Programmation sous GNUstep (1)

Nicolas Roard et Fabien Vallon

13 mars 2003

Suite à la présentation du projet GNUstep parue dans le numéro 47, nous allons commencer une petite application que nous allons faire évoluer et étendre au cours de l'année.

News

- Le nouveau «text-system» (voir l'interview parue dans le LMF numéro 47) a fusionné avec la branche principale début février.
- Optimisations pour gnustep-gui (AppKit), notamment les méthodes de dessin ainsi que diverses corrections liées au focus.
- Maintenance/Corrections mineures (du déjà stable) -base et -make.

Les commandes clavier sous GNUstep s'utilisent via la touche <commande> (pomme pour Apple), en général on affecte cette touche à <ALT> sur un clavier de type PC. Les principales actions associées à la touche commande sont :
- q pour Quitter
- w pour Fermer la fenêtre qui a le focus
- h pour Cacher
- o pour Ouvrir un document
- n pour Ouvrir un nouveau document
- s pour Sauvegarder
- t pour Afficher le Panneau de Fontes (si il y a)
- c pour Afficher le Panneau de Couleurs (si il y a)
- w pour Fermer la fenêtre qui a le focus
- c pour Copier
- v pour Coller
- x pour Supprimer

Par défaut GNUstep n'utilise qu'un seul bouton de la souris. Par exemple, en utilisant Gorm, on selectionne un objet par un seul clic. En double-cliquant sur l'objet on «l'ouvre» : si l'objet contient d'autres objets (cas d'une boîte) ou si l'objet est un conteneur, on accède aux objets à l'intérieur de celui-ci. Sinon on peut éditer «sur place» cet objet.

Description de l'application

Ce mois-ci nous allons donc commencer par quelque chose de simple. Le programme que nous allons réaliser a pour but de noter et gérer des tâches à faire ; le nom de l'application sera «Todo.app».
L'interface sera pour le moment très simple, affichant la liste des tâches dans sa partie supérieure et le contenu d'une tâche (description, date, etc.) dans sa partie inférieure. Il sera bien sûr possible de rajouter ou supprimer une tâche, et même de sauver le tout dans un fichier.
Cela nous permettra d'aborder l'outil de RAD[1] fourni avec GNUstep, Gorm, et d'introduire quelques uns des Design Patterns[2] couramment utilisés dans une application GNUstep.
Vous constaterez d'ailleurs au fil des articles que le framework GNUstep utilise lui-même intensément nombre de patterns connus.

Modèle – Vue – Contrôleur

Commençons par le très classique Modèle-Vue-Contrôleur, que l'on peut voir sur la figure 1 page suivante.
Ce pattern consiste à séparer en 3 parties une application.
- Le Modèle représente votre partie métier, c'est-à-dire la partie du code indépendante de la partie graphique ou de l'interaction avec l'utilisateur.
- La Vue est chargée de représenter à l'utilisateur le Modèle.
- Le Contrôleur sert de lien entre le Modèle et la Vue.

Cette séparation en trois unités permet une conception plus propre : rien n'empêche de réutiliser votre modèle ailleurs (dans une application GNUstepWeb par exemple), ou de changer complètement votre vue en étant sûr de ne pas affecter le reste de votre application.
Dans notre cas l'interface graphique avec laquelle l'utilisateur va interagir sera la partie «Vue». Cette interface

[1] Rapid Application Development

[2] Design Patterns de Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. ISBN 0201633612

graphique sera créée avec Gorm.

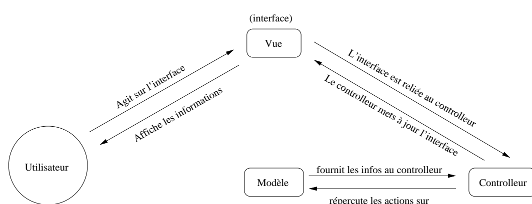


FIG. 1 – Le pattern MVC

La Délégation

Ce pattern consiste à renvoyer vers un objet «aidant», dit délégué, certains travaux. L'approche classique en programmation objet pour améliorer ou spécialiser un objet est de le sous-classer. La délégation consiste à ne pas modifier l'objet, mais à simplement demander certaines infos ou certains traitements à un objet «d'aide». Ce pattern permet souvent de se passer de la création d'une sous-classe, et simplifie d'autant le programme.
Pour reprendre l'analogie donnée par Aaron Hillegass[3], le sous-classage revient à l'approche «Robocop» : pour améliorer le policier, on emploie des dizaines de chirurgiens, et il faut connaître parfaitement le fonctionnement du corps humain. C'est un outil puissant, mais qui peut être complexe à manipuler.
La délégation revient à l'approche «K2000» : pour améliorer Michael, on utilise simplement un outil créé pour lui, la voiture Kit, ayant tout ce qu'il faut comme gadgets indispensables à la vie téméraire d'un justicier à roulettes.
Par exemple, quand un widget NSTableView (affichant un tableau ou une liste) doit s'afficher, au lieu de le sous-classer pour qu'il réponde à nos besoins, on lui fournit un objet délégué.
Quand le NSTableView voudra se dessiner, il demandera simplement à son délégué des choses comme «Combien ai-je de lignes ?» ou «Qu'est-ce qui doit être affiché dans la première colonne de la troisième ligne ?».

Réalisation

Voyons comment nous pouvons appliquer ces patterns à notre programme.

[3]«Cocoa Programming for Mac OS X» de Aaron Hillegass, ISBN 0-201-72683-1

Le Modèle

Notre modèle sera ici la tâche à effectuer, c'est-à-dire un objet contenant toutes les informations liées à une tâche donnée. Nous appellerons cette classe d'objets «Todo». Un objet Todo contiendra pour le moment trois données membres : une chaîne de caractères contenant la description de la tâche, une chaîne de caractères contenant une note éventuellement plus détaillée, et enfin un entier contenant l'indice de progression de la tâche (sur 100).
Chaque objet pourra éventuellement contenir des sous-tâches, c'est-à-dire d'autres objets de classe «Todo».
Pour simplifier les choses, nous allons permettre d'avoir plusieurs tâches au niveau de l'application ; nous aurons ainsi simplement un tableau contenant un ou plusieurs objets «Todo» au niveau du contrôleur.
Voici donc l'interface de notre modèle (mis dans un fichier Todo.h) :

```
#ifndef __TODO_H__
#define __TODO_H__
#include <Foundation/Foundation.h>

@interface Todo : NSObject
{
  NSString *_note;
  NSString *_description;
  int _progress;
  NSMutableArray *_childs;
  id _parent;
}
// Constructeur
-(id) initWithDescription: (NSString*)
    description andNote: (NSString*) note;
// modifieurs
-(void) setDescription: (NSString *)
    description;
-(void) setNote: (NSString *) note;
-(void) setProgress: (int) progress;
-(void) setChilds: (id) childs;
-(void) addChild: (id) child;
-(void) setParent: (id) parent;
-(void) removeChild: (id) child;
// accesseurs
-(NSString *) desc;
-(NSString *) note;
-(int) progress;
-(id) parent;
-(NSArray*) childs;
@end

#endif
```

Chaque objet Todo pourra donc contenir éventuellement des sous-tâches (stockées dans le tableau _childs) ; on pourra accéder à la tâche parente si elle existe en envoyant le message parent :
id tacheParente = [maTache parent] ;
Voici le code de notre modèle (mis dans un fichier Todo.m) :

```
#include "Todo.h"

@implementation Todo

/* Constructeurs */

-(id) init {
  self = [super init];
  _note = [[NSString alloc] init];
  _description = [[NSString alloc] init];
  _childs = [[NSMutableArray alloc] init];
  _parent = nil;
  _progress = 0;
  return self;
}

-(id) initWithDescription: (NSString*)
    description andNote: (NSString*) note {
  self = [super init];
  _description = [[NSString alloc]
      initWithString: description];
  _note = [[NSString alloc] initWithString
      : note];
  _childs = [[NSMutableArray alloc] init];
  _parent = nil;
  _progress = 0;
  return self;
}

/* Destructeur */

-(void) dealloc {
  RELEASE(_childs);
  RELEASE(_note);
  RELEASE(_description);
  [super dealloc];
}

/* Accesseurs */

-(NSString *) desc { return _description; }
-(NSString *) note { return _note; }
-(int) progress; { return _progress; }
-(NSArray *) childs { return _childs; }
-(id) parent { return _parent; }

/* Modifieurs */

-(void) setDescription : (NSString *)
    description {
  [_description release];
  _description = [[NSString alloc]
      initWithString: description];
}

-(void) setNote : (NSString *) note {
  [_note release];
  _note = [[NSString alloc] initWithString
      : note];
}

-(void) setProgress: (int) progress {
  if ((progress >= 0) && (progress < 100))
  {
      _progress = progress;
  }
}

-(void) addChild: (id) child {
  [_childs addObject: child];
}

-(void) setParent: (id) parent {
  ASSIGN (_parent, parent);
}

-(void) setChilds: (id) childs {
  ASSIGN (_childs, childs);
}

-(void) removeChild: (id) child {
  [_childs removeObject: child];
}

- (void) encodeWithCoder: (NSCoder*) coder
    {
      [coder encodeObject: _description];
      [coder encodeObject: _note];
      [coder encodeValueOfObjCType: @encode
          (int) at: &_progress];
      [coder encodeObject: _parent];
      [coder encodeObject: _childs];
}

- (id) initWithCoder: (NSCoder*) coder {
```

```
        if (self = [super init])
        {
                [self setDescription: [coder
                    decodeObject]];
                [self setNote: [coder
                    decodeObject]];
                [coder decodeValueOfObjCType:
                    @encode (int) at: &_progress
                    ];
                [self setParent: [coder
                    decodeObject]];
                [self setChilds: [coder
                    decodeObject]];
        }
        return self;
}

@end
```

La Vue

Même si il est tout à fait possible de développer l'interface «à la main», il existe sous la plateforme de développement GNUstep un outil RAD, clône de l'Interface Builder d'OPENSTEP/MacOSX, appelé Gorm[4].

Bien que le numéro de version ne soit que 0.2.6, Gorm est déjà fort utilisable (et utilisé) pour le développement d'interfaces graphiques.

Gorm fait partie du projet GNUstep et se trouve donc dans le CVS.

Installation de Gorm

```
cvs -z3 -d :pserver :anoncvs@subversions.gnu.org :2401/cvsroot/gnustep co Gorm
cd Gorm
make
make install
```

Vous pouvez aussi récupérer des packages tgz sur le site du projet GNUstep (`http://www.gnustep.org`).

[4]pour Graphic Object Relationship Modeler (ou aussi GNUstep Object Relationship Modeler)

Création de l'interface graphique

> *«Pour le novice ou l'utilisateur occasionnel, l'interface doit être simple et facile à apprendre et à retenir. Elle ne devrait pas nécessiter un réaprentissage après une longue absence de l'ordinateur.»*
> (guide de l'interface NeXT)

Lancez Gorm : `openapp Gorm.app`.
Créons une nouvelle application : Document→New Application.

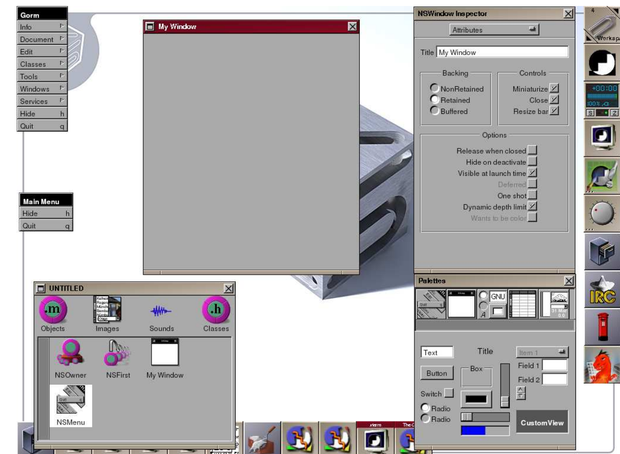


FIG. 2 – Cr´eation d'une application

Dans les outils (Menu Tools) cliquez sur Palettes et Inspector.
La fenêtre palette contient les divers objets nécessaires à la constitution d'une interface (fenêtres, boutons,textfields...)
Dans cette fenêtre (voir figure 3 page suivante), en cliquant sur les différentes sections, nous avons dans l'ordre et de gauche à droite :

- Les Menus et Items de menus (pour le menu de l'application)
- Les Fenêtres/Panneaux
- Les Contrôles (Boutons/TextFields/Sliders..)
- Les Conteneurs (Tableaux/OutlineView (TreeView)/TextView)

La fenêtre Inspecteur représente les différentes vues de l'objet que l'on manipule ; la figure 2 nous montre par exemple dans l'inspecteur les attributs de la fenêtre (NSWindow) de notre application.
Les inspecteurs se retrouvent très souvent dans les applications *step ; cela permet d'afficher au besoin plus de détails sur un document par exemple, au lieu d'encombrer

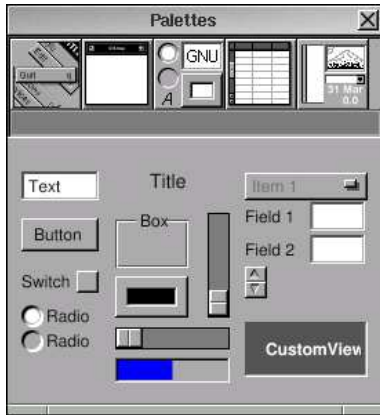


FIG. 3 – Palette

inutilement l'écran avec ces informations quand elles ne sont pas nécessaires.
Dans l'inspecteur de Gorm, nous avons les vues suivantes (en manipulant la liste déroulante) :

- Attributs : permet de definir les attributs du widget sélectionné
- Connection : permet de définir les connexions entre les différents objets (graphiques ou non)
- Size : la taille et le resizing de l'objet
- Help : l'aide

L'interface de notre programme Todo.app sera constituée d'un widget NSOutlineView, affichant la liste des tâches de façon arborescente (puisque nous aurons éventuellement des sous-tâches), accompagné de quelques boutons (ajouter une tâche («+»), ajouter une sous-tâche («<»), supprimer une tâche («-»)).
Pour le moment, la partie inférieure de la fenêtre de Todo.app affichera directement le contenu de la tâche sélectionnée. Elle contiendra deux NSTextFields (Description et Note) et un NSProgressView affichant la progression de la tâche. On ajoutera également un bouton «Update» qui sera utilisé pour mettre à jour une tâche que l'on a modifiée.
Nous changerons ça le mois prochain pour avoir un inspecteur (une fenêtre séparée) sur une tâche au lieu de tout inclure dans la fenêtre principale.
Allez dans la section «Conteneur» du panneau Palette et posez un widget NSOutlineView. Lorsque vous déplacez les objets une fois posés dans la fenêtre, vous verrez des «aimants» (les traits rouges) permettant de positionner correctement vos objets en suivant la guideline (figure 4).

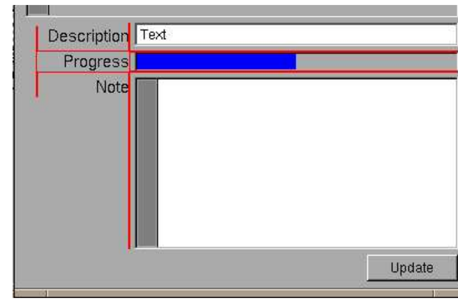
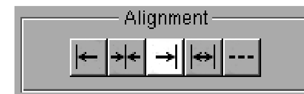


FIG. 4 – Guides pour positionner les objets

Double-cliquez ensuite dessus, puis une deuxième fois sur le titre de la colonne 1 (pour l'éditer), et écrivez «Description». Dans l'inspecteur, mettez l'identifieur DESCRIPTIONTAG à cette colonne. Faites de même avec la seconde colonne avec «Status» en titre de colonne et STATUSTAG en identifieur.
Posez ensuite un NSTextField (qui contiendra la description d'un Todo), un NSTextView (qui contiendra le détail éventuel du Todo), et un NSSlider horizontal pour la progression.
Ajoutez les labels correspondants à côté des widgets. Pour éditer un label, double-cliquez simplement dessus et entrez le texte («Description»,«Note»,«Progression»). Alignons le texte des labels a droite.



Passons au menu ; ajoutons l'item «infos».

Attention l'ordre est important ; mettre par exemple les Préférences ou le Panneau d'information ailleurs que dans le sous menu Infos (qui doit être le premier menu) serait à priori une faute d'ergonomie (du moins, un non-respect de la guideline *step).
La figure 5 page suivante montre le résultat que vous devez obtenir pour l'interface graphique.

Connexion de la vue au contrôleur

Nous avons maintenant une Vue (notre interface graphique), ainsi qu'un modèle (notre classe Todo). Il nous manque un contrôleur pour faire fonctionner le tout.

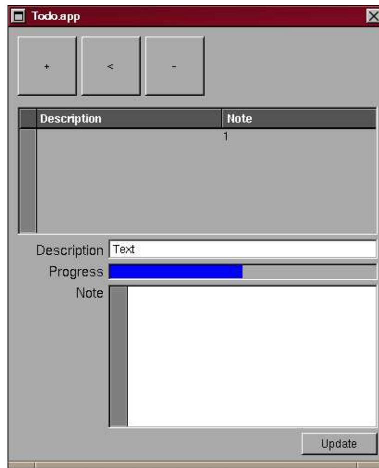


FIG. 5 – L'interface graphique de notre Application

Quelles vont être les actions du contrôleur ? On veut pouvoir :

– visualiser une tâche
– ajouter une tâche : addTodo
– ajouter une sous-tâche : addSubTodo
– supprimer une tâche : removeTodo
– mettre à jour une tâche : updateTodo

Notre contrôleur stockera la liste des tâches dans un tableau (NSMutableArray), les différentes actions seront directement reliées aux boutons de notre interface. Un clic sur une ligne de notre liste de tâches mettra à jour les champs Note, Description, pour visualiser le contenu de la tâche sélectionnée.

Le contrôleur devra donc avoir accès aux champs Note et Description, ainsi qu'au NSOutlineView listant les tâches, pour mettre à jour leur état. Le contrôleur aura donc des «pointeurs» vers ces widgets ; ce type de pointeurs est appellé «outlet» dans la terminologie GNUstep.

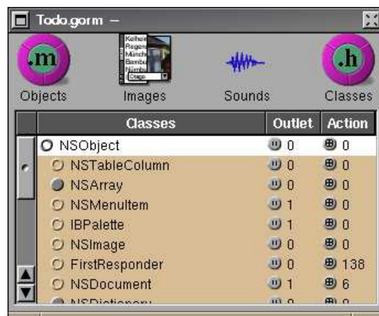


FIG. 6 – Le gestionnaire de classes

Créons notre contrôleur. Sous Gorm, dans le panneau Document allez dans le Class Manager (figure 6). Nous allons sous-classer la classe NSObject. Cliquez sur NSObject pour la sélectionner si ce n'est déja fait.

Dans le Menu «Classes» de Gorm (je vous conseille de le détacher et de le mettre près du panneau Document), sélectionnez «Create Subclass...». Une nouvelle classe apparaît, appellée «NewClass». Double-cliquez dessus pour changer son nom en «TodoController», puis appuyez sur entrée pour valider le changement de nom.

Cliquez sur le rond gris dans la colonne «Outlet» ressemblant à une espèce de prise électrique, et ajoutez un Outlet (Classes→Add Outlet/Action). Renommez l'outlet en «descriptionText». Ajoutez ensuite successivement les outlets «noteTextView» et «todolistView» (figure 7).

FIG. 7 – Ajout des outlets

Désélectionnez la classe puis cliquez sur le deuxième point gris de TodoController représentant les Actions et ajoutez (toujours par Classes→Add Outlet/Action) les actions addTodo, addSubTodo, removeTodo, et updateTodo (figure 8).

FIG. 8 – Ajout des actions

Notre contrôleur est prêt à être utilisé. Re-sélectionnez la classe TodoController dans le Class Manager et cliquez sur le menu Classes→Instanciate. Une instance de la classe TodoController (figure 9 page suivante) se retrouve alors dans la section Object du panneau Document.

Il reste à connecter nos objets à cette instance de la classe TodoController.

FIG. 9 – la classe TodoController instanci´ee

Cliquez sur l'instance de TodoController en laissant appuyé la touche control <Ctrl>. L'icône (comme Source) apparaît, on tire la souris jusqu'au widget NSOutlineView utilisé pour lister nos tâches. L'icône (comme Target) apparaît, on relache la souris. Dans l'inspecteur, cliquez sur l'outlet «todolistView» proposé, puis sur «Connect». L'outlet «todolistView» de notre contrôleur est ainsi connecté avec le widget NSOutlineView.
On fait de même pour connecter les outlets restants (champs Note et Description).
On effectue la manipulation inverse (appui sur control, etc.), des boutons («+»,«<»,«-» et «Update») de notre interface vers l'icone de l'instance TodoController pour connecter les Actions ; dans l'inspecteur, cliquez sur «target» puis choisissez les actions du contrôleur correspondantes aux boutons.
Notre contrôleur est ainsi connecté avec les actions et outlets dont il a besoin.
Refaites la même manipulation entre l'outline view (la liste des tâches) et le contrôleur, avec l'outline view en tant que source et le contrôleur en temps que cible. Dans l'inspecteur, sélectionnez l'outlet «dataSource» de l'outline view et cliquez sur «Connect». Refaites encore la même chose mais cette fois en connectant l'outlet «delegate» de l'outline view au contrôleur. Ainsi, l'outline view demandera au contrôleur les infos nécessaires à son affichage.
Sauvegardons le tout (Document→Save As...) sous le nom Todo.gorm.

Navigation au clavier

Il est possible de connecter les objets graphiques entre eux, pour indiquer dans quel ordre les objets s'activeront quand on naviguera au clavier (touche tab).
Pour cela sélectionnez le bouton Ajouter («+») et appuyez simultanément sur la touche «control» (CTRL) et le bouton de la souris.

Vous avez alors l'image qui s'affiche dans le bouton Ajouter, tirez alors la souris jusqu'au boutton Ajouter une sous-tâche («<»). L'image apparaît alors. Dans l'Inspecteur sélectionnez alors nextKeyView. et cliquez sur «connect». Procédez de même entre le bouton Ajouter une sous-tâche («<») et le bouton Supprimer («-»), puis entre Supprimer et le champs Description, puis entre le champs Description et le champs Note, entre le champs Note et le bouton Update, et finalement, bouclez entre le bouton Update et le bouton Ajouter («+»). Cela permettra à l'utilisateur de cycler entre tous les widgets en utilisant la touche «Tab».
Ainsi votre interface sera plus facilement utilisable au clavier. Pensez à sauver vos ajouts.

Code du Contrôleur

Nous pouvons finir de nous occuper de notre contrôleur : dans le panneau Document, retournez dans le Class Manager, et sélectionnez TodoController. Puis cliquez dans le menu «Classes» de Gorm et selectionnez l'entrée «Create Class Files». Ainsi le squelette de notre contrôleur sera automatiquement généré par Gorm. Nous rajoutons dans le contrôleur un tableau pour contenir la liste des Todo, que l'on appelle todoArray.
Pour le moment notre contrôleur reste très simple :

```
@interface TodoController : NSObject
{
  id descriptionText;
  id noteTextView;
  id todolistView;
  NSMutableArray *todoArray;
}
- (void) addTodo: (id)sender;
- (void) addSubTodo: (id)sender;
- (void) removeTodo: (id)sender;
- (void) updateTodo: (id)sender;
```

Nous allons y rajouter les méthodes «Data Source» utilisées par l'outline view pour son affichage :

```
- (id) outlineView: (NSOutlineView *)
    outlineView child: (int) index ofItem: (
    id) item;
- (int)outlineView: (NSOutlineView *)
    outlineView numberOfChildrenOfItem: (id
    ) item;
- (BOOL)outlineView: (NSOutlineView *)
    outlineView isItemExpandable: (id) item
    ;
- (id)outlineView: (NSOutlineView *)
    outlineView objectValueForTableColumn
```

```
    : (NSTableColumn *) tableColumn byItem
    : (id) item;
```

Si notre objet TodoController répond à ces méthodes, l'outline view s'en servira pour son affichage. Les méthodes sont assez simples en fait :

- La première doit retourner le fils placé à la position index de l'objet item.
- La seconde doit retourner le nombre de fils que contient un objet item.
- La troisième méthode doit retourner YES (vrai) si l'objet passé en paramètre item contient ou non des fils.
- La quatrième méthode doit retourner la valeur de la cellule pour une colonne donnée par rapport à l'objet fourni en paramètre (donc d'une ligne de l'outline view). On peut se servir des «tags» que l'on a rentré tout à l'heure dans Gorm pour faire la sélection.

On peut noter que le widget NSOutlineView est dérivé du widget NSTableView ; son fonctionnement est un peu plus complexe (structure arborescente oblige), mais est basé sur les mêmes principes. En fait, la version originale de cet article utilisait un NSTableView, et ne proposait pas une structure arborescente pour les Todo, mais puisqu'on a été décalé d'un mois, on en a profité pour rajouter des trucs ;)

On ajoute également à notre contrôleur une fonction déléguée de l'outline view, qui sera appelée quand l'utilisateur sélectionnera un item (un Todo) ; ainsi on pourra mettre à jour les champs Note, Description et Progression par rapport au Todo sélectionné.

L'interface du contrôleur (on le sauvera dans un fichier TodoController.h) sera donc :

```
#ifndef __TODOCONTROLLER_H__
#define __TODOCONTROLLER_H__
#include <AppKit/AppKit.h>

@interface TodoController : NSObject
{
  id descriptionText;
  id noteTextView;
  id todolistView;
  id sliderView;
  id progressView;
  NSMutableArray *todoArray;
}

// Actions
- (void) addTodo: (id)sender;
- (void) addSubTodo: (id)sender;
- (void) removeTodo: (id)sender;
- (void) updateTodo: (id)sender;
- (void) setProgress: (id)sender;

// NSOutlineView Data Source
- (id) outlineView: (NSOutlineView *)
   outlineView child: (int) index ofItem: (
   id) item;
- (int)outlineView: (NSOutlineView *)
   outlineView numberOfChildrenOfItem: (id
   ) item;
- (BOOL)outlineView: (NSOutlineView *)
   outlineView isItemExpandable: (id) item
   ;
- (id)outlineView: (NSOutlineView *)
   outlineView objectValueForTableColumn
   : (NSTableColumn *) tableColumn byItem
   : (id) item;

// NSOutlineView méthode déléguée
- (BOOL)outlineView: (NSOutlineView *)
   outlineView shouldSelectItem: (id) item
   ;

@end
#endif
```

Il ne reste donc qu'à rajouter le code correspondant (dans un fichier TodoController.m) :

```
#include "Todo.h"
#include "TodoController.h"

@implementation TodoController

-(id) init {
  self = [super init];
  todoArray=[[NSMutableArray alloc] init];
  return self;
}

-(void) dealloc {
  RELEASE(todoArray);
  [super dealloc];
}

// Actions

-(void) addTodo:(id) sender {
  Todo *aTodo = [[Todo alloc]
      initWithDescription: [descriptionText
       stringValue]
      andNote: [noteTextView string]];
  [todoArray addObject: aTodo];
```

```
  [aTodo release];
  [todolistView reloadData];
}

-(void) addSubTodo:(id) sender {
  int row = [todolistView selectedRow];
  if (row != -1)
  {
    id item = [todolistView itemAtRow: row
        ];
    Todo *aTodo = [[Todo alloc]
        initWithDescription: [
        descriptionText stringValue]
                andNote: [noteTextView
                    string]];
    [aTodo setParent: item];
    [item addChild: aTodo];
    [aTodo release];
    [todolistView reloadData];
  }
}

-(void) removeTodo:(id) sender {
  int selectedRow = [todolistView
      selectedRow];
  if (selectedRow != -1)
  {
     id item = [todolistView itemAtRow:
         selectedRow];
     id parent = [item parent];
     if (parent != nil)
     {
         [parent removeChild: item];
     }
     else
     {
         [todoArray removeObject: item];
     }
     [todolistView reloadData];
  }
}

-(void) updateTodo:(id) sender {
  int selectedRow = [todolistView
      selectedRow];
  if (selectedRow != -1)
  {
     Todo* todo = [todolistView itemAtRow:
         selectedRow];
     [todo setDescription: [descriptionText
           stringValue]];
     [todo setNote: [noteTextView string]];
     [todo setProgress: [progressView
         doubleValue]];
     [todolistView reloadData];
  }
}

-(void) setProgress:(id) sender {
     [progressView setDoubleValue: [sender
         intValue]];
}

/* méthodes dataSource de l' outline view */

- (id) outlineView: (NSOutlineView *)
    outlineView child: (int) index ofItem: (
    id) item
{
  if (item == nil) // racine
   {
    return [todoArray objectAtIndex: index
        ];
   }
  return [[item childs] objectAtIndex: index
      ];
}

- (int)outlineView: (NSOutlineView *)
    outlineView numberOfChildrenOfItem: (id
    ) item
{
  if (item == nil) return [todoArray count];
  return [[item childs] count];
}

- (BOOL)outlineView: (NSOutlineView *)
    outlineView isItemExpandable: (id) item
{
  if ([[item childs] count] > 0) return YES;
  return NO;
}

- (id)outlineView: (NSOutlineView *)
    outlineView objectValueForTableColumn
    : (NSTableColumn *) tableColumn byItem
    : (id) item
{
  if ( [[tableColumn identifier]
      isEqualToString: @"DESCRIPTIONTAG"
      ] )
   {
```

```
    if (item == nil)
      return [[todoArray objectAtIndex: 0]
          desc];

    return [item desc];
  }
  else if ( [[tableColumn identifier]
      isEqualToString: @"STATUSTAG"] )
    return [NSString stringWithFormat:@"%i/100
        ", [item progress]];
  return [NSString stringWithString: @"VOID"
      ];
}

/* méthodes delegate de l' outline view */

- (BOOL)outlineView: (NSOutlineView *)
    outlineView shouldSelectItem: (id) item
{
  if (item)
  {
    [descriptionText setStringValue: [item
        desc]];
    [noteTextView setString: [item note]];
    [progressView setDoubleValue: [item
        progress]];
    [sliderView setIntValue: [item
        progress]];
    return YES;
  }
  return NO;
}

@end
```

Le main de notre programme

Le main de notre programme se contente d'appeler l'objet NSApplication et de faire les initialisations nécessaires :

```
int main(int argc, const char *argv[])
{
      return NSApplicationMain(argc, argv);
}
```

On le sauvegarde dans un fichier main.m.

Les Makefiles

GNUstep fourni son propre système de gestion de projet pour les différentes machines et environnement sans passer par les complexes autoconf/automake. Le fichier makefile doit s'appeller GNUmakefile.
gnustep-make est basé sur des variables d'environnements. Voici un exemple de GNUmakefile pour un outil (TOOL) – c'est-à-dire une application non graphique. Les différents scripts se trouvent dans /System/Makefiles/

```
include $(GNUSTEP_MAKEFILES)/common.make
TOOL_NAME= MonOutils
MonOutils_OBJC_FILES=main.m source.m
include $(GNUSTEP_MAKEFILES)/tool.make
```

Voici une petite liste de variables courramment utilisées ; On les préfixes par le <nom> de l'application graphique :

– SUBPROJECT= liste des sous-projets
– <nom>_OBJC_FILES les fichiers ObjC
– <nom>_C_FILES les fichiers C
– <nom>_HEADERS les en-têtes des notre programme
– <nom>_HEADER_FILES Les en-têtes des notre programme que l'on veut installer (dans le cas d'un Framework)
– <nom>_RESOURCE_FILES les ressources nécessaires (les images,.gorm...)
– <nom>_LOCALIZED_RESOURCE_FILE les ressources localisables
– <nom>_LANGUAGE Langues supportées

Voici donc le fichier GNUmakefile de notre application Todo.app :

```
include $(GNUSTEP_MAKEFILES)/common.make
APP_NAME=Todo
Todo_OBJC_FILES=main.m TodoController.m
    Todo.m
Todo_RESOURCE_FILES=Todo.gorm
Todo_MAIN_MODEL_FILE=Todo.gorm
include $(GNUSTEP_MAKEFILES)/application.
    make
```

Compilons : make
Notre application se nomme Todo.app.
Pour le moment, vous pouvez rajouter, modifier et supprimer des tâches ... mais il nous manque un léger détail : la sauvegarde et la lecture de fichiers Todo !

Archiver un objet

Archiver un objet consiste à le transformer en un flux binaire, indépendant de l'architecture, qui préserve l'identité et les relations entre les objets et leur valeurs. Cela

permet par exemple d'envoyer un objet sur le réseau ou de le sauver sur disque simplement. Désarchiver un objet, c'est réaliser l'opération inverse, recréer un objet à partir d'un flux binaire. GNUstep permet très simplement cela.
Pour qu'un objet puisse être archivé/désarchivé, il suffit qu'il réponde au protocole NSCoding, c'est-à-dire qu'il réponde aux messages encodeWithCoder : (pour archiver un objet) et initWithCoder : (pour désarchiver un objet).
Ces deux fonctions reçoivent un objet de type NSCoder en paramètre. La classe abstraite NSCoder sert à représenter le flux binaire, et à lui rajouter les différentes données membres de l'objet que l'on veut archiver. En effet, l'objet est responsable de l'archivage de ses données membres. Il n'est pas obligé d'archiver toutes ses données membres (certaines peuvent ne pas être importantes par exemple). Il faut par contre veiller à ce que l'ordre d'archivage et de désarchivage des données soit le même sous peine de problèmes (voir le source suivant) ! Si vous voulez archiver un objet dérivant d'un objet qui répond également au protocole NSCoding, vous devez alors ajouter la ligne :

```
[super encodingWithCoder: coder];
```

au début de la fonction d'archivage encodeWithCoder et

```
self = [super initWithCoder: coder];
```

au début de la fonction de désarchivage initWithCoder, de façon à ce que les données membres de l'objet père puissent être éventuellement archivées si besoin est.
Dans notre cas, ce n'est pas nécessaire, Todo dérivant directement de NSObject.
Les objets du framework GNUstep savent directement s'archiver ou se désarchiver, il suffira donc d'utiliser la fonction encodeObject : sur l'objet coder passé en paramètre.
Par contre, pour des types C de base, il faut passer par les fonctions encodeValueOfObjcType : et decodeValueOfObjCType : auquel on passe une macro @encode() définissant le type et l'adresse de la variable que l'on veut archiver.
Nous rajoutons donc les fonctions encodeWithCoder : et initWithCoder : à notre objet Todo :

```
/* Archivage de l'objet */

- (void) encodeWithCoder: (NSCoder*) coder
    {
      [coder encodeObject: _description];
      [coder encodeObject: _note];
      [coder encodeValueOfObjCType: @encode
          (int) at: &_progress];
      [coder encodeObject: _parent];
      [coder encodeObject: _childs];
}

- (id) initWithCoder: (NSCoder*) coder {
      if (self = [super init])
      {
            [self setDescription: [coder
                decodeObject]];
            [self setNote: [coder
                decodeObject]];
            [coder decodeValueOfObjCType:
                @encode (int) at: &_progress
                ];
            [self setParent: [coder
                decodeObject]];
            [self setChilds: [coder
                decodeObject]];
      }
      return self;
}
```

Et voilà ! nos objets Todo savent désormais s'archiver et se désarchiver. On doit maintenant ajouter des fonctions pour lire et sauver nos Todo dans notre contrôleur.

Modification de TodoController

On rajoute deux prototypes de méthodes dans TodoController.h :

```
-(void) saveFile: (id) sender;
-(void) loadFile: (id) sender;
```

On rajoute le corps de ces fonctions dans TodoController.m :

```
-(void) saveFile: (id) sender {
  int ret;
  NSSavePanel* panel = [NSSavePanel
      savePanel];
  [panel setRequiredFileType: @"todo"];
  [panel setDirectory: [[NSFileManager
      defaultManager] currentDirectoryPath
      ]];

  ret = [panel runModal];
  if (ret == NSFileHandlingPanelOKButton)
  {
     [[NSArchiver
        archivedDataWithRootObject:
        todoArray] writeToFile: [panel
        filename] atomically: YES];
```

```
  }
}

-(void) loadFile: (id) sender {
  int ret;
  NSOpenPanel* panel = [NSOpenPanel
      openPanel];
  [panel setAllowsMultipleSelection: NO];
  [panel setDirectory: [[NSFileManager
      defaultManager] currentDirectoryPath
      ]];

  ret = [panel runModalForTypes: [NSArray
      arrayWithObject: @"todo"]];
  if (ret == NSOKButton)
  {
      id file = [NSUnarchiver
          unarchiveObjectWithData:
              [NSData dataWithContentsOfFile
                  : [[panel filenames]
                  objectAtIndex: 0]]];
      ASSIGN (todoArray, file);
      [todolistView reloadData];
  }
}
```

Quelques remarques : on a choisi d'utiliser ici le FileType «todo», c'est à dire l'extension .todo, pour nos fichiers sauvegardés. Le point intéressant est ici l'utilisation des classes NSArchiver et NSUnarchiver, auquel on passe simplement le tableau todoArray contenant nos objets Todo.

Modification du fichier gorm

Maintenant que tout fonctionne parfaitement et que notre application sait lire et sauver des fichiers Todos, il serait peut être intéressant de permettre à l'utilisateur de le faire...
Relançons Gorm sur notre fichier Todo.gorm : gopen Todo.gorm. Allez dans le Class Manager, et rechargez la classe TodoController («Classes→Load Class...» puis lire le fichier TodoController.h). Maintenant, rajouter deux items au menu de notre application :

Ensuite, reliez simplement les entrées «Open...» et «Save as...» au contrôleur TodoController, et connectez les aux actions «loadFile :» et «saveFile :» maintenant présentes dans le contrôleur. Sauvez le fichier Gorm, refaites un make, et voila !
Vous avez maintenant une version de Todo.app qui fonctionne, certes de façon basique, mais complètement. Le mois prochain nous modifierons le programme pour ajouter un Inspecteur ... d'ici là, n'hésitez pas à passer sur le channel #gnustep sur l'irc freenode (irc.debian.org par exemple) !

Nicolas Roard, <nicolas@roard.com>
Fabien Vallon, <fabien@tuxfamily.org>

Merci à Vincent Ricard et Cyril Siman pour la relecture de cet article !

Quelques liens :

Le wiki GNUstep : http://wiki.gnustep.org/
La page tutoriels du wiki : http://wiki.gnustep.org/index.php/Tutorials
Un très bon tutorial de Yen-Ju : http://www.people.virginia.edu/~yc2w/GNUstep/Tutorial/