Architecture d'application graphique

NFA035

S. Rosmorduc

Le MVP (modèle, vue, présentateur)

- Plus simple que MVC
- Moins adapté si on a beaucoup de vues *du même modèle*
- pratique pour les formulaires

Modèle/Vue/ Présentateur

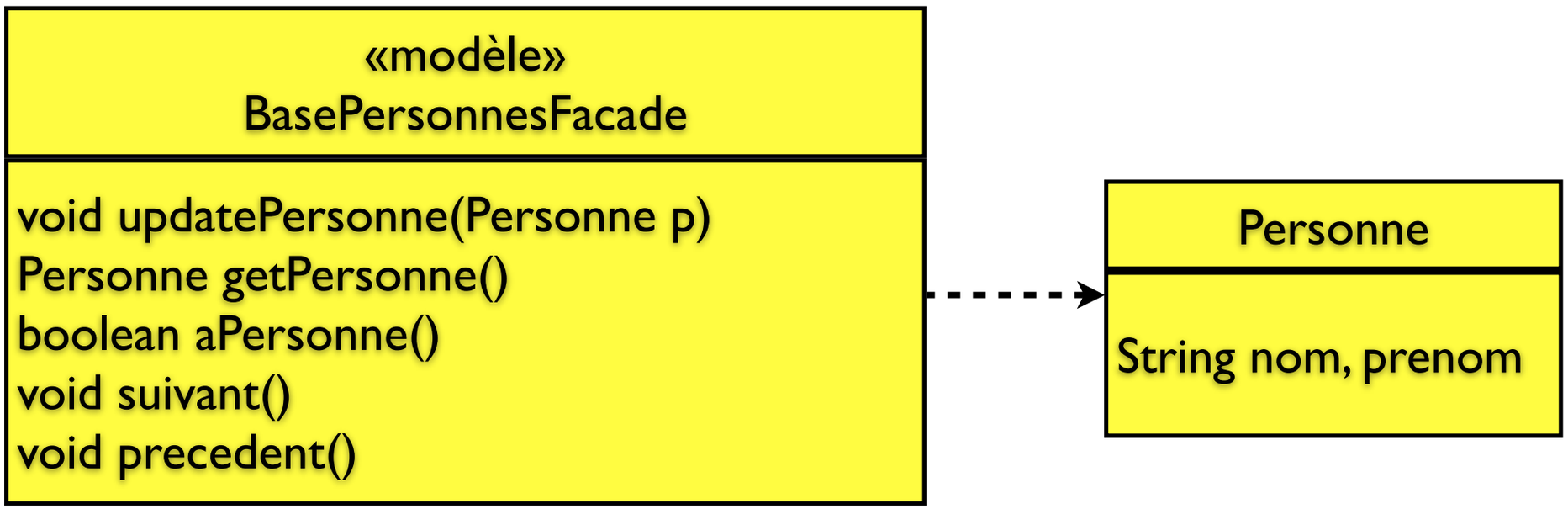
- Le modèle décrit les données
- La vue les affiche
- Le présentateur reçoit les événements, les traite (en modifiant le modèle), **et met à jour les vues**
- Différence avec MVC: c'est le présentateur qui rafraîchit les objets graphiques

Le MVC

- (déjà vu)
- facile à mettre en œuvre au niveau des composants
- plus délicat (mais possible) à utiliser pour la totalité des données d'une application
- En pratique, il peut arriver qu'on mélange les styles dans une même application

MVP sur un formulaire

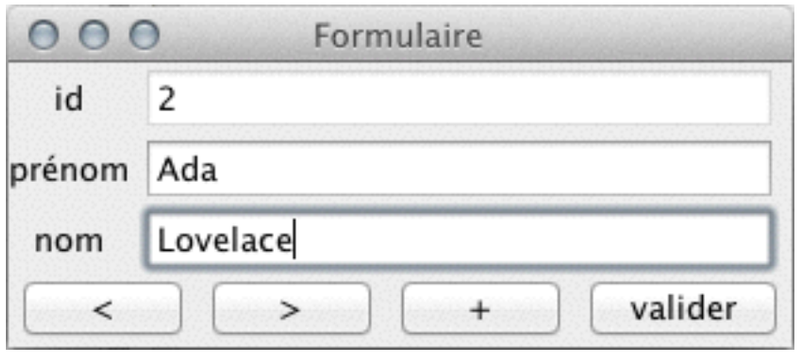
- Exemple: une application qui parcourt une base de données de personnes et permet de modifier les données associées



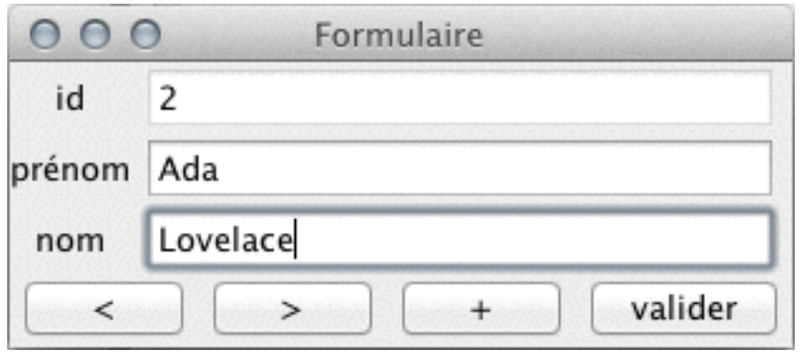
- Note: pour simplifier l'architecture, les objets Personne sont des *valeurs*. Pour mettre à jour les données d'une personne dans le modèle, il faut appeler updatePersonne().

- Le modèle est ici un **modèle de l'application** (avec la notion de navigation dans la liste)

Exemple MVP

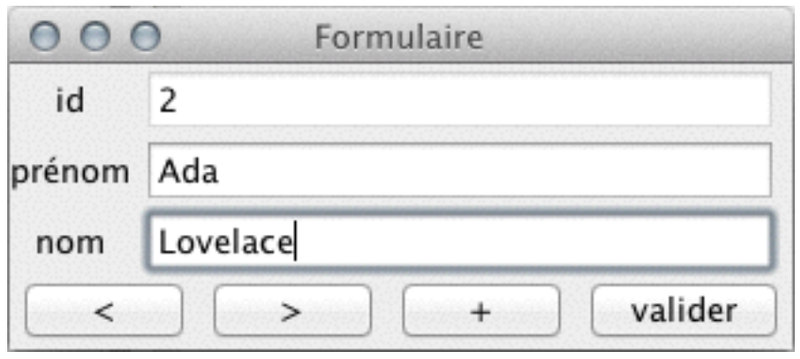


MVP



- On presse sur le bouton «suivant»
- le présentateur
  - appelle la méthode «suivant» du modèle
  - récupère la personne correspondante
  - copie les données de la personne dans les champs du formulaire

MVP



- On presse sur le bouton «mettre à jour»
- Le présentateur
  - récupère les données nom et prénom dans les champs du formulaire
  - appelle la méthode updatePersonne() du modèle pour mettre à jour celui-ci.

Vue...

```
public class JPersonneFormulaire {
  private JPanel panel= new JPanel();
  private JButton suivantButton= new JButton(">");
  private JButton precedentButton= new JButton("<");
  private JButton ajouterButton= new JButton("+");
  private JButton validerButton= new JButton("valider");
  private JTextField idField= new JTextField(20);
  private JTextField nomField= new JTextField(20);
  private JTextField prenomField= new JTextField(20);

  public JPersonneFormulaire() {
   idField.setEditable(false);
   mettreEnPage();
  }
  ... getters...
}
```

```
public class PersonnesFacade {
  // Invariant pour cet exemple :
  //il y a toujours une personne à la position "position".
  private ArrayList<Personne> personnes= new ArrayList<Personne>();
  private int position= 0;
  public PersonnesFacade() {
   personnes.add(new Personne(1, "", ""));
  }
  public Personne getPersonne() {
   return personnes.get(position);

  }
```

```
/**
 * Ajoute une entrée après l'entrée courante...
 */
public void ajouter() {
 personnes.add(position+1,
   new Personne(personnes.size() + 1, "", ""));
 suivant();
}

public void suivant() {
 if (position < personnes.size() -1 )
    position++;
}
```

Modèle

```
public class PersonnesFacade {
  ...

  public void precedent() {
   if (position > 0)
      position--;
  }

  public void mettreAJour(Personne nouveau) {
   // On peut directement utiliser "nouveau",
     // car les objets personnes sont immuables.
   // pas de risque qu'on le modifie derrière notre dos.
   personnes.set(position, nouveau);
  }

}
```

Modèle

Présentateur

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;

  public PersonnesPresentateur(JPersonneFormulaire vue,
                       PersonnesFacade modele) {
   this.vue = vue;
   this.modele = modele;
   activer();
   charger();
}
/**
 * Copie les données de la  personne "courante" vers
 * le formulaire
 */
public void charger() {
 Personne p = modele.getPersonne();
 vue.getIdField().setText("" + p.getId());
 vue.getNomField().setText(p.getNom());
 vue.getPrenomField().setText(p.getPrenom());
}
```

Présentateur

```
public class PersonnesPresentateur {
  ........
  private void activer() {
   vue.getAjouterButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "ajouter"));
   vue.getPrecedentButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "precedent"));
   vue.getSuivantButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "suivant"));
 vue.getValiderButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "valider"));
}

/**
 * Passe à la personne suivante (si possible).
 */
public void suivant() {
 modele.suivant(); // on modifie le modèle...
 charger(); // on met à jour l'affichage
}
```

Présentateur

```
public class PersonnesPresentateur {
  ........


  /**
   * Met à jour la personne courante.
   */
  public void valider() {
     // récupération des données depuis la vue
  String pr=   vue.getPrenomField().getText()
  String n=   vue.getNomField().getText(),
  Personne ancien = modele.getPersonne();
  Personne nouveau =
     new Personne(ancien.getId(), n, pr);
     // modification des données dans le modèle
  modele.mettreAJour(nouveau);
     // mise à jour de la vue
  charger();  }

}
```

MVC «pur»

- Toute modification du modèle déclenche un événement
- les champs du formulaire ont des modèles (de type Document) qui «écoutent» eux-même ces événements et se mettent à jour automatiquement
- Attention cependant : les données du formulaire ne sont pas toujours les données du modèle. Par exemple, tant que je n'ai pas validé ma saisie, les modifications ne sont pas prises en compte.

Un peu plus de patterns

pattern commande

- **Problème** : on veut représenter explicitement une action dans l'application, pour pouvoir la manipuler:
  - la lier à plusieurs composants graphiques
  - l'activer/la désactiver
  - pouvoir éventuellement gérer un historique des action (fonction undo/redo)

Pattern Commande

- Solution : réifier l'action (de *res*, «chose» en latin).
- En clair: représenter l'action par un objet.
- Deux principales variantes:
  - on représente une action avec ses données contextuelles associées (fonction undo) : chaque exécution de l'action crée une instance.
  - on représente l'action «en général» (menu, etc...) : l'action est représentée par un seul objet. Cas de l'interface Action.

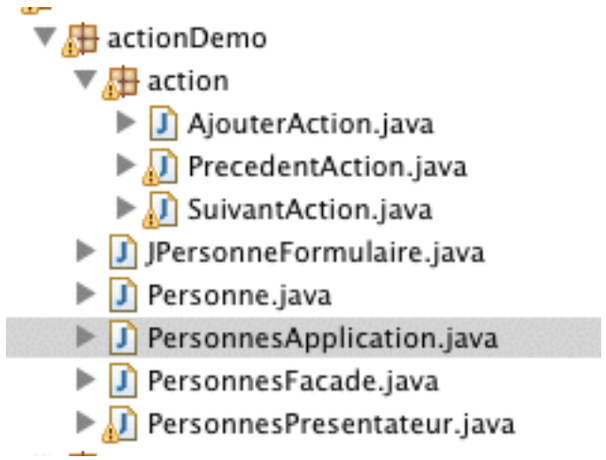
L'interface Action et la classe AbstractAction

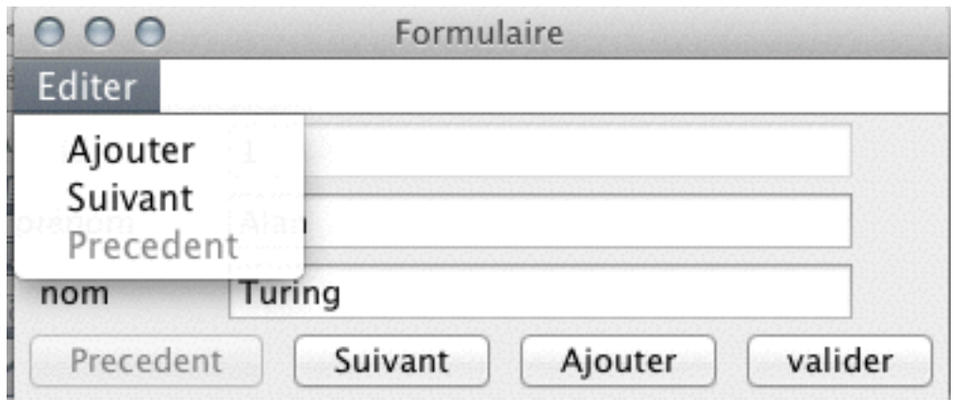
- Une action est un actionlistener qui a de l'ambition :
- elle peut être activée/désactivée (isEnabled; setEnabled)
- elle a des propriétés (par exemple pour lui ajouter des raccourcis, des icônes, des toolstips)
- On utilise généralement AbstractAction pour travailler.

AbstractAction

- Constructeurs:
  - AbstractAction()
  - AbstractAction(String name)
  - AbstractAction(String name, Icon icon)
- permet de dire quelle texte/icône seront attachés aux menus, boutons... qui utilisent l'action.

Utilisation





Actions...

```
@SuppressWarnings("serial")
public class AjouterAction extends AbstractAction {
  public static final String ID = "AJOUTER";
  PersonnesPresentateur presentateur;

  public AjouterAction(PersonnesPresentateur presentateur) {
     // Label de l'action (boutons, menus...)
   super("Ajouter");
   this.presentateur= presentateur;
      // Propriété : raccourci clavier.
   putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke("control a"));
  }

  @Override
  public void actionPerformed(ActionEvent ev) {
   presentateur.ajouter();
  }
}
```

Actions

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;
  private HashMap<String, Action> actMap =
                                      new HashMap<String, Action>();

  public PersonnesPresentateur(JPersonneFormulaire vue,
                               PersonnesFacade modele) {
   this.vue = vue;
   this.modele = modele;
   creerActions(); activer(); charger();
  }

  private void creerActions() {
   actMap.put(AjouterAction.ID, new AjouterAction(this));
   actMap.put(SuivantAction.ID, new SuivantAction(this));
   actMap.put(PrecedentAction.ID, new PrecedentAction(this));
   ...
  }
  private void activer() {
    vue.getAjouterButton().setAction(actMap.get(AjouterAction.ID));

vue.getPrecedentButton().setAction(actMap.get(PrecedentAction.ID));
    vue.getSuivantButton().setAction(actMap.get(SuivantAction.ID));
    ...
  }
```

Actions

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;
  private HashMap<String, Action> actMap =
                                    new HashMap<String, Action>();

  ....
  /**
   * Charge la personne "courante".
   */
public void charger() {
 Personne p = modele.getPersonne();
 vue.getIdField().setText("" + p.getId());
 vue.getNomField().setText(p.getNom());
 vue.getPrenomField().setText(p.getPrenom());
 mettreAjourActions();
}

private void mettreAjourActions() {
 actMap.get(SuivantAction.ID).setEnabled(modele.aSuivant());
 actMap.get(PrecedentAction.ID).setEnabled(modele.aPrecedent());

}
```

Actions *(et menus)*

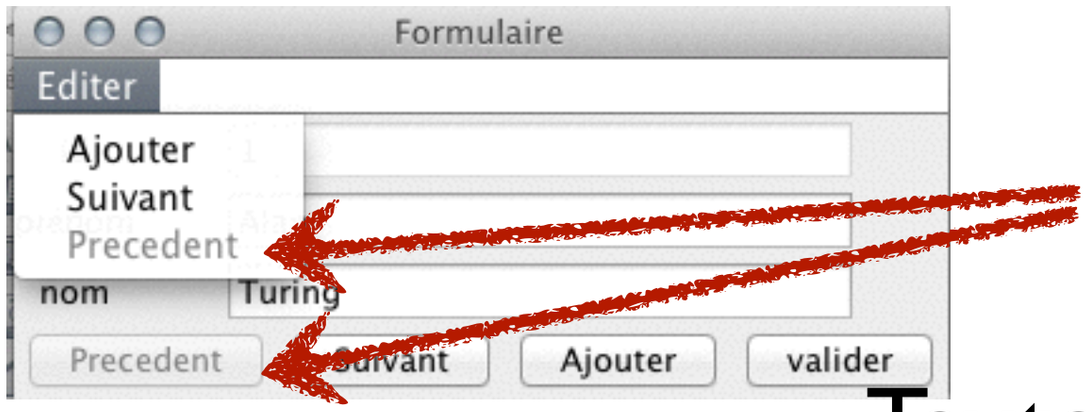
```
public class PersonnesApplication {
  private PersonnesFacade modele;
  private JPersonneFormulaire vue;
  private PersonnesPresentateur presentateur;
  private JFrame frame;

  public PersonnesApplication() {
   ....
   frame= new JFrame("Formulaire");
   frame.add(vue.getPanel());
   creerMenu();
   frame.pack();
   frame.setVisible(true);
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  private void creerMenu() {
   JMenuBar menuBar= new JMenuBar();
   JMenu menu= new JMenu("Editer");
   menu.add(presentateur.getAction(AjouterAction.ID));
   menu.add(presentateur.getAction(SuivantAction.ID));
   menu.add(presentateur.getAction(PrecedentAction.ID));

   menuBar.add(menu);
   frame.setJMenuBar(menuBar);
  }
```

Actions (et menus)



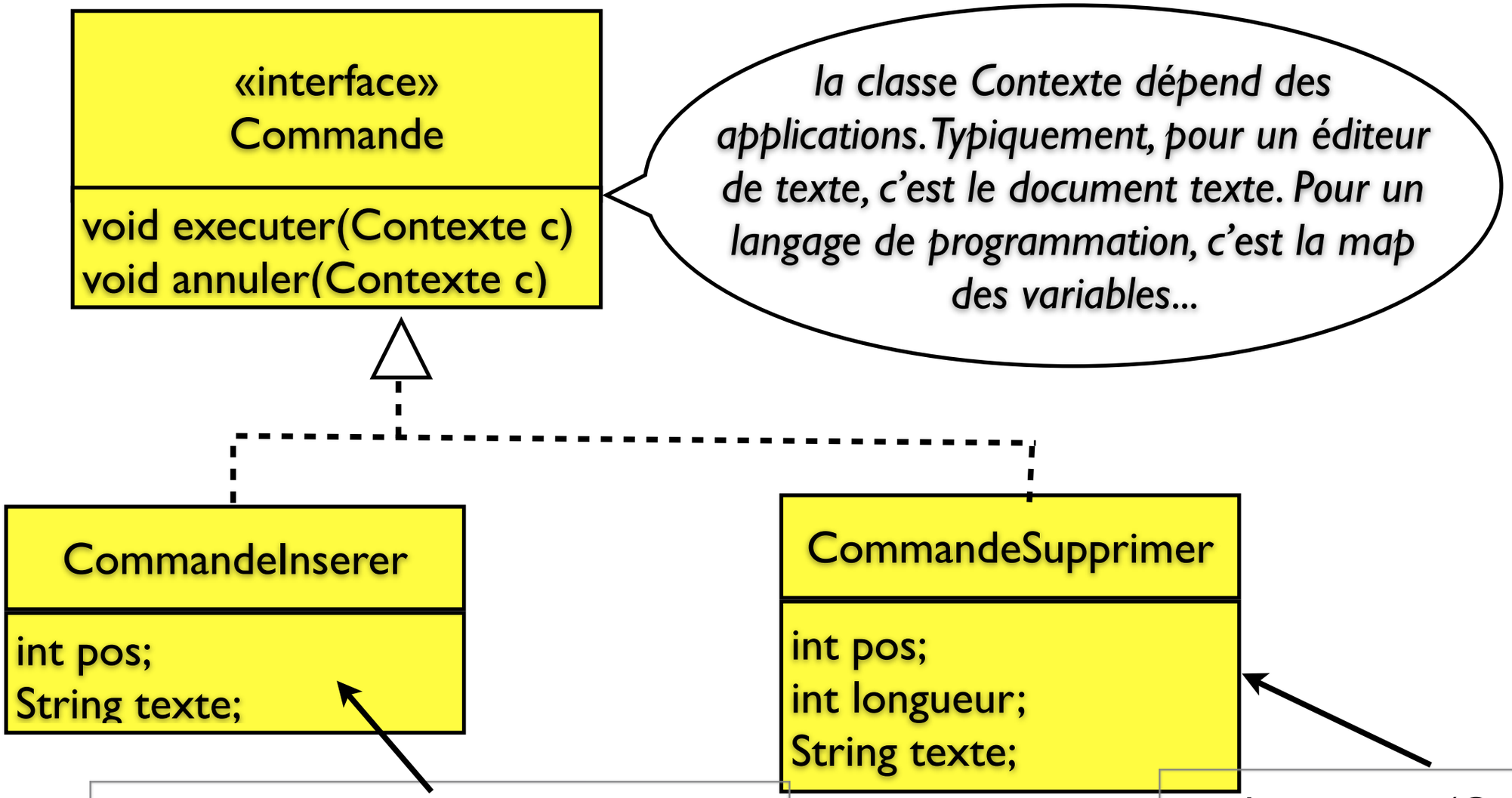
même objet
action, même
état

Texte

Pattern commande et gestion de la fonction undo

- Idée : on stocke dans une commande les informations nécessaires pour l'annuler
- Exemple: une commande qui supprime du texte va stocker le texte supprimé et sa position
- pour annuler la commande, il suffira de remettre le texte en question

Pattern commande




```
void  executer(Contexte ctx) {
    ctx.inserer(pos, texte);
}
void annuler(Contexte ctx) {
    ctx.supprimer(pos, texte.length);
}
```

```
void  executer(Contexte ctx) {
    texte= ctx.getTexte(pos, longueur);
    ctx.supprimer(pos, longueur);
}
void annuler(Contexte ctx) {
    ctx.inserer(pos, texte);
}
```