

Introduction à l'Interaction Homme-Machine

RICM4

2009-2010

Renaud Blanch

IIHM - LIG - UJF

<mailto:renaud.blanch@imag.fr>

<http://iihm.imag.fr/blanch/>

Remerciements

François Bérard

(LIG - IIHM, INPG)

Éric Lecolinet

(ENST-GET)

Alan Dix

(Université de Lancaster)

1. Principes pour le développement de logiciel interactif

1.0 Éléments d'architecture

1.1 Principes de réalisation

1.0 Éléments d'architecture

Séparation de l'interface

Principe n°1

Séparer strictement
le **noyau fonctionnel** et
l'**interface**.

Noyau fonctionnel vs. Interface

noyau fonctionnel :

Services offerts par le système,
indépendamment de la façon dont l'utilisateur
s'en sert.

le “quoi”

Noyau fonctionnel vs. Interface

interface :

Parties du système

dépendantes des périphériques

grâce auxquels l'utilisateur se sert du système.

le “**comment**”

Noyau fonctionnel vs. Interface

séparation :

- des **préoccupations**
- des **expertises**
- des **méthodes**

Principe n°2

Le code du noyau fonctionnel
est **indépendant**
du code de l'interface.

Indépendance du noyau fonctionnel

bonne pratique : **minimiser les dépendances**

L'**interface** risquant d'**évoluer** en cours de réalisation, il faut mieux que le **noyau fonctionnel** soit conçu **sans dépendance** envers celle-ci.

Principe n°3

Le noyau fonctionnel doit offrir des services nécessaires à l'interaction :

- la **notification** ;
- la **prévention des erreurs** ; et
- l'**annulation**.

Notification

“possibilité pour un **module externe**
d’être **prévenu** lorsque
l’état du noyau sémantique **change.**”

Prévention des erreurs

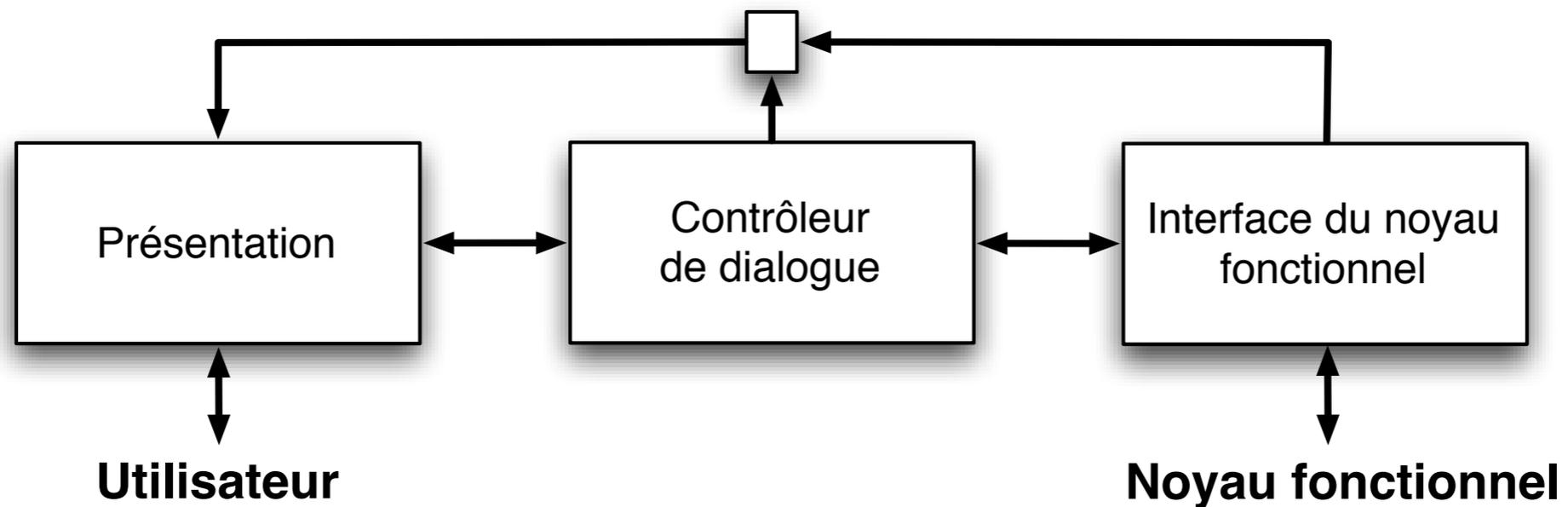
“possibilité de **savoir** si
un **appel** de fonction est **licite**
dans un **contexte**.”

Annulation

“possibilité de **revenir**
à des **états précédants**
du noyau sémantique.”

Modèle séminal de l'interface

Le modèle de **Seeheim** (1983) propose de séparer l'**interface** en trois niveaux.



Modèle séminal de l'interface

Le modèle de **Seeheim** (1983) propose de séparer l'**interface** en trois niveaux :

- la **présentation**
- le **contrôleur de dialogue**
- l'**interface du noyau fonctionnel**

Modèle séminal de l'interface

Le modèle de **Seeheim** (1983) propose de séparer l'**interface** en trois niveaux :

- la **présentation**
- le **contrôleur de dialogue**
- l'**interface du noyau fonctionnel**

correspondant à trois niveaux d'abstraction :
lexical, syntaxique, et sémantique.

1.1 Principes de réalisation

Communication

pour **modifier** l'état du noyau fonctionnel,

pour **notifier** l'interface :

- les **appels de fonction**
- les **fonctions de rappel** (*callbacks*)
- les **variables actives**

Appel de fonction

ne pas faire :

```
namespace nf {  
    name = ihm::entry.getText();  
}
```

Appel de fonction

ne pas faire :

```
namespace nf {  
    name = ihm::entry.getText();  
}
```

mais faire :

```
namespace nf {  
    void setName(string new_name) { name = new_name; }  
}
```

```
namespace ihm {  
    nf::setName(entry.getText());  
}
```

Appel de fonction

ne pas faire :

```
namespace nf {  
    ihm::label.setText(name);  
}
```

Appel de fonction

ne pas faire :

```
namespace nf {  
    ihm::label.setText(name);  
}
```

mais faire :

```
namespace nf {  
    string getName() { return name; }  
}
```

```
namespace ihm {  
    label.setText(nf::getName());  
}
```

Fonctions de rappel

ne pas faire :

```
namespace nf {  
    void name_changed() {  
        ihm::label.setName(name);  
    }  
}
```

Fonctions de rappel

mais faire :

```
namespace nf {
    typedef void callback(string);
    void register_name_notifier(callback *notify) {
        name_notifiers.append(notify);
    }
    void name_changed() {
        for(notify in name_notifiers) { (*notify)(name); }
    }
}

namespace ihm {
    nf::callback name_notify;
    nf::register_name_notifier(&name_notify);
    void name_notify(string name) { label.setText(name); }
}
```

Variables actives

notification quand le **contenu**
d'une variable est **modifié**

Variables actives

en **tcl**, instrumentation du code :

```
proc trace_i { args } {  
    # do something ...  
}  
trace add variable i write trace_i  
  
set i 12
```

Variables actives

en **python**, utilisation des *properties* :

```
class Example(object):
    def get_i(self):
        return self._i
    def set_i(self, value):
        if self._i != value:
            self._i = value
            # do something ...
    i = property(get_i, set_i)
```

```
example = Example()
example.i = 12
```

Variables actives

en c++ avec **Qt**, utilisation des *signals/slots* :

```
class Example : public QObject {
    Q_OBJECT
    int _i;
signals:
    void valueChanged(int i);
public slots:
    void setValue(int i) {
        if(_i != i) {
            _i = i;
            emit valueChanged(i);
        }
    }
};
```

Variables actives

en **java**, utilisation des *getters* et des *setters* :

```
class Example {
    private Object var;

    public Object getVar() {
        return var;
    }

    public void setVar(value) {
        if(var != value) {
            var = value;
            // do something
        }
    }
};
```

Variables actives

en **java**, utilisation des *properties* avec les *beans* :

```
import java.beans.PropertyChangeSupport;

class Example {
    private final PropertyChangeSupport pcs =
        new PropertyChangeSupport(this);

    ...
    public void setVar(value) {
        Object old_var = var;
        var = value;
        pcs.firePropertyChange("var", old_var, var);
    }
    ...
}
```

Variables actives

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

class Example {
    ...
    public void addPropertyChangeListener(
        PropertyChangeListener listener) {
        pcs.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(
        PropertyChangeListener listener) {
        pcs.removePropertyChangeListener(listener);
    }
};
```

Variables actives

```
import java.beans.PropertyChangeListener;  
import java.beans.PropertyChangeEvent;  
  
class Listener implements PropertyChangeListener {  
    public void propertyChange(PropertyChangeEvent e) {  
        String propName = e.getPropertyName();  
        if(propName.equals("var")) {  
            System.out.println("old: " + e.getOldValue());  
            System.out.println("new: " + e.getNewValue());  
        }  
    }  
};  
  
void test() {  
    Example example = new Example();  
    example.addPropertyChangeListener(new Listener());  
}
```

Événements

changement de paradigme :

L'utilisateur n'est plus au service du programme
mais le **programme au service de l'utilisateur.**

Programme classique

L'utilisateur fait ce que lui demande l'application.

```
int main() {  
    string name;  
    // ...  
    cout << "name = "; // user prompt  
    cin >> name;       // waiting user input  
    // ...  
}
```

Programme par événements

L'utilisateur a le contrôle.

```
int main() {  
    // ...  
    while(not quit) {  
        event = getNextEvent();  
        process(event);  
    }  
}
```

Structure d'un événement

Structure envoyée à l'**application**
à **chaque action élémentaire** de l'utilisateur
contenant des **informations** sur l'événement
et le **contexte** de son émission :

- **type** de l'événement,
- ***timestamp***,
- **position** du curseur,
- **état** des boutons de la souris,
- **identifiant** de la fenêtre (où se situe le curseur),
- ...

Types d'événements

quelques **types** d'événements (XWindow) :

- KeyPress, KeyRelease,
- ButtonPress, ButtonRelease, MotionNotify,
- EnterNotify, LeaveNotify,
- FocusIn,
- DestroyNotify, MapNotify, UnmapNotify, Expose,
- ClientMessage,
- ...

Types d'événements

quelques **types** d'événements (Win32) :

- WM_ACTIVATEAPP, WM_CLOSE,
- WM_CREATE, WM_DESTROY,
- WM_MOVE, WM_MOVING,
- WM_SIZE, WM_SIZING,
- WM_SHOWWINDOW,
- WM_KEYDOWN, WM_KEYUP,
- WM_MOUSEMOVE,
- WM_LBUTTONDOWN, WM_LBUTTONUP,
- ...

Types d'événements

quelques **types** d'événements (Mac OS X/carbon) :

- kEventWindowClosed,
kEventWindowBoundsChanged,
- kEventMouseMoved, kEventMouseDragged,
- kEventMouseDown, kEventMouseUp
- ...

Types d'événements

quelques **types** d'événements (Java/AWT) :

- `java.awt.event.MouseEvent`,
- `java.awt.event.KeyEvent`,
- `java.awt.event.WindowEvent`,
- ...

Gestion des événements

- 1) phase d'initialisation
- 2) boucle de traitement des événements

Boucle *REPL*

“*read-eval-print loop*” (*REPL*)

à la charge du programmeur

```
while(not quit) {
    event = getNextEvent();
    switch(event->type) {
    case TYPE_1:
        process_type_1(event->>window, event->position);
        break;
    case TYPE_2:
        // ...
    }
}
```

Boucle *REPL*

le cas de :

- XWindow
- Win32
- Mac OS X

Boucle *REPL*

- **complexe à mettre au point**
il faut prendre en compte toutes les combinaisons d'événements (spatiales et temporelles)
- introduit des **dépendances implicites**
- risque de **comportements non-standards**

Dispatcher

aiguilleur (*event dispatcher*) fourni par le système

```
void handle_quit() { exit(0); }

int main() {
    Menu menu = new Menu();
    menu.addAction("Quit", handle_quit);

    // ...

    return main_loop();
}
```

Dispatcher

le cas de :

- Xt/Motif
- Java AWT, SWING, ...
- GLUT
- ...

Dispatcher

Les protocoles d'interaction sont “**embarqués**” par les objets interactifs.

Cela **résout** les problèmes de la *REPL* mais :

- **cache** le mécanisme de *callback*,
les fonctions sont appelées dans une boucle aussi
ce qui peut **nuire à l'interactivité**.

Dispatcher

L'aiguilleur se construit au-dessus de la REPL :

Mac OS X (Carbon)

```
EventRef event;
while(ReceiveNextEvent(0, 0, kEventDurationForever,
                      true, &event) == noErr) {
    SendEventToEventTarget(event, GetEventDispatcherTarget());
    ReleaseEvent(event);
}
```

Win32

```
MSG msg;
while(GetMessage(&msg, 0, 0, 0) != 0) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Interactivité

La solution pour les **traitements longs** :

- un **processus** (léger) dédié,
- qui **notifie** lorsqu'il se termine.

Interactivité

La solution pour les **traitements longs** :

- un **processus** (léger) dédié,
- qui **notifie** lorsqu'il se termine.

Qui crée de nouveaux problèmes :

- de **synchronisation** des structures de données,
- d'**accès concurrents** aux ressources
(en particulier **graphiques**).

Communication avec le noyau fonctionnel

**Utilisation d'événements particuliers
non graphiques.**

Machines à états

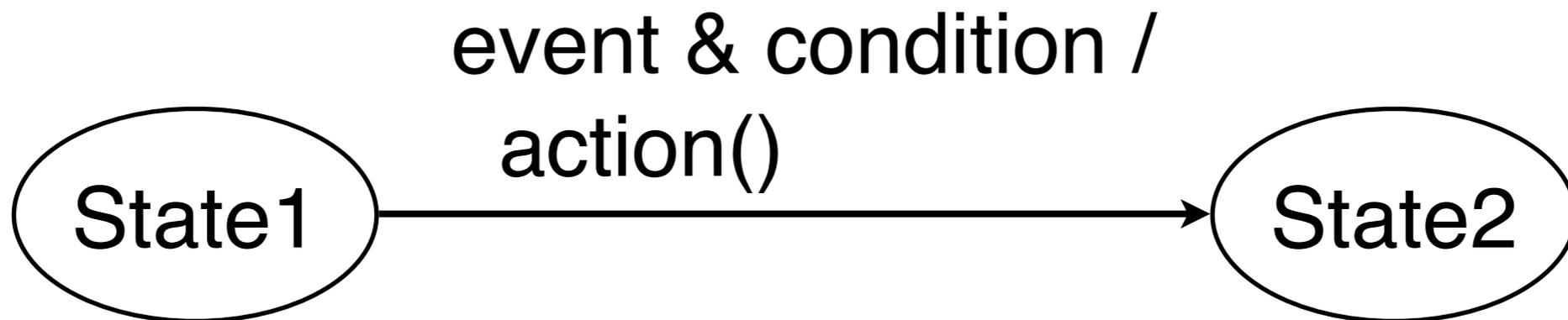
Pour faire face à l'**explosion combinatoire** due aux **types** possibles d'événements et à l'**ordre** dans lequel ils arrivent, il faut adopter une **démarche systématique**.

Machines à états

Automates déterministes finis
(ensemble d'**états** et de **transitions**)

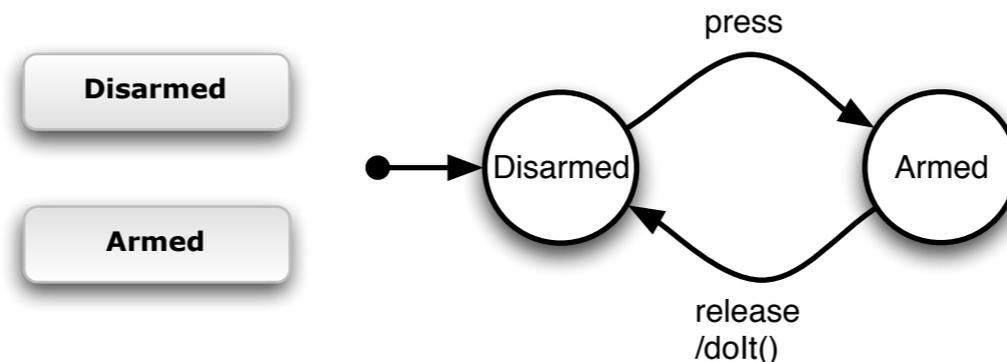
étendus par :

- des **conditions**,
- des **actions**.



Machines à états

Exemple : le bouton



En fait, une grande partie du comportement n'est pas décrit.

exercices :

- prendre en compte les événements ***enter*** et ***leave***,
- ajouter les événements ***activate***, ***desactivate***.

Machines à états

Exemple : le **clic** et le **drag**

Modes

exemple du **multiplexage** du *clic* et du *drag*.
avec des **événements simplifiés** contenant :

- leur type (MOUSE_MOVE,
BUTTON_DOWN, BUTTON_UP)
 - la position du curseur
- gérés** par un aiguilleur.

Modes

```
enum State { UP, CLIC_OR_DRAG, DRAG };
State state = UP;
Position p0;

void handle_BUTTON_DOWN(p) {
    switch(state) {
    case UP:
        p0 = p;
        state = CLIC_OR_DRAG;
        break;
    default:
        assert(false);
    }
}
```

Modes

```
void handle_MOUSE_MOTION(p1) {
    switch(state) {
    case UP:
        break;
    case CLIC_OR_DRAG:
        if(not p1 - p0 > D_DRAG) return;
        start_drag(p0);
        state = DRAG;
    case DRAG:
        drag(p1 - p0);
        p0 = p1;
        break;
    default:
        assert(false);
    }
}
```

Modes

```
void handle_BUTTON_UP(p) {
    switch(state) {
    case CLIC_OR_DRAG:
        clic(p0);
        sate = UP;
        break;
    case DRAG:
        finish_drag(p);
        state = UP;
        break;
    default:
        assert(false);
    }
}
```

Adaptation du langage

Les **langages impératifs** sont **peu adaptés** à ce type de programmation car l'**état** est codé dans des **variables globales**.

Adaptation du langage

Les **langages impératifs** sont **peu adaptés** à ce type de programmation car **l'état** est codé dans des **variables globales**.

Les **langages à objets** permettent de rendre **l'état local**.

Adaptation du langage

Les **langages impératifs** sont **peu adaptés** à ce type de programmation car **l'état** est codé dans des **variables globales**.

Les **langages à objets** permettent de rendre **l'état local**.

Cependant un **formalisme adapté** permettant de

- **spécifier, vérifier,**
- **générer, exécuter**

les **interactions** serait le bienvenu.