ÉTS
Le génie pour l'industrie

LOG121

Conception orientée objet

Patron Mémento et retour sur MVC

Enseignante: Souad Hadjres

Plan

- Patron Mémento
- Retour sur l'architecture MVC

Exemple de problème de conception

- Un gestionnaire simplifié de transaction permet d'exécuter des transactions sur les tables d'une base de données (BD)
- Une transaction peut contenir plusieurs opérations (update, delete, insert) agissant sur les tables de la BD
  - Exemple: un client qui transfère de l'argent de son compte courant à un compte d'épargne
    - Cette transaction doit mettre à jour les deux comptes en conséquence et enregistrer la transaction dans un journal

Exemple de problème de conception

- Notre exemple de transaction aura la forme:

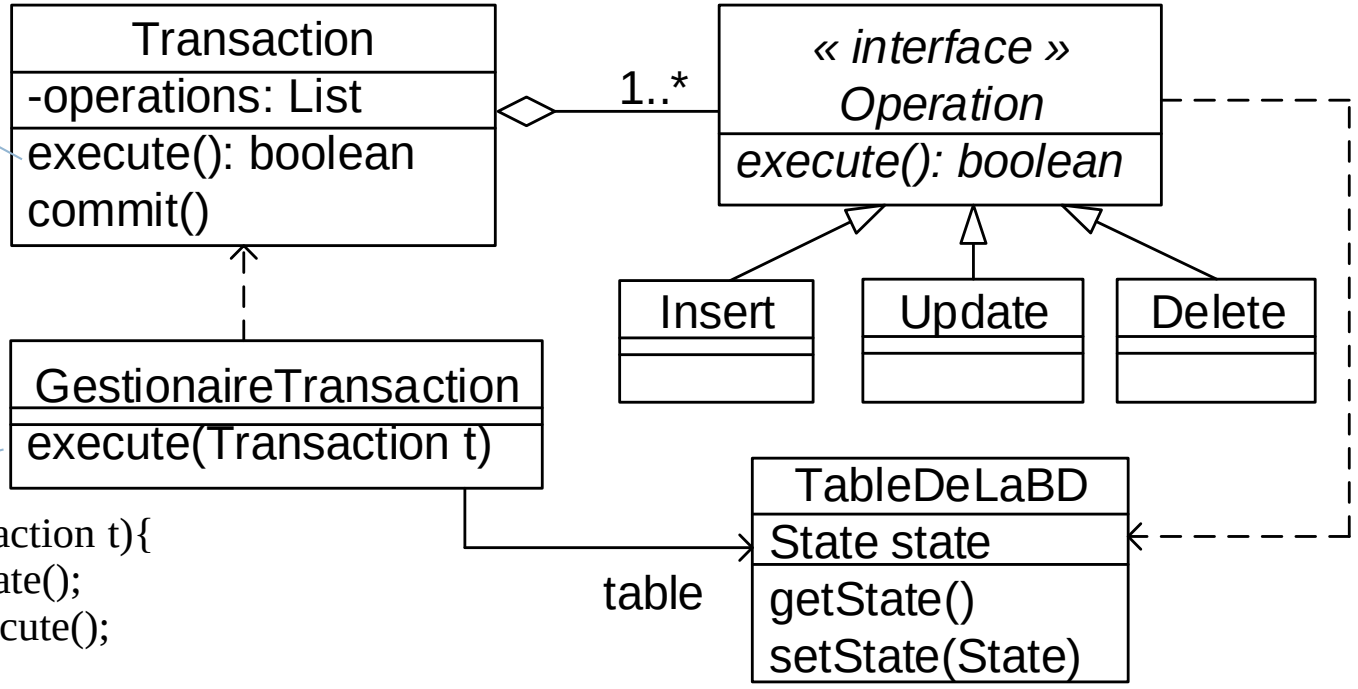
```
UPDATE compte-courant
    SET solde = solde – 50
    WHERE num-compte = 81566979
UPDATE compte-epargne
    SET solde = solde + 50
    WHERE num-compte = 81566988
INSERT INTO journal-transactions VALUES
    (num_transaction, type_transaction, 81566979,
    81566988, 50)
COMMIT WORK;
```

Exemple de problème de conception

- Une transaction a la propriété d'être ACID
  - A: Atomique, C: Cohérence, I: Isolée et D: Durable
  - Si toutes les opérations d'une transaction sont exécutées avec succès, le gestionnaire appelle la méthode « commit » de la transaction
  - Si une opération échoue, toute la transaction doit être annulée. Dans ce cas, les tables concernées par la transaction doivent retourner à leur état avant le début de la transaction

Exemple de problème de conception

Cette méthode fait une boucle sur toutes les operations dans la liste « operations » pour appeler leur méthode execute(). Si chacune des opérations s'exécute avec succès, elle retourne « true ».



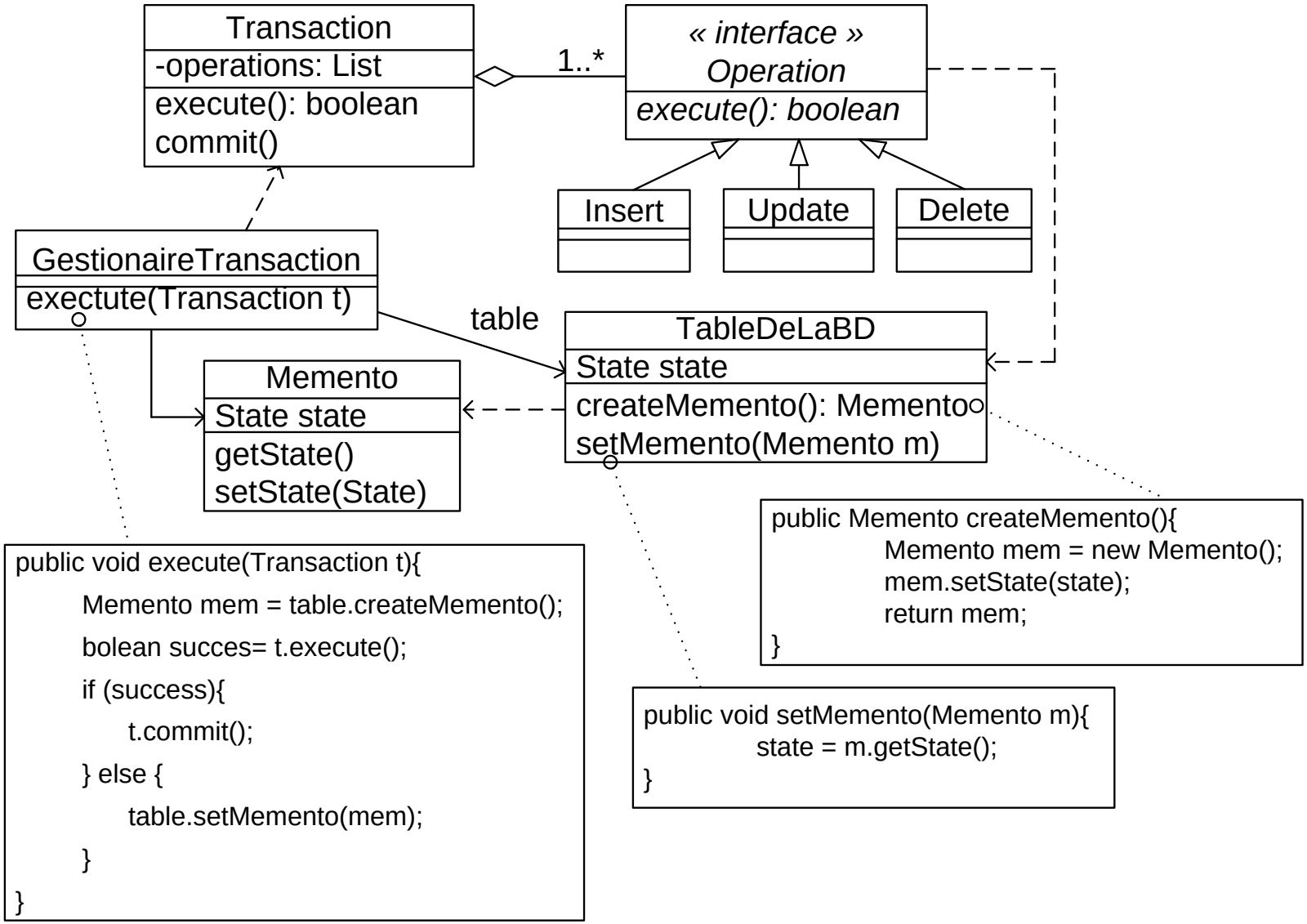

```
public void execute(Transaction t){
    State old = table.getState();
    boolean succes = t.execute();
    if (success) {
        t.commit();
    } else {
        table.setState(old);
    }
}
```

- Quel est le problème avec cette conception?

Solution au problème

- On crée un objet (Memento) qui garde trace de l'état d'un objet de type TableDeLaBD.
- Avant d'exécuter une transaction, le gestionnaire de transaction demande à un objet TableDeLaBD son objet Memento.
- Un objet TableDeLaBD est le seul responsable de créer son Memento, de l'initialiser et de le mettre à jour pour qu'il garde une copie de l'état courant de la table.
- Lorsqu'une transaction échoue, le gestionnaire de transaction renvoie le Memento à l'objet TableDeLaBD
- L'objet TableDeLaBD utilise le Memento reçu pour récupérer l'information sur son état précédent.

Solution au problème



Patron Mémento

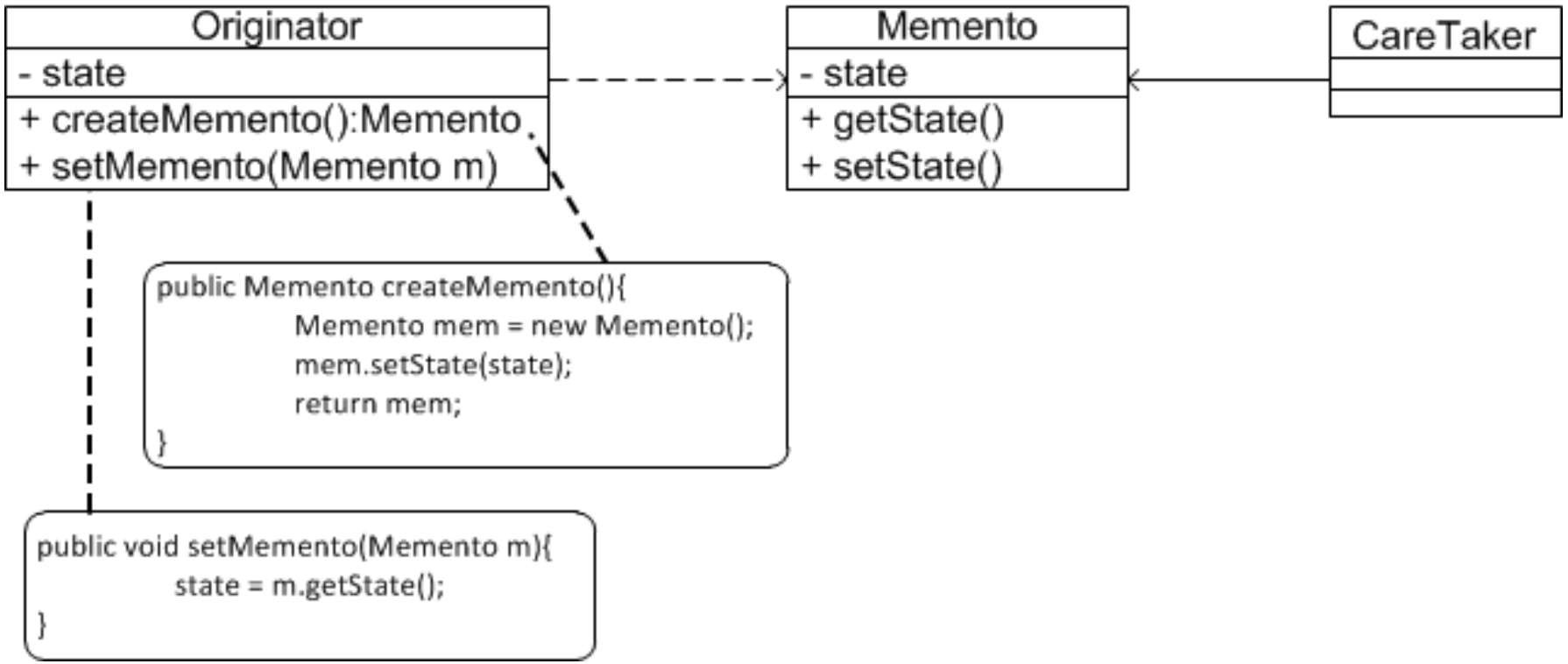
- Contexte
  - L'état (ou une partie de l'état) d'un objet doit être sauvegardé pour que l'objet puisse retourner à cet état en cas de besoin
  - Un accès direct à l'état de l'objet exposerait des détails d'implémentation et violerait l'encapsulation de l'objet

Patron Mémento

- Solution
  - On crée un objet Memento (aussi appelé Snapshot) qui garde trace de l'état de l'objet initial (Originator). Selon le contexte, le Memento peut garder juste une partie de l'état de l'objet initial.
  - L'objet Originator est le seul responsable de créer son Memento, de l'initialiser et de le mettre à jour.
  - Un objet (Caretaker) qui utilise l'objet original est responsable de garder une référence sur le Memento et renvoie le Memento à l'objet original en cas de besoin.
  - L'objet Caretaker ne change pas l'objet Memento.

Patron Mémento

- La structure du patron dans GoF



Patron Mémento

Avantages:

- garde l'état sauvegardé d'un objet à l'extérieur de celui-ci
- Assure l'encapsulation des données de l'objet et fournit un moyen simple de les récupérer

Mais:

- les opérations de sauvegarde et de restauration de l'état peuvent consommer beaucoup de temps et d'espace.

Patron Mémento


```java
public class Originator {
      private String state;

      public void setState(String state){
         this.state = state;
      }

      public String getState(){
         return state;
      }

      public Memento saveStateToMemento(){
         return new Memento(state);
      }

      public void getStateFromMemento(Memento memento){
         state = memento.getState();
      }
 }
```

```java
public class Memento {
      private String state;

      public Memento(String state){
         this.state = state;
      }

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

```java
public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

```java
public class MementoTest {
      public static void main(String[] args) {

         Originator originator = new Originator();
         CareTaker careTaker = new CareTaker();

         originator.setState("State #1");
         originator.setState("State #2");
         careTaker.add(originator.saveStateToMemento());

         originator.setState("State #3");
         careTaker.add(originator.saveStateToMemento());

         originator.setState("State #4");
         System.out.println("Current State: " + originator.getState());

         originator.getStateFromMemento(careTaker.get(0));
         System.out.println("First saved State: " + originator.getState());
         originator.getStateFromMemento(careTaker.get(1));
         System.out.println("Second saved State: " + originator.getState());
      }
}
```

Patrons Mémento et Commande

- Que peuvent-ils accomplir ensemble ?

l'implémentation du « undo » d'une commande n'est pas possible ➔enregistrer l'état avant l'exécution de la commande pour pouvoir le restaurer en cas de besoin.

Plan

- Patron Mémento
- Retour sur l'architecture MVC

Retour sur l'architecture Model-View-Controller

- Exigences d'une application interactive*
  - La même information peut être présentée différemment aux utilisateurs
  - L'affichage et le comportement de l'application doit immédiatement refléter les manipulations faites sur les données
  - Les changements des interfaces doivent être faciles
    - Changer le « look and feel » ne devrait pas affecter le noyau de l'application

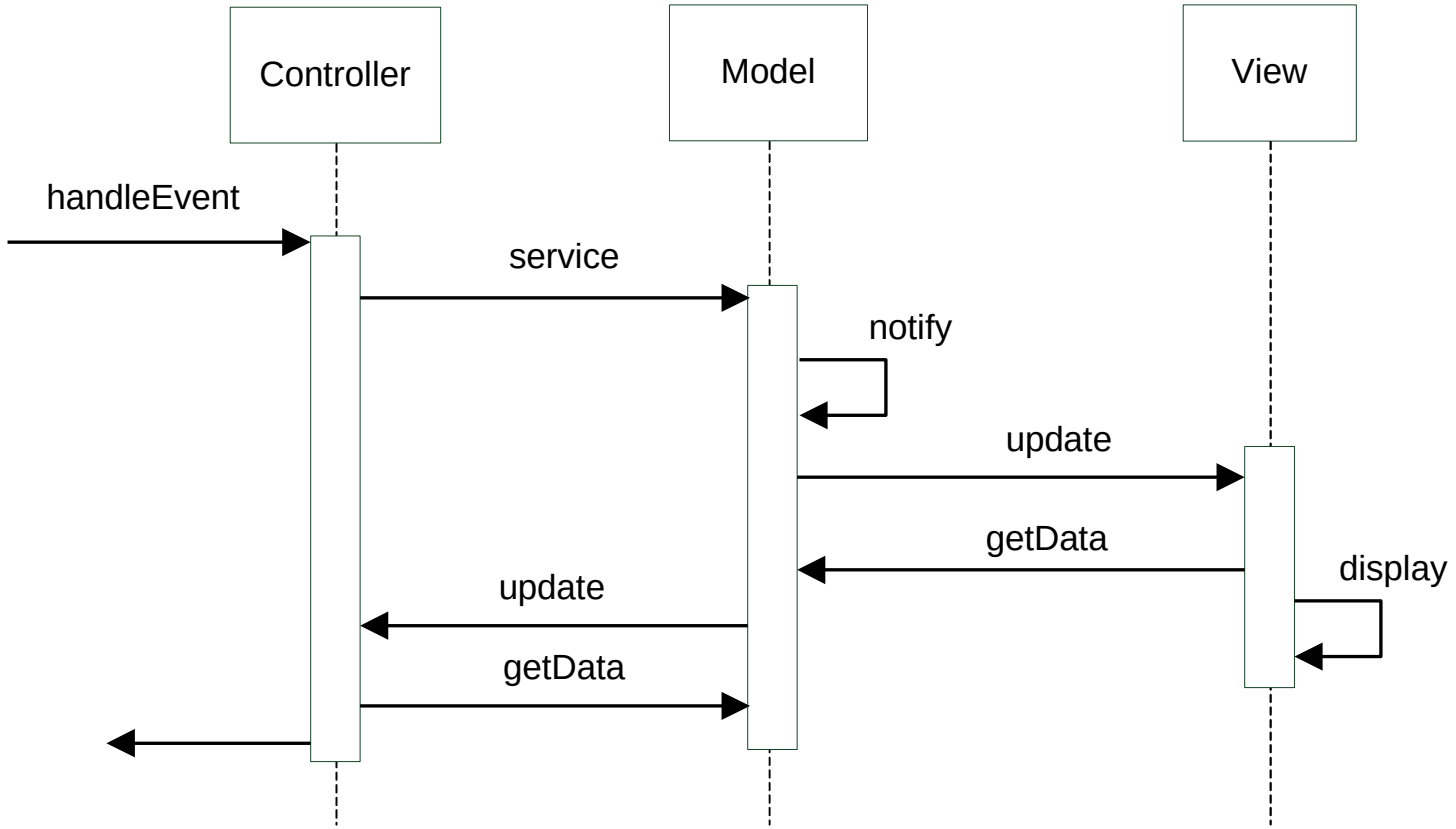
* «pattern-oriented software architecture», Buschman et al., 1996

Retour sur l'architecture Model-View-Controller

- MVC divise une application en trois parties
  - Le Modèle: il encapsule les données et les fonctions noyau de l'application. Il est indépendant des représentations visuelles.
  - Les vues: une vue affiche l'information à l'Utilisateur. Elle lit les données à partir du modèle. Une vue correspond aussi à une interface à travers laquelle l'utilisateur déclenche des actions.
  - Contrôleurs: un contrôleur est associée à chaque vue. Un contrôleur reçoit les entrées sous forme d'événements. Ces événements sont traduits en requêtes sur le modèle ou la vue.

Retour sur l'architecture Model-View-Controller

- Scénario où l'entrée faite par l'utilisateur change l'état du modèle*



* Diagramme adapté de «pattern-oriented software architecture», Buschman et al., 1996

Retour sur l'architecture Model-View-Controller

- Éléments minimaux à retenir sur le fonctionnement de MVC
  - Le modèle doit offrir des méthodes qui permettent aux vues et aux contrôleurs de lire les données.
  - Durant l'initialisation, chaque vue crée un contrôleur. La vue offre, au besoin, les fonctionnalités qui permettent au contrôleur de manipuler l'affichage.
  - Le contrôleur reçoit les entrées de l'utilisateur sous forme d'événements qu'il va traduire en demandes de service au modèle.
  - Lorsque ces demandes de service changent l'état du modèle, toutes les vues reliées à ce modèle doivent se mettre à jour.

Retour sur l'architecture Model-View-Controller

- Exemple simple d'une application MVC* comprenant:

- un modèle comportant comme seule variable qui contient la somme de deux entiers entrés par l'utilisateur.

- une vue qui constitue une simple interface utilisateur

- un contrôleur qui coordonne l'interaction entre le modèle et la vue.

*http://www.newthinktank.com/2013/02/mvc-java-tutorial/

```java
public class CalculatorModel {

  private int calculationValue;

  public void addTwoNumbers(int firstNumber, int secondNumber){
      calculationValue = firstNumber + secondNumber;
  }

  public int getCalculationValue(){
      return calculationValue;
  }
}
```

```java
public class MVCCalculator {
  public static void main(String[] args) {
    CalculatorView theView = new CalculatorView();
    CalculatorModel theModel = new CalculatorModel();
    CalculatorController theController = new CalculatorController(theView,theModel);
    theView.setVisible(true);
  }
}
```

```java
public class CalculatorController {

  private CalculatorView theView;
  private CalculatorModel theModel;

  public CalculatorController(CalculatorView theView, CalculatorModel theModel) {
      this.theView = theView;
      this.theModel = theModel;

     // Tell the View that when ever the calculate button
     // is clicked to execute the actionPerformed method
     // in the CalculateListener inner class

      this.theView.addCalculateListener(new CalculateListener());
  }

  class CalculateListener implements ActionListener{

      public void actionPerformed(ActionEvent e) {
          int firstNumber, secondNumber = 0;

          // Surround interactions with the view with
          // a try block in case numbers weren't
          // properly entered

          try{
              firstNumber = theView.getFirstNumber();
              secondNumber = theView.getSecondNumber()  ;
              theModel.addTwoNumbers(firstNumber, secondNumber);
              theView.setCalcSolution(theModel.getCalculationValue());
          }

          catch(NumberFormatException ex){
              System.out.println(ex);
              theView.displayErrorMessage("You Need to Enter 2 Integers");
          }
      }
  }
}
```

```java
public class CalculatorView extends JFrame{
    private JTextField firstNumber  = new JTextField(10);
    private JLabel additionLabel = new JLabel("+");
    private JTextField secondNumber = new JTextField(10);
    private JButton calculateButton = new JButton("Calculate");
    private JTextField calcSolution = new JTextField(10);

    CalculatorView(){
        JPanel calcPanel = new JPanel();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(600, 200);

        calcPanel.add(firstNumber);
        calcPanel.add(additionLabel);
        calcPanel.add(secondNumber);
        calcPanel.add(calculateButton);
        calcPanel.add(calcSolution);
        calcSolution.setEditable(false);
        this.add(calcPanel);
    }

    public int getFirstNumber(){
        return Integer.parseInt(firstNumber.getText());
    }

    public int getSecondNumber(){
        return Integer.parseInt(secondNumber.getText());
    }

    public int getCalcSolution(){
        return Integer.parseInt(calcSolution.getText());
    }

    public void setCalcSolution(int solution){
        calcSolution.setText(Integer.toString(solution));
    }

    void addCalculateListener(ActionListener listenForCalcButton){
        calculateButton.addActionListener(listenForCalcButton);
    }

    void displayErrorMessage(String errorMessage){
        JOptionPane.showMessageDialog(this, errorMessage);
    }
}
```

Retour sur l'architecture Model-View-Controller

- Quels patrons de conception seraient utiles pour implémenter une application organisée selon l'architecture MVC?

Retour sur l'architecture Model-View-Controller

- L'architecture MVC et les patrons de conception
  - Les vues doivent s'enregistrer auprès du modèle pour qu'il les notifie en cas de changement.
  - Certains contrôleurs ont aussi besoin d'être informés du changement.
    - Patron Observateur

Retour sur l'architecture Model-View-Controller

- L'architecture MVC et les patrons de conception
  - Une vue peut être composée d'autres vues
    - Patron Composite
  - Certains composants graphiques de la vue peuvent nécessiter des comportements additionnels selon leur taille ou forme (ex: scrolling)
    - Patron Décorateur

Retour sur l'architecture Model-View-Controller

- L'architecture MVC et les patrons de conception
  - Une vue contient des composants graphiques (ex. Barre d'outils) pouvant correspondre à différentes actions ou des actions similaires qu'on veut manipuler, défaire ou refaire.
    - Patron Commande
    - Patron Mémento

Retour sur l'architecture Model-View-Controller

- Implémentation de MVC dans Java
  - Le Modèle: un ensemble de classes.
  - Les vues: des interfaces avec des composants graphiques (Swing): boutons, zones de textes, etc.
  - Contrôleurs: Ce sont les listeners (ex: ActionListener)