

Étude d'une Boîte à Outils Multi-Dispositifs

Pierre Dragicevic

Jean-Daniel Fekete

ÉCOLE DES MINES DE NANTES
4, rue Alfred Kastler, La Chantrerie
44 307 Nantes Cedex 3
{dragice, feket} @emn.fr

RÉSUMÉ

Les boîtes à outils graphiques actuelles ne gèrent qu'un clavier, une souris et un nombre limité de techniques d'interaction. La prolifération de nouveaux dispositifs, en particulier pour les jeux, la CAO, le dessin et l'accès aux handicapés n'est donc pas prise en compte.

Cet article présente une étude des difficultés à gérer les modes d'interaction étendus au niveau des boîtes à outils, ainsi qu'une architecture de widget découplant la gestion des événements de la présentation. Cette architecture est réminiscente du modèle MVC de Smalltalk : elle redonne son existence à la composante contrôle qui avait disparue des boîtes à outils. Cependant, notre version définit une interface générique de communication avec la partie vue du widget. Plusieurs contrôleurs peuvent fonctionner en parallèle dans le même widget, autorisant l'interaction parallèle tout en évitant les conflits à un niveau de granularité correspondant aux zones visuelles gérées par le widget.

Nous montrons une implémentation partielle de l'architecture dans Java Swing et décrivons quelques contrôleurs spécifiques que nous avons implémentés : un pointeur contrôlé par le clavier et un autre par la voix. Ces pointeurs coexistent et fonctionnent parallèlement sans que l'interface utilisateur ne soit modifiée.

MOTS CLÉS : architecture logicielle, dispositifs d'entrée, boîte à outils, interfaces pour handicapés.

INTRODUCTION

Les boîtes à outils graphiques supposent que l'application qu'elles servent à construire n'utilisera jamais qu'une souris, un clavier et un ensemble limité de techniques d'interaction. Cependant, plusieurs raisons nous poussent à diversifier les dispositifs d'entrée et les styles d'interaction (que nous appellerons ensemble méthodes d'interaction). Une première raison est la prolifération de nouveaux dispositifs d'interaction, due au développement d'applications très avides de ces dispositifs tels les jeux, la CAO, la modélisation 3D ou l'illustration graphique [9] et la musique. Une seconde raison est que les interfaces graphiques ne sont plus limitées aux machines de bureau ou portables ; les *palmtops* et les ordinateurs vestimentaires utilisent aussi des interfaces graphiques, bien qu'avec des dispositifs autres qu'une souris et un clavier. La portabilité d'applications entre les machines de bureau et les machi-

nes nomades simplifierait considérablement la tâche des développeurs et des utilisateurs. Une troisième raison est l'utilisabilité par les handicapés, qui ont besoin de méthodes d'interaction adaptées. Enfin, les applications fonctionnant sur des machines de bureau pourraient aussi bénéficier des nouvelles méthodes d'interaction comme la reconnaissance de geste, de parole, ou l'interaction bimanuelle, pour ne citer qu'elles [3, 15, 26].

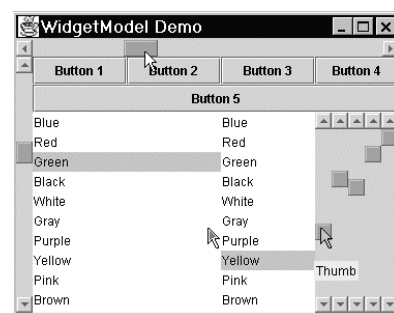


Figure 1 : Exemple de Swing multi-dispositifs avec un curseur pour la souris, le clavier et un dispositif vocal.

Cet article décrit une architecture de boîte à outils qui découple les widgets de leur méthode d'entrée. Comme preuve de faisabilité, nous montrons une modification de la boîte à outils Swing de Java et quelques exemples d'extensions (figure 1). Une propriété intéressante de cette architecture est d'être presque transparente au programmeur d'application utilisant la boîte à outils. Cette caractéristique permet de réutiliser des applications déjà écrites.

Cette architecture ne peut pas être qualifiée de multimodale dans le sens où elle ne s'intéresse ni à la fusion de modalités [30], ni à l'interprétation sémantique de modalités comme la parole. Cependant, une application multimodale gagnerait à reposer sur une architecture facilitant l'accès à des dispositifs étendus.

Cet article est organisé comme suit : la section 1 répertorie les difficultés posées par la gestion de dispositifs multiples dans une boîte à outils. La section 2 décrit les modifications faites à Swing pour que les widgets puissent être séparés de la gestion des méthodes d'entrée. La section 3 décrit des exemples de méthodes d'entrée que nous avons ajoutées à Swing. La section 4 compare nos résultats à des travaux similaires avant de conclure.

ÉTUDE DES PROBLÈMES POSÉS PAR LA GESTION DE DISPOSITIFS MULTIPLES DANS LES BOÎTES À OUTILS

Nous avons classé les difficultés en 4 parties :

- 1) intégration de nouveaux types d'événements et du contrôle des dispositifs ;
- 2) gestion du retour des dispositifs positionnels ;
- 3) mécanismes de distribution des événements, et de gestion du focus ;
- 4) gestion des événements au niveau des widgets.

Événements multi-dispositifs

La gestion d'événements multi-dispositifs pose deux types de problèmes : la gestion unifiée de nouveaux types d'événements et le support de dispositifs virtuels.

Gestion unifiée de nouveaux types d'événements. Les dispositifs sont considérés soit de façon logique, soit de façon structurelle. Foley décrit les dispositifs d'entrée dans [Foleetal90] et les classe en cinq *dispositifs logiques* de base : *locator*, *pick*, *valuator*, *keyboard* et *choice*. Cette classification vient de GKS et PHIGS [2, 32]. CORE inclut aussi un dispositif *button* [14]. Les dispositifs logiques sont mieux adaptés à l'interaction modale où l'application attend des classes d'événements spécifiques, qu'à la manipulation directe où par exemple un clic de souris peut signifier une sélection ou une position.

Au lieu de dispositifs logiques, la *X Input Extension* [12] décrit un dispositif structurellement comme un ensemble de classes. Chaque dispositif concret gère une ou plusieurs classes de dispositifs. Six classes sont définies : *key*, *button*, *valuator* ; *proximity*, *focus* et *feedback*. Les dispositifs sont aussi accessibles comme objets afin d'être configurés ou interrogés sur leur état. Tandis que les dispositifs de GKS/PHIGS/CORE ressemblent à des méthodes d'interaction primitives, les dispositifs étendus de X11 décrivent concrètement les signaux envoyés par les dispositifs physiques, ainsi que leur état et leur configuration.

Le second choix étend de façon naturelle les événements gérés par les boîtes à outils graphiques telles que Motif, MFC ou Java AWT [17, 33, 13].

Très peu de nouveaux types d'événements sont nécessaires pour gérer tous les événements concrets : *proximity in*, *proximity out* et *device state changed*. Tous les autres types (*key press*, *key release*, *button press*, *button release* et *move*) existent déjà et s'adaptent naturellement aux dispositifs étendus. De plus, les événements logiques peuvent être synthétisés à partir d'événements structurels (et non l'inverse). Cela introduit une autre dimension dans les types d'événements et de dispositifs: les dispositifs physiques et virtuels [31].

Les dispositifs virtuels. Toutes les boîtes à outils transforment des événements concrets en événements synthétiques ; par exemple un *key press* en événement « chaîne de caractères ». De nombreuses boîtes à outils comme MFC

ou AWT considèrent aussi les widgets comme des dispositifs virtuels produisant des événements synthétiques. De même, ces événements synthétiques sont produits par les systèmes de reconnaissance vocale [28], de geste [34], et d'écriture [7].

Dans la cascade, l'identification des dispositifs physiques à l'œuvre est nécessaire pour permettre une interaction indépendante, parallèle et combinée [10].

Affichage de la position des dispositifs

Les systèmes de fenêtrage classiques affichent la position de la souris à l'aide d'un curseur. PenPoint de GO [20] et Windows CE [7] gèrent une *trace*. Ces mécanismes sont très primitifs dans le système, et ne peuvent être utilisés pour des dispositifs positionnels supplémentaires. La X Input Extension permet l'affichage de curseurs multiples mais ne gère qu'un curseur et fait une distinction importante entre le pointeur principal et les autres dispositifs.

La gestion de l'affichage multi-dispositifs nécessiterait une couche graphique partagée (*overlay*) où des objets graphiques pourraient être installés et animés. L'utilisation d'une telle couche dans une application est commune [4, 6, 16, 29], mais aucun mécanisme inter application standard n'existe pour l'instant, ce problème ne peut donc pas être entièrement résolu par les boîtes à outils.

Aiguillage d'événements et gestion du focus

L'aiguillage d'événements et la gestion du focus sont gérés à trois niveaux : le système de fenêtrage, la boîte à outils et parfois le widget.

Le système de fenêtrage. Celui-ci envoie les événements positionnels à la fenêtre placée sous la position de l'événement et les autres événements à la fenêtre ayant le focus. Ce focus peut être géré explicitement ou suivre le pointeur. Bien entendu, des exceptions à ce schéma existent : lorsqu'une fenêtre reçoit un événement *button down*, elle reçoit aussi tous les événements envoyés par la souris jusqu'au *button up* (*grab*). De plus, des événements sont envoyés pour signaler à chaque fenêtre lorsque la souris y entre ou en sort.

La boîtes à outils. Celle-ci envoie les événements positionnels au widget placé sous la position de l'événement et les autres au widget ayant le focus. Ce focus peut être géré explicitement ou suivre le dernier clic de la souris. L'exception du *grab* existe aussi à ce niveau. Les widgets reçoivent aussi des événements lorsque la souris y entre ou en sort. De plus, les événements produits par le clavier sont dérivés pour implémenter les raccourcis claviers.

Le widget. Les menus et les boîtes de dialogue modales déroutent les événements. Les menus redirigent parfois les événements positionnels pour ouvrir ou fermer un autre menu. Les boîtes de dialogue empêchent certains événements de se propager ailleurs que dans les widgets fils du dialogue.

Selon notre expérience, les mécanismes d'aiguillage des événements ne sont jamais simples à comprendre et leur

implémentation est généralement très difficile à suivre car les responsabilités sont distribuées à plusieurs objets.

Pour les dispositifs étendus, aucun système de fenêtrage n'applique les mêmes règles que pour la souris et le clavier. Pourtant, le mécanisme se généralise ainsi :

- Les widgets doivent implémenter le *grab* pour tous les dispositifs positionnels.
- La boîte à outils doit envoyer des événements lorsque les événements positionnels entrent et sortent des widgets.
- La cascade de gestion des raccourcis clavier doit être étendue aux événements non positionnels autres que ceux du clavier.

Des problèmes subsistent néanmoins :

- La stratégie de gestion du focus n'est pas claire. Dans les boîtes à outils traditionnelles, le focus est laissé sur le widget qui a subi la dernière manipulation. Pendant et après une interaction parallèle ou concurrente avec des dispositifs positionnels, ce focus passe alternativement d'un widget à un autre et l'utilisateur peut avoir des doutes sur celui qui prime.
- L'extension du mécanisme de raccourcis clavier ne s'étend pas si facilement à tous les événements non positionnels qui sont généralement plus complexe que les événements clavier. L'interface des widgets doit être modifiée.

Gestion des événements au niveau des widgets

Certaines boîtes à outils 3D gèrent des dispositifs étendus [35, 38, 36], mais seulement pour la manipulation directe d'objets 3D. Par exemple, seuls les dispositifs standard peuvent être utilisés pour la sélection dans un menu.

Très peu d'articles décrivent des boîtes à outils 2D multi-dispositifs. Parmi ceux-ci, MMM [5] est un multi-éditeur multi-utilisateur multi-dispositifs implémenté dans l'environnement Cédar de Xerox. Plusieurs souris peuvent être utilisées par des utilisateurs différents, mais il n'est pas possible d'utiliser d'autres dispositifs. Chatty décrit l'extension de WHIZZ [10] pour la gestion de l'interaction bimanuelle. Toutefois, cet outil est spécialisé dans la manipulation d'objets graphiques, et non de widgets.

Widgets avec dispositifs étendus. Il y a deux interprétations d'un widget gérant des dispositifs étendus. Les widgets peuvent être conçus pour gérer des dispositifs étendus et permettre des interactions plus ou moins sophistiquées selon la présence ou non de certains dispositifs. Une autre interprétation est l'aiguillage d'événements d'une méthode d'interaction vers plusieurs widgets, d'une manière similaire aux raccourcis clavier.

Par exemple, une souris à molette est un dispositif souris ordinaire qui peut envoyer des événements non-positionnels pour le défilement. Cette situation est analogue à un clavier avec des touches de fonction ; le clavier étant un clavier ordinaire et les touches de fonction étant une extension émettant des événements non-positionnels.

Les touches de fonction peuvent initier des actions dans d'autres widgets de la même manière qu'une molette peut initier une action de défilement dans un autre widget. La seule différence est la quantité d'information à transmettre ; dans le cas du clavier, seule la touche concernée est nécessaire alors que pour des dispositifs généraux, des paramètres doivent être passés pour l'action du raccourci.

UNE NOUVELLE ARCHITECTURE DE WIDGETS

Dans cette section, nous décrivons une nouvelle architecture de widgets inspirée du paradigme MVC, qui permet de gérer des dispositifs multiples au niveau du widget.

Présentation de l'architecture

Notre architecture est représentée sur la figure 2. Elle comporte quatre composants principaux: l'objet *Modèle* (*M*), deux types de contrôleurs (*C*): des *Contrôleurs de Modèle* et des *Contrôleurs de Vue*, et un objet *Vue/Zones/Manipulateurs* (*Vzm*).

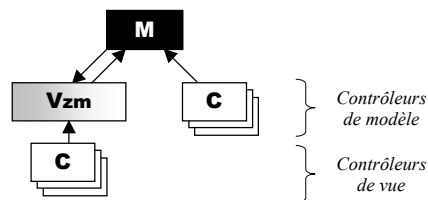


Figure 2: Une architecture de widget modulaire

L'objet "Modèle". Cet objet est le même que celui décrit dans MVC. Il représente le type de données que le widget sert à manipuler.

Le Contrôleur de Modèle. Cet objet correspond à l'objet *Contrôleur* de MVC, qui traduit certains événements utilisateur en changements dans le modèle. Il est appelé ainsi par opposition au *Contrôleur de Vue*. L'objet *Modèle* peut gérer des *Contrôleurs de Modèle* multiples.

Le Contrôleur de Vue. Le *Contrôleur de Vue* traduit certains événements utilisateur (en général, des événements positionnels) en manipulations de la vue. Il communique avec la vue (ici, l'objet *Vzm*) à travers un protocole spécial que nous décrivons par la suite. Ce protocole autorise aussi les contrôleurs multiples.

L'objet "Vue/Zones/Manipulateurs". Il s'agit d'un objet *Vue* classique, mais qui implémente une interface lui permettant d'être manipulé par des contrôleurs. Cette interface exhibe la vue comme étant un *objet graphique contenant un ensemble de zones qui réagissent à des manipulations de base*. L'objet *Vzm* joue aussi le rôle de *Contrôleur* pour le modèle, car il traduit les manipulations de la vue en manipulations du modèle.

Dans la suite, nous présentons les objets *Zone* et *Manipulateur* qui constituent les briques de base du protocole de communication entre l'objet *Vzm* et les contrôleurs.

Les zones d'un widget. La plupart des widgets possèdent un ensemble de zones caractéristiques. Pour une barre de défilement, il s'agit des zones *incrémentaire unitaire*, *décrémentaire unitaire*, *pouce*, *incrémentaire par bloc*, et *décrémentaire par bloc* (figure 3).

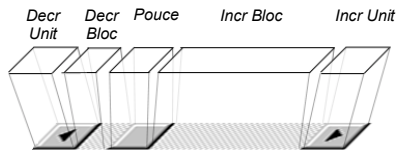


Figure 3: Les zones d'une barre de défilement

L'objet *Vzm* peut communiquer la liste des types de zone qui sont susceptibles d'être affichés et manipulés, et possède une méthode permettant d'identifier la zone concrète située sous un point particulier. L'objet *Zone* joue principalement un rôle d'identificateur. Cependant, les zones d'un widget peuvent varier dynamiquement (par exemple, un widget *Liste* ne contient pas toujours les mêmes éléments).

Manipulation de zones et interaction concurrente. A chaque zone, l'objet *Vzm* associe un objet *Manipulateur* similaire aux manipulateurs d'Unidraw ou aux interacteurs de Garnet [37, 29]. Cet objet décrit les principales manipulations atomiques qui peuvent être effectuées sur une zone :

```
Start(Point p)
Stop(Point p)
Press(Point p)
Release(Point p)
Drag(Point p, Point delta)
```

Les méthodes de manipulation prennent un point comme unique argument (sauf la méthode *Drag* qui prend un paramètre *delta* supplémentaire, permettant les déplacements non modaux). Ce point est l'emplacement de l'événement dans le widget. Les manipulations sont initiées par la méthode *start* et terminées par la méthode *stop*. Le protocole se résume à un ensemble de contraintes :

1. Un manipulateur ne peut pas être utilisé par deux contrôleurs à la fois,
2. des règles doivent être respectées quant à l'ordre d'appel des méthodes.

La contrainte 1. est vérifiée par le contrôleur qui s'assure que le manipulateur n'est pas déjà en cours d'utilisation avant de le démarrer. La plupart du temps, une série de manipulations est initiée par l'appui d'un bouton et se termine lorsque le bouton est relâché, comme décrit par Buxton [8]. La gestion des interférences présente dans le *Modèle* permet de désactiver temporairement certains manipulateurs afin d'éviter les manipulations contradictoires.

La contrainte 2. impose que lorsqu'un contrôleur utilise un manipulateur, il doit s'assurer que ses méthodes sont appelées dans le bon ordre. Cet ordre correspond aux règles de l'automate sous-jacent.

Les avantages de cette architecture

La modularité est un critère essentiel pour une gestion efficace des dispositifs multiples au sein des widgets. Elle offre non seulement des possibilités de configuration dynamique, mais surtout une extensibilité qui, dans notre cas, compense rapidement le surcoût initial de développement du widget. En guise d'exemple, nous nous placerons dans une situation d'extension incrémentale d'un widget dans le but de gérer de nouveaux types d'événements. Nous introduirons les notions de *manipulation au niveau Modèle* et de *manipulation au niveau Vue*, afin de montrer que le problème peut être séparé en deux cas. Les architectures naïves offrent une modularité triviale dans le premier cas, mais pas dans le deuxième, où notre modèle fournit une solution bien plus acceptable.

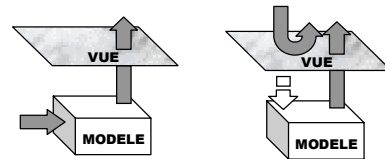


Figure 4: Illustration des concepts de manipulation au niveau Modèle (à gauche) et de manipulation au niveau Vue (à droite).

Manipulations au niveau Modèle. Un utilisateur effectue une manipulation au niveau *Modèle* lorsqu'il modifie directement le modèle du widget (figure 4, à gauche). C'est le cas par exemple lorsqu'il sélectionne un élément d'une liste avec des touches fléchées ou avec des commandes vocales. Dans notre architecture, ces manipulations sont gérées par des *Contrôleurs de Modèle*. Ce schéma est réductible à un modèle MVC « idéal », dans lequel la vue et le contrôleur ne se connaissent pas. A chaque fois qu'un widget doit prendre en compte un nouveau type d'événement, il suffit d'ajouter un nouvel objet *Contrôleur* (figure 5) : chaque *Contrôleur* (*Ca*, *Cb*, *Cc*) reçoit et interprète un type particulier d'événement concret (*a*, *b*, *c*).

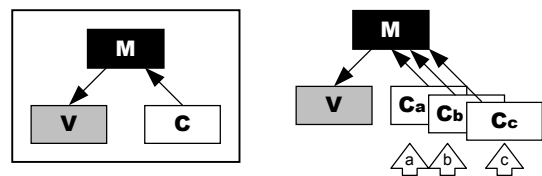


Figure 5: Architecture MVC, et schéma d'extension d'un widget pour prendre en compte les types d'événements *a*, *b*, puis *c*.

Manipulations au niveau Vue. Un utilisateur effectue une manipulation au niveau *Vue* lorsqu'il interagit directement avec la vue du widget (figure 4, à droite). C'est le cas lorsqu'il clique sur un élément d'une liste, ou lorsqu'il déplace le pouce d'une barre de défilement. Le plus souvent, les actions de l'utilisateur (en général, des événements positionnels) ne peuvent pas être interprétées indépendamment de la vue. La vue doit déterminer quel élément d'une liste a été sélectionné, ou comment interpréter le déplacement d'un pouce (figure 6). De plus, elle gère la plupart du temps des états d'interaction et des retours graphiques de façon locale.



Figure 6: Déplacement d'un slider linéaire et angulaire.

Dans ce cas, on a souvent recours à une architecture MVC modifiée où la vue et le contrôleur forment un objet unique, correspondant au *look-and-feel* du widget. Outre la complexité d'un tel objet, il est nécessaire de le dériver à chaque fois qu'un nouveau type d'événement doit être pris en compte (figure 7).

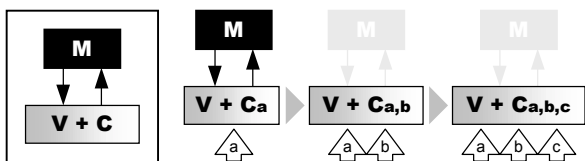


Figure 7: Architecture « M, V+C », et schéma d'extension d'un widget.

Dans notre architecture, l'objet *V+C* (*Vue / Contrôleur*) est remplacé par l'objet *Vzm* (*Vue / Zones / Manipulateurs*), qui conserve sa présentation (*look*) et son comportement associé (*feel*), mais ne reçoit plus les événements concrets. La gestion de ces événements est déplacée dans des *Contrôleurs de Vue*, ce qui rend à nouveau l'architecture modulaire. L'extension d'un widget pour des manipulations au niveau *Vue* devient plus naturelle, et beaucoup moins coûteuse (figure 8).

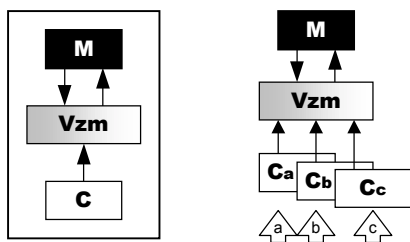


Figure 8: Notre architecture et le schéma d'extension correspondant, pour des manipulations au niveau *Vue*.

Contrôleurs spécifiques et génériques. Un *Contrôleur de Modèle* doit connaître explicitement son *Modèle*, et il est par conséquent spécifique à une classe de widget. Par exemple, il n'est pas naturel d'utiliser le même contrôleur clavier pour tous les types de widgets.

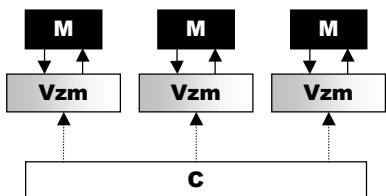


Figure 9: Schéma d'un Contrôleur de Vue générique et partagé.

En revanche, l'utilisation de zones et de manipulateurs permet d'écrire des *Contrôleurs de Vue* génériques. Il est ainsi possible d'écrire un contrôleur clavier qui peut mani-

puler n'importe quel type de widget, en utilisant par exemple un pointeur. Il en est de même bien sûr pour tout contrôleur recevant des événements positionnels. En plus d'être *génériques*, ces contrôleurs sont aussi *partagés*, c'est-à-dire qu'ils contrôlent l'ensemble des widgets affichés à l'écran, en gardant une référence sur le manipulateur courant (figure 9).

Parfois, il peut être aussi utile d'écrire des contrôleurs de vue spécifiques au widget. Les contrôleurs génériques ne savent rien sur les zones qu'ils manipulent, alors que les contrôleurs spécifiques connaissent la sémantique des zones du widget. Cela permet par exemple d'écrire un contrôleur qui déplace le pouce d'une barre de défilement avec une molette de souris étendue.

EXTENSION DE SWING POUR UNE GESTION DES ENTRÉES MULTIPLES

Dans cette section, nous décrivons brièvement les modifications que nous avons apportées à la boîte à outils Swing afin de prendre en compte les dispositifs multiples au niveau widget. Ces modifications ont un impact minimal sur les applications existantes, l'interface du widget restant inchangée.

L'architecture des widgets dans Swing

Dans l'AWT (prédécesseur de Swing), les objets *Widget*¹ sont monolithiques [13]. Dans Swing, ces widgets ont été reconçus selon une architecture *M, V+C* [19]. L'objet *V+C* est appelé *UI* (*User Interface*). Cet objet communique directement avec l'objet *Widget*, dont une partie des méthodes est déléguée au *Modèle* (figure 10).

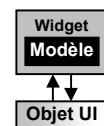


Figure 10: Architecture d'un widget dans Swing

Cette architecture a été introduite afin de faciliter la programmation orientée modèle, et fournir un support pour les *look-and-feels* multi-plateformes. Cependant, Swing gère uniquement une souris et un clavier ; son architecture ne permet pas d'ajouter de nouvelles méthodes d'entrée.

Adaptation des widgets Swing

Afin d'adapter l'architecture des widgets Swing à celle décrite dans la partie précédente, nous avons modifié le rôle de certains composants et introduit de nouveaux objets (*Contrôleur*, *Zone* et *Manipulateur*, ainsi qu'une interface *InterfaceUI*) (figure 11).

La classe *UI* a été modifiée afin de pouvoir enregistrer un ou plusieurs contrôleurs. Pour des raisons d'implémentation, l'objet *UI* doit connaître ses contrôleurs afin de leur déléguer la gestion des événements (que lui-même se voyait déléguer par l'objet *Widget*, source de

¹ Bien que dans Swing, les widgets soient appelés *Components*, nous utilisons le terme *Widget*.

l'événement). L'objet *Widget*, qui ne voit que son *UI*, reste inchangé.

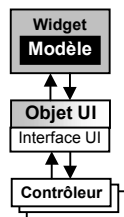


Figure 11: L'architecture de Swing remaniée

Chaque objet *UI* concret doit maintenir un ensemble de *Zones* et de *Manipulateurs*, et implémenter l'interface *UIInterface* afin de permettre aux contrôleurs d'accéder à ces objets.

Dans cette implémentation, il n'y a pas de distinction explicite entre les *Contrôleurs de Vue* et les *Contrôleurs de Modèle*, ce qui permet des comportements hybrides (exemple d'une souris avec molette). Tous les contrôleurs communiquent directement avec l'objet *UI*, qui, au besoin, leur donne accès à l'objet *Widget*, ainsi que son *Modèle*.

Nous avons réécrit un ensemble de widgets Swing existants, afin de gérer les contrôleurs standard selon notre architecture. Ces widgets possèdent des objets *UI* qui sont entièrement déconnectés des événements d'entrée. La gestion des entrées a été déplacée dans deux contrôleurs : un contrôleur clavier (par widget) et un contrôleur souris générique. Grâce à cet ensemble de widgets revisités, il est possible de choisir entre les contrôleurs souris ou clavier, ou les deux ensembles.

Gestion des dispositifs non-standard au niveau toolkit

Nous nous sommes principalement intéressés à une architecture permettant de gérer les multi-dispositifs au niveau widget. Cependant, des modifications supplémentaires sont été nécessaires pour gérer les dispositifs non-standard au niveau boîte à outils. Dans notre implémentation, nous fournissons des solutions partielles aux problèmes invoqués en première partie, notamment pour gérer les dispositifs positionnels.

La gestion des événements non-standard. Swing contient deux types d'événements en entrée: *KeyEvent* et *MouseEvent*, qui dérivent de *InputEvent*. Swing n'y intègre pas la notion de dispositif source, et par conséquent, seuls le clavier et la souris standard sont pris en compte. Nous n'avons pas introduit d'objet *dispositif source*, cependant nous avons ajouté un nouveau type d'événement, *PointeurEvent*, afin de gérer les pointeurs multiples.

Le retour des dispositifs. Nous avons implémenté une classe *AlternateCursor* qui affiche et anime un curseur par-dessus une fenêtre. Plusieurs curseurs peuvent s'installer au sein d'une fenêtre, et s'afficher dans des couches sépa-

rées. L'objet *AlternateCursor* a été conçu pour s'intégrer très facilement dans les interfaces existantes, sans en modifier le fonctionnement.

L'aiguillage des événements et la gestion du focus.

Dans Swing, les mécanismes d'aiguillage des événements et de focus sont complexes et répartis dans plusieurs objets. Une révision de ce mécanisme impliquerait la modification d'une grande partie de la boîte à outils. Par conséquent, nous utilisons une technique détournée permettant de prendre en compte les événements positionnels dans Swing sans modifier les widgets. Les événements en provenance des différents pointeurs sont considérés comme des événements souris standard par Swing, tout en étant différenciables par les contrôleurs.

EXEMPLES D'APPLICATION

Nous avons expérimenté des nouveaux contrôleurs de vue qui exploitent notre architecture. Ces contrôleurs fournissent des méthodes alternatives de manipulation directe, et peuvent être utilisés ensemble dans la même application. Sur la figure 1, le curseur blanc représente le pointeur standard (la souris). Le curseur gris est un pointeur contrôlé par le clavier, et le curseur noir et blanc représente un pointeur dirigé à la voix.

Le pointeur clavier

Nous avons implémenté un pointeur clavier qui permet un contrôle aisé et fluide d'un curseur par l'intermédiaire des touches fléchées. Des touches supplémentaires permettent d'afficher les noms des zones à la manière des info-bulles, et de passer d'une zone à l'autre (figure 12).

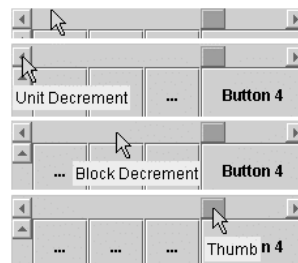


Figure 12: Les info-bulles et l'exploration des zones avec le pointeur clavier

Le pointeur clavier est une méthode d'entrée qui se révèle utile dans les environnements sans souris. Les info-bulles et l'exploration des zones sont deux exemples qui illustrent comment notre architecture peut être utilisée pour augmenter une technique d'entrée de possibilités d'exploration de widgets.

Le pointeur vocal

Un pointeur vocal, semblable à [27], a été développé à l'aide de l'API *Java Speech* [22] et du système de reconnaissance vocale *ViaVoice* d'IBM [21]. Il permet à l'utilisateur de déplacer un curseur et simuler le bouton de la souris, à l'aide de simples commandes vocales telles que *en haut*, *à droite*, *plus vite*, *appuyer*, *doucement*, ou *stop*.

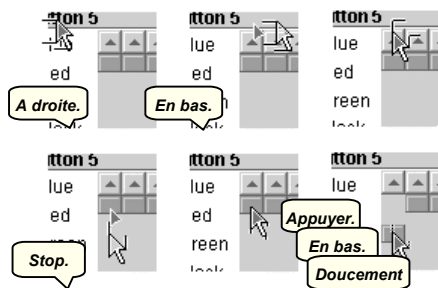


Figure 13 : Une technique de "curseur divisible" pour la reconnaissance vocale retardée

Nous avons ajouté une technique d'interaction spéciale afin de surmonter les délais de reconnaissance importants (environ une seconde), principalement dus au système de reconnaissance utilisé. Quand la parole commence, le curseur se divise en deux parties : une partie fixe, et une partie qui continue à se déplacer. Selon la commande reconnue, le curseur peut retourner à son état précédent ou continuer à se déplacer (figure 13). Cette technique permet aux commandes vocales (commandes de direction, d'arrêt ou simulation de bouton) d'être exécutées dans le contexte du début de parole. En même temps, elle évite des arrêts indésirables du déplacement du curseur en réponse à des bruits, des commandes non reconnues ou des commandes qui contrôlent la vitesse déplacement.

Ce pointeur vocal est particulièrement intéressant, car il exploite avec succès la parole en tant que dispositif positionnel et de manipulation. Ainsi, toute interface développée avec notre architecture est directement accessible par la reconnaissance vocale.

DISCUSSION

Cette architecture est liée au modèle MVC de Smalltalk, à d'autres technologies Java ainsi qu'aux interacteurs de Garnet. Nous les comparons dans cette partie.

Différences avec le modèle MVC de Smalltalk

La séparation des contrôleurs et de la vue n'est pas une technique nouvelle. La principale différence entre le modèle MVC de Smalltalk [25] et notre architecture apparaît sur la figure 3 : l'existence de *Contrôleurs de Vue* qui modifient le modèle par l'intermédiaire de la vue, ce qui résout le problème délicat du fort couplage entre vue et contrôleur. De plus, notre architecture autorise les contrôleurs multiples, et permet le partage des contrôleurs ce qui réduit le besoin de spécialiser les contrôleurs pour chaque vue.

D'autre part, notre architecture est spécifique à la gestion des objets graphiques interactifs alors que MVC ne l'est pas particulièrement (même si la plupart de ses utilisations sont orientées graphisme).

L'Accessibilité Java

Swing implémente l'API d'accessibilité de Java, destiné à connecter des technologies d'assistance en entrée/sortie pour les utilisateurs handicapés [11, 1]. Contrairement à notre architecture, la boîte à outils n'est pas consciente de

l'existence de technologies d'assistance et ne peut pas s'adapter à elles. La technologie d'assistance surveille les actions de la boîte à outils et agit sur son état. Toutefois, pour des handicaps spécifiques, Java préconise de spécialiser le *look-and-feel* [1]. Grâce à notre architecture, les utilisateurs peuvent aussi spécialiser des contrôleurs pour des méthodes d'interaction spécifiques.

Personal Java et Embedded Java

Personal Java (Java Personnel) est une API Java conçue pour utiliser Java sur des environnements limités comme les ordinateurs de poche [24]. *Personal Java* possède une version réduite de l'AWT Java avec un mécanisme de préférence très simple permettant à chaque widget de préconiser la saisie clavier, la saisie positionnelle, ou le déclenchement d'actions. Lorsqu'un widget reçoit le focus, le système déclenche la méthode d'entrée adaptée (comme par exemple faire apparaître un clavier virtuel sur un portable sans clavier). Le mécanisme de préférence est plus simple mais bien moins extensible que notre architecture.

Embedded Java (Java Embarqué) est une API Java conçue pour utiliser Java au sein de dispositifs électroniques comme un four à micro-ondes, une télévision, ou un tableau de bord de voiture [23]. Toutefois, les applications ne peuvent pas s'appuyer sur une réelle interface graphique. Au mieux, elles utilisent un affichage à cristaux liquides de faible résolution, ou un simple affichage en mode texte. Les dispositifs non-standard ont une grande importance dans *Embedded Java*, mais pas le graphisme.

Les interacteurs de Garnet

Nos manipulateurs ressemblent aux interacteurs de Garnet [29]. Les interacteurs sont des objets qui décrivent des comportements interactifs, et peuvent être connectés à des objets graphiques. Nos manipulateurs diffèrent en ce qu'ils ne traitent pas des événements concrets, mais des événements abstraits produits par les contrôleurs.

TRAVAUX FUTURS ET CONCLUSION

Dans cet article, nous avons étudié l'architecture d'une boîte à outils à widgets découplant la présentation et la gestion de l'interaction pour permettre l'utilisation de dispositifs d'interaction et de modes d'interactions variés. Ce travail est plus une preuve de faisabilité qu'une réalisation finalisée.

Néanmoins, les perspectives ouvertes sont nombreuses et nous travaillons actuellement sur la gestion des événements non positionnels de manière similaire aux raccourcis clavier. Nous voudrions aussi que les widgets modifient leurs *affordances* en fonction des contrôleurs qui leur sont connectés.

Nous pensons qu'une boîte à outils telle que nous la décrivons est un premier pas vers le déploiement des applications multi-dispositifs et à terme multimodales.

BIBLIOGRAPHIE

- [1] M. Andrews. *Accessibility and the Swing Set*. The Swing Connection, <http://java.sun.com>. Sun microsystems, 1999.

- [2] ANSI (American National Standards Institute), *American National Standard for Information Processing Systems--Computer Graphics--Graphical Kernel System (GKS) Functional Description*. ANSI X3.124-1985. ANSI, 1985.
- [3] R.M. Baecker, J. Grudin, W. Buxton and S. Greenberg, S. (Eds.), *Readings in Human Computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers. 1995
- [4] B. Bederson and J.D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics *ACM UIST '94*, 1994
- [5] E. A. Bier, S. Freeman, MMM: A User Interface Architecture for Shared Editors on a Single Screen. In *Proceedings of UIST'91*, pages 79-85, 1991.
- [6] E. Bier, M. Stone, K. Pier, B. Buxton, T. DeRose. Toolglass and Magic Lenses: the See-through Interface. In *Proceedings of SIGGRAPH*, pages 73-80. Addison-Wesley, 1993.
- [7] D. Boring. *Programming Windows CE*. Microsoft Press, 1998.
- [8] B. Buxton. Chunking and Phrasing and the Design of Human-Computer Dialogues, In Information Processing, *Proceedings of the IFIP 10th World Computer Congress*, pages 475-480. 1986.
- [9] W. Buxton. (Ed). Interaction in 3D Graphics. *Computer Graphics*, vol. 32 N° 4. ACM.
- [10] S. Chatty. Extending a Graphical Toolkit for two-handed Interaction. In *Proceedings of the ACM CHI*, page 195-203. Addison-Wesley. 1994.
- [11] R. Eckstein, M. Loy, D. Wood. *Java Swing*. O'Reilly & Associates, 1998.
- [12] P. Ferguson. The X11 Input Extension: A tutorial. *The X Resource*, vo. 4 n° 1 pp. 171-194.
- [13] D. Flanagan. *Java in a Nutshell, 2nd Edition*, O'Reilly, 1997.
- [14] Graphics Standards Planning Committee. Status Report of the Graphics Standards Planning Committee. *Computer Graphics*, 1979.
- [15] J. D. Fekete. TicTacToon: A Paperless System for Professional 2D Animation. In *Proceedings of SIGGRAPH*, pages 79-90. Addison-Wesley, 1995.
- [16] J.- D. Fekete. Using the Multi-Layer Model for Building Interactive Graphical Applications. In *Proceedings of UIST'96*, pages 109-118.
- [17] P. Ferguson, D. Brennan. *Motif Reference Manual*. pub-ORA, 1993.
- [18] J. D. Foley, A. van Dam, Steven K. Feiner, John F. Hugues. *Computer Graphics Principles and Practice*. Addison-Wesley. 1990.
- [19] A. Fowler. *A Swing Architecture Overview* (The Inside Story of JFC Component Design). The Swing Connection, <http://java.sun.com>. Sun Microsystems, 1999.
- [20] Go Corporation, *PenPoint User Interface Design Reference*, Addison-Wesley, 1992.
- [21] *IBM Speech for Java Home Page*. <http://www.alphaworks.ibm.com/formula/speech>. IBM, 1998.
- [22] *Java Speech API Specification 1.0*. Online documentation, <http://java.sun.com>. Sun Microsystems, 1998.
- [23] <http://java.sun.com/products/embeddedjava/>
- [24] <http://java.sun.com/products/personaljava/>
- [25] W. R. Lalonde, J. R. Pugh. *Inside Smalltalk vol. II*. Prentice-Hall International, 1991.
- [26] J. Landay, B. Myers. Extending an Existing User Interface Toolkit to Support Gesture Recognition. In *Proceedings of ACM INTERCHI*. Addison-Wesley, 1993.
- [27] B. Manaris, A. Harkreader. SUITEKeys: A Speech Understanding Interface for the Motor-Control Challenged. In *Proceedings of ACM ASSETS*, pages 108-115. 1998.
- [28] *Microsoft Speech API 4.0*. Online documentation, <http://www.microsoft.com>. Microsoft Corporation, 1998.
- [29] B. A. Myers. The Garnet and Amulet User Interface Development Environments. In *Proceedings of ACM CHI*. Addison-Wesley, 1995.
- [30] L. Nigay and J. Coutaz. A design space for multimodal systems: concurrent processing and data fusion. In *Proceedings of the ACM CHI*, pages 172-178. Addison-Wesley. 1993.
- [31] Dan R. Olsen, Jr. *Developing User Interfaces*. Morgan Kaufmann. 1998.
- [32] PHIGS+ Comittee, A. van Dam, chair. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics*, pages 125-218, 1988.
- [33] J. Prosize. *Programming Windows with MFC*. Microsoft Press, 1999.
- [34] D. Rubine, Specifying Gestures by Example. In *Proceedings of SIGGRAPH*, pages 329-337. 1991.
- [35] Sense8, *WorldToolKit Manual*, 1999.
- [36] H. Sowizral, K. Rushforth, M. Deering. *The Java 3D API Specification (Java Series)*. Addison-Wesley, 1997.
- [37] J. Vlissides, M. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. In *Proceedings of UIST*, pages 158-167. 1989.
- [38] J. Wernecke, Open Inventor Architecture Group. *The Inventor Mentor, Release 2*. Addison-Wesley, 1994.