CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

The System for Knowledge-based Layout (SKY) Project

by

Masaru Tamaki
and
Dr. John Kunz

TECHNICAL REPORT
Number 76

October, 1992

Stanford University

The System for Knowledge-based Layout (SKY) Project

Masaru Tamaki, Visiting Fellow from TAISEI Corp.

Dr. John Kunz, Senior research associate at CIFE,
Stanford University

1992.10

SUMMARY
CIFE TECHNICAL REPORT #76

Header:
Title: The System for Knowledge-based Layout (SKY) Project
Authors: Masaru Tamaki
Dr. John Kunz
Publication Date: October 1992
Funding Sources: Masaru Tamaki was supported by Taisei Corp. Japan.

1. Abstract:

This report is about the system for knowledge-based layout which focuses on industrial facilities' conceptual layout. Layout is very important in industrial facilities such as a factory, a warehouse, a distribution center and so on.

Layout used to be made usually according to the designers' experiences. Experience may produce preconception that can obstruct the designers to reach the best solution. This system explores every possibility to look for the best solution under very basic constraint such as site conditions and facility requirements.

This system consists of three parts. They are adjacency definition component, layout generation component and evaluation component. The adjacency component defines departmental adjacencies according to their requirements. The layout generation component generates all possible layout by search technique with pruning by some rules. The evaluation component evaluates alternative layouts according to the preference factors entered by the user.

This report describes the background of the system and the details of the system which includes the algorithm of reasoning, knowledge representation, computer source code and so on.

2. Subject:

This system addresses the conceptual layout of industrial facilities. The knowledge to make a conceptual layout is implemented in the computer using some basic AI techniques. By using this system, the users can explore every possibility without any preconception and make clear the requirements for the layout. The users can easily customize the system according to their own knowledge.

3. Objectives/Benefits:

The goal of this study is to develop a prototype layout system which has a potential for the real use.

4. Methodology:

The system is developed by an object-oriented programming environment called ProKappa. A Macintosh computer is also used to make the from/to chart which describes an information about a material flow in the facility. The data is transformed by a network.

5. Results:

The output of this study is an application software by ProKappa named SKY and a video which shows the concept and the basic operation of the system.

6. Research Status:

The system is successfully finished as a prototype system. There remained several problems for the real use. The main problem is the execution time. In order to reduce the execution time, the more powerful machine is desired. A new search pruning knowledge is also desired to reduce the execution time. The system is carefully designed to allow the users to add a new knowledge.

This system is so flexible that there are many potential future developments for it such as integration with facility management, addition of 3D features, addition of dimensional information, suggestion of material handling routes and handling methods, and so on.

The System for Knowledge-based Layout
SKY Project

Contents

I. Goal and Scope of the research

A. Final goal of the research

The final goal of the research is to utilize computer technology to make rational decisions in each step of the facility engineering process. Facility engineering includes the following components.

1 Programming:

In this phase the following issues are determined.

- What to build, What to invest (Decision making about investment)
- Where to build (Site selection)
- How large it should be (Market research, Future prediction)

2 Planning:

In this phase conceptual through detailed design is performed, including;

- Layout
- Architectural Planning (Structure, Utilities, Materials)
- Machine, equipment planning (Handling system, Production system)

3 Project Execution:

In this phase the project is executed in practice and the following things are involved.

- Project Management
- Construction
- Procurement

4 Maintenance:

The last stage of facility engineering is the maintenance.

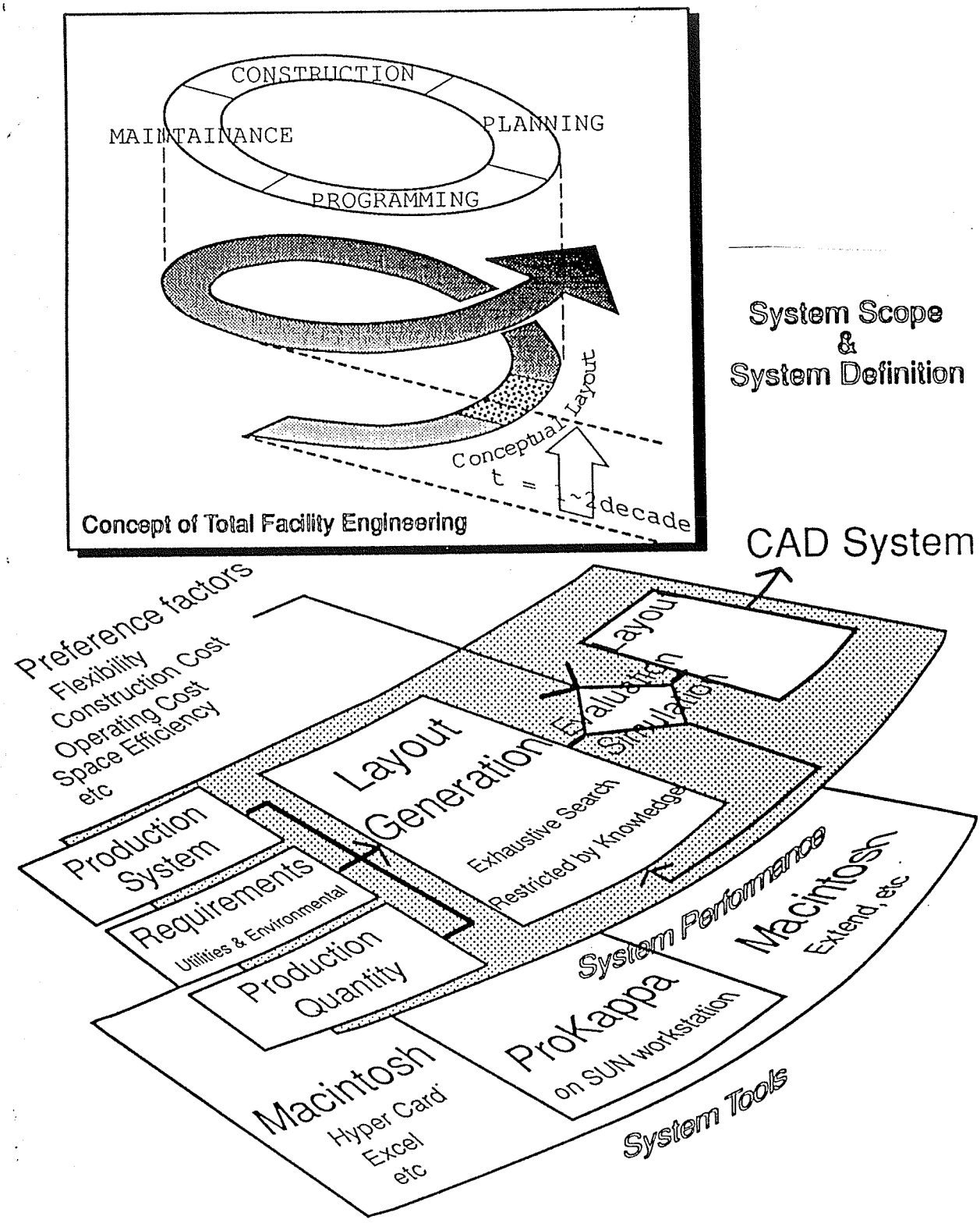
- Facility management
- (the data acquainted in this process should be input data of Programming process)

Computer technology can be applied to deal with information to make decisions rationally. There is much room for computer application in this process.

This research demonstrates a system named the System for Knowledge-Based Layout (SKY) as an example of computer usage for layout analysis and of industrial factories and warehouses.

Fig. 1-1 shows the concept of the total facility engineering and the scope of this research.

Fig. 1-1


CONSTRUCTION
MAINTAINANCE
PLANNING
PROGRAMMING
Conceptual Layout
t = ~ decade
Concept of Total Facility Engineering
System Scope
&
System Definition
CAD System
Preference factors
Flexibility
Construction Cost
Operating Cost
Space Efficiency
etc
Layout
Evaluation
Simulation
Layout Generation
Exhaustive Search
Restricted by Knowledge
Production System
Requirements
Utilities & Environmental
Production Quantity
System Performance
Macintosh
Extend, etc.
ProKappa
on SUN workstation
Macintosh
Hyper Card
Excel
etc
System Tools

B. Scope of the research and importance of layout study

The SKY project addresses the conceptual layout of industrial facilities such as a factory or warehouse or distribution center in which engineering issues are more important than aesthetic issues. This project is related to the work of Chinowsky[1] which considers architectural layout using "bubble diagrams".

In these facilities, layout study is very important. There are several reasons for this.

First a poor layout is a source of constant loss to the company because of additional material handling, poor stock locations and other inefficiencies. These losses accumulate day after day, month after month, year after year. The cost of changing the layout once it has been made is too great. So the losses continue, as a constant drain on the business. Profits which could have been secured at little or no extra cost when the layout was originally made are lost forever.

Second, a poor layout causes danger such as cross traffic of workers and machinery.

Third, bad conditions caused by a poor layout reduces workers' motivation and effectiveness.

These losses and dangers can be minimized by making a good layout.

SKY focuses on conceptual layout, the first step in the layout process. Effective and clear conceptual layout makes the later detailed layout steps relatively quick and simple. SKY also makes layout requirements explicit and easy to change. Since layout is fast, clients can do repeated what-if studies to clarify their requirements and to do much higher quality layouts than if they can try only a small number of layouts. SKY also allows evaluates layout alternatives.

SKY is an attempt to rationalize this conceptual layout by using computer technology such as object oriented programming.

[1]Paul Scott Chinowsky "The CAADIE Project", CIFE Technical report #54, May 1991

II. Background of the research and problem definition

A. Conventional way of designers' thinking

In the conceptual planning phase, designers consider mainly two things:
"Site condition" and "facility requirements".

Site condition includes the following issues as fig. 2-1 shows.
Property lines,
Street locations,
Compass direction,
and Landscape features including existing vegetation and so on.

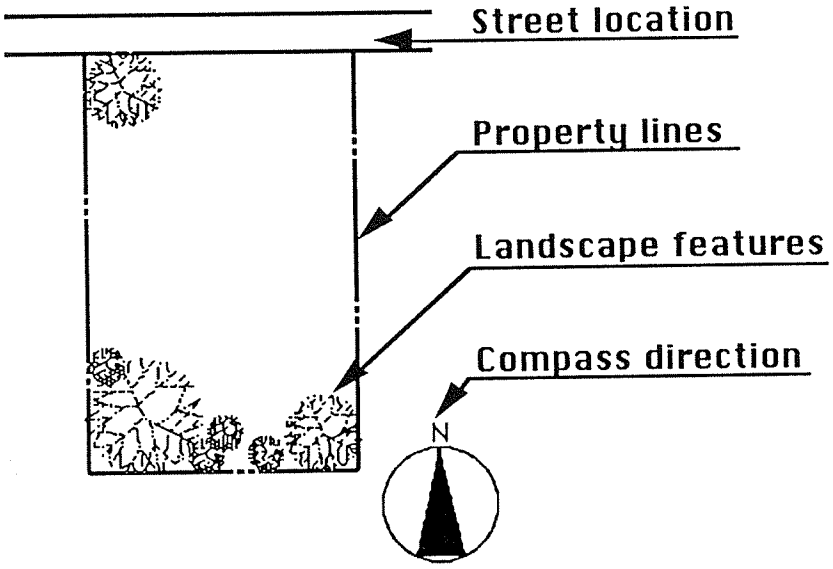


Fig.2-1

Attributes of the site considered in facility design

Factories are organized by departments or major functions. Facility requirements designers must also know:

Number of departments and their names;
These departments' functions and special requirements; (such as needsAccess; useFire)
Their size;
Their shape; (Block type or Linear type)
The flow of materials from department to department
Location for delivering and shipping of product.

And the relationships among departments (such as preferred adjacencies and preferred negative adjacencies ;)

There are several conventional ways to express these departmental requirements (Fig. 2-2).

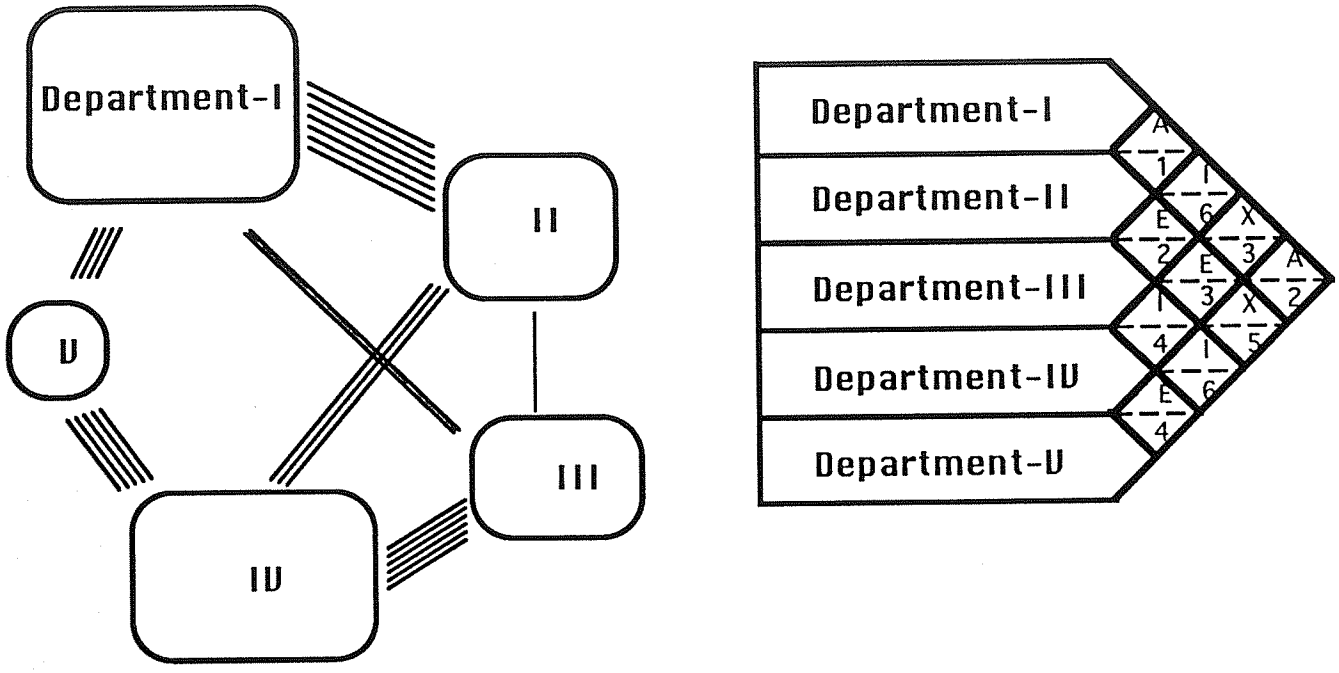


Fig.2-2

Examples of departmental requirement's description

Fig. 2-3 summarizes the department requirements in SKY.

Department name	Material Flow					Area	Special usage	Requirements
	to D1	to D2	to D3	to D4	to D5	(sqm)		
D1		98	3	12	8	150	Receiving	needs access
D2	4		88	21	12	50	use fire	
D3	11	15		56	11	100	human space	
D4	2	5	8		87	200	store Explosive	
D5	45	12	23	22		150	Shipping	needs access

Fig.2-3

Department attributes used in SKY

First column shows the names of departments. In this case there are five department to be laid out.

Second column shows the relative amount of material flow from one department to another.

Third column shows the department size.

Forth column shows the department function, and fifth column shows special requirements to enable each department to perform its function. For example, a shipping department needs access to a street to perform its function.

Considering these conditions and requirements, designers generate several alternative layouts, i.e., arrangements of departments.

After generating layout alternatives, designers evaluate alternative plans to select the best one.

To evaluate the alternatives, they consider if the alternative fits the requirements such as adjacency requirements, view requirements, access requirements and so on.

After selection of the departmental layout, designers develop a detailed floor plan.

So the process of human layout planning is as Fig. 2-4 shows.

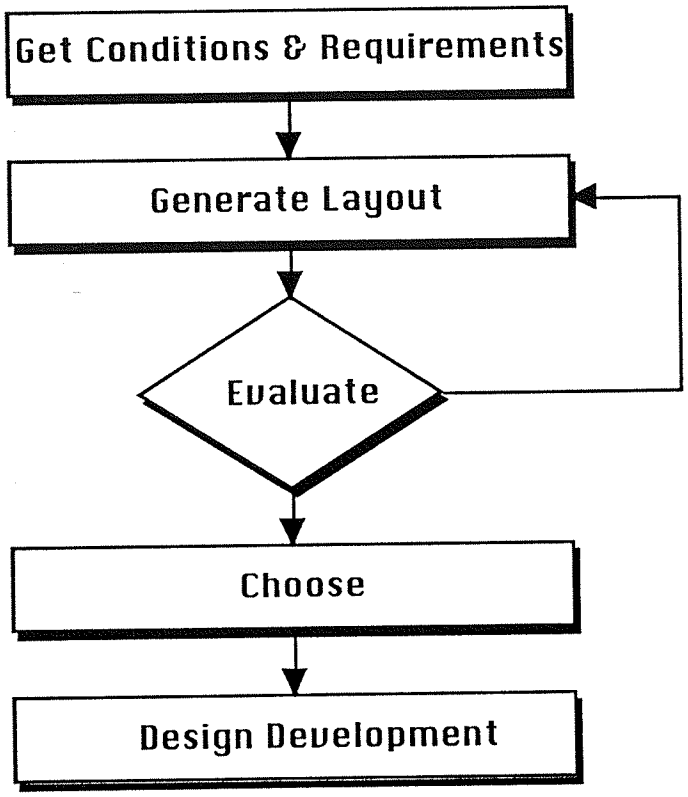


Fig.2-4

Human layout planning process

SKY attempts to transfer this human layout-planning process to the computer.

III. System configuration

A. Process analysis to get From/To Chart by HyperCard

Material flow is one of big issue which affects the layout of industrial facilities. A from/to chart (Fig. 3-1) expresses the material flow; that is, each cell of the matrix shows the number of material handling trips per unit time from each department to every other department. The numbers suggest the cost or difficulty of material handling, and they imply the required adjacencies among departments. The bigger the number is, the harder it is to carry materials between two departments. Department pairs with the largest numbers, e.g. D1 and D2, should be allocated adjacent to each other to facilitate material flow.

Department name	Material Flow				
	to D1	to D2	to D3	to D4	to D5
D1		98	3	12	8
D2	4		88	21	12
D3	11	15		56	11
D4	2	5	8		87
D5	45	12	23	22	

Fig. 3-1

Example of a from/to chart

Each number in this matrix must be normalized by the same standard. As Fig. 3-2 ~ Fig. 3-4 show, in this process analysis system we can set the type of material, handling method and terms to normalize the difficulty of handling different materials by different methods.

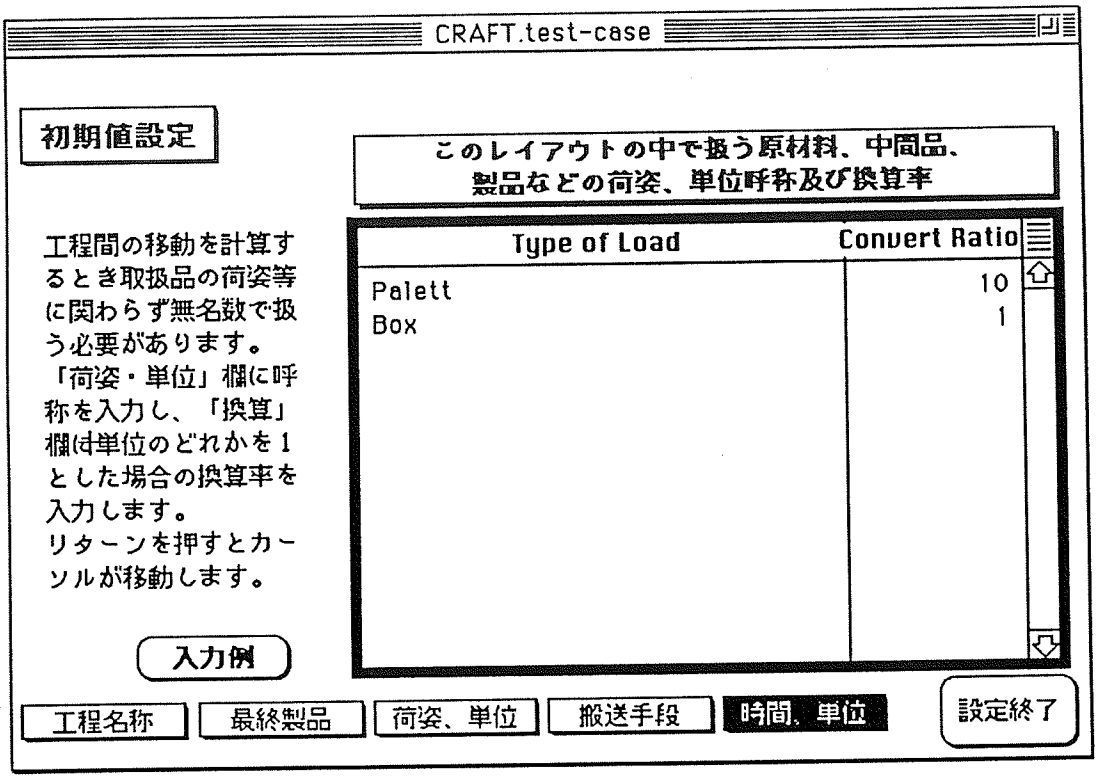


Fig.3-2 : Normalize load type

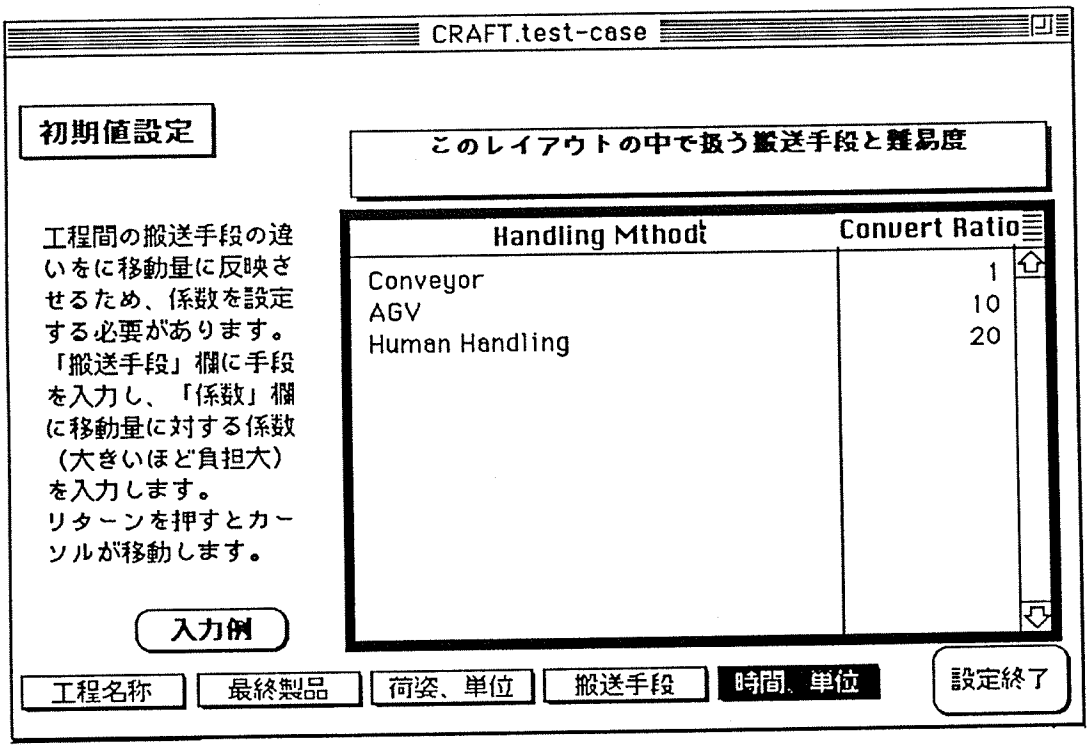


Fig.3-3 : Normalize handling method

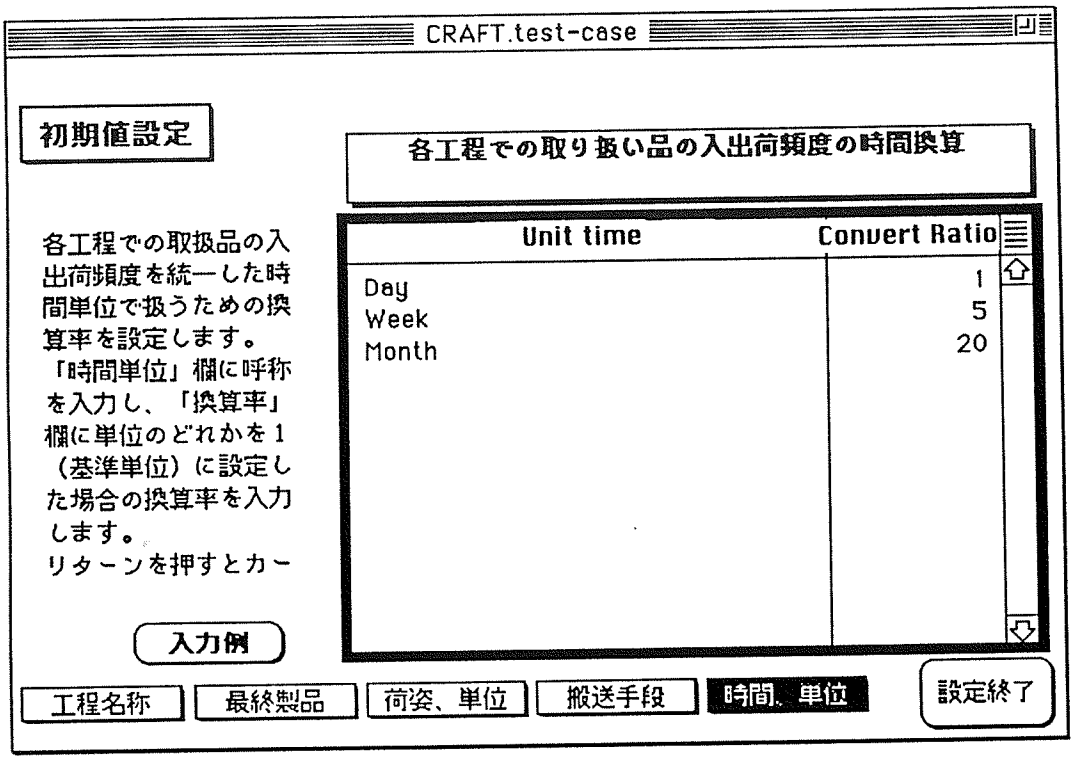


Fig.3-4

Normalization of unit term

For example, fig. 3-2 says that it is as 10 times more difficult to carry one pallet than to carry one box. Fig. 3-3 says human handling should have higher weight rather than any other method.

These weighting factors are used to calculate the numbers of a From/To Chart. Fig. 3-5 shows how the numbers are normalized.

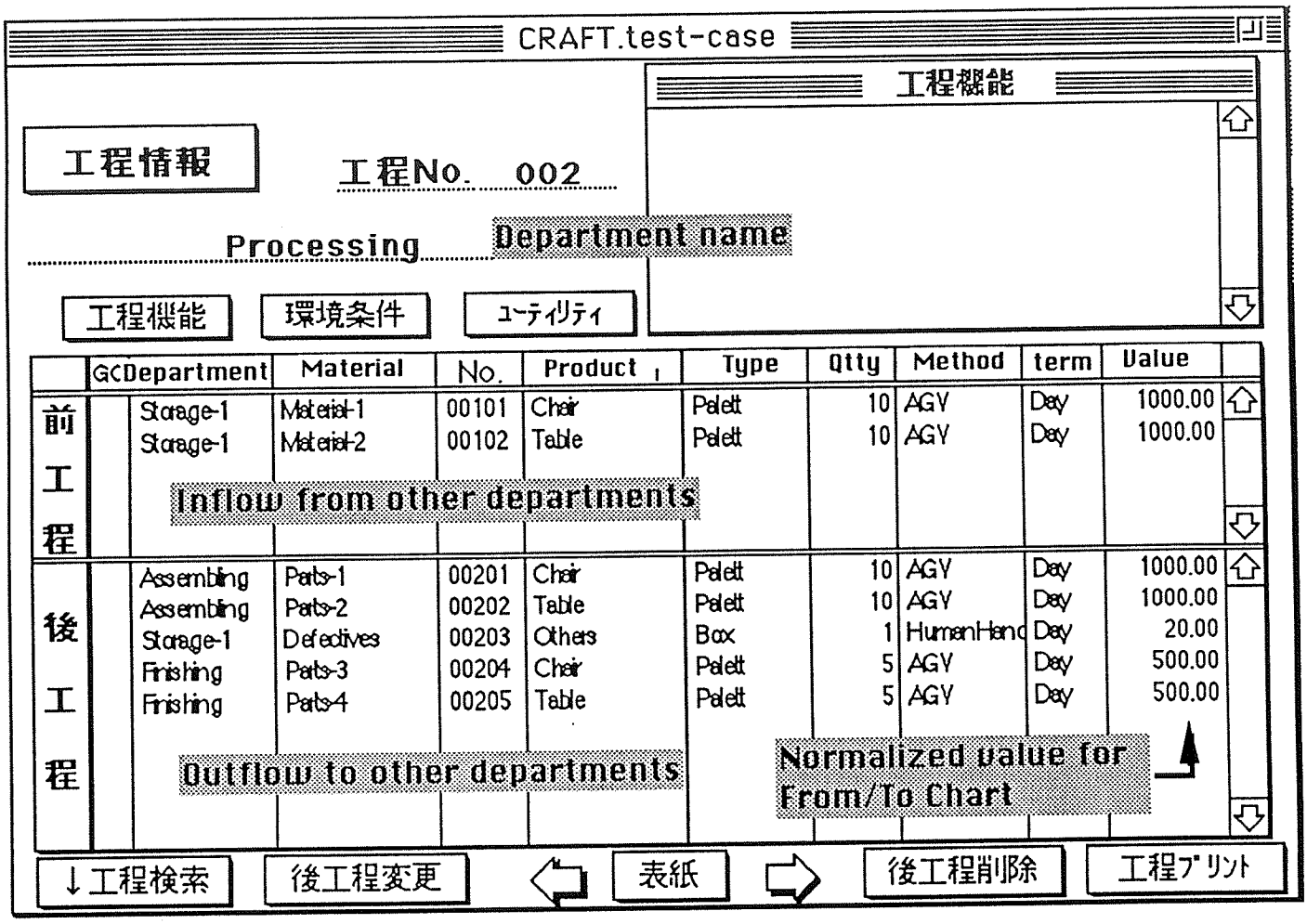


Fig.3-5

Normalization of flow value

As the output of this process analysis system we can get the From/To Chart such as fig. 3-1.

B. Data transfer from a Macintosh to a SUN workstation

The From/To Chart form should be transformed to the form such as fig. 3-6 so that ProKappa[2] can read and create instance objects for each departments with parents, slots and slot facets as shown in the figure. A macro program by Excel does this transformation. This program is shown in fig. 3-7.

```
Application sky

Instance D1
        Parent -> Departments
        Slot MaterialFlow -> ?
                Facet D1 ->     0
                Facet D2 ->     98
                Facet D3 ->     3
                Facet D4 ->     12
                Facet D5 ->     8

Instance D2
        Parent -> Departments
        Slot MaterialFlow -> ?
                Facet D1 ->     4
                Facet D2 ->     0
                Facet D3 ->     88
                Facet D4 ->     21
                Facet D5 ->     12

Instance D3
        Parent -> Departments
        Slot MaterialFlow -> ?
                Facet D1 ->     11
                Facet D2 ->     15
                Facet D3 ->     0
                Facet D4 ->     56
                Facet D5 ->     11

Instance D4
        Parent -> Departments
        Slot MaterialFlow -> ?
                Facet D1 ->     2
                Facet D2 ->     5
                Facet D3 ->     8
                Facet D4 ->     0
                Facet D5 ->     87

Instance D5
        Parent -> Departments
        Slot MaterialFlow -> ?
                Facet D1 ->     45
                Facet D2 ->     12
                Facet D3 ->     23
                Facet D4 ->     22
                Facet D5 ->     0
```

Fig. 3-6 : Data format for ProKappa

---

[2] ProKappa is an expert system shell that implements the object oriented environment. It is developed by Intelli Corp.

```
                    convertMacro

                    "=FOPEN(""TextFile"",3)"
Application sky     "=FWRITE(B3,A4&CHAR(13)&CHAR(13))"
                    "=SET.NAME(""Site"",SELECTION())"
5                   "=SET.VALUE(A6,ROWS(SELECTION())-1)"
5                   "=SET.VALUE(A7,COLUMNS(SELECTION())-1)"

                    "=FOR(""y"",1,A6)"
                    "=SELECT(""R[1]C"")"
Instance            "=FWRITE(B3,A11&CHAR(32)&ACTIVE.CELL()&CHAR(13))"
Parent -> Departments "=FWRITE(B3,CHAR(9)&A12&CHAR(13))"
Slot MaterialFlow -> ? "=FWRITE(B3,CHAR(9)&A13&CHAR(13))"

                    "=FOR(""x"",1,A7)"
                    "=SELECT(""RC[1]"")"
0                   "=IF(ACTIVE.CELL()="""",SET.VALUE(A17,0),SET.VALUE(A17,ACTIVE.CELL()))"
                    "=FWRITE(B3,CHAR(9)&CHAR(9)&""Facet ""&INDEX(Site,1,x+1)&""
                    ->""&CHAR(9)&A17&CHAR(13))"
                    =NEXT()
                    "=FWRITE(B3,CHAR(13))"
                    "=SELECT(OFFSET(ACTIVE.CELL(),0,1-x))"
                    =NEXT()
                    =FCLOSE(B3)

                    =RETURN()
```

Fig. 3-7 : Convert Macro by Excel

The text file transformed as fig. 3-6 can be used as an input data to ProKappa. We can transfer the text file from a Macintosh to a SUN workstation by MacIP and MacTCP. Fig. 3-8 shows the process of data transfer from the Mac to a SUN.

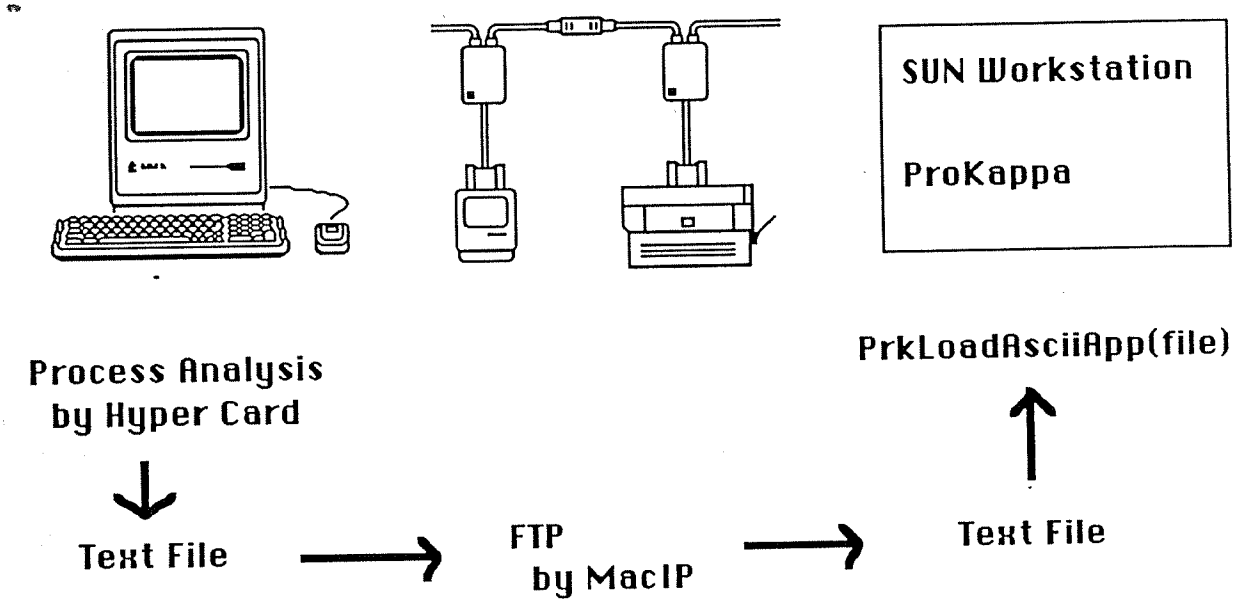


Fig.3-8

C. A SUN workstation and ProKappa

The system configuration of SKY is shown in fig. 3-9.

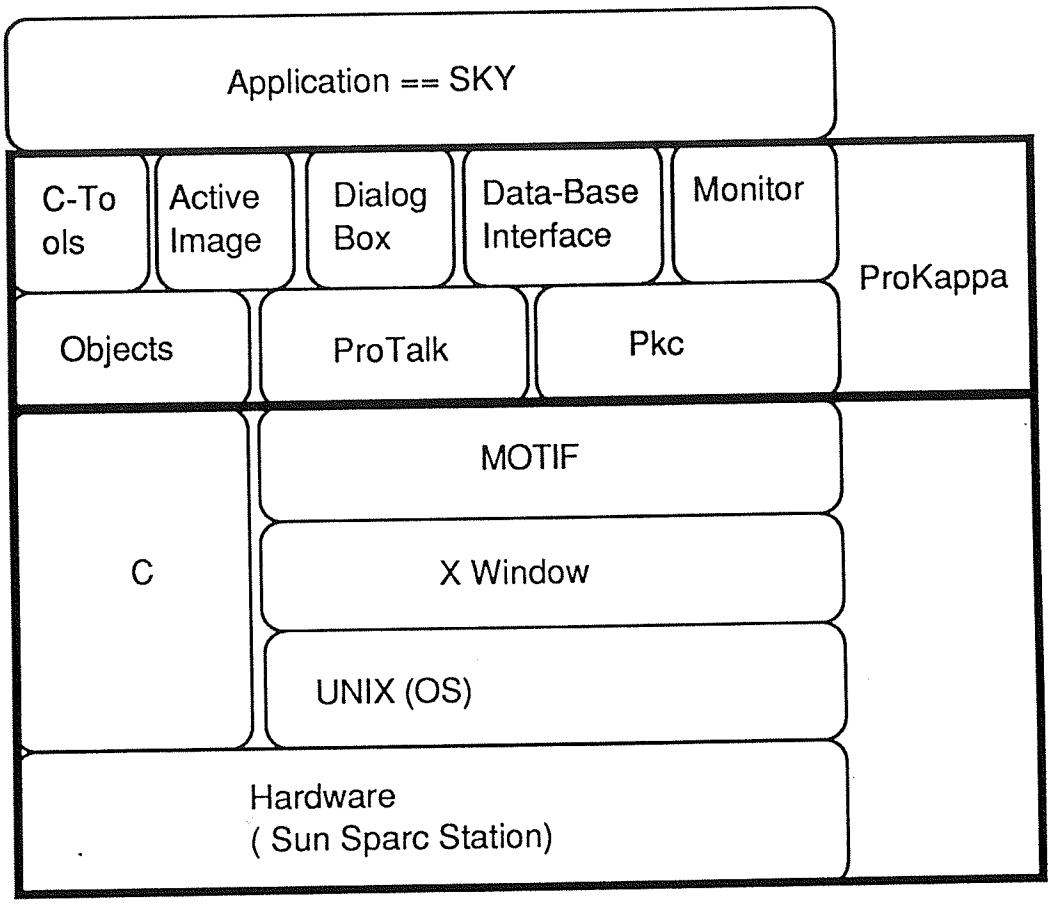


Fig.3-9

SKY is built using ProKappa, runnig on a SUN with UNIX, C, X and Motif

IV. Modeling

A. Modeling and objects

SKY attempts to transfer the human layout-planning process which was discussed in chapter 2 to the computer. To let the computer perform these design and analysis tasks, we need to make a model of the real problem inside the computer.

Site condition is represented by a grid. (Fig. 4-1) We call the grid elements "rooms". The size of the site is determined by the number of rooms.

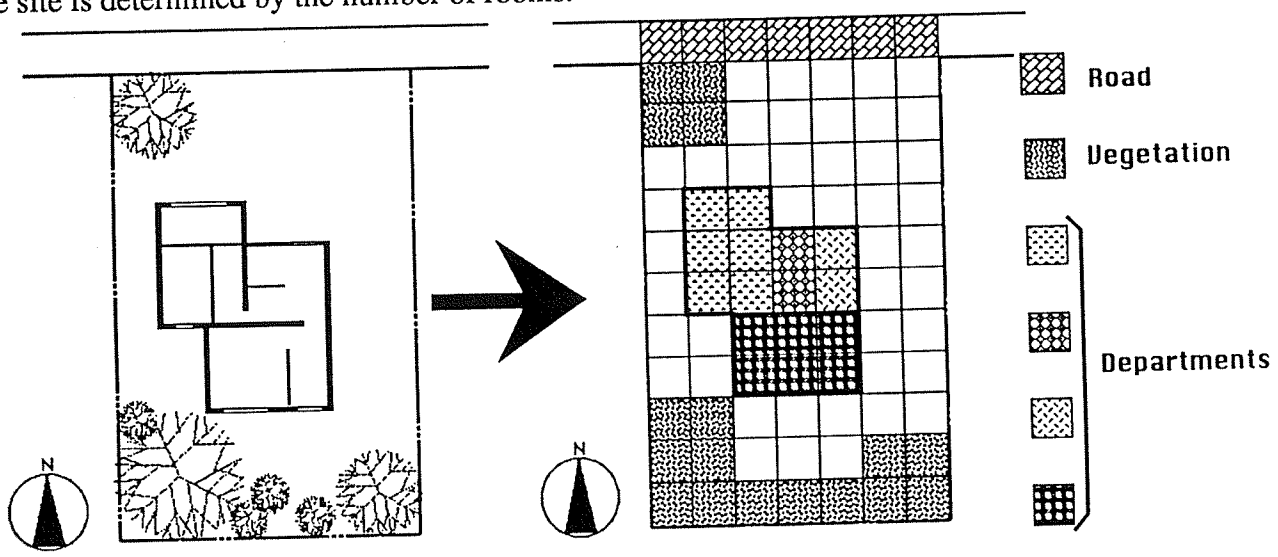


Fig.4-1

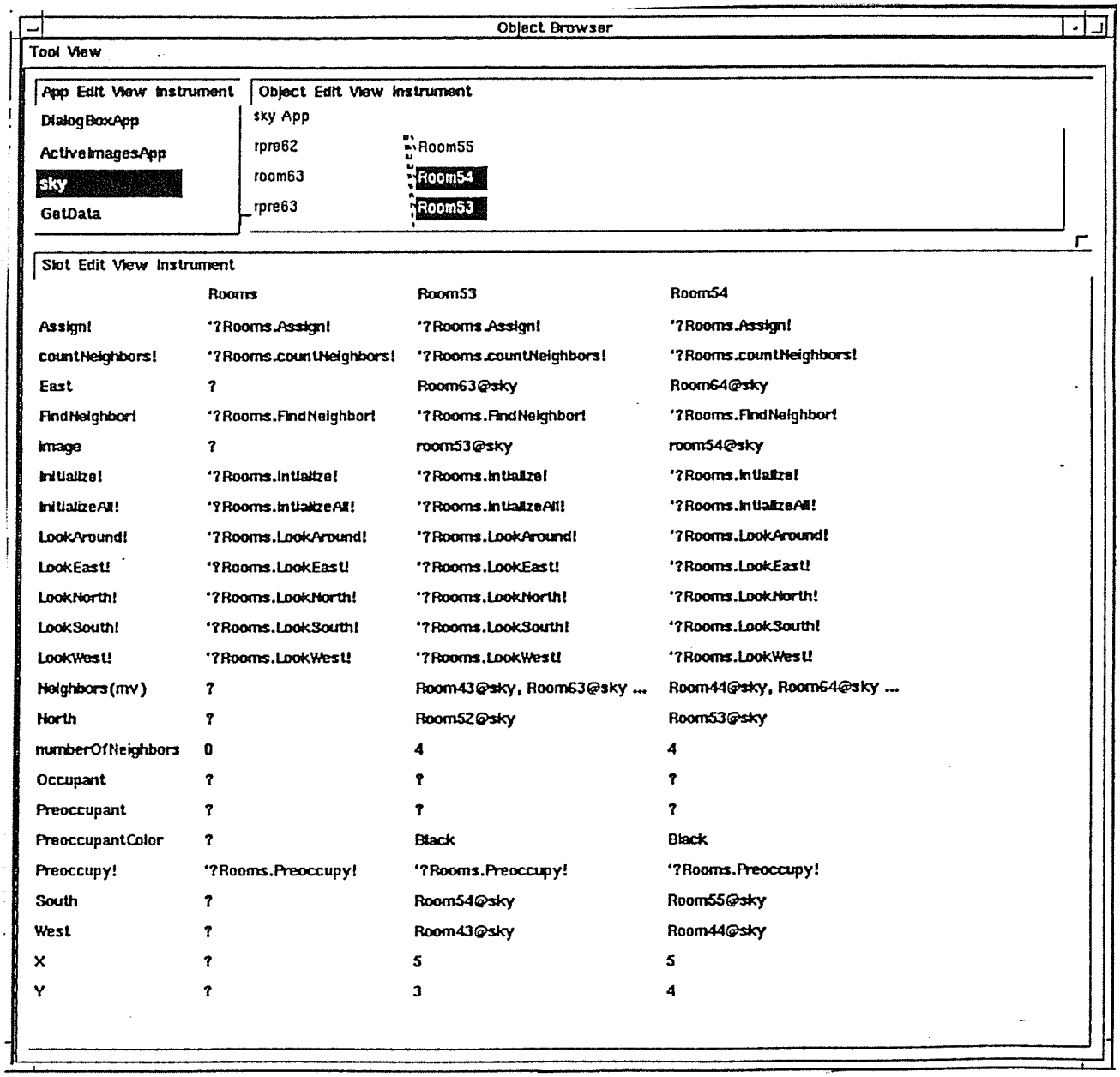
SKY represents the site by a grid

The street location and the landscape features are represented by pre-occupation of rooms.
The upper direction is always considered to be North.
Departments to be allocated are represented by occupants of rooms.
In the modeling of SKY, rooms and occupants are two major objects.

The attributes of these two objects are shown in fig. 4-2 and fig. 4-3.

Fig. 4-2

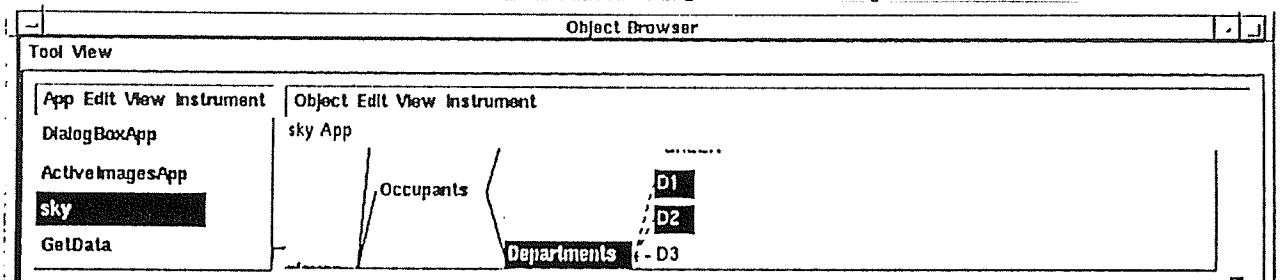
The attributes of the "Room" object.



	Rooms	Room53	Room54
Assign!	'?Rooms.Assign!	'?Rooms.Assign!	'?Rooms.Assign!
countNeighbors!	'?Rooms.countNeighbors!	'?Rooms.countNeighbors!	'?Rooms.countNeighbors!
East	?	Room63@sky	Room64@sky
FindNeighbor!	'?Rooms.FindNeighbor!	'?Rooms.FindNeighbor!	'?Rooms.FindNeighbor!
image	?	room53@sky	room54@sky
Initialize!	'?Rooms.Initialize!	'?Rooms.Initialize!	'?Rooms.Initialize!
InitializeAll!	'?Rooms.InitializeAll!	'?Rooms.InitializeAll!	'?Rooms.InitializeAll!
LookAround!	'?Rooms.LookAround!	'?Rooms.LookAround!	'?Rooms.LookAround!
LookEast!	'?Rooms.LookEast!	'?Rooms.LookEast!	'?Rooms.LookEast!
LookNorth!	'?Rooms.LookNorth!	'?Rooms.LookNorth!	'?Rooms.LookNorth!
LookSouth!	'?Rooms.LookSouth!	'?Rooms.LookSouth!	'?Rooms.LookSouth!
LookWest!	'?Rooms.LookWest!	'?Rooms.LookWest!	'?Rooms.LookWest!
Neighbors(mv)	?	Room43@sky, Room63@sky ...	Room44@sky, Room64@sky ...
North	?	Room52@sky	Room53@sky
numberOfNeighbors	0	4	4
Occupant	?	?	?
Preoccupant	?	?	?
PreoccupantColor	?	Black	Black
Preoccupy!	'?Rooms.Preoccupy!	'?Rooms.Preoccupy!	'?Rooms.Preoccupy!
South	?	Room54@sky	Room55@sky
West	?	Room43@sky	Room44@sky
X	?	5	5
Y	?	3	4

Fig. 4-3

The attributes of the "Department" object



Slot Edit View Instrument

	Departments	D1	D2
Adjacency	?	?	?
AverageFlow!	'?Departments.AverageFlow!	'?Departments.AverageFlow!	'?Departments.AverageFlow!
CalDis!	'?Departments.CalDis!	'?Departments.CalDis!	'?Departments.CalDis!
Color	?	Red	Yellow
DesiredNeighbors(mv)	?	?	?
feasibility	?	?	?
Features(mv)	?	needsAccess@sky	useFire@sky
FindEastViewRoom!	'?Departments.FindEastViewRoom!	'?Departments.FindEastViewRoom!	'?Departments.FindEastViewRoo
FindGoodViewRoom!	'?Departments.FindGoodViewRoom!	'?Departments.FindGoodViewRoom!	'?Departments.FindGoodViewRo
findNegative!	'?Departments.findNegative!	'?Departments.findNegative!	'?Departments.findNegative!
FindNorthViewRoom!	'?Departments.FindNorthViewRoom!	'?Departments.FindNorthViewRoom!	'?Departments.FindNorthViewRo
findPositive!	'?Departments.findPositive!	'?Departments.findPositive!	'?Departments.findPositive!
FindSouthViewRoom!	'?Departments.FindSouthViewRoom!	'?Departments.FindSouthViewRoom!	'?Departments.FindSouthViewRo
FindSunlightRoom!	'?Departments.FindSunlightRoom!	'?Departments.FindSunlightRoom!	'?Departments.FindSunlightRoom
FindWestViewRoom!	'?Departments.FindWestViewRoom!	'?Departments.FindWestViewRoom!	'?Departments.FindWestViewRoo
FirstRoom	?	Room33@sky	Room34@sky
Friends(mv)	?	?	?
GetRidOfMe!	'?Departments.GetRidOfMe!	'?Departments.GetRidOfMe!	'?Departments.GetRidOfMe!
LastRoom	?	Room32@sky	Room61@sky
MaterialFlow	?	?	?
MaterialFlowAverage	?	24	25
needsAccess	?	YES	NO
needsSunlight	?	NO	NO
needsView	?	NO	NO
negativeAdjacencies(mv)	?	?	D4@sky
NoOfRooms	?	4	4
NoOfStranger	?	0	3
NoOfWaiting	?	4	4
Occupants(mv)	?		
positiveAdjacencies(mv)	?	D2@sky, D5@sky	D3@sky
PreAssignRoom	?	?	?
SetAdjInfo!	'?Departments.SetAdjInfo!	'?Departments.SetAdjInfo!	'?Departments.SetAdjInfo!
SetFeatureInfo!	'?Departments.SetFeatureInfo!	'?Departments.SetFeatureInfo!	'?Departments.SetFeatureInfo!
Strangers(mv)	?	?	D1@sky, D5@sky ...
Test	'!Departments_Test	'!Departments_Test	'!Departments_Test
Type	?	Block	?

B. Modeling of process

The human layout-planning process which is discussed in chapter 2 is represented by the flow diagram shown on the left and implemented in the control panel shown on the right in fig. 4-4. The numbers indicate the correspondence between the human process and computer process.

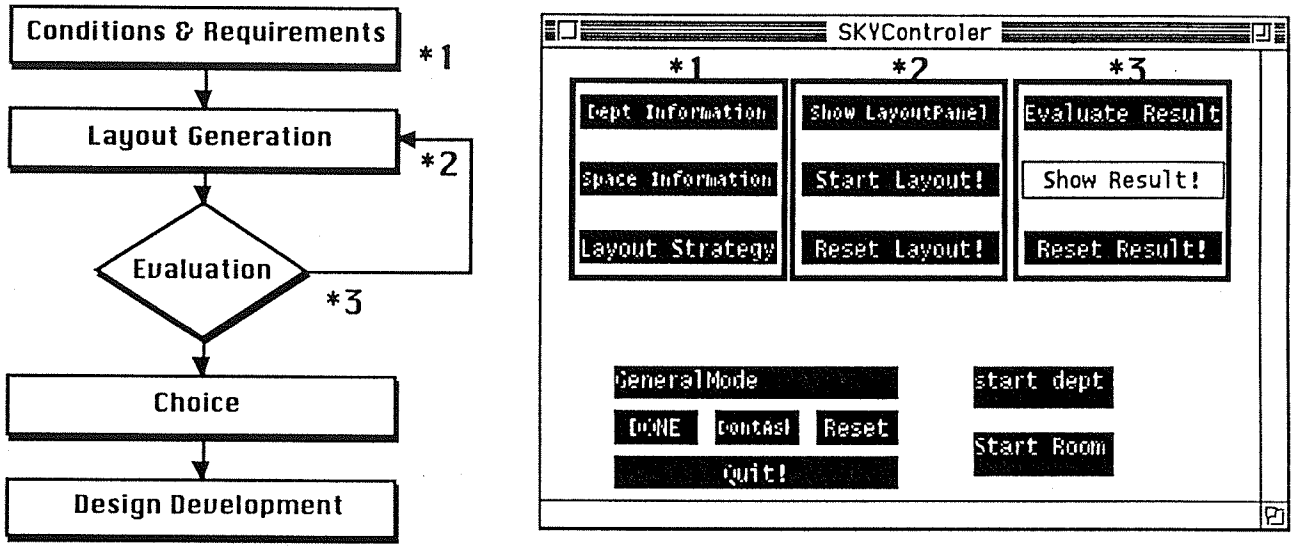


Fig.4-4

Each black button on the SKY control panel performs a procedure. The first set (*1) allows the user to input data. The second and third sets invoke SKY procedures.

V. Knowledge Representation

There are several kinds of knowledge and procedures which are needed to make a layout. They include,

knowledge about how to set adjacency of each department
procedures to generate layout alternatives
knowledge about when to backtrack in the process of layout generation
knowledge about how to evaluate alternatives
knowledge about evaluation criteria according to the situation of a project.

They are implemented in SKY as follows;

1) Adjacency---> OODB in object features
2) Layout generation ---> Exhaustive search
3) Search pruning ---> Rules
4) Evaluations ---> Several parameters
5) Recognition of criteria ---> Preference factor

The details of the implementation are discussed in the following sections.

A. Adjacency definition

How to set departments' adjacencies according to their features is a very important part of layout study. SKY stores this knowledge as objects' attributes and rules.

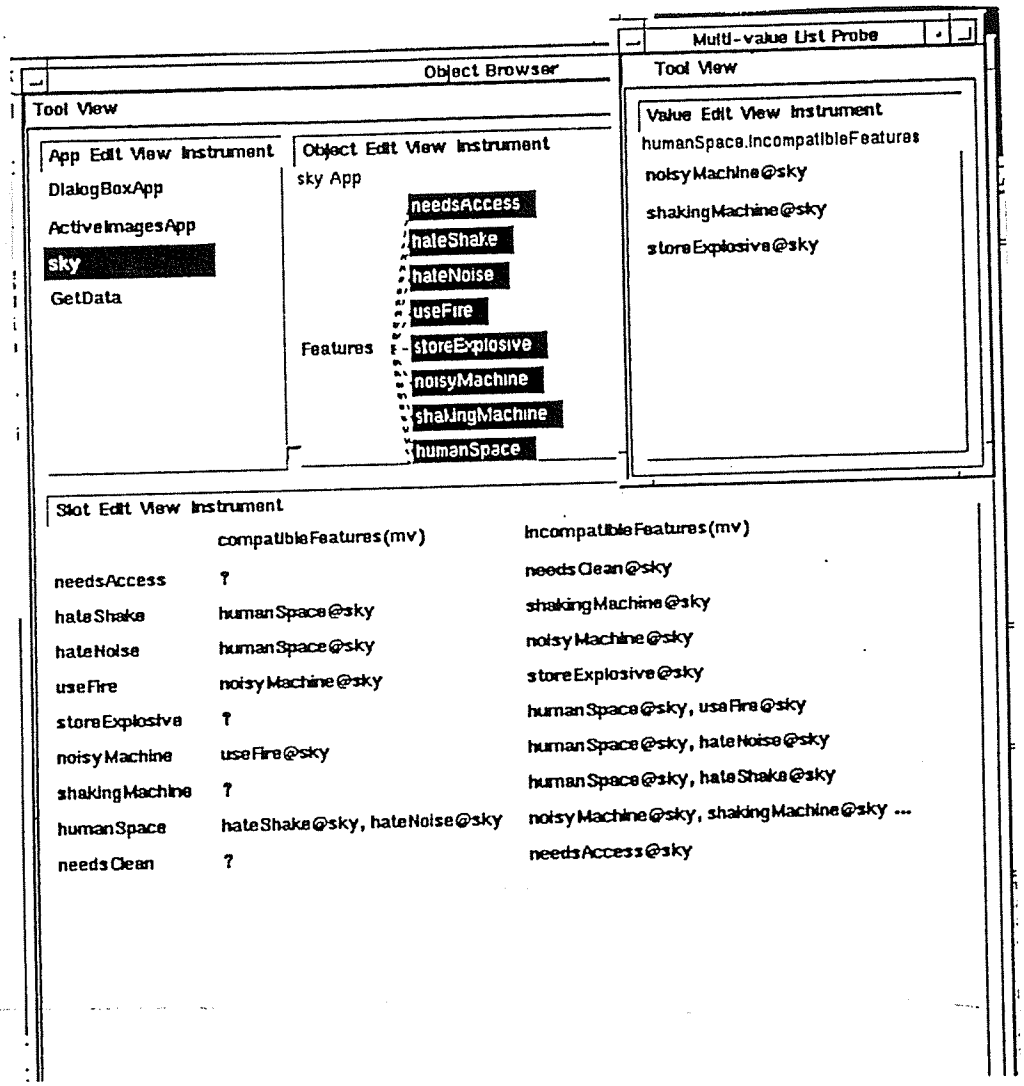
1. Features and their compatibility

SKY has an object named "Features" and that object has attributes named "incompatible features" and "compatible features". For instance, a feature "hateNoise" is compatible with a feature "humanSpace", because these features do not disturb each other even if they are allocated near each other. On the other hand, a feature "useFire" is incompatible with a feature "storeExplosive", because these two features adjacent can be very dangerous.

Fig. 5-1. shows the values of each feature. This table itself represents adjacency knowledge.

Fig. 5-1

Object hierarchy of "Features" and attributes of them



2. Procedure of setting adjacency

The rules which define adjacencies are as follows:

First, a department has a negative adjacency with a department which has an incompatible feature with it.

Second, a department has a positive adjacency with a department which has same or compatible feature with it and has no negative adjacency with it. (Usually a department may have more than two features.)

Third, a department can have a positive adjacency with another department if there is a large amount of material flow between them and they have no a negative adjacency.

The details are as follows;

a) Negative Adjacency

Negative adjacency is set between two departments which have incompatible features. For instance, negative adjacency is set between a department which has a feature "useFire" and a department which has a feature "storeExplosive".

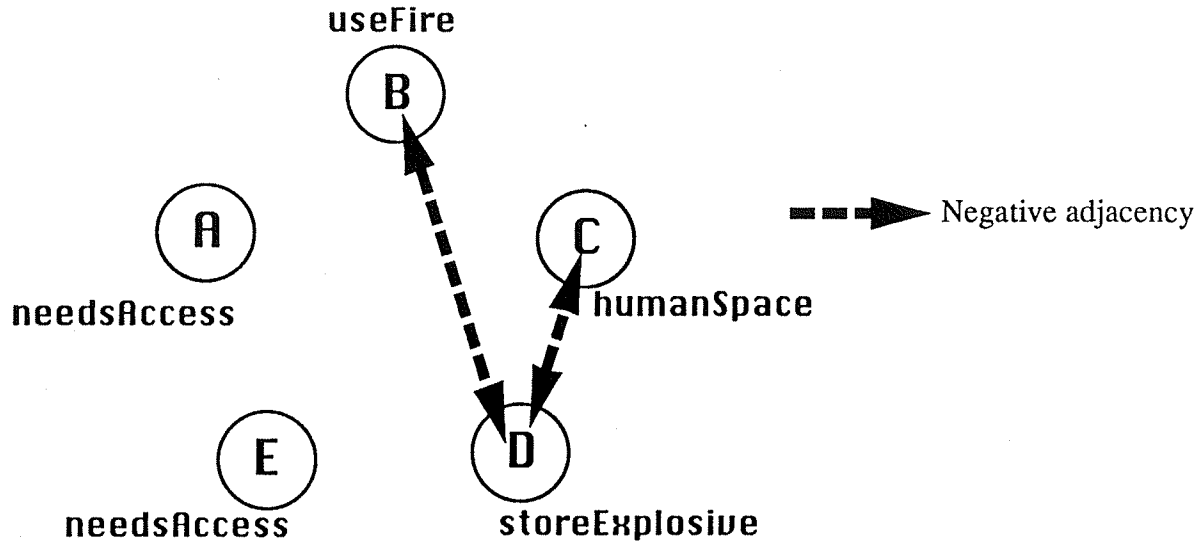


Fig.5-2

The code to set negative adjacency is as follows;

```
---------------------------------------------------------------------------------
method Departments.findNegative! ()
{

    ClearValues(?self,negativeAdjacencies);

    for ?D2 = find [adjacencyRules] ?self.negativeAdjacencies;     /* invoke rules     */
    do ;

  return Null;
```

```
}

bcrule ad_rule1 in adjacencyRules
{if:
    ?D2 == instanceof Departments;
    ?D2 != ?D1;
    ?f1 = ?D1.Features;
    ?f2 = ?D2.Features;
    ?f1.incompatibleFeatures == ?f2;
then:
    ?D1.negativeAdjacencies +== ?D2;
    Print ("\n", ?D1, "negativeAdjacencies:", ?D2);
}
```

---

b) Positive Adjacency is set afterward.

Two departments which have a negative adjacency can never have a positive adjacency.

Positive adjacency is set between two department which have compatible features and do not have a negative adjacency. For instance, a positive adjacency is set between a department which has a feature "hateNoise" and a department which has a feature "humanSpace" if they have no negative adjacency. The same feature is considered compatible, so a positive adjacency is set between two departments which have the same feature if they do not have a negative adjacency.

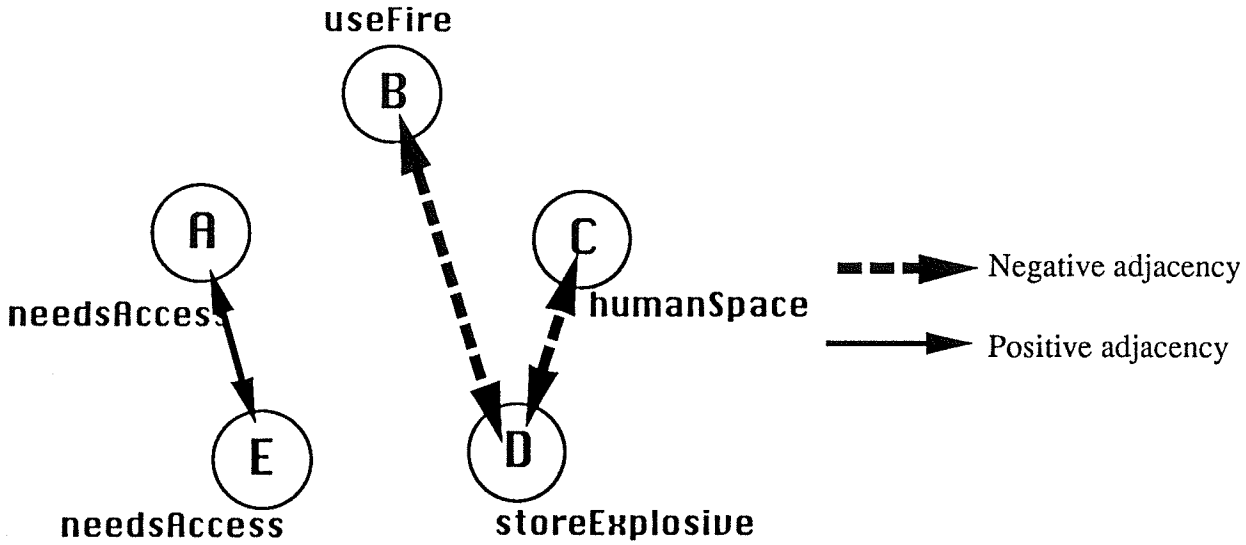


Fig.5-3

The code to set positive adjacency is as follows;

```
method Departments.findPositive! ()
{

    ClearValues(?self,positiveAdjacencies);
/*  find [positiveAdRules] ?D2 = ?self.positiveAdjacencies; **
    ---This code can get only one solution.          */
```

```
    for ?D2 = find [positiveAdRules] ?self.positiveAdjacencies;      /* invoke rules */
    do ;

  return Null;
}

bcrule po_rule1 in positiveAdRules
{if:
    ?D2 == instanceof Departments;
    ?D1 != ?D2;
    ?f1 = ?D1.Features;
    ?f2 = ?D2.Features;
    ?f1 == ?f2;
    ?D1.negativeAdjacencies != ?D2;
then:
    ?D1.positiveAdjacencies +== ?D2;
    Print ("\n", ?D1, "positiveAdjacencies:", ?D2);
}

bcrule po_rule2 in positiveAdRules
{if:
    ?D2 == instanceof Departments;
    ?D1 != ?D2;
    ?f1 = ?D1.Features;
    ?f2 = ?D2.Features;
    ?f1.compatibleFeatures == ?f2;
    ?D1.negativeAdjacencies != ?D2;
then:
    ?D1.positiveAdjacencies +== ?D2;
    Print ("\n", ?D1, "positiveAdjacencies:", ?D2);
}
```

---

c) Adding positive adjacency according to the material flow.

At this point we can draw a directed graph in which vertices represent departments and branches represent positive adjacencies.

If every department is included in that graph, the adjacency setting is done. If some departments are isolated, the following procedure is executed to connect every department in one graph.

1) First, we look for the dominant department which has the greatest number of successors. Please see fig. 5-4. In this case, the dominant department is Da1.

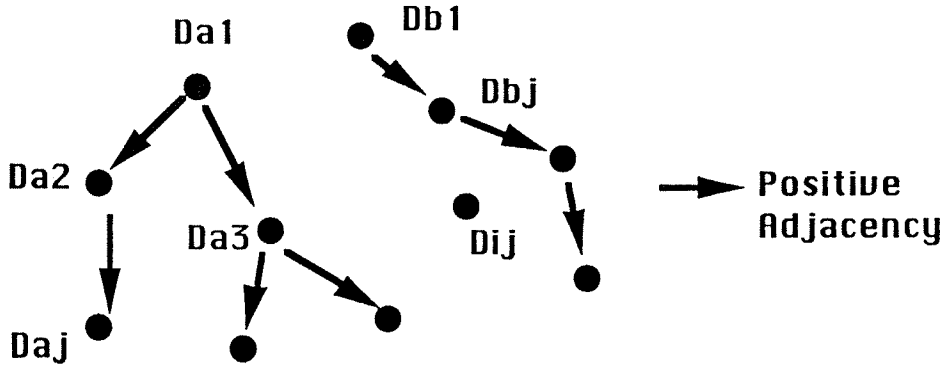


Fig.5-4

2) Second, we look for the orphan departments which are not connected in the graph which includes the dominant department. In the case of fig. 5-4., Dd1, Dbj and Dij are the orphan departments.

3) Third, we look for the dominant department in the orphan departments. In case of fig. 5-4., Db1 is the dominant orphan.

4) Forth, we look for a department X (in this case it is Da3) from which a great number of material flows to the orphan dominant department B(in this case Db1), and set a positive adjacency from X to B.

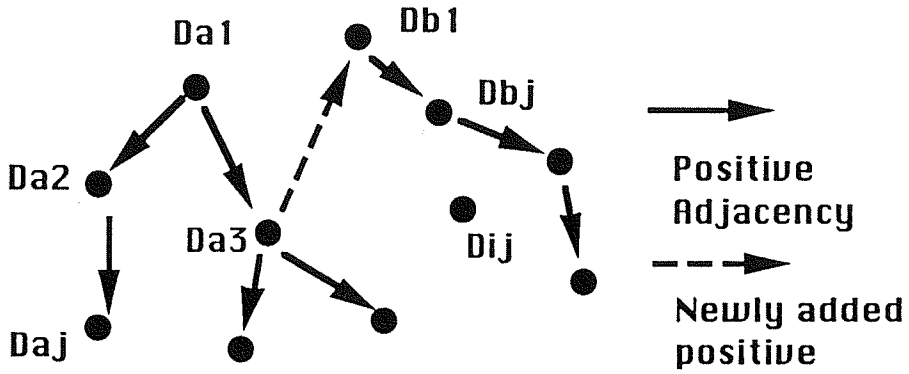


Fig.5-5

5) At this point we draw the adjacency graph again and if every department is included in that graph, the adjacency setting is done. If not we follow the same procedure from 1).

The code to add positive adjacencies is as follows;

```
---------------------------------------------------------------------------------
method DeptControler.lookAdjacency! ()
{
    for ?D = find direct instanceof Departments;
       do  SendMsg(?D,AverageFlow!);
    for ?D = find direct instanceof Departments;
    do  SendMsg(?D,findNegative!);
    for ?D = find direct instanceof Departments;
    do  SendMsg(?D,findPositive!);

    ?dD = SendMsg(?self, Dominant!);
    ?self.DominantDept = ?dD;
    ?self.NumberOfOrphans = ?dD.NoOfStranger;

    while ?dD.NoOfStranger >0;
    do {
        ?orphanList = GetValues(?dD, Strangers);
        ?oD = DominantD(?orphanList);
        ?oDnegativeList = GetValues(?oD,negativeAdjacencies);
        for ?dept = find direct instanceof Departments;
        do {
            ?dept != ?oD;
            ?dept.MaterialFlow..?oD > ?dept.MaterialFlowAverage;
            ?judge = Member_of(?dept,?oDnegativeList);
            ?judge != -1;

            ?dept.positiveAdjacencies +== ?oD;
          }
```

```
        ?dD = SendMsg(?self, Dominant!);

        if ?self.NumberOfOrphans == ?dD.NoOfStranger;
        then {
            SendMsg(CantGoAnyMoreDialogBox,PutOnScreenAndWait!);
            return Null;
            }

        ?self.DominantDept = ?dD;
        ?self.NumberOfOrphans = ?dD.NoOfStranger;
      }
  return Null;
}
```

---

d) Completion of adjacency tree

As the search to layout department is done along the graph of positive adjacency, every department should be included in the graph like fig 5-6. If one is not, we must connect it manually. Otherwise the unconnected department will not appear in the final layout.

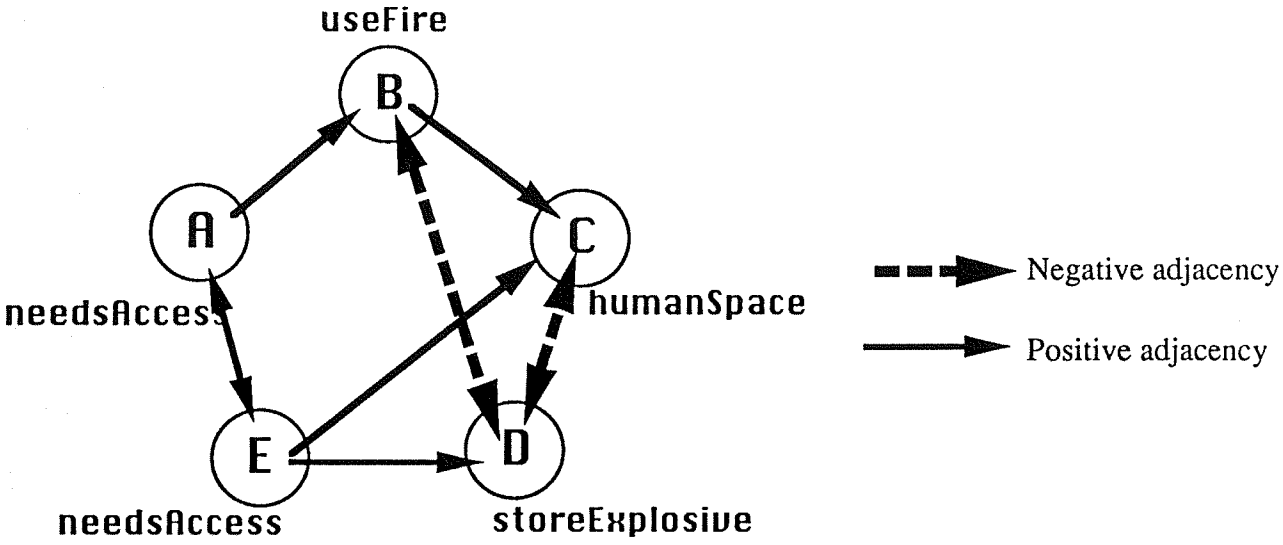


Fig.5-6

Example adjacency graph

B. Search logic

The main part of the reasoning of this system is Search. Human problem solving involves search. The layout generating process is represented by a state-action tree in which the root vertex represents the initial state which shows the starting room and starting department; branches represent alternative operations which may be performed; internal vertices represent partially-specified states; and terminal vertices represent fully-specified states.

Let's see a very simple example which has 4 rooms and 2 departments which need 2 rooms each. The whole search tree is like fig. 5-7.

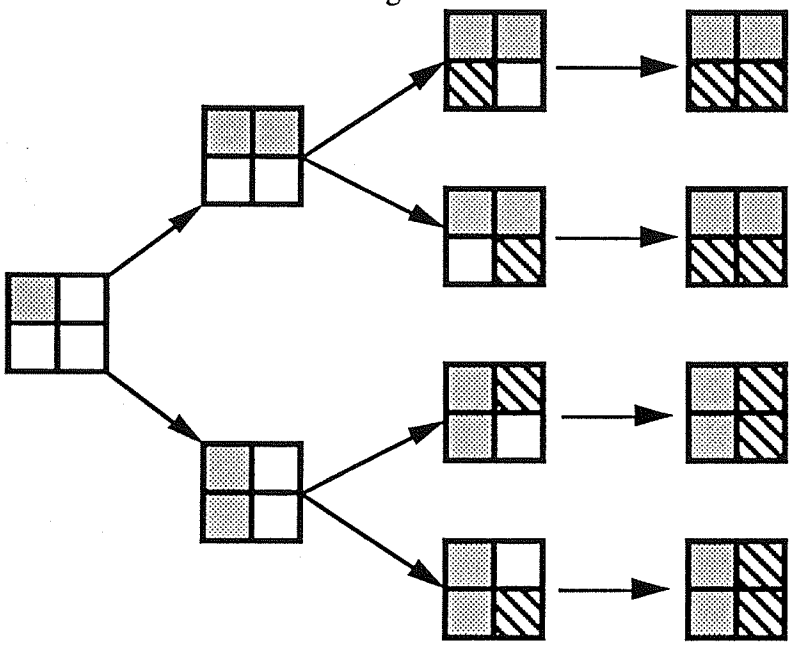


Fig.5-7

Simple example of search tree to layout 2 departments each with 2 rooms

Every room has a method slot that tells a room to assign itself to a given department.
The method says as follows;

1) When you receive a message, let the department be your occupant.

2) If the department is not fully assigned, then look for your unoccupied neighbor and send a message to it.

3) If the department is fully assigned, then look for the department which has positive adjacency with the current one, and look for an unoccupied room which is a neighbor of one of the rooms occupied with the previous department and send a message to the next room and ask it to assign itself to the next department.

1. Backtrack and exhaustive search

ProKappa is very powerful to do this kind of search, because it can backtrack automatically after fully exploring a possibility.

The code for search by ProTalk is as follows;

```
---------------------------------------------------------------------------------------------
method Rooms.Assign! ()
{
/*----------Assign to the current department.-----------------------------------------------*/
?RoomsInDept = generalControler.CurrentDept.NoOfRooms;
?CurrentD = generalControler.CurrentDept;          /* Remember current Dept */
?self.Occupant = ?CurrentD;                        /* Record current occupant */
generalControler.CurrentRoom = ?self;
?CurrentD.Occupants +== ?self;
roomControler.OccupiedRooms +== ?self;
?CurrentD.NoOfWaiting = ?CurrentD.NoOfWaiting - 1;
?self.image.Background = ?CurrentD.Color;
/*----------End of assignment.-------------------------------------------------------------*/

/*----------If the department is fully assigned---------------------------------------------*/
if find count ?CurrentD.Occupants >= ?RoomsInDept;
                                                   /* Done with this department */
    {

/*Feasibility Check  */
    /*  Feasibility Check by Rules. */
    if SendMsg(DeptTest, checkAll!) == NO;
            then
               {
               SendMsg(?self,Initialize!);
               fail;}
            else
/* End of Feasibility Check */

    for find ?R = roomControler.OccupiedRooms;
    do {
              ?nextD = find ?R.Occupant.positiveAdjacencies;

              ?nextD.NoOfWaiting > 0;
              ?nextD.NoOfRooms > 0;              /* Next dept must need rooms */
          ?NextR = find ?R.Neighbors;    /* NextR = Find neighbor */
          ?NextR.Occupant == Null;       /* Neighbor must be available */
              if ?NextR.Preoccupant != Null;  /* PreAssignCheck */
                then ?NextR.Preoccupant == ?nextD;

              generalControler.CurrentDept = ?nextD;
                                                   /* Get ready to do next dept */
              ?CurrentD.LastRoom = ?self;
              ?nextD.FirstRoom = ?NextR;
              if generalControler.Mode != DONE;
              then SendMsg(?NextR, Assign!);       /* Go assign next room, dept */
              generalControler.CurrentDept = ?CurrentD;
                                                   /* Reset on return */
     }

/*--- Evaluation of this alternative. ----------------------------------------*/
    (Omit.  Please refer tp the original for the detail.)
/*---End Of Evaluation. ---------------------------------------------------*/

    SendMsg(?self, Initialize!);
    return Null;
```

```
    }
/*----------------End if "If the department is fully assigned.----------------------------------------*/

for find {      ?NextR = ?self.Neighbors;   /* Get all remaining neighbors */
                }
do {            ?NextR.Occupant == Null;            /* Next must be available */
    if ?NextR.Preoccupant != Null;      /* preAssign Check for Next */
      then ?NextR.Preoccupant == generalControler.CurrentDept;

    SendMsg(?NextR, Assign!);
    }
SendMsg(?self, Initialize!);
fail;                                               /* Backtrack */
return Null;
}
-------------------------------------------------------------------------------------------
```

Fig 5-8 on the next 3 pages shows how the message is sent and the backtrack occurs, and how all possible alternatives can be explored.

In this simple case, first the "Assign!" message is sent to the starting room which is assigned by the planner (In this case the starting room is R11). Then the room looks for an empty neighbor, and sends "Assign!" message to it.

After receiving a message, the second room lets the department be its occupant, and changes the current department to D2 which has a positive adjacency with the previous fully assigned department D1. Then it looks for an empty room which is a neighbor of D1's room, and it sends a message to it.

These procedures are continued till there comes to a dead end, i.e., there is no room to send a message or there is no more department to be assigned. When a room cannot find any room to send a message to, it remembers where it got the message and sends a message back. The room which received a message back to it looks for another possibility.

This is one example of exhaustive search in which all layout options are tried.

Fig. 5-8 (1)

1: The message is sent to "R11". "R11" assigns "D1" to itself. 2: "R11" sends a message to its one of empty neighbors, "R12". 3: "D1" is fully allocated. The current department is changed to "D2". After that "R12" sends a message to its empty neighbor "R22". 4: "R22" sends a message to its empty neighbor "R21". 5: There is no more department, no more room. "R21" send back a message to "R22" from which it received a message previously. 6: "R22" looks for another possibility but in vain. It sends back a message to "R12" from which it received a message previously.

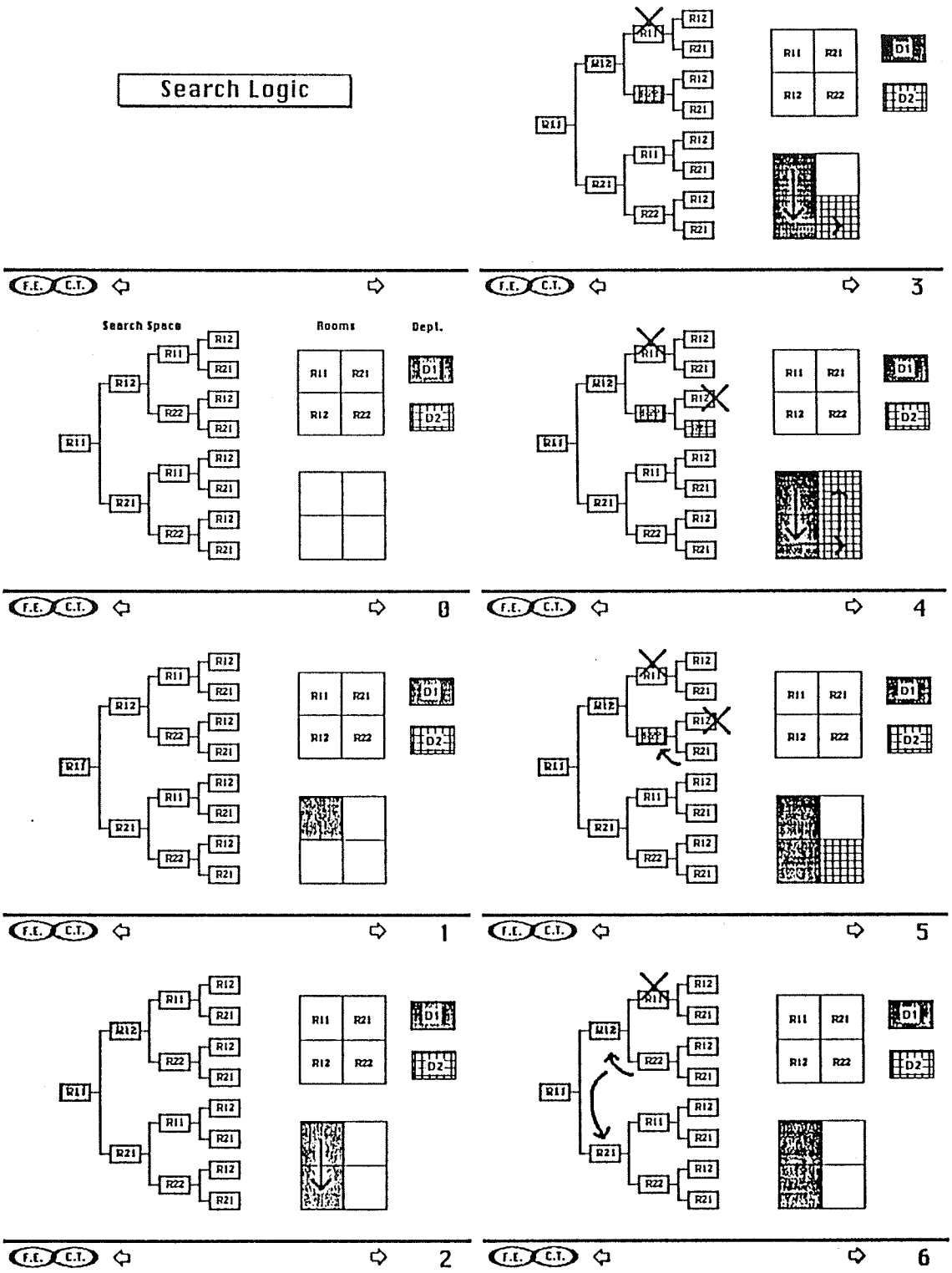


Fig. 5-8 (2)

7: "R12" looks for another possibility. The rooms assigned to "D1" have another empty neighbor, "R21". "R12" sends a message to "R21" 8: "R21" sends a message to its empty neighbor "R22". 9: Backtrack begins because it comes to an dead end. 10: A message is sent back to "R12". 11: "R12" cannot find any alternative, so it sends a message back to "R11". 12: "R11" looks for another empty neighbor "R21". 13~: The same procedure is done until every possibility is explored.

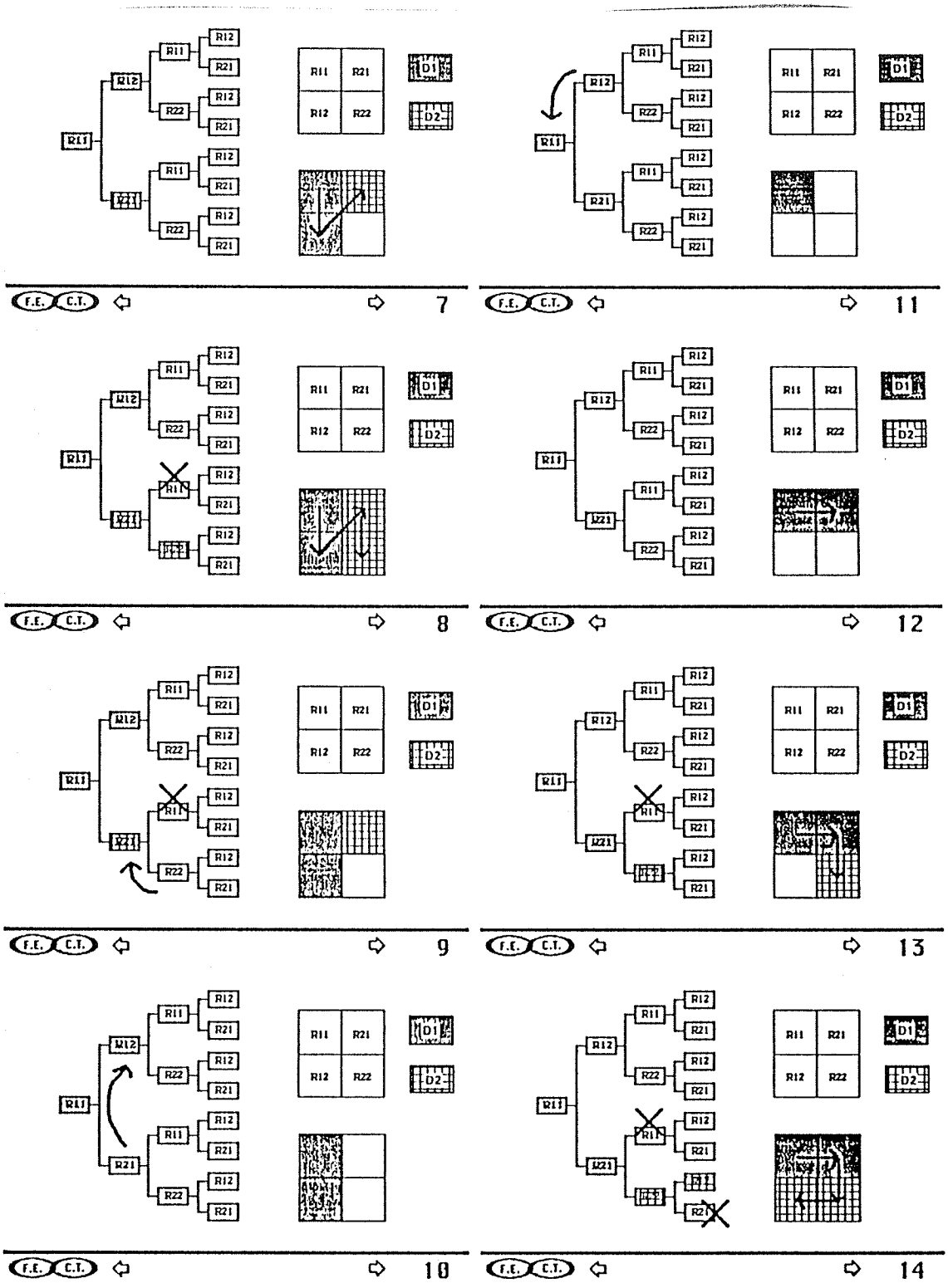
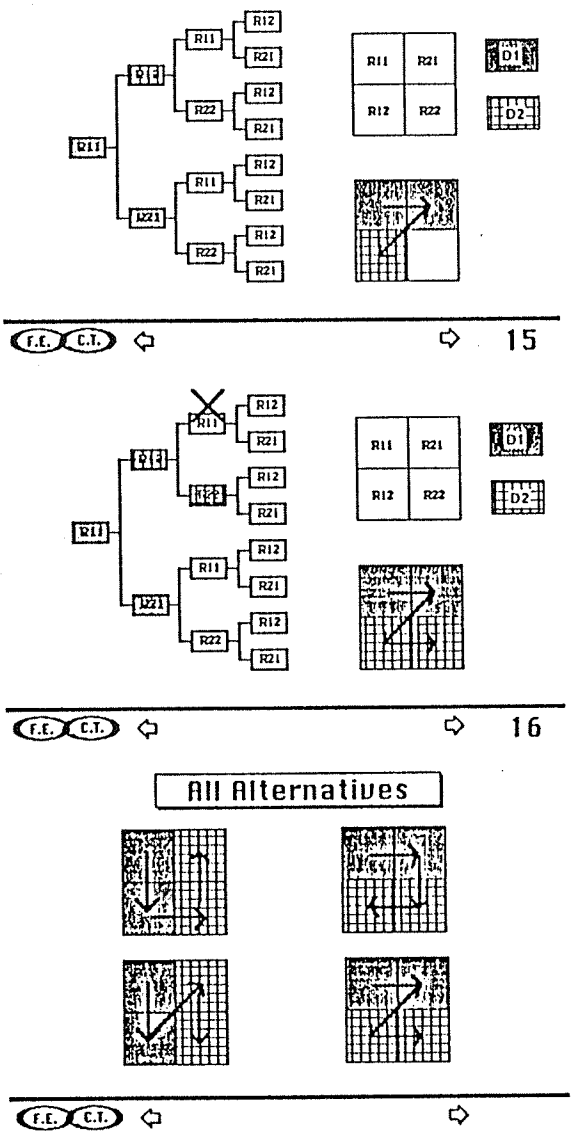


Fig. 5-8(3)



2. Search pruning

If we do not have any limitation of the search, the search space size is enormous. One of the most significant abilities of human beings is to recognize a dead end from which no satisfactory solution can result, to stop searching, and to go back to the previous state to restart the search along an alternative branch.

SKY does the same thing to identify and eliminate infeasible alternatives.

SKY has several tests to check whether a partial layout is feasible or not. These tests are Pre-assign test, Block test, Access test, Negative adjacency test, View test and Sunlight test.

Rules are invoked after each department is allocated. If allocation of a department is infeasible, further elaboration of this layout option is stopped, and the search routine backs up to try another layout options.

The ProTalk code for pruning is as follows;

```
/*------------------in Assign! method-------------------------------*/
            /*  Feasibility Check by Rules.
            if SendMsg(DeptTest, checkAll!) == NO;
              then
                  {
                  SendMsg(?self,Initialize!);
                  fail;}
              else
            */
/*-----------------------------------------------*/

method Test.checkAll! ()-----------------------------------------------
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

          if for ?test = find instanceof ?self;
          always SendMsg(?test,checkFeasibility!) == YES;

          then return YES;
          else return NO;
}
-------------------------------------------------

method Test.checkFeasibility! ()---------------------------------------
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");
          ?D = generalControler.CurrentDept;
          ?D.feasibility = Null;
          if ?self.Active == NO;  return YES;

          ?object = GetValues(?self, Depts);
    ?judge = Member_of(?D, ?object);
          if ?judge != -1;       return YES;
```

```
            if {    find [?self.Rules] ?x = ?D.feasibility;
                    /* need step below to invoke rules */
                    ?x == NO;}
            then return NO;
            else return YES;
}
-------------------------------------------------------

Sample of a rule.-----------------------------------
/* sunlightTest ---------------------------------
 * This Dept needs empty neighbor to get sunlight.
 * Also this room may not block neighbor's sunlight.
 *
 *-------------------------------------------------*/
bcrule sunlightrule1 in sunlightTestRules priority 0
{if:
            ?D = generalControler.CurrentDept;
            ?R = generalControler.CurrentRoom;
            ?D.needsSunlight == YES;
            ?empty = SendMsg(?R, LookAround!);
            ?empty == 0;
 then:
            ?D.feasibility = NO;
}

bcrule sunlightrule2 in sunlightTestRules
{if:
            ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
            ?neighbor = ?R.Neighbors;
            ?nD = ?neighbor.Occupant;
            ?nD.needsSunlight == YES;
            ?nDR = SendMsg(?nD, FindSunlightRoom!);
            ?nDR == 0;
 then:
            ?D.feasibility = NO;
}
-------------------------------------------------------
-----------------------------------------------------------------------------------------
```

The logic for search pruning is very important to reach good solutions effectively. Pruning rules are an important part of layout knowledge. So the system should be open to new pruning rules. SKY is open. That means we can easily add a new rule or knowledge. All we have to do is add a new instance of TEST-object and to write a rule in its method slot as fig. 5-9 shows.

This modularity is a very important feature of this system to enable it to grow.

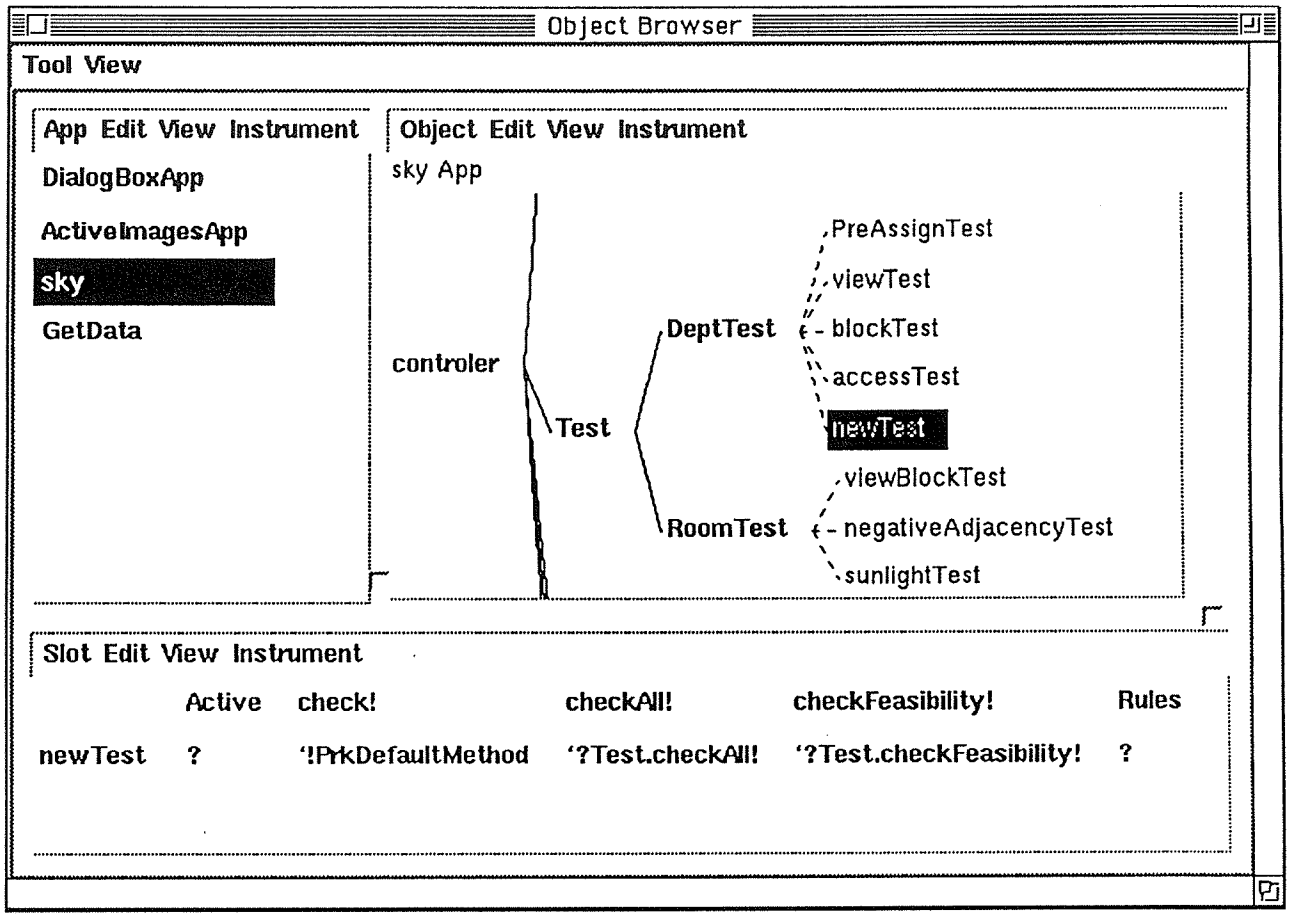


Fig.5-9

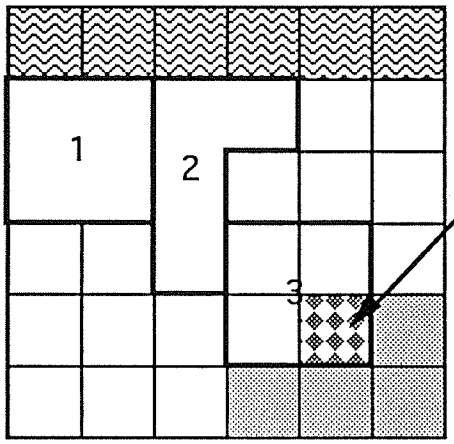
New test can be easily added as a new instance of Test object.

a) Pre-assign Test

Some spaces may be pre-assigned to specified departments, to reflect prior design decisions. In those cases, a planner can assign a room to the special department before starting layout. This test checks if the department includes the room which is pre-assigned to that department.

As fig. 5-10 shows, when a pre-assigned room is a member of the rooms which are assigned to the department, then it is OK.

Pre-assign Test



Preassigned room.

Every allocation of D3 fails if the allocation does not include preassigned room.
By preassign the planner can direct the place of D3, and can make search efficient.

Fig.5-10

The code to check this test is as follows;

```
method PreAssignTest.check! ()
{

    if ?self.Active == NO; return YES;

    ?D = generalControler.CurrentDept;
    ?target = ?D.PreAssignRoom;

    if ?target == Null;
      then return YES;
      else {
    ?list = GetValues(?D, Occupants);
    ?judge = Member_of(?target, ?list);
    if ?judge == -1;
      then return YES;
      else return NO;
        }
}
```

b) Block Test

The size of a department is represented by the number of rooms it needs.

The shape of a department is represented either as a Block-type or a Linear-type. As fig. 5-11 shows, a block-type department needs to be allocated as a cluster, on the other hand a linear type department can be allocated as a stripe also.

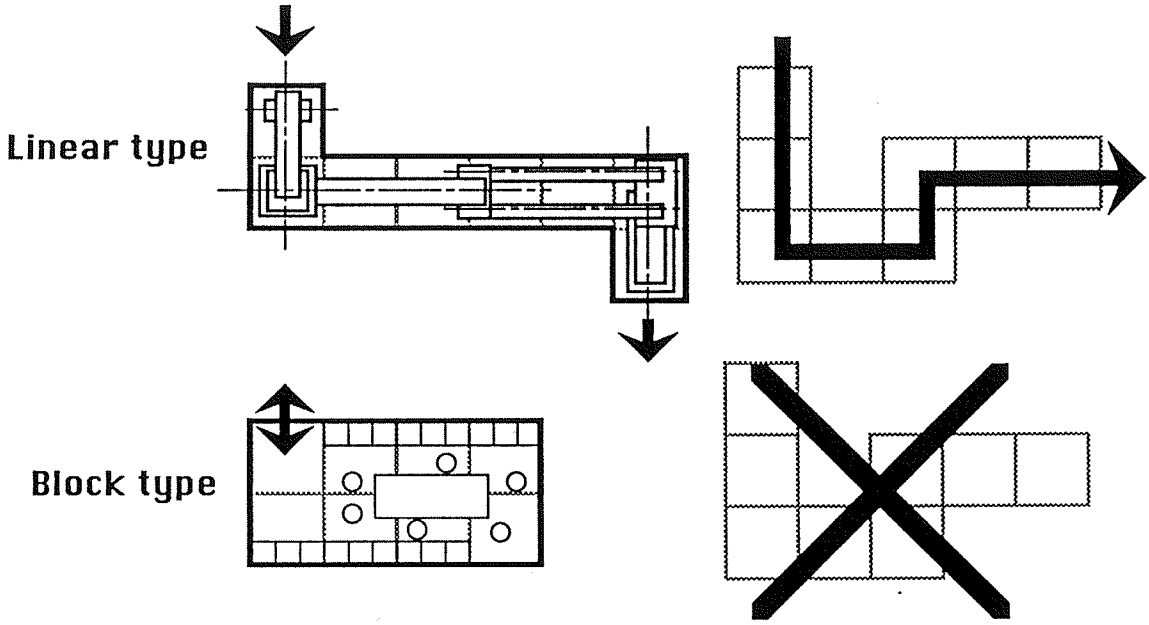


Fig.5-11

The type can be checked by the following algorithm. (see fig. 5-12)

If there is at least one room which has only one neighbor occupied with the same department, that allocation is not block type. In block type, every room has at least two neighbors who are occupied with the same department. Thus minimum number of rooms for a block type is 4. A 3 room department could only be linear.

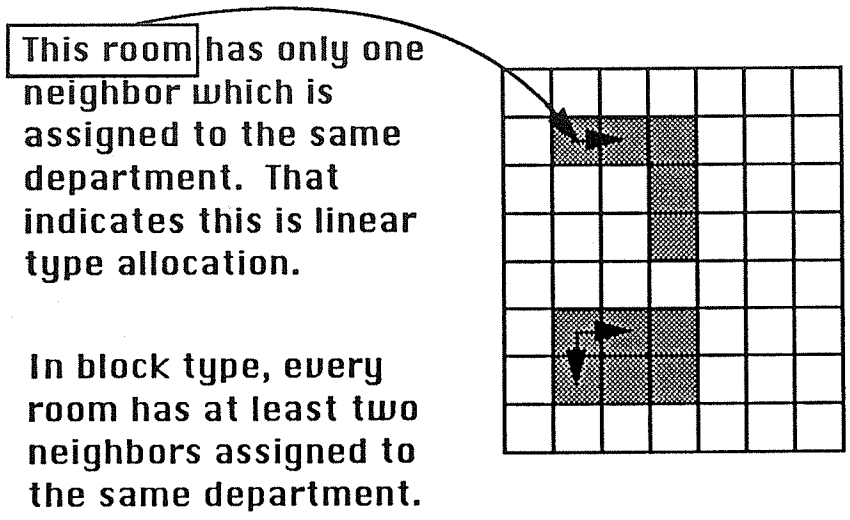


Fig.5-12

The code to check this test is as follows;

```
method blockTest.check! ()
{
    if ?self.Active == NO; return YES;

    if generalControler.CurrentDept.Type == Block;
    then
    {
    ?rooms = GetValues(generalControler.CurrentDept, Occupants);
    for ?room inlist ?rooms;
    do {
      ?neighbors = GetValues(?room, Neighbors);
      ?n = 0;
      for ?neighbor inlist ?neighbors;
      do {
        if ?neighbor.Occupant == ?room.Occupant;
          then ?n = ?n+1;
        }
      if ?n == 1; return NO;
      }
    return YES;
    }
    else return YES;
}
```

c) Access Test

This test is about special allocation related to the road.

Some departments such as a receiving department and a shipping department must be allocated next to the road, otherwise the function of those departments cannot be accomplished.

This test checks whether a department which needs access is allocated next to the road or not. If one of the rooms which were assigned to the department has a neighbor which is assigned to the road, then it is OK.

At least one of the rooms has to share a border with the road.

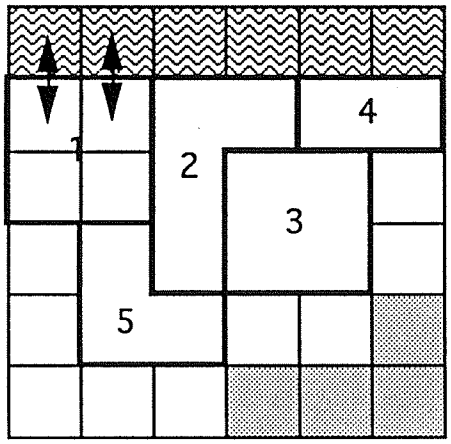


Fig.5-13

The code to check this test is as follows;

```
---------------------------------------------------------------------------------
method accessTest.check! ()
{
    if ?self.Active != YES; return YES;

    ?D = generalControler.CurrentDept;
    if ?D.needsAccess != YES; return YES;
    ?rooms = GetValues(?D, Occupants);

    for ?room inlist ?rooms;
    do{   ?neighbors = GetValues(?room, Neighbors);
        for ?neighbor inlist ?neighbors;
        do {
            if ?neighbor.Occupant == ROAD@; return YES;
              fail;
        }
    }
    return NO;
}
---------------------------------------------------------------------------------
```

d) Negative Adjacency Test

Departments which have a negative adjacency with each other cannot share a border.

For the neighbor of every room which was assigned to the department, this test checks whether the neighbor's occupant department has a negative adjacency with the department. If it has, then it fails and backtracks.

The sequence of allocation follows a positive adjacency. But still we need to check the negative adjacency. In this case if department1 and department 3 have a negative adjacency with each other, then this allocation cannot be allowed.

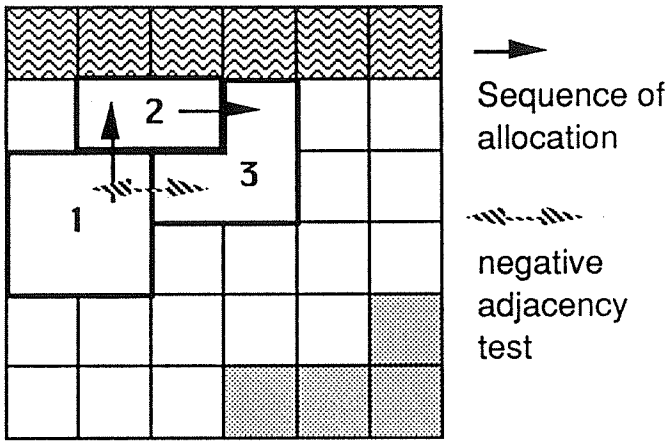


Fig.5-14

The code to check this test is as follows;

```
method negativeAdjacencyTest.check! ()
{
    ?d = generalControler.CurrentDept;
    ?rooms = GetValues(?d, Occupants);
    for ?room inlist ?rooms;
    do {
      ?neighbors = GetValues(?room, Neighbors);
      for ?neighbor inlist ?neighbors;
      do {
        ?nD = ?neighbor.Occupant;
        ?nD == find direct instanceof Departments;
        if ?d.Adjacency..?nD < 0;  return NO;
        }
      }
    return YES;
}
```

e) View Test

In SKY, the good view requirement is implemented as requirement for visibility of GREEN. When there is no obstacle between a room and GREEN, we think the room has a good view.

This test is a good example of a recursive definition.

In order to check the visibility every room should know what is seen from its four cardinal points. SKY has the method slots named "lookSouth!", "lookNorth!", "lookEast!", and "lookWest!" to let a room know what is seen.

For example, a room can know what is seen from its south window by the following method.

1) If the value of "south" slot of the room is Null (i.e., the room has no neighbor in its south direction), the method returns "DEAD" which means nothing can be seen.
2) If the value of "south" slot of the room is not Null (i.e., the room has a neighbor in its south direction), and the south neighbor is occupied with something, then the method returns the occupant of the south room.
3) If the value of "south" slot of the room is not Null (i.e., the room has a neighbor in its south direction), and the south neighbor is not occupied, then the method returns the neighbor's "lookSouth!" value.

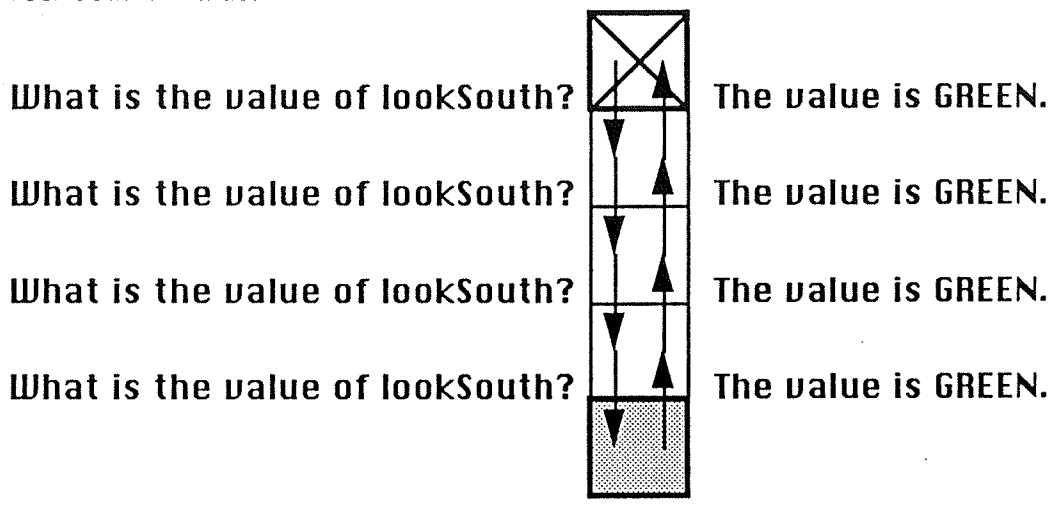


Fig.5-15

By this recursive function, every room can know what is seen from its four directions.

When a department which needs a good view has at least one room from which GREEN can be seen in any direction, this test is successful.

This test is applied not only to the current department, but also to the departments which were allocated already because we need to check if the current department blocks the previously allocated departments' view or not.

The code to check this test is as follows;

```
-------------------------------------------------------------------------------
method viewTest.check! ()
{
    if ?self.Active != YES;  return YES;

/* View Test for the Current Department.  */
    ?D = generalControler.CurrentDept;
    if ?D.needsView == YES;
    then {
        ?judge = SendMsg(?D, FindGoodViewRoom!);
        if ?judge == 0.0;
        then return NO;
        }

/* View Test for previously allocated Departments.  */
    ?pD = find direct instanceof Departments;
      ?pD != ?D;
      ?pD.needsView == YES;
      ?pDo = GetValues(?pD, Occupants);
      ?pDo != Null;
      ?pDo != `();
      ?judge2 = SendMsg(?pD, FindGoodViewRoom!);
      {
      if ?judge2 == 0.0;
        then return NO;
        else fail;
       }
    return YES;
  return Null;
}

method Rooms.LookSouth! ()
{
     if ?self.South == Null;
     then return DEAD;
     else
        if ?self.South.Occupant == Null;
        then return SendMsg(?self.South, LookSouth!);
        else return ?self.South.Occupant;
}
-------------------------------------------------------------------------------
```

f) Sunlight Test

Some departments such as a cafeteria need to be adjacent to outer space to get sunlight or to have a window.

If at least one of the rooms which are assigned to the department has an empty neighbor, or neighbor which is pre-occupied with GREEN or ROAD , then this test is successful. This test is applied not only to the current department but also to the previously allocated departments to check if the current department does not block the other's outer window.

The code to check this test is as follows;

```
-------------------------------------------------------------------------------
method sunlightTest.check! ()
{
```

```
    if ?self.Active != YES;  return YES;

/* Sunlight Test for the Current Department.  */
    ?D = generalControler.CurrentDept;
    if ?D.needsSunlight == YES;
    then {
        ?judge = SendMsg(?D, FindSunlightRoom!);
        if ?judge == 0.0;  return NO;
        }

/* Sunlight Test for the previously allocated departments. */
    ?pD = find direct instanceof Departments;
    ?pD != ?D;
    ?pD.needsSunlight == YES;
    ?pDo = GetValues(?pD, Occupants);
    ?pDo != Null;
    ?pDo != `();
    ?judge2 = SendMsg(?pD, FindSunlightRoom!);
        {
        if ?judge2 == 0.0;
          then return NO;
          else fail;
        }

    return YES;
}


method Departments.FindSunlightRoom! ()
{
  ?sunlightRoom = 0.0;
  ?rooms = GetValues(?self, Occupants);
  for ?room inlist ?rooms;
  do {
    ?empty = SendMsg(?room, LookAround!);
    if ?empty >0;
      then ?sunlightRoom = ?sunlightRoom +1.0;
    }
  return ?sunlightRoom;
}

method Rooms.LookAround! ()
{
  ?neighbors = GetValues(?self, Neighbors);
  ?empty = 0.0;
  for ?neighbor inlist ?neighbors;
  do {
    if ?neighbor.Occupant == Null;
      then ?empty = ?empty + 1.0;
      else if ?neighbor.Preoccupant != Null;
              then ?empty = ?empty +1.0;
    }

  return ?empty;
}
```

C. Evaluation

Evaluation provides a measure of the quality of a layout. There are two important parts in how to make an evaluation.

1. Choosing parameters to consider in evaluation. When we think a layout is good, what features of the layout make us think so? It is very important for good evaluation to set appropriate parameters. In SKY, 6 parameters can be set to measure the value of alternatives. They are "Material handling cost", "Construction cost", "Flexibility", "The number of sunlight rooms", "The number of rooms with access to the road" and "The number of rooms with good view".

2. Weight for each factor. In some cases material handling cost is most important, and in some other cases construction cost is most important, and so on. Human planners judge the situation of the project to decide the weight of each evaluation criteria. SKY calculates the total value of an alternative according to the user's preference weights .

1. Criteria

a) Material handling cost

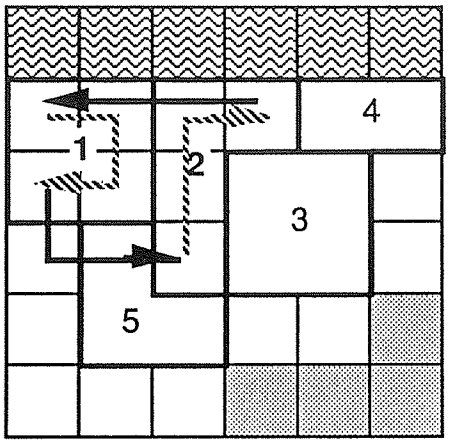
The From/To chart shows the information about the material flow from a department to another department. As it is discussed in chapter 3, the numbers in table of Fig. 5-16 show the quantity of the flow and difficulty of the handling. The bigger a number is, the more frequently materials are carried and the more difficult it is to carry them. So a big number indicates a greater positive adjacency between two departments.

Department name	Material Flow				
	to D1	to D2	to D3	to D4	to D5
Department-1		98	3	12	8
Department-2	4		88	21	12
Department-3	11	15		56	11
Department-4	2	5	8		87
Department-5	45	12	23	22	

Fig.5-16

After generating a layout, a distance chart can be made. The table in fig 5-17 shows the information about the distances between each department. The distance is measured from an exit of a department to the entrance of another department as a rectilinear distance. For example the distance from A department to B department is given by the formula

Dis(A,B) =abs((X of A's Exit)-(X of B's Entrance)) + abs((Y of A's Exit)-(Y of B's Entrance))



Plain arrow shows the distance between two departments.

Dashed arrows indicate inner flow of each departments

Department name	Distance				
	to D1	to D2	to D3	to D4	to D5
Department-1		3	3	5	2
Department-2	3		1	1	4
Department-3	4	3		1	4
Department-4	6	5	3		6
Department-5	5	1	3	5	

Fig.5-17

The reason why this chart is not symmetric is that we consider the entrance and the exit. The entrance is the room which was allocated first to the department. The exit is the room allocated last. The order of allocation during the search procedure indicates the material flow inside the department.

When the two figures in the same cell of these two charts (Material flow chart and distance chart) are multiplied and the results of all cells are summed up, we can get a number which represents a handling cost.

The code to calculate this value is as follows;

```
method MaterialHandling.Evaluate! ()
{
    ?Eval = 0.0;
    ?Evaluation = 0.0;
    for ?D1 = find direct instanceof Departments;
    do {
        ?D1.NoOfRooms > 0;
        find  count ?D1.Occupants > 0 ;
        sum ?Evaluation into ?Eval;
        for ?D2 = find direct instanceof Departments;
        do {
            ?D2 != ?D1;
            ?D2.NoOfRooms >0;
            find count ?D2.Occupants >0;
            ?re = ?D1.MaterialFlow..?D2;
```

```
                ?e = CalDis!(?D1, ?D2)* ?re;
                sum ?e into ?Evaluation;
            }
        }
      ?Eval = ?Eval+?Evaluation;
    ?self.Value = ?Eval;
   return Null;
}
```

```
function CalDis!(?D1, ?D2)
{
    bound inputs;
        ?x1 = ?D1.LastRoom.X;
        ?y1 = ?D1.LastRoom.Y;
        ?x2 = ?D2.FirstRoom.X;
        ?y2 = ?D2.FirstRoom.Y;
        ?Distance = Abs(?x1-?x2) + Abs(?y1-?y2);
    return ?Distance;
}
```

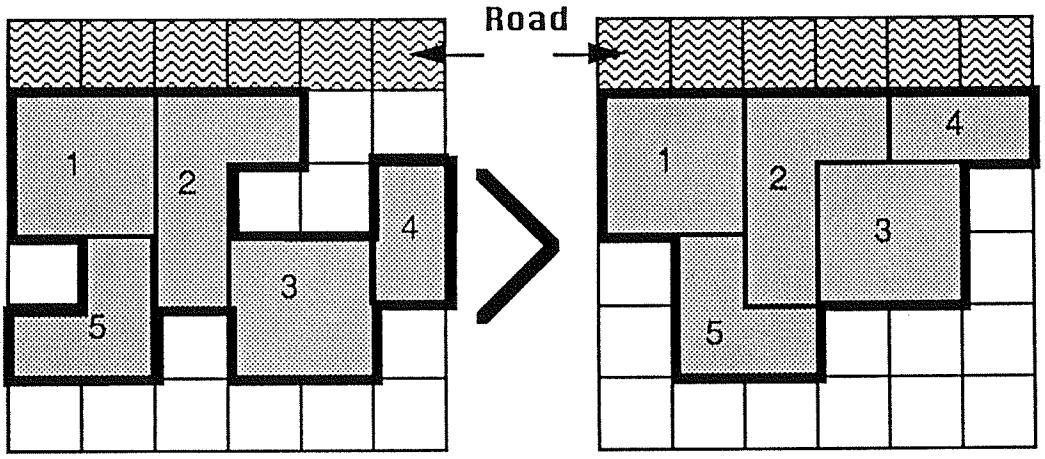
---

b) Construction Cost (Length of external walls)

Construction cost is approximated by the length of the external walls.

If the total areas are same, a square is most cost effective layout. The more complicated a layout is, the longer the external wall will be and the more expensive it is to build the building.

The length of the external walls can be calculated by counting how many empty neighbors each room has.



The construction cost of the left plan is more expensive than that of the right plan because the left plan has longer external wall.

Fig.5-18

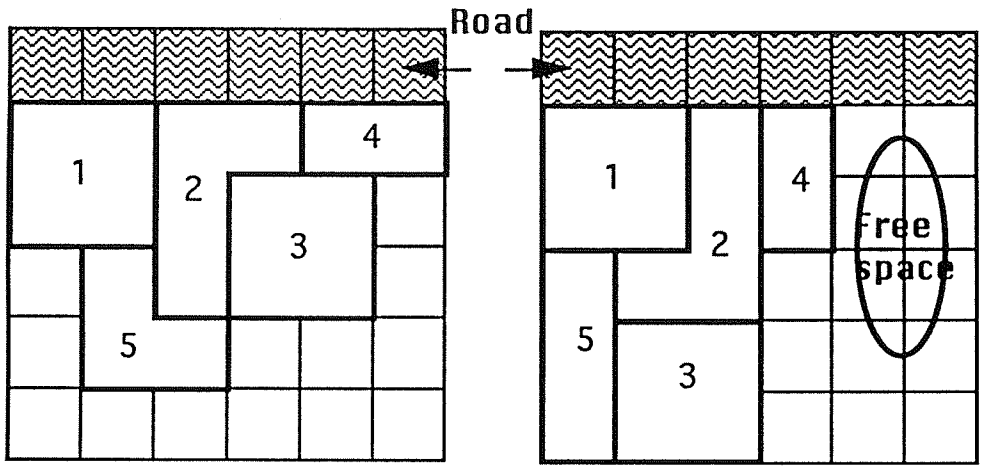
The code to calculate this value is as follows;

```
method LengthOfWalls.Evaluate! ()
{
   ?eval = 0.0;
   for ?r = find direct instanceof Rooms;
   do {
     ?r.Preoccupant != ROAD@;
     ?r.Preoccupant != GREEN@;
     ?r.Occupant != Null;
     ?value = SendMsg(?r,LookAround!)+ 4 - ?r.numberOfNeighbors;
     sum ?value into ?eval;
      }

   ?self.Value = ?eval;
  return Null;
}
```

c) Flexibility (The number of empty rooms adjacent to the road)

It is very difficult to evaluate the flexibility of a layout. SKY evaluates the flexibility using a simple initial measure of flexibility. (See Fig. 5-19)



The right plan has more flexibility than the left plan because it has more free space which can access the road.

Fig.5-19

The code to calculate this value is as follows;

```
method LengthOfFreeRoad.Evaluate! ()
  {
    ?eval = 0.0;
    for ?r = find direct instanceof Rooms;
    do {
      if ?r.Occupant != ROAD@; then fail;
      ?n = GetValues(?r, Neighbors);
      for ?nei inlist ?n;
      do {
         if ?nei.Occupant == Null;
         then ?eval = ?eval + 1.0;
         else {
            if ?nei.Occupant == GREEN@;
              then ?eval = ?eval +1.0;
              else fail;
              }
        }
       }
    ?self.Value = ?eval;
   return Null;
  }
```

d) The number of sunlight rooms

Usually we want as many rooms with windows as possible.

SKY lists the departments which need sunlight and measures the length of external walls of rooms occupied by those departments.

The code to calculate this value is as follows;

```
method NoOfSunlightRooms.Evaluate! ()
{
   ?eval = 0.0;
   for ?D = find direct instanceof Departments;
   do {
     if ?D.needsSunlight != YES; then fail;
     ?rooms = GetValues(?D, Occupants);
     for ?r inlist ?rooms;
     do {
        ?e = SendMsg(?r, LookAround!);
        sum ?e into ?eval;
      }
     }
   ?self.Value = ?eval;
  return Null;
}
```

e) The number of rooms with Access to the road

SKY picks up the departments which need access to the road and measures the length of external walls of rooms which share a border with the road.

The bigger the number is, the more freely we can make inside detailed plans so the layout seems to be good.

The code to calculate this value is as follows;

```
method NoOfAccessRooms.Evaluate! ()
{
     ?eval = 0.0;
     for ?D = find direct instanceof Departments;
     do {
       if ?D.needsAccess != YES; then fail;
       else
       ?rooms = GetValues(?D, Occupants);
         for ?room inlist ?rooms;
         do{ ?neighbors = GetValues(?room, Neighbors);
            for ?neighbor inlist ?neighbors;
            do {
               if ?neighbor.Occupant == ROAD@;
                then ?eval = ?eval + 1.0;
               fail;
              }
          }
        }
   ?self.Value = ?eval;
  return Null;
```

f) The number of rooms with good view

SKY picks up the departments which need good view and counts the number of rooms which have a good view in those departments.

The bigger the number is, the better the layout is.

The code to calculate this value is as follows;

```
method NoOfGoodViewRooms.Evaluate! ()
{
      ?eval = 0.0;
      for ?d = find direct instanceof Departments;
      do {
        if ?d.needsView != YES; then fail;
        else
        ?value = SendMsg(?d,FindGoodViewRoom!);
        sum ?value into ?eval;
        }
   ?self.Value = ?eval;
  return Null;
}

method Departments.FindGoodViewRoom! ()
{
   ?goodView = 0.0;
   ?e = SendMsg(?self, FindEastViewRoom!);
   ?n = SendMsg(?self, FindNorthViewRoom!);
   ?s = SendMsg(?self, FindSouthViewRoom!);
   ?w = SendMsg(?self, FindWestViewRoom!);
   ?goodView = ?e + ?n + ?s + ?w;
   return ?goodView;
}

method Departments.FindSouthViewRoom! ()
{
   ?view = 0.0;
     ?rooms = GetValues(?self, Occupants);
     for ?room inlist ?rooms;
     do {
       ?east = SendMsg(?room, LookSouth!);
       if ?east == GREEN@;
         then ?view = ?view + 1.0;
       }
     return ?view;
  return Null;
}
```

g) Modularity of the criteria

As modularity of evaluation criteria is very important, the system should be open to a new evaluation criterion. When we find a new parameter to evaluate a layout, this factor should be added easily to the system. SKY accept a new criterion very easily. All we have to do is to add a new instance of a class object named "Evaluation" as fig. 5-20 shows.

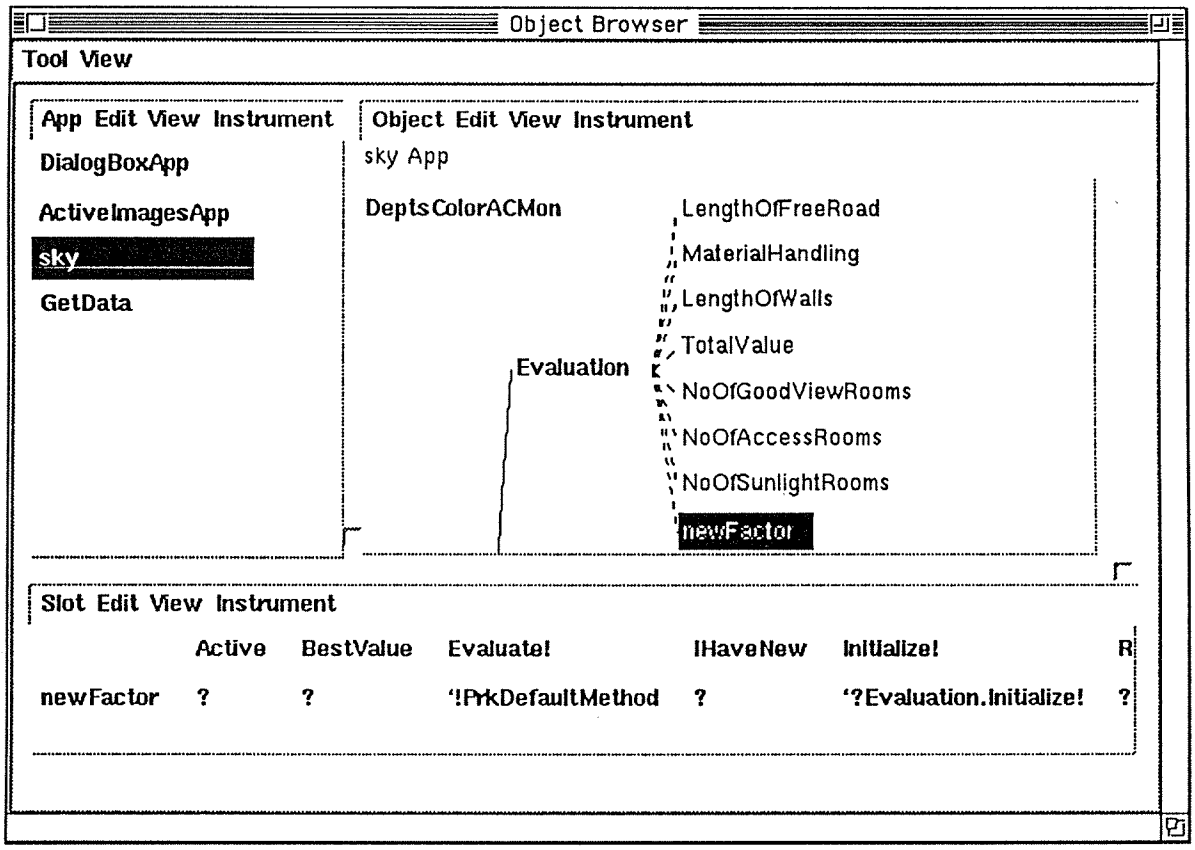


Fig.5-20

A new criteria can be easily added

2. Total value

In order to calculate the total value of each alternative, we have two problems to solve. The first thing is value normalization and the second one is setting weights of criteria.

a) Normalization of the value

The values which are discussed in the former part of this chapter have two types. For the ascending type, the smaller the number is the better it is. For the descending type, the bigger the number is the better it is. For instance the material handling cost is ascending type and the number of sunlight rooms is descending type.

Normalization of the values is done as fig. 5-20. For every criteria, the best value becomes 100, worst value becomes 0, and every value ranges from 0 to 100.

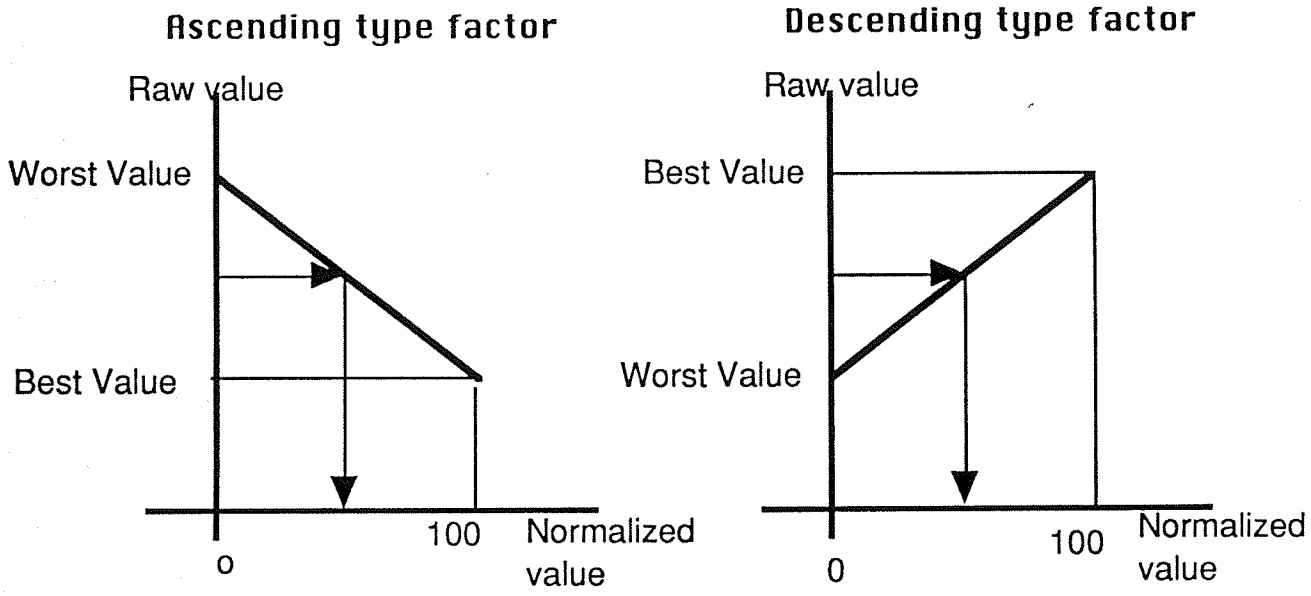


Fig.5-21

Normalization of the value

The code to calculate the total value is as follows;

```
---------------------------------------------------------------------------------------------
method General.CalculateValue! ()
{
   /*--- Measure range for normalization----------------------*/
   TotalValue.Weight =0;
   for ?e = find instanceof Evaluation;
   do {
            ?e.Active == NO;
            ?e.Weight = 0;
     }
   for ?e = find instanceof Evaluation;
   do{
    ?e != TotalValue;
    ?e.Active == YES;
    TotalValue.Weight = TotalValue.Weight + ?e.Weight;
```

```
    if ?e.Type == A;
    then{
            ?e.Range = ?e.WorstValue - ?e.BestValue;
       }
    if ?e.Type == D;
    then {
            ?e.Range = ?e.BestValue - ?e.WorstValue;
       }
     }
  /*--- End of Measure--------------------------------*/

  /*--- Calculate Value.---------------------- */
  for ?r = find direct instanceof Results;
  do SendMsg(?r, EvaluateMe!);

  return Null;
}

method Results.EvaluateMe! ()
{
  /*--- Transfer to normalized value. ----------------------------*/
  for ?e = find instanceof Evaluation;
  do {
    ?e != TotalValue;
    ?e.Active == YES;
    if ?e.Type == A;
    then {
            if ?e.Range != 0; then
            ?self.Evaluation..?e = (?e.WorstValue-?self.?e)/?e.Range * 100;
            else ?self.Evaluation..?e =50;
            }
   if ?e.Type == D;
   then {
            if ?e.Range != 0; then
            ?self.Evaluation..?e = (?self.?e-?e.WorstValue)/?e.Range *100;
            else ?self.Evaluation..?e =50;
       }
    }
  /*--- End of Value Normalization ----------------------------*/

  /*--- Calculate Final Value considering preference factor. ------*/
  ?value = 0;
  for ?e = find direct instanceof Evaluation;
  do {
    ?e != TotalValue;
    ?e.Active == YES;
    ?v = ?self.Evaluation..?e * ?e.Weight;
    sum ?v into ?value;
    }
  ?self.FinalValue = ?value/TotalValue.Weight;
  /*--- End of Calculation. ---------------------------------------*/
  return Null;
}
```

---

b) Preference factor and total value

The total value is calculated as a weighted average by the formula

$$\text{Total value} = \sum(normalizedValue * weight) \Big/ \sum weight$$

By this formula total value has the value between 0 and 100. The weight represents the preference of the planner, and is entered by him. The planner can get the best result according to his preferences.

Fig. 5-22 shows an example of normalized value for each alternative.

	Alternative-A	Alternative-B	Alternative-C
Material Handling Cost	90	50	60
Construction Cost	50	90	20
Flexibility	20	60	30
Number of Sunlight rooms	50	80	80
Number of Good View rooms	30	60	70
Number of Access rooms	50	50	60

Fig. 5-22

Example of normalized values for each alternative

Fig. 5-22 says that Alternative-A is very good at material handling and Alternative-B is good at construction cost. How can we decide which one is the best solution? Shall we sum up these values? Absolutely no. In order to choose the best solution, weight for each evaluation criteria is needed. Fig. 5-23 is an example of weights setting.

	Case-1	Case-2	Case-3
Material Handling Cost	95	10	10
Construction Cost	10	95	10
Flexibility	0	0	10
Number of Sunlight rooms	10	10	95
Number of Good View rooms	50	10	80
Number of Access rooms	10	10	50

Fig. 5-23

Example of weights for evaluation criteria

Cases in Fig. 5-23 represent situation of a project. For instance, in Case-1 material handling cost is crucial for the project and in Case-2 construction cost is most important. The value of an alternative should be decided according to these weights. Fig. 5-24 shows the results of calculation of final value.

	Alternative-A	Alternative-B	Alternative-C
Case-1	66.00	56.86	61.71
Case-2	51.48	81.11	34.07
Case-3	44.12	66.27	67.84

Fig. 5-24

Example of final values according to the preference factors

As you can see, in Case-1 Alternative-A is the best result and in Case-2 Alternative-B is best.

c) How to make records of the results

After generating an alternative layout, we must decide if the result is worth recording or not. In SKY the best 10 results for every evaluation factor are recorded.

	result-a	result-b	result-c	result-d
Material handling cost		1	2	
Construction cost	1		6	4
Flexibility		2	1	
Sunlight rooms	3			4
Access rooms		3		1
Good view rooms			1	2
Total		1		2

When a result is included in best 10 of at least one evaluation factor, the result is recorded.

Fig.5-25

At this stage we cannot calculate the total value because we do not know the prèference factor nor the best and worst value of each evaluation factor. But we need to evaluate the total value tentatively in order to ensure that every good result is recorded. Tentative total value is calculated assuming the preferences are all same

	result-i	result-j	result-k	result-l
Material handling cost	100	0	0	80
Construction cost	0	100	0	80
Flexibility	0	0	100	80
Sunlight rooms	0	0	0	80
Access rooms	0	0	0	80
Good view rooms	0	0	0	80
Total				

We must consider the tentative total value, otherwise we will miss a result like result-l.

Fig.5-26

VI. Conclusion, Benefit of this system

SKY has three significant features.

First, it is complete. All layout options are tried and evaluated by exhaustive search. Completeness is one of the significant features that the computer allows. The computer can find some solution alternatives that will never occur to a human designer.

Second, it is modular and extensible. New knowledge, such as new search pruning knowledge, a new evaluation criteria and new adjacency knowledge etc. , is easily added to SKY. The benefit of the modularity is that we can apply SKY to another kind of facility. For example, a hospital operating room has a requirement "needsClean", and a reception room has a requirement "needsAccess". Because "needsClean" is an incompatible feature of "needsAccess", an operating room and a reception room should have negative adjacency. When we implement this kind of knowledge about a hospital layout, SKY can be a layout system for a hospital.

Third, it is clear. It is very clear for the user to see what SKY does and does not do. The SKY user interface is designed to make it clear what is needed to make an effective layout. The SKY output is also very clear. SKY identifies real requirements to help separate important ones from those that are less important or even wrong, as well as it gives a high quality layout.

VII. Further Effort

As SKY is a very flexible system, there are many potential future developments for it.

A. Integration with Facility Management

Each department has its equipment to accomplish its function. By adding functions to infer the departments' features from their equipment, SKY can become a more sophisticated system. Functions to infer the necessary space of each department from its equipment list, and functions to infer the type (Block or Linear) from equipment are also desirable. The information about each department's equipment is stored by a database through the whole life time of a facility. Integration with this database is strongly desired.

B. Addition of 3D features

The concept of adjacency in 2D can be extended to top and bottom relationships in 3D. It can be called "vertical adjacency". When the production process uses gravity, the sequence of the production implies top and bottom relationships (vertical adjacency). The load of each department also implies top and bottom relationships. A requirement of a department whose ceiling needs to be high can be implemented as the number of vertical rooms it needs. There are many clues to discuss about 3D layout problem. Rooms in SKY have only planar information about their neighbors right now. By adding slots named "Above", "Below" to the room object, we can implement 3D features in SKY. And then SKY can be applied to a multi-floor factory.

C. Addition of dimensional information

Some building code requires that some material should be stored in a proper distance from other operations. A grid in SKY does not now have any information about dimension. If SKY knows how long one grid is, then we can solve the dimension requirement of building code.

D. Suggestion of material handling route and handling method

This suggestion is a sort of diagnosis . The purpose of this diagnosis is to minimize the traffic crossing and to select the best handling method according to the length of the route and the frequency of handling.

E. Development of several types of adjacency definition package

House version, Hospital version, Furniture plant version, etc.

F. Refine evaluation parameter

There may be other evaluation criteria and criteria may have non-linear interactions.

VIII. Appendix -- Source code by ProTalk

```
/*
 *
 * ProTalk method source file
 *
 */


#include <prk/lib.pth>

bcrule ad_rule1 in adjacencyRules
{if:
        ?D2 == instanceof Departments;
        ?D2 != ?D1;
        ?f1 = ?D1.Features;
        ?f2 = ?D2.Features;
        ?f1.incompatibleFeatures == ?f2;
then:
        ?D1.negativeAdjacencies +== ?D2;
        Print ("\n", ?D1, "negativeAdjacencies:", ?D2);
}

bcrule po_rule1 in positiveAdRules
{if:
        ?D2 == instanceof Departments;
        ?D1 != ?D2;
        ?f1 = ?D1.Features;
        ?f2 = ?D2.Features;
        ?f1 == ?f2;
        ?D1.negativeAdjacencies != ?D2;
then:
        ?D1.positiveAdjacencies +== ?D2;
        Print ("\n", ?D1, "positiveAdjacencies:", ?D2);
}

bcrule po_rule2 in positiveAdRules
{if:
      ?D2 == instanceof Departments;
      ?D1 != ?D2;
      ?f1 = ?D1.Features;
      ?f2 = ?D2.Features;
      ?f1.compatibleFeatures == ?f2;
      ?D1.negativeAdjacencies != ?D2;
then:
      ?D1.positiveAdjacencies +== ?D2;
      Print ("\n", ?D1, "positiveAdjacencies:", ?D2);
}


function DominantD (?list)
{
      ?newList = SortBySlot(?list, NoOfStranger, ">");
      ?dD = ListFirst(?newList);
      return ?dD;
```

```
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.setAdjacency!
 * Note:  This method should be called from the class object, since it
 * maps down all Departments.
 *
 */

method Departments.setAdjacency! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        find  [adjacencyRules] ?D2 = ?self.negativeAdjacencies;
        find  [positiveAdRules] ?D2 = ?self.positiveAdjacencies;


   return Null;
}

/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.lookAdjacency!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.lookAdjacency! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        for ?D = find direct instanceof Departments;
        do   SendMsg(?D,AverageFlow!);
        for ?D = find direct instanceof Departments;
        do  SendMsg(?D,findNegative!);
        for ?D = find direct instanceof Departments;
        do   SendMsg(?D,findPositive!);
```

```
        ?dD = SendMsg(?self, Dominant!);
        ?self.DominantDept = ?dD;
        ?self.NumberOfOrphans = ?dD.NoOfStranger;

        if ?self.Mode == StepByStep; then return Null;

        while ?dD.NoOfStranger >0;
        do {
                ?orphanList = GetValues(?dD, Strangers);
                ?oD = DominantD(?orphanList);
                ?oDnegativeList = GetValues(?oD,negativeAdjacencies);
                for ?dept = find direct instanceof Departments;
                do {
                        ?dept != ?oD;
                        ?dept.MaterialFlow..?oD > ?dept.MaterialFlowAverage;
                        ?judge = Member_of(?dept,?oDnegativeList);
                        ?judge != -1;

                        ?dept.positiveAdjacencies +== ?oD;
                   }

                ?dD = SendMsg(?self, Dominant!);

                if ?self.NumberOfOrphans == ?dD.NoOfStranger;
                then {
                        SendMsg(CantGoAnmyMoreDialogBox,PutOnScreenAndWait!);
                        return Null;
                   }

                ?self.DominantDept = ?dD;
                ?self.NumberOfOrphans = ?dD.NoOfStranger;
           }
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.findNegative!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.findNegative! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
```

```
        ClearValues(?self,negativeAdjacencies);
/*      find  [adjacencyRules] ?D2 = ?self.negativeAdjacencies;  **
        ---by this code we can get only one solution.
*/
        for ?D2 = find [adjacencyRules] ?self.negativeAdjacencies;
        do ;

   return Null;
}


/*-------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.findPositive!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method Departments.findPositive! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        ClearValues(?self,positiveAdjacencies);
/*      find  [positiveAdRules] ?D2 = ?self.positiveAdjacencies; **
        ---This code can get only one solution.              */
        for ?D2 = find [positiveAdRules] ?self.positiveAdjacencies;
        do ;

   return Null;
}


/*-------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.lookForOrphan!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method DeptControler.lookForOrphan! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
```

```
        "\n  object ", ?self,
        "\n  slot   ", ?slot,
        "\n");

      /*
      ?self.AllDepts = all direct instanceof Departments;
      -----this expression makes list=(value,value,,) not MultiValue.
      */

      for ?D = find direct instanceof Departments;
      do {
      ?self.AllDepts +== ?D;
      ClearValues(?D,Strangers);
        }

      for ?D = find direct instanceof Departments;
      do {
            /* initialize of orphans.  */
            ?DeptsList = GetValues(?self,AllDepts);
            for ?dept inlist ?DeptsList;
            do{
             ?self.OrphanDepts +== ?dept;
             }

            SendMsg(?D,GetRidOfMe!);

            ?D.NoOfStranger = find count ?self.OrphanDepts;
            ?OrphanList = GetValues(?self,OrphanDepts);
            for ?orphan inlist ?OrphanList;
            do{
            ?D.Strangers +== ?orphan;
            }

        }

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.GetRidOfMe!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.GetRidOfMe! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
```

```
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");

        if GetValues(deptControler,OrphanDepts) == Null; return Null;
        deptControler.OrphanDepts -== ?self;
        ?nD = find ?self.positiveAdjacencies;
        ?nD == find deptControler.OrphanDepts;
        SendMsg(?nD,GetRidOfMe!);
        fail;

    return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.Dominant!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method DeptControler.Dominant! ()
{
    /* Methods must always have all their inputs bound: */
    bound inputs;

    Print ("\nMethod:",
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");
        SendMsg(?self,lookForOrphan!);

        ?list = all instanceof Departments;
        ?newList = SortBySlot(?list, NoOfStranger, ">");
        ?dD = ListFirst(?newList);
        return ?dD;

}

/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.SetDeptFeatures!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method DeptControler.SetDeptFeatures! ()
{
    /* Methods must always have all their inputs bound: */
```

```
    bound inputs;

    Print ("\nMethod:",
           "\n  object ", ?self,
           "\n  slot   ", ?slot,
           "\n");
          ClearValues(FeaturesInDialogBox,SelectionItems);
          for ?feature = find direct instanceof Features;
          do {
             FeaturesInDialogBox.SelectionItems +== ?feature;
             }

          for ?d = find direct instanceof Departments;
          do {
            DeptsInDialogBox.SelectionItems +== ?d;
            }

          SendMsg(SetDeptFeatureDialogBox,PutOnScreenAndWait!);

    return Null;
}

method adjacency_CommandRow_19.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;

    ?DeptNames_value  =
          GetControlValue (?dialog_box, `DeptNames);
    ?FeatureNames_values  =
          GetControlValues (?dialog_box, `FeatureNames);
    ?DnFeatures = GetControlValues(?dialog_box, `DnFeatures);

    select {
          case: ?button == "Show";
                {
                ClearValues(DnFeatureInDialogBox,SelectionItems);
                ?fs = GetValues(?DeptNames_value,Features);
                if ?fs == Null; ?fs = `();
                for ?f inlist ?fs;
                do {
                  DnFeatureInDialogBox.SelectionItems +== ?f;
                  }
                }
       case: ?button == "Set";
                {
           Print (?button, "\n");
                for ?f inlist ?FeatureNames_values;
                do {
                  ?DeptNames_value.Features +== ?f;
                  }
                ClearValues(DnFeatureInDialogBox,SelectionItems);
           ?fs = GetValues(?DeptNames_value,Features);
```

```
            if ?fs == Null; ?fs = `();
          for ?f inlist ?fs;
          do {
            DnFeatureInDialogBox.SelectionItems +== ?f;
            }
              }
      case: ?button == "Remove";
              {
        Print (?button, "\n");
              if ?DnFeatures != Null; then
              {
              for ?dnf inlist ?DnFeatures;
              do {
                ?DeptNames_value.Features -== ?dnf;
                }
              ClearValues(DnFeatureInDialogBox,SelectionItems);
              ?fs = GetValues(?DeptNames_value,Features);
              if ?fs == Null; ?fs = `();
          for ?f inlist ?fs;
          do {
            DnFeatureInDialogBox.SelectionItems +== ?f;
            }
              }
              }
      case: ?button == "Done";
              {
        Print (?button, "\n");
              SendMsg (?dialog_box, TakeOffScreen!);
              }
        case: ?button == "Reset";
              {
              ClearValues(?DeptNames_value,Features);
              ClearValues(DnFeatureInDialogBox,SelectionItems);
         ?fs = GetValues(?DeptNames_value,Features);
              if ?fs == Null; ?fs = `();
          for ?f inlist ?fs;
          do {
            DnFeatureInDialogBox.SelectionItems +== ?f;
            }
              }
   }

   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.ShowResult!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
```

```
 */

method DeptControler.ShowResult! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        ClearValues(DeptsInAdjDialogBox,SelectionItems);
        ClearValues(TargetDepts,SelectionItems);
        ClearValues(NegativeList,SelectionItems);
        ClearValues(PositiveList,SelectionItems);

        for ?d = find direct instanceof Departments;
      do {
        DeptsInAdjDialogBox.SelectionItems +== ?d;
          TargetDepts.SelectionItems +== ?d;
        }

      SendMsg(AdjacencyListDialogBox,PutOnScreenAndWait!);

   return Null;
}

method adjacency_CommandRow_22.React!  (?button)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
   ?dialog_box = ?self.DialogBox;

   ?NegativeLists_values =
         GetControlValues (?dialog_box, `NegativeLists);
   ?DeptsNames_value =
         GetControlValue (?dialog_box, `DeptsNames);
   ?PositiveLists_values =
         GetControlValues (?dialog_box, `PositiveLists);
   ?TargetDepts_value =
         GetControlValue (?dialog_box, `TargetDepts);

   select {
      case: ?button == "ShowAdjacency";
         { Print (?button, "\n");
               if ?DeptsNames_value != Null; then
               {
               ClearValues(NegativeList,SelectionItems);
               ?ds = GetValues(?DeptsNames_value,negativeAdjacencies);
               if ?ds != Null;
               then {
               for ?d inlist ?ds;
               do NegativeList.SelectionItems +== ?d;
                  }
```

```
            ClearValues(PositiveList,SelectionItems);
            ?ds = GetValues(?DeptsNames_value,positiveAdjacencies);
            if ?ds != Null;
            then {
            for ?d inlist ?ds;
            do PositiveList.SelectionItems +== ?d;
                }

            ClearValues(DnsStrangerList,SelectionItems);
            ?ds = GetValues(?DeptsNames_value,Strangers);
            if ?ds != Null;
            then {
                for ?d inlist ?ds;
                do DnsStrangerList.SelectionItems +== ?d;
                }
            }
        }

case: ?button == "SetNegative";
  {  Print (?button, "\n");
            if ?DeptsNames_value != Null; then
            {
            ?DeptsNames_value.negativeAdjacencies +== ?TargetDepts_value;
            ClearValues(NegativeList,SelectionItems);
     ?ds = GetValues(?DeptsNames_value,negativeAdjacencies);
     if ?ds != Null;
     then {
     for ?d inlist ?ds;
     do NegativeList.SelectionItems +== ?d;
         }
          }
     }

case: ?button == "DeleteNegative";
  {  Print (?button, "\n");
            if ?DeptsNames_value != Null; then
            {
            if ?NegativeLists_values != Null; then
             {
            for ?tD inlist ?NegativeLists_values;
            do {
            ?DeptsNames_value.negativeAdjacencies -== ?tD;
              }
             }
        ClearValues(NegativeList,SelectionItems);
     ?ds = GetValues(?DeptsNames_value,negativeAdjacencies);
     if ?ds != Null;
     then {
     for ?d inlist ?ds;
     do NegativeList.SelectionItems +== ?d;
         }
          }
     }
```

```
        case: ?button == "SetPositive";
           { Print (?button, "\n");
                 if ?DeptsNames_value != Null; then
                 {
              ?DeptsNames_value.positiveAdjacencies +== ?TargetDepts_value;
              ClearValues(PositiveList,SelectionItems);
            ?ds = GetValues(?DeptsNames_value,positiveAdjacencies);
            if ?ds != Null;
            then {
            for ?d inlist ?ds;
            do PositiveList.SelectionItems +== ?d;
                }
                 }
            }

        case: ?button == "DeltePositive";
           { Print (?button, "\n");
                 if ?DeptsNames_value != Null; then
                 {
                  if ?PositiveLists_values != Null;
                  then{
                     for ?tD inlist ?PositiveLists_values;
                     do {
               ?DeptsNames_value.positiveAdjacencies -== ?tD;
                           }
                      }
                 ClearValues(PositiveList,SelectionItems);
            ?ds = GetValues(?DeptsNames_value,positiveAdjacencies);
            if ?ds != Null;
            then {
            for ?d inlist ?ds;
            do PositiveList.SelectionItems +== ?d;
                }
                 }
            }
        case: ?button == "Done";
           { Print (?button, "\n");
                 SendMsg (?dialog_box, TakeOffScreen!);
            }
    }

    return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.DataImport!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */
```

```
method DeptControler.DataImport! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

       SendMsg(DataImportDialogBox,PutOnScreenAndWait!);

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.AddPositive!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.AddPositive! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       if ?self.NumberOfOrphans <= 0; then return Null;

        ?orphanList = GetValues(?self.DominantDept, Strangers);
     ?oD = DominantD(?orphanList);
     ?oDnegativeList = GetValues(?oD,negativeAdjacencies);

     for ?dept = find direct instanceof Departments;
     do {
          ?dept != ?oD;
          ?dept.MaterialFlow..?oD > ?dept.MaterialFlowAverage;
          ?judge = Member_of(?dept,?oDnegativeList);
          ?judge != -1;
       ?dept.positiveAdjacencies +== ?oD;
       }

     ?dD = SendMsg(?self, Dominant!);

       if ?self.NumberOfOrphans == ?dD.NoOfStranger;
       then { SendMsg(CantGoAnyMoreDialogBox,PutOnScreenAndWait!);
```

```
            return Null;
           }

      ?self.DominantDept = ?dD;
      ?self.NumberOfOrphans = ?dD.NoOfStranger;

  return Null;
}


/*---------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.DataExport! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

       SendMsg(DataExportDialogBox,PutOnScreenAndWait!);

  return Null;
}


/*---------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.AverageFlow!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.AverageFlow! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
```

```
      ?n = find count direct instanceof Departments;
      ?flow = 0;
      for ?d = find direct instanceof Departments;
      do {
        ?d != ?self;
        ?f = ?self.MaterialFlow..?d;
        sum ?f into ?flow;
      }
      ?self.MaterialFlowAverage = ?flow/?n;

  return Null;
}


/*-------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.AdjToLayout!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.AdjToLayout! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
/*---This Method transfers data from Adjacency to Layout.-------------*/

      /*---Transfer of Dept Features.-----------------------------*/
      for ?d = find direct instanceof Departments;
      do SendMsg(?d,SetFeatureInfo!);
      /*---End of Feature Transfer --------------------------------*/

      /*---Transfer of Adjacency----------------------------------*/
      for ?d = find direct instanceof Departments;
      do SendMsg(?d,SetAdjInfo!);
      /*---End of Adjacency Transfer.-----------------------------*/

      SendMsg(AdjacencyDefinePanel,TakeOffScreen!);

  return Null;
}


/*-------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.GetInfoFromAdj!
 *
```

```
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method DeptControler.GetInfoFromAdj! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

      SendMsg(DidYouChangeDeptNo,PutOnScreenAndWait!);

  return Null;
}
```

```
#include <prk/lib.pth>

function Member_of(?Target, ?List)
{
        bound inputs;

        if ?Target == Null; return 0;
        if ?List == Null;  return 0;
        if ?List ==`() ;  return 0;
        if ?Target == ListFirst(?List); return -1;
        return Member_of(?Target, ListRest(?List));
}

function CalDis!(?D1, ?D2)
{
        bound inputs;
                ?x1 = ?D1.LastRoom.X;
                ?y1 = ?D1.LastRoom.Y;
                ?x2 = ?D2.FirstRoom.X;
                ?y2 = ?D2.FirstRoom.Y;
                ?Distance = Abs(?x1-?x2) + Abs(?y1-?y2);
        return ?Distance;
}

/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.Intialize!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.Intialize! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

/*    Print ("\nMethod:",
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");                  */
         Print (".");
         ClearValues(?self,Occupant);
         ?self.image.Background = ?self.PreoccupantColor;
         roomControler.OccupiedRooms -== ?self;
         generalControler.CurrentDept.Occupants -== ?self;
         generalControler.CurrentDept.NoOfWaiting =
generalControler.CurrentDept.NoOfWaiting + 1;
   return Null;
}


/*-------------------------------------------------------------------------
```

```
 *
 * ProTalk method  --  Rooms.IntializeAll!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.IntializeAll! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:", "\n  object ", ?self, "\n  slot   ", ?slot,
          "\n");
        for find ?room = direct instanceof Rooms;
        do
                SendMsg(?room, Initialize!);
        generalControler.TotalRoomsToAllocate = 0;
        generalControler.ResultNo = 0;
        for find ?dept = direct instanceof Departments;
        do
                {?dept.NoOfWaiting = ?dept.NoOfRooms;
                ?dept.Occupants = Null;
                ?dept.LastRoom = Null;
                ?dept.FirstRoom = Null;
                sum ?dept.NoOfRooms into ?total;
                }
        generalControler.NoOfDeptsWaiting = generalControler.NoOfDepts;
        for find ?result = direct instanceof Results;
                do DeleteObject(?result);
   return Null;
}
/*-------------------------------------------------------------------------
 *
 *  This is the main method.
 *
 */
method Rooms.Assign! ()
{
bound inputs;
?RoomsInDept = generalControler.CurrentDept.NoOfRooms;
?CurrentD = generalControler.CurrentDept;          /* Remember current Dept */
?self.Occupant = ?CurrentD;                        /* Record current occupant */
generalControler.CurrentRoom = ?self;
?CurrentD.Occupants +== ?self;
roomControler.OccupiedRooms +== ?self;
?CurrentD.NoOfWaiting = ?CurrentD.NoOfWaiting - 1;
?self.image.Background = ?CurrentD.Color;

if find count ?CurrentD.Occupants >= ?RoomsInDept;
                                        /* Done with this department */
        {
```

```
/*Feasibility Check  */
        /*   Feasibility Check by Rules.
        if SendMsg(DeptTest, checkAll!) == NO;
              then
                 {
                  SendMsg(?self,Initialize!);
                  fail;}
              else
        */
        ?judge = SendMsg(PreAssignTest, check!);
                 if ?judge == NO;
                   then {
                          SendMsg(?self,Initialize!);
                          fail;
                          }
        ?judge = SendMsg(blockTest, check!);
                 if ?judge == NO;
                   then
                          {SendMsg(?self,Initialize!);
                          fail;
                          }
                   else
        ?judge = SendMsg(accessTest, check!);
                 if ?judge == NO;
                   then {
                          SendMsg(?self, Initialize!);
                          fail;
                          }
                   else
        ?judge = SendMsg(negativeAdjacencyTest, check!);
                 if ?judge == NO;
                   then {
                          SendMsg(?self,Initialize!);
                          fail;
                          }
                   else
        ?judge = SendMsg(viewTest, check!);
                 if ?judge == NO;
                   then {
                          SendMsg(?self, Initialize!);
                          fail;
                          }
                   else
        ?judge = SendMsg(sunlightTest, check!);
                 if ?judge == NO;
                   then {
                          SendMsg(?self, Initialize!);
                          fail;
                          }
                   else

/* End of Feasibility Check */

        for find ?R = roomControler.OccupiedRooms;
```

```
        do {

                ?nextD = find ?R.Occupant.positiveAdjacencies;

                ?nextD.NoOfWaiting > 0;
                ?nextD.NoOfRooms > 0;              /* Next dept must need rooms */
                ?R.Occupant.Adjacency..?nextD >= 0;
                                /* Next dept must have positive
                                adjacency with Current Dept. */

           ?NextR = find ?R.Neighbors;    /* NextR = Find neighbor */
           ?NextR.Occupant == Null;       /* Neighbor must be available */
                if ?NextR.Preoccupant != Null;  /* PreAssignCheck */
                  then ?NextR.Preoccupant == ?nextD;

                generalControler.CurrentDept = ?nextD;
                                                /* Get ready to do next dept */
                ?CurrentD.LastRoom = ?self;
                ?nextD.FirstRoom = ?NextR;
                if generalControler.Mode != DONE;
                then SendMsg(?NextR, Assign!);    /* Go assign next room, dept */
                generalControler.CurrentDept = ?CurrentD;
                                                /* Reset on return */
          }

/*--- Evaluation of this alternative. ------------------------------------*/

        if for ?d == find direct instanceof Departments;
                always find count ?d.Occupants >= ?d.NoOfRooms;
        then  {
                ?CurrentD.LastRoom = ?self;

          generalControler.ResultNo = generalControler.ResultNo + 1;

                for ?E = find instanceof Evaluation;
                do {
                  ?E != TotalValue;
                  ?E.Active == YES;
                  SendMsg(?E, Evaluate!);
                  ?E.SumUpValue = ?E.SumUpValue + ?E.Value;
                  ?ave = ?E.SumUpValue/generalControler.ResultNo;
                  ?point = ?E.Value/?ave;
                  if ?E.Type == A;
                  then ?E.ValueForTotal = ?point;
                  else ?E.ValueForTotal = 1/?point;
                  }
                SendMsg(TotalValue,Evaluate!);

        /*--- reset the Best & Worst record ---------------------------------*/
                for ?e = find instanceof Evaluation;
                do {
                  select{
                   case: ?e.Type == A;
                         {if ?e.Value <= ?e.BestValue; ?e.BestValue = ?e.Value;
```

```
                    if ?e.Value >= ?e.WorstValue; ?e.WorstValue = ?e.Value;
                    }
               case: ?e.Type == D;
                    {if ?e.Value >= ?e.BestValue; ?e.BestValue = ?e.Value;
                    if ?e.Value <= ?e.WorstValue; ?e.WorstValue = ?e.Value;
                    }
                    }
             }
/*---End of reset the Best & Worst Record ----------------------*/

/*---Make records for best 10 ----------------------------------*/
if generalControler.ResultNo <= 10;
then {
        SendMsg(Results, MakeRecord!);
   if generalControler.ResultNo == 10;
        then{
        SendMsg(Results, FindWorst!);
           }
   }
else {
   generalControler.NewResult = 0;
   for ?e = find instanceof Evaluation;
   do{
        ?e.Active == YES;
        ?e.IHaveNew = 0;
        select{
         case: ?e.Type == A;
                {
        if ?e.Value <= ?e.WorstResult.?e;
        then{
           generalControler.NewResult =1;
           ?e.IHaveNew =1;
           ?e.Results -== ?e.WorstResult;
           }
                }          /* end of Case-1 */
         case: ?e.Type == D;
        {
        if ?e.Value >= ?e.WorstResult.?e;
  then{
           generalControler.NewResult =1;
           ?e.IHaveNew = 1;
     ?e.Results -== ?e.WorstResult;
    }
          }                  /* end of Case-2 */
           }          /* end of select */
        }               /* end of for_do loop */

  /*---Delete Result if it is not contained in any Evaluation.-----*/
  for ?r = find direct instanceof Results;
  do {
        ?judge = SendMsg(?r,DeleteResults!);
        if ?judge == OK; DeleteObject(?r);
        fail;
        }
```

```
            /*---End of Delete Result.-----------------------------------*/

            SendMsg(Results, DeleteResults!);
            if generalControler.NewResult ==1; SendMsg(Results,MakeRecord!);
            SendMsg(Results, FindWorst!);

             }                  /* end of else   */
          /*---End of Making best 10 Records ---------------------------------*/

                  }           /* end of then */
/*---End Of Evaluation. ------------------------------------------------*/



        SendMsg(?self, Initialize!);
        return Null;
        }

for find {      ?NextR = ?self.Neighbors;   /* Get all remaining neighbors */
                }
do {    ?NextR.Occupant == Null;            /* Next must be available */
        if ?NextR.Preoccupant != Null;       /* preAssign Check for Next */
          then ?NextR.Preoccupant == generalControler.CurrentDept;

        if generalControler.Mode != DONE;
        then SendMsg(?NextR, Assign!);
        }
SendMsg(?self, Initialize!);
Print("\nBacktrack", ?self);
if generalControler.Mode != DONE;
then fail;                                  /* Backtrack */
Print("\n We Should never get here.");

return Null;
}

/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  RoomControler.Start!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method RoomControler.Start! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
```

```
        find ?firstRoom = direct instanceof Rooms;
        ?firstRoom.Preoccupant == Null;
        ?firstRoom.Occupant != Null;
        ?self.StartRoom = ?firstRoom;
        generalControler.StartDept = ?firstRoom.Occupant;

  return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  General.FindAlternativeLayouts!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.FindAlternativeLayouts! ()
{
  /* Top level methods to find all layouts */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

/* Initialize before Starting Layout */

        for find ?d = direct instanceof Departments;
        do {
          ?d.NoOfWaiting = ?d.NoOfRooms;
          }
        for find ?e = instanceof Evaluation;
        do  SendMsg(?e,Initialize!);

        generalControler.ResultNo = 0;

/* End of Initialize */

        SendMsg(roomControler, Start!);

        ?self.CurrentDept = ?self.StartDept;
        ?self.StartDept.FirstRoom = roomControler.StartRoom;
        SendMsg(roomControler.StartRoom, Assign!);


  return Null;
}

/*-----------------------------------------------------------------------
```

```
 *
 * ProTalk method  --  blockTest.check!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method blockTest.check! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

         if ?self.Active == NO; return YES;

         if generalControler.CurrentDept.Type == Block;
         then
         {
         ?rooms = GetValues(generalControler.CurrentDept, Occupants);
         for ?room inlist ?rooms;
         do {
           ?neighbors = GetValues(?room, Neighbors);
           ?n = 0;
           for ?neighbor inlist ?neighbors;
           do {
                 if ?neighbor.Occupant == ?room.Occupant;
                   then ?n = ?n+1;
              }
           if ?n == 1; return NO;
           }
         return YES;
         }
         else return YES;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Test.checkAll!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Test.checkAll! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
```

```
   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        if for ?test = find instanceof ?self;
        always SendMsg(?test,checkFeasibility!) == YES;

        then return YES;
        else return NO;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Test.checkFeasibility!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Test.checkFeasibility! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        ?D = generalControler.CurrentDept;
        ?D.feasibility = Null;
        if ?self.Active == NO;   return YES;

        ?object = GetValues(?self, Depts);
      ?judge = Member_of(?D, ?object);
        if ?judge != -1;        return YES;

        if {    find [?self.Rules] ?x = ?D.feasibility;
                /* need step below to invoke rules */
                ?x == NO;}
        then return NO;
        else return YES;
}



/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.LookAround!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
```

```
 *
 */

method Rooms.LookAround! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        ?neighbors = GetValues(?self, Neighbors);
        ?empty = 0.0;
        for ?neighbor inlist ?neighbors;
        do {
          if ?neighbor.Occupant == Null;
            then ?empty = ?empty + 1.0;
            else if ?neighbor.Preoccupant != Null;
                  then ?empty = ?empty +1.0;
          }

   return ?empty;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.LookSouth!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.LookSouth! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        if ?self.South == Null;
     then return DEAD;
     else
     if ?self.South.Occupant == Null;
     then return SendMsg(?self.South, LookSouth!);
     else return ?self.South.Occupant;
}
```

```
/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.LookNorth!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.LookNorth! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

         if ?self.North == Null;
         then return DEAD;
         else
         if ?self.North.Occupant == Null;
         then return SendMsg(?self.North, LookNorth!);
         else return ?self.North.Occupant;

}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.LookEast!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.LookEast! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        if ?self.East == Null;
      then return DEAD;
      else
      if ?self.East.Occupant == Null;
      then return SendMsg(?self.East, LookEast!);
```

```
      else return ?self.East.Occupant;

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.LookWest!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.LookWest! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        if ?self.West == Null;
      then return DEAD;
      else
      if ?self.West.Occupant == Null;
      then return SendMsg(?self.West, LookWest!);
      else return ?self.West.Occupant;

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.FindSunlightRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindSunlightRoom! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
```

```
        ?sunlightRoom = 0.0;
        ?rooms = GetValues(?self, Occupants);
        for ?room inlist ?rooms;
        do {
          ?empty = SendMsg(?room, LookAround!);
          if ?empty >0;
            then ?sunlightRoom = ?sunlightRoom +1.0;
          }
        return ?sunlightRoom;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.FindEastViewRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindEastViewRoom! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        ?view = 0.0;
        ?rooms = GetValues(?self, Occupants);
        for ?room inlist ?rooms;
        do {
          ?east = SendMsg(?room, LookEast!);
          if ?east == GREEN@;
            then ?view = ?view + 1.0;
          }
        return ?view;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.FindNorthViewRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindNorthViewRoom! ()
{
```

```
    /* Methods must always have all their inputs bound: */
    bound inputs;

    Print ("\nMethod:",
           "\n  object ", ?self,
           "\n  slot   ", ?slot,
           "\n");
         ?view = 0.0;
       ?rooms = GetValues(?self, Occupants);
       for ?room inlist ?rooms;
       do {
         ?east = SendMsg(?room, LookNorth!);
         if ?east == GREEN@;
           then ?view = ?view + 1.0;
         }
       return ?view;

    return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.FindSouthViewRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindSouthViewRoom! ()
{
    /* Methods must always have all their inputs bound: */
    bound inputs;

    Print ("\nMethod:",
           "\n  object ", ?self,
           "\n  slot   ", ?slot,
           "\n");
         ?view = 0.0;
       ?rooms = GetValues(?self, Occupants);
       for ?room inlist ?rooms;
       do {
         ?east = SendMsg(?room, LookSouth!);
         if ?east == GREEN@;
           then ?view = ?view + 1.0;
         }
       return ?view;

    return Null;
}


/*-------------------------------------------------------------------------
```

```
 *
 * ProTalk method  --  Departments.FindWestViewRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindWestViewRoom! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        ?view = 0.0;
     ?rooms = GetValues(?self, Occupants);
     for ?room inlist ?rooms;
     do {
       ?east = SendMsg(?room, LookWest!);
       if ?east == GREEN@;
         then ?view = ?view + 1.0;
       }
     return ?view;

   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  PreAssignTest.check!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method PreAssignTest.check! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        if ?self.Active == NO; return YES;

        ?D = generalControler.CurrentDept;
        ?target = ?D.PreAssignRoom;
```

```
        if ?target == Null;
          then return YES;
          else {
        ?list = GetValues(?D, Occupants);
        ?judge = Member_of(?target, ?list);
        if ?judge == -1;
          then return YES;
          else return NO;
                }
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  viewTest.check!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method viewTest.check! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

        if ?self.Active != YES;  return YES;

/* View Test for the Current Department.  */

        ?D = generalControler.CurrentDept;
        if ?D.needsView == YES;
        then {
                ?judge = SendMsg(?D, FindGoodViewRoom!);
                if ?judge == 0.0;
                then return NO;
                }

/* View Test for previously allocated Departments.  */

        ?pD = find direct instanceof Departments;
          ?pD != ?D;
          ?pD.needsView == YES;
          ?pDo = GetValues(?pD, Occupants);
          ?pDo != Null;
          ?pDo != `();
          ?judge2 = SendMsg(?pD, FindGoodViewRoom!);
          {
```

```
            if ?judge2 == 0.0;
              then return NO;
              else fail;
             }

        return YES;
   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.FindGoodViewRoom!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Departments.FindGoodViewRoom! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        ?goodView = 0.0;
        ?e = SendMsg(?self, FindEastViewRoom!);
        ?n = SendMsg(?self, FindNorthViewRoom!);
        ?s = SendMsg(?self, FindSouthViewRoom!);
        ?w = SendMsg(?self, FindWestViewRoom!);
        ?goodView = ?e + ?n + ?s + ?w;
        return ?goodView;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  sunlightTest.check!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method sunlightTest.check! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
```

```
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");
          if ?self.Active != YES;  return YES;

/* Sunlight Test for the Current Department.  */
          ?D = generalControler.CurrentDept;
          if ?D.needsSunlight == YES;
          then {
               ?judge = SendMsg(?D, FindSunlightRoom!);
                   if ?judge == 0.0;  return NO;
              }

/* Sunlight Test for the previously allocated departments. */
          ?pD = find direct instanceof Departments;
          ?pD != ?D;
          ?pD.needsSunlight == YES;
          ?pDo = GetValues(?pD, Occupants);
          ?pDo != Null;
          ?pDo != `();
          ?judge2 = SendMsg(?pD, FindSunlightRoom!);
          {
          if ?judge2 == 0.0;
            then return NO;
            else fail;
           }

      return YES;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  accessTest.check!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method accessTest.check! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
            "\n  object ", ?self,
            "\n  slot   ", ?slot,
            "\n");

          if ?self.Active != YES; return YES;

          ?D = generalControler.CurrentDept;
          if ?D.needsAccess != YES; return YES;
```

```
        ?rooms = GetValues(?D, Occupants);

        for ?room inlist ?rooms;
        do{    ?neighbors = GetValues(?room, Neighbors);
               for ?neighbor inlist ?neighbors;
               do {
                  if ?neighbor.Occupant == ROAD@; return YES;
                        fail;
               }
        }

        return NO;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  negativeAdjacencyTest.check!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method negativeAdjacencyTest.check! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
        if ?self.Active == NO; return YES;

        ?d = generalControler.CurrentDept;
        ?rooms = GetValues(?d, Occupants);
        for ?room inlist ?rooms;
        do {
          ?neighbors = GetValues(?room, Neighbors);
          for ?neighbor inlist ?neighbors;
          do {
                ?nD = ?neighbor.Occupant;
                ?nD == find direct instanceof Departments;
                if ?d.Adjacency..?nD < 0;  return NO;
            }
          }
        return YES;

}

method feasibilityCheck.React!  (?button)
{
   /* Methods must always have all their inputs bound: */
```

```
    bound inputs;
    ?dialog_box = ?self.DialogBox;
    SendMsg (?dialog_box, TakeOffScreen!);

    /* If named objects are used and the dialog box WILL NEVER be
     * converted to a blueprint, GetValue(s) can be used instead of
     * GetControlValue(s).
     */


    select {
        case: ?button == "Continue";
            Print (?button, "\n");
        case: ?button == "Stop";
            Print (?button, "\n");
    }

    return Null;
}



/*-------------------------------------------------------------------------
 *
 * ProTalk method -- Results.MakeRecord!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Results.MakeRecord! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

      bound inputs;
      ?number = generalControler.ResultNo;
      ?result =
MakeInstance(ConvertToSymbol(AppendStrings("result",ConvertToString(?number))),
sky, Results);

         for ?e = find instanceof Evaluation;
         do {
           ?e.Active == YES;
           ?result.?e = ?e.Value;
           }
```

```
    for ?d = find instanceof Departments;
    do {
      ?Occupants = GetValues (?d, Occupants);
      if ?Occupants != Null;
      then for ?o inlist ?Occupants;
          do {
            ?Slot = ConvertToSymbol(?d);
            ?result.?Slot +== ?o;
            }
      }

      if ?number <= 10;
       then {
              for ?e = find instanceof Evaluation;
              do { ?e.Active == YES;
                   ?e.Results +== ?result;
              }
              }
       else {
              for ?e = find instanceof Evaluation;
              do {
                ?e.Active == YES;
                if ?e.IHaveNew == 1;
                then ?e.Results +== ?result;
                }
              }

    return Null;

}
```

```
#include <prk/lib.pth>

/* sunlightTest ---------------------------------
 *  This Dept needs empty neighbor to geet sunlight.
 *  Also this room may not block neighbor's sunlight.
 *
 *-------------------------------------------------*/
bcrule sunlightrule1 in sunlightTestRules priority 0
{if:
        ?D = generalControler.CurrentDept;
        ?R = generalControler.CurrentRoom;
        ?D.needsSunlight == YES;
        ?empty = SendMsg(?R, LookAround!);
        ?empty == 0;
 then:
        ?D.feasibility = NO;
}

bcrule sunlightrule2 in sunlightTestRules
{if:
        ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
        ?neighbor = ?R.Neighbors;
        ?nD = ?neighbor.Occupant;
        ?nD.needsSunlight == YES;
        ?nDR = SendMsg(?nD, FindSunlightRoom!);
        ?nDR == 0;
 then:
        ?D.feasibility = NO;
}

/* viewTest ---------------------------------------
 *  This Dept must have at least one room with good view.
 *  viewBlockTest tests if this room blocks neighbor's good view.
 *
 * ------------------------------------------------ */

bcrule viewrule1 in viewTestRules
{if:
        ?D = generalControler.CurrentDept;
        ?D.needsView == YES;
        ?sView = SendMsg(?D, FindSouthViewRoom!);
        ?nView = SendMsg(?D, FindNorthViewRoom!);
        ?eView = SendMsg(?D, FindEastViewRoom!);
        ?wView = SendMsg(?D, FindWestViewRoom!);
        ?sView == 0;
        ?nView == 0;
        ?eView == 0;
        ?wView == 0;
 then:
        ?D.feasibility = NO;
}

/*
```

```
bcrule viewrules1 in viewTestRules
{if:
        ?D = generalControler.CurrentDept;
        ?R = generalControler.CurrentRoom;
        ?s = SendMsg(?R, LookSouth!);
        ?n = SendMsg(?R, LookNorth!);
        ?e = SendMsg(?R, LookEast!);
        ?w = SendMsg(?R, LookWest!);
        ?D.NoOfWaiting <= 0;
        ?s != GREEN@;
        ?n != GREEN@;
        ?e != GREEN@;
        ?w != GREEN@;
 then:
        ?D.feasibility = NO;
}

bcrule viewrules2 in viewTestRules
{if:
      ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
      ?S = SendMsg(?R, LookWest!);
      ?S == GREEN@;
 then:
      ?D.feasibility = YES;
}

bcrule viewrules3 in viewTestRules
{if:
      ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
      ?S = SendMsg(?R, LookNorth!);
      ?S == GREEN@;
 then:
      ?D.feasibility = YES;
}

bcrule viewrules4 in viewTestRules
{if:
      ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
      ?S = SendMsg(?R, LookEast!);
      ?S == GREEN@;
 then:
      ?D.feasibility = YES;
}
*/

bcrule viewrules5 in viewTestRules priority 100
{if:
        ?D = generalControler.CurrentDept;
        ?D.needsView != YES;
then:
        ?D.feasibility = YES;
```

```
}


/* viewBlockTest ------------------------------------
 *  This room may not block neighbor's good view.
 *
 *-------------------------------------------------*/

bcrule viewBlockrule1 in viewBlockTestRules
{if:
        ?D = generalControler.CurrentDept;
        ?R = generalControler.CurrentRoom;
        ?nR = ?R.North;
        ?nD = ?nR.Occupant;
        ?nD != ?D;
        ?nD.needsView == YES;
        ?nDv = SendMsg(?nD, FindSouthViewRoom!);
        ?nDv == 0;
 then:
        ?D.feasibility = NO;
}

bcrule viewBlockrule2 in viewBlockTestRules
{if:
        ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
      ?sR = ?R.South;
        ?sD = ?sR.Occupant;
        ?sD != ?D;
        ?sD.needsView == YES;
        ?sDv = SendMsg(?sD, FindNorthViewRoom!);
        ?sDv == 0;
 then:
        ?D.feasibility = NO;
}

bcrule viewBlockrule3 in viewBlockTestRules
{if:
        ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
      ?eR = ?R.East;
        ?eD = ?eR.Occupant;
        ?eD != ?D;
        ?eD.needsView == YES;
        ?eDv = SendMsg(?eD, FindWestViewRoom!);
        ?eDv == 0;
 then:
        ?D.feasibility =NO;
}

bcrule viewBlockrule4 in viewBlockTestRules
{if:
        ?D = generalControler.CurrentDept;
      ?R = generalControler.CurrentRoom;
```

```
      ?wR = ?R.West;
        ?wD = ?wR.Occupant;
        ?wD != ?D;
        ?wD.needsView == YES;
        ?wDv = SendMsg(?wD, FindEastViewRoom!);
        ?wDv == 0;
 then:
        ?D.feasibility = NO;
}

/* blockTest ----------------------------------------------
 *  This Dept must be allocated as a block.
 *
 * ----------------------------------------------------------*/

/*
bcrule rule1 in blockTestRules
{if:
        generalControler.CurrentDept.NoOfRooms != 2;
        generalControler.CurrentDept.NoOfRooms != 3;
        generalControler.CurrentDept.NoOfRooms != 5;
        generalControler.CurrentDept.NoOfRooms != 7;
 then:
        ?IsItBlock = YES;
}

bcrule rule2 in blockTestRules
{if:
      ?rooms == GetValues(generalControler.CurrentDept, Occupants);
        ?room == inlist ?rooms;
        find count ?room.Neighbors <= 3;
 then:
      ?IsItPeripheral = YES;
}

bcrule rule3 in blockTestRules
{if:
        ?IsItBlock != YES;
        generalControler.CurrentDept.Type == Block;
        ?IsItPeripheral == Yes;
 then:
      generalControler.CurrentDept.feasibility = NO;

}
*/

/* accessTest ---------------------------------------
 *  This Dept needs access to exterior.
 *  So, this Dept must have at least one room whose neighbor is ROAD.
 *                              room in peripheral.
 *
 * ----------------------------------------------------*/
bcrule rule1 in accessTestRules
        /* YES if any Room in Dept has access */
```

```
{if:
        ?D = generalControler.CurrentDept;
        ?room = ?D.Occupants;
        ?N = ?room.Neighbors;
        ?N.Occupant == ROAD;
 then:
        ?D.feasibility = YES;
}

bcrule rule2 in accessTestRules
{if:
        ?D = generalControler.CurrentDept;
        ?room = ?D.Occupants;
        ?room.numberOfNeighbors <=3;
 then:
        ?D.feasibility = YES;
}

bcrule rule3 in accessTestRules
{if:
        ?D = generalControler.CurrentDept;
        ?D.needsAccess == YES;
        ?D.feasibility != YES;
 then:
        ?D.feasibility = NO;
}
```

```
/*
 *
 * ProTalk method source file
 *
 */


#include <prk/lib.pth>


/*---------------------------------------------------------------------
 *
 * ProTalk method  --  MaterialHandling.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method MaterialHandling.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

      ?Eval = 0.0;
      ?Evaluation = 0.0;
      for ?D1 = find direct instanceof Departments;
      do {
            ?D1.NoOfRooms > 0;
            find  count ?D1.Occupants > 0 ;
            sum ?Evaluation into ?Eval;
            for ?D2 = find direct instanceof Departments;
            do {
                  ?D2 != ?D1;
                  ?D2.NoOfRooms >0;
                  find count ?D2.Occupants >0;
                  ?re = ?D1.MaterialFlow..?D2;
                  ?e = CalDis!(?D1, ?D2)* ?re;
                sum ?e into ?Evaluation;
              }
        }
      ?Eval = ?Eval+?Evaluation;

        ?self.Value = ?Eval;

   return Null;
}
```

```
/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  LengthOfWalls.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method LengthOfWalls.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

         ?eval = 0.0;
         for ?r = find direct instanceof Rooms;
         do {
           ?r.Preoccupant != ROAD@;
           ?r.Preoccupant != GREEN@;
           ?r.Occupant != Null;
           ?value = SendMsg(?r,LookAround!)+ 4 - ?r.numberOfNeighbors;
           sum ?value into ?eval;
           }

         ?self.Value = ?eval;
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  LengthOfFreeRoad.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method LengthOfFreeRoad.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
```

```
        ?eval = 0.0;
        for ?r = find direct instanceof Rooms;
        do {
          if ?r.Occupant != ROAD@; then fail;
          ?n = GetValues(?r, Neighbors);
          for ?nei inlist ?n;
          do {
                if ?nei.Occupant == Null;
                then ?eval = ?eval + 1.0;
                else {
                    if ?nei.Occupant == GREEN@;
                        then ?eval = ?eval +1.0;
                        else fail;
                        }
            }
          }

        ?self.Value = ?eval;

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  NoOfGoodViewRooms.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method NoOfGoodViewRooms.Evaluate! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

     ?eval = 0.0;
     for ?d = find direct instanceof Departments;
     do {
       if ?d.needsView != YES; then fail;
       else
       ?value = SendMsg(?d,FindGoodViewRoom!);
       sum ?value into ?eval;
       }

       ?self.Value = ?eval;
```

```
   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  NoOfAccessRooms.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method NoOfAccessRooms.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

      ?eval = 0.0;
      for ?D = find direct instanceof Departments;
      do {
        if ?D.needsAccess != YES; then fail;
        else
        ?rooms = GetValues(?D, Occupants);
          for ?room inlist ?rooms;
          do{ ?neighbors = GetValues(?room, Neighbors);
              for ?neighbor inlist ?neighbors;
              do {
                  if ?neighbor.Occupant == ROAD@;
                   then ?eval = ?eval + 1.0;
                  fail;
                  }
            }
         }

        ?self.Value = ?eval;

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  NoOfSunlightRooms.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */
```

```
method NoOfSunlightRooms.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        ?eval = 0.0;
        for ?D = find direct instanceof Departments;
        do {
          if ?D.needsSunlight != YES; then fail;
          ?rooms = GetValues(?D, Occupants);
          for ?r inlist ?rooms;
          do {
                ?e = SendMsg(?r, LookAround!);
                sum ?e into ?eval;
            }
           }

        ?self.Value = ?eval;
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  NumberOfProperRooms.Evaluate!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method NumberOfProperRooms.Evaluate! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        ?eval = 0.0;

/*  Advanced Version will be as follows;
        ?weight = 0;
        for ?subEvaluation = find direct instanceof ?self;
        do {
          sum ?subEvaluation.Weight into ?weight;
```

```
            ?value = ?subEvaluation.Weight * ?subEvaluation.Value;
*** But.....*/

        for ?subEvaluation = find direct instanceof ?self;
        do {
          sum ?subEvaluation.Value into ?eval;
          }

        ?self.Value = ?eval;

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  TotalValue.Evaluate!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method TotalValue.Evaluate! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

        ?value = 0.0;

        for ?E = find instanceof Evaluation;
        do {
          ?E != ?self;
          ?E.Active == YES;
          sum ?E.ValueForTotal into ?value;
          }
        ?self.Value = ?value;

  return Null;
}



/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Results.FindWorst!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
```

```
 *
 */

method Results.FindWorst! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

         for ?e = find instanceof Evaluation;
         do {
           ?e.Active == YES;
           ?list = GetValues(?e, Results);
         if ?list == `();
         then {
            SendMsg(NoResult, PutOnScreenAndWait!);
            return Null;
            }
           select {
                 case: ?e.Type == A;
                     ?list = SortBySlot(?list, ?e, "<");
                 case: ?e.Type == D;
                      ?list = SortBySlot(?list, ?e, ">");
                   }
           ?e.WorstResult = ListFirst(?list);
           }

   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Results.DeleteResults!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Results.DeleteResults! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
```

```
/*--if the result is not contained in any Evaluation, it should be deleted.--*/

        for ?e = find instanceof Evaluation;
        do {
                ?list = GetValues(?e, Results);
                ?judge = Member_of(?self,?list);
                if ?judge == -1; return NO;
                fail;
          }
        return OK;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Results.SearchRange!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method Results.SearchRange! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Evaluation.Initialize!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method Evaluation.Initialize! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
```

```
        ?self.IHaveNew = 0;
        ?self.SumUpValue = 0;
        ClearValues(?self,Results);
        ClearValues(?self,WorstResult);
        if ?self.Type == A;
        then {
            ?self.BestValue = 99999999;
            ?self.WorstValue = -99999999;
            }
        if ?self.Type == D;
        then {
            ?self.BestValue = -99999999;
            ?self.WorstValue = 99999999;
            }
  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.CalculateValue!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method General.CalculateValue! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       /*--- Unification of values for evaluation ----------*/
       TotalValue.Weight =0;
       for ?e = find instanceof Evaluation;
       do {
              ?e.Active == NO;
              ?e.Weight = 0;
         }
       for ?e = find instanceof Evaluation;
       do{
        ?e != TotalValue;
        ?e.Active == YES;
        TotalValue.Weight = TotalValue.Weight + ?e.Weight;
        if ?e.Type == A;
        then{
              ?e.Range = ?e.WorstValue - ?e.BestValue;
           }
```

```
        if ?e.Type == D;
        then {
              ?e.Range = ?e.BestValue - ?e.WorstValue;
            }
        }
      /*--- End of Unification ------------------------------*/

      /*--- Initialize Value---------- */
      shownResult.Rank = 0;
      for ?r = find direct instanceof Results;
      do{
        ?r.FinalValue = 0;
        ClearValues(?r,Shown);
        }
      /*--- End of Initialize.---------- */

      /*---Panel On Off.-------------------------*/
      SendMsg(StartEvaluation,TakeOffScreen!);
      SendMsg(ResultValueGraph,PutOnScreen!);
      /*---End of Panel OnOff--------------------*/

      /*--- Calculate Value.---------- */
      for ?r = find direct instanceof Results;
      do SendMsg(?r, EvaluateMe!);

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Results.EvaluateMe!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method Results.EvaluateMe! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

      /*--- Transfer to standard value. ------------------------------*/
      for ?e = find instanceof Evaluation;
      do {
        ?e != TotalValue;
        ?e.Active == YES;
        if ?e.Type == A;
```

```
            then {
                  if ?e.Range != 0; then
                  ?self.Evaluation..?e = (?e.WorstValue-?self.?e)/?e.Range * 100;
                  else ?self.Evaluation..?e =50;
                  }
            if ?e.Type == D;
            then {
                  if ?e.Range != 0; then
                  ?self.Evaluation..?e = (?self.?e-?e.WorstValue)/?e.Range *100;
                  else ?self.Evaluation..?e =50;
                }
            }
          /*--- End of Value Standardization. ----------------------------*/

          /*--- Calculate Final Value considering preference factor. ------*/
          ?value = 0;
          for ?e = find direct instanceof Evaluation;
          do {
            ?e != TotalValue;
            ?e.Active == YES;
            ?v = ?self.Evaluation..?e * ?e.Weight;
            sum ?v into ?value;
            }
          ?self.FinalValue = ?value;
          /*--- End of Calculation. ---------------------------------------*/


    return Null;
}


method sky_CommandRow_186.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;
    SendMsg (?dialog_box, TakeOffScreen!);

    /*  If named objects are used and the dialog box WILL NEVER be
     *  converted to a blueprint, GetValue(s) can be used instead of
     *  GetControlValue(s).
     */


    select {
        case: ?button == "OK";
            Print (?button, "\n");
                  SendMsg (?dialog_box, TakeOffScreen!);
                  SendMsg(AdjacencyListDialogBox,PutOnScreen!);

        case: ?button == "Cancel";
            Print (?button, "\n");
                  SendMsg (?dialog_box, TakeOffScreen!);
    }
```

```
    return Null;
}
```

```
#include <prk/lib.pth>

function FindNextResult(?list)
{
        for ?r inlist ?list;
        do {
          ?r.Shown != DONE;
          return ?r;
        }
        return Null;
}

/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.FindNeighbor!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.FindNeighbor! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

       for find ?candidate = direct instanceof Rooms;
       do {
          ?candidate.X == ?self.X;
          ?dy = ?candidate.Y - ?self.Y;
          if ?dy == 1;
               then {
                        ?self.South = ?candidate;
                        ?self.Neighbors += ?candidate;
                    }
          if ?dy == -1;
                then {
                        ?self.North = ?candidate;
                        ?self.Neighbors += ?candidate;
                     }
          fail;
          }

       for find ?candidate = direct instanceof Rooms;
    do {
          ?candidate.Y == ?self.Y;
       ?dx = ?candidate.X - ?self.X;
          if ?dx == 1;
          then {
```

```
                ?self.East = ?candidate;
                ?self.Neighbors +== ?candidate;
              }
        if ?dx == -1;
            then {
                ?self.West = ?candidate;
                ?self.Neighbors +== ?candidate;
              }
        fail;
        }

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  General.Reset!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.Reset! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
 ClearValues(generalControler, StartDept);
 ClearValues(roomControler, StartRoom);

 generalControler.BestEval = 99999999;
 generalControler.WorstEval = 0;

 for find ?room = direct instanceof Rooms;
      do
           SendMsg(?room, Initialize!);
      generalControler.TotalRoomsToAllocate = 0;
      generalControler.ResultNo = 0;
      for find ?dept = direct instanceof Departments;
      do
           {?dept.NoOfWaiting = ?dept.NoOfRooms;
           ?dept.Occupants = Null;
           ?dept.LastRoom = Null;
           ?dept.FirstRoom = Null;
                ?dept.feasibility = Null;
           sum ?dept.NoOfRooms into ?total;
           }
/*     generalControler.TotalRoomsToAllocate = ?total; Probably not needed. */
```

```
      generalControler.NoOfDeptsWaiting = generalControler.NoOfDepts;
      for find ?result = direct instanceof Results;
           do DeleteObject(?result);

/*       SendMsg(InputPanel, Iconify!);  */
   return Null;
}




/*-------------------------------------------------------------------
 *
 * ProTalk method  --  General.ShowResult!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.ShowResult! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
/*---ReDraw of Layout Panel.-----------------------------------------------*/
        for ?rm = find instanceof Rooms;
        do  {
           if ?rm.Preoccupant == Null;
           SendMsg(?rm, Initialize!);
           }
/*---End Of ReDraw.--------------------------------------------------------*/

        ?list = all direct instanceof Results;
        if ?list == `();
          then {
                SendMsg(NoResult, PutOnScreenAndWait!);
                return Null;
                }
     ?newList = SortBySlot(?list, FinalValue, "<");

        ?r = FindNextResult(?newList);
        if ?r == Null;
        then {
                SendMsg(shownResultReset,PutOnScreen!);
                return Null;
                }
        else
        if shownResult.Rank == 0;
        then { shownResult.BestValue = ?r.FinalValue/TotalValue.Weight;}
```

```
        shownResult.Rank = shownResult.Rank + 1;
        for ?e = find instanceof Evaluation;
        do {
          ?e.Active == YES;
          ?e != TotalValue;
          shownResult.?e = ?r.Evaluation..?e;
          }
        shownResult.TotalValue = ?r.FinalValue/TotalValue.Weight;

        for ?d = find instanceof Departments;
        do {
          ?room = GetValues(?r, ConvertToString(?d));
          for ?rm inlist ?room;
          do {
                ?rm.Occupant = ?d;
                ?rm.image.Background = ?d.Color;
            }
          }
        ?r.Shown = DONE;

  return Null;
}

/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.Quit!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method General.Quit! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;
        for ?panel = find instanceof Panel@ActiveImageApp;
        do SendMsg(?panel,TakeOffScreen!);

        /*---Old Codes----------------------------------*/
        SendMsg(ArrangeDeptFeaturePanel, TakeOffScreen!);
        SendMsg(LayoutPanel, TakeOffScreen!);
        SendMsg(InputPanel, TakeOffScreen!);
        SendMsg(EvaluationGraph, TakeOffScreen!);
        /*---End of ole codes---------------------------*/

        C:{exit(0);}

  return Null;
}


/*-------------------------------------------------------------------------
```

```
 */

function AbsoluteStart (?App, ?Argv)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   SendMsg(SKYControler, PutOnScreen!);

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.countNeighbors!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.countNeighbors! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        ?self.numberOfNeighbors = find count ?self.Neighbors;

   return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  RoomControler.InitializeRooms!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method RoomControler.InitializeRooms! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
```

```
            "\n slot  ", ?slot,
            "\n");
/*
        SendMsg(generalControler, ResetLayout!);
        SendMsg(generalControler, ResetResult!);
        SendMsg(LayoutPanel, TakeOffScreen!);
*/

/*---ClearValue before Delete Object.----------------------------------*/
        ClearValues(generalControler,CurrentRoom);
        ClearValues(roomControler,CurrentRoom);
        ClearValues(roomControler,NextRoom);
        for ?d = find direct instanceof Departments;
        do ClearValues(?d,PreAssignRoom);
/*---End of Clear Values.-----------------------------------------------*/

/* Delete the previous objects */
      for find ?room = direct instanceof Rooms;
            do DeleteObject(?room);

/* Delete image objects */
      for find ?r_image = direct instanceof StringEditor@ActiveImagesApp;
          do { if ?r_image.Panel == LayoutPanel@sky;
                 then DeleteObject(?r_image);
                 }
         for find ?r = direct instanceof MessageButton@ActiveImagesApp;
             do { if ?r.Panel == RoomPreoccupation;
                    then DeleteObject(?r);
               }

/* Create Rooms instances */
      for ?i from 1 to roomControler.numberOfX;
      do {
            for ?j from 1 to roomControler.numberOfY;
            do {
      ?room =
MakeInstance(ConvertToSymbol(AppendStrings("Room",ConvertToString(?i),ConvertTo
String(?j))), sky, Rooms);
            ?room.X = ?i;
            ?room.Y = ?j;
                 ?room.PreoccupantColor = Black;

/* Create images of rooms in LayoutPanel & PreOccupationPanel */
      ?r_image =
MakeInstance(ConvertToSymbol(AppendStrings("room",ConvertToString(?i),ConvertToS
tring(?j))), sky, StringEditor@ActiveImagesApp);
            ?r_image.Panel = LayoutPanel@sky;
            ?r_image.MonitoredSlot = Occupant@?room@sky;
            ?r_image.Height = 900;
            ?r_image.Width = 900;
            ?r_image.Title = AppendStrings("R",ConvertToString(?i),ConvertToString(?j));
            ?r_image.PositionX = 500 + 900*(?i-1);
            ?r_image.PositionY = 500 + 900*(?j-1);
            ?room.image = ?r_image;
```

```
        ?r_image2 =
MakeInstance(ConvertToSymbol(AppendStrings("rpre",ConvertToString(?i),ConvertToStr
ing(?j))), sky, MessageButton@ActiveImagesApp);
            ?r_image2.Panel = RoomPreoccupation@sky;
            ?r_image2.MonitoredSlot = Preoccupy!@?room@sky;
            ?r_image2.Height = 900;
            ?r_image2.Width = 900;
            ?r_image2.PositionX = 500 + 900*(?i-1);
            ?r_image2.PositionY = 500 + 900*(?j-1);

              }
      }

/* Set Neighbors for each room and cout the nomber */
     for find ?room = direct instanceof Rooms;
     do  {  SendMsg(?room, FindNeighbor!);
            SendMsg(?room, countNeighbors!);
         }

/* Remove the Panel  */
        SendMsg(ArrangeRoomPanel, TakeOffScreen!);

        SendMsg(LayoutPanel, PutOnScreen!);

  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  RoomControler.ArrangeRooms!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method RoomControler.ArrangeRooms! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       SendMsg(generalControler, ResetResult!);
       SendMsg(ArrangeRoomPanel, PutOnScreen!);
  return Null;
}


/*-------------------------------------------------------------------------
```

```
 *
 * ProTalk method  --  DeptControler.ArrangeDeptNumber!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.ArrangeDeptNumber! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

/* We must delete the Result Slot before reset the number of Dept */
        for ?i from 1 to deptControler.NumberOfDept;
        do DeleteSlot(Results,
(ConvertToSymbol(AppendStrings("D",ConvertToString(?i)))));

        SendMsg(ArrangeDeptNumberPanel, PutOnScreen!);
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.InitializeDeptNumber!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.InitializeDeptNumber! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

/*--- Delete image objects ----------------------*/
        for find ?d_image = direct instanceof HorizontalSlider@ActiveImagesApp;
          do {
                if ?d_image.Panel == ArrangeDeptFeaturePanel@;
           then DeleteObject(?d_image);
            }
```

```
        for find ?d_image = direct instanceof StringEditor@ActiveImagesApp;
        do {
           if ?d_image.Panel == ArrangeDeptFeaturePanel@;
           then DeleteObject(?d_image);
          }
                /*---End of Image Delete----------------------------*/

/*--- Create Departments instances ------------*/
      ?height = 7000/deptControler.NumberOfDept;

        for ?d= find direct instanceof Departments;
      do {
       /*      ?d =
MakeInstance(ConvertToSymbol(AppendStrings("D",ConvertToString(?i))), sky,
Departments);
        -------------this is already done. --------------------------*/

                ?i=ConvertToNumber(Substring(?d, 1));

/* Create images of Departments in ArrangeDeptFeaturePanel for input */
            ?d_image =  MakeInstance(AppendStrings(ConvertToString(?d),"Rooms"), sky,
HorizontalSlider@ActiveImagesApp);
            ?d_image.MonitoredSlot = NoOfRooms@?d@sky;
            ?d_image.ValueType = PrkFixnum;
            ?d_image.RangeMaximum = 10;
            ?d_image.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_image.Height = ?height;
            ?d_image.Width = 900;
            ?d_image.Title = AppendStrings("D",ConvertToString(?i));
            ?d_image.PositionX = 500;
            ?d_image.PositionY = 2100 + ?height*(?i-1);

            ?d_color = MakeInstance(AppendStrings(ConvertToString(?d),"Color"), sky,
StringEditor@ActiveImagesApp);
            ?d_color.MonitoredSlot = Color@?d@sky;
            ?d_color.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_color.Height = ?height;
            ?d_color.Width = 900;
            ?d_color.Title = "Color";
            ?d_color.PositionX = 1500;
            ?d_color.PositionY = 2100 + ?height*(?i-1);

            ?d_type = MakeInstance(AppendStrings(ConvertToString(?d),"Type"), sky,
StringEditor@ActiveImagesApp);
            ?d_type.MonitoredSlot = Type@?d@sky;
            ?d_type.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_type.Height = ?height;
            ?d_type.Width = 900;
            ?d_type.Title = "Type";
            ?d_type.PositionX = 2500;
            ?d_type.PositionY = 2100 + ?height*(?i-1);

            ?d_negative = MakeInstance(AppendStrings(ConvertToString(?d),"Negative"),
sky, StringEditor@ActiveImagesApp);
```

```
            ?d_negative.MonitoredSlot = negativeAdjacencies@?d@sky;
            ?d_negative.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_negative.Height = ?height;
            ?d_negative.Width = 1900;
            ?d_negative.Title = "Negaitve";
            ?d_negative.PositionX = 3500;
            ?d_negative.PositionY = 2100 + ?height*(?i-1);

            ?d_access = MakeInstance(AppendStrings(ConvertToString(?d),"Access"), sky,
StringEditor@ActiveImagesApp);
            ?d_access.MonitoredSlot = needsAccess@?d@sky;
            ?d_access.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_access.Height = ?height;
            ?d_access.Width = 900;
            ?d_access.Title = "Access";
            ?d_access.PositionX = 5500;
            ?d_access.PositionY = 2100 + ?height*(?i-1);

            ?d_view = MakeInstance(AppendStrings(ConvertToString(?d),"View"), sky,
StringEditor@ActiveImagesApp);
            ?d_view.MonitoredSlot = needsView@?d@sky;
            ?d_view.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_view.Height = ?height;
            ?d_view.Width = 900;
            ?d_view.Title = "View";
            ?d_view.PositionX = 6500;
            ?d_view.PositionY = 2100 + ?height*(?i-1);

            ?d_sunlight = MakeInstance(AppendStrings(ConvertToString(?d),"Sunlight"),
sky, StringEditor@ActiveImagesApp);
            ?d_sunlight.MonitoredSlot = needsSunlight@?d@sky;
            ?d_sunlight.Panel = ArrangeDeptFeaturePanel@sky;
            ?d_sunlight.Height = ?height;
            ?d_sunlight.Width = 900;
            ?d_sunlight.Title = "Sunlight";
            ?d_sunlight.PositionX = 7500;
            ?d_sunlight.PositionY = 2100 + ?height*(?i-1);

/* Reset the Result Slot */
                MakeMultiValueSlot(Results,?d);
          }
/*
        SendMsg(ArrangeDeptNumberPanel, TakeOffScreen!);
        SendMsg(ArrangeDeptFeaturePanel, PutOnScreen!);
*/
        return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method -- DeptControler.ArrangeDeptFeature!
 *
 *      This is the default method, a simple tracer.
```

```
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.ArrangeDeptFeature! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       SendMsg(ArrangeDeptFeaturePanel, PutOnScreen!);
  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.ShowLayoutPanel!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.ShowLayoutPanel! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       SendMsg(LayoutPanel, PutOnScreen!);
  return Null;
}



/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.ResetResult!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.ResetResult! ()
{
  /* Methods must always have all their inputs bound: */
```

```
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
/*
        for ?e = find instanceof Evaluation;
        do {
          ClearValues(?e,BestResult);
          ClearValues(?e,WorstResult);
          if ?e.Type == A;
          then {
                ?e.BestValue = 99999999;
                ?e.WorstValue = 0;
                }
          else {
                ?e.BestValue = 0;
                ?e.WorstValue = 9999999;
                }
          }
*/
/*
        ?self.BestEval = 9999999;
        ?self.WorstEval = 0;
        ?self.BestResult = Null;
        ?self.ResultNo = 0;
        ?self.WorstResult = Null;
*/

/*---ClearVlaues before Delete Results instance ----------------*/
        for ?e = find instanceof Evaluation;
        do {
          ClearValues(?e,Results);
          ClearValues(?e,WorstResult);
          }
        /*--these should be done by ResetLayout.----------------
        for ?r = find direct instanceof Rooms;
        do ClearValues(?r,Occupant);                              */
/*---End of ClearValues.-----------------------------------------*/

         for find ?result = direct instanceof Results;
      do     DeleteObject(?result);

        generalControler.ResultNo = 0;

   return Null;
}


/*-------------------------------------------------------------------
 *
 * ProTalk method  --  General.ResetLayout!
 *
```

```
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.ResetLayout! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

 ClearValues(generalControler, StartDept);
 ClearValues(roomControler, StartRoom);

 for find ?room = direct instanceof Rooms;
    do{
        if ?room.Preoccupant != Null;
        then {
                ?preo = ?room.Preoccupant;
                if ?preo == find direct instanceof Preoccupants;
                then ;
                else SendMsg(?room, Initialize!);
                }
      else SendMsg(?room, Initialize!);
      }

 for find ?dept = direct instanceof Departments;
      do
           {?dept.NoOfWaiting = ?dept.NoOfRooms;
           ?dept.Occupants = Null;
           ?dept.LastRoom = Null;
           ?dept.FirstRoom = Null;
           ?dept.feasibility = Null;
           }
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.ResetDeptFeature!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.ResetDeptFeature! ()
{
   /* Methods must always have all their inputs bound: */
```

```
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

 for find ?dept = direct instanceof Departments;
      do
           {?dept.NoOfWaiting = ?dept.NoOfRooms;
           ?dept.Occupants = Null;
           ?dept.LastRoom = Null;
           ?dept.FirstRoom = Null;
           ?dept.feasibility = Null;

                ?icolor =
ConvertToSymbol(AppendStrings(ConvertToString(?dept),"Color"));
                ?icolor.Background = ?dept.Color;

                ?iType =
ConvertToSymbol(AppendStrings(ConvertToString(?dept),"Type"));
          ?iAccess =
ConvertToSymbol(AppendStrings(ConvertToString(?dept),"Access"));
          ?iView = ConvertToSymbol(AppendStrings(ConvertToString(?dept),"View"));
          ?iSunlight =
ConvertToSymbol(AppendStrings(ConvertToString(?dept),"Sunlight"));

                ?iType.Background = Black;
                if ?dept.Type == Block;
                then ?iType.Background = Red;

                ?iAccess.Background = Black;
                if ?dept.needsAccess == YES;
                then ?iAccess.Background =Red;

                ?iView.Background = Black;
                if ?dept.needsView == YES;
                then ?iView.Background = Red;

                ?iSunlight.Background = Black;
                if ?dept.needsSunlight == YES;
                then ?iSunlight.Background =Red;

/*-------- the following codes may not be necessary.  ----------------------**
                if ?dept.needsSunlight == YES;
                  then sunlightTest.Depts +== ?dept;
                if ?dept.needsAccess == YES;
                  then accessTest.Depts +== ?dept;
                if ?dept.Type == Block;
                  then blockTest.Depts +== ?dept;
                if ?dept.needsView  == YES;
                  then viewTest.Depts +== ?dept;
**-------------------------------------------------------------------------*/
          }
```

```
  return Null;
}


/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Rooms.Preoccupy!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method Rooms.Preoccupy! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");
       ?preo = roomControler.Preoccupation;
       if ?preo != Reset;
       then {
              ?self.Preoccupant = ?preo;
              ?self.PreoccupantColor = ?preo.Color;
              ?self.Occupant = ?self.Preoccupant;
              ?self.image.Background = ?self.PreoccupantColor;
              if ?preo == find direct instanceof Departments;
                then {
                       ?preo.PreAssignRoom = ?self;
                       ClearValues(?self, Occupant);
                       }
          }
       else
          {
              ?D = ?self.Preoccupant;
              ClearValues(?self, Occupant);
              ClearValues(?self, Preoccupant);
              ?self.PreoccupantColor = Black;
              ?self.image.Background = ?self.PreoccupantColor;
              if ?D == find direct instanceof Departments;
                then {
                       ClearValues(?D, PreAssignRoom);
                       }
          }

  return Null;
}
```

```
/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  RoomControler.SetPreoccupation!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method RoomControler.SetPreoccupation! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");
         SendMsg(RoomPreoccupation, PutOnScreen!);

   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk AfterChanged monitor method  --  D1ColorACMon_MonitorMethod
 *
 *       This is the default monitor method, a simple tracer.
 *       It prints the name of the monitor, the object and slot that
 *       were changed, and the new and old value arguments.
 *
 *       Note: Only the "function" syntax is allowed for ProTalk monitor methods.
 */

function D1ColorACMon_MonitorMethod (?new_value, ?old_value, ?info)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nAfterChanged method:",
          "\n  monitor   ", MonInfoMonitor  (?info),
          "\n  object    ", MonInfoObject   (?info),
          "\n  slot      ", MonInfoSlotName (?info),
          "\n  new value ", ?new_value,
          "\n  old value ", ?old_value,
          "\n");

         D1color@.Background = D1@.Color;
}


/*-------------------------------------------------------------------------
```

```
 *
 * ProTalk AfterChanged monitor method  --  DeptsColorACMon_MonitorMethod
 *
 *       This is the default monitor method, a simple tracer.
 *       It prints the name of the monitor, the object and slot that
 *       were changed, and the new and old value arguments.
 *
 *       Note: Only the "function" syntax is allowed for ProTalk monitor methods.
 */

function DeptsColorACMon_MonitorMethod (?new_value, ?old_value, ?info)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nAfterChanged method:",
          "\n  monitor   ", MonInfoMonitor  (?info),
          "\n  object    ", MonInfoObject   (?info),
          "\n  slot      ", MonInfoSlotName (?info),
          "\n  new value ", ?new_value,
          "\n  old value ", ?old_value,
          "\n");
        for ?d = find direct instanceof Departments;
        do {
        ?dColor = ConvertToSymbol(AppendStrings(ConvertToString(?d),"Color"));
        ?dColor.Background = ?d.Color;
          }
}
method OKToFirstResult.React!  (?button)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
   ?dialog_box = ?self.DialogBox;
   SendMsg (?dialog_box, TakeOffScreen!);

   /*  If named objects are used and the dialog box WILL NEVER be
    *  converted to a blueprint, GetValue(s) can be used instead of
    *  GetControlValue(s).
    */


   select {
      case: ?button == "OK";
         Print (?button, "\n");
               for ?result = find direct instanceof Results;
               do{
                 ClearValue(?result, Shown);
                 }
      case: ?button == "Cancel";
         Print (?button, "\n");
   }

   return Null;
}
```

```
/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.SetDeptInfo!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.SetDeptInfo! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        SendMsg(DeptInfo, PutOnScreen!);
   return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.SetSpaceInfo!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method General.SetSpaceInfo! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        SendMsg(RoomInfo, PutOnScreen!);
   return Null;
}

method PreAssignRadioButton.React!  (?moused_item, ?old_item)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
```

```
    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                PreAssignTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                PreAssignTest.Active = NO;
    }

    return Null;
}


method BlockTestRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                blockTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                blockTest.Active = NO;
    }

    return Null;
}


method AccessTestRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                accessTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                accessTest.Active = NO;
    }

    return Null;
}


method NegativeAdjTestRadioButton.React!  (?moused_item, ?old_item)
```

```
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                negativeAdjacencyTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                negativeAdjacencyTest.Active = NO;
    }

    return Null;
}


method ViewTestRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                viewTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                viewTest.Active = NO;
    }

    return Null;
}


method SunlightTestRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                sunlightTest.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                sunlightTest.Active = NO;
    }

    return Null;
}
```

```
method FlowEvalRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                MaterialHandling.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                MaterialHandling.Active = NO;
    }

    return Null;
}


method CostEvalRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                LengthOfWalls.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                LengthOfWalls.Active = NO;
    }

    return Null;
}


method RoomEvalRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                NumberOfProperRooms.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                NumberOfProperRooms.Active = NO;
    }
```

```
    return Null;
}


method RoadEvalRadioButton.React!  (?moused_item, ?old_item)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;


    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                LengthOfFreeRoad.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                LengthOfFreeRoad.Active = NO;
    }

    return Null;
}


method StrategyCommandRow.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;
    SendMsg (?dialog_box, TakeOffScreen!);

    /*  If named objects are used and the dialog box WILL NEVER be
     *  converted to a blueprint, GetValue(s) can be used instead of
     *  GetControlValue(s).
     */

    ?PreAssignTest_value  =
            GetControlValue (?dialog_box, `PreAssignTest);
    ?BlockTest_value  =
            GetControlValue (?dialog_box, `BlockTest);
    ?AccessTest_value  =
            GetControlValue (?dialog_box, `AccessTest);
    ?NegativeAdjacencyTest_value  =
            GetControlValue (?dialog_box, `NegativeAdjacencyTest);
    ?ViewTest_value  =
            GetControlValue (?dialog_box, `ViewTest);
    ?SunlightTest_value  =
            GetControlValue (?dialog_box, `SunlightTest);
    ?MaterialFlowEval_value  =
            GetControlValue (?dialog_box, `MaterialFlowEval);
    ?ConstructionCostEval_value  =
            GetControlValue (?dialog_box, `ConstructionCostEval);
    ?ProperRoomEval_value  =
            GetControlValue (?dialog_box, `ProperRoomEval);
```

```
    ?FreeRoadEval_value =
        GetControlValue (?dialog_box, `FreeRoadEval);

    select {
      case: ?button == "OK";
        Print (?button, "\n");
            SendMsg(LayoutStrategy, TakeOffScreen!);
      case: ?button == "Cancel";
        Print (?button, "\n");
            SendMsg(LayoutStrategy, TakeOffScreen!);
    }

    return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.SetStrategy!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method General.SetStrategy! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
         "\n  object ", ?self,
         "\n  slot   ", ?slot,
         "\n");

       SendMsg(LayoutStrategy, PutOnScreenAndWait!);
  return Null;
}


/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  General.SetPreference!
 *
 *      This is the default method, a simple tracer.
 *      It prints the name of the object and method slot.
 *
 */

method General.SetPreference! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;
```

```
    Print ("\nMethod:",
           "\n  object ", ?self,
           "\n  slot   ", ?slot,
           "\n");

         SendMsg(StartEvaluation, PutOnScreen!);
    return Null;
}

method DataImportCommandRow.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;
    SendMsg (?dialog_box, TakeOffScreen!);

    /*  If named objects are used and the dialog box WILL NEVER be
     *  converted to a blueprint, GetValue(s) can be used instead of
     *  GetControlValue(s).
     */

    ?DataImport_value  =
          GetControlValue (?dialog_box, `DataImport);

    select {
        case: ?button == "OK";
            Print (?button, "\n");
                SendMsg(generalControler, Initialize!);
                SendMsg(?self, TakeOffScreen!);
        case: ?button == "Cancel";
            Print (?button, "\n");
                SendMsg(?self, TakeOffScreen!);
    }

    return Null;
}

/*-----------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.SetFeatureInfo!
 *
 *     This is the default method, a simple tracer.
 *     It prints the name of the object and method slot.
 *
 */

method Departments.SetFeatureInfo! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
```

```
        "\n  slot   ", ?slot,
        "\n");
/*---Data Initialize.-------------------------------------------------------*/
      ?self.needsSunlight = NO;
      ?self.needsView = NO;
      ?self.needsAccess = NO;
/*---End of Initialize.-----------------------------------------------------*/

/*---Arrange Features between Adjacency and Layout.-------------------------*/
      ?list = GetValues(?self,Features);
      ?judge = Member_of(humanSpace@,?list);
      if ?judge == -1;
        then{
            ?self.needsSunlight = YES;
            ?self.needsView = YES;
            }
      ?judge = Member_of(needsAccess@,?list);
      if ?judge == -1; ?self.needsAccess = YES;

  return Null;
}

/*-------------------------------------------------------------------------
 *
 * ProTalk method  --  Departments.SetAdjInfo!
 *
 *     This is the default method, a simple tracer.
 *     It prints the name of the object and method slot.
 *
 */

method Departments.SetAdjInfo! ()
{
  /* Methods must always have all their inputs bound: */
  bound inputs;

  Print ("\nMethod:",
        "\n  object ", ?self,
        "\n  slot   ", ?slot,
        "\n");
/*---Data Initialize.-------------------------------------------------------*/
      for ?d = find direct instanceof Departments;
      do {
        ?self.Adjacency..?d = 0;
        }
/*---End of Initialize.-----------------------------------------------------*/

/*---Data Transfer for Adjacency.-------------------------------------------*/
      ?pList = GetValues(?self,positiveAdjacencies);
      ?nList = GetValues(?self,negativeAdjacencies);

      for ?d = find direct instanceof Departments;
      do {
        ?pjudge = Member_of(?d,?pList);
```

```
        if ?pjudge == -1; ?self.Adjacency..?d = 1;
        ?njudge = Member_of(?d,?nList);
        if ?njudge == -1; ?self.Adjacency..?d = -1;
        }
/*---End of Transfer.-------------------------------------------------------*/

   return Null;
}


method GoodViewRoomButton.React!  (?moused_item, ?old_item)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;


   select {
      case: ?moused_item == `YES;
         Print (?moused_item, "\n");
               NoOfGoodViewRooms.Active = YES;
      case: ?moused_item == `NO;
         Print (?moused_item, "\n");
               NoOfGoodViewRooms.Active = NO;
   }

   return Null;
}


method AccessRoomButton.React!  (?moused_item, ?old_item)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;


   select {
      case: ?moused_item == `YES;
         Print (?moused_item, "\n");
               NoOfAccessRooms.Active = YES;
      case: ?moused_item == `NO;
         Print (?moused_item, "\n");
               NoOfAccessRooms.Active = NO;
   }

   return Null;
}


method SunlightRoomRadioButton.React!  (?moused_item, ?old_item)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
```

```
    select {
        case: ?moused_item == `YES;
            Print (?moused_item, "\n");
                NoOfSunlightRooms.Active = YES;
        case: ?moused_item == `NO;
            Print (?moused_item, "\n");
                NoOfSunlightRooms.Active = NO;
    }

    return Null;
}



method DeptNoChCommandRow.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;
   /*  SendMsg (?dialog_box, TakeOffScreen!); */

    /*  If named objects are used and the dialog box WILL NEVER be
     *  converted to a blueprint, GetValue(s) can be used instead of
     *  GetControlValue(s).
     */

    ?DeptNoChangeRadioButton_value  =
            GetControlValue (?dialog_box, `DeptNoChangeRadioButton);

    select {
        case: ?button == "OK";
            Print (?button, "\n");
                if ?DeptNoChangeRadioButton_value == YES;
                then{
                ?n = 0;
                for ?d= find direct instanceof Departments;
                do {?n = ?n + 1;}
                deptControler.NumberOfDept = ?n;
                generalControler.NoOfDepts = ?n;
                SendMsg(DeptControler,AdjToLayout!);
                SendMsg(DeptControler,InitializeDeptNumber!);
                    }
                else SendMsg(DeptControler,AdjToLayout!);

                SendMsg (?dialog_box, TakeOffScreen!);

        case: ?button == "Cancel";
            Print (?button, "\n");
                SendMsg (?dialog_box, TakeOffScreen!);
    }

    return Null;
}
```

```
/*-------------------------------------------------------------------
 *
 * ProTalk method  --  DeptControler.CallAdjApp!
 *
 *       This is the default method, a simple tracer.
 *       It prints the name of the object and method slot.
 *
 */

method DeptControler.CallAdjApp! ()
{
   /* Methods must always have all their inputs bound: */
   bound inputs;

   Print ("\nMethod:",
          "\n  object ", ?self,
          "\n  slot   ", ?slot,
          "\n");

        SendMsg(AdjacencyDefinePanel, PutOnScreen!);

   return Null;
}

method FromScratchCommandRow.React!  (?button)
{
    /* Methods must always have all their inputs bound: */
    bound inputs;
    ?dialog_box = ?self.DialogBox;
    /* SendMsg (?dialog_box, TakeOffScreen!); */

    /*  If named objects are used and the dialog box WILL NEVER be
     *  converted to a blueprint, GetValue(s) can be used instead of
     *  GetControlValue(s).
     */


    select {
       case: ?button == "OK";
          Print (?button, "\n");
               /*---initialize every information about Departments.---------*/
/*---ClearValues before Delete Objects.-------------------------------------*/
         ClearValues(DeptsInAdjDialogBox,SelectionItems);
         ClearValues(DeptsInDialogBox,SelectionItems);
         ClearValues(TargetDepts,SelectionItems);
         ClearValues(DnsStrangerList,SelectionItems);
         ClearValues(PositiveList,SelectionItems);

         ClearValues(deptControler,AllDepts);
         ClearValues(deptControler,DominantDept);
         ClearValues(deptControler,PreAssignedDepts);
```

```
        ClearValues(deptControler,CurrentDept);
        ClearValues(generalControler,StartDept);
        ClearValues(generalControler,CurrentDept);
        ClearValues(deptControler,OrphanDepts);

        for ?room = find direct instanceof Rooms;
        do ClearValues(?room,Occupant);
        for ?evaluation = find instanceof Evaluation;
        do ClearValues(?evaluation,Results);
/*---End of ClearValues-----------------------------------------------------*/

        SendMsg (?dialog_box, TakeOffScreen!);

/* Delete the previous objects-----------------------------------------------*/
      for find ?dept = direct instanceof Departments;
           do {       DeleteObject(?dept);
                      DeleteSlot(Results,?dept);
                 }
        for find ?result = direct instanceof Results;
              do    DeleteObject(?result);

/* Delete image objects */
      for find ?d_image = direct instanceof HorizontalSlider@ActiveImagesApp;
        do {
           if ?d_image.Panel == ArrangeDeptFeaturePanel@;
           then DeleteObject(?d_image);
          }

      for find ?d_image = direct instanceof StringEditor@ActiveImagesApp;
        do {
           if ?d_image.Panel == ArrangeDeptFeaturePanel@;
           then DeleteObject(?d_image);
          }

              /*---End of initialize.-----------------------------------*/

              SendMsg(FileNameEntry@GetData,PutOnScreenAndWait!);

       case: ?button == "Cancel";
          Print (?button, "\n");
             SendMsg (?dialog_box, TakeOffScreen!);
   }

   return Null;
}


method sky_CommandRow_134.React!  (?button)
{
   /* Methods must always have all their inputs bound: */
   bound inputs;
   ?dialog_box = ?self.DialogBox;

   /*  If named objects are used and the dialog box WILL NEVER be
```

```
    * converted to a blueprint, GetValue(s) can be used instead of
    * GetControlValue(s).
    */


   select {
      case: ?button == "OK";
         Print (?button, "\n");
               for ?r = find direct instanceof Results;
               do {
                 ClearValues(?r,Shown);
                 }
               shownResult.Rank = 0;
               SendMsg (?dialog_box, TakeOffScreen!);

      case: ?button == "Cancel";
         Print (?button, "\n");
               SendMsg (?dialog_box, TakeOffScreen!);
   }

   return Null;
}
```