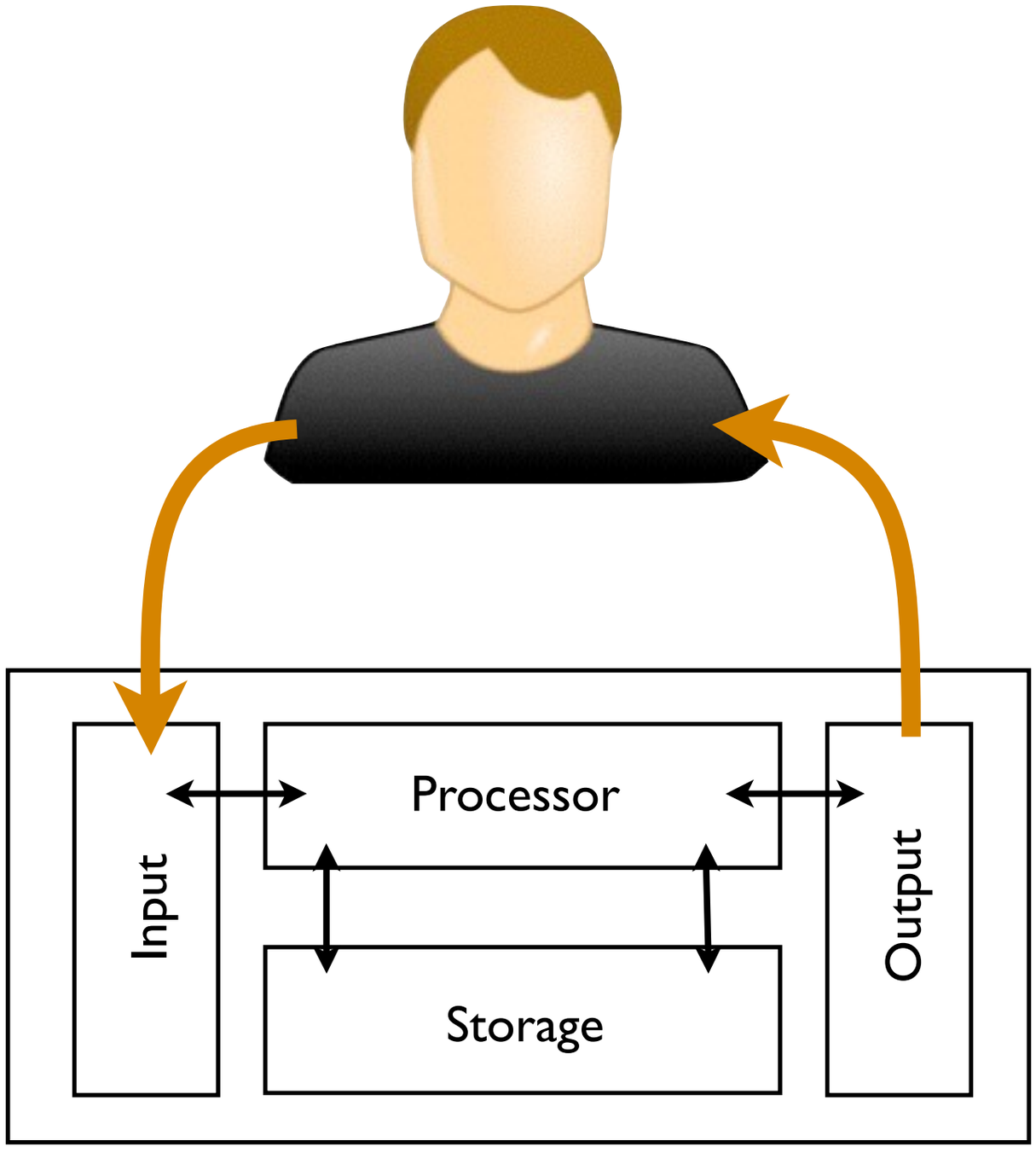
Commands, actions et Raccourcis clavier

Sylvain Malacria http://www.malacria.fr

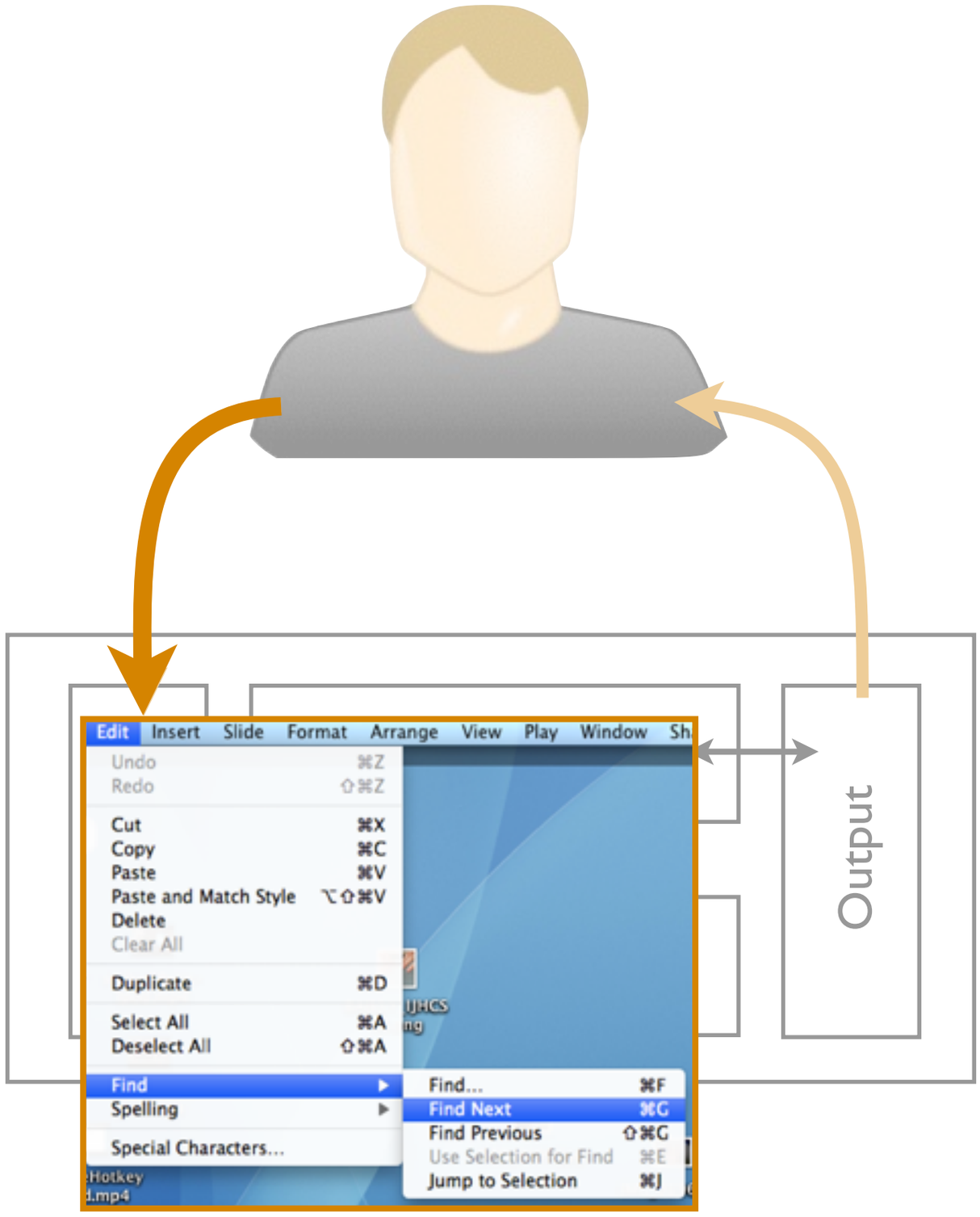
IHM Master 1 informatique – Université de Lille 1

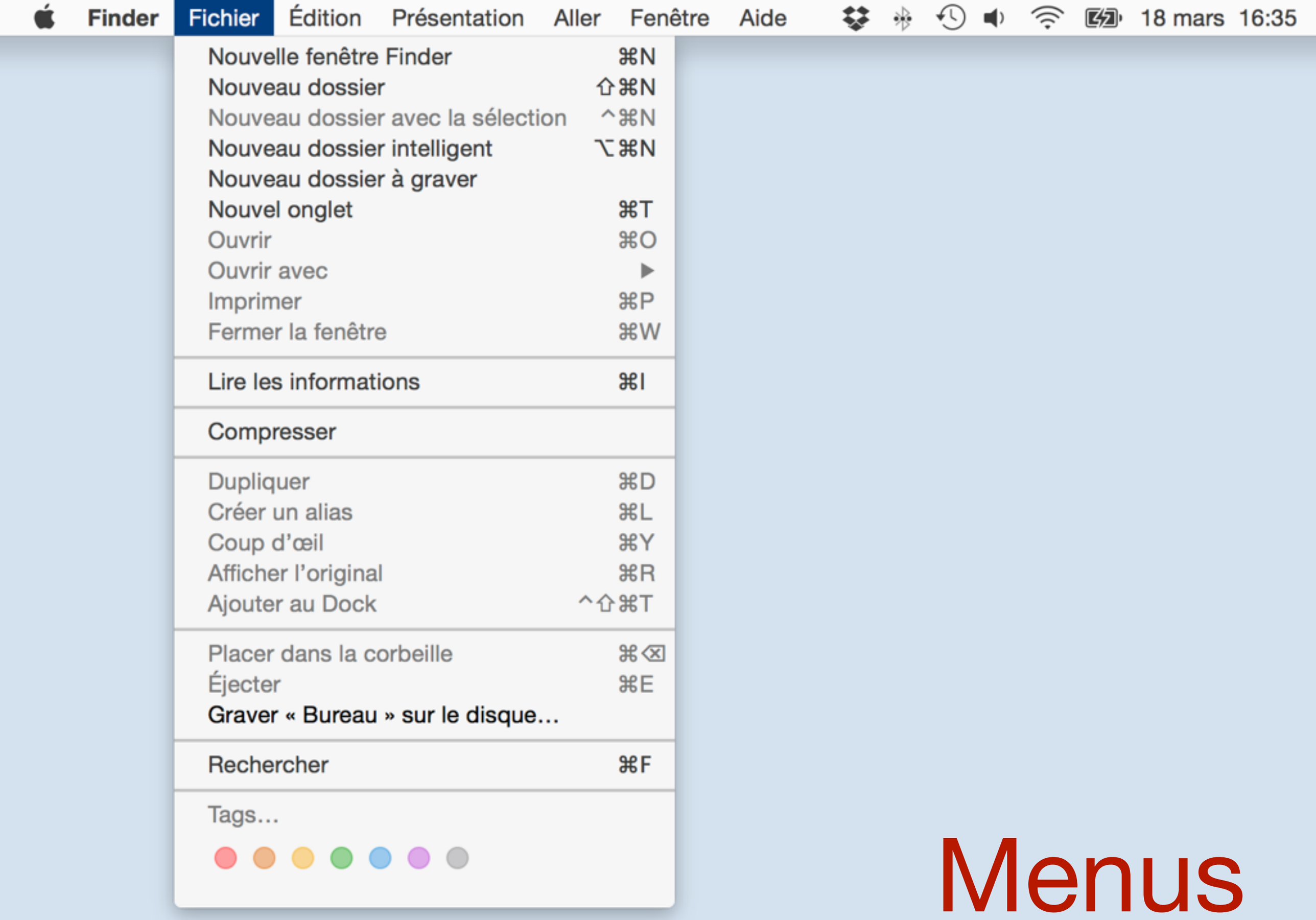
Adapté de Géry Casiez

Systèmes interactifs



Sélection de commandes



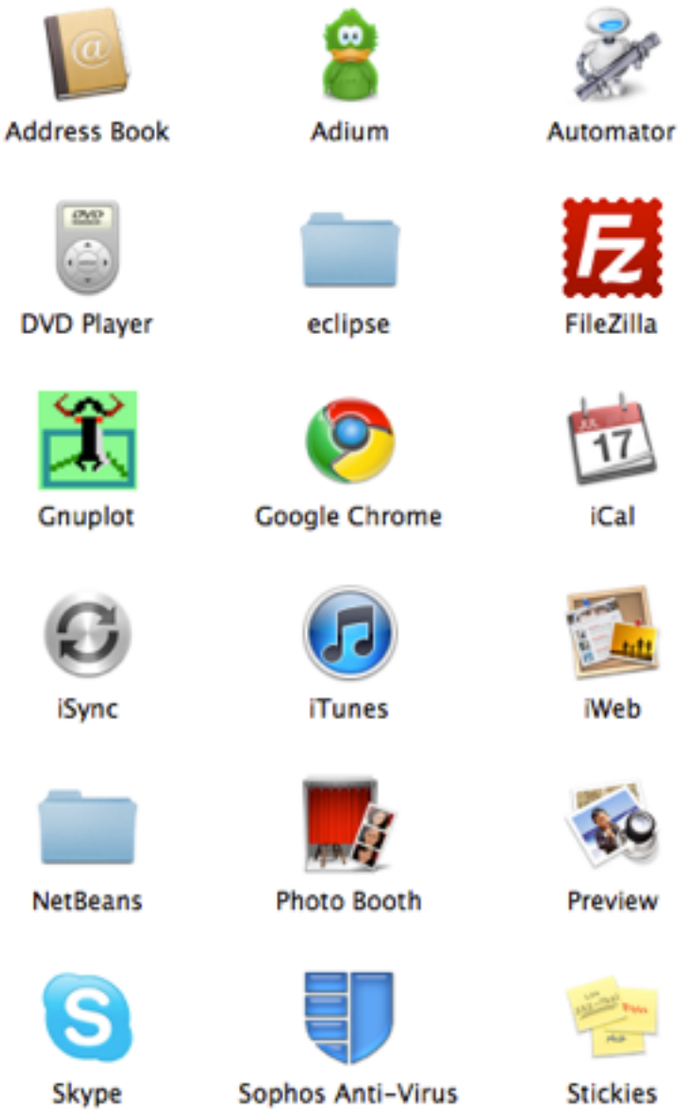

Finder
Fichier
Édition
Présentation
Aller
Fenêtre
Aide
18 mars 16:35
Nouvelle fenêtre Finder ⌘N
Nouveau dossier ⇧⌘N
Nouveau dossier avec la sélection ^⌘N
Nouveau dossier intelligent ⌥⌘N
Nouveau dossier à graver
Nouvel onglet ⌘T
Ouvrir ⌘O
Ouvrir avec ▶
Imprimer ⌘P
Fermer la fenêtre ⌘W
Lire les informations ⌘I
Compresser
Dupliquer ⌘D
Créer un alias ⌘L
Coup d'œil ⌘Y
Afficher l'original ⌘R
Ajouter au Dock ^⇧⌘T
Placer dans la corbeille ⌘⌫
Éjecter ⌘E
Graver « Bureau » sur le disque…
Rechercher ⌘F
Tags…

Menus

Pourquoi est-ce important?

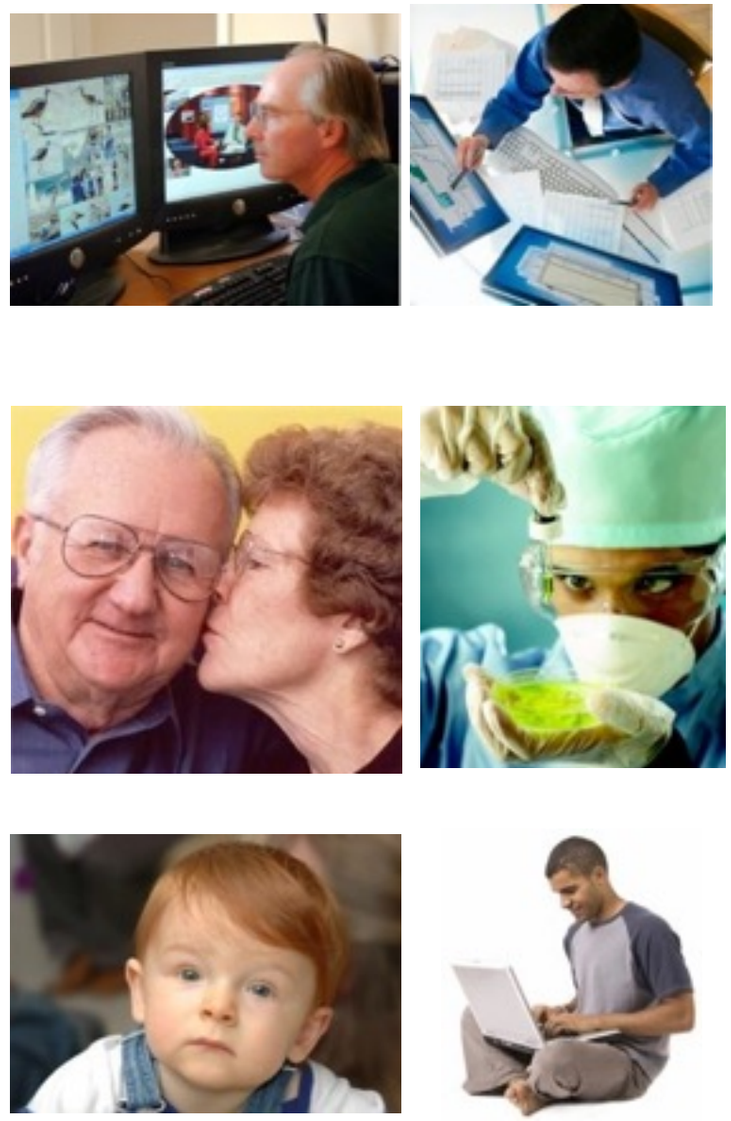
1.Ubiquitaire

Présent dans toutes les **applications**, sur toutes les **plateformes** et utilisé par tous les **utilisateurs**.

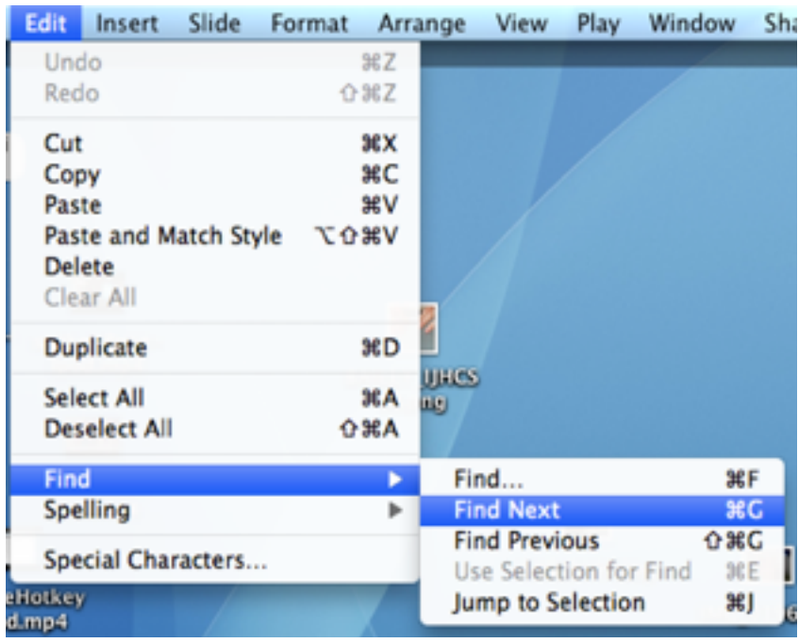


Application

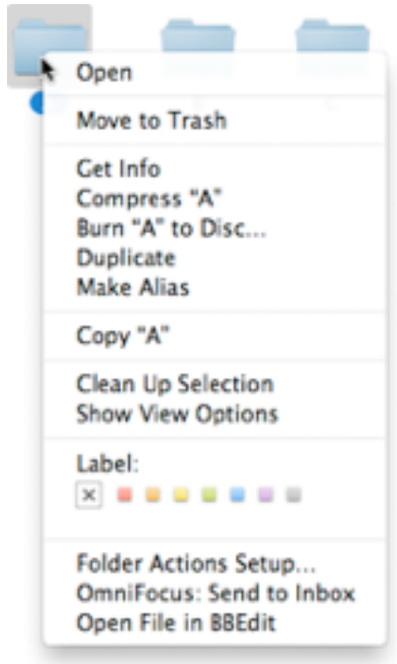
Plateforme



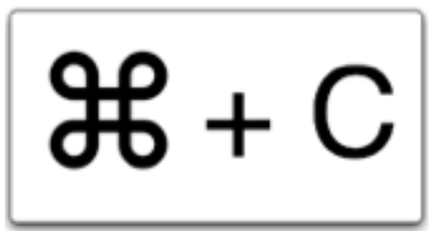
Utilisateur



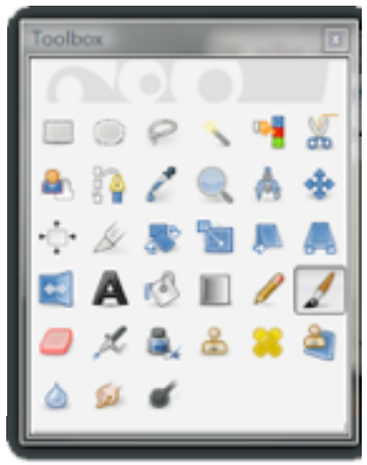
Barre de menus



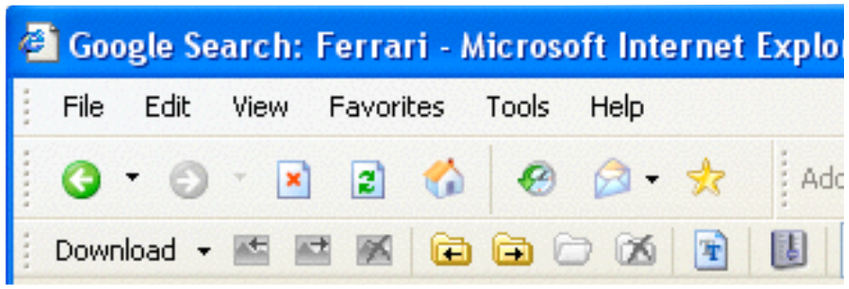
Menu contextuel



Raccourci clavier

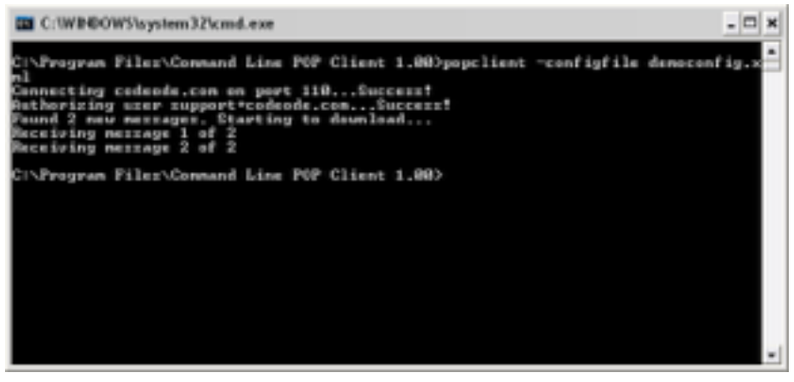


Boite à outils



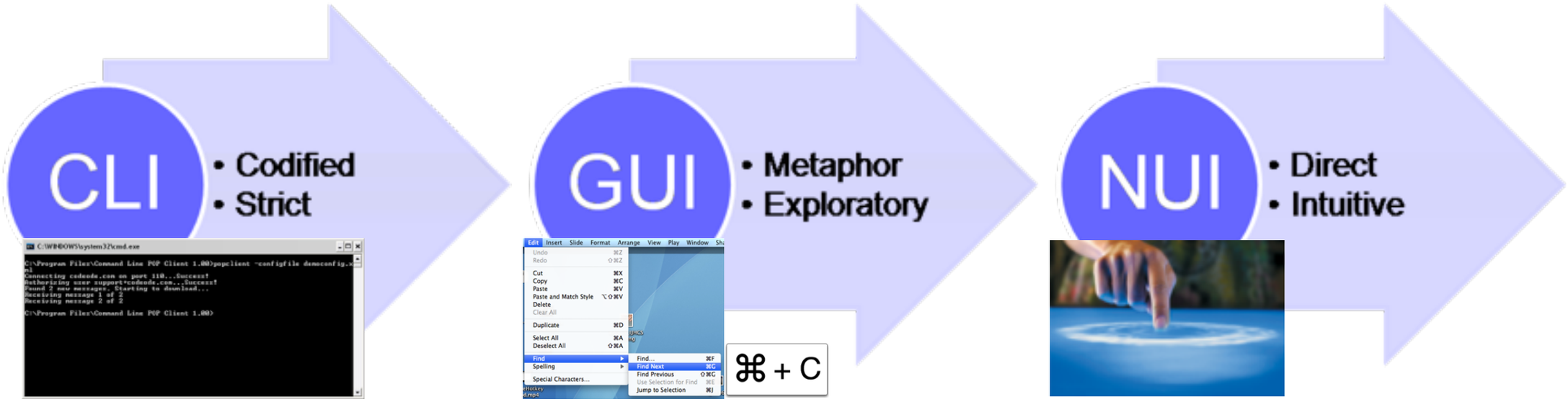
Barre d'outils

Ruban



Ligne de commande

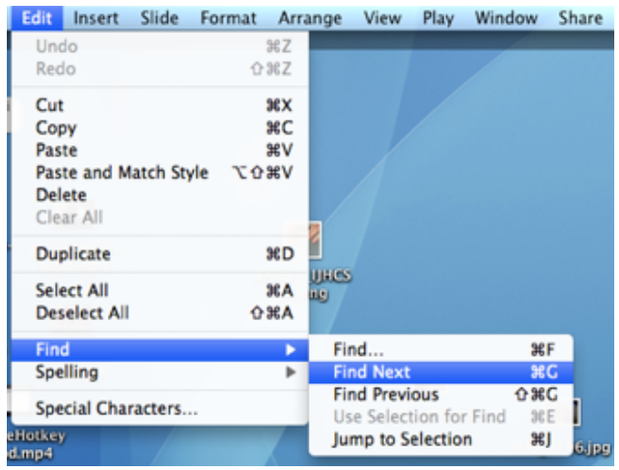
2.Touche tous les paradigmes d'interaction



Il est important de:

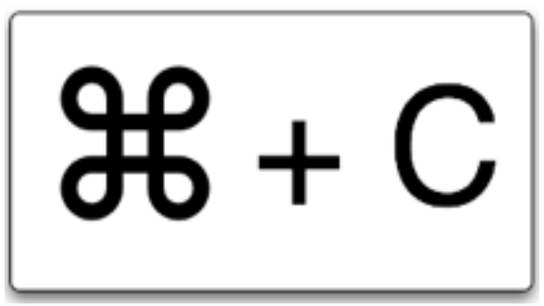
- **presenter**, **organiser** les commandes
- permettre aux utilisateurs de les **sélectionner**

3. Peut avoir un impact considérable



Technique 1

VS.



Technique 2

Un peu de calcul …

0.5 s

500 million d'utilisateurs

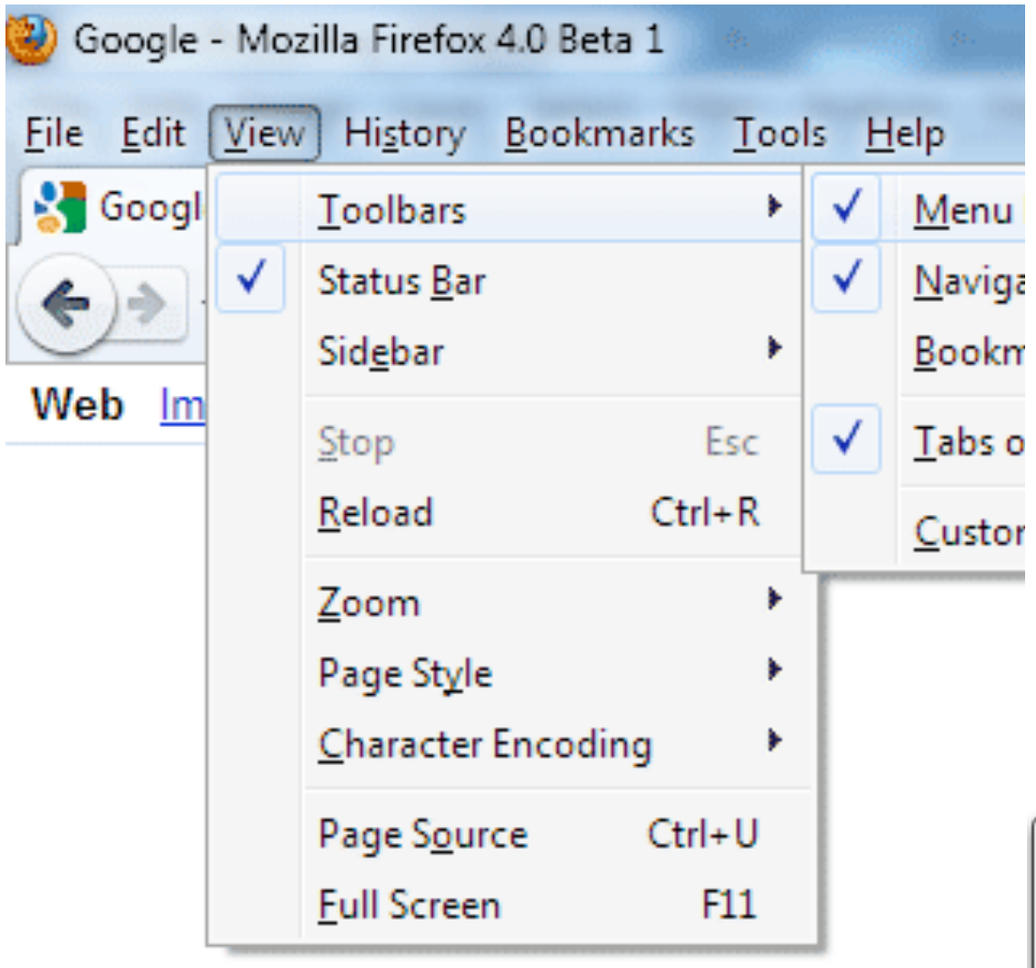
5 commandes par utilisateur par jour

= 465 milliards de secondes par années

= 126 millions d'heures

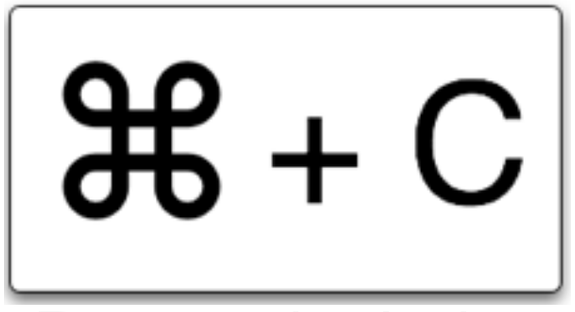
= 14,400 années

Brainstorming (30s): Pour et contre

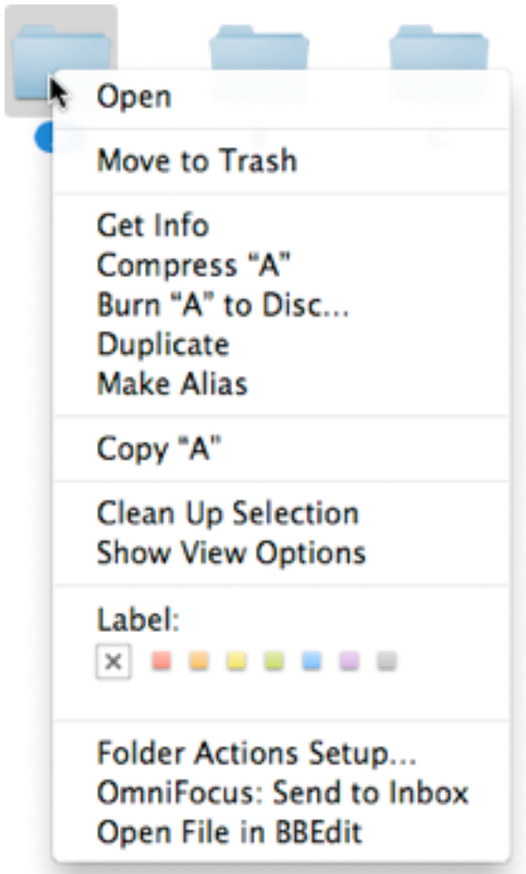


Barre de menu

Toolbox

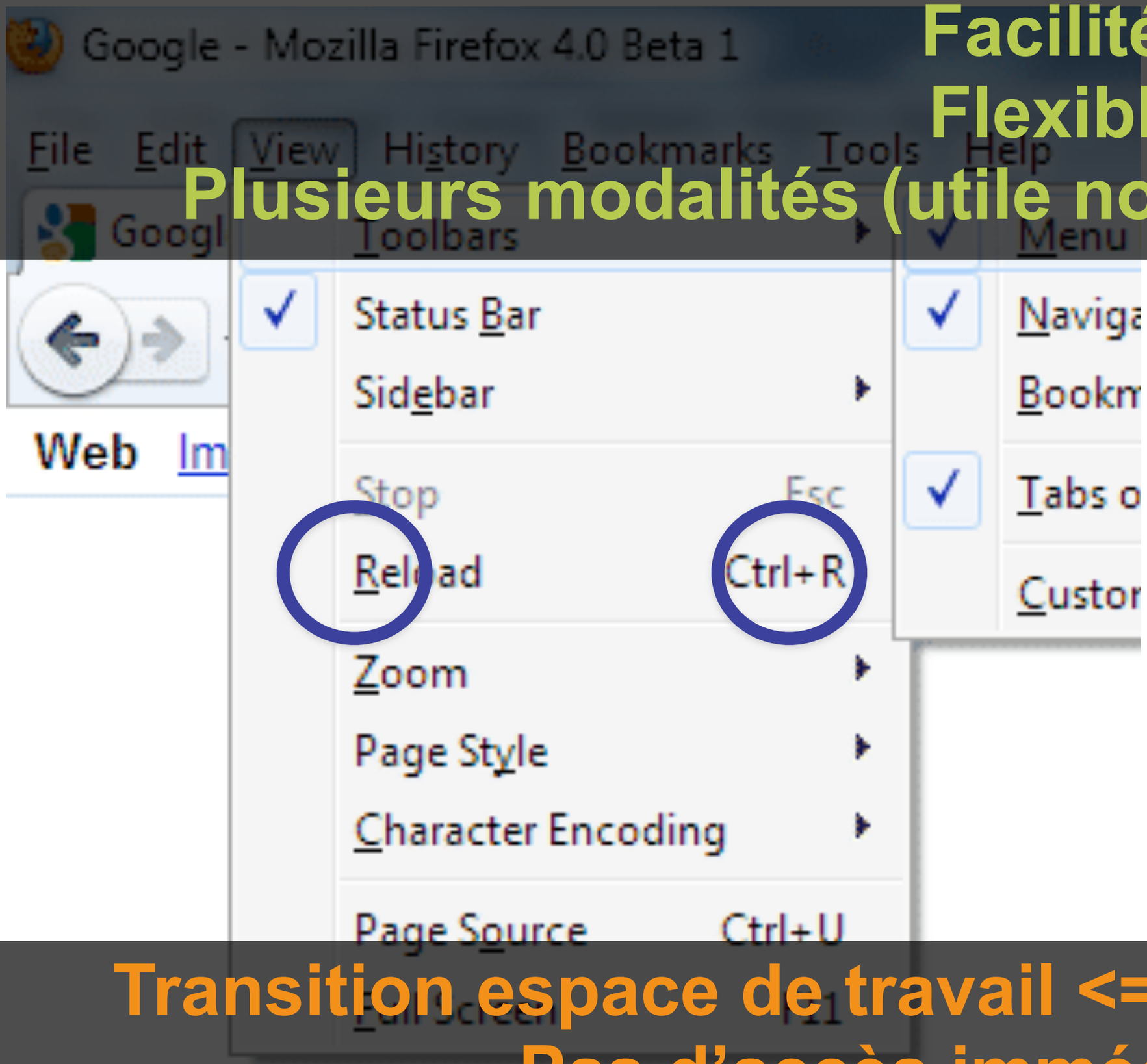


Raccourcis clavier

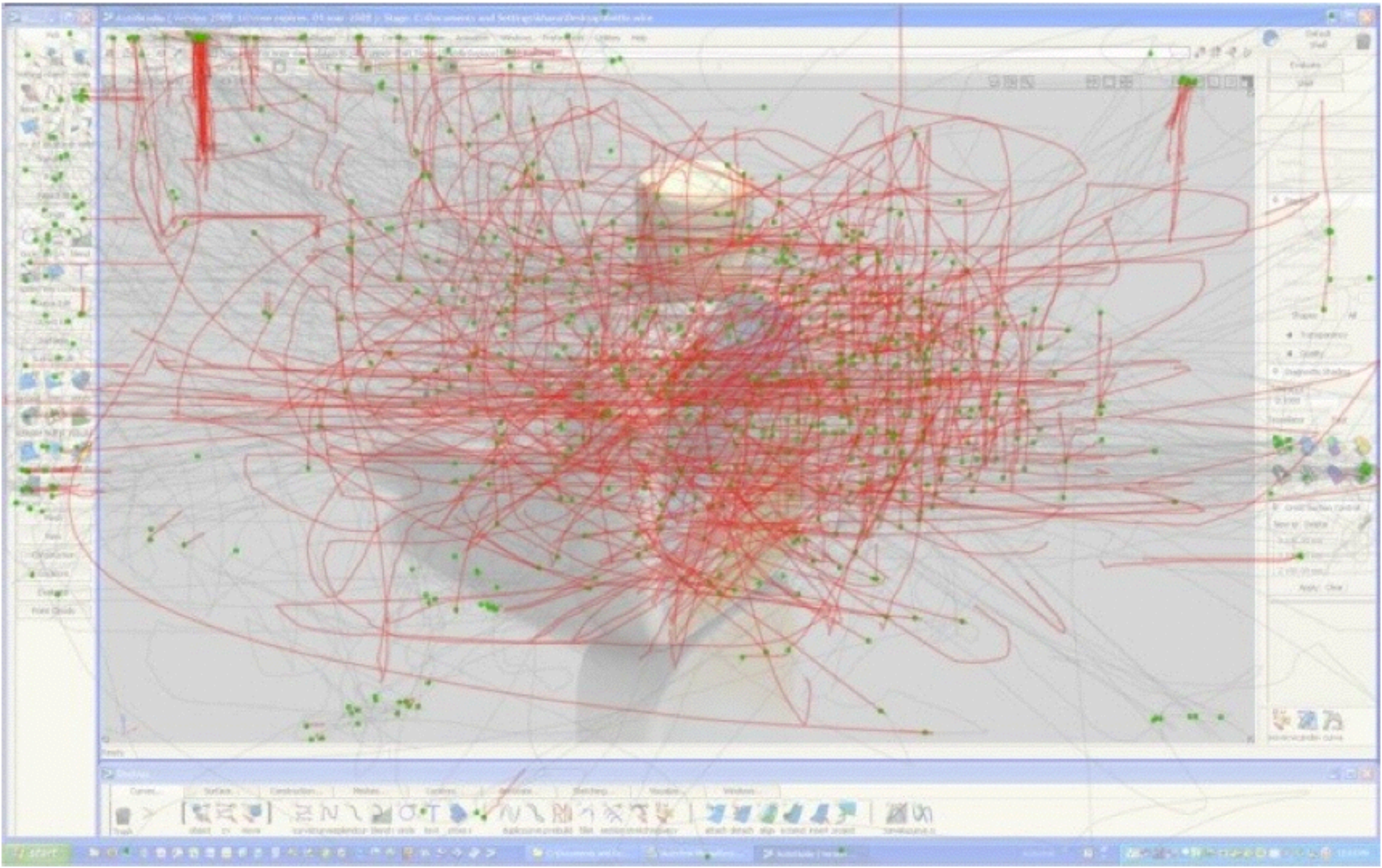


Menu contextuel

Organisation hiérarchique
Facilité d'exploration
Flexible (taille écran)
Plusieurs modalités (utile novices/experts)



Transition espace de travail <=> commandes
Pas d'accès immédiat à la souris



Toujours visible
Commandes modales

Transition espace de travail <=> commandes

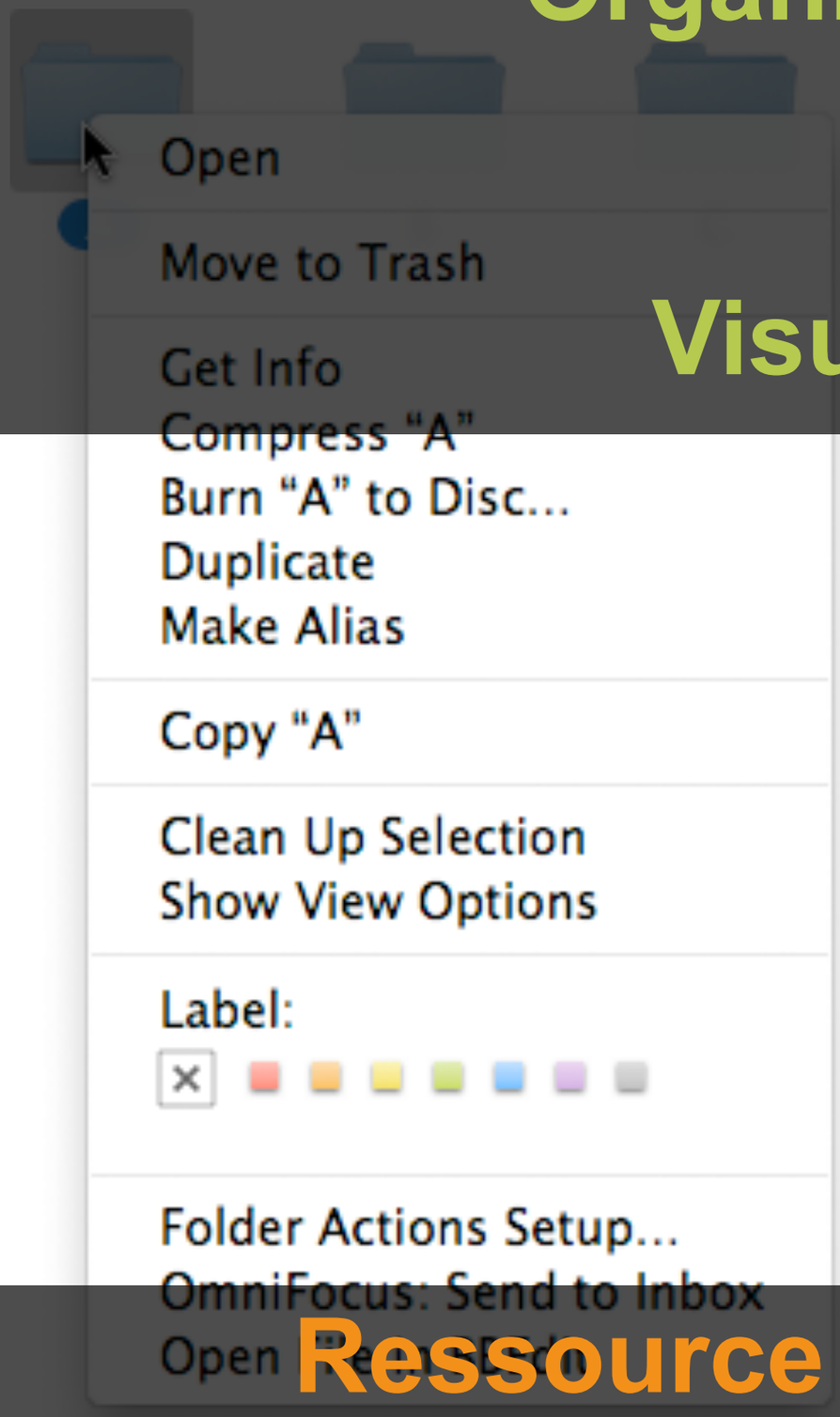
Petite cibles

Pas de label textuel

Occultation possible + occupe espace écran

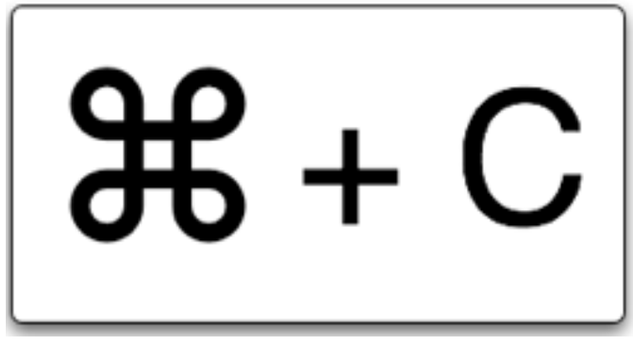
Pas de catégorisation (généralement)

Organisation hiérarchique
En place
Contextuel
Visualisation transiente



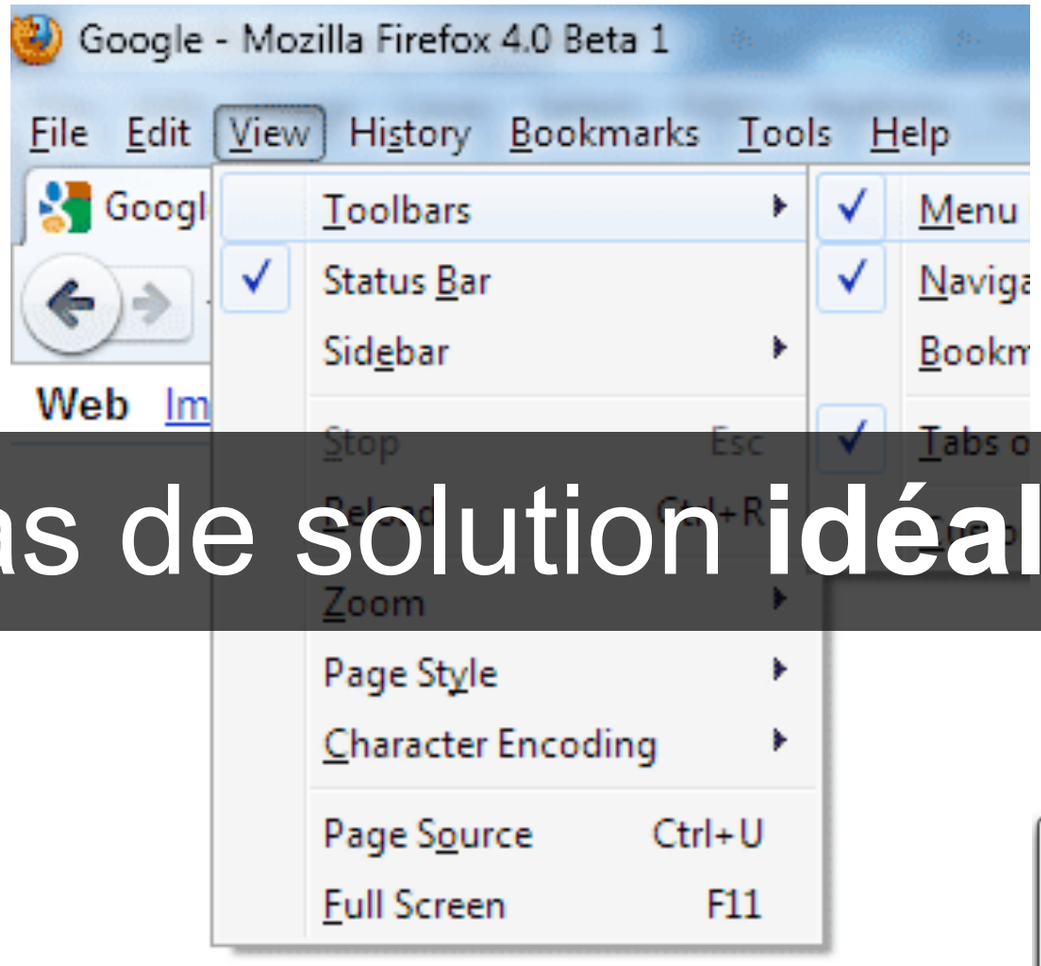
Ressource dédiée pour activer TouchScreen?

Accès direct et rapide
Main gauche
Pas de transition Clavier<->Souris nécessaire

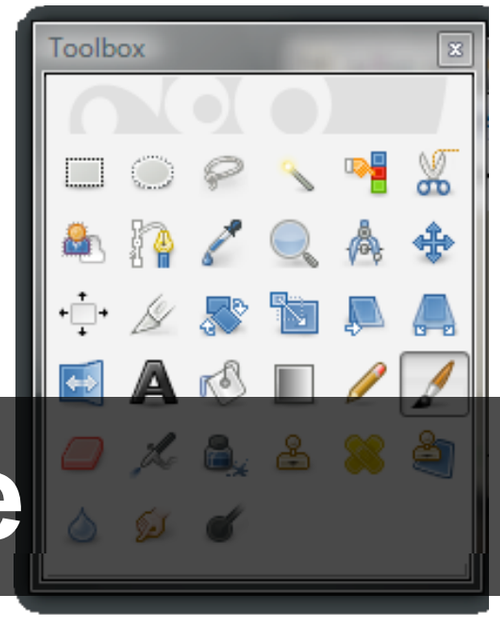


Mémorisation préalable
Collisions & Associations arbitraires
Coordination des doigts
Besoin d'un clavier (TouchScreen?)

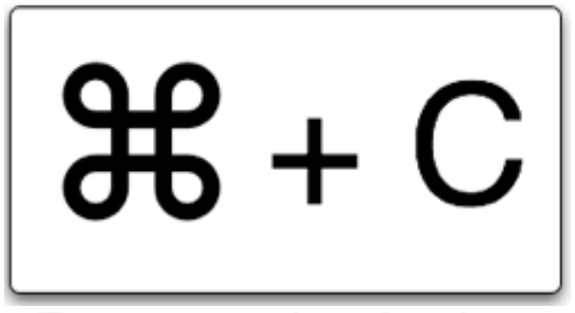
Pas de solution **idéale**



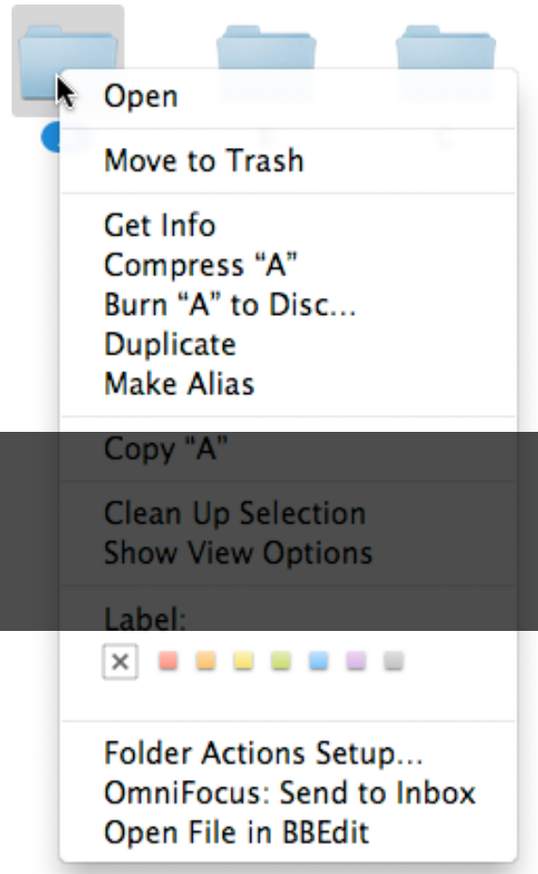
Barre de menu



Toolbox



Raccourcis clavier

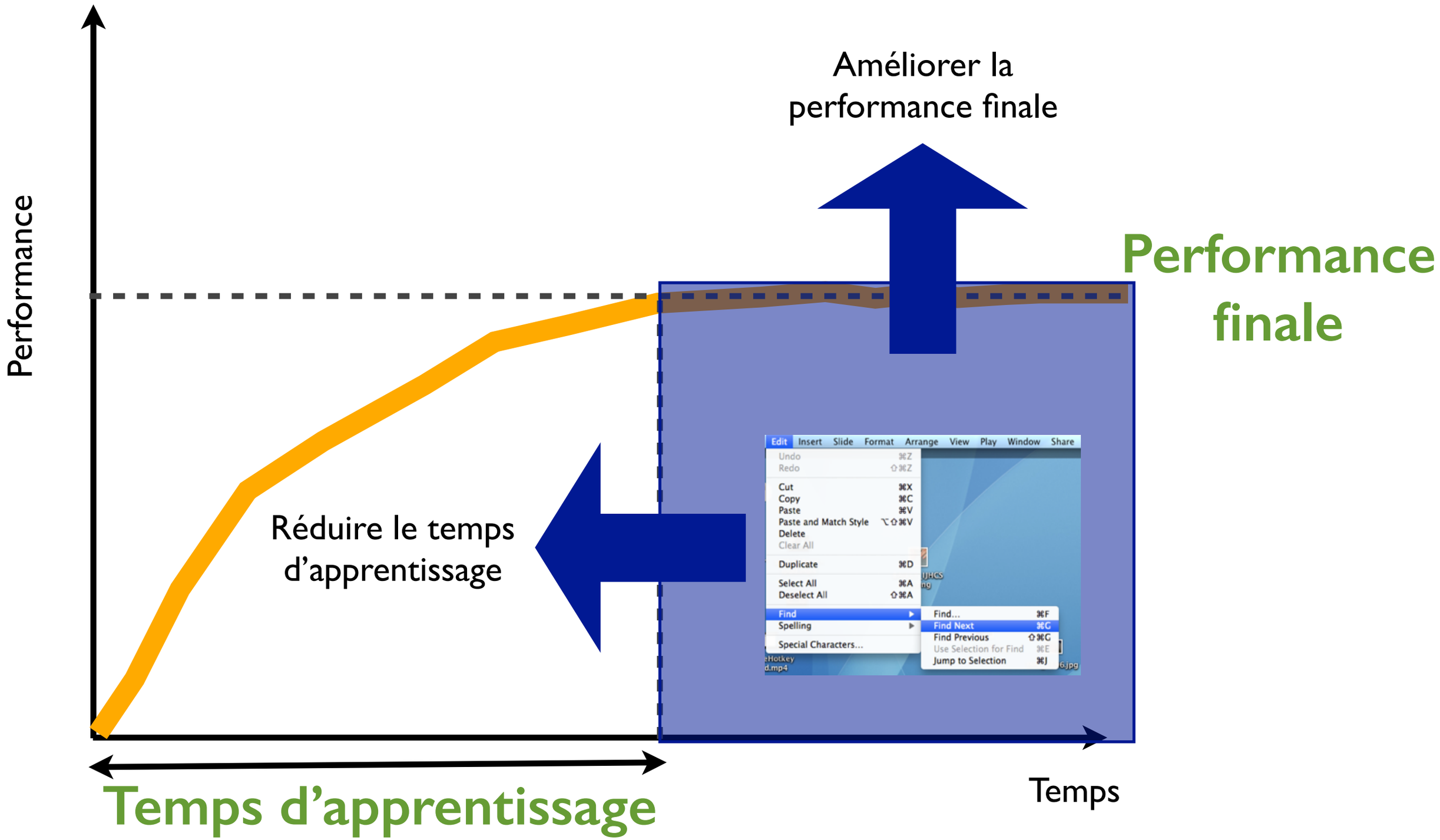


Menu contextuel

Objectifs

Une modélisation « simple » de la performance dans la sélection de commande

J Scarr, A Cockburn, C Gutwin and P Quinn.
Dips and Ceilings: Understanding and Supporting Transitions to Expertise in User Interfaces. Proceedings of ACM CHI'2011 Conference on Human Factors in Computing Systems. Vancouver, Canada. 2011. 2741-2750.

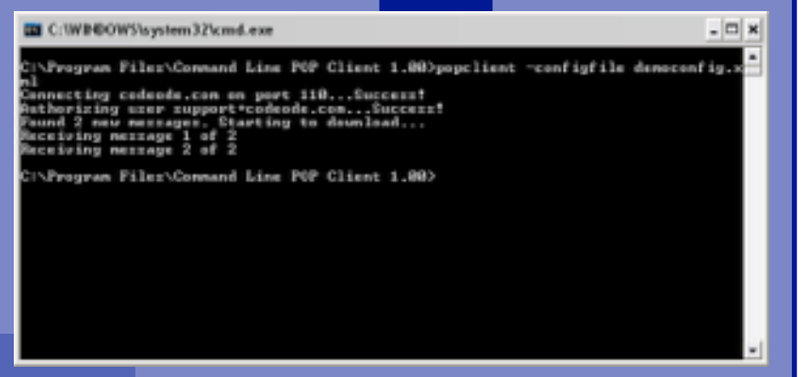

Performance
Améliorer la
performance finale
Performance
finale
Edit Insert Slide Format Arrange View Play Window Share
Undo ⌘Z
Redo ⇧⌘Z
Cut ⌘X
Copy ⌘C
Paste ⌘V
Paste and Match Style ⌥⇧⌘V
Delete
Clear All
Duplicate ⌘D
Select All ⌘A
Deselect All ⇧⌘A
Find
Spelling
Special Characters...
Find... ⌘F
Find Next ⌘G
Find Previous ⇧⌘G
Use Selection for Find ⌘E
Jump to Selection ⌘J
Réduire le temps
d'apprentissage
Temps d'apprentissage
Temps

Performance

Performance finale

Améliorer la performance finale

Performance finale



Réduire le temps d'apprentissage

Temps d'apprentissage

Temps

Performance

Améliorer la
performance finale

Performance
finale

Réduire le temps
d'apprentissage

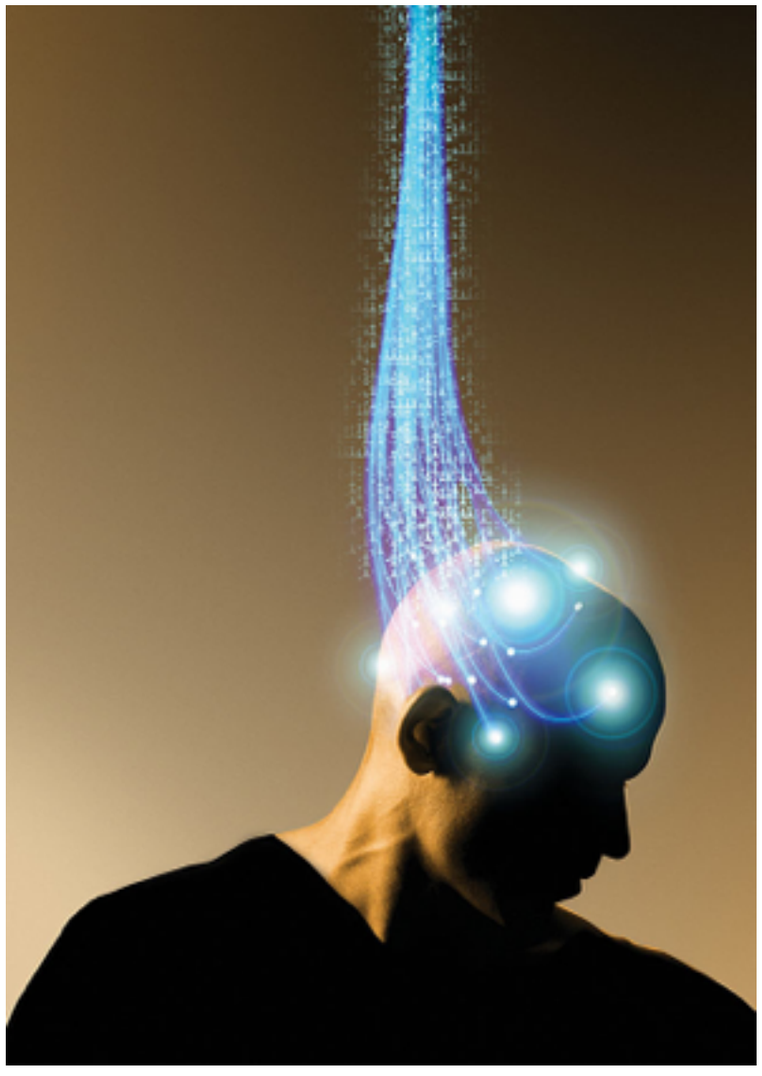
New Play View Guides Themes Masters

Temps

Temps d'apprentissage

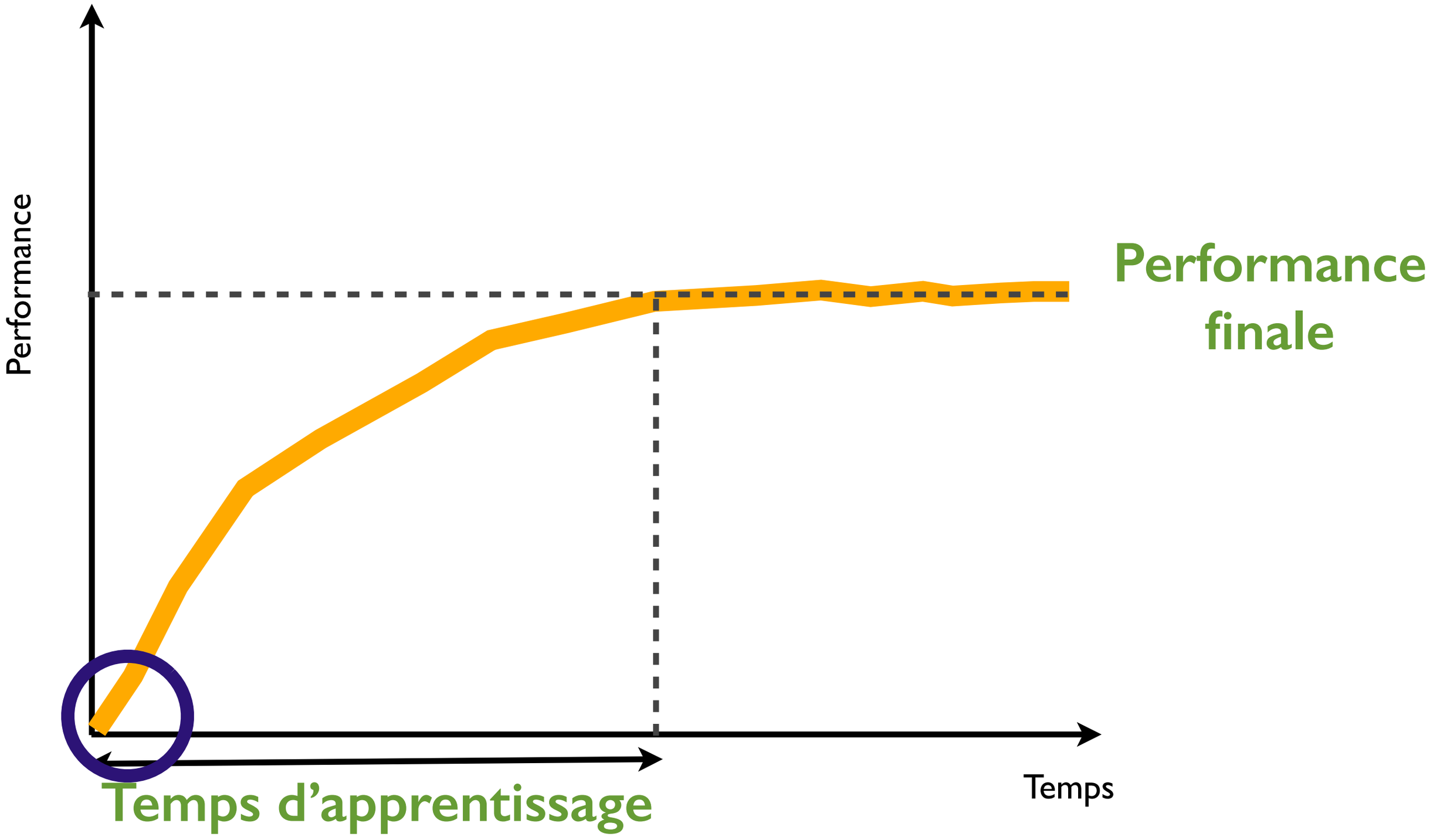
Performance

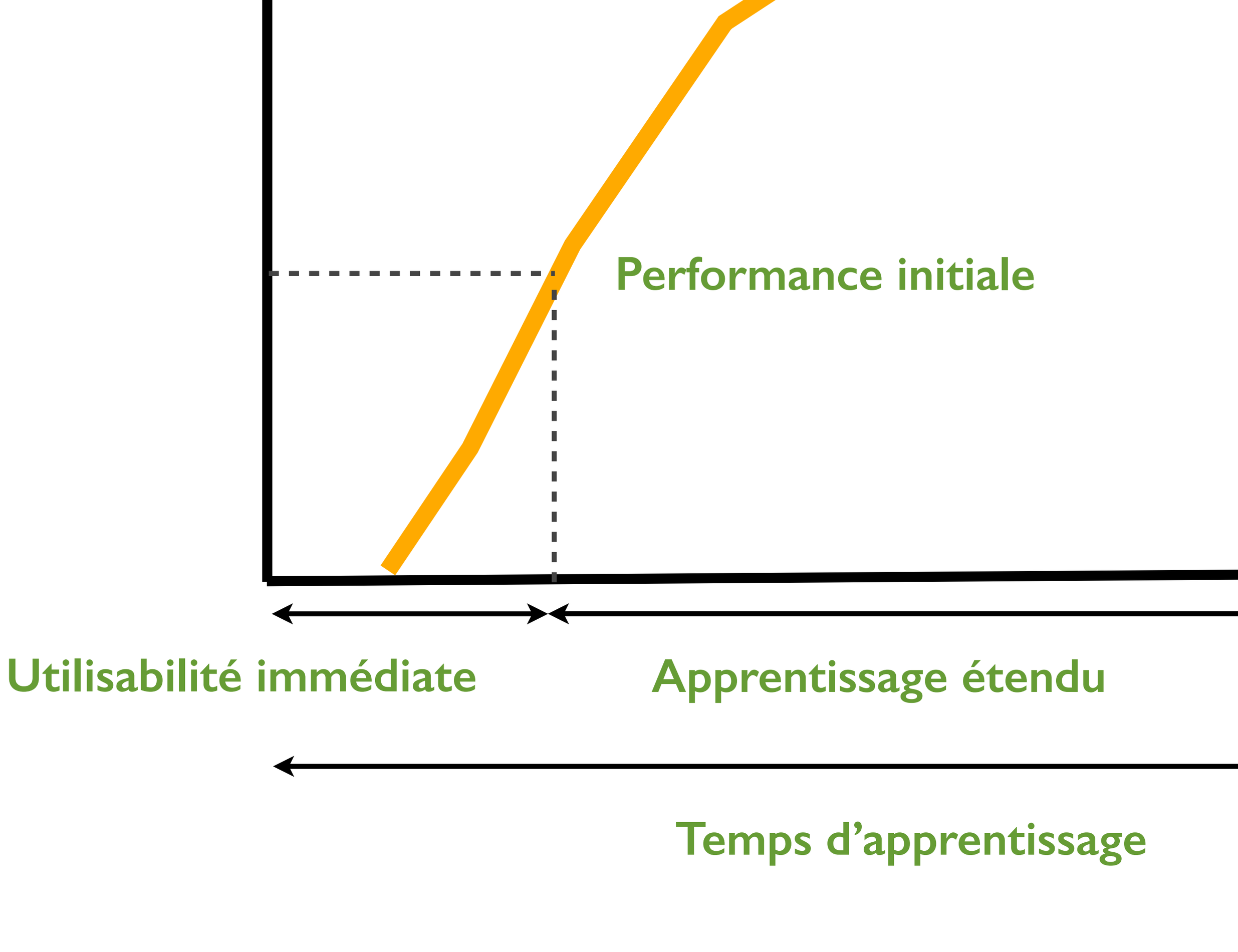
Performance finale

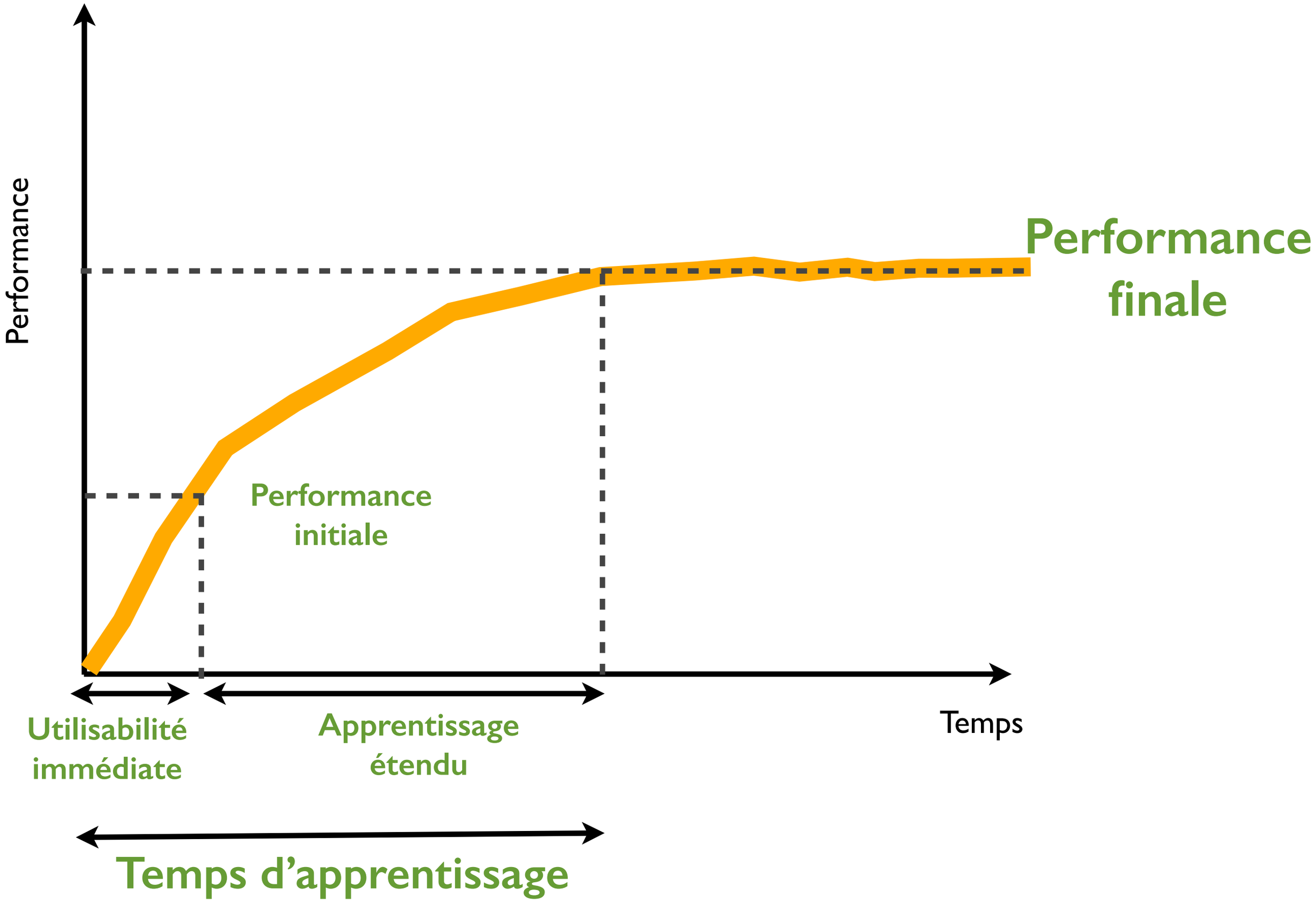


Temps d’apprentissage

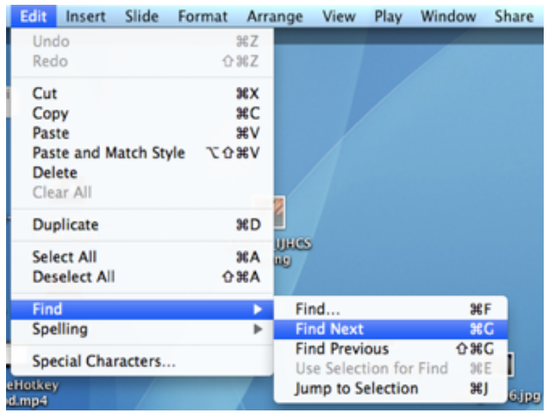
Temps


Performance
Performance finale
Temps d’apprentissage
Temps


Performance initiale
Utilisabilité immédiate
Apprentissage étendu
Temps d’apprentissage


Performance
Performance finale
Performance initiale
Utilisabilité immédiate
Apprentissage étendu
Temps
Temps d’apprentissage

Plusieurs modalités



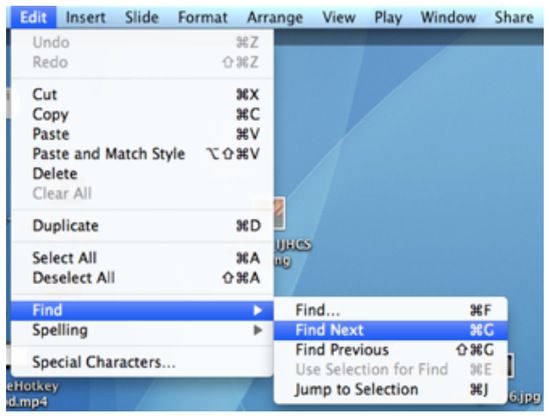
Première modalité
(menu)

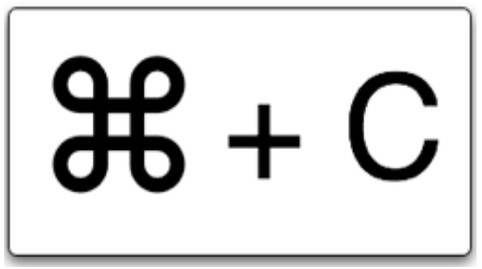
Deuxième modalité
(raccourci clavier)

Changement de modalité

Performance

Perte de performance





Temps

Modalité 1

Modalité 2

Performance

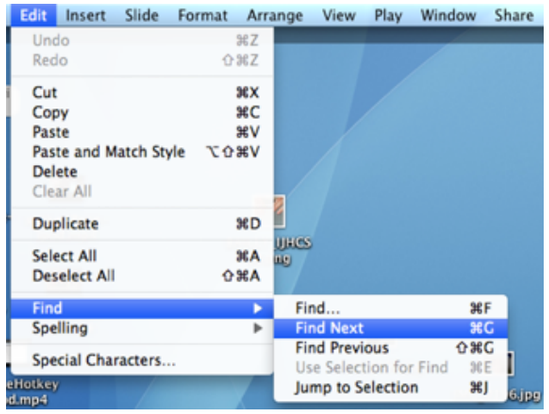
Performance finale (F2)

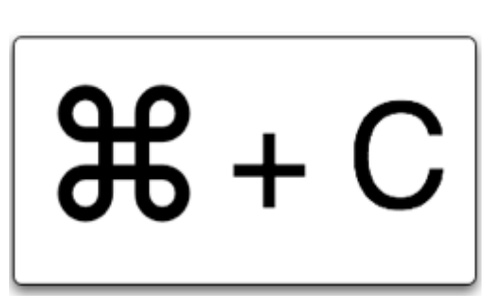
Performance finale (F1)

Performance initiale

Chute de performance

Changement de modalité





Utilisabilité immédiate

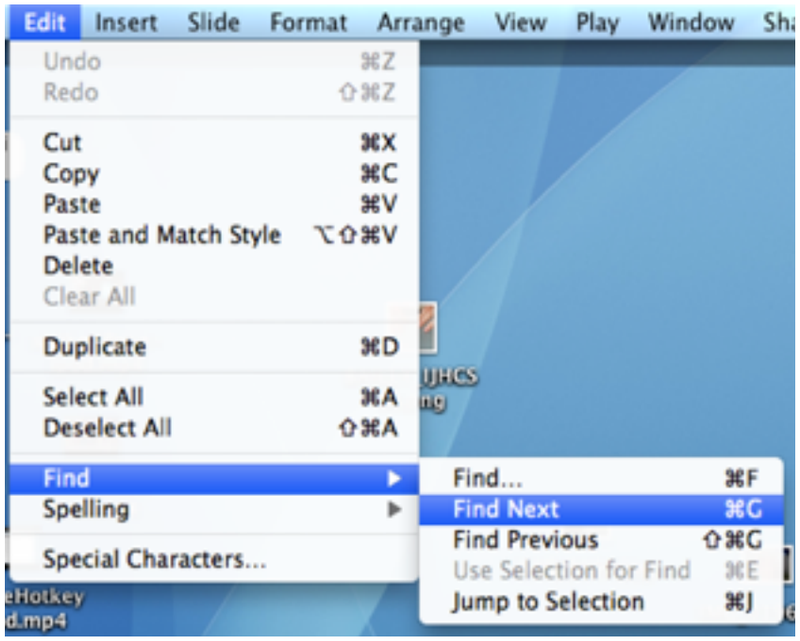
Apprentissage étendu

Temps

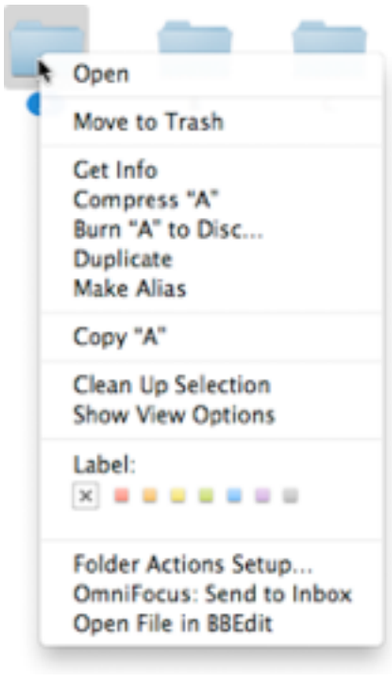
Modalité 1

Modalité 2

Implémentation commandes



Barre de menus



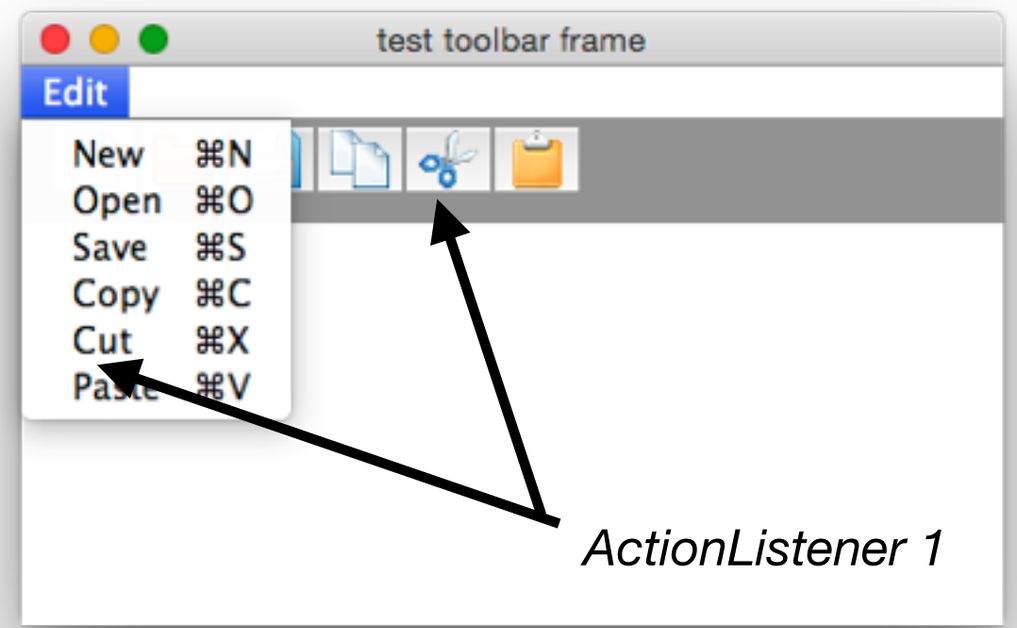
Menu contextuel

Barre d'outils

Raccourci clavier

Implémentation commandes?

```
.........
CutActionListener cutListener = new CutActionListener();
JMenuItem cutMenuItem = new JMenuItem("Cut");
cutMenuItem.addActionListener(cutListener);
JButton cutButton = new JButton("Cut");
cutButton.setIcon(scissorsIcon);
.........

public void selectionHasChanged(Object obj){
    boolean selection = (obj == null);
    cutMenuItem.setEnabled(selection);
    cutButton.setEnabled(selection);
}
```



Implémentation commandes?

Actions

Action est une sous-interface de l'interface *ActionListener* particulièrement utile quand une même commande (action) peut être appelée par différents composants.

Ex: un bouton et un menu réalisent la même opération

- Option1: Utilisation d'un même listener
- Option2: Association des 2 composants à une même ***action***

Comment ajouter raccourcis clavier, activation/désactivation de l'action?

Utiles dès que plusieurs composants sont utilisés pour la même opération.

Actions

Un objet Action permet non-seulement d'implémenter un ActionListener, mais également de regrouper:

1. Une ou plusieurs chaines de texte qui décrivent la commande. Peuvent-être utilisés dans le menu ou comme bulle d'aide.
2. Une ou plusieurs icônes qui illustrent la fonction. Il existe différentes tailles (petite ou grande) pour différents contrôles (menu ou toolbar)
3. Gère les états « actifs/inactifs » de la commande. Au lieu de devoir désactiver les composants, il suffit de désactiver l'action. Tous les composants qui ont cette action sont alors desactivés automatiquement.

Implémentation Actions?

Sous-classe la classe *AbstractAction* et implémente la méthode *actionPerformed(ActionEvent)*

```
static class CutAction extends AbstractAction {
      public CutAction(String text, ImageIcon icon, String desc,
            KeyStroke keystroke) {
         super(text, icon);
         putValue(NAME, text);
         putValue(SHORT_DESCRIPTION, desc);
         putValue(ACCELERATOR_KEY, keystroke);
         putValue(LARGE_ICON_KEY,icon);
      }

      // Implement actionPerformed due to the
      // ActionListener interface
      public void actionPerformed(ActionEvent arg0) {
         Toolkit.getSelectionAndCut();
      }
}
```

Implémentation Actions?

Composants supportés

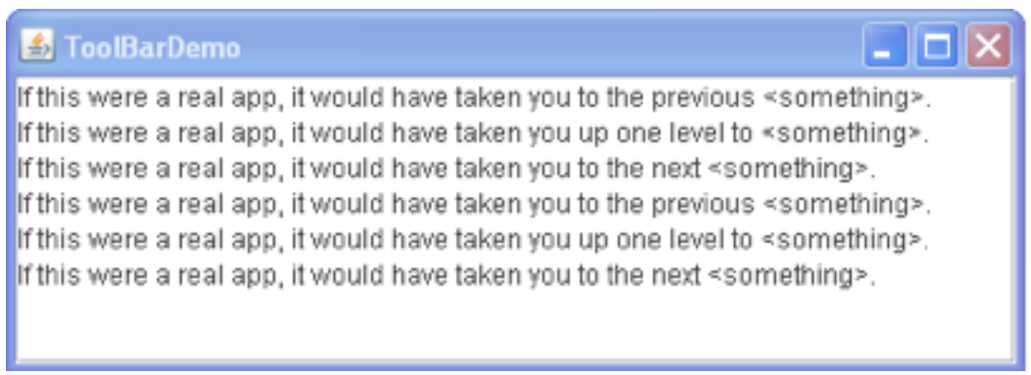
Composants qui supportent un ActionListener:

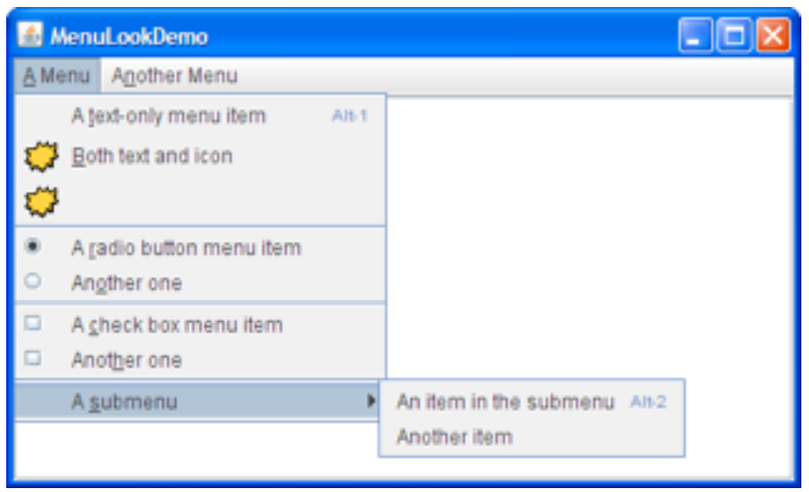
Toolbar buttons

Menus items

Boutons: JButton, JCheckBox, JRadioButton, JMenuItem, JCheckBox, JRadioButton, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem, JToggleButton

Champs de texte: JTextField, JFormattedTextField, JComboBox, JSpinner





Caractéristiques

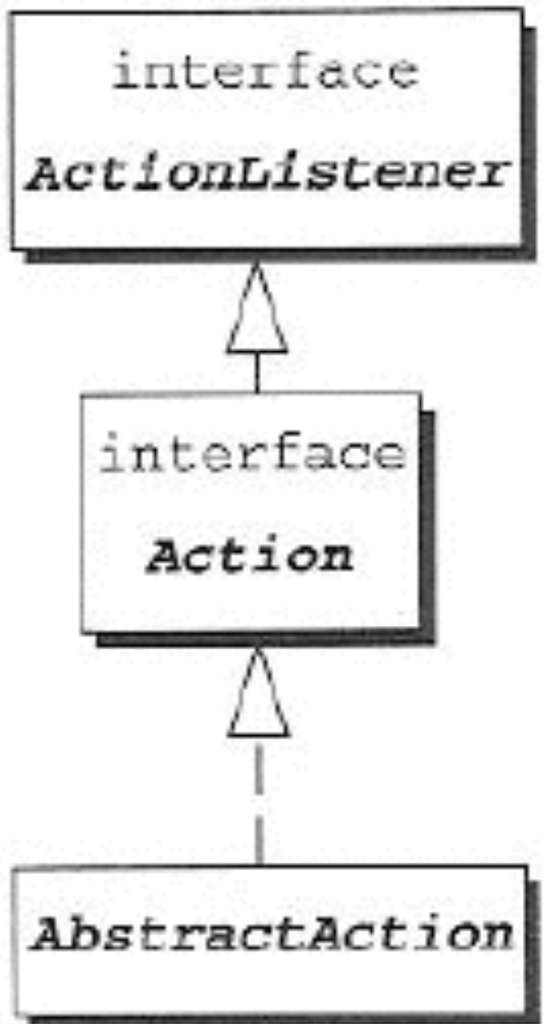
Une fois associés à une action, les composants ont:

- Le même listener
- Le même texte s’il y en a un
- La même icône s’il y en a une
- Le même raccourci clavier
- … les mêmes propriétés liées à l’action

Mise en oeuvre

Une action correspond à un ActionListener particulier

Utilisation de setAction



Propriétés

Une action dispose d'un dictionnaire de propriétés identifiables par des chaînes de caractères

- Le nom (NAME): chaîne de caractères utilisée comme texte à afficher sur un bouton ou un item de menu
- Une icône (SMALL_ICON): également utilisée pour paramétrer les composants graphiques représentant l'action
- Une description courte (SHORT_DESCRIPTION): utilisée pour le texte du tooltip
- Un raccourci clavier (ACCELERATOR_KEY) (l'item menu peut ne pas être visible)

Un mnemonic (MNEMONIC_KEY) (l'item menu doit être visible)

Propriétés

L’interface Action possède deux méthodes (putValue et getValue) pour ajouter des propriétés dans un dictionnaire ainsi que des constantes correspondant aux clés des propriétés dans le dictionnaire

Les composants graphiques reçoivent une notification lorsque les propriétés de l’action changent et ils mettent à jour leur état

Action

Interface qui dérive d'ActionListener

Field Summary	
static String	**ACCELERATOR_KEY** The key used for storing a `KeyStroke` to be used as the accelerator for the action.
static String	**ACTION_COMMAND_KEY** The key used to determine the command string for the `ActionEvent` that will be created when an `Action` is going to be notified as the result of residing in a `Keymap` associated with a `JComponent`.
static String	**DEFAULT** Not currently used.
static String	**LONG_DESCRIPTION** The key used for storing a longer description for the action, could be used for context-sensitive help.
static String	**MNEMONIC_KEY** The key used for storing an int key code to be used as the mnemonic for the action.
static String	**NAME** The key used for storing the name for the action, used for a menu or button.
static String	**SHORT_DESCRIPTION** The key used for storing a short description for the action, used for tooltip text.
static String	**SMALL_ICON** The key used for storing a small icon for the action, used for toolbar buttons.

Method Summary	
void	**addPropertyChangeListener**(PropertyChangeListener listener) Adds a `PropertyChange` listener.
Object	**getValue**(String key) Gets one of this object's properties using the associated key.
boolean	**isEnabled**() Returns the enabled state of the `Action`.
void	**putValue**(String key, Object value) Sets one of this object's properties using the associated key.
void	**removePropertyChangeListener**(PropertyChangeListener listener) Removes a `PropertyChange` listener.
void	**setEnabled**(boolean b) Sets the enabled state of the `Action`.

setAction

AbstractButton.setAction

```
public void setAction(Action a) {
    Action oldValue = getAction();
    if (action==null || !action.equals(a)) {
        action = a;
        if (oldValue!=null) {
            removeActionListener(oldValue);
            oldValue.removePropertyChangeListener(actionPropertyChangeListener);
            actionPropertyChangeListener = null;
        }
        configurePropertiesFromAction(action);
        if (action!=null) {
            // Don't add if it is already a listener
            if (!isListener(ActionListener.class, action)) {
                addActionListener(action);
            }
            // Reverse linkage:
            actionPropertyChangeListener = createActionPropertyChangeListener(action);
            action.addPropertyChangeListener(actionPropertyChangeListener);
        }
        firePropertyChange("action", oldValue, action);
        revalidate();
        repaint();
    }
}
```

AbstractAction

Classe qui dérive d'Action

Constructor Summary
AbstractAction() Defines an Action object with a default description string and default icon.
AbstractAction(String name) Defines an Action object with the specified description string and a default icon.
AbstractAction(String name, Icon icon) Defines an Action object with the specified description string and a the specified icon.

Method Summary	
void	**addPropertyChangeListener**(PropertyChangeListener listener) Adds a PropertyChangeListener to the listener list.
protected Object	**clone**() Clones the abstract action.
protected void	**firePropertyChange**(String propertyName, Object oldValue, Object newValue) Supports reporting bound property changes.
Object[]	**getKeys**() Returns an array of Objects which are keys for which values have been set for this AbstractAction, or null if no keys have values set.
PropertyChangeListener[]	**getPropertyChangeListeners**() Returns an array of all the PropertyChangeListeners added to this AbstractAction with addPropertyChangeListener().
Object	**getValue**(String key) Gets the Object associated with the specified key.
boolean	**isEnabled**() Returns true if the action is enabled.
void	**putValue**(String key, Object newValue) Sets the Value associated with the specified key.
void	**removePropertyChangeListener**(PropertyChangeListener listener) Removes a PropertyChangeListener from the listener list.
void	**setEnabled**(boolean newValue) Enables or disables the action.

Exemple

```
JButton b1 = new JButton();
AbstractAction monaction = new ExempleAction();
monaction.putValue(Action.NAME, "Bouton 1");
monaction.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_A);
monaction.putValue(Action.ACCELERATOR_KEY, KeyStroke.getKeyStroke(
        KeyEvent.VK_T, ActionEvent.ALT_MASK));
b1.setAction(monaction);

class ExempleAction extends AbstractAction {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Action accomplie");
    }
}
```

JToolBar, Jmenu et JPopupMenu

JButton, JToolBar, Jmenu, JPopupMenu possèdent des méthodes add qui prennent en paramètre une référence de type Action

Constructor Summary

JButton() Creates a button with no set text or icon.
JButton(Action a) Creates a button where properties are taken from the Action supplied.
JButton(Icon icon) Creates a button with an icon.
JButton(String text) Creates a button with text.
JButton(String text, Icon icon) Creates a button with initial text and an icon.

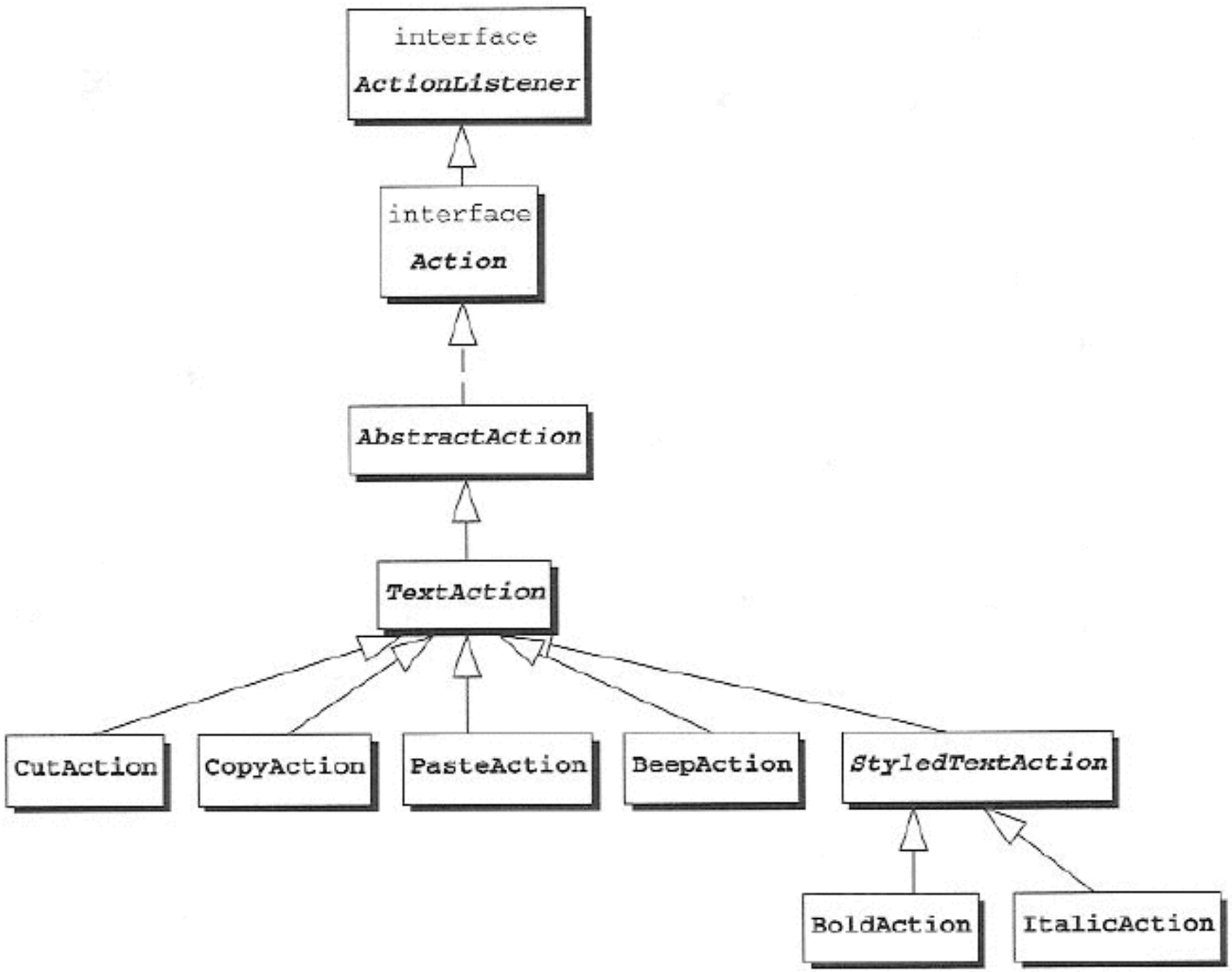
Actions sur les composants texte

Pour les composants texte, utilisation de la classe abstraite TextAction qui dérive d'AbstractAction

DefaultEditorKit définit 48 sous-classes de TextAction, identifiables par des chaînes de caractères correspondant aux noms de ces actions. 8 sont des classes incluses publiques qui peuvent être instanciées:

- CopyAction, CutAction, PasteAction …

Actions sur les composants texte



Raccourcis claviers (Key Bindings)

Fournir un raccourci clavier pour un composant

- Ex: appeler une méthode du composant quand ce dernier a le focus et que la barre espace est pressée

Modifier le comportement de raccourcis clavier existants

Associer un raccourci clavier pour une action existante

- Ex: utilisation de control+shift+Insert pour coller

Le plus souvent, pas besoin de prendre en charge directement les raccourcis clavier: directement gérés par les actions avec les mnemonics et les accelerators

Raccourcis claviers (Key Bindings)

Le support pour les raccourcis clavier est fourni par JComponent et repose sur les classes InputMap et ActionMap

Un objet InputMap établit la correspondance entre un raccourci clavier et le nom d'une action

Un objet ActionMap établit la correspondance entre le nom d'une action et l'action

Une alternative au Key Bindings est l'utilisation de key listener mais plus difficiles à gérer

Raccourcis claviers (Key Bindings)

Chaque JComponent a une ActionMap et 3 InputMap

- JComponent.WHEN_FOCUSED: quand le composant a le focus
- JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT: utilisé pour les composants composites (utilisé par JTable par ex)
- JComponent.WHEN_IN_FOCUSED_WINDOW: la fenêtre du composant a le focus (utilisé par les mnemonics et accelerators)

Raccourcis claviers (Key Bindings)

Fonctionnement

- Quand l'utilisateur appuie sur une touche, le gestionnaire d'événements clavier de JComponent cherche un raccourci correspondant parmi les différentes InputMap
- Quand le raccourci est trouvé, le gestionnaire cherche l'action correspondante à exécuter dans l'ActionMap
- Si l'Action est activée, elle est exécutée, sinon le gestionnaire continue sa recherche parmi les autres InputMap. S'il existe plusieurs raccourcis pour une Action, seule la première trouvée est utilisée.
- Les InputMaps sont vérifiées dans cet ordre: WHEN_FOCUSED, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT (dans l'ordre inverse de la hiérarchie), WHEN_IN_FOCUSED_WINDOW

Raccourcis claviers (Key Bindings)

Exemple: Un bouton a le focus et l'utilisateur appuie sur la barre espace

- Les key listener du bouton (s'ils existent) sont notifiés de l'événement
- En supposant que l'événement n'est pas consommé, la WHEN_FOCUSED InputMap est consultée
- Un raccourci est trouvé parce que JButton utilise cette InputMap pour associer la touche espace à une action
- Le nom de l'action est obtenu par l'ActionMap et la méthode actionPerformed est alors appelée
- L'événement clavier est consommé et le processus s'arrête

Exemple

Associer le raccourci coller à un label

```
label.getInputMap().put(KeyStroke.getKeyStroke(KeyEvent.VK_V, InputEvent.CTRL_MASK), "paste");
label.getActionMap().put("paste", TransferHandler.getPasteAction());
```

TransferHandler.getPasteAction() renvoie une action qui se comporte comme un coller, cad que l'action va appeler la méthode importData du TransferHandler associé au JComponent qui est la source de l'événement.

Patron de conception commande

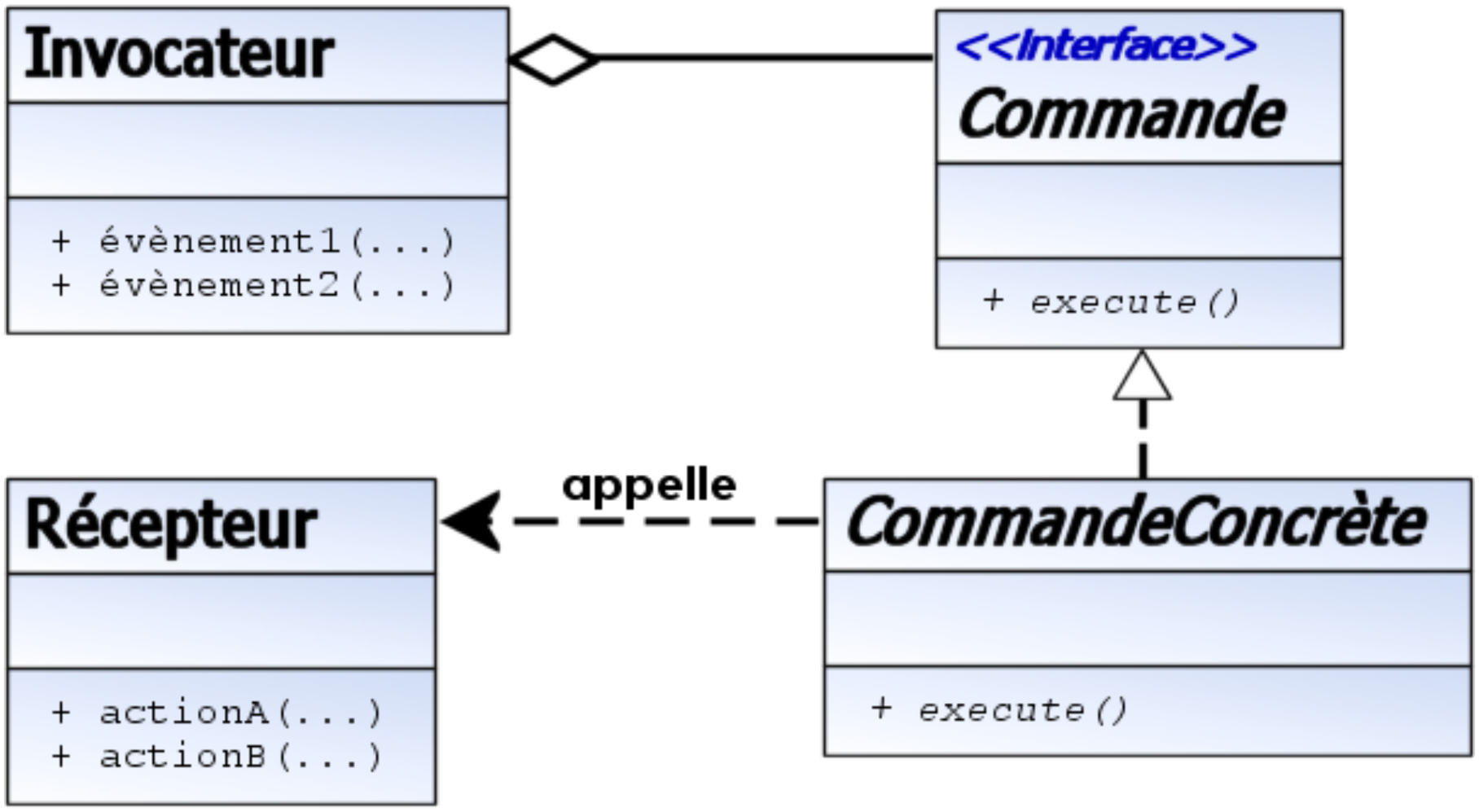
Offre une solution au problème suivant: certains objets doivent envoyer l'ordre d'exécuter une action à d'autres objets. Les objets qui déclenchent l'ordre ne savent pas en quoi consiste cette action, pas plus qu'il ne connaissent les objets récepteurs de cet ordre.

Patron de conception commande

Utilisation d'une classe abstraite Commande qui fournit une interface pour exécuter des opérations

Les sous classes concrètes de Commande doivent connaître le récepteur pour l'action (en enregistrant le récepteur comme une instance de classe) et implémenter Execute pour invoquer la requête sur le récepteur

Patron de conception commande



Patron de conception commande

```
/* Invocateur */
public class Switch
{
    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd,Command flipDownCmd)
    {
        this.flipUpCommand=flipUpCmd;
        this.flipDownCommand=flipDownCmd;
    }

    public void flipUp()
    {
        flipUpCommand.execute();
    }

    public void flipDown()
    {
        flipDownCommand.execute();
    }
}
```

Patron de conception commande

```
/* Récepteur */
public class Light
{
    public Light() {  }

    public void turnOn()
    {
        System.out.println("The light is on");
    }

    public void turnOff()
    {
        System.out.println("The light is off");
    }
}

/* Commande */
public interface Command
{
    void execute();
}
```

Patron de conception commande

```
/* Commande concrète pour allumer la lumière */
public class TurnOnCommand implements Command
{
    private Light theLight;

    public TurnOnCommand(Light light)
    {
        this.theLight=light;
    }

    public void execute()
    {
        theLight.turnOn();
    }
}
```

Patron de conception commande

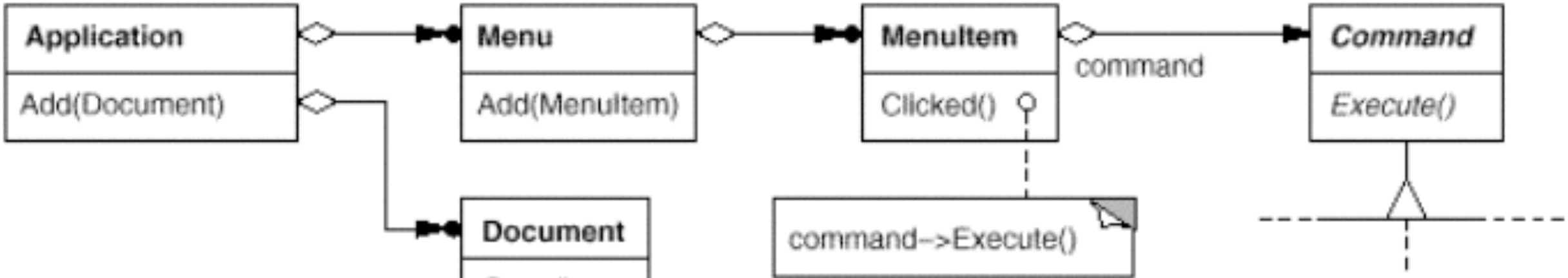
```
/* Commande concrète pour éteindre la lumière */
public class TurnOffCommand implements Command
{
    private Light theLight;

    public TurnOffCommand(Light light)
    {
        this.theLight=light;
    }

    public void execute()
    {
        theLight.turnOff();
    }
}
```

Patron de conception commande

```
/* Classe de test */
public class TestCommand
{
    public static void main(String[] args)
    {
        Light lamp = new Light();
        Command switchUp=new TurnOnCommand(lamp );
        Command switchDown=new TurnOffCommand(lamp );

        Switch s = new Switch(switchUp,switchDown);
        s.flipUp();
        s.flipDown();
    }
}
```

Patron de conception commande

Exemple pour un menu



Cas d'un menu

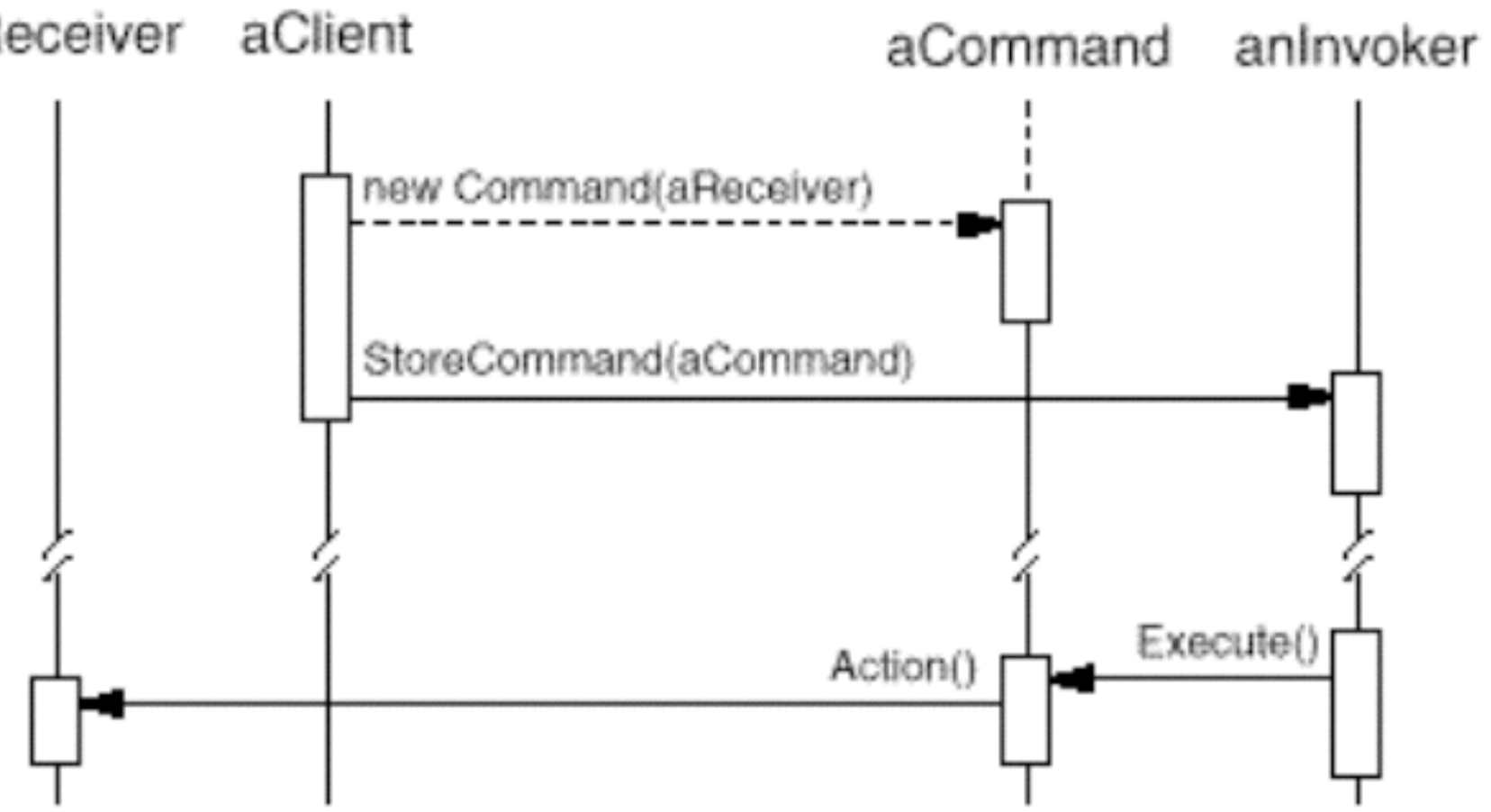
```
// the Command Pattern in Java
public interface Command
{
  public void execute();
}

public class FileOpenMenuItem extends JMenuItem implements Command
{
  public void execute()
  {
    // your business logic goes here
  }
}


public void actionPerformed(ActionEvent e)
{
  Command command = (Command)e.getSource();
  command.execute();
}
```

Patron de conception commande

aClient est l'application

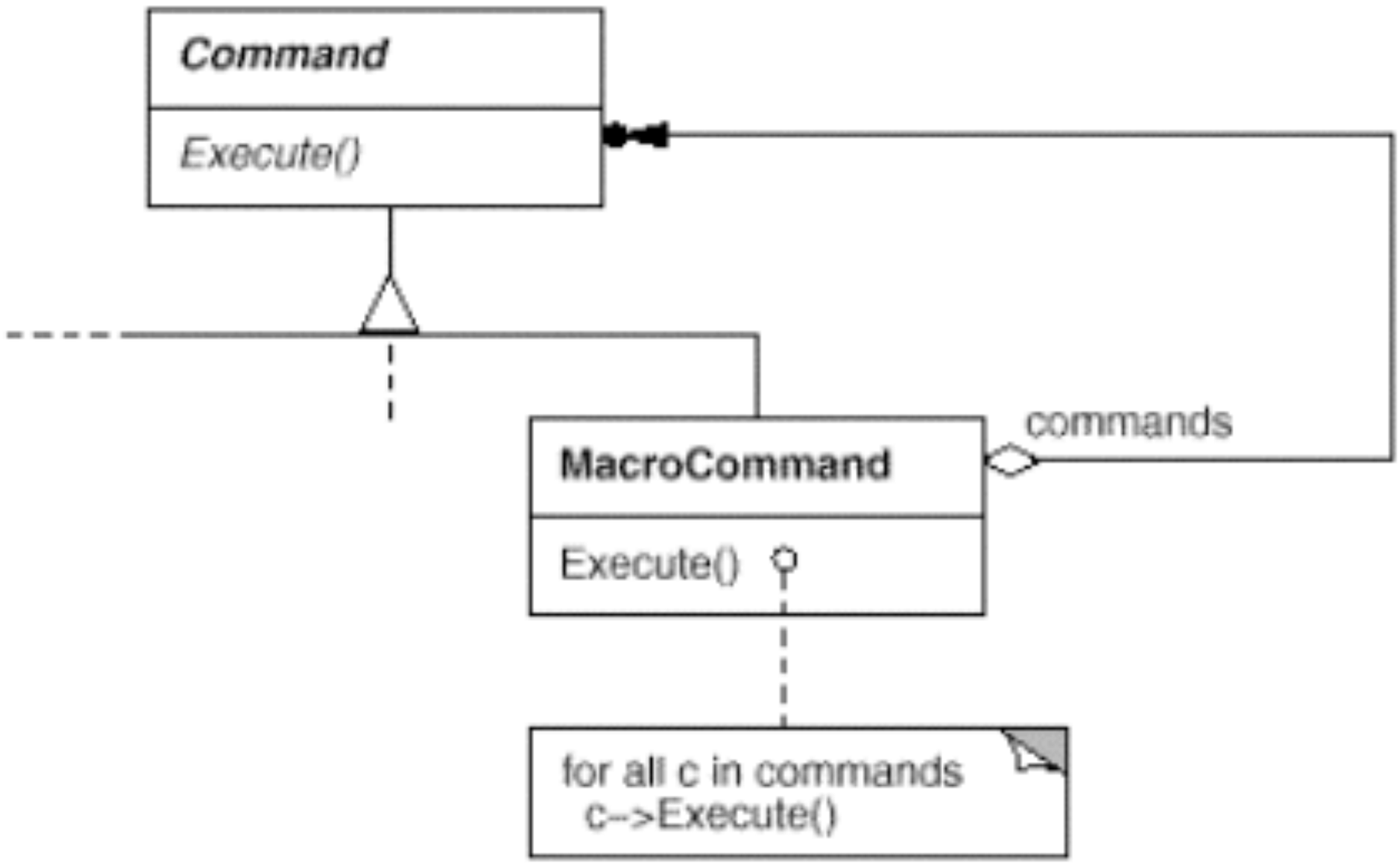
anInvoker est un itemMenu par exemple

aReceiver est un document par exemple



Patron de conception commande
Macro commandes

Exécution d'une séquence de commandes



Patron de conception commande

Partage d’instances

Possibilité de remplacer les commandes dynamiquement

Les composants graphiques peuvent ainsi déclencher une opération sur un autre objet qu’ils ne connaissent pas

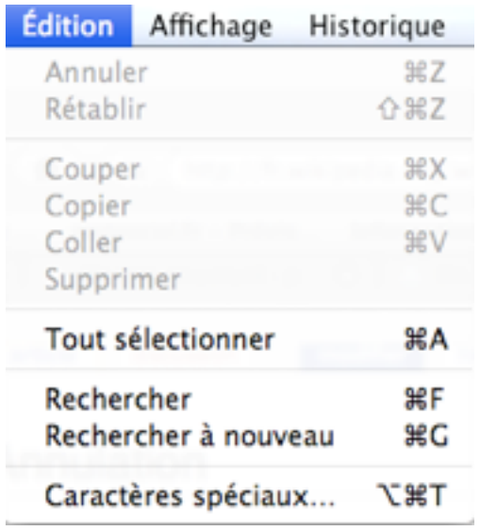
Annuler/restaurer (Undo/Redo)

Différents accès

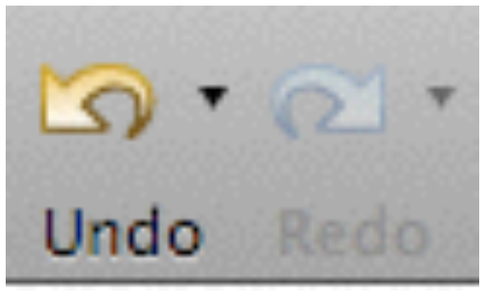
Raccourcis clavier

Opération	Microsoft Windows	Mac OS	KDE / GNOME
Annulation	Ctrl + Z	⌘ + Z	Ctrl + Z
Restauration	Ctrl + Y	⌘ + Maj + Z	Ctrl + Maj + Z

Menus



Barres d'outils



Intérêts

Fonctionnalité essentielle de toute interface

Permet d’explorer des fonctionnalités d’un logiciel sans crainte

Difficultés

Quelles sont les opérations annulables?

Diversité des opérations annulables

- Ajout de texte, déplacement d'objet…

Quel niveau de détail utiliser?

- Ex: annuler lettre par lettre ou mot par mot, annuler un déplacement pixel par pixel ou retour à la position initiale?

Quelles informations sauvegarder pour refaire l'opération? Vaut-il mieux sauvegarder l'opération ou utiliser la méthode inverse?

Nombre d'opérations annulables

- 1 sous Notepad, 3 sous Paint, 20 sous Photoshop

Différents modèles

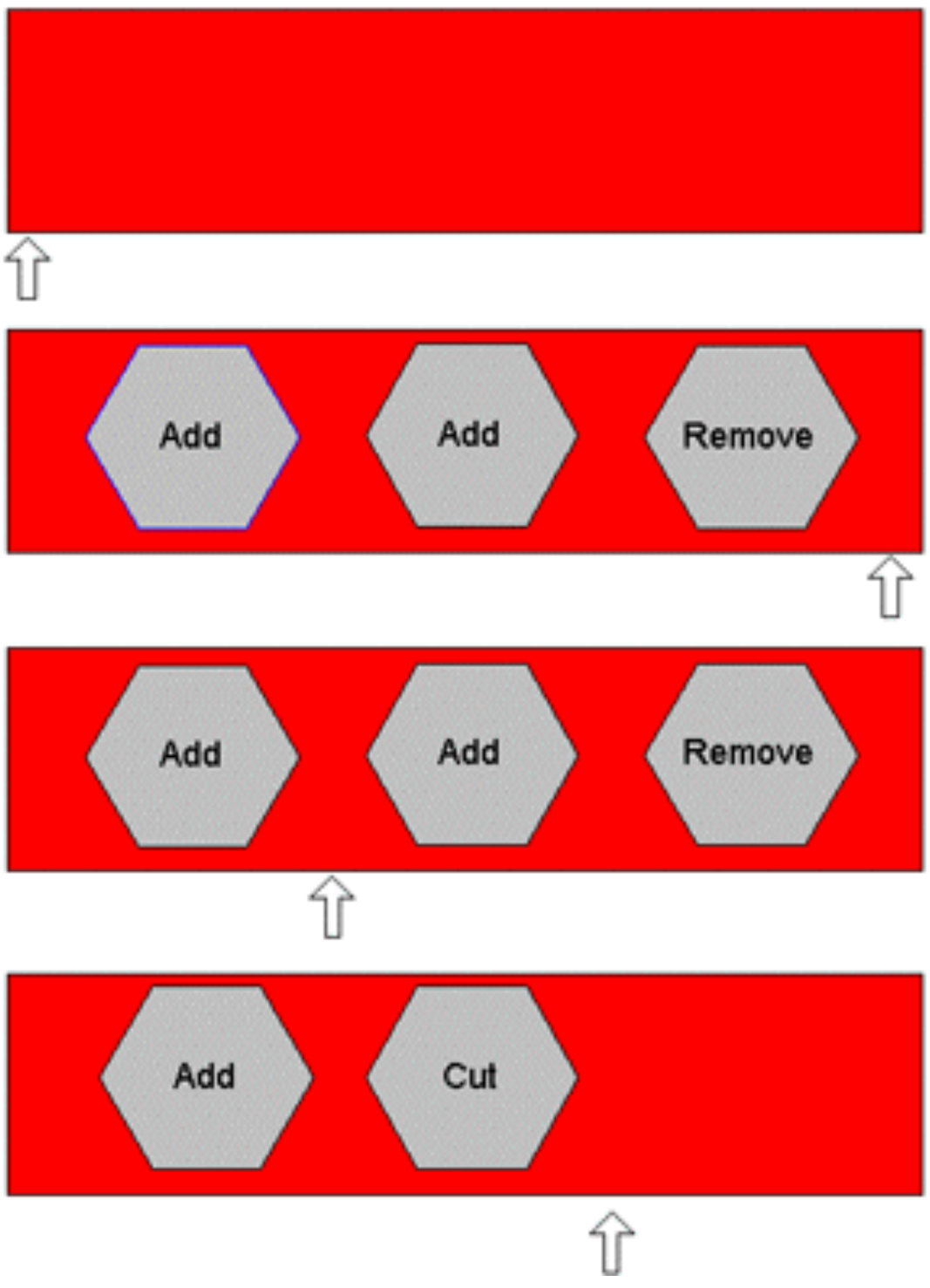
Modèle linéaire

- L'utilisateur doit annuler la dernière opération avant d'annuler les précédentes

Modèle non-linéaire

- L'utilisateur est libre de choisir l'opération à annuler dans l'historique

Modèle linéaire



(a) *Queue vide*

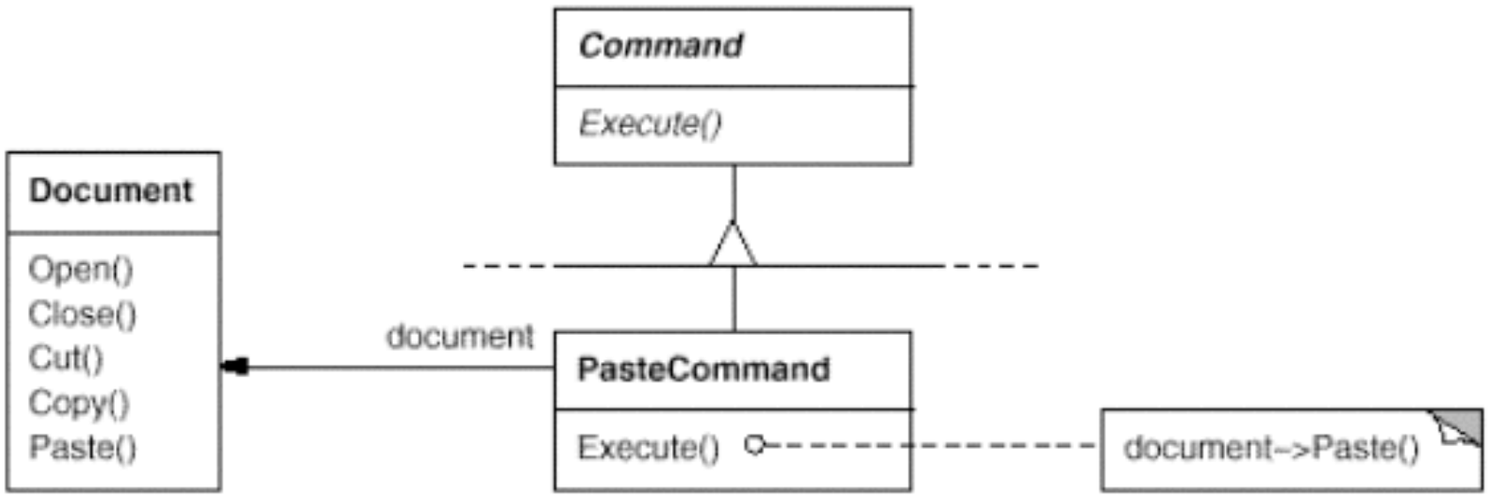
(b) *3 actions exécutées et ajoutées dans la queue*

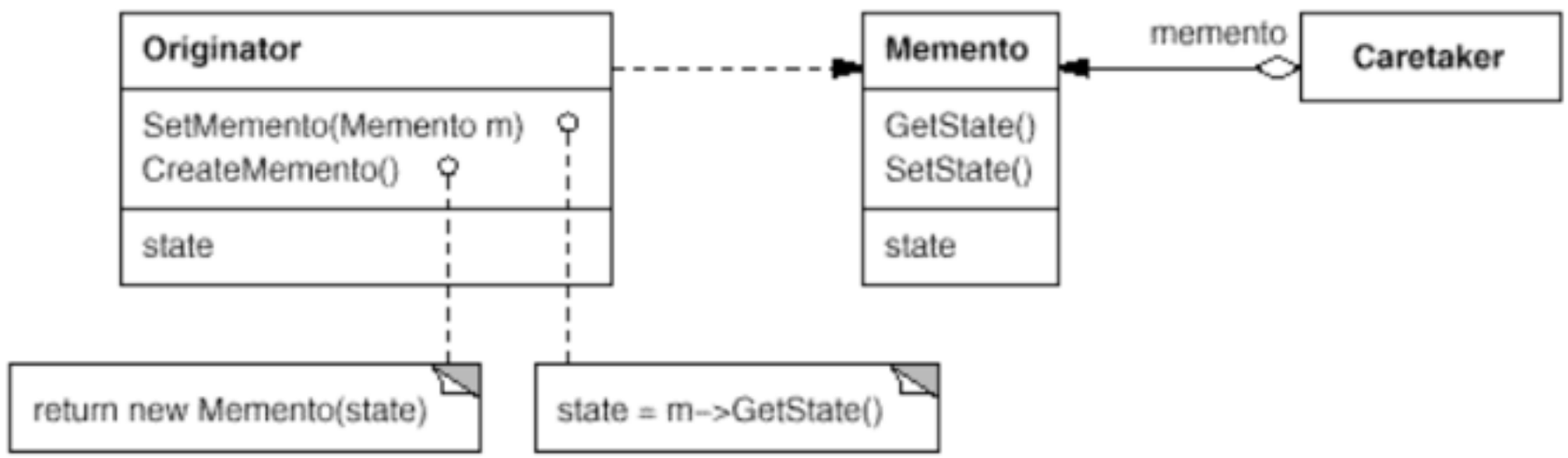
(c) *Undo execute sur deux actions*

(d) *Exécution d'une nouvelle action*

Mise en oeuvre

Utilisation des patrons de conception Command et Memento





Utilisation du patron de conception Commande

La méthode Execute de Commande peut enregistrer des informations pour annuler ses effets plus tard.

L'annulation se fait en ajoutant une méthode UnExecute à Commande

Les commandes exécutées sont enregistrées dans une liste

Un nombre illimité d'undo et redo est obtenu en parcourant la liste en avant et en arrière et en appelant les méthode UnExecute et Execute

Utilisation du patron de conception Commande

Informations additionnelles à enregistrer lors de l'appel à Execute

- L'objet récepteur qui applique les opérations en réponse à la requête
- Les arguments de l'opération effectuée sur le récepteur
- Des valeurs du récepteur qui peuvent être modifiées suite à une réponse à la requête
- Le récepteur doit fournir des méthodes pour permettre à l'objet commande de rétablir l'état antérieur du récepteur

Utilisation du patron de conception Commande

Pour supporter un seul niveau de undo, l'application a seulement besoin d'enregistrer la dernière commande exécutée

Pour plusieurs niveaux de undo/redo, l'application doit enregistrer un historique sous forme de liste des commandes

Utilisation du patron de conception Commande

Eviter l'accumulation d'erreurs dans le processus d'annulation

- De petites erreurs peuvent s'accumuler au fil des opérations de undo/redo et faire diverger l'application de son état initial
- Il faut s'assurer que toutes les informations nécessaires sont enregistrées pour revenir à l'état précédent

Utilisation du patron de conception Memento

Patron de conception Memento

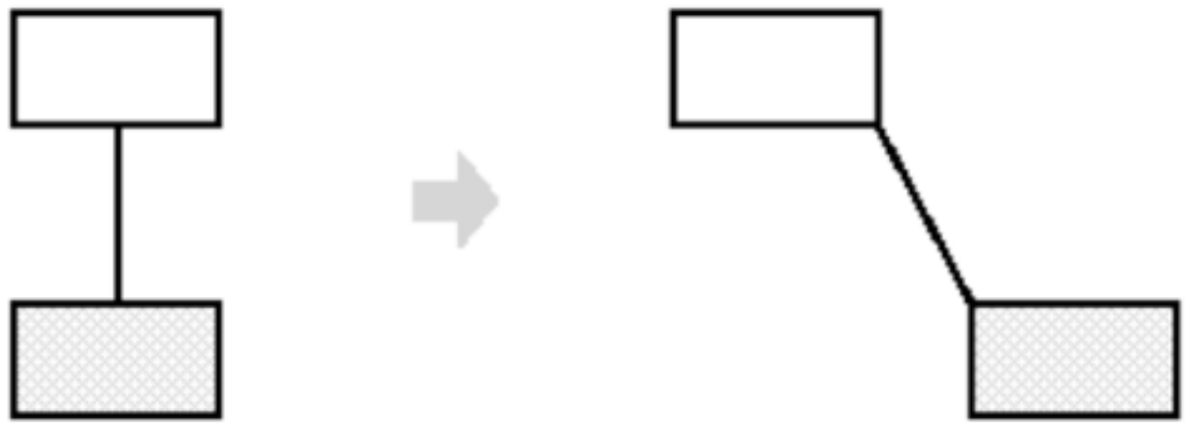
Les objets encapsulent normalement leur état, le rendant inaccessible à d'autres objets. Exposer leur état violerait l'encapsulation

Le patron de conception Memento permet de restaurer un objet à un état antérieur en enregistrant son état interne sans violer l'encapsulation

Patron de conception Memento

Exemple: éditeur graphique qui permet de connecter des objets

Utilisation d'un solveur de contraintes



Patron de conception Memento

Solution undo simple qui consiste à déplacer l'objet d'une distance équivalente à la distance d'origine

Patron de conception Memento

Memento est un objet qui enregistre une photo de l'état interne d'un autre objet (l'originator)

Le mécanisme d'annulation va demander un objet memento à l'originator. L'originator initialise alors le memento avec l'information qui caractérise son état interne

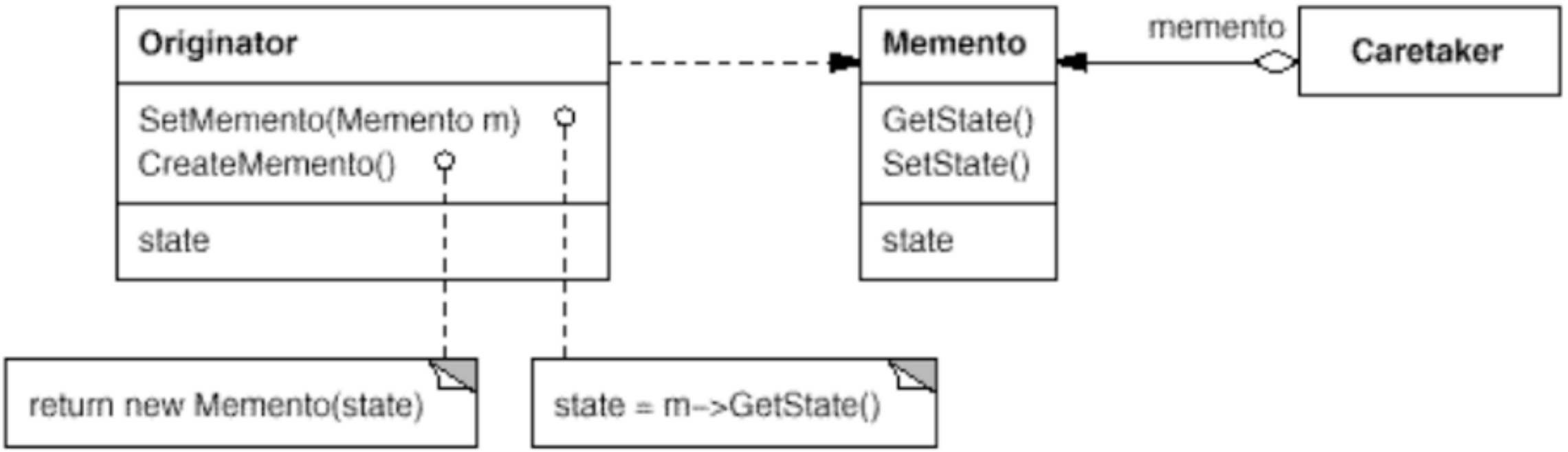
Seul l'originator peut lire et écrire l'objet memento. Memento est dit opaque aux autres objets.

Patron de conception Memento

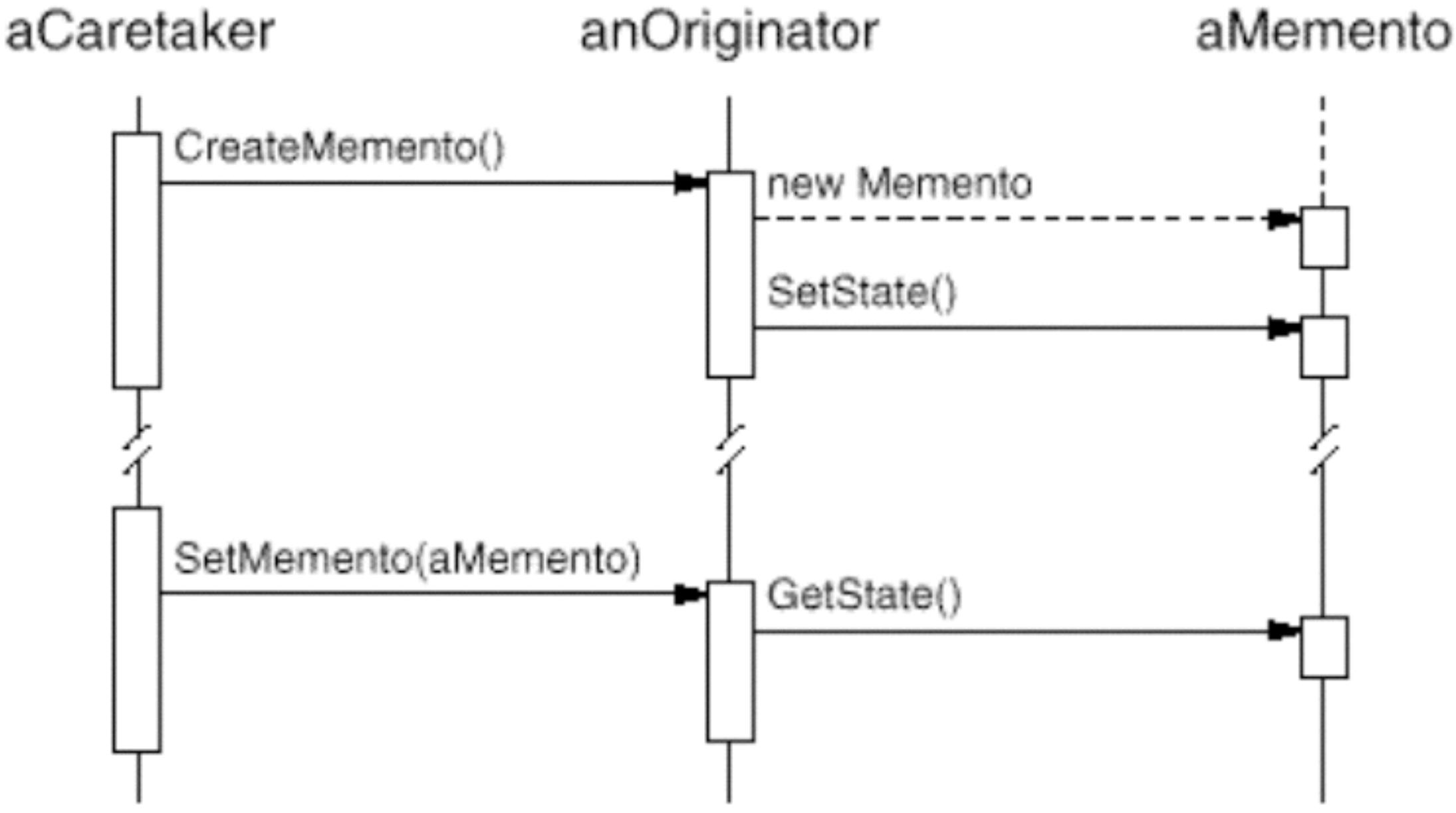
Retour sur l'éditeur graphique

1. L'éditeur demande un memento au ConstraintSolver suite au déplacement d'un objet
2. ConstraintSolver crée et renvoie un memento. Le memento contient les structures de données qui décrivent l'état courant des équations et des variables internes du solveur de contraintes
3. Si l'opérateur annule le déplacement, l'éditeur rend le memento au solveur de contraintes
4. Le solveur met à jour ses structures internes pour que ses équations et variables internes retournent à leur état initial

Patron de conception Memento



Patron de conception Memento



Patron de conception Memento

```
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        public String getSavedState() {
            return state;
        }
    }
}
```

Patron de conception Memento

```
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new
        ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

Patron de conception Memento

En Java

```
class MementoExample {

    public static void main(String[] args) {

        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State3");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(caretaker.getMemento(1));

    }

}
```

Undo/Redo en Java

Les actions ne possèdent pas de méthode undo

Mise en œuvre du patron de conception Command avec les classes UndoManager et UndoEdit

Classe javax.swing.undo.UndoManager qui enregistre un ensemble d'opérations annulables dans un Vector (de taille 100 par défaut)

Undo/Redo en Java

Les opérations annulables doivent implémenter l'interface javax.swing.undo.UndoableEdit et en particulier les méthodes undo et redo

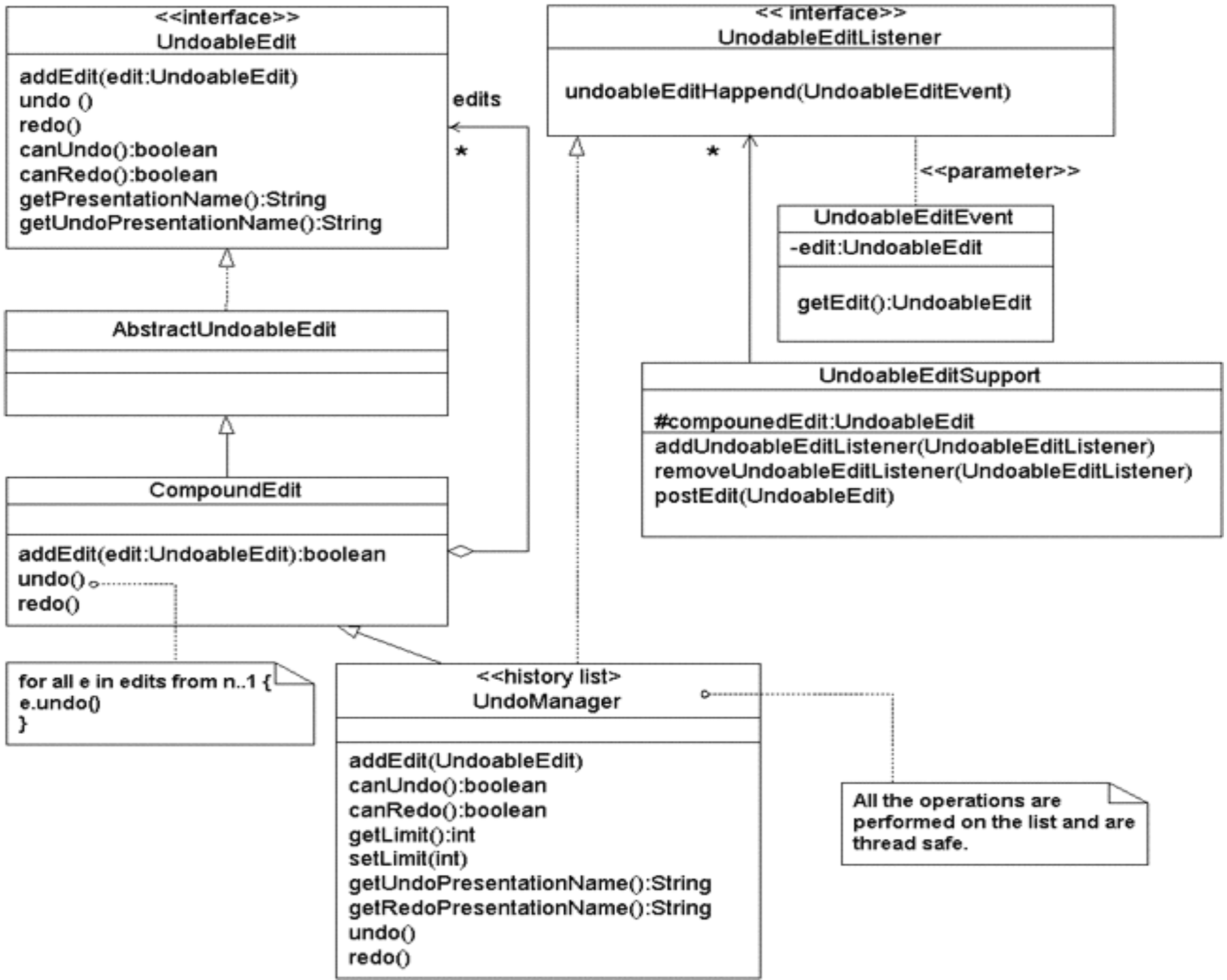
Pour éviter d'avoir à implémenter les 9 autres méthodes de UndoableEdit, il suffit de créer une classe d'opérations annulable en héritant de la classe AbstractUndoableEdit qui implémente cette interface

Listener : UndoableEditListener

UndoableEdit

Method Summary	
boolean	**addEdit**(UndoableEdit anEdit) This UndoableEdit should absorb anEdit if it can.
boolean	**canRedo**() True if it is still possible to redo this operation.
boolean	**canUndo**() True if it is still possible to undo this operation.
void	**die**() May be sent to inform an edit that it should no longer be used.
String	**getPresentationName**() Provides a localized, human readable description of this edit suitable for use in, say, a change log.
String	**getRedoPresentationName**() Provides a localized, human readable description of the redoable form of this edit, e.g.
String	**getUndoPresentationName**() Provides a localized, human readable description of the undoable form of this edit, e.g.
boolean	**isSignificant**() Returns false if this edit is insignificant--for example one that maintains the user's selection, but does not change any model state.
void	**redo**() Re-apply the edit, assuming that it has been undone.
boolean	**replaceEdit**(UndoableEdit anEdit) Returns true if this UndoableEdit should replace anEdit.
void	**undo**() Undo the edit that was made.

Diagramme des classes de gestion des opérations annulables



Evénements

Il existe l'interface UndoableEditListener et une classe spécifique UndoableEditEvent

Un tel événement comporte une source et un UndoableEdit

Les auditeurs sont de l'interface UndoableEditListener, avec la méthode undoableEditHappened(UndoableEditEvent e)

UndoManager implémente UndoableEditListener avec la méthode

```
public void undoableEditHappened(UndoableEditEvent e) {
    addEdit(e.getEdit());
}
```

Evénements

Problème: seuls les composants texte sont capables d'abonner un UndoableEditListener

Solution: ajouter les méthodes nécessaires (alternative: utiliser UndoableEditSupport)

```
class JListe extends JList {
    JListe(DefaultListModel listModel) {
        super(listModel);
    }

    public void addUndoableEditListener(UndoableEditListener listener){
        listenerList.add(UndoableEditListener.class,listener);
    }

    public void removeUndoableEditListener(UndoableEditListener listener){
        listenerList.remove(UndoableEditListener.class,listener);
    }
    public void fireUndoableEditUpdate(UndoableEditEvent e){
        Object[]listeners=listenerList.getListenerList();
        for(int i=listeners.length-2;i>=0;i-=2){
            if(listeners[i]==UndoableEditListener.class)
                ((UndoableEditListener)listeners[i+1]).undoableEditHappened(e);
        }
    }
}
```

Création de nouveaux composants

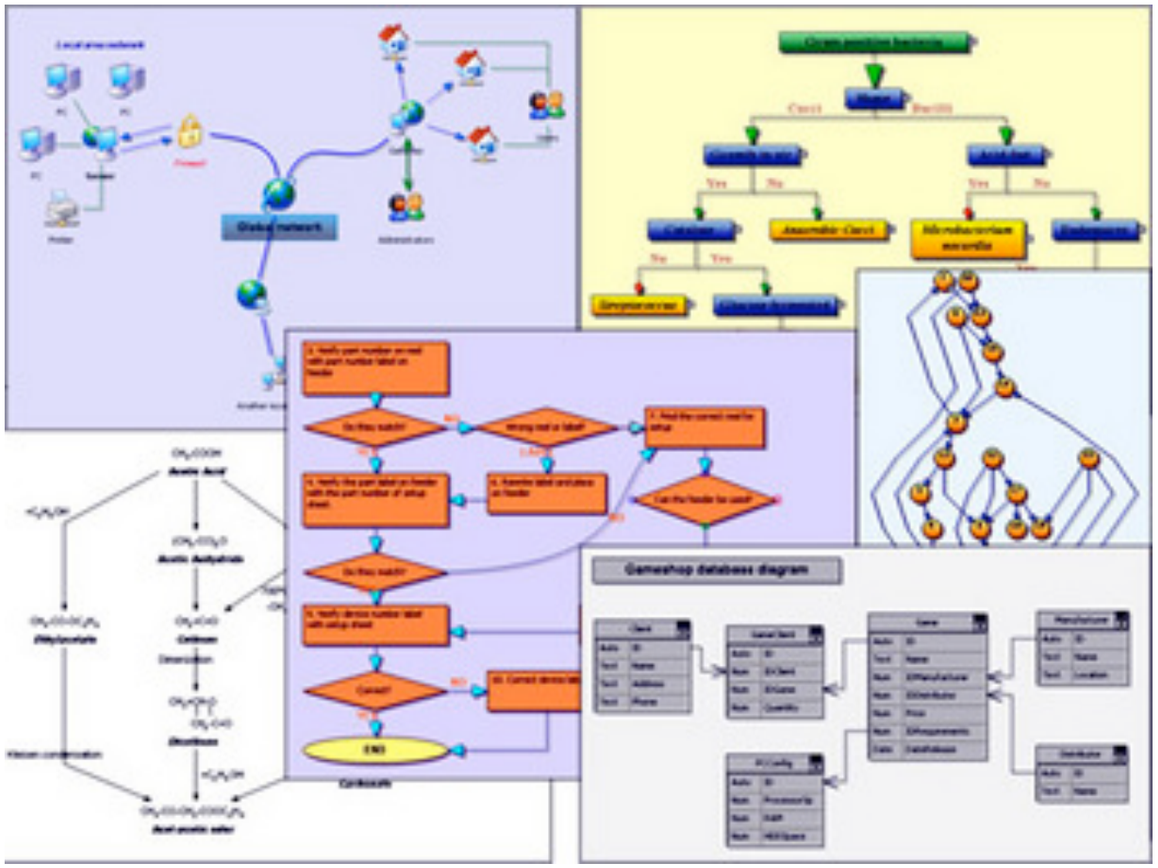
Intérêt

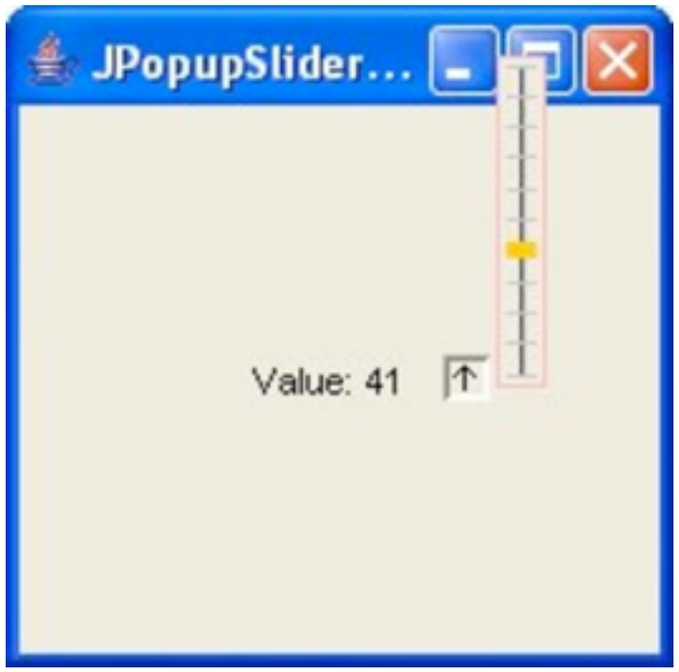
Besoin de composants qui n'existent pas par défaut: calendriers…

Contrairement à AWT, Swing prend en charge complètement l'affichage des composants

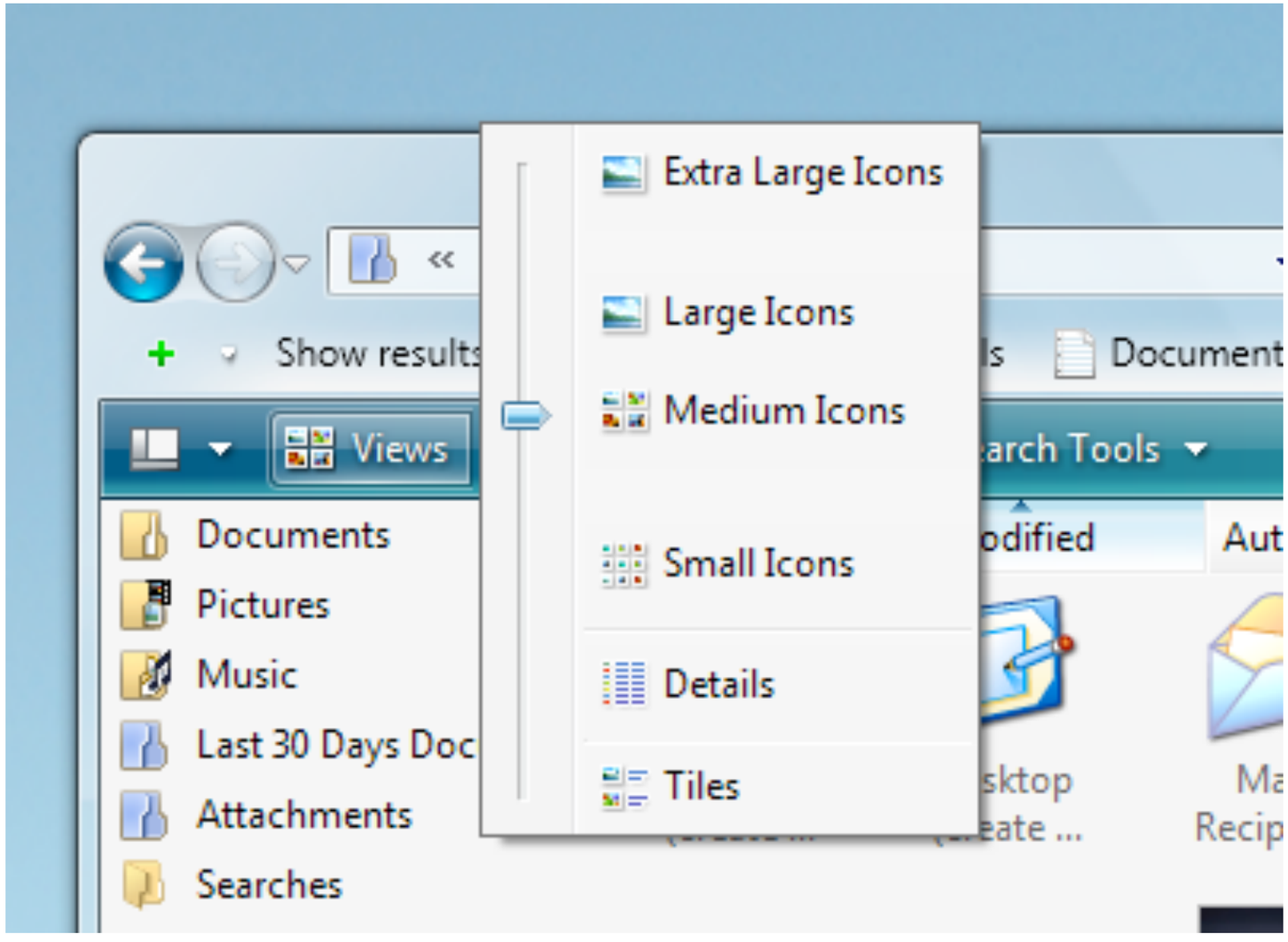
Possibilité de créer des composants indépendants d'une plateforme

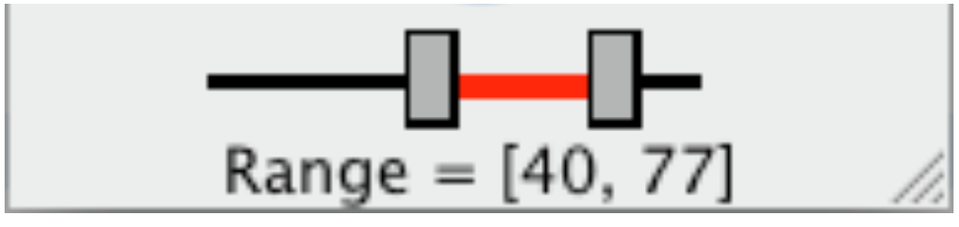
Exemples de nouveaux composants





Exemples de nouveaux composants

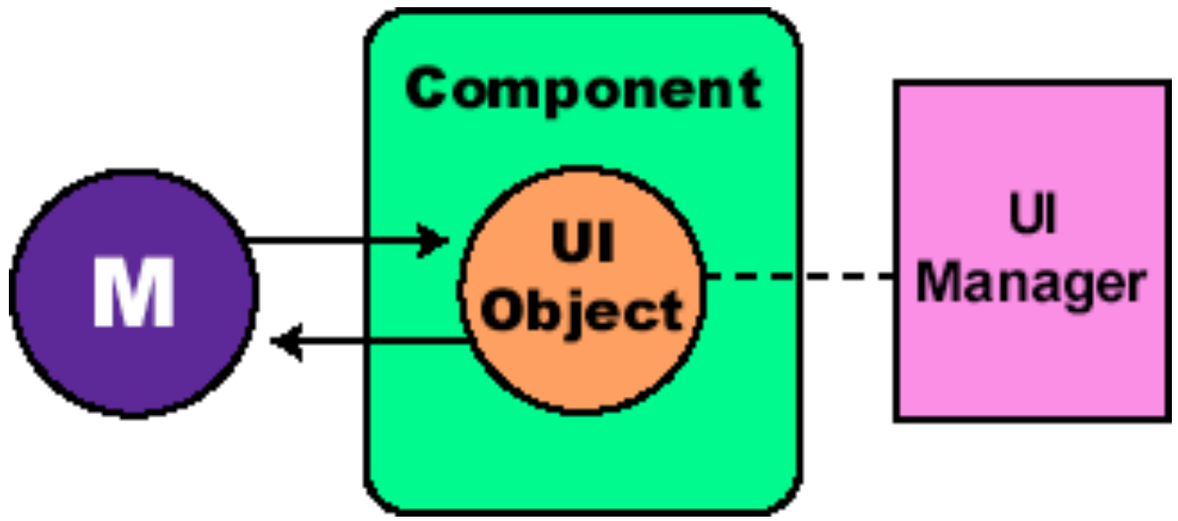




Règles générales

Respecter l'architecture Swing

Respecter le modèle MVC modifié de Swing



Permet une intégration facile avec les JavaBeans

UI delegate

Correspond aux parties Vue et Contrôleur de MVC

Le modèle reste séparé

La superclasse de toutes les classes UI Delegate est swing.plaf.ComponentUI

Permet de faire fonctionner le Plugable look-and-feel

Gère l'installation et la désinstallation d'une UI ainsi que l'affichage de la géométrie

JComponent invoque des méthodes de cette classe pour déléguer l'affichage, les calculs de Layout qui peuvent varier suivant les look-and-feel installés…

UI delegate

Installation/désinstallation d'UI

- public void installUI(JComponent c)
- public void uninstallUI(JComponent c)

Jcomponent.setUI():

```
protected void setUI(ComponentUI newUI) {
    if (ui != null) {
        ui.uninstallUI(this);
    }
    ComponentUI oldUI = ui;
    ui = newUI;
    if (ui != null) {
        ui.installUI(this);
    }
    invalidate();
    firePropertyChange("UI", oldUI, newUI);
}
```

ComponentUI

javax.swing.plaf

Class ComponentUI

java.lang.Object
  └ **javax.swing.plaf.ComponentUI**

Direct Known Subclasses:

ButtonUI, ColorChooserUI, ComboBoxUI, DesktopIconUI, DesktopPaneUI, FileChooserUI, InternalFrameUI, LabelUI, ListUI, MenuBarUI, OptionPaneUI, PanelUI, PopupMenuUI, ProgressBarUI, RootPaneUI, ScrollBarUI, ScrollPaneUI, SeparatorUI, SliderUI, SpinnerUI, SplitPaneUI, TabbedPaneUI, TableHeaderUI, TableUI, TextUI, ToolBarUI, ToolTipUI, TreeUI, ViewportUI

---

public abstract class **ComponentUI**
extends Object

The base class for all UI delegate objects in the Swing pluggable look and feel architecture. The UI delegate object for a Swing component is responsible for implementing the aspects of the component that depend on the l and feel. The `JComponent` class invokes methods from this class in order to delegate operations (painting, layout calculations, etc.) that may vary depending on the look and feel installed. **Client programs should not invo methods on this class directly.**

See Also:

`JComponent`, `UIManager`

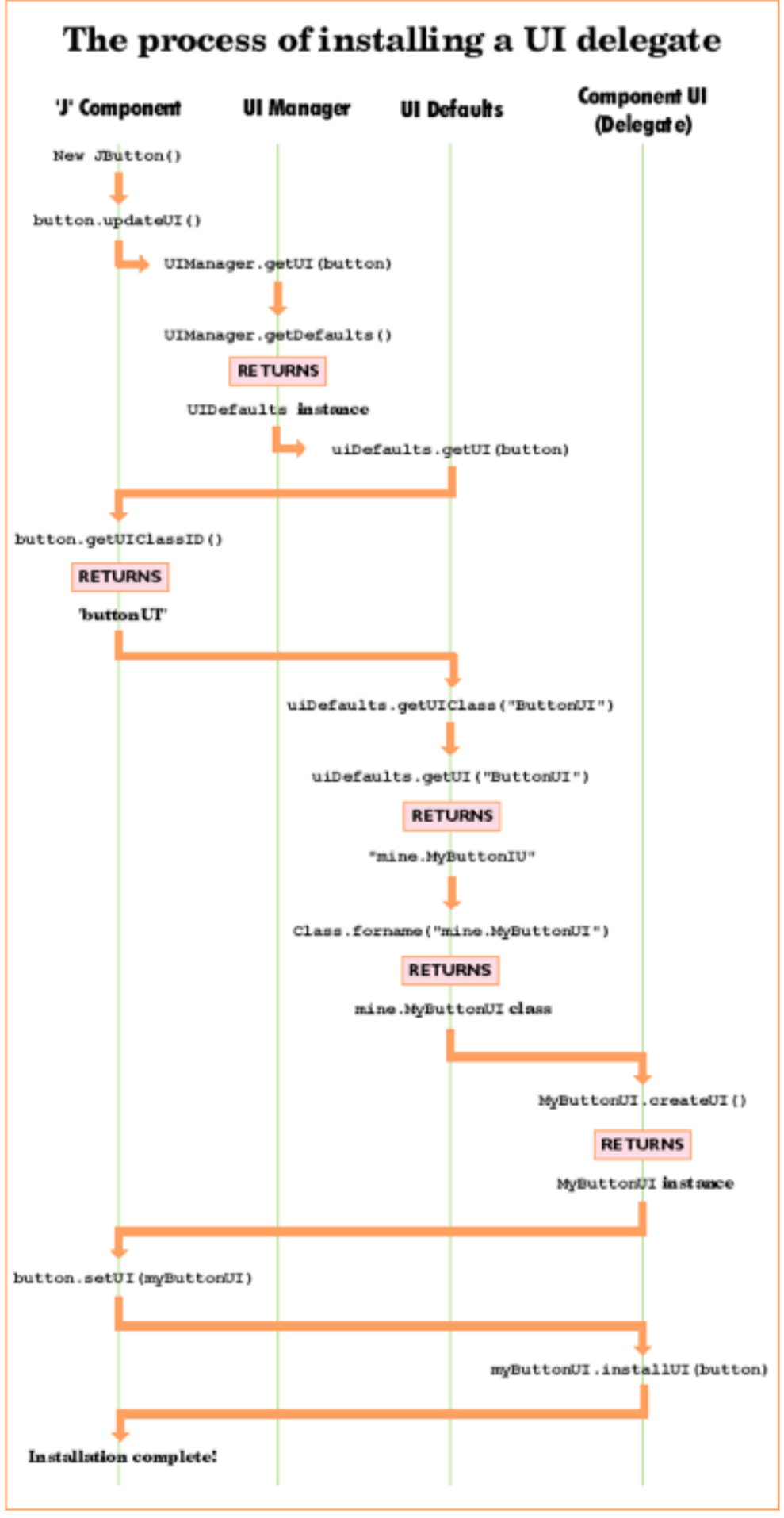
Constructor Summary

ComponentUI()	Sole constructor.

Method Summary

boolean	**contains**(JComponent c, int x, int y) Returns `true` if the specified *x,y* location is contained within the look and feel's defined shape of the specified component.
static ComponentUI	**createUI**(JComponent c) Returns an instance of the UI delegate for the specified component.
Accessible	**getAccessibleChild**(JComponent c, int i) Returns the ith `Accessible` child of the object.
int	**getAccessibleChildrenCount**(JComponent c) Returns the number of accessible children in the object.
Dimension	**getMaximumSize**(JComponent c) Returns the specified component's maximum size appropriate for the look and feel.
Dimension	**getMinimumSize**(JComponent c) Returns the specified component's minimum size appropriate for the look and feel.
Dimension	**getPreferredSize**(JComponent c) Returns the specified component's preferred size appropriate for the look and feel.
void	**installUI**(JComponent c) Configures the specified component appropriate for the look and feel.

UI delegate


The process of installing a UI delegate
'J' Component
UI Manager
UI Defaults
Component UI (Delegate)
New JButton()
button.updateUI()
UIManager.getUI(button)
UIManager.getDefaults()
RETURNS
UIDefaults instance
uiDefaults.getUI(button)
button.getUIClassID()
RETURNS
'buttonUI'
uiDefaults.getUIClass("ButtonUI")
uiDefaults.getUI("ButtonUI")
RETURNS
"mine.MyButtonIU"
Class.forname("mine.MyButtonUI")
RETURNS
mine.MyButtonUI class
MyButtonUI.createUI()
RETURNS
MyButtonUI instance
button.setUI(myButtonUI)
myButtonUI.installUI(button)
Installation complete!

UI delegate

installUI est responsable de:

- Définir la police, couleur, bordure, et propriétés d'opacité du composant
- Installer le layout manager approprié au composant
- Installer les sous-composants du composant
- Enregistrer les event listeners sur le composant
- Enregistrer les raccourcis clavier spécifiques au look-and-feel
- Enregistrer les listeners du modèle du composant pour être notifié de mettre à jour l'affichage
- Initialiser les données nécessaires

UI delegate

Exemple pour une extension de ButtonUI

```
protected MyMouseListener mouseListener;
protected MyChangeListener changeListener;

public void installUI(JComponent c) {
   AbstractButton b = (AbstractButton)c;

   // Install default colors & opacity
   Color bg = c.getBackground();
   if (bg == null || bg instanceof UIResource) {
      c.setBackground(
          UIManager.getColor("Button.background"));
   }
   Color fg = c.getForeground();
   if (fg == null || fg instanceof UIResource) {
      c.setForeground(
          UIManager.getColor("Button.foreground"));
   }
   c.setOpaque(false);

         // Install listeners
   mouseListener = new MyMouseListener();
   c.addMouseListener(mouseListener);
   c.addMouseMotionListener(mouseListener);
   changeListener = new MyChangeListener();
   b.addChangeListener(changeListener);
}
```

UI delegate

Règles pour initialiser les propriétés

- Toutes les valeurs utilisées pour définir les couleurs, police, bordures etc doivent être obtenues de la Defaults table (table de hachage contenant les propriétés par défaut de couleurs (en fonction du look-and–feel)…): UIManager.getColor()
- Les couleurs, polices … doivent être spécifiées seulement si l’application ne les a pas déjà fixées: toujours vérifier que la propriété n’est pas à la valeur null

UI delegate

La désinstallation doit rétablir toutes les propriétés modifiées par l'installation pour préparer le terrain à un nouveau UIdelegate

```
public void uninstallUI(JComponent c) {
    AbstractButton b = (AbstractButton)c;

    // Uninstall listeners
    c.removeMouseListener(mouseListener);
    c.removeMouseMotionListener(mouseListener);
    mouseListener = null;
    b.removeChangeListener(changeListener);
    changeListener = null;
}
```

Définition de la géométrie

Le LayoutManager a besoin de connaître la prefferedSize de chaque composant (et minimumSize, maximumSize) suivant l'algorithme

```
public Dimension
    getPreferredSize(JComponent c)
public Dimension
    getMinimumSize(JComponent c)
public Dimension
    getMaximumSize(JComponent c)
public boolean
    contains(JComponent c, int x, int y)
```

Définition de la géométrie

JComponent.getPreferredSize()

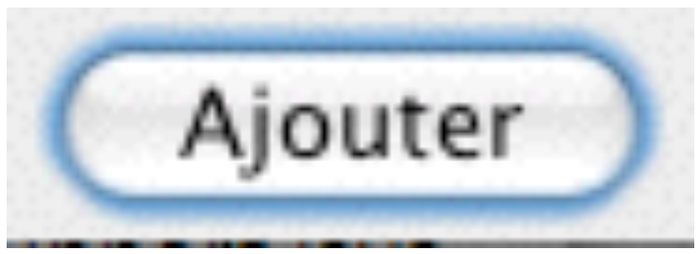
```
public Dimension getPreferredSize() {
    if (preferredSize != null) {
        return preferredSize;
    }
    Dimension size = null;
    if (ui != null) {
        size = ui.getPreferredSize(this);
    }
    return (size != null) ? size :
            super.getPreferredSize();
}
```

Définition de la géométrie

Possibilité d'avoir des composants non rectangulaires (ex: boutons avec coins arrondis)

JComponent.contains()

```
public boolean contains(JComponent c, int x, int y) {
   return (ui != null) ?
         ui.contains(this, x, y) :
               super.contains(x, y);
}
```



Affichage

public void paint(Graphics g, JComponent c)

public void update(Graphics g, JComponent c)

- Efface l'écran (si objet opaque) et affiche (paint)

JComponent.paintComponent():

```
protected void paintComponent(Graphics g) {
   if (ui != null) {
      Graphics scratchGraphics =
         SwingGraphics.createSwingGraphics(g.create());
      try {
         ui.update(scratchGraphics, this);
      }
      finally {
         scratchGraphics.dispose();
      }
   }
}
```

Différentes parties

3 parties à créer:

- La classe du composant qui fournit l'API pour créer, modifier et obtenir l'état de base du composant
- L'interface du modèle et l'implémentation par défaut du modèle qui prend en charge la logique du composant et les notifications
- L'UI delegate qui prend en charge le placement des différentes parties du composant, la gestion des événements (clavier et souris) et l'affichage

Exemple de JSlider

La classe JSlider définit l'API pour créer, modifier et obtenir l'état de base du composant (~1200 lignes)

BoundedRangeModel (~250 lignes) fournit l'interface pour le modèle implémentée par DefaultBoundedRangeModel (~250 lignes)

La classe BasicSliderUI définit l'UI delegate(~1800 lignes)

Exemple de JSlider

Affichage de la piste

```
public void paintTrack(Graphics g)  {
    int cx, cy, cw, ch;
    int pad;

    Rectangle trackBounds = trackRect;

    if ( slider.getOrientation() == JSlider.HORIZONTAL ) {
        pad = trackBuffer;
        cx = pad;
        cy = (trackBounds.height / 2) - 2;
        cw = trackBounds.width;

        g.translate(trackBounds.x, trackBounds.y + cy);

        g.setColor(getShadowColor());
        g.drawLine(0, 0, cw - 1, 0);
        g.drawLine(0, 1, 0, 2);
        g.setColor(getHighlightColor());
        g.drawLine(0, 3, cw, 3);
        g.drawLine(cw, 0, cw, 3);
        g.setColor(Color.black);
        g.drawLine(1, 1, cw-2, 1);

        g.translate(-trackBounds.x, -(trackBounds.y + cy));
}
```

Exemple de JSlider

Mise à jour du modèle lors du drag du slider:

```
/**
* Set the models value to the position of the top/left
* of the thumb relative to the origin of the track.
*/
public void mouseDragged(MouseEvent e) {
    int thumbMiddle = 0;

    if (!slider.isEnabled()) {
        return;
    }

    currentMouseX = e.getX();
    currentMouseY = e.getY();

    if (!isDragging) {
        return;
    }

    slider.setValueIsAdjusting(true);

    switch (slider.getOrientation()) {
    case JSlider.VERTICAL:
        int halfThumbHeight = thumbRect.height / 2;
        int thumbTop = e.getY() - offset;
        int trackTop = trackRect.y;
        int trackBottom = trackRect.y + (trackRect.height - 1);
        int vMax = yPositionForValue(slider.getMaximum() -
                                     slider.getExtent());

        if (drawInverted()) {
```