Comparaison de deux méthodes pour implémenter la Programmation sur Exemple.

Loé SANOU - Patrick GIRARD – Laurent GUITTET

Laboratoire d'Informatique Scientifique et Industrielle (LISI / ENSMA)
Téléport 2 - 1, avenue Clément Ader BP 40109
86961, Futuroscope, France
{sanou, girard, guittet}@ensma.fr

RESUME
Cet article compare deux techniques d'implémentation permettant d'enregistrer et de rejouer les événements utilisateurs dans une application interactive. Une analyse de l'interaction utilisateur a permis de définir les opérations de base à implémenter : l'enregistrement et le rejeu. Les deux techniques d'implémentation utilisées portent l'une sur l'enregistrement systématique des événements produits sur la file système et l'autre sur la spécialisation des composants.

MOTS CLES : Programmation sur Exemple, boîte à outils, interaction homme machine, événements Swing.

ABSTRACT
This paper describes two techniques of implementation that make it possible to record and replay Swing events. An analysis of user's interactions made it possible to define basic operations to implement: recording and replaying. The two proposed techniques lean on systematic recording of system events, and on components specialisation.

CATEGORIES AND SUBJECT DESCRIPTORS: D.2.3 [Software Engineering]: Coding Tools and Techniques.

GENERAL TERMS: Experimentation, Standardization

KEYWORDS: Programming by Demonstration, toolkit, human computer interaction, Swing events.

INTRODUCTION

Un utilisateur peut être amené à exécuter une même tâche un certain nombre de fois. Il paraît judicieux de lui fournir systématiquement un outil lui permettant de satisfaire un tel besoin. La programmation sur exemple (PbD) est une solution à ce problème [1] [2]. Il s'agit de mettre à la disposition de l'utilisateur les moyens de réaliser ses propres programmes à partir du déroulement d'un exemple [1]. Malheureusement, concevoir une application de PbD ou incluant les techniques de PbD s'avère très complexe [3]. Il n'existe pas de système ou de plate-forme proposant aux développeurs les outils pour atteindre ce but avec un minimum d'effort.

Le besoin le plus courant d'un utilisateur en terme de PbD consiste à pouvoir automatiser une tâche répétitive. Cette automatisation passe par l'enregistrement des interactions de l'utilisateur, leur analyse, et leur réexécution [4]. Les fonctionnalités de base d'un tel système consistent en la possibilité d'enregistre et de rejouer actions utilisateur.

Ce travail est un premier pas vers la mise en place d'une boîte à outils dédiée à la programmation sur exemple, intégrée à la bibliothèque Swing de Java. Dans un premier temps, nous dégageons les besoins principaux de l'enregistrement et du rejeu, à partir d'un exemple connu de la littérature. Puis, nous décrivons deux techniques d'implémentation. Enfin, nous terminons par une comparaison de ces deux solutions.

EXPRESSION DES BESOINS

L'un des exemples les plus connus en programmation sur exemple est certainement le système Eager [5]. C'est un système de démonstration des techniques de PbD, qui a pour but de proposer une assistance à l'utilisateur dans sa tâche courante. Développé avec HyperCard, Eager s'appuie sur une application composée d'un mailer et d'un éditeur de texte. L'objectif consiste à automatiser une tâche quelconque faisant intervenir les deux applications. L'exemple proposé par l'auteur consiste à faire une liste numérotée des en-têtes de mails présent dans le mailer. L'analyse de la tâche peut être vue sous deux angles : du coté de l'utilisateur et du côté du programmeur.

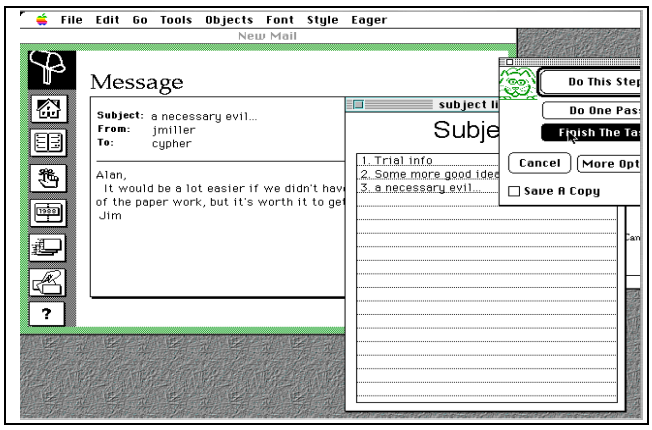

Figure 1 : L'interface du système Eager en phase de finition de la tâche.

Du point de vue de l'utilisateur

La réalisation interactive de la tâche consiste à répéter pour tous les mails présents dans le mailerl a succession des actions utilisateurs suivantes :

- *Mettre le focus sur le champs « Subject »*
- *Selectionner le sujet*
- *Copier le sujet*
- *Mettre le focus sur l'éditeur de texte*
- *Taper le chiffre « 1 »*
- *Taper le point « . »*
- *Coller le texte copié*
- *Faire un retour à la ligne*
- *Mettre le focus sur le mailer.*
- *Cliquer sur le bouton « Suivant » du mailer*

L'utilisateur doit répéter cette séquence sur l'ensemble de ses mails.

Eager apporte une assistance à l'utilisateur dès qu'il détecte une itération. Lorsqu'une séquence de commande est répétée pour la deuxième fois, Eager propose à l'utilisateur les actions qu'il « pense » devoir répéter. L'utilisateur, en confirmant la prédiction d'Eager (*Figure 1*) confirme le programme pas à pas. Lorsque la troisième itération est complètement validée, Eager est en mesure d'effectuer toute la tâche automatiquement.

Du point de vue du programmeur

Le système Eager observe et analyse les interactions de l'utilisateur en continu. En effet, il espionne en permanence le comportement du système et les entrées de l'utilisateur. Comment programmer un tel système ? Une solution naïve consisterait à simplement enregistrer les événements utilisateurs, de manière à pourvoir les analyser et les rejouer. Cela étant, deux types de rejeu doivent être envisagés. Durant la phase de validation de l'interaction, un rejeu très fin doit être produit, avec reproduction de tous les feedbacks, pour permettre à l'utilisateur de valider chaque étape de l'interaction. En revanche, lors de la phase de terminaison de la tâche, seules les action sde haut niveau doivent être rejouées. Pour satisfaire le premier besoin, un travail au niveau des événements les plus élémentaires (*MouseEvent* en Swing) doit être effectué. En revanche, dans le deuxième cas, des événements plus évolués (*ActionEvent*) sont suffisants. Il convient au programmeur d'une solution de PbD de choisir le niveau sur lequel il souhaite intervenir. Notons cependant que l'utilisation d'événement de haut-niveau doit être particulièrement réfléchie. Lorsque l'on conçoit une application Swing, la programmation de la réaction à un événement comme *ActionEvent* (le clic sur un bouton) est explicite. Ce n'st pas toujours le cas ; certains widgets permettent sans programmation explicite des comportements avancés dans les applications WIMP (Windows, Icons, Menus and Pointers) [6]. Par exemple, les widgets « texte » (TextField, TextArea) acceptent en standard le couper-coller, et ce par « Drag and Drop » ou par raccourcis claviers. Il est donc nécessaire, si l'on veut gérer toutes les possibilités d'interaction des widgets évolués, de travailler à un niveau sémantique.

La *figure 2* presente de façon simplifiée l'architecture de fonctionnement d'un système utilisant la PbD.

IMPLEMENTATION

A la base du point de vue du programmeur, nous nous limiterons pour l'exemple aux événements tels que le clic sur un bouton, le déplacement de la souris ou la frappe d'une touche sur le clavier. Ces événements sont des événements directement issur de l'action de l'utilisateur sur un dispositif matériel. L'événement produit est donc un événement souris (*MouseEvent*) ou un événement clavier (*KeyEvent*). La programmation consiste à abonner les composants aux événements que l'on souhaite traiter. Cependant, la notion d'événement est étendue à des actions ne concernant pas le matériel, par exemple la création d'une fenêtre, son apparition à l'écran, etc. Le système de fenêtrage génère alors des événements pour les applications. Ainsi, on trouve par exemple *WindowEvent*, *ComponentEvent*, etc.

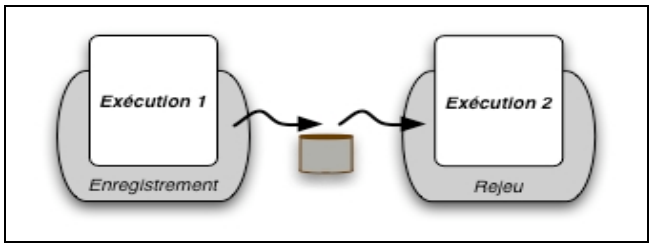

Figure 2 : Une solution d'exécution du rejeu.

Le système gère les événements de manière asynchrone. Il place les événements dans une file d'attente que les applications scrutent en continu. Certaines applications possèdent leur propre file d'attente. Mais l'ensemble des événements transitent par la file du système. Espionner les interactions de l'utilisateur consiste donc à écouter les messages ou événements de la file système. Mais comme les applications peuvent avoir leur propre file d'événements, il est aussi possible de n'écouter que cette

file. Comme le programmeur lui-même définit explicitement les abonnements des différents composants de l'interface utilisateur, il lui est est possible de choisir quels sont ceux qu'il veut soumettre au système de PbD.

Il en décolule deux possibilités d'implémentation : soit par la file des événements système, soit par une file propre à l'application.

Par la file des événements système

Tous les événements transitent par la file des événements avant d'être transmis à l'objet destinataire. En Swing, cette file est la classe *EventQueue* de *java.awt. EventQueue*. Elle ncapsule une machine asynchrone d'expédition des événements extraits à partir de la file d'attente et les achemine en appelant la méthode *dispatchEvent(AWTEvent)* avec l'événement comme argument. Le schéma de cette classe se présente comme suit :

```
public synchronized class java.awt.EventQueue
        extends java.lang.Object {
        //Le constructeur principal
        Public EventQueue()

        //Quelques méthodes : 2/12
        protected void dispatchvent(AWTEvent event)
        public void postEvent(AWTEvent event)
}
```

Pour satisfaire le besoin de rejeu, il faut d'abord que les événements soient sauvegardés. On modifie ainsi la file des événements pour permettre l'enregistrement des événements acheminés. La modification porte essentiellement sur la méhode dispatchEvent qui icoporera une fonction de génération de l'historique. Cette file est définie de la manière suivante :

```
public class PbDEventQueue extends EventQueue {
        //Le constructeur principal
        Public EventQueue(){
                super() ;
                ……..……}
        //une seule méthode rédefinie
        protected void dispatchEvent(AWTEvent event){
                super.dispatchEvent(event) ;
                //Construction de l'historique
                ……..……..……..…… }
}
```

Cette classe constitue la partie centrale du système d'enregistrement. Elle est incorporée à d'autres classes permettant la gestion complète du système avec une interface pour démarrer ou arrêter l'enregistrement, ou pouvoir lancer le rejeu des actions. Une classe historique est mise en place pour contenir les événements. C'est de cette classe que le rejeu est activé. La construction de l'historique consiste simplement à ajouter l'élément événement à la liste. Une partie de l'implémentation de la classe historique et de rejeu est donnée dans les lignes suivantes :

```
public class ModeleVecteur implements Runnable {
        ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
        public void run() {
            for (int i=0; i < this.Size(); i++) {
                synchronized (this) {
                    Thread.yield();
                ,,,,,,,,,,,,,,,,,,,,,,,,
        this.getEventFromVecteur(i).launchEvent();
        ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
}
```

La *figure 3* présente un schéma d'intégration des différents composants des modules d'enregistrement et de rejeu avec une application Java.

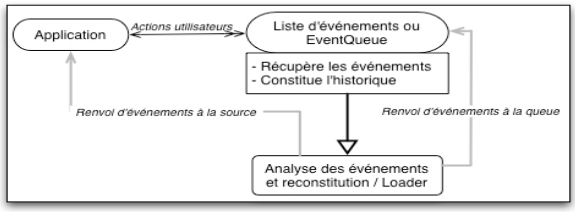

Figure 3 : Architecture du prototype intégrant les modules.

Par une file propre à l'application

En Swing, on écoute les événements à l'aide des « Listener ». Ces « Listener » contiennent chacun des méthodes à implémenter par le développeur. Ces méthodes sont appelées lors de la réception d'un événement. On positionne les « Listener » appropriés sur les widgets subissant les interactions de l'utilisateur. La deuxième solution consiste à détourner ce mécanisme, pour permettre explicitement de choisir quels seront les événements enregistrés. Le rejeu consistera à un renvoi de l'événement à l'objet destinataire par l'objet source. Pour un événement du type « Clic » sur un bouton, nous avons deux possibilités de rejeu. La première possibilité est l'utilisation de la méthode « *doClick()* » de la classe « *AbstractButton* » (*javax.swing*). Son invocation se fait explicitement sur la source de l'événement : *((AbstractButton)ElemDeListEvents.getSource()).doClick()*

La deuxième possibilité est l'utilisation de la méthode *fireActionEvent()*. Cette méthode informe l'ensemble des composants qui ont enregistrés un intérêt pour le type d'événement invoqué. Sa structure en implémentation dans le module de rejeu se décompose en la définition de la méthode elle même *(i)*, la construction de l'événement pour invocation *(ii)*, et l'invocation de la méthode pour le lancement *(iii)*.

```
(i) : public voide fireActionEvent(ActionEvent ae){
        if(listener != null) listener.actioPerformed(ae) ; }
(ii) : public void actionPeformed(ActionEvent ae){
        fireActionEvent(new actionEvent(ae.getSource(),
ae.getID(), ae.getactionCommand(), ae.getWhen(),
ae.getModifiers())) ; }
(iii) : fireActionEvent(new actionEvent(evt.getSource(),
evt.getID(), evt.getactionCommand(), evt.getWhen(),
evt.getModifiers())) ;
```

Discussion

L'implémentation à partir de la file des événements standard prend en compte l'ensemble des événements résultant de l'interaction matériel de l'utilisateur. Ce sont des événements de bas niveau. Elle présente l'avantage de ne demander au programmeur d'application souhaitant incorporer la PbD aucun effort de codage supplémentaire pour effectuer l'enregistrement. Tous les événements de bas-niveau sont effectivement enregistrés. Le premier besoin (enregistrement et rejeu très fin de l'interaction) est ainsi facilement satisfait. Le revers de la médaille réside dans le fait que beaucoup d'événements inutiles (les actions articulatoires de l'utilisateur) sont enregistrées, rendant l'utilisation de l'historique parfois délicat.

Il est parfois souhaitable de ne pas prendre en compte toutes les interactions de bas-niveau, comme dans le cas du deuxième besoin exprimé avec Eager. L'installation d'un filtre avant la construction de l'historique permet de gérer cette situation, mais la mise en place d'un tel filtre est difficile et particulièrement inefficace.

La technique de l'écoute par abonnement fixe dès le depart le niveau des événements à enregistrer. Les événements de haut niveau générés par les composants comme JButton, JTextField, ou encore JTextArea sont traité facilement et leur rejeu donne un rendu visuel complet. Cette technique peut être vue d'un coté comme étant lourde, mais la spécialisation de composants « PbD » susceptibles de travailler de concert avec les composants natifs de Swing autorise de ne prendre en compte que certains événements sans utiliser la notion de filtrage d'événements. C'est une manière élégante de répondre au second besoin. La difficulté de la technique consiste à choisir les bons événements à enregistrer ; en effet, l'existence de widgets évolués encapsulant un comportement automatique (comme le couper/coller décrit ci-dessus) peut aboutir à un comportement incohérent de l'enregistrement.

CONCLUSION

Nous avons comparé dans cet contribution deux techniques pour implémenter les composants élémentaires à l'implémentation d'un système incluant des facilités de PbD. Ces deux techniques répondent à des besoins différents, qui peuvent être exprimés ensemble dans une même application. Il convient donc de les rendre utilisables ensemble. C'est la prochaine étape de notre étude. Celle-ci doit nous conduire à proposer un framework de PbD permettant de mettre cete dernière à la portée de tout programmeur de système interactif.

La réflexion sur l'aspect méthodologique ainsi que la création d'outils de vérification de la cohérence de la mise en œuvre de la PbD seront les étapes suivantes de notre étude.

BIBLIOGRAPHIE

1. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. 1993, The MIT Press: Cambridge, Massachusetts. 604.
2. Lieberman, H., *Your Wish is my command*. 2001: Morgan Kaufmann. 416.
3. Cypher, A., D.S. Kosbie, and D. Maulsby, *Characterizing PBD Systems*, in *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor. 1993, The MIT Press: Cambridge, Massachusetts. p. 467-484.
4. Girard, P., *Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur*, in *LISI/ENSMA*. 2000, Université de Poitiers: Poitiers. p. 92.
5. Cypher, A., *Eager : Programming Repetitive Tasks by Demonstration*, in *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor. 1993, The MIT Press: Cambridge, Massachusetts. p. 205-217.
6. Curtis, B. and B. Hefley, *A Wimp No More: The Maturing of User Interface Engineering*. Interactions, 1994: p. 23-34.