arrow appears and may be moved with mouse pointer. It can be dragged only along one of the existing **Type** relationships.

After having chosen the right location, a single click will fix the label. It may be moved later if required.

3.8.7.3. Edit label string

When a label is selected in drawing area, *label* field may be edited within *General* tab of text input area.




Labels may have different meaning regarding the style of **Type** relationship. They have a pure informational role, but they should help understanding the diagram. If destination **Module** is a **Class**, there is no ambiguity on which **Type** is used, but if it provides several **Types**, a label could be used to specify which **Types** are actually used. for **Attributes** relationship, if origin **Module** is a **Class**, a label could be used to specify which **Attributes** are of remote **Type**.

label field may contain any string enclosed by double quote characters, or a list of valid identifiers delimited by commas. This string is displayed on the diagram. When the string needs to contain double quote characters, **HOOD** text delimiters -- | | -- should be used to en close the overall string.

3.8.7.4. *Move a label*

A label arrow may be moved with the mouse, along carrying **Type** relationship. It cannot be moved to another **Type** relationship.

3.8.7.5. *Delete a label*

To delete a label, it should be selected first, and **D**elete item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **U**ndo menu item.



3.9. Constants and Data

HOOD Constants (respectively **Data**) describe instances of **HOOD Types** (target language predefined **Types** or applicative **Types**), that are global to current **Module** (if defined in the **Internals**), or global to the parent of current **Module** (if defined in the **Provided Interface**).

To comply with information hiding principles, variable **Data** should only be located inside the **Internals** of a **Terminal Module**. On the contrary, due to their read-only property, **Constants** may be located either in the **Provided Interface**, either in the **Internals** of a **Module**.

Other instances of **Type** that are not global to a given **Module**, are of following kinds:

- **Attributes**, which scope is related **Type**, sub-Classes and all relevant instances.
- **Parameters**, which scope is related **Operation** declarative part and body.
- Local variables and **Constants**, which scope is related block of code (**Operation Control Structure** or **Object Control Structure**).

At architectural design phase, **Constants** and **Data** are fully defined by their name. More information will have to be provided during detailed design phase (refer to part III).


3.9.1. Select existing Constant or Data

Within drawing area, a **Constant** (respectively: **Data**) may be selected by a single click on its name. The name of currently selected **Constant** (resp. **Data**) is surrounded by a thin rectangle, so it may easily be identified. Another single click will deselect it.

3.9.2. Create new Constant or Data

As described in § 2, when display area shows *constant* (respectively *data*) view, **constant** (resp. **data**) item of display area pop-up menu, or of *create* menu,



or  button may be used to create a new **Constant** (resp. **Data**). After having performed one of these actions, a string called `constant` (resp. `data`) appears and may be moved with mouse pointer. It can be dragged to one of the following locations:

- The **Provided Interface** of parent **Module** (except for **Data**).
- The **Provided Interface** of any child **Module** (except for **Data**).
- The **Internals** of parent **Module** if there is no child.

After having chosen the right location, a single click will fix the **Constant** (resp. **Data**). It may be moved later if required (refer to § 3.9.4).

3.9.3. Edit Constant or Data name

When a **Constant** (resp. **Data**) is selected in drawing area, its name may be edited inside *General* tab of text input area.



constant name (resp. *data name*) field must be used to insert actual name of selected **Constant** (resp. **Data**). Default name is constant (resp. data) and should be changed into a more explicit name. If it remains unchanged, a warning message will appear while checking HOOD rules, as **Constant** (resp. **Data**) is a reserved word and should not be used for identifiers.

When giving a name to a **Constant** (resp. **Data**), please take care not to use forbidden characters nor a too long string. Restrictions may come from:

- Compliancy with **HOOD** identifiers naming rules. These rules are checked when accepting text input, and a dialog box will be displayed in case of lexical or syntactic error.
- Target language (**Ada**, **C**, **C++**) naming rules. Except abnormal use of reserved words, uncompliances to these rules will not be checked by **STOOD**, and will lead to compilation errors.
- Current file system naming rules. **Constant** (resp. **Data**) names are used to create files in **Application** storage area. Uncompliances to these rules could lead to information loss at detailed design phase.

3.9.4. Move Constant or Data

A **Constant** (resp. **Data**) may be moved inside a same diagram, under certain conditions. In all cases, move of a **Constant** (resp. **Data**) is a simple "drag and drop" action, that is following sequence:


- select wished **Constant** (resp. **Data**) without releasing mouse button.
- drag it while keeping left mouse button pressed.
- release left mouse button at destination location.

Allowed move actions are:

- changing the rank of selected **Constant** (resp. **Data**) within the list.
- moving a **Constant** (resp. **Data**) from or to an open **Set**
- moving from **Internals** to **Provided Interface** (or reverse)
- moving to **Provided Interface** of sibling **Modules**.

Note that moves from parent to child **Provided Interfaces** (or reverse) are forbidden.

3.9.5. Delete Constant or Data

To delete a **Constant** (resp. **Data**), it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.



3.10. Include and Implemented_By relationships

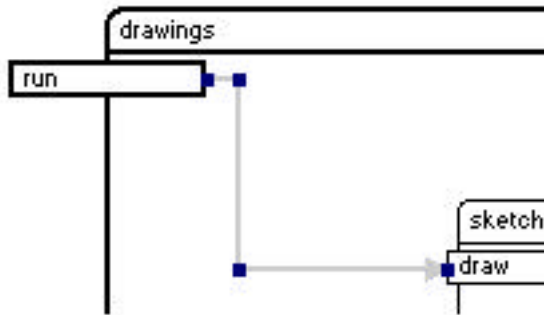
During architectural design phase, **Non Terminal Modules** are broken down into lower level **Modules**. The link between a **Non Terminal Module** (the parent) and its child **Modules** is called the **Include** relationship. Combination of all **Include** relationships constitute **HOOD Design Tree** of current **Application**.

While designing, **Include** relationships are indirectly controlled by creating, moving and deleting **Modules**, and do not need other formalized actions.

An **Include** relationship is a one to many link between **Modules**. It does not provide any information on the way each **Component Provided** by a parent **Module** is linked to related entities of child **Modules**. **Implemented_By** relationships should be defined to perform this task.



An **Implemented_By** relationship is a one to one link between one parent **Component** and one related child **Component**. It should be defined for each **Operation**, **Operation_Set**, **Type**, **Constant** and **Exception Provided** by a **Non Terminal Module**. As an extension to **HOOD4**, **STOOD** also supports **Implemented_By** relationships for **Type_Sets**, **Constant_Sets** and **Exception_Sets**.

3.10.1. Select an existing Implemented_By relationship



Within drawing area, an **Implemented_By** relationship may be selected by a single click on one of its segments. Dragging handles are then shown at the ends of each segment, so it may easily be identified.

3.10.2. Create a new Implemented_By relationship

As described in § 2,  item of drawing area pop-up menu, or of *create* menu, or  button may be used to create a new **Implemented_By** relationship. After having performed one of these actions, one parent **Provided Component** should be selected by a single click, and one child **Provided Component** should be selected by another single click.

Note that actual effect of this creation will depend on which **Component** has been selected first:

- If first selected **Component** is parent's one, then a top-down update process will be performed.
- If first selected **Component** is child's one, then a bottom-up update process will be performed.

These update processes are detailed below for each kind of **Component**.

3.10.2.1. Create an Implemented_By for Operations

With a top-down create action, child's **Operation** declarative part is replaced by parent's one, but **Operation** names and **Parameters Type** are kept unchanged.

With a bottom-up create action, parent's declarative part is replaced by child's one, but **Operation** names and **Parameters Type** are also kept unchanged.

After creation, any change in declarative part of any **Operations** linked along a chain of **Implemented_By** relationships, will be automatically propagated to the others, except for changes regarding the name of **Operations** and the **Type** of **Parameters**.

3.10.2.2. Create an Implemented_By for Operation_Sets

With a top-down create action, child's **Operation_Set** contents is replaced by parent's one. With a bottom-up create action, parent's **Operation_Set** contents is replaced by child's one.

These actions will automatically create all required **Operations** with their **Parameters** list. After creation, no change will be automatically propagated.

Although **HOOD** design process is mainly top-down, it may be profitable to define only empty **Operation_Sets** at high level in **HOOD** hierarchy, and only complete full **Operations** declarative parts when reaching **Terminal Modules**. Then, creating **Implementing_By** in bottom-up mode, will automatically update higher levels interfaces.

3.10.2.3. Create an Implemented_By for Types

With a top-down create action, child's **Type** declarative part is replaced by parent's one, but **Type** names are kept unchanged.

With a bottom-up create action, parent's **Type** declarative part is replaced by child's one, but **Type** names are also kept unchanged.

Update of *inherits* field may behave differently depending on whether super-**Class** is also **Implemented_By** or not, and according to any existing **Inheritance** relationships.

After creation, any change in declarative part of any **Type** linked along a chain of **Implemented_By** relationships, will be automatically propagated to the others, except for changes regarding the name of **Types**, and *inherits* fields.


3.10.2.4. Create an Implemented_By for Constants or Exceptions

No specific action is performed while drawing top-down or bottom-up **Implemented_By** relationships for **Constants** or **Exceptions**.

3.10.3. Move an Implemented_By relationship

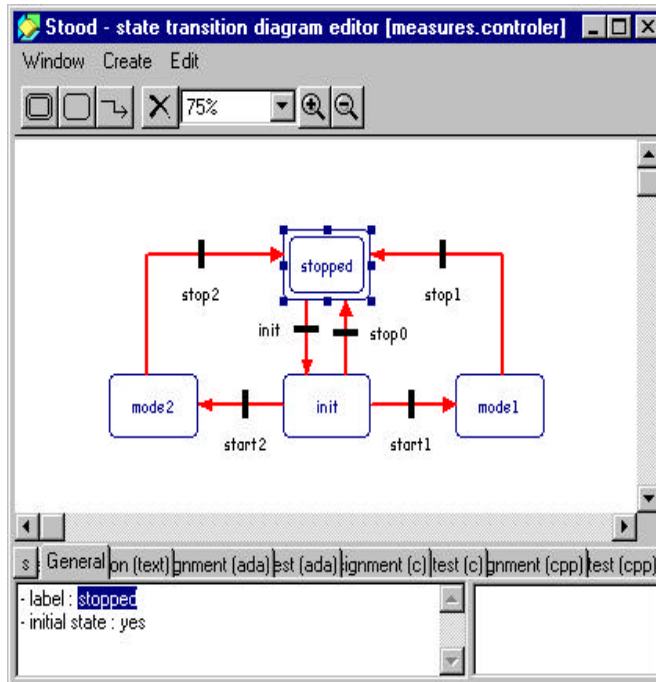
An **Implemented_By** relationship may be moved by dragging one of its handles with the mouse. If additional handles are required, select one of existing handles, and use **Lenghten** item of *edit* menu to create new segments. If useless segments need to be removed, use **Shorten** item of *edit* menu (refer to § 2.6.4).

3.10.4. Delete an Implemented_By relationship

To delete an **Implemented_By** relationship, it should be selected first, and **Delete** item of *edit* menu or  button should be used. There is no dialog box to confirm action, but it can be cancelled with **Undo** menu item.



4. State-Transition Diagrams editor



4.1 Drawing area.....	p.138
4.2 Text input area.....	p.139
4.3 Window menu.....	p.142
4.4 Create menu.....	p.143
4.5 Edit menu.....	p.145

As explained in § 3.3.3.3, **HOOD Modules** may provide **Operations** that are **Constrained** by **State**. This means that actual use of specified functional service depends on current internal **State** of server **Module**.

During architectural design phase, **States** of a **Module** are defined by drawing a **State-Transition Diagram (STD)**. Additional information will be required during detailed design phase (refer to part III).

HOOD4 State-Transition Diagrams are based on a simplified use of finite state automata standard formalisms. The aim is to explain how **Provided Operations** calls may be constrained, and is not to describe fully internal behaviour of a **Module**.

State-Transition Diagrams are composed of:

- **States**, simply identified by their name at this level, and which will be linked later to state variables values (**Data, Attributes, ...**).
- **Transitions**, which are triggered by an **Event**, and which may imply a change of state. In **HOOD4**, Events must be directly linked to **Provided Operations** invocations. Such **Provided Operations** are thus **Constrained** by **State**.

Note that it is quite easy to map **Transition Events** to **Provided Operations**, and this mapping is performed while drawing the **STD**, but it may be much more difficult to map **States** to state variables values, and this will be only feasible during detailed design phase (please, refer to part III).

Concerning this point, it should be noted that designing a **STD** for a **Terminal** or a **Non Terminal Module** have not the same impact on **HOOD** design. A **STD** for a **Terminal Module** may be fully implemented at detailed design phase and be then be used to automatically produce code. At the opposite, A **STD** for a **Non Terminal Module** will remain purely informative, and may simply help understanding **Application** behaviour at high level, but will not be used to produce any code (state variables for **Non Terminal Modules** are undefined).

A **STD** is thus attached to a **HOOD Module**. To launch a **STD** editor, one of the following actions should be performed:

- select *state transition diagram editor* item of *editors* menu of *main editor*: this will open a **STD** on **Root Module**.
- select *state-transition diagram* item of pop-up menu of *modules* area of *main editor*: this will open a **STD** on selected **Module**.
- select *state-transition diagram* item of *edit* menu of *graphic editor*, or press related shortcut button: this will open a **STD** on selected **Module**.

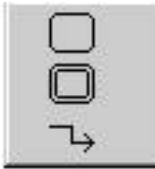
Like **HOOD** diagrams editor, **STD** editor is composed of a drawing area, a text input area, a menu bar, and a button bar. For general information regarding the use of **STD** editor (graphical or textual edition), please refer to chapters describing these functions for *graphic editor* (§ 2).

Following chapters describe only specific functions provided by **STD** editor.

4.1. Drawing area

Drawing area of **State-Transition Diagram** editor is very similar to drawing area of **HOOD** diagrams graphic editor. Please refer to § 2.1 for general purpose information about diagram scale and visible part (§ 2.2.1), selecting, moving and resizing graphical entities (§ 2.2.2).

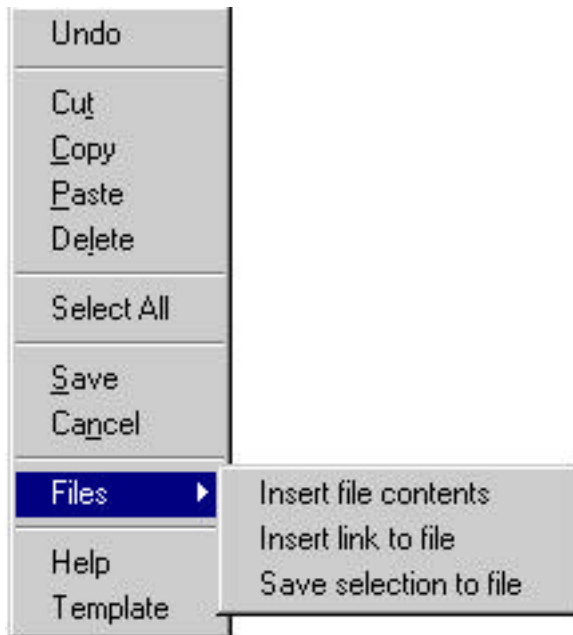
STD editor drawing area also provides a pop-up menu to ease creating new graphical entities:



Please refer to § 4.4 below for further details.

4.2. Text input area

Text input area of **STD** editor behaves the same as text input area of **HOOD** diagrams graphical editor. In particular, please refer to § 2.2 to get information on using text input area pop-up menu:



Anyway, contents of text input area is different and refer to **STD** graphical entities:

4.2.1. Editing a State

When a **State** is selected in drawing area, its properties may be changed within *General* tab of text input area:



A screenshot of a text input area with a scroll bar on the right. The text inside the area is:

```
label : stopped  
initial state : yes
```

label field refer to **State** name. **State** names are **HOOD** design identifiers, but they may not be a coding identifier (except in simplest cases where **States** are implemented by an enumeration type).

initial state is a simple flag to be set to *yes* or *no*. A unique initial state should be defined in a **STD**.

4.2.2. Editing a Transition

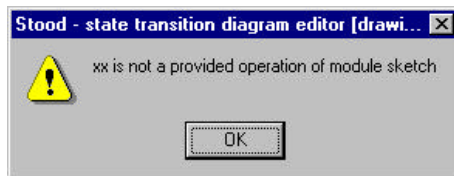
When a **Transition** is selected in drawing area, its properties may be changed within *General* tab of text input area:



label : start
event : draw

label field refer to **Transition** name. **Transition** names are **HOOD** design identifiers, but they may not be a coding identifier (except in simplest cases where **Transition** names are the same as their triggering **Operation**).

event field identifies **Provided Operation** that triggers selected **Transition**. This field should contain the name of an existing **Provided Operation** of current **Module**, or be left empty, else following warning message will be displayed:



Note that a same **Provided Operation** may trigger several **Transitions**. Actual determinism of the automaton is not controlled automatically. During detailed design phase, additional information will have to be provided (transition condition) in the case of several **Transitions** from the same **State** and triggered by the same **Operation**.

4.3. Window menu of STD editor

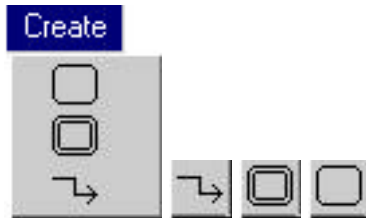


Window menu of **STD** editor has is similar behaviour as the same menu in **HOOD** Diagrams graphic editor. Please refer to § 2.3 for further details.


To close a **STD**, use the *Quit* command of *Window* menu. A **STD** is always attached to a parent editor (generally a **HOOD** diagram editor). When parent editor is closed, a warning message asks for confirmation to also close all attached **STD** editors.




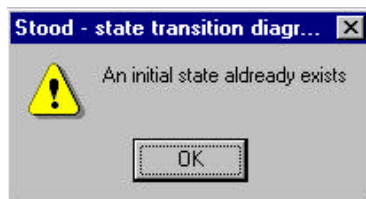
4.4. Create menu of STD editor




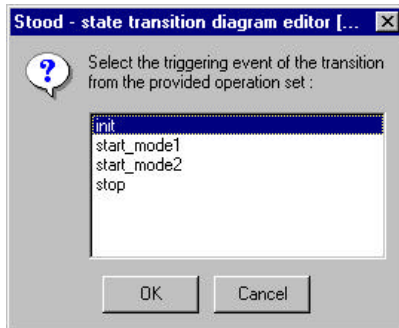
This menu, which is redundant with drawing area pop-up menu and buttons, should be used to create graphical entities on a **STD**.

 : create a new **State**. A grey rectangle is shown at mouse pointer location, and may be dragged to chosen place. A single mouse button click will actually create the **State** box. New **State** is given a default name that may be changed in *label* field within text input area. *initial state* field of text input area is set to *no*.

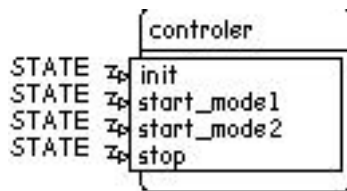
 : same as above, but automatically create an initial **State**. *initial state* field of text input area is set to *yes*. Internal concurrency is not managed at present within **STD** designed with **STOOD**. There should be only one initial **State** in a given **STD**. While trying to create a second initial **State**, a warning message is displayed in a dialog box:



 : create a new **Transition** between two existing **States**. Origin and destination **States** should be designated by mouse clicks. A dialog box ask then to identify which **Provided Operation** call will trigger the **Transition**. If cancel button is selected, event field of text input area will be left empty.



If selected **Provided Operation** is not constrained by **STATE** yet, this **Constraints** label will be automatically added to its properties (refer to § 3.3.3.3):



4.5. Edit menu of STD editor



copy : copy selected **State** into an internal buffer. No multiple selection is presently supported by **STOOD**.

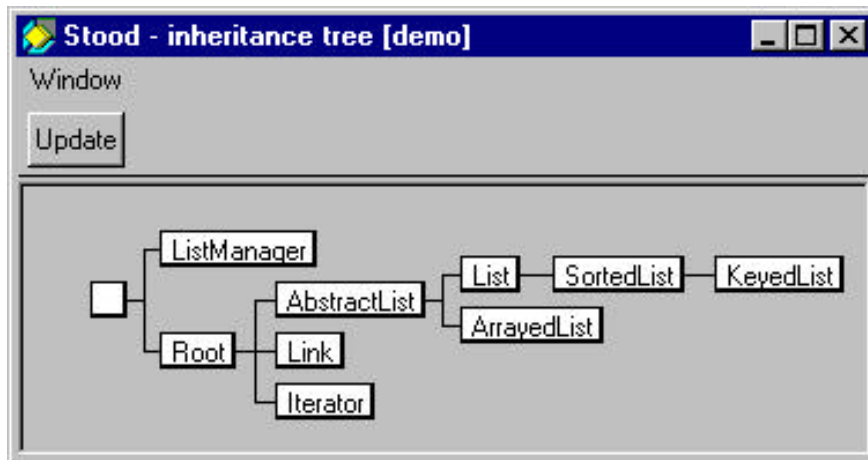
paste : create a new **State** with a copy of current contents of internal buffer.

delete or  button : delete selected **State** or **Transition**. This action can be cancelled with *undo* menu item.

undo : cancel last *delete* action.

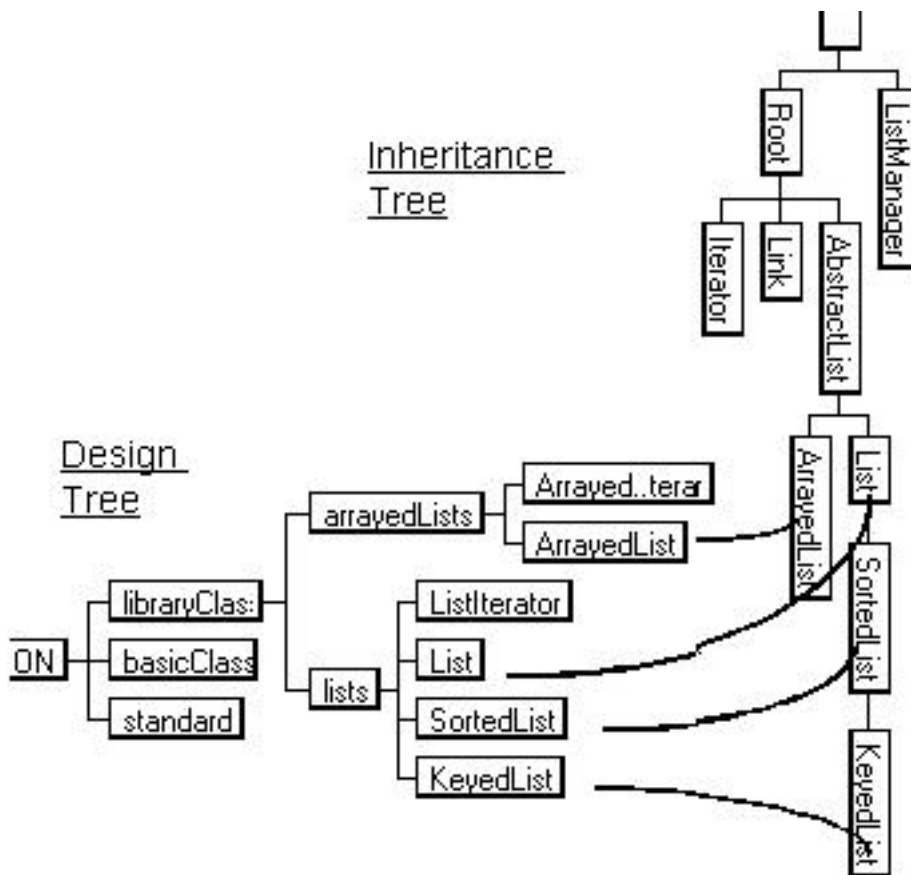


5. Inheritance Tree



Inheritance Tree may be launched from *editors* menu of *main editor*.

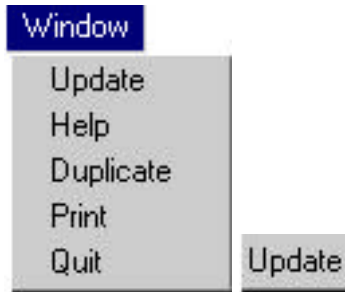
STOOD Inheritance Tree provides an additional observation point for the **Application**. With *type* view of **HOOD** diagrams, **Inheritance** relationships may be defined (refer to § 3.8.4), and super-**Classes** may be identified precisely within text input area (refer to § 3.7.3.3). These relationships describe a graph of dependencies between **HOOD Modules**, which is transverse to **HOOD** breakdown **Design Tree**.



Inheritance Tree shows **Inheritance** links that are defined inside current **Application**, and also those regarding **Provided Classes** of **Environment Modules**. That is why an **Inheritance Tree** may show **Modules** that are not part of current **Application**

In case of multiple **Inheritance**, only one link is displayed in order to draw a simple tree and not a full graph. A **Class** which inherits from several super-**Classes** is shown by a grayed box to indicate that part of the information has been hidden.

Inheritance Tree is not an editor, and nothing may be modified at this level. Only a *window* menu is provided to control the display.



Update menu item or button may be used to redraw Inheritance Tree. It may not be always be updated automatically when Inheritance information is changed within **HOOD** diagrams editor.

The other menu items are similar to those of graphic editor (refer to § 2.3). in particular, Inheritance Tree may be printed:

Print : direct print of current **HOOD** Inheritance Tree on standard printer. For **Windows**, used printer is the default one. For **UNIX**, a PostScript file is created in launching directory, and direct printing is performed if a printer queue has first been defined in `fastprint.sh` file of `internalTools` configuration directory.

