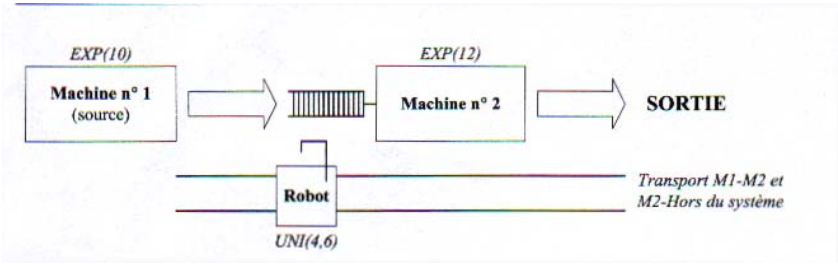
DESP-C++ "Easy Handler" Un outil Java pour modéliser une simulation

I) Introduction

Le but de notre projet est simple : pendant ses années de thèse, notre encadrant Jérôme Darmont a développé un logiciel de simulation répondant à ses propres critères de fiabilité, de facilité et surtout de rapidité. Cet outils a été baptisé DESP-C++ (Descret Evenement Paskage for C++). Le fonctionnement exact de cet outil est décrit dans le rapport de J. Darmont dont quelques pages sont en fin de rapport

Concrètement ce logiciel simule une file d'attente mettant en scène différentes sortes d'objets (ressources dites actives et ressources dites passives) qui sont liés les uns aux autres d'une manière bien définie comme l'aura souhaité l'utilisateur.

Illustrons ces propos par un exemple tiré du rapport de thèse de J. Darmont :



Ici, les Machines 1 et 2 modélisent des ressources actives (qui traite des produits) et le robot représente quant à lui une ressources dites passive (qui n'agit pas directement su les " clients " qui entrent et sont traités par le système.

Une fois que la modélisation a été décidée sur le papier, l'utilisateur doit aller directement modifier deux fichier .h du code source du programme pour voir fonctionner son modèle de simulation. Ce qui peut devenir rapidement fastidieux et inconfortable sur pour un très grand nombre de machines, voire même de ressources.

C'est donc ici que nous allons intervenir : le but de notre projet est de construite une interface très simple, qui va nous permettre de "dessiner" le problème de modélisation directement sur l'écran. Une fois un modèle validé, le code des deux fichiers .h se génèrera automatiquement. Il en résultera un bien plus grand confort d'utilisation et voire même pourquoi pas quelques multiples fantaisies dans la modélisation de système qui n'aurait pas été évident a réécrire.

II) Caractéristiques et fonctionnalité du Easy Handler

1- caractéristiques

Comme ce qui a été dit précédemment, l'intérêt de construire une telle application est le confort : confort d'utilisation, de reprogrammation etc. En automatisant les tâches on rend les actions de réédition beaucoup plus simple et rapide pour l'utilisateur. Il est facile d'imaginer l'ennui que risques de procurer la modélisation d'un système avec une centaine de ressources actives, une cinquantaine de ressources passives avec plusieurs centaines de lien entre toutes.

Le programme que nous avons créé est très simple d'utilisation
puisqu'il ne nécessite aucune connaissance du langage Java ou C++ pour être utilisé. Tout du moins, dans les représentations les moins compliquées (ce point là sera explicité par la suite). En effet l'utilisateur lance une interface qui lui permet de dessiner à l'écran sa modélisation : pas besoin de reprogrammer quoi que se soit.

Notre application a été entièrement programmée en Java. Il est donc tout d'abord multi-plateforme (i.e.) portable, ce qui respecte un des souhaits de J. Darmont en voulant, en programmant en C++ pouvoir facilement exporter ses sources vers différents systèmes.

De plus, de plus en plus de programmeurs connaissent le Java. Si l'un d'entre nous souhaite remodifier les sources pour l'éditer à sa manière, cela ne sera pas très difficile avec un minimum de connaissances : en effet, le code est simple, et bien construit (intérêt de la modélisation objet du Java). Il est facile de s'y retrouver.

De plus, il n'est pas très long. Il comporte environ 2000 lignes, mais un quart doit être réservé à l'allocation de chaînes de caractère pour pouvoir reconstituer le code des deux fichiers .h lors de la validation du modèle de l'utilisateur.

Enfin le langage Java est peut être un langage dit "lent" mais cela ne nous aura pas du tout empêché de modéliser des systèmes légèrement complexe et un peu plus "imposants" en mémoire.

2- Fonctionnalités

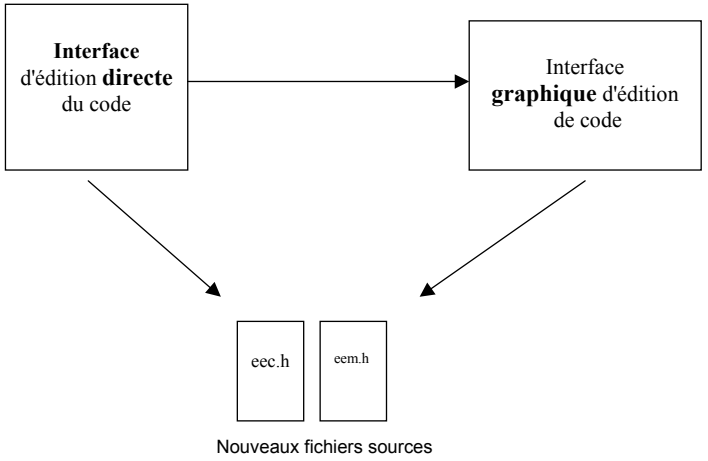
Un ensemble de classe a été programmé pour nous permettre de décrire ce qui a été mentionné précédemment. Plus précisément, deux modes d'édition nous sont proposés :

Une édition rapide qui à travers une interface nous permet de visualiser ce qui existe déjà au sein des fichiers à modifier pour que la simulation se déroule bien.

Une édition graphique qui permet à un utilisateur étranger à la programmation de pouvoir tout de même étudier une simulation dont il sera le créateur virtuel (bien entendu).

On notera qu'il sera tout à fait possible de remodifier avec la première interface (édition directe) ce qui a été fait grâce à la seconde (édition graphique). Par contre l'inverse sera bien entendu impossible.

On pourra schématiser les moyens d'édition des fichiers sources de la sorte :



III) Architecture du Easy Handler

Le programme que nous avons obtenu est un ensemble de classe réduit mais complet.

Une première classe Fenetre() est lancée au début du programme. C'est cette première fenêtre qui nous permet de réaliser une correction rapide des fichiers qui sont à compléter. A partir de là, on peut soit valider et terminer notre simulation pour la faire tourner, soit créer une instance d'une autre classe : la classe CadreDessin().
Celle-ci est bien plus complète que la précédente : En fait les fonctionnalités sont les mêmes voire même un peu plus réduites : Elle possède pourtant un avantage indéniable : l'édition graphique du problème.

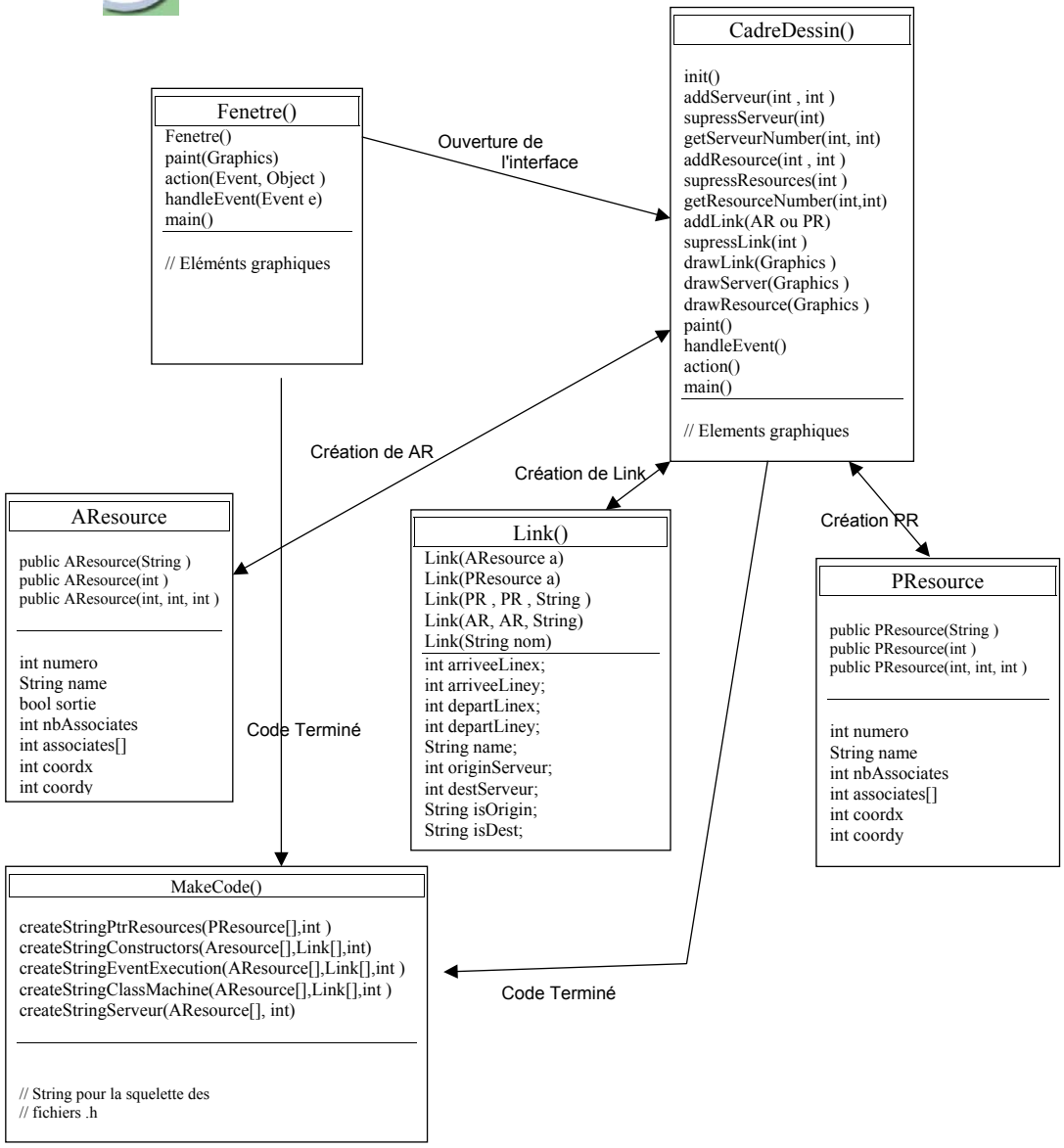
Chaque outil (ressources ou lien) qui pourra être représenté dans cet éditeur sera une instance d'une des trois classes suivantes : *Aresource()*, *Presource(),* ou Link(). Les deux premières nous permette d'implanter en mémoire l'existence de ressources passives et actives (comme ce qui a été vu en introduction). Les instances de classe Link() établissent des relations entre les différentes ressources.

Chaque ressource son nom (*name*), son numéro (*numero*), et ses coordonnées (*coordx* et *coordy*) de représentation sur la page d'édition. Les deux catégories de ressources ont aussi la possibilité de connaître les arcs qui leur sont associées grâce à la variable *associates* qui les contient tous.

De même chacun des *Link* possède dans sa structure, en plus de nom (*name*), son numéro (*numero*), la possibilité de connaître les ressources qui sont les points de départ de ces liens (*originServeur*), les destinations (*destServeur*) , de connaître leur numéro surtout de pouvoir les identifier grâce aux variable "*isOrigin*" et "*isDest"* qui précisent si les ressources qui sont liées sont des PR ou des AR (par exemple un *Link* n'existera jamais entre deux ressources passives).

Enfin, il existe une dernière classe *MakeCode()* activable par les instances des classes *CadreDessin* et *Fenetre*. une instance de la classe *MakeCode()* construit l'ensemble des chaînes de caractère qui vont être nécessaires pour constituer des nouveaux fichiers .h. en collectant les données qui ont été rentrées par l'utilisateur dans l'une des deux interfaces. Cette classe est l'une des plus importante (au niveau de la taille). Nous avons décider qu'il fallait mieux une réactualisation totale des fichiers plutôt que partielle, ce qui simplifie énormément la tâche. Cette classe possède donc un grand nombre de String qui forment ainsi le "squelette" du fichier source .h. Ce que nous avons programmé se charge de placer correctement ces Strings dans les fichiers C++ pour les rendre cohérents, et ainsi faire fonctionner le programme en C++.

L'architecture est schématisée en diagramme type UML ci-après



Schématisation du fonctionnement par un diagramme de Classe

IV) Mise en route

Lancement du programme :

On doit donc lancer le programme en exécutant la fenêtre "Fenetre.class". Une première interface simple est alors lancée. Elle se compose essentiellement de deux TextArea() qui permettent de place d'un part le texte qui a été ou qui va être modifié (donc les .h) et de l'autre coté un ensemble de sources "types" qui pourront nous servir.

Les fichiers qui ont peut-être été déjà créé après d'anciennes manipulations sont directement lues au démarrage du programme et réactualisable grâce a un FileReader (si le fichier n'existe pas, une exception est renvoyée)

```
try {
            look = new FileReader ("e:\\vince\\ter\\eem.h");
            BufferedReader buff = new BufferedReader ( look
) ;
            boolean eof = false;
            while (!eof){
                 String line = buff.readLine();
                 if (line==null)
                 eof = true;

                 else lineReturn += line+ret ;
            }buff.close();
       }catch (IOException e){}
        catch (SecurityException se){}

        buddy2.append(lineReturn);
```

La méthode action(Event e, Object arg) va nous permettre soit de lancer la modification immédiate du fichiers en train de se faire éditer, ou bien de lancer l'interface d'édition graphique du problème.

```
public boolean action(Event e, Object arg) {


       if (e.target instanceof Button) {
       //   Edition du fichier grâce à un FileWriter
       }

       if (Code.equals("Modeliser")) {
                 first = new CadreDessin();
```



```
                 first.show();
            }
       }
       return true;
  }
```

Création d'une nouvelle instance de CadreDessin :

a) Les outils nécessaires

On a trois catégories d'éléments qui peuvent être crées : Les ressources Actives (AResource()), les ressources Passives (PResource()) et les liens (Link()) qui peuvent les faire interagir entre elles. Voici les spécificités de chacune de ces classes

- Pour les AR et les PR : chacune d'entre elles est munie donc d'un numéro (numero) d'un nom (name) et de leur coordonnées graphiques (coordx et coordy). Ils sont munis tous les deux d'un tableau d'entier (associate[]). Ce tableau est initialisés avec des valeurs négatives, comme par exemple avec le constructeur d'AR suivant :

```
    public AResource(String nom) {
         name = nom;
         for (int i = 0; i < 100; i++)
              associate[i] = -1;
    }
```

associate[i] d'une AR ou PR vaut 1 si il existe un lien dont l'une des extrémités soit cette AR ou PR. Sinon elle vaut –1. Cette représentation est intéressante puisqu'elle évite les itérations : en effet, s'il nous fait savoir si deux machines sont reliées entre elles par le lien 11 par exemple on va juste aller voir dans leur associate[] respectifs. On n'aura pas à parcourir une liste d'arcs dont ils sont une extrémité pour les mettre en commun. C'est donc un moyen rapide de représentation.

De plus, les AR on un élément en plus :

```
    boolean Sortie = false;
```

Ceci nous permet de savoir quel AR (et non pas PR) est la dernière machine du circuit. La machine d'entré est par convention la machine n°1.

Les AR sont représentés graphiquement par un rectangle plein, et les PR par un ovale plein.

- Pour les Links : on retrouve en partie les outils qui ont servi à construire les AR et les PR : numéro (numero) leur coordonnées graphiques (arriveeLinex, arriveeLiney, int departLinex, int departLiney).

On obtient aussi les renseignements nécessaires sur les ressources avec qui elles sont en contact : machine à l'origine du tracé (originServeur), la machine de destination (destServeur). On nous donne en plus la nature de ces deux éléments avec isOrigin et isDest.

```
class Link {

     int arriveeLinex;
     int arriveeLiney;
     int departLinex;
     int departLiney;

     String name;
     String occ = "occupe";

     int originServeur;
     int destServeur;

     String isOrigin;
     String isDest;
}
```

Un lien est représenté par un ligne entre les deux éléments qu'il relie.

b) Comment çà marche ?

Créer une telle instance va nous permettre de modéliser notre simulation de manière graphique (on ne détaillera pas les techniques de mise en page de cette interface).

L'initialisation de cette procédure se fait comme suit

```
public void init() {
          for (int i = 0; i < 100; i++) {
               tabserveur[i] = new AResource("rien");
               tablink[i] = new Link("rien");
               tabresource[i] = new PResource("rien");
               repaint();

          }
     }
```


On crée des tableaux qui vont nous permettre d'utiliser facilement les éléments dont on aura besoin : il y a un tableau de ressources actives, un autre de ressources passives et un dernier qui s'occupe des liens. Tous ces éléments sont initialisés avec le nom "rien". On verra par la suite que leurs appellations sont les suivantes : "rien" est un élément qui n'a jamais été touché. Dans un tableau, le premier élément qui s'appelle "rien" est le successeur du dernier élément qui a été utilisé ou est encore utilisé. Les éléments qui ont été utilisés au moins une fois mais qui ne le sont pas actuellement portent le nom de "libre" et enfin sont qui sont occupées portent le nom "occupe".

NOTA : On voit qu'on utilise les tableaux et pas des Vectors. Les programmeurs n'ont appris l'existence des Vectors que trop tard pour pouvoir tout changer. En fait, ils savent que partout ou on a des tableaux, on peut remplacer par des Vectors. L'intérêt principal étant le caractère dynamique de ces structures et pas statiques. Mais la logique du programme n'en aurait absolument pas été modifiée.

Le reste des activités repose principalement sur la gestion de la méthode handleEvent particulièrement chargée.

En effet notre interface est munie d'un ensemble de bouton radios qui nous permettent de modifier (soit en ajoutant soi en supprimant un élément du dessin) le dessin à notre guise. On bénéficie des options "*ajout de AR*", "*ajout de PR*", "*ajout de Link*", ainsi que toutes les suppressions qui correspondent à ces mêmes éléments.

Lorsqu'un bouton est coché, la méthode *handleEvent* repère quel bouchon est actif et réagit en fonction de la demande de l'utilisateur. Prenons un exemple simple : la création d'un ressource active. Nous appellerons le bouton radio qui active la création de ressources actives p1. On obtient :

```
public boolean handleEvent(Event e) {
               if (e.id == Event.MOUSE DOWN) {

          if ((p1.getState() == true) && (nbServeurs < 5)) {
                    addServeur(e.x, e.y);
                    repaint();
          }
```

Les méthodes *addServeur()* et *repaint()* – ou *paint()*, c'est la même chose - sont activées. On ajoute une Aresource au dessin. En mémoire l'ajout d'un AR est faite comme telle :

```
public int addServeur(int x, int y) {
          int i = 0;
while ((tabserveur[i].name != "rien") && (tabserveur[i].name != "libre")) {
```

```
            i++;
        }
        nbServeurs++;
        tabserveur[i] = new AResource(i, x, y);
        return i;
    }
```

Dans le tableau tabserveur[], à la première case non " occupée " on range ce nouveau serveur. La fonction retourne le rang où a été rangé ce nouveau serveur.

Expliquons un cas un peu plus compliqué : l'ajout de liens entre deux ressources actives :

```
public boolean handleEvent(Event e) {

        if (e.id == Event.MOUSE_DOWN) {

        int numtemp = getServeurNumber(e.x, e.y);

    if ((p2.getState() == true) && (nbServeurs >= 2)) {
        if (numtemp > -1) {
    lastTouched = addLink(tabserveur[numtemp]);
    // renvoie le rang du premier Link libre
    tablink[lastTouched].isOrigin = "serveur";
                        check1 = 1;
                }
            }
        }
```

Il s'agit de la première étape de création du lien : il réagit lorsque le bouton radio p2 est actif, et que la souris, bouton enfoncé est sur une ressource active (getServeurNumber() va nous permettre de définir si oui ou non, la souris est placée au-dessus d'une machine ou pas).

Si en effet on clique sur une ressource active, on recherche une case de libre dans le tableaux "Links" pour y placer l'origine du futur lien (on le verra effacer par la suite si le lien ne se prolonge pas correctement (i.e.) la destination n'est pas une ressource ou si elle est dans le vide etc.)

On place aussi ce "début" de lien dans le tableau de lien *associate[]* de la ressource concernée. Une variable check1 est mise à 1 pour bien montrer que la ressource a été touchée par la souris. La deuxième étape est comme telle :

```
    if (e.id == Event.MOUSE_UP) {

        int numtemp = getServeurNumber(e.x, e.y);

    if ((p2.getState() == true) && (nbServeurs >= 2)) {
```



```
                if (numtemp > -1) {
    check2 = 1;
tablink[lastTouched].arriveeLinex = tabserveur[numtemp].coordx
+ 10;
tablink[lastTouched].arriveeLiney = tabserveur[numtemp].coordy
+ 10;
tablink[lastTouched].destServeur = tabserveur[numtemp].numero;
    tablink[lastTouched].isDest = "serveur";

            if ((check1 == check2) && (check2 == 1)) {
                            repaint();
                            nbLinks++;
    addAssociate(
    tabserveur[tablink[lastTouched].originServeur],
    tabserveur[tablink[lastTouched].destServeur],
    lastTouched);
                            }

                        }
                    else {
                     tablink[lastTouched] = new Link("libre");
                            }
                        }
                        check1 = 0;
                        check2 = 0;
                        lastTouched = -1;
                    }
                }
```

On fait glisser la souris jusqu'à la prochaine machine qui nous intéresse. Si on tombe sur une AR on complète le lien qui a été entamé à l'action précédente. Puis les fonction `addAssociate(tabserveur,tabserveur,int)` active pour chacun des deux sommets la case correspondante de leur tableau *associate[]*

Par exemple on vient de créer le 14 Link au total, toutes machines confondues, et ce Link relie 1 et 4. alors la 14 case du tableau associate[] de ces deux sommets passent à 1. On sait qui est le sommet d'origine et qui est le sommet source grâce au tableau tablink qui contient ces informations.

On voit que l'existence des liens dépendent de celles des AR ou PR. Il est évident que la supression d'une de ces ressources entraînent immédiatement la suppression de ses Link. Voyons çà par exemple pour les AR :

```
    public void supressServeur(int i) {

            int j = 0;
            while (j < 100) {
                if (tabserveur[i].associate[j] != -1)
                    supressLink(j);
```

```
                    j++;
                }
                tabserveur[i] = new AResource("libre");
                nbServeurs--;
                repaint();
            }
```

La ressource qui est alors supprimé est remplacée par une autre nommée "Libre" , en attendant la fin de l'exécution du programme.

Une fois une modélisation correcte obtenue, on définie quel est le serveur de sortie en cochant le bouton radio approprié (ici p5) et en choisissant une AR.

```
if ((p5.getState() == true) && (nbServeurs >= 1)) {
                    if (numtemp > -1) {
                        tabserveur[numtemp].Sortie = true;
                        repaint();
                    }
                }
```

Voilà, la modélisation est terminée, il ne nous manque plus qu'a compléter le code des fichiers .h. Pour cela, on active la classe MakeCode() par les lignes suivantes

```
public boolean action(Event e, Object arg) {

                if (Code.equals("go")) {
new MakeCode(tabserveur, tabresource, tablink, nbServeurs, nbResources,
nbLinks);
                }
            }
            return true;
        }
```

Créer les fichiers grâce à MakeCode() :

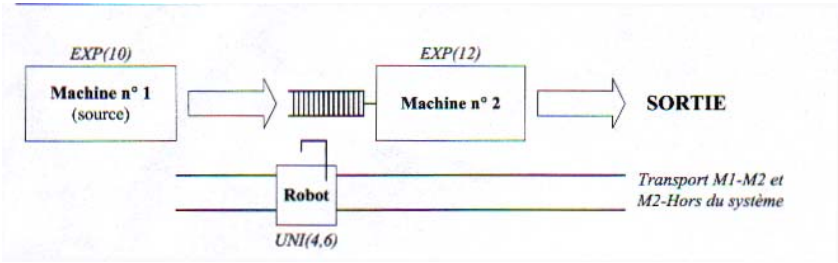
Cette classe est surtout un squelette des Strings qui nous permette de créer deux nouveau fichiers grâce à deux FileWriter (un pour chaque fichier !). Les informations qui ont été dessinées sont prises en compte (nombre de serveurs, les liens qui existent, vers quelle machine etc .). Et le squelette est ainsi complété.

V) Expérience de validation

On a décidé pour pouvoir appuyer nos dires d'essayer de faire quelques modélisations afin de bien vérifier si les fichiers qui sont crées sont longs a créés, s'ils sont bien sur, correctement créés (compilation) etc.

La seule condition du moment est de placer obligatoirement au moins une ressource passive.

Reprenons tout simplement l'exemple du début

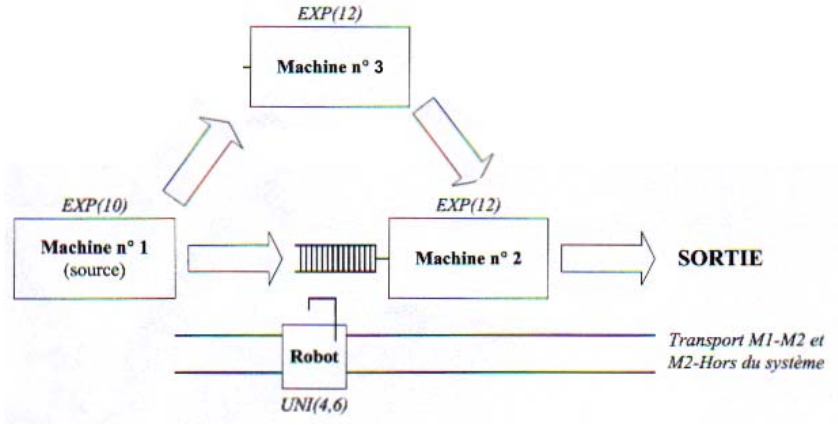


L'exemple n'est pas très difficile à traiter, les fichiers sont crées instantanément et compilent parfaitement bien. Donnons l'exemple de déclaration de la machine1

```
     class Machine1: public Resource {
       public:
         // Constructor
         Machine1(char n[STRS], int cap, Simulation *sim, Resource *rob
,Machine2 * suiv2);
     // Events
       void Event10(Client *client);       // Arrival on M1
         void Event11(Client *client);       // Begin product generation
         void Event12(Client *client);       // Product out
         void Event13(Client *client);       // Transfert toward M2

     private:                                // Pointers
     Machine2 * m2;
```

Prenons un exemple légèrement plus compliqué :



La difficulté de ce process est que la machine 1 a deux successeurs. On est donc obligé de tenir parfaitement compte de çà.

```
class Machine1: public Resource {
  public:
    // Constructor
    Machine1(char  n[STRS],  int  cap,  Simulation  *sim,  Resource  *rob
,Machine2 * suiv2,Machine3 * suiv3);
// Events
  void Event10(Client *client);          // Arrival on M1
    void Event11(Client *client);        // Begin product generation
    void Event12(Client *client);        // Product out
    void Event13(Client *client);        // Transfert toward M2
void Event14(Client *client);            // towards M3
private:                                 // Pointers
Machine2 * m2;
Machine3 * m3;
};
```

On notera bien évidemment les différences avec la première déclaration : on voit que les différents successeurs sont bien présents autant dans l'entête de déclaration de la fonction que dans son corps (les pointeurs vers m2 et m3).

Enfin pour terminer cette page d'essai on a pris un maximum de AR et de PR pour voir en combien de temps les fichiers étaient créés. Ils le sont instantanément. La mémoire est dons correctement gérée.

VI) Apports personnels et avis

Hugues : La difficulté dans la programmation d'une telle interface est d'abord "logique". Il a fallu prévoir un bon système pour mémoriser correctement les liens entre les machines et ces machines elles-mêmes. Notre souci a été un souci d'optimisation dans ces moments là. On voit par exemple que lorsque qu'une AR est supprimée le tableau n'est pas "reclassé" et réorganisé sans cette AR : les autres ne bougent pas de leur place dans le tableau. Celle qui a été supprimée est remplacée par un élément dit " vide " , c'est à dire qui n'a aucun effet sur ce la modélisation. On aurait fait de même en employant des Vectors. Notre but était donc d'éviter le maximum d'itération possible quitte à en faire une ou deux grosses sur la fin. Dans la même direction, nous avons essayé de simplifier au maximum l'implémentation de la classe MakeCode(). C'est l'ensemble de tous les Strings utilisés qui la rendent obscure de prime abord. Elle est en fait très compréhensible une fois que l'on sait ce que l'on veut.

Vincent : Moi, je vois surtout deux points particuliers qui ont du être traités avec un peu plus de patience que tous les autres : d'abord, ce n'est pas une Applet qui a été créée mais une application : certains éléments graphiques (comme l'insertion d'image) ne se traitent pas vraiment de la même façon dans une appliquette et dans une application. Il a fallu chercher avec un peu plus de patience comment programmer certaines évènements graphiques. L'autre point qui m'a paru important était (une fois que l'on a trouvé la meilleure façon de stocker les éléments en mémoire) de bien faire attention à la constitution des fichiers C++ grâce à la classe MakeCode(). Bien sur comme le dit Hugues, on a essayé de faire un code complet mais simple à comprendre. mais ce qui a été un peu plus long à penser et à programmer était la partie qui s'occupait des déclarations des éléments avec leur(s) successeur(s). c'est dans ces lignes où il paraît y avoir le plus de "bidouille" (une virgule à rajouter par-ci, un point-virgule par-là). Mais quand on y regarde de plus près, le code est satisfaisant et permet de créer tout ce qu'on veut. En y ayant mis quelques efforts, nous sommes donc parvenus à une implémentation très satisfaisante.

Ensemble : Ce projet était quelque chose de très intéressant. Nous sommes toutefois convaincus que nous aurions pu faire légèrement mieux sur certains points (comme par exemple remplacer les tableaux par des Vectors), mais faute de temps (la fin de l'année a été très difficile) nous n'avons pas pu finaliser notre projet comme nous le voulions. 50 heures de travail étaient normalement consacrées au projet TER dans la filière. Nous y avons accordé beaucoup plus de temps (Nous avons eu des cours de Java au 1er semestre, mais pas assez pour vraiment commencer à maîtriser).Mais il paraît quand même bien adapté à la demande de notre encadrant, qui nous l'espérons sera lui aussi satisfait.