Lazarus - Développement rapide : Le Free Pascal

par Matthieu Giroux (TheBookEdition)

Date de publication : 18/07/2010

Dernière mise à jour : 25/08/2010

L'intérêt de LAZARUS, c'est
le Développement rapide
et les composants RAD.
Un composant utilise l'objet
et son EDI pour s'installer
facilement, se standardiser
avec l'ensemble des
composants. Ainsi
le développeur n'est
jamais dépaysé par de
nouveaux modèles.

I - Je crée mon premier logiciel 4
I.A - 4
I.A.1 - Introduction 4
I.A.2 - A faire avant 4
I.A.3 - L'Exemple 4
I.A.4 - Création de l'Interface Homme Machine 4
I.A.5 - Tester ses composants 4
I.A.6 - L'exemple 4
I.A.6.a - Une présentation soignée 4
I.A.6.b - L'Interface Homme Machine 5
I.B - Caractères accentués 6
I.C - Chercher des composants ou projets 6
I.C.1 - Vérifier les licences 7
I.C.2 - Compilateur FREE PASCAL 7
I.D - Gestion des erreurs 7
I.E - Les exceptions 7
I.E.1 - Renvoyer les exceptions 8
I.F - Bases de données 8
I.F.1 - Le pouvoir du libre 8
I.G - Bases de données embarquées 9
II - L'objet 9
II.A - Un Objet 9
II.B - Une classe 9
II.B.1 - Les attributs et propriétés 9
II.B.2 - Les méthodes et événements 10
II.C - L'Héritage 11
II.C.1 - La surcharge 11
II.C.1.a - Le mot clé « virtual » 11
II.C.1.b - Le mot clé « override » 12
II.D - L'Encapsulation 12
II.D.1 - La déclaration « private » 12
II.D.2 - La déclaration « protected » 13
II.D.3 - La déclarations « public » 13
II.D.4 - La déclaration « published » 13
II.D.5 - Les constructeurs et destructeurs 14
II.E - Le Polymorphisme 14
II.E.1 - Le type « interface » 15
II.E.2 - Polymorphe avec les propriétés publiées 15
II.F - L'Abstraction 15
II.G - Une Instance d'Objet 16
II.H - Les propriétés 16
III - L'UML pour programmer en Objets 16
III.A - STAR UML 16
IV - Créer son savoir-faire 16
IV.A - Créer des unités de fonctions 17
IV.B - Les composants 17
IV.C - Créer un composant 17
IV.C.1 - Choix de l'image 20
IV.D - Enregistrement du composant 21
IV.D.1 - Modifier ou Surcharger ? 22
IV.D.2 - Surcharger un composant 22
IV.E - Créer une librairie homogène 22
IV.F - Le libre et l'entreprise 23
V - De PASCAL vers FREE PASCAL 23
V.A - Introduction 23
V.B - De TURBO PASCAL vers FREE PASCAL 23
V.B.1 - Choisir un gestionnaire de données 23
V.C - De DELPHI vers LAZARUS 23

V.D - Traduction de composants.....24

I - Je crée mon premier logiciel

I.A -

I.A.1 - Introduction

Voici comment procéder pour créer son premier logiciel.

I.A.2 - A faire avant

Avant de créer son premier logiciel, il faut d'abord savoir si c'est utile de créer le logiciel voulu. Il faut d'abord vérifier que ce qu'on veut ne se fait pas déjà. A l'heure actuelle, seuls les logiciels avec une personnalisation accrue sont à créer.

I.A.3 - L'Exemple

Dans notre premier exemple, nous allons tester la création de notre interface homme machine. On créera un utilitaire de recherche ou de remplacement à personnaliser. Nous créerons aussi notre propre savoir-faire centralisé.

I.A.4 - Création de l'Interface Homme Machine

Un logiciel se doit d'être créé rapidement. Nous allons rapidement créer une interface Homme Machine. LAZARUS est un outil permettant de créer rapidement un logiciel grâce à son interface graphique et du code uniquement dédié au développement.

I.A.5 - Tester ses composants

Avant de commencer, il faut savoir s'il est possible de créer rapidement son logiciel. Aller dans « Fichier » puis ..
Pour créer rapidement son logiciel, il suffit de placer correctement les composants après avoir cliqué sur un composant de la palette de composants. La palette de composants comporte un ensemble d'onglets identiques à des onglets de classeur. Lorsque vous cliquez sur un onglet portant un certain nom, un ensemble de composants classés sous ce nom s'affichent.
Vous voyez qu'il est possible de placer un certain nombres de composants sans limites de présentation.
Certains composants que vous placez sur votre formulaire ne seront représentés que par le même graphisme de même taille que son icône situé dans la palette. Ces composants n'ont en général aucune propriété graphique. Ce sont des composants invisibles capables de réaliser certaines opérations visuelles ou pas.
D'autres composants que vous placerez ajouteront un visuel à votre formulaire. Ce sont des composants visuels. Avant de placer vos composants visuels, n'oubliez pas d'utiliser un composant "TPanel" pour une présentation soignée. Les Composants "TPanel" permettent à vos composants visuels de se disposer sur l'ensemble de l'espace visuel disponible.

I.A.6 - L'exemple

Disposez un composant de présentation.

I.A.6.a - Une présentation soignée

Pour commencer notre Interface Homme Machine, il faut d'abord disposer un composant "TPanel". Les "TPanel" servent à ce que l'interface utilise le maximum de l'espace de travail.

Notre premier "TPanel" servira à disposer les boutons. Cliquez sur F11 pour afficher l'inspecteur d'objet. Votre "TPanel" y est sélectionné. Cliquez sur la boîte à option de la propriété « Align » et affectez « alBottom » à « Align ».
Le deuxième "TPanel" servira à disposer le composant de sélection du répertoire. Cliquez sur F11 pour afficher l'inspecteur d'objet. Votre "TPanel" y est sélectionné. Cliquez sur la boîte à option de la propriété « Align » et affectez « alTop » à « Align ».
Le troisième « TPanel » affichera les fichiers à convertir.
Disposez un deuxième « TPanel » dans la zone où il n'y a pas de « TPanel ».
Votre "TPanel" doit donc s'aplatir sur tout le formulaire. Affectez « alClient » à la propiété « Align » de ce « TPanel » afin de disposer d'un maximum d'espace. Vous pouvez enlever les bordures de vos « TPanel »...

I.A.6.b - L'Interface Homme Machine

Ensuite, il faut disposer les composants de sélection de répertoire et de fichiers dans les "TPanel". Il faut aussi disposer les boutons.
Disposez un composant « TDirectoryEdit » sur le panneau « TPanel » du haut. Le composant « TDirectoryEdit » peut sembler insatisfaisant graphiquement. Vous pouvez consulter le chapitre suivant pour rechercher un meilleur savoir-faire.
Il est possible d'ancrer le composant « TDirectoryEdit » sur la longueur du « TPanel » ou bien de choisir un composant de sélection de répertoire plus abouti. Les composants fournis avec LAZARUS sont testés sur l'ensemble des plateformes disponibles.
Disposez un composant de sélection de fichiers « TFileListBox » dans le panneau du milieu. Vous pouvez lui affecter la valeur « alClient » à la propriété « Align ». Vos fichiers seront disposés sur l'ensemble de l'interface et donc visibles pour l'utilisateur.
Disposez les boutons « TButton » sur le panneau du bas. Les boutons n'ont pas besoin d'être alignés car le texte qui leur sera affecté aura une longueur connue. Ils sont alignés à partir de la gauche. Vous pouvez par exemple créer un bouton qui affiche la date du fichier ou son chemin, etc.
Nous allons créer un logiciel de conversion de fichiers LAZARUS.
Créez un bouton avec le nom « Convertir ». Vous voyez que son libellé visuel change. Vous avez affecté aussi la propriété « Caption » du composant. Maintenant, il faudra changer le nom et son libellé visuel séparément.
Double-cliquez sur l'événement « Onclick » du bouton.
Placez ce code source :

```
procedure TForm1.ConvertirClick(Sender: TObject);
var li_i : Integer ;
begin
  For Li_i := 0 To FileListBox.Items.Count - 1 do
    Begin
      ConvertitFichier ( DirectoryEdit.Directory + DirectorySeparator + FileListBox.Items.Strings
 [ li_i ], FileListBox.Items.Strings [ li_i ] );
    End ;
end;
```

Ne compilez pas. Il manque la procédure **ConvertitFichier** ici :

```
procedure TForm1.ConvertitFichier(const FilePath, FileName : String);
var li_i : Integer ;
    ls_Lignes : WideString ;
    lb_DFM ,
    lb_LFM : Boolean ;
    lst_Lignes : TStringList ;
begin
  lb_DFM := PosEx ( '.dfm', LowerCase ( FilePath ), length ( FilePath ) - 5 ) >= length ( FilePath )
 - 5 ;
  lb_LFM := PosEx ( '.lfm', LowerCase ( FilePath ), length ( FilePath ) - 5 ) >= length ( FilePath )
 - 5 ;
  lst_Lignes := TStringList.Create;
  try
    lst_Lignes.LoadFromFile ( FilePath );
    ls_Lignes := lst_Lignes.Text;
```

```
    Application.ProcessMessages;
        // Conversion Extended
    ls_Lignes := StringReplace ( ls_Lignes, 'TADOQuery', 'TZQuery', [rfReplaceAll,rfIgnoreCase] );

     if lb_DFM
     or lb_LFM Then
       Begin
       End;
      else
       Begin
       End;

    Application.ProcessMessages ;
    lst_Lignes.Text :=  ls_Lignes ;
    lst_Lignes.SaveToFile   ( FilePath );
  finally
    lst_Lignes.Free;

  end;
end;
```

Une fois que vous avez créé la procédure ConvertitFichier, appuyez simultanément sur « Ctrl »puis « Shift » puis « C ». Ainsi, la procédure ConvertitFichier sera renseignée dans votre fiche en fonction de la déclaration.
Au début, on teste si l'extension de fichier est « lfm », « dfm », ou « pas ».
La fonction ProcessMessages de TApplication sert pendant les boucles. Elle permet à l'environnement de travailler afin d'afficher la fenêtre.
La fonction StringReplace retourne une chaine dont une sous-chaine a éventuellement été remplacée. Elle possède des options comme le remplacement sur toute la chaine, la distinction ou pas des majuscules et minuscules.
A la fin, on affecte le « TStringlist » par sa propriété « Text », puis on remplace le fichier en utilisant la procédure « SaveToFile » avec le nom de fichier passé en paramètre de la procédure « ConvertitFichier ». Il faut donc faire une copie du projet avant de le convertir par cette moulinette.
Vous pouvez maintenant vous amuser à changer les composants de projets LAZARUS en série.
Il n'y a plus qu'à filtrer les fichiers de la liste de fichiers en fonctions du résultat demandé par l'utilisateur : Convertir tout, des fichiers « lfm », des fichiers « dfm » ou des fichiers « pas ».
Notre interface est prête à être renseignée. Ce qui est visible par l'utilisateur a été créé rapidement. Nous n'avons pas eu besoin de tester l'interface. Pour voir le comportement de cet exemple, exécutez en appuyant sur « F9 » ou en cliquant sur le bouton ... .
Vous pouvez agrandir votre formulaire ou le diminuer pour tester votre interface.

I.B - Caractères accentués

Si vous observez que les caractères accentués ne sont pas gardés, c'est à cause des caractères ANSI. En effet, il existe des caractères deux fois plus longs, les caractères UTF16. Ces caractères permettent une lecture presque internationale pour les pays possédant un alphabet de lettres.
Il faut vérifier si le TStringlist utilise des WideStrings. Ce sont des chaînes avec des caractères deux fois plus volumineux.

I.C - Chercher des composants ou projets

Ce chapitre est une aide servant à améliorer vos projets grâce à INTERNET.
LAZARUS possède à son installation un nombre limité de composants. Il est possible de trouver des savoir-faire sur INTERNET. Pour trouver un composant, il faut définir ce que l'on souhaite ainsi que les alternatives possibles. Ce ne sera qu'avec l'expérience que l'on trouvera des composants adéquats.
Sur votre moteur de recherche, il faudra de préférence taper des mots anglais comme « component », « project » ou « visual » avec « LAZARUS » voire « DELPHI ». Puis avec ces mots tapez votre spécification en anglais. Changez de mots ou de domaines si vous ne trouvez pas.

I.C.1 - Vérifier les licences

Après avoir téléchargé, il faut vérifier l'utilisation des licences. Chaque composant possèdera des particularités de licence avec le mot « Modified » ou un autre mot générique.
Si vous avez téléchargé des composants gratuits, voici les différents types de licences que vous pouvez trouver :

- Aucun fichier de licence : En général vous faites ce que vous voulez. Mais il faut avoir contacté l'auteur.
- GPL : Cette licence vous oblige à diffuser les sources gratuitement de votre logiciel ou descendant si vous diffusez le composant, ceci en respectant l'auteur.
- Creative Common : Cette licence vous oblige à respecter l'auteur du composant.
- BSD : Cette licence libre vous autorise à revendre le composant comme vous le voulez.
- Etc.

Il faut savoir qu'une licence commerciale pour laquelle vous avez acheté des composants peut disposer d'un ensemble de restrictions à lire.

I.C.2 - Compilateur FREE PASCAL

Le compilateur qui a permis de créer LAZARUS est aussi celui utilisé pour créer les programmes. La seule différence est qu'il est semi-automatisé et qu'il utilise les librairies LAZARUS. Pour compiler sur LAZARUS et vérifier son code créé, appuyez sur « Ctrl » et « F9 » en même temps. Pour exécuter sur LAZARUS cliquez sur le bouton Play ( ) ou sur « F9 ».
Vous pouvez exécuter ligne après ligne votre programme et appuyant sur « F7 » ou « F8 ». Ces touches permettent de voir le code s'exécuter en regardant vos sources. La touche « F8 » passera certaines sources non scutées dans votre éditeur.
L'utilisation des touches « F7 » et « F8 » est fastidieuse. Vous pouvez ajouter un voire plusieurs points d'arrêts en cliquant dans votre éditeur de code sur la marge grisée de la ou des lignes à vérifier.
Le compilateur FREE PASCAL crée des fichiers exécutables volumineux. Pourquoi ? Parce que LAZARUS ne déporte pas l'utilisation des librairies de chaque environnement. Autrement dit, votre exécutable peut ne pas utiliser les librairies que vous aurez utilisées pour créer l'exécutable. Un exécutable LAZARUS a de très grandes chances de fonctionner seul.

I.D - Gestion des erreurs

Dans votre logiciel, le développeur et l'utilisateur doivent être guidés. Des erreurs peuvent se produire dans le code servant au développement ou celui servant à l'utilisateur. En général, les erreurs de développement doivent être en anglais tandis que les erreurs de l'utilisateur sont multilingues.
Il faut donc anticiper ce qui va se passer dans son programme. Imaginez un utilisateur tombant sur une erreur en anglais alors qu'il ne connaît pas cette langue. Il faut donc en plus un manuel de l'utilisateur permettant de trouver les informations nécessaires à la poursuite de l'utilisation du logiciel.

I.E - Les exceptions

Les exceptions permettent de gérer les erreurs ou bien de les rediriger correctement vers l'utilisateur. Méfiez-vous des logiciels qui rapportent peu d'erreurs. En général, on ne sait pas comment les utiliser.

```
try
  for i:=0 to 20 do
    chaine := chaine + chaine;
except
    Showmessage ( 'La chaine de ma formulaire est trop longue. Il n'y a plus assez de mémoire.' );
end;
```

Cette gestion d'exception commence à « try » et finit à « end ». La partie entre « try » et « except » rapporte vers la partie entre « except » et « end » l'erreur éventuelle.
Dans cette gestion d'exception, on induit qu'une erreur produite est due au manque de mémoire.
Voici la même gestion d'exception mieux construite :

```
try
  for i:=0 to 20 do
    chaine := chaine + chaine;
except
  On E:EOutOfMemory do
    Begin
       Showmessage ( 'La chaine de ma formulaire est trop longue. Il n'y a plus assez de mémoire.' );
    End;
  On E:Exception do
    Showmessage ('Erreur avec la chaine. Le résultat peut être tronqué.');
end;
```

L'instruction **« On E:Exception do »** suivie de sa gestion récupère toutes les erreurs d'exception produites. Si elle est seule, elle peut être éludée. C'est pourquoi on place une exception héritée au-dessus de l'exception ancêtre lorsqu'on utilise ce genre de gestion d'exception.

I.E.1 - Renvoyer les exceptions

Au départ, comme ici, les erreurs peuvent s'adresser à soi. Mais à force d'utiliser son logiciel, on redéfinira les exceptions pour l'utilisateur.
Dans un code réutilisé, il peut être intéressant de renvoyer l'exception vers le programme afin que lui seul gère le message à donner à l'utilisateur.
Le mot clé **« raise »** permet de faire cela.

```
type EStringTooLarge : EOutOfMemory;
begin
  if length ( chaine ) > 1000 Then raise (  EStringTooLarge );
  for i:=0 to 20 do
    chaine := chaine + chaine;
end;
```

Voici la même gestion d'erreur que précédemment mais dans un code source réutilisé. On anticipe toujours sur le développeur en réutilisant au mieux l'héritage des erreurs. Il est possible d'utiliser les exceptions de LAZARUS existantes pour ce que l'on souhaite rediriger. C'est d'ailleurs recommandé. Il faut par ailleurs pratiquer la veille technologique afin de voir si son erreur n'a pas été ajoutée avec une nouvelle version de son kit de développement.

I.F - Bases de données

Les bases de données sont des espaces de stockage organisés permettant de sauvegarder les informations nombreuses de ou des utilisateurs d'un logiciel. Un logiciel créé grâce à une base de donnée pourra en général accepter un grand nombre d'informations voire d'utilisateurs.

I.F.1 - Le pouvoir du libre

Des bases de données libres permettent de sauver beaucoup d'informations utilisateur. Elles permettent aussi d'accepter un grand nombre d'utilisateurs. Elles sont utilisées sur les meilleurs serveurs Web.
Beaucoup de serveurs Web utilisent mySQL. Les logiciels complexes utilisent POSTGRESQL. FIREBIRD est utilisé en embarqué pour des logiciels prêts à fonctionner. SQLLite est aussi utilisé pour créer des logiciels simples.

I.G - Bases de données embarquées

SQL Lite permet de créer des logiciels possédant une base de données embarquées. Cette base de données embarquées sera indépendante de tout serveur de données centralisé. Cela permet de créer des logiciels possédant beaucoup de données sans avoir besoin d'être connecté à un réseau.
Il sera possible d'envoyer des données et de les réceptionner. Il faudra donc vérifier si le Système de Gestion de Données peut le faire. Il est possible de créer des composants de communication de données. Il en existe sur DELPHI.

II - L'objet

La programmation orientée objets permet dans LAZARUS de réaliser vos propres composants, votre savoir-faire. Elle permet de réaliser des logiciels complexes. Ce chapitre et les deux chapitres suivants sont complémentaires. Ils doivent être bien compris afin de surpasser ses capacités de programmeur.
Dans ce chapitre, nous allons définir ce qu'est l'objet. Cela va permettre de mieux comprendre de LAZARUS et donc de sublimer cet outil par l'automatisation RAD. LAZARUS est un outil de Développement Rapide d'Application, aboutissement de l'utilisation de la programmation orientée objets.
Si vous ne connaissez pas l'Objet, vos composants seront mal faits. Il existe des règles simples et des astuces permettant de réaliser vos composants.
Nous allons dans ce chapitre vous parler simplement de l'objet. Il faudra étoffer vos connaissances avec des livres sur l'analyse Objet, analyse proche de l'humain qui nécessite du travail et de la technique. Chaque compilateur possèdera des spécificités ou améliorations pour faciliter le travail du développeur. Nous vous les expliquerons.

II.A - Un Objet

Un objet est une entité interagissant avec son environnement. Par exemple, votre « clavier » d'ordinateur est un objet. Ce « clavier » possède des « touches » qui peuvent aussi être des objets « touche ». Vous voyez qu'il est possible de définir un objet « Clavier » ou un objet « Touches ». Préférez l'unicité en mettant en tableau votre objet « Touche » au sein de votre objet « Clavier ». Vous créez un attribut au sein de votre objet « Touche ».

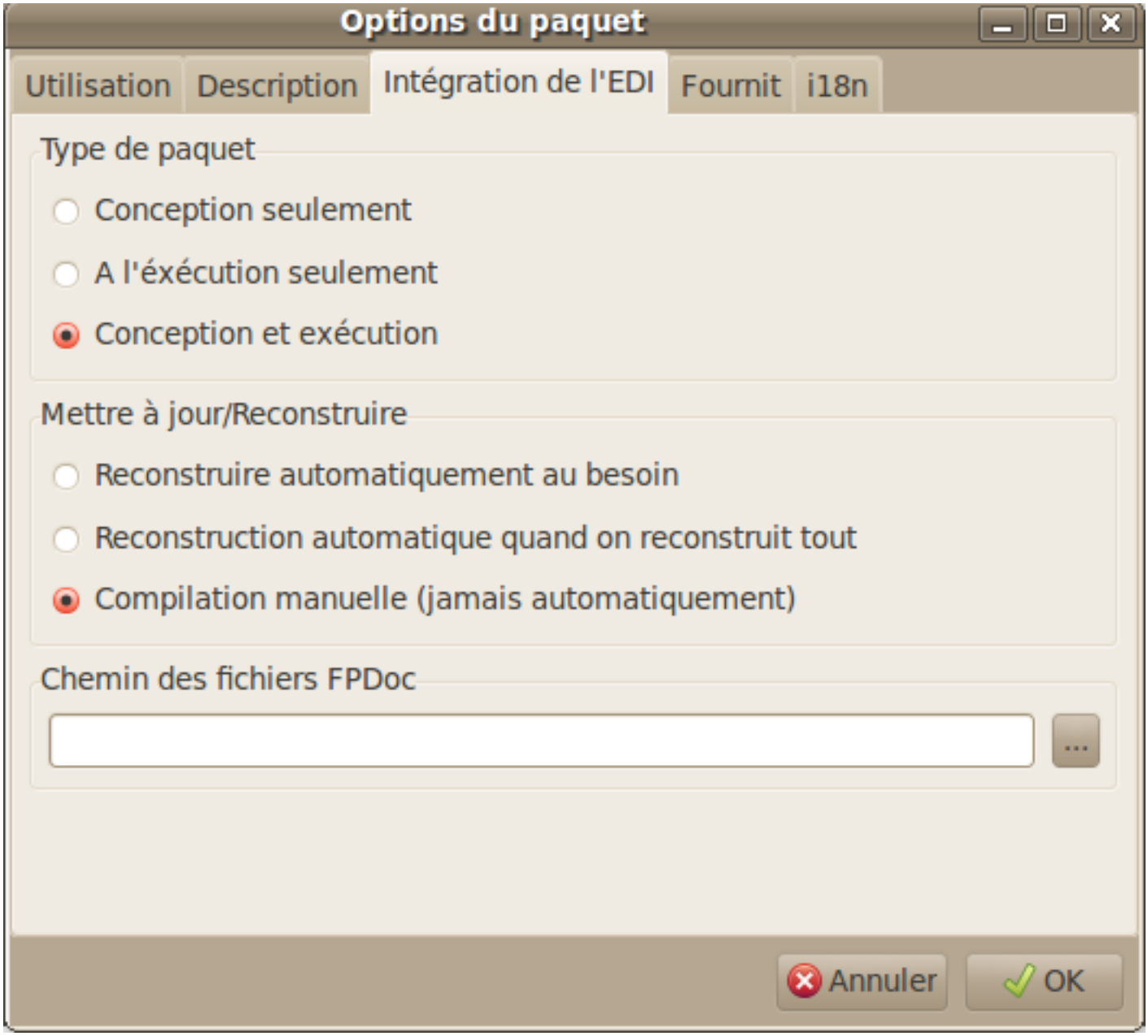
II.B - Une classe

Avec la notion de programmation par objets, une méthode de programmation proche de l'humain, il convient d'énoncer la notion de classe. Lors du tout premier exemple, on avait déclaré une classe dans le premier chapitre. Cette notion de classe est présente dans le FREE PASCAL. Elle a été ajoutée au PASCAL standard.
Ce que l'on a pu nommer jusqu'à présent objet est, pour FREE PASCAL, une classe d'objet. Il s'agit donc d'un type FREE PASCAL. L'objet FREE PASCAL est une instance de classe, plus simplement un exemplaire d'une classe, son utilisation en mémoire.
On peut remarquer que FreePascal pour sa part définit une classe comme un "pointeur vers un objet (type « classe ») ou un enregistrement (type « record »)".

II.B.1 - Les attributs et propriétés

Deux objets « Clavier » peuvent être faits de la même manière avec une couleur différente. Ce seront toujours deux objets « Clavier » avec un attribut différent. L'attribut est une propriété d'un objet. Un attribut peut être une couleur, un nombre, du texte, ou bien un autre objet.
Seulement, en LAZARUS, les variables peuvent être enregistrées dans l'EDI comme des propriétés. Elles sont alors sauvées dans l'objet « TForm » visible ou le « TDataModule » invisible. Seulement, ces objets n'ont pas besoin de LAZARUS pour charger leurs propriétés à l'exécution.
Au début du livre, un formulaire possédait des propriétés permettant d'automatiser facilement le développement. Les propriétés de l'inspecteur d'objets sont du code source servant uniquement au développement. Elles vous ont permis de gagner beaucoup de temps dans la création de votre premier logiciel LAZARUS.
Les développeurs LAZARUS ont soigneusement séparé les unités de propriétés servant au développement du code source exécuté. Vous ne trouverez pas les objets lisant les propriétés dans l'inspecteur d'objets inclues dans votre exécutable. Lorsque vous concevrez votre premier composant, il faudra aussi séparer le code source de

développement du code source exécuté. Vous ne devrez jamais lier dans vos projets une unité de propriétés puisqu'elles ne servent qu'à LAZARUS. C'est pour cette raison que des unités spécifiques à l'enregistrement des composants sont à part des unités de composants.



II.B.2 - Les méthodes et événements

Le clavier agit. Il possède alors des méthodes influençant ses attributs ou d'autres objets. On pourrait créer une méthode d'appui sur une touche quelconque. La méthode est une procédure ou une méthode centralisée dans un objet. Une méthode modifie les attributs ou variables du programme.
Seulement, en LAZARUS il y a mieux que la méthode. L'événement permet de renseigner rapidement tout objet. Ainsi, on ne crée pas l'objet Clavier LAZARUS va facilement se propager dans le formulaire de votre projet, puis dans chaque objet du formulaire. Chaque objet possède alors une méthode captant le clavier, que vous pouvez renseigner en cliquant sur un des différents événements Clavier.
L'événement consiste à transformer une méthode en variable pour pouvoir, comme les propriétés, transférer facilement un événement à un autre objet.
L'Objet est, contrairement aux langages procéduraux, plus proche de l'homme que de la machine. Il permettra de réaliser des systèmes humains plus facilement. L'objet étant plus proche des représentations humaines, il

complexifiera au minimum le travail de la machine en l'organisant en objets. Un objet possède trois propriétés fondamentales :

- L'Héritage
- L'Encapsulation
- Le Polymorphisme

L'objet peut être représenté schématiquement grâce à une boîte à outils nommée UML. Il existe des logiciels libres permettant de créer des schémas objets comme STAR UML ou TOP CASED. L'idéal est de disposer d'un savoir-faire utilisant des schémas UML grâce aux fichiers XML du logiciel. LAZARUS permet, grâce à l'inspecteur d'objets, d'arriver rapidement à ce genre de librairies homogènes.

II.C - L'Héritage

Les formulaires LAZARUS utilisent l'objet. Vous voyez une définition d'un type Objet dans chacune des unités de votre formulaire. Vous avez donc utilisé de l'Objet sans le savoir.
Un type Objet se déclare de cette manière :

```
type
 TForm1 = class ( TForm )
 end;
```

On crée ici une nouvelle classe qui hérite de la class « TForm ». Vous pouvez appuyer sur « Ctrl » puis cliquer sur « TForm » pour aller sur la déclaration du type « TForm ». Vous ne voyez alors que des propriétés car « TForm » hérite lui aussi de « TCustomForm », etc.
Notre formulaire se crée donc grâce à l'objet « TForm ». Cet objet possède la particularité de gérer un fichier « lfm » contenant les objets visibles dans la conception de le formulaire.
Votre formulaire est compris par LAZARUS par le procédé de l'héritage. Il n'utilise que rarement toutes les possibilités du type formulaire surchargé.
Vous pourriez donc utiliser un autre type descendant de « TForm » pour votre formulaire. C'est possible. Cependant, tout composant LAZARUS est hérité du composant le plus proche de son résultat demandé. Il faut donc hériter votre formulaire du composant « TForm » ou « TCustomForm ». Vous pouvez voir que ces deux Objets ne diffèrent que par la visibilité de leurs propriétés car « TForm » hérite de « TCustomForm ».
Vous voyez des déclarations de variables dans l'objet « TForm ». Ces variables seront mises en mémoire par le constructeur du formulaire. Vous ne voyez en général pas de constructeur dans un formulaire descendant de « TForm ». En effet, LAZARUS met automatiquement en mémoire vos variables de composants déclarées grâce à chaque fichier « .lfm » créé. Ce fichier « .lfm », c'est votre formulaire visuellement modifiable grâce à la souris et à l'inspecteur d'objets.

II.C.1 - La surcharge

La surcharge consiste dans l'héritage à modifier le comportement d'une méthode surchargée. Par exemple, on peut hériter de l'objet « TForm » afin de surcharger la méthode « DoClose ». Ainsi on automatise la fermeture de ses formulaires dans un composant héritant de « TForm ».

II.C.1.a - Le mot clé « virtual »

Par défaut, toute méthode déclarée ne peut être surchargée. Une méthode est donc par défaut statique car il est impossible de la surcharger. En effet, une méthode statique ne peut qu'être éludée, pas surchargée.
Pour qu'une méthode puisse être modifiée par ses héritières, il faut ajouter le mot clé « virtual ». Vous trouvez ainsi ce code source dans la première déclaration de DoClose :

```
procedure DoClose(var CloseAction: TCloseAction); virtual;
```

II.C.1.b - Le mot clé « override »

Pour surcharger une méthode virtuelle, il faut ajouter cette source dans une classe descendante. Vous pouvez par exemple ajouter ce code dans tout formulaire :

```
procedure DoClose(var CloseAction: TCloseAction); override;
```

Cette déclaration dans le formulaire est identique à l'affectation de l'événement « OnClose » de votre formulaire hérité de « TForm ». Seulement, créer cette méthode permet de créer un composant personnalisé hérité de « TForm ».
Il suffit d'appuyer en même temps sur « Ctrl », puis « Shift », puis « C » pour créer le code source à automatiser. Ce code source est ainsi créé :

```
procedure TForm1.DoClose(var CloseAction: TCloseAction);
begin
  inherited DoClose(CloseAction);
end;
```

Ainsi, lorsque tout objet propriétaire de « Doclose » appelle cette méthode, on appelle d'abord la méthode « Doclose » au-dessus de toutes, qui peut éventuellement appeler les méthodes ascendantes « Doclose » par le mot clé « inherited ». Choisissez presque toujours d'utiliser le mot clé « inherited » lorsque vous surchargez votre composant. Vous pourriez créer un composant hérité de « TForm » avec vos options de fermeture de formulaires. Ainsi du code source ne serait plus dupliqué.

II.D - L'Encapsulation

Nous venons de voir par la surcharge le procédé de l'encapsulation. L'encapsulation permet de cacher des comportements d'un objet afin d'en créer un nouveau.
L'encapsulation existe déjà dans les langages procéduraux. Vous pouvez déjà réutiliser une fonction ou procédure en affectant le même nom à la nouvelle fonction ou procédure. Seulement, il faut alors utiliser le nom de l'unité pour distinguer les deux fonctions ou procédures. L'encapsulation en PASCAL non Objet est donc approximative car on peut se tromper facilement de fonction ou procédure identique. En effet, on évite de créer les mêmes noms de fonctions ou procédures car cela crée des conflits sans forcément que l'on s'en aperçoive.
Au travers des unités et autres librairies, l'encapsulation prend une dimension objet.
L'intérêt de l'encapsulation est subtil et humain. Si vous créez un descendant d'un bouton en surchargeant la méthode « Click » afin de créer un événement centralisé et que vous utilisez ce bouton dans votre formulaire, vous vous apercevez que vous ne pourrez jamais appeler la méthode « Click » de ses ancêtres. Autrement dit, l'encapsulation permet de modifier légèrement le comportement d'un objet afin d'en créer un nouveau répondant mieux à ce que l'on cherche.
Autrement dit, on s'aperçoit ici que la programmation objet demande plus de mémoire que la programmation procédurale. En effet, il est possible d'éluder le code source de la méthode encapsulée.
Cependant, l'objet permet de gagner du temps dans le développement grâce à l'automatisation des systèmes humains puis informatiques. L'objet permet aussi de maintenir plus facilement le code source créé. Avec LAZARUS, tout objet composant d'un formulaire est consultable par simple clic de souris grâce à l'inspecteur d'objet.
Une variable ne peut pas être être héritée mais peut être de nouveau déclarée. Cependant, l'ancienne variable prend alors la place réservée à son pointeur à l'exécution. Il est même possible que vous ne puissiez empêcher le constructeur de la variable de mettre en mémoire ses autres variables s'il s'agit d'une classe. Quand trop de variables fantômes sont présentes dans l'ancien composant hérité, il faut se demander s'il est utile de créer un nouveau composant héritant du composant précédent. En effet, les variables fantômes sont tout de même en mémoire.

II.D.1 - La déclaration « private »

Une méthode « private » ne peut pas être surchargée. Une variable privée n'est accessible dans aucun descendant. On ne peut accéder à une méthode ou une variable « private » que dans l'Objet la possédant.

Vous voyez dans les sources de LAZARUS que beaucoup de variables sont déclarées privées. Elles sont tout de même accessibles dans l' « Inspecteur d'Objets » sous un autre nom. Souvent, il suffit d'enlever le début de la variable pour la retrouver dans l'« Inspecteur d'Objets ». Vous avez donc compris que les déclarations privées sont réutilisées afin d'être visibles dans l'« Inspecteur d'Objets ».
La déclaration « private » permet de regrouper les variables, les « getters » et les « setters » des propriétés créées. Ainsi, le programmeur ne se trompe pas lorsqu'il réutilise la variable. Il utilise la propriété permettant d'initialiser le composant en affectant la variable.

II.D.2 - La déclaration « protected »

Une méthode « protected » ne peut être surchargée que dans l'unité de l'Objet et les Objets hérités. Surcharger un composant permet donc d'accéder aux méthodes ou variables protégées.
Le type « TForm1 » peut accéder aux méthodes et variables « protected » de « TForm » et de ses ascendants. Par contre, ce type ne peut pas accéder aux méthodes et variables protégées des composants qu'il utilise.
On met dans la zone « protected » les méthodes de fonctionnement du composant pouvant être surchargées afin d'être modifiées dans un héritage. Si on ne sait pas s'il faut mettre une méthode dans la zone « private » ou « protected », il est préférable de placer sa méthode dans la zone « protected ».

II.D.3 - La déclarations « public »

Une méthode « public » peut être appelée dans tout le logiciel qui l'utilise. Une variable et une méthode publiques sont toujours accessibles. Ainsi un constructeur est toujours dans la zone « public » de sa classe.
Si vous ne savez pas s'il faut mettre une méthode en « public » ou en « protected », cherchez si votre méthode est utilisée plutôt par le développeur utilisateur en « public » que par le développeur de composants. Le développeur de composants modifie les facultés de votre composant.

II.D.4 - La déclaration « published »

Une méthode « published » peut être appelée dans tout le logiciel qui l'utilise. Une variable et une méthode publiées sont toujours accessibles.
Vous voyez dans vos composants que les propriétés « published » sont visibles dans l'inspecteur d'Objet de LAZARUS. La déclaration « published » et les propriétés dénommées par « property » permettent d'améliorer le polymorphisme du PASCAL Objet. Ce procédé permet de manipuler les propriétés et méthodes « published » sans avoir à connaître leur propriétaire. Vous pouvez réaliser des procédés de lecture indépendants du type d'objet grâce à l'unité « PropEdits ».
Ainsi une méthode publiée peut être appelée indépendamment de la classe objet qui l'utilise. Une propriété publiée est sauvegardée dans le formulaire et accessible indépendamment de la classe objet qui l'utilise.

```
// récupère une propriété d'objet
// aComp_ComponentToSet : Composant cible
// as_Name      : Propriété cible
// a_ValueToSet         : Valeur à affecter
function fmet_getComponentMethodProperty ( const aComp_Component : TComponent ; const as_Name : String
 ) : TMethod ;
Begin
  if   assigned ( GetPropInfo ( aComp_Component, as_Name ))
  and  PropIsType      ( aComp_Component, as_Name , tkMethod)
  then Result := GetMethodProp    ( aComp_Component, as_Name );
End ;
```

Cette méthode permet de récupérer toute méthode publiée « as_Name » de tout composant. Il suffit ensuite de forcer le type de votre événement. Par exemple :

```
{$mode Delphi} // Directive de compilation permettant d'enlever les ^ de pointeurs

Var MonClick : TNotifyEvent ;
```

```
Begin
MonClick :=TNotifyEvent ( fmet_getComponentMethodProperty ( AComponent , 'OnClick' ));
if assigned ( MonClick ) Then
  MonClick (  Acomponent );
End;
```

Ce code source permet d'exécuter tout événement « OnClick » de tout composant s'il existe.
Dans l'inspecteur d'Objet, vous pouvez voir aussi des événements. Les événements sont un appel vers une méthode grâce à une variable de méthode. Le type de la variable méthode permettra d'affecter des paramètres à une méthode.
Nous avons déjà utilisé les évènements dans un chapitre précédent.

II.D.5 - Les constructeurs et destructeurs

Les variables simples comme on l'a vu se libèrent automatiquement. FREE PASCAL ne libère pas automatiquement certaines variables, notamment les classes objets utilisées.
Il faut cependant savoir qu'un descendant de TComponent détruit automatiquement les objets TComponent qui lui ont été affectés. Donc si vous utilisez un descendant de TComponent dans votre classe, il suffit pour le détruire automatiquement d'affecter le propriétaire du composant comme étant votre classe d'objet.
Pour les autres classes d'objet, il est nécessaire d'utiliser les constructeurs et destructeurs pour d'abord les initialiser, enfin les libérer.
Voici un exemple de constructeur et de destructeur :

```
Interface
uses Classes ;

  TReadText = class(TComponent)
  MonFichierTexte : TStringlist;
  public
    constructor Create(TheOwner : TComponent);
    destructor Destroy;
  end ;

implementation

constructor TReadText.Create(TheOwner : TComponent);
begin
  Inherited Create(TheOwner);
  MonFichierTexte := nil;
end ;
destructor TReadText.Destroy;
begin
  Inherited Create(TheOwner);
  MonFichierTexte.Free;
end;
```

Tout objet défini dans une classe doit être initialisé à « nil ». Ainsi la méthode statique « Free » de la classe « TObject » vérifie si la variable est à nil. Puis, s'il n'est pas à « nil », elle appelle le destructeur de l'objet de type **TStringList**.

II.E - Le Polymorphisme

Le terme polymorphisme est certainement celui que l'on appréhende le plus. Pour le comprendre analysons la sémantique du mot : « poly » signifie plusieurs et « morphisme » signife forme. Le polymorphisme traite de la capacité de l'objet à posséder plusieurs formes.
Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, un objet hérite des champs et méthodes de ses ancêtres. On peut redéfinir une méthode afin de la réécrire ou de la compléter.
Le concept de polymorphisme permet de choisir en fonction des besoins quelle méthode ancêtre appeler, et ce pendant l'exécution. Le comportement d'un objet polymorphe devient donc modifiable à volonté.
Le polymorphisme est donc la capacité du système à choisir dynamiquement la méthode de l'objet en cours. Ainsi, si l'on considère un objet Véhicule et ses descendants Bateau, Avion, Voiture. Ces objets possèdent tous une méthode

Avancer, le système appelle la fonction Avancer spécifique suivant que le véhicule est un Bateau, un Avion ou bien une Voiture.

Le concept de polymorphisme en orienté objet ne doit pas être confondu avec celui d'héritage multiple en objet pur. L'héritage multiple n'est pas supporté par le FREE PASCAL. Il permet à un objet d'hériter des champs et méthodes de plusieurs objets à la fois. Le polymorphisme permet de modifier le comportement d'un objet et celui de ses descendants au cours de l'exécution.

II.E.1 - Le type « interface »

Le type « interface » est le polymorphisme en FREE PASCAL en son essence.
Il est impossible d'hériter une classe de deux classes voire plus. Il faut utiliser le type « interface » pour remédier à ce problème. Une classe FREE PASCAL hérite d'une classe voire d'un ensemble d'interfaces.
Le type « interface » permet de créer des objets polymorphes. Il peut s'ajouter à un héritage. On crée le type interface comme si on créait un objet classe. Cependant, il n'est possible que d'y ajouter des méthodes abstraites. Nous parlons du type « interface » dans le chapitre suivant.

II.E.2 - Polymorphe avec les propriétés publiées

Nous avons vu précédemment qu'il est possible d'appeler n'importe quel événement « Onclick » de n'importe quel composant. Le type classe ou Objet « TForm » c'est un composant particulier. Le composant et Objet « TForm » permet de créer d'autres Objets Composants à l'aide d'un fichier avec l'extension « .lfm ». Vous pouvez donc changer de composants dans vos formulaires en changeant le type de votre composant source vers votre type destination. Il faut changer le type de votre composant à la fois dans le fichier « .pas » de votre ficher puis dans le fichier « .lfm »
Tout d'abord, allez sur le formulaire à transformer. Ajoutez un composant « TLabel ». Redimensionnez-le.
Allez sur la déclaration classe du formulaire dans le fichier « .pas ».
Changez le type de son composant vers le nouveau type de composant sans vous tromper. Vous pouvez, par exemple, lui affecter le type « TButton ».
Allez sur le formulaire visuel et cliquez droit dessus. Choisissez « Afficher le source ». Retrouvez votre composant en faisant une recherche puis changez le type par le même type « TButton ».
Fermez et rouvrez le formulaire. Des propriétés n'existent plus. LAZARUS demande de les effacer.
Votre nouveau composant « TButton » a remplacé l'ancien avec les propriétés identiques de ce dernier. C'est la nomenclature LAZARUS qui a permis d'adapter les propriétés anciennes vers le nouveau composant. Autrement dit, il faut connaître les propriétés standards de tout objet LAZARUS avant de créer une nouvelle propriété LAZARUS. Une propriété LAZARUS est en anglais.
Les types Objet « interface » ne permettent pas de déclarer de véritables Objets mais ne font que centraliser la déclaration de méthodes. Il faudra n'utiliser qu'un seul Objet. Les unités de fonctions permettent de centraliser le code.
La déclaration « published » et les propriétés permettent d'accéder à d'autres composants sans avoir à connaître leur type « classe » ou « interface ». La déclaration « published » et les unités de fonctions résolvent partiellement le polymorphisme en permettant cependant une hiérarchisation de vos sources de code grâce aux paquets.

II.F - L'Abstraction

L'Abstraction ne sert en LAZARUS qu'à centraliser plusieurs objets en héritant d'un objet abstrait, donc pas utilisable. Cet objet abstrait possède en fait des méthodes dites « abstraites » ou « abstract » non implémentées dans l'objet abstrait. Voici à quoi l'abstraction ressemble :

```
procedure Nager; abstract;
```

Cette déclaration est la seule à ne pas demander d'implémentation. Elle crée une erreur malgré tout si son descendant n'implémente pas la méthode « Nager ».
Ainsi, l'objet abstrait qui nage n'a pas besoin de savoir comment il nage. Les différents descendants nagent en fonction du type classe créé. Une classe « Poisson » nage en oscillations horizontales. Une classe « Humain » nage

en brasse, en crawl, en papillon, etc. On crée donc dans la classe Humain des méthodes secondaires. Il reste à définir comment on choisit le type de nage en fonction de l'objet créé.

II.G - Une Instance d'Objet

Une instance d'objet en UML est un objet en mémoire en FREE PASCAL.
L'objet en UML est une classe d'objet en FREE PASCAL.
Le PASCAL Objet permet de créer des Instances d'Objets. Une Instance d'Objet, c'est un objet mis dans la mémoire de l'ordinateur à l'exécution. Une Instance d'Objet peut être dupliquée et sera personnalisée grâce à ses variables. Si votre Classe d'Objet et ses Classes parentes ne contiennent aucune variable, il faut se poser la question de leur utilité. Puis-je gagner du temps en créant une unité de fonctions à la place ?
Vous créerez votre Objet, qui est instancié une ou n fois en exécution. Par exemple, les formulaires prennent beaucoup de place en mémoire donc on les instancie en général une fois.
Les variables simples de votre objet sont dupliquées et initialisées à zéro pour chaque Instance créée.

II.H - Les propriétés

FREE PASCAL possède une syntaxe permettant de créer des composants qui sont des Objets avec des propriétés. Les propriétés permettent, si elles sont correctement créées, de mieux gérer le polymorphisme. Les propriétés publiées permettent la rapidité en étant présentes dans l'inspecteur d'objets.
Il ne faut pas être créatif pour nommer ses propriétés. Il faut reprendre ce qui existe déjà au sein de LAZARUS afin de mieux gérer le polymorphisme. Une propriété doit être nommée du même nom que les propriétés de même nature. Il faut donc connaître le nom des propriétés des composants de même nature afin de créer des propriétés identiques. On essaie d'être compatible avec certains composants de même nature en définissant des propriétés identiques.

III - L'UML pour programmer en Objets

Des outils UML libres existent pour automatiser la création de vos logiciels orientés objets.
UML en anglais signifie Unified Modeling Language, pour Langage de Modélisation Unifié. Ainsi tout outil UML peut être compatible avec d'autres outils UML. Il faut cependant vérifier si c'est bien le cas.

III.A - STAR UML

STAR UML est un outil libre donc partagé en licence GNU/GPL.
Il a été fait sous DELPHI. Vous pouvez donc tenter de le traduire vers LAZARUS après avoir vérifié que quelqu'un ne le fait pas déjà. Dans ce cas, il faut participer au projet de traduction qui peut être intégré au projet STAR UML.

IV - Créer son savoir-faire

Pour créer un savoir-faire, il faut respecter un seul principe : Éviter le copier-coller.
On crée donc avec cette technique un savoir-faire :

- Au début, il n'y a que des unités de fonctions
- Puis on utilise et surcharge des composants
- On crée des paquets de composants
- On ouvre alors sa librairie aux autres API
- On automatise les paquets en une librairie
- La librairie nécessite peu de code ou aucun

On crée des unités de fonctions, puis des composants regroupés en paquets LAZARUS, puis une voire plusieurs librairies. Ces librairies seront indépendantes de LAZARUS en étant entièrement automatisées grâce à des fichiers. L'aboutissement sera la prise en compte de ce qui est demandé par les clients en créant des fichiers automatisant la librairie uniquement. Ces fichiers sont dédiés à ce qui est demandé. Ce sont les fichiers métiers. Ces fichiers peuvent

être automatiquement créés par l'analyse. Il n'y a alors plus d'inadéquation entre l'analyse et le logiciel. En effet, il est difficile d'avoir une analyse identique au logiciel demandé sans service qualité ou sans automatisation.

IV.A - Créer des unités de fonctions

Vous pourrez vous aussi créer votre savoir-faire en créant d'abord des projets. En créant ces projets, nous vous apprendrons à centraliser ce qui aurait été redondant. Il faut éviter le copier-coller. Vous créez des unités de fonctions en centralisant un code redondant.
Une unité de fonctions, c'est un regroupement de fonctions autour d'un même thème. Si la fonction que vous ajoutez n'est pas immédiatement associée à ce thème, n'hésitez pas à en trouver un nouveau. On crée alors une nouvelle unité.

IV.B - Les composants

LAZARUS sans les composants ne serait pas utile. LAZARUS permet de créer rapidement des logiciels grâce aux composants. La programmation par composants est rapide sur LAZARUS grâce à l'inspecteur d'objets. Vous pouvez décupler votre vitesse de création avec LAZARUS et ses composants visuels.
Il est recommandé d'améliorer ou de créer des composants qui sont facilement mis en place. Ainsi, votre savoir-faire ou vos composants sont centralisés dans des paquets de composants puis dans une voire plusieurs librairies utilisant vos composants.
Un paquet est un regroupement de composants. Les paquets sont installés rapidement sur LAZARUS.
Les composants sont visibles dans LAZARUS. Les composants visuels sont directement modifiables dans LAZARUS. Vous disposez d'un inspecteur d'objet permettant de voir en temps réel la visibilité de votre composant.
Vos unités de fonctions pourront ensuite améliorer des composants par le procédé de l'héritage ou par la participation Open Source. Vous créerez votre premier paquet en apprenant l'objet. Vos unités de fonctions seront utilisées et améliorées. Vous participerez à un projet Open Source après avoir vérifié la licence Open Source. Si la participation au projet est refusée, il faut se poser des question de l'utilité de cette participation. L'entreprise qui refuse la participation peut avoir d'autres objectifs que les vôtres.
LAZARUS, c'est une interface facilitant la programmation. Un composant LAZARUS est mis en place rapidement grâce à l'inspecteur d'objets. Cet inspecteur permet d'affecter les propriétés de vos composants comme si vous étiez un simple utilisateur. Les propriétés permettant de développer sont un supplément sur le formulaire ouvert. Elles sont et doivent donc être situés en dehors du code source d'exécution.
Votre composant peut être amélioré grâce à la communauté. Cela ne vous empêche pas, par contre, de toujours être compétitif en recherchant d'autres projets à créer pour hériter vos ou d'autres composants.

IV.C - Créer un composant

Nous allons changer la couleur d'un bouton et lui affecter une image. A chaque bouton créé sera affectée une seule image. Ainsi, l'exécutable sera plus léger. Ce bouton permettra de fermer un formulaire.
Le composant qui va être créé va donc répondre à une demande technique : La taille de l'exécutable et la centralisation des sources. La centralisation des sources va cependant permettre de répondre à l'automatisation de son savoir-faire. Cela va donc permettre de répondre plus rapidement à la demande du client.
Pour créer votre paquet personnalisé, faites « Fichier » puis « Nouveau » puis « Paquet ».

« Ajouter » un « Nouveau fichier » « Unité », puis « Ajouter » la condition « LCL ». Une condition est un paquet utilisé. Le paquet LCL contient tous les composants standards.
Sauvegardez votre unité sous le nom « u_buttons_appli ».
Créez dans le répertoire du paquet un fichier vide « u_buttons_appli.lrs ».
Enregistrez votre paquet en cliquant sur « Enregistrer » dans le projet paquet. Affectez-lui le nom « LazBoutonsPersonnalises ».
Ajoutez ses deux types dans la partie interface de votre unité :

```
type
   IMyButton = interface
   ['{620FE27F-98C1-4A6D-E54F-FE57A06207D5}']
   End ;
```

On déclare un type interface permettant d'indiquer que nos boutons supportent tous le type « IMyButton ». La chaine hexadécimale permet d'utiliser la méthode « support » dans toute classe afin de reconnaître le type « IMyButton ». Ainsi, nous retrouvons plus facilement nos boutons. Nous utilisons le polymorphisme pour reconnaître nos propres boutons.
Voici le code source permettant de reconnaître nos boutons :

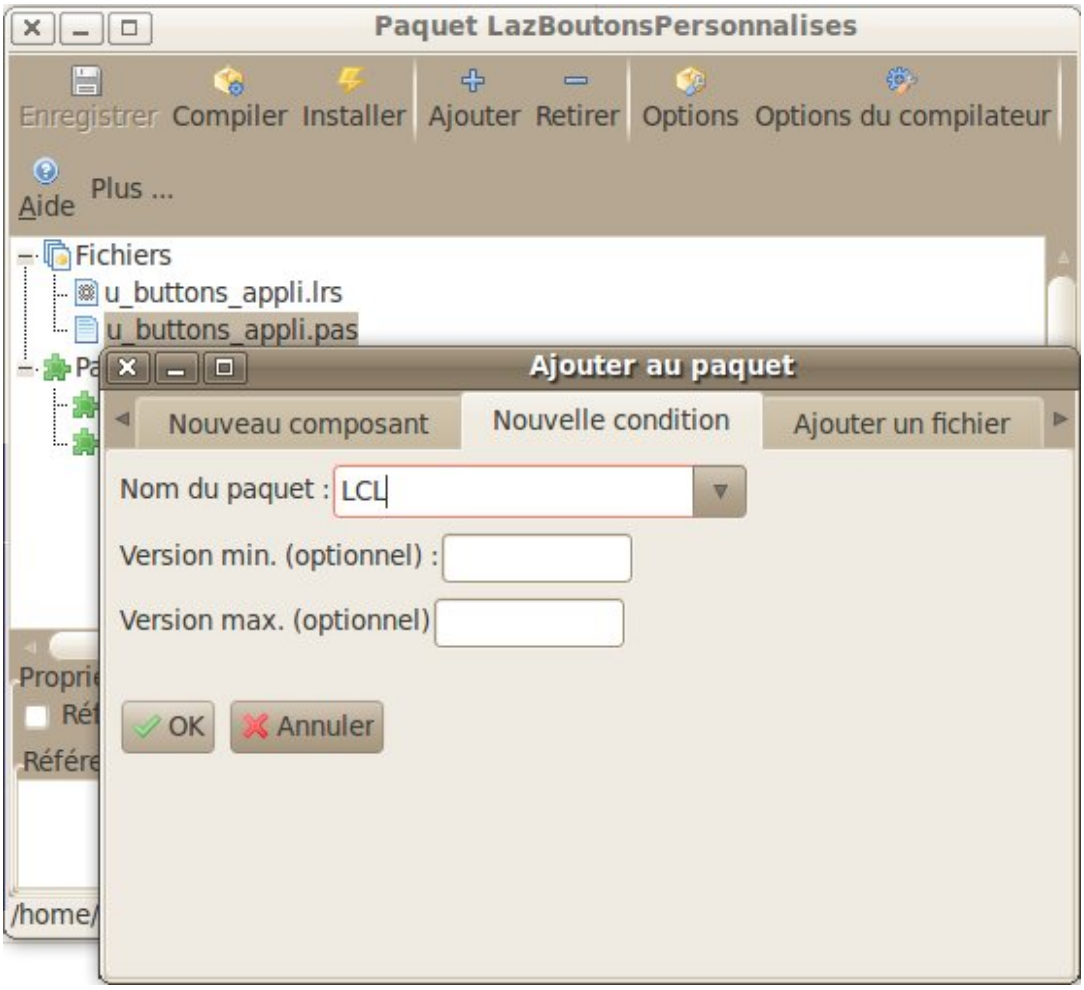
```
If VariableDeTypeClasse.Support ( IMyButton ) Then
  Begin
    ShowMessage ( 'C'est mon bouton !' );
  End;
```

Il faut en FREE PASCAL définir au maximum ce que l'on fait. Les définitions permettent d'utiliser l'objet afin d'améliorer l'utilisation de nos composants.
Juste après la définition de l'interface « IMyButton », placez ce code :

```
TMyClose = class ( TBitBtn,IMyButton )
```

```
    private
    public
     constructor Create(TheOwner: TComponent); override;
     procedure Click; override;
    published
     property Glyph stored False;
   End;
```

Le constructeur Create et la méthode Click sont présentes dans le composant TBitButton par l'héritage. Nous les surchargeons par le mot clé « override » suivi d'un « ; ».
Grâce au polymorphisme LAZARUS, vous pouvez changer l'ancêtre « TBitBtn » par un ancêtre bouton possédant un « Glyph ». D'autres boutons avec « Glyph » existent en dehors du projet LAZARUS.
Tous les composants LAZARUS possèdent le constructeur Create avec, comme paramètre, le propriétaire du composant. Notre composant sera donc automatiquement détruit lorsque son parent se détruira.
Aussi tous les composants descendants de la classe TLCLComponent possèdent la méthode « Click » sans aucun paramètre. Une méthode qui peut être surchargée possède le mot clé « virtual ». Le composant ancêtre définissant la méthode utilise donc le mot clé « virtual » sur la méthode créée. Vous pouvez donc facilement vérifier si c'est bien TLCLComponent qui définit « Click » en maintenant « Ctrl » enfoncée puis en cliquant sur chaque type ascendant utilisé.
Pour chercher l'unité du bouton « TBitBtn », cherchez dans tous les répertoires du code source de LAZARUS la bonne unité. Sinon, cette unité s'appelle « Buttons ». Cette unité est incluse dans le paquet LCL. Il faut donc « Ajouter » la « Condition » « LCL » dans le paquet.



Si votre code source est bon, vous pouvez appuyez sur « Ctrl »+ « Shift » + « C ». Cela va créer les deux méthodes. Vous pouvez compiler le paquet pour trouver des erreurs.
Maintenant, nous allons remplir les deux méthodes surchargées dans la partie « implémentation » :

```
uses Forms;

 // Créer des constantes permet d'éviter de se tromper  pour leur réutilisation
const
   CST_FWCLOSE='TMyClose';  // Nom du fichier XPM

{ TMyClose }

procedure TMyClose.Click;
begin
  if not assigned ( OnClick )
  and ( Owner is TCustomForm ) then
    Begin
      ( Owner as TCustomForm ).Close; // Fermeture du formulaire
      Exit;
    End;
  inherited;
end;
constructor TMyClose.Create(TheOwner: TComponent);
begin
  inherited Create ( TheOwner );
  if Glyph.Empty then
    Begin
      Glyph.LoadFromLazarusResource ( CST_FWCLOSE); // Charge le fichier XPM
    End;
end;
```

Voilà l'essentiel de notre composant « TMyClose ». La procédure surchargée « Click » ferme le formulaire s'il n'y a pas d'événement de Click sur le Bouton dans le formulaire. Il suffit que ce code source soit exécuté une seule fois dans l'application pour que ce code source ait remplit un objectif d'automatisation.
On passe la méthode « Click » héritée par la méthode « Exit ». On a vérifié, avant d'éluder la méthode « Click » de l'ancêtre, qu'il n'y avait que la gestion de l'évènement « OnClick » que l'on évitait. Nous n'éludons effectivement pas l'événement « OnClick » en vérifiant s'il existe. Ainsi, on évite de revérifier l'évènement « OnClick ».
Le code spécifique est automatisé avec le maximum de composants possédant différents objectifs. Il faut cependant se rendre compte que, bien que le composant permette de centraliser, il augmente aussi la taille de l'exécutable.
Si la fonction de fermeture était sur chaque événement de chaque bouton, cela alourdirait l'exécutable. Aussi, si les conditions techniques de fermeture du formulaire changeaient, il faudrait changer le code source de chaque bouton.

IV.C.1 - Choix de l'image

Le code source du constructeur va créer une erreur à l'exécution car il n'y a pas de fichier image.
Le constructeur de notre bouton charge l'image du bouton grâce au fichier « lrs ».
Choisissez votre image de fermeture sur un site web contenant des images libres de droits avec une licence « Creative Common ». Vérifiez la licence.
Le fichier image doit être un fichier « XPM » avec comme nom le type de la classe de notre bouton.
Convertissez avec « GIMP » votre image au format « XPM ». Pour faire cela, sauvegardez l'image avec « GIMP » en remplaçant l'ancienne extension de fichier par « XPM » sans enlever le point. L'extension, ce sont les dernières lettres après le dernier point du fichier.
Allez dans votre dossier contenant « lazarus ». Allez dans le répertoire « tools ». Compilez le projet « lazres ».
Ouvrez un terminal ou allez dans votre accessoire « Ligne de commande ».
Copiez le lien vers l'exécutable « lazres ». Copiez le lien vers le fichier ressources « lrs » qui a été créé. Puis copiez le lien vers le fichier XPM. Vous pouvez éventuellement ajouter d'autres images si vous voulez créer d'autres boutons.
Maintenant, il faut ajouter le fichier ressource à l'unité. Allez tout à la fin de l'unité de votre composant. Insérez ce code source avant le « end. » final :

```
initialization
  {$I u_buttons_appli.lrs}
end.
```

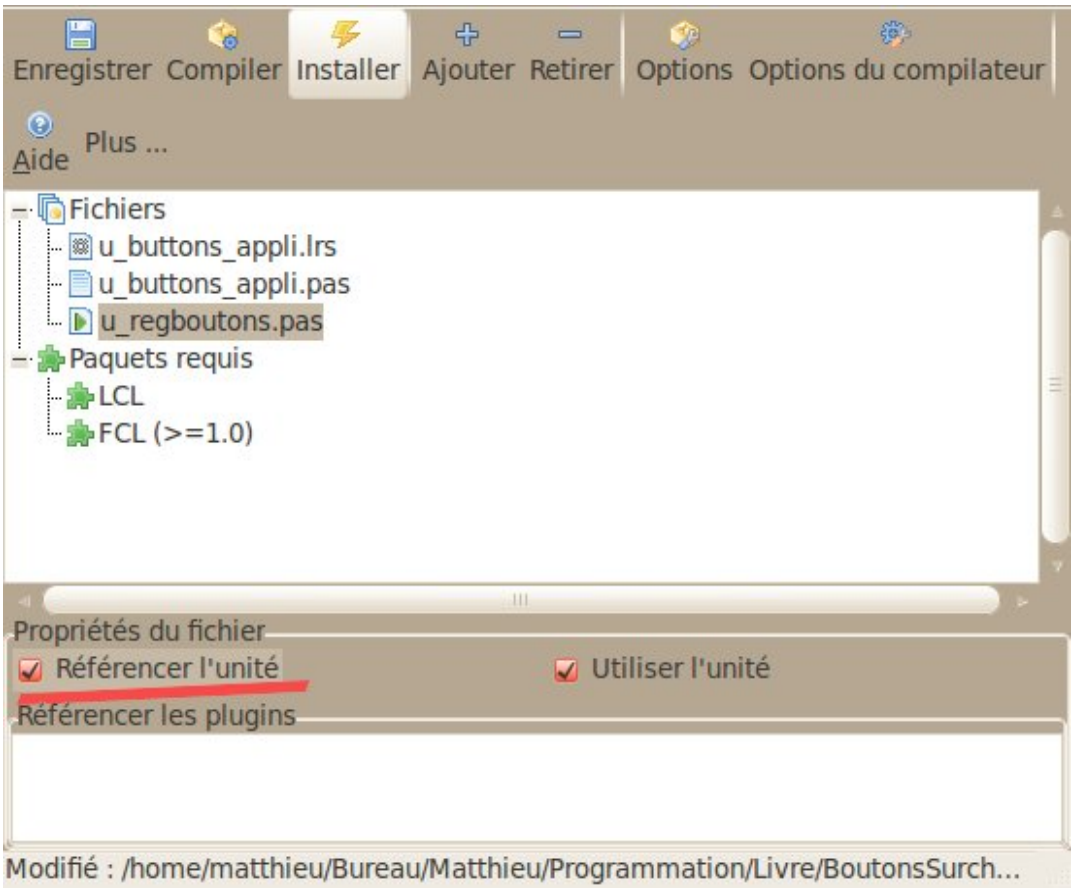
La directive **{$I}** ajoute le fichier ressource au code source.

IV.D - Enregistrement du composant

A l'enregistrement du composant et en fonction de la surcharge effectuée, celui-ci voit certaines de ses propriétés reconnues par LAZARUS afin d'éditer rapidement le composant. Il est possible d'utiliser des propriétés voire de les surcharger dans son unité d'enregistrement de composants.
Un composant est un objet descendant de l'objet TComponent. Tout composant LAZARUS descendant de TComponent peut être accessible dans la palette de composants. Pour qu'un composant soit enregistré dans une palette, voici le code source à affecter à une unité d'enregistrement du composant.
L'unité d'enregistrement ne doit jamais être utilisée dans votre programme. En effet, le code source de développement est inutile pour le programme exécutable. Créez cette nouvelle unité dans votre paquet :

```
Unit u_regboutons;

interface

uses Classes ;

procedure Register ;

implementation

uses u_buttons_appli;

procedure Register;
begin
  RegisterComponents('MesBoutons', [TMyClose]);
end ;

end.
```

La procédure « Register » doit être exécutée par le paquet qui installe le composant. Il faut donc vérifier pour cette unité si la case à cocher « Référencer l'unité » est cochée.



IV.D.1 - Modifier ou Surcharger ?

Si votre composant ajoute un objectif au composant initial, il est préférable de créer une autre unité surchargeant le composant permettant de ne pas avoir de code inutile pour le premier objectif du composant.
La modification d'un composant se fait en deux étapes. Pour pouvoir modifier un composant, il faut avoir l'accord de l'auteur. En général, il faut avoir accès au gestionnaire de version permettant de le modifier.
Il faut, sinon, que le composant soit abandonné et que sa licence permette de le modifier. S'il n'y a aucune licence, il faut contacter l'auteur.
Sinon vous ne faites qu'ajouter des options mineures. Seulement, sans l'accord de participation de l'auteur, vous ne pourrez pas correctement mettre à jour le composant et serez seul responsable de ses évolutions. Les licences libres possèdent plus de facilités quant aux modifications. Il suffit souvent de contacter l'auteur, qui sera content de vous aider à améliorer son savoir-faire.

IV.D.2 - Surcharger un composant

N'hésitez pas à surcharger tout composant, surtout si vous disposez des sources qui permettent de trouver plus facilement les contournements pour les erreurs d'héritage. Lorsque vous surchargez un composant, vous avez accès aux méthodes protégées. Ces méthodes sont inaccessibles sans la surcharge.
Il faut quelquefois être astucieux afin de surcharger correctement certaines méthodes de certains composants. Vous pouvez indiquer à l'auteur du composant les modifications nécessaires à un héritage facilement effectué. Il faut donc éviter d'utiliser des contournements car ils pourront provoquer de nouvelles erreurs à la mise à jour.

IV.E - Créer une librairie homogène

Un paquet de composants est mis en place rapidement. En voulant être encore plus rapide, vos paquets deviennent une librairie complète paramétrable au minimum voulu. Vous sublimerez alors votre capacité d'adaptation en ne créant que l'utile.

Ce qui sera personnalisé sera paramétrable. Le reste se configurera automatiquement. Une librairie fournit des outils et des procédés pour créer son logiciel sans perdre de temps. Il faudra améliorer l'IDE LAZARUS ou bien éviter d'utiliser l'IDE avec des fichiers de configuration. Ces fichiers pourront être idéalement des fichiers analytiques UML ou MERISE. Ces fichiers vous permettront d'être indépendant de LAZARUS en utilisant les mêmes savoirs-faire sur d'autres outils de programmation.

IV.F - Le libre et l'entreprise

LAZARUS contredit les professeurs ou institutions indiquant que les composants passent plus de temps à être maintenus qu'à être utilisés. Un composant LAZARUS fait gagner beaucoup de temps grâce à l'inspecteur d'objets et au code source de développement.
Votre composant, s'il est bien documenté, peut devenir un atout fiable faisant connaître votre société. Il peut aussi être sauvé de l'abandon grâce à une communauté toujours plus avare de composants libres. Un composant qui devient libre veut soit être populaire soit se faire connaître pour survivre.
Créer un composant libre permet de certes faire connaître votre composant, mais aussi de trouver le moyen de savoir si on reste le meilleur dans son domaine en créant une communauté sondant les utilisateurs et contributeurs. N'espérez pas des contributeurs qu'ils collaborent sans être rémunérés. Mais attendez d'eux une expertise sur votre savoir, des aides pour vous promouvoir. Ces aides ne sont peut-être pas celles que vous souhaitez. Elles permettent donc de trouver de nouveaux marchés.

V - De PASCAL vers FREE PASCAL

V.A - Introduction

FREE PASCAL permet de créer des savoirs-faire multi-plateformes à partir de savoirs-faire WINDOWS comme DELPHI ou TURBO PASCAL.
A partir de ces savoirs-faire ou logiciels, on peut créer des librairies FREE PASCAL. Mais cela demande l'adaptation d'un partie des sources.

V.B - De TURBO PASCAL vers FREE PASCAL

Pour transférer un logiciel TURBO PASCAL textuel, il faut juste réutiliser les bonnes unités. Elles se trouvent nécessairement dans LAZARUS ou FREE PASCAL.
Pour transférer un savoir-faire TURBO PASCAL graphique, il faut soit :

- Insérer la partie graphique dans une fenêtre et adapter la taille graphique à la fenêtre
- Recréer la partie graphique avec les composants visuels

V.B.1 - Choisir un gestionnaire de données

Si on souhaite beaucoup d'utilisateurs ou de données, il faudra créer un lien vers un Système de Gestion de Base de Données comme FIREBIRD, SQL LITE, MY SQL, voire ORACLE. FIREBIRD et SQL LITE sont des S.G.B.D. mi-lourds suffisants pour tout logiciel installé facilement. Ils s'utilisent en embarqué.
MY SQL permet, lui, de faire évoluer votre serveur de données en serveurs de données répartis. ORACLE, lui, permet de créer un seul serveur de données très lourd.

V.C - De DELPHI vers LAZARUS

Lorsque son logiciel est créé en DELPHI, le transfert consiste à utiliser des composants LAZARUS à la place des composants DELPHI. Il est possible que son logiciel reste compatible DELPHI grâce aux directives de compilation.
Vous pouvez trouver les composants qui vous manquent grâce à une recherche sur un site web de composants comme lazarus-components.org.

V.D - Traduction de composants

Pour traduire un composant DELPHI vers LAZARUS, il est préférable que ce composant reste compatible DELPHI grâce aux directives de compilation. Cela permet de participer au projet existant et de centraliser les sources.
Il faut remplacer le code graphique et les liens vers les librairies WINDOWS par du code source LAZARUS. LAZARUS possède une large bibliothèque de code source.
Le code graphique LAZARUS utilise les différentes librairies libres de chaque plateforme. Ces différentes librairies sont homogénéisées en des librairies génériques.
Les unités à ne pas utiliser sont les unités spécifiques à une seule plateforme comme les unités :

- win pour WINDOWS
- gtk pour LINUX
- carbon pour MAC OS
- unix pour UNIX

Si vous êtes obligé d'utiliser une de ces unités, sachez que vous aurez peut-être une unité générique dans la prochaine version de LAZARUS. Votre composant sera compatible avec la seule plateforme de l'unité non générique utilisée.
Si vous ne trouvez pas ce qu'il vous faut, recherchez dans les sources LAZARUS ou FREE PASCAL. Contactez sinon un développeur LAZARUS. Il vous indiquera ce que vous pouvez faire.