Delphi 8 pour .NET

Olivier Dahan

ISBN : 2-212-11309-9

EYROLLES

6

Concevoir ses interfaces

Le présent chapitre a pour but de vous sensibiliser par quelques exemples à l'amélioration de l'interface de vos applications loin de toutes les théories que vous pouvez découvrir dans des ouvrages spécialisés. Dans un premier temps, nous nous intéresserons à la différence entre VCL .NET et Windows Forms, deux visions de la conception des IHM entre lesquelles le développeur Delphi doit faire un choix. Nous prendrons ensuite quelques exemples que nous commenterons largement et pour lesquels nous vous proposerons des solutions d'implémentation sous VCL .NET et sous Windows Forms.

La conception des IHM

Un bon logiciel est un logiciel qui sait se faire oublier, qui fonctionne sans avoir à se poser de question. L'interface homme-machine (IHM) joue pour beaucoup dans une telle réussite.

Réussir une interface, c'est arriver à ce qu'elle soit transparente et que l'utilisateur puisse exercer son métier (le but) sans se focaliser sur l'application (l'outil). Les choix de conception sont essentiels pour atteindre l'objectif ultime, l'effacement total de l'interface à un point tel que l'utilisateur puisse prendre en main le logiciel sans même s'apercevoir qu'il manipule quelque chose de nouveau. Penser le travail de l'autre et l'assimiler pour le mettre en valeur est toujours frustrant vis-à-vis de son propre travail. Mais la prochaine fois que vous tiendrez un marteau en main, pensez à l'enfer que serait de planter ce clou si les concepteurs de marteaux se conduisaient comme certains informaticiens. L'auteur est certain que vous ne vous êtes jamais posé la moindre question à propos d'un outil si trivial. Et pourtant… Si le manche avait une forme biscornue de conte de fées, si la masse devenait ronde ou totalement plate pour satisfaire le design à la

mode du jour… Le marteau est un outil et il doit s'effacer devant le but de l'utilisateur. Toutefois, vous aurez remarqué qu'il existe de nombreux modèles de marteaux adaptés à diverses tâches et que certaines marques jouissent d'une plus grande réputation que d'autres auprès des professionnels. Cela correspond à certaines raisons : l'originalité, la passion, l'envie de plaire, qui au lieu d'être passée par la modification de l'apparence habituelle, s'est exprimée sur le fond, la conception, le choix judicieux des matériaux, l'équilibrage du manche et de la masse ainsi que de nombreux détails dont vous seriez surpris d'apprendre l'existence à propos d'un outil qui vous semble si familier et si rustique ! Que ce marteau puisse devenir votre inspiration dans la voie à suivre pour concevoir des interfaces homme-machine intelligentes !

Un logiciel n'est qu'un outil destiné le plus souvent à des utilisateurs sans grande formation en informatique qui sont très vite perturbés par une interface trop « originale ». L'objectif de l'informaticien est de valoriser le travail de l'utilisateur, de lui rendre moins pénible les tâches stupides et répétitives, de le seconder pour qu'il n'oublie pas un détail, de l'avertir de ses possibles erreurs voire les prévenir, tout en étant tolérant à l'égard de celles qu'il commettra inéluctablement.

L'invention la plus imaginative en termes d'interface ? Non, vous n'y êtes pas… ce n'est pas cette animation sous Flash qui en met plein la vue sur le site d'un de vos amis, ce n'est pas non plus cette fenêtre aux formes de vaisseau spatial du dernier lecteur multimédia à la mode. Cette innovation majeure se résume en un mot et une lettre : *contrôle-z*. Le droit à l'erreur, la possibilité de revenir en arrière sans tout recommencer. Voici un superbe exemple où la technique, la créativité d'un informaticien est mise en valeur. Tout le monde comprend à quoi cela sert, tout le monde s'en sert sans se poser de question. Pourtant, cette simple possibilité d'annuler la dernière action peut complexifier la programmation sans que l'utilisateur n'en sache rien. Pensez à un simple « Paint ». Sans possibilité d'annuler la dernière action, ce n'est qu'un canevas qui se remplit de traits au fur et à mesure que l'on dessine dessus, une suite d'éléments qui se code en quelques minutes. Pour supporter l'annulation de toutes les actions, il faut conceptualiser l'application à un niveau bien plus élevé : prendre en compte les différents outils de dessin, gérer une liste des actions exécutées, savoir redessiner tout le dessin action par action. Une bonne interface passe ainsi par un « effort de conceptualisation » et souvent par plus de code qui ne se voit pas et dont il serait vain de vouloir expliquer le fonctionnement à l'utilisateur. Il clique, ça marche, c'est simple pour lui, il pense que c'est simple aussi pour vous… N'en soyez pas frustré, bien au contraire ! Pensez simplement que vous avez atteint votre but en réalisant un magnifique et très fonctionnel marteau…

Un bon logiciel n'est au final qu'une suite de bonnes décisions qui toutes influencent, directement ou indirectement, l'ergonomie générale du produit. Après avoir abordé le choix Windows Forms vs VCL .NET, nous tenterons, par des exemples simples, d'illustrer cette nécessaire quête du confort de l'utilisateur. Loin de toute théorie, nous pensons que c'est par des petites choses concrètes qu'un concept a le plus de chance d'être

compris. L'ergonomie est un métier, de nombreuses publications existent sur ce sujet : n'hésitez pas à lire ces ouvrages[1].

VCL ou Windows Forms ?

Avec Delphi .NET, un problème nouveau se pose au développeur : quel outil choisir pour concevoir ses interfaces ? Windows Forms ou VCL.NET ? En toute honnêteté, disons d'emblée qu'il n'y a aucun argument décisif qui puisse permettre d'apporter une réponse nette, tranchée et définitive à cette question. Les deux approches ont leurs intérêts et toutes deux permettent de concevoir des interfaces visuelles. Néanmoins, nous verrons aussi que la VCL.NET n'est pas forcément qu'un outil de transition.

Les Windows Forms (appellation semble-t-il aujourd'hui préférée par Microsoft à WinForms) représentent une somme importante de classes parfaitement adaptées à la conception d'interfaces sous Windows. Les Windows Forms sont donc exclusivement dédiées à la plate-forme Windows et ce, sous .NET. C'est un premier point important de comparaison puisque la VCL existe sous Win32, Linux (CLX) et .NET, et que des applications conçues dans l'esprit d'un portage entre ces trois plates-formes peuvent être compilées nativement pour chacune d'elles. En effet, si .NET se veut une plate-forme indépendante du matériel, ce qui est vrai sur le papier, sa bibliothèque de développement pour Windows est totalement liée à cet environnement. La VCL marque donc un point pour sa portabilité avérée.

Comme nous le disions dans un chapitre précédent, l'avenir de Windows s'appelle LongHorn et ce système d'exploitation sera basé sur XAML pour l'interface. XAML est en quelque sorte un mélange de HTLM, de Flash et de PDF gérant une interface totalement unifiée entre les clients PC et Internet. La bibliothèque de classes Windows Forms n'est pas compatible avec XAML, l'approche étant très différente. Microsoft prépare d'ailleurs un nouveau framework qui porte pour l'instant le nom de Avalon. D'un autre côté, et même si aucune annonce officielle n'a été faite en ce sens, la VCL a prouvé son pouvoir d'adaptation et on peut parier que Borland écrira une VCL basée sur XAML ou au moins une passerelle simplifiant le portage. Il ne s'agit pas d'élucubrations venant d'un « fan » de Borland puisque cette vision de l'avenir de la VCL est celle que présente Danny Thorpe, architecte du compilateur Delphi et chef de l'équipe de développement sous .NET au travers d'un papier paru sur le site américain[2] de Borland. À ce jour, Microsoft ne propose aucune passerelle de ce genre, ne serait-ce que pour aider les déve-

---

1. Voici quelques références :
(a) Crampes (J.B), *Interfaces graphiques ergonomiques*, Ellipses.
(b) Gillet (J.M.), *L'interface graphique*, Masson.
(c) Baroca (C.)*, Graphismes et ergonomie des sites Web*, Dunod.
Ces ouvrages sont disponibles à la librairie Eyrolles à Paris ou sur le site Web de cet éditeur.
2. Thorpe (Danny), « Why VCL for .NET », *Borland Developer Network*, article n° ID 31983, 9 mars 2004.

loppeurs C++ ou VB à passer sous Windows Forms. Ils ne feront certainement pas plus d'efforts lors du passage à XAML, il s'agirait alors d'une grande nouveauté pour cet éditeur. Borland nous a prouvé que la VCL pouvait traverser les technologies, Win16, Win32, QT sous Linux, .NET aujourd'hui. Même si ce n'est à l'heure actuelle qu'une spéculation, cela offre ici encore un léger avantage à la VCL.

Si vous venez du monde Java ou Microsoft, nous vous souhaitons la bienvenue dans le monde Borland et l'argument qui vient ne vous concerne pas. Par contre, si vous êtes déjà un développeur Delphi, cas le plus fréquent des utilisateurs de Delphi .NET, alors vous avez forcément un investissement à rentabiliser. Cet investissement peut être du code déjà écrit, des dialogues pour vos logiciels, des fenêtres de saisies. La VCL .NET vous permettra de reprendre rapidement cet acquis et d'être efficace tout de suite. Même si vous ne souhaitez pas reprendre des fenêtres existantes, vous avez un autre investissement à rentabiliser : votre connaissance de la VCL. Vous avez travaillé avec cette bibliothèque, vous la connaissez, autant valoriser cet acquis et gagner en compétitivité en utilisant la VCL .NET. Ce troisième point est encore à l'avantage de la VCL.

L'argumentation technique pourrait-elle renverser cet avantage ? Pas si sûr ! La bibliothèque Windows Forms est native .NET et a été conçue pour cette plate-forme. Mais il faudra attendre LongHorn pour que les API Windows soient fournies en code managé… Il en résulte que la bibliothèque Windows Forms est construite sur des appels directs aux API Win32. Par exemple, c'est bien un appel à `CreateWindow` qui est exécuté pour créer une fenêtre, de même que `WndProc` est *hookée* pour gérer l'arrivée des messages Windows en destination de la fenêtre. Les Windows Forms ne sont ainsi qu'un habillage au-dessus des API Win32 de Windows. Que cet habillage soit bien réalisé est un fait certain, mais techniquement, Windows Forms se repose sur les API Win32, exactement ce que fait la VCL .NET… Il n'y a donc aucun avantage technique particulier pour l'une ou l'autre de ces deux solutions.

Pour aller plus loin, on peut évoquer le support du Compact Framework (CF), la version light de .NET pour assistants personnels principalement. Qu'en est-il dans la réalité ? Dans les faits, le CF propose un sous-ensemble du framework .NET, et qui dit sous-ensemble dit inéluctablement carences. Certaines classes sont donc absentes de CF, ce qui signifie que les applications écrites pour Windows Forms ne sont pas portables directement. Il y a même des différences pour certaines classes, ce qui complexifie encore le portage. Même si vous concevez une application sous Windows Forms qui respecte en tout point les limitations de CF, elle ne pourra pas s'exécuter sous ce dernier car les assemblages utilisés (même ceux portant le même nom et décrivant les mêmes classes) sont signés par un nom fort qui diffère des assemblages Windows Forms. Cela oblige en pratique à recompiler l'application après avoir effacer les références aux assemblages Windows Forms et à ajouter les références à ceux de CF. Autant dire que la portabilité est loin d'être directe. Delphi .NET supportera CF, mais cela n'est pas encore le cas. Le bilan technique de la comparaison est ici mitigé. CF est différent de Windows Forms, il n'y a donc pas de réel avantage à choisir ce dernier plutôt que la VCL .NET dans le seul but de faire du code portable. D'un autre côté, Delphi n'intègre pas encore le CF, mais ce n'est

qu'une question de *release* qui n'a que peu à voir avec notre débat. Il est donc très difficile de conclure à un avantage réel pour Windows Forms, même s'il faut le concéder, cette bibliothèque est plus proche de CF que ne l'est VCL .NET.

L'interopérabilité ne permet pas non plus de trancher. Windows Forms est un habillage natif .NET pour les API Win32, la VCL .NET l'est aussi et le code compilé utilisant l'une ou l'autre de ces bibliothèques est natif .NET et peut être réutilisé avec tous les autres langages fonctionnant sur .NET. Comme nous l'avons vu dans les chapitres de présentation des nouveautés du langage, il est même possible d'appeler du code Delphi .NET depuis des applications Win32 et cela le plus simplement du monde. Sur ce point technique essentiel, il nous faut convenir que les deux solutions sont donc logées à la même enseigne et qu'aucune des deux ne s'impose sur l'autre.

Côté déploiement, il n'y a pas non plus de grandes différences puisque tous les langages .NET imposent la fourniture de certaines DLL. VB.NET impose au minimum sa DLL qui pèse dans les 300 ko, JScript oblige la présence d'une DLL de 700 ko environ et C#, selon ce que l'on déploie, il faut compter entre 3 et 20 Mo de surcharge. Au minimum, les applications Delphi .NET réclament une DLL de 64 ko, ce qui est fort peu, auquel s'ajouteront les DLL de la VCL si vous désirez les fournir à part. Néanmoins, ces dernières peuvent être liées directement à l'exécutable en imposant quelques centaines de kilo-octets supplémentaires. Tout langage confondu, Delphi n'oblige à rien de plus ou de moins, et le poids de la VCL .NET ne change rien au bilan.

Pour résumer, voici la synthèse de ce tour d'horizon :

- Portabilité Win16, Win32, CLX,.NET : avantage à la VCL.
- Portabilité et comptabilité avec Avalon sous LongHorn : la VCL et Windows Forms ne fonctionnent pas sous XAML. La VCL propose certainement plus de solutions que Windows Forms.
- Réutilisation du code des interfaces Win32 existantes: avantage à la VCL.
- Valorisation de votre savoir-faire sous Win32 : avantage à la VCL.
- Qualité technique et indépendance vis-à-vis des API : égalité entre Windows Forms et la VCL qui sont conçus de la même façon.
- Comptabilité avec le Compact Framework : léger avantage aux Windows Forms.
- Interopérabilité entre langages : égalité entre les deux solutions.
- Déploiement : égalité entre les deux solutions.

À chacun d'apprécier des points étudiés ici et de leur donner un poids pour établir un score final. De façon neutre, si nous attribuons la même valeur à chacun, la VCL .NET remporte avec une bonne avance la comparaison. Malgré tout, il se peut que le poids d'un argument pèse plus lourd ponctuellement pour un développement précis ou un contexte donné. Nous ne pouvons ici que vous fournir des éléments de réflexion, à vous de trancher.

L'ergonomie des dialogues

Il y a beaucoup de choses à dire sur l'ergonomie des fiches et des dialogues, parce que ce sont, in fine, les parties les plus présentes de l'interface d'une application.

Nous aborderons ici quelques thèmes qui semblent le plus souvent être oubliés ou mal compris. Précisons à nouveau qu'il ne s'agit ni de théoriser ni d'être exhaustif mais d'attirer votre attention par l'exemple plus que par un discours théorique, sur la démarche consistant à créer des interfaces ergonomiques.

Tant que faire se peut, nous présenterons les exemples à la fois pour la VCL .NET et pour les Windows Forms.

Le focus d'entrée

Il n'y a rien de plus irritant qu'une fiche de saisie ne positionne pas correctement le focus sur le premier champ ayant le plus d'intérêt dès son ouverture.

VCL

La VCL met à la disposition du développeur, dans l'objet `TForm`, la propriété `ActiveControl`. Cette propriété permet, durant l'exécution, de connaître le contrôle ayant le focus et permet d'indiquer au premier affichage quel contrôle doit recevoir le focus. C'est de cette dernière façon que l'on l'exploitera systématiquement.

Il ne faut donc jamais oublier, lors de la conception d'un dialogue, de renseigner la propriété `ActiveAcontrol` des fiches. Si cela n'est pas suffisant ou si cela réclame des tests particuliers, il faut effectuer cette opération dans le gestionnaire d'un événement de la fiche. Vous noterez que placer le focus dans `OnCreate` génère une erreur à l'exécution « impossible de focaliser une fenêtre désactivée ou invisible ». Cela est dû au fait que l'objet à focaliser n'a pas terminé son initialisation et qu'il n'est pas encore prêt à accepter un message de focalisation. Le moyen de régler ce problème est, en général, de placer le changement de focus dans l'événement `OnShow` en utilisant un drapeau pour ne faire l'opération qu'une fois. Il existe une autre possibilité, plus lourde, qui consiste à créer un message Windows utilisateur et à l'envoyer dans le `OnCreate` en destination de la fenêtre. Dans le code qui gère la réponse à ce message, le focus peut être manipulé. La raison en est que le message est posé sur la pile et ne sera pris en compte qu'une fois la création de la fiche totalement effectuée.

Windows Forms

Les Windows Forms ne disposent pas d'événements totalement similaires à ceux des fiches de la VCL mais il est possible de trouver des équivalences. C'est le cas de l'événement `Load` qui est déclenché à l'instanciation de la fiche. Pratiquement, cet événement est similaire à `OnCreate` d'une `TForm` et l'on y retrouve les mêmes limitations que celles concernant le focus.

Pour régler le problème du focus à l'entrée d'une fiche, il faut ainsi préférer l'événement `Activated`. Toutefois, celui-ci se déclenche à chaque fois que la fiche est activée. Si nous plaçons le focus d'entrée de fiche dans le gestionnaire de cet événement, cela marche très bien, mais le focus est à nouveau forcé si l'utilisateur bascule d'une fiche à l'autre ce qui n'est pas souhaitable. Pour régler ce problème il faut créer une variable dans la fiche, par exemple `Initialized`, qui testée dans le code plaçant le focus afin d'éviter de reforcer ce dernier de façon parasite.

Extrait de la déclaration de la Windows Form :

```
  private
    { Déclarations privées }
    var Initialized : Boolean ; // le drapeau
```

Extrait du code :

```
// Gestionnaire d'événement Activated de la fiche
procedure TWinForm.TWinForm_Activated(sender: System.Object; e: System.EventArgs);
begin
 InitForm; // procédure globale d'initialisation
end;

// procédure chargée des initialisations, dont le focus d'entrée
procedure TWinForm.InitForm;
begin
 if initialized then exit;
 TextBox2.Focus; // le contrôle à focusser
 initialized := true;
end;
```

Comme le montre le code ci-dessus, nous avons choisi de créer une méthode `InitForm` qui est appelée par le gestionnaire d'événement `Activated` plutôt que de mettre le code directement dans ce dernier. La raison est simple : nous pouvons avoir besoin de gérer bien d'autres initialisations au lancement de la fiche. À la place d'avoir un peu de code d'initialisation dans le gestionnaire `Activated` et un peu ailleurs, il est plus rationnel de tout centraliser. À noter que `Load` peut être utilisé pour les initialisations mais comme l'on ne peut pas y traiter le focus et pour éviter, une fois encore, l'éparpillement du code, il est préférable de tout centraliser.

Le focus dynamique

Gérer le focus d'entrée d'un dialogue est une chose importante, mais il ne faut pas oublier non plus que durant l'utilisation du dialogue, il est souvent nécessaire de forcer le focus pour améliorer l'ergonomie de la saisie.

Par exemple, sur une fiche de type saisie sur une base de données, lorsque l'utilisateur déclenche (par quelque moyen) la création d'un nouvel enregistrement, le focus doit être transféré au premier contrôle de celui-ci automatiquement...

VCL

Cela s'effectue facilement en prenant l'événement le plus proche de la source de l'action. Dans ce cas, il s'agit de `OnNewRecord` de l'objet (descendant de) `TDataSet`. Une simple ligne de code de type `DBEdit1.SetFocus` suffit (si le premier contrôle pertinent de la fiche de saisie est `DBEdit1`, bien entendu). Dans les cas particuliers de fiches à panneaux multiples ou de champs variables, selon le contexte, ajoutez les tests nécessaires, au minimum un appel à la méthode `CanFocus` du contrôle pour éviter le déclenchement d'une exception vers l'utilisateur (qui ne la comprendrait pas) dans le cas où le contrôle ne peut pas accepter la focalisation (s'il n'est pas visible principalement).

Windows Forms

Les contrôles Windows Forms disposent d'une propriété, `Focused`, permettant de savoir s'ils possèdent ou non le focus et une méthode `Focus()` permettant de leur donner le focus. Le code exemple précédent en montre l'utilisation. Dans le cas d'une fiche de saisie de données, il faut bien entendu utiliser d'autres événements que ceux évoqués plus haut. La gestion des données sous Windows Forms étant différente de celle de la VCL nous reviendrons sur ces aspects dans un chapitre consacré à ce sujet.

La dynamique du focus

Parmi les choses agaçantes, voire rédhibitoires (l'auteur a déjà désinstallé des logiciels pourtant puissants pour cette seule et unique raison), les déplacements anarchiques du curseur tiennent une place d'honneur. Il n'est pas normal en effet que le curseur puisse se « balader » un peu partout sur une fiche, et pire, dans n'importe quel ordre. Il est donc essentiel de veiller à l'ordre des tabulations.

De même, si certains boutons n'ont pas un intérêt vital dans l'optique d'une saisie manuelle, il n'est pas nécessaire de les inclure dans la chaîne de tabulation (propriété `TabStop` des contrôles aussi bien sous VCL que Windows Forms), ce qui est le cas par défaut. Et même si leur activation depuis le clavier a un sens, il est préférable de programmer un accélérateur plutôt que de voir le curseur, lors d'une saisie, être obligé de faire le tour de tous les boutons pour revenir au premier contrôle (un accélérateur de bouton se programme en ajoutant le symbole « `&` » devant une lettre de son invite). Bien entendu, dans certains cas, il est préférable de laisser les boutons dans le circuit de tabulation. Répétons-le une fois encore : ce chapitre fait appel au bon sens, seule instance capable de trancher pour une application donnée...

Les grilles n'échappent pas à cette logique, qu'il s'agisse de grilles orientées, données ou non. Si une telle grille est placée sur une fiche et si elle ne sert pas directement à la saisie (donc uniquement à la visualisation d'informations), il est indispensable de veiller à ce que le focus ne se promène pas à l'intérieur des cases. Ainsi, la touche de tabulation doit permettre de sortir de la grille pour ne pas que le curseur reste coincé à l'intérieur. Il est même préférable, le plus souvent, de supprimer ces grilles de la chaîne de tabulation.

VCL

Sous VCL .NET, le réglage du circuit de tabulation se fait depuis l'EDI en mode conception en agissant sur la propriété `TabOrder` des contrôles dérivant de `TWinControl`.

Windows Forms

La propriété indiquant l'ordre de tabulation des contrôles s'appelle `TabIndex` et joue le même rôle que `TabOrder` sous VCL.

Menu, options, boutons : grisé ou invisible ?

L'une des questions qui rongent le développeur pointilleux et soucieux d'ergonomie est celle du choix grisé ou invisible pour les boutons, items de menus et autres éléments d'interface. Rappelons alors quelques règles de bon sens en accord avec l'esprit de Windows.

Un élément d'interface est grisé (`Enabled:=False` sous VCL et Windows Forms) lorsqu'il est indisponible mais reste tout de même visible car il peut servir à un autre moment.

Lorsqu'il est invisible, l'utilisateur ne peut pas se poser la question de savoir s'il est disponible ou non.

Qu'il est bon parfois d'enfoncer les portes ouvertes...

On en conclut que :

- L'invisibilité (`Visible:=False` VCL et Windows Forms) supprime une fonction que l'utilisateur n'a même pas à connaître. L'invisibilité ne donne aucune information, au contraire, elle soustrait du champ visuel une fonction inutile.
- L'indisponibilité (`Enabled:=False`) déconnecte une fonction de façon temporaire car le contexte l'impose. Cette indisponibilité visible renseigne l'utilisateur sur l'état de l'application.

De là, on peut dire que :

- Si une fonction n'est pas disponible pour des raisons externes (liées à l'utilisateur, à l'état d'un périphérique…), les éléments d'interface la représentant doivent être grisés. C'est le cas d'un bouton Calculer si aucune formule n'a été saisie, d'une entrée de menu Coller si le presse-papiers est vide, d'un bouton de fermeture d'un écran de détail si celui-ci n'a pas été ouvert…
- Si une fonction n'est pas disponible pour des raisons internes (fonction non autorisée pour le profil de l'utilisateur, fonction déconnectée car n'ayant pas de sens dans le contexte…), les éléments d'interface la représentant doivent être invisibles. C'est le cas d'un bouton Création sur une fiche de données permettant la création mais utilisée dans un contexte où cette création n'est pas autorisée, d'un item de menu non accessible à l'utilisateur car celui-ci n'a pas les droits d'accès nécessaires…

Échappement et Validation

Lorsqu'on manipule souvent un logiciel, on attrape très vite des automatismes grâce aux raccourcis clavier, qu'il s'agisse de ceux de l'application ou de Windows. Dès lors, moins on quitte les mains du clavier, plus l'application est agréable à utiliser.

Si nous ne pouvons que vous conseiller de soigner les raccourcis clavier de vos applications et de veiller à ce qu'elles soient parfaitement utilisable sans souris, nous insisterons aussi sur la gestion des touches d'échappement (*escape*) et de validation (*return*). Il est un peu stupide en effet de constater que certaines applications intègrent fort bien la gestion du clavier mais qu'il faille absolument quitter ce dernier pour prendre la souris et cliquer sur la croix de fermeture de la fiche pour mettre fin au dialogue. Cela est surtout valable pour toutes les fiches utilisées en empilement dans une application. Cela serait le cas d'une fiche article appelée pendant la saisie d'une commande ou de la fiche représentant appelée pendant la consultation d'une fiche client.

Il existe certes des raccourcis Windows comme Alt + F4 pour la fermeture, mais ce raccourci est bien plus inconfortable que la simple touche d'échappement. Prévoyez dès lors dans toutes vos fiches la possibilité de les fermer en appuyant sur la touche d'échappement ou la touche de validation. On utilisera l'échappement systématiquement, quel que soit le type du dialogue. La gestion de la touche de validation sera ajoutée uniquement si la fiche contient une saisie devant être acceptée.

VCL

Cela se fait de façon très simple en plaçant à `True` la propriété `KeyPreview` de chaque `TForm` et en fournissant un gestionnaire à l'événement `OnKeyPress` de ces mêmes objets. Si vous avez opté pour le principe d'une fiche mère dont héritent toutes les autres fiches de votre application (ou une hiérarchie de classes selon le type de dialogue), vous n'aurez à faire cette modification que dans la classe mère. Ceux qui hésitent encore à utiliser les méthodes de développement objet pour ce qu'elles savent bien faire devront faire la modification dans chacune des fiches du projet (La Fontaine aurait conclu quelque chose comme « cette leçon vaut bien une classe mère sans doute ! »).

Si la fiche ne doit pas retourner de code modal, le code proprement dit de gestion de l'événement `OnKeyPress` est de ce type :

```
If Key in [#13,#27] then
 begin
  Key := #0 ;
  Close ;
 end ;
```

S'il s'agit d'une fiche de dialogue devant retourner un code modal, le code est plutôt comme ce qui suit :

```
If Key = #13 then modalresult := mrOk else
  If Key = #27 then modalresult := mrCancel;
If Key in [#13,#27] then Key := #0;
```

La principale nuance entre les deux versions étant la différenciation du code modal de retour selon l'appui sur la touche d'échappement ou de validation.

`Key` est un paramètre variable donné par l'événement. En entrée de méthode, il contient le code de la touche enfoncée que vous pouvez modifier. Pour ignorer une touche, l'astuce utilisée dans les codes ci-dessus consiste à retourner ce paramètre à zéro (en ASCII). Aucun bip désagréable ne se fera entendre et la touche sera ignorée.

Windows Forms

Les fiches Windows Forms possèdent aussi une propriété `KeyPreview` qui doit être mise à `True` pour intercepter les événements clavier. L'événement à gérer s'appelle alors `KeyPress`. Les ressemblances avec le code VCL reste ici aussi très frappantes.

La seule véritable nuance s'effectue dans la façon de traiter le caractère tapé dans le gestionnaire d'événement :

```
procedure TWinForm1.TWinForm1_KeyPress(sender: System.Object;
➥e: System.Windows.Forms.KeyPressEventArgs);
begin
 if e.KeyChar in [#13,#27,'A'] then
  begin
   messagebox.Show('ESC RET !');
   e.Handled := true;
  end;
end;
```

On remarque que le caractère frappé est passé dans le paramètre « `e` » qui est une classe exposant deux propriétés, `KeyChar` le caractère tapé et `Handled` une valeur booléenne permettant d'indiquer au gestionnaire que l'événement a été géré. La propriété `e.KeyChar` est ainsi équivalente à l'argument `Key` de l'événement VCL et `e.Handled:=true` est équivalent à `Key:=#0`.

Stratégie de placement et retaille automatique

Il semble important de glisser quelques mots à propos de la stratégie de placement des composants visuels sur les fiches et des problèmes de retaille automatique de ces derniers car ce point est souvent négligé dans les applications. Il est dommage de frustrer un utilisateur après tous les efforts fournis pour qu'il ait entre les mains une application d'exception (c'est-à-dire sans… exception justement !).

Concernant le placement des objets visuels, essayez de raisonner par zones ou régions, chacune étant contenue dans un panneau (`TPanel` sous VCL ou `Panel` sous Windows Forms) ou un objet équivalent. Ensuite, choisissez avec précision l'alignement de chacun d'eux en faisant en sorte que les objets pouvant le plus bénéficier d'un changement de taille de la fenêtre se trouvent alignés en `alClient` sous VCL (et `Dock:=Fill` sous Windows Forms) dans le panneau qui lui-même aura ce type d'alignement.

L'effet recherché est de permettre, lors d'un agrandissement de la fenêtre, que les objets pouvant en profiter voient leur taille augmenter.

Par exemple, une fiche de saisie contenant un champ mémo doit être construite de telle façon que ce soit ce dernier qui s'agrandisse automatiquement lors d'une maximisation de la taille de la fenêtre. De même, si la fiche contient une liste de données, il sera intéressant que cette dernière s'agrandisse en même temps que la fenêtre.

Pourquoi ? simplement parce que si un utilisateur dispose d'un écran de meilleure résolution et qu'il souhaite agrandir une fenêtre, ce n'est pas pour voir le texte passer en police de corps 30 mais bien de profiter de l'espace supplémentaire pour voir plus de données…

VCL

En plus d'une stratégie exploitant des panneaux, vous pouvez aussi tirer un grand bénéfice de l'exploitation des propriétés `Anchors` et `Constraints`. La première permet de préciser sur quel bord le contrôle est ancré, la seconde permet d'imposer des tailles maximales et minimales au contrôle.

En utilisant correctement `Anchors`, vous pouvez faire en sorte que des contrôles qui ne peuvent pas être alignés sur un bord entier ou dans tout l'espace client puissent bénéficier d'un élargissement ou d'un agrandissement de la fiche. Par exemple, un champ d'édition pourra être ancré à droite et à gauche afin de s'allonger proportionnellement à la fiche qui le contient. Un champ de type mémo sera ancré en haut et en bas pour s'agrandir dans les mêmes circonstances.

N'oubliez pas d'exploiter `Constraints` pour fixer une taille minimale à vos fiches. Vous pouvez le faire contrôle par contrôle mais cela est souvent une perte de temps. Visuellement, en conception, il est facile de diminuer la taille d'une fiche jusqu'à ce que ce qu'elle contienne soit à la limite de l'acceptable. Vous notez alors la hauteur et la largeur que vous inscrivez dans les contraintes minimales correspondantes de la fiche. S'il y a une limite visuelle en agrandissement (ce qui est plus rare), effectuez la même opération en agrandissant la fiche jusqu'au maximum acceptable visuellement puis inscrivez les largeur et hauteur limites dans les contraintes maximales ad hoc.

Vous comprenez ici que l'éternel débat sur l'auto-adaptation des fiches à la résolution des écrans n'a que peu de sens : vos fiches doivent être conçues pour tenir dans la taille écran minimale fixée par le cahier des charges et disposer d'une stratégie de placement comme exposé ici pour profiter des meilleures résolutions et permettre d'afficher plus d'information.

Windows Forms

Les raisonnements restent identiques, seules les propriétés à utiliser diffèrent. La gestion de l'alignement sur la fiche s'effectue, par exemple, via la propriété `Dock` de la fiche qui peut prendre des valeurs fonctionnellement identiques à la propriété `Align` des contrôles VCL.

Si vous optez pour une gestion du placement par les ancres, la propriété à utiliser s'écrit `Anchor` au lieu de `Anchors`… Les similitudes en deviennent troublantes…

Concernant la gestion des contraintes de taille, au lieu de la propriété `Constraints` de la VCL, vous utilisez les propriétés `MaximumSize` et `MinimumSize`.

La gestion du presse-papiers

Le presse-papiers est un outil essentiel pour l'utilisateur, il est le mécanisme le plus simple qui illustre la coopération entre les diverses applications et reste le symbole même des environnements graphiques, comme celui des machines Apple ou Windows.

Avant même l'invention du DDE ou de l'OLE, la seule réelle coopération qui faisait la différence entre ces environnements et les autres était justement de pouvoir copier des informations d'une zone vers une autre et ce, même depuis et vers des applications différentes.

Alors qu'elles semblent négligées par bon nombre de développeurs, les opérations sur le presse-papiers sont ainsi essentielles dans une application professionnelle et elles doivent faire l'objet d'une attention toute particulière.

Quand copier n'est pas suffisant

Sous Delphi, les contrôles de la VCL automatisent la gestion du Couper/Copier/Coller. Mais le sens de ces opérations est orienté « contrôle » alors qu'un humain travaille plutôt sur des concepts, ce que, bien entendu, le meilleur des environnements de développement ne sait absolument pas gérer...

Prenons l'exemple d'une fiche individuelle client au sein d'une gestion commerciale. Elle contiendra certainement l'adresse du client. Techniquement, celle-ci sera matérialisée selon toute vraisemblance par les contrôles du type adresse 1 et 2, code postal et ville.

Les fonctions presse-papiers automatiques de Delphi concerneront chaque contrôle mais pas la totalité du groupe de contrôles formant l'adresse qui n'est qu'un concept.

Si l'utilisateur souhaite copier l'adresse pour écrire un courrier avec Word, la gestion de base sera très lourde (plusieurs va-et-vient entre Word et l'application pour copier/coller chacun des contenus des contrôles).

Pour offrir un plus grand confort, il est nécessaire de proposer au minimum une fonction Copier spéciale qui copiera toute l'adresse en une seule opération. Ce traitement peut être automatique (selon le contexte de votre application) ou bien être matérialisé par un bouton ou une entrée dans le menu Édition (que toute bonne application doit proposer).

VCL

Techniquement, vous devrez piloter l'opération de copie par code en utilisant l'unité `ClipBrd`. Pour grouper plusieurs informations dans une même opération « copier », il suffit de les mettre bout à bout, séparées par un caractère CR ou une paire CR/LF, puis de placer l'ensemble dans le presse-papiers.

Le code suivant démontre cette logique :

```
Procedure Tform1.CopierAdresseClient;
Const CRLF = #13#10;
Begin
 Clipboard.AsText := MaTable.FieldByName('Société').AsString + CRLF +
MaTable.FieldByName('Contact').AsString + CRLF +
MaTable.FieldByName('Adresse1').AsString + CRLF +
    MaTable.FieldByName('Adresse2').AsString + CRLF +
    MaTable.FieldByName('CP').AsString + ', ' +
MaTable.FieldByName('Ville').AsString ;
End;
```

Cette technique peut largement être exploitée dans vos applications qui n'en seront que plus conviviales, d'autant que cela ne réclame que peu de code.

Parallèlement, il est bon de constater que nous n'avons traité ici qu'un seul sens : la copie. Pour le collage, l'opération est plus complexe, notamment lorsque elle est réalisée entre plusieurs applications différentes.

Par exemple, pour récupérer une suite de lignes copiées depuis Word dans la série de contrôle formant l'adresse dans votre application, il faudrait découper le *buffer* du presse-papiers puis appliquer tous les contrôles de saisie avant d'accepter ou de refuser le collage (par exemple, tester la nature numérique d'un code postal ou la conformité syntaxique d'un champ d'adresse Internet).

Ces difficultés auxquelles vient s'ajouter la nécessité de détourner les messages Windows entrant de type « coller » pour les traiter de façon spécifique, font que les applications ne gèrent presque jamais ce type de collage multicontrôle.

Par contre, il est tout à fait possible d'exploiter le copier/coller multi-élément au sein d'une même application qui, elle, peut reconnaître les constituants du groupe à coller. Delphi, pour les besoins de son IDE, enregistre un format spécial dans le presse-papiers pour les objets. C'est ce qui permet de copier/coller des composants sur une fiche en mode conception. Ainsi, votre application peut définir des descendants de `TComponent` qui contiennent des informations fortement typées (des propriétés de l'objet) servant aux opérations de presse-papiers complexes entre les divers modules du logiciel (structures dynamiques, objets…). Vous noterez que cette technique n'est pas exclusive, il est possible de poser dans le presse-papiers les mêmes données sous différents formats, le format texte devant être proposé systématiquement (si cela à un sens). Quand une zone d'adresse est copiée dans votre application, l'opération doit poser dans le presse-papiers une version texte de l'adresse (avec des `CR+LF` entre les champs par exemple) ainsi qu'une version objet réexploitable par votre application. Si l'utilisateur colle le contenu du presse-papiers dans cette dernière, elle sait utiliser le format objet pour reprendre chaque champ correctement. Si l'utilisateur colle le contenu du presse-papiers dans un traitement de texte, ce dernier exploite automatiquement la version texte et ignore la version objet. Selon ce principe, vous pouvez copier dans une multitude de formats ! Dans certains cas, il peut être utile de créer une image bitmap des données et de la mettre aussi dans le presse-papiers. Si l'utilisateur souhaite faire un collage dans un logiciel de dessin,

ce dernier récupérera la version bitmap. On peut aussi créer une version mise en forme en HTML, en RTF, en XML… Tous ces formats d'une même donnée peuvent être enregistrés en même temps, augmentant encore plus l'ergonomie de votre logiciel.

Généralement, les opérations de collage acceptant des formats inhabituels (texte ou image) font l'objet d'un menu spécifique. Ce dernier est souvent appelé Coller/spécial et son affichage dans vos applications s'effectue en scannant le contenu du presse-papiers et en énumérant les formats qu'il contient.

La possibilité de pouvoir manipuler des objets persistants dans le presse-papiers est une formidable ouverture qui peut simplifier le codage de vos applications tout en améliorant l'ergonomie (par exemple copier une fiche client entière pouvant être récupérée, même partiellement, par tous les modules de l'application ou même pour autoriser la duplication de fiches entières dans une base de données). L'utilisateur reste maître des « instants » et des « lieux » et le code, objet, peut être centralisé.

Pour illustrer ce mécanisme, voici le code principal d'un exemple que vous pourrez retrouver dans sa totalité sur le CD-Rom du livre :

```
{-------------------------------------------------------------------
 Unit Name: upp1 / Version VCL.NET
 Author:    od
 Purpose:   Démontrer le copier / coller des objets
-------------------------------------------------------------------}
unit upp1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls,
  ClipBrd,                  // indispensable pour gérer le presse-papiers
  System.ComponentModel; // ajouté par Delphi .NET

type
  // définition d'une classe conteneur pour
  // les informations à copier/coller
  TClient = class(TComponent)
   private
    FSociete : string;
    FContact : string;
    FAdresse : tstrings;
    procedure SetAdresse(ts:tstrings);
   public
    constructor Create(aowner:tcomponent); override;
    Destructor Destroy; override;
   published
    // seules les propriétés Published sont sauvegardées
    Property Societe: string read FSociete write FSociete;
    Property Contact:string read FContact write FContact;
    Property Adresse: tstrings read FAdresse write SetAdresse;
   end;
```

```
  // la fiche de test
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject); // copier
    procedure Button3Click(Sender: TObject); // effacer écran
    procedure Button2Click(Sender: TObject); // coller
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

// Classe TClient

// Init de la classe
Constructor TClient.Create(aowner:tcomponent);
begin
 inherited create(aowner);
 fAdresse := TStringlist.Create;
end;

// Destructeur
Destructor TClient.Destroy;
begin
 FreeAndNil(FAdresse);
 inherited destroy;
end;

// Modificateur du champ Adresse
procedure TClient.SetAdresse(ts:tstrings);
begin
 FAdresse.Assign(ts);
end;

// ***** fin de la classe TClient ****

// Opération COPIER
procedure TForm1.Button1Click(Sender: TObject);
var Client:Tclient;
begin
 Client := TClient.Create(self); // création du conteneur
 with Client do                  // remplissage du conteneur
```

```
 try
   Societe := edit1.Text;
   Contact := edit2.Text;
   Adresse := memo1.Lines;
   Clipboard.SetComponent(Client); // copie vers le presse-papiers
 finally
  FreeAndNil(Client);                // libération du conteneur
 end;
end;

// Vider les contrôles écran pour voir l'effet du COLLER
procedure TForm1.Button3Click(Sender: TObject);
begin
 edit1.Text:='';
 edit2.Text:='';
 memo1.Lines.Clear;
end;

// Opération COLLER
procedure TForm1.Button2Click(Sender: TObject);
var Client:TComponent; // on récupère un TComponent du clipboard
begin
 // le presse-papier contient-il un objet ?
 if not clipboard.HasFormat(CF_COMPONENT) then exit;
 // obtenir l'objet (Tcomponent)
 Client := clipboard.GetComponent(nil,nil);
 try
  // ce composant est-il un Tclient ?
  if not (Client is TClient) then exit;
  with TClient(Client) do
   begin // on décharge le conteneur dans les contrôles écran
    edit1.text:= Societe;
    edit2.text:= Contact;
    memo1.lines := Adresse;
   end;
 finally
  FreeAndNil(Client); // libération du conteneur
 end;
end;

initialization
 // Indispensable au fonctionnement : le recensement de la classe
 RegisterClass(TClient);
end.
```

Visuellement, la fiche ressemble à la figure 6-1 sur laquelle on peut voir trois champs de saisie (Société, Contact et Adresse) ainsi que trois boutons (Copier, Effacer et Coller).

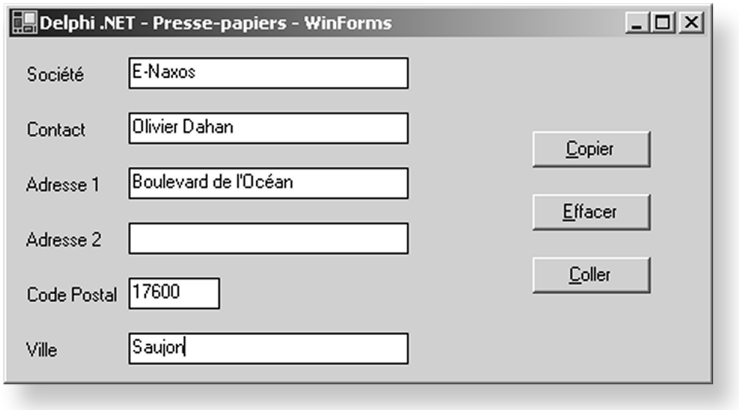
Figure 6-1
Copier/Coller multizone VCL .NET

On utilise l'application en saisissant du texte dans les trois contrôles et en effectuant un clic sur les trois boutons dans l'ordre : Copier, Effacer puis Coller. L'utilisation d'un bouton Effacer sert uniquement à vider les contrôles pour mieux voir ce que fait l'opération Coller. Les lecteurs de notre précédent ouvrage « Delphi 7 Studio » reconnaîtront cet exemple que nous avons conservé sans modifier une ligne afin d'illustrer l'extraordinaire portabilité du code Win32 sous Delphi .NET.

Windows Forms

Les différences avec la VCL sont ici plus nombreuses, ce qui n'empêche nullement de mettre en place une logique équivalente comme nous allons le voir. La figure 6-2 nous montre la fiche servant de support à l'exemple.

Figure 6-2
Copier/Coller multizone version Windows Forms



Plutôt que de vous fournir le code complet de l'unité que vous pouvez trouver sur le CD-Rom du livre, nous allons étudier certains aspects très spécifiques à l'implémentation sous Windows Forms. Pour commencer, nous avons créé une classe pour mémoriser les divers éléments, comme nous l'avions fait sous VCL :

```
    [Serializable]
    TAdresse = Class (System.object)
     strict private
      FSociete : string;
      FContact : string;
      FAdresse1: string;
      FAdresse2: string;
      FCP      : string;
      FVille   : string;
     protected
     public
      constructor Create;
      property Societe : string read FSociete write FSociete;
      property Contact : string read FContact write FContact;
      property Adresse1: string read FAdresse1 write FAdresse1;
      property Adresse2: string read FAdresse2 write FAdresse2;
      property CP      : string read FCP      write FCP;
      property Ville   : string read FVille   write FVille;
     end;

implementation
constructor TAdresse.Create;
begin
  inherited;
  // initialisation à vide pour le bindings
  // les références null ne sont pas acceptées.
  Fsociete := '';
  FContact := '';
  FAdresse1 := '';
  FAdresse2 := '';
  FCP := '';
  FVille :='';
end;
```

Première constatation, la classe dérive de `System.Object` et non de `TComponent`. Puisque tous les objets sont sérialisables sous .NET, autant utiliser l'ancêtre le plus simple qui soit. Nous aurions pu utiliser la même démarche sous VCL .NET puisque Delphi pour .NET n'impose pas qu'une classe descende de `TPersistent` pour gérer la sérialisation. Toutefois, comme nous l'avons présenté dans la partie traitant du langage, il ne faut pas confondre la sérialisation à la mode VCL, même sous .NET, et celle disponible par le framework et utilisable sous Delphi .NET quel que soit le mode d'interface choisi… Si nous avions opté pour une sérialisation par le framework dans la version VCL, nous n'aurions pas pu reprendre le code Delphi 7 tel quel. Le code aurait alors ressemblé à celui de l'exemple courant pour Windows Forms.

Seconde constatation, l'objet est marqué sérialisable par l'utilisation de l'attribut personnalisé `[Serializable]`. En effet, pour placer une instance dans le presse-papiers, il est nécessaire que la classe soit sérialisable, ce qui peut se comprendre aisément.

Dernière constatation, la classe possède un constructeur dont le rôle est de mettre à zéro les champs internes en leur assignant une chaîne vide. Cela n'est pas nécessaire dans l'absolu mais l'est dans ce cas précis car nous allons utiliser une autre facilité du framework, le *databinding*. Cette possibilité permet de relier tout contrôle à une source de données. Cette dernière peut être une table mais aussi n'importe quel objet, ce qui s'avère très pratique. Le code ci-dessous est le gestionnaire de l'événement `Load` de la fiche (équivalent à `OnCreate` de la classe `TForm` de la VCL) :

```
procedure TWinForm1.TWinForm1_Load(sender: System.Object; e: System.EventArgs);
begin
  // création de l'objet adresse
  Adr := TAdresse.Create;
  // binding des contrôles visuels aux propriétés de l'objet
  TextBox1.DataBindings.Add('Text',Adr,'Societe');
  TextBox2.DataBindings.Add('Text',Adr,'Contact');
  TextBox3.DataBindings.Add('Text',Adr,'Adresse1');
  TextBox4.DataBindings.Add('Text',Adr,'Adresse2');
  TextBox5.DataBindings.Add('Text',Adr,'CP');
  TextBox6.DataBindings.Add('Text',Adr,'Ville');
end;
```

Ce code appelle quelques commentaires, le premier étant que le champ `Adr` créé ici ne sera pas détruit dans un équivalent de `OnDestroy` pour Windows Forms. Cela est inutile puisque le ramasse-miettes du framework s'en chargera. Le second point à noter est la façon dont nous lions les contrôles de saisie (classe `TTextBox` équivalente à `TEdit` de la VCL) à la source de données objet. Pour ce faire, nous utilisons la méthode `Add` du `DataBindings` de chaque contrôle. Cette méthode réclame trois paramètres. Le premier est le nom de la propriété de l'objet en cours qui est liée à la source de données. Dans notre cas, c'est la propriété `Text` du contrôle qui est utilisée. Le second paramètre indique l'adresse de l'objet source de données, ici `Adr` (objet appartenant à la fiche). Enfin, le troisième paramètre est le nom de la propriété dans la source de données, c'est lui qui change sur les six lignes d'affectation du code ci-dessus.

Databinding

Il faut noter que nous détournons un peu ici le databinding qui, pour être totalement efficace, impose que les objets sources de données implémentent certaines interfaces, ce qui n'est pas tout à fait le cas dans cet exemple. Nous verrons plus loin dans cet ouvrage des réalisations plus claires et totalement fonctionnelles de databinding, ne vous arrêtez pas à l'utilisation qui en est faite ici.

Une fois tout ceci en place, regardons le code de la fonction de copie des données :

```
procedure TWinForm1.btnCopier_Click(sender: System.Object; e: System.EventArgs);
begin
  System.Windows.Forms.Clipboard.SetDataObject(Adr,true);
end;
```

La copie dans le presse-papiers s'effectue en appelant la méthode `SetDataObject` de l'objet `Clipboard` exposé par le framework de gestion des fenêtres (espace de nommage `System.Windows.Forms`). Le premier paramètre de la méthode est l'adresse de l'objet à copier, le second indique si ce dernier doit être dupliqué ou non.

Le code du bouton Effacer est très simple puisqu'il consiste à vider les propriétés `Text` des contrôles visuels. Ceci n'a qu'un but, permettre de voir l'effet de la fonction Coller… Le code de cette dernière est le suivant :

```
procedure TWinForm1.btnColler_Click(sender: System.Object; e: System.EventArgs);
var ad:TAdresse;
begin
Ad :=
System.Windows.Forms.Clipboard.GetDataObject.GetData(
 'Winform1.TAdresse') as TAdresse;
if not assigned(Ad) then exit;
TextBox1.Text:=Ad.Societe;
TextBox2.Text:=Ad.Contact;
TextBox3.Text:=Ad.Adresse1;
TextBox4.Text:=Ad.Adresse2;
TextBox5.Text:=Ad.CP;
TextBox6.Text:=Ad.Ville;
Adr := ad;
end;
```

Pour récupérer les données objet du presse-papiers, nous commençons par créer une variable de type `TAdresse`, `Ad`. Nous lui assignons ensuite le résultat d'un appel à la fonction `GetData` de l'interface `IDataObject` retournée par `GetDataObject` du presse-papiers. Nous transtypons en même temps cette interface en un `TAdresse`. À partir de là, tout devient simple, il ne reste plus qu'à recopier les valeurs dans les contrôles et à assigner l'objet récupéré du presse-papiers à l'instance `Adr` que nous maintenons dans la fiche. Les techniques de *binding* permettent de récupérer ces informations plus automatiquement, mais notre exemple concerne le presse-papiers et non le *databinding* que nous verrons ultérieurement dans des contextes plus appropriés.

Les traitements longs

Dans une grande majorité des applications, il existe des traitements qui prennent un temps important : requêtes sur de grandes tables, interrogations d'une machine distante via le réseau ou Internet, calculs divers… Il est indispensable que l'utilisateur soit averti de la progression d'un tel traitement dès qu'il risque d'être long.

Certains développeurs pensent avoir résolu le problème en affichant un laconique « Traitement en cours ». Autant dire tout de suite que cela ne sert à rien du tout ! Un « traitement en cours » peut durer 5 secondes ou 5 heures. Dans ce dernier cas, l'utilisateur aura redémarré sa machine bien avant, la croyant bloquée, soyez-en convaincu…

Si votre application possède de tels traitements (dépassant quelques secondes ou pouvant, en charge réelle, les dépasser), créez une fiche simple proposant un message d'attente ainsi qu'une barre de progression et un éventuel bouton permettant de stopper l'action en cours.

Testez en conformité avec la réalité !

Une grande majorité, hélas, des développeurs se contente de tester leurs applications en cours de conception sur les bases d'un jeu de test alimenté au fur et à mesure que les fiches de saisie sont disponibles. Ces fichiers de test sont bien plus petits que les fichiers réels des futurs utilisateurs et leur contenu est bien souvent totalement farfelu et sans rapport avec la réalité (des dizaines de « toto » et autres « titi », des champs remplis de « aaaa » ou « zzzz », etc.). De telles données ne peuvent pas prétendre constituer un jeu de test sérieux. Pour développer vos applications, créez un jeu de test réaliste, à la fois en contenu et en taille. Vous vous apercevrez bien plus vite qu'un traitement est trop lourd s'il prend 10 minutes sur un 1000 fiches que s'il prend 1,8 secondes sur 3 fiches… De même, vous constaterez bien plus aisément qu'une requête retourne des données erronées si elle affiche « chaussure », tiré du fichier article, à la place de « durant » dans un listing des clients que si ce dernier liste des lignes de « aaaa », « zzzz » ou « toto »… Les traitements longs ainsi que les grosses erreurs se détectent mieux avec des jeux de test cohérents et réalistes, tenez-en compte pour vos prochains projets…

Souvent, les développeurs, toujours tentés par un peu de dirigisme à l'égard des utilisateurs, pensent qu'il n'est pas nécessaire de proposer un bouton d'arrêt car « cela n'a pas de sens » dans le contexte. C'est un point de vue de technicien peu prévoyant… En effet, dans la réalité, si l'utilisateur décide qu'il a trop attendu, il tuera la tâche ou redémarrera la machine ! Autant prévoir le cas afin de le gérer le plus proprement possible… simple sagesse liée à l'expérience !

Dans le même esprit, lorsque vous attendez dans une salle d'attente, vous savez bien que le temps passe de façon plus agréable si l'on vous annonce à l'avance la durée approximative de votre attente, même si celle-ci doit être longue. Pensez-y aussi et, dans le cas de traitements qui peuvent avoir des durées très variables, essayez d'afficher une estimation du temps restant. Cela est nettement plus agréable pour celui qui patiente.

Pour la création de la fiche, utilisez la technique de la procédure appelante (approche proposée dans un autre chapitre). Nous vous conseillons d'appeler la procédure d'ouverture de cette fiche `OpenWait(Const msg :String)`, `msg` étant le message qui est vu par l'utilisateur. Dans l'idéal, une telle fonction devrait faire partie de votre unité `Common`, unité présente dans tout projet qui contient les services courants utilisables potentiellement par toutes les unités de votre application.

La procédure d'attente aura pour tâche de créer une instance de la fiche d'attente, d'affecter le message et mettre la barre de progression à zéro puis d'appeler la fiche en mode `Show` non modal (sinon le traitement long en question serait bloqué jusqu'au retour de la fiche, ce qui ferait perdre beaucoup de son charme à cet affichage...).

La fiche d'attente doit être de type `TForm.FormStyle := fsStayOnTop`, ce qui signifie que nous désirons qu'elle soit affichée au-dessus de toutes les autres quoi qu'il arrive.

Créez une procédure `CloseWait` dont la tâche sera de détruire la fiche d'attente.

Pour mettre à jour la barre de progression, créez enfin une procédure du type :

```
SendWait(Const NewMsg :STRING ; GaugeValue : Integer)
```

Si le paramètre `NewMsg` est vide, le message courant est conservé, sinon, il est modifié pour prendre la valeur de `NewMsg`. Le paramètre `GaugeValue` sera une valeur entre 0 et 100 représentant le pourcentage d'avancement du traitement.

Cette procédure, après affectation des différents paramètres, se terminera par un appel à `Application.ProcessMessages` afin d'assurer le rafraîchissement de l'image. D'ailleurs, au sein de votre traitement long, placez au moins un appel à cette procédure afin d'assurer la lecture fréquente (mais pas trop !) de la pile des messages de Windows.

Cela était indispensable sous Windows 3.1 pour simuler le multitâche coopératif (justement). Certains développeurs pensent que le multitâche préemptif des environnements 32 bits rend cet appel caduc. Ils se trompent. Si la commutation de tâches n'est plus assurée par la coopération des tâches entre elles, la lecture de la pile des messages Windows par l'application reste conditionnée à l'appel de `ProcessMessages` qui reprend alors tout son sens premier (traiter les messages).

Certains traitements réellement longs doivent toujours pouvoir être interrompus par l'utilisateur comme nous le disions plus haut. Il est indispensable de prévoir cette possibilité. Cela se fait en ajoutant un bouton Annuler dans la fiche d'attente. L'appui sur ce bouton positionne une propriété de la fiche d'attente `Annulation_demandee := True`, propriété initialisée à `False` par la procédure `OpenWait`.

Le traitement long, en plus de faire des appels réguliers à `SendWait`, teste après ces derniers la valeur de la propriété `Annulation_demandee` qui est lisible à travers une fonction `AnnulationDemandee : Boolean` qui, toujours dans l'esprit d'une isolation totale entre les fiches, est la seule à savoir lire la véritable propriété de la fiche d'attente.

Si aucun appel à `ProcessMessages` n'est effectué dans la boucle du traitement long, cette variable ne sera jamais positionnée correctement.

Charge à votre application d'annuler le traitement en cours proprement en rétablissement la situation précédente ou en interdisant certaines opérations tant que le dit traitement n'a pas été relancé.

Si vous devez lancer un traitement réellement long, en plus de la barre de progression nous vous conseillons d'afficher une estimation du temps restant, à la manière des copies de fichiers de Windows (mais en moins farfelu !). Ce temps s'analyse simplement en fonction du pourcentage d'avance et du nombre de pas réellement réalisés et de ceux restants. L'affichage devra être effectué sous la forme Heure, Minute, Seconde.

À titre d'exemple voici le code source d'une mise eu œuvre possible de ce principe. Vous le retrouverez ainsi que son projet de test complet sur le CD-Rom du livre.

```
{-------------------------------------------------------------------
 Unit Name: uatt2
 Author:    od
 Purpose:   Système d'attente exemple pour livre Delphi .NET
-------------------------------------------------------------------}
unit uatt2;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls,
  ExtCtrls, DateUtils, System.ComponentModel;

type
  TFAttente = class(TForm)
    lmsg: TLabel;              // le message à afficher
    pbAttente: TProgressBar;    // la jauge de progression
    lEstimation: TLabel;        // contient l'estimation de temps restant
    timEstimation: TTimer;      // le timer utilise pour l'estimation
    procedure timEstimationTimer(Sender: TObject); // événement du timer
  private
    { Déclarations privées }
    FDepart : TDateTime; // heure de départ
  end;

  // Ouverture d'un message d'attente
  // Les appels peuvent être superposés
  // Si WithGauge est Vrai, une jauge est affichée
  Procedure OpenWait(Const msg:string;WithGauge:boolean=false);

  // Fermeture du dernier message
  // Si CloseAll est Vrai, tous les messages empilés
  // sont fermés
  Procedure CloseWait(CloseAll:boolean=False);

  // permet la mise à jour du message d'attente
  // ainsi que celle de la jauge
  Procedure SendWait(NewMsg : string;GaugeValue : integer = 0);

implementation
{$R *.dfm}
var
 MemoMsg : TStrings;  // Mémoire des messages "empilables"
 Fiche   : TFAttente; // la fiche d'attente

Procedure OpenWait(Const msg:string;WithGauge:boolean=false);
begin
 if assigned(Fiche) then
```

```
   // empilage du message et de la valeur de jauge
   MemoMsg.AddObject(
     Fiche.lmsg.Caption,tobject(Fiche.pbAttente.position)) else
   // sinon création de la fiche
   Fiche := TFAttente.Create(Application);
  Fiche.lmsg.Caption := msg;              // le message d'attente
  Fiche.pbAttente.visible := WithGauge;   // afficher ou non la jauge
  Fiche.pbAttente.position := 0;          // raz de la jauge
  Fiche.FDepart := now;                   // heure de départ
  Fiche.lEstimation.visible := WithGauge; // Estimation seulement si jauge présente
  Fiche.timEstimation.Enabled:=WithGauge; // Enclenche le timer si jauge visible
  Fiche.Show;                              // afficher la fiche
  Fiche.Update;                            // mettre à jour l'affichage
end;

Procedure CloseWait(CloseAll:boolean=False);
begin
 if not assigned(Fiche) then exit;
 if closeAll then MemoMsg.Clear;
 if MemoMsg.Count>0 then // s'il existe des messages empilés
  begin
   // récupération du message au sommet de la pile
   Fiche.lmsg.Caption := MemoMsg[pred(MemoMsg.Count)];
   // récupération de l'état de la jauge
   Fiche.pbAttente.Visible :=
     assigned(MemoMsg.Objects[pred(MemoMsg.Count)]);
   // récupération de la position de la jauge
   Fiche.pbAttente.Position :=
    integer(MemoMsg.Objects[pred(MemoMsg.Count)]);
   Fiche.Update; // mise à jour de la fiche
   MemoMsg.Delete(pred(MemoMsg.Count)); // suppression du message empilé
  end else
  begin
   Fiche.Release;
   Fiche := nil;
  end;
end;

Procedure SendWait(NewMsg : string;GaugeValue : integer);
begin
 if not assigned(Fiche) then exit;
 if NewMsg<>'' then
  begin
   Fiche.lmsg.Caption:=NewMsg;
   Fiche.lmsg.Update;
  end;
 if GaugeValue in [0..100] then
  begin
   Fiche.pbAttente.Position := GaugeValue;
   Application.ProcessMessages;
  end;
```

```
end;

procedure TFAttente.timEstimationTimer(Sender: TObject);
var elapsed,estimation : double;
begin
 elapsed := MilliSecondSpan(Now,FDepart); // ms écoulées
 if pbAttente.Position=0 then estimation := 0 else
   estimation :=
   (elapsed*(100.0-pbAttente.Position))/pbAttente.Position;
 lEstimation.visible := estimation>0;
 lEstimation.Caption :=
   'Temps restant: '+TimeToStr((estimation+1000)/msecsperday);
 lEstimation.Update;
end;

Initialization
MemoMsg := TStringlist.Create;
Fiche   := Nil;
Finalization
CloseWait(true);
FreeAndNil(MemoMsg);
end.
```

Le code ci-dessus est fonctionnel mais ne prend pas en compte la possibilité d'annulation. À vous de l'améliorer !

Vous noterez que le but est bien d'illustrer un principe d'ergonomie et non de proposer une solution technique parfaitement adaptée à une situation réelle. Sous .NET, nous pourrions notamment utiliser des techniques plus Objet pour la mise en œuvre. Dans certains cas, une implémentation utilisant des threads peut être mieux appropriée. Tout dépend de l'application, de son code et de la conception de ce dernier.

Le blocage de l'application

Lors des traitements longs ou potentiellement longs, il faut s'assurer que l'utilisateur ne peut pas cliquer et déclencher d'autres fonctions de votre application.

Normalement, cela est interdit par le fait que toutes les actions sont réalisées par le thread principal (ce qui est bloquant). Mais si vous ajoutez des `ProcessMessages` en mode VCL ou que vous prévoyez des possibilités d'annulation du traitement en cours, l'utilisateur pourra cliquer sur une entrée de menu, un bouton…

Attention, Danger !

Vos procédures ne sont pas forcément réentrantes et cela finira certainement par « planter » l'application…

Si votre application lance des traitements de ce type (mise à jour de lignes d'une table par exemple) qui sont longs mais malgré tout balisés par des pauses de traitement des messages (`ProcessMessages`), prévoyez absolument un système de sémaphores.

La mise en œuvre dépend du contexte comme toujours, mais les techniques les plus utilisées sont, dans l'ordre croissant de sophistication :

- une variable booléenne globale, locale à la fiche ou variable de classe ;
- une propriété de la fiche avec ses accesseurs ;
- l'utilisation des sémaphores ou équivalents proposés par Windows ;
- la définition d'un automate, etc.

Une fois le système choisi, vous devez débuter tous vos gestionnaires d'événements par un test d'occupation de l'application. Si une fonction bloquante est en cours, un simple `Exit` permettra d'éviter l'exécution de l'action demandée.

Si vous utilisez le système des actions de la VCL, utilisez l'événement `OnUpdate` de `TActionManager`, voire de chaque action, pour simplement désactiver les actions lorsque la variable d'occupation de l'application est à `True`. On voit d'ailleurs ici l'intérêt d'utiliser ce système d'actions par la centralisation et la simplification du code qu'il offre. Sous Windows Forms, d'autres techniques sont utilisables, mais comme dans tout ce chapitre, l'implémentation est secondaire à notre propos. Ce qui compte réside dans le principe ergonomique présenté, il correspond à une situation réelle pour l'utilisateur. Et lorsqu'un utilisateur peste devant une application qui plante, il se moque bien de savoir ce que vous avez utilisé comme outils de développement !

Les bulles d'aide

La norme est assez floue à propos des bulles qui ne font pas partie du fonctionnement de base historique de Windows. Toutefois, on peut affirmer que les textes doivent être très courts : une bulle d'aide ne doit pas être un pavé d'aide ! Le mot bulle, lui-même, implique une certaine légèreté, ne l'oubliez pas...

On préférera des bulles ne contenant qu'un seul mot ou expression avec le minimum d'articles et de mots de liaison : Quitter, Mise à jour, et non « Quitter l'application » ou « Lancer la mise à jour ».

Si la traduction française donnée à *hint* en informatique (bulle) insiste sur la légèreté, le sens original anglais de hint insiste lui sur le caractère fugace de l'information donnée et sur sa fonction première qui est de donner une indication et non un cours complet.

Une définition

HINT : (1) Allusion ; To drop a hint = faire une allusion (à noter « drop », action rapide en général), (2) Conseil (piece of advise : morceau de conseil), (3) Soupçon...

Une bulle, un soupçon, un morceau de conseil... voici presque poétiquement bien résumé ce que doit être une bulle d'aide !

Proscrivez l'aspect amateur et paradoxal des bulles multilignes sauf dans des cas bien précis (peu nombreux). Toutefois, court et précis ne signifie pas devoir utiliser le style télégraphique ou une sibylline notation hiéroglyphique...

Le choix d'un texte pour une bulle est une démarche de même nature que celle du choix et du dessin d'une icône, d'une musique d'accompagnement, d'une séquence vidéo. Le caractère multimédia de Windows impose des compétences artistiques à tout concepteur. Ainsi, choisir un mot ou une expression qui résumera bien une fonction est une tâche parfois difficile réclamant des qualités qui ne sont pas forcément celles d'un informaticien. L'humilité, en partie, consiste à connaître ses limites. Si vous ne trouvez pas, faites dans la simplicité et l'évidence. S'il faut être un informaticien ou un nouveau Champollion pour décrypter vos bulles, mieux vaut ne pas en mettre du tout...

Dans certains cas rares, il se peut que vous n'arriviez pas à trouver pas une formulation courte et sensée. Ce n'est pas trop grave, mais veillez à ce que la bulle ne soit pas aussi large que l'écran. Pour y remédier, on peut alors admettre (et uniquement dans ce cas) de placer le texte sur deux lignes, pour « ramasser » l'allure générale de la bulle. L'éditeur de Delphi ne permet pas de placer des changements de ligne dans le texte d'une bulle. Ce dernier est considéré comme une propriété `String`, le retour chariot dans l'inspecteur d'objet met simplement fin à la saisie en la validant.

Cela ne doit pas vous décourager, la fenêtre affichant les bulles prévoit l'affichage sur plusieurs lignes. Pour faire la jonction entre l'éditeur de propriétés qui n'autorise pas la saisie du caractère #13 et la classe de fenêtre des bulles qui, elle, le gère, vous devez saisir le texte par programmation :

```
MonObject.Hint := '1ère ligne'#13#10 'seconde ligne' ;
```

Vous noterez l'insertion du couple `CR+LF` (`#13#10`) pour le changement de ligne.

Cette méthode vous pousse à ne pas abuser de la chose. Sachez que vous pouvez aussi utiliser l'éditeur de propriétés `Hint` spécialisé acceptant la saisie des changements de ligne.

Une fois le texte des bulles saisi, il est important de prévoir, par un paramètre persistant de l'application, la possibilité de déconnecter l'affichage des bulles d'aide. Si ces dernières sont appréciées lors de la prise en main d'une nouvelle application, il faut avouer que leur affichage intempestif peut finir par agacer lorsqu'on connaît bien le logiciel.

Chaque objet `TForm` dispose d'une propriété `ShowHint` permettant de contrôler l'affichage des bulles d'aide et tous les objets possédant une propriété `Hint` ont aussi une propriété `ParentShowHint` qu'on laisse à `True` (sauf cas particuliers). La visibilité globale pour la fiche est contrôlée par `TForm.ShowHint`.

Pour gérer de façon habile cette option sans intervenir sur chaque fiche, positionnez systématiquement la propriété `ShowHint` des fiches à `True` en conception. Vous contrôlez ensuite le comportement de l'application en modifiant la propriété `ShowHint` de l'objet `Application`.

Remarque

L'objet Application est créé automatiquement par Delphi depuis la classe `TApplication` et ce dans l'unité `Forms`. Cet objet donne accès à de nombreux paramètres et événements, prenez-en connaissance en lisant l'aide de cette classe. Il faut noter que Delphi 6 a introduit un composant `TApplicationEvents` qui simplifie la programmation de gestionnaires d'événement pour l'objet Application. Ce composant a été porté dans la VCL .NET.

Dans le `OnCreate` et le `OnDestroy` de votre fiche principale, programmez respectivement la lecture et la sauvegarde de l'état de cette propriété et le tour est joué… Il ne reste plus qu'à ajouter dans les options de votre application une case à cocher ou une entrée de menu de type bouton radio pour laisser l'utilisateur contrôler l'affichage des bulles d'aide.

Windows Forms

Le principe d'activation reste le même mais les voies sont différentes. Chaque contrôle Windows Forms possède une propriété `ToolTip` qui joue un rôle équivalent à la propriété `Hint` des contrôles VCL. Néanmoins, l'astuce du symbole *pipe* permettant de placer un second texte affiché par l'application n'existe pas. En réalité, `ToolTip` est une fausse propriété, elle déclenche dans le code généré par le concepteur l'ajout d'une ligne du type :

```
Self.ToolTip1.SetToolTip(Self.Button1, 'Bulle du bouton 1');
```

La propriété `ToolTip` n'est qu'une ruse du concepteur visuel de Delphi, elle n'existe pas dans le framework…

Comme le montre ce petit morceau de code, pour que l'affichage des bulles d'aide soit activé, il est nécessaire de poser un composant `ToolTip` sur la fiche et d'en régler les paramètres (ceux fournis par défaut sont très standard et peuvent convenir sans intervention).

Vous remarquerez assez vite que le framework n'est pas toujours aussi souple et maniable que ne l'est la VCL ! Par exemple, il n'est pas possible de changer la couleur de la bulle d'aide. Si vous cherchez bien sur Internet, nombreux sont les développeurs C# à poser ce genre de questions sur les forums. Hélas, pas de réponse simple. Le framework Windows Forms est très séduisant par de nombreux aspects, mais il est préférable de se contenter des classes fournies, vouloir juste comme ici changer une couleur peut impliquer des heures de développement. Une raison de plus d'utiliser la VCL .NET.

Amélioration des bulles d'aide

L'objet `Application` (classe `TApplication`) que nous évoquions précédemment dispose de nombreuses autres propriétés pour contrôler les bulles d'aide. Servez-vous en pour personnaliser le *look & feel* de votre application. Par exemple, `HintColor`, `HintHidePause`, `HintPause`, `HintShortCuts`, `HintShortPause` sont autant de paramètres à votre disposition.

La décision de fournir ou non à l'utilisateur un moyen de personnaliser ces options est laissée à votre discrétion. Que l'utilisateur puisse commander l'affichage des bulles est une chose, que la page des options du logiciel ressemble à un tableau de bord de 747 en est une autre (trop d'ergonomie tue l'ergonomie)…

N'oubliez pas non plus que vous pouvez pousser plus loin encore la personnalisation des bulles d'aide puisque vous pouvez écrire votre propre classe de fenêtre utilisée pour les bulles.

On trouve dans le commerce des bibliothèques en shareware (voire en freeware) proposant des bulles en forme de nuage ou de phylactère de bande dessinée. Mais restez toujours sobre… Dans certains cas, cela peut ajouter un attrait à vos applications (une bulle nuage

dans une application destinée aux enfants fait moins austère, afficher des images dans la bulle en fonction du texte sera aussi un plus…).

Le code suivant démontre qu'il peut être simple de personnaliser la fenêtre d'affichage des bulles :

```
unit Unit1;


interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, System.ComponentModel;

type
  TForm1 = class(TForm)       // fiche de test avec ShowHint à True
    Edit1: TEdit;             // un TEdit avec une bulle d'aide
    procedure FormCreate(Sender: TObject); // gestionnaire du OnCreate
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

Type  // création d'une version personnalisée de la fenêtre des Hints
  TMyHintWindow = Class (THintWindow)// THintWindow est fourni par Delphi
    Constructor Create (AOwner: TComponent); override;
    end;

// constructeur de la nouvelle classe
Constructor TMyHintWindow.Create (AOwner: TComponent);
Begin
  Inherited Create (Aowner);
  // personnalisation : changer la fonte, à vous de faire mieux !
  Canvas.Font.Name := 'Times New Roman';
  Canvas.Font.Size := 14;
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // initialisation de la fiche
  Application.ShowHint := False;// nécessaire avant le changement
  HintWindowClass := TMyHintWindow; // votre classe de fenêtre de Hint
  Application.ShowHint := True;// remise en route des hints
end;
end.
```

Conclusion

Comme nous le disions en introduction de ce chapitre, l'ergonomie est un vaste sujet et une attention de chaque instant lors du développement. S'il reste impossible de faire le tour de la question en quelques pages, l'auteur espère avoir attiré votre regard sur ces petits détails qui font un bon logiciel agréable à utiliser. Nous aurions pu aborder les lignes d'état et leur rôle essentiel, les dialogues de tout type ainsi que de nombreux autres sujets. Mais notre seul but était de vous sensibiliser aux problèmes d'ergonomie. Nul doute que désormais vous découvrirez par vous-mêmes de nombreux autres aspects de cette partie importante de la conception d'une application et que vous aurez à cœur de noter et de systématiser vos pratiques pour que vos interfaces gagnent en cohérence et en simplicité.