

Matthieu GIROUX

LAZARUS FREE PASCAL



Développement rapide

Réduire le temps de création des logiciels
Créer des logiciels interactifs multiplate-forme
Créer et maintenir son savoir-faire
Anticiper l'avenir

Licences CREATIVE COMMON by SA ou by NC ND

Niveau : Néophyte



ISBN 978295312516

Éditions LIBERLOG
Éditeur n° 978-2-9531251

Droits d'auteur RENNES 2009
Dépôt Légal RENNES 2010

Sommaire

A) A lire.....	4
B) Biographie.....	6
C) LAZARUS FREE PASCAL.....	7
D) Programmer facilement.....	17
E) Le langage PASCAL.....	39
F) Programmation procédurale avancée.....	58
G) Calculs et types complexes.....	76
H) Les boucles.....	88
I) Créer ses propres types.....	96
J) Ma première application.....	114
K) L'Objet.....	129
L) Créer son savoir-faire.....	154
M) De PASCAL vers FREE PASCAL.....	179
N) L'Objet et les jeux.....	186
O) La persistance d'un Logiciel.....	205
P) Logiciel centralisé.....	224
Q) Programmation avec FIREBIRD.....	237
R) Le Web : EXTPASCAL et SQLITE.....	253
S) Collaborer.....	282
T) Cross-compilation LAZARUS.....	289
U) Création du livre.....	300
V) Glossaire.....	307

A) A LIRE

Les mots que vous ne comprenez pas, avec leur première lettre en majuscule, peuvent être trouvés dans le glossaire à la fin du livre. Les exemples peuvent se télécharger sur le site www.liberlog.fr.

Les mots en gras indiquent des chapitres.

Les mots en majuscules indiquent des groupements, des langages, des outils, des entités, des marques.

Les mots entre guillemets indiquent des mots de LAZARUS, des mots à chercher sur le Web, ou des mots de votre environnement.

Les mots entre crochets indiquent des mots à remplacer.

1) **OBJECTIFS DU LIVRE**

L'objectif du livre est d'apprendre facilement LAZARUS afin de créer rapidement un Logiciel. On met en avant la création de Composants, étape indispensable dans l'automatisation et l'optimisation. On donne la démarche pour créer très rapidement un Logiciel. Cette démarche consiste à éviter les copiés-collés.

Je mets une nouvelle méthode en avant dans ce livre. Le Développement Très Rapide d'Applications est l'aboutissement de tout savoir-faire automatisant la création de Logiciel. On prépare la création pour sauter une étape : La création du Logiciel à partir de l'analyse. C'est l'analyse qui crée le Logiciel grâce à votre moteur.

2) LICENCE

Ce livre a été écrit par Matthieu GIROUX et Jean-Michel BERNABOTTO pour le langage PASCAL. Ce livre est sous deux licences : CREATIVE COMMON BY SA et CREATIVE COMMON BY NC-ND. Les licences sont indiquées au début des chapitres.

Vous pouvez utiliser ce livre gratuitement en citant l'auteur. Vous devez être bénévole pour utiliser les chapitres en licence NC-ND.

B) BIOGRAPHIE

Matthieu GIROUX aime l'écriture, cette recherche de la vérité en utilisant des moments de réflexion.

Il s'intéresse au Développement Rapide d'Applications afin de trouver de meilleures techniques de programmation. Sauter une étape la création du Logiciel est possible grâce à l'automatisation, maître mot de l'informatique dans l'entreprise.

LAZARUS n'est donc qu'une étape dans la réalisation d'un savoir-faire ou Framework. Il faudra ensuite automatiser à partir d'autres frameworks afin de mieux faire connaître le sien. Recherchez le partage pour vous enrichir et enrichir les autres.

1) *DU MÊME AUTEUR*

- L'astucieux LINUX d'après www.aides-informatique.com
- Comment écrire des histoires d'après www.comment-ecrire.fr
- Nos Nouvelles Nos Vies
- Poèmes et Sketchs – De 2003 à 2008

Visibles sur GOOGLE BOOKS.

Disponibles sur www.comment-ecrire.fr, www.lazarus-components.org.
Site d'informations : www.france-analyse.com.

C) LAZARUS FREE PASCAL

CREATIVE COMMON BY SA

1) ***POURQUOI CHOISIR LAZARUS ?***

Si vous voulez apprendre la programmation d'interfaces homme-machine LAZARUS est fait pour vous. LAZARUS c'est un EDI, un Environnement de Développement Intégré, disponible sur un maximum de machines, compatible DELPHI. DELPHI est beaucoup enseigné dans les écoles françaises. Nous souhaitons le même destin à LAZARUS qui ne possède pas de licence payante. LAZARUS c'est une utilisation des bibliothèques Libres les plus fiables avec une interface facile à utiliser. LAZARUS est un Logiciel Libre.

Vous pouvez adapter des Composants DELPHI pour que vos Logiciels deviennent multiplate-forme. Ils seront donc utilisables sur beaucoup de machines. DELPHI a été créé en 1995. Le projet LAZARUS a commencé en 1999. Il est actuellement préférable d'utiliser LAZARUS plutôt que KYLIX, un DELPHI sous LINUX créé en 2001. En effet KYLIX a été abandonné dès sa troisième version.

Un logiciel Client/Serveur c'est en quelque sorte un logiciel qui n'a pas besoin de navigateur web pour fonctionner. Si vous voulez créer ce genre de Logiciel, vous pouvez compter sur des bibliothèques de composants visuels, permettant de réaliser vos interfaces exécutables, sur vos plates-formes préférées.

Le multiplate-forme consiste à diffuser largement son Logiciel sur un maximum de machines. On utilise LAZARUS pour créer des Logiciels Client/Serveur multiplate-forme.

Si vous voulez créer votre jeu multiplate-forme, LAZARUS possède des bibliothèques permettant de créer des jeux sur 2 Dimensions ou sur 3 Dimensions. On utilise LAZARUS pour créer des Logiciels graphiques multiplate-forme. Il existe des bibliothèques permettant de lire des formats de fichiers 2D ou 3D, Libres ou propriétaires.

Vous pouvez rester de longues heures devant votre ordinateur, à programmer un logiciel répondant à une demande spécifique. Ne vous en faites pas, il existe des écrans n'émettant pas de lumière. Ils permettent de ne pas fatiguer ses yeux, afin de créer vos logiciels en toute sérénité.

2) ARCHITECTURES FREE PASCAL

Avec LAZARUS, en 2010, vous pouvez créer des exécutables fonctionnant sous WINDOWS, sous BSD, sous MAC OS X, sous LINUX, sous DEBIAN, sous UNIX, sous OS X, sous WIN CE, sous I PHONE, sous GPHONE, sous SMARTPHONE. Pour certaines plates-formes il est nécessaire de participer au projet LAZARUS.

Nous utilisons actuellement une architecture 32 bits voire 64 bits. On peut schématiser cela comme 32 ou 64 voies d'autoroutes.

Ainsi une adresse 32 bits peut adresser jusqu'à 2^{32} entiers de 32 bits. Une adresse 64 bits peut adresser jusqu'à 2^{64} entiers de 64 bits. Une adresse 64 bits ne fonctionne pas sur une adresse 32 bits. Par contre une adresse 32 bits peut fonctionner sur une architecture 64 bits, dans la limite de ses possibilités.

LAZARUS fonctionne sur les architectures 32 et 64 bits. Cependant vous devez vérifier que les composants complémentaires fonctionnent en 64 bits. En effet, en 2011, c'est toujours l'architecture 32 bits qui est

prioritaire.

3) APPLICATIONS LIBRES LAZARUS

Une Application Open Source n'est que partagée. Une Application Libre peut s'utiliser, s'étudier, se distribuer et se modifier grâce au partage.

Il existe des applications multiplate-forme faites avec LAZARUS. LAZARUS est choisi pour le multiplate-forme.

SKYCHART et VIRTUAL MOON sont des applications scientifiques en 3D et Libres.

Différents logiciels de gestion vous permettent de gérer votre café, entreprise ou restaurant.

PEA ZIP est un compresseur/décompresseur Libre concurrent de 7 ZIP. Il existe des applications Libres pour les données pour PARADOX ou SQLITE.

ORTHONET est un Logiciel éducatif Libre. Des professeurs ont choisi LAZARUS.

Il existe aussi des utilitaires Libres pour la vidéo ou le son, d'autres projets scientifiques.

Des sites Web vous permettent de télécharger pour participer à ces Logiciels Libres.

Vous avez une liste de logiciels utilisant LAZARUS avec ce lien :
http://wiki.lazarus.freepascal.org/Projects_using_Lazarus/fr

4) DU PASCAL ORIENTÉ OBJET

LAZARUS est basé sur un langage nommé FREE PASCAL. FREE PASCAL est aussi un Compilateur PASCAL orienté Objet. Le langage FREE PASCAL possède une grammaire et des instructions PASCAL, basées sur l'algorithmique. Le PASCAL est donc enseigné pour cette raison particulière.

La programmation par Objets est la programmation la plus proche de l'humain. Cette programmation regroupe des parties du programme, créées dans des Objets. On peut ainsi facilement représenter son programme en entités.

Ces entités ont un comportement, une hiérarchie, des valeurs. Le Pascal Objet possède des Objets pouvant être enregistrés comme Composants. Les Composants PASCAL améliorent le Polymorphisme du PASCAL orienté Objet, cette capacité des Objets à accepter plusieurs formes.

Le langage PASCAL Objet est l'un des plus simple du marché. Il permet de trouver facilement les erreurs. Le Compilateur FREE PASCAL permet de créer des exécutables indépendants de toute librairie complémentaire.

LAZARUS a certes été créé en 1999. Mais il ne fait que reprendre les instructions de PASCAL Objet existantes, grâce au Compilateur multiplate-forme FREE PASCAL, point de départ de départ de LAZARUS. Le Compilateur FREE PASCAL évolue maintenant grâce à LAZARUS.

Comme tout outil Objet RAD, LAZARUS est accessible. Il peut cependant vous emmener, petit à petit, vers les Composants. Vous apprenez alors réellement l'Objet, permettant d'améliorer ce que vous utilisez.

Nous vous apprenons à connaître les instructions permettant de réaliser votre Logiciel, adapté à vos besoins. Pour aller plus loin, connaître l'Objet et les différentes architectures logicielles permet d'améliorer LAZARUS.

5) LA COMMUNAUTÉ

LAZARUS dispose d'une communauté active. Elle ajoute au fur et à mesure de nouvelles possibilités. Des communautés LAZARUS s'ouvrent régulièrement dans d'autres pays.

Il existe un wiki LAZARUS, ainsi qu'un espace de documentation français. Un espace de traduction facilite la compréhension de LAZARUS. Vous pouvez participer à ces espaces, ou donner vos avis.

6) LAZARUS EST PARTAGÉ

Les sites de www.sourceforge.net et Code.google.com ne diffusent que des Logiciels Open Source. SOURCEFORGE et GOOGLE CODE permettent en effet de partager vos Sources de Logiciel. Alors d'autres développeurs peuvent améliorer votre Logiciel, car les Sources sont la recette de votre Logiciel.

LAZARUS est donc Open Source. Vous pouvez le modifier. Ce site permet aussi de disposer d'une mise à jour des Sources, fichier par fichier, selon les modifications. Ces fichiers Sources ne sont pour certains pas testés.

7) LES VERSIONS DE LAZARUS

LAZARUS est un Logiciel Libre. Autrement dit vous avez accès aux Sources, et vous avez le droit de les modifier librement. Vous pouvez donc intégrer le projet LAZARUS.

On a intérêt à mettre à jour LAZARUS, afin de disposer des dernières améliorations Libres. Les développeurs LAZARUS gèrent des numéros de version.

LAZARUS évolue continuellement. Évitez certaines versions.

Le Versioning LAZARUS se présente avec des chiffres, disposés de cette façon :

A.B.CC-D

Le premier nombre nommé "A" présente une version majeure de LAZARUS. Par exemple la version 1.0.00-0 de LAZARUS permettra de traduire entièrement tout Composant DELPHI 6 vers LAZARUS. Plus ce chiffre ou nombre est grand et récent, plus cette version est recommandée.

Le deuxième chiffre ou nombre présente une version mineure de LAZARUS. Plus ce nombre ou chiffre est grand, plus cette version est recommandée.

Le troisième nombre présente la maturité de la version mineure. Si le nombre est pair, la version est stable et utilisable, car elle a été testée.

Si le nombre est impair, il s'agit d'une version non testée, créée un jour donné. Les versions à troisième nombre impair ne veulent rien dire sans leur jour de création. Les versions à troisième nombre impair permettent de montrer un aperçu de la prochaine version paire. Ces versions ne sont jamais recommandées. Avec on a la possibilité de créer des Composants de LAZARUS avec une grammaire FREE PASCAL récente.

Le quatrième chiffre ou nombre montre une création d'une version stable de LAZARUS, à partir d'une base plus récente. En fait le quatrième chiffre ne vous apporte que la réparation de Bogues ou erreurs trouvées, ce qui est déjà intéressant, rien de nouveau sinon. Si le quatrième chiffre est impair la version de LAZARUS n'a pas été testée. Elle a cependant de grandes chances d'être suffisamment stable.

Vous avez la possibilité de télécharger un "snapshot LAZARUS" WINDOWS de la dernière version déboguée de LAZARUS. Pour les autres plates-formes, les Sources du "snapshot" permettent de créer un nouveau LAZARUS.

8) TÉLÉCHARGER LAZARUS

Vous pouvez donc maintenant télécharger la dernière version de LAZARUS pour votre environnement, sur la page de téléchargement du projet LAZARUS à <http://sourceforge.net/projects/LAZARUS>.

Vous pouvez par la même occasion voir l'avancement du projet à <http://www.lazarus.freepascal.org>, dans la page "Roadmap". Si vous souhaitez connaître l'état de la prochaine version allez à <http://bugs.freepascal.org>.

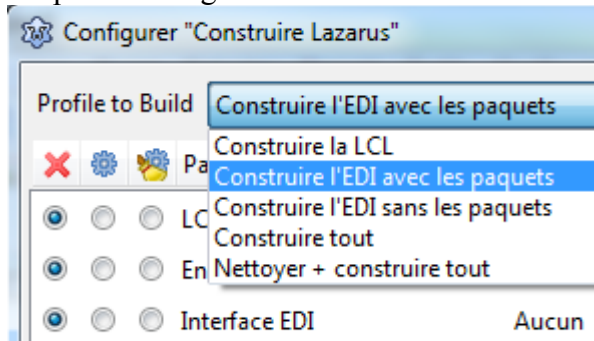
Il est possible de télécharger sa version LAZARUS du jour sur <http://www.hu.freepascal.org/LAZARUS>. On télécharge un Snapshot LAZARUS, c'est à dire une vue sur les modifications du jour apportées à LAZARUS. Un "snapshot" sert à travailler sur des Composants LAZARUS en cours de publication. Il compile rarement un exécutable. Si on souhaite participer à LAZARUS il est possible de télécharger directement les Sources SVN sur le site Web de sourceforge à <http://sourceforge.net/projects/LAZARUS>, dans la partie "Develop".

a) Snapshot sur LINUX

Pour construire avec les sources d'un "snapshot" LAZARUS, remplacez les sources de LAZARUS par celles du "snapshot", puis démarrez LAZARUS comme ceci :

```
su root & start Lazarus
```

Dans "Options" puis "Configurer Build LAZARUS" "Construire tout".



Construire LAZARUS

Allez dans le menu "Outils" puis "Construire LAZARUS".

Si vous voulez travailler sur les Composants LAZARUS, le mieux est de changer le propriétaire du répertoire LAZARUS vers votre compte. Seul votre compte peut alors construire le noyau LAZARUS.

LAZARUS s'installe dans "/usr/lib/lazarus". Si vous souhaitez travailler souvent sur les Composants, il vous est possible d'accéder à ce répertoire, en devenant Propriétaire du répertoire, grâce au terminal LINUX. Utilisez cette commande en mode Administrateur :

```
chown -R mon_login.root /usr/lib/LAZARUS
```

9) INSTALLER LAZARUS SOUS WINDOWS

LAZARUS s'installe en vous demandant de relier les fichiers PASCAL ou LAZARUS au Logiciel LAZARUS. Notez le répertoire d'installation de LAZARUS.

10) INSTALLER LAZARUS SOUS LINUX

LINUX centralise ses paquets de logiciels. Ainsi un logiciel est très léger car il utilise des bibliothèques dans d'autres paquets.

Sachez que LAZARUS utilise des bibliothèques C. Vous devez disposer des bibliothèques disponibles sur LINUX par défaut.

Une version moins récente est disponible dans votre gestionnaire de paquets. Pour disposer de la version plus récente ajoutez le serveur de tests à votre gestionnaire de paquets. Avant vérifiez si la version de www.sourceforge.net est plus récente que celle de votre gestionnaire.

Une version de LAZARUS est dans le gestionnaire de paquets. Il suffit d'installer le paquet "LAZARUS" graphiquement ou en tapant cette commande sur DEBIAN en mode "root" (Administrateur) :

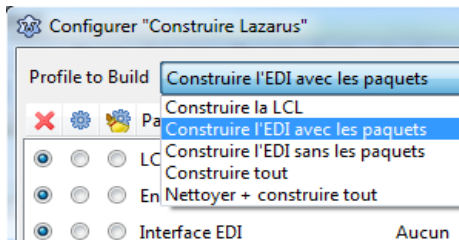
```
apt-get install lazarus
```

Sinon installez l'ensemble des paquets FREE PASCAL COMPILER débutant par "fpc" et "fp-" ainsi que les paquets LAZARUS débutant par "LAZARUS". FREE PASCAL COMPILER peut être trop récent. Faites attention à la version reconnue par LAZARUS.

11) CONFIGURER LAZARUS

LAZARUS se recompile presque entièrement par défaut. Allez dans le menu "Outils" puis "Configurer Build LAZARUS".

Choisissez "Construire l'EDI et les paquets". Ainsi vous reconstruisez l'exécutable de LAZARUS en permettant d'installer de nouveaux Composants.



"Construire l'EDI avec les paquets"

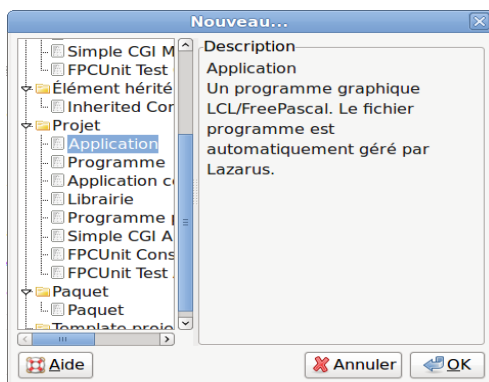
"Construire la LCL", "Nettoyer et construire tout" ou "Construire tout" sont à choisir lorsque des Sources LAZARUS en version d'essai ont été installées. Le nettoyage consiste à supprimer les unités compilées. Pensez à modifier le répertoire de LAZARUS dans "Configuration" puis "Options".

D) PROGRAMMER FACILEMENT

CREATIVE COMMON BY SA

1) CRÉER UN LOGICIEL

Après avoir créé un "Nouveau" projet "Application" allez dans l'"Éditeur de Source". Vous voyez une Source créée automatiquement ainsi qu'une fenêtre qui ne réagit pas comme une fenêtre habituelle. C'est votre base de travail permettant de créer un programme fenêtre.




Nouvelle "Application" LAZARUS

Vous voyez aussi une fenêtre, ou un formulaire, avec une grille en pointillés permettant de travailler graphiquement. Allez dans le menu "Projet" puis dans "Inspecteur de Projet".

Vous voyez les fichiers qui ont été créés automatiquement. Le fichier commençant par "Unit1" c'est le formulaire de votre nouveau projet en

Source PASCAL Objet.

Vous voyez aussi un fichier ".lpr". Cette extension de fichier signifie LAZARUS Project Resources, ou Ressources d'un Projet LAZARUS. Ce fichier, liant le projet vers les ressources LAZARUS, permet de compiler et d'exécuter votre formulaire.

Pour compiler et exécuter ce projet vide, cliquez sur  dans la barre d'outils LAZARUS.

Vous voyez la même fenêtre vide sans les pointillés de présentation. C'est votre formulaire qui est exécuté. Vous pouvez le déplacer, l'agrandir, le diminuer et le fermer, contrairement au formulaire précédent.

Il s'agit bien d'un programme exécuté. Pour exécuter ce programme avec ce formulaire le Compilateur FREE PASCAL a d'abord compilé le formulaire grâce à ces Sources et celles de LAZARUS.

LAZARUS c'est un Environnement de Développement Intégré. Vous avez eu peu de programmation à effectuer pour créer ce logiciel.

a) Le nom d'unité

Allez dans l'"Éditeur de Sources". Placez-vous au début de l'unité du formulaire.

Du Code PASCAL Objet commence toujours par la clause "Unit" reprenant lettre pour lettre le nom du fichier PASCAL. Le nom d'unité se change dans "Fichier", puis "Enregistrer sous".

En général chaque instruction PASCAL est terminée par un ";". Il existe de rares exceptions pour lesquelles le Compilateur vous avertit.

Pour la compatibilité UNIX ou LINUX il est préférable de renommer les noms d'unités en minuscules.

b) La clause "uses"

Ensuite vous avez sûrement à réutiliser des existants. La clause "uses" est faite pour cela. La clause "uses" permet d'aller chercher les existants créés par l'équipe LAZARUS ou bien par un développeur indépendant de LAZARUS.

Lorsque vous ajoutez un Composant grâce à l'onglet des Composants cela ajoute aussi l'unité du Composant dans la clause "uses". Aussi un lien est créé dans le projet vers le regroupement d'unités nommé paquet.

Vous aurez sans doute à ajouter les unités communes LAZARUS comme "LCLType" ou "LCLIntf". Ces bibliothèques remplacent certaines unités de DELPHI comme l'unité "windows".

Pourquoi n'a-t-on pas gardé certaines unités DELPHI ?

Les bibliothèques LAZARUS possèdent une nomenclature. Une bibliothèque possédant le mot "win" est une bibliothèque système spécifique à la plateforme WINDOWS. Elle ne doit pas être utilisée directement dans vos programmes. Les unités "gtk" servent à LINUX, "carbon" à MAC OS, "unix" à UNIX.

Vous avez un aperçu de ces mots clés dans les "Outils". La configuration de construction de LAZARUS vous montre l'ensemble des plates-formes indiquant les unités à éviter. Dans les Sources de LAZARUS, vous voyez des unités commençant pas ces plates-formes ou comportant ces noms de systèmes.

c) L'Exemple

L'exemple consiste à créer une fenêtre permettant d'afficher "Hello World !".

Nous allons éditer un nouveau projet.

Démarrez LAZARUS.

Allez dans "Fichier" puis "Nouveau". Choisissez "Application".

Tout d'abord nous allons rechercher l'unité "Dialogs" avec beaucoup de fainéantise.

Placez-vous après une virgule de la clause "uses". Comme chaque instruction, Cette dernière se termine par un ";;".

En respectant la présentation LAZARUS, tapez uniquement "di".

Ce qui va suivre ne fonctionne que si votre Code Source est correct, du début de l'unité jusqu'à votre ligne éditée.

Appuyez en même temps sur "Ctrl" avec ensuite la barre d'espace. Relâchez. Il s'affiche une liste d'unités LAZARUS. Choisissez "dialogs" avec les flèches et la touche "Entrée". N'oubliez pas de placer ensuite une virgule pour ne pas créer d'erreur de compilation. Vous avez ajouté l'unité Dialogs.

Pour voir ce que vous avez ajouté, maintenez enfoncé "Ctrl" tout en cliquant sur l'unité "Dialogs". L'unité "Dialogs" s'affiche. Vous allez vers la Source liée en faisant de cette manière. LAZARUS va chercher ce qu'il connaît grâce aux liens de vos projets et un Code correct.

On voit que la clause "uses" de "Dialogs" contient d'autres unités. Celles-ci sont elles aussi ajoutées à votre projet, si elles ne l'étaient pas déjà. En effet des unités obligatoires sont toujours utilisées.

Toute écriture de Code ajoute des informations à votre exécutable, avec en plus des unités qui sont intégrées. Votre programme compilé contient beaucoup de Code, mais il est indépendant.

Grâce aux onglets de l'"Éditeur de Sources", vous pouvez retourner sur votre Source en cliquant sur l'onglet "Unit1".

Ce nom "Unit1" n'est pas explicite. Nous allons le renommer en gardant une partie des informations importantes.

Cliquez dans le menu LAZARUS sur "Fichier" puis "Enregistrer". Créez votre répertoire de projet, puis sauvegardez votre unité en commençant par "u_". Cela indique comme avant que l'on sauvegarde une unité. Les fichiers sont ainsi regroupés au même endroit. Ensuite on met le thème de l'unité : "hello". Cela donne "u_hello". Sauvegardez.

Si vous avez mis des majuscules dans le nom d'unité, LAZARUS peut vous demander de renommer ce nom d'unité en minuscule. N'allez que rarement contre ses recommandations.

Votre nom d'unité est donc "u_hello". Elle peut être rajoutée dans une clause "uses" d'une autre unité du projet.

La présentation

Nous allons rajouter un bouton à notre formulaire. Pour aller sur le formulaire de l'unité "u_hello" appuyez sur "F12", quand vous êtes sur l'unité dans l'"Éditeur de Source". Cliquez de nouveau sur "F12" permet

de revenir sur l'"éditeur de Source". Revenez sur la fiche.

Allez sur la palette des Composants. Sélectionnez dans le premier onglet "Standard" le bouton "TButton". Cliquez l'icône "TButton". Il s'enfonce.

Cliquez sur le formulaire pour placer votre "TButton". Vous voyez à l'exécution la même fenêtre avec son "TButton" sans les points noirs. Ces points noirs servent à se repérer à la conception.

Cliquez sur le bouton de votre formulaire puis sur "F11". Vous sélectionnez l'"Inspecteur d'Objets", qui permet de modifier votre "TButton".

Vous êtes sur l'onglet "Propriétés" de l'"Inspecteur d'Objets". Cet onglet classe les propriétés par ordre alphabétique.

Vous pouvez changer la propriété "Name" de votre "TButton", afin de renommer votre "TButton" en "BCliquez". Vous voyez alors le texte du "TButton" changer. C'est une automatisation de ce Composant qui change sa propriété "Caption", quand on change pour la première fois son nom.

Pour vous rendre compte, allez sur la propriété "Caption" et enlevez le "B". Le nom de votre Composant n'a pas changé. Pourtant la propriété "Caption" a bien changé le texte de votre "TButton".

Le Code Source

Allez sur l'onglet "Evènements" de l'"Inspecteur d'Objets". Double cliquez sur l'évènement "OnClick". Cela fait le même effet que d'appuyer sur les trois points : Vous créez un lien vers une procédure dans votre "Éditeur de Source".

Tapez entre le "Begin" et le "End;" de cette nouvelle procédure deux espaces puis "show". Puis appuyez en même temps sur "Ctrl" suivi de "Espace".

Si vous avez ajouté l'unité "Dialogs" vous voyez la procédure "ShowMessage". Cette procédure simple possède un seul paramètre chaîne. Sélectionnez cette procédure avec les flèches et entrée. Vous pouvez aussi utiliser la souris.

Vous avez maintenant à ajouter le paramètre permettant d'afficher du texte. Les paramètres des procédures sont intégrés grâce aux parenthèses.

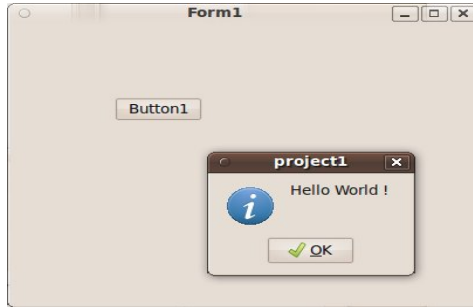
Ouvrez votre parenthèse. Les chaînes sont ajoutées en les entourant par un caractère "". L'instruction se termine par un ";". Écrivez donc :

```
Begin  
  ShowMessage ( 'Hello World' );  
End;
```

Nous allons afficher un message "Hello World" quand on clique sur le bouton "BCliquez".

Appuyez simultanément sur "Ctrl" suivi de "F9". Cela compile votre exécutable, en vérifiant la syntaxe et en donnant des avertissements.

Cliquez sur ou appuyez sur "F9" pour exécuter votre Logiciel fenêtre. Après avoir appuyé sur le bouton voici ci-après ce que cela donne :



Un exemple mondial avec un événement

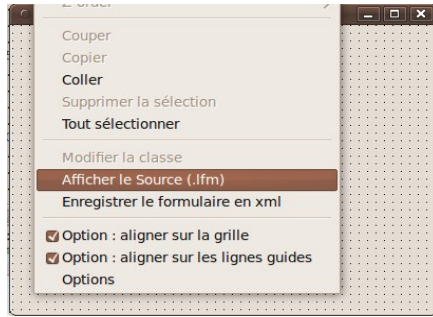
d) La Source du formulaire

Le type égal à "class (TForm)" décrit l'élément essentiel de l'EDI LAZARUS. Il s'agit de la fenêtre affichée contenant le bouton. On ne voit pas dans cette déclaration l'endroit où on a placé le bouton. L'endroit où on a placé le bouton se modifie dans l'"Inspecteur d'Objets".

L'"Inspecteur d'Objets" sauve les propriétés dans le fichier portant l'extension ".lfm". Vous pouvez voir ce fichier en affichant sur le formulaire modifiable un menu, grâce au bouton droit de la souris. Dans ce menu cliquez sur "Afficher le Source".

En scrutant le fichier ".lfm" vous voyez le bouton et ses coordonnées affichées.

Le type égal à "class (TForm)" permet d'afficher la fenêtre et son bouton en lisant ce fichier ".lfm".



"Afficher le Source" dans la conception du formulaire

Les modifications apportées au fichier permettent entre autre d'invertir des Composants. Pour faire cela vous avez relié le paquet du Composant au projet grâce au "Gestionnaire de projet".

2) PAQUET POUR DÉBUTANTS

Dans la version 0.9.30 de LAZARS a été ajouté un paquet, une bibliothèque donc, permettant de faciliter le développement pour les débutants.

Ce paquet installe des fonctionnalités LAZARUS permettant de s'habituer au nommage, tout en facilitant la création de Code au sein de LAZARUS.

Allez dans "Paquets", puis "Configurer les paquets". Une fenêtre s'ouvre.

Sur la liste de droite, sélectionnez le paquet "educationlaz", pour l'"Installer". "Reconstruisez LAZARUS".

Après que LAZARUS ait redémarré, dans "Configuration", puis "Options", vous pouvez changer les paramètres pour débutants dans

"Éducation".

Des les options d'"Education", dans "Général", activez le mode "Débutant".

Dans ces options toujours et dans la "Palette de Composants" "Montrez tout". Il est en effet possible d'y cacher les composants que l'on ne souhaite pas utiliser. Mais cette option, activée pour les professeurs, est surtout réservée aux démonstrations.

Vous pouvez afficher ou cacher d'autres outils, comme les boutons de référencement d'unités, de classes.

3) INDENTATION PASCAL

Vous voyez dans l'exemple que le Code Source est joliment présenté. Cela n'est pas une futilité puisque vous avez eu envie de le regarder. Vous avez peut-être eu envie de prendre exemple sur ce qui avait été fait. Voici la recette à suivre pour faire de même et mieux encore.

Un Code Source bien présenté ce sont des développeurs avec un comportement clair et précis. L'indentation c'est la présentation du Code Source pour qu'il devienne lisible. En effet vous créez du Code machine, alors que ce sont des hommes et femmes qui seuls peuvent le scruter, et le modifier correctement. Les erreurs sont toujours humaines.

L'indentation c'est donc du Code Source lisible. L'indentation permet de présenter le Code Source afin de le comprendre comme clair, sans avoir à fouiller pour trouver ce que l'on cherche.

L'indentation des éditeurs PASCAL est simple. En voici un résumé :

- Chaque instruction ou nœud (var, type, class, if, while, case, etc.) est à la même position que la ligne précédente

- Chaque imbrication d'un nœud doit être décalée de deux espaces vers la droite
- Les instructions incluses entre les "Begin" et "End ;" sont décalées de deux espaces vers la droite

Cette présentation permet de revoir du Code Source plus facilement. Les développeurs qui regardent du Code Source avec cette présentation sont plus précis et plus sûrs d'eux.

Pour aider à indenter deux combinaisons de touches sont très importantes dans les éditeurs PASCAL :

- Ctrl + K + U permet de décaler votre sélection de lignes de deux espaces vers la gauche.
- Ctrl + K + I permet de décaler votre sélection de lignes de deux espaces vers la droite.

Vous pouvez aussi ajouter des règles de présentation du Code qui facilitent la maintenance. Nous vous disons pourquoi appliquer ces règles de présentation :

- Des commentaires sont obligatoirement avant chaque Code Source de fonctions ou procédures. Ainsi les commentaires avant chaque fonction ou procédure sont vus dans l'éditeur FREE PASCAL lorsque vous mettez la souris sur la fonction ou procédure.
- Des commentaires sont dans le Code Source si on y trouve un Bogue ou erreur. Si on corrige un Bogue dans le Code Source, il est possible que l'erreur revienne s'il n'y a pas de commentaires.
- Des commentaires sont dans le Code Source dès que l'on réfléchit pour écrire une Source. Si le Code demande réflexion pour être fait, c'est que vous y réfléchirez à nouveau si vous n'y mettez pas de commentaires. Notre mémoire oublie ce que l'on a écrit.
- Chaque fonction doit tenir sur la hauteur d'un écran. Une fonction qui ne tient pas sur la hauteur d'un écran est non

structurée, illisible, difficile à maintenir, difficilement compilable si retravaillée.

Il existe d'autres méthodes de structuration, comme par exemple mettre des séries d'instructions alignées mot à mot. Cela permet de retrouver facilement les différences si le Code Source se répète.

Comme l'éditeur FREE PASCAL peut terminer les mots, il est possible d'affecter des longs noms de variables et procédures.

Toute nouveauté de l'éditeur est à saisir. Cela facilite le travail. Vous pouvez définir votre présentation et la tester.

Les exemples que nous vous montrons après sont indentés.

a) Paramétrer l'indentation

Vous pouvez modifier les "Options" d'"Indentation" dans "Configuration", puis "Options", puis "Outils de Code", puis "Général", puis "Indentation".

4) STRUCTURE DU CODE SOURCE

Nous allons décrire le Code Source précédemment montré.

En général une unité faite en PASCAL se structure avec d'abord un en-tête, puis des déclarations publiques, puis le Code Source exécutable. Toute unité PASCAL se termine par "end."

a) La déclaration Unit

La déclaration "Unit" est la première déclaration d'un fichier PASCAL. Elle est obligatoire et obligatoirement suivi du nom de fichier PASCAL, sans l'extension. On change cette déclaration en sauvegardant l'unité.

b) Les déclarations publiques

Les déclarations publiques comprennent obligatoirement le mot clé "interface" indiquant le début des déclarations publiques.

Vous pouvez ensuite y placer l'instruction "uses" permettant d'utiliser dans les déclarations publiques les unités déjà existantes.

Vous pouvez ensuite placer toute déclaration constante, type, variable, fonction ou procédure. Toute déclaration incluse dans la compilation ajoute du Code à l'exécutable.

Les variables allouent soit un pointeur soit un type simple.

Un pointeur est une adresse pointant vers un endroit de la mémoire. Il permet de définir et de stocker des types complexes.

Les constantes publiques peuvent être centralisées dans une ou plusieurs unités.

c) Le Code Source exécuté

Le Code Source commence par le mot clé "implementation".

En plus des déclarations déjà définies, on place dans cette partie le Code Source exécuté grâce aux fonctions et procédures.

Dans le Code Source exécuté on peut placer les mêmes déclarations que celles données précédemment. Seulement elles ne sont plus publiques donc pas utilisables ailleurs.

La procédure "ShowMessage" vous a permis d'afficher une boîte avec un message et un bouton. Si vous regardez le Code Source de "ShowMessage" grâce à "Ctrl" vous voyez qu'elle est déclarée avec le mot clé "procedure".

Une fonction s'appelle de la même façon mais renvoie une variable retour. En effet le rôle des fonctions et procédures est de modifier une partie de votre Logiciel. Elles transforment une entrée en sortie comme le ferait une équation mathématique.

L'événement que vous avez créé est une procédure particulière qui a permis d'afficher votre message à l'écran. La sortie ici c'est l'affichage de "Hello World" dans une fenêtre de dialogue.

Toute fonction ou procédure déclarée publiquement doit avoir une correspondance dans le Code Source exécuté. Le Compilateur vous le rappelle si vous oubliez. Le Code Source que vous avez ajouté était dans la partie "implementation".

d) Le Code Source exécuté au chargement

Il est possible d'exécuter du Code Source dès le chargement de l'unité afin d'automatiser la gestion de l'unité. Ces instructions sont à utiliser en dernier recours.

Le Code Source placé après le mot clé "initialization" permet de créer des Objets toujours en mémoire dès que l'on utilise l'unité.

Le Code Source placé après le mot clé "finalization" permet de détruire les Objets en mémoire à la fermeture du programme.

e) Fin de l'unité

La fin de l'unité est définie par "end."

5) LES FICHIERS RESSOURCES

A la fin d'un fichier PASCAL contenant votre formulaire, voici ce que l'on peut mettre :

```
{$IFDEF FPC}  
  {$i Unite.lrs}  
{$ENDIF}
```

Ces trois lignes signifient : Si la directive FPC ou Compilateur FREE PASCAL est activée, alors inclure le fichier "Unite.lrs".

LAZARUS utilise d'autres fichiers ressources que Turbo PASCAL, ou DELPHI. On appelle les fichiers ressources LAZARUS les fichiers ".lrs" qui nécessitent idéalement d'avoir des fichiers images en format image compressée XPM. Les fichiers ".lrs" peuvent contenir aussi le fichier ".lfm" dans les unités de formulaires, pour les versions anciennes de LAZARUS.

Nous vous expliquons plus loin la création de votre premier Composant. Une partie de ce chapitre décrit la création et l'utilisation du fichier ".lrs".

6) TOUCHES DE RACCOURCIS DE COMPLÉTION

La complétion permet d'écrire à votre place. C'est indispensable pour tout programmeur cherchant à gagner du temps.

Ctrl+Espace Rechercher dans le Code lié :

Ctrl+Maj
+Espace Afficher la syntaxe des méthodes :

Ctrl+Maj+ Dans l'éditeur : Aller directement de la déclaration d'une
flèche haut ou méthode à son implémentation.

bas

Ctrl+Maj+C Complément de Code__ : vous entrez **procedure**
toto(s:string) dans :

```
type  
  TForm1 = class(TForm)  
    private  
      procedure toto(s:string);  
    public
```

Il vous écrit, dans la partie implémentation :

```
procedure TForm1.toto(s:string);  
begin
```

```
end;
```

Ctrl+Maj+i Indenter plusieurs lignes à la fois

ou Passer de

Ctrl+Maj+u

begin	begin
if X >0 then x=0;	if X >0 then x=0;
if X<0 then x=-1;A	if X<0 then x=-1;
A:=truc+machintruc;	A:=truc+machintruc;
end;	end;

Sans le faire ligne par ligne ?

Sélectionner les lignes puis faire Ctrl+Maj+i pour déplacer les lignes vers la droite, ou Ctrl+Maj+u pour les déplacer vers la gauche.

Maj+touche
fléchée

et

Ctrl+touche
fléchée

Ctrl+j

Positionner ou dimensionner au pixel près un Composant

Faire appel aux modèles de Code.

Nota : les modèles de Code permettent d'écrire du Code tout fait par exemple: en choisissant le modèle de Code if then else après avoir fait Ctrl + j, on obtient

```

if then
begin

end
else
begin

end;

```

Autre astuce : chaque modèle de Code possède une abréviation. Tapez cette abréviation puis Ctrl + j. Tapez par exemple "if" puis Ctrl + j. Les lignes correspondant à "if then else" s'écrivent à votre place. Les abréviations disponibles sont visibles en faisant Ctrl + j.

- Ctrl+Maj + 1 à 9
et
Ctrl+ 1 à 9
- Placer des marques (des signets) dans un Source pour pouvoir y revenir ultérieurement
Vous êtes sur un bout de Source et vous vous aller voir ailleurs dans l'unité puis revenir rapidement :
- Tapez :CTRL SHIFT 1 (ou un chiffre de 1 a 9 au dessus des lettres et non dans le pavé numérique) l'éditeur met un "1" dans la marge.
 - Pour revenir vous faites CTRL avec 1
 - Pour annuler la marque, soit vous vous placez sur la ligne et vous refaites CTRL SHIFT 1, soit vous vous placez ailleurs et vous refaites CTRL SHIFT 1, pour déplacer la marque.
- Ctrl+ flèche droite
ou gauche
- Se déplacer au mot suivant ou précédent.
- Ctrl+Maj + droit
gauche
- Se déplacer au mot suivant ou précédent tout en sélectionnant.
- Ctrl +haut ou
bas
- Revient au même que d'utiliser l'ascenseur

a) Touches de raccourci projet

Touche	Action du menu
Ctrl+F11	Fichier Ouvrir un projet
Maj+F11	Projet Ajouter au projet
F9	Exécuter Exécuter
Ctrl + F9	Compiler le projet
Ctrl+S	Fichier Enregistrer

Ctrl+F2	Réinitialiser le programme = arrête votre programme et revient en mode édition
Ctrl+F4	Ferme le fichier en cours

7) TOUCHES DE RACCOURCIS DE VISIBILITÉ

Touche	Action du menu
Ctrl+F3	Voir Fenêtres de débogage Pile d'appels
F1	Affiche l'aide contextuelle (si si je vous assure)
F11	Voir Inspecteur d'Objets
F12	Voir Basculer sur Formulaire/Unité
Ctrl+F12	Voir Unités
Maj+F12	Voir formulaires
Ctrl+Maj+E	Voir l'explorateur de Code
Ctrl+Maj+B	Voir l'explorateur de classe
Ctrl+ Maj+T	Ajouter un élément à faire
Alt+F10	Affiche un menu contextuel
Alt+F11	Fichier Utiliser l'unité
Alt+F12	Bascule Form visuel / Source de la forme

8) TOUCHES DE RACCOURCIS DE DÉBOGAGE

Les raccourcis de débogage sont généralement utilisés dans l'éditeur de Source. Ils permettent de scruter le Code Source. Il est possible de placer un point rouge sur une ligne de votre Source, dans la marge

grisée à gauche de votre Source. C'est un point d'arrêt. Il va être appelé dès que l'on exécutera la ligne avec ce point d'arrêt.

a) Exercice

Placez un point d'arrêt à "ShowMessage" dans votre exemple. Exécutez et cliquez sur le bouton. En général le programme se fige, puis LAZARUS se place sur le "Point d'arrêt" que vous avez mis.

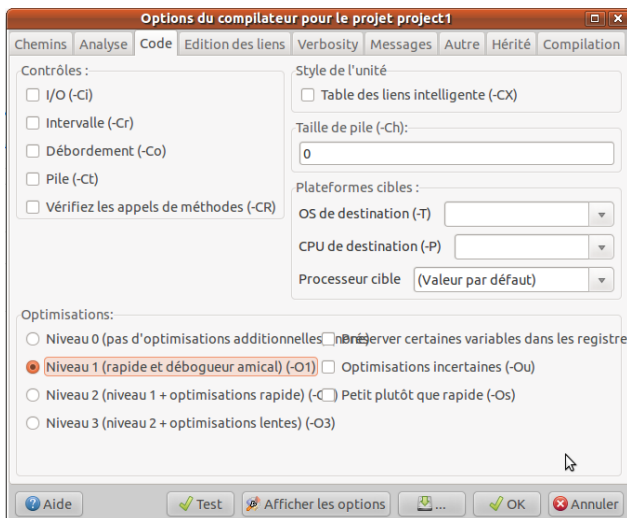
b) Mes points d'arrêt sont inactifs

Pour que ces points d'arrêt fonctionnent, votre projet doit se compiler d'une certaine manière. Votre exécutable devient ainsi dix fois plus important qu'habituellement. Si les points d'arrêt fonctionnent, votre exécutable est très lourd.

Allez dans le menu "Projet", puis "Options du Projet", puis "Options de compilation".

Allez dans l'onglet "Édition des liens". Cochez "Afficher les numéros de lignes". Ces numéros servent à ce que le débogueur se repère. Ces numéros de ligne alourdissent votre exécutable.

Allez dans l'onglet Code. En bas dans "Optimisations" choisissez l'option "Niveau 0" ou "Niveau 1». Le niveau 0 crée un très gros exécutable mais scrute tous les points d'arrêt.



Le niveau 1 par défaut crée un gros exécutable

Touche	Action du menu
F4	Exécuter Jusqu'au curseur
F5	Exécuter Ajouter un point d'arrêt
F7	Exécuter Pas à pas approfondi
Maj+F7	Exécuter Jusqu'à la prochaine ligne
F8	Exécuter Pas à pas
Ctrl+F5	Ajouter un point de suivi sous le curseur
Ctrl+F7	Evaluer/Modifier

9) TOUCHES DE RACCOURCIS DE L'ÉDITEUR

Touche	Action
Ctrl+K+B	Marque le début d'un bloc

Ctrl+K+C	Copie le bloc sélectionné
Ctrl+K+H	Affiche/cache le bloc sélectionné
Ctrl+K+K	Marque la fin d'un bloc
Ctrl+K+L	Marque la ligne en cours comme bloc
Ctrl+K+N	Fait passer un bloc en majuscules
Ctrl+K+O	Fait passer un bloc en minuscules
Ctrl+K+P	Imprime le bloc sélectionné
Ctrl+K+R	Lit un bloc de procedure puis un fichier
Ctrl+K+T	Marque un mot comme bloc
Ctrl+K+V	Déplace le bloc sélectionné
Ctrl+K+W	Écrit le bloc sélectionné dans un fichier
Ctrl+K+Y	Supprime le bloc sélectionné

10) TOUCHES DE RACCOURCIS DE L'ENVIRONNEMENT

Touche	Action du menu
Ctrl+C	Édition Copier
Ctrl+V	Édition Coller
Ctrl+X	Édition Couper

E) LE LANGAGE PASCAL

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Nous avons appris à utiliser un programme existant. Cela semblait facile. Nous ne savions pas pourtant l'utilité des instructions et de la syntaxe que nous avons testées.

Nous allons dans ce chapitre apprendre la syntaxe et les instructions nécessaires à la création d'un programme fait en FREE PASCAL.

2) INTRODUCTION

Nous allons créer des programmes non graphiques. Pour exécuter et voir un programme textuel, utilisez, dans les "Accessoires", la "Ligne de commande", ou le "Terminal" sous LINUX.

3) INSTRUCTION PASCAL

Une instruction est une commande PASCAL exécutant une tâche.

Dans l'exemple précédent l'appel à l'instruction "ShowMessage" se terminant par un ";" est une instruction.

Une instruction PASCAL se termine souvent par un ";". Choisissez d'affecter un ";" quand vous écrivez la plupart des instructions.

4) LES FONCTIONS ET PROCÉDURES

Comme tous les langages, PASCAL permet de découper un programme en plusieurs parties nommées souvent "modules". Cette "programmation modulaire" se justifie pour différentes raisons:

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Aussi il devient difficilement modifiable.
- Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le programme principal les instructions des modules en décrivant les enchaînements.
- Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée en modules plus élémentaires; ce processus de décomposition pouvant être répété autant de fois que nécessaire. Il est à noter que les méthodes de "programmation structurée" conduisent tout naturellement à ce découpage.
- La programmation modulaire permet d'éviter des séquences d'instructions répétitives. En particulier, nous verrons comment la notion d'argument permet de "paramétrer" certains modules.
- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois.

La programmation modulaire a été suivie par la programmation par Objets, une programmation encore plus proche de l'humain.

a) Définition

Les fonctions et procédures sont du Code Source exécuté avec les paramètres ou variables en entrée. Cela donne un résultat ou sortie.

5) TESTER SON PROGRAMME

Définir une fonction ou procédure permet de savoir comment tester cette fonction ou procédure à partir des règles de gestion de votre programme. Les règles de gestion sont ce qui permet de réaliser des instructions élémentaires et personnalisées. Nous vous expliquons des règles de gestion "Objet" dans le chapitre sur la "Persistance d'un Logiciel".

Lorsqu'on écrit une fonction on définit ce que fait cette fonction. On sait donc les limites de sa fonction. Par exemple une fonction retournant un âge retourne en 2011 un nombre entier entre 0 et 150. Une fonction ajoutant des âges retourne un entier positif. Plus l'entier a une limite importante plus la somme peut être importante. On peut par exemple choisir le type "Cardinal", un entier plus grand que "Integer".

Les jeux de tests consistent à vérifier que les fonctions et procédures retournent les bons résultats dans et sur les limites en entrée et en sortie de sa fonction ou procédure.

Les jeux de tests sont très utiles dans l'industrie, ou pour tester ses Composants.

6) LES TYPES SIMPLES DÉFINIS

Le langage PASCAL vous demande de définir ce que vous faites. Ainsi on définit ce qui est utilisé par les fonctions et procédures.

Vous pouvez utiliser des types définis ci-dessous sans avoir à utiliser quelque unité que ce soit :

Type	Définition
Shortint	Entier compris entre -127 et 128
Byte	Entier compris entre 0 et 255
Integer	Entier compris entre -32 768 et 32 767
Word	Entier compris entre 0 et 65535
Longint	Entier long compris entre - 2 147 483 648 et 2 147 483 647
Cardinal	Entier long compris entre 0 et 4294967295
Int64	Entier d'architecture 64 bits compris entre -2^{63} et $2^{63}-1$
UInt64	Entier d'architecture 64 bits compris entre 0 et $2^{64}-1$
Real	Réel compris entre $2,9 \text{ e } 10^{-39}$ et $1,7 \text{ e } 10^{38}$ avec 11 décimales
Double	Réel double précision compris entre $5,0\text{e}10^{-324}$ et $1,7\text{e}10^{308}$ avec 15 décimales
Extended	Réel compris entre $1,9\text{e } 10^{-4951}$ et $1,1\text{e } 10^{4932}$
Char	Un caractère alphanumérique de même taille que Byte
ShortString	Chaîne de caractères courte à taille variable
String	Chaîne de caractères à taille variable
WideString	Chaîne de caractères lourde à taille variable
Boolean	Valeur logique égale à TRUE ou FALSE

a) Exemple

Une procédure s'écrit par exemple comme ceci :

```
procedure hello ( nom : String );  
Begin  
    ShowMessage ( 'Bonjour Monsieur ou Madame ' + Uppercase  
( nom ) + ' ! ' );  
End;
```

Cette procédure utilise une variable paramètre de type "String" ou chaîne. Elle utilise une fonction de mise en majuscules qui permet d'afficher une fenêtre à l'utilisateur avec le message "Bonjour Monsieur ou Madame [nom en majuscules] !".

Un bouton "OK" que l'on ne voit aucunement dans le Code Source permet de quitter cette fenêtre. Une fenêtre ShowMessage peut servir à tester son Logiciel en cas de défaillance du débogueur.

La procédure porte le nom hello et possède un paramètre chaîne. Le Code Source modifiant la sortie de la procédure commence par le mot "Begin" et se termine par "End;". On place entre le "Begin" et le "End ;" ses instructions, ici "ShowMessage".

La procédure hello appelle la procédure "ShowMessage", qui permet d'afficher en premier plan une fenêtre avec la chaîne "Bonjour" suivie du paramètre "nom", puis un point d'exclamation.

Pour appeler la procédure nommée "hello" appelez ce nom suivi d'un paramètre chaîne entre parenthèses. Vous remarquez que cela se fait de la même manière que l'appel à "ShowMessage".

"ShowMessage" est aussi une procédure que vous pouvez scruter en maintenant enfoncée la touche "Ctrl" et en cliquant dessus.

7) LES VARIABLES GLOBALES

Notre exemple réalisait l'affichage d'un message dans une fenêtre. Pour afficher le message nous avons utilisé une procédure. Par contre nous avons écrit très peu de Code pour créer la fenêtre. Comme on le voit on déclare un type fenêtre. Puis on le déclare en variable.

Comme vous le remarquez la déclaration de la variable fenêtre avec "var" n'est incluse dans rien. C'est donc une variable globale. Cette variable a pu être utilisée dans la création de l'Application en demandant l'unité de la fenêtre nommée "Forms".

a) But d'une variable globale

Une variable globale est une variable pouvant être réutilisée dans tout un programme.

b) Inconvénients d'une variable globale

Une variable globale ancêtre de "TForm" ne prend que peu d'espace au départ. Ce n'est qu'un simple pointeur, autrement dit une adresse, ne contenant aucun lien viable vers votre formulaire. Si vous essayez d'accéder à votre formulaire avant sa création se produit alors une erreur.

Seulement dans l'exemple l'ensemble visible de la fenêtre cette variable globale est créée à la création de l'Application avec CreateForm. Cette

variable peut ne pas être détruite lorsqu'on ne l'utilise plus, afin d'être réutilisée dans tout le programme.

Si on ne détruit pas cette variable globale à sa fermeture la variable globale prend autant d'espace que lorsqu'elle était visible. Les formulaires ancêtres de "TForm" contiennent beaucoup d'Objets et de Composants. Les instances des ces Objets ou Composants ne sont pas détruits par défaut exceptés s'ils possèdent un parent. Seulement si le parent n'est pas détruit la fenêtre n'est pas détruite.

Détruisez la variable globale fenêtre si vous ne la réutilisez pas.

Si votre Logiciel possède beaucoup de formulaires ou fenêtres vous pouvez ajouter cette Source à l'événement "OnClose" de votre formulaire :

```
Procedure TmaVariableGlobaleFormulaire.FormClose (Sender: TObject; var CloseAction: TCloseAction);  
begin  
    CloseAction := caFree;  
    MaVariableGlobaleFormulaire := nil ;  
end;
```

Il est préférable de libérer un formulaire car les formulaires prennent beaucoup de place en mémoire.

Mettre la variable de formulaire à "nil" peut parfois être nécessaire quand on automatise la gestion des formulaires.

Dans le chapitre sur la création de Composants nous vous donnons la possibilité de ne pas faire de copié-collé à chaque création de fenêtre.

c) **Exercice**

Créez une Application avec deux formulaires. Le premier formulaire

qui est toujours le formulaire principal permet d'afficher le second. Le second formulaire se détruit à la fermeture.

8) ***LES VARIABLES LOCALES***

En fait, dans notre précédent exemple, la variable fenêtre était "globale" parce qu'elle était indépendante. On ne la voyait imbriquée dans rien.

Or il est tout à fait possible d'effectuer au sein d'une procédure des déclarations de types, de variables, d'étiquettes, de procédures... Dans ces conditions, les nouvelles "entités" variables, types, étiquettes, procédures ainsi déclarées ne sont connues qu'à l'intérieur de la procédure dans laquelle elles ont été déclarées. On dit qu'elles sont "locales" à la procédure, ou encore que leur portée est limitée à la procédure.

Dans ce premier exemple nous laissons le Code Source créé par LAZARUS lors de la création d'un "Nouveau" "Programme". Il est préférable de faire cela. Faites de même. Voici l'exemple :

```
program exemple_procedure_3 ;  
  
{ $mode DELPHI }  
uses  
  { $IFDEF UNIX } { $IFDEF UseCThreads }  
  cthreads,  
  { $ENDIF } { $ENDIF }  
Classes  
  { you can add units after this } ;  
  
var c1, c2 : char ;  
procedure tricar ;
```

```

var c : char ;
begin
    if c1>c2 then
        begin
            c := c1;
            c1:=c2;
            c2:=c;
        end;
    end ;
begin
    write ('donnez 2 caractères : ');
    readln (c1, c2) ;
    tricar ;
    write ('caractères tries : ');
    writeln (c1,c2);
end.

```

Le rôle de la procédure "tricar" est simplement de ranger par ordre alphabétique les caractères contenus dans les deux variables globales "c1" et "c2" en procédant, si nécessaire, à un échange de leurs valeurs. Pour faire cela elle utilise la variable "c" comme variable intermédiaire.

Or vous constatez que "c" a été déclarée au sein de la procédure "tricar". Cette fois, c n'est connue qu'au sein de tricar; on dit que sa portée est limitée à la procédure "tricar" ou encore que "c" est "locale" à tricar. Elle n'est mise en mémoire que lors de l'utilisation de "tricar".

Que se passerait-il si l'on cherchait à utiliser "c" en dehors de "tricar", par exemple, en ajoutant une simple instruction telle que:

```
write (c);
```

Après l'appel de "tricar" dans le programme principal. Le Compilateur signalerait simplement que l'identificateur "c" n'est pas connu à ce niveau.

Si, en revanche, nous déclarions, en plus de la déclaration faite dans "tricar", une variable "c" dans le programme principal, celle-ci serait indépendante de la variable "c" connue dans "tricar". Voyez cet exemple qui reprend le précédent auquel nous avons ajouté la déclaration:

```
var c : char;
```

Ainsi que les deux instructions:

```
c := 'A' ;  
writeln ('et dans c : ', c);
```

Elles permettent d'affecter une valeur à la variable globale "c" avant l'appel de "tricar" et de l'afficher après cet appel, afin de montrer qu'elle n'a effectivement pas été modifiée. N'oubliez pas de laisser le Code Source créé par LAZARUS selon le premier exemple. Voici une amélioration de tri de caractères :

```
program exemple_procedure4 ;  
var c1, c2 : char ;  
    c : char ;  
procedure tricar ;  
var c : char ;  
    begin  
        if c1 > c2 then  
            begin  
                c := c1; c1 := c2 ; c2 := c ;  
            end;  
    end ;  
begin  
    c := 'A';  
    write ('donnez 2 caractères : ');  
    readln (c1, c2) ;  
    tricar ;  
    write ('caractères triés      ');
```

```
writeln (c1,c2) ;  
writeln (' et dans c : ', c);  
end.
```

9) LA NOTION DE PROCÉDURE

Vous avez été amené à utiliser une procédure "ShowMessage" et une fonction "Uppercase".

Vous voyez que la fonction "Uppercase" peut être imbriquée dans une procédure. Il est cependant impossible d'écrire :

```
UpperCase ( Showmessage (nom));
```

Si vous écrivez cette instruction il y aura une erreur de compilation à "ShowMessage". On vous indiquera que c'est une procédure ne retournant pas de résultat. On demande à UpperCase un type chaîne en paramètre. Comme "ShowMessage" ne retourne rien cela crée une erreur.

L'écriture correcte est bien :

```
ShowMessage ( UpperCase ( nom ));
```

La fonction "Uppercase" renvoie une chaîne à "ShowMessage".

Une fonction possède généralement un ou plusieurs arguments et elle fournit un résultat. Quant à l'aspect modulaire, si on mentionne le nom d'une fonction dans un programme, on entraîne l'appel de tout un ensemble d'instructions, regroupées dans le Code Source de la fonction. Ces instructions réalisent le calcul pour donner la valeur à renvoyer.

La procédure, quant à elle, n'est rien d'autre qu'une généralisation de la notion de fonction. Elle peut elle aussi posséder des arguments. En

revanche, elle peut en retour fournir un ou plusieurs résultats, ou même aucun. Mais ces résultats s'ils existent ne constituent pas nécessairement la seule utilité de la procédure. Par exemple cette dernière peut imprimer un message.

On peut dire de la procédure qu'elle réalise une action, terme plus général que calcul. Notez que nous avons déjà été amené à utiliser des procédures prédéfinies comme "ShowMessage", ou "writeln" pour les manipulations de chaînes.

En ce qui concerne la notion d'argument, nous verrons que celle-ci est plus générale qu'en mathématiques. En particulier, elle permet de paramétrer le calcul comme on le souhaite.

Nous allons commenter une succession d'exemples introduisant progressivement les notions fondamentales (variables globales, variables locales, arguments muets et effectifs, arguments transmis par valeur, arguments transmis par adresse). Nous parlerons des procédures en premier. Les fonctions apparaissent ensuite comme un cas particulier.

a) Les arguments

Lorsque nous avons évoqué la possibilité de paramétrage d'une procédure, nous avons précisé qu'il existait deux façons de le faire en PASCAL. Les variables globales sont des pointeurs ou types qui restent en mémoire. Nous allons maintenant comprendre la notion d'argument.

Voici un nouvel exemple. Il s'agit d'un "Nouveau" "Programme" en ligne de commande. Autrement dit c'est un programme qui n'utilise pas d'interface graphique. Les programmes sans interface graphique consomment moins de mémoire car ils ne font qu'utiliser des caractères pour s'afficher.

```

program exemple_procedure_5 ;
type note = (nut, nre, nmi, nfa, nsol, nla, nsi) ;
var n1 : note ;
    i : integer ;
procedure imprime_note(n : note) ;
    begin
        case n of
            nut : write ('do ') ;
            nre : write ('re ') ;
            nmi : write ('mi ') ;
            nfa : write ('fa ') ;
            nsol : write ('sol ') ;
            nla : write ('la ') ;
            nsi : write ('si ')
        end ;
    end ;
begin
    writeln ('voici les notes de la gamme') ;
    for n1 := nut to nsi do
        imprime_note (n1) ;
    writeln ;
    write ('donnez un numero : ') ;
    readln (i) ;
    write ('la ', i, ' eme note est ') ;
    imprime_note (note(i)) ;
    writeln;
end.

```

Cette procédure affiche en chaine la valeur de type note qu'on lui transmet dans une interface textuelle.

Avant de l'examiner en détail, portons d'abord notre attention sur la définition de la procédure `imprime_note`, et plus particulièrement sur son en-tête:

procedure imprime_note (n : note) ;

Cette fois, celle-ci comporte, en plus du nom de procédure "imprime_note", une indication signifiant que cette procédure possède un argument nommé "n", de type note. Ce type est défini dans le programme principal.

L'argument "n" signifie que lorsque cette procédure est appelée, on lui transmet une valeur de type note. Les instructions de la procédure sont l'utilisation de l'argument, sachant que c'est l'identificateur "n" qui désigne la dite valeur.

Notez bien que "n" n'a aucune signification en dehors de la procédure, ni aucun rapport avec une éventuelle variable globale de même nom. On peut dire que "n" se comporte un peu comme une variable locale à la procédure, avec cette différence importante que sa valeur provient de l'extérieur.

"n" est un argument muet. Autrement dit ce nom en soi n'a aucune importance. La même procédure pourrait être définie en remplaçant "n" par n'importe quel autre identificateur. Ainsi on passe n'importe quel argument de même type sans avoir à attribuer le même nom lorsqu'on appelle la procédure. Si le type est mauvais ou inexact il se produit une erreur de compilation.

Il s'agit là du même principe que lorsque nous définissons une fonction en mathématiques: Nous pouvons indifféremment définir f par :

$f(x) = ax^2 + bx + c$

x est une variable muette

Une fonction ayant le même résultat serait :

$f(u) = au^2 + bu + c$

u est une variable muette

Pour utiliser la procédure "imprime_note" nous faisons suivre son nom

d'une expression de type note entre parenthèses. Par exemple:

```
imprime_note (n1) ;
```

Cette instruction appelle la procédure "imprime_note", en lui transmettant la valeur de "n1".

De même:

```
imprime_note (fa) ;
```

Cette instruction appelle la même procédure en lui transmettant la valeur constante "fa".

Nous utilisons donc le même calcul avec deux variables différentes.

"n1" ou "fa" sont les arguments effectifs de l'appel de la procédure. Ces arguments ne sont plus muets comme dans la définition de la procédure puisqu'ils ne représentent plus quelque chose d'indéfini. Ils ont au contraire une valeur bien précise.

10) LE MOT CLÉ "VAR" DANS LES PARAMÈTRES

On pourrait écrire la fonction "get_note_chaine" comme ceci :

```
procedure set_chaine_note( n : note; var Result : String);
```

Ajoutez le même Code de "get_note_chaine" ensuite.

Le Code Source ainsi créé fonctionne de la même manière mais il est plus rapide à l'exécution.

La seule différence est que l'on déclare une variable de "imprime_note" cette fois-ci.

Voici le Code Source :

```
procedure imprime_note ( n : note) ;  
var chaine : String;  
begin
```

```
set_chaine_note(n, chaine)
writeln ( chaine );
end;
```

On a donc ajouté deux lignes pour le même résultat. Le mot clé `var` nécessite donc d'avoir une variable pré-existante. Préférez l'utilisation de fonctions par défaut.

Le mot clé "var" placé juste avant la déclaration du paramètre va cette fois-ci indiquer trois choses :

- La variable passée en paramètre est modifiable
- La variable passée en paramètre n'est pas recopiée (cf **Programmation procédurale avancée**)
- Une déclaration de la variable doit être créée à l'extérieur de la procédure et peut être utilisée.

a) Exemple

Voici comme exemple un programme textuel montrant l'utilisation d'une fonction calculant la somme des valeurs d'un tableau.

```
program Exemple_procedure ;
const n_max = 10 ;
type tligne = array [1..n_max] of integer ;
var a, b : tligne ;
    i : integer ;
    sa, Sb : integer ;
function somme ( t : tligne ) : integer ;
var i : integer ;
begin
    Result := 0 ;
    for i := low ( t ) to high ( t ) do
```

```

    Result := Result + t [ i ];
end ;
begin
    for i := low ( a ) to high ( a ) do
        begin
            a[i] := i ;
            b[i] := sqr(i) ;
        end ;
    sa := somme ( a ) ;
    Sb := somme ( b ) ;
    writeln ( ' somme des ', n_max, ' premiers entiers : ', sa ) ;
    writeln ( ' somme des ', n_max, ' carrés des premiers entiers : ',
Sb) ;
end.

```

Somme des 10 premiers entiers : 55

Somme des 10 carrés des premiers entiers : 385

Vous constatez tout d'abord que l'en-tête de procédure est devenu un en-tête de fonction:

```

function somme ( t: tligne ): integer;

```

Le terme "procedure" a été remplacé par celui de "function". En ce qui concerne les arguments, vous remarquez que son nom n'apparaît plus dans la liste. En revanche, le mot "integer" a été ajouté à la suite.

Cet en-tête précise, en définitive, le nom de la fonction avec les arguments d'entrée, qui correspondent aux valeurs qui lui seront transmises en argument. Enfin le type du résultat de la fonction est obligatoire lorsqu'on déclare une fonction. Ici le type du résultat est "integer". C'est l'unique résultat qu'une fonction peut fournir.

La fonction est alors réutilisée dans des calculs, des procédures ou d'autres fonctions. Elle peut être passée en paramètre. Elle peut en d'autres termes être manipulée plus facilement.

En ce qui concerne la description de la fonction elle-même, vous constatez la présence d'une variable nommée "Result". Cette variable est la variable locale de toutes les fonctions définie par le type de résultat de la fonction. Dans une procédure, il s'agirait d'un argument nommé Result précédé de var. Elle sert au calcul de la somme désirée.

Il est recommandé de ne jamais déclarer de variable "Result" afin ne pas induire en erreur. Le Code Source doit être clair et documenté.

Enfin ce Code est un Code Source de calcul :

```
Result := Result + t [ i ];
```

Ici on ajoute au résultat de la fonction une ligne pour l'attribuer toujours au résultat de la fonction. Le calcul à droite est d'abord réalisé. Une fois que ce calcul est terminé il est affecté dans la partie gauche de l'instruction de calcul.

Les opérations de calcul et les accès directs aux tableaux ne sont pas optimisés. Cela signifie que l'ordinateur met plus de temps à les exécuter. Nous verrons plus loin comment optimiser les tableaux et opérations de calcul.

b) Le nom de la fonction

Par convention, le nom de la fonction sert à définir la valeur du résultat de la fonction. Ou alors le nom sert à définir le travail effectué. Les noms longs de fonctions n'ajoutent aucunement plus de mémoire à l'exécutable. En effet les noms de variables et de fonctions sont remplacés par des adresses dans l'exécutable.

L'identificateur somme apparaît ainsi à la fois comme identificateur de fonction et comme identificateur de variable. Toutefois, dans ce dernier cas, on ne le voit qu'à gauche d'une instruction d'affectation. En aucun

cas on ne le voit dans une expression. En particulier, nous n'aurions pas pu utiliser directement le nom de fonction "somme". Ce Code est erroné dans la fonction :

```
somme := somme + t [ i ];
```

Dans chaque fonction affectez la variable locale "Result".

F) PROGRAMMATION PROCÉDURALE AVANCÉE

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Dans LAZARUS vous utilisez des Objets afin de créer des Logiciels graphiques complexes. Seulement il est nécessaire de bien connaître la programmation procédurale. Cette programmation sans Objet permet de réaliser un Code Source plus proche du langage machine.

Vous découvrez dans ce chapitre certains aspects peu connus des programmeurs utilisant d'autres langages de développement.

2) CENTRALISATION

Nous avons vu que les fonctions ne renvoyaient qu'un paramètre en sortie. Dans le chapitre précédent une procédure imprimait une note. Modifions la procédure "imprime" afin de créer une fonction pouvant être réutilisée facilement. Transformez le Code afin de procéder à réutilisation de votre Code.

```
function get_note_chaine(n : note) : String;  
  begin  
    Result := "";  
    case n of  
      nut : Result := 'do ' ;  
      nre : Result := 're ' ;  
      nmi : Result := 'mi ' ;  
      nfa : Result := 'fa ' ;
```

```
    nsol : Result := 'sol ' ;
    nla : Result := 'la ' ;
    nsi : Result := 'si ' ;
    end ;
end ;
procedure imprime_note (n : note) ;
begin
  writeln ( get_note_chaine(n));
end;
```

La procédure "imprime_note" exécute exactement la même action que celle montrée précédemment. La seule différence est que cette fois-ci on peut réutiliser la fonction "get_note_chaine" pour envoyer la chaîne autrement.

On pourrait donc réutiliser la fonction "get_note_chaine", pour par exemple l'utiliser en mode graphique. Ainsi on pourrait couper-coller cette fonction dans une unité à part, appelée par exemple "u_fonctions_music", ou "u_fonctions_musique". On pourrait y placer le type "note". Ainsi d'autres applications de musiques pourraient utiliser, de la même manière, ces sources centralisées.

Attention !

Il faudra éviter au maximum de lier une unité, pour la lier de nouveau vers la même unité. Cela voudrait dire que les deux unités ont un lien tellement fort qu'elles sont dépendantes l'une de l'autre. Il n'y aurait presque aucun intérêt à lier récursivement comme cela. Préférez un seul lien entre chaque unité. L'unité la plus élémentaire ne doit pas faire appel à une unité plus complexe.

Nous vous expliquerons, dans le chapitre sur les composants, comment centraliser votre savoir-faire, dans votre regroupement d'unités appelé paquet. LAZARUS automatise la gestion de ses paquets.

a) Exercice

A partir du programme textuel, centralisez les informations en créant une unité de fonctions non redondante permettant de retourner une note en chaîne.

Ensuite transformez l'Application textuelle en programme graphique.

3) OPTIMISATION DES ARGUMENTS

Il est possible d'optimiser la vitesse d'exécution avec des mots clés situés avant les définitions d'argument.

Voyons la ligne de déclaration de la procédure paramétrée :

```
function get_note_chaine(n : note) : String;
```

La procédure d'impression de la note possède un argument. Cet argument va être réutilisé ensuite sans être réécrit . Si vous essayez de changer la valeur de "n" dans la procédure "n" sera toujours de la même valeur après l'utilisation de la fonction.

Pourquoi ?

Ne mettre aucun mot clé avant le paramètre indique au Compilateur de recopier la valeur "n" partiellement ou entièrement dans une deuxième variable. Cela permet de travailler sur "n" sans avoir à recréer la variable.

Seulement dans l'exemple précédent nous ne modifions pas la note "n" dans la procédure. La variable est donc recopiée pour rien.

Mettre le mot clé "const" juste avant la déclaration du paramètre va cette fois-ci indiquer deux choses :

- La variable passée en paramètre n'est pas modifiable
- La variable passée en paramètre n'est pas recopiée

La déclaration de la procédure optimisée idéale est donc :

```
function get_note_chaine ( const n : note) : String;
```

Le Compilateur FREE PASCAL protège les pointeurs, autrement dit les adresses permettant de retrouver les variables, par les mots clés "const" et "var". Lorsqu'on déclare un paramètre par ces deux mots clés on utilise des pointeurs cachés. FREE PASCAL protège votre programme d'une mauvaise utilisation de la mémoire par le mot clé "var".

Il vous permet aussi de savoir si votre pointeur peut ne pas se modifier par le mot clé "const". En effet le mot clé "const" empêche l'écriture d'un type simple ou l'affectation de la variable d'un Objet.

On pourrait utiliser un paramètre "const" en tant que paramètre "var". Cela ne gênerait pas. Seulement cela indiquerait au programmeur que l'on modifie la variable ce qui l'induirait en erreur. Préférez toujours la déclaration avec le mot clé "const" avant de devoir faire une déclaration de fonction, puis créer un deuxième paramètre "var" si votre Code l'exige, sinon laissez tel quel.

4) CRÉER DES SOURCES CORRECTES

Pour créer un programme l'anticipation est nécessaire. C'est pourquoi la centralisation mais aussi la surcharge de Composants permettent de mieux préparer l'avenir. Le bouton que vous avez utilisé dans le premier exemple est un Composant mais aussi un Objet.

L'essentiel est d'avoir une structure du Code Source bien organisée. On doit savoir où se situe le Code Source facilement. L'Objet permet de regrouper le Code Source dans un Composant possédant des propriétés en anglais. Il est préférable de nommer ses fonctions en anglais afin de partager ses Sources plus tard.

Ensuite le Code Source doit être clair, léger, facile à comprendre. Un Code Source clair c'est un Code Source optimisé allant à l'essentiel. Au départ on ne sait pas créer un Code Source optimisé.

Créer un Code Source clair permettra ensuite de créer du Code Source léger. Chaque fonction ou procédure doit tenir sur l'écran. Sinon on passe trop de temps à chercher les erreurs.

Pour être facile à comprendre on doit revenir sur le Code Source créé, le simplifier, et tester. C'est pourquoi mettre en place des jeux de tests automatisés permet de mieux comprendre sa façon de programmer.

a) Exercice

Placez cette Source dans un formulaire manipulant des fichiers. Affichez les résultats dans le Composant TMemo.

Il est possible d'arrêter le programme grâce à LAZARUS. Essayez d'améliorer cela en permettant d'arrêter le programme en cours d'exécution de la routine.

b) Optimisation

Il est encore possible d'optimiser le Code Source précédent. L'optimisation demande de la recherche afin de trouver les meilleures

instructions pour ne pas avoir à réécrire ce qui est déjà fait. Il est intéressant de chronométrer une répétition d'instructions.

Cette instruction peut être optimisée :

```
Result := Result + t [ i ];
```

Quand on trouve du Code Source optimisé sur INTERNET on voit qu'une addition entière peut s'écrire avec l'instruction "inc". C'est une vieille instruction PASCAL qui optimise PASCAL. Elle est donc disponible sur tous les compilateurs PASCAL.

Cette instruction est plus rapide et réalise la même opération :

```
inc ( Result, t [ i ] );
```

Vous voyez que somme n'est utilisé qu'une seule fois au lieu de deux fois. C'est donc une optimisation.

La fonction inc peut être écrite en langage machine grâce au Compilateur FREE PASCAL. Il est intéressant d'optimiser en assembleur certaines fonctions utilisées dans des boucles.

Ceci additionne 1 à somme :

```
inc ( Result );
```

Ceci soustrait 1 à somme :

```
dec ( Result );
```

"dec" est une soustraction entière qui s'utilise de la même manière que "inc".

La division entière n'est quant à elle qu'un simple opérateur de calcul :


```
var resultat, dividende, diviseur : integer;  
begin  
resultat := dividende div diviseur;  
end;
```

Le reste d'une opération entière est perdu. Avec "div" il n'y a pas d'approximation entière.

L'approximation est faite avec la fonction "round" utilisant un réel en paramètre.

```
var resultat, dividende, diviseur : real;  
begin  
resultat := round ( dividende / diviseur, 2 );  
end;
```

Ce programme calcule une division avec une approximation éventuelle à deux chiffres après la virgule.

Voici une autre approximation utilisant "round" :

```
resultat := round ( dividende / diviseur );
```

Si le chiffre est supérieur ou égal à 0,5 celui-ci sera remplacé par 1.
Si le chiffre est égal à 0,49 cela donne 0.

5) OPTIMISER AVEC LES POINTEURS

L'instruction "inc" montrée précédemment n'optimise que des additions sur des entiers. Pour optimiser la lecture d'une chaîne on peut utiliser la procédure "AppendStr", ajoutant à une chaîne une autre chaîne. On peut aussi utiliser les pointeurs sur caractères.

Les pointeurs sont des variables pour lesquelles on peut changer leur place dans la mémoire. En effet l'ordinateur alloue de la mémoire lorsque l'on crée une variable. Il est possible de récupérer l'adresse d'une variable avec la directive d'accès "@".

Les chaînes sont des tableaux de caractères. On peut donc scruter une chaîne comme un tableau.

Les pointeurs, les adresses permettant de retrouver les variables, utilisent une simple addition pour scruter une chaîne ou un tableau. Un pointeur c'est du Code Source proche de la machine. Ce jeu d'instructions met la première lettre en majuscule puis les suivantes en minuscules :

```
Var chaine:string; Pointeur : PChar; i:longint;  
Begin  
  chaine:='eSsai'; // On affecte sa chaîne.  
  Pointeur := @chaine[1]; // Récupère l'adresse du premier caractère de  
la chaîne  
  for i := 0 to length (chaine) -1 do // Scrute toute la chaîne  
  begin  
    if i>0 then // Sommes-nous après le premier caractère  
      // Caractère(s) en minuscule(s)  
      Pointeur^:= LowerChar(Pointeur^)  
    else Pointeur^:= UpperChar(Pointeur^); // En majuscule  
      inc (Pointeur); // Passe au caractère suivant en ajoutant 1 à  
l'adresse  
    end;  
  end;  
End;
```

En testant vous pouvez vérifier la cohésion de votre routine. Le pointeur va récupérer l'adresse du premier caractère de la chaîne grâce à "@".

De même pour récupérer le caractère au lieu de l'adresse mettez un "^" à la fin du pointeur.

Attention !

Le Compilateur ne vous avertit pas d'une erreur d'utilisation sur les pointeurs. Il est donc préférable d'utiliser des jeux de tests pour tester ses routine optimisées.

LAZARUS protège les pointeurs pour éviter les erreurs.

a) Exercice

Refaire le Code Source en non optimisé.

6) RÈGLES GÉNÉRALES

Nous reprenons ici en les généralisant l'ensemble des règles permettant une bonne écriture des procédures ou des fonctions. Celles relatives à leur utilisation seront étudiées dans le paragraphe suivant.

a) Structure générale

Chaque procédure peut être définie ou déclarée dans la partie déclaration entre le mot clé "interface" et "implementation". Cela permet de réutiliser la procédure dans d'autres unités.

Par ailleurs, la portée d'un identificateur est limitée à la procédure où il est défini, ainsi qu'à celles qui sont internes à cette dernière.

Les identificateurs définis dans le programme principal sont connus partout. Cependant sont-ils réutilisables facilement ?

b) L'en-tête et les arguments formels

L'en-tête d'une procédure ou d'une fonction précise:

- son nom
- les arguments formels
- leur type
- leur mode de transmission (par valeur ou par adresse)

De plus, pour les fonctions, il spécifie le type du résultat.

Voici quelques exemples d'en-têtes corrects :

```
procedure ex1(const n, p : integer ; const c : real);  
procedure ex2(const n : integer ; var p : integer ; var x, y : real ; q :  
integer) ;
```

En revanche, cet en-tête compile mais est incorrect :

```
procedure ex3(n : 0..100; var x : real);
```

Le type de n n'est pas explicité dans l'en-tête. il faudrait le définir dans le Code Source l'englobant et placer son identificateur de type à la place de "0..100".

Cette Source ne compile pas :

```
function ex4(x : real ; c : char);
```

Il manque le type de la fonction.

Exercices

- 1) Écrire une procédure permettant de déterminer si un nombre est premier. Elle comportera deux arguments : le nombre à examiner, un indicateur booléen précisant si ce nombre est premier ou non.
- 2) Écrire la procédure précédente sous forme de fonction. (facultatif)
- 3) Écrire une fonction calculant la norme d'un vecteur à 3 composantes réelles.
- 4) Écrire la fonction précédente sous forme d'une procédure.
- 5) Écrire une procédure supprimant tous les espaces d'une chaîne de longueur maximale de 50 caractères.
- 6) Écrire une fonction calculant le produit vectoriel de deux vecteurs à 3 composantes réelles.
- 7) Écrire une procédure simulant le lancé d'un dé à six faces. Utilisez la fonction "random" et la procédure "randomize" initialisant les "random". (facultatif)
- 8) Écrire un programme du jeu de 421 auquel jouent deux joueurs en utilisant un maximum de procédures ou fonctions.

7) *TESTER SON PROGRAMME*

L'ordinateur est tellement différent de l'humain qu'il est obligatoire de tester dès tout changement dans le Code Source. Toute erreur de sémantique peut provoquer des dysfonctionnements dans l'exécutable. C'est pourquoi il est nécessaire d'utiliser une grammaire, dans l'écriture d'un logiciel.

a) Tests d'interface

Les tests d'interface consistent à tester ce qui a été fait en situation réelle après avoir modifié du Code Source. Les erreurs que l'on ne trouve pas dans les tests d'interface doivent être analysées afin de comprendre ce qui a échappé à notre attention. Il faut se méfier de soi lorsqu'on teste.

b) Tests unitaires

Les tests unitaires ce sont une sur-couche à son programme. Cette sur-couche teste automatiquement un programme, méthode par méthode, en remplissant automatiquement le programme. Une sur-couche de tests unitaires n'est jamais liée à un serveur de données. Ce sont les tests de surcharge qui permettent de tester un serveur de données.

Les jeux de tests sont mis en place par un partenaire. Si on les met en place seul, il faut connaître ses défauts et se méfier de soi.

Les jeux de tests utilisent généralement une librairie qui surcharge le Code Source. On peut utiliser FPCUnit, des jeux de tests pour LAZARUS. Mettre en place des jeux de tests consiste à tester les limites de chaque procédure ou fonction.

On teste les valeurs à la bordure des limites voulues. C'est à dire que l'on teste autour de l'extérieur, autour de l'intérieur, au milieu. En même temps on simplifie les procédures et fonctions. Il est préférable de ne pas utiliser l'accès aux données. Cela se fait avec les tests de surcharge. On surcharge les formulaires et Objets utilisés en définissant des valeurs retour à trouver en résultat des procédures et fonctions.

c) Tests de fuites en mémoire

Même en langage JAVA il est conseillé de tester la mémoire. Lorsqu'on exécute un programme il est possible qu'une programmation mal faite laisse des blocs de mémoire inutilisables dans votre zone mémoire.

Ainsi le programme plante, malgré lui, au bout d'un certain temps, parce qu'il n'a plus assez de mémoire viable.

8) DESTRUCTION DE VARIABLES

Le langage PASCAL Objet permet de créer des Objets grâce à ses variables typées. Pour créer du Code Source propre vous devez respecter la syntaxe, ainsi que les règles de destruction de vos variables en mémoire.

Les variables simples comme les nombres, les chaînes de caractères, les tableaux typés, sont détruits, dès que leur Objet ou unité parente est détruite.

Vous ne trouvez pas chez LAZARUS de destruction automatique des variables d'Objets. Seulement les Composants Objets se détruisent dès que leur fenêtre ou module parent est détruit.

On voit que l'auto-destruction si besoin n'est pas aussi optimisée qu'elle le prétendait. LAZARUS permet donc de créer des interfaces hommes machines fiables, si vous informez correctement vos Constructeurs et Destructeurs d'Objets.

9) ***LES OBJETS ET LEUR DESTRUCTION***

Lorsque nous avons créé une interface graphique au tout début du livre nous avons utilisé un Objet "TForm". Cet Objet se détruisait correctement.

Les Objets PASCAL ne sont détruits que sur demande. Si vous utilisez un Objet PASCAL il doit être détruit ensuite si ce n'est pas un Composant. En effet les Composants sont détruits par un Composant parent si tous les paramètres du Constructeur du Composant sont utilisés.

Par exemple l'Objet "TStringlist" permet de gérer des chaînes et des fichiers de chaînes. Ce n'est pas un Composant car il ne possède aucun paramètre lorsqu'on le crée.

```
uses Classes, SysUtils ;

Procedure sauvegarderNombresPremiers ( const i_limite : Int64 ;
const s_CheminFichier : String ) ;
Var stl_Liste : TStringlist;
    t_Entiers : Array of LongWord;
    i, j : LongWord;
    premier : Boolean;
Begin
    stl_Liste := TStringlist.Create;
    setLength ( t_Entiers, 2 );
    t_Entiers [ 0 ] := 1;
    t_Entiers [ 1 ] := 2;
    try
        stl_Liste.Add ( IntToStr ( 1 ));
        stl_Liste.Add ( IntToStr ( 2 ));
        i := 2 ;
```



```

while i <= i_limite do
  begin
    inc ( i );
    premier := True;
    for j := low ( t_Entiers ) + 1 to high ( t_Entiers ) do
      if i mod t_Entiers [ j ] = 0 then
        begin
          premier := False;
          Break;
        end;
      if premier Then
        Begin
          stl_Liste.Add ( IntToStr ( i ));
          stl_Liste.SaveToFile ( s_CheminFichier );
          setLength ( t_Entiers, high ( t_Entiers ) + 2 );
          t_Entiers [ high ( t_Entiers ) ] := i;
        End;
      end;
    finally
      stl_Liste.Free;
    end;
  End;

begin
  sauvegarderNombresPremiers ( 200, '!'+
  DirectorySeparator+'NbPremiers.txt');
end.

```

On définit ici un tableau et un Objet "TStringlist". Le "TStringlist" se détruit par FREE PASCAL lorsqu'on n'en a plus besoin, donc quand on arrête la routine exécutée.

Par contre l'Objet "TStringlist" doit être à tout prix détruit. C'est pourquoi on utilise une gestion des erreurs afin de détruire cet Objet

même s'il y a une erreur, comme une erreur d'enregistrement du fichier. Commencez par faire cela lorsqu'une routine de destruction ou d'édition est mise en place.

Ensuite on écrit la routine qui teste les nombres premiers. Si le nombre premier est trouvé on le sauve dans le fichier en paramètre, puis on l'ajoute dans le tableau de nombres premiers.

On peut utiliser un programme ou une unité de tests de mémoire pour vérifier que sa mémoire est bien utilisée. Cela ralentit le programme.

10) RAPPORT DE FUITES EN MÉMOIRE

Les Objets LAZARUS ne sont pas détruits automatiquement. Il est nécessaire de tester la destruction des Objets, en activant l'unité "Heaptrc".

Allez dans le menu "Projet", puis "Options du Projet", puis "Options de compilation", puis "Édition des liens". Cochez "Utiliser l'unité Heaptrc".

Allez dans votre fichier "lpr" de projet. Ajoutez l'unité "SysUtils".

A la fin de l'exécution de votre projet, placez cette procédure :

```
Application.Run;  
SetHeapTraceOutput (ExtractFilePath (ParamStr (0)) +  
'heaptrclog.trc');  
end.
```

Après avoir exécuté votre projet, vous avez, dans le fichier "heaptrclog.trc" du dossier de l'exécutable, l'ensemble des types ou

Objets ajoutés ou "allocated", et ceux libérés ou "freed", qui doivent être dénombrés identiquement.

Si les types d'Objets enlevés sont moins nombreux que les types ajoutés, vous avez ces types d'Objets non libérés à la fin du fichier de traçage.

Vous pouvez alors placer une autre ligne de traçage, enregistrant dans un autre fichier, là où vous pensez que le type doit se libérer.

Si vous ne trouvez toujours pas, après avoir fait une sauvegarde, supprimez du Code source après les Objets créés, ou dans la personnalisation de vos Objets.

11) TESTER DES ROUTINES

Nous ne vous décrivons pas précisément comment créer des tests unitaires. Mais sachez que les tests unitaires existent. Ils vous permettent de modifier votre logiciel sans vous soucier des erreurs que vous engendrez, puisque vous vous fiez à vos jeux tests.

Pour mettre en place ses jeux de tests, il faut utiliser le projet FPCUnit, fourni avec LAZARUS. Ce projet vous permet de surcharger votre projet initial avec des jeux de tests, demandant un résultat à votre projet. Il faut pouvoir utiliser le projet sans et avec les jeux de tests.

Tester consiste à vérifier les limites de votre routine. Dans la dernière routine on change les limites en entrée pour vérifier le résultat en sortie. Autrement dit il est inutile ici de vérifier m,n,o et p qui se situent au milieu de l'alphabet. Il faut par contre vérifier une fois m, M puis A, puis a, puis z, puis Z, puis 0, puis 9, puis /, puis `, puis é, puis Â.

On vérifie donc une fois une lettre du milieu de la limite, puis les différents contours, puis des lettres en dehors de la zone, puis les fameux caractères accentués. En faisant des tests unitaires, puis des tests de mémoire, de surcharge ou d'optimisation, vous pourrez devenir un programmeur capable de proposer et de participer à LAZARUS grâce à un Code Source maintenu correctement.

Il faudra donc créer un maximum de procédures ou fonctions, afin de faciliter les jeux de tests. Votre exécutable sera non seulement lisible, mais testable. N'hésitez pas non plus à optimiser.

G) CALCULS ET TYPES COMPLEXES

CREATIVE COMMON BY NC-ND

1) LES COMPARAISONS

Les comparaisons servent en général aux instructions d'aiguillage.

La comparaison utilisant l'opérateur "=" se présente comme ceci :

```
A = B
```

A et B doivent être de types identiques.

Une comparaison PASCAL renvoie soit true soit false.

a) Les opérateurs booléens

Opérateur booléen	Définition
or	Renvoie true si la partie gauche ou droite est à true
xor	Renvoie true si la partie gauche ou droite est à true et si elles sont différentes
and	Renvoie true si la partie gauche et droite sont à true
not	Inverse la valeur du booléen

Cette instruction de test ci-après vérifie si B ou C sont égaux à A :

```
A = B or C
```

Le langage PASCAL possède des opérateurs numériques et des opérateurs logiques. Ces opérateurs permettent de comparer au moins tous les types simples.

b) Les opérateurs numériques de comparaison

Comparateur numérique	Définition
=	Renvoie true si la partie gauche est égale à la partie de droite.
≠	Renvoie true si la partie gauche est différente à la partie de droite.
>	Renvoie true si la partie gauche est supérieure numériquement à la partie de droite.
<	Renvoie true si la partie gauche est inférieure numériquement à la partie de droite.
>=	Renvoie true si la partie gauche est supérieure ou égale numériquement à la partie de droite.
<=	Renvoie true si la partie gauche est inférieure ou égale numériquement à la partie de droite.

2) L'INSTRUCTION "IF"

Elle peut prendre une de ces deux formes :

```
If expression_booléenne
then
instruction;
If expression_booléenne then instruction_1
else instruction_2;
```

L'instruction "if" se termine par un ";". Il n'y a pas de ";" avant le "else" de l'instruction if.

L'expression booléenne peut être une comparaison, une fonction booléenne. Elle peut contenir une fonction comparée, etc..

3) ***AFFECTATION***

L'instruction d'affectation est ":=".

```
A:=3; // A prend la valeur 3
B:=A; // B prend la valeur de A, ici 3
```

a) ***Exercice***

Écrire un programme permutant deux variables a et b.

4) ***LES OPÉRATEURS NUMÉRIQUES***

Une instruction d'addition de chaîne ou de nombre est écrite ainsi :

```
var A,B,C : longint;  
begin  
C := A + B;
```

Cette instruction de calcul additionne A à B pour l'affecter à C en utilisant l'opérateur d'addition "+".

Ce genre d'instruction est utilisé pour ajouter des chaînes ou des nombres.

```
var A,B,C : string;  
begin  
A:='Voici '  
B:='une chaîne';  
C := A + B;
```

L'opérateur "+" sert ici à concaténer deux chaînes. C contient "Voici une chaîne" après le calcul.

Si on utilise des nombres voici les opérateurs de calcul :

Opérateur de calcul	Définition
+	Addition
-	Soustrait le nombre de droite à celui de gauche. On peut mettre cette opérateur devant une variable numérique relative pour l'inverser.
*	Multiplication
/	Division de la partie gauche par la partie droite retournant un réel.
div	Renvoie le quotient de la division entière.
mod	Renvoie le reste de la division entière.

Les opérateurs de multiplications et de divisions sont prioritaires sur les

additions et les soustractions :

```
C:=A+B*5;
```

Cette instruction de calcul multiplie B par 5 pour l'additionner à "A". Les parenthèses servent à rendre prioritaire un calcul sur un autre.

```
C:=(A+B)*5;
```

Cette opération est différente. Ici on additionne A à B pour ensuite le multiplier par 5. Le résultat sera donc supérieur ou égal au résultat calculé précédemment.

Voici ci-après une opération utilisant l'opérateur "mod". L'opérateur "mod" ou modulo retourne le reste de la division entière.

```
if ( i mod 2 = 1 ) then
```

Ce test permet de savoir si i est impair.

a) Exemple

Agrandir de 10 Pixels un bouton quand on clique dessus.
Doublé un autre bouton quand on clique dessus.

5) LE TYPE ENREGISTREMENT

Parmi les types structurés dont dispose PASCAL, nous avons déjà étudié le type tableau. Celui-ci nous permettait de réunir, au sein d'une même structure, des éléments de même type ayant un certain rapport entre eux. Cela nous autorisait, au bout du compte, soit à manipuler globalement l'ensemble du tableau dans des affectations, soit à appliquer un même traitement à chacun de ses éléments grâce à la

notion d'indice.

Mais, on peut également souhaiter regrouper au sein d'une même structure des informations n'ayant pas nécessairement toutes le même type. Par exemple, on regroupe les différentes informations relatives à un employé d'une entreprise comme le nom, prénom, sexe, nombre d'enfants, etc.

En PASCAL, le type enregistrement va nous permettre d'y parvenir. Ses différents éléments, nommés alors champs, peuvent être de type quelconque. Par exemple, un enregistrement correspondant à un employé comporte les informations suivantes:

- nom, de type string[30]
- prénom, de type string[20]
- sexe, de type boolean
- nombre d'enfants, de type 0..50

Comme les tableaux, les enregistrements peuvent être manipulés soit globalement, soit élément par élément. En revanche, nous ne retrouvons pas la possibilité qu'offrait le tableau de répéter un même traitement sur les différents champs. Celle-ci n'aurait d'ailleurs en général aucun sens, à partir du moment où les champs sont de nature différente.

Comme nous le verrons dans le prochain chapitre, le type enregistrement est fréquemment associé aux fichiers. Néanmoins, ce n'est pas là une règle générale: il existe des variables de type enregistrement n'ayant pas de rapport avec des fichiers et il existe des fichiers ne faisant pas appel à un type enregistrement. C'est pour cette raison que nous vous proposons d'étudier ici la notion d'enregistrement indépendamment de la notion de fichier.

a) Exemples

Voici une déclaration utilisant le mot clé "record" :

```
type tpersonne = record
  nom : string [30] ;
  prénom : string [20] ;
  masculin : boolean ;
  nbenfants : 0..50
end ;
var employe, courant : tpersonne
```

Cette déclaration correspond à l'exemple évoqué en introduction. Elle définit un type nommé "tpersonne" comme étant formé de 4 champs nommés nom, prénom, masculin et nb_enfants et ayant le type spécifié. Ensuite sont déclarées deux variables nommées "employe" et "courant" comme étant du type "tpersonne".

Ces variables peuvent être manipulées champ par champ, de la même manière que n'importe quelle information ayant le type du champ correspondant. Pour ce faire, un champ d'une variable de type enregistrement est désigné par le nom de la variable, suivi d'un point et du nom du champ concerné.

Par exemple:

"employe.nom" désigne le champ nom de l'enregistrement "employe". Il s'agit donc d'une information de type string[30].

"courant.masculin" désigne le champ masculin de l'enregistrement "courant". Il s'agit donc d'une information de type booléen.

Voici quelques exemples d'instructions faisant référence à certains champs de nos enregistrements "personne" et "courant":

```
readln (employe.nom);  
employe.nbenfants := 3 ;  
if courant.masculin then ...
```

Par ailleurs, les variables employe et courant peuvent être manipulées globalement ici :

```
courant := employe;
```

Cette instruction recopie toutes les valeurs des différents champs de "employe" dans les champs correspondants de "courant". Elle remplace avantageusement les 4 affectations:

```
courant.nom := employe.nom ;  
courant.prenom := employe.prenom ;  
courant.masculin := employe.masculin ;  
courant.nbenfants := employe.nbenfants ;
```

Comme pour les tableaux ces affectations globales ne sont possibles que pour des enregistrements déclarés explicitement du même type.

b) Les types énumérés

Des types plus complexes ont aussi été créés.

On définit en PASCAL des types énumérant des valeurs.

Voici des déclarations d'un type énuméré :

```
type MonEnumeration = (meJeVeux, meJeNeVeuxPas,  
meJeNeSaisPas);  
var VeuxTu : MonEnumeration;  
var Incertitude : set of MonEnumeration;
```

La variable "VeuxTu" utilisant le type énuméré "MonEnumeration" sans le mot clé "set of" ne contient qu'une seule valeur.

La variable "Incertitude" utilisant le type énuméré "MonEnumeration" avec le mot clé "set of" contient plusieurs valeurs.

On commence chaque constante par l'abréviation du type énumération. Cela permet de retrouver facilement les valeurs du type énumération.

Le type énumération sert à se passer des constantes difficiles à retrouver. Il permet aussi de bien affecter ses variables. En effet le Compilateur indique une erreur dans le cas d'une mauvaise affectation de type énuméré.

Une variable énumérée est héritée d'un simple octet. Créer une variable énumérée permet de faire moins d'erreurs de comparaison.

Compilation d'une énumération

Le Compilateur vérifie si l'énumération est correctement utilisée. Une énumération est transformée en entiers allant de 0 à n. Une variable énumérée peut donc être traduite en valeur entière.

Exemple

Affecter la propriété Align du bouton de l'exemple à alClient à la création du formulaire du tout premier exemple.

c) Comparer une variable d'un type énuméré

Les types énumérations peuvent être comparés. Une variable énumérée à valeurs multiples est un tableau de valeurs

énumérées.

Type énuméré à valeur unique

"VeuxTu = meJeVeux" sert à savoir si "VeuxTu" vaut "meJeVeux". Cette opération renvoie true si VeuxTu vaut "meJeVeux".

"VeuxTu in [meJeVeux,meJeNeSaisPas]" sert à savoir si "VeuxTu" contient "meJeVeux" ou "meJeNeSaisPas". Cette opération est égale à "VeuxTu <> meJeNeVeuxPas ».

Type énuméré à valeurs multiples

"meJeveux in Incertitude" sert à savoir si "Incertitude" contient meJeVeux. Cette opération renvoie true ou false.

"[meJeveux,meJeNeVeuxPas,meJeNeSaisPas] * Incertitude = [meJeveux]» sert à savoir si "Incertitude" ne possède que "meJeVeux". Cette opération renvoie true dans ce cas là, false dans tous les autres cas.

Opérateurs de comparaisons et de calculs

Opérateur de calcul	Définition
in	Renvoie "true" si la valeur énumérée est comprise dans le tableau d'énumération.
+	Union de deux type énumérés à valeurs multiples.
-	Complément de deux types énumérés à valeurs multiples.

*	Intersection de deux type énumérés à valeurs multiples.
---	---

6) **LES VARIANTS**

Quelquefois nous ne sommes pas sûr du type que nous voulons utiliser. Lorsqu'on lit un fichier ou des données des informations lues sont à définir. Si celles-ci ne sont pas définies il est possible d'utiliser le type "variant".

Les variants utilisent beaucoup de place en mémoire. Ils sont à choisir en dernier recours ou bien lorsqu'on crée des Composants génériques lisant des types de données inconnues.

```

Uses Variants;
Var v_DonneeInconnue : variant;
procedure p_QuelTypeSuisje ( const Avariant : Variant );
Begin
  If VarIsClear ( Avariant ) Then
    writeln ( 'Mon variant n"est pas affecté.' )
  Else If VarIsNumeric ( Avariant ) Then
    writeln ( 'Mon variant est un nombre.' )
  Else if VarIsStr ( Avariant ) Then
    writeln ( 'Mon variant est une chaine.' )
  Else if VarIsArray ( Avariant ) Then
    writeln ( 'Mon variant est un tableau de variants.' )
  Else
    writeln ( 'Mon variant est d"un autre type.' );
End;
Begin
  p_QuelTypeSuisje (v_DonneeInconnue);

```

```
v_DonneeInconnue := "";
p_QuelTypeSuisje (v_DonneeInconnue);
v_DonneeInconnue := 1;
p_QuelTypeSuisje (v_DonneeInconnue);
v_DonneeInconnue := False;
p_QuelTypeSuisje (v_DonneeInconnue);
v_DonneeInconnue := VarArrayCreate ( [0,2], varVariant );
p_QuelTypeSuisje (v_DonneeInconnue);
v_DonneeInconnue[0] := 'test' ;
p_QuelTypeSuisje (v_DonneeInconnue[0]);
End.
```

VarArrayCreate fonctionne comme ceci :

- Le premier chiffre du tableau d'entiers est la dimension moins 1.
- Le deuxième chiffre c'est la taille de la dimension n.
- Le deuxième paramètre c'est le variant à affecter au tableau.

H) LES BOUCLES

CREATIVE COMMON BY NC-ND

FREE PASCAL permet de répéter à l'infini des instructions de calculs. Les boucles permettent de répéter simplement des instructions afin de procéder à des calculs lourds.

1) LA BOUCLE "FOR"

L'instruction "for" permet de boucler sur une période finie. Le début et la fin sont donc délimités.

a) Syntaxe de l'instruction for

Voici deux écritures de l'instruction "for".

```
for compteur:= debut to fin do instruction;  
for compteur:= fin downto debut do instruction;
```

Une boucle commence au début, c'est la boucle montante. L'autre commence à la fin et est descendante.

Si "debut" est plus grand que "fin" l'instruction des boucles n'est pas exécutée du tout. La boucle n'a alors aucun effet.

"instruction" peut être remplacée par un ensemble d'instructions :

```
for compteur:= debut to fin do  
  Begin  
  instruction1;  
  instruction2;
```

End;

b) La boucle montante

```
program utilisation_de_la_boucle1;  
var n : integer;  
begin  
    for n:=1 to 7 do  
        writeln(n,' a pour triple ',3*n)  
end.
```

Ce programme écrit "1 a pour triple 3", puis "2 a pour multiple 6", ceci jusqu'à 7.

c) La boucle descendante

```
program utilisation_de_la_boucle2;  
var n : integer;  
begin  
    for n:=7 downto 1 do  
        writeln(n,' a pour triple ',3*n)  
end.
```

Ce programme écrit la même chose que l'exemple précédent. Seulement la phrase va démarrer à 7 pour terminer à 1.

d) Recommandations

Prohibez l'affectation de la variable de la boucle. La variable d'une

boucle for ne doit pas être testée en dehors de la boucle.

e) Exercices

Écrire un programme affichant l'alphabet complet. Utilisez les fonctions "ord" et "chr".

2) LA BOUCLE "WHILE"

La boucle "while" permet de boucler sur une période indéterminée après avoir effectué un test. Le test est exécuté à chaque fin de boucle, afin de pouvoir sortir de la boucle.

S'il est impossible de sortir de la boucle, le programme devient incontrôlable.

a) Remarque importante

Pour interrompre un programme sous LAZARUS :

- Retournez sur le menu LAZARUS.
- Allez sur "Exécuter"
- Cliquez sur "Arrêter" voire sur "Réinitialiser le débogueur" si cela ne marche pas.

Il convient donc de tester une boucle sur l'ensemble des ses limites, en créant des procédures ou fonctions permettant de simplifier les tests, en en ajoutant.

b) Syntaxe de l'instruction "while"

```
while expression_booléenne do instruction;
```

Si l'expression booléenne est fausse l'instruction n'est jamais exécutée. L'instruction est exécutée tant que l'expression booléenne est vraie. "instruction" peut ici aussi être remplacée par un ensemble d'instructions. Cet ensemble peut aussi commencer par "begin" pour finir par "end;" en englobant un ensemble d'instructions.

```
Program exemple_boucle_jusqu_a;  
var n : integer;  
begin  
  repeat  
    write('Donner un entier positif : ');  
    readln(n);  
  until n<0;  
  writeln('vous venez de taper un nombre négatif');  
end.
```

Voici un programme à durée infinie si on répond à la demande. On sort du programme quand le nombre donné est négatif.

3) LA BOUCLE "REPEAT"

La boucle "repeat" permet de boucler sur une période indéterminée. Un test est effectué à chaque fin de boucle. On rentre donc toujours au moins une fois dans une boucle "repeat".

a) Syntaxe de l'instruction "repeat"

```
repeat
instruction_1;
instruction_2;
...
instruction_n;
until expression_booléenne ;
```

Exécutez les instructions jusqu'à ce que "expression_booléenne" soit à vrai.

b) Exercices

Créez une fenêtre avec un bouton qui s'agrandit à l'affichage de la fenêtre pour prendre toute la fenêtre.

c) Exercices

Calculez grâce à votre Application la moyenne de notes fournies au clavier avec un dialogue défilant des questions ainsi:

combien de notes ? 4

note 1 : 12

note 2 : 15.25

note 3 : 13.5

note 4 : 8.75

moyenne de ces 4 notes : 12.37

4) ***"CONTINUE" ET "BREAK"***

Il est possible d'écrire cela :

```
While true do  
Begin  
  if expression_booléenne1 Then Break;  
  instructions;  
  if expression_booléenne2 Then Break;  
End;
```

Si " expression_booléenne2" prend la valeur "true" alors la boucle se termine.

Si " expression_booléenne1" prend la valeur "true" alors on passe au nœud suivant sans exécuter les instructions suivantes.

"Break" permet de quitter une boucle. "Continue" annule un nœud.

Vérifiez une boucle sur l'ensemble de ses limites, afin de ne pas planter votre programme.

5) ***LES PROCÉDURES OU FONCTIONS RÉCURSIVES***

Les boucles permettent de gérer des instructions répétitives définies. Il est possible de gérer des instructions répétitives non définies avec les procédures et fonctions récursives.

```
program Project1;
```

```
{ $mode objfpc } { $H+ }
```

```
uses
```

```
{ $IFDEF UNIX } { $IFDEF UseCThreads }
```

```
cthreads,
```

```
{ $ENDIF } { $ENDIF }
```

```
Classes,
```

```
{ you can add units after this }
```

```
Variants, SysUtils;
```

```
{ $IFDEF WINDOWS } { $R project1.rc } { $ENDIF }
```

```
Var MonTableau : Variant;
```

```
Procedure AfficheVariant ( const Avariant : Variant );
```

```
var i : Integer;
```

```
Begin
```

```
  If VarIsClear ( Avariant ) Then
```

```
    write ( 'Clear' )
```

```
  Else If VarIsOrdinal ( Avariant ) Then
```

```
    write ( IntToStr( Avariant ) )
```

```
  Else If VarIsStr ( Avariant ) Then
```

```
    write ( ''' + Avariant + ''' )
```

```
  Else If VarIsArray ( Avariant ) Then
```

```
    Begin
```

```
      write ( '[' );
```

```
      for i := VarArrayLowBound( Avariant, 1 ) to
```

```
VarArrayHighBound ( Avariant, 1 ) do
```

```
        Begin
```

```
          AfficheVariant ( Avariant [ i ] );
```

```
          if i < VarArrayHighBound ( Avariant, 1 ) Then
```

```
            write ( ',' );
```

```
        End;
```

```
      writeln ( ']' );
```

```
    End
```

```
  Else If VarIsFloat ( Avariant ) Then
```

```
    write ( FloatToStr( Avariant ) );  
End;  
Begin  
    MonTableau := VarArrayCreate ( [0,2], varVariant );  
    MonTableau [0] := VarArrayCreate ( [0,2], varVariant );  
    MonTableau [0][1] := 'unechaine1';  
    MonTableau [1] := 'unechaine2';  
    AfficheVariant ( MonTableau );  
End.
```

Ce programme crée un tableau de variant et l'affiche grâce à une fonction récursive, pouvant être centralisée au sein d'un paquet.

Vous voyez que ce programme n'est pas suffisamment modulaire. Créez donc plus de procédures, afin de simplifier sa maintenance.

Ainsi la procédure "AfficheVariant" s'appelle elle-même. Pour ne pas créer une boucle infinie, elle s'appelle en changeant de variant tout en éliminant un niveau de tableau.

I) CRÉER SES PROPRES TYPES

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Pourquoi définir vos propres types ?

Les types prédéfinis permettent de manipuler aisément des nombres ou des caractères. Mais, vous pouvez être amené à traiter d'autres sortes d'informations, par exemple: des mois de l'année (janvier, février...), les jours de la semaine (lundi, mardi...), des notes de musique, des codes couleurs (pourpre, brun van dick...).

Vous pouvez dans ce cas optimiser l'information correspondante, par exemple 1 pour janvier, 2 pour février... ou encore 'A' pour as, 'R' pour roi...

Mais PASCAL vous permet de manipuler ces informations d'une manière plus naturelle en leur attribuant le nom de votre choix tel que janvier ou février. Pour ce faire, il vous suffit de fabriquer ce que l'on appelle un type défini par énumération. On appelle ce type le type énuméré. C'est un type dans lequel vous énumérez chacune des valeurs possibles.

Par ailleurs, il arrive fréquemment que l'on ait à manipuler des informations dont les valeurs ne peuvent couvrir qu'un intervalle restreint de valeurs. Par exemple, un âge pourrait être un entier compris entre 0 et, disons... 150. Une note d'élève peut être un entier compris entre 0 et 20. Une lettre minuscule est un caractère compris entre 'a' et 'z'.

Lorsque l'ensemble des valeurs possibles est ainsi restreint, il peut être intéressant d'en tenir compte dans le programme. PASCAL vous autorise ainsi à définir de nouveaux types comme intervalles d'un autre type scalaire, ce qui vous permet de bénéficier:

- d'une meilleure lisibilité du programme.
- d'un contrôle des valeurs qui seront affectées aux variables de ce type.
- d'un gain de place mémoire lié au fait que, la plupart du temps, les variables d'un type intervalle occupent moins de place que les variables du type initial dit type "hôte" ou type "de base".

a) Recommandations

La déclaration d'un type doit être effectuée avant la déclaration des variables du dit type.

C'est pourquoi les déclarations publiques sont après la clause uses et avant le Code Source exécuté.

2) CRÉER SES TYPES ÉNUMÉRÉS

```
program Exemple_de_type_enumere ;  
type tjour = (jlundi, jmardi, jmercredi, jjeudi, jvendredi, jsamedi,  
jdimanche);  
var jdate : tjour ;  
begin  
  for jdate:= jlundi to jdimanche do  
    begin  
      writeln ('Voici un nouveau jour');  
    end  
end
```

```
if jdate = jmercredi then
  writeln ('Les enfants sont en congé');
if jdate = jvendredi then
  writeln ('Bientôt le week-end !');
if (jdate=jsamedi) or (jdate=jdimanche) then
  writeln ('On se repose' );
end.
```

3) RÈGLES À RESPECTER

```
type nom_du_type = (ndtidentificateur_1,ndtidentificateur_2, .....,
ndtidentificateur_n) ;
```

Il est préférable de mettre, devant chaque identificateur, jusqu'à trois lettres, disposant de l'abréviation du nom de votre type énuméré.

Ne perdez pas de vue les remarques suivantes:

- 1) Un identificateur ne peut être un mot réservé.
- 2) Un même identificateur ne peut pas désigner plusieurs choses différentes.
- 3) Une constante n'est pas un identificateur.
- 4) En PASCAL, tout identificateur doit être déclaré avant d'être utilisé.

a) Un identificateur ne peut être un mot réservé

Ainsi, pour déclarer un type "note de musique", ce type énuméré n'est pas valide :

```
type note = (do, re, mi, fa, sol, la,si);
```

Car do est un mot réservé.

On définit en général un type énuméré en commençant chaque énumération par l'abréviation du type. Aussi un type commence en général par "t".

```
type tnote = (ndo, nre, nmi, nfa, nsol, nla, nsi);
```

On retrouve ainsi facilement chaque note ou le type, grâce à la complétion. La complétion permet de terminer un mot, en allant le chercher dans le programme édité. La complétion s'active avec "Ctrl" et "Espace".

b) Un même identificateur ne peut pas désigner plusieurs choses différentes

Si vous souhaitez définir en plus du type jour déjà rencontré, un type "jour_travail" correspondant aux jours de la semaine, vous songez peut-être à procéder ainsi:

```
type tjour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;  
ttravail = (lundi, mardi, mercredi, jeudi, vendredi) ;
```

Cela serait rejeté par PASCAL puisqu'un identificateur tel que mardi représenterait deux entités de types différents.

Bien entendu, dans votre esprit, le type "ttravail" n'est pas totalement différent du type "tjour". Il est plutôt inclus dedans. Cependant le Compilateur FREE PASCAL ne peut pas le deviner. Le type intervalle permet de venir à bout de ce problème.

c) Une constante n'est pas un identificateur

Il n'est pas question de déclarer un type par une énumération de nombres comme:

```
type timpair = (1, 3, 5, 7, 9, 11);
```

Dans ce cas il est impossible de calculer sans utiliser de conditions alourdissant le programme.

Un type voyelle ne peut se construire ainsi:

```
type tvoyelle = ('a','e','i','o','u','y');
```

Dans ce cas le Compilateur transforme ces voyelles en nombres entiers en débutant par 0. L'utilisateur ne voit pas les voyelles car les constantes définies ont été remplacées.

Le type énuméré est traduisible en chaîne. Cependant une fonction de test de voyelle est plus efficace que le type énumération qui ne fait que compliquer la création de la fonction de test.

Voici une meilleure écriture d'une gestion de voyelles :

```
Program voyelles;  
  
{$mode objfpc}{$SH+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes,  
  SysUtils  
  { you can add units after this };
```

```
{SIFDEF WINDOWS}{$R voyelles.rc}{$ENDIF}
```

```
Function estVoyelle ( const lettre : Char ): Boolean;
```

```
Begin
```

```
  case lowercase(lettre) of
```

```
    'a','e','i','o','u','y': Result := True;
```

```
  else
```

```
    Result := False;
```

```
  End;
```

```
End;
```

```
Var i, les_voyelles :integer;
```

```
  chaine : String;
```

```
Begin
```

```
  writeln('Entrez une chaîne :');
```

```
  readln ( chaine );
```

```
  les_voyelles := 0 ;
```

```
  for i:= 1 to length ( chaine ) do
```

```
    if estVoyelle ( chaine [i] ) then
```

```
      Begin
```

```
        inc ( les_voyelles );
```

```
      End;
```

```
      writeln ( 'La chaîne ' + chaine + ' contient ' + IntToStr  
( les_voyelles ) + ' voyelle(s).');
```

```
End.
```

Ce programme qui écrit la chaîne "machaine" contient 4 voyelle(s)."

Les types PASCAL sont faits pour simplifier la création de type, pas la compliquer.

4) **TYPE INTERVALLE**

Voici, par exemple, comment déclarer un type nommé "tage" dont les valeurs seraient des entiers compris entre 0 et 150 :

```
type tage = 1 .. 150 ;
```

Ou encore, en utilisant une constante définie avant la déclaration de type:

```
const age_max = 150 ;  
...  
type tage = 1 .. age_max ;
```

Il est alors ensuite possible de déclarer des variables du type "tage", par exemple:

```
var age_pere, age_mere, courant : tage ;
```

Notez bien que le type "tage" n'est plus, à proprement parler, un nouveau type. Ses valeurs appartiennent simplement à un intervalle d'un type déjà défini. Ici le type est prédéfini en type entier.

Ainsi, des variables du type "tage" peuvent être manipulées de la même manière que des variables entières, c'est-à-dire être lues, écrites ou intervenir dans des calculs.

Voici des exemples corrects :

```
var n : Integer;  
...  
n := age_pere + age_mere ;  
courant := age_pere + 10 ;
```

Notez bien que "age_pere" + "age_mere" est une expression de type entier. Sa valeur peut dépasser 150 sans que cela ne pose de problème.

Une seule restriction est imposée aux variables du type âge : elles ne peuvent se voir affecter de valeurs sortant de l'intervalle prévu.

Ainsi, en principe, l'affectation "courant := age_pere + 10" entraîne une erreur si la valeur de "age_pere + 10" est supérieure à 150. Il en irait de même si, en réponse à l'instruction "read" précédente, l'utilisateur fournissait une valeur en dehors des limites 0 .. 150.

a) L'exemple

L'exemple précédent définissait un type intervalle à partir d'un type entier. On dit que le type hôte type de base est le type entier. Mais le type hôte peut être un quelconque type ordinal.

Voici deux exemples de type intervalle définis à partir d'un type énumération, à savoir le type jour du paragraphe 2:

```
type tjour=(jlundi, jmardi, jmercredi, jjeudi, jvendredi, jsamedi,
jdimanche) ;
type jour_travail=jlundi .. jvendredi ;
type week_end=jsamedi .. jdimanche ;
```

Ainsi, les valeurs du type "jour_travail" sont les 5 constantes: "jlundi", "jmardi", "jmercredi", "jjeudi" et "jvendredi". Celles du type "week_end" sont les deux constantes "jsamedi" et "jdimanche".

Là encore, les variables déclarées de type "jour_travail" ou "week_end" peuvent être manipulées comme n'importe quelle valeur du type jour. Ainsi on déclare:

```
var aujourd_hui : jour_travail ; courant : jour ;
```

Ces affectations seront correctes:


```
aujourd_hui:= jvendredi ;  
courant:= succ (aujourd_hui) ;
```

Notez bien que "courant" prend la valeur "samedi". En effet, bien que "aujourd_hui" soit de type "jour_travail", le résultat de "succ(aujourd_hui)" est du type hôte "jour_travail", à savoir, du type "jour". Donc "jsamedi" est une valeur acceptable pour cette expression ainsi que pour la variable "courant".

Règle concernant la déclaration d'un type intervalle :

```
type nom_du_type = début_de_l_intervalle..fin_de_l_intervalle;
```

5) ***LE TYPE TABLEAU***

Nous souhaitons déterminer, à partir de 20 notes fournies en donnée, combien d'élèves ont une note supérieure à la moyenne de la classe. Pour parvenir à un tel résultat, nous devons:

- Déterminer la moyenne des 20 notes.
- Déterminer combien, parmi ces 20 notes, sont supérieures à la moyenne précédemment obtenue.

Vous constatez que pour ne pas être obligé de demander deux fois les notes à l'utilisateur, il nous faut les conserver en mémoire. Pour ce faire, il est inefficace de prévoir 20 variables différentes. Cette façon de procéder serait difficilement transposable à un nombre important de notes.

Le type tableau va nous offrir une solution convenable à ce problème, à savoir:

- Par des déclarations appropriées, nous choisissons un identificateur unique pour repérer notre ensemble

tableau de 20 notes.

- Nous pouvons accéder individuellement à chacune des valeurs de ce tableau, en la repérant par un indice ou index précisant la position dans le tableau.

Voici le programme complet résolvant le problème posé:

```
program Exemple_tableau_1 ;
const nb_eleves = 7 ;
type tab_notes = array [1..nb_eleves] of real ;
var notes:tab_notes ;
    i,nombre:integer ;
    somme,moyenne:real ;
begin
  writeln ('donnez vos ',nb_eleves,' notes');
  for i:=1 to nb_eleves do readln(notes[i]);
  somme:= 0.0;
  for i:= 1 to nb_eleves do somme:= somme + notes[i] ;
  moyenne:= somme/nb_eleves;
  nombre:= 0;
  for i:= 1 to nb_eleves do if notes[i]> moyenne then nombre:=
nombre+1;
  writeln('moyenne de ces ', nb_eleves, ' notes', moyenne:8:2 );
  writeln(nombre,' eleves ont plus de cette moyenne');
end.
```

Avec ce programme vous donnez vos 7 notes, par exemple :
11 12.5 13 5 9 13.5 10

Moyenne de ces 7 notes 10.57
4 élèves ont plus de cette moyenne

6) **UTILISATION DU TYPE ARRAY**

Lorsque l'on veut manipuler beaucoup de variables temporaires de même type il est intéressant de créer un ou des types tableaux.

Une variable temporaire sert à créer un résultat. C'est une variable de calcul.

Une variable temporaire peut cependant être persistante. Elle est alors stockée sur le disque dur de l'ordinateur. Dans ce cas on n'utilise pas le type array mais un gestionnaire de données.

a) **Syntaxe**

```
type nom_du_tableau = array [type_intervalle] of type_éléments ;
```

Le type intervalle a été décrit précédemment. Il est décrit comme ceci :

```
debut_intervalle..fin_intervalle
```

"debut_intervalle" et "fin_intervalle" déterminent des entiers ou une partie d'une énumération.

Cas des tableaux à plusieurs indices. Ces deux déclarations sont identiques :

```
array[type_intervalle _1] of [type_intervalle _2] of [type_intervalle  
_2] of type_element  
array[type_intervalle _1, type_intervalle _2, type_intervalle _3 ] of  
type_element
```

Attention !

Créer un tableau à plusieurs dimensions demande beaucoup de mémoire. Il est préférable de vérifier la taille que prend un tableau à plusieurs dimensions.

b) Exercices

- 1) Remplir un tableau avec des nombres aléatoires, puis calculer le maximum et le minimum de ce tableau.
- 2) Même chose que précédemment avec en plus les numéros des cases du tableau où se trouvent le maximum et le minimum.
- 3) Écrire un programme permettant de déterminer les k premiers nombres premiers (1 exclu), la valeur de k étant fixé dans le programme par une instruction const. On conserve les nombres premiers dans un tableau, au fur et à mesure de leur découverte. On utilise la remarque suivante pour décider si un nombre entier est premier : n est premier s'il n'est divisible par aucun nombre premier (1 exclu) inférieur ou égal à la racine carrée de n.
- 4) Réaliser un programme de tri alphabétique de mots fournis au clavier. Le nombre de mots est dans une instruction "const". Chacun d'entre eux ne comporte plus de 20 caractères.
- 5) Réaliser un programme remplissant aléatoirement un tableau 10*10 des 6 nombres ,1 à 6. Puis calculer la moyenne de sortie de chaque nombre.
- 6) Écrire un programme permettant de calculer la multiplication de 2 tableaux de même dimension n*n. Le tableau résultat est créé à partir la multiplication des deux tableaux de base.

c) Syntaxe

Au sein d'une déclaration type ou var, la définition d'un type enregistrement peut se faire comme suit :

```
Type Enregistrement :  
record  
    identificateur1 : description_de type_1 ;  
    identificateur2,identificateur3 : description de type_2 ;  
    identificateur_n : description_de_type_n ;  
end;
```

Notez que l'instruction "end;" termine la description de l'enregistrement. Celle-ci est comparable au contenu d'une instruction var, avec ses listes d'identificateurs et ses descriptions de types. Ces derniers peuvent être aussi bien des identificateurs de type que des descriptions effectives.

Jusqu'ici, nous avons vu qu'un même identificateur ne pouvait être défini qu'une seule fois. Le type enregistrement apporte en quelque sorte une exception logique à cette règle, car il est possible de donner le même nom à des champs de deux types enregistrement différents.

En effet, dans un tel cas, le Compilateur FREE PASCAL est en mesure de lever l'ambiguïté grâce au préfixe nom d'enregistrement, précédant ce nom de champ.

Néanmoins, pour des raisons de clarté des programmes, il est conseillé de ne pas abuser de cette possibilité. Elle peut entraîner quelques ambiguïtés, en cas d'utilisation d'instructions "with" multiples.

Aucune restriction n'est apportée à la nature des différents champs qui peuvent être à leur tour, de type structuré, de sorte que l'on peut très bien définir des enregistrements, comportant eux-mêmes des

enregistrements ou des tableaux. Nous avons déjà fait une remarque analogue, à propos des tableaux. On peut aussi définir des tableaux d'enregistrements.

Examinons dès maintenant un exemple d'enregistrements d'enregistrements. Nous rencontrerons ultérieurement un exemple de tableaux d'enregistrements.

d) Exemples

Supposons qu'à l'intérieur de nos enregistrements de type "tpersonne", nous ayons besoin d'introduire deux dates: la date d'embauche "date_embauche" et la date d'entrée, dans le dernier poste occupé "date_poste".

Ces dates sont elles-mêmes formées de plusieurs champs. "jour_s" contient les jours de la semaine. "jour" contient un jour du mois. "mois" contient un mois. "annee" contient une année.

Nous pourrions effectuer les déclarations suivantes:

```
type tdate = record
  jour_s : (jlun, jmar, jmer, jjeu, jven, jsam, jdim) ;
  jour : 1..31 ;
  mois : 1..12 ;
  annee : longint;
end ;
type tpersonne = record
  nom : string [30] ;
  prenom : string [20] ;
  masculin : boolean ;
  date_embauche : date ;
  date_poste : date ;
```

```
nbenfants : 0..60;  
end ;  
var employe, courant : tpersonne ;
```

Quelques exemples de références à des champs de la variable "employe" vous permettent de comprendre l'utilisation d'un type "record".

"employe.date_embauche.annee" désigne l'année d'embauche de l'enregistrement "employe". Cette information est de type longint.

"employe.date_embauche" désigne la date d'embauche de l'enregistrement "employe". Cette information est de type date.

Cette instruction permet de tester si un employé a été embauché un lundi :

```
if employe.date_embauche.jour_s = lun then
```

7) ***L'INSTRUCTION WITH***

a) **Exemples**

La manipulation globale d'enregistrements, définis par l'instruction "record", n'est possible que dans le cas d'affectations, à condition que l'on manipule tous les champs en même temps.

Supposons que nous souhaitons écrire les informations de l'enregistrement "employe". Nous pourrions procéder ainsi en supposant que "c" est de type char :

```
writeln ('nom : ', employe.nom) ;  
writeln ('prenom : ', employe.prenom) ;  
if employe.masculin then c := 'M'  
                else c := 'F' ;  
writeln ('sexe : ', c, ' nombre enfants' , employe.nbenfants) ;
```

L'instruction "with" nous permet de simplifier les choses:

```
with employe do begin  
    writeln ('nom : ', nom);  
    writeln ('prénom : ', prenom) ;  
    if masculin then c := 'M' else c := 'F';  
    writeln ('sexe : ', c, ' nombre enfants' , nbenfants);  
end;
```

Nous avons pu omettre le nom d'enregistrement à l'intérieur de:

```
with employe do  
    begin  
    ...  
    end;
```

L'instruction "with employe" demande au Compilateur de préfixer, par "employe", tous les identificateurs qui correspondent à un nom de champ, de l'enregistrement "employe". Sa portée est naturellement limitée à l'instruction souvent composée, mentionnée à la suite du "do".

Un identificateur qui ne correspond pas à un nom de champ, de l'enregistrement "employe", n'est pas affecté par cette instruction. Ainsi l'instruction "with employe" va demander à vérifier si une variable utilisée à l'intérieur de l'instruction est définie dans "employe". Si elle n'y est pas, le Compilateur cherche dans les variables locales, puis globales.

b) Imbrication

Avec les déclarations des types "tdate", "tpersonne" et la variable "courant" du paragraphe précédent, nous pouvons écrire ces instructions utilisant deux "with" imbriqués:

```
with courant do begin
  nom := ...
  prenom :=
  with date_emploi do
    begin
      jour_s := ...
      jour := ...
      mois := ...
      annee := ...
    end ;
end ;
```

On obtient le même résultat avec:

```
with courant, date_emploi do
  begin
    nom := ...
    prenom := ...
    jour_s := ...
    jour := ...
    mois := ...
    annee := ...
    ...
  end ;
```

En effet les noms de champ appartenant à "date_emploi" et seulement ceux-là sont préfixés par "date_emploi". Puis, les champs appartenant à "courant", y compris ceux éventuellement préfixés par "date_emploi", sont préfixés par "courant".

Ce mécanisme d'imbrication des "with" risque d'entraîner quelques ambiguïtés, lorsque des noms de champs identiques sont définis au sein d'enregistrements différents.

Dans ce cas le Compilateur FREE PASCAL, qui se trouve confronté à deux préfixes possibles pour un nom de champ, choisit celui qui a été cité en dernier donc celui de niveau le plus imbriqué. Nous vous conseillons d'éviter ce genre de situation, bien compromettante pour la compréhension du programme.

c) Exercices

Écrire deux programmes centralisés, l'un textuel puis l'autre graphique créant un annuaire de 10 personnes, à trier par ordre alphabétique du nom.

J) MA PREMIÈRE APPLICATION

CREATIVE COMMON BY SA

1) A FAIRE AVANT

Avant de créer sa première Application, sachez avant si c'est utile de créer le Logiciel voulu. Vérifiez que ce que vous voulez ne se fait pas déjà. A l'heure actuelle, seuls les Logiciels avec une personnalisation accrue sont à créer.

2) L'EXEMPLE

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

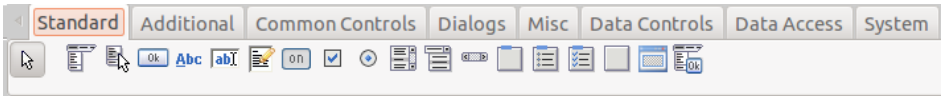
3) CRÉATION DE L'INTERFACE

Un Logiciel doit 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.

4) TESTER SES COMPOSANTS

Il est possible de créer rapidement son Logiciel.

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.



La palette de Composants

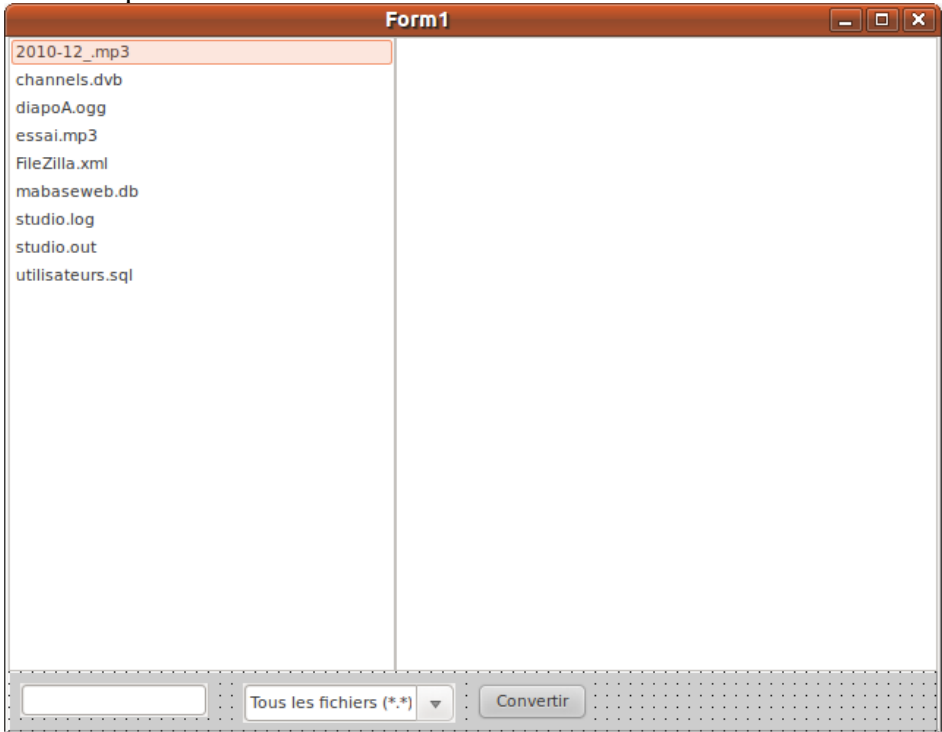
Vous voyez qu'il est possible de placer un certain nombres de Composants, sans limite de présentation.

Certains Composants ne sont 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 ajoutent 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.

5) L'EXEMPLE

Nous allons créer cette interface ci-après. Elle s'adapte automatiquement à sa fenêtre :



Ma première Application

a) Une présentation soignée

Pour commencer notre Interface Homme-Machine, disposez un Composant "TPanel".

Les "TPanel" permettent à l'interface d'utiliser le maximum d'espace de

travail.

Notre premier "TPanel" sert à disposer les boutons. En cliquant sur l'onglet "Standard" vous pouvez ajouter un panneau, en anglais un "panel". Cliquez dessus, puis cliquez sur le formulaire.

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" sert à disposer le Composant de sélection du répertoire. Ajoutez le "TPanel" de la même manière mais ajoutez-le dans la zone vide.

Cliquez sur F11 pour afficher l'Inspecteur d'Objet". Votre "TPanel" y est sélectionné. Cliquez sur la boîte à option de la propriété "Align", puis affectez "alTop" à "Align".

Le troisième "TPanel" affiche 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 propriété "Align" de ce "TPanel" afin de disposer d'un maximum d'espace. Vous pouvez enlever les bordures de vos "TPanel"...

b) L'Interface Homme Machine

Ensuite nous allons disposer les Composants de sélection de répertoire et de fichiers, dans les "TPanel". Nous allons aussi disposer les boutons.

Disposez un Composant "TDirectoryEdit" sur le panneau "TPanel" du haut. Le Composant "TDirectoryEdit" peut sembler insatisfaisant graphiquement. En effet, il n'affiche pas l'arborescence détaillée des répertoires. 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 alors choisissez un Composant de sélection de répertoire plus abouti. Les Composants fournis avec LAZARUS sont testés sur l'ensemble des plates-formes disponibles, en fonction de la "RoadMap" LAZARUS. Consultez-la sur www.lazarus.freepascal.org. Par exemple la "Roadmap" LAZARUS indique en 2011 que des Composants ne marchent pas sur MAC OS.

Disposez un Composant de sélection de fichiers "TFileListBox" dans le panneau du milieu. Vous pouvez lui affecter la valeur "alClient" à sa propriété "Align". Vos fichiers sont disposés sur l'ensemble de l'interface. Ils sont 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 est affecté a une longueur connue. Ils sont alignés à partir de la gauche.

Vous pouvez par exemple créer un bouton qui affiche la date du fichier, son chemin, etc.

Nous allons créer un Logiciel de conversion de fichiers LAZARUS.

Créer un bouton avec le nom "Convertir". Vous voyez que son libellé visuel change. Vous avez affecté aussi la propriété "Caption" du Composant. Changez le nom et son libellé visuel séparément.

Il est nécessaire de relier l'éditeur de répertoire avec la liste de fichier.

Double-cliquez sur l'événement "OnChange" du "TDirectoryEdit".
Placez-y cette Source :

```
procedure TForm1.DirectoryEditChange(Sender: TObject);  
begin  
    FileListBox.Directory := DirectoryEdit.Directory ;  
end;
```

Double-cliquez sur l'événement "OnClick" du bouton.
Placez-y cette 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 d'une chaine du fichier
  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;

```

Vous voyez que cette méthode prend beaucoup de place. Vous pouvez la couper, afin de mieux programmer.

Une fois que vous avez créé la procédure "ConvertitFichier", appuyez simultanément sur "Ctrl", "Shift", avec "C". Ainsi, si la source compile, la procédure "ConvertitFichier" est 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 chaîne, dont une sous-chaîne en paramètre a éventuellement été remplacée. Elle possède des options, comme le remplacement sur toute la chaîne, 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". Faites une copie du projet, avant de le convertir par cette moulinette.

Il n'y a plus qu'à filtrer les fichiers de la liste de fichiers en fonction du résultat demandé par l'utilisateur : Convertir tout ou convertir des fichiers "lfm", des fichiers "dfm" ou des fichiers "pas".

Placez une TFilterComboBox. Renseignez les types de fichiers voulus en renseignant "Filter". S'il n'y a pas d'éditeur de propriété avec le lien vers l'éditeur, placez ce genre de filtre :

```
Tous les fichiers (*.*)|*.lfm|Fichiers DELPHI et LAZARUS|
*.dpr;*.pas;*.dfm;*.lfm|Fichiers projet|.dpr|Unités DELPHI|.pas|
Propriétés DELPHI|.dfm|Propriétés LAZARUS|.lfm
```


On voit qu'avec la version 0.9.28 et sans éditeur de propriété il y a du temps perdu.

Puis renseignez l'événement "OnChange" avec cette Source :

```
procedure TForm1.FilterComboBoxChange(Sender: TObject);
begin
```

```
FileListBox.Mask := FilterComboBox.Mask;  
end;
```

Vous pouvez maintenant vous amuser à changer les Composants de projets LAZARUS, en série.

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.

6) CARACTÈRES ACCENTUÉS

Si vous observez que les caractères accentués ne sont pas gardés, c'est à cause d'un problème de traduction en général. Sinon c'est à cause des caractères ANSI ou UTF8. 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.

Vérifiez si le "TStringlist" utilise des "WideStrings". Ce sont des chaines avec des caractères deux fois plus volumineux.

7) CHERCHER DES 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, définissez ce que vous souhaitez, ainsi que les alternatives possibles. Ce n'est qu'avec l'expérience que l'on trouve des Composants adéquates.

Sur votre moteur de recherche tapez plutôt 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.

8) INSTALLER DES COMPOSANTS

Avant de compiler LAZARUS, Pensez à dupliquer l'exécutable LAZARUS.

Pour installer des Composants décompressez votre archive, puis placez-la dans un dossier qui ne bougera pas.

Avec LAZARUS ouvrez le ou les paquets portant l'extension ".lpk".

Compilez et installez. Ne recompilez LAZARUS que si vous n'avez plus de Composants à ajouter.

Si le paquet ne compile pas cherchez dans le dossier ou sur Internet les paquets manquants et installez les.

9) LAZARUS NE DÉMARRE PLUS

a) Sur LINUX

Si LAZARUS ne redémarre pas alors allez dans votre "dossier personnel". Dans "Affichage" "Montrez les fichiers cachés". Allez dans le dossier ".LAZARUS" puis dans "bin". Effacez "LAZARUS". Si vous ne trouvez pas l'exécutable LAZARUS, réinstallez le paquet "LAZARUS-ide".

b) Sur WINDOWS

Effacez le fichier "LAZARUS.exe". Il est soit dans "Documents and Settings" ou "Users", puis dans votre compte utilisateur, puis dans "Local Settings", "Application Data" et "LAZARUS".

Ou bien il est dans "[C:\LAZARUS](#)". Dans ce cas recopiez l'exécutable LAZARUS ou réinstallez LAZARUS.

10) VÉRIFIER LES LICENCES

Après avoir téléchargé vérifiez l'utilisation des licences. Chaque Composant possède 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 :


- Pas de Sources. Vos composants s'installent sur certains LAZARUS. Vous n'avez pas la possibilité de reprendre le projet s'il est abandonné.
- Aucun fichier de licence : En général vous faites ce que vous voulez. Vous devez contacter l'auteur pour définir une licence.
- GPL : Cette licence vous oblige à diffuser les Sources de votre Logiciel ou descendant, si vous diffusez le Composant, ceci en

annonçant l'auteur.

- Creative Common by : Cette licence vous oblige à annoncer l'auteur du Composant.
- BSD : Cette licence entièrement Libre vous autorise à revendre le Composant, comme vous le voulez.
- Etc.

Une licence commerciale, pour laquelle vous avez acheté des Composants, peut disposer d'un ensemble de restrictions à lire.

11) 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. Aussi il utilise les bibliothèques 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 exécutez 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" passe certaines Sources.

L'utilisation des touches "F7" et "F8" est fastidieuse. Vous pouvez ajouter un voire plusieurs points d'arrêts, en cliquant dans votre "Éditeur de Source", sur la marge grisée d'une ou de plusieurs lignes à vérifier.

Le Compilateur FREE PASCAL crée des fichiers exécutables volumineux. Pourquoi ?

Parce que LAZARUS ne déporte pas l'utilisation des bibliothèques de chaque environnement. Autrement dit votre exécutable peut ne pas utiliser les bibliothèques que vous avez utilisées pour créer l'exécutable. Un exécutable LAZARUS a de très grandes chances de fonctionner seul. Aussi LAZARUS, en version 0.9.28, peut ajouter des unités, liées par les paquets, que vous n'utilisez pas dans votre exécutable.

12) 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 dans celui servant à l'utilisateur. En général les erreurs de développement doivent être en anglais, tandis que les erreurs de l'utilisateur sont multilingues.

Anticipez ce qui va se passer dans votre programme. Imaginez un utilisateur tombant sur une erreur en anglais, alors qu'il ne connaît pas cette langue. En plus, est nécessaire un manuel de l'utilisateur, permettant de trouver les informations nécessaires à l'utilisation du Logiciel.

13) 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 mon 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
        Showmessage ( 'La chaine de mon formulaire est trop longue. Il
n'y a plus assez de mémoire.' );
    On E:Exception do
        Showmessage ('Erreur avec la chaine. Le résultat peut être
tronqué. ');
end;
```

L'instruction de l'exception " E:EOutOfMemory" est appelée quand il n'y a plus assez de mémoire. Cette exception se produit en général avec les manipulations de tableaux à longueur variable.

L'instruction de l'exception "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.

a) Renvoyer les exceptions

Au départ les erreurs peuvent s'adresser à soi. Mais à force d'utiliser son Logiciel on redéfinit 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 renvoyer une erreur vers l'utilisateur.

```
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.

Pratiquer la veille technologique permet de voir si son erreur n'a pas été ajoutée, avec une nouvelle version de son kit de développement.

K) L'OBJET

CREATIVE COMMON BY SA

1) *INTRODUCTION*

La programmation orientée Objets permet, dans LAZARUS, de réaliser vos propres Composants, votre savoir-faire. Avec l'Objet, elle permet aussi de réaliser des Logiciels complexes. Ce chapitre et les deux chapitres suivants sont complémentaires. Ils doivent être bien compris afin de surpasser les capacités du programmeur.

Dans ce chapitre nous allons définir ce qu'est l'Objet. Cela va permettre de mieux comprendre LAZARUS, donc de sublimer cet outil par l'automatisation RAD. LAZARUS est un outil de Développement Rapide d'Applications, aboutissement de l'utilisation de la programmation orientée Objets.

Si vous ne connaissez pas l'Objet vos Composants sont 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. Étouffez vos connaissances avec des livres sur l'analyse Objet, analyse proche de l'humain. Elle nécessite du travail et de la technique. Nous parlons succinctement des relations en Objet dans le chapitre sur les **Logiciels centralisés**.

Chaque Compilateur possède des spécificités ou améliorations, pour faciliter le travail du développeur. Nous vous les expliquons.

2) 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 "Touche" 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".

3) 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, en créant un formulaire hérité de la classe "TForm". 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 avec la possibilité d'être dupliqué.

On peut remarquer que FREE PASCAL définit une classe comme un

Objet (type "classe") ou comme un enregistrement (type "record"). L'enregistrement record ne permet que de définir des méthodes. La classe est l'Objet analytique pouvant agir autour de lui.

Une classe accepte l'instruction "with", Vous pouvez utiliser "with" en prenant garde, toutefois, aux variables ou méthodes identiques, dans des entités différentes.

4) 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. Elle est aussi personnalisée grâce à ses variables.

Si votre Classe d'Objet et ses Classes parentes ne contiennent aucune variable, il est nécessaire de se poser la question de leur utilité. Puis-je gagner du temps en créant une unité de fonctions à la place ?

Vous créez 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.

a) Les attributs et propriétés

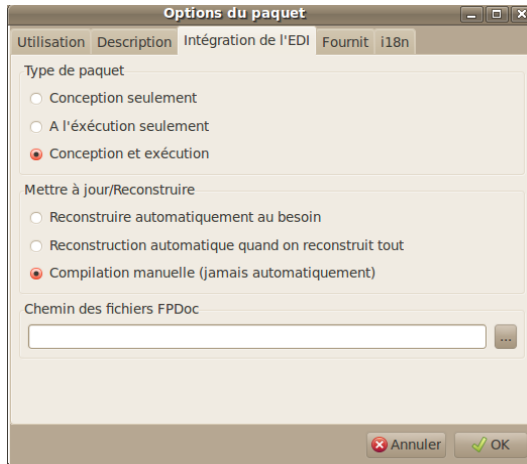
Deux Objets "Clavier" peuvent être faits de la même manière, avec une couleur différente. Ce sont toujours deux Objets "Clavier" avec un attribut différent. Un attribut peut être une couleur, un nombre, du texte, ou bien un autre Objet.

Seulement en FREE PASCAL les attributs, nommés variables, peuvent être enregistrés dans l'EDI comme des propriétés. Ces variables sont alors sauvées dans l'Objet "TForm", visible en exécution, ou le "TDataModule" invisible. Dès que vous modifiez un formulaire LAZARUS vous utilisez l'Objet. Seulement ces Objets n'ont pas besoin de l'EDI, pour charger leurs propriétés à l'exécution. Une partie du Code Source de développement est exécuté.

Au début du livre un formulaire possédait des propriétés permettant d'automatiser facilement le développement. Les propriétés manipulables dans 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 d'édition de propriétés, servant au développement du Code Source exécuté. Les Objets, éditant les propriétés dans l'"Inspecteur d'Objets", ne sont pas incluses dans votre exécutable. Lorsque vous concevrez votre premier Composant, il faudra aussi séparer le Code Source de développement du Code Source exécuté.

Vous ne devez jamais lier dans vos projets directement vers une unité d'enregistrement de Composants, puisqu'elle ne sert qu'à LAZARUS. C'est pour cette raison que des unités, spécifiques à l'enregistrement des Composants, sont à part des unités de Composants.



Le Code Source exécuté peut être séparé de la conception

b) Un attribut dans LAZARUS

Dans les classes LAZARUS, les attributs se déclarent avant les méthodes. Une méthode est une fonction ou procédure au sein d'une classe.

```
type  
  TForm1 = class ( TForm )  
    MonBouton : TButton ;  
  public  
    constructor Create ( AOwner : TComponent ) ;  
  end;
```

L'attribut "MonBouton" crée un Objet bouton. L'attribut du type bouton sert à créer un bouton, dans le formulaire TForm1.

c) Une propriété dans LAZARUS

type

```
TForm1 = class ( TForm )  
  MonBouton : TButton ;  
  procedure OnClickMonBouton ( Sender : TObject ) ;  
  private  
    FClickBouton : Boolean ;  
  public  
    constructor Create ( AOwner : TComponent ) ;  
    property ClickBouton : Boolean read FClickBouton write  
FClickBouton default false;  
  end;
```

L'événement "OnClick" du bouton "MonBouton" fait appel à la méthode "OnClickMonBouton".

L'attribut privé "FClickBouton" ne peut être lu dans son descendant. Ainsi on accède à cet attribut par sa propriété publique. Ainsi on peut transformer la fiche en Composant plus facilement.

La propriété permet d'accéder aux attributs directement ou indirectement. Vous pourriez créer dans la propriété une fonction appelée getter, et une procédure appelée setter, permettant de préparer l'attribut avant de l'utiliser.

La propriété "ClickBouton" permet ici de ne pas avoir à définir de getter et setter afin d'accéder au clique de bouton. Ainsi un getter ou setter peuvent être ajoutés ensuite à la propriété. Le Code Source n'est que très peu changé.

On vous montre les différentes options de déclaration de propriétés. FClickBouton doit être initialisé à "false", dans le Constructeur de la

fiche. En effet, la déclaration "property" ne permet pas d'initialiser de valeur par défaut. Une déclaration n'agit pas en général.

d) 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 fonction centralisée dans un Objet. Une méthode modifie les attributs ou variables du programme.

Seulement en FREE PASCAL il y a plus adaptable que la méthode. L'événement permet de renseigner rapidement tout Objet. Ainsi on ne crée pas l'Objet "Clavier".

LAZARUS va facilement propager des événements dans le formulaire de votre projet, puis dans chaque Objet de formulaire. Chaque Objet possède alors une méthode captant le clavier. Vous pouvez renseigner les différents événements clavier dans l'"Inspecteur d'Objets". Ces événements sont accessibles aussi, au sein des différents Objets, grâce à des méthodes.

L'événement consiste à transformer une méthode en variable, pour pouvoir, comme les propriétés, transférer facilement un événement à un autre Objet.

5) LES COMPORTEMENTS DE L'OBJET

L'Objet est, contrairement aux langages procéduraux, plus proche de

l'homme que de la machine. Il permet de réaliser des systèmes humains plus facilement. L'Objet étant plus proche des représentations humaines il complexifie 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.

6) ***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 classe "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 du 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. Cependant tout Composant LAZARUS est hérité du Composant le plus proche de son résultat demandé. On peut donc hériter votre formulaire du Composant "TCustomForm" si certaines propriétés gênent.

Vous pouvez regarder 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". Certaines variables, comme les Objets, sont 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".

7) 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". Cette méthode surchargée contient le comportement spécifique de toutes les fiches que l'on utilise. Ainsi, dans un Composant héritant de "TForm", on peut automatiser la fermeture de ses formulaires. Il y a moins de Code puisque la fermeture est dans un seul Objet.

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, ajoutez le mot clé "virtual". Si vous scrutez la toute première déclaration de la méthode DoClose vous trouvez ainsi cette Source dans la première déclaration de DoClose :

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

On peut aussi utiliser cette déclaration :

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

Il est préférable d'utiliser la déclaration "virtual". En effet cette déclaration favorise la vitesse, plus importante que la taille en mémoire avec "dynamic".

Il existe d'autres moyens pour gagner de l'espace. Nous vous les citons dans le chapitre sur les Composants.

b) Le mot clé "override"

Pour surcharger une méthode virtuelle, ajoutez cette Source dans une classe descendante. Vous pouvez par exemple ajouter ce Code Source 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. cette 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 toujours d'utiliser le mot clé "inherited", lorsque vous surchargez votre méthode. Sinon votre programme boucle à l'infini.

Vous pourriez créer un Composant hérité de "TForm", avec vos options de fermeture de formulaires. Ainsi du Code Source ne serait plus dupliqué.

8) 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, dans une nouvelle unité, qui réutilise la fonction ou procédure. Seulement vous utilisez le nom de l'unité pour distinguer les deux fonctions ou procédures.

On pourrait ainsi appeler la fonction ou procédure imbriquée comme l'original. On pourrait donc se tromper facilement de fonction ou procédure identique, en éludant une unité. L'Encapsulation en PASCAL non Objet est donc approximative. 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 bibliothèques, l'Encapsulation est améliorée avec FREE PASCAL en 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. On peut hériter du Composant au-dessus, si l'on veut utiliser la nouvelle gestion, éludant l'ancienne, par le nouveau Composant.

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 Open Source d'un formulaire est consultable, par simple clic de souris, grâce à l'inspecteur d'Objet.

Une variable ne peut pas 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, demandez-vous s'il est utile de créer un nouveau Composant, héritant d'un des Composants ascendants. En effet les variables fantômes sont en mémoire, en partie, dans chaque instance de classe.

a) 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.

Allez sur la déclaration de "TForm" :

```
TCustomForm = class(TScrollingWinControl)
private
  FActiveControl: TWinControl;
  procedure SetActiveControl(AWinControl: TWinControl);
published
  property ActiveControl: TWinControl read FActiveControl write
```

```
SetActiveControl;  
end;
```

Vous voyez, dans les Sources de LAZARUS, que beaucoup de variables sont déclarées en privées. Elles sont tout de même accessibles dans l' "Inspecteur d'Objets", sous un autre nom. Souvent, il suffit d'enlever la première lettre de la variable, pour la retrouver dans l'"Inspecteur d'Objets". Des déclarations privées sont utilisées grâce aux getters et setters, eux aussi privés, afin d'être visibles dans l'"Inspecteur d'Objets".

La déclaration "private" permet de regrouper les variables, avec les "getters" et les "setters" des propriétés créées. Un "setter" est une affectation d'une variable, contenant généralement son nom, initialisant une partie du Composant. Le "getter" fait de même, en lisant cette fois la variable privée, contenue dans son libellé.

Ainsi le programmeur ne se trompe pas lorsqu'il réutilise la variable. Il utilise la propriété déclarée par "property", définissant d'éventuels "getters" et "setters", permettant d'initialiser le Composant en affectant la variable. Vous voyez des propriétés dans les Composants LAZARUS.

b) La déclaration "protected"

Une méthode "protected" ne peut être surchargée que dans l'unité de l'Objet, vers les Objets hérités. Surcharger un Composant permet donc d'accéder aux méthodes ou variables protégées, afin de modifier le comportement d'un Composant.

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 non hérités, en dehors de son unité.

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 comment mettre une méthode dans la zone "private" ou "protected" il est préférable de placer sa méthode dans la zone "protected".

c) La déclaration "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 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 en "protected". Le développeur de Composants modifie les facultés de votre Composant.

d) 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, accessible indépendamment de la classe Objet qui l'utilise.

Voici comment récupérer une méthode publiée sans connaître sa classe :

```
// 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, Elle est à placer au début de l'unité

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

Cette Source permet d'exécuter tout événement "OnClick" de tout Composant, s'il est publié.

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 permet d'affecter des paramètres à une méthode.

Nous avons déjà utilisé les événements dans un chapitre précédent. Un événement est une définition en variable d'une méthode. Il est cependant possible d'accéder à une méthode publiée, sans avoir à utiliser l'événement, en utilisant le type "TMethod". Ce type permet d'accéder à une méthode publiée dans un Composant.

```
{ $Mode DELPHI }  
var lmet_MethodeDistribueeSearch: TMethod;  
Begin  
  lmet_MethodeDistribueeSearch.Data := Self ;  
  lmet_MethodeDistribueeSearch.Code :=  
  MethodAddress('MaMethodePubliee') ;  
  ( lmet_MethodeDistribueeSearch as MonTypeMethod )  
  ( monparametre ) ;  
End;
```

Si ce que vous avez défini n'existe pas déjà on définit un type méthode ainsi :

```
TMaMethode = procedure(const monparametre:TypeParametre)  
  of object;
```

e) Les propriétés publiées

Lorsqu'on publie une propriété dans un Composant référencé celle-ci se retrouve dans l'inspecteur d'Objet.

Il est préférable de créer une propriété au plus proche de l'Objet, en surcharge, voire en participation Libre, si la modification est utile pour la communauté.

Voici un bouton amélioré pour une utilisation particulière :

```
type  
  TMyButton = class ( TSpeedButton )  
  private  
    FClickBouton : Boolean ;  
    function getClickBouton:Boolean;  
  public  
    procedure Click ; override ;  
  published  
    property ClickButton : Boolean read getClickBouton write  
FClickBouton default false ;  
    property Glyph stored false ;  
  end;
```

La propriété "ClickButton" permet de savoir si le bouton a été cliqué. Le getter getClickBouton permet de réinitialiser FClickBouton à la

lecture du clique.

On empêche ici d'enregistrer l'image du bouton, en surchargeant la déclaration publiée du "TSpeedButton". Cela permet d'empêcher d'enregistrer dans chaque fiche l'image qu'on aurait chargée. Nous vous expliquons cette optimisation dans le chapitre suivant.

L'option "stored" à "false" n'enregistre pas la propriété dans l'"Inspecteur d'Objets", donc dans l'exécutable. "stored", qui est à "true" par défaut, permet l'enregistrement dans l'"Inspecteur d'Objets", si votre Composant est dans la palette des Composants.

f) 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.

Un descendant de TComponent détruit automatiquement les Objets de type 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.

Tout descendant de TComponent peut être enregistré dans l'"Inspecteur d'Objets".

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". Le Constructeur initialise le "TStringList". Il peut alors être créé dès qu'on l'utilise, grâce à une méthode centralisée.

Le Destructeur du Composant libère les Objets mis en mémoire. 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".

9) ***LE POLYMORPHISME***

Le terme Polymorphisme est certainement le plus redouté. Pour le comprendre, voici la sémantique du mot : "poly" signifie plusieurs et "morphisme" signifie 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. 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.

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.

Attention !

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.

L'Héritage multiple possède lui aussi ses défauts. Il est encore plus lourd en mémoire que le Polymorphisme en orienté Objet.

a) L'Abstraction

L'Abstraction ne sert, en FREE PASCAL, 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 dans une classe d'Objet :

```
procedure Nager; abstract;
```

Cette déclaration est la seule à ne pas demander d'implémentation. Elle crée une erreur malgré tout, à l'exécution, 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 horizontale. 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. On peut définir le type de nage grâce à un nombre en paramètre.

b) Le type "interface"

Le type "interface" est le Polymorphisme en FREE PASCAL en son

essence.

Il est impossible d'hériter de deux classes voire plus. Utilisez le type "interface", pour remédier à ce problème. Une classe FREE PASCAL hérite d'une classe, avec, éventuellement, 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 crée une classe. Cependant il n'est possible que d'y ajouter des méthodes abstraites.

Nous parlons du type "interface" dans le chapitre suivant.

c) 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" permettent 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. Changez alors le type de votre Composant, à la fois dans le fichier ".pas" de votre fichier, 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 de sa 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 sur le bouton 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 alors 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, avant de créer une nouvelle propriété LAZARUS, il est nécessaire de connaître les propriétés standards de tout Objet LAZARUS. Une propriété LAZARUS est en anglais.

Les types Objets "interface" ne permettent pas de déclarer de véritables Objets. Elles ne font que centraliser la déclaration de méthodes. 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. Elles permettent cependant une hiérarchisation de vos Sources, grâce aux paquets.

10) 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 de conception, en étant présentes dans l'"Inspecteur d'Objets".

Il ne faut pas être créatif pour nommer ses propriétés. Reprenez 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 est donc nécessaire de connaître les noms des propriétés des Composants de même nature, afin de créer des propriétés identiques.

11) 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. Vérifiez cependant si c'est bien le cas.

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, participez au projet de traduction, qui peut être intégré au projet STAR UML.

L) CRÉER SON SAVOIR-FAIRE

CREATIVE COMMON BY SA

1) INTRODUCTION

Pour créer un savoir-faire respectez 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 à créer ou aucun

2) CRÉER DES UNITÉS DE FONCTIONS

Vous pouvez vous aussi créer votre savoir-faire en créant d'abord des projets. Pour créer ces projets, nous vous apprenons à centraliser ce qui aurait été redondant. Évitez le copier-coller. Vous créez des unités de fonctions en centralisant un Code redondant.

Une unité de fonction 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.

3) **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.

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. La librairie contient en plus des outils préparant le travail.

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.

Vos unités de fonctions peuvent ensuite améliorer des Composants par le procédé de l'Héritage, ou par la participation Open Source. Vous créez alors votre premier paquet, en apprenant l'Objet. Vos unités de fonctions sont utilisées et améliorées.

Vous participez alors à un projet Open Source après avoir vérifié la licence Open Source. Si la participation au projet est refusée il est nécessaire de s'interpeller sur l'utilité de cette participation. L'entreprise qui refuse la participation peut avoir d'autres objectifs que les vôtres. Vous pouvez alors surcharger le Composant.

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 au formulaire ouvert. Elles sont et doivent donc être situées 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 que vos ou d'autres Composants en héritent.

4) DÉVELOPPEMENT TRÈS RAPIDE D'APPLICATIONS

Le Développement Très Rapide d'Applications c'est l'aboutissement de tout savoir-faire RAD. C'est l'utilisation de la Programmation Objet afin de séparer, ce qui est demandé par le client, du Logiciel en lui-même.

On supprime alors une étape de programmation, à savoir la transcription de l'analyse en Logiciel, car celle-ci est automatisée par la Programmation Objet, et les fichiers passifs. Les fichiers passifs contenant une analyse du Logiciel sont lus par le moteur DTRA, afin de créer le Logiciel.

Un Logiciel créé en Développement Très Rapide est toujours conforme à l'analyse, car l'analyse crée le Logiciel.

On crée des unités de fonctions, puis des Composants regroupés en paquets LAZARUS, puis une voire plusieurs librairies. Ces librairies sont indépendantes de LAZARUS, en étant entièrement automatisées grâce à des fichiers.

L'aboutissement est la prise en compte de ce qui est demandé par les clients, en créant des fichiers automatisant la librairie. Ces fichiers sont dédiés à ce qui est demandé. Ce sont les fichiers métiers. Ces fichiers peuvent être automatiquement créés par l'analyste.

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.

5) INTÉRÊTS DU DTRA

Mis à part le gain de temps, supprimer une étape de programmation permet d'améliorer aussi la qualité des Logiciels créés. L'analyse correspond toujours au Logiciel, puisque l'analyse crée le Logiciel. Les jeux de tests sont mis en place pour le moteur DTRA uniquement.

Même si un moteur DTRA nécessite au moins un ingénieur développeur, ce dernier pense fonctionnalités et pérennité du système.

Les fichiers passifs peuvent être lus par d'autres moteurs. Il est possible de créer des Forks de LEONARDI sur d'autres langages, grâce à des Frameworks de gestion, avec fiches et relations.

6) CRÉER UN FRAMEWORK DTRA

LEONARDI est un Framework ou savoir-faire DTRA JAVA, permettant de créer des Logiciels de gestion avec fiabilité.

LEONARDI utilise des fichiers de description de ses méta-données, ces données servant à définir comment on utilise une quelconque donnée,

afin d'homogénéiser les logiciels, pour pouvoir les créer plus vite.

Il existe aussi les modèles de méta-données MICROSOFT, permettant de décrire encore plus d'entités de programmation.

Il existe un Framework ou savoir-faire DTRA LAZARUS Libre nommé XML Frames. C'est un savoir-faire Client/Serveur, qui crée des applications de gestion sur WINDOWS, GNOME, ou KDE.

Il est possible de créer son Framework Web de Développement Très Rapide, si la création de son Logiciel est répétitive. Ainsi l'automatisation permet de gagner du temps, ensuite.

Un Logiciel de gestion c'est un Logiciel gérant des processus. Il est composé d'un lien vers les données, de fiches, de relations, de filtres, de statistiques, de feuilles de calcul, etc. Il est simple à modéliser donc simple à automatiser.

Les Composants ou plugins permettent de créer une partie du Logiciel selon les spécificités demandées. Ces Composants ou plugins permettent d'étendre les capacités du moteur. Ils nécessitent une approche à la fois élémentaire, pour répondre à une partie infime de la demande, puis globale pour être inclus dans toute analyse. Ils sont utilisés dans l'EDI DTRA.

Il suffit alors d'utiliser une architecture pilotée par modèle, afin de centraliser ce genre de Logiciel dans un moteur contenant les Composants. Les Composants sont renseignés par les modèles analytiques, pouvant lire toute analyse. Les Logiciels, créés sans le moteur DTRA, peuvent uniquement renseigner le Composant formulaire.

Ainsi il y a au moins deux couches dans un moteur DTRA. Une fois cette étape passée les Logiciels peuvent être en partie traduits, par le

reverse engineering traduisant les données en un début de Logiciel. Vous créez alors la partie dynamique, non incluse dans les données.

On peut créer une troisième couche, qui va permettre d'analyser les Logiciels à mettre en place.

Il est donc possible de créer des plugins, dans les Frameworks Rails ou tout Framework de gestion, afin de permettre leur automatisation.

7) CRÉER UN COMPOSANT

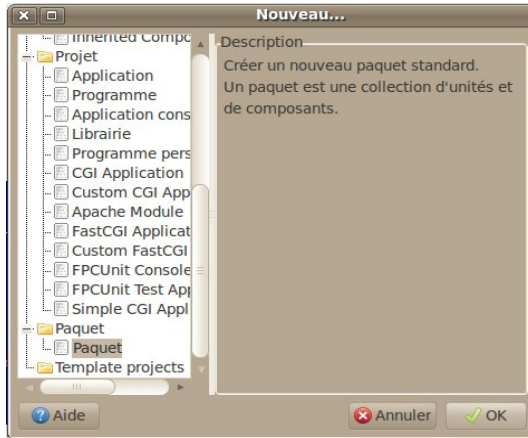
Nous allons changer la couleur d'un bouton de fermeture, et lui affecter une image.

Pour tout bouton créé, sera affectée une seule image. Ainsi l'exécutable est plus léger. Ce bouton permet 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 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 Faire "Fichier" puis "Nouveau" puis "Paquet".



Créer un nouveau paquet dans l'"EDI LAZARUS"

Nommez le paquet "LightButtons".

"Ajoutez" la condition "LCL". Une condition est un paquet utilisé. Le paquet LCL contient tous les Composants standards.

"Ajoutez" un "Nouveau fichier" "Unité", puis 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 de paquet. Affectez lui le nom "LazBoutonsPersonnalisés".

Ajoutez ses deux types dans la partie interface de votre unité :

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

On utilise le mode DELPHI afin de ne pas avoir à gérer les adresses de pointeurs. Le mode DELPHI est expliqué dans le chapitre suivant.

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

En FREE PASCAL on définit 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;
```

TMyClose est le descendant de TBitBtn. Il supporte l'interface IMyButton.

Créer la clause uses dans la partie "interface". Ajoutez ensuite les noms d'unités "StdCtrls, Classes, lresources".

L'unité "StdCtrls" contient les boutons. L'unité "Classes" contient le type composant. L'unité "lresources" vous permettra d'ajouter les images à vos boutons.

Le Constructeur Create et la méthode Click sont présents 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. Vous pouvez donc facilement changer de type Bouton par la classe que vous créez.

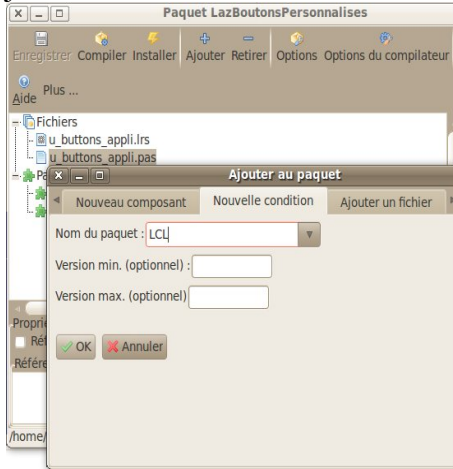
Tous les Composants LAZARUS possèdent le Constructeur Create avec comme paramètre le Propriétaire du Composant. Notre Composant est donc automatiquement détruit lorsque son parent se détruit.

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 "StdCtrls". Cette unité est incluse dans le paquet LCL. Dans le paquet "Ajoutez" la "Condition" "LCL".



Ajouter la condition "LCL" au paquet

Vous pouvez "Compiler" le paquet pour trouver des erreurs. Si votre Code Source est bon vous pouvez appuyez sur "Ctrl"+ "Shift" + "C". Cela va créer les deux méthodes.

Maintenant nous allons remplir les deux méthodes surchargées dans la partie "implémentation" :

implémentation

uses Forms;

// Créer des constantes permet d'éviter de se tromper pour leur réutilisation

const

CST_FWCLOSE='TMyClose'; **// Nom du fichier Image**

{ 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
  // Charge le fichier Image
    Glyph.LoadFromLazarusResource ( CST_MYCLOSE);
    End;
end;

```

Ajout la constante CST_MYCLOSE à la partie interface de votre unité de composant. Affectez lui le nom de votre classe créée, à savoir "TMyClose".

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 cette Source soit exécutée deux fois dans l'Application pour que cette Source ait rempli 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'événement "OnClick" que l'on évitait. Nous n'éludons effectivement pas l'événement "OnClick" en vérifiant s'il

existe. Ainsi nous permettons d'utiliser le bouton de fermeture uniquement pour l'image centralisée.

Le Code spécifique est automatisé avec le maximum de Composants possédant différents objectifs. On se rend compte que, bien que le Composant permette de centraliser, celui-ci 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 ne descendant pas de "TMyClose".

a) 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 avec une licence "Art Libre", ou "Creative Common" avec ou sans "by" avec ou sans "SA".

Vérifiez la licence. Si vous vous posez des questions sur les licences vous pouvez contacter un Groupe d'Utilisateurs LINUX local.

Les licences Libres possèdent plus de facilités quant aux modifications. Vous avez une définition de Libre dans le glossaire.

Le fichier image doit être un fichier "XPM", de 24 pixels sur 24 pixels,

avec comme nom le type de la classe de notre bouton. Cela permet de charger l'image du bouton dans la palette de composants.

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", si ce n'est pas déjà fait.

Ouvrez un "Terminal", ou allez dans votre accessoire "Ligne de commande".

Copiez-y le lien vers l'exécutable "lazres". Copiez le lien vers le fichier ressources "lrs" qui va être créé. Puis copiez le lien vers le fichier XPM.

Vous pouvez éventuellement ajouter d'autres images, si vous voulez créer d'autres boutons.

Nous allons ajouter le fichier ressource à l'unité.

Allez tout à la fin de l'unité de votre Composant. Insérez cette Source avant le "end." final :

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

L'unité "ressources" est nécessaire.

La directive $\{ $\$$ I}$ ajoute le fichier ressource au Code Source.

b) 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 Composant.

Pour qu'un Composant soit enregistré dans une palette, voici le Code Source à affecter à une unité d'enregistrement du Composant.

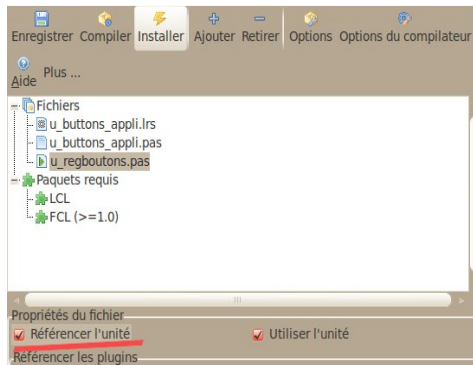
Créez cette nouvelle unité dans votre nouveau paquet :

```
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. Vérifiez pour cette unité si la case à cocher "Référencer l'unité" est cochée.

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.



"Référencer l'unité" est coché pour l'unité

c) Modifier ou Surcharger ?

Si votre Composant ajoute un objectif au Composant initial, il est préférable de créer une autre unité surchargeant le Composant. Cette dernière permet 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, l'accord de l'auteur est nécessaire. Vous avez

alors accès au gestionnaire de version permettant de le modifier.

Sinon que le Composant peut être abandonné, et sa licence permet de le modifier. S'il n'y a aucune licence, contactez l'auteur.

Sinon vous ne faites qu'ajouter des options mineures. Seulement, sans l'accord de participation de l'auteur, vous ne pouvez pas correctement mettre à jour le Composant, Vous êtes alors seul responsable de ses évolutions. Il suffit souvent de contacter l'auteur, qui sera en général content de vous aider à améliorer son savoir-faire.

8) 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. Évitez d'utiliser des contournements, car ils peuvent provoquer de nouvelles erreurs à la mise à jour.

9) 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 subliment 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 savoir-faire sur d'autres outils de programmation.

10) LE LIBRE ET L'ENTREPRISE

LAZARUS contredit les professeurs ou institutions, indiquant que les Composants nécessitent plus de temps en maintenance, qu'en gain de temps à l'utilisation. 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. Cela n'empêche pas forcément une activité commerciale.

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. Vous serez par contre peut-être rémunéré pour des modifications importantes. Attendez de votre communauté 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.

11) TESTER SON SAVOIR-FAIRE

Un composant est au cœur de votre savoir-faire. Si vous modifiez votre composant cela implique forcément des réactions en chaîne dans votre logiciel. Il est donc très intéressant de mettre en place des jeux de tests en surchargeant vos paquets de composants.

12) L'EXEMPLE

FPCUnit vous permet de tester les logiciels FREE PASCAL. Nous allons mettre en place des jeux de tests pour notre bouton de fermeture.

Dans le répertoire de votre composants créez un dossier "tests". Dans le menu "Fichier", créez une "Nouvelle" "Application de tests FPCUnit", en remplaçant le nom du jeu de tests "TestCase" par "TestButtons". "Enregistrez" le projet en le nommant "fpcunitbuttons", dans le dossier créé.

Ajoutez-y dans la clause "uses" du projet l'unité "Interfaces". Cette unité est nécessaire pour gérer les formulaires, dont nous avons besoin pour tester le bouton de fermeture du formulaire.

Dans le menu "Projet", puis "Inspecteur de Projet", ajoutez avec "+" votre paquet en "Nouvelle Condition". Ou bien ajoutez le dossier "../" dans les "Options du projet", puis "Options de compilation", puis le dossier des bibliothèques. Cela permet de tester le composant sans avoir à l'installer.

a) Le formulaire

Ajoutez un nouveau formulaire nommé "testform". Dans la partie "interface", ajoutez l'unité "u_buttons_appli".

Dans le formulaire, nous allons mettre en place le bouton que nous avons créé.

Le formulaire possède ces méthodes et variables, dont certaines existaient déjà dans l'ascendant de la fiche :

```
{ TTestForm }
```

```
TTestForm = class(TForm)
```

```
public
```

```
  Bouton : TMyClose;
```

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

```
  procedure RunBouton;
```

```
  destructor Destroy; override;
```

```
end;
```

Nous n'avons pas besoin de tester le formulaire, mais le bouton. Nous créons donc le bouton manuellement, sans utiliser l'IDE :

```
procedure TTestForm.RunBouton;  
begin  
  Bouton := TMyClose.Create(Self);  
end;
```

Il est intéressant d'utiliser d'autres moyens de création, que ceux utilisés par le développeur, afin de vérifier le composant autrement. Ne négligez pas cependant les jeux de tests à effectuer.

Nous créons l'auto-destruction du formulaire, afin de vérifier plus tard s'il est correctement détruit :

```
procedure TTestForm.DoClose(var AAction: TCloseAction);  
begin  
  inherited DoClose(AAction);  
  AAction := caFree;  
end;  
  
destructor TTestForm.Destroy;  
begin  
  inherited Destroy;  
  TestForm := nil;  
end;
```

Nous testons le bouton et ne testons pas le formulaire. Nous nous assurons donc que le formulaire est correctement affecté à "nil", même si dans les sources il est censé être correctement affecté à "nil".

b) La classe de tests unitaires

Voici à quoi ressemble notre classe de tests :

```
{ TTestButtons }
```

```
TTestButtons= class(TTestCase)  
protected  
  procedure SetUp; override;  
  procedure TearDown; override;  
published  
  Procedure Tests;  
  procedure TestHookUp;  
end;
```

Les deux méthodes publiées sont affectées dans les jeux des tests par l'initialisation d'unité suivante, située tout à la fin de votre unité de classe de tests :

```
initialization  
  RegisterTest(TTestButtons);  
end.
```

Les tests sont récupérés par le projet de tests avec cette méthode.

Voici la méthode de création des Objets testés :

```
procedure TTestButtons.SetUp;  
begin  
  // Initialize  
  Application.CreateForm( TTestForm, TestForm );  
  TestForm.RunBouton;  
end;
```

Dans votre savoir-faire il est préférable de mettre les commentaires en anglais, afin d'internationaliser vos composants.

Le formulaire a été créé. Il est préférable de le détruire par nous même :

```
procedure TTestButtons.TearDown;
```

```
begin  
  // Freeing  
  TestForm.Free;  
end;
```

Passons aux jeux de tests :

```
procedure TTestButtons.Tests;  
begin  
  SetUp;  
  // testing  
  AssertTrue('Testing Glyph of  
TMyClose',assigned(TestForm.Bouton.Glyph));  
  TestForm.Bouton.Click;  
  AssertTrue('TestForm of TMyClose not  
visible',Assigned( TestForm ));  
  TearDown;  
end;
```

L'implémentation précédente est une des deux méthodes publiées qui sera appelée pour tester le bouton. Il est donc nécessaire de créer les composants avant les jeux de tests avec "SetUp", pour les détruire à la fin avec "TearDown".

La classe TTestCase possède elle-même des méthodes permettant d'effectuer les tests en les enregistrant dans le projet.

Un jeu de test FPCUnit commence par "Assert", suivi du type de jeu de tests.

"AssertTrue" va donc tester si le deuxième paramètre est bien à "True". Quant au premier paramètre il permet de savoir quel jeu de test nous réalisons. Si vous voulez tester une valeur utilisez "AssertEquals".

Nous testons bien le fait que le "Glyph" soit affecté à la création, sans

avoir besoin de formulaire. Nous testons aussi la fermeture du formulaire par le bouton "TMyClose". Nous ne testons pas cependant le fait que le Glyph ne soit pas enregistré dans le formulaire.

Cette méthode publiée permet de tester si les jeux de tests sont effectivement exécutés :

```
procedure TTestButtons.TestHookUp;  
begin  
  Fail('Test of button finished');  
end;
```

La méthode "Fail" envoie un message d'erreur. Elle devra donc être vue à l'exécution. Elle pourra être enlevée en utilisant d'autres utilitaires de tests plus évolués.

c) Le projet de tests

Dans le fichier "lpr" créez cette classe :

```
{ TMyTestRunner }  
  
TMyTestRunner = class(TTestRunner)  
  // override the protected methods of TTestRunner to customize  
  its behavior  
  public  
    procedure RunTests;  
  end;
```

Cette méthode permet de démarrer tous les jeux de tests :

```
{ TMyTestRunner }
```

```
procedure TMyTestRunner.RunTests;  
begin  
  DoTestRun(GetTestRegistry);  
end;
```

Ensuite nous créons l'exécution du projet :

```
var  
  Application: TMyTestRunner;  
  
begin  
  Application := TMyTestRunner.Create(nil);  
  Application.Initialize;  
  Application.Title := 'FPCUnit Console test runner';  
  Application.RunTests;  
  Application.Free;  
end.
```

Comme dans un projet standard, nous créons un Objet central, qui gère les Objets de tests cette fois ci.

Après avoir initialisé, puis nommé notre logiciel, nous démarrons les tests. Enfin nous libérons notre logiciel.

13) *EXERCICE*

Créez un rapport de tests, grâce à la documentation de FPCUnit.

14) CONCLUSION

Les composants doivent être testés. Mais ils permettent de gagner beaucoup de temps grâce à votre IDE RAD.

Vos composants permettent de gagner beaucoup de temps, tout en créant des logiciels personnalisés et intuitifs.

A la fin vous disposez d'un moteur créant vos logiciels grâce à une analyse uniforme. Les gains de temps sont minimes avec le moteur, mais ce dernier permet de fiabiliser votre création de logiciels.

M) DE PASCAL VERS FREE PASCAL

CREATIVE COMMON BY SA

1) INTRODUCTION

FREE PASCAL permet de créer des savoir-faire multiplate-forme à partir de savoir-faire WINDOWS comme DELPHI ou TURBO PASCAL.

A partir de ces savoir-faire ou Logiciels, on crée des bibliothèques FREE PASCAL. Cependant cela demande l'adaptation d'une partie des Sources.

2) 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 :

- Insérez la partie graphique dans une fenêtre et adaptez la taille graphique à la fenêtre
- Recréez la partie graphique avec les Composants visuels

3) GESTIONNAIRE DE DONNÉES

Si vous souhaitez beaucoup d'utilisateurs ou de données créez un lien vers un Système de Gestion de Base de Données, comme FIREBIRD, SQLITE, MY SQL, voire ORACLE.

FIREBIRD et SQLITE 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 nommés "clusters". ORACLE lui permet de créer un seul serveur de données très lourd.

Il est possible d'utiliser des gestionnaires de données mi-lourds, si votre entreprise est répartie. On crée un réseau maillé permettant de dupliquer les informations. Le siège prend le pas sur les filiales.

4) 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. Cela permet d'alléger l'exécutable WINDOWS.

Vous pouvez trouver les Composants qui vous manquent, grâce à une recherche sur un site Web de Composants comme www.LAZARUS-components.org.

5) **LES DIRECTIVES DE COMPILATION**

Les instructions de compilation permettent de porter son programme vers les différentes plates-formes. Elles définissent une façon de compiler l'unité. Elles sont donc à placer au début de l'unité.

a) **Compatibilité DELPHI**

Voici une instruction de compilation FREE PASCAL :

```
{$mode DELPHI} // Compilation compatible DELPHI
```

Il existe des DELPHI gratuits, permettant de réaliser des applications non commerciales. Si quelqu'un vous demande d'être compatible DELPHI vous pouvez le faire grâce à une vieille licence de DELPHI 7, voire DELPHI 6.

Les directives de compilations permettent de compiler uniformément l'unité, en éludant des Sources incompatibles avec certaines plates-formes. Ainsi, les Sources d'une plate-forme sont éludées lorsqu'on compile sur une autre plate-forme.

L'instruction de compilation, qui permet à FREE PASCAL de rendre le Code Source compatible DELPHI, n'est pas lisible par DELPHI. On peut y ajouter une directive de compilation pour DELPHI, si on l'utilise.

Nous avons vu la notion de pointeur dans le chapitre sur la **Programmation Procédurale avancée**.

Le mode DELPHI permet de supprimer la notion d'adresse de pointeur. Cela n'est pas gênant, au contraire, car DELPHI protège les pointeurs.

Les pointeurs sont des adresses en mémoire, redirigeant vers un autre

endroit de la mémoire. Ils permettent de manipuler les Objets. En mettant en mode DELPHI vous pouvez oublier les pointeurs en partie, car le compilateur FREE PASCAL n'est pas entièrement compatible avec ce mode, surtout avec les dernières grammaires FREE PASCAL.

Le développeur a besoin de manipuler des Objets, pas de gérer des pointeurs. Le pointeur peut cependant être utile pour scruter les tableaux. Mais PASCAL Objet permet toujours cela.

Il est possible en Pascal d'éluder totalement les affectations de pointeurs, grâce à cette directive à ajouter à tout formulaire :

```
{$IFDEF FPC}  
{ $mode DELPHI}  
{ $R *.lfm}  
{ $ELSE}  
{ $R *.dfm}  
{ $ENDIF}
```

Ces directives peuvent être mises en tout début d'unité et une seule fois. Elles définissent comment va être compilée l'unité.

Ici l'instruction de compilation "Mode DELPHI" est exécutée si on utilise le Compilateur FREE PASCAL. Aussi on charge le fichier "lfm". Sinon on charge les Composants contenus dans le fichier "dfm". Si votre unité n'est ni un formulaire, ni un module de données, vous pouvez enlever les directives de chargements de fichiers "lfm" et "dfm".

Un fichier "dfm" est une correspondance DELPHI de fichier "lfm" avec LAZARUS. Les fichiers "dfm" et "lfm" contiennent les informations des Composants chargés dans un module de donnée, ou un formulaire. Ces Composants sont visibles dans l'"Inspecteur d'Objets".

Cette directive de compilation permet de placer une instruction

DELPHI dans le Code :

```
{$IFDEF FPC}  
var DirectorySeparator : Char = '\';  
{$ENDIF}
```

Ici on définit le caractère de séparation de répertoire quand on est sur DELPHI.

b) Compatibilités avec les plates-formes

Cette Source permet de déclarer sur WINDOWS le séparateur de répertoires des chemins :

```
{$IFDEF WINDOWS}  
DirectorySeparator := '\';  
{$ENDIF}
```

Cette Source permet de déclarer sur les systèmes UNIX le séparateur de répertoires :

```
{$IFDEF UNIX}  
DirectorySeparator := '/';  
{$ENDIF}  
{$IFDEF LINUX}  
DirectorySeparator := '/';  
{$ENDIF}
```

En effet lorsqu'on crée un Logiciel multiplate-forme, il est nécessaire de penser aux différences entre les différentes plates-formes. Aussi penser aux erreurs pouvant se produire permet de sécuriser son Logiciel.

Nous ne sommes pas des machines. Pourtant nous créons des Logiciels

répondant à une logique. Si cette logique n'est pas respectée notre programme n'est pas sûr.

6) 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.

Remplacez le Code graphique, et les liens vers les librairies WINDOWS, par du Code Source LAZARUS. LAZARUS possède une large bibliothèque de Codes Sources. Des Composants Libres DELPHI peuvent être traduits vers LAZARUS.

Le Code graphique LAZARUS utilise les différentes librairies Libres de chaque plate-forme. Ces différentes librairies sont homogénéisées en des librairies génériques.

Les unités à ne pas utiliser sont les unités spécifiques à une seule plate-forme, 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 plate-forme 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 indique alors ce que vous pouvez faire.

N) L'OBJET ET LES JEUX

CREATIVE COMMON BY NC-ND

1) *INTRODUCTION*

Créer un jeu permet de comprendre l'Objet, si on utilise un moteur de sprites ou un moteur 3D. Les Objets d'un jeu sont des représentations plus ou moins humaines de personnages, les acteurs du jeu. Ces Objets, pouvant devenir des personnages grâce à l'Intelligence Artificielle, permettent de sublimer les capacités de la Programmation Objet.

Vous pouvez vous référer au chapitre sur **L'Objet**, afin d'en comprendre les principes.

2) *POURQUOI CRÉER DES JEUX 2D ?*

Les jeux 2D reviennent au goût du jour, avec les téléphones portables. LAZARUS permet de créer des jeux pour WIN CE, ANDROID voire IPHONE, ou tout autre téléphone portable. En effet la plupart des téléphones portables disposent d'un noyau UNIX, voire d'un noyau WINDOWS.

Si on créait un émulateur grâce à LAZARUS, il serait disponible sur l'ensemble des environnements les plus connus.

Les jeux en 2D sont un ensemble de graphismes en 2 dimensions, bougeant selon les désirs du joueur. Les graphismes des jeux 2D sont

appelés sprites.

Un sprite est une image disposée dans un décor. Elle est capable ou pas de se déplacer. Un sprite se détruit. Il peut éventuellement toucher un autre sprite, être repoussé, etc.

Il existe un créateur de jeux nommé GAME MAKER, si vous ne souhaitez pas créer de jeux pour les SMARTPHONES.

3) LES JEUX 2D AVEC ZEN GL

ZEN GL est une librairie permettant de créer des jeux 2D. Elle gère l'écran, des sprites, les sons, les collisions. Elle est disponible sur WINDOWS, LINUX, MAC OS. Elle est compatible avec certains SMARTPHONES.

ZEN GL utilise l'accélération OPEN GL, qui permet l'accélération graphique par un processeur vidéo 3D, compatible OPEN GL. Nous utilisons donc peu de librairies LAZARUS. Nous utilisons avant tout les librairies de ZEN GL.

La librairie ZEN GL peut être utilisée sans LAZARUS, avec seulement le compilateur FREE PASCAL. Elle peut aussi être utilisée avec DELPHI.

Il est possible avec ZEN GL de créer des jeux pour les SMARTPHONES utilisant un processeur compatible OPEN GL.

Les Objets peuvent être utilisés au sein de ZEN GL, mais vous pouvez faire sans le moteur de sprites Objets. Si vous faites sans l'Objet, votre logiciel se complexifiera difficilement.

4) ZEN GL Sous LINUX

Sous LINUX, l'OPEN GL n'est pas installé par défaut. En effet l'OPEN GL augmente l'utilisation du processeur graphique. L'ordinateur consomme plus alors.

Vous devez installer ces deux bibliothèques de développement :

- libgl1-mesa-dev
- libglu1-mesa-dev

Pour faire fonctionner le jeu vous n'aurez pas besoin de ces deux bibliothèques, mais des paquets libgl1-mesa-glx et libglu1-mesa. Vous créez alors un paquet contenant ces deux dépendances, avec aussi la bibliothèque ZEN GL.

5) UN PROJET DE JEU

Il est préférable de reprendre un projet de démonstration de ZEN GL.

Votre nouveau projet sera dans le même répertoire que le projet de démonstration. Sinon vous devez rediriger correctement les liens vers ZEN GL, dans les "Options de compilation" du projet.

Copiez, à partir des exemples sur www.liberlog.fr, les dossiers "Images" et "Sons", l'unité "zengl_sprite_image", vers le dossier de votre projet.

L'unité "zengl_sprite_image" va bientôt être intégrée à ZEN GL.

Voici un exemple de projet Code source :

program Multi;

```
{SI definitions.inc}  
{IFDEF FPC}  
{SR MULTI.res}  
{ENDIF}
```

uses

Main in 'Main.pas' **{MainForm}**,

```
{IFDEF STATIC}
```

```
zglHeader
```

```
{ELSE}
```

```
zgl_main,
```

```
zgl_screen,
```

```
zgl_window,
```

```
zgl_timers,
```

```
zgl_keyboard,
```

```
zgl_render_2d,
```

```
zgl_fx,
```

```
zgl_textures,
```

```
zgl_textures_png,
```

```
zgl_textures_jpg,
```

```
zgl_sprite_2d,
```

```
zgl_primitives_2d,
```

```
zgl_font,
```

```
zgl_text,
```

```
zgl_math_2d,
```

```
zgl_utils,
```

```
zgl_Mouse,
```

```
zgl_sound
```

```
{ENDIF}
```

```
;
```

```

//-----
// Name: Initialisation
// Création de la fenêtre de jeu
//-----

Begin
  {$IFDEF STATIC}
  zglLoad( libZenGL );
  {$ENDIF}

  randomize();

  timer_Add( @TimerGame, 16 );

  zgl_Reg( SYS_LOAD, @Init );
  zgl_Reg( SYS_DRAW, @Draw );
  zgl_Reg( SYS_EXIT, @Quit );

  // EN: Enable using of UTF-8, because this unit saved in UTF-8
encoding and here used
  // string variables.
  zgl_Enable( APP_USE_UTF8 );

  wnd_SetCaption( 'Casse-Briques' );

  wnd_ShowCursor( TRUE );

  scr_SetOptions( EcranX, EcranY, REFRESH_MAXIMUM, FALSE,
FALSE );

  zgl_Init();
End.

```

Ne compilez pas : Il manque les procédures "TimerGame", "Init", "Draw", "Quit".

Il est nécessaire de laisser les commentaires en anglais. Seuls les commentaires contradictoires doivent être remplacés.

Vous voyez que votre jeu peut être utilisé avec seulement l'unité "zglHeader". Cette unité permet de charger la librairie ZEN GL en dehors de l'exécutable.

ZEN GL est alors dans une librairie exécutable, en DLL sur WINDOWS ou en SO sous LINUX. On utilise cette librairie une fois que votre jeu est finalisé, afin de réduire la taille de l'exécutable tout en centralisant le Code. Cette librairie est disponible, au sein du répertoire "src/LAZARUS" du projet ZENG GL.

"randomize" initialise le hasard pour la procédure "random".

La procédure ajoutée par "timer_Add" est le Timer, appelé régulièrement.

Les procédures ajoutées par "zgl_Reg" permettent :

- D'initialiser les sprites et les sons avec "SYS_LOAD".
- De dessiner les sprites avec "SYS_DRAW".
- De libérer les variables affectées avec "SYS_EXIT" en quittant.

"zgl_Enable" vous permet d'ajouter des propriétés à votre jeu. Ici nous utilisons les caractères étendus UTF8 de LINUX et LAZARUS.

"scr_SetOptions" vous permet de modifier l'affichage à l'écran. Par exemple vous pouvez paramétrer le mode plein écran avec le quatrième paramètre.

"zgl_Init" démarre le jeu et appelle la procédure d'initialisation. Ensuite le Timer est activé. Les procédures ou fonctions placées ensuite seront exécutées à la fermeture du logiciel.

Créez une nouvelle unité nommée "Main", intégrée au projet, contenant la même clause "uses" que celle montrée ci-avant.

Après les unités, dans interface, ajoutez ces déclarations de procédures :

```
{ Zen GL Méthodes }  
procedure Quit;  
procedure Draw;  
procedure Init;  
procedure TimerGame;  
  
{ Procédures de jeu appelée par le timer }  
procedure SceneFlipping;  
procedure SceneMain;  
  
{ Procédure de nouveau jeu }  
procedure StartSceneMain;  
{ Procédure de fin de jeu }  
procedure EndSceneMain;
```

"TimerGame" appelle "SceneFlipping" sous condition, qui n'appelle pas forcément "SceneMain".

"StartSceneMain" permet d'initialiser les variables de jeu, comme un constructeur. Son antagoniste est "EndSceneMain".

Dans votre unité Main, ajoutez ces constantes :

```
const  
{ $IFDEF FPC }  
    DirectorySeparator = '\';  
{ $ENDIF }  
{ Constantes globales }  
    ScoreVie =1000;  
    AjouteVie = 16;  
    EcranX=640;
```

```
EcranY=400;
```

{Constantes liées aux niveaux }

```
Fin = 50;
```

{Constantes liées à l'écran }

```
BitCompte = 16;
```

```
LimiteXImages = 100 ; {94;}
```

```
LimiteYImages = 50 ;{47; }
```

{Constantes liées aux balles }

```
Lente =20; //Vitesse lente
```

```
VitesseMax = 60;
```

```
ZBalles =3;
```

```
MaxiBalles = 60;
```

```
DecalX = 2;
```

```
DecalY = 2;
```

{Constantes liées aux briques }

```
ZBRiques = 1;
```

```
ZItems = 0;
```

```
BriquesX = 8;
```

```
BriquesY = 30;
```

```
NeCasse = 1;
```

```
Meurt = 2;
```

```
TempsChange = 30000;
```

```
TempsAccelere = 20000;
```

```
TempsInversion=5000;
```

```
TempsImmobile=3000;
```

{Constantes liées au joueur }

```
ZJoueurs = 2;
```

// Joueur 1

```
JoueurBasX = EcranX/2-40;
```

```
JoueurBasY = EcranY-30;
```

```
BougePaletteX= 30;
```

{Constantes liées aux sprites }

```
PCasse = 100;
```

```
Vitesse = 20;
```

```
Ralenti = 18;
```

```
dir_Images = 'Images' + DirectorySeparator;
```

```
dir_Menus = 'Menus' + DirectorySeparator;
```

```
dir_Niveaux = 'Niveaux' + DirectorySeparator;
```

```
dir_Data = 'DATA' + DirectorySeparator ;
```

```
dir_Items = 'Items' ;
```

```
NbBriquesX = 20; {12;}
```

```
NbBriquesY = 20; {14;}
```

```
FinCouche = 5;
```

```
BriqueMinX = 32;
```

```
BriqueMinY = 20 ;
```

```
BBase = 1;
```

```
TempsEnvoi = 600;
```

```
BSpeedUp = 8;
```

```
BSpeedDown = 9;
```

```
BIndestructible = 2;
```

```
FinBriques = 28;
```

```
ImageBalle = 'balle' ;
```

```
ImageBrique= 'brick' ;
```

```
ImagePalette= 'palette' ;
```

```
ImageExtension = '.png' ;
```

```
SonExtension = '.wav';
```

```
MaxBriques = 28 ;
```

```
MaxCouche = 3;
```

```
MOVECOUNT_WITHOUT_TIME = 1/100 ;
```

Certaines de ces constantes sont compréhensibles. Il y a la taille de l'écran, utilisée dans l'initialisation de l'écran. Il faut donc relier le projet à notre nouvelle unité.

A la fin la constante MaxCouche permet de créer des couches de briques dans un tableau. Il est préférable d'utiliser une constante, comme fin de tableau, afin d'optimiser le Code.

La constante "MOVECOUNT_WITHOUT_TIME" permet d'ajuster la vitesse des sprites par rapport au timer ZEN GL, exécuté régulièrement.

a) Les types non Objets

Les types non Objets permettent de simplifier la programmation :

```
type TIndice = 0..3;
```

```
TCases      =ARRAY[0..MaxCouche] of TIndice;
```

```
TInputType = (itInactive, itMouse, itKeyboard1, itKeyboard2,  
itJoystick1, itJoystick2);
```

```
TBalleModif = ( bmBalleDessus, bmAccelere, bmRalenti,  
bmPlomb, bmColle, bmNormal, bmFolleHautBas, bmFolle,  
bmFolleGaucheDroite, bmPlusGrosse, bmMoinsGrosse);
```

```
TPlayerModif = ( pmErased, pmEnlever, pmAfficher,  
pmEnvoiBalle, pmBouge );
```

```
TModeBrique = ( mbAucun, mbMeurt );
```

"TIndice" représente les différents numéros de briques, dont l'indice 0 n'indiquant aucune brique. Ces numéros permettent d'aller chercher les images des briques.

"TCases" est le tableau contenant les briques à afficher après que la première brique soit cassée. Ce tableau n'est pas dynamique, pour ne pas avoir à le redimensionner.

Les types énumérés représentent différents états. Ils permettent de ne pas avoir à utiliser de constantes, tout en sécurisant la création de Code.

b) Les Objets ZEN GL

La programmation Objet permet de modéliser les acteurs du jeu. Ainsi vous pourrez améliorer facilement ce jeu, pour qu'il devienne complexe.

Les Objets suivants peuvent être mis dans la partie "implementation" des sources, afin de ne pas surcharger l'exécutable. En effet les déclarations dans "interface" ajoutent du Code déclaratif, à utiliser dans d'autres unités, ce que nous ne voulons pas.

Nous déclarons d'abord la classe "TBalle", en abstrait, afin de pouvoir l'utiliser au sein des autres types, avant de déclarer réellement "TBalle" :

```
type  
TBalle = class;  
  
{ TCollisionSprite }
```

```

TCollisionSprite = class(TAnimatedSprite)
public
  procedure CollisionBalle ( const Balle : TBalle; const AllowPlomb
: Boolean ); virtual;
end;

```

Avec cet Objet nous surchargeons un Objet de base de l'unité "zengl_sprite_image".

On fusionne des possibilités en les centralisant dans quelques Objets, qui sont utilisés ensuite. Par exemple une balle sera toujours capable de réagir à d'autres sprites, dans un casse-briques.

Voici maintenant la déclaration du deuxième Objet, abstrait, du jeu :

```

// Sprite du ou des joueur
{ TPlayerSprite }

TPlayerSprite = class(TAnimatedSprite)
private
  Deplacement1,Deplacement2,Deplacement3 :Extended;
  BalleModif : TBalleModif;
  ControlX,ControlY,
  Mode,NoPalette : Integer; // Mode (mort vivant touché)
  Controle : TInputType ;
  PlayerModifs : set of TPlayerModif;
  Intervalle, // Intervalle de rebondissement des balles
  CompteEnvoi,
  CompteurInversion,
  CompteurTire,
  CompteImmobile,
  PaletteEnCours,
  Inversion,
  VitesseBalles : integer;

```

```

NoJoueur : Integer ;
procedure Erase;
procedure EnvoiBalle;
procedure UneCollisionX ( const Sprite : TSprite ; var Done:
Boolean);
procedure ChangeVitesses( const Vitesse : integer);
function InitPos ( const TheCoord : Double ; const NewCoord :
Double ):Double;
procedure Deplace(const InputCoord: Double);
public
constructor Create( const AParent: TSpriteEngine; const
Apparaître : Boolean; const Controle : TInputType); overload;
virtual;
destructor Destroy; override;
// Methodes abstraites
procedure BallesModif; overload; virtual; abstract;
procedure CreatePlayer; virtual; abstract;
// Methodes implémentées
procedure SetPalette; virtual;
procedure DoMove(MoveCount: Double); override;
procedure DetruitJoueur; virtual;
procedure Cree; virtual;
procedure BallesModif ( const Modif : TBalleModif ); overload;
dynamic;
procedure UnMouvement; dynamic;
procedure UnMouvementX; dynamic;
procedure BallePlus; dynamic;
procedure SetBalleColle ( const UneBalle : TBalle ); virtual;
procedure SetBalleCollePosition ( const UneBalle : TBalle );
virtual;
procedure Controleur(Control : TInputType); virtual;
procedure AppuieGaucheDroite ( const
ToucheAEnfoncerGauche , ToucheAEnfoncerDroite : Byte );
virtual;

```

```
procedure AppuieButton ( const ToucheAEnfoncer : Byte );  
virtual;  
procedure ErasePlayer (); virtual;  
end;
```

Cet Objet représente le joueur, où qu'il soit. En effet il existe, un peu plus loin, un autre Objet joueur, héritant de ce premier Objet, qui lui se place automatiquement en bas.

Nous pourrions représenter cet Objet avec un schéma UML, possédant alors des attributs ou variables avec des méthodes. Ce schéma permettrait de comprendre le jeu globalement. Quant aux détails, on s'en occupe lorsqu'il est nécessaire de les comprendre, en ajoutant des commentaires.

Vous voyez que les méthodes publiques ont des déclarations supplémentaires. La classe "TPlayerSprite" est abstraite, car elle possède des méthodes abstraites. Elle ne peut être utilisée telle quelle, sinon cela provoque une erreur à l'exécution, si ces méthodes abstraites ne sont pas surchargées dans son descendant.

Nous ne vous demandons pas de tout comprendre. Par ailleurs, même moi, qui ai créé entièrement le jeu, ne comprend le jeu qu'en scrutant les sources.

Que faut-il comprendre ?

Dans chaque Objet du jeu, on surcharge la méthode "DoMove". Cette méthode permet de déplacer le sprite à un instant donné, par la méthode "Move" du moteur de sprites. La plupart des autres méthodes, mis à part le constructeur, sont dépendantes du mouvement du sprite.

Lorsque j'ai créé cet Objet, j'ai pensé à la modularité de cette création, permettant de scinder les sources afin d'éviter le copier-coller, pour

faciliter la programmation. Il est plus facile de scinder les sources avec l'Objet, si on utilise les différentes facultés de la programmation Objet.

En scrutant cette déclaration précédente, on voit qu'il existe une variable "Mode" déclarée en entier. Ce mode pourrait être déclaré en énuméré, afin d'éviter les erreurs.

Voici l'Objet représentant la balle :

```
{ TBalle }

TBalle = class(TAnimatedSprite)
private
    DecalePalette,
    Vitesse,
    Accelere,
    NoBalle,
    BougeX,BougeY,
    Change :integer;
    Mode :integer;
    Activite : set of TBalleModif;// Toute l'activité de la balle
    Grosseur : Integer ;
function PerdJoueur ( const Perdu : Boolean ):boolean;
procedure Limites ( var Bouge : Integer );
procedure ChangeVitesse;
procedure BalleModif;
procedure SetImageGrosseur;
procedure SetImagePosition;
public
procedure DoMove(MoveCount: Double); override;
procedure ChangeUneDirection ( const Decalage : Double );
virtual;
    // Gestion des collisions
procedure BeforeCollision ( const Sprite: TCollideSprite );
override;
```

```

procedure DoCollision(const Sprite: TCollideSprite; var Done:
Boolean); override;
procedure CollisionBalle ( const Balle : TBalle ); virtual;

procedure ColleBalle;
procedure Envoi; virtual;
constructor Create ( const AParent: TSpriteEngine; const X, Y :
Double ); overload;
destructor Destroy; override;
procedure Show ; virtual;
// Détruit la balle
procedure Destruction;
end;

ABalle = ^TBalle;

```

C'est ZEN GL qui détruit les sprites avec la variable "kill", que nous affectons avec la méthode "Dead". J'ai créé cette méthode dans ma surcharge du moteur de sprites ZEN GL, afin d'être compatible avec une autre librairie plus aboutie à l'époque, mais abandonnée : Cette librairie de jeu 2D s'appelait ANDORRA 2D. Garder la compatibilité me permet de réutiliser mon jeu.

Mes critiques sur l'ancien moteur m'ont permis d'optimiser le nouveau, si bien que ZEN GL n'est pas entièrement compatible avec ANDORRA 2D.

Certaines méthodes vous serviront, plus tard, à améliorer le casse-briques. Pour l'instant la balle de ne peut pas grossir. Mais il est possible d'implémenter cette possibilité, grâce à la méthode "SetImageGrosseur".

A la fin nous créons un pointeur de balle, afin de manipuler la balle. Référez-vous à la fin du chapitre sur la **Programmation Procédurale**

Avancée.

Voici le joueur du bas, le seul joueur disponible pour l'instant :

```
{ TPlayerBas }  
  
TPlayerBas = class(TPlayerSprite)  
public  
  procedure SetBalleColle ( const UneBalle : TBalle ); override;  
  procedure UnMouvement; override;  
  procedure SetBalleCollePosition ( const UneBalle : TBalle );  
override;  
  // Gestion des collisions  
  procedure DoCollision ( const Sprite: TCollideSprite; var Done:  
Boolean); override;  
  procedure SetPalette; override;  
  procedure CreatePlayer; override;  
  destructor Destroy; override;  
end;
```

Cet Objet hérite de l'Objet créé précédemment. Nous surchargeons certaines méthodes abstraites, comme "BallesModif", parce qu'il est nécessaire de centraliser des sources gérant la balle, dans l'Objet parent, afin d'éviter le copier-coller, nuisible au maintien du logiciel.

Voici ci-après l'Objet créant les sprites des briques :

```
{ TBrique }  
  
TBrique = class(TCollisionSprite)  
private  
  Cases : TCases;  
  NoBrique : Array [ 0 .. MaxCouche ] of TIndice;  
  Mode : TModeBrique;  
  procedure Touche;  
  procedure Destruction;
```

```
procedure DoMove(MoveCount: Double); override;  
public  
  procedure CollisionBalle ( const Balle : TBalle ); overload;  
  procedure InitBrique ( const DesCases : TCases );  
  constructor Create ( const AParent: TSpriteEngine; const  
DesCases : TCases ); overload;  
  destructor Destroy ; override;  
end;
```

Dans votre unité "Main", vous pouvez maintenant copier-coller le reste du Code source, afin de le comprendre.

6) **CONCLUSION**

Lorsque l'on scrute du Code source, il faut penser au Code source central, puis aux sources appelées moins souvent. Cela permet de trouver plus facilement les Bogues.

Regardez bien la gestion du Timer ZEN GL, avec les méthodes de mouvement de chaque Objet, appelées par le moteur de sprites, nommé "SpriteEngine". Ces procédures sont centrales.

La surcharge m'a permis de :

- Éviter le copier-coller.
- Utiliser une librairie existante : ZEN GL.
- D'améliorer la librairie existante, en séparant le jeu d'une nouvelle unité. Les améliorations que j'ai portées à ZEN GL permettent d'accélérer la création de jeux.
- Créer de nouveaux Objets indépendants, utilisant du Code centralisé.

La surcharge permet d'améliorer un Objet. Pensez bien à comprendre l'utilité de la surcharge, permettant de créer de nouveaux Objets, afin d'améliorer le jeu. Les réflexions sur les Objets de jeux sont très proches de l'humain. Le Code doit cependant être optimisé.

O) LA PERSISTANCE D'UN LOGICIEL

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

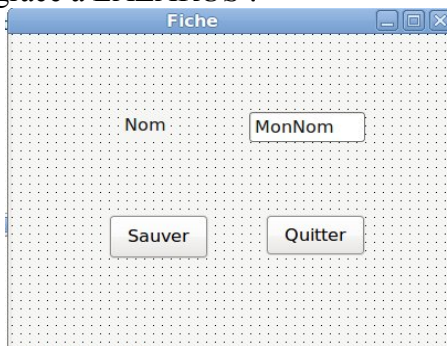
Un Logiciel persistant sauve certaines de ses informations dans des fichiers, afin de garder ces informations pour les réutiliser plus tard.

2) LES FICHIERS ".INI"

Les fichiers avec l'extension ".ini" permettent de sauvegarder des informations de votre Logiciel, afin de les réutiliser plus tard. LAZARUS possède une gestion de fichiers INI, permettant de sauvegarder certaines de vos variables de votre Logiciel.

Un fichier ".ini" stocke peu d'informations. Il sauve en général des informations de personnalisation de l'Application.

Créez cette fiche grâce à LAZARUS :



Une fiche non redimensionnable

Vous pouvez utiliser le bouton de fermeture que nous avons créé précédemment. Renseignez avant tout la propriété "Name" de vos Composants.

Renseignez l'événement "OnClick" du bouton "Sauver" :

```
const CST_EXTENSION_INI = '.ini' ;

implementation

uses Inifiles;

procedure TMaFiche.SauverChange(Sender: TObject);
var Ini_Inifile : TInifile ;
begin
  Ini_IniFile := nil;
  try
    Ini_IniFile := TInifile.Create ( Application.ExeName +
CST_EXTENSION_INI );
    Ini_IniFile.WriteString(Self.Name,MonNom.Name,MonNom.Text);
    Ini_IniFile.UpdateFile;
  Except
    On E:Exception do
      Showmessage ( 'Impossible d'écrire dans le fichier ' +
Application.ExeName + CST_EXTENSION_INI
+ ':'+#13#10 + E.Message ) ;
  End;
  Ini_IniFile.Free ;
end;
```

Ici nous écrivons dans un fichier, en le créant s'il n'existe pas. Une gestion d'exception est nécessaire, car nous gérons une entrée/sortie qui ne marchera pas à chaque fois.

L'Objet gérant le fichier INI " Ini_IniFile" est initialisé à "nil", afin de le libérer éventuellement par sa méthode "free". En effet, nous créons le fichier ".ini" en créant l'Objet "Ini_IniFile". Il y a donc gestion d'entrée/sortie dès la création de l'Objet " Ini_IniFile". La méthode "free" accepte que son Objet soit à "nil".

Si on ne peut créer le fichier ".ini", un message est déclenché. Il contient le nom du fichier en erreur, ainsi que le message d'erreur. L'utilisateur peut alors résoudre le problème.

Si aucune erreur ne se produit, alors le fichier contenant le nom de votre exécutable plus ".ini" est créé. Si vous avez renseigné les noms de Composants, il contient par exemple ceci :

```
[MaFiche]  
MonNom=CONSTANT
```

"CONSTANT" est le nom qui a été renseigné par l'utilisateur. Il peut être remplacé par l'utilisateur.

Si on modifie le nom du Composant ou de la fiche les données sont ajoutées dans le fichier ".ini".

```
[MaFiche]  
MonNom=CONSTANT  
MonNomModifie=CONSTANT  
[MaFicheModifie]  
MonNom=CONSTANT
```

Vous voyez des informations redondantes.

Le fichier ".ini" doit alors être effacé, en perdant ou pas les données sauvegardées. En effet il est possible de charger le fichier ".ini", puis de l'effacer, puis de le sauvegarder avec les nouveaux noms modifiés.

Étudions cette ligne :

```
Ini_IniFile.WriteString(Self.Name,MonNom.Name,MonNom.Text);
```

La méthode "WriteString", de l'Objet de type "TIniFile", écrit une chaîne dans le fichier ".ini".

Le premier paramètre de la méthode est le nom de section. Vous pouvez le vérifier, en tapant avec le curseur entre les parenthèses, simultanément sur "Ctrl" puis "Shift" puis espace. Le nom de section est le nom du Composant fiche. C'est le nom entre crochets dans le fichier INI.

Le nom d'une valeur est le deuxième paramètre. Il est suivi de sa valeur de type chaîne pour la méthode "WriteString". Il existe aussi d'autres méthodes commençant par "Write", que vous pouvez trouver en tapant simultanément sur "Ctrl" puis espace.

Maintenant que le fichier ".ini" existe, utilisez-le pour retrouver les données de votre Logiciel. Renseignez l'événement "OnShow" de votre fiche :

```
procedure TMaFiche.FormShow(Sender: TObject);  
var Ini_Inifile : TInifile ;  
begin  
  Ini_IniFile := nil;  
  try  
    Ini_IniFile := TInifile.Create ( Application.ExeName +  
CST_EXTENSION_INI );  
    MonNom.Text :=  
Ini_IniFile.ReadString(Self.Name,MonNom.Name,MonNom.Text);  
  Except  
    On E:Exception do  
      Showmessage ( 'Impossible de lire le fichier ' +  
Application.ExeName + CST_EXTENSION_INI + ':'+#13#10 +
```

```
E.Message ) ;  
End;  
Ini_IniFile.Free ;  
end;
```

Cette méthode permet de lire dans le fichier INI, de la même manière qu'elle a écrit dans le fichier. Ainsi vous retrouvez le nom saisi, après avoir cliqué sur le bouton.

a) **Exercice**

Il existe deux évolutions à effectuer sur ce Logiciel. Tout d'abord, il est possible de sauvegarder le fichier ".ini", sans avoir à utiliser le bouton "Sauver". On sauve à la fermeture du Logiciel. On renseigne l'événement "OnClose" de la fiche.

Ensuite on voit des informations redondantes. Il est possible de centraliser l'Objet " Ini_Inifile" au sein de l'Objet "MaFiche". Ainsi on enlève la déclaration de variable "Ini_File : TIniFile", dans chaque méthode.

Testez alors si "Ini_File" existe déjà. Ce test peut être centralisé dans une seule méthode.

Renseignez à nil "Ini_File" uniquement dans l'événement de construction "OnCreate", de la fiche. L'Objet est alors créé sur demande. Cela permet de ne pas forcément utiliser le fichier ".ini".

La destruction de l'Objet "Ini_File" se fait alors dans l'événement de destruction "OnDestroy" de la fiche.

Réalisez ce Logiciel.

Vous pouvez trouver un résultat plus abouti de cet exercice dans un Composant du projet "Extended" sur INTERNET.

3) LES FICHIERS ".CSV"

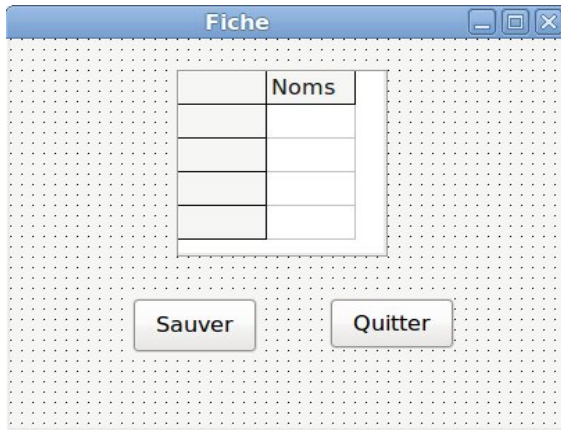
Nous pouvons aussi enregistrer plusieurs noms de type identique dans un fichier.

Les fichiers ".csv" servent à exporter ou importer des données d'autres exécutable.

a) Exercice

Sauvegardez votre premier exercice.

Remplacez le "TLabel" et le "TEdit" par un "TStringGrid" :



Modifiez la propriété "Columns" pour afficher le titre "Noms"

Sur le TStringGrid renommé dans l'"Inspecteur d'Objets", allez dans "Options", puis rendez éditable la grille.

Ajoutez la colonne "Prénoms". Enlevez la colonne grise à gauche avec la propriété "FixedCols" à 0. Vous pouvez adapter la taille des colonnes automatiquement, avec la propriété "AutoFillColumns" de la grille.

Si nous sauvions cette grille dans un fichier ".ini", nous devrions soit attribuer un nom de case à chaque case sauvegardée, soit sauver toute la colonne "Noms" dans une seule valeur.

Ces deux façons de procéder nous seraient coûteuses, si nous ajoutions d'autres colonnes à sauvegarder. Soit il y aurait beaucoup de noms et valeurs dans le fichier ".ini", soit les colonnes pourraient se décaler, avec des types de colonnes mal gérés.

Les fichiers ".csv" ont été conçus pour palier à ce problème.

Un fichier CSV est un fichier de chaînes contenant des colonnes et des lignes. Un fichier CSV est un fichier texte qui sépare les colonnes de son tableau par un ";" ou une ",".

Le "TSDFDataset" est descendant de "TDataSet". Les "Datasets" sont des Composants reliés à des fichiers voire un Système évolué de Gestion de Données. Certains peuvent même utiliser le réseau, pour chercher des informations persistantes.

En général, les "Datasets" peuvent être utilisés avec des Composants dont le nom commence par "TDB". Ces contrôles de données simplifient la persistance de votre Logiciel. Ils possèdent, pour certains, des fonctions et propriétés simplifiant leur utilisation.

Le Composant "TSDFDataset" permet d'exporter des données, vers un fichier CSV. Il n'est pas utilisable aussi facilement qu'un simple Composant de données. Le "TSDFDataset" ne peut s'utiliser facilement, avec les Composants visuels commençant par "TDB".

Le "TDBGrid" s'utilise avec d'autres descendants de TDataSet. Vous pouvez améliorer le projet LAZARUS avec un "Dataset" modifiable pouvant être utilisé avec les contrôles de données.

Nous allons remplacer les méthodes créées précédemment. La sauvegarde a permis de garder l'exercice précédent.

Vous pouvez utiliser le "Refactoring", s'il existe déjà sur LAZARUS, pour modifier votre exercice précédent. Sinon il est possible d'utiliser le remplacement de chaînes, en vérifiant ce qui est remplacé.

Ajoutez le Composant TSDFDataset à votre fiche. Renommez le selon le Code Source ci-après.

Créez, dans la propriété "FieldDefs", les champs chaînes "Nom" et "Prenom".

Créez, dans la propriété "Schema", la ligne "Nom,Prenom". Cela va créer le fichier CSV.

Normalement, on charge la grille lorsque la fenêtre s'ouvre pour la première fois. Puis on sauve dans le fichier les informations de la grille, à la fermeture du Logiciel. Renseignez l'événement "OnShow" ici :

```
const CST_EXTENSION_CSV = '.csv' ;
      CST_FIELD_NOM = 'Nom' ;
      CST_FIELD_PRENOM = 'Prenom' ;

implementation

procedure TMaFiche.FormShow(Sender: TObject);
begin
  // On cherche ou crée le fichier CSV
  SdfDataSetNoms.FileName := Application.ExeName +
  CST_EXTENSION_CSV;
  try
    // Un Dataset s'ouvre pour lire et écrire les données
    SdfDataSetNoms.Open;
  Except
    // Il s'agit d'une entrée/sortie donc on gère les exceptions
    on e:Exception do
      Begin
        ShowMessage ( 'Impossible d'ouvrir le fichier CSV : ' +
  E.Message );
        Exit;
      end;
    end;
    // Le with permet de travailler au sein du SdfDataSet
    with SdfDataSetNoms do
      try
        // On adapte la grille au fichier CSV
        if Noms.RowCount < RecordCount Then
          Noms.RowCount := RecordCount;
        // On se place au début
```

```

First;
// Tant que la End Of File n'est pas atteinte on est dans la boucle
while not EOF do
  Begin
    // Affectation de la grille
    Noms.Cells [ 0, RecNo - 1 ] := SdfDataSetNoms.FieldByName
(CST_FIELD_NOM).AsString;
    Noms.Cells [ 1, RecNo - 1 ] := SdfDataSetNoms.FieldByName
(CST_FIELD_PRENOM).AsString;
    // Enregistrement suivant
    Next;
  end;

Except
// Il s'agit d'une entrée/sortie donc on gère les exceptions
on e:Exception do
  Begin
    ShowMessage ( 'Impossible de lire le fichier CSV : ' +
E.Message );
    Exit;
  end;
end;
end;

```

Vous voyez que cette Source tient difficilement sur une seule page. Cette Source écrite est difficilement testable et gérable. Nous ne pouvons créer que peu de jeux de tests unitaires.

Créez deux méthodes à partir de cette Source. L'une sauvegarde les noms de colonnes, l'autre sauvegardant les lignes. N'oubliez pas d'utiliser la déclaration protégée des pointeurs, avec le mot réservé "const", pour vos paramètres de méthodes.

Exemple :

```

procedure TMaFiche.FillGrid ( const sdf_Data : TSdfDataSet );
Begin
// Le with permet de travailler au sein du SdfDataSet
with sdf_Data do
  try
    // On adapte la grille au fichier CSV
    if Noms.RowCount < RecordCount Then
      Noms.RowCount := RecordCount;
    // On se place au début
    First;
    // Tant que la End Of File n'est pas atteinte on est dans la boucle
    while not EOF do
      Begin
        // Affectation de la grille
          Noms.Cells [ 0, RecNo - 1 ] := Data.FieldByName
(CST_FIELD_NOM).AsString;
          Noms.Cells [ 1, RecNo - 1 ] := SdfDataSetNoms.FieldByName
(CST_FIELD_PRENOM).AsString;
          // Enregistrement suivant
          Next;
        end;

      Except
        // Il s'agit d'une entrée/sortie donc on gère les exceptions
        on e:Exception do
          Begin
            ShowMessage ( 'Impossible de lire le fichier CSV : ' +
E.Message );
            Exit;
          end;
        end;
      end;

```

Utilisez "Ctrl" puis "Shift" puis "C", pour renseigner votre déclaration

de méthode dans la fiche.

Renseignez l'événement "OnClose" de la fiche :

```
procedure TMaFiche.FormClose(Sender: TObject; var CloseAction:
TCloseAction);
var li_i : Integer;
begin
  // Avec le SdfDataset
  with SdfDataSetNoms do
    Begin
      // On efface les anciennes colonnes
      // Le SdfDataset ne gère pas l'édition
      First ;
      while not EOF do
        Delete;
      // On remplit le fichier CSV
      for li_i := 1 to Noms.RowCount -1 do
        Begin
          // Insertion car "Edit" ne marche pas ici
          Insert;
          // On affecte le nom et le prénom
          SdfDataSetNoms.FieldByName(CST_FIELD_NOM).Value :=
Noms.Cells [ 0, li_i ];
          SdfDataSetNoms.FieldByName(CST_FIELD_PRENOM).value :
= Noms.Cells [ 1, li_i ];
          // Validation de la ligne
          Post;
          end;
          // Validation du fichier
          Updated;
          end;
end;
```

Cette Source produit ce genre d'exemple de fichier CSV, créé dans votre répertoire contenant l'exécutable :

```
Nom,Prenom  
CONSTANT,Philippe  
DUPONT,Julien  
BARBIER,Lucien  
PHILIPPE,Catherine
```

b) Exercice

Permettez l'ajout de lignes dans la grille de la fiche.

c) Correction

Il fallait ajouter cet événement Click ou Change à un bouton :

```
procedure TMaFiche.AjouterChange(Sender: TObject);  
begin  
  Noms.RowCount := Noms.RowCount+1;  
end;
```

d) Exercice

Démarrez le Logiciel et testez-le. Améliorez-le en permettant une suppression de ligne.

e) Correction

Utilisez la méthode "DeleteColRow" et la propriété "Row" de votre StringGrid. Testez.

4) **LES FICHIERS ".DBF"**

Les fichiers ".dbf" sont des fichiers binaires permettant d'enregistrer beaucoup de données, tout en retrouvant facilement les informations enregistrées.

a) Exercice

Sauvegardez votre projet CSV.
Renommez le en projet DBF.

Nous allons utiliser cet exercice pour créer une véritable Application, liée à un serveur de données.

Supprimez les boutons "Ajouter" et "Supprimer", le Code Source des événements "OnShow" et "OnClose", les constantes.

Remplacez le "TSDFDataset" par un "TDBFDataset". Pour faire cela remplacez le type "TSDFDataset" par "TDBFDataset" dans le fichier ".pas" et le fichier ".lfm". Vous pouvez accéder au fichier ".lfm" en cliquant sur le bouton de droite de la souris sur la fiche.

Ajoutez au dessus de la grille, le Composant "TDBNavigator" de la palette "Data Controls". Ce Composant automatise la gestion d'une Source de données.

Le DBFDataset gère ces bases de fichiers :

- TableLevel 3 - dBase III+
- TableLevel 4 - dBase IV;
- TableLevel 7 - Visual dBase VII;
- TableLevel 25 - FoxPro.

Affectez 7 à la propriété "TableLevel" du DBFDataset.

Allez sur la propriété "FieldDefs", puis éditez-la. Cette propriété contient les définitions de champ du fichier de table DBF. Le fichier DBF à créer est véritablement une table, possédant un classement.

Nous allons enfin créer une clé pour notre fichier. Une clé permet de trouver le bon enregistrement facilement et rapidement.

"Ajoutez" un champ.

Créez le champ avec le nom "Cle" en affectant la propriété "Name" comme champ Auto-Incrémenté. Le type est "ftAutoInc". Dans la propriété "Attributes" mettre "faRequired" à "True".

"Ajoutez" un champ.

Créez le champ avec le nom "Nom" comme champ chaîne. Le type est "ftString".

"Ajoutez" un champ.

Créez le champ avec le nom "Prenom" comme champ chaîne.

Affectez la propriété "OpenMode" à "omAutoCreate". Cette valeur de propriété permet de créer le fichier "dbf", dès l'ouverture.

Pour enregistrer les définitions affectez "True" à "StoreDefs".

b) Les contrôles de données

Ajoutez un "TDataSource" dans la palette "Data Acces", un

"TDBNavigator" dans la palette "Data Controls".

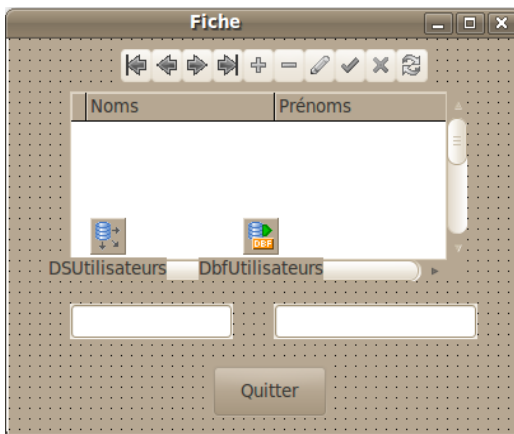
Ajoutez deux Composants "TDBEdit". L'un porte le nom "DBENom" l'autre porte le nom "DBEPrenom".

Dans "DBENom" affectez le "DataField" "NOM" et le "Datasource" "DSUtilisateur".

Dans "DBEPrenom" affectez le "DataField" "PRENOM" et le "Datasource" "DSUtilisateur".

deux "TDBEdit" servant à modifier le nom et le prénom.

Voici ci-après le résultat :



Une fiche évoluée d'édition de données

c) Les Index

Les Index permettent de trouver rapidement les données indexées. Ils existent dans tout Système de Gestion de Données rapide.

Nous allons créer notre premier "Index".

Dans l'"Inspecteur d'Objets" sur "DbfUtilisateurs", éditez la propriété "Indexes"...

Ajoutez un nouvel index.

Créez le fichier index "FICHECLE" en affectant la propriété "IndexFile". Cet Index est sur la clé. Il est donc primaire.

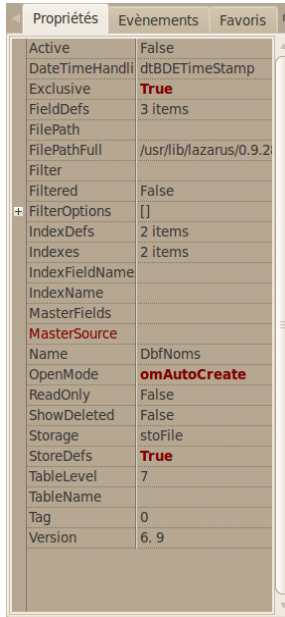
Affectez à « SortField" "CLE" qui est le nom de votre clé primaire. Une clé primaire est la clé principale d'une table représentée ici par un fichier ".dbf".

L'index est aussi unique. On trouve forcément un seul enregistrement grâce à une affectation automatique de l'auto-incrémentation. Affectez à la propriété "Options" "ixPrimary" et "ixUnique".

Ajoutez un nouvel index.

Créez le fichier index "FICHENOM" vers le champ "nom". Cet index permet donc de retrouver un enregistrement du champ "nom". Ce champ est une chaîne. Affectez "ixCaseInsensitive" à "Options". Cela permet de retrouver le nom sans distinction de majuscules, ni minuscules.

Ces index posséderont des fichiers à l'exécution.



Le DBFDataset modifié dans l'inspecteur d'Objets

Vos constantes sont ci-après :

```

const CST_EXTENSION_DBF = '.dbf' ;
        CST_NOM_FICHER   = 'fiches';
        CST_FIELD_NOM   = 'Nom' ;
        CST_FIELD_PRENOM = 'Prenom' ;

```

A l'événement "OnShow" de votre fiche placez cette Source :

```

procedure TMaFiche.FormShow(Sender: TObject);
begin
  // On cherche ou crée le fichier DBF
  with DbfUtilisateurs do
    Begin
      FilePathFull := ExtractFileDir ( Application.ExeName );
      TableName:=CST_NOM_FICHER+CST_EXTENSION_DBF;
    try

```

```
Open;
Except
    on E:Exception do Showmessage ( 'Impossible d'accéder à la
Source de données : ' + #13#10+E.Message );
    end;
end;
End;
```

Nous avons certes passé du temps à définir notre fichier. Mais les Sources prennent maintenant moins de place pour l'utilisation du "Dataset". Le DBFDataset automatise le renseignement de notre grille.

d) Tester

Il est possible que le "DBGrid" gère mal la modification de données. Affectez "dgEditing" à "False" dans les "Options" de votre grille.

e) Conclusion

Cet exemple pourrait être amélioré avec des "TPanel" permettant au Logiciel de prendre toute la page. Nous avons déjà montré la gestion des "TPanel" dans un premier exemple.

Le fichier DBF permet de sauvegarder beaucoup de données. Seulement nous sommes freinés par l'utilisation simplifiée de fichiers. Ces fichiers ne sont pas manipulables aussi facilement qu'avec un serveur centralisé.

Il existe aussi un projet de création automatisée de jeu nommé GAME

MAKER. C'est un projet Open Source avec une communauté française.

<http://www.gamemaker.fr/>

P) LOGICIEL CENTRALISÉ

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Un Logiciel centralisé, donc persistant, accède aux informations d'un ordinateur, le serveur de données. Les données sont les informations identiques pour tous les utilisateurs du Logiciel. Il peut y avoir beaucoup d'utilisateurs, répartis sur différents ordinateurs.

Si votre Logiciel accède à un serveur, il y a de grande chance que celui-ci soit centralisé.

Un Système de Gestion de Données permet de gérer la partie persistante d'un Logiciel. Des fichiers permettent de retrouver ce qui avait été fait avant. Une sauvegarde est nécessaire, car toute donnée électronique peut s'effacer à tout moment.

Un Système de Données comme des fichiers CSV, ou un Système de Données sans gestion du tri, peut s'avérer insuffisant pour gérer des données aussi complexes que celles d'un serveur centralisé. Le serveur ne peut répondre à une demande forte des utilisateurs.

2) ORACLE ET MY SQL

Un Logiciel centralisé utilise en général un Système de Gestion de Base de Données ou SGBD. Le SGBD le plus haut de gamme est ORACLE. Il peut gérer beaucoup d'utilisateurs sur un seul serveur. Il est

automatisable difficilement. Mais il nécessite une licence et un administrateur ORACLE.

Un Système de Gestion de Base de Données gère en général bien le tri et un nombre suffisant d'utilisateurs. Il en existe beaucoup en Libres maintenant. Les programmeurs, utilisant des Serveurs de Données Libres comme MY SQL SERVER, ne s'embêtent pas à tout centraliser sur un seul serveur.

La version CLUSTER de MY SQL SERVER permet de déporter l'utilisation d'un serveur vers un autre serveur. Cette version est propriétaire. La plupart des développeurs ne vont même pas acheter MY SQL CLUSTER, qui permet de transférer les calculs sur des serveurs à côtés.

Pourquoi la version libre suffit-elle ?

En général les systèmes humains sont décentralisés. En effet le siège est certes à un seul endroit, mais les filiales sont situées ailleurs. Si on veut utiliser son Logiciel, pour l'essentiel de ce qui est demandé, on s'aperçoit qu'on a besoin de plusieurs Logiciels communicant entre eux, possédant des serveurs répartis dans le monde.

3) ORACLE VS LE LIBRE

J'ai déjà vu des entreprises passer de ORACLE à MY SQL SERVER à cause vraisemblablement de frais engagés coûteux. MY SQL SERVER, le serveur de données évolutif, peut aussi être en concurrence avec POSTGRESQL. POSTGRESQL est un serveur de données gérant des données plus complexes que MY SQL, dont celles des Systèmes d'Informations Géographiques.

ORACLE a acheté MY SQL en 2010. Des Forks de MY SQL, donc des clones Libres de MY SQL, ont été créés.

Sachez que beaucoup de Forks existent pour MY SQL. En effet MY SQL appartient maintenant à ORACLE. Or ORACLE diffuse un serveur de données Propriétaire et fermé. Il ne respecte pas la logique du Libre visant à partager toute Source de MY SQL. Vous pouvez tester DRIZZLE ou MARIA DB, des Forks de la communauté MY SQL.

Un serveur de données comme FIREBIRD, permet de créer un Logiciel complexe, non relié à un serveur. Le Logiciel peut ensuite être utilisé à plusieurs. FIREBIRD est le projet Libre du gestionnaire de données INTERBASE de DELPHI.

Les Logiciels sont actuellement si aboutis qu'ils se complexifient facilement. Il est donc intéressant de penser à une utilisation complexe de son Logiciel. Un Logiciel accédant à certains systèmes de données, comme FIREBIRD ou SQLITE, peut fonctionner indépendamment de tout serveur. Les SGBD SQL disposent de fonctions automatisées de calcul.

4) MODÉLISER SON LOGICIEL

Un SGBD, ou Système de Gestion de Bases de Données, peut être modélisé de la même manière que des Objets. Nous vous apprenons ici le diagramme de classes UML. UML est une boîte à outils de diagrammes permettant de modéliser une partie explicite de votre Logiciel.

MERISE permet de modéliser les données. Des outils existent pour créer vos données à partir de modèles MERISE.

UML permet de modéliser la création d'un Logiciel. Nous avons utilisé

des diagrammes de classes UML en exemples.

Voici des outils UML sur LINUX ou WINDOWS :

- KIVIO
- UMBRELLO
- STAR UML
- GAPHOR
- ARGO UML
- BOUML

Il est possible d'installer STAR UML sur LINUX avec WINE DOORS et le paquet PLAYONLINUX, permettant de télécharger les bibliothèques DCOM et MSXML, avec STAR UML. Pour installer WINE DOORS, vous devez disposer d'une licence WINDOWS 32 bits, comme WINDOWS 95 par exemple.

Il existe aussi des Logiciels payants permettant de créer automatiquement du Code Source, ou de créer des données. Il est intéressant, pour gagner du temps, de travailler à la fois avec un outil BORLAND et avec LAZARUS.

5) PAS DE DOUBLONS MAIS DES RELATIONS

Un système de données bien conçu ne possède pas de doublons, sur le serveur qu'il utilise. Contrairement aux simples fichiers qui regroupent tout en leur sein, Les SGDB sont en général Relationnels, ou SGBDR.

Une table, qui pourrait être notre fichier, est reliée à une autre table par une relation. La relation peut être unique, multiple, voire complexe. Le SGBDR va gérer ces relations et les utilisateurs.

6) L'EXEMPLE

Nous voulons répertorier les citoyens de mairies de villages.

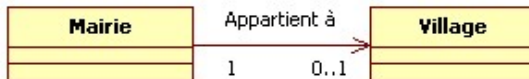
7) PENSER À DÉFINIR

L'informatique gère de l'information. Si vos entités sont mal définies, vous pouvez vous tromper dans la structure de vos données. Une structure de données est difficile à modifier. Il faut en effet garder les données existantes. N'hésitez pas à ajouter des commentaires dans vos modèles !

8) RELATION UNIQUE

Une relation unique est facile à gérer. Des Composants de LAZARUS permettent de créer une relation unique entre deux tables.

Par exemple un "Village" possède une mairie ou n'en possède pas. Le "Village" a droit à 0 ou 1 "Mairie". Tandis qu'une "Mairie" est toujours au sein d'un "Village", qui devient une commune. Une "Mairie" a droit à 1 et 1 seul "Village".

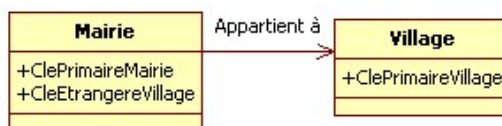


Une et une seule "Mairie" appartient à un "Village"

Un "Village" possède zéro ou une "mairie"

On a l'habitude de nommer les entités au singulier dans tout serveur de données. Si on sépare la "Mairie" du "Village", la table "Village" est logiquement plus grande que la table "Mairie". On évite, grâce au Système de Données Relationnel, de créer trop de doublons "Village", au sein d'une seule table.

On crée donc deux tables : La table "Mairie" et la table "Village". La clé "Village" est cependant dupliquée dans la table "Mairie" par une relation. La relation 1-1 crée une clé étrangère dans la table "Mairie". Les relations sont les uniques doublons au sein des SGBDR.



*Une Mairie appartient à un seul village
Un village contient une seule mairie*

Cette présentation peut être créée automatiquement par votre outil UML.

Il est peu judicieux de créer une clé étrangère au sein de la table "Mairie", car celle-ci est sûrement reliée à un "Village".

La table "Village" est informée sur sa "Mairie", grâce à la table "Mairie". On recherche alors dans la table "Mairie" quelle est la clé étrangère "Village", au sein de la table "Mairie".

Si on compte faire beaucoup de recherches communales pour les villages, il est intéressant de trier, par avance, cette clé étrangère avec la clé "Mairie", toujours triée. Un cluster, au sein d'un SGBD, permet de dupliquer certaines données, comme les clés étrangères, afin de trier par

avance les données des tables.

Il est à noter que si "Village" avait été remplacé par "Commune" la relation unique aurait été celle-ci :



*Une et une seule "Mairie" appartient à une et une seule "Commune"
Une "Commune" possède une seule et une seule "Mairie"*

Ce genre de relation peut aussi se modéliser ainsi :



La table "Commune" possède un attribut "Mairie"

Une table d'une relation unique peut intégrer ses attributs au sein de la table parent. En effet la relation ne possède pas de doublons, car "Mairie" appartient forcément à une même "Commune", en toute réciprocité.

L'attribut "Mairie" pourrait s'appeler aussi "Nom Mairie".

En effet un attribut est une information unique. Il deviendra une variable avec un type au sein du Logiciel. Si l'information est gardée à la fermeture du Logiciel, elle peut être sauvegardée au sein d'un fichier, ou d'un SGBDR.



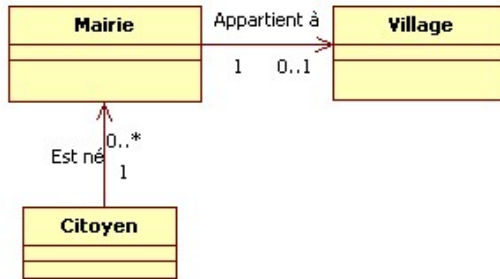
L'attribut "Nom Mairie" est inclus dans la table "Commune"

"Nom Mairie" est une chaîne.

9) RELATION MULTIPLE 1-N

Créons la classe "Citoyen".

Un Citoyen est né dans un seul village.



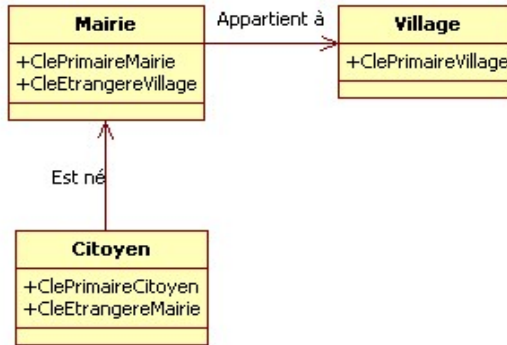
*Un "Citoyen" est né administrativement,
pour une et une seule "Mairie"
Le "Village" a vu naître 0 ou n "Citoyens"*

Nous choisissons le mot "Citoyen" en fonction du contexte choisi.

Nous indiquons ici que le village peut ne pas contenir de "Citoyen". Il est donc possible que le village existe encore s'il n'a plus d'administré.

Il y aura à terme plus de "Citoyens" que de "Mairie". En mettant une clé étrangère "Mairie" dans "Citoyen", nous pouvons dédoubler un

enregistrement "Citoyen", et lui affecter forcément une "Mairie" de naissance.

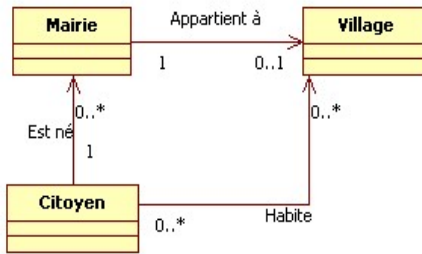


Le modèle de données :
Un citoyen est né administrativement,
dans une "Mairie" dans un "Village"

Une relation multiple 1-N possède le même modèle de données qu'une relation unique. Seulement le programme permettra de créer plusieurs citoyens pour une mairie. En effet il sera possible de dupliquer "Citoyen".

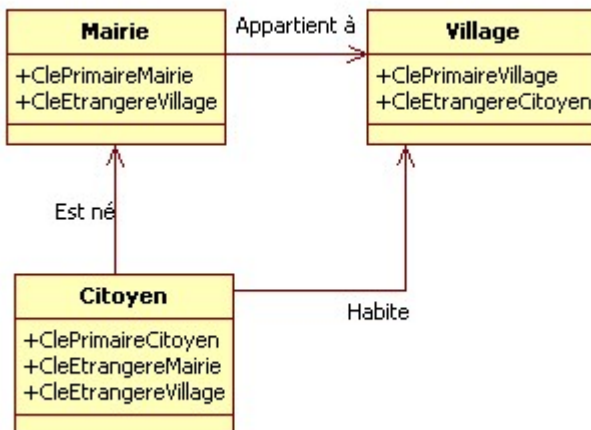
10) RELATION MULTIPLE N-N

Créons une relation entre Village et Citoyen.



Un "Citoyen" "Habite" dans un ou plusieurs "Villages"

Ces relations nécessitent ce genre de structuration de données :



Les clés étrangères de la relation sont dans les tables

On pourrait aussi gérer plus facilement la relation N-N dans une table, entièrement dédiée à la relation. Les clés étrangères seraient aussi la clé primaire multiple de la table. La relation pourrait ainsi être gérée à part. Cela allège les colonnes des deux tables.

Vous pouvez créer votre Composant pour gérer facilement ce genre de relation. Sachez seulement qu'il existe déjà en Libre au sein de

"ManFrames".

a) Exercice

Notez qu'avec ce modèle, on ne sait pas dans quel village le Citoyen est né.

Modifiez le modèle pour savoir dans quel village chaque citoyen est né.

b) Correction

Transposez la relation entre "Citoyen" et "Mairie", et mettez-la entre "Citoyen" et "Village".

Ensuite changez la relation "Appartient à" en relation "Administre", pour créer une relation 1..n sur Village.

La clé étrangère "Village" est supprimée "dans Mairie". La clé étrangère "Mairie" est créée dans "Village".

11) ***RELATION MULTIPLE N-N-N***

Les relations complexes reliant 3 tables ou plus sont rares. La création d'une table relation, de 3 clés étrangères, nécessite que les liens créés soient interdépendants. Sinon créez deux relations N-N. En effet il ne faut pas oublier que vous pouvez retrouver une partie de la relation non affectée. Des clés étrangères nulles alourdiraient la base.

Vous pouvez créer votre Composant pour gérer facilement ce genre de

relation. Il sera inspiré du Composant de relation N-N. Sachez cependant qu'il existe au sein du paquet "ManFrames" sur Internet.

12) CONCLUSION

L'exercice que nous vous avons montré montre l'importance des définitions et du périmètre d'action. Aussi nous pourrions remplacer l'entité "Village" par " Entité urbaine". Cela permettrait d'ajouter les villes pour une future utilisation.

13) BASES DE DONNÉES

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

14) LE POUVOIR DU LIBRE

Des bases de données Libres permettent de sauver beaucoup d'informations utilisateurs. Elles acceptent aussi un grand nombre d'utilisateurs. Elles sont utilisées sur les meilleurs serveurs Web.

Beaucoup de serveurs Web utilisent MY SQL. Intéressez-vous à ses Forks. Les Logiciels complexes utilisent POSTGRESQL.

FIREBIRD et SQLITE sont utilisés en Embarqué, pour des Logiciels prêts à fonctionner. SQLLite est aussi utilisé pour créer des Logiciels Web. FIREBIRD permet de créer des Logiciels Client/Serveur, sur DELPHI et LAZARUS.

15) BASES DE DONNÉES EMBARQUÉES

SQLITE et FIREBIRD permettent de créer des Logiciels possédant une base de données embarquée. Cette base de données embarquée peut être 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.

Dans un Logiciel, il est possible d'envoyer des données et de les réceptionner. Vérifiez 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 LAZARUS et DELPHI.

Q) PROGRAMMATION AVEC FIREBIRD

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Nous allons comprendre la programmation avancée d'une persistance d'informations logicielles.

2) UTILISATION DE TABLES DANS LAZARUS

Les composants ZEOS permettent de gérer des informations persistantes sur un serveur centralisé. ZEOS peut être utilisé avec les contrôles de données commençant par "TDB". ZEOS ne fonctionne en 2010 que sur les architectures 32 bits.

Téléchargez les Composants ZEOS.

Le projet est Libre sur <http://sourceforge.net/projects/zeoslib>.

Installez une version stable sur LAZARUS. LAZARUS redémarre...

3) POINTS FORTS DE FIREBIRD

FIREBIRD permet de créer des Logiciels persistants complexes, en utilisant les contrôles de données, de LAZARUS ou DELPHI.

FIREBIRD est utile pour créer des Logiciels Client/Serveur. Un Logiciel Client/Serveur va directement chercher les informations sur un serveur de données. Il est possible de gérer beaucoup d'utilisateurs sur FIREBIRD.

FIREBIRD est un SGBD SQL. Il possède donc un Système de Langage de Requêtes normalisé permettant son utilisation par LAZARUS, ou tout EDI avec accès SQL.

FIREBIRD est un SGBD multiplate-forme. Il stocke les informations d'une base de données dans un seul fichier. Il n'est donc pas dépendant du système de fichiers d'un système d'exploitation.

FIREBIRD peut être utilisé, avec, ou indépendamment, d'un serveur centralisé. On appelle FIREBIRD sans serveur FIREBIRD EMBEDDED.

FIREBIRD permet de gérer des données d'entreprise. C'est un Système de Gestion de Base de Données reconnu. Ce SGBD gère avec fiabilité les informations qu'il stocke grâce :

- Aux relations permettant de lier des entités d'informations.
- Aux TRIGGERS vérifiant les relations.
- Aux fonctions permettant de préparer les informations.
- Aux tris préparés avec les CLUSTERS.

4) POINTS FAIBLES D'UN SGBD SQL

Un SGBD SQL n'est pas utile pour :

- Les applications Web peu gourmandes.
- Renvoyer beaucoup d'informations.
- Faire peu de mises à jour sur les données.

Si vous voulez créer des applications Web, préférez les outils NoSQL, utilisés sur des serveurs Web. Le NoSQL permet de faire ce que ne fait pas un SGBD SQL.

Les points faibles des SGBD SQL sont les atouts des outils NoSQL. SQLite peut servir d'intermédiaire pour des applications Web. Il est allégé, comme les outils NoSQL. Mais il peut être utilisé facilement avec LAZARUS, comme il utilise le SQL.

5) RÉCAPITULATIF

Dans un système de données élaboré seules les relations sont en doublons dans des clés étrangères. Ainsi le système est facilement gérable.

6) INSTALLATION DE FIREBIRD

FIREBIRD peut être géré par un serveur grâce à l'utilitaire "flamerobin".

a) FIREBIRD sur WINDOWS

Téléchargez une version stable de FIREBIRD sur www.firebirdsql.org. Vérifiez si la version à télécharger est disponible dans le Composant "TZConnection", que vous avez installé sur LAZARUS.

Téléchargez et installez "flamerobin" sur www.flamerobin.org. Copiez maintenant les "dll" de FIREBIRD du répertoire "bin" dans chaque nouveau projet ZEOS.

b) FIREBIRD sur LINUX

Allez dans le gestionnaire de paquets et sélectionnez les paquets "firebird-server", "firebird-dev", "libfbclient", "flamerobin".

Créez avec l'utilisateur "root" ce lien symbolique dans le terminal :

```
ln -s /usr/lib/libfbclient.so.n.n.n /usr/lib/libfbclient.so
```

Les "n" sont à remplacer par les numéros de version de FIREBIRD installé. Ce shell est à placer dans le futur paquet LINUX de votre Logiciel.

Il y a une protection des droits. Ajoutez "firebird" à votre compte dans le terminal en tant que "root" :

```
usermod -G firebird moncomptelinux
```

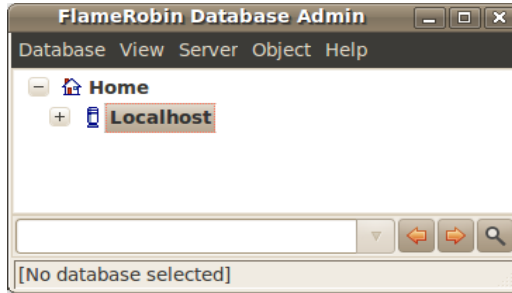
7) CRÉATION D'UNE TABLE FIREBIRD

Vous pouvez créer des nouveaux utilisateurs grâce à cette commande :

```
gsec -user SYSDBA -pass masterkey -add nouvelutilisateur -pw motdepasse
```

On voit ici que le mot de passe administrateur par défaut de firebird est "masterkey".

Démarrez "flamerobin". Enregistrez le serveur "localhost", sans renseigner le port.



Fenêtre "flamerobin"

Une fois le serveur enregistré, cliquez sur le bouton droit de souris, puis cliquez sur "Create new database".

Affectez, pour "Display Name", le nom de fichier de base que vous avez sélectionné. Renseignez le nom d'utilisateur que vous avez créé avec son mot de passe.

Le jeu de caractère français est "ISO8859_1". Vous pouvez aussi utiliser le jeu de caractère international "UTF8". Le reste n'est pas à changer.

a) Sur LINUX

Sur LINUX il y a une sécurité contre les virus. Vous devez créer votre base de données dans le répertoire "/var/lib/firebird/n.n/data", en affectant l'utilisateur "firebird".

Tapez ceci dans le terminal avec l'utilisateur "root" :

```
cp chemin_database /var/lib/firebird/n.n/data
chown firebird.root /var/lib/firebird/n.n/data/madatabase.fdb
```

Les "n" sont à remplacer par le numéro de version "firebird".

Redémarrez "flamerobin"

Ajoutez la base de données à relier.

Vous pouvez accéder à votre base de données.

8) **CRÉER UNE TABLE**

Dans votre base de données Libre, vous voyez l'ensemble des possibilités offertes par FIREBIRD.

Cliquez, avec le bouton droit de la souris, sur "Tables", puis "Create New". Vous voyez alors s'afficher la sémantique à respecter.

Effacez le Code puis remplacez-le par ceci :

```
CREATE TABLE UTILISATEUR
(
CLEP INTEGER NOT NULL,
NOM VARCHAR(100),
PRENOM VARCHAR(100),
PRIMARY KEY (CLEP)
);
```

Validez avec la case à cocher du gestionnaire de requête.

Nous créons la table "UTILISATEUR", possédant les champs "CLEP", "NOM", "PRENOM". "NOM" et "PRENOM".

Ces informations sont des chaînes de 100 caractères maximum.

"CLEP" est un entier mais est aussi la clé primaire.

Créez le générateur de clés :

```
CREATE GENERATOR gen_t1_id ;  
SET GENERATOR gen_t1_id TO 0;
```

Ce générateur permet de créer en série une clé auto-incrémentée.
Validez avec la case à cocher du gestionnaire de requête.

Utilisons ce générateur de clé dans un "TRIGGER", se déclenchant à l'enregistrement des données :

```
set term !! ;  
CREATE TRIGGER UTILISATEUR_GEN FOR UTILISATEUR  
ACTIVE BEFORE INSERT POSITION 0  
AS  
BEGIN  
if (NEW.CLEP is NULL) then NEW.CLEP  
=GEN_ID(GEN_T1_ID, 1);  
END!!  
set term ; !!
```

Validez avec la case à cocher du gestionnaire de requête.

Nous avons créé notre première table. Nous allons maintenant y accéder.

Pour revenir en arrière voici les lignes à taper :

```
DROP TABLE UTILISATEUR ;  
DROP GENERATOR gen_t1_id;  
DROP TRIGGER UTILISATEUR_GEN;
```

Validez la transaction sur vos mises à jour.

9) **EXERCICE**

Sauvegardez votre dernier exercice avec les tables DBF.

Nous allons utiliser cet exercice pour créer une véritable Application liée à un serveur de données.

a) **Une connexion centralisée**

Ajoutez le Composant ZEOS "TZConnection" pour le renommer en "Connection".

Renseignez la propriété "Database" avec le chemin de votre fichier de base de données.

Renseignez la propriété "Hostname" avec "localhost". C'est l'adresse du serveur de données. Elle n'est pas à renseigner si on utilise FIREBIRD EMBED.

Renseignez la propriété contenant les SGBD avec la version de votre "firebird". FIREBIRD EMBED contient deux "d". Sélectionnez "FIREBIRD" avec un seul "d".

Testez la connexion en mettant "Connected" à "True".

b) **La Source de données table**

Supprimez le Composant de type "TDBFDataSet". Ajoutez le

Composant ZEOS "TZTable" pour le renommer en "ZUtilisateur".

On a l'habitude de nommer les entités au singulier dans tout serveur de données. Le Composant "TZTable" est un descendant de "TDataSet". Il peut être utilisé facilement au sein de LAZARUS.

Affectez "Connection" à la propriété "Connection" de "ZUtilisateur". Renseignez sa propriété "TableName" avec le nom de la table : "UTILISATEUR".

Nous allons enfin créer un vrai lien de données LAZARUS. Dans la palette "Data Access", sélectionnez le Composant "TDataSource", et placez le sur la fiche.

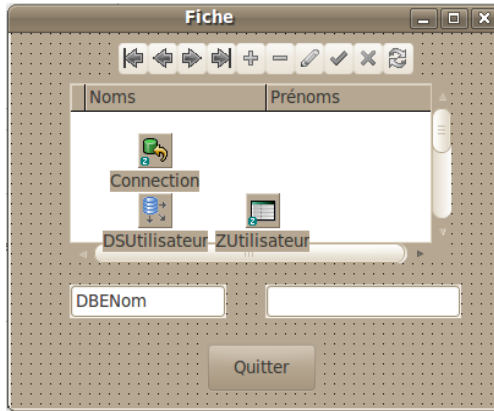
Le "TDataSource" permet de relier tout "TDataSet" à tout contrôle de données. Renommez-le en "DSUtilisateur".

Affectez "ZUtilisateur" à la propriété "Dataset" de "DSUtilisateur". Dans le "DBGrid" affectez "DSUtilisateur" à sa propriété "DataSource".

Placez ce Code Source pour l'événement "OnShow" :

```
procedure TMaFiche.FormShow(Sender: TObject);  
begin  
  try  
    Connection.Connected := True;  
    ZUtilisateur.Open;  
  Except  
    on E:Exception do Showmessage ( 'Impossible de se connecter à la  
Source de données :'+#13#10+E.Message );  
  end;  
end;
```

Voici ci-après le résultat pour la fiche :



Le "Datasource" est à côté de son "Dataset" sur la grille

10) **LES RELATIONS**

Nous allons créer une gestion d'onglets, avec un deux onglets : L'un est la fiche déjà créée, l'autre est une fiche "Département", permettant de grouper les utilisateurs.

Dans "flamerobin" sur votre base de données ouverte, nous allons créer la table département.

Cliquez droit sur "Tables" puis "Create New". Vous voyez alors s'afficher la sémantique à respecter.

Effacez le Code puis remplacez-le par ceci :

```
CREATE TABLE DEPARTEMENT
(  

CLEP INTEGER NOT NULL,  

NOM VARCHAR(100),  

PRIMARY KEY (CLEP)  

);
```

Validez avec la case à cocher du gestionnaire de requête.

Créez le générateur d'Id :

```
CREATE GENERATOR gen_t2_id ;  
SET GENERATOR gen_t2_id TO 0;
```

Validez avec la case à cocher du gestionnaire de requête.

Ce générateur permet de générer une clé auto-incrémentée.

Voici le "TRIGGER" créant la clé dans l'enregistrement :

```
set term !! ;  
CREATE TRIGGER DEPARTEMENT_GEN FOR  
DEPARTEMENT  
ACTIVE BEFORE INSERT POSITION 0  
AS  
BEGIN  
if (NEW.CLEP is NULL) then NEW.CLEP  
=GEN_ID(GEN_T2_ID, 1);  
END!!  
set term ; !!
```

Validez avec la case à cocher du gestionnaire de requête.

Nous avons vu dans le chapitre précédent les relations entre les tables.

Le département contient n utilisateurs. Un utilisateur appartient à un département. Nous allons donc créer une clé étrangère dans la table "UTILISATEUR" :

```
ALTER TABLE UTILISATEUR ADD IDDEPARTEMENT  
INTEGER;
```

Validez la transaction.

Dans cette configuration, si on efface un département, la table utilisateur voit la clé étrangère du département rester intacte. Nous ne respectons alors pas l'unicité de notre base de données.

De même il est possible que la clé du "DEPARTEMENT" soit mise à jour, ce qui n'est pourtant pas recommandé. En effet une clé ne doit jamais se modifier. Créons tout de même deux "TRIGGERS" permettant de respecter l'unicité.

Voici deux "TRIGGERS" mettant à jour la table "UTILISATEUR", à partir de la table "DEPARTEMENT". L'un se déclenche à l'effacement de tout enregistrement de "DEPARTEMENT", l'autre à la mise à jour de tout enregistrement de "DEPARTEMENT".

```
SET TERM | ;  
CREATE TRIGGER  
DELETE_DEPARTEMENT_UTILISATEUR FOR  
DEPARTEMENT  
ACTIVE BEFORE DELETE POSITION 0  
AS  
BEGIN  
UPDATE UTILISATEUR SET  
UTILISATEUR.IDDEPARTEMENT = NULL WHERE  
IDDEPARTEMENT = OLD.CLEP;  
END|  
CREATE TRIGGER  
UPDATE_DEPARTEMENT_UTILISATEUR FOR  
DEPARTEMENT  
ACTIVE BEFORE UPDATE POSITION 0  
AS  
BEGIN  
IF (NEW.CLEP <> OLD.CLEP) THEN  
BEGIN  
UPDATE UTILISATEUR SET  
UTILISATEUR.IDDEPARTEMENT = NEW.CLEP WHERE
```

```
IDDEPARTEMENT = OLD.CLEP;  
END  
END|  
SET TERM ; |
```

Validez la transaction.

Il est possible de changer le caractère de fin de ligne choisi par "SET TERM". Le principal est de respecter scrupuleusement la syntaxe demandée.

Nous avons créé notre deuxième table. Nous allons maintenant y accéder.

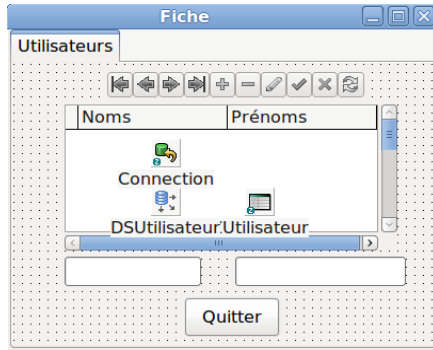
Créez un deuxième projet à partir de l'exercice précédent, en copiant le répertoire contenant l'exercice.

Sélectionnez tous les Composants visibles de la fiche, en maintenant enfoncée la touche "Shift". Cliquez à droite de la souris sur un Composant, puis "Coupez". Les Composants se cachent.

Ajoutez un "TPageControl" de la palette "Common Controls". Affectez "alClient" à sa propriété "Align". Cliquez à droite de la souris sur l'onglet, puis "Ajoutez une page". Cliquez sur la page ajoutée puis "Coller".

Renommez le nom de "Tabsheet1" en "TabUtilisateurs". Modifiez la propriété "Caption" en lui affectant "Utilisateurs".

Voici ci-après le résultat :



Nous avons placé notre fiche dans un onglet

De la même manière créez l'onglet "Départements" à partir de l'onglet "Utilisateurs".

Maintenant, le bouton "Quitter" est dans chaque onglet. Au lieu de placer à chaque fois le bouton "Quitter" dans chaque onglet, nous allons le placer en haut.

Nous allons maintenant dupliquer les informations de l'onglet pour créer un deuxième onglet.

Sélectionnez tous les Composants visibles de l'onglet, excepté un "TEdit" et le bouton "Quitter".

Créez un deuxième onglet. Cliquez sur la page ajoutée et "Collez". Nommez la page "TabDepartement" avec pour "Caption" "Département".

Nous ne voyons plus le bouton "Quitter", qui sert à quitter la fiche. Il n'est nécessaire que pour la fiche, pas pour chaque onglet.

Cliquez sur les onglets. Affectez "alBottom" à la propriété "Align" du "TPageControl".

Redimensionnez-le pour voir un peu le fond du formulaire. Ajoutez un "TPanel" sur le formulaire.

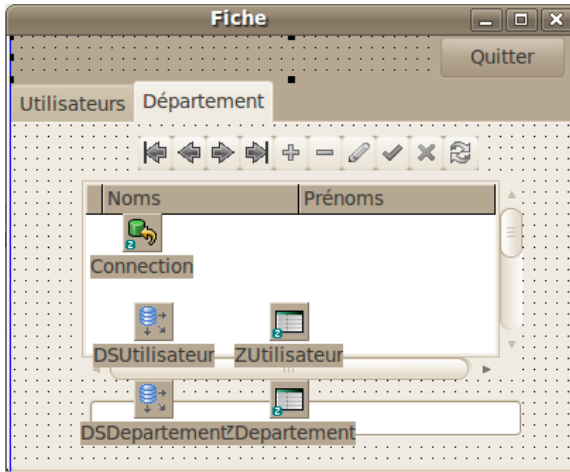
Affectez sur le "TPanel" "alClient" à la propriété "Align", et "bvNone" à "BevelOuter".

Puis Affectez "PanBoutons" à son nom, et rien à "Caption". Cela permet de ne pas voir le "TPanel".

Coupez-collez le "TToggleButtons" vers le "TPanel".

Copiez-collez sur le même formulaire les Composants "DSUtilisateurs" et "ZQUtilisateurs".

Voici ci-après le résultat :



La fiche avec les départements

Une fois que ce nouveau "Dataset" est relié, puis ouvert, vous avez alors créé une véritable Application centralisée, avec des liens entre vos fiches. Ce genre de Logiciel est robuste. Il nécessite moins de tests que des applications non RAD ou Web.

11) CONCLUSION

Vous pouvez adapter le Logiciel à la taille de fenêtre avec des "TPanel".

FIREBIRD est très utile pour créer des applications robustes Client/Serveur . Il permet grâce au SQL et aux "TRIGGERS" de respecter l'unicité d'applications Client/Serveur.

Il est maintenant intéressant d'utiliser SQLite pour les applications Web. Il est possible d'utiliser ZEOS ou SQLDB pour accéder à SQLite.

R) LE WEB : EXTPASCAL ET SQLITE

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

EXTPASCAL est en 2011 en pleine évolution. Ce Framework vous permet de créer des Logiciels réactifs en Web 2.0. Le Web 2.0 est un Web participatif dans lequel l'utilisateur peut facilement agir.

LAZARUS est un EDI permettant de créer des exécutables pour une ou plusieurs plates-formes. Il est possible avec LAZARUS de programmer sur une plate-forme, pour mettre en place sur une autre.

Il est recommandé de créer des exécutables Web pour des systèmes UNIX, comme LINUX ou MAC-OS. En effet les systèmes UNIX sont mieux sécurisés que WINDOWS. LINUX est libre et gratuit.

MICROSOFT possède un vieil environnement graphique n'ayant pas assez évolué en tant que serveur Web. Il est recommandé de supprimer de nombreux services WINDOWS, afin de sécuriser un serveur WINDOWS. Cela n'empêche pas votre serveur de crasher à cause d'un virus informatique.

LINUX et UNIX vous permettent de gérer votre serveur sans interface graphique. Cela permet d'allouer plus de mémoire, pour le service à l'utilisateur.

2) LE WEB

Le HTML est le langage de présentation de vos navigateurs Web. Les traitements de textes vous permettent de créer des pages Web. Le Javascript va donner du mouvement à vos pages Web. EXTPASCAL automatise cette création Web. En général vous n'avez pas de HTML ni de Javascript à créer.

3) LES FASTCGI

La technologie des CGI, puis des FastCGI, est utilisée par LAZARUS pour créer des applications Web. C'est une technologie permettant une réponse quasi immédiate.

Les CGI et FastCGI LAZARUS sont des exécutables d'un serveur Web comme APACHE. Elles renvoient une voire plusieurs pages Web lisibles dans un navigateur Web.

Les CGI vous permettent de programmer de simples pages retournant votre résultat.

Les FastCGI sont une amélioration réussie des CGI, permettant de réaliser un véritable Logiciel. Leur communication est différente et optimisée.

La FastCGI est une technologie qui est utilisée pour traduire les pages PHP en HTML. Si vous utilisez LAZARUS pour vos FastCGI, vous n'utilisez pas d'intermédiaire, donc vos Logiciels Web ont du répondant. L'utilisation des CGI et FastCGI est accessible avec LAZARUS. Il suffit de sélectionner votre "Nouveau" projet afin de programmer ce que vous souhaitez.

4) ***EXTPASCAL***

EXTPASCAL avec EXTJS vous permet de créer des exécutables Web fonctionnant avec du Javascript.

Le Javascript de EXTJS permet de faire croire à de véritables formulaires LAZARUS, dans votre navigateur Web. Vous créez des formulaires LAZARUS, transformés ensuite en HTML et en événements Javascript.

Contrairement à Google Web Toolkit, un kit de traduction de Code Java en Javascript, une partie du Code Source créé est réellement exécutée. Vous pouvez ainsi améliorer plus facilement vos projets EXTPASCAL.

5) ***FIABILITÉ D'UN SERVEUR LAZARUS***

La fiabilité d'un serveur Web dépend de la destruction complète des Objets créés par votre Application. Si un seul Objet n'est pas détruit, votre serveur peut alors planter, et doit être redémarré.

Il est très intéressant de créer des tests de mémoire, permettant de savoir si un serveur Web est fiable. En effet un serveur Web UNIX doit pouvoir fonctionner un an sans aucun plantage. N'oubliez pas de tester les compteurs de temps sur leurs limites !

Il est donc très facile de faire régresser un serveur LAZARUS. Vous pouvez tester la mémoire à la fin du chapitre sur la **Programmation Procédurale avancée**.

Le Web : EXTPASCAL et SQLITE - Fiabilité d'un serveur LAZARUS

6) **SQLITE**

SQLITE est très allégé. Il n'y a pas besoin de le gérer en tant que service.

C'est votre FastCGI qui gère les données. Si votre FastCGI n'est pas sur plusieurs serveurs en même temps SQLITE suffit. Une FastCGI sur plusieurs serveurs permet cependant de faire face à l'afflux de visiteurs.

7) **INSTALLER EXTPASCAL**

Téléchargez "Orpheus For LAZARUS" et "EXTPASCAL" grâce à votre navigateur Web. Vérifiez vos Sources grâce aux adresses Web et au référencement.

Les liens en 2011 sont ceux-ci :

http://sourceforge.net/projects/tporpheus/files/tporpheus_docs/
<http://code.google.com/p/extpascal>

Décompressez les fichiers et enlevez le numéro de version inscrit dans les répertoires. Cela permet de mettre à jour vos Composants plus facilement.

Installez le paquet "orpheus.lpk" sans redémarrer LAZARUS.

Puis renommez vos répertoires, afin de n'avoir aucun espace jusqu'aux paquets à installer.

Puis installez dans EXTPASCAL les 3 paquets du "Toolkit" : Les contrôles ou "Ctrls", la grille ou "Grid" et le projet ou "Proj".

8) **INSTALLER UN SERVEUR EXTPASCAL**

a) **Sur WINDOWS**

Téléchargez "WAMP" et installez-le, si vous ne l'avez pas déjà fait. Créez le répertoire "C:\wamp\www\cgi" et ajoutez-y la DLL SQLITE. Ce répertoire sera à affecter à la place de "/var/www/cgi/cgiext".

Téléchargez le module Fast CGI sur <http://www.fastcgi.com>. Ajoutez le module dans "C:\wamp\bin\apache\ApacheN.N.N\modules". Renommez l'extension DLL en so, comme pour le reste des modules.

Dans le fichier httpd.conf, dans "C:\wamp\bin\apache\ApacheN.N.N\conf", ajoutez ceci après la série les LoadModule :

```
LoadModule fastcgi_module modules/mod_fastcgi.so
```

Cela ajoute le module fastcgi dans la configuration d'APACHE.

APACHE est émulé. Vous devez remplacer le caractère de changement de répertoire "\" par "/". Regardez bien comment sont orthographiés les répertoires, dans le fichier de configuration APACHE.

b) Sur LINUX

Voici une installation sur LINUX, un environnement gratuit et Libre. Si vous voulez installer ce genre de serveur fiable, vous pouvez contacter votre GUL local. Un GUL est un Groupe d'Utilisateurs LINUX.

Allez dans le gestionnaire de paquet, et téléchargez le paquet "apache2".

Nous allons installer le module APACHE FastCGI. Avec votre gestionnaire de paquets installez "libapache2-mod-fastcgi", ou tapez en mode administrateur :

```
apt-get install libapache2-mod-fastcgi
```

Mode administrateur

Le mode administrateur est soit enclenché par la commande "sudo su", soit par la commande "su root".

Le nom de module peut changer. Prenez alors le plus proche, celui contenant "fastcgi" pour APACHE.

Le module fastcgi est alors activé sur APACHE.

Téléchargez le module EXTJS sur <http://www.extjs.com>, en vérifiant la compatibilité de la version avec EXTPASCAL. Vous devez télécharger la bonne version de EXTJS, par rapport à celle demandée par EXTPASCAL.

Mettez les sous-répertoires "extjs" dans le répertoire "/ext", à la racine de votre site Web.

La commande LINUX qui permet de copier est celle-ci :

```
cp source destination
```

Il suffit de faire un glisser-déplacer vers le terminal pour placer les chemins "source" et "destination".

La racine de votre site Web est à "/var/www". L'utilisateur doit être "www-data", ou bien le groupe doit être "www-data".

Attention !

Voici la commande à exécuter en administrateur :

```
chown -R www-data.www-data /var/www
```

Cette commande, qui permet de changer le propriétaire du dossier et de ses fichiers, est à exécuter à chaque fois que vous n'avez plus accès à votre exécutable, lorsque vous exécutez votre FastCGI.

Le gestionnaire EXTPASCAL vous permet de contrôler votre projet EXTPASCAL.

Avec LAZARUS, ouvrez le projet "ExtPascalSamples", dans le répertoire "ExtPascalSamples". Compilez le projet pour votre serveur, avec la bonne plate-forme.

Si le projet ne compile pas vous avez peut-être à ajouter "../" dans les "Autres fichiers unités" du Compilateur. Allez dans "Projet", puis "Options du Projet", puis "Options de compilation", afin de régler ce problème.

Copiez l'exécutable du projet dans le fichier "/var/www/cgi/cgiext".

Créez le répertoire des CGI. Utilisez la commande LINUX "cp", permettant de copier grâce au terminal, en mode administrateur :

Le Web : EXTPASCAL et SQLITE - Installer un serveur EXTPASCAL

```
mkdir /var/www/cgi  
cp repertoire-en-glissé-déplacé /var/www/cgi/cgiext
```

9) **CONFIGURER LE SERVEUR FASTCGI**

Dans vos Composants EXTPASCAL, copiez l'intérieur du répertoire "codepress", à la racine de vos pages Web dans le répertoire "www". Sur LINUX, ajoutez la ligne du serveur CGI externe dans le fichier "/etc/apache2/mods-enabled/fastcgi.conf", grâce à cette commande en mode administrateur :

```
gedit /etc/apache2/mods-enabled/fastcgi.conf
```

WINDOWS

Sur WINDOWS, ajoutez le chemin complet entre guillemets, avec le caractère "/" remplaçant le caractère "\".

Sur WINDOWS toujours, ajoutez la configuration ci-dessous, à la fin du fichier "C:\wamp\bin\apache\ApacheN.N.N\conf\httpd.conf".

La partie FastCGI, de LINUX ou WINDOWS, ressemble à celle-ci :

```
<IfModule mod_fastcgi.c>  
# AddHandler fastcgi-script .fcgi # Sur LINUX  
# AddHandler fastcgi-script .exe # Sur WINDOWS  
#FastCgiWrapper /usr/lib/apache2/suexec  
# FastCgiIpcDir /var/lib/apache2/fastcgi # Sur LINUX  
fastcgiexternalserver /var/www/cgi/cgiext -host localhost:2014 -idle-  
timeout 5  
</IfModule>
```

Vous voyez que votre CGI peut porter le nom "cgiext.cgi" sous UNIX,

ou "cgiext.exe" sous WINDOWS.

Remplacez "mon.adresse.ip" par l'adresse IP de votre machine, par le nom de domaine ou bien par localhost.

Créez le fichier "/etc/apache2/cgiext.conf", ou "C:\wamp\bin\apache\ApacheN.N.N\conf\cgiext.conf", sur WINDOWS grâce à cette commande en mode administrateur :

```
gedit /etc/apache2/cgiext.conf
```

Sur WINDOWS créez et éditez ce fichier texte.

Placez-y cette configuration APACHE :

```
<Directory /var/www/cgi>
    UseCanonicalName off
    Options +ExecCGI +Indexes +FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

Ce fichier indique que nous exécutons des CGI dans le répertoire /var/www/cgi.

Dans le fichier "/etc/apache2/apache2.conf" cherchez la ligne "Include httpd.conf", et ajoutez après cette ligne :

```
Include cgiext.conf
```

Vous pouvez maintenant sauvegarder votre travail. Faites une copie de tous les fichiers ajoutés.

Tapez ceci en administrateur :

```
/etc/init.d/apache2 restart
```

Le Web : EXTPASCAL et SQLITE - Configurer le serveur FastCGI 263

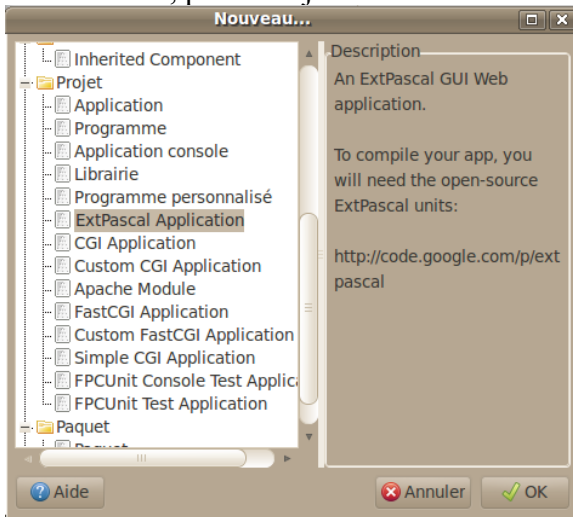
Avec FIREFOX allez sur l'adresse IP de votre serveur en tapant dans la barre d'adresse :

`http://localhost/cgi/cgiext`

Votre Logiciel FastCGI démarre.

10) L'EXEMPLE

Créez une nouvelle Application EXTPASCAL. Pour ce faire allez dans "Fichier", puis "Nouveau", puis "Projet" "EXTPASCAL Application".



Créez une application "EXTPASCAL"

Vous avez une nouvelle fiche EXTPASCAL. Modifiez les propriétés "Maximizable", "Maximized" et "Plain" à "True". Modifiez la propriété "Resizable" à "False". Nous changeons la présentation du formulaire FastCGI.

Nous allons remplir le formulaire avec un gestionnaire d'onglets. Ajoutez avec la palette "EXTPASCAL" un "TextTabPanel", et adaptez-le à la fiche.

"Ajoutez une page" à votre "TextTabPanel". Donnez-lui le nom de "Utilisateurs".

Nous allons remplir la moitié de l'onglet "Utilisateurs" avec une grille. Ajoutez le Composant "TextGridEditorGridPanel", et adaptez-le à la fiche.

Ajoutez deux cellules de texte "TextFormTextField", deux labels "TextFormLabel", deux zones de texte "TextFormTextField". Les premières pour le "Nom", les secondes pour le prénom.

Ajoutez un bouton EXTPASCAL "Ajouter", un bouton "Supprimer", un bouton "Ouvrir".

Créez deux colonnes dans la grille grâce à "ColCount".

Dans "Columns", associez chaque colonne, la première vers la cellule du Nom, la deuxième vers la cellule du Prénom.

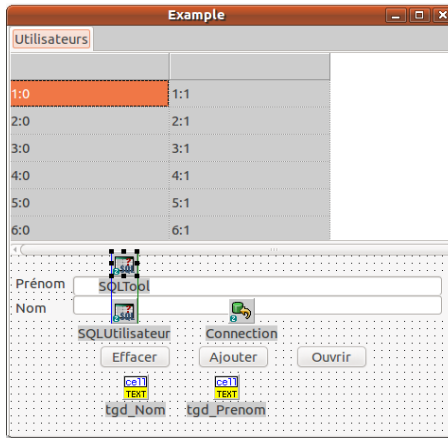
Dans la cellule du Nom, mettez "DataIndex" à 0, car c'est la première colonne de la grille.

Dans la cellule du Prénom, mettez "DataIndex" à 1, car c'est la deuxième colonne de la grille.

Ajoutez les queries ZEOS, et nommez les "SQLTool" et "SQLUtilisateur".

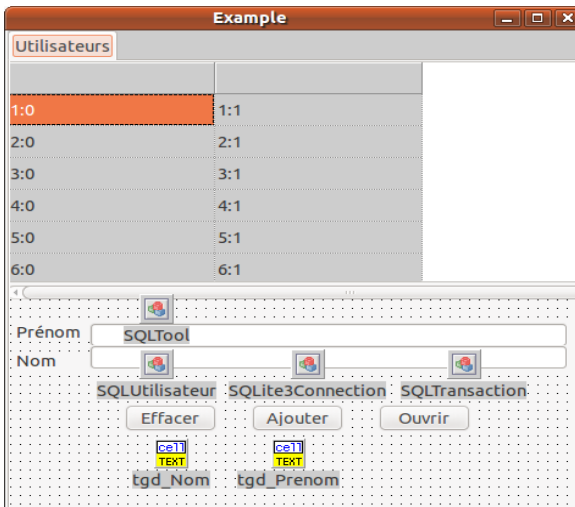
Ajoutez une connexion ZEOS et nommez la "Connection". Reliez les queries SQLDB à cet connexion.

Voici ci-après ce que cela donne :



Les liens de données ZEOS

Vous pouvez aussi ajouter des composants SQL DB avec un composant de transaction :



Le premier exemple EXTPASCAL

11) ***COMPILER LA FASTCGI***

Nous utilisons la librairie EXTJS pour afficher les fenêtres, dynamiquement.

Dans le "toolkit" ou boîte à outils, Ouvrez le projet "fmtoextp" puis compilez le projet.

Dans un terminal, ou dans la ligne de commande, copiez-collez le lien vers l'exécutable créé, puis après un espace copiez-collez le lien vers votre fichier "lfm", puis un espace, puis copiez-collez le lien vers votre fichier projet "lpi".

```
fmtoextp copié-collé-du-lfm copié-collé-du-lpi
```

Cette commande peut être exécutée automatiquement, à chaque fois que vous compilez votre FastCGI.

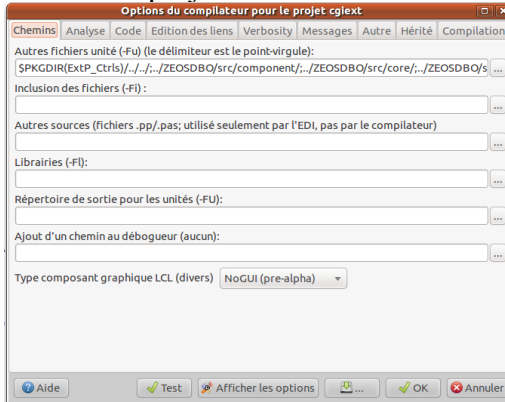
Vous pouvez compiler votre FastCGI.

12) ***LIER AUX DONNÉES***

Avant de lier aux données, sachez que LAZARUS 0.9.28 compile par défaut avec beaucoup trop d'unités. En effet, par défaut, LAZARUS 0.9.28 ajoute le paquet "LCL", qui ajoutant toutes les unités "LCL". Nous vous proposons dans l'exemple de supprimer le lien vers le paquet LCL.

13) CONFIGURER ZEOS

Mais vous pouvez aussi lier votre projet directement aux unités, afin d'utiliser ZEOS. ZEOS est beaucoup plus évolué que SQL DB. Pour lier votre projet directement aux unités, ajoutez des chemins dans les "Options de compilation" du projet.



Les liens vers ZEOS DBO, mis dans le répertoire connexe au projet

Voici les chemins qu'on rajoute :

- Laisser \$PKGDIR(ExtP_Ctrls)/.././
- ../ZEOSDBO/src/component/
- ../ZEOSDBO/src/core/
- ../ZEOSDBO/src/dbc/
- ../ZEOSDBO/src/parsesql/
- ../ZEOSDBO/src/plain/

Avec LAZARUS 0.9.28, cette façon de lier permet d'économiser des unités.

14) CONFIGURER SQLDB

Avant d'utiliser SQLDB il faut le délier du paquet LCL. Sinon LAZARUS 0.9.28 va lier toutes les unités de la LCL.

Cette manœuvre ne change en rien la compatibilité du paquet "sqldbblaz".

Voici comment procéder.

Allez dans le répertoire de LAZARUS dans "components" puis "sqldb".

Créez dedans le répertoire "reg".

Déplacez-y les fichiers "sqlstringspropertyeditordlg/*" et "registersqldb/*".

Créez le paquet regsqldb.lpk dans ce répertoire.

Ajoutez-y les deux unités précédemment déplacées.

Créez le fichier "sqldbblazminireg.pas" dans ce répertoire.

Déplacez dans cette unité la procédure "RegisterUnitSQLdb". Créez une procédure d'enregistrement nommée "Register".

Déplacez dans l'exécution de cette procédure l'enregistrement de la procédure "RegisterUnitSQLdb".

Voilà ce à quoi ressemble l'unité "sqldbblazminireg.pas" :

```
unit sqldbblazminireg;  
{ $mode objfpc } { $H+ }
```

```
{$IFDEF win64}  
{$DEFINE HASMYSQL4CONNECTION}  
{$DEFINE HASORACLECONNECTION}  
{$DEFINE HASPQCONNECTION}  
{$ENDIF}
```

```
{ SQLITE }  
{$IFDEF WIN64}  
{$IFDEF VER2_2_0}  
{$DEFINE HASSQLITE3CONNECTION}  
{$ENDIF}  
{$ENDIF}
```

```
{ SQLSCRIPT }  
{$IFDEF VER2_2_0}  
{$DEFINE HASSQLSCRIPT}  
{$ENDIF}
```

interface

uses

Classes,
SysUtils;

procedure Register;

implementation

uses LazarusPackageIntf,
ibconnection,
{\$IFDEF HASPQCONNECTION}
pqconnection,
{\$ENDIF}

```

{SIFDEF HASORACLECONNECTION}
    oracleconnection,
{SENDIF}
{SIFDEF HASMYSQL4CONNECTION}
    mysql40conn, mysql41conn,
{SENDIF}
{SIFDEF HASSQLITE3CONNECTION}
    sqlite3conn,
{SENDIF}
    mysql50conn,
    sqlldb, odbconn;

procedure RegisterUnitSQLdb;
begin
    RegisterComponents('SQLdb',[TSQLQuery,
        TSQLTransaction,
{SIFDEF HASSQLSCRIPT}
        TSQLScript,
        TSQLConnector,
{SENDIF}
{SIFDEF HASPQCONNECTION}
        TPQConnection,
{SENDIF}
{SIFDEF HASORACLECONNECTION}
        TOracleConnection,
{SENDIF}
        TODBCConnection,
{SIFDEF HASMYSQL4CONNECTION}
        TMySQL40Connection,
        TMySQL41Connection,
{SENDIF}
{SIFDEF HASSQLITE3CONNECTION}
        TSQLite3Connection,
{SENDIF}

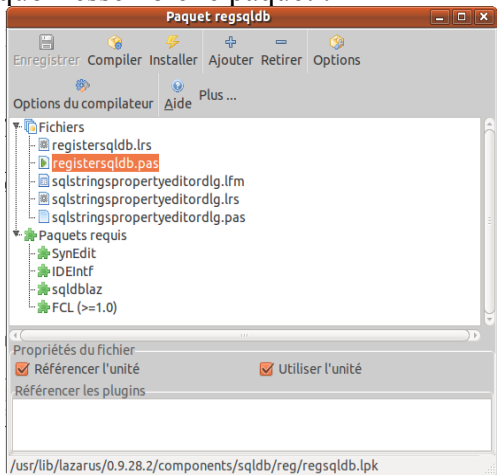
```

```
        TMySQL50Connection,  
        TIBConnection]);  
  
end;  
  
procedure Register;  
Begin  
    RegisterUnit('sqldb',@RegisterUnitSQLdb);  
End;  
  
end.
```

Ajoutez cette unité dans le paquet "sqldbblaz". Il ne doit rester que l'unité "sqldb" et l'unité "sqldbblazminireg.pas".

Dans le paquet "sqldbblaz.lpk", après avoir noté les liens, supprimez tous les liens vers les autres paquets. Ajoutez le paquet "FCL". Dans le paquet "regsqldb.lpk", ajoutez les liens effacés vers les paquets. Ajoutez le lien vers l'unité "sqldbblaz".

Voici ci-après à quoi ressemble le paquet :



Le nouveau paquet "regsqldb.lpk"

Après avoir compilé le paquet, installez-le dans LAZARUS. Recompilez LAZARUS.

Vous pouvez maintenant utiliser SQLDB dans EXTPASCAL avec l'EDI.

15) **SQLITE**

SQLITE permet de gérer les données de votre site Web, en supprimant les fonctions et les triggers. En effet votre Application Web peut gérer vos données, car le serveur Web peut accéder à vos bases sans concurrence d'autre serveur Web. Ainsi les données ne sont pas tronquées, si elles sont gérées par un seul serveur Web.

a) **Installation LINUX**

Sur LINUX, téléchargez les paquets "sqlite3" et "sqlitebrowser". SQLITE Browser a besoin d'une interface graphique.

Tapez cette commande en monde administrateur :

```
ln -d /usr/lib/libsqlite3.so.0 /usr/lib/libsqlite3.so
```

b) **Installation WINDOWS**

Allez à <http://www.sqlite.org/> et téléchargez SQLITE pour WINDOWS.

Mettez la DLL SQLITE dans le répertoire de la FastCGI.

c) **SQLITE**

Dans les "Accessoires", dans la "Ligne de commande", ou dans le "Terminal", placez-vous sur le répertoire de votre FastCGI.

```
cd repertoire\en\copié-collé\ou\glissé-déplacé
```

Démarrez le shell sqlite3, suivi de votre base de données, en validant par entrée :

```
sqlite3 mabaseweb.db
```

Copiez-collez ce script SQL suivi de entrée :

```
PRAGMA encoding = "UTF-8";  
CREATE TABLE UTILISATEUR (  
    CLEP INTEGER NOT NULL PRIMARY KEY  
    AUTOINCREMENT,  
    NOM VARCHAR(255),  
    PRENOM VARCHAR(255)  
);
```

Quittez le shell SQLITE :

```
.quit
```

Vous pouvez maintenant accéder à votre base avec votre Logiciel Web. Copiez votre base dans le répertoire de la FastCGI.

16) L'EXEMPLE

Revenons sur notre fiche.

Dans la fenêtre principale nous allons créer les requêtes entièrement dans le Code Source, car EXTPASCAL 0.9.8 ne reconnaît pas encore SQLDB. Vous pouvez participer à EXTPASCAL pour remédier à ce problème.

Vérifiez cependant si EXTPASCAL ne construit pas automatiquement les requêtes SQLDB. Regardez dans les Sources créées, avec l'extension ".inc", à partir du projet de traduction des fiches LAZARUS.

a) ZEOS DBO

Si vous ne voyez pas vos queries SQLDB construits dans le fichier ".inc" , renseignez le Constructeur de la fiche :

```
constructor TExample.Create;
Begin
  inherited;
  {$IFDEF UseRuntime}
  {$I *.inc}
  gi_RowIndex := -1 ;
  gd_Utilisateurs.StripeRows:=True;
  Connection := TZConnection.Create(nil);
                                     Connection.Database
:=IncludeTrailingPathDelimiter(GetAppConfigDirUTF8(False))
+'mabaseweb.db';
// Connection.charset:= 'UTF-8' ;
```

```

SQLTool      := TZQuery.Create(nil);
SQLTool      .Connection := Connection;
SQLUtilisateur := TZQuery.Create(nil);
SQLUtilisateur.Connection := Connection;
SQLUtilisateur.SQL.Text := 'SELECT * FROM UTILISATEUR' ;
gstl_CleUtilisateur := TStringList.Create();
gstl_CleUtilisateur.Add ( 'CLEP' );
gd_Utilisateurs.OnRowClick := GridRowClick;
{SENDIF}
end;

```

ZEOS DBO nécessite moins de sources. Il est donc mieux automatisé.

Renseignez le Destructeur de la fiche :

```

destructor TExample.Destroy;
begin
  inherited;
  {SIFDEF UseRuntime}
  Connection.Free;
  SQLTool.Free;
  SQLUtilisateur.Free;
  gas_Enregistrements.free;
  gstl_CleUtilisateur.Free;
  {SENDIF}
end;

```

b) SQL DB

Si vous ne voyez pas vos queries SQLDB construits dans le fichier ".inc" , renseignez le Constructeur de la fiche :

```

constructor TExample.Create;

```

Begin

inherited;

{\$IFDEF UseRuntime}

{\$I u_form.inc}

gi_RowIndex := -1 ;

gd_Utilisateurs.StripeRows:=True;

SQLite3Connection := TSQLite3Connection.Create(nil);

SQLite3Connection.DatabaseName

:=IncludeTrailingPathDelimiter(GetAppConfigDirUTF8(False))

+**'mabaseweb.db'**;

SQLite3Connection.charset:= 'UTF-8' ;

SQLTransaction := TSQLTransaction.Create(nil);

SQLTransaction.Database := SQLite3Connection;

SQLTransaction.Action := caCommitRetaining;

SQLTool := TSQLQuery.Create(nil);

SQLTool .Transaction := SQLTransaction;

SQLTool .Database := SQLite3Connection;

SQLUtilisateur := TSQLQuery.Create(nil);

SQLUtilisateur.Transaction := SQLTransaction;

SQLUtilisateur.Database := SQLite3Connection;

SQLUtilisateur.SQL.Text := **'SELECT * FROM UTILISATEUR' ;**

'DELETE FROM UTILISATEUR WHERE CLEP='

SQLUtilisateur.InsertSQL.Text := 'INSERT INTO UTILISATEUR
(CLEP,NOM,PRENOM) VALUES (:CLEP,:NOM,:PRENOM)';

SQLUtilisateur.UpdateSQL.Text := **'UPDATE UTILISATEUR**
SET NOM=:Nom,PRENOM=:Prenom WHERE CLEP=:Clep';

SQLUtilisateur.DeleteSQL.Text := **'DELETE FROM**
UTILISATEUR WHERE CLEP=:Clep';

gstl_CleUtilisateur := TStringList.Create();

gstl_CleUtilisateur.Add (**'CLEP'**);

gd_Utilisateurs.OnRowClick := GridRowClick;

{\$ENDIF}

end;

SQL DB nécessite plus de sources. Il est donc moins automatisé.

Renseignez le Destructeur de la fiche :

```
destructor TExample.Destroy;  
begin  
  inherited;  
  {$IFDEF UseRuntime}  
    SQLite3Connection.Free;  
    SQLTransaction.Free;  
    SQLTool.Free;  
    SQLUtilisateur.Free;  
    gas_Enregistrements.free;  
    gstl_CleUtilisateur.Free;  
  {$ENDIF}  
end;
```

c) Le logiciel

Nous allons ajouter les éléments inclus dans la base SQLDB :

```
procedure TExample.p_CreeChaineEnregistrements ( var chaine :  
String );  
begin  
  p_CreeChaineEnregistrements ( chaine,  
  SQLUtilisateur.FieldName('Nom').AsString,  
  SQLUtilisateur.FieldName('Prenom').AsString,  
  SQLUtilisateur.FieldName('Clep').AsString );  
  SQLUtilisateur.Next;  
End;  
  
function TExample.CreeChaineAjax ( const s : String ):String;  
Begin
```

```

// Le Code Javascript affiche les chaines entre des guillemets
Result := StringReplace(s,'"','\"', [rfReplaceAll]);
end;

procedure TExample.p_CreeChaineEnregistrements(var chaine:
String; const s_Nom,
s_Prenom, s_cle : String);
begin
if chaine <> " Then
chaine := chaine + ',';
chaine := chaine + '[' + CreeChaineAjax (s_Nom) + ', ' +
CreeChaineAjax (s_Prenom) + ', ' + s_Cle + ']' ;
End;

procedure TExample.AjouterElements;
var i : Longint ;
s_Records : String;
begin
try
SQLUtilisateur.First;

// Création du tableau
while not SQLUtilisateur.EOF do
Begin
p_CreeChaineEnregistrements ( s_Records );
End;
Except
On e: Exception do
ExtMessageBox.Alert('Erreur', 'Problème de gestion des
données : '+#13#10+e.Message, Ajax(CloseAjax, ['OK', '%0']));
End;
// Création d'un tableau AJAX
gas_Enregistrements:= TExtDataArrayStore.Create;
With gas_Enregistrements Do

```

Begin

//Création de la structure de données

```
with TExtDataField.AddTo(Fields) do Begin TypeJS := 'String';
Name :='Nom'; End;
with TExtDataField.AddTo(Fields) do Begin TypeJS := 'String';
Name :='Prenom'; End;
with TExtDataField.AddTo(Fields) do Begin TypeJS := 'Integer';
Name :='Clep'; End;
// ExtMessageBox.Alert('Erreur', 'Impossible d'enregistrer :
'+#13#10 + 'Il n'y a pas de nom d'utilisateur. ');
end;
// Affectation des gas_Enregistrements
gas_Enregistrements.Data:=gas_Enregistrements.JSArray(s_Records
,True);
with gd_Utilisateurs do
  Begin
    SetDisabled(True);
  Reconfigure(gas_Enregistrements,
  TExtGridColumnModel(ColModel));
    SetDisabled(False);
  end;
  ExtMessageBox.Alert('Erreur', s_Records);
end;
```

Puis on ajoute les enregistrements en reneignant l'événement OnClick du bouton :

```
procedure TExample.AjouterClick;
Var chaine : String ;
    Compteur : Int64;
    s_Records : String;
begin
  if ( tf_nom.JsString ( tf_nom.GetValue ) = "" ) Then
    Begin
      ExtMessageBox.Alert('Erreur', JsString ( tf_nom.GetValue )
```



```

+'Impossible d'enregistrer : '+#13#10 + 'Il n'y a pas de nom
d'utilisateur. ');
    Exit;
End;
with SQLUtilisateur do
    try
        Insert;
        FieldByName('PRENOM').AsString := JsString
(tf_prenom.GetValue );
        FieldByName('NOM').AsString := JsString ( tf_nom
.GetValue );
        Post;
        ApplyUpdates;
        SQLTransaction.CommitRetaining;
        ExtMessageBox.Alert('Erreur', 'Impossible d'enregistrer :
'+#13#10+SQLTool.SQL.Text+JsString ( tf_nom.GetValue ));
    Except
        On e: Exception do
            with SQLUtilisateur do
                Begin
                    ExtMessageBox.Alert('Erreur', 'Impossible d'enregistrer :
'+#13#10+e.Message, Ajax(CloseAjax, ['OK', '%0']));
                    Exit;
                end;
            End;
            s_records := "";
            p_CreeChaineEnregistrements ( s_Records, JsString ( tf_nom
.GetValue ), JsString ( tf_prenom.GetValue ), IntToStr ( Compteur ) );
            try
                gd_Utilisateurs.SetDisabled(True);
                TExtDataArrayStore(gd_Utilisateurs.GetStore ).Insert(0,
JSArray('new Ext.data.Record ( "'+JsString ( tf_nom .GetValue )
+'", "'+JsString ( tf_prenom.GetValue )+'")');
            finally

```

```
    gd_Utilisateurs.SetDisabled(False);  
end;  
End;
```

Voici l'événement "OnClick" du bouton "Effacer" :

```
procedure TExample.EffacerClick;  
var datastore: TTextDataStore;  
    datarecord : TTextDataRecord;  
    ATemp : String ;  
begin  
    datastore := TTextDataStore ( gd_Utilisateurs.GetStore );  
    ATemp := JSString(datastore.GetCount );  
    if SQLUtilisateur.Active  
    and not SQLUtilisateur.IsEmpty  
    and ( gi_RowIndex >= 0 )  
    and ( gi_RowIndex < StrToInt(ATemp))  
    Then  
        Begin  
            datastore := TTextDataStore ( gd_Utilisateurs.GetStore );  
            ATemp :=  
JsString(TTextDataRecord(datastore.GetAt( gi_RowIndex )).Get('Clep  
'));  
            SQLTool.SQL.Text := 'DELETE FROM UTILISATEUR  
WHERE CLEP='+ATemp;  
            SQLTool.ExecSQL;  
            SQLTransaction.CommitRetaining;  
            with gd_Utilisateurs do  
                Begin  
                    setDisabled ( True );  
                    datastore := TTextDataStore ( GetStore );  
                    datarecord :=  
TTextDataRecord(datastore.GetAt( gi_RowIndex ));  
                    datastore := TTextDataStore ( GetStore );  
                    datastore.remove ( datarecord );
```

```
Reconfigure ( datastore, TExtGridColumnModel(ColModel) );
setDisabled ( False );
end;
End;
end;
```

Compilez.

Mettez en place la FastCGI sur APACHE dans le répertoire cgi, en enlevant l'extension, redémarrez le serveur.

Tapez <http://localhost/cgi/fastcgi>. Vous voyez l'exemple s'afficher.

Si les événements ne sont pas affectés dans la CGI, Exécutez l'Application de conversion d'unité EXTPASCAL, comme nous l'avons vu précédemment. Recompilez alors la FastCGI.

S) COLLABORER

CREATIVE COMMON BY NC-ND

1) INTRODUCTION

Lorsque son projet, ou celui d'un autre, demande l'intervention d'au moins un autre développeur, il convient d'installer un outil de collaboration. On utilise un outil collaboratif public, si nos sources sont libres, ou privé, si nos sources sont commerciales.

2) PARTICIPER

Une fois que l'on a créé ses Composants et qu'ils évoluent correctement, on peut enfin participer aux projets LAZARUS.

Tout projet est Open Source afin que d'autres y participent. Seulement pour participer au projet LAZARUS il faut déjà améliorer des composants. Vous avez vu, dans le chapitre permettant la création de son savoir-faire :

- Qu'il faut avoir en tête la structure des composants ancêtres.
- Que toute nouvelle propriété doit avoir un nom bien choisi.
- Qu'il faut connaître l'anglais.
- Qu'il faut documenter et commenter.
- Qu'à terme on pense en Objet, avec des outils adéquates.
- Qu'à terme on sait s'adapter à la machine, avec plein d'astuces.

Nous vous avons donné des clés pour optimiser votre Code, à vous de trouver de nouvelles astuces.

Plutôt que d'ajouter sans avoir d'idée sur la façon de le faire, il est préférable de faire avancer le projet LAZARUS, et de contacter des développeurs. Tout développeur utilisant LAZARUS peut vous aider. Les passionnés documentent et essaient d'être disponibles. Seule l'objectivité permet de s'améliorer pour participer à un projet.

On commence avec un composant à déboguer, puis on ajoute une fonctionnalité, puis on optimise son composant sans cesse, puis on partage son composant pour être critiqué.

3) *PRINCIPE*

Les sources sont accessibles à tous les développeurs participant au projet, grâce à un serveur de sources accessible.

Les développeurs, à chaque fois qu'ils veulent modifier une unité, mettent à jour leurs unités locales, sur leur ordinateur. Puis ils modifient.

Une fois l'unité modifiée, et après avoir testé, il est préférable de transférer, vers le serveur central, toutes les sources modifiées. En effet, les sources sont souvent imbriquées.

Les autres développeurs accèdent aux nouvelles sources.

4) *POUR UNE PETITE ÉQUIPE*

Lorsque l'équipe qui travaille sur les sources est petite, il est intéressant

d'installer un outil de collaboration avec jeton exclusif.

Le fonctionnement en jeton exclusif est simple : Si quelqu'un veut modifier une unité, il a l'exclusivité sur la modification. Ainsi il n'y aura pas à fusionner la même unité modifiée à deux endroits différents.

Par exemple JEDI CVS est un outil collaboratif libre permettant la distribution de sources par jetons exclusifs. JEDI CVS ne fonctionne malheureusement que sous WINDOWS.

5) POUR UNE GRANDE ÉQUIPE

Lorsque beaucoup de développeurs participent à un projet, il convient de choisir un outil collaboratif permettant la comparaison et la fusion de sources.

Le principe est simple : Quiconque peut modifier quelque source que ce soit.

Si chacun ne modifie pas en même temps les mêmes sources, l'outil s'utilise simplement.

Il peut cependant arriver que certains modifient en même temps une source. Alors le premier qui publie ses unités n'a rien à faire. Celui qui est en retard doit fusionner ses sources avec celui qui a publié avant, puis tester évidemment.

Il est préférable d'utiliser ce genre d'outils avec des alertes mails, afin que les développeurs soient au courant de ce qui est modifié.

En 2011, un très bon outil collaboratif s'appelle MERCURIAL, avec

TORTOISE HG comme client. Il permet de réaliser des patches facilement, tout en distribuant les sources à beaucoup de développeurs.

En 2011, les outils SVN et CVS n'ont pas autant de répondant, ni autant de possibilités que MERCURIAL.

6) CHOISIR SON OUTIL

Un patch est un ajout de Code ou de Sources, s'insérant automatiquement dans le Code ou les Sources existantes, en fonction d'une ou de plusieurs versions de départ.

Un bon outil de collaboration permet de :

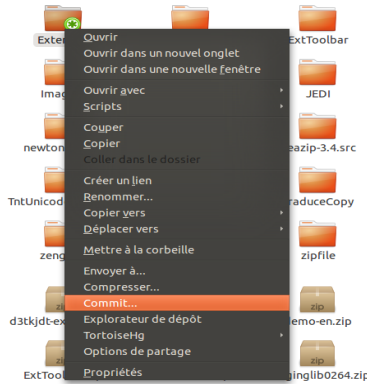
- Créer des archives permettant de recréer les anciennes versions
- Communiquer facilement entre les développeurs
- Créer des patches facilement
- Créer plusieurs versions de son exécutable
- Travailler sur les différents systèmes d'exploitation
- Travailler simplement, sans avoir à se soucier des difficultés d'un partage

Les outils TORTOISE sont intégrés à votre bureau.

TORTOISE SVN et TORTOISE CVS impliquent une compatibilité de l'outil de programmation.

TORTOISE HG pour MERCURIAL s'utilise facilement.

Vous avez ci-après un exemple d'intégration d'un outil TORTOISE dans le bureau :



Les outils TORTOISE sont intégrés au bureau graphique

RAPID SVN pour SVN, GCVS sous LINUX avec WIN CVS sous WINDOWS, permettent de répondre au multiplate-forme de LAZARUS. Ces outils graphiques sont simples d'utilisation.

7) LES FORGES

Une forge est un serveur de sources regroupant les sources de différents savoir-faire, de différents logiciels.

Une forge partagée regroupe des sources mondiales et n'accepte que des sources partageables. Les licences des logiciels de ces forges sont des licences Open Source.

a) Serveurs gratuits

Il existe différentes forges partagées ou libres.

www.sourceforge.net regroupe des sources partagées. En 2011, il n'utilise malheureusement que SVN et CVS.

code.google.com regroupe des sources partagées. Il peut utiliser MERCURIAL.

b) Serveur français

En France www.adullact.net est une forge partagée, réservée aux collectivités et administrations.

8) AMÉLIORER LAZARUS

LAZARUS est bien conçu. Les développeurs de LAZARUS vérifient la rapidité de leur outil. Les modifications qui ne s'assemblent pas de la même manière que DELPHI sont des améliorations structurelles. Elles permettent par exemple d'améliorer la lecture d'images, de réutiliser des API graphiques. LAZARUS c'est une réutilisation du meilleur des bibliothèques existantes.

Seulement si on s'arrête là LAZARUS ne peut pas s'améliorer. Les Composants RAD sont non seulement intuitifs, mais permettent d'améliorer le Polymorphisme.

Utilisons l'objectivité. Vous voyez que tous les Composants LAZARUS ne suffisent pas pour créer un Logiciel. Il est possible de participer à LAZARUS en traduisant un Composant Libre DELPHI vers LAZARUS. Le Composant peut rester compatible DELPHI grâce aux directives de compilation. Ainsi dès que la source diffère on ajoute une

directive FREE PASCAL.

Si votre composant est primordial pour une utilisation standard de LAZARUS, vous pouvez proposer votre Composant en Patch pour LAZARUS. Les développeurs de LAZARUS vérifient ensuite la licence, et le respect de la programmation par Objets. Ensuite si le Composant est nécessaire au projet il est validé.

Sinon vous pouvez diffuser votre Composant sur un site Web regroupant des Composants. Faites-le connaître par le réseau LAZARUS. Vous pouvez diffuser votre composant grâce à lazarus.freepascal.org. Il est nécessaire de créer un message en anglais, sur ce site web, dans le forum contenant les messages de "Tierce partie".

Si votre Composant est en cours d'importation il est possible de devenir développeur LAZARUS. Vous devenez alors un auteur de LAZARUS. Il est aussi possible de faire évoluer FREE PASCAL pour que des instructions DELPHI soient lues par LAZARUS. Si vous avez l'accord des développeurs de FREE PASCAL, vous créez alors une sémantique Libre FREE PASCAL, basée sur DELPHI.

T) CROSS-COMPILATION LAZARUS

CREATIVE COMMON BY NC-ND

1) *INTRODUCTION*

Vos exécutables créés sur LAZARUS ne peuvent pas fonctionner sur toutes les machines.

Le Code machine que vous créez avec LAZARUS permet certes d'être plus proche de la machine, donc plus rapide, mais il demande de posséder un certain environnement logiciel et matériel.

Tout d'abord le processeur est l'architecture matérielle. L'environnement est l'architecture logicielle. Ces deux architectures nécessitent des exécutables différents.

2) *LES ARCHITECTURES*

Une plate-forme est une architecture logicielle possédant une architecture matérielle.

La cross-compilation consiste à compiler un exécutable d'une plate-forme, vers une autre plate-forme.

Il est en théorie possible, avec LAZARUS, de compiler un exécutable pour une autre plate-forme. Cela est très utilisé lorsqu'on veut diffuser ses exécutables. Par exemple vous pouvez diffuser votre exécutable pour les téléphones mobiles, WINDOWS, LINUX, MAC OSX, etc.

a) **Architectures matérielles**

On répertorie beaucoup d'architectures matérielles. Vous pouvez voir beaucoup d'architectures matérielles en téléchargeant un LINUX sur le site web de LINUX DEBIAN.

Architectures actuelles

Les vieux ordinateurs 16 bits, donc possédant 2^{16} entiers de nombres binaires allant jusqu'à 2^{16} , ne sont plus compatibles avec les environnements actuels.

En effet un entier de 16 bits possède un maximum de 65 335 entiers de 16 bits à gérer en mémoire. Ce n'est pas suffisant pour une mémoire actuelle.

En 2011 vous compilez donc des exécutables pour les architectures 32 bits, les architectures les plus diffusées. Vous compilez aussi pour les architectures puissantes en 64 bits, grâce à LAZARUS et aux unités agréées 64 bits.

Les architectures récentes possèdent évidemment d'autres contraintes que la taille de leurs adresses. Nous vous expliquons cela ci-après.

Architecture CISC

CISC signifie en anglais Complex Instruction Set Computer. Un processeur CISC est un microprocesseur à jeu d'instruction étendu.

L'architecture CISC est une architecture avec beaucoup d'instructions, permettant les explosions combinatoires, pour les jeux 3D par exemple.

Les explosions combinatoires peuvent faire planter un processeur.

L'architecture i386 est l'architecture PC en 32 bits. Les processeurs Intel 386 étaient les premiers ordinateurs PC 32 bits.

L'architecture i386 accepte WINDOWS, LINUX, voire MAC OSX. Les environnements i386 sont compatibles avec un processeur 64 bits. L'architecture i386 permet d'utiliser moins de 4 Go de mémoire vive.

Si on possède au moins 4 Go de mémoire il semble intéressant de passer à une architecture 64 bits très puissante.

L'architecture AMD 64 bits est la première architecture 64 bits. Cette architecture est disponible sur LINUX, WINDOWS, MAC OSX. Les processeurs AMD 64 bits et Intel 64 bits sont compatibles avec un environnement AMD 64 bits.

Architecture RISC

RISC signifie en anglais Reduced Instruction Set Computer. Un processeur RISC est un microprocesseur à jeu d'instruction réduit. L'architecture RISC comporte peu d'instructions au sein du processeur. Cela permet de l'économie.

Un processeur RISC est plus fiable qu'un processeur CISC. L'aviation utilise de vieux processeurs RISC.

Les processeurs CISC évoluaient plus facilement que les processeurs RISC. Les deux architectures sont à égalité maintenant. Il est cependant important de connaître les générations de processeurs RISC.

L'architecture ARM est une alternative économique aux processeurs Intel ou AMD.

Le processeur ARM nécessite peu d'instructions. On peut donc mettre les processeurs de la carte mère dessus. Cependant WINDOWS, trop gourmand, est très lent dessus.

Le processeur ARM est disponible dans les SMARTPHONES, les GOOGLE PHONES, les SMARTBOOKS. Le mot "SMART" signifie qu'on utilise des processeurs ARM. Ce sont des téléphones et des ordinateurs très fins.

Les téléphones ARM utilisent LINUX ou WIN CE, un WINDOWS amoindri pour supporter l'ARM. LE GOOGLE PHONE possède l'environnement ANDROID, basé sur LINUX. SYMBIAN OS, l'environnement le plus répandu en 2010, est plus proche de WINDOWS.

Il existe encore d'autres architectures moins utilisées, comme SPARC pour JAVA, MIPS encore plus économique que ARM mais peu puissant, des processeurs HP ou IBM spécifiques.

b) Architectures logicielles

Les trois architectures logicielles les plus connues sont :

- WINDOWS
- LINUX, basé sur UNIX
- MAC OS

Le monopole WINDOWS

WINDOWS a été la première architecture logicielle graphique

largement diffusée. MICROSOFT, le créateur de WINDOWS, a donc utilisé son monopole pour garder ses clients, car WINDOWS est payant. Vous payez WINDOWS entre 30 € et 2000 €. Le prix de 30 € de WINDOWS, sur une machine d'un constructeur, est dû à la concurrence LINUX, demandant un maximum de 30 € d'installation sur une machine d'un constructeur. Si vous achetez une licence d'un WINDOWS récent seule vous payez 200 €, sans pouvoir utiliser l'environnement à plein escient.

WINDOWS est un environnement qui évolue peu. MICROSOFT le modifie que lorsqu'on lui signale des failles de sécurité, ou lorsqu'il y a une nouvelle version de WINDOWS.

Pour trouver les failles de sécurité WINDOWS, des développeurs étudient le Code binaire de l'environnement, sans forcément divulguer leurs trouvailles.

WINDOWS gère toujours très mal les virus. C'est le seul environnement où l'anti-virus est obligatoire. C'est aussi un environnement avec publicités.

WINDOWS, comme il évolue peu, est de plus en plus gourmand. WINDOWS 7 demande 1 Go de mémoire vive pour fonctionner correctement. Pourtant WINDOWS 95 demandait 64 Mo de mémoire vive pour fonctionner, soit 20 fois moins.

WIN CE et WINDOWS Mobile, des WINDOWS peu gourmands, ont donc beaucoup de mal à s'implanter en Embarqué. Ils sont lents, avec peu de possibilités.

WINDOWS est disponible sur processeurs INTEL et ARM.

Choisir LINUX

LINUX peut être installé gratuitement, en téléchargeant un CD d'installation sur INTERNET. LINUX permet d'avoir des serveurs puissants consommant peu de ressources, ceci à moindre coût. Ces serveurs n'ont pas besoin d'interface graphique, car cela consomme de la mémoire.

LINUX est un descendant d'UNIX. C'est un UNIX sous licence libre. On peut donc le modifier comme on le souhaite. Des projets LAZARUS existent pour gérer LINUX.

Comme LINUX est libre, il existe toutes sortes de distribution LINUX, répondant à une demande de l'utilisateur. Les distributions LINUX DEBIAN et LINUX UBUNTU se mettent à jour facilement grâce au réseau.

Après WINDOWS, LAZARUS est beaucoup développé sur LINUX. En effet, LAZARUS permet de créer des Logiciels pour les bureaux LINUX GNOME, et LINUX KDE.

GNOME et KDE sont les deux principaux environnements graphiques de LINUX. Par exemple GIMP utilise l'environnement GNOME, par ailleurs disponible sur WINDOWS. LINUX GNOME est référencé dans LAZARUS par GTK, puis GTK2 plus joli. KDE est référencé par QT.

LINUX est utilisé en Embarqué sur les SMARTPHONES comme GOOGLE ANDROID, sur la plupart des BOXS. En général, quand un fabricant a développé la partie logicielle de son ordinateur à moindre coût, il a choisi LINUX.

Les processeurs POWER PC sont les processeurs des IMACS. Vous pouvez installer un LINUX POWER PC sur un IMAC. Cependant, laissez le MAC OSX dessus car les IMACS sont tatoués. Ils interdisent

le remplacement complet de l'environnement.

LINUX manque de développeurs RAD. En effet beaucoup de développeurs LINUX sont encore adeptes de l'éditeur de textes sans outil supplémentaire.

LINUX est disponible sur la plupart des architectures matérielles. En effet les développeurs, qui participent à LINUX, s'attachent à l'interopérabilité de leur environnement préféré.

L'accessibilité LINUX coûte en 2011 10 € grâce à LINUX DEBIAN sur GNOME, contre 2000 € pour WINDOWS. L'accessibilité LINUX DEBIAN sera bientôt gratuite et sera aussi sur LINUX KDE.

Le livre "L'Astucieux LINUX" vous permet d'installer LINUX à partir de WINDOWS.

MAC OS

MAC OS est un descendant d'UNIX. C'est la Rolls-Royce des environnements. En effet il est extrêmement coûteux, alors que l'architecture logicielle d'origine coûte peu cher.

Sur MAC OS, vous trouverez rarement des logiciels gratuits. Comme WINDOWS, MAC OS est un système d'exploitation en location : Si vous changez le matériel vous devez racheter l'environnement.

MAC OS était à l'origine sur architecture RISC, avec les processeurs POWER PC, puis ARM. MAC OSX est maintenant disponible sur processeur INTEL.

3) CROSS-COMPILER

La cross-compilation consiste à compiler d'une plate-forme vers une autre. Cela permet de diffuser son exécutable vers des petites machines.

Il est donc très intéressant de créer d'autres exécutables, sur d'autres processeurs, avec son PC. Il existe une règle : Plus on est proche de l'architecture utilisée, moins le travail de cross-compilation est important.

Il est possible de télécharger une Cross-compilation pour processeur ARM avec WIN CE, sur WINDOWS.

Si vous voulez créer un exécutable pour LINUX ARM et GOOGLE PHONE, il est intéressant d'utiliser LINUX. ANDROID, le LINUX propriétaire de GOOGLE, gagne beaucoup de parts de marché en 2011. On compile vers SYMBIAN OS avec WINDOWS.

4) SMARTPHONES

En 2011, il existe différents tutoriels pour cross-compiler vers les SMARTPHONES ou vers les IPHONES :

http://wiki.lazarus.freepascal.org/Smartphone_Development

Le compilateur FREE PASCAL est disponible sur LINUX ARM, donc sur ANDROID. Cependant l'OPEN GL est plus difficile à mettre en œuvre, au début d'année 2011.

Peut-être que vous avez dorénavant vos paquets de cross-compilation ARM sur LINUX i386. Si vous compilez un exécutable pour les processeurs ARM génération 5, vous êtes en théorie compatible avec beaucoup de SMARTPHONES.

Le processeur ARM consomme peu de courant car il chauffe peu. Il est utilisé modifié, sans le dire, par APPLE dans les IPHONES et IPODS. Cependant, en 2010, il faut posséder MAC OSX pour cross-compiler vers les IPHONES.

Ensuite, dans LAZARUS, puis dans le menu "Outils", puis "Configurer Build LAZARUS", puis "Options de construction avancées", choisissez la bonne architecture voulue. Pour faire cela placez l'"OS de destination" comme "linux", puis le processeur ou "CPU de destination" comme "arm".

Les plates-formes de téléphonie, de livres électroniques, ou de tablettes ont une configuration particulière. Elles disposent en plus de gadgets supplémentaires.

Il est intéressant de tester directement après avoir compilé. C'est pourquoi les gadgets et la configuration peuvent être testés avec les SDK de chaque environnement, disponibles gratuitement ou pas. Les environnements de téléphones portables, les plus largement diffusés, disposent en général d'un SDK gratuit.

APPLE propose ainsi de tester les logiciels pour IPODS ou IPHONES avec un des derniers SDK pour MAC OSX. WIN CE peut être testé sous WINDOWS. GOOGLE propose des SDK sur l'ensemble des plates-formes afin de favoriser la richesse de son environnement.

On cross-compile vers SYMBIAN OS sur WINDOWS, avec le SDK SYMBIAN. Attention ! Le projet doit être terminé pour être sûr de la compatibilité.

Tapez sur votre moteur de recherche "SDK SYMBIAN", "SDK ANDROID" "SDK IPHONE", ou "WIN-CE SDK".

5) **JAVA vs LAZARUS**

Indéniablement le réseau LAZARUS manque encore de bibliothèques concurrentes. Cependant, il existe toutes sortes de Composants Libres ou gratuits sur DELPHI, facilement traduisibles.

LAZARUS ce sont les atouts de JAVA, sans des contraintes de JAVA. On s'aperçoit que les Destructeurs permettent une meilleure optimisation de la mémoire. Au lieu d'installer un serveur avec TOMCAT, qui devra être optimisé, il est plus judicieux de détecter les fuites mémoires au sein des Destructeurs afin d'optimiser les variables créées. En effet libérer la mémoire dès qu'elle est inutilisée permet déjà d'optimiser.

Il n'y a pas de JVM sous LAZARUS, ce moteur traduisant les exécutables JAVA en interfaces compréhensibles par l'utilisateur. Votre exécutable LAZARUS est donc plus rapide qu'un logiciel JAVA sous une JVM.

Votre exécutable LAZARUS est aussi suffisamment complet pour être indépendant de bibliothèques. En général, Les bibliothèques utilisées par LAZARUS ne sont pas nécessaires à votre exécutable. Votre exécutable devient rapide et autonome.

Vous ne connaissez peut-être pas les BEANS sous JAVA. La partie Composant des BEANS JAVA ne permet pas de gagner autant de temps que les Composants RAD. On appelle cela les LCL sur LAZARUS, pour Librairie de Composants LAZARUS. Dans LAZARUS, même les Composants non visuels sont dans cette bibliothèque.

Un EDI JAVA nécessite plus de temps de maintenance, et de développement, qu'un Framework LAZARUS, à cause d'une gestion rendue complexe par la programmation JAVA, nécessitant d'utiliser toujours des classes, sans automatisation dans la création.

JAVA nécessite la réinvention d'un nouvel EDI afin de gagner du temps. Il faut à chaque fois changer d'EDI, les plus récents devant se réinventer pour gagner du temps. LAZARUS est déjà optimisé pour gagner du temps. Si on crée un deuxième EDI, on peut utiliser LAZARUS pour arriver à ses fins.

Dans LAZARUS ou DELPHI tout Composant est facilement remplaçable par un autre. Il suffit pour cela de changer le type du Composant dans le fichier PASCAL et dans la Source du formulaire.

L'inspecteur d'Objets permet de mettre en place rapidement tout Composant. Vous gagnez un temps précieux à créer des Composants, en Développement Rapide d'Applications, contrairement à JAVA.

La programmation par Objets permet de créer des interfaces proches de l'humain. Elle est au sein de JAVA et de LAZARUS. Contrairement à JAVA, LAZARUS permet d'utiliser une programmation proche de l'humain et une programmation proche de l'ordinateur. En effet une programmation proche de l'humain ralentit votre exécutable.

U) CRÉATION DU LIVRE

CREATIVE COMMON BY NC-ND

1) HISTORIQUE

En 2008 il n'existait pas de livre français sur LAZARUS. Matthieu GIROUX a donc créé des essais, puis des articles sur LAZARUS. Puis des images de LAZARUS ont été ajoutées. Le livre s'est ordonné en chapitres.

Il a fait connaître son livre sur www.developpez.com et www.framasoft.net, en diffusant des articles devenus Libres.

Le livre évolue toujours en 2011, avec les améliorations de LAZARUS et les nouveaux paquets LAZARUS.

Les chapitres sur la programmation procédurale en FREE PASCAL sont un article retravaillé de Jean-Michel BERNABOTTO.

Retrouver un chapitre

A) A lire.....	4
1) Objectifs du livre.....	4
2) Licence.....	5
B) Biographie.....	6
1) Du même auteur.....	6
C) LAZARUS FREE PASCAL.....	7
1) Pourquoi choisir LAZARUS ?.....	7
2) Architectures FREE PASCAL.....	8
3) Applications Libres LAZARUS.....	9
4) Du PASCAL orienté Objet.....	10
5) La communauté.....	11
6) LAZARUS est partagé.....	11
7) Les versions de LAZARUS.....	11
8) Télécharger LAZARUS.....	13
9) Installer LAZARUS sous WINDOWS.....	15
10) Installer LAZARUS sous LINUX.....	15
11) Configurer LAZARUS.....	15
D) Programmer facilement.....	17
1) Créer un logiciel.....	17
2) Paquet pour débutants.....	25
3) Indentation PASCAL.....	26
4) Structure du Code Source.....	28
5) Les fichiers ressources.....	31
6) Touches de raccourcis de complétion.....	32
7) Touches de raccourcis de visibilité.....	35
8) Touches de raccourcis de débogage.....	35
9) Touches de raccourcis de l'éditeur.....	37
10) Touches de raccourcis de l'environnement.....	38
E) Le langage PASCAL.....	39
1) Introduction.....	39
2) Introduction.....	39

3) Instruction PASCAL.....	39
4) Les fonctions et procédures.....	40
5) Tester son programme.....	41
6) Les types simples définis.....	41
7) Les variables globales.....	44
8) Les variables locales.....	46
9) La notion de procédure.....	49
10) Le mot clé "var" dans les paramètres.....	53
F) Programmation procédurale avancée.....	58
1) Introduction.....	58
2) Centralisation.....	58
3) Optimisation des arguments.....	60
4) Créer des Sources correctes.....	61
5) Optimiser avec les pointeurs.....	64
6) Règles générales.....	66
7) Tester son programme.....	68
8) Destruction de variables.....	70
9) Les Objets et leur destruction.....	70
10) Enlever les fuites mémoires.....	73
11) Tester des routines.....	74
G) Calculs et types complexes.....	76
1) Les comparaisons.....	76
2) L'instruction "if".....	77
3) Affectation.....	78
4) Les opérateurs numériques.....	78
5) Le type Enregistrement.....	80
6) Les variants.....	86
H) Les boucles.....	88
1) La boucle "for".....	88
2) La boucle "while".....	90
3) La boucle "repeat".....	91
4) "Continue" et "Break".....	93
5) Les procédures ou fonctions récursives.....	93
I) Créer ses propres types.....	96

1) Introduction.....	96
2) Créer ses types énumérés	97
3) Règles à respecter.....	98
4) Type intervalle.....	102
5) Le type tableau.....	104
6) Utilisation du type array.....	106
7) L'instruction With.....	110
J) Ma première application.....	114
1) A faire avant.....	114
2) L'Exemple.....	114
3) Création de l'interface.....	114
4) Tester ses Composants.....	115
5) L'Exemple.....	116
6) Caractères accentués.....	122
7) Chercher des Projets.....	122
8) Installer des Composants.....	123
9) LAZARUS ne démarre plus.....	123
10) Vérifier les licences.....	124
11) Compilateur FREE PASCAL.....	125
12) Gestion des erreurs.....	126
13) Les exceptions.....	126
K) L'Objet.....	129
1) Introduction.....	129
2) Un Objet.....	130
3) Une classe.....	130
4) Une Instance d'Objet.....	131
5) Les comportements de l'Objet.....	135
6) L'Héritage.....	136
7) La surcharge.....	137
8) L'Encapsulation.....	139
9) Le Polymorphisme.....	149
10) Les propriétés.....	152
11) L'UML pour programmer en Objets.....	153
L) Créer son savoir-faire.....	154

1) Introduction.....	154
2) Créer des unités de fonctions.....	154
3) Les Composants.....	155
4) Développement Très Rapide d'Applications.....	156
5) Intérêts du DTRA.....	157
6) Créer un Framework DTRA.....	157
7) Créer un Composant.....	159
8) Surcharger un Composant.....	169
9) Créer une librairie homogène.....	169
10) Le Libre et l'entreprise.....	170
11) Tester son savoir-faire.....	171
12) L'exemple.....	171
13) Exercice.....	177
14) Conclusion.....	178
M) De PASCAL vers FREE PASCAL.....	179
1) Introduction.....	179
2) De TURBO PASCAL vers FREE PASCAL.....	179
3) Gestionnaire de données.....	179
4) De DELPHI vers LAZARUS.....	180
5) Les directives de compilation.....	181
6) Traduction de Composants.....	184
N) L'Objet et les jeux.....	186
1) Introduction.....	186
2) Pourquoi créer des jeux 2D ?.....	186
3) Les jeux 2D avec ZEN GL.....	187
4) ZEN GL Sous LINUX.....	188
5) Un projet de jeu.....	188
6) Conclusion.....	203
O) La persistance d'un Logiciel.....	205
1) Introduction.....	205
2) Les fichiers ".ini".....	205
3) Les fichiers ".csv".....	210
4) Les fichiers ".dbf".....	217
P) Logiciel centralisé.....	224

1) Introduction.....	224
2) ORACLE et MY SQL.....	224
3) ORACLE vs le libre.....	225
4) Modéliser son Logiciel.....	226
5) Pas de doublons mais des relations.....	227
6) L'Exemple.....	228
7) Penser à définir.....	228
8) Relation unique.....	228
9) Relation multiple 1-N.....	231
10) Relation multiple N-N.....	232
11) Relation multiple N-N-N.....	234
12) Conclusion.....	234
13) Bases de données.....	235
14) Le pouvoir du Libre.....	235
15) Bases de données embarquées.....	235
Q) Programmation avec FIREBIRD.....	237
1) Introduction.....	237
2) Utilisation de tables dans LAZARUS.....	237
3) Points forts de FIREBIRD.....	237
4) Points faibles d'un SGBD SQL.....	238
5) Récapitulatif.....	239
6) Installation de FIREBIRD.....	239
7) Création d'une table FIREBIRD.....	240
8) Créer une table.....	242
9) Exercice.....	244
10) Les relations.....	246
11) Conclusion.....	252
R) Le Web : EXTPASCAL et SQLITE.....	253
1) Introduction.....	253
2) Le Web.....	254
3) Les FastCGI.....	254
4) EXTPASCAL.....	255
5) Fiabilité d'un serveur LAZARUS.....	255
6) SQLITE.....	256

7) Installer EXTPASCAL.....	256
8) Installer un serveur EXTPASCAL.....	257
9) Configurer le serveur FastCGI.....	260
10) L'exemple.....	262
11) Compiler la FastCGI.....	265
12) Lier aux données.....	265
13) Configurer ZEOS.....	266
14) Configurer SQLDB.....	267
15) SQLITE.....	271
16) L'exemple.....	273
S) Collaborer.....	282
1) Introduction.....	282
2) Participer.....	282
3) Principe.....	283
4) Pour une petite équipe.....	283
5) Pour une grande équipe.....	284
6) Choisir son outil.....	285
7) Les forges.....	286
8) Améliorer LAZARUS.....	287
T) Cross-compilation LAZARUS.....	289
1) Introduction.....	289
2) Les architectures.....	289
3) Cross-compiler.....	296
4) SMARTPHONES.....	296
5) JAVA vs LAZARUS.....	298
U) Création du livre.....	300
1) Historique.....	300
V) Glossaire.....	307

V) GLOSSAIRE

1) RETROUVER UN MOT

Abstraction	L'Abstraction permet le Polymorphisme en orienté Objet en créant une classe abstraite qui sera renseignée avec ses classes filles.
Application	Logiciel permettant de réaliser une ou plusieurs tâches.
Bogue ou Bug	Anciennement, ce terme anglais désigne la punaise, qui mange les circuits électroniques. Cette punaise créait des erreurs. Les programmeurs se sont déchargés alors des erreurs dans le Code, pour les attribuer à cet animal.J'ai mis en place le paiement par carte avec paypal api
Client/Serveur	Interface logicielle connectée à un serveur de données. Un serveur peut fournir l'interface. On appelle cette hiérarchie le Trois Tiers : <ul style="list-style-type: none">- Navigateur Web- Serveur d'interface Web- Serveur de données
Code	Ensemble de 0 et de 1 créés par le compilateur. Le Code sert à agir sur l'ordinateur.
Compilateur	Programme reprenant les différentes Sources de Code pour les compiler en un langage machine, des 0 et des 1 exécutables par le processeur.

Composant	Partie réutilisable de ses Logiciels.
Composant RAD	Composant mis en place facilement.
Constructeur	Méthode paramétrée permettant de créer les Objets d'un Objet et d'initialiser l'Objet. On déclare cette méthode particulière en commençant par "constructor".
Destructeur	Méthode paramétrée permettant de détruire les Objets d'un Objet. En PASCAL Objet on déclare cette méthode particulière en commençant par "destructor".
Embarqué ou Embedded	L'Embarqué c'est placer un Logiciel dans du matériel électronique.
Encapsulation	Déclarations permettant de réutiliser une partie d'un Objet.
Fonction	Procédure ou Méthode paramétrée retournant une variable.
Fork	Copie d'un projet Libre voire Open Source
Framework	Savoir-faire Logiciel réutilisable permettant de réaliser un certain nombres de tâches.
Héritage	Procédés permettant à un Objet d'hériter d'un autre Objet.
IDE ou EDI	Integrated Development Environment ou Environnement de Développement Intégré. LAZARUS est un IDE. Il permet de créer un Logiciel grâce à un ensemble d'outils homogènes.
Librairie	Ensemble de Composants et d'outils permettant des fonctionnalités ou un gain

	de temps.
Libre	<p>Une création ou un Logiciel Libre c'est une création pour laquelle la licence Libre permet de :</p> <ul style="list-style-type: none"> - Étudier la Source. - Utiliser la création librement. - Modifier le Logiciel ou la création. - Dupliquer tout ce qui a été fait. - Diffuser commercialement ou pas ce qui a été créé. <p>Une licence Libre peut être :</p> <ul style="list-style-type: none"> - Virale : La communauté grandit facilement car on est obligé de diffuser la Source modifiée ou le Logiciel l'utilisant. La licence Libre virale la plus connue est la licence GPL qui est utilisée pour partie avec LAZARUS. - Entièrement Libre. La création peut être modifiée commercialement sans avoir aucune contrainte de diffusion forcée de la Source. La licence BSD est une licence entièrement Libre. - Du domaine public car les auteurs ont abandonné les droits commerciaux ou l'œuvre a vu sa période de fin de droits d'auteur terminée. En général il est possible de s'appropriier le projet du domaine public.
Logiciel	Ensemble de traitements permettant la résolution de tâches.

Méthode	Procédure ou fonction dans un Objet. Une méthode peut être encapsulée ou surchargée.
Objet (Analyse ou Programmation Objet)	L'analyse Objet est une analyse des systèmes d'informations qui est proche de l'humain, tout en offrant une architecture compréhensible par la machine. On représente toute entité abstraite ou concrète par un Objet qui dispose de propriétés Objet. Les quatre genres de propriétés d'un Objet sont l'Héritage, le Polymorphisme, l'Encapsulation, l'Abstraction. En respectant ces propriétés, on crée un programme d'Objets disposant de spécificités propres au langage utilisé.
Open Source	Projet à Sources partagées. Il peut être possible de participer au projet en fonction de la licence et de la disponibilité de l'entreprise.
Paquet	Projet LAZARUS permettant de réunir et d'installer des Composants.
Patch	Modification à ajouter à un exécutable. Le patch augmente ou diminue la taille de l'exécutable.
Pointeur	Un pointeur permet de stocker les variables pour les retrouver. Un pointeur est une adresse pointant vers un endroit de la mémoire.
Polymorphisme	Procédé permettant par différentes manières de réutiliser plusieurs Objets en même temps. Le Polymorphisme est résolu par le type Objet "interface" permettant de

	dénommer les méthodes communes de futurs Objets.
Procédure	Bout de Code commençant par la déclaration implémentée et paramétrable "procedure", délimitée par un "Begin" et un "End".
Procédural (Langage)	Les langages procéduraux sont des langages non Objets. Il sont architecturés selon des unités de procédures et fonctions s'appelant entre elles. Le langage PASCAL était au début un langage procédural.
Propriétaire ou Privateur	Un Logiciel Propriétaire ne diffuse aucune source. Il est restreint. WINDOWS est Privateur dans le sens où c'est un Système d'Exploitation en location. En effet, vous êtes restreint par la nécessité de rester compatible avec les dernières Applications payantes.
RAD ou DRA	Rapid Application Development. Développement Rapide d'Applications grâce à un IDE ou une librairie.
Source	Ce qui permet de créer un Logiciel ou un Composant. Il est intéressant d'utiliser des projets Open Source ou Libres afin de ne pas travailler dans le flou.
Type	Définition d'une variable permettant de la manipuler facilement grâce au compilateur.
UML	Unified Modeling Language ou Langage Unifié de Modélisation. Boîte à outils de modèles d'analyse servant à analyser un projet basé sur la programmation Objet.

Variable	Partie de la mémoire, définie par un type, pouvant varier à l'exécution. Une variable stocke des chiffres, lettres, adresses, Objets.
Web	Réseau maillé pouvant contenir un nombre presque illimité d'ordinateurs, grâce au protocole IPV6. Les ordinateurs utilisent un navigateur Web, qui peut accéder aux applications participatives, dites Web 2.0.

ISBN 978295312516

Éditions LIBERLOG
Éditeur n° 978-2-9531251

Droits d'auteur RENNES 2009
Dépôt Légal RENNES 2010

Imprimé en France en Avril 2011 par :
JOUVE
1, rue du Docteur-Sauvé
BP 3
53101 Mayenne cedex

Matthieu GIROUX

LAZARUS

FREE PASCAL

A l'heure où l'on s'intéresse aux économies sur la création de logiciels, LAZARUS permet de répondre à une demande budgétaire en centralisant son savoir-faire ou framework. LAZARUS est un outil libre et gratuit, utilisable facilement et rapidement.

Le Développement Rapide d'Applications n'en est qu'à ses balbutiements. Les outils de développement rapides n'ont pas percé. Dans ce livre nous vous offrons la possibilité de mettre en place votre savoir-faire de Développement Très Rapide d'Applications, la raison d'être de tout outil de DRA.

Après avoir décrit LAZARUS, vous explorez cet outil et le FREE PASCAL. Vous créez alors une application interactive. Ensuite l'objet et un jeu vous permettent de programmer votre premier composant. Enfin vous abordez les données dans les applications évoluées. La cross-compilation LAZARUS et LINUX permettent alors de distribuer vos exécutables vers beaucoup de machines.



Papier 29 € - E-Livre 18 €
Editions LIBERLOG
Editeur n° 978-2-9531251
ISBN 978295312516