# Buildoo

M. Superina[1]

*1: Université de Nice Sophia-Antipolis*
*Ecole Supérieure des Sciences Informatiques*
*930, Route des Colles*
*06903 Sophia-Antipolis, France*
`msuperin@masterweb.it`

**Résumé**

Buildoo est un constructeur d'interfaces graphiques pour Bigloo, une version étendue du langage Scheme, et sa librairie graphique Biglook. Buildoo est intégré dans l'environnement de programmation Bee.

Dans cet article nous présentons ses principales fonctionnalités, son architecture et le code Scheme qu'il produit à partir de la description d'une interface. Nous montrons aussi que s'appuyer sur un langage de haut niveau a facilité l'implantation de Buildoo. En particulier, nous mettrons l'accent sur l'importance de l'utilisation du mécanisme d'introspection proposé par Bigloo.

Même si Buildoo n'est encore qu'un prototype, nous sommes parvenu à l'auto-générer. C'est-à-dire que l'interface graphique utilisée dans Buildoo est construite avec Buildoo lui-même.

## Introduction

Buildoo est un constructeur d'interfaces graphiques pour le langage Bigloo. Il exploite la librairie graphique Biglook. Il a été entièrement écrit en Bigloo, il est donc utilisable sur toutes les plate-formes pouvant accueillir Bigloo [8]. Buildoo a été conçu pour s'intégrer dans l'environnement de programmation déjà existant de Bigloo.

Un constructeur d'interfaces graphiques est un outil qui permet de dessiner à la souris une interface et d'en spécifier le comportement. Il génère ensuite le code source correspondant. Le rôle du constructeur d'interfaces est de faciliter la tâche du programmeur. En effet il est possible d'écrire à la main le code source d'une interface graphique, mais c'est une tâche fastidieuse et peu gratifiante. Par ailleurs, l'élaboration d'une interface nécessite des réglages graphiques fins qui sont difficiles à programmer "a priori". Ces réglages (choix des couleurs, placement des objets) sont beaucoup plus faciles à réaliser si on dispose d'un constructeur qui permet de "voir" instantanément les modifications appliquées. Autrement, pour voir le résultat des modifications, il faut re-compiler à chaque fois et relancer une exécution.

L'objectif de Buildoo est de fournir un outil de programmation complet, efficace et d'utilisation intuitive. Parmi les constructeurs d'interfaces graphiques existant dans d'autres environnements de programmation (Visual C++, Visual Caffe, Glade, Allegro, ...), certains, pour exploiter pleinement la puissance d'un langage ou d'une librairie graphique, sont trop compliqués à utiliser. Souvent ils proposent de nombreuses fonctionnalités et un grand nombre de réglages. Il en résulte que la période d'apprentissage de ces outils est assez longue.

Souvent, le code généré par le constructeur d'interfaces ne s'intègre pas facilement avec du code déjà existant. Il est souvent indispensable d'éditer le code généré par le constructeur d'interfaces pour insérer les fonctions de calcul associées aux objets graphiques. Ceci présente un inconvénient majeur: le code généré par le constructeur d'interfaces est exhibé à l'utilisateur. Lorsque ce dernier édite ce code pour y ajouter ses spécifications, il doit impérativement prendre soin de ne pas modifier les parties maintenues par le constructeur sous peine de ne plus pouvoir éditer l'interface graphiquement. De plus nous pensons que séparer la partie interface de la partie calcul permet une programmation plus souple (avec ce schéma, il est, par exemple, plus aisé d'ajouter une interface à un programme déjà existant).

Pour maintenir des temps de compilation aussi petits que possible, la génération du code de Buildoo a été particulièrement soignée. Le code produit est de taille réduite et les parties graphiques et de calcul ne sont jamais mélangées. L'objectif est qu'un utilisateur de Buildoo n'ait jamais besoin de modifier du code généré. La connexion entre le code généré de l'interface et le code utilisateur décrivant les comportements des objets graphiques est assurée par l'environnement de programmation de Bigloo (Bee).

Buildoo a été conçu de façon à pouvoir s'adapter facilement aux évolutions de la librairie graphique. Pour cela nous avons utilisé le mécanisme d'introspection proposé par Bigloo.

Dans la section 1 nous présenterons brièvement Bigloo et Biglook. Dans la section 2 nous exposerons quelques unes des fonctionnalités de Buildoo qui le distinguent des autres constructeurs d'interfaces. Dans la section 3 nous présenterons les mécanismes internes de Buildoo. Dans la section 4 nous montrerons la chaîne de compilation et les différents formats des fichiers utilisés par Buildoo. Dans la section 5 nous présenterons le code Scheme et dans la section 6 comment Buildoo peut être auto généré. Enfin dans la section 7 nous comparerons Buildoo à d'autres constructeurs d'interfaces.

## 1. Plate-forme de développement

Buildoo est un générateur d'interfaces graphiques dédié au système Bigloo. Dans cette section, nous donnons une présentation sommaire de ce système.

### 1.1. Bigloo

Bigloo est une implantation d'une version étendue du langage de programmation Scheme. Il respecte la norme $R^5RS$ [7]. Parmi ses principales extensions, on peut noter la connexion avec C[1], un langage de modules et une couche objet. Les modules de Bigloo permettent la compilation séparée. La couche objet est largement inspirée de CLOS [5]. Les méthodes sont placées dans des fonctions génériques et il existe un mécanisme d'introspection. À la différence de CLOS, Bigloo ne propose que de l'héritage simple.

Les champs des instances sont accédés au moyen de fonctions générées par le compilateur ; une pour la lecture et une pour l'écriture. Bigloo ne dissocie pas les relations de sous typage de celles d'héritage. Bigloo ne propose aucune construction permettant de restreindre l'héritage. C'est-à-dire que toute classe $\mathcal{C}$ dérivant d'une autre classe $\mathcal{C}_{\text{super}}$ (sa “super” classe) en hérite tous les champs. Ainsi une instance de la classe $\mathcal{C}$ possède tous les champs de $\mathcal{C}$ *et* ceux de $\mathcal{C}_{\text{super}}$. Par exemple, si on considère une classe `simple-widget` qui possède les champs `background` et `border-width` et une sous classe `label` définissant les champs `anchor` et `foreground`, alors les instances de cette classe disposeront des champs `anchor`, `foreground`, `background` et `border-width`.

Tout programme Bigloo est composé d'un ensemble de modules. Chaque module est constitué d'une déclaration de module suivie du corps du module. Un module peut être éclaté sur plusieurs fichiers si nécessaire; la partie déclarative du module doit obligatoirement être la première expression du premier fichier contenant le module. La compilation de chaque module produit un fichier objet. L'édition de liens rassemble les fichiers objets pour produire un exécutable.

1. Bigloo peut appeler du code C et vice versa

### 1.2. Biglook

Biglook[6] est une librairie graphique pour Bigloo. Cette librairie est entièrement objet. Tous les éléments de la librairie dérivent de la classe `Biglook-Object` (Figure 1). Chaque élément graphique (que nous nommerons *widget* dans la suite de ce texte) pouvant être inséré dans une interface, dérive de la classe `widget`.

Dans sa version actuelle, Biglook est implanté au dessus de la librairie Tk. Ce choix d'implantation n'est pas apparent aux utilisateurs car la programmation avec Biglook ne se fait qu'au travers du modèle objet. Les utilisateurs de Biglook peuvent définir de nouvelles *widgets* réutilisant celles définies dans la librairie. Cela permet de construire ainsi des objets graphiques complexes composés d'objets élémentaires.

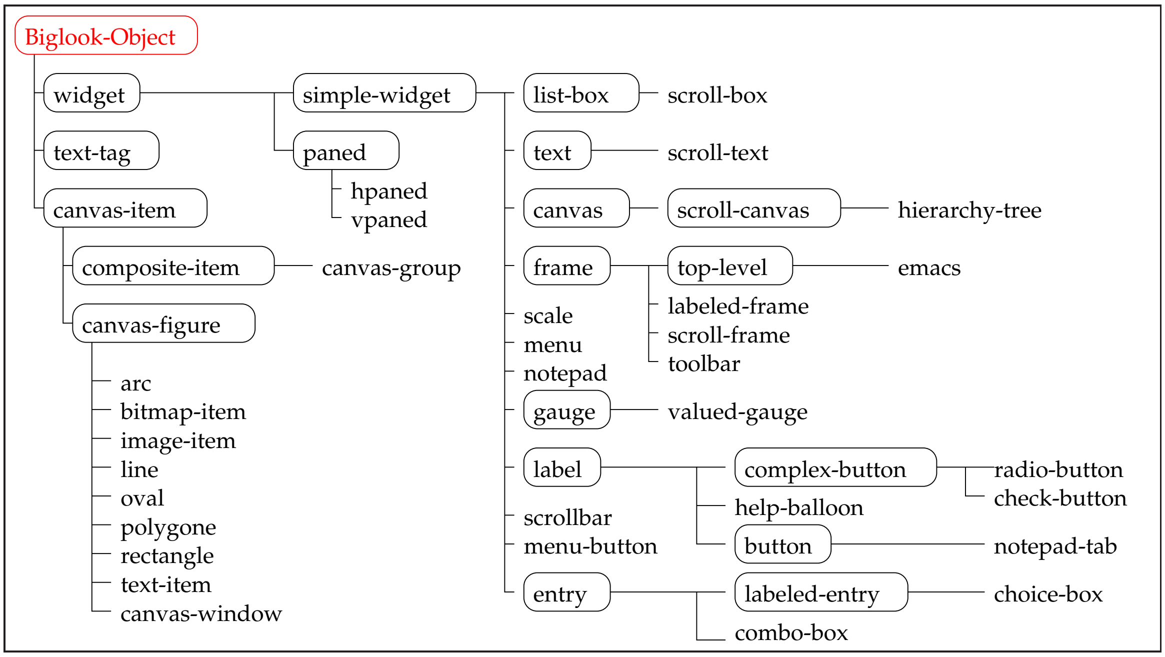


FIG. 1 –: *Les éléments de la librairie Biglook.*

## 2. Fonctionnalités

Dans cette section, nous allons montrer comment Buildoo s'intègre dans l'environnement de programmation de Bigloo. Pour cela nous commencerons par présenter une "session standard" d'utilisation de Buildoo. Ensuite nous étudierons deux exemples de fonctionnalités proposées par Buildoo.

### 2.1. Intégration dans l'environnement de programmation.

Buildoo est un nouveau composant de l'environnement de programmation de Bigloo nommé Bee [8]. Il s'agit d'un environnement de programmation intégré qui propose des fonctionnalités standard (documentation en ligne, gestion de projets, contrôle de versions, arpenteurs de code source, outil de mise au point et d'évaluation de performances).

Intégrer Buildoo dans l'environnement Bee facilite son utilisation. Créer ou modifier une interface existante se fait de façon entièrement graphique. Le gestionnaire de projet de Bee, est capable de gérer des fichiers textes et des fichiers représentant des interfaces. Ainsi la chaîne de compilation pour les projets contenant des interfaces est entièrement gérée par l'environnement de programmation. Cela dispense l'utilisateur d'acquérir une connaissance sur les divers formats de fichiers produits et utilisés par Buildoo. L'utilisateur "dessine" ses interfaces à la souris sous Buildoo. Il implante les codes sources définissant le comportement des objets graphiques depuis l'éditeur de textes de Bee.


FIG. 2 –: *Compteur réalisé avec Buildoo*

## 2.2. Une session Buildoo

Imaginons que l'on souhaite réaliser une application affichant un compteur et deux boutons permettant d'augmenter ou de diminuer sa valeur. La Figure 2 présente une copie d'écran de cette application.

Pour réaliser une telle application en utilisant Bee, un utilisateur doit définir le module principal (ici nommé `counter`) et l'enregistrer. Une implantation possible de ce module est:

```
(module counter
   (library biglook)
   (import gui-counter)
   (export (decremente)
           (incremente))
   (main main))

(define count-value 0)

(define (decremente) (set! count-value (- count-value 1)))
(define (incremente) (set! count-value (+ count-value 1)))

(define (main argv)
   (set! count-value (string->integer (cadr argv))))
```

Pour construire la partie graphique de cette application, il suffit maintenant de "lancer" Buildoo (cela se fait directement depuis Bee par un simple clic souris). Le panneau principal de configuration de Buildoo apparaît alors (Figure 3).


FIG. 3 –: *Fenêtre principale de Buildoo*

Les différents objets composant l'interface sont placés et configurés graphiquement. Spécifier les comportements des deux boutons de notre interface se fait en indiquant les noms des fonctions Scheme à exécuter lors des clics souris (dans notre exemple il suffit de préciser que l'action associée au bouton gauche est implantée par la fonction `decrement`). La figure 4 présente l'interface en cours de réalisation. Pour connecter notre interface ainsi réalisée au projet en cours, il faut donner un nom à l'interface (ici `gui-counter`) puis spécifier que l'interface utilise des définitions du module `count`. Compiler l'application entière est alors pris en charge par Bee.

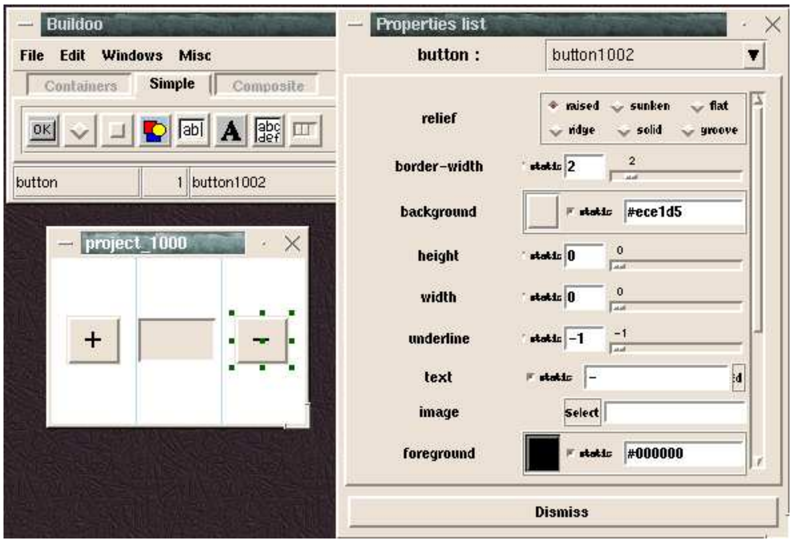


FIG. 4 –: *Une interface en cours de réalisation*

## 2.3. La sélection des widgets

Concevoir une interface simple et conviviale pour Buildoo a été une préoccupation majeure tout au long de son développement. En plusieurs circonstances nous avons mené des expérimentations pour concevoir la "meilleure" interface. Il n'est pas possible, dans cet article, de faire une présentation exhaustive de tous les choix effectués. Aussi nous ne présentons ici qu'un des problèmes et sa solution: la sélection multiple de widgets. La solution que nous présentons est révélatrice du travail que nous avons dû mener à plusieurs reprises.

Dans les constructeurs d'interfaces existants il est souvent difficile de sélectionner plusieurs objets de façon simple. La sélection d'une *seule* widget s'effectue directement à la souris ou par un panneau de propriétés. Pour la sélection multiple, c'est-à-dire de plusieurs widgets, le mécanisme se complique. Dans Buildoo, la sélection se fait en décrivant une zone à la souris. Des objets individuels peuvent être ajoutés ou supprimés de la sélection (avec l'association de la touche Ctrl et de la souris). Les sélections peuvent être utilisées pour factoriser les opérations à appliquer à plusieurs objets (par exemple, on peut supprimer tous les objets d'une sélection ou modifier les couleurs de fond de tous ces objets).

La difficulté vient de ce que les widgets d'une sélection ne sont pas obligatoirement des instances de la même classe et ne possèdent donc pas forcément les mêmes champs. Il devient alors impossible d'utiliser la sélection pour modifier les valeurs des champs qui ne sont pas présents dans tous les objets. Buildoo introduit la notion de classe "filtre". Les classes filtres sont utilisées pour sélectionner les objets pour lesquels s'applique une configuration d'une sélection.

Prenons l'exemple d'une interface contenant 3 `labels` et 4 `buttons` (Figure 5). Les 7 widgets sont sélectionnées et la classe filtre est fixée à `button`. Lors d'un réglage des couleurs des widgets, seules

les instances de la classe filtre sont modifiées (les 4 `buttons`). Pour changer les couleurs des `labels` il suffit de changer la classe filtre.

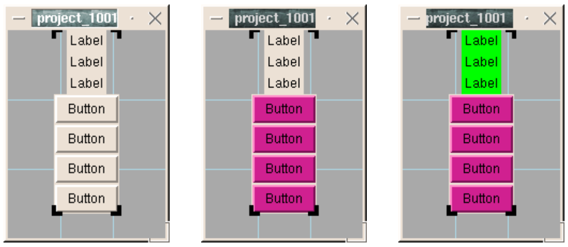


FIG. 5 –: *Fonctionnement de la multi-sélection.*

### 2.4. Fonctions utilisateur associées à l'interface (*callbacks*)

Les comportements des objets graphiques doivent être définis indépendamment de Buildoo. Par exemple, les comportements peuvent être définis dans des fonctions globales Scheme comme dans l'exemple du module `counter` de la Section 2.2. "Attacher" un comportement à un objet se fait en affectant un nom de fonction au champ `command` de l'objet. Par exemple, affecter un comportement au bouton gauche de notre interface, se fait en attribuant la fonction `increment` à ce bouton.

Certaines widgets ne possèdent pas de champ `command` mais la librairie Biglook permet toutefois d'associer un comportement à tous les objets. Pour cela, il suffit d'associer une fonction Scheme à une action spécifique (par exemple, Biglook permet de spécifier l'action à exécuter lorsque la souris pénètre dans une fenêtre alors que les instances de cette classe n'ont pas de champ `command`).

## 3. Implantation de Buildoo

Plutôt que de présenter l'architecture logicielle complète de Buildoo, nous présentons deux aspects techniques de son implantation. Nous montrons comment l'introspection de Bigloo est utilisée pour gérer automatiquement les panneaux de configuration des objets graphiques (une autre utilisation de l'introspection sera présentée dans la Section 4). Ensuite nous présenterons une difficulté due au mécanisme d'héritage de Bigloo.

### 3.1. Introspection

La principale particularité de Buildoo du point de vue de son fonctionnement est l'utilisation continue de l'introspection sur les widgets. En effet, lorsque l'on dessine une interface avec Buildoo, une instance de chaque objet composant l'interface est réellement créée. La plupart des opérations que Buildoo doit exécuter nécessitent des informations internes sur les widgets. Accéder à ces informations peut se faire par introspection sur les objets "réels" alloués par Buildoo. Par exemple, avant de détruire une widget, il faut d'abord détruire les widgets qui y sont contenues. Pour savoir quelles sont ces widgets Buildoo utilise l'introspection, qui est une méthode moins coûteuse que de garder trace des objets créés. La sauvegarde d'un projet fonctionne quasiment de la même manière, il s'agit de parcourir toute l'interface et, par introspection, retrouver toutes les widgets présentes.

Attardons nous sur le fonctionnement interne des panneaux de configuration des widgets. Cette partie de l'interface graphique est entièrement construite par introspection sur les widgets. C'est-à-dire que les panneaux de configuration changent d'aspect en fonction de la widget sélectionnée. Ceci est automatique, puisque Buildoo peut avoir directement accès aux noms et aux valeurs de tous les champs des widgets. Cette propriété est très importante, puisque l'on peut imaginer que la librairie Biglook pourra évoluer. De cette façon le code de Buildoo concernant les panneaux de configuration ne devra pas être modifié. Si des champs de certaines widgets changent le panneau de configuration s'adaptera automatiquement. Si des nouvelles widgets sont introduites dans la librairie, il suffira de rajouter une ligne descriptive de la nouvelle widget dans un des fichiers source du constructeur et Buildoo (après compilation) prendra en compte la nouvelle widget.

### 3.2. Un code compact

Le code de Buildoo est très compact, il tient en moins de 10000 lignes (interface graphique comprise). Il serait impensable de réaliser un constructeur d'interfaces graphiques en si peu de lignes en C. Cette compaction tient en partie à l'introspection parce que cette dernière permet de ne pas expliciter dans le code source de Buildoo une grande partie des caractéristiques de la bibliothèque graphique.

## 4. Chaîne de compilation

Dans cette section nous présentons la chaîne de compilation utilisée par Buildoo. Nous exhibons les différentes étapes qui permettent de passer de la description graphique d'une interface à un fichier source Scheme.

Lors de la phase d'élaboration des interfaces graphiques Buildoo ne produit aucun code Scheme. Comme nous l'avons montré dans la Section 3, Buildoo crée des objets graphiques "réels". Buildoo sauvegarde alors la description d'une interface en enregistrant les caractéristiques de ses objets dans un fichier binaire nommé `bld`. Le code Scheme associé à cette interface ne sera produit que lors de la compilation du projet. La Figure 6 illustre la chaîne complète de compilation utilisée par Buildoo.

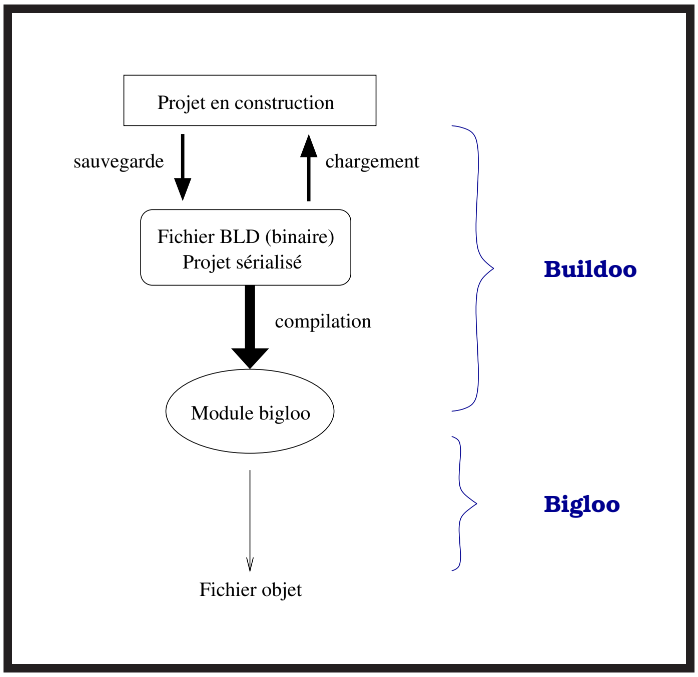


FIG. 6 –: *Chaine de compilation*

### 4.1. Le format `bld`

Lors d'une sauvegarde, Buildoo parcourt par introspection l'interface dessinée et récupère les caractéristiques de chaque widget. Avec ces informations il crée une structure où il ne sauvegarde que les valeurs qui ne sont pas les valeurs par défaut des objets (les définitions des classes peuvent fournir une valeur par défaut aux divers champs) de toutes les widgets de l'interface. Cette structure est ensuite sérialisée et écrite sur disque pour pouvoir, lors d'un chargement, récupérer une instance complète de l'interface. Dans le fichier `bld` sont écrits d'abord le nom du module et ensuite la structure sérialisée. Ce format est imposé par Bee. Il est impératif pour l'environnement que les fichiers `bld` contiennent, comme les fichiers sources standard de Bigloo un nom de module. Les fichiers `bld` ont en quelque sorte la même structure que les modules Bigloo (d'abord une partie déclarative et ensuite un corps). La figure 7 contient un extrait de la déclaration de la classe qui est utilisée pour sérialiser les objets d'une interface.

```
(class serialize-widget
   (id (default #unspecified))     ;; l'identificateur de l'objet graphique
   (class (default #unspecified))  ;; la classe de l'objet
   (parent (default #unspecified)) ;; l'identificateur du conteneur de l'objet
   (fields (default '()))          ;; les valeurs des champs modifiés
   (bindings (default '()))        ;; les fonctions associées aux évènements
   (slaves (default '()))          ;; les fils de cet objet
   (placement (default '()))       ;; positionnement dans l'interface
   (size (default 0))              ;; dimensions de l'objet
   ...)
```

FIG. 7 –: *Sérialiser les widgets*

### 4.2. Les fichiers Scheme produits

La création des fichiers Scheme est retardée au maximum et est invisible aux yeux de l'utilisateur. L'intégration dans l'environnement permet de mélanger des fichiers `bld` et des fichiers source Scheme dans le même projet. Bee gère les fichiers `bld` et c'est lui qui prend en charge la génération des modules Scheme. Lorsqu'un projet contenant une interface est compilé, Bee commence par forcer la création du code Scheme de cette interface. Pour cela, il invoque Buildoo en lui passant le paramètre `-generate`, ce qui met en marche un compilateur de fichier `bld`. Ce dernier "déserialise" la structure qui est dans le fichier `bld` et la parcourt pour créer le code Scheme.

## 5. Qualité du code produit

Comme nous l'avons précédemment mentionné, le code utilisateur et le code généré sont distinctement séparés. Ceci simplifie la génération de code. Pour chaque interface (il faut noter qu'une application réelle peut contenir plusieurs interfaces) Buildoo génère un module Scheme contenant le code produit. Ce module importe les modules implémentant les fonctions spécifiant les comportements des objets graphiques. Nous avons pensé que cette séparation code produit/code utilisateur rendait la gestion de projets plus simple et plus efficace. En effet, comme l'utilisateur n'a pas le moyen de modifier le code Scheme généré, Bee est capable d'assurer l'exacte correspondance entre les fichiers `bld` et les modules Scheme générés par Buildoo. Ceci assure qu'à tout moment, une interface graphique peut-être chargée et modifiée dans Buildoo. Les modifications de l'interface sont faites directement à la souris en chargeant un fichier `bld` avec Buildoo, et pour modifier un comportement, un module Scheme est édité depuis Bee. Supposons, par exemple, qu'on souhaite fournir une interface graphique à un programme déjà existant. Avec Buildoo nous pourrons le faire sans devoir modifier (ou presque) le code déjà écrit. Ce schéma s'adapte harmonieusement au modèle de compilation séparée mis en œuvre dans Bigloo.

Buildoo produit un code Scheme compact et lisible. Même s'il est vrai que l'utilisateur n'a pas accès à ce code, nous avons pensé que notre tâche d'implantation serait simplifiée par un code produit lisible. En effet, grâce à cet effort, il nous est maintenant facile de récupérer et d'étudier du code généré par Buildoo. Lors de la sérialisation, seules les valeurs non standard des champs (les valeurs des champs qui ont été modifiés), obtenues par introspection sur la widget, sont enregistrées. Ceci permet de diminuer la taille des fichiers `bld` et des fichiers Scheme produits.

À titre d'exemple, nous présentons dans la figure 8 le code produit pour l'interface décrite Section 2.2:

```
(define (compteur)
    (let* ((Compteur (instantiate::top-level
                (title "Compteur")
                (min-size '(100 100))
                (geometry "172x100+774+594")))
           (pseudo-Compteur (instantiate::frame
                (parent Compteur)))
           (label1005 (instantiate::label
                (parent pseudo-Compteur)
                (relief "ridge")
                (width 5)
                (text "0")
                (foreground "red")
                (font "(helvetica 20 bold roman)")))
           (button1004 (instantiate::button
                (parent pseudo-Compteur)
                (text "-")))
           (button1003 (instantiate::button
                (parent pseudo-Compteur)
                (text "+"))))
       (button-command-set! button1004 (lambda ()
                (decrement)
                (label-text-set! label1005 (count-value))))
       (button-command-set! button1003 (lambda ()
                (increment)
                (label-text-set! label1005 (count-value))))
       (pack pseudo-Compteur :expand #t :fill "both")
       (grid label1005 :row 0 :column 1)
       (grid button1004 :row 0 :column 2)
       (grid button1003 :row 0 :column 0)
       Compteur))
```

FIG. 8 –: *Code de l'interface*

Le module `counter` contient les fonctions `decrement`, `increment` et `count-value`.

## 6. Auto génération (*Bootstrap*)

L'auto-génération est un procédé qui consiste à utiliser un logiciel pour sa propre construction. Dans le cadre d'un constructeur d'interfaces, l'auto-génération consiste à générer l'interface graphique du constructeur en utilisant le constructeur lui-même.

L'auto-génération est un procédé intéressant sous plusieurs aspects. Il permet en un premier temps de valider un logiciel. C'est-à-dire qu'il permet de voir si ce logiciel est suffisamment efficace et complet pour répondre à toutes les attentes de l'utilisateur. Si le logiciel a des lacunes ou s'il contient des erreurs d'implantations, il serait impossible de créer une version auto-générée. En deuxième lieu, l'auto-génération a l'avantage d'aider la mise au point. Par exemple, pour redessiner un constructeur d'interfaces graphiques on a été obligé de se servir de toutes les fonctionnalités de Buildoo. Cela nous a permis de trouver de nombreuses erreurs. Le dernier point favorable de l'auto-génération dans ce cas est que l'on simplifie remarquablement l'évolution de Buildoo. En effet une fois que nous avons un constructeur autogène d'interfaces, il devient vraiment très simple de le modifier et de l'étendre. Il suffit d'ouvrir une session de Buildoo, d'apporter graphiquement les modifications à son interface, et de le recompiler.

L'auto-génération de Buildoo n'est pas encore complètement terminée par manque de temps. Toutes les principales fenêtres graphiques ont été générées. Il ne manque que quelques panneaux de configuration, qui sont rendus plus complexes par leur caractère automatique. Il faut en effet se souvenir que les panneaux de configuration sont générés automatiquement grâce aux mécanismes d'introspection. Il a été simple d'insérer morceau par morceau toutes les parties autogènes de l'interface (grâce à la modularité de Bigloo). Il n'a quasiment pas été nécessaire de faire des modifications à la main dans les modules déjà existants, mis à part quelques variables renommées.

Afin de montrer la qualité du code produit par Buildoo, nous avons opéré quelques mesures de performance:

| | Buildoo autogène | Buildoo |
|---|---|---|
| Taille de l'exécutable(en Kb) | 1015 | 917 |
| Lignes de code Scheme | 8178 | 9199 |
| Temps de compilation | 107,8 | 100 |

Pour les temps de compilation, nous considérons le temps de compilation de la version originale de Buildoo comme temps de référence (100).

En conclusion, nous pouvons affirmer que le fait d'avoir pu implanter l'interface graphique de Buildoo avec Buildoo lui-même prouve que Buildoo est un constructeur d'interfaces graphiques efficace et puissant.

## 7. État de l'art

Nous présenterons dans cette section une comparaison avec d'autre constructeurs d'interfaces existants. En particulier nous étudierons Visual Caffe 2.0[4], Visual C++5.0[3], Glade 0.5.5[2] et Allegro 5.0.1[1].

Pour comparer ces constructeurs nous avons conçu une interface de référence constituée de quelques widgets élémentaires (`radiobuttons`,`buttons`,...) et nous l'avons réalisée avec ces différents constructeurs.

Les principaux critères que nous avons retenus pour cette étude sont: la simplicité d'utilisation, les fonctionnalités proposées, la qualité et la taille du code généré.

Tous les constructeurs d'interfaces, utilisent des stratégies de placement des widgets. Ces stratégies sont en fait un ensemble de règles qui indiquent les règles de placement des widgets dans leur fenêtre.

Trois grandes familles de stratégies existent: `PACK`, `GRID` et `PLACE`. La stratégie `PACK` positionne les widgets les unes par rapport aux autres. La stratégie `GRID` positionne les widgets dans une grille qui recouvre la fenêtre et dont on peut régler le nombre de lignes et de colonnes. Enfin, la stratégie `PLACE` positionne simplement les widgets librement à la souris. Il existe aussi des stratégies hybrides mélangeant les stratégies précédentes.

## 7.1. Glade

Le placement des widgets avec Glade n'est pas réellement intuitif. Ce constructeur utilise la stratégie `GRID`. Si ce choix permet d'obtenir des interfaces avec une certaine cohérence géométrique (les widgets sont bien alignées entre elles), il rend pratiquement impossible la création d'interfaces où des widgets se recouvrent partiellement. En ne proposant que cette stratégie de placement, Glade manque de flexibilité. Une interface telle que celle présentée Figure 9 serait difficile à réaliser avec Glade.

Glade ne propose pas la multi-sélection. Ceci a sûrement simplifié la réalisation du constructeur, mais c'est à nos yeux une grosse lacune comme nous l'avons exposé en 2.

Les interfaces dessinées avec Glade sont sauvegardées dans un fichier au format XML. Le vrai code n'est généré qu'à partir de cette représentation intermédiaire. Glade ne s'occupe pas vraiment de la connexion du code extérieur, il ne permet que d'indiquer un nom de fonction correspondant à une action d'une widget de l'interface. Glade génère un squelette pour ces fonctions nommé `callback.c`. Ce procédé oblige l'utilisateur à éditer le fichier généré par Glade. Par contre, choisir un format intermédiaire neutre permet à Glade de produire des interfaces dans différents langages de programmation.

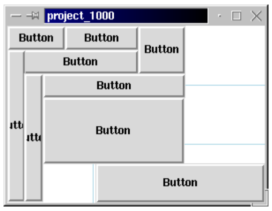


FIG. 9 –: *Une interface compliquée*

## 7.2. Visual Caffe, Visual C++ et Allegro

Visual Caffe, Visual C++ et Allegro Common Lisp ont de nombreux points communs: ce sont tous trois des environnements intégrés de programmation. Ils utilisent tous les trois la stratégie `PLACE`. Cette stratégie de placement des objets est très simple et intuitive, mais limitée. En effet, lorsque l'on développe une interface complexe, il est fastidieux d'ajuster l´alignement des widgets et les interfaces qui utilisent cette stratégie supportent mal d'être retaillées. Notons que Visual Caffe offre aussi l'accès aux autres stratégies (nombreuses) de placement de Java, mais seule la stratégie `PLACE` est réellement utilisable. Ces trois environnements sont peu intuitifs et parfois déroutants. Allegro, par exemple, présente à l'utilisateur un panneau de configuration trop compliqué où certains champs ne sont pas modifiables.

Le code correspondant aux interfaces réalisées est généré au fur et à mesure que l'utilisateur dessine l'interface sans passer par une représentation intermédiaire. D'autre part, ils n'offrent que très peu d'aide pour la saisie des callbacks. L'utilisateur doit donc insérer son code dans le fichier produit par le constructeur.

| | Buildoo/Bee | Allegro | Glade | Visual C++ | Visual Caffe |
|---|---|---|---|---|---|
| Statut | Free | Comm. | Free | Comm. | Comm. |
| Langage produit | Bigloo | LISP | C, C++, Perl, Java, Ada 95, Eiffel | C++ | Java |
| Gestion des projets | non / oui | oui | oui | oui | oui |
| Multi-sélection | oui | oui | - | oui | oui |
| Représantatioon intermédiaire | oui | - | oui | - | - |
| Code autonome | oui | - | - | - | - |
| Code généré[1]: | | | | | |
| Nombre de fichiers | 1 | 4 | 8 | 16 | 6 |
| Lignes de code | 23 | 126 | 324 | 1149 | 201 |
| Stratégies de palcement: | | | | | |
| `GRID` | oui | - | oui | - | oui[2] |
| `PACK` | oui | - | - | - | oui[2] |
| `PLACE` | oui | oui | - | oui | oui |

FIG. 10 –: *Les différentes caractéristiques des constructeurs étudiés*
*[1] Code correspondant à une interface contenant un button et un radiobutton, sans faire aucune autre modification.*
*[2] Présent mais non utilisable.*

## 8. Conclusion

Dans cet article nous avons présenté Buildoo, un constructeur d'interfaces graphiques pour le système Bigloo. Buildoo est un nouveau composant de l'environnement de programmation intégré Bee.

Buildoo est encore un prototype et n'a pas été utilisé dans des projets de taille réelle. Cependant, cet outil est dans un état d'avancement suffisant pour permettre son auto-génération (la nouvelle interface de Buildoo est construite avec Buildoo lui-même).

Les atouts de Buildoo par rapport aux autres constructeurs d'interface, sont sans doute, la complète séparation entre le code et l'interface, la qualité et la taille du code produit ainsi que sa facilité d'évolution.

Les prochaines versions de Buildoo verront le développement d'un module permettant la conception de nouvelles widgets et leur intégration automatique. Par ailleurs, nous nous attacherons à faciliter la personnalisation de l'interface et à la rendre plus intuitive.

## 9. Remerciements

Je tiens à remercier Manuel Serrano et Erick Gallesio qui m'ont beaucoup aidé dans la rédaction de cet article, et Philippe Guigue, Nicolas Karasek et Cedric Bonnin qui sont les autres auteurs de Buildoo.

## Références

[1] Allegro Home Page. http://www.franz.com/.
[2] Glade Home Page. http://glade.pn.org/.

[3] Visual C++ Home Page. http://msdn.microsoft.com/visualc/.

[4] Visual Caffe Home Page. http://www.symantec.com/domain/cafe/index.html.

[5] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. In *special issue*, number 23 in SIGPLAN Notices, September 1988.

[6] E. Gallesio and M. Serrano. Biglook user's manual. Rt, I3S, France, January 1999.

[7] R. Kelsey, W. Clinger, and J. (eds) Rees. The Revised$^5$ Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[8] M. Serrano. Bigloo user's manual. RT 0169, INRIA-Rocquencourt, France, December 1994.